MULTIPLE-THREAD PROCESSOR FOR THREADED SOFTWARE APPLICATIONS

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)

Assignee: Sun Microsystems, Inc.

Filed: Apr. 6, 2004

Related U.S. Application Data
Continuation of application No. 09/204,480, filed on Dec. 3, 1998, now Pat. No. 6,718,457.

Publication Classification
Int. Cl: G06F 15/00
U.S. Cl: 712/10

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.
FIG. 2
Broadcast Writes (5)

316
RF3

314
RF2

312
RF1

310
RF0

3 Read Ports

FIG. 3

Global Registers
12R/4W or
12R/5W

216

Register File

MFU1

220

MFU2

220

MFU3

220

GFU

222

FIG. 4
Data In
(up to 3 32-bit registers)

5 Cycles
(globally visible latency)

Data Out
(one 32-bit register)

Hardware bypasses provide shortest internal latency (4, 2 and 1 cycle latency)

FIG. 5

FIG. 6
REGISTER FILE INCLUDING MEMORY ARRAY

FIG. 7A

FIG. 7B

word lines (metal 2)

bit lines (metal 3) 1 micron
FIG. 8
### FIG. 10A

<table>
<thead>
<tr>
<th>hdr</th>
<th>opcode</th>
<th>rd</th>
<th>rs1</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdr</td>
<td>opcode</td>
<td>rd</td>
<td>rs1</td>
<td>immediate</td>
</tr>
</tbody>
</table>

### FIG. 10B

<table>
<thead>
<tr>
<th>hdr</th>
<th>opcode</th>
<th>rd</th>
<th>immediate</th>
</tr>
</thead>
</table>

### FIG. 10C

<table>
<thead>
<tr>
<th>hdr</th>
<th>opcode</th>
<th>address</th>
</tr>
</thead>
</table>

### FIG. 10D

<table>
<thead>
<tr>
<th>hdr</th>
<th>opcode</th>
<th>HT</th>
<th>rd</th>
<th>displacement</th>
</tr>
</thead>
</table>

### FIG. 10E

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>opcode</td>
<td>rd</td>
<td>rs1</td>
<td>rs2</td>
</tr>
<tr>
<td>01</td>
<td>opcode</td>
<td>rd</td>
<td>rs1</td>
<td>rs2</td>
</tr>
<tr>
<td>10</td>
<td>opcode</td>
<td>rd</td>
<td>rs1</td>
<td>immediate</td>
</tr>
<tr>
<td>11</td>
<td>opcode</td>
<td>rd</td>
<td>rs1</td>
<td>immediate</td>
</tr>
</tbody>
</table>
FIG. 10F

FIG. 11

n, m unsigned
16 bits.
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.

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.

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.

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 allocated to each functional unit each contain the same content. A multiported 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

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.

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.

FIG. 2 is a schematic block diagram showing the core of the processor.

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

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

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

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

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.

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

FIG. 9 is a schematic timing diagram that illustrates timing of the processor pipeline.

FIGS. 10A, 10B, 10C, 10D, 10E and 10F illustrate instruction formats.

FIG. 11 illustrates operation of a bitext instruction.

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

DESCRIPTION OF THE EMBODIMENT(S)

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 operational units executing an instruction set having special data-handling instructions that are advantageous in a multiple-thread environment.

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 multi-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.

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 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 scalable 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 is 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 are 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 (SIMD) 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 16K capacity is suitable for performing graphic loops, other multimedia tasks or processes, and general-purpose Java™ code. Coherence 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 store2. Software uses the store2 instruction to maintain coherence 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 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 logic 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 have 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 still 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,前期qrt 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 decomposition 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-
The active 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 are 32 bits wide and are used to write operations of the general functional unit 222 and the media functional units 220.

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

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.

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.

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.

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.

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.

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

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.

In various embodiments, dedicating fields for global, trap, and the like leverages the split register file 216.

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 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.

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 RAN 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 RAN 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 1mm separates the word lines 744 and a distance of approximately 1μm 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 1μm apart, one memory cell would have an integrated circuit area of 256μm² (16x16). 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 write 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 1μm between lines has an area of approximately 49 μm². 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
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.

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.

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.

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.

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.

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.

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 1 μm. In other processors the spacing between lines is less than 1 μm.

Exemplary Instruction Set Architecture

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.

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.

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.

General Purpose Registers

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

[R]_eligit<eligits>
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.

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

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 assembler, but some users believe it’s useful to see it in the source.

r0 is a flat-zero source operand and a result-sink registers.

Control and Status Registers

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

Program Counters

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.

Bit 24 of %psr specifies The processor ID. Clear denotes cp0, and set denotes cp1.

Bits 23 and 22 of %psr specify the current Trap Level.

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.

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

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

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

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

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

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.

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.

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

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

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 S.13 fixed-point. The simulator is using bits 8 and 9 of %psr for this specification.

<table>
<thead>
<tr>
<th>mode</th>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000000000...0</td>
<td>011111111...1</td>
</tr>
<tr>
<td>01</td>
<td>100000000...0</td>
<td>011111111...1</td>
</tr>
<tr>
<td>10</td>
<td>100000000...0</td>
<td>011111111...1</td>
</tr>
<tr>
<td>11</td>
<td>111100000...0</td>
<td>001000000...0</td>
</tr>
</tbody>
</table>

(For those of us habituated to the common floating-point representation, the Sif 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 S.2,13 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.)

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 list 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<<\((N>>5))\).

