TRIPS Processor Reference Manual

Robert McDonald Doug Burger Stephen W. Keckler
Karthikeyan Sankaralingam Ramadass Nagarajan

March 10, 2005 - Version 1.2

Tech Report TR-05-19
Department of Computer Sciences
The University of Texas at Austin

This document describes the TRIPS Processor, including its instruction set, register set, and general processing model. TRIPS is a novel, scalable, and low power architecture for future technologies.

Notices

Acknowledgements

This work is supported by DARPA / IPTO and the Air Force Research Laboratory under contracts F33615-01-C-1892 and F33615-03-C-4106.

Contact Information

Computer Architecture and Technology Laboratory
Department of Computer Sciences
1 University Station C0500
The University of Texas at Austin
Austin, TX 78712-0233
cart@cs.utexas.edu

Contents

Chapter 1 - Introduction

1.1 Overview

This document describes the TRIPS processor, including its instruction set, register set, and general processing model. It defines the interface between TRIPS hardware and TRIPS software and is intended to be used by both hardware and software developers.

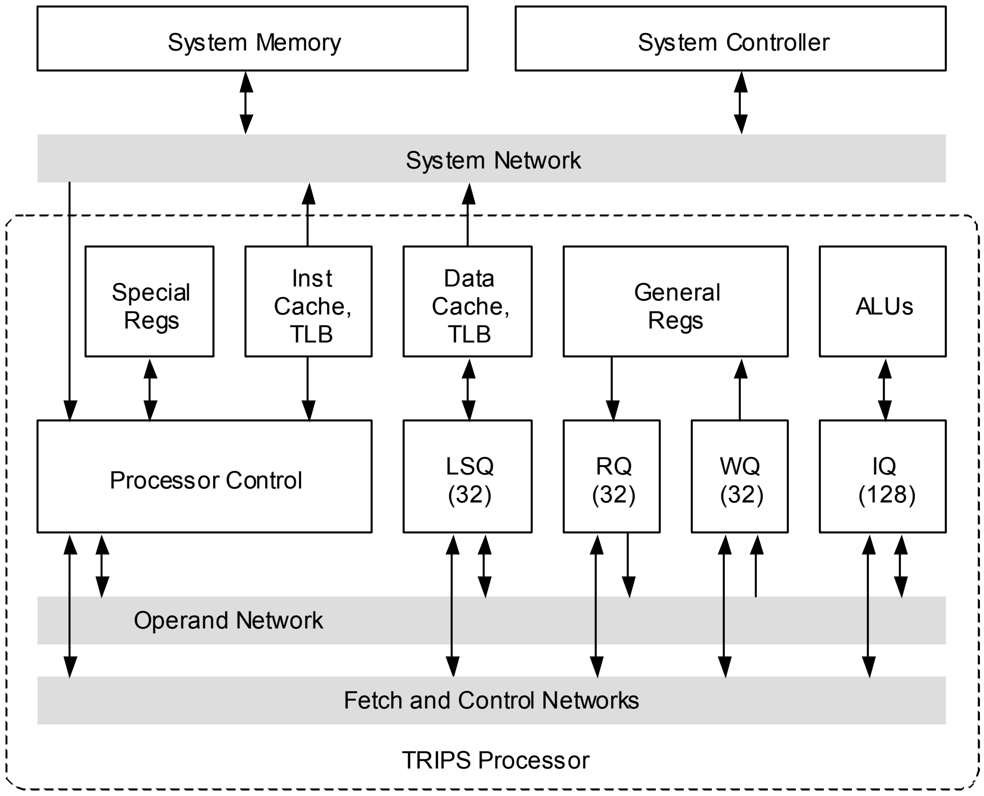
The TRIPS processor is part of a broader TRIPS chip and TRIPS system that will demonstrate several innovative techniques for building high-performance, highly-adaptive computing systems. The TRIPS architecture is the first member of a class of post-RISC architectures referred to as EDGE architectures. Key features include a block-atomic execution model and explicit support for data-driven (out-of-order) instruction execution.

The TRIPS processor is a prototype. It embodies many, but not all, of the ideas that have been developed as part of the TRIPS project. Some potential features have been omitted from the prototype in the interest of simplicity and a manageable development schedule.

Throughout this document, we sometimes distinguish between the processor architecture and the processor implementation. The term *architecture* refers to an abstracted view of the TRIPS processor and its software-visible resources. The term *implementation* refers to a more detailed view of the TRIPS processor and its internal organization. Most aspects of the implementation are beyond the scope of this document.

1.2 Processor Architecture

The following diagram shows an abstract representation of the TRIPS processor. Most elements of the architecture are based upon this abstract view of the processor.



Like a conventional processor, the TRIPS processor includes arithmetic units, caches, and registers. One or more arithmetic units (ALUs) are available for performing integer and floating-point operations. Separate caches are defined for holding instruction and data memory. TLBs are defined for translating virtual addresses to physical addresses. Registers are divided into two categories. General Registers may be used by a program to hold any type of data. Special Registers have predefined purposes and are used for configuration, control, and status.

Unlike a conventional processor, the TRIPS processor explicitly defines a set of internal queues. These queues are a fundamental part of the TRIPS instruction set and execution model. Architected queues enable efficient processing of whole blocks of instructions, rather than just individual instructions. An Instruction Queue (IQ) holds up to 128 executable instructions. A Read Queue (RQ) holds information for up to 32 pending General Register reads. A Write Queue (WQ) holds information for up to 32 pending General Register writes. A Load & Store Queue (LSQ) holds information for up to 32 pending memory loads and stores. Queues only hold transient execution state. Persistent program state is stored in registers, caches, and system memory.

The TRIPS processor connects to the rest of the system using a system interconnection network. The processor may access instructions and data stored in system memory. A system controller running system software may access registers within the processor. The system controller is responsible for configuring and servicing the processor.

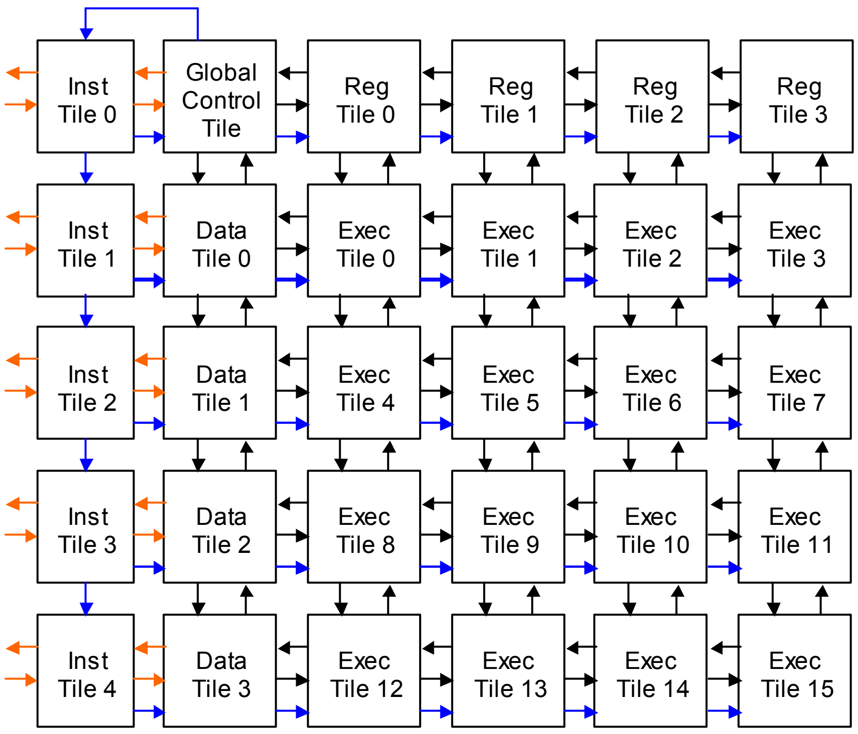
1.3 Processor Implementation

In order to fully understand the TRIPS architecture, it is useful to understand a little more about the processor implementation. TRIPS processors are partitioned into *tiles* that are placed adjacent to one another on a chip. These tiles are arranged in a *grid* (or 2D array) and connected by one or more light-weight networks, as depicted in the following diagram.

The five tile types are summarized below.

- *Execution Tiles* – These tiles hold the Instruction Queue and the arithmetic units.
- *Register Tiles* – These tiles hold the General Registers, the Read Queue, and the Write Queue.
- *Data Tiles* – These tiles hold the Data Cache, the Data TLB, and the Load & Store Queue.
- *Instruction Tiles* – These tiles hold the Instruction Cache and the Instruction TLB.
- *Global Control Tile* – This tile holds the Special Registers and the global processor control logic.

Most of the processor's resources are divided evenly into banks and distributed across multiple tiles. The Instruction Queue, for example, is divided into sixteen banks and distributed across sixteen Execution Tiles. Each tile can perform certain types of operations on every cycle. For example, on every cycle an Execution Tile can execute an instruction, transfer a previously-computed value to an adjacent tile, and receive a value from an adjacent tile. A high level of concurrency is possible.



■ Operand Network Links
🟧 On-Chip Network Links
🟦 Global Fetch Network Links

Some networks are not shown

This approach (referred to as a Grid Processor Architecture) fits well with future semiconductor technologies. It allows faster clock rates than conventional approaches, while also providing superior instruction-level parallelism and extensibility.

Another important aspect of the processor implementation is that it includes multiple copies of the architected queues and registers. The processor includes eight copies of each queue, allowing it to process up to 8 blocks and 1024 instructions (128 x 8) at once. The processor includes four copies of most registers, allowing it to process up to four threads at once. These capabilities will be described in subsequent chapters.

1.4 Document Organization

The remainder of this document is divided into the following chapters.

- Chapter 2 describes the processing model.
- Chapter 3 describes the program format.
- Chapter 4 describes the registers.
- Chapter 5 describes the queues.
- Chapter 6 describes the memory model.
- Chapter 7 describes the exception model.
- Chapter 8 describes the instruction set.
- Chapter 9 describes the performance monitor.

Chapter 2 - Processing Model

2.1 Processor States

A TRIPS processor will always be in one of the following two states.

- *Running* – The processor is executing one or more program threads.
- *Halted* – The processor has encountered an exceptional condition and is waiting to be serviced.

The processor state is reflected in a Processor Status Register (PSR) and also on a HALTED output signal.

When the processor is first powered on (or otherwise reset), it will enter the halted state. The processor will remain in the halted state until serviced by the system software (running on an external system controller). The system software may configure the processor by programming its registers and may restart the processor by clearing the PSR. This will cause the processor to enter the running state. The processor will remain in the running state until it encounters an exception condition (which includes external interrupts and system calls). See Chapter 7 for more information about exceptions.

2.2 Processor Modes

A TRIPS processor may operate in any of the following modes.

- *Default Mode (D-Morph)* – In this mode, the processor devotes all of its resources to executing a single program thread.
- *Multi-Threading Mode (T-Morph)* – In this mode, the processor divides or shares its resources for simultaneously executing up to four program threads.

A field in the Processor Control Register (PCR) determines the processor mode. (Other modes have also been proposed, but are not demonstrated in this prototype.)

2.3 Threads Slots

A TRIPS processor includes support for four *thread slots*. Thread slots are numbered starting from zero (thread slot 0, thread slot 1, etc). Each thread slot includes a set of registers used to hold and maintain program state for a single program thread as it executes. These registers are described in Chapter 4.

In its default mode (D-Morph), a TRIPS processor processes the thread loaded into thread slot 0 and other thread slots are ignored (they should be marked invalid). In its multi-threading mode (T-Morph), a TRIPS processor simultaneously processes threads from all four thread slots. The threads may belong to a single multi-threaded program or to separate programs.

2.4 Data Formats

The TRIPS architecture supports byte (8-bit), halfword (16-bit), word (32-bit), and doubleword (64-bit) data.

Some instructions interpret the data as a signed (2's complement) or unsigned integer. Some instructions interpret the data as an IEEE-754 compatible single-precision or double-precision floating-point number. Other instructions consider the data to be generic binary data.

Data stored in registers is always right-justified (stored in the least-significant bits). Data stored in memory is always aligned to a natural boundary (its address is always a multiple of its size). Data stored in memory is always stored in big-endian byte order.

2.5 Block-Atomic Execution

TRIPS programs execute in a block-atomic manner. Large groups of instructions (up to 128) are processed together as a block, rather than as individual instructions. This enables more efficient processing and helps expose more instruction-level parallelism.

A Program Counter (PC) holds the address of the block that is currently being processed (when the processor is running) or the block that will be processed next (when the processor is halted). For each block, the processor performs the following steps:

- *Block Fetch* – Fetch the block from memory (or cache) into the processors queues.
- *Block Execute* – Execute the instructions in dataflow order, computing and saving one or more results in the queues.
- *Block Commit* – Copy the results from the queues to the committed program state.

Each step is conditioned upon the success of the previous step. If an exception is discovered while fetching a block, the fetch is aborted. The block execute step only occurs if the fetch was successful. If an exception is discovered while executing a block, the execution is aborted. The block commit step only occurs if the execution was successful.

From a software perspective, blocks are never partially executed. If an exception (or interrupt) occurs, the processor will always stop the program thread on a precise block boundary. The intermediate results are not visible.

TRIPS processors preserve these simple sequential semantics at the block level, but are free to use techniques such as pipelining and speculative execution to improve performance. Our implementation does, in fact, use these techniques to overlap the processing of up to 8 blocks. A *branch predictor* is employed to predict which block will execute next. By default, the processor speculatively fetches and executes multiple blocks. Blocks are always committed non-speculatively and in program order.

2.6 Dataflow Execution

While a familiar sequential execution model is defined at the block level, a very different execution model is defined at the instruction level. Instructions inside a block execute in *dataflow order*, rather than in *program order*. Instruction dependences are explicitly encoded in the instructions. The TRIPS processor is capable of executing each instruction as soon as its operands (and the required execution resources) become available.

This model of execution can be better understood by looking at a simple example. Consider the following trivial bit of C code:

```
// assume x, y, and z are local variables
x = x * y;
y = y + 7;
z = x + 5;
```

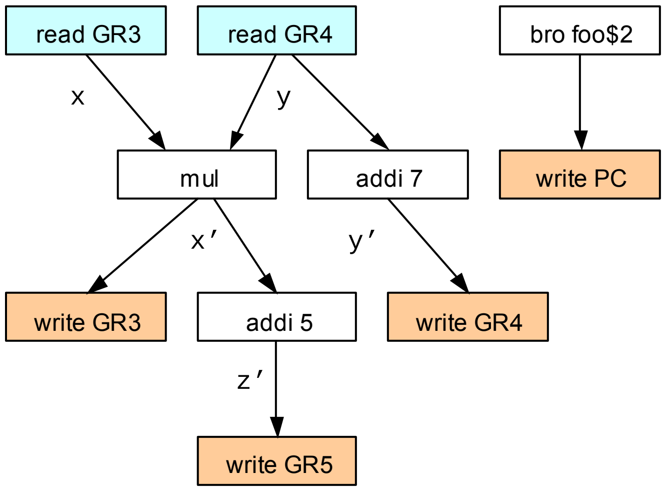
This code might be compiled to the following TRIPS Assembly Language (TASL).

```
.bbegin foo$1
  R[3] read G[3] N[2,0]        ; read x from GR3
  R[4] read G[4] N[2,1] N[1,0] ; read y from GR4

  N[2] mul N[3,0] W[3]         ; x' = x * y
  N[1] addi 7 W[4]             ; y' = y + 7
  N[3] addi 5 W[5]             ; z' = x' + 5
  N[0] bro foo$2               ; branch to foo$2

  W[3] write G[3]              ; write x' to GR3
  W[4] write G[4]              ; write y' to GR4
  W[5] write G[5]              ; write z' to GR5
.bbend
```

With this notation, each line beginning with “R” defines a General Register (GR) read instruction and a corresponding entry in the Read Queue (RQ). Each line beginning with “N” defines a regular instruction and a corresponding entry in the Instruction Queue (IQ). Each line beginning with “W” defines a GR write instruction and a corresponding entry in the Write Queue (WQ). Notice that the queue entries do not necessarily need to be assigned in order. (The complete TASL syntax is described in a separate document.)

Each instruction sends its result to one or more dependent target instructions. In some cases, the targets reside in the IQ. For example, "N[2,0]" refers to the OP0 slot of IQ entry 2. In other cases, the targets reside in the WQ. For example, "W[3]" refers to WQ entry 3. The branch instruction implicitly targets a Program Counter (PC).

The following diagram shows a corresponding dataflow graph. In this example, GR3 and GR4 are considered *block inputs*. GR3, GR4, GR5, and the PC are considered *block outputs*. The read instructions (shown in blue) retrieve the block inputs. The write instructions and implied PC write (shown in red) deliver the block outputs (at commit time). The regular instructions (shown in white) compute intermediate values.



