US 20050010743A1

(19) **United States**

(12) **Patent Application Publication**

(10) **Pub. No.: US 2005/0010743 A1**

(43) **Pub. Date: Jan. 13, 2005**

Tremblay et al.

(54) **MULTIPLE-THREAD PROCESSOR FOR THREADED SOFTWARE APPLICATIONS**

(75) Inventors: **Marc Tremblay**, Menlo Park, CA (US); **William Joy**, Aspen, CO (US)

Correspondence Address:
ZAGORIN O'BRIEN & GRAHAM, L.L.P.
7600B N. CAPITAL OF TEXAS HWY.
SUITE 350
AUSTIN, TX 78731 (US)

(73) Assignee: **Sun Microsystems, Inc.**

(21) Appl. No.: **10/818,785**

(22) Filed: **Apr. 6, 2004**

Related U.S. Application Data

(63) Continuation of application No. 09/204,480, filed on Dec. 3, 1998, now Pat. No. 6,718,457.

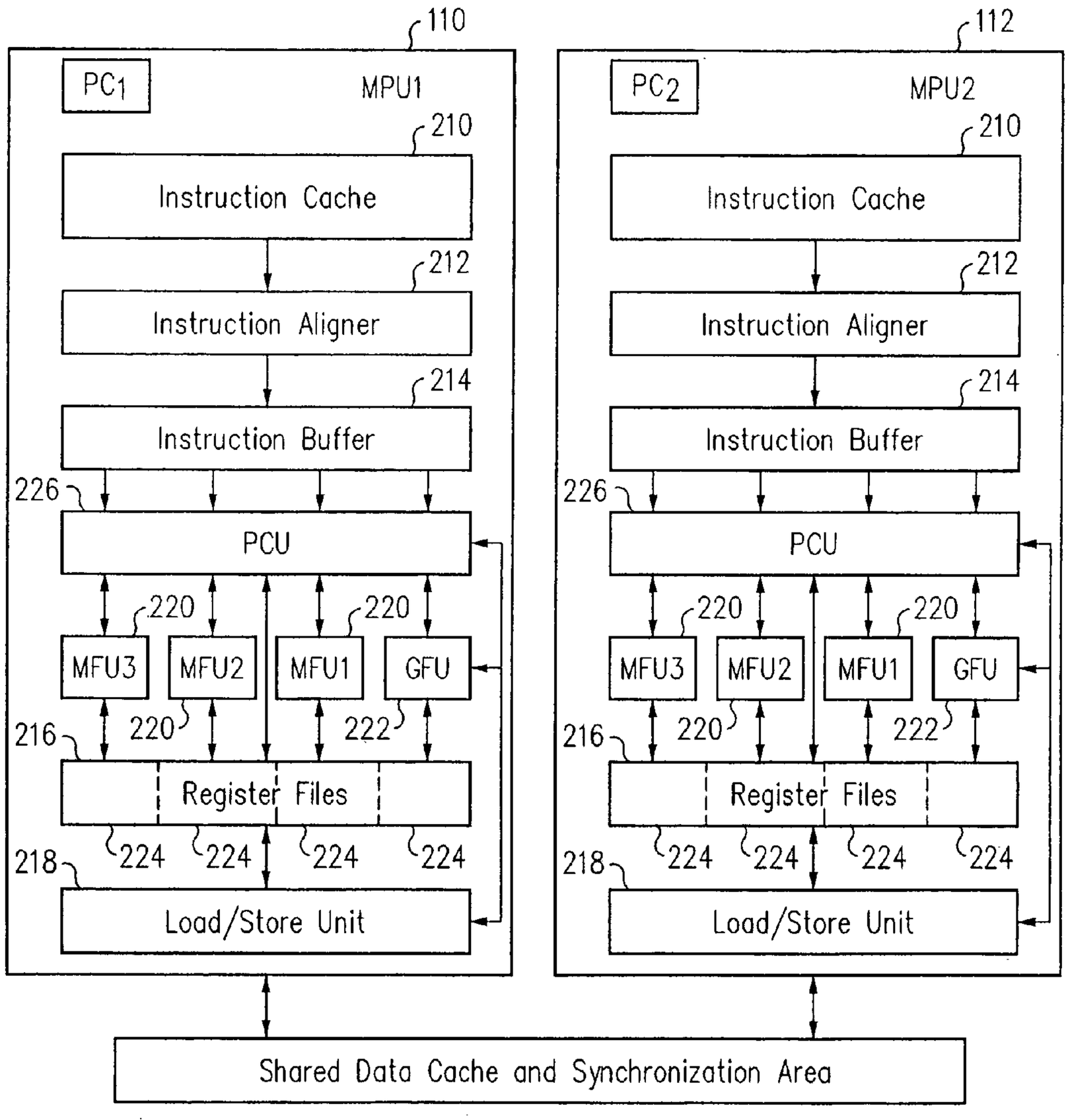
Publication Classification

(51) **Int. Cl.**[7] **G06F 15/00**

(52) **U.S. Cl.** **712/10**

(57) **ABSTRACT**

A processor has an improved architecture for multiple-thread operation on the basis of a highly parallel structure including multiple independent parallel execution paths for executing in parallel across threads and a multiple-instruction parallel pathway within a thread. The multiple independent parallel execution paths include functional units that execute an instruction set including special data-handling instructions that are advantageous in a multiple-thread environment.



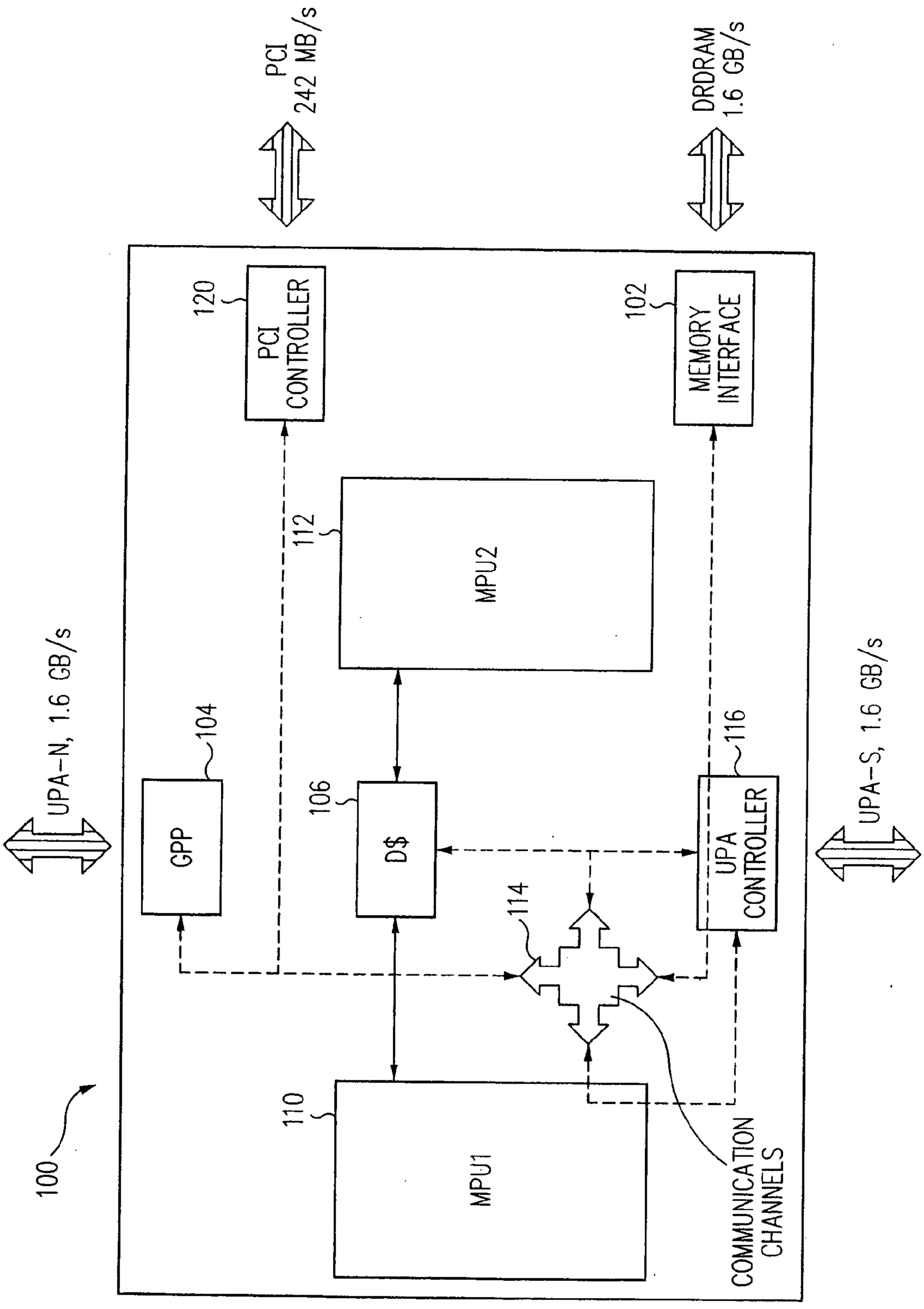

PCI
242 MB/s
DRDRAM
1.6 GB/s
120
PCI CONTROLLER
102
MEMORY INTERFACE
112
MPU2
UPA–N, 1.6 GB/s
104
GPP
106
D$
114
116
UPA CONTROLLER
UPA–S, 1.6 GB/s
110
MPU1
COMMUNICATION CHANNELS
100

FIG. 1

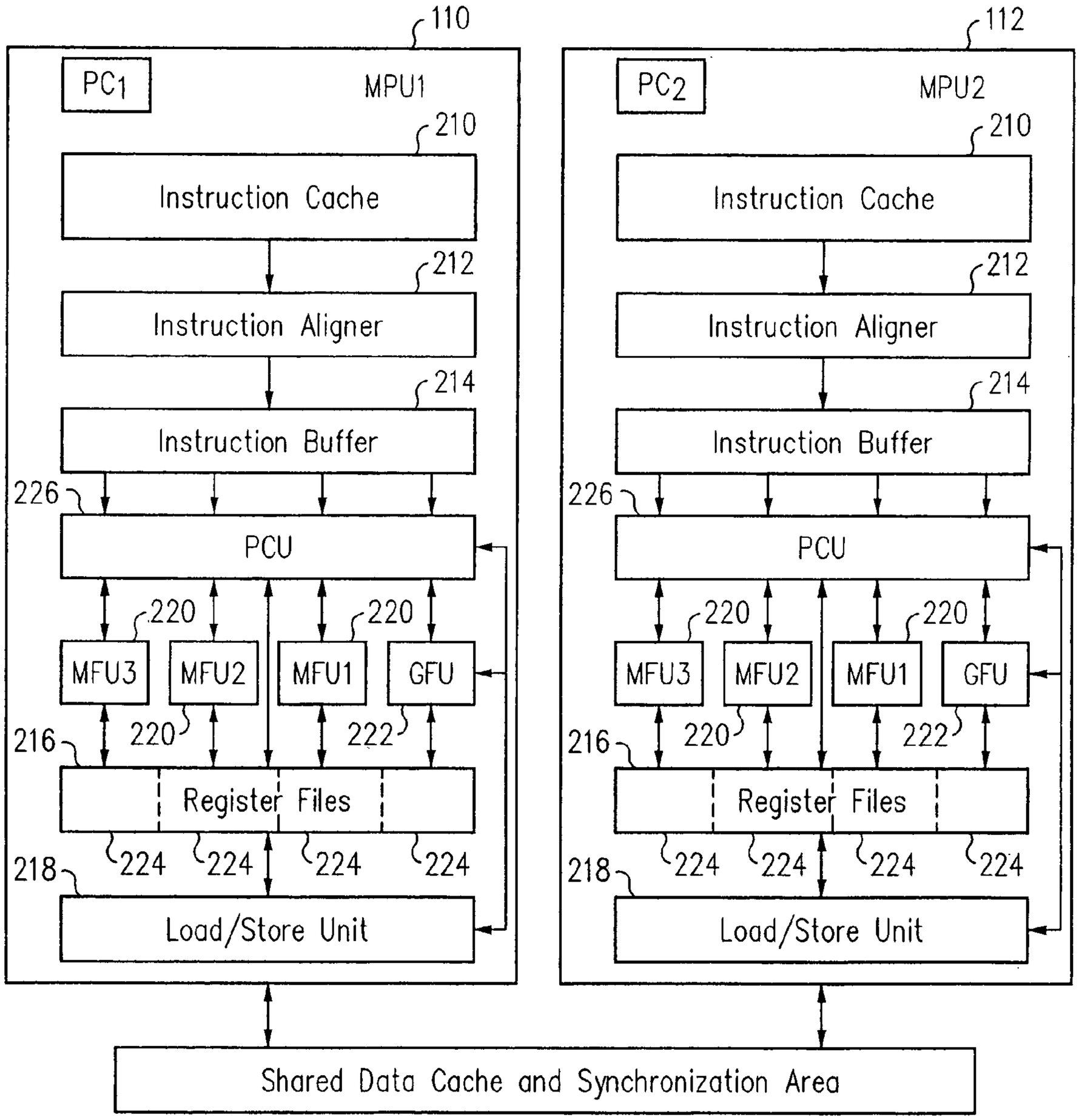

110
PC1
MPU1
210
Instruction Cache
212
Instruction Aligner
214
Instruction Buffer
226
PCU
220
220
MFU3
MFU2
MFU1
GFU
220
222
216
Register Files
224
224
224
224
218
Load/Store Unit
112
PC2
MPU2
210
Instruction Cache
212
Instruction Aligner
214
Instruction Buffer
226
PCU
220
220
MFU3
MFU2
MFU1
GFU
220
222
216
Register Files
224
224
224
224
218
Load/Store Unit
Shared Data Cache and Synchronization Area