Trap Base Register

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 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] Algebraically, 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 values and instruction word contents they indicate are:

<table>
<thead>
<tr>
<th>header value</th>
<th>instructions in instruction word</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SFU only</td>
</tr>
<tr>
<td>01</td>
<td>SFU + UFU1</td>
</tr>
<tr>
<td>10</td>
<td>SFU + UFU1 + UFU2</td>
</tr>
<tr>
<td>11</td>
<td>SFU + UFU1 + UFU2 + UFU3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>header value</th>
<th>Instruction class</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Call and branch</td>
</tr>
<tr>
<td>01</td>
<td>Compute</td>
</tr>
<tr>
<td>10</td>
<td>Memory (uncacheable)</td>
</tr>
<tr>
<td>11</td>
<td>Memory (cacheable)</td>
</tr>
</tbody>
</table>

[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:

<table>
<thead>
<tr>
<th>Memory (cacheable, leading 11) Opcodes</th>
<th>opcode[7:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode[7:3]</td>
<td>0xx-(unsigned)</td>
</tr>
<tr>
<td>byte</td>
<td>short</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
</tr>
<tr>
<td>ldb</td>
<td>lds</td>
</tr>
<tr>
<td>ldx</td>
<td>lduo</td>
</tr>
<tr>
<td>ldx</td>
<td>ldu</td>
</tr>
<tr>
<td>estb</td>
<td>ests</td>
</tr>
<tr>
<td>estb</td>
<td>ests</td>
</tr>
<tr>
<td>estb</td>
<td>ests</td>
</tr>
</tbody>
</table>
### Memory (uncacheable, leading 10) Opcodes

<table>
<thead>
<tr>
<th>opcode [7:3]</th>
<th>byte</th>
<th>short</th>
<th>word</th>
<th>long</th>
<th>byte</th>
<th>short</th>
<th>word</th>
<th>long</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>neldub</td>
<td>neldas</td>
<td>neldaw</td>
<td>neldg</td>
<td>neldlb</td>
<td>nelds</td>
<td>neldw</td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td>nelduo</td>
<td>nelduwo</td>
<td>nelduo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>nestb</td>
<td>nesets</td>
<td>nesetw</td>
<td>nesetpr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 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

<table>
<thead>
<tr>
<th>opcode[7:3]</th>
<th>opcode[2:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>01ixx</td>
<td>000</td>
</tr>
<tr>
<td>i = 0</td>
<td>01000</td>
</tr>
<tr>
<td></td>
<td>01001</td>
</tr>
<tr>
<td></td>
<td>01010</td>
</tr>
<tr>
<td></td>
<td>01011</td>
</tr>
<tr>
<td>i = 1</td>
<td>01100</td>
</tr>
<tr>
<td></td>
<td>01101</td>
</tr>
<tr>
<td></td>
<td>01110</td>
</tr>
<tr>
<td></td>
<td>01111</td>
</tr>
</tbody>
</table>

#### 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 bsz, 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

<table>
<thead>
<tr>
<th>opcode[7:3]</th>
<th>opcode[2:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>06axx</td>
<td>000</td>
</tr>
<tr>
<td>00000</td>
<td>call</td>
</tr>
<tr>
<td>00001</td>
<td>bnez</td>
</tr>
<tr>
<td>00010</td>
<td>softrap</td>
</tr>
<tr>
<td>00011</td>
<td>—</td>
</tr>
<tr>
<td>00100</td>
<td>nop</td>
</tr>
<tr>
<td>00101</td>
<td>—</td>
</tr>
<tr>
<td>00110</td>
<td>softrap</td>
</tr>
<tr>
<td>00111</td>
<td>—</td>
</tr>
</tbody>
</table>
The SFU instruction formats are:

- Memory and compute
- Sethi
- Call

Only the Sethi instruction uses this format.

Only the bz and bnz instructions use this format.

UFU Instruction Formats

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>argument list</th>
<th>opcode</th>
<th>L</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-0100000</td>
<td>1</td>
<td>Add</td>
</tr>
<tr>
<td>addc</td>
<td>rs1,rs2,rs3,rd</td>
<td>U-000000</td>
<td>1</td>
<td>Add with carry</td>
</tr>
<tr>
<td>cmovnez</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-0101001</td>
<td>1</td>
<td>Move if not zero</td>
</tr>
<tr>
<td>cmovz</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-010000</td>
<td>1</td>
<td>Move if zero</td>
</tr>
<tr>
<td>cmpmt</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-0110010</td>
<td>1</td>
<td>Compare equals</td>
</tr>
<tr>
<td>cmpl</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-101100</td>
<td>1</td>
<td>Compare less</td>
</tr>
<tr>
<td>cmpn</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-101101</td>
<td>1</td>
<td>Conditionally (non-zero) pick</td>
</tr>
<tr>
<td>cmpk</td>
<td>rs1,reg_or_imm8,reg_or_imm8,rd</td>
<td>U-110101</td>
<td>1</td>
<td>Conditionally (zero) pick</td>
</tr>
<tr>
<td>cmpllt</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-0110101</td>
<td>1</td>
<td>Compare less</td>
</tr>
<tr>
<td>genborrow</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-101100</td>
<td>1</td>
<td>Generate borrow</td>
</tr>
<tr>
<td>gencarry</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-101101</td>
<td>1</td>
<td>Generate carry</td>
</tr>
<tr>
<td>getreg</td>
<td>rs1,rd</td>
<td>S-0101110</td>
<td>1</td>
<td>Get internal register</td>
</tr>
<tr>
<td>sdiv</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-0100000</td>
<td>2</td>
<td>Divide</td>
</tr>
<tr>
<td>sdiv</td>
<td>rs1,[rs2]</td>
<td>U-100010</td>
<td>2</td>
<td>Move indirect</td>
</tr>
<tr>
<td>mul</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-001000</td>
<td>2</td>
<td>Multiply</td>
</tr>
<tr>
<td>muladd</td>
<td>rs1,reg_or_imm8,reg_or_imm8,rd</td>
<td>U-010000</td>
<td>2</td>
<td>Multiply add</td>
</tr>
<tr>
<td>mulsub</td>
<td>rs1,reg_or_imm8,reg_or_imm8,rd</td>
<td>U-010001</td>
<td>2</td>
<td>Multiply add</td>
</tr>
<tr>
<td>padd</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-000100</td>
<td>1</td>
<td>Parallel add</td>
</tr>
<tr>
<td>padd</td>
<td>rs1,rs2,rd</td>
<td>U-110100</td>
<td>1</td>
<td>Saturating parallel add</td>
</tr>
<tr>
<td>pmovnez</td>
<td>rs1,rs2,rd</td>
<td>U-111000</td>
<td>1</td>
<td>Parallel move if not zero</td>
</tr>
<tr>
<td>pemove</td>
<td>rs1,rs2,rd</td>
<td>U-111001</td>
<td>1</td>
<td>Parallel move if zero</td>
</tr>
<tr>
<td>opcpeq</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-110001</td>
<td>1</td>
<td>Parallel compare</td>
</tr>
<tr>
<td>opcplt</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-110101</td>
<td>1</td>
<td>Parallel compare less than</td>
</tr>
<tr>
<td>pmul</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-000111</td>
<td>2</td>
<td>Parallel multiply</td>
</tr>
<tr>
<td>pmuladd</td>
<td>rs1,rs2,rs3,rd</td>
<td>U-001010</td>
<td>2</td>
<td>Parallel multiply add</td>
</tr>
<tr>
<td>pmuladd</td>
<td>rs1,rs2,rs3,rd</td>
<td>U-111010</td>
<td>2</td>
<td>Saturating parallel multiply add</td>
</tr>
<tr>
<td>pmulsub</td>
<td>rs1,rs2,rs3,rd</td>
<td>U-001011</td>
<td>2</td>
<td>Parallel multiply subtract</td>
</tr>
<tr>
<td>pmulsub</td>
<td>rs1,rs2,rs3,rd</td>
<td>U-111011</td>
<td>2</td>
<td>Saturating parallel multiply subtract</td>
</tr>
<tr>
<td>psub</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-000100</td>
<td>1</td>
<td>Parallel subtract</td>
</tr>
<tr>
<td>psub</td>
<td>rs1,rs2,rd</td>
<td>U-110101</td>
<td>1</td>
<td>Saturating parallel subtract</td>
</tr>
<tr>
<td>rem</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-0101001</td>
<td>1</td>
<td>Remainder</td>
</tr>
<tr>
<td>seth</td>
<td>imm23,rd</td>
<td>S-0100100</td>
<td>1</td>
<td>Sethi</td>
</tr>
<tr>
<td>set</td>
<td>rs1,rd</td>
<td>S-0101111</td>
<td>1</td>
<td>Set internal register</td>
</tr>
<tr>
<td>sub</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-0100001</td>
<td>1</td>
<td>Subtract</td>
</tr>
<tr>
<td>subc</td>
<td>rs1,rs2,rs3,rd</td>
<td>U-000011</td>
<td>1</td>
<td>Subtract with carry</td>
</tr>
</tbody>
</table>
[0159] Logical Instructions

