Orig

IN THE UNITED STATES DISTRICT COURT
FOR THE EASTERN DISTRICT OF TEXAS
MARSHALL DIVISION

RECEIVED CLERK U.S. DISTRICT COURT
03 MAY -5 PM 6:02
EASTERN DISTRICT OF TEXAS

FILED
U. S. DISTRICT COURT
EASTERN DISTRICT OF TEXAS
MAY 5 - 2003
DAVID MALAND, CLERK
By Deputy

TEXAS INSTRUMENTS INCORPORATED,	)	
Plaintiff,	)	
v.	)	Civil Action No. 2-03CV-115
INTERGRAPH CORPORATION and Z/I, IMAGING CORPORATION, Defendants.	)	JURY TRIAL DEMANDED

COMPLAINT FOR INFRINGEMENT OF U.S. PATENT NO. 5,297,279, U.S. PATENT NO. 5,329,471, U.S. PATENT NO. 5,437,027, U.S. PATENT NO. 5,742,538 AND U.S. PATENT NO. 6,065,113

For its Complaint, Plaintiff Texas Instruments Incorporated states as follows:

THE PARTIES

1. Plaintiff Texas Instruments Incorporated ("TI") is a Delaware corporation that maintains its principal place of business in Dallas, Texas.

2. Defendant Intergraph Corporation ("Intergraph") is a Delaware corporation that maintains its principal place of business at 288 Dunlop Boulevard, Huntsville, Alabama, 35824 and may be served through its registered agent for service, Prentice-Hall Corporation, 800 Brazos, Austin, Texas 78701. Intergraph manufactures and sells software products and hardware systems that are used in a variety of industries, including in the State of Texas and in this District.

3. Defendant Z/I Imaging Corporation ("Z/I Imaging") is a Delaware corporation that maintains it principle place of business at 301 Cochrane Road, Suite 9, Huntsville, Alabama, 35824 and may be served through its registered agent for service, Corporation Service Company d/b/a CSC Lawyers Incorporating Service Co., 800 Brazos, Austin, Texas 78701. Z/I Imaging

manufactures and sells hardware systems that are used in a variety of industries, including in the State of Texas and in this District.

JURISDICTION AND VENUE

4. This action arises under the patent laws of the United States, Title 35, United States Code. The jurisdiction of this Court is proper under 35 U.S.C. §§ 271, et seq., and 28 U.S.C. §§ 1331 and 1338.

5. Personal jurisdiction exists generally over Intergraph because it has minimum contacts with this forum as a result of business regularly conducted within the State of Texas and within this district and specifically as a result of, at least, Intergraph's distribution network wherein Intergraph places its products and services that infringe TI's patents within the stream of commerce, wherein said stream is directed at this district as well as Texas, and by committing and/or inducing the tort of patent infringement within Texas and this district.

6. Personal jurisdiction exists generally over Z/I Imaging because it has minimum contacts with this forum as a result of business regularly conducted within the State of Texas and within this district and specifically as a result of, at least, Z/I Imaging's distribution network wherein Z/I Imaging places its products and services that infringe TI's patents within the stream of commerce, wherein said stream is directed at this district as well as Texas, and by committing and/or inducing the tort of patent infringement within Texas and this district.

7. Venue is proper in this Court under 28 U.S.C. §§ 1391(c) and 28 U.S.C. § 1400(b).

PATENTS AT ISSUE

8. On March 22, 1994, United States Patent No. 5,297,279 ("the '279 patent"), was duly and legally issued to TI, with Thomas J. Bannon, Stephen J. Ford, Vappala J. Joseph,

Edward R. Perez, Robert W. Peterson, Diana M. Sparacin, Satish M. Thatte, Craig W. Thompson, Chung C. Wang and David L. Wells named as inventors, for inventions relating to a system and method for database management supporting object-oriented programming languages. Since March 22, 1994, TI has been and still is the owner of the '279 patent and continues to hold all rights in said patent. A copy of the '279 patent is attached for the Court's convenience as Exhibit A.

9. On July 12, 1994, United States Patent No. 5,329,471 ("the '471 patent"), was duly and legally issued to TI, with Gary Swoboda, Martin D. Daniels and Joseph A. Coomes named as inventors, for inventions relating to testing and emulation of logic circuitry. Since July 12, 1994, TI has been and still is the owner of the '471 patent and continues to hold all rights in said patent. A Certificate of Correction (Certificate B1 5,329,471) for '471 patent duly and legally issued on April 11, 1995. A copy of the '471 patent and said Certificate of Correction is attached for the Court's convenience as Exhibit B.

10. On July 25, 1995, United States Patent No. 5,437,027 ("the '027 patent"), was duly and legally issued to TI, with Thomas J. Bannon, Stephen J. Ford, Vappala J. Joseph, Edward R. Perez, Robert W. Peterson, Diana M. Sparacin, Satish M. Thatte, Craig W. Thompson, Chung C. Wang and David L. Wells named as inventors, for inventions relating to a system and method for database management supporting object-oriented programming languages. The '027 patent is related to the '279 patent discussed above. The '027 patent issued as a result of divisional election made during the prosecution of the application that matured into the '279 patent. Since July 25, 1995, TI has been and still is the owner of the '027 Patent and continues to hold all rights in said patent. The copy of the '027 patent is attached for the Court's convenience as Exhibit C.

11. On April 21, 1998, United States Patent No. 5,742,538 ("the '538 patent"), was duly and legally issued to TI, with Karl M. Guttag, Christopher J. Read and Keith Balmer named as inventors, for inventions relating to the operation of the multiplier and arithmetic logic unit of data processing circuitry. Since April 21, 1998, TI has been and still is the owner of the '538 patent and continues to hold all rights in said patent. A copy of the '538 patent is attached for the Court's convenience as Exhibit D.

12. On May 16, 2000, United States Patent No. 6,065,113 ("the '113 patent"), was duly and legally issued to TI, with Jonathan H. Sheill, Joel J. Graber and Donald E. Steiss named as inventors, for inventions relating to a system and method for the identification of a microprocessor at an instruction set level. Since May 16, 2000, TI has been and still is the owner of the '113 patent and continues to hold all rights in said patent. A copy of the '113 patent is attached for the Court's convenience as Exhibit E.

COUNT I

PATENT INFRINGEMENT

13. TI incorporates by reference and incorporates herein the allegations of paragraphs 1-12 above.

14. Defendant Intergraph has been and is now directly infringing, and/or indirectly infringing by way of inducing infringement and/or contributing to the infringement of, the '279, '471, '027, '538 and '113 patents in this District and elsewhere by making, using, offering for sale, and selling software products and hardware systems covered by at least one claim of each of the '279, '471, '027, '538 and '113 patents, all to the injury of TI.

15. Defendant Z/I Imaging has been and is now directly infringing, and indirectly infringing by way of inducing infringement and/or contributing to the infringement of, the '471, '538 and '113 patents in this District and elsewhere by making, using, offering for sale, and

selling hardware systems covered by at least one claim of each of the '471, '538 and '113 patents, all to the injury of TI.

16. Defendant Intergraph will continue to infringe the '279, '471, '027, '538 and '113 patents unless enjoined by this Court.

17. Defendant Z/I Imaging will continue to infringe the '471, '538 and '113 patents unless enjoined by this Court.

18. TI has suffered damages as a result of Defendant Intergraph's infringement of the '279, '471, '027, '538 and '113 patents and will continue to suffer damages in the future unless permanently enjoined from infringing the same.

19. TI has suffered damages as a result of Defendant Z/I Imaging's infringement of the '471, '538 and '113 patents and will continue to suffer damages in the future unless permanently enjoined from infringing the same.

PRAYER FOR RELIEF

TI respectfully requests the following relief:

A. A judgment that Defendant Intergraph has infringed the '279, '471, '027 '538 and '113 patents directly and indirectly by way of inducing infringement and/or contributing to the infringement of the '279, '471, '027 '538 and '113 patents.

B. A judgment that Defendant Z/I Imaging has infringed the '471, 538 and '113 patents directly and indirectly by way of inducing infringement and/or contributing to the infringement of the '471, 538 and '113 patents.

C. An injunction preventing Defendant Intergraph from infringing, inducing the infringement of, or contributing to the infringement of the '279, '471, '027 '538 and '113 patents and an injunction preventing Defendant Z/I Imaging from infringing, inducing the infringement

of, or contributing to the infringement of the '471, 538 and '113 patents;

D. A judgment and order requiring Defendants Intergraph and Z/I Imaging to pay TI monetary damages under 35 U.S.C. § 284 with interest.

E. A judgment and order that the Court finds this an exceptional case and requires Defendants Intergraph and Z/I Imaging to pay TI's costs (including all disbursements) and attorney's fees incurred in this action, as is provided for in 35 U.S.C. § 285; and

F. Such other and further relief as the Court deems just and proper.

DEMAND FOR JURY TRIAL

TI hereby demands that all issues so eligible be determined by a jury.

Respectfully submitted,

Carl Roth by perm MCS

Carl R. Roth
State Bar No. 17312000
THE ROTH LAW FIRM
115 N. Wellington, Suite 200
P.O. Box 876
Marshall, Texas 75761-1249
Telephone: 903-935-1665
Facsimile: 903-935-1797
Email: cr@rothfirm.com

Kenneth R. Adamo
State Bar No. 10496700
Thomas R. Jackson
State Bar No. 10496700
Michael Newton
State Bar No. 24003844
JONES DAY
2727 N. Harwood Street
Dallas, Texas 75201
Telephone: 214-220-3939
Facsimile: 214-969-5100
Email: kradamo@jonesday.com
trjackson@jonesday.com

ATTORNEYS FOR TEXAS INSTRUMENTS INCORPORATED

Of Counsel:

Jay C. Johnson
Richard R. Andrews
TEXAS INSTRUMENTS INCORPORATED
P.O. Box 655474
Mail Station 3999
Dallas, Texas 75265
Telephone: 972-917-5557
Facsimile: 972-917-4418

EXHIBIT A

US005297279A

United States Patent [19]

Bannon et al.

[11] Patent Number: 5,297,279

[45] Date of Patent: Mar. 22, 1994

[54] SYSTEM AND METHOD FOR DATABASE MANAGEMENT SUPPORTING OBJECT-ORIENTED PROGRAMMING

[75] Inventors: **Thomas J. Bannon**, Dallas; **Stephen J. Ford**; **Vappala J. Joseph**, both of Plano; **Edward R. Perez**, Dallas; **Robert W. Peterson**; **Diana M. Sparacin**, both of Plano; **Satish M. Thatte**, Richardson; **Craig W. Thompson**, Plano; **Chung C. Wang**; **David L. Wells**, both of Dallas, all of Tex.

[73] Assignee: **Texas Instruments Incorporated**, Dallas, Tex.

[21] Appl. No.: **531,493**

[22] Filed: **May 30, 1990**

[51] **Int. Cl.**[5] **G06F 3/00**; G06F 15/40

[52] **U.S. Cl.** **395/600**; 395/500; 364/DIG. 1; 364/282.1; 364/283.1; 364/283.4

[58] **Field of Search** ..... 364/DIG. 1, DIG. 2; 395/425, 600, 650, 700, 725, 500

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,525,780	6/1985	Bratt et al.	395/650
4,853,842	8/1989	Thatte et al.	395/425
4,989,132	1/1991	Mellender et al.	395/425
5,075,842	12/1991	Lai	395/425
5,075,845	12/1991	Lai et al.	395/425
5,075,848	12/1991	Lai et al.	395/425
5,079,695	1/1992	Dysart et al.	395/700

OTHER PUBLICATIONS

Stephen Ford, et al., 'Zeitgeist: Database Support for Object-Oriented Programming'; Advances in Object-Oriented Database Systems, 27 Sep. 1988, Ebernburg, Germany, pp. 23–42.

R. Agrawal, et al., 'ODE (Object Database and Environment): The Language and The Data Model'; SIGMOD RECORD, vol. 18, No. 2, 31 May 1989, Portland, Oreg., USA, pp. 36–45.

A Straw, et al., 'Object Management in a Persistent Smalltalk System', Software Practice & Experience, vol. 19, No. 8, Aug. 1989, CHICHESTER GB, pp. 719–737.

Peter Lyngbaek, et al. "A Data Modeling Methodology for the Design and Implementation of Information Systems", Int'l Workshop on Object-Oriented Databases Systems, 1986,p. 6+.

Kevin Wilkinson, et al. "The IRIS Architecture and Implementation", IEEE Trans. on Knowledge and Data Engineering, V2N1 Mar. 1990, pp. 63+.

David Maier, et al. "Development of an Object-Oriented DBMS", Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1986. pp. 472+.

Timothy Andrews, et al. "Combining Language and Database Advances in an Object-Oriented Development Environment", Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1987. pp. 430+.

Won Kim, et al. "Integrating an Object-Oriented Programming System with a Database System", Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1988. pp. 142+.

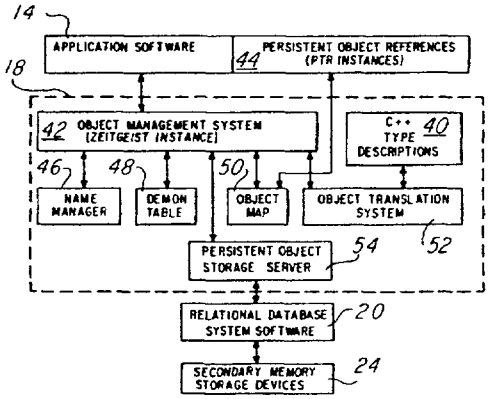
(List continued on next page.)

Primary Examiner—Paul V. Kulik
Assistant Examiner—John C. Loomis
Attorney, Agent, or Firm—Richard A. Stoltz; Richard L. Donaldson; William E. Hiller

[57] **ABSTRACT**

A system and method for database management for providing support for long-term storage and retrieval of objects created by application programs written at least in part in object-oriented programming languages consists of a plurality of software modules. These modules provide data definition language translation, object management, object translation, and persistent object storage service. Such system implements an object fault capability to reduce the number of interactions between the application, the database management system, and the database.

4 Claims, 5 Drawing Sheets



OTHER PUBLICATIONS

Won Kim, et al. "Architecture of the ORION Next-Generation Database Sytem", IEEE Trans. on Knowledge and Data Engineering, V2N1 Mar. 1990. pp. 109+.

Michael Stonebraker, "Object Management in POSTGRES Using Procedures", Int'l Workshop on Object-Oriented Database (OODB) Systems 1986. pp. 66+.

Michael Stonebraker, et al. "The Implementation of POSTGRES", IEEE Trans. on Knowledge and Data Engineering, V2N1, Mar. 1990. pp. 125+.

Puknraj Kachhwaha, et al. "An Object-Oriented Data Model for the Research Laboratory", Int'l Workshop on Object-Oriented Database (ODOB) Systems 1986. p. 218.

Puknraj Kachhwaha, "LCE: An Object-Oriented Database Application Development Tool", SIGMOD Int'l Conference on Management of Data 1988. p. 207.

Laura M. Haas, et al. "Starburst Mid-Flight: As the Dust Clears", IEEE Trans. on Knowledge and Data Engineering, V2N1 Mar. 1990. [IBM's Starburst] pp. 143+.

Ted Kaehler, "Virtual Memory on a Narrow Machine for an Object-Oriented Language", Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1986. [Xerox PARC's LOOM]pp. 87+.

Karen E. Smith, et al. "Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems", Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1987. [Brown's Intermedia System] pp. 452+.

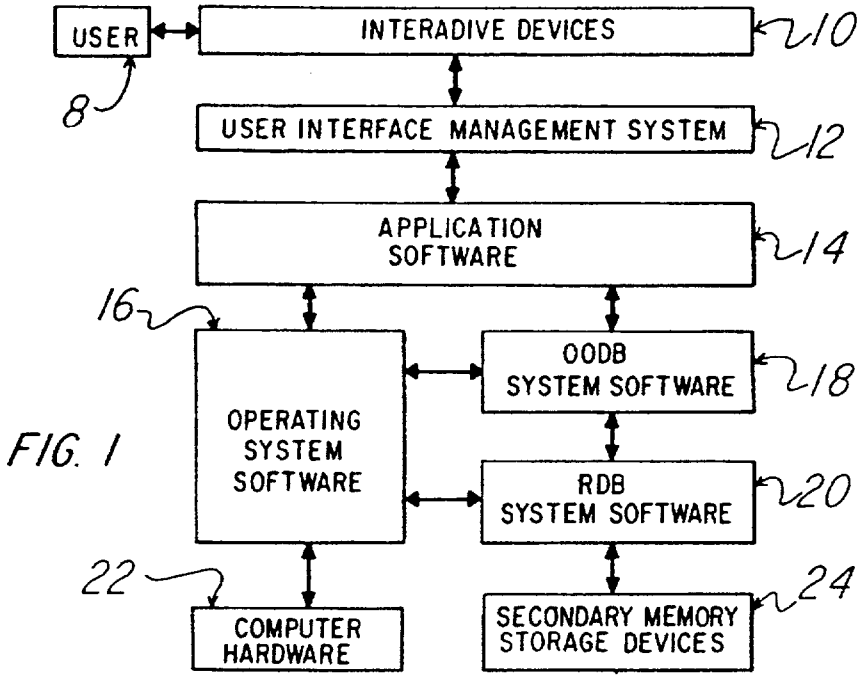

USER
8
INTERADIVE DEVICES
10
USER INTERFACE MANAGEMENT SYSTEM
12
APPLICATION SOFTWARE
14
16
OPERATING SYSTEM SOFTWARE
OODB SYSTEM SOFTWARE
18
RDB SYSTEM SOFTWARE
20
22
COMPUTER HARDWARE
SECONDARY MEMORY STORAGE DEVICES
24

FIG. 1

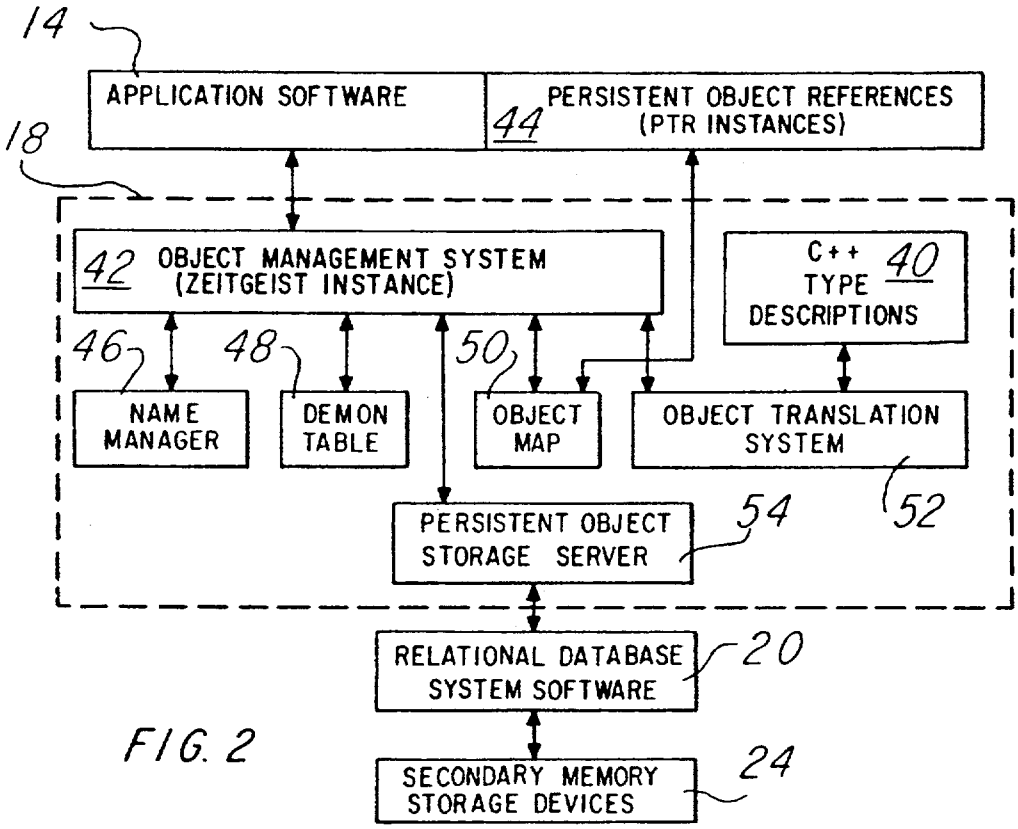

14
APPLICATION SOFTWARE
PERSISTENT OBJECT REFERENCES (PTR INSTANCES)
44
18
42
OBJECT MANAGEMENT SYSTEM (ZEITGEIST INSTANCE)
C++ TYPE DESCRIPTIONS
40
46
48
50
NAME MANAGER
DEMON TABLE
OBJECT MAP
OBJECT TRANSLATION SYSTEM
52
PERSISTENT OBJECT STORAGE SERVER
54
RELATIONAL DATABASE SYSTEM SOFTWARE
20
SECONDARY MEMORY STORAGE DEVICES
24

FIG. 2

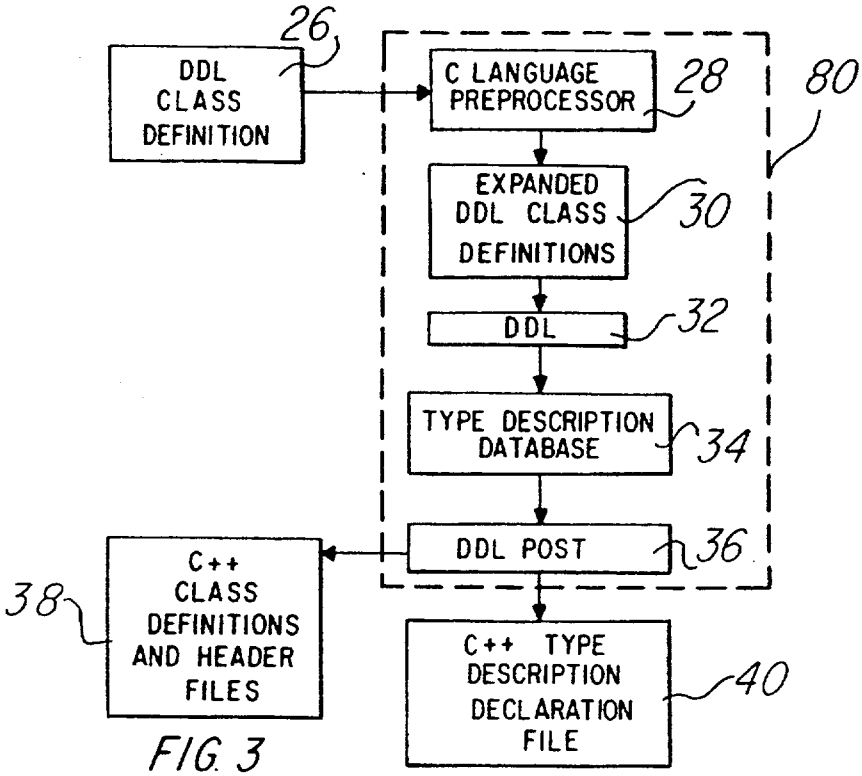

26
DDL CLASS DEFINITION
C LANGUAGE PREPROCESSOR
28
80
EXPANDED DDL CLASS DEFINITIONS
30
DDL
32
TYPE DESCRIPTION DATABASE
34
DDL POST
36
38
C++ CLASS DEFINITIONS AND HEADER FILES
C++ TYPE DESCRIPTION DECLARATION FILE
40

FIG. 3

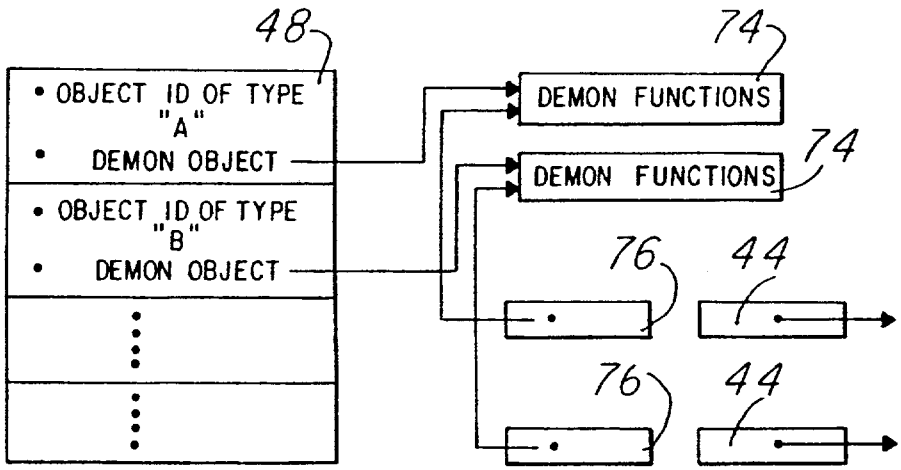

48
• OBJECT ID OF TYPE "A"
• DEMON OBJECT
• OBJECT ID OF TYPE "B"
• DEMON OBJECT
74
DEMON FUNCTIONS
74
DEMON FUNCTIONS
76
44
76
44

FIG. 4

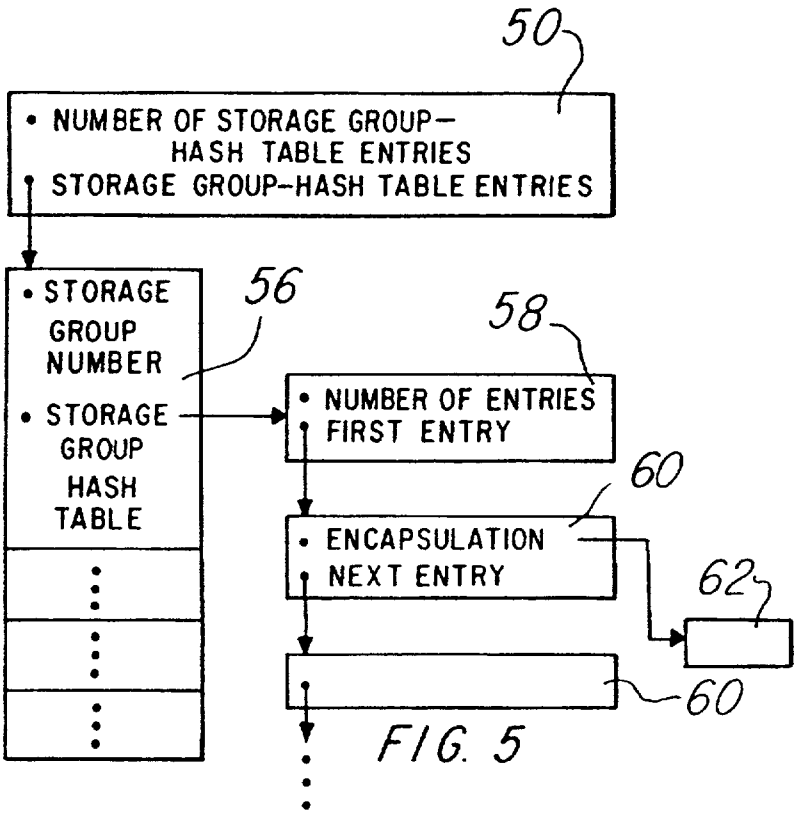

50
• NUMBER OF STORAGE GROUP-HASH TABLE ENTRIES
• STORAGE GROUP-HASH TABLE ENTRIES
• STORAGE GROUP NUMBER
• STORAGE GROUP HASH TABLE
56
58
• NUMBER OF ENTRIES
• FIRST ENTRY
60
• ENCAPSULATION
• NEXT ENTRY
62
60

FIG. 5

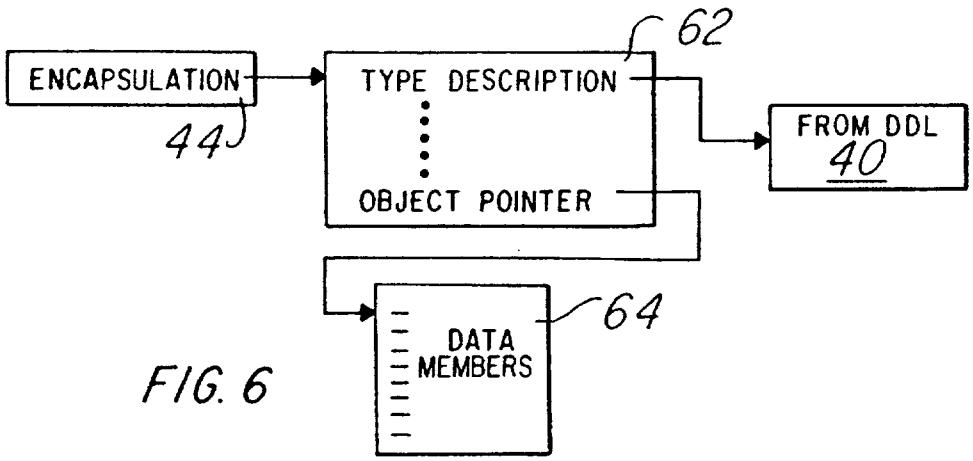

ENCAPSULATION
44
62
TYPE DESCRIPTION
OBJECT POINTER
FROM DDL
40
64
DATA MEMBERS

FIG. 6

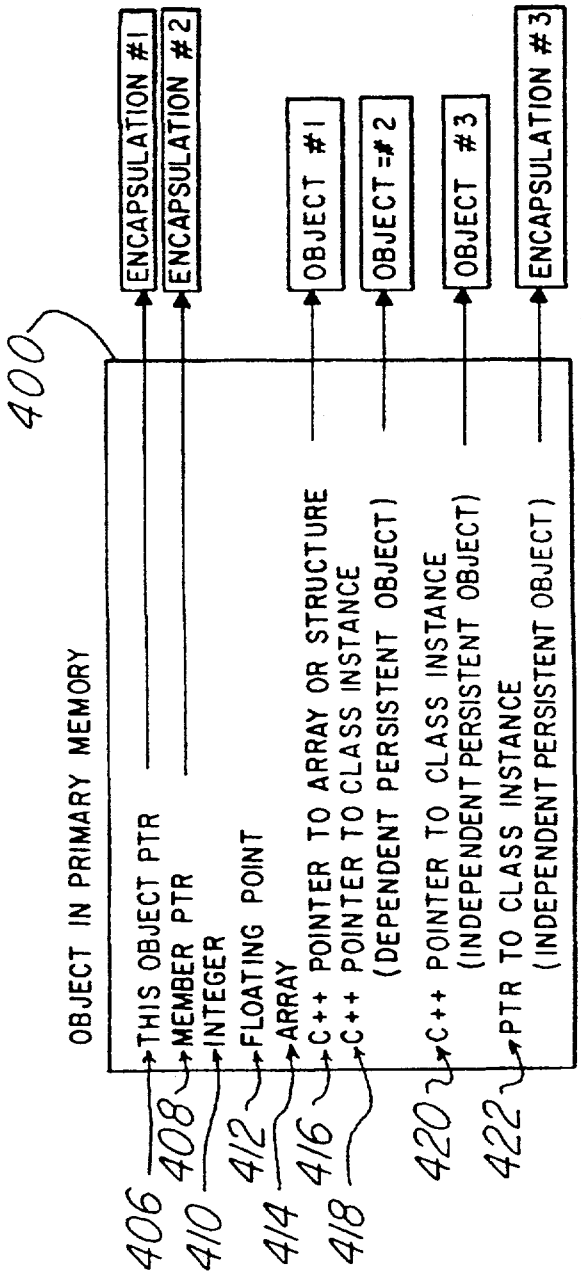

400
OBJECT IN PRIMARY MEMORY
THIS OBJECT PTR
MEMBER PTR
INTEGER
FLOATING POINT
ARRAY
C++ POINTER TO ARRAY OR STRUCTURE
C++ POINTER TO CLASS INSTANCE
(DEPENDENT PERSISTENT OBJECT)
C++ POINTER TO CLASS INSTANCE
(INDEPENDENT PERSISTENT OBJECT)
PTR TO CLASS INSTANCE
(INDEPENDENT PERSISTENT OBJECT)
406
408
410
412
414
416
418
420
422
ENCAPSULATION #1
ENCAPSULATION #2
OBJECT #1
OBJECT #2
OBJECT #3
ENCAPSULATION #3

FIG. 7a

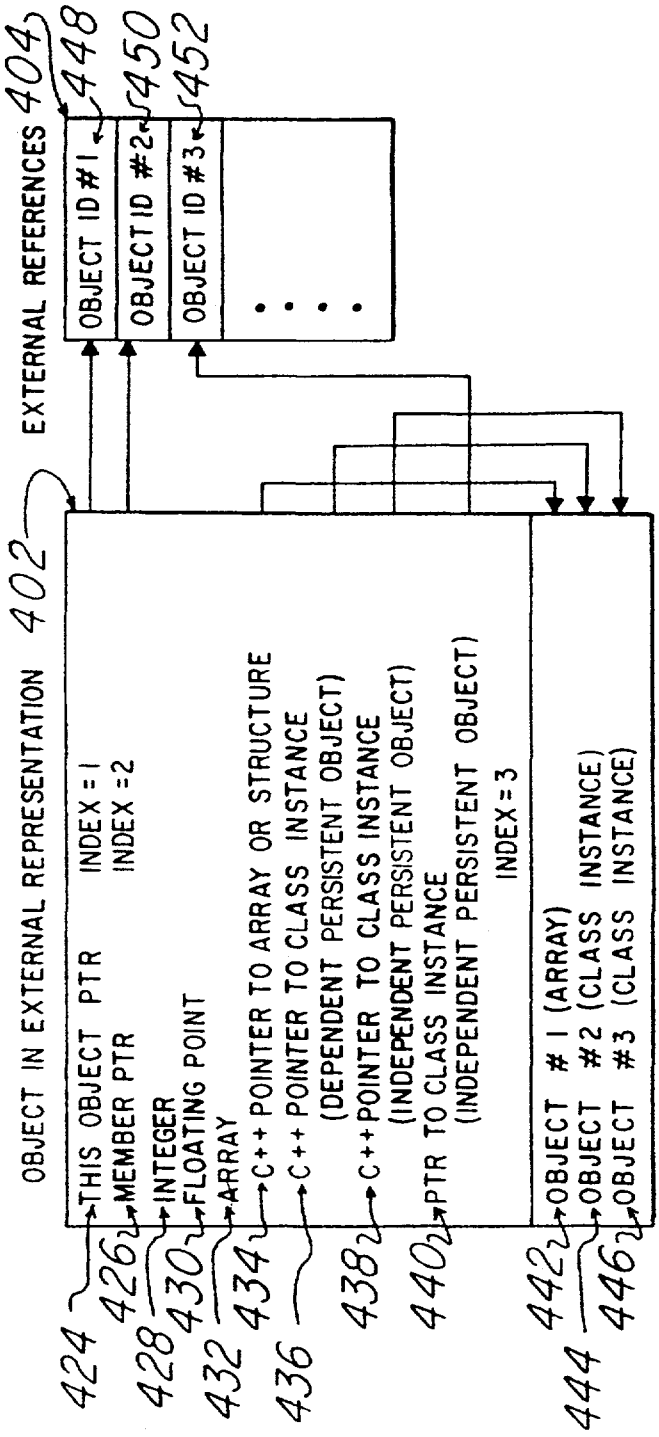

EXTERNAL REFERENCES 404
OBJECT ID #1
OBJECT ID #2
OBJECT ID #3
448
450
452
OBJECT IN EXTERNAL REPRESENTATION 402
THIS OBJECT PTR
INDEX = 1
MEMBER PTR
INDEX = 2
INTEGER
FLOATING POINT
ARRAY
C++ POINTER TO ARRAY OR STRUCTURE
C++ POINTER TO CLASS INSTANCE
(DEPENDENT PERSISTENT OBJECT)
C++ POINTER TO CLASS INSTANCE
(INDEPENDENT PERSISTENT OBJECT)
PTR TO CLASS INSTANCE
(INDEPENDENT PERSISTENT OBJECT)
INDEX = 3
OBJECT # 1 (ARRAY)
OBJECT #2 (CLASS INSTANCE)
OBJECT #3 (CLASS INSTANCE)
424
426
428
430
432
434
436
438
440
442
444
446

FIG. 7b

SYSTEM AND METHOD FOR DATABASE MANAGEMENT SUPPORTING OBJECT-ORIENTED PROGRAMMING

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to database management systems and more particularly to a system and method providing support for long-term storage and retrieval of objects created by application programs written in object-oriented programming languages.

2. Description of Related Art

Many new computer software applications, such as Computer-Aided Design and Manufacturing, Computer-Aided Software Engineering, multimedia and hypermedia information systems, and Artificial Intelligence Expert systems, have data models that are much more complex than previous system, both in content and interobject relationships. Object-oriented languages provide the application developer the mechanism to create and manipulate the data models inherent in these applications. Database systems provide long term storage of the data created by these applications. However, existing languages and databases are insufficient to develop these applications because existing object-oriented languages do not provide direct support for long-term storage and sharing of objects, existing commercial database systems (hierarchical, network, and relational) do not support the necessary complex object-oriented data models, and existing database systems require an application developer to use different languages and modeling paradigms when building applications.

There have been various research and commercial efforts aimed at developing OODBs. These OODBs vary in type of data model employed, application program interaction, object access method, method of persistent object store, etc. Examples of these current OODBs, and their weaknesses, will now be considered.

Iris (Hewlett Packard) and GemStone (Servio Corporation) are representative OODBs employing new proprietary object-oriented data models, while Vbase (Ontologic), Orion (Microelectronics and Computer Technology Corporation), and DOME (Dome Software Corporation) are examples of OODBs incorporating proprietary extensions to existing programming language data models. In these types of OODBs, application developers are required to learn a new proprietary data model in order to effectively use the OODB. Since their data model is new, using it often results in a loss of productivity as application developers learn the new language. In Orion, instances of user-defined classes cannot be stored in the database unless they have been derived from Orion-defined classes. In addition, GemStone and Orion do not allow for an instance of their classes to be transient; that is, every object created in a GemStone- or Orion-based application will be stored in the database unless it is specifically deleted. Another problem with developing a new data model is that it requires application developers to rely on a single source of application development tools, such as language compilers, object libraries, and program debuggers, which limits widespread acceptance of these OODBs.

POSTGRES (University of Berkeley) is an example of an OODB employing another type of data model, that of a proprietary extension to an existing relational

database. POSTGRES is a combination of an extended relational and object-oriented database. Objects are created using relational table descriptions, while functions to manipulate the objects are created using the POSTQUEL query language as well as conventional languages (C and LISP). In addition, if the application developer wishes to add indices over user-defined types, they must write and register with POSTGRES functions to perform the various comparison operations between two objects of the same user defined type. Since the latter mode of creating functions requires the application developers to map between the POSTGRES and C/LISP data models, which can be error prone and distracting from the task of developing the application system, this strategy does nothing to alleviate the burdensome requirement to use different languages and modeling paradigms when building applications. This problem of mapping between the object-oriented and relational data models was discussed in-depth in the Intermedia OOPSLA '87 conference paper.

Ontos (Ontologic) and Object Store (Object Design) are representative of OODBs employing the last type of data model, namely the use of an existing programming language data model (e.g., using the C++ programming language data model for writing software programs and interacting with the database). Both systems, however, require the use of a proprietary language compiler to add additional code (Ontos) or translate new and non-standard C++ language constructs (Object Store). As with the first two types of OODBs, this approach requires application developers to rely on a single source of application development tools, which also limits widespread acceptance of these OODBs.

In addition to problems inherent with the type of data models selected, difficulties occur when an application program interacts with an OODB. In the Iris OODB, application developers define object types and develop functions to manipulate the objects using the proprietary Iris language. Iris provides an interactive interface where requests can be made to retrieve or manipulate Iris objects. The requests are evaluated by performing relational queries (since the objects are stored in relational tables) and the result is returned as an Iris expression, not as object values or references. Iris provides an embedded object SQL interface, a C language interface (which is not object-oriented), and allows the application developer to register foreign functions written in existing (possibly non-object-oriented) programming languages. These approaches require the application developer to map the Iris objects into data structures accessible by the programming language, reintroducing the problems discussed above.

Similarly, the developers of the GemStone OODB also defined a new language, OPAL, which the application developer uses to define object types and functions to manipulate the objects. GemStone provides an interactive development environment for developing OPAL objects and functions. GemStone also provides a mechanism for existing programming languages (C and Smalltalk) to interact with GemStone. However, unless the applications developer uses only the OPAL language, two data models and languages must be used to interact with the database, mapping the OPAL objects into structures accessible by the programming language, and thereby resulting in the problems associated discussed previously.

The Vbase OODB requires two separate languages, TDL to define object types, and COP (an extension to the C programming language) to develop application programs. Although application developers do not need to map objects between the data model and the programming language, they must still use two languages during the development of their programs, with the attendant problems considered above. A further restriction of this system includes the failure to provide access to the database from other programming languages.

Although the Orion OODB developers used an existing programming language, Common Lisp, for their data model, they developed several proprietary extensions to the language. As with Vbase, there is no need to map between the data model and programming language with the Orion OODB. However, this approach requires the use of a proprietary language translator.

The developers of POSTGRES, on the other hand, expect most application developers to write programs that interact with the database primarily using the POSTGRES query language, POSTQUEL. Navigation between objects is possible; however, a query must be issued to perform the navigation instead of accessing the referenced object directly. Application developers can define and implement their own functions including programming language statements, POSTQUEL query statements, and/or calls to POSTGRES' internal functions. Thus, application developers may have to deal with two or more data models to build their application systems. Such requirement fails to alleviate the problems considered above.

The Ontos approach provides an interface from the C++ language to the database. However, the amount of interaction between the program and Ontos is much higher than is reasonable or necessary due to the requirement of specialized functions that must be provided by the application developer (e.g., object construction, translation, storage/retrieval, etc.). This burdens the application developer with more work that could have been performed by the database system. Object Store also provides an interface from the C++ language to the database. However, the interface is accomplished by redefining the semantics of or adding new C++ language constructs, thereby requiring the use of Object Design's proprietary C++ language translator, which limits widespread acceptance of their system.

Access to an object in an OODB is performed by manipulating the object using predefined functions, using an explicit query, or by coding explicit references in a programming language.

In the Iris OODB, application developers call functions to retrieve or change values in the object. A program cannot receive a reference to an object which could be passed to other functions. In the GemStone, Vbase, and Orion OODBs, individual objects can be accessed and passed to functions to retrieve or assign values.

In the POSTGRES database, application developers perform queries to retrieve or change values in the object (actually, relational tuples). POSTGRES allows a foreign function to access an object, but as stated above, it must be mapped from the relational data model to the data model of the foreign function's programming language.

Although most OODBs allow the application developer to explicitly retrieve an object from the database (Iris and POSTGRES do not), they do not allow the application developer to specify when objects related to the original object should be retrieved. For example, application developers can access objects in Ontos using one of two modes. In the first mode, an object is explicitly retrieved and referenced objects are implicitly retrieved using an object fault capability. In the other mode, one or more related objects can be explicitly retrieved, but the application must continually check to see if a referenced object is already in memory, and then explicitly retrieve it if is not. This requires the application developer to employ two completely different models of accessing persistent objects in the same program, which can easily cause errors in the program by the inadvertent and natural use of one mode where the other mode should have been used.

The approach taken by Object Store is quite different from the above OODBs with regard to object access. Object Store's model is more like a persistent memory (an extension of virtual memory computer operating system) than an OODB. Object Design chose to completely reimplement the virtual memory management functions of the C++ programming language and the UNIX (TM) operating system. Whenever a persistent object is created or retrieved from the database, it is installed in a portion of primary memory controlled by Object Design. Thus, references to the object are, in essence, monitored by Object Design's software. If the object is not currently in primary memory, it will be retrieved from the database and installed in primary memory. This style of memory management requires that any class or class library requiring persistence must be written using this memory management scheme, or perform no dynamic memory management thereby resulting in one version of the library for persistent usage and one version for transient usage. Although this approach improves the object storage and retrieval performance, it is inherently dependent on the underlying computer operating system and memory architecture, and thus not portable to other computer systems.

Therefore, these approaches either limit how an application program can access an object, or require additional work in order for the program to access an object.

Most OODBs (except for Iris and DOME) have developed their persistent object storage facility utilizing an existing file management system. They had to develop new implementations of the disk storage structures and management, concurrency control, transaction management, communication, and storage management subsystems. This approach increases the complexity of the overall database system software.

The Iris and DOME OODBs, on the other hand, use existing commercial Relational Database Management Systems (RDBMS) to store their objects. Although the Iris OODB uses Hewlett Packard's relational database HP-SQL, it does not use the SQL interface to that database, restricting access to the objects to the available Iris functions, Iris interactive browser, C language interface, and embedded Iris SQL. Although Iris allows the application developer to define how objects are to be stored, the use of Hewlett Packard's RDBMS imposes a limit on the size of an object. The DOME OODB, which uses Oracle Corporation's Oracle RDBMS, and the POSTGRES system, which has its own relational storage system, decomposes objects into one or more entries in one or more relational tables. This approach requires a relational join whenever more than one attribute value from an object is retrieved.

Relational join operations are computationally expensive.

In the GemStone and Object Store OODBs, the unit of concurrency control is not an object but a secondary memory segment, or page. This approach can improve the performance of secondary memory reads and writes, but results in having the storage facility read, write, and lock more data than may be necessary. In addition, this restricts the amount of concurrent access to objects since the OODB system, and not the application developer, chooses the unit of concurrency control.

Most of the OODBs allow related objects to be clustered together in the persistent object storage. GemStone and Orion only allow clustering controls to be specified when the entire database is defined. Vbase and Ontos allow runtime specification of clustering controls to store one persistent object as close as possible to another persistent object. Object Store also allows runtime specification of clustering controls to store statically allocated objects in a specific database and dynamically allocated objects in a specific database or as close as possible to another persistent object. This requires the application developer to treat similar objects with different models of clustering, which can cause errors in the program by the inadvertent use of one mode where the other mode should have been used. These systems indicate that such clustering specifications are purely hints which the system may ignore. These clustering hints may require rebuilding of the database if they are changed, thereby restricting the ability of the application developers to tune the database's performance by altering the physical grouping of objects. Furthermore, the systems based on relational storage, such as Iris, POSTGRES, and DOME, do not allow user-defined clustering of objects.

SUMMARY OF THE INVENTION

In view of the above problems associated with the related art, it is an object of the present invention to provide a database management system and method which supports long term storage and retrieval of objects created by application programs, and which uses existing object-oriented programming languages to thereby enable such system and method to be ported to other computer platforms without requiring any modifications to existing language translators or computer operating systems and thereby not unduly restrict application developers in their choice of computer platform or language translator.

It is a further object of the present invention to provide a database management system and method providing a standard object-oriented programming interface for its database functionality, thereby eliminating any requirement for mixing of object-oriented and functional, or other, programming styles to confuse the application developer when coding a program's interface to that of the present invention.

It is yet another object of the present invention to provide a database management system and method for adding persistence to existing language objects orthogonally, thereby allowing application programmers to treat persistent and nonpersistent objects in nearly the same manner and eliminating the need to use two or more data models when building application systems.

Another object of the present invention is to provide a database management system and method that allows the application developer to specify at object definition time how related objects, whether created dynamically or statically, should be clustered when stored, to thereby provide a capability to adjust the size of storage objects to enhance the overall system performance.

Still another object of the present invention is to provide a database management system and method that reduces the number of interactions with the database management system that an application developer must code to access objects stored in the database.

It is a further object of the present invention to provide a database management system and method that allows the application developer to specify at application execution time prior to saving a persistent object whether or not to install in primary memory the persistent objects referenced from the given persistent object at the same time when the given object is later installed in primary memory, either due to explicit or implicit retrieval, to enhance the overall system performance.

A further object of the present invention is to provide maximization of concurrent usage of the objects in the database by making the unit of locking the individual persistent object instead of a page of persistent objects.

It is still another object of the present invention to store objects in a persistent object storage server utilizing a relational database management system by storing an external representation of the object and external references from the object without decomposing the objects into multiple relational tuples, to enhance the overall system performance.

yet another object of the present invention is to provide a database management system and method which uses a uniform object translation methodology thereby eliminating the need for application developers to perform this complex computer-and language-dependent task.

In accordance with the above objects of the invention, the preferred embodiment of the present invention consists of four software modules to provide database services to application developers. They are referred to as the Data Definition Language (DDL) translator, the Object Management System (OMS), the Object Translation System (OTS), and the Persistent Object Storage Server (POS Server).

The present invention presents an application interface for programming languages which does not require any extensions to the languages, modifications to existing language translators, or development of proprietary language translators. Furthermore, the present invention implements an object fault capability which reduces the number of interactions that an application must perform with the database management system and database itself. Access of, and navigation between, objects can be performed using existing language operations in a transparent manner.

Furthermore, instead of requiring the application developers to use one data model to interact with the database and another data model to manipulate the objects in a programming language, the present invention uses the data model of existing standard object-oriented languages, such as C_{++} and CLOS, as the data model for the database. This alleviates problems associated with the art discussed above.

Although the present invention can be implemented in any object-oriented programming language, and should therefore not be limited in any way to any specific language, it has been implemented in both C_{++} and Common Lisp. In the C_{++} embodiment, application developers interact solely with the DDL module,

in a batch processing mode, and with the OMS module using standard C++ syntax in their application programs. The DDL module accepts object type descriptions on standard C++ programming language statements (with a few additional syntactic constructs) and extracts sufficient information from the descriptions to enable the OTS module to translate objects between their primary and secondary memory representations. This process is required because this type description information is not available in the C++ run-time system. To achieve architecture-independent translation, the DDL translator also accepts information describing a specific computer architecture and software system environment in which the present invention's applications are to be executed. The POS Server uses a standard SQL interface to a commercial relational database.

In the Common Lisp embodiment, application developers interact solely with the OMS module using standard Common Lisp syntax in their application programs. The DDL module is not implemented since the OTS module can extract the necessary information from the CLOS descriptions during program execution as that information is already available in the Common Lisp run-time system. This embodiment uses a raw disk-based implementation of the POS Server developed by the co-inventors.

The OMS module presents an application interface to perform standard database operations: initializing and terminating the present invention, beginning and committing or aborting database transactions (saving modified objects or discarding them, respectively), designating objects as persistent (to be saved to the database), explicitly retrieving objects from the database, designating objects as having been modified, removing objects from memory, defining the default storage group for logical clustering of objects, etc. The OMS module also supports an automatic and implicit retrieval of objects from the database when an application references a previously saved object that is currently not in primary memory. OMS also provides a facility to associate user-defined names with persistent objects to simplify retrieval of objects. These associations are also stored in the present invention's database. This name-object management module has a well-defined interface and can be replaced with a module of the application developer's choice.

As stated above, the present invention allows the application program to retrieve persistent objects from the database and reference the persistent object's data members or functions. The present invention accomplishes this by defining a new data type or class, the ZG_PTR, that functions equivalently to the current language constructs for referencing persistent objects (pointers in C++; symbols and values in Common Lisp). In addition, the present invention allows the application program to implicitly retrieve a persistent object from the database using an object faulting mechanism. When an application program references a persistent object, if the object is already in primary memory, the application program continues with its operations. If the object is not in primary memory, OMS automatically retrieves the object from the POS and calls upon the OTS module to translate and install the object in primary memory. Finally, the application program is allowed to proceed, unaware of this object faulting processing.

The OTS module is responsible for translating objects between their primary and secondary memory representations in a computer architecture-independent manner. When an object is being saved, the OTS module uses the information extracted by the DDL translator to determine the extent, or boundary, of the object and then translates all of the objects within the boundary to a computer architecture independent representation. When an object is retrieved from the POS, OTS creates the appropriate primary memory representation, assigns the object's values from the stored representation, and allocates OMS data structures for every reference contained in the object to other persistent objects.

The POS Server module provides a stable storage facility for the objects made persistent by the application program. Objects are stored in the computer's long term, or secondary, memory. The POS Server also provides to the OMS module concurrency control primitives and atomic transactions (all objects are saved or none are saved). Objects are stored as an untyped array of bytes which only OTS understands.

The present invention stores objects via the POS Server in a computer architecture-independent representation utilizing information about the computer's computational, or primary, memory architecture. Information on the content and structure of the objects is extracted from the object definitions declared in the supported languages. This allows applications written in any of the supported languages to store objects in the same POS. Currently the POS Server is implemented in a modular and portable fashion using an existing commercial Relational Database Management System (RDBMS). The POS Server interacts with the RDBMS using an embedded Structured Query Language (SQL) interface.

These and other features and advantages of the invention will be apparent to those skilled in the art from the following detailed description of a preferred embodiment, taken together with the accompanying drawings in which:

BRIEF DESCRIPTION OF DRAWINGS

FIG. 1 is a block diagram showing the operating context of the present invention operates within a computer;

FIG. 2 is a block diagram of the architecture of a preferred embodiment of the present invention;

FIG. 3 is a flow chart depicting the process flow of the DDL translator during execution according to the present invention;

FIG. 4 is a block diagram representing an example of a demon table used within the OMS module of the present invention;

FIG. 5 is a block diagram representing an example of an object map used within the OMS module of the present invention;

FIG. 6 is a block diagram depicting an example of the relationships between a PTR, an encapsulation, an object, and the type description of that object according to the present invention; and

FIGS. 7*a* and 7*b* are block diagrams showing the process of object translation between its primary and secondary memory representations according to the present invention.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT

The present invention is applicable in the development of object-oriented application systems which require the management, persistence and sharing of com-

plex and interrelated data. Before considering the present invention in detail, however, it should be noted that although the C++ implementation will be fully described, the preferred embodiment of the present invention can be implemented in any object-oriented programming language. Currently, a C++ implementation of the present invention runs on systems such as the Sun Microsystems Incorporated Sun4 or the Digital Equipment Corporation DEC3100 series of computer workstations. Also, a Common LISP implementation of the present invention runs on such systems as the Texas Instruments Incorporated Explorer series of computer workstations.

FIG. 1 shows a block diagram of the operating context of the present invention within a computer. A user 8, which may be human, another computer, or another application, interacts with the interactive devices 10 to send information to, and receive information from, user interface management system 12. Interactive devices 10 are also known as communication hardware. User interface management system 12, which is also known as communication software in turn sends the information to, and receives information from, an application software program 14 (hereinafter referred to as "application 14"). Application 14 interfaces with the computer's operating system software 16 (hereinafter referred to as "OS 16") and the present invention (hereinafter referred to as "OODB 18"). OODB 18 interfaces with OS 16 to utilize various operating system services. OODB 18 also interfaces with a Relational Database Management System Software 20 (hereinafter referred to as "RDBMS 20") to store and retrieve objects created by application 14. RDBMS 20 interfaces with operating system 16 to utilize various OS system services and with secondary memory storage devices 24 to physically store or retrieve the objects created by application 14 and managed by OODB 18. OS 16 also interfaces with additional computer hardware 22 as necessary to provide its system services to application 14, OODB 18, and RDBMS 20.

FIG. 2 shows a block diagram of the architecture of a preferred embodiment of the present invention during execution of application 14. All of the modules in OODB 18 are linked together along with a C++ type description 40 (hereinafter referred to as "types 40"), produced by running the data definition language translator (hereinafter referred to as "DDL 80"; shown in FIG. 3) on a set of application-defined classes, to form a library that can be linked with application 14 to form a program that can be executed in the computer environment shown in FIG. 1.

Application 14 interfaces directly with the object management system 42 (hereinafter referred to as "OMS 42") and creates or deletes Persistent Object References 44 (hereinafter referred to as "PTR 44") to create, retrieve, and access persistent objects managed by OMS 42.

OMS 42 interfaces with name manager 46 to manage an association of names supplied by application 14 with independent persistent objects created or retrieved by application 14 using PTRs 44. OMS 42 interfaces with demon table 48 to execute functions defined by application 14 and registered with OODB 18 by running DDL 80. OMS 42 also interfaces with object map 50 to manage the actual persistent objects referenced by PTR 44. OMS 42 further interfaces with object translation system 52 (hereinafter referred to as "OTS 52") to translate persistent objects between their primary and secondary memory representations. OTS interfaces with types 40 created by DDL 80. Finally, OMS 42 interfaces with persistent object server 54 (hereinafter referred to as "POS Server 54") to store and retrieve the persistent objects managed by OMS 42.

POS Server 54 interfaces with RDBMS 20 using a standard Structured Query Language (SQL) interface to physically store and retrieve the persistent objects managed by OMS 42. RDBMS 20 interfaces with secondary memory storage devices 24 to physically store and retrieve the objects created by application 14 and managed by OODB 18.

The various modules of the present (DDL 80, OMS 42, name manager 46, demon table 48, object map 50, PTR 44, OTS 52, and POS Server 54) will now be considered in more detail.

Prior to developing an application system that interfaces with the present invention, the application developer must register the C++ classes to be used in the application system with the present invention. This is accomplished by executing the DDL translator described below.

The DDL Translator 80 (hereinafter referred to as "DDL 80") is based on three separate application programs that are executed in sequence (see FIG. 3). DDL 80 receives as input one or more data definition language source files 26 written by the developer of application 14. These source files 26 contain class definitions in the C++ language plus additional keywords and syntactic constructs defined by the present invention (see Table 1 for an example of such a DDL source file). DDL 80 then extracts sufficient information from source files 26 to generate a C++ source file 40 (hereinafter referred to as "types 40"; see Table 2 for an example of such a C++ source file) containing type description information for use by OMS 42 and OTS 52, and a set of C++ source files 38 for every class definition in the source file(s). These files are then subsequently used by the application programmer when writing, compiling, and executing C++ programs that create and manipulate instances of the classes, as well as save or retrieve them using the present invention.

The first program in the sequence depicted in FIG. 3 is the commercially available C language preprocessor cpp 28. This program receives as input source files 26 and produces a copy of the input file(s) with all cpp directives evaluated and expanded (expanded file 30). Use of this program in the present invention does not require any modification.

TABLE 1

```
class ptrs : persistent                  //to indicate independent persistence
{
private:
int   len;                               //controls length of data member 'mpd'
//Pointers to array of data types
char   *[10] cpd;                        //constant length
char   *[len] mpd;                       //length controlled by data member 'len'
char   *[sentinel ' 0'] spd;             //length terminated by hexadecimal zero
boundary char *bpd;                      //referenced data is ignored when saved
```

TABLE 1-continued

```
classPTR pers_ref;                  //reference To independent persistent object
public:
//Definition of Demon functions
demon create int setup();           //creation time demon
demon restore int resetup();        //fetch time demon
demon commit int ready();           //commit time demon
demon abort   int cleanup 1);       //abort time demon
};
```

TABLE 2

```
char *Type-Descriptions□ = {  /* DDL RELEASE 0.1.5 */
/* 0-2: OCDB Classes */
"16|&|0|1|0|1|254|0|.|&|7|0|4|1|255|1|1|&|7|0|8|1|255|1|0|&",
"24|&|0|1|0|1|254|0|.|&|7|0|16|1|255|1|1|&|7|3|20|1|255|1|1|&~,
"8|&|7|3|0|1|.|.|2|&",
/* 3: User Class A */
"12|&|7|3|0|1|.|.|3|&",
/* 4: User Class B */
"8|&|7|3|0|1|.|.|4|&",
/* 5: User Class C */
"|2|&|7|3|8|1|.|.|5|&",
};
Interpretation of Fields
"12|&|7|3|8|1|.|.|5|&'
|  | | | | | | |Index into this array of referenced data
|  | | | | | |Number of instances in the array
|  | | | | |How to determine number of distances in array
|  | | | |Number of dimensions in array
|  | | |Offset of this pointer or PTR
|  | |Type of pointer
|  |Data type of referenced data
|Size of an instance of this class
```

TABLE 3

```
-#include "zg-ptr.h"
// The $PTR class
class $;
class $PTR : public ?
{
    public:
      /* methods */
      $& operator* ();
      $* operator-> ();
      $PTR& operator= (Zg-Eo&);
      $PTR& operator= ($ *);
      $PTR& operator= ($PTR &);
      operator $* ();
      void make-absent ();
      /* constructors & destructors */
      inline $PTR () ;
      inline $PTR ($ *);
    private:
      #
      /* friends */
      friend $PTR& persist ($& object, zg-Uint sg=0);
};
      The '$' are replaced with the name of the user defined class.
      The '?' is replaced with the name(s) of class from which this
            class is derived.
      The '#' is replaced with a C++ class definition if this class
            has defined Demons.
```

The second program in the sequence is the DDL processor ddl 32. This program receives as input expanded file 30 produced by the first program, scans expanded file 30 for specific information, and generates a typeobject database 34 for use by the third program, ddlpost 36. The ddl 32 program uses a lexical scanner based on the lexical analysis program generator lex (not shown) to scan source files 26 and return each token (syntactic unit not shown) to the main functions of ddl 32. In addition to scanning source files 26 for specific information, ddl 32 performs full syntactic and limited semantic checking on the input file, and generates appropriate error and warning messages.

ddl 32 processes each class definition in the input file as follows. First, it scans the input tokens until it recognizes the C++ keyword "class". If the "class" keyword is found, ddl 32 continues to scan for and then extract the class name, the name(s) of any classes from which this class is to be derived, and a keyword, "persistent", that indicates whether or not instances of this class should become "independent persistent objects" (IPO) when saved to the database. If this class is derived from another class and does not have the "persistent" keyword, ddl 32 determines if the latter class is also a persistent class. If so, this class is marked as an persistent class. If "persistent" is not present and if this class is not derived from a persistent class, instances of this class can still be saved to the database, but they will become "dependent persistent objects" (DPO) physically stored with an IPO. If the class(es) from which this class is to be derived has(have) not been previously processed by ddl 32 in this file or in previous executions of DDL 80, an error message is generated.

Second, ddl 32 scans the rest of the class definition for the declarations of data members and extracts the name of the data member, the name of the data type (fundamental C++ or user-defined type, including classes), and then determines the class's memory alignment, size, padding between it and the next data member, and its offset from the beginning of the class. ddl 32 also extracts information indicating if the data member is an array, the number of elements in the array, if the data member is a C++ pointer, the name of the data type of the referenced data, the number of data items referenced by the pointer (which can be specified by a sentinel, a decimal or hexadecimal value, or the name of a integer data member in this class), whether it has the keyword "boundary" before it (the referenced data will

not be saved if an instance of this class is saved to the database), or if the data member is a PTR **44**, the name of the referenced class. Unless the data types used for data members or referenced from this class (using C++ pointers or PTRs **44**) have been processed by DDL **32** during the eighth step discussed below, an error message is generated.

Third, if ddl **32** determines that any of the data members are in the public portion of the class definition, an error message will be generated and the class definition will not be processed during the eighth step discussed below. Fourth, ddl **32** also scans for the existence of the C++ keyword "virtual" before any function declaration inside the class definition, as this keyword affects the overall size of an instance of this class. Fifth, ddl **32** also scans for the existence of the keyword "demon" before any function declaration inside the class definition and notes this for later processing by ddlpost **36**.

Sixth, once the entire class definition has been scanned, ddl **32** computes the size for instances of this class by adding the size of all classes from which this class derives (retrieved from the type description database) to the size of all data members in this class (if the data member is an embedded instance of a class, the size is retrieved from the type description database). If this class has virtual functions (as detected in the fourth step discussed above), the size of the class is increased by an amount equal to the size of a C++ pointer to a C++ virtual table pointer. If the keyword "persistent" is present (as detected in the first step discussed above), the size of the class is increased by an amount equal to the size of an instance of the PTR **44** class. The alignment and padding of this class are then calculated based on the overall size computed.

Seventh, these scanning and extraction steps continue until the end of the expanded file **30** is encountered.

Eighth, after all class definitions are scanned and the end of the input file is encountered, ddl **32** then generates and outputs the following information into type description database **34**: the name of the class, the size, alignment, and padding information for the entire class, a string containing information on every C++ pointer and PTR included in the class, the names of all classes or structures that this class references (using C++ pointers or PTRs) or contains, the names of functions defined for the various demons supported by the present invention, and a copy of the entire class definition from source file **26**.

The third program in the sequence is ddlpost **36** which receives as input type description **34** generated by ddl **32** and generates C++ source and header files source files **38** and types **40** as follows.

First, ddlpost **36** reads the contents of type description **34** and creates data structures to hold the information. Second, for every class in type description database **34**, source files **38** is produced, which may be composed of two separate C++ source files.

If the file is a persistent class, a C++ header file is produced containing the definition of a new class derived from the PTR **44** class. This file is produced by editing a predefined template file (see Table 3) using a stream editor to convert certain character strings in the template file to the name of this class. A second C++ header file is produced that contains cpp **28** include directives for every class from which this class is derived, is used as the data type for a data member, or is referenced by a C++ pointer or PTR **44**; the class definition; and C++ statements to allow the PTR **44** class functions to manipulate instances of this class. If this class has demons, this second C++ header file is also includes C++ statements to allow the Demon class functions to manipulate instances of this class. Similarly, if this class is persistent, this second C++ header file also contains definitions for the PTR **44** class functions generated in the above-described PTR **44** file.

This second C++ source file is temporary and is produced by appending the first item to the file based on the information extracted in the first two steps of ddl **32**. The second item is then appended to the file as obtained from type description database **34** (sixth item in type description database **34**). The third item is appended to the file by generating a temporary file which resulted from editing a predefined template file (see Table 4) using a stream editor to convert certain character strings in the template file to the name of this class. The final two items are appended to the file in a similar manner.

TABLE 4

```
--inline $PTR::$PTR ()
{
    #
    eop->teop = eop->init-teop (1, @, 0);
}
inline $& $PTR::operator* ()
{
    return (*(($ *) (eop->rop ? eop->rop : fault ())));
}
inline $* $PTR::operator-> ()
{
    return (( $ *) (eop->rop ? eop->rop : fault ()));
}
inline $PTR& $PTR::operator= (Zg-Eo& rhs)
{
    return (($PTR&) this->assign-eo (rhs));
}
inline $PTR& $PTR::operator= ($PTR& rhs)
{
    return (($PTR&) this->assign-ptr (rhs));
}
inline $PTR& $PTR::operator= ($* rhs)
{
    if (rhs == NULL) return (($PTR&) this->assign-null ());
      else return (($PTR&) this>assign-star (rhs, rhs->thisptr));
}
inline $PTR::operator $* ()
{
    return (($ *) (eop->rop?eop->rop:
    (eop->oid==NullOid?NULL:fault())));
}
inline $PTR::make-absent ()
{
    if (!this->remove object ()) delete ($ *) this->eop->rop;
    this->eop->rop = NULL;
}
inline $PTR::$PTR ($* rhs)
{
    *this = rhs;
}
inline $PTR& persist ($& object, Zg-Uint sg)
{
    return (($PTR&) object.thisptr.set-persist (&object, sg));
}
```

The '$' are replaced with the name of the user defined class
The '#' is replaced with a C++ statement if this class has defined Demons.

Third, after all classes in type description database **34** have been processed, C++ source file types **40** is generated containing C++ statements which define an array of character strings that contains the type descriptions of the classes processed, as described above in connection with the eighth step of ddl **32**. This file is compiled by the application developer and linked with the software modules of application **14** and OODB **18** to

allow the array to be used by OMS 42 and OTS 52 during execution of application 14 interfacing with OODB 18.

Although eight processing steps are discussed with regard to ddl 32, to extract the class and type description information, the same effect could be achieved by combining steps.

The application program interfaces with one instance of OMS 42 as well as numerous instances of PTR 44 to create, manipulate, store, and retrieve persistent objects. Each independent persistent object created by the application is assigned an object ID 78. Each object ID 78 is composed of a storage group number (indicates in which storage group the object is stored), an object number (indicates the specific object within the storage group), and a time stamp (indicates the time that object was saved to the database). Each of these fields is represented by a 32 bit unsigned integer value. A NULL object ID 78 is used to indicate the absence of a valid object ID. In addition, an encapsulation 62 (hereinafter referred to as "encapsulation 62") is created and associated with each independent persistent object. Encapsulations 62 are described later during the description of PTRs 44.

An instance of OMS 40 contains the following data members:

Architecture ID is a reference to information describing the architecture of the current computer hardware. This data member is added to the database when the database is originally initialized for use with the current computer hardware. This data member is used by OTS 52 when translating objects between their primary and secondary memory representations.

Default storage group contains the number of the storage group where objects will be stored by default. It can be set using Default-Storage-Group 108.

Name manager is a reference to name manager 46 for this instance of OMS 40. It can be set using Name-Manager 116.

Demon table is a reference to demon table 48 for this instance of the OMS 40. It is set during Create 100 and used during Create 300 (described later), Commit-Transaction 112, Abort-Transaction 114, and Fetch 120–124.

Object map is a reference to object map 50 for this instance of OMS 40. It is set during ZG-Create 100 and used by OMS 42.

Transaction Id contains the current transaction number. It is used during Begin-Transaction 110, Commit-Transaction 112, and Abort-Transaction 114.

POS Server is a reference to an instance of POS Server 54. It is set during Create 100 and used by OMS 42 to store or retrieve objects in the database.

Active indicates whether or not this instance of OMS 40 is active; that is, Startup 102 has been called before a call to Shutdown 106 has been made.

There is also a global variable named Exists that indicates whether or not an instance of OMS 40 has been created by application 14. This variable is checked during Startup 102.

Table 5 lists interface functions 100–130 together with their function names and arguments, which OMS 42 provides to application 14. A description of each function is given below. These functions either update the private state or return some information about the private state of an instance of OMS 42. Interface functions 100–128 update the private state of the instance, while interface function 130 returns information about the private state of the instance to application programs.

In most functions, error checking is performed after most individual actions. Error checking is not considered fundamental to the understanding of and operation of the present invention, and will therefore not be further described.

TABLE 5

OMS Application Interface Functions

	Name	Argument
100	Create	none
102	Startup	none
104	Delete	none
106	Shutdown	none
108	Default-Storage-Group	Storage Group
110	Begin-Transaction	none
112	Commit-Transaction	none
114	Abort-Transaction	none
116	Name-Manager	Name Manager
118	Default-Name-Context	Name Context
120	Fetch	Object Name, Lock, Time Context
122	Fetch	PTR, Lock, Time Context
124	Fetch	Object ID, Lock, Time Context
126	Lock-State	Object Name, Time Context
128	Lock-State	PTR, Time Context
130	Time	none

Create 100 creates an instance of OMS 42. This function is called whenever an instance of OMS 42 is created statically, automatically, or dynamically, depending on the C++ variable declaration used in the application program. During creation, function Startup 102 is called.

Startup 102 can also be called at any time after Shutdown 106 is called. Since only one OMS 42 instance is allowed per application program, the value of the Exists global variable is checked to see if one already exists. If one does exist, an error value is returned. If one does not exist, the default storage group data member is set to the minimum storage group number if this is the first time an OMS 42 instance is being created. If the current OMS 42 instance had previously been created during this execution of application 14, the previous value in the default storage group data member will be used. Demon table 48 and object map 50 are created and assigned to their associated data members. Next, an instance of POS Server 54 is created and assigned to the POS Server data member. The architecture ID object for the computer hardware within which application 14 and OODB 18 are executing is retrieved from POS Server 54 and assigned to the architecture ID data member. An instance of name manager 46 is created and assigned to the name manager data member. Begin-Transaction 110 is called, followed by a call to Default-Name-Context 116 to create the default name context in the database. If this is the first time this database has been used, a call is made to Commit-Transaction 112 to save the new default name context in the database. If not, Abort-Transaction 114 is called. Finally, the active data member is set to indicate that this instance of OMS 42 is active, the exists global variable is set to indicate that an instance of OMS 42 exists, and control is returned to the caller of this function.

Delete 104 deletes an instance of OMS 42. This function is called whenever an OMS 42 instance is deleted statically, automatically, or dynamically, depending on the C++ variable declaration used in the application program. During deletion, function Shutdown 106 is called.

Shutdown 106 checks the value of the Active data member to determine whether this instance of OMS 42

is active. If it is not, control is returned immediately to the caller of this function, as no work must be performed. If this instance is active, for every encapsulation 62 currently in object map 50, its modified data member is set to "Not Modified" and then encapsulation 62 is deleted. Name manager 46, object map 50, and POS Server 54 are deleted and their associated data members are set to a null pointer, thereby indicating that the value is not valid. Finally, the transaction ID data member is to zero, the active data member is set to indicate that this instance of OMS 42 is inactive, and control is returned to the caller of this function.

Default-Storage-Group 108 updates the private state of this instance of OMS 42 as follows. If the supplied storage group number is greater than the minimum storage group number, Is-Sg-Valid 504 is called to insure that a storage group with that number exists in the database. If one does exist, the default storage group data member is assigned using the supplied storage group number and the previous value of this data member is returned. Otherwise, a value of 0 0 (zero) is returned.

Begin-Transaction 110 is used to mark the beginning boundary of a transaction. All interactions between the application program and OMS 42 must occur within a transaction. Begin-Transaction 516 is called to insure that POS Server 54 begins a transaction. If the call succeeds, the transaction ID data member is incremented by one and that value is returned. Otherwise, a value of −1 is returned.

Commit-Transaction 112 is used to mark the ending boundary of a transaction. If the transaction ID data member indicates that there is not a current transaction, a value of −1 is returned. Otherwise, an instance of POS Encapsulation 70 (herein referred to as "POS encapsulation 70"; discussed below) is created if one does not already exist. Next, every encapsulation 62 in object map 50 is examined to determine how many do not have an object ID 78 (those objects which have been created by the application program since the last call to Commit-Transaction). If the number is greater than zero, Alloc-Symbolic-Name 506 is called. If the requested number of object IDs 78 could not obtained, an error is returned. If the call was successful, Begin-Commit 508 is called, which returns a timestamp for this commit (which will be saved in each POS encapsulation 70 during translation), or a error value. For every encapsulation 62 currently in object map 50, the following actions take place. If the modified data member in encapsulation 62 indicates that the object has not been modified, it is bypassed for processing. Otherwise, if the concurrent lock data member in the encapsulation 62 indicates that the application program does not have a WRITE lock on the object, an error is returned since the application program must have a WRITE lock on all objects in order to create new versions of them in the database. Otherwise, if a Commit Demon for the object's type was registered in demon table 48, that Commit Demon is called (see description of demon table 48). Next, OTS Internal2External is called and passed POS encapsulation 70, which it updates with an external representation 402 of the object. If the translation succeeds, Put-Object 514 is called to create a new version of the object in the database. After every encapsulation 62 in object map 50 has been processed in this manner, End-Commit 510 is called to commit the changes to the database, followed by a call to End-Transaction 510 to end POS Server's 54 transaction. Since the transaction has completed, other application programs may retrieve the objects from the database, update them, and create new versions of them. Versions accessible through this instance of OMS 42 are then considered invalid. Thus, for each encapsulation 62 currently in Object Map 50, the following actions take place. The concurrent lock data member is set to "Invalid" and the modified data member is set to "Not Modified". If the number of references to this encapsulation 62 from the application program is greater than zero, the object is deleted if OMS 42 created it. Otherwise, the object pointer data member in encapsulation 62 is set to null. If the number of references is zero, encapsulation 62 is deleted, which may delete the object if OMS 42 created it. Finally, the transaction ID data member is decremented by one and the number of objects committed to the database is returned.

Abort-Transaction 114 is also used to mark the ending boundary of a transaction but does not commit any objects to the database. If the transaction ID data member indicates there is not a current transaction, a value of −1 is returned. Otherwise, for every encapsulation 62 currently in object map 50, if an Abort Demon for the object's type was registered in demon table 48, that Abort Demon is called. Next, the concurrent lock data member is set to "Invalid" and the modified data member is set to "Not Modified". Finally, Abort-Transaction 520 is called to allow POS Server 54 to terminate its current transaction, the transaction ID data member is decremented by one, and control is returned to the caller of this function.

Name-Manager 116 simply assigns the name manager data member to the supplied instance of name manager 46 and returns the previous value of this data member.

Default-Name-Context 118 calls Name-Context 204, to ensure that the supplied name context exists, and returns.

Fetch 120–124 has three forms. The first takes an object name, the second takes a PTR 44 associated with an object, and the third takes an object ID 78 of an object. The first form calls Return-OID 210 which returns object ID 78 of the supplied object name, or a null object ID 78 if the user-defined name had not been registered, in which case a null address is returned. The second form retrieves object ID 78 from encapsulation 62 associated with supplied PTR 44. In any case, the following actions occur once an object ID 78 has been obtained. Encapsulation 62 associated with object ID 78 is retrieved from object map 50 and the object pointer data member is checked to determine if the associated object is or is not in primary memory. If the value indicates that it is in primary memory, the address of the object in primary memory is returned. If the value indicates that it is not in primary memory, Get-Object 512 is called with the supplied lock and time context (which default to READ and MOST-RECENT, respectively) and returns a POS encapsulation 70 (which contains the object's external representation 402 and its references to other persistent objects). Next, OTS External2Internal is called with POS encapsulation 70 to create a primary memory representation of the object using the object's external representation 402 in POS encapsulation 70. The references to other persistent objects are registered with OMS if they have not already been registered. Finally, the modified and deleteable data members of encapsulation 62 are set to "False" and "True", respectively (since OMS 42 just created the object), and the address of the object in primary memory is returned.

Lock-State 126–128 has two forms similar to Fetch 120–124. The first takes an object name, while the second takes a PTR 44 associated with an object. As in Fetch 120–124, the first form calls Return-OID 210 to return object ID 78 of the supplied object name or a null object ID 78 if the user-defined name had not been registered, in which case a value indicating that application 14 does not currently have a valid lock on the object is returned. The second form retrieves object ID 78 from encapsulation 62 associated with the supplied PTR. In any case, encapsulation 62 associated with object ID 78 is retrieved from object map 50 and the concurrent lock data member is checked to determine if the application program currently has a valid lock on the object. If so, the value of the concurrent lock data member is returned. Otherwise, a value indicating that the application does not currently have a valid lock on the object is returned.

Time 130 returns the current time received from OS 16.

OMS 42 interacts with name manager 46 to manage the association of names supplied by the application with independent persistent objects created or retrieved by application 14 using PTRs 44.

An instance of Name Manager 46 contains the following data members:

First context references the first name context in a list of all name contexts defined in the database currently being accessed by application 14 using the present invention. It is set during Create 200 and accessed when searching for name contexts or name entries.

Current context references the current name context in a list of all name contexts defined in the database currently being accessed by application 14 using the present invention. It is set during Create 200 to the first context and during Name-Context 204 to the new default name context. It is accessed when searching for name entries.

Table 6 lists interface functions 200–210, with their function names arguments, which name manager 46 provides to OMS 42. A description of each function is given below. These functions either update or return some information about the private state of an instance of the name manager class. Interface functions 200–208 update the private state of the instance, while interface function 210 returns information about the private state of the instance to application programs.

TABLE 6

Name Manager Interface Functions

	Name	Argument
200	Create	none
202	Delete	none
204	Name-Context	Context Name
206	Name	Object Name, Object Oid
208	Unname	Object Name, Object Id
210	Return-OID	Object Name

Create 200 creates an instance of a name manager 46 as follows. Begin-Transaction 110 is called and then Fetch 124 is called to retrieve the default root context. If the default root context does not exist, one is created, assigned to the first context data member, a new object ID 78 obtained by indirectly calling Alloc-Symbolic-Name 506, and Commit-Transaction 112 is called to save the default root context in the database. If the default root context already exists, Abort-Transaction 114 is called. Finally, the current context data member is set to the new or previously existing default root context object and control is returned to the caller of this function.

Delete 202 simply deletes the supplied instance of name manager 46.

Name-Context 204 searches the list of contexts, accessible from the first context data member, for an context entry which has the supplied context name. If the context name is in an entry, a 0 (zero) is returned. If it is not in any entry, a new context entry is created using the supplied context name. It is added to the front of the list of contexts, the current context data member is assigned to the new context entry, and a 1 (one) is returned.

Name 206 accesses the current context data member's list of name entries and searches for a name entry which has the supplied object name. If the object name is in an entry, a −1 is returned since no duplicate object names are allowed. If it is not in any entry, a new name entry is created using the supplied object name and object ID 78 and added to the end of the list. A 0 (zero) is returned.

Unname 208 accesses the current Context data member's list of name entries and searches for a name entry which has the supplied object name. If the object name is not in an entry, a −1 is returned. If it is in an entry and the object ID 78 in the entry matches supplied object ID 78, the name entry is deleted from the list and a 0 (zero) is returned. If the two object ID 78s did not match, a −1 is returned.

Return-object ID 210 accesses the current context data member's list of name entries and searches for a name entry which has the supplied object Name. If the object name is not in an entry, a null object ID 78 is returned. If it is in an entry, object ID 78 in the name entry is returned.

FIG. 4 is a block diagram of demon table 48 used by OMS 42 during certain OMS 42 functions. Demon table 48 is an array of entries that contains two data members: an object ID 78 for a user-defined type or class, and a reference to demon object 74.

During the DDL translation process, the application developer can add the keyword "demon" and an event keyword before a class function name (see Table 1 above, bottom portion of the class declaration). Classes with these demon keywords are referred to as "demon classes". The existence of these keywords is recognized by DDL 32 and the names of the associated functions are added to type description database 34. During execution of ddlpost 36, additional C++ statements and functions are added to source files 38 for each class annotated with the demon keywords. First, an additional data member, PTR Demon 76, is defined that will be shared by all instances of this class' PTRs 44 during execution of an application program using this class. Second, another class, demon object 74, is defined and includes functions which will call the user-defined functions named in the original class definition.

Prior to execution of application 14, the application developer compiles source files 38 and links them with the software modules of application 14 and OODB 18. After execution of application 14 is started, code required to statically create instances of PTRs 76 is executed (this is the code generated in source files 38). During execution of application 14, whenever an instance of OMS 42, an instance of demon table 48 is initialized. Whenever an instance of a demon class PTR 44 is created by the application program, the value of its

associated PTR Demon 76 is checked to see if an instance of demon object 74 exists. If one does exist, processing continues. If one does not exist, one is created and PTR Demon 76 is set to reference new demon object 74. When demon object 74 is created, an entry is added to demon table 48. Object ID 78 data member is set to object ID 78 of the type description object (in Types 40) for this demon class, while the demon object reference data member is set to the newly created instance of demon object 74.

While processing persistent objects during certain OMS 42 functions, OMS 42 checks demon table 48 to see if a demon object 74 for the type of the persistent object has been registered. It accomplishes this by comparing object ID 78 of the type description for the given object and object IDs 78 in demon table 48. If there is not a match, processing of the given object continues. However, if there is a match, OMS 42 will call one of the demon functions as specified in the demon object 74 referenced from the appropriate demon table 48 entry. The demon function can then manipulate the object prior to OMS 42 continuing with its processing.

FIG. 5 is a block diagram of object map 50 used by OMS 42. Object map 50 manages encapsulations 62 created when an independent persistent object is created by application 14 or retrieved from the database. Object map 50 is constructed as a two-level index based on storage group and object number. The first level is array 56 whose elements are records containing a storage group number and a pointer to a storage group hash table 58. The second level is storage group hash table 58 whose entries 60 contain pointers to encapsulations 62 and the next entry 60 in hash table 58. The interface provided by object map 26 to OMS 42 is as follows:

Add Encapsulation locates the appropriate hash table 56 (creating a new one if necessary), creates new entry 60, associates it with supplied encapsulation 62, and adds it to storage group hash table 58.

Find Encapsulation locates the appropriate hash table object map 58 and searches hash table 58's entries 60 for one that references encapsulation 62 which has the same object ID 78 as the supplied object ID 78. If an entry 60 is found, the address of the associated encapsulation 62 is returned. If an entry is not found, or there is not a hash table 58 for the supplied storage group, a null address is returned.

Remove Encapsulation locates the appropriate hash table 58 and searches hash table 58's entries 60 for one that references encapsulation 62 which has the same object ID 78 as the supplied object ID 78. If an entry 60 is found, entry 60 is removed from storage group hash table 58 and deleted. No action occurs if an entry 60 is not found.

As described earlier, application 14 creates instances of PTR 44 to create, manipulate, store, and retrieve persistent objects using the present invention. As can been seen in FIG. 6, an instance of a PTR 44 contains only one data member, encapsulation, which references an instance of the encapsulation class (encapsulation 62).

An instance of encapsulation 62 contains the following data members.

Type (class) description ID references a limited description of the definition of the C++ class of which the object is an instance. Specifically, it is a C++ pointer to a character string that contains information on every C++ pointer and PTR 44 in an instance of the class type of the object referenced via the Object Pointer data member. It is extracted from types 40.

Number of references to encapsulation 62 is incremented by 1 (one) every time a PTR 44 is created that references encapsulation 62, and decremented by 1 (one) every time a PTR 44 that references encapsulation 62 is deleted.

Concurrent lock indicates the current lock held on the object associated with the encapsulation. It is set when encapsulation 62 is created, when an object is retrieved from the database, or when a lock is upgraded.

Time stamp indicates the time that the associated object was saved to the database.

Object identifier is object ID 78 of the object associated with this PTR 44.

Object pointer is a C++ pointer that references the primary memory representation of the associated object. Unless the value is null (thereby indicating that the value is not valied), the associated object is resident in primary memory.

Persistent indicates whether this persistent object should be saved or already has been saved to the database. It is set by calling Persist 304.

Modified indicates whether this persistent object has been modified by application 14. Newly-created persistent objects are marked as "Modified", while persistent objects retrieved from the database are marked as "Not Modified". Persistent objects can be marked as "Not Modified" by calling Set-Modified 310.

Deleteable indicates whether OMS 42 can delete this persistent object from primary memory. If application 14 created this object, OMS 42 cannot delete it. If OMS 42 created the object (when retrieving it from the database), it can delete the persistent object.

Therefore, it can be seen in FIG. 6 that given a PTR 44, one can access its associated encapsulation 62, and from there, access the associated object 64.

Table 7 lists interface functions 300–326 together with their function names and arguments, which PTR 44 provides to application 14. A description of each function is given below. These functions either update the private state or return some information about the private state of PTR 44. Interface functions 300–310, 316, and 320–322 update the private state of PTR 44, while Interface functions, 312 314, 318, 324, and 326 return information about the private state of PTR 44 to application 14.

TABLE 7

PTR Application Interface Functions

	Name	Argument
300	Create	none
302	Delete	none
304	Persist	Object, Storage Group
306	Name	Object Name
308	Unname	Object Name
310	Set-Modified	none
312	Is-Modified	none
314	Lock-State	none
316	Upgrade-Lock	New Lock
318	In-Memory	none
320	Make-Absent	none
322	Make-Present	none
324	Timecontext	none
326	Timestamp	none

Create 300 creates an instance of PTR 44 for application 14 to reference objects. Instances are created statically, automatically, or dynamically, depending on the

C++ variable declaration used in application 14. First, an encapsulation 62 is created and assigned to the encapsulation data member. Next, encapsulation 62 for the type description object (in types 40) for this PTR's 44 class definition is obtained from object map 50, assigned to the associated encapsulation's 62 type description ID data member, and control is returned to the caller of this function.

Delete 302 deletes an instance of PTR 44 class. If there are not any references to encapsulation 62 by any other PTRs 44, encapsulation 62 is removed from object map 50. If the deleteable data member of encapsulation 62 indicates that associated object 64 was created by OMS 42, object 64 is deleted. Finally, encapsulation 62 is deleted and control is returned to the caller of this function.

Persist 304 is used to indicate that supplied object 64 should be saved to the database when the next Commit-Transaction 112 is performed. First, the persistent data member of encapsulation 62 is set to "True". Then, POS Alloc-Symbolic-Name 506 is called to obtain object ID 78 for the object, using the supplied storage group number or the value of the default storage group data member of OMS 42, if one was not supplied. Finally, if a Persist Demon for the object's type was registered in demon table 48, that persist demon is called, and control is returned to the caller of this function.

Name 306 associates the supplied object name with object 64 with PTR 44. Since names may only be associated with independent persistent objects, if the object ID data member of associated encapsulation 62 is null object ID 78, POS Alloc-Symbolic-Name 506 is called to obtain object ID 78 for object 64. Next, Name 206 is called to create the new name and control is returned to the caller of this function.

Unname 308 dissociates the supplied object name from object 64 associated with PTR 44 by calling Unname 208, and returns control to the caller of this function.

Set-Modified 310 shows that the application program has modified object 64 (this is checked during Zeitgeist Commit-Transaction 112 processing). First, the modified data member of encapsulation 62 is set to "True", Upgrade-Lock 212 is called requesting a WRITE lock, and then control is returned to the caller of this function.

Is-Modified 312 retrieves the value of the modified data member of encapsulation 62 and returns that value.

Lock-State 314 returns either the current value of the concurrent lock data member of encapsulation 62 or "Invalid" if the data member indicates that the current lock is not valid.

Upgrade-Lock 316 attempts to upgrade an existing lock on object 64 to the requested lock. If the supplied lock is not a valid lock type, an error is returned. If the supplied lock is a READ-ONLY lock, the concurrent lock data member of encapsulation 62 is set to that value. If the current lock is valid and stronger than or equal to the supplied lock, control is returned to the caller of this function. If neither of these conditions apply, Set-Lock 524 is called requesting the supplied lock. If the lock was set, the concurrent lock data member of encapsulation 62 is set to the supplied lock and control is returned to the caller of this function.

In-Memory 318 checks the value of the object pointer data member of encapsulation 62 and returns a 1 (one) if the value indicates associated object 64 is in primary memory or 0 (zero) otherwise.

Make-Absent 320 updates encapsulation 62 such that subsequent references to object 64 using PTR 44 will cause OMS 42 to retrieve object 64 from POS Server 54. First, if the deleteable data member of encapsulation 62 indicates that OMS 42 allocated object 64, the deleteable data member of encapsulation 62 is set to indicate that OMS 42 did not allocate object 64 and the modified data members is set to "Not Modified". Next, object 64 and any dependent persistent objects it references are deleted. Regardless whether OMS 42 allocated object 64, the object pointer data member of encapsulation 62 is set to null to indicate that object 64 is not resident in primary memory, and control is returned to the caller of this function.

Make-Present 322 makes object 64 accessible through PTR 44. First, the object pointer data member of encapsulation 62 is checked to see if object 64 is already in primary memory. If so, control is returned to the caller of this function. Otherwise, Fetch 124 is called to retrieve object 64 object from POS Server 54 and install it in primary memory. If that is successful, the primary memory address of object 64 is returned.

Timecontext 324 retrieves and returns the value of the time context data member of object ID 78 of encapsulation 62. This value indicates the time context used when retrieving the object from the database.

Timestamp 326 retrieves and returns the value of the time stamp data member of encapsulation 62. This value indicates the time object 64 was saved to the database.

OMS 42 provides an additional capability to allow an application program to have independent persistent objects implicitly and automatically retrieved from POS Server 54 without having to call Fetch 120–124 or Make-Present 322. This capability is called object faulting and PTR 44 processes an occurrence of an object fault as follows. When an application program dereferences an instance of PTR 44, code is executed to check the value of the object pointer data member of associated encapsulation 62. If the value indicates that associated object 64 is already in primary memory, the application program continues. If the value indicates that associated object 64 is not in primary memory, Fetch 124 is called using object ID 78 data member in encapsulation 62. After object 64 has been installed in primary memory by Fetch 124, the application program continues. In the C++ embodiment, the code necessary to perform this processing is automatically generated by DDL 80 (included in the C++ files of source files 38). Specifically, a definition is provided to overload the C++ dereference operators ("a→func( )" and "(*a).func( )") for independent persistent object classes.

FIG. 7*a* shows primary memory representation 400 (hereinafter referred to as "object 400") of a C++ object. Although a C++ object can contain various data members, the present invention supports the following three categories of data types for the data members. The first category comprises those data types which can be fully embedded in the object (data members 410–414). The second category comprises those data types which are a reference to an independent or dependent persistent object using a C++ pointer (data members 416–420). The third category comprises those data types which are a reference to an independent persistent object using a PTR 44 (data members 406, 408, and 422). FIG. 7*b* shows the secondary memory representation of a C++ object, which is composed of two items, external representation 402 and external references 404.

OTS 52 provides two interface functions to OMS 42 to translate persistent objects between their primary and secondary memory representations.

Internal2External is the first function and translates an object from its primary (or internal) to its secondary (or external) memory representation. It accomplishes the translation by first allocating two memory buffers to hold external representation 402 and external references 404. Next, information about the object (size, type ID) how and each reference in the object (type of pointer, type of referenced object, how to determine the number of objects referenced, etc.) is obtained from types 40, as generated by DDL 80. Second, the data in object 400 is copied to external representation 402, starting at the beginning of external representation 402.

Then, for every reference, the following actions are performed. If the reference is a C++ pointer and it is not a null value, information on the referenced object is obtained from types 40. The referenced object is copied to external representation 402 after any other copied data and the offset of the copied referenced object (relative to the beginning of external representation 402) is stored at the location in external representation 402 of the original C++ pointer. For example, data member 418 references another object. After copying, data member 436 contains the offset from the beginning of external representation 402 of the copy of that object, namely, object 444. If the data type of the referenced object is a C++ structure or class, a similar translation process is performed on the referenced object. If the C++ pointer is a "boundary" pointer, the value in external representation 402 is set to 0 (zero). For example, if data member 416 were a boundary pointer, data member 434 would contain a 0(zero). If the reference is a PTR 44, object ID 78 of the referenced object is obtained from associated encapsulation 62 and copied to external references 404 after any other copied object IDs 78, and the offset of that copy (relative to the beginning of external references 404) is stored in the appropriate location in external representation 402. For example, data member 422 is a PTR 44. Object ID 78 of the referenced object is stored in external reference 452; the location of external reference 452 (in external references 404) is stored in data member 440. Essentially, referenced objects or object IDs 78 are copied to a location and that location is stored where the original reference was copied. Finally, POS encapsulation 70 is updated (with this object's object ID 78, the architecture ID from the current instance of OMS 42, the type description ID associated with this object, and newly-created external representation 402 and external references 404) and returned to the caller of this function.

External2Internal is the second function and translates an object from its secondary (or external) to its primary (or internal) memory representation. It accomplishes the translation by first checking the architecture ID in supplied POS encapsulation 70 to see that it matches the architecture Id data member in the current instance of OMS 42. In the current embodiment, if they do not match, the object cannot be translated and an error is returned to the caller of this function. Second, information about the object is obtained as is done in Internal2External above. Third, sufficient primary memory to hold the object is allocated and the object is copied from external representation 402 to the newly allocated memory (for example, object 400).

Next, the following actions are performed for every reference. If the reference is a C++ pointer, information on the referenced object is obtained from types 40. Sufficient primary memory is allocated to hold the referenced object and the referenced object is copied from the location in external representation 402 as indicated by the value of the original reference. The corresponding reference in object 400 is updated to reference the newly allocated referenced object. For example, data member 436 is a C++ pointer whose referenced object is stored at object 444. After memory is allocated to hold the referenced object, the referenced object is copied from object 444 to the newly-allocated memory and data member 418 is updated to reference the newly allocated referenced object. If the data type of the referenced object is a C++ structure or class, a similar translation process is performed on the referenced object. If the reference is PTR 44, the following actions are performed. If object ID 78 of the referenced object is not null, the timestamp in object ID 78 is set to the timestamp of this object's encapsulation 62. This insures that the referencing object's timestamp will be used when the referenced object is retrieved from POS Server 54. Next, a reference to encapsulation 62 for the referenced object is obtained from object map 50 (one will be created if the object is not currently known to OMS 42). If application 14 does not currently hold a valid lock on the referenced object (determined by examining encapsulation 62 just obtained), the value of the concurrent lock data member of referenced object's encapsulation 62 is set to the value of the concurrent lock data member of this object's encapsulation 62. This insures that the referencing object's lock will be used when the referenced object is retrieved from POS Server 54. If application 14 holds a valid lock on the referenced object, Upgrade-Lock 316 Lock is called requesting a lock equal to the value of the concurrent lock data member in this object's encapsulation 62. Lastly, once all the references in the object have been processed, control is returned to the caller of this function.

POS Server 54 is used by OMS 42 to store and retrieve persistent objects in the database. In the C++ embodiment, the present invention uses a commercially available RDBMS 20 to store external representation 402 and external references 404 of an independent persistent object described above. The present invention interacts with RDBMS 20 using the embedded Standard Query Language (SQL) interface provided by the vendor. This allows the present invention to replace one vendor's RDBMS with another vendor's RDBMS with insignificant modifications to the present invention.

In order to store the persistent objects created by application 14 and managed by OMS 42, the following relational tables are defined in RDBMS 20.

The first table is the groups table which contains two attributes, storage group and object number. The purpose of the table is to control the allocation of object identifiers (object IDs 78) within storage groups. See the description of Alloc-Symbolic-Name below for details on how the object numbers are allocated.

The second table is the value table with the attributes shown below in Table 8. The purpose of this table is to hold sufficient information about an independent persistent object in order to identify it by its object ID 78 (composed of the first three attributes), identify the architecture of the computer hardware in which application 14 and OODB were executing when the object was saved, identify the object's type (class) description,

identify the number of independent persistent objects it references, recreate the object, and install it in primary memory. If external representation 402 of the object is longer than the length allowed for attribute values by RDBMS 20, there are multiple tuples in this table for the single large object, with all values the same except for the "sequence number" (which begins with one and is incremented by one for every additional tuple) and the "external representation" (which continues where the previous tuple left off).

TABLE 8

Attributes in Groups Table
Storage Group;
Object Number;
Commit time;
Sequence number;
Object size;
Architecture Object Storage Group;
Architecture Object Number;
Architecture Object Commit time;
Type Descriptor Storage group;
Type Descriptor Object Number;
Type Descriptor Commit time;
Number of user defined attributes associated with the object;
Number of system defined attributes associated with the object;
Number of references to other persistent objects, including this object; and
External representation of object.

The last table is refto table with the attributes shown below in Table 9. The purpose of this table is to hold the references from one independent persistent object to other independent persistent objects. Each tuple set (one or more tuples with the same object ID) in the value table is associated with one or more tuples in this table by virtue of the storage group, object number, and commit time being the same as the associated value table tuple. If there are multiple references from an object, there are multiple tuples in this table with the values for the storage group, object number, and commit time attributes in the associated value tuple set, except for the "sequence number" (which begins with one and is incremented by one for every additional tuple). The number of tuples in this table associated with a tuple set in the value table equals the value of the "number of references" attribute in the associated value table tuple set.

TABLE 8

Attributes in Refto Table
Storage Group;
Object Number;
Commit time;
Sequence number;
Referenced Object Storage Group;
Referenced Object Object Number; and
Referenced Object Commit time.

Tables 10 and 11 show an example of how the object seen in FIGS. 7a and 7b might appear stored in the value and refto tables, respectively.

TABLE 10

VALUE Table Tuples						
Storage Group	Object Number	Commit Time	Sequence Number	Object Size	Other.. Attributes	External Representation
5	1438	654318	1	83468	—	[array of bytes]
5	1438	654318	2	83468	—	[array of

TABLE 10-continued

VALUE Table Tuples						
Storage Group	Object Number	Commit Time	Sequence Number	Object Size	Other.. Attributes	External Representation
						bytes]
5	1438	654318	3	83468	—	[array of bytes]

TABLE 11

REFTO Table Tuples						
Storage Group	Object Number	Commit Time	Sequence Number	Referenced Storage Group	Referenced Object Name	Referenced Commit Time
5	1438	654318	1	5	1438	654318
5	1438	654318	2	8	3481	654318
5	1438	654318	3	12	3347	654318

OTS 52 and POS Server 54 pass between each other encapsulations 62 and POS encapsulations 70. An instance of POS encapsulation 70 contains the following data members.

Object ID is object ID 78 of the object being passed in this POS encapsulation 70.

Architecture ID is object ID 78 of a persistent object describing the architecture of the computer hardware in which application 14 and OODB 18 are currently executing.

Type description ID is object ID 78 of a persistent object that describes the primary memory representation of the object being passed in this POS encapsulation 70.

Size of object external representation is the number of bytes that object external representation 402 in this POS encapsulation 70 contains.

Object external representation is the external representation 402 of the object being passed in this POS encapsulation 70 contains.

Number of external references is the number of external references of the object being passed in this POS encapsulation 70.

External references is the external references 404 of the object being passed in this POS encapsulation 70 contains.

A instance of POS Server 54 contains the following data members.

Commit in progress records whether a commit operation is currently in progress. A commit starts when Begin-Commit 508 is called and ends when End-Commit 510 is called.

Transaction is a reference to a transaction machine (not shown) which monitors transactions being performed by multiple programs as they access the database.

Table 12 lists interface functions 500–524 together their function names and arguments, which POS Server 54 provides to OMS 42. A description of each function is given below. These functions either update the private state or return some information about the private state of an instance of POS Server 54. Interface functions 500–502, 506–510, and 514–524 update the private state of the instance, while interface function 504 and 512 returns information about the private state of the instance to OMS 42.

TABLE 12

	POS Server Interface Functions	
	Name	Argument
500	Create	none
502	Delete	none
504	Is-Sg-Valid	Storage Group
506	Alloc-Symbolic-Name	Storage Group, Number Requested
508	Begin-Commit	none
510	End-Commit	none
512	Get-Object	Encapsulation
514	Put-Object	POS Encapsulation
516	Begin-Transaction	none
518	End-Transaction	none
520	Abort-Transaction	none
522	Set-Lock	Encapsulation, New Lock, Wait?
524	Set-Lock	Encapsulation, New Lock

Create **500** creates an instance of POS Server **54** and connects to RDBMS **20** using the appropriate SQL statements (this allows further calls from POS Server **54** to RDBMS **20**). Next an instance of the transaction machine is created and assigned to the transaction data member. Finally, control is returned to the caller of this function.

Delete **502** deletes an instance of a POS Server **54**, calls Abort-Transaction **520** in case End-Commit **510** had not been called by OMS **42**. Next, a disconnect from RDBMS **20** is performed using the appropriate SQL statements, making certain that any uncommitted changes previously made by OMS **42** are rolled back or deleted. In addition, this signals the end of calls from POS Server **54** to RDBMS **20**. Finally, control is returned to the caller of this function.

Is-Sg-Valid **504** issues a SQL query to RDBMS **20** to determine if a tuple exists in the groups table with the supplied storage group. If a tuple exists, a 1 (one) is returned, otherwise a 0 (zero) is returned.

Alloc-Symbolic-Name **506** issues a SQL query to RDBMS **20** to retrieve the tuple in the groups table with the supplied storage group and makes a copy of the value of the object number attribute from the returned tuple. The object number attribute is incremented by the number requested and an SQL query is issued to update the modified tuple and commit the update in RDBMS **20**. Finally, the copied value of the object number attribute is returned.

Begin-Commit **508** is used to record the beginning of a commit. If the value of the commit in progress data member indicates that a commit is in progress, an error is returned since only one commit can be in progress at any time. Otherwise, the current time from OS **16** is obtained, the value of the commit in progress data member is set to indicate that a commit is in progress, and the time obtained from OS **16** is returned.

End-Commit **510** is used to record the end of a commit. If the value of the commit in progress data member indicates that a commit is not in progress, an error is returned. Otherwise, an SQL query is issued to commit all pending changes previously sent to RDBMS **20** by POS Server **54**. If that query fails, another SQL query is issued to rollback any pending changes to insure that none of the pending changes are seen by any other application **14** which may access this database. Finally, the value of the commit in progress data member is set to indicate that a commit is not in progress, and control is returned to the caller of this function.

Get-Object **512** begins by calling Is-Sg-Valid **504** to insure that the storage group in object ID **78** of supplied encapsulation **62** exists. If it does not exist, an error is returned. Otherwise, an SQL query is issued to RDBMS **20** to retrieve the first tuple in the value table which matches object ID **78** of supplied encapsulation **62**. Next, using the value of the object size attribute in the retrieved tuple, a memory buffer sufficient to hold the entire object is allocated. If the object size attribute indicates that there are additional tuples with the same object ID (because the retrieved external representation **402** was too large to fit in one tuple), additional SQL queries are issued to retrieve the remaining value tuples. The portions of external representation **402** from the tuples retrieved are copied into the memory buffer. Next, an SQL query is issued to RDBMS **20** requesting the first tuple in the refto table which matches object ID **78** of supplied encapsulation **62**. If the "number of references" attribute in the value tuple indicates that there are additional tuples with the same object ID, additional SQL queries are issued to retrieve the remaining refto tuples. The values of the referenced object storage group, object number, and commit time from these tuples are used to create an external references **404**. Next, an SQL query is issued to RDBMS **20** to commit any pending work to insure that any RDBMs locks on any of the tuples retrieved are not further retained. Next, a POS encapsulation **70** is created and updated with a copy of object ID **78** from supplied encapsulation **62**, the architecture ID, type description ID, size of external representation **402** and external representation **402** collected from the value tuple(s), and the number of external references and external references **404** collected from the refto tuple(s). Finally, this newly-created POS encapsulation **70** is returned to the caller of this function.

Put-Object **514** begins by checking the value of the commit in progress data member to insure that a commit is in progress. If one is not in progress, an error is returned. Otherwise, the length of external representation **402** in supplied POS encapsulation **70** is used to calculate how many value tuples will be needed to store external representation **402**. An SQL query is issued to insert sufficient value tuples to store external representation **402**, using object ID **78**, architecture ID and type description ID data members in supplied POS encapsulation **70** for the other attribute values in the new value tuples (see Table 10). Next, the number of external references data member in supplied POS encapsulation **70** is used to determine how many refto tuples will be needed to store external references **404**. An SQL query is issued to insert sufficient refto tuples to store external references **404**, using object ID **78** in supplied POS encapsulation **70** for the other attribute values in the new refto tuples (see Table 11). Finally, control is returned to the caller of this function.

Begin-Transaction **516** is used to mark the beginning of a transaction started by application **14**. If the transaction data member indicates that a transaction is already in progress, an error is returned. Otherwise, a new transaction ID is obtained from the transaction machine, and control is returned to the caller of this function.

End-Transaction **518** is used to mark the end of a transaction started by application **14**. If the transaction data member indicates that a transaction is not currently in progress, an error is returned. Otherwise, the transaction machine is called to end the current transaction and control is returned to the caller of this function.

Abort-Transaction **520** is also used to mark the end of a transaction started by application **14**. If the transaction

data member indicates that a transaction is not currently in progress, an error is returned. Otherwise, the transaction machine is called to end the current transaction and control is returned to the caller of this function.

Set-Lock 522-524 has two forms which attempt to set a lock on the object associated with supplied encapsulation 62. The first form will wait until the lock has been granted while the second will return if the lock cannot be granted on the first attempt. If object ID 78 of supplied encapsulation 62 is a null object ID, control is returned to the caller of this function since the object does not yet exist in the database and is implicitly WRITE locked by application 14. If the request is for a READ-ONLY lock, the concurrent lock data member in supplied encapsulation 62 is set to that value and control is returned to the caller of this function. If the request is for a READ or a WRITE Lock, the transaction machine is called to obtain the lock. If the lock could not be granted due to an error, an error is returned. If the lock were granted, the concurrent lock data member in the supplied encapsulation 62 is set to that value and control is returned to the caller of this function.

Application 14 interface with OMS 20 and PTR 44 by embedding function calls to OMS 20 and PTR 44 as well as use instances of PTR 44 in programming language statements in the application software. In the preferred embodiment of the present invention, OMS 42 consists of one library of software and one C++ header file corresponding to OMS 42. The application developer includes the OMS 42 header file along with source files 38 generated by DDL 80 into the application software during compilation. Types 40 is also compiled by the application developer. The library and object files produced during the compilation of the C++ source files are then linked to form an application load module. In the preferred embodiment of the present invention, OODB 18 and application 14 using OODB 18 execute in the same address space, while RDBMS 20 executes in a different address space.

If the function calls to OODB 18 are extracted from application 14 software, the resulting set of instructions would have the following basic control flow.

First, an instance of OMS 42 would be created to begin the interface with OMS

Second, one or more instances of PTR 44 would be created to allow application 14 to create and manipulate as well as store and/or retrieve persistent objects using the present invention.

Third, application 14 would call OMS Begin-Transaction 110.

Fourth, OMS Default-Storage-Group 108 would be called to define a new default storage group, if so desired by application 14, in which newly-created objects would be stored.

Fifth, OMS Default-Name-Context 118 would be called to define a new default name context, if so desired by application 14, in which new object names would be registered.

Sixth, application 14 would create instances of objects and manipulate them using functions defined for the instances' classes, including assigning references from one object to one or more other objects.

Seventh, for those objects to be saved to the database, application 14 would assign the objects to the appropriate PTR 44 instances and call PTR Set-Modified 310 and PTR Persist 304 on those PTRs 44.

Eighth, for those objects to be explicitly retrieved after they have been saved to the database, application 14 would call PTR Name 306 to associate an object name with each object.

Ninth, application 14 would call OMS Commit-Transaction 112 to end the transaction and save the objects to the database.

Tenth, application 14 would either call OMS Shutdown 106 or delete the instance of OMS 42 to terminate the interface with OMS 42.

Eleventh, if application 14 had called OMS Shutdown 106, it would call OMS Startup 102 to restart the interface with OMS 42. If application 14 had deleted its instance of OMS 42, it would create a new instance of OMS 42 to restart the interface with OMS 42.

Twelfth, application 14 would call OMS Begin-Transaction 110 to begin a new transaction.

Thirteenth, OMS Default-Storage-Group 108 would be called to define a new default storage group, if so desired by application 14, in which newly-created objects would be stored.

Fourteenth, OMS Default-Name-Context 118 would be called to define a new default name context, if so desired by application 14, in which new object names would be registered.

Fifteenth, application 14 would call OMS Fetch 120-124 to explicitly retrieve one or more persistent objects from the database.

Sixteenth, application 14 could then manipulate the retrieved objects using functions defined for the instances' classes, including assigning references from one object to one or more other objects. Manipulation of these objects would automatically retrieve other persistent objects as they are accessed by application 14. New objects could also be created by application 14.

Seventeenth, for those objects to be saved to the database, application 14 would call PTR Set-Modified 310 on the appropriate PTRs 44. Application 14 would also need to call PTR Persist 304 on the newly-created objects.

Eighteenth, for those newly-created objects to be explicitly retrieved after they have been saved to the database, application 14 would call PTR Name 306 to associate an object name with each object.

Ninteenth, application 14 would call OMS Commit-Transaction 112 to save the modified objects and newly-created objects to the database. Alternatively, application 14 would call OMS Abort-Transaction 114 to discard the modified objects and newly-created objects.

Twentieth, application 14 would either call OMS Shutdown 106 or delete the instance of OMS 42 to terminate the interface with OMS 42.

While a specific embodiment of the invention has been shown and described, various modifications and alternate embodiments will occur to those skilled in the art. Accordingly, it is intended that the invention be limited only in terms of the appended claims.

We claim:

1. A system for storing objects in at least one relational database management system for retrieval during later execution of an application program, comprising:

an object manager;

a persistent object storage server with a SQL interface to said at least one relational database manager and said object manager; and

an object translator accessible by said object manager to generate a first buffer containing at least one

object and a second buffer containing at least one reference from said at least one object to additional at least one objects; said first buffer and said second buffer interpretable by said at least one relational database management system, wherein said object manager passes said retrieved objects to said object translator for use by said application program during execution;

wherein said persistent object storage server stores said first buffer and said second buffer into said at least one relational database management system; and

wherein said persistent object storage server retrieves said first buffer and said second buffer from said at least one relational database management system for return to said object manager.

2. The system for storing objects of claim 1, including:

said object translator generating said first and second buffers by using at least one object type description of user-specified class definitions generated by a data definition language processor and accessible from said object manager.

3. The system for storing objects of claim 1, including:

said persistent object storage server stores in a first table said first buffer contents using a first object identifier as a key for the buffer, along with an object type identifier, and an architecture identifier, wherein said architecture identifier indicates the architecture of the computer where said application program is running; and

said persistent object storage server stores in a second table said second buffer contents using a second object identifier as a key for the buffer.

4. The system of claim 3, wherein said first object identifier and said second object 3dentifier are identical.

* * * * *

EXHIBIT B

US005329471A

United States Patent [19]

Swoboda et al.

[11] Patent Number: 5,329,471

[45] Date of Patent: Jul. 12, 1994

[54] EMULATION DEVICES, SYSTEMS AND METHODS UTILIZING STATE MACHINES

[75] Inventors: **Gary L. Swoboda**, Sugar Land; **Martin D. Daniels**, Houston; **Joseph A. Coomes**, Missouri City, all of Tex.

[73] Assignee: **Texas Instruments Incorporated**, Dallas, Tex.

[21] Appl. No.: **84,787**

[22] Filed: **Jun. 29, 1993**

Related U.S. Application Data

[63] Continuation of Ser. No. 911,250, Jul. 7, 1992, abandoned, which is a continuation of Ser. No. 387,549, Jul. 31, 1989, abandoned, which is a continuation-in-part of Ser. No. 093,463, Sep. 4, 1987, abandoned, and a continuation-in-part of Ser. No. 057,078, Jun. 2, 1987, Pat. No. 4,860,290.

[51] **Int. Cl.**[5] **G06F 15/20**

[52] **U.S. Cl.** **364/578**; 364/DIG. 1; 364/264.3; 364/264.5; 364/267.4; 364/267.7; 371/16.2; 395/500

[58] **Field of Search** 364/578, 579, 580; 371/16.1, 16.2, 22.3; 395/500

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,023,142	5/1977	Woessner	36.1/200
4,268,902	5/1981	Berglund et al.	364/200
4,277,827	7/1981	Carlson et al.	364/200
4,312,066	1/1982	Bantz et al.	371/22.3 X
4,314,333	2/1982	Shibayama et al.	364/200
4,441,154	4/1984	McDonough et al.	364/200
4,513,418	4/1985	Bardell, Jr. et al.	371/25
4,519,078	5/1985	Komonytsky	371/22.3
4,594,711	4/1986	Thatte	371/25
4,597,080	6/1986	Thatte et al.	371/25
4,601,034	5/1986	Sridhar	371/25
4,615,029	9/1986	Hu et al.	370/89
4,621,363	11/1986	Blum	371/25
4,680,733	7/1987	Duforestel et al.	364/900
4,687,988	8/1987	Eichelberger et al.	371/22.3
4,698,588	10/1987	Hwang et al.	324/73 R
4,701,921	10/1987	Powell et al.	371/25
4,710,931	12/1987	Bellay et al.	371/25
4,710,933	12/1987	Powell et al.	371/25
4,788,683	11/1988	Hester et al.	371/16.1
4,801,870	1/1989	Eichelberger et al.	371/22.3
4,855,954	8/1989	Turner et al.	364/716 X
4,857,835	8/1989	Whetsel, Jr.	324/73 R
4,872,169	10/1989	Whetsel	371/22.3
4,879,688	11/1989	Turner et al.	364/716 X
4,896,296	1/1990	Turner et al.	365/189.08
5,103,450	4/1992	Whetsel	371/22.1

FOREIGN PATENT DOCUMENTS

2195185A 3/1988 United Kingdom .

OTHER PUBLICATIONS

Y. Mochida et al., "A High Performance LSI Digital Signal Processor for Communication", *IEEE Journal on Selected Areas in Communications, vol. SAC-3, No. 2, pp. 347-356, Mar. 1985.*

WE DSP16 Digital Signal Processor Information Manual, pp. 1-5, 1987.

Second-Generation TMS320 User's Guide, Texas Instruments, pp. D-1-E-8, Dec. 1987.

P. Gifford, "Sequent's Symmetry Series: Software Breadboarding Caught 95% of the Design Errors", *VLSI Systems Design*, pp. 2-6, Jun. 1988.

(List continued on next page.)

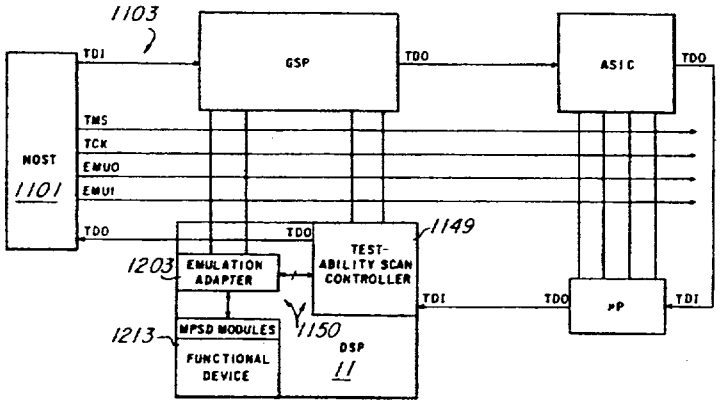
Primary Examiner—Edward R. Cosimano

Attorney, Agent, or Firm—James F. Hollander; Richard Donaldson; James C. Kesterson

[57] **ABSTRACT**

An emulation device including a serial scan testability interface having at least first and second scan paths, and state machine circuitry connected and responsive to said second scan path generally operable for emulation control of logical circuitry associated with said emulation device.

69 Claims, 41 Drawing Sheets



OTHER PUBLICATIONS

"Application Development Environment", AT&T Technologies, Inc., 1988, Single page.

"DSP56001: 56-Bit General Purpose Digital Signal Processor, Motorola", pp. 1-20, 1988.

G. Sohie, et al., "A Digital Signal Processor with IEEE Floating-Point Arithmetic", *IEEE Micro*, pp. 49-67, Dec. 1988.

J. R. Boddie, et al., "A Floating Point DSP with Optimizing C Compiler" IEEE 1988, pp. 2009-2012.

"DSP96001: 96-Bit General-Purpose Floating-Point Digital-Signal Processor (DSP), Motorola", pp. 1-22, 1988.

TMS370 Family Data Manual Texas Instruments, pp. 14-6, and 14-11 through 14-16, Mar. 1988.

First-Generation TMS320 User's Guide, Texas Instruments, pp. E-1-E-8, Apr. 1988.

"Test-Bus Interface Unit", Honeywell HTIU214PG, undated, received Jul. 1989.

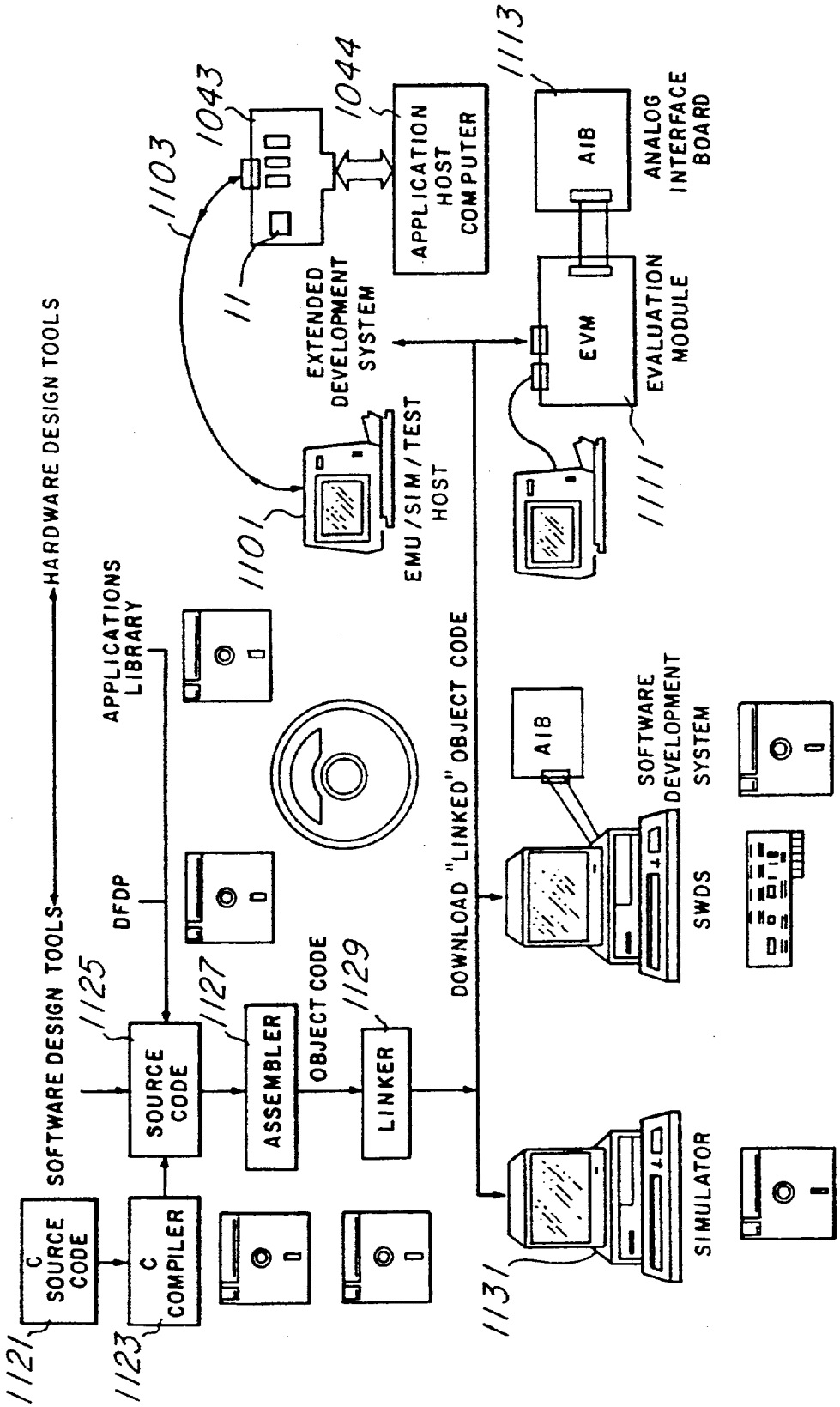

HARDWARE DESIGN TOOLS
SOFTWARE DESIGN TOOLS
C SOURCE CODE
C COMPILER
SOURCE CODE
ASSEMBLER
OBJECT CODE
LINKER
DFDP
APPLICATIONS LIBRARY
DOWNLOAD "LINKED" OBJECT CODE
EMU/SIM/TEST HOST
EXTENDED DEVELOPMENT SYSTEM
APPLICATION HOST COMPUTER
AIB
ANALOG INTERFACE BOARD
EVM
EVALUATION MODULE
AIB
SOFTWARE DEVELOPMENT SYSTEM
SWDS
SIMULATOR
1101
1103
11
1043
1044
1113
1111
1121
1123
1125
1127
1129
1131

Fig. 44

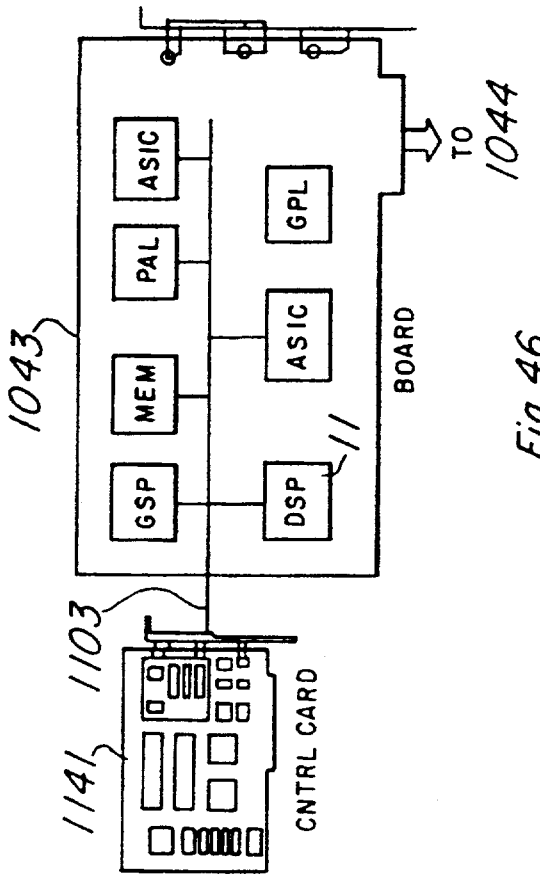

1043
GSP
MEM
PAL
ASIC
DSP
ASIC
GPL
11
BOARD
TO
1044
1103
1141
CNTRL CARD

Fig. 46

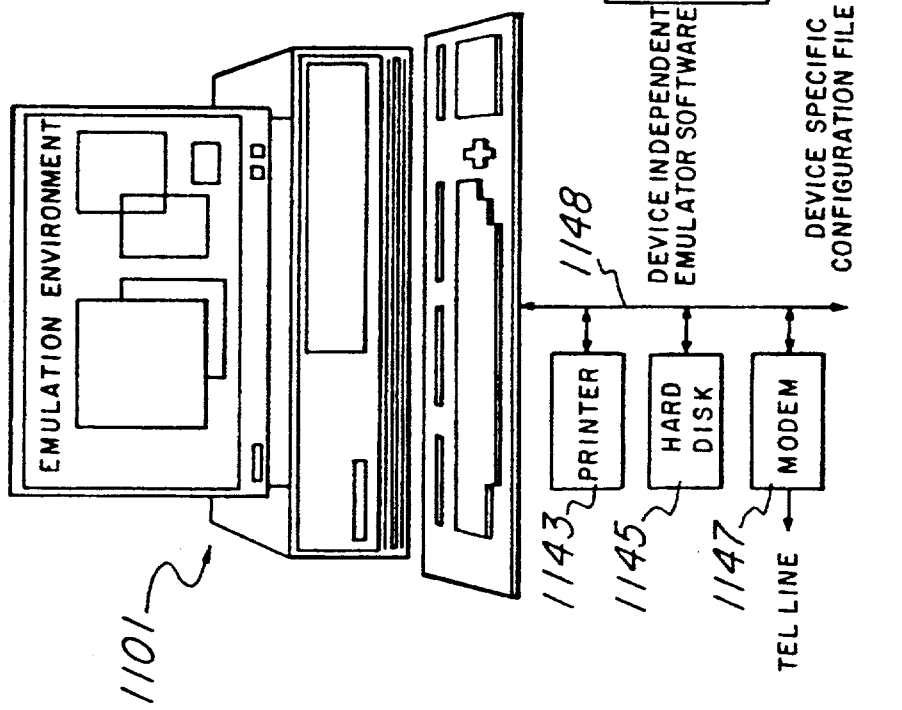

USER INTERFACE (WINDOW DRIVEN)
(TEST EXECUTIVE)
GPL REGISTER
ASIC #1
PAL
GSP CONFIG FILE
DSP CONFIG FILE
DEVICE INDEPENDENT EMULATOR SOFTWARE
DEVICE SPECIFIC CONFIGURATION FILE
EMULATION ENVIRONMENT
1148
PRINTER
HARD DISK
MODEM
1143
1145
1147
TEL LINE
1101

Fig. 45

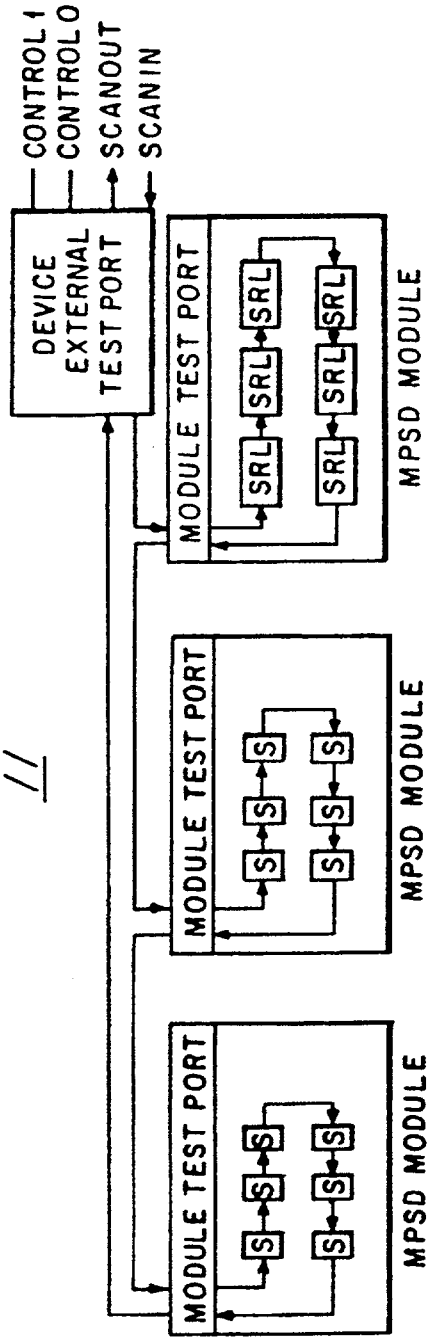

CONTROL 1
CONTROL 0
SCANOUT
SCANIN
DEVICE EXTERNAL TEST PORT
MODULE TEST PORT
SRL
MPSD MODULE
MODULE TEST PORT
S
MPSD MODULE
MODULE TEST PORT
S
MPSD MODULE
11

Fig. 47

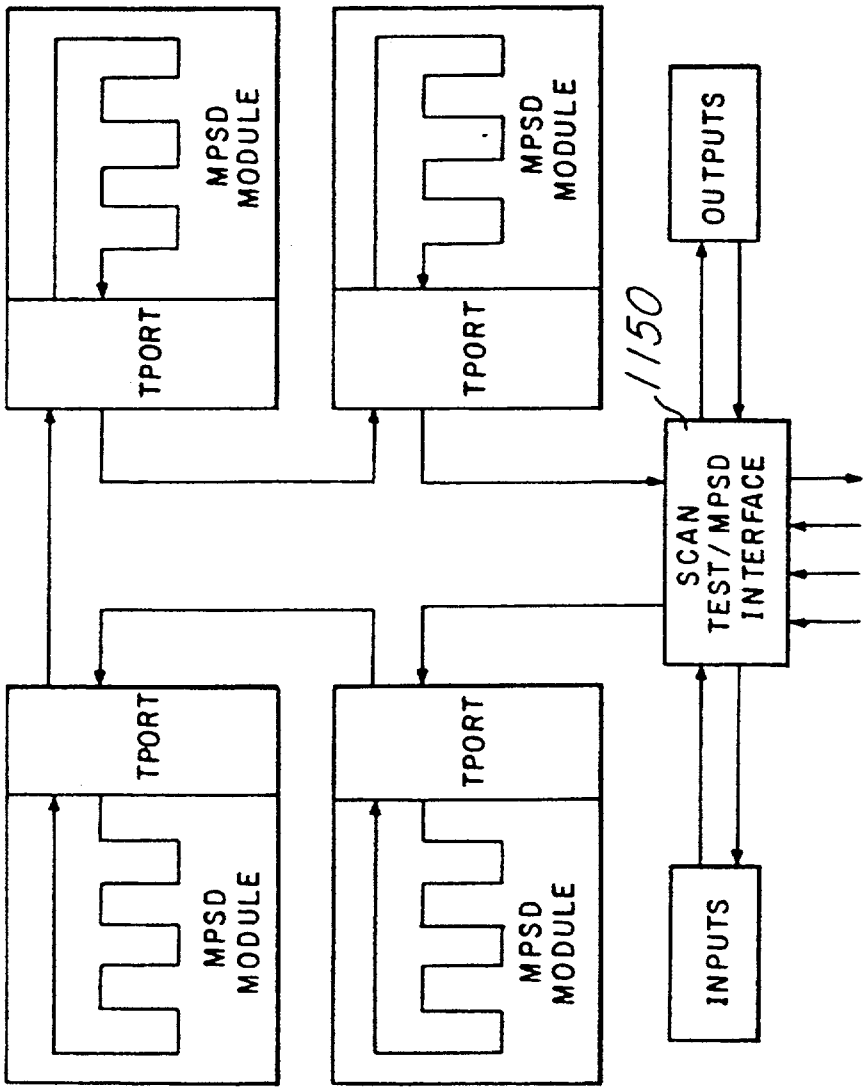

MPSD MODULE
TPORT
MPSD MODULE
TPORT
TPORT
MPSD MODULE
TPORT
MPSD MODULE
OUTPUTS
1150
SCAN TEST/MPSD INTERFACE
INPUTS

Fig. 48

11

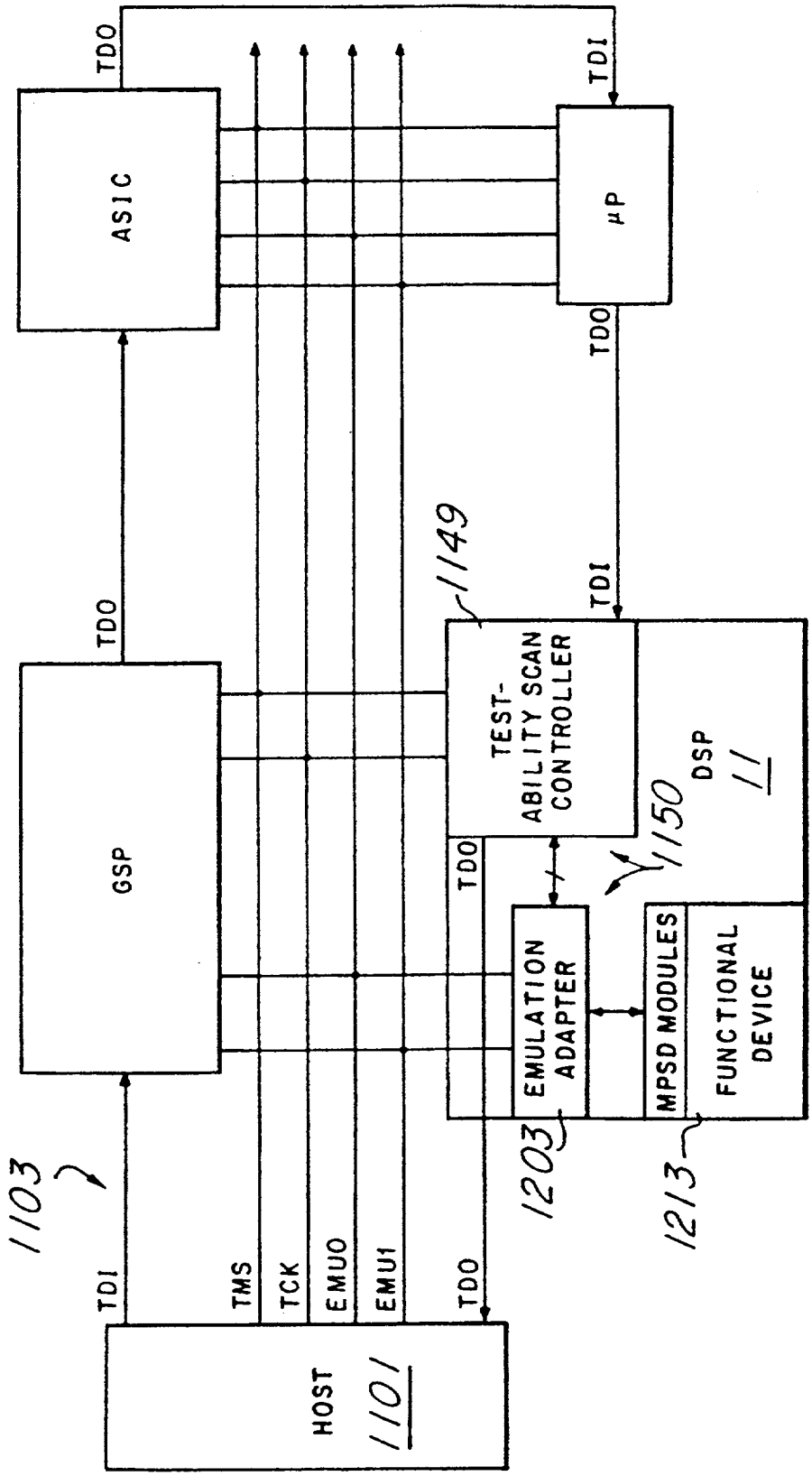

TDO
ASIC
TDI
μP
TDO
TDO
GSP
1149
TDI
TEST-
ABILITY SCAN
CONTROLLER
DSP
11
1150
TDO
EMULATION
ADAPTER
MPSD MODULES
FUNCTIONAL
DEVICE
1203
1213
1103
TDI
TMS
TCK
EMU0
EMU1
TDO
HOST
1101

Fig. 49

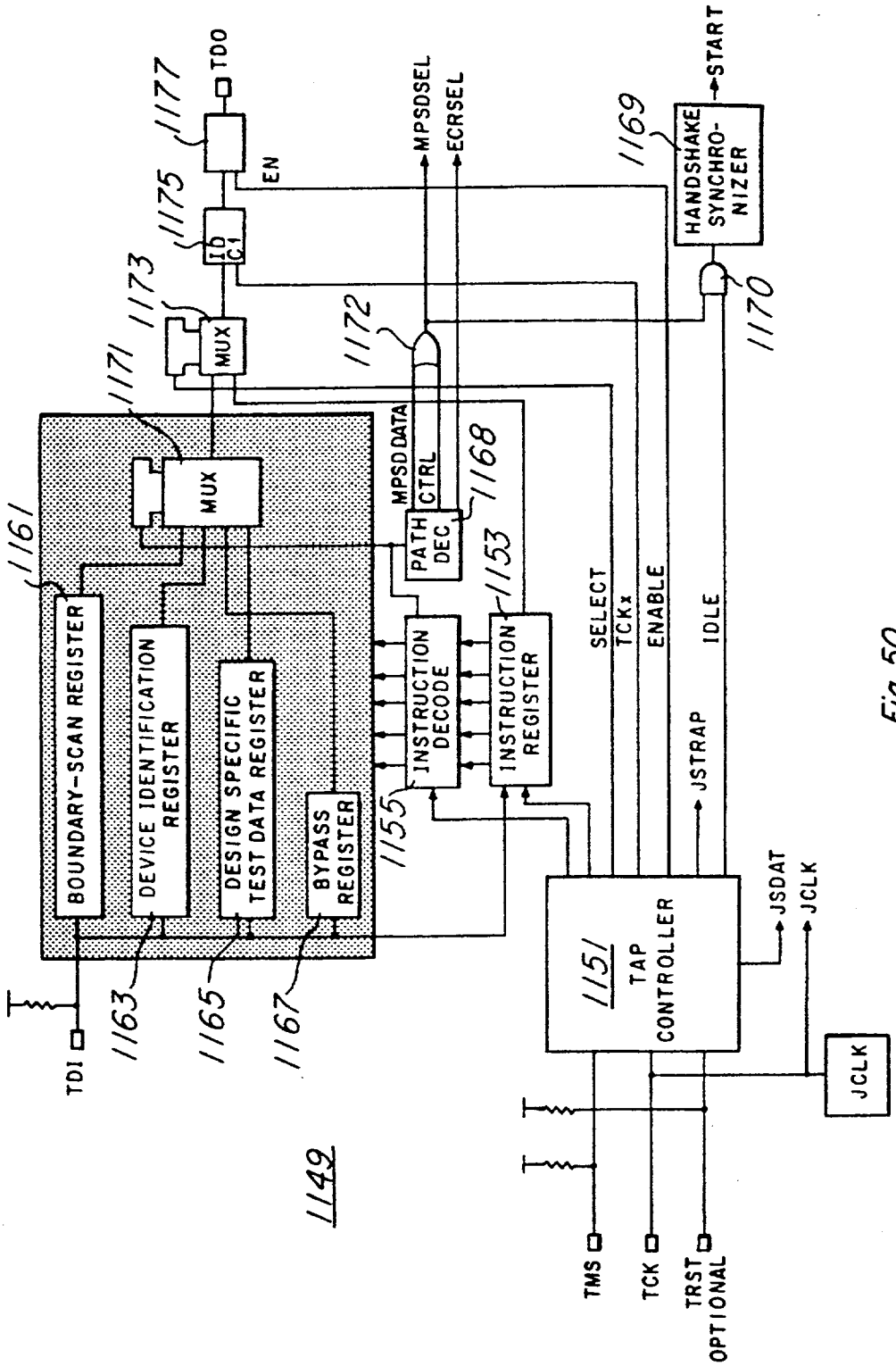

TDO
1177
EN
1175
ID
C1
1173
MUX
1171
MUX
1161
BOUNDARY-SCAN REGISTER
DEVICE IDENTIFICATION REGISTER
DESIGN SPECIFIC TEST DATA REGISTER
BYPASS REGISTER
TDI
1163
1165
1167
1149
MPSDSEL
ECRSEL
1172
MPSD DATA
CTRL
PATH DEC
1168
1153
INSTRUCTION DECODE
INSTRUCTION REGISTER
1155
SELECT
TCKx
ENABLE
IDLE
JSTRAP
1151
TAP CONTROLLER
JSDAT
JCLK
JCLK
TMS
TCK
TRST OPTIONAL
1169
HANDSHAKE SYNCHRO-NIZER
START
1170

Fig. 50

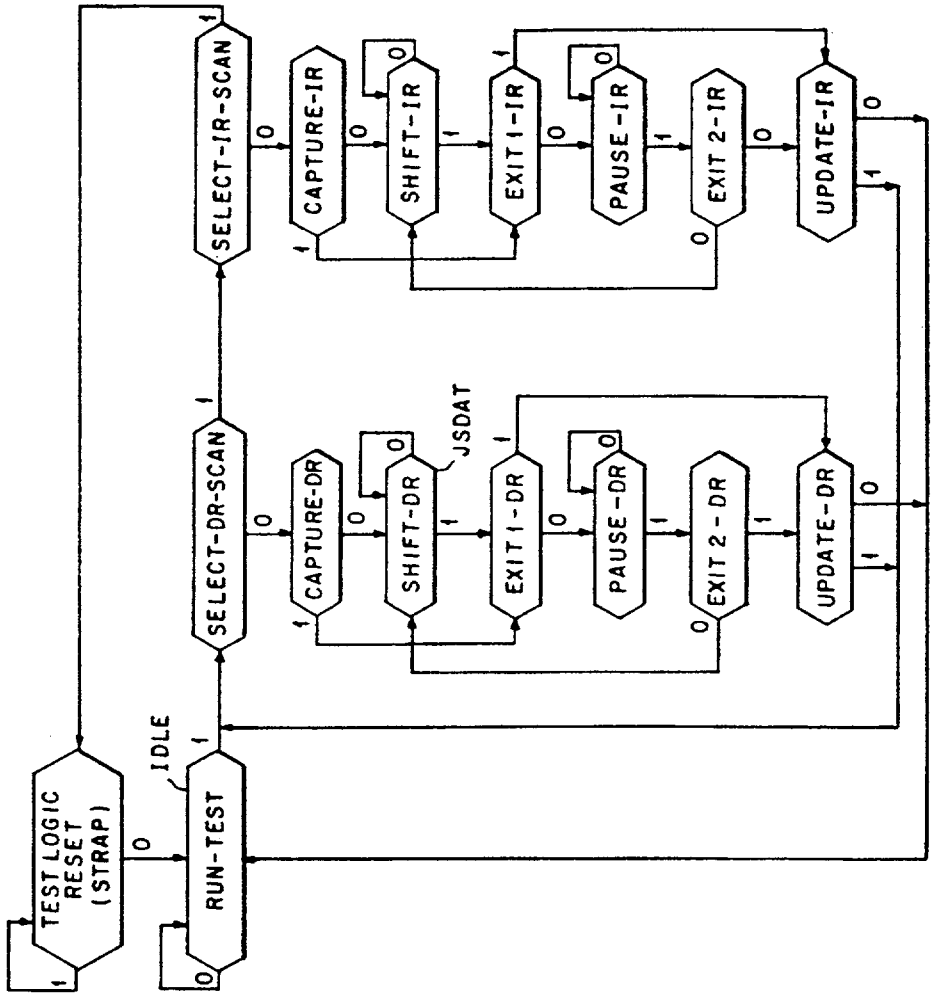

TEST LOGIC RESET (STRAP)
RUN-TEST
IDLE
SELECT-DR-SCAN
CAPTURE-DR
SHIFT-DR
JSDAT
EXIT 1-DR
PAUSE-DR
EXIT 2-DR
UPDATE-DR
SELECT-IR-SCAN
CAPTURE-IR
SHIFT-IR
EXIT 1-IR
PAUSE-IR
EXIT 2-IR
UPDATE-IR

Fig.50a

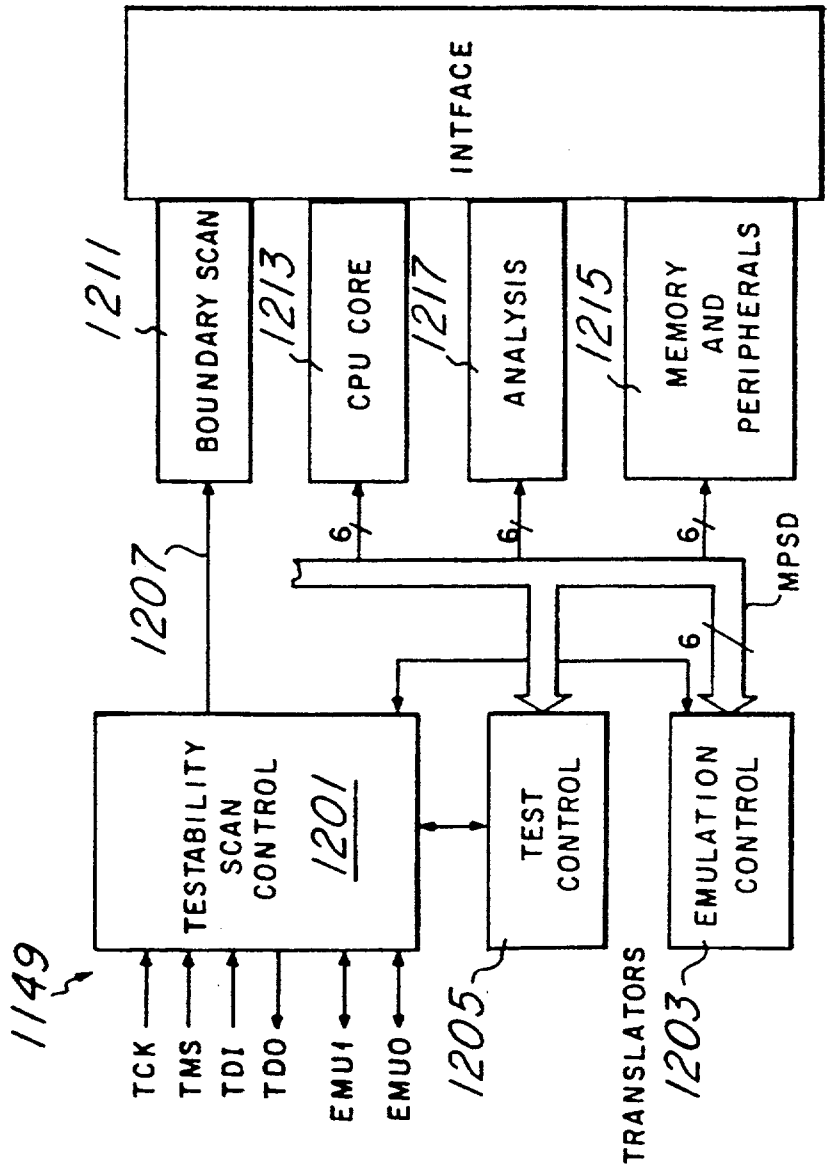

INTFACE
BOUNDARY SCAN
1211
CPU CORE
1213
ANALYSIS
1217
MEMORY AND PERIPHERALS
1215
6
6
6
MPSD
1207
TESTABILITY SCAN CONTROL
1201
TEST CONTROL
EMULATION CONTROL
6
1149
TCK
TMS
TDI
TDO
EMU1
EMU0
1205
TRANSLATORS
1203

Fig. 51

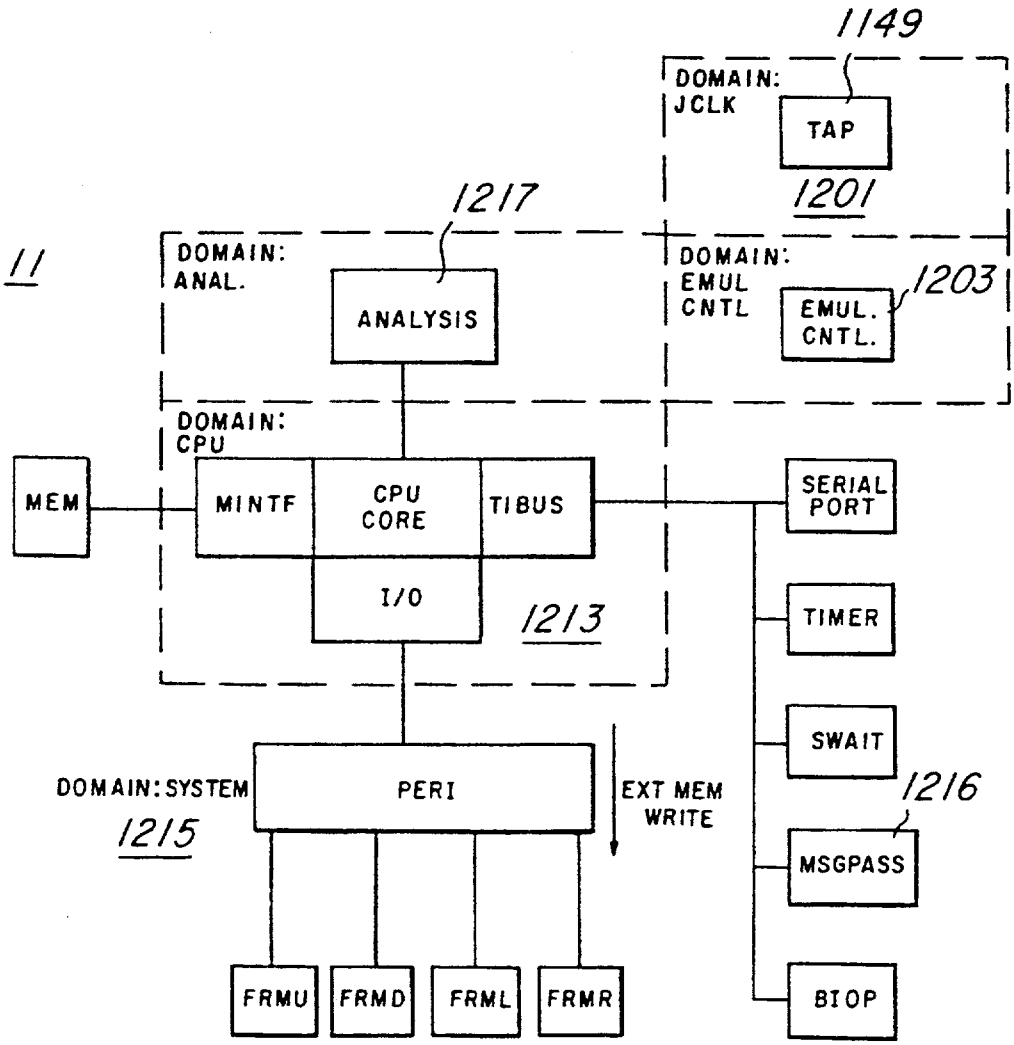

1149
DOMAIN:
JCLK
TAP
1201
1217
11
DOMAIN:
ANAL.
ANALYSIS
DOMAIN:
EMUL
CNTL
EMUL.
CNTL.
1203
DOMAIN:
CPU
MEM
MINTF
CPU
CORE
TIBUS
I/O
1213
SERIAL
PORT
TIMER
SWAIT
1216
MSGPASS
BIOP
DOMAIN: SYSTEM
1215
PERI
EXT MEM
WRITE
FRMU
FRMD
FRML
FRMR

Fig. 52

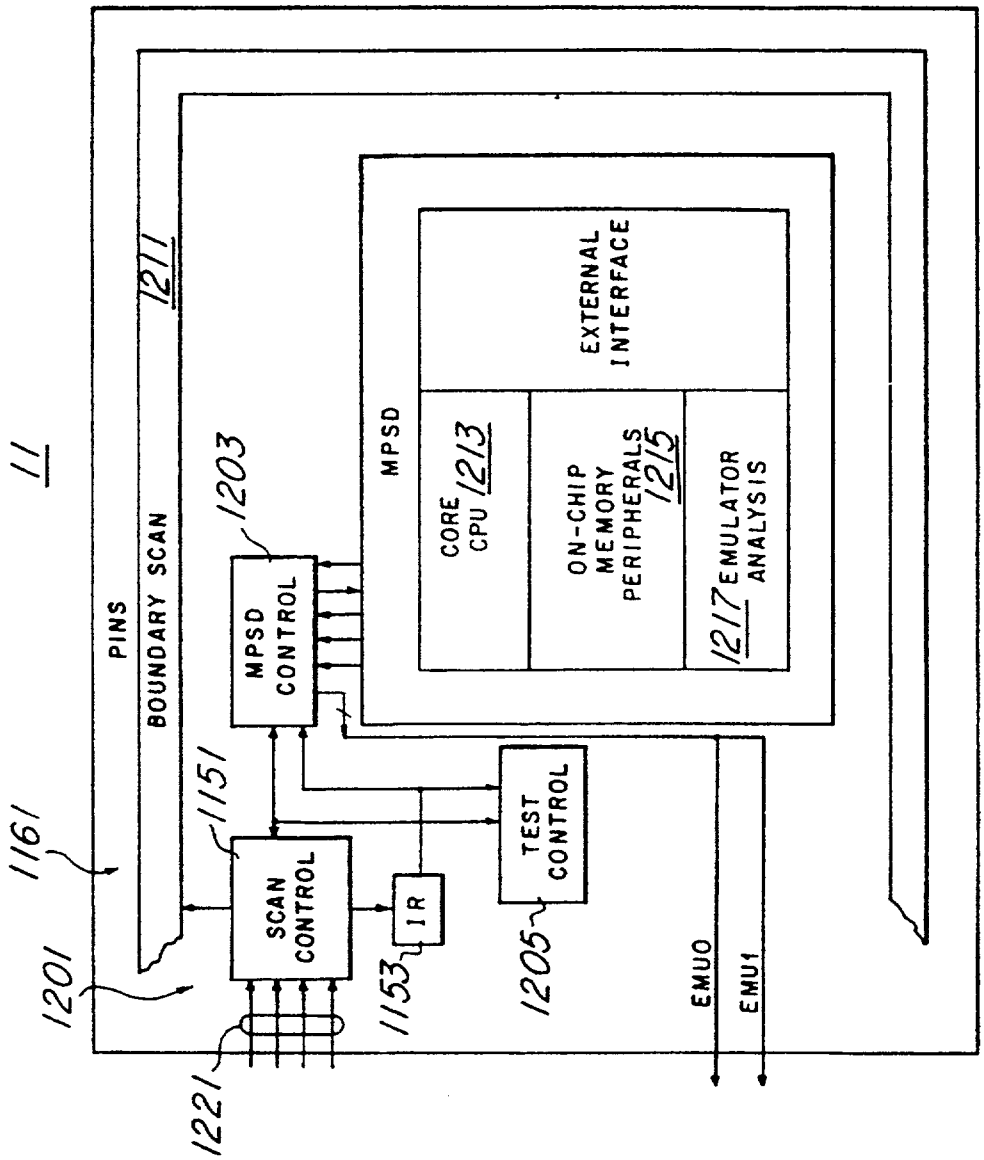

11
PINS
BOUNDARY SCAN
1211
1161
1201
1221
1151
SCAN CONTROL
1153
IR
1203
MPSD CONTROL
MPSD
CORE CPU 1213
ON-CHIP MEMORY PERIPHERALS 1215
EXTERNAL INTERFACE
1217 EMULATOR ANALYSIS
TEST CONTROL
1205
EMU0
EMU1

Fig. 53

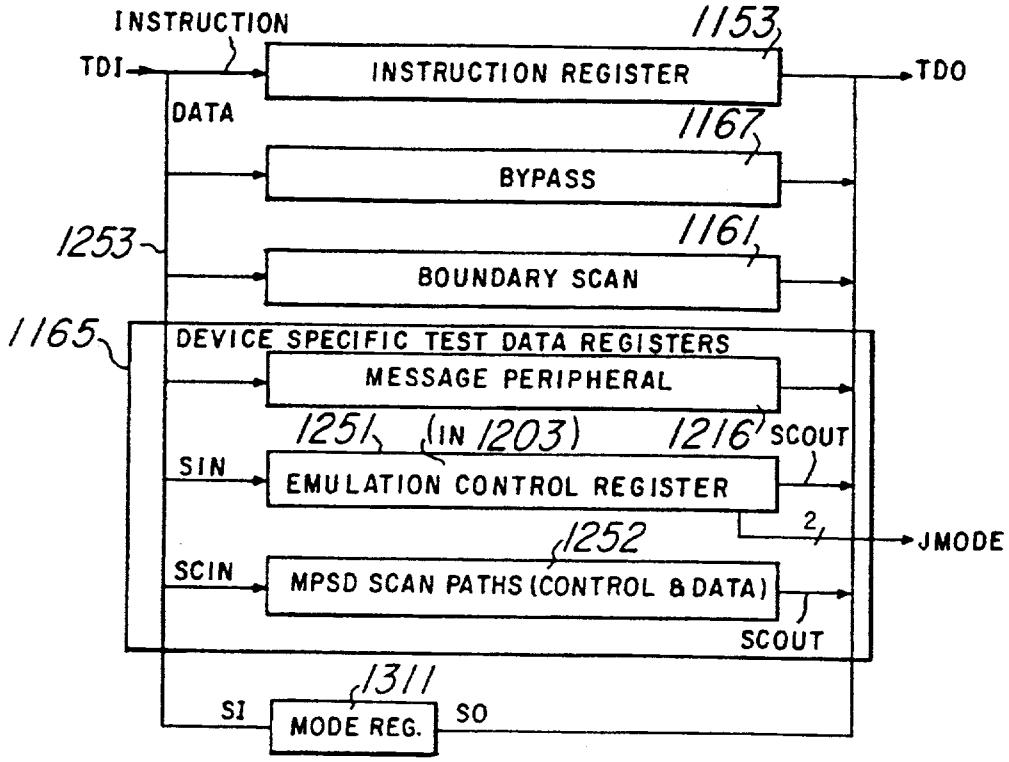

INSTRUCTION
TDI
1153
INSTRUCTION REGISTER
TDO
DATA
1167
BYPASS
1253
1161
BOUNDARY SCAN
1165
DEVICE SPECIFIC TEST DATA REGISTERS
MESSAGE PERIPHERAL
1251
(IN 1203)
1216
SCOUT
SIN
EMULATION CONTROL REGISTER
2
JMODE
1252
SCIN
MPSD SCAN PATHS (CONTROL & DATA)
SCOUT
1311
SI
MODE REG.
SO