FIG. 2

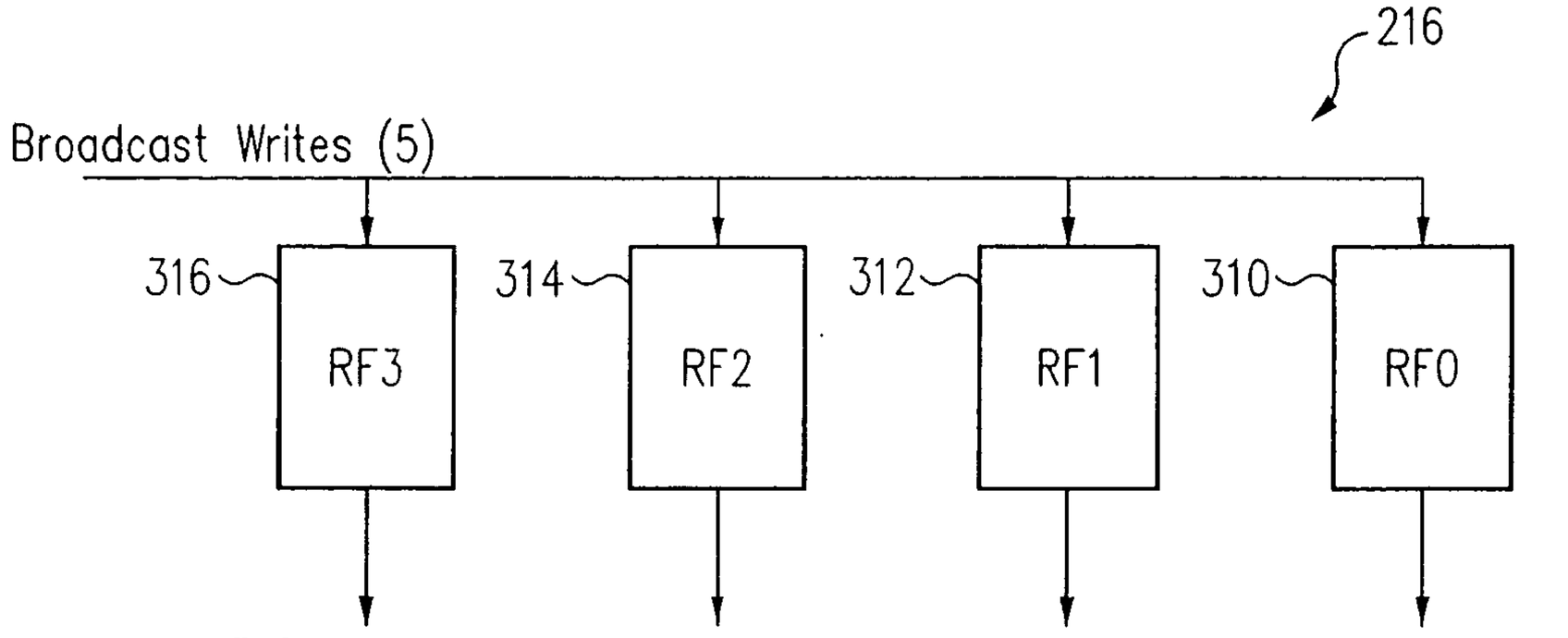

216
Broadcast Writes (5)
316
RF3
314
RF2
312
RF1
310
RF0
3 Read Ports
3 Read Ports
3 Read Ports
3 Read Ports

FIG. 3

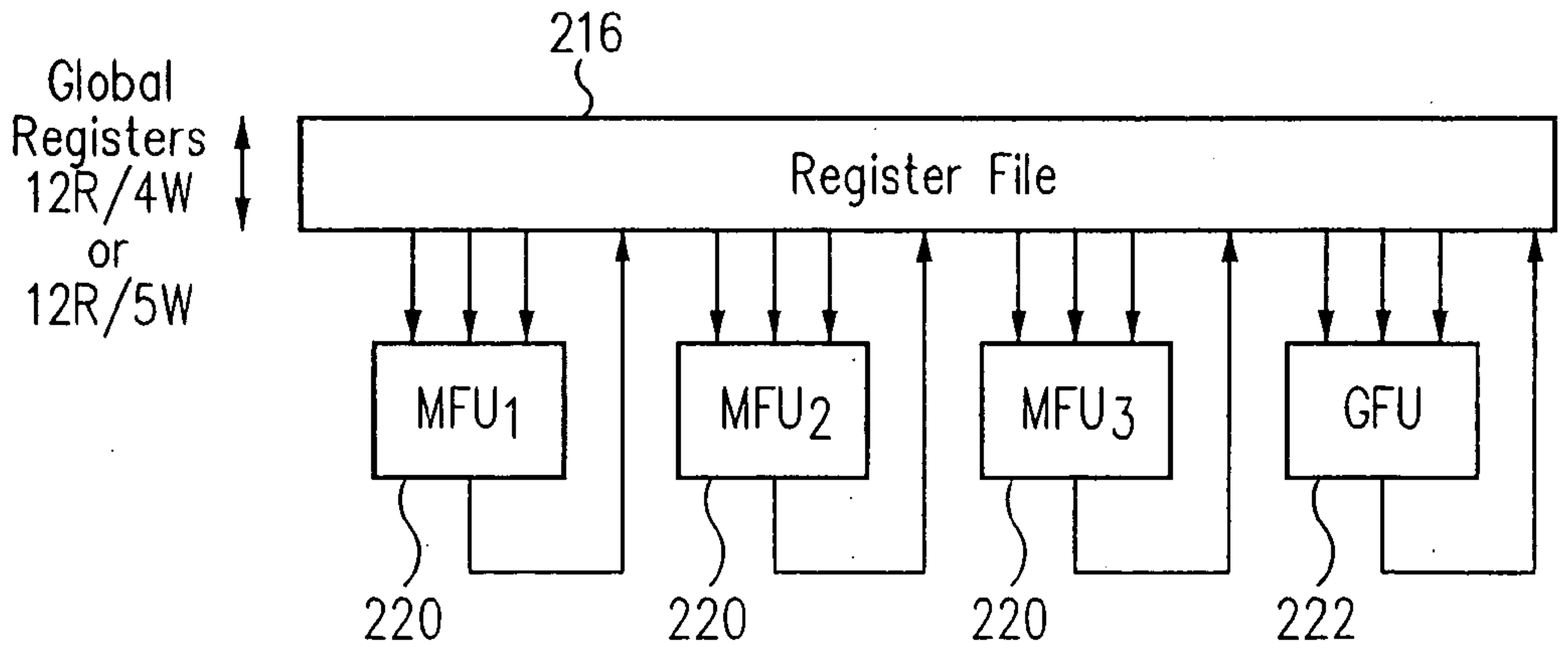

216
Global
Registers
12R/4W
or
12R/5W
Register File
MFU1
MFU2
MFU3
GFU
220
220
220
222

FIG. 4

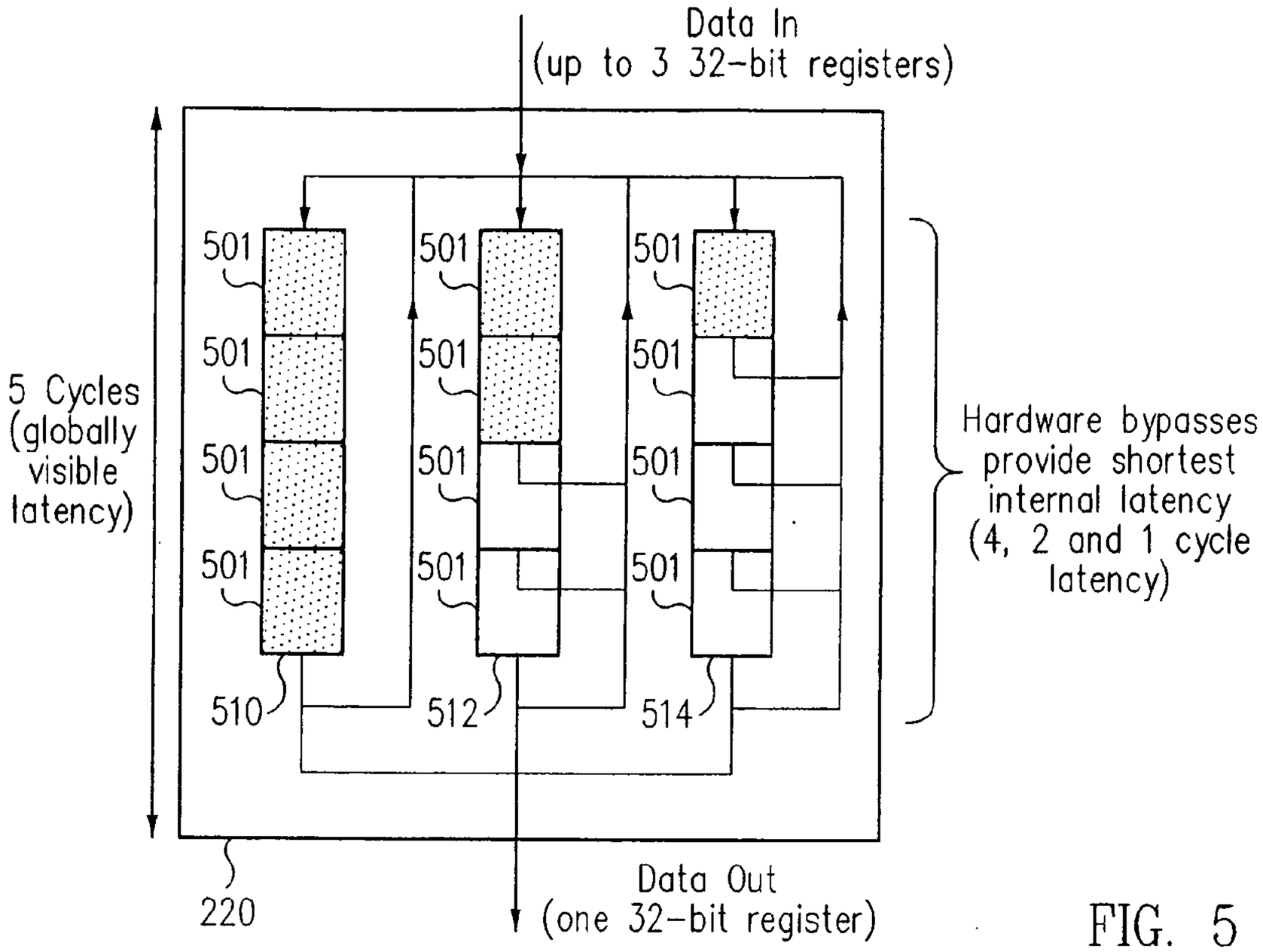

Data In
(up to 3 32-bit registers)
5 Cycles
(globally
visible
latency)
501
510
512
514
Hardware bypasses
provide shortest
internal latency
(4, 2 and 1 cycle
latency)
220
Data Out
(one 32-bit register)

FIG. 5

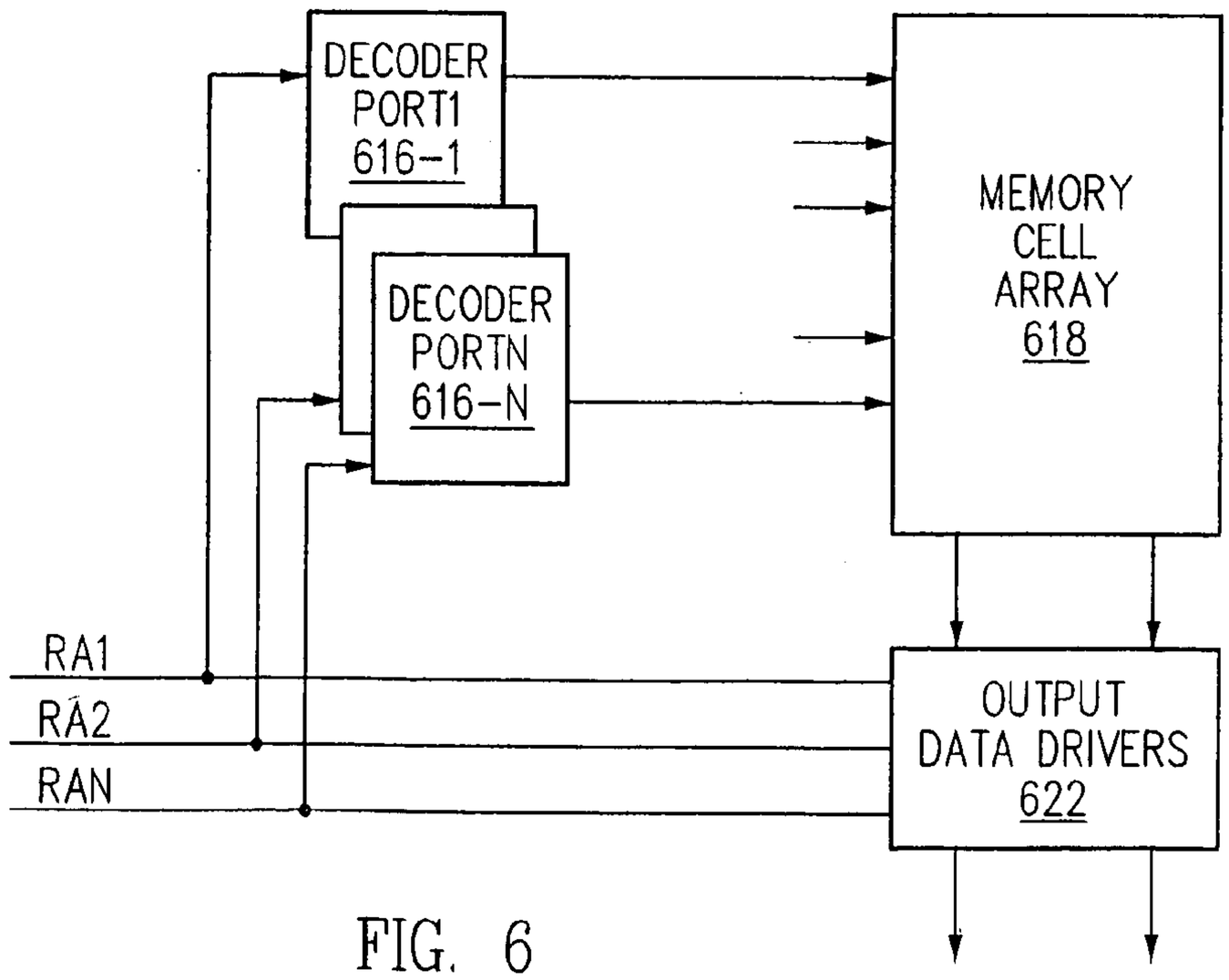

DECODER
PORT1
616-1
DECODER
PORTN
616-N
MEMORY
CELL
ARRAY
618
OUTPUT
DATA DRIVERS
622
RA1
RA2
RAN

FIG. 6

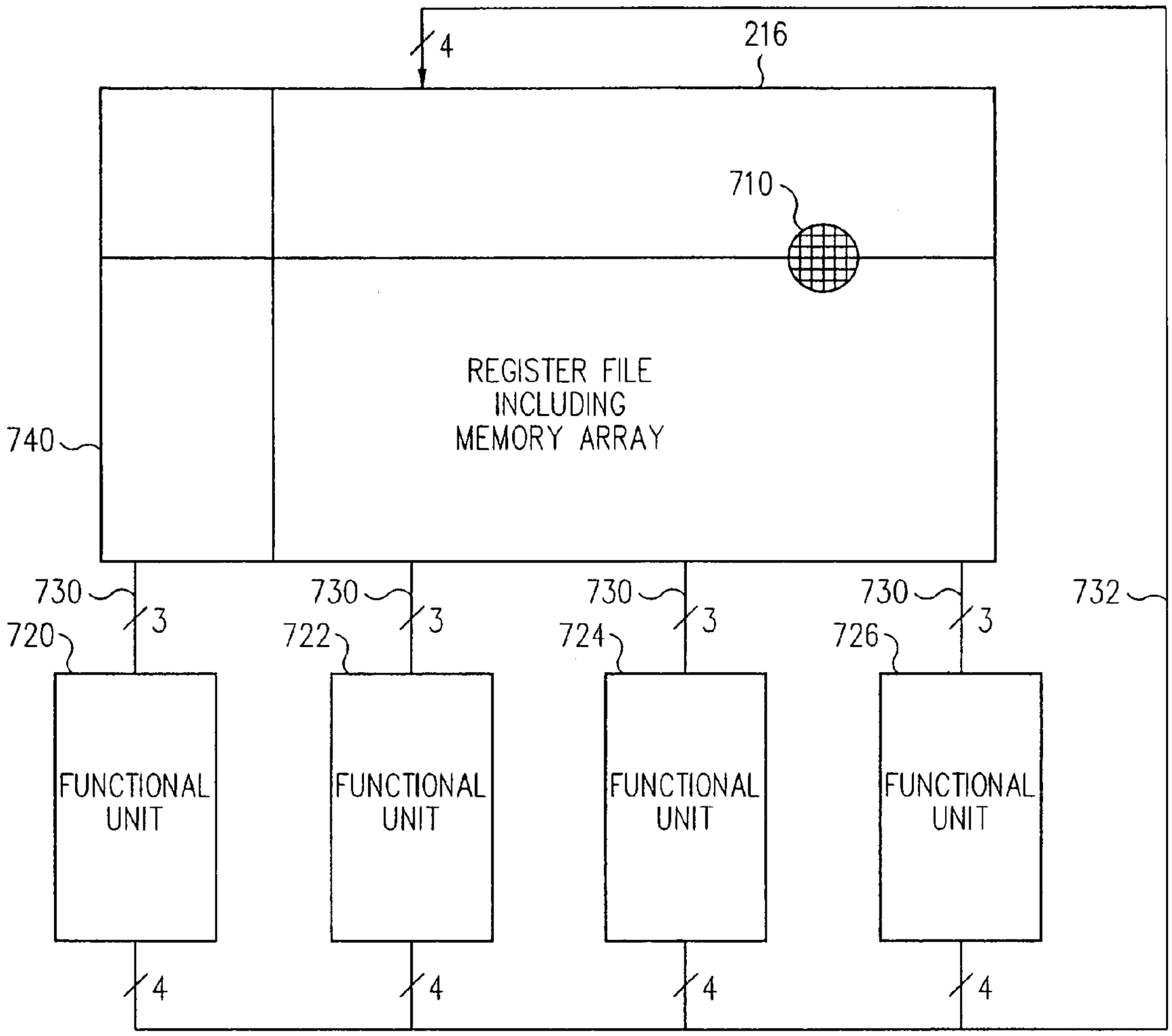

4
216
710
REGISTER FILE
INCLUDING
MEMORY ARRAY
740
730
3
720
722
724
726
732
FUNCTIONAL
UNIT
4

FIG. 7A

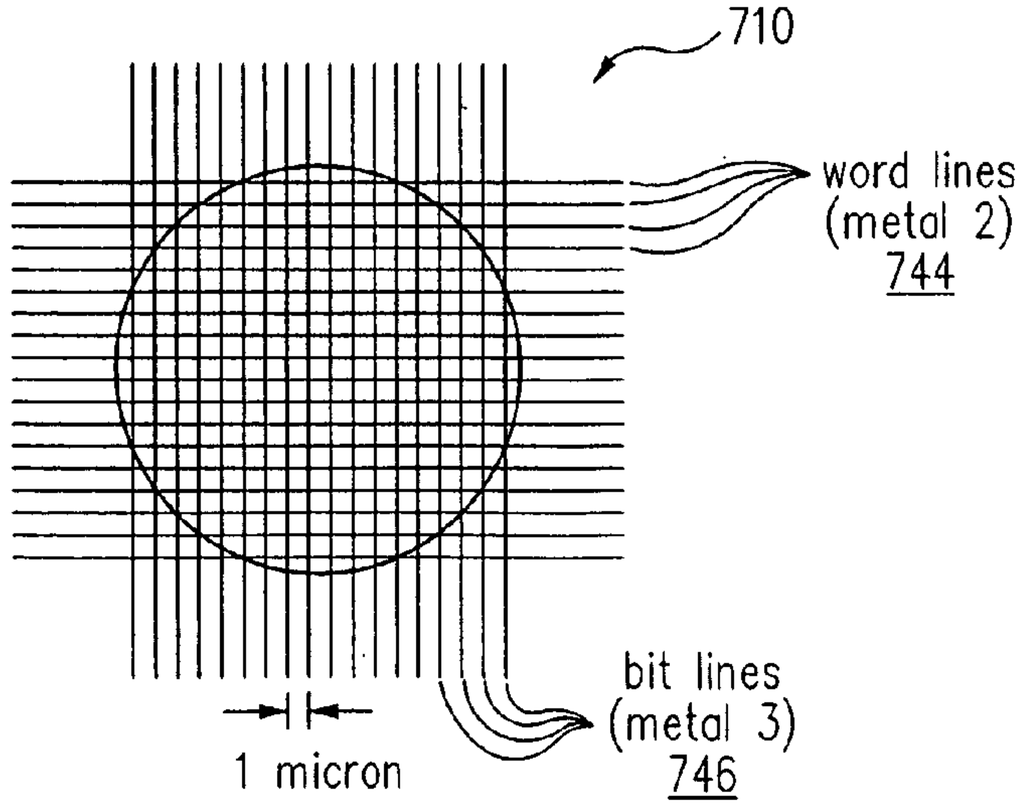

710
word lines
(metal 2)
744
bit lines
(metal 3)
746
1 micron

FIG. 7B

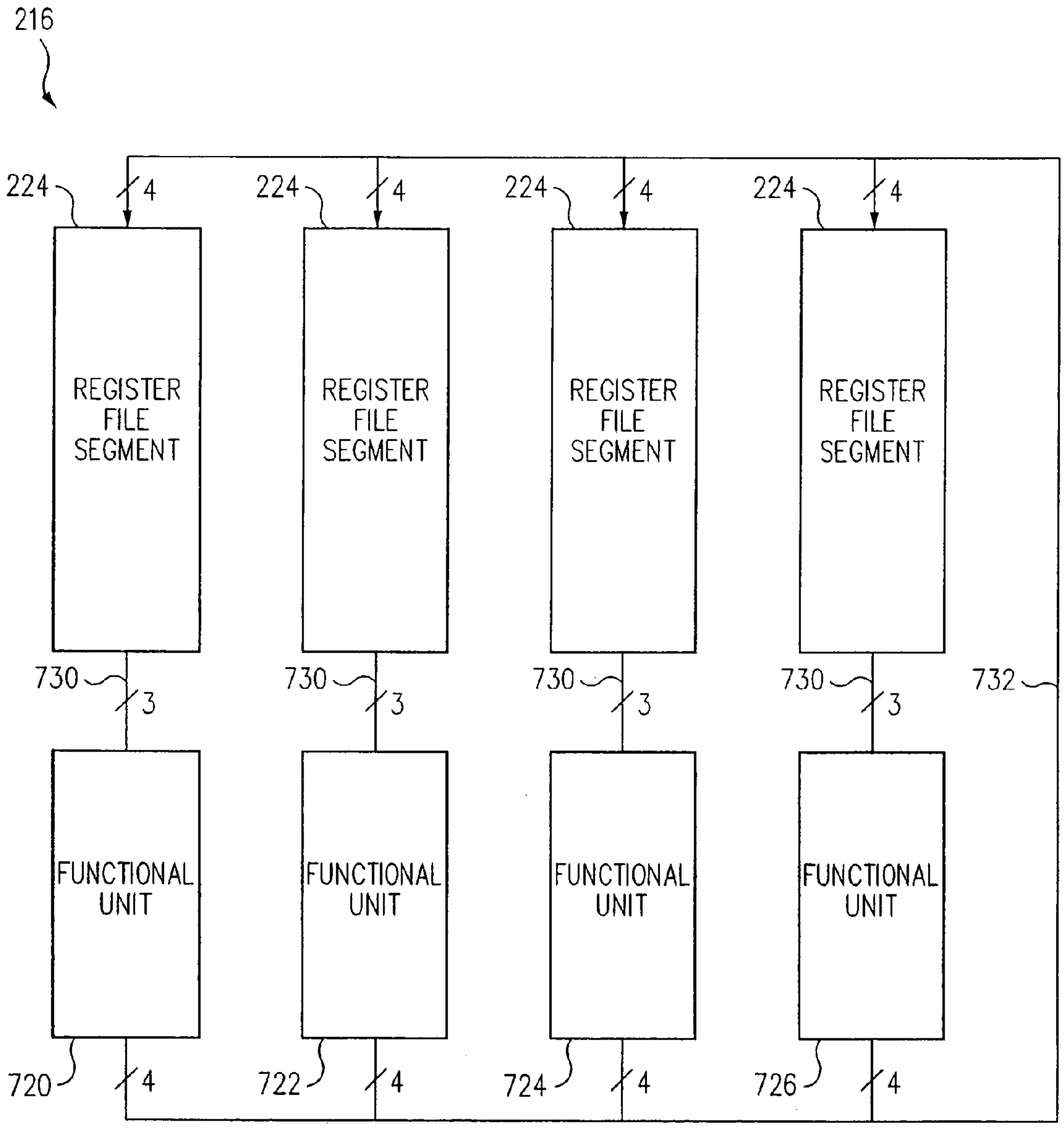

216
224
4
REGISTER FILE SEGMENT
730
3
FUNCTIONAL UNIT
720
722
724
726
732

FIG. 8

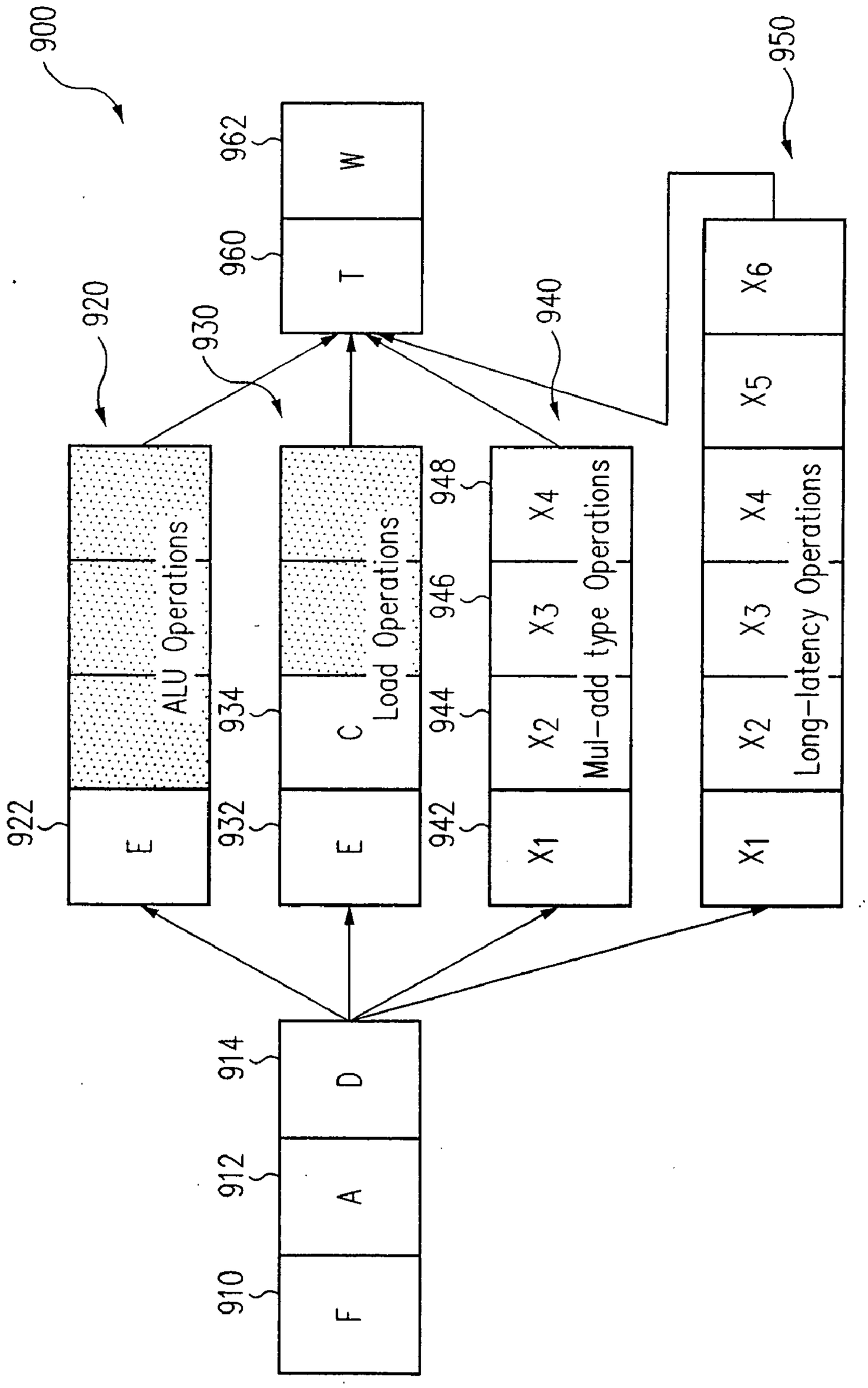

900
910
F
912
A
914
D
920
922
E
ALU Operations
930
932
E
934
C
Load Operations
940
942
X1
944
X2
946
X3
948
X4
Mul-add type Operations
950
X1
X2
X3
X4
X5
X6
Long-latency Operations