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>argument list</th>
<th>opcode</th>
<th>L operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-01001010</td>
<td>1 Add</td>
</tr>
<tr>
<td>cccb</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-011110</td>
<td>1 Count consecutive clear bits</td>
</tr>
<tr>
<td>not</td>
<td>rs1,rd</td>
<td>S-01001000</td>
<td>1 Not</td>
</tr>
<tr>
<td>or</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-01001011</td>
<td>1 Or</td>
</tr>
<tr>
<td>psbll</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-011100</td>
<td>1 Parallel shift left logical</td>
</tr>
<tr>
<td>psbra</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-011110</td>
<td>1 Parallel shift right arithmetic</td>
</tr>
<tr>
<td>psbhr</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-011101</td>
<td>1 Parallel shift right logical</td>
</tr>
<tr>
<td>shll</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-01100000</td>
<td>1 Shift left logical</td>
</tr>
<tr>
<td>shra</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-01100000</td>
<td>1 Shift left arithmetic</td>
</tr>
<tr>
<td>shrl</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-01100001</td>
<td>1 Shift right logical</td>
</tr>
<tr>
<td>xor</td>
<td>rs1,reg_or_imm14,rd</td>
<td>S-01001111</td>
<td>1 Exclusive or</td>
</tr>
</tbody>
</table>

[0160] Floating Point Instructions

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>argument list</th>
<th>opcode</th>
<th>L operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>clip</td>
<td>rs1,rs2,rs3,rd</td>
<td>U-011111</td>
<td>1 Clip</td>
</tr>
<tr>
<td>fadd</td>
<td>rs1,rs2,rd</td>
<td>U-0101100</td>
<td>4 Single precision addition</td>
</tr>
<tr>
<td>fempeq</td>
<td>rs1,rs2,rd</td>
<td>S-01011000</td>
<td>1 FP compare equals</td>
</tr>
<tr>
<td>femple</td>
<td>rs1,rs2,rd</td>
<td>S-01011010</td>
<td>1 FP compare less than or equals</td>
</tr>
<tr>
<td>fcmplt</td>
<td>rs1,rs2,rd</td>
<td>S-01011001</td>
<td>1 FP compare less than or equals</td>
</tr>
<tr>
<td>fldv</td>
<td>rs1,rs2,rd</td>
<td>S-01011100</td>
<td>6 Single precision division</td>
</tr>
<tr>
<td>fnmul</td>
<td>rs1,rs2,rd</td>
<td>U-003110</td>
<td>4 Single precision multiplication</td>
</tr>
<tr>
<td>fnmuladd</td>
<td>rs1,rs2,rs3,rd</td>
<td>U-010000</td>
<td>4 Single-precision multiply-add</td>
</tr>
<tr>
<td>fnmulsub</td>
<td>rs1,rs2,rs3,rd</td>
<td>U-010001</td>
<td>4 Single precision multiply-subtract</td>
</tr>
<tr>
<td>fnmsqrt</td>
<td>rs1,rd</td>
<td>S-01011010</td>
<td>6 Single precision reciprocal square root</td>
</tr>
<tr>
<td>fsub</td>
<td>rs1,rs2,rd</td>
<td>U-003110</td>
<td>4 Single precision subtraction</td>
</tr>
</tbody>
</table>

[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.

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>argument list</th>
<th>opcode</th>
<th>L operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppower</td>
<td>rs1,rs2,rd</td>
<td>S-01010100</td>
<td>6 Parallel exponentiation</td>
</tr>
<tr>
<td>presqrt</td>
<td>rs1,rd</td>
<td>S-01011101</td>
<td>6 Parallel reciprocal square root</td>
</tr>
</tbody>
</table>

[0163] Convert Instructions

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>argument list</th>
<th>opcode</th>
<th>L operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>fix2flt</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-010111</td>
<td>4 Fixed point to single precision</td>
</tr>
<tr>
<td>flt2fix</td>
<td>rs1,reg_or_imm14,rd</td>
<td>U-010110</td>
<td>4 Fixed point to fixed point</td>
</tr>
</tbody>
</table>