Some instructions (like MUL) require two operands. Some instructions (like ADDI) require only one operand. Other instructions (like BRO) do not require any operands. Each instruction may execute as soon as all its required operands have been delivered. The precise order of execution is not defined by the architecture and not necessarily knowable at compile time.

Block execution is considered complete when all of the block outputs have been produced. As explained in the next section, it is sometimes possible to produce all of the outputs without executing all of the instructions.

2.7 Predicated Execution

Predication plays an important role in the TRIPS architecture. Predication allows a value to be selected from multiple potential values. It is essential for encoding conditional branches and for constructing large, efficient blocks from general-purpose programs. All control dependences inside of a block are represented using predication, rather than control flow.

Almost any regular instruction can be *predicated*. Almost any regular instruction can produce a *predicate*. Instructions may be predicated upon a *true* or a *false* predicate. Predicated instructions must receive all of their required operands plus a matching (true or false) predicate before they can execute. Instructions that do not receive a matching predicate will not execute (and will not deliver a result). Instructions that do not receive all required operands or predicates are considered *indirectly* predicated.

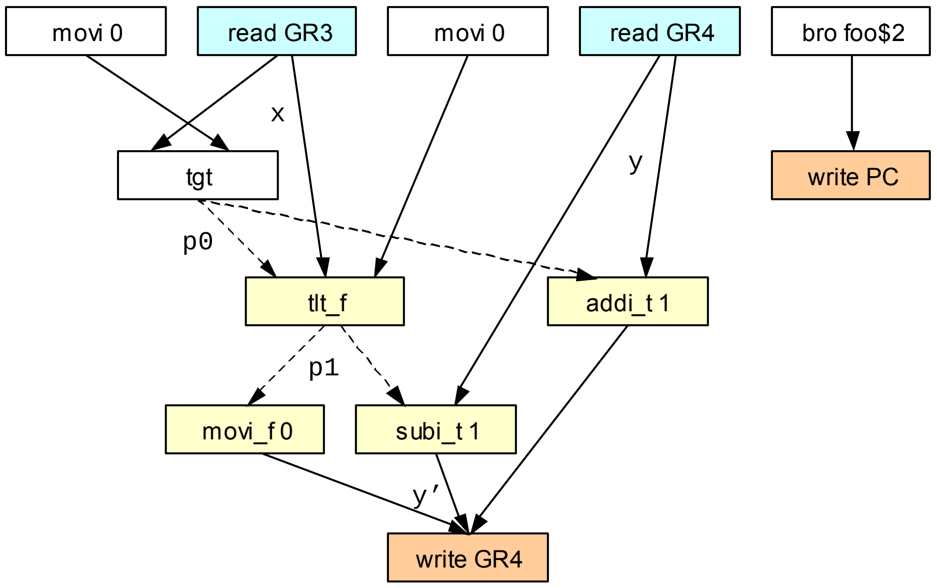
To better understand predication, let's look at a simple example. Consider the following bit of C code.

```
// assume x and y are local variables
if (x > 0) y++;
else if (x < 0) y--;
else y = 0;
```

This code might be compiled to the following TRIPS Assembly Language (TASL). Instruction names ending in "_t" and "_f" are predicated on true and false, respectively.

```
.bbegin foo$1
  R[3] read G[3] N[1,0] N[4,0] ; read x from GR3
  R[4] read G[4] N[2,0] N[5,0] ; read y from GR4

  N[0] movi 0 N[1,1]
  N[1] tgt N[2,P] N[4,P]       ; p0 = x > 0
  N[2] addi_t 1 W[4]           ; if (p0) y' = y + 1
  N[3] movi 0 N[4,1]
  N[4] tlt_f N[5,P] N[6,P]     ; if (!p0) p1 = x < 0
  N[5] subi_t 1 W[4]           ; if (p1) y' = y - 1
  N[6] movi_f 0 W[4]           ; if (!p1) y' = 0
  N[7] bro foo$2               ; branch to foo$2

  W[4] write G[4]              ; write y' to GR4
.bbend
```

The following diagram shows the corresponding dataflow graph. This graph uses the same conventions as the previous example. Normal lines represent operands, while dashed lines represent predicates. Predicated instructions are shown in yellow.



For this example, two predicates are computed and used to control each of the three possible assignments. The first predicate (p0) is based upon the result of the first test (x > 0). The second predicate (p1) is based upon the result of the second test (x < 0), which is only performed if the first predicate is false. Three different instructions (N[2], N[5], and N[6]) target W[4], but the predicates guarantee that only one will actually execute and deliver a result. For any possible outcome, only a subset of the instructions will execute.

Although the previous example is simple, a similar approach may be used to form larger and more complex predicated blocks (which are usually called *hyperblocks*).

From a compiler's (or hand-coder's) perspective, each predicate may be thought of as a 3-state variable. The possible states are *true*, *false*, and *undefined*. By default, each predicate begins in the undefined state. Any number of predicates may be defined, computed, and used inside of a block – but predicates do not exist outside of a block.

2.8 Output Nullification

When using predication, it is possible that some block outputs will only need to be produced under certain conditions. The TRIPS architecture supports a technique called *output nullification* that allows individual writes or stores to be cancelled. When a write or store instruction is nullified, the write or store is not performed when the block execution is committed.

Output nullification is similar to but different than instruction predication. In order for block execution to be considered complete, a value must be produced for every potential block output. In most cases, a normal value is delivered. But in some cases, a special value may be delivered to nullify the output. Null values are produced by NULL instructions and represented as null tokens (described in the next section).

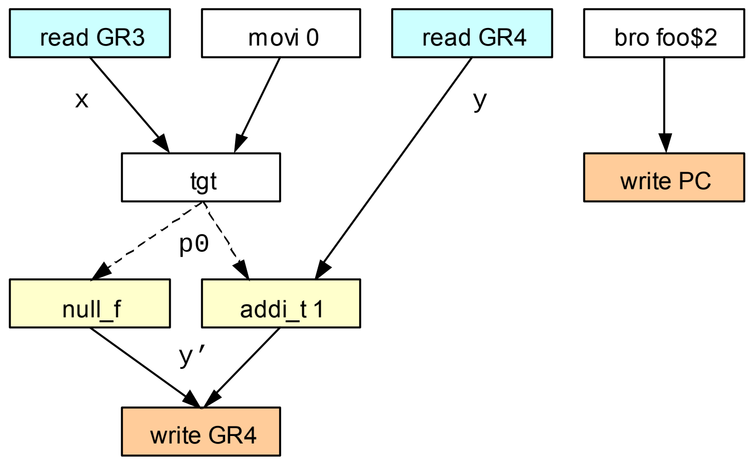
To better understand nullification, let's look at a simple example. Consider the following bit of C code.

```
// assume x and y are local variables
if (x > 0) y++;
```

This code might be compiled to the following TRIPS Assembly Language (TASL).

```
.bbegin foo$1
  R[3] read G[3] N[1,0]         ; read x from GR3
  R[4] read G[4] N[2,0]         ; read y from GR4

  N[0] movi 0 N[1,1]
  N[1] tgt N[2,P] N[3,P]        ; p0 = x > 0
  N[2] addi_t 1 W[4]            ; if (p0) y' = y + 1
  N[3] null_f W[4]              ; if (!p0) y' = NULL
  N[7] bro foo$2                ; branch to foo$2

  W[4] write G[4]               ; write y' to GR4
.bbend
```

And here is a corresponding dataflow graph.



Based upon the predicate, variable y is either incremented or left unchanged. A similar approach may be used to nullify stores. In that case, the target of the NULL instruction is a store instruction instead of a write instruction.

2.9 Dataflow Tokens

The TRIPS processor uses *dataflow tokens* to represent values being delivered from one instruction to another as a block executes. Each arrow in the previous dataflow diagrams represents a dataflow token. These tokens usually communicate data, but must sometimes communicate nullifications or exceptions.

Each dataflow token includes a type field, which must be one of the following:

- Normal Token
- Null Token
- Exception Token

Most instructions generate normal tokens when executed. The NULL instruction generates null tokens. Null tokens that arrive at a write or store instruction result in a nullified block output (as described in section 2.8). Instructions that encounter exceptions as they execute generate exception tokens. Exception tokens that arrive at a write or store instruction result in a block execute exception.

When an instruction receives a null token or an exception token, it usually propagates a similar token to each of its targets. Because some instructions receive multiple tokens as operands and predicates, a set of policies are needed to define the expected result for every possible combination of tokens. These policies are defined in Chapter 5.

With predication, it is possible for instructions to generate exception tokens that never reach a write or store instruction. For these cases, the exception is considered a false exception and should not be reported. To ensure that all real exceptions are reported and that no false exceptions are reported, predicated code must preserve the program's original control dependences.

2.10 Load Dependences

Although most dependences are known at compile time and explicitly encoded in the instructions, load dependences present a special challenge. Because load and store addresses are computed at run time, it is difficult to know whether a load and a store will be accessing the same location in memory. Loads may sometimes be dependent upon prior stores in the same block. When a dependence does exist, the processor must ensure that the correct value is forwarded from the store to the load. This is called *store forwarding*.

The architecture allows instructions to execute in dataflow order, rather than program order – this is true even for load and store instructions. The processor continuously checks for load dependence violations (using the LSQ). As long as each load executes after all stores that it depends upon, store forwarding will occur as needed and everything is fine. If a load happens to execute before a store that it depends upon, the processor must take special action to enforce the correct load and store ordering. This usually involves re-executing the block.

Our processor implementation employs a *load dependence predictor* to dynamically predict potential load dependences in a program. For each load executed, the processor predicts whether it will be dependent upon a not-yet-executed store. If a dependence is predicted, the processor will defer execution of the load until all prior stores have executed.

Some information about the original program order must be maintained to allow the processor to recognize load dependences. The LSQ entries must be assigned in such a way that each load has a greater index than all prior stores. The LSQ index is also referred to as a Load Store Identifier (LSID) throughout this document. See Chapter 5 for more information about the LSQ.

2.11 Execution Flags

The TRIPS processor defines a set *execution flags* that control various aspects of block execution. Separate copies of these flags exist in the Processor Control Register (PCR) and in each block's header. Both copies are combined to produce the actual execution flags for each particular program block.

The name and function of each execution flag is described below:

- *Inhibit Branch Predictor* – When this flag is set, the processor's branch predictor will not be used to predict the next block. Instead, the processor will wait for the block to execute a branch instruction and then use the result to control subsequent speculation.

- *Inhibit Load Predictor* – When this flag is set, the processor's load predictor will not be used to predict dependences for loads in the block. Instead, the processor will defer execution of each load in the block until all prior stores have executed.
- *Block Synchronization Required* – When this flag is set, the processor will synchronize the execution of the block with all prior and subsequent blocks from the same thread. The processor enforces a synchronization "barrier" just before and just after the block. The synchronized block will not be allowed to execute until all prior blocks have executed and their results have been committed. This includes synchronization of all external stores in the system. Similarly, subsequent blocks will not be allowed to execute until the synchronized block has been executed and all of its results have been committed throughout the system.
- *Thread Synchronization Required* – When this flag is set, the processor will synchronize the execution of the block with similarly flagged blocks from other active threads. The processor guarantees that it will not overlap execution for these "exclusive" blocks. It will execute and commit one of these blocks before allowing another of these blocks to execute.
- *Break Before Block* – When this flag is set, the processor will report a Breakpoint Exception just prior to executing the block. This flag is used to support debugger breakpoints.
- *Break After Block* – When this flag is set, the processor will report a Breakpoint Exception just after executing and committing the block. This flag is used to support debugger single-stepping.

Chapter 3 - Programs

3.1 Program Format

Each TRIPS program consists of one or more *program blocks* (or just “blocks” for short). Each block may contain multiple instructions that are to be executed together. TRIPS programs are executed in a block-atomic manner – one block at a time rather than one instruction at a time. Based upon the program characteristics and the compiler techniques applied, TRIPS blocks may be formed from conventional *basic blocks*, *superblocks,* or *hyperblocks* (these are compiler terms whose definition is beyond the scope of this document).

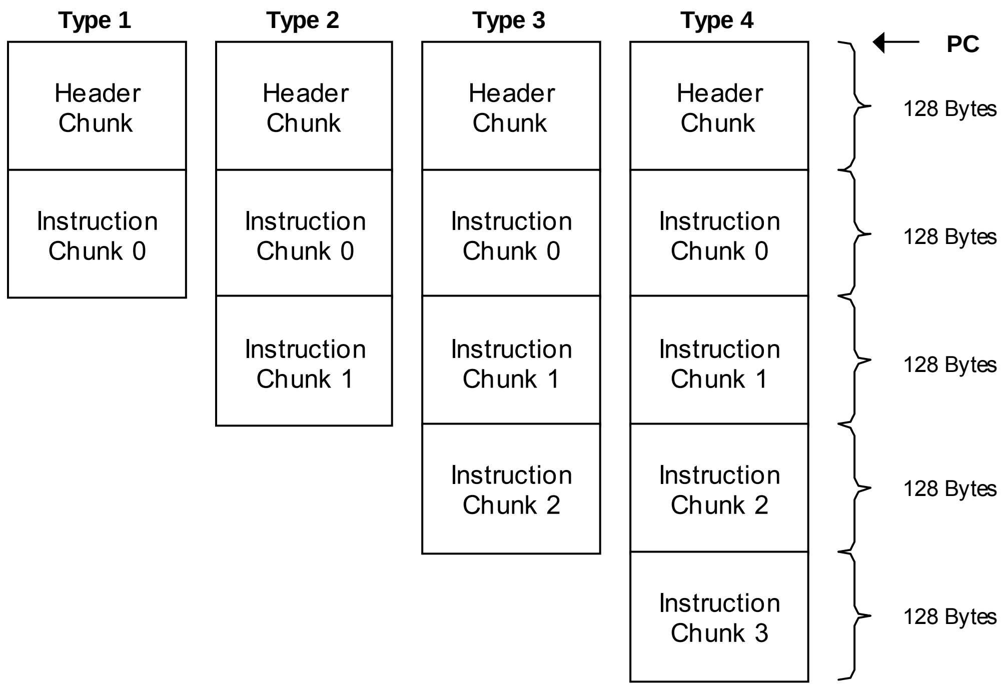
During execution, a *Program Counter (PC)* is maintained that always holds the base address of the currently executing program block.

Each TRIPS program block consists of one or more *program chunks*. Chunks are always 128 bytes in size and aligned to a 128-byte boundary in memory. Therefore, each program block must also be aligned to a 128-byte boundary in memory.

Most programs will consist of multiple program blocks. These blocks will usually be placed adjacent to one another in memory, but no specific sequence or placement of blocks is required. In the TRIPS architecture, there is no notion of sequential control flow. Every block must include a branch instruction that explicitly updates the PC, transferring control from one block to the next.

3.2 Block Format

The following diagram illustrates the supported formats for a TRIPS program block. Each block consists of a *header chunk*, followed by one to four *instruction chunks*. The header chunk holds a block header, up to 32 read instructions, and up to 32 write instructions. The instruction chunks each hold 32 regular instructions (up to 128 per block).



TRIPS Program Block Formats

3.3 Header Chunk Format

The following diagram describes the format of a header chunk. Each header chunk consists of 32 consecutive words (each 32 bits wide). Each word includes the following:

- One 4-bit header nibble
- One 22-bit read instruction
- One 6-bit write instruction

Bit Offsets

Byte Offsets	31	27 6	5 0
0	H0	Read 0	Write 0
4	H1	Read 1	Write 1
8	H2	Read 2	Write 2
12	H3	Read 3	Write 3
16	H4	Read 4	Write 4
20	H5	Read 5	Write 5
	...	...	...
104	H26	Read 26	Write 26
108	H27	Read 27	Write 27
112	H28	Read 28	Write 28
116	H29	Read 29	Write 29
120	H30	Read 30	Write 30
124	H31	Read 31	Write 31

Header Chunk Format

Each header chunk includes a block header that provides general information about the block. The block header is assembled from multiple 4-bit nibbles. These nibbles are concatenated together to form a 128-bit header. Nibbles are stored in a big-endian order (nibble H0 holds the most significant bits and nibble 31 holds the least significant bits).

The 128-bit block header includes multiple fields. Its format is described in the following diagram.

127 120	119 112	111 104	103 64
MARK	TYPE	XFLAGS	-

63 32	31 0
SMASK	-

These fields are described in the following table.