FIG. 9

2	8	8	8	8	6
hdr	opcode	rd	rs1	rs2	
hdr	opcode	rd	rs1	immediate	

FIG. 10A

2	8	8	22
hdr	opcode	rd	immediate

FIG. 10B

2	6	32
hdr	opcode	address

FIG. 10C

2	6	1	1	8	22
hdr	opcode	H	T	rd	displacement

FIG. 10D

1	1	6	8	8	8	8
0	0	opcode	rd	rs1	rs2	rs3
0	1	opcode	rd	rs1	rs2	immediate
1	0	opcode	rd	rs1	immediate	rs3
1	1	opcode	rd	rs1	immediate	immediate

FIG. 10E

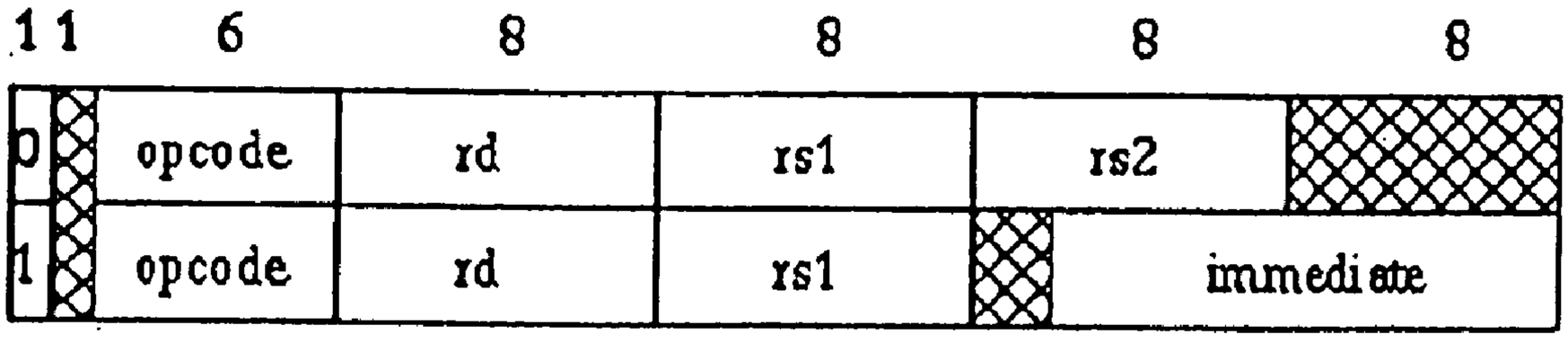


FIG. 10F

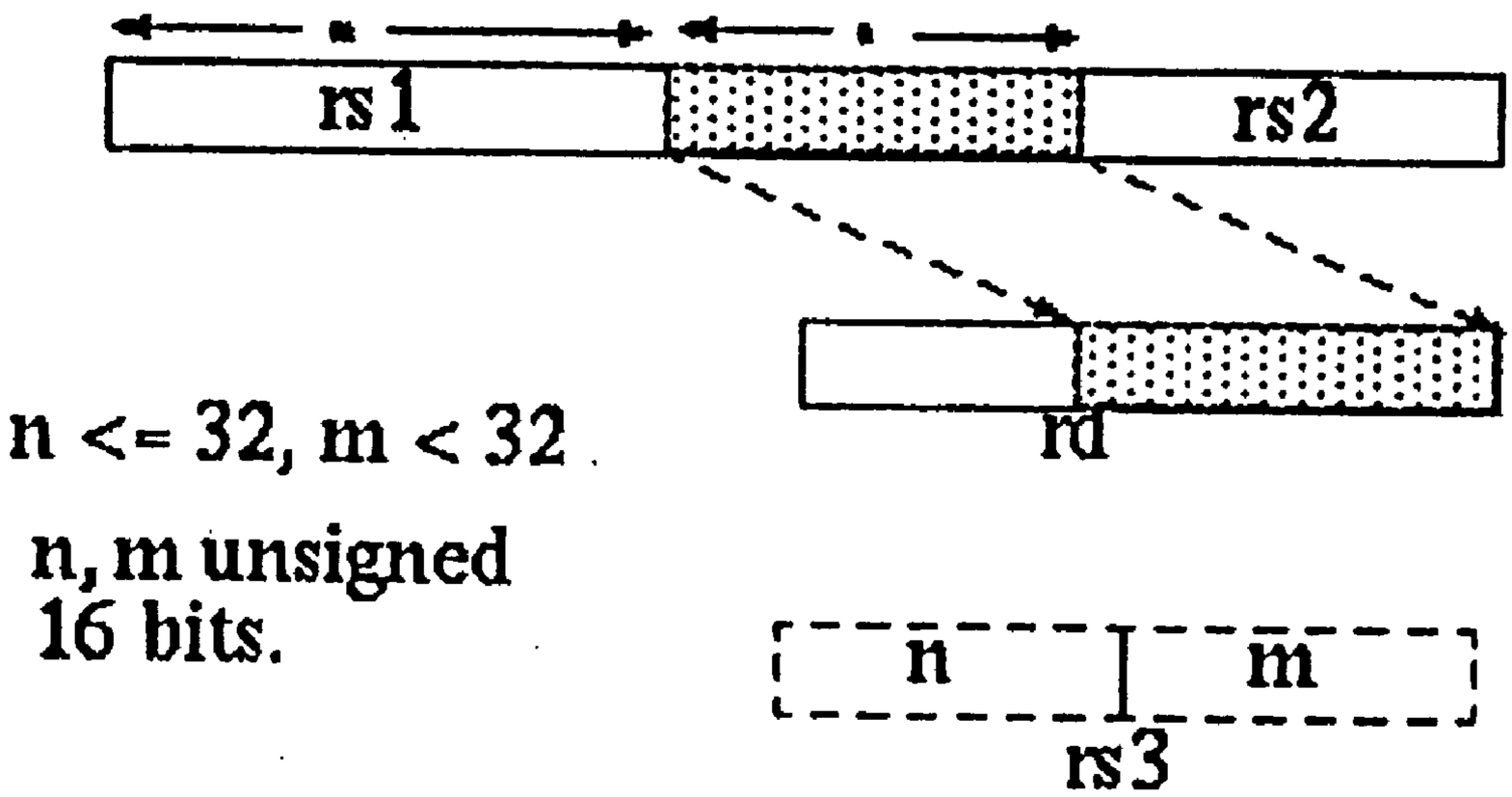


FIG. 11

MULTIPLE-THREAD PROCESSOR FOR THREADED SOFTWARE APPLICATIONS

BACKGROUND OF THE INVENTION

[0001] 1. Field of the Invention

[0002] The present invention relates to a processor architecture. More specifically, the present invention relates to a single-chip processor architecture including structures for multiple-thread operation.

[0003] 2. Description of the Related Art

[0004] For various processing applications, an automated system may handle multiple events or processes concurrently. A single process is termed a thread of control, or "thread", and is the basic unit of operation of independent dynamic action within the system. A program has at least one thread. A system performing concurrent operations typically has many threads, some of which are transitory and others enduring. Systems that execute among multiple processors allow for true concurrent threads. Single-processor systems can only have illusory concurrent threads, typically attained by time-slicing of processor execution, shared among a plurality of threads.

[0005] Some programming languages are particularly designed to support multiple-threading. One such language is the Java™ programming language that is advantageously executed using an abstract computing machine, the Java Virtual Machine™. A Java Virtual Machine™ is capable of supporting multiple threads of execution at one time. The multiple threads independently execute Java code that operates on Java values and objects residing in a shared main memory. The multiple threads may be supported using multiple hardware processors, by time-slicing a single hardware processor, or by time-slicing many hardware processors in 1990 programmers at Sun Microsystems developed a universal programming language, eventually known as "the Java™ programming language". Java™, Sun, Sun Microsystems and the Sun Logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks, including UltraSPARC I and UltraSPARC II, are used under license and are trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

[0006] Java™ supports the coding of programs that, though concurrent, exhibit deterministic behavior, by including techniques and structures for synchronizing the concurrent activity of threads. To synchronize threads, Java™ uses monitors, high-level constructs that allow only a single thread at one time to execute a region of code protected by the monitor. Monitors use locks associated with executable objects to control thread execution.

[0007] A thread executes code by performing a sequence of actions. A thread may use the value of a variable or assign the variable a new value. If two or more concurrent threads act on a shared variable, the actions on the variable may produce a timing-dependent result, an inherent consequence of concurrent programming.

[0008] Each thread has a working memory that may store copies of the values of master copies of variables from main memory that are shared among all threads. A thread usually accesses a shared variable by obtaining a lock and flushing the working memory of the thread, guaranteeing that shared values are thereafter loaded from the shared memory to the working memory of the thread. By unlocking a lock, a thread guarantees that the values held by the thread in the working memory are written back to the main memory.

[0009] Several rules of execution order constrain the order in which certain events may occur. For example, actions performed by one thread are totally ordered so that for any two actions performed by a thread, one action precedes the other. Actions performed by the main memory for any one variable are totally ordered so that for any two actions performed by the main memory on the same variable, one action precedes the other. Actions performed by the main memory for any one lock are totally ordered so that for any two actions performed by the main memory on the same lock, one action precedes the other. Also, an action is not permitted to follow itself Threads do not interact directly but rather only communicate through the shared main memory.

[0010] The relationships among the actions of a thread and the actions of main memory are also constrained by rules. For example, each lock or unlock is performed jointly by some thread and the main memory. Each load action by a thread is uniquely paired with a read action by the main memory such that the load action follows the read action. Each store action by a thread is uniquely paired with a write action by the main memory such that the write action follows the store action.

[0011] An implementation of threading incurs some overhead. For example, a single processor system incurs overhead in time-slicing between threads. Additional overhead is incurred in allocating and handling accessing of main memory and local thread working memory.

[0012] What is needed is a processor architecture that supports multiple-thread operation and reduces the overhead associated with multiple-thread operation.

SUMMARY OF THE INVENTION

[0013] A processor has an improved architecture for multiple-thread operation on the basis of a highly parallel structure including multiple independent parallel execution paths for executing in parallel across threads and a multiple-instruction parallel pathway within a thread. The multiple independent parallel execution paths include functional units that execute an instruction set including special data-handling instructions that are advantageous in a multiple-thread environment.

[0014] In accordance with one embodiment of the present invention, a general-purpose processor includes two independent processor elements in a single integrated circuit die. The dual independent processor elements advantageously execute two independent threads concurrently during multiple-threading operation. When only a single thread is executed on a first of the two processor elements, the second processor element is advantageously used to perform garbage collection, Just-In-Time (JIT) compilation, and the like. Illustratively, the independent processor elements are Very Long Instruction Word (VLIW) processors. For example, one illustrative processor includes two independent Very Long Instruction Word (VLIW) processor ele-

ments, each of which executes an instruction group or instruction packet that includes up to four instructions, otherwise termed subinstructions. Each of the instructions in an instruction group executes on a separate functional unit.

[0015] The two threads execute independently on the respective VLIW processor elements, each of which includes a plurality of powerful functional units that execute in parallel. In the illustrative embodiment, the VLIW processor elements have four functional units including three media functional units and one general functional unit. All of the illustrative media functional units include an instruction that executes both a multiply and an add in a single cycle, either floating point or fixed point.

[0016] In accordance with an aspect of the present invention, an individual independent parallel execution path has operational units including instruction supply blocks and instruction preparation blocks, functional units, and a register file that are separate and independent from the operational units of other paths of the multiple independent parallel execution paths. The instruction supply blocks include a separate instruction cache for the individual independent parallel execution paths, however the multiple independent parallel execution paths share a single data cache since multiple threads sometimes share data. The data cache is dual-ported, allowing data access in both execution paths in a single cycle.

[0017] In addition to the instruction cache, the instruction supply blocks in an execution path include an instruction aligner, and an instruction buffer that precisely format and align the full instruction group to prepare to access the register file. An individual execution path has a single register file that is physically split into multiple register file segments, each of which is associated with a particular functional unit of the multiple functional units. At any point in time, the register file segments as allocated to each functional unit each contain the same content. A multi-ported register file is typically metal limited to the area consumed by the circuit proportional with the square of the number of ports. It has been discovered that a processor having a register file structure divided into a plurality of separate and independent register files forms a layout structure with an improved layout efficiency. The read ports of the total register file structure are allocated among the separate and individual register files. Each of the separate and individual register files has write ports that correspond to the total number of write ports in the total register file structure. Writes are fully broadcast so that all of the separate and individual register files are coherent.

BRIEF DESCRIPTION OF THE DRAWINGS

[0018] The features of the described embodiments are specifically set forth in the appended claims. However, embodiments of the invention relating to both structure and method of operation, may best be understood by referring to the following description and accompanying drawings.

[0019] **FIG. 1** is a schematic block diagram illustrating a single integrated circuit chip implementation of a processor in accordance with an embodiment of the present invention.

[0020] **FIG. 2** is a schematic block diagram showing the core of the processor.

[0021] **FIG. 3** is a schematic block diagram that illustrates an embodiment of the split register file that is suitable for usage in the processor.

[0022] **FIG. 4** is a schematic block diagram that shows a logical view of the register file and functional units in the processor.

[0023] **FIG. 5** is a pictorial schematic diagram depicting an example of instruction execution among a plurality of media functional units.

[0024] **FIG. 6** illustrates a schematic block diagram of an SRAM array used for the multi-port split register file.

[0025] **FIGS. 7A and 7B** are, respectively, a schematic block diagram and a pictorial diagram that illustrate the register file and a memory array insert of the register file.

[0026] **FIG. 8** is a schematic block diagram showing an arrangement of the register file into the four register file segments.

[0027] **FIG. 9** is a schematic timing diagram that illustrates timing of the processor pipeline.

[0028] **FIGS. 10A, 10B, 10**C, **10**D, **10**E and **10**F illustrate instruction formats.

[0029] **FIG. 11** illustrates operation of a bitext instruction.

[0030] The use of the same reference symbols in different drawings indicates similar or identical items.

DESCRIPTION OF THE EMBODIMENT(S)

[0031] Referring to **FIG. 1**, a schematic block diagram illustrates a processor **100** having an improved architecture for multiple-thread operation on the basis of a highly parallel structure including multiple independent parallel execution paths, shown herein as two media processing units **110** and **112**. The execution paths execute in parallel across threads and include a multiple-instruction parallel pathway within a thread. The multiple independent parallel execution paths include functional units executing an instruction set having special data-handling instructions that are advantageous in a multiple-thread environment.

[0032] The multiple-threading architecture of the processor **100** is advantageous for usage in executing multiple-threaded applications using a language such as the Java™ language running under a multiple-threaded operating system on a multiple-threaded Java Virtual Machine™. The illustrative processor **100** includes two independent processor elements, the media processing units **110** and **112**, forming two independent parallel execution paths. A language that supports multiple threads, such as the Java™ programming language generates two threads that respectively execute in the two parallel execution paths with very little overhead incurred. The special instructions executed by the multiple-threaded processor include instructions for accessing arrays, and instructions that support garbage collection.

[0033] A single integrated circuit chip implementation of a processor **100** includes a memory interface **102**, a geometry decompressor **104**, the two media processing units **110** and **112**, a shared data cache **106**, and several interface controllers. The interface controllers support an interactive graphics environment with real-time constraints by integrating fundamental components of memory, graphics, and input/output bridge functionality on a single die. The components are mutually linked and closely linked to the processor core with high bandwidth, low-latency communica-

tion channels to manage multiple high-bandwidth data streams efficiently and with a low response time. The interface controllers include a an UltraPort Architecture Interconnect (UPA) controller **116** and a peripheral component interconnect (PCI) controller **120**. The illustrative memory interface **102** is a direct Rambus dynamic RAM (DRDRAM) controller. The shared data cache **106** is a dual-ported storage that is shared among the media processing units **110** and **112** with one port allocated to each media processing unit. The data cache **106** is four-way set associative, follows a write-back protocol, and supports hits in the fill buffer (not shown). The data cache **106** allows fast data sharing and eliminates the need for a complex, error-prone cache coherency protocol between the media processing units **110** and **112**.

[**0034**] The UPA controller **116** is a custom interface that attains a suitable balance between high-performance computational and graphic subsystems. The UPA is a cache-coherent, processor-memory interconnect. The UPA attains several advantageous characteristics including a scaleable bandwidth through support of multiple bused interconnects for data and addresses, packets that are switched for improved bus utilization, higher bandwidth, and precise interrupt processing. The UPA performs low latency memory accesses with high throughput paths to memory. The UPA includes a buffered cross-bar memory interface for increased bandwidth and improved scaleability. The UPA supports high-performance graphics with two-cycle single-word writes on the 64-bit UPA interconnect. The UPA interconnect architecture utilizes point-to-point packet switched messages from a centralized system controller to maintain cache coherence. Packet switching improves bus bandwidth utilization by removing the latencies commonly associated with transaction-based designs.

[**0035**] The PCI controller **120** is used as the primary system I/O interface for connecting standard, high-volume, low-cost peripheral devices, although other standard interfaces may also be used. The PCI bus effectively transfers data among high bandwidth peripherals and low bandwidth peripherals, such as CD-ROM players, DVD players, and digital cameras.

[**0036**] Two media processing units **110** and **112** are included in a single integrated circuit chip to support an execution environment exploiting thread level parallelism in which two independent threads can execute simultaneously. The threads may arise from any sources such as the same application, different applications, the operating system, or the runtime environment. Parallelism is exploited at the thread level since parallelism is rare beyond four, or even two, instructions per cycle in general purpose code. For example, the illustrative processor **100** is an eight-wide machine with eight execution units for executing instructions. A typical "general-purpose" processing code has an instruction level parallelism of about two so that, on average, most (about six) of the eight execution units would be idle at any time. The illustrative processor **100** employs thread level parallelism and operates on two independent threads, possibly attaining twice the performance of a processor having the same resources and clock rate but utilizing traditional non-thread parallelism.

[**0037**] Thread level parallelism is particularly useful for Java™ applications, which are bound to have multiple threads of execution. Java™ methods including "suspend", "resume", "sleep", and the like include effective support for threaded program code. In addition, Java™ class libraries are thread-safe to promote parallelism. Furthermore, the thread model of the processor **100** supports a dynamic compiler which runs as a separate thread using one media processing unit **110** while the second media processing unit **112** is used by the current application. In the illustrative system, the compiler applies optimizations based on "on-the-fly" profile feedback information while dynamically modifying the executing code to improve execution on each subsequent run. For example, a "garbage collector" may be executed on a first media processing unit **110**, copying objects or gathering pointer information, while the application is executing on the other media processing unit **112**.