[0164] Control Flow Instructions

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>argument list</th>
<th>opcode</th>
<th>L operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bneck</td>
<td>rs1,reg_or_imm8,reg_or_imm8</td>
<td>U-011011</td>
<td>? Bound check</td>
</tr>
<tr>
<td>bne</td>
<td>rs1,reg_or_imm8</td>
<td>S-00001001</td>
<td>1 Branch if not zero</td>
</tr>
<tr>
<td>bne</td>
<td>rs1,reg_or_imm8</td>
<td>S-00001000</td>
<td>1 Branch if zero</td>
</tr>
<tr>
<td>call</td>
<td>label</td>
<td>S-0000000</td>
<td>1 Call</td>
</tr>
<tr>
<td>done</td>
<td>label</td>
<td>S-00001001</td>
<td>1 Skip trapped instruction</td>
</tr>
<tr>
<td>jmp</td>
<td>rs1,rd</td>
<td>S-00001000</td>
<td>2 Jump and link</td>
</tr>
<tr>
<td>null</td>
<td>label</td>
<td>S-010000000</td>
<td>2 Null operation</td>
</tr>
<tr>
<td>retry</td>
<td>label</td>
<td>S-00001100</td>
<td>2 Retry trapped instruction</td>
</tr>
<tr>
<td>return</td>
<td>rs1</td>
<td>pseudo-op</td>
<td>2 Return</td>
</tr>
<tr>
<td>sir</td>
<td>label</td>
<td>S-00001111</td>
<td>2 Software-initiated reset</td>
</tr>
<tr>
<td>softtrap</td>
<td>rs1,reg_or_imm14</td>
<td>S-0010000</td>
<td>2 Software-initiated trap</td>
</tr>
</tbody>
</table>
Memory Access Instructions

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.

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.

Most memory access instructions use a two-component effective address specification, denoted in this document by [address]. [address] may be [rs1+rs2], [rs1+ imm14], or [rs1]. When [rs1] is specified, the assembler infers an immediate zero for the second address component of the instruction.

### Pixel Instructions

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>argument list</th>
<th>opcode</th>
<th>L. operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cas</td>
<td>rs1, {rs2, rd}</td>
<td>S-11011110</td>
<td>? Compare and swap (atomic)</td>
</tr>
<tr>
<td>cstub</td>
<td>rd, rs1, {rs2}</td>
<td>S-11001000</td>
<td>? Conditional store byte</td>
</tr>
<tr>
<td>cstb</td>
<td>rd, rs1, {rs2}</td>
<td>S-11010101</td>
<td>? Conditional store short</td>
</tr>
<tr>
<td>csw</td>
<td>rd, rs1, {rs2}</td>
<td>S-11010110</td>
<td>? Conditional store word</td>
</tr>
<tr>
<td>lflush</td>
<td>no arguments</td>
<td>S-00010000</td>
<td>? Flush instruction pipe</td>
</tr>
<tr>
<td>ld</td>
<td>[address], rd</td>
<td>S-11000100</td>
<td>2 Load byte</td>
</tr>
<tr>
<td>ldph</td>
<td>[address], rd</td>
<td>S-11000011</td>
<td>? Load pair</td>
</tr>
<tr>
<td>ld</td>
<td>[address], rd</td>
<td>S-11001010</td>
<td>2 Load short</td>
</tr>
<tr>
<td>ldso</td>
<td>[address], rd</td>
<td>S-11010110</td>
<td>2 Load short other-endian</td>
</tr>
<tr>
<td>ldub</td>
<td>[address], rd</td>
<td>S-11000010</td>
<td>2 Load unsigned byte</td>
</tr>
<tr>
<td>ldus</td>
<td>[address], rd</td>
<td>S-11000001</td>
<td>2 Load unsigned short</td>
</tr>
<tr>
<td>ldso</td>
<td>[address], rd</td>
<td>S-11010110</td>
<td>2 Load unsigned short other-endian</td>
</tr>
<tr>
<td>ldow</td>
<td>[address], rd</td>
<td>S-11011010</td>
<td>2 Load unsigned word other-endian</td>
</tr>
<tr>
<td>ldw</td>
<td>[address], rd</td>
<td>S-11011010</td>
<td>2 Load word</td>
</tr>
<tr>
<td>ldwo</td>
<td>[address], rd</td>
<td>S-11011110</td>
<td>2 Load word other-endian</td>
</tr>
<tr>
<td>membar</td>
<td>no arguments</td>
<td>S-00110001</td>
<td>? Memory barrier</td>
</tr>
<tr>
<td>acldb</td>
<td>[address], rd</td>
<td>S-10000100</td>
<td>? Non-cacheable load byte</td>
</tr>
<tr>
<td>acldg</td>
<td>[address], rd</td>
<td>S-10000111</td>
<td>? Non-cacheable load group</td>
</tr>
<tr>
<td>aclds</td>
<td>[address], rd</td>
<td>S-10000101</td>
<td>? Non-cacheable load short</td>
</tr>
<tr>
<td>acldo</td>
<td>[address], rd</td>
<td>S-10011011</td>
<td>? Non-cacheable load short other-endian</td>
</tr>
<tr>
<td>acldb</td>
<td>[address], rd</td>
<td>S-10000000</td>
<td>? Non-cacheable load unsigned byte</td>
</tr>
<tr>
<td>acldu</td>
<td>[address], rd</td>
<td>S-10000001</td>
<td>? Non-cacheable load unsigned short</td>
</tr>
<tr>
<td>acldso</td>
<td>[address], rd</td>
<td>S-10010110</td>
<td>? Non-cacheable load signed short other-endian</td>
</tr>
<tr>
<td>aclduw</td>
<td>[address], rd</td>
<td>S-10100000</td>
<td>? Non-cacheable load unsigned word</td>
</tr>
<tr>
<td>aclduwo</td>
<td>[address], rd</td>
<td>S-10100100</td>
<td>? Non-cacheable load unsigned word other-endian</td>
</tr>
<tr>
<td>acldw</td>
<td>[address], rd</td>
<td>S-10000110</td>
<td>? Non-cacheable load word</td>
</tr>
<tr>
<td>acldwo</td>
<td>[address], rd</td>
<td>S-10010110</td>
<td>? Non-cacheable load word other-endian</td>
</tr>
<tr>
<td>acsb</td>
<td>rd, [address]</td>
<td>S-10101000</td>
<td>? Non-cacheable load word other-endian</td>
</tr>
<tr>
<td>nestp</td>
<td>rd, [address]</td>
<td>S-10100111</td>
<td>? Non-cacheable store pair</td>
</tr>
<tr>
<td>nCaps</td>
<td>rd, [address]</td>
<td>S-10101001</td>
<td>? Non-cacheable store short</td>
</tr>
<tr>
<td>acsso</td>
<td>rd, [address]</td>
<td>S-10111001</td>
<td>? Non-cacheable store short other-endian</td>
</tr>
<tr>
<td>nestw</td>
<td>rd, [address]</td>
<td>S-10100101</td>
<td>? Non-cacheable store word</td>
</tr>
<tr>
<td>nestwo</td>
<td>rd, [address]</td>
<td>S-10111010</td>
<td>? Non-cacheable store word other-endian</td>
</tr>
<tr>
<td>s2bd</td>
<td>rd, [address]</td>
<td>S-11110000</td>
<td>2 Store byte to instruction</td>
</tr>
<tr>
<td>sb</td>
<td>rd, [address]</td>
<td>S-11110000</td>
<td>1 Store byte</td>
</tr>
<tr>
<td>stp</td>
<td>rd, [address]</td>
<td>S-11110011</td>
<td>? Store pair</td>
</tr>
<tr>
<td>stor</td>
<td>rd, [address]</td>
<td>S-01110001</td>
<td>1 Store short</td>
</tr>
<tr>
<td>stso</td>
<td>rd, [address]</td>
<td>S-01111001</td>
<td>1 Store short other-endian</td>
</tr>
<tr>
<td>stw</td>
<td>rd, [address]</td>
<td>S-10100101</td>
<td>1 Store word</td>
</tr>
<tr>
<td>stw</td>
<td>rd, [address]</td>
<td>S-10111010</td>
<td>1 Store word other-endian</td>
</tr>
</tbody>
</table>