Fig.54

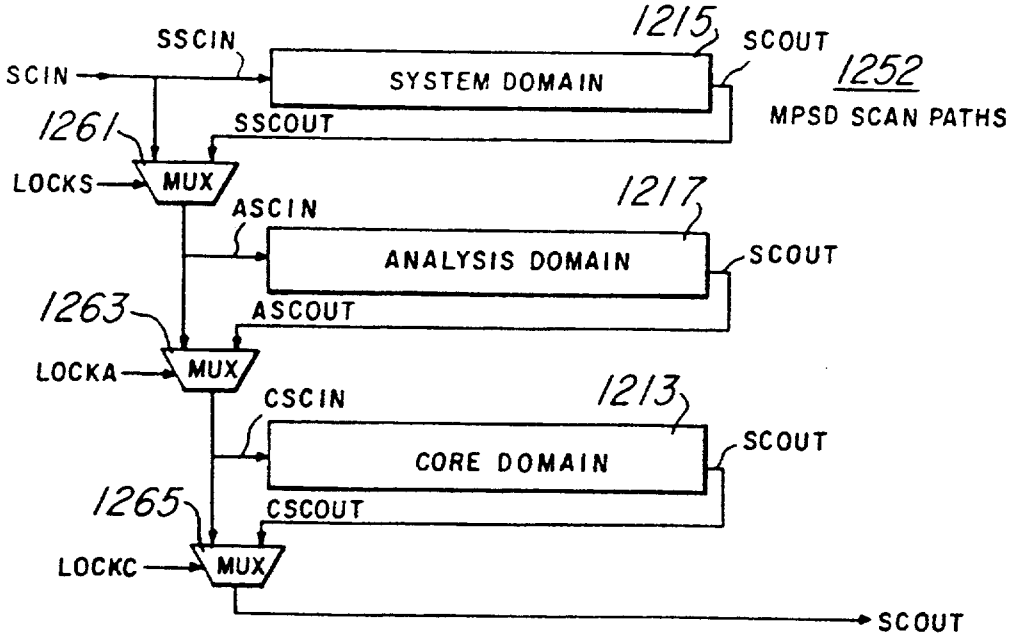

SSCIN
1215
SCOUT
SCIN
SYSTEM DOMAIN
1252
MPSD SCAN PATHS
1261
SSCOUT
LOCKS
MUX
ASCIN
1217
ANALYSIS DOMAIN
SCOUT
1263
ASCOUT
LOCKA
MUX
CSCIN
1213
CORE DOMAIN
SCOUT
1265
CSCOUT
LOCKC
MUX
SCOUT

Fig.55

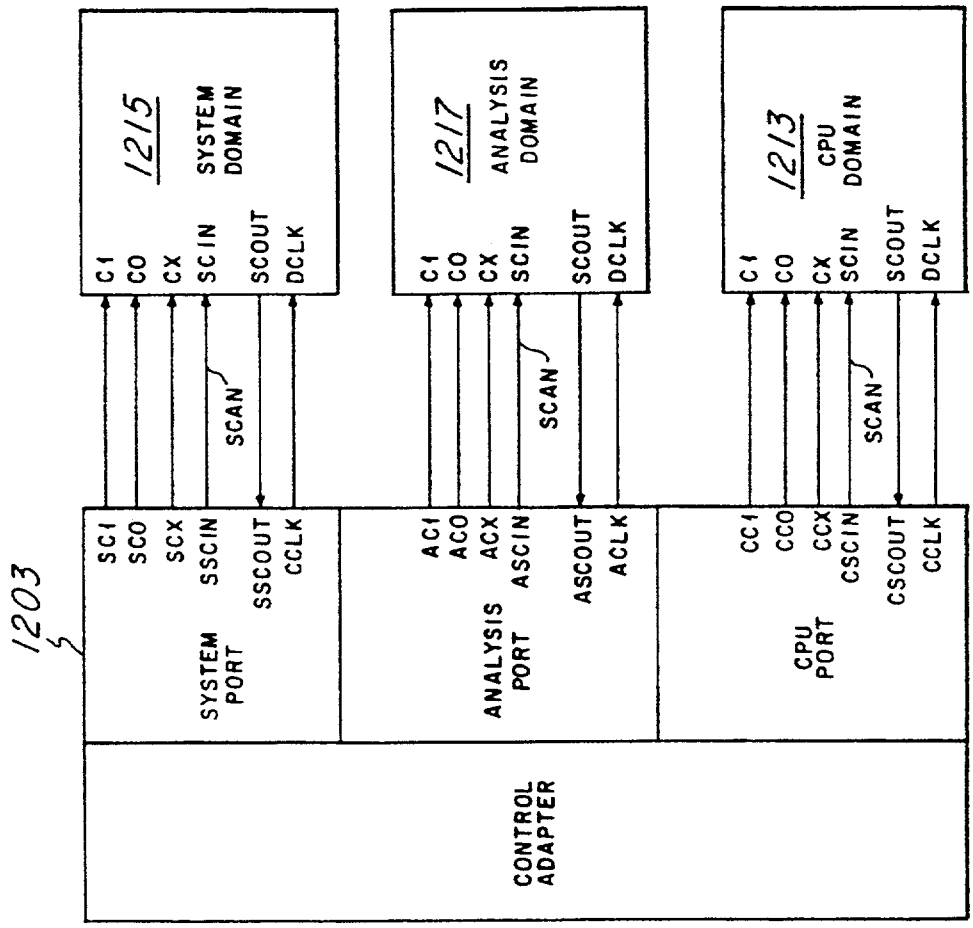

1203
SYSTEM PORT
SC1
SCO
SCX
SSCIN
SSCOUT
CCLK
ANALYSIS PORT
AC1
ACO
ACX
ASCIN
ASCOUT
ACLK
CPU PORT
CC1
CCO
CCX
CSCIN
CSCOUT
CCLK
CONTROL ADAPTER
1215
SYSTEM DOMAIN
C1
CO
CX
SCIN
SCOUT
DCLK
SCAN
1217
ANALYSIS DOMAIN
C1
CO
CX
SCIN
SCOUT
DCLK
SCAN
1213
CPU DOMAIN
C1
CO
CX
SCIN
SCOUT
DCLK
SCAN

Fig. 56

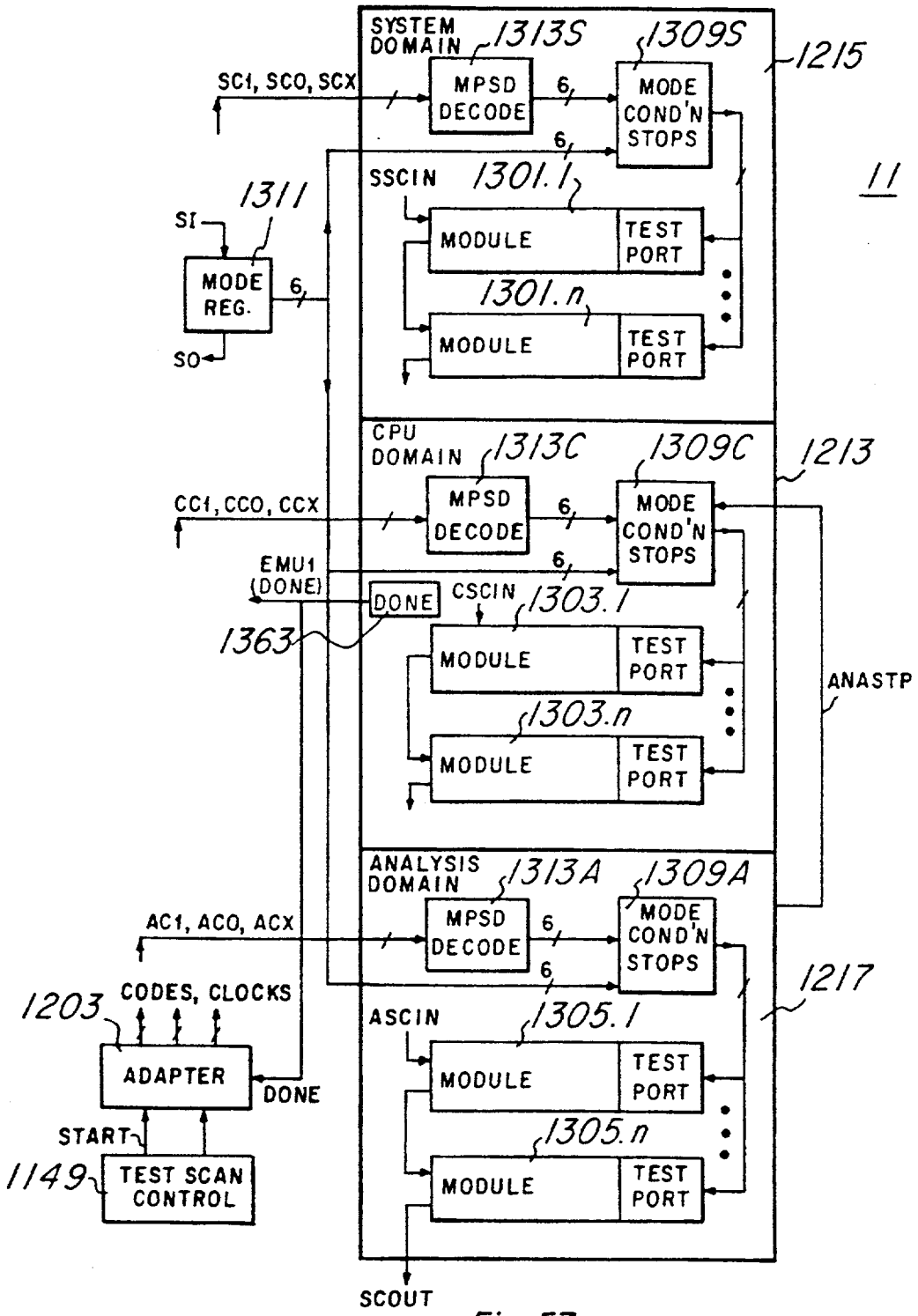

SYSTEM DOMAIN
1313S
1309S
1215
SC1, SCO, SCX
MPSD DECODE
6
MODE COND'N STOPS
6
SSCIN
1301.1
11
1311
SI
MODE REG.
6
SO
MODULE
TEST PORT
1301.n
MODULE
TEST PORT
CPU DOMAIN
1313C
1309C
1213
CC1, CCO, CCX
MPSD DECODE
6
MODE COND'N STOPS
6
EMU1 (DONE)
DONE
CSCIN
1303.1
1363
MODULE
TEST PORT
ANASTP
1303.n
MODULE
TEST PORT
ANALYSIS DOMAIN
1313A
1309A
AC1, ACO, ACX
MPSD DECODE
6
MODE COND'N STOPS
6
1217
1203
CODES, CLOCKS
ASCIN
1305.1
ADAPTER
DONE
MODULE
TEST PORT
START
1305.n
1149
TEST SCAN CONTROL
MODULE
TEST PORT
SCOUT

Fig. 57

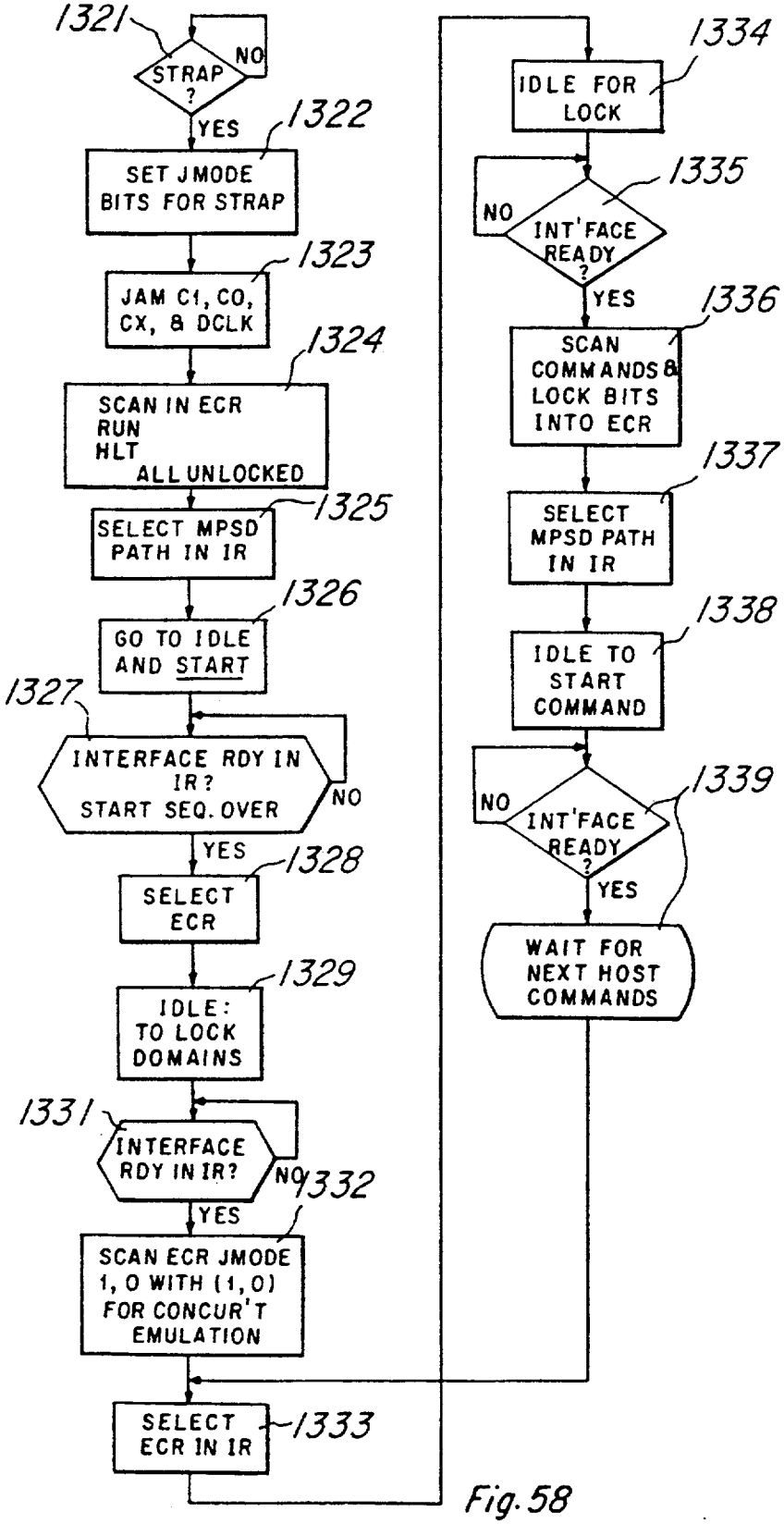

1321
STRAP ?
NO
YES
1322
SET JMODE BITS FOR STRAP
1323
JAM C1, CO, CX, & DCLK
1324
SCAN IN ECR
RUN
HLT
ALL UNLOCKED
1325
SELECT MPSD PATH IN IR
1326
GO TO IDLE AND START
1327
INTERFACE RDY IN IR?
START SEQ. OVER
NO
YES
1328
SELECT ECR
1329
IDLE: TO LOCK DOMAINS
1331
INTERFACE RDY IN IR?
NO
1332
YES
SCAN ECR JMODE 1, 0 WITH (1, 0) FOR CONCUR'T EMULATION
SELECT ECR IN IR
1333
1334
IDLE FOR LOCK
1335
NO
INT'FACE READY ?
YES
1336
SCAN COMMANDS & LOCK BITS INTO ECR
1337
SELECT MPSD PATH IN IR
1338
IDLE TO START COMMAND
1339
NO
INT'FACE READY ?
YES
WAIT FOR NEXT HOST COMMANDS

Fig. 58

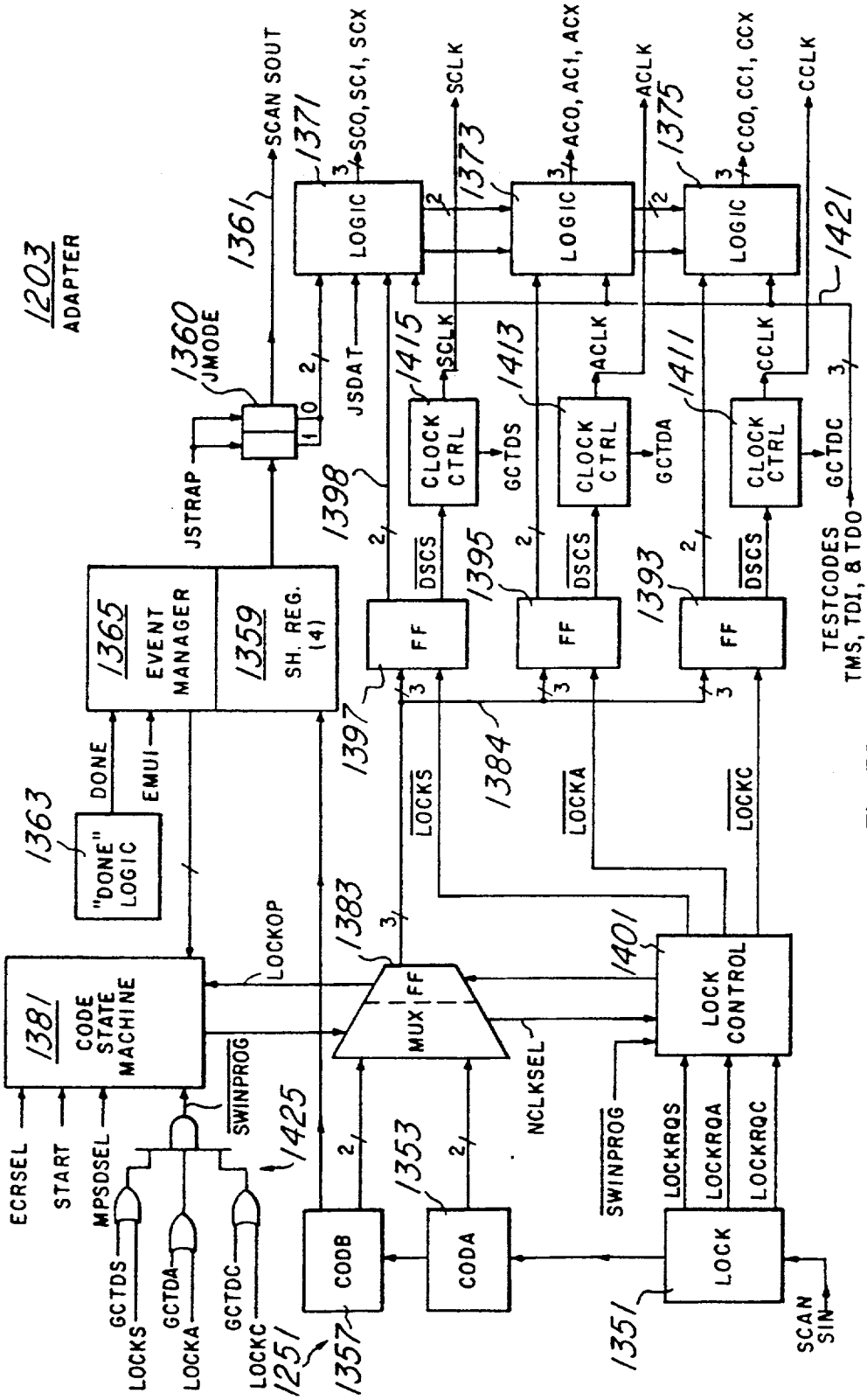

1203
ADAPTER
1361
SCAN SOUT
1371
SCO, SC1, SCX
LOGIC
1373
SCLK
ACO, AC1, ACX
ACLK
1375
CCO, CC1, CCX
CCLK
1421
1360
JMODE
JSTRAP
JSDAT
1415
1413
1411
CLOCK CTRL
GCTDS
GCTDA
GCTDC
1398
1365
EVENT MANAGER
1359
SH. REG. (4)
DSCS
1395
1393
FF
TESTCODES TMS, TDI, & TDO
1397
1384
DONE
EMUI
1363
"DONE" LOGIC
LOCKS
LOCKA
LOCKC
1383
1401
1381
CODE STATE MACHINE
LOCKOP
MUX FF
LOCK CONTROL
NCLKSEL
SWINPROG
1425
ECRSEL
START
MPSDSEL
1353
LOCKRQS
LOCKRQA
LOCKRQC
CODB
CODA
LOCK
1251
1357
1351
SCAN SIN

Fig. 59

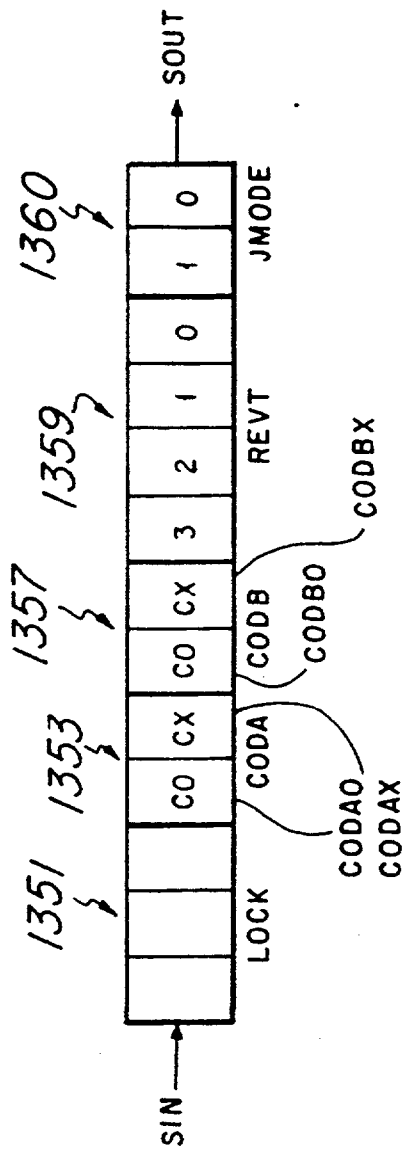

SIN
SOUT
1351
1353
1357
1359
1360
LOCK
CO
CX
CO
CX
CODA
CODB
3
2
1
0
1
0
REVT
JMODE
CODAO
CODAX
CODBO
CODBX

Fig. 59a

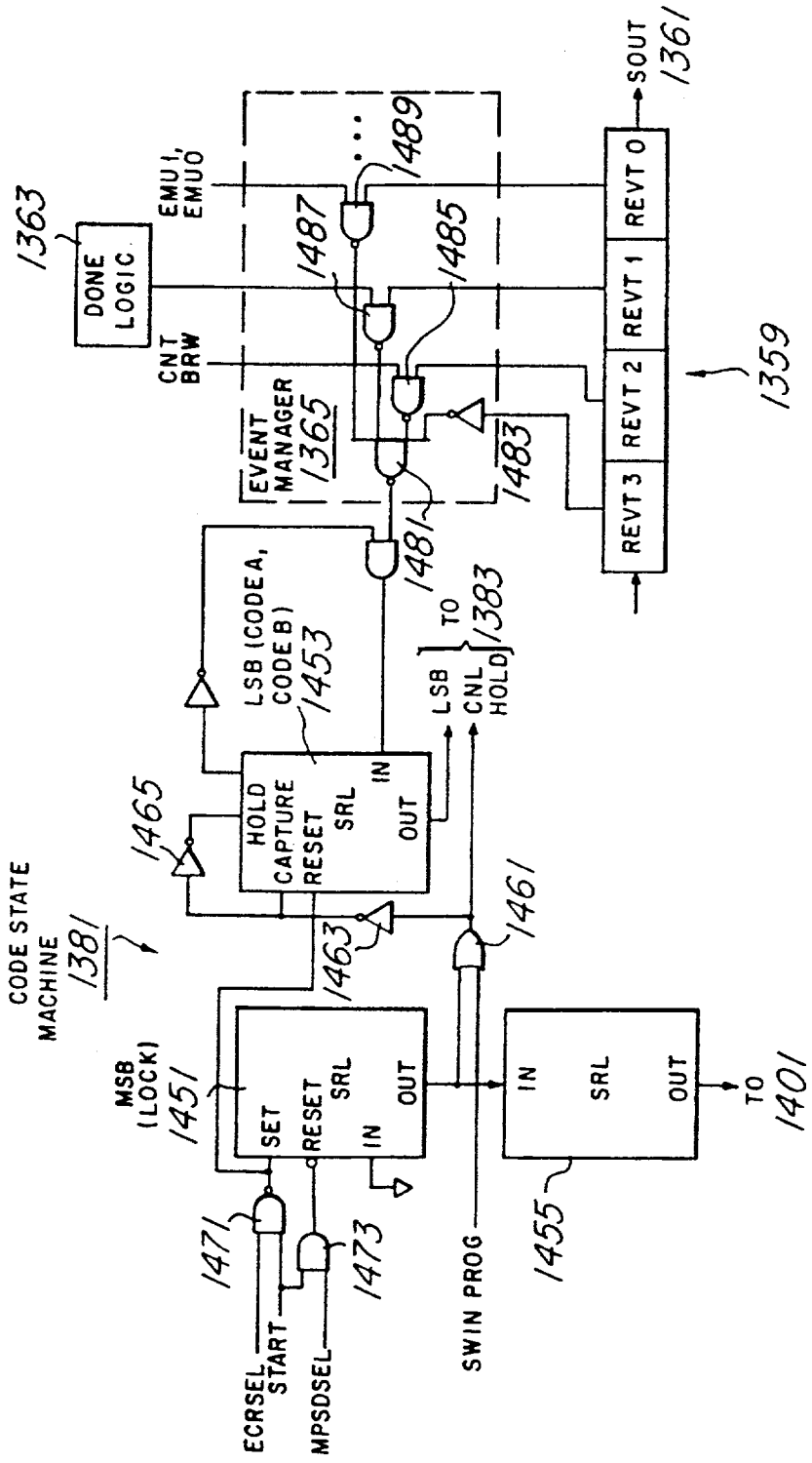

1363
DONE LOGIC
EMU1, EMU0
CNT BRW
EVENT MANAGER 1365
1487
1489
1485
1483
1481
REVT 0
REVT 1
REVT 2
REVT 3
SOUT
1361
1359
LSB (CODE A, CODE B)
1453
TO 1383
LSB
CNL HOLD
HOLD
CAPTURE
RESET
SRL
IN
OUT
1465
1461
1463
CODE STATE MACHINE 1381
MSB (LOCK) 1451
SET
RESET
SRL
IN
OUT
1471
1473
ECRSEL START
MPSDSEL
SWIN PROG
1455
TO 1401

Fig. 60

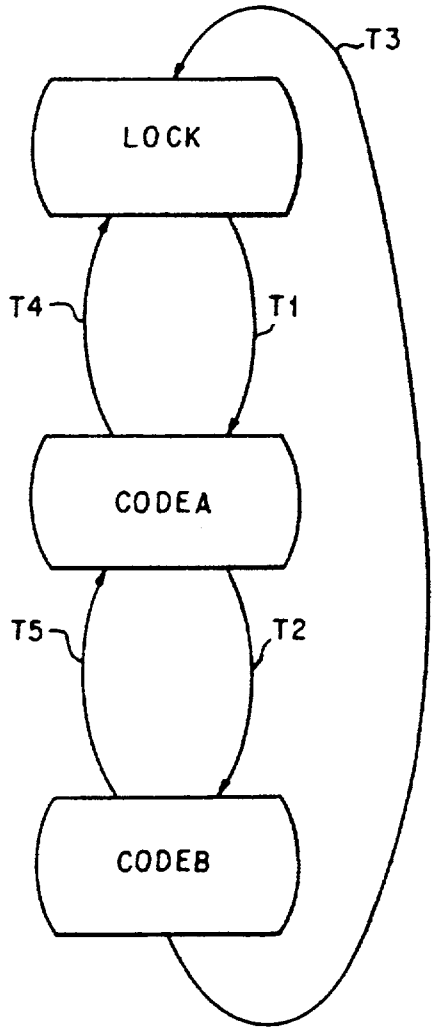


Fig. 61

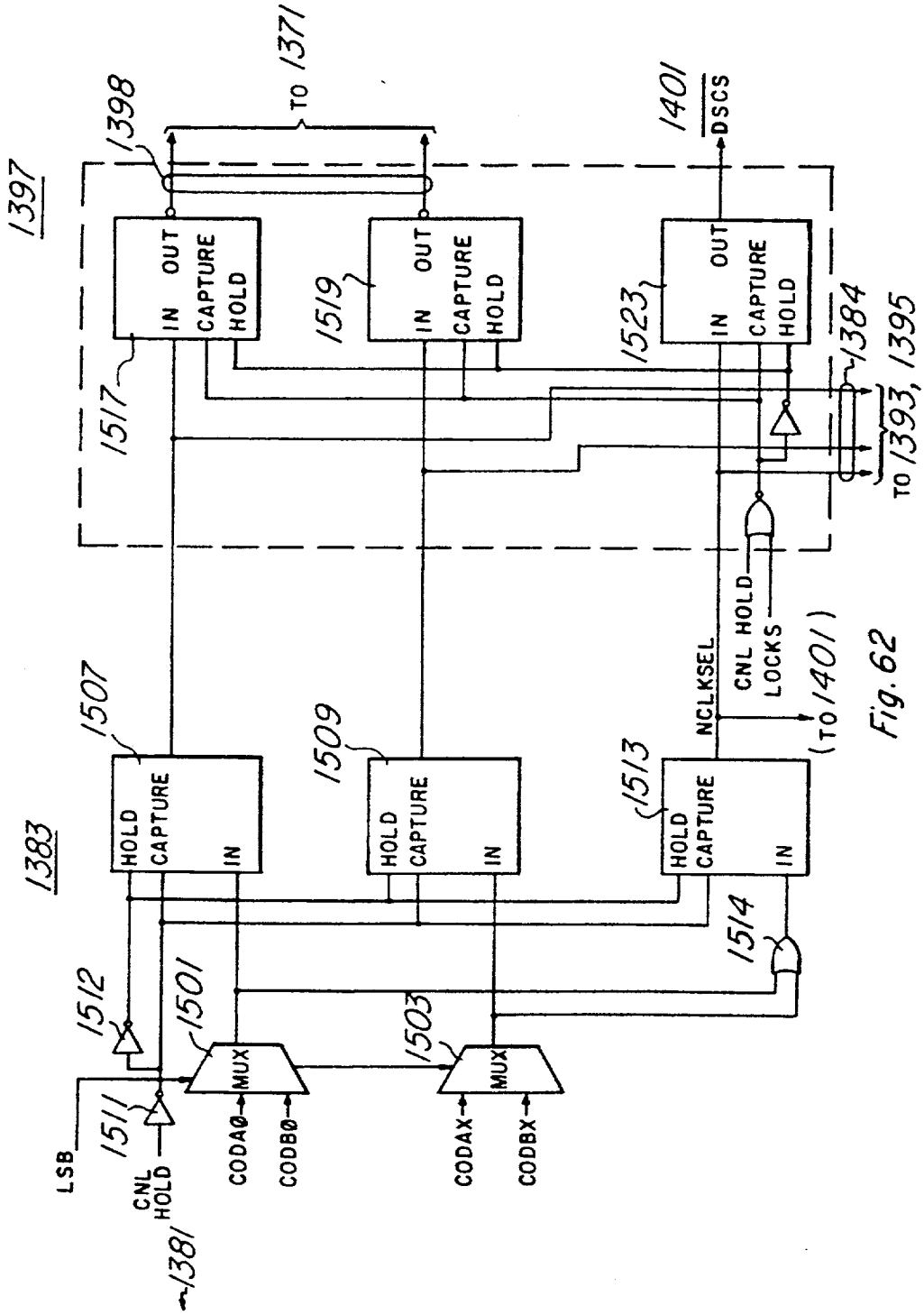

1397
1398
TO 1371
1401
DSCS
IN OUT
CAPTURE
HOLD
1517
1519
1523
1384
1395
TO 1393
CNL HOLD
LOCKS
NCLKSEL
(TO 1401)
1383
1507
1509
1513
HOLD
CAPTURE
IN
1514
1512
1501
1503
MUX
1511
LSB
CNL HOLD
1381
CODAØ
CODBØ
CODAX
CODBX

Fig. 62

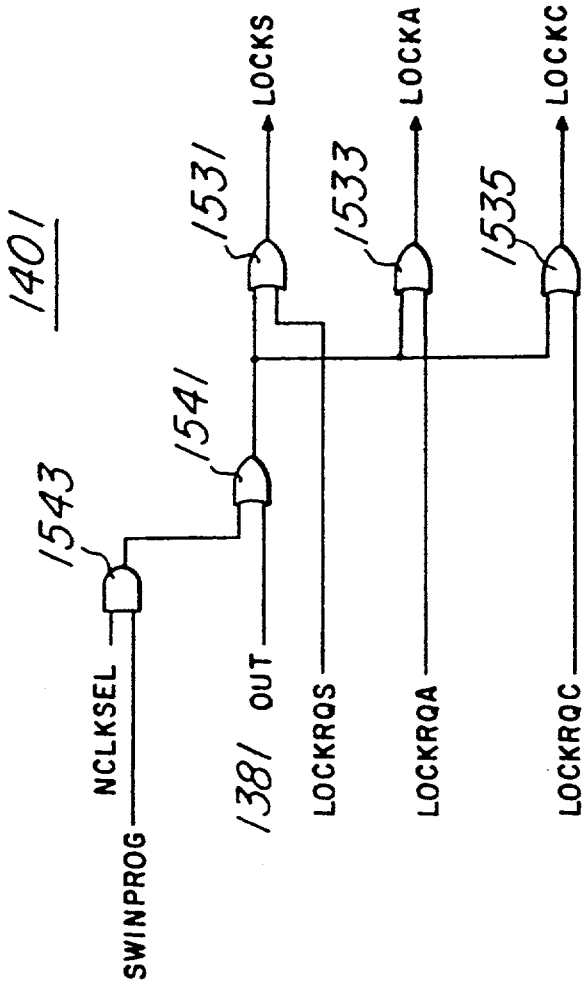

1401
1543
1541
1531
1533
1535
SWINPROG
NCLKSEL
1381
OUT
LOCKRQS
LOCKRQA
LOCKRQC
LOCKS
LOCKA
LOCKC

Fig. 63

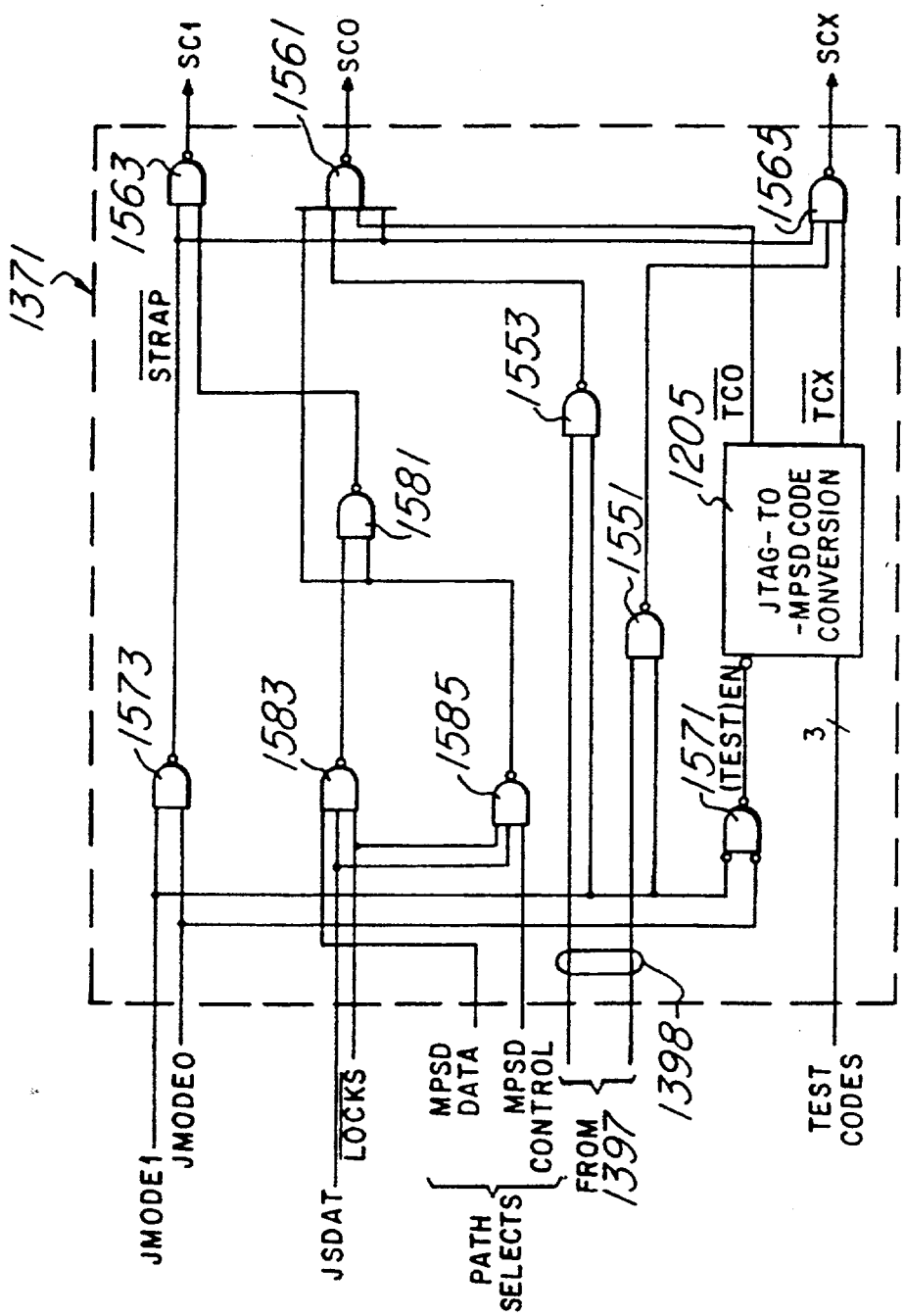

SC1
SCO
SCX
1371
1563
1561
1565
STRAP
1553
1581
1205
TCO
TCX
JTAG- TO
-MPSD CODE
CONVERSION
1551
1573
1583
1585
1571
(TEST)EN
3
1398
JMODE1
JMODEO
JSDAT
LOCKS
MPSD
DATA
MPSD
CONTROL
FROM
1397
PATH
SELECTS
TEST
CODES

Fig. 64

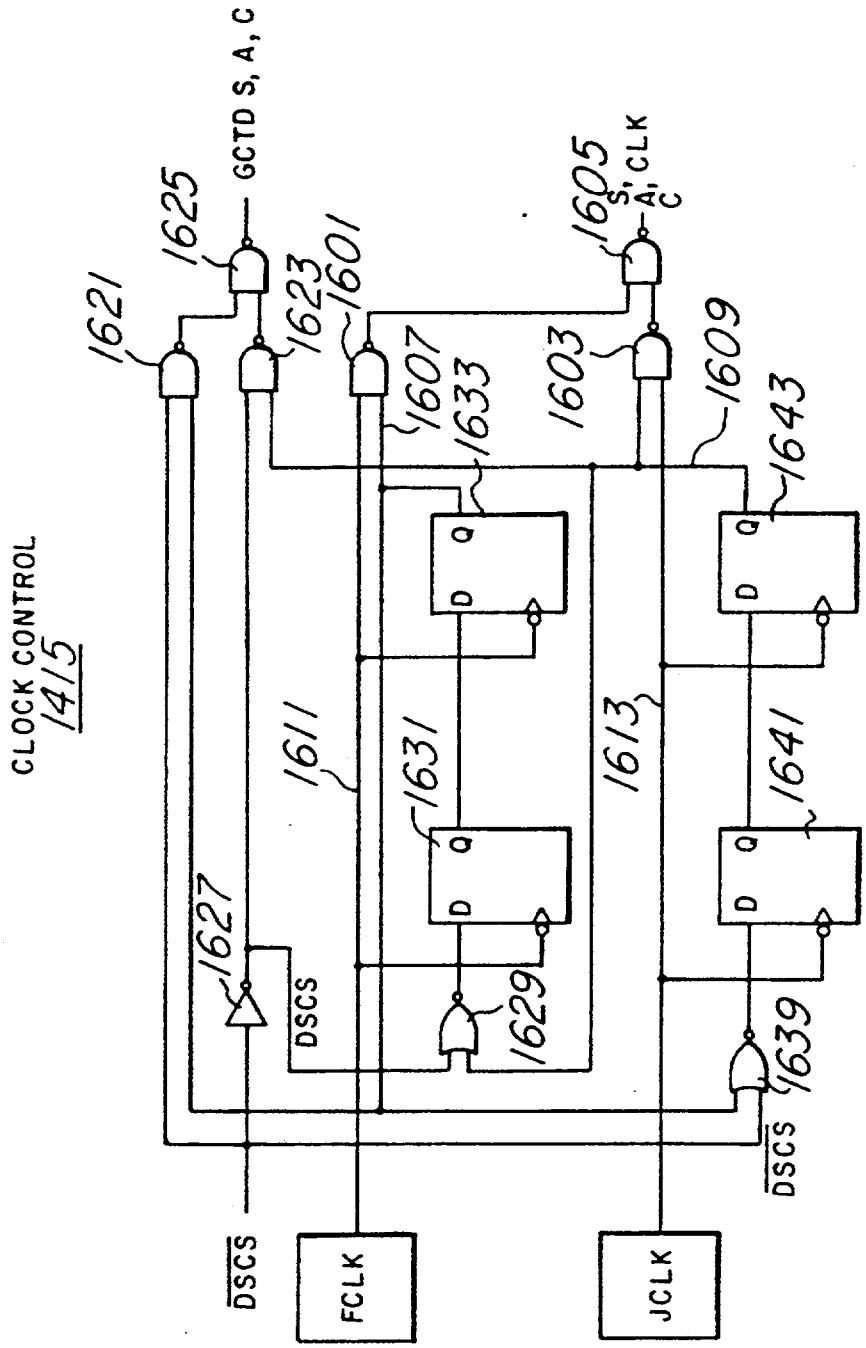

CLOCK CONTROL
1415
GCTD S, A, C
1625
1621
1623
1601
1605
S, A, C
CLK
1607
1633
1603
1609
1643
1611
1631
1613
1641
1627
DSCS
1629
1639
$\overline{DSCS}$
FCLK
JCLK
D
Q

Fig. 65

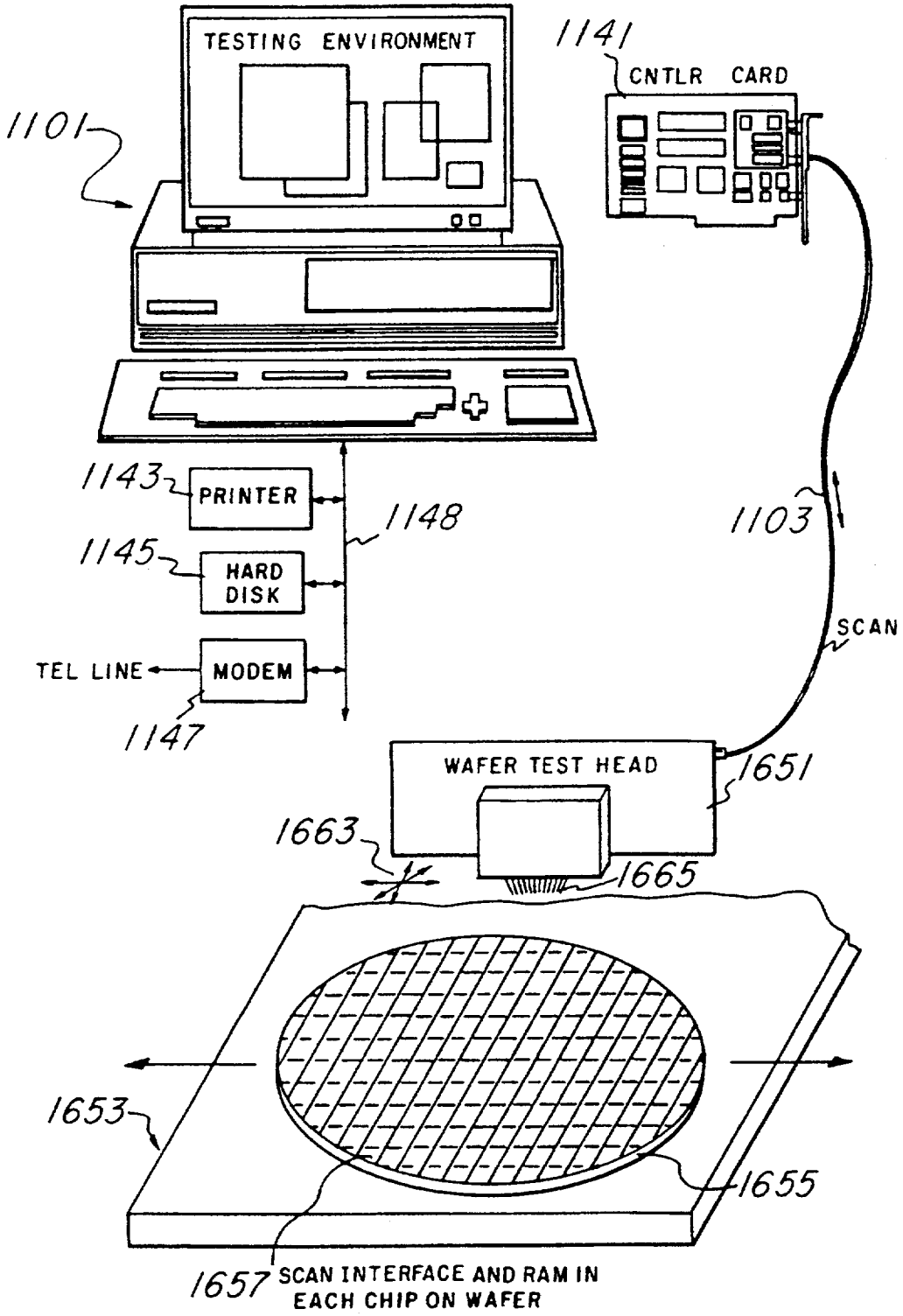

TESTING ENVIRONMENT
1101
1141
CNTLR CARD
1143
PRINTER
1148
1145
HARD DISK
TEL LINE
MODEM
1147
1103
SCAN
WAFER TEST HEAD
1651
1663
1665
1653
1655
1657
SCAN INTERFACE AND RAM IN EACH CHIP ON WAFER

Fig 66

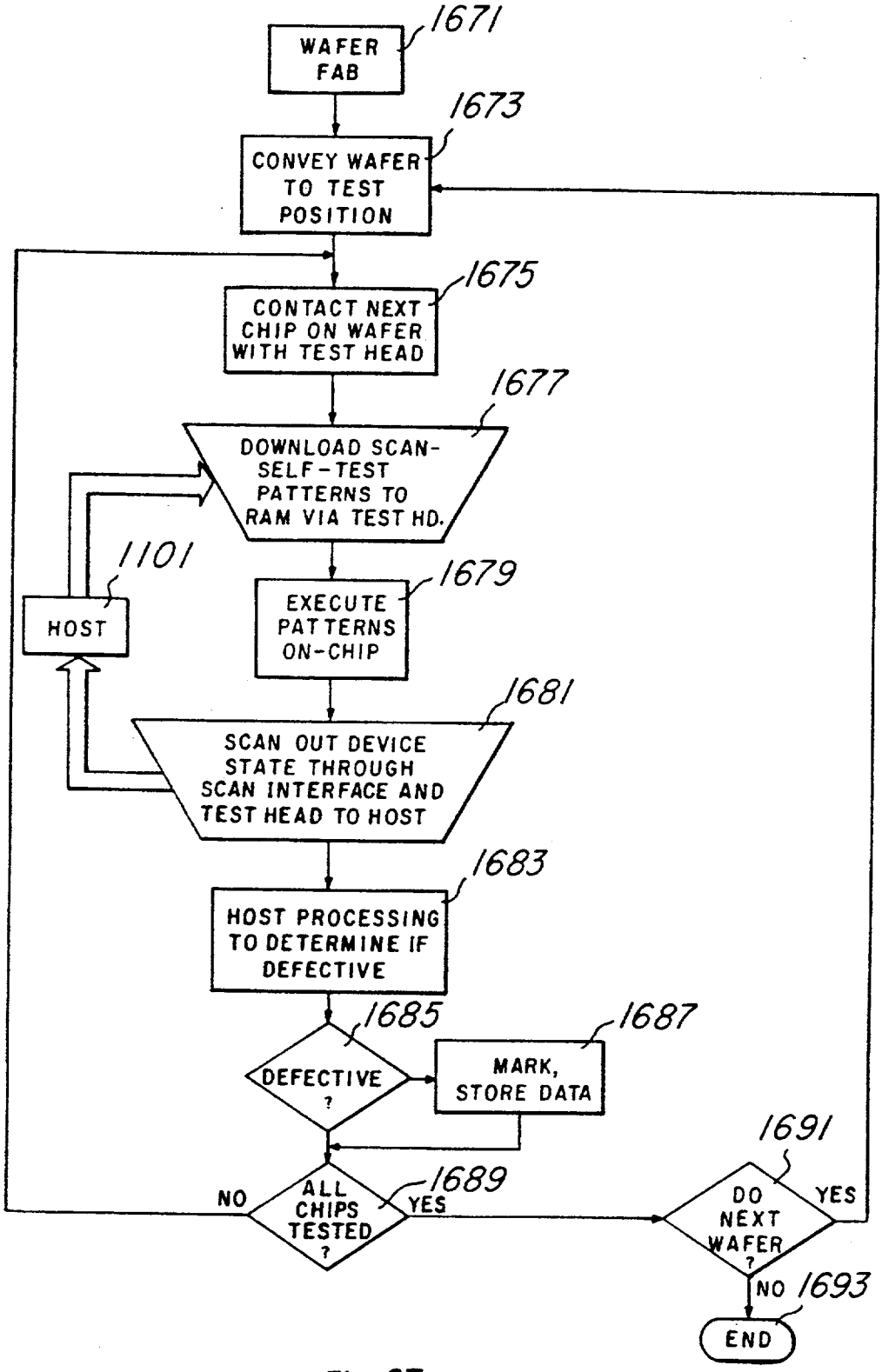

1671
WAFER FAB
1673
CONVEY WAFER TO TEST POSITION
1675
CONTACT NEXT CHIP ON WAFER WITH TEST HEAD
1677
DOWNLOAD SCAN-SELF-TEST PATTERNS TO RAM VIA TEST HD.
1101
HOST
1679
EXECUTE PATTERNS ON-CHIP
1681
SCAN OUT DEVICE STATE THROUGH SCAN INTERFACE AND TEST HEAD TO HOST
1683
HOST PROCESSING TO DETERMINE IF DEFECTIVE
1685
DEFECTIVE ?
1687
MARK, STORE DATA
1689
ALL CHIPS TESTED ?
NO
YES
1691
DO NEXT WAFER ?
YES
NO
1693
END

Fig. 67

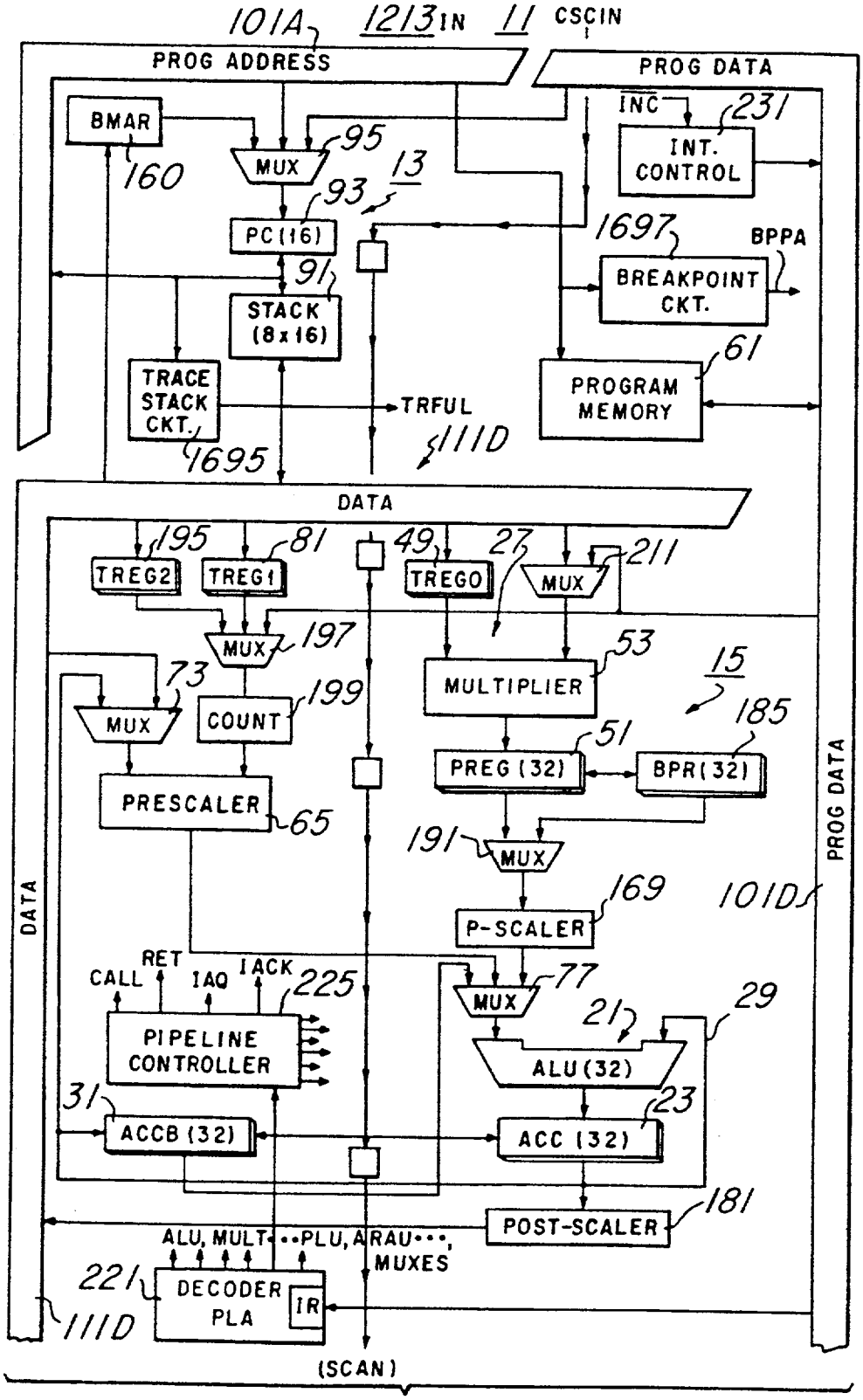

101A
1213 IN
11 CSCIN
PROG ADDRESS
PROG DATA
BMAR
160
MUX
95
13
93
PC (16)
91
STACK (8x16)
TRACE STACK CKT.
1695
TRFUL
111D
INC
231
INT. CONTROL
1697
BPPA
BREAKPOINT CKT.
61
PROGRAM MEMORY
DATA
195
TREG2
81
TREG1
49
TREGO
27
MUX
211
MUX
197
73
MUX
COUNT
199
PRESCALER
65
MULTIPLIER
53
15
185
PREG (32)
51
BPR (32)
191
MUX
169
P-SCALER
101D
PROG DATA
DATA
CALL
RET
IAQ
IACK
225
PIPELINE CONTROLLER
MUX
77
21
29
ALU (32)
31
ACCB (32)
ACC (32)
23
POST-SCALER
181
ALU, MULT•••PLU, ARAU•••, MUXES
221
DECODER PLA
IR
111D
(SCAN)
JOIN FIG.68B

Fig.68a

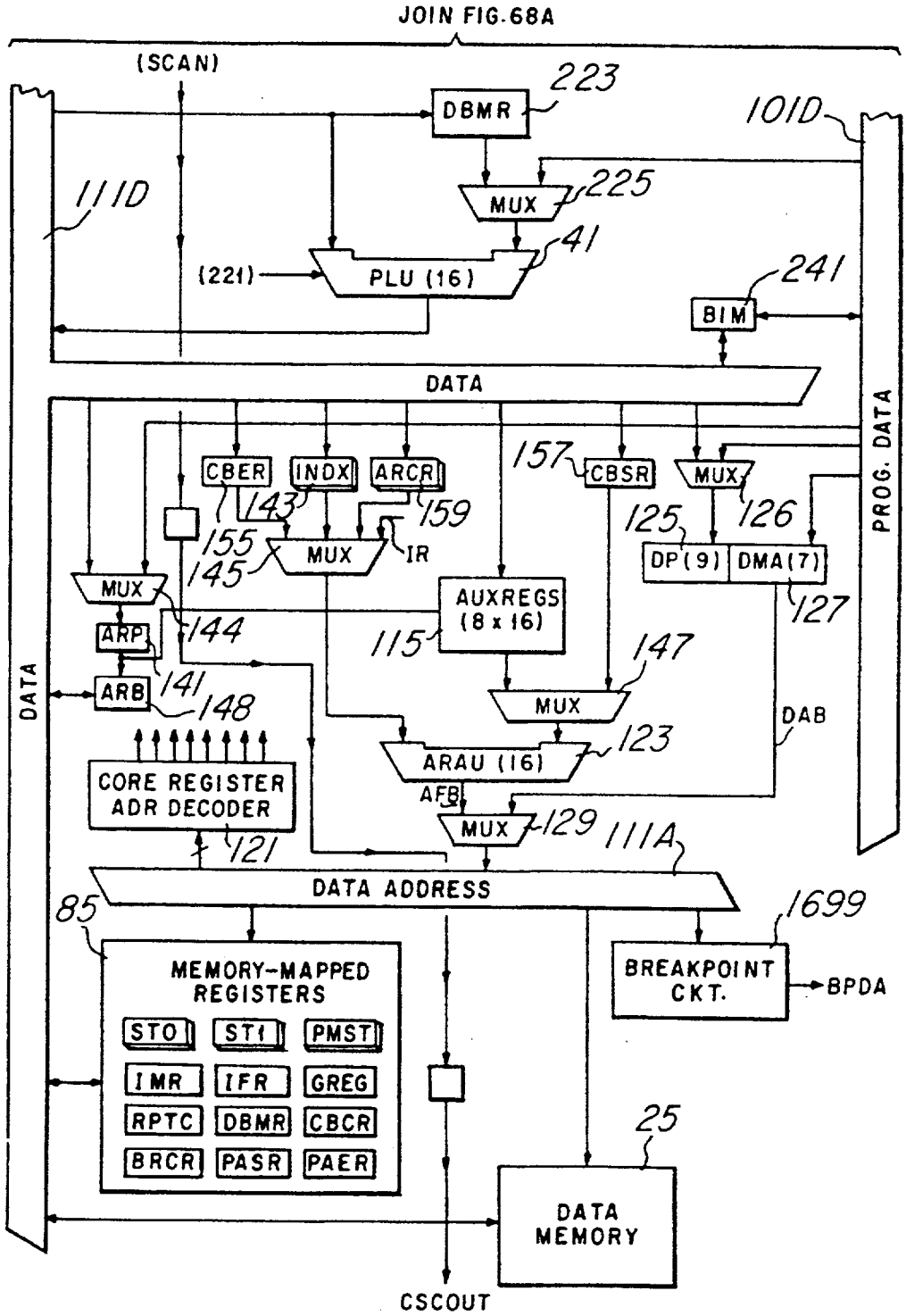

JOIN FIG.68A
(SCAN)
DBMR
223
101D
111D
MUX
225
41
(221)
PLU (16)
241
BIM
DATA
PROG. DATA
CBER
INDX
ARCR
143
159
155
145
MUX
IR
157
CBSR
MUX
125
126
DP(9)
DMA(7)
127
MUX
ARP
144
AUXREGS (8 x 16)
115
147
ARB
141
148
MUX
DAB
ARAU (16)
123
AFB
CORE REGISTER ADR DECODER
121
MUX
129
111A
DATA ADDRESS
85
1699
MEMORY-MAPPED REGISTERS
STO
ST1
PMST
IMR
IFR
GREG
RPTC
DBMR
CBCR
BRCR
PASR
PAER
BREAKPOINT CKT.
BPDA
25
DATA MEMORY
CSCOUT
DATA

Fig.68b

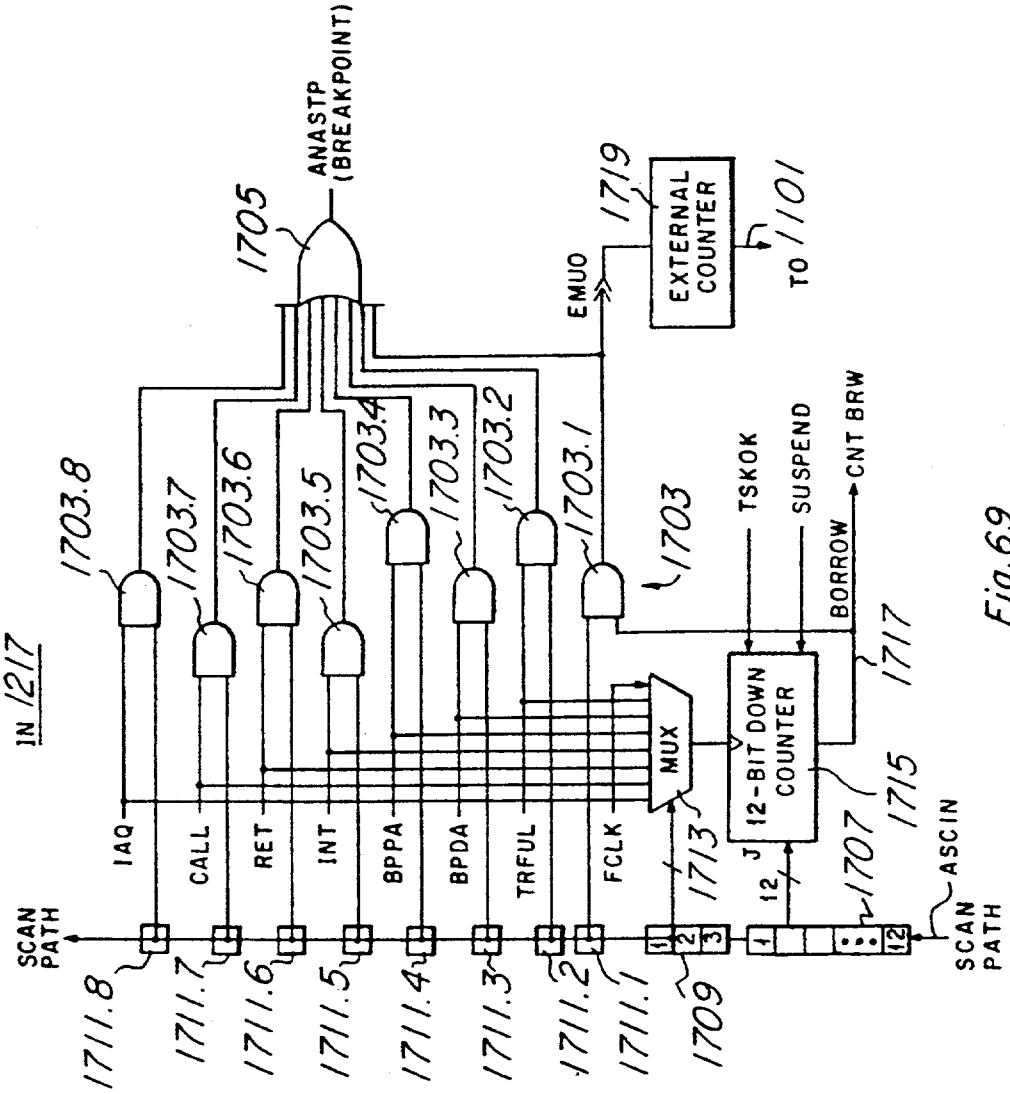

ANASTP (BREAKPOINT)
1705
EMUO
1719
EXTERNAL COUNTER
TO 1101
1703.8
1703.7
1703.6
1703.5
1703.4
1703.3
1703.2
1703.1
1703
TSKOK
SUSPEND
CNT BRW
BORROW
1717
IN 1217
MUX
12-BIT DOWN COUNTER
1715
1713
1707
ASCIN
IAQ
CALL
RET
INT
BPPA
BPDA
TRFUL
FCLK
SCAN PATH
1711.8
1711.7
1711.6
1711.5
1711.4
1711.3
1711.2
1711.1
1709
SCAN PATH

Fig. 69

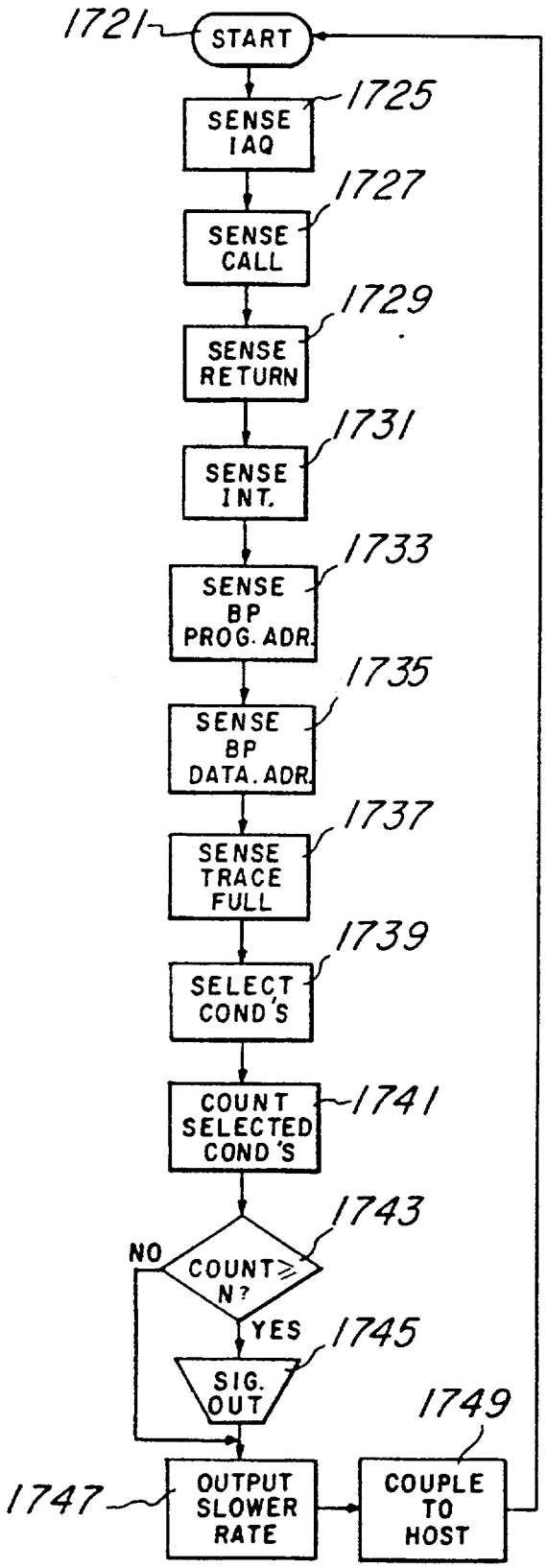
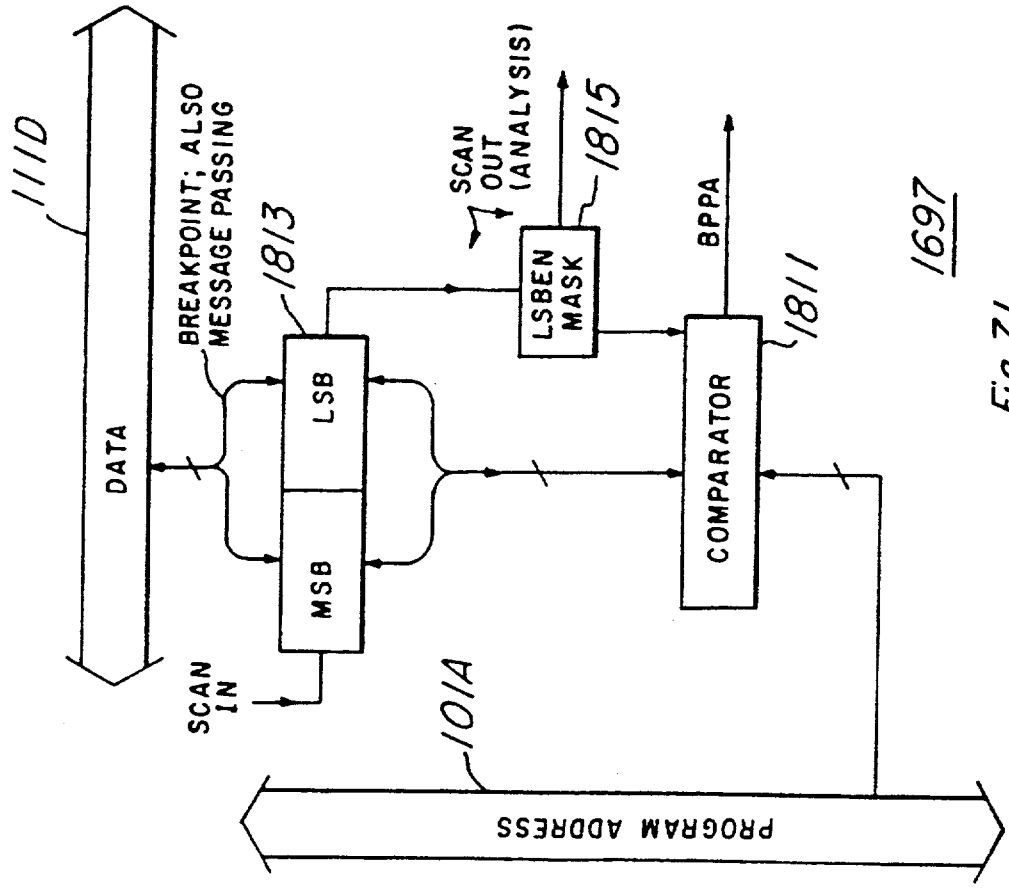

1721
START
1725
SENSE IAQ
1727
SENSE CALL
1729
SENSE RETURN
1731
SENSE INT.
1733
SENSE BP PROG. ADR.
1735
SENSE BP DATA. ADR.
1737
SENSE TRACE FULL
1739
SELECT COND'S
1741
COUNT SELECTED COND'S
1743
COUNT≥ N?
NO
YES
1745
SIG. OUT
1747
OUTPUT SLOWER RATE
1749
COUPLE TO HOST

Fig. 70


1110
DATA
BREAKPOINT; ALSO MESSAGE PASSING
1813
LSB
MSB
SCAN IN
SCAN OUT (ANALYSIS)
1815
LSBEN MASK
BPPA
1811
COMPARATOR
1014
PROGRAM ADDRESS

1697

Fig. 71

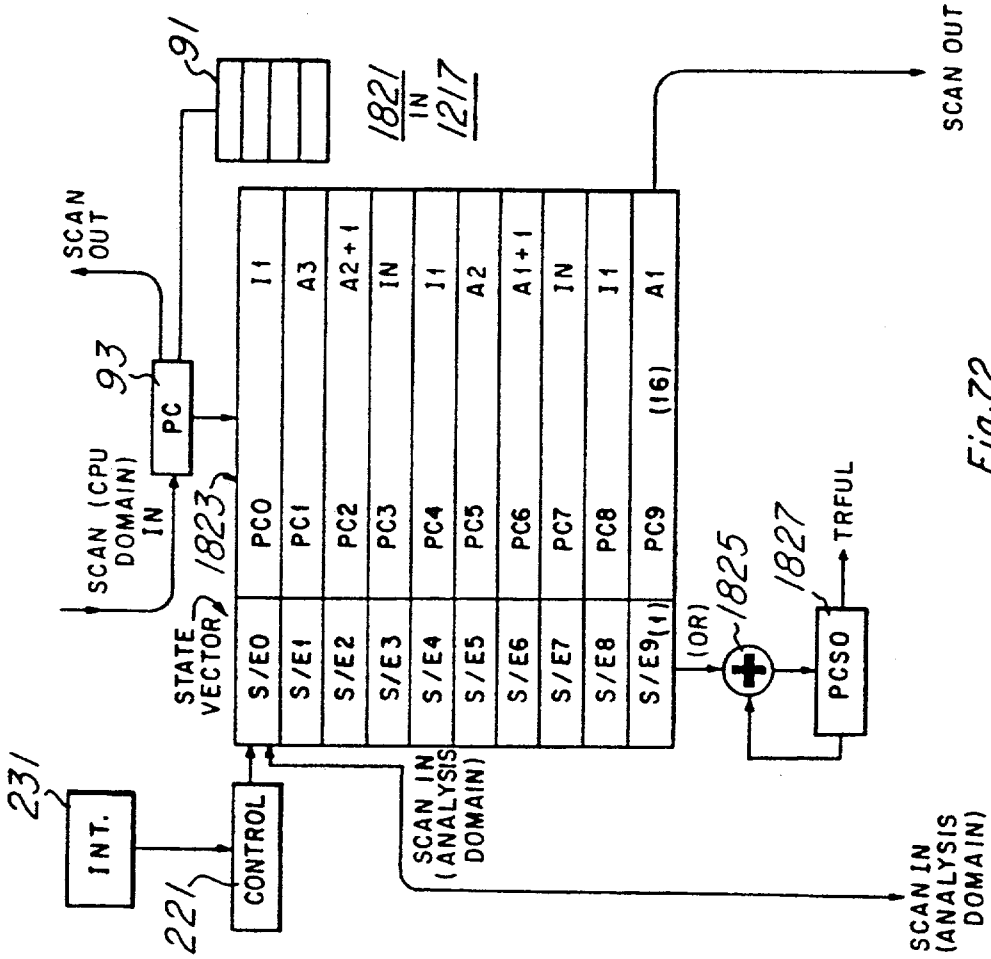

231
INT.
221
CONTROL
STATE VECTOR
1823
S/E0
S/E1
S/E2
S/E3
S/E4
S/E5
S/E6
S/E7
S/E8
S/E9
PC0
PC1
PC2
PC3
PC4
PC5
PC6
PC7
PC8
PC9
I1
A3
A2+1
IN
I1
A2
A1+1
IN
I1
A1
(16)
(1)
(OR)
1825
1827
PCSO
TRFUL
SCAN IN (ANALYSIS DOMAIN)
SCAN IN (ANALYSIS DOMAIN)
SCAN OUT
1821
IN
1217
91
93
PC
SCAN (CPU DOMAIN) IN
SCAN OUT

Fig.72

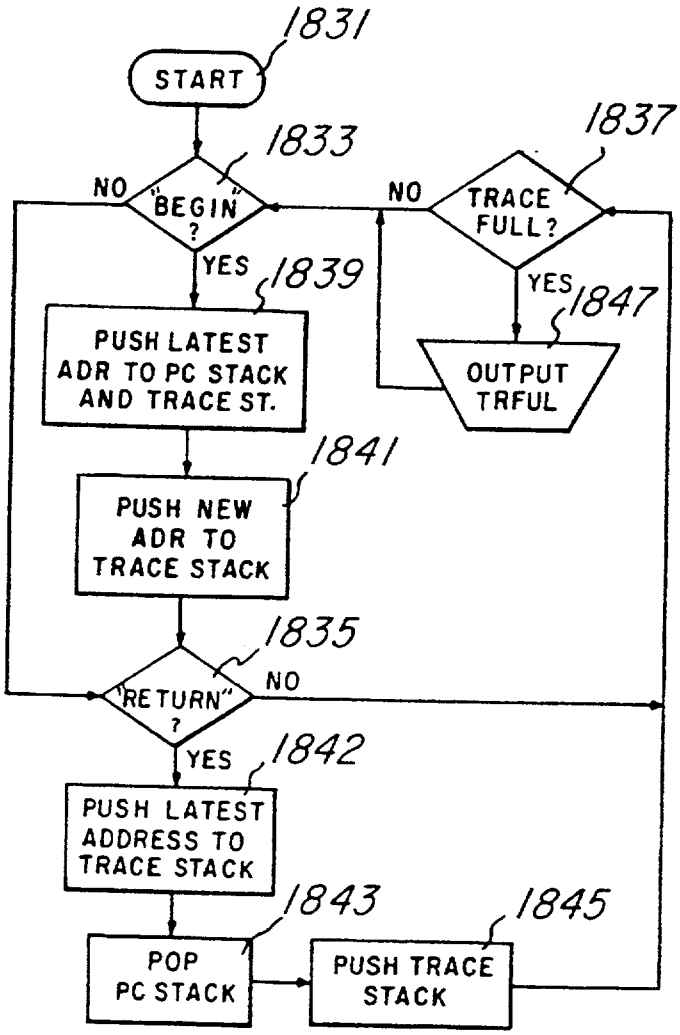


Fig. 73

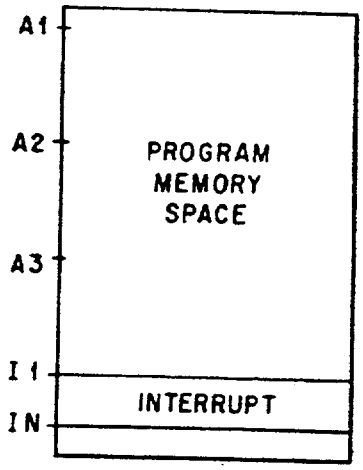


Fig. 74

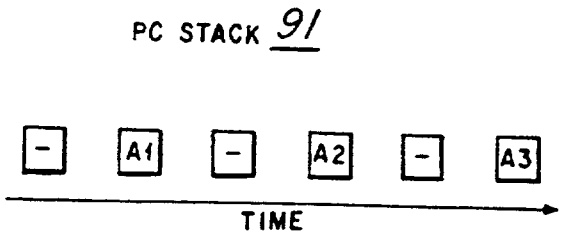


Fig. 75

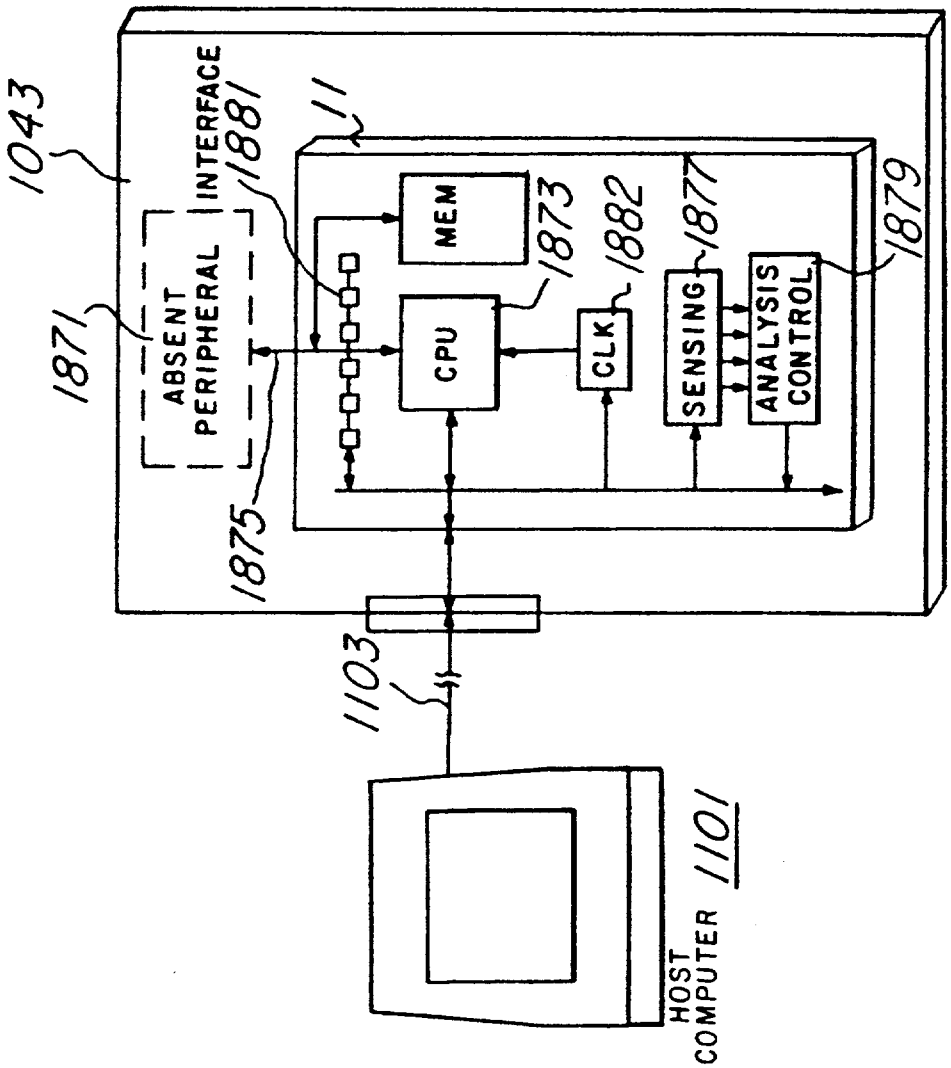

1043
1871
ABSENT PERIPHERAL
INTERFACE
1881
1875
11
MEM
1873
CPU
CLK
1882
SENSING
1877
ANALYSIS CONTROL
1879
1103
HOST COMPUTER
1101

Fig.76

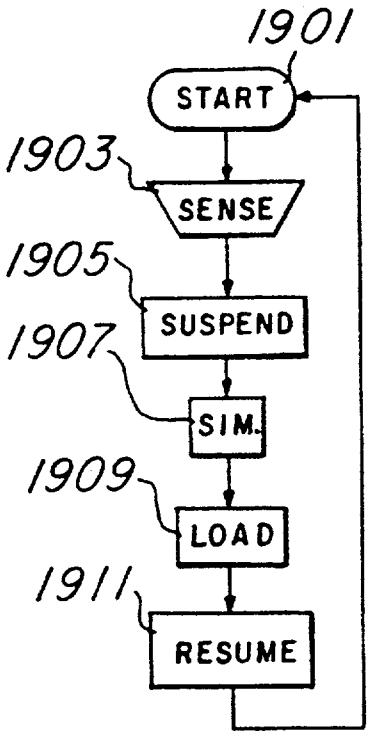

1901
START
1903
SENSE
1905
SUSPEND
1907
SIM.
1909
LOAD
1911
RESUME

Fig. 77

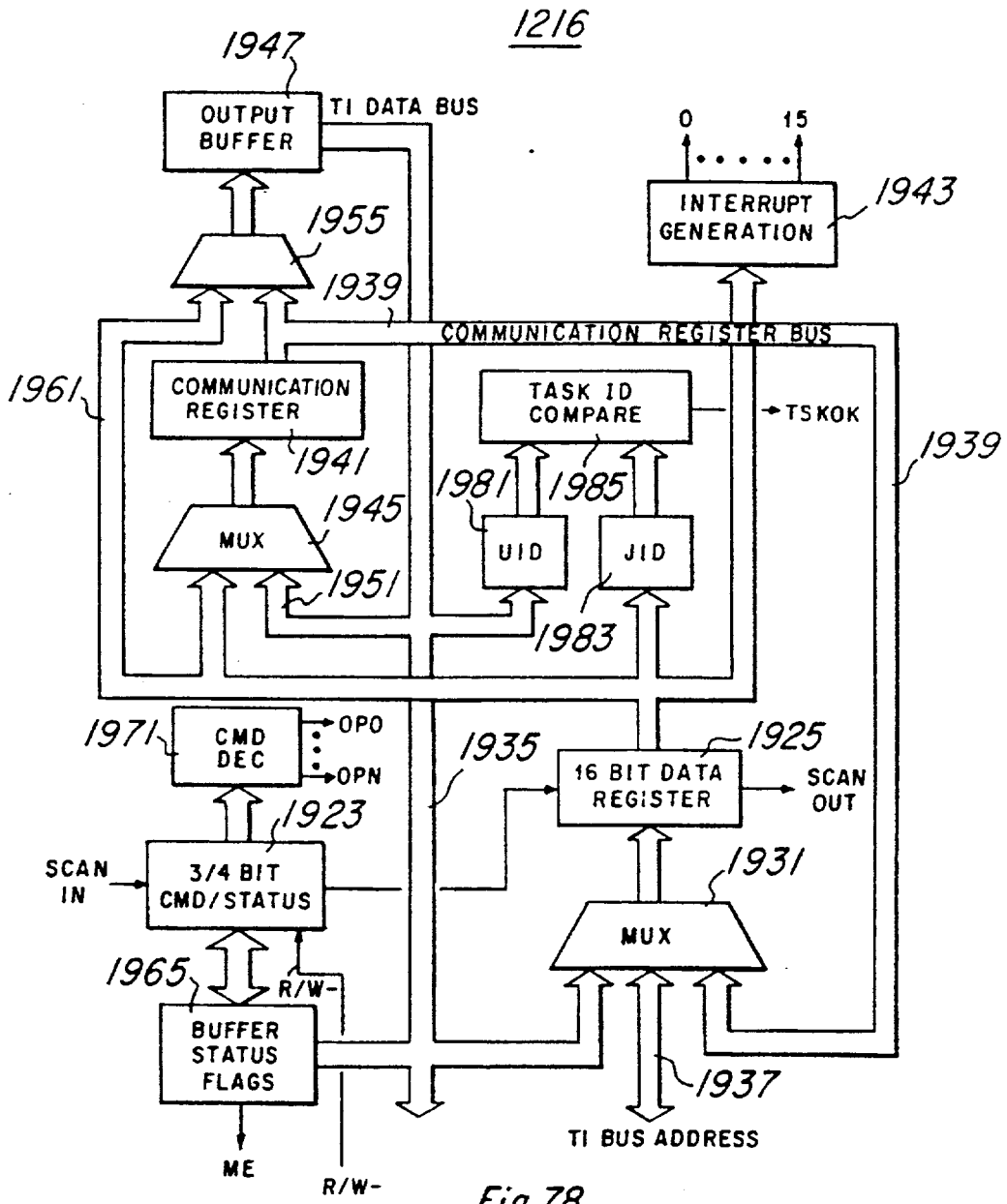

1216
1947
OUTPUT BUFFER
TI DATA BUS
0
15
INTERRUPT GENERATION
1943
1955
1939
COMMUNICATION REGISTER BUS
1961
COMMUNICATION REGISTER
TASK ID COMPARE
TSKOK
1939
1941
1981
1985
1945
MUX
UID
JID
1951
1983
1971
CMD DEC
OPO
OPN
1935
1925
16 BIT DATA REGISTER
SCAN OUT
1923
SCAN IN
3/4 BIT CMD/STATUS
1931
MUX
1965
R/W-
BUFFER STATUS FLAGS
1937
TI BUS ADDRESS
ME
R/W-

Fig.78

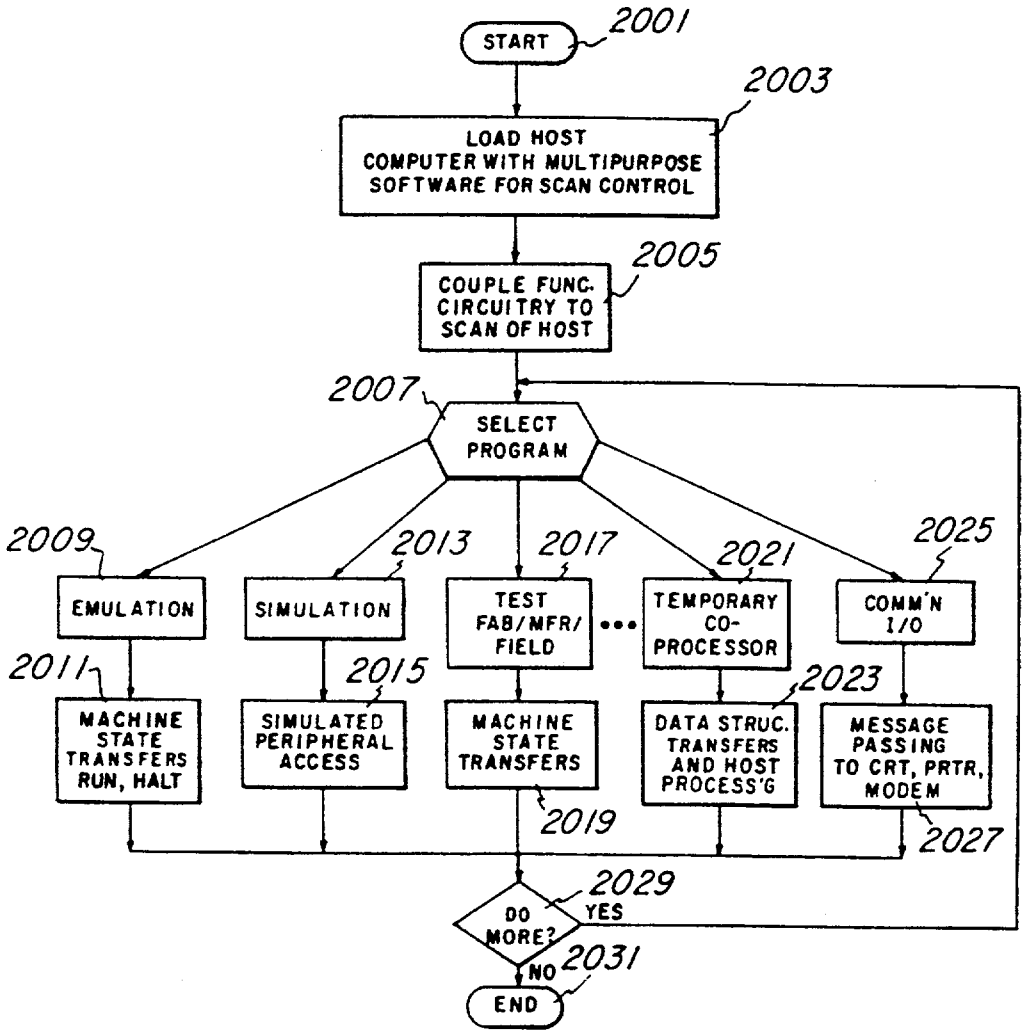

START
2001
2003
LOAD HOST COMPUTER WITH MULTIPURPOSE SOFTWARE FOR SCAN CONTROL
2005
COUPLE FUNC. CIRCUITRY TO SCAN OF HOST
2007
SELECT PROGRAM
2009
EMULATION
2013
SIMULATION
2017
TEST FAB/MFR/ FIELD
2021
TEMPORARY CO-PROCESSOR
2025
COMM'N I/O
2011
MACHINE STATE TRANSFERS RUN, HALT
2015
SIMULATED PERIPHERAL ACCESS
MACHINE STATE TRANSFERS
2019
2023
DATA STRUC. TRANSFERS AND HOST PROCESS'G
MESSAGE PASSING TO CRT, PRTR, MODEM
2027
2029
DO MORE?
YES
NO
2031
END

Fig.79

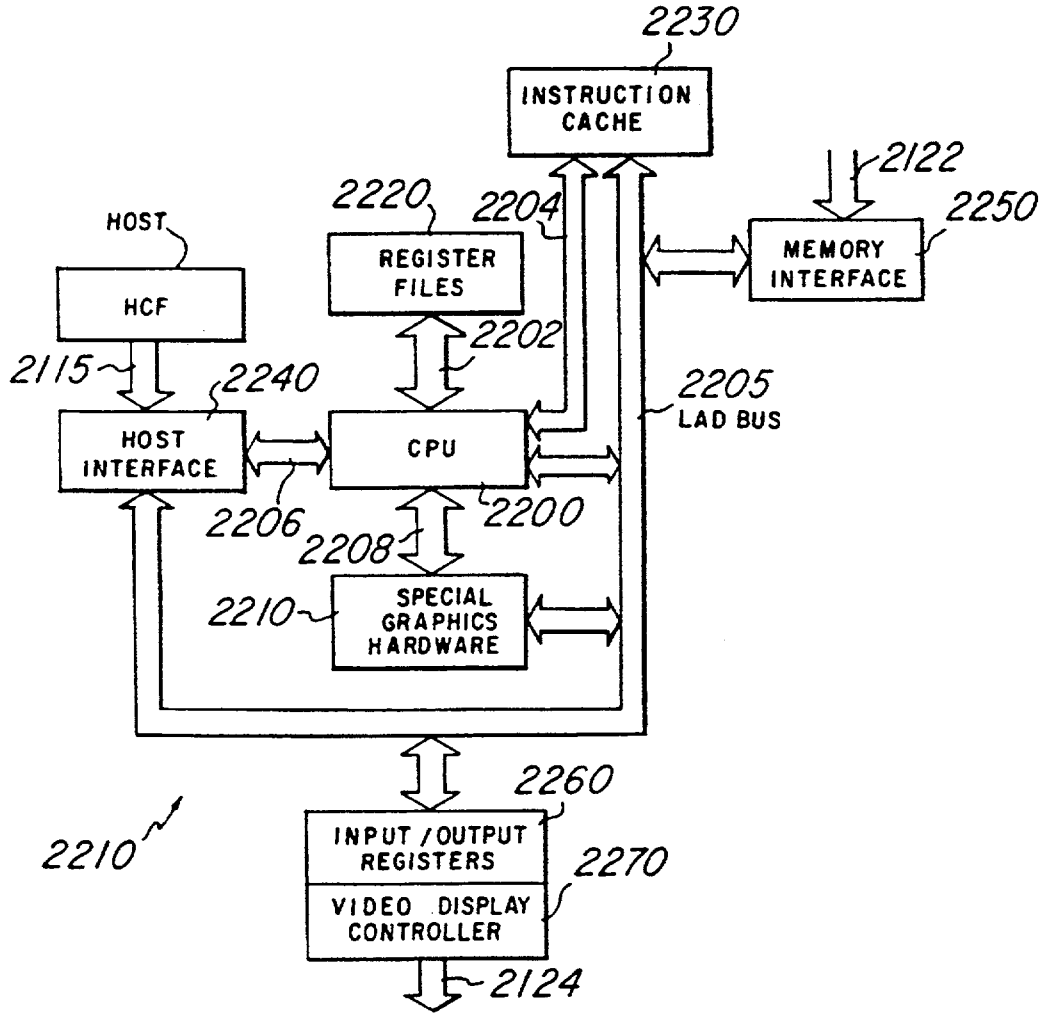

2230
INSTRUCTION CACHE
2122
2250
MEMORY INTERFACE
2220
2204
REGISTER FILES
HOST
HCF
2202
2115
2240
2205
LAD BUS
HOST INTERFACE
CPU
2206
2200
2208
2210
SPECIAL GRAPHICS HARDWARE
2260
2210
INPUT / OUTPUT REGISTERS
2270
VIDEO DISPLAY CONTROLLER
2124

Fig. 80

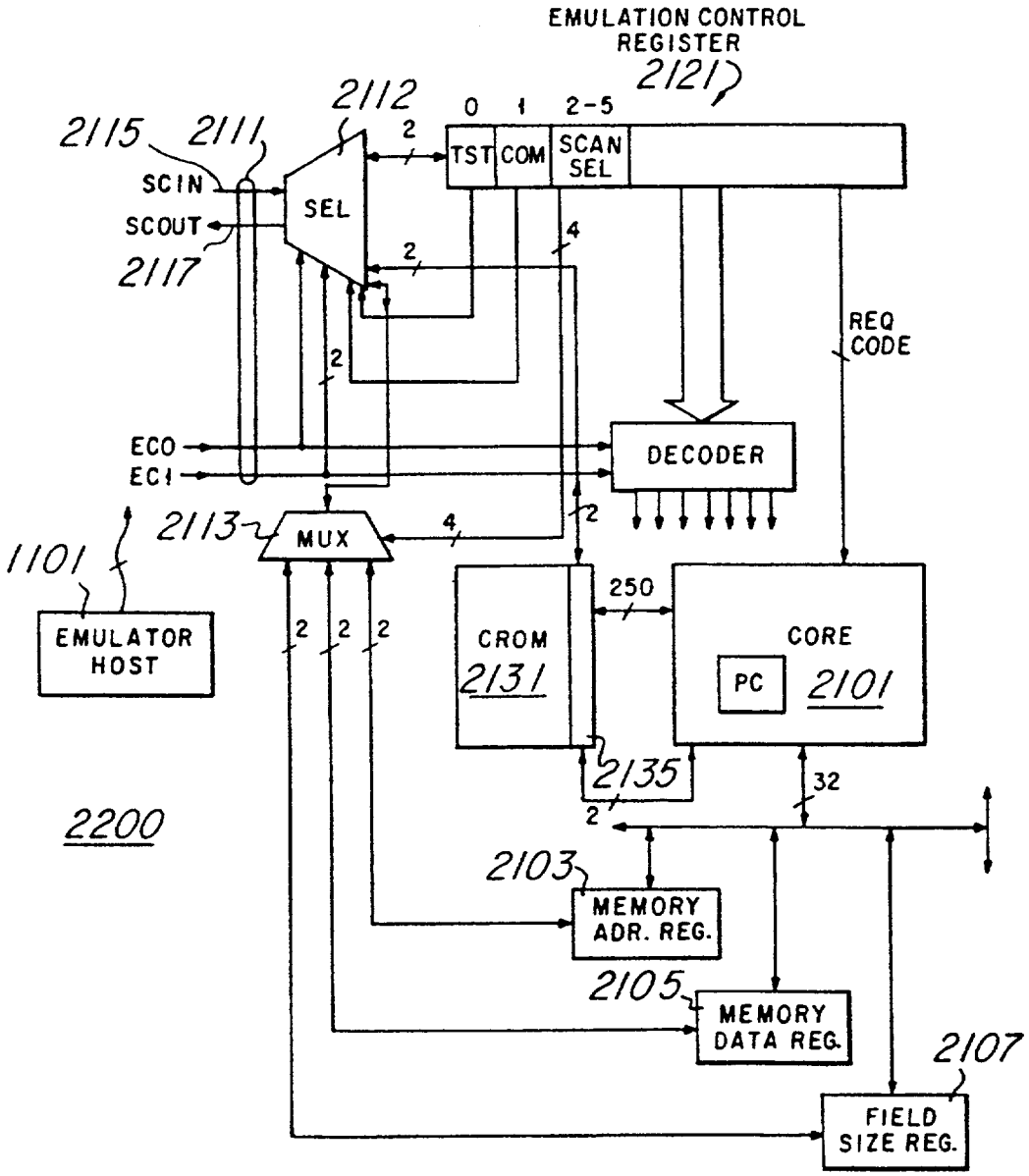

EMULATION CONTROL REGISTER
2121
0
1
2-5
TST
COM
SCAN SEL
2112
2115
2111
SCIN
SCOUT
2117
SEL
2
4
REQ CODE
ECO
ECI
DECODER
2113
MUX
1101
EMULATOR HOST
CROM
2131
250
CORE
PC
2101
2135
32
2200
2103
MEMORY ADR. REG.
2105
MEMORY DATA REG.
2107
FIELD SIZE REG.

Fig. 81

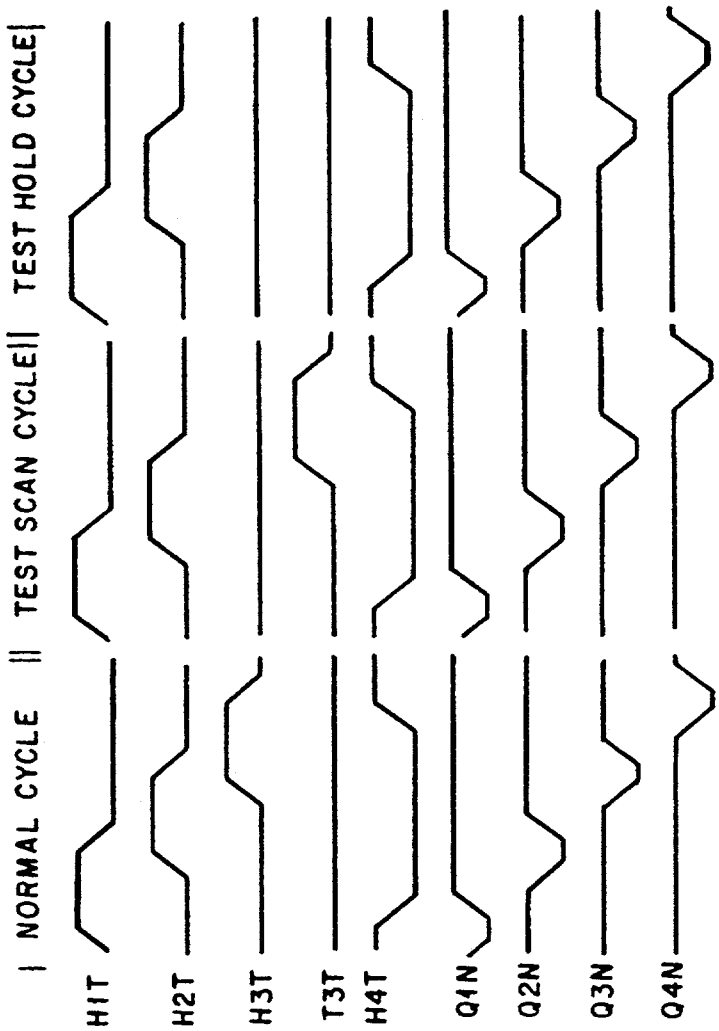

NORMAL CYCLE
TEST SCAN CYCLE
TEST HOLD CYCLE
H1T
H2T
H3T
T3T
H4T
Q1N
Q2N
Q3N
Q4N

Fig. 82

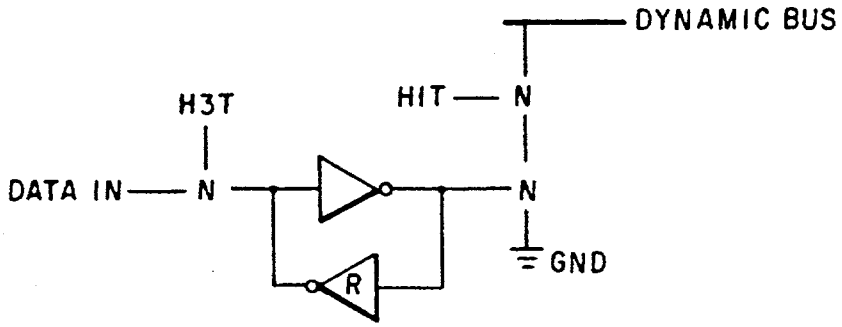

DYNAMIC BUS
H3T
HIT
N
DATA IN
N
N
GND
R

Fig. 83

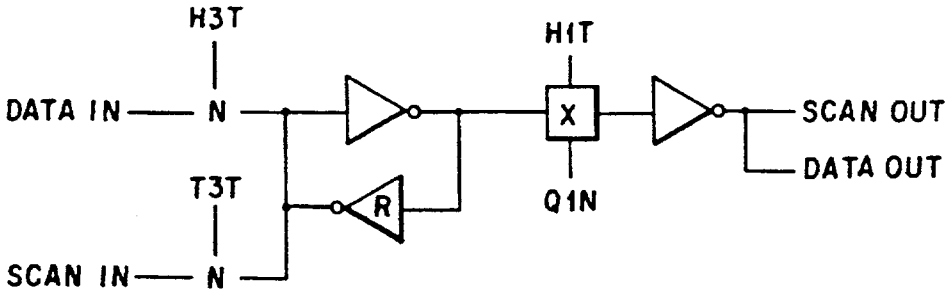

H3T
HIT
DATA IN
N
X
SCAN OUT
DATA OUT
T3T
R
Q1N
SCAN IN
N

Fig. 84

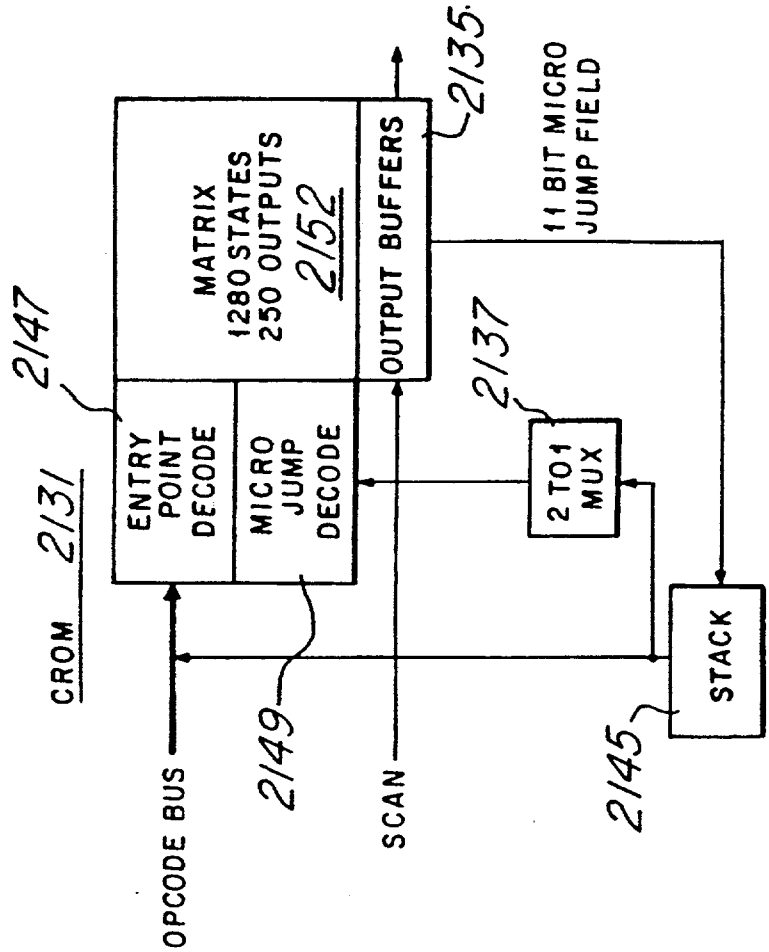

CROM 2131
2147
ENTRY POINT DECODE
MICRO JUMP DECODE
MATRIX
1280 STATES
250 OUTPUTS
2152
OUTPUT BUFFERS
2135
11 BIT MICRO JUMP FIELD
2137
2 TO 1 MUX
STACK
2145
2149
OPCODE BUS
SCAN

Fig. 85

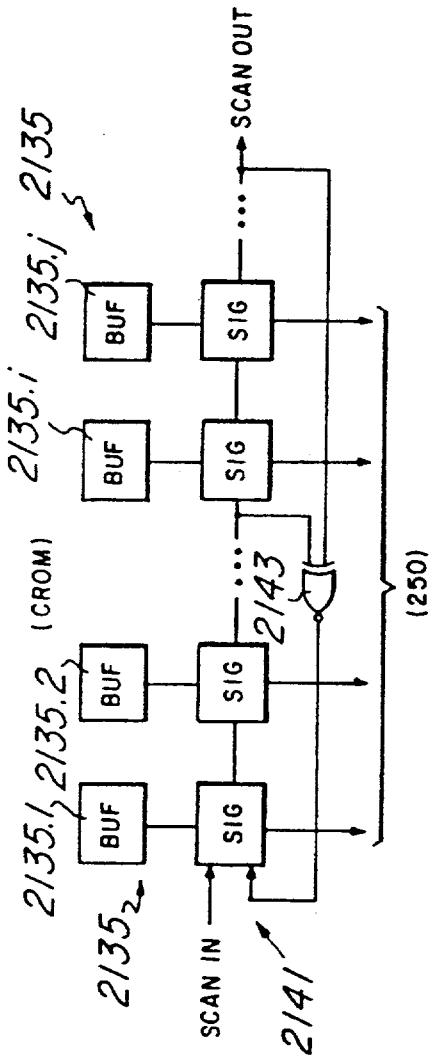

2135
2135.j
2135.i
2135.2
2135.1
2135
(CROM)
BUF
BUF
BUF
BUF
SIG
SIG
SIG
SIG
SCAN OUT
SCAN IN
2141
2143
(250)

Fig. 86

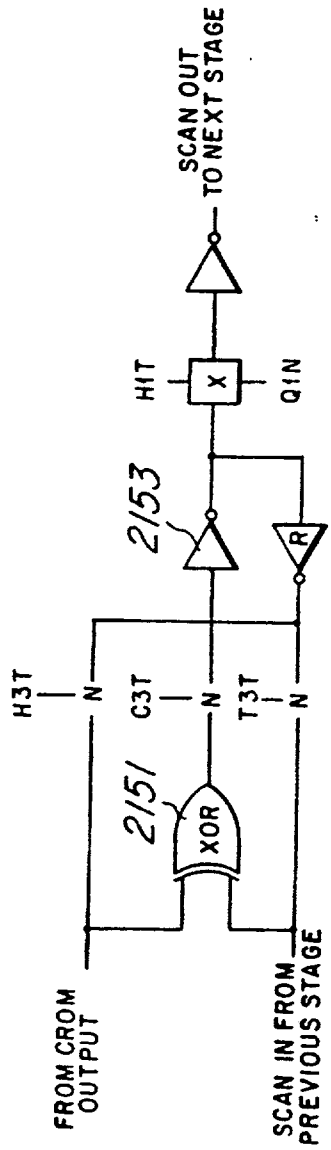

SCAN OUT TO NEXT STAGE
H1T
X
Q1N
2153
R
H3T
N
C3T
N
T3T
N
2151
XOR
FROM CROM OUTPUT
SCAN IN FROM PREVIOUS STAGE

Fig. 87

EMULATION DEVICES, SYSTEMS AND METHODS UTILIZING STATE MACHINES

NOTICE

(C) Copyright 1989 Texas Instruments Incorporated. A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation of Ser. No. 07/911,250 filed Jul. 7, 1992 and now abandoned, which is a continuation of Ser. No. 07/387,549, filed Jul. 31, 1989 and now abandoned, which is a continuation-in-part of Ser. No. 093,463, filed Sep. 4, 1987, now abandoned, which is a continuation-in-part of Ser. No. 057,078, now U.S. Pat. No. 4,860,290, filed Jun. 2, 1987. The related coassigned patent documents listed below are hereby incorporated herein by reference.

CROSS REFERENCE TABLE

Docket No.	PTO Reference	Effective Filing Date
TI-12033	Patent No. 4 860 290	06/02/1987
TI-12451	Patent No. 5 109 494	12/31/1987
TI-12543	Patent No. 5 101 498	12/31/1987
TI-14083	Serial No. 08/001 915	05/04/1989
TI-14147	Serial No. 07/918 902	05/04/1989
TI-14079	Serial No. 07/347 605	05/04/1989
TI-14080	Patent No. 5 072 418	05/04/1989
TI-14081	Patent No. 5 142 677	05/04/1989
TI-14082	Patent No. 5 155 812	05/04/1989
TI-14145	Serial No. 07/967 942	05/04/1989
TI-14141	Serial No. 07/846 459	07/31/1989
TI-14142	Serial No. 07/832 661	07/31/1989
TI-14143	Serial No. 07/827 549	07/31/1989
TI-14282	Serial No. 07/949 757	07/31/1989
TI-14308	Serial No. 07/979 396	07/31/1989
TI-12016	Serial No. 08/108 775	09/07/1988
TI-13371	Serial No. 08/087/020	09/07/1988
TI-13363	Patent No. 5 084 874	09/07/1988
TI-12015	Patent No. 4 872 169	03/06/1987
TI-12698	Serial No. 07/440 454*	09/04/1987
TI-14312	Patent No. 5 237 672	07/28/1989
TI-14315	Serial No. 07/387 569*	07/28/1989
TI-14316	Serial No. 07/387 455*	07/28/1989
TI-14320	Serial No. 07/386 850*	07/28/1989
TI-13495	Patent No. 5 233 690	07/28/1989
TI-11398	Patent No. 5 140 687	12/31/1986

* = abandoned

This invention relates to electronic data processing and emulation, simulation, and testability devices and systems, and methods of their manufacture and operation.

BACKGROUND OF THE INVENTION

Advanced wafer lithography and surface-mount packaging technology are integrating increasingly complex functions at both the silicon and printed circuit board level of electronic design. Diminished physical access is an unfortunate consequence of denser designs and shrinking interconnect pitch. Designed-in testability is needed, so that the finished product is still both controllable and observable during test and debug. Any manufacturing defect is preferably detectable during final test before a product is shipped. This basic necessity is difficult to achieve for complex designs without taking testability into account in the logic design phase, so that automatic test equipment can test the product.

In addition to testing for functionality and for manufacturing defects, application software development requires a similar level of simulation, observability and controllability in the system or sub-system design phase. The emulation phase of design should ensure that an IC (integrated circuit), or set of ICs, functions correctly in the end equipment or application when linked with the software programs.

With the increasing use of ICs in the automotive industry, telecommunications, defense systems, and life support systems, thorough testing and extensive real-time debug becomes a critical need.

Functional testing, wherein a designer is responsible for generating test vectors that are intended to ensure conformance to specification, still remains a widely used test methodology. For very large systems this method proves inadequate in providing a high level of detectable fault coverage. Automatically generated test patterns would be desirable for full testability, and controllability and observability are key goals that span the full hierarchy of test (from the system level to the transistor level).

Another problem in large designs is the long time and substantial expense involved. It would be desirable to have testability circuitry, system and methods that are consistent with a concept of design-for-reusability. In this way, subsequent devices and systems can have a low marginal design cost for testability, simulation and emulation by reusing the testability, simulation and emulation circuitry, systems and methods that are implemented in an initial device. Without a proactive testability, simulation and emulation approach, a large of subsequent design time is expended on test pattern creation and grading.

Even if a significant investment were made to design a module to be reusable and to fully create and grade its test patterns, subsequent use of module may bury it in application specific logic, and make its access difficult or impossible. Consequently, it is desirable to avoid this pitfall.

The advances in IC design, for example, are accompanied by decreased internal visibility and control, reduced fault coverage and reduced ability to toggle states, more test development and verification problems, increased complexity of design simulation and continually increasing cost of CAD (computer aided design) tools. In the board design the side effects include decreased register visibility and control, complicated debug and simulation in design verification, loss of conventional emulation due to loss of physical access by packaging many circuits in one package, increased routing complexity on the board, increased costs of design tools, mixed-mode packaging, and design for produceability. In application development, some side effects are decreased visibility of states, high speed emulation difficulties, scaled time simulation, increased debugging complexity, and increased costs of emulators. Production side effects involve decreased visibility and control, complications in test vectors and models, increased test complexity, mixed-mode packaging, continually increasing costs of automatic test equipment even into the 7-figure range, and tighter tolerances.

SUMMARY OF THE INVENTION

Among the objects of the present invention are to provide improved emulation, simulation and testability architectures and methods which provide visibility and control without physical probing or special test fixtures; to provide improved emulation, simulation and testability architectures and methods which are applicable to critical components of system designs to support test and integration of both hardware and software; to provide improved emulation, simulation and testability architectures and methods that are a viable alternative to high capital-cost test equipment and systems; to provide improved emulation, simulation and testability architectures and methods which integrate access to sophisticated operations in hardware emulation, fault emulation, simulation and built-in tests to provide improved emulation, simulation and testability architectures and methods which apply hardware and software visibility and control to reduce application development time and thus reduce the user manufacturer's time-to-market on new products; and to provide improved emulation, simulation and testability architectures and methods to leverage hierarchical partitioning and automatically generate reusable tests for related chips and systems.

Generally, one form of the invention is an emulation device including a serial scan testability interface having at least first and second scan paths, and state machine circuitry connected and responsive to said second scan path generally operable for emulation control.

Other device, system and method forms of the invention are also disclosed and claimed herein. Other objects of the invention are disclosed and still other objects will be apparent from the disclosure herein.

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The preferred embodiments of the invention as well as other features and advantages thereof will be be best understood by reference to the detailed description which follows, read in conjunction with the accompanying drawings, wherein FIGS. **1–43** are incorporated from any of applications 07/347,605, 08/001,915, 07/967,942 and 07/918,902 and U.S. Pat. Nos. 5,072,418, 5,142,677, 5,155,812 wherein:

FIG. **44** is a pictorial diagram of development tools for developing integrated circuit chips and software;

FIG. **45** is a partially pictorial, partially block diagram of a system configuration for emulation, simulation, testability and attached processor data processing, communications I/O and peripheral access;

FIG. **46** is a diagram of a software configuration for a host computer of FIG. **45**;

FIG. **47** is a block diagram of a modular port scan (MPSD) arrangement;

FIG. **48** is a block diagram of a scan test/MPSD configuration;

FIG. **49** a block diagram of an integrated approach to test and emulation circuitry;

FIG. **50** is a partially block, partially schematic diagram of a scan testability interface;

FIG. **50A** is a state transition diagram of a test access port (TAP) controller in FIG. **50**;

FIG. **51** is a block diagram of processor chip domains, boundary scan and scan test/emulation circuitry on chip;

FIG. **52** is a block diagram of the processor chip of FIG. **51** showing functional blocks of the chip allocated to the various domains, and showing a message passing circuit;

FIG. **53** is partially pictorial, partially block diagram of the processor chip of FIGS. **51** and **52**;

FIG. **54** is a block diagram of scan paths in greater detail than that of FIG. **50**;

FIG. **55** is a block of scan paths in greater detail than that of FIG. **54**;

FIG. **56** is a block diagram of connections of a control adapter to the domains, showing nomenclature;

FIG. **57** is a block diagram of modules in the domains, also illustrating a mode-driven stops process;

FIG. **58** is a process diagram of operation of the system of FIGS. **45**, **50**, **57** and **59** for emulation, simulation and testability;

FIG. **59** is a detailed block diagram of the adapter of FIGS. **49**, **51**, **52**, **53**, **56** and **57**;

FIG. **59A** is a compact diagram of shift register latches SRLs in a scan chain in FIG. **59**;

FIG. **60** is a schematic diagram of a code state machine and an event manager circuit therefor in the adapter of FIG. **59**;

FIG. **61** is a state transition diagram of the code state machine of FIG. **60**;

FIG. **62** is a schematic diagram of selection and flip-flop circuitry of the adapter of FIG. **59**;

FIG. **63** is a schematic diagram of a lock control circuit of the adapter of FIG. **59**;

FIG. **64** is a schematic diagram of one of three identical logic circuits of the adapter of FIG. **59** supplying codes to a domain;

FIG. **65** is a schematic diagram of one of three identical clock control circuits of the adapter of FIG. **59** for switching functional clock FCLK or test clock JCLK to a domain;

FIG. **66** is a pictorial diagram of a testing system for testing numerous integrated circuits on a wafer in wafer fabrication;

FIG. **67** is a process flow diagram of operation of the testing system of FIG. **66**;

FIGS. **68A** and **68B** are two halves of a block diagram of a central processing unit CPU core improved for emulation, simulation and testability;

FIG. **69** is a block diagram of an analysis circuit for monitoring the operations of an integrated circuit device;

FIG. **70** is a process flow diagram of operations of the analysis circuit of FIG. **69**;

FIG. **71** is a block diagram of a hardware breakpoint circuit in FIG. **68A**;

FIG. **72** is a block diagram of a trace stack in FIG. **68A**;

FIG. **73** is a process flow diagram of operations of the trace stack and a program counter stack of FIG. **68A**;

FIG. **74** is an address map of a processor device;

FIG. **75** is a time-series diagram of the contents of the program counter stack and not the trace stack;

FIG. **76** is a partially pictorial, partially block diagram of a system for simulated peripheral accesses;

FIG. **77** is a process flow diagram of operations of the system of FIG. **76**;

FIG. **78** is a block diagram of the message passing circuitry of FIG. **52**;

FIG. **79** is a process flow diagram of an attached processor method of operating the system of FIG. **45**;

FIG. 80 is a block diagram of a graphic system processor GSP chip;

FIG. 81 is a more detailed block diagram of a CPU portion of the GSP chip of FIG. 80 showing testability, emulation and simulation circuitry;

FIG. 82 is a waveform diagram of clock waveforms for operating the GSP chip of FIG. 80;

FIG. 83 is a schematic of a parallel register latch for use in the GSP chip of FIG. 80;

FIG. 84 is a schematic of a serial register latch for use in the GSP chip of FIG. 80;

FIG. 85 is a block diagram of a control read only memory (CROM) for the GSP chip of FIG. 80;

FIG. 86 is a detailed block diagram of signature analysis test circuitry for the CROM of FIG. 85; and

FIG. 87 is a schematic diagram of a cell in the signature analysis test circuitry of FIG. 86.

Corresponding numerals and other corresponding symbols refer to corresponding parts in the various Figures of drawing except where the context indicates otherwise.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Various inventive electronic architectures, devices, systems and methods were described extensively in the detailed description and drawings 1-43 common to all of the coassigned applications with Ser. Nos. 347,605; 347,596 issued Oct. 10, 1991 (U.S. Pat. No. 5,072,418) 347,615; 347,966; 347,968; 347,967; and 347,969. All of these foregoing coassigned applications are incorporated herein by reference. Numbering of Figures in the present application begins with FIG. 44 to continue the sequence of detailed description. Corresponding numerals in this application and said coassigned applications refer to corresponding parts for clarity of exposition.

A device 11, described in the coassigned applications and further described herein, is adapted for sophisticated interfacing with development tools illustrated in FIG. 44. Hardware design tools include an extended development system 1101 interfaced by a serial line 1103 to a circuit board 1043 holding device 11. Circuit board 1043 is advantageously operable with application host computer 1044. Also provided in the development tools are an evaluation module 1111 connected to an analog interface board AIB 1113.

A software development system SWDS provides for user entry of source code 1121 in the C computer language which source code then is compiled by a C compiler 1123 into code 1125.

C compiler 1123 is an optimizing compiler fully implementing the standard Kernighan and Ritchie C language, for instance. The compiler 1123 accepts programs written in C and produces assembly language source code, which is then converted into object code by the assembler 1127. This high-level language compiler 1123 allows time-critical routines written in assembly language to be called from within the C program. Conversely, assembly routines may call C functions. The output of the compiler is suitably edited before assembly and link to further optimize the performance of the code. The compiler 1123 supports the insertion of assembly language code into C source code, so that the relative proportions of high-level and assembly language code are tailored according to the needs of a given application.

The code 1125 is assembled by an assembler 1127 into relocatable object code. A linker 1129 produces non-relocatable machine code or linked object code which is then downloaded into the device 11 through the development system.

Assembler 1127 and linker 1129 comprise a software development tool that converts assembly language files into executable object code. Key features are macro capabilities and library functions, conditional assembly, relocatable modules, complete error diagnostics, and symbol table and cross reference. Four programs address specific software development needs, discussed next.

The assembler 1127 translates assembly language source files into machine language object files. Source files contain instructions, assembler directives and macro directives. Assembler directives are used to control various aspects of the assembly process, such as the source listing format, data alignment and section content.

The linker 1129 combines object files into a single executable object module. As the linker creates an executable module, it performs relocation and resolves external references. The linker accepts relocatable object files created by the assembler as input. It also accepts archive library members and output modules created by a previous linker run. Linker directives allow combining or binding of file sections or symbols to addresses and defining or redefining global symbols.

An archiver allows collection of a group of files into a single archive file. For example, several macros are suitably collected into a macro library. The assembler searches through the library and uses the members that are called as macros by the source code 1125. The archiver also suitably collects a group of object files into an object library such as files that resolve external references during linking.

An object format converter converts an object file into any one of several EPROM programmer formats, such as TI-TAG format. The converted file is then downloaded to an EPROM programmer so that the EPROM code so established is then executed on the device 11 target chip in system 1043.

Simulator 1131 executes a software program that simulates operation of the target chip for cost-effective software development and program verification in non-realtime. The simulator simulates the entire target chip instruction set and simulates the key peripheral features including DMA, timers and serial port when the target chip includes them. Command entry is accepted from either menu-driven keystrokes (menu mode) or from a batch file (line mode). Help menus are provided for all screen modes. Its standard interface can be user customized. Simulation parameters are quickly stored/retrieved from files to facilitate preparation for individual sessions. Reverse assembly allows editing and reassembly of source statements. Memory is displayed as hexadecimal 32 bit values and assembled source code, separately or at the same time.

Simulator 1131 execution modes include 1) single/-multiple instruction count, 2) single/multiple cycle count, 3) Until Condition Is Met, 4) While Condition Exists, 5) For Set Loop Count and 6) Unrestricted Run with Halt by Key Input. Trace expressions are readily defined. In trace execution, display choices include 1) designated expression values, 2) cache registers, and 3) instruction pipeline for easy optimization of code. Breakpoint conditions include Address Read, Address Write, Address Read or Write, Address Execute, and Expression Valid. Simulator 1131 simulates cache utili-

zation and does cycle counting. For example, in cycle counting the number of clock cycles in single step mode or run mode are displayed. External memory is suitably configured with wait states for accurate cycle counting.

Simulator **1131** accepts object code produced by the assembler **1127** and linker **1129**. Input and output files are suitable associated with the port addresses of the I/O instructions to simulate I/O devices connected to the processor. Before starting program execution, any breakpoints are set and the trace format defined.

During program execution on simulator **1131**, the internal registers and memory of the simulated target chip are modified as each instruction is interpreted by the simulator **1131**. Execution is suspended when a breakpoint or error is encountered or when execution is halted. When program execution is suspended, the internal registers and both program and data memories can be inspected and modified. A trace memory is also displayable. A record of the simulation session can be maintained in a Journal file so that it can be re-executed to regain the same machine state during another simulation session.

The simulator **1131** allows verification and monitoring of the state of the target chip without the requirements of hardware. Simulation speed is on the order of hundreds or thousands of instructions per second depending on the operating system and hardware selected for simulator **1131**. A state-accurate simulation might be as slow as 1-2 instructions per second. Emulation at the higher real-time functional clock rate is performed by development system **1101** instead of simulator **1131**.

Simulator **1131** provides for complete computer simulation not only of the device **11**, but also its peripherals on the board **1043** through file I/O for example.

Extended development system **1101** provides full-speed, in-circuit emulation for system design and for hardware and software debug on widely available personal computer systems. The development tools provide technological support from system concept to prototype. The development system elements provide ease of use and offer the designer the tools needed to significantly reduce application system development time and cost to put designs into production faster.

FIG. **45** illustrates in even more detail the emulation environment provided by the extended development system **1101**. A controller card **1141** compatible with IEEE JTAG standards is included in the emulation host computer **1101**. This controller card **1141** communicates by serial line **1103** to PC board **1043** and DSP device **11** of FIG. **45**. System **1043** has Texas Instruments Scope (TM) testability meshed with Texas Instruments MPSD (Modular Port Scan Design) emulation for a complete solution from development, through manufacture, and including field test. The inventive approaches are applicable in digital signal processors (DSP), graphics signal processors (GSP), memories (MEM), programmable array logic (PAL), application specific integrated circuits (ASIC), and general purpose logic (GPL) general purpose Micro Computers and Micro processors, and any device requiring test or code development.

Host computer **1101** of FIG. **45** has peripherals including a printer **1147**, hard disk **1145**, and telecommunications modem **1143** connected to a telephone line for uploading to a remote mainframe in field test and other procedures. The peripheral capabilities of bus **1148** of host computer **1101** are not only available for emulation, but also provide access by application system **1043** to these peripherals along serial line **1103**. Host computer **1101** thus is not only available to the system **1043** as an emulation host but also as an attached processor itself and as a port for communications I/O and to other peripheral capabilities temporarily needed by system **1043** but ordinarily unavailable to system **1043**.

FIG. **46** illustrates an emulation and simulation software configuration for computer **1101** wherein device independent emulator software has a window driven user interface and a test executive program.

Device specific configuration files for each of the devices on board **1043** are provided. For example, there is a DSP configuration file, a GSP (graphic signal processor) configuration, a programmable array logic (PAL) file, an ASIC file and a GPL register file.

The emulation hardware and software of FIGS. **45** and **46** provide a user-friendly, personal-computer or work station-based development system which provides all the features necessary to perform full-speed in-circuit emulation with target chips on board **1043**. For example, DSP **11** is suitably a Texas Instruments 320 series digital signal processor disclosed in coassigned application Ser. No. 025,417, issued Mar. 27, 1990 (U.S. Pat. No. 4,912,636) filed Mar. 13, 1987 and hereby incorporated herein by reference; or a 320C50 digital signal processor disclosed in U.S. Pat. No. 5,072,418 which is incorporated herein by reference. An exemplary graphics signal processor is the Texas Instruments 34020 GSP disclosed in the GSP coassigned applications incorporated hereinabove and having inventive emulation circuitry more fully described hereinbelow.

The emulator comprised of FIG. **45** host computer **1101** with controller card **1141** and software of FIG. **46** allows the user to perform software and hardware development, and to integrate the software and hardware with the target system. An important emulation interface provides control and access to every memory location and register of the target chip and extend the device architecture as an attached processor.

Emulator controller card **1141** provides full-speed execution and monitoring of each target chip such as device **11** in the user's target system **1043** via a multi-pin target connector. In one embodiment, thirty software and hardware breakpoints, software and hardware trace and timing, and single-step execution are provided. The emulator has capability to load, inspect, and modify all device **11** registers. Program data and program memory can be uploaded or downloaded. The user interface of host computer **1101** for emulation purposes is a windowed user interface designed to be identical to the windowed user interface of simulator **1131** for the corresponding target chip. The emulator **1101** is portable and reconnectable for multiprocessing. Emulator **1101** provides a benchmark of execution time clock cycles in realtime.

Full-speed execution and monitoring of the target system is suitably controlled via a multi-wire interface or scan path in the multi-pin target connector. The scan path controls the target chip in the system **1043**, providing access to all the registers as well as associated internal and external memory.

Program execution takes place on the target chip (e.g. **11**) in the target system **1043**. Accordingly, there are no timing differences during emulation, as might occur without the in-circuit emulation provided by this preferred embodiment. Heretofore, emulation may have involved sending signals over a cable to emulate the

target chip 11 in its absence. Advantageously, the present embodiment is a non-intrusive system that utilizes chip 11 itself, and avoids cable length and transmission problems. Loading problems on signals are avoided, and artificial memory limitations are obviated. Emulation performance coincides with specifications for the emulated target chip itself.

Software breakpoints allow program execution to be halted at a specified instruction address. Hardware breakpoints are also advantageously operative on-chip. When a given breakpoint is reached, the program either halts execution to permit user observation of memory and status registers, or the breakpoint is included in a more complex condition, which when satisfied results in an appropriate stop mode being executed. At this point, the status of the target chip or system is available for display by the user with as little as a single command.

Software trace and hardware program counter trace permit the user to view the state of target chip 11 when a breakpoint is reached. This information is suitably saved on command in a file for future analysis. Software timing allows the user to track clock cycles between breakpoints for benchmarking time critical code.

Single-step execution gives the user the ability to step through the program one instruction at a time. After each instruction, the status of the registers and CPU are displayed. This provides greater flexibility during software debug and helps reduce development time.

Object code is downloaded on command to any valid program memory location or data memory location via the interface. Downloading a 1K-byte object program illustratively takes on the order of 100 milliseconds. By inspecting and modifying the registers while single-stepping through a program, the user can examine and modify program code or parameters.

A windowed user interface for emulator 1101 is suitably made identical to that of simulator 1131, affording a straightforward migration from simulator-based development to emulator-based development. The user-friendly screen displays the program code in mnemonics and equivalent hexadecimal code. Windowed displays are suitably provided for extended precision registers, the CPU status and memory locations.

A first screen option is a primary screen that includes a command line displayed at top of screen, functions of special-function keys, and four status windows which are individually accessed using the F1 key of commercially available keyboards. The windows include a source code window, an auxiliary display window, a CPU status window, and an extended precision registers window. The contents of the windows are made accessible for user inspection and modification.

Commands are entered in a MENU mode or a LINE mode. In the MENU mode, a menu at the top of the screen permits the user to view every option available while entering a single command. Further menus are then displayed until the entire command has been entered. The LINE mode allows user to enter an entire command expression. A summary of commands is provided in the appendix.

Emulator card 1141 of FIG. 45 suitably occupies slots in an IBM PC-XT/AT computer when the latter is used as host computer 1101. The card 1141 is detached and transferred to another PC (personal computer of equivalent functionality) as needed, affording emulator portability. For simulation, a memory map for the controller card 1141, which may include EPROM (erasable programmable read only memory), SRAM (static random access memory), DRAM (dynamic random access memory), and on-chip memory and peripherals, can be configured by the designer to reflect the actual environment of the target system 1043, including wait states and access privileges. In this way, card 1141 and host computer 1101 simulate peripherals which are as yet absent from board 1043 in a particular development context.

In one embodiment, multiprocessing applications are emulated by extending line 1103 between each of several application boards from one to the next, maintaining real-time emulation and preserving the information on each target chip.

The development system 1141 operates in two modess emulation mode and algorithm development and verification mode. In the algorithm verification mode, the target chip 11 debugs its software at full speed before the target system is complete. To accomplish this, code is downloaded into the memory on the board 1043 and executed at full speed via the interface on an application board used in place of the incomplete target system. A suitable application board includes a DSP 11, 16K×32 bits of full-speed (zero wait states) SRAM on a primary bus, two selectable banks of 8K×32 bits full speed (zero wait state) SRAM on an expansion bus, and 512K×32 bits DRAM. With ample SRAM, the user has realtime emulation capabilities and memory storage flexibility for a variety of algorithms. Zero wait state capability in SRAM allows memory read/write in real-time.

For algorithim development and code verification the system can single step and run until breakpoint is reached. Algorithim verifiction runs data through the algorithim and verifies its function. Burst exection, I/O and other functions are available.

Page mode DRAM improves bulk storage performance. Three types of DRAM cycles are used on one example of an application board. These are single-word read, single-word write and page-mode read which respectively have wait states of four, two, and one wait state per access. Page mode read cycles are automatically evoked when device 11 performs two or more back-to-back read cycles on the same memory page (256 words). Utilizing page-mode results in a decrease in wait states when accessing on application board 1043 DRAM on application board 1043.

In FIG. 45 both test and development support system access to the application system resource is via a serial scan bus master or scan interface on controller card 1141, and described later hereinbelow. Sophisticated emulation and simulation functions are built out of primitives. Primitives are sets of bits that define control operations (like commands or instructions) available through controller card 1141.

The functionality of the device 11 can be accessed by each of two illustrative inventive serial implementations. A first implementation is Texas Instruments Modular Port Scan Design (MPSD) as shown in FIG. 47 and disclosed in coassigned application Ser. No. 057,078 issued Aug. 22, 1989, (U.S. Pat. No. 4,860,290) and incorporated herein by reference. Shift register latches (SRLs) designated "S" are distributed through the device 11 like a string of beads on a serial scan path respective to each module to provide access to all important registers.

In FIG. 48, a second approach uses a SCOPE transmission medium combined with MPSD technology in a SCOPE interface 1150.

In FIG. 49 device 11 has an on-chip JTAG interface 1149 as described herein. The scan interface is connected to line 1103 of FIG. 45 and has inputs for test clock TCK, mode select TMS, and test data input TDI (scan in), as well as a test data output TDO (scan out). A special emulation adapter 1203 is connected between the scan interface 1149 and MPSD modules of the functional circuitry 1213 of device 11. Emulation adapter 1203 in different forms involves hardwired state machine circuitry, assembly lanquage, or microcoded state machine embodiments.

The characteristics of some implementations when used in support of emulation are shown in Table I:

TABLE I

	MPSD	SCOPE	SCOPE/MPSD
Industry Standard Communication	No	Yes	Yes
Max Clock Period	Depends	Unlimited	Unlimited
Functional Clock Independence	No	Yes	Yes
Boundary Scan Support	No	Yes	Yes
Silicon Efficiency	Yes	No	Yes
Most Emulation Capability	No	Yes	Yes
Number of Extra Pins	Four	Six	Six

The implementation SCOPE/MPSD capitalizes on the strengths of MPSD and SCOPE individually to create a hybrid emulation technology.

FIG. 50 shows a block diagram of improved SCOPE hardware which is provided on each of the chips such as device 11 on PC board 1043. Four pins TDI, TM8, TCK and TDO communicate with the system. TMS and TCK communicate with a tap controller 1151 which is connected to an instruction register 1153 and an instruction decoding circuit 1155.

Test access port (TAP) controller 1151 is in turn coupled to instruction register (IR) 1153 and a first multiplexer 1173. The instruction register can receive serial scan signals from the TDI line and output serially to MUX 1173. MUX 1173 is under control of the TAP and can select the output signal from the instruction register or from another MUX 1171.

The instruction register also controls a bypass register (BR) 1167 and one or more boundary scan registers (BSR) 1161. The bypass register receives the TDI signal and outputs it to MUX 1171. MUX 1171 is under control of the instruction register 1153. Based on the instruction loaded into the instruction register, MUX 1171 outputs its input from the bypass register or its input from one or more BSRs, or internal device register scan. Each boundary scan register is controlled via the test access port and the instruction register.

The boundary scan arrangement operates in a normal mode or a test mode. During the normal mode, input data entering terminals of IC logic passes through the boundary scan register, into the IC logic and out to the normal output terminals without any change due to the BSR. During the test mode, normal input data is interrupted, and test input data is captured, shifted, and updated within the boundary scan register. The boundary scan register includes two memories, a first memory for receiving and shifting data from the TDI line and a second memory for holding output data. The second memory is selectively operable to transfer data from the first memory to the second memory.

Generally, in FIG. 50, serial information is down loaded from emulation computer 1101 via the SCOPE controller card 1141 through pin TDI and enters any one of a number of shift registers, including a boundary scan register 1161, a device identification register 1163 and design specific test data registers 1165. A bypass register 1167 is also provided. These shift registers or serial scan registers are selected via a MUX 1171 under the control of instruction decode circuitry 1155. The selected output from MUX 1171 is fed to a MUX 1173 so that under control of tap controller 1151 the instruction register 1153 or MUX 1171 is selected by MUX 1173. JTAG clock TCK and MUX 1173 output are fed to flip flop 1175 which in turn is connected to a serial return circuit 1177 which is suitably enabled to return or send serial outputs from all parts of the on-chip JTAG circuitry back to computer JTAG card 1141 via output serial pin TDO.

In FIG. 50A a state transition diagram of TAP controller 1151 has one and zero signal values entered adjacent to each state transition arc. These are values of signal TMS at the time of a rising edge on signal TCK. The states of the JTAG TAP (Test Access Port) controller are described in "A Standard Test Bus and Boundary Scan Architecture" by L. Whetsel, *Texas Instruments Technical Journal,* Vol. 5, No 4, 1988, pp 48–59 and in above-referenced documents 08/108,775 and 08/087,020 and U.S. Pat. Nos. 5,084,874 and 4,872,169.

Turning to basic concepts recognized and utilized herein, emulation involves hardware support built around each circuit so that operations can be executed within the circuit while doing analysis in parallel as the circuit runs. Emulation permits the circuits to be run at full speed in real time as the emulator computer 1101 monitors the circuits and starts and stops them. The user defines and develops software in the environment of the target system. Put another way, emulation reads inputs from the board 1043 and produces outputs to the board as if device 11 were absent, for the purpose of determining appropriate software and operation signals. Ultimately, when the device 11 is supplied with the appropriate software resulting from emulation work, the device 11 operates in a manner which is compatible with the rest of the circuitry of board 1043. Advantageously, in the improved system disclosed herein, the device 11 is actually on the board and with the serial communication capabilities, all of the operations of device 11 are monitored directly from the device itself. In view of the extremely high speed of device 11, the device itself assists in its own emulation.

In a previous approach, a cable is terminated in a pin-plug that mates to a socket provided on the board in place of the emulated device. The socket introduces a noise issue. A socket may be impractical when a surface mount device is to be emulated, due to limited board space. Advantageously, device 11 is soldered onto board 1043 and emulation is mediated by the device itself.

The few pins utilized by the scan interface 1150 eliminate the need for conventional full pin-out target connectors and eliminate problems associated with cable reliability, transmission effects and timing differences. In this way, board 1043 can be probed with logic analyzers and oscilloscopes in the improved system without physical or electromagnetic interference from a heavy cable. Moreover, clock rates in excess of 20 megahertz for device 11 are so fast that previous emulation schemes may be incapable of emulating it.

Simulation as the term is used herein creates a software representation of the target board **1043** so that the entire board can be developed in simulation on simulator **1131** of FIG. **44** (or by running the simulator program on computer **1101**). In another aspect of simulation, when the device **11** is available but the rest of the circuitry for target board **1043** is incomplete, the simulator can mimic the planned complete board by serial scan upload or download from device **11** to computer **1101**, and then serial scan download or upload from computer **1101** to device **11** in substitution for the missing circuitry of board **1043**. In this aspect, simulation is accelerated by running the device **11** itself at full speed according to the improvements described herein. Even when computer **1101** runs at a slower speed than device **11**, simulation is effective to simulate peripherals which are accessed infrequently by device **11**.

Test as the term is used herein has four different areas. The first area—Device Test—is test of a device **11** itself before the device manufacturer ships it.

The second area of test is Device Verification—verification of full functionality of the device in every aspect.

The third area of test is Device Characterization. Characterization determines timings of the device to define exactly the way the actual manufactured device works.

The fourth area of test is User Test. In user test, the entire board is tested so that the functionality of device **11** in the context of the entire board **1043** is examined.

Returning to FIGS. **47** and **48**, each MPSD module has two scan paths. One of the scan paths is termed the MPSD data path which usually has numerous shift register latches S (or SRL) serially interconnected like a string of beads throughout the module. The second scan path is termed the MPSD control path which generally has fewer shift register latches and which selects which MPSD data paths are to be scanned. These scan paths are described in above-cited application Ser. No. 057,078 issued Aug. 22, 1989, (U.S. Pat. No. 4,860,290).

In FIGS. **49** and **51**, the improved emulation arrangement recognizes that device **11** is dividable into a few major areas which are clocked by different clocks when desired. These major areas are called clock domains or just "domains". The domains in a DSP device such as device **11** are suitably a CPU core domain, memory and peripherals (system) domain and an analysis domain. For another chip, the domains can defined in whatever manner is consistent with the parts of the chip that are to be sometimes clocked from different clocks. However, for modularity of chip design, emulation and test, the modules should usually be smaller units than a whole domain. This affords greater flexibility in designing other chips using the modules as building blocks, and reduces the time required to scan data into modules (the time is a nonlinear power function of the size of the modules).

Accordingly, it is contemplated that each domain usually include more than one module. In FIG. **49**, emulation adapter **1203** directs different clocks to the different domains or may supervise bit by bit transfers between the scan interface and a specific domain. Furthermore, adapter **1203** directs different MPSD control signals to the control paths of the different domains.

In FIG. **51**, the on-chip emulation blocks are further illustrated wherein JTAG control is wrapped around the emulation according to MPSD (Modular Port Scan Design). Principles of modular scan testing are also disclosed in coassigned U.S. Pat. No. 4,701,921 which is also incorporated herein by reference.

The JTAG control of FIG. **50** is indicated as JTAG control block **1201** of FIG. **51**. Emulation control according to MPSD is provided as a block **1203**. Test control block **1205** links JTAG to MPSD. Serial scan line **1207** enters the serial bit stream into any one or more of the selected areas of boundary scan area **1211** which includes BSR **1161** of FIG. **50** and scans the pin boundary of device **11**. The FIG. **51** domains—CPU core domain **1213**, system domain **1215** and analysis domain **1217** are shown in FIG. **50** and interface through the shift register latches of FIG. **47** to all of the various parts of the chip.

FIG. **52** provides a further perspective of the domains of device **11**. The CPU core domain **1213** includes the circuitry of FIGS. **1A** and **1B** in incorporated by reference U.S. Pat. No. 5,072,418. Analysis circuitry is connected to the CPU core as described more fully herein. The analysis circuitry includes condition sensors such as hardware breakpoint sensors for controlled stops and trace stack circuitry for real-time trace recordkeeping. The analysis circuitry is serial-scan accessible and designated the analysis domain **1217**. All peripherals including memory and serial and parallel ports are denominated as the system domain **1215**. For uniformity of description, JTAG control **1201** is regarded as a clock domain also in which test clock JCLK is active. Emulation control circuitry **1203** is a further domain of FIG. **52**. Special message passing circuitry **1216** is also included in the system or analysis domain, to even more fully use the host computer **1101** as an attached processor by interfacing the TIBUS to the serial scan line **1103** of FIG. **45**.

FIG. **53** shows a physical perspective of the various domains on the chip of device **11**. JTAG control **1201** interfaces with the pins via a serial boundary scan assembly including boundary scan register **1161** which allows all logic states at the actual pins of device **11** to be read or written. JTAG TAP controller **1151** and JTAG instruction register IR **1153** are provided on-chip. Test control **1205** and MPSD control **1203** are integrated into the circuitry. MPSD control **1203** serially interfaces with the domains for core **1213**, system **1215** and analysis **1217** for the device **11**. Bi-directional pins EMU0 and EMU1 are provided for external interfacing in addition to the four JTAG terminals **1221**. Combining JTAG testability interface technology with MPSD modular port scan with the additional pins EMU0 and EMU1 synergistically opens up capabilities for integrating emulation, software development, and manufacturing and field test processes.

A medical analogy assists in describing the overall concept of this remarkable emulation feature. Device **11** is analogous to a human patient on a medical operating table wherein a dialysis machine and an electro cardiogram are connected to different parts of the patient's body. While the patient's heart (functional clock FCLK) is pumping blood through the patient's torso and head (CPU core), the dialysis machine (test clock JCLK) is forcing blood through the vessels of the patient's legs (peripherals in system domain) in substantial isolation from the patient's heart. An electrocardiogram is attached to a separate part of the patient's body. All of these medical and physiological functions are operating at the same time so that the emergency medical needs of the patient are fulfilled in the shortest amount of time.

Correspondingly, device **11** can have its own system clock FCLK running at full speed to run part of device **11** while another part of device **11** is operated by a different clocking system JCLK under the JTAG/MPSD control and then in a third feature, the JTAG control can be controlling the JTAG boundary scan. Moreover, the parts of each chip are selectively fed FCLK or JCLK upon command, affording a dynamic control process. In this way, the development requirements of the device **11** are fulfilled in an integrated manner in the shortest possible time.

In FIGS. **52** and **53**, the development system capabilities for the preferred embodiments address applications development support in a fashion that allows the selection of a range of capability. Utilizing all of the disclosed development support hardware components provides development capabilities that include concurrency, ANALYSIS for on-chip breakpoint and trace, and realtime message passing MSGPAS8 between the emulation controller **1101** of FIG. **45** and the device **11** of FIG. **52**. Various sections of the hardware support of FIG. **52** can be included or omitted in cost sensitive applications. For example, a basic system would support basic MPSD (Modular Port Scan Design) emulation or at the extreme, no emulation and only test with or without a boundary scan.

The total development systems advantages in the areas of emulation, simulation, and chip speed measurement interrelate with various test and functional features. The preferred embodiments have three architectures, a) functional architecture, (e.g., CPU memory and I/O), b) test architecture including JTAG and MPSD serial scan-based testability circuitry and c) emulation/support architecture such as message passing circuitry, simulation features, and boundary scan test. The three architectures advantageously minimize complication of the CPU, memory and I/O functional architecture and maximize the synergism with test.

The support architecture of FIGS. **51** and **52** provides means to match cost and capability requirements through the life cycle of the device **11**. The support capability is deletable for devices created for some market segments. This partitioning does not affect the testability of the device by a test control **1205** of FIG. **51**.

In this way, a hierarchical scan architecture combines and improves a scan support/test front end with an MPSD interface as in FIG. **51**.

The architecture of FIG. **51** provides uniform interfaces for boundary scan **1211**, CPU core **1213**, analysis **1217** and memory and peripherals **1215** which allow the selection of capability based on the choice of blocks **1215–1217** used.

Table II below describes the levels of capability created when additional hardware blocks are added to the CPU and system domains.

TABLE II

	JTAG CONTROL	TEST CONTROL	EMUL. CONTROL	ANALYSIS	MSG. PERIPHERAL
MPSD Only					
MPSD Test and Emulation					
MPSD Test, Analysis, and Emulation				x	
JTAG Device Test	x	x			
JTAG Emulation Concurrency	x	x	x		
JTAG Emulation Concurrency with Analysis	x	x	x	x	
JTAG Emulation Concurrency, Analysis, Message Passing	x	x	x	x	x

Referring to FIG. **52** and recapitulating, support architecture views the device **11** as the following five distinct clocking domains in order to control domain data transfers with the scan clock (JCLK) and application execution with the functional clock (FCLK).

1) CPU core domain **1213**
2) Analysis domain **1217**
3) Peripherals, memory, interfaces, and busing (system domain) **1215**
4) Emulation control domain **1203**
5) JTAG port and boundary scan domain **1201**

There are two data path configurations, one for MPSD and one for JTAG. In the JTAG data paths of FIG. **50**, data is scanned to and from the device **11** through internal scan paths that are selected through a JTAG instruction register **1153**. A unique JTAG opcode for each path allows entry of and access to internal scan data.

The support architectures utilize two different clocks to support their operation. The two clocks are the functional clock (FCLK), and the scan clock (JCLK).

The emulation environment contemplates that some domains have a different clock source than others while test operation usually makes the device **11** fully synchronous and places all domains on the test clock JCLK. The clock separation provided by the domains of FIGS. **51** and **52** provides the ability to:

1) Connect all domains independently to either the JCLK or FCLK via the emulation control block **1203**.
2) Apply test clock JCLK to all domains for test.
3) Run any domain with functional clock FCLK and then scan its contents with JCLK.
4) Halt the CPU domain with a stop response compatible with functional clock FCLK and then scan the CPU domain on test clock JCLK while the system domain of FIG. **52** continues to run on FCLK.

Clock domains of FIGS. **51** and **52** and the emulation control **1203** allow independent selection of functional clock FCLK (chip clock rate divided by two) or scan clock JCLK (TCK pin of FIG. **50**). Each domain **1213**, **1215**, **1217** can have its clock individually selected while other domain selections are locked (unchanged). The selection process is a synchronized transfer of control between JCLK and FCLK in each domain. This transfer mechanism is located in the emulation control

1203 of FIGS. 49, 51 and 52 as discussed later hereinbelow.

FIG. 54 shows a further diagrammatic perspective of the registers of FIG. 50 wherein JTAG instruction register IR 1153 is selected for scan between the terminals TDI and TDO. The IR 1153 is decoded in FIG. 50 to access the other serial shift registers or scan paths when requested by the control card 1141 of FIG. 45 via the serial line 1103. These shift registers are the bypass register 1167, the boundary scan register 1161, the message peripheral 1216 of FIG. 52, the emulation control register 1251 and a pair of MPSD scan paths 1252 in the various domains and modules in the domains.

In FIG. 55, a scan line 1253 from FIG. 54 is denominated SCIN and this line is selectively connected to three scan paths 1252 wherein each of the three paths is internally split by internal MUX selection into an internal scan control path and a scan data path. A set of external MUXes 1261, 1263 and 1265 are controlled by lock signals LOCKS (lock system domain), LOCKA (lock analysis domain), and LOCKC (lock core domain) from emulation control 1203 to bypass all but a selected one domain, if any, for scan and execution purposes. A locked domain has the MPSD codes (discussed hereinbelow) which have been supplied to that domain be frozen for the period the domain is locked. When any one domain (e.g. analysis domain) is to be scanned, its corresponding MUX 1263 deselects line ASCIN (analysis scan in data) and selects line ASCOUT (analysis scan out). In this way serial scan bits entering line SCIN enter analysis domain 1217 on line ASCIN, leave analysis domain via line ASCOUT, and bypass the other two domains. Each of the other two domains is selected analogously. Scan output exits through line SCOUT at the bottom of FIG. 55.

For each domain, MPSD command code bits C0, C1 and CX in FIG. 56 enter each domain from control adapter 1203. These MPSD command code bits C0, C1 and CX are parallel, not serial, and are distinct from scan control signals and scan data signals entering a line SCIN for each domain in FIGS. 55 and 56. Particular operations in particular modules in a given domain are determined by the scan control bits passed into the daisy chained modules of a given domain on line SCIN, when a particular code C1,C0=01 sets the internal selection to receive line SCIN scan bits into scan control bit SRLs in each module of the domain.

C1 is a one when not scanning and a zero when scanning. CO and CX are sourced from the emulation control block adapter 1203 register 1251. When host computer 1101 detects Ready to Scan for all unlocked domains for a designated device on the target board 1043, C1 and CO are modified when in the Scan Data state JSDAT of FIG. 50A to make C1 and CO both be zero (C1,C0=00) to set the internal selection to receive line SCIN scan bits into scan data bit SRLs in each module of a selected domain. When C1,C0=01, then the internal selection is set to receive the SCIN scan bits into scan control SRLs as stated in the previous paragraph.

The command code lines C1,C0,CX, the scan in SCIN and scan out SCOUT lines, and clock lines are shown in FIG. 56. The scan control bits are scanned in on line SCIN to control and select one or more modules in a selected domain. Then scan data bits also on line SCIN are scanned into the selected domain into the selected one or more modules. Thus, the system affords individualized access of the MPSD scan data and scan control bits to the system domain 1215, analysis domain 1217 and CPU core domain 1213.

Identical nomenclature is applied to the domain inputs of FIG. 56 even though they are not connected together. Prefixes of S, A and C are applied to the nomenclature of the outputs of adapter 1203 that go to each of the domains. For example, AC1 is the C1 output for the A (Analysis) domain from the adapter 1203.

In the control 1203 clock switching circuits (1411, 1413, 1415 of FIG. 59) fed by FCLK and JCLK individually provide respective domain clocks to clock each of the domains separately or together as desired by the user. FIG. 56 also shows clock lines SCLK, ACLK, and CCLK going to domain clock inputs DCLK for system domain 1215, analysis domain 1217 and CPU domain 1213 respectively. Prefixes of S, A and C are denoted for the same lines at respective system, analysis, and CPU ports of adapter 1203 of FIG. 56.

The MPSD codes are tabulated in Table III:

TABLE III

C1	C0	CX	SCIN	SCOUT	REMARKS
1	1	1	X	1	Functional run, emulation logic held reset (strap) (7)
1	1	0	X	1	Execute a command which has been scanned in: Emulation run, device running (6)
1	1	0	X	1,0	Emulation run, device halted on one-to-zero transition (6)
1	0	1	X	1	Halt applied, device still running. (5)
1	0	1	X	1,0	Halt applied, device halted on one-to-zero transition (5)
1	0	0	X	Do	Scan pause (4) (Cause serial data transfer to pause)
0	1	X	Di	Do	Scan control path data (2,3)
0	0	X	Di	Do	Scan data path data (0,1)

Since there are two different MPSD scan paths in each module (FIGS. 47 and 48 being simplified suggest both paths with only one serpentine loop), the C1,C0 codes act as a selection code. C1,C0=01 selects the control path, whence control bits are scanned in on line SCIN and control information scanned out on SCOUT. C1,C0=00 selects the data path, whence data bits are scanned in on line SCIN (FIG. 55) and data developed in device 11 scanned out on SCOUT.

When C1=1 (command code C1 active), then control line CX is available for defining further codes for a whole domain as tabulated above.

The scan out line SCOUT has a dual function. In a first function SCOUT serves as a handshake or interrupt by representing whether the device is running or halted in response to the command entries 7, 6 and 5. In the second function, it serves as a line for scanning out serial data in the last two tabulated codes. Transitions from shift operations (0,1,2,3) to execute operations (5,6,7) or from execute to shift, preferably include a pause (4) to halt (5) or halt-to-pause sequence. This is not mandatory, but can be used to effect orderly behavior of device 11 internal buses and state machines in such transitions.

It is of interest that Pause state 100 electrically borders both the command codes above it (herein called Execute codes) in the table as well as the command codes to scan (herein called Scan codes), below it.

In FIG. 57, the adapter or emulation control block 1203 is used to independently manage the clock signals that it routes to the CPU, analysis, and system domains.

1213, 1217 and 1215 respectively. Block 1203 also applies a sequence of the command code triplets (CO, C1, CX) to generate emulation and simulation functions. Each domain has modules, such as 1301.1 and 1301.n in system domain 1215, 1303.1 and 1303.n in CPU core domain 1213, and 1305.1 and 1305.n in analysis domain 1217. Associated with each module is a test port as described in application Ser. No. 057,078 issued May 22, 1989 (U.S. Pat. No. 4,860,290).

Connected to each test port is mode conditioned stop logic circuitry 13098, 1309C and 1309A in the domains respectively. The modes are established by a mode register 1311 which is scanable in FIGS. 54 and 57 to establish the type of stop and any other desired mode characteristics for the domains. The mode conditioned stop logic circuits 13098, 1309C and 1309A are respectively fed by MPBD decoders 1313B, 1313C and 1313A that have multiline outputs to the stop mode conditioned logic circuitry.

Scan control 1149 scans in an initial state (test vector) into the registers of device 11 to set up a test or emulation sequence. This is done with all domains locked, meaning that test clock JCLK is applied for scan purposes. When all data and control registers are loaded, circuitry 1149 sends a START signal to adapter 203. The CPU domain, for an example, is unlocked, meaning that it is started running on functional clock FCLK.

Then in a stop feature, the analysis domain 1217 is scan programmable to direct the CPU domain to stop on a predetermined condition. When the predetermined condition occurs, a signal ANASTP (Analysis Stop) is sent to the CPU domain 1213 to make it stop according to the stop mode established for it in mode register 1311 and mode conditioned stop circuitry 1309C. DONE circuitry 1363 detects when the stop is completed and signals back to adapter (emulation mode control) 1203 to lock the CPU, for example, by putting it on test clock JCLK instead of functional clock FCLK. The DONE signal is generated when all instructions in a CPU pipeline are executed and all currently pending memory access cycles are completed. Another definition of DONE may of course be used or mode-selected. Then using test clock JCLK, the important registers of the domains can be scanned out for recordkeeping, display and study at host computer 1101 of FIG. 45.

FIG. 58 illustrates in even further detail an example of process steps by which the scan control 1149 including IR 1153, adapter 1203 including ECR (emulation control register) 1251, and host computer 1101 cooperate to enter and perform sequences of commands on-chip.

Operations in FIG. 58 commence in a step 1321 with a reset STRAP state of FIG. 50A in the scan control 1149 in FIG. 50. Output JSTRAP of TAP controller 1151 of FIG. 50 becomes active and in a step 1322 sets two JMODE bits in ECR 1251 of FIGS. 54 and 59. Adapter 1203 jams a functional 111 command code and sets the domain clocks in a step 1323.

Next in a step 1324, the host 1101 sends TMS signals to scan control 1149 to reach the FIG. 50A TAP controller state "Select-IR-Scan" and then scans ECR select into the IR 1153. Further in step 1324 the host sends more TM8 signals to the TAP controller to reach the state "Select-DR-Scan". This means that the scan controller is ready to accept scan into a DR (data register) identified just before as the ECR 1251 by the host to the IR 1153. Into ECR 1251 the host 1101 then scans two-bit portions C0,CX of the triplets for RUN and HALT of Table III. Lock bits are also loaded to unlock all of the domains. The lock bits are scanned for this purpose into a FIG. 59 ECR 1251 portion called LOCK 1351.

In a succeeding step 1325, host computer 1101 sends TM8 signals to the scan control 1149 to reach the TAP controller state "Select-IR-Scan" again. This time the host then scans MPSD path 1252 select into the IR. In a step 1326, more host TM8 signals are sent to reach the state IDLE in FIG. 50A. Hardware in FIG. 50 thereupon produces the signal START to activate the code state machine 1381 in the adapter 1203. A decision step 1327 determines whether the scan interface 1149, 1203 is ready. When ready, the host 1101 scans an ECR select into the IR 1153 in a step 1328, followed by more TM8 to reach the IDLE state in a step 1329. This deactivates the start signal and permits scan-out of domain information by DR scan from the MPSD path 1252 in steps which are omitted for conciseness from FIG. 58.