[**0038**] Although the processor **100** shown in **FIG. 1** includes two processing units on an integrated circuit chip, the architecture is highly scaleable so that one to several closely-coupled processors may be formed in a message-based coherent architecture and resident on the same die to process multiple threads of execution. Thus, in the processor **100**, a limitation on the number of processors formed on a single die thus arises from capacity constraints of integrated circuit technology rather than from architectural constraints relating to the interactions and interconnections between processors.

[**0039**] The processor **100** is a general-purpose processor that includes the media processing units **110** and **112**, two independent processor elements in a single integrated circuit die. The dual independent processor elements **110** and **112** advantageously execute two independent threads concurrently during multiple-threading operation. When only a single thread executes on the processor **100**, one of the two processor elements executes the thread, the second processor element is advantageously used to perform garbage collection, Just-In-Time (JIT) compilation, and the like. In the illustrative processor **100**, the independent processor elements **110** and **112** are Very Long Instruction Word (VLIW) processors. For example, one illustrative processor **100** includes two independent Very Long Instruction Word (VLIW) processor elements, each of which executes an instruction group or instruction packet that includes up to four instructions. Each of the instructions in an instruction group executes on a separate functional unit.

[**0040**] The usage of a VLIW processor advantageously reduces complexity by avoiding usage of various structures such as schedulers or reorder buffers that are used in superscalar machines to handle data dependencies. A VLIW processor typically uses software scheduling and software checking to avoid data conflicts and dependencies, greatly simplifying hardware control circuits.

[**0041**] The two threads execute independently on the respective VLIW processor elements **110** and **112**, each of, which includes a plurality of powerful functional units that execute in parallel. In the illustrative embodiment shown in **FIG. 2**, the VLIW processor elements **110** and **112** have four functional units including three media functional units **220** and one general functional unit **222**. All of the illustrative media functional units **220** include an instruction that executes both a multiply and an add in a single cycle, either floating point or fixed point. Thus, a processor with two VLIW processor elements can execute twelve floating point

operations each cycle. At a 500 MHz execution rate, for example, the processor runs at an 6 gigaflop rate, even without accounting for general functional unit operation.

[**0042**] Referring to **FIG. 2**, a schematic block diagram shows the core of the processor **100**. The media processing units **110** and **112** each include an instruction cache **210**, an instruction aligner **212**, an instruction buffer **214**, a pipeline control unit **226**, a split register file **216**, a plurality of execution units, and a load/store unit **218**. In the illustrative processor **100**, the media processing units **110** and **112** use a plurality of execution units for executing instructions. The execution units for a media processing unit **110** include three media functional units (MFU) **220** and one general functional unit (GFU) **222**.

[**0043**] An individual independent parallel execution path **110** or **112** has operational units including instruction supply blocks and instruction preparation blocks, functional units **220** and **222**, and a register file **216** that are separate and independent from the operational units of other paths of the multiple independent parallel execution paths. The instruction supply blocks include a separate instruction cache **210** for the individual independent parallel execution paths, however the multiple independent parallel execution paths share a single data cache **106** since multiple threads sometimes share data. The data cache **106** is dual-ported, allowing data access in both execution paths **110** and **112** in a single cycle. Sharing of the data cache **106** among independent processor elements **110** and **112** advantageously simplifies data handling, avoiding a need for a cache coordination protocol and the overhead incurred in controlling the protocol.

[**0044**] In addition to the instruction cache **210**, the instruction supply blocks in an execution path include the instruction aligner **212**, and the instruction buffer **214** that precisely format and align a full instruction group of four instructions to prepare to access the register file **216**. An individual execution path has a single register file **216** that is physically split into multiple register file segments, each of which is associated with a particular functional unit of the multiple functional units. At any point in time, the register file segments as allocated to each functional unit each contain the same content. A multi-ported register file is typically metal limited to the area consumed by the circuit proportional with the square of the number of ports. The processor **100** has a register file structure divided into a plurality of separate and independent register files to form a layout structure with an improved layout efficiency. The read ports of the total register file structure **216** are allocated among the separate and individual register files. Each of the separate and individual register files has write ports that correspond to the total number of write ports in the total register file structure. Writes are fully broadcast so that all of the separate and individual register files are coherent.

[**0045**] The media functional units **220** are multiple single-instruction-multiple-datapath (MSIMD) media functional units. Each of the media functional units **220** is capable of processing parallel 16-bit components. Various parallel 16-bit operations supply the single-instruction-multiple-datapath capability for the processor **100** including add, multiply-add, shift, compare, and the like. The media functional units **220** operate in combination as tightly coupled digital signal processors (DSPs). Each media functional unit **220** has an separate and individual sub-instruction stream, but all tree media functional units **220** execute synchronously so that the subinstructions progress lock-step through pipeline stages.

[**0046**] The general functional unit **222** is a RISC processor capable of executing arithmetic logic unit (ALU) operations, loads and stores, branches, and various specialized and esoteric functions such as parallel power operations, reciprocal square root operations, and many others. The general functional unit **222** supports less common parallel operations such as the parallel reciprocal square root instruction:

[**0047**] The illustrative instruction cache **210** is two-way set-associative, has a 16 Kbyte capacity, and includes hardware support to maintain coherence, allowing dynamic optimizations through self-modifying code. Software is used to indicate that the instruction storage is being modified when modifications occur. The **16**K capacity is suitable for performing graphic loops, other multimedia tasks or processes, and general-purpose Java™ code. Coherency is maintained by hardware that supports write-through, non-allocating caching. Self-modifying code is supported through explicit use of "store-to-instruction-space" instruction store2i. Software uses the store2i instruction to maintain coherency with the instruction cache **210** so that the instruction caches **210** do not have to be snooped on every single store operation issued by the media processing unit **110**.

[**0048**] The pipeline control unit **226** is connected between the instruction buffer **214** and the functional units and schedules the transfer of instructions to the functional units. The pipeline control unit **226** also receives status signals from the functional units and the load/store unit **218** and uses the status signals to perform several control functions. The pipeline control unit **226** maintains a scoreboard, generates stalls and bypass controls. The pipeline control unit **226** also generates traps and maintains special registers.

[**0049**] Each media processing unit **110** and **112** includes a split register file **216**, a single logical register file including **128** thirty-two bit registers. The split register file **216** is split into a plurality of register file segments **224** to form a multi-ported structure that is replicated to reduce the integrated circuit die area and to reduce access time. A separate register file segment **224** is allocated to each of the media functional units **220** and the general functional unit **222**. In the illustrative embodiment, each register file segment **224** has 128 32-bit registers. The first 96 registers (0-95) in the register file segment **224** are global registers. All functional units can write to the 96 global registers. The global registers are coherent across all functional units (MFU and GFU) so that any write operation to a global register by any functional unit is broadcast to all register file segments **224**. Registers 96-127 in the register file segments **224** are local registers. Local registers allocated to a functional unit are not accessible or "visible" to other functional units.

[**0050**] The media processing units **110** and **112** are highly structured computation blocks that execute software-scheduled data computation operations with fixed, deterministic and relatively short instruction latencies, operational characteristics yielding simplification in both function and cycle time. The operational characteristics support multiple instruction issue through a pragmatic very large instruction word (VLIW) approach that avoids hardware interlocks to

account for software that does not schedule operations properly. Such hardware interlocks are typically complex, error-prone, and create multiple critical paths. A VLIW instruction word always includes one instruction that executes in the general functional unit (GFU) **222** and from zero to three instructions that execute in the media functional units (MFU) **220**. A MFU instruction field within the VLIW instruction word includes an operation code (opcode) field, three source register (or immediate) fields, and one destination register field.

[**0051**] Instructions are executed in-order in the processor **100** but loads can finish out-of-order with respect to other instructions and with respect to other loads, allowing loads to be moved up in the instruction stream so that data can be streamed from main memory. The execution model eliminates the usage and overhead resources of an instruction window, reservation stations, a re-order buffer, or other blocks for handling instruction ordering. Elimination of the instruction ordering structures and overhead resources is highly advantageous since the eliminated blocks typically consume a large portion of an integrated circuit die. For example, the eliminated blocks consume about 30% of the die area of a Pentium II processor.

[**0052**] To avoid software scheduling errors, the media processing units **110** and **112** are high-performance but simplified with respect to both compilation and execution. The media processing units **110** and **112** are most generally classified as a simple 2-scalar execution engine with full bypassing and hardware interlocks on load operations. The instructions include loads, stores, arithmetic and logic (ALU) instructions, and branch instructions so that scheduling for the processor **100** is essentially equivalent to scheduling for a simple 2-scalar execution engine for each of the two media processing units **110** and **112**.

[**0053**] The processor **100** supports full bypasses between the first two execution units within the media processing unit **110** and **112** and has a scoreboard in the general functional unit **222** for load operations so that the compiler does not need to handle nondeterministic latencies due to cache misses. The processor **100** scoreboards long latency operations that are executed in the general functional unit **222**, for example a reciprocal square-root operation, to simplify scheduling across execution units. The scoreboard (not shown) operates by tracking a record of an instruction packet or group from the time the instruction enters a functional unit until the instruction is finished and the result becomes available. A VLIW instruction packet contains one GFU instruction and from zero to three MFU instructions. The source and destination registers of all instructions in an incoming VLIW instruction packet are checked against the scoreboard. Any true dependencies or output dependencies stall the entire packet until the result is ready. Use of a scoreboarded result as an operand causes instruction issue to stall for a sufficient number of cycles to allow the result to become available. If the referencing instruction that provokes the stall executes on the general functional unit **222** or the first media functional unit **220**, then the stall only endures until the result is available for intra-unit bypass. For the case of a load instruction that hits in the data cache **106**, the stall may last only one cycle. If the referencing instruction is on the second or third media functional units **220**, then the stall endures until the result reaches the writeback stage in the pipeline where the result is bypassed in transmission to the split register file **216**.

[**0054**] The scoreboard automatically manages load delays that occur during a load hit. In an illustrative embodiment, all loads enter the scoreboard to simplify software scheduling and eliminate NOPs in the instruction stream.

[**0055**] The scoreboard is used to manage most interlock conditions between the general functional unit **222** and the media functional units **220**. All loads and non-pipelined long-latency operations of the general functional unit **222** are scoreboarded. The long-latency operations include division idiv, fdiv instructions, reciprocal square root frecsqrt, precsqrt instructions, and power ppower instructions. None of the results of the media functional units **220** is scoreboarded. Non-scoreboarded results are available to subsequent operations on the functional unit that produces the results following the latency of the instruction.

[**0056**] The illustrative processor **100** has a rendering rate of over fifty million triangles per second without accounting for operating system overhead. Therefore, data feeding specifications of the processor **100** are far beyond the capabilities of cost-effective memory systems. Sufficient data bandwidth is achieved by rendering of compressed geometry using the geometry decompressor **104**, an on-chip real-time geometry decompression engine. Data geometry is stored in main memory in a compressed format. At render time, the data geometry is fetched and decompressed in real-time on the integrated circuit of the processor **100**. The geometry decompressor **104** advantageously saves memory space and memory transfer bandwidth. The compressed geometry uses an optimized generalized mesh structure that explicitly calls out most shared vertices between triangles, allowing the processor **100** to transform and light most vertices only once. In a typical compressed mesh, the triangle throughput of the transform-and-light stage is increased by a factor of four or more over the throughput for isolated triangles. For example, during processing of triangles, multiple vertices are operated upon in parallel so that the utilization rate of resources is high, achieving effective spatial software pipelining. Thus operations are overlapped in time by operating on several vertices simultaneously, rather than overlapping several loop iterations in time. For other types of applications with high instruction level parallelism, high trip count loops are software-pipelined so that most media functional units **220** are fully utilized.

[**0057**] Referring to **FIG. 3**, a schematic block diagram illustrates an embodiment of the split register file **216** that is suitable for usage in the processor **100**. The split register file **216** supplies all operands of processor instructions that execute in the media functional units **220** and the general functional units **222** and receives results of the instruction execution from the execution units. The split register file **216** operates as an interface to the geometry decompressor **104**. The split register file **216** is the source and destination of store and load operations, respectively.

[**0058**] In the illustrative processor **100**, the split register file **216** in each of the media processing units **110** and **112** has 128 registers. Graphics processing places a heavy burden on register usage. Therefore, a large number of registers is supplied by the split register file **216** so that performance is not limited by loads and stores or handling of intermediate results including graphics “fills” and “spills”. The illustra-

tive split register file **216** includes twelve read ports and five write ports, supplying total data read and write capacity between the central registers of the split register file **216** and all media functional units **220** and the general functional unit **222**. The five write ports include one 64-bit write port that is dedicated to load operations. The remaining four write ports are **32** bits wide and are used to write operations of the general functional unit **222** and the media functional units **220**.

[**0059**] A large total read and write capacity promotes flexibility and facility in programming both of hand-coded routines and compiler-generated code.

[**0060**] Large, multiple-ported register files are typically metal-limited so that the register area is proportional with the square of the number of ports. A sixteen port file is roughly proportional in size and speed to a value of **256**. The illustrative split register file **216** is divided into four register file segments **310**, **312**, **314**, and **316**, each having three read ports and four write ports so that each register file segment has a size and speed proportional to 49 for a total area for the four segments that is proportional to 196. The total area is therefore potentially smaller and faster than a single central register file. Write operations are fully broadcast so that all files are maintained coherent. Logically, the split register file **216** is no different from a single central register file. However, from the perspective of layout efficiency, the split register file **216** is highly advantageous, allowing for reduced size and improved performance.

[**0061**] The new media data that is operated upon by the processor **100** is typically heavily compressed. Data transfers are communicated in a compressed format from main memory and input/output devices to pins of the processor **100**, subsequently decompressed on the integrated circuit holding the processor **100**, and passed to the split register file **216**.

[**0062**] Splitting the register file into multiple segments in the split register file **216** in combination with the character of data accesses in which multiple bytes are transferred to the plurality of execution units concurrently, results in a high utilization rate of the data supplied to the integrated circuit chip and effectively leads to a much higher data bandwidth than is supported on general-purpose processors. The highest data bandwidth requirement is therefore not between the input/output pins and the central processing units, but is rather between the decompressed data source and the remainder of the processor. For graphics processing, the highest data bandwidth requirement is between the geometry decompressor **104** and the split register file **216**. For video decompression, the highest data bandwidth requirement is internal to the split register file **216**. Data transfers between the geometry decompressor **104** and the split register file **216** and data transfers between various registers of the split register file **216** can be wide and run at processor speed, advantageously delivering a large bandwidth.

[**0063**] The register file **216** is a focal point for attaining the very large bandwidth of the processor **100**. The processor **100** transfers data using a plurality of data transfer techniques. In one example of a data transfer technique, cacheable data is loaded into the split register file **216** through normal load operations at a low rate of up to eight bytes per cycle. In another example, streaming data is transferred to the split register file **216** through group load operations, which transfer thirty-two bytes from memory directly into eight consecutive 32-bit registers. The processor **100** utilizes the streaming data operation to receive compressed video data for decompression.

[**0064**] Compressed graphics data is received via a direct memory access (DMA) unit in the geometry decompressor **104**. The compressed graphics data is decompressed by the geometry decompressor **104** and loaded at a high bandwidth rate into the split register file **216** via group load operations that are mapped to the geometry decompressor **104**.

[**0065**] Load operations are non-blocking and scoreboarded so that early scheduling can hide a long latency inherent to loads.

[**0066**] General purpose applications often fail to exploit the large register file **216**. Statistical analysis shows that compilers do not effectively use the large number of registers in the split register file **216**. However, aggressive in-lining techniques that have traditionally been restricted due to the limited number of registers in conventional systems may be advantageously used in the processor **100** to exploit the large number of registers in the split register file **216**. In a software system that exploits the large number of registers in the processor **100**, the complete set of registers is saved upon the event of a thread (context) switch. When only a few registers of the entire set of registers is used, saving all registers in the full thread switch is wasteful. Waste is avoided in the processor **100** by supporting individual marking of registers. Octants of the thirty-two registers can be marked as "dirty" if used, and are consequently saved conditionally.

[**0067**] In various embodiments, dedicating fields for globals, trap registers, and the like leverages the split register file **216**.

[**0068**] Referring to **FIG. 4**, a schematic block diagram shows a logical view of the register file **216** and functional units in the processor **100**. The physical implementation of the core processor **100** is simplified by replicating a single functional unit to form the three media functional units **220**. The media functional units **220** include circuits that execute various arithmetic and logical operations including general-purpose code, graphics code, and video-image-speech (VIS) processing. VIS processing includes video processing, image processing, digital signal processing (DSP) loops, speech processing, and voice recognition algorithms, for example.

[**0069**] Referring to **FIG. 5**, a simplified pictorial schematic diagram depicts an example of instruction execution among a plurality of media functional units **220**. Results generated by various internal function blocks within a first individual media functional unit are immediately accessible internally to the first media functional unit **510** but are only accessible globally by other media functional units **512** and **514** and by the general functional unit five cycles after the instruction enters the first media functional unit **510**, regardless of the actual latency of the instruction. Therefore, instructions executing within a functional unit can be scheduled by software to execute immediately, taking into consideration the actual latency of the instruction. In contrast, software that schedules instructions executing in different functional units is expected to account for the five cycle latency. In the diagram, the shaded areas represent the stage

at which the pipeline completes execution of an instruction and generates final result values. A result is not available internal to a functional unit a final shaded stage completes. In the example, media processing unit instructions have three different latencies—four cycles for instructions such as fmuladd and fadd, two cycles for instructions such as pmuladd, and one cycle for instructions like padd and xor.

[**0070**] Although internal bypass logic within a media functional unit **220** forwards results to execution units within the same media functional unit **220**, the internal bypass logic does not detect incorrect attempts to reference a result before the result is available.

[**0071**] Software that schedules instructions for which a dependency occurs between a particular media functional unit, for example **512**, and other media functional units **510** and **514**, or between the particular media functional unit **512** and the general functional unit **222**, is to account for the five cycle latency between entry of an instruction to the media functional unit **512** and the five cycle pipeline duration.

[**0072**] Referring to **FIG. 6**, a schematic block diagram depicts an embodiment of the multiport register file **216**. A plurality of read address buses RA1 through RAN carry read addresses that are applied to decoder ports **616-1** through **616**-N, respectively. Decoder circuits are well known to those of ordinary skill in the art, and any of several implementations could be used as the decoder ports **616-1** through **616**-N. When an address is presented to any of decoder ports **616-1** through **616**-N, the address is decoded and a read address signal is transmitted by a decoder port **616** to a register in a memory cell array **618**. Data from the memory cell array **618** is output using output data drivers **622**. Data is transferred to and from the memory cell array **618** under control of control signals carried on some of the lines of the buses of the plurality of read address buses RA1 through RAN.

[**0073**] Referring to **FIGS. 7A and 7B**, a schematic block diagram and a pictorial diagram, respectively, illustrate the register file **216** and a memory array insert **710**. The register file **216** is connected to a four functional units **720**, **722**, **724**, and **726** that supply information for performing operations such as arithmetic, logical, graphics, data handling operations and the like. The illustrative register file **216** has twelve read ports **730** and four write ports **732**. The twelve read ports **730** are illustratively allocated with three ports connected to each of the four functional units. The four write ports **732** are connected to receive data from all of the four functional units.

[**0074**] The register file **216** includes a decoder, as is shown in **FIG. 6**, for each of the sixteen read and write ports. The register file **216** includes a memory array **740** that is partially shown in the insert **710** illustrated in **FIG. 7B** and includes a plurality of word lines **744** and bit lines **746**. The word lines **744** and bit lines **746** are simply a set of wires that connect transistors (not shown) within the memory array **740**. The word lines **744** select registers so that a particular word line selects a register of the register file **216**. The bit lines **746** are a second set of wires that connect the transistors in the memory array **740**. Typically, the word lines **744** and bit lines **746** are laid out at right angles. In the illustrative embodiment, the word lines **744** and the bit lines **746** are constructed of metal laid out in different planes such as a metal **2** layer for the word lines **744** and a metal **3** layer for the bit lines **746**. In other embodiments, bit lines and word lines may be constructed of other materials, such as polysilicon, or can reside at different levels than are described in the illustrative embodiment, that are known in the art of semiconductor manufacture. In the illustrative example, a distance of about lm separates the word lines **744** and a distance of approximately 1 μm separates the bit lines **746**. Other circuit dimensions may be constructed for various processes. The illustrative example shows one bit line per port, other embodiments may use multiple bit lines per port.