Bits	Field	Description
127:120	MARK	Header Mark – This field helps to distinguish header chunks from other chunks. It should always hold the value 0xFF.
119:112	TYPE	Block Type – This field identifies the type of program block. The following block types are allowed. 0 – Default Block (same as type 4) 1 – 32 Inst Block (1 Inst Chunk) 2 – 64 Inst Block (2 Inst Chunks) 3 – 96 Inst Block (3 Inst Chunks) 4 – 128 Inst Block (4 Inst Chunks)
111:104	XFLAGS	Block Execution Flags – This field holds one or more special execution flags. These flags indicate special execution requirements.
103:64	-	Reserved
63:32	SMASK	Store Mask – This field indicates which of the 32 LSQ entries (if any) have been assigned to stores. The mask bits are numbered from 31 down to 0 (with bit *i* corresponding to LSID *i*).
31:0	-	Reserved

A *Header Mark* is used to help distinguish header chunks from other program chunks. All header chunks should include the mark 0xFF. Attempts to fetch a program block without a valid header mark will result in a Fetch Exception.

The *Block Type* is used to determine the size of the program block. Attempts to fetch a program block with an illegal block type will result in a Fetch Exception. The legal block types are defined in the table above.

Block Execution Flags are included in the header and control certain aspects of block execution. Normally these flags are all clear (zeroed), but in some cases they will need to be set to ensure proper program execution. The XFLAGS field holds eight bits, numbered from 7 (msb) down to 0 (lsb). The following individual flags are defined. Each flag's function was described in Chapter 2.

Flag Bit	Description
XFLAGS[0]	Inhibit Branch Predictor
XFLAGS[1]	Inhibit Load Predictor
XFLAGS[2]	Block Synchronization Required
XFLAGS[3]	Thread Synchronization Required
XFLAGS[4]	Break Before Block
XFLAGS[5]	Break After Block
XFLAGS[6]	Reserved
XFLAGS[7]	Reserved

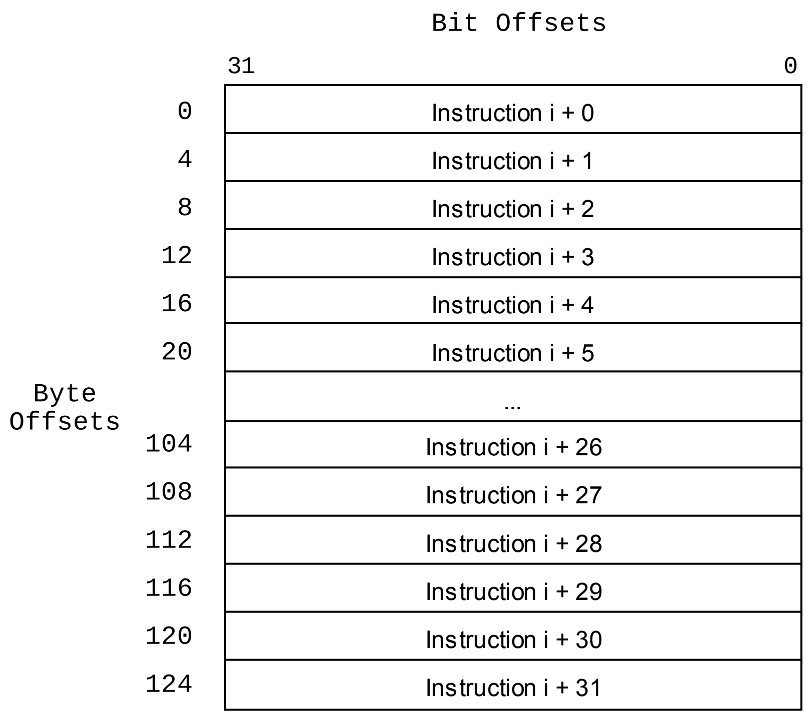
The *Store Mask* identifies individual LSQ entries that have been assigned for stores. Information must be received for each of these stores before the block is allowed to complete. See Chapter 5 for more information about the LSQ.

The read and write instructions within a header chunk allow access to a program's General Registers. Read instructions are numbered in order from 0 to 31. Write instructions are also numbered in order from 0 to 31. These numbers are referred to, respectively, as *Read IDs* and *Write IDs*. They correspond to positions within the Read Queue and Write Queue. See Chapter 5 for more information about the queues.

3.4 Instruction Chunk Format

The following diagram describes the format of an instruction chunk. Each instruction chunk consists of 32 words (each 32 bits wide). Each word holds a 32-bit instruction.

Instructions are numbered in order, beginning with instruction 0. The first instruction chunk holds instructions 0 through 31, the second holds instructions 32 through 63, the third holds instructions 64 through 95, and the fourth holds instructions 96 through 127. These numbers are referred to as *Instruction IDs* and correspond to positions within the Instruction Queue.



Instruction Chunk Format

3.5 Block Capacity Restrictions

The specified block format, along with other constraints in the architecture, lead to a number of capacity restrictions. These restrictions are summarized here.

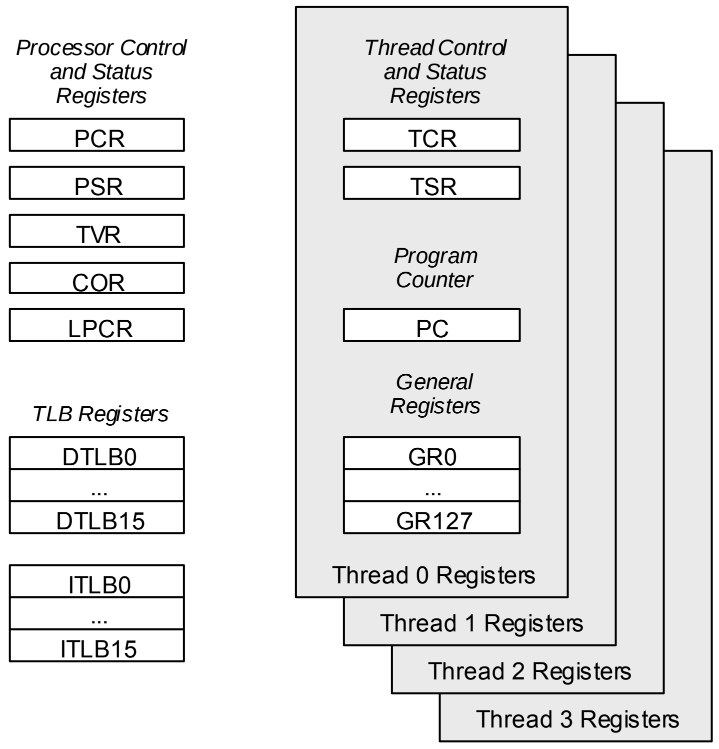
- Each block may hold up to 32 read instructions (8 per bank)
- Each block may hold up to 32 write instructions (8 per bank)
- Each block may hold up to 128 regular instructions
- Each block may hold up to 32 total loads and stores
- Each block may hold up to 8 total branches (or unique exits)

Chapter 4 - Registers

4.1 Register Set

4.1.1 Register Summary

The following diagram summarizes the TRIPS Processor register set. Some registers are associated with the entire processor, while others are associated with individual processor thread slots. Registers associated with individual thread slots are replicated for each supported thread.



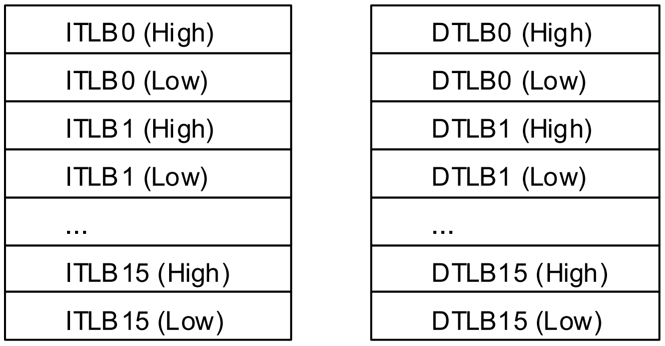
4.1.2 Processor Control and Status Registers

A few internal registers are defined for configuring and controlling the processor. A *Processor Control Register (PCR)* is defined for configuring the TRIPS Processor. A *Processor Status Register (PSR)* provides processor-level status information. The *Timeout Value Register (TVR)* is used to specify a timeout limit. The *Cache Operation Register (COR)* is used to initiate cache flush operations. The *Load Predictor Control Register (LPCR)* is used to configure the load predictor. See section 4.2 for descriptions of each control and status register.

4.1.3 TLB Registers

The TRIPS Processor includes two sets of *Translation-Lookaside Buffer* (TLB) registers. A set of sixteen *ITLB Registers* are used to describe memory segments that the processor may fetch from. A set of sixteen *DTLB Registers* are used to describe memory segments that the processor may load from or store to. Each TLB register is 16-bytes wide (formed from two 8-byte doublewords). This is illustrated in the following diagram.

See section 4.2 for descriptions of the TLB registers. See section 6.3 for an overview of the TLBs.



ITLB *ITLB*

4.1.4 Thread Control and Status Registers

Several internal registers are defined for configuring and controlling individual processor thread slots. A set of *Thread Control Registers (TCRs)* allow each thread slot to be configured. A set of *Thread Status Registers (TSRs)* provide thread-level status information. See section 4.2 for descriptions of each control and status register.

4.1.5 Program Counters

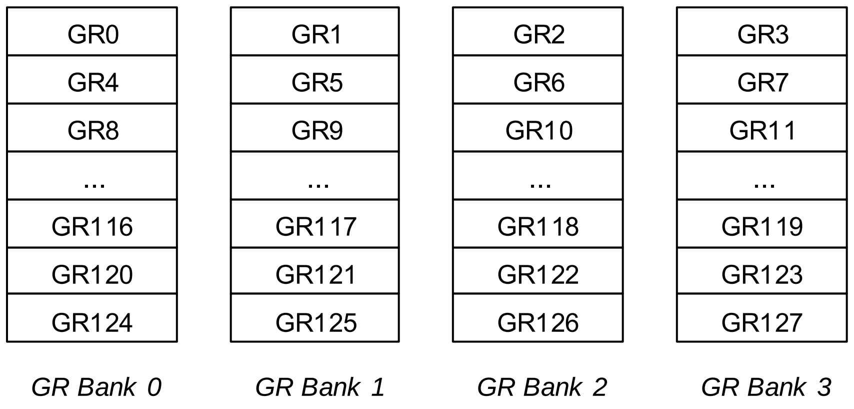
A set of *Program Counters (PCs)* are defined that maintain the current program address for each active thread. These are actually just special registers (not counters). Each PC may be explicitly loaded by software (when the processor is halted) and is also updated during thread execution as a thread transitions from one program block to another.

4.1.6 General Registers

The TRIPS Processor architecture defines 128 *General Registers (GRs)* for each processor thread slot. Each of these registers may be used to store a 64-bit value (which can be an integer, a floating-point number, or any other generic value). For each thread slot, the General Registers are numbered from 0 to 127 and referred to as GR0 – GR127. These registers may be used for any purpose (but software is expected to define register usage conventions).

Each thread's General Register set is divided into four register *banks*. Each bank holds 32 General Registers. (This banking technique makes it much easier for the processor to support multiple concurrent General Register accesses and speculative execution.)

General Registers are distributed across register banks so that bank 0 holds GR0, GR4, GR8, etc. Bank 1 holds GR1, GR5, GR9, etc. In general, register bank `i` holds General Registers `i + 4*j` (where `j` ranges from 0 to 31). This is illustrated in the following diagram.



General Registers may be read and written by an external controller when the processor is halted. This allows an external controller to perform context switches, handle system calls, etc.

General Registers may also be read and written by a program thread during execution. Of course, each thread can only access its own copy of the General Registers. See Chapter 5 for more information about General Register reads and writes during execution.

4.1.7 Performance Monitor Registers

Several additional registers are associated with a built-in Performance Monitor. These special registers are not defined in this chapter. See Chapter 9 for a full description of the Performance Monitor and its registers.

4.2 Register Map

Each TRIPS Processor provides memory-mapped access to its internal registers. A configuration port on the TRIPS Processor allows an external controller to read or write the registers (only when the processor is halted). A 64 KB window of address space may be used to map the following registers. (Some registers are grouped for convenience. All other locations within the 64 KB window are reserved.)

Offset	Bytes	Registers
0x0008	4	Processor Control Register (PCR)
0x000C	4	Processor Status Register (PSR)
0x0014	4	Timeout Value Register (TVR)
0x0018	4	Cache Operation Register (COR)
0x001C	4	Load Predictor Control Register (LPCR)
0x0400	4	Thread 0 Thread Control Register (TCR)
0x0404	4	Thread 0 Thread Status Register (TSR)
0x0408	8	Thread 0 Program Counter (PC)
0x0440	4	Thread 1 Thread Control Register (TCR)
0x0444	4	Thread 1 Thread Status Register (TSR)
0x0448	8	Thread 1 Program Counter (PC)
0x0480	4	Thread 2 Thread Control Register (TCR)
0x0484	4	Thread 2 Thread Status Register (TSR)
0x0488	8	Thread 2 Program Counter (PC)
0x04C0	4	Thread 3 Thread Control Register (TCR)
0x04C4	4	Thread 3 Thread Status Register (TSR)
0x04C8	8	Thread 3 Program Counter (PC)
0x0800	256	ITLB Registers (ITLB0 – ITLB15)
0x0C00	256	DTLB Registers (DTLB0 – DTLB15)
0x1000	1024	Thread 0 General Registers (GR0 – GR127)
0x1400	1024	Thread 1 General Registers (GR0 – GR127)
0x1800	1024	Thread 2 General Registers (GR0 – GR127)
0x1C00	1024	Thread 3 General Registers (GR0 – GR127)
0x2000	(512)	Performance Monitor Registers

External accesses must always read or write 4 or 8 bytes at a time. Registers that are 4 bytes wide must be accessed as 4 bytes. Registers that are 8 bytes wide must be accessed as 8 bytes. Registers that are 16 bytes wide require two 8-byte accesses. All accesses must use naturally-aligned addresses.

4.3 Register Descriptions

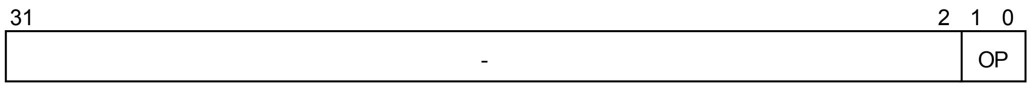
4.3.1 Alphabetical Register List

The rest of this chapter includes individual register descriptions, listed in alphabetical order.

COR

Cache Operation Register

Format:



Description:

The Cache Operation Register (COR) is used to initiate special cache operations. In order to initiate a cache operation, the processor must first be halted. Writing a non-zero value into the OP field will initiate a cache flush operation. When the cache operation is complete, the processor will automatically clear the OP field.

Fields:

The following table describes each field of the COR.

Bits	Field	Description
1:0	OP	Operation Status: 0 – Idle 1 – Inst Cache Flush 2 – Data Cache Flush
31:2	-	Reserved

Notes:

All COR fields are cleared (zeroed) during reset.

DTLB0 – DTLB15

Data TLB Registers

Format:

127 104	103 80	79 68	67	66	65	64
-	PHYSICAL BASE	-	M	C	W	R

63 40	39 16	15 12	11 8	7 2	1	0
-	VIRTUAL BASE	-	ASID	SIZE	-	V

Description:

The Data TLB Registers (DTLBs) define attributes and address translations for up to sixteen memory segments. Each register includes 128-bits of information (accessible as a high 64-bits and a low 64-bits). See section 6.3 for more information about TLBs.

Fields:

The following table describes each field of a DTLB register.

Bits	Field	Description
0	V	Valid Bit – Identifies whether the TLB register contains valid information. 0 – Invalid 1 – Valid
1	-	Reserved
7:2	SIZE	Segment Size – This field describes the segment size. The minimum size is 64 KB. The size is computed as 64 * 2^{SIZE} KB. Here are some sample encodings. 0 – 64 KB 1 – 128 KB 4 – 1 MB 24 – 1 TB

Bits	Field	Description
11:8	ASID	Address Space Identifier – This field determines which virtual address space may use this TLB entry.
15:12	-	Reserved
39:16	VIRTUAL BASE	Virtual Base Address – Specifies the virtual base address of the segment. Depending upon the segment size, some of the lower bits may be ignored.
63:40	-	Reserved
64	R	Readable – Specifies whether bytes within this segment may be read.
65	W	Writeable – Specifies whether bytes within this segment may be written.
66	C	Cacheable – Specifies whether bytes within this segment may be cached.
67	M	Mergeable – Specifies whether individual loads and stores may be merged and issued to the system as larger loads and stores.
79:68	-	Reserved
103:80	PHYSICAL BASE	Physical Base Address – Specifies the physical base address of the segment. Depending upon the segment size, some of the lower bits may be ignored.
127:104	-	Reserved

Notes:

All DTLB fields are cleared (zeroed) during reset.