[still missing pefetch, ld_diag and st_diag, at least]
more as I learn more . . .

Note: This section is superseded by the “Code Scheduling Guidelines” 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.

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

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.

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.

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.

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.

Multiple writes to the same register of the register file in the 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:

<table>
<thead>
<tr>
<th>cycle operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Before being too awestruck by this tight scheduling, consider what happens if there is a 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.

add is an integer instruction that computes “if rs1[1]+rs2[2]” or “(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 [rd].

The suggested assembler syntax is:

```
add rs1, reg_or_imm14, rd
```

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

addc, subc

addc and subc are integer instructions that compute “if rs1[1]+rs2[2][rs3]” or “(rs1)[rs2][rs3]”, respectively. Only the least significant bit of [rs3], which is expected to be the result of a carry or borrow, is used. The result is left in [rd].

The suggested assembler syntax is:

```
addc rs1, rs2, rs3, rd
subc rs1, rs2, rs3, rd
```

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

and

and is a logical instruction that computes “if rs1[1]& rs2[2]” or “(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 [rd].

The suggested assembler syntax is:

```
and rs1, reg_or_imm14, rd
```

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

Bitext

bitext is a pixel instruction that extracts bits from the pair of registers rs1[1] and rs2[2]. The extracted field is described by a 6-bit length in bits 21 . . . 16 of [rs3]and a 5-bit skip count in bits 4 . . . 0 of [rs3]. The skip count is applied at the left-most (high-order) end of rs1[1]. The extracted field is right justified with [rd] without sign-extension.

(bitext is appropriated from someone else’s slide. I hope the stylistic change is not too jarring, but prose description has not been lucid for some readers.)

See FIG. 11

The suggested assembler syntax is:

```
bitext rs1, rs2, rs3, rd
```

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

budeck

budeck is a control flow instruction that causes a trap if “if rs1[1]=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 is trap is not yet defined.
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.

The suggested assembler syntax is:

```
bnz r1, reg_or_imm8, reg_or_imm8
```

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.

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

The suggested assembler syntax is:

```
bnz rd, label
bnz, pt rd, label
```

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

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.

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.

The suggested assembler syntax is:

```
call label
```

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

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 $r_{[rd]}$.

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

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

The suggested assembler syntax is:

```
cas rs1, [rs2], rd
```

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.

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 may be either a register or an immediate. In either case, only the low-order 5 bits of the skip-count are used. The count of clear bits is left in $r_{[rd]}$.

The suggested assembler syntax is:

```
cccb rs1, reg or_imm14, rd
```

cccb is an SFU operation that uses the SFU 2-source format.
clip is a floating point instruction that computes:

\[(rs1 > rs2? 1:0) \ll 1\]

\[(rs1 < -rs2? 1:0) \ll 2\]

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

The suggested assembler syntax is:

\[\text{clip } rs1, rs2, rs3, rd\]

clip is a UFU operation that uses the UFU 2-source format, but the variants of that format that allow immediates for the second and third source operand are not used.

\[\text{cmovnz}\]

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

The suggested assembler syntax is:

\[\text{cmovznz } rs1, \text{reg_or_imm14}, rd \quad \text{SFU}\]
\[\text{cmovznz } rs1, \text{reg_or_imm8}, rd \quad \text{UFU}\]

The SFU version of the cmovznz uses the SFU compute format. The UFU version of cmovznz is a pseudo-op for the cpicknz instruction with \(rd\) replicated in the \(rs3\) field.

\[\text{cmove}z\]

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

The suggested assembler syntax is:

\[\text{cmovez } rs1, \text{reg_or_imm14}, rd \quad \text{SFU}\]
\[\text{cmovez } rs1, \text{reg_or_imm8}, rd \quad \text{UFU}\]

The SFU version of cmovez uses the SFU compute format. The UFU version of cmovez is a pseudo-op for the cpicknz instruction with \(rd\) replicated in the \(rs3\) field.

\[\text{cmpeq}\]

cmpeq is an integer instruction that computes \("rs1==rs2\) or \("\text{sign_ext}(imm14)\)"). 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 \(rd\).

The suggested assembler syntax is:

\[\text{cmpeq } rs1, \text{reg_or_imm14}, rd\]

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

\[\text{cmplt}\]

cmplt is an integer instruction that computes \("rs1<rs2\) or \("\text{sign_ext}(imm14)\)"). 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 \(rd\).