[**0075**] When a particular functional unit reads a particular register in the register file **216**, the functional unit sends an address signal via the read ports **730** that activates the appropriate word lines to access the register. In a register file having a conventional structure and twelve read ports, each cell, each storing a single bit of information, is connected to twelve word lines to select an address and twelve bit lines to carry data read from the address.

[**0076**] The four write ports **732** address registers in the register file using four word lines **744** and four bit lines **746** connected to each cell. The four word lines **744** address a cell and the four bit lines **746** carry data to the cell.

[**0077**] Thus, if the illustrative register file **216** were laid out in a conventional manner with twelve read ports **730** and four write ports **732** for a total of sixteen ports and the ports were 1 μm apart, one memory cell would have an integrated circuit area of 256 μm^2 (16×16). The area is proportional to the square of the number of ports.

[**0078**] The register file **216** is alternatively implemented to perform single-ended reads and/or single-ended writes utilizing a single bit line per port per cell, or implemented to perform differential reads and/or differential writes using two bit lines per port per cell.

[**0079**] However, in this embodiment the register file **216** is not laid out in the conventional manner and instead is split into a plurality of separate and individual register file segments **224**. Referring to **FIG. 8**, a schematic block diagram shows an arrangement of the register file **216** into the four register file segments **224**. The register file **216** remains operational as a single logical register file in the sense that the four of the register file segments **224** contain the same number of registers and the same register values as a conventional register file of the same capacity that is not split. The separated register file segments **224** differ from a register file that is not split through elimination of lines that would otherwise connect ports to the memory cells. Accordingly, each register file segment **224** has connections to only three of the twelve read ports **730**, lines connecting a register file segment to the other nine read ports are eliminated. All writes are broadcast so that each of the four register file segments **224** has connections to all four write ports **732**. Thus each of the four register file segments **224** has three read ports and four write ports for a total of seven ports. The individual cells are connected to seven word lines and seven bit lines so that a memory array with a spacing of 1 μm between lines has an area of approximately 49 μm^2. In the illustrative embodiment, the four register file segments **224** have an area proportion to seven squared. The total area of the four register file segments **224** is therefore proportional to 49 times 4, a total of 196.

[**0080**] The split register file thus advantageously reduces the area of the memory array by a ratio of approximately

256/196 (1.3× or 30%). The reduction in area further advantageously corresponds to an improvement in speed performance due to a reduction in the length of the word lines **744** and the bit lines **746** connecting the array cells that reduces the time for a signal to pass on the lines. The improvement in speed performance is highly advantageous due to strict time budgets that are imposed by the specification of high-performance processors and also to attain a large capacity register file that is operational at high speed. For example, the operation of reading the register file **216** typically takes place in a single clock cycle. For a processor that executes at 500 MHz, a cycle time of two nanoseconds is imposed for accessing the register file **216**. Conventional register files typically only have up to about 32 registers in comparison to the 128 registers in the illustrative register file **216** of the processor **100**. A register file **216** substantially larger than the register file in conventional processors is highly advantageous in high-performance operations such as video and graphic processing. The reduced size of the register file **216** is highly useful for complying with time budgets in a large capacity register file.

[**0081**] Referring to **FIG. 9**, a simplified schematic timing diagram illustrates timing of the processor pipeline **900**. The pipeline **900** includes nine stages including three initiating stages, a plurality of execution phases, and two terminating stages. The three initiating stages are optimized to include only those operations necessary for decoding instructions so that jump and call instructions, which are pervasive in the Java™ language, execute quickly. Optimization of the initiating stages advantageously facilitates branch prediction since branches, jumps, and calls execute quickly and do not introduce many bubbles.

[**0082**] The first of the initiating stages is a fetch stage **910** during which the processor **100** fetches instructions from the 16 Kbyte two-way set-associative instruction cache **210**. The fetched instructions are aligned in the instruction aligner **212** and forwarded to the instruction buffer **214** in an align stage **912**, a second stage of the initiating stages. The aligning operation properly positions the instructions for storage in a particular segment of the four register file segments **310**, **312**, **314**, and **316** and for execution in an associated functional unit of the three media functional units **220** and one general functional unit **222**. In a third stage, a decoding stage **914** of the initiating stages, the fetched and aligned VLIW instruction packet is decoded and the scoreboard (not shown) is read and updated in parallel. The four register file segments **310**, **312**, **314**, and **316** each holds either floating-point data or integer data. The register files are read in the decoding (D) stage.

[**0083**] Following the decoding stage **914**, the execution stages are performed. The two terminating stages include a trap-handling stage **960** and a write-back stage **962** during which result data is written-back to the split register file **216**.

[**0084**] While the invention has been described with reference to various embodiments, it will be understood that these embodiments are illustrative and that the scope of the invention is not limited to them. Many variations, modifications, additions and improvements of the embodiments described are possible. For example, those skilled in the art will readily implement the steps necessary to provide the structures and methods disclosed herein, and will understand that the process parameters, materials, and dimensions are given by way of example only and can be varied to achieve the desired structure as well as modifications which are within the scope of the invention. Variations and modifications of the embodiments disclosed herein may be made based on the description set forth herein, without departing from the scope and spirit of the invention as set forth in the following claims.

[**0085**] For example, while the illustrative embodiment specifically discusses advantages gained in using the Java™ programming language with the described system, any suitable programming language is also supported. Other programming languages that support multiple-threading are generally more advantageously used in the described system. Also, while the illustrative embodiment specifically discusses advantages attained in using Java Virtual Machines with the described system, any suitable processing engine is also supported. Other processing engines that support multiple-threading are generally more advantageously used in the described system.

[**0086**] Furthermore, although the illustrative register file has one bit line per port, in other embodiments more bit lines may be allocated for a port. The described word lines and bit lines are formed of a metal. In other examples, other conductive materials such as doped polysilicon may be employed for interconnects. The described register file uses single-ended reads and writes so that a single bit line is employed per bit and per port. In other processors, differential reads and writes with dual-ended sense amplifiers may be used so that two bit lines are allocated per bit and per port, resulting in a bigger pitch. Dual-ended sense amplifiers improve memory fidelity but greatly increase the size of a memory array, imposing a heavy burden on speed performance. Thus the advantages attained by the described register file structure are magnified for a memory using differential reads and writes. The spacing between bit lines and word lines is described to be approximately 1 μm. In some processors, the spacing may be greater than lam. In other processors the spacing between lines is less than 1 μm.

[**0087**] Exemplary Instruction Set Architecture

[**0088**] The material that follows provides a detailed description of an exemplary instruction set suitable for use in a processor architecture such as illustrated in the above-referenced drawings and described elsewhere herein.

[**0089**] Except for symbols reserved for registers, Café assembler symbols are just like those for the SPARC assembler. See the SPARC assembler manual for those details.

[**0090**] The Café assembler uses the .proc pseudo-op similar to SPARC's, but defines no operand for it. This pseudo-op should be used to mark the beginning of a function so the assembler can know to require the beginning of an instruction word. This only makes a practical difference if an immediately preceding function ends with an instruction that does not appear to consummate an instruction word, but using the .proc pseudo-op is a good habit in any case.

[**0091**] General Purpose Registers

[**0092**] Café has 256 32-bit general purpose registers, numbered 0 through 255. The assembler reserves symbols of the form:

[*Rr*]<digit><digit>*

[0093] for general purpose register specifiers. That is, the letter R in either case followed by a non-empty string of decimal digits denotes a general purpose register. Strings of digits indicating values greater than 255 are diagnosed as errors.

[0094] In addition to these canonical register names, a few symbols are reserved for aliases of registers that have special uses.

sp	Stack Pointer, an alias for r1.
1p	Link Pointer. The call instruction puts the return address in 1p, which is an alias for r2.
fp	Frame Pointer, an alias for r3.
gmr	GPU-to-MPU by-pass register, an alias for r6.
mgr	MPU-to-GPU by-pass register, an alias for r7. Others register alias symbols may be defined as register use conventions evolve. None of these symbols is case-sensitive.

[0095] In addition to register symbols, a general purpose register can be denoted by a register expression of the form:

% % [*Rr*]?<constant-expression>

[0096] That is, double percent sign, optionally followed the letter R in either case, followed by the first pass (no forward references, no relocations) constant expression in the range of zero to 255. The optional R is of no value to the the assembler, but some users believe it's useful to see it in the source.

[0097] r0 is a fiat-zero source operand and a result-sink registers.

[0098] Control and Status Registers

[0099] In order to provide an extensible namespace for control and status registers, symbols denoting them begin with %. None of these symbols is case-sensitive.

[0100] Program Counters

[0101] Documentation refers to a program counter, % pc, and its sidekick % npc, but it's not apparent that either is used by the assembler.

[0102] Processor Status Register

[0103] Café has a program status register, for which this assembler uses the symbol % psr. The layout of % psr follows. % psr can be read and modified by the getir and set ir instructions using its internal register ordinal, 1.

[0104] Bit **24** of % psr specifies The processor ID. Clear denotes cpu0, and set denotes cpu1.

[0105] Bits **23** and **22** of % psr specify the current Trap Level.

[0106] Bit **21** of % psr determines the endianness of loads and stores. The initial state of this bit is clear, which means big-endian. When set it means little-endian.

[0107] Bit **20** of % psr is the Instruction Address Check Enable flag. Its description is yet to be supplied.

[0108] Bit **19** of % psr is the Data Address Check Enable flag. Its description is yet to be supplied.

[0109] Bit **18** of % psr is the Garbage Check Enable flag. Its description is yet to be supplied.

[0110] Bit **17** of % psr is the Data Cache Enable flag. When set, the data cache is enabled; when clear, data cache is disabled.

[0111] Bit **16** of % psr is the Instruction Cache Enable flag. When set, the instruction cache is enabled; when clear, instruction cache is disabled.

[0112] Bit **15** of % psr is the Supervisor Mode flag. When set it indicates the processor is in supervisor mode, which allows certain privileged activities. Among these privileged activities is the ability to change all but the two right-most fields of % psr. The Supervisor Mode flag is most often set during trap handling, which is explained in the Traps section of the microarchitecture manual.

[0113] Bit **14** of % psr is the Interrupt Enable flag. When set, interrupts are enabled. Its use is explained in the Traps section of the microarchitecture manual.

[0114] Bits **13** through **10** of % psr is the Processor Interrupt Level, which is explained in the Traps section of the microarchitecture manual.

[0115] The % psr fields that can be set when the Supervisor Mode flag is clear are grouped together at the low-order end. They follow.

[0116] A 2-bit field of % psr specifies the mode in effect for the saturated arithmetic performed by some of the parallel integer operations. The bounds for saturation are given in the adjacent table. Modes 00 and 01 are expressed as two's-complement 16-bit integers. Mode 10 is expressed in S.15 fixed-point. Mode 11 is S2.13 fixed-point. The simulator is using bits **8** and **9** of % psr for this specification.

	bounds	
mode	low	high
00	000000000 . . . 0	011111111 . . . 1
01	100000000 . . . 0	011111111 . . . 1
10	100000000 . . . 0	011111111 . . . 1
11	111000000 . . . 0	001000000 . . . 0

[0117] (For those of us habituated to the common floating-point representation, the Si.*f* notation in the preceding paragraph can be confusing. In these fixed point formats, the sign-bit S is one bit of the integer part of the number. For example, in S2.13 format, the integer part is a two's complement 3-bit number. There is no "dedicated" sign-bit as with the floating-point representation, and thus no negative zero to worry about.)

[0118] The low-order eight bits of % psr are used as dirty bits for octants of the general purpose register file. A new process begins with all the dirty bits clear, and a octant's dirty bit is set when a register in that octant is written. Café's large register file is a formidable lot of state to manage during a context-switch. An isolated region of a long-running program that causes dirty bits to be set should clear them when it's safe to do so. Within this 8-bit field a given register number N corresponds to the bit (1<<(N>>5)).

[0119] Trap Base Register

[0120] A vector of trap handler addresses is pointed-to by a trap base register, for which the assembler uses the symbol

% tbr. Only the high-order 19 bits of % tbr are used to address the vector, so the vector must be positioned at an 8192-byte boundary. Details for use of the vector are described in the "Traps" chapter of the Café Architecture Manual.

[0121] The assembler's only concern is the ability to read or set % tbr using the getir and setir instructions. Reads of the low-order 13 bits of % tbr always return zero, and writes to the low-order 13 bits of % tbr are always ignored.

[0122] No other control or status registers are described yet. Those that will be defined that can be read or set will be accessible using the getir and setir instructions.

[0123] Instruction Set

[0124] Instruction Formats

[0125] Note: Since this document was written the term "SFU" has evolved to "GFU", and the term "UFU" has evolved to "MFU". Until there's no problem more important than changing all occurrences of the old terms here, the author apologizes for the inconvenience.

[0126] Café instructions are issued in instruction words composed of one SFU instruction and zero to three UFU instructions. An SFU instruction begins with a 2-bit header field that is a count of the UFU instructions that follow in the instruction word. All of the instructions in an instruction word are issued in the same cycle.

[0127] When there isn't useful work to do on all the UFUs, UFU instructions need not be present. However, the UFU on which an instruction executes is determined by the position of the instruction in the instruction word. To cause an instruction to execute on the second or third UFU, there must have been instructions in the previous slots of the instruction word. This is an issue when trying to avoid the latency of propagating a result from one FU to another.

[0128] The assembler infers the beginning of an instruction word from the presence of an SFU instruction. UFU instructions that follow form the rest of the instruction word. More than three consecutive UFU instructions are reported as a fatal error, since the assembler cannot create a well-formed Café instruction word from that.

[0129] Several mnemonics denote instructions implemented both as SFU and UFU operations. These mnemonics indicate an SFU instruction only when used at the beginning of an instruction word. An instruction word boundary is established when the immediately preceding instruction word uses all three of its UFU slots or by the presence of two adjacent semicolons (;;), the instruction word delimiter.

[0130] Algebraicly, if not lexically, the double semicolon is a full colon, meaning it's time to flush the instruction word. For example:

[0131] ;; add r6,1,r6; add r7,1,r7;; is a single instruction word beginning with an SFU add operation and having a single UFU operation, also an add. But the similar pattern:

[0132] ;; add r6,1,r6;; add r7,1,r7;; is to instruction words, each consisting of an SFU add operation.

[0133] SFU Instruction Formats

[0134] An SFU instruction begins with a 2-bit header (labeled hdr in the instruction format diagrams appearing later in this section) that gives the number of UFU instructions that follow the SFU instruction in the instruction word. That is, the header vaules and instruction word contents they indicate are:

header value	instructions in instruction word
00	SFU only
01	SFU + UFU1
10	SFU + UFU1 + UFU2
11	SFU + UFU1 + UFU2 + UFU3

[0135] The first two bits of an SFU opcode determine the class of the operation. The values and classes are:

00	Call and branch
01	Compute
10	Memory (uncacheable)
11	Memory (cacheable)

[0136] Generally, the third bit of an SFU opcode, the i-bit, is set when an operation uses an immediate for its second source operand and clear when it does not. SFU opcodes beginning with 00 (call and branch) are 6 bits, and all others are 8 bits.

[0137] Opcodes for the memory operations can be shown in a matrix where the bits usually indicate cacheability, signedness, size, and direction:

Memory (cacheable, leading 11) Opcodes

	opcode[2.0]							
	0xx-(unsigned)				1xx-(signed)			
opcode[7:3] 11ixx	byte 000	short 001	word 010	long 011	byte 100	short 101	word 110	long 111
11i00	ldub	ldus	lduw	ldpair	ldb	lds	ldw	(ldg)
11i01	—	lduso	lduwo	ld_diag	—	ldso	ldwo	prefetch
11i10	stb	sts	stw	stpair	cstb	csts	cstw	—
11i11	s2ib	stso	stwo	st_diag	—	—	cas	—

[0138]

Memory (uncacheable, leading 10) Opcodes								
	opcode[2.0]							
	0xx-(unsigned)				1xx-(signed)			
opcode [7:3] 10ixx	byte 000	short 001	word 010	long 011	byte 100	short 101	word 110	long 111
10i00	ncldub	ncldus	nclduw	ncldg	ncldb	nclds	ncldw	—
10i01	—	nclduso	nclduwo	—	—	ncldso	ncldwo	—
10i10	ncstb	nests	ncstw	ncstpair	—	—	—	—
10i11	—	ncstso	ncstwo	—	—	—	—	—

[0139] Opcodes in the compute (leading 01) quadrant of the SFU opcode space generally are not assigned in ways where the bit patterns reveal much other than where the i-bit is used. Mnemonics in both the upper and lower halves of this table are those for opcodes that are the same except for a clear or set i-bit. Note that there are only three free spaces in the upper half and sixteen free in the upper half.

Compute (leading 01) Opcodes									
opcode[7:3]		opcode[2:0]							
	01ixx	000	001	010	011	100	101	110	111
i = 0	01000	add	—	sub	—	not	or	and	xor
	01001	idiv	rem	ppower	cmovenz	sethi	cmovez	blockaddr	—
	01010	shll	shrl	shra	—	cmpeq	cmplt	frecsqrt	—
	01011	fcmpeq	fcmplt	fcmple	cmpult	fdiv	precsqrt	getir	setir
i = 1	01100	add	—	sub	—	—	or	add	xor
	01101	idiv	rem	—	cmovenz	—	cmovez	—	—
	01110	shll	shrl	shra	—	cmpeq	cmplt	—	—
	01111	—	—	—	cmpult	—	—	—	—

[0140] Opcodes in the call and branch (leading 00) quadrant of the SFU opcode space have some irregularities compared to other SFU opcodes. Since call and nop opcodes must be unique in their higher-order six bits, they have effective footprints of four opcodes each. Similarly, bz and bnz, with their prediction qualifiers, each use up four opcode slots. This quadrant does not use the i-bit as the other three do.

Call and branch (leading 00) Opcodes								
opcode[7:3]	opcode[2:0]							
00xxx	000	001	010	011	100	101	110	111
00000	call	n/a	n/a	n/a	bz	bz, pt	bz, ph	bz, ph, pt
00001	bnz	bnz, pt	bnz, ph	bnz, ph, pt	jmpl	done	retry	sir
00010	softtrap	iflush	—	—	—	—	—	—
00011	—	—	—	—	—	—	—	—
00100	nop	nop	nop	nop	—	—	—	—
00101	—	—	—	—	—	—	—	—
00110	softtrap	membar	—	—	—	—	—	—
00111	—	—	—	—	—	—	—	—

[0141] The SFU instruction formats are:

[0142] memory and compute

[0143] (See **FIG. 10A**)

[0144] sethi

[0145] (See **FIG. 10B**)

[0146] Only the sethi instruction uses this format.

[0147] call

[0148] (See **FIG. 10C**)

[0149] Only the sethi instruction uses this format.

[0150] (See **FIG. 10D**)

[0151] Only the bz and bnz instructions use this format.

[0152] UFU Instruction Formats

[0153] three-source operand

[0154] (See **FIG. 10E**)

[0155] two-source operand

[0156] (See **FIG. 10F**)

[0157] The immediate field of the UFU two-source instruction is 14 bit to be consistent with very similar operations using the SFU compute format.