When the interface is again ready in a step 1331 operations proceed to a step 1332 and a step 1333 to select and scan the ECR 1251 JMODE bits JMODE1 and JMODE0 with logic I and logic zero respectively to enable concurrent emulation. Then in a step 1334 the host 1101 sends TMS signals to reach the IDLE state of FIG. 50A for lock. When the interface is ready after a step 1335, the host 1101 scans command and lock bits into ECR 1251 in a step 1336. Next in a step 1337, the host 1101 sends TMS to select scan to the IR, scans a MPSD path select into the IR. Next in a step 1338, the host 1101 sends more TMS to reach the IDLE state of FIG. 50A whereupon hardware of FIG. 50 initiates the START signal to the adapter 1203 code state machine 1381. A step 1339 has the host wait until the interface is ready, whereupon operations loop back to step 1333 to have the host 1101 scan further commands into the ECR and then start the on-chip emulation circuitry to run the chip in real-time.

It is to be understood that the description of steps is by way of example only. Data and control information are scanned into and out of the domains on test clock JCLK, and the domains are independently and selectively started on functional clock FCLK and stopped, in extensive sequences to accomplish emulation, simulation and test functions with a wide degree of flexibility as circumstances of the development, manufacturing and field environments dictate.

This remarkable capability of scanning in emulation data and retrieving it from each domain, and further of individually clocking the domains with either the real time functional clock FCLK or a test clock JCLK is mediated by the emulation control 1203 in response to the host computer 1101 and improved JTAG circuitry of FIG. 50. The emulation control 1203 is illustrated in greater detail in FIG. 59.

In FIG. 59, serial scan bits enter the emulation control register ECR 1251 which is subdivided into a shift register LOCK 1351 for holding bits to lock and unlock domains, a first C0,CX control code shift register named CODA 1353, a second C0,CX control code shift register named CODB 1357, a shift register 1359 associated with event manager circuitry 1365, and a two-bit register JMODE 1360. These registers are compactly illustrated in FIG. 59A. Thus, the serial scan enters on a scan line SIN, passes serially through the shift registers 1351, 1353, 1357, 1359 and 1360 until all the shift registers are loaded. Any serial scan output is scanned out via serial line 1361 SCUT.

A set of selector logic circuits **1371**, **1373** and **1375** of FIG. **59** supply signals on lines as shown in FIG. **56** to the domains for the core **1213**, system **1215** and analysis **1217**. Also, individualized domain clock lines DCLK of FIG. **56** are supplied respectively with clock signals SCLK, ACLK and CCLK in FIG. **59**.

A code state machine **1381** controls a two input MUX **1383**. MUX **1383** selects the C0,CX two-bit contents of shift register **1353** or **1357** and loads them into an enabled one of three flip flops **1393**, **1395** and **1397**. A lock control circuit **1401** operating under the control of lock shift register **1351** and code state machine **1381** sends lock signals to disable or enable each of the flip flops **1393**, **1395** and **1397**, depending on which selected domain **1213**, **1215** or **1217** is to be unlocked while other domains are locked.

Each flip flop has three sections for C0, CX and a clock control signal DSCS−.

Three clock control signals, each independent of the others and all designated DSCS−, are supplied from the flip flops **1393**, **1395** and **1397** to respective clock control circuits **1411**, **1413** and **1415** which provide the individualized domain clock outputs—core clock CCLK, analysis clock ACLK and system clock SCLK.

Test codes from the TMS, TDI and TD0 lines from host **1101** enter FIG. **59** on three lines **1421** to selection logic **1371**, **1373** and **1375**. Command codes for each domain can be derived directly from the test codes on line **1421**, thus overriding the code state machine feature. This option is selected by scanning JMODE register with "00" (both bits zero). Thus, the preferred embodiment is accommodates direct host control of the domains, wherein the latter is the JMODE 00 option.

If a switch of any domain **1213**, **1215**, **1217** between JCLK and FCLK is in progress under control of any of the clock controls **1411**, **1413**, **1415**, then code state machine **1381** should be temporarily stopped. This is accomplished by a disabling input low active SWINPROG-(Switch In Progress) which is fed from a logic network **1425** in response to the LOCKC, LOCKA and LOCKS inputs from lock control **1401** and from three clock domain signals GCTDC, GCTDA and GCTDS. The latter three signals signify Good Clock This Domain for each of the domains—Core, Analysis and System, respectively.

In FIG. **59**, the control block **1203** functions are programmable and allow one to:

1) Apply MPSD codes from one of two preloaded registers, CODA **1353**, and CODB **1357** as directed by a code state machine **1381**;

2) Program state machine **1381** operations via REVT (Register Event) register **1359** to respond to stimuli including:

a) START from scan control **1149**

b) DONE from CPU core **1213**, **1363**

3) Control the clock switching from FCLK to JCLK (with a code 00 in both registers **1353** and **1357**) and vice versa for each domain, via LOCK register **1351**; and

4) Lock domains in their current state while operating with other domains.

The control block **1203** allows the following clocking options:

a) The orderly switch of the domain clock lines between JCLK and FCLK clock pulse sources.

b) Domain clock lines may be locked in the current state by lock bits on a scan data path.

c) The JTAG strap (reset) state or MPSD strap state cause the functional clock to be selected.

d) Test mode selection allows the entire chip to be driven by JCLK.

This clocking selectability allows configuration of a fully synchronous chip for test, and the ability to scan any one or more of the domains, or the whole chip with data transfers being synchronous to the JCLK.

The operation of code state machine **1381** is now further described. When the JTAG IR (Instruction Register **1153**) is loaded with a scan path select command for path **1251**, a line ECRSEL feeds a signal to state machine **1381** whereupon the state machine **1381** enters a lock state. This allows the registers **1351**, **1353** and **1357**, the event manager register **1359**, and JMODE register **1360** to be changed without disturbing the MPSD codes and clocks supplied from flip flops **1393**, **1395**, **1397** and selection circuits **1371**, **1373** and **1375** to the domains **1215**, **1213** and **1217**. Lock register LOCK **1351** holds bits which selectively cause the CPU, analysis and system domains **1213**, **1217** and **1215** to be frozen in their current configuration and state in response to MPSD command codes presently applied to them from flip-flops **1393**, **1395**, **1397** and logic **1371**, **1373** and **1375**.

The START signal for state machine **1381** of FIG. **59** is generated in the circuit of FIG. **50** when a scan data path select signal is present and TAP controller **1151** reaches the JTAG IDLE state of FIG. **50A**. The JTAG interface of FIG. **50** becomes passive and the dynamic operations of circuit **1203** of FIG. **59** commence.

In FIG. **50** the START signal is generated as follows. A path decoder **1168** connected to instruction decoder **1155** qualifies AND gate **1170** when its OR gate **1172** signals that the MPSD scan data path **1252** of FIG. **54** or MPSD scan control path are selected. An IDLE line from TAP controller **1151** is connected to another input of AND gate **1170**. When the IDLE state occurs with gate **1170** qualified for path, AND gate **1170** signals a handshake synchronizer **1169** to supply START to code state machine **1381** of FIG. **59**.

Handshake synchronizer **1169** is provided because emulation control **1203** is operated on pulses of functional clock FCLK at times while the JTAG circuitry operates on the test clock JCLK. Handshake synchronizer **1169** includes two state machines to control the generation of the START signal to the code state machine **1381**. One state machine is in the JTAG environment of FIG. **50** and the other is in the MPSD control environment of FIG. **59**. In this way, the clock boundary is crossed.

The emulation control block **1203** of FIG. **59** serves to create MPSD control codes for the MPSD domains **1213**, **1215** and **1217** which perform the necessary emulation, simulation and test functions. A general programmable implementation is illustrated, although a fixed hard coded implementation is also feasible. Moreover, a microcoded control ROM (CROM) implementation of the circuitry of FIGS. **50**, **51**, **57** and **59** is an alternative embodiment contemplated in the practice of the invention.

The code state machine **1381** controls the generation of MPSD code sequences to the domains. The clock control circuits **1411**, **1413**, and **1415** each include a state machine that controls the switching of the clocks of the domains between JCLK and FCLK in an orderly way before allowing a new MPSD code (C0, C1, CX) to be applied to the domains. "State Machine" is used in

the computer science sense of the phrase to denote any software or hardware based circuit that is represented by a state transition diagram that has at least two states. This state machine approach minimizes the number of JTAG opcodes assigned to emulation functions as well as greatly simplifying the MPSD interface.

From one perspective, the Pause command code (C1,C0,CX=100) of Table III is a default state or "anchor" to which the other states relate. The code state machine and registers CODA and CODB operate with their C0,CX contents to alter the Pause state to HALT (101), CNTRL (110) or FUNC (111) of Table III. When a scan into a domain is desired, the code state machine **1381** is directed by host **1101** software to enter C0,CX=00 in both CODA and CODB, and in this way a Pause (100) is applied to the device **11**.

From a clock switching point of view, a transition from HALT **101** to Pause **100** causes a functional clock FCLK to JCLK test clock switch-over. The switch-over occurs in the Pause state applied to the domain inputs C1,C0,CX by the logic **1371**, **1373** and **1375**.

A transition from Pause **100** to any of FUNC **111**, CNTRL **110** or HALT **101** causes the interface to freeze in Pause **100** until all unlocked domains switch to functional clock FCLK and thereupon operate on FCLK. Thus all clock switching occurs with the **100** Pause code applied.

The host **1101** software is programmed to operate this interface, for example, on the assumption that when the Pause (100) code is present on all unlocked domains, and it may now load the MPSD path select into the IR **1153** and then scan MPSD data or control bits into a selected domains. The logic **1371**, **1373** and **1375** of FIG. **64** responds to the path select as data control to set a ScanData (00x) or Scan Control (01x) MPSD command code for the domains.

A Scan Ready SCANRDY interface-ready bit can be set in IR **1153** for use by host **1101** software to signify that all domains unlocked have no switch in progress and Pause **100** is applied to them, so that it is all right to scan data or control bits into the domains.

Since the logic **1371**, **1373** and **1375** responds to the path selects when scan codes **00x** or **01x** are needed, it should be apparent that the code state machine **1381** and registers CODA and CODX job primarily involves delivering code bits C0,CX from among the group **00** Ready to Scan, **01** Apply HALT, **10** Apply CNTRL and **11** Apply FUNC. Code bit C1 of Table III is a one (1) due to host **1101** software not attempting to scan while the state machine is applying a code C0,CX of **01**, **10** or **11**. Thus, the MPSD command codes C1,C0,CX are respectively constructed—**101** HALT, **110** CNTRL and **111** FUNC.

FIG. **60** shows a schematic diagram of code state machine **1381** and its event manager **1365**. Code state machine **1381** includes two interconnected SRLs **1451** and **1453** that sequence through a state transition diagram of FIG. **61** that has three states—LOCK, CODEA and CODEB and transitions T1, T2, T3, T4 and T5 between the states. The respective outputs of the SRLs **1451** and **1453** are regarded as the MSB and LSB (most, least significant bits) of a two bit digital signal. The MSB represents the LOCK state, and LSB high and low respectively represent the CODEA and CODEB states. Event manager **1365** signals to SRL **1453** that if the state machine is in the CODEA state, then a transition to the CODEB state is requested.

A third SRL **1455** of FIG. **60** is connected between SRL **1451** and lock control **1401** of FIGS. **59** and **63**. An OR gate **1461** has inputs connected to the output of SRL **1451** and to switch-in-progress signal SWINPROG from logic **1425** of FIG. **59**. OR-gate **1461** has its output connected to series-connected inverters **1463** and **1465**. The inverters are connected respectively to CAPTURE and HOLD inputs of SRL **1453** to provide state transition inputs related to the state of SRL **1451** and SWINPROG. OR-gate **1461** itself supplies a control hold CNLHOLD output to circuit **1383** of FIG. **59**, to which circuit line LSB is also connected. Event manager **1365** is connected to an input of SRL **1453**.

A NAND gate **1471** supplies a SET input of SRL **1451** in response to two inputs ECRSEL (emulation control register select) and START from FIG. **50**. An AND gate **1473** supplies a RESET input of SRL **1451** in response to MPSDSEL (MPSD data or control scan path select) and START.

Event manager **1365** includes a set of logic circuits connected to signals CNTBRW, DONE, EMU1 and EMU0 and any other analysis, core status, or other signals which the skilled worker elects. Signal CNTBRW is counter borrow from analysis block circuitry discussed in FIG. **69**. DONE signals completion of a stop as discussed in connection with FIG. **58**. DONE is suitably provided to the state machine as the differentiated trailing edge of a CPU core signal that indicates the core is running, so that just when the core ceases running, the DONE signal is provided. Pins EMU1 and EMU0 carry signals of same designation originating internally or externally of the chip for emulation signaling.

Shift register **1359** includes scannable register-event bits REVT3, REVT2, REVT1 and REVT0. REVT3 is a scanable bit inverted by an inverter **1483** to signal a NAND gate **1481** (which operates as a low-active OR) that feeds SRL **1453**. REVT2, REVT1 and REVT0 are control bits cause event manager **1365** to selectively ignore or respond to signals CNTBRW, DONE, and EMU1 respectively. For this purpose, respective NAND gates **1485**, **1487** and **1489** have their outputs connected to NAND gate **1481**. One input apiece of NAND gates **1485**, **1487** and **1489** is connected to REVT2, REVT1 and REVT0 respectively. A second input apiece of said NAND gates is connected to CNTBRW, DONE and EMU1. Additional NAND gates for EMU0 and other signals are readily providable, as are further event register cells in shift register **1359**.

The circuitry of FIG. **60** is inventively configured in any appropriate manner to implement the inventive methods and structures. For instance in this embodiment, generally speaking, the code state machine **1381** locks the MPSD control code currently applied to the domains when the emulation control path ECR **1251** is selected by the instruction register IR **1153** and a START pulse is generated. The code state machine **1381** exits the lock state upon a START pulse and a MPSD scan path selected and can apply one of two MPSD control codes at times responsive to occurrence of a predetermined condition of CPU core or other event preprogrammed by REVT bits before returning to LOCK state. Also, the code state machine, as shown in FIG. **61** can automatically alternate between the first and second control codes before returning to LOCK state. While only two codes have been shown for illustration, it is apparent that any number of control codes

can be scanned into or otherwise stored in corresponding shift registers analogous to 1353 and 1357 of FIG. 59, and a code state machine can be provided to have corresponding states for each of the codes with circuitry to implement transitions between the states to apply all codes in programmable sequences depending on various operational conditions. Some pairs of MPSD control codes that are of particular interest for automatic entry into the domains are tabulated in Table IV:

TABLE IV

CODA	CODB		Operation Performed
Run 10	Halt 01	a)	Execute program portion: Analysis domain determines when to stop; stop mode determines how to stop.FCLK
Pause 00	Pause 00	b)	Module Setup: download from host memory to SRLs in scan control path of a domain to select and deselect modules for subsequent data scan; upload from domain scan control path to host. JCLK See NAND 1585 Fig. 64
Halt 01	Pause 00	c)	Load machine image via scan in (b); then load CODA and CODB. Start with Halt applied, machine then signals DONE to advance state machine to select CODB; now scan in next sequence. Repeat for each word of memory, for example.
Pause 00	Pause 00		Download from host memory to SRLs in scan data path of a domain by b; upload from domain to host; Lock 1351 determines which domain is loaded. Loading Analysis domain determines when to stop in Run/Halt above. Uploading from analysis retrieves PC trace stack. Loading Core establishes program point from which to being Run. Uploading Core recovers CPU core state when halted. Loading/uploading System domain initializes/recovers System domain state. See NAND 1583, FIG. 64. and control path select.

In the state transition diagram of FIG. 61, some exemplary Boolean equations for the transitions T1–T5 are as follows:

T1=LOCK & NOT SWINPROG & START & MPSDSEL

T2=CODEA & NOT SWINPROG & ((REVT1=1 & DONE)+((EMU1+EMU0) & REVT0=1)+-(REVT2=1 & CNTBRW)+(REVT3=1))

T3=CODEB & ECRSEL & START

T4=CODEA & ECRSEL & START

T5=CODEB & NOT SWINPROG & START & MPSDSEL

The CPU provides a run signal (RUN) the trailing edge of which is designated DONE and used in the T2 equation.

SWINPROG is the indication that any of the clock domains are in the middle of clock transition.

START is set on the second clock cycle of the entry into the IDLE state with the device scan data or control path selected.

In words, the code state machine 1381 makes transition T1 in FIG. 61 from the LOCK state to the CODEA state provided the state machine is in the LOCK state, the START signal is present, MPSDSEL is present and there is no clock switching in progress. Transition T2 from CODEA to CODEB state occurs if the state machine is in CODEA state, there is no clock switching in progress and the event manager 1365 so requests. Transition T3 from CODEB to LOCK state occurs if the state machine is in CODEB state, the host 1101 has loaded an ECR request into JTAG IR 1153, and START is present. Transition T4 from CODEA to LOCK state occurs if the state machine is in CODEA state, the START signal is present, and the host 1101 has loaded an ECR request into JTAG IR 1153. Transition T5 from CODEB to CODEA state occurs if the state machine is in CODEB state, and there is no clock switching in progress, MPSDSEL is active and a START signal is present.

The three clock control circuits or clock switches 1411, 1413, 1415 work in tandem with the code state machine 1381 and circuits 1383, 1393, 1395, 1397 and 1401. Each clock control circuit 1411–1415 of FIGS. 59 and 65 supervises the transitions between clocks FCLK and JCLK as mandated by the MPSD codes in the CODA and CODB registers 1353 and 1357 selected by the code state machine 1381. When the code state machine 1381 points to a register CODA or CODB containing a different clock source for that state, the clock switch 1411, 1413 or 1415 corresponding to the unlocked domain selected by LOCK register 1351 and lock control 1401 locks the code of the old state by signaling NOT GCTD (Not Good Clock This Domain) to make SWINPROG active and synchronously switches the clocks. When the new clock pulses have been applied for at least one clock cycle from the new clock source, the clock switch then signals GCTD to release SWINPROG and allow circuits 1383 and a circuit pair such as 1397 and 1371 to pass the new control C0,CX control code to the enabled domain as described in more detail next in connection with FIGS. 62–65.

In FIG. 62, selecting circuit 1383 has two MUXes 1501 and 1503 for respectively selecting the C0,CX control code from register CODA or CODB in response to signal LSB from the code state machine 1381 of FIG. 59. The C0,CX codes in register CODA are designated CODA0 and CODAX, and in register CODB are CODB0 and CODBX. The selected code is held in a pair of SRLs 1507 for CO and 1509 for CX in response to control hold signal CNLHOLD fed to respective CAPTURE and HOLD inputs via inverters 1511 and 1512.

SRLs 1507 and 1509 thus latch in the new code CODB selected when state machine 1381 makes the T2 transition to CODEB state. However, this new code is not instantaneously sent to its intended domain. First it is checked by a circuit 1514 to determine whether the code implies functional clock FCLK or test clock JCLK for use therewith. In other words, the circuit advantageously determines by itself which clock is needed, and does not require an optional bit that could be provided by user to define a clock request. In this exemplary embodiment circuit 1514 is an OR-gate that selects test clock JCLK when both CO and CX are low (MPSD code for Pause). (MPSD code bit C1 is held at one by NAND logic 1563, 1573, 1581, 1583,1585 of FIG. 64). The clock request from circuit 1514 is held in

SRL 1513 in response to CNLHOLD, and SRL 1513 produces a clock select output nclksel to lock control 1401 and each circuit 1393, 1395 and 1397 of FIG. 59.

Circuits 1393, 1395 and 1397 (only 1397 shown in FIG. 62) identically include SRLs 1517, 1519 for MPSD command codes CO and CX and an SRL 1523 to hold a clock select signal. These circuits hold previously entered MPSD commands and clock select information to control all of the domains. When any given one domain is to be updated, the corresponding one of circuits 1393, 1395 or 1397 is updated while the other two of them retain their information. SRLs 1517, 1519, and 1523 each have an input connected to a corresponding output of SRLs 1507, 1509 and 1513. Data from circuit 1383 is strobed into a selected one of circuits 1393, 1395 or 1397 depending on which of them is unlocked by lock signal LOCKS, LOCKC or LOCKA. Moreover, it should be apparent that LOCK 1351 can have bits to lock or unlock some or all the domains at once. When scan of more than one domain is desired, the bits are appropriately set, and host computer 1101 can update all domains concatenated by MUXes of FIG. 55 at once.

Simply put, the C0,CX codes are delayed by a clock cycle to the domains so that the clock information that is implicit in C1,C0,CX of Table III can be examined and acted upon prior to the code being applied. If action is necessary, then Switch In Progress SWINPROG inhibits the code from being applied to the domain until the clock is switched.

In FIG. 63 the lock control circuit 1401 includes three OR gates 1531, 1533 and 1535 each having first inputs connected respectively to lines LOCKRQS, LOCKRQC and LOCKRQA from LOCK register 1351 of FIG. 59, and outputs connected respectively to lines LOCKS, LOCKA and LOCKC of FIGS. 59 and 62. These three OR gates each have second inputs connected together to the output of an OR gate 1541. A first input of OR gate 1541 is connected to the output of an AND gate 1543. A second input of OR gate 1541 is connected to the output of SRL 1455 of FIG. 60. The AND gate 1543 has two inputs connected to SWINPROG and nclksel. In this way, LOCK register 1351 bits override any other signals when the bits call for locking the domain (test clock JCLK only). However, if register 1351 unlocks any one or more domains (calling for functional clock FCLK to each such domain), each such domain can be locked internally by either the delayed locking MSB output of SRL 1455 or a conjunction of switch in program SWINPROG and clock select nclksel high. Every unlocked domain is locked without need of further selection circuitry by having OR-gate 1541 send locking request indiscriminately to all three OR-gates 1531, 1533 and 1535.

In FIG. 64 identical circuits 1371, 1373 and 1375 (1371 being shown) are respectively connected to circuits 1397, 1395 and 1393. For example, the output of SRL 1517 and the output of SRL 1519 of FIG. 62 are connected by two lines 1398 to a first input each of NAND gates 1551 and 1553 of circuit 1371 of FIG. 64. Circuit 1371 supplies the MPSD three wire control signals for the system domain (SC0, SC1, SCX) at the outputs of three NAND gates 1561, 1563 and 1565. The output of each NAND gate 1551 and 1553 is connected to an input of NAND gate 1565 and 1561 respectively. When JMODE =00, a code conversion logic block 1205 is activated to convert three wire test codes on lines TMB, TDI and TDO of FIGS. 50 and 59 and produce two active-low outputs TC0- and TCX- to an input of each of NAND gates 1561 and 1565. This block 1205 is enabled by a low active OR 1571 supplied from the two bits of JMODE register 1360 of FIG. 59. Code conversion logic block 1205 thus converts to MPSD code from any three-wire testability code scheme other than the MPSD codes of Table III, and thus increases the flexibility and adaptability of the system.

When the JMODE1 or JMODE0 bit is high, code conversion is disabled. Instead, for example when JMODE1 bit is high, a second input of each of NAND gates 1551 and 1553 is qualified. When both JMODE bits are high, a NAND gate 1573 supplies a low output indicative of STRAP state (allows device to run, effectively disables emulation and testability features). This low output forces high outputs by all three NAND gates 1561, 1563 and 1565 corresponding to the (111) MPSD control code 7 for functional run.

NAND gate 1563 has a further input connected to the output of a NAND gate 1581 that in turn has two inputs respectively connected to an output of each of two NAND gates 1583 and 1585. The output of NAND gate 1585 is also connected to an input of NAND gate 1561. A LOCKS—low-active line qualifies NAND gates 1583 and 1585 when the particular domain (system here) is to be unlocked (JCLK select). When the TAP 1151 of FIG. 50 is in the JSDAT state of FIG. 50A for scan input, a line JSDAT from the TAP goes high in FIG. 64, qualifying both NAND gates 1583 and 1585.

The role of NAND gates 1583 and 1585 is to specially determine the first two MPSD control codes C1 and C0 (SC1,SC0 for system domain) when the host computer 1101 has sent TMS signals to put TAP in the data scan JSDAT state, thereby telling the circuits that the host 1101 wants to scan MPSD data in a first case or to scan MPSD control bits in a second case. In the first case is in state JSDATA and IR 1153 is already loaded by host 1101 with MPSD data select that selects the MPSD path 1252 of FIG. 54. The system domain, in this operational example, happens to be locked, but must be unlocked to allow scan. In FIG. 64 line MPSDDATA goes high and MPSD CONTROL is low since MPSD control SRLs are not desired as the path for scan. NAND gate 1583 output goes low, forcing NAND gate 1581 output high. Since the TAP is not in the STRAP state, JSTRAP- is inactive high at NAND gate 1563 and SC1 goes low. All inputs of NAND gate 1561 are high, forcing SC0 low. SC1, SC0=0,0 defines the MPSD data scan command for this domain, which is precisely what is desired.

In the second case, the host 1101 has loaded IR 1153 with the MPSD control path select, so IR 1153 selects the MPSD path 1252 as in the first case. However, this time IR 1153 decode sets MPSD CONTROL line active in FIG. 64 and leaves MPSD DATA line low. The domain is locked, but must be unlocked to allow scan. Inspection of NAND gates 1583, 1585, 1561, 1581 and 1563 shows that the code SC1,SC0=01 is produced. In this way the desired MPSD control scan command is defined for the domain.

As thus described, MPSD scanning of CPU, analysis, and system domains involves slaving these domains to the JTAG environment 1149 by providing a scan control or scan data code at the domain interface with the domain connected to JCLK test clock.

In FIG. 65, identical clock control circuits 1411, 1413 and 1415 (1415 being shown) include NAND gates 1601 and 1603 supplied with functional clock FCLK and test

clock JCLK. A NAND gate **1605** supplies domain clock output (e.g. SCLK for clock control **1415**) has inputs connected to the output of NAND gates **1601** and **1603** respectively. NAND gates **1601**, **1603** and **1605** effectively act as a clock MUX (multiplexer) controlled by respective second inputs **1607** and **1609**.

Physically, on the chip **11** it is contemplated that clock lines **1611** and **1613** and switching elements **1601,1603,1605** for functional clock FCLK and test clock JCLK be separated or isolated from the rest of the adapter **1203** of FIG. **59**. In other words, FIG. **59** shows a diagrammatic and conceptual close relationship of clock control circuits **1411**, **1413** and **1415** to the rest of the adapter **1203**, but as a matter of chip design, it is believed preferable to isolate the circuits physically on the chip. Control conductors from the part of the circuits **1411**, **1413**, and **1415** in the adapter section of the chip would be routed over to the clock generator elsewhere on the chip to the physical switching elements to select which of the two clocks is routed to each domain.

Advantageously, the clock control circuits **1411**, **1413**, and **1415** cause part or all of the device **11** to run in the functional mode, accurately exercising the device at real-time rate for the purposes that it is programmed to accomplish. On the other hand, scan is operation of actually sending bits serially into and out of the machine to establish the machine state, and recover an image of a subsequently changed machine state. Scan clock JCLK is preferably used to enter and recover the serial data for an emulation host computer **1101**. Put another way, it can be important for the domains to accept a clock which can enter the bits in the domains at the rate which they are being sent to device **11**. In this way complications of synchronizing functional clock with some external clock are eliminated.

Also, the circuitry can support bit-by-bit asynchronous transfers at a low scan rate. In the bit-by-bit approach, test clock JCLK is a lower rate than FCLK functional clock. A one clock width pulse corresponding in width to a pulse of FCLK is then ANDed with the scan or substituted for it in FIG. **65**.

Further in FIG. **65**, assume that lines **1607** and **1609** are high and low respectively to select functional clock FCLK, that the circuit has stabilized in response to Domain Switch Clock Select DSCS—inactive high. Then SRL **1523** of FIG. **62** is caused to drive DSCS—active low to select the test clock JCLK. The GCTD signal is immediately driven low by a network of three NAND gates **1621**, **1623** and **1625**. DSCS—is connected to an input of NAND gate **1621** and its complement via an inverter **1627** to an input of NAND gate **1623**. Second inputs of NAND gates **1621** and **1623** are respectively connected to lines **1607** and **1609**. The outputs of NAND gates **1621** and **1623** are connected to respective inputs of NAND gate **1625** which produces GCTD output.

With DSCS- now active low, inverter **1627** produces a high that forces a NOR gate **1629** to produce an output low. This low propagates through serially connected D flip flops **1631** and **1633** in two clock cycles of functional clock FCLK, driving line **1607** low and shutting off FCLK from the domain. However, test clock JCLK is not yet supplied to the domain. The low on line **1607** now qualifies a NOR gate **1639** for JCLK selection purposes. Since DSCS- is active low, NOR gate **1639** output goes high. This high propagates through serially connected D flip flops **1641** and **1643** in two clock cycles of test clock JCLK, driving line **1609** high and admitting test clock JCLK to the domain via NAND gate **1605**. The line **1609** high simultaneously drives qualified NAND gate **1623** output low, forcing GCTD output high to signal Good Clock This Domain. Switching from JCLK back to FCLK occurs by an analogously reverse process in circuit of FIG. **65**.

Returning to general considerations, the development system capabilities are composed of ANALYSIS of bus traffic, inspection and modification of the machine state, execution of the user's program, observation of PERIPHERAL operation, and reading and writing of the MEMORY. There is a basic set of capability provided by MPSD which is supplemented by additional concurrent capability when the JTAG front end is added to the system.

In the preferred embodiment, a feature called Mode Driven Stops herein involves establishing one of six stop modes and then providing decode circuitry so that in response to the selection of any of these particular modes, the particular type of stop is effectuated as described hereinabove. Mode driven stops are a particularly advantageous improvement because they allow for example, the development engineer to shut down a processor in one way when simulating peripherals and to stop the processor in a different way when doing emulation braking. For example, in emulation braking, it may be desirable to stop the CPU, but not stop the peripherals. An emulation counter preferably continues counting so that the sample rate is correct on the target board **1043** of FIG. **45**, but it is desirable to stop the CPU so that information can be scanned out or parallel accessed between samples. On the other hand, in the case of clock stepping it is desirable to stop all domains. Also in the case of simulation of peripherals, it is desirable to stop all domains because the peripheral is being simulated.

Thus, development system capability is implemented as a basic set of functions available through the interface which stops the CPU core to access the Analysis, CORE, Peripheral, Memory, or Interface information.

A degree of execution concurrency is present when the system is programmed to halt the CPU core while the other parts of the system, Peripherals, Memory and Interfaces continue to operate.

In FIG. **57**, the stop modes are specified at the beginning of an emulation session. They address the operational mode of the device for the entire session. The operating mode is specified in a separate emulation mode register or module **1311** which is scan-loaded while the CPU core is halted. This advantageous mode feature involves mode driven stops.

In other words, operating modes are defined primarily by the characteristics the chip exhibits when the device halts execution. These operational modes are herein called Pipe Flush, Pipe Step, and Clock Step. Trap enable maps analysis stops to traps and enables emulation trap opcodes.

In pipe flush, the CPU block halts when a stop condition is detected. The CPU ceases instruction fetches, executes all instructions currently in the pipeline, and completes all memory activity in progress prior to signaling DONE to code state machine **1381** of FIG. **57** and signaling the stop condition for scan purposes on line SCOUT. All other blocks continue to run, and all control lines from the CPU core associated with communications with the peripherals or memory are maintained in an inactive state once the pipeline is flushed

when the chip is in emulation mode. In simulation mode peripherals stop with CPU.

In pipe step, the CPU block is the only block that halts when a stop condition is detected. The CPU freezes on a pipeline clock boundary, the pipeline freezes, and all external memory accesses appear to complete successfully whence DONE is supplied. All external memroy control lines go to an inactive state. The address lines are still valid when write enable WE- and Read/Write R/W- go high. If the CPU is started in this mode and an external memory access is determined to have been in progress when the device was stopped, the access is restatted to the external world when execution begins. All control lines associated with communications with the peripherals or memory are reset inactive.

Clock Step steps the device one clock cycle; memory signals stay if multicycle. The CPU traps to a predetermined location as an NMI (nonmaskable interrupt) when an analysis stop condition is detected. Code execution continues and all subsequent program generated stop conditions are disabled until a return from emulation trap instruction is executed. All other domains continue to execute. The word trap denotes a hard call of a subroutine that is handled by the processor. The jump address is not set up by the software being executed, but instead is set within the processor itself. An alternate mechanism is an emulation trap instruction. In a development-type operation, a specific instruction can be replaced with the trap instruction, which is called a software breakpoint for this purpose. If the software is in RAM, a software breakpoint can be used. When the software is in ROM, a hardware breakpoint is provided herein to overcome the problem that an instruction in ROM code would not be replaced dynamically.

In simulation pipe flush, the entire chip halts when a stop condition is detected. The CPU ceases instruction fetches, executes all instructions currently in the pipeline, and completes all memory activity tied to instructions currently in the pipeline, prior to signaling DONE to code state machine **1381** of FIG. **57** and signaling the stop condition for scan purposes on SCOUT. All other blocks stop when the CPU stops.

In simulation freeze, the entire chip stops immediately and signals DONE when a stop condition is detected. All external control lines go to an inactive state.

In simulation trap, the CPU traps to a predetermined location as an NMI when an analysis stop condition is detected; otherwise a stop occurs. Code execution continues and all subsequent program generated stop conditions are disabled until a return from emulation trap instruction is executed. Peripherals are suspended during a time period from the time the trap is taken to the return from emulation trap instruction.

The particular stop mode of the processor is determined by mode register **1311** of FIG. **57**, as already discussed. The location, placement in any particular register, and scanability of each of the bits of the stop mode is quite flexible. In another example, the particular stop mode can also be implemented using five control bits TEST, SIM, EMU, TRAPEN and PFLUSH. The just-mentioned bits resides in the analysis domain in this embodiment. If the stop mode is generally not changed frequently in practice, then it can be put in a separate mode register **1311** to avoid unnecessary scan bits in the various domains. Otherwise, the various stop mode bits can be allocated to domains as described in the second example.

	TEST	SIM	PFLSH
EMU Freeze	0	0	0
EMU Pipe Flush	0	0	1
EMU Trap	0	0	0
SIM Freeze	0	1	0
SIM Pipe Flush	0	1	1
SIM Trap	0	1	0
TEST Freeze	1	0	X

There are subtle differences between the emulation and simulation functions and they are outlined below. Generally speaking, a key difference between Emulation and Simulation stopping modes is that with emulation, the peripheral logic remains active to the extent possible with the CPU core stopped. In the simulation modes, the peripheral logic is also stopped.

Pipe Step stops the CPU on pipe-stage boundaries. Emulation and Simulation differences are determined by what domains are directed to respond to the MPSD port. In Emulation mode, only the CPU domain is connected while in Simulation mode all domains are connected. Since the peripherals and interface domains continue to run in the emulation mode, memory cycles complete and peripherals continue to run. Simulation mode has all domains operating together, resulting in the CPU, peripherals, and interfaces freezing simultaneously.

Pipe Flush—Pipe flush stops the CPU on instruction boundaries. Both simulation and emulation modes require the CPU core to complete all instructions fetched, and clear the pipeline of activity prior to executing the required freeze sequence as described above. In emulation mode peripherals continue to run if directed locally to do so. In simulation mode, peripherals stop.

Trap—Trap does not stop the CPU but instead takes a trap. Emulation and simulation trap differ in that simulation trap causes the peripheral domain to stop when the trap is taken until the emulation trap return is executed.

The CPU core generates a signal called SUSPEND which indicates to the remainder of the device that the CPU has halted the execution of the user program. The behavior of the SUSPEND signal in each of the operating modes is shown below:

EMU Freeze – Asserted immediately when CPU stops.
EMU Pipe Flush – Asserted immediately when CPU stops.
EMU Trap – Asserted when CPU traps for emulation or CPU steps per mode specification or opcode execution.
SIM Freeze – Asserted Immediately when CPU stops.
SIM Pipe Flush – Asserted Immediately when CPU halts.
SIM Trap – Asserted immediately when CPU taks trap. Deasserted with execution of Emtrap return.

In a Suspend Interlock function, the CPU has a scanable bit which causes the SUSPEND signal to be asserted to the remainder of the chip. When the CPU stops, SUSPEND is asserted by core hardware until the CPU is restarted. Since the CPU is asked to run during memory operations, a scanable interlock bit (SUSILOCK) is in the CPU so that SUSPEND can be made to remain active upon software command. In this way SUSILOCK allows the CPU to execute scan initiated memory operations. This bit is initialized to the not suspend (inactive condition) by the JTAG strap state of FIG. **50A**. The SUSPEND signal broadcast to the chip

is the logical OR of the CPU stopped signal and the SUSILOCK bit. The presence of this bit facilitates the use of macros and other program sequences such as fills, finds, or download assists.

The core is used to gain access to memory and peripheral resources. Memory operations are suitably generated using CPU resources. Memory accesses are generated by scanning in a CPU state including appropriate CPU memory access instructions, which causes memory accesses to be generated to the appropriate memory or I/O space. This is accomplished by loading a machine state with the pipe flush bit set, and appropriate instructions in the pipeline to cause the desired memory operation.

Before any memory activity is initiated, the SUSPEND bit is set in the CPU image to prevent the remainder of the system from detecting that the CPU enters the execution mode for a short period.

When the state has been loaded, the CPU is taken from a MPSD pause state to a halt state. The CPU then executes the loaded instruction as though it is finishing a normal halt sequence, setting SCOUT to indicate that an execute is in progress and then signaling DONE on SCOUT when the pipe is empty and all memory operations generated by the instruction are completed. Memory activity beyond that initiated by the instruction scanned into the pipeline does not occur. When the operation is over, the machine appears as if it has completed a normal halt sequence when the halt code is applied from the MPSD port, with the pipe flush bit on.

Multiple memory operations, such as memory dump or file, utilize macro operations. The repeat operations are scanable to set up a single instruction scan load with the repeat operation already established. It is therefore not necessary to load both the repeat and executable instruction via the scan.

In any stop mode the analysis domain continues to function.

Instruction sequences and resource to be used to be for memory and I/O operations are: Program Memory Read, Program Memory Write, Data Memory Read, Data Memory Write, I/O Read, I/O Write.

In order to facilitate fast memory downloads, the preferred embodiment advantageously includes a CPU scan path which minimizes the number of bits transferred to initiate a memory or I/O transaction, especially when the memory transfer is a single word. When the entire register file may be used to create a block transfer, multiple scan modules may be used.

The short scanpath includes all the CPU resources necessary to implement the above mentioned type of memory operations. The repeat instruction and the autoincrementing characteristics of the auxiliary registers are suitably used to create efficient load sequences.

When cache is part of the architecture, it is easily loadable and unloadable via scan operations. This allows the cache to be set up with macros which terminate with software breakpoint istructions. The program counter and cache management hardware are set up to assure that program execution will take place out of cache, the SUSPEND bit is set, and execution is initiated. This allows fast memory transfers, fills, finds and other Macros to be implemented.

The advantage of the cache macro method is the ability to effectively create a hidden program memory which cannot be accessed as the result of instructions.

Advantageously, the preferred embodiment confers a level of concurrency beyond mere JTAG boundary scan. Microprocessors, for one example, are a very valuable and complex application with access to the internal information very important. Since JTAG boundary scan involves a test port, this test port is even more effectively utilized for communication to specific chips one at a time or 211 together, concurrently. The preferred embodiment puts commands in emulation control register 1251 and loosely couples the communication so that the device 11 can run in real time when desired. In this way a merely static test environment is improved to provide dynamic operation of device 11 in response to the commands such as CODA and CODB in FIG. 59, in contrast to loading the JTAG IR 1153 and executing an operation by decode.

Instead of directing the chip clock cycle by clock cycle, the preferred embodiment sets up a condition in the analysis domain, and then the analysis domain effectively monitors the chip as it runs in real time, then detects when the condition occurs, stops the chip and notifies the emulation host computer 1101 that the chip is stopped. In effect, the preferred embodiment of FIG. 59 acts as an emulation speed step-up transformer by reducing the number of commands required of the emulator host by freeing the emulator host from clock cycle-by-clock-cycle supervision and accommodates modern chips that run at clock rates that far outstrip the speed of the emulation host computer. Moreover, the preferred embodiment is upwardly compatible with cycle-by-cycle control, since the wires of the scan interface can be used to generate MPSD command codes by the conversion block or translator 1205 in the STRAP state of FIG. 50A, for instance.

The preferred embodiment has further uses in simulation acceleration and other device debug operations. The device is run on functional clock FCLK and then stopped and a device state is recovered, observed and studied. The JTAG testability interface is thus used to input stimuli via scan and achieve simulation rates on the order of even 10,000 instructions per second which are state-wise accurate. Prototype silicon patterns are readily debugged off line without investment in tester apparatus that may exceed six figures in magnitude of cost. Whereas JTAG boundary scan suggests to the art to reach out to the board, the present embodiment reaches into the bowels of each device on the board as well. Device debug is accelerated because every internal scan state is rapidly produced and recovered.

Accordingly, the capability of now downloading functional code through an emulation system has important implications for the field of test as well. Now the user can do self test. In self test according a preferred method, the user has a random access memory (RAM) in the device 11 or accessible to the device 11. The user downloads an extensive test program through the emulation port that is the equivalent of BIST (Built In Self Test). No dedicated device hardware is thus required for BIST, since the emulation hardware on chip advantageously accommodates this additional use without further investment. The user can download tests using emulation functions, and can run the same test patterns in serial sequence as the user would when doing a go/no-go device test on a manufacturing production line.

Moreover, as illustrated in FIG. 66, tests are downloaded in the chip manufacturing process itself. The host computer 1101 is advantageously coupled to a test head 1651 of a wafer fabrication line 1653 to detect device defects on each wafer 1655 before the wafer is

divided into chips or at any point in manufacturing. A scan interface as described in connection with FIGS. 49, 54, 55, 56, 57 and 59-65 is microscopically provided in each of numerous locations 1657 on the wafer from which many chips are derived respectively.

The host computer 1101 in FIG. 66 is loaded with a testing program and communicates via controller card 1141 and serial line 1103 to a wafer test head 1651. Test head 1661 is precisely positioned in X,Y,Z coordinates 1663 to reliably press contact wires 1665 against microscopic contact pads for each die location 1657 in the wafer 1655. The circuit at location 1657 is the circuit of device 11 for example. A full complement of peripheral resources is available to computer 1101 and device on the wafer, in the form of printer 1143, hard disk 1145, and modem 1147 on bus 1148.

According to a process illustrated in FIG. 67 operations start with wafer fabrication 1671 and then in a step 1673 the wafer 1655 is conveyed to the test position shown in FIG. 66. Then a step 1675 positions test head 1651 in XYZ coordinates 1663 to contact the next chip on the wafer 1655 with test head 1651. A succeeding step 1677 downloads scan-self-test patterns to RAM in the chip via test head 1651. The chip is switched to functional clock to execute the test patterns on-chip in step 1679. Then the device state of the chip is scanned out through the microscopic interface having circuits 1150 and 1203 of FIG. 49 at location 1657 in step 1681. The signals pass through test head 1651 to host computer 1101 for processing and data storage and display in the peripheral resources. In step 1683, host computer 1101 determines whether the chip at location 1657 is defective. If defective, a step 1685 branches to an action step 1687 to optionally dot-mark the wafer location and/or to store data on it to keep a record of the defect which may also be useful for microscopic repairs of the circuit. Then in a step 1689, if all chip locations are not yet tested, operations loop back to step 1675 to precisely position the contacts 1665 of test head 1651 against the next chip location to be tested on wafer 1655. If all locations are tested, operations branch from step 1689 to a decision step 1691. If a next wafer is to be tested, operations loop back to step 1673 to convey another wafer to test position, otherwise the process comes to END 1693.

In device 11, the core CPU implements the following capabilities:

Strap functional when directed from the scan interface.

Run and halt when directed from the scan interface.

Halt when either a software breakpoint or a hardware breakpoint is encountered.

Select the core clock to be a scan clock and prepare to scan.

Assert a suspend signal to the remainder of the chip in order to direct peripheral start/stop features.

Keep a clear record to indicate the reason that the device halted.

Manage interrupt occurrences.

Provide pipeline management for breakpoint and software interrupt occurrences when they interact with delayed branches and other pipeline-relevant occurrences.

Instruction step such as executing the interrupt service routine one instruction at a time if an interrupt occurs.

Generate memory accesses from CPU core while the core is halted.

Program counter stack traces a number of preceding program discontinuities.

Read and write to memory while the CPU core is executing code; communicating with a debug monitor or SPOX debugger resident on the chip or in memory.

FIGS. 68A and 68B show a block diagram of functional circuitry of CPU core domain 1213 which is improved with a series of scan registers indicated as small squares fed from input CSCIN. The CPU core was mostly described in connection with FIGS. 1A and 1B of incorporated by reference U.S. Pat. No. 5,072,418. The detail of the organization of the scan registers and the associated MPSD module circuits for several modules is suppressed for clarity in FIGS. 68A and 68B.

CPU core is further improved by providing a trace stack circuit 1695 distinct from program counter stack 91. Unlike stack 91, trace stack circuit 1695 develops a history of program counter discontinuities and produces a TRFUL trace stack full signal when it is filled to capacity.

A hardware breakpoint circuit 1697 is connected to program address bus 101A and produces a program address break point signal BPPA when a particular program address or address in a predetermined range of program addresses is encountered.

In FIG. 68B, a hardware breakpoint circuit 1699 is connected to data address bus 111A and produces a data address break point signal BPDA when a particular data address or address in in a predetermined range of data addresses is encountered.

For scan purposes, trace stack 1695, and breakpoint circuits 1697 and 1699 are on the separate scan path for the analysis domain 1217. The core however is on the scan path for core domain 1213.

FIG. 69 shows circuitry in the analysis domain 1217 of FIGS. 51-53 and 55-57 which produces a breakpoint signal ANASTP for analysis stop of the core according to the appropriate mode selected. Circuit 1217 includes on-chip circuits for providing signals representative of particular processor conditions. These signals are designated IAQ, CALL, RET, INT, BPPA, BPPDA, and TRFUL. Each of these sensed signals is provided to respective selection circuits 1703 shown as AND gates. The outputs of circuits 1703 are delivered to a combining circuit 1705 shown as an OR- gate 1705, the output of which is ANASTP, the breakpoint signal. Selection circuits 1703 act under the control of stored bits that are loaded via the analysis domain 1217 part of scan path 1252 as shown in FIGS. 54 and 55.

Thus, the bits which are scanned in are loaded into a 12 bit register 1707, 3 bit register 1709 and single bit registers 1711.1 through 1711.8 in this section of the analysis domain 1217.

The contents of each of the registers 1711.1 through 1711.8 qualify or disable a corresponding one of the selection circuits 1703 so that the overall device 11 condition which can trigger a ANASTP breakpoint signal is completely defined.

Line IAQ is activated when there is instruction acquisition by pipeline controller 225 of FIG. 68A. This way of initiating a stop facilitates single step operations, even in ROM resident code. Any cycle in which an instruction is read from program memory is an instruction acquisition. In the pipeline of this embodiment, instruction fetch is the first of four pipeline steps FIG. 29 of incorporated by reference U.S. Pat. No. 5,072,618, and when fetch occurs, line IAQ is activated.

CALL is activated upon a subroutine call. RET becomes active upon a return from subroutine. If desired, therefore, the analysis domain can be scanably programmed to start the device at the beginning of a subroutine and then automatically stop when the subroutine return is reached. Conversely, the device can be scan loaded to start to begin somewhere in a main routine, run at full speed and then automatically stop when a subroutine call is encountered.

INT goes active in response to an interrupt occurrence. When a stop is programmed to occur upon interrupt occurrence, the stop suitably executes on any machine vectored program counter load except the emulator trap. Interrupts that occur while the CPU is stopped are latched but are not executed until the CPU is restarted. When the CPU is restatted it executes the currently addressed instruction before allowing a trap to the interrupt vector. When code is being single-stepped, the code takes the interrupt trap as soon as the pipeline is flushed, as it would in real time.

BPPA is a line responsive to a program address breakpoint circuit. BPDA line is responsive to a data address breakpoint circuit. An example of use of the BPDA data address breakpoint involves a debug problem in which the processor is running correctly most of the time, but a particular address occasionally gets garbage values. By inserting a breakpoint at the affected address, the user stops the processor at the instruction that wrote to that address, scans the instruction out of the stopped processor to host computer **1101** and inspects the state of the processor to determine how to fix the bug. In this manner, system debugging is much more efficient of time and system resources.

Another way of determining system state at the instant a particular address is accessed might be to replace the contents of that address with a trap instruction. This instruction is called a software breakpoint, and the insertion is a development-type of operation. A trap is a hard call of a subroutine that is handled by the processor. The subroutine is programmed to dump the device status so that the user can debug it. However, when the software to be debugged is in ROM, it is impossible to enter the trap instruction in the ROM, since the ROM is read-only by definition. Advantageously, the hardware breakpoint approach is not only applicable to debug in RAM as is software breakpoint, but also in ROM.

A MUX **1713** has eight inputs, and seven of the inputs are respectively connected to lines IAQ, CALL, RET, INT, BPPA, BPDA and TRFUL. The eighth line is connected to a clock line FCLK for functional clock. The clock line can be selected for single-stepping or execution for any selected number of clock cycles whereupon the processor stops.

The three bits in shift register **1709** make a one of eight selection by MUX **1713** and supply the selected line to a 12 bit down counter **1715**. A predetermined count is loaded into the J (jam) parallel input of counter **1715** from 12 shift register **1707**. As signals on the line selected by MUX **1713** occur, the 12 bit down counter counts down until the number represented by the contents of shift register **1707** is exhausted, whereupon a borrow line **1717** goes active and is fed to selector **1703.1**. The borrow line signal is called CNTBRW for purposes of event manager **1365** of FIG. **60**.

The output of selector **1703.1** is not only connected to an input of the combining circuit **1705** for supplying ANASTP, but also is connected directly to an output pin EMU0. EMU0 is advantageously connected to external counter **1719** which communicates with host **1101**.

In this way, the on-chip condition sensor includes a counter selectively connectable to sensor circuits. A logic network is connected to the sensor circuits and a serial scan circuit with SRLs is interconnected with the logic network for determining selections of sensor circuits by the logic network. The serial scan circuit is further interconnected with the counter for loading the counter with the value indicative of a predetermined count to which the condition sensor is thereby made sensitive. The condition sensor further includes a plurality of sensor circuits responsive to particular internal conditions of the electronic processor and a multiplexer **1713** having inputs connected to the sensor circuits and an output connected to the counter.

An example of the utility of the down counter **1715** (besides single-stepping) is as follows. Assume that the counter **1715** is set by scan register **1707** to **200** and the MUX **1713** is set by register **1709** to select BPDA data address breakpoint. The particular data address is scan-entered in a register **1813** in FIG. **71**. This configuration stops the processor after the specific scan-identified data address has been addressed 200 times.

This exemplary use of address breakpoint counting is valuable in designing a digital filter that does not stabilize until it has processed a certain number of signal samples so that its taps are filled. For instance the response of an FIR filter may not be measurable until the number of samples required to fill all the multiply-accumulate filter taps are present. The output of the filter which is of interest in evaluating whether the filter operation is correct thus begins when the signal has traversed the filter. In one type of **16** tap FIR filter it is desirable to stop only after sixteen events and every other time after that to examine the output of the filter.

Advantageously, the analysis circuitry cooperates with the host computer **1101** for emulation, simulation and test of digital filters as Just described.

In another filter example, the counter borrow line is selected by scan register **1711.1** and fed out of pin EMU0 to permit external logic to count events at a rate stepped down by frequency division by the value in the downcounter **1715**. Timing analysis of an algorithm may consume on the order of five billion cycles, wherein timing analysis determines the number of cycles needed to execute the algorithm. (If the counter is made scanable and extended to the appropriate number of bits, this function is advantageously executed entirely on-chip.) A particular filter design may require at least a certain sample rate to meet the performance criteria specified for the filter. The maximum number of instructions available to achieve that sample rate is related to the sample rate and the computer clock rate. The present circuit permits accurate counting of the number of clock cycles consumed by the filter to perform the algorithm, so that the algorithm can be developed to meet the specifications of the filter.

When external logic is used, its resolution is equal to the number set for the counter **1715** by register **1707**. Full resolution is obtained by reading out the value in the 12 bit down counter, allowing determination of the exact cycle count of an algorithm between two break events, further illustrating the advantages of this preferred embodiment. Direct counting of break events and/or clock cycles by counter **1715** on chip is further advantageous because available external counting logic

may be too slow to keep up with the new processors to monitor them.

A break event herein is a condition that causes the processor to stop or to affect counter **1715**. The event detection in one exemplary processor is split between the CPU and Analysis domains. Together, the two domains provide nine different events, all of which can be programmed to cause the processor to stop. The events and their respective domains are shown below:

		Comes From:
1.	Software Interrupt (SWI)	CPU
2.	Instruction Acquisition (IAQ)	CPU
3.	Subroutine Call (CALL)	CPU
4.	Subroutine Return (RET)	CPU
5.	Interrupt/Trap (INT)	CPU
6.	Clocks (CLK)	CPU
7.	Breakpoint Program Memory Address (BPPMA)	ANA
8.	Breakpoint Data Address (BPDMA)	ANA
9.	Trace Buffer Full (TBF)	ANA
10.	Item Counter Borrow (ICB)	ANA

All events are enabled, detected and latched in the analysis domain as discussed in connection with FIG. **69**.

The CPU core should not process local or analysis inputs once a stop condition has been processed. This includes the time from when an emulation trap occurs to when the emulation trap return is executed.

The core responds to the consolidated stop signals of:
CPULSTP—Core local stop stimulus
ANASTP—Analysis stop
SWBP—Software breakpoint detect
HALT—MPBD halt code

When the halt is detected, LSTPCND (Latch stop condition) is asserted to the CORE and Analysis blocks. It is desirable that the CPU stop indications be read from the analysis domain when the analysis domain information is current. This is due to the Emulation Trap mode in which the core domain continues to run while the analysis domain is halted.

Functional reset is gated off when the halts occur due to emulation stop mechanism. It is contemplated that the reset logic (not shown) for device **11** assure that any reset input entering the device past the gating function is stretched to sufficient length so as to correctly complete the reset function. When reset occurs simultaneous to a stop condition the reset is completed and the device stops when the reset is completed and the interrupt trap vector has been fetched.

The processing of interrupts by device **11** should mesh with emulation run/halt operations implemented by CODA and CODB in adapter **1203**. When executing an instruction or clock step, interrupts are serviced when enabled. This assures that single instruction stepping through code will allow the processing of interrupts.

FIGS. **69** and **45** and the description herein thus illustrate an electronic system that has a data processing device including a semiconductor chip and an electronic processor on the chip. Host computer circuitry off-chip is connected to the data processing device. The host computer (e.g. computer **1101**) has a speed of operation which is slower than the electronic processor. The data processing device **11** further has an on-chip hardware breakpoint address circuit, trace stack, pipeline controller condition sensing circuits and other on-chip condition sensors including the counter **1715** for signaling the processor in real time, as well as signaling to the slower host computer. The adapter **1203** acts as a step-up transformer of control speed between slower host **1101** and state-of-the-art speed of device **11**. The analysis circuitry also mediates the real-time control function as well as provides a step-down transformation of data via the counter **1715** to the slower external environment. It is emphasized that the circuitry of FIG. **69** is but one example, and numerous variation can be provided by the skilled worker according to the principles set forth herein to provide sensor logic for any logical combination of conditions so that occurrences of any complex combination of conditions or sequence of conditions can be sensed. The breakpoint signal can be a stop signal ANASTP as illustrated or any other control signal besides a stop signal that should be responsive to sensed target device electrical conditions.

In FIG. **70**, a method of operating the analysis circuitry of FIG. **69** commences with a START **1721** and proceeds to a step **1725** to sense instruction acquisition. Step **1727** senses a subroutine call, and a step **1729** senses a Return. In step **1731**, an interrupt condition is sensed. Step **1733** senses a breakpoint program address, and step **1735** senses a breakpoint data address. In step **1737**, a trace stack full condition is sensed. Then a step **1739** selects which conditions are relevant using shift register **1711** and logic **1703** of FIG. **69** for example. Of the conditions selected, a count is kept in step **1741**. A decision step **1743** determines whether the count exceeds a predetermined count N and if so, a signal of the count N being reached is output in a step **1745**. Operations proceed from either step **1743** or **1745** to a step **1747** which provides an output to the external processing equipment at a slower rate than the rate of operation of the device **11**. In step **1749**, this output is coupled to a host computer whereupon operations return to START **1721** to repeat the steps indefinitely.

In FIG. **71**, a circuit for breakpoint sensor **1697** of FIG. **68A** supplies the signal BPPA for analysis circuitry of FIG. **69**. The circuit is suitably replicated for breakpoint sensor **1699** of FIG. **68B** and connected as shown therein for producing the signal BPDA.

In FIG. **71**, program address bus **101A** of FIG. **68A** is connected to a digital comparator **1811**. A reference value is scan-loaded into a further register **1813** in the analysis domain having most significant bits MSB and least significant bits LSB. When a program address asserted on address bus **101A** is identical to the contents of register **1813**, then comparator **1811** produces an output indicative of a breakpoint address occurrence on line BPPA.

In a further advantageous feature of the breakpoint circuit, a breakpoint may be taken on any address within a selected group of addresses such as the ranp; indicated by the most significant bits MSB of register **1813**. In such case, a scanable mask register LSBEN is scan-loaded to disable the response of comparator **1811** to the LSB bits of register **1813**. Only the most significant bits are compared by comparator **1811** in this mask condition, thereby providing a breakpoint on occurrence of a program address in a particular range of addresses.

Scanable register **1813** for breakpoint purposes requires no connection to data bus **111D**. However, this register **1813** is advantageously reused for message passing access between the emulation/simulation/test host computer **1101** of FIG. **45** and the data bus **111D** of the

target device. The message passing function is used when breakpoint sensing need not occur, and vice-versa, so that register **1813** feasibly performs different functions at different times.

In FIG. **72**, a special program counter trace stack circuit **1821** in analysis domain **1217** holds a predetermined number of addresses defining a history of address discontinuities in operation of program counter **93** of FIG. **68A** in the CPU core domain. A scanable trace stack register section **1823** responds to control circuit **221** to push a program counter **93** address value onto the trace stack when a program counter discontinuity occurs. A leading bit S/E1, . . . S/E9 of each level PC0, PC1, . . . PC9 of stack **1823** stores a state vector representing whether a value PC0, . . . PC9 is a beginning or ending address of a discontinuity.

For example, in FIG. **74**, a program memory space has addresses A1, A2 and A3 indicative of addresses in a main routine from which interrupts are taken. An interrupt routine begins at an address I1 and ends at an address IN. Returning to FIG. **72**, an example of a history of discontinuities is entered as addresses at right on the stack levels PC9 . . . PC0. This history indicates at level PC9 that an interrupt occurred at when a main routine was executing at address A1. Then, just above entry A1 in PC9, there is an entry I1 indicating the beginning address of the interrupt routine of FIG. **74** entered in stack level PC8. Thus, the entry in PC9 is a beginning address of a discontinuity and the state vector bits S/E9 and S/ES have opposite logic levels. For example, a one bit in S/E9 indicates a beginning of a discontinuity. I1 in level PCS is a ending address of a discontinuity.

Then further in this example, level PC7 shows that the interrupt routine ends at address IN and a return occurs to address A1+1 of the main routine. The main routine then executes until illustratively a further address A2 is indicated at level PC5 whereupon the beginning interrupt address I1 is entered in stack level PC4. The interrupt is executed to address IN of level PC3 whereupon operations return to memory address A2+1 of level PC2. The main routine continues to execute and reaches address A3 entered at level PC1 whereupon an interrupt to address I1 occurs and is entered in level PC0.

The state vector leading bits of the program counter trace stack **1821** resemble a shift register for stack purposes and the S/E9 end of the shift register is output to an OR gate **1825**. OR gate **1825** is further connected to a program count shift out register PCSO **1827**. The output of PCSO **1827** is fed back to a second input of OR gate **1825**. When a first logic one is pushed from the bottom of the stack **1821** into OR gate **1825**, register **1827** indicates the output trace stack full TRFUL for analysis circuitry of FIG. **69**. Subsequent push onto stack **1823** pushes out a subsequent zero from the stack into OR gate **1825**. However, register **1827** continues to be loaded by OR-gate **1825** (by virtue of the feedback from PCSO to **1825**) with a one indicative of the trace stack being full.

In this way, the state vector leading bits act as a means for counting the number of discontinuities by a code of alternating ones and zeros. The trace stack **1821** has entries pushable thereon and storage elements for extra bits for the entries. PCSO **1827** acts as an overflow storage element **1827** having an input and an output. OR-gate **1825** acts as a signal combining circuit that has a first input connected to one of the storage elements (e.g. PC9). OR-gate **1825** also has a second input and an output respectively connected to the output and input of the overflow storage element.

Circuit **221** is generally operative when a branch or an interrupt or other discontinuity occurs to enter a new address into the program counter **93** in substitution for an address in what would otherwise be a continuous series of addresses, thereby establishing a discontinuity. Control circuit **221** includes circuitry for pushing the latest address onto the program counter stack and the new address onto the trace stack. Control circuit **221** is also responsive to addresses from memory and is also operative on completion of the interrupt routine for popping program counter stack **91** and pushing the trace stack **1821** once again.

When consecutive instructions or interrupts cause discontinuities, more than five discontinuities can be traced. If the program counter PC is loaded on two consecutive cycles then one less stack level is used, since the current value is the same as the previous new value.

FIG. **73** illustrates a process of operating the circuitry of FIG. **72**. Operations commence at a START **1831** and proceed to a test step **1833** to determine whether a beginning address of a discontinuity is occurring. **1833**. If not, operations proceed to a test step **1835** to determine whether there is a return from a subroutine. If not, operations then loop back to a test step **1837** to determine whether the trace stack is full. If not, operations return to test step **1833**. During the execution of a main routine, for example, the process of FIG. **73** involves a repeated cycle of monitoring steps **1833**, **1835** and **1837**.

On the other hand, when a discontinuity does occur, operations go from step **1833** to a step **1839** to push the latest address (e.g. of the main routine) onto both PC stack **91** and trace stack **1821**. Then in a step **1841**, the new address to which operations have branched or been interrupted is then pushed onto trace stack **1821**. Then in step **1835**, so long as the interrupt routine is executing, operations cycle through steps **1835**, **1837**, **1833**, **1835** and so on indefinitely. When the interrupt routine is completed, step **1835** branches to a step **1842** to push into trace stack **1821** the latest address of the interrupt routine from which operations are returning. Operations then proceed to a step **1843** to pop PC stack **91** to allow the program counter **93** to return to and proceed from the address in main routine from which the interrupt was originally taken.

Next in FIG. **73**, operations proceed from step **1843** to a step **1845** wherein the latest value to which operations have returned is pushed onto trace stack **1823**. Then operations go to step **1837**. When a substantial history of discontinuities has been built up, the trace stack is full at step **1837** and a branch is taken to a step **1847** to output the signal TRFUL.

In FIG. **75**, the operations of the PC stack **91** are perhaps most effectively contrasted with the operations shown in FIG. **72** of program counter trace stack **1821** in the case of a series of discontinuities wherein no subroutine nesting or other nesting is involved. In FIG. **72**, the addresses of the discontinuities are pushed deeper and deeper into the trace stack **1823** even though there is no nesting. However in FIG. **75**, the PC stack **91** either has entered therein Just one address or none, due to push followed by pop, precisely because nesting is absent. The address that is pushed is a main routine address A1, A2 or A3 from which operations have been interrupted. Upon return, the PC stack **91** is popped and

has no entries as indicated by successive hyphens in the PC stack boxes.

FIG. 76 illustrates a simulated peripheral access feature of the preferred embodiment. In FIG. 76, as in FIG. 45, host computer 1101 is connected by a serial line 1103 to apparatus 1043 which includes a circuit board with device 11 thereon and under development. The apparatus 1043 lacks a peripheral 1871 which is to be provided later. Device 11 includes an electronic processor CPU 1873 which is operable to generate a first signal to access the peripheral 1871. Peripheral 1871, if it were present, would reply with a second signal on a line ME 1875 if the access is either a Read or a Write. When the access is a Read, the peripheral also replies with data.

A sensing circuit 1877 is connected to the electronic processor 1873 to temporarily suspend operations of CPU 1873 when the first signal is sent by CPU 1873 in an attempt to access the peripheral 1871. Sensing circuit 1877 is interconnected with analysis and control circuitry 1879. An scanable interface 1881 is connected to CPU 1873 and supplies the signals from CPU 1873 to host computer 1101 via the scan path earlier described. Host computer 1101 simulates the absent peripheral 1871 and determines what second signal the peripheral 1871 would supply. Then in simulation of that peripheral 1871, host computer 1101 down loads a serial bit stream along line 1103 into interface 1881. Thereupon the interface 1881 supplies the second signal which peripheral 1871 would have supplied in response to the CPU 1873. In this way, CPU 1873 receives a signal as if the absent peripheral were present. A clock circuit 1882 for CPU 1873 provides a clock signal to the processor.

The arrangement Just described is implemented in a preferred embodiment by providing the scanable interface as the message passing peripheral 19-16 of FIG. 52. The message passing and analysis 1879 are suitably integrated with the analysis domain 1217 which is interconnected with the CPU core domain 1213.

In this way, the simulation of the absent peripheral 1871 causes the device 11 to be stopped and started in a manner that allows the host computer 1101 to provide signals in substitution for the absent peripheral 1871 and yet to operate the CPU 1873 at full speed when it is running. A visual analogy would be that of a ballerina executing a dance under a strobe light.

As described, host computer 1101 is operative upon occurrence of signal from CPU 1873 of FIG. 76 to simulate the absent peripheral 1871 and load the interface 1881 with a representation of the second signal with which peripheral 1871 would reply if it were present. Then control circuit 1879 resumes operation of CPU 1873 so that it receives the second signal from the interface even though peripheral 1871 is absent. It is to be understood that the arrangement of FIG. 76 is merely illustrative and may be applied to a variety of circuits as well as a digital processor, such as DMA controllers, UARTs, ASICs and any other circuits which need to be developed in the temporary absence of additional circuitry which is able to be "impersonated" by the host computer 1101 operating to simulate the additional circuitry.

FIG. 77 illustrates a method of operating a system such as system 1043 of FIG. 75 that is under development and has a first circuit but lacks a second circuit which is to be provided later. The first circuit sends a first signal to which a second circuit when present would reply with a second signal.

In FIG. 77 the method commences with a START 1901 and proceeds to a step 1903 to sense the first signal sent by the first circuit to access the second circuit. Then in a step 1905, the process temporarily suspends operation by the first circuit when the first signal is sensed. Next, a step 1907 simulates the second circuit to generate a representation of the second signal. A subsequent step 1909 loads the representation of a second signal into an interface to the first circuit. Final step 1911 resumes operation of the first circuit so that the first circuit receives the second signal as a simulated reply from the interface. Upon completion of step 1911, operations return to start 1901 to repeat the process.

FIG. 78 shows a block diagram of message passing circuitry 1216 and located on chip. The message passing circuit 1216 is interconnected with the analysis domain 1217, core domain and communicates with emulation adapter 1203 and scan control 1149. Interrupt generation circuitry 1943 also interfaces the rest of message passing circuitry 1216 to the 16 interrupt lines in the device 11. A serial scan path of FIGS. 54 and 78 has serial data MSIN enter a shift register 1923 CMD/STATUS for entry of commands to operate the message passing circuitry 1216 and for scan out of status information. The serial path continues to a further serial register 1925 designated 16 BIT DATA REGISTER whereupon the scan path exits on a line designated MSOUT. These shift registers correspond to serial/parallel interface 1881 of FIG. 76. The function of shift register 1925 can be implemented by register 1813 in the analysis domain and reuse principles can generally minimize the chip real estate required for message passing.

The shift register 1925 is connected to the output of a MUX 1931 which selects one of three paths to load into the register 1925. Two of these paths are the data and address portions of the TIBUS peripheral bus of FIG. 52 and 58. The data portion is designated 1935 and the address portion is designated 1937 in FIG. 78. The third path called the communication register bus 1939 is connected to a communication register 1941.

The message passing circuitry 1216 is useful for simulated peripheral accesses, for communications I/O (input/output) with host computer 1101 as an attached processor, and for transferring data structures between host computer 1101 and device 11.

The structure and operation of message passing circuitry 1216 is further described in connection with an example of simulated peripheral accesses. The device 11 suitably parallel-loads the register 1925 via MUX 1931 when a peripheral access or other outbound communication is commenced. Host computer 1101 scan uploads the contents of register 1925, and then determines the expected response of the absent peripheral by simulation computations. Host computer 1101 then scan down-loads the simulated response of the absent peripheral into the register 1925. This information in register 1925 includes the data which would be returned from the absent peripheral in response to a Read. To convey the data to the peripheral bus, register 1925 is selected by a MUX 1945 to be loaded into a communication register 1941. Communication register 1941 then supplies the data through a MUX 1955 and then an output buffer 1947 onto the data bus portion 1935 of the TIBUS peripheral bus return to the appropriate part of device 11 under the control circuitry of device 11 as if the peripheral were present.

MUX 1945 can also accomplish reverse data transfers wherein communications outward bound on TI data

bus 1935 reach MUX 1945 at an input 1951 and are communicated via communication register 1941 through communication register bus 1939 and MUX 1931 to the 16-bit data register 1925.

MUX 1955 selects either the communication register bus 1939 or an additional bus 1961 directly connected to data register 1925. In this way, data can be even more directly communicated from register 1925 via path 1961, MUX 1955 and output buffer 1947 to the TI data bus.

Buffer status flags are communicated from hardware 1965 of device 11 along with Read/Write- signal R/W- to CMD/STATUS register 1923 for scan out to host computer 1101. The host computer receives these buffer status flags and returns reply command signals simulating the peripheral, including its "impersonated" reply on line ME.

Some of the command bits from register 1923 are communicated to a command decoder CMD DEC 1971. Decoder 1971 decodes the commands and selectively activates operation output lines OPO . . . OPN to the MUXes and registers of the message passing circuitry 1216 to operate circuitry in accordance with the commands. Thus, processor-level sophistication and flexibility are available in message passing circuitry 1216. In further aspects, MUX 1945 has an input 1951 connected to the data portion of the peripheral bus TIBUS for further flexibility. Register 1925 is connected to interrupt generation block 1943 so that even the interrupt status of device 11 can be scan loaded from host computer 1101.

It is to be emphasized that functional clock FCLK operates when data is loaded into register 1925 from the device 11 peripheral bus and when buffer status flags are loaded into register 1923. Test clock JCLK operates when the data in registers 1923 and 1925 are scan uploaded to host computer 1101, and when data is scan down-loaded to these two registers. Then functional FCLK operates to send data from register 1923 to command decoder 1971 and to send data from register 1925 to the MUXes, registers and buffers and buses of the message passing circuitry and the rest of device 11. These operations and clock switching functions are accomplished by scan control 1149 and adapter 1203 as discussed hereinabove with FIGS. 50 and 59 for instance.

A register UID 1981 is connected to the data bus 1935. A further register JID 1983 is connected to the data register 1925. The outputs of registers UID and JID are supplied to a task identification compare circuit 1985. When the identifications match, an output signal TSKOK is output. Thus, when the message passing circuitry has completed its work it can signal its internal condition to any circuit that can advantageously utilize the information. For example, the task OK signal TSKOK can be used to release a SUSPEND hold on the 12-bit down counter 1715 of FIG. 69.

In some cases of message passing, there is a need to perform emulation and simulation functions without halting a CPU in device 11, thus preserving its ability to service interrupts and perform other functions. This capability permits stop stimuli that normally direct the core to halt to instead invoke a trap to a reserved location. The user then links an emulation monitor program to user software to service the trap. When the trap occurs, the monitor communicates with the emulation host computer 1101 through a TIBUS peripheral such as message passing circuitry 1216 register 1925 having an address that resides in the TIBU8 address space. Once a trap has been taken and until a return from emulation trap has been executed the CPU and analysis stop stimuli are ignored.

FIG. 79 shows a process flow diagram of steps to use host computer 1101 of the development tools of FIG. 44 in expanded ways, when the on-chip interface circuitry 1149, 1203, 1216 is present. A method of operating an electronic system including a host computer serially connected to an application commences with a START 2001 and proceeds to a step 2003 to load host computer 1101 with multipurpose software for scan control including emulation and testability software. Then in a step 2005, functional circuitry such as application system 1043 including device 11 is coupled to the scan line 1103 from host computer 1101.

In the next step 2007 the user or an operating system selects a software program for emulation 2009, simulation 2013, test in wafer fab/manufacturing/field test 2017, and attached processor modes such as temporary coprocessor 2021 and communications I/O 2025. The attached processor modes communicate data related to functional operations of the application system by means of a peripheral such as message passing peripheral circuitry 1216 between the host computer and the application system via the same serial line which also is used for emulation and testability communications.

Emulation operations 2009 include step 2011 scan operations, machine state transfers, run, pause and halt among other operations as described at length herein. In this way signals are produced and inputs are read from the system board 1043 as the chip to be used as device 11 were absent, so that ultimately when the final chip for device 11 put on the board in production, it will have the appropriate ROM code and will be operating in a manner compatible with the board in the application.

Simulation operations 2013 involves executing software in host computer 1101 to simulate the target board so that software development for the device 11 can be performed by one group of engineers while another group of engineers is designing the as-yet-unfinished target system 1043. The device 11 could be software simulated, but if a prototype is available as in FIG. 45, then simulation can be accelerated by executing the device 11 software under development on the device 11 itself and only simulating the rest of the board 1043 on the host computer 1101. This capability of accelerating simulation using device 11 itself is of major importance when the host computer 1101 is of an inexpensive widely available type that is not fast enough to simulate a device running as fast as a DSP, for example.

Instead, the device 11 runs at full speed and then the peripherals (such as off-chip fast and slow memory) are simulated since they are not accessed nearly as frequently as memory and registers inside device 11. Step 2015 is the simulated peripheral access operations as discussed in FIG. 76.

A nonexistent or deleted peripheral 1871 is replaced by a single data port 1216 accessible through the scan test port 1149. An access directed at a non-responding TIBUS address causes the CPU core to halt after the first clock of a TIBUS access. The emulation controller 1101 then extracts the address and the type of access (read or write). The emulation controller 1101 then provides the data through a register 1925 on reads or reads the data directly off the data bus on writes. READY signals for the completion of the cycle are also

provided serially through the scan path to register **1923**. After the appropriate transfers take place, the CPU core of device **11** is restarted.

Test step **2017** involves machine state transfers **2019** as host computer **1101** scan-loads machine states or test patterns into the SRLs of device **11**, which are then processed by the logic of device **11** and scanned out and evaluated.

Use of host computer **1101** as a temporary coprocessor for device **11** in step **2021** involves data structure transfers via message passing circuitry **1216** between host computer **1101** and device **11** as represented by step **2023**. Host computer processes the transferred data structures and then transfers the results back to the device **11** or elsewhere on the target board **1043**.

The data and results are also able to be sent to video terminal, printer, hard disk, telecommunications modem or other peripheral resources of the host computer **1101** which may be unavailable to device **11** otherwise. For this purpose, communication I/O step **2025** performs message passing from the device **11** and target board **1043** generally to the peripheral resources of the host computer **1101** via message passing circuitry **1216** as represented by step **2027**. Transfers and line control are governed by the host computer **1101**.

After any of the above operations are complete, decision step **2029** of FIG. **29** determines whether more operations are required, in which case, the process flow returns to select program step **2007**. If no more operations are required, the process flow ends, as represented by step **2031**.

The ability to qualify all analysis with a user program provided task ID is implemented through the Message Passing Peripheral. The user's program provides a task ID through a TIBUS register value. This value is compared to a value loaded via scan. The comparison is enabled via an extra bit which may be used to force a valid compare to the Analysis section.

The register **1923** in FIG. **78** is implemented as a 3 bit opcode, a four bit status field, and a nine bit TIBUS address/read/write latch for a total of 32 bits.

The delivery of data to algorithms in simulation is readily performed because the message passing circuitry **1216** and other circuitry described herein provide the following features:

1. Peripheral frames (blocks of addresses) may be disabled from decoding addresses, generating interrupts, and driving the TIBUS peripheral bus with any new code (C0, C1, CX) to be applied to domains.

2. Any peripheral interrupt may be generated from the message module via block **1943** of FIG. **78**.

3. The TIBUS peripheral bus can be programmed by scan to Read and Write to register **1925** when no frame recognizes a select.

4. The TIBUS peripheral bus can be programmed via scan to stop the core and device with the second cycle of the bus active, allowing the host **1101** to load or unload register **1925** and obtain address and a read/-write indicator.

5. Restart the device execution from the message passing circuitry **1216** register **1923**.

Each module has a module disable bit which when set through scan, disables the address decode, bus drives, and interrupts. When no module acknowledges the TI bus address and the TI bus block is addressed during simulation pipe freeze with simulated peripheral access in register **1923** enabled, the core stops before the second cycle of the peripheral access is complete. In this mode, reads are targeted at the message peripheral if no other frame is decoded. All writes are directed at the message peripheral. When the simulated peripheral access bit in register **1923** is enabled, four status bits are used to specify the number of wait states associated with the peripheral access.

Turning to the subject of interrupt generation, two methods of inserting interrupts exist. The first supports the simulated peripheral access mode where the interrupts are asserted for one clock when SUSPEND goes inactive after the device starts. The second is the assertion of an interrupt when the device is running in either the simulation or emulation modes.

The JTAG/MPSD interface of the preferred embodiment herein provides enhanced emulation capabilities at low cycle rates of test clock JCLK, and enables simulation of peripheral functions. The interface further provides extensive internal testing for complex devices in low pin-count packages. The flexible circuitry of the interface used with host computer **1101** reduces device prototype to production time, and improves fault testing capabilities in production. The circuitry makes it possible to do boundary scan at printed wire board (system) level. This boundary scan capability is particularly important as board densities increase and the use of surface mount devices with less accessible pins increases.

It is emphasized that while the preferred embodiment is discussed in connection with one processor, an important advantage of its organization is that it is architecture independent. Access and control reach all internal latches. Load/store instruction accesses data RAM. On-chip peripherals are accessed and controlled. The modular approach to each die with which the circuitry is associated allows isolation and test of each module independently, and addition or subtraction of modules in creation of additional chip members of a chip family. Not only standard products and their derivatives, but also semicustom chips and ASIC devices, are supported with a uniform emulation approach and minimum and fully adequate investment. The preferred embodiment makes possible increased fault coverage, and quicker device debug. Emulator support can be made availabile almost coincident with device availability. In this way there is timely availability of emulation and software development tools upon user's receipt of functional chips in silicon, gallium arsenide and other material systems. The system emulation obviates target cables and is nonintrusive and more fully reliable. Full speed emulation is available over the life of a chip family even as functional clock speed is increased, including chips with bus cycle times exceeding 20 megahertz. Emulation circuitry upgrading for new members of a chip family is significantly reduced. The type of emulation circuitry is advantageously independent of the package in which the chip is manufactured, unlike the target cable approach. Less new information needs to be digested by user for development of systems using new chips with software upgrades for emulation. The software can provide built-in documentation.

Significantly, the preferred embodiment merges test and emulation methodology. Both test and emulation have common features involving 1) putting the chip into a known state, 2) start/stop execution and 3) dumping the machine state. Scan paths both dump and restore machine states, and provide a mechanism for invoking memory read/writes. Both real-time and non-real-time applications development operations are addressed with

the same toolset and technology, instead of using separate test equipment and emulation apparatus.

High speed chips are easily accommodated because specialized functionality for emulation is fabricated on-chip. In processor chips, the number of base sets for a given processor core is reduced by eliminating special emulation devices.

Concurrency is herein recognized as a variable defining a spectrum or matrix of emulation technology into which the preferred embodiment and other embodiments of the invention are seen as new advances. The next table categorizes the technology according to the concurrency concept:

TABLE

Level	Acronym	Scan	Execute
1	LSSD	Entire chip	Entire chip
2	MPSD T	Module	Entire chip
3	MPSD E	Module	Module
4	JTAG/MPSD	Module JCLK	Module FCLK
		System Scan Concurrency	
5	JTAG/MPSD	Module JCLK	Module FCLK
		System Execute Concurrency	

In concurrency level 1, the entire chip is scanned and then the entire chip is run in order to execute operations. In concurrency level 2, individual modules in the chip can be selected for scan, and then the entire chip is run. In concurrency level 3, individual modules in the chip can be selected for scan, and then any selected one or more modules can be run. In concurrency level 4, boundary scan is integrated with MPSD modular port scan with system **1043** scan concurrency, so that entire systems can be developed and tested from any level of module through chip through system. In concurrency level 5, system execute concurrency is added to level 4. Each level comprehends the capability of all previous levels.

A message passing aspect of the preferred embodiment involves at least four functions. In a first function, a middle-of-access transfer involves a CPU stop and peripherals stop. Simulated peripheral access is accomplished, for example, by using this first function. A second function uses the message passing circuitry to latch interrupts while message passing occurs. A third function compares task identifications (IDs) and signals that message passing is still in progress or is completed. A fourth function passes messages through the scan serial interface **1149** to a host computer **1101**.

Often a manufactured system board in the context of actual application lacks associated video terminals and printers for testing purposes. Using the fourth function, in an attached processor aspect of the preferred embodiment, the development system acquires control of the application system board in its normal user operational aspects as distinguished from emulation, simulation and debug aspects.

For example if the system board includes an embedded microcontroller, the development system in the preferred embodiment has a mode of operation by which the development system polls status through the scan serial port or receives interrupts from the microcontroller via the EMUO or EMU1 pins of FIG. **53**.

A software interrupt or software trap function in the embedded microcontroller occurs at a predetermined breakpoint therein to signal the host computer **1101** of FIG. **45** for service. In other words, the host computer in the development system is called as a coprocessor to the embedded microcontroller in the application system. For example, the embedded microcontroller can do file transfers to the development system acting as attached processor for display and printer purposes.

Two categories of implementation involve 1) non-concurrent execution and 2) concurrent execution.

In category 1 the embedded microcontroller CPU execution ceases for application purposes and the CPU transfers data, for example, to the host computer **1101**. Host computer **1101** does memory reads and writes using the microcontroller CPU registers and then restores the CPU state when the transfers are completed.

In category 2 (concurrent execution) a message passing peripheral MeGPASS **1216** of FIGS. **52**, **54** and **78** is included in the preferred embodiment combination. Advantageously MSGPASS **1216** allows the microcontroller to execute other tasks after calling the host computer **1101** for service. Then code from the application system is sent via the scan serial line **1103**, and inserts interrupts over EMU0 line to software control to make host computer **1101** perform the attached processor functions.

In this way, a dual function is provided in the six wire SCOPE/MPSD interface of the preferred embodiment. The system board and its microcontroller are used in a more realistic way using all of the serial and parallel ports on the system board for their application purposes without having to temporarily use any of those application ports for testing and normal development system functions. Thus, the SCOPE/MPSD interface is not only useful in prototype and manufacturing test, but also for field test and diagnosis, and retrieving application system operational history and accumulated data for display and printing.

Block transfers are accomplished by use of the message passing peripheral **1216** by loading the register **1925** from communication register **1941**, and with JTAG controller in the IDLE state, do N (e.g 16) bit serial shift, and then cycling back to load register **1925** again. The block transfers can be directed to any other serial interface to which the scan path is connected. With shift rates well in excess of 10 megahertz, substantial communication potential is opened up by dual use as a communications channel of what otherwise might be a mere test port.

In FIG. **78**, a half duplex communications protocol accomplishes alternate download and upload between host computer **1101** of FIG. **45** and message passing peripheral MSGPASS **1216** of FIG. **78**. Host computer **1101** scans bits into registers **1923** for use by the microcontroller. The application microcontroller uses the information and then subsequently loads status and data bits into registers **1923** and **1925**, and then sends a request to the host computer **1101** to upload. Upon a signal back from host computer **1101**, serial transfer from registers **1923** and **1925** to host computer **1101** is performed. (In an alternative embodiment full duplex hardware and communication are provided.)

The message passing peripheral is advantageously further useful for development system purposes. Host computer **1101** operating as a development system downloads a command to register **1923** requesting the machine state of the embedded microcontroller. The microcontroller responds by trapping (analogous to an interrupt) to prestored software code in its memory. The prestored code is executed to cause the contents of the core registers in the microcontroller to be communicated through message passing peripheral **1216** of FIG. **78** back to the computer **1101**. Even as the message

passing peripheral operates, concurrency is maintained and the CPU of the embedded microcontroller is free to accept interrupts while doing debug so that real time control functions are not disturbed.

In this way, message passing peripheral **1216** acts as an electronic system which is programmable by the skilled worker for many uses. For example, the development system can be detached and another non-development-system microprocessor connected in the field for other purposes. In this way the SCOPE/MPSD port comprises a highly flexible communications channel for systems applications. Furthermore, the system **1043** is free to communicate with its application host processor **1044** if one is present.

The exposition of emulation, simulation and test now turns to still further aspects. A coassigned scan test patent U.S. Pat. No. 4,710,933 is hereby incorporated herein by reference.

A preferred embodiment of a graphics system processor, FIG. **80** shows a block diagram of a GSP chip **2120** having a central processing unit **2200** connected by buses **2202**, **2204**, **2206** and **2208** to register files **2220**, instruction cache **2230**, host interface **2240** and graphics hardware **2210** respectively. A further bus **2205** interconnects a host interface **2240**, memory interface **2250**, instruction cache **2230**, and Input/Output registers **2260**. Host interface **2240** and memory interface **2250** are respectively externally accessible via pins and buses **2115** and **2122**. A video display controller **2270** associated with I/O registers **2260** supplies its output on a bus **2124**.

FIG. **81** shows a block diagram of unit **2200** of FIG. **80**.

For the purposes of testability, the GSP **2120** memory elements are split into two types:

(1) Multiple-bit registers such as those in the register file **2220**, the Cache RAM **2230**, a memory address register **2103**, memory data register **2105** and a field size register **2107**. These are all on wide buses, and sufficient logic is included to ensure that there is a route from each of these registers to local address data (LAD) pins of the chip.

(2) Serial latches, such as an emulation control register **2121**, buffer SRLs **2135** of a control ROM (CROM) **2131**, and scanable registers of core processing circuitry **2101** on chip. These are put on two scan paths and are accessible via two bidirectional pins SCIN and SCOUT- in test mode. Extra latches are suitably placed as desired to easily observe key logic elements.

One approach to testability herein is called parallel serial scan design (PSSD). A rule is imposed in which every register bit and serial latch are only loaded by some function ANDED with, or conditional on, a single clock phase (H3T) of FIG. **82**. FIG. **82** also shows clock signals H1T, H2T, T3T, H4T, Q1N, Q2N, Q3N and Q4N. The state of the chip can be "frozen" by keeping H3T at a zero level. All other clocks can occur as normal.

The GSP **2120** incorporates four-phase active-low clocks QIN through Q4N of FIG. **82** generated from the input clock pin. Also present are four active-high half-phase clocks HIT through H4T. As stated previously, all memory elements are loaded only during the H3T phase. During the normal operation of the circuit, the clocks look like the normal cycle shown in FIG. **82** leftmost column. During a special scan test mode cycle (middle column), H3T is held low, freezing the normal load of the memory elements. A special test clock, T3T, is enabled to shift the elements along the scan path. During a hold test mode (rightmost column), both H3T and T3T are held low, thereby freezing the state of the machine.

Each parallel register cell has a circuit shown in FIG. **83**. It is loaded on the (normally conditional) H3T phase and is sampled on a (conditional) HIT phase. Control logic and microcode are included on-chip to enable every parallel register to be loaded and dumped onto the LAD bus. Thus, the machine state can be loaded up, executed, and then the results dumped out.

A circuit for each serial latch is shown in FIG. **84**. It is similar in form to the parallel register circuit of FIG. **83**, but contains an additional serial input called the scan input Scan In. For the purposes of testability, the HIT sample is connected to the T3T input of the next element in the scan chain, so that all the latches are Joined together in long shift registers. The test clock T3T has the same phase as H3T, but is disabled (zero) in normal operation. In special scan-in/scan-out test modes, T3T is enabled and H3T is disabled. Data is shifted along the scan chain. The clocking scheme minimizes the number of extra transistors required to implement the scan path to as little as one transistor plus the routing of the test clock.

In order to utilize the parallel and serial latches, control hardware is included and connected to reset, run-/emu, local interrupt, and hold pins. When both reset and run/emu are pulled low, the values presented on two local interrupt pins and the hold pin provide a 3-bit code which is decoded into one of seven possible test modes.

The test modes will normally be used in the following sequence:

(1) Parallel load—Load all registers on the parallel path via the local memory interface.

(2) Scan-in—Data is shifted into the serial scan paths without corrupting any of the data in the parallel path registers.

(3) Execute—For one or more clock cycles.

(4) Scan-out—Data is shifted serially out of the scan paths without corrupting any of the data in the parallel path registers.

(5) Parallel dump—Dump all registers on the parallel path via the local memory interface.

Another useful test mode is the hold mode. During this mode the machine state is frozen by disabling H3T and T3T.

Video controller **2270** has its own independent two-phase clocking scheme with internal phases V5T and V6T derived from a video input clock pin (VCLK). Registers **2260** used by the video controller are loaded on V6T only. To get around this in all test modes, the video clocks are disconnected from the VCLK pin and the two phases are "joined" to H3T and HIT. Then all the video registers are loaded on V6T, the phase corresponding to H3T.

Parallel Load/Dump Interface—The control of the parallel load and dump is performed by the CPU **2200** but the reads to writes to the LAD **2205** are done by the memory controller **2250**. The CPU is Just one of several sources **2200**, **2210**, **2240**, **2260** which can interact with the memory controller. It is possible for instance for a DRAM refresh controller to request a memory cycle in the middle of the load/dump cycle, but this would upset the sequence and data would be lost by the tester.

These other sources need to be disabled. This is done by first doing a scan out before a parallel load/dump.

This flushes the scan path and clears all requests to the memory controller, ensuring that the CPU is the only active source.

Cache RAM—For performance reasons relating the interface to the LAD bus, the cache is loaded on an H1T. Thus, in order to stop the update of the memory elements in the cache, all the cache registers are loaded based on a signal which is disabled whenever H3T is disabled.

Turning now to FIG. **81**, emulation in a preferred embodiment is implemented with a core **2101** on a GSP chip soldered into a target system **1043** of FIG. **45**. Another discussion of a data processing apparatus with a self-emulation capability is in coassigned U.S. Pat. No. 5,140,687, the disclosure of which was originally filed as application Ser. No. 948,337 filed Dec. 31, 1986 and which is hereby incorporated herein by reference.

A memory address register MA **2103**, a memory data register MD **2105** and a field size register **2107** are associated with main core **2101**. A four wire scan interface or port **2111** is connected to a selecting multiplexer MUX **2112** and connected to serial data in SCIN **2115** and serial data out SCOUT- **2117** pins. Emulation control pins EC0 and EC1 provide further control inputs. All of the four wires of interface **2111** are connected via a selector circuit **2112** to an emulation control register **2121**. Register **2121** is also called a scan control register herein. Special test TST and Compress COM bits 0 and 1 in register **2121** are connected to selector circuit **2112** to route the lines **2115** and **2117** to one of three scan paths. The first path allows scan to register **2121** itself. The second path allows scan of CROM buffers **2135** and core **2101**. The third path connects to a MUX **2113**. The selection is also controllable by emulation control pins EC0 and EC1, which correspond to pins EMU1 and EMU0 earlier described.

A register select code is supplied by emulation control register **2121** bits 2–5 SCAN SEL on a line **2125** to control the MUX **2113**. In this way, register selection of a selected one of registers **2103**, **2105** and **2107** is controlled by emulation control register **2121** in its operation of MUX **2113**. Scan data in and data out on lines **2115** and **2117** are thus selectively routed to registers **2103**, **2105** and **2107**.

Registers **2103**, **2105** and **2107** provide parallel digital communications to and from main CPU **2101**. Advantageously, they are accessible serially via MUX **2113** for scan input and output.

Test modes are controlled via the EC1, EC0 and SCIN pins, and two bits TEST and COMPRESS of the emulation control register **2121**.

The control pins EC1, EC0 and SCIN initially define the state of the emulation control port. Scanning a 1 into the TEST bit zero (0) of the control register **2121** redefines the port as a test control port as long as a code **111** (for EC1, EC0 and SCIN) is not applied to the interface **2111**. The **111** code is a normal user run mode and also clears the entire emulation control register **2121**, including the TEST bit, thus resetting the port. The relationship of the codes to the MPSD codes tabulated earlier hereinabove should be apparent. A scan control circuit **1149** of FIG. **50** is suitably combined with this arrangement according to the configuration of FIG. **49** for even further testability, simulation and emulation and message passing advantages.

In FIG. **81** control ROM (CROM) **2131** is connected to main CPU **2101**. The second bit COMPRESS of the emulation control register **2121** extends the possible number of test states available via the interface and is used for CROM compressions in a type of testing called signature analysis. In such signature analysis, a scanable linear feedback shift register **2141** of FIG. **86** is combined with the CROM buffers **2135** of FIGS. **81** and **85** and is used to self test the CROM **2131**. A compression test clock C3T is also used in this self-test operation. For earlier signature analysis approaches see coassigned Sridhar U.S. Pat. No. 4,601,034 and Thatte U.S. Pat. No. 4,594,711 hereby both incorporated herein by reference. Alternative embodiments can use any BIST (Built In Self Test) configuration and process.

The CROM on the GaP2 has approximately 450,000 transistor sites; there are 256 outputs and 1,568 states. The states are accessed via a novel use of two memory maps. A total of 1,280 states are controlled by a 11-bit micro-Jump (UJ) address code, and the 256 entry-point states are directly controlled by explicit opcode decodes. The selection between entry point or UJ is controlled by a CROM output to a MUX **2137** of FIG. **85**.

In FIG. **86**, the scanable linear feedback shift register **2141** utilizes a data compression method of self-testing the CROM. The method accesses every state. For each access, a word is generated in the CROM output buffer stage **2135.1**, **2135.2**, **2135.i**, **2135.j** etc., from the XOR (exclusive OR) of the new data with the data previously held in the adjacent buffer stage. A further XOR gate **2143** has two inputs connected to a midpoint tap and a right most line. XOR gate **2143** has its output feeding back to the leftmost signature block. The CROM buffers **2135** with signature circuits **2141** thus form a signature analyzer. The basic component circuit of the CROM buffer is shown in FIG. **87**. An XOR circuit **2151** has inputs connected to a CROM output line and a scan line from the adjacent buffer stage. The output of the XOR **2151** is connected to a latch **2153**.

C3T is a special compress clock. It is held low during normal and scan operations, but is enabled during COMPRESS mode (when H3T and T3T are disabled) to generate the signature.

This flow of data causes an incorrect data bit to invert the sense of one bit of data being shifted around the buffers. When the entire CROM has been accessed in this manner, the contents of the CROM buffer signature analyzer are re-examined by scanning them out.

The data (signature) that is scanned out is compared to the expected data so validity of the code in the CROM is determined.

The analysis method herein takes into account the possibility that multiple faults may be undetected. While a single inverted bit, representing a fault, is being shifted around the signature analyzer, it could be "hit" by another fault, and so get get toggled back to the correct value. This happens if the second fault is both N outputs "downstream" and N addresses away. To overcome this problem, the circuitry and method implemented herein preferably presents addresses to the address decoders in two different sequences, the first time counting up and the second time counting down, virtually eliminating the possibility of undetected faults.

The feedback term for the signature analizer is the XOR of the final term and a term near the middle of the CROM. The exact position of this middle term is flexible.

A stack register **2145** in normal operation is used during micro-state pops and pushes in the circuit of FIG. **85**. In CROM test mode this register **2145** is reused as a 13-bit counter. The two most significant bits of

the counter are used to control whether it is counting up or down, and whether the value is to go to an opcode decoder **2147** or a micro-Jump address decoder **2149** associated with a logic matrix **2152** of the CROM. The total number of cycles required to test the CROM is therefore 2^{13} (8K) plus the initial scan-in to initialize the registers and counters, plus the final scan-out to check the resulting signature. Therefore, at a clock frequency of 10 MHz, the complete CROM test requires less than 1 millisecond.

One major advantage of this type of test is that it can be performed with a simple, low-performance, low-pin count tester, but gives excellent fault coverage.

The codes for testability are shown in Table V. Table V has 5 columns vertically labeled TEST, COMP (compress), EC1, EC0, and SCIN in that order from left to right.

TABLE V

TEST	COMP	EC1	EC0	SCIN	REMARKS	
1	X	1	0	0	HOLD	No. H3T phase clocks, i.e. no H3T, V6T, T3T or C3T
1	X	1	0	1	RUN TEST	Clocks not affected, port is testport.
1	X	0	0	D	SCAN	Scan the serial path, T3T active; H3T, C3T, V6T inactive
1	0	1	1	0	EXE-CUTE	Enable H3T, V6T locked to H3T
1	X	0	1	0	PARAL-LEL LOAD	Enable H3T, V6T locked to H3T
1	X	0	1	1	PARAL-LEL DUMP	Enable H3T, V6T locked to H3T
1	1	1	1	0	CROM	Compression. Activates C3T. No H3T or T3T or V6T
X	X	1	1	1	EXIT	From test mode, clears TEST and COMPRESS bits

The scan sequences are as follows wherein hyphens separate the steps in each sequence. A glossary of mnemonics follows the list of sequences.

1. Initialize State—Used to start the machine in a specific known state:

SCANTEST-PLOAD-SCAN(I)-EXIT/RUN

2. Single Stepping—Used to execute for one cycle (e.g. from ATG pattern) and examine the result

SCANTEST-PLOAD-SCAN(I)-EXECUTE-SCAN(O)-PDUMP

3. Test Examine—Examine the state of the machine in the middle of a functional pattern:

SCANTEST-RUNTEST-HOLD-SCAN(O)-PDUMP

4. Compress—Performs signature analysis/data compression on CROM output:

SCANCOMP-PLOAD-SCAN(I)-CROM-SCAN(O)

SCANTEST means "Scan TEST bit into control register".

SCANCOMP means "Scan TEST and COMPRESS bits into control register".

SCAN is a simultaneous scan in and scan out. Scan(I) implies the scan is scan-in for data initialization. SCAN(O) implies the scan is scan-out for data examination.

For parallel load and dump PLOAD and PDUMP, start and execute states are generated internally. Externally, the pins are timed with a set up time of 20 nanoseconds and a zero hold time with respect to rising transition of a clock signal LCLK1.

In order to test the emulator functions, the functional code **111** is applied for two cycles to exit the test mode and clear the entire contents of the scan control register. If the TEST bit is set, the first cycle clears only the test bit. In the second cycle, the code **111** and the cleared TEST bit then clears the rest of the scan control register **2121**. In this way, exit occurs from the test mode into an emulation mode such as an emulation controlled run.

The TEST and COMPRESS bits in this embodiment are only set via the scan control mode and then are only cleared by application of the code **111**. These two bits TEST and COMPRESS are not on the serial test scan path.

The foregoing discussion has generally emphasized test aspects in the GSP **2120**. The following discussion generally emphasizes emulation aspects.

Using the scan sequences, emulation functions are provided in the embodiment of FIG. **81** that include:

1. Simple breakpoints, qualification on IAQ (instruction acquisition) only, and multiple breakpoints set before execution period via memory substitution. A stop point occurs before instruction execution.

2. Memory modification, inspect and/or alter, while the processor is in stop mode without change to user environment. This includes internal (I/O register) and external memory spaces. Inspection and/or modification while in stop mode is also performed. Modification of all internal registers including PC (program counter), ST(status register), and SP (stack pointer) while in stop mode is also available.

3. Single stepping of instructions.

Functions involving the use of a target cable include the following:

1. Hardware breakpoints, qualification on memory write (MW), memory read (MR), instruction acquisition (IAQ), as well as address and data hardware breakpoints. The breakpoints are multiple or sequenced. An event counter **1715** of FIG. **64** provides a breakpoint on occurrence of a repeated occurante of a predetermined condition. A stop point is defined at a predetermined time after a breakpoint event has occurred.

2. A time stamp on trace information is provided as delta time, marked time, or elapsed time.

3. Performance analysis is provided on inner and outer program loops with an overall loop counter.

4. Emulator memory is overlaid for software development. In this way, an external memory can hold external software and be accessed by CPU **2101** using addresses which access on-chip memory after development is completed.

5. Trace of memory operation instruction acquisition IAQ, memory read, and memory write are provided in real time.

SCOUT- is an output pin that is used for scan output of data in the scan mode and for output of a stop acknowledge indication in run modes.

It is emphasized that the use of a target cable can also be eliminated by utilizing the on-chip adapter **1203** discussed at length earlier hereinabove.

An external event such as a hardware breakpoint or user keyboard indicates or signals a stop. The emulation hardware sets a halt code on EC1, EC0, and SCIN and awaits a stop acknowledge STOPACK on the output pin SCOUT-.

Internal state information utilized by the emulator host computer **1101** of FIG. **45** includes the program counter PC, the register file of CPU **9.101** and cache, segment registers and p-flags.

The codes available at the pins EC1, EC0 and SCIN are as follows: Normal functional mode, controlled execution mode, halt, pause, emulation control register scan and scan of data registers **9.103**, **2105** and **2107**. See Table VI. These codes are essentially the MPSD codes discussed earlier hereinabove.

TABLE VI

EC1	EC0	SCIN	
1	1	1	Normal functional mode; Scan control register disabled
1	1	0	Controlled execution mode; CPU active scan control register enabled
1	0	1	Halt; (CPU only)
1	0	0	Pause/Stop scan clocks
0	1	D	Emulation control register scan with (D) from SCIN
0	0	D	Data register scan with (D) from SCIN

The normal functional mode is utilized in design of systems and is obtainable by letting the emulation pins float electrically. The pins have a pull up resistor on chip. The normal functional mode disables the internal control registers **2103**, **2105** and **2107** for emulation and tests, advantageously eliminating preloading of these registers through scan paths.

CROM **2131** is preloaded with microcode which establishes the emulator functions, memory inspect-/modify and internal state load/dump.

All of the registers are scanned in through the LBB (least significant bit) and out through the MSB (most significant bit). Emulator control register **2121** is scannable in response to the Table VI code "emulation control register scan" applied to pins EC1 and EC0.

The normal functional mode is the normal operating mode without an emulator. It is used to initially reset the chip upon power up. Reset is a global signal within the chip. The functional run mode code continuously clears the emulation control register **2121**.

The controlled execution mode is used for running code or for execution special functions such as load and dump. In this mode, reset is gated with a Block Reset bit in the emulation control register. The emulation control register **2121** is not cleared on reset in this mode.

A portion of the microcode responds to the HALT code to halt the CPU for emulation. The halt state is entered in any of the following when any of the following conditions is present:

1. A halt code on the emulation pins;
2. An emulation breakpoint opcode;
3. Completion of a single-step operation; or
4. A memory access completion requested by the emulator.

The emulation modes are recognized on instruction boundaries or normally interruptable points of operation in the CPU **2101**.

In the normal functional mode, no emulator is assumed and a software trap is taken in executing the microcode when a stop condition is required. If the emulation control pins are in any other state, the presence of an emulator or host computer **1101** is assumed and the CPU **2101** waits in a loop awaiting a halt code when a stop condition is requested by the emulator.

The CPU **2101** enters the halt state by performing the following operations when in the normal functional mode. First, the CPU waits for any CPU initiated memory cycles to complete. Second, the CPU stores the contents of the program counter PC in the memory data register **2105**. Third, the CPU signals the stop by generating a STOPACK signal, forcing scan out line SCOUT low. Fourth, the CPU takes a software trap provided for this situation.

If the CPU is in any state other than the normal functional mode, the CPU enters its HALT state by performing the following set of actions. The CPU performs all of the previously stated functions except the software trap. Instead of the software trap, the CPU waits for a HALT code on the emulator pins and then lastly enters halt microcode. Where CPU **2101** is part of a graphics signal processing chip (GSP), the memory controller, host interface and video timing logic continue to operate normally.

A further mode called the pause mode is used to stop serial shift logic in the middle of data being shifted and immediately stop the chip for the tester or emulator. PAUSE is accomplished by inhibiting all clocks on chip.

Scan modes of the system involve scans which occur at the rate of one bit per cycle of LCLK1. Internally, data is latched at the end of H3 which is approximately the rising edge of LCLK1. Set up and hold times are shown in Table VII.

TABLE VII

	Parameter	Min	Max	Unit
Tsu (EC-CLK1H)	Setup time of EC pins valid before CLK1 high	20		ns
Thd (EC-CLK1H)	Hold time of EC pins valid after CLK1 high	0		ns
Td (CLK1H-SCOUTL)	Delay time from CLK1 high to SCOUT low		20	ns
Td (CLK1H-SCOUTH)	Delay time from CLK1 high to SCOUT high		20	ns

Establishing either the control mode or the data scan mode inhibits CPU state machine activity while the mode code is applied. This allows data to be scanned without being acted upon until the data is in its final desired position in the scan chain. SCANIN and SCANOUT are both accomplished in the same scan.

The scannable registers are memory address register **2013**, memory data register **2105** and field size register **2107**. Register **2103** and **2105** each have 32 bits, For example, the field size register **2107** is 6 bits. The register to be scanned is determined by a data scan select field (bits 2–5) of the emulation control register **2121**.

The microcode in the CROM accomplishes four main functions on command. First, it transfers data from a selected device register or from cache or from program counter to the memory data MD register **2105**. Second, it transfers from the HD register **2105** to a selected device register or to cache or to program counter. Third, it executes MPSD code to do step-by-step operations. Fourth, it sends instructions to the memory interface **2250** to transfer data either way between external memory and the register pair MD **2105** and MA **2013**.

Memory address register **2103** holds the address for all CPU initiated memory accesses including those of the emulator. After a memory access is completed, register **2103** is incremented by 32 bits to point to the next word address. The low 5 bits are left unchanged. When the chip is halted, the program counter PC contents are loaded into the upper 28 bits and the halt condition code occupies the low four bits of this register.

Memory data register **2105** passes data between the emulator and memory controller. Register **2105** serves as a data latch for passing data between the emulator and the CPU for loads and dumps. When the CPU is initially halted, this register **2105** contains an image of the CPU program counter and a halt code in the low order four bits. This code indicates the type of cycle that is halted. The halt codes are shown in Table VIII.

TABLE VIII

MD3	MD2	MD1	MD0	
0	0	0	0	Normal halt code
0	0	0	1	Halt during pixel block transfer (pixblt) or IDLE
0	0	1	0	Halt during RESET
0	1	0	0	Halt from EMU opcode
1	0	0	0	Halt during single step
1	1	1	1	Halt due to Bus Fault on EMU memory access

The field size register **2107** makes it possible to do memory accesses to data fields of various sizes specifiable by the field size register.

The value loaded into register **2107** is a 6 bit code that indicates the number of bits to be written. Using register **2107** permits the emulator to write to bits or fields without having to do a read-modify-write operational sequence, which could interfere with host computer access operations occurring between the read and write of the sequence. When the CPU is initially halted, the register **2107** contains an indeterminate value, since it is the actual latch and not a copy. Scan in of a value into register **2107** enters the value in the most significant 6 bits of register **2107**. Upon scanning out the value, the value is in the least significant 6 bits of register **2107**.

The emulation control register has bits as specified in Table IX. The contents of emulation control register **2121** are not executed until control scan mode is changed to another MUX.

TABLE IX

Bit	Function	Description
0	Test mode enable	Puts device in production test mode TEST
1	Signature enable	Puts device in CROM signature mode (COMPRESS)
5-2	Data Scan Select - 4 bits	Selects which register is scanned on a data scan
9-6	EMU Function Code - 4 bits	Read/write/load/dump select
10	EMU busy enable	Connects CPU to emulator busy rather than memory controller
11	Load mapper enable	Conditions control to load overlay mapper
12	Memory cycle abort	Aborts current memory cycle
13	MAP/WP enable	Enables mapping and write protect logic
14	Macro mode	Blocks RESET, NMI and flushing the CACHE
15	Block host port	Blocks host port cycles, causes not ready
16	Single step	Forces CPU to execute one instruction
17	CPU priority	Raises CPU priority above Host

TABLE IX-continued

Bit	Function	Description
18	Device disable	Disables the device and tri-states all pins
19	EMUReset	Emulator generated reset
20	EMUINT	Forces emulator to halt during an idle instruction or pixblt
29	Cache flush flag	Indicates a cache flush condition
29	BUSERR flag	Indicates a bus error condition
30	RETRY flag	Indicates a memory retry condition
31	MEMCTL BUSY flag	Indicates that the memory controller is busy or that the emubusy bit is set

Four bits 2–5 in the emulation control register **2121** select one of the registers **2103**, **2105** and **2107** for serial scan during Data Register Scan mode. During normal functional mode, these four bits are cleared to zero. Table X shows the scannable registers and their scan codes.

TABLE X

SCN3	SCN2	SCN1	SCN0	Select Code
0	0	0	0	Scan MA
0	0	0	1	Scan MD
0	0	1	0	Scan Data Size Latch

The emulator supports a set of functions that are tabulated in Table XI. The proper function code is placed in the emulation control register. The processor CPU **2101** is then placed in the controlled run mode. The CPU **2101** then forces line SCOUT high, indicating that the function is being executed. When the operation is complete, CPU **2101** forces line SCOUT low again. During normal functional mode, these four bits are cleared to zero.

TABLE XI

FCN3	FCN2	FCN1	FCN0	
0	0	0	0	Reserved
0	0	0	1	Run
0	0	1	0	Reserved
0	0	1	1	Return to reset
0	1	0	0	Reserved
0	1	0	1	Resume Interrupted Instruction
0	1	1	0	Reserved
0	1	1	1	Run Macro
1	0	0	0	Dump ST, PC
1	0	0	1	Dump Reg. File
1	0	1	0	Dump Cache
1	0	1	1	Load ST, PC
1	1	0	0	Load Reg
1	1	0	0	Load Reg. File
1	1	0	1	Load Cache
1	1	1	0	Read Memory (inc address)
1	1	1	1	Write Memory (inc address)

The bits FCN3, FCN2, FCN1 and FCN0 occupy bits 9–6 of the emulation control register **2121**.

An emulator run mode of Table XI is used by the emulator to execute user code on CPU **2101** by running or single stepping.

In emulator dump functions, the emulator scans in a request for a dump. The CPU **2101** then fetches the requested parts of the machine state and loads them into the memory data register **2105** one by one. For each group of 32 bits, the emulator host computer **1101** scans out memory data register **2105** serially to obtain the data. More specifically, the operation for emulator

dump is as follows. First, the emulator scans in the four bit function code of Table XI to dump the state using the control scan mode and sets the emulator busy enable bit **10** of register **2121**. Second, the emulator enters the controlled execution mode. Third, the CPU **2101** forces SCOUT pin high. Fourth, the CPU **2101** places a 32 bit word of the machine state in register **2105** and forces SCOUT pin low. Fifth, CPU waits for the cycle to complete. When emulator busy bit is enabled, the CPU signals stop acknowledge STOPACK on the SCOUT pin low. Sixth, the emulator enters a data scan mode and scans the register **2105**. When scanning, the CPU **2101** is inhibited from concurrent activity in this embodiment. Seventh, operations return to step two for the control execution mode. Exiting the scan mode clears the emulator busy flag.

The end of the process is determined by the known number of words to dump. After dumping the cache and the register file, an extra controlled run is executed after the last word has been scanned out so that the CPU can complete its state sequence and return to halt. The CPU **2101** signals the return to halt by asserting STOPACK.

The emulator then clears the emulator busy bit in the emulation control register **2121**.

For example, in the function DUMP ST,PC, the "1000" function code causes the CPU program counter and status register to be dumped. The status register is dumped first, followed by the PC.

In the DUMP REG.FILE function, the "1001" function code causes the A and B register files to be dumped in that order.

In the DUMP CACHE function, the "1010" function code causes the cache to be dumped. (The cache has registers in different segments. A least recently used LRU segment is overwritten from external memory in normal cache operation.) Cache dump occurs in the following sequence. The data registers, each followed by respective Present (P) flags, are dumped first starting with segment A so that 32 segment A registers are followed by segment A Present flags, then the same for the other segments. Next, the A segment start address is dumped with 9 LSBs set to zero. The next three words contain the segment B, C and D addresses. The final word contains the LRU stack that identifies the segments according to the least recently used LRU criterion. The two LSBs contain the number of the least recently used segment. The next recently used segment numbers are packed into adjoining bits up to bits **6** and **7** which contain the most recently used segment number. The total number of words dumped is **137**.

In emulator load functions, designated by codes **1011**, **1100** and **1101**, the emulator scans in a load request and values into the register **2105**. CPU **2105** then builds the machine state from values in register **2105**. First, the emulator scans in the code to load the state using the emulation control register scan mode and then sets the emulator busy bit. Second, the emulator scans in register **2105** using the data register scan mode. Third, the emulator enters the controlled execution mode, and fourth, the CPU **2101** forces line SCOUT high. Fifth, the CPU requests a write of a 32 bit word of the machine state and waits for the cycle to complete. After the data has been loaded, line SCOUT is forced low. Sixth, the emulator scans a succeeding 32 bit word into register **2105**. Exiting the data scan register mode clears the busy flag. Seventh, operations return to the third step of entering the third execution mode. The end of this process is determined by the number of words to load which is a predetermined number. The emulator then clears the emulation busy enable bit **10** in the emulation control register **2121**.

In the LOAD PC,ST function, the **1011** function code causes the status register to be loaded followed by the CPU program counter. In the LOAD REG8 **1100** function code, the A and B register files are loaded in that order. In the LOAD CACHE function **1101** code, the cache is loaded starting with segment A followed by LRU stack then P flags for segment A followed by 32 segment A data registers. Then the same operations are performed for segments B, C and D. After loading segment D. After loading segment D, a dummy load is loaded. The total number of words loaded is 138 in this embodiment.

The emulator can access any part of the chip address space including I/O registers by scanning in address values to register **2103** and data values to register **2105**, together with a memory read or write function code to emulation control register **2121**. When the CPU is in the emulator halt state, these registers are available to the emulator and the controlled execution mode is then entered. The CPU is then controlled in such a way that it requests the memory access and then upon completing the access, the CPU returns to the emulator halt state. In this way on-chip functions are implemented with transitions from state to state in the CROM acting as a state machine in this alternative embodiment to the hardwired adapter **1203** circuitry of FIG. **59**. The normal halt sequence then signals the emulator that the memory access is complete. After downloading code using this mechanism, the emulator flushes the cache by setting the cache flush bit **29** in the emulation control register **2121**.

The functions of the bits of emulation control register **2121** (which is analogous to emulation control register **1251** of FIG. **59**) are now discussed in even further specific detail. If emulation busy enable bit **10** is set when the CPU requests a memory access, then to CPU **2101** the memory interface appears to be busy. This inhibits the CPU from modifying registers **2103** and **2105** and gives the emulator time to scan data out. The busy flag remains set until controlled functional mode is re-entered.

Load Mapper enable bit **11** forces the memory controller to generate a special type of memory write cycle. Bit **11** accomplishes this by forcing a load mapper bus status code and by blocking the RAS and buffer control outputs. This allows the CPU to use memory write microcode to support loading of the Mapper. During normal functional mode, this bit is cleared to zero. The emulator insures that the "data" part of the MA register **2103** contents is not contained in the least significant five bits as these bits of the register **2103** are not output to the LAD bus. The emulator insures that the least significant five bits are loaded with zeros to insure that the memory controller does not perform the cycle twice regarding it as a non-aligned write.

A memory cycle port bit **12** signals that the current memory cycle should be aborted. Before another memory cycle can be started, this bit is cleared by the emulator. During normal functional mode, this bit is cleared to sero.

A MAP/WP enable bit when set, enables Overlay Mapping and Write Protect features. When this bit is set, the time multiplexing on the PAGMD-, BUSER,

and the Size 16- pin is also enabled during normal functional mode this bit is cleared to zero.

A Macro mode bit **14** makes it possible to run programs in cache without being affected by a functional host computer HCF of FIG. **80**. This bit blocks reset, all interrupts, and the cache enable bit. The cache P flags are not checked during Macro mode, and can be cleared by the host HCF. When reloading the cache, the P flags should not be changed. During normal functional mode, this bit is cleared to zero.

A block host port bit **15** prevents the functional host computer HCF from asserting accesses through the host port lines **2115**. If the host HCF makes an access when this bit is set, the host port **2240** is put in the not ready state until the bit is cleared. This feature is used in conjunction with a host port protocol. During normal functional mode, this bit is cleared to zero.

A single step control bit **16** in the emulation control register **2121** causes core **2101** to execute only one instruction before generating a stop acknowledge STOPACK signal on the SCOUT- pin to indicate an emulation stopped condition. This is similar to forcing an emulation instruction into the instruction stream after the current instruction. This bit ORed with the single step bit in the status register before going to the microcontroller. During normal functional mode, this bit is cleared to zero.

A CPU priority bit raises the CPU's priority above host HCF accesses. This allows the emulator to steal cycles to load the Happer and memory without completely blocking the host port **2240**. During normal functional mode, this bit is cleared to zero.

A device disable bit **18** disables all outputs including the clocks. Normal functional mode forces an exit from this mode. During normal functional mode, the bit is cleared to zero.

An EMUReset bit is ORed with the reset input. Writing a "1" to this bit generates a reset condition. This bit is cleared when further operations are to be executed.

An emulation interrupt bit EMUINT when set forces an emulation interrupt. In this way, an IDLE instruction can be interrupted. This bit is effective when both it is set and the halt code is placed on the emulator pins.

A cache flush flag **29** indicates that a cache flush has occurred during a current emulator access. This flag is automatically cleared when scanned out. During normal functional mode, this flag is cleared to zero.

A BUSERR flag indicates that a memory bus error has been detected on the BUSER and LRDY pins during a current emulator memory access. This flag is automatically cleared when scanned out. During normal functional mode, this flag is cleared to zero. If a bus fault occurs on either an emulator **1101** initiated access or during a macro, the CPU asserts a STOPACK and waits for a halt signal from the emulator hardware. When the halt is received, the CPU **2101** inserts a code **1111** (halt due to bus fault) on the least significant four bits of memory address register **2103**, along with the contents of the program counter.

A RETRY flag **30** indicates that the target system has requested a memory retry on the BUSERR and LRDY pins during the current emulator memory access. This flag is automatically cleared when scanned out. During normal functional mode, this flag is cleared to zero.

A memory controller busy flag MEMCTL BUSY is used by scan logic to detect that an emulator requested memory cycle has completed. This bit is not latched and is read by the emulator.

Turning now to the subject of breakpoints, software breakpoints are suitably used for software code development and debug. Multiple breakpoints can be set during the stop mode (control mode). When the user initiates a run, any of the breakpoints insures a processor **2101** stop. Upon the occurrence of a breakpoint event, the breakpoint is cleared from the user's breakpoint stack. This permits continuation of program flow without interruption by the breakpoint just encountered.

The mechanism for software breakpoints utilizes an emulation instruction designated "EMU". This instruction when encountered in the instruction stream by CPU **2101** causes the CPU to send a STOPACK signal. In this way, the program counter PC is left pointing to the EMU instruction which it has encountered. To remove the breakpoint, the emulator reinsefts the original instruction into memory and flushes the cache.

For software debug and emulation purposes, there are two versions of the EMU instruction. Illustratively, the opcodes are **0100h** for a "normal" EMU instruction. This causes the CPU to take a software trap. Another opcode **0110h** for an "EMU present" instruction causes CPU **2101** to generate STOPACK and wait in a loop until a halt code is present. When the emulator computer **1101** establishes the halt signal on pins EC0 and EC1, CPU **2101** jumps to emulation halt microcode.

A single step mode bit **16** in the emulation control register and a similar bit in CPU **2101** status register, control CPU function for single step. When either of the single step bits are set to "1", the CPU halts after instruction execution and executes operand transfer cycles for that particular instruction. That is, the next instruction is executed which is either the instruction presently identified by the program counter PC or the first instruction of an interrupt service routine. It is to be noted that the single step operation is similar to the emulation stop sequence. The cache behaves normally during single step. If the emulator is to disable fetches of other instructions (cache fill) it sets the cache disable or cache flush bits before single step. In this way, fetching of other instructions is preventing.

In order to make a transition into a single step or normal run mode, interrupts are sampled prior to the instruction being allowed to execute. Then if an unmasked interrupt is pending, the core **2101** takes a trap and the first instruction (or the only instruction in single step mode) is that instruction to which the interrupt vector points.

Interrupt logic associated with core **2101** monitors for interrupts regardless of the state of the emulation control register **2121**. Thus, the state of an INTPEND IO register will be the same as if the core **2101** has interrupts masked for any HALT/SCAN periods. On a transition into run or single step with the interrupt enable bit of the status register set, the highest priority pending interrupt is taken. In this way, the interrupt enable bit is cleared of status which inhibits further interrupts without emulator **1101** or software intervention. The interrupt acknowledgment is suitably a status code output during the interrupt vector fetch. Since the CPU initiates the memory cycles, this status code output is completed before the stop acknowledge signal STOPACK- is issued. Both interrupts and emulator stops occur on instruction boundaries, or when interrupts are sampled on interruptable instructions. In the event that both interrupts and emulator stop are requested on a given instruction boundary, emulator stop takes precedence.

In the emulation mode, for example, the emulation hardware uses the multiplexed emulation pin functions to start and stop, single step, execute macro instructions, scan out and scan in internal machine status. A typical emulation sequence is RUN-STOP-RUN as illustrated by Table XII.

TABLE XII

Emulation Pins	Scan Data
HALT	
Wait for SCOUT- low	
Scan EMU Control	Set Data SCAN=MA
Scan DATA	MA=OX801000
Scan EMU Control	SCAN=MD, FCN=WRITE MEM, EMU Busy EN=1
Scan Data	MD=data
Controlled Run	
SCOUT- Goes High	
Wait for SCOUT-Low	
HALT or SCAN	

Hardware reset should reset the chip without destroying the contents of overlay memory. In normal functional mode, reset becomes a global reset that is intended for initial power up. Reset should be blocked when the halt, pause, or scan modes are placed on the emulator pins. Reset should also be blocked when in the MACRO mode. Further, in other than normal functional mode, the memory controller should complete any memory cycles in progress (memory abort) without destroying memory contents upon reset and should perform refreshes while reset is low.

The overlay memory consists of one or two pages of DRAM that can be mapped on programmable boundaries. The mapping is done by high speed static RAMs connected to the latched bus **2122**. The output of one of the RAMs when active low indicates that the memory access should come from the overlay memory and not the chip containing core **2102**. This is done by blocking the normal outputs and providing new signals to the overlay memory. This memory is considered local to the emulator and is not accessed from the target system including the chip.

It should be understood that various embodiments of the invention can employ, hardware, software or microcoded firmware. Process diagrams herein are also representative of flow diagrams for microcoded and software based embodiments.

While this invention has been described with reference to illustrative embodiments, this description is not intended to be construed in a limiting sense. Various modifications and combinations of the illustrative embodiments, as well as other embodiments of the invention, will be apparent to persons skilled in the art upon reference to this description. It is therefore contemplated that the appended claims cover any such modifications or embodiments as fall within the true scope of the invention.

APPENDIX

Command Syntax	Operation Description
	EMULATOR COMMANDS
	BREAKPOINT COMMANDS
BA address	Breakpoint on Acquisition - breakpoint occurs when instruction is loaded into the instruction register.
BD id number	Breakpoint Delete - delete breakpoint number id number.
BL filename	Breakpoint Load - load breakpoints stored in filename.
BR	Breakpoint(s) Remove - remove all breakpoints.
BS filename	Breakpoint Save - save all breakpoints in filename.
	CONFIGURATION COMMANDS
CC	Configure Colors - set color, reversing, and blinking for screen.
CM	Configure Memory mode - enter mode; mnemonic's final letter and operand(s) define action.
CMA low address, hi address, type, {R | R/W}	Configure Memory Add - designate a block of memfined by the id number assigned by the CMA command.
CMD id number	Configure Memory Delete - delete the memory defined by the id number assigned by the CMA command.
CME id number, low addr, high addr, type {R | R/W}	Configure Memory Edit - edits the memory block defined by the id number assigned by the CMA command.
CML filename	Configure Memory Load - load the memory configuration stored in filename.
CMR	Configure Memory Reset - reset configured memory to default initialization.
CMS filename	Configure Memory Save - save memory configuration in filename.
	DISPLAY COMMANDS
DB [start position]	Display Breakpoints - display breakpoint configuration as set by the breakpoint commands.
DC [start position]	Display memory Configuration - display memory configuration beginning with start position.
DE	Display Expression - display expressions in display window.
DF filename, [line number]	Display File - display text file starting at line number.
DM address	Display Memory - display memory starting at address.
DS [start position]	Display Symbols - display all symbols or start at start position line number.
DT [start position]	Display Trace - display trace configuration as set by trace commands.
DV	Display Version - clear display area and print banner.

APPENDIX-continued

Command Syntax	Operation Description
	EMULATOR COMMANDS
	EXPRESSION COMMANDS
EA expr, [name], [radix], {B | W | D}	Expression Add - add expression with optional descriptors.
ED id number	Expression Delete - delete expression identified by id number.
EF id number, {A | B | D | F | H | O}	Expression Format - format expression as set by numerical parameters.
EL filename	Expression Load - load expression from filename.
EN id number, name	Expression reName - change expression name to name.
ES filename	Expression Save - save expression in filename.
	JOURNAL COMMANDS
JC filename	Journal Capture - in Command mode, save in filename line commands for batch execution.
JE filename	Journal Execute - execute commands store in filename by JC command.
JS	Journal capture Stop - halt JC command execution.
	LOAD COMMANDS
LB filename	Load Breakpoint configuration - load from filename the breakpoint configuration.
LC filename	Load memory Configuration - load from filename the memory configuration.
LE filename	Load Expressions - load expression from filename.
LM filename	Load Memory dump - load filename into memory at locations set by the corresponding SM command.
LO filename	Load COFF file - load the COFF file named filename.
LR filename	Load Register configuration - load the registers with the configuration stored in filename.
LT filename	Load Trace Configuration - load in trace configuration stored in filename.
	MEMORY COMMANDS
MA address, statement	Memory Assemble - show source statement in address. Also, input/assemble new statement.
MF state address, end address, value	Memory Fill - fill memory from start address to end address with value.
MM address, value	Memory Modify - change location address to value.
	OP SYSTEM, REGISTER/EXPRESSION COMMANDS
O	Escape to Operating System. (Emulator retained in memory.)
Q	Quit emulation session. (Emulation session not retained in memory.)
R register, value	Fill register with value (could be expression results).
	SAVE COMMANDS
SB filename	Save Breakpoint configuration - save the breakpoint configuration in filename.
SC filename	Save memory Configuration - save the memory configuration in filename.
SE filename	Save Expressions - save the expression in filename.
SM filename, start address, end address	Save Memory dump - save the memory dump from start address to end address - save in filename.
SR filename	Save Registers - save registers in filename.
ST filename	Save Trace configuration - save trace configuration in filename.
	TRACE COMMANDS
T	Trace mode - enter mode; mnemonic's final letter and operand(s) defined action.
TB	Trace Bottom - display bottom of trace file.
TD	Trace Disable - halts collecting trace samples in a trace and closes file opened by TE command.
TE filename, expression	Trace Enable - enables collecting of trace samples in a trace as long as expression is nonzero, and opens filename as new file to collect samples.
TF column number, expression	Trace Format - trace report to show results of expression in format column specified by column number (expression not needed for column 0).
TL filename	Trace Load - load trace configuration saved in filename.
TP sample number	Trace Position - display trace beginning at sample number.
TS filename	Trace Save configuration - save trace configuration in filename.
TT	Trace Top - display at top of trace file.
TU id number	Trace Unformatted - delete id number expression from trace configuration.
TX	Trace eXecute - execute in format specified by operand (similar to operands for eXecute commands).
	EXECUTE COMMANDS
X	eXecute mode - enter mode; mnemonic's final letter

APPENDIX-continued

EMULATOR COMMANDS	
Command Syntax	Operation Description
	and operand(s) defined action.
XB	eXecute Benchmark - executes beginning at current PC value. Halts at first breakpoint; the number of clock cycles executed from start to a breakpoint displays in the CLK register.
XC [cycle count]	eXecute Cycle - execute for cycle count (clock cycles).
XD	eXecute Disconnect - puts the emulator in user run mode, thus, functionally disconnecting the emulator from the target system.
XG	eXecute Go - execute, beginning at present PC value.
XI [count]	eXecute Loop - Monitors execution, counting the number of times the PC value returns to the same value it was at execution start.
XO	eXecute cOnnect - functionally connects the emulator to the target system.
XR	eXecute Reset - reset TMS320C30 as if the pin RESET was asserted. The value at the reset vector address (0x000000) is placed in the PC.

What is claimed is:

1. An emulation device which enables a functional circuit to support its own emulation, comprising:

a serial scan testability interface having at least first and second scan paths, said first scan path being provided for applying digital information to the functional circuit for use in emulation of the functional circuit; and

state machine circuitry connected to said second scan path, said state machine circuitry having a sequence of states responsive to said second scan path and corresponding to emulation command codes.

2. The emulation device of claim 1 wherein said serial scan testability interface comprises a JTAG interface.

3. The emulation device of claim 1 wherein said second scan path includes scan registers for holding respective emulation command codes.

4. The emulation device of claim 1 wherein said second scan path includes a scan register for holding domain locking signals.

5. The emulation device of claim 1 further comprising a processor and a logic circuit for connection to the processor, said logic circuit operable to produce a done signal representing that the processor is done executing an instruction, wherein said second scan path includes a scan register for holding a selection signal determining whether the state machine circuitry is to be responsive to the done signal.

6. The emulation device of claim 1 for use with a test clock and a functional clock for a processor circuit wherein said second scan path includes a scan register and said state machine circuitry includes clock control circuitry coupling the test clock or the functional clock to the processor circuit depending on a signal in the scan register.

7. The emulation device of claim 1 for use with a test clock and a functional clock for circuitry having domains wherein said second scan path includes a scan register and said state machine circuitry includes clock control circuitry having respective outputs for the domains so that the clock control circuitry independently couples the test clock to one domain and the functional clock to another domain depending on a signal in the scan register.

8. The emulation device of claim 1 for use with circuitry having domains wherein said second scan path includes scan registers for holding command codes designating a selected domain and first and second command codes for the selected domain, wherein said state machine circuitry includes emulation control code registers for each of the domains, selection circuitry coupling said scan registers to the emulation control code registers, and a state machine connected to operate the selection circuitry.

9. The emulation device of claim 1 for use with circuitry to be emulated producing a done signal indicative of a predetermined electrical condition of the circuitry, wherein said state machine circuitry has respective inputs for a start signal from the testability interface and for the done signal.

10. The emulation device of claim 1 wherein said testability interface includes a decoding circuit having an output signaling to said state machine circuitry when a particular scan path is selected.

11. The emulation device of claim 1 or use with a test clock connected to said testability interface and a functional clock connected to said state machine circuitry and further comprising a handshake synchronizer connected between said testability interface and said state machine circuitry to produce a start signal for said state machine circuitry in response to said testability interface.

12. The emulation device of claim 1 wherein said first scan path includes a boundary scan path.

13. The emulation device of claim 1 for use with circuitry including shift register latches wherein said testability interface includes a third scan path interconnecting said shift register latches.

14. The emulation device of claim 1 for use with circuitry having domains, wherein said testability interface includes additional scan paths for each of the domains and said second scan path includes a serial register for lock signals, the emulation device further comprising switching circuits connecting the additional scan paths in response to the lock signals.

15. The emulation device of claim 1 wherein said serial scan testability interface includes an instruction register, and scan path selection circuitry responsive to said instruction register.

16. The emulation device of claim 15 wherein said serial scan testability interface includes a separate state machine connected to control said instruction register

and having a sequence of states responsive to an externally supplied digital signal.

17. An electronic device, comprising:

a semiconductor chip with an integrated circuit fabricated thereon, said integrated circuit being provided to support emulation of its own functional operation;

a serial scan testability interface on-ship having at least first and second scan paths, said first scan path being provided for applying digital information to said integrated circuit for use in emulation of said integrated circuit; and

state machine circuitry on-chip connected to said second scan path, said state machine circuitry having a sequence of states responsive to said second scan path and providing control signals for use in emulation of said integrated circuit.

18. The electronic device of claim 17 wherein said testability interface includes additional scan paths defining domains in the integrated circuit, and said second scan path includes a serial shift register for lock signals, the device further comprising switching circuits connecting the additional scan paths in response to the lock signals.

19. The electronic device of claim 17 wherein said second scan path includes scan registers for holding respective emulation command codes.

20. The electronic device of claim 17 wherein said integrated circuit includes a processor and a logic circuit connected to said processor, said logic circuit operable to produce a done signal for said state machine circuitry representing that the processor is done executing an instruction.

21. The electronic device of claim 17 wherein said testability interface has additional scan paths defining domains in the integrated circuit, said second scan path including scan registers holding command codes designating a selected domain and first and second command codes for controlling the selected domain, wherein said state machine circuitry includes emulation control code registers for each of the domains, selection circuitry coupling said scan registers to the emulation control code registers, and a state machine connected to operate the selection circuitry.

22. The electronic device of claim 17 further comprising a circuit responsive to said integrated circuit to produce a done signal indicative of a predetermined electrical condition of the integrated circuit, wherein said state machine circuitry has respective inputs for a start signal from the testability interface and for the done signal.

23. The electronic device of claim 17 wherein said testability interface includes a decoding circuit having an output to signal to said state machine circuitry when a particular scan path is selected.

24. The electronic device of claim 17 including a test clock connected to said testability interface and a functional clock connected to said state machine circuitry and further comprising a handshake synchronizer connected between said testability interface and said state machine circuitry to produce a start signal for said state machine circuitry in response to said testability interface.

25. The electronic device of claim 17 wherein said first scan path includes a boundary scan path.

26. The electronic device of claim 17 further comprising a test clock and a functional clock wherein said second scan path includes a scan register and said state machine circuitry includes clock control circuitry coupling the test clock or the functional clock to the integrated circuit depending on a signal in the scan register.

27. The electronic device of claim 26 wherein said clock control circuitry has respective outputs for different domains in the integrated circuit so that the clock control circuitry independently couples the test clock to one domain and the functional clock to another domain depending on the signal in the scan register.

28. An electronic system comprising a printed wiring board with serial-scan interconnected electronic devices, at least one of the electronic devices including a semiconductor chip with an integrated circuit fabricated thereon, said at least one electronic device being provided to support emulation of its own functional operation, a serial scan testability interface on-chip having at least first and second scan paths, said first scan path being provided for applying digital information to said at least one of the electronic devices for use in emulation of said at least one of the electronic devices, and said at least one of the electronic devices including state machine circuitry on-chip which is connected to said second scan path, said state machine circuitry having a sequence of states responsive to said second scan path and providing control signals for use in emulation of said at least one of the electronic devices.

29. The electronic system of claim 28 wherein said integrated circuit includes a processor and a logic circuit connected to said processor, said logic circuit operable to produce a done signal for said state machine circuitry representing that the processor is done executing an instruction.

30. The electronic device of claim 28 wherein said testability interface has additional scan paths defining domains in a said integrated circuit, said second scan path including scan registers holding command codes designating a selected domain and first and second command codes for controlling the selected domain, wherein said state machine circuitry includes emulation control code registers for each of the domains, selection circuitry coupling said scan registers to the emulation control code registers, and a state machine connected to said selection circuitry.

31. The electronic system of claim 28 further comprising a test clock and a functional clock wherein said second scan path includes a scan register and said state machine circuitry includes clock control circuitry coupling the test clock or the functional clock to the integrated circuit depending on a signal in the scan register.

32. The electronic system of claim 31 wherein said clock control circuitry has respective outputs for different domains in the integrated circuit so that the clock control circuitry independently couples the test clock to one domain and the functional clock to another domain depending on the signal in the scan register.

33. An electronic system comprising a host computer, a serial scan interface associated with said host computer for downloading testability codes and emulation command codes, and an electronic system connected to said serial scan interface and including a printed wiring board with at least one electronic device that includes a semiconductor chip with an integrated circuit fabricated thereon, said integrated circuit being provided to support emulation of its own functional operation, a serial scan testability interface on-chip having at least first and second scan paths for receiving said testability codes and emulation command codes respectively, and state machine circuitry on-chip which is connected to

said second scan path and which has a sequence of states responsive to said second scan path, wherein said sequence of states provides control signals for use in emulation of said integrated circuit.

34. The electronic system of claim **33** wherein said second scan path includes scan registers for holding respective emulation command codes.

35. The electronic system of claim **33** wherein said integrated circuit includes a processor and a logic circuit connected to said processor, said logic circuit operable to produce a done signal for said state machine circuitry representing that the processor is done executing an instruction.

36. The electronic system of claim **33** further comprising a test clock and a functional clock wherein said second scan path includes a scan register and said state machine circuitry includes clock control circuitry coupling the test clock or the functional clock to the integrated circuit depending on a signal in the scan register.

37. The electronic system of claim **36** wherein said clock control circuitry has respective outputs for different domains in the processor circuit so that the clock control circuitry independently couples the test clock to one domain and the functional clock to another domain depending on the signal in the scan register.

38. The electronic device of claim **33** wherein said testability interface has additional scan paths defining domains in the integrated circuit, said second scan path including scan registers holding command codes designating a selected domain and first and second command codes for controlling the selected domain, wherein said state machine circuitry includes emulation control code registers for each of the domains, selection circuitry coupling said scan registers to the emulation control code registers, and a state machine connected to said selection circuitry.

39. The electronic system of claim **33** wherein said testability interface includes additional scan paths defining domains in the integrated circuit, and said second scan path includes a serial register for lock signals, the device further comprising switching circuits connecting the additional scan paths in response to the lock signals.

40. The electronic system of claim **39** wherein said domains include a domain including a processor core, a domain including peripheral circuitry, and a domain including analysis circuitry.

41. A method of operating an emulation device with an integrated circuit which is segmentable into domains and which is provided to support emulation of its own functional operation, comprising the steps of downloading testability codes and emulation command codes to respective scan paths of the integrated circuit and sequentially executing the emulation command codes so that a first command code is executed and then a subsequent emulation command code is executed at a time depending upon completion of a predetermined electronic operation by the integrated circuit wherein at least one domain is operating functionally while at least one domain is operating in accordance with said emulation command codes.

42. A method of operating an emulation device so as to utilize an integrated circuit in support of its own emulation, comprising the steps of downloading emulation command codes to a scan path of the integrated circuit, the emulation command codes identifying different ones of a plurality of domains of the integrated circuit and which of a test clock and a functional clock is to be applied to each domain, and executing the emulation command codes to couple the test clock or the functional clock to the domains of the integrated circuit in accordance with the emulation command codes.

43. An electronic device, comprising:

a semiconductor chip having an integrated circuit fabricated thereon;

said integrated circuit including functional circuitry for performing normal operating functions of said integrated circuit, said auxiliary circuitry for performing auxiliary functions that provide information regarding the machine state of said functional circuitry, said auxiliary circuitry including state machine circuitry having an input which is accessible externally of said integrated circuit for permitting external control of said auxiliary functions;

said auxiliary circuitry further including a test architecture operatively associated with said functional circuitry and capable of executing operations according to a test methodology, said test architecture having a control interface through which said test architecture is controllable, said control circuitry being embedded within said integrated circuit; and

said auxiliary circuitry further including adaptor circuitry interfacing between said state machine circuitry and said test architecture, said adaptor circuitry including a serial scan path for receiving data selected by said state machine circuitry, said adaptor circuitry also including circuitry responsive to said state machine circuitry and said data in said serial scan path for selectively operating said embedded control interface of said test architecture.

44. The device of claim **43**, wherein said circuitry for selectively driving said embedded control interface includes further state machine circuitry responsive to said first-mentioned state machine circuitry and to said serial scan path for providing control signals for use in operating said embedded interface of said test architecture.

45. The device of claim **43**, wherein said state machine circuitry is a TAP controller according to IEEE STD 1149.1.

46. The device of claim **43**, wherein said state architecture is Modular Port Scan Design architecture.

47. The device of claim **43**, wherein said functional circuitry is operable at a first clock rate to perform said normal operating functions, wherein said data selected by said state machine circuitry is scanned through said serial scan path at a second clock rate which differs from said first clock rate, and wherein said auxiliary circuitry is operable to permit scanning data through said serial scan path at said second clock rate while said functional circuitry is concurrently performing normal operating functions at said first clock rate.

48. The device of claim **43**, wherein said functional circuitry is operable at a first clock rate to perform said normal operating functions, wherein said test architecture includes a further serial scan path extending through said functional circuitry for scanning data through said functional circuitry at a second clock rate which differs from said first clock rate, and wherein said adaptor circuitry is operable to permit scanning data through a portion of said functional circuitry at said second clock rate while another portion of said function circuitry is concurrently performing normal operating functions at said first clock rate.

49. The device of claim 45, wherein said adaptor circuitry includes a plurality of clock control circuits operatively associated with respective portions of said functional circuitry, wherein said further serial scan path includes a plurality of serially connectable scan path sections operatively associated with respective portions of said functional circuitry, and each said clock control circuit being operable independently of the remaining said clock control circuits for supplying a selected one of a first clock signal having said first clock rate for use by the associated portion of said functional circuit and a second clock signal having said second clock rate for use by the associated scan path section.

50. The device of claim 48, wherein said first-mentioned serial scan path and said further serial scan path have a common, externally accessible scan data input and are selectively enabled in response to operation of said state machine circuitry to receive data at said scan data input.

51. An electronic system, comprising:

a semiconductor chip having an integrated circuit fabricated thereon:

said integrated circuit having functional circuitry for performing normal operating functions of said integrated circuit, said auxiliary circuitry for performing auxiliary functions that provide information regarding the machine state of said functional circuitry, said auxiliary circuitry including state machine circuitry having an input which is accessible externally of said integrated circuit for permitting external control of said auxiliary functions;

said auxiliary circuitry further including a test architecture operatively associated with said functional circuitry and capable of executing operations according to a test methodology, said test architecture having a control interface through which said test architecture is controllable, said control interface being embedded within said integrated circuit;

said auxiliary circuitry further including adaptor circuitry interfacing between said state machine circuitry and said test architecture, said adaptor circuitry including a serial scan path for receiving data selected by said state machine circuitry, said adaptor circuitry also including circuitry responsive to said state machine circuitry and said data in said serial scan path for selectively operating said embedded control interface of said test architecture; and

wherein said electronic system includes a plurality of said semiconductor chips, said externally accessible inputs of said state machine circuitries being connected together to form a common externally accessible input, and said serial scan paths being connected together in series.

52. The system of claim 51, wherein said test architecture includes a further serial scan path extending through said functional circuitry for scanning data through said functional circuitry, said first-mentioned serial scan path and said further serial scan path having a common, externally accessible scan data input and a common, externally accessible scan data output, each said scan path being selectively enabled in response to operation of the associated state machine circuitry to receive input data at said scan data input and provide output data at said scan data output.

53. The system of claim 51, wherein one of said integrated circuits includes a digital signal processor (DSP) and another of said integrated circuits in an application specific integrated circuit (ASIC).

54. The system of claim 51, wherein one of said integrated circuits includes a microprocessor and another of said integrated circuits is an application specific integrated circuit (ASIC).

55. The system of claim 51, wherein one of said integrated circuits includes a graphics signal processor and another of said integrated circuits is an application specific integrated circuit (ASIC).

56. The system of claim 51, wherein one of said integrated circuits includes a graphics signal processor and another of said integrated circuits includes a digital signal processor (DSP).

57. The system of claim 51, wherein one of said integrated circuits includes a graphics signal processor and another of said integrated circuits includes a microprocessor.

58. The system of claim 51, wherein one of said integrated circuits includes a digital signal processor (DSP) and another of said integrated circuits includes a microprocessor.

59. The system of claim 51, wherein said semiconductor chips are provided on a printed circuit board.

60. The system of claim 51, wherein said functional circuitry of one of said integrated circuits differs from said functional circuitry of another of said integrated circuits.

61. The system of claim 60, wherein each of said integrated circuits has identical state machine circuitry.

62. The system of claim 51, wherein said adaptor circuitry of one of said integrated circuits differs from said adaptor circuitry of another of said integrated circuits.

63. The system of claim 62, wherein each of said integrated circuits has identical state machine circuitry.

64. The system of claim 51, wherein said test architecture of one of said integrated circuits differs from said test architecture of another of said integrated circuits.

65. The system of claim 64, wherein each of said integrated circuits has identical state machine circuitry.

66. A method of evaluating the functionality of an integrated circuit, comprising the steps of:

providing within the integrated circuit a test architecture capable of executing operations according to a test methodology;

embedding a control interface of the test architecture within the integrated circuit;

providing state machine circuitry within the integrated circuit;

accessing the state machine circuitry externally of the integrated circuit to select a serial scan path provided in the integrated circuit;

scanning data into the selected serial scan path; and

selectively operating the embedded control interface of the test architecture in response to operation of the state machine circuitry and to the data in the serial scan path.

67. The method of claim 66, including scanning data through one portion of said integrated circuit at a first clock rate while another portion of said integrated circuit is concurrently performing normal operation functions at a second clock rate which differs from the first clock rate.

68. The method of claim 66, including supplying one portion of said integrated circuit with a selected one of a first clock signal having a first clock rate for use in performing normal operating functions and a second

clock signal having a second clock rate different form said first clock rate for use in scanning data through said one portion, and performing said supplying step independently of any clock signal concurrently applied to any another portion of said integrated circuit.

69. The method of claim 66, including performing said scanning step while said integrated circuit is concurrently performing normal operating functions.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,329,471
DATED : July 12, 1994
INVENTOR(S) : Gary L. Swoboda et al.

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Claim 17, line 6, (Column 71); change "on-ship" to --on-chip--.

Claim 43, line 6, (Column 74); change "said" to --and--.

lines 18-19, (Column 74); change hyphenated word "circuitry" to --interface--.

Claim 46, line 1, (Column 74); change "state" to --test--.

Claim 48, line 10, (Column 74); change "function" to --functional--.

Claim 49, line 1, (Column 75); change "45" to --48--.

Claim 51, line 6, (Column 75); change "said" to --and--.

Claim 53, line 3, (Column 76); change "in" to --is--.

Claim 68, line 5, (Column 77); change "form" to --from--.

Signed and Sealed this

Eleventh Day of April, 1995

Attest:

Bruce Lehman

BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks

EXHIBIT C

US005437027A

United States Patent [19]

Bannon et al.

[11] Patent Number: 5,437,027

[45] Date of Patent: * Jul. 25, 1995

[54] SYSTEM AND METHOD FOR DATABASE MANAGEMENT SUPPORTING OBJECT-ORIENTED PROGRAMMING

[75] Inventors: **Thomas J. Bannon**, Dallas; **Stephen J. Ford**; **Vappala J. Joseph**, both of Plano; **Edward R. Perez**, Dallas; **Robert W. Peterson**; **Diana M. Sparacin**, both of Plano; **Satish M. Thatte**, Richardson; **Carig W. Thompson**, Plano; **Chung C. Wang**; **David L. Wells**, both of Dallas, all of Tex.

[73] Assignee: **Texas Instruments Incorporated**, Dallas, Tex.

[*] Notice: The portion of the term of this patent subsequent to Mar. 22, 2011 has been disclaimed.

[21] Appl. No.: **110,040**

[22] Filed: **Aug. 20, 1993**

Related U.S. Application Data

[62] Division of Ser. No. 531,493, May 30, 1990, Pat. No. 5,297,279.

[51] **Int. Cl.**[6] **G06F 17/30**

[52] **U.S. Cl.** **395/600**; 395/700; 364/DIG. 1; 364/282.1; 364/283.1; 364/283.4

[58] **Field of Search** 395/425, 600, 500, 650, 395/700, 725

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,525,780	6/1985	Bratt et al.	395/650
4,853,842	8/1989	Thatte et al.	395/425
4,989,132	1/1991	Mellender et al.	395/425
5,075,842	12/1991	Lai	395/425
5,075,845	12/1991	Lai et al.	395/425
5,075,848	12/1991	Lai et al.	395/425
5,079,695	1/1992	Dysart et al.	395/700
5,297,279	3/1994	Bannon et al.	395/600

OTHER PUBLICATIONS

Maier, D. et al, "Development of an Object Oriented DBMS", Object Oriented Programming; Systems, Languages & Applications (OOPSLA), 1986, p. 472+.

Peter Lyngbaek, et al. "A Data Modeling Methodology for the Design and Implementation of Informaiton Systems", Int'L Workshop on Object-Oriented Databases Systems, 1986, p. 6+.

Kevin Wilkinson, et al. "The IRIS Architecture and Implementation", IEEE Trans. on knowledge and Data Engineering, V2N1 Mar. 1990, p. 63+.

Timothy Andrews, et al, "Combining Language and Database Advances in an Object-Oriented Development Environment", Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1987, p. 430+.

Won Kim, et al. "Integrating an Object-Oriented Programming System with a Database System", Object Oriented Programming: Systems, Languages and Applications (OOPSLA) 1988, p. 142+.

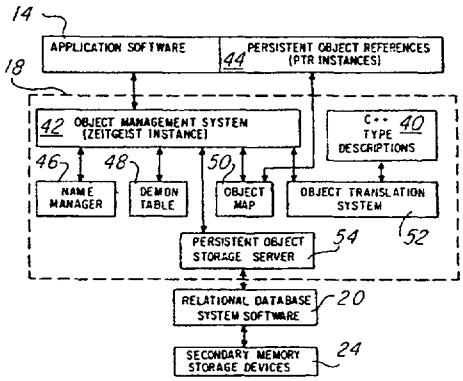
(List continued on next page.)

Primary Examiner—Thomas G. Black
Assistant Examiner—John C. Loomis
Attorney, Agent, or Firm—Ruben C. DeLeon; James C. Kesterson; Richard L. Donaldson

[57] **ABSTRACT**

A system and method for database management for providing support for long-term storage and retrieval of objects created by application programs written at least in part in object-oriented programming languages consists of a plurality of software modules. These modules provide data definition language translation, object management, object translation, and persistent object storage service. Such system implements an object fault capability to reduce the number of interactions between the application, the database management system, and the database.

21 Claims, 5 Drawing Sheets



OTHER PUBLICATIONS

Won Kim, et al. "Architecture of the ORION Next--Generation Database Syhstem", IEEE Trans. on knowledge and Data Engineering, V2N1 Mar. 1990, p. 109+.

Michael Stonebraker, "Object Management in POSTGRES Using Procedures", Int'l Workshop on Object-Oriented Database (0008) Systems 1986, p. 66+.

Michael Stonebraker, et al. "The Implementation of POSTGRES", IEEE Trans. on knowledge and Data Engineering, V2N1, Mar. 1990, p. 125+.

Puknraj Kachhwaha, et al. "An Object-Oriented Data Model for the Research Laboratory", Int'l Workshop on Object-Oriented Database (0008) Systems 1986, p. 218.

Puknraj Kachhwaha, "LCE: An Object-Oriented Database Application Developemtn Tool", SIGMOD Int'l Conference on Management of Data 1988, p. 207.

Laura M. Haas, et al. "Starburst Mid-Flight: As the Dust Clears", IEEE Trans. on knowledge and Data Engineering, V2N1 Mar. 1990, [IBM's Starburst] p. 143+.

Ted Kaehler, "Virtual Memory on a Narrow Machine for an Object-Oriented Language", Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1986. [Xerox PARC's LOOM] p. 87+.

Karen E. Smith, et al. "Intermedia: A Case Study of the Difference Between Relational and Object-Oriented Database Systems", Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1987. [Brown's Intermedia System] p. 452+.