The suggested assembler syntax is:

\[\text{cmplt } rs1, \text{reg_or_imm14}, rd\]

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

\[\text{cmpzl}\]

cmpzl is an integer instruction that computes \("\text{sign_ext}(imm)\)"). 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 \(rd\).

The suggested assembler syntax is:

\[\text{cmpzl } rs1, \text{reg_or_imm8}, rd\]

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

\[\text{cpicknz}\]

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

\[\text{cpickz}\]

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

The suggested assembler syntax is:

\[\text{cpickz } rs1, \text{reg_or_imm8}, \text{reg_or_imm8}, rd\]

\[\text{cpicknz } rs1, \text{reg_or_imm8}, \text{reg_or_imm8}, rd\]

\[\text{cpickz}\]

cpickz is an integer instruction that assigns its second source operand to \(rd\) if the first source operand, \(rs1\), is zero; otherwise, the third source operand is
assigned to [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:

```assembly
cpickZ rS1, 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 [rs1] is non-zero, store the value the register [rd] at the address in register [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:

```assembly
cStb rd, rs1, rs2
cSts rd, rs1, rs2
cstw rd, rs1, rs2
```

0289 cStb, cSts, and cstw 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:

```assembly
done
```

0293 done is a SFU operation that uses the SFU compute format, but is has no use for any operand.

0294 fadd
0295 fadd is a floating point instruction that computes “[rs1] + [rs2]”, where the values of the source operands are IEEE single-precision floating point numbers. The result is delivered in [rd].

0296 The suggested assembler syntax is:

```assembly
fadd rs1, rs2, rd
```

0297 fadd is a UFU operation that uses the UFU 2-source format.

0298 fcmpeq
0299 fcmpeq is a floating point instruction that compares the single-precision floating point operands in its source registers [rs1] and [rs2] for equality. The destination register [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.

0300 The suggested assembler syntax is:

```assembly
fcmpeq rs1, rs2, rd
```

0301 fcmpeq is a UFU operation that uses the UFU compute format.

0302 femple
0303 femple is a floating point instruction that compares the single-precision floating point operands in its source registers [rs1] and [rs2] for inequality. The destination register [rd] set set to the integer value 1 if the source operands are not equal and zero otherwise. If the value of either source operand is a NaN, the result is zero.

0304 The suggested assembler syntax is:

```assembly
femple rs1, rs2, rd
```

0305 femple is a SFU operation that uses the SFU compute format.

0306 fcmpit
0307 fcmpit is a floating point instruction that compares the single-precision floating point operands in its source registers [rs1] and [rs2] for inequality. The destination register [rd] set set to the integer value 1 if the single-precision floating point value in [rs1] is less than or equal to the single-precision floating point value in [rs2] and to zero otherwise. If the value of either source operand is a NaN, the result is zero.

0308 The suggested assembler syntax is:

```assembly
fcmpit rs1, rs2, rd
```

0309 fcmpit is a SFU operation that uses the SFU compute format.

0310 fdiv
0311 fdiv is a floating point instruction that computes “[rs1] / [rs2]”, where the value of the source operands are IEEE single-precision floating point numbers. The result is delivered in [rd].

0312 The suggested assembler syntax is:

```assembly
fdiv rs1, rs2, rd
```

0313 fdiv is a UFU operation that uses the UFU compute format.
0314] fix2flt is a convert instruction that converts a fixed point value in \( f_{rs1} \), with its binary point specified by the low-order 5 bits of \( f_{rs2} \) or imm14, to a single precision floating point result in \( f_{rd} \).

0317] The suggested assembler syntax is:

\[
\text{fix2flt} \quad rs1, reg \_ or \_ imm14, rd
\]

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

0319] flt2fix is a convert instruction that converts a single precision floating point value in \( f_{rs1} \) to a fixed point result in \( f_{rd} \) with the binary point as specified by the low-order 5 bits of \( f_{rs2} \) or imm14.

0321] The suggested assembler syntax is:

\[
\text{flt2fix} \quad rs1, reg \_ or \_ imm14, rd
\]

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

0323] fmul is a floating point instruction that computes \( "f_{rs1} \cdot f_{rs2}" \), where the values of the source operands are IEEE single-precision floating point numbers. The result is delivered in \( f_{rd} \).

0325] The suggested assembler syntax is:

\[
\text{fmul} \quad rs1, rs2, rd
\]

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

0327] fmuladd is a floating point instruction that computes \( "f_{rs1} \cdot f_{rs2} + f_{rs3}" \), where the values of the source operands are IEEE single-precision floating point numbers. The result is delivered in \( f_{rd} \).

0329] The suggested assembler syntax is:

\[
\text{fmuladd} \quad rs1, rs2, rs3, rd
\]

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

0331] fmulab is a floating point instruction that computes \( "f_{rs1} \cdot f_{rs2} - f_{rs3}" \), where the values of the source operands are IEEE single-precision floating point numbers. The result is delivered in \( f_{rd} \).

0333] The suggested assembler syntax is:

\[
\text{fmulab} \quad rs1, rs2, rs3, rd
\]

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

0335] frecsqrt is a floating point instruction that computes the reciprocal square root of the single-precision floating-point number in \( f_{rs1} \) and puts that result in \( f_{rd} \). What will this do with an argument less than or equal to zero?

0337] The suggested assembler syntax is:

\[
\text{frecesqrt} \quad rs1, rd
\]

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

0339] fsab is a floating point instruction that computes \( "f_{rs1} - f_{rs2}" \), where the values of the source operands are IEEE single-precision floating point numbers. The result is delivered in \( f_{rd} \).

0341] The suggested assembler syntax is:

\[
\text{fsab} \quad rs1, rs2, rd
\]

0342] fsab 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 \( f_{rd} \) 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 subsequent addc and subc operations.

0345] The suggested assembler syntax is:

\[
\text{genborrow, gencarry} \quad rs1, reg \_ or \_ imm14, rd
\]

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

0347] getir is an integer instruction that gets the value of the internal register the ordinal of which is its \( f_{rs1} \) operand and puts that value in the register specified by \( f_{rd} \).
The suggested assembler syntax is:

\[
\text{getir} \quad \text{rs1, rd}
\]

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.

\[
\text{idiv} \quad \text{rs1, rd}
\]

idiv is an integer instruction that computes \(\text{rs1} \div \text{rs2}\) or \(\text{rs1} \div \text{imm14}\). The use of an immediate for the second source operand sets the i-bit of the opcode. The result is left in \(\text{rd}\).

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

The suggested assembler syntax is:

\[
\text{idiv} \quad \text{rs1, reg_or_imm14, rd}
\]

idiv is an SFU operation that uses the SFU compute format.

iflush

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.

The suggested assembler syntax is:

\[
\text{iflush} \quad \text{rs1, reg_or_imm14}
\]

The suggested assembler syntax is:

\[
\text{idiv} \quad \text{rs1, reg_or_imm14, rd}
\]

Iflush is an SFU operation that uses the SFU compute format, but uses none of that format’s operands.

\[
\text{idiv} \quad \text{rs1, reg_or_imm14, rd}
\]

Jmpl

Jmpl is a control flow instruction that causes a register-indirect control transfer to the address in \(\text{rs1}\). The current value of %npc is left in \(\text{rd}\).

If the branch target is an entry-point that might also expect to be reached by a call instruction, a register other than \text{ip} is a poor choice for \(\text{rd}\).

The suggested assembler syntax is:

\[
\text{jmpl} \quad \text{rs1, rd}
\]

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

ldb, lds, ldw

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 \(\text{rd}\). 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.

\[
\text{ldb} \quad \text{address, rd}
\]

ldb, lds, and ldw are SFU operations that use the SFU memory format.

\[
\text{ldo} \quad \text{address, rd}
\]

ldo and ldwo are memory access instructions that load a 16-bit short or a 32-bit word from address into the destination register \(\text{rd}\) 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.

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.

The suggested assembler syntax is:

\[
\text{ldo} \quad \text{address, rd}
\]

ldo and ldwo are SFU operations that use the SFU memory format.

ldpair

Ldpair is a memory access instruction that performs a load into a pair of adjacent registers beginning at the register specified by \(\text{rd}\) from address. The use of an immediate for the second component of the address sets the i-bit of the opcode, \(\text{rd}\) must be an even-numbered register.

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

The suggested assembler syntax is:

\[
\text{ldpair} \quad \text{rd, address}
\]

Ldpair is an SFU operation that uses the SFU memory format.

ldub, ldus, lduw

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

0382 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 Cafe registers are 32-bits the behavior of ldwu and ldw is identical. Future extension is liable to change that, so appropriate consideration should be used when choosing between these instructions.

0385 The suggested assembler syntax is:

```
ldwb [address],rd
ldws [address],rd
ldlw [address],rd
```

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

0387 ldsu, ldwu
0388 lduso, and ldwu are memory access instructions that load an unsigned 16-bit short or an unsigned 32-bit word from address into the destination register [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 Cafe registers are 32-bits the behavior of ldwu and ldw is identical. Future extension is liable to change that, so appropriate consideration should be used when choosing between these instructions.

0391 The suggested assembler syntax is:

```
ldws [address],rd
ldwu [address],rd
```

lduso and ldwu 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:

```
membar
```

0396 membar is an SFU operation that uses the compute instruction format, but uses none of that format's operands.

0397 moveind

0398 moveind is an integer instruction that copies the content of its first source operand, [rs1], to the register indicated by the least significant eight bits of its second source register, [rs2].

0400 The suggested assembler syntax is:

```
moveind rs1, [rs2]
```

0400 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 “[rs1] * [rs2]” or “[rs1] sign_ext(imm1:4)” The use of an immediate for the second source operand sets the first bit of the instruction header. The result is left in [rd].

0405 The suggested assembler syntax is:

```
mul rs1, reg_or_imm14, rd
```

0406 mul is a UFU operation that uses the UFU 2-source format.

0407 mul is a UFU operation that uses the UFU 3-source format.

0408 muladd

0409 muladd is an integer instruction that computes “[rs1]*[rs2]+[s3]”, “([s1]*[s2]+sign ext(imm8))”, “([s1]*sign ext(imm8))”, “([s1]*sign ext(imm8))” and puts the result in [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:

```
muladdl rs1, reg_or_imm8, reg_or_imm8, rd
```

0411 muladdl is a UFU operation that uses the UFU 3-source format.

0412 mulsub

0413 mulsub is an integer instruction that computes “([s1]*[s2]+[s3])”, “([s1]*[s2]+sign ext(imm8))”, “([s1]*sign ext(imm8))”, “([s1]*sign ext(imm8))” and puts the result in [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:

```
mulsub rs1, reg_or_imm8, reg_or_imm8, rd
```

0415 mulsub is a UFU operation that uses the UFU 3-source format.

0416 mulsub is a UFU operation that uses the UFU 3-source format.

0417 nclbu, nclus, ncluw

0418 nclbu, nclus, and ncluw are memory access instructions that perform a non-cachable load of an 8-bit byte, a 16-bit short, or a 32-bit word from address into the destination register [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:

```
nclbu [address],s1
nclus [address],s1
ncluw [address],s1
```
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.

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

The suggested assembler syntax is:

```
ncldg [address], rd
```

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

Note that there is no complementary nestg instruction.

ncldg is an SFU operation that use the SFU memory format.

ncldso, ncldwo

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.

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.

The suggested assembler syntax is:

```
ncldso [address], rd
ccldwo [address], rd
```

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

ncldub, nclhus, nclduw

ncldub, nclhus, 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.

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.

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

ncldus and ncldwo 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.

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.

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

ncldsu, ncldusw

ncldsu and ncldusw are SFU operations that use the SFU memory format.

nestb, nests, nestw

nestb, nests, and nestw 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.

For nests and nestw 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.

The suggested assembler syntax is:

```
nestb rd, [address]
nests rd, [address]
nestw rd, [address]
```

ncstb, ncstso, ncstwo

ncstb, ncstso, and ncstwo are memory access instructions that perform a non-cacheable store an 8-bit byte, an unsigned 16-bit short, or an unsigned 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:

<table>
<thead>
<tr>
<th>ncstso</th>
<th>rd.[address]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ncstwo</td>
<td>rd.[address]</td>
</tr>
</tbody>
</table>

[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 uncacheable store of a pair of adjacent registers beginning at the register specified by [rd] to address. The use of an immediate for the second component of the address sets the i-bit of the opcode. [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:

| nop |

[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 [rs1], leaving the result is [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.

| not |

[0470] or

[0471] or is a logical instruction that computes “[rs1] [rs2]” or “[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 [rd].

[0472] The suggested assembler syntax is:

| or |

[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, [rs1] and [rs2], as two pair of unsigned 16-bit operands. Each 16-bit operand is shifted right by the amount of the lower-order 4 bits of the third source operand, [rs3].

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

[0478] The suggested assembler syntax is:

| pack |

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

[0480] padd, padds

[0481] padd and padds are integer operations that compute “[rs1] * [rs2]”, where each of the sources is treated as a pair of independent 16-bit quantities yielding a pair of independent 16-bit sums in [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 [rs1] and the two 16-bit sums are left in [rd].

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

[0483] The suggested assembler syntax is:

| padd |

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

[0485] pcmovnzs

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

\[
\text{pcmovenz } rS1, rs2, rd
\]

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

pcmovenz is an integer instruction that uses two 16-bit flags in rS1 to control whether the corresponding 16-bit fields of rs2 are copied to the same positions of rd. If rS1 is zero, the corresponding flag field of rS1 is set to one if “rs2<15:0>=sign_ext(imm14)” and zero otherwise, and the short in bits 15:0 of rd is set to one if “rs1<15:0><sign_ext(imm14)” and zero otherwise.

The suggested assembler syntax is:

\[
\text{pcmovenz } rS1, rs2, rd
\]

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

pcmovenz 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.

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

The suggested assembler syntax is:

\[
\text{pcmovenz } rS1, rs2, rd
\]

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

pcmovenz is an integer instruction that does a “compare less than” of 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.

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

The suggested assembler syntax is:

\[
\text{pcmovenz } rS1, rs2, rd
\]

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

pcmovenz is an integer instruction that treats each of its two source registers, rs1 and 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 rd.

The suggested assembler syntax is:

\[
\text{pcmovenz } rS1, rs2, rd
\]

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

pcmovenz 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.

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

The suggested assembler syntax is:

\[
\text{pcmovenz } rS1, rs2, rd
\]
The suggested assembler syntax is:

```
pmul rs1, reg_or_imm14, rd
```

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

```
pmuladd, pmuladds
```

pmuladd and pmuladd are integer instructions that compute 

\[ ((r[s1]) \cdot (r[s2]) + (r[s3])) \]

where each of the sources is treated as a pair of independent 16-bit quantities yielding 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.

```
ppower
```

ppower is a SFU operation that computes

\[ 16 \text{-bit} \times 16 \text{-bit} = 32 \text{-bit} \]

16-bit quantities yielding a pair of independent 16-bit powers in [r[rd]].

```
precsqrt rS1, rd
```

precsqrt is a SFU operation that computes a pair of independent 16-bit powers in [r[rd]]. By stipulation, zero to any power is zero.

```
pshill rs1, reg_or_imm14, rd
```

pshill is a UFU operation that uses the UFU 2-source format.

```
pshra rs1, reg_or_imm14, rd
```

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

```
pshirl rs1, reg_or_imm14, rd
```

pshirl is a UFU operation that uses the UFU 2-source format.
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:

\[
\text{psub } \text{rs1.reg or imm14,rd} \\
\text{psubs } \text{rs1,rs2,rd}
\]

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 “\(r[rs1] \% \text{sign ext}(\text{imm14})\)”. 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:

\[
\text{rem } \text{rs1.reg or imm14,rd}
\]

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 Cafe Microarchitecture specification for the complete description.

0558 The suggested assembler syntax is:

\[
\text{retry }
\]

0559 return

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

0561 return

0562 return is a control flow instruction that causes a register-indirect control transfer to the address in “\(r[rs1]\)”.

0563 The suggested assembler syntax is:

\[
\text{return } \text{rs1}
\]

0564 return is a pseudo-op. What it really means is “jmp \(r[rs1]+4, r0\)”.

0565 s2ib, s2 is, s2iw

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

0567 s2ib, s2is, and s2iw are memory access instruction that stores 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 Cafe instructions can be at any byte boundary, care must be taken in choosing the right instruction.

0569 The suggested assembler syntax is:

\[
s2ib \quad \text{rd}[\text{address}] \\
s2is \quad \text{rd}[\text{address}] \\
s2iw \quad \text{rd}[\text{address}]
\]

0570 $s2$: $[sb, s, w]$ 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:

\[
\text{sethi imm22,rd} \\
\text{sethi %hi(label),rd}
\]

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:

\[
\text{setir rs1,rd}
\]

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 “\(r[rs1] << \text{imm}\)”. 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.
The suggested assembler syntax is:

```
shill rS1,reg_or_imm,rd
```

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

```
shra rS1,reg_or_imm,rd
```

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.

```
shrl rS1,reg_or_imm,rd
```

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.

```
softtrap rs1.reg_or_imm14
```

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

```
stb, sts, stw
```

stb, sts, and stw are memory access instructions 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.

```
softtrap rs1.reg_or_imm14
```

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.

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

Si [s, w] are SFU operations that use the SFU memory format.

```
softtrap rs1.reg_or_imm14
```

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.

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

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.

```
softtrap rs1.reg_or_imm14
```

stso, stwo

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.

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

stsv, stsw are SFU operations that use the SFU compute format, but has no use for any of that format’s operands.
stso and stwo are SFU operations that use the SFU memory format.

stpair

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

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

The suggested assembler syntax is:

\begin{center}
\begin{tabular}{ll}
\texttt{stpair} & \texttt{rd[address]} \\
\end{tabular}
\end{center}

stpair is an SFU operation that uses the SFU memory format.

sub

sub is an integer instruction that computes \texttt{f[rs1]}-\texttt{f[rs2]} or \texttt{f[rs1]}-\texttt{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 result is left in \texttt{f[rd]}.

The suggested assembler syntax is:

\begin{center}
\begin{tabular}{ll}
\texttt{sub} & \texttt{rs1,reg_or_imm14,rd} \\
\end{tabular}
\end{center}

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

xor

xor is a logical instruction that computes \texttt{f[rs1]} \texttt{f[rs2]} or \texttt{f[rs1]} \texttt{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 \texttt{f[rd]}.

The suggested assembler syntax is:

\begin{center}
\begin{tabular}{ll}
\texttt{xor} & \texttt{rs1,reg_or_imm14,rd} \\
\end{tabular}
\end{center}

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.