[0158] Instruction Categories

Integer Instructions

mnemonic	argument list	opcode	L	operation
add	rs1,reg_or_imm14,rd	S-01i00000 U-000000	1	Add
addc	rs1,rs2,rs3,rd	U-000001	1	Add with carry
cmovenz	rs1,reg_or_imm14,rd rs1,reg_or_imm8,rd	S-01i01011 U-pseudo-op	1	Move if not zero
cmovez	rs1,reg_or_imm14,rd rs1,reg_or_imm8,rd	S-01i01101 U-pseudo-op	1	Move if zero
cmpeq	rs1,reg_or_imm14,rd	S-01i10100 U-101100	1	Compare equals
cmplt	rs1,reg_or_imm14,rd	S-01i10101 U-101110	1	Compare less than
cpicknz	rs1,reg_or_imm8,reg_or_imm8,rd	U-101010	1	Conditionally (non-zero) pick
cpickz	rs1,reg_or_imm8,reg_or_imm8,rd	U-101011	1	Conditionally (zero) pick
cmpult	rs1,reg_or_imm14,rd	S-01i11011 U-101111	1	Compare less than
genborrow	rs1,reg_or_imm14,rd	U-101000	1	Generate borrow
gencarry	rs1,reg_or_imm14,rd	U-101001	1	Generate carry
getir	rs1,rd	S-01011110	?	Get internal register
idiv	rs1,reg_or_imm14,rd	S-01i01000	?	Division
moveind	rs1,[rs2]	U-100010	2	Move indirect
mul	rs1,reg_or-imm14,rd	U-000110	2	Multiply
muladd	rs1,reg_or_imm8,reg_or_imm8,rd	U-001000	2	Multiply add
mulsub	rs1,reg_or_imm8,reg_or_imm8,rd	U-001001	2	Multiply add
padd	rs1,reg_or_imm14,rd	U-000100	1	Parallel add
padds	rs1,rs2,rd	U-110100	1	Saturating parallel add
pcmovenz	rs1,rs2,rd	U-111100	1	Parallel move if not zero
pcmovez	rs1,rs2,rd	U-111101	1	Parallel move if zero
pcmpeq	rs1,reg_or_imm14,rd	U-110001	1	Parallel compare equal
pcmplt	rs1,reg_or_imm14,rd	U-110011	1	Parallel compare less than
pmul	rs1,reg_or_imm14,rd	U-000111	2	Parallel multiply
pmuladd	rs1,rs2,rs3,rd	U-001010	2	Parallel multiply add
pmuladds	rs1,rs2,rs3,rd	U-111010	2	Saturating parallel multiply add
pmulsub	rs1,rs2,rs3,rd	U-001011	2	Parallel multiply subtract
pmulsubs	rs1,rs2,rs3,rd	U-111011	2	Saturating parallel multiply subtract
psub	rs1,reg_or_imm14,rd	U-000101	1	Parallel subtract
psubs	rs1,rs2,rd	U-110101	1	Saturating parallel subtract
rem	rs1,reg_or_imm14,rd	S-01i01001	?	Remainder
sethi	imm22,rd	S-01001100	1	Sethi
setir	rs1,rd	S-01011111	1	Set internal register
sub	rs1,reg_or_imm14,rd	S-01i00010 U-000010	1	Subtract
subc	rs1,rs2,rs3,rd	U-000011	1	Subtract with carry

[0159]

Logical Instructions

mnemonic	argument list	opcode	L	operation
add	rs1,reg_or_imm14,rd	S-01i00110 U-010110	1	Add
cccb	rs1,reg_or_imm14,rd	U-111110	1	Count consecutive clear bits
not	rs1,rd	S-01i00100 U-010100	1	Not
or	rs1,reg_or_imm14,rd	S-01i00101 U-010101	1	Or
pshll	rs1,reg_or_imm14,rd	U-011100	1	Parallel shift left logical
pshra	rs1,reg_or_imm14,rd	U-011110	1	Parallel shift right arithmetic
pshrl	rs1,reg_or_imm14,rd	U-011101	1	Parallel shift right logical
shll	rs1,reg_or_imm14,rd	S-01i10000 U-011000	1	Shift left logical
shra	rs1,reg_or_imm14,rd	S-01i10010 U-011010	1	Shift right arithmetic
shrl	rs1,reg_or_imm14,rd	S-01i10001 U-011001	1	Shift right logical
xor	rs1,reg_or_imm14,rd	S-01i00111 U-010111	1	Exclusive or

[0160]

Floating Point Instructions

mnemonic	argument list	opcode	L	operation
clip	rs1,rs2,rs3,rd	U-011111	1	Clip
fadd	rs1,rs2,rd	U-001100	4	Single precision addition
fcmpeq	rs1,rs2,rd	S-01011000	1	FP compare equals
fcmple	rs1,rs2,rd	S-01011010	1	FP compare less than or equals
fcmplt	rs1,rs2,rd	S-01011001	1	FP compare less than
fdiv	rs1,rs2,rd	S-01011100	6	Single precision division

-continued

Floating Point Instructions

mnemonic	argument list	opcode	L	operation
fmul	rs1,rs2,rd	U-001110	4	Single precision multiplication
fmuladd	rs1,rs2,rs3,rd	U-010000	4	Single-precision multiply-add
fmulsub	rs1,rs2,rs3,rd	U-010001	4	Single precision multiply-subtract
frecsqrt	rs1,rd	S-01010110	6	Single precision reciprocal square root
fsub	rs1,rs2,rd	U-001101	4	Single precision subtraction

[0161] Fixed-Point Instructions

[0162] The fixed-point operands of these instructions are in S2.13 format; that is, these instructions are unaffected by the fixed-point mode bits of the psr. Precision may be lost as a result is rendered in that format. Overflows saturate.

mnemonic	argument list	opcode	L	operation
ppower	rs1,rs2,rd	S-01i01010	6	Parallel exponentiation
precsqrt	rs1,rd	S-01011101	6	Parallel reciprocal square root

[0163]

Convert Instructions

mnemonic	argument list	opcode	L	operation
fix2flt	rs1,reg_or_imm14,rd	U-100111	4	Fixed point to single precision
flt2fix	rs1,reg_or_imm14,rd	U-100110	4	Single precision to fixed point

[0164]

Control Flow Instructions

mnemonic	argument list	opcode	L	operation
bndck	rs1,reg_or_imm8,reg_or_imm8	U-011011	?	Bound check
bnz	rd,label	S-000010ht	1	Branch if not zero
bz	rd,label	S-000001ht	1	Branch if zero
call	label	S-000000	1	Call
done	no arguments	S-00001101	?	Skip trapped instruction
jmpl	rs1,rd	S-00001100	2	Jump and link
nop	no arguments	S-001000xx	1	Null operation
retry	no arguments	S-00001110	?	Retry trapped instruction
return	rs1	pseudo-op	2	Return
sir	no arguments	S-00001111	?	Software-initiated reset
softtrap	rs1,reg_or_imm14	S-00i10000	?	Software-initiated trap

[0165] Memory Access Instructions

[0166] Café memory accesses can be done either big- or little-endian, with big-endian being the default. Endianness is controlled by a bit in % psr.

[0167] Instructions are stored big-endian. The assembler and linker assume that relocations and other initializations in sections other than code sections should also be treated as big-endian.

[0168] Most memory access instructions use a two-component effective address specification, denoted in this document by [address]. [address] may be [rs1+rs2], [rs1+simm14], or [rs1]. When [rs1] is specified, the assembler infers an immediate zero for the second address component of the instruction.

[0169]

Pixel Instructions

mnemonic	argument list	opcode	L	operation
bitext	rs1,rs2,rs3,rd	U-111111	2	Bit extract
byteshuffle	rs1,rs2,rs3,rd	U-100001	2	Byte extract
pack	rs1,rs2,rs3,rd	U-100000	1	Pack
pdist	rs1,rs2,rd	U-100101	4	Pixel distance
pmean	rs1,rs2,rd	U-100100	1	Parallel mean

mnemonic	argument list	opcode	L	operation
cas	rs1,{rs2],rd	S-11011110	?	Compare and swap(atomic)
cstb	rd,rs1,[rs2]	S-11010100	?	Conditional store byte
csts	rd,rs1,[rs2]	S-11010101	?	Conditional store short
cstw	rd,rs1,[rs2]	S-11010110	?	Conditional store word
iflush	no arguments	S-00010001	?	Flush instruction pipe
ldb	[address],rd	S-11i00100	2	Load byte
ldpair	[address],rd	S-11i00011	?	Load pair
lds	[address],rd	S-11i00101	2	Load short
ldso	[address],rd	S-11i01101	2	Load short other-endian
ldub	[address],rd	S-11i00000	2	Load unsigned byte
ldus	[address],rd	S-11i00001	2	Load unsigned short
lduso	[address],rd	S-11i01001	2	Load unsigned short other-endian
lduw	[address],rd	S-11i00010	2	Load unsigned word
lduwo	[address],rd	S-11i01010	2	Load unsigned word other-endian
ldw	[address],rd	S-11i00110	2	Load word
ldwo	[address],rd	S-11i01110	2	Load word other-endian
membar	no arguments	S-00110001	?	Memory barrier
ncldb	[address],rd	S-10i00100	?	Non-cacheable load byte
ncldg	[address],rd	S-10i00011	?	Non-cacheable load group
nclds	[address],rd	S-10i00101	?	Non-cacheable load short
ncldso	[address],rd	S-10i01101	?	Non-cacheable load short other-endian
ncldub	[address],rd	S-10i00000	?	Non-cacheable load unsigned byte
ncldus	[address],rd	S-10i00001	?	Non-cacheable load unsigned short
nclduso	[address],rd	S-10i01001	?	Non-cacheable load unsigned short other-endian
ncldуw	[address],rd	S-10i00010	?	Non-cacheable load unsigned word
ncldуwo	[address],rd	S-10i01010	?	Non-cacheable load unsigned word other-endian
ncldw	[address],rd	S-10i00110	?	Non-cacheable load word
ncldwo	[address],rd	S-10i01110	?	Non-cacheable load word other-endian
ncstb	rd,[address]	S-10i10000	?	Non-cacheable store byte
ncstpair	rd,[address]	S-10i10011	?	Non-cacheable store pair
nCsts	rd,[address]	S-10i10001	?	Non-cacheable store short
ncstso	rd,[address]	S-10i11001	?	Non-cacheable store short other-endian
ncstw	rd,[address]	S-10i10010	?	Non-cacheable store word
ncstwo	rd,[address]	S-10i11010	?	Non-cacheable store word other-endian
s2ib	rd,[address]	S-11i11000	2	Store byte to instruction
stb	rd,[address]	S-11i10000	1	Store byte
stpair	rd,[address]	S-11i10011	?	Store pair
sts	rd,[address]	S-01i10001	1	Store short
stso	rd,[address]	S-01i11001	1	Store short other-endian
stw	rd,[address]	S-10i10010	1	Store word
stwo	rd,[address]	S-10i11010	1	Store word other-endian

[still missing pefetch, ld_diag and st_diag, at least]

[0170] more as I learn more . . .

[0171] Scheduling

[0172] Note: This section is superseded by the "Code Scheduling Guidlines" section of the Café Microprocessor Architecture Manual. Please feel free to recommend improvements or corrections to this material, but refer to the manual for the definitive treatment.

[0173] None of the results of UFU operations is scoreboarded. All loads and a few non-pipelined long-latency SFU operations such as idiv, fdiv, frecsqrt, precsqrt, and ppower (the complete list is not determined) are scoreboarded.

[0174] Non-scoreboarded results are available to subsequent operations on the unit that produces them after their latencies; earlier use is erroneous. Latencies are shown in the columns labeled "L" in the tables in the preceding section.

[0175] A result produced on one unit is available as an operand on another unit when it is being written to the register file (that is, when it reaches pipe stage **W1**). Earlier use is erroneous. This takes 5 cycles on a UFU. On the SFU this takes one cycle more than the latency of the producing operation. The difference is because UFU results have to pass through all 4 E-stages of a pipe before reaching **W1**, and SFU results do not.

[0176] Use of a scoreboarded result register as an operand causes instruction issue to stall for as many cycles as it takes for that result to become available. If the referencing instruction that provokes the stall is also on the SFU, the stall is only until the result is available for intra-unit bypass. In the case of a load that hits in the cache, the stall could be as short as a single cycle. If the referencing instruction is on a UFU, stall lasts until the result reaches stage **W1**, where it can be bypassed on its way to the register file.

[0177] To help improve the latency of feeding operations not available on all units, special bypass registers are available. A completed (that is, finished but not yet in **W1**) instruction's result can be bypassed from the first UFU to the SFU if its destination register is r4. Likewise, a completed instruction's result can be bypassed from the SFU to the the first UFU if its destination register is r5.

[0178] Multiple writes to the same register of the register file in the same same cycle is an erroneous condition with an undefined result. More generally, any time a given register is the destination of more than operation in progress, the program is erroneous. Suppose that unit's 4-E-stage pipe sees a sequence of operations like:

cycle	operation
0	4-cycle op → r10
1	4-cycle op → r10
2	anything
3	anything
4	any op using cycle zero's r10 as a source
5	any op using cycle one's r10 as a source

[0179] Before being too awestruck by this tight scheduling, consider what happens if issue has to stall, say for icache fill, in cycle two or three: both uses of r10 will get the result produced by the op that started in cycle one. Never issue a redefinition of a register's value between a definition and a use. The value that will reach the use is not deterministic.

[0180] Instruction Details

[0181] add

[0182] add is an integer instruction that computers "r[rs1]+r[rs2]" or "r[rs1]+sign_ext(imm14) ". The use of an immediate for the second source operand sets the i-bit of the opcode of the SFU version or the first header bit of the UFU version. The resulting sum is left in r[rd].

[0183] The suggested assembler syntax is:

[0184] add rs1, reg_or_imm14, rd

[0185] The SFU version of this instruction uses the SFU compute format, and the UFU version uses the UFU 2-source format.

[0186] addc, subc

[0187] addc and subc are integer instructions that compute "r[rs1]+r[rs2]+r[rs3]" or "r[rs1]−r[rs2]−r[rs3]", respectively. Only the least significant bit of r[rs3], which is expected to be the result of a gencarry or genborrow, is used. The result is left in r[rd].

[0188] The suggested assembler syntax is:

[0189] addc rs1, rs2, rs3, rd

[0190] subc rs1, rs2, rs3, rd

[0191] addc and subc are UFU instructions that use the UFU 3-source format.

[0192] and

[0193] and is a logical instruction that computes "r[rs1] & r[rs2]" or "r[rs1] & imm14". The use of an immediate for the second source operand sets the i-bit of the opcode of the SFU version or the first header bit of the UFU version. The result is left in r[rd].

[0194] The suggested assembler syntax is:

[0195] and rs1, reg_or_imm14, rd

[0196] The SFU version of and uses the SFU compute format, and the UFU version uses the UFU 2-source format.

[0197] Bitext

[0198] bitext is a pixel instruction that extracts bits from the pair of registers r[rs1] and r[rs2]. The extracted field is described by a 6-bit length in bits **21** . . . **16** of r[rs3]and a 5-bit skip count in bits **4** . . . **0** of r[rs3]. The skip count is applied at the left-most (high-order) end of r[rs1]. The extracted field is right justified with r[rd] without sign-extension.

[0199] (The illustration below is appropriated from a someone else's slide. I hope the stylistic change is not too jarring, but prose description has not been lucid for some readers.)

[0200] See **FIG. 11**

[0201] The suggested assembler syntax is:

[0202] bitext rs1, rs2, rs3, rd

[0203] bitext is an UFU operation that uses the UFU 3-source format.

[0204] bndck

[0205] bndck is a control flow instruction that causes a trap if "r[rs1]==0", the second source operand is less than zero, of if the second source operand is greater than or equal to the third source operand. What it means to trap is not yet defined.

[0206] The second source operand can be either r[rs2] or sign_ext(imm8), and the third source operand can be either r[rs3] or sign_ext(imm8). The use of an immediate for either causes the appropriate header bit to be set in the instruction.

[0207] The suggested assembler syntax is:

[0208] bndck rs1, reg_or_imm8, reg_or_imm8 bndck is a UFU operation that uses the UFU 3-source format. The operation has no use for this format's r[rd], which the assembler will make appear to be zero.

[0209] bnz

[0210] bnz is a control flow instruction for a branch to the offset implied by the difference between the program counter and the specified label if the value in "r[rd]" is not equal to the integer zero. "label" is a label at the branch target; the assembler will either determine the displacement or generate relocation information so a linker can determine it. Whenever the displacement is determined, it must be expressible in a signed 22-bit field.

[0211] The mnemonic may be followed optionally by the qualifier,pt, which means this conditional branch is staticly predicted to be taken. The use of this qualifier sets the T-bit in the instruction.

[0212] The suggested assembler syntax is:

[0213] bnz rd, label

[0214] bnz, pt rd, label

[0215] bnz is an SFU operation that uses the branch instruction format.

[0216] byteshuffle

[0217] byteshuffle is a pixel instruction that copies the bytes from its sources r[rs1] and r[rs2] to byte positions of r[rd] according to the pattern described by the bits of the least significant two bytes of r[rs3].

[0218] Each group of four contiguous bits the lower-order two bytes of r[rs3] is the ordinal of the byte position of the eight bytes of the register pair r[rs1]–r[rs2] from which a byte is copied to the corresponding byte of r[rd]. An out-of-range byte ordinal (that is, a value greater than 7) means the corresponding byte of r[rd] will be zeroed.

[0219] The suggested assembler syntax is:

[0220] byteshuffle rs1, rs2, rs3, rd

[0221] byteshuffle is a UFU operation that uses the UFU 3-source format.

[0222] bz

[0223] bz is a control flow instruction for a branch to the offset implied by the difference between the program counter and the specified label if the value in "r[rd]" is equal to the integer zero. "label" is a label at the branch target; the assembler will either determine the displacement or generate relocation information so a linker can determine it. Whenever the displacement is determined, it must be expressible in a signed 22-bit field.

[0224] The mnemonic may be followed optionally by the qualifier, pt, which means this conditional branch is staticly predicted to be taken. The use of this qualifier sets the T-bit in the instruction.

[0225] Unconditional branches are commonly coded using bz with r0 for the register operand. When the assembler sees this "unconditional conditional" branch without prediction, it will infer the ",pt" qualification, which can improve instruction prefetching.

[0226] The suggested assembler syntax is:

[0227] bz rd, label

[0228] bz, pt rd, label

[0229] bz is an SFU operation that uses the branch instruction format.

[0230] call

[0231] call is a control flow instruction causes a control transfer to the address specified by its label operand. A call to address zero is an illegal instruction.

[0232] The return address (% npc at the time of the call) is left in r2, an implicit operand of this instruction. For that reason the assembler has the alias lp ("link pointer") for r2.

[0233] The suggested assembler syntax is:

[0234] call label

[0235] call is an SFU operation that uses a format of its own.

[0236] cas

[0237] cas is a memory access instruction that compare the content of register r[rs1] with the content of the 32-bit word in memory addressed by r[rs2]. If those values are equal, the content of register r[rd] is swapped with the word addressed by r[rs2]. Otherwise, the content of the addressed memory word is unchanged, but the value at that memory address replaces the content of register r[rd].

[0238] The effective address from which to load must be word-aligned, but the consequences of failing to do that are not defined.

[0239] Note that cas uses dcache, which makes it unsuitable for thread synchronization in a multi-Café configuration.

[0240] The suggested assembler syntax is:

[0241] cas rs1, [rs2], rd

[0242] cas is an SFU operation that use the 2-source register variant of the SFU memory format. Since it always uses two source registers, the i-bit of its opcodes is always clear.

[0243] cccb

[0244] cccb is a logical instruction that counts consecutive clear bits is its first source operand, r[rs1], beginning from the high-order bit, first skipping the number of bits specified by the second source operand. The second source operand my be either a register or an immediate. In either case, only the the low-order 5 bits of the skip-count are used. The count of clear bits is left in r[rd].

[0245] The suggested assembler syntax is:

[0246] cccb rs1, reg_or_imm14, rd

[0247] cccb is an UFU operation that uses the UFU 2-source format.

[0248] clip

[0249] clip is a floating point instruction that computes:

[0250] ((r[rs1] > r[rs2]? 1:0) <<1) |

[0251] (r[rs1] < −r[rs2]? 1:0) |

[0252] (r[rs3] << 2)

[0253] and leaves the result in r[rd]. The practical effect of this operation is to return a copy of r[rs3] shifted left 2 bits with the two least significant bits occupied by indications of how the single-precision floating point numbers in r[rs1] and r[rs2] compare.

[0254] The suggested assembler syntax is:

[0255] clip rs1, rs2, rs3, rd

[0256] clip is a UFU operation that uses the UFU 3-source format, but the variants of that format that allow immediates for the second and third source operand are not used.

[0257] cmovenz

[0258] cmovenz is an integer instruction that copies the value of the second source operand, specified by "r[rs2]" or "sign_ext(imm)", to the result register r[rd] only if the first source operand, r[rs1], is non-zero. Note that cmovenz allows a 14-bit immediate on the SFU but only an 8-bit immediate on a UFU.

[0259] The suggested assembler syntax is:

cmovenz	rs1,reg_or_imm14,rd	!	SFU
cmovenz	rs1,reg_or_imm8,rd	!	UFU

[0260] The SFU version of the cmovenz uses the SFU compute format. The UFU version of cmovenz is a pseudo-op for the cpicknz instruction with r[rd] replicated in the r[rs3] field.

[0261] cmovez

[0262] cmove z is an integer instruction that copies the value of the second source operand, specified by "r[rs2]" or "sign_ext(imm)", to the result register r[rd] only if the first source operand, r[rs1], is zero. Note that cmovez allows a 14-bit immediate on the SFU but only an 8-bit immediate on a UFU.