ITLB0 – ITLB15

Instruction TLB Registers

Format:

127 104	103 80	79 67	66	65	64
-	PHYSICAL BASE	-	C	-	X

63 40	39 16	15 12	11 8	7 2	1	0
-	VIRTUAL BASE	-	ASID	SIZE	-	V

Description:

The Instruction TLB Registers (ITLBs) define attributes and address translations for up to sixteen segments. Each register includes 128-bits of information (accessible as a high 64-bits and a low 64-bits). See section 6.3 for more information about TLBs.

Fields:

The following table describes each field of an ITLB register.

Bits	Field	Description
0	V	Valid Bit – Identifies whether the TLB register contains valid information. 0 – Invalid 1 – Valid
1	-	Reserved
7:2	SIZE	Segment Size – This field describes the segment size. The minimum size is 64 KB. The size is computed as 64 * 2^{SIZE} KB. Here are some sample encodings. 0 – 64 KB 1 – 128 KB 4 – 1 MB 24 – 1 TB

Bits	Field	Description
11:8	ASID	Address Space Identifier – This field determines which virtual address space may use this TLB entry.
15:12	-	Reserved
39:16	VIRTUAL BASE	Virtual Base Address – Specifies the virtual base address of the segment. Depending upon the segment size, some of the lower bits may be ignored.
63:40	-	Reserved
64	X	Executable – Specifies whether blocks within this segment may be executed.
65	-	Reserved
66	C	Cacheable – Specifies whether bytes within this segment may be cached.
79:65	-	Reserved
103:80	PHYSICAL BASE	Physical Base Address – Specifies the physical base address of the segment. Depending upon the segment size, some of the lower bits may be ignored.
127:104	-	Reserved

Notes:

All ITLB fields are cleared (zeroed) during reset.

LPCR

Load Predictor Control Register

Format:

31 8	7 1	0
RI	-	M

Description:

The Load Predictor Control Register (LPCR) is used to configure the processor’s load predictor.

Fields:

The following table describes each field of the LPCR.

Bits	Field	Description
0	M	Predictor Mode 0 – Always predict no dependence 1 – Use the predictor history table
7:1	-	Reserved
31:8	RI	Reset Interval – Specifies the number of blocks that should be allowed to commit before resetting the load predictor’s history. The reset interval is computed by multiplying this field’s value by 256. Each time a block commits, the processor increments a committed block count and compares the results to the reset interval. If the values are equal, the load predictor and committed block count are both reset. A zero in this field is interpreted as an interval of 2^{32} blocks.

Notes:

The LPCR is cleared (zeroed) during reset.

PCs

Program Counters

Format:

63 40	39 0
-	ADDR

Description:

The TRIPS processor includes four *Program Counters (PCs)*. Each PC is associated with an independent processor thread slot. These are similar to conventional program counters. Each PC is used to hold the virtual address of the currently-executing program block.

The address stored in a PC must always be aligned to a 128-byte boundary (the lower seven bits must always be zero). Otherwise, a Fetch Exception will occur when a fetch is attempted.

During execution, the PC may be read using a MFPC instruction and written (or modified) using a branch instruction.

Fields:

The following table describes each field of a PC.

Bits	Field	Description
63:40	-	Reserved
39:0	ADDR	Address

Notes:

Each PC is cleared (zeroed) during reset.

PCR

Processor Control Register

Format:

31 16	15 8	7 6 1	0
-	XFLAGS	-	M

Description:

The *Processor Control Register (PCR)* is used to configure the TRIPS processor. It controls the overall mode of the processor and includes a number of global execution flags that affect how program blocks are executed.

Fields:

The following table describes each field of the PCR.

Bits	Field	Description
0	M	Mode Control – Configures the processor to run in a specific mode. 0 – Default Mode (D-Morph) 1 – Multi-Threading Mode (T-Morph)
7:1	-	Reserved
15:8	XFLAGS	Global Execution Flags XFLAGS[0] – Inhibit Branch Predictor XFLAGS[1] – Inhibit Load Predictor XFLAGS[2] – Block Synchronization Required XFLAGS[3] – Thread Synchronization Required XFLAGS[4] – Break Before Block XFLAGS[5] – Break After Block XFLAGS[6] – Reserved XFLAGS[7] – Reserved

Bits	Field	Description
63:16	-	Reserved

Notes:

All PCR fields are cleared (zeroed) during reset.

PSR

Processor Status Register

Format:

31 20	19	18	17	16	15 4	3	2	1	0
-	T3	T2	T1	T0	-	IE	-	RE	E

Description:

The Processor Status Register (PSR) is used to record status information for the entire TRIPS processor. Only the processor may set status flags. Software must clear this register to restart the processor.

Fields:

The following table describes each field of the PSR.

Bits	Field	Description
0	E	Exception Flag – Indicates that the processor has halted due to an exception. One or more of the following flags will also be set to indicate the exception type.
1	RE	Reset Exception Flag – Indicates that a Reset Exception has occurred.
2	-	Reserved
3	IE	Interrupt Exception Flag – Indicates that an Interrupt Exception has occurred.
15:4	-	Reserved
16	T0	Thread 0 Exception Flag – Indicates that an exception has occurred for thread slot 0.
17	T1	Thread 1 Exception Flag – Indicates that an exception has occurred for thread slot 1.
18	T2	Thread 2 Exception Flag – Indicates that an exception has occurred for thread slot 2.

Bits	Field	Description
19	T3	Thread 3 Exception Flag – Indicates that an exception has occurred for thread slot 3.
31:20	-	Reserved

Notes:

All PSR fields are cleared (zeroed) during reset, but the processor will immediately report a Reset Exception after the reset signal has been deasserted.

The processor will come to a halt as soon as possible after a processor exception has been detected.

This register is not generally writeable, just clearable. A write to this register will cause all fields to be cleared.

In some cases, the processor may set multiple exception flags to indicate simultaneous exceptions.

See Chapter 7 for additional details related to exceptions.

TCRs

Thread Control Registers

Format:

31 … 8	7 … 4	3 … 1	0
-	ASID	-	V

Description:

The TRIPS processor includes four *Thread Control Registers (TCRs)*. Each TCR is associated with an independent processor thread slot. TCRs associated with threads 1-3 are ignored when the processor is not in the Multi-Threading Mode (T-Morph).

Fields:

The following table describes each field of the TCR.

Bits	Field	Description
0	V	Valid Bit – Indicates whether this thread is valid and should be run or is not valid and should remain idle. 0 – Thread is not valid 1 – Thread is valid
3:1	-	Reserved
7:4	ASID	Address Space Identifier – This field determines which virtual address space the thread will use.
31:8	-	Reserved

Notes:

All TCR fields are cleared (zeroed) during reset.

TSRs

Thread Status Registers

Format:

31 … 7	6	5	4	3	2	1	0
-	SC	ST	EX	TO	BR	FE	E

Description:

The TRIPS processor includes four *Thread Status Registers (TSRs)*. Each TSR is associated with an independent processor thread slot. The TSR is used to record status information for the corresponding program thread. Only the processor may set status flags. Software must clear this register after servicing a thread exception.

Fields:

The following table describes each field of the TSR.

Bits	**Field**	**Description**
0	E	Exception Flag – Indicates that a thread exception has occurred. One or more of the following flags will also be set to indicate the exception type.
1	FE	Fetch Exception Flag – Indicates that a Fetch Exception has occurred.
2	BR	Breakpoint Exception – Indicates that a Breakpoint Exception has occurred.
3	TO	Timeout Exception – Indicates that a Timeout Exception has occurred.
4	EX	Execute Exception – Indicates that an Execute Exception has occurred.
5	ST	Store Exception – Indicates that a Store Exception has occurred.

Bits	Field	Description
6	SC	System Call Exception – Indicates that a System Call Exception has occurred.
31:7	-	Reserved

Notes:

All TSR fields are cleared (zeroed) during reset.

This register is not generally writeable, just clearable. A write to this register will cause all fields to be cleared.

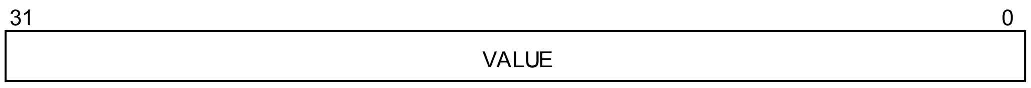
In some cases, the processor may set multiple exception flags to indicate simultaneous exceptions.

See Chapter 7 for additional details related to exceptions.

TVR

Timeout Value Register

Format:



Description:

The Timeout Value Register (TVR) is used to specify a Timeout Value for the TRIPS processor. This value specifies the maximum number of clock cycles that the processor will wait for an individual program block to execute before reporting a Timeout Exception. See Chapter 7 for additional details related to Timeout Exceptions.

Fields:

The following table describes each field of the TVR.

Bits	**Field**	**Description**
31:0	VALUE	Timeout Value

Notes:

The TVR is cleared (zeroed) during reset.

A zero in the Timeout Value field is interpreted as 2^{32}, rather than 0.

Chapter 5 - Queues

5.1 Execution Queues

This chapter describes the processor's architected queues, which include the *Instruction Queue*, *Read Queue*, *Write Queue*, and *Load & Store Queue*. Collectively, these are referred to as execution queues. They are used to buffer instructions, operands, and results as a program block executes.

NOTE: The following high-level descriptions correspond to the TRIPS processor architecture. Details may vary within the actual processor implementation.

5.2 Instruction Queue

The Instruction Queue (IQ) manages up to 128 instructions for each executed program block. Instructions are loaded into the queue when a block is fetched. Each instruction executes only after receiving all required operands and predicates. Instructions are invalidated from the queue after being executed or when the block is committed.

Each instruction within the IQ is assigned an Instruction ID that corresponds to its position in the queue. Instructions with lower IDs generally have higher priority than instructions with higher IDs.

The following information is kept for each instruction in the IQ.

- VALID – Indicates a valid IQ entry
- INST – 32-bit instruction (as fetched from memory)
- OP0 – Operand 0 pending status and 64-bit value
- OP1 – Operand 1 pending status and 64-bit value
- PRED – Predicate pending status
- NULL – Null token received status

- EXCEPT – Exception token received status
- READY – Ready to execute status

Instructions are decoded as they are fetched into the IQ. If the instruction requires one operand, the OP0 pending status will be set to true. If the instruction requires two operands, both the OP0 and OP1 pending status will be set to true. If the instruction is predicated, the PRED pending status will be set to true. The NULL and EXCEPT status are initialized to false.

If an instruction is valid (and not a NOP) but has no pending operands or pending predicate, it will be immediately ready for execution and its READY status will be set to true. Otherwise, its READY status will be set to false. As long as an instruction has pending operands or a pending predicate, the instruction will remain unready. Dataflow tokens must be received (from other instructions) in order for a pending instruction to become ready for execution.

If a dataflow token targets an instruction's operand slot, the 64-bit value will be saved and the corresponding OP0 or OP1 pending status will be set to false. If the token is an exception or null token, the NULL or EXCEPT status will also be set to true.

If a dataflow token targets an instruction's predicate slot, the predicate value will be examined. If the predicate value matches the instruction's predicating condition (true or false), the PRED pending status will be set to false. However, special handling is required if the token is an exception or null token. If an exception token is received for a predicate, the predicate value will always be interpreted as a false predicate and the instruction's EXCEPT status will be set to true. If a null token is received for a predicate, the token will be completely ignored.

Any time that an instruction's operand or predicate pending status is modified, the READY status is recomputed. If an instruction has no pending operands or predicate, it will be marked ready for execution.

When an instruction executes, the NULL and EXCEPT status information will be examined. If the EXCEPT status is true, the instruction will always produce an exception token as a result. Otherwise, if the NULL status is true, the instruction will produce a null token as a result. If the EXCEPT and NULL status are both false, the instruction will produce a normal token as a result.

5.3 Read Queue

The Read Queue (RQ) manages up to 32 read instructions for each executed program block. Read instructions are loaded into the queue when a block is fetched. Each read executes as soon as the required execution resources become available. Instructions are invalidated from the queue after being executed or when the block is committed.

Each instruction within the RQ is assigned a Read ID that corresponds to its position in the queue. Instructions with lower IDs generally have higher priority than instructions with higher IDs.

The RQ is divided into four banks, just like the General Register set. RQ bank 0 holds read instructions 0, 4, 8, etc. RQ bank 1 holds reads instructions 1, 5, 9, etc. In general, bank `i` holds reads instructions `i + 4*j` (where `j` ranges from 0 to 7). Read instructions in RQ bank `i` may only read General Registers in GR bank `i`.

The following information is kept for each read instruction in the RQ.

- VALID – Indicates a valid RQ entry
- GR – Identifies the GR to be read (from the same bank)
- RT0 – Describes the primary target of the read
- RT1 – Describes the secondary target of the read (optional)

See Chapter 8 for more information about the read instruction.

When speculative block execution is implemented, the Read Queue may also need to wait for a previous write value to be written or forwarded. Those details are beyond the scope of this document.

5.4 Write Queue

The Write Queue (WQ) manages up to 32 write instructions for each executed program block. Write instructions are loaded into the queue when a block is fetched. Write instructions operate only after block execution is complete and during the block commit step.

Each instruction within the WQ is assigned a Write ID that corresponds to its position in the queue. Instructions with lower IDs generally have higher priority than instructions with higher IDs.

The WQ is divided into four banks, just like the General Register set. WQ bank 0 holds write instructions 0, 4, 8, etc. WQ bank 1 holds write instructions 1, 5, 9, etc. In general, bank `i` holds write instructions `i + 4*j` (where `j` ranges from 0 to 7). Write instructions in WQ bank `i` may only write General Registers in GR bank `i`.

The following information is kept for each write instruction in the WQ.

- VALID – Indicates a valid WQ entry
- GR – Identifies the GR to be written (from the same bank)
- DATA – 64-bit data value
- NULL – Null token received status
- EXCEPT – Exception token received status
- READY – Ready to complete status

Write instructions are examined as they are fetched into the WQ. For each valid instruction, the NULL, EXCEPT, and READY status are initialized to false. Dataflow tokens must be received in order for each write instruction to become ready to complete. When a dataflow token targets a write instruction, the corresponding READY status will be set to true. If the token is a null or exception token, the NULL or EXCEPT status will also be set to true.

Every valid write instruction must become READY in order for block execution to complete. If any write instruction's EXCEPT status is true, an Execute Exception will be reported and no writes will be committed. If the block does commit, the NULL status information determines whether each individual write will be committed or nullified.

When speculative block execution is implemented, the Write Queue also implements block-level register renaming and register forwarding. Those details are beyond the scope of this document.

5.5 Load & Store Queue

The Load & Store Queue (LSQ) manages up to 32 load and store instructions for each executed program block. Store instructions are identified when a block is fetched, but the load and store instructions do not actually arrive at the LSQ until they are first executed from the IQ. Information is added to the LSQ as each load and store instruction arrives.

Each instruction within the LSQ is assigned a Load & Store ID that corresponds to its position in the queue. Instructions with lower IDs generally have higher priority than instructions with higher IDs. However, the IDs are primarily used to establish load dependences and to enforce correct load and store ordering. A load may be dependent upon one or more stores from the same block with matching addresses and lower IDs. The LSQ is normally used in conjunction with a load dependence predictor to satisfy load dependences without inhibiting independent loads.

When a dependence is discovered, the processor forwards one or more bytes of data from the store to the load. Because loads and stores may vary in size from one byte to eight bytes, several *store forwarding* scenarios are possible. In some cases, the processor may need to forward data from multiple stores to satisfy the load. In some cases, the store(s) may provide only part of the required data and the processor must retrieve the remaining bytes from the Data Cache or memory.

The following information is kept for each load instruction in the LSQ.

- LVALID – Indicates a valid LSQ load entry
- TYPE – Information about the load size and type
- ADDR – Virtual address for the load

Load instructions are delivered to the LSQ from the IQ as dataflow tokens carrying information about the load instruction and address. If a null or exception token is received, the load instruction produces an exception token as a result. Otherwise, the load produces a normal token carrying the load data.

The following information is kept for each store instruction in the LSQ.

- SVALID – Indicates a valid LSQ store entry
- TYPE – Information about the store size and type

- ADDR – Virtual address for the store
- DATA – Data to be stored (up to 64 bits)
- NULL – Null token received status
- EXCEPT – Exception token received status
- READY – Ready to complete status

Store instruction slots are identified (using the store mask) and marked in the LSQ as each block is fetched. For each store instruction, the NULL, EXCEPT, and READY status are initialized to false. Dataflow tokens must be received in order for each store instruction to become ready to complete (and ready to forward). These tokens are delivered to the LSQ from the IQ and carry information about the store instruction, address, and data. When a store instruction arrives at the LSQ, the corresponding READY status will be set to true. If the token is a null or exception token, the NULL or EXCEPT status will also be set to true. Stores that receive null or exception tokens are not allowed to forward data to subsequent loads. The store TYPE, ADDR, and DATA are also saved in the LSQ.