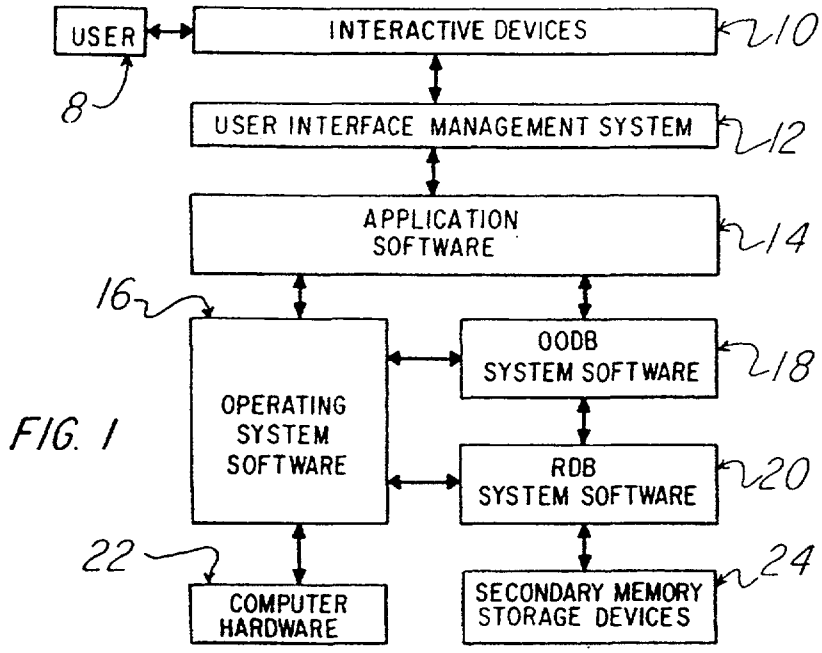

USER
INTERACTIVE DEVICES
10
8
USER INTERFACE MANAGEMENT SYSTEM
12
APPLICATION SOFTWARE
14
16
OPERATING SYSTEM SOFTWARE
OODB SYSTEM SOFTWARE
18
RDB SYSTEM SOFTWARE
20
22
COMPUTER HARDWARE
SECONDARY MEMORY STORAGE DEVICES
24

FIG. 1

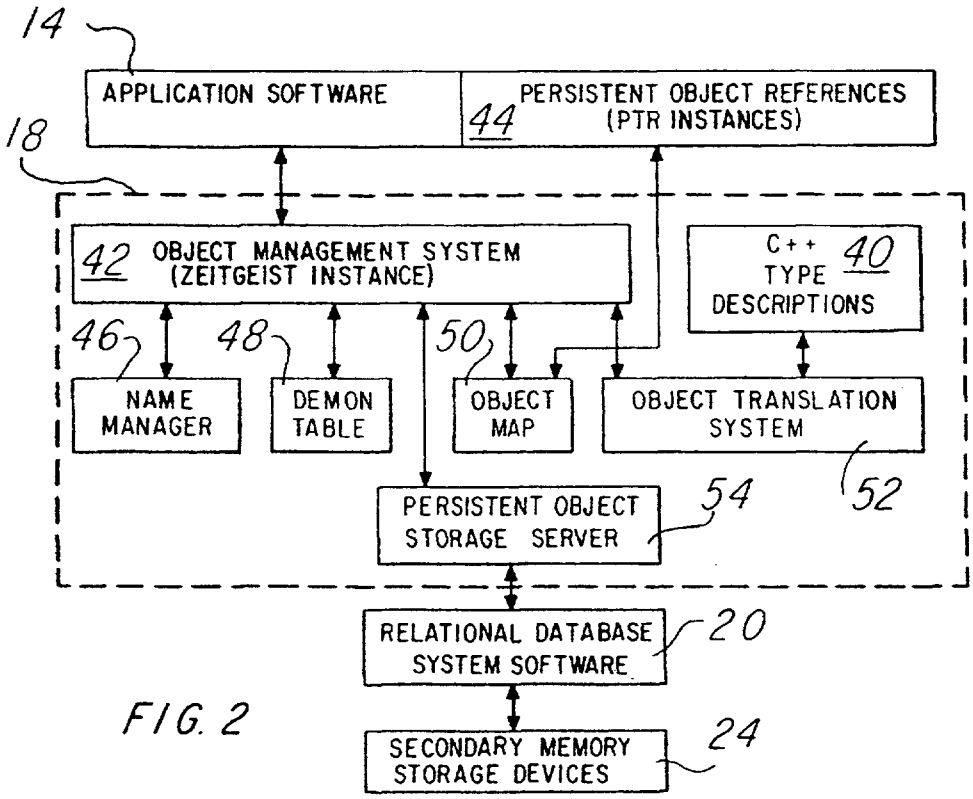

14
APPLICATION SOFTWARE
PERSISTENT OBJECT REFERENCES (PTR INSTANCES)
44
18
42
OBJECT MANAGEMENT SYSTEM (ZEITGEIST INSTANCE)
C++ TYPE DESCRIPTIONS
40
46
48
50
NAME MANAGER
DEMON TABLE
OBJECT MAP
OBJECT TRANSLATION SYSTEM
52
PERSISTENT OBJECT STORAGE SERVER
54
RELATIONAL DATABASE SYSTEM SOFTWARE
20
SECONDARY MEMORY STORAGE DEVICES
24

FIG. 2

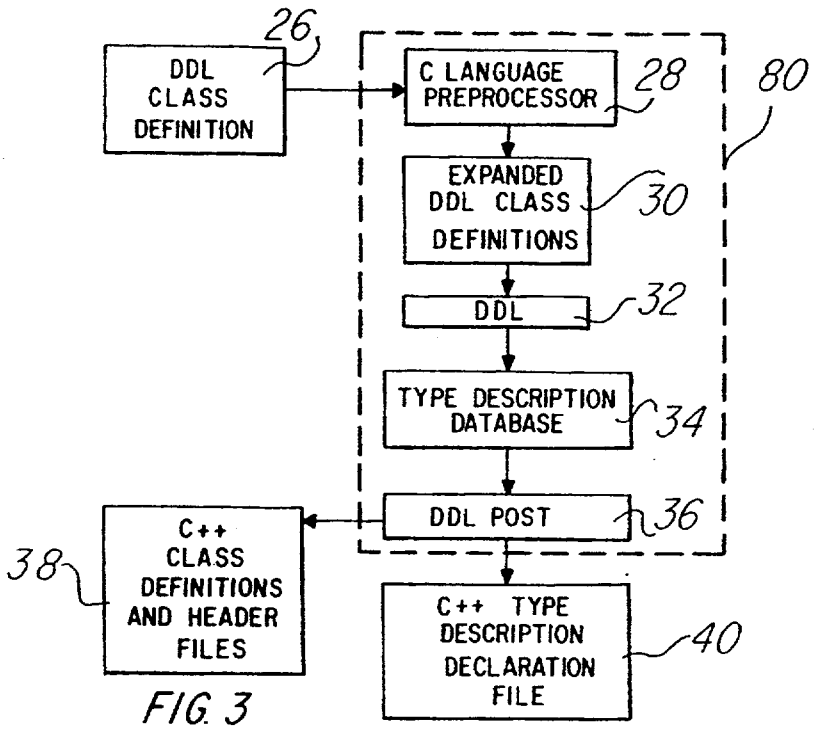

26
DDL CLASS DEFINITION
C LANGUAGE PREPROCESSOR
28
80
EXPANDED DDL CLASS DEFINITIONS
30
DDL
32
TYPE DESCRIPTION DATABASE
34
DDL POST
36
38
C++ CLASS DEFINITIONS AND HEADER FILES
C++ TYPE DESCRIPTION DECLARATION FILE
40

FIG. 3

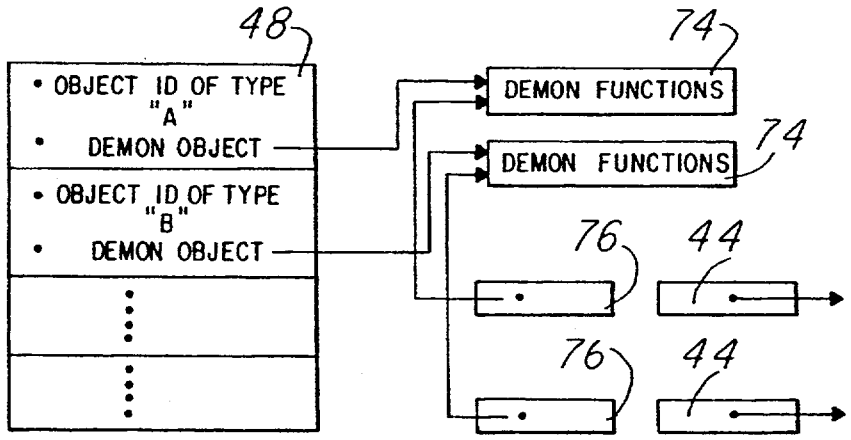

48
• OBJECT ID OF TYPE "A"
• DEMON OBJECT
• OBJECT ID OF TYPE "B"
• DEMON OBJECT
74
DEMON FUNCTIONS
DEMON FUNCTIONS
74
76
44
76
44

FIG. 4

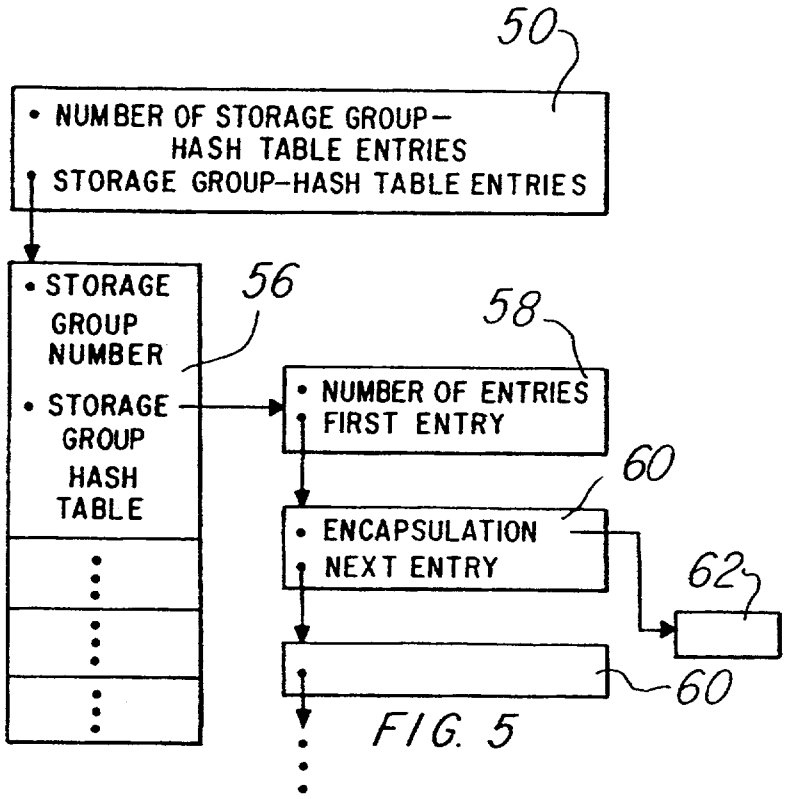

50
• NUMBER OF STORAGE GROUP-HASH TABLE ENTRIES
• STORAGE GROUP-HASH TABLE ENTRIES
• STORAGE GROUP NUMBER
• STORAGE GROUP HASH TABLE
56
58
• NUMBER OF ENTRIES
• FIRST ENTRY
60
• ENCAPSULATION
• NEXT ENTRY
62
60

FIG. 5

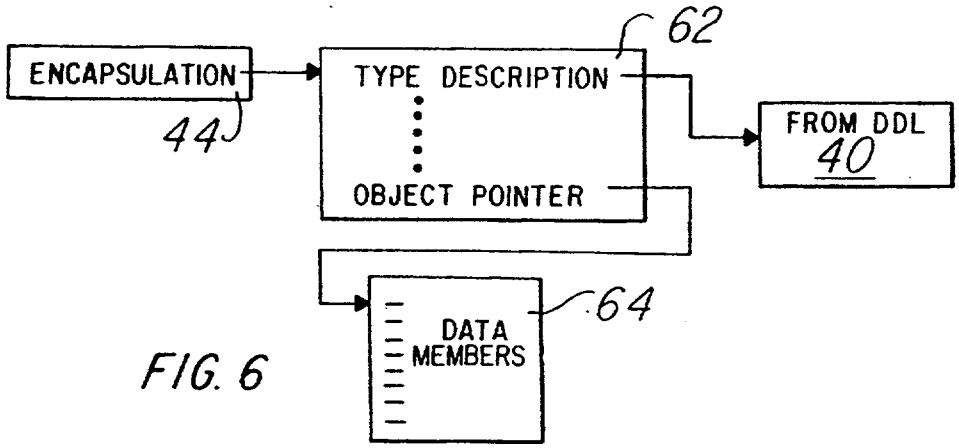

ENCAPSULATION
44
62
TYPE DESCRIPTION
OBJECT POINTER
FROM DDL
40
DATA MEMBERS
64

FIG. 6

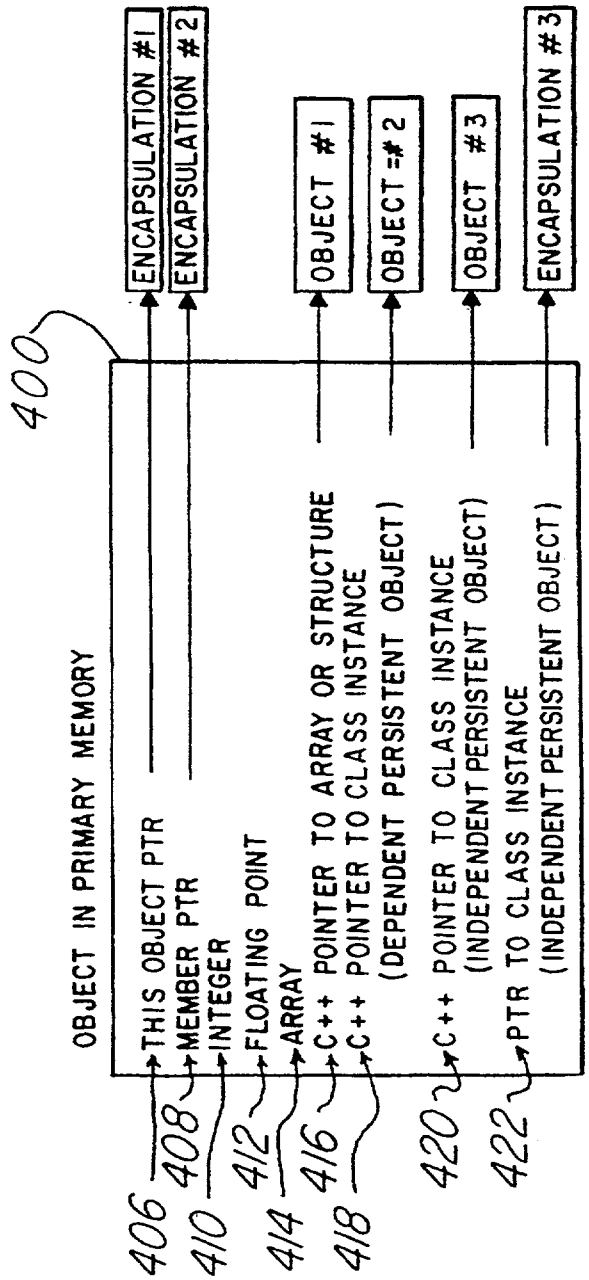

400
OBJECT IN PRIMARY MEMORY
THIS OBJECT PTR
MEMBER PTR
INTEGER
FLOATING POINT
ARRAY
C++ POINTER TO ARRAY OR STRUCTURE
C++ POINTER TO CLASS INSTANCE
(DEPENDENT PERSISTENT OBJECT)
C++ POINTER TO CLASS INSTANCE
(INDEPENDENT PERSISTENT OBJECT)
PTR TO CLASS INSTANCE
(INDEPENDENT PERSISTENT OBJECT)
406
408
410
412
414
416
418
420
422
ENCAPSULATION #1
ENCAPSULATION #2
OBJECT #1
OBJECT #2
OBJECT #3
ENCAPSULATION #3

FIG. 7a

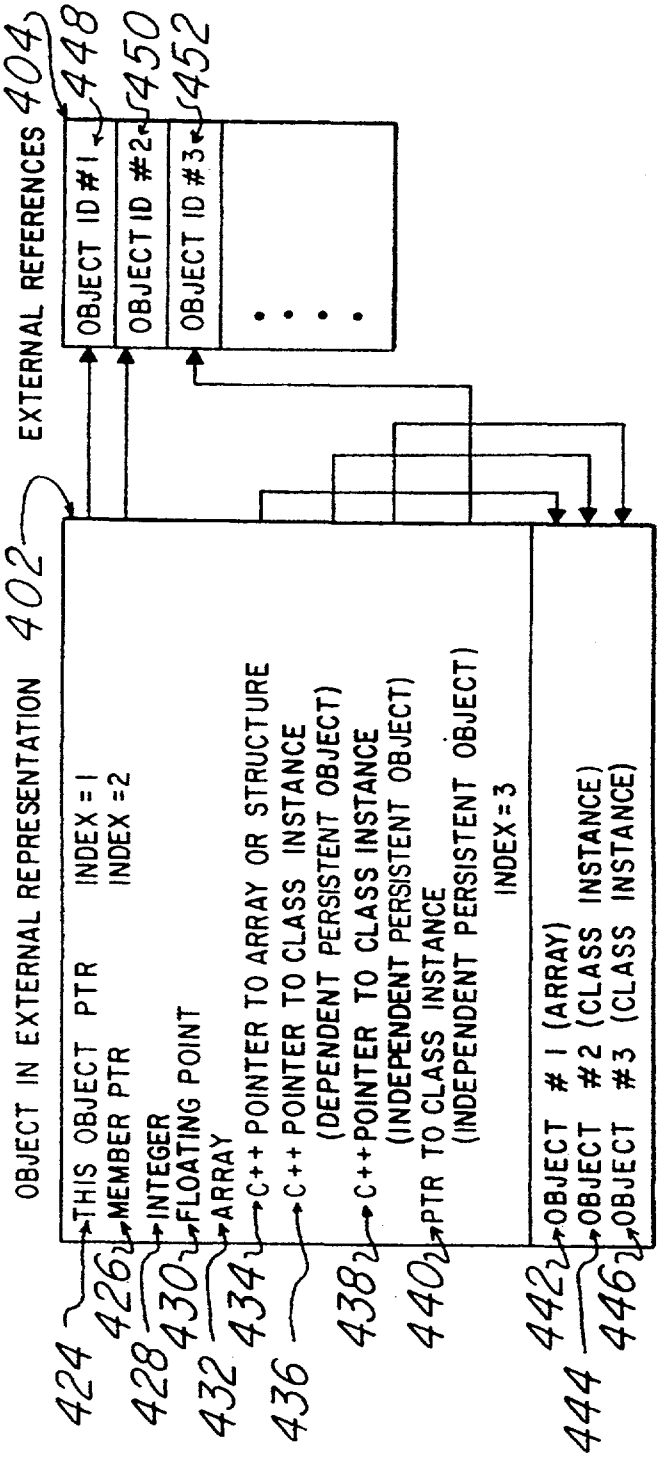

EXTERNAL REFERENCES 404
OBJECT ID #1 448
OBJECT ID #2 450
OBJECT ID #3 452
OBJECT IN EXTERNAL REPRESENTATION 402
THIS OBJECT PTR INDEX = 1
MEMBER PTR INDEX = 2
INTEGER
FLOATING POINT
ARRAY
C++ POINTER TO ARRAY OR STRUCTURE
C++ POINTER TO CLASS INSTANCE (DEPENDENT PERSISTENT OBJECT)
C++ POINTER TO CLASS INSTANCE (INDEPENDENT PERSISTENT OBJECT)
PTR TO CLASS INSTANCE (INDEPENDENT PERSISTENT OBJECT) INDEX = 3
OBJECT # 1 (ARRAY)
OBJECT #2 (CLASS INSTANCE)
OBJECT #3 (CLASS INSTANCE)
424
426
428
430
432
434
436
438
440
442
444
446

FIG. 7b

SYSTEM AND METHOD FOR DATABASE MANAGEMENT SUPPORTING OBJECT-ORIENTED PROGRAMMING

NOTICE

©1990 Texas Instruments Incorporated. A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent disclosure, as it appears in the Patents and Trademark Office patent file or record, but otherwise reserves all copyright rights whatsoever.

This is a Divisional of application Ser. No. 07/531,493, filed May 30, 1990, now U.S. Pat. No. 5,297,279.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to database management systems and more particularly to a system and method providing support for long-term storage and retrieval of objects created by application programs written in object-oriented programming languages.

2. Description of Related Art

Many new computer software applications, such as Computer-Aided Design and Manufacturing, Computer-Aided Software Engineering, multimedia and hypermedia information systems, and Artificial Intelligence Expert systems, have data models that are much more complex than previous systems, both in content and inter-object relationships. Object-oriented languages provide the application developer the mechanism to create and manipulate the data models inherent in these applications. Database systems provide long term storage of the data created by these applications. However, existing languages and databases are insufficient to develop these applications because existing object-oriented languages do not provide direct support for long-term storage and sharing of objects, existing commercial database systems (hierarchical, network, and relational) do not support the necessary complex object-oriented data models, and existing database systems require an application developer to use different languages and modeling paradigms when building applications.

There have been various research and commercial efforts aimed at developing OODBs. These OODBs vary in type of data model employed, application program interaction, object access method, method of persistent object store, etc. Examples of these current OODBs, and their weaknesses, will now be considered.

Iris (Hewlett Packard) and GemStone (Servio Corporation) are representative OODBs employing new proprietary object-oriented data models, while Vbase (Ontologic), Orion (Microelectronics and Computer Technology Corporation), and DOME (Dome Software Corporation) are examples of OODBs incorporating proprietary extensions to existing programming language data models. In these types of OODBs, application developers are required to learn a new proprietary data model in order to effectively use the OODB. Since their data model is new, using it often results in a loss of productivity as application developers learn the new language. In Orion, instances of user-defined classes cannot be stored in the database unless they have been derived from Orion-defined classes. In addition, GemStone and Orion do not allow for an instance of

their classes to be transient; that is, every object created in a GemStone- or Orion-based application will be stored in the database unless it is specifically deleted. Another problem with developing a new data model is that it requires application developers to rely on a single source of application development tools, such as language compilers, object libraries, and program debuggers, which limits widespread acceptance of these OODBs.

POSTGRES (University of Berkeley) is an example of an OODB employing another type of data model, that of a proprietary extension to an existing relational database. POSTGRES is a combination of an extended relational and object-oriented database. Objects are created using relational table descriptions, while functions to manipulate the objects are created using the POSTQUEL query language as well as conventional languages (C and LISP). In addition, if the application developer wishes to add indices over user-defined types, they must write and register with POSTGRES functions to perform the various comparison operations between two objects of the same user defined type. Since the latter mode of creating functions requires the application developers to map between the POSTGRES and C/LISP data models, which can be error prone and distracting from the task of developing the application system, this strategy does nothing to alleviate the burdensome requirement to use different languages and modeling paradigms when building applications. This problem of mapping between the object-oriented and relational data models was discussed in-depth in the Intermedia OOPSLA '87 conference paper.

Ontos(Ontologic) and Object Store(Object Design) are representative of OODBs employing the last type of data model, namely the use of an existing programming language data model (e.g., using the C++ programming language data model for writing software programs and interacting with the database). Both systems, however, require the use of a proprietary language compiler to add additional code (Ontos) or translate new and non-standard C++ language constructs (Object Store). As with the first two types of OODBs, this approach requires application developers to rely on a single source of application development tools, which also limits widespread acceptance of these OODBs.

In addition to problems inherent with the type of data models selected, difficulties occur when an application program interacts with an OODB. In the Iris OODB, application developers define object types and develop functions to manipulate the objects using the proprietary Iris language. Iris provides an interactive interface where requests can be made to retrieve or manipulate Iris objects. The requests are evaluated by performing relational queries (since the objects are stored in relational tables) and the result is returned as an Iris expression, not as object values or references. Iris provides an embedded object SQL interface, a C language interface (which is not object-oriented), and allows the application developer to register foreign functions written in existing (possibly non-object-oriented) programming languages. These approaches require the application developer to map the Iris objects into data structures accessible by the programming language, reintroducing the problems discussed above.

Similarly, the developers of the GemStone OODB also defined a new language, OPAL, which the application developer uses to define object types and functions

to manipulate the objects. GemStone provides an interactive development environment for developing OPAL objects and functions. GemStone also provides a mechanism for existing programming languages (C and Smalltalk) to interact with GemStone. However, unless the applications developer uses only the OPAL language, two data models and languages must be used to interact with the database, mapping the OPAL objects into structures accessible by the programming language, and thereby resulting in the problems associated discussed previously.

The Vbase OODB requires two separate languages, TDL to define object types, and COP (an extension to the C programming language) to develop application programs. Although application developers do not need to map objects between the data model and the programming language, they must still use two languages during the development of their programs, with the attendant problems considered above. A further restriction of this system includes the failure to provide access to the database from other programming languages.

Although the Orion OODB developers used an existing programming language, Common Lisp, for their data model, they developed several proprietary extensions to the language. As with Vbase, there is no need to map between the data model and programming language with the Orion OODB. However, this approach requires the use of a proprietary language translator.

The developers of POSTGRES, on the other hand, expect most application developers to write programs that interact with file database primarily using the POSTGRES query language, POSTQUEL. Navigation between objects is possible; however, a query must be issued to perform the navigation instead of accessing the referenced object directly. Application developers can define and implement their own functions including programming language statements, POSTQUEL query statements, and/or calls to POSTGRES' internal functions. Thus, application developers may have to deal with two or more data models to build their application systems. Such requirement fails to alleviate the problems considered above.

The Ontos approach provides an interface from the C++ language to the database. However, the amount of interaction between the program and Ontos is much higher than is reasonable or necessary due to the requirement of specialized functions that must be provided by the application developer (e.g., object construction, translation, storage/retrieval, etc.). This burdens the application developer with more work that could have been performed by the database system. Object Store also provides an interface from the C++ language to the database. However, the interface is accomplished by redefining the semantics of or adding new C++ language constructs, thereby requiring the use of Object Design's proprietary C++ language translator, which limits widespread acceptance of their system.

Access to an object in an OODB is performed by manipulating the object using predefined functions, using an explicit query, or by coding explicit references in a programming language.

In the Iris OODB, application developers call functions to retrieve or change values in the object. A program cannot receive a reference to an object which could be passed to other functions. In the GemStone, Vbase, and Orion OODBs, individual objects can be accessed and passed to functions to retrieve or assign values.

In the POSTGRES database, application developers perform queries to retrieve or change values in the object (actually, relational tuples). POSTGRES allows a foreign function to access an object, but as stated above, it must be mapped from the relational data model to the data model of the foreign function's programming language.

Although most OODBs allow the application developer to explicitly retrieve an object from the database (Iris and POSTGRES do not), they do not allow the application developer to specify when objects related to the original object should be retrieved. For example, application developers can access objects in Ontos using one of two modes. In the first mode, an object is explicitly retrieved and referenced objects are implicitly retrieved using an object fault capability. In the other mode, one or more related objects can be explicitly retrieved, but the application must continually check to see if a referenced object is already in memory, and then explicitly retrieve it if it is not. This requires the application developer to employ two completely different models of accessing persistent objects in the same program, which can easily cause errors in the program by the inadvertent and natural use of one mode where the other mode should have been used.

The approach taken by Object Store is quite different from the above OODBs with regard to object access. Object Store's model is more like a persistent memory (an extension of virtual memory computer operating system) than an OODB. Object Design chose to completely reimplement the virtual memory management functions of the C++ programming language and the UNIX (TM) operating system. Whenever a persistent object is created or retrieved from the database, it is installed in a portion of primary memory controlled by Object Design. Thus, references to the object are, in essence, monitored by Object Design's software. If the object is not currently in primary memory, it will be retrieved from the database and installed in primary memory. This style of memory management requires that any class or class library requiring persistence must be written using this memory management scheme, or perform no dynamic memory management thereby resulting in one version of the library for persistent usage and one version for transient usage. Although this approach improves the object storage and retrieval performance, it is inherently dependent on the underlying computer operating system and memory architecture, and thus not portable to other computer systems.

Therefore, these approaches either limit how an application program can access an object, or require additional work in order for the program to access an object.

Most OODBs (except for Iris and DOME) have developed their persistent object storage facility utilizing an existing file management system. They had to develop new implementations of the disk storage structures and management, concurrency control, transaction management, communication, and storage management subsystems. This approach increases the complexity of the overall database system software.

The Iris and DOME OODBs, on the other hand, use existing commercial Relational Database Management Systems (RDBMS) to store their objects. Although the Iris OODB uses Hewlett Packard's relational database HP-SQL, it does not use the SQL interface to that data-

base, restricting access to the objects to the available Iris functions, Iris interactive browser, C language interface, and embedded Iris SQL. Although Iris allows the application developer to define how objects are to be stored, the use of Hewlett Packard's RDBMS imposes a limit on the size of an object. The DOME OODB, which uses Oracle Corporation's Oracle RDBMS, and the POSTGRES system, which has its own relational storage system, decomposes objects into one or more entries in one or more relational tables. This approach requires a relational join whenever more than one attribute value from an object is retrieved. Relational join operations are computationally expensive.

In the GemStone and Object Store OODBs, the unit of concurrency control is not an object but a secondary memory segment, or page. This approach can improve the performance of secondary memory reads and writes, but results in having the storage facility read, write, and lock more data than may be necessary. In addition, this restricts the amount of concurrent access to objects since the OODB system, and not the application developer, chooses the unit of concurrency control.

Most of the OODBs allow related objects to be clustered together in the persistent object storage. GemStone and Orion only allow clustering controls to be specified when the entire database is defined. Vbase and Ontos allow runtime specification of clustering controls to store one persistent object as close as possible to another persistent object. Object Store also allows runtime specification of clustering controls to store statically allocated objects in a specific database and dynamically allocated objects in a specific database or as dose as possible to another persistent object. This requires the application developer to treat similar objects with different models of clustering, which can cause errors in the program by the inadvertent use of one mode where the other mode should have been used. These systems indicate that such clustering specifications are purely hints which the system may ignore. These clustering hints may require rebuilding of the database if they are changed, thereby restricting the ability of the application developers to tune the database's performance by altering the physical grouping of objects. Furthermore, the systems based on relational storage, such as Iris, POSTGRES, and DOME, do not allow user-defined clustering of objects.

SUMMARY OF THE INVENTION

In view of the above problems associated with the related art, it is an object of the present invention to provide a database management system and method which supports long term storage and retrieval of objects created by application programs, and which uses existing object-oriented programming languages to thereby enable such system and method to be ported to other computer platforms without requiring any modifications to existing language translators or computer operating systems and thereby not unduly restrict application developers in their choice of computer platform or language translator.

It is a further object of the present invention to provide a database management system and method providing a standard object-oriented programming interface for its database functionality, thereby eliminating any requirement for mixing of object-oriented and functional, or other, programming styles to confuse the application developer when coding a program's interface to that of the present invention.

It is yet another object of the present invention to provide a database management system and method for adding persistence to existing language objects orthogonally, thereby allowing application programmers to treat persistent and nonpersistent objects in nearly the same manner and eliminating the need to use two or more data models when building application systems.

Another object of the present invention is to provide a database management system and method that allows the application developer to specify at object definition time how related objects, whether created dynamically or statically, should be clustered when stored, to thereby provide a capability to adjust the size of storage objects to enhance the overall system performance.

Still another object of the present invention is to provide a database management system and method that reduces the number of interactions with the database management system that an application developer must code to access objects stored in the database.

It is a further object of the present invention to provide a database management system and method that allows the application developer to specify at application execution time prior to saving a persistent object whether or not to install in primary memory the persistent objects referenced from the given persistent object at the same time when the given object is later installed in primary memory, either due to explicit or implicit retrieval, to enhance the overall system performance.

A further object of the present invention is to provide maximization of concurrent usage of the objects in the database by making the unit of locking the individual persistent object instead of a page of persistent objects.

It is still another object of the present invention to store objects in a persistent object storage server utilizing a relational database management system by storing an external representation of the object and external references from the object without decomposing the objects into multiple relational tuples, to enhance the overall system performance.

Yet another object of the present invention is to provide a database management system and method which uses a uniform object translation methodology thereby eliminating the need for application developers to perform this complex computer-and language-dependent task.

In accordance with the above objects of the invention, the preferred embodiment of the present invention consists of four software modules to provide database services to application developers. They are referred to as the Data Definition Language (DDL) translator, the Object Management System (OMS), the Object Translation System (OTS), and the Persistent Object Storage Server (POS Server).

The present invention presents an application interface for programming languages which does not require any extensions to the languages, modifications to existing language translators, or development of proprietary language translators. Furthermore, the present invention implements an object fault capability which reduces the number of interactions that an application must perform with the database management system and database itself. Access of, and navigation between, objects can be performed using existing language operations in a transparent manner.

Furthermore, instead of requiring the application developers to use one data model to interact with the

database and another data model to manipulate the objects in a programming language, the present invention uses the data model of existing standard object-oriented languages, such as C++ and CLOS, as the data model for the database. This alleviates problems associated with the art discussed above.

Although the present invention can be implemented in any object-oriented programming language, and should therefore not be limited in any way to any specific language, it has been implemented in both C++ and Common Lisp. In the C++ embodiment, application developers interact solely with the DDL module, in a batch processing mode, and with the OMS module using standard C++ syntax in their application programs. The DDL module accepts object type descriptions on standard C++ programming language statements (with a few additional syntactic constructs) and extracts sufficient information from the descriptions to enable the OTS module to translate objects between their primary and secondary memory representations. This process is required because this type description information is not available in the C++ run-time system. To achieve architecture-independent translation, the DDL translator also accepts information describing a specific computer architecture and software system environment in which the present invention's applications are to be executed. The POS Server uses a standard SQL interface to a commercial relational database.

In the Common Lisp embodiment, application developers interact solely with the OMS module using standard Common Lisp syntax in their application programs. The DDL module is not implemented since the OTS module can extract the necessary information from the CLOS descriptions during program execution as that information is already available in the Common Lisp run-time system. This embodiment uses a raw disk-based implementation of the POS Server developed by the co-inventors.

The OMS module presents an application interface to perform standard database operations: initializing and terminating the present invention, beginning and committing or aborting database transactions (saving modified objects or discarding them, respectively), designating objects as persistent (to be saved to the database), explicitly retrieving objects from the database, designating objects as having been modified, removing objects from memory, defining the default storage group for logical clustering of objects, etc. The OMS module also supports an automatic and implicit retrieval of objects from the database when an application references a previously saved object that is currently not in primary memory. OMS also provides a facility to associate user-defined names with persistent objects to simplify retrieval of objects. These associations are also stored in the present invention's database. This name-object management module has a well-defined interface and can be replaced with a module of the application developer's choice.

As stated above, the present invention allows the application program to retrieve persistent objects from the database and reference the persistent object's data members or functions. The present invention accomplishes this by defining a new data type or class, the ZG_PTR, that functions equivalently to the current language constructs for referencing persistent objects (pointers in C++; symbols and values in Common Lisp). In addition, the present invention allows the application program to implicitly retrieve a persistent object from the database using an object faulting mechanism. When an application program references a persistent object, if the object is already in primary memory, the application program continues with its operations. If the object is not in primary memory, OMS automatically retrieves the object from the POS and calls upon the OTS module to translate and install the object in primary memory. Finally, the application program is allowed to proceed, unaware of this object faulting processing.

The OTS module is responsible for translating objects between their primary and secondary memory representations in a computer architecture-independent manner. When an object is being saved, the OTS module uses the information extracted by the DDL translator to determine the extent, or boundary, of the object and then translates all of the objects within the boundary to a computer architecture independent representation. When an object is retrieved from the POS, OTS creates the appropriate primary memory representation, assigns the object's values from the stored representation, and allocates OMS data structures for every reference contained in the object to other persistent objects.

The POS Server module provides a stable storage facility for the objects made persistent by the application program. Objects are stored in the computer's long term, or secondary, memory. The POS Server also provides to the OMS module concurrency control primitives and atomic transactions (all objects are saved or none are saved). Objects are stored as an untyped array of bytes which only OTS understands.

The present invention stores objects via the POS Server in a computer architecture-independent representation utilizing information about the computer's computational, or primary, memory architecture. Information on the content and structure of the objects is extracted from the object definitions declared in the supported languages. This allows applications written in any of the supported languages to store objects in the same POS. Currently the POS Server is implemented in a modular and portable fashion using an existing commercial Relational Database Management System (RDBMS). The POS Server interacts with the RDBMS using an embedded Structured Query Language (SQL) interface.

These and other features and advantages of the invention will be apparent to those skilled in the art from the following detailed description of a preferred embodiment, taken together with the accompanying drawings in which:

BRIEF DESCRIPTION OF DRAWINGS

FIG. 1 is a block diagram showing the operating context of the present invention operates within a computer;

FIG. 2 is a block diagram of the architecture of a preferred embodiment of the present invention;

FIG. 3 is a flow chart depicting the process flow of the DDL translator during execution according to the present invention;

FIG. 4 is a block diagram representing an example of a demon table used within the OMS module of the present invention;

FIG. 5 is a block diagram representing an example of an object map used within the OMS module of the present invention;

FIG. 6 is a block diagram depicting an example of the relationships between a PTR, an encapsulation, an ob-

ject, and the type description of that object according to the present invention; and

FIGS. *7a* and *7b* are block diagrams showing the process of object translation between its primary and secondary memory representations according to the present invention.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT

The present invention is applicable in the development of object-oriented application systems which require the management, persistence and sharing of complex and interrelated data. Before considering the present invention in detail, however, it should be noted that although the C++ implementation will be fully described, the preferred embodiment of the present invention can be implemented in any object-oriented programming language. Currently, a C++ implementation of the present invention runs on systems such as the Sun Microsystems Incorporated Sun4 or the Digital Equipment Corporation DEC3100 series of computer workstations. Also, a Common LISP implementation of the present invention runs on such systems as the Texas Instruments Incorporated Explorer series of computer workstations.

FIG. **1** shows a block diagram of the operating context of the present invention within a computer. A user **8**, which may be human, another computer, or another application, interacts with the interactive devices **10** to send information to, and receive information from, user interface management system **12**. Interactive devices **10** are also known as communication hardware. User interface management system **12**, which is also known as communication software in turn sends the information to, and receives information from, an application software program **14** (hereinafter referred to as "application **14**"). Application **14** interfaces with the computer's operating system software **16** (hereinafter referred to as "OS **16**") and the present invention (hereinafter referred to as "OODB **18**"). OODB **18** interfaces with OS **16** to utilize various operating system services. OODB **18** also interfaces with a Relational Database Management System Software **20** (hereinafter referred to as "RDBMS **20**") to store and retrieve objects created by application **14**. RDBMS **20** interfaces with operating system **16** to utilize various OS system services and with secondary memory storage devices **24** to physically store or retrieve the objects created by application **14** and managed by OODB **18**. OS **16** also interfaces with additional computer hardware **22** as necessary to provide its system services to application **14**, OODB **18**, and RDBMS **20**.

FIG. **2** shows a block diagram of the architecture of a preferred embodiment of the present invention during execution of application **14**. All of the modules in OODB **18** are linked together along with a C++ type description **40** (hereinafter referred to as "types **40**"), produced by running the data definition language translator (hereinafter referred to as "DDL **80**"; shown in FIG. **3**) on a set of application-defined classes, to form a library that can be linked with application **14** to form a program that can be executed in the computer environment shown in FIG. **1**.

Application **14** interfaces directly with the object management system **42** (hereinafter referred to as "OMS **42**") and creates or deletes Persistent Object References **44** (hereinafter referred to as "PTR **44**") to create, retrieve, and access persistent objects managed by OMS **42**.

OMS **42** interfaces with name manager **46** to manage an association of names supplied by application **14** with independent persistent objects created or retrieved by application **14** using PTRs **44**. OMS **42** interfaces with demon table **48** to execute functions defined by application **14** and registered with OODB **18** by running DDL **80**. OMS **42** also interfaces with object map **50** to manage the actual persistent objects referenced by PTR **44**. OMS **42** further interfaces with object translation system **52** (hereinafter referred to as "OTS **52**") to translate persistent objects between their primary and secondary memory representations. OTS interfaces with types **40** created by DDL **80**. Finally, OMS **42** interfaces with persistent object server **54** (hereinafter referred to as "POS Server **54**") to store and retrieve the persistent objects managed by OMS **42**.

POS Server **54** interfaces with RDBMS **20** using a standard Structured Query Language (SQL) interface to physically store and retrieve the persistent objects managed by OMS **42**. RDBMS **20** interfaces with secondary memory storage devices **24** to physically store and retrieve tile objects created by application **14** and managed by OODB **18**.

The various modules of the present (DDL **80**, OMS **42**, name manager **46**, demon table **48**, object map **50**, PTR **44**, OTS **52**, and POS Server **54**) will now be considered in more detail.

Prior to developing an application system that interfaces with the present invention, the application developer must register the C++ classes to be used in the application system with the present invention. This is accomplished by executing the DDL translator described below.

The DDL Translator **80** (hereinafter referred to as "DDL **80**") is based on three separate application programs that are executed in sequence (see FIG. **3**). DDL **80** receives as input one or more data definition language source files **26** written by the developer of application **14**. These source files **26** contain class definitions in the C++ language plus additional keywords and syntactic constructs defined by the present invention (see Table 1 for an example of such a DDL source file). DDL **80** then extracts sufficient information from source files **26** to generate a C++ source file **40** (hereinafter referred to as "types **40**"; see Table 2 for an example of such a C++ source file) containing type description information for use by OMS **42** and OTS **52**, and a set of C++ source files **38** for every class definition in the source file(s). These files are then subsequently used by the application programmer when writing, compiling, and executing C++ programs that create and manipulate instances of the classes, as well as save or retrieve them using the present invention.

The first program in the sequence depicted in FIG. **3** is the commercially available C language preprocessor cpp **28**. This program receives as input source files **26** and produces a copy of the input file(s) with all cpp directives evaluated and expanded (expanded file **30**). Use of this program in the present invention does not require any modification.

TABLE 1

class ptrs : persistent	// to indicate independent persistence

TABLE 1-continued

```
{
private:
    int  len;                          // controls length of data member 'mpd'
  // Pointers to array of data types
  char   *[10] cpd;                    // constant length
  char   *[len] mpd;                   // length controlled by data member 'len'
  char   *[sentinel'  0'] spd;         // length terminated by hexadecimal zero
  boundary char *bpd;                  // referenced data is ignored when saved
  classPTR pers_ref;                   // reference To independent persistent object
public:
// Definition of Demon functions
demon create int setup();              // creation time demon
demon restore int resetup();           // fetch time demon
demon commit int ready ();             // commit time demon
demon abort   int cleanup 1);          //abort time demon
};
```

TABLE 2

```
char *Type-Descriptions[] = {  /* DDL RELEASE 0.1.5 */
/* 0-2: OCDB Classes */
"16|&|0|1|0|1|254|0|.|&|7|0|4|1|255|1|1|&|7|0|8|1|255|1|0|&",
"24|&|0|1|0|1|254|0|.|&|7|0|16|1|255|1|1|&|7|3|20|1|255|1|1|&~,
"8|&|7|3|0|1|.|.|2|&",
/* 3: User Class A */
"12|&|7|3|0|1|.|.|3|&",
/* 4: User Class B */
"8|&|7|3|0|1|.|.|4|&",
/* 5: User Class C */
"12|&|7|3|8|1|.|.|5|&",
};
Interpretation of Fields
"12|&|7|3|8|1|.|.|5|&'
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |Index into this array of referenced data
|  |  |  |  |  |  |Number of instances in the array
|  |  |  |  |  |How to determine number of instances in array
|  |  |  |  |Number of dimensions in array
|  |  |  |Offset of this pointer or PTR
|  |  |Type of pointer
|  |Data type of referenced data
|Size of an instance of this class
```

The second program in the sequence is the DDL processor ddl 32. This program receives as input expanded file 30 produced by the first program, scans expanded file 30 for specific information, and generates a typeobject database 34 for use by the third program, ddlpost 36. The ddl 32 program uses a lexical scanner based on the lexical analysis program generator lex (not shown) to scan source files 26 and return each token (syntactic unit not shown) to the main functions of ddl 32. In addition to scanning source files 26 for specific information, ddl 32 performs full syntactic and limited semantic checking on the input file, and generates appropriate error and warning messages.

ddl 32 processes each class definition in the input file as follows. First, it scans the input tokens until it recognizes the C++ keyword "class". If the "class" keyword is found, ddl 32 continues to scan for and then extract the class name, the name(s) of any classes from which this class is to be derived, and a keyword, "persistent", that indicates whether or not instances of this class should become "independent persistent objects" (IPO) when saved to the database. If this class is derived from another class and does not have the "persistent" keyword, ddl 32 determines if the latter class is also a persistent class. If so, this class is marked as an persistent class. If "persistent" is not present and if this class is not derived from a persistent class, instances of this class can still be saved to the database, but they will become "dependent persistent objects" (DPO) physically stored with an IPO. If the class(es) from which this class is to be derived has(have) not been previously processed by ddl 32 in this file or in previous executions of DDL 80, an error message is generated.

Second, ddl 32 scans the rest of the class definition for the declarations of data members and extracts the name of the data member, the name of the data type (fundamental C++ or user-defined type, including classes), and then determines the class's memory alignment, size, padding between it and the next data member, and its offset from the beginning of the class. ddl 32 also extracts information indicating if the data member is an array, the number of elements in the array, if the data member is a C++ pointer, the name of the data type of the referenced data, the number of data items referenced by the pointer (which can be specified by a sentinel, a decimal or hexadecimal value, or the name of a integer data member in this class), whether it has the keyword "boundary" before it (the referenced data will not be saved if an instance of this class is saved to the database), or if the data member is a PTR 44, the name of the referenced class. Unless the data types used for data members or referenced from this class (using C++ pointers or PTRs 44) have been processed by DDL 32 during the eighth step discussed below, an error message is generated.

Third, if ddl 32 determines that any of the data members are in the public portion of the class definition, an error message will be generated and the class definition will not be processed during the eighth step discussed below. Fourth, ddl 32 also scans for the existence of the

C++ keyword "virtual" before any function declaration inside the class definition, as this keyword affects the overall size of an instance of this class. Fifth, ddl 32 also scans for the existence of the keyword "demon" before any function declaration inside the class definition and notes this for later processing by ddlpost 36.

Sixth, once the entire class definition has been scanned, ddl 32 computes the size for instances of this class by adding the size of all classes from which this class derives (retrieved from the type description database) to the size of all data members in this class (if the data member is an embedded instance of a class, the size is retrieved from the type description database). If this class has virtual functions (as detected in the fourth step discussed above), the size of the class is increased by an amount equal to the size of a C++ pointer to a C++ virtual table pointer. If the keyword "persistent" is present (as detected in the first step discussed above), the size of the class is increased by an amount equal to the size of an instance of the PTR 44 class. The alignment and padding of this class are then calculated based on the overall size computed.

Seventh, these scanning and extraction steps continue until the end of the expanded file 30 is encountered.

Eighth, after all class definitions are scanned and the end of the input file is encountered, ddl 32 then generates and outputs the following information into type description database 34: the name of the class, the size, alignment, and padding information for the entire class, a string containing information on every C++ pointer and PTR included in the class, the names of all classes or structures that this class references (using C++ pointers or PTRs) or contains, the names of functions defined for the various demons supported by the present invention, and a copy of the entire class definition from source file 26.

The third program in the sequence is ddlpost 36 which receives as input type description 34 generated by ddl 32 and generates C++ source and header files source files 38 and types 40 as follows.

First, ddlpost 36 reads the contents of type description 34 and creates data structures to hold the information. Second, for every class in type description database 34, source files 38 is produced, which may be composed of two separate C++ source files.

If the file is a persistent class, a C++ header file is produced containing the definition of a new class derived from the PTR 44 class. This file is produced by editing a predefined template file (see Table 3) using a stream editor to convert certain character strings in the template file to the name of this class. A second C++ header file is produced that contains cpp 28 include directives for every class from which this class is derived, is used as the data type for a data member, or is referenced by a C++ pointer or PTR 44; the class definition; and C++ statements to allow the PTR 44 class functions to manipulate instances of this class. If this class has demons, this second C++ header file is also includes C++ statements to allow the Demon class functions to manipulate instances of this class. Similarly, if this class is persistent, this second C++ header file also contains definitions for the PTR 44 class functions generated in the above-described PTR 44 file.

TABLE 3

```
-#include "zg-ptr.h"
// The $PTR class
class $;
class $PTR: public ?
{
    public:
        /* methods */
        $& operator* ();
        $* operator→ ();
        $PTR& operator=(Zg-Eo&);
        $PTR& operator=($ *);
        $PTR& operator=($PTR &);
        operator $* ();
        void make-absent ();
        /* constructors & destructors */
        inline $PTR ()
        inline $PTR ($ *);
    private:
        #
        /* friends */
        friend $PTR& persist ($& object, zg-Uint sg=0);
};
```

The '$' are replaced with the name of the user defined class.
The '?' is replaced with the name(s) of class from which this class is derived.
The '#' is replaced with a C++ class definition if this class has defined Demons.

This second C++ source file is temporary and is produced by appending the first item to the file based on the information extracted in the first two steps of ddl 32. The second item is then appended to the file as obtained from type description database 34 (sixth item in type description database 34). The third item is appended to the file by generating a temporary file which resulted from editing a predefined template file (see Table 4) using a stream editor to convert certain character strings in the template file to the name of this class. The final two items are appended to the file in a similar manner.

TABLE 4

```
-inline $PTR::$PTR ()
{
    #
    eop→teop = eop→init-teop (1, @, 0);
}
inline $& $PTR::operator* ()
{
    return (*(($ *) (eop→rop ? eop→rop : fault ())));
}
inline $* $PTR::operator→()
{
    return (( $ *) (eop→rop ? eop→rop : fault ()));
}
inline $PTR& $PTR::operator= (Zg-Eo& rhs)
{
    return (($PTR&) this→assign-eo (rhs));
}
inline $PTR& $PTR::operator= ($PTR& rhs)
{
```

TABLE 4-continued

```
    return ((SPTR&) this→assign-ptr (rhs));
}
inline SPTR& SPTR::operator= (S* rhs)
{
    if (rhs == NULL) return ((SPTR&) this→assign-null ());
    else return ((SPTR&) this>assign-star (rhs, rhs→thisptr));
}
inline SPTR::operator S* ()
{
    return ((S *) (eop→rop?eop→rop:(eop→oid==NullOid?NULL:fault())));
}
inline void SPTR::make-absent ()
{
    if (!this→remove-object ()) delete (S *) this→eop→rop;
    this→eop→rop = NULL;
}
inline SPTR::SPTR (S* rhs)
{
    *this = rhs;
}
inline SPTR& persist (S& object, Zg-Uint sg)
{
    return ((SPTR&) object.thisptr.set-persist (&object, sg));
}
    The'S' are replaced with the name of the user defined class
    The '#' is replaced with a C++ statement if this class has
            defined Demons.
```

Third, after all classes in type description database **34** have been processed, C++ source file types **40** is generated containing C++ statements which define an array of character strings that contains the type descriptions of the classes processed, as described above in connection with the eighth step of ddl **32**. This file is compiled by the application developer and linked with the software modules of application **14** and OODB **18** to allow the array to be used by OMS **42** and OTS **52** during execution of application **14** interfacing with OODB **18**.

Although eight processing steps are discussed with regard to ddl **32**, to extract the class and type description information, the same effect could be achieved by combining steps.

The application program interfaces with one instance of OMS **42** as well as numerous instances of PTR **44** to create, manipulate, store, and retrieve persistent objects. Each independent persistent object created by the application is assigned an object ID **78**. Each object ID **78** is composed of a storage group number (indicates in which storage group the object is stored), an object number (indicates the specific object within the storage group), and a time stamp (indicates the time that object was saved to the database). Each of these fields is represented by a 32 bit unsigned integer value. A NULL object ID **78** is used to indicate the absence of a valid object ID. In addition, an encapsulation **62** (hereinafter referred to as "encapsulation **62**") is created and associated with each independent persistent object. Encapsulations **62** are described later during the description of PTRs **44**.

An instance of OMS **40** contains the following data members:

Architecture ID is a reference to information describing the architecture of the current computer hardware. This data member is added to the database when the database is originally initialized for use with the current computer hardware. This data member is used by OTS **52** when translating objects between their primary and secondary memory representations.

Default storage group contains the number of the storage group where objects will be stored by default. It can be set using Default-Storage-Group **108**.

Name manager is a reference to name manager **46** for this instance of OMS **40**. It can be set using Name-Manager **116**.

Demon table is a reference to demon table **48** for this instance of the OMS **40**. It is set during Create **100** and used during Create **300** (described later), Commit-Transaction **112**, Abort-Transaction **114**, and Fetch **120–124**.

Object map is a reference to object map **50** for this instance of OMS **40**. It is set during ZG-Create **100** and used by OMS **42**.

Transaction Id contains the current transaction number. It is used during Begin-Transaction **110**, Commit-Transaction **112**, and Abort-Transaction **114**.

POS Server is a reference to an instance of POS Server **54**. It is set during Create **100** and used by OMS **42** to store or retrieve objects in the database.

Active indicates whether or not this instance of OMS **40** is active; that is, Startup **102** has been called before a call to Shutdown **106** has been made.

There is also a global variable named Exists that indicates whether or not an instance of OMS **40** has been created by application **14**. This variable is checked during Startup **102**.

Table 5 lists interface functions **100–130** together with their function names and arguments, which OMS **42** provides to application **14**. A description of each function is given below. These functions either update the private state or return some information about the private state of an instance of OMS **42**. Interface functions **100–128** update the private state of the instance, while interface function **130** returns information about the private state of the instance to application programs. In most functions, error checking is performed after most individual actions. Error checking is not considered fundamental to the understanding of and operation of the present invention, and will therefore not be further described.

TABLE 5

	OMS Application Interface Functions	
	Name	Argument
100	Create	none
102	Startup	none
104	Delete	none
106	Shutdown	none
108	Default-Storage-Group	Storage Group
110	Begin-Transaction	none
112	Commit-Transaction	none
114	Abort-Transaction	none
116	Name-Manager	Name Manager
118	Default-Name-Context	Name Context
120	Fetch	Object Name, Lock, Time Context
122	Fetch	PTR, Lock, Time Context
124	Fetch	Object ID, Lock, Time Context
126	Lock-State	Object Name, Time Context
128	Lock-State	PTR, Time Context
130	Time	none

Create **100** creates an instance of OMS **42**. This function is called whenever an instance of OMS **42** is created statically, automatically, or dynamically, depending on the C++ variable declaration used in the application program. During creation, function Startup **102** is called.

Startup **102** can also be called at any time after Shutdown **106** is called. Since only one OMS **42** instance is allowed per application program, the value of the Exists global variable is checked to see if one already exists. If one does exist, an error value is returned. If one does not exist, the default storage group data member is set to the minimum storage group number if this is the first time an OMS **42** instance is being created. If the current OMS **42** instance had previously been created during this execution of application **14**, the previous value in the default storage group data member will be used. Demon table **48** and object map **50** are created and assigned to their associated data members. Next, an instance of POS Server **54** is created and assigned to the POS Server data member. The architecture ID object for the computer hardware within which application **14** and OODB **18** are executing is retrieved from POS Server **54** and assigned to the architecture ID data member. An instance of name manager **46** is created and assigned to the name manager data member. Begin-Transaction **110** is called, followed by a call to Default-Name-Context **116** to create the default name context in the database. If this is the first time this database has been used, a call is made to Commit-Transaction **112** to save the new default name context in the database. If not, Abort-Transaction **114** is called. Finally, the active data member is set to indicate that this instance of OMS **42** is active, the exists global variable is set to indicate that an instance of OMS **42** exists, and control is returned to the caller of this function.

Delete **104** deletes an instance of OMS **42**. This function is called whenever an OMS **42** instance is deleted statically, automatically, or dynamically, depending on the C++ variable declaration used in the application program. During deletion, function Shutdown **106** is called.

Shutdown **106** checks the value of the Active data member to determine whether this instance of OMS **42** is active. If it is not, control is returned immediately to the caller of this function, as no work must be performed. If this instance is active, for every encapsulation **62** currently in object map **50**, its modified data member is set to "Not Modified" and then encapsulation **62** is deleted. name manager **46**, object map **50**, and POS Server **54** are deleted and their associated data members are set to a null pointer, thereby indicating that the value is not valid. Finally, the transaction ID data member is to zero, the active data member is set to indicate that this instance of OMS **42** is inactive, and control is returned to the caller of this function.

Default-Storage-Group **108** updates the private state of this instance of OMS **42** as follows. If the supplied storage group number is greater than the minimum storage group number, Is-Sg-Valid **504** is called to insure that a storage group with that number exists in the database. If one does exist, the default storage group data member is assigned using the supplied storage group number and the previous value of this data member is returned. Otherwise, a value of 0 (zero) is returned.

Begin-Transaction **110** is used to mark the beginning boundary of a transaction. All interactions between the application program and OMS **42** must occur within a transaction. Begin-Transaction **516** is called to insure that POS Server **54** begins a transaction. If the call succeeds, the transaction ID data member is incremented by one and that value is returned. Otherwise, a value of −1 is returned.

Commit-Transaction **112** is used to mark the ending boundary of a transaction. If the transaction ID data member indicates that there is not a current transaction, a value of −1 is returned. Otherwise, an instance of POS Encapsulation **70** (herein referred to as "POS encapsulation **70**"; discussed below) is created if one does not already exist. Next, every encapsulation **62** in object map **50** is examined to determine how many do not have an object ID **78** (those objects which have been created by the application program since the last call to Commit-Transaction). If the number is greater than zero, Alloc-Symbolic-Name **506** is called. If the requested number of object IDs **78** could not obtained, an error is returned. If the call was successful, Begin-Commit **508** is called, which returns a timestamp for this commit (which will be saved in each POS encapsulation **70** during translation), or a error value. For every encapsulation **62** currently in object map **50**, the following actions take place. If the modified data member in encapsulation **62** indicates that the object has not been modified, it is bypassed for processing. Otherwise, if the concurrent lock data member in the encapsulation **62** indicates that the application program does not have a WRITE lock on the object, an error is returned since the application program must have a WRITE lock on all objects in order to create new versions of them in the database. Otherwise, if a Commit Demon for the object's type was registered in de,non table **48**, that Commit Demon is called (see description of demon table **48**). Next, OTS Internal2External is called and passed POS encapsulation **70**, which it updates with an external representation **402** of the object. If the translation succeeds, Put-Object **514** is called to create a new version of the object in the database. After every encapsulation **62** in object map **50** has been processed in this manner, End-Commit **510** is called to commit the changes to the database, followed by a call to End-Transaction **510** to end POS Server's **54** transaction. Since the transaction has completed, other application programs may retrieve the objects from the database, update them, and create new versions of them. Versions accessible through this instance of OMS **42** are then considered invalid. Thus, for each encapsulation **62** currently in Object Map **50**,

the following actions take place. The concurrent lock data member is set to "Invalid" and the modified data member is set to "Not Modified". If the number of references to this encapsulation 62 from the application program is greater than zero, the object is deleted if OMS 42 created it. Otherwise, the object pointer data member in encapsulation 62 is set to null. If the number of references is zero, encapsulation 62 is deleted, which may delete the object if OMS 42 created it. Finally, the transaction ID data member is decremented by one and the number of objects committed to the database is returned.

Abort-Transaction 114 is also used to mark the ending boundary of a transaction but does not commit any objects to the database. If the transaction ID data member indicates there is not a current transaction, a value of −1 is returned. Otherwise, for every encapsulation 62 currently in object map 50, if an Abort Demon for the object's type was registered in demon table 48, that Abort Demon is called. Next, the concurrent lock data member is set to "Invalid" and the modified data member is set to "Not Modified". Finally, Abort-Transaction 520 is called to allow POS Server 54 to terminate its current transaction, the transaction ID data member is decremented by one, and control is returned to the caller of this function.

Name-Manager 116 simply assigns the name manager data member to the supplied instance of name manager 46 and returns the previous value of this data member.

Default-Name-Context 118 calls Name-Context 204, to ensure that the supplied name context exists, and returns.

Fetch 120–124 has three forms. The first takes an object name, the second takes a PTR 44 associated with an object, and the third takes an object ID 78 of an object. The first form calls Return-OID 210 which returns object ID 78 of the supplied object name, or a null object ID 78 if the user-defined name had not been registered, in which case a null address is returned. The second form retrieves object ID 78 from encapsulation 62 associated with supplied PTR 44. In any case, the following actions occur once an object ID 78 has been obtained. Encapsulation 62 associated with object ID 78 is retrieved from object map 50 and the object pointer data member is checked to determine if the associated object is or is not in primary memory. If the value indicates that it is in primary memory, the address of the object in primary memory is returned. If the value indicates that it is not in primary memory, Get-Object 512 is called with the supplied lock and time context (which default to READ and MOST-RECENT, respectively) and returns a POS encapsulation 70 (which contains the object's external representation 402 and its references to other persistent objects). Next, OTS External2Internal is called with POS encapsulation 70 to create a primary memory representation of the object using the object's external representation 402 in POS encapsulation 70. The references to other persistent objects are registered with OMS if they have not already been registered. Finally, the modified and deleteable data members of encapsulation 62 are set to "False" and "True", respectively (since OMS 42 just created the object), and the address of the object in primary memory is returned.

Lock-State 126–128 has two forms similar to Fetch 120–124. The first takes an object name, while the second takes a PTR 44 associated with an object. As in Fetch 120–124, the first form calls Return-OID 210 to return object ID 78 of the supplied object name or a null object ID 78 if the user-defined name had not been registered, in which case a value indicating that application 14 does not currently have a valid lock on the object is returned. The second form retrieves object ID 78 from encapsulation 62 associated with the supplied PTR. In any case, encapsulation 62 associated with object ID 78 is retrieved from object map 50 and the concurrent lock data member is checked to determine if the application program currently has a valid lock on the object. If so, the value of the concurrent lock data member is returned. Otherwise, a value indicating that the application does not currently have a valid lock on the object is returned.

Time 130 returns the current time received from OS 16.

OMS 42 interacts with name manager 46 to manage the association of names supplied by the application with independent persistent objects created or retrieved by application 14 using PTRs 44.

An instance of Name Manager 46 contains the following data members:

First context references the first name context in a list of all name contexts defined in the database currently being accessed by application 14 using the present invention. It is set during Create 200 and accessed when searching for name contexts or name entries.

Current context references the current name context in a list of all name contexts defined in the database currently being accessed by application 14 using the present invention. It is set during Create 200 to the first context and during Name-Context 204 to the new default name context. It is accessed when searching for name entries.

Table 6 lists interface functions 200–210, with their function names and arguments, which name manager 46 provides to OMS 42. A description of each function is given below. These functions either update or return some information about the private state of an instance of the name manager class. Interface functions 200–208 update the private state of the instance, while interface function 210 returns information about the private state of the instance to application programs.

TABLE 6

Name Manager Interface Functions

	Name	Argument
200	Create	none
202	Delete	none
204	Name-Context	Context Name
206	Name	Object Name, Object Oid
208	Unname	Object Name, Object Id
210	Return-OID	Object Name

Create 200 creates an instance of a name manager 46 as follows. Begin-Transaction 110 is called and then Fetch 124 is called to retrieve the default root context. If the default root context does not exist, one is created, assigned to the first context data member, a new object ID 78 obtained by indirectly calling Alloc-Symbolic-Name 506, and Commit-Transaction 112 is called to save the default root context in the database. If tile default root context already exists, Abort-Transaction 114 is called. Finally, the current context data member is set to the new or previously existing default root context object and control is returned to the caller of this function.

Delete 202 simply deletes the supplied instance of name manager 46.

Name-Context 204 searches the list of contexts, accessible from the first context data member, for an context entry which has the supplied context name. If the context name is in an entry, a 0 (zero) is returned. If it is not in any entry, a new context entry is created using the supplied context name. It is added to the front of the list of contexts, the current context data member is assigned to the new context entry, and a 1 (one) is returned.

Name 206 accesses the current context data member's list of name entries and searches for a name entry which has the supplied object name. If the object name is in an entry, a −1 is returned since no duplicate object names are allowed. If it is not in any entry, a new name entry is created using the supplied object name and object ID 78 and added to the end of the list. A 0 (zero) is returned.

Unname 208 accesses the current Context data member's list of name entries and searches for a name entry which has the supplied object name. If the object name is not in an entry, a −1 is returned. If it is in an entry and the object ID 78 in the entry matches supplied object ID 78, the name entry is deleted from the list and a 0 (zero) is returned. If the two object ID 78s did not match, a −1 is returned.

Return-object ID 210 accesses the current context data member's list of name entries and searches for a name entry which has the supplied object Name. If the object name is not in an entry, a null object ID 78 is returned. If it is in an entry, object ID 78 in the name entry is returned.

FIG. 4 is a block diagram of demon table 48 used by OMS 42 during certain OMS 42 functions. Demon table 48 is an array of entries that contains two data members: an object ID 78 for a user-defined type or class, and a reference to demon object 74.

During the DDL translation process, the application developer can add the keyword "demon" and an event keyword before a class function name (see Table 4 above, bottom portion of the class declaration). Classes with these demon keywords are referred to as "demon classes". The existence of these keywords is recognized by DDL and the names of the associated functions are added to type description database 34. During execution of ddlpost 36, additional C++ statements and functions are added to source files 38 for each class annotated with the demon keywords. First, an additional data member, PTR Demon 76, is defined that will be shared by all instances of this class' PTRs 44 during execution of an application program using this class. Second, another class, demon object 74, is defined and includes functions which will call the user-defined functions named in the original class definition.

Prior to execution of application 14, the application developer compiles source files 38 and links them with the software modules of application 14 and OODB 18. After execution of application 14 is started, code required to statically create instances of PTRs 76 is executed (this is the code generated in source files 38). During execution of application 14, whenever an instance of OMS 42, an instance of demon table 48 is initialized. Whenever an instance of a demon class PTR 44 is created by the application program, the value of its associated PTR Demon 76 is checked to see if an instance of demon object 74 exists. If one does exist, processing continues. If one does not exist, one is created and PTR Demon 76 is set to reference new demon object 74. When demon object 74 is created, an entry is added to demon table 48. Object ID 78 data member is set to object ID 78 of the type description object (in Types 40) for this demon class, while the demon object reference data member is set to the newly created instance of demon object 74.

While processing persistent objects during certain OMS 42 functions, OMS 42 checks demon table 48 to see if a demon object 74 for the type of the persistent object has been registered. It accomplishes this by comparing object ID 78 of the type description for the given object and object IDs 78 in demon table 48. If there is not a match, processing of the given object continues. However, if there is a match, OMS 42 will call one of the demon functions as specified in the demon object 74 referenced from the appropriate demon table 48 entry. The demon function can then manipulate the object prior to OMS 42 continuing with its processing.

FIG. 5 is a block diagram of object map 50 used by OMS 42. Object map 50 manages encapsulations 62 created when an independent persistent object is created by application 14 or retrieved from the database. Object map 50 is constructed as a two-level index based on storage group and object number. The first level is array whose elements are records containing a storage group number and a pointer to a storage group hash table 58. The second level is storage group hash table 58 whose entries 60 contain pointers to encapsulations 62 and the next entry 60 in hash table 58. The interface provided by object map 26 to OMS 42 is as follows:

Add Encapsulation locates the appropriate hash table 56 (creating a new one if necessary), creates new entry 60, associates it with supplied encapsulation 62, and adds it to storage group hash table 58.

Find Encapsulation locates the appropriate hash table object map 58 and searches hash table 58's entries 60 for one that references encapsulation 62 which has the same object ID 78 as the supplied object ID 78. If an entry 60 is found, the address of the associated encapsulation 62 is returned. If an entry is not found, or there is not a hash table 58 for the supplied storage group, a null address is returned.

Remove Encapsulation locates the appropriate hash table 58 and searches hash table 58's entries 60 for one that references encapsulation 62 which has the same object ID 78 as the supplied object ID 78. If an entry 60 is found, entry 60 is removed from storage group hash table 58 and deleted. No action occurs if an entry 60 is not found.

As described earlier, application 14 creates instances of PTR 44 to create, manipulate, store, and retrieve persistent objects using the present invention. As can been seen in FIG. 6, an instance of a PTR 44 contains only one data member, encapsulation, which references an instance of the encapsulation class (encapsulation 62).

An instance of encapsulation 62 contains the following data members.

Type (class) description ID references a limited description of the definition of the C++ class of which this object is an instance. Specifically, it is a C++ pointer to a character string that contains information on every C++ pointer and PTR 44 in an instance of the class type of the object referenced via the Object Pointer data member. It is extracted from types 40.

Number of references to encapsulation 62 is incremented by 1 (one) every time a PTR 44 is created that references encapsulation 62, and decremented by 1

(one) every time a PTR 44 that references encapsulation 62 is deleted.

Concurrent lock indicates the current lock held on the object associated with the encapsulation. It is set when encapsulation 62 is created, when an object is retrieved from the database, or when a lock is upgraded.

Time stamp indicates the time that the associated object was saved to the database.

Object identifier is object ID 78 of the object associated with this PTR 44.

Object pointer is a C++ pointer that references the primary memory representation of the associated object. Unless the value is null (thereby indicating that the value is not valied), the associated object is resident in primary memory.

Persistent indicates whether this persistent object should be saved or already has been saved to the database. It is set by calling Persist 304.

Modified indicates whether this persistent object has been modified by application 14. Newly-created persistent objects are marked as "Modified", while persistent objects retrieved from the database are marked as "Not Modified". Persistent objects can be marked as "Not Modified" by calling Set-Modified 310.

Deleteable indicates whether OMS 42 can delete this persistent object from primary memory. If application 14 created this object, OMS 42 cannot delete it. If OMS 42 created the object (when retrieving it from the database), it can delete the persistent object.

Therefore, it can be seen in FIG. 6 that given a PTR 44, one can access its associated encapsulation 62, and from there, access the associated object 64.

Table 7 lists interface functions 300–326 together with their function names and arguments, which PTR 44 provides to application 14. A description of each function is given below. These functions either update the private state or return some information about the private state of PTR 44. Interface functions 300–310, 316, and 320–322 update the private state of PTR 44, while Interface functions 312, 314, 318, 324, and 326 return information about the private state of PTR 44 to application 14.

TABLE 7

PTR Application Interface Functions

	Name	Argument
300	Create	none
302	Delete	none
304	Persist	Object, Storage Group
306	Name	Object Name
308	Unname	Object Name
310	Set-Modified	none
312	Is-Modified	none
314	Lock-State	none
316	Upgrade-Lock	New Lock
318	In-Memory	none
320	Make-Absent	none
322	Make-Present	none
324	Timecontext	none
326	Timestamp	none

Create 300 creates an instance of PTR 44 for application 14 to reference objects. Instances are created statically, automatically, or dynamically, depending on the C++ variable declaration used in application 14. First, an encapsulation 62 is created and assigned to the encapsulation data member. Next, encapsulation 62 for the type description object (in types 40) for this PTR's 44 class definition is obtained from object map 50, assigned to the associated encapsulation's 62 type description ID data member, and control is returned to the caller of this function.

Delete 302 deletes an instance of PTR 44 class. If there are not any references to encapsulation 62 by any other PTRs 44, encapsulation 62 is removed from object map 50. If the deleteable data member of encapsulation 62 indicates that associated object 64 was created by OMS 42, object 64 is deleted. Finally, encapsulation 62 is deleted and control is returned to the caller of this function.

Persist 304 is used to indicate that supplied object 64 should be saved to the database when the next Commit-Transaction 112 is performed. First, the persistent data member of encapsulation 62 is set to "True". Then, POS Alloc-Symbolic-Name 506 is called to obtain object ID 78 for the object, using the supplied storage group number or the value of the default storage group data member of OMS 42, if one was not supplied. Finally, if a Persist Demon for the object's type was registered in demon table 48, that persist demon is called, and control is returned to the caller of this function.

Name 306 associates the supplied object name with object 64 with PTR 44. Since names may only be associated with independent persistent objects, if the object ID data member of associated encapsulation 62 is null object ID 78, POS Alloc-Symbolic-Name 506 is called to obtain object ID 78 for object 64. Next, Name 206 is called to create the new name and control is returned to the caller of this function.

Unname 308 dissociates the supplied object name from object 64 associated with PTR 44 by calling Unname 208, and returns control to the caller of this function.

Set-Modified 310 shows that the application program has modified object 64 (this is checked during Zeitgeist Commit-Transaction 112 processing). First, the modified data member of encapsulation 62 is set to "True", Upgrade-Lock 212 is called requesting a WRITE lock, and then control is returned to the caller of this function.

Is-Modified 312 retrieves the value of the modified data member of encapsulation 62 and returns that value.

Lock-State 314 returns either the current value of the concurrent lock data member of encapsulation 62 or "Invalid" if the data member indicates that the current lock is not valid.

Upgrade-Lock 316 attempts to upgrade an existing lock on object 64 to the requested lock. If the supplied lock is not a valid lock type, an error is returned. If the supplied lock is a READ-ONLY lock, the concurrent lock data member of encapsulation 62 is set to that value. If the current lock is valid and stronger than or equal to the supplied lock, control is returned to the caller of this function. If neither of these conditions apply, Set-Lock 524 is called requesting the supplied lock. If the lock was set, the concurrent lock data member of encapsulation 62 is set to the supplied lock and control is returned to the caller of this function.

In-Memory 318 checks the value of the object pointer data member of encapsulation 62 and returns a 1 (one) if the value indicates associated object 64 is in primary memory or 0 (zero) otherwise.

Make-Absent 320 updates encapsulation 62 such that subsequent references to object 64 using PTR 44 will cause OMS 42 to retrieve object 64 from POS Server 54. First, if the deleteable data member of encapsulation 62 indicates that OMS 42 allocated object 64, the delete-

able data member of encapsulation 62 is set to indicate that OMS 42 did not allocate object 64 and the modified data members is set to "Not Modified". Next, object 64 and any dependent persistent objects it references are deleted. Regardless whether OMS 42 allocated object 64, the object pointer data member of encapsulation 62 is set to null to indicate that object 64 is not resident in primary memory, and control is returned to the caller of this function.

Make-Present 322 makes object 64 accessible through PTR 44. First, the object pointer data member of encapsulation 62 is checked to see if object 64 is already in primary memory. If so, control is returned to the caller of this function. Otherwise, Fetch 124 is called to retrieve object 64 object from POS Server 54 and install it in primary memory. If that is successful, the primary memory address of object 64 is returned.

Timecontext 324 retrieves and returns the value of the time context data member of object ID 78 of encapsulation 62. This value indicates the time context used when retrieving the object from the database.

Timestamp 326 retrieves and returns the value of the time stamp data member of encapsulation 62. This value indicates the time object 64 was saved to the database.

OMS 42 provides an additional capability to allow an application program to have independent persistent objects implicitly and automatically retrieved from POS Server 54 without having to call Fetch 120–124 or Make-Present 322. This capability is called object faulting and PTR 44 processes an occurrence of an object fault as follows. When an application program dereferences an instance of PTR 44, code is executed to check the value of the object pointer data member of associated encapsulation 62. If the value indicates that associated object 64 is already in primary memory, the application program continues. If the value indicates that associated object 64 is not in primary memory, Fetch 124 is called using object ID 78 data member in encapsulation 62. After object 64 has been installed in primary memory by Fetch 124, the application program continues. In the C++ embodiment, the code necessary to perform this processing is automatically generated by DDL 80 (included in the C++ files of source files 38). Specifically, a definition is provided to overload the C++ dereference operators ("a func()" and "(*a).-func()") for independent persistent object classes.

FIG. 7*a* shows primary memory representation 400 (hereinafter referred to as "object 400") of a C++ object. Although a C++ object can contain various data members, the present invention supports the following three categories of data types for the data members. The first category comprises those data types which can be fully embedded in the object (data members 410–414). The second category comprises those data types which are a reference to an independent or dependent persistent object using a C++ pointer (data members 416–420). The third category comprises those data types which are a reference to an independent persistent object using a PTR 44 (data members 406,408, and 422). FIG. 7*b* shows the secondary memory representation of a C++ object, which is composed of two items, external representation 402 and external references 404.

OTS 52 provides two interface functions to OMS 42 to translate persistent objects between their primary and secondary memory representations.

Internal2External is the first function and translates an object from its primary (or internal) to its secondary (or external) memory representation. It accomplishes the translation by first allocating two memory buffers to hold external representation 402 and external references 404. Next, information about the object (size, type ID) and each reference in the object (type of pointer, type of referenced object, how to determine the number of objects referenced, etc.) is obtained from types 40, as generated by DDL 80. Second, the data in object 400 is copied to external representation 402, starting at the beginning of external representation 402.

Then, for every reference, the following actions are performed. If the reference is a C++ pointer and it is not a null value, information on the referenced object is obtained from types 40. The referenced object is copied to external representation 402 after any other copied data and the offset of the copied referenced object (relative to the beginning of external representation 402) is stored at the location in external representation 402 of the original C++ pointer. For example, data member 418 references another object. After copying, data member 436 contains the offset from the beginning of external representation 402 of the copy of that object, namely, object 444. If the data type of the referenced object is a C++ structure or class, a similar translation process is performed on the referenced object. If the C++ pointer is a "boundary" pointer, the value in external representation 402 is set to 0 (zero). For example, if data member 416 were a boundary pointer, data member 434 would contain a 0 (zero). If the reference is a PTR 44, object ID 78 of the referenced object is obtained from associated encapsulation 62 and copied to external references 404 after any other copied object IDs 78, and the offset of that copy (relative to the beginning of external references 404) is stored in the appropriate location in external representation 402. For example, data member 422 is a PTR 44. Object ID 78 of the referenced object is stored in external reference 452; the location of external reference 452 (in external references 404) is stored in data member 440. Essentially, referenced objects or object IDs 78 are copied to a location and that location is stored where the original reference was copied. Finally, POS encapsulation 70 is updated (with this object's object ID 78, the architecture ID from the current instance of OMS 42, the type description ID associated with this object, and newly-created external representation 402 and external references 404) and returned to the caller of this function.

External2Internal is the second function and translates an object from its secondary (or external) to its primary (or internal) memory representation. It accomplishes the translation by first checking the architecture ID in supplied POS encapsulation 70 to see that it matches the architecture ID data member in the current instance of OMS 42. In the current embodiment, if they do not match, the object cannot be translated and an error is returned to the caller of this function. Second, information about the object is obtained as is done in Internal2External above. Third, sufficient primary memory to hold the object is allocated and the object is copied from external representation 402 to the newly allocated memory (for example, object 400).

Next, the following actions are performed for every reference. If the reference is a C++ pointer, information on the referenced object is obtained from types 40. Sufficient primary memory is allocated to hold the referenced object and the referenced object is copied from the location in external representation 402 as indicated by the value of the original reference. The corre-

sponding reference in object **400** is updated to reference the newly allocated referenced object. For example, data member **436** is a C++ pointer whose referenced object is stored at object **444**. After memory is allocated to hold the referenced object, the referenced object is copied from object **444** to the newly-allocated memory and data member **418** is updated to reference the newly allocated referenced object. If the data type of the referenced object is a C++ structure or class, a similar translation process is performed on the referenced object. If the reference is PTR **44**, the following actions are performed. If object ID **78** of the referenced object is not null, the timestamp in object ID **78** is set to the timestamp of this object's encapsulation **62**. This insures that the referencing object's timestamp will be used when the referenced object is retrieved from POS Server **54**. Next, a reference to encapsulation **62** for the referenced object is obtained from object map **50** (one will be created if the object is not currently known to OMS **42**). If application **14** does not currently hold a valid lock on the referenced object (determined by examining encapsulation **62** just obtained), the value of the concurrent lock data member of referenced object's encapsulation **62** is set to the value of the concurrent lock data member of this object's encapsulation **62**. This insures that the referencing object's lock will be used when the referenced object is retrieved from POS Server **54**. If application **14** holds a valid lock on the referenced object, Upgrade-Lock **316** Lock is called requesting a lock equal to the value of the concurrent lock data member in this object's encapsulation **62**. Lastly, once all the references in the object have been processed, control is returned to the caller of this function.

POS Server **54** is used by OMS **42** to store and retrieve persistent objects in the database. In the C++ embodiment, the present invention uses a commercially available RDBMS **20** to store external representation **402** and external references **404** of an independent persistent object described above. The present invention interacts with RDBMS **20** using the embedded Standard Query Language (SQL) interface provided by the vendor. This allows the present invention to replace one vendor's RDBMS with another vendor's RDBMS with insignificant modifications to the present invention.

In order to store the persistent objects created by application **14** and managed by OMS **42**, the following relational tables are defined in RDBMS **20**.

The first table is the groups table which contains two attributes, storage group and object number. The purpose of the table is to control the allocation of object identifiers (object IDs **78**) within storage groups. See the description of Alloc-Symbolic-Name below for details on how the object numbers are allocated.

The second table is the value table with the attributes shown below in Table 8. The purpose of this table is to hold sufficient information about an independent persistent object in order to identify it by its object ID **78** (composed of the first three attributes), identify the architecture of the computer hardware in which application **14** and OODB were executing when the object was saved, identify the object's type (class) description, identify the number of independent persistent objects it references, recreate the object, and install it in primary memory. If external representation **402** of the object is longer than the length allowed for attribute values by RDBMS **20**, there are multiple tuples in this table for the single large object, with all values the same except for the "sequence number" (which begins with one and is incremented by one for every additional tuple) and the "external representation" (which continues where the previous tuple left off).

TABLE 8

Attributes in Groups Table
Storage Group;
Object Number;
Commit time;
Sequence number;
Object size;
Architecture Object Storage Group;
Architecture Object Number;
Architecture Object Commit time;
Type Descriptor Storage group;
Type Descriptor Object Number;
Type Descriptor Commit time;
Number of user defined attributes associated with the object;
Number of system defined attributes associated with the object;
Number of references to other persistent objects, including this object; and
External representation of object.

The last table is refto table with the attributes shown below in Table 9. The purpose of this table is to hole the references from one independent persistent object to other independent persistent objects. Each tuple set (one or more tuples with the same object ID) in the value table is associated with one or more tuples in this table by virtue of the storage group, object number, and commit time being the same as the associated value table tuple. If there are multiple references from an object, there are multiple tuples in this table with the values for the storage group, object number, and commit time attributes in the associated value tuple set, except for the "sequence number" (which begins with one and is incremented by one for every additional tuple). The number of tuples in this table associated with a tuple set in the value table equals the value of the "number of references" attribute in the associated value table tuple set.

TABLE 9

Attributes in Refto Table
Storage Group;
Object Number;
Commit time;
Sequence number;
Referenced Object Storage Group;
Referenced Object Object Number; and
Referenced Object Commit time.

Tables 10 and 11 show an example of how the object seen in FIGS. *7a* and *7b* might appear stored in the value and refto tables, respectively.

TABLE 10

VALUE Table Tuples						
Storage Group	Object Number	Commit Time	Sequence Number	Object Size	Other.. Attributes	External Representation
5	1438	654318	1	83468	—	[array of bytes]
5	1438	654318	2	83468	—	[array of bytes]
5	1438	654318	3	83468	—	[array of bytes]

TABLE 11

REFTO Table Tuples						
Storage Group	Object Number	Commit Time	Sequence Number	Referenced Storage Group	Referenced Object Name	Referenced Commit Time
5	1438	654318	1	5	1438	654318
5	1438	654318	2	8	3481	654318
5	1438	654318	3	12	3347	654318

OTS 52 and POS Server 54 pass between each other encapsulations 62 and POS encapsulations 70. An instance of POS encapsulation 70 contains the following data members.

Object ID is object ID 78 of the object being passed in this POS encapsulation 70.

Architecture ID is object ID 78 of a persistent object describing the architecture of the computer hardware in which application 14 and OODB 18 are currently executing.

Type description ID is object ID 78 of a persistent object that describes the primary memory representation of the object being passed in this POS encapsulation 70.

Size of object external representation is the number of bytes that object external representation 402 in this POS encapsulation 70 contains.

Object external representation is the external representation 402 of the object being passed in this POS encapsulation 70 contains.

Number of external references is the number of external references of the object being passed in this POS encapsulation 70.

External references is the external references 404 of the object being passed in this POS encapsulation 70 contains.

A instance of POS Server 54 contains the following data members.

Commit in progress records whether a commit operation is currently in progress. A commit starts when Begin-Commit 508 is called and ends when End-Commit 510 is called.

Transaction is a reference to a transaction machine (not shown) which monitors transactions being performed by multiple programs as they access the database.

Table 12 lists interface functions 500–524 together their function names and arguments, which POS Server 54 provides to OMS 42. A description of each function is given below. These functions either update the private state or return some information about the private state of an instance of POS Server 54. Interface functions 500–502, 506–510, and 514–524 update the private state of the instance, while interface function 504 and 512 returns information about the private state of the instance to OMS 42.

TABLE 12

POS Server Interface Functions		
	Name	Argument
500	Create	none
502	Delete	none
504	Is-Sg-Valid	Storage Group
506	Alloc-Symbolic-Name	Storage Group, Number Requested
508	Begin-Commit	none
510	End-Commit	none
512	Get-Object	Encapsulation
514	Put-Object	POS Encapsulation
516	Begin-Transaction	none

TABLE 12-continued

POS Server Interface Functions		
	Name	Argument
518	End-Transaction	none
520	Abort-Transaction	none
522	Set-Lock	Encapsulation, New Lock, Wait?
524	Set-Lock	Encapsulation, New Lock

Create 500 creates an instance of POS Server 54 and connects to RDBMS 20 using the appropriate SQL statements (this allows further calls from POS Server 54 to RDBMS 20). Next an instance of the transaction machine is created and assigned to the transaction data member. Finally, control is returned to the caller of this function.

Delete 502 deletes an instance of a POS Server 54, calls Abort-Transaction 520 in case End-Commit 510 had not been called by OMS 42. Next, a disconnect from RDBMS 20 is performed using the appropriate SQL statements, making certain that any uncommitted changes previously made by OMS 42 are rolled back or deleted. In addition, this signals the end of calls from POS Server 54 to RDBMS 20. Finally, control is returned to the caller of this function.

Is-Sg-Valid 504 issues a SQL query to RDBMS 20 to determine if a tuple exists in the groups table with the supplied storage group. If a tuple exists, a 1 (one) is returned, otherwise a 0 (zero) is returned.

Alloc-Symbolic-Name 506 issues a SQL query to RDBMS 20 to retrieve the tuple in the groups table with the supplied storage group and makes a copy of the value of the object number attribute from the returned tuple. The object number attribute is incremented by the number requested and an SQL query is issued to update the modified tuple and commit the update in RDBMS 20. Finally, the copied value of the object number attribute is returned.

Begin-Commit 508 is used to record the beginning of a commit. If the value of the commit in progress data member indicates that a commit is in progress, an error is returned since only one commit can be in progress at any time. Otherwise, the current time from OS 16 is obtained, the value of the commit in progress data member is set to indicate that a commit is in progress, and the time obtained from OS 16 is returned.

End-Commit 510 is used to record the end of a commit. If the value of the commit in progress data member indicates that a commit is not in progress, an error is returned. Otherwise, an SQL query is issued to commit all pending changes previously sent to RDBMS 20 by POS Server 54. If that query fails, another SQL query is issued to rollback any pending changes to insure that none of the pending changes are seen by any other application 14 which may access this database. Finally, the value of the commit in progress data member is set to indicate that a commit is not in progress, and control is returned to the caller of this function.

Get-Object 512 begins by calling Is-Sg-Valid 504 to insure that the storage group in object ID 78 of supplied encapsulation 62 exists. If it does not exist, an error is returned. Otherwise, an SQL query is issued to RDBMS 20 to retrieve the first tuple in the value table which matches object ID 78 of supplied encapsulation 62. Next, using the value of the object size attribute in the retrieved tuple, a memory buffer sufficient to hold the entire object is allocated. If the object size attribute indicates that there are additional tuples with the same

object ID (because the retrieved external representation 402 was too large to fit in one tuple), additional SQL queries are issued to retrieve the remaining value tuples. The portions of external representation 402 from the tuples retrieved are copied into the memory buffer. Next, an SQL query is issued to RDBMS 20 requesting the first tuple in the refto table which matches object ID 78 of supplied encapsulation 62. If the "number of references" attribute in the value tuple indicates that there are additional tuples with the same object ID, additional SQL queries are issued to retrieve the remaining refto tuples. The values of the referenced object storage group, object number, and commit time from these tuples are used to create an external references 404. Next, an SQL query is issued to RDBMS 20 to commit any pending work to insure that any RDBMs locks on any of the tuples retrieved are not further retained. Next, a POS encapsulation 70 is created and updated with a copy of object ID 78 from supplied encapsulation 62, the architecture ID, type description ID, size of external representation 402 and external representation 402 collected from the value tuple(s), and the number of external references and external references 404 collected from the refto tuple(s). Finally, this newly-created POS encapsulation 70 is returned to the caller of this function.

Put-Object 514 begins by checking the value of the commit in progress data member to insure that a commit is in progress. If one is not in progress, an error is returned. Otherwise, the length of external representation 402 in supplied POS encapsulation 70 is used to calculate how many value tuples will be needed to store external representation 402. An SQL query is issued to insert sufficient value tuples to store external representation 402, using object ID 78, architecture ID and type description ID data members in supplied POS encapsulation 70 for the other attribute values in the new value tuples (see Table 10). Next, the number of external references data member in supplied POS encapsulation 70 is used to determine how many refto tuples will be needed to store external references 404. An SQL query is issued to insert sufficient refto tuples to store external references 404, using object ID 78 in supplied POS encapsulation 70 for the other attribute values in the new refto tuples (see Table 11). Finally, control is returned to the caller of this function.

Begin-Transaction 516 is used to mark the beginning of a transaction started by application 14. If the transaction data member indicates that a transaction is already in progress, an error is returned. Otherwise, a new transaction ID is obtained from the transaction machine, and control is returned to the caller of this function.

End-Transaction 518 is used to mark the end of a transaction started by application 14. If the transaction data member indicates that a transaction is not currently in progress, an error is returned. Otherwise, the transaction machine is called to end the current transaction and control is returned to the caller of this function.

Abort-Transaction 520 is also used to mark the end of a transaction started by application 14. If the transaction data member indicates that a transaction is not currently in progress, an error is returned. Otherwise, the transaction machine is called to end the current transaction and control is returned to the caller of this function.

Set-Lock 522–524 has two forms which attempt to set a lock on the object associated with supplied encapsulation 62. The first form will wait until the lock has been granted while the second will return if the lock cannot be granted on the first attempt. If object ID 78 of supplied encapsulation 62 is a null object ID, control is returned to the caller of this function since the object does not yet exist in the the database and is implicitly WRITE locked by application 14. If the request is for a READ-ONLY lock, the concurrent lock data member in supplied encapsulation 62 is set to that value and control is returned to the caller of this function. If the request is for a READ or a WRITE Lock, the transaction machine is called to obtain the lock. If the lock could not be granted due to an error, an error is returned. If the lock were granted, the concurrent lock data member in the supplied encapsulation 62 is set to that value and control is returned to the caller of this function.

Application 14 interface with OMS 20 and PTR 44 by embedding function calls to OMS 20 and PTR 44 as well as use instances of PTR 44 in programming language statements in the application software. In the preferred embodiment of the present invention, OMS 42 consists of one library of software and one C++ header file corresponding to OMS 42. The application developer includes the OMS 42 header file along with source files 38 generated by DDL 80 into the application software during compilation. Types 40 is also compiled by the application developer. The library and object files produced during the compilation of the C++ source files are then linked to form an application load module. In the preferred embodiment of the present invention, OODB 18 and application 14 using OODB 18 execute in the same address space, while RDBMS 20 executes in a different address space.

If the function calls to OODB 18 are extracted from application 14 software, the resulting set of instructions would have the following basic control flow.

First, an instance of OMS 42 would be created to begin the interface with OMS 42.

Second, one or more instances of PTR 44 would be created to allow application to create and manipulate as well as store and/or retrieve persistent objects using the present invention.

Third, application 14 would call OMS Begin-Transaction 110.

Fourth, OMS Default-Storage-Group 108 would be called to define a new default storage group, if so desired by application 14, in which newly-created objects would be stored.

Fifth, OMS Default-Name-Context 118 would be called to define a new default name context, if so desired by application 14, in which new object names would be registered.

Sixth, application 14 would create instances of objects and manipulate them using functions defined for the instances' classes, including assigning references from one object to one or more other objects.

Seventh, for those objects to be saved to the database, application 14 would assign the objects to the appropriate PTR 44 instances and call PTR Set-Modified 310 and PTR Persist 304 on those PTRs 44.

Eighth, for those objects to be explicitly retrieved after they have been saved to the database, application 14 would call PTR Name 306 to associate an object name with each object.

Ninth, application 14 would call OMS Commit-Transaction 112 to end the transaction and save the objects to the database.

Tenth, application **14** would either call OMS Shutdown **106** or delete the instance of OMS **42** to terminate the interface with OMS **42**.

Eleventh, if application **14** had called OMS Shutdown **106**, it would call OMS Startup **102** to restart the interface with OMS **42**. If application **14** had deleted its instance of OMS **42**, it would create a new instance of OMS **42** to restart the interface with OMS **42**.

Twelfth, application **14** would call OMS Begin-Transaction **110** to begin a new transaction.

Thirteenth, OMS Default-Storage-Group **108** would be called to define a new default storage group, if so desired by application **14**, in which newly-created objects would be stored.

Fourteenth, OMS Default-Name-Context **118** would be called to define a new default name context, if so desired by application **14**, in which new object names would be registered.

Fifteenth, application **14** would call OMS Fetch **120–124** to explicitly retrieve one or more persistent objects from the database.

Sixteenth, application **14** could then manipulate the retrieved objects using functions defined for the instances' classes, including assigning references from one object to one or more other objects. Manipulation of these objects would automatically retrieve other persistent objects as they are accessed by application **14**. New objects could also be created by application **14**.

Seventeenth, for those objects to be saved to the database, application **14** would call PTR Set-Modified **310** on the appropriate PTRs **44**. Application **14** would also need to call PTR Persist **304** on the newly-created objects.

Eighteenth, for those newly-created objects to be explicitly retrieved after they have been saved to the database, application **14** would call PTR Name **306** to associate an object name with each object.

Ninteenth, application **14** would call OMS Commit-Transaction **112** to save the modified objects and newly-created objects to the database. Alternatively, application **14** would call OMS Abort-Transaction **114** to discard the modified objects and newly-created objects.

Twentieth, application **14** would either call OMS Shutdown **106** or delete the instance of OMS **42** to terminate the interface with OMS **42**.

While a specific embodiment of the invention has been shown and described, various modifications and alternate embodiments will occur to those skilled in the art. Accordingly, it is intended that the invention be limited only in terms of the appended claims.

We claim:

1. A system for storing objects in a relational database for retrieval by an application program, comprising:

an object manager which interfaces with said application program and performs a plurality of database operations;

a persistent object storage server with a SQL interface to said relational database and an interface to said object manager, wherein said persistent object storage server stores said objects made persistent by said application program; and

an object translator accessible by said object manager, wherein said object translator translates said objects between an object oriented representation and a relational database representation.

2. The system of claim **1**, wherein said system further includes:

a first buffer generated by said object translator, wherein said first buffer contains at least one object; and

a second buffer generated by said object translator, wherein said second buffer contains at least one reference from said at least one object to additional at least one objects, and wherein said first buffer and said second buffer are interpretable by said at least relational database.

3. The system of claim **1**, wherein said persistent object storage server stores said first buffer and said second buffer in said relational database and said persistent object storage server retrieves said first buffer and said second buffer from said relational database for return to said object manager.

4. The system of claim **3**, wherein said object manager passes retrieved objects to said object translator for use by said application program during execution of said application program.

5. The system of claim **1**, wherein said application program is written at least in part in an object-oriented programming language.

6. The system of claim **1**, wherein said interface with said application program manages storage, in a computer's primary memory, of at least one object created by said application program or by said object manager, and said interface directs retrieval of said at least one object from said persistent object storage server upon request by and for use of said application program, and said interface also creates and maintains internal objects in said primary memory accessible only by said object manager.

7. The system of claim **6**, wherein said object translator is adapted to receive said internal objects and said at least one object from said object manager, and

wherein said object translator translates said at least one object from a database management representation to an external representation for said application program to create a translated at least one object,

wherein said object translator translates from said external representation back to said database management representation; and

wherein said object translator passes said translated at least one object back to said object manager.

8. The system of claim **7**, wherein said persistent object storage server interfaces between said relational database and said object manager to store and retrieve said translated at least one object created by said application program or created by said object manager.

9. The system of claim **6**, wherein said internal objects include an architecture identifier that defines the architecture of said internal object and at least one object type description that defines said internal object.

10. The system of claim **1**, wherein said at least one relational database management system provides interfaces to store and retrieve said at least one object and wherein said at least one relational database management system also manages and retains said at least one object until explicitly removed from said at least one relational database system.

11. The system of claim **1**, wherein said system further includes:

a data definition language translator accessible by a user which processes at least one user-specified class definition and generates an equivalent at least one class definition and additional data structures for use by said object manager; and

wherein said data definition language translator generates at least one object type description for use by said object translator as well as said object manager during execution of said at least one application program and generates class definitions for use by said user in building an application program.

12. The system of claim 1, wherein said interface with said application program and said at least one relational database management system includes:

an indicator and a retriever that allows said application program to dynamically specify retrieval of objects during execution of said application program;

wherein said indicator is set by said application program and specifies whether a referenced object should be retrieved from said at least one database when any referencing object is retrieved; and

wherein said retriever examines said indicator and determines whether said referenced objects are to be retrieved when their referencing object is retrieved, or whether to retrieve said referenced objects only when they are accessed through said referencing object.

13. A system for storing objects in a relational database for retrieval by an application program, comprising:

an object manager which interfaces with said application program and performs a plurality of database operations;

a persistent object storage server with a SQL interface to said relational database and an interface to said object manager, wherein said persistent object storage server stores said objects made persistent by said application program; and

an object translator accessible by said object manager, wherein said object translator translates said objects between primary and secondary memory representations in a computer architecture independent method;

a first buffer generated by said object translator, wherein said first buffer contains at least one object; and

a second buffer generated by said object translator, wherein said second buffer contains at least one reference from said at least one object to additional at least one objects, wherein said first buffer and said second buffer are interpretable by said at least relational database; and

wherein said object manager passes retrieved objects to said object translator for use by said application program during execution of said application program;

wherein said persistent object storage server stores said first buffer and said second buffer into said relational database; and

wherein said persistent object storage server retrieves said first buffer and said second buffer from said relational database for return to said object manager.

14. The system of claim 13, wherein said application program is written at least in part in an object-oriented programming language.

15. The system of claim 13, wherein said interface with said application program manages storage, in a computer's primary memory, of at least one object created by said application program or by said object manager, and said interface directs retrieval of said at least one object from said persistent object storage server upon request by and for use of said application program, and said interface also creates and maintains internal objects in said primary memory accessible only by said object manager.

16. The system of claim 15, wherein said object translator is adapted to receive said internal objects and said at least one object from said object manager, and

wherein said object translator translates said at least one object from a database management representation to an external representation for said application program to create a translated at least one object,

wherein said object translator translates from said external representation back to said database management representation; and

wherein said object translator passes said translated at least one object back to said object manager.

17. The system of claim 16, wherein said persistent object storage server interfaces between at least one relational database management system and said object manager to store and retrieve said translated at least one object created by said application program or created by said object manager.

18. The system of claim 15, wherein said internal objects include an architecture identifier that defines the architecture of said internal object and at least one object type description that defines said internal object.

19. The system of claim 13, wherein said at least one relational database management system provides interfaces to store and retrieve said at least one object and wherein said at least one relational database management system also manages and retains said at least one object until explicitly removed from said at least one relational database system.

20. The system of claim 13, wherein said system further includes:

a data definition language translator accessible by a user which processes at least one user-specified class definition and generates an equivalent at least one class definition and additional data structures for use by said object manager; and

wherein said data definition language translator generates at least one object type description for use by said object translator as well as said object manager during execution of said at least one application program and generates class definitions for use by said user in building an application program.

21. The system of claim 13, wherein said interface with said application program and said at least one relational database management system includes:

an indicator and a retriever that allows said application program to dynamically specify retrieval of objects during execution of said application program;

wherein said indicator is set by said application program and specifies whether a referenced object should be retrieved from said at least one database when any referencing object is retrieved; and

wherein said retriever examines said indicator and determines whether said referenced objects are to be retrieved when their referencing object is retrieved, or whether to retrieve said referenced objects only when they are accessed through said referencing object.

* * * * *

EXHIBIT D

US005742538A

United States Patent [19]

Guttag et al.

[11] Patent Number: 5,742,538

[45] Date of Patent: Apr. 21, 1998

[54] LONG INSTRUCTION WORD CONTROLLING PLURAL INDEPENDENT PROCESSOR OPERATIONS

[75] Inventors: **Karl M. Guttag**, Missouri City; **Christopher J. Read**, Houston, both of Tex.; **Keith Balmer**, Bedford, England

[73] Assignee: **Texas Instruments Incorporated**, Dallas, Tex.

[21] Appl. No.: **632,785**

[22] Filed: **Apr. 15, 1996**

Related U.S. Application Data

[62] Division of Ser. No. 160,297, Nov. 30, 1993, Pat. No. 5,509,129.

[51] **Int. Cl.**[6] **G06F 7/52**

[52] **U.S. Cl.** **364/754.01**; 364/760.01

[58] **Field of Search** 364/754.01, 757, 364/760.01

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,319,325	3/1982	Hoff, Jr. et al.	364/200
4,519,031	5/1985	Magar et al.	364/200
4,760,545	7/1988	Inagami et al.	364/200
4,771,379	9/1988	Ando et al.	364/200
4,809,212	2/1989	New et al.	364/754.01
4,811,269	3/1989	Hirose et al.	364/754
4,864,528	9/1989	Nishiyama et al.	364/754
4,866,657	9/1989	Nishiyama et al.	364/768
4,868,777	9/1989	Nishiyama et al.	364/754
4,985,848	1/1991	Pfeiffer et al.	364/518
5,001,662	3/1991	Baum	364/787
5,031,136	7/1991	Nishiyama et al.	364/746.2
5,065,052	11/1991	Sakagami et al.	307/480
5,126,964	6/1992	Zurawski	364/787
5,197,140	3/1993	Balmer	395/400
5,442,799	8/1995	Murakami et al.	395/800
5,530,661	6/1996	Garbe et al.	364/728.01
5,537,601	7/1996	Kimura et al.	395/800
5,586,070	12/1996	Purcell	364/754.01

OTHER PUBLICATIONS

Julie Shipnes, Graphics Processing with the 88110 RISC Microprocessor, IEEE 1992, pp. 169–174.

Real DSP Hardware, Richard J. Higgins, Digital Signal Processing in VLSI, 1990 by Analog Devices, pp. 307–327, 365–400.

(List continued on next page.)

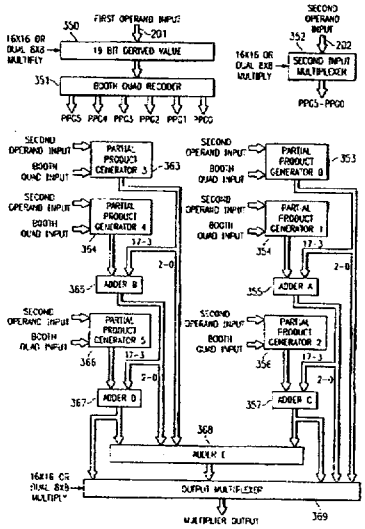
Primary Examiner—Tan V. Mai

Attorney, Agent, or Firm—Robert D. Marshall, Jr.; James C. Kesterson; Richard L. Donaldson

[57] **ABSTRACT**

A data processing apparatus including a multiplier unit forming a product from L bits of each two data buses of N bits each N is greater than L. The multiplier forms a N bit output having a first portion which is the L most significant bits of the of product and a second portion which is M other bits not including the L least significant bits of the product, where N is the sum of M and L. In the preferred embodiment the M other bits are derived from other bits of the two input data busses, such as the M other bits of the first input data bus. An arithmetic logic unit performs parallel operations (addition, subtraction, Boolean functions) controlled by the same instructions. This arithmetic logic unit is divisible into a selected number of sections for performing identical operations on independent sections of its inputs. The multiplier unit may form dual products from separate parts of the input data. A single instruction controlling both the multiplier unit and the arithmetic logic unit permits addition of dual products. The dual products are temporarily stored in a data register permitting the multiply and add operations to be pipelined. The dual products are formed in one data word and added by a rotate/mask and add operation in a three input arithmetic unit.

29 Claims, 37 Drawing Sheets



OTHER PUBLICATIONS

DSP Microcomputer, Analog Devices (undated).

The S2811 Signal Processing Peripheral, William Nicholson et al, American Microsystems, Inc., pp. 1–12 (undated).

A Single Chip NMoS Signal Processor, M. Townsend et al, Intel Corp., pp. 390–393, IEEE 1980.

An NMOS Microprocessor Analog Signal Processing, Matt Townsend et al, IEEE Journal of Solid–State Circuits, vol. SC–15, No. 1, pp. 33–38 Feb. 1980.

TMS320C2X User's Guide, Dec. 1990, pp. 4–117–4–129.

DSP96002 IEEE Floating–Point Dual Port Processor User's Manual, DSP, Motorola, 1989, pp. 3–6–3–9, 6–2, A–138–A149.

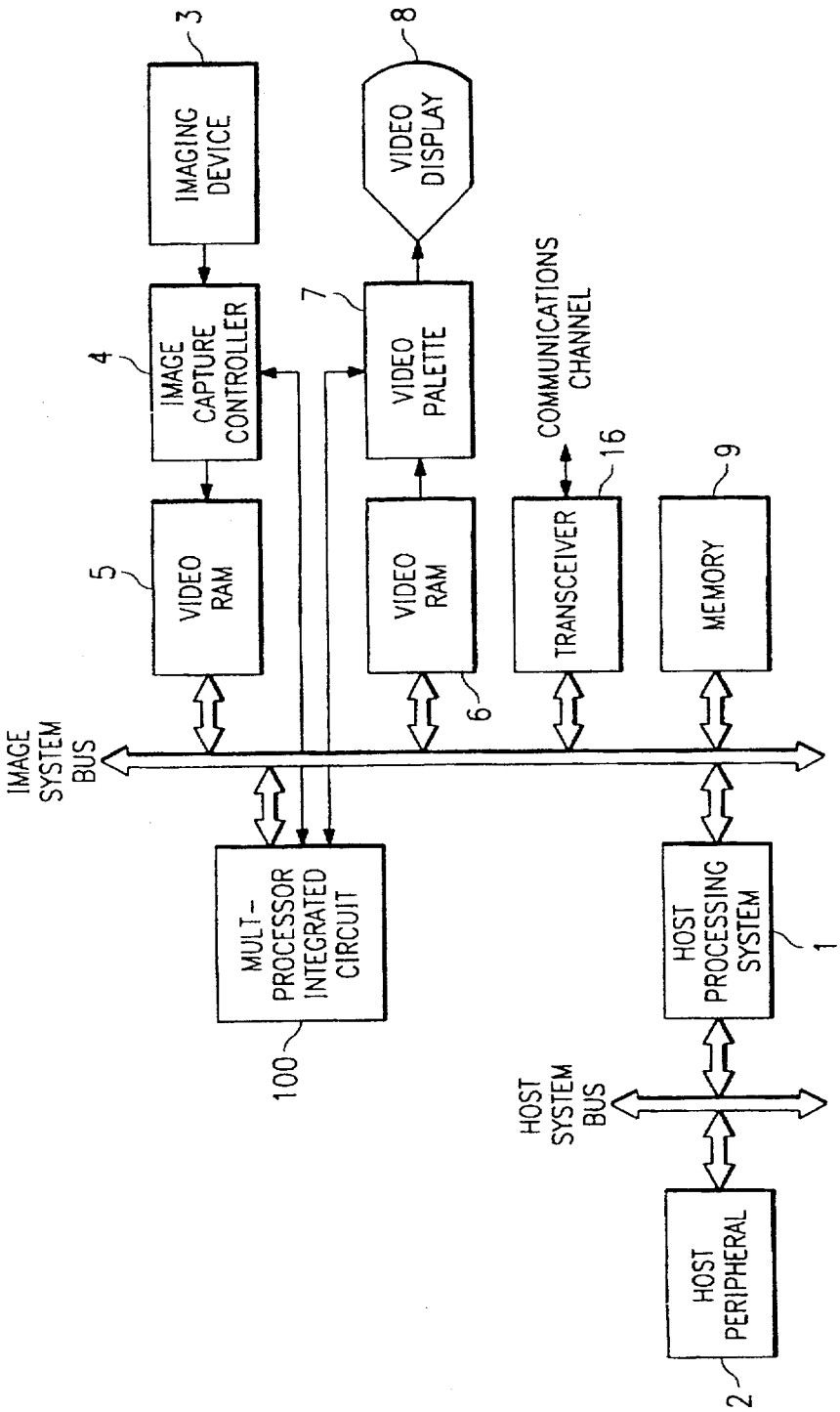

IMAGING DEVICE
3
IMAGE CAPTURE CONTROLLER
4
VIDEO RAM
5
VIDEO DISPLAY
8
VIDEO PALETTE
7
VIDEO RAM
6
COMMUNICATIONS CHANNEL
TRANSCEIVER
16
MEMORY
9
IMAGE SYSTEM BUS
MULT-PROCESSOR INTEGRATED CIRCUIT
100
HOST PROCESSING SYSTEM
1
HOST SYSTEM BUS
HOST PERIPHERAL
2

FIG. 1

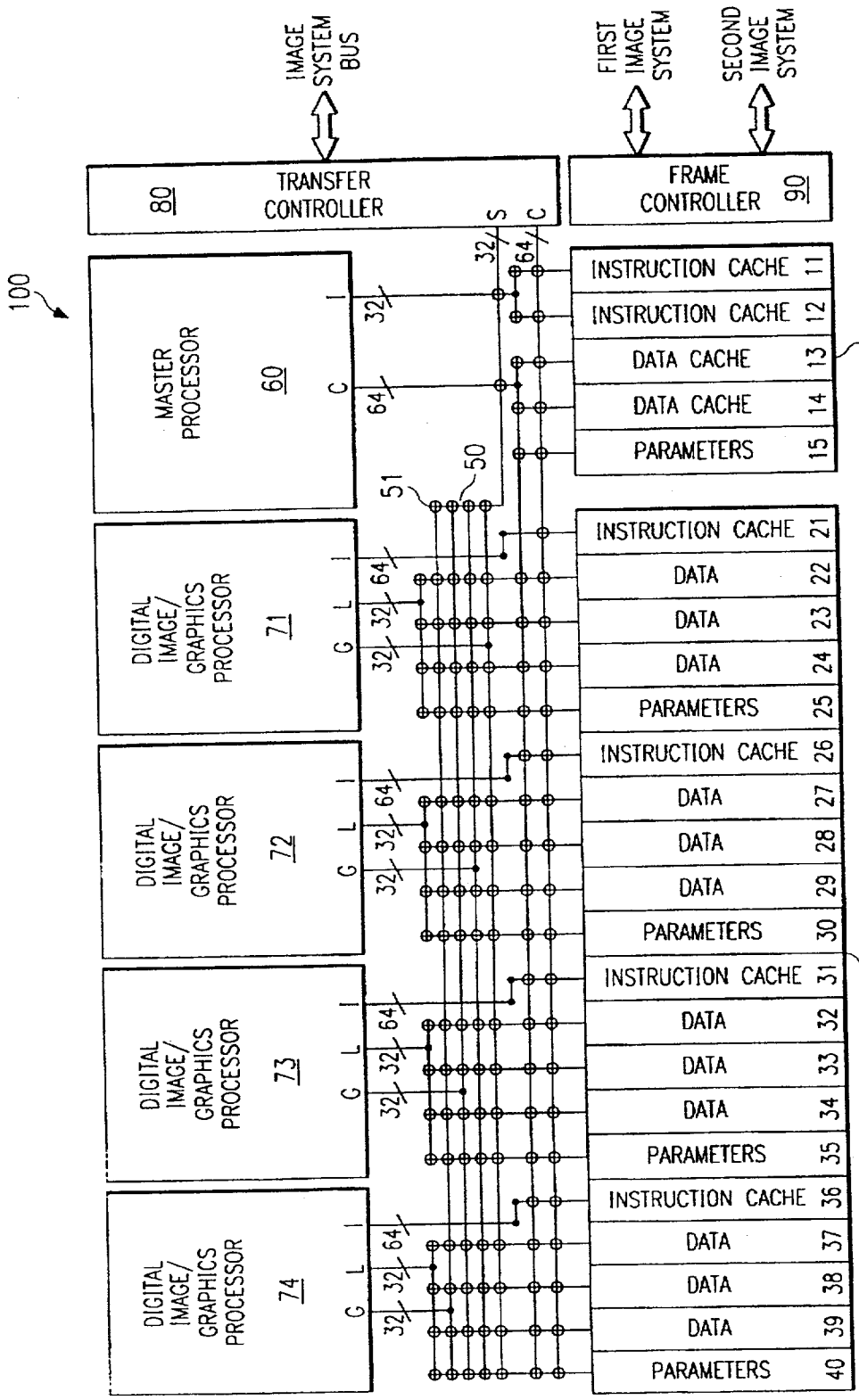


IMAGE SYSTEM BUS
FIRST IMAGE SYSTEM
SECOND IMAGE SYSTEM
TRANSFER CONTROLLER 80
FRAME CONTROLLER 90
100
MASTER PROCESSOR 60
DIGITAL IMAGE/ GRAPHICS PROCESSOR 71
DIGITAL IMAGE/ GRAPHICS PROCESSOR 72
DIGITAL IMAGE/ GRAPHICS PROCESSOR 73
DIGITAL IMAGE/ GRAPHICS PROCESSOR 74
I
C
G
L
S
32
64
50
51
INSTRUCTION CACHE 11
INSTRUCTION CACHE 12
DATA CACHE 13
DATA CACHE 14
PARAMETERS 15
10
INSTRUCTION CACHE 21
DATA 22
DATA 23
DATA 24
PARAMETERS 25
INSTRUCTION CACHE 26
DATA 27
DATA 28
DATA 29
PARAMETERS 30
INSTRUCTION CACHE 31
DATA 32
DATA 33
DATA 34
PARAMETERS 35
INSTRUCTION CACHE 36
DATA 37
DATA 38
DATA 39
PARAMETERS 40
20

FIG. 2

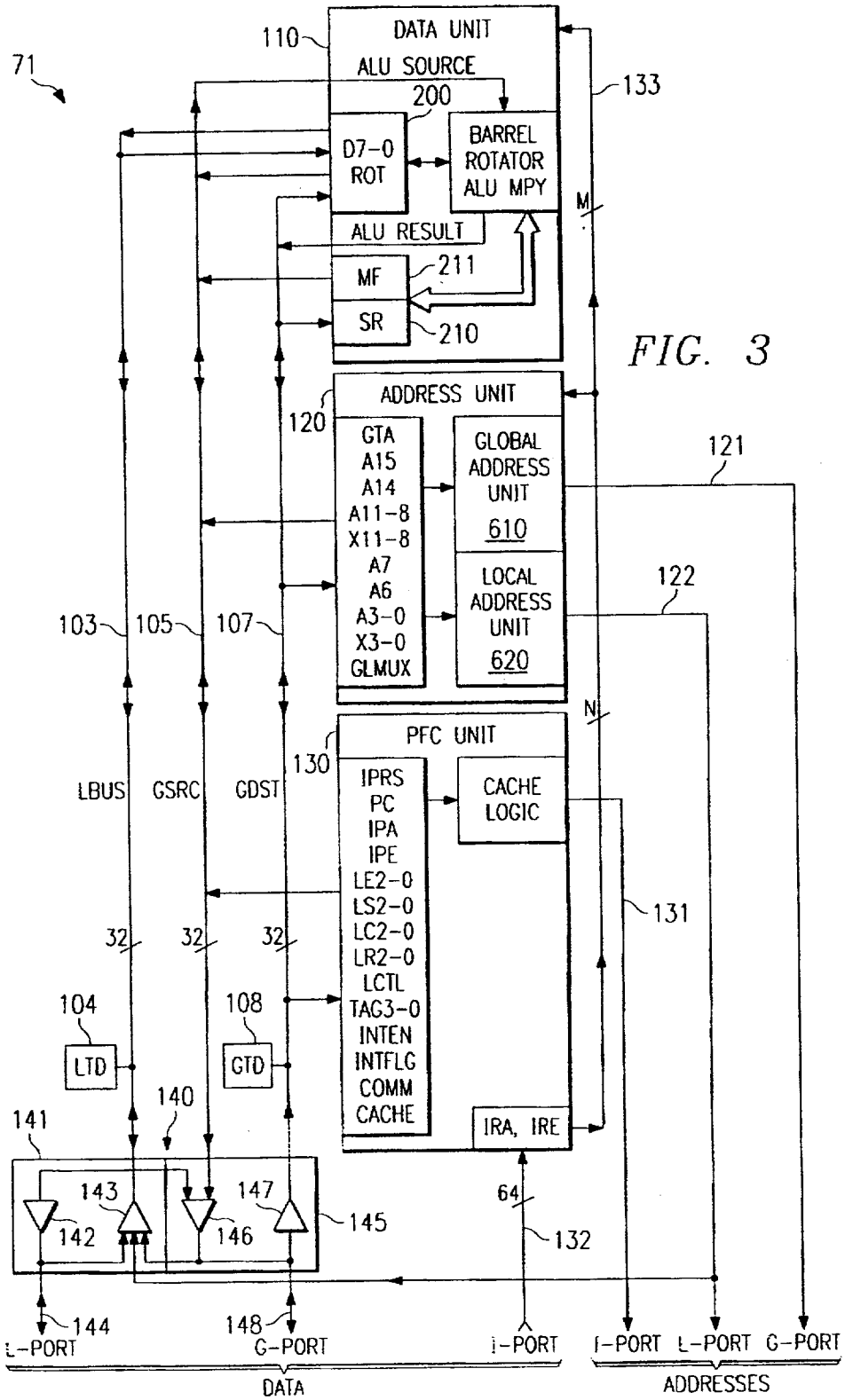

71
110
DATA UNIT
ALU SOURCE
200
D7-0
ROT
BARREL
ROTATOR
ALU MPY
133
M
ALU RESULT
MF
211
SR
210
ADDRESS UNIT
120
GTA
A15
A14
A11-8
X11-8
A7
A6
A3-0
X3-0
GLMUX
GLOBAL
ADDRESS
UNIT
610
LOCAL
ADDRESS
UNIT
620
121
122
103
105
107
N
PFC UNIT
130
IPRS
PC
IPA
IPE
LE2-0
LS2-0
LC2-0
LR2-0
LCTL
TAG3-0
INTEN
INTFLG
COMM
CACHE
CACHE
LOGIC
IRA, IRE
131
LBUS
GSRC
GDST
32
32
32
104
108
LTD
GTD
140
141
143
147
142
146
145
64
132
144
148
L-PORT
G-PORT
I-PORT
I-PORT
L-PORT
G-PORT
DATA
ADDRESSES

FIG. 3

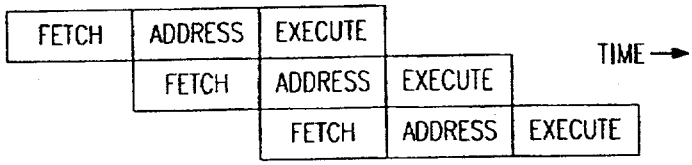


FIG. 4

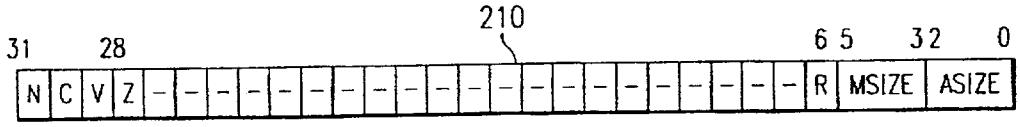


FIG. 6

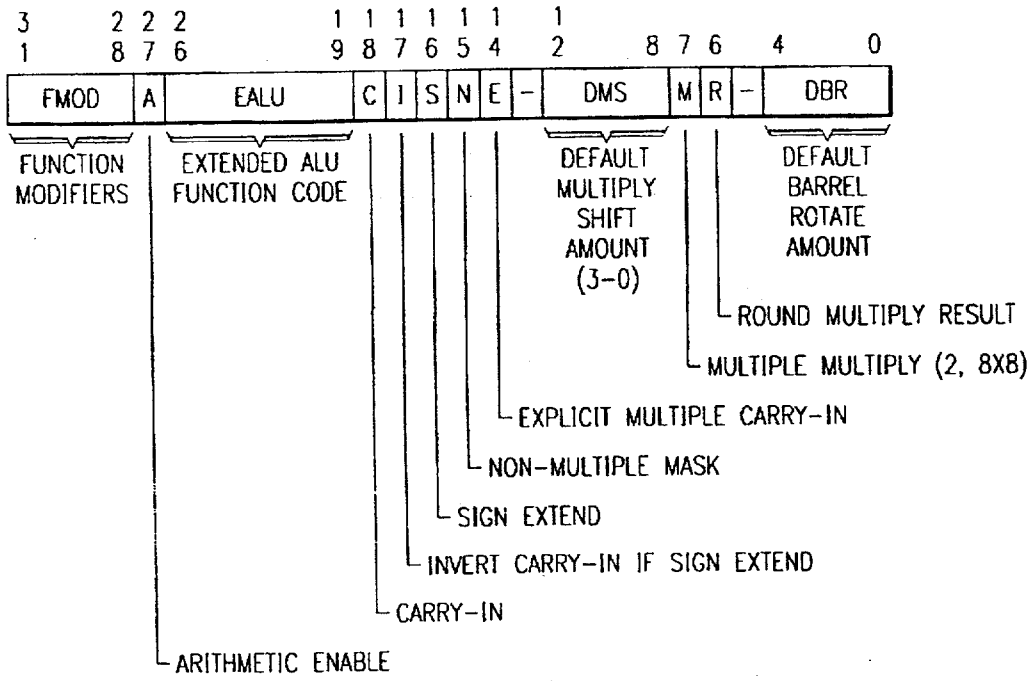


FIG. 9

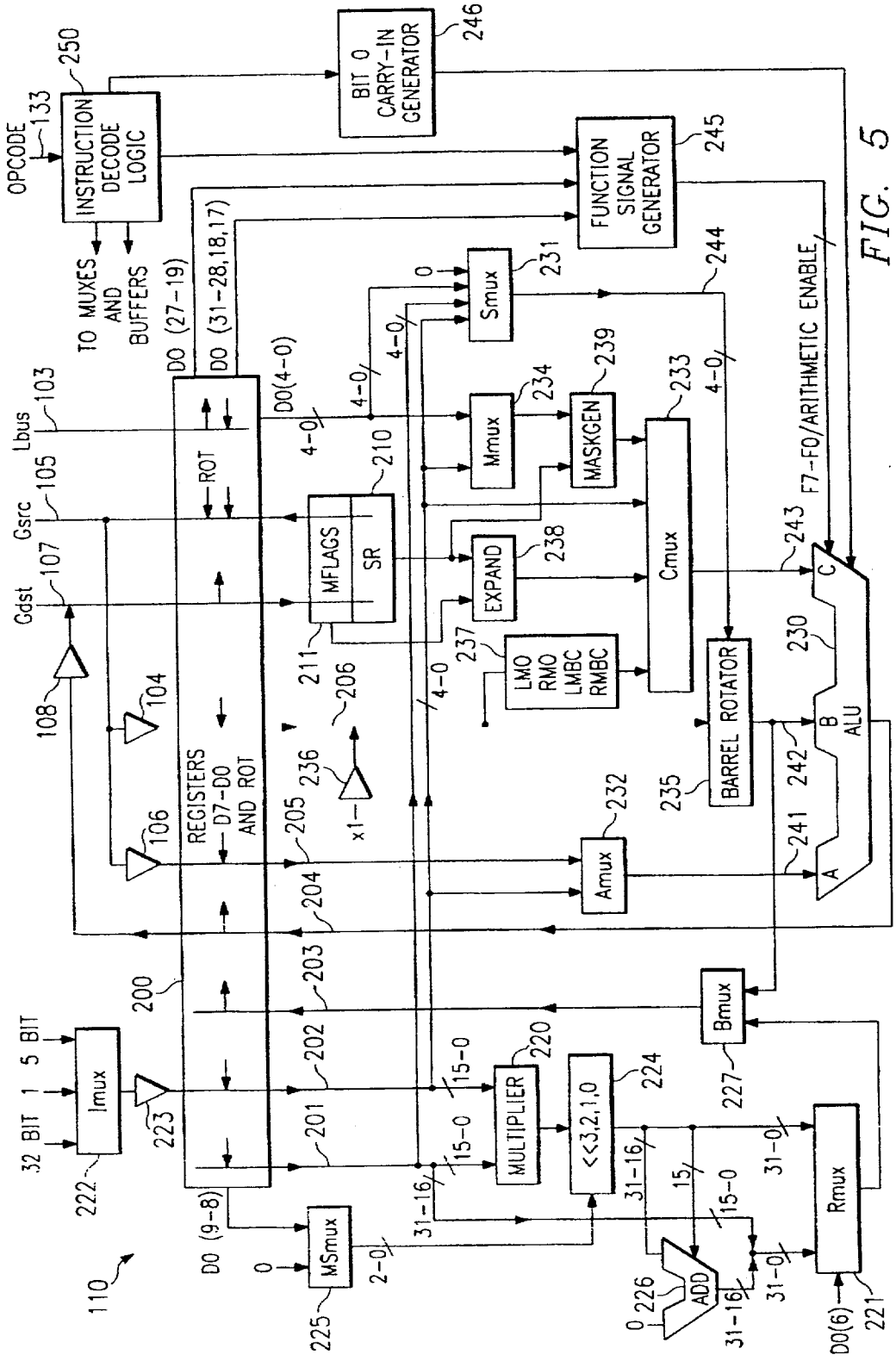

OPCODE
133
INSTRUCTION DECODE LOGIC
250
TO MUXES AND BUFFERS
BIT 0 CARRY-IN GENERATOR
246
FUNCTION SIGNAL GENERATOR
245
DO (27-19)
DO (31-28,18,17)
DO(4-0)
Lbus
103
Gsrc
105
Gdst
107
ROT
REGISTERS D7-D0 AND ROT
MFLAGS
SR
210
211
Smux
231
Mmux
234
MASKGEN
239
Cmux
233
EXPAND
238
LMO RMO LMBC RMBC
237
BARREL ROTATOR
235
F7-F0/ARITHMETIC ENABLE
243
C
230
B
ALU
242
A
241
Amux
232
Bmux
227
Imux
222
32 BIT
1
5 BIT
223
200
201
202
203
204
205
206
236
x1
MULTIPLIER
220
<<3,2,1,0
224
MSmux
225
DO (9-8)
ADD
226
Rmux
221
DO(6)
110
4-0
15-0
31-16
31-0
2-0
15
0

FIG. 5

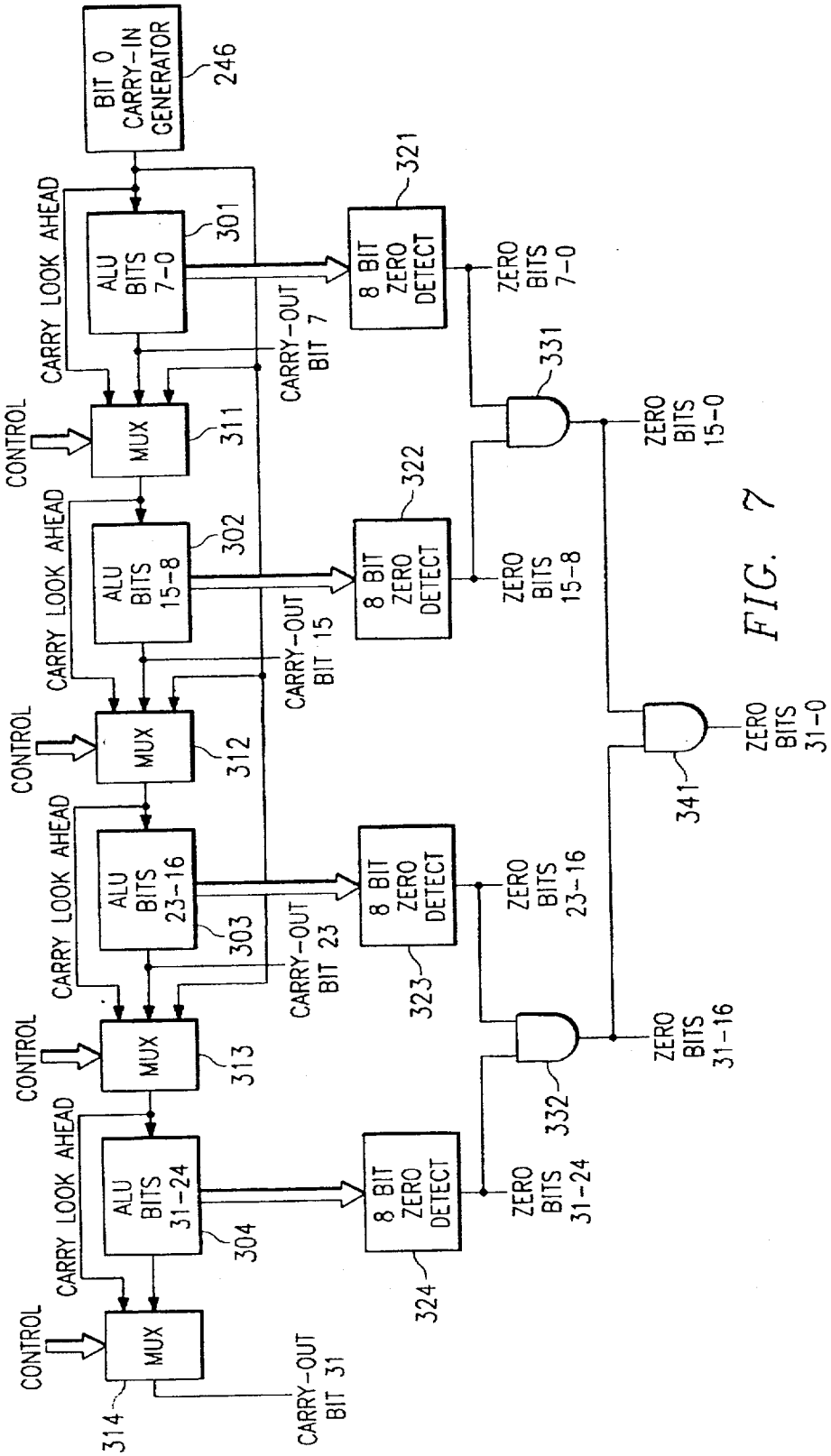

BIT 0 CARRY-IN GENERATOR
246
CARRY LOOK AHEAD
ALU BITS 7-0
301
CONTROL
MUX
311
CARRY-OUT BIT 7
8 BIT ZERO DETECT
321
ZERO BITS 7-0
331
ZERO BITS 15-0
CARRY LOOK AHEAD
ALU BITS 15-8
302
CONTROL
MUX
312
CARRY-OUT BIT 15
8 BIT ZERO DETECT
322
ZERO BITS 15-8
ZERO BITS 31-0
341
CARRY LOOK AHEAD
ALU BITS 23-16
303
CONTROL
MUX
313
CARRY-OUT BIT 23
8 BIT ZERO DETECT
323
ZERO BITS 23-16
ZERO BITS 31-16
332
CARRY LOOK AHEAD
ALU BITS 31-24
304
CONTROL
MUX
314
CARRY-OUT BIT 31
8 BIT ZERO DETECT
324
ZERO BITS 31-24

FIG. 7

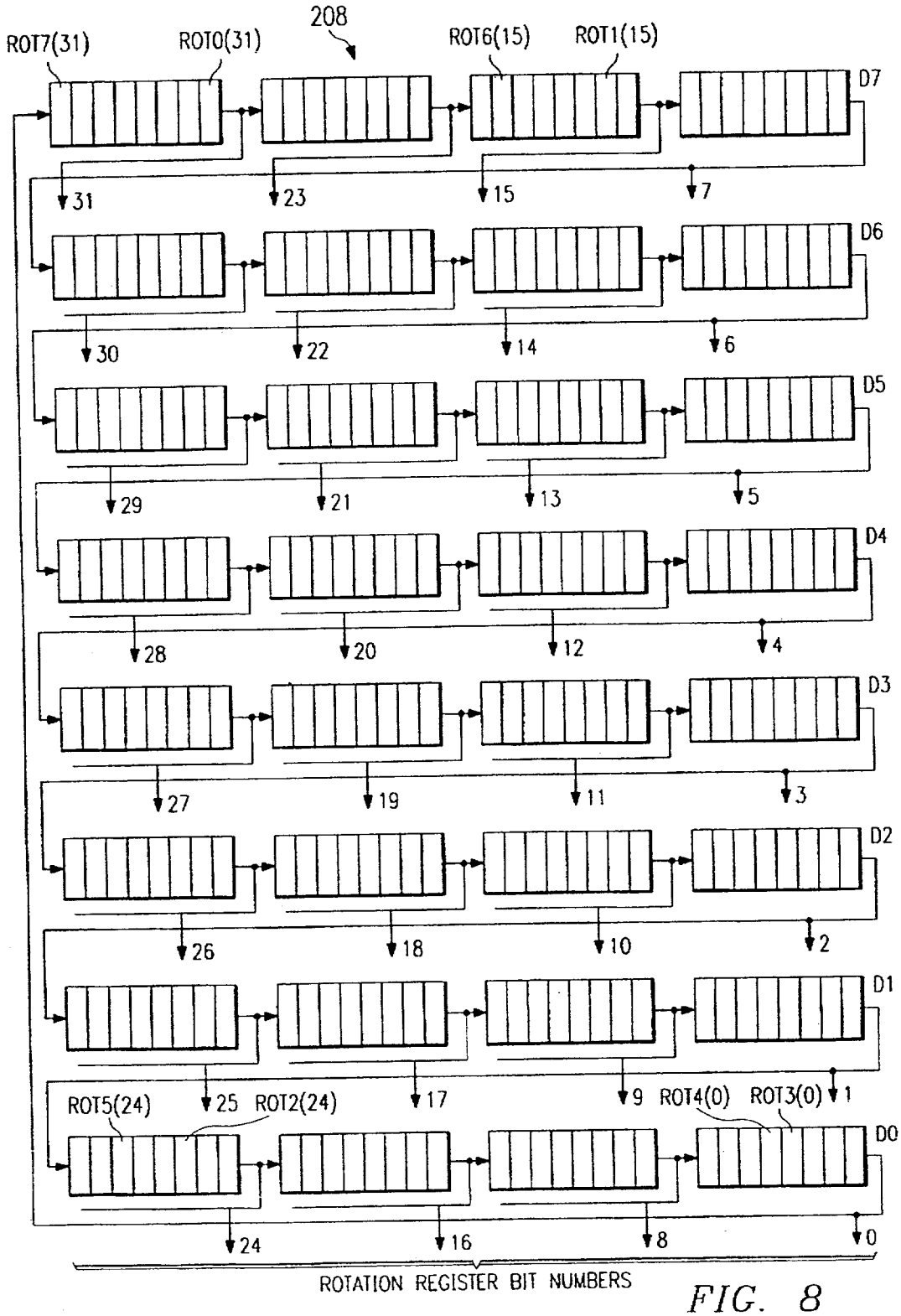

208
ROT7(31)
ROT0(31)
ROT6(15)
ROT1(15)
D7
31
23
15
7
D6
30
22
14
6
D5
29
21
13
5
D4
28
20
12
4
D3
27
19
11
3
D2
26
18
10
2
D1
ROT5(24)
25
ROT2(24)
17
9
ROT4(0)
ROT3(0)
1
D0
24
16
8
0
ROTATION REGISTER BIT NUMBERS

FIG. 8

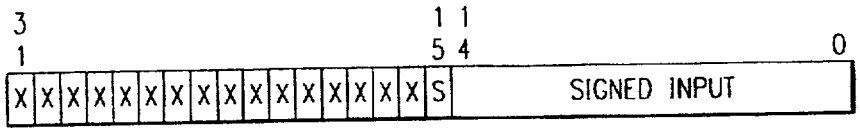


FIG. 10a

FIG. 10b ("01" IF HEX "8000" X HEX "8000")

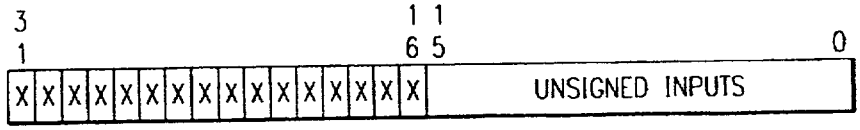


FIG. 10c

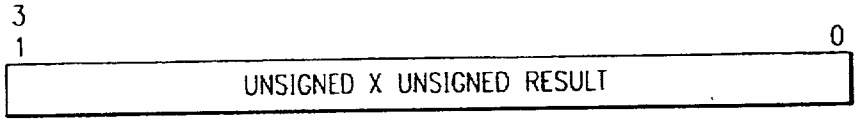


FIG. 10d

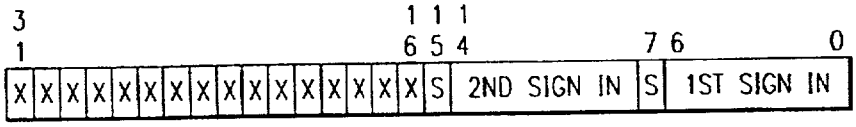


FIG. 11a

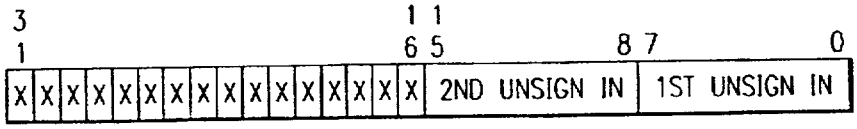


FIG. 11b

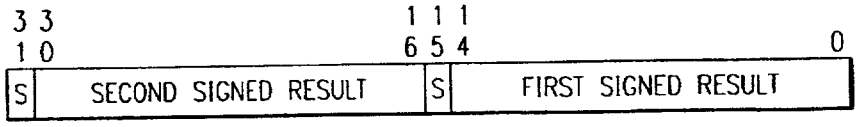


FIG. 11c

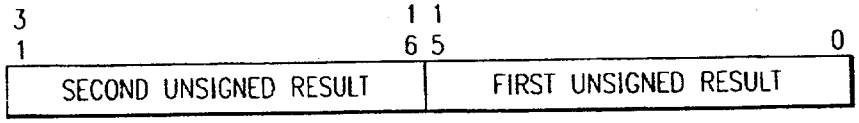


FIG. 11d

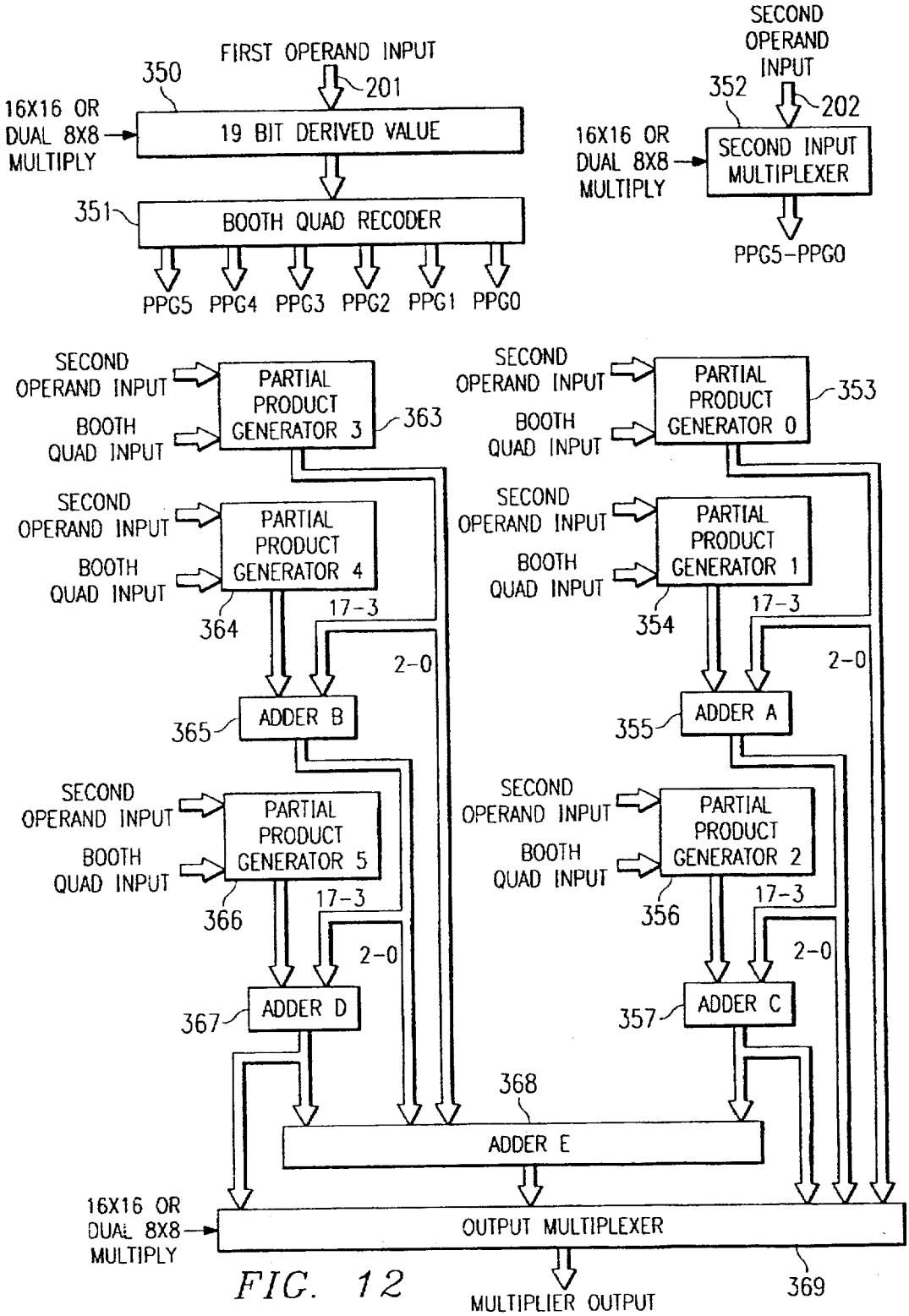

FIRST OPERAND INPUT
201
350
16X16 OR DUAL 8X8 MULTIPLY
19 BIT DERIVED VALUE
351
BOOTH QUAD RECODER
PPG5 PPG4 PPG3 PPG2 PPG1 PPG0
SECOND OPERAND INPUT
202
352
16X16 OR DUAL 8X8 MULTIPLY
SECOND INPUT MULTIPLEXER
PPG5-PPG0
SECOND OPERAND INPUT
BOOTH QUAD INPUT
PARTIAL PRODUCT GENERATOR 3
363
PARTIAL PRODUCT GENERATOR 0
353
PARTIAL PRODUCT GENERATOR 4
364
PARTIAL PRODUCT GENERATOR 1
354
17-3
2-0
ADDER B
365
ADDER A
355
PARTIAL PRODUCT GENERATOR 5
366
PARTIAL PRODUCT GENERATOR 2
356
ADDER D
367
ADDER C
357
368
ADDER E
16X16 OR DUAL 8X8 MULTIPLY
OUTPUT MULTIPLEXER
369
MULTIPLIER OUTPUT

FIG. 12

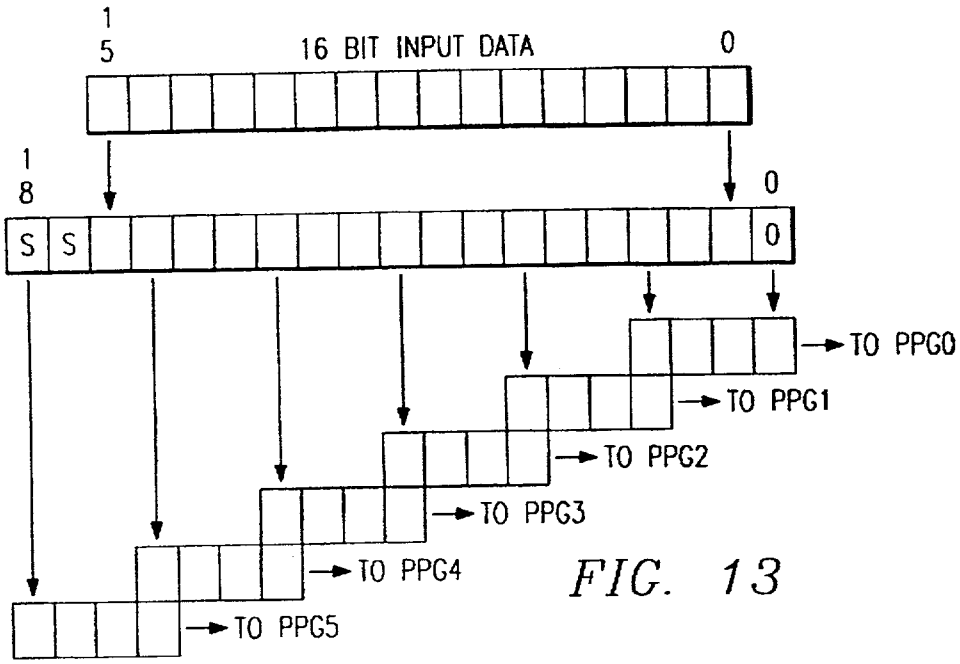

15
16 BIT INPUT DATA
0
18
0
S
S
0
TO PPG0
TO PPG1
TO PPG2
TO PPG3
TO PPG4
TO PPG5

FIG. 13

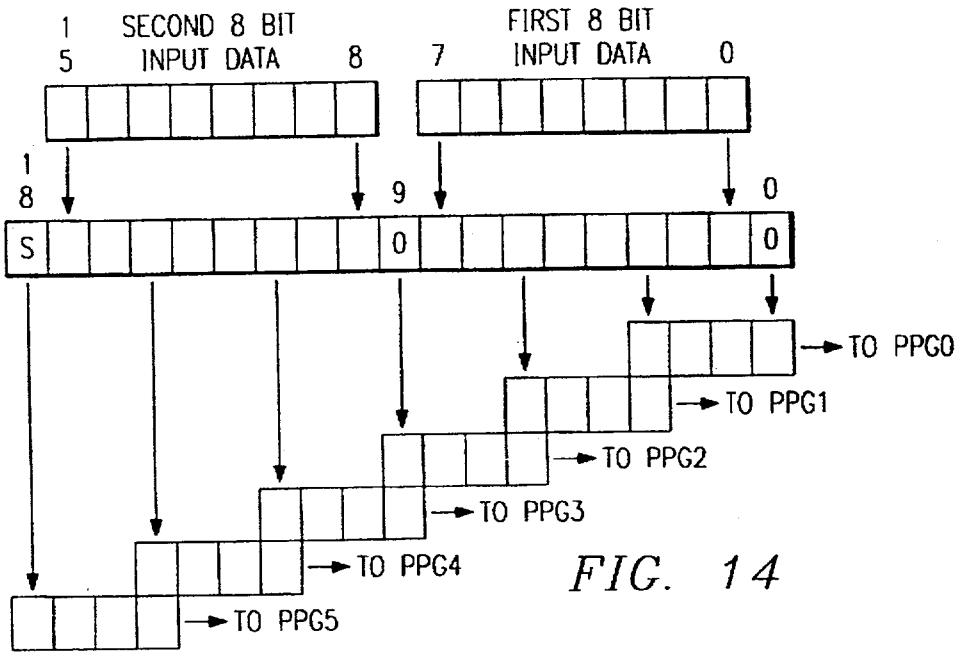

15
SECOND 8 BIT INPUT DATA
8
7
FIRST 8 BIT INPUT DATA
0
18
9
0
S
0
0
TO PPG0
TO PPG1
TO PPG2
TO PPG3
TO PPG4
TO PPG5

FIG. 14

FIG. 15a

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	16 BIT UNSIGNED OPERAND															

FIG. 15b

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	S	S	15 BIT SIGNED OPERAND														

FIG. 16a

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	8 BIT UNSIGN OP							

FIG. 16b

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	S	S	S	S	S	S	S	S	S	S	7 BIT SIGN OP						

FIG. 16c

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	8 BIT UNSIGN OP								0	0	0	0	0	0	0	0

FIG. 16d

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	S	S	7 BIT SIGN OP							0	0	0	0	0	0	0	0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
E22	E21	E20	E19	E18	E17	E16	E15	E14	E13	E12	E11	E10	E9	E8	E7	E6	E5	E4	E3	E2	E1	E0	C2	C1	C0	A2	A1	A0	P2	P1	P0

FIG. 17a

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	C9	C8	C7	C6	C5	C4	C3	C2	C1	C0	A2	A1	A0	P2	P1	P0

FIG. 17b

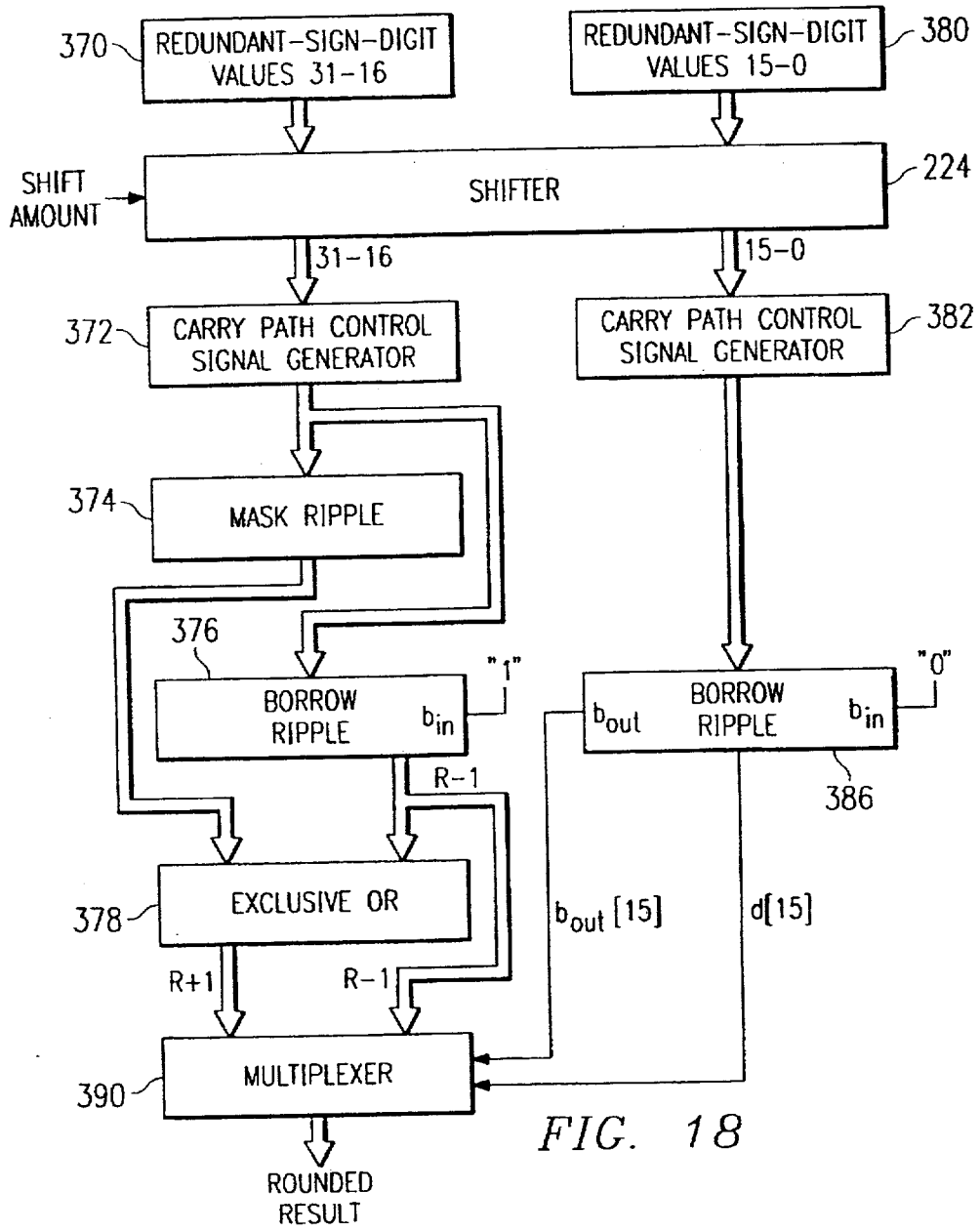

370
REDUNDANT-SIGN-DIGIT VALUES 31-16
REDUNDANT-SIGN-DIGIT VALUES 15-0
380
SHIFT AMOUNT
SHIFTER
224
31-16
15-0
372
CARRY PATH CONTROL SIGNAL GENERATOR
CARRY PATH CONTROL SIGNAL GENERATOR
382
374
MASK RIPPLE
376
BORROW RIPPLE
b_{in}
"1"
b_{out}
BORROW RIPPLE
b_{in}
"0"
R-1
386
378
EXCLUSIVE OR
b_{out} [15]
d[15]
R+1
R-1
390
MULTIPLEXER
ROUNDED RESULT

FIG. 18

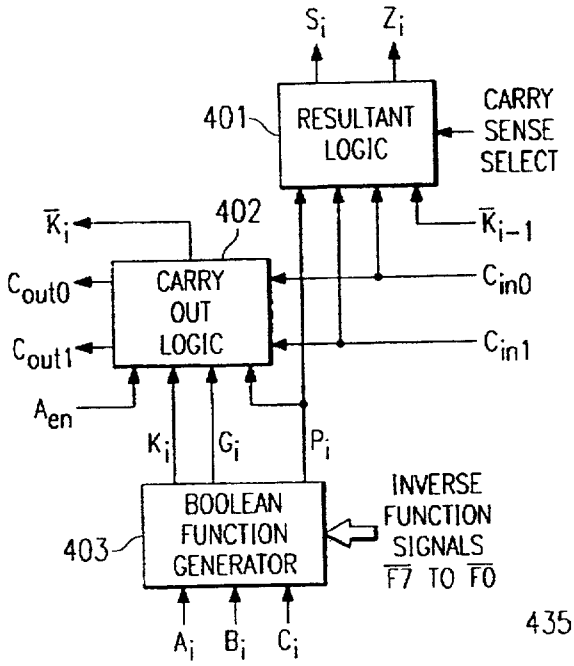

S_i
Z_i
401
RESULTANT LOGIC
CARRY SENSE SELECT
402
$\overline{K}_i$
$\overline{K}_{i-1}$
C_{out0}
CARRY OUT LOGIC
C_{in0}
C_{out1}
C_{in1}
A_{en}
K_i
G_i
P_i
403
BOOLEAN FUNCTION GENERATOR
INVERSE FUNCTION SIGNALS $\overline{F7}$ TO $\overline{F0}$
A_i
B_i
C_i

FIG. 19

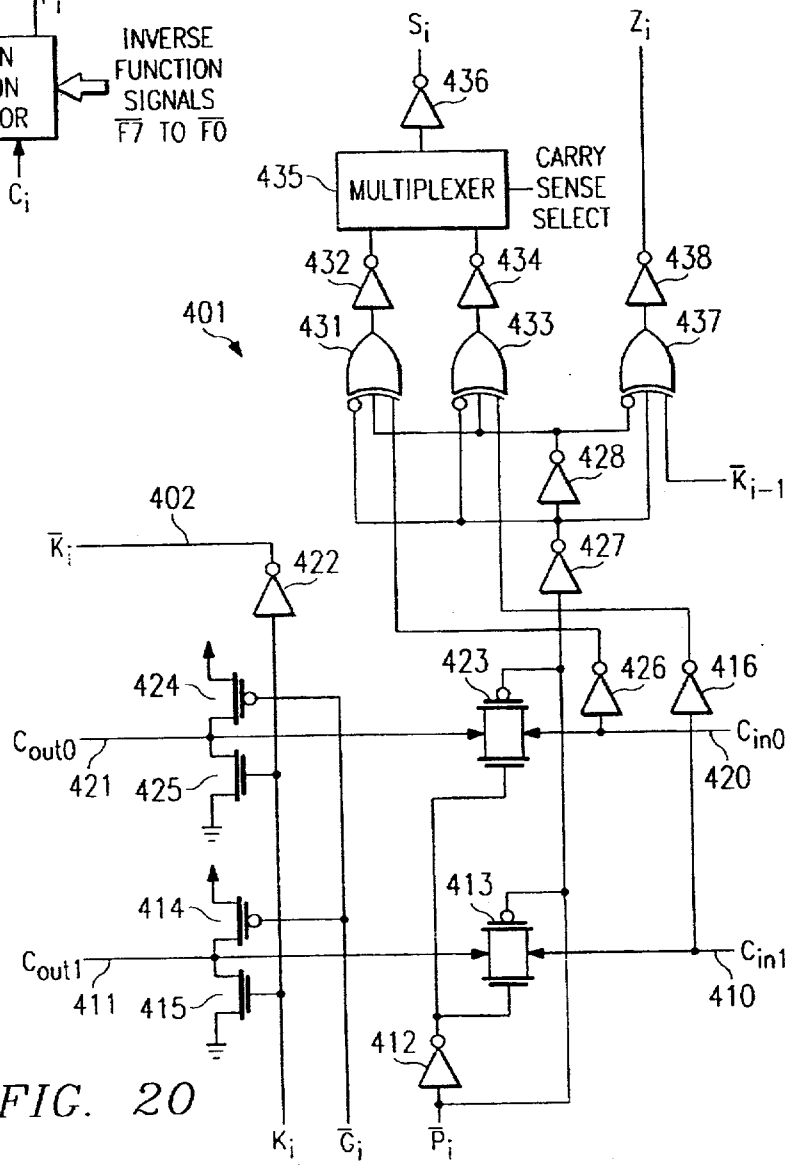

S_i
Z_i
436
435
MULTIPLEXER
CARRY SENSE SELECT
432
434
438
401
431
433
437
428
$\overline{K}_{i-1}$
402
$\overline{K}_i$
422
427
424
423
426
416
C_{out0}
C_{in0}
421
425
420
414
413
C_{out1}
C_{in1}
411
415
410
412
K_i
$\overline{G}_i$
$\overline{P}_i$

FIG. 20

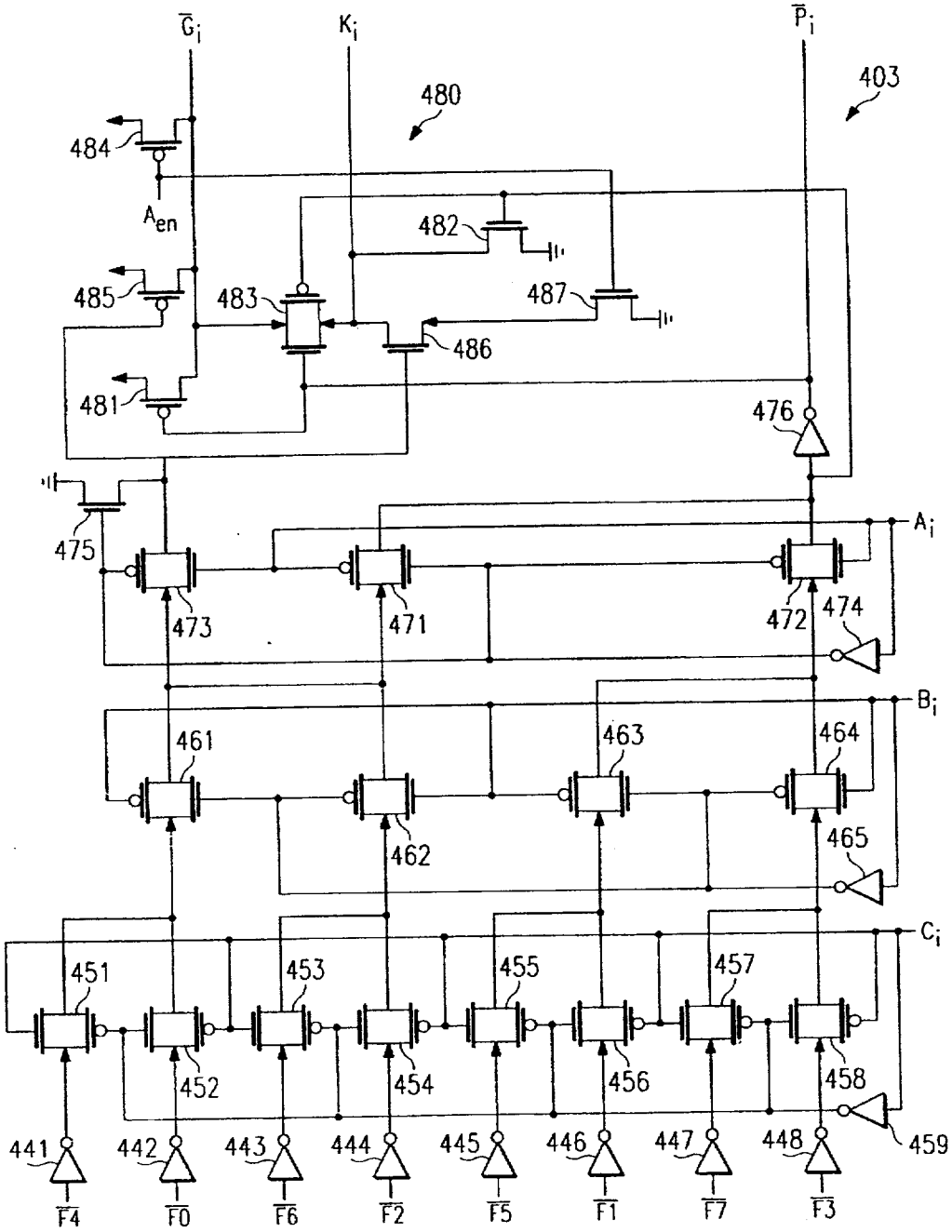

$\overline{G}_i$
K_i
$\overline{P}_i$
403
480
484
A_{en}
482
485
483
487
486
481
476
475
A_i
473
471
472
474
B_i
461
463
464
462
465
C_i
451
453
455
457
452
454
456
458
459
441
442
443
444
445
446
447
448
$\overline{F4}$
$\overline{F0}$
$\overline{F6}$
$\overline{F2}$
$\overline{F5}$
$\overline{F1}$
$\overline{F7}$
$\overline{F3}$

FIG. 21

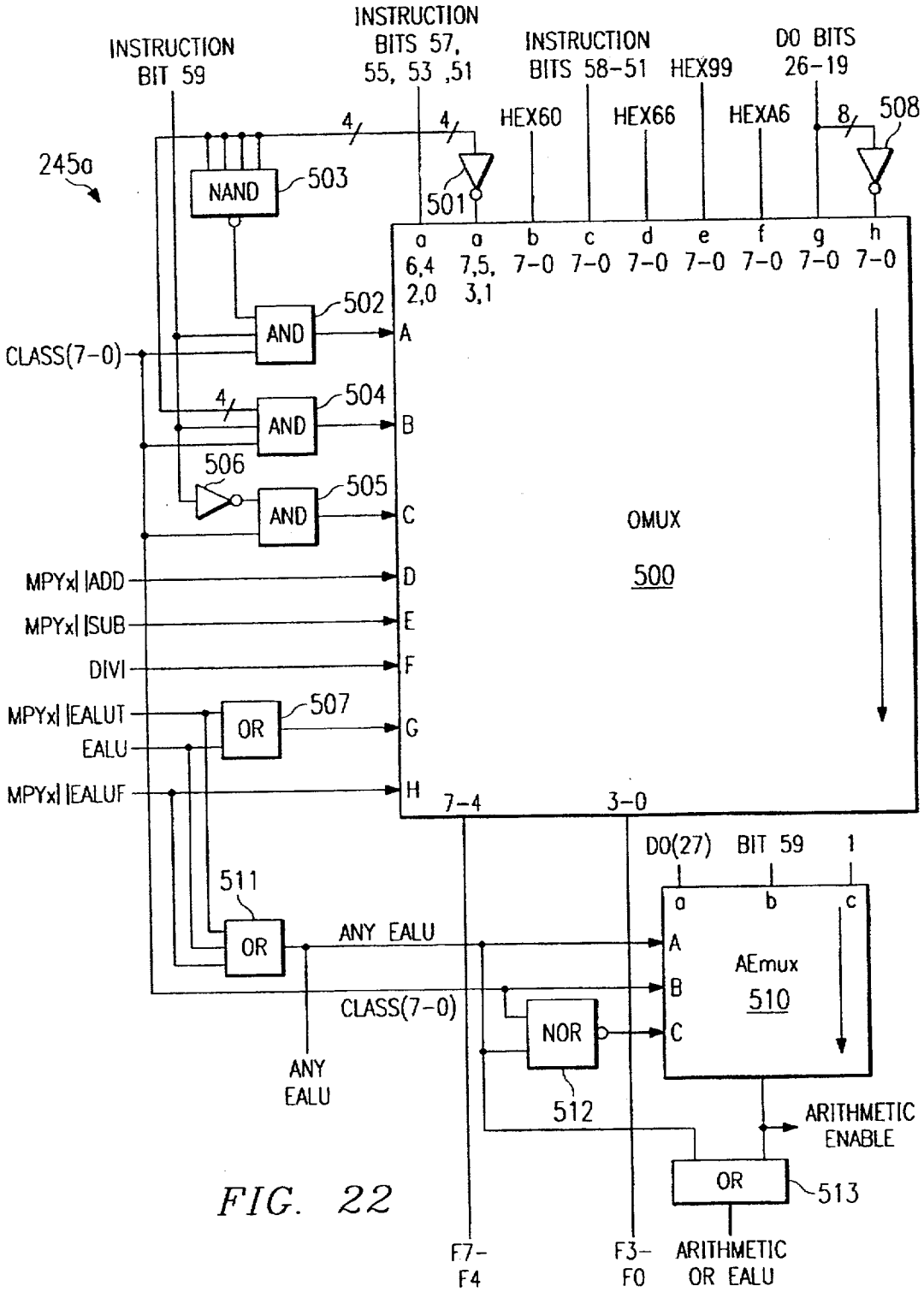

INSTRUCTION BIT 59
INSTRUCTION BITS 57, 55, 53 ,51
INSTRUCTION BITS 58-51
HEX99
DO BITS 26-19
HEX60
HEX66
HEXA6
508
245a
NAND
503
501
a 6,4 2,0
a 7,5, 3,1
b 7-0
c 7-0
d 7-0
e 7-0
f 7-0
g 7-0
h 7-0
502
AND
A
CLASS(7-0)
504
AND
B
506
505
AND
C
OMUX
500
MPYx| |ADD
D
MPYx| |SUB
E
DIVI
F
MPYx| |EALUT
507
OR
G
EALU
MPYx| |EALUF
H
7-4
3-0
DO(27)
BIT 59
1
511
OR
ANY EALU
a
b
c
A
AEmux
510
B
CLASS(7-0)
NOR
C
ANY EALU
512
ARITHMETIC ENABLE
OR
513
F7-F4
F3-F0
ARITHMETIC OR EALU