[0263] The suggested assembler syntax is:

cmovez	rs1,reg_or_imm14,rd	!	SFU
cmovez	rs1,reg_or_imm8,rd	!	UFU

[0264] The SFU version of cmovez uses the SFU compute format. The UFU version of cmovez is a pseudo-op for the cpickz instruction with r[rd] replicated in the r[rs3] field.

[0265] cmpeq

[0266] cmpeq is an integer instruction that computes "r[rs1]==r[rs2]" or "r[rs1]=sign_ext(imm14)". The use of an immediate for the second source operand sets the i-bit of the opcode. The resulting zero or one is left in r[rd].

[0267] The suggested assembler syntax is:

cmpeq	rs1,reg_or_imm14,rd

[0268] The SFU version of this instruction uses the SFU compute format, and the UFU version uses the UFU 2-source format.

[0269] cmplt

[0270] cmplt is an integer instruction that computes "r[rs1] <r[rs2]" or "r[rs1]<sign_ext(imm14)". The use of an immediate for the second source operand sets the i-bit of the opcode. The resulting zero or one is left in r[rd].

[0271] The suggested assembler syntax is:

cmplt	rs1,reg_or_imm14,rd

[0272] The SFU version of this instruction uses the SFU compute format, and the UFU version uses the UFU 2-source format.

[0273] cmpult

[0274] cmpult is an integer instruction that computes "(*unsigned*) r[rs1]<(unsigned) r[rs2]" or "(unsigned) r[rs1] <(unsigned) imml 4". The use of an immediate for the second source operand sets the i-bit of the opcode. The resulting zero or one is left in r[rd].

[0275] The suggested assembler syntax is:

cmpult	rs1,reg_or_imml4,rd

[0276] The SFU version of this instruction uses the SFU compute format, and the UFU version uses the UFU 2-source format.

[0277] cpicknz

[0278] cpicknz is an integer instruction that assigns its second source operand to r[rd] if the first source operand, r[rs1], is non-zero; otherwise, the third source operand is assigned to r[rd]. Each of the second and third source operands can be either a register or a sign-extended 8-bit immediate.

[0279] The suggested assembler syntax is:

cpicknz	rs1,reg_or_imm8,reg_or_imm8,rd

[0280] cpicknz is a UFU operation that uses the UFU 3-source format.

[0281] cpickz

[0282] cpickz is an integer instruction that assigns its second source operand to r[rd] if the first source operand, r[rs1], is zero; otherwise, the third source operand is

assigned to r[rd]. Each of the second and third source operands can be either a register or a sign-extended 8-bit immediate.

[0283] The suggested assembler syntax is:

cpickz	rs1,reg_or_imm8,reg_or_imm8,rd

[0284] cpickz is a UFU operation that uses the UFU 3-source format.

[0285] cstb, csts, cstw

[0286] cstb, csts, and cstw are memory access instructions that, if the value in the register r[rs1] is non-zero, store the value the register r[rd] at the address in register r[rs2].

[0287] The effective address to which to store must be aligned to a natural boundary for the size of the store, but the consequences of failing to do that are not defined.

[0288] The suggested assembler syntax is:

cstb	rd rs1,[rs2]
csts	rd,rs1,[rs2]
cstw	rd,rs1,[rs2]

[0289] cst[b, s, w]are SFU operations that use the 2-source register variant of the SFU memory format. Since they always use two source registers, the i-bit of their opcodes is always clear.

[0290] done

[0291] done is a control flow instruction causes a control transfer from a trap handler to the next instruction word after the instruction that caused the trap. Please refer to the Traps chapter of the Café Microarchitecture specification for the complete description.

[0292] The suggested assembler syntax is:

[0293] done

[0294] done is a SFU operation that uses the SFU compute format, but is has no use for any operand.

[0295] fadd

[0296] fadd is a floating point instruction that computes "r[rs1]+r[rs2]", where the values of the source operands are IEEE single-precision floating point numbers. The result is delivered in r[rd].

[0297] The suggested assembler syntax is:

fadd	rs1,rs2,rd

[0298] fadd is a UFU operation that uses the UFU 2-source format.

[0299] fcmpeq

[0300] fcmpeq is a floating point instruction that compares the single-precision floating point operands in its source registers r[rs1] and r[rs2] for equality. The destination register r[rd] set set to the integer value 1 if the source operands are equal and zero otherwise. If either source operand is a NaN, they are not equal.

[0301] The suggested assembler syntax is:

fcmpeq	rs1,rs2,rd

[0302] fcmpeq is a SFU operation that uses the SFU compute format.

[0303] fcmple

[0304] fcmple is a floating point instruction that set its destination register r[rd] to the integer value 1 if the single-precision floating point value in r[rs1] is less than or equal to the single-precision floating point value in r[rs2] and to zero otherwise. If the value of either source operand is a NaN, the result is zero.

[0305] The suggested assembler syntax is:

fcmple	rs1,rs2,rd

[0306] fcmple is a SFU operation that uses the SFU compute format.

[0307] fcmplt

[0308] fcmpit is a floating point instruction that set its destination register r[rd] to the integer value 1 if the single-precision floating point value in r[rs1] is less than the single-precision floating point value in r[rs2] and to zero otherwise. If the value of either source operand is a NaN, the result is zero.

[0309] The suggested assembler syntax is:

fcmplt	rs1,rs2,rd

[0310] fcmpit is a SFU operation that uses the SFU compute format.

[0311] fdiv

[0312] fdiv is a floating point instruction that computes "r[rs1]|r[rs2]", where the value of the source operands are IEEE single-precision floating point numbers. The result is delivered in r[rd].

[0313] The suggested assembler syntax is:

fdiv	rs1,rs2,rd

[0314] fdiv is a SFU operation that uses the SFU compute format.

[0315] fix2flt

[0316] fix2 flt is a convert instruction that converts a fixed point value in r[rs1], with its binary point specified by the low-order 5 bits of r[rs2] or imm14, to a single precision floating point result in r[rd].

[0317] The suggested assembler syntax is:

fix2flt	rs1,reg_or_imm14,rd

[0318] fix2flt is an UFU operation that uses the UFU 2-source format.

[0319] flt2fix

[0320] flt2fix is a convert instruction that converts a single precision floating point value in r[rs1] to a fixed point result in r[rd] with the binary point as specified by the low-order 5 bits of r[rs2] or imm14.

[0321] The suggested assembler syntax is:

flt2fix	rs1,reg_or_imm14,rd

[0322] flt2fix is an UFU operation that uses the UFU 2-source format.

[0323] fmul

[0324] fmul is a floating point instruction that computes "r[rs1]*r[rs2]", where the values of the source operands are IEEE single-precision floating point numbers. The result is delivered in r[rd].

[0325] The suggested assembler syntax is:

fmul	rs1,rs2,rd

[0326] fmul is a UFU operation that uses the UFU 2-source format.

[0327] fmuladd

[0328] fmuladd is a floating point instruction that computes "(r[rs1]*r[rs2])+r[rs3]", where the values of the source operands are IEEE single-precision floating point numbers. The result is delivered in r[rd].

[0329] The suggested assembler syntax is:

fmuladd	rs1,rs2,rs3,rd

[0330] fmuladd is a UFU operation that uses the UFU 3-source format.

[0331] fmulsub

[0332] fmul sub is a floating point instruction that computes "(r[rs1]*r[rs2])−r[rs3]", where the values of the source operands are IEEE single-precision floating point numbers. The result is delivered in r[rd].

[0333] The suggested assembler syntax is:

fmulsub	rs1,rs2,rs3,rd

[0334] fmulsub is a UFU operation that uses the UFU 3-source format.

[0335] frecsqrt

[0336] frecsqrt is a floating point instruction that computes the reciprocal square root of the single-precision floating-point number in r[rs1] and puts that result in r[rd]. What will this do with an argument less than or equal to zero?

[0337] The suggested assembler syntax is:

frecsqrt	rs1,rd

[0338] frecsqrt is an SFU operation uses the SFU compute format, but has no use for the second source operand of that format.

[0339] fsub

[0340] fsub is a floating point instruction that computes "r[rs1]−r[rs2]", where the values of the source operands are IEEE single-precision floating point numbers. The result is delivered in r[rd].

[0341] The suggested assembler syntax is:

fsub	rs1,rs2,rd

[0342] fsub is a UFU operation that uses the UFU 2-source format.

[0343] genborrow, gencarry

[0344] genborrow and gencarry are integer instructions that generate a one or zero in r[rd] if subtracting or adding, respectively, the source operands generates a borrow or carry, respectively. The result would be useful as the third source operand of sesequent addc and subc operations.

[0345] The suggested assembler syntax is:

genborrow	rs1,reg_or_imm14,rd
gencarry	rs1,reg_or_imm14,rd

[0346] genborrow and gencarry are are UFU operations that use the UFU 2-source format.

[0347] getir

[0348] getir is an integer instruction that gets the value of the internal register the ordinal of which is its r[rs1] operand and puts that value in the register specified by r[rd].

[0349] The suggested assembler syntax is:

getir	rs1,rd

[0350] getir is an SFU operation that uses the SFU compute format, though in an irregular way. It has no use for the second source field, and the first source operand is an internal register number, NOT one of the general purpose registers.

[0351] idiv

[0352] idiv is an integer instruction that computes "r[rs1]|r[rs2]" or "r[rs1]‖sign_ext(imm14)". The use of an immediate for the second source operand sets the i-bit of the opcode. The result is left in r[rd].

[0353] If the second source operand is zero, idiv will trap.

[0354] The suggested assembler syntax is:

idiv	rs1,reg_or_imm14,rd

[0355] idiv is an SFU operation that uses the SFU compute format.

[0356] iflush

[0357] iflush is a memory access instruction that is used to make sure that modifications to code space are visible by the processor executing the iflush iflush invalidates all younger instructions that have already entered the pipe.

[0358] The suggested assembler syntax is:

[0359] iflush

[0360] iflush is an SFU operation that uses the compute instruction format, but uses none of that format's operands.

[0361] jmpl

[0362] jmpl is a control flow instruction that causes a register-indirect control transfer to the address in r[rs1]. The current value of % npc is left in r[rd].

[0363] If the branch target is an entry-point that might also expect to be reached by a call instruction, a register other than lp is a poor choice for r[rd].

[0364] The suggested assembler syntax is:

jmpl	rs1,rd

[0365] jmpl is an SFU operation that uses the compute instruction format. The second source operand of that format is not used by jmpl.

[0366] ldb, lds, ldw

[0367] ldb, lds, and ldw are memory access instructions that load an 8-bit byte, a 16-bit short, or a 32-bit word from address into the destination register r[rd]. The value loaded is a sign-extended in the destination register. The use of an immediate for the second component of the address sets the i-bit of the opcode.

[0368] The effective address from which to load must be aligned to a natural boundary for the size of the load, but the consequences of failing to do that are not defined.

[0369] The suggested assembler syntax is:

ldb	[address],rd
lds	[address],rd
ldw	[address],rd

[0370] ld [b, s, w] instructions are SFU operations that use the SFU memory format.

[0371] ldso, ldwo

[0372] ldso and ldwo are memory access instructions that load a 16-bit short or a 32-bit word from address into the destination register r[rd] using the opposite endianness from that indicated by the endian-bit of % psr. The value loaded is sign-extended in the destination register. The use of an immediate for the second component of the address sets the i-bit of the opcode.

[0373] The effective address from which to load must be aligned to a natural boundary for the size of the load, but the consequences of failing to do that are not defined.

[0374] The suggested assembler syntax is:

ldso	[address],rd
ldwo	[address],rd

[0375] ldso and ldwo are SFU operations that use the SFU memory format.

[0376] ldpair

[0377] ldpair is a memory access instruction that performs a load into a pair of adjacent registers beginning at the register specified by r[rd] from address. The use of an immediate for the second component of the address sets the i-bit of the opcode, r[rd] must be an even-numbered register.

[0378] The effective address from which to load must be even word-aligned, but the consequences of failing to do that are not defined.

[0379] The suggested assembler syntax is:

ldpair	rd,[address]

[0380] ldpair is an SFU operation that uses the SFU memory format.

[0381] ldub, ldus, lduw

[0382] ldub, ldus, and lduw are memory access instructions that load an unsigned 8-bit byte, an unsigned 16-bit short, or an unsigned 32-bit word from address into the

destination register r[rd]. The use of an immediate for the second component of the address sets the i-bit of the opcode.

[0383] The effective address from which to load must be aligned to a natural boundary for the size of the load, but the consequences of failing to do that are not defined.

[0384] While Café registers are 32-bits the behavior of lduw and ldw is identical. Future extension is liable to to change that, so appropriate consideration should be used when choosing between these instructions.

[0385] The suggested assembler syntax is:

ldub	[address],rd
ldus	[address],rd
lduw	[address],rd

[0386] ldu [b, s, w] instructions are SFU operations that use the SFU memory format.

[0387] lduso, lduwo

[0388] lduso, and lduwo are memory access instructions that load an unsigned 16-bit short or an unsigned 32-bit word from address into the destination register r[rd] using the opposite endianness from that indicated by the endian-bit of % psr. The use of an immediate for the second component of the address sets the i-bit of the opcode.

[0389] The effective address from which to load must be aligned to a natural boundary for the size of the load, but the consequences of failing to do that are not defined.

[0390] While Café registers are 32-bits the behavior of lduwo and ldwo is identical. Future extension is liable to to change that, so appropriate consideration should be used when choosing between these instructions.

[0391] The suggested assembler syntax is:

lduso	[address],rd
luwo	[address],rd

[0392] lduso and lsuwo are SFU operations that use the SFU memory format.

[0393] membar

[0394] membar is memory access instruction that specifies that all memory reference instructions already issued must be performed before any subsequent memory reference instruction may be initiated.

[0395] The suggested assembler syntax is:

[0396] membar

[0397] membar is an SFU operation that uses the compute instruction format, but uses none of that format's operands.

[0398] moveind

[0399] moveind is an integer instruction that copies the content of its first source operand, r[rs1], to the register indicated by the least significant eight bits of its second source register, r[rs2].

[0400] The suggested assembler syntax is:

[0401] moveind rs1, [rs2]

[0402] moveind is a UFU operation that uses the UFU 2-source format, but it has no use for the destination register field and does not accept an immediate for the second source operand.

[0403] mul

[0404] mul is an integer instruction that computes "r[rs1] *r[rs2]" or "r[rs1]* sign_ext(imm14)". The use of an immediate for the second source operand sets the first bit of the instruction header. The result is left in r[rd].

[0405] The suggested assembler syntax is:

[0406] mul rs1, reg_or_imm14, rd

[0407] mul is a UFU operation that uses the UFU 2-source format.

[0408] muladd

[0409] muladd is an integer instruction that computes "(r[s1]*r[s2])+r[s3]", "(r[s1 *r[s2])+sign ext(imm8)", "(r [s1]*sign_ext(imm8))+r[s3]", or "(r[s1]* sign_ext(imm8))+ sign_ext(imm8)" and puts the result in r[rd]. The use of an immediate for the second or third source operand sets the first or second bit, respectively, of the instruction header.

[0410] The suggested assembler syntax is:

[0411] muladd rs1, reg_or_imm8, reg_or_imm8, rd muladd is a UFU operation that uses the UFU 3-source format.

[0412] mulsub

[0413] mulsub is an integer instruction that computes "(r[s1]*r[s2])−r[s3]""(r[s]*r[s2])−sign_ext(imm8)", "(r[s1] *sign_ext(imm8))−r[s3]", or "(r[s1]* sign_ext(imm8))− sign_ext(imm8)" and puts the result in r[rd]. The use of an immediate for the second or third source operand sets the first or second bit, respectively, of the instruction header.

[0414] The suggested assembler syntax is:

[0415] mulsub rs1, reg_or_imm8, reg_or_imm8, rd

[0416] mul sub is a UFU operation that uses the UFU 3-source format.

[0417] ncldb, ncld s, ncldw

[0418] ncldb, ncld s, and ncldw are memory access instructions that perform a non-cacheable load of an 8-bit byte, a 16-bit short, or a 32-bit word from address into the destination register r[rd]. The value loaded is sign-extended in the destination register. The use of an immediate for the second component of the address sets the i-bit of the opcode.

[0419] The effective address from which to load must be aligned to a natural boundary for the size of the load, but the consequences of failing to do that are not defined.

[0420] The suggested assembler syntax is:

ncldb	[address],rd
ncld s	[address],rd
ncldw	[address],rd

[0421] ncld[b, s, w] instructions are SFU operations that use the SFU memory format.

[0422] ncldg

[0423] ncldg is a memory access instruction that does an uncached load of a group of eight consecutive 32-bit words from address into eight consecutive registers beginning with the one specified by r[rd]. The use of an immediate for the second component of the address sets the i-bit of the opcode. r[rd] must be 8-register aligned.

[0424] The effective address from which to load must be eight-word-aligned, but the consequences of failing to do that are not defined.

[0425] The suggested assembler syntax is:

[0426] ncldg [address], rd

[0427] ncldg was formerly known as ldg. The assembler temporarily knows the former name as an alias for the new name to ease the transition.

[0428] Note that there is no complementary ncstg instruction.

[0429] ncldg is an SFU operation that use the SFU memory format.

[0430] ncldso, ncldwo

[0431] ncldso and ncldwo are memory access instructions that perform a non-cacheable load of a 16-bit short or a 32-bit word from address into the destination register r[rd] using the opposite endianness from that indicated by the endian-bit of % psr. The value loaded is sign-extended in the destination register. The use of an immediate for the second component of the address sets the i-bit of the opcode.

[0432] The effective address from which to load must be aligned to a natural boundary for the size of the load, but the consequences of failing to do that are not defined.

[0433] The suggested assembler syntax is:

ncldso	[address],rd
ncldwo	[address],rd

[0434] ncldso and ncldwo are SFU operations that use the SFU memory format.

[0435] ncldub, ncldus, nclduw

[0436] ncldub, ncldus, and nclduw are memory access instructions that perform a non-cacheable load an unsigned 8-bit byte, an unsigned 16-bit short, or an unsigned 32-bit word from address into the destination register r[rd]. The use of an immediate for the second component of the address sets the i-bit of the opcode.

[0437] The effective address from which to load must be aligned to a natural boundary for the size of the load, but the consequences of failing to do that are not defined.

[0438] Note that while Café registers are 32-bits the behavior of nclduw and ncldw is identical, future extension is liable to to change that, so appropriate consideration should be used when choosing between these instructions.

[0439] The suggested assembler syntax is:

ncldub	[address],rd
ncldus	[address],rd
nclduw	[address],rd

[0440] ncldu [b, s, w] instructions are SFU operations that use the SFU memory format.

[0441] nclduso, nclduwo

[0442] nclduso and nclduwo are memory access instructions that perform a non-cacheable load an unsigned 16-bit short or an unsigned 32-bit word from address into the designation register r[rd] using the opposite endianness from that indicated by the endian-bit of % psr. The use of an immediate for the second component of the address sets the i-bit of the opcode.

[0443] The effective address from which to load must be aligned to a natural boundary for the size of the load, but the consequences of failing to do that are not defined.

[0444] Note that while Café registers are 32-bits the behavior of nclduwo and ncldwo is identical, future extension is liable to to change that, so appropriate consideration should be used when choosing between these instructions.

[0445] The suggested assembler syntax is:

nclduso	[address],rd
nclduwo	[address],rd

[0446] nclduso and nclduwo are SFU operations that use the SFU memory format.

[0447] ncstb, ncsts, ncstw

[0448] ncstb, ncsts, and ncstw are memory access instruction that perform a non-cacheable store an 8-bit byte, a 16-bit short, or a 32-bit word from the register specified by r[rd] to address. The use of an immediate for the second component of the address sets the i-bit of the opcode.

[0449] For ncsts and ncstw the effective address to which to store must be 2-byte aligned or 4-byte aligned, respectively, but the consequences of failing to do that are not defined.

[0450] The suggested assembler syntax is:

ncstb	rd,[address]
ncsts	rd,[address]
ncstw	rd,[address]

[0451] ncst [sb, s, w] are SFU operations that use the SFU memory format.

[0452] ncstso, ncstwo

[0453] ncsts and ncstw are memory access instruction that perform a non-cacheable store of a 16-bit short or a 32-bit word from the register specified by r[rd] to address using the

opposite endianness from that indicated by the endian-bit of % psr. The use of an immediate for the second component of the address sets the i-bit of the opcode.

[0454] For ncstso and ncstwo the effective address to which to store must be 2-byte aligned or 4-byte aligned, respectively, but the consequences of failing to do that are not defined.