Every valid store instruction must become READY in order for block execution to complete. If any store instruction's EXCEPT status is true, an Execute Exception will be reported and no stores will be committed. If the block does commit, the NULL status information determines whether each individual store will be committed or nullified.

The LSQ is one of the more complex mechanisms in the TRIPS processor. Some implementation details have been omitted from the previous description – in particular, details related to detecting and enforcing load dependences. Those details are beyond the scope of this document.

5.6 Program Counter Writes

Although Program Counter (PC) writes do not target a queue, they are handled in much the same way as a register write. Every block must execute a branch instruction in order for block execution to complete. A branch instruction performs an implicit PC write and delivers its result as a dataflow token. If the branch instruction produces a null or exception token, an Execute Exception will be reported for the block.

Chapter 6 - Memory

6.1 Addressing

TRIPS programs use 40-bit addresses to reference memory. Program addresses are always treated as *virtual* addresses and translated to *physical* addresses by the processor. Fetch addresses are translated by an *Instruction TLB (ITLB)*. Load and store addresses are translated by a *Data TLB (DTLB)*.

Each TRIPS program runs within a distinct virtual address space. A 4-bit *Address Space Identifier (ASID)* is used to distinguish between multiple address spaces. For example, one program may be assigned ASID 0 and a second program may be assigned ASID 1. The first program's address 0x0 will normally be translated differently than the second program's address 0x0. ASIDs are specified for each active thread in the TCRs.

The upper 24 bits of a 64-bit address operand should always be zero and are ignored by the processor.

6.2 Segments

Each program's virtual address space is divided into *virtual segments*. Virtual segments may range is size from 64 KB all the way up to 2^{40} bytes (covering the entire virtual address space). Segment sizes must always be a power of two (2^N bytes, with $16 \le N \le 40$).

Each virtual segment has a virtual base address that identifies its position in the overall virtual address space. Each virtual segment corresponds to an associated physical segment in the physical address space. The physical segment is the same size as the virtual segment, but its base address may be different than the virtual base address. Virtual and physical segments must always be naturally aligned in memory.

Each segment may also be assigned protection attributes that allow or prohibit certain types of access to that segment. For example, some segments might be considered writeable, while others are not writeable. The following segment attributes are defined:

- *Executable* – Indicates whether program blocks may be fetched and executed from a segment.
- *Readable* – Indicates whether data may be read (or loaded) from a segment.
- *Writeable* – Indicates whether data may be written (or stored) to a segment.
- *Cacheable* – Indicates whether data from a segment may be cached in a processor's data caches.
- *Mergeable* – Indicates whether individual loads and stores may be merged and issued to the system as larger loads and stores.

Multiple virtual and physical segments may be defined. Segments are defined by software and communicated to the processor by programming the TLBs. These segments may vary in size. Overlapping virtual segments are disallowed. Virtual synonyms (two distinct virtual addresses that translate to the same physical address) are also disallowed.

The TRIPS segment definitions are intended to support simple segment-based memory management, but could also be used to support page-based memory management.

6.3 TLBs

Each TRIPS processor includes an ITLB that may be used to automatically translate virtual addresses to physical addresses for program fetches. The ITLB can define up to sixteen segments at once. Segments are defined by programming the corresponding ITLB Registers (ITLB0 – ITLB15). See section 4.3 for a description of the ITLB register format.

Each TRIPS processor includes a DTLB that may be used to automatically translate virtual addresses to physical addresses for loads and stores. The DTLB can define up to sixteen segments at once. Segments are defined by programming the corresponding DTLB Registers (DTLB0 – DTLB15). See section 4.3 for a description of the DTLB register format.

System software is responsible for managing segment definitions and programming the TLBs appropriately. Depending upon the software implementation and the runtime environment, the ITLB and DTLB registers may need to be reprogrammed during system initialization, in response to thread switches, and/or in response to dynamic TLB misses.

NOTE: Although the architecture manual defines a single ITLB and a single DTLB, the processor implementation actually includes multiple copies of the DTLB (distributed across rows). It is the hardware's responsibility to manage these redundant copies.

6.4 Caches

Each TRIPS processor includes a set of local (level 1) caches. An *Instruction Cache* holds program blocks as they are fetched from system memory by the processor. A *Data Cache* holds data values as they are loaded from system memory by the processor.

A standard line size will be used when implementing the caches. Each cache manages 64-byte cache lines.

The processor caches are not guaranteed to remain coherent with each other or with other caches in a multi-processor system. No special hardware cache coherency support should be assumed.

A Cache Operation Register (COR) is defined to allow system software to explicitly manage the TRIPS processor's caches. Special cache operations may be initiated by writing an appropriate value to the COR. Cache operations may only be initiated when the processor is halted. Once an operation has been initiated, the COR must be read to determine when the operation is complete. See section 4.3 for a description of the COR.

The following cache operations are supported:

- *Inst Cache Flush* – Invalidate all lines in the Instruction Cache
- *Data Cache Flush* – Evict then invalidate all lines in the Data Cache

NOTE: Although the architecture manual defines a single Instruction Cache and a single Data Cache, the processor implementation actually includes multiple cache banks (distributed across rows). It is the hardware's responsibility to manage these distributed cache banks.

6.5 Byte Ordering

When performing multi-byte (halfword, word, doubleword, or chunk) memory accesses, the TRIPS processor uses *big-endian* byte ordering. For example, a doubleword of data at address 0x0 would have its most significant byte stored at address 0x0 and its least significant byte at address 0x7.

6.6 Byte Alignment

When performing multi-byte (halfword, word, doubleword, or chunk) memory accesses, the data being accessed must always be naturally aligned in memory. A halfword address must always be even, with its least significant bit equal to zero. A word address must always have its two least significant bits equal to zero. And so on.

Chapter 7 - Exceptions

7.1 Exception Model

The TRIPS processor operates with a relatively simple exception model. Although a more sophisticated exception model is possible, we have chosen to keeps things simple.

When an exceptional condition occurs, the TRIPS processor will come to a halt and deliver a signal to a separate control processor. This separate processor is expected to execute an exception handler that reads and writes various registers in the TRIPS processor to diagnose and service the exception. The TRIPS processor itself does not need to vector to an exception handler, switch between operating modes, mask interrupts, or perform special context synchronization while modifying its own control registers.

7.2 Exception Status

7.2.1 Processor Exception Status

At any given time, a TRIPS processor may either be *running* or *halted*. The processor status is indicated in the Processor Status Register (PSR). The PSR's Exception Flag will be clear (zero) when the processor is running and set (one) when the processor is halted. The PSR also includes some exception type information. The processor status is reflected both in the PSR and on a special HALTED output pin.

The processor begins in the halted state after reset. Once halted, the processor will remain halted until an external controller services the processor and explicitly restarts it by clearing the Exception Flag in the PSR. When running, the processor will continue running until an exceptional event occurs that requires special service. Once an exceptional event occurs, the processor will come to a halt as soon as possible and report the exception by raising the Exception Flag in the PSR.

7.2.2 Thread Exception Status

Status information is also maintained for each processor thread slot and reflected in the Thread Status Registers (TSRs). Each TSRs Exception Flag will be clear (zero) when the associated thread slot is inactive or free of exceptions and will be set (one) when the associated thread has encountered a thread exception. Every thread exception also triggers a processor exception (which halts the processor).

Exception handling software is expected to first examine the PSR to determine what type of processor exception occurred. When the PSR indicates that a thread exception has occurred, the exception handler must then examine a specific TSR to determine additional exception type information. The TSR must be cleared in order to restart a thread after a thread exception has occurred. The PSR must also be cleared in order to restart the processor.

7.3 Exception Types

The TRIPS architecture defines two broad categories of exceptions – *Processor Exceptions* and *Thread Exceptions*. Processor exceptions are associated with an entire TRIPS processor, while thread exceptions are associated with just a single thread slot.

7.3.1 Processor Exception Types

Processor exceptions are divided into several types. These are summarized in the following table.

Type	PSR Bit
Reset Exception	1
Interrupt Exception	3
Thread 0 Exception	16
Thread 1 Exception	17
Thread 2 Exception	18
Thread 3 Exception	19

When a processor exception occurs, the processor will record the exception type in the Processor Status Register (PSR) by setting one or more exception type flags. It is possible for multiple processor exceptions to be reported simultaneously.

Reset exceptions occur when the processor is first powered on (or otherwise reset). Interrupt exceptions occur in response to an external interrupt request. Thread exceptions occur when an active program thread encounters a thread exception.

Most processor exceptions are considered *asynchronous* and may be reported independent of any particular processor thread. If several processor exceptions occur simultaneously, they may be prioritized by software in the order listed above – with reset exceptions having the highest priority and thread exceptions having the lowest priority.

Section 7.4 includes detailed descriptions for each type of exception.

7.3.2 Thread Exception Types

Thread exceptions are divided into several types. These are summarized in the following table.

Type	**TSR Bit**
Fetch Exception	1
Breakpoint Exception	2
Timeout Exception	3
Execute Exception	4
Store Exception	5
System Call Exception	6

When a thread exception occurs, the processor will record the exception type in the corresponding Thread Status Register (TSR) by setting one or more exception type flags. It is possible for multiple thread exceptions to be reported simultaneously. For each block, thread exceptions are normally detected and reported in the order listed above.

Most thread exceptions are *block-precise*, meaning that they are discovered and reported while attempting to fetch or execute a particular program block. The program state is precisely maintained, such that all prior blocks (but no subsequent blocks) have completed before the exception is reported. In most cases, the block that caused the exception will not itself complete and an exception handler will be triggered with the PC pointing to the block that caused the exception.

The System Call Exception is also considered block-precise, but triggers the exception handler only after completing the block that issued the system call instruction.

The Store Exception is a special case and may be reported *imprecisely* for performance reasons. Stores are typically allowed to be buffered and committed to memory in the background, without delaying subsequent program execution. For this reason, one or more additional program blocks may be allowed to complete before the exception is discovered and reported.

Some thread exceptions are recoverable, while others are considered fatal to the associated thread and program (but not the system). A software exception handler is expected to diagnose each reported exception and decide whether to proceed or to kill the associated program.

Section 7.4 includes detailed descriptions for each type of exception.

7.4 Exception Descriptions

7.4.1 Alphabetical Exception List

The rest of this chapter includes individual exception descriptions, listed in alphabetical order. Thread exception descriptions and processor exception descriptions are intermixed.

BREAKPOINT EXCEPTION

Type:

Thread Exception

TSR[BR] == 1

Description:

A Breakpoint Exception occurs anytime a breakpoint is encountered during program execution. Breakpoints are normally associated with specific program blocks and are discovered after fetching the associate program block. Breakpoint Exceptions may be triggered either before or after executing the block, depending upon the type of breakpoint.

Breakpoints are defined by setting an appropriate Block Execution Flag in a block header or a Global Execution Flag in the Program Control Register (PCR). If the *Break Before Block* flag is set, the Breakpoint Exception will occur just prior to executing the block. If, instead, the *Break After Block* flag is set, the Breakpoint Exception will occur just after executing and committing the block.

For blocks that execute a system call when the *Break After Block* flag is set, the processor will report a Breakpoint Exception and a System Call Exception simultaneously. The system call should be serviced before processing the breakpoint.

Debugging systems are expected to use the Break Before Block flag for implementing normal software breakpoints and the Break After Block flag for single-stepping.

EXECUTE EXCEPTION

Type:

Thread Exception

TSR[EX] == 1

Description:

An Execute Exception may occur while attempting to execute a program block. This type of exception indicates that the program block could not be fully or properly executed due to some form of execution error (normally caused by a program error).

In most cases, this is considered a non-recoverable exception and the exception handler must abort the associated program. In some cases, it may be possible (or necessary) to emulate program block execution using software and then resume normal execution from the subsequent program block.

Several conditions are defined that can trigger an Execute Exception. These are described below:

- *Divide-By-Zero Error* – This type of error occurs whenever an integer divide instruction attempts to divide by zero.
- *Misaligned Branch Error* – This type of error occurs whenever a branch instruction executes and computes a target address or an offset that is not aligned to a 128-byte program chunk boundary.
- *Misaligned Load Error* – This type of error occurs whenever a load instruction executes and computes an address that is not naturally aligned.
- *Misaligned Store Error* – This type of error occurs whenever a load instruction executes and computes an address that is not naturally aligned.
- *DTLB Translation Error* – This type of error occurs whenever a load or store address cannot be translated by any valid DTLB register.

- *DTLB Protection Error* – This type of error occurs whenever a load addresses an unreadable memory segment or a store addresses an unwriteable memory segment.
- *Lock Error* – This type of error occurs whenever a LOCK instruction addresses a cacheable memory segment.
- *External Load Error* – This type of error occurs whenever an external load request results in an error message from the system.

The TRIPS execution model and its support for predicated execution lead to some special requirements for processing Execute Exceptions. Execute Exceptions are represented using exception tokens. These tokens are generated whenever exceptions are detected during instruction execution. Tokens propagate from instruction to instruction. If a block produces one or more exception tokens as block outputs (write, store, or branch), an Execute Exception will be reported for that block. See Chapter 5 for more information about exception tokens and exception propagation.

FETCH EXCEPTION

Type:

Thread Exception

TSR[FE] == 1

Description:

A Fetch Exception may occur while attempting to fetch a program block. This type of exception indicates that the block could not be properly fetched or that it included some illegal information.

In most cases, this is considered a non-recoverable exception and the exception handler must abort the associated program. It may be necessary to emulate program block fetching in order to diagnose the exception and decide whether to continue or abort the program.

Several conditions are defined that can trigger a Fetch Exception. These are described below:

- *Misaligned PC Error* – This type of error occurs whenever the Program Counter (PC) holds an address that is not aligned to a 128-byte program chunk boundary. This should only occur if the PC is misprogrammed by the system software.
- *ITLB Translation Error* – This type of error occurs whenever a fetch address cannot be translated by any valid ITLB register.
- *ITLB Protection Error* – This type of error occurs whenever a fetch address accesses an unexecutable memory segment.
- *External Fetch Error* – This type of error occurs whenever an external fetch request results in an error message from the system.
- *Block Header Error* – This type of error occurs whenever a block is fetched that does not include the proper header mark and a legal block type.

INTERRUPT EXCEPTION

Type:

Processor Exception

PSR[IE] == 1

Description:

Interrupt Exceptions occur in response to an external interrupt request signal. Whenever the interrupt request signal is asserted, the processor will halt all active threads and trigger an Interrupt Exception. All threads will stop on valid block boundaries. In some cases, threads may be stopped with one or more thread exceptions pending.

Interrupts may be used for thread switching, aborting programs, or other types of periodic service.

RESET EXCEPTION

Type:

Processor Exception

PSR[RE] == 1

Description:

A Reset Exception occurs any time the processor is reset. This could be due to system power-on or a system soft reset mechanism. During the reset or immediately after the reset condition is deasserted, the processor will report a Reset Exception.

The processor also treats the case in which all thread slots are invalid (based upon the TCR bits) as a reset condition.

The system software will normally respond to the Reset Exception by configuring the processor and activating one or more threads. In some cases, it may choose to ignore the exception and leave the processor halted.

STORE EXCEPTION

Type:

Thread Exception

TSR[ST] == 1

Description:

A Store Exception occurs whenever an error is encountered while attempting to store to system memory. Because external stores are typically buffered and allowed to complete in the background (without delaying subsequent execution), Store Exceptions may be discovered and reported imprecisely. This means that one or more additional program blocks may be allowed to complete before the Store Exception is reported. Subsequent exceptions, however, should never be reported ahead of the Store Exception. It is possible for a Store Exception to be reported simultaneously with one or more additional thread exceptions.

Only one condition is currently defined that can trigger a Store Exception:

- *External Store Error* – This type of error occurs whenever an external store request results in an error message from the system.

SYSTEM CALL EXCEPTION

Type:

Thread Exception

TSR[SC] == 1

Description:

A System Call Exception occurs in response to a system call instruction (SCALL) being executed. The processor will allow the program block that executed the system call instruction to commit its results (assuming that there are no other exceptions) before reporting the System Call Exception.

A software convention determines how the system call will be processed. The exception handler is expected to read one or more General Registers to determine which system function to perform and to obtain any required arguments. After performing the system function, execution should resume at the address indicated in the return address register (see the TRIPS ABI document for details). The exception handler is responsible for copying the return address into the PC before restarting the processor.