FIG. 22

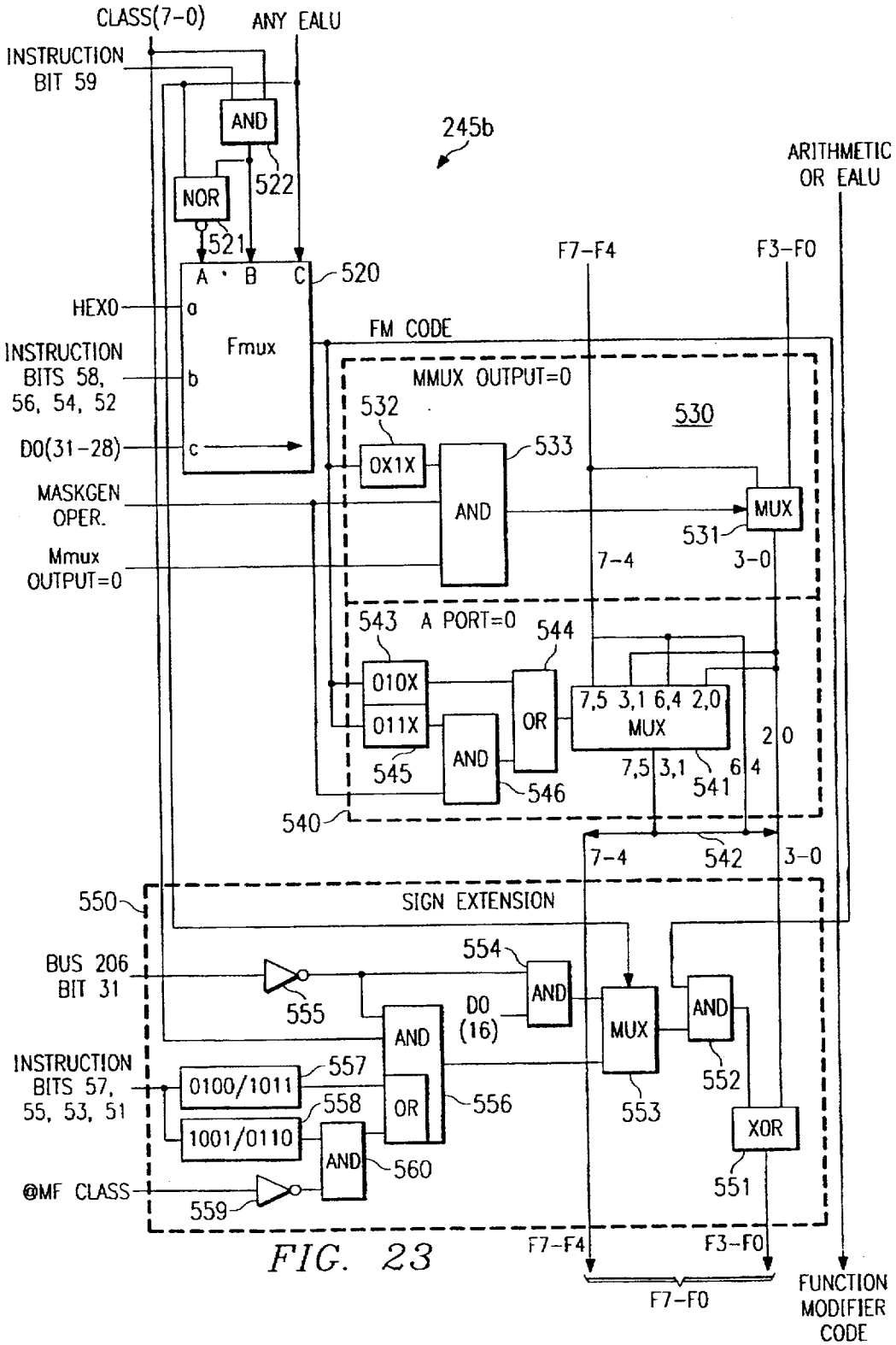

CLASS(7-0)
ANY EALU
INSTRUCTION BIT 59
AND
522
NOR
521
245b
ARITHMETIC OR EALU
F7-F4
F3-F0
A B C
520
HEX0
a
Fmux
FM CODE
INSTRUCTION BITS 58, 56, 54, 52
b
MMUX OUTPUT=0
532
530
D0(31-28)
c
0X1X
533
AND
MASKGEN OPER.
MUX
531
Mmux OUTPUT=0
7-4
3-0
543
A PORT=0
544
010X
011X
OR
7,5 3,1 6,4 2,0
MUX
2 0
545
AND
546
7,5 3,1
541
6 4
540
7-4
542
3-0
550
SIGN EXTENSION
554
BUS 206 BIT 31
555
AND
D0 (16)
AND
MUX
AND
INSTRUCTION BITS 57, 55, 53, 51
0100/1011
557
558
OR
556
552
553
1001/0110
AND
560
XOR
@MF CLASS
559
551
F7-F4
F3-F0
F7-F0
FUNCTION MODIFIER CODE

FIG. 23

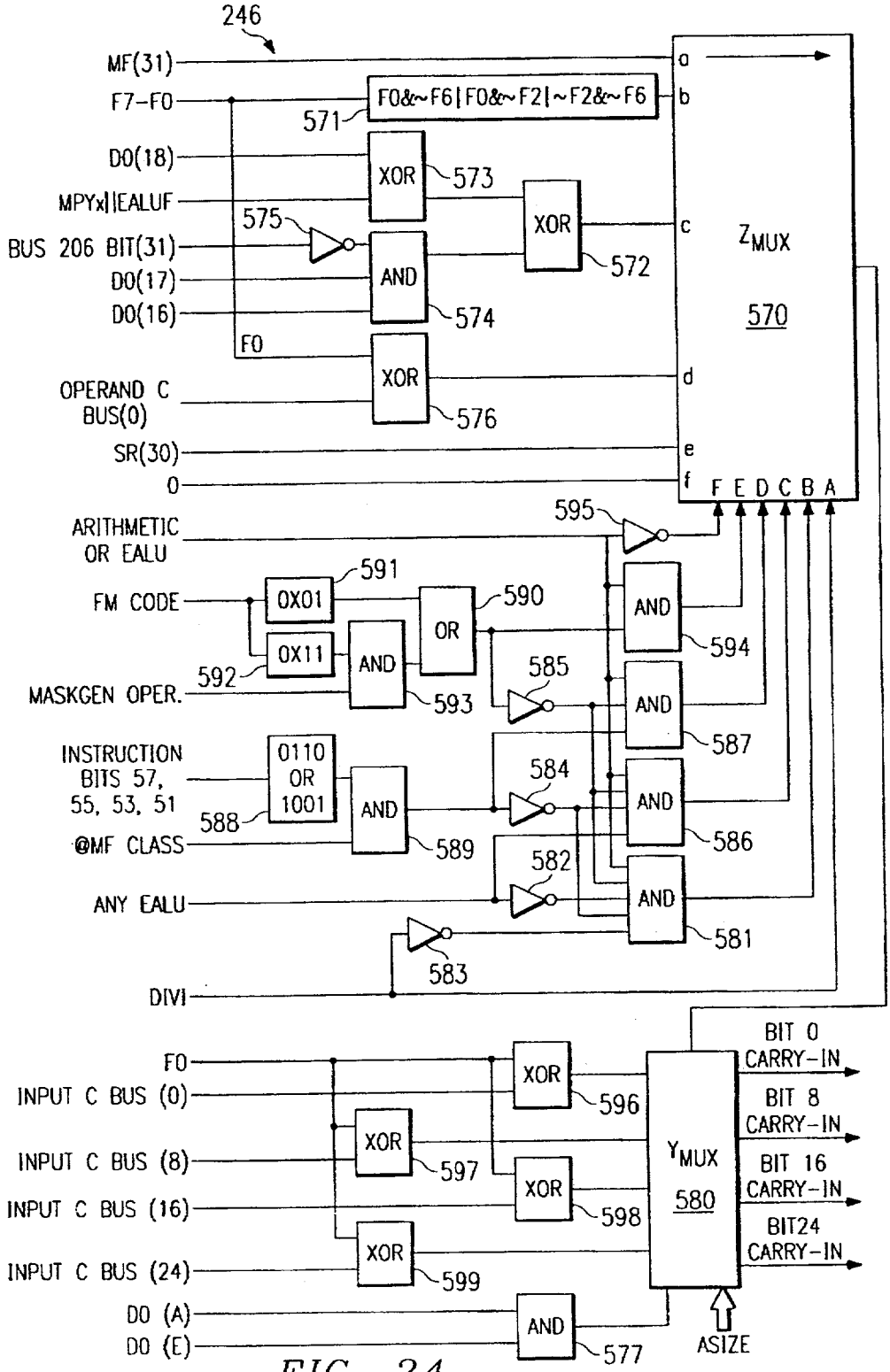

246
MF(31)
F7-F0
F0&~F6|F0&~F2|~F2&~F6
571
D0(18)
XOR
573
MPYx||EALUF
575
XOR
BUS 206 BIT(31)
AND
572
D0(17)
D0(16)
574
F0
XOR
OPERAND C BUS(0)
576
SR(30)
0
a
b
c
d
e
f
ZMUX
570
F E D C B A
595
ARITHMETIC OR EALU
591
FM CODE
0X01
590
OR
AND
594
0X11
592
MASKGEN OPER.
AND
593
585
AND
587
INSTRUCTION BITS 57, 55, 53, 51
0110 OR 1001
584
AND
588
@MF CLASS
AND
589
586
582
ANY EALU
AND
581
583
DIVI
F0
INPUT C BUS (0)
XOR
596
INPUT C BUS (8)
XOR
597
INPUT C BUS (16)
XOR
598
INPUT C BUS (24)
XOR
599
YMUX
580
BIT 0 CARRY-IN
BIT 8 CARRY-IN
BIT 16 CARRY-IN
BIT24 CARRY-IN
D0 (A)
D0 (E)
AND
577
ASIZE

FIG. 24

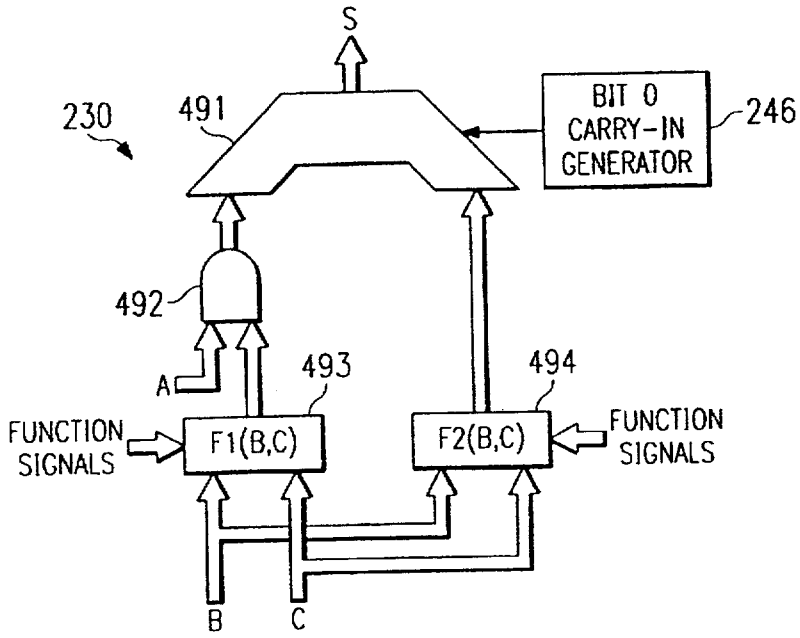

S
230
491
BIT 0
CARRY-IN
GENERATOR
246
492
A
493
494
FUNCTION
SIGNALS
F1(B,C)
F2(B,C)
FUNCTION
SIGNALS
B
C

FIG. 25

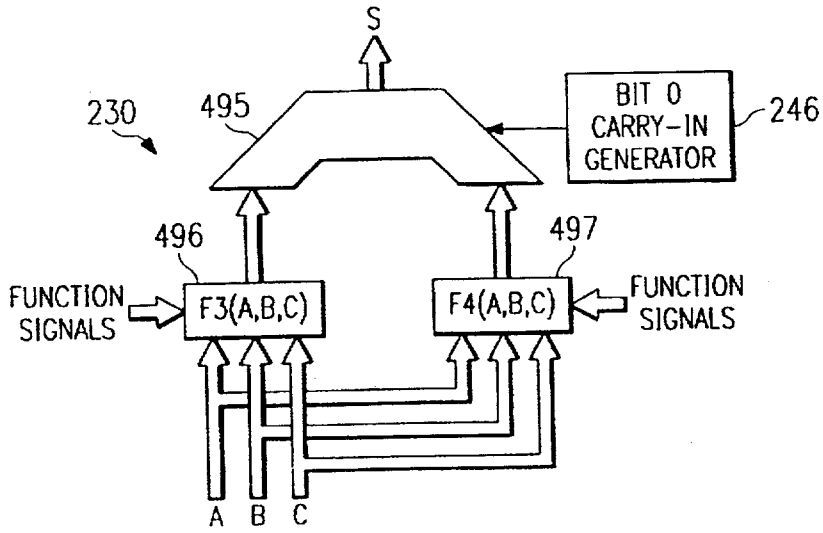

S
230
495
BIT 0
CARRY-IN
GENERATOR
246
496
497
FUNCTION
SIGNALS
F3(A,B,C)
F4(A,B,C)
FUNCTION
SIGNALS
A
B
C

FIG. 26

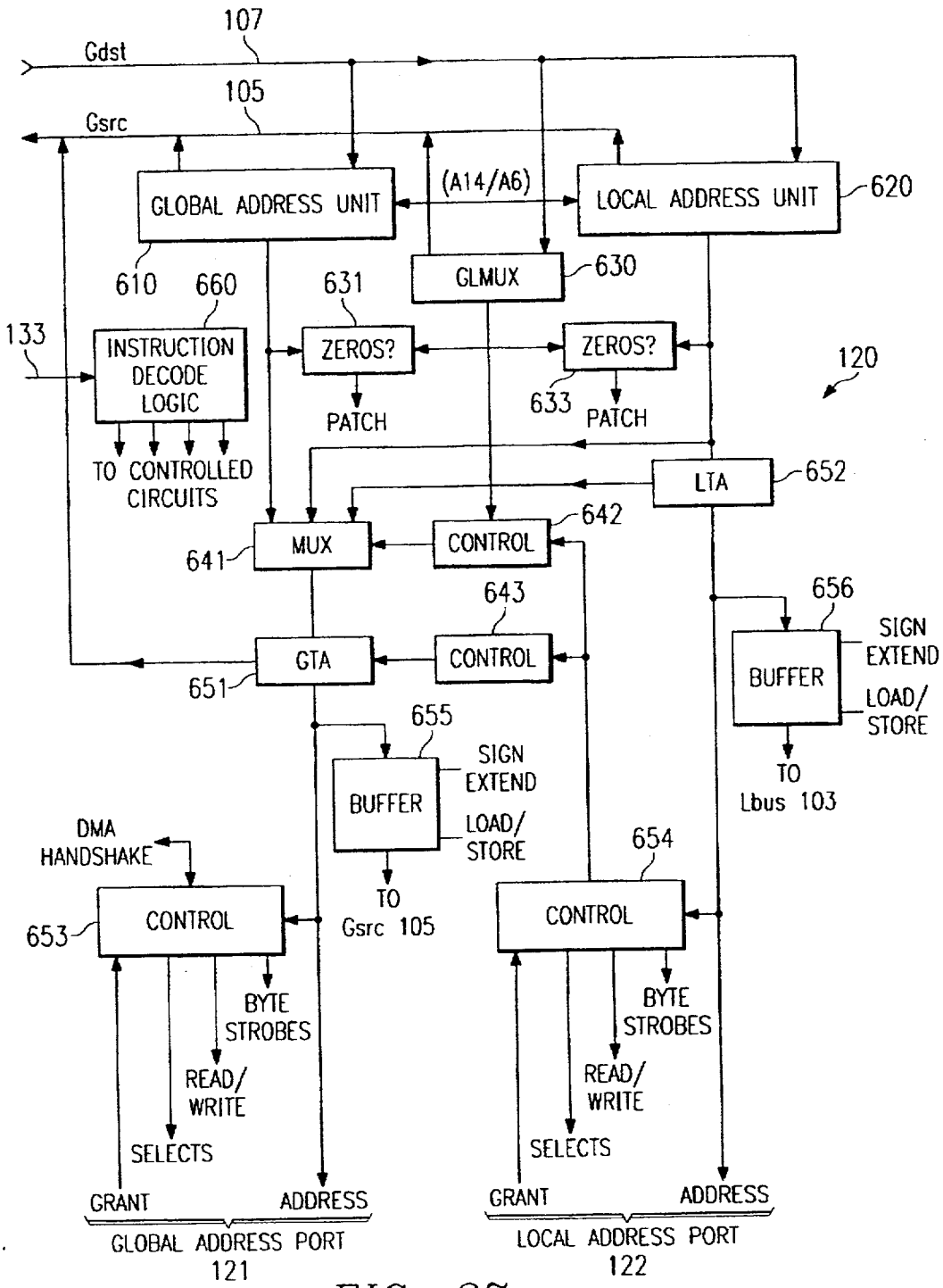

107
Gdst
105
Gsrc
GLOBAL ADDRESS UNIT
(A14/A6)
LOCAL ADDRESS UNIT
620
610
660
631
GLMUX
630
133
INSTRUCTION DECODE LOGIC
ZEROS?
ZEROS?
120
633
PATCH
PATCH
TO CONTROLLED CIRCUITS
LTA
652
642
MUX
CONTROL
641
643
656
GTA
CONTROL
BUFFER
SIGN EXTEND
LOAD/ STORE
651
655
BUFFER
SIGN EXTEND
LOAD/ STORE
TO Lbus 103
DMA HANDSHAKE
654
TO Gsrc 105
CONTROL
CONTROL
653
BYTE STROBES
READ/ WRITE
SELECTS
BYTE STROBES
READ/ WRITE
SELECTS
GRANT
ADDRESS
GRANT
ADDRESS
GLOBAL ADDRESS PORT 121
LOCAL ADDRESS PORT 122

FIG. 27

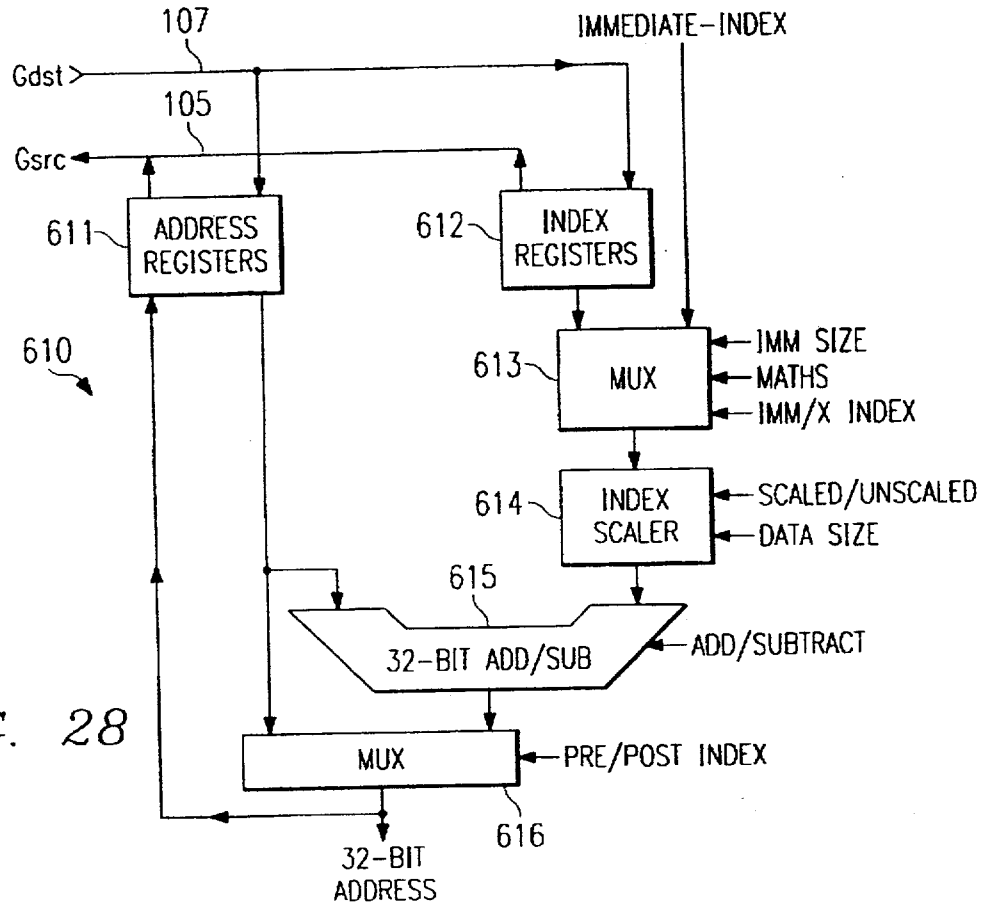


FIG. 28

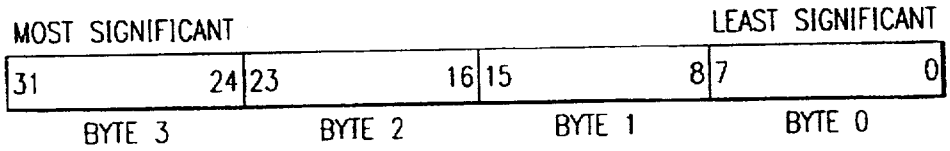


FIG. 29a

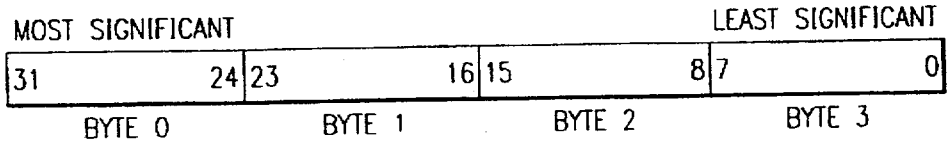


FIG. 29b

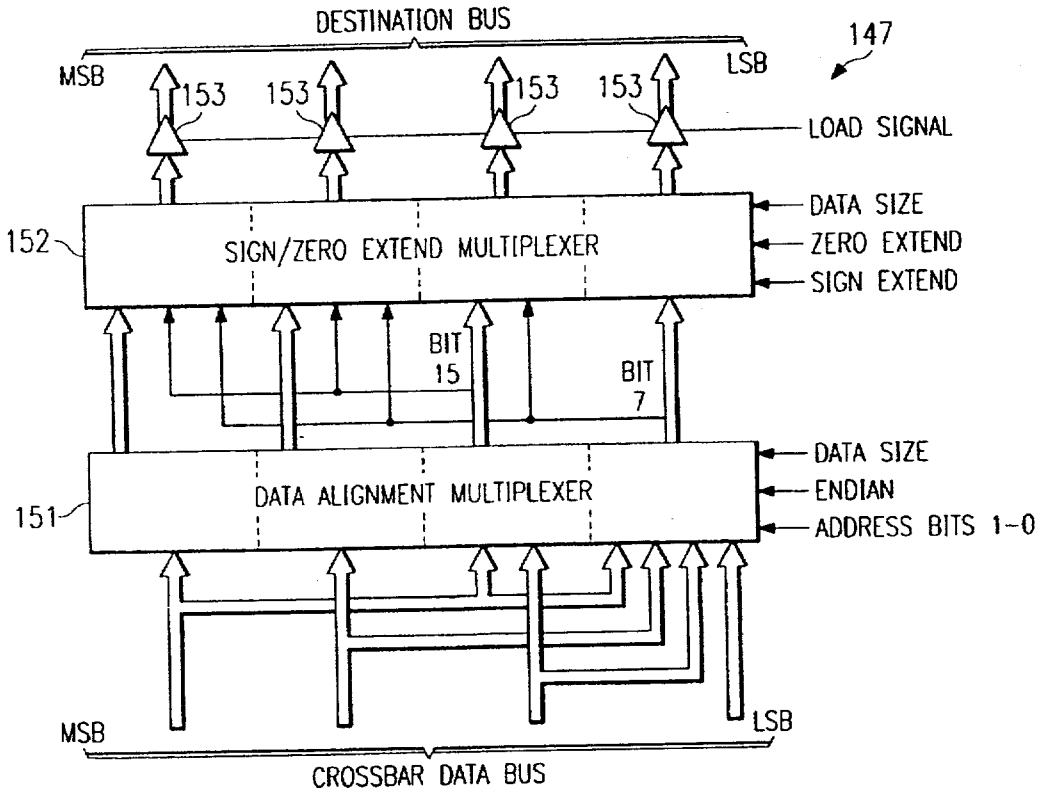

DESTINATION BUS
MSB
LSB
147
153
153
153
153
LOAD SIGNAL
152
SIGN/ZERO EXTEND MULTIPLEXER
DATA SIZE
ZERO EXTEND
SIGN EXTEND
BIT 15
BIT 7
151
DATA ALIGNMENT MULTIPLEXER
DATA SIZE
ENDIAN
ADDRESS BITS 1-0
MSB
LSB
CROSSBAR DATA BUS

FIG. 30

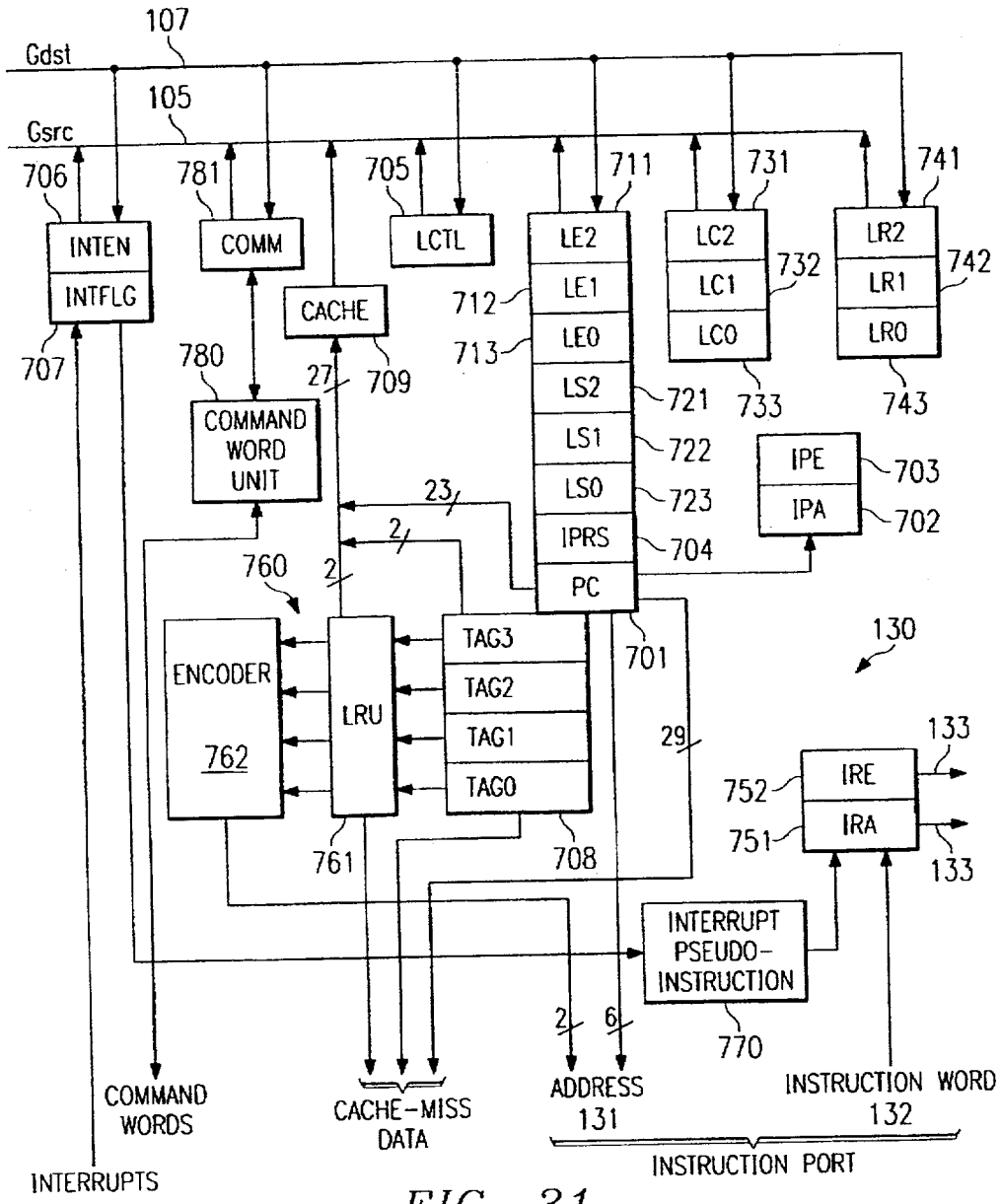

107
Gdst
105
Gsrc
706
781
705
711
731
741
INTEN
INTFLG
COMM
LCTL
LE2
LE1
LE0
LS2
LS1
LS0
IPRS
PC
LC2
LC1
LC0
LR2
LR1
LR0
CACHE
712
713
732
742
707
780
27
709
721
733
743
COMMAND WORD UNIT
722
IPE
IPA
703
702
23
723
704
2
760
2
130
ENCODER
762
LRU
TAG3
TAG2
TAG1
TAG0
701
29
133
752
IRE
751
IRA
133
761
708
INTERRUPT PSEUDO-INSTRUCTION
2
6
770
COMMAND WORDS
CACHE-MISS DATA
ADDRESS 131
INSTRUCTION WORD 132
INSTRUCTION PORT
INTERRUPTS

FIG. 31

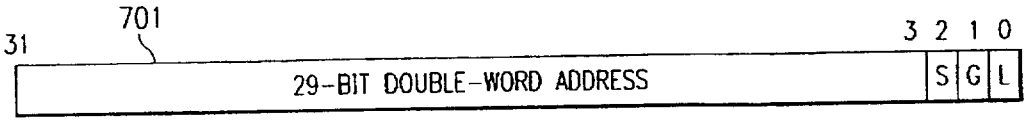


FIG. 32

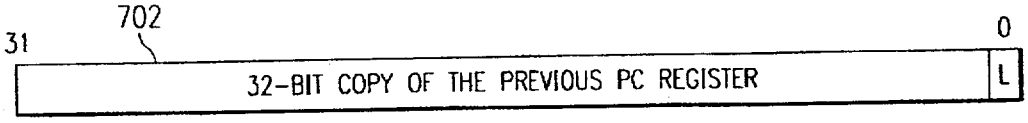


FIG. 33

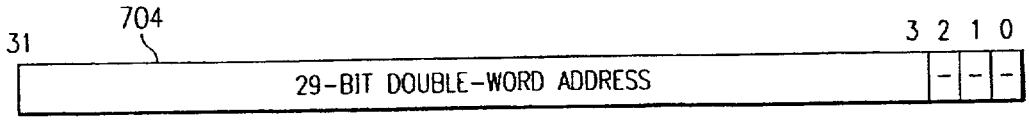


FIG. 34

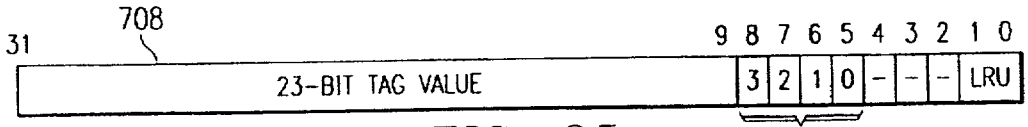


FIG. 35 SUB-BLOCK PRESENT BITS

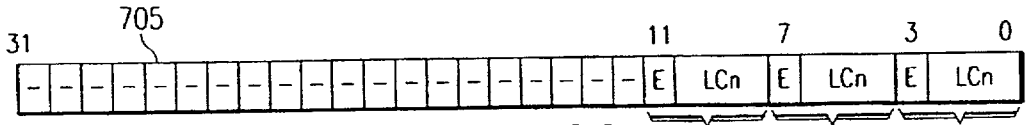


FIG. 36

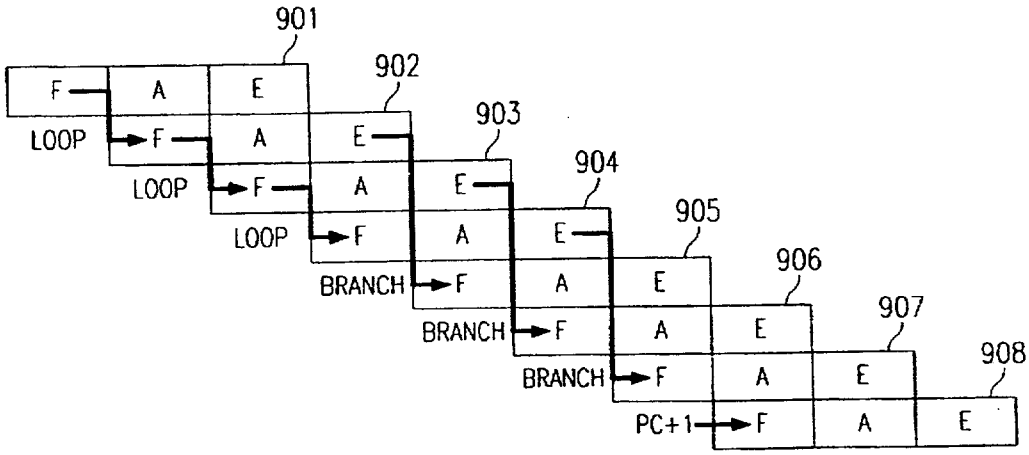


FIG. 39

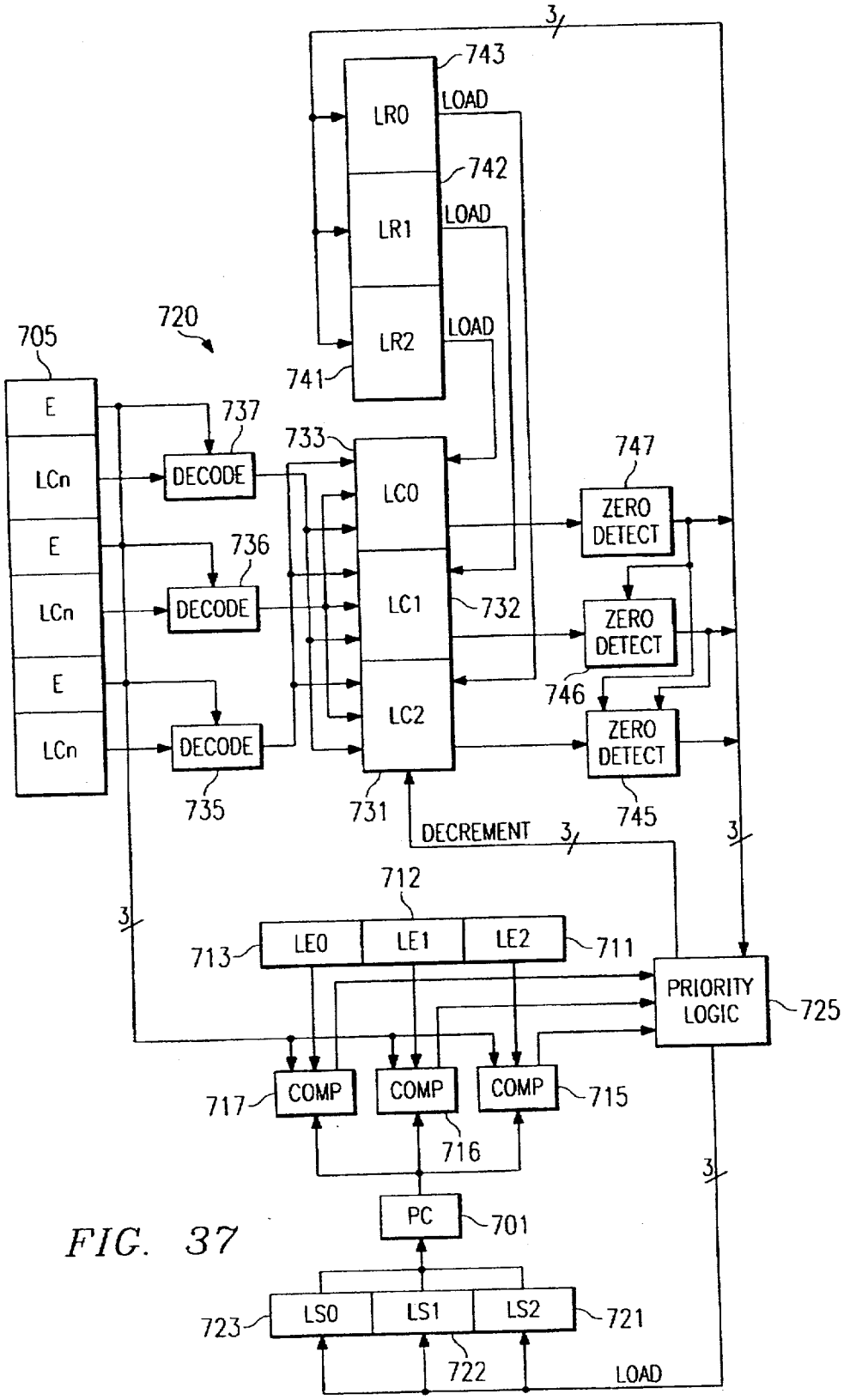

3
743
LR0
LOAD
742
LR1
LOAD
LR2
LOAD
741
720
705
E
LCn
E
LCn
E
LCn
737
DECODE
736
DECODE
DECODE
735
733
LC0
LC1
LC2
731
732
747
ZERO DETECT
ZERO DETECT
ZERO DETECT
746
745
DECREMENT
3
712
713
LE0
LE1
LE2
711
PRIORITY LOGIC
725
COMP
COMP
COMP
717
716
715
PC
701
LS0
LS1
LS2
723
722
721
LOAD

FIG. 37

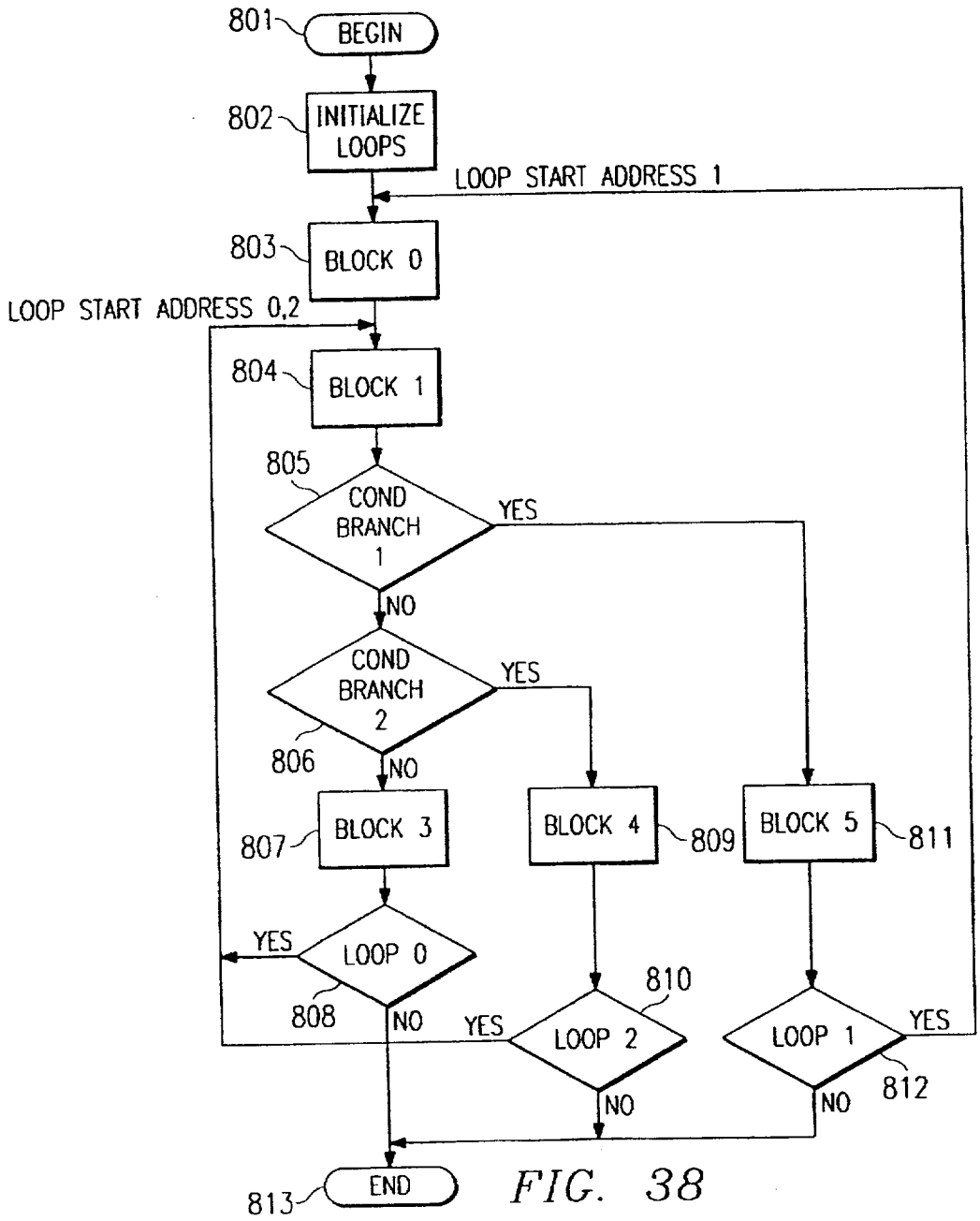

801
BEGIN
802
INITIALIZE LOOPS
LOOP START ADDRESS 1
803
BLOCK 0
LOOP START ADDRESS 0,2
804
BLOCK 1
805
COND BRANCH 1
YES
NO
COND BRANCH 2
YES
806
NO
807
BLOCK 3
BLOCK 4
809
BLOCK 5
811
YES
LOOP 0
808
NO
YES
LOOP 2
810
LOOP 1
YES
812
NO
NO
813
END

FIG. 38

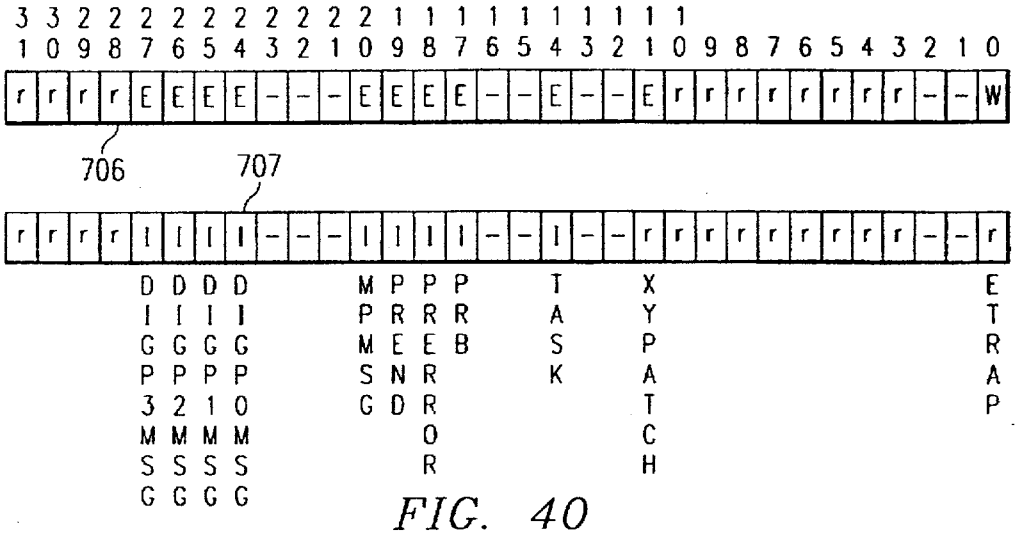


FIG. 40

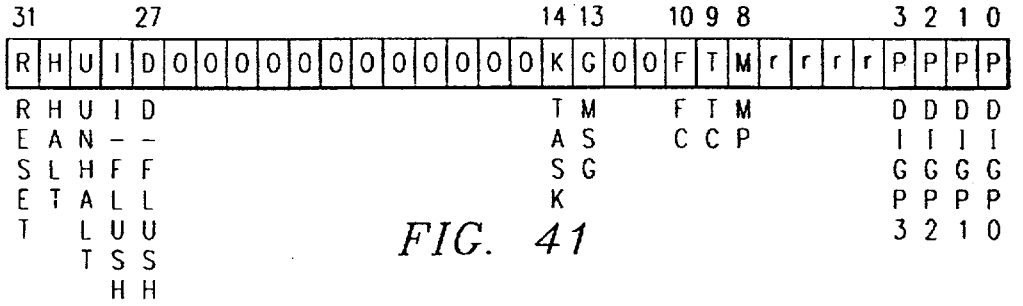


FIG. 41

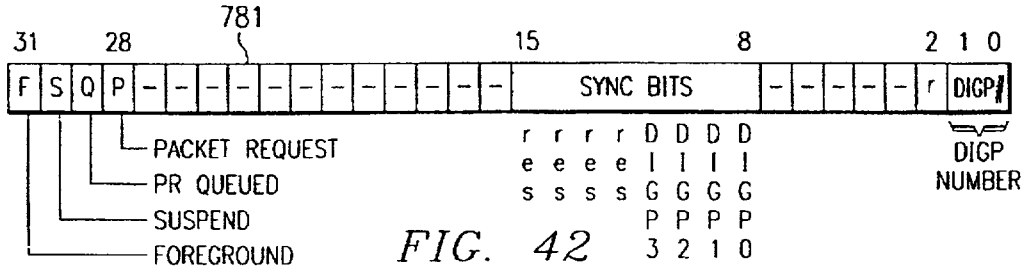


FIG. 42

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	0	DATA UNIT FORMAT
1	Class			A	8-bit ALU code								dst			src1			0	immed.				Parallel Transfers										A
																			1 0	-	scr2			Parallel Transfers										B
																			1 1	dstbank				s1bnk	cond.			32-bit imm.						C
1	-	-	-	1	-	0	-	0	-	0	-	0	-	-	-	-	-	-	0	operation				Parallel Transfers										D
0	1	1	oper				src3			dst2			dst1			src1			src4			src2		Parallel Transfers										E
0	0		Reserved																															
0	1	0	Reserved																															

38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	DATA TRANSFER FORMAT
Lmode				d			e	siz		s	La			Gim/x			L	Obank			L	Gmode			reg			e	siz		s	Ga			Lim/x			1
Lmode				o	a	n	0	0	0	0	La			Gim/x			bank				L	Gmode			reg			e	siz		s	Ga			Lim/x			2
Lmode				d			e	siz		s	La			0	Lrm		dstbank				L	0	0	0	0	src		srcbank				dst			Lim/x			3
Lmode				d			e	siz		s	La			0	itm		dstbank				L	0	0	0	1	src		e	siz		D	dst			Lim/x			4
Lmode				reg			e	siz		s	La			1	Lrm		bank				L	0	0	Local Long Offset/x														5
0	0	Global Long Offset/x															bank				L	Gmode			reg			e	siz		s	Ga			0	Grm		6
Lmode				d			e	siz		s	La			0	Lrm		Adstbank				L	0	0	1	-	-	-	-	As1bank			-	-	-	Lim/x			7
0	0	-		cond.			c	r	g	N	C	V	Z	0	-	-	dstbank				-	0	0	0	0	src		srcbank				dst			-	-	-	8
0	0	-		cond.			c	r	g	N	C	V	Z	0	itm		dstbank				-	0	0	0	1	src		e	siz		D	dst			-	-	-	9
0	0	-		cond.			c	r	g	N	C	V	Z	Gim/x			bank				L	Gmode			reg			e	siz		s	Ga			1	Grm		10
0	0	-		cond.			c	r	-	N	C	V	Z	0	-	-	Adstbank				-	0	0	1	-	-	-	-	As1bank			-	-	-	-	-	R	11

FIG. 43

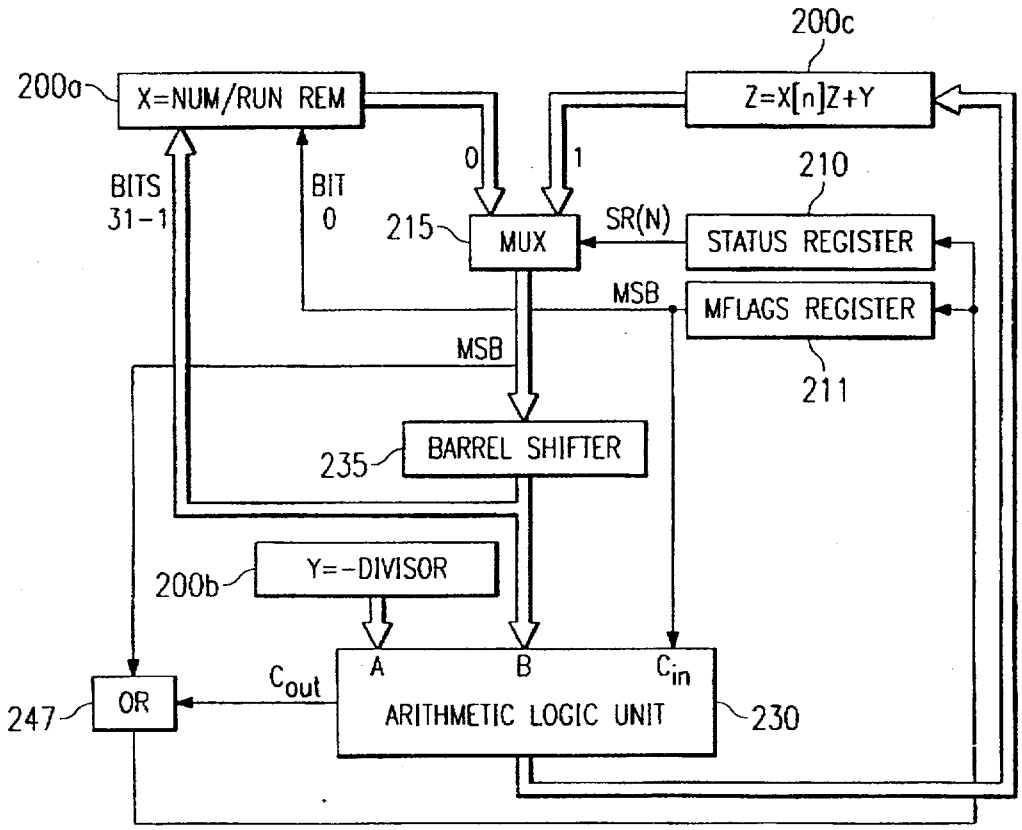

200c
200a
X=NUM/RUN REM
Z=X[n]Z+Y
0
1
210
BITS
31-1
BIT
0
215
MUX
SR(N)
STATUS REGISTER
MSB
MFLAGS REGISTER
MSB
211
235
BARREL SHIFTER
200b
Y=-DIVISOR
A
B
C_{in}
C_{out}
247
OR
ARITHMETIC LOGIC UNIT
230

FIG. 44

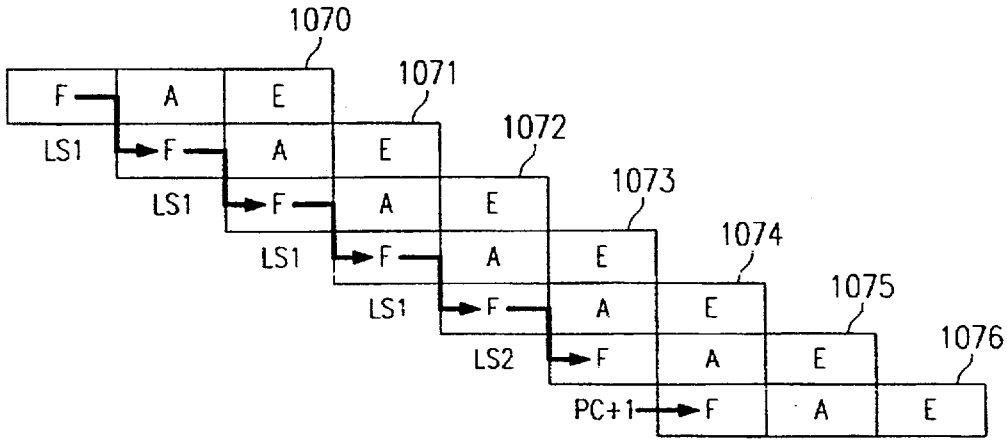

1070
1071
1072
1073
1074
1075
1076
F
A
E
LS1
LS2
PC+1

FIG. 49

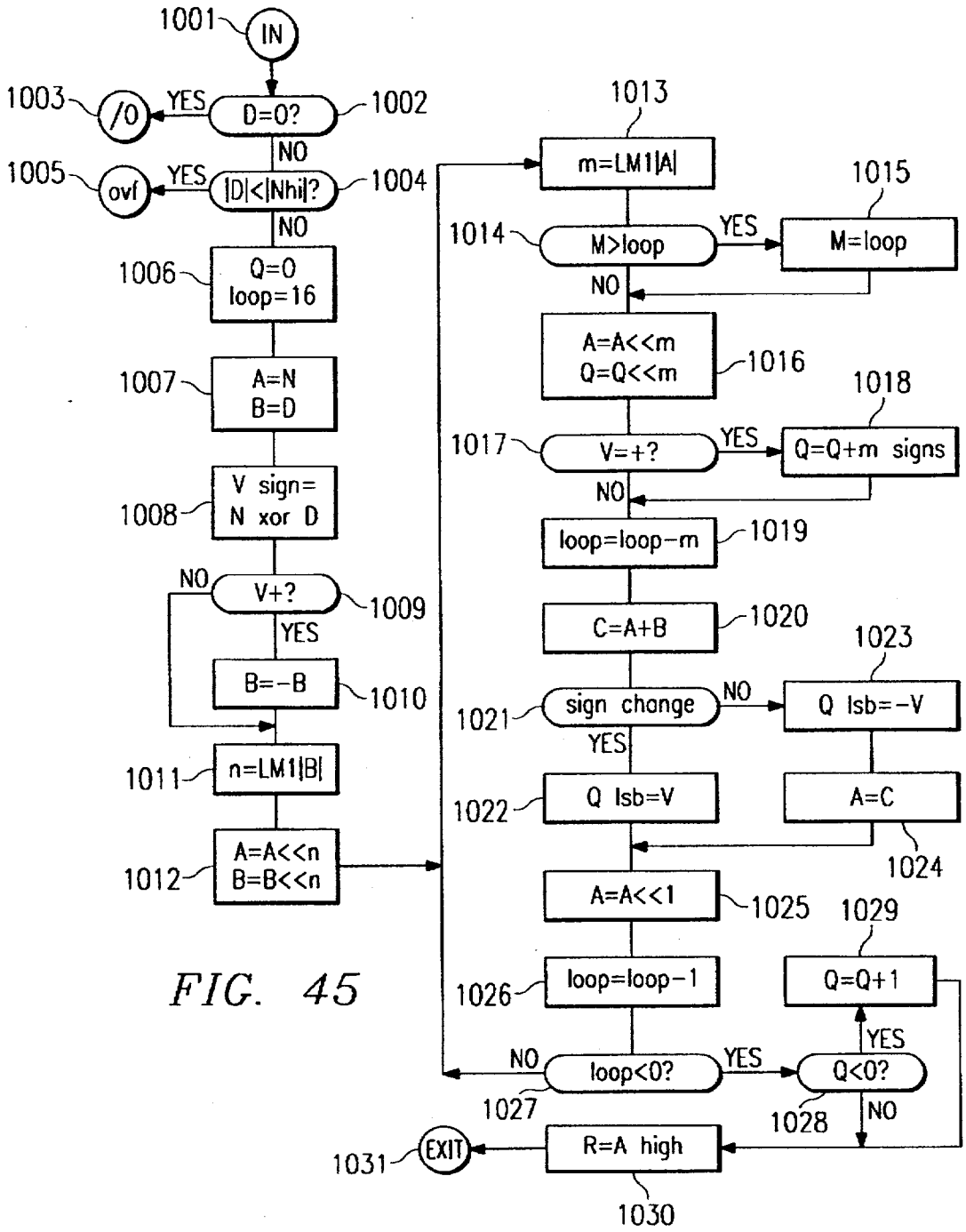

1001
IN
1003
/0
YES
D=0?
1002
NO
1005
ovf
YES
|D|<|Nhi|?
1004
NO
1006
Q=0
loop=16
1007
A=N
B=D
1008
V sign=
N xor D
NO
V+?
1009
YES
B=-B
1010
1011
n=LM1|B|
1012
A=A<<n
B=B<<n
1013
m=LM1|A|
1014
M>loop
YES
1015
M=loop
NO
A=A<<m
Q=Q<<m
1016
1017
V=+?
YES
1018
Q=Q+m signs
NO
loop=loop-m
1019
C=A+B
1020
1021
sign change
NO
1023
Q lsb=-V
YES
1022
Q lsb=V
A=C
1024
A=A<<1
1025
1026
loop=loop-1
1029
Q=Q+1
YES
NO
loop<0?
YES
Q<0?
1027
1028
NO
1031
EXIT
R=A high
1030

FIG. 45

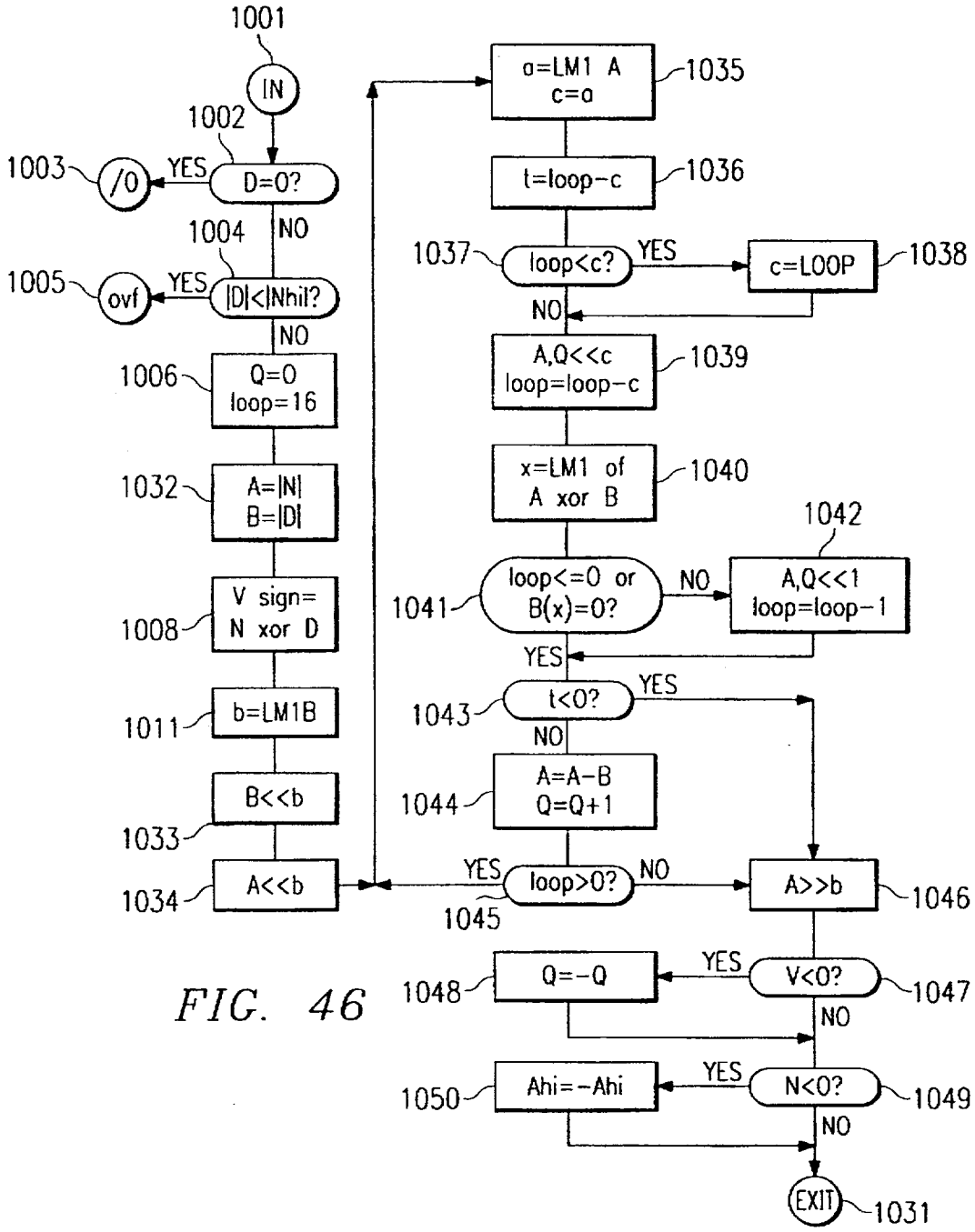

1001
IN
1002
D=0?
YES
1003
/0
NO
1004
|D|<|Nhi|?
YES
1005
ovf
NO
1006
Q=0
loop=16
1032
A=|N|
B=|D|
1008
V sign=
N xor D
1011
b=LM1B
B<<b
1033
A<<b
1034
a=LM1 A
c=a
1035
t=loop−c
1036
1037
loop<c?
YES
c=LOOP
1038
NO
A,Q<<c
loop=loop−c
1039
x=LM1 of
A xor B
1040
1042
1041
loop<=0 or
B(x)=0?
NO
A,Q<<1
loop=loop−1
YES
1043
t<0?
YES
NO
1044
A=A−B
Q=Q+1
YES
loop>0?
NO
1045
A>>b
1046
1048
Q=−Q
YES
V<0?
1047
NO
1050
Ahi=−Ahi
YES
N<0?
1049
NO
EXIT
1031

FIG. 46

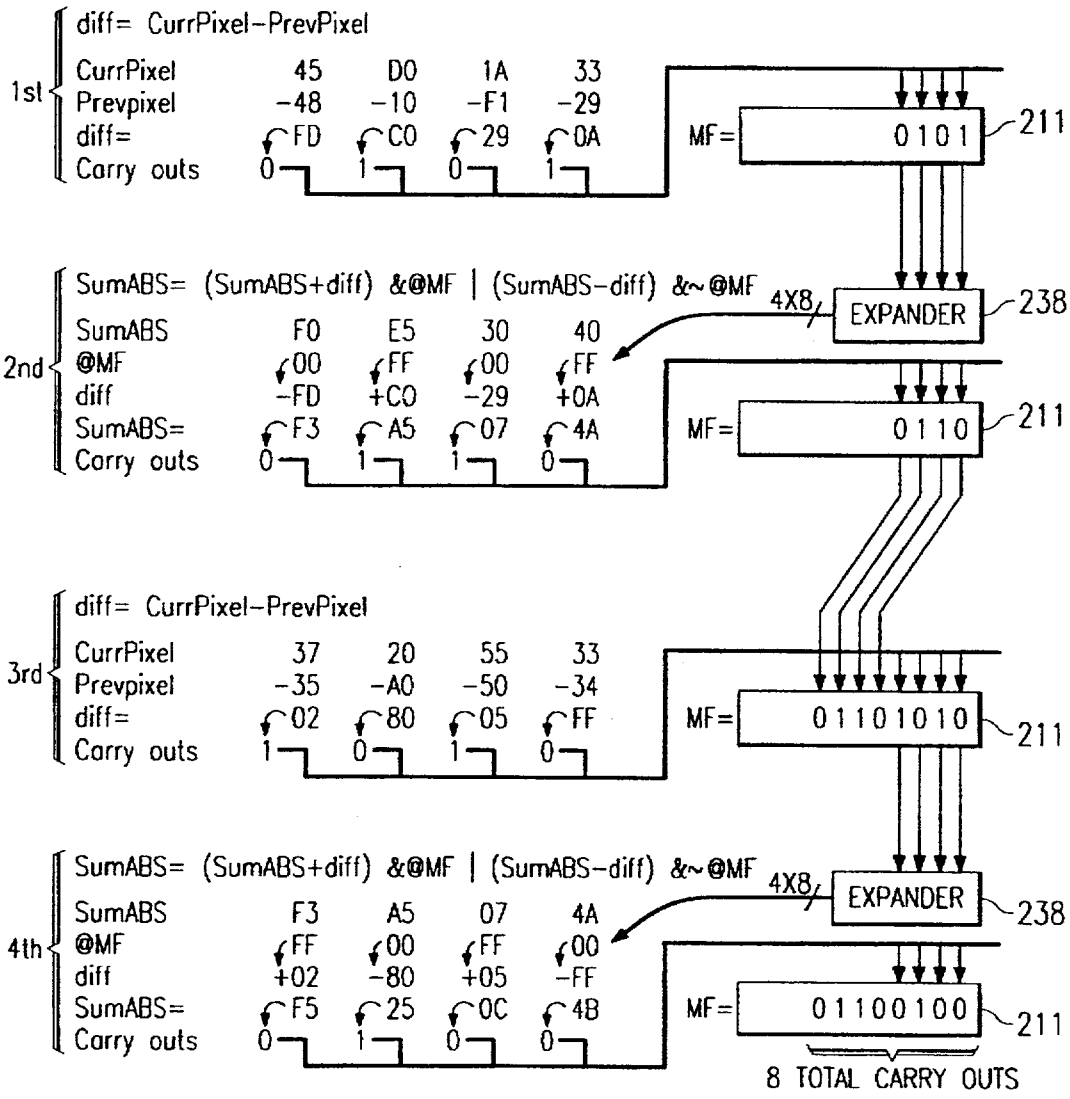

1st
diff= CurrPixel-PrevPixel
CurrPixel 45 D0 1A 33
Prevpixel -48 -10 -F1 -29
diff= FD C0 29 0A
Carry outs 0 1 0 1
MF= 0101
211
2nd
SumABS= (SumABS+diff) &@MF | (SumABS-diff) &~@MF
4X8
EXPANDER
238
SumABS F0 E5 30 40
@MF 00 FF 00 FF
diff -FD +C0 -29 +0A
SumABS= F3 A5 07 4A
Carry outs 0 1 1 0
MF= 0110
211
3rd
diff= CurrPixel-PrevPixel
CurrPixel 37 20 55 33
Prevpixel -35 -A0 -50 -34
diff= 02 80 05 FF
Carry outs 1 0 1 0
MF= 01101010
211
4th
SumABS= (SumABS+diff) &@MF | (SumABS-diff) &~@MF
4X8
EXPANDER
238
SumABS F3 A5 07 4A
@MF FF 00 FF 00
diff +02 -80 +05 -FF
SumABS= F5 25 0C 4B
Carry outs 0 1 0 0
MF= 01100100
211
8 TOTAL CARRY OUTS

FIG. 47

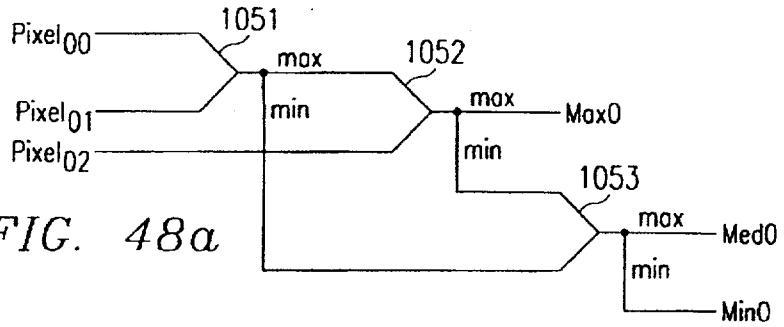


FIG. 48a

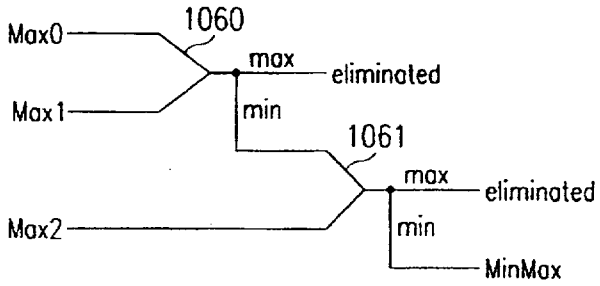


FIG. 48b

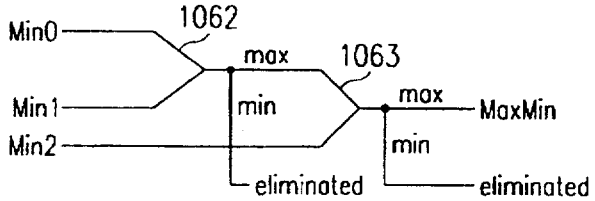


FIG. 48c

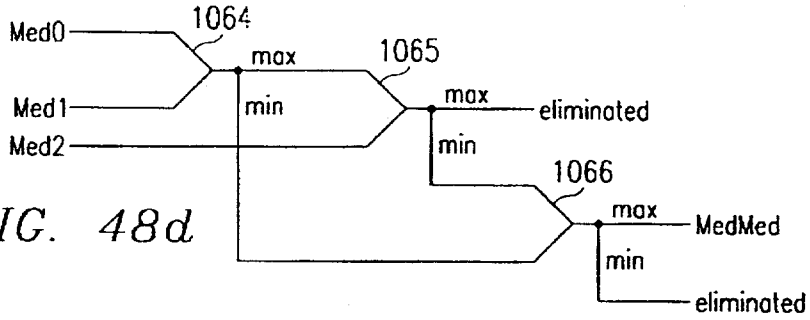


FIG. 48d

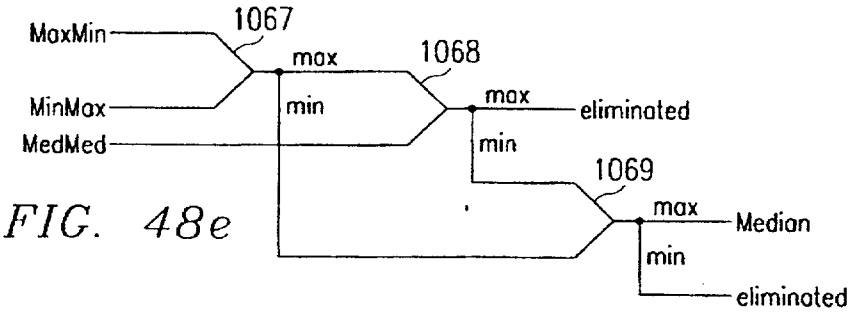


FIG. 48e

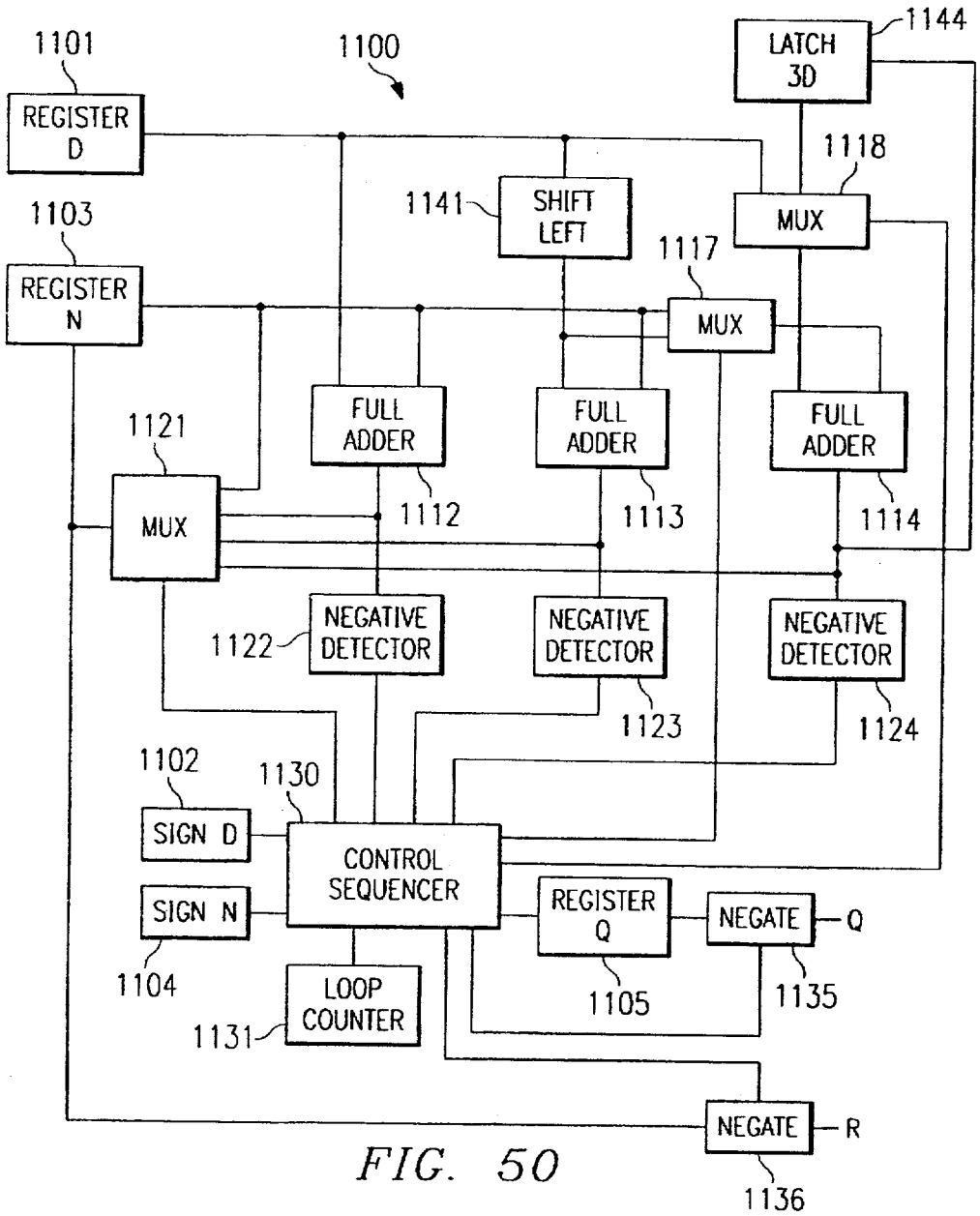

1101
REGISTER D
1100
LATCH 3D
1144
1118
MUX
1141
SHIFT LEFT
1103
REGISTER N
1117
MUX
FULL ADDER
FULL ADDER
FULL ADDER
1121
MUX
1112
1113
1114
NEGATIVE DETECTOR
NEGATIVE DETECTOR
NEGATIVE DETECTOR
1122
1123
1124
1102
1130
SIGN D
CONTROL SEQUENCER
SIGN N
REGISTER Q
NEGATE
Q
1104
LOOP COUNTER
1105
1135
1131
NEGATE
R
1136

FIG. 50

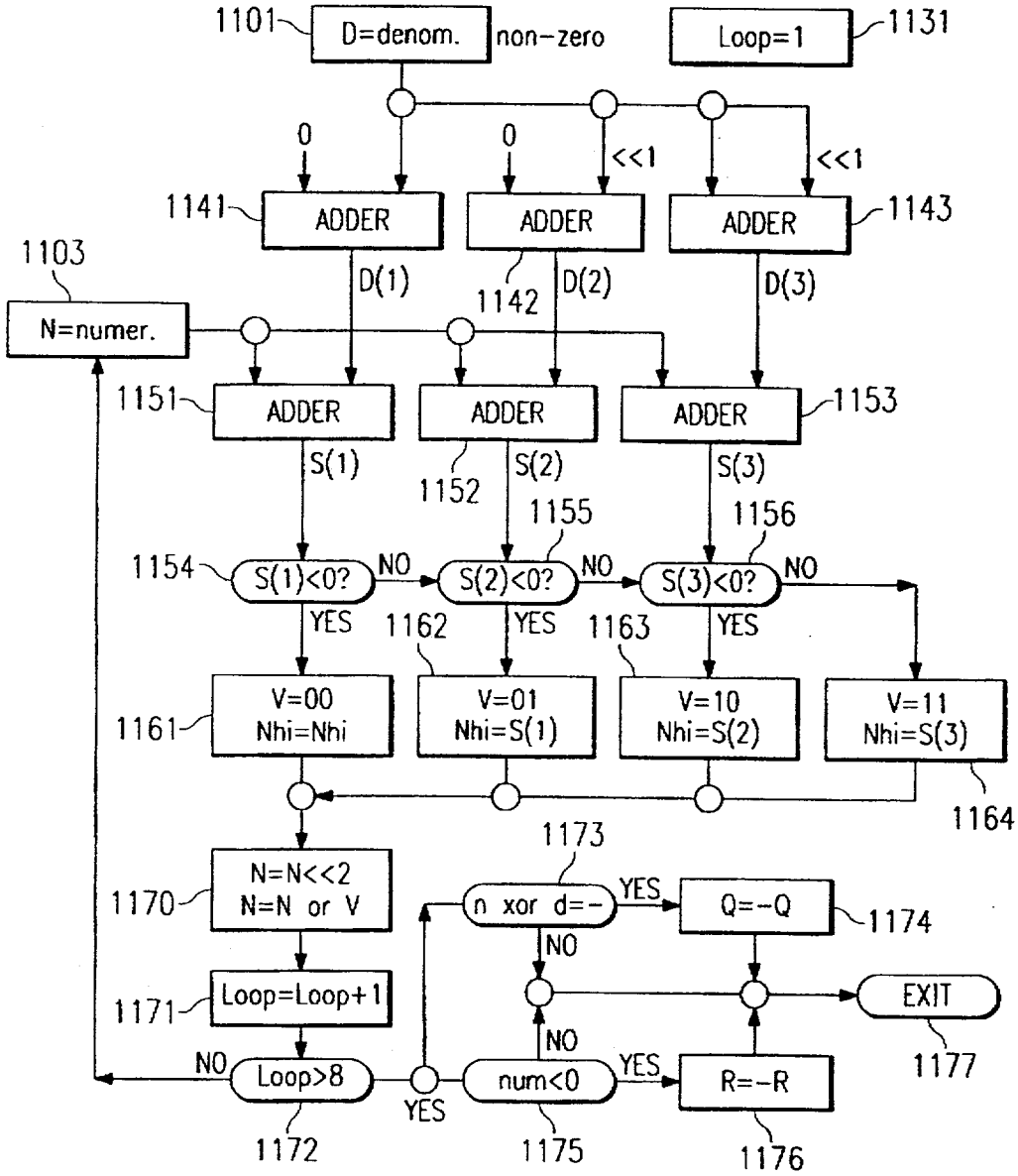

1101
D=denom.
non-zero
Loop=1
1131
0
0
<<1
<<1
1141
ADDER
ADDER
ADDER
1143
1103
D(1)
1142
D(2)
D(3)
N=numer.
1151
ADDER
ADDER
ADDER
1153
S(1)
1152
S(2)
1155
S(3)
1156
1154
S(1)<0?
NO
S(2)<0?
NO
S(3)<0?
NO
YES
1162
YES
1163
YES
1161
V=00
Nhi=Nhi
V=01
Nhi=S(1)
V=10
Nhi=S(2)
V=11
Nhi=S(3)
1164
1173
1170
N=N<<2
N=N or V
n xor d=-
YES
Q=-Q
1174
NO
1171
Loop=Loop+1
EXIT
NO
NO
Loop>8
num<0
YES
R=-R
1177
YES
1172
1175
1176

FIG. 51

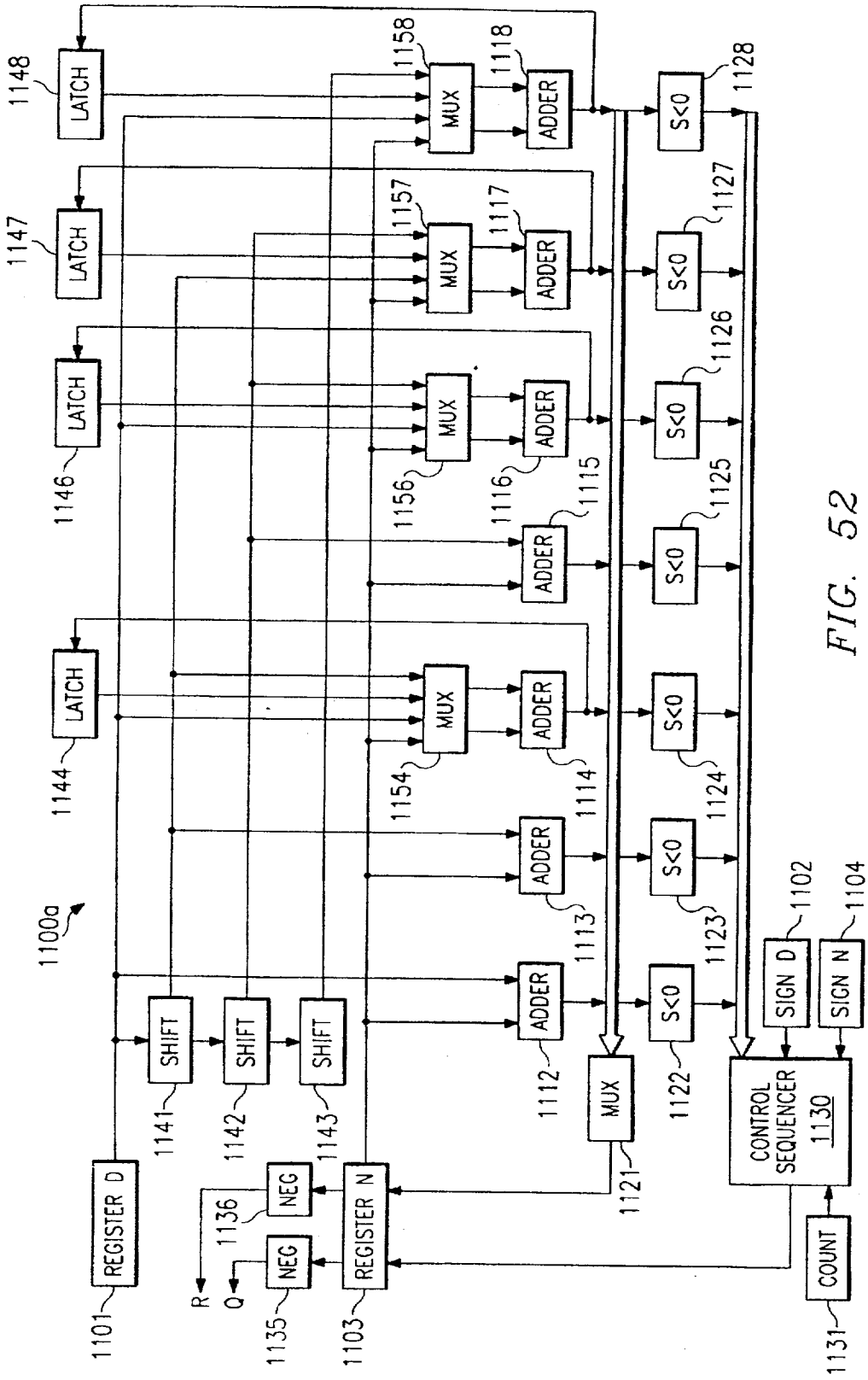

LATCH
1148
1147
1146
1144
1100a
MUX
1158
1157
1156
1154
ADDER
1118
1117
1116
1115
1114
1113
1112
S<0
1128
1127
1126
1125
1124
1123
1122
SHIFT
1141
1142
1143
MUX
1121
REGISTER D
1101
NEG
1136
1135
R
Q
REGISTER N
1103
CONTROL SEQUENCER
1130
SIGN D
1102
SIGN N
1104
COUNT
1131

FIG. 52

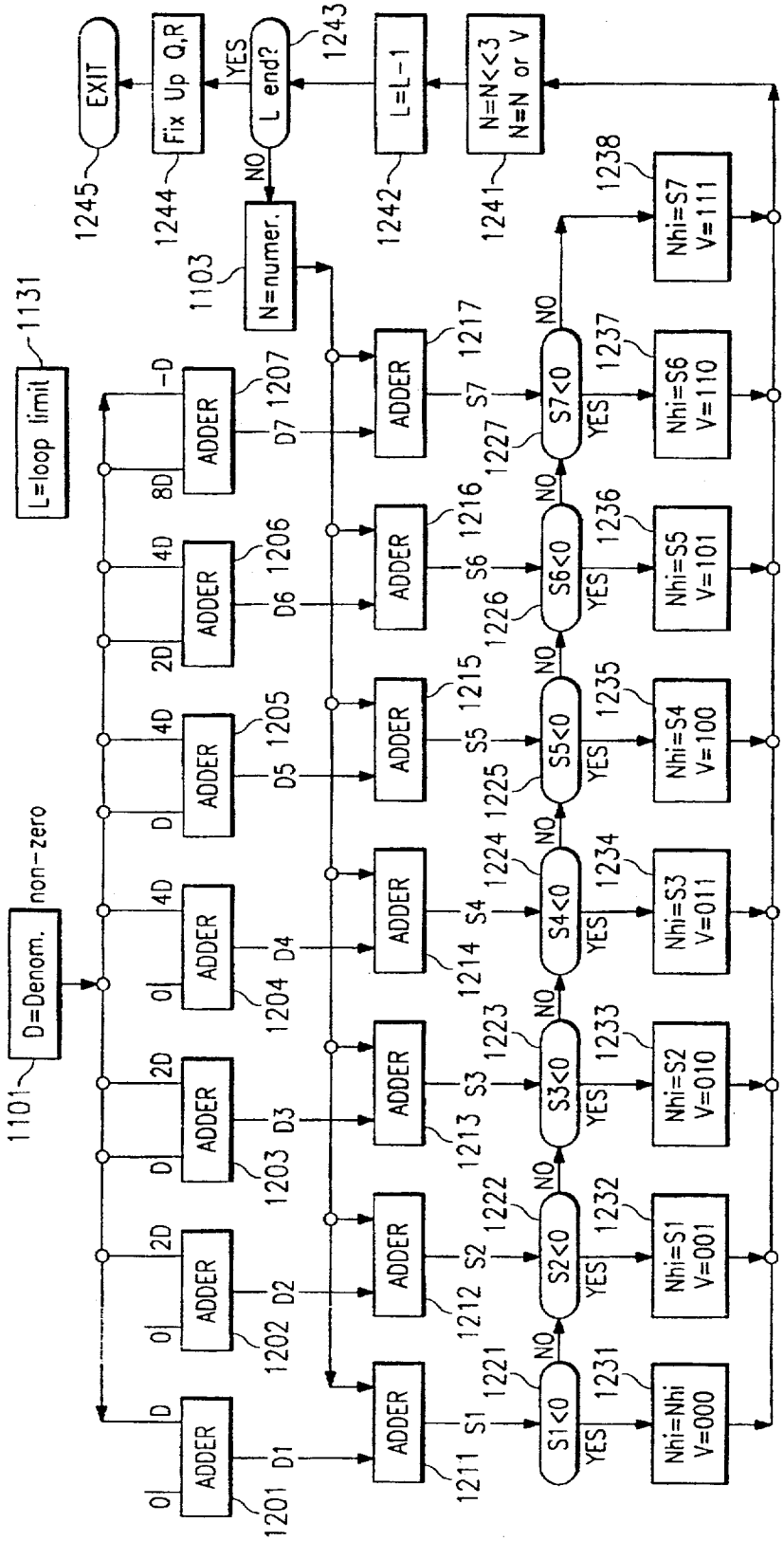

D=Denom.
non-zero
L=loop limit
N=numer.
L end?
YES
NO
L=L-1
N=N<<3
N=N or V
Fix Up Q,R
EXIT
ADDER
S1<0
S2<0
S3<0
S4<0
S5<0
S6<0
S7<0
Nhi=Nhi
V=000
Nhi=S1
V=001
Nhi=S2
V=010
Nhi=S3
V=011
Nhi=S4
V=100
Nhi=S5
V=101
Nhi=S6
V=110
Nhi=S7
V=111

FIG. 53

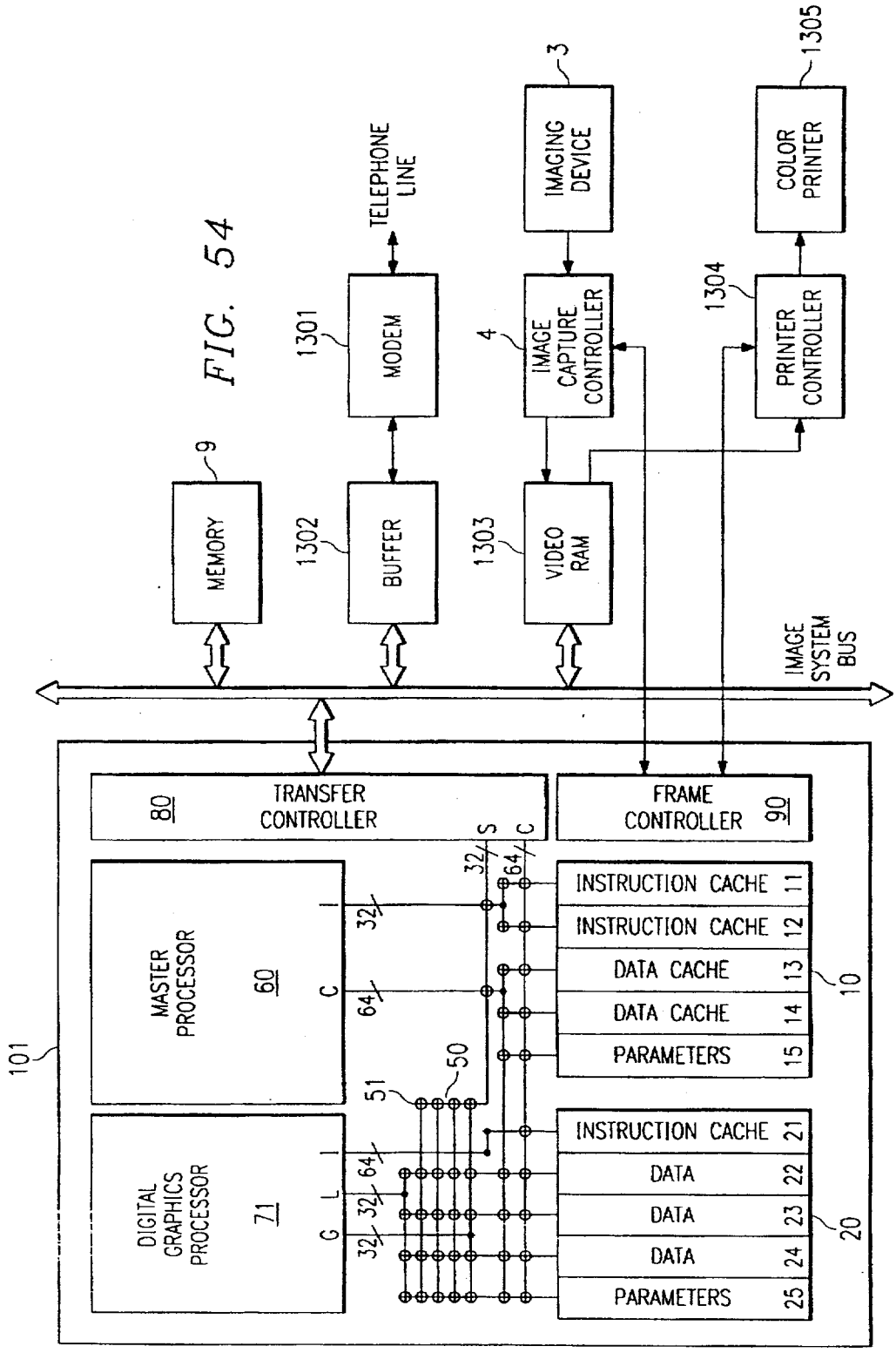

TELEPHONE LINE
MODEM
1301
BUFFER
1302
MEMORY
9
IMAGING DEVICE
3
IMAGE CAPTURE CONTROLLER
4
VIDEO RAM
1303
COLOR PRINTER
1305
PRINTER CONTROLLER
1304
IMAGE SYSTEM BUS
TRANSFER CONTROLLER
80
FRAME CONTROLLER
90
MASTER PROCESSOR
60
DIGITAL GRAPHICS PROCESSOR
71
101
INSTRUCTION CACHE 11
INSTRUCTION CACHE 12
DATA CACHE 13
DATA CACHE 14
PARAMETERS 15
10
INSTRUCTION CACHE 21
DATA 22
DATA 23
DATA 24
PARAMETERS 25
20
50
51
32
64
S
C
I
L
G

FIG. 54

LONG INSTRUCTION WORD CONTROLLING PLURAL INDEPENDENT PROCESSOR OPERATIONS

CROSS REFERENCE RELATED APPLICATIONS

This application is a divisional of U.S. patent application Ser. No. 08/160,297 filed Nov. 30, 1993, now U.S. Pat. No. 5,509,129.

This application relates to improvements in the inventions disclosed in the following copending U.S. patent applications, all of which are assigned to Texas Instruments:

U.S. patent application Ser. No. 08/263,504 filed Jun. 21, 1994, now U.S. Pat. No. 5,471,592 issued Nov. 28, 1995 and entitled MULTI-PROCESSOR WITH CROSSBAR LINK OF PROCESSORS AND MEMORIES AND METHOD OF OPERATION; which is a continuation of U.S. patent application Ser. No. 08/135,754 filed Oct. 12, 1993, now abandoned; which is a continuation of U.S. patent application Ser. No. 07/933 865 filed Aug. 21, 1992, now abandoned; which is a continuation of U.S. patent application Ser. No. 07/435,591 filed Nov. 17, 1989, now abandoned.

U.S. patent application Ser. No. 07/437,858 filed Nov. 17, 1989, now U.S. Pat. No. 5,212,777 issued May 18, 1993 and entitled MULTI-PROCESSOR RECONFIGURABLE IN SINGLE INSTRUCTION MULTIPLE DATA (SIMD) AND MULTIPLE INSTRUCTION MULTIPLE DATA (MIMD) MODES AND METHOD OF OPERATION.

U.S. patent application Ser. No. 08/264,111 filed Jun. 22, 1994, now U.S. Pat. No. 5,522,083 issued May 28, 1996 and entitled RECONFIGURABLE MULTI-PROCESSOR OPERATING IN SIMD MODE WITH ONE PROCESSOR FETCHING INSTRUCTIONS FOR USE BY REMAINING PROCESSORS; which is a continuation of U.S. patent application Ser. No. 07/895,565 filed Jun. 5, 1992, now abandoned; which is a continuation of U.S. patent application Ser. No. 07/437,856 filed Nov. 17, 1989, now abandoned.

U.S. patent application Ser. No. 08/264,582 filed Jun. 22, 1994 now pending and entitled REDUCED AREA OF CROSSBAR AND METHOD OF OPERATION; which is a continuation of U.S. patent application Ser. No. 07/437,852 filed Nov. 17, 1989, now abandoned.

U.S. patent application Ser. No. 08/032,530 filed Mar. 15, 1993 now pending and entitled SYNCHRONIZED MIMD MULTI-PROCESSING SYSTEM AND METHOD; which is a continuation of U.S. patent application Ser. No. 07/437,853 filed Nov. 17, 1989, now abandoned.

U.S. patent application Ser. No. 07/437,946 filed Nov. 17, 1989, now U.S. Pat. No. 5,197,140 issued Mar. 23, 1993 and entitled SLICED ADDRESSING MULTI-PROCESSOR AND METHOD OF OPERATION.

U.S. patent application Ser. No. 07/437,857 filed Nov. 17, 1989, now U.S. Pat. No. 5,339,447 issued Aug. 16, 1994 and entitled ONES COUNTING CIRCUIT, UTILIZING A MATRIX OF INTERCONNECTED HALF-ADDERS, FOR COUNTING THE NUMBER OF ONES IN A BINARY STRING OF IMAGE DATA.

U.S. patent application Ser. No. 07/437,851 filed Nov. 17, 1989, now U.S. Pat. No. 5,239,654 issued Aug. 24, 1993 and entitled DUAL MODE SIMD/MIMD PROCESSOR PROVIDING REUSE OF MIMD INSTRUCTION MEMORIES AS DATA MEMORIES WHEN OPERATING IN SIMD MODE.

U.S. patent application Ser. No. 07/911,562 filed Jun. 29, 1992, now U.S. Pat. No. 5,410,649 issued Apr. 25, 1995 and entitled IMAGING COMPUTER AND METHOD OF OPERATION; which is a continuation of U.S. patent application Ser. No. 07/437,854 filed Nov. 17, 1989, now abandoned.

U.S. patent application Ser. No. 07/437,875 filed Nov. 17, 1989, now U.S. Pat. No. 5,226,125 issued Jul. 6, 1993 and entitled SWITCH MATRIX HAVING INTEGRATED CROSSPOINT LOGIC AND METHOD OF OPERATION.

This application is also related to the following concurrently filed U.S. patent applications, which include the same disclosure:

U.S. patent application Ser. No. 08/160,229 "THREE INPUT ARITHMETIC LOGIC UNIT WITH BARREL ROTATOR";

U.S. patent application Ser. No. 08/158,742 "ARITHMETIC LOGIC UNIT HAVING PLURAL INDEPENDENT SECTIONS AND REGISTER STORING RESULTANT INDICATOR BIT FROM EVERY SECTION";

U.S. patent application Ser. No. 08/160,118 "MEMORY STORE FROM A REGISTER PAIR CONDITIONAL";

U.S. patent application Ser. No. 08/324,323 "ITERATIVE DIVISION APPARATUS, SYSTEM AND METHOD FORMING PLURAL QUOTIENT BITS PER ITERATION" continuation of U.S. patent application Ser. No. 08/160,115 concurrently filed with this application and now abandoned;

U.S. patent application Ser. No. 08/158,285 "THREE INPUT ARITHMETIC LOGIC UNIT FORMING MIXED ARITHMETIC AND BOOLEAN COMBINATIONS";

U.S. patent application Ser. No. 08/160,119 "METHOD, APPARATUS AND SYSTEM FORMING THE SUM OF DATA IN PLURAL EQUAL SECTIONS OF A SINGLE DATA WORD";

U.S. patent application Ser. No. 08/159,359 "HUFFMAN ENCODING METHOD, CIRCUITS AND SYSTEM EMPLOYING MOST SIGNIFICANT BIT CHANGE FOR SIZE DETECTION";

U.S. patent application Ser. No. 08/160,296 now U.S. Pat. No. 5,479,166 issued Dec. 26, 1995 entitled "HUFFMAN DECODING METHOD, CIRCUIT AND SYSTEM EMPLOYING CONDITIONAL SUBTRACTION FOR CONVERSION OF NEGATIVE NUMBERS";

U.S. patent application Ser. No. 08/160,112 "METHOD, APPARATUS AND SYSTEM FOR SUM OF PLURAL ABSOLUTE DIFFERENCES";

U.S. patent application Ser. No. 08/484,113 filed Jun. 7, 1995 now U.S. Pat. No. 5,596,519 issued Jan. 21, 1997 entitled "ITERATIVE DIVISION APPARATUS, SYSTEM AND METHOD EMPLOYING LEFT MOST ONE'S DETECTION AND LEFT MOST ONE'S DETECTION WITH EXCLUSIVE OR", which is a continuation of U.S. patent application Ser. No. 08/160,120;

U.S. patent application Ser. No. 08,160,114 "ADDRESS GENERATOR EMPLOYING SELECTIVE MERGE OF TWO INDEPENDENT ADDRESSES";

U.S. Pat. No. 5,420,809 "METHOD, APPARATUS AND SYSTEM METHOD FOR CORRELATION";

U.S. patent application Ser. No. 08/160,297 now U.S. Pat. No. 5,509,129 issued Apr. 16, 1996 entitled "LONG INSTRUCTION WORD CONTROLLING PLURAL INDEPENDENT PROCESSOR OPERATIONS";

U.S. patent application Ser. No. 08/159,346 "ROTATION REGISTER FOR ORTHOGONAL DATA TRANSFORMATION";

U.S. patent application Ser. No. 08/159,652 "MEDIAN FILTER METHOD, CIRCUIT AND SYSTEM";

U.S. patent application Ser. No. 08/159,344 "ARITHMETIC LOGIC UNIT WITH CONDITIONAL REGISTER SOURCE SELECTION";

U.S. patent application Ser. No. 08/160,301 "APPARATUS, SYSTEM AND METHOD FOR DIVISION BY ITERATION"

U.S. patent application Ser. No. 08/159,650 "MULTIPLY ROUNDING USING REDUNDANT CODED MULTIPLY RESULT";

U.S. Pat. No. 5,446,651 "SPLIT MULTIPLY OPERATION";

U.S. patent application Ser. No. 08/482,697 filed Jun. 27, 1995 "MIXED CONDITION TEST CONDITIONAL AND BRANCH OPERATIONS INCLUDING CONDITIONAL TEST FOR ZERO" a continuation of U.S. patent application Ser. No. 08/158,741 concurrently filed with this application and now abandoned;

U.S. patent application Ser. No. 08/160,302 "PACKED WORD PAIR MULTIPLY OPERATION";

U.S. patent application Ser. No. 08/160,573 "THREE INPUT ARITHMETIC LOGIC UNIT WITH SHIFTER";

U.S. patent application Ser. No. 08/159,282 now U.S. Pat. No. 5,590,350 issued Dec. 31, 1996 entitled "THREE INPUT ARITHMETIC LOGIC UNIT WITH MASK GENERATOR";

U.S. patent application Ser. No. 08/160,111 "THREE INPUT ARITHMETIC LOGIC UNIT WITH BARREL ROTATOR AND MASK GENERATOR";

U.S. patent application Ser. No. 08/160,298 "THREE INPUT ARITHMETIC LOGIC UNIT WITH SHIFTER AND MASK GENERATOR";

U.S. patent application Ser. No. 08/159,345 now U.S. Pat. No. 5,485,411 issued Jan. 16, 1996 entitled "THREE INPUT ARITHMETIC LOGIC UNIT FORMING THE SUM OF A FIRST INPUT ADDED WITH A FIRST BOOLEAN COMBINATION OF A SECOND INPUT AND THIRD INPUT PLUS A SECOND BOOLEAN COMBINATION OF THE SECOND AND THIRD INPUTS";

U.S. Pat. No. 5,465,224 "THREE INPUT ARITHMETIC LOGIC UNIT FORMING THE SUM OF FIRST BOOLEAN COMBINATION OF FIRST, SECOND AND THIRD INPUTS PLUS A SECOND BOOLEAN COMBINATION OF FIRST, SECOND AND THIRD INPUTS";

U.S. patent application Ser. No. 08/426,992 filed Apr. 24, 1995 now U.S. Pat. No. 5,493,542 issued Feb. 20, 1996 entitled "THREE INPUT ARITHMETIC LOGIC UNIT EMPLOYING CARRY PROPAGATE LOGIC" which is a continuation of U.S. patent application Ser. No. 08/159,640 now abandoned; and

U.S. patent application Ser. No. 08/160,300 "DATA PROCESSING APPARATUS, SYSTEM AND METHOD FOR IF, THEN, ELSE OPERATION USING WRITE PRIORITY."

TECHNICAL FIELD OF THE INVENTION

The technical field of this invention is the field of digital data processing and more particularly microprocessor circuits, architectures and methods for digital data processing especially digital image/graphics processing.

BACKGROUND OF THE INVENTION

This invention relates to the field of computer graphics and in particular to bit mapped graphics. In bit mapped graphics computer memory stores data for each individual picture element or pixel of an image at memory locations that correspond to the location of that pixel within the image. This image may be an image to be displayed or a captured image to be manipulated, stored, displayed or retransmitted. The field of bit mapped computer graphics has benefited greatly from the lowered cost and increased capacity of dynamic random access memory (DRAM) and the lowered cost and increased processing power of microprocessors. These advantageous changes in the cost and performance of component parts enable larger and more complex computer image systems to be economically feasible.

The field of bit mapped graphics has undergone several stages in evolution of the types of processing used for image data manipulation. Initially a computer system supporting bit mapped graphics employed the system processor for all bit mapped operations. This type of system suffered several drawbacks. First, the computer system processor was not particularly designed for handling bit mapped graphics. Design choices that are very reasonable for general purpose computing are unsuitable for bit mapped graphics systems. Consequently some routine graphics tasks operated slowly. In addition, it was quickly discovered that the processing needed for image manipulation of bit mapped graphics was so loading the computational capacity of the system processor that other operations were also slowed.

The next step in the evolution of bit mapped graphics processing was dedicated hardware graphics controllers. These devices can draw simple figures, such as lines, ellipses and circles, under the control of the system processor. Many of these devices can also do pixel block transfers (PixBlt). A pixel block transfer is a memory move operation of image data from one portion of memory to another. A pixel block transfer is useful for rendering standard image elements, such as alphanumeric characters in a particular type font, within a display by transfer from nondisplayed memory to bit mapped display memory. This function can also be used for tiling by transferring the same small image to the whole of bit mapped display memory. The built-in algorithms for performing some of the most frequently used graphics functions provide a way of improving system performance. However, a useful graphics computer system often requires many functions besides those few that are implemented in such a hardware graphics controller. These additional functions must be implemented in software by the system processor. Typically these hardware graphics controllers allow the system processor only limited access to the bit map memory, thereby limiting the degree to which system software can augment the fixed set of functions of the hardware graphics controller.

The graphics system processor represents yet a further step in the evolution of bit mapped graphics processing. A graphics system processor is a programmable device that has all the attributes of a microprocessor and also includes special functions for bit mapped graphics. The TMS34010 and TMS34020 graphics system processors manufactured by Texas instruments Incorporated represent this class of devices. These graphics system processors respond to a stored program in the same manner as a microprocessor and include the capability of data manipulation via an arithmetic logic unit, data storage in register files and control of both program flow and external data memory. In addition, these devices include special purpose graphics manipulation hardware that operate under program control. Additional instructions within the instruction set of these graphics system processors controls the special purpose graphics hardware. These instructions and the hardware that supports them are

selected to perform base level graphics functions that are useful in many contexts. Thus a graphics system processor can be programmed for many differing graphics applications using algorithms selected for the particular problem. This provides an increase in usefulness similar to that provided by changing from hardware controllers to programmed microprocessors. Because such graphics system processors are programmable devices in the same manner as microprocessors, they can operate as stand alone graphics processors, graphics co-processors slaved to a system processor or tightly coupled graphics controllers.

New applications are driving the desire to provide more powerful graphics functions. Several fields require more cost effective graphics operations to be economically feasible. These include video conferencing, multi-media computing with full motion video, high definition television, color facsimile and digital photography. Each of these fields presents unique problems, but image data compression and decompression are common themes. The amount of transmission bandwidth and the amount of storage capacity required for images and particular full motion video is enormous. Without efficient video compression and decompression that result in acceptable final image quality, these applications will be limited by the costs associated with transmission bandwidth and storage capacity. There is also a need in the art for a single system that can support both image processing functions such as image recognition and graphics functions such as display control.

SUMMARY OF THE INVENTION

A data processing apparatus including a multiplier unit forming a product from a set of L bits of each two data buses of N bits each N is greater than L. The multiplier forms a N bit output having a first portion and a second portion. The first portion is the L most significant bits of the of product. The second portion is M other bits not including the L least significant bits of the product, where N is the sum of M and L. In the preferred embodiment the M other bits are derived from other bits of the two input data busses, such as the M other bits of the first input data bus.

The data processing apparatus includes an arithmetic logic unit performing parallel operations controlled by the same instructions. This arithmetic logic unit is divisible into a selected number of sections for performing identical operations on independent sections of its inputs. Preferably the arithmetic logic unit includes 32 bits and may be divided into two 16 bit sections and into four 8 bit sections. The arithmetic logic unit operation may be addition, subtraction or a Boolean function.

The multiplier unit may operate on signed inputs to generate a signed product or on unsigned inputs to generate an unsigned product. The multiplier unit may form dual products from separate parts of the input data.

A single instruction may controlling both the multiplier unit and the arithmetic logic unit permits addition of dual products. The dual products are temporarily stored in a data register permitting the multiply and add operations to be pipelined. In the preferred embodiment the dual products are formed in one data word and added by a rotate/mask and add operation in a three input arithmetic unit.

In the preferred embodiment of this invention, the data unit including the data registers, the multiplication unit and the arithmetic logic unit, the address unit and the instruction decode logic are embodied in at least one digital image/graphics processor as a part of a multiprocessor formed in a single integrated circuit used in image processing.

BRIEF DESCRIPTION OF THE FIGURES

These and other aspects of the present invention are described below together with the Figures, in which:

FIG. 1 illustrates the system architecture of an image processing system such as would employ this invention;

FIG. 2 illustrates the architecture of a single integrated circuit multiprocessor that forms the preferred embodiment of this invention;

FIG. 3 illustrates in block diagram form one of the digital image/graphics processors illustrated in FIG. 2;

FIG. 4 illustrates in schematic form the pipeline stages of operation of the digital image/graphics processor illustrated in FIG. 2;

FIG. 5 illustrates in block diagram form the data unit of the digital image/graphics processors illustrated in FIG. 3;

FIG. 6 illustrates in schematic form field definitions of the status register of the data unit illustrated in FIG. 5;

FIG. 7 illustrates in block diagram form the manner of splitting the arithmetic logic unit of the data unit illustrated in FIG. 5;

FIG. 8 illustrates in block diagram form the manner of addressing the data register of the data unit illustrated in FIG. 5 as a rotation register;

FIG. 9 illustrates in schematic form the field definitions of the first data register of the data unit illustrated in FIG. 5;

FIG. 10*a* illustrates in schematic form the data input format for 16 bit by 16 bit signed multiplication operands;

FIG. 10*b* illustrates in schematic form the data output format for 16 bit by 16 bit signed multiplication results;

FIG. 10*c* illustrates in schematic form the data input format for 16 bit by 16 bit unsigned multiplication operands;

FIG. 10*d* illustrates in schematic form the data output format for 16 bit by 16 bit unsigned multiplication results;

FIG. 11*a* illustrates in schematic form the data input format for dual 8 bit by 8 bit signed multiplication operands;

FIG. 11*b* illustrates in schematic form the data input format for dual 8 bit by 8 bit unsigned multiplication operands;

FIG. 11*c* illustrates in schematic form the data output format for dual 8 bit by 8 bit signed multiplication results;

FIG. 11*d* illustrates in schematic form the data output format for dual 8 bit by 8 bit unsigned multiplication results;

FIG. 12 illustrates in block diagram form the multiplier illustrated in FIG. 5;

FIG. 13 illustrates in schematic form generation of Booth quads for the first operand in 16 bit by 16 bit multiplication;

FIG. 14 illustrates in schematic form generation of Booth quads for dual first operands in 8 bit by 8 bit multiplication;

FIG. 15*a* illustrates in schematic form the second operand supplied to the partial product generators illustrated in FIG. 12 in 16 bit by 16 bit unsigned multiplication;

FIG. 15*b* illustrates in schematic form the second operand supplied to the partial product generators illustrated in FIG. 12 in 16 bit by 16 bit signed multiplication;

FIG. 16*a* illustrates in schematic form the second operand supplied to the first three partial product generators illustrated in FIG. 12 in dual 8 bit by 8 bit unsigned multiplication;

FIG. 16*b* illustrates in schematic form the second operand supplied to the first three partial product generators illustrated in FIG. 12 in dual 8 bit by 8 bit signed multiplication;

FIG. 16*c* illustrates in schematic form the second operand supplied to the second three partial product generators illustrated in FIG. 12 in dual 8 bit by 8 bit unsigned multiplication;

FIG. 16*d* illustrates in schematic form the second operand supplied, to the second three partial product generators illustrated in FIG. 12 in dual 8 bit by 8 bit signed multiplication;

FIG. 17*a* illustrates in schematic form the output mapping for 16 bit by 16 bit multiplication;

FIG. 17*b* illustrates in schematic form the output mapping for dual 8 bit by 8 bit multiplication;

FIG. 18 illustrates in block diagram form the details of the construction of the rounding adder 226 illustrated in FIG. 5;

FIG. 19 illustrates in block diagram form the construction of one bit circuit of the arithmetic logic unit of the data unit illustrated in FIG. 5;

FIG. 20 illustrates in schematic form the construction of the resultant logic and carry out logic of the bit circuit illustrated in FIG. 19;

FIG. 21 illustrates in schematic form the construction of the Boolean function generator of the bit circuit illustrated in FIG. 19;

FIG. 22 illustrates in block diagram form the function signal selector of the function signal generator of the data unit illustrated in FIG. 5;

FIG. 23 illustrates in block diagram form the function signal modifier portion of the function signal generator of the data unit illustrated in FIG. 5;

FIG. 24 illustrates in block diagram form the bit 0 carry-in generator of the data unit illustrated in FIG. 5;

FIG. 25 illustrates in block diagram form a conceptual view of the arithmetic logic unit illustrated in FIGS. 19 and 20;

FIG. 26 illustrates in block diagram form a conceptual view of an alternative embodiment of the arithmetic logic unit;

FIG. 27 illustrates in block diagram form the address unit of the digital image/graphics processor illustrated in FIG. 3;

FIG. 28 illustrates in block diagram form an example of a global or a local address unit of the address unit illustrated in FIG. 27;

FIG. 29*a* illustrates the order of data bytes according to the little endian mode;

FIG. 29*b* illustrates the order of data bytes according to the big endian mode;

FIG. 30 illustrates a circuit for data selection, data alignment and sign or zero extension in each data port of a digital image/graphics processor;

FIG. 31 illustrates in block diagram form the program flow control unit of the digital image/graphics processors illustrated in FIG. 3;

FIG. 32 illustrates in schematic form the field definitions of the program counter of the program flow control unit illustrated in FIG. 31;

FIG. 33 illustrates in schematic form the field definitions of the instruction pointer-address stage register of the program flow control unit illustrated in FIG. 31;

FIG. 34 illustrates in schematic form the field definitions of the instruction pointer-return from subroutine register of the program flow control unit illustrated in FIG. 31;

FIG. 35 illustrates in schematic form the field definitions of the cache tag registers of the program flow control unit illustrated in FIG. 31;

FIG. 36 illustrates in schematic form the field definitions of the loop logic control register of the program flow control unit illustrated in FIG. 31;

FIG. 37 illustrates in block diagram form the loop logic circuit of the program flow control unit;

FIG. 38 illustrates in flow chart form a program example of a single program loop with multiple loop ends;

FIG. 39 illustrates the overlapping pipeline stages in an example of a software branch from a single instruction hardware loop;

FIG. 40 illustrates in schematic form the field definitions of the interrupt enable register and the interrupt flag register of the program flow control unit illustrated in FIG. 31;

FIG. 41 illustrates in schematic form the field definitions of a command word transmitted between processors of the single integrated circuit multiprocessor illustrated in FIG. 2;

FIG. 42 illustrates in schematic form the field definitions of the communications register of the program flow control unit illustrated in FIG. 31;

FIG. 43 illustrates in schematic form the instruction word controlling the operation of the digital image/graphics processor illustrated in FIG. 3;

FIG. 44 illustrates in schematic form data flow within the data unit during execution of a divide iteration instruction;

FIG. 45 illustrates in flow chart form the use of a left most one's function in a division algorithm;

FIG. 46 illustrates in flow chart form the use of a left most one's function and an exclusive OR in a division algorithm;

FIG. 47 illustrates in schematic form within the data flow during an example sum of absolute value of differences algorithm;

FIGS. 48*a*, 48*b*, 48*c*, 48*d* and 48*e* illustrate in schematic form a median filter algorithm;

FIG. 49 illustrates the overlapping pipeline stages in an example of a single instruction hardware loop with a conditional hardware branch;

FIG. 50 illustrates in schematic form a hardware divider that generates two bits of the desired quotient per divide iteration;

FIG. 51 illustrates in schematic form the data flow within the hardware divider illustrated in FIG. 48;

FIG. 52 illustrates in schematic form a hardware divider that generates three bits of the desired quotient per divide iteration;

FIG. 53 illustrates in schematic form the data flow within a hardware divider illustrated in FIG. 51; and

FIG. 54 illustrates in schematic form the multiprocessor integrated circuit of this invention having a single digital image/graphics processor in color facsimile system.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

FIG. 1 is a block diagram of an image data processing system including a multiprocessor integrated circuit constructed for image and graphics processing according to this invention. This data processing system includes a host processing system 1. Host processing system 1 provides the data processing for the host system of data processing system of FIG. 1. Included in the host processing system 1 are a processor, at least one input device, a long term storage device, a read only memory, a random access memory and at least one host peripheral 2 coupled to a host system bus. Arrangement and operation of the host processing system are considered conventional. Because of its processing functions, the host processing system 1 controls the function of the image data processing system.

Multiprocessor integrated circuit 100 provides most of the data processing including data manipulation and computa-

tion for image operations of the image data processing system of FIG. 1. Multiprocessor integrated circuit **100** is bi-directionally coupled to an image system bus and communicates with host processing system **1** by way of this image system bus. In the arrangement of FIG. 1, multiprocessor integrated circuit **100** operates independently from the host processing system **1**. The multiprocessor integrated circuit **100**, however, is responsive to host processing system **1**.

FIG. 1 illustrates two image systems. Imaging device **3** represents a document scanner, charge coupled device scanner or video camera that serves as an image input device. Imagine device **3** supplies this image to image capture controller **4**, which serves to digitize the image and form it into raster scan frames. This frame capture process is controlled by signals from multiprocessor integrated circuit **100**. The thus formed image frames are stored in video random access memory **5**. Video random access memory **5** may be accessed via the image system bus permitting data transfer for image processing by multiprocessor integrated circuit **100**.

The second image system drives a video display. Multiprocessor integrated circuit **100** communicates with video random access memory **6** for specification of a displayed image via a pixel map. Multiprocessor integrated circuit **100** controls the image data stored in video random access memory **6** via the image system bus. Data corresponding to this image is recalled from video random access memory **6** and supplied to video palette **7**. Video palette **7** may transform this recalled data into another color space, expand the number of bits per pixel and the like. This conversion may be accomplished through a look-up table. Video palette **7** also generates the proper video signals to drive video display **8**. If these video signals are analog signals, then video palette **7** includes suitable digital to analog conversion. The video level signal output from the video palette **7** may include color, saturation, and brightness information. Multiprocessor integrated circuit **100** controls data stored within the video palette **7**, thus controlling the data transformation process and the timing of image frames. Multiprocessor integrated circuit **100** can control the line length and the number of lines per frame of the video display image, the synchronization, retrace, and blanking signals through control of video palette **7**. Significantly, multiprocessor integrated circuit **100** determines and controls where graphic display information is stored in the video random access memory **6**. Subsequently, during readout from the video random access memory **6**, multiprocessor integrated circuit **100** determines the readout sequence from the video random access memory **6**, the addresses to be accessed, and control information needed to produce the desired graphic image on video display **8**.

Video display **8** produces the specified video display for viewing by the user. There are two widely used techniques. The first technique specifies video data in terms of color, hue, brightness, and saturation for each pixel. For the second technique, color levels of red, blue and green are specified for each pixel. Video palette **7** the video display **8** is designed and fabricated to be compatible with the selected technique.

FIG. 1 illustrates an addition memory **9** coupled to the image system bus. This additional memory may include additional video random access memory, dynamic random access memory, static random access memory or read only memory. Multiprocessor integrated circuit **100** may be controlled either in wholly or partially by a program stored in the memory **9**. This memory **9** may also store various types of graphic image data. In addition, multiprocessor integrated circuit **100** preferably includes memory interface circuits for video random access memory, dynamic random access memory and static random access memory. Thus a system could be constructed using multiprocessor integrated circuit **100** without any video random access memory **5** or **6**.

FIG. 1 illustrates transceiver **16**. Transceiver **16** provides translation and bidirectional communication between the image system bus and a communications channel. One example of a system employing transceiver **16** is video conferencing. The image data processing system illustrated in FIG. 1 employs imaging device **3** and image capture controller **4** to form a video image of persons at a first location. Multiprocessor integrated circuit **100** provides video compression and transmits the compressed video signal to a similar image data processing system at another location via transceiver **16** and the communications channel. Transceiver **16** receives a similarly compressed video signal from the remote image data processing system via the communications channel. Multiprocessor integrated circuit **100** decompresses this received signal and controls video random access memory **6** and video palette **7** to display the corresponding decompressed video signal on video display **8**. Note this is not the only example where the image data processing system employs transceiver **16**. Also note that the bidirectional communications need not be the same type signals. For example, in an interactive cable television signal the cable system head in would transmit compressed video signals to the image data processing system via the communications channel. The image data processing system could transmit control and data signals back to the cable system head in via transceiver **16** and the communications channel.

FIG. 1 illustrates multiprocessor integrated circuit **100** embodied in a system including host processing system **1**. Those skilled in the art would realize from the following disclosure of the invention that multiprocessor integrated circuit **100** may be employed as the only processor of a useful system. In such a system multiprocessor integrated circuit **100** is programmed to perform all the functions of the system.

This invention is particularly useful in a processor used for image processing. According to the preferred embodiment, this invention is embodied in multiprocessor integrated circuit **100**. This preferred embodiment includes plural identical processors that embody this invention. Each of these processors will be called a digital image/graphics processor. This description is a matter of convenience only. The processor embodying this invention can be a processor separately fabricated on a single integrated circuit or a plurality of integrated circuits. If embodied on a single integrated circuit, this single integrated circuit may optionally also include read only memory and random access memory used by the digital image/graphics processor.

FIG. 2 illustrates the architecture of the multiprocessor integrated circuit **100** of the preferred embodiment of this invention. Multiprocessor integrated circuit **100** includes: two random access memories **10** and **20**, each of which is divided into plural sections; crossbar **50**; master processor **60**; digital image/graphics processors **71**, **72**, **73** and **74**; transfer controller **80**, which mediates access to system memory; and frame controller **90**, which can control access to independent first and second image memories. Multiprocessor integrated circuit **100** provides a high degree of operation parallelism, which will be useful in image processing and graphics operations, such as in the multi-media computing.

Multiprocessor integrated circuit **100** includes two random access memories. Random access memory **10** is primarily devoted to master processor **60**. It includes two instruction cache memories **11** and **12**, two data cache memories **13** and **14** and a parameter memory **15**. These memory sections can be physically identical, but connected and used differently. Random access memory **20** may be accessed by master processor **60** and each of the digital image/graphics processors **71**, **72**, **73** and **74**. Each digital image/graphics processor **71**, **72**, **73** and **74** has five corresponding memory sections. These include an instruction cache memory, three data memories and one parameter memory. Thus digital image/graphics processor **71** has corresponding instruction cache memory **21**, data memories **22**, **23**, **24** and parameter memory **25**; digital image/graphics processor **72** has corresponding instruction cache memory **26**, data memories **27**, **28**, **29** and parameter memory **30**; digital image/graphics processor **73** has corresponding instruction cache memory **31**, data memories **32**, **33**, **34** and parameter memory **35**; and digital image/graphics processor **74** has corresponding instruction cache memory **36**, data memories **37**, **38**, **39** and parameter memory **40**. Like the sections of random access memory **10**, these memory sections can be physically identical but connected and used differently. Each of these memory sections of memories **10** and **20** preferably includes 2K bytes, with a total memory within multiprocessor integrated circuit **100** of 50K bytes.

Multiprocessor integrated circuit **100** is constructed to provide a high rate of data transfer between processors and memory using plural independent parallel data transfers. Crossbar **50** enables these data transfers. Each digital image/graphics processor **71**, **72**, **73** and **74** has three memory ports that may operate simultaneously each cycle. An instruction port (I) may fetch 64 bit data words from the corresponding instruction cache. A local data port (L) may read a 32 bit data word from or write a 32 bit data word into the data memories or the parameter memory corresponding to that digital image/graphics processor. A global data port (G) may read a 32 bit data word from or write a 32 bit data word into any of the data memories or the parameter memories or random access memory **20**. Master Processor **60** includes two memory ports. An instruction port (I) may fetch a 32 bit instruction word from either of the instruction caches **11** and **12**. A data port (C) may read a 32 bit data word from or write a 32 bit data word into data caches **13** or **14**, parameter memory **15** of random access memory **10** or any of the data memories, the parameter memories of random access memory **20**. Transfer controller **80** can access any of the sections of random access memory **10** or **20** via data port (C). Thus fifteen parallel memory accesses may be requested at any single memory cycle. Random access memories **10** and **20** are divided into 25 memories in order to support so many parallel accesses.

Crossbar **50** controls the connections of master processor **60**, digital image/graphics processors **71**, **72**, **73** and **74**, and transfer controller **80** with memories **10** and **20**. Crossbar **50** includes a plurality of crosspoints **51** disposed in rows and columns. Each column of crosspoints **51** corresponds to a single memory section and a corresponding range of addresses. A processor requests access to one of the memory sections through the most significant bits of an address output by that processor. This address output by the processor travels along a row. The crosspoint **51** corresponding to the memory section having that address responds either by granting or denying access to the memory section. If no other processor has requested access to that memory section during the current memory cycle, then the crosspoint **51** grants access by coupling the row and column. This supplies the address to the memory section. The memory section responds by permitting data access at that address. This data access may be either a data read operation or a data write operation.

If more than one processor requests access to the same memory section simultaneously, then crossbar **50** grants access to only one of the requesting processors. The crosspoints **51** in each column of crossbar **50** communicate and grant access based upon a priority hierarchy. If two requests for access having the same rank occur simultaneously, then crossbar **50** grants access on a round robin basis, with the processor last granted access having the lowest priority. Each granted access lasts as long as needed to service the request. The processors may change their addresses every memory cycle, so crossbar **50** can change the interconnection between the processors and the memory sections on a cycle by cycle basis.

Master processor **60** preferably performs the major control functions for multiprocessor integrated circuit **100**. Master processor **60** is preferably a 32 bit reduced instruction set computer (RISC) processor including a hardware floating point calculation unit. According to the RISC architecture, all accesses to memory are performed with load and store instructions and most integer and logical operations are performed on registers in a single cycle. The floating point calculation unit, however, will generally take several cycles to perform operations when employing the same register file as used by the integer and logical unit. A register score board ensures that correct register access sequences are maintained. The RISC architecture is suitable for control functions in image processing. The floating point calculation unit permits rapid computation of image rotation functions, which may be important to image processing.

Master processor **60** fetches instruction words from instruction cache memory **11** or instruction cache memory **12**. Likewise, master processor **60** fetches data from either data cache **13** or data cache **14**. Since each memory section includes 2K bytes of memory, there is 4K bytes of instruction cache and 4K bytes of data cache. Cache control is an integral function of master processor **60**. As previously mentioned, master processor **60** may also access other memory sections via crossbar **50**.

The four digital image/graphics processors **71**, **72**, **73** and **74** each have a highly parallel digital signal processor (DSP) architecture. FIG. **3** illustrates an overview of exemplary digital image/graphics processor **71**, which is identical to digital image/graphics processors **72**, **73** and **74**. Digital image/graphics processor **71** achieves a high degree of parallelism of operation employing three separate units: data unit **110**; address unit **120**; and program flow control unit **130**. These three units operate simultaneously on different instructions in an instruction pipeline. In addition each of these units contains internal parallelism.

The digital image/graphics processors **71**, **72**, **73** and **74** can execute independent instruction streams in the multiple instruction multiple data mode (MIMD). In the MIMD mode, each digital image/graphics processor executes an individual program from its corresponding instruction cache, which may be independent or cooperative. In the latter case crossbar **50** enables inter-processor communication in combination with the shared memory. Digital image/graphics processors **71**, **72**, **73** and **74** may also operate in a synchronized MIMD mode. In the synchronized MIMD mode, the program control flow unit **130** of each digital image/graphics processor inhibits fetching the next instruc-

tion until all synchronized processors are ready to proceed. This synchronized MIMD mode allows the separate programs of the digital image/graphics processors to be executed in lock step in a closely coupled operation.

Digital image/graphics processors **71**, **72**, **73** and **74** can execute identical instructions on differing data in the single instruction multiple data mode (SIMD). In this mode a single instruction stream for the four digital image/graphics processors comes from instruction cache memory **21**. Digital image/graphics processor **71** controls the fetching and branching operations and crossbar **50** supplies the same instruction to the other digital image/graphics processors **72**, **73** and **74**. Since digital image/graphics processor **71** controls instruction fetch for all the digital image/graphics processors **71**, **72**, **73** and **74**, the digital image/graphics processors are inherently synchronized in the SIMD mode.

Transfer controller **80** is a combined direct memory access (DMA) machine and memory interface for multiprocessor integrated circuit **100**. Transfer controller **80** intelligently queues, sets priorities and services the data requests and cache misses of the five programmable processors. Master processor **60** and digital image/graphics processors **71**, **72**, **73** and **74** all access memory and systems external to multiprocessor integrated circuit **100** via transfer controller **80**. Data cache or instruction cache misses are automatically handled by transfer controller **80**. The cache service (S) port transmits such cache misses to transfer controller **80**. Cache service port (S) reads information from the processors and not from memory. Master processor **60** and digital image/graphics processors **71**, **72**, **73** and **74** may request data transfers from transfer controller **80** as linked list packet requests. These linked list packet requests allow multidimensional blocks of information to be transferred between source and destination memory addresses, which can be within multiprocessor integrated circuit **100** or external to multiprocessor integrated circuit **100**. Transfer controller **80** preferably also includes a refresh controller for dynamic random access memory (DRAM) which require periodic refresh to retain their data.

Frame controller **90** is the interface between multiprocessor integrated circuit **100** and external image capture and display systems. Frame controller **90** provides control over capture and display devices, and manages the movement of data between these devices and memory automatically. To this end, frame controller **90** provides simultaneous control over two independent image systems. These would typically include a first image system for image capture and a second image system for image display, although the application of frame controller **90** is controlled by the user. These image systems would ordinarily include independent frame memories used for either frame grabber or frame buffer storage. Frame controlled **90** preferably operates to control video dynamic random access memory (VRAM) through refresh and shift register control.

Multiprocessor integrated circuit **100** is designed for large scale image processing. Master processor **60** provides embedded control, orchestrating the activities of the digital image/graphics processors **71**, **72**, **73** and **74**, and interpreting the results that they produce. Digital image/graphics processors **71**, **72**, **73** and **74** are well suited to pixel analysis and manipulation. If pixels are thought of as high in data but low in information, then in a typical application digital image/graphics processors **71**, **72**, **73** and **74** might well examine the pixels and turn the raw data into information. This information can then be analyzed either by the digital image/graphics processors **71**, **72**, **73** and **74** or by master processor **60**. Crossbar **50** mediates inter-processor communication. Crossbar **50** allows multiprocessor integrated circuit **100** to be implemented as a shared memory system. Message passing need not be a primary form of communication in this architecture. However, messages can be passed via the shared memories. Each digital image/graphics processor, the corresponding section of crossbar **50** and the corresponding sections of memory **20** have the same width. This permits architecture flexibility by accommodating the addition or removal of digital image/graphics processors and corresponding memory modularly while maintaining the same pin out.

In the preferred embodiment all parts of multiprocessor integrated circuit **100** are disposed on a single integrated circuit. In the preferred embodiment, multiprocessor integrated circuit **100** is formed in complementary metal oxide semiconductor (CMOS) using feature sizes of 0.6 μm. Multiprocessor integrated circuit **100** is preferably constructed in a pin grid array package having 256 pins. The inputs and outputs are preferably compatible with transistor-transistor logic (TTL) logic voltages. Multiprocessor integrated circuit **100** preferably includes about 3 million transistors and employs a clock rate of 50M Hz.

FIG. **3** illustrates an overview of exemplary digital image/graphics processor **71**, which is virtually identical to digital image/graphics processors **72**, **73** and **74**. Digital image/graphics processor **71** includes: data unit **110**; address unit **120**; and program flow control unit **130**. Data unit **110** performs the logical or arithmetic data operations. Data unit **110** includes eight data registers D7–D0, a status register **210** and a multiple flags register **211**. Address unit **120** controls generation of load/store addresses for the local data port and the global data port. As will be further described below, address unit **120** includes two virtually identical addressing units, one for local addressing and one for global addressing. Each of these addressing units includes an all "0" read only register enabling absolute addressing in a relative address mode, a stack pointer, five address registers and three index registers. The addressing units share a global bit multiplex control register used when forming a merging address from both address units. Program flow control unit **130** controls the program flow for the digital image/graphics processor **71** including generation of addresses for instruction fetch via the instruction port. Program flow control unit **130** includes; a program counter PC **701**; an instruction pointer-address stage IRA **702** that holds the address of the instruction currently in the address pipeline stage; an instruction pointer-execute stage IRE **703** that holds the address of the instruction currently in the execute pipeline stage; an instruction pointer-return from subroutine IPRS **704** holding the address for returns from subroutines; a set of registers controlling zero overhead loops; four cache tag registers TAG3–TAG0 collectively called **708** that hold the most significant bits of four blocks of instruction words in the corresponding instruction cache memory.

Digital image/graphics processor **71** operates on a three stage pipeline as illustrated in FIG. **4**. Data unit **110**, address unit **120** and program flow control unit **130** operate simultaneously on different instructions in an instruction pipeline. The three stages in chronological order are fetch, address and execute. Thus at any time, digital image/graphics processor **71** will be operating on differing functions of three instructions. The phrase pipeline stage is used instead of referring to clock cycles, to indicate that specific events occur when the pipeline advances, and not during stall conditions.

Program flow control unit **130** performs all the operations that occur during the fetch pipeline stage. Program flow

control unit **130** includes a program counter, loop logic, interrupt logic and pipeline control logic. During the fetch pipeline stage, the next instruction word is fetched from memory. The address contained in the program counter is compared with cache tag registers to determine if the next instruction word is stored in instruction cache memory **21**. Program flow control unit **130** supplies the address in the program counter to the instruction port address bus **131** to fetch this next instruction word from instruction cache memory **21** if present. Crossbar **50** transmits this address to the corresponding instruction cache, here instruction cache memory **21**, which returns the instruction word on the instruction bus **132**. Otherwise, a cache miss occurs and transfer controller **80** accesses external memory to obtain the next instruction word. The program counter is updated. If the following instruction word is at the next sequential address, program control flow unit **130** post increments the program counter. Otherwise, program control flow unit **130** loads the address of the next instruction word according to the loop logic or software branch. If the synchronized MIMD mode is active, then the instruction fetch waits until all the specified digital image/graphics processors are synchronized, as indicated by sync bits in a communications register.

Address unit **120** performs all the address calculations of the address pipeline stage. Address unit **120** includes two independent address units, one for the global port and one for the local port. If the instruction calls for one or two memory accesses, then address unit **120** generates the address(es) during the address pipeline stage. The address (es) are supplied to crossbar **50** via the respective global port address bus **121** and local port address bus **122** for contention detection/prioritization. If there is no contention, then the accessed memory prepares to allow the requested access, but the memory access occurs during the following execute pipeline stage.

Data unit **110** performs all of the logical and arithmetic operations during the execute pipeline stage. All logical and arithmetic operations and all data movements to or from memory occur during the execute pipeline stage. The global data port and the local data port complete any memory accesses, which are begun during the address pipeline stage, during the execute pipeline stage. The global data port and the local data port perform all data alignment needed by memory stores, and any data extraction and sign extension needed by memory loads. If the program counter is specified as a data destination during any operation of the execute pipeline stage, then a delay of two instructions is experienced before any branch takes effect. The pipelined operation requires this delay, since the next two instructions following such a branch instruction have already been fetched. According to the practice in RISC processors, other useful instructions may be placed in the two delay slot positions.

Digital image/graphics processor **71** includes three internal 32 bit data busses. These are local port data bus Lbus **103**, global port source data bus Gsrc **105** and global port destination data bus Gdst **107**. These three buses interconnect data unit **110**, address unit **120** and program flow control unit **130**. These three buses are also connected to a data port unit **140** having a local port **141** and global port **145**. Data port unit **140** is coupled to crossbar **50** providing memory access.

Local data port **141** has a buffer **142** for data stores to memory. A multiplexer/buffer circuit **143** loads data onto Lbus **103** from local port data bus **144** from memory via crossbar **50**, from a local port address bus **122** or from global port data bus **148**. Local port data bus Lbus **103** thus carries 32 bit data that is either register sourced (stores) or memory sourced (loads). Advantageously, arithmetic results in address unit **120** can be supplied via local port address bus **122**, multiplexer buffer **143** to local port data bus Lbus **103** to supplement the arithmetic operations of data unit **110**. This will be further described below. Buffer **142** and multiplexer buffer **143** perform alignment and extraction of data. Local port data bus Lbus **103** connects to data registers in data unit **110**. A local bus temporary holding register LTD **104** is also connected to local port data Lbus **103**.

Global port source data bus Gsrc **105** and global port destination data bus Gdst **107** mediate global data transfers. These global data transfers may be either memory accesses, register to register moves or command word transfers between processors. Global port source data bus Gsrc **105** carries 32 bit source information of a global port data transfer. The data source can be any of the registers of digital image/graphics processor **71** or any data or parameter memory corresponding to any of the digital image/graphics processors **71**, **72**, **73** or **74**. The data is stored to memory via the global port **145**. Multiplexer buffer **146** selects lines from local port data Lbus **103** or global port source data bus Gsrc **105**, and performs data alignment. Multiplexer buffer **146** writes this data onto global port data bus **148** for application to memory via crossbar **50**. Global port source data bus Gsrc **105** also supplies data to data unit **110**, allowing the data of global port source data bus Gsrc **105** to be used as one of the arithmetic logic unit sources. This latter connection allows any register of digital image/graphics processor **71** to be a source for an arithmetic logic unit operation.

Global port destination data bus Gdst **107** carries 32 bit destination data of a global bus data transfer. The destination is any register of digital image/graphic processor **71**. Buffer **147** in global port **145** sources the data of global port destination data bus Gdst **107**. Buffer **147** performs any needed data extraction and sign extension operations. This buffer **147** operates if the data source is memory, and a load is thus being performed. The arithmetic logic unit result serves as an alternative data source for global port destination data bus Gdst **107**. This allows any register of digital image/graphics processor **71** to be the destination of an arithmetic logic unit operation. A global bus temporary holding register GTD **108** is also connected to global port destination data bus Gdst **107**.

Circuitry including multiplexer buffers **143** and **146** connect between global port source data bus Gsrc **105** and global port destination data bus Gdst **107** to provide register to register moves. This allows a read from any register of digital image/graphics processor **71** onto global port source data bus Gsrc **105** to be written to any register of digital image/graphics processor **71** via global port destination data bus Gdst **107**.

Note that it is advantageously possible to perform a load of any register of digital image/graphics processor **71** from memory via global port destination data bus Gdst **107**, while simultaneously sourcing the arithmetic logic unit in data unit **110** from any register via global port source data bus Gsrc **105**. Similarly, it is advantageously possible to store the data in any register of digital image/graphics processor **71** to memory via global port source data bus Gsrc **105**, while saving the result of an arithmetic logic unit operation to any register of digital image/graphics processor **71** via global port destination data bus Gdst **107**. The usefulness of these data transfers will be further detailed below.

Program flow control unit **130** receives the instruction words fetched from instruction cache memory **21** via

instruction bus **132**. This fetched instruction word is advantageously stored in two 64 bit instruction registers designated instruction register-address stage IRA **751** and instruction register-execute stage IRE **752**. Each of the instruction registers IRA and IRE have their contents decoded and distributed. Digital image/graphics processor **71** includes opcode bus **133** that carries decoded or partially decoded instruction contents to data unit **110** and address unit **120**. As will be later described, an instruction word may include a 32 bit, a 15 bit or a 3 bit immediate field. Program flow control unit **130** routes such an immediate field to global port source data bus Gsrc **105** for supply to its destination.

Digital image/graphics processor **71** includes three address buses **121**, **122** and **131**. Address unit **120** generates addresses on global port address bus **121** and local port address bus **122**. As will be further detailed below, address unit **120** includes separate global and local address units, which provide the addresses on global port address bus **121** and local port address bus **122**, respectively. Note that local address unit **620** may access memory other than the data memory corresponding to that digital image/graphics processor. In that event the local address unit access is via global port address bus **121**. Program flow control unit **130** sources the instruction address on instruction port address bus **131** from a combination of address bits from a program counter and cache control logic. These address buses **121**, **122** and **131** each carry address, byte strobe and read/write information.

FIG. **5** illustrates details of data unit **110**. It should be understood that FIG. **5** does not illustrate all of the connections of data unit **110**. In particular various control lines and the like have been omitted for the sake of clarity. Therefore FIG. **5** should be read with the following description for a complete understanding of the operation of data unit **110**. Data unit **110** includes a number of parts advantageously operating in parallel. Data unit **110** includes eight 32 bit data registers **200** designated D7–D0. Data register D0 may be used as a general purpose register but in addition has special functions when used with certain instructions. Data registers **200** include multiple read and write ports connected to data unit buses **201** to **206** and to local port data bus Lbus **103**, global port source data bus Gsrc **105** and global port destination data bus Gdst **107**. Data registers **200** may also be read "sideways" in a manner described as a rotation register that will be further described below. Data unit **110** further includes a status register **210** and a multiple flags register **211**, which stores arithmetic logic unit resultant status for use in certain instructions. Data unit **110** includes as its major computational components a hardware multiplier **220** and a three input arithmetic logic unit **230**. Lastly, data unit **110** includes: multiplier first input bus **201**, multiplier second input bus **202**, multiplier destination bus **203**, arithmetic logic unit destination bus **204**, arithmetic logic unit first input bus **205**, arithmetic logic unit second input bus **206**; buffers **104**, **106**, **108** and **236**; multiplexers Rmux **221**, Imux **222**, MSmux **225**, Bmux **227**, Amux **232**, Smux **231**, Cmux **233** and Mmux **234**; and product left shifter **224**, adder **226**, barrel rotator **235**, LMO/RMO/LMBC/RMBC circuit **237**, expand circuit **238**, mask generator **239**, input A bus **241**, input B bus **242**, input C bus **243**, rotate bus **244**, function signal generator **245**, bit **0** carry-in generator **246**, and instruction decode logic **250**, all of which will be further described below.

The following description of data unit **110** as well as further descriptions of the use of each digital image/graphics processor **71**, **72**, **73** and **74** employ several symbols for ease of expression. Many of these symbols are standard mathematical operations that need no explanation. Some are logical operations that will be familiar to one skilled in the art, but whose symbols may be unfamiliar. Lastly, some symbols refer to operations unique to this invention. Table 1 lists some of these symbols and their corresponding operation.

TABLE 1

Symbol	Operation
~	bit wise NOT
&	bit wise AND
|	bit wise OR
^	bit wise exclusive OR
@	multiple flags register expand
%	mask generation
%!	modified mask generation
\\	rotate left
<<	shift left
>>u	shift right zero extend
>>s	shift right sign extend
>>	shift right sign extend
	default case
||	parallel operation
*(A±X)	memory contents at address base register A ± index register X or offset X
&*(A±X)	address unit arithmetic address base register A ± index register X or offset X
*(A±[X])	memory contents at address base register A ± scaled index register X or offset X

The implications of the operations listed above in Table 1 may not be immediately apparent. These will be explained in detail below.

FIG. **6** illustrates the field definitions for status register **210**. Status register **210** may be read from via global port source data bus Gsrc **105** or written into via global port destination data bus Gdst bus **107**. In addition, status register **210** may write to or load from a specified one of data registers **200**. Status register **210** is employed in control of operations within data unit **110**.

Status register **210** stores four arithmetic logic unit result status bits "N", "C", "V" and "Z". These are individually described below, but collectively their setting behavior is as follows. Note that the instruction types listed here will be fully described below. For instruction words including a 32 bit immediate fields, if the condition code field is "unconditional" then all four status bits are set according to the result of arithmetic logic unit **230**. If the condition code field specifies a condition other than "unconditional", then no status bits are set, whether or not the condition is true. For instruction words not including a 32 bit immediate field operations and not including conditional operations fields, all status bits are set according to the result of arithmetic logic unit **230**. For instruction words not including a 32 bit immediate field that permit conditional operations, if the condition field is "unconditional", or not "unconditional" and the condition is true, instruction word bits **28–25** indicate which status bits should be protected. All unprotected bits are set according to the result of arithmetic logic unit **230**. For instruction words not including a 32 bit immediate field, which allow conditional operations, if the condition field is not "unconditional" and the condition is false, no status bits are set. There is no difference in the status setting behavior for Boolean operations and arithmetic operations. As will be further explained below, this behavior, allows the

conditional instructions and source selection to perform operations that would normally require a branch.

The arithmetic logic unit result bits of status register **210** are as follows. The "N" bit (bit **31**) stores an indication of a negative result. The "N" bit is set to "1" if the result of the last operation of arithmetic logic unit **230** was negative. This bit is loaded with bit **31** of the result. In a multiple arithmetic logic unit operation, which will be explained below, the "N" bit is set to the AND of the zero compares of the plural sections of arithmetic logic unit **230**. In a bit detection operation performed by LMO/RMO/LMBC/RMBC circuit **237**, the "N" bit is set to the AND of the zero compares of the plural sections of arithmetic logic unit **230**. Writing to this bit in software overrides the normal arithmetic logic unit result writing logic.

The "C" bit (bit **30**) stores an indication of a carry result. The "C" bit is set to "1" if the result of the last operation of arithmetic logic unit **230** caused a carry-out from bit **31** of the arithmetic logic unit. During multiple arithmetic and bit detection, the "C" bit is set to the OR of the carry outs of the plural sections of arithmetic logic unit **230**. Thus the "C" bit is set to "1" if at least one of the sections has a carry out. Writing to this bit in software overrides the normal arithmetic logic unit result writing logic.

The "V" bit (bit **29**) stores an indication of an overflow result. The "V" bit is set to "1" if the result of the last operation of arithmetic logic unit **230** created an overflow condition. This bit is loaded with the exclusive OR of the carry-in and carry-out of bit **31** of the arithmetic logic unit **230**. During multiple arithmetic logic unit operation the "V" bit is the AND of the carry outs of the plural sections of arithmetic logic unit **230**. For left most one and right most one bit detection, the "V" bit is set to "1" if there were no "1's" in the input word, otherwise the "V" bit is set to "0". For left most bit change and right most bit change bit detection, the "V" bit is set to "1" is all the bits of the input are the same, or else the "V" bit is set to "0". Writing to this bit in software overrides the normal arithmetic logic unit result writing logic.

The "Z" bit (bit **28**) stores and indication of a "0" result. The "Z" bit is set to "1" if the result of the last operation of arithmetic logic unit **230** produces a "0" result. This "Z" bit is controlled for both arithmetic operations and logical operations. In multiple arithmetic and bit detection operations, the "Z" bit is set to the OR of the zero compares of the plural sections of arithmetic logic unit **230**. Writing to this bit in software overrides the normal arithmetic logic unit result writing logic circuitry.

The "R" bit (bit **6**) controls bits used by expand circuit **238** and rotation of multiple flags register **211** during instructions that use expand circuit **238** to expand portions of multiple flags register **211**. If the "R" bit is "1", then the bits used in an expansion of multiple flags register **211** via expand circuit **238** are the most significant bits. For an operation involving expansion of multiple flags register **211** where the arithmetic logic unit function modifier does not specify multiple flags register rotation, then multiple flags register **211** is "post-rotated left" according to the "Msize" field. If the arithmetic logic unit function modifier does specify multiple flags register rotation, then multiple flags register **211** is rotated according to the "Asize" field. If the "R" bit is "0", then expand circuit **238** employs the least significant bits of multiple flags register **211**. No rotation takes place according to the "Msize" field. However, the arithmetic logic unit function modifier may specify rotation by the "Asize" field.

The "Msize" field (bits **5–3**) indicates the data size employed in certain instruction classes that supply mask data from multiple flags register **211** to the C-port of arithmetic logic unit **230**. The "Msize" field determines how many bits of multiple flags register **211** uses to create the mask information. When the instruction does not specify rotation corresponding to the "Asize" field and the "R" bit is "1", then multiple flags register **211** is automatically "post-rotated left" by an amount set by the "Msize" field. Codings for these bits are shown in Table 2.

TABLE 2

Msize Field			Data Size	Multiple Flags Register			
				Rotate	No. of	Bit(s) used	
5	4	3	bits	amount	bits used	R=1	R=0
0	0	0	0	64	64	—	—
0	0	1	1	32	32	31–0	31–0
0	1	0	2	16	16	31–16	15–0
0	1	1	4	8	8	31–24	7–0
1	0	0	8	4	4	31–28	3–0
1	0	1	16	2	2	31–30	1–0
1	1	0	32	1	1	31	0
1	1	1	64	0	0	—	—

As noted above, the preferred embodiment supports "Msize" fields of "100", "101" and "110" corresponding to data sizes of 8, 16 and 32 bits, respectively. Note that rotation for an "Msize" field of "001" results in no change in data output. "Msize" fields of "001", "010" and "011" are possible useful alternatives. "Msize" fields of "000" and "111" are meaningless but may be used in an extension of multiple flags register **211** to 64 bits.

The "Asize" field (bits **2–0**) indicate the data size for multiple operations performed by arithmetic logic unit **230**. Arithmetic logic unit **230** preferably includes 32 parallel bits. During certain instructions arithmetic logic unit **230** splits into multiple independent sections. This is called a multiple arithmetic logic unit operation. This splitting of arithmetic logic unit **230** permits parallel operation on pixels of less than 32 bits that are packed into 32 bit data words. In the preferred embodiment arithmetic logic unit **230** supports: a single 32 bit operation; two sections of 16 bit operations; and four sections of 8 bit operations. These options are called word, half-word and byte operations.

The "Asize" field indicates: the number of multiple sections of arithmetic logic unit **230**; the number of bits of multiple flags register bits **211** set during the arithmetic logic unit operation, which is equal in number to the number of sections of arithmetic logic unit **230**; and the number of bits the multiple flags register should "post-rotate left" after output during multiple arithmetic logic unit operation. The rotation amount specified by the "Asize" field dominates over the rotation amount specified by the "Msize" field and the "R" bit when the arithmetic logic unit function modifier indicates multiple arithmetic with rotation. Codings for these bits are shown in Table 3. Note that while the current preferred embodiment of the invention supports multiple arithmetic of one 32 bit section, two 16 bit sections and four 8 bit sections the coding of the "Asize" field supports specification of eight sections of 4 bits each, sixteen sections of 2 bits each and thirty-two sections of 1 bit each. Each of these additional section divisions of arithmetic logic unit **230** are feasible. Note also that the coding of the "Asize" field further supports specification of a 64 bit data size for possible extension of multiple flags register **211** to 64 bits.

TABLE 3

Asize Field			Data Size	Multiple Flags Register		
2	1	0	bits	Rotate amount	No. of bits set	Bit(s) set
0	0	0	0	64	64	—
0	0	1	1	32	32	31–0
0	1	0	2	16	16	15–0
0	1	1	4	8	8	7–0
1	0	0	8	4	4	3–0
1	0	1	16	2	2	1–0
1	1	0	32	1	1	0
1	1	1	64	0	0	—

The "Msize" and "Asize" fields of status register **210** control different operations. When using the multiple flags register **211** as a source for producing a mask applied to the C-port of arithmetic logic unit **230**, the "Msize" field controls the number of bits used and the rotate amount. In such a case the "R" bit determines whether the most significant bits or least significant bits are employed. When using the multiple flags register **211** as a destination for the status bits corresponding to sections of arithmetic logic unit **230**, then the "Asize" field controls the number and identity of the bits loaded and the optional rotate amount. If a multiple arithmetic logic unit operation with "Asize" field specified rotation is specified with an instruction that supplies mask data to the C-port derived from multiple flags register **211**, then the rotate amount of the "Asize" field dominates over the rotate amount of the combination of the "R" bit and the "Msize" field.

The multiple flags register **211** is a 32 bit register that provides mask information to the C-port of arithmetic logic unit **230** for certain instructions. Global port destination data bus Gdst bus **107** may write to multiple flags register **211**. Global port source bus Gsrc may read data from multiple flags register **211**. In addition multiple arithmetic logic unit operations may write to multiple flags register **211**. In this case multiple flags register **211** records either the carry or zero status information of the independent sections of arithmetic logic unit **230**. The instruction executed controls whether the carry or zero is stored.

The "Msize" field of status register **210** controls the number of least significant bits used from multiple flags register **211**. This number is given in Table 2 above. The "R" bit of status register **210** controls whether multiple flags register **211** is pre-rotated left prior to supply of these bits. The value of the "Msize" field determines the amount of rotation if the "R" bit is "1". The selected data supplies expand circuit **238**, which generates a 32 bit mask as detailed below.

The "Asize" field of status register **210** controls the data stored in multiple flags register **211** during multiple arithmetic logic unit operations. As previously described, in the preferred embodiment arithmetic logic unit **230** may be used in one, two or four separate sections employing data of 32 bits, 16 bits and 8 bits, respectively. Upon execution of a multiple arithmetic logic unit operation, the "Asize" field indicates through the defined data size the number of bits of multiple flags register **211** used to record the status information of each separate result of the arithmetic logic unit. The bit setting of multiple flags register **211** is summarized in Table 4.

TABLE 4

Data Size	ALU carry-out bits setting MF bits				ALU result bits equal to zero setting MF bits			
bits	3	2	1	0	3	2	1	0
8	31	23	15	7	31–24	23–16	15–8	7–0
16	—	—	31	15	—	—	31–16	15–0
32	—	—	—	31	—	—	—	31–0

Note that Table 4 covers only the cases for data sizes of 8, 16 and 32 bits. Those skilled in the art would easily realize how to extend Table 4 to cover the cases of data sizes of 64 bits, 4 bits, 2 bits and 1 bit. Also note that the previous discussion referred to storing either carry or zero status in multiple flags register **211**. It is also feasible to store other status bits such as negative and overflow.

Multiple flags register **211** may be rotated left a number of bit positions upon execution of each arithmetic logic unit operation. The rotate amount is given above. When performing multiple arithmetic logic unit operations, the result status bit setting dominates over the rotate for those bits that are being set. When performing multiple arithmetic logic unit operations, an alternative to rotation is to clear all the bits of multiple flags register **211** not being set by the result status. This clearing is after generation of the mask data if mask data is used in that instruction. If multiple flags register **211** is written by software at the same time as recording an arithmetic logic unit result, then the preferred operation is for the software write to load all the bits. Software writes thus dominate over rotation and clearing of multiple flags register **211**.

FIG. **7** illustrates the splitting of arithmetic logic unit **230** into multiple sections. As illustrated in FIG. **7**, the 32 bits of arithmetic logic unit **230** are separated into four sections of eight bits each. Section **301** includes arithmetic logic unit bits **7–0**, section **302** includes bits **15–8**, section **303** includes bits **23–16** and section **304** includes bits **31–24**. Note that FIG. **7** does not illustrate the inputs or outputs of these sections, which are conventional, for the sake of clarity. The carry paths within each of the sections **301**, **302**, **303** and **303** are according to the known art.

Multiplexers **311**, **312** and **313** control the carry path between sections **301**, **302**, **303** and **304**. Each of these multiplexers is controlled to select one of three inputs. The first input is a carry look ahead path from the output of the previous multiplexer, or in the case of the first multiplexer **311** from bit **0** carry-in generator **246**. Such carry look ahead paths and their use are known in the art and will not be further described here. The second selection is the carry-out from the last bit of the corresponding section of arithmetic logic unit **230**. The final selection is the carry-in signal from bit **0** carry-in generator **246**. Multiplexer **314** controls the output carry path for arithmetic logic unit **230**. Multiplexer **314** selects either the carry look ahead path from the carry-out selected by multiplexer **313** or the carry-out signal for bit **31** from section **304**.

Multiplexers **311**, **312**, **313** and **314** are controlled based upon the selected data size. In the normal case arithmetic logic unit **230** operates on 32 bit data words. This is indicated by an "Asize" field of status register **210** equal to "110". In this case multiplexer **311** selects the carry-out from bit **7**, multiplexer **312** selects the carry-out from bit **15**, multiplexer **313** selects the carry-out from bit **23** and multiplexer **314** selects the carry-out from bit **31**. Thus the four sections **301**, **302**, **303** and **304** are connected together into a single 32 bit arithmetic logic unit. If status register **210**

selected a half-word via an "Asize" field of "101", then multiplexer **311** selects the carry-out from bit **7**, multiplexer **312** selects the carry-in from bit **0** carry-in generator **246**, multiplexer **313** selects the carry-out from bit **23** and multiplexer **314** selects the carry-out from bit **31**. Sections **301** and **302** are connected into a 16 bit unit and sections **303** and **304** are connected into a 16 bit unit. Note that multiplexer **312** selects the bit **0** carry-in signal for bit **16** just like bit **0**, because bit **16** is the first bit in a 16 bit half-word. If status register **210** selected a byte via an "Asize" field of "100", then multiplexers **311**, **312** and **313** select the carry-in from bit **0** carry-in generator **246**. Sections **301**, **302**, **303** and **304** are split into four independent 8 bit units. Note that selection of the bit **0** carry-in signal at each multiplexer is proper because bits **8**, **16** and **24** are each the first bit in an 8 bit byte.