[0455] The suggested assembler syntax is:

ncstso	rd,[address]
ncstwo	rd,[address]

[0456] ncstso and ncstwo are SFU operations that use the SFU memory format.

[0457] ncstpair

[0458] ncstpair is a memory access instruction that performs an uncached store of a pair of adjacent registers beginning at the register specified by r[rd] to address. The use of an immediate for the second component of the address sets the i-bit of the opcode. r[rd] must be an even-numbered register.

[0459] The effective address to which to store must be even word-aligned, but the consequences of failing to do that are not defined.

[0460] The suggested assembler syntax is:

ncstpair	rd,[address]

[0461] ncstpair is an SFU operation that uses the SFU memory format.

[0462] nop

[0463] nop is a control flow instruction that does nothing. It has the special property of being unique in its leading byte with its remaining bytes ignored.

[0464] The suggested assembler syntax is:

[0465] nop

[0466] nop is an SFU operation that uses the branch instruction format, but only the leading 6 bits of its opcode are significant and none of the other fields is used.

[0467] not

[0468] not is a logical instruction that computes the bit-wise complement of r[rs1], leaving the result is r[rd].

not	rs1,rd

[0469] The SFU version of not uses the SFU compute format, and the UFU version uses the UFU 2-source format. This instruction has no use for a second source operand; the assembler infers r0 in its place for purely neurotic reasons.

[0470] or

[0471] or is a logical instruction that computes "r[rs1] r[rs2]" or "r[rs1]|imm14". The use of an immediate for the second source operand sets the i-bit of the opcode of the SFU version or the first header bit of the UFU version. The bit-wise logical result is left in r[rd].

[0472] The suggested assembler syntax is:

[0473] or rs1, reg_or_imm14, rd

[0474] The SFU version of or uses the SFU compute format, and the UFU version uses the UFU 2-source format.

[0475] pack

[0476] pack is a pixel instruction that treats its first two source operands, r[rs1] and r[rs2], as two pair of unsigned 16-bit operands. Each 16-bit operand is shifted right by the the value of the the low-order 4 bits of the third source operand, r[rs3].

[0477] The low-order 8 bits of the resulting values are packed into the result register r[rd], with the value derived from 31:16 of r[rs1] in bits 31:24, the value derived from 15:0 of r[rs1] in bits 23:16, the value derived from 31:16 of r[rs2] in bits 15:8, and the value derived from 15:0 of r[rs2] in bits 7:0.

[0478] The suggested assembler syntax is:

pack	rs1,rs2,rs3,rd

[0479] pack is a UFU operation that uses the UFU 3-source format.

[0480] padd, padds

[0481] padd and padds are integer instructions that compute "r[rs1]+r[rs2]", where each of the sources is treated as a pair of independent 16-bit quantities yeilding a pair of independent 16-bit sums in r[rd]. padd also can have a 14-bit immediate as its second source operand, in which case the sign-extended immediate is added to each of 16-bit numbers in r[rs1] and the two 16-bit sums are left in r[rd].

[0482] padd produces ordinary two's complement integer results, and padds produces saturated results.

[0483] The suggested assembler syntax is:

padd	rs1, reg_or_imm14, rd
padds	rs1, rs2, rd

[0484] padd and padds are UFU operations that use the UFU 2-source format.

[0485] pcmovenz

[0486] pcmovenz is an integer instruction that uses two 16-bit flags in r[rs1] to control whether the corresponding 16-bit fields of r[rs2] are copied to the same positions of r[rd]. A field of r[rs2] is copied to r[rd] is the corresponding flag field of r[rs1] is non-zero. The flags in r[rs1] will most likely be the result of a preceding pcmpeq or pcmplt.

[0487] The suggested assembler syntax is:

pcmovenz	rs1,rs2,rd

[0488] pcmovenz is a UFU operation that uses the UFU 2-source format.

[0489] pcmovez

[0490] pcmove z is an integer instruction that uses two 16-bit flags in r[rs1] to control whether the corresponding 16-bit fields of r[rs2] are copied to the same positions of r[rd]. A field of r[rs2] is copied to r[rd] is the corresponding flag field of r[rs1] is zero. The flags in r[rs1] will most likely be the result of a preceding pcmpeq or pcmplt.

[0491] The suggested assembler syntax is:

pcmovez	rs1,rs2,rd

[0492] pcmovez is a UFU operation that uses the UFU 2-source format.

[0493] pcmpeq

[0494] pcmpeq is an integer instruction that compares for equality the pair of shorts in its first source register with either a pair of shorts in its second source register or with a signed 14-bit immediate.

[0495] That is, when the second source operand is a register, the short in bits 31:16 of r[rd] is set to one if "r[rs1]<31.16>==r[rs2]<31:16>" and zero otherwise, and the short in bits 15:0 of r[rd] is set to one if "r[rs1]<15:0>== r[rs2]<15:0>" and zero otherwise. When the second source operand is an immediate, the short in bits 31:16 of r[rd] is set to one if "r[rs1]<31:16>==sign_ext(imm14)" and zero otherwise, and the short in bits 15:0 of r[rd] is set to one if "r[rs1]<15:0>==sign_ext(imm14)" and zero otherwise.

[0496] The suggested assembler syntax is:

pcmpeq	rs1,reg_or_imm14,rd

[0497] pcmpeq is a UFU operation that uses the UFU 2-source format.

[0498] pcmplt

[0499] pcmplt is an integer instruction that does a "compare less than" of pair of shorts in its first source source register with either a pair of shorts in its second source register or with a signed 14-bit immediate.

[0500] That is, when the second source operand is a register, the short in bits 31:16 of r[rd] is set to one if "r[rs1]<31:16><r[rs2]<31:16>" and zero otherwise, and the short in bits 15:0 of r[rd] is set to one if "r[rs1]<15:0><r [rs2]<15:0>" and zero otherwise. When the second source operand is an immediate, the short in bits 31:16 of r[rd] is set to one if "r[rs1]<31:16><sign_ext(imm14)" and zero otherwise, and the short in bits 15:0 of r[rd] is set to one if "r[rs1]<15:0><sign_ext(imm14)" and zero otherwise.

[0501] The suggested assembler syntax is:

pcmplt	rs1,reg_or_imm14,rd

[0502] pcmplt is a UFU operation that uses the UFU 2-source format.

[0503] pdist

[0504] pdist is a pixel instruction that treats each of its two source registers, r[rs1] and r[rs2], as four unsigned 8-bit values, subtracts the corresponding pairs, and adds the sum of the absolute values of those differences to the value in the register specified by r[rd].

[0505] The suggested assembler syntax is:

pdist	rs1,rs2,rd

[0506] pdist is a UFU operation that uses the UFU 2-source format.

[0507] pmean

[0508] pmean is a pixel instruction that treats its source operands r[rs1] and r[rs2] as pairs of unsigned 16-bit integers and computes a pair of mean values in r[rd]. These means are rounded high according to the formula: "r[rd] <15:0>=(r[rs1]<15:0>+r[rs2]<15:0>+1)>>1" and likewise for the other halves.

[0509] The suggested assembler syntax is:

pmean	rs1,rs2,rd

[0510] pmean is a UFU operation that uses the UFU 2-source format.

[0511] pmul

[0512] pmul is an integer instruction that multiplies the pair of 16-bit operands in r[rs1] with either a a pair of 16-bit operands in r[rs2] or with a sign-extended 14-bit immediate, placing a pair of independent 16-bit products in r[rd].

[0513] That is, when the second source operand is a register, bits 31:16 of r[rd] are set to the product bits 31:16 of r[rs1] and bits 31:16 of r[rs2] and bits 15:0 of r[rd] are set to the product bits 15:0 of r[rs1] and bits 15:0 of r[rs2]. When the second source operand is an immediate, bits 31:16 of r[rd] are set to the product bits 31:16 of r[rs1] and sign_ext(imm14) and bits 15:0 of r[rd] are set to the product bits 15:0 of r[rs1] and sign_ext(imm14).

[0514] The format of the operands and results is indicated by the saturation field of the Processor Status Register, but this operation does not saturate.

[0515] The suggested assembler syntax is:

[0516] pmul rs1, reg_or_imm14, rd

[0517] pmul is a UFU operation that uses the UFU 2-source format.

[0518] pmuladd, pmuladds

[0519] pmuladd and pmuladd are integer instructions that compute "(r[rs1]*r[rs2])+r[s3]", where each of the sources is treated as a pair of independent 16-bit quantities yeilding a pair of independent 16-bit results in r[rd]. The format of the operands and results is indicated by the saturation field of the Processor Status Register, but pmuladd does not saturate and pmuladds does.

[0520] The suggested assembler syntax is:

pmuladd	rs1,rs2,rs3,rd
pmuladds	rs1,rs2,rs3,rd

[0521] pmuladd and pmuladd are UFU operations that use the UFU 3-source format.

[0522] pmulsub, pmulsubs

[0523] pmulsub and pmulsub are integer instructions that compute "(r[rs1]*r[rs2])−r[s3]", where each of the sources is treated as a pair of independent 16-bit quantities yeilding a pair of independent 16-bit results in r[rd]. The format of the operands and results is indicated by the saturation field of the Processor Status Register, but pmulsub does not saturate and pmulsubs does.

[0524] The suggested assembler syntax is:

pmulsub	rs1,rs2,rs3,rd
pmulsubs	rs1,rs2,rs3,rd

[0525] pmul sub and pmul subs are UFU operations that use the UFU 3-source format.

[0526] ppower

[0527] ppower is an fixed-point instruction that computes "r[rs1]**r[rs2]", where each of the sources is treated as a pair of independent 16-bit quantities yeilding a pair of independent 16-bit powers in r[rd].

[0528] By stipulation, zero to any power is zero.

ppower	rs1,rs2,rd

[0529] ppower is a SFU operation that uses the SFU compute format.

[0530] precsqrt

[0531] precsqrt is a fixed-point instruction that computes a pair of fixed-point reciprocal square roots of the 16-bit values of r[rs1]. The results are delivered in r[rd].

[0532] The suggested assembler syntax is:

precsqrt	rs1,rd

[0533] precsqrt is a SFU operation that uses the SFU compute format. The second source operand for the format is unused by this instruction.

[0534] pshll

[0535] pshll is a logical instruction that shifts each of the the pair of 16-bit operands in r[rs1] left by either the lower-order 4 bits of the corresponding half of r[rs2] or the low-order 4 bits of its immediate. The shifted results are left in r[rd].

[0536] The suggested assembler syntax is:

pshll	rs1,reg_or_imm14,rd

[0537] pshll is a UFU operation that uses the UFU 2-source format.

[0538] pshra

[0539] pshra is a logical instruction that performs a right arithmetic shift of the pair of 16-bit operands in r[rs1]. The shift count of each is either the low-order 4 bits of the corresponding half of r[rs2] or the low-order 4 bits of its immediate. The shifted results are left in r[rd].

[0540] The suggested assembler syntax is:

pshra	rs1,reg_or_imm14,rd

[0541] pshra is a UFU operation that uses the UFU 2-source format.

[0542] pshrl

[0543] pshrl is a logical instruction that performs a right logical shift of the pair of 16-bit operands in r[rs1]. The shift count of each is either the low-order 4 bits of the corresponding half of r[rs2] or the low-order 4 bits of its immediate. The shifted results are left in r[rd].

[0544] The suggested assembler syntax is:

pshrl	rs1,reg_or_imm14,rd

[0545] pshrl is a UFU operation that uses the UFU 2-source format.

[0546] psub, psubs

[0547] psub and psubs are integer instructions that compute "r[rs1]−r[rs2]", where each of the sources is treated as a pair of independent 16-bit quantities yielding a pair of independent 16-bit differences in r[rd]. psub also can have a 14-bit immediate as its second source operand, in which case

the sign-extended immediate is subtracted from each of 16-bit numbers in r[rs1] and the two 16-bit differences are left in r[rd].

[0548] psub produces ordinary two's-complement integer results, and psubs produces saturated results.

[0549] The suggested assembler syntax is:

psub	rs1,reg_or_imm14,rd
psubs	rs1,rs2,rd

[0550] psub and psubs are UFU operations that use the UFU 2-source format.

[0551] rem

[0552] rem is an integer instruction that computes "r[rs1] % r[rs2]" or "r[rs1] % sign_ext(imm14)". The use of an immediate for the second source operand sets the i-bit of the opcode. The result is left in r[rd].

[0553] If the second source operand is zero, rem will trap.

[0554] The suggested assembler syntax is:

rem	rs1,reg_or_imm14,rd

[0555] rem is an SFU operation that uses the SFU compute format.

[0556] retry

[0557] retry is a control flow instruction causes a control transfer from a trap handler to the instruction word that caused the trap. Please refer to the Traps chapter of the Café Microarchitecture specification for the complete description.

[0558] The suggested assembler syntax is:

[0559] retry

[0560] retry is a SFU operation that uses the SFU compute format, but it has no use for any operand.

[0561] return

[0562] returnis a control flow instruction that causes a register-indirect control transfer to the address in "r[rs1]".

[0563] The suggested assembler syntax is:

return	rs1

[0564] return is a pseudo-op. What it really means is "jmpl r[rs1]+0, r0".

[0565] s2ib, s2 is, s2iw

[0566] Note: s2 is and s2iw might be removed. This section will be rewrtten when the matter is settled.

[0567] s2ib, s2is, and s2iw are memory access instruction that store an 8-bit byte, a 16-bit short, or a 32-bit word from the register specified by r[rd] to address. The use of an immediate for the second component of the address sets the i-bit of the opcode. The intent is that these are the instructions to be used to modify code on the fly, so these stores guarantee instruction cache consistency.

[0568] For s2 is and s2iw the effective address to which to store must be 2-byte aligned or 4-byte aligned, respectively, but the consequences of failing to do that are not defined. Because of this alignment requirement and the fact that Café instructions can be at any byte boundary, care must be taken in choosing the right instruction.

[0569] The suggested assembler syntax is:

s2ib	rd,[address]
s2is	rd,[address]
s2iw	rd,[address]

[0570] s2i [sb, s, w] are SFU operations that use the SFU memory format.

[0571] sethi

[0572] sethi is an integer instruction that places its immediate operand in the high-order 22 bits of r[rd] and clears the low-order 10 bits. It is frequently used with the % hi operator to form base addresses for subsequent memory references.

[0573] The suggested assembler syntax is:

sethi	imm22,rd
sethi	%hi(label),rd

[0574] sethi is an SFU operation that uses a format of its own.

[0575] setir

[0576] setir is an integer instruction that sets the internal register the ordinal of which is its r[rd] operand to the value in its r[rs1] operand.

[0577] The suggested assembler syntax is:

setir	rs1,rd

[0578] setir is an SFU operation that uses the SFU compute format in an irregular way. It has no use for the second source operand, and the destination register is an internal register number, NOT one of the general purpose registers.

[0579] shll

[0580] shll is a logical instruction that computes "r[rs1] <<r[rs2]" or "r[rs1]<<imm". The use of an immediate for the second source operand sets the i-bit of the opcode of the SFU version or the first header bit of the UFU version. The result is left in r[rd]. Only the low-order 5 bits of the second source operand are used.

[0581] The suggested assembler syntax is:

shll	rs1,reg_or_imm,rd

[0582] The SFU version of sh 1 uses the SFU compute format, and the UFU version uses the UFU 2-source format.

[0583] shra

[0584] shra is a logical instruction that computes "r[rs1] >>r[rs2]" or "r[rs1]>>imm". The use of an immediate for the second source operand sets the i-bit of the opcode of the SFU version or the first header bit of the UFU version. The result is left in r[rd]. The first source operand is treated as a signed integer, so the result is sign-extended. Only the low-order 5 bits of the second source operand are used.

[0585] The suggested assembler syntax is:

shra	rs1,reg_or_imm,rd

[0586] The SFU version of shra uses the SFU compute format, and the UFU version uses the UFU 2-source format.

[0587] shrl

[0588] shrl is a logical instruction that computes "r[rs1] >>r[rs2]" or "r[rs1]>>imm". The use of an immediate for the second source operand sets the i-bit of the opcode of the SFU version or the first header bit of the UFU version. The result is left in r[rd]. The first source operand is treated as an unsigned integer, so the result is not sign-extended. Only the low-order 5 bits of the second source operand are used.

[0589] The suggested assembler syntax is:

[0590] shrl rs1, reg_or_imm, rd

[0591] The SFU version of shri uses the SFU compute format, and the UFU version uses the UFU 2-source format.

[0592] sir

[0593] sir is a control flow instruction that resets the machine. sir is a privileged instruction; executing it when the Supervisor Mode flag of % psr is clear causes a privileged instruction trap.

[0594] The suggested assembler syntax is:

[0595] sir

[0596] sir is an SFU operation that uses the SFU compute format, but has no use for any of that format's operands.

[0597] softtrap

[0598] softtrap is a control flow instruction that generates a trap. The ordinal of the trap is specified by r[rs1]+r[rs2] or r[rs1]+sign_ext(imm14). For details on how traps work, see the Traps chapter of the Café Microarchitecture specification.

[0599] The suggested assembler syntax is:

softtrap	rs1,reg_or_imm14

[0600] softtrap is an SFU operation that uses the SFU compute format but has no use for the destination register field of that format.

[0601] stb, sts, stw

[0602] stb, sts, and stw are memory access instruction that store an 8-bit byte, a 16-bit short, or a 32-bit word from the register specified by r[rd] to address. The use of an immediate for the second component of the address sets the i-bit of the opcode.

[0603] For sts and stw the effective address to which to store must be 2-byte aligned or 4-byte aligned, respectively, but the consequences of failing to do that are not defined.

[0604] The suggested assembler syntax is:

stb	rd,[address]
sts	rd,[address]
stw	rd,[address]

[0605] St [sv, s, w] are SFU operations that use the SFU memory format.

[0606] stg

[0607] stg would be the mnemonic for a store group instruction if the were one, but there is not. Most careful readers notice this asymmetry and ask whether it is an oversight, so this note hopes to explain and forestall that question.

[0608] Since the register file read port can deliver no more than 64 bits per cycle, a store of a group would induce a 3-cycle stall. Doing the stores with four stpair instructions would make better use of the issue bandwidth because that would allow work to proceed on the other units.

[0609] stso, stwo

[0610] sts and stw are memory access instruction that store a 16-bit short or a 32-bit word from the register specified by r[rd] to address using the opposite endianness from that indicated by the endian-bit of % psr. The use of an immediate for the second component of the address sets the i-bit of the opcode.

[0611] For stso and stwo the effective address to which to store must be 2-byte aligned or 4-byte aligned, respectively, but the consequences of failing to do that are not defined.

[0612] The suggested assembler syntax is:

stso	rd,[address]
stwo	rd,[address]

[0613] stso and stwo are SFU operations that use the SFU memory format.

[0614] stpair

[0615] stpair is a memory access instruction that performs a store of a pair of adjacent registers beginning at the register specified by r[rd] to address. The use of an immediate for the second component of the address sets the i-bit of the opcode. r[rd] must be an even-numbered register.

[0616] The effective address to which to store must be even word-aligned, but the consequences of failing to do that are not defined.

[0617] The suggested assembler syntax is:

stpair	rd,[address]

[0618] stpair is an SFU operation that uses the SFU memory format.

[0619] sub

[0620] sub is an integer instruction that computes r[rs1]−r[rs2] or r[rs1]−sign_ext(imml4). The use of an immediate for the second source operand sets the i-bit of the opcode of the SFU version or the first header bit of the UFU version. The result is left in r[rd].

[0621] The suggested assembler syntax is:

sub	rs1,reg_or_imm14,rd

[0622] The SFU version of sub uses the SFU compute format, and the UFU version uses the UFU 2-source format.

[0623] xor

[0624] xor is a logical instruction that computes "r[rs1] ^r[rs2]" or "r[rs1]^imm]4". The use of an immediate for the second source operand sets the i-bit of the opcode of the SFU version or the first header bit of the UFU version. The bit-wise logical result is left in r[rd].

[0625] The suggested assembler syntax is:

xor	rs1,reg_or_imm14,rd

[0626] The SFU version of xor uses the SFU compute format, and the UFU version uses the UFU 2-source format.

What is claimed is:

1. A processor comprising:

a plurality of independent processor elements in a single integrated circuit chip capable of executing a respective plurality of threads concurrently in a multiple-thread mode of operation; and

at least some of the independent processing elements including plural processing units,

wherein at least some of the threads are executable in parallel on plural ones of the processing units in accordance with an instruction set that encodes the parallel execution.

* * * * *