THREAD EXCEPTION

Type:

Processor Exception

PSR[T#] == 1

Description:

Thread Exceptions occur whenever an active thread encounters an exceptional condition. The Processor Status Register (PSR) identifies which thread encountered the exception. One of the Thread Status Registers (TSRs) must be examined in order to determine the specific type of thread exception.

If multiple threads are active, it is possible for multiple Thread Exceptions to be reported simultaneously. The system software is expected to service (or switch) all threads before restarting the processor.

TIMEOUT EXCEPTION

Type:

Thread Exception

TSR[TO] == 1

Description:

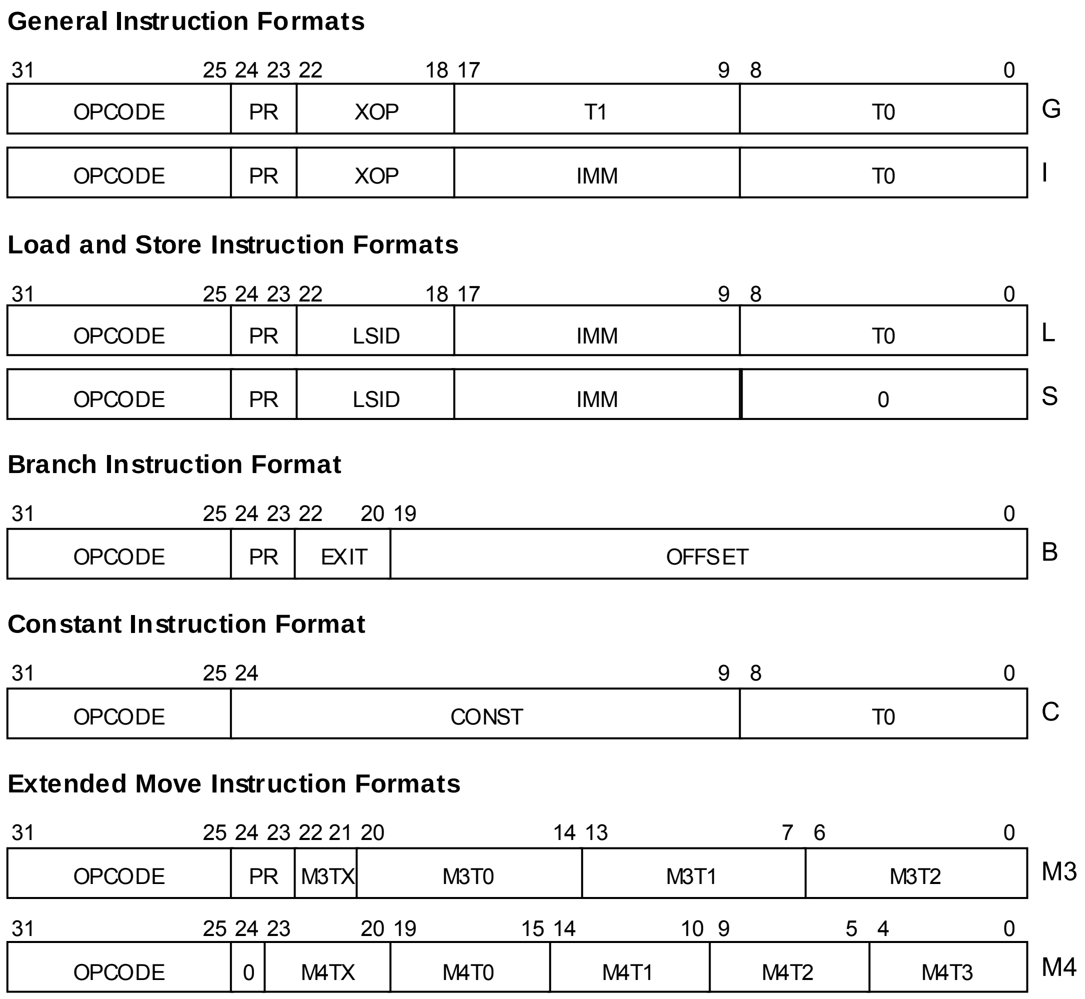
A Timeout Exception may occur while attempting to execute a program block. With the TRIPS execution model, programming errors can sometimes lead to missing operands and instructions that never fire. If block execution does not complete within a specified number of cycles, a Timeout Exception will be reported.

The *Timeout Value Register (TVR)* specifies the maximum number of cycles that the processor will wait for execution to complete before reporting a Timeout Exception. System software is expected to program the TVR with an appropriate value during system initialization.

Chapter 8 - Instructions

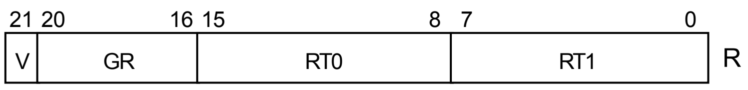
8.1 Instruction Formats

The TRIPS processor architecture defines eight regular instruction formats (G, I, L, S, B, C, M3, and M4), as shown in the following diagram.

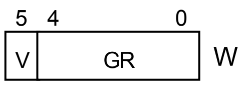


Two special formats (R and W) are used for General Register read and write instructions.

Read Instruction Format



Write Instruction Format



Some instructions use identical formats and fields, but require a different number of input operands. The number of operands is not explicitly encoded in the instructions. This information must be determined from the operation codes.

An extended *format specifier* is defined and used throughout the rest of this chapter to describe both the instruction encoding and the number of input operands. The extended format specifiers are defined below.

Format	Description
B : 0	Uses the B encoding and no operands
B : 1	Uses the B encoding and one operand
C : 0	Uses the C encoding and no operands
C : 1	Uses the C encoding and one operand
G : 0	Uses the G encoding and no operands
G : 1	Uses the G encoding and one operand
G : 2	Uses the G encoding and two operands
I : 0	Uses the I encoding and no operands
I : 1	Uses the I encoding and one operand
L : 1	Uses the L encoding and one operand
M3 : 1	Uses the M3 encoding and one operand
M4 : 1	Uses the M4 encoding and one operand

Format	Description
R : 0	Uses the R encoding and no operands
S : 2	Uses the S encoding and two operands
W : 1	Uses the W encoding and one operand

8.2 Instruction Fields

The fields used in the various instruction formats are defined here.

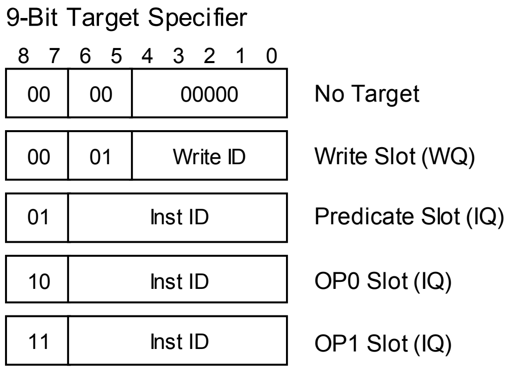
Field	Description
0	*Zero* – This field must include just zero bits.
CONST	*Constant Value* – This 16-bit field is used to encode a constant value. Depending upon the individual instruction, this value may be treated as signed or unsigned.
EXIT	*Exit Number* – A 3-bit number is assigned to each unique block exit and encoded with each branch instruction. Although the exit number has no functional meaning, it is included to facilitate hardware branch prediction.
GR	*General Register Number* – Each read and write instruction includes a 5-bit number identifying the General Register to be read or written. This General Register must always be located in the same bank as the read or write instruction.
IMM	*Immediate Value* – This 9-bit field is used to encode a signed immediate value (ranging from -256 to 255).
LSID	*Load & Store Identifier* – This 5-bit field identifies the Load & Store Queue entry to be used by each load or store instruction. In addition, this identifier defines a relative order for loads and stores within a block.
M3T0	*M3 Target 0* – This 7-bit field provides the lower bits of the first target of a MOV3 instruction.
M3T1	*M3 Target 1* – This 7-bit field provides the lower bits of the second target of a MOV3 instruction.

Field	Description
M3T2	*M3 Target 2* – This 7-bit field provides the lower bits of the third target of a MOV3 instruction.
M3TX	*M3 Target X* – This 2-bit field provides the upper bits for all three targets of a MOV3 instruction.
M4T0	*M4 Target 0* – This 5-bit field provides the lower bits of the first target of a MOV4 instruction.
M4T1	*M4 Target 1* – This 5-bit field provides the lower bits of the second target of a MOV4 instruction.
M4T2	*M4 Target 2* – This 5-bit field provides the lower bits of the third target of a MOV4 instruction.
M4T3	*M4 Target 3* – This 5-bit field provides the lower bits of the fourth target of a MOV4 instruction.
M4TX	*M4 Target X* – This 4-bit field provides the upper bits for all four targets of a MOV4 instruction.
OFFSET	*Offset Value* – This 20-bit field is used to encode a signed branch offset value. This value is always treated as a chunk offset, rather than a byte offset.
OPCODE	*Primary Operation Code* – This 7-bit field is used to encode a unique operation code for each instruction or group of instructions. Instructions that share the same primary operation code must include an extended operation code. The primary operation code is also used to determine the instruction format.
PR	*Predicate Field* – This 2-bit field describes whether an instruction is predicated and, if so, upon what condition. This field is encoded as follows: 00 – Not predicated 01 – (Reserved) 10 – Predicated upon False 11 – Predicated upon True
RT0	*Read Target 0* – This field describes the first target associated with a read instruction.

Field	Description
RT1	*Read Target 1* – This field describes the second target associated with a read instruction. The RT1 field is ignored if its value is identical to the RT0 field.
T0	*General Target 0* – This field describes the first target associated with a regular instruction.
T1	*General Target 1* – This field describes the second target associated with a regular instruction.
V	*Valid Bit* – This single-bit field describes whether each read or write instruction is treated as valid or ignored.
XOP	*Extended Operation Code* – This 5-bit field is used to encode a unique operation code for each instruction within a primary operation code group.

8.3 Target Specifiers

Most instructions are encoded with one or more target fields that determine where the instruction's result should be sent. A general 9-bit target specifier is defined, as described in the following diagram.



This 9-bit target specifier allows any regular instruction to send its result to the predicate slot, OP0 slot, or OP1 slot of any other regular instruction (in the Instruction Queue). Alternatively, it may send its result to a write instruction (in the Write Queue). The all-zero encoding is used to indicate the absence of a target. When producing a predicate, only the least significant bit of the result is used. All other values are 64 bits wide.

Read instructions encode their targets using 8-bit fields, but this is just a subset of the general 9-bit target specifier. The upper, ninth bit is implicit and always set to one. This allows a read instruction to send a value to either operand slot of any regular instruction (in the Instruction Queue), but not to predicate slots or write slots. Additionally, if the RT1 field is equal to the RT0, this indicates the absence of a second target. The read value is 64 bits wide.

Extended move instructions (MOV3 and MOV4) encode their targets in a special way. Each 9-bit target specifier is formed by concatenating a set of common upper bits with a set of unique lower bits.

8.4 Instruction Set Summary

The TRIPS processor instruction set is divided into several classes of instructions. These classes, as well as the instructions in each class, are described below.

8.4.1 Read Instructions

Read instructions are used to retrieve a General Register value and deliver it to one or more targets within the execution grid. These instructions use a special R instruction format and may only be executed from the Read Queue. There is only one type of read instruction.

Name	Full Name	Format
READ	Read General Register	R : 0

8.4.2 Write Instructions

Write instructions are used to conditionally write a new value to a General Register. These instructions use a special W instruction format and may only be executed from the Write Queue. There is only one type of write instruction.

Name	Full Name	Format
WRITE	Write General Register	W : 1

8.4.3 Load Instructions

Load instructions are used to retrieve a byte, halfword, word, or doubleword value from memory. A simple set of load instructions are used to load integers, floating-point numbers, or any other arbitrary data type. Special load instructions are defined for loading and sign-extending signed integers. The following load instructions are defined. Each instruction accepts one operand (an address).

Name	Full Name	Format
LB	Load Byte	L : 1
LBS	Load Byte Signed	L : 1
LH	Load Halfword	L : 1
LHS	Load Halfword Signed	L : 1
LW	Load Word	L : 1
LWS	Load Word Signed	L : 1
LD	Load Doubleword	L : 1

8.4.4 Store Instructions

Store instructions are used to modify a byte, halfword, word, or doubleword value in memory. A single set of store instructions are used to store integers, floating-point numbers, or any arbitrary data type. The following store instructions are defined. Each instruction accepts two operands (one address and one data value).

Name	Full Name	Format
SB	Store Byte	S : 2
SH	Store Halfword	S : 2
SW	Store Word	S : 2
SD	Store Doubleword	S : 2

8.4.5 Integer Arithmetic Instructions

The following instructions are defined for performing arithmetic operations with signed and unsigned integers. Each instruction accepts two operands. Supported operations include addition, subtraction, multiplication, and division.

Name	Full Name	Format
ADD	Add	G : 2
SUB	Subtract	G : 2
MUL	Multiply	G : 2
DIVS	Divide Signed	G : 2
DIVU	Divide Unsigned	G : 2

The following immediate variations allow a small constant to be encoded directly in the instruction. Each instruction accepts one operand.

Name	Full Name	Format
ADDI	Add Immediate	I : 1
SUBI	Subtract Immediate	I : 1
MULI	Multiply Immediate	I : 1
DIVSI	Divide Signed Immediate	I : 1
DIVUI	Divide Unsigned Immediate	I : 1

8.4.6 Integer Logical Instructions

The following instructions are defined for performing bitwise logical operations. Each instruction accepts two operands. Supported operations include AND, OR, and XOR.

Name	Full Name	Format
AND	Bitwise AND	G : 2
OR	Bitwise OR	G : 2
XOR	Bitwise XOR	G : 2

The following immediate variations allow a small constant to be encoded directly in the instruction. Each instruction accepts one operand.

Name	Full Name	Format
ANDI	Bitwise AND Immediate	I : 1
ORI	Bitwise OR Immediate	I : 1
XORI	Bitwise XOR Immediate	I : 1

8.4.7 Integer Shift Instructions

The following instructions are defined for performing shift operations with signed and unsigned integers. Each instruction accepts two operands (one data value and one shift amount).

Name	Full Name	Format
SLL	Shift Left Logical	G : 2
SRL	Shift Right Logical	G : 2
SRA	Shift Right Arithmetic	G : 2

The following immediate variations allow the shift amount to be encoded directly in the instruction. Each instruction accepts one operand.

Name	Full Name	Format
SLLI	Shift Left Logical Immediate	I : 1
SRLI	Shift Right Logical Immediate	I : 1
SRAI	Shift Right Arithmetic Immediate	I : 1

8.4.8 Integer Extend Instructions

The following instructions are defined for extending signed integers to a full 64-bit representation. Each instruction accepts one operand.

Name	Full Name	Format
EXTSB	Extend Signed Byte	G : 1
EXTSH	Extend Signed Halfword	G : 1
EXTSW	Extend Signed Word	G : 1
EXTUB	Extend Unsigned Byte	G : 1
EXTUH	Extend Unsigned Halfword	G : 1
EXTUW	Extend Unsigned Word	G : 1

8.4.9 Integer Relational Instructions

The following instructions are defined for performing relational and equivalence tests with signed and unsigned integers. Each instruction accepts two operands.

Name	Full Name	Format
TEQ	Test EQ	G : 2
TNE	Test NE	G : 2
TLE	Test LE	G : 2
TLEU	Test LE Unsigned	G : 2
TLT	Test LT	G : 2
TLTU	Test LT Unsigned	G : 2
TGE	Test GE	G : 2
TGEU	Test GE Unsigned	G : 2
TGT	Test GT	G : 2
TGTU	Test GT Unsigned	G : 2

The following immediate variations allow a small constant to be encoded directly in the instruction. Each instruction accepts one operand.

Name	Full Name	Format
TEQI	Test EQ Immediate	I : 1
TNEI	Test NE Immediate	I : 1
TLEI	Test LE Immediate	I : 1
TLEUI	Test LE Unsigned Immediate	I : 1
TLTI	Test LT Immediate	I : 1
TLTUI	Test LT Unsigned Immediate	I : 1
TGEI	Test GE Immediate	I : 1
TGEUI	Test GE Unsigned Immediate	I : 1
TGTI	Test GT Immediate	I : 1
TGTUI	Test GT Unsigned Immediate	I : 1

8.4.10 Floating-Point Arithmetic Instructions

The following instructions are defined for performing arithmetic with double-precision floating-point numbers. Each instruction accepts two operands.

Name	Full Name	Format
FADD	FP Add	G : 2
FSUB	FP Subtract	G : 2
FMUL	FP Multiply	G : 2
FDIV	FP Divide	G : 2

8.4.11 Floating-Point Conversion Instructions

The following instructions are defined for performing conversions to or from a single-precision or double-precision floating-point representation. Each instruction accepts one operand.