FIG. **7** further illustrates zero resultant detection. Each 8 bit zero detect circuit **321**, **322**, **323** and **324** generates a "1" output if the resultant from the corresponding 8 bit section is all zeros "00000000". AND gate **331** is connected to 8 bit zero detect circuits **321** and **322**, thus generating a when all sixteen bits **15–0** are "0". AND gate **332** is similarly connected to 8 bit zero detect circuits **321** and **322** for generating a "1" when all sixteen bits **31–16** are "0". Lastly, AND gate **341** is connected to AND gates **331** and **332**, and generates a "1" when all 32 bits **31–0** are "0".

During multiple arithmetic logic unit operations multiple flags register **211** may store either carry-outs or the zero comparison, depending on the instruction. These stored resultants control masks to the C-port during later operations. Table 4 shows the source for the status bits stored. In the case in which multiple flags register **211** stores the carry-out signal(s), the "Asize" field of status register **210** determines the identity and number of carry-out signals stored. If the "Asize" field specifies word operations, then multiple flags register **211** stores a single bit equal to the carry-out signal of bit **31**. If the "Asize" field specifies half-word operations, then multiple flags register **211** stores two bits equal to the carry-out signals of bits **31** and **15**, respectfully. If the "Asize" field specifies byte operations, then multiple flags register **211** stores four bits equal to the carry-out signals of bits **31**, **23**, **15** and **7**, respectively. The "Asize" field similarly controls the number and identity of zero resultants stored in multiple flags register **211** when storage of zero resultants is selected. If the "Asize" field specifies word operations, then multiple flags register **211** stores a single bit equal to output of AND gate **341** indicating if bits **31–0** are "0". If the "Asize" field specifies half-word operations, then multiple flags register **211** stores two bits equal to the outputs of AND gates **331** and **332**, respectfully. If the "Asize" field specifies byte operations, then multiple flags register **211** stores four bits equal to the outputs of 8 bit zero detect circuits **321**, **322**, **323** and **324**, respectively.

It is technically feasible and within the scope of this invention to allow further multiple operations of arithmetic logic unit **230** such as: eight sections of 4 bit operations; sixteen sections 2 bit operations; and thirty-two sections single bit operations. Note that both the "Msize" and the "Asize" fields of status register **210** include coding to support such additional multiple operation types. Those skilled in the art can easily modify and extend the circuits illustrated in FIG. **7** using additional multiplexers and AND gates. These latter feasible options are not supported in the preferred embodiment due to the added complexity in construction of arithmetic logic unit **230**. Note also that this technique can be extended to a data processing apparatus employing 64 bit data and that the same teachings enable such an extension.

Data registers **200**, designated data registers D7–D0 are connected to local port data bus Lbus **103**, global port source data bus Gsrc **105** and global port destination data bus Gdst **107**. Arrows within the rectangle representing data registers **200** indicate the directions of data access. A left pointing arrow indicates data recalled from data registers **200**. A right pointing arrow indicates data written into data registers **200**. Local port data bus Lbus **103** is bidirectionally coupled to data registers **200** as a data source or data destination. Global port destination data bus Gdst **107** is connected to data registers **200** as a data source for data written into data registers **200**. Global port source data bus Gsrc **107** is connected to data registers **200** as a data destination for data recalled from data registers **200** in both a normal data register mode and in a rotation register feature described below. Status register **210** and multiple flags register **211** may be read from via global port source data bus Gsrc **106** and written into via global port destination data bus Gdst **107**. Data registers **200** supply data to multiplier first input bus **201**, multiplier second input bus **202**, arithmetic logic unit first input bus **205** and arithmetic logic unit second input bus **206**. Data registers **200** are connected to receive input data from multiplier destination bus **203** and arithmetic logic unit destination bus **204**.

Data registers **200**, designated registers D7–D0, are connected to form a 256 bit rotate register as illustrated in FIG. **8**. This rotate register is collectively designated rotation (ROT) register ROT **208**. This forms a 256 bit register comprising eight 32 bit rotation registers ROT0, ROT1, . . . ROT7. FIG. **8** illustrates in part the definitions of the rotation registers ROT0, ROT1, . . . ROT7. These rotation registers are defined sideways with respect to data registers D7–D0. The rotation register **208** may be rotated by a non-arithmetic logic unit instruction DROT, as described below. During this rotation the least significant bit of data register D7 rotates into the most significant bit of data register D6, etc. The least significant bit of data register D0 is connected back to the most significant bit of data register D7. ROT register **208** may be read in four 8 bit bytes at a time. The four 8 bit bytes are respective octets of bits having the same bit number in each of data registers **200** as shown below in Table 5 and illustrated in FIG. **8**.

TABLE 5

Rotation Register bits	Octet of bits from each D7–D0 Bit
31–24	24
23–16	16
15–8	8
7–0	0

When a DROT instruction is executed the 256 bit rotation register **208** is rotated right one bit place. The least significant bit **0** of each byte A, B, C, D of each register such as D7 is mapped as shown to a particular bit number of the ROT register output onto the global port source data bus Gsrc **105**. ROT register **208** is read only in the preferred embodiment, but can be writable in other embodiments.

ROT register **208** is useful in image rotations, orthogonal transforms and mirror transforms. Performing 32 bit stores to memory from the rotation register **208** in parallel with eight DROT instructions rotates four 8 by 8 bit patches of data clockwise ninety degrees. The rotated data is stored in the target memory locations. Various combinations of register loading, memory address storing, and data size alteration, can enable a variety of clockwise and counter-

clockwise rotations of 8 by 8 bit patches to be performed. Rotation of larger areas can then be performed by moving whole bytes. This remarkable orthogonal structure that provides register file access to registers D7–D0 in one mode, and rotation register access in the DROT operation, is only slightly more complex than a register file alone.

The data register D0 has a dual function. It may be used as a normal data register in the same manner as the other data registers D7–D1. Data register D0 may also define certain special functions when executing some instructions. Some of the bits of the most significant half-word of data register D0 specifies the operation of all types of extended arithmetic logic unit operations. Some of the bits of the least significant half-word of data register D0 specifies multiplier options during a multiple multiply operation. The 5 least significant bits of data register D0 specify a default barrel rotate amount used by certain instruction classes. FIG. 9 illustrates the contents of data register D0 when specifying data unit 110 operation.

The "FMOD" field (bits 31–28) of data register D0 allow modification of the basic operation of arithmetic logic unit 230 when executing an instruction calling for an extended arithmetic logic unit (EALU) operation. Table 6 illustrates these modifier options. Note, as indicated in Table 6, certain instruction word bits in some instruction formats are decoded as function modifiers in the same fashion. These will be further discussed below.

TABLE 6

Function Modifier Code				Modification Performed
0	0	0	0	normal operation
0	0	0	1	cin
0	0	1	0	%! if mask generation instruction LMO if not mask generation instruction
0	0	1	1	(%! and cin) if mask generation instruction RMO if not mask generation instruction
0	1	0	0	A-port=0
0	1	0	1	A-port=0 and cin
0	1	1	0	(A-port=0 and %!) if mask generation instruction LMBC if not mask generation instruction
0	1	1	1	(A-port=0 and %! and cin) if mask generation instruction RMBC if not mask generation instruction
1	0	0	0	Multiple arithmetic logic unit operations, carry-out(s) —> multiple flags register
1	0	0	1	Multiple arithmetic logic unit operations, zero result(s) —> multiple flags register
1	0	1	0	Multiple arithmetic logic unit operations, carry-out(s) —> multiple flags register, rotate by "Asize" field of status register
1	0	1	1	Multiple arithmetic logic unit operations, zero results(s) —> multiple flags register, rotate by "Asize" field of status register
1	1	0	0	Multiple arithmetic logic unit operations, carry-out(s) —> multiple flags register, clear multiple flags register
1	1	0	1	Multiple arithmetic logic unit operations, zero result(s) —> multiple flags register, clear multiple flags register
1	1	1	0	Reserved
1	1	1	1	Reserved

Instruction word bit	Data Register D0 bit
52	28
54	29
56	30
58	31

The modified operations listed in Table 6 are explained below. If the "FMOD" field is "0000", the normal, unmodified operation results. The modification "cin" causes the carry-in to bit 0 of arithmetic logic unit 230 to be the "C" bit of status register 210. This allows add with carry, subtract with borrow and negate with borrow operations. The modification "%!" works with mask generation. When the "%!" modification is active mask generator 239 effectively generates all "1's" for a zero rotate amount rather than all "0's". This function can be implemented by changing the mask generated by mask generator 239 or by modifying the function of arithmetic logic unit 230 so that mask of all "0's" supplied to the C-port operates as if all "1's" were supplied. This modification is useful in some rotate operations. The modifications "LMO", "RMO", "LMBC" and "RMBC" designate controls of the LMO/RMO/LMBC/RMBC circuit 237. The modification "LMO" finds the left most "1" of the second arithmetic input. The modification "RMO" finds the right most "1". The modification "LMBC" finds the left most bit that differs from the sign bit (bit 31). The "RMBC" modification finds the right most bit that differs from the first bit (bit 0). Note that these modifications are only relevant if the C-port of arithmetic logic unit 230 does not receive a mask from mask generator 239. The modification "A-port=0" indicates that the input to the A-port of arithmetic logic unit 230 is effectively zeroed. This may take place via multiplexer Amux 232 providing a zero output, or the operation of arithmetic logic unit 230 may be altered in a manner having the same effect. An "A-port=0" modification is used in certain negation, absolute value and shift right operations. A "multiple arithmetic logic unit operation" modification indicates that one or more of the carry paths of arithmetic logic unit 230 are severed, forming in effect two or more independent arithmetic logic units operating in parallel. The "Asize" field of status register 210 controls the number of such multiple arithmetic logic unit sections. The multiple flags register 211 stores a number of status bits equal to the number of sections of the multiple arithmetic logic unit operations. In the "carry-out(s)→multiple flags" modification, the carry-out bit or bits are stored in multiple flags register 211. In the "zero result(s)→multiple flags" modification, an indication of the zero resultant for the corresponding arithmetic logic unit section is stored in multiple flags register 211. This process is described above together with the description of multiple flags register 211. During this storing operation, bits within multiple flags register 211 may be rotated in response to the "rotate" modification or cleared in response to the "clear" modification. These options are discussed above together with the description of multiple flags register 211.

The "A" bit (bit 27) of data register D0 controls whether arithmetic logic unit 230 performs an arithmetic or Boolean logic operation during an extended arithmetic logic unit operation. This bit is called the arithmetic enable bit. If the "A" bit is "1", then an arithmetic operation is performed. If the "A" bit is "0", then a logic operation is performed. If the "A" bit is "0", then the carry-in from bit 0 carry-in generator 246 into bit 0 of the arithmetic logic unit 230 is generally "0". As will be further explained below, certain extended arithmetic logic unit operations may have a carry-in bit of "1" even when the "A" bit is "0" indicating a logic operation.

The "EALU" field (bits 19–26) of data register D0 defines an extended arithmetic logic unit operation. The eight bits of the "EALU" field specify the arithmetic logic unit function control bits used in all types of extended arithmetic logic unit operations. These bits become the control signals to arithmetic logic unit 230. They may be passed to arithmetic logic unit 230 directly, or modified according to the "FMOD" field. In some instructions the bits of the "EALU" field are inverted, leading to an "EALUF" or extended

arithmetic logic unit false operation. In this case the eight control bits supplied to arithmetic logic unit **230** are inverted.

The "C" bit (bit **18**) of data register **D0** designates the carry-in to bit **0** of arithmetic logic unit **230** during extended arithmetic logic unit operations. The carry-in value into bit **0** of the arithmetic logic unit during extended arithmetic logic unit operations is given by this "C" bit. This allows the carry-in value to be specified directly, rather than by a formula as for non-EALU operations.

The "I" bit (bit **17**) of data register **D0** is designated the invert carry-in bit. The "I" bit, together with the "C" bit and the "S" bit (defined below), determines whether or not to invert the carry-in into bit **0** of arithmetic logic unit **230** when the function code of an arithmetic logic unit operation are inverted. This will be further detailed below.

The "S" bit (bit **16**) of data register **D0** indicates selection of sign extend. The "S" bit is used when executing extended arithmetic logic unit operations ("A" bit=1). If the "S" bit is "1", then arithmetic logic unit control signals **F3–F0** (produced from bits **22–19**) should be inverted if the sign bit (bit **31**) of the data first arithmetic logic unit input bus **206** is "0", and not inverted if this sign bit is "1" The effect of conditionally inverting arithmetic logic unit control signals **F3–F0** will be explained below. Such an inversion is useful to sign extend a rotated input in certain arithmetic operations. If the extended arithmetic logic unit operation is Boolean ("A" bit=0), then the "S" bit is ignored and the arithmetic logic unit control signals **F3–F0** are unchanged.

Table 7 illustrates the interaction of the "C", "I" and "S" bits of data register **D0**. Note that an "X" entry for either the "I" bit or the first input sign indicates that bit does not control the outcome, i.e. a "don't care" condition.

TABLE 7

S	I	First Input Sign	Invert C?	Invert F3–F0
0	X	X	no	no
1	0	0	no	no
1	0	1	no	yes
1	1	0	no	no
1	1	1	yes	yes

If the "S" bit equals "1" and the sign bit of the first input destined for the B-port of arithmetic logic unit **230** equals "0", then the value of the carry-in to bit **0** of arithmetic logic unit **230** set by the "C" bit value can optionally be inverted according to the value of the "I" bit. This allows the carry-in to be optionally inverted or not, based on the sign of the input. Note also that arithmetic logic unit control signals **F3–F0** are optionally inverted based on the sign of the input, if the "S" bit is "1". This selection of inversion of arithmetic logic unit control signals **F3–F0** may be overridden by the "FMOD" field. If the "FMOD" field specifies "Carry-in= Status Register's Carry bit", then the carry-in equals the "C" bit of status register **210** whatever the value of the "S" and "I" bits. Note also that the carry-in for bit **0** of arithmetic logic unit **230** may be set to "1" via the "C" bit for extended arithmetic logic unit operations even if the "A" bit is "0" indicating a Boolean operation.

The "N" bit (bit **15**) of data register **D0** is used when executing a split or multiple section arithmetic logic unit operation. This "N" bit is called the non-multiple mask bit. For some extended arithmetic logic unit operations that specify multiple operation via the "FMOD" field, the instruction specifies a mask to be passed to the C-port of arithmetic logic unit **230** via mask generator **239**. This "N" bit determines whether or not the mask is split into the same number of sections as arithmetic logic unit **230**. Recall that the number of such multiple sections is set by the "Asize" field of status register **210**. If the "N" bit is "0", then the mask is split into multiple masks. If the "N" bit is "1", then mask generator **239** produces a single 32 bit mask.

The "E" bit (bit **14**) designates an explicit multiple carry-in. This bit permits the carry-in to be specified at run time by the input to the C-port of arithmetic logic unit **230**. If both the "A" bit and the "E" bit are "1" and the "FMOD" field does not designate the cin function, then the effects of the "S", "I" and "C" bits are annulled. The carry input to each section during multiple arithmetic is taken as the exclusive OR of the least significant bit of the corresponding section input to the C-port and the function signal **F0**. If multiple arithmetic is not selected the single carry-in to bit **0** of arithmetic logic unit **230** is the exclusive OR of the least significant bit (bit **0**) the input to the C-port and the function signal **F0**. This is particularly useful for performing multiple arithmetic in which differing functions are performed in different sections. One extended arithmetic logic unit operation corresponds to (A^B)&C|(A^~B)&C. Using a mask for the C-port input, a section with all "0's" produces addition with the proper carry-in of "0" and a section of all "1's" produces subtraction with the proper carry-in of "1".

The "DMS" field (bits **12–8**) of data register **D0** defines the shift following the multiplier. This shift takes place in product left shifter **224** prior to saving the result or passing the result to rounding logic. During this left shift the most significant bits shifted out are discarded and zeroes are shifted into the least significant bits. The "DMS" field is effective during any multiply/extended arithmetic logic unit operation. In the preferred embodiment data register **D0** bits **9–8** select 0, 1, 2 or 3 place left shifting. Table 8 illustrates the decoding.

TABLE 8

DMS field		
9	8	Left shift amount
0	0	0
0	1	1
1	0	2
1	1	3

The "DMS" field includes 5 bits that can designate left shift amounts from 0 to 31 places. In the preferred embodiment product left shifter **224** is limited to shifts from 0 to 3 places for reasons of size and complexity. Thus bits **12–10** of data register **D0** are ignored in setting the left shift amount. However, it is feasible to provide a left shift amount within the full range from 0 to 31 places from the "DMS" field if desired.

The "M" bit (bit **7**) of data register **D0** indicates a multiple multiply operation. Multiplier **220** can multiply two 16 bit numbers to generate a 32 bit result or of simultaneously multiplying two pair of 8 bit numbers to generate a pair of 16 bit resultants. This "M" bit selects either a single 16 by 16 multiply if "M"="0", or two 8 by 8 multiplies if "M"= "1". This operation is similar to multiple arithmetic logic unit operations and will be further described below.

The "R" bit (bit **6**) of data register **D0** specifies whether a rounding operation takes place on the resultant from multiplier **220**. If the "R" bit is "1", the a rounding operation, explained below together with the operation of multiplier **220**, takes place. If the "R" bit is "0", then no rounding takes place and the 32 bit resultant form multiplier **220** is written into the destination register. Note that use of a predetermined

bit in data register D0 is merely a preferred embodiment for triggering this mode. It is equally feasible to enable the rounding mode via a predetermined instruction word bit.

The "DBR" field (bits 4–0) of data register D0 specifies a default barrel rotate amount used barrel rotator 235 during certain instructions. The "DBR" field specifies the number of bit positions that barrel rotator 235 rotates left. These 5 bits can specify a left rotate of 0 to 31 places. The value of the "DBR" field may also be supplied to mask generator 239 via multiplexer Mmux 234. Mask generator 239 forms a mask supplied to the C-port of arithmetic logic unit 230. The operation of mask generator 239 will be discussed below.

Multiplier 220 is a hardware single cycle multiplier. As described above, multiplier 220 operates to multiply a pair of 16 bit numbers to obtain a 32 bit resultant or to multiply two pairs of 8 bit numbers to obtain two 16 bit resultants in the same 32 bit data word.

FIGS. 10*a*, 10*b*, 10*c* and 10*d* illustrate the input and output data formats for multiplying a pair of 16 bit numbers. FIG. 10*a* shows the format of a signed input. Bit 15 indicates the sign of this input, a "0" for positive and a "1" for negative. Bits 0 to 14 are the magnitude of the input. Bits 16 to 31 of the input are ignored by the multiply operation and are shown as a don't care "X". FIG. 10*b* illustrates the format of the resultant of a signed by signed multiply. Bits 31 and 30 are usually the same and indicate the sign of the resultant. if the multiplication was of Hex "8000" by Hex "8000", then bits 31 and 30 become "01". FIG. 10*c* illustrates the format of an unsigned input. The magnitude is represented by bits 0 to 15, and bits 16 to 31 are don't care "X". FIG. 10*d* shows the format of the resultant of an unsigned by unsigned multiply. All 32 bits represent the resultant.

FIG. 11 illustrates the input and output data formats for multiplying two pair of 8 bit numbers. In each of the two 8 bit by 8 bit multiplies the two first inputs on multiplier first input bus 201 are always unsigned. The second inputs on multiplier second input bus 202 may be both signed, resulting in two signed products, or both unsigned, resulting in two unsigned products. FIG. 11*a* illustrates the format of a pair of signed inputs. The first signed input occupies bits 0 to 7. Bit 7 is the sign bit. The second signed input occupies bits 8 to 15, bit 15 being the sign bit. FIG. 11*b* illustrates the format of a pair of unsigned inputs. Bits 0 to 7 form the first unsigned input and bits 8 to 16 form the second unsigned input. FIG. 11*c* illustrates the format of a pair of signed resultants. As noted above, a dual unsigned by signed multiply operation produces such a pair of signed resultants. The first signed resultant occupies bits 0 to 15 with bit 15 being the sign bit. The second signed resultant occupies bits 16 to 31 with bit 31 being the sign bit. FIG. 11*d* illustrates the format of a pair of unsigned resultants. The first unsigned resultant occupies bits 1 to 15 and the second unsigned resultant occupies bits 16 to 31.

Multiplier first input bus 201 is a 32 bit bus sourced from a data register within data registers 200 selected by the instruction word. The 16 least significant bits of multiplier first input bus 201 supplies a first 16 bit input to multiplier 220. The 16 most significant bits of multiplier first input bus 201 supplies the 16 least significant bits of a first input to a 32 bit multiplexer Rmux 221. This data routing is the same for both the 16 bit by 16 bit multiply and the dual 8 bit by 8 bit multiply. The 5 least significant bits multiplier first input bus 201 supply a first input to a multiplexer Smux 231.

Multiplier second input bus 202 is a 32 bit bus sourced from one of the data registers 200 as selected by the instruction word or from a 32 bit, 5 bit or 1 bit immediate value imbedded in the instruction word. A multiplexer Imux 222 supplies such an immediate multiplier second input bus 202 via a buffer 223. The instruction word controls multiplexer imux 222 to supply either 32 bits, 5 bits or 1 bit from an immediate field of the instruction word to multiplier second input bus 202 when executing an immediate instruction. The short immediate fields are zero extended in multiplexer Imux 222 upon supply to multiplier second input bus 202. The 16 least significant bits of multiplier second input bus 202 supplies a second 16 bit input to multiplier 220. This data routing is the same for both the 16 bit by 16 bit multiply and the dual 8 bit by 8 bit multiply. Multiplier second input bus 202 further supplies one input to multiplexer Amux 232 and one input to multiplexer Cmux 233. The 5 least significant bits of multiplier second input bus 202 supply one input to multiplexer Mmux 234 and a second input to multiplexer Smux 231.

The output of multiplier 220 supplies the input of product left shifter 224. Product left shifter 224 can provide a controllable left shift of 3, 2, 1 or 0 bits. The output of multiply shift multiplexer MSmux 225 controls the amount of left shift of product left shifter 224. Multiply shift multiplexer MSmux 225 selects either bits 9–8 from the "DMS" field of data register D0 or all zeroes depending on the instruction word. In the preferred embodiment, multiply shift multiplexer MSmux 225 selects the "0" input for the instructions MPYx||ADD and MPYx||SUB. These instructions combine signed or unsigned multiplication with addition or subtractions using arithmetic logical unit 230. In the preferred embodiment, multiply shift multiplexer MSmux 225 selects bits 9–8 of data register D0 for the instructions MPYx||EALUx. These instructions combine signed or unsigned multiplication with one of two types of extended arithmetic logic unit instructions using arithmetic logic unit 230. The operation of data unit 110 when executing these instructions will be further described below. Product left shifter 224 discards the most significant bits shifted out and fills the least significant bits shifted in with zeros. Product left shifter 224 supplies a 32 bit output connected to a second input of multiplexer Rmux 221.

FIG. 12 illustrates internal circuits of multiplier 220 in block diagram form. The following description of multiplier 220 points out the differences in organization during 16 bit by 16 bit multiplies from that during dual 8 bit by 8 bit multiplies. Multiplier first input bus 201 supplies a first data input to multiplier 220 and multiplier second input bus 202 supplies a second data input. Multiplier first input bus 201 supplies 19 bit derived value circuit 350. Nineteen bit derived value circuit 350 forms a 19 bit quantity from the 16 bit input. Nineteen bit derived value circuit 350 includes a control input indicating whether multiplier 220 executes a single 16 bit by 16 bit multiplication or dual 8 bit by 8 bit multiplication. Booth quad re-coder 351 receives the 19 bit value from 19 bit derived value circuit 350 and forms control signals for six partial product generators 353, 354, 356, 363, 364 and 366 (PPG5–PPG0). Booth quad re-coder 351 thus controls the core of multiplier 220 according to the first input or inputs on multiplier first input bus 201 for generating the desired product or products.

FIGS. 13 and 14 schematically illustrate the operation of 19 bit derived value circuit 350 and Booth quad re-coder 351. For all modes of operation, the 16 most significant bits of multiplier first input bus 201 are ignored by multiplier 220. FIG. 13 illustrates the 19 bit derived value for 16 bit by 16 bit multiplications. The 16 bits of the first input are left shifted by one place and sign extended by two places. In the unsigned mode, the sign is "0". Thus bits 18–17 of the 19 bit

derived value are the sign, bits 16–1 correspond to the 16 bit input, and bit 0 is always "0". The resulting 19 bits are grouped into six overlapping four-bit units to form the Booth quads. Bits 3–0 form the first Booth quad controlling partial product generator PPG0 353, bits 6–3 control partial product generator PPG1 354, bits 9–6 control partial product generator PPG2 356, bits 12–9 control partial product generator PPG3 363, bits 15–12 control partial product generator PPG4 364, and bits 18–15 control partial product generator PPG5 366. FIG. 14 illustrates the 19 bit derived value for dual 8 bit by 8 bit multiplications. The two inputs are pulled apart. The first input is left shifted by one place, the second input is left shifted by two places. Bits 0 and 9 of the 19 bit derived value are set to "0", bit 18 to the sign. The Booth quads are generated in the same manner as in 16 bit by 16 bit multiplication. Note that placing a "0" in bit 9 of the derived value makes the first three Booth quads independent of the second 8 bit input and the last three Booth quads independent of the first 8 bit input. This enables separation of the two products at the multiplier output.

The core of multiplier 220 includes: six partial product generators 353, 354, 356, 363, 364 and 366, which are designated PPG0 to PPG5, respectively; five adders 355, 365, 357, 267 and 368, designated adders A, B, C, D and E; and an output multiplexer 369. Partial product generators 353, 354, 356, 363, 364 and 366 are identical. Each partial product generator 353, 354, 356, 363, 364 and 366 forms a partial product based upon a corresponding Booth quad. These partial products are added to form the final product by adders 355, 365, 357, 367 and 368.

The operation of partial product generator 353,354, 356, 363, 364 and 366 is detailed in Tables 9 and 10. Partial product generators 353, 354, 356, 363, 364 and 366 multiply the input data derived from multiplier second input bus 202 by integer amounts ranging from −4 to +4. The multiply amounts for the partial product generators are based upon the value of the corresponding Booth quad. This relationship is shown in Table 9 below.

TABLE 9

Quad	Multiply Amount
0000	0
0001	1
0010	1
0011	2
0100	2
0101	3
0110	3
0111	4
1000	−4
1001	−3
1010	−3
1011	−2
1100	−2
1101	−1
1110	−1
1111	−0

Table 10 lists the action taken by the partial product generator based upon the desired multiply amount.

TABLE 10

Multiply Amount	Partial Product Generator Action
±0	select all zeroes
±1	pass input straight through

TABLE 10-continued

Multiply Amount	Partial Product Generator Action
±2	shift left one place
±3	select output of 3x generator
±4	shift left two places

In most cases, the partial product is easily derived. An all "0" output is selected for a multiply amount of 0. A multiply amount of 1 results in passing the input unchanged. Multiply amounts of 2 and 4 are done simply by shifting. A dedicated piece of hardware generates the multiple of 3. This hardware essentially forms the addition of the input value and the input left shifted one place.

Each partial product generator 353, 354, 356, 363, 364 and 366 receives an input value based upon the data received on multiply second input bus 202. The data on multiply second input bus 202 is 16 bits wide. Each partial product generator 353, 354, 356, 363, 364 and 366 needs to be 18 bits to hold the 16 bit number shifted two places left, as in the multiply by 4 case. The output of each partial product generator 353, 354, 356, 363, 364 and 366 is shifted three places left from that of the preceding partial product generator 353, 354, 356, 363, 364 and 366. Thus each partial product generator output is weighted by 8 from its predecessor. This is shown in FIG. 12, where bits 2–0 of each partial product generator 353, 354, 356, 363, 364 and 366 is handled separately. Note that adders A, B, C, D and E are always one bit wider than their input data to hold any overflow.

The adders 355, 357, 365, 367 and 368 used in the preferred embodiment employ redundant-sign-digit notation. In the redundant-sign-digit notation, a magnitude bit and a sign bit represents each bit of the number. This known format is useful in the speeding the addition operation in a manner not important to this invention. However this invention is independent of the adder type used, so for simplicity this will not be further discussed. During multiply operations data from the 16 least significant bits on multiply second input bus 202 is fed into each of the six partial product generator 353, 354, 356, 363, 364 and 366, and multiplied by the amount determined by the corresponding Booth quad.

Second input multiplexer 352 determines the data supplied to the six partial produce generators 353, 354, 356, 363, 364 and 366. This data comes from the 16 least significant bits on multiply second input bus 202. The data supplied to partial products generators 353, 354, 356, 363, 364 and 366 differ depending upon whether multiplier 220 executes a single 16 bit by 16 bit multiplication or dual 8 bit by 8 bit multiplication. FIG. 15 illustrates the second input data supplied to the six partial produce generators 353, 354, 356, 363, 364 and 366 during a 16 bit by 16 bit multiply. FIG. 15*a* illustrates the case of unsigned multiplication. The 16 bit input is zero extended to 18 bits. FIG. 15*b* illustrates the case of signed multiplication. The data is sign extended to 18 bits by duplicating the sign bit (bit 15). During 16 bit by 16 bit multiplication and of the six partial produce generators 353, 354, 356, 363, 364 and 366 receives the same second input.

The six partial produce generators 353, 354, 356, 363, 364 and 366 do not receive the same second input during dual 8 bit by 8 bit multiplication. Partial product generators 353, 345 and 356 receive one input and partial product generators 363, 364 and 366 receive another. This enables separation of

the two inputs when operating in multiple multiply mode. Note that in the multiple multiply mode there is no overlap of second input data supplied to the first three partial product generators **353**, **345** and **356** and the second three partial product generators **363**, **364** and **366**. FIG. **16** illustrates the second input data supplied to the six partial produce generators **353**, **354**, **356**, **363**, **364** and **366** during a dual 8 bit by 8 bit multiply. FIG. **16***a* illustrates the second input data supplied to partial product generators **353**, **354** and **356** for an unsigned input. FIG. **16***a* illustrates the input zero extended to 18 bits. FIG. **16***b* illustrates the second input data supplied to partial product generators **353**, **354** and **356** for a signed input, which is sign extended to 18 bits. FIG. **16***c* illustrates the second input data supplied to partial product generators **363**, **364** and **366** for an unsigned input. FIG. **16***c* illustrates the input at bits **15–8** with the other places of the 18 bits set to "0". FIG. **16***d* illustrates the second input data supplied to partial product generators **363**, **364** and **366** for a signed input. The 7 bit magnitude is at bits **14–8**, bits **17–15** hold the sign and bits **7–0** are set to "0".

Note that it would be possible to have added the partial products of partial product generators **353**, **354**, **356**, **363**, **364** and **366** in series. The present embodiment illustrated in FIG. **12** has two advantages over such a series of additions. This embodiment offers significant speed advantages by performing additions in parallel. This embodiment also lends itself well to performing dual 8 bit by 8 bit multiplies. These can be very useful in speeding data manipulation and data transfers where an 8 bit by 8 bit product provides the data resolution needed.

A further multiplexer switches between the results of a 16 bit by 16 bit multiply and dual 8 bit by 8 bit multiplies. Output multiplexer **369** is controlled by a signal indicating whether multiplier **220** executes a single 16 bit by 16 bit multiplication or dual 8 bit by 8 bit multiplication. FIG. **17** shows the derivation of each bit of the resultant. FIG. **17***a* illustrates the derivation of each bit for a 16 bit by 16 bit multiply. Bits **31–9** of the resultant come from bits **22–0** of adder E **368**, respectively. Bits **8–6** come from bits **2–0** of adder C **357**, respectively. Bits **5–3** come from bits **2–0** of adder A **355**, respectively. Bits **2–0** come from bits **2–0** of partial product generator **353**. FIG. **17***b* illustrates the derivation of each bit for the case of dual 8 bit by 8 bit multiplication. Bits **31–16** of the resultant in this case come from bits **15–0** of adder D **367**, respectively. Bits **15–6** of the resultant come from bits **9–0** of adder C **357** respectively. As in the case illustrated in FIG. **17***a*, bits **5–3** come from bits **2–0** of adder A **355** and bits **2–0** come from bits **2–0** of partial product generator **353**.

It should be noted that in the actual implementation of output multiplexer **369** requires duplicated data paths to handle both the magnitude and sign required by the redundant-sign-digit notation. This duplication has not been shown or described in detail. The redundant-sign-digit notation is not required to practice this invention, and those skilled in the art would easily realize how to construct output multiplexer **369** to achieve the desired result in redundant-sign-digit notation. Note also when using the redundant-sign-digit notation, the resultant generally needs to be converted into standard binary notation before use by other parts of data unit **110**. This conversion is known in the art and will not be further described.

It can be seen from the above description that with the addition of a small amount of logic the same basic hardware can perform 16 bit by 16 multiplication and dual 8 bit by 8 bit multiplications. The additional hardware consists of multiplexers at the two inputs to the multiplier core, a modification to the Booth re-coder logic and a multiplexer at the output of the multiplier. This additional hardware permits much greater data through put when using dual 8 bit by 8 bit multiplication.

Adder **226** has three inputs. A first input is set to all zeros. A second input receives the 16 most significant bits (bits **31–16**) of the left shifted resultant of multiplier **220**. A carry-in input receives the output of bit **15** of this left shifter resultant of multiplier **220**. Multiplexer Rmux **221** selects either the entire 32 bit resultant of multiplier **220** as shifted by product left shifter **224** to supply to multiply destination bus **203** via multiplexer Bmux **227** or the sum from adder **226** forms the 16 most significant bits and the 16 most significant bits of multiplier first input bus **201** forms the 16 least significant bits. As noted above, in the preferred embodiment the state of the "R" bit (bit **6**) of data register **D0** controls this selection at multiplexer Rmux **221**. If this "R" bit is "0", then multiplexer Rmux **221** selects the shifted 32 bit resultant. If this "R" bit is "1", then multiplexer Rmux **221** selects the 16 rounded bits and the 16 most significant bits of multiplier first input bus **201**. Note that it is equally feasible to control multiplexer Rmux **221** via an instruction word bit.

Adder **226** enables a multiply and round function on a 32 bit data word including a pair of packed 16 bit half words. Suppose that a first of the data registers **200** stores a pair of packed half words (a :: b), a second data register stores a first half word coefficient (X :: c1) and a third data register stores a second half word coefficient (X :: c2), where X may be any data. The desired resultant is a pair of packed half words (a*c2 :: b*c1) with a*c2 and b*c1 each being the rounded most significant bits of the product. The desired resultant may be formed in two instructions using adder **226** to perform the rounding. The first instruction is:

$$\begin{array}{lcl} mdst & = & msrc1 \ * \ msrc2 \\ (b*c1 :: a) & = & (a :: b) \ * \ (X :: c1) \end{array}$$

As previously described multiplier first input bus **201** supplies its 16 least significant bits, corresponding to b, to the first input of multiplier **220**. At the same time multiply second input bus **202** supplies its 16 least significant bits, corresponding to c1, to the second input of multiplier **220**. This 16 by 16 bit multiply produces a 32 bit product. The 16 most significant bits of the 32 bit resultant form one input to adder **226** with "0" supplied to the other input of adder **226**. If bit **15** of the 32 bit resultant is "1", then the 16 most significant bits of the resultant is incremented, otherwise these 16 most significant bits are unchanged. Thus the 16 most significant bits of the multiply operation are rounded in adder **226**. Note that one input to multiplexer Rmux **221** includes the 16 bit resultant from adder **226** as the 16 most significant bits and the 16 most significant bits from multiplier first input bus **201**, which is the value a, as the least significant bits. Also note that the 16 most significant bits on multiplier second input bus **202** are discarded, therefore their initial state is unimportant. Multiplexer Rmux selects the combined output from adder **226** and multiplier first input bus **201** for storage in a destination register in data registers **200**.

The packed word multiply/round operation continues with another multiply instruction. The resultant (b*c1 :: a) of the first multiply instruction is recalled via multiply first input bus **201**. This is shown below:

$$mdst = msrc1 * msrc2$$
$$(a*c2 :: b*c1) = (b*c1 :: a) * (X :: c2)$$

The multiply occurs between the 16 least significant bits on the multiplier first input bus **201**, the value a, and the 16 least significant bits on the multiplier second input bus **202**, the value c2. The 16 most significant bits of the resultant are rounded using adder **226**. These bits become the 16 most significant bits of one input to multiplexer Rmux **221**. The 16 most significant bits on multiplier first input bus **201**, the value b*c1, becomes the 16 least significant bits of the input to multiplexer Rmux **221**. The 16 most significant bits on the multiplier second input bus **202** are discarded. Multiplexer Rmux **221** then selects the desired resultant (a*c2 :: b*c1) for storage in data registers **200** via multiplexer Bmux **227** and multiplier destination bus **203**. Note that this process could also be performed on data scaled via product left shifter **224**, with adder **226** always rounding the least significant bit retained. Also note that the factors c1 and c2 may be the same or different.

This packed word multiply/round operation is advantageous because the packed 16 bit numbers can reside in a single register. In addition fewer memory loads and stores are needed to transfer such packed data than if this operation was not supported. Also note that no additional processor cycles are required in handling this packed word multiply/rounding operation. The previous description of the packed word multiply/round operation partitioned multiplier first input bus **201** into two equal halves. This is not necessary to employ the advantages of this invention. As a further example, it is feasible to partition multiplier first input bus **201** into four 8 bit sections. In this further example multiplier **220** forms the product of the 8 least significant bits of multiplier first input bus **201** and the 8 least significant bits of multiplier second input bus **202**. After optional scaling in product left shifter **224** and rounding via adder **226**, the 8 most significant bits of the product form the most significant bits of one input to multiplexer Mmux **221**. In this further example, the least significant 24 bits of this second input to multiplexer Mmux **221** come from the most significant 24 bits on multiplier first input bus **201**. This further example permits four 8 bit multiplies on such a packed word in **4** passes through multiplier **220**, with all the intermediate results and the final result packed into one 32 bit data word. To further generalize, this invention partitions the original N bit data word into a first set of M bits and a second set of L bits. Following multiplication and rounding, a new data word is formed including the L most significant bits of the product and the first set of M bits from the first input. The data order in the resultant is preferably shifted or rotated in some way to permit repeated multiplications using the same technique. As in the further example described above, the number of bits M need not equal the number of bits L. In addition, the sum of M and L need not equal the original number of bits N.

In the preferred embodiment the round function selected by the "R" (bit **6**) of data register **D0** is implemented in a manner to increase its speed. Multiplier **220** employs a common hardware multiplier implementation that employs internally a redundant-sign-digit notation. In the redundant-sign-digit notation each bit of the number is represented by a magnitude bit and a sign bit. This known format is useful in the internal operation of multiplier **220** in a manner not important to this invention. Multiplier **220** converts the resultant from this redundant-sign-digit notation to standard binary notation before using the resultant. Conventional conversion operates by subtracting the negative signed magnitude bits from the positive signed magnitude bits. Such a subtraction ordinarily involves a delay due to borrow ripple from the least significant bit to the most significant bit. In the packed multiply/round operation the desired result is the 16 most significant bits and the rounding depends upon bit **15**, the next most significant bit. Though the results are the most significant bits, the borrow ripple from the least significant bit may affect the result. Conventionally the borrow ripple must propagate from the least significant bit to bit **15** before being available to make the rounding decision.

FIG. **18** illustrates in block diagram form hardware for speeding this rounding determination. In FIG. **18** the 32 bit multiply resultant from multiplier **220** is separated into a most significant 16 bits (bits **31–16**) coded in redundant-sign-digit form stored in register **370** and a least significant 16 bits (bits **15–0**) coded in redundant-sign-digit form stored in register **380**. In FIG. **18** product left shifter **224** is used for scaling as previously described. Product left shifter **224** left shifts both the magnitude bit and the sign bit for each bit of the of redundant-sign-digit form stored in registers **370** and **380** of multiplier **220** prior to forming the resultant. The shift amount comes from multiply shift multiplexer MSmux **225** as previously described above.

Conventionally such redundant-sign-digit notation is converted to standard binary notation by generating carry/borrow control signals. Carry path control signal generator **382** forms three carry path control signals, propagate, kill and generate, from the magnitude and sign bits of the corresponding desired resultant bit. These signals are easily derived according to Table 11.

TABLE 11

Magnitude	Sign	Indicates	Carry Path Control Signal
0	X	Zero (0)	Propagate (P)
1	0	Plus One (1)	Kill (K)
1	1	Minus One (T)	Generate (G)

Carry path control signal generator **382** supplies these carry path control signals to borrow ripple unit **386**. Borrow ripple unit **386** uses the bit wise carry path control signals to control borrow ripple during the subtraction of the negatively signed bits from the positively signed bits. Note from Table 11 that the three signals propagate, kill and generate are mutually exclusive. One and only one of these signals is active at any particular time. A propagate signal causes any borrow signal from the previous less significant bit to propagate unchanged to the next more significant bit. A kill signal absorbs any borrow signal from the prior bit and prevents propagation to the next bit. A generate signal produces a borrow signal to propagate to the next bit whatever the received borrow signal. Borrow ripple unit **386** propagates the borrow signal from the least significant bit to the most significant bit. As illustrated in FIG. **18**, bits **15–0** are converted in this manner. The only part of the result used is the data of bit **15** d[15] and the borrow output signal of bit **15** b_{out}[15].

The circuit illustrated in FIG. **18** employs a different technique to derive the 16 most significant bits. Note, that except for the rounding operation that depends upon bit **15**, only the 16 most significant bits are needed in the packed multiply/round operation. There are two possible resultants for bits **31–16** depending upon the rounding determination. The circuit of FIG. **18** computes both these possible resultants in parallel and the selects the appropriate resultant depending upon the data of bit **15** d[15] and the borrow

output signal of bit 15 b_{out}[15]. This substantially reduces the delay forming the rounded value. Note that using adder 226 to form the rounded value as illustrated in FIG. 5 introduces an additional carry ripple delay within adder 226 when forming the sum.

The circuit illustrated in FIG. 18 forms the minimum and maximum possible rounded results simultaneously. If R is the simple conversion of the 16 most significant bits, then the rounded final result may be R−1, R or R+1. These are selected based upon the data of bit 15 d[15] and the borrow output signal of bit 15 b_{out}[15] according to Table 12.

TABLE 12

d [15]	b_{out} [15]	Final Result	
0	0	R	Neither Increment nor decrement
0	1	R−1	Decrement only
1	0	R+1	Increment only
1	1	R	Both increment and decrement

The circuit of FIG. 18 computes the value R−1 for the 16 most significant bits employing carry path control signal generator 372 and borrow ripple unit 376. Carry path control signal generator 372 is the same as carry path control signal generator 382 and operates according to Table 11. Borrow ripple unit 376 is the same as borrow ripple unit 386. Borrow ripple unit 376 computes the value R−1 because the borrow-in input is always supplied with a borrow value of "1", thus always performing a decrement of the simple conversion value R.

The circuit of FIG. 18 forms the value R+1 by adding 2 to the value of R−1. Note that a binary number may be incremented by 1 by toggling all the bits up to and including the right most "0" bit in the original binary number. The circuit of FIG. 18 employs this technique to determine bits 31–17. This addition takes place in two stages in a manner not requiring a carry borrow for the entire 16 bits. In the first stage, mask ripple unit 374 generates a mask from the carry path control signals. An intermediate mask is formed with a "1" in any bit position in which the converted result is known to be "0" or known to differ from the result of the prior bit. Mask ripple unit 374 sets other bit positions to "0". The manner of forming this intermediate mask is shown in Table 13.

TABLE 13

Bit [n]		Bit [n−1]		Final Result of Bit [n]	Intermediate Mask Value
T	(G)	T	(G)	0	1
0	(P)	T	(G)	1	0
1	(K)	T	(G)	0	1
T	(G)	0	(P)	Different from Bit [n−1]	1
0	(P)	0	(P)	Same as Bit [n−1]	0
1	(K)	0	(P)	Different from Bit [n−1]	1
T	(G)	1	(K)	1	0
0	(P)	1	(K)	0	1
1	(K)	1	(K)	1	0

Review of the results of Table 13 reveal that this operation can be performed by the function P[n] XNOR K[n−1]. Thus a simple circuit generates the intermediate mask for each bit. Mask ripple unit 374 ripples through the intermediate mask until reaching the right most "0". Those bits including the right most "0" bit are set to "1", and all more significant bits are set to "0". This toggle mask and the R−1 result from borrow ripple unit 376 are supplied to exclusive OR unit 378. Exclusive OR unit 378 toggles those bits from borrow ripple unit 376 corresponding to the mask generated by mask ripple unit 374.

Multiplexer 390 assembles the rounded resultant. This operation takes place as shown in Tables 14 and 15. Table 14 shows the derivation of bit 16, the least significant rounded bit of the desired resultant, depending upon the data of bit d[15] and the borrow output signal of bit 15 b_{out}[15]. These results from the 16 least significant bits of the output of multiplier 220 are available from borrow ripple unit 386.

TABLE 14

d [15]	b_{out} [15]	Final Result for Bit [16]
0	0	~R−1 [16]
0	1	R−1 [16]
1	0	R−1 [16]
1	1	~R−1 [16]

The data of bit 15 d[15], the borrow output signal of bit 15 b_{out}[15] and the final result of bit 16 determine bits 31–17 according to Table 15.

TABLE 15

d [15]	b_{out} [15]	Final Result of Bit [16]	Final Result Bits 31–17
0	0	0	R+1 [31–17]
0	0	1	R−1 [31–17]
0	1	X	R−1 [31–17]
1	0	X	R+1 [31–17]
1	1	0	R+1 [31–17]
1	1	1	R−1 [31–17]

Thus multiplexer 390 forms the desired rounded resultant, which is the same as formed by adder 226. The manner of generation of the rounded resultant substantially eliminates the carry ripple delay associated with adder 226. Note that FIG. 5 contemplates circuits similar to carry path control signal generators 372 and 382 and borrow ripple units 376 and 386 to generate the output of multiplier 220 in normal coded form. Thus the circuit illustrated in FIG. 18 substitutes the delay of exclusive OR unit 378 and multiplexer 390 for the carry ripple delay of adder 226. The delay of exclusive OR unit 378 and multiplexer 390 is expected to be considerably less than the delay of adder 226. This is in a critical path, because the rounding performed by adder 226 follows the operation of multiplier 220. Thus this reduction in delay enables speeding up of the entire execute pipeline stage. This in turn enhances the rate of operation of multiprocessor integrated circuit 100.

Note that the circuit illustrated in FIG. 18 is employed as described above only if the "R" bit of data register 200 D0 selects the packed word multiply/rounding operation. In the event that the "R" bit of data register 200 D0 is "0", the packed word multiply/round operation is not enabled. In this event borrow ripple units 376 and 386 may be connected conventionally, with the signal b_{out}[15] from borrow ripple unit 386 coupled to the borrow input b_{in} of borrow ripple unit 376. Borrow ripple units 376 and 386 thus produce the shifted 32 bit resultant of multiplier 220 for selection by multiplexer Rmux 221.

Arithmetic logic unit 230 performs arithmetic and logic operations within data unit 110. Arithmetic logic unit 230 advantageously includes three input ports for performing three input arithmetic and logic operations. Numerous buses and auxiliary hardware supply the three inputs.

Input A bus 241 supplies data to an A-port of arithmetic logic unit 230. Multiplexer Amux 232 supplies data to input A bus 241 from either multiplier second input bus 202 or

arithmetic logic unit first input bus **205** depending on the instruction. Data on multiplier second input bus **202** may be from a specified one of data registers **200** or from an immediate field of the instruction via multiplexer Imux **222** and buffer **223**. Data on arithmetic logic unit first input bus **205** may be from a specified one of data registers **200** or from global port source data bus Gsrc bus **105** via buffer **106**. Thus the data supplied to the A-port of arithmetic logic unit **230** may be from one of the data registers **200**, from an immediate field of the instruction word or a long distance source from another register of digital image/graphics processor **71** via global source data bus Gsrc **105** and buffer **106**.

Input B bus **242** supplies data to the B-port of arithmetic logic unit **230**. Barrel rotator **235** supplies data to input B bus **242**. Thus barrel rotator **235** controls the input to the B-port of arithmetic logic unit **230**. Barrel rotator **235** receives data from arithmetic logic unit second input bus **206**. Arithmetic logic unit second input bus **206** supplies data from a specified one of data registers **200**, data from global port source data bus Gsrc bus **105** via buffer **104** or a special data word from buffer **236**. Buffer **236** supplies a 32 bit data constant of "00000000000000000000000000000001" (also called Hex "1") to arithmetic logic unit second input bus **206** if enabled. Note hereinafter data or addresses preceded by "Hex" are expressed in hexadecimal. Data from global port source data bus Gsrc **105** may be supplied to barrel rotator **235** as a long distance source as previously described. When buffer **236** is enabled, barrel rotator **235** enables generation on input B bus **242** of any constant of the form 2^N, where N is the barrel rotate amount. Constants of this form are useful in operations to control only a single bit of a 32 bit data word. The data supplied to arithmetic logic unit second input bus **206** and barrel rotator **235** depends upon the instruction.

Barrel rotator **235** is a 32 bit rotator that may rotate its received data from 0 to 31 positions. It is a left rotator, however, a right rotate of n bits may be obtained by left rotating 32–n bits. A five bit input from rotate bus **244** controls the amount of rotation provided by barrel rotator **235**. Note that the rotation is circular and no bits are lost. Bits rotated out the left of barrel rotator **235** wrap back into the right. Multiplexer Smux **231** supplies rotate bus **244**. Multiplexer Smux **231** has several inputs. These inputs include: the five least significant bits of multiplier first input bus **201**; the five least significant bits of multiplier second input bus **202**; five bits from the "DBR" field of data register **D0**; and a five bit zero constant "00000". Note that because multiplier second input bus **202** may receive immediate data via multiplexer Imux **222** and buffer **223**, the instruction word can supply an immediate rotate amount to barrel rotator **235**. Multiplexer Smux **231** selects one of these inputs to determine the amount of rotation in barrel rotator **235** depending on the instruction. Each of these rotate quantities is five bits and thus can set a left rotate in the range from 0 to 31 bits.

Barrel rotator **235** also supplies data to multiplexer Bmux **227**. This permits the rotated data from barrel rotator **235** to be stored in one of the data registers **200** via multiplier destination bus **203** in parallel with an operation of arithmetic logic unit **230**. Barrel rotator **235** shares multiplier destination bus **203** with multiplexer Rmux **221** via multiplexer Bmux **227**. Thus the rotated data cannot be saved if a multiply operation takes place. In the preferred embodiment this write back method is particularly supported by extended arithmetic logic unit operations, and can be disabled by specifying the same register destination for barrel rotator **235** result as for arithmetic logic unit **230** result. In this case only the result of arithmetic logic unit **230** appearing on arithmetic logic unit destination bus **204** is saved.

Although the above description refers to barrel rotator **235**, those skilled in the art would realize that substantial utility can be achieved using a shifter which does not wrap around data. Particularly for shift and mask operations where not all of the bits to the B-port of arithmetic logic unit **230** are used, a shifter controlled by rotate bus **244** provides the needed functionality. In this event an additional bit, such as the most significant bit on the rotate bus **244**, preferably indicates whether to form a right shift or a left shift. Five bits on rotate bus **244** are still required to designate the magnitude of the shift. Therefore it should be understood in the description below that a shifter may be substituted for barrel rotator **235** in many instances.

Input C bus **243** supplies data to the C-port of arithmetic logic unit **230**. Multiplexer Cmux **233** supplies data to input C bus **243**. Multiplexer Cmux **233** receives data from four sources. These are LMO/RMO/LMBC/RMBC circuit **237**, expand circuit **238**, multiplier second input bus **202** and mask generator **239**.

LMO/RMO/LMBC/RMBC circuit **237** is a dedicated hardware circuit that determines either the left most "1", the right most "1", the left most bit change or the right most bit change of the data on arithmetic logic unit second input bus **206** depending on the instruction or the "FMOD" field of data register **D0**. LMO/RMO/LMBC/RMBC circuit **237** supplies to multiplexer Cmux **233** a 32 bit number having a value corresponding to the detected quantity. The left most bit change is defined as the position of the left most bit that is different from the sign bit **32**. The right most bit change is defined as the position of the right most bit that is different from bit **0**. The resultant is a binary number corresponding to the detected bit position as listed below in Table 16. The values are effectively the big endian bit number of the detected bit position, where the result is 31-(bit position).

TABLE 16

bit position	result
0	31
1	30
2	29
3	28
4	27
5	26
6	25
7	24
8	23
9	22
10	21
11	20
12	19
13	18
14	17
15	16
16	15
17	14
18	13
19	12
20	11
21	10
22	9
23	8
24	7
25	6
26	5
27	4
28	3
29	2

TABLE 16-continued

bit position	result
30	1
31	0

This determination is useful for normalization and for image compression to find a left most or right most "1" or changed bit as an edge of an image. The LMO/RMO/LMBC/RMBC circuit **237** is a potential speed path, therefore the source coupled to arithmetic logic unit second input bus **206** is preferably limited to one of the data registers **200**. For the left most "1" and the right most "1" operations, the "V" bit indicating overflow of status register **210** is set to "1" if there were no "1's" in the source, and "0" if there were. For the left most bit change and the right most bit change operations, the "V" bit is set to "1" if all bits in the source were equal, and "0" if a change was detected. If the "V" bit is set to "1" by any of these operations, the LMO/RMO/LMBC/RMBC result is effectively 32. Further details regarding the operation of status register **210** appear above.

Expand circuit **238** receives inputs from multiple flags register **211** and status register **210**. Based upon the "Msize" field of status register **210** described above, expand circuit **238** duplicates some of the least significant bits stored in multiple flags register **211** to fill 32 bits. Expand circuit **238** may expand the least significant bit 32 times, expand the two least significant bits 16 times or expand the four least significant bits 8 times. The "Asize" field of status register **210** controls processes in which the 32 bit arithmetic logic unit **230** is split into independent sections for independent data operations. This is useful for operation on pixels sizes less than the 32 bit width of arithmetic logic unit **230**. This process, as well as examples of its use, will be further described below.

Mask generator **239** generates 32 bit masks that may be supplied to the input C bus **243** via multiplexer Cmux **233**. The mask generated depends on a 5 bit input from multiplexer Mmux **234**. Multiplexer Mmux **234** selects either the 5 least significant bits of multiplier second input bus **202**, or the "DBR" field from data register **D0**. In the preferred embodiment, an input of value N causes mask generator **239** to generate a mask generated that has N "1's" in the least significant bits, and 32–N "0's" in the most significant bits. This forms an output having N right justified "1's". This is only one of four possible methods of operation of mask generator **239**. In a second embodiment, mask generator **239** generates the mask having N right justified "0's", that is N "0's" in the least significant bits and N–32 "1's" in the most significant bits. It is equally feasible for mask generator **239** to generate the mask having N left justified "1's" or N left justified "0's". Table 17 illustrates the operation of mask generator **239** in accordance with the preferred embodiment when multiple arithmetic is not selected.

TABLE 17

Mask Generator Input	Mask - Nonmultiple Operation							
00000	0000	0000	0000	0000	0000	0000	0000	0000
00001	0000	0000	0000	0000	0000	0000	0000	0001
00010	0000	0000	0000	0000	0000	0000	0000	0011
00011	0000	0000	0000	0000	0000	0000	0000	0111

TABLE 17-continued

Mask Generator Input	Mask - Nonmultiple Operation							
00100	0000	0000	0000	0000	0000	0000	0000	1111
00101	0000	0000	0000	0000	0000	0000	0001	1111
00110	0000	0000	0000	0000	0000	0000	0011	1111
00111	0000	0000	0000	0000	0000	0000	0111	1111
01000	0000	0000	0000	0000	0000	0000	1111	1111
01001	0000	0000	0000	0000	0000	0001	1111	1111
01010	0000	0000	0000	0000	0000	0011	1111	1111
01011	0000	0000	0000	0000	0000	0111	1111	1111
01100	0000	0000	0000	0000	0000	1111	1111	1111
01101	0000	0000	0000	0000	0001	1111	1111	1111
01110	0000	0000	0000	0000	0011	1111	1111	1111
01111	0000	0000	0000	0000	0111	1111	1111	1111
10000	0000	0000	0000	0000	1111	1111	1111	1111
10001	0000	0000	0000	0001	1111	1111	1111	1111
10010	0000	0000	0000	0011	1111	1111	1111	1111
10011	0000	0000	0000	0111	1111	1111	1111	1111
10100	0000	0000	0000	1111	1111	1111	1111	1111
10101	0000	0000	0001	1111	1111	1111	1111	1111
10110	0000	0000	0011	1111	1111	1111	1111	1111
10111	0000	0000	0111	1111	1111	1111	1111	1111
11000	0000	0000	1111	1111	1111	1111	1111	1111
11001	0000	0001	1111	1111	1111	1111	1111	1111
11010	0000	0011	1111	1111	1111	1111	1111	1111
11011	0000	0111	1111	1111	1111	1111	1111	1111
11100	0000	1111	1111	1111	1111	1111	1111	1111
11101	0001	1111	1111	1111	1111	1111	1111	1111
11110	0011	1111	1111	1111	1111	1111	1111	1111
11111	0111	1111	1111	1111	1111	1111	1111	1111

A value N of "0" thus generates 32 "0's". In some situations however it is preferable that a value of "0" generates 32 "1's". This function is selected by the "%!" modification specified in the "FMOD" field of status register **210** or in bits **52**, **54**, **56** and **58** of the instruction when executing an extended arithmetic logic unit operation. This function can be implemented by changing the mask generated by mask generator **239** or by modifying the function of arithmetic logic unit **230** so that mask of all "0's" supplied to the C-port operates as if all "1's" were supplied. Note that similar modifications of the other feasible mask functions are possible. Thus the "%!" modification can change a mask generator **239** which generates a mask having N right justified "0's" to all "0's" for N=0. Similarly, the "%1" modification can change a mask generator **239** which generates N left justified "1's" to all "1's" for N=0, or change a mask generator **239** which generates N left justified "0's" to all "0's" for N=0.

Selection of multiple arithmetic modifies the operation of mask generator **239**. When the "Asize" field of status register is "110", this selects a data size of 32 bits and the operation of mask generator **239** is unchanged from that shown in Table 17. When the "Asize" field of status register is "101", this selects a data size of 16 bits and mask generator **239** forms two independent 16 bit masks. This is shown in Table 18. Note that in this case the most significant bit of the input to mask generator **239** is ignored. Table 18 shows this bit as a don't care "X".

TABLE 18

Mask Generator Input	Mask - Half Word Operation							
X0000	0000	0000	0000	0000	0000	0000	0000	0000
X0001	0000	0000	0000	0001	0000	0000	0000	0001

TABLE 18-continued

Mask Generator Input	Mask - Half Word Operation							
X 0 0 1 0	0000	0000	0000	0011	0000	0000	0000	0011
X 0 0 1 1	0000	0000	0000	0111	0000	0000	0000	0111
X 0 1 0 0	0000	0000	0000	1111	0000	0000	0000	1111
X 0 1 0 1	0000	0000	0001	1111	0000	0000	0001	1111
X 0 1 1 0	0000	0000	0011	1111	0000	0000	0011	1111
X 0 1 1 1	0000	0000	0111	1111	0000	0000	0111	1111
X 1 0 0 0	0000	0000	1111	1111	0000	0000	1111	1111
X 1 0 0 1	0000	0001	1111	1111	0000	0001	1111	1111
x 1 0 1 0	0000	0011	1111	1111	0000	0011	1111	1111
x 1 0 1 1	0000	0111	1111	1111	0000	0111	1111	1111
X 1 1 0 0	0000	1111	1111	1111	0000	1111	1111	1111
X 1 1 0 1	0001	1111	1111	1111	0001	1111	1111	1111
X 1 1 1 0	0011	1111	1111	1111	0011	1111	1111	1111
X 1 1 1 1	0111	1111	1111	1111	0111	1111	1111	1111

The function of mask generator **239** is similarly modified for a selection of byte data via an "Asize" field of "100". Mask generator **239** forms four independent masks using only the three least significant bits of its input. This is shown in Table 19.

TABLE 19

Mask Generator Input	Mask - Byte Operation							
X X 0 0 0	0000	0000	0000	0000	0000	0000	0000	0000
X X 0 0 1	0000	0001	0000	0001	0000	0001	0000	0001
X X 0 1 0	0000	0011	0000	0011	0000	0011	0000	0011
X X 0 1 1	0000	0111	0000	0111	0000	0111	0000	0111
X X 1 0 0	0000	1111	0000	1111	0000	1111	0000	1111
X X 1 0 1	0001	1111	0001	1111	0001	1111	0001	1111
X X 1 1 0	0011	1111	0011	1111	0011	1111	0011	1111
X X 1 1 1	0111	1111	0111	1111	0111	1111	0111	1111

As noted above, it is feasible to support multiple operations of 8 sections of 4 bits each, 16 sections of 2 bits each and 32 single bit sections. Those skilled in the art would realize that these other data sizes require similar modification to the operation of mask generator **239** as shown above in Tables 17, 18, and 19.

Data unit **110** includes a three input arithmetic logic unit **230**. Arithmetic logic unit **230** includes three input busses: input A bus **241** supplies an input to an A-port; input B bus **242** supplies an input to a B-port; and input C bus **243** supplies an input to a C-port. Arithmetic logic unit **230** supplies a resultant to arithmetic logic unit destination bus **204**. This resultant may be stored in one of the data registers of data registers **200**. Alternatively the resultant may be stored in another register within digital image/graphics processor **71** via buffer **108** and global port destination data bus Gdst **107**. This function is called a long distance operation. The instruction specifies the destination of the resultant. Function signals supplied to arithmetic logic unit **230** from function signal generator **245** determine the particular three input function executed by arithmetic logic unit **230** for a particular cycle. Bit **0** carry-in generator **246** forms a carry-in signal supplied to bit **0**, the first bit of arithmetic logic unit **230**. As previously described, during multiple arithmetic operations bit **0** carry-in generator **246** supplies the carry-in signal to the least significant bit of each of the multiple sections.

FIG. **19** illustrates in block diagram form the construction of an exemplary bit circuit **400** of arithmetic logic unit **230**. Arithmetic logic unit **230** preferably operates on data words of 32 bits and thus consists of 32 bit circuits **400** in parallel. Each bit circuit **400** of arithmetic logic unit **230** receives: the corresponding bits of the three inputs A_i, B_i and C_i; a zero carry-in signal designated c_{in0} from the previous bit circuit **400**; a one carry-in signal designated c_{in1} from the previous bit circuit **400**; an arithmetic enable signal A_{en}; an inverse kill signal $\overline{K}_{i-1}$ from the previous bit circuit; a carry sense select signal for selection of carry-in signal c_{in0} or c_{in1}; and eight inverse function signals $\overline{F7}$–$\overline{F0}$. The carry-in signals c_{in0} and c_{in1} for the first bit (bit **0**) are identical and are generated by a special circuit that will be described below. Note that the input signals A_i, B_i and C_i are formed for each bit of arithmetic logic unit **230** and may differ. The arithmetic enable signal A_{en} and the inverted function signals $\overline{F7}$–$\overline{F0}$ are the same for all of the 32 bit circuits **400**. Each bit circuit **400** of arithmetic logic unit **230** generates: a corresponding one bit resultant S_i; an early zero signal Z_i; a zero carry-out signal designated c_{out0} that forms the zero carry-in signal c_{in0} for the next bit circuit; a one carry-out signal designated c_{out1} that forms the one carry-in signal c_{in1} for the next bit circuit; and an inverse kill signal $\overline{K}_i$ that forms the inverse kill signal $\overline{K}_{i-1}$ for the next bit circuit. A selected one of the zero carry-out signal c_{out0} or the one carry-out signal c_{out1} of the last bit in the 32 bit arithmetic logic unit **230** is stored in status register **210**, unless the "C" bit is protected from change for that instruction. In addition during multiple arithmetic the instruction may specify that carry-out signals from separate arithmetic logic unit sections be stored in multiple flags register **211**. In this event the selected zero carry-out signal c_{out0} or the one carry-out signal c_{out1} will be stored in multiple flags register **211**.

Bit circuit **400** includes resultant generator **401**, carry out logic **402** and Boolean function generator **403**. Boolean function generator **403** forms a Boolean combination of the respective bits inputs A_i, B_i and C_i according to the inverse function signals $\overline{F7}$–$\overline{F0}$. Boolean function generator produces a corresponding propagate signal P_i, a generate signal G_i and a kill signal K_i. Resultant logic **401** combines the propogate signal P_i with one of the carry-in signal c_{in0} or carry-in signal c_{in1} from a prior bit circuit **400** as selected by the carry sense select signal and forms the bit resultant S_i and an early zero signal Z_i. Carry out logic **402** receives the propagate signal P_i, the generate signal G_i, the kill signal K_i, the two carry-in signals c_{in0} and c_{in1} and an arithmetic enable signal A_{en}. Carry out logic **402** produces two carry-out signals c_{out0} and c_{out1} that are supplied to the next bit circuit **400**.

FIGS. **20** and **21** together illustrate an exemplary bit circuit **400** of arithmetic logic unit **230**. FIG. **20** illustrates the details of a resultant logic **401** and carry out logic **402** of each bit circuit **400** of arithmetic logic unit **230**. FIG. **21** illustrates the details of the corresponding Boolean function generator **403** of each bit circuit **400** of arithmetic logic unit **230**.

Each resultant logic **401** generates a corresponding resultant signal S_i and an early zero signal Z_i. Resultant logic **420** forms these signals from the two carry-in signals, an inverse propagate signal $\overline{P}_i$, an inverse kill signal $\overline{K}_{i-1}$ from the previous bit circuit and a carry sense select signal. The carry out logic **402** forms two carry-out signals and an inverse kill signal $\overline{K}_i$. These signals are formed from the two carry-in signals, an inverse propagate signal $\overline{P}_i$, an inverse generate signal $\overline{G}_i$ and a kill signal K_i for that bit circuit **400**. Each propagate signal indicates whether a "1" carry-in signal propagates through the bit circuit **400** to the next bit circuit **400** or is absorbed. The generate signal indicates whether the inputs to the bit circuit **400** generate a "1" carry-out signal

to the next bit circuit **400**. The kill signal indicates whether the input to the bit circuit **400** generate a "0" carry-out signal to the next bit circuit. Note that the propagate signal P_i, the generate signal G_i and the kill signal K_i are mutually exclusive. Only one of these signals is generated for each combination of inputs.

Each bit circuit **400** of arithmetic logic unit **230** employs a technique to reduce the carry ripple time through the 32 bits. Arithmetic logic unit **230** is divided into carry sections, preferably 4 sections of 8 bits each. The least significant bit circuit **400** of each such section has its zero carry-in signal c_{in0} hardwired to "0" and its one carry-in signal c_{in1} hardwired to "1". Each bit circuit **400** forms two resultants and two carry-out signals to the next bit circuit. Once the carry ripple through each section is complete, the actual carry output from the most significant bit of the previous carry section forms the carry sense select signal. This carry select signal permits selection of the actual resultant generated by the bits of a section via a multiplexer. The first carry section receives its carry select signal from bit **0** carry-in generator **246** described in detail below. This technique permits the carry ripple through the carry sections to take place simultaneously. This reduces the length of time required to generate the resultant at the cost of some additional hardware for the redundant carry lines and the carry sense selection.

Carry out logic **402** controls transformation of the carry-in signals into the carry-out signals. Carry out logic **402** includes identical circuit operating on the two carry-in signals c_{in0} and c_{in1}. The inverse propagate signal $\overline{P}_i$ and its inverse, the propagate signal P_i formed by invertor **412**, control pass gates **413** and **423**. If the propagate signal P_i is "1", then one carry-in line **410** is connected to one carry-out line **411** via pass gate **413** and zero carry-in line **420** is connected to zero carry-out line **421** via pass gate **423**. Thus the carry-in signal is propagated to the carry-out. If the propagate signal P_i is "0", then one carry-in line **410** is isolated from one carry-out line **411** and zero carry-in line **420** is isolated from carry-out line **421**. If the generate signal G_i is "1", that is if the inverse generate signal $\overline{G}_i$ is "0", then P-channel MOSFET (metal oxide semiconductor field effect transistor) **414** is turned on to couple the supply voltage to carry-out line **411** and P-channel MOSFET **424** is turned on to couple the supply voltage to carry-out line **421**. If the generate signal G_i is "0", that is if the inverse generate signal $\overline{G}_i$ is "1", then the P-channel MOSFETs **414** and **424** are cut off and do not affect the carry-out lines **411** and **421**. If the kill signal K_i is "1", then N-channel MOSFET **415** couples ground to carry-out line **411** and N-channel MOSFET **425** couples ground to carry-out line **421**. If the kill signal K_i is "0", then the N-channel MOSFETs **415** and **425** are cut off and do not affect the carry-out lines **411** and **421**. Invertor **422** generates the inverse kill signal $\overline{K}_i$ supplied to the next bit circuit.

Exclusive OR circuits **431** and **433** form the two resultants of resultant logic **401**. Exclusive OR circuits **431** and **433** each receive the propagate signal P_i from invertor **427** on an inverting input and the inverse propagate signal $\overline{P}_i$ from invertor **428** on a noninverting input. Exclusive OR circuit **431** receives the inverse zero carry-in signal $\overline{c}_{in0}$ from invertor **426** on a noninverting input and forms the resultant for the case of a "0" carry-in to the least significant bit of the current carry section. Likewise, exclusive OR circuit **433** receives the inverse one carry-in signal $\overline{c}_{in1}$ from invertor **416** on a noninverting input and forms the resultant for the case of a "1" carry-in to the least significant bit of the current carry section. Invertors **432** and **434** supply inputs to multiplexer **435**. Multiplexer **435** selects one of these signals based upon the carry sense select signal. This carry sense select signal corresponds to the actual carry-out signal from the most significant bit of the previous carry section. The inverted output of multiplexer **435** from invertor **436** is the desired bit resultant S_i.

Resultant logic **401** also forms an early zero signal Z_i for that bit circuit. This early zero signal Z_i gives an early indication that the resultant S_i of that bit circuit **400** is going to be "0". Exclusive OR circuit **437** receives the propagate signal P_i from invertor **427** on an inverting input and the inverse propagate signal $\overline{P}_i$ from invertor **428** on a noninverting input. Exclusive OR circuit **437** also receives the inverse kill signal $\overline{K}_{i-1}$ from the previous bit circuit **400** on a noninverting input. Exclusive OR circuit **437** forms early zero signal Z_i for the case in which the previous bit kill signal K_{i-1} generates a "0" carry-out signal and the propagate signal P_i is also "0". Note that if K_{i-1} is "0", then both the zero carry-out signal c_{out0} and the one carry-out signal c_{out1} are "0" whatever the state of the carry-in signals c_{in0} and c_{in1}. Note that this early zero signal Z_i is available before the carry can ripple through the carry section. This early zero signal Z_i may thus speed the determination of a zero output from arithmetic logic unit **230**.

Boolean function generator **403** of each bit circuit **400** of arithmetic logic unit **230** illustrated in FIG. **21** generates the propagate signal P_i, the generate signal G_i and the kill signal K_i for bit circuit **400**. Boolean function generator **403** consists of four levels. The first level includes pass gates **451**, **452**, **453**, **454**, **455**, **456**, **457** and **458**. Pass gates **451**, **453**, **455** and **457** are controlled in a first sense by input C_i and inverse input $\overline{C}_i$ from invertor **459**. Pass gates **452**, **454**, **456** and **458** are controlled in an opposite sense by input C_i and inverse input $\overline{C}_i$. Depending on the state of input C_i, either pass gates **451**, **453**, **455** and **457** are conductive or pass gates **452**, **454**, **456** and **458** are conductive. The second level includes pass gates **461**, **462**, **463** and **464**. Pass gates **461** and **463** are controlled in a first sense by input B_i and inverse input $\overline{B}_i$ from invertor **465**. Pass gates **462** and **464** are controlled in the opposite sense. Depending on the state of input B_i, either pass gates **461** and **463** are conductive or pass gates **462** and **464** are conductive. The third level includes pass gates **471**, **472** and **473**. Pass gates **471** is controlled in a first sense by input A_i and inverse input $\overline{A}_i$ from invertor **473**. Pass gates **472** and **473** are controlled in the opposite sense. Depending on the state of input A_i, either pass gates **471** is conductive or pass gates **472** and **473** are conductive. The first level includes invertors **441**, **442**, **443**, **444**, **445**, **446**, **447** and **448** that are coupled to corresponding inverted function signals $\overline{F7}$–$\overline{F0}$. invertors **441**, **442**, **443**, **444**, **445**, **446**, **447** and **448** provide input drive to Boolean function generator **403** and determine the logic function performed by arithmetic logic unit **230**.

Boolean function generator **403** forms the propagate signal P_i based upon the corresponding input signals A_i, B_i and C_i and the function selected by the state of the inverted function signals $\overline{F7}$–$\overline{F0}$. The propagate signal P_i at the input to invertor **476** is "1" if any path through pass gates **451**, **452**, **453**, **454**, **455**, **456**, **457**, **458**, **461**, **462**, **463**, **464**, **471** or **472** couples a "1" from one of the invertors **441**, **442**, **443**, **444**, **445**, **446**, **447** or **448**. In all other cases this propagate signal P_i is "0". Invertor **476** forms the inverse propagate signal $\overline{P}_i$, which is connected to resultant logic **401** illustrated in FIG. **20**.

Each pass gate **451**, **452**, **453**, **454**, **455**, **456**, **457**, **458**, **461**, **462**,**463**,**464**, **471**, **472** and **473** consists of an N-channel MOSFET and a P-channel MOSFET disposed in

parallel. The gate of the N-channel MOSFET receives a control signal. This field effect transistor is conductive if its gate input is above the switching threshold voltage. The gate of the P-channel MOSFET is driven by the inverse of the control signal via one of the invertors **459**, **465** or **474**. This field effect transistor is conductive if its gate input is below a switching threshold. Because the P-channel MOSFET operates in inverse to the operation of N-channel MOSFET, the corresponding invertor **459**, **467** or **474** assures that these two field effect transistors are either both conducting or both non-conducting. The parallel N-channel and P-channel field effect transistors insure conduction when desired whatever the polarity of the controlled input.

Tri-state AND circuit **480** forms the generate signal G_i and the kill signal K_i. The generate signal G_i, the kill signal K_i and the propagate signal P_i are mutually exclusive in the preferred embodiment. Therefore the propagate signal P_i controls the output of tri-state AND circuit **480**. If the propagate signal P_i is "1", then tri-state AND circuit **480** is disabled and both the generate signal G_i and the kill signal K_i are "0". Thus neither the generate signal G_i nor the kill signal K_i change the carry signal. Pass gate **473** couples the output from part of Boolean function generator **403** to one input of tri-state AND circuit **480**. The gate inputs of pass gate **473** are coupled to the first input bit A_i in the first sense. An N-channel MOSFET **475** conditionally couples this input of tri-state AND circuit **480** to ground. The inverse of the first input bit $\overline{A}_i$ supplies the gate input to N-channel MOSFET **475**. Pass gate **473** and N-channel MOSFET **475** are coupled in a wired OR relationship, however no OR operation takes place because their gate inputs cause them to be conductive alternately. N-channel MOSFET **475** serves to force a "0" input into tri-state AND circuit **480** when A_i="0". An arithmetic enable signal supplies the second input to tri-state AND circuit **480**.

The tri-state AND gate **480** operates as follows. If the propagate signal P_i is "1", then both P-channel MOSFET **481** and N-channel MOSFET **482** are conductive and pass gate **483** is non-conductive. This cuts off P-channel MOSFETs **414** and **424** and N-channel MOSFETs **415** and **425** so that none of these field effect transistor conducts. The output of tri-state AND circuit **480** thus is a high impedance state that does not change the signal on the carry-out lines **411** and **421**. If the propagate signal P_i is "0", then both P-channel MOSFET **481** and N-channel MOSFET **482** are non-conductive and pass gate **483** is conductive. The circuit then forms a logical AND of the two inputs. If either arithmetic enable or the signal at the junction of N-channel MOSFET **475** and pass gate **473** is "0" or both are "0", then at least one of P-channel MOSFET **484** or P-channel MOSFET **485** connects the supply voltage V+ (a logic "1") as the inverse generate signal $\overline{G}_i$ to the gates of P-channel MOSFETs **414** and **424** of carry out logic **402**. Thus P-channel MOSFETs **414** and **424** are non-conductive. At the same time pass gate **483** is conductive and supplies this "1" signal as kill signal K_i to the gates of N-channel MOSFETs **415** and **425** of carry out logic **402**. This actively pulls down the signal on zero carry-out line **421** forcing the zero carry-out signal c_{out0} to "0" and one carry-out line **411** forcing the one carry-out signal c_{out1} to "0". If both the inputs are "1", then the series combination of N-channel MOSFET **486** and N-channel MOSFET **487** supplies ground (a logic "0") to the gates of N-channel MOSFETs **415** and **425**. N-channel MOSFETs **415** and **425** of carry out logic **402** are cut off and non-conductive. At the same time pass gate **483** couples this "0" to the gates of P-channel MOSFETs **414** and **424**. Thus P-channel MOSFETs **414** and **424** of carry out logic **402** are conductive. This actively pulls up the signal on zero carry-out line **421** forcing the zero carry-out signal c_{out0} to "1" and one carry-out line **411** forcing the one carry-out signal c_{out1} to "1".

The bit circuit construction illustrated in FIG. **20** and **21** forms a propagate term, a generate term, a resultant term and two carry-out terms. Bit circuit **400** forms the propagate term P_i as follows:

$$P_i = F0\&(\sim A_i\&\sim B_i\&\sim C_i)|F1\&(A_i\&\sim B_i\&\sim C_i)|F2\&(\sim A_i\&B_i\&\sim C_i)|F3\&(A_i\&B_i\&\sim C_i)|F4\&(\sim A_i\&\sim B_i\&C_i)|F5\&(A_i\&\sim B_i\&C_i)|F6\&(\sim A_i\&B_i\&C_i)|F7\&(A_i\&B_i\&C_i)$$

Bit circuit **400** forms the generate term G_i as follows:

$$G_i = A_i\&[(F0\&\sim F1\&\sim B_i\&\sim C_i)|(F2\&\sim F3\&B_i\&\sim C_i)|(F4\&\sim F5\&\sim B_i\&C_i)|(F6\&\sim F7\&B_i\&C_i)]$$

Bit circuit **400** forms the kill term K_i as follows:

$$K_i = \sim G_i\&\sim P_i$$

Bit circuit **400** forms the resultant term S_i as follows:

$$S_i = P_i\hat{}(c_{in0}\&CSS|c_{in1}\&\sim CSS)$$

where: CSS is the carry sense select signal. Bit circuit **400** forms the two carry-out signals c_{out0} and c_{out1} as follows:

$$c_{out0} = (P_i\&c_{in0})|(G_i\&A_{en})|\sim(K_i\&A_{en})$$

$$c_{out1} = (P_i\&c_{in1})|(G_i\&A_{en})|\sim(K_i\&A_{en})$$

Note that for any particular bit i the propagate signal P_i, the generate signal G_i and the kill signal K_i are mutually exclusive. No two of these signals occurs simultaneously.

The construction of each bit circuit **400** enables arithmetic logic unit **230** to perform any one of 256 possible 3 input Boolean functions or any one of 256 possible 3 input mixed Boolean and arithmetic functions depending upon the inverted function signals $\overline{F7}$–$\overline{F0}$. The nine inputs including the arithmetic enable signal and the inverted function signals $\overline{F7}$–$\overline{F0}$ permit the selection of 512 functions. As will be further described below the data paths of data unit **110** enable advantageous use of three input arithmetic logic unit **230** to speed operations in many ways.

Table 20 lists the simple Boolean logic functions of bit circuit **400** in response to single function signals **F7–F0**. Since these are Boolean logic functions and the arithmetic enable signal is "0", both the generate and kill functions are disabled. Note that for Boolean extended arithmetic logic unit operations it is possible to specify the carry-in signals c_{in0} and c_{in1} from bit **0** carry-in generator **246** as previously described, thus permitting a carry ripple.

TABLE 20

8-bit ALU code field	Function Signal	Logical Operation
58	F7	A & B & C
57	F6	~A & B & C
56	F5	A & ~B & C
55	F4	~A & ~B & C
54	F3	A & B & ~C
53	F2	~A & B & ~C
52	F1	A & ~B & ~C
51	F0	~A & ~B & ~C

These functions can be confirmed by inspecting FIGS. **20** and **21**. For the example of F7="1" and **F6–F0** all equal to "0", invertors **441**, **442**, **443**, **444**, **446**, **447** and **448** each output a "0". Only invertor **445** produces a "1" output. The

propagate signal is "1" only if C_i="1" turning on pass gate **455**, B_i="1" turning on pass gate **463** and A_i="1" turning on pass gate **472**. All other combinations result in a propagate signal of "0". Since this is a logical operation, both the zero carry-in signal c_{in0} and the one carry-in signal c_{in1} are "0". Thus S_i="1" because both exclusive OR circuits **431** and **433** return the propagate signal. The other entries on Table 20 may be similarly confirmed.

A total of 256 Boolean logic functions of the three inputs A, B and C are enabled by proper selection of function signals **F7–F0**. Note that the state table of three inputs includes 8 places, thus there are 2^8=256 possible Boolean logic functions of three inputs. Two input functions are subset functions achieved by selection of function signals **F7–F0** in pairs. Suppose that a Boolean function of B and C, without relation to input A, is desired. Selection of **F7=F6**, **F5=F4**, **F3=F2** and **F1=F0** assures independence from input A. Note that the branches of Boolean function generator **403** connected to pass gates **471** and **472** are identically driven. This ensures that the result is the same whether A_i="1" or A_i="0". Such a selection still provides 4 controllable function pairs permitting specification of all 16 Boolean logic functions of inputs B and C. Note that the state table of two inputs includes four places, thus there are 2^4=16 possible Boolean logic functions of three inputs. Similarly, selection of **F7=F5**, **F6=F4**, **F3=F1** and **F2=F0** ensures independence from input B and provides 4 controllable function pairs for specification of 16 Boolean logic functions of inputs A and C. Selection of **F7=F3**, **F6=F2**, **F5=F1** and **F4=F0** permits selection via 4 controllable function pairs of 16 Boolean logic functions of inputs A and B independent of input C.

The instruction word determines the function performed by arithmetic logic unit **230** and whether this operation is arithmetic or Boolean logic. As noted in Table 20, the instruction word includes a field coded with the function signals for Boolean logic operations. This field, the "8 bit arithmetic logic unit" field (bits **58–51**) of the instruction word, is directly coded with the function signals when the instruction specifies a Boolean logic operation for arithmetic logic unit **230**.

The "8 bit arithmetic logic unit" field is differently coded when the instruction specifies arithmetic operations. Study of the feasible arithmetic functions indicates that a subset of these arithmetic functions specify the most often used operations. If the set of function signals **F7–F0** is expressed as a two place hexadecimal number, then these most often used functions are usually formed with only the digits a, **9**, **6** and **5**. In these sets of function signals **F7=~F6**, **F5=~F4**, **F3=~F2** and **F1=~F0**. Bits **57**, **55**, **53** and **51** specify fifteen operations, with an "8 bit arithmetic logic unit" field of all zeros reserved for the special case of non-arithmetic logic unit operations. Non-arithmetic logic unit operations will be described below. When executing an arithmetic operation function signal F6=bit **57**, function signal F4=bit **55**, function signal F4=bit **53** and function signal F2=bit **51**. The other function signals are set by **F7=~F6**, **F5=~F4**, **F3=~F2** and **F1=~F0**. These operations and their corresponding function signals are shown in Table 21. Table 21 also shows the modifications to the default coding.

8-bit ALU code field				Derived Function Signal		
5 7	5 5	5 3	5 1	FFFFFFFF 76543210	Hex	Description of operation
0	0	0	0	10101010	AA	reserved for non-arithmetic logic unit operations
0	0	0	1	10101001	A9	A–B shift left "1" extend
0	0	1	0	10100110	A6	A+B shift left "0" extend
0	0	1	1	10100101	A5	A–C
0	1	0	0	10011010	9A	A–B shift right "1" extend if sign=0 flips to 95 A–B shift right sign extend
0	1	0	1	10011001	99	A–B
0	1	1	0	10010110	96	A+B/A–B depending on C if ~@MF flips to 99 A–B if sign=1 A+|B|
0	1	1	1	10010101	95	A–B shift right "0" extend
1	0	0	0	01101010	6A	A+B shift right "0" extend
1	0	0	1	01101001	69	A–B/A+B if ~1EMF flips to 66 A+B if sign=1 A–|B|
1	0	1	0	01100110	66	A+B
1	0	1	1	01100101	65	A+B shift right "1" extend if sign=0 flips to 6A A+B shift right sign extend
1	1	0	0	01011010	5A	A+C
1	1	0	1	01011001	59	A–B shift left "0" extend
1	1	1	0	01010110	56	A+B shift left "1" extend
1	1	1	1	01100000	60	(A&C) + (B&C), field A+B

Several codings of instruction word bits **57**, **55**, **53** and **51** are executed in modified form as shown in Table 21. Note that the functions that list left or right shifts are employed in conjunction with barrel rotator **235** and mask generator **238**. These operations will be explained in detail below. The "sign" referred to in this description is bit **31** of arithmetic logic unit second input bus **206**, the bus driving barrel rotator **235**. This is the sign bit of a signed number. A "0" in this sign bit indicates a positive number and a "1" in this sign bit indicates a negative (two's complement) number. A bit **57**, **55**, **53** and **51** state of "0100" results in a normal function of A–B with shift right "1" extend. If bit **31** of arithmetic logic unit second input bus **206** is "0", then the operation changes to A–B with shift right sign extend. A bit **57**, **55**, **53** and **51** state of "0110" results in a normal function of A–B or A+B depending on the bit wise state of C. If the instruction does not specify a multiple flags register mask operation (@MF) then the operation changes to A–B. If bit **31** of arithmetic logic unit second input bus **206** is "1", then the operation changes to A+|B| (A plus the absolute value of B). A bit **57**, **55**, **53** and **51** state of "1011" results in a normal function of A+B or A–B depending on the bit wise state of C. If the instruction does not specify a multiple flags register mask operation (~@MF) then the operation changes to A+B. If bit **31** of arithmetic logic unit second input bus **206** is "1", then the operation changes to A–|B| (A minus the absolute value of B). A bit **57**, **55**, **53** and **51** state of "1001" results in a normal function of A+B with shift right "1" extend. If bit **31** of arithmetic logic unit second input bus **206** is "0", then the operation changes to A+B with shift right sign extend.

Two codes are modified to provide more useful functions. A bit **57**, **55**, **53** and **51** state of "0000" results in a normal function of ~A (not A), which is reserved to support non-arithmetic logic unit operations as described below. A bit **57**, **55**, **53** and **51** state of "1111" results in a normal function of A. This is modified to (A&C)+(B&C) or a field add of A and B controlled by the state of C.

The base set of operations listed in Table 21 may be specified in arithmetic instructions. Note that instruction word bits 58, 56, 54 and 52 control modifications of these basic operations as set forth in Table 6. These modifications were explained above in conjunction with Table 6 and the description of status register 210. As further described below certain instructions specify extended arithmetic logic unit operations. It is still possible to specify each of the 256 arithmetic operations via an extended arithmetic logic unit (EALU) operation. For these instructions the "A" (bit 27) of data register D0 specifies either an arithmetic or Boolean logic operation, the "EALU" field (bits 26–19) specifies the function signals F7–F0 and the "FMOD" field (bits 31–28) specifies modifications of the basic function. Also note that the "C", "T", "S", "N" and "E" fields of data register D0 permit control of the carry-in to bit 0 of arithmetic logic unit 230 and to the least significant bit of each section if multiple arithmetic is enabled. There are four forms of extended arithmetic logic unit operations. Two of these specify parallel multiply operations using multiplier 220. In an extended arithmetic logic unit true (EALUT) operation, the function signals F7–F0 equal the corresponding bits of the "EALU" field of data register D0. In an extended arithmetic logic unit false (EALUF) operation, the individual bits of the "EALU" field of data register D0 are inverted to form the function signals F7–F0. The extended arithmetic logic unit false operation is useful because during some algorithms the inverted functions signals perform a useful related operation. Inverting all the function signals typically specifies an inverse function. Thus this related operation may be accessed via another instruction without reloading data register 208. In the other extended arithmetic logic unit operations the function signals F7–F0 equal the corresponding bits of the "EALU" field of data register D0, but differing data paths to arithmetic logic unit 230 are enabled. These options will be explained below.

Data unit 110 operation is responsive to instruction words fetched by program flow control unit 130. Instruction decode logic 250 receives data corresponding to the instruction in the execute pipeline stage via opcode bus 133. Instruction decode logic 250 generates control signals for operation of multiplexers Fmux 221, Imux 222, MSmux 225, Bmux 227, Amux 232, Cmux 233, Mmux 234 and Smux 231 according to the received instruction word. Instruction decode logic 250 also controls operation of buffers 104, 106, 108, 223 and 236 according to the received instruction word. Control lines for these functions are omitted for the sake of clarity. The particular controlled functions of the multiplexers and buffers will be described below on description of the instruction word formats in conjunction with FIG. 43. Instruction decode logic 250 also supplies partially decoded signals to function signal generator 245 and bit 0 carry-in generator 246 for control of arithmetic logic unit 230. Particular hardware for this partial decoding is not shown, however, one skilled in the art would be able to provide these functions from the description of the instruction word formats in conjunction with FIG. 43. Instruction decode logic 250 further controls the optional multiple section operation of arithmetic logic unit 230 by control of multiplexers 311, 312, 313 and 314, previously described in conjunction with FIG. 7.

FIG. 22 illustrates details of the function signal selector 245*a*. Function signal selector 245*a* forms a part of function signal generator 245 illustrated in FIG. 5. For a full picture of function signal generation, FIG. 22 should be considered with the function signal modifier 245*b* illustrated in FIG. 23. Multiplexers are shown by rectangles having an arrow representing the flow of bits from inputs to outputs. Inputs are designated with lower case letters. Control lines are labeled with corresponding upper case letters drawn entering the multiplexer rectangle perpendicular to the arrow. When a control line designated with a particular upper case letter is active, then the input having the corresponding lower case letter is selected and connected to the output of the multiplexer.

Input "a" of multiplexer Omux 500 receives an input in two parts. Bits 57, 55, 53 and 51 of the instruction word are connected to bit lines 6, 4, 2 and 0 of input "a", respectively. Invertor 501 inverts the respective instruction word bits and supplies them to bit lines 7, 5, 3 and 1 of input "a". Input "a" is selected if control line "A" goes active, and when selected the eight input bit lines are connected to their eight corresponding numbered output bit lines 7–4 and 3–0. Control line "A" is fed by AND gate 502. AND gate 503 receives a first input indicating execution of an instruction in any of the instruction classes 7–0. Instruction word bit 63 indicates this. These instruction classes will be further described below. AND gate 502 has a second input fed by bit 59 of the instruction word. As will be explained below, a bit 59 equal to "1" indicates an arithmetic operation. NAND gate 503 supplies a third input to AND gate 502. NAND gate 503 senses when any of the four instruction word bits 57, 55, 53 or 51 is low. Control input "A" is thus active when any of the instruction classes 7–0 is selected, and arithmetic bit 59 of the instruction word is "1" and instruction word bits 57, 55, 53 and 51 are not all "1". Recall from Table 21 that a bit 57, 55, 53 and 51 state of "1111" results in the modified function signals Hex "60" rather than the natural function signals.

Input "b" to multiplexer Omux 500 is a constant Hex "60". Multiplexer Omux 500 selects this input if AND gate 504 makes the control "B" active. AND gate 504 makes control "B" active if the instruction is within classes 7–0 as indicate by instruction word bit 63, the instruction word bit 59 is "1" indicating an arithmetic operation, and a bit 57, 55, 53 and 51 state of "1111". As previously described in conjunction with Table 21, under these conditions the function Hex "60" is substituted for the function signals indicated by the instruction.

Input "c" to multiplexer Omux 500 receives all eight instruction word bits 58–51. Multiplexer Omux 500 selects this input if AND gate 505 makes control "C" active. AND gate 505 receives instruction word bit 59 inverted via invertor 506 and an indication of any of the instruction classes 7–0. Thus instruction word bits 58–51 are selected to perform any of the 256 Boolean operations in instruction classes 7–0.

Instruction words for the operations relevant to control inputs "D", "E", "F", "G" and "H" have bits 63–61 equal to "011". If this condition is met, then bits 60–57 define the type of operation. These operations are further described below in conjunction with Table 35.

Input "d" to multiplexer Omux 500 is a constant Hex "66". This input is selected for instructions that execute a parallel signed multiply and add (MPYS||ADD) or a parallel unsigned multiply and add (MPYU||ADD). These instructions are collectively referred to by the mnemonic MPYx||ADD.

Input "e" to multiplexer Omux 500 is a constant Hex "99". This input is selected for instructions that execute a parallel signed multiply and subtract (MPYS||SUB) or a parallel unsigned multiply and subtract (MPYU||SUB). These instructions are collectively referred to by the mnemonic

Input "f" to multiplexer Omux **500** is a constant Hex "A6". This input is selected for the DIVI operation. The operation of this DIVI operation, which is employed in division, will be further described below.

Input "g" to multiplexer Omux **500** is supplied from the "EALU" field (bits **26–19**) of data register **D0** according to an extended arithmetic logic unit function code from bits **26–19** therein. Control input "G" goes active to select this "EALU" field from data register **D0** if OR gate **507** detects either a MPYx||EALUT operation or and an EALU operation. As previously described, the T suffix in EALUT signifies EALU code true in contrast to the inverse (false) in EALUF. The EALU input is active to control input "G" when the "EALU" field of data register **D0** indicates either EALU or EALU %.

Invertor **508** inverts the individual bits of the "EALU" field of data register **D0** for supply to input "h" of multiplexer Omux **500**. Input "h" of multiplexer Omux **500** is selected in response to detection of a MPYx||EALUF operation at control input "H". As previously described, the F suffix of EALUF indicates that the individual bits of the "EALU" field of register **D0** are inverted for specification of function signals **F7–F0**.

Multiplexer AEmux **510**, which is also illustrated in FIG. **22**, generates the arithmetic enable signal. This arithmetic enable signal is supplied to tri-state AND gate **480** of every bit circuit **400**. The "a" input to multiplexer AEmux **510** is the "A" bit (bit **27**) of data register **D0**. OR gate **511** receives three inputs: MPYx||EALUT, EALU, and MPYx||EALUF. If the instruction selects any of these three operations, then control input "A" to multiplexer AEmux selects the "A" bit (bit **27**) of data register **D0**. The "b" input to multiplexer AEmux **510** is the "ari" bit (bit **59**) of the instruction word. As will be described below, this "ari" bit selects arithmetic operations for certain types of instructions. This input is selected if the instruction is any of the instruction classes **7–0**. In this case the "ari" bit signifying an arithmetic operation ("ari"="1") or a Boolean operation ("ari"="0") is passed directly to the arithmetic logic unit **230**. The "c" input of multiplexer AEmux **510** is a constant "1". The gate **512** selects this input if the instruction is neither an extended arithmetic logic unit instruction nor within instruction classes **7–0**. Such instructions include the DIVI operation and the MPYx||ADD and MPYx||SUB operations. OR gate **513** provides an arithmetic or EALU signal when the instruction is either an arithmetic operation as indicated by the output of multiplexer AEmux **510** or an "any EALU" operation as indicated by OR gate **511**.

FIG. **23** illustrates function signal modifier **245*b***. Function signal modifier **245*b*** modifies the function signal set from function signal generator **245*a*** according to the "FMOD" field of data register **D0** or the instruction bits **58**, **56**, **54** and **52** depending on the instruction. Multiplexer Fmux **520** selects the function modifier code.

The "a" input to multiplexer Fmux **520** is all "0's" (Hex "0"). NOR gate **521** supplies control line "A" of multiplexer Fmux **520**. NOR gate **521** has a first input receiving the "any EALU" signal from OR gate **511** illustrated in FIG. **22** and a second input connected to the output of AND gate **522**. AND gate **522** receives a first input from the "ari" bit (bit **59**) of the instruction word and a second input indicating the instruction is in instruction classes **7–0**. Thus NOR gate **521** generates an active output that selects the Hex "0" input to Fmux **520** if the instruction is not any extended arithmetic logic unit operation and either the "ari" bit of the instruction word is "0" or the instruction is not within instruction classes class **7–0**.

The "b" input to multiplexer Fmux **520** receives bits **58**, **56**, **54** and **52** of the instruction word. The control input "B" receives the output of AND gate **522**. Thus multiplexer Fmux **520** selects bits **58**, **56**, **54** and **52** of the instruction word when the instruction is in any instruction class **7–0** and the "ari" bit of the instruction is set.

The "c" input of multiplexer Fmux **520** receives bits of the "FMOD" field (bits **31–28**) of data register **D0**. The control input "C" receives the "any EALU" signal from OR gate **511**. Multiplexer Fmux **520** selected the "FMOD" field of data register **D0** if the instruction calls for any extended arithmetic logic unit operation.

Multiplexer Fmux **520** selects the active function modification code. The active function modification code modifies the function signals supplied to arithmetic logic unit **230** as described below. The function modification code is decoded to control the operations specified in Table 6. As explained above, these modified operations include controlled splitting of arithmetic logic unit **230**, setting one or more bits of multiple flags register **211** by zero(es) or carry-out(s) from arithmetic logic unit **230**, rotating or clearing multiple flags register **211**, operating LMO/RMO/LMBC/RMBC circuit **237** in one of its four modes, operating mask generation **239** and operating bit **0** carry-in generator **246**. The operations performed in relation to a particular state of the function modification code are set forth in Table 6.

Three circuit blocks within function modifier **245*b*** may modify the function signals **F7–F0** from multiplexer Omux **500** illustrated in FIG. **22**. Mmux block **530** may operate to effectively set the input to the C-port to all "1's". A-port block **540** may operate to effectively set the input to the A-port to all "0's". Sign extension block **550** is a sign extension unit that may flip function signals **F3–F0**.

Mmux block **530** includes a multiplexer **531** that normally passes function signals **F3–F0** without modification. To effectively set the input to the C-port of arithmetic logic unit **230** to "1's", multiplexer **531** replicates function signals **F7–F4** onto function signals **F3–F0**. Multiplexer **531** is controlled by AND gate **533**. AND gate **533** is active to effectively set the input to the C-port to all "1's" provided all three of the following conditions are present: 1) the function modifier code multiplexer Fmux **520** is any of the four codes "0010", "0011", "0110" or "0111" as detected by "0X1X" match detector **532** (X=don't care); 2) the instruction calls for a mask generation operation; and 3) the output from multiplexer Mmux **234** is "0". As previously described above, duplication of functions signals **F7–F4** onto function signals **F3–F0**, that is selection of **F7=F3**, **F6=F2**, **F5=F1** and **F4=F0**, enables selection of the 16 Boolean logic functions of inputs A and B independent of input C. Note from Table 6 that the four function modifier codes "0X1X" include the "%!" modification. According to FIG. **23**, the "%!" modification is achieved by changing the function signals sent to arithmetic logic unit **230** rather than by changing the mask generated by mask generator **239**.

A-port block **540** includes multiplexer **541** and connection circuit **542** that normally pass function signals **F7–F0** without modification. To effectively set the input to the A-port of arithmetic logic unit **230** to all "0's", multiplexer **541** and connection circuit **541** replicates function signals **F6**, **F4**, **F2** and **F0** onto function signals **F7**, **F5**, **F3** and **F1**, respectively. Multiplexer **541** and connection circuit **542** make this substitution when activated by OR gate **544**. OR gate **544** has a first input connected to "010X" match detector **543**, and a second input connected to AND gate **546**. AND gate **546** has a first input connected to "011X"

match detector **545**. Both match detectors **543** and **545** determine whether the function modifier code matches their detection state. AND gate **546** has a second input that receives a signal indicating whether the instruction calls for a mask generation operation. The input to the A-port of arithmetic logic unit **230** is effectively zeroed by swapping function signals F6, F4, F2 and F0 for function signals F7, F5, F3 and F1, respectively. As previously described, this substitution makes the output of arithmetic logic unit **230** independent of the A input. This substitution takes place if: 1) the function modifier code finds a match in "010X" match detector **543**; or 2) the instruction calls for a mask generation operation and the function modifier code find a match in "010X" match detector **545** and the instruction calls for a mask generation operation.

Sign extension block **550** includes exclusive OR gate **551**, which normally passes function signals F3–F0 unmodified. However, these function signals F3–F0 are inverted for arithmetic logic unit sign extension and absolute value purposes under certain conditions. Note that function signals F7–F4 from A-port block **540** are always passed unmodified by sign extension block **550**. AND gate **552** controls whether exclusive OR gate **551** inverts function signals F3–F0. AND gate **552** has a first input receiving the arithmetic or extended arithmetic logic unit signal from OR gate **513** illustrated in FIG. **22**. The second input to AND gate **552** is from multiplexer **553**.

Multiplexer **553** is controlled by the "any EALU" signal from OR gate **511** of FIG. **22**. Multiplexer **553** selects a first signal from AND gate **554** when the "any EALU" signal is active and selects a second signal from compound AND/OR gate **556** when the "any EALU" signal is inactive. The output of AND gate **554** equals "1" when the data on arithmetic logic unit second input bus **206** is positive, as indicated by the sign bit (bit **31**) as inverted by invertor **555**, and the "S" bit (bit **16**) of data register D0 is "1". The output of compound AND/OR gate **556** is active if: 1) the data on arithmetic logic unit second input bus **206** is positive, as indicated by the sign bit (bit **31**) as inverted by invertor **555**; 2) the instruction is within instruction classes **7–0**; and 3) either a) instruction bits **57**, **55**, **53** and **51** find a match in "0100"/"1011" match detector **557** or b) AND gate **560** detects that instruction word bits **57**, **55**, **53** and **51** find a match in "1001"/"0110" match detector **558**, and the instruction does not call for a multiple flags register mask operation (@MF) as indicated by invertor **559**.

Sign extension block **550** implements the exceptions noted in Table 21. An inactive "any EALU" signal, which indicates that the instruction specified an arithmetic operation, selects the second input to multiplexer **553**. Compound AND/OR gate **556** determines that the instruction is within instruction classes **7–0** and that the sign bit is "0". Under these conditions, if instruction word bits **57**, **55**, **53** and **51** equal "0100" and then the function signal flips from Hex "9a" to Hex "95" by inverting function signal bits F3–F0. Similarly, if instruction word bits **57**, **55**, **53** and **51** equal "1011" and then the function signal flips from Hex "65" to Hex "6a" by inverting function signal bits F3–F0. If instruction word bits **57**, **55**, **53** and **51** equal "1001" and the instruction does not call for a multiple flags register mask operation as indicated by invertor **599**, then the function signal flips from Hex "69" to Hex "66". This set of function signals causes arithmetic logic unit **230** to implement A–|B|, A minus the absolute value of B. If instruction word bits **57**, **55**, **53** and **51** equal "0110" and the instruction does not call for a multiple flags register mask operation, then the function signal flips from Hex "96" to Hex "99". This executes the function A+|B|, A plus the absolute value of B. Note that these flips of the function signals are based on the sign bit (bit **31**) of the data on arithmetic logic unit second input bus **206**.

FIG. **24** illustrates bit **0** carry-in generator **246**. As previously described bit, **0** carry-in generator **246** produces the carry-in signal c_{in} supplied to the first bit of arithmetic logic unit **230**. In addition this carry-in signal c_{in} from bit **0** carry-in generator **246** is generally supplied to the first bit of each of the multiple sections, if the instruction calls for a multiple arithmetic logic unit operation. Multiplexer Zmux **570** selects one of six possible sources for this bit **0** carry-in signal c_{in} based upon six corresponding controls inputs from instruction decode logic **250**.

Input "a" of multiplexer Zmux **570** is supplied with bit **31** of multiple flags register **211**. Multiplexer Zmux **570** selects this input as the bit **0** carry-in signal c_{in} if the instruction calls for a DIVI operation.

Inputs "b", "c" and "d" to multiplexer Zmux **570** are formed of compound logic functions. Input "b" of multiplexer Zmux **570** receives a signal that is a Boolean function of the function signals F6, F2 and F0. This Boolean expression, which is formed by circuit **571**, is (F0 & ~F6)|(F0 & ~F2)|(~F2 & ~F6). Input "c" of multiplexer Zmux **570** is fed by exclusive OR gate **572**, which has a first input supplied by exclusive OR gate **573** and a second input supplied by AND gate **574**. The exclusive OR gate **573** has as a first input the "C" bit (bit **18**) of data register D0, which indicates whether the prior operation of arithmetic logic unit **230** produced a carry-out signal c_{out} at bit **31**, the last bit. The second input of XOR gate **573** receives a signal indicating the instruction calls for a MPYx||EALUF operation. AND gate **574** has a first input from invertor **575** inverting the sign bit (bit **31**) present on arithmetic logic unit second input bus **206** for detecting a positive sign. AND gate **574** has a second input from the "T" bit (bit **17**) of data register D0 and a third input from the "S" bit (bit **16**) of data register D. As explained above, the "T" bit causes inversion of carry-in when the "S" bit indicates sign extend is enabled. This operation complements the sign extend operation of AND gate **554** and XOR gate **551** of the function modifier **246*b*** illustrated in FIG. **23**. Input "d" of multiplexer Zmux **570** comes from XOR gate **576**. XOR gate **576** has a first input supplied the function signal F0 and a second input supplied bit **0** of the data on input C bus **243**.

Input "b" of multiplexer Zmux **570** is selected when AND gate **581** sets control input "B" active. This occurs when the "arithmetic or EALU" from OR gate **513** is active, the instruction does not call for an extended arithmetic logic unit operation as indicated by invertor **582** and no other multiplexer Zmux **570** input is applicable as controlled by invertors **583**, **584** and **585**.

Input "c" of multiplexer Zmux **570** is selected when AND gate **586** supplies an active output to control input "C". AND gate **586** is responsive to a signal indicating the instruction calls for "any EALU" operation. The rest of the inputs to AND gate **586** assure that AND gate **586** is not active if any of inputs "d", "e" or "f" are active via invertors **584**, **585** and **595**.

Input "d" of multiplexer Zmux **570** is selected when control line "D" is from AND gate **587**. AND gate **587** is active when the instruction is an arithmetic operation or an extended arithmetic logic unit operation, AND gate **589** is active and input "e" is not selected as indicated by invertor **585**. AND gate **589** is active when the instruction specifies a multiple flags register mask operation (@MF) expansion and instruction word bits **57**, **55**, **53** and **51** find a match in

"0110"/"1001" match circuit **588**. These instruction word bits correspond to function signals Hex "69" and Hex "96", which cause addition or subtraction between ports A and B depending on the input to port C. No function signal flipping is involved since the instruction class involves multiple flags register expansion. FIG. **7** illustrates providing this carry-in signal to plural sections of a split arithmetic logic unit in multiple mode.

Input "e" of multiplexer Zmux **570** comes from the "C" bit (bit **30**) of status register **210**. As previously described, this "C" bit of status register **210** is set to "1" if the result of the last operation of arithmetic logic unit **230** caused a carry-out from bit **31**. AND gate **594** supplies control input "E". AND gate **594** goes active when the instruction specifies an arithmetic operation or an extended arithmetic logic unit operation and the following logic is true: 1) the function modifier code finds a match in "0X01" match detector **591**; or (OR gate **590**) 2) the instruction calls for a mask generation operation and (AND gate **593**) the function modifier code finds a match in "0X11" match detector **592**.

Input "f" of multiplexer Zmux **570** is supplied with a constant "0". Multiplexer Zmux **570** selects this input when the "arithmetic or EALU" signal from OR gate **513** indicates the instruction specifies a Boolean operation as inverted by invertor **595**.

The output of Zmux **570** normally passes through Ymux **580** unchanged and appears at the bit **0** carry-in output. In a multiple arithmetic operation in which data register **D0** "A" bit (bit **27**) and "E" bit (bit **14**) are not both "1", Ymux produces plural identical carry-in signals. Selection of half word operation via "Asize" field of status register **210** causes Ymux to produce the supply the output of Zmux **570** to both the bit **0** carry-in output and the bit **16** carry-in output. Likewise, upon selection of byte operation Ymux **580** supplies the output of Zmux **570** to the bit **0** carry-in output, the bit **8** carry-in output, the bit **16** carry-in output and the bit carry-in output.

The operation of Ymux **580** differs when data register **D0** "A" bit (bit **27**) and "E" bit (bit **14**) are both "1". AND gate **577** forms this condition and controls the operation of Ymux **580**. This is the only case in which the carry-in signals supplied to different sections of arithmetic logic unit **230** during multiple arithmetic differ. If AND gate **577** detects this condition, then the carry-in signals are formed by the exclusive OR of function signal **F0** and the least significant bit of the C input of the corresponding section of arithmetic logic unit **230**. If the "Asize" field selects word operation, that is if arithmetic logic unit **230** forms a single 32 bit section, then the bit **0** carry-in output formed by Ymux **580** is the exclusive OR of function signal **F0** and input C bus bit **0** formed by XOR gate **596**. No other carry-in signals are formed. If the "Asize" field selects half word operation forming two 16 bit sections, then the bit **0** carry-in output formed by Ymux **580** is the output of XOR gate **596** and the carry-in to bit **16** is the exclusive OR of function signal **F0** and input C bus bit **16** formed by XOR gate **598**. Lastly, for byte multiple arithmetic the bit **0** carry-in output formed by Ymux **580** is the output of XOR gate **596**, the bit **8** carry-in is formed by XOR gate **597**, and the bit **16** carry-in is formed by XOR gate **598** and the bit **24** carry-in is formed by XOR gate **599**.

FIGS. **22**, **23** and **24** not only represent specific blocks implementing the Tables but also illustrates the straightforward process by which the Tables and Figures compactly define logic circuitry to enable the skilled worker to construct the preferred embodiment even when a block diagram of particular circuitry may be absent for conciseness. Note that the circuits of FIGS. **22** and **23** do not cover control for the various multiplexers and special circuits via instruction decode logic **250** that are a part of data unit **110** illustrated in FIG. **5**. However, control of these circuits is straight forward and within the capability of one of ordinary skill in this art. Therefore these will not be further disclosed for the sake of brevity.

Arithmetic logic unit **230** includes three 32 bit inputs having differing hardware functions preceding each input. This permits performance of many different functions using arithmetic logic unit **230** to combine results from the hardware feeding each input. Arithmetic logic unit **230** performs Boolean or bit by bit logical combinations, arithmetic combinations and mixed Boolean and arithmetic combinations of the 3 inputs. Mixed Boolean and arithmetic functions will hereafter be called arithmetic functions due to their similarity of execution. Arithmetic logic unit **230** has one control bit that selects either Boolean functions or arithmetic functions. Boolean functions generate no carries out of or between bit circuits **400** of arithmetic logic unit **230**. Thus each bit circuit **400** of arithmetic logic unit **230** combines the 3 inputs to that bit circuit independently forming 32 individual bit wise results. During arithmetic functions, each bit circuit **400** may receive a carry-in from the adjacent lesser significant bit and may generate a carry-out to the next most significant bit location. An 8 bit control signal (function control signals **F7–F0**) control the function performed by arithmetic logic unit **230**. This enables selection of one of 256 Boolean functions and one of 256 arithmetic functions. The function signal numbering of function signals **F7–F0** is identical to that used in Microsoft® Windows. Bit **0** carry-in generator **246** supplies carry-in signals when in arithmetic mode. in arithmetic mode, arithmetic logic unit **230** may be split into either two independent 16 bit sections or four independent 8 bit sections to process in parallel multiple smaller data segments. Bit **0** carry-in generator 246 supplies either one, two or four carry-in signals when arithmetic logic unit **230** operates in one, two or four sections, respectively. In the preferred embodiment, an assemblier for data unit **110** includes an expression evaluator that selects the proper set of function signals based upon an algebraic input syntax.

The particular instruction being executed determines the function of arithmetic logic unit **230**. As will be detailed below, in the preferred embodiment the instruction word includes a field that indicates either Boolean or arithmetic operations. Another instruction word field specifies the function signals supplied to arithmetic logic unit **230**. Boolean instructions specify the 8 function signals **F7–F0** directly. In arithmetic instructions a first subset of this instruction word field specifies a subset of the possible arithmetic logic unit operations according to Table 21. A second subset of this instruction word field specifies modifications of instruction function according to Table 6. All possible variations of the function signals and the function modifications for both Boolean and arithmetic instructions may be specified using an extended arithmetic logic unit (EALU) instruction. In this case the predefined fields within data register **D0** illustrated in FIG. **9** specify arithmetic logic unit **230** operation.

Though arithmetic logic unit **230** can combine all three inputs, many useful functions don't involve some of the inputs. For example the expression A&B treats the C input as a don't care, and the expression A|C treats the B input as a don't care. Because different data path hardware precedes each input, the ability to use or ignore any the inputs supports the selection of data path hardware needed for the desired function. Table 22 shows examples of useful three

input expressions where the C-input is treated as a mask or a merging control. Because data unit **110** includes expand circuit **238** and mask generator **239** in the data path of the C-input of arithmetic logic unit **230**, it is natural to employ the C-input as a mask.

TABLE 22

Logical Function	Typical use
(A&C)|(B&~C)	Bit by bit multiplexing (merge) of A and B based on C. A chosen if corresponding bit in C is 1
(A&~C)|(B&C)	Bit by bit multiplexing (merge) of A and B based on C. B chosen if corresponding bit in C is 1
(A|B)&~C	Logic OR of A and B and then force to 0 everywhere that C is a 1
(A&B)&~C	Logic AND of A and B and then force to 0 everywhere C is a 1
A|(B&C)	If C is 0 then force the B-input to 0 before logical ORing with A
A|(B|~C)	If C is 0 then force the B-input to 1 before logical ORing with A

The three input arithmetic logic unit **230** can perform mixed Boolean and arithmetic functions in a single pass through arithmetic logic unit **230**. The mixed Boolean and arithmetic functions support performing Boolean functions prior to an arithmetic function. Various compound functions such as shift and add, shift and subtract or field masking prior to adding or subtracting can be performed by the appropriate arithmetic logic unit function in combination with other data path hardware. Note arithmetic logic unit **230** supports 256 different arithmetic functions, but only a subset of these will be needed for most programming. Additionally, further options such as carry-in and sign extension need to be controlled. Some examples expected to be commonly used are listed below in Table 23.

TABLE 23

Func Code Hex	Function	Default Carry-In	Common Use
66	A+B	0	A+B ignore C
99	A−B	1	A−B ignore C
5A	A+C	0	A+C ignore B
A5	A−C	1	A−C ignore B
6A	A+(B&C)	0	A+B shift right "0" extend C shift mask
95	A−(B&C)	1	A−B shift right "0" extend C shift mask
56	A+(B|C)	0	A+B shift left "0" extend C shift mask
A9	A−(B|C)	1	A−B shift left "1" extend C shift mask
A6	A+(B&~C)	0	A+B shift left "0" extend C shift mask
59	A−(B&~C)	1	A−B shift left "0" extend C shift mask
65	A+(B|~C)	0	A+B shift right sign extend C shift mask
9A	A−(B|~C)	1	A−B shift right sign extend C shift mask
60	(A&C)+(B&C)	0	A+B mask by C

TABLE 23-continued

Func Code Hex	Function	Default Carry-In	Common Use
9F	(A&C)−(B&C)	1	A−B mask by C
06	(A&~C)+(B&~C)	0	A+B mask by ~C
F9	(A&~C)−(B&~C)	1	A−B mask by ~C
96	A+((−B&C)|(B&~C))	LSB of C	A+B or A−B based on ~C
69	A+((B&C)|(−B&~C))	LSB of ~C	A+B or A−B based on C
CC	B	0	B ignore A and C
33	~B	1	Negative B ignore A and C
F0	C	0	C ignore A and B
0F	−C	1	Negative C ignore A and B
C0	(B&C)	0	B shift right "0" extend C shift mask
3F	−(B&C)	1	Negative B shift right "0" extend C shift mask
FC	(B|C)	0	B shift left "1" extend C shift mask
03	−(B|C)	1	Negative B shift left "1" extend C shift mask
0C	(B&~C)	0	B shift left "0" extend C shift mask
F3	−(B&~C)	1	Negative B shift left "0" extend C shift mask
CF	(B|~C)	0	B shift right sign extend C shift mask
30	−(B|~C)	1	Negative B shift right sign extend C shift mask
3C	(−B&C)|(B&~C)	LSB of C	−B or B based on ~C
C3	(B&C)|(−B&~C)	LSB of ~C	B or −B based on C

The most generally useful set of arithmetic functions combined with default carry-in control and sign extension options are available directly in the instruction set in a base set of operations. These are listed in Table 21. This base set include operations that modify the arithmetic logic unit's functional controls based on sign bits and that use default carry-in selection. Some examples of these are detailed below.

All 256 arithmetic functions along with more explicit carry-in and sign extension control are available via the extended arithmetic logic unit (EALU) instruction. In extended arithmetic logic unit instructions the function control signals, the function modifier and the explicit carry-in and sign extension control are specified in data register **D0**. The coding of data register **D0** during such extended arithmetic logic unit instructions is described above in relation to FIG. **9**.

Binary numbers may be designated as signed or unsigned. Unsigned binary numbers are non-negative integers within the range of bits employed. An N bit unsigned binary number may be any integer between 0 and 2^N-1. Signed binary numbers carry an indication of sign in their most significant bit. If this most significant bit is "0" then the number is positive or zero. If the most significant bit is "1" then the number is negative or zero. An N bit signed binary number may be any integer from -2^{N-1} to $2^{N-1}-1-1$. Knowing how and why numbers produce a carry out or overflow is important in understanding operation of arithmetic logic unit **230**.

The sum of two unsigned numbers overflows if the sum can no longer be expressed in the number of bits used for the numbers. This state is recognized by the generation of a carry-out from the most significant bit. Note that arithmetic logic unit **230** may be configured to operation on numbers of 8 bits, 16 bits or 32 bits. Such carry-outs may be stored in Mflags register **211** and employed to maintain precision. The difference of two unsigned numbers underflows when the difference is less than zero. Note that negative numbers cannot be expressed in the unsigned number notation. The examples below show how carry-outs are generated during unsigned subtraction.

The first example shows 7 "00000111" minus 5 "00000110". Arithmetic logic unit **230** performs subtraction by two's complement addition. The two's complement of an unsigned binary number can be generated by inverting the number and adding 1, thus $-X=\sim X+1$. Arithmetic logic unit **230** negates a number by logically inverting (or one's complementing) the number and injecting a carry-in of 1 into the least significant bit. First the 5 is bit wise inverted producing the one's complement "11111001". Arithmetic logic unit **230** adds this to 7 with a "1" injected into the carry-in input of the first bit. This produces the following result.

```
    00000111     7
+   11111010    -5
+          1
  100000010     2
```

Note that this produces a carry-out of "1" from the most significant bit. In two's complement subtraction, such a carry-out indicates a not-borrow. Thus there is no underflow during this subtraction. The next example shows 7–5. Note that the 8 bit one's complement of "00000111 "is "11111000".

```
    00000101     5
+   11111000    -7
+          1
  011111110    -2
```

In this case the carry-out of "0" indicates a borrow, thus the result is less than zero and an underflow has occurred. The last example of unsigned subtraction is 0–0. Note that the 8 bit one's complement of 0 is "11111111".

```
    00000000     0
+   11111111    -0
+          1
  000000000     0
```

The production of a carry-out of "1" indicates no underflow.

The situation for signed numbers is more complex. An overflow on a signed add occurs if both operands are positive and the sign bit of the result is a 1 (i.e., negative) indicating that the result has rolled over from positive to negative. Overflow on an add also occurs if both operands are negative and the result has a 0 (i.e., positive) sign bit. Or in other words overflow on addition occurs if both of the sign bits of the operands are the same and the result has a different sign bit. Similarly a subtraction of can overflow if the operands have the same sign and the result has a different sign bit.

When setting the carry bit in status register **210** or in the Mflags register **211**, the bit or bits are always the "natural" carry outs generated by arithmetic logic unit **230** Most other microprocessors set "carry status" based upon the carry-out bit during addition but set it based upon not-carry-out (or borrow) during subtraction. These other microprocessors must re-invert the not-carry when performing subtract with borrow to get the proper carry-in to the arithmetic logic unit. This difference results in a slightly different set of conditional branch equations using this invention than other processors to get the same branch conditions. Leaving the sense of carries/not-borrows the same as those generated by arithmetic logic unit **230** simplifies many ways in which each digital image/graphics processor can utilize them.

In the base set of arithmetic instructions, the default carry-in is "0" for addition and "1" for subtraction. The instruction set and the preferred embodiment of the assembler will automatically set the carry-in correctly for addition or subtraction in 32-bit arithmetic operations. The instruction set also supports carry-in based on the status registers carry-out to support multiple precision add-with-carry or subtract-with-borrow operations.

As will be explained in more detail later, some functions arithmetic logic unit **230** support the C-port controlling whether the input to the B-port is added to or subtracted from the input to the A-port. Combining these arithmetic logic unit functions with multiple arithmetic permits the input to the C-port to control whether each section of arithmetic logic unit **230** adds or subtracts. The base set of operations controls the carry-in to each section of arithmetic logic unit **230** to supply a carry-in of "0" that section is performing addition and a carry-in of "1" if that section is performing subtraction. The hardware for supplying the carry-in to these sections is described above regarding FIG. **24**.

The following details the full range of arithmetic functions possible using digital image/graphics processor **71** 3-input arithmetic logic unit **230**. For most algorithms, the subset of instructions listed above will be more than adequate. The more detailed description following is included for completeness.

Included in the description below is information about how to derive the function code for arithmetic logic unit **230**. Some observations about function code **F7–F0** will be helpful in understanding how arithmetic logic unit **230** can be used for various operations and how to best use extended arithmetic logic unit instructions. The default carry-in is equal to **F0**, the least significant bit of the function code, except for the cases where the input to the C-port controls selection of addition or subtraction between A and B. Inverting all the function code bits changes the sign of the operation. For example the function codes Hex "66", which specifies A+B, and Hex "99", which specifies A–B, are bit wise inverses. Similarly, function code Hex "65" (A+(B|~C)) and Hex "9A" (A–(B|~C)) are bit wise inverses. Extended arithmetic logic unit instructions come in the pairs of extended arithmetic logic unit true (EALUT) and extended arithmetic logic unit false (EALUF). The extended arithmetic logic unit false instruction inverts the arithmetic logic unit control code stored in bits **26–19** of data register **D0**. As noted above, this inversion generally selects between addition and subtraction. Inverting the 4 least significant bits of the function code Hex "6A" for A+(B&C) yields gives Hex "65" that is the function A+(B|~C). Similarly, inverting the 4 least significant bits of function code Hex "95" for A–(B&C) yields the function code Hex "9A" that is A–(B|~C). The B&C operation zero's bits in B where C is "0" and the operation B|~C forces bits in B to "1" where C is "0". This achieves the opposite masking function with respect to C. As will be explained below selectively invert-

ing the 4 least significant bits of the function code based on a sign bit performs sign extension before addition or subtraction.

All the 256 arithmetic functions available employing arithmetic logic unit **230** can be expressed as:

$$S=A \,\&\, F1(B,C)+F2(B,C)$$

where: S is the arithmetic logic unit resultant; and F1(B,C) and F2(B,C) can be any of the 16 possible Boolean functions of B and C shown below in Table 24.

TABLE 24

F1 Code	F2 Code	Subfunction	Common Use
00	00	0	Zeros term
AA	FF	all 1's = −1	Sets term to all 1's
88	CC	B	B
22	33	−B−1	Negate B
A0	F0	C	C
0A	0F	−C−1	Negate C
80	C0	B&C	Force bits in B to 0 where C is 0
2A	3F	−(B&C)−1	Force bits in B to 0 where C is 0 and negate
A8	FC	B|C	Force bits in B to 1 where C is 1
02	03	−(B|C)−1	Force bits in B to 1 where C is 1 and negate
08	0C	B&~C	Force bits in B to 0 where C is 1
A2	F3	−(B&~C)−1	Force bits in B to 0 where C is 1 and negate
8A	CF	B|~C	Force bits in B to 1 where C is 0
20	30	−(B|~C)−1	Force bits in B to 1 where C is 0 and negate
28	3C	(B&~C)|((−B−1)&C)	Choose B if C=all 0's and −B if C=all 1's
82	C3	(B&C)|((−B−1)&~C)	Choose B is C=all 1's and −B if C=all 0's

FIG. **25** illustrates this view of arithmetic logic unit **230** in block diagram form. Arithmetic unit **491** forms the addition of the equation. Arithmetic unit **491** receives a carry input for bit **0** from bit **0** carry-in generator. The AND gate **492** forms A AND F1(B,C). Logic unit **493** forms the subfunction F1(B,C) from the function signals as listed in Table 24. Logic unit **494** forms the subfunction F2(B,C) from the function signals as listed in Table 24. This illustration of arithmetic logic unit **230** shows that during mixed Boolean and arithmetic operations the Boolean functions are performed before the arithmetic functions. A set of the bit circuits **400** illustrated in FIGS. **19**, **20** and **21** together with the function generator illustrated in FIG. **22**, the function modifier illustrated in FIG. **23** and the bit **0** carry-in generator illustrated in FIG. **24** form the preferred embodiment of the arithmetic logic unit **230** illustrated in FIG. **25**. Those skilled in the art would recognize that there are many other feasible ways to implement arithmetic logic unit **230** illustrated in FIG. **25**.

As clearly illustrated in FIG. **25**, the subfunctions F1(B,C) and F2(B,C) are independent and may be different subfunctions for a single operation of arithmetic logic unit **230**. The subfunction F2(B,C) includes both the negative of B and the negative of C. Thus either B or C may be subtracted from A by adding its negative. The codes for the subfunctions F1(B,C) and F2(B,C) enable derivation of the function code **F7–F0** for arithmetic logic unit **230** illustrated in FIGS. **20** and **21**. The function code **F7–F0** for arithmetic logic unit **230** is the exclusive OR of the codes for the corresponding subfunctions F1(B,C) and F2(B,C). Note the codes for the subfunctions have been selected to provide this result, thus these subfunctions do not have identical codes for the same operation.

The subfunctions of Table 24 are listed with the most generally useful ways of expression. There are other ways to represent or factor each function. For example by applying DeMorgan's Law, the function B|~C is equivalent to ~(~B&C). Because ~X=−X−1, ~(~B&C) is equivalent −(~B&C)−1 and B|~C is equivalent to B|(−C−1). Note that the negative forms in Table 24 each have a trailing "−1" term. As explained above negative numbers are two's complements. These are equivalent to the bit wise logical inverse, which forms the 1's complement, minus 1. A carry-in of "1" may be injected into the least significant bit to cancel out the −1 and form the two's complement. In the most useful functions with a negative subfunction, only the F2(B,C) subfunction produces a negative.

Often it will be convenient to think of the Boolean subfunctions in Table 24 as performing a masking operation. As noted in Table 24, the subfunction B&C can be interpreted as forcing the B input value to "0" where the corresponding bit in C is "0. The subfunction B|~C can be interpreted as forcing the B input value to "1" for every bit where the C input is "0" Because mask generator **234** and expand circuit **238** feed the C-port of arithmetic logic unit **230** via multiplexer **233**, in most cases the C-port will be used as a mask in subfunctions that involve both B and C terms. Table 24 has factored the expression of each subfunction in terms assuming that the input to the C-port is used as a mask. The equation above shows that the A-input cannot be negated in the arithmetic expression. Thus arithmetic logic unit **230** cannot subtract A from either B or C. On the other hand, either B or C can be subtracted from A because the subfunctions F1(B,C) and F2(B,C) support negation/inversion of B and C.

The subfunctions of Table 24 when substituted into the above equation produces all of the 256 possible arithmetic functions that arithmetic logic unit **230** can perform. Occasionally, some further reduction in the expression of the resultant yields an expression that is equivalent to the original and easier to understand. When reducing such expressions, several tips can be helpful. The base instruction set defaults to a carry-in of "0" for addition and a carry-in of "1" when the subfunction F2(B,C) has a negative B or C term as expressed in Table 24 . This carry-in injection has the effect of turning the one's complement (logical inversion) into a two's complement by effectively canceling the −1 on the right hand side of the expression of these subfunctions. The logic AND of A all "1's" equals A. Thus subfunction F1(B,C) may be set to yield all "1's" to get A on the left side of the equation. Note also that all "1's" equals two's complement signed binary number minus 1 (−1).

The examples below show how to use the equation and the subfunctions of Table 24 to derive any of the possible arithmetic logic unit functions and their corresponding function codes. The arithmetic function A+B can be expressed as A&(all "1's")+B. This requires F1(B,C)=all "1's" and F2(B,C)=B. The F1 code for all "1's" is Hex "AA" and the F2 code for B is Hex "CC" Bit-wise XORing Hex "AA" and Hex "CC" gives Hex "66" Table 23 shows that Hex "66" is function code for A+B.

The arithmetic function A−B can be expressed as A&(all "1's")+(−B−1)+1. This implies F1(B,C)=all "1's" (F1 code

Hex "AA") and F2(B,C)=−B−1 (F2 code Hex "33") with a carry-in injection of "1". Recall that a carry-in of "1" is the default for subfunctions F2 that include negation. Bit-wise XORing the F1 code of Hex "AA" and with the F2 code of Hex "33" gives Hex "99" Table 23 shows that Hex "99" is the function code for A−B assuming a carry-in of "1".

The arithmetic function A+C is derived similarly to A+B. Thus A+C=Aa(all "1's")+C. This can be derived by choosing F1(B,C)=all "1's" and F2(B,C)=C. The exclusive OR of the F1 code of Hex "AA" and the F2 code of Hex "F0" produces Hex "5A" the function code for A+C. Likewise, A−C is the same as A&(all "1's")+(−C−1)+1. The exclusive OR of the F1 code of Hex "AA" and the F2 code of Hex "OF" produces Hex "A5" the function code for A−C.

Three input arithmetic logic unit **230** provides a major benefit by providing masking and/or conditional functions between two of the inputs based on the third input. The data path of data unit **110** enables the C-port to be most useful as a mask using mask generator **234** or conditional control input using expand circuit **238**. Arithmetic logic unit **230** always performs Boolean functions before arithmetic functions in any mixed Boolean and arithmetic function. Thus a carry could ripple out of unmasked bits into one or more bits that were zeroed or set by a Boolean function. The following examples are useful in masking and conditional operations.

The function A+(B&C) can be expressed as A&(all "1's") +(B&C). Choosing F1(B,C)=all "1's" (F1 code of Hex "AA") and F2(B,C)=B&C (F2 code of Hex "C0") gives A+(B&C). The bit-wise exclusive OR of HEX "AA" and Hex "C0" gives the arithmetic logic unit function code of Hex "6A" listed in Table 23. This function can strip off bits from unsigned numbers. As shown below, this function can be combined with barrel rotator **235** and mask generator **234** in performing right shift and add operations. In this case C acts as a bit mask that zeros bits of B everywhere C is "0" Since mask generator **234** can generate a mask with right justified ones, selection of mask generator **234** via multiplexer Cmux **233** permits this function to zero some of the most significant bits in B before adding to A. Another use of this function is conditional addition of B to A. Selection of expand circuit **238** via multiplexer Cmux **233** enables control of whether B is added to A based upon bits in Mflags register **211**. During multiple arithmetic, bits in Mflags register **211** can control corresponding sections of arithmetic logic unit **230**.

The function A+(B|~C) can be expressed as A&(all '1' S")+(B|~C). Choosing F1(B,C)=all "1's" (F1 code of Hex "AA") and F2(B,C)=B|~C (F2 code of "CF") yields this expression. The bit-wise exclusive OR of Hex "AA" and Hex "C0" obtains the function code of Hex "65" as listed in Table 23.

The function A−(B&C) can be expressed as A&(all "1's") +(−(B&−C)−1)+1. Choosing F1(B,C)=all "1's" (F1 code Hex "AA") and F2(B,C)=−(B&C)−1 (F2 code Hex "3F") with a carry-in injection of "1" yields this expression. The bit-wise exclusive OR of Hex "AA" and Hex "3F" yields the function code Hex "95" as listed in Table 23. This function can strip off or mask bits in the B input by the C input before subtracting from A.

There are 16 possible functions where the subfunction F1(B,C)=0. These functions are commonly used with other hardware to perform negation, absolute value, bit masking, and/or sign extension of the B-input by the C-input. When subfunction F1(B,C)=0 then the arithmetic logic unit function is given by subfunction F2(B,C).

The function −(B&C) may be expressed as (A&"0")+(−(B&C)). This expression can be formed by choosing F1(B,C)=0 (F1 code Hex "00") and F2(B,C)=−(B&C)−1 (F2 code Hex "3F") with a carry-in injection of "1". The exclusive OR of Hex "00" and Hex "3F" yields the function code Hex "3F" as shown in Table 23. This function masks bits in B by a mask C and then negates the quantity. This function can be used as part of a shift right and negate operation.

Several functions support masking both terms of the sum in the equation above in a useful manner. The function (A&C)+(B&C) can be achieved by choosing F1(B,C)=C (F1 code Hex "A0") and F2(B,C)=B&C (F2 code Hex "C0"). The exclusive OR of Hex "A0" and Hex "F0" yields the function code Hex "60" as shown in Table 23. This function will effectively zero the corresponding bits of the A and B inputs where C is "0" before adding. It should be noted that the Boolean function is applied before the addition and that one or more carries can ripple into the bits that have been zeroed. When using multiple arithmetic such carries do not cross the boundaries between the split sections of arithmetic logic unit **230**. A common use for this function is to sum multiple smaller quantities held in one register. The B-port receives a rotated version of the number going to the A-port and the C-port provides a mask for the bits that overlap. Four 8 bit numbers can be summed into two 16 bit numbers or two 16 bit numbers summed into one 32 bit number in a single instruction.

The similar function (A&C)−(B&C) is achieved by choosing F1(B,C)=C (F1 code Hex A0") and F2(B,C)=−(B&C)−1 and injecting a carry-in of "1". The exclusive OR of Hex "A0" and Hex "3F" yields the function code Hex "9F" as shown in Table 23. This function can produce negative sums with the C-port value acting as a mask of the A and B inputs.

The function (A&B)+B is achieved by choosing F1(B,C) =C (F1 code Hex "A0") and F2(B,C)=B (F2 code Hex "CC"). The exclusive OR of Hex "A0" and Hex "CC" yields the function code Hex "6C". This function can conditionally double B based on whether A is all "1's" or all "0's".

FIG. **26** illustrates in block diagram form an alternative embodiment of arithmetic logic unit **230**. The arithmetic logic unit **230** of FIG. **26** forms the equation:

$$S=F3(A,B,C)+F4(A,B,C)$$

where: S is the arithmetic logic unit resultant; and F3(A,B, C) and F4(A,B,C) can be any of the **256** possible Boolean functions of A, B and C. Adder **495** forms the addition of this equation and includes an input for a least significant bit carry input from bit **0** carry-in generator **246**. Boolean function generator **496** forms the function F3(A,B,C) as controlled by input function signals. Boolean function generator **497** similarly forms the function F4(A,B,C) as controlled by input function signals. Note that Boolean function generators **496** and **497** independently form selected Boolean combinations of A, B and C from a set of the 256 possible Boolean combinations of three inputs. Note that it is clear from this construction that arithmetic logic unit **230** forms the Boolean combinations before forming the arithmetic combination. The circuit in FIG. **21** can be modified to achieve this result. The generate/kill function illustrated in FIG. **21** employs a part of the logic tree used in the propagate function. This consists of pass gates **451**, **452**, **453**, **454**, **461** and **462**. Providing a separate logic tree for this function that duplicates pass gates **451**, **452**, **453**, **454**, **461** and **462** and eliminating the NOT A gate **475** results in a structure embodying FIG. **26**. Note in this construction one of the generate or kill terms may occur simultaneously with the propagate term. This construction provides even greater flexibility than that illustrated in FIG. **25**.

The three input arithmetic logic unit **230**, the auxiliary data path hardware and knowledge of the binary number system can be used to form many useful elementary functions. The instruction set of the digital image/graphics processors makes more of the hardware accessible to the programmer than typical in microprocessors. Making hardware more accessible to the programmer exposes some aspects of architecture that are hidden on most other processors. This instruction set supports forming custom operations using the elemental functions as building blocks. This makes greater functionality accessible to the programmer beyond the hardware functions commonly found within other processors, the digital image/graphics processors have hardware functions that can be very useful for image, graphics, and other processing. This combination of hardware capability and flexibility allows programmers to perform in one instruction what could require many instructions on most other architectures. The following describes some key elemental functions and how two or more of them can be combined to produce a more complex operation.

The previous sections described the individual workings of each functional block of data unit **110**. This section will discuss how these functions can be used in combination to perform more complex operations. Barrel rotator **235**, mask generator **239** and 3-input arithmetic logic unit **230** can work together to perform shift left, unsigned shift right, and signed shift right either alone or combination with addition or subtraction in a single arithmetic logic unit instruction cycle. An assembler produces program code for digital image/graphics processors **71**, **72**, **73** and **74**. This assemblier preferably supports the symbols ">>u" for unsigned (logical) right shift, ">>" or ">>s" for arithmetic (signed) right shift, and "<<" for a left shift. These shift notations are in effect macro functions that select the appropriate explicit functions in terms of rotates, mask generation, and arithmetic logic unit function. The assemblier also preferably supports explicitly specifying barrel rotation ("\\"), mask generation ("%" and "%!"), and the arithmetic logic unit function. The explicit notation will generally be used only when specifying a custom function not expressible by the shift notation.

Data unit **110** performs left shift operations in a single arithmetic logic unit cycle. Such a left shift operation includes barrel rotator via barrel rotator **235** by the number of bits of the left shift. As noted above during such rotation, bits that rotate out the left wrap around into the right and thus need to be stripped off to perform a left shift. The rotated output is sent to the B-port of arithmetic logic unit **230**. Mask generator **239** receives the shift amount and forms a mask with a number of right justified ones equal to the shift amount. Note that the same shift amount supplies the rotate control input of barrel rotator **235** from second input bus **202** via multiplexer Smux **231** and mask generator **239** from second input bus **202** via multiplexer Mmux **234**. Mask generator **239** supplies the C-port of arithmetic logic unit **230**. Arithmetic logic unit **230** combines the rotated output with the mask with the Boolean function B&~C. Left shifts are expressed in the assemblier below:

```
Left_Shift=Input<<Shift_Amount
```

This operation is equivalent to the explicit notation:

```
Left_Shift=(Input\\Shift_Amount)&~% Shift_Amount
```

The following example shows of a left shift of Hex "53FFFFA7" by 4 bits. While shown in several steps, data unit **110** performs this in a single pass arithmetic logic unit cycle The original number in binary notation is:

0101 0011 1111 1111 1111 1111 1010 0111

Rotation by 4 places in barrel rotator **235** yields:

0011 1111 1111 1111 1111 1010 0111 0101

Mask generator **239** forms the following mask:

0000 0000 0000 0000 0000 0000 0000 0000 1111

Arithmetic logic unit **230** forms the logical combination B&~C. This masks bits in the rotated amount causing them to be "0" and retains the other bits. This yields the left shift result:

0011 1111 1111 1111 1111 1111 1010 0111 0000

The left shift of the above example results in an arithmetic overflow, because some bits have "overflowed". During a shift left, arithmetic overflow occurs for unsigned numbers if any bits are shifted out. Arithmetic overflow may also occur for signed numbers if the resulting sign bit differs from the original sign bit. Arithmetic logic unit **230** of this invention does not automatically detect arithmetic overflow on left shifts. Left shift overflow can be detected by subtracting the left-most-bit-change amount of the original number generated by LMO/RMO/LMBC/RMBC circuit **237** from the left shift amount. If the difference is less than or equal to zero, then no bits will overflow during the shift. If the difference is greater than zero, this difference is the number of bits that overflow.

The assemblier further controls data unit **110** to perform left shift and add operations and left shift and subtract operations. The assemblier translates the A+(B<<n) function into control of barrel rotator shifter **235**, mask generator **239**, and arithmetic logic unit **230** to performed the desired operation. A shift left and add operation works identically to the above example of a simple shift except for the operation of arithmetic logic unit **230**. instead of performing the logical function B&~C as in a simple shift, the arithmetic logic unit performs the mixed arithmetic and logical function A+(B&~C). A left shift and add operation is expressed in the assemblier notation as:

```
LShift_Add=input1+input2<<Shift_Amount
```

This operation is equivalent to:

```
LShift_Add=Input1+[(Input2\\Shift_Amount)&~% Shift_
    Amount]
```

The following example shows a left shift of Hex "53FFFFA7" by 4 bits followed by addition of Hex "000000AA". Note that all these steps require only a single arithmetic logic unit cycle. The original Input2 in binary notation is:

0101 0011 1111 1111 1111 1111 1010 0110 1111

Rotation by 4 places in barrel rotator **235** yields:

0011 1111 1111 1111 1111 1010 0111 0101

Mask generator **239** forms the mask:

0000 0000 0000 0000 0000 0000 0000 1111

Arithmetic logic unit **230** forms the logical combination B&~C producing a left shift result:

0011 1111 1111 1111 1111 1010 0111 0000

The other operand Input1 in binary notation is:

Finally the sum is:

0011 1111 1111 1111 1111 1011 0001 1010

Note that arithmetic logic unit **230** forms the logical combination and the arithmetic combination in a single cycle and that the left shift result shown above is not available as an intermediate result. Note also that the sum may overflow even if the left shift does not produce an overflow. Overflow of the sum is detected by generation of a carry-out from the most significant bit of arithmetic logic unit **230**. This condition is detected and stored in the "V" bit of status register **210**.

The shift left and subtract operation also breaks down into a set of functions performed by barrel rotator **235**, mask generator **237**, and arithmetic logic unit **230** in a single arithmetic logic unit cycle. The left shift and subtract operation differs from the previously described left shift operation and left shift and add operation only in the function of arithmetic logic unit **230**. During left shift and subtract arithmetic logic unit **230** performs the mixed arithmetic and logical function A+(B|~C)+1. Arithmetic logic unit **230** performs the "+1" operation by injection of a "1" into the carry input of the least significant bit. This injection of a carry-in takes place at bit **0** carry-in generator **246**. Most subtraction operations with this invention take place using such a carry-in of "1" to the least significant bit. The assemblier notation expresses left shift and subtract operation as follows:

LShift_Sub=Input1 -input2<<Shift_Amount

This operation is equivalent to:

LShift_Sub=Input1-[(Input2\\Shift_Amount)&~% Shift_Amount]+1

The following example shows a left shift of Hex "53FFFFA7" by 4 bits followed by subtraction of Hex "000000AA". Note that all these steps require only a single arithmetic logic unit cycle. The original Input2 in binary notation is:

0101 0011 1111 1111 1111 1111 1010 0111

Rotation by 4 places in barrel rotator **235** yields:

Mask generator **239** forms the mask:

0000 0000 0000 0000 0000 0000 0000 1111

The result of the logical combination ~B|C is as follows:

1100 0000 0000 0000 0000 0101 1000 1111

The other operand Input1 in binary notation is:

0000 0000 0000 0000 0000 0000 1010 1010

The sum A+(~B|C) is:

1100 0000 0000 0000 0000 0110 0011 1001

Finally the addition of the "1" injected into the least significant bit carry-in yields:

1100 0000 0000 0000 0000 0110 0011 1010

Note that arithmetic logic unit **230** forms the logical combination and the arithmetic combination in a single cycle and that neither the left shift result nor the partial sum shown above are available as intermediate results.

The assemblier of the preferred embodiment can control data unit **110** to perform an unsigned right shift with zeros shifted in from the left in a single arithmetic logic unit cycle. Since barrel rotator **235** performs a left rotate, at net right rotate may be formed with a rotate amount of 32−n, where n is the number of bits to rotate right. Note, only the 5 least significant bits of the data on second input bus **202** are used by barrel rotator **235** and mask generator **239**. Therefore the amounts 32 and 0 are equivalent in terms of controlling the shift operation. The assembler will immediate right shift amount. The assemblier of the preferred embodiment requires the programmer form the quantity 32−n on register based shifts.

Once the accommodation for right rotation is made, the unsigned shift right works the same as the shift left except that arithmetic logic unit **230** performs a different function. This operation includes rotation by the quantity 32−n via barrel rotator **235**. The result of this net rotate right will to have bits wrapped around from the least significant to the most significant part of the word. The same quantity (32−n) controls mask generator **239**, which will generate 32−1 n right justified ones. Mask generator **239** is controlled with the "!" option so that a shift amount of zero produces a mask of all "1's". In this case no bits are to be stripped off. Arithmetic logic unit **230** then forms a Boolean combination of the outputs of barrel rotator **235** and mask generator **239**.

An example of an unsigned right shift operation is shown below. The assemblier notation for an unsigned right shift is:

Unsigned_Right_Shift=Input>>u(32−Shift_Amount)

The equivalent operation explicitly showing the functions performed is:

Unsigned_Right_Shift=(input\\(32Shift_Amount))& % ! (32−Shift_Amount)

Note in the equation above the mask operator "% !" specifies that if the shift amount is zero, an all "1" mask will be generated. The example below shows the unsigned shifting the number Hex "53FFFFA7" right by 4 bit positions. The original number in binary form is:

0101 0011 1111 1111 1111 1111 1010 0111

This number when left rotated by 32−4=28 places becomes:

0111 0101 0011 1111 1111 1111 1111 1010

Mask generator **239** forms a mask from the input 32−4=28, which is:

0000 1110 1111 1111 1111 1111 1111 1111

Lastly arithmetic logic unit **230** forms the Boolean combination B&C yielding the result:

0000 0101 0011 1111 1111 1111 1111 1010

Data unit **110** may perform either unsigned right shift and add or unsigned right shift and subtract operations. In the preferred embodiment the assemblier translates the notation A+B>>u(n) into an instruction that controls barrel rotator **235**, mask generator **239** and arithmetic logic unit **230** to performed an unsigned right shift and add operation. The

unsigned shift right and add works identically to the previous example of a simple unsigned shift right except that arithmetic logic unit **230** performs the function A+(B&C). In the preferred embodiment the assemblier translates the notation A–B>>u(n) into an instruction that controls barrel rotator **235**, mask generator **239** and arithmetic logic unit **230** to performed an unsigned right shift and subtract operation The unsigned shift right and subtract works similarly to the previous example of a simple unsigned shift right except that arithmetic logic unit **230** performs the function A–(~B|C)+1. As with left shift and subtract the "+1" operation involves injection of a "1" carry-in into the least significant bit via bit **0** carry-in generator **246**.

The assemblier of the preferred embodiment can control data unit **110** to perform a signed right shift with sign bits shifted in from the left in a single arithmetic logic unit cycle. The assembler will automatically make the 32–n computation for such shifts with an immediate right shift amount. Data unit **110** includes hardware that detects that state of the most significant bit, called the sign bit, of the input into barrel rotator **235**. This sign bit may control the 4 least significant bits of the function code. When using this hardware, the 4 least significant bits of the function code are inverted if the sign bit is "0". Signed right shift operations use this sign detection hardware to control the function arithmetic logic unit **230** performs based on the sign of the input to barrel rotator **235**. This operation can be explained using the following elemental functions. Barrel rotator **235** performs a net rotate right by rotating left by 32 minus the number of bits of the desired signed right shift (32–n). This shift amount (32–n) is supplied to mask generator **237**, which will thus generate 32–n right justified "1's". The "1's" of this mask will select the desired bits of the number that is right shifted. The "0's" of this mask will generate sign bits equal to the of the most significant bit input to barrel rotator **235**. Arithmetic logic unit **230** then combines the rotated number from barrel rotator **235** and the mask from mask generator **237**. The Boolean function performed by arithmetic logic unit **230** depends upon the sign bit at the input to barrel rotator **235**. If this sign bit is "0", then arithmetic logic unit **230** receives function signals to perform B&C. While selecting the rotated number unchanged, this forces "0" any bits that are "0" in the mask. Thus the most significant bits of the result are "0" indicating the same sign as the input to barrel rotator **235**. If the sign bit is "1", then arithmetic logic unit **230** received function signal to perform B|∞C. This function selects the rotated amount unchanged while forcing to "1" any bits that are "0" in the mask. The change in function code involves inverting the 4 least significant bits if the detected sign bit is "0". Thus the most significant bits of the result are "1", the same sign indication as the input to barrel rotator **235**.

Two examples of the unsigned right shift operation are shown below. Signed right shift is the default assemblier notation for right shifts. The two permitted assemblier notations for a signed right shift are:

Signed_Right_Shift=Input>>s(32–Shift_Amount)

Signed_Right_Shift=input>>(32–Shift_Amount)

Because this operation uses the sign detection hardware, there is no explicit way in the notation of the preferred embodiment of the assemblier to specify this operation in terms of rotation and masking. In the preferred embodiment the sign of the input to barrel rotator **235** controls inversion of the function signals **F3–F0**. The first example shows a 4 place signed right shift of the negative number Hex "ECFFFFA7". The original number in binary notation is:

1110 1100 1111 1111 1111 1111 1010 0111

Left rotation by 28 (32–4) places yields:

0111 1110 1100 1111 1111 1111 1111 1010

Mask generator **237** forms this mask:

0000 1111 1111 1111 1111 1111 1111 1111

Because the most significant bit of the input to barrel rotator **235** is "1", arithmetic logic unit **230** forms the Boolean combination of B|~C. This yields the result:

1111 1110 1100 1111 1111 1111 1111 1010

In this example "1's" are shifted into the most significant bits of the shifted result, matching the sign bit of the original number. The second example shows a 4 place signed right shift of the positive number Hex "5CFFFFA7". The original number in binary notation is:

0101 1100 1111 1111 1111 1111 1010 0111

Left rotation by 28 (32–4) places yields:

0111 0101 1100 1111 1111 1111 1111 1010

Mask generator **237** forms this mask:

0000 1111 1111 1111 1111 1111 1111 1111

Because the most significant bit of the input to barrel rotator **235** is "0", arithmetic logic unit **230** forms the Boolean combination of B&C by inversion of the four least significant bits of the function code. This yields the result:

Note that upon this right shift "0's" are shifted in the most significant bits, matching the sign bit of the original number.

Data unit **110** may perform either signed right shift and add or signed right shift and subtract operations. In the preferred embodiment the assemblier translates the notations A+B>>(n) or A+B>>s(n) into an instruction that controls barrel rotator **235**, mask generator **239** and arithmetic logic unit **230** to perform a signed right shift and add operation. The signed shift right and add works identically to the previous example of the signed shift right except for the function performed by arithmetic logic unit **230**. In the signed right shift and add operation arithmetic logic unit **230** performs the function A+(B&C) if the sign bit of the input to barrel rotator **235** is "0" If this sign bit is "1", then arithmetic logic unit **230** performs the function A+(B|~C). In the preferred embodiment the assemblier translates the notations A–B>>s(n) or A–B>>(n) into an instruction that controls barrel rotator **235**, mask generator **239** and arithmetic logic unit **230** to perform a signed right shift and subtract operation. The signed shift right and subtract operation works similarly to the previous example of a simple signed shift right except for the function of arithmetic logic unit **230**. When the sign bit is "1", arithmetic logic unit **230** performs the function A–(B&C)+1. When the sign bit is "0", arithmetic logic unit **230** performs the alternate function A–(B|~C)+1. As in the case of left shift and subtract the "+1" operation involves injection of a "1" carry-in into the least significant bit via bit **0** carry-in generator **246**.

Barrel rotator **235**, mask generator **237** and arithmetic logic unit **230** can perform field extraction in a single cycle. A field extraction takes a field of bits in a word starting at

any arbitrary bit position, strips off the bits outside the field and right justifies the field. Such a field extraction is performed by rotating the word left the number of bits necessary to right justify the field and masking the result of the rotation by the number of bits in the size of the field. Unlike the cases for shifting, the rotation amount, which is based on the bit position, and the mask input, which is based on the field size, are not necessarily the same amount. The assemblier of the preferred embodiment employs the following notation for field extraction:

Field_Extract=(Value\\(32-starting_bit))& % ! Field size

The "% !" operator causes mask generator **237** to form a mask having a number of right justified "1's" equal to the field size, except for an input of zero. In that case all bits of the generated mask are "1" so that no bits are masked by the logical AND operation. This rotation and masking may produce wrapped around bits if the field size is greater than the starting bit position. These parameters specify an anomalous case in which the specified field extends beyond the end of the original word. Data unit **110** provides no hardware check to for this case. It is the responsibility of the programmer to prevent this result. The example below demonstrates field extraction of a 4-bit field starting at bit **24**, which is the eight bit from the left, of the number Hex "5CFFFFA7". The number in binary form is:

0101 1100 1111 1111 1111 1111 1010 0111

The number must be rotated left by 32–24 or 8 bits to right justify the field. The output from barrel rotator **235** is:

1111 1111 1111 1111 1010 0111 0101 1100

Mask generator **237** forms the following mask from the field size of 4 bits:

0000 0000 0000 0000 0000 0000 0000 1111

Lastly, arithmetic logic unit **230** forms the Boolean combination B&C. This produces the extracted field as follows:

0000 0000 0000 0000 0000 0000 0000 1100

Mflags register **211** is useful in a variety of image and graphics processing operations. These operations fall into two classes. The first class of Mflags operations require a single pass through arithmetic logic unit **230**. A number is loaded into Mflags register **211** and controls the operation of arithmetic logic unit **230** via expand circuit **238**, multiplexer Cmux **233** and the C-port of arithmetic logic unit **230**. Color expansion is an example of these single pass operations. The second class of Mflags operations require two passes through arithmetic logic unit **230**. During a first pass certain bits are set within Mflags register **211** based upon the carry of zero results of arithmetic logic unit **230**. During a second pass the contents of Mflags register **211** control the operation of arithmetic logic unit **230** via expand circuit **238**, multiplexer Cmux **233** and the C-port of arithmetic logic unit **230**. Such two pass Mflags operations are especially useful when using multiple arithmetic. Numerous match and compare, transparency, minimum, maximum and saturation operations fall into this second class.

A basic graphics operation is the conversion of one bit per pixel shape descriptors into pixel size quantities. This is often called color expansion. In order to conserve memory space the shape of bit mapped text fonts are often stored as shapes of one bit per pixel. These shapes are then "expanded" into the desired color(s) when drawn into the display memory. Generally "1's" in the shape descriptor select a "one color" and "0's" in the shape descriptor select a "zero color". A commonly used alternative has "0's" in the shape descriptor serving as a place saver or transparent pixel.

The following example converts 4 bits of such shape descriptor data into 8 bit pixels. In this example the data size of the multiple arithmetic operation is 8 bits. Thus arithmetic logic unit **230** operates in 4 independent 8 bit sections. The four bits of descriptor data "0110" are loaded into Mflags register **211**:

XXXXXXXX XXXXXXXX XXXXXXXX XXXX0110

The bits listed as "X" are don't care bits that are not involved in the color expansion operation. Expand circuit **238** expands these four bits in Mflags register **211** into blocks of 8 bit "1's" and "0's" as follows:

00000000 11111111 11111111 00000000

The one color is supplied to the A-port of arithmetic logic unit **230** repeated for each of the 4 pixels within the 32 bit data word:

11110000 11110000 11110000 11110000

The zero color is supplied to the B-port of arithmetic logic unit **230**, also repeated for each of the 4 pixels:

10101010 10101010 10101010 10101010

Arithmetic logic unit **230** forms the Boolean combination (A&C)|(B&~C) which yields:

10101010 11110000 11110000 10101010

Color expansion is commonly used with a PixBlt algorithm. To perform a complete PixBlt, the data will have to be rotated and merged with prior data to align the bits in the data to be expanded with the pixel alignment of the destination words. Barrel rotator **235** and arithmetic logic unit **230** can align words into Mflags register **211**. This example assumed that the shape descriptor data was properly aligned to keep the example simple. Note also that Mflags register **211** has its own rotation capability upon setting bits and using bits. Thus a 32 bit word can be loaded into Mflags register **211** and the above instruction repeated 8 times to generate 32 expanded pixels.

Simple color expansion as in the above example forces the result to be one of two solid colors. Often, particularly with kerned text letters whose rectangular boxes can overlap, it is desirable to expand "1's" in the shape descriptor to the one color but have "0's" serve as place saver or transparent pixels. The destination pixel value is unchanged when moving such a transparent color. Data unit **110** can perform a transparent color expand by simply using a register containing the original contents of the destination as the zero value input. An example of this appears below. Arithmetic logic unit **230** performs the same function as the previous color expansion example. The only difference is the original destination becomes one of the inputs to arithmetic logic unit **230**. The four bits of descriptor data "0110" are loaded into Mflags register **211**:

XXXXXXXX XXXXXXXX XXXXXXXX XXXX0110

Expand circuit **238** expands these four bits in Mflags register **211** into blocks of 8 bit "1's" and "0's" as follows:

00000000 11111111 11111111 00000000

The one color is supplied to the A-port of arithmetic logic unit **230** repeated for each of the 4 pixels within the 32 bit data word:

1110000 11110000 11110000 11110000

The original destination data is supplied to the B-port of arithmetic logic unit **230**, original destination data including 4 pixels:

11001100 10101010 11101110 11111111

Arithmetic logic unit **230** again forms the Boolean combination (A&C)|(B&~C) which yields:

11001100 11110000 11110000 11111111

Note that the result includes the one color for pixels corresponding to a "1" in Mflags register **211** and the original pixel value for pixels corresponding to a "0" in Mflags register **211**.

Data unit **110** can generate a **1** bit per pixel mask based on an exact match of a series of 8 bit quantities to a fixed compare value. This is shown in the example below. The compare value is repeated four times within the 32 bit word. Arithmetic logic unit **230** subtracts the repeated compare value from a data word having four of the 8 bit quantities. During this subtraction, arithmetic logic unit **230** is split into 4 sections of 8 bits each. The zero detectors **321**, **322**, **323** and **324** illustrated in FIG. **7** supply are data to be stored in Mflags register **211**. This example includes two instructions in a row to demonstrate accumulating by rotating Mflags register **211**. initially Mflags register **211** stores don't care data:

XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX

The first quantity for comparison is:

00000011 00001111 00000001 00000011

The compare value is "00000011". This is repeated four times in the 32 bit word as:

00000011 00000011 00000011

Arithmetic logic unit **230** subtracts the compare value from the first quantity. The resulting difference is:

00000000 00001100 11111110 00000000

This forms the following zero compares "1001" that are stored in Mflags register **211**. In this example Mflags register **211** is pre-cleared before storing the zero results. Thus Mflags register **211** is:

00000000 00000000 00000000 00001001

The second quantity for comparison is:

00000111 11111100 00000011 00000000

The result of a second subtraction of the same compare value is:

00000100 11111001 00000000 11111101

This forms the new zero compares "0010" that are stored in Mflags register **211** following rotation of four places:

00000000 00000000 00000000 10010010

Additional compares may be made in the same fashion until Mflags register **211** stores 32 bits. Then the contents of Mflags register **211** may be moved to another register or written to memory.

Threshold detection involves comparing pixel values to a fixed threshold. Threshold detection sets a 1 bit value for each pixel which signifies the pixel value was greater than or less than the fixed threshold. Depending on the particular application, the equal to case is grouped with either the greater than case or the less than case. Data unit **110** may be programmed to from the comparison result in a single arithmetic logic unit cycle. Arithmetic logic unit **230** forms the difference between the quantity to be tested and the fixed threshold. The carry-outs from each section of arithmetic logic unit **230** are saved in Mflags register **211**. If the quantity to be tested I has the fixed threshold T subtracted from it, a carry out will occur only if I is greater than or equal to T. As stated above, arithmetic logic unit **230** performs subtraction by two's complement addition and under these circumstances a carry-out indicates a not-borrow. Below is an example of this process for four 8 bit quantities in which the threshold value is "00000111". Let four 8 bit quantities I to be tested be:

00001100 00000001 00000110 00000111

The threshold value T repeated four times within the 32 bit word is:

00000111 00000111 00000111 00000111

The difference is:

00000101 11111010 11111111 00000000

which produces the following carry-outs "1001". This results in a Mflags register **211** of:

XXXXXXXX XXXXXXXX XXXXXXXX XXXX1001

As in the case of match detection, this single instruction can be repeated for new data with Mflags resister rotation until 32 bits are formed.

When adding two unsigned numbers, a carry-out indicates that the result is greater than can be expressed in the number of bits of the result. This carry-out represents the most significant bit of precision of the result. Thus saving the carry-outs in Mflags register **211** can be used to maintain precision. These carry-out bits may be saved for later addition to maintain precision. Particularly when used with multiple arithmetic, limiting the precision to fewer bits often enables the same process to be performed in fewer arithmetic logic unit cycles.

Mflags operations of the second type employ both setting bits within Mflags register **211** and employing bits stored in Mflags register **211** to control the operation of arithmetic logic unit **230**. Multiple arithmetic can be used it in combination with expands of Mflags register **211** to perform multiple parallel byte or half-word operations. Additionally, the setting of bits in Mflags register **211** and expanding Mflags register **211** to arithmetic logic unit **230** are inverse space conversions that can be used in a multitude of different ways.

The example below shows a combination of an 8 bit multiple arithmetic instruction followed by an instruction using expansion to perform a transparency function. Transparency is commonly used when performing rectangular PixBlts of shapes that are not rectangular. The transparent pixels are used as place saver pixels that will not affect the

destination and thus are transparent so the original destination shows through. With transparency, only the pixels in the source that are not equal to the transparent code are replaced in the destination. In a first instruction the transparent color code is subtracted from the source and Mflags register **211** is set based on equal zero. If a given 8 bit quantity matches the transparent code, a corresponding "1" will be set in Mflags register **211**. The second instruction uses expansion circuit **238** to expand Mflags register **211** to control selection on a pixel by pixel basis of the source or destination. Arithmetic logic unit **230** performs the function (A&C)|(B&~C) to make this selection. While this Boolean function is performed bit by bit, Mflags register **211** has been expanded to the pixel size of 8 and thus it selects between pixels. The pixel source is:

00000011 01110011 00000011 00000001

The transparent code TC is "00000011". Repeated 4 times to fill the 32 bit word this becomes:

00000011 00000011 00000011 00000011

The difference SRC-TC is:

00000000 01110000 00000000 11111110

which produces the zero detection bits "1010". Thus Mflags register **211** stores:

XXXXXXXX XXXXXXXX XXXXXXXX XXXX1010

In the second instruction, expand circuit **238** expands Mflags register **211** to:

11111111 00000000 11111111 00000000

The original destination DEST is:

11110001 00110011 01110111 11111111

The original source SRC forms a third input to arithmetic logic unit **230**. Arithmetic logic unit **230** then forms the Boolean combination (DEST&@MF)|(SRC&~@MF) which is:

11110001 00010011 01110111 00000001

Note that the resultant has the state of the source where the source was not transparent, otherwise it has the state of the destination. This is the transparency function.

Data unit **110** can perform maximum and minimum functions using Mflags register **211** and two arithmetic logic unit cycles. The maximum function takes the greater of two unsigned pixel values as the result. The minimum function takes the lesser of two unsigned pixel values as the result. In these operations the first instruction performs multiple subtractions, setting Mflags register **211** based on carry-outs. Thus for status setting arithmetic logic unit **230** forms OP1−OP2. This first instruction only sets Mflags register **211** and the resulting difference is discarded. When performing the maximum function the second instruction, arithmetic logic unit **230** performs the operation (OP1&@MF)|(OP2&~@MF). This forms the maximum of the individual pixels. Let the first operand OP1 be:

00000001 11111110 00000011 00000100

and the second operand OP2 be:

00000011 00000111 00000111 00000011

The difference OP1−OP2 is:

11111110 11110111 11111100 00000000

This produces carry-outs (not-borrows) "0101" setting Mflags register **211** as:

XXXXXXXX XXXXXXXX XXXXXXXX XXXX0101

In the second instruction the four least significant bits in Mflags register **211** are expanded via expand circuit **238** producing:

00000000 11111111 00000000 11111111

Arithmetic logic unit **230** performs the Boolean function (OP1&@MF)|(OP2&~@MF). This produces the result:

00000011 11111110 00000111 00000100

Note that each 8 bit section of the result has the state of the greater of the corresponding sections of OP1 and OP2. This is the maximum function. The minimum function operates similarly to the maximum function above except that in the second instruction arithmetic logic unit **230** performs the Boolean function (OP1&~@MF)|(OP2&@MF). This Boolean function selects the lesser quantity rather than greater quantity for each 8 bit section.

Data unit **110** may also perform an add-with-saturate function. The add-with-saturate function operates like a normal add unless an overflow occurs. In that event the add-with-saturate function clamps the result to all "1's". The add-with-saturate function is commonly used in graphics and image processing to keep small integer results from overflowing the highest number back to a low number. The example below shows forming the add-with-saturate function using multiple arithmetic on four 8 bit pixels in two instructions. First the addition takes place with the carry-outs stored in Mflags register **211**. A carry-out of "1" indicates an overflow, thus that sum should be set to all "1's", which is the saturated value. Then expand circuit **238** expands Mflags register **211** to control selection of the sum or the saturated value. The first operand OP1 is:

00000001 11111001 00000011 00111111

The second operand OP2 is:

11111111 00001011 00000111 01111111

Arithmetic logic unit **230** forms the sum OP1+OP2= RESULT resulting in:

00000000 00000100 00001010 10111110

with corresponding carry-outs of "1100". These are stored in Mflags register **211** as:

XXXXXXXX XXXXXXXX XXXXXXXX XXXX1100

In the second instruction expand circuit **238** expands the four least significant bits of Mflags register **211** to:

11111111 11111111 00000000 00000000

Arithmetic logic unit **230** performs the Boolean function RESULT|@MF forming:

11111111 11111111 00001010 10111110

Note the result of the second instruction equals the sum when the sum did not overflow and equals "11111111" when the sum overflowed.

Data unit **110** can similarly perform a subtract-with-saturate function. The subtract-with-saturate function operates like a normal subtract unless an underflow occurs. In that event the subtract-with-saturate function clamps the result to all "0's". The subtract-with-saturate function may also be commonly used in graphics and image processing. The data unit **110** performs the subtract-with-saturate function similarly to the add-with-saturate function shown above. First the subtraction takes place with the carry-outs stored in Mflags register **211**. A carry-out of "0" indicates a borrow and thus an underflow. In that event the difference should be set to all "0's", which is the saturated value. Then expand circuit **238** expands Mflags register **211** to control selection of the difference or the saturated value. During this second instruction arithmetic logic unit **230** performs the Boolean function RESULT&@MF. This forces the combination to "0" if the corresponding carry-out was "0", thereby saturating the difference at all "0's". On the other hand if the corresponding carry-out was "1", then the Boolean combination is the same as RESULT.

FIG. **27** illustrates in block diagram form the construction of address unit **120** of digital image/graphics processor **71** according to the preferred embodiment of this invention. The address unit **120** includes: a global address unit **610**; a local address unit **620**; a global/local multiplexer control register GLMUX **631**; a pair of zero detectors **631** and **632**; a multiplexer **641**; four control circuits **642**, **643**, **653**, **654**; a global temporary address register GTA **651**; a local temporary address register LTA **652**; a pair of address unit arithmetic buffers **655** and **656**; an instruction decode logic **660**; a global address port **121**; and a local address port **122**. As illustrated in FIG. **27**, global/local address multiplexer register GLMUX **630** is coupled to global port source data bus Gsrc **105** and to global port destination data bus Gdst **107**. Global/local address multiplexer register GLMUX **630** is in the register space of digital image/graphics processor **71** and may be written to or read from as any other register. Global temporary address register GTA **651** is connected to global port source data bus Gsrc **105** only. Though global temporary address register GTA is within the register space of digital image/graphics processor **71**, the preferred embodiment allows reads from but not writes to this register. An attempted write to global temporary address register GTA **651** is ignored. Note that local temporary address register LTA **652** is coupled to neither global port source data bus Gsrc **105** nor global port destination data bus Gdst **107**. This register is not within the register space of digital image/graphics processor **71** and cannot be accessed. As previously described each digital image/graphics processor **71**, **72**, **73** and **74** includes both a global data port and a local data port, which may operate simultaneously. Separate global and local address units allow generation of independent addresses for these independent data transfers. In addition, some combined addresses are permitted as will be further described below. Note that all the functions of address unit **120** are controlled by instruction decode logic **660**, which is responsive to the instruction currently in the address pipeline stage via opcode bus **133**. The details of these control lines are omitted from FIG. **27** for the sake of clarity. However, these control functions are within the capability of one skilled in the art from this description and the description of the instruction word formats in conjunction with FIG. **43**.

Tables 25 and 26 detail the permitted addresses generated by the respective global and local data ports of digital image/graphics processors **71**, **72**, **73** and **74**. Table 25 indicates the permitted data space addresses in hexadecimal according to the form Hex "0000????", where the range of the final four digits "????" is shown in Table 25.

TABLE 25

Global	Local Ports			
Ports	DIGP 71	DIGP 72	DIGP 73	DIGP 74
0000–3FFF 8000–8fff 9000–97FF A000–A7FF B000–B7FF	0000–0FFF 8000–87FF	1000–1FFF 9000–97FF	2000–2FFF A000–A7FF	3000–3FFF B000–B7FF

In a similar fashion, Table 26 indicates the permitted parameter space addresses in hexadecimal according to the form Hex "0100????", where the range of the final four digits "????" is shown in Table 26.

TABLE 26

Global	Local Ports			
Ports	DIGP 71	DIGP 72	DIGP 73	DIGP 74
0000–07FF 1000–17FF 2000–27FF 3000–37FF	0000–07FF	1000–17FF	2000–27FF	3000–37FF

Tables 25 and 26 show the limitations on addressing of the local data ports. As previously described, the global data ports (G) of the four digital image/graphics processors **71**, **72**, **73** and **74** may address any location within a data memory or a parameter memory. At the same time the local data ports (L) of each digital image/graphics processor **71**, **72**, **73** and **74** may only address the data and parameter memories corresponding to that digital image/graphics processor.

FIG. **28** illustrates in block diagram form the construction of global address unit **610**. In accordance with the preferred embodiment, local address unit **620** is constructed identically. Global address unit **610** includes: a set of address registers **611**; a set of index registers **612**; multiplexers **613** and **616**; an index scaler circuit **614**; and an addition/subtraction unit **615**. According to the preferred embodiment the addresses include 32 bits, therefore address registers **611** and index registers **612** store data words of 32 bits and addition/subtraction unit **615** operates on data words of 32 bits.

Table 27 lists the address register assignments. Note that address registers **611** are coupled to both global port source data bus Gsrc **105** and global port destination data bus Gdst **107**. These connections allow register loads from memory, register stores to memory, and register to register data transfer with other registers within that digital image/graphics processor, such as data registers **200** within data unit **110**. Various uses of these connections will be described below.

TABLE 27

Address Register	Register Assignment
A0	Lodal address unit
A1	Local address unit
A2	Local address unit
A3	Local address unit

TABLE 27-continued

Address Register	Register Assignment
A4	Local address unit
A5	reserved
A6	Global/Local address units shared stack pointer
A7	Local address unit read only, all zeros
A8	Global address unit
A9	Global address unit
A10	Global address unit
A11	Global address unit
A12	Global address unit
A13	reserved
A14	Global/Local address units shared stack pointer
A15	Global address unit read only, all zeros

Address registers **A0**, **A1**, **A2**, **A3** and **A4** are within local address unit **620** and are available for general use. Address register **A5** is not supported in the current embodiment, but its address is reserved for future expansion of the local address unit **620**. Address registers **A8**, **A9**, **A10**, **A11** and **A12** are within global address unit **620** and are available for general use. Address register **A13** is not supported in the current embodiment, but its address is reserved for future expansion of the global address unit **610**. Address registers **A6** and **A14** are embodied by a single register accessible by local address unit **620** at address **A6** and by address unit **610** at address **A14**. This combined register **A14/A6** will generally be used as a stack pointer. Note that stack operations are only allowed on aligned 32 bit word boundaries. Consequently the two least significant bits of combined register **A14/A6** are hardwired to "00". Writing to these two bits has no effect and they are always read as "00". Registers **A7** and **A15** are also embodied by the same hardware and both global address sun-unit **610** and local address unit **620** may use this combined register in the same instruction. Register **A7** is accessible to local address unit **620** and register **A15** is accessible to global address unit **610**. Combined register **A15/A7** is hardwired to all "0's". Writing to either of these two registers has no effect and they are always read as all "0's". In the preferred embodiment these two registers are embodied by the same hardware accessible at differing addresses.

Table 28 lists the index register assignments. Index registers **612** are coupled to both global port source data bus Gsrc **105** and global port destination data bus Gdst **107**. These connections permits register loads from memory, register stores to memory, and register to register data transfer with other registers within that digital image/graphics processor, such as data registers **200** within data unit **110**. Various uses of these connections will be described below.

TABLE 28

Index Register	Register Assignment
X0	Local address unit
X1	Local address unit
X2	Local address unit
X3	reserved
X4	reserved
X5	reserved

TABLE 28-continued

Index Register	Register Assignment
X6	reserved
X7	reserved
X8	Global address unit
X9	Global address unit
X10	Global address unit
X11	reserved
X12	reserved
X13	reserved
X14	reserved
X15	reserved

Index registers **X0**, **X1** and **X2** are within local address unit and are available for general use. index registers **X3**, **X4**, **X5**, **X6** and **X7** are not supported in the current embodiment, but their addresses are reserved for future expansion of the local address unit **620**. index registers **X8**, **X9** and **X10** are within global address unit **620** and are available for general use. Index registers **X11**, **X12**, **X13**, **X14** and **X15** are not supported in the current embodiment, but their addresses are reserved for future expansion of the global address unit **610**. Global address unit **610** generates a 32 bit address. Either an index stored in a specified index register within index registers **612** or an offset field from the instruction word is selected at multiplexer **613**. This selection is controlled by the instruction via instruction decode logic **660** (FIG. **27**). Multiplexer **613** also selects the size of the offset field again based on the instruction. As will be further discussed below, global address unit **610** may receive a 15 bit offset field or a 3 bit offset field. Whether the offset field is 15 bits or 3 bits, this value is zero extended to 32 bits before use.

Index scaler **614** optionally left shifts the data selected by multiplexer **613**. This optional left shift is selected by a scaled/unscaled input that corresponds to the function of the instruction. This left shift is 0, 1 or 2 bits depending on the indicated data size. As previously described the pixel data may be specified as 8 bits (byte), 16 bits (half word) or 32 bits (word). If scaling is selected, then the data is left shifted with zero filling 0 bit places for byte data, 1 bit place for half word data and 2 bit places for word data. Since no scaling ever occurs for byte data transfers, the instruction word bit specifying scaling is available for other purposes. In the preferred embodiment this instruction word bit is used as an additional offset bit. Thus if the data size is 8 bits, the instruction can supply a 16 bit offset index rather than a 15 bit offset index or a 4 bit offset index rather than a 3 bit offset index. This address index scaling feature permits addressing that is independent from the data size. This feature is useful in certain applications such as look up table operations.

Addition/subtraction unit **615** receives a base address from an address register selected by the instruction and the index. The instruction selects either addition of the index to the base address or subtraction of the index from the base address. The resultant forms one input to multiplexer **616**. The base address from the selected address register forms the other input to multiplexer **616**. Multiplexer **616** selects one of these addresses depending on whether the instruction specifies pre-indexing or post-indexing. If the instruction specifies pre-indexing, then the resultant of addition/subtraction unit **615** is selected by multiplexer **616** as the output address. If the instruction specified post-indexing, then the base address from address registers **611** is selected by multiplexer **616** as the output address.

The modified address may be written into the selected address register. In pre-indexing, then instruction selects

whether to write the modified address into the source address register within address registers **611**. In post-indexing, then the modified address is always written into the source address register within address registers **611**. In the preferred embodiment, the instruction word specifies one of 12 modes for each of the global address unit **610** and the local address unit **620**. These twelve modes include: pre-addition of an offset index without base address modification; pre-addition of an offset index with base address modification; post-addition of an offset index with base address modification; pre-subtraction of an offset index without base address modification; pre-subtraction of an offset index with base address modification; post-subtraction of an offset index with base address modification; pre-addition from an index register without base address modification; pre-addition from an index register with base address modification; post-addition from an index register with base address modification; pre-subtraction from an index register without base address modification; pre-subtraction from an index register with base address modification; and post-subtraction from an index register with base address modification.

Special read only zero value address registers A15/A7 permit special functions. Specification of the corresponding one of these registers as the source of the base address converts the index address into an absolute address. Specification of one of these zero value address registers may also load an offset index.

Hardware associated with each address unit permits specification of the base address of the data memories and the parameter memory corresponding to each digital image/graphics processor. This specification occurs employing two pseudo address registers. Specification of "PBA" as the address register produces the address of the parameter memory corresponding to that digital image/graphics processor. The parameter memory base address register of each digital image/graphics processor permanently stores the base address of the corresponding parameter memory. The parameter memory **25** corresponds to digital image/graphics processor **71**, parameter memory **30** corresponds to digital image/graphics processor **72**, parameter memory **35** corresponds to digital image/graphics processor **73**, and parameter memory **40** corresponds to digital image/graphics processor **74**. Specification of "DBA" as the address register produces the address of the base data memory corresponding to that digital image/graphics processor. The data memory **22** includes the lowest address corresponding to digital image/graphics processor **71**, data memory **27** includes the lowest address corresponding to digital image/graphics processor **72**, data memory **32** includes the lowest address corresponding to digital image/graphics processor **73** and data memory **37** includes the lowest address corresponding to digital image/graphics processor **74**.

These pseudo address registers may be used in global address unit **610** and local address unit **620** and with indices in any of the 12 permitted combinations of pre- and post-addition or subtraction, except that these may not be address destinations. There are restrictions on the permitted data transfers when using these pseudo address registers. These are called pseudo address registers because no actual address register corresponds to these designations. Instead each address unit employs hardware in conjunction with an identifier in a command register (to be later described) to produce the required address.

The particular addresses for the preferred embodiment of this invention are listed below in Table 29. The pseudo address register PBA produces an address of the form Hex "0100#000" and the pseudo address register DBA produces an address of the form Hex "0000#000", where # is the digital image/graphics processor number.

TABLE 29

Digital Image/ Graphics Processor Number	Parameter Memory Base Address	Data Memory Base Address
0	01000000	00000000
1	01001000	00001000
2	01002000	00002000
3	01003000	00003000

These pseudo address registers are advantageously used in programs written independent of the particular digital image/graphics processor. These pseudo address registers allow program specification of addresses that correspond to the particular digital image/graphics processor. Thus programs may be written which are independent of the particular digital image/graphics processor executing the programs.

Referring back to FIG. 27, address unit **120** forms respective addresses on global address port **121** and local address port **122**. In the least complex case, the global address generated by global address unit **610** passes through multiplexer **641** and is stored in global temporary address register GTA **651**. Global address port **121** passes this address together with byte strobe, read/write and select signals to crossbar **50**. Similarly the local address generated by local address unit **620** is stored in local temporary address register LTA **652** for supply to crossbar **50** via local address port **122** together with accompanying byte strobe, read/write and select signals. Global temporary address register **651** and local temporary address register **652** hold the generated addresses for reuse in case of crossbar contention. This is more convenient than recomputing the address for reuse because the possibility of address register modification would require conditional recomputation.

Sometimes an address generated by local address unit **620** passes to crossbar **50** via global address port **121** rather than by local address port **122**. Control circuit **654** determines if the address generated by local address unit **620** is a legal local address. Note that the local ports may only address the corresponding data or parameter memory. If local address unit **620** generates an address outside its permitted range, and no global port access is specified, then control circuit **654** signals control circuit **642** to cause multiplexer **641** to select the local address generated by local address unit **620**. This address is then stored in global temporary address register GTA **651**. If a global port access is specified, this is serviced first and then control circuit **654** signals control circuit **642** to cause multiplexer **641** to select the address stored in local temporary address register LTA **652**. in either case global temporary address register GTA **653** supplies the address to the global address port **121**.

Global/local address multiplexer register GLMUX **630** permits a single address to be formed from parts of the addresses generated by global address unit **610** and local address unit **620**. This is known as XY patching that forms a patched address. Global/local address multiplexer register GLMUX **630** is coupled to both global port source data bus Gsrc **105** and global port destination data bus Gdst **107** and can be accessed within the register space of digital image/graphics processor **71**. Global/local address multiplexer register GLMUX **630** includes 30 bits. For each bit position of global/local address multiplexer register GLMUX **630** a

TABLE 27-continued

Address Register	Register Assignment
A4	Local address unit
A5	reserved
A6	Global/Local address units shared stack pointer
A7	Local address unit read only, all zeros
A8	Global address unit
A9	Global address unit
A10	Global address unit
A11	Global address unit
A12	Global address unit
A13	reserved
A14	Global/Local address units shared stack pointer
A15	Global address unit read only, all zeros

Address registers A0, A1, A2, A3 and A4 are within local address unit **620** and are available for general use. Address register A5 is not supported in the current embodiment, but its address is reserved for future expansion of the local address unit **620**. Address registers A8, A9, A10, A11 and A12 are within global address unit **620** and are available for general use. Address register A13 is not supported in the current embodiment, but its address is reserved for future expansion of the global address unit **610**. Address registers A6 and A14 are embodied by a single register accessible by local address unit **620** at address A6 and by address unit **610** at address A14. This combined register A14/A6 will generally be used as a stack pointer. Note that stack operations are only allowed on aligned 32 bit word boundaries. Consequently the two least significant bits of combined register A14/A6 are hardwired to "00". Writing to these two bits has no effect and they are always read as "00". Registers A7 and A15 are also embodied by the same hardware and both global address sun-unit **610** and local address unit **620** may use this combined register in the same instruction. Register A7 is accessible to local address unit **620** and register A15 is accessible to global address unit **610**. Combined register A15/A7 is hardwired to all "0's". Writing to either of these two registers has no effect and they are always read as all "0's". In the preferred embodiment these two registers are embodied by the same hardware accessible at differing addresses.

Table 28 lists the index register assignments. Index registers **612** are coupled to both global port source data bus Gsrc **105** and global port destination data bus Gdst **107**. These connections permits register loads from memory, register stores to memory, and register to register data transfer with other registers within that digital image/graphics processor, such as data registers **200** within data unit **110**. Various uses of these connections will be described below.

TABLE 28

Index Register	Register Assignment
X0	Local address unit
X1	Local address unit
X2	Local address unit
X3	reserved
X4	reserved
X5	reserved

TABLE 28-continued

Index Register	Register Assignment
X6	reserved
X7	reserved
X8	Global address unit
X9	Global address unit
X10	Global address unit
X11	reserved
X12	reserved
X13	reserved
X14	reserved
X15	reserved

Index registers X0, X1 and X2 are within local address unit and are available for general use. index registers X3, X4, X5, X6 and X7 are not supported in the current embodiment, but their addresses are reserved for future expansion of the local address unit **620**. index registers X8, X9 and X10 are within global address unit **620** and are available for general use. Index registers X11, X12, X13, X14 and X15 are not supported in the current embodiment, but their addresses are reserved for future expansion of the global address unit **610**. Global address unit **610** generates a 32 bit address. Either an index stored in a specified index register within index registers **612** or an offset field from the instruction word is selected at multiplexer **613**. This selection is controlled by the instruction via instruction decode logic **660** (FIG. **27**). Multiplexer **613** also selects the size of the offset field again based on the instruction. As will be further discussed below, global address unit **610** may receive a 15 bit offset field or a 3 bit offset field. Whether the offset field is 15 bits or 3 bits, this value is zero extended to 32 bits before use.

Index scaler **614** optionally left shifts the data selected by multiplexer **613**. This optional left shift is selected by a scaled/unscaled input that corresponds to the function of the instruction. This left shift is 0, 1 or 2 bits depending on the indicated data size. As previously described the pixel data may be specified as 8 bits (byte), 16 bits (half word) or 32 bits (word). If scaling is selected, then the data is left shifted with zero filling 0 bit places for byte data, 1 bit place for half word data and 2 bit places for word data. Since no scaling ever occurs for byte data transfers, the instruction word bit specifying scaling is available for other purposes. In the preferred embodiment this instruction word bit is used as an additional offset bit. Thus if the data size is 8 bits, the instruction can supply a 16 bit offset index rather than a 15 bit offset index or a 4 bit offset index rather than a 3 bit offset index. This address index scaling feature permits addressing that is independent from the data size. This feature is useful in certain applications such as look up table operations.

Addition/subtraction unit **615** receives a base address from an address register selected by the instruction and the index. The instruction selects either addition of the index to the base address or subtraction of the index from the base address. The resultant forms one input to multiplexer **616**. The base address from the selected address register forms the other input to multiplexer **616**. Multiplexer **616** selects one of these addresses depending on whether the instruction specifies pre-indexing or post-indexing. If the instruction specifies pre-indexing, then the resultant of addition/subtraction unit **615** is selected by multiplexer **616** as the output address. If the instruction specified post-indexing, then the base address from address registers **611** is selected by multiplexer **616** as the output address.

The modified address may be written into the selected address register. In pre-indexing, then instruction selects

access is a memory read or memory write operation. A single bit field in the instruction field for each active port indicates whether the data transfer is a load operation, which is a memory read, or a store operation, which is a memory write. Control circuits **653** and **654** generate the corresponding read/write signal to crossbar **50** based upon the corresponding single bit field in the instruction word.

Each control circuit **653** and **654** generates two strobe signals. An active data-space select signal indicates that the memory transfer is to data memory. An active parameter-space select signal indicates that the memory transfer is to parameter memory. Neither select signal is active during execution of an instruction not specifying a data transfer operation via that port. Bit **24** of the generated address controls these select signals due to the address partitioning. The data-space select signal is active when bit **24** of the address is "0" and the parameter-space select signal is active when bit **24** of the address is "1".

Global address unit **610** and local address unit **620** may be used for additional arithmetic operations. The use of an address unit for arithmetic operations is called address unit arithmetic. An address unit arithmetic operation may be substituted for any memory load operation. Any instruction word with specifies data transfer operations includes a bit that specifies whether the data transfer is a load (data transfer from memory to a register) or a store (data transfer from a register to memory). These instruction words also include a bit that specifies whether the data is sign extended on load. Sign extension fills the higher order bits of the data written to the destination with the same state as the most significant bit of the data in case the data size is less than 32 bits. The otherwise meaningless combination of store with sign extend enables address unit arithmetic. Rather than fetching the memory data located at the address generated by the address unit and storing it in the destination register, an address unit arithmetic operation stores the calculated address in the destination register. Buffer **655** supplies the output from global temporary address register GTA **651** to global port source data bus Gsrc **105** for supply to a specified destination register when the instruction word indicates sign extend and a load operation. Similarly, buffer **656** supplies the output from local temporary address register LTA **652** to local port bus Lbus **103** for supply to a specified destination register when the instruction word indicates sign extend and a load operation. Under these conditions control circuits **653** and **654** do not generate their control signals to crossbar **50**. Thus the generated address is diverted from the address bus of crossbar **50** to the corresponding digital image/graphics processor data bus.

Address unit arithmetic operations enable additional parallel arithmetic operations. In the preferred embodiment, each digital image/graphics processor **71**, **72**, **73** and **74** can perform a multiply and three additions in one instruction. It is preferably also possible to perform a multiply, two additions and a data transfer operation in parallel in one instruction. All of the indexing, address modification and offset operations available for the corresponding load operation are available during address unit arithmetic. Thus an address unit arithmetic operation can compute a result to be stored in the destination register while also modifying a base address register either by pre-incrementing, post-incrementing, pre-decrementing or post-decrementing. An address unit arithmetic operation adding an offset index to a zero base address from address registers A15/A7 can load an offset field in parallel with any data unit operation. Address unit arithmetic operations can be performed conditionally in the same manner as conditional data transfers. As other conditional data transfers modification of the base address register occurs unconditionally, only the transfer of the result is conditional. The preferred embodiment also supports address unit arithmetic of patched addresses. Like all other address computations address unit arithmetic calculations occur in the address pipeline stage and are written to the destination register during the execute pipeline stage. Note that the "address" computed during an address unit arithmetic operation is not checked for range. This is because no actual memory access occurs when an address unit arithmetic operation executes.

Address unit arithmetic operations are best used to reduce the number of instructions needed for a loop kernel in a loop that is repeated a large number of times. Graphics and image operations often require large numbers of repetitions of short loops. Often reduction of a loop kernel by only a single instruction can greatly improve the performance of the process.

Data transfers between digital image/graphics processor **71** and memory **20** are made via data port unit **140**. Data port unit **140** handles data alignment, sign or zero extension and the like for data passing through. FIG. **30** illustrates details of this portion of buffer **147** illustrated in FIG. **3**. Note that this same structure could also be used within multiplexer buffer **143** of local data port **141**. Data from the crossbar data bus is divided into four data streams of 8 bits each. Data alignment multiplexer **151** selects and aligns the received data based upon the current data size, endian mode and the two least significant bits of the generated address. For a data size of 32 bits, no selection or alignment is needed and the four 8 bit data streams pass through data alignment multiplexer **151** unchanged. For a data size of 16 bits, data alignment multiplexer **151** selects either the most significant 16 bits or the least significant 16 bits for supply via the 16 least significant output bits. This selection contemplates the current endian mode and address bits **1–0**. If address bit **1** is "0", then data alignment multiplexer **151** selects the least significant 16 bits in little endian mode and the most significant bits in big endian mode. The opposite selection is made if address bit **1** is "1". Similarly, if the data size is 8 bits, data alignment multiplexer **151** selects either bits **31–24**, bits **23–16**, bits **15–8** or bits **7–0** based upon the current endian mode and address bits **1–0**.

Once the data selection and alignment have been made, sign/zero extend multiplexer **152** provides sign or zero extension. For the case of 32 bit data, no sign or zero extend is made and the data passes through sign/zero extend multiplexer **152** unchanged. Bus drivers **153** then supply the corresponding destination bus; global port data destination bus Gdst **107** for the global port and local port data bus Lbus **103** for the local port. If the data size is 16 bits, then sign/zero extend multiplexer **152** passes data bits **15–0** unchanged. For this case data bits **31–16** are filled with "0" if zero extension is selected. Data bits **31–16** are sign extended, that is filled with the state of bit **15**, is sign extension is selected. For 8 bit data, sign/zero extend multiplexer **152** passes bits **7–0** unchanged. Bits **31–8** are filled with "0" if zero extension is selected and filled with the state of bit **7** is sign extension is selected.

This data selection, alignment, and sign or zero extension is available for register to register moves as well as register loads from memory. For register to register moves the instruction word includes a field that specifies a two bit item number. This item number, treated as if in little endian mode, substitutes for the address bits **1–0**. In other respects the circuit illustrated in FIG. **30** operates as just described.

Data port unit **140** operates specially for local port illegal addresses. Recall that each local port can only address

memories corresponding to that digital image/graphics processor. If the local address unit **620** generates an address outside its permitted range, then this address is shunted to global address port **121**. If a global port access is also specified for that instruction, this is serviced first and then the local port access is serviced via global address port **121**. Under these conditions during a store operation data from local data port bus Lbus **103** supplies buffer multiplexer **146**, which supplies to the addressed memory location via global data port **148**. Similarly, when using the global port for a local load operation buffer multiplexer **143** supplies the received data from global data port **148** to local port data bus Lbus **103**.

FIG. **31** illustrates in block diagram form program flow control unit **130**. Program flow control unit **130** performs all the operations that occur during the fetch pipeline stage. Program flow control unit **130** controls: fetching instruction words from the corresponding instruction cache; instruction cache management including handshakes with transfer controller **80**; program counter modification by branches, interrupts and loops; pipeline control, including control over data unit **110** and address unit **120**; synchronization with other digital image/graphics processors in synchronized MIMD mode; and receipt of command words from other processors. As illustrated in FIG. **31** program flow control unit **130** includes the following registers: program counter PC **701**; instruction pointer-address stage IPA **702**; instruction pointer-execute stage IPE **703**; instruction pointer-return from subroutine IPRS **704**; three loop end registers LE2–LE0 **711**, **712** and **713**; three loop start registers LS2–LS0 **721**, **722** and **723**; three loop counter registers LC2–LC0 **731**, **732** and **733**; three loop reload registers LR2–LR0 **741**, **742** and **743**; loop control register LCTL **705**; interrupt enable register INTEN **706**; interrupt flag register INTFLG **707**; four cache tag registers TAG3–TAG0, collectively called cache tag registers **708**; a read only CACHE register **709**; and a communications register COMM **781**. There are two sets of write only register addresses (LRS2–LRS0 and LRSE2–LRSE0) employed for fast hardware loop initialization. These will be further discussed below.

Program flow control unit **130** also includes an instruction register-address stage IRA **751** and an instruction register-execution stage IRE **752**. These registers are not user accessible and do not appear in the register space. Instruction register-address stage IRA **751** contains the instruction word for the current address pipeline stage. Instruction register-execution stage IRE **752** contains the instruction word for the current execute pipeline stage. These registers control the operations during the respective address and execute pipeline stages. The program flow control unit **130** pushes the fetched instruction word located at the address in program counter PC **701** into the instruction register-address stage IRA **751**. In addition, the pipeline pushes the instruction word in the instruction register-address stage IRA **751** into the instruction register-execute stage IRE **752** upon each pipeline stage advance.

Program flow control unit **130** operates predominantly in the Fetch pipeline. Since the program flow control unit **130** contains the instruction register-address stage IRA **751** and instruction register-execute stage IRE **752**, it extracts and distributes control information needed by data unit **110** and address unit **120** via opcode bus **133**. Program flow control unit **130** also controls the aligner/extractors on the data port unit **140**.

The major task of program flow control unit **130** is control of instruction fetch during the fetch pipeline stage. The address of the next instruction word to be fetched is stored in program counter PC **701**. FIG. **32** illustrates schematically the bits of program counter PC **701**. In the preferred embodiment of this invention, internal and external memory is byte addressable. That is, each address word points to a byte (8 bits) of data in memory. As explained in detail below, each instruction word of digital image/graphics processor **71** is a 64 bit double word, which is 8 bytes. Since these instruction words are aligned on even double word boundaries, only 29 bits are necessary to specify any such instruction word. As illustrated in FIG. **32** bits **31–3** of program counter PC **701** provide this 29 bit double word address. During normal sequential instruction operation program flow control unit **130** increments bit **3** of program counter PC **701** to address the next 64 bit instruction.

Program counter PC **701** has two write register addresses. Writing to program counter PC **701** executes a subroutine call. The write alters program counter PC **701**. At the same time program flow control unit **130** causes the previous contents of program counter PC **701** to be written into instruction pointer-return from subroutine IPRS **704**. This enables a return instruction to reload program counter PC **701** from instruction pointer-return from subroutine IPRS **704**. Writing to a different register address designated branch BR executes a software branch. This write alters only program counter PC **701** and instruction pointer-return from subroutine IPRS **704** is unchanged.

As noted above bits **2–0** of program counter PC **701** are not needed to specify instruction words. These otherwise unused bits are employed to specify other things. These bits include an "S" bit (bit **2**), a "G" bit (bit **1**) and an "L" bit (bit **0**).

The "S" bit (bit **2**) indicates whether the digital image/graphics processor **71** is in the synchronized MIMD mode. As previously described, when in the synchronized MIMD mode program control flow unit **130** inhibits fetching the next instruction word until all synchronized processors are ready to proceed. If the "S" bit is "1", then the digital image/graphics processor **71** is currently executing synchronized code. Note that the identity of the other digital image/graphics processors synchronized to digital image/graphics processor **71** is stored in the communications register COMM **781**. Otherwise, digital image/graphics processor **71** will not wait for other digital image/graphics processors to be ready before fetching the next instruction word. Execution of a lock instruction (LCK) sets this "S" bit of program counter PC **701** during the address pipeline stage to enable synchronized MIMD mode. Execution of an unlock (UNLCK) instruction clears this "S" bit during the address pipeline stage thus disabling the synchronized MIMD mode. Normal register writes to program counter PC **701** do not change the state of this "S" bit.

The "G" bit (bit **1**) indicates whether global interrupts are enabled. When this "G" bit is "0", the program flow control unit **130** ignores all interrupt sources, except the emulation trap. If this "G" bit is "1", then program flow control unit **130** responds to those interrupt sources individually enabled in interrupt enable register INTEN **706**. Execution of an enable interrupt instruction (EINT) sets this "G" bit of program counter PC **701** during the address pipeline stage to enable interrupts. Execution of a disable interrupt instruction (DINT) clears this "G" bit during the address pipeline stage of thereby disabling most interrupt sources. Normal register writes to program counter PC **701** do not change the state of this "G" bit.

The "L" bit (bit **0**) indicates whether hardware loop logic is enabled. This hardware loop logic will be fully described

below. If the "L" bit is "1", then the hardware loop logic is disabled. Otherwise, hardware loops are individually enabled according to the loop control register LCTL **708**. Hardware loops are normally disabled via this "L" bit only during the return sequence from an interrupt, because loops are "unwrapped" during the entry into an interrupt routine. Normal register writes to program counter PC **701** do not change the state of this "L" bit.

FIG. **33** illustrates schematically the bits of instruction pointer-address stage IPA **702**. This register is loaded with the contents of program counter PC **701** upon each pipeline stage advance. In the first two pseudo-instructions of an interrupt, the "L" bit (bit **0**) of instruction pointer-address stage IPA **702** is forced to "1" whatever the state of this bit in program counter PC **701**. The other bits of program counter PC **701** are copied into instruction pointer-address stage IPA **702** without alteration. This register stores the address of the instruction currently in the Address pipeline stage.

Instruction pointer-execute stage IPE **703** is loaded with the contents of instruction pointer-address stage IPA **702** upon each pipeline stage advance. This register is useful in relative program counter computations. Note that instruction pointer-execute stage IPE **703** stores the address of the instruction currently in the execute pipeline stage. Using this register for relative program counter computations is better than using program counter PC **701** due to the possibility of branches, loops or interrupts and because no offset is required.

Instruction pointer-return from subroutine register IPRS **704** stores the subroutine return address. FIG. **34** illustrates the bits of this register schematically. instruction pointer-return from subroutine register IPRS **704** is updated with the address previously stored in program counter PC **701** incremented at bit **3** whenever software writes to program counter PC **701**. This is the address following the second delay slot of the software branch. Thus, as implied by the name, instruction pointer-return from subroutine register IPRS **704** stores the address for returns from subroutines. Executing a return instruction loads the address stored in instruction pointer-return from subroutine register IPRS **704** into program counter PC **701** during the execute pipeline stage. Only bits **31–3** of instruction. pointer-return from subroutine register IPRS **704** are used. Bits **2–0** of program counter PC **701** are not stored in instruction pointer-return from subroutine IPRS **704** upon a software branch and these bits are not read from instruction pointer-return from subroutine IPRS **704** during restoration of program counter PC **701**.

The program flow control unit of each digital image/graphics processor includes an instruction cache controller **760**. This instruction cache controller **760** includes a set of four cache tag registers TAG3–TAG0 **708**, a least recently used control circuit **761** and an address encoder **762**. The instruction cache controller **760** controls a section of memory dedicated to instruction caching for that digital image/graphics processor. This instruction cache memory is preferably 2K bytes in size. Instruction cache controller **760** treats the instruction cache memory as holding 256, 64 bit instructions in one set with 4 blocks supported by 4-way least recently used operations. Each block has 4 sub-blocks of 16 instructions. Thus each of the cache tag registers TAG3–TAG0 **708** includes 4 "present" bits for a total of 16 "present" bits.

FIG. **35** illustrates the fields of each cache tag register TAG3–TAG0. The tag value field (bits **31–9**) of each of the tag registers holds a tag value. This tag value is the virtual address of the start of the corresponding cache block in the instruction cache memory. Sub-block present bits (bits **8–5**) of each cache tag register TAG3–TAG0 are associated with the respective four sub-blocks **3–0** in the block to which that cache tag register relates. Thus bit **8** represents the most significant sub-block and bit **5** represents the least significant sub-block. The "LRU" field (bits **1–0**) indicates how recently the block was used. These bits are as defined in Table 31.

TABLE 31

LRU bits		Position in
1	0	use stack
0	0	most-recently used
0	1	next-most recently used
1	0	next-least recently used
1	1	least recently used

Bits **4** to **2** of cache tag registers TAG3–TAG0 **708** are not implemented. These bits are reserved for a possible extension of the instruction cache memory to include additional sub-blocks. Cache tag registers TAG3–TAG0 **708** appear in the register map as listed in Tables 37 and 38.

Instruction cache controller **760** of each digital image/graphics processor **71**, **72**, **73** or **74** may be flushed by master processor **60** or by the digital image/graphics processor itself. Note that a cache flush resets only the cache tag registers TAG3–TAG0 **708** within program flow control unit **130** and does not clear data from the corresponding instruction cache memory. An instruction cache flush is performed by writing a cache flush command word to address register A15 with the "F" bit (bit **28**) set. Reset does not automatically flush the cache. An instruction cache flush causes the cache tag value field to be set to the cache tag register's own number (i.e., TAG3=3, TAG2=2, TAG1=1, TAG0=0), clears all their present bits, and sets the LRU bits to the tag register's own number (i.e., TAG3(LRU)="11", TAG2(LRU)="10", TAG1(LRU)="1" and TAG0(LRU)="00"). Cache tag register TAG3 is thus the least-recently-used following a cache flush.

Program flow control unit **130** compares corresponding bits of the address stored in program counter PC **701** to the cache tag registers TAG3–TAG0 **708** during each fetch pipeline stage. This comparison yields either a cache miss result or a cache hit result. A cache miss may be either a block miss or a sub-block miss. In a block miss the most significant 23 bits of program counter PC **701** does not equal the corresponding 23 bits of any of the cache tag registers TAG3–TAG0 **708**. In this case, least recently used control circuit **761** chooses the least recently used block to discard, and clears all the present bits of the corresponding cache tag register. In a sub-block miss the most significant 23 bits of program counter PC **701** matches the corresponding 23 bits of one of the cache tag registers TAG3–TAG0 **708**, but the present bits (one of bits **8–5** of the tag register) indicating presence of the sub-block corresponding to bits **8–7** of program counter PC **701** is "0". This means that one of the cache tag registers TAG3–TAG0 **708** is assigned that memory block, but that the sub-block is not present within the instruction cache.

If either type of cache miss occurs, then program flow control unit **130** requests transfer controller **80** to service the instruction cache memory via an external access. Program control flow unit **130** passes the external address and the internal sub-block address to the transfer controller **80**. Program flow control unit **130** signals transfer controller **80**

the cache miss information via crossbar **50**. Transfer controller **80** services the cache miss by fetching the entire sub-block of instructions including the address of the currently sought instruction word. This block of instructions is stored in the least recently used block within the instruction cache memory **21**, **26**, **31** and **36** corresponding to the requesting digital image/graphics processor **71**, **72**, **73** and **74**, respectively. Program flow control unit **130** then sets the proper values in the corresponding cache tag register TAG3–TAG0 **708**. The instruction fetch operation is then repeated, with a cache hit guaranteed.

Cache miss information may be accessed by reading from the register in the register space at register bank "1111" register number "000". This register is called the CACHE register **709** in Table 38. Program flow control unit **130** provides 27 bits. These 27 bits are the 23 most significant address of program counter PC **701** (the tag bits) plus 2 sub-block bits from cache tag registers TAG3–TAG0 **708** and two bits encoding the identity of the least-recently-used block from least recently used control circuit **761**. CACHE register **709** is read only, any attempt to write to write to this register is ignored. Thus CACHE register **709** is connected to only global port source data bus Gsrc bus **105** and not connected to global port destination data bus Gdst **107**.

If a cache hit occurs, then the desired instruction word is stored in the corresponding instruction cache. As previously described, each instruction cache memory **21**, **26**, **31**, **36** includes 2K bytes. Since internal and external memory is byte addressable in the preferred embodiment, 11 address bits are required. However, each instruction is aligned with a 64 bit double word boundary and thus the three least significant bits of an instruction address are always "000". The 2 most significant bits of the 11 bit instruction address on instruction port address bus **131** correspond to the cache tag register TAG3–TAG0 **708** successfully matched with program counter PC **701**. These address bits **10–9** are encoded as shown in Table 32.

TABLE 32

Address bits		Cache tag
10	9	register
0	0	TAG0
0	1	TAG1
1	0	TAG2
1	1	TAG3

The bits **8–3** of the instruction address on instruction port address bus **131** are bits **8–3** of the 29 bit double word address stored in program counter PC **701**. The cache tag comparison is made fast enough to output the 8 bit address via the instruction port with an implied read signal from the digital image/graphics processor to the corresponding instruction cache memory. This retrieves the addressed 64 bit instruction word into instruction register-address stage IRA **751** before the end of the fetch pipeline stage.

Program flow control unit **130** next updates program counter PC **701**. If the next instruction is at the next sequential address, program control flow unit **130** post increments program counter PC **701** during the fetch pipeline stage. Note this post increment means that program counter PC **701** stores the address of the next instruction to be fetched. Otherwise, program control flow unit **130** loads the address of the next instruction into program counter PC **701** according to loop logic **720** (FIG. 37) or software branch. When in the synchronized MIMD mode, program flow control unit delays the instruction fetch until all the digital image/graphics processors specified by sync bits in communications register COMM **781** are synchronized.

Program flow control unit **130** includes loop logic **720** employed with a number of registers in nested zero-overhead looping and a variety of other powerful instruction flow control functions. Examples of these other functions include: multiple ends to the same loop; zero-delay branches without necessarily returning; zero-delay "calls and returns"; and conditional zero-delay branches. The basic function of loop logic **720** is nested zero-overhead looping. For each of three possible loops there are four registers. These are: loop end registers LE2 **711**, LE1 **712** and LE0 **713**; loop start registers LS2 **721**, LS1 **722** and LS0 **723**; loop count registers LC2 **731**, LC1 **732** and LC0 **733**; and loop reload registers LR2 **741**, LR1 **742** and LR0 **743**. The entire loop logic process is controlled by the status of loop logic control register LCTL **705** in conjunction with the loop enable bit (bit **0**) of program counter PC **701**. In addition there are several register address locations LRS2–LRS0 and LRSE2–LRSE0 that simultaneously load more than one of the primary registers.

Each set of four registers controls an independent zero-overhead loop. A zero-overhead loop is the solution to a problem caused by the pipeline structure. A software branch performed by loading an address into program counter PC **701** occurs during the execute pipeline stage. Such a branch does not take place immediately because it does not change two instructions that were already fetched and in the instruction pipeline. These two instructions were fetched during the previous two fetch pipeline stages. This delay in branch implementation is called a pipeline hit and the two instructions following the branch instruction are called delay slots. Sometimes clever programming enables useful work during the delay slots, but this is not always possible. Loop logic **720** operates during the fetch pipeline stage and, once some set up is accomplished, enables loops and branches without pipeline hits. Note that once the appropriate registers are loaded loop logic **720** does not require a branch instruction during looping and does not produce any delay slots. This loop logic **720** may be especially useful in algorithms with nested loops with numerous repetitions.

A simple example of loop logic **720** operation follows. Set up of loop logic **720** includes loading a particular loop end register, and the corresponding loop start register, loop count register and loop reload register. For example the loop end address is loaded into loop end register LE0 **713**, the loop start address is loaded into loop start register LS0 **723** and the number of loop repetitions desired is loaded into loop count register LC0 **733** and loop reload register LR0 **743**. During each fetch pipeline stage loop logic compares the address stored in program counter PC **701** with the loop end address stored in loop end register LE0 **713**. If the current program address equals the loop end address, loop logic **720** determines if the loop count stored in the corresponding loop count register, in this case loop count register LC0 **733**, is "0". If the loop count is not "0", then loop logic **720** loads the loop start address stored in loop start register LS0 **723** into program counter PC **701**. This repeats the loop starting from the loop start address. In addition, loop logic **720** decrements the loop count stored in the corresponding loop count register, in this case loop count register LC0 **733**. If the loop count in the corresponding loop count register is "0", then no branch is taken. Program flow control unit **130** increments program counter PC **701** normally to the next sequential instruction. In addition, loop logic **720** loads the loop count stored in the loop reload register LR0 into the

loop count register LC0. This prepares loop logic 720 for another set of repetitions and is useful for inner loops of nested loops. Because all these processes occur during the fetch pipeline state no pipeline hit takes place.

FIG. 36 illustrates loop logic control register 705. Loop logic control register 705 controls operation of loop logic 720 based upon data stored in three sets of bits corresponding to the three loop end registers LE2–LE0 711–713. Loop logic control register 705 bits 3–0 control the loop associated with loop end register LE0 713, bits 7–4 control the loop associated with loop end register LE1 712, and bits 11–8 control the loop associated with loop end register LE2 711. The "E" bits (bits 11, 7 and 3) are enable bits. A "1" in the "E" bit enables the loop corresponding the associated loop end register. A "0" disables the associated loop. Thus setting bits 11, 7 and 3 to "0" completely disables loop logic 720. Each loop end register LE2–LE0 has an associated "LCn" field that assigns a loop count register LC2–LC0 for that loop end register. The coding of the "LCn" field is given in Table 33.

TABLE 33

LCn field			Loop Count Register
0	0	0	none
0	0	1	LC0
0	1	0	LC1
0	1	1	LC2
1	X	X	reserved

The assigned loop count register stores the corresponding loop count and is decremented each time the program address reaches the associated loop end address. Although the "LCn" field is coded to allow every loop end register to use any loop count register, not all combinations are supported in the preferred embodiment. In the preferred embodiment the "LCn" field may assign: loop count register LC2 or LC0 to loop end register LE2 711; register LC1 or LC0 to loop end register LE1 712; and only loop count register LC0 to loop end register LE0 713. In the case of a "LCn" field of "000", no loop count register is used and the program always branches to the loop start address stored in the corresponding loop start register. Also note that if bit 0 of program counter PC 701 is "0", then loop logic 720 is inhibited regardless of the status of loop control register LCTL 705. This permits loop logic inhibition without losing the assignment of loop count registers to loop end registers. When the count in the assigned loop count register reaches "0", encountering the loop end address does not load program counter PC 701 with the address in the corresponding loop start register. Instead the loop count register is reloaded with the contents of the corresponding loop reload register LR2–LR0. By assigning loop counter register LC0 733 to two or three loop end registers LE2–LE0, multiple end points to a loop are supported. Note that the most significant bits of loop control register LCTL 705 and the "1XX" codings of the respective "LCn" fields are reserved for a possible extension of the loop logic to include more loops.

FIG. 37 illustrates loop logic 720. Loop logic 720 includes previously mentioned: program counter PC 701; loop logic control register LCTL 705; the three loop end registers LE2–LE0 711, 712 and 713; the three loop start registers LS2–LS0 721, 722 and 723; the three loop counter registers LC2–LC0 731, 732 and 733; the three loop reload registers LR2–LR0 741, 742 and 743; comparitors 715, 716 and 717; priority logic 725; loop logic control register "LCn" field decoders 735, 736 and 737; and zero detectors 745, 746 and 47. The respective "E" fields of loop logic control register LCTL 705 selectively enable comparitors 715, 716 and 717 and loop logic control register "LCn" field decoders 735, 736 and 737. Comparitors 715, 716 and 717 compare the address stored in program counter PC 701 with respective loop end registers LE2 711, LE1 712 and LE0 713. Loop logic control register "LCn" field decoders 735, 736 and 737 decode respective "LCn" fields of loop logic control register LCTL 705, ensuring that the assigned loop count register LC2–LC0 is decremented upon reaching a loop end. Zero detectors 745, 746 and 747 enable reload of respective loop count registers 731, 732 and 733 from the corresponding loop reload registers 741, 742 and 743 when the loop count reaches "0".

Priority logic 725 decrements the assigned loop count register LC2–LC0 or loads program counter PC with the loop start address in loop start register LS2–LS0 depending upon the corresponding zero detection. If two or three loops end at the same address then priority logic 725 set priorities for the loop end registers in the order from loop end register LE2 (highest) to loop end register LE0 (lowest). If no zero detector 745, 756 or 747 detects "0", then the loop start register LS2–LS0 associated with the highest priority loop end register LE2–LE0 matching the program counter PC 701 is loaded into program counter PC 701 and the loop count register LC2–LC0 assigned to that highest priority loop end register LE2–LE0 is decremented. If at least one zero detector 745, 756 or 747 detects zero, then the zero-value loop count register LC2–LC0 corresponding to each zero value loop end register LE2–LE0 matched is reloaded from the corresponding loop reload register LR2–LR0 and the non-zero loop count register LC2–LC0 assigned to the highest priority non-zero loop end register LE2–LE0 matched is decremented. Program counter PC 701 is loaded with the loop start address associated with the highest priority loop end register that has a corresponding non-zero loop count register. Zero detector 747 has a disable line to zero detector 746 to disable zero detector 746 from causing reload if zero detector 747 detects a zero. Both zero detectors 747 and 746 may disable zero detector 745 from causing reload if either zero detector 747 or 746 detect zero. Thus three nested loops may end at the same instruction with the loop associated with loop end register LS2 711 the inner loop, and the loop associated with loop end register LS0 the outer loop.

Loops can have any number of instructions within the address limit of the loop end registers LE2–LE0. Loop end registers LE2–LE0 and loop start registers LS2–LS0 preferably include 29 address bits in the same fashion as program counter PC 701. The number of repetitions possible is limited by the capacity of the loop count registers and the loop reload registers. In the preferred embodiment the loop count registers LC2–LC0 and the loop reload registers LR2–LR0 each have 32 bits as most registers on digital image/graphics processor 71. For the sake of size, the capacity of the loop count and loop reload registers may be limited to 16 bits rather than 32 bits. In this case, the most significant 16 bits of these registers are not implemented. With 16 bit loop count and loop reload registers loops larger than 2^{16}=65536 can be implemented using outside software loops to restart the hardware loops. The addresses for loop starts and loop ends can be coincident, resulting in a single instruction loop.

FIG. 38 illustrates an example of a program having three ends to one loop. This is achieved by assigning loop count register LC0 733 to each of the loop end registers LE2–LE0. In the example illustrated in FIG. 38 loop start register LC0

723 and loop start register LC2 721 store the same address. Loop start register LC1 722 stores a different start address. The program begins at block 801. Processing block 802 initializes the loops including storing the respective loop end addresses in loop end registers LE2–LE0, storing the respective loop start addresses in loop start registers LS2–LS0, loading loop control register LCTL 705 to enable all three loops and assign loop count register LC0 733 to all loop end registers LE2–LE0. Processing block 803 is an instruction block 0 starting at loop start address 1. Processing block 804 is an instruction block 1 starting at start address 0 and 2. Decision block 805 is a conditional branch instruction 1. Decision block 806 is a conditional branch instruction 2. Assuming neither condition 1 nor condition 2 is satisfied, then the program executes processing block 807 consisting of instruction block 3. Decision block 808 is the hardware loop decision corresponding to the loop end address stored in loop end register LE0 713. If the count stored in loop count register LC0 is non-zero, the program flow returns to loop start address 0 that repeats the loop starting with instruction block 1. If the count stored in loop count register LC0 is "0", the program ends at end block 813. In the case that condition 1 is not satisfied and condition 2 is satisfied, then the program executes processing block 809 consisting of instruction block 4. Decision block 810 is the hardware loop decision corresponding to the loop end address stored in loop end register LE2 711. If the count stored in loop count register LC0 is non-zero, the program flow returns to loop start address 2 that is the same as loop start address 0 which repeats the loop starting with instruction block 1. If the count stored in loop count register LC0 is "0", the program ends at end block 813. In the case that condition 1 is satisfied, then the program executes processing block 811 consisting of instruction block 5. Decision block 812 is the hardware loop decision corresponding to the loop end address stored in loop end register LE1 712. If the count stored in loop count register LC0 is non-zero, the program flow returns to loop start address 1 and repeats the loop starting with instruction block 0. If the count stored in loop count register LC0 is "0", the program ends at end block 813. The loop could finally terminate at any of the loop end addresses according to the condition encountered by the conditional branches on the final time through the loop.

To save instructions during loop initialization, any write to a loop reload register LR2–LR0 writes the same data to the corresponding loop count register LC2–LC0. In the preferred embodiment, writing to a loop count register LC2–LC0 does not affect the corresponding loop reload register LR2–LR0. The reason for this difference will be explained below. When restoring loop values after task switches, the loop reload registers LR2–LR0 should be restored before restoring the loop count registers LC2–LC0. Thus the form for initializing a single loop is:

LSn=loop start address

LEn=loop end address

LRn=loop count this also sets LCn=loop count

Load LCTL with bits to enable loop n, and assign LCn to LEn

Begin loop

This procedure is suitable for loading a number of loops, which execute for a long time. This initialization procedure is repeated to implement additional loops. Note that since the loop registers are loaded by software in the execute pipeline stage and used by the hardware in the fetch pipeline stage, there should be at least two instructions between loading any loop register and the loop end address where that loop register will be used.

The loop start address and the loop end address can be made independent of the position of the loop within the program by loading the loop start register LS2–LS0 and the loop end register LE2–LE0 as offsets to instruction pointer-execute stage register IPE 703. Recall that instruction pointer-execute stage register IPE 703 stores the address of the instruction currently in the execute pipeline stage. For example, the instruction:

LS0=IPE+88

loads loop start register LS0 723 with a value 11 instructions (88 bytes) ahead of the current instruction. A similar instruction can load a loop end register LE2–LE0.

The preferred embodiment of this invention includes additional register addresses to support even faster loop initialization for short loops. There are two sets of such register addresses, one set for multi-instruction loops and one set for single instruction loops. Writing to one of the register addresses LRS2–LRS0 used for multi-instruction loops loads the corresponding loop reload register LR2–LR0 and its corresponding loop counter LC2–LC0. This write operation also loads the corresponding loop start LS2–LS0 register with the address following the current address stored in program counter PC 701. This write operation also sets corresponding bits in loop control register LCTL 708 to enable the relevant loop. Thus, if n is a register set number from 2–0, writing to LRSn: loads LRn and LCn with the specified count; loads LSn with PC+1; loads LCTL to enable LEn and assign LCn. These operations all occur in a single cycle, during the execute pipeline stage. There thus must be two delay slots between this instruction and the start of the loop. The instruction sequence for this multi-instruction loop short form initialization is:

LEn=loop end address

LRSn=count

delay slot 1

delay slot 2

loop start address:

1st_instruction_in_loop loop_instruction loop_instruction

loop end address:

last_instruction_in_loop

Note that the loop could be as long as desired within the register space of the corresponding loop end register and loop start register. Also note that writing to LRSn automatically sets the loop start address as the instruction following the second delay slot.

Another set of register addresses is used for short form initialization of a single instruction loop. Writing to one of the register addresses LRSE2–LRSE0 initializes a single instruction loop. If n is a register set number from 2–0, writing to LRSEn: loads loop reload register LRn and loop count register LCn with the count; loads loop start register LSn with the address following the address currently in program counter PC 701; loads loop end register LEn with the address following the address currently in program counter PC 701; and sets loop control register LCTL 705 to enable loop end register LEn and assign loop count register LCn. As with writing to LRSn: these operations all occur in a single cycle during the execute pipeline stage and two delay slots are required between this instruction and the start of the loop. The instruction sequence for this single instruction loop short form initialization is:

LRSEn=count

delay slot 1

delay slot 2

loopn:

one_instruction_loop

This instruction sequence sets the loop start and loop end to the same address. This thus allows a single-instruction to be repeated count+1 times.

These short form loop initializations calculate the loop start address and the loop end address values from the address stored in program counter PC **701**. They should therefore be used with care within the delay slots of a branch. If the branch is taken, the loop start address, and the loop end address for the case of LRSE2—LRSE0, is calculated after program counter PC **701** is loaded with the branch address. This effect can be annulled if the branch is conditional, by setting the loop initialization to be conditional upon the inverse condition.

These short form loop initializations and the standard loop initialization, do involve delay slots in much the same manner as software branches. However, the delay slots necessary for loop initialization occur once each loop initialization. The delay slots for branches formed with software loops occur once each branch instruction. In addition, there is a greater likelihood that useful instructions can occupy the delay slots during loop initialization than during loop branches. Thus the overhead needed for loop initialization can be much less than the overhead involved in software branches, particularly in short loops.

Software branches have priority over loop logic **720**. That is if a loop end register LE2–LE0 stores the address of the second delay slot instruction following a program counter load operation, then loop logic **720** is inhibited for that cycle. Thus the loop counter is not decremented, nor will any loop logic **720** program counter load take place. This enables a conditional software exit from a loop. If the loop logic **720** hardware loop has a single conditional branch instruction, then this instruction may be executed three times if the condition remains true. This is illustrated in FIG. **39**. In instruction slot **901** the branch condition is not true so the branch is unsuccessful. Loop logic **720** has already reloaded the same instruction during the fetch pipeline stage of instruction slot **902**. In instruction slot **902** the branch condition is true and the branch is taken, thereby loading the address of a target instruction into program counter PC **701**. This change in program counter PC **701** does not change the two already loaded examples of the branch instruction in the pipeline in instruction slots **903** and **904**. Assuming the branch condition is still true, the execute pipeline stage of these instruction slots loads the address of the target instruction into program counter PC **701**. Thus the branch is taken three times in instruction slots **902**, **903** and **904** and the target instruction executes three times in instruction slots **905**, **906** and **906**. Finally in instruction slot **908** the instruction following the target instruction is reached. As further explained below, the single branch instruction may be coded with parallel operations that would also be executed multiple times and that may change the branch condition.

Loop control logic **720** permits zero delay branches and zero delay conditional branches. In these cases the address of the point from which the branch is to be taken is loaded into a loop end register LE2–LE0. The destination address of the branch is loaded into the. assigned loop start register LS2–LS0. Zero-delay branches may be implemented in two ways. Following loop initialization, the assigned loop count register LC2–LC0 is set to a non-zero number. Alternatively, the corresponding "LCn" field in loop control register LCTL **705** may be set to "000". In either case the branch will always be taken during the fetch pipeline stage with no pipeline hit or delay slots. Conditional zero-delay branches (flow chart diamonds) are implemented similarly. During initialization the corresponding loop count register LC2–LC0 is assigned to the loop end register LE2–LE0 by setting the corresponding "LCn" field in loop control register LCTL. Before the conditional branch, a conditional value is loaded into the assigned loop count register LC2–LC0. Upon encountering the loop end address, either the branch is taken to the loop start address stored in the corresponding loop start register LS2–LS0 if the conditional value is non-zero, or the branch is not taken if the conditional value is zero. Since the loop registers are loaded by software in the execute pipeline stage and used by the hardware in the fetch pipeline stage, there should be at least two instructions between loading any loop register and the branch or conditional branch instruction at the loop end address. Otherwise, the previous value for that loop register is used by loop logic **720**.

Referring back to FIG. **31**, program flow control unit **130** handles interrupts employing interrupt enable register INTEN **706** and interrupt flag register INTFLG **707**. Program flow control unit **130** may support up to 32 interrupt sources represented by selectively setting bits of interrupt flag register INTFLG **707**. Each source can be individually enabled via interrupt enable register INTEN **706**. Pending interrupts are recorded in interrupt flag register INTFLG **707**, which latches interrupt requests until they are specifically cleared by software, normally during the interrupt routine. The individual interrupt flag can alternatively be polled and cleared by a software loop.

FIG. **40** illustrates the field definitions for interrupt enable register INTEN **706** and interrupt flag register INTFLG **707**. The bits labeled "r" are reserved for future use and bits labeled "–" are not implemented in the preferred embodiment but may be used in other embodiments. Interrupts are prioritized from left to right. Each interrupt source can be individually enabled by setting a "1" in the corresponding Enable (E) bit of interrupt enable register INTEN **706**. The interrupt source bits of interrupt flag register INTFLG **707** are in descending order of priority from right to left: Emulation interrupt ETRAP, which is always enabled; XY patch interrupt; task interrupt; packet request busy interrupt PRB; packet request error interrupt PRERR; packet request successful interrupt PREND; master processor **60** message interrupt MPMSG; digital image/graphics processor **71** message interrupt DiGPOMSG; digital image/graphics processor **72** message interrupt DIGP1MSG; digital image/graphics processor **73** message interrupt DIGP2MSG; digital image/graphics processor **74** message interrupt DIGP3MSG. Bits **31–28** are reserved for message interrupts from four additional digital image/graphics processors in an implementation of multiprocessor integrated circuit **100** including eight digital image/graphics processors.

The "W" bit (bit **0**) of interrupt enable register INTEN **706** controls writes to interrupt flag register INTFLG **707**. This bit would ordinarily control whether the emulation interrupt is enabled. Since in the preferred embodiment the emulation interrupt cannot be disabled there is no need for an enable bit for this interrupt in interrupt enable register INTEN **706**. Bit **0** of interrupt enable register INTEN **706** modifies the behavior of the interrupt flag register INTFLG **707**. When the "W" bit of interrupt enable register INTEN **706** is "1", software writes to interrupt flag register INTFLG **707** can only set bits to "1". Under these conditions, an attempt to write a "0" to any bit of interrupt flag register INTFLG **707** has no effect. When this "W" bit "0", writing a "1" to any bit of interrupt flag register INTFLG **707** clears that bit to "0". An attempt to write a "0" to any bit of

interrupt flag register INTFLG 707 has no effect. This allows individual interrupt flags within interrupt flag register INTFLG 707 to be cleared without disturbing the state of others. Each interrupt service routine should clear its corresponding interrupt flag before returning because these flags are not cleared by hardware in the preferred embodiment. The emulation interrupt ETRAP, the only exception to this, is cleared by hardware because this interrupt is always enabled. If a particular interrupt source is trying to set a bit within interrupt flag register INTFLG 707 simultaneously as a software write operation attempts to clear it, logic causes the bit to be set.

The ETRAP interrupt flag (bit 0 of interrupt flag register INTFLG 707) is set from either analysis logic or an ETRAP instruction. This interrupt is normally serviced immediately because it cannot be disabled, however interrupt servicing does wait until pipeline stall conditions such as memory contention via crossbar 50 are resolved. The ENTRAP interrupt flag is the only interrupt bit in interrupt flag register INTFLG 707 cleared by hardware when the interrupt is serviced.

The XY PATCH interrupt flag (bit 11 of interrupt flag register INTFLG 707) is set under certain conditions when employing the global address unit 610 and local Address unit 620 combine to perform XY addressing. As previously described in conjunction with FIG. 27 and the description of address unit 120, XY patched addressing may generate interrupts on certain conditions. The instruction word calling for XY patched addressing indicates whether such an interrupt may be generated and whether a permitted interrupt is made on an address inside or outside a designated patch.

The TASK interrupt flag (bit 14 in interrupt flag register INTFLG 707) is set upon receipt of a command word from master processor 60. This interrupt causes digital image/graphics processor 71 to load its TASK interrupt vector. This interrupt may cause a selected digital image/graphics processor 71, 72, 73 or 74 to switch tasks under control of master processor 70, for instance.

The packet request busy interrupt flag PRB (bit 17 of interrupt flag register INTFLG 707) is set if software writes a "1" to the packet request bit of communications register COMM 781 when the queue active bit is a "1". This allows packet requests to be submitted without checking that the previous one has finished. If the previous packet request is still queued then this interrupt flag becomes set. This will be further explained below in conjunction with a description of communications register COMM 781.

The packet request error interrupt flag PRERR (bit 18 of interrupt flag register INTFLG 707) is set if transfer controller 80 encounters an error condition while executing a packet request submitted by the digital image/graphics processor.

The packet request end interrupt flag PREND (bit 19 of interrupt flag register INTFLG 707) is set by transfer controller 80 when it encounters the end of the digital image/graphics processor's linked-list, or when it completes a packet request that instructs transfer controller 80 to interrupt the requesting digital image/graphics processor upon completion.

The master processor message interrupt flag MPMSG (bit 20 of interrupt flag register INTFLG 707) becomes set when master processor 60 sends a message-interrupt to that digital image/graphics processor.

Bits 27–24 of interrupt flag register INTFLG 707 log message interrupts from digital image/graphics processors 71, 72, 73 and 74. Note that a digital image/graphics processor 71, 72, 73 or 74 can send a message to itself and interrupt itself via the corresponding bit of interrupt flag register INTFLG 707. The digital image/graphics processor 0 message interrupt flag DIGPOMSG (bit 24 of interrupt flag register INTFLG 707) is set when digital image/graphics processor 71 sends a message interrupt to the digital image/graphics processor. In a similar fashion, digital image/graphics processor 1 message interrupt flag DIGP1MSG (bit 25 of interrupt flag register INTFLG 707) is set when digital image/graphics processor 72 sends a message interrupt; digital image/graphics processor 2 message interrupt flag DIGP2MSG (bit 26 of interrupt flag register INTFLG 707) is set when digital image/graphics processor 73 sends a message interrupt, and digital image/graphics processor 3 message interrupt flag DIGP3MSG (bit 27 of interrupt flag register INTFLG 707) is set when digital image/graphics processor 74 sends a message interrupt. As previously stated, bits 31–28 of interrupt flag register INTFLG 707 are reserved for message interrupts from four additional digital image/graphics processors in an implementation of multiprocessor integrated circuit 100 including eight digital image/graphics processors.

When an enabled interrupt occurs, an interrupt pseudo-instruction unit 770, which may be a small state machine, injects the following a set of pseudo-instructions into the pipeline at instruction register-address stage 751:

*(A14-=16)=SR

*(A14+12)=PC

BR=*vectadd; Two LS bits of vectadd="11", to load S, G and L

*(A14+8)=IPA

*(A14+4)=IPE

These pseudo-instructions are referred to as PS1, PS2, PS3, PS4 and PS5, respectively. instruction pointer-return from subroutine IPRS 704 is not saved by this sequence. If an interrupt service routine performs any branches then instruction pointer-return from subroutine IPRS 704 should first be pushed by the interrupt service routine, and then restored before returning. Note that the vector fetch is a load of the entire program counter PC 701, with instruction pointer-return from subroutine IPRS 704 protected. Since this causes the S, G and L bits of program counter PC 701 to be loaded, the three least significant bits of all interrupt vectors are made "0". One exception to this statement is that the task vector fetched after a reset should have the "L" bit (bit 0 of program counter PC 701) set, in order to disable looping.

The respective addresses of starting points of interrupt service routines for any interrupt represented in the interrupt flag register INTFLG 707 are called the digital image/graphics processor interrupt vectors. These addresses are generated by software and loaded as data to the parameter memory 25, 30, 35 and 40 corresponding to the respective interrupted digital image/graphics processor 71, 72, 73 and 74 at the fixed addresses shown in Table 34. interrupt pseudo-instruction PS3 takes the 32 bit address stored in the indicated address in the corresponding parameter memory 25, 30, 35 or 40 and stored this in program counter PC 701. interrupt pseudo-instruction unit 770 computes the addresses for the corresponding parameter memory based upon the highest priority interrupt enabled via interrupt enable register 706. Interrupt pseudo-instruction unit 770 operates to include the digital image/graphics processor number from communications register COMM 781 in order to generate unique addresses for each digital image/graphics

processor. Note interrupt pseudo-instruction PS4 and PS5 are in the delay slots following this branch to the interrupt service routine.

TABLE 34

INTFLG bit	Interrupt Name	Address
31	Reserved for DIGP7 Message	0100#1FC
30	Reserved for DIGP6 Message	0100#1F8
29	Reserved for DIGP5 Message	0100#1F4
28	Reserved for DIGP4 Message	0100#1F0
27	DIGP3 Message	0100#1EC
26	DIGP2 Message	0100#1E8
25	DIGP1 Message	0100#1E4
24	DIGP0 Message	0100#1E0
23	Spare	0100#1DC
22	Spare	0100#1D8
21	Spare	0100#1D4
20	Master Processor Message	0100#1D0
19	Packet Request Successful	0100#1CC
18	Packet Request Error	0100#1C8
17	Packet Request Busy	0100#1C4
16	Spare	0100#1C0
15	Spare	0100#1BC
14	TASK interrupt	0100#1B8
13	Spare	0100#1B4
12	Spare	0100#1B0
11	XY Patching	0100#1AC
10	Reserved	0100#1A8
9	Reserved	0100#1A4
8	Reserved	0100#1A0
7	Reserved	0100#19C
6	Reserved	0100#198
5	Reserved	0100#194
4	Reserved	0100#190
3	Reserved	0100#18C
2	Spare	0100#188
1	Spare	0100#184
0	Emulation	0100#180

In each address the "#" is replaced by the digital image/graphics processor number obtained from communications register COMM **781**.

The final 4 instructions of an interrupt service routine should contain the following (32 bit data, unshifted-index) operations:

SR=*(A14++=4)

BR=*(A14++=7)

BR=*(A14++=5)

BR=*(A14++=5)

These instructions are referred to as RETI1, RETI2, RETI3 and RETI4, respectively, Other operations can be coded in parallel with these if desired, but none of these operations should modify status register **211**.

The interrupt state can be saved if a new task is to be executed on the digital image/graphics processor, and then restored to the original state after finishing the new task. The write mode controlled by the "W" bit on interrupt enable register INTEN **706** allows this to be done without missing any interrupts during the saving or restoring operations. This may be achieved by the following instruction sequence. First, disable interrupts via a DINT instruction. Next save both interrupt enable register INTEN **706** and interrupt flag register INTFLG **707**. Set the "W" bit (bit **0**) of interrupt enable register INTEN **706** to "0" and then write Hex "FFFFFFFF" to interrupt flag register INTFLG **707**. Run the new task, which may include enabling interrupts. Following completion of the new task, recover the original task. First, disable interrupts via the DINT instruction. Set the "W" bit of interrupt enable register INTEN **706** to "1". Restore the status of interrupt flag register INTFLG **707** from memory. Next, restore the status of interrupt enable register INTEN from memory. Last, enable interrupts via the EINT instruction.

Each digital image/graphics processor **71**, **72**, **73** and **74** may transmit command words to other digital image/graphics processors and to master processor **60**. A register to register move with a destination of register A15, the zero value address register of the global address unit, initiates a command word transfer to a designated processor. Note that this register to register transfer can be combined in a single instruction with operations of data unit **110** and an access via local data port **144**, as will be described below. This command word is transmitted to crossbar **50** via global data port **148** accompanied by a special command word signal. This allows master processor **60** and digital image/graphics processors **71**, **72**, **73** and **74** to communicate with the other processors of multiprocessor integrated circuit **100**.

FIG. **41** illustrates schematically the field definitions of these command words. In the preferred embodiment command words have the same 32 bit length as data transmitted via global data port **148**. The least significant bits of each command word define the one or more processors and other circuits to which the command word is addressed. Each recipient circuit responds to a received command word only if these bits indicate the command word is directed to that circuit. Bits **3–0** of each command word designate digital image/graphics processors **74**, **73**, **72** and **71**, respectively. Bits **7–4** are not used in the preferred embodiment, but are reserved for use in a multiprocessor integrated circuit **100** having eight digital image/graphics processors. Bit **8** indicates the command word is addressed to master processor **60**. Bit **9** indicates the command word is directed to transfer controller **80**. Bit **10** indicates the command word is directed to frame controller **90**. Note that not all circuits are permitted to send all command words to all other circuits. For example, system level command words cannot be sent from a digital image/graphics processor to another digital image/graphics processor or to master processor **60**. Only master processor **60** can send command words to transfer controller **80** or to frame controller **90**. The limitations on which circuit can send which command words to which other circuits will be explained below in conjunction with the description of each command word field.

The "R" bit (bit **31**) of the command word is a reset bit. Master processor **60** may issue this command word to any digital image/graphics processor, or a digital image/graphics processor may issue this command word to itself. No digital image/graphics processor may reset another digital image/graphics processor. Note throughout the following description of the reset sequence each digit "#" within an address should be replaced with the digital image/graphics processor number, which is stored in bits **1–0** of command register COMM **781**. When a designated digital image/graphics processor receives a reset command word, it first sets its halt latch and sends a reset request signal to transfer controller **80**. Transfer controller **80** sends a reset acknowledge signal to the digital image/graphics processor. The resetting digital image/graphics processor performs no further action until 'receipt of this reset acknowledge signal from transfer processor **80**. Upon receipt of the reset acknowledge signal, the digital image/graphics processor initiates the following sequence of operations: sets the halt latch if not already set; clears to "0" the "F", "P", "Q" and "S" bits of communications register COMM **781** (the use of these bits will be

described below); clears any pending memory accesses by address unit **120**; resets any instruction cache service requests; loads into instruction register-execute stage IRE **752** the instruction

BR={u,ncvz}A14<<1 ||A14=Hex "0100#7F0"

which unconditionally loads the contents of the stack pointer **A14** left shifted one bit to program counter PC **701** with the negative, carry, overflow and zero status bits protected from change and with the "R" bit set to reset stack pointer **A14** in parallel with a load of the stack pointer **A14**; loads into instruction register-address stage IRA **751** the instruction

**(PBA+Hex "FC")=PC*

which instruction stores the contents of program counter PC **701** at the address indicated by the sum of the address PBA and Hex "FC"; sets interrupt pseudo-instruction unit **770** to next load interrupt pseudo-instruction PS3; sets bit **14** of interrupt flag register INTFLG **707** indicating a task interrupt; clears bit **0** of interrupt flag register INTFLG **707** thus clearing the emulator trap interrupt ETRAP; and clears bits **11**, **7** and **3** of loop control register LCTL thus disabling all three loops.

Execution by the digital image/graphics processor begins when master processor **60** transmits an unhalt command word. Once execution begins the digital image/graphics processor: save address stored in program counter PC **701** to address Hex "0100#7FC", this saves the prior contents of stack pointer **A14** left-shifted by one place and the current value of the control bits (bits **2–0**) of program counter PC **701**; loads the address Hex "0100#7F0" into stack pointer **A14**; loads program counter PC **701** with the task interrupt vector, where control bits **2–0** are "000"; stores the contents of instruction register-address stage IPA **751** including control bits **2–0** at address Hex "0100#7F8"; stores the contents of instruction register-execute stage IPE including control bits **2–0** at address Hex "0100#7F4"; and begins program execution at the address given by the Task interrupt. The stack-state following reset is shown in Table 35.

TABLE 35

Address	Contents
Hex "0100#7FC"	stack pointer register A14 from before reset left shifted one place
Hex "0100#7F8"	instruction register-address stage IRA from before reset
Hex "0100#7F4"	instruction register-execute stage IRE from before reset

The prior states of instruction register-address stage IRA**751** and instruction register-execute stage IRE **752** include the control bits **2–0**. Note that stack pointer **A14** now contains the address Hex "0100#7F0".

The "H" bit (bit **30**) of the command word is a halt bit. Master processor **60** may issue this command word to any digital image/graphics processor, or a digital image/graphics processor may issue this command word to itself. No digital image/graphics processor may halt another digital image/graphics processor. When a designated digital image/graphics processor receives this command word, the digital image/graphics processor sets a halt latch and stalls the pipeline. The digital image/graphics processor after that behaves as if in an infinite crossbar memory contention. Nothing is reset and no interrupts occur or are recognized. Note that when a digital image/graphics processor halts itself by sending a command word, the two instructions following the instruction sending the halt command word are in its instruction pipeline. Note that the address pipeline stage of the first instruction following an instruction issuing a halt command word will have already executed its address pipeline stage due to the nature of the instruction pipeline. This halt state can only be reversed by receiving an unhalt command word from master processor **60**.

The Halt condition reduces power consumption within the digital image/graphics processor because its state is unchanging. Further reduced power may be achieved by stopping the clocks while the digital image/graphics processor is in this mode.

The "U" bit (bit **29**) of the command word is an unhalt bit. This command word can only be issued by master processor **60** to one or more of digital image/graphics processors **71**, **72**, **73** and **74**. An unhalt command word clears halt latch of the destination digital image/graphics processor. The digital image/graphics processor then recommences code execution following a halt as if nothing had happened. This is the preferable way to start a digital image/graphics processor following a hardware or command word reset. Upon execution of an unhalt command word, the destination digital image/graphics processor begins code execution at the address given by its task interrupt vector. The "U" bit takes priority over the "H" bit of a single command word. Thus receipt of a single command word with both the "H" bit and the "U" bit set results in execution of the unhalt command. Note that simultaneously receipt of an unhalt command word from master processor **60** and a halt command word transmitted by the digital image/graphics processor itself grants priority to the master processor **60** unhalt command word. The "R" bit takes priority over the "U" bit. Thus receipt of a single command word from master processor **60** having both the "R" bit and the "U" bit set results in the digital image/graphics processor reset to the halted condition.

The "I" bit (bit **28**) of the command word is an instruction cache flush bit. Master processor **60** may issue this command word to any digital image/graphics processor, or a digital image/graphics processor may issue such a command word to itself. No digital image/graphics processor may order an instruction cache flush by another digital image/graphics processor. A designated digital image/graphics processor receiving this command word flushes its instruction cache. An instruction cache flush causes the cache tag value field to be set to the cache tag register's own number, clears all their present bits, and sets the LRU bits to the tag register's own number.

The "D" bit (bit **27**) of the command word indicates a data cache flush. Digital image/graphics processors **71**, **72**, **73** and **74** do not employ data caches, therefore this command word does not apply to digital image/graphics processors and is ignored by them. Master processor **60** may send this command word to itself to flush its data cache memories **13** and **14**.

The "K" bit (bit **14**) of the command word indicates a task interrupt. Master processor **60** may send this command word to any digital image/graphics processor **71**, **72**, **73** or **74**, but no digital image/graphics processor may send this command word to another digital image/graphics processor or to master processor **60**. Upon receipt of a task command word, any digital image/graphics processor designated in the command word takes a task interrupt if enabled by bit **14** of interrupt enable register INTEN **706**.

The "G" bit (bit **13**) of the command word indicates a message interrupt. Any digital image/graphics processor may send this message interrupt to any other digital image/

graphics processor or to master processor **60**. Any digital image/graphics processor designated in such a command word will set its message interrupt flag, and take a message interrupt if message interrupts are enabled via bit **20** of interrupt enable register INTEN **706**. In the preferred embodiment this command word is not sent to transfer controller **80**.

When a digital image/graphics processor issues a command word to itself, to halt itself via the "H" bit or flush its instruction cache via the "I" bit, this command word should have the corresponding digital image/graphics processor designator bit set, to execute the command. This is for consistency, and to allow future expansion of command word functions.

FIG. **42** illustrates schematically the field definitions of communications register COMM **781**. The "F", "S", "Q" and "P" bits (bits **31–28**) are employed in communication of packet requests from a digital image/graphics processor **71**, **72**, **73** or **74** and transfer controller **80**. The "F" and "S" bits are normal read/write bits. The "P" bit may be written to only if the "S" bit is "0" or is being simultaneously cleared to "0". The "Q" bit is read only. Packet requests are requests by a digital image/graphics processor **71**, **72**, **73** or **74** for data movement by transfer controller **80**. These data movements may involve only memories **11–14** and **21–40** internal to multiprocessor integrated circuit **100** or may involve both internal memory and external memory. Packet requests are stored as a linked-list structure and only a single packet request may be active at a time for each digital image/graphics processor. A linked-list pointer at a dedicated address within the parameter memory **25**, **30**, **35** or **40** corresponding to the requesting digital image/graphics processor **71**, **72**, **73** or **74** points to the beginning of the active linked-list. Each entry in the linked-list contains a pointer to the next list entry.

Initializing a packet request involves the following steps. First, the digital image/graphics processor sets the desired packet request parameters into its corresponding parameter memory. Next, the digital image/graphics processor stores the address of the first link of the linked-list at the predetermined address Hex "0100#0FC" in its corresponding parameter memory, where "#" is replaced with the digital image/graphics processor number. Setting the "P" bit (bit **28**) of communications register COMM **781** to "1" alerts transfer controller **80** of the packet request. The digital image/graphics processor may request a high priority by setting the "F" bit (bit **31**) to "1" or a low priority by clearing the "F" bit "0".

Transfer controller **80** recognizes when the "P" bit is set and assigns a priority to the packet request based upon the state of the "F" bit. Transfer controller **80** clears the "P" bit and sets the "Q" bit, indicating that a packet request is in queue. Transfer controller **80** then accesses the predetermined address Hex "0100#0FC" within the corresponding parameter memory and services the packet request based upon the linked-list. Upon completion of the packet request, transfer controller **80** clears the "Q" bit to "0" indicating that the queue is no longer active. The digital image/graphics processor may periodically read this bit for an indication that the packet request is complete. Alternatively, the packet request itself may instruct transfer controller **80** to interrupt the requesting digital image/graphics processor when the packet request is complete. In this case, transfer controller **80** sends an interrupt to the digital image/graphics processor by setting bit **19**, the packet request end interrupt bit PREND, in interrupt flag register INTFLG **707**. If transfer controller **80** encounters an error in servicing the packet request, it sends an interrupt to the digital image/graphics processor by setting bit **18**, the packet request error interrupt bit PRERROR, in interrupt flag register INTFLG **707**. The digital image/graphics processor has the appropriate interrupt vectors stored at the locations noted in Table 34 and the appropriate interrupt service routines.

The digital image/graphics processor may request another packet while transfer controller **80** is servicing a prior request. In this event the digital image/graphics processor sets the "P" bit to "1" while the "Q" bit is "1". If this occurs, transfer controller **80** sends a packet request busy interrupt PRB to the digital image/graphics processor by setting bit **17** of interrupt flag register INTFLG **707**. Transfer controller **80** then clears the "P" bit to "0". The interrupt service routine of requesting digital image/graphics processor may suspend the second packet request while the first packet request is in queue, cancel the packet request or take some other corrective action. This feature permits the digital image/graphics processor to submit packet requests without first checking the "Q" bit of communications register COMM **781**.

The digital image/graphics processor may suspend service of the packet request by setting the "S" bit to "1". Transfer controller **80** detects when the "S" bit is "1". If this occurs while a packet request is in queue, the transfer controller copies the "Q" bit into the "P" bit and clears the "Q" bit. This will generally set the "P" bit to "1". Software within the requesting digital image/graphics processor may then change the status of the "S" and "P" bits. Transfer controller **80** retains in memory its location within the linked-list of the suspended packet request. If transfer controller **80** determines that the "S" bit is "0" and the "P" bit is simultaneously "1", then the suspended packet request is resumed.

The "Sync bits" field (bits **15–8**) of communications register COMM **781** are used in a synchronized multiple instruction, multiple data mode. This operates for any instructions bounded by a lock instruction LCK, which enables the synchronized multiple instruction, multiple data mode, and an unlock instruction UNLCK, which disables this mode. Bits **11–8** indicate whether instruction fetching is to be synchronized with digital image/graphics processors **74**, **73**, **72** and **71**, respectively. A "1" in any of these bits indicates the digital image/graphics processor delays instruction fetch until the corresponding digital image/graphics processor indicates it has completed execution of the prior instruction. The other digital image/graphics processors to which this digital image/graphics processor is to be synchronized will similarly have set the corresponding bits in their communication register COMM **781**. It is not necessary that the "Sync bit" corresponding to itself be set when a digital image/graphics processor is in the synchronized multiple instruction, multiple data mode, but this does no harm. Note that bits **15–12** are reserved for a possible extension to eight digital image/graphics processors.

The "DIGP#" field (bits **2–0**) of communications register COMM **781** are unique to each particular digital image/graphics processor on multiprocessor integrated circuit **100**. These bits are read only, and any attempt to write to these bits fails. This is the only part of the digital image/graphics processors **71**, **72**, **73** and **74** that is not identical. Bits **1–0** are hardwired to a two bit code that identifies the particular digital image/graphics processor as shown in Table 36.

TABLE 36

COMM field		Parallel
1	0	Processor
0	0	DIGP0 (71)
0	1	DIGP1 (72)
1	0	DIGP2 (73)
1	1	DIGP3 (74)

Note that bit 2 is reserved for future use in a multiprocessor integrated circuit **100** having eight image/graphics processors. In the current preferred embodiment this bit is hardwired to "0" for all four digital image/graphics processors **71**, **72**, **73** and **74**.

This part of communications register COME **781** serves to identify the particular digital image/graphics processor. The identity number of a digital image/graphics processor may be extracted by ANDing communications register COMM **781** with 7 (Hex "0000007"). The instruction "D0=COMM&7" does this, for example. This instruction returns only the data in bits **2–0** of communications register COMM **781**. Note that this instruction is suitable for embodiments having eight digital image/graphics processors. Since the addresses of the data memories and parameter memories corresponding to each digital image/graphics processor depend on the identity of that digital image/graphics processor, the identity number permits software to compute the addresses for these corresponding memories. Using this identity number makes it is possible to write software that is independent of the particular digital image/graphics processor executing the program. Note that digital image/graphics processor independent programs may also use registers PBA and DBA for the corresponding parameter memory base address and data memory base address.

Table 37 lists the coding of registers called the lower 64 registers. instruction words refer to registers by a combination of register bank and register number. If no register bank designation is permitted in that instruction word format, then the register number refers to one of the data registers **200** D7–D0. Some instruction words include 3 bit register bank fields. For those instructions words the register is limited to the lower 64 registers listed in Table 37, with a leading "0" implied in the designated register bank. Otherwise, the instruction word refers to a register by a four bit register bank and a three bit register number.

TABLE 37

Reg. Bank	Reg. No.	Register Name	Reg. Bank	Reg. No.	Register Name
0000	000	A0	0100	000	D0
0000	001	A1	0100	001	D1
0000	010	A2	0100	010	D2
0000	011	A3	0100	011	D3
0000	100	reserved	0100	100	D4
0000	101	reserved	0100	101	D5
0000	110	A6	0100	110	D6
0000	111	A7	0100	111	D7
0001	000	A8	0101	000	ROT
0001	001	A9	0101	001	SR
0001	010	A10	0101	010	MF
0001	011	A11	0101	011	reserved
0001	100	reserved	0101	100	reserved
0001	101	reserved	0101	101	reserved
0001	110	A14	0101	110	reserved
0001	111	A15	0101	111	reserved

TABLE 37-continued

Reg. Bank	Reg. No.	Register Name	Reg. Bank	Reg. No.	Register Name
0010	000	X0	0110	000	GLMUX
0010	001	X1	0110	001	reserved
0010	010	X2	0110	010	reserved
0010	011	X3	0110	011	reserved
0010	100	reserved	0110	100	reserved
0010	101	reserved	0110	101	reserved
0010	110	reserved	0110	110	reserved
0010	111	reserved	0110	111	reserved
0011	000	X8	0111	000	PC/CALL
0011	001	X9	0111	001	IPA/BR
0011	010	X10	0111	010	IPE
0011	011	X11	0111	011	IPRS
0011	100	reserved	0111	100	INTEN
0011	101	reserved	0111	101	INTFLG
0011	110	reserved	0111	110	COMM
0011	111	reserved	0111	111	LCTL

Registers A**0** through A15 are address unit base address registers **611**. Registers X**0** through X15 are address unit index address registers **612**. Registers DO through D7 are data unit data registers **200**. Register ROT is the rotation data register **208**. Register SR is the data unit status register **210**. Register MF is the data unit multiple flags register **211**. Register GLMUX is the address unit global/local address multiplex register **630**. Register PC is the program flow control unit **130** program counter PC **701** that points to the instruction being fetched. Reading from this register address obtains the address of the next instruction to be fetched. Writing to this register address causes a software call (CALL). This changes the next instruction pointed to by program counter PC **701** and loads the previous contents of program counter PC **701** into instruction pointer-return from subroutine IPRS **704**. Register IPA is the program flow control unit instruction pointer-address stage **702**, which holds the address of the instruction currently controlling the address pipeline stage. Reading from this register address obtains the address of the instruction currently in the address pipeline stage. Writing to this register address executes a software branch (BR). This alters the address stored in program counter PC **701** without changing the address stored in either instruction pointer-address stage IPA **702** or instruction pointer-return from subroutine IPRS **704**. Register IPE is the program flow control unit instruction pointer-execute stage **703**, which holds the address of the instruction currently controlling the execute pipeline stage. Software would not ordinarily write to either of these two registers. Register IPRS is the program flow control unit instruction pointer-return from subroutine **704**. instruction pointer-return from subroutine IPRS **704** is loaded with the value of program counter PC **701** incremented in bit **3** upon every write to program counter PC **701**. This provides a return address for a subroutine call as the next sequential instruction. Register INTEN is the program flow control unit interrupt enable register **706** that controls the enabling and disabling of various interrupt sources. Register INTFLG is the program flow control unit interrupt flag register **707**. This register contains bits representative of the interrupt sources that are set upon receipt of a corresponding interrupt. Register COMM is the program flow control unit **130** communications register **781**. This register controls packet requests by the digital image/graphics processor to the transfer controller **80**, synchronization between digital image/graphics processors during synchronized MIMD operation and includes hardwired bits identifying the digital image/graphics processor. Register LCTL is the program

flow control unit loop control register **705**, which controls whether hardware loop operations are enabled and which loop counter to decrement.

Table 38 lists the coding of registers called the upper 64 registers. These registers have register banks in the form "1XXX".

TABLE 38

Reg. Bank	Reg. No.	Register Name	Reg. Bank	Reg. No.	Register Name
1000	000	reserved	1100	000	LC0
1000	001	reserved	1100	001	LC1
1000	010	reserved	1100	010	LC2
1000	011	reserved	1100	011	reserved
1000	100	reserved	1100	100	LR0
1000	101	reserved	1100	101	LR1
1000	110	reserved	1100	110	LR3
1000	111	reserved	1100	111	reserved
1001	000	reserved	1101	000	LRSE0
1001	001	reserved	1101	001	LRSE1
1001	010	reserved	1101	010	LRSE2
1001	011	reserved	1101	011	reserved
1001	100	reserved	1101	100	LRS0
1001	101	reserved	1101	101	LRS1
1001	110	reserved	1101	110	LRS2
1001	111	reserved	1101	111	reserved
1010	000	ANACNTL	1110	000	LS0
1010	001	ECOMCNTL	1110	001	LS1
1010	010	ANASTAT	1110	010	LS2
1010	011	EVTCNTR	1110	011	reserved
1010	100	CNTCNTL	1110	100	LE0
1010	101	ECOMCMD	1110	101	LE1
1010	110	ECOMDATA	1110	110	LE2
1010	111	BRK1	1110	111	reserved
1011	000	BRK2	1111	000	CACHE
1011	001	TRACE1	1111	001	GTA
1011	010	TRACE2	1111	010	reserved
1011	011	TRACE3	1111	011	reserved
1011	100	reserved	1111	100	TAG0
1011	101	reserved	1111	101	TAG1
1011	110	reserved	1111	110	TAG2
1011	111	reserved	1111	111	TAG3

In Table 38 the registers ANACNTL, ECOMCNTL, ANASTAT, EVTCNTR, CNTCNTL, ECOMCMD, ECOMDATA, BRK1, BRK2, TRACE1, TRACE2 and TRACE3 are used with an on chip emulation technique. These registers form no part of the present invention and will not be further described. The registers LC0, LC1 and LC2 are loop count registers **733**, **732** and **731**, respectively, within the program flow control unit **130** that are assigned to store the current loop count for hardware loops. The registers LR0, LR1 and LR2 are program flow control unit **130** loop reload registers **743**, **742** and **741**, respectively. These registers store reload values for the corresponding loop count registers LC0, LC1 and LC2 permitting nested loops. The register addresses corresponding to LRSE0, LRSE1, LRSE2, LRS0, LRS1 and LRS2 are write only addresses used for fast loop initialization. Any attempt to read from these register addresses returns null data. Writing a count into one of registers LRS0, LRS1 or LRS2 writes the same count into corresponding loop count register and loop reload register; writes the address stored in program counter PC **701** incremented in bit **3** into the corresponding loop start address register; and writes to loop control register LCTL **705** to enable the corresponding hardware loop. These registers enable fast initialization of a multi-instruction loop. Writing a count into one of registers LRSE0, LRSE1 or LRSE2: writes the same count into corresponding loop count register and loop reload register; writes the address stored in program counter PC **701** incremented in bit **3** into the corresponding loop start address register and loop end address register; and writes to loop control register LCTL **705** to enable the corresponding hardware loop. These registers enable fast initialization of a loop of a single instruction. The registers LS0, LS1 and LS2 are loop start address registers **723**, **722** and **721**, respectively, for corresponding hardware loops. The registers LE0, LE1 and LE2 are loop end address registers **713**, **712** and **711**, respectively, for corresponding hardware loops. Register CACHE is register **709** that mirrors the digital image/graphics processor instruction cache coding. Register GTA is the global temporary register **108** that stores the results of the global address unit operation for later reuse upon contention or pipeline stall. This register is read only and an attempt to write to this register is ignored. Registers TAG3, TAG2, TAG1 and TAG0 are cache tag registers designated collectively as **708**, which store the relevant address portions of data within the data cache memory corresponding to that digital image/graphics processor.

FIG. **43** illustrates the format of the instruction word for digital image/graphics processors **71**, **72**, **73** and **74**. The instruction word has 64 bits, which are generally divided into two parallel sections as illustrated in FIG. **43**. The most significant 25 bits of the instruction word (bits **63–39**) specify the type of operation performed by data unit **110**. The least significant 39 bits of the instruction word (bits **38–0**) specify data transfers performed in parallel with the operation of data unit **110**. There are five formats A, B, C, D and E for operation of data unit **110**. There are ten types of data transfer formats 1 to 10. The instruction word may specify a 32 bit immediate value as an alternative to specifying data transfers. The instruction word is not divided into the two sections noted above when specifying a 32 bit immediate value, this being the exception to the general rule. Many instructions perform operations that do not use data unit **110**. These instructions may allow parallel data transfer operations or parallel data transfer operations may be prohibited depending on the instruction. In other respects the operations specified for data unit **110** are independent of the operations specified for data transfer.

The instruction word alternatives are summarized as follows. The operation of data unit **110** may be a single arithmetic logic unit operation or a single multiply operation, or one of each can be performed in parallel. All operations of data unit **110** may be made conditional based upon a field in the instruction word. The parallel data transfers are performed on local port **141** and global port **145** of data port unit **140** to and/or from memory. Two data transfer operations are independently specified within the instruction word. Twelve addressing modes are supported for each memory access, with a choice of register or offset index. An internal register to register transfer within data unit ! **10** can be specified in the instruction word instead of a memory access via global port **145**. When an operation of data unit **110** uses a non-data unit register as a source or destination, then some of the parallel data transfer section of the instruction word specifies additional register information, and the global port source data bus Gsrc **105** and global port destination data bus Gdst **107** transfer the data to and from data unit **110**.

A part of the instruction word that normally specifies the local bus data transfer has an alternative use. This alternative use allows conditional data unit **110** operation and/or global memory access or a register to register move. Limited conditional source selection is supported in the operation of data unit **110**. The result of data unit **110** can be conditionally saved or discarded, advantageously conditionally performing an operation without having to branch. Update of each

individual bit of a status register can also be conditionally selected. Conditional stores to memory choose between two registers. Conditional loads from memory either load or discard the data. Conditional register to register moves either write to the destination, or discard the data.

Description of the types of instruction words of FIG. 43 and an explanation or glossary of various bits and fields of the five data unit operation formats follows. The bits and fields define not only the instruction words but also the circuitry that decodes the instruction words according to the specified logic relationships. This circuitry responds to a particular bit or field or logical combination of the instruction words to perform the particular operation or operations represented. Accordingly, in this art the specification of bits, fields, formats and operations defines important and advantageous features of the preferred embodiment and specifies corresponding logic circuitry to decode or implement the instruction words. This circuitry is straight forwardly implemented from this specification by the skilled worker in a programmable logic array (PLA) or in other circuit forms now known or hereafter devised. A description of the legal operation combinations follows the description of the instruction word format.

Data unit format A is recognized by bit **63**="1" and bit **44**="0". Data unit format A specifies a basic arithmetic logic unit operation with a 5 bit immediate field. The "class" field (bits **62–60**) designates the data routing within data unit **110** with respect to arithmetic logic unit **230**. Table 39 shows the definition of the data routings corresponding to the "class" field for data unit formats A, B and C.

TABLE 39

Class field 62	61	60	Input A	Input B	Input C	maskgen	rotate
0	0	0	src2/im	src1	@MF	—	0
0	0	1	dstc	src1	src2/im	—	D0 (4–0)
0	1	0	dstc	src1	mask	src2/im	0
0	1	1	dstc	src1	mask	src2/im	src/2im
1	0	0	src2/im	src1	mask	D0 (4–0)	D0 (4–0)
1	0	1	src2/im	src1	@MF	—	D0 (4–0)
1	1	0	dstc	src1	src2/im	—	0
1	1	1	src1	Hex "1"	src2/im	—	src2/im

In Table 39 "Input A" is the source selected by Amux **232** for input A bus **241**. The source "src2/im" is either the five bit immediate value of "immed" field (bits **43–39**) in data unit format A, the data register **200** designated by the "src2" field (bits **41–39**) in data unit format B, or the 32 bit immediate value of the "32-bit immediate" field (bits **31–0**) in data unit format C. The source "dstc" is a companion data register **200** to the destination of the arithmetic logic unit **230** result. This companion data register **200** has a register designation with the upper four bits equal to "0110", thereby specifying one of data registers **200**, and a lower three bits specified by the "dst" field (bits **50–48**). Companion registers are used with transfer formats **6** and **10** which use an "Adstbnk" field (bits **21–18**) to specify the register bank of the destination and an "As1bank" (bits **9–6**) to specify the register bank of Input B. This is known as a long distance destination, because the destination is not one of data registers **200**. Thus one source and the destination may have different register banks with the same register numbers. Table 40 shows the companion registers to various other digital image/graphics processor registers based upon the register bank specified in the "Adstbnk" field. Note that with any other transfer formats this source register is the data register **200** having the register number specified by the "dst" field.

TABLE 40

Adstbnk				Companion Data Registers D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	0	A0	A1	A2	A3	A4	—	A6	A7
0	0	0	1	A8	A9	A10	A11	A12	—	A14	A15
0	0	1	0	X0	X1	X2	—	—	—	—	—
0	0	1	1	X8	X9	X10	—	—	—	—	—
0	1	0	0	D0	D1	D2	D3	D4	D5	D6	D7
0	1	0	1	—	SR	MF	—	—	—	—	—
0	1	1	1	CALL	BR	IPS	IPRS	INTEN	INTFLG	COMM	LCTL
1	1	0	0	LC0	LC1	LC2	—	LR0	LR1	LR2	—
1	1	0	1	LRSE0	LRSE1	LRSE2	—	LRS0	LRS1	LRS2	—
1	1	1	0	LS0	LS1	LS2	—	LE0	LE1	LE2	—
1	1	1	1	—	—	—	—	TAG0	TAG1	TAG2	TAG3

In Table 40 "–" indicates a reserved register. Note that Table 40 does not list register banks "0110, "1000", "1001", "1010" or "1011". All the registers in these banks are either reserved or assigned to emulation functions and would not ordinarily be used as long distance destinations.

In Table 39 "Input B" is the source for barrel rotator **235** which supplies input B bus **242**. The "Input B" source designated "src1" is the data register **200** indicated by the "src1" field (bits **47–45**) in data unit formats A and B, or by the register bank of the "s1bank" field (bits **38–36**) and the register number of the "src1" field (bits **48–45**), which may be any of the 64 lower addressable registers within data unit **110** listed in Table 37, in data format C. The "Hex 1" source for "Input B" is the 32 bit constant equal to "1" from buffer **236**. In Table 39 "input C" is the source selected by Cmux **233** for input C bus **243**.

The "input C" source "@MF" is one or more bits from multiple flags register **211** as expanded by expand circuit **238** in accordance with the "Msize" field (bits **5–3**) of status register **210**. See Table 2 for the definition of the "Msize" field of status register **210**. The "src2/im" source has been previously described in conjunction with the "input A" source. The "mask" source is the output of mask generator **239**. In Table 39 "maskgen" is the source selected by Mmux **234** for mask generator **239**. This source may be "src2/im" as previously described or "D0(4–0)", which is the default barrel rotate amount of the "DBR" field (bits **4–0**) of data register **D0**. In Table 39 "rotate" is the source selected by Smux **231** for control of the rotate amount of barrel rotator

235. This source may be "0", which provides no rotate, "D0(4–0)", which is the default barrel rotate amount of the "DBR" field (bits 4–0) of data register D0, or "src2/im" as previously described.

The "ari" bit (bit 59) designates whether arithmetic logic unit 230 of data unit 110 is used for an arithmetic operation or for a Boolean logic operation. If the "ari" bit is "1" then an arithmetic operation occurs, if "0" then a Boolean logic operation occurs.

Data unit format A permits instruction word specification of the operation of arithmetic logic unit 230. The "8-bit ALU code" field (bits 58–51) designates the operation performed by arithmetic logic unit 230. This field designates an arithmetic operation if the "ari" bit is "1". If this is the case then "8-bit ALU code" bits 57, 55, 53 and 51 designate the arithmetic operation according to Table 21 as modified by the "FMOD" field consisting of "8-bit ALU code" bits 58, 56, 54 and 52 according to Table 6. If the "ari" bit is "0", then this is a Boolean operation and the "8-bit ALU code" field translates into function signals F7–F0 according to Table 20. The details of these encodings were described above in conjunction with the description of data unit 110.

Data unit format A designates two sources and a destination for arithmetic logic unit 230. The "dst" field (bits 50–48) designates a register as the destination for arithmetic logic unit 230. The "dst" field may refer to one of data registers 200 by register number or the register number of the "dst" field may be used in conjunction with a register bank to specify a long distance register depending on the transfer format. The "src1" field (bits 47–45) designate a register as the first source for arithmetic logic unit 230. This may be one of data registers 200 or may be used in conjunction with a register bank to specify a long distance register depending on the transfer format. The "immed" field (bits 43–39) designates a 5 bit immediate value used as the second source for arithmetic logic unit 230. In use this 5 bit immediate value is zero extended to 32 bits. The use of register banks will be further discussed below in conjunction with description of the transfer formats.

The storing of the resultant in the destination register occurs only if the condition noted in the "cond." field is true. The "cond." field (bits 35–32) designates the conditions for a conditional operation. Note that this "cond." field falls within the portion of the instruction word generally used for the transfer format. Transfer formats 7, 8, 9 and 10 include this field. Thus conditional storing of the resultant of arithmetic logic unit 230 occurs only when these transfer formats are used. In the preferred embodiment the "cond." field is decoded as shown below in Table 41.

TABLE 41

Condition field bits						
35	34	33	32	Mnemonic	Condition Description	Status bits Compared
0	0	0	0	u	unconditional	—
0	0	0	1	p	positive	~N&~Z
0	0	1	0	ls	lower than or same	~C|Z
0	0	1	1	hi	higher than	C&~Z
0	1	0	0	lt	less than	(N&~V)|(~N&V)
0	1	0	1	le	less than or equal to	(N&~V)|(~N&~V)|Z
0	1	1	0	ge	greater than or equal to	(N&V)|(~N&~V)
0	1	1	1	gt	greater than	(N&V&~Z)|(~N&~V&~Z)
1	0	0	0	hs, c	lower than, carry	C
1	0	0	1	lo, nc	higher than or same, no carry	~C
1	0	1	0	eq, z	equal, zero	Z
1	0	1	1	ne, nz	not equal, not zero	~Z
1	1	0	0	v	overflow	V
1	1	0	1	nv	no overflow	~V
1	1	1	0	n	negative	N
1	1	1	1	nn	non-negative	~N

The conditions are detected with reference to status register 210. As previously described, status register 210 stores several bits related to the condition of the output of arithmetic logic unit 230. These conditions include negative, carry, overflow and zero. The conditional operation of arithmetic logic unit 230 related to status register 210 was detailed above in conjunction with the description of data unit 110.

The data unit format B is recognized by bit 63="1", bit 44="0". Data unit format B specifies a basic arithmetic logic unit operation with a register specified for the second source of arithmetic logic unit 230. The "class" field designates the data routing within data unit 110 as previously described in conjunction with Table 39. The "ari" bit designates whether arithmetic logic unit 230 of data unit 110 is used for an arithmetic operation or for a Boolean logic operation. The "8 bit ALU code" field designates the operation performed by arithmetic logic unit 230 in the manner described above. The "src2" field (bits 41–39) designates one of the data registers 200 as the second source for arithmetic logic unit 230. In data unit format B the second source for arithmetic logic unit 230 is the data register designated in the "src2" field. Some data transfer formats permit designation of banks of registers for the first source and the destination of arithmetic logic unit 230. In other respects data unit format B is the same as data unit format A.

The data unit format C is recognized by bit 63="1", bit 44="1" and bit 43="1". Data unit format C specifies a basic

arithmetic logic unit operation with a 32 bit immediate field. The "class" field designates the data routing within data unit **110** as previously described in conjunction with Table 39. The "ari" bit designates whether arithmetic logic unit **230** of data unit **110** is used for an arithmetic operation or for a Boolean logic operation. The "8 bit ALU code" field designates the operation performed by arithmetic logic unit **230** as described above. The first source is the data register designated by the "src1" field. The second source is the 32 bit immediate value of the "32-bit imm." field (bits **31–0**). This data unit format leaves no room to specify parallel data transfers, so none are permitted. The "dstbank" field (bits **42–39**) designates a bank of registers within data unit **110**. The "dstbank" field is employed with the "dst" field (bits **50–48**) to designate any of 64 registers of data unit **110** listed in Tables 37 and 38 as the destination for arithmetic logic unit **230**. The "s1bnk" field (bits **38–36**) designates a bank of registers within data unit **110**. This designation is limited to a lower half of the registers of data unit **110** and is employed with the "src1" field to designate any of 64 lower half registers in data unit **110** listed in Table 37 as the first source for arithmetic logic unit **230**. Operations can be made conditional based upon the "cond." field (bits **35–32**) in a manner detailed below.

Data unit format D has bit **63**="1", bit **44**="0", the "class" field is "000", bit **59**="1" (which normally selects arithmetic as opposed to Boolean logic operation) and bits **57**, **55**, **53** and **51** of the "8 bit ALU code" are all "0". Data unit format D specifies non-arithmetic logic unit operations. The "operation" field (bits **43–39**) designates a non-arithmetic logic unit operation. In the preferred embodiment this "operation" field is decoded as shown below in Table 42.

TABLE 42

Operation field 43	42	41	40	39	Non-ALU Operation
0	0	0	0	0	no operation
0	0	0	0	1	idle
0	0	0	1	0	enable global interrupts
0	0	0	1	1	disable global interrupts
0	0	1	0	0	lock synchronization of instruction fetching
0	0	1	0	1	unlock synchronization of instruction fetching
0	0	1	1	0	reserved
0	0	1	1	1	rotate D registers right 1
0	1	0	0	0	null
0	1	0	0	1	halt instruction execution
0	1	0	1	0	reserved
0	1	0	1	1	reserved
0	1	1	0	0	go to emulator interrupt
0	1	1	0	1	issue emulator interrupt 1
0	1	1	1	0	issue emulator interrupt 2
0	1	1	1	1	reserved
1	X	X	X	X	reserved

The non-arithmetic logic unit instructions null, halt instruction execution, go to emulator interrupt, issue emulator interrupt 1 and issue emulator interrupt 2 prohibit parallel data transfers. Any parallel data transfers specified in the instruction word are ignored. The other non-arithmetic logic unit instructions permit parallel data transfers.

Data unit format E is recognized by bits **63–61** being "011". Data unit format E specifies parallel arithmetic logic unit and multiply operations. These operations are referred to as "six operand operations" because of the six operands specified in this format. In the preferred embodiment the "operation" field (bits **60–57**) specifies the operations shown below in Table 43. The symbol "‖" indicates that the listed operations occur in parallel within data unit **110**. Note that only 11 of the 16 possible operations are defined.

TABLE 43

Operation field bits 60	59	58	57	Six Operand Operations
0	0	0	0	MPYS ‖ ADD
0	0	0	1	MPYS ‖ SUB
0	0	1	0	MPYS ‖ EALUT
0	0	1	1	MPYS ‖ EALUF
0	1	0	0	MPYU ‖ ADD
0	1	0	1	MPYU ‖ SUB
0	1	1	0	MPYU ‖ EALUT
0	1	1	1	MPYU ‖ EALUF
1	0	0	0	EALU ‖ ROTATE
1	0	0	1	EALU% ‖ ROTATE
1	0	1	0	DIVI
1	0	1	1	reserved
1	1	0	0	reserved
1	1	0	1	reserved
1	1	1	0	reserved
1	1	1	1	reserved

The mnemonics for these operations were defined above. To review: MPYS‖ADD designates a parallel signed multiply and add; MPYS‖SUB designates a parallel signed multiply and subtract; MPYS‖EALUT designates a parallel signed multiply and extended arithmetic logic unit true operation; MPYS‖EALUF designates a parallel signed multiply and extended arithmetic logic unit false operation; MPYU‖ADD designates a parallel unsigned multiply and add; MPYU‖SUB designates a parallel unsigned multiply and subtract; MPYU‖EALUT designates a parallel unsigned multiply and extended arithmetic logic unit true operation; MPYU‖EALUF designates a parallel unsigned multiply and extended arithmetic logic unit false operation; EALU‖ROTATE designates an extended arithmetic logic unit operation with the output of barrel rotator **235** separately stored; EALU %‖ROTATE designates an extended arithmetic logic unit operation employing a mask generated by mask generator **239** with the output of barrel rotator **235** separately stored; and DIVI designates a divide iteration operation used in division. The arithmetic logic unit operation in an MPYx‖EALUT instruction is selected by the "EALU" field (bits **19–26**) of data register **D0**, with the "A" bit (bit **27**) selecting either an arithmetic operation or a logic operation as modified by the "FMOD" field (bits **31–28**). The coding of these fields has been described above. The arithmetic logic unit operation in an MPYx‖EALUF instruction is similarly selected except that the sense of the "EALU" field bits is inverted. The arithmetic logic unit operations for the EALU and EALU % instructions are similarly selected. These operations employ part of the data register **D0** of data unit **110** to specify the arithmetic logic unit operation. Data register **D0** is pre-loaded with the desired extended arithmetic logic unit operation code. The DIVI operation will be further detailed below. Any data transfer format may be specified in parallel with the operation of data unit **110**.

Six operands are specified in data unit format E. There are four sources and two destinations. The "src3" field (bits **56–54**) designates one of the data registers **200** as the third source. This is the first input for multiplier **220** if a multiply operation is specified, otherwise this is the barrel rotate amount of barrel rotator **235**. The "dst2" field (bits **53–51**) designates one of the data registers **200** as the second

destination. If the instruction specifies a multiply operation, then "dst2" is the destination for multiplier **220**. Otherwise "dst2" specifies the destination for the output of barrel rotator **235**. The "dst1" field (bits **50–48**) designates one of the data registers **200** as the destination for arithmetic logic unit **230**. The "src1" field (bits **47–45**) designates a register as the first input for arithmetic logic unit **230**. If this instruction includes a transfer format **6** or **10**, which include an "Aslbank" field (bits **9–6**), then this register source may be any register within data unit **110** with the "As1bank" field designating the register bank and the "src1" field designating the register number. In such a case this data cannot be rotated by barrel rotator **235**. This is called a long distance arithmetic logic unit operation. For other transfer formats, the "src1" field specifies one of the data registers **200** by register number. Transfer formats **7**, **8**, **9** and **10** permit the register source to be conditionally selected from a pair of data registers **200** based on the "N" bit of status register **210**. If the "N" bit (bit **31**) of status register **211** is "1" then the designated data register is selected as the first source for arithmetic logic unit **230**. If the "N" bit is "0" then the data register one less is selected. If this option is used, then the register number of the "src1" field must be odd. The "src3" field (bits **56–54**) designates one of the data registers **200** as the second input for multiplier **220**. The "src4" field (bits **44–42**) designates one of the data registers **200** as the second input for multiplier **220**.

Table 44 shows the data path connections for some of the operations supported in data unit format E. input C is the signal supplied to input C bus **243** selected by multiplexer Cmux **233**. Maskgen is the signal supplied to mask generator **239** selected by multiplexer Mmux **234**. Rotate is signal supplied to the control input of barrel rotator **235** selected by multiplexer Smux **231**. Product left shift is the signal supplied to the control input of product left shifter **224** supplied to the control input of product left shifter **224** selected by multiply shift multiplexer MSmux **225**. Note that the special case of the DIVI operation will be described later.

TABLE 44

Six Operand Operation	Input C	maskgen	rotate	product left shift
MPYS ‖ ADD	—	—	0	0
MPYS ‖ SUB	—	—	0	0
MPYS ‖ EALUT	mask	D0 (4–0)	D0 (4–0)	D0 (9–8)
MPYS ‖ EALUF	mask	D0 (4–0)	D0 (4–0)	D0 (9–8)
MPYU ‖ ADD	—	—	0	0
MPYU ‖ SUB	—	—	0	0
MPYU ‖ EALUT	mask	D0 (4–0)	D0 (4–0)	D0 (9–8)
MPYU ‖ EALUF	mask	D0 (4–0)	D0 (4–0)	D0 (9—8)
EALU	src4	—	src3	—
EALU%	mask	src4	src1	—

For all the six operand instructions listed in Table 44, the first input to multiplier **220** on bus **201** is the register designated by the "src3" field (bits **56–54**), the second input to multiplier **220** on bus **202** is the register designated by the "src4" field (bits **44–42**), the input to barrel rotator **235** is the register designated by the "src1" field (bits **41–39**) and the input to input A bus **241** is the register designated by the "src2" field (bits **47–45**). Also note that multiplier **220** is not used in the EALU and EALU % instructions, instead the results of barrel rotator **235** are saved in the register designated by the "dst2" field bits **53–51**) via multiplexer Bmux **227**.

The DIVI operation uses arithmetic logic unit **230** and does not use multiplier **220**. The DIVI operation may be used in an inner loop for unsigned division. Signed division may be performed using instructions to handle the sign of the quotient. It is well known in the art that division is the most difficult of the four basic arithmetic operations (addition, subtraction, multiplication and division) to implement in computers.

The DIVI instruction employs the hardware of data unit **110** to compute one digit of the desired quotient per execute pipeline stage, once properly set up. Note that the DIVI data unit instruction can only be used with a data transfer format that supports conditional data transfers (and consequently conditional data unit operations). These data transfer formats **7**, **8**, **9** and **10** will be fully described below. FIG. **44** illustrates in schematic form the data flow within data unit **110** during the DIVI instruction. Refer to FIG. **5** for details of the construction of data unit **110**. Multiplexer Amux **232** selects data from data register **200*b*** designated by the "src2" field on arithmetic logic unit first input bus **205** for supply to arithmetic logic unit **230** via input A bus **241**. Multiplexer Imux **222** selects the constant Hex "1" for supply to multiplier second input bus **202** and multiplexer Smux **231** selects this Hex "1" on multiplier second input bus **202** for supply to rotate bus **244**. Data from one of the data registers **200** designated by the "src1" field supplies barrel rotator **235**. This register can only be data register D7, D5, D3 or D1 and is a conditional register source selected by multiplexer **215** based upon the "N" bit (bit **31**) of status register **210**. If the "N" bit of status register **210** is "0", then data register **200*a*** designated by the "src1" field is selected. This register selection preferably uses the same hardware used to provide conditional register selection in other instructions employing arithmetic logic unit **230**, except with the opposite sense. This register selection may be achieved via a multiplexer, such as multiplexer **215** illustrated in FIG. **44**, or by substituting the inverse of the "N" bit of status register **210** for the least significant bit of the register field during specification of the register. If the "N" bit of status register **210** is "1", then data register **200*c***, which is one less than the register designated by the "src1" field, is selected. Barrel rotator **235** left rotates this data by one bit and supplies the resultant to arithmetic logic unit **230** via input B bus **241**. The output of barrel rotator **235** is also saved to data register **200*a*** via multiplexer Bmux **227**, with bit **31** of multiple flags register **211** (before rotating) substituted for bit **0** of the output of barrel rotator **235**. This destination register is the register designated by the "src1" field. Multiplexer Mmux **234** selects the constant Hex "1" on multiplier second input bus **202** for supply to mask generator **239**. Multiplexer Cmux **233** selects the output from mask generator **239** for supply to arithmetic logic unit **230** via input C bus **243**. Bit **0** carry-in generator **246** supplies bit **31** of multiple flags register **211** (before rotating) to the carry-in input of arithmetic logic unit **230**.

During the DIVI instruction arithmetic logic unit **230** receives a function code F7–F0 of Hex "A6". This causes arithmetic logic unit **230** to add the inputs upon input A bus **241** and input B bus **242** and left shift the result with zero extend. This left shift is by one bit due to the mask supplied by mask generator **239** in response to the Hex "1" input. This function is mnemonically A+B<0<. The resultant of arithmetic logic unit **230** is stored in data register **200*c*** designated by the "dst1" field. Multiple flags register **211** is rotated by one bit, and the least significant bit (bit **0**) of multiple flags register **211** is set according to the resultant produced by arithmetic logic unit **230**. This same bit is stored in the "N" bit (bit **31**) of status register **210**. OR gate **247** forms this bit stored in multiple flags register **211** and status register **210** from c_{out} of arithmetic logic unit **230**

ORed with bit 31 of the input to barrel rotator 235. Note that other status register 210 bits "C", "V" and "Z" are set normally. If the data in data register 200*a* is X, the data in data register 200*b* is Y and the data in data register 200*c* is Z, then the DIVI instruction forms X=X<<1 and Z=X[n]Z+Y. The "n" mnemonic indicates register source selection based upon the "N" status register bit.

The DIVI instruction operates to perform iterations of a conditional subtract and shift division algorithm. This instruction can be used for a 32 bit numerator divided by a 16 bit divisor to produce a 16 bit quotient and a 16 bit remainder or a 64 bit numerator divided by a 32 bit divisor to produce a 32 bit quotient and a 32 bit remainder. In the 64 bit numerator case the 32 most significant bits of the numerator are stored initially in data register 200*a* and the 32 least significant bits are initially stored in multiple flags register 211. Data register 200*b* stores the inverse of the divisor. For the first iteration of a division operation either the DIVI instruction is executed unconditionally or the "N" bit of status register 210 is set to "0". The rotated number from barrel rotator 235 is stored in data register 200*a*. Barrel rotator 235 and the rotation of multiple flags register 211 effectively shift the 64 bit numerator one place. Note that the most significant bit of multiple flags register 211 is the next most significant bit of the 64 bit numerator and is properly supplied to the carry-in input of arithmetic logic unit 230. The quantity stored in data register 200*a* is termed the numerator/running remainder. The result of the trial subtraction is stored in data register 200*c*.

There are two cases for the result of the trial subtraction. If either the most significant bit of the initial numerator was "1" or if the addition of the negative divisor generates a carry, then the corresponding quotient bit is "1". This is stored in the first bit of multiple flags register 211 and in the "N" bit of status register 210. For the next trial subtraction, multiplexer 215 selects data register 200*c* for the B input for the next iteration by virtue of the "1" in the "N" bit of status register 210. Thus the next trial subtraction is taken from the prior result. If OR gate 247 generates a "0", then the corresponding quotient bit is "0". Thus the next trial subtraction is taken from the prior numerator/running remainder stored in data register 200*a* shifted left one place. This iteration continues for 32 cycles of DIVI, forming one bit of the quotient during each cycle. The 32 bit quotient is then fully formed in multiple flags register 211. The 32 bit remainder is found in either data register 200*a* or data register 200*c* depending upon the state of the "N" bit of status register 210.

The process for a 32 bit by 16 bit division is similar. The negated divisor is left shifted 16 places before storing in data register 200*b*. The entire numerator is stored in data register 200*a*. The DIVI instruction is repeated only 16 times, whereupon the quotient is formed in the 16 least significant bits of multiple flags register 211 and the remainder in the 16 most significant bits of either data register 200*a* or data register 200*c* depending on the state of the "N" bit of status register 210.

This technique employs hardware already available in data unit 100 to reduce the overhead of many microprocessor operations. The DIVI instruction essentially forms one bit of an unsigned division. Additional software can be employed to support signed division. Four divide subroutines may be written for the cases of unsigned half word (32 bit/16 bit) divide, unsigned word (64 bit/32 bit) divide, signed half word (32 bit/16 bit) divide, and signed word (64 bit/32 bit) divide. Each of the four subroutines includes three phases: divide preparation; divide iteration in a single instruction loop; and divide wrap-up. It is preferable to employ zero overhead looming and single 64 bit DIVI instruction within the loop kernel.

The first part of each division subroutine is divide preparation. This first includes testing for a divisor of zero. If the divisor is "0", then the division subroutine is aborted and an error condition is noted. Next the sign bits are determined for the numerator and divisor. In the signed division subroutines the sign of the quotient is set as an OR of the sign bits of the numerator and divisor. Then in signed division, if either the numerator or divisor is negative they are negated to obtain a positive number. The numerator is spit between a selected odd data register and the multiple flags register 211. For a word division, the upper 32 bits of the numerator are stored in the selected data register and the lower 32 bits of the numerator are stored in multiple flags register 211. For a half word division all 32 bits of the numerator are stored in the selected data register. For the half word division, the unused lower bits of multiple flags register 211 are zero filled. For half word division the divisor is stored in the upper 16 bits of a data register with the lower bits being zero filled. The divisor should be negated so that arithmetic logic unit 230 can form subtraction by addition. The subroutines may compare the absolute values of the most significant bits of the numerator and denominator to determine if the quotient will overflow.

The heart of each divide subroutine is a loop including a single DIVI instruction. It is very advantageous to write to one of the register addresses LSRE2–LSRE0 to initialize a zero overhead one instruction loop. Sixteen iterations are needed for half word quotients and 32 for word quotients. Since the loop logic 720 decrements to zero, the loop counter should be loaded with one less than the desired number of iterations. It is also possible to place up to two iterations of the DIVI instruction in the delay slots following loop logic initialization. The single instruction within this loop is the DIVI instruction, which has been fully described above.

Each division subroutine is completed with divide wrap-up. Divide wrap-up includes the following steps. The quotient is moved from multiple flags register 211 to a data register. If the sign of the quotient is negative, then "1" is added to the quotient in the data register to convert from "1's" complement representation to twos complement representation. If the remainder is needed it is selected based upon the "N" bit of status register 210.

A further refinement increases the power of the DIVI instruction in each of the divide subroutines when the numerator/running remainder has one or more strings of consecutive "0's". Before beginning the inner loop, the divisor is tested for leading "0's" via LMO/RMO/LMBC/RMBC circuit 237. The input on bus 206 is directed through LMO/RMO/LMBC/RMBC circuit 237 using the "FMOD" field of data register D0 or bits 52, 54, 56 and 56 of the "8-bit ALU code" of an arithmetic instruction word. The data register holding the divisor and the data register holding the high order bits of the numerator/running remainder is left shifted by a number of places equal to this number of leading "0's". In the same fashion, the data in multiple flags register 211 is left shifted, with zeros inserted into lower order bits corresponding to the zeros in the quotient bits. The inner loop includes additional operations in this refinement. One additional operation searches for strings of consecutive "0's" in the numerator/running remainder. The quotient bit for each place where the numerator/running remainder is "0" is also "0". Thus if such strings of consecutive "0's" can be detected, then the DIVI instruction for those places can be

eliminated. This additional operation employs a conditional source register in the same manner as the DIVI instruction. The input on bus **206** is directed through LMO/RMO/LMBC/RMBC circuit **237** using the "FMOD" field. Arithmetic logic unit **230** generates a resultant equal to the data on input C bus **243**, which is the number of "0's" in leading bits of the numerator/running remainder. This result is stored in one of data registers **200** D7–D0 not otherwise used by the subroutine. The loop count stored in the loop count register LC2—LC0 used for the divide iteration loop is decremented by this number of consecutive "0's". The following DIVI employs this count as the shift amount via multiplier second input bus **202**. Multiple flags register **211** is slightly modified to also rotate by this amount and transfer the rotated out most significant bits into the least significant bits of data register **200***a*. The least significant bits of multiple flags register **211** are zero filled during this rotate. Using this instruction skips over consecutive "0's" in the numerator/running remainder, placing "0's" in the corresponding quotient bits and rotating past the consecutive "0's". In instances where the numerator/running remainder has strings of consecutive "0's", this two instruction loop produces the quotient faster than the single instruction loop.

This is illustrated in flow chart form in FIG. **45**. The divide algorithm is begun at In block **1001**. First, decision block **1002** tests for a divisor of 0 and if true the algorithm is exited at divide by zero (/0) exit block **1003**. Next decision block **1004** compares the absolute value of the divisor to the high order bits of the numerator. If the absolute value of the divisor is less than the high order bits of the numerator, then the algorithm is exited at overflow exit block **1005**.

Block **1006** sets the quotient stored in multiple flags register **211** to zero and sets the loop count to 16. Note that this example is of a 32 bit by 16 bit divide. The loop count would be set to 32 for a 64 bit by 32 bit divide. Block **1007** sets two registers by loading the numerator into register A and the divisor into register B. Block **1008** sets V, the sign of the quotient, equal to the exclusive OR of the sign of the numerator and the denominator. Decision block **1009** tests to determine if the sign of the quotient is positive. If so, then block **1010** negates the data in register B, which is the divisor. If not, then register B is not changed. Block **1011** sets n equal to the left most one place of the absolute value of the data in register B. This tests for leading zeros in the division. Block **1012** left shifts the data in register A, the numerator/running remainder, and the data in register B, the divisor, n places.

The division loop begins with block **1013**. Block **1013** sets m equal to the left most one place of the data in register A. Decision block **1014** compares m to the loop count. If m is greater than the loop count, then block **1015** sets m equal to the loop count. Block **1016** left the numerator/running remainder and the quotient m places. Decision block **1017** tests to determine if the previously computed sign of the quotient is positive. If V is positive, then block **1018** sets the quotient Q equal to Q plus number including a string of m number of sign bits, filling the places vacated in block **1016**. Block **1019** decrements the loop count by the left most one place amount m.

Block **1020** performs the trial subtraction of the data in register A, the numerator/running remainder, and the divisor in register B. Note that blocks **1009** and **1010** insure that the data in register B is negative. Decision block **1021** determines if the trial subtraction changes sign. If there is a sign change, then block **1022** sets the least significant bit of the quotient equal to the sign V. If there is no sign change, then block **1023** sets the least significant bit of the quotient equal to the inverse of the sign V and block **1024** sets A equal to the sum C. In either case, block **1025** left shifts register A one place. Note that as described above, the single DIVI instruction performs the actions of blocks **1020** through **1025**.

Blocks **1026** and **1027** handle the loop. Block **1026** decrements the loop count. Block **1027** determines if the loop count is less than zero. If not, then algorithm control returns to block **1013** to repeat the loop. If the loop count is less than zero, then the loop is complete. Preferably the zero-overhead loop logic handles the operations of blocks **1026** and **1027**.

Upon exiting the loop, some clean up steps are needed. Decision block **1028** determines if the quotient is less than zero. If so, then block **1029** adds one to the quotient. This provides the proper conversion from one's complement to two's complement. Block **1030** sets the remainder equal to the high order bits stored in the A register. The algorithm is exited via exit block **1031**.

Note the DIVI instruction advantageously performs several crucial functions in the inner loop. Thus the DIVI instruction is highly useful in this algorithm. Note also, in the absence of such a DIVI instruction, digital image/graphics processor **71** may still perform this algorithm using a determination of the left most ones in accordance with the program illustrated in FIG. **45**.

FIG. **46** illustrates an alternative embodiment of the division algorithm that additionally uses a left most ones determination of the exclusive OR of the data in registers A and B. The initial steps divide by 0 and overflow steps illustrated in FIG. **46** are identical to those illustrated in FIG. **45**. Block **1032** sets register A equal to the absolute value of the numerator and register B equal to the absolute value of the divisor. Block **1008** sets the sign V of the quotient as before.

Block **1011** determines the left most one place b of the absolute value of the divisor. Block **1033** left shifts the data in register B the number of places of the left most one. Block **1034** left shifts register A by b, the number of places of the left shift of register B.

Block **1035** begins the loop. Block **1035** determines the left most one place of the data in register A and sets c equal to 29 minus the left most one place a. Block **1036** sets t equal to the loop count minus c. Decision block **1037** determines if the loop count is less than c. If so, then block **1038** sets c equal to the loop count. Block **1039** left shifts both the data in register A and the quotient c places. Block **1039** also decrements the loop count by c. This step skips over trial subtraction for zeros in the numerator/running remainder.

Block **1040** determines the left most zero place of A^B. Block **1041** determines if the loop count is less than or equal to zero or if x, the left most zero place of A^B, is zero. If not, then both the data in register A and the quotient are left shifted one place and the loop count is decremented by 1.

Block **1043** determines if t, the difference of the loop count and c computed in block **1036**, is less than zero. If so, then the loop is exited. If not, then block **1044** computes the trial subtraction A–B and increments the quotient by 1. Block **1045** determines if the loop count is greater than zero. If so, then the algorithm repeats the loop starting at block **1035**. If not, or if t was less than zero, then the data in register A, now forming the remainder, is right shifted by b places.

The remaining steps involve clean up. Decision block **1047** determines if the sign of the quotient is less than zero. If so, then the quotient is replaced by its inverse. In either event, decisions block **1049** determines if the numerator/

running remainder N is less than zero. If so, then the remainder stored in as the higher order bits in register A is replaces by its inverse. The algorithm is exited via exit block **1031**.

A description of the data transfer formats and an explanation or glossary of various bits and fields of the parallel data transfer formats of instruction words of FIG. **43** follows. As previously described above in conjunction with the glossary of bits and fields of the data unit formats these bits and fields define not only the instruction word but also the circuitry that enable execution of the instruction word.

Transfer format **1** is recognized by bits **38–37** not being "00", bits **30–28** not being "000" and bits **16–15** not being "00". Transfer format **1** is called the double parallel data transfer format. Transfer format **1** permits two independent accesses of memory **20**, a global access and a local access limited to the memory sections corresponding to the digital image/graphics processor. The "Lmode" field (bits **38–35**) refers to a local transfer mode, which specifies how the local address unit of address unit **120** operates. This field is preferably decoded as shown in Table 45.

TABLE 45

Lmode field					
38	37	36	35	Expression Syntax	Operation Description
0	0	X	X		no operation
0	1	0	0	*(An++=Xm)	post-addition of index register with modify
0	1	0	1	*(An—=Xm)	post-subtraction of index register with modify
0	1	1	0	*(An++=Imm)	post-addition of offset with modify
0	1	1	1	*(An—=Imm)	post-subtraction of offset with modify
1	0	0	0	*(An+Xm)	pre-addition of index register
1	0	0	1	*(An-31 Xm)	pre-subtraction of index register
1	0	1	0	*(An+Imm)	pre-addition of offset
1	0	1	1	*(An-Imm)	pre-subtraction of offset
1	1	0	0	*(An+=Xm)	pre-addition of index register with modify
1	1	0	1	*(An-=Xm)	pre-subtraction of index register with modify
1	1	1	0	*(An+=Imm)	pre-addition of offset with modify
1	1	1	1	*(An-=Imm)	pre-subtraction of offset with modify

The "d" field (bits **34–32**) designates one of the data registers **D0–D7** to be the source or destination of a local bus transfer. The "e" bit (bit **31**) if "1" designates sign extend, else if "0" designates zero extend for the local data transfer. This is operative in a memory to register transfer when the local "siz" field (bits **30–29**) indicates less than a full 32 bit word size. This "e" bit is ignored if the data size is 32 bits. The combination of "e" (bit **31**)="1" and "L" (bit **21**)="0", which would otherwise be meaningless, indicates a local address unit arithmetic operation. The local "siz" field (bits **30–29**) is preferably coded as shown in Table 46.

TABLE 46

Size field		
30	29	Data word size
0	0	byte 8 bits
0	1	half word 16 bits

TABLE 46-continued

Size field		
30	29	Data word size
1	0	whole word 32 bits
1	1	reserved

The "s" bit (bit **28**) sets the scaling mode that applies to local address index scaling. If the "s" bit is "1" the index in the address calculation, which may be recalled from an index register or an instruction specified offset, is scaled to the size indicated by the "siz" field. If the "s" bit is "0", then no scaling occurs. As previously described this index scaling takes place in index scaler **614**. If the selected data size is 8 bits (byte), then no scaling takes place regardless of the status of the "s" bit. In this case only, the "s"bit may be used as an additional offset bit. If the "Lmode" field designates an offset then this "s" bit becomes the most significant bit of the offset and converts the 3 bit offset index of the "Lim/x" field to 4 bits. The "La" field (bits **27–25**) designates an address register within local address unit **620** of address unit **120** for a local data transfer. The "L" bit (bit **21**) indicates the local data transfer is a load transferring data form memory to register (L="1") or a store transferring data from register to memory (L="0") transfer. The "Lim/x" field (bits **2–0**) specifies either the register number of an index register or a 3 bit offset depending on the coding of the "Lmode" field.

The global data transfer operation is coded in a fashion similar to the coding of the local data transfer. The "L" bit (bit **17**) is a global load/store select. This bit determines whether the global data transfer is a memory to register ("L"="1") transfer, also known as a load, or a register to memory ("L"="0") transfer, also known as a store. The "Gmode" field (bits **16–13**) defines a global transfer mode in the same way the local transfer mode is defined by the "Lmode" field. This field is preferably decoded as shown in Table 47.

TABLE 47

Gmode field					
16	15	14	13	Expression Syntax	Operation Description
0	0	X	X		no operation
0	1	0	0	(An++=Xm)	post-addition of index register with modify
0	1	0	1	*(An—=Xm)	post-subtraction of index register with modify
0	1	1	0	*(An++=Imm)	post-addition of offset with modify
0	1	1	1	*(An—=Imm)	post-subtraction of offset with modify
1	0	0	0	*(An+Xm)	pre-addition of index register
1	0	0	1	*(An-Xm)	pre-subtraction of index register
1	0	1	0	*(An+Imm)	pre-addition of offset
1	0	1	1	*(An-Imm)	pre-subtraction of offset
1	1	0	0	*(An+=Xm)	pre-addition of index register with modify
1	1	0	1	*(An-=Xm)	pre-subtraction of index register with modify
1	1	1	0	*(An+=Imm)	pre-addition of offset with modify
1	1	1	1	*(An-=Imm)	pre-subtraction of offset with modify

The "reg" field (bits **12–10**) identifies a register. The "reg" field designates the number of the source register in the case of a store, or the number of the destination register in the

case of a load. The "0bank" field (bits **20–18**) contains three bits and identifies a bank of registers in the lower 64 registers. These registers have register bank numbers in the form "0XXX". The 3 bit "0bank" filed combines with the 3 bit "reg" field to designate any register in the lower 64 registers as the data source or destination for the global data transfer. The "e" bit (bit **9**) if "1" designates sign extend, else if "0" designates zero extend for the global data transfer. This is operative in a memory to register transfer when the global "siz" field (bits **8–7**) indicates less than a full 32 bit word size. This "e" bit is ignored if the data size is 32 bits. The combination of "e" (bit **9**)="1" and "L" (bit **17**)="0" indicates a global address unit arithmetic operation. The global "siz" field (bits **8–7**) is preferably coded as shown in Table 48.

TABLE 48

Size field		
8	7	Data word size
0	0	byte 8 bits
0	1	half word 16 bits
1	0	whole word 32 bits
1	1	reserved

The "s" bit (bit **6**) sets the scaling mode that applies to global address index scaling. If the "s" bit is "1" the index in the address calculation, which may be recalled from an index register or an instruction specified offset, is scaled to the size indicated by the "siz" field. If the "s" bit is "0", then no scaling occurs. No scaling takes place regardless of the status of the "s" bit if the "siz" field designates a data size of 8 bits. If the "Gmode" field designates an offset then this "s" bit becomes the most significant bit of the offset and converts the 3 bit offset index of the "Gim/x" field to 4 bits. The "Ga" field (bits **5–3**) designates an address register within global address unit **610** of address unit **120** for a local bus transfer. The "Gim/x" field (bits **24–22**) specifies either the register number of an index register or a 3 bit offset depending on the coding of the "Gmode" field. The "Ga" field (bits **5–3**) specifies the register number of the address register used in computing the memory address of the global data transfer.

Data transfer format **2** is recognized by bits **38–37** not being "00", bits **30–28** being "000" and bits **16–15** not being "00". Data transfer format **2** is called the XY patch format. Data transfer format **2** permits addressing memory **20** in an XY patch manner multiplexing addresses from both the global and local address units of address unit **120**. The "o" bit (bit **34**) enables outside XY patch detection. When "o" bit is set to "1", the operations specified by the bits "a" and "n" are performed if the specified address is outside the XY patch. Otherwise, when "o" bit is "0", the operations are performed if address is inside the patch. The "a" bit (bit **33**) specifies XY patch memory access mode. When the "a" bit is set to "1", the memory access is performed regardless of whether the address is inside or outside the XY patch. When the "a" bit is set to "0", the memory access is inhibited if the address is outside (if the "o" bit is "1") or inside (if the "o" bit is "0") the patch. The "n" bit (bit **32**) specifies XY patch interrupt mode. When the "n" bit is set to "1", an interrupt flag register bit for XY patch is set to "1" if the address is outside (if "o" bit is "1") or inside (if "o" bit is "0") the patch. When "n" bit is set to "0", the XY patch interrupt request flag is not set.

Other fields are defined in the same manner detailed above. The "Lmode" field specifies the local address calculation mode as shown in Table 45. This local address calculation includes a local address register designated by the "La" field and either a 3 bit unsigned offset or a local index register designated by the "Lim/x" field. The "Gmode" field specifies the global address calculation. A global unsigned 3 bit offset or a global index register indicated by the "Gim/X" field is combined with the address register specified by the "Ga" field to form the global address. The 4 bit "bank" field (bits **21–18**) identifies a data register bank and is combined with the 3 bit "reg" field identifying a register number to designate any register as the data source or destination for an XY Patch access. The "L" bit is a load/store select. This bit determines whether an XY Patch access is a memory to register ("L"="1") transfer, also know as a load, or register to memory ("L"="0") transfer, also known as a store. The "e" bit if "1" designates sign extend, else if "0" designates zero extend. This is operative in a load operation (memory to register data transfer) when the "siz" field indicates less than a full 32 bit word size. This "e" bit is ignored if the data size is 32 bits. The combination of "e"="1" with "L"="0" indicates a patched address unit arithmetic operation. The "s" bit sets the scaling mode that applies to global address index scaling. If the "s" bit is "1" the data recalled from memory is scaled to the size indicated by the "siz" field. If the "s" bit is then no scaling occurs. If the selected data size is 8 bits (byte), then no scaling takes place regardless of the status of the "s" bit. In this case only, the "s" bit is used as the most significant bit of the offset converting the 3 bit "Gim/x" offset index to 4 bits.

Data transfer format **3** is recognized by bits **38–37** not being "00", bit **24** being "0" and bits **16–13** being "0000" Data transfer format **3** is called the move and local data transfer format. Data transfer format **3** permits a load or store of one of the data registers **200** via the local data port in parallel with a register to register move using global port source data bus Gsrc **105** and global port destination data bus Gdst **107**. The local data port operation is defined by the fields "Lmode", "d", "e", "siz", "s", "La", "L" and "Lim/x" in the manner described above. The register to register move is from the register defined by the bank indicated by the "srcbank" field (bits **9–6**) and the register number indicated by the "src" field (bits **12–10**) to the register defined by the bank indicated by the "dstbank" field (bits **21–18**) and the register number indicated by the "dst" field (bits **5–3**).

Data transfer format **3** supports digital image/graphics processor relative addressing. The "Lrm" field (bits **23–22**) indicate the type of addressing operation. This is set forth in Table 49.

TABLE 49

Lrm field		
8	7	Addressing Mode
0	0	normal aadressing
0	1	reserved
1	0	Data memory base address DBA
1	1	Parameter memory base address PBA

Specification of DBA causes local address unit **620** to generate the base address of its corresponding memory. Likewise, specification of PBA causes local address generator **620** to generate the base address of the corresponding parameter memory. The base address generated in this manner may be combined with the index stored in an index register or an offset field in any of the address generation operations specified in the "Lmode" field shown in Table 45.

This data transfer format also supports command word generation. If the destination of the register to register move is the zero value address register of the global address unit A15, then the instruction word decoding circuitry initiates a command word transfer to a designated processor. This command word is transmitted to crossbar **50** via the global data port accompanied by a special command word signal. This allows interprocessor communication so that, for example, any of digital image/graphics processors **71**, **72**, **73** and **74** may issue an interrupt to other processors. This process is detailed above.

Data transfer format **4** is recognized by bits **38–37** not being "00", bit **24** being "0" and bits **16–13** being "0001". Data transfer format **4** is called the field move and local data transfer format. Data transfer format **4** permits a load or store of one of the data registers **200** via the local data port in parallel with a register to register field move using global port source data bus Gsrc **105** and global port destination data bus Gdst **107**. The local data port operation is defined by the fields "Lmode", "d", "e" (bit **31**), "siz" (bits **30–29**), "s", "La", "L" and "Lim/x" in the manner described above.

The register to register field move is from the data register defined by the register number indicated by the "src" field (bits **12–10**) to the register defined by the bank indicated by the "dstbank" field (bits **21–18**) and the register number indicated by the "dst" field (bits **5–3**). The "D" bit (bit **6**) indicates if the field move is a field replicate move if "D"="1", or a field extract move if "D"="0". In a field replicate move the least significant 8 bits of the source register are repeated four times in the destination register if the "siz" field (bits **8–7**) indicates a byte size, and the least significant 16 bits of the source register are duplicated in the destination register if the "siz" field (bits **8–7**) indicates a half word size. If the "siz" field indicate a word size, then the whole 32 bits of the source register are transferred to the destination register without replication regardless of the state of the "D" bit. In a field extract move the "itm" field (bits **23–22**) indicates the little endian item number to be extracted from the source register. The particular bits extracted also depends upon the "siz" field. When the data size of the "siz" field (bits **8–7**) is byte, then "itm" may be 0, 1, 2 or 3 indicating the desired byte. When the data size of the "siz" field (bits **8–7**) is half word, then "itm" may be 0 or 1 indicating the desired half word. The "itm" field is ignored if the "siz" field (bits **8–7**) is word. The extracted field from the source register is sign extended if the "e" bit (bit **9**) is "1" and zero extended if the "e" bit (bit **9**) is "0". The "e" field is ignored during field replicate moves.

Data transfer format **5** is recognized by bits **38–37** not being "00", bit **24** being "1" and bits **16–15** being "00". Data transfer format **5** is called local long offset data transfer. Data transfer format **5** permits a global port memory access using an address constructed in the local address unit because no global data transfer is possible. The local data port operation is defined by the fields "Lmode", "d", "e", "siz", "s", "La" and "L" in the manner described above. The register source or destination corresponds to the register number designated in the "reg" field (bits **34–32**) in the bank of registers designated in the "bank" field (bits **21–18**). The "Local Long Offset/x" field (bits **14–0**) specifies a 15 bit local address offset or the three least significant bits specify an index register as set by the "Lmode" field. A programmer might want to use this data transfer format using an index register rather than the "Local long offset" field because data transfer format 5 permits any data unit register as the source for a store or as the destination for a load. The "Lmode" field indicates whether this field contains an offset value or an index register number. If the selected data size is 8 bits (byte), then no scaling takes place regardless of the status of the "s" bit. In this case only, the "s" bit becomes the most significant bit of the offset converting the 15 bit "Local long offset" field into 16 bits. The "Lrm" field (bits **23–22**) specify a normal address operation, a data memory base address operation or a parameter memory base operation as listed above in Table 49.

Data transfer format **6** is recognized by bits **38–37** being "00", bits **16–15** not being "00" and bit **2** being "0". Data transfer format **6** is called global long offset data transfer. Data transfer format **6** is similar to data transfer format **5** except that the address calculation occurs in the global address unit. The fields "bank", "L", "Gmode", "reg", "e", "siz", "s" and "Ga" are as defined above. The "Global Long Offset/x" field (bits **36–22**) specifies a global offset address or an index register depending on the "Gmode" field. This is similar to the "Local Long Offset/x" field discussed above. The "Grm" field (bits **1–0**) indicate the type of addressing operation. This is set forth in Table 50.

TABLE 50

Grm field		
1	0	Addressing Mode
0	0	normal addressing
0	1	reserved
1	0	Data memory base address DBA
1	1	Parameter memory base address PBA

This operates in the same fashion as the "Lrm" field described above except that the address calculation takes place in global address unit **610**.

Data transfer format **7** is recognized by bits **38–37** not being "00", bit **24** being "0" and bits **16–14** being "001". Data transfer format **7** is called the non-data register data unit operation and local data transfer format. Data transfer format **7** permits a local port memory access in parallel with a data unit operation where the first source for arithmetic logic unit **230** and the destination for arithmetic logic unit **230** may be any register on digital image/graphics processor **71**. The local data port operation is defined by the fields "Lmode", "d", "e", "siz", "s", "La", "Lrm", "L" and "Lim/x" in the manner described above. The "Adstbnk" field (bits **21–18**) specifies a bank of registers for the arithmetic logic unit destination. This field specifies a register source in combination with the "dst" field in data unit formats A, B and C, and the "dst1" field in data unit format D. The "As1bank" field specifies a bank of registers for the first arithmetic logic unit source. This specifies a register source in combination with the "src1" field in data unit formats A, B, C and D. These data unit operations are called long distance arithmetic logic unit operations because the first source and the destination need not be the data registers **200** of data unit **110**.

Data transfer format **8** is recognized by bits **38–37** being "00", bit **24** being "0" and bits **16–13** being "0000". Data transfer format **8** is called the conditional data unit operation and conditional move transfer format. Data transfer format **8** permits conditional selection of the first source for arithmetic logic unit **230** and conditional storing of the resultant of arithmetic logic unit **230**. The conditional arithmetic logic unit operations are defined by the fields "cond.", "c", "r", "g" and "N C V Z".

The "cond." field (bits **35–32**) defines an arithmetic logic unit operation from conditional register sources and condi-

tional storage of the arithmetic logic unit resultant. This field is defined in Table 41. These conditions are evaluated based upon the "N", "C", "V" and "Z" bits of status register **210**.

The specified condition may determine a conditional register source, a conditional storage of the result of arithmetic logic unit **230** or a conditional register to register move. The "c" bit (bit **31**) determines conditional source selection. If the "c" bit is "0", then the first source for arithmetic logic unit **230** is unconditionally selected based upon the "src1" field (bits **47–45**) of the data unit format portion of the instruction word. If the "c" bit is "1", then the register source is selected between an odd and even register pair. Note that in this case the "src1" field must specify an odd numbered data register **200**. If the condition is true, then the specified register is selected as the first source for arithmetic logic unit **230**. If the condition is false, then the corresponding even data register one less than the specified data register is selected as the source. The preferred embodiment supports conditional source selection based upon the "N" bit of status register **210**. If the "N" field of status register **210** is "1" then the designated data register is selected as the first source for arithmetic logic unit **230**. If the "N" field of status register **210** is "0", then the data register one less is selected. This selection can be made by a multiplexer, such as multiplexer **215** illustrated in FIG. **44**, or by substitution of the "N" field of status register **210** for the least significant bit of the register number. While the preferred embodiment supports only conditional source selection based upon the "N" bit of status register **210**, it is feasible to provide conditional source selection based upon the "C", "V" and "Z" bits of status register **210**.

Data transfer format **8** supports conditionally storing the resultant of arithmetic logic unit **230**. The "r" bit (bit **30**) indicates if storing the resultant is conditional. If the "r" bit is "1" then storing the resultant is conditional based upon the condition of the "cond." field. If the "r" bit is "0", then storing the resultant is unconditional. Note that in a conditional result operation, the status bits of status register **210** are set unconditionally. Thus these bits may be set even if the result is not stored.

Data transfer format **8** also permits a conditional register to register move operation. The condition is defined by the same "cond." field that specifies conditional data unit operations. The register data source of the move is defined by the bank indicated by the "srcbank" field (bits **9–6**) and the register number indicated by the "src" field (bits **12–10**). The register data destination is defined by the bank indicated by the "dstbank" field (bits **21–18**) and the register number indicated by the "dst" field (bits **5–3**). The "g" bit (bit **29**) indicates if the data move is conditional. If the "g" bit is "1", the data move is conditional based upon the condition specified in the "cond." field. If the "g" bit is "0", the data move is unconditional. Note that a destination of the zero value address register A15 of the global address unit generates a command word write operation as previously described above. Thus data transfer format **8** permits conditional command word generation.

The "N C V Z" field (bits **28–25**) indicates which bits of the status are protected from alteration during execution of the instruction. The conditions of the status register are: N negative; C carry; V overflow; and Z zero. If one or more of these bits are set to "1", the corresponding condition bit or bits in the status register are protected from modification during execution of the instruction. Otherwise the status bits of status register **210** are set normally according to the resultant of arithmetic logic unit **230**.

Data transfer format **9** is recognized by bits **38–37** being "00", bit **24** being "0" and bits **16–13** being "0001". Data transfer format **9** is called the conditional data unit operation and conditional field move transfer format. Data transfer format **9** permits conditional selection of the first source for arithmetic logic unit **230** and conditional storing of the resultant of arithmetic logic unit **230** in the same manner as data transfer format **8**. The conditional arithmetic logic unit operations are defined by the fields "cond.", "c", "r" and "N C V Z" as noted above in the description of data transfer format **8**.

Data transfer format **9** also supports conditional register to register field moves. The condition is defined by the same "cond." field that specifies conditional data unit operations. The source of the field move must be one of data registers **200**. The "src" field (bits **12–10**) specifies the particular data register. The destination of the register to register move is the register defined by the register bank of the "dstbank" field (bits **21–18**) and the register number of the "dst" field (bits **5–3**). The fields "g" (bit **29**), "itm" (bits **23–22**), "e" (bit **9**), "siz" (bits **8–7**) and "D" (bit **6**) define the parameters of the conditional field move. The "g" bit determines that the field move is unconditional if "g"="0" and that the field move is conditional if "g"="1". The "D" bit indicates if the field move is a field replicate move if "D"="1", or a field extract move if "D"="0". These options have been described above. In a field extract move the "itm" field (bits **23–22**) indicates the little endian item number to be extracted from the source register base upon the data size specified by the "siz" field. The extracted field from the source register is sign extended if the "e" bit (bit **9**) is "1" and zero extended if the "e" bit (bit **9**) is "0". The "e" field is ignored during field replicate moves.

Data transfer format **10** is recognized by bits **38–37** being "00", bits **16–15** not being "00" and bit **2** being "1". Data transfer format **10** is called the conditional data unit operation and conditional global data transfer format. Data transfer format **10** permits conditional selection of the first source for arithmetic logic unit **230** and conditional storing of the resultant of arithmetic logic unit **230**. The conditional arithmetic logic unit operations are defined by the fields "cond.", "c", "r" and "N C V Z" as noted above in the description of data transfer format **8**.

Data transfer format **10** also supports conditional memory access via global address unit **610**. The conditional memory access is specified by the fields "g", "Gim/x", "bank", "L", "Gmode", "reg", "e", "siz", "s", "Ga" and "Grm" as previously described. The "g" bit (bit **29**) indicates if the data move is conditional in the manner previously described above. The "Gim/x" field specifies either an index register number or an offset field depending upon the state of the "Gmode" field. The "bank" field specifies the register bank and the "reg" field specifies the register number of the register source or destination of the global memory access. The "L" indicates a load operation (memory to register transfer) by a "1" and a store operation (register to memory transfer) by a "0". The "Gmode" field indicates the operation of global data unit **610** as set forth in Table 47. The "e" bit indicates sign or zero extension for load operations. Note an "L" field of "0" and an "e" field of "1" produces an address arithmetic operation. The "siz" field specifies the data size as set forth in Table 48. The "s" bit indicates whether the index is scaled to the data size as described above. The "Ga" field specifies the address register used in address computation. The "Grm" field indicates the type of addressing operation as set forth in Table 50.

Data transfer format **11** is recognized by bits **38–37** being "00", bit **24** being "0" and bits **16–14** being "001". Data transfer format **11** is called the conditional non-data register

data unit format. Data transfer format **11** permits no memory accesses. Instead data transfer format **11** permits conditional data unit operation with one source and the destination for arithmetic logic unit **230** as any register within digital image/graphics processor **71**. These are called long distance arithmetic logic unit operations. The "As1bank" field (bits **9–6**) specifies a bank of registers that defines the first arithmetic logic unit source in combination with the "src1" field (bits **47–45**) in the data unit format of the instruction. Thus this source may be any register within digital image/graphics processor **71**. The "Adstbnk" field (bits **21–18**) specifies a bank of registers that defines the arithmetic logic unit destination in combination with the "dst" field (bits **50–48**) in data unit formats A, B and C, and the "dst1" field (bits **50–48**) in data unit format E. The conditional arithmetic logic unit operations are defined by the fields "cond.", "c", "r" and "N C V Z" as noted above in the description of data transfer format **8**.

The "R" bit (bit **0**) is a reset bit. The "R" bit is used only at reset. This "R" bit is used only upon reset. The bit determines whether the stack pointer register **A14** is initialized upon reset of digital image/graphics processor **71**. This "R" bit is not available to users via the instruction set and will not be further described.

With so many operations possible within a single instruction, it is possible that more than one operation of a single instruction specifies the same destination data register **200**. In such an event a fixed order of priority determines which operation saves its result in the commonly specified destination register. This fixed order of priority is shown in Table 51 in order from highest priority to lowest priority.