Name	Full Name	Format
FITOD	Convert Integer to Double FP	G : 1
FDTOI	Convert Double FP to Integer	G : 1
FDTOS	Convert Double FP to Single FP	G : 1
FSTOD	Convert Single FP to Double FP	G : 1

8.4.12 Floating-Point Relational Instructions

The following instructions are defined for performing relational and equivalence tests with floating-point numbers. Each instruction accepts two operands.

Name	Full Name	Format
FEQ	FP Test EQ	G : 2
FNE	FP Test NE	G : 2
FLE	FP Test LE	G : 2
FLT	FP Test LT	G : 2
FGE	FP Test GE	G : 2
FGT	FP Test GT	G : 2

8.4.13 Branch Instructions

The following instructions are defined for controlling program execution. Every block must include at least one branch class instruction. Most instructions accept one operand (an address). The SCALL instruction does not require an operand, since its target is special and implicit.

Name	Full Name	Format
BR	Branch	B : 1

Name	Full Name	Format
CALL	Call	B : 1
RET	Return	B : 1
SCALL	System Call	B : 0

The following offset variations allow a PC-relative offset to be encoded directly in the instruction. No additional operands are needed.

Name	Full Name	Format
BRO	Branch with Offset	B : 0
CALLO	Call with Offset	B : 0

8.4.14 Other Instructions

Several additional instructions are defined for moving data values, formulating large constants, and representing empty instruction slots.

Name	Full Name	Format
NULL	Nullify Output	G : 0
MOV	Move	G : 1
MOVI	Move Immediate	I : 0
MFPC	Move From PC	I : 0
GENS	Generate Signed Constant	C : 0
GENU	Generate Unsigned Constant	C : 0
APP	Append Constant	C : 1
NOP	No Operation	C : 0
MOV3	Move To 3 Targets	M3 : 1
MOV4	Move To 4 Targets	M4 : 1
LOCK	Load and Lock	L : 1

8.5 Instruction Codes

8.5.1 Primary Opcode Map

The primary operation codes are assigned such that the instruction format may be easily decoded. The general partitioning is summarized below. Asterisks indicate primary operation codes assigned to multiple instructions (an extended operation code is also assigned).

6:3,2:0	000	001	010	011	100	101	110	111
0000	C:0 NOP	C:0 GENS	C:0 GENU		C:1 APP		M3 : 1 MOV3	M4 : 1 MOV4
0001	B:0 BRO	B:0 CALLO	B:0 SCALL		B:1 BR	B:1 CALL	B:1 RET	
0010	G:0 *		G:1 *	G:1 *			G:2 *	G:2 *
0011	I:0 *							I:1 *
0100	L:1 LB	L:1 LH	L:1 LW	L:1 LD	L:1 LBS	L:1 LHS	L:1 LWS	L:1 LOCK
0101	S:2 SB	S:2 SH	S:2 SW	S:2 SD				
0110								
0111								
1XXX								

8.5.2 Opcode Table

The following table lists all of the assigned primary and extended operation codes, along with each corresponding instruction.

Primary Opcode	Extended Opcode	Format	Instruction
0 (0x00)	-	C : 0	NOP
1 (0x01)	-	C : 0	GENS
2 (0x02)	-	C : 0	GENU

Primary Opcode	Extended Opcode	Format	Instruction
4 (0x04)	-	C : 1	APP
6 (0x06)	-	M3 : 1	MOV3
7 (0x07)	-	M4 : 1	MOV4
8 (0x08)	-	B : 0	BRO
9 (0x09)	-	B : 0	CALLO
10 (0x0A)	-	B : 0	SCALL
12 (0x0C)	-	B : 1	BR
13 (0x0D)	-	B : 1	CALL
14 (0x0E)	-	B : 1	RET
16 (0x10)	0 (0x00)	G : 0	NULL
18 (0x12)	0 (0x00)	G : 1	FITOD
18 (0x12)	1 (0x01)	G : 1	FDTOI
18 (0x12)	2 (0x02)	G : 1	FSTOD
18 (0x12)	3 (0x03)	G : 1	FDTOS
19 (0x13)	0 (0x00)	G : 1	MOV
19 (0x13)	1 (0x01)	G : 1	EXTSB
19 (0x13)	2 (0x02)	G : 1	EXTSH
19 (0x13)	3 (0x03)	G : 1	EXTSW
19 (0x13)	4 (0x04)	G : 1	EXTUB
19 (0x13)	5 (0x05)	G : 1	EXTUH
19 (0x13)	6 (0x06)	G : 1	EXTUW
22 (0x16)	0 (0x00)	G : 2	FADD
22 (0x16)	1 (0x01)	G : 2	FSUB
22 (0x16)	2 (0x02)	G : 2	FMUL
22 (0x16)	3 (0x03)	G : 2	FDIV
22 (0x16)	4 (0x04)	G : 2	FEQ
22 (0x16)	5 (0x05)	G : 2	FLE
22 (0x16)	6 (0x06)	G : 2	FLT
22 (0x16)	7 (0x7)	G : 2	FNE
22 (0x16)	8 (0x8)	G : 2	FGT
22 (0x16)	9 (0x9)	G : 2	FGE
23 (0x17)	0 (0x00)	G : 2	ADD
23 (0x17)	1 (0x01)	G : 2	SUB
23 (0x17)	2 (0x02)	G : 2	MUL

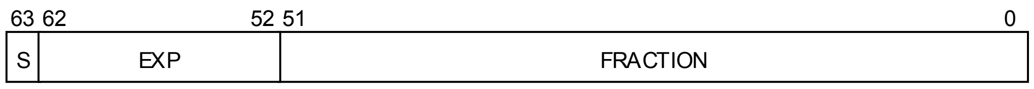
Primary Opcode	Extended Opcode	Format	Instruction
23 (0x17)	3 (0x03)	G : 2	DIVS
23 (0x17)	4 (0x04)	G : 2	DIVU
23 (0x17)	8 (0x08)	G : 2	AND
23 (0x17)	9 (0x09)	G : 2	OR
23 (0x17)	10 (0x0A)	G : 2	XOR
23 (0x17)	12 (0x0C)	G : 2	SLL
23 (0x17)	13 (0x0D)	G : 2	SRL
23 (0x17)	14 (0x0E)	G : 2	SRA
23 (0x17)	16 (0x10)	G : 2	TEQ
23 (0x17)	17 (0x11)	G : 2	TLE
23 (0x17)	18 (0x12)	G : 2	TLT
23 (0x17)	19 (0x13)	G : 2	TLEU
23 (0x17)	20 (0x14)	G : 2	TLTU
23 (0x17)	21 (0x15)	G : 2	TNE
23 (0x17)	22 (0x16)	G : 2	TGT
23 (0x17)	23 (0x17)	G : 2	TGE
23 (0x17)	24 (0x18)	G : 2	TGTU
23 (0x17)	25 (0x19)	G : 2	TGEU
24 (0x18)	0 (0x00)	I : 0	MOVI
24 (0x18)	1 (0x01)	I : 0	MFPC
31 (0x1F)	0 (0x00)	I : 1	ADDI
31 (0x1F)	1 (0x01)	I : 1	SUBI
31 (0x1F)	2 (0x02)	I : 1	MULI
31 (0x1F)	3 (0x03)	I : 1	DIVSI
31 (0x1F)	4 (0x04)	I : 1	DIVUI
31 (0x1F)	8 (0x08)	I : 1	ANDI
31 (0x1F)	9 (0x09)	I : 1	ORI
31 (0x1F)	10 (0x0A)	I : 1	XORI
31 (0x1F)	12 (0x0C)	I : 1	SLLI
31 (0x1F)	13 (0x0D)	I : 1	SRLI
31 (0x1F)	14 (0x0E)	I : 1	SRAI
31 (0x1F)	16 (0x10)	I : 1	TEQI
31 (0x1F)	17 (0x11)	I : 1	TLEI
31 (0x1F)	18 (0x12)	I : 1	TLTI

Primary Opcode	Extended Opcode	Format	Instruction
31 (0x1F)	19 (0x13)	I : 1	TLEUI
31 (0x1F)	20 (0x14)	I : 1	TLTUI
31 (0x1F)	21 (0x15)	I : 1	TNEI
31 (0x1F)	22 (0x16)	I : 1	TGTI
31 (0x1F)	23 (0x17)	I : 1	TGEI
31 (0x1F)	24 (0x18)	I : 1	TGTUI
31 (0x1F)	25 (0x19)	I : 1	TGEUI
32 (0x20)	-	L : 1	LB
33 (0x21)	-	L : 1	LH
34 (0x22)	-	L : 1	LW
35 (0x23)	-	L : 1	LD
36 (0x24)	-	L : 1	LBS
37 (0x25)	-	L : 1	LHS
38 (0x26)	-	L : 1	LWS
39 (0x27)	-	L : 1	LOCK
40 (0x28)	-	S : 2	SB
41 (0x29)	-	S : 2	SH
42 (0x2A)	-	S : 2	SW
43 (0x2B)	-	S : 2	SD

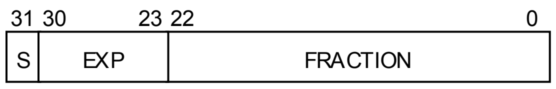
8.6 Floating-Point Support

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. Standard double-precision (64-bit) and single-precision (32-bit) representations are used.

Double-Precision Floating-Point Format



Single-Precision Floating-Point Format



The S field holds a sign bit that determines whether the number is positive (S == 0) or negative (S == 1). The EXP field holds a biased binary exponent. The FRACTION field holds a binary fraction. Depending upon the EXP field, all numbers will be interpreted as either a zero, a normalized value (with an implied leading one), or an infinity. (NaNs and denormalized numbers are not supported and will be interpreted as infinities and zeroes, respectively.)

S	EXP	FRACTION	VALUE
0	Max	x	+Infinity
0	0 < EXP < Max	x	+Normalized
0	0	x	+Zero
1	0	x	-Zero
1	0 < EXP < Max	x	-Normalized
1	Max	x	-Infinity

A full description of the floating-point model is beyond the scope of this document, but some important behaviors and limitations of the TRIPS implementation are summarized below.

- *Single-Precision* – Single-precision floating-point values may be loaded from or stored to memory, but single-precision arithmetic is not supported. All single-precision values must first be converted to double-precisions values before performing arithmetic, comparisons, or integer conversions.
- *Rounding* – For most floating-point operations, TRIPS uses a *round to nearest even* policy. When performing conversion to an integer, TRIPS uses a *round to zero* policy.
- *Overflows* – If overflows occur during floating-point arithmetic or conversion, the resulting value is undefined.
- *Infinities* – For infinity representations, the FRACTION field is undefined and may actually hold a non-zero value.
- *Exceptions* – Floating-point exceptions are not supported.

8.7 Instruction Descriptions

8.7.1 Notation and Conventions

The following symbols are used for referencing or assigning special values.

Symbol	Description
OP0	Instruction's First Operand
OP1	Instruction's Second Operand
PC	Program Counter
READ_ID	Read Queue Position or Identifier
WRITE_ID	Write Queue Position or Identifier

The following pseudo-code operators are used for describing the operations performed by individual instructions. Most of these resemble and behave like the equivalent operators from the C language.

Operator	Description
←	Assignment
+	Integer Addition
-	Integer Subtraction
*	Integer Multiplication
/	Integer Division
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
? :	Conditional Selection

The following pseudo-code functions are used for describing the behavior of various instructions.

Function	**Description**
EQ(v1, v2)	Equivalence Test
FP_ADD(v1, v2)	Floating-Point Add
FP_DIV(v1, v2)	Floating-Point Divide
FP_DTOI(value)	Convert Double FP to Integer
FP_DTOS(value)	Convert Double FP to Single FP
FP_EQ(v1, v2)	Floating-Point Equivalence Test
FP_ITOD(value)	Convert Integer to Double FP
FP_GE(v1, v2)	Floating-Point Greater Than or Equal Test
FP_GT(v1, v2)	Floating-Point Greater Than Test
FP_LE(v1, v2)	Floating-Point Less Than or Equal Test
FP_LT(v1, v2)	Floating-Point Less Than Test
FP_MUL(v1, v2)	Floating-Point Multiply
FP_NE(v1, v2)	Floating-Point Non-Equivalence Test
FP_STOD(value)	Convert Single FP to Double FP
FP_SUB(v1, v2)	Floating-Point Subtract
GE(v1, v2)	Signed Greater Than or Equal Test
GEU(v1, v2)	Unsigned Greater Than or Equal Test
GT(v1, v2)	Signed Greater Than Test
GTU(v1, v2)	Unsigned Greater Than Test
LE(v1, v2)	Signed Less Than or Equal Test
LEU(v1, v2)	Unsigned Less Than or Equal Test
LT(v1, v2)	Signed Less Than Test
LTU(v1, v2)	Unsigned Less Than Test
MEM(addr, bits, LSID)	Memory Accessor
NE(v1, v2)	Non-Equivalence Test
REG(number)	Register Accessor

Function	**Description**
SEXT(value, bits)	Sign-Extend
SLL(value, amount)	Shift Left Logical
SRA(value, amount)	Shift Right Arithmetic (Signed)
SRL(value, amount)	Shift Right Logical (Unsigned)
TARGET(upper, lower)	Target Specifier Bit Concatenation
ZEXT(value, bits)	Zero-Extend

8.7.2 Alphabetical Instruction List

The rest of this chapter includes individual instruction descriptions, listed in alphabetical order.

ADD

Add

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	0	T1	T0	G : 2

Description:

The OP0 value is added to the OP1 value to produce a 64-bit sum. Addition is performed using 64-bit 2's complement arithmetic. Arithmetic overflows are ignored and only the lowest 64 bits of the result are saved.

Operation:

T0, T1 $\leftarrow$ OP0 + OP1

Exceptions:

None

Notes:

The ADD instruction may be used to perform both signed and unsigned addition.

ADDI

Add Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	0	IMM	T0	I : 1

Description:

The OP0 value is added to a sign-extended immediate value to produce a 64-bit sum. Addition is performed using 64-bit 2's complement arithmetic. Arithmetic overflows are ignored and only the lowest 64 bits of the result are saved.

Operation:

T0 $\leftarrow$ OP0 + *SEXT*(IMM, 9)

Exceptions:

None

Notes:

The ADDI instruction may be used to perform both signed and unsigned addition.

AND

Bitwise AND

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	8	T1	T0	G : 2

Description:

A bitwise logical AND operation is performed using the OP0 and OP1 values.

Operation:

T0, T1 $\leftarrow$ OP0 & OP1

Exceptions:

None

Notes:

None

ANDI

Bitwise AND Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	8	IMM	T0	I : 1

Description:

A bitwise logical AND operation is performed using the OP0 value and a sign-extended immediate value.

Operation:

T0 $\leftarrow$ OP0 & *SEXT*(IMM, 9)

Exceptions:

None

Notes:

None

APP

Append Constant

Format:

31 25	24 9	8 0	
4	CONST	T0	C : 1

Description:

A 16-bit constant value is appended to the OP0 value. The OP0 value is shifted left by 16 bits and then OR'd with the zero-extended constant value.

Operation:

T0 ← (OP0 << 16) | *ZEXT*(CONST, 16)

Exceptions:

None

Notes:

This instruction may be used in conjunction with the GENS or GENU instruction to generate signed and unsigned 32-bit, 48-bit, and 64-bit constants.

Because this instruction uses the C format, it may not be predicated.

BR

Branch

Format:

31 25	24 23	22 20	19 0	
12	PR	EXIT	0	B : 1

Description:

The OP0 value is treated as a program address and used to direct the processor to the next program block. The processor will first finish executing the current program block. The Program Counter is an implicit target.

Operation:

address $\leftarrow$ OP0

PC $\leftarrow$ address

Exceptions:

Execute Exception – Misaligned Branch Error

Notes:

The address must be aligned to a 128-byte (chunk) boundary. If any of the seven least-significant bits of the address are non-zero, an Execute Exception occurs.

Any block that executes a BR instruction must not execute any other *branch instruction*.

BRO

Branch with Offset

Format:

31 25	24 23	22 20	19 0	
8	PR	EXIT	OFFSET	B : 0

Description:

A sign-extended program offset is used to direct the processor to the next program block. The processor will first finish executing the current program block. The Program Counter is an implicit source and target.

Operation:

```
byte_offset ← SLL( SEXT( OFFSET, 20 ), 7 )
address ← PC + byte_offset
PC ← address
```

Exceptions:

None

Notes:

Since the offset is specified in chunks, the computed address will always be properly aligned.

Any block that executes a BRO instruction must not execute any other *branch instruction*.

CALL

Call

Format:

31 25	24 23	22 20	19 0	
13	PR	EXIT	0	B : 1

Description:

The OP0 value is treated as a program address and used to direct the processor to the next program block. The processor will first finish executing the current program block. The Program Counter is an implicit target.