TABLE 51

Priority Rank	Operation
highest	Global address unit data transfer
median	Local address unit data transfer
lowest	Data unit Multiply/ALU => Multiply Rotate ALU => ALU

Thus global address unit data transfers have the highest priority and data unit operations have the lowest priority. Since more than one data unit operation can take place during a single instruction, there is a further priority rank for such operations. If a multiply operation and an arithmetic logic unit operation have the same destination register, then only the result of the multiply operation is stored. In this case no status bits are changed by the aborted arithmetic logic unit operation. Note that if the storing of the result of an arithmetic logic unit operation is aborted due to conflict with a global or local address unit data transfer, then the status bits are set normally. If a barrel rotation result and an arithmetic logic unit operation have the same destination, then only the results of the arithmetic logic unit operation is stored. In this case the status bits are set normally for the completed arithmetic logic unit operation.

This application will now describe how multiprocessor integrated circuit **100** can be programmed to solve some typical graphics processing problems.

One key problem in graphics processing is image encoding. In facsimile transmission, video conferencing, multimedia computing and high definition television a key problem is the amount of data to be transmitted or stored in full motion video. There are known techniques for data compression of individual images that can be used for each frame of video. Current technology cannot simultaneously provide sufficient image compression and acceptable video quality for real time video. Much interest is directed toward algorithms and processors that can provide image compression for full motion video.

There is a proposed motion picture compression standard from the Motion Picture Experts Group (MPEG) which utilizes motion estimation. In motion estimation consecutive frames are compared to detect changes. These changes can then be encoded and transmitted rather than the data of the entire frame. The current proposed MPEG standard compares 16 by 16 pixel blocks of consecutive pixels. One block is displaced to differing positions ±7 pixels in the vertical dimension and ±7 pixels in the horizontal direction. For each displaced position, the proposed standard computes the sum of the absolute value of respective differences between pixels. The displaced position yielding the least sum of the absolute value of differences defines a motion vector for that 16 by 16 pixel block. Once the entire image has been compared, then some frames are transmitted in large degree via motion vectors rather than by pixel values.

This motion estimation computation involves a very large amount of computation. Each displaced position needs 256 differences, whose absolute values are summed. There are 225 such displaced positions (15×15) for each 16 by 16 pixel block. In relatively modest image resolutions such as the h.261 standard proposed for video conferencing with 352 columns lines and 288 rows, each frame includes 198 such 16 by 16 pixel blocks. Thus each frame requires about 23 million subtractions, 23 million absolute values and numerous other computations. This task requires enormous computation capability since full motion video requires at least 24 to 30 frames per second. The most voluminous portion of these computations are the subtractions for each pixel of each displaced position of each 16 by 16 pixel block and the absolute value function. Though there are many other computations, if there were an efficient manner of performing these most voluminous calculations the entire task would be feasible. FIG. **47** illustrates schematically the operation of digital image/graphics processor **71** in a four instruction inner loop for MPEG motion estimation. Note that the example data values indicated are in hexadecimal numbers. Within this four instruction loop, digital image/graphics processor **71** computes 8 differences on 8 bit pixels, forms the absolute values and updates a running sum of the absolute values. This operation will be described in detail to demonstrate the computation power of digital image/graphics processor **71** illustrated in FIG. **3**. The four instructions of the inner loop are:

```
1a. CurrPixel=mzc CurrPixel–PrevPixel
1b. || GX_CNTIndex=MF
1c. || CurrPixel=*(LA_Curr++=4)
2a. SumABS=mc (SumABS+CurrPixel)& @MF|
    (SumABS–CurrPixel)& ~@MF
2b. || GA_CarryCount=&*(GA_CarryCount+GX_
    NumCout)
2c. || PrevPixel=*(LA_Prev++=4)
3a. CurrPixel=mrc CurrPixel–PrevPixel
3b. || GX_NumCout=*(GA_1CntTbl+GX_CNTIndex)
3c. || CurrPixel=*(LA_Curr++=4)
4a. SumABS=mc (SumABS+CurrPixel)& ~@MF|
    (SumABS–CurrPixel)& ~@MF
4b. || PrevPixel=*(LA_Prev++=4)
```

This loop kernel is preferably controlled using hardware loop logic **720** for zero overhead looping in the manner described above.

The complex interactions of these four instructions will be described in detail. In summary, instructions 1a and 3a form the difference between pixels of the current frame and pixels of the previous frame and set bits in multiple flags register **211**. Instructions 2a and 4a add or subtract this difference from a running sum of absolute values. The selection of addition or subtraction is based on the previously set bits within multiple flags register **211**. The local address unit **620** handles fetching the pixel data from the corresponding local memory. This data is placed in a memory accessible by the local port of the digital image/graphics processor executing this algorithm. Note that the data is preferably organized as four adjacent 8 bit pixels per 32 bit data word. The global address unit **610** computes the higher order bits in the running sum of absolute values. This computation of the higher order bits employs a 256 element look up table and address unit arithmetic. Note that all the data unit operations are multiple operations on 8 bit data where both the "Msize" field and the "Asize" field of status register **210** are set to "100".

Table 52 shows the register assignments used in the example of this algorithm listed above. Those skilled in the art would realize that other register assignments may also be used to perform this same loop kernel.

TABLE 52

Register	Variable Name	Data Assignment
D0	—	instruction parameters
D1	PrevPixel	4 previous frame pixels
D2	CurrPixel	4 current frame pixels
D3	PrevPixel	4 previous frame pixels
D4	CurrPixel	4 current frame pixels
D5	SumABS	running sum of absolute value of differences
A0	LA_Prev	previous frame pixel address
A1	LA_Curr	current frame pixel address
A8	GA_CarryCount	running sum of carries
A9	GA_1CntTbl	carry count loop up table base address
X0	—	4
X8	GX_CNTIndex	count of carries from multiple flags register
X9	GX_NumCout	loop up table result

In Table 52: **D0** through **D5** are data registers in data unit **110**; **A8** and **A9** are address registers in global address unit **610**; **X8** and **X9** are index registers in global address unit **610**; **A0** and **A1** are address registers in local address unit **620**; **X0** is an index register in local address unit **620**.

The data unit operation of instruction 1 of the loop forms the difference value CurrPixel–PrevPixel. This difference is between the values of four pixels of the current frame stored in data register **D2** and the values of four corresponding pixels of the previous frames stored in data register **D1**. The "mzc" mnemonic for this instruction indicates: a multiple operation; multiple flags register **211** is zeroed to begin the instruction; and multiple flags register **211** has its least significant bits set by the carry-out results of the multiple sections of arithmetic logic unit **230**. As previously stated, arithmetic logic unit **230** forms this difference while split into four 8 bit sections. The multiple flags register **211** has its four least significant bits set from the respective carry-outs of the four sections. Note that a "0" carry-out result indicates the difference is negative and a "1" carry-out result indicates the difference is not negative.

Global address unit **610** moves the data stored in multiple flags register **211** to index register **X8**. Note that this move takes place during the address pipeline stage of this instruction, which is prior to any data unit **110** operation. Thus this data is the result of instruction 4 of the previous loop and not the result of any operation of data unit **110** during instruction 1.

Local address unit **620** loads data in the address stored in address register **A1** into data register **D4**. This moves data for four pixels of the current frame into position for use in instruction 3. Address register **A1** is pre-incremented and modified by the value in index register **X0**. According to Table 52 this value is "4". Note that it is feasible to employ a 5 bit offset field for this increment value rather than an index register. After this post-increment, address register **A1** holds the address of the word in memory storing the current four pixels of the current frame.

Instruction 2 forms the absolute value of the difference and adds this to a running sum of absolute values. The "mc" mnemonic indicates this is a multiple instruction and that the least significant bits of multiple flags register **211** are set by the respective carry-outs. In this case the carry-outs replace the four least significant bits set in instruction 1. Note that the data unit operation (SumABS+CurrPixel)&@MF|| (SumABS-CurrPixel)& ~@MF is a readily obtainable arithmetic operation using the translated function code "10011010" Hex "9a") as shown in Table 21. The four least significant bits of multiple flags register **211** are expended into 32 bits in expand circuit **238** and supplied to input C bus **243** via multiplexer Cmux **233**. This expanded version of the four least significant bits of multiple flags register **211** forms the terms on the "@MF" line in FIG. **47**. This forms the absolute value and adds it to the running sum. Note that if the difference was negative, then the carry-out bit was "0" and the corresponding expanded multiple flags term is Hex "00". This effectively causes the negative difference to be subtracted from the running sum. On the other hand, if the difference was positive, the corresponding multiple flags term is Hex "FF" and the difference is added to the running sum. Using the expanded multiple flags register bits thus enables the formation of the pixel difference, the absolute value and the running sum in only two instructions. Note that in two cases the sumgenerates a carry-out. This carry-out is stored in multiple flags register **211** to be used later in computation of the higher order bits of the running sum of absolute values.

Global address unit **610** performs address unit arithmetic. The data from the higher order bit look up table stored in index register **X9** is added to a running sum of the higher order bits stored in address register **A8**. Note that the sum of the absolute values of 256 differences of 8 bit pixels may very well overflow the capacity of 8 bits. Thus some manner of accounting for such overflow bits is needed. Index register **X9** holds the count of the number of such overflow accumulated in multiple flags register **211** during one pass through the loop. Instruction 2b sums these into a running sum of these overflow bits, which later forms the higher order bits of the desired sum of absolute value of differences.

Local address unit **620** loads data in the address stored in address register **A0** into data register **D3**. This moves data for four pixels of the previous frame into position for use in instruction 3. Address register **A0** is pre-incremented by the value in index register **X0**, which is 4. Address register **A0** thus points to the current word of previous frame pixel data. Note that this load operation occurs during the address pipeline stage of instruction 2 and is thus available for use in the execute pipeline stage of instruction 3.

Instruction 3a is similar to instruction 1a. Instruction 3a also forms a difference value (CurrPixel–PrevPixel). This difference is between the values of four pixels of the current frame stored in data register **D4** and the values of four

corresponding pixels of the previous frames stored in data register D3. The "mrc" mnemonic for this instruction indicates: a multiple operation; multiple flags register **211** is rotated to begin the instruction; and multiple flags register **211** has its least significant bits set by the carry-out results of the multiple sections of arithmetic logic unit **230**. The rotate in multiple flags register **211** of the carry-outs formed in instruction 2 occurs at the beginning of the execute pipeline stage and makes room for storage of four new carry-outs from this difference. This rotate in multiple flags register **211** thus retains the carry-outs from the instruction 2.

Global address unit **610** performs a table look up operation. The address stored in address register A9 is the base address of a 256 element look up table. Each element in this look up table stores data corresponding to the number of "1's" in the table address. Thus the first element in the table, having a table address of "00000000", stores "0", the second element with a table address of "00000001" stores "1", the third element with a table address of "00000010" stores "1", the fourth element with a table address of "00000011" stores "2" and so forth. The index register X8 stored the carry-outs from the prior pass through the loop as loaded in instruction 1b. Each bit stores the carry-out from a corresponding running sum of the absolute value of the differences. A "1" indicates overflow of the 8 bit word. The look up table returns the number of such "1's", effectively the sum of the overflow bits. This resultant, which is stored in index register X9, is added to the running sum of the overflow bits stored in address register A8 in instruction 2b.

Local address unit **620** loads data in the address stored in address register A1 into data register D2. This moves data for four pixels of the current frame into position for use in instruction 1 of the next loop. Address register A1 is pre-incremented and modified by the value in index register X0, which is "4".

Instruction 4 forms the absolute value of the difference and adds this to the running sum of absolute values in a manner similar to instruction 2. The "mc" mnemonic indicates this is a multiple instruction and that the least significant bits of multiple flags register **211** are set by the respective carry-outs, which replace the four least significant bits set in instruction 3. Data unit **110** effectively forms the absolute value and adds it to the running sum. Note that the running sum SumABS carry-outs are stored in multiple flags register **211** to be used later in computation of the higher order bits of the running sum of absolute values.

There is no global address unit operation in instruction 4 in this example.

Local address unit **620** loads data in the address stored in address register A0 into data register D1. This moves data for four pixels of the previous frame into position for use in instruction 1 of the next pass through the loop. Address register A0 is pre-incremented and modified by the value in index register X0, which is 4.

Some clean up operations follow after this loop kernel has computed the sum of the absolute value of the differences for an entire 16 by 16 pixel block. Once completed data register D5 holds separate sum data in four 8 bit bytes. In addition, address register A8 holds the sum of the higher order bits of the desired sum of absolute value of differences. To obtain the correct sum the data in the four sections of data register D5 are added. An arithmetic operation using the translated function code "01100000" (Hex "60"), which is a field addition, is very helpful in this addition. A method herein called summing 4 bytes into 2 into 1 is described below. This operation starts with partial sum bytes d,c,b,a as follows in a first data register:

dddddddcccccccbbbbbbbbaaaaaaaa

Two masks are needed for this operation. The first mask is alternating Hex "00" and Hex "FF" bytes:

00000000111111110000000011111111

This mask could be formed from Hex "0101" stored in Mflags register **211** via expand circuit **238** when the "Asize" field indicating a byte data size. This first mask could also be stored in a data register. The second mask is a Hex "0000FFFF" mask:

0000000000000000111111111111111

This second mask could be formed by mask generator 239 from an input of 16. Data register D0 is loaded with a default barrel rotate amount "DBR" field indicating an 8 bit rotate. Once these preliminary steps are accomplished, then the sum of 4 bytes into 2 bytes into one byte requires only two instructions. In the first instruction the 4 byte sum data in data register D5 is supplied to both the input A bus **241** via multiplexer Amux **232** and to barrel rotator **235**. The rotation amount is set at 8 bits via the default barrel rotate amount "DBR" field of data register D0. The first mask is supplied to input C bus **243** via multiplexer Cmux **233** and second multiplier input bus **202**. This requires an instruction class field of "001" from Table 39. Arithmetic logic unit **230** performs a field addition (A&C)+(B&C). The resultant sum is returned to the source data register D5. This process is explained as follows. Rotation of the original data by 8 bits yields:

aaaaaaaadddddddcccccccbbbbbbbb

Arithmetic logic unit **230** effectively masks both the original and rotated data and then adds them in two separate fields as controlled by the first mask. Applying the first mask to the original data yields:

00000000cccccccc00000000aaaaaaaa

Applying the first mask to the rotated data yields:

00000000ddddddd00000000bbbbbbbb

The addition of the these two values results in two 9 bit intermediate sums in a single data word:

0000000uuuuuuuuu0000000vvvvvvvvv

which is stored back into the first source register. Note that the addition of two 8 bit numbers may yield a 9 bit number as shown above. The power of the three input arithmetic logic unit **230** is shown here where the shift, mask and addition are performed in a single cycle of arithmetic logic unit **230**.

The second instruction is similar to the first instruction. In the second instruction the partial sum data stored in a data register is supplied to both the input A bus **241** via multiplexer Amux **232** and to barrel rotator **235**. The rotation amount is set at 16 bits via a 5 bit offset field of "10000" selected by multiplexer Imux **222**, supplied to second multiplier input bus **202** and selected by multiplexer Smux **231**. The second mask is supplied to input C bus **243** via the 5 bit offset field selected by multiplexer Imux **222**, supplied to second multiplier input bus **202**, selected by multiplexer Mmux **234**, formed into the 16 bit second mask via mask generator **239** according to Table 19 and further selected by multiplexer Cmux **233**. This requires an instruction class

field of "011" from Table 39. Arithmetic logic unit **230** performs a field addition (A&C)+(B&C). The resultant sum is returned to the source register. This process is explained as follows. Rotating this partial sum by 16 bits produces:

0000000vvvvvvvvv0000000uuuuuuuuu

Applying the second mask to the original partial sum data yields:

00000000000000000000000vvvvvvvv

Applying the second mask to the rotated partial sum data mask yields:

00000000000000000000000uuuuuuuuu

The field addition of the these two values results in one 10 bit sum of the four byte partial sums:

0000000000000000000000rrrrrrrrr

which may be stored into the original source data register. Note that addition of the two 9 bit numbers may overflow into a 10 bit sum.

The final desired sum of the motion estimation process is formed by adding the above four byte partial sum to the running overflow sum rotated left 8 places. A simple rotate and add accomplishes this final addition.

This field addition is particularly useful when doing multiple arithmetic. As illustrated above it provides a fast final addition of four partial sums that are initially spread across four bytes, requiring only two instructions. Because this final addition is fast, digital image/graphics processor multiple arithmetic can have a speed advantage over single-byte arithmetic even when only a small number of additions are needed to provide the partial sums. This method is particularly useful in the clean up of the sum of absolute value of differences described above.

Suitable outer loops are needed to supplement this loop kernel. By way of example only, a suitable outer loop could so load the pixel data for the current and previous frame that an entire 16 by 16 pixel block may be handled without interrupting the inner loop. Alternatively, outer loops insure proper registration of the pixel data when employing the inner loop. Displacement of the 16 by 16 pixel blocks are also handled by larger loops. Larger loops also make the selection of the motion vector for each pixel is based upon the least sum of absolute value of differences. All these program features are within the capability of one skilled in the art. Note that these outer loops are executed much less frequently, therefore maximum coding density is not as important than in the inner loop kernel listed above.

Another function used in the proposed MPEG encoding standard is variable length codes. This is often called Huffman encoding. Huffman encoding has many other uses in addition to video encoding. Variable length codes are employed for discrete data elements to be transmitted. In order to reduce the amount of data to be transmitted, more frequently used data is encoded using fewer bits.

Huffman variable length encoding specifies both encoding and decoding techniques. In an application such as multimedia computing, the software media vendor performs the encoding. The user's computer decodes the encoded data when used. In this event, large computing resources can be employed during encoding or the encoding may be performed taking longer than the real time length of the video sequence. This is feasible since encoding is done only once. Thus in such applications only decoding need be done in real time. In other applications such as video conferencing both encoding and decoding must be done in real time by the user's apparatus.

An example of such variable length coding is shown in Table 53 below. Each coded number consists of a size field and a value field. Table 53 shows an example using a 2 bit size field and a value field of up to 3 bits.

TABLE 52

Size	Value	Encoded Number
00	—	0
01	0	−1
01	1	1
10	00	−3
10	01	−2
10	10	2
10	11	3
11	000	−7
11	001	−6
11	010	−5
11	011	−4
11	100	4
11	101	5
11	110	6
11	111	7

Table 53 shows only some examples of Huffman encoding. Other combinations of the number of size bits and the number of value bits are feasible. Table 54 shows the range of numbers which can be encoded with various numbers of size bits and numbers of value bits.

TABLE 54

Number of Size Bits	Number of Value Bits	Range of Encoded Numbers
1	0	0
1	1	−1, 1
2	0	0
2	1	−1, 1
2	2	−3, −2, 2, 3
2	3	−7 to −4, 4 to 7
3	0	0
3	1	−1, 1
3	2	−3, −2, 2, 3
3	3	−7 to −4, 4 to 7
3	4	−15 to −8, 8 to 15
3	5	−31 to −16, 16 to 31
3	6	−63 to −32, 32 to 63
3	7	−127 to −64, 64 to 127
4	0	0
4	1	−1, 1
4	2	−3, −2, 2, 3
4	3	−7 to −4, 4 to 7
4	4	−15 to −8, 8 to 15
4	5	−31 to −16, 16 to 31
4	6	−63 to −32, 32 to 63
4	7	−127 to −64, 64 to 127
4	8	−255 to −128, 128 to 255
4	9	−511 to −256, 256 to 511
4	10	−1023 to −512, 512 to 1023
4	11	−2047 to −1024, 1024 to 2047
4	12	−4095 to −2048, 2048 to 4095
4	13	−8191 to −4096, 4096 to 8191
4	14	−16383 to −8192, 8192 to 16383
4	15	−32768 to −16384, 16384 to 32768

Thus a single bit size permits only up to one bit for value and can encode −1, 0 and 1. A two bit size permits the value to be represented by up to 3 bits and can encode from −7 to 7. A 3 bit size permits up to 7 bits for value and can encode from −127 to 127. If size is encoded in 4 bits, then the value can have up to 15 bits and can encode from −32768 to 32768. For any particular application of Huffman encoding

the number of size bits is constant. The number of value bits is selected to provide a range including the number to be encoded. From Table 54 it is clear that numbers near zero require fewer bits to encode than numbers further from zero. The raw data is preferably quantized or otherwise selected or manipulated so that numbers near zero occur more frequently than numbers distant from zero. Thus the more frequently encountered data requires fewer bits to encode. This feature reduces the average number of encoded bits that must be transmitted or stored.

An algorithm for Huffman encoding a sample appears below. This example assumes a range of values to be encoded from algorithm presupposes that the range of numbers is from −2047 to 2047 represented by 12 bits. These numbers are right justified in sign extended 32 bit words. Note that conversion from packed signed extended 16 bit data can be accomplished using sign extended half word memory loads or register to register moves, or using half word masks coupled with rotation of 16 bit data located in the most significant bits of a 32 bit word. Inspection of Table 54 indicates this range of numbers can be encoded using 4 size bits and up to 10 value bits. Thus the data length of the Huffman encoded data may vary from 4 to 14 bits.

This example includes the following steps: forming the absolute value, determining the size via left most "1" detection; generation of the value bits for negative numbers; and packing the size and value.

1. RawData=RawData
2a. AbsValue=[.n] 0−RawData
2b. || AbsValue=[ge] RawData
3. Size=[.n] LMO AbsValue
4. Value=[n] RawData+% Size
5. RotSize=Size \\Size
6. SizeValue=RotSize & ~%Size|Value & % Size

Table 55 shows the register assignments in this example of Huffman encoding.

TABLE 55

Register	Variable Name	Data Assignment
D1	RawData	raw data to be encoded
	Value	corrected value portion of encoded data
D2	AbsValue	absolute value of raw data
	RotSize	rotated data size portion of encoded data
D3	Size	data size portion of encoded data
D4	SizeValue	packed encoded data

Instruction 1 sets the status bits stored in status register SR **210**. The negative "N" bit will be used in two later instructions. Instruction 2 forms the absolute value of RawData. Note the register to register move operation has priority over the arithmetic logic unit operation. If RawData≧0, then the register move takes place according to the greater than or equal to "ge" mnemonic and AbsValue is set to RawData. If RawData<0, then the register move does not take place and the arithmetic logic unit operation takes place. This priority of operation is in accordance with Table 51. Thus AbsValue is set to 0-RawData. This effectively sets AbsValue to the absolute value of RawData. Note the ".n" mnemonic in instruction 2a preserves the status of the negative "N" status bit regardless of the results of the arithmetic logic unit operation.

Instruction 3 determines the size of the original data. Instruction 3 employs LMO/RMO/LMBC/RMBC circuit **237** to determine the left most one in AbsValue. This is the most significant bit in the raw data. The value returned by LMO/RMO/LMBC/RMBC circuit **237** in the form shown in Table 16 yields the number of significant bits in the raw data, thus the desired size portion of the encoded number. The absolute value formed in instruction 2 ensures that this left most one operation generates the correct result for negative numbers. The ".n" mnemonic preserves the status of the negative "N" status bit. This same result can be achieved by replacing instructions 2 and 3 with Size=[.n] LMBC RawData. LMO/RMO/LMBC/RMBC circuit **237** would detect the most significant "1" for positive data and the most significant "0" for negative data. The form listed above may be preferred if the algorithm requires more data transfer operations.

Instruction 4 corrects the RawData into the Huffman form as shown in Table 54. Note that Value and RawData are the same register according to Table 55. Thus if RawData is greater than or equal to zero, the condition of instruction 4 fails and Value is RawData. If RawData is less than zero according to the "n" mnemonic, then the addition takes place. This realizes the encoding of negative numbers of the form shown in Table 53.

Instructions 5 and 6 form packed data including the size and value. Instruction 5 rotates Size by the previously determined number of bits of value. Instruction 6 merges these into a single data word. Note that any practical implementation of such Huffman encoding would require additional data handling operations. These would be required to input the raw data and to pack complete data words of encoded data and output these packed words. These functions are known in the art and will not be described in detail.

A simplified example of Huffman decoding on the multiprocessor integrated circuit of this invention is described below.

1. L_WordAddressX=BitAddress>>u 5
2. Nop
3. ThisWord=*(L_WordAddressBase+=[L_WordAddressX])
4a. AlignedWord=ThisWord<<BitAddress
4b. || NextWord=*(L_WordAddressBase+[1])
5. Cur32Bits=AlignedWord & ~%BitAddress|NextWord\\BitAddress & % BitAddress
6a. L_HuffLUTX=Cur32BIts>>u 26
6b. || Dummy0000=&*(L_WordAddressBase−=[L_WordAddressX])
7. Nop
8. UsedBits=sb*(L_BitsUsedAddress+[L_HuffLUTX])
9a. BitAddress=BitAddress+UsedBits
9b. || L_BitsUsedAddress=*(G_Space+O_AC_BitsUsedAddress)
9c. || RunSize=ub*(L_RunSizeAddress+[L_HuffLUTX])

HuffmanLoopStart:
Jump_Back_In:

10a. WordAddress=BitAddress>>5
10b. || BR=[c]*(G_Space+O_ExtendedTableDecode)
11a. PosOffset=0−(RunSize\\28 & % 28)+cin
11b. || L_WordAddressX=WordAddress
11c. || FunctionEalu=*(L_Space+Tealu_Function)
12a. FieldSize=FunctionEalul(RunSize & % 4)
12b. || LC1=RunSize

13a. G_OffsetX=G__OffsetX+PosOffset

13b. ‖ ThisWord=*(L__WordAddressBase+=[L__WordAddressX])

14a. AlignedValue=EALU(D1. Cur32Bits\\UsedBits, % FieldSize)

14b. ‖ LC1=[le] A15

15a. AlignedWord=ThisWord<<BitAddress

15b. ‖ G__ZigZagDCTX=ub*(G__ZigZagLUTop—[G__Offset])

15c. ‖ NextWord=*(L__WordAddressBase+[1])

16a. Cur32Bits=AlignedWord & ~% BitAddress|NextWord\\BitAddress & % BitAddress

16b. ‖ L__RunSizeAddress=*(G__Space+O__AC__RunSizeAddress)

16c. ‖ Bit31=*(L__Space+tBit31)

17a. Dummy0001=AlignedValue & (Bit31 \\FieldSize)

17b. ‖ L__HuffLUTX=ub3 Cur32Bits

17c. ‖ Dummy0003=&*(L__WordAddressBase~=[L__WordAddressX])

18a. AdjustedValue=[z] AlignedValue—% FieldSize

18b. ‖ QuantStep=h*(G__QuantizationTable—[G__OffsetX])

19a. IDCTValue=QuantStep*AdjustedValue

19b. ‖ UsedBits=sb*(L__BitsUsedAddress+[L__HuffLUTX])

End__of__Tight__Loop:

20a. BitAddress=BitAddress+UsedBits

20b. ‖ *(G__IDCTBase+[G__ZigZagDCTX])=h IDCTValue

20c. ‖ RunSize=ub*(L__RunSizeAddress+[L__HuffLUTX])

Table 56 shows the data register assignments employed in this example of the Huffman decode algorithm.

TABLE 56

Register	Variable Name	Data Assignment
D0	FieldSize	number of bits in value field
	FunctionEalu	extended arithmetic logic function code
D1	BitAddress	bit address of next bit to decode
D2	AlignedWord	data word containing next bit in most significant bit
	Cur32Bits	data word containing next 32 bits of data
D3	Dummy0000	register set but not used
	AlignedValue	stripped aligned value
	AdjustedValue	negative corrected decoded value
	IDCTValue	dequantized value ready for inverse discrete cosine transform operation
	WordAddress	base address of word including first bit to decode
D4	NextWord	following data word
	Dummy0001	register set but not used
	UsedBits	total number of bits used by Huffman code and encoded value
	Bit31	Hex "80000000"

TABLE 56-continued

Register	Variable Name	Data Assignment
D5	ThisWord	data word containing next bit to decode
	Dummy0003	register set but not used
	QuantStep	quantization multiplier
D6	RunSize	packed size of field and zero run length (4 bits each)
D7	PosOffset	run length of zeros plus 1

Table 57 lists proposed address register assignments for implementing this example of a Huffman decode algorithm.

TABLE 57

Address Register	Variable Name	Data Assignment
A0	L__Space	pointer to local scratchpad memory
A1	L__BitsUsedAddress	base address for bits used
A2	L__WordAddressBase	base address of word containing the first bit to decode
A3	L__RunSizeAddress	base address of size/run
A8	G__QuantizationTable	quantization table base address
A9	G__IDCTBase	base address of 8 by 8 output block
A10	G__ZigZagLUTop	address register zig-zag scan table look-ups
A11	G__Space	pointer to global scratchpad memory

Table 58 lists proposed index register assignments for implementing this example of a Huffman decode algorithm.

TABLE 58

Index Register	Variable Name	Data Assignment
X0	L__WordAddressX	address word containing next bit to decode
X1	L__HuffLUTX	offset address for Huffman look-up table
X8	G__OffsetX	index register for zig-zag scan table look-ups
X10	G__ZigZagDCTX	index register for zig-zag scan table look-ups

This example of Huffman decoding includes two parts. Instructions 1 to 9 involve initial loop set up. This portion of the program also deals with an initial DC term which has a size of 6 bits. Instructions 10 to 20 form a loop for decoding the stream of Huffman encoded data. These are AC terms and include a run value of 4 bits and a size value of 4 bits. Each pass through the loop decodes one instance of Huffman encoded data. Note that instructions 1 to 9 do not include the necessary loop set up for the loop including instructions 10 to 20. This is accomplished in a manner previously described.

Instruction 1 sets a word address index L__WordAddressX. The algorithm keeps a bit address BitAddress which points to the next bit to be decoded. Instruction 1 sets L__WordAddressX as BitAddress right rotated 5 bits. Thus BitAddress is divided by 2^5=32 to obtain

the address of the next 32 bit word. The Nop of Instruction 2 is required by the pipeline so that the value of L_WordAddressX set in the execute pipeline stage of instruction 1 is available during the address pipeline stage operation of instruction 3.

Instruction 3 loads the data word including the next bit to be decoded. Instruction 3 is a local address unit operation. A register is loaded from the memory location equal to the sum of a base address L_WordAddressBase and the just computed index address L_WordAddressX. The syntax of this instruction indicates that L_WordAddressX as scaled to the selected data size is pre-added to L_WordAddressBase, which is modified by the addition.

Instruction 4a forms an aligned version of the next bits to be decoded. ThisWord just loaded from memory contains the next bit to be decoded. The left rotate by the value BitAddress aligns the next bit to be decoded into bit **31** of AlignedWord, the most significant bit. Note that only the five least significant bits of BitAddress are used by the hardware of data unit **110** in this rotate operation. Thus the rotate is limited to the range of 31 bits. Instruction 4b is a local address unit operation. Instruction 4b loads the next data word in memory following ThisWord. Note that the base address of L_WordAddressBase was set to the address of ThisWord in instruction 3. Thus L_WordAddressBase plus 1 scaled to the data size is the address of the next data word.

Instruction 5 forms Cur32Bits as the next 32 bits to be decoded. Cur32Bits differs from AlignedWord because AlignedWord probably includes less than 32 of the next bits to be decoded. AlignedWord is masked with the inverse of BitAddress. This mask ~% BitAddress has a number of least significant "0's" equal to the number of the five least significant bits of BitAddress with the most significant bits equal to "1's". This ANDed with AlignedWord selects the next following data to be decoded. The mask % BitAddress has a number of least significant "1's" equal to the number of the five least significant bits of BitAddress with the most significant bits of this mask equal to "0's" NextWord is left rotated by the number of the five least significant bits of BitAddress. The AND thus selects the number of most significant bits of NextWord to fill the 32 bits of Cur32Bits.

Instruction 6a sets an address index L_HuffLUTX. Instruction 6a is an unsigned right rotate of Cur32Bits by 26 places. This puts the 6 most significant bits of Cur32Bits into the 6 least significant places and zero fills the remaining places. The address index L_HuffLUTX is used as an index into a look-up table. Instruction 6b resets the address L_WordAddressBase in an address arithmetic operation. The syntax of instruction 6b pre-subtracts L_WordAddressX as scaled by the data size from L_WordAddressBase. This reverses the base address modification of instruction 3. The address register is modified in this way because it makes loading NextWord easier. Without such modification of L_WordAddressBase by L_WordAddressX, computing the address of Next Word would require an arithmetic unit operation an consequent delay slots before the computed address could be used in the load operation. This is an example where using address arithmetic saves operations. Note that the same net operation could be achieved using a memory load into Dummy0000. An actual memory load operation is not used in this example to reduce the possibility of memory contention at crossbar **50**. The Nop of instruction 7 is required by the pipeline so that the value of L_HuffLUTX set in the execute pipeline stage of instruction 6 is available during the address pipeline stage operation of instruction 8.

Instruction 8 is a local address unit operation. This is a look-up table operation using a base address of L_BitsUsedAddress and an index of L_HuffLUTX scaled to the data size. The load operation is a signed byte operation according to the "sb" mnemonic. UsedBits is set to a sign extended byte equal to the data stored at the address of the sum of L_BitsUsedAddress and L_HuffLUTX scaled to the data size. This look-up table operation converts the next 6 bits to be decoded into a number of bits used, expanding the size quantity into the sum of the run, size and value bits.

Instruction 9a updates BitAddress by adding the just determined UsedBits. Instruction 9b loads into L_BitsUsedAdddress an address stored in a global scratch-pad memory at location O_AC_BitsUsedAddress. This address is the address of the beginning of a look-up table. Note that O_AC_BitsUsedAddress is not an index register but rather a code for a short offset value. This instruction 9c loads RunSize. This unsigned byte load (mnemonic "ub") is from a look-up table having a base address L_RunSizeAddress and a location equal to the index L_HuffLUTX scaled to the data size. Thus the index L_HuffLUTX serves as an index into two tables, a first to determine UsedBits (instruction 8) and a second to determine RunSize.

A loop used for Huffman decoding starts at instruction 10, which is given the labels HuffmanLoopStart and Jump_Back_In. Many of the steps previously described in the start up portion of the program are repeated within the loop. Instruction 10a sets WordAddress equal to BitAddress right shifted 5 places. This converts a bit address into a word address in a manner previously described regarding instruction 1. Instruction 10b is a branch instruction. The branch destination is stored in a location corresponding to O_ExtendedTableDecode within the global scratchpad memory starting at G_Space. Note O_ExtendedTableDecode is an instruction specified short offset value. The "c" mnemonic indicates this branch is taken if the arithmetic logic unit operation BitAddress=BitAddress+UsedBits generates a carry output. Note that this arithmetic logic unit operation setting the carry output is the same for initial entry into the loop via instruction 9 and return to the loop start from instruction 20. This branches the program out of this loop for the case in which the space for storing the next bits to be decoded, which are pointed to by BitAddress, is exceeded. The program continues from the location stored at O_ExtendedTableDecode to reuse the memory holding the next bits to be decoded by loading additional bits from another memory. Once this house keeping is complete, the program returns to instruction 10 via the label Jump_Back_In.

Instruction 11a computes PosOffset. RunSize is left rotated 28 bits and masked by a mask having bits **31** to **28** all "0's" and bits **27** to **0** having all "1's" (%28). This effectively right shifts RunSize by 4 bits. Note that this particular manner of generating the right shift takes advantage of a 5 bit offset value setting both the rotate amount and the mask input. Since cin is set by the arithmetic logic unit operation of the previous instruction, which is only a rotate operation, cin is always "1". Thus PosOffset is set equal to one more than 0-Run. Instruction 11b sets the index register L_WordAddressX equal to the previously computed value WordAddress. This technique sets L_WordAddressX rather than directly setting this register as in instruction 1 because the direct setting of the non-data register requires global port source bus Gsrc **105** and global port destination bus Gdst **107** is inconsistent with the condition branch instruction in instruction 10b. Instruction 11c loads data register **D0** with

a code used in a later extended arithmetic logic unit operation. This code is stored in the local scratchpad memory at a location corresponding to an offset value Tealu_Function.

Instruction 12a modifies the extended arithmetic unit operation code stored in data register D0. FieldSize, which is also stored in data register D0, is replaced with the AND of the just recalled FunctionEalu and the four least significant bits of RunSize. These are extracted with the mask %4. This extracts the size from RunSize and stores it in the default barrel rotate amount field "DBR" of data register D0. Thus the default barrel rotate amount in the later extended arithmetic logic unit operation is set by this merge instruction. To facilitate this merge, the data stored in bits 4 to 0 at index Tealu_Function within the local scratchpad memory should be "00000".

Instruction 12b sets the loop counter LC1 equal to RunSize. In the MPEG standard blocks of graphic data are transformed via a discrete cosine transform (DCT). This transformation converts the pixel data into two dimensional frequency data. The two dimensional frequency data is scanned via a zig-zag pattern from low frequency data to high frequency data. This moves low frequency data into the first transformed values and high frequency data into later transformed values. Most graphic blocks will have a minimum of high frequency data. This means that many of the transformed data values will be near zero and suitable for encoding according to the technique shown in Table 54. This transformation also means that in most instances a point in the data stream will be reached where the remaining transformed values are all zero. In the MPEG standard this state is signaled by a RunSize value of "00000000". When such a RunSize value is found, then an entire block of data is decoded and the loop should be re-initialized. Thus if RunSize is an end of block marker equal to "00000000", then the loop count is zero and the loop is not re-entered.

Instruction 13a updates the value of G_OffsetX. G_OffsetX determines if all 64 bins of a block have been used. Note this would only occur if the last bin were nonzero. Otherwise a RunSize of zero would be the last data for a block. The index G_OffsetX stores the accumulated runs of RunSize via PosOffset. Since PosOffset is negative, G_OffsetX becomes less than or equal to zero when the 64 bins of a block are complete. Note that the additional 1 in PosOffset is needed to insure that each instance of a bin value is counted. Instruction 13b loads the data word including the next bits to be decoded into ThisWord in the same manner as instruction 3.

Instruction 14a is an extended arithmetic logic unit operation. This instruction performs the logic operation AlignedValue=Cur32Bits\\UsedBits&% FieldSize. The left rotate of Cur32BIts by UsedBits replaces the next bits to be decoded from the most significant bits to the least significant bits. This is masked by FieldSize. This aligns the value portion of the next bits to be decoded into the least significant bits of AlignedValue. Instruction 14b sets the loop count in LC1 to "0" from the zero value address register A15 if the arithmetic logic unit operation of instruction 13a generates a result less than or equal to zero according to the "le" mnemonic. As previously discussed, this indicates that an entire block has been decoded and thus the loop should be exited.

Instruction 15a is similar to instruction 4a. This places the next bits to be decoded from ThisWord into the most significant bits of AlignedWord. Instruction 15b sets an index G_ZigZagDCT from a look-up table starting at the address stored in G_ZigZagLUTop based upon the previously computed index value G_Offset. As previously stated the MPEG encoding technique involves standard blocks of graphic data transformed via a discrete cosine transform (DCT). Decoding requires computation of an inverse discrete cosine transform (IDCT). The order of use of the decoded values depends upon the algorithm computing the inverse discrete cosine transform. Use of the look-up table starting at the address of G_ZigZagLUTop, enables a single look-up table to handle a zig-zag scan pattern as well as this preferred ordering of components for the inverse discrete cosine transform algorithm. Instruction 15c loads NextWord from memory in the same manner as previously described at instruction 4b.

Instruction 16a is similar to instruction 5. This instruction forms Cur32Bits as a full 32 bit word with the next bit to be decoded to in the most significant bit.

Instruction 16b is a global memory load. The address L_RunSizeAddress is loaded with the value from the global scratchpad memory pointed to by offset value O_AC_RunSizeAddress. Instruction 16c sets Bit31 equal to the data stored in the local scratchpad memory at a location indicated by offset tBit31. In accordance with this example, the data at this address is Hex "80000000", or bit 31 set to "1" and all other bits "0". This is used in a masking operation to be described below.

Instruction 17a performs a test on the data of AlignedValue. AlignedValue is ANDed with Bit31 (Hex "8000000") as left rotated by FieldSize. Bit31 as left rotated by FieldSize sets a "1" at the most significant bit of the value stored in AlignedValue. As evident from the examples of Table 54, negative values have a "0" in this location and positive values have a "1" in this location. Thus if the encoded value is negative, then the result is zero and the "Z" bit of status register SR 210 is set. If the encoded value is positive, then the result is nonzero and the "Z" bit of status register SR 210 is not set. As indicated by the register designation Dummy0001, the data stored in the destination register is never used. This instruction only sets the status bits in status register SR 210. Instruction 17b performs a function similar to instruction 6a. Instruction 17b loads L_HuffLUTX with the third unsigned byte of Cur32Bits. Note that the DC term handled in instruction 6a had 6 size bits, while the AC term handled in instruction 17b includes a byte consisting of 4 run bits and 4 size bits. Instruction 17c is an address arithmetic instruction which recovers the base word address stored in L_WordAddressBase. This is similar to instruction 6b.

Instruction 18a used the zero status bit "Z" set in instruction 17a. AdjustedValue is replaced with the difference of AdjustedValue and a mask of FieldSize if the result of instruction 17a was zero. Thus if the encoded value is negative it is subtracted from constant having a number of "1's" equal to the field size. Inspection of Table 53 indicates that this subtraction recovers the encoded number in signed form. Note in instruction 17a that AlignedValue and AdjustedValue are assigned the same data register D3, thus the data is unchanged if the test fails. Instruction 18b is a memory load operation. QuantStep is loaded with a quantization multiplier constant corresponding to the current bin of the 64 bins of a data block. This quantization multiplier constant is stored in a look-up table beginning at the address stored in G_QuantizationTable at a location corresponding to the value of index G_OffsetX. Note that G_OffsetX is set at instruction 13a and corresponds to the current bin.

Instruction 19a is a multiplication operation. The product of the just loaded QuantStep and AdjustedValue determines IDCTValue. IDCTValue is a dequantized value ready for inverse discrete cosine transform. This is the desired result of the Huffman decode operation. Instruction 19b updates the value of UsedBits in the same manner as instruction 8.

Instruction 20 is the last instruction of the loop and is labeled End_of_Tight_Loop. Instruction 20a updates BitAddress in the same fashion as instruction 9a. Note that the carry of this operation determines whether the conditional branch is taken at instruction 10b for the next iteration of the loop. Instruction 20b stores the just determined value of IDCTValue in a variable table starting at the address of G_IDCTBase. The index G_ZigZagDCTX which selects the location within this table was set in instruction 15b based upon the current bin stored in G_OffsetX. Thus the decoded value is stored in the order optimal for the inverse discrete cosine transform algorithm. Note the "h" mnemonic indicates that this is a half word or 16 bit data transfer. Instruction 20c loads RunSize in the same fashion as instruction 9c.

The loop of instruction 10 to 20 repeats until encountering one of three exits. If BitAddress+UsedBits generates a carry, the instruction 10b branches to another program sequence to handle loading additional data. Generally, once new data is loaded this loop will be re-entered at instruction 10, label jump_Back_In. The loop exits when an end of block RunSize of "00000000" occurs. This indicates the end of a block of data. The loop also exits when G_OffsetX is decremented to zero via PosOffset.

Another widely used operation in graphics processing is the mean squared error. Mathematically this is expressed as:

$$MSE = \frac{1}{n \times m} \sum_{x=0}^{n} \sum_{y=0}^{m} (x-y)^2$$

A straight forward approach involves two nested loops forming the summations into a running sum. The division by the product of n and m takes place following the completion of the nested loops. The kernel includes forming the difference and the square and the data move operations to transfer data from memory **20** to the data registers of the particular digital image/graphics processor **71**, **72**, **73** or **74**. This process is similar to the process noted above with respect to the sum of the absolute difference values.

Such a straight forward approach may not use the hardware resources with the greatest efficiency. Multi-processor integrated circuit **100** may provide several techniques for performing the same function. As examples only, address unit arithmetic may replace arithmetic operations employing data unit **110** or register-to-register moves with field extraction and sign/zero extension may replace mask and rotate operations employing data unit **110**. In many cases these alternate operations involve differing characteristics in precision supported, timing and availability of intermediate results and the like. As an example, multiple arithmetic can greatly speed many operations, if the algorithm needs only the reduced number of bits available. Suppose as an example that the quantities x and y are only eight bit values. Using multiple arithmetic to simultaneously form four differences may result in a 9 bit difference with the borrow term formed as the section carry output. This ninth bit can be stored in multiple flags register **211** for later use. Note that the square of the difference is the same as the square of the absolute value of the difference. Thus it is possible to limit the differences formed to 8 bits using the absolute value technique described above. Then multiplier **220** can perform a multiple 8 by 8 multiply to form two squares simultaneously. The lower two bytes are properly positioned for such a multiple multiply operation. The upper two bytes may be extracted and positioned using either barrel rotator **235** or field extract/extend moves. Two running sums are formed, one for the upper byte differences and one for the lower byte differences. The squared error terms are 16 bits, therefore 32 bits are needed to store these running sums. As in the case of the sum of absolute difference values described above, the two running sums are added during wrap up.

An inner loop kernel for the mean squared error algorithm is listed below.

```
1a. Err=mc CurrBlk-PredBlk
1b. || LX_SqErro=uh0 Sq_ErrA
1c. || Dummy=&*(LA_SumA+=LX_SqErr2)
2a. ABS_Err=m (0+Err)& @MF|(0-Err)& ~@MF
2b. || LX_SqErr1=uh1 Sq_ErrA
2c. || CurrBlk=w *LA_Curr
3a. SQ_ErrA=mu ABS_Err * ABS_Err
3b. || ABS_ErrB=EALUT(Hex "00", ABS_Err)
3c. || LX_SqErr2=uh0 SQ_ErrB
3d. || Dummy=&*(LA_SumA+=LX_SqErr0)
4a. Sq_ErrB=mu ABS_ErrB*ABS_ErrB
4b. || MSE_SumB=EALUT(MSE_SumB, Sq_ErrB)
4c. || PredBlk=w*GA_Pred
4d. || Dummy=&*(LA_SumA+=LX_SqErr1)
5a. LX_SqErr0=uh0 Sq_ErrA
5b. || Dummy=&*(LA_SumA+=LX_SqErr2)
6. LX_SqErr1=uh1 Sq_ErrA
7a. LX_SqErr2=uh0 Sq_ErrB
7b. || Dummy=&*(LA_SumA+=LX_SqErr0)
```

Table 59 shows the register assignments used in the example of this algorithm listed above. Those skilled in the art would realize that other register assignments may also perform this same loop kernel.

TABLE 59

Register	Variable Name	Data Assignment
D0		default rotate amount 16
D1	MSE_SumB	second running sum
D2	Sq_ErrB	second squared error
D3		Hex "00000000"
D4	ABS_Err	absolute value of error
	Sq_ErrA	first squared error
D5	dummy	unused result
	PredBlk	preceding block value
D6	CurrBlk	current block value
D7	ABS_ErrB	second absolute error
	Err	error difference
A0	LA_SumA	first sum address
A1	LA_Curr	current block address
A8	GA_Pred	preceding block address
X0	LX_SqErr0	first square error index address
	LX_SqErr2	second square error index address
X1	LX_SqErr1	third square error index address

In Table 59: **D0** through **D7** are data registers in data unit **110**; **A8** is an address register in global address unit **610**; **A0** and **A1** are address registers in local address unit **620**; **X0** and **X1** are index registers in local address unit **620**.

The data unit operation of the first instruction (1a) forms the difference between the current block value CurrBlk and the preceding block value PredBlk. The "mc" mnemonic indicates this is a multiple operation and that the carries are stored in multiple flags register **211**. In this example, there are four eight bit subtracts taking place simultaneously. The global address unit operation of the first instruction (1b) loads the first byte of the first squared error into index register **X0**. Note that the mnemonic "uh0" indicates that

this load operation extracts the first byte (byte 0) into a half word (16 bits) of the destination with zero extension. The local address unit operation of the first instruction (1c) performs an address unit arithmetic operation. The "+=" operator indicates that this address unit operation employs pre-addition of the index register to modify the base address register. This operation adds a second squared error term LX_SqErr1 stored in index register X0 to a running sum stored in address register A0. Note that the destination register D5 is a dummy and the data is stored in the modified address register A0.

The data unit operation of the second instruction (2a) forms the absolute value of the differences. Note that the carry-outputs stored in multiple flags register 211 controls whether the addition or the subtraction takes place. The "m" mnemonic indicates that this is a multiple operation, thus individual bits from multiple flags register control corresponding multiple sections. As explained above, this absolute value restricts the difference to eight bits enabling an 8 bit by 8 bit split multiply operation, thereby doubling the speed of computation over a 16 bit by 16 bit multiply operation. The global address unit operation (2b) is a byte load. The "uh1" mnemonic indicates that this load operation extracts the second byte (byte 1) into a half word (16 bits) of the destination with zero extension. The local address unit operation is a data-load. The current block data stored in memory at the address stored in address register A0 is loaded into data register D6. The "w" mnemonic indicates that this is a word (32 bit) data transfer. The address register A1 is post incremented corresponding to the data size to point to the next 32 bit data word.

Instruction 3 includes a multiply operation forming the square. The first data unit operation (3a) in a multiple unsigned "mu" 8 bit by 8 bit multiply operation. The data is the absolute value of the difference stored in data register D4 and the result is stored in D4. The second data unit operation is an extended arithmetic logic unit true (EALUT) operation. Note that the multiple multiply operation is supported only in conjunction with an extended arithmetic logic unit operation. Thus the desired set of function signals are pre-loaded in the "EALU" field (bits 26–19) of data register D0. This should occur during a set up portion of the program not shown above. The particular extended arithmetic logic unit operation called for in instruction 4b is a rotate and add. The rotate is the default barrel rotate amount stored in the "DBR" field (bits 4–0) of data register D0, which is 16. Note that data register D3 is pre-loaded with the value Hex "00000000", thus adding zero during the rotate and add operation. This prepares the two differences in the most significant bits for multiple multiplication by rotating them to the 16 least significant bits. The global address unit operation (3c) loads the first byte (byte 0) of data register D2 into a half word (16 bits) of index register X0 with zero extension. The local address unit operation (3d) performs an address unit arithmetic operation using pre-addition of the index register to modify the base address register. This adds a first squared error term LX_SqErr0 stored in index register X0 to a running sumstored in address register A0. The destination register D5 is a dummy and the desired data is stored in the modified address register A0.

The operations of instruction 4 are similar to those of instruction 3. Instruction 4 includes a multiple unsigned multiply operation (4a), which forms another set of squared error terms. Instruction 4 also includes an extended arithmetic logic unit operation (4b), which is a rotate and add operation the same as instruction 3b. In this case a second squared error term Sq_ErrB stored in data register D4 is rotated 16 bits and added to the most significant bits of a running sum MSE_SumB stored in data register D1. The global address unit operation loads a word "w" of data from the address stored in address register A8 into data register D5. This operation loads the preceding block data into data register D5, which is subtracted during instruction 1a of the next cycle through the loop kernel. The local address unit operation (4d) is an address unit arithmetic operation using pre-addition of the index register to modify the base address register. This adds the second squared error term LX_SqErr1 stored in index register X1 to the running sum stored in address register A0. Note that the destination register D5 is a dummy and the global address unit load operation aborts this local address unit load operation. However, this is of no consequence because the desired data is stored in the modified address register A0.

Instruction 5 includes only address unit operations. The global address unit loads index register X0 with a zero extended half word from the first byte (byte 0) of data register D4. This operation loads a squared error term formed during instruction 3a into the index register. The local address unit performs an address arithmetic operation incrementing a running sum stored in address register A0 by a third squared error term. Note that the data stored in data register D5 is not used.

Instruction 6 includes only a global address unit operation. The global address unit loads index register X1 with a zero extended half word from the second byte (byte 1) of data register D4. This operation loads the other squared error term formed during instruction 3a into the index register.

Instruction 7 includes only address unit operations. The global address unit loads index register X0 with a zero extended half word from the first byte (byte 0) of data register D2. This operation loads a squared error term formed during instruction 4a into the index register. The local address unit performs an address arithmetic operation incrementing a running sum stored in address register A0 by a first squared error term.

This loop kernel assumes use of hardware loop logic 720 for control of the iterations necessary to form the summation. This may involve two nested loops as mathematically implied in the double summation or some form of unrolled loop that traverses the same terms. Note that this loop kernel also presupposes that the data terms are properly loaded in memory accessible by local address unit 620, that is all the data is stored in the corresponding memories. Additional outer loop operations handle the case where the number of elements in the summation is too large to be stored in the corresponding memories. Some wrap up operations complete the mean squared error computation. The two running sums stored in data register D1 and address register A0 are added to form the final summation. This summation is divided by the number of elements to determine the final mean squared error. However, since this loop kernel forms the most often executed portion of the program, efficiency at this point is most critical.

Median filtering is another technique widely used in image processing. Median filtering is a nonlinear signal processing technique useful in image noise suppression. Each input pixel is replaced with the median value pixel within a block surrounding the input pixel. It is known to employ a 3 pixel by 3 pixel block surrounding the input pixel at the center. Median filtering does not effect step functions or ramp functions in the image data. However, median filtering is very effective against discrete impulse noise, especially single pixel noise. Real time implementation of median filtering requires comparisons of each 3 by 3 pixel block at the pixel input rate.

FIG. 48 illustrates a median filter algorithm suitable for use by multiprocessor integrated circuit **100**. This algorithm operates separately on each column of the 3 by 3 block of pixels having the current pixel at the center. The comparisons for each column then determine the median value. In the example described in detail below, four 3 by 3 blocks of 8 bit pixels are processed simultaneously using multiple arithmetic logic unit operations. When moving to the next adjacent 3 by 3 pixel block, the column comparisons for the two overlapping columns are retained. The new comparison values for the new third column are found, and then employed in determining the new median. This technique permits reduction in the determination of the column comparisons. The algorithm advantageously employs conditional operations to eliminate branches and their corresponding pipeline delay slots.

FIG. 48*a* illustrates the processing of each column of the 3 by 3 block. This processing makes comparison of the pixel values of each of the three pixels in the column. FIG. 48*a* illustrates the comparisons for column **0**, but the comparisons for columns **1** and **2** are identical. Comparison **1051** determines the minimum and the maximum of $Pixel_{00}$ and $Pixel_{01}$. The maximum of this comparison is passed to comparison **1051**, which determined the minimum and the maximum of this maximum and $Pixel_{02}$. The maximum of comparison **1052** is the maximum of the column, designated Max**0**. Comparison **1053** determines the minimum and maximum of the minimums of comparisons **1051** and **1052**. The maximum of comparison **1053** is the median of the column, designated Med**0**. The minimum of comparison **1053** is the minimum of the column designated Min**0**. As noted above, this same set of comparisons is applied to the pixel values of column **1** yielding Max**1**, Med**1** and Min**1** and to the pixel values of column **2** yielding Max**2**, Med**2** and Min**2**.

FIG. 48*b* illustrates the processing of the respective column maximum values Max**0**, Max**1** and Max**2**. Comparison **1060** determines the minimum of Max**0** and Max**1**. Note that the maximum of comparison **1060** is discarded. Comparison **1061** determines the minimum of the minimum result of comparison **1060** and Max**2**. The maximum of comparison **1061** is discarded and the minimum is designated MinMax. The value of MinMax is the minimum of the column maximum values.

FIG. 48*c* illustrates the processing of the respective column minimum values Min**0**, Min**1** and Min**2**. Comparison **1062** determines the maximum of Min**0** and Min**1**. Note that the minimum of comparison **1062** is discarded. Comparison **1063** determines the maximum of the maximum result of comparison **1062** and Min**2**. The minimum of comparison **1063** is discarded and the maximum is designated MaxMin. The value of MaxMin is the maximum of the column minimum values.

FIG. 48*d* illustrates the processing of the respective column median values Med**0**, Med**1** and Med**2**. Comparison **1064** determines the minimum and maximum of Med**0** and Med**1**. Comparison **1065** determines the minimum of the maximum result of comparison **1064** and Med**2**. Note that the maximum determined by comparison **1065** is discarded. Comparison **1066** determines the maximum of the minimum of comparison **1064** and the minimum of comparison **1065**. This value designated MedMed is the median of the column median values. Note that the minimum value of comparison **1066** is discarded.

FIG. 48*e* illustrates the process of determining the block median from MaxMin, MinMax and MedMed. Comparison **1067** finds the minimum and maximum of MaxMin and MinMax. Comparison **1068** determines the minimum of the maximum of comparison **1067** and MedMed. The maximum determined by comparison **1068** is discarded. Comparison **1069** finds the maximum of the minimum of comparison **1068** and the minimum of comparison **1067**. This value designated Median is the median value of the 3 by 3 block of pixels. Note that the minimum determined by comparison **1069** is discarded.

Below are the instructions of a loop executing this median filter algorithm. Note that instructions 1 to 9 generally perform the column comparison processes illustrated in FIG. 48*a* for column **2** of the block, the last column. In this example it is assumed that two column comparisons have already been made and are stored for use. This would be the case if the algorithm were used repeatedly for an entire row of the image data. For the first columns of each row, the steps of instructions 1 to 9 must be repeated for column **0** and colunm **1**. Instructions 10 to 13 perform the column maximum comparison processes illustrated in FIG. 48*b*. Instructions 14 to 17 perform the column minimum comparison processes illustrated in FIG. 48*c*. Instructions 18 to 24 perform the column median comparison processes illustrated in FIG. 48*d*. Lastly, instructions 25 to 31 perform the formation of the median processes illustrated in FIG. 48*e*.

```
1a. Dummy=mc Pack0–Pack1
1b. ||*(G_Col2SortAddr+=[3])=BlockMed
2a. TmpMax=@MF & Pack0|~@MF & Pack1
2b. || Out1=b*(G_Col2SortAddr+1)
3a. TmpMin=~@MF & Pack0|@MF & Pack1
3b. || Out2=b*(G_Col2SortAddr+2)
3c. ||*(L_OutAddr+LX_Tile1Index)=b Out1
4a. Dummy=mc TmpMax–Pack2
4b. || Out3=b *(G_Col2SortAddr+3)
4c. ||*(L_OutAddr+LX_Tile2Index)=b Out2
5a. Max2=@MF & TmpMax|~@MF & Pack2
5b. || Out0=b*G_Col2SortAddr
5c. ||*(L_OutAddr+LX_Tile3Index)=b 0ut3
6a. TmpMed=~@MF & TmpMax|@MF & Pack2
6b. ||*(G_Col2SortAddr–=[3])=Max2
6c. ||*L_OutAddr++=[b] Out0
7a. Dummy=mc TmpMin–TmpMed
7b. || Max0=*G_Col0SortAddr
8a. Med2=@MF & TmpMin|~@MF & TmpMed
8b. || Max1=*G_Col1SortAddr
9a. Min2=~@MF & TmpMin|@MF & TmpMed
9b. ||*(G_Col2SortAddr+[1])=Med2
10a. Dummy=mc Max0–Max1
10b. ||*(G_Col2SortAddr+[2])=Min2
11a. TmpMin=~@MF & Max0|@MF & Max1
11b. || Max2=*G_Col2SortAddr
12a. Dummy=mc Max2–TmpMin
12b. || Min0=*(G_Col0SortAddr+[2])
13a. MinMax=~@MF & Max2|@MF & TmpMin
13b. || Min1=*(G_Col1SortAddr+[2])
14a. Dummy=mc Min0–Min1
14b. ||*(G_Col1SortAddr+[3])=MinMax
15a. TmpMax=@MF & Min0|~@MF & Min
15b. || Min2=*(G_Col2SortAddr+[2])
16a. Dummy=mc Min2–TmpMax
16b. || Med0=*(G_Col0SortAddr+[1])
```

17a. MaxMin=@MF & Min2|~@MF & TmpMax
17b. || Med1=*(G_Col1SortAddr+[1])
18a. Dummy=mc Med0–Med1
18b. ||* (G_Col0SortAddr+[3])=MaxMin
19a. TmpMax=@MF & Med0|~@MF & Med1
19b. || Med2=*(G_Col2SortAddr+[1])
20a. TmpMin=~@MF & Med0|@MF & Med1
20b. || In0=b*(G_inputRow2Addr+=1)
21a. Dummy=mc Med2–TmpMax
21b. || In1=b*(G_InputRow2Addr+GX_Tile1Index)
21c. ||*L_packedRow2Addr++=b In0
22a. TmpMedB=~@MF & Med2|@MF & TmpMax
22b. || In2=b*(G_InputRow2Addr+GX_Tile2Index)
22b. ||*L_packedRow2Addr++=b In1
23a. Dummy=mc TmpMedB–TmpMin
23b. || In3=b*(G_InputRow2Addr+GX_Tile3Index)
23c. ||*L_packedRow2Addr++=b In2
24a. MedMed=@MF & TmpMedB|~@MF & TmpMin
24b. || MinMax=*(G_Col1SortAddr+[3])
25a. Dummy=mc MinMax–MedMed
25b. || NewCol1SortAddr=G_Col2SortAddr
25c. ||*L_PackedRow2Addr++=b In3
26a. TmpMaxB=@MF & MinMax|~@MF & MedMed
26b. || MaxMin=*(G_Col0SortAddr+[3])
27a. TmpMin=~@MF & MinMax|@MF & MedMed
27b. || NewCol2SortAddr=G_Col0SortAddr
28a. Dummy=mc MaxMin–TmpMaxB
28b. || G_Col2SortAddr=NewCol2SortAddr
29a. TmpMedB=~@MF & MaxMin|@MF & TmpMaxB
29b. || NewCol0SortAddr=G_Col1SortAddr
29c. || Pack2=*(L PackedRow2Addr–[1])
30a. Dummy=mc TmpMin–TmpMedB
30b. || G_Col0SortAddr=NewCol0SortAddr
30c. || Pack1=*L_PackedRow1Addr++
31a. BlockMed=@MF & TmpMin|~@MF & TmpMedB
31b. || G_Col1SortAddr=NewCol1SortAddr
31c. || Pack0=*L_PackedRow0Addr++

Table 60 lists proposed data register assignments for implementing this example of a median filter algorithm.

TABLE 60

Data Register	Variable Name	Data Assignment
D1	Pack0	packed column 2 row 0 pixels
	Max0	packed column 0 maximum pixels
	Med0	packed column 0 median pixels
	Min0	packed column 0 minimum pixels
	NewCol1SortAddr	temporary for address pointer swap
D2	Pack1	packed column 2 row 1 pixels
	Max1	packed column 1 maximum pixels
	Med1	packed column 1 median pixels
	Min1	packed column 1 minimum pixels

TABLE 60-continued

Data Register	Variable Name	Data Assignment
	MedMed	packed median of column medians
	NewCol2SortAddr	temporary for address pointer swap
D3	Pack2	packed column 2 row 2 pixels
	Med2	packed column 2 median pixels
	Min2	packed column 2 minimum pixels
	MaxMin	packed maximum of column minimums
D4	MinMax	packed minimum of column maximums
	TmpMax	packed intermediate maximums
	TmpMedB	packed intermediate medians
D5	TmpMin	packed intermediate minimums
D6	Max2	packed column 2 maximum pixels
	TmpMaxB	packed intermediate maximums
	TmpMed	packed intermediate medians
	BlockMed	fianl packed block medians
	Out1	block B median pixel
	Out2	block C median pixel
	Out3	block D median pixel
	In0	input block A pixel
	In1	input block B pixel
	In2	input block C pixel
	In3	input block D pixel
	NewCol0SortAddr	temporary for address pointer swap
D7	Dummy	unused result
	Out0	block A median pixel

As shown in Table 60, more than one variable is assigned to each data register. The complexity of the algorithm requires this reassignment of the data registers. Note that several of the variables are listed as packed variables. This algorithm operates on 4 blocks of eight bit pixels simultaneously employing multiple arithmetic. A packed variable is divided into 4 sections as follows:

|block A pixel||block B pixel||block C pixel||block D pixel||

Packing the variables in this way speeds processing because four pixels may be handled during each arithmetic logic unit operation and fewer memory loads and stores are required.

Table 61 lists proposed address register assignments for implementing this example of the median filter algorithm.

TABLE 61

Address Register	Variable Name	Data Assignment
A0	L_PackedRow0Addr	packed row n input pointer
A1	L_PackedRow1Addr	packed row n+1 input pointer
A2	L_PackedRow2Addr	packed row n+2 input pointer
A3	L_OutAddr	output pointer
A8	G_Col2SortAddr	pointer to sorted column 2 data

TABLE 61-continued

Address Register	Variable Name	Data Assignment
A9	G_InputRow2Addr	pointer to unpacked row n+2
A10	G_Col1SortAddr	pointer to sorted column 1 data
A11	G_Col0SortAddr	pointer to sorted column 0 data

Table 62 lists proposed index register assignments for implementing this example of the median filter algorithm.

TABLE 62

Index Register	Variable Name	Data Assignment
X0	LX_Tile1Index	pitch between blocks A and B
X1	LX_Tile2Index	pitch between blocks A and C
X2	LX_Tile3Index	pitch between blocks A and D
X9	GX_Tile1Index	pitch between blocks A and B
X10	GX_Tile2Index	pitch between blocks A and C
X11	GX_Tile3Index	pitch between blocks A and D

All the comparisons are made in a manner not requiring branches. This substantially reduces the time to execute the algorithm due to the elimination of pipeline delay slots. These comparisons used conditional operations based upon the expanded state of multiple flags register **211**. Such conditional operations permit selection of either the lesser or the greater of two sets of packed values following a subtraction to set multiple flags register **211**.

Instructions 1 to 9 perform the column comparison processes illustrated in FIG. **48***a*. Instruction 1a forms the difference between two sets of packed pixels. These are the top and center rows of column **2** of the 3 by 3 block. As noted, the actual value of the difference is unimportant for this algorithm and so is designated Dummy. The "mc" mnemonic indicates a multiple operation that stores the respective carry bits in multiple flags register **211**. This example operates on pixels of 8 bits, thus arithmetic logic unit **220** is divided into four sections of 8 bits each. This is accomplished by setting both the "Msize" field and the "Asize" field of status register **210** to "100". Thus each packed variable Pack**0** and Pack**1** include a pixel from an A, a B, a C and a D block. Instruction 1b is a store operation controlled by global address unit **610** that temporarily stores packed block median data from the prior loop at the global column **2** sort address designated by G_Col2SortAddr as incremented by an offset value of 3 as scaled via index scaler **614** by the data size. Since this is a word access the scaling is three bit positions. The instruction format indicates that G_Col2SortAddr is pre-incremented and modified.

Instruction 2a merges the maximums of the packed column **0** and column **1** pixels. If Pack**0**-Pack**1**>**0** and thus Pack**0**>Pack**1** for any of the blocks A, B, C or D, then instruction 1a generates a carry/borrow signal of "1". Multiple flags register **211** stores this "1". During instruction 2a this "1" is expanded in expander **238** to "11111111" (@MF). Thus the OR of instruction 2a returns the value from Pack**0**. Alternatively, if Pack**0**-Pack**1**<**0** and thus Pack**0**<Pack**1**, then instruction 1a generates a carry/borrow signal of "0". Multiple flags register **211** stores this "0" until instruction 2a, when expander **238** expands it to "00000000" (~@MF). Thus the OR of instruction 2a returns the value from Pack**1**. Thus TmpMax stores the block wise maximums of rows **0** and **1** of column 2 of the blocks A, B, C and D. This completes determination of the maximum of comparison **1051**. Instruction 2b loads the median value of block A from the prior loop stored in one more than the global column **2** sort address into a data register employing global address unit **610**. The "b" mnemonic indicates that this is a byte load operation.

Instruction 3a is the inverse of instruction 2a. Note that the @MF term in instruction 3a is of the opposite sense in the two halves of the OR statement than that of instruction 2a. Instruction 3a uses the carry/borrow data stored in multiple flags register **211** and expander **238** to select the minimums of the packed column **2** pixel values of Pack**0** and Pack**1**. This completes determination of the minimum of comparison **1051**. Instruction 3b is a global byte load operation of the block B median pixel into a data register. Instruction 3c is a byte memory store operation. The data stored in data register D**6** (Out**1**) is stored in the memory location having an address equal to the sum of the output pointer L_OutAddr and the n+1 packed row pointer LX_Tile1Index.

Instruction 4a is another subtraction setting carry/borrow bits of multiple flags register **211**. In this case the difference is between the packed temporary maximums and the packed row **2** data. This begins comparison **1051**. Instruction 4b is a global address unit byte load of the block D median pixel stored at address G_Col2SortAddr plus **2**. Instruction 4c is a local address unit byte store of the block B median pixel.

Instruction 5a is similar to instruction 2a. This instruction determines and merges block wise the maximums of TmpMax and the row **3** data stored in Pack**2** using the carry/borrow data stored in multiple flags register **211**. These merged maximums are stored in Max**2**. Instruction 5b is a global address unit byte load of the block A medial pixel. Instruction 5c is a local address unit byte store of the block D median pixel.

Instruction 6a is similar to instruction 3a. This instruction determines and forms a block wise merge of the minimums of TmpMax and the row **3** data stored in Pack**2** using the carry/borrow data still stored in multiple flags register **211**. These merged minimums are stored in TmpMin. Instruction 6b is a global address unit store of the Max**2** data formed in instruction 5a. This completes comparison **1052**. The instruction mnemonic indicates that global address register G_Col2SortAddr is pre-decremented and modified by the offset value **3** as scaled to the data size in index scaler **614**. instruction 6c is a local address unit store of the median pixel value of block A at the local output pointer address stored in L_OutAddr. This address register is pre-incremented by **1**.

Instruction 7a forms a difference to set the carry/borrow signals in multiple flags register **211**. As in the case of instructions 1a and 4a the actual difference in discarded. This subtraction begins comparison **1053**. Instruction 7b loads the packed column **0** maximum pixels via global address unit **610** from the global column **0** sort address.

Instruction 8a determines the maximum of comparison **1053**. This result is the column median Med**2**. Instruction 8b loads the packed column **1** maximum pixels via global address unit **610** from the global column **1** sort address.

Instruction 9a determines the minimum of comparison **1053**. This result is the column minimumMin**2**. Instruction 9b stores the packed column medians Med**2** into memory at the global column **2** sort address plus **1** scaled to the data size.

Instructions 10 to 13 perform the column maximum comparison processes illustrated in FIG. **48***b*. This involves a comparison of the column maximum pixels for the three column, retaining only the minimum of these column maxi-

mums. Instruction 10a forms the difference of Max0 and Max1, setting multiple flags register **211** for the minimum determination in instruction 11. This begins comparison **1060**. Instruction 10b stores the packed column 2 minimums to memory via global address unit **610**.

Instruction 11a determines the block wise minimums of the column **0** and column **1** maximums. As previously described, this determination is made from the expanded carry/borrow signals stored in multiple flags register **211**. This produces TmpMin and completes comparison **1060**. Instruction 11b loads the packed column **2** maximums from memory via global address unit **610**.

The subtraction of instruction 12a begins comparison **1061**. This subtraction sets multiple flags register **211** based upon the carry/borrow output. This begins comparison **1061**. Instruction 12b loads the packed column **0** minimums from memory via global address unit **610**.

Instruction 13a completes comparison **1061**. MinMax is set to the minimum of the respective column maximums for each block A, B, C and D. Instruction 13b loads the packed column **1** minimums from memory via global address unit **610**.

Instructions 14 to 17 perform the column minimum comparison processes illustrated in FIG. **48***c*. Instructions 14a and 15a form the maximums of the packed column **0** and column **1** minimums. This performs comparison **1062**. Instruction 16a and 17a perform comparison **1063** between the maximum of comparison **1062** and the column **2** minimums. Instruction 14a stores the packed minimum of the column maximums MinMax formed instruction 13a via global address unit. Instructions 15b, 16b and 17b load the column **2** minimums Min2, the column **0** medians and the column **1** medians, respectively, via global address unit **610**.

Instructions 18 to 24 perform the column median comparison processes illustrated in FIG. **48***d*. Instructions 18a, 19a and 20a perform comparison **1064**. Instruction 19a determines the maximums of the column **0** and column **1** medians. Instruction 20a determines the minimums of the column **0** and column **1** medians. Instruction 18b stores the MinMax results of instruction 17a in memory via global address unit **610**. Instruction 19a loads the column **2** packed median data Med2. Instruction 20a employs global address unit **610** to load a byte of block A pixel data. This begins a process of rearranging data to be in the desired packed column format for the next loop.

Instructions 21a and 22a perform comparison **1065**. The result is TmpMedB, the packed column temporary median values. Instruction 21b loads the pixel data of block B via Global address unit **610**. Instruction 21c stores the byte of pixel data of block A via local address unit **620**. Instruction 22b loads a byte of block C pixel data employing Global address unit **610**. Instruction 22c employs local address unit **620** to store the byte of block B pixel data.

Instructions 23a and 24a perform comparison **1066**. The result is MedMed, the block wise packed median of the column medians. Instruction 23b performs a block load of block D pixel employing Global address unit **610**. Instruction 23c stores a byte of the block C pixel data using local address unit **620**. Instruction 24b loads the packed minimums of column maximum MinMax employing Global address unit **610**.

Instructions 25 to 31 perform the formation of the median processes illustrated in FIG. **48***e*. Instructions 25a, 26a and 27a perform comparison **1067**. Instruction 26a determines the maximums of MinMax and MedMed. Instruction 27a determines the minimums of MinMax and MedMed. Instruction 25b begins the process of realigning the address pointers for the next loop by setting a temporary value NewCol1SortAddr equal to the prior column **2** global sort address G_Col2SortAddr. Instruction 25c stores a byte of pixel block D data using local address unit **620**. Instruction 26b loads the maximum of the column minimums MaxMin via global address unit **610**. Instruction 27b continues realigning the address pointers for the next loop by setting a temporary value NewCol2SortAddr equal to the prior column **0** global sort address G_Col0SortAddr.

Instructions 28 and 29 perform comparison **1068**. Instruction 28a is a subtraction setting multiple flags register **211**. Instruction 29a determines the minimums of MaxMin and the temporary maximum TmpMaxB from instruction 26a. Instruction 28b continues the pointer rotation by setting the global column **2** sort address equal to the new column **2** sort address set in instruction 27b. Instruction 29b continues the pointer rotation by setting a temporary value NewCol0SortAddr equal to the global column **1** sort address. Instruction 29c loads the packed column **2** pixels using local address unit **620**.

Instructions 30 and 31 perform comparison **1069** and determine the block medians BlockMed. Instruction 30a is the subtraction setting multiple flags register **211**. Instruction 31a determines the maximum of comparison **1069**, which is the block medians BlockMed. Instruction 30b continues the pointer rotation by setting the global column **0** sort address equal to the new column **0** sort address NewCol0SortAddr set in instruction 29b. Instruction 30c loads the packed column **1** pixels via local address unit **6320**. Instruction 31b completes the pointer rotation by setting the global column **1** sort address equal to the new column **1** sort address NewCol1SortAddr set in instruction 25b. Instruction 31c loads the packed column **0** pixels using local address unit **620**.

Several other programming techniques are supported by the above described hardware of the digital image/graphics processors **71**, **72**, **73** and **74**. These include: employing the write priority of Table 51 to perform single instruction "if . . . then . . . else . . . " operations; mixed conditional operations; and zero overhead hardware branches with conditional test for zero.

An example of a single instruction "if . . . then . . . else . . . " operation is listed below. Note that a condition of status register **210** must be set before the single instruction "if . . . then . . . else . . . " operation can be performed. In this example the condition is Data=0.

1. Data=Data

2a. Zero_Run=Zero_Run+1

2b. || Zero_Run=[nz] A15

Table 63 shows an example of the register assignments for this program code example.

TABLE 63

Register	Variable Name	Data Assignment
D6	Data	test data
D7	Zero_Run	number of consecutive examples of Data = 0

Instruction 1 doesn't change the contents of the data register D6. This instruction does cause the status register **210** to set the negative "N", carry "C", overflow "V" and zero "Z" status bits based upon the result of arithmetic logic unit **230**. Though instruction 1 does not change the contents of data register D6, this instruction may still set the negative status "N" if D6<0 or the zero status "Z" if D6=0.

Instruction 2 performs the "if . . . then . . . else . . . " operation. If Data ≠0, then the condition of instruction 2b is

true. Thus Hex "0" is moved from the zero value address register A15 to data register DT. Thus if Data ≠0, then the number of consecutive zeros is set to zero. Note that according to Table 51 this address unit operation has priority over the data unit operation. Thus if the condition is true, the register to register move operation occurs and the data unit operation aborts. Only if Data=0 does the data unit operation of instruction 2a increment Zero_Run. Thus instruction 2 performs "if Data ≠0, then Zero_Run=0, else Zero_Run=Zero_Run+1."

Below is a second example of a single instruction "if . . . then . . . else . . . " operation. This example uses a compare for the conditional operation.

1a. Dummy=Data1−Data2

1b. || Dummy=Dummy

2a. Data1=Data2

2b. || Data1=[lt] A15

Table 64 shows an example of the register assignments for this program code example.

TABLE 64

Register	Variable Name	Data Assignment
D5	Data2	second data element
D6	Data1	first test element
D7	Dummy	dummy register not used

The subtraction of instruction 1a effectively compares the numbers Data1 and Data2. If Data1<Data2, then the negative "N" status is set in status register **210**. If Data1=Data2, then the zero "Z" status is set. Lastly, if Data1>Data2, then neither of these bits are set. This example illustrates another use of the write priority rules of Table 51. The unconditional address unit register move of Dummy to Dummy, protects Dummy from change while permitting status register **210** to be set based upon the arithmetic logic unit result. The register to register move aborts storing the arithmetic logic unit result. If instruction 1a sets the negative "N" status bit, the instruction 2b sets Data1 equal to zero. Otherwise instruction 2a sets Data1 equal to Data2. Thus instruction 2 performs the operation "if Data1<Data2, then Data1=0, else Data1=Data2."

This same sequence can perform other "if . . . , then . . . , else . . . " operations. The sequence requires a first arithmetic logic unit operation to set status register **210**. A following instruction performs the "if . . . , then . . . , else . . . " operation. This instruction includes a conditional data unit register move or load operation based upon at least one condition set in the first instruction. The "else" operation is a data unit operation having the same destination as the register move or load operation.

It is possible to set conditions for conditional operations based upon plural tests. In a first example two tests are ANDed.

1. Dummy=D1−D2
2. Dummy=[z] D3−D4
3. BR=[z] IPRS

Instruction 1 sets the zero "Z" status bit if D1=D2. Instruction 2 is conditional based upon the zero "Z" status bit. If the zero "Z" status bit is "0", then instruction 2 is not performed and no status bits are changed. If the zero "Z" status bit is "1", then instruction 2 is performed, and the status bits are set based upon the difference of D3 and D4. Instruction 3 is a conditional subroutine return. Note writing to BR changes only program counter PC **701** and does not change instruction pointer return from subroutine IPRS **704**. Writing to program counter PC **701** places the previous address stored in program counter PC **701** into instruction pointer return from subroutine IPRS **704**. This subroutine return is conditional on the zero "Z" status bit. Thus the subroutine return occurs only if D1=D3 and D3=D4 is true. Note that this conditional operation could also be based upon the negative "N" status bit, the carry "C" status bit or the overflow "V" status bit. This conditional operation could also be based upon any of the compound conditions listed in Table 41.

Instruction 3 above is only an example of a conditional instruction. Any desired conditional instruction based upon the selected status bit or bits could be placed here. This could be an arithmetic logic unit operation, a register load operation, a memory store operation of a register to register move operation. Other program flow control operations such as a branch or call are also possible. This conditional instruction may be an "if . . . , then . . . , else . . . " operation such as described above.

In a second example two tests are ORed. This is listed below.

1. Dummy=D1−D2
2. Dummy=[nz] D3−D4
3. BR=[z] IPRS

Instruction 1 sets the zero "Z" status bit if D1=D2. Instruction 2 is conditional based upon the inverse of the zero "Z" status bit (not zero). If the zero "Z" status bit is "1", that is D1=D2, then instruction 2 is not performed and no status bits are changed. If the zero "Z" status bit is "0", then instruction 2 is performed, and the status bits are set based upon the difference of D3 and D4. Instruction 3 is a conditional subroutine return. As stated above, instruction 3 could be any conditional instruction based upon the zero "Z" status bit. If D1=D2, the zero "Z" status bit is "1" and instruction 2 aborted without changing any status bits. Thus instruction 3 executes. If D1≠D2, then instruction 2 executes and the zero "Z" status bit is set to "1" if D3=D4. So instruction 3 executes if D1=D2 OR D3=D4. Note that instructions 2 and 3 could be based upon any single status bit or any compound condition so long as they are logical inverses.

This technique may also be used for mixed conditions. An example of this is listed below.

1. Dummy=D1−D2
2. Dummy=[u.z] D3−D4
3. BR=[le] IPRS

Instruction 1 sets the zero "Z" status bit if D1=D2. The "u.z" mnemonic of instruction 2 indicates this instruction is unconditional and that the zero "Z" status bit is protected form change by this operation. Thus the negative "N" status bit is set if D3<D4, but the zero "Z" status bit is not set if D3=D4. Instruction 3 is conditional based upon a "less than or equal" condition. As seen in Table 41, this condition is formed by (N&~V)|(~N&V)|Z. Thus the subroutine return is taken if D1=D2 and D3<D4. This is not the only mixed conditional operation feasible. Any compound condition listed in Table 51 (positive p, lower than or same is, higher than hi, less than lt, less than or equal le, greater than or equal ge or greater than gt) can be used for instruction 3 of this example. Note as previously stated, any conditional instruction can be substituted into instruction 3 for the conditional subroutine return of this example.

Conditional "hardware branching" using the zero overhead loop logic were described above in conjunction with the description of the zero-overhead loop logic. Below is an example of a character search routine using a single instruc-

tion with conditional hardware branching. This character search routine makes four byte comparisons per loop using multiple arithmetic.

1. Match=Hex "F0F0F0F0"
2. LE2=Loop2_End
3. LRS2=0
4. LRSE1=511
5. LS2=Loop2_Start
6. Data=*(A0=DBA)

Loop1_Start:

Loop1_End:

Loop2_End:

7a. Dummy=mz Data−Match

7b. || LS2=MF

7c. || Data=*A0++

8. . . .

Loop2_Start:

10. A0=A0−4

11. . . .

Instruction 1 loads the pattern to be matched into a register. In this case the pattern is one byte long and is repeated 4 times when stored. Instruction 2 sets the loop end address LE2 to the single instruction loop address. Instruction 3 writes the count "0" into both the loop count register LC2 and the loop reload register LR2. Instruction 4 is a single instruction loop fast initialization. Writing "511" to LRSE1 writes the loop count 511 into both loop count register LC1 and loop reload register LR1, loads the value PC+3 into both the loop start register LS1 and the loop end register LE1, and sets the program flow control unit loop control register LCTL to associate loop end register LE1 with loop count register LC1. Instruction 5 loads the loop start register LS2 with the branch address. Lastly, instruction 6 initializes address pointer A0 and loads the first word to be searched from the memory location pointed to by address pointer A0.

Instruction 7 forms the single instruction loop. Instruction 7a forms the difference between the data loaded in instruction 6 and the reference data Match. The "mz" mnemonic indicates that instruction 7a is a multiple instruction and that the zero status bits are stored in multiple flags register 211. Note that the "Msize" field of data register DO must have been set to the desired data size. This sets the multiple flags register 211 according to the multiple differences. Instruction 7b loads loop count register LC2 with the data stored in multiple flags register 211. Note that multiple flags register 211 was set by the difference Data-Match of the prior loop. Instruction 7c modifies the address register A0 to point to the next data, and loads this data for the next loop. Instruction 8 starts the portion of the program that handles the case if no match is found before 512 loops recorded by loop count register LC1. Instruction 10 starts the portion of the program that handles the case when a match is found. Note that this instruction is at the address corresponding to Loop2_Start stored in loop start register LS2.

While none of the four bytes of Data and Match are identical, each difference is nonzero. Thus multiple flags register 211 stores all zeros for the four sections. This status result is loaded into loop count register LS2. With loop count register LS2 equal to zero, and loop count register LC1 not equal to zero: loop count register LC1, the outer loop, is decremented; loop count register LC2 is reloaded with the value of loop reload register LR2, which is zero; program counter 701 is loaded with the address stored in loop start register LS1, which is the address of the one instruction loop. Thus the instruction repeats.

The loop may end in two ways. First, loop count register LC1 may decrement to zero. In this case the program continues with instruction 8, the next following instruction. Second, the multiple difference may detect at least one match. In this event multiple flags register 211 is nonzero because at least one of the multiple differences is zero. When this nonzero result is loaded into loop count register LC2, the hardware loop logic branches to the second loop start address, which is Loop2_Start at instruction 10.

Instruction 10 subtracts 4 from address register A0. This resets address register A0 to the memory location having the match. As illustrated in FIG. 49, the program executes the single loop instruction 7 four times before the branch is taken. In FIG. 49 instruction slot 1070 does not detect a match, thus multiple flags register 211 stores "000". The global address operation of instruction slot 1070 stores a nonzero result in loop count register LC2 from the previous iteration of the loop. In instruction slot 1071 a match is found and at least one of the bits of multiple flags register 211 is not zero. The global address operation of instruction slot 1071 stores the zero multiple flags register 211 contents from the arithmetic operation of instruction slot 1070 in loop count register LC2. The global address operation of instruction slot 1072 stores the nonzero multiple flags register 211 contents from the arithmetic operation of instruction slot 1071 in loop count register LC2. There follows two delay slots, instruction slots 1073 and 1074, which occur because the global address operation executes at the beginning of the execute pipeline stage and two instructions are in the pipeline before the branch can be taken. During each of these instructions the hardware loop logic continues to load the single loop instruction due to the state of loop count register LC1. At instruction slot 1075 the branch is taken and the hardware loop logic branches to Loop2_Start. In instruction slot 1076 program counter 701 advances normally to the next memory address.

FIGS. 50, 51, 52 and 53 illustrate members of a family of hardware dividers. FIG. 50 illustrates the hardware in a divider that forms two bits of the quotient per iteration. FIG. 51 illustrates in a schematic form the data flow through the apparatus of FIG. 50. FIG. 52 illustrates the hardware in a divider that forms three bits of the quotient per iteration. FIG. 53 illustrates in schematic form the data flow in a divider that forms three bits of the quotient per iteration. Each of the members of this family of hardware dividers employs a conditional subtract and rotate algorithm. Each of the members of this family employs hardware parallelism to speed the division process.

FIG. 50 illustrates hardware divider 1100. Register 1101 stores the unsigned portion of the divisor, if the divisor is a signed number and sign latch 1102 stores the sign bit. If the divisor is unsigned, then register 1101 stores the entire divisor and sign latch 1102 stores a bit indicating a positive number. Register 1103 stores the unsigned portion of the numerator with sign latch 1104 storing the sign bit. If the numerator is unsigned, register 1103 stores the entire numerator and sign latch 1104 stores a bit indicating a positive number. Control sequencer 1130, which may be a state machine, controls loops of an iteration process with reference to a loop count stored in loop counter 1131. Control sequencer 1130 controls data flow via multiplexers 1117, 1118 and 1121 and forms two bits of the quotient each iteration. This quotient is stored in register 1105.

Hardware divider 1110 includes three full adders 1112, 1113 and 1114. These operate in parallel during the condi-

tional subtract and rotate operation. Those skilled in the art would realize that the numerator will generally have more bits than the denominator. The DIVI instruction discussed above provided for division of a 64 bit numerator by a 32 bit divisor and division of a 32 bit numerator by a 16 bit divisor. Hardware divider **1100** is suitable for either case with suitable capacity of registers and the full adders. In the preferred embodiment the numerator will have two times the number of bits of the divisor. Full adders **1112**, **1113** and **1114** operate on the full width of data stored in register **1101** and the most significant half of data stored in register **1103**. To prevent loss of data during carries (borrows), full adders **1112**, **1113** and **1114** should have one more bit than the number of bits of register **1101**.

Full adders **1112**, **1113** and **1113** operate in parallel during each iteration. Full adder **1112** subtracts the number stored in register **1101** from the most significant bits of the number stored in register **1103**, effectively subtracting the divisor from the most significant bits of the numerator/running remainder. Full adder **1113** subtracts the number stored in register **1101**, left shifted one place by shift left circuit **1141**, from the most significant bits stored in register **1103**. This effectively subtracts two times the divisor from the most significant bits of the numerator/running remainder. Full adder **1114** has two alternate operations. In an initial operation, control sequencer **1130** controls multiplexer **1117** to select the output from shift left circuit **1141** and multiplexer **1118** to select the output from register **1101**. Thus full adder **1114** adds the divisor to two times the divisor. The resultant of three times the divisor is stored in latch **1144**. During normal operation, control sequencer **1130** controls multiplexer **1117** to select the most significant bits of register **1103** and multiplexer **1118** to select the output of latch **1144**. Full adder **1114** then subtracts three times the divisor from the most significant bits of the numerator/running remainder.

Control sequencer **1130** controls the loop operation of hardware divider **1100**. Negative detectors **1122**, **1123** and **1124** determine if the subtractions performed by the respective full adders **1112**, **1113** and **1114** result in a negative difference. Based upon these determinations, control sequencer **1130** generates two bits of the quotient, which are stored in register **1105**, and controls multiplexer **1121**. Multiplexer **1121** selects either the original data in register **1103** or the resultant of one of full adders **1112**, **1113** or **1114** for storage in register **1103** depending upon the results of the negative determinations. Following each such storage operation, control sequencer **1130** controls register **1103** to shift left two places. Note that the storing the data selected according the negative detectors **1122**, **1123** and **1124** insures that no data is lost in this shift operation. Control sequencer **1130** repeats this operation a number of times as set by the loop count in loop counter **1131**. The quotient from register **1105** may be negated by negate circuit **1135** based upon the original sign bits stored in sign latches **1102** and **1103**. If needed, the remainder is stored in register **1103** and may be negated by negate circuit **1136** depending upon the original sign bits stored in sign latches **1102** and **1103**.

FIG. **51** illustrates in schematic form the data flow during operation of hardware divider **1100**. Initially the apparatus simultaneously forms the quantities D, 2D and 3D, where D is the divisor stored in register **1101**. These quantities may be formed using simultaneous addition blocks **1141**, **1143** and **1143**, respectively, employing the three full adders **1112**, **1113** and **1114** as shown in FIG. **51** with the results stored in corresponding latches. Addition block **1141** adds "0" and D to get D. Addition block **1142** adds "0" and D left shifted one place to get 2D. Addition block **1143** adds D and D left shifted one place to get 3D. Alternatively, only 3D need be formed by an adder (block **1143**) and stored as illustrated in FIG. **50** because the quantities D and 2D can easily be formed in real time during each iteration.

Next, hardware divider **1100** simultaneously forms the differences N(hi)–D, N(hi)–2D and N(hi)–3D using the three full adders **1112**, **1113** and **1114** in blocks **1151**, **1152** and **1153**, where N(hi) is the most significant bits of the numerator/running remainder stored in register **1103**. The results of these three trial subtractions determine the two bit partial quotient and the data to be recirculated as the numerator/running remainder. Simultaneous negative test blocks **1154**, **1155** and **1156** determine if the quantities N(hi)–D, N(hi)–2D and N(hi)–3D are negative. There are four possible results of these simultaneous negative tests. If N(hi)–D<0, then the two quotient bits V are "00" and N(hi) is recirculated (block **1161**). If N(hi)–D>0 and N(hi)–2D<0, and then the two quotient bits V are "01" and N(hi)–D is recirculated (**1162**). If N(hi)–2D>0 and N(hi)–3D<0, then the two quotient bits V are "10" and N(hi)–2D is recirculated (**1163**). Lastly, if N(hi)–3D>0, then the two quotient bits V are "11" and N(hi)–3D is recirculated (block **1164**). These results represent the four possible outcomes for the trial subtractions and the corresponding quotient and recirculation quantities.

The data within register **1103** is then left shifted by two places (block **1170**). As previously described, the selection of the recirculated data based upon the trial subtraction insures that no data is lost in this shift operation. Block **1170** also forms an OR of the shifted numerator/running remainder and V. Since the two least significant bit places have just been cleared by the left shift, this OR operation places the just calculated quotient bits into the least significant bits of register **1103**. Since the numerator has the same number of bits as the sum of the bits of the remainder and the quotient, this process permits the same register to initially hold the numerator, the running remainder and to hold the final remainder and quotient at the end of the process. Note that this same result can be achieved by shifting in the two bits of V during the left shift operation. This is similar to the manner of shifting data register **200***a* and multiple flags register **211** as illustrated in FIG. **44**, except that two bits are shifted in rather than only one. The loop count is incremented in block **1171**. If the loop count is not greater than 8 (block **1172**), then another iteration begins with simultaneous subtractions blocks **1151**, **1152** and **1153**. Note that the loop count of 8 is appropriate for a division of a 32 bit numerator by a 16 bit divisor yielding a 16 bit quotient. For the division of a 64 bit numerator by a 32 bit divisor yielding a 32 bit quotient a loop count of 16 is selected.

Two clean up operations occur following completion of the selected number of iterations. Block **1173** determines the sign of the quotient from an exclusive OR of the sign of the numerator and divisor. If the sign of the quotient is negative, then block **1174** forms the inverse of the computed quotient. In parallel is a determination of the sign of the remainder. Block **1175** determines if the numerator was less than zero. If the numerator was less than zero, then block **1176** forms the inverse of the computed remainder that is stored in register **1103**. In any case the division operation is complete and ended at exit block **1177**.

A hardware divider such as illustrated in FIG. **50** may be as useful as multiplier **220** illustrated in FIG. **5**. In the preferred embodiment a division operation employs similar data paths and instruction word formats as those used for multiplication. It is feasible to employ some of the adders used in the common Booth adder type multiplier circuit to

embody full adders **1112**, **1113** and **1114**. Thus the hardware divider would require few additional components.

FIG. **52** illustrates the major components of hardware divider **1100***a* that forms three bits of the quotient per iteration. Hardware divider **1100***a* includes register **1101**, sign latch **1102**, register **1103**, sign latch **1104**, control sequencer **1130** and loop counter **1131**, which are similar to the corresponding parts illustrated in FIG. **50**. Hardware divider **1110***a* includes seven full adders **1112**, **1113**, **1114**, **1115**, **1116**, **1117** and **1118**. These operate in parallel during the conditional subtract and shift operation. During the initial step, multiplexer **1154** supplies the divisor from register **1101** and the divisor from register **1101** left shifted via shift left circuit **1141** to full adder **1114**. Full adder **1114** thus forms three times the divisor, which is stored in latch **1144**. During the initial step, multiplexer **1156** supplies the divisor from register **1101** and the divisor from register **1101** left shifted two places via shift left circuits **1141** and **1142** to full adder **1116**, thus forming five times the divisor, which is stored in latch **1146**. During the initial step, multiplexer **1157** supplies the divisor from register **1101** left shifted via shift left circuit **1141** and the divisor from register **1101** left shifted two places via shift left circuits **1141** and **1142** to full adder **1117**. This forms six times the divisor, which is stored in latch **1147**. Also during the initial step, multiplexer **1158** supplies the divisor from register **1101** and the divisor from register **1101** left shifted three places via shift left circuits **1141**, **1142** and **1143** to full adder **1118**. Full adder **1118** then subtracts the divisor from eight times the divisor, forming seven times the divisor, which is stored in latch **1148**. During each iteration, full adders **1112**, **1113**, **1114**, **1115**, **1116**, **1117** and **1118** subtract respectively one times, two times, three times, four times, five times, six times and seven times the divisor stored in register **1101** from the most significant bits of register **1102**. Note that during each iteration multiplexers **1154**, **1156**, **1157** and **1158** select the numerator and the multiple of the divisor.

Control sequencer **1130** controls the loop operation of hardware divider **1100**. Negative detectors **1122**, **1123**, **1124**, **1125**, **1126**, **1127** and **1128** determine if the subtractions performed by the respective full adders **1112**, **1113**, **1114**, **1115**, **1116**, **1117** and **1118** result in a negative difference. Based upon these determinations, control sequencer **1130** generates three bits of the quotient. These three bits of the quotient are stored in the least significant bits of register **1103**. Note that register **1103** is shifted three bits each iteration, making room for the quotient bits. In other respects control sequencer **1130** of FIG. **52** operates like that previously described with regard to FIG. **50**. The quotient from the least significant bits of register **1103** may be negated by negate circuit **1135** based upon the original sign bits stored in sign latches **1102** and **1103**. If needed, the remainder stored in the most significant bits of register **1103** may be negated by negate circuit **1136** depending upon the original sign bits stored in sign latches **1102** and **1103**.

FIG. **53** illustrates schematically data flow within hardware divider **1100***a* illustrated in FIG. **52**. The divisor is stored in register **1101**, the numerator in register **1103** and the loop count limit in register **1131**. Initially the process uses seven full adders to compute seven multiples of the divisor. This is accomplished by simultaneous addition blocks **1201**, **1202**, **1203**, **1203**, **1204**, **1205**, **1206** and **1207**. Addition block **1201** forms 0+D=D; addition block **1202** forms 0+D<<1=2D; addition block **1203** forms D+D<<1=3D; addition block **1204** forms 0+D<<2=4D; addition block **1205** forms D+D<<2=5D; addition block **1206** forms D<<1+D<<2=6D; addition block **1207** forms D<<3−D=7D; where<<n is left shifting n places. Thus simultaneous addition blocks **1201**, **1202**, **1203**, **1203**, **1204**, **1205**, **1206** and **1207** form each multiple of D from 1 to 7. At least 3D, 5D, 6D and 7D are stored in latches for use each iteration. Note that D, 2D and 4D need not be stored in latches because these quantities can be easily formed from D stored in register **1101**.

Next the respective multiples of D are subtracted from the most significant bits of the numerator/running remainder stored in register **1103**. Simultaneous subtractions **1211**, **1212**, **1213**, **1214**, **1215**, **1216** and **1217** form the differences between N(hi) and D, 2D, 3D, 4D, 5D, 6D and 7D, respectively. As in simultaneously addition blocks **1201**, **1202**, **1203**, **1203**, **1204**, **1205**, **1206** and **1207** above, these simultaneous subtractions are formed using seven full adders. The results of these seven trial subtractions determine the three bit partial quotient and the data to be recirculated as the numerator/running remainder. Simultaneous negative test blocks **1221**, **1222**, **1223**, **1224**, **1225**, **1226** and **1227** determine if the quantities N(hi)−D, N(hi)−2D, N(hi)−3D, N(hi)−4D, N(hi)−5D, N(hi)−6D and N(hi)−7D are negative. There are eight possible results of these simultaneous negative tests. If N(hi)−D<0, then V="000" and N(hi) is recirculated (block **1231**). If N(hi)−D>0 and N(hi)−2D<0, and then V="001" and N(hi)−D is recirculated (block **1232**). If N(hi)−2D>0 and N(hi)−3D<0, then V="010" and N(hi)−2D is recirculated (block **1233**). If N(hi)−3D>0 and N(hi)−4D<0, then V="011" and N(hi)−3D is recirculated (block **1234**). If N(hi)−4D>0 and N(hi)−5D<0, then V="100" and N(hi)−4D is recirculated (block **1235**). If N(hi)−5D>0 and N(hi)−6D<0, then V="101" and N(hi)−5D is recirculated (block **1236**). If N(hi)−6D>0 and N(hi)−7D<0, then V="110" and N(hi)−6D is recirculated (block **1237**). If N(hi)−7D>0, then V="111" and N(hi)−7D is recirculated (block **1238**).

The data within register **1103** is then left shifted by three places (block **1241**). Block **1241** also forms an OR of the shifted numerator/running remainder and V. This OR operation places the just calculated three quotient bits into the least significant bits of register **1103**. Similarly to that discussed above in conjunction with block **1170** of FIG. **51**, this same result can be achieved by shifting in the three bits of V during the left shift operation.

The loop count is decremented in block **1242**. If the loop count has not reached zero (block **1243**), then another iteration begins with simultaneous subtractions blocks **1211**, **1212**, **1213**, **1214**, **1215**, **1216** and **1217**. Note that FIG. **52** illustrates decrementing the loop count from a set loop limit to zero rather than incrementing the loop count from 1 to a limit. Either of these techniques may be employed in hardware dividers of this type. If iterations are complete, then block **1244** representing a clean-up operation occurs. This process has been previously described in conjunction with blocks **1173**, **1174**, **1175** and **1176** of FIG. **51**. The division operation is complete and ended at exit block **1245**.

As previously mentioned, FIGS. **50**, **51**, **52** and **53** illustrate members of a family of hardware dividers. Each member of this family of hardware dividers employs 2^N-1 parallel full adders to form every trial subtraction from 1 to 2^N-1 times the divisor. N bits of the quotient and a running remainder are determined from the results of these trial subtractions. The quotient may be formed in a separate register. Alternatively, the quotient may be shifted into the emptied bits of the numerator/running remainder register. This takes advantage of the relationship between the number of bits of the numerator, final remainder and quotient. Table 65 illustrates the properties of members of this family of

hardware divider. Note that the DIVI instruction described above falls into the first member of this family, hardware divider **1100** illustrated in FIG. **50** the second member of this family and hardware divider **1100***a* illustrated in FIG. **52** the third member of this family.

TABLE 65

Quotient bits per iteration	Number of parallel adders	Number of iterations for 32/16	Number of iterations for 64/32
1	1	16	32
2	3	8	16
3	7	6	11
4	15	4	8
5	31	4	7
6	63	3	6
7	127	3	5
8	255	2	4
16	65535	1	2
32	4294967295	1	1

Table 65 illustrates a startling diminishing return to scale. If the number of bits per iteration is N, the then number of parallel full adders needed is 2^N-1. The greatest number of bits per iteration for practical devices in current semiconductor technology is probably 3 or 4. Current Booth re-coding multiply circuits may have 9 full adders. Thus 15 full adders for division is not unreasonable, particularly if the adders can be used for both hardware multiply and hardware divide. Use of additional hardware for divides of more than 4 bits per iteration is not currently economically feasible.

FIG. **54** illustrates an alternative embodiment of this invention. In FIG. **54** multiprocessor integrated circuit **101** includes master processor **60** and a single digital image/graphics processor **71**. Multiprocessor integrated circuit **101** requires less silicon substrate area than multiprocessor integrated circuit **100** and consequently can be constructed less expensively. Multiprocessor integrated circuit **101** is constructed using the same techniques as previously noted for construction of multiprocessor integrated circuit **100**. Because the width of each digital image/graphics processor matches the width of its corresponding memory and the associated portions of crossbar **50**, multiprocessor integrated circuit **100** may be cut between digital image/graphics processors **71** and **72** to obtain the design of multiprocessor integrated circuit **101**. Multiprocessor integrated circuit **101** can be employed for applications when the processing capacity of four digital image/graphics processors is not required.

Multiprocessor integrated circuit **101** is illustrated in FIG. **54** as part of a color facsimile apparatus. Modem **1301** is bidirectionally coupled to a telephone line for sending and receiving. Modem **1301** also communicates with buffer **1302**, which is further coupled the image system bus. Modem **1301** receives a fascimile signal via the telephone line. Modem **1301** demodulates these signals, which are then temporarily stored in buffer **1302**. Transfer controller **80** services buffer **1302** by transferring data to data memories **22**, **23** and **24** for processing by digital image/graphics processor **71**. In the event that digital image/graphics processor **71** cannot keep ahead of the incoming data, transfer controller **80** may also transfer data from buffer **1302** to memory **9**. Digital image/graphics processor **71** processes the image data of the incoming facsimile. This may include image decompression, noise reduction, error correction, color base correction and the like. Once processed, transfer controller **80** transfers image data from data memories **22**, **23** and **24** to video random access memory (VRAM) **1303**. Printer controller **1304** recalls the image data under control of frame controller **90** and supplies it to color printer **1305**, which forms the hard copy.

The apparatus of FIG. **54** can also send a color facsimile. imaging device **3** scans the source document. Imaging device **3** supplies the raw image data to image capture controller **4** that operates under control of frame controller **90**. This image data is stored in video random access memory **1303**. Note that the embodiment illustrated in FIG. **54** shares video random access memory **1303** for both image capture and image display in contrast to the embodiment of FIG. **1**, which uses separate video random access memories. Transfer controller **80** transfers this image data to data memories **22**, **23** and **24**. Digital image/graphics processor **71** then processes the image data for image compression, error correction redundancy, color base correction and the like. The processed data is transferred to buffer **1303** by transfer controller **80** as needed to support the facsimile transmission. Depending upon the relative data rates, transfer controller **80** may temporarily store data in memory **9** before transfer to buffer **1302**. This image data in buffer **1302** is modulated by modem **1301** and transmitted via the telephone line.

Note that the presence of an imaging device and a color printer in the same system permits this system to also operate as a color copier. In this event data compression and decompression may not be required. However, digital image/graphics processor **71** is still useful for noise reduction and color base correction. It is also feasible for digital image/graphics processor **71** to be programmed to deliberately shift colors so that the copy has different coloring than the original. This technique, known as false coloring, is useful to conform the dynamic range of the data to the dynamic range of the available print colors.

We claim:

1. A method of data processing comprising the steps of:

receiving a first data word of N bits;

receiving a second data word of N bits;

multiplying a first set of L bits of said first data word by a second set of L bits of said second data word, thereby obtaining a product having 2 L bits, where N is greater than L; and

forming a resultant data word of N bits having a third set of L bits corresponding to a most significant L bits of said product and a fourth set of M bits, where said fourth set of M bits does not include a least significant set of L bits of said product, where $N=L+M$ and $M \geqq L$.

2. The method of claim **1**, wherein:

said fourth set of M bits of said resultant data word are derived from at least some bits of at least one of said first data word other than said first set of L bits and said second data word other than said second set of L bits.

3. The method of claim **2**, wherein:

said fourth set of M bits of said resultant data word consists of bits of said first data word other than said first set of L bits.

4. The method of claim **1**, wherein:

said step of receiving said first data word comprises recalling said first data word from a first instruction specified one of a plurality of data registers;

said step of receiving said second data word comprises recalling said second data word from a second instruction specified one of said plurality of data registers; and

said method further comprises storing said resultant data word in a third instruction specified one of said plurality of data registers.

5. The method of claim 1, wherein:

said number of bits M is greater than said number of bits L.

6. The method of claim 1, wherein:

said number of bits L equals 16 bits.

7. The method of claim 1, wherein;

said number of bits L equals 8 bits.

8. A data processing apparatus comprising:

a first input bus of N bits;

a second input bus of N bits;

a multiplier having a first input of L bits connected to a first set of L bits of said first input bus, L being less than N, a second input of L bits connected to a second set of L bits of said second input bus, and a product output of 2 L bits producing a product of data that was supplied to said first and second inputs; and

an output bus of N bits coupled to said product output of said multiplier, said output bus including a first portion consisting of a most significant set of L bits of said product output of 2 L bits and a second portion of M bits, where said second portion of M bits does not include a least significant set of L bits of said product, and where N=L+M and M≧L.

9. The data processing apparatus of claim 8, wherein:

said second portion of M bits of said output bus are derived from at least some bits of at least one of said first N bit input bus other than said first set of L bits and said second N bit input bus other than said second set of L bits.

10. The data processing apparatus of claim 9, wherein:

said second portion of M bits of said output bus consists of bits of said first input bus other than said first set of L bits.

11. The data processing apparatus of claim 8, further comprising:

a data register file including

a plurality of data registers for storing data,

a first source output bus connected to said first input bus for recalling data stored in a first data register of said plurality of data registers,

a second source output bus connected to said second input bus for recalling data stored in a second data register of said plurality of data registers, and

a first destination input bus connected to said output bus for storing in a third data register of said plurality of data registers data on said output bus.

12. The data processing apparatus of claim 8, wherein:

said number of bits M is greater than said number of bits L.

13. The data processing apparatus of claim 8, wherein:

said number of bits L equals 16 bits.

14. The data processing apparatus of claim 8, wherein; said number of bits L equals 8 bits.

15. An data processing system comprising:

an data system bus transferring data and addresses;

a system memory connected to said data system bus, said system memory storing data and transferring data via said data system bus;

an data processor circuit connected to said data system bus, said data processor circuit including

a first input bus of N bits;

a second input bus of N bits;

a multiplier having a first input of L bits connected to a first set of L bits of said first input bus, L being less than N, a second input of L bits connected to a second set of L bits of said second input bus, and a product output of 2 L bits producing a product of data that was supplied to said first and second inputs; and

an output bus of N bits connected to said product output of said multiplier, said output bus including a first portion consisting of a most significant set of L bits of said product output of 2 L bits and a second portion of M bits, where said second portion of M bits does not include a least significant set of L bits of said product, and

where N=L+M and M≧L.

16. The data processing system of claim 15, wherein:

said data processor circuit wherein said second portion of M bits of said output bus are derived from at least some bits of at least one of said first N bit input bus other than said first set of L bits and said second N bit input bus other than said second set of L bits.

17. The data processing system of claim 16, wherein:

said data processor circuit wherein said second portion of M bits of said output bus consists of bits of said first N bit input bus other than said first set of L bits.

18. The data processing system of claim 15, wherein:

said data processor circuit further includes

a data register file including

a plurality of data registers for storing data,

a first source output bus connected to said first input bus for recalling data stored in a first data register of said plurality of data registers,

a second source output bus connected to said second input bus for recalling data stored in a second data register of said plurality of data registers, and

a first destination input bus connected to said output bus for storing in a third data register of said plurality of data registers data on said output bus.

19. The data processing system of claim 15, wherein:

said data processor circuit wherein said number of bits M is greater than said number of bits L.

20. The data processing system of claim 15, wherein:

said data processor circuit wherein said number of bits L equals 16 bits.

21. The data processing system of claim 15, wherein;

said data processor circuit wherein said number of bits L equals 8 bits.

22. The data processor system of claim 15, wherein:

said system memory consists of an image memory storing image data in a plurality of pixels; and

said data processor system further comprising:

a printer connected to said image memory generating a printed output of an image consisting of a plurality of pixels stored in said image memory.

23. The data processor system of claim 22, wherein:

said printer consists of a color printer.

24. The data processor system of claim 22, further comprising:

a printer controller forming a connection between said image memory and said printer, said printer controller transforming pixels recalled from said image memory into print signals driving said printer; and

wherein said data processor circuit further includes

a frame controller connected to said print controller controlling said print controller transformation of pixels into print signals.

25. The data processor system of claim 15, wherein:

said system memory consists of an image memory storing image data in a plurality of pixels; and

said data processor system further comprising:

an imaging device connected to said image memory generating an image signal input.

26. The data processor system of claim 25, further comprising:

an image capture controller forming a connection between said imaging device and said image memory, said image capture controller transforming said image signal into pixels supplied for storage in said image memory; and

wherein said data processor circuit further includes

a frame controller connected to said image capture controller controlling said image capture controller transformation of said image signal into pixels.

27. The data processor system of claim 15, further comprising:

a modem connected to said data system bus and to a communications line.

28. The data processor system of claim 15, further comprising:

a host processing system connected to said data system bus.

29. The data processor system of claim 28, further comprising:

a host system bus connected to said host processing system transferring data and addresses; and

at least one host peripheral connected to said host system bus.

* * * * *

EXHIBIT E

US006065113A

United States Patent [19]

Shiell et al.

[11] **Patent Number: 6,065,113**

[45] **Date of Patent: May 16, 2000**

[54] **CIRCUITS, SYSTEMS, AND METHODS FOR UNIQUELY IDENTIFYING A MICROPROCESSOR AT THE INSTRUCTION SET LEVEL EMPLOYING ONE-TIME PROGRAMMABLE REGISTER**

[75] Inventors: **Jonathan H. Shiell**, Plano; **Joel J. Graber**; **Donald E. Steiss**, both of Richardson, all of Tex.

[73] Assignee: **Texas Instruments Incorporated**, Dallas, Tex.

[21] Appl. No.: **08/813,887**

[22] Filed: **Mar. 7, 1997**

[51] **Int. Cl.**[7] **G06F 9/00**

[52] **U.S. Cl.** **712/227**

[58] **Field of Search** 395/568, 800.42; 712/227, 42

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,671,435	9/1997	Alpert	395/800.38
5,732,207	3/1998	Allen et al.	395/182.03
5,774,544	6/1998	Lee et al.	380/4
5,790,663	8/1998	Lee et al.	380/4
5,794,066	8/1998	Dreyer et al.	395/800.42

OTHER PUBLICATIONS

Intel, Pentium Pro Family Developer's Manual, vol. 2: Programmer's Reference Manual, Dec., 1995, pp. 11–73 to 11–79.

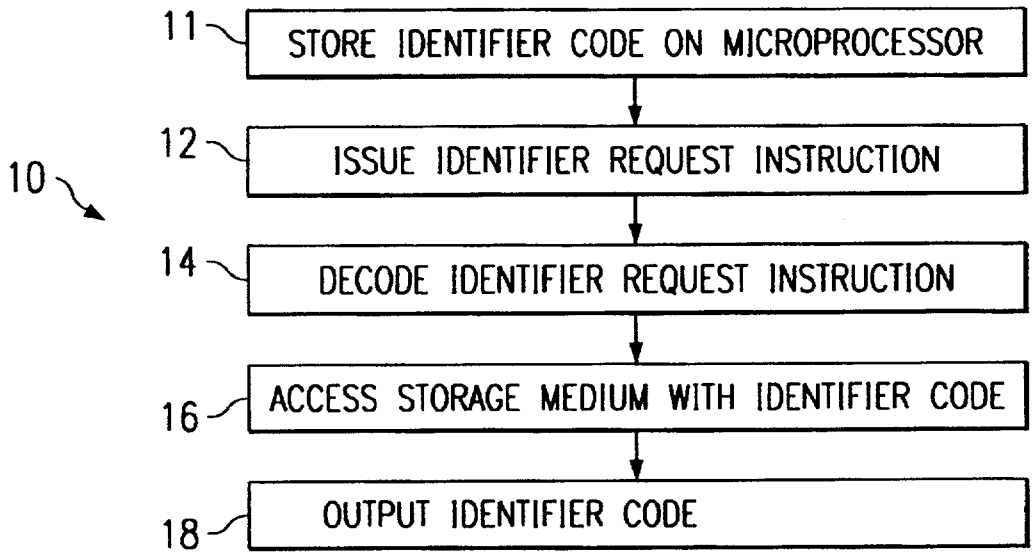
Primary Examiner—Richard L. Ellis

Attorney, Agent, or Firm—Robert D. Marshall, Jr.; Gerald E. Laws; Richard L. Donaldson

[57] **ABSTRACT**

In a method embodiment **(10)**, the method operates a microprocessor **(110)**, and the microprocessor has an instruction set. The method first **(11)** stores an identifier code uniquely identifying the particular microprocessor in a one-time programmable register. The method second **(12)** issues to the microprocessor an identifier request instruction from the instruction set. The method then, and in response to the identifier request instruction, provides **(18)** from the microprocessor an identifier code. Other circuits, systems, and methods are also disclosed and claimed.

49 Claims, 2 Drawing Sheets



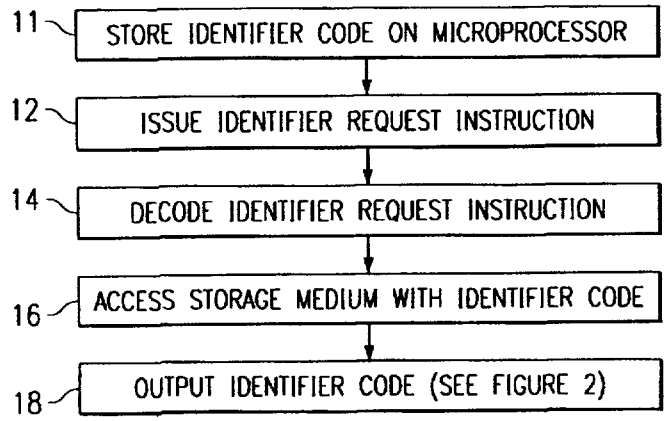


FIG. 1

20

	BITS	INFORMATION
20a	4–0	X COORDINATE OF MICROPROCESSOR ON WAFER
20b	9–5	Y COORDINATE OF MICROPROCESSOR ON WAFER
20c	15–10	WAFER NUMBER IN LOT
20d	39–16	LOT NUMBER FOR WAFER
20e	42–40	FACILITY WHERE MICROPROCESSOR IS MANUFACTURED
20f	60–43	ERROR CORRECTION CODE

FIG. 2

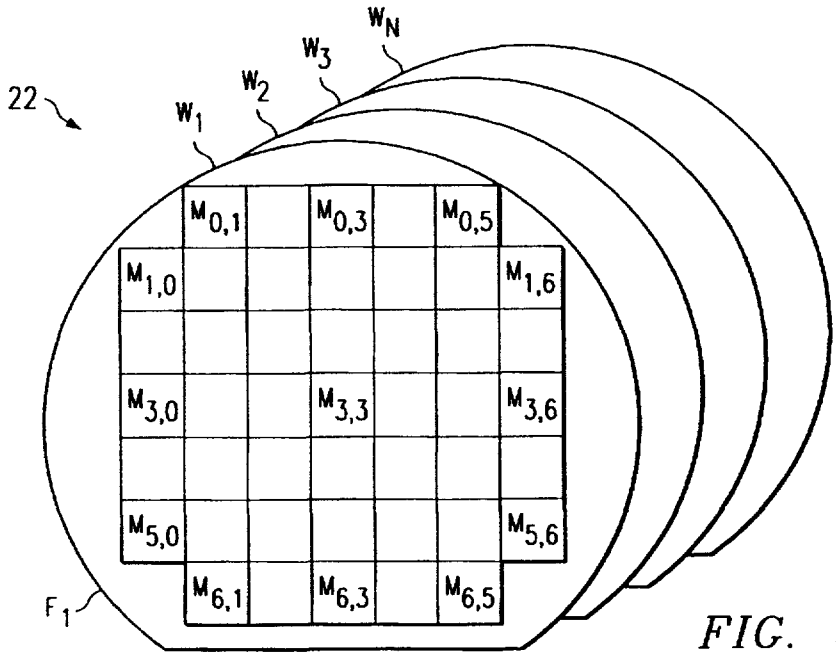


FIG. 3

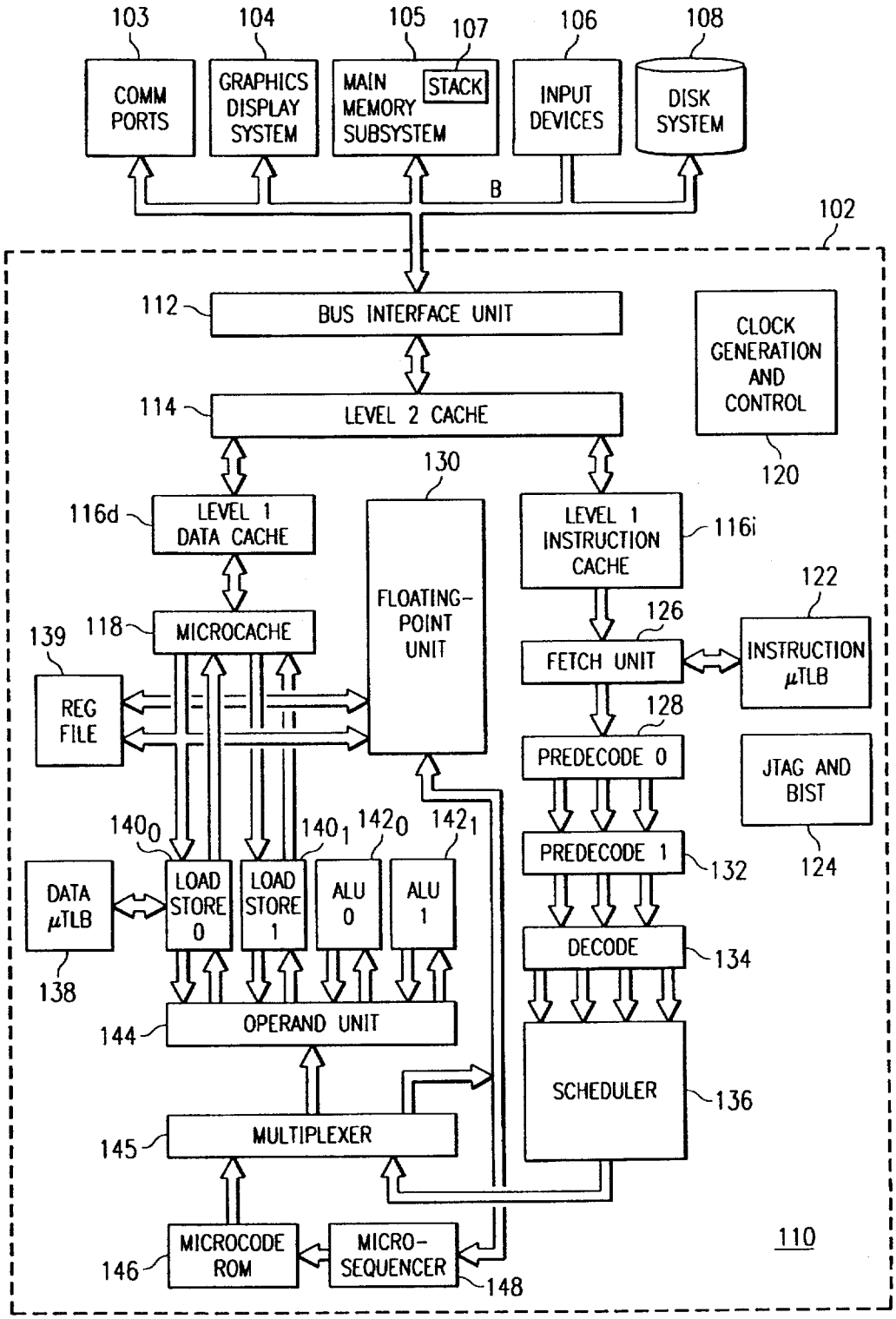

103
104
105
107
106
108
COMM PORTS
GRAPHICS DISPLAY SYSTEM
MAIN MEMORY SUBSYSTEM
STACK
INPUT DEVICES
DISK SYSTEM
B
102
112
BUS INTERFACE UNIT
CLOCK GENERATION AND CONTROL
114
LEVEL 2 CACHE
120
130
116d
LEVEL 1 DATA CACHE
FLOATING-POINT UNIT
LEVEL 1 INSTRUCTION CACHE
116i
122
126
139
118
MICROCACHE
FETCH UNIT
INSTRUCTION μTLB
REG FILE
128
PREDECODE 0
JTAG AND BIST
140₀
140₁
142₀
142₁
PREDECODE 1
132
124
DATA μTLB
LOAD STORE 0
LOAD STORE 1
ALU 0
ALU 1
DECODE
134
138
144
OPERAND UNIT
SCHEDULER
136
145
MULTIPLEXER
110
146
MICROCODE ROM
MICRO-SEQUENCER
148

FIG. 4

CIRCUITS, SYSTEMS, AND METHODS FOR UNIQUELY IDENTIFYING A MICROPROCESSOR AT THE INSTRUCTION SET LEVEL EMPLOYING ONE-TIME PROGRAMMABLE REGISTER

TECHNICAL FIELD OF THE INVENTION

The present embodiments relate to microprocessor systems, and are more particularly directed to circuits, systems, and methods for uniquely identifying a microprocessor at the instruction set level.

BACKGROUND OF THE INVENTION

The embodiments described below involve the field of microprocessor identification in computer systems. Microprocessor-based computer systems have become incredibly prolific and are used at all levels of the public and private sector. With the vast increase of microprocessors in circulation, there sometimes arises the need to identify various details about the microprocessor within a system. Moreover, this identification process is preferably achieved electronically as opposed to some physical and/or external indication such as a bar code or printed label. For example, in the Intel x86 microprocessors, there is included an instruction at the instruction set architecture ("ISA") level known as CPUID. As known in the art, the CPUID instruction provides information to software including the vendor, family, model, and stepping of the microprocessor on which it is executing. This information may then be used by the software for purposes known in the art.

Other systems include certain electronic identification techniques. For example, some systems by IBM include a storage device separate from the microprocessor, where a system level identifier is stored in that separate storage device. This system, however, suffers various drawbacks. For instance, the identifier only identifies the system and not necessarily the specific microprocessor included within that system. In addition, because the identifier is in a storage device apart from the microprocessor, the identifier may fail its purpose if either the storage device or the microprocessor is replaced without updating the identifier in the storage device to reflect this changing event. As another example of current systems, some microprocessors include an identifier which is accessible via the JTAG scan. This approach, however, also suffers various drawbacks. For example, the JTAG scan is commonly a technique requiring access to a particular microprocessor port and with particularized equipment In addition, the JTAG scan may only be performed meaningfully given knowledge about the scan chain of the scanned registers. Still further, this technique is commonly only used at the development and manufacturing stage.

In view of the above, there arises a need to address the drawbacks of current systems.

SUMMARY OF THE INVENTION

The present embodiments relate to microprocessor systems, and are more particularly directed to circuits, systems, and methods for uniquely identifying a microprocessor at the instruction set level. In a method embodiment, the method operates a microprocessor, and the microprocessor has an instruction set. The method first issues to the microprocessor an identifier request instruction from the instruction set. The method then, and in response to the identifier request instruction, provides from the microprocessor an identifier code. The identifier code uniquely identifies the microprocessor. Other circuits, systems, and methods are also disclosed and claimed.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** illustrates a flowchart of a method embodiment which may be used with a microprocessor to uniquely identify the microprocessor using an instruction from the microprocessor instruction set;

FIG. **2** illustrates the preferred information included within the unique identifier code of the microprocessor;

FIG. **3** diagrammatically illustrates a lot of semiconductor wafers, with one of those wafers further demonstrating the formation and location of microprocessors along the wafer; and

FIG. **4** illustrates an exemplary data processing system within which the preferred embodiments may be implemented.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Although either the CPUID instruction or the IBM system described in the above Background provide some information about the microprocessor which executes that instruction, the inventors of the present embodiments have discovered that such information by itself may be insufficient in certain instances. Indeed, the present inventors have further discovered that as microprocessor systems advance in development and complexity, it may be desirable to identify more information about a microprocessor than that provided by the CPUID instruction or by the IBM system. For example, the inventors of the present embodiments believe it highly beneficial to be able to uniquely and electronically identify each microprocessor, that is, in a manner where each microprocessor is uniquely distinguished from all other microprocessors. In contrast, the information output in response to the CPUID instruction does not accomplish this functionality. Thus, the following embodiments seek to provide such functionality, and include it at the ISA level.

FIG. **1** illustrates a flowchart of a method embodiment designated generally at **10** and which may be used with a microprocessor in accordance with the principles described below; note also that an example of such a microprocessor is described later in connection with FIG. **4**. Before discussing the details of method **10** as well as the exemplary microprocessor of FIG. **4**, note in general that the embodiments below operate to uniquely identify the microprocessor at the instruction set architecture ("ISA") level. Various benefits from these embodiments are set forth below, and still additional such benefits will be apparent to a person skilled in the art.

Method **10** begins with step **11**, where a code is stored on the microprocessor chip and which uniquely identifies the microprocessor. For purposes of this document, this information is referred to as an identifier code. In the preferred embodiment, and as detailed below, this identifier code constitutes several groups of bits stored in a dedicated register included on the microprocessor chip. These groups of bits, when viewed together, form a pattern which is unique to the particular microprocessor which stores those bits. FIG. **2**, discussed below, sets forth the preferred information of these bit groups. In the preferred embodiment, note that step **11** is performed at the stage where the microprocessor is manufactured. For example, and as

detailed below, the identifier code is preferably stored to a one-time programmable register on the microprocessor. Once this event occurs, steps **12** through **18** may be performed by operating the microprocessor as discussed below.

Step **12** issues an instruction to the microprocessor at the ISA level and to cause the microprocessor to uniquely identify itself as described below. For purposes of this document, such an instruction, or a like instruction which accomplishes this functionality, is referred to as an identifier request instruction. In the preferred embodiment, the identifier request instruction is included within the instruction set of the microprocessor and, therefore, the instruction may be issued to the microprocessor by any feature having access to the ISA level of the microprocessor. For example, the identifier request instruction may be issued by a computer BIOS, an operating system, or an application program. In the preferred embodiment the identifier request instruction is a dedicated instruction which, as detailed below, operates as a request to read a register within the microprocessor; however, other instructions which perform the operations set forth below may be used in lieu of the preferred embodiment. For example, the CPUID instruction described in the above Background may be extended to achieve this functionality.

Step **14** decodes the identifier request instruction according to principles known in the art, and according to the particular decoding techniques applicable to the particular microprocessor which is processing the instruction. The description of FIG. **4**, below, sets forth the decoding operation in accordance with the preferred embodiment.

Step **16** operates in response to the instruction issuance and decode step by accessing the identifier code stored on the microprocessor chip. This access step operates to allow reading of whatever resource is storing the identifier code. Thus, lastly in step **18**, method **10** outputs the microprocessor identifier code. This information may be used internally for further processing by the microprocessor, or may be output externally for use by features such as those set forth above (i.e., BIOS, operating system, application program). Given this access, therefore, one skilled in the art will appreciate various benefits from having access to a unique identification of the microprocessor. For example, if a manufacturer learns of the same or similar operation problem within a group of different but uniquely identified microprocessors, the manufacturer may be better able to ascertain the commonality giving rise to the problem with those microprocessors. For instance, the manufacturer may determine that each of the problematic microprocessors were manufactured at the same facility and, therefore, that the particular facility should be reviewed to better understand the cause of the operation problem. As another example, it is unfortunately the case that microprocessor theft has vastly increased in recent years. Because the present inventive embodiments provide a unique identification for each microprocessor, microprocessors manufactured in accordance with the present embodiments may be traced and identified in the event of such a theft. As yet another example, unique identification of a microprocessor may permit a software seller to license software on a per microprocessor basis. Still other examples of benefits are either given below, or will be ascertainable by a person skilled in the art.

FIG. **2** illustrates the preferred identifier code **20** which is stored in a dedicated microprocessor register as described below, and which is output in step **18** as described above. As shown, identifier code **20** preferably includes 61 bits of information, which are separated into six separate groups designated **20***a* through **20***f*. Each of these bit groups is discussed below. Before discussing those groups, note that the register which stores identifier code **20** may exceed 61 bits and, in this instance, other information may be stored in that register to take advantage of the additional bit storage available in a dedicated register. For purposes of the present embodiments, however, only those bits shown in FIG. **2** are discussed.

Bit groups **20***a* through **20***e* are better understood by first reviewing various known principles as reflected in FIG. **3**, and then returning to those groups. Particularly, FIG. **3** diagrammatically illustrates a lot **22** of semiconductor wafers. Note that FIG. **3** is simplified and not drawn to scale, but diagrammatically demonstrates the principles necessary to demonstrate various aspects of the present embodiments. Each wafer in FIG. **3** is designated separately by a capital "W" combined with a subscript designating that lot **22** includes N wafers. Further, each wafer W is of the type which is typical to construct microprocessors. Further, such wafers may be of various types, sizes, shapes and so forth, but commonly each is collected in a group known as a lot. Typically, a lot of wafers are assigned a lot number for tracking during the manufacturing process. Each wafer typically includes a reference point, such as a flat region which from the perspective of FIG. **3** is visible only on wafer W_1 and designated as F_1. Also as known in the art, each wafer lot such as lot **22** is processed at a fabrication facility in order to construct semiconductor chip devices. In the present embodiments, these chip devices are microprocessors which operate in accordance with the principles set forth in this document. Moreover, a plurality of microprocessors are formed on each such wafer, again according to whatever appropriate technique. For purposes of illustration, therefore, the perspective of FIG. **3** shows various boxes along wafer W_1, with each such box depicting diagrammatically a microprocessor manufactured on wafer W_1. For identification purposes, the location of each microprocessor on a given wafer is preferably assigned according to an X-Y coordinate system. As examples, therefore, note that certain microprocessors on wafer W_1 include a specific designation including a capital "M" and an "x,y" subscript. The "M" indicates that the die is a microprocessor while the "x,y" indicates the location of the microprocessor along the corresponding wafer. Note also that to simplify FIG. **3**, only a few such designations are included. Moreover, due to the circular shape of each wafer and the orientation of the rows relative to the flat F_1, certain rows along with wafer will include more columns than others. Typically, the numbering of rows and columns is relative to the row or rows containing the largest number of microprocessors along that row. For example, the second row shown in FIG. **3** includes seven microprocessors and the far left microprocessor is designated in column **0** of that row; however, for the first row immediately above it and due to the circular shape of that row, there is no microprocessor at the zero location of the row and, instead, the first numbered microprocessor in the first row is in column **1** rather than column **0**. Of course, this numbering scheme is only by way of example, and others could be implemented by a person skilled in the art so that each microprocessor has a unique location coding along its corresponding semiconductor wafer.

Returning now to FIG. **2**, and given the illustration of FIG. **3**, groups **20***a* through **20***e* are better understood. Note that each of these groups in the preferred embodiment reflect information regarding the manufacturing location of the microprocessor. First, groups **20***a* and **20***b* specify the x and y coordinate, respectively, of the microprocessor on its

corresponding wafer. For example, if the microprocessor located on the far left of the top row of FIG. 3 returned its identifier code **20**, groups **20***a* and **20***b* would indicate, in whatever preferred manner, the x and y coordinates of **0** and **1**, respectively. Second, group **20***c* identifies the number of the wafer within lot **22** on which the microprocessor was manufactured. Thus, in the instance of FIG. 3, group **20***c* would designate one of the wafers of the wafers indicated at W_1 through W_N. Third, group **20***d* identifies the lot number of lot **22** assigned to the particular wafer which included the microprocessor at issue. Fourth, group **20***e* identifies the facility where the particular microprocessor was manufactured.

As demonstrated above, the preferred information within identifier code **20** designates, for a given microprocessor, a single location along a single wafer within a single wafer lot and at a single fabrication facility. Note that this information may be particularly beneficial for purposes of identifying operability problems with certain microprocessors. For example, if a group of microprocessors is identified having a common problem, statistical analyses may be performed on the above information to determine whether there is commonality of one of the above factors for those microprocessors have the same or similar operability problems. For instance, it may be determined that all, or most, of the problematic microprocessors came from a single lot of wafers. In another instance, it may be determined that all, or most, of the problematic microprocessors were located at a common x-y coordinate along various different wafers. Still other instances will be ascertainable by a person skilled in the art and, indeed, could be further enhanced if the particular type of information included within the identifier code were expanded by such a person. Thus, these additional advantages further demonstrate the inventive scope of the present embodiments.

From the above, note also that groups that groups **20***a* through **20***e* may be combined to form a code which uniquely identifies the microprocessor. In other words, assuming correct encoding of those bits, those groups in combination should provide a series of bits which is unique for each microprocessor because only one microprocessor will be at a single location along a single wafer within a single wafer lot and at a single fabrication facility. In addition, each of those groups, as well as group **20***f* provides additional functionality as set forth above for each respective group. Note, however, that one skilled in the art could select less than all of those groups, or even alternative groups, and still achieve aspects of the inventive scope set forth herein. For example, the preferred embodiment described above involves manufacture of wafers in lots, which typically occurs using various mask sets for that lot. As an alternative, however, individual wafers may be constructed alone and not in lots, such as by using e-beam lithography. In this instance, alternative information could be written to each microprocessor to identify the particular instance of the lithographic formation of the microprocessor so that that instance, as well as the microprocessor itself, may be distinguished from other microprocessors and other instances of forming a microprocessor using the same or similar lithographic processes.

Completing the groups of FIG. 2, group **20***f* is an eighteen bit error correction code. Particularly, and as mentioned previously, in the preferred embodiment each bit of the identifier code is stored in a one-time programmable storage medium on the microprocessor. This medium is preferably a register which, during manufacture of the microprocessor, includes a number of fuses where the total number of fuses exceeds the number of bits necessary to uniquely identify the microprocessor. In order to encode the identifier code on the microprocessor, selected ones of the fuses are broken during the manufacturing stage. In the preferred embodiment, this selective breaking of fuses is achieved using a laser, as in the same manner as is known in the memory fabrication art. Indeed, note in the preferred embodiment described below in connection with FIG. 4 that the microprocessor preferably includes various cache structures. Some or all of these cache structures are preferably also constructed using the same fuse technique and, therefore, the additional cost of using that technique to encode the identifier code is substantially less than that compared to a microprocessor which is otherwise being constructed without that technique. Note further that in certain instances it may occur that one or more of the fuses, either before or after the initial fuse breaking operation, is erroneously configured; therefore, the erroneous fuses will not properly convey the intended information for the identifier code. As a result, the error correction code represented by group **20***f* permits selective changing of this bit group to indicate the erroneous configuration of the fuses and to provide the corrected information. Note that this feature is particularly beneficial because, without this aspect, the microprocessor would include only the minimal number of fuses to encode the identifier code; however, in such event, if it were learned that one of these limited number of fuses were erroneously configured, either the microprocessor would have to be discarded or it would not operate to output a correct identifier code.

In addition to the above, note that the one-time programmable storage medium on the microprocessor which stores the identifier code may be constructed by techniques other than fuses, and even if fuses are used, they may be created or broken by techniques other than laser breaking. As one example, fuses could be created but selectively broken with current. As another example, anti-fuses could be used to selectively form fuses to encode the preferred information. As another example, an EPROM could be used. As yet another example, if the e-beam lithography technology described above, or some like technology, were used to manufacture the microprocessor, then the beam could be written to directly encode the identifier onto the microprocessor. Still other techniques will be ascertainable to a person skilled in the art.

In another aspect of the present invention, each microprocessor having a unique identifier code stored in a register such as that set forth above will further include a dedicated register in addition to the fuse register, where that additional register however is of a more common read/write type register such as a static latch. Further, in this embodiment, the contents of the fuse register are copied at some point to the read/write type register, such as during reset of the microprocessor. Thereafter, the unique identifier code may be read from the read/write type register rather than the fuse register. Note that this additional aspect is beneficial for various reasons. For example, the fuse register is likely to consume a relatively larger amount of current; therefore, the copied information may instead be maintained and read (after the initial copy) in connection with the read/write type register, thereby reducing overall power consumption of the microprocessor and, more significantly, reducing standby power consumption as well.

Having described the above embodiments, FIG. 4 illustrates a block diagram of a microprocessor embodiment into which the above embodiments may be incorporated. Referring now to FIG. 4, an exemplary data processing system

102, including an exemplary superscalar pipelined microprocessor 110 within which the preferred embodiment is implemented, will be described. It is to be understood that the architecture of system 102 and of microprocessor 110 is described herein by way of example only, as it is contemplated that the present embodiments may be utilized in microprocessors of various architectures. It is therefore contemplated that one of ordinary skill in the art, having reference to this specification, will be readily able to implement the present embodiments in such other microprocessor architectures.

Microprocessor 110, as shown in FIG. 4, is connected to other system devices by way of bus B. While bus B, in this example, is shown as a single bus, it is of course contemplated that bus B may represent multiple buses having different speeds and protocols, as is known in conventional computers utilizing the PCI local bus architecture; single bus B is illustrated here merely by way of example and for its simplicity. System 102 contains such conventional subsystems as communication ports 103 (including modem ports and modems, network interfaces, and the like), graphics display system 104 (including video memory, video processors, a graphics monitor), main memory system 105 which is typically implemented by way of dynamic random access memory (DRAM) and includes a stack 107, input devices 106 (including keyboard, a pointing device, and the interface circuitry therefor), and disk system 108 (which may include hard disk drives, floppy disk drives, and CD-ROM drives). It is therefore contemplated that system 102 of FIG. 4 corresponds to a conventional desktop computer or workstation, as are now common in the art. Of course, other system implementations of microprocessor 110 can also benefit from the present embodiments, as will be recognized by those of ordinary skill in the art.

Microprocessor 110 includes a bus interface unit ("BIU") 112 that is connected to bus B, and which controls and effects communication between microprocessor 110 and the other elements in system 102. BIU 112 includes the appropriate control and clock circuitry to perform this function, including write buffers for increasing the speed of operation, and including timing circuitry so as to synchronize the results of internal microprocessor operation with bus B timing constraints. Microprocessor 110 also includes clock generation and control circuitry 120 which, in this exemplary microprocessor 110, generates internal clock phases based upon the bus clock from bus B; the frequency of the internal clock phases, in this example, may be selectably programmed as a multiple of the frequency of the bus clock.

As is evident in FIG. 4, microprocessor 110 has three levels of internal cache memory, with the highest of these as level 2 cache 114, which is connected to BIU 112. In this example, level 2 cache 114 is a unified cache, and is configured to receive all cacheable data and cacheable instructions from bus B via BIU 112, such that much of the bus traffic presented by microprocessor 110 is accomplished via level 2 cache 114, Of course, microprocessor 110 may also effect bus traffic around cache 114, by treating certain bus reads and writes as "not cacheable". Level 2 cache 114, as shown in FIG. 4, is connected to two level 1 caches 116; level 1 data cache 116_d is dedicated to data, while level 1 instruction cache 116_i is dedicated to instructions. Power consumption by microprocessor 110 is minimized by only accessing level 2 cache 114 only in the event of cache misses of the appropriate one of the level 1 caches 116. Furthermore, on the data side, microcache 118 is provided as a level 0 cache, and in this example is a fully dual-ported cache.

As shown in FIG. 4 and as noted hereinabove, microprocessor 110 is of the superscalar type. In this example multiple execution units are provided within microprocessor 110, allowing up to four instructions to be simultaneously executed in parallel for a single instruction pointer entry. These execution units include two ALUs 144_0, 144_2 for processing conditional branch, integer, and logical operations, floating-point unit (FPU) 130, two load-store units 140_0, 140_1, and microsequencer 148. The two load-store units 140 utilize the two ports to microcache 118, for true parallel access thereto, and also perform load and store operations to registers in register file 139. Data microtranslation lookaside buffer (μTLB) 138 is provided to translate logical data addresses into physical addresses, in the conventional manner.

These multiple execution units are controlled by way of multiple pipelines with seven stages each, with write back. The pipeline stages are as follows:

- F Fetch: This stage generates the instruction address and reads the instruction from the instruction cache or memory
- PD0 Predecode stage 0: This stage determines the length and starting position of up to three fetched x86-type instructions
- PD1 Predecode stage 1: This stage extracts the x86 instruction bytes and recodes them into fixed length format for decode
- DC Decode: This stage translates the x86 instructions into atomic operations (AOps)
- SC Schedule: This stage assigns up to four AOps to the appropriate execution units
- OP Operand: This stage retrieves the register operands indicated by the AOps
- EX Execute: This stage runs the execution units according to the AOps and the retrieved operands
- WB Write back: This stage stores the results of the execution in registers or in memory

Referring back to FIG. 4, the pipeline stages noted above are performed by various functional blocks within microprocessor 110. Fetch unit 126 generates instruction addresses from the instruction pointer, by way of instruction microtranslation lookaside buffer (μTLB) 122, which translates the logical instruction address to a physical address in the conventional way, for application to level 1 instruction cache 116_i. Instruction cache 116_i produces a stream of instruction data to fetch unit 126, which in turn provides the instruction code to the predecode stages in the desired sequence. Speculative execution is primarily controlled by fetch unit 126, in a manner to be described in further detail hereinbelow.

Predecoding of the instructions is broken into two parts in microprocessor 110, namely predecode 0 stage 128 and predecode 1 stage 132. These two stages operate as separate pipeline stages, and together operate to locate up to three x86 instructions and apply the same to decoder 134. As such, the predecode stage of the pipeline in microprocessor 110 is three instructions wide. Predecode 0 unit 128, as noted above, determines the size and position of as many as three x86 instructions (which, of course, are variable length), and as such consists of three instruction recognizers; predecode 1 unit 132 recodes the multi-byte instructions into a fixed-length format, to facilitate decoding.

Decode unit 134, in this example, contains four instruction decoders, each capable of receiving a fixed length x86 instruction from predecode 1 unit 132 and producing from one to three atomic operations (AOps); AOps are substan-

tially equivalent to RISC instructions. Three of the four decoders operate in parallel, placing up to nine AOps into the decode queue at the output of decode unit **134** to await scheduling; the fourth decoder is reserved for special cases. Scheduler **136** reads up to four AOps from the decode queue at the output of decode unit **134**, and assigns these AOps to the appropriate execution units. In addition, the operand unit **144** receives and prepares the operands for execution. As indicated in FIG. **4**, operand unit **144** receives an input from scheduler **136** and also from microcode ROM **146**, via multiplexer **145**, and fetches register operands for use in the execution of the instructions. In addition, according to this example, operand unit performs operand forwarding to send results to registers that are ready to be stored, and also performs address generation for AOps of the load and store type.

Microsequencer **148**, in combination with microcode ROM **146**, control ALUs **142** and load/store units **140** in the execution of microcode entry AOps, which are generally the last AOps to execute in a cycle. In this example, microsequencer **148** sequences through microinstructions stored in microcode ROM **146** to effect this control for those microcoded microinstructions. Examples of microcoded microinstructions include, for microprocessor **110**, complex or rarely-used x86 instructions, x86 instructions that modify segment or control registers, handling of exceptions and interrupts, and multi-cycle instructions (such as REP instructions, and instructions that PUSH and POP all registers).

Microprocessor **110** also includes circuitry **124** for controlling the operation of JTAG scan testing, and of certain built-in self-test functions, ensuring the validity of the operation of microprocessor **110** upon completion of manufacturing, and upon resets and other events.

Given the description of FIG. **4**, as well as the descriptions above such as those relating to the prior Figures, one skilled in the art may appreciate that method **10** of FIG. **1**, and the additional embodiments to accomplish the descriptions accompanying it as described in connection with FIGS. **2** and **3**, may be incorporated in connection with various components shown in FIG. **4**. For example, microprocessor **110** could be used as any microprocessor shown within FIG. **3**. Various related functionality may be further performed by the appropriate circuitry within FIG. **4**.

From the above, it may be appreciated that the above embodiments provide circuits, systems, and methods for uniquely identifying a microprocessor at the instruction set level. Various benefits have been set forth above and others will be appreciated by a person skilled in the art. Still further, the while the present embodiments have been described in detail, various substitutions, modifications or alterations could be made to the descriptions set forth above without departing from the inventive scope. In addition to the many examples set forth above, in another example the various information encoded in the identifier code could be changed. As another example, the identifier code may be stored in a medium other than a one-time programmable register. Still further, as stated above, the identifier code could be encoded in a register using a configuration other than laser fuses. As yet another example, the identifier request instruction could include functionality in addition to that set forth above. The examples as well as others ascertainable by a person skilled in the art further demonstrate the flexibility and span of the inventive scope, as further demonstrated by the following claims.

What is claimed is:

1. A method of operating a microprocessor, wherein the microprocessor comprises an instruction set, the method comprising the steps of:

first, following manufacture of the microprocessor storing a identifier code uniquely identifying the particular microprocessor in a one-time programmable register in the microprocessor;

second, issuing to the microprocessor an identifier request instruction from the instruction set; and

third, in response to the identifier request instruction, reading from the one-time programmable register of the microprocessor the identifier code.

2. The method of claim **1** wherein the microprocessor comprises one of a plurality of microprocessors manufactured on a wafer, and wherein said step of storing the identifier code comprises storing a location of the microprocessor on the wafer upon which the microprocessor is manufactured.

3. The method of claim **2** wherein said step of storing the location of the microprocessor on the wafer comprises storing an X-Y coordinate location of the microprocessor on the wafer upon which the microprocessor is manufactured.

4. The method of claim **1** wherein the microprocessor is manufactured on a wafer, and wherein said step of storing the identifier code comprises storing an identifier of the wafer upon which the microprocessor is manufactured.

5. The method of claim **1** wherein the microprocessor is manufactured on a wafer, wherein the wafer comprises one of plurality of wafers comprising a lot of wafers, and wherein said step of storing identifier code comprises storing an identifier of the lot of wafers within which the microprocessor is manufactured.

6. The method of claim **1** wherein the microprocessor is manufactured at a facility, and wherein said step of storing the identifier code comprises storing an identifier of the facility within which the microprocessor is manufactured.

7. The method of claim **1** wherein the microprocessor comprises one of a plurality of microprocessors manufactured on a wafer, and wherein said step of storing the identifier code comprises:

storing a location of the microprocessor on the wafer upon which the microprocessor is manufactured; and

storing an identifier of the wafer upon which the microprocessor is manufactured.

8. The method of claim **7** wherein the wafer comprises one of plurality of wafers comprising a lot of wafers, and wherein said step of storing the identifier code comprises storing an identifier of the lot of wafers within which the microprocessor is manufactured.

9. The method of claim **7** wherein the microprocessor is manufactured at a facility, and wherein said step of storing the identifier code comprises storing an identifier of the facility within which the microprocessor is manufactured.

10. The method of claim **1** wherein the microprocessor comprises one of a plurality of microprocessors manufactured on a wafer, wherein the wafer comprises one of plurality of wafers comprising a lot of wafers, wherein the microprocessor is manufactured at a facility and wherein said step of storing the identifier code comprises:

storing a location of the microprocessor on the wafer upon which the microprocessor is manufactured;

storing an identifier of the wafer upon which the microprocessor is manufactured;

storing an identifier of the lot of wafers within which the microprocessor is manufactured; and

storing an identifier of the facility within which the microprocessor is manufactured.

11. The method of claim **1** wherein the one-time programmable register comprises a plurality of fuses on the micro-

processor and wherein said storing step comprises breaking selected ones of the plurality of fuses to form the identifier code.

12. The method of claim **11** wherein said step of breaking selected ones of the plurality of fuses to form the identifier code comprises selectively exposing the selected ones of the plurality of fuses to a laser beam such that the selectively exposed fuses are selectively broken.

13. The method of claim **11** wherein the identifier code comprises a number of bits, wherein the plurality of fuses comprises a number of fuses, and wherein the number of fuses exceeds the number of bits.

14. The method of claim **13** wherein said step of storing the identifier code comprises storing an error correction code.

15. The method of claim **1** and further comprising the step, after said storing step and prior to said second step, the step of copying the identifier code from the one-time programmable register on the microprocessor to a read/write register, and wherein said step of reading the identifier code comprises reading from the read/write type register the identifier code.

16. The method of claim **15**, wherein the step of copying the identifier code from the one-time programmable register on the microprocessor to a read/write register includes selectively powering the one-time programmable register until said step of copying is complete and thereafter not powering the one-time programmable register.

17. A method of operating a microprocessor, wherein the microprocessor comprises an instruction set, the method comprising the steps of:

first, following manufacture of the microprocessor storing a identifier code uniquely identifying the particular microprocessor in a one-time programmable register in the microprocessor;

second, issuing to the microprocessor an identifier request instruction from the instruction set, wherein said step of issuing comprises issuing to the microprocessor a request to read the one-time programmable register in the microprocessor; and

third, in response to the identifier request instruction, reading the identifier code from the one-time programmable register.

18. The method of claim **17** wherein the microprocessor comprises one of a plurality of microprocessors manufactured on a wafer, wherein the wafer comprises one of plurality of wafers comprising a lot of wafers, wherein the microprocessor is manufactured at a facility and wherein said step of storing the identifier code comprises:

storing a location of the microprocessor on the wafer upon which the microprocessor is manufactured;

storing an identifier of the wafer upon which the microprocessor is manufactured;

storing an identifier of the lot of wafers within which the microprocessor is manufactured; and

storing an identifier of the facility within which the microprocessor is manufactured.

19. The method of claim **17** wherein the one-time programmable register comprises a plurality of fuses on the microprocessor and, said first step further comprising the step of breaking selected ones of the plurality of fuses to form the identifier code, wherein said step of breaking selected ones of the plurality of fuses to form the identifier code comprises selectively exposing the selected ones of the plurality of fuses to a laser beam such that the selectively exposed fuses are selectively broken.

20. A microprocessor for operating in response to an instruction set, said microprocessor comprising:

a one-time programmable register having stored therein following manufacture of the microprocessor an identifier code uniquely identifying said particular microprocessor;

circuitry for first issuing an identifier request instruction from the instruction set; and

circuitry for second, and in response to the identifier request instruction, reading said identifier code from said one-time programmable register of said microprocessor.

21. The microprocessor of claim **20** wherein the microprocessor comprises one of a plurality of microprocessors manufactured on a wafer, and wherein said identifier code stored in said ont-time programmable register comprises a location of the microprocessor on the wafer upon which the microprocessor is manufactured.

22. The microprocessor of claim **21** wherein said location of the microprocessor on the wafer comprises an X-Y coordinate location of the microprocessor on the wafer upon which the microprocessor is manufactured.

23. The microprocessor of claim **20** wherein the microprocessor is manufactured on a wafer, and wherein said identifier code comprises an identifier of the wafer upon which the microprocessor is manufactured.

24. The microprocessor of claim **20** wherein the microprocessor is manufactured on a wafer, wherein the wafer comprises one of plurality of wafers comprising a lot of wafers, and wherein said identifier code comprises an identifier of the lot of wafers within which the microprocessor is manufactured.

25. The microprocessor of claim **20** wherein the microprocessor is manufactured at a facility, and wherein said identifier code comprises an identifier of the facility within which the microprocessor is manufactured.

26. The microprocessor of claim **20** wherein the microprocessor comprises one of a plurality of microprocessors manufactured on a wafer, and wherein said identifier code comprises:

a location of the microprocessor on the wafer upon which the microprocessor is manufactured; and

an identifier of the wafer upon which the microprocessor is manufactured.

27. The microprocessor of claim **26** wherein the wafer comprises one of plurality of wafers comprising a lot of wafers, and wherein said identifier code comprises an identifier of the lot of wafers within which the microprocessor is manufactured within which the microprocessor is manufactured.

28. The microprocessor of claim **26** wherein the microprocessor is manufactured at a facility, and wherein said identifier code comprises an identifier of the facility within which the microprocessor is manufactured.

29. The microprocessor of claim **20** wherein the microprocessor comprises one of a plurality of microprocessors manufactured on a wafer, wherein the wafer comprises one of plurality of wafers comprising a lot of wafers, wherein the microprocessor is manufactured at a facility and wherein said an identifier code comprises:

a location of the microprocessor on the wafer upon which the microprocessor is manufactured;

an identifier of the wafer upon which the microprocessor is manufactured;

an identifier of the lot of wafers; and

an identifier of the facility within which the microprocessor is manufactured.

30. The microprocessor of claim 26 wherein the one-time programmable storage medium comprises a plurality of fuses.

31. The microprocessor of claim 30 wherein the plurality of fuses comprise a plurality of laser-breakable fuses.

32. The microprocessor of claim 30 wherein the identifier code comprises a number of bits, wherein the plurality of fuses comprises a number of fuses, and wherein the number of fuses exceeds the number of bits.

33. The microprocessor of claim 32 wherein said circuitry for reading said identifier code comprises circuitry for reading an error correction code.

34. The microprocessor of claim 20 wherein said circuitry for reading said identifier code comprises:

a read/write register; and

circuitry for copying the identifier code from the one-time programmable storage medium on the microprocessor to said read/write register upon reset of the microprocessor.

35. The microprocessor of claim 34, wherein:

said circuitry for copying the identifier code from the one-time programmable storage medium on the microprocessor to said read/write register upon reset of the microprocessor selectively powers the one-time programmable register until the copying is complete and thereafter removes power from the one-time programmable register.

36. A microprocessor for operating in response to an instruction set, said microprocessor comprising:

a one-time programmable register having stored therein following manufacture of the microprocessor an identifier code uniquely identifying said particular microprocessor;

circuitry for first issuing an identifier request instruction from the instruction set; and

circuitry for second, and in response to the identifier request instruction, reading said identifier code from said one-time programmable register.

37. The microprocessor of claim 36 wherein the microprocessor comprises one of a plurality of microprocessors manufactured on a wafer, wherein the wafer comprises one of plurality of wafers comprising a lot of wafers, wherein the microprocessor is manufactured at a facility and wherein said identifier code comprises:

a location of the microprocessor on the wafer upon which the microprocessor is manufactured;

an identifier of the wafer upon which the microprocessor is manufactured;

an identifier of the lot of wafers within which the microprocessor is manufactured; and

an identifier of the facility within which the microprocessor is manufactured.

38. A microprocessor-based computer system, comprising:

an input device;

a display system;

a main memory; and

a microprocessor for operating in response to an instruction set, said microprocessor comprising:

a one-time programmable register having stored therein following manufacture of said microprocessor an identifier code uniquely identifying said particular microprocessor;

circuitry for first issuing an identifier request instruction from the instruction set; and

circuitry for second, and in response to the identifier request instruction, reading said identifier code from the one-time programmable register of said microprocessor.

39. The system of claim 38 wherein the microprocessor comprises one of a plurality of microprocessors manufactured on a wafer, and wherein said identifier code comprises a location of the microprocessor on the wafer upon which the microprocessor is manufactured.

40. The system of claim 39 wherein said location of the microprocessor on the wafer comprises an X-Y coordinate location of the microprocessor on the wafer upon which the microprocessor is manufactured.

41. The system of claim 38 wherein the microprocessor is manufactured on a wafer, and wherein said identifier code comprises identifier of the wafer upon which the microprocessor is manufactured.

42. The system of claim 38 wherein the microprocessor is manufactured on a wafer, wherein the wafer comprises one of plurality of wafers comprising a lot of wafers, and wherein said identifier code comprises an identifier of the lot of wafers within which the microprocessor is manufactured.

43. The system of claim 38 wherein the microprocessor is manufactured at a facility, and wherein said identifier code comprises identifier of the facility within which the microprocessor is manufactured.

44. The system of claim 38 wherein the one-time programmable storage medium comprises a plurality of fuses.

45. The system of claim 44 wherein the plurality of fuses comprise a plurality of laser-breakable fuses.

46. The system of claim 44 wherein the identifier code comprises a number of bits, wherein the plurality of fuses comprises a number of fuses, and wherein the number of fuses exceeds the number of bits.

47. The system of claim 44 wherein said identifier code comprises an error correction code.

48. The system of claim 38 further comprising:

a read/write register; and

circuitry for copying said identifier code from said one-time programmable register on the microprocessor to said read/write register upon reset of the microprocessor.

49. The microprocessor of claim 48, wherein:

said circuitry for copying the identifier code from the one-time programmable storage medium on the microprocessor to said read/write register upon reset of the microprocessor selectively powers the one-time programmable register until the copying is complete and thereafter removes power from the one-time programmable register.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,065,113

DATED : 5/16/00

INVENTOR(S) : Jonathan H. Shiell, Joel J. Graber, Donald E. Steiss

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Cover Page, insert Item [60] under Related U.S. Application Data

--Provisional Application No. 60/013,053 March 8, 1996.--

Signed and Sealed this

Fifteenth Day of May, 2001

Attest:

Nicholas P. Godici

NICHOLAS P. GODICI

Attesting Officer

Acting Director of the United States Patent and Trademark Office