Operation:

address $\leftarrow$ OP0

PC $\leftarrow$ address

Exceptions:

Execute Exception – Misaligned Branch Error

Notes:

The address must be aligned to a 128-byte (chunk) boundary. If any of the seven least-significant bits of the address are non-zero, an Alignment Exception occurs.

Any block that executes a CALL instruction must not execute any other *branch instruction*.

This instruction behaves just like the BR instruction, but informs the processor that a subroutine is being called. This may allow the hardware to accurately predict return addresses when needed.

CALLO

Call with Offset

Format:

31 25	24 23	22 20	19 0	
9	PR	EXIT	OFFSET	B : 0

Description:

A sign-extended program offset is used to direct the processor to the next program block. The processor will first finish executing the current program block. The Program Counter is an implicit source and target.

Operation:

```
byte_offset ← SLL( SEXT( OFFSET, 20 ), 7 )
address ← PC + byte_offset
PC ← address
```

Exceptions:

None

Notes:

Since the offset is specified in chunks, the computed address will always be properly aligned.

Any block that executes a CALLO instruction must not execute any other *branch instruction*.

This instruction behaves just like the BRO instruction, but informs the processor that a subroutine is being called. This may allow the hardware to accurately predict return addresses when needed.

DIVS

Divide Signed

Format:

31 – 25	24 23	22 – 18	17 – 9	8 – 0	
23	PR	3	T1	T0	G : 2

Description:

The OP0 value is divided by the OP1 value to form a 64-bit quotient. Both operands are treated as 64-bit signed integers.

Operation:

T0, T1 $\leftarrow$ OP0 / OP1

Exceptions:

Execute Exception – Divide-By-Zero Error

Notes:

An attempt to divide by zero will result in an Execute Exception.

An attempt to divide the maximum negative value by -1 will produce an undefined result.

The resulting quotient is a signed integer that satisfies the equation *dividend = (quotient * divisor) + remainder*, where the remainder has the same sign as the dividend. In other words, the quotient is rounded toward zero.

The remainder may be computed using a sequence of divide, multiply, and subtract instructions.

DIVSI

Divide Signed Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	3	IMM	T0	I : 1

Description:

The OP0 value is divided by a sign-extended immediate value to form a 64-bit quotient. Both operands are treated as 64-bit signed integers.

Operation:

T0 ← OP0 / *SEXT*(IMM, 9)

Exceptions:

Execute Exception – Divide-By-Zero Error

Notes:

An attempt to divide by zero will result in an Execute Exception.

An attempt to divide the maximum negative value by -1 will produce an undefined result.

The resulting quotient is a signed integer that satisfies the equation *dividend = (quotient * divisor) + remainder*, where the remainder has the same sign as the dividend. In other words, the quotient is rounded toward zero.

The remainder may be computed using a sequence of divide, multiply, and subtract instructions.

DIVU

Divide Unsigned

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	4	T1	T0	G : 2

Description:

The OP0 value is divided by the OP1 value to form a 64-bit quotient. Both operands are treated as 64-bit unsigned integers.

Operation:

T0, T1 $\leftarrow$ OP0 / OP1

Exceptions:

Execute Exception – Divide-By-Zero Error

Notes:

An attempt to divide by zero will result in an Execute Exception.

The resulting quotient is an unsigned integer that satisfies the equation *dividend = (quotient * divisor) + remainder*, where the remainder is also unsigned. In other words, the quotient is rounded toward zero.

The remainder may be computed using a sequence of divide, multiply, and subtract instructions.

DIVUI

Divide Unsigned Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	4	IMM	T0	I : 1

Description:

The OP0 value is divided by a zero-extended immediate value to form a 64-bit quotient. Both operands are treated as 64-bit unsigned integers.

Operation:

T0 ← OP0 / *SEXT*(IMM, 9)

Exceptions:

Execute Exception – Divide-By-Zero Error

Notes:

An attempt to divide by zero will result in an Execute Exception.

The resulting quotient is an unsigned integer that satisfies the equation *dividend = (quotient * divisor) + remainder*, where the remainder is also unsigned. In other words, the quotient is rounded toward zero.

The remainder may be computed using a sequence of divide, multiply, and subtract instructions.

Even though the divide is unsigned, the IMM field will still be sign-extended. For this instruction, the IMM field may encode an unsigned value between 0 and 255.

EXTSB

Extend Signed Byte

Format:

31 25	24 23	22 18	17 9	8 0	
19	PR	1	T1	T0	G : 1

Description:

The low-order byte of the OP0 value is sign-extended and written to one or more targets.

Operation:

T0, T1 ← *SEXT*(OP0, 8)

Exceptions:

None

Notes:

None

EXTSH

Extend Signed Halfword

Format:

31 25	24 23	22 18	17 9	8 0	
19	PR	2	T1	T0	G : 1

Description:

The low-order halfword of the OP0 value is sign-extended and written to one or more targets.

Operation:

T0, T1 ← *SEXT*(OP0, 16)

Exceptions:

None

Notes:

None

EXTSW

Extend Signed Word

Format:

31 25	24 23	22 18	17 9	8 0	
19	PR	3	T1	T0	G : 1

Description:

The low-order word of the OP0 value is sign-extended and written to one or more targets.

Operation:

T0, T1 ← *SEXT*(OP0, 32)

Exceptions:

None

Notes:

None

EXTUB

Extend Unsigned Byte

Format:

31 – 25	24 23	22 – 18	17 – 9	8 – 0	
19	PR	4	T1	T0	G : 1

Description:

The low-order byte of the OP0 value is zero-extended and written to one or more targets.

Operation:

T0, T1 ← *ZEXT*(OP0, 8)

Exceptions:

None

Notes:

None

EXTUH

Extend Unsigned Halfword

Format:

31 25	24 23	22 18	17 9	8 0	
19	PR	5	T1	T0	G : 1

Description:

The low-order halfword of the OP0 value is zero-extended and written to one or more targets.

Operation:

T0, T1 ← *ZEXT*(OP0, 16)

Exceptions:

None

Notes:

None

EXTUW

Extend Unsigned Word

Format:

31 25	24 23	22 18	17 9	8 0	
19	PR	6	T1	T0	G : 1

Description:

The low-order word of the OP0 value is zero-extended and written to one or more targets.

Operation:

T0, T1 ← *ZEXT*(OP0, 32)

Exceptions:

None

Notes:

None

FADD

FP Add

Format:

31 25	24 23	22 18	17 9	8 0	
22	PR	0	T1	T0	G : 2

Description:

The OP0 value is added to the OP1 value to produce a sum. Both operands are treated as double-precision floating-point values. The result is also a double-precision floating-point value.

Operation:

T0, T1 ← *FP_ADD*(OP0, OP1)

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

When necessary, a *round to nearest even* policy is used to produce the double-precision floating-point result.

FDIV

FP Divide

Format:

31 25	24 23	22 18	17 9	8 0	
22	PR	3	T1	T0	G : 2

Description:

The OP0 value is divided by the OP1 value to produce a quotient. Both operands are treated as double-precision floating-point values. The result is also a double-precision floating-point value.

Operation:

T0, T1 ← *FP_DIV*(OP0, OP1)

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

When necessary, a *round to nearest even* policy is used to produce the double-precision floating-point result.

This instruction is supported in the TRIPS simulators but NOT supported by the TRIPS prototype hardware! The floating-point divide computation must be performed by software.

FDTOI

Convert Double FP to Integer

Format:

31 25	24 23	22 18	17 9	8 0	
18	PR	1	T1	T0	G : 1

Description:

The OP0 value is treated as a double-precision floating-point value and converted to an equivalent 65-bit signed integer value. The lower 64 bits of the result are delivered to the target(s).

Operation:

T0, T1 ← *FP_DTOI*(OP0)

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

When necessary, a *round to zero* policy is used to produce the integer result.

The FDTOI instruction may be used for converting to both signed and unsigned integers. When an overflow occurs, the result may differ from the IEEE standard. When converting from infinities or NaNs, the result is undefined.

FDTOS

Convert Double FP to Single FP

Format:

31 25	24 23	22 18	17 9	8 0	
18	PR	3	T1	T0	G : 1

Description:

The OP0 value is treated as a double-precision floating-point value and converted to an equivalent single-precision floating-point value.

Operation:

T0, T1 ← *FP_DTOS*(OP0)

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

When necessary, a *round to nearest even* policy is used to produce the single-precision floating-point result.

FEQ

FP Test EQ

Format:

31 25	24 23	22 18	17 9	8 0	
22	PR	4	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. Both operands are treated as double-precision floating-point values. If the OP0 value is equal to the OP1 value, then a true value (1) is produced. Otherwise, a false value (0) is produced.

Operation:

T0, T1 $\leftarrow$ *FP_EQ*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

FGE

FP Test GE

Format:

31 … 25	24 23	22 … 18	17 … 9	8 … 0	
22	PR	9	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. Both operands are treated as double-precision floating-point values. If the OP0 value is greater than or equal to the OP1 value, then a true value (1) is produced. Otherwise, a false value (0) is produced.

Operation:

T0, T1 $\leftarrow$ *FP_GE*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

FGT

FP Test GT

Format:

31 25	24 23	22 18	17 9	8 0	
22	PR	8	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. Both operands are treated as double-precision floating-point values. If the OP0 value is greater than the OP1 value, then a true value (1) is produced. Otherwise, a false value (0) is produced.

Operation:

T0, T1 ← *FP_GT*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

FITOD

Convert Integer to Double FP

Format:

31 25	24 23	22 18	17 9	8 0	
18	PR	0	T1	T0	G : 1

Description:

The OP0 value is treated as a 64-bit signed integer value and converted to an equivalent double-precision floating-point value.

Operation:

T0, T1 ← *FP_ITOD*(OP0)

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

When necessary, a *round to nearest even* policy is used to produce the double-precision floating-point result.

Since the operand is interpreted as a 64-bit signed integer, this instruction will NOT properly convert very large unsigned numbers (greater than or equal to $2^{63)}$. This limitation may be worked around by the compiler.

FLE

FP Test LE

Format:

31 25	24 23	22 18	17 9	8 0	
22	PR	5	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. Both operands are treated as double-precision floating-point values. If the OP0 value is less than or equal to the OP1 value, then a true value (1) is produced. Otherwise, a false value (0) is produced.

Operation:

T0, T1 ← *FP_LE*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

FLT

FP Test LT

Format:

31 25	24 23	22 18	17 9	8 0	
22	PR	6	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. Both operands are treated as double-precision floating-point values. If the OP0 value is less than the OP1 value, then a true value (1) is produced. Otherwise, a false value (0) is produced.

Operation:

T0, T1 ← *FP_LT*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

FMUL

FP Multiply

Format:

31 25	24 23	22 18	17 9	8 0	
22	PR	2	T1	T0	G : 2

Description:

The OP0 value is multiplied by the OP1 value to produce a product. Both operands are treated as double-precision floating-point values. The result is also a double-precision floating-point value.

Operation:

T2, T1 $\leftarrow$ *FP_MUL*(OP0, OP1)

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

When necessary, a *round to nearest even* policy is used to produce the double-precision floating-point result.

FNE

FP Test NE

Format:

31 25	24 23	22 18	17 9	8 0
22	PR	7	T1	T0

G : 2

Description:

The OP0 value is compared with the OP1 value. Both operands are treated as double-precision floating-point values. If the OP1 value is NOT equal to the OP1 value, then a true value (1) is produced. Otherwise, a false value (0) is produced.

Operation:

T0, T1 ← *FP_NE*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

FSTOD

Convert Single FP to Double FP

Format:

31 25	24 23	22 18	17 9	8 0	
18	PR	2	T1	T0	G : 1

Description:

The OP0 value is treated as a single-precision floating-point value and converted to an equivalent double-precision floating-point value.

Operation:

T0, T1 ← *FP_STOD*(OP0)

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

FSUB

FP Subtract

Format:

31 … 25	24 23	22 … 18	17 … 9	8 … 0	
22	PR	1	T1	T0	G : 2

Description:

The OP1 value is subtracted from the OP0 value to produce a difference. Both operands are treated as double-precision floating-point values. The result is also a double-precision floating-point value.

Operation:

T0, T1 ← *FP_SUB*(OP0, OP1)

Exceptions:

None

Notes:

The TRIPS processor's floating-point support is compatible but not fully compliant with the IEEE-754 floating-point standard. NaNs and denormalized numbers are unsupported. Exceptional conditions are not reported. See section 8.6 for more information.

When necessary, a *round to nearest even* policy is used to produce the double-precision floating-point result.

GENS

Generate Signed Constant

Format:

31 25	24 9	8 0	
1	CONST	T0	C : 0

Description:

A 16-bit constant value is sign-extended and written to a target.

Operation:

T0 ← *SEXT*(CONST, 16)

Exceptions:

None

Notes:

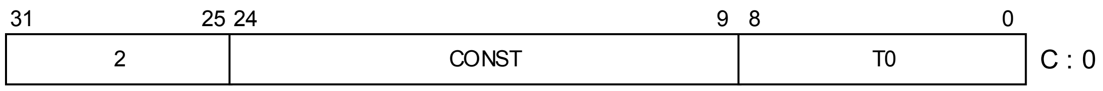
This instruction may be used in conjunction with the APP instruction to generate signed 32-bit, 48-bit, and 64-bit constants.

Because this instruction uses the C format, it may not be predicated.

GENU

Generate Unsigned Constant

Format:



Description:

A 16-bit constant value is zero-extended and written to a target.

Operation:

T0 ← *ZEXT*(CONST, 16)

Exceptions:

None

Notes:

This instruction may be used in conjunction with the APP instruction to generate unsigned 32-bit, 48-bit, and 64-bit constants.

Because this instruction uses the C format, it may not be predicated.

LB

Load Byte

Format:

31 25	24 23	22 18	17 9	8 0	
32	PR	LSID	IMM	T0	L : 1

Description:

An address is computed and used to load a byte from memory. The byte is zero-extended and written to a target.

Operation:

```
address ← OP0 + SEXT( IMM, 9 )
value ← MEM( address, 8, LSID )
T0 ← ZEXT( value, 8 )
```

Exceptions:

Execute Exception – Misaligned Load Error

Execute Exception – DTLB Translation Error

Execute Exception – DTLB Protection Error

Execute Exception – External Load Error

Notes:

Although a 64-bit address is computed, the upper 24 bits should always be zero and will be ignored by the processor.

The LSID is used to check for potential dependences on prior stores within the same block.

LBS

Load Byte Signed

Format:

31 25	24 23	22 18	17 9	8 0	
36	PR	LSID	IMM	T0	L : 1

Description:

An address is computed and used to load a byte from memory. The byte is sign-extended and written to a target.

Operation:

address ← OP0 + *SEXT*(IMM, 9)

value ← *MEM*(address, 8, LSID)

T0 ← *SEXT*(value, 8)

Exceptions:

Execute Exception – Misaligned Load Error

Execute Exception – DTLB Translation Error

Execute Exception – DTLB Protection Error

Execute Exception – External Load Error

Notes:

Although a 64-bit address is computed, the upper 24 bits should always be zero and will be ignored by the processor.

The LSID is used to check for potential dependences on prior stores within the same block.

LD

Load Doubleword

Format:

31 25	24 23	22 18	17 9	8 0	
35	PR	LSID	IMM	T0	L : 1

Description:

An address is computed and used to load a doubleword from memory.

Operation:

address ← OP0 + *SEXT*(IMM, 9)

T0 ← *MEM*(address, 64, LSID)

Exceptions:

Execute Exception – Misaligned Load Error

Execute Exception – DTLB Translation Error

Execute Exception – DTLB Protection Error

Execute Exception – External Load Error

Notes:

The address must be naturally aligned. If any of the three least-significant bits of the address are non-zero, an Execute Exception occurs.

Although a 64-bit address is computed, the upper 24 bits should always be zero and will be ignored by the processor.

The LSID is used to check for potential dependences on prior stores within the same block.

LH

Load Halfword

Format:

31 25	24 23	22 18	17 9	8 0	
33	PR	LSID	IMM	T0	L : 1

Description:

An address is computed and used to load a halfword from memory. The halfword is zero-extended and written to a target.

Operation:

address ← OP0 + *SEXT*(IMM, 9)

value ← *MEM*(address, 16, LSID)

T0 ← *ZEXT*(value, 16)

Exceptions:

Execute Exception – Misaligned Load Error

Execute Exception – DTLB Translation Error

Execute Exception – DTLB Protection Error

Execute Exception – External Load Error

Notes:

The address must be naturally aligned. If the least-significant bit of the address is non-zero, an Execute Exception occurs.

Although a 64-bit address is computed, the upper 24 bits should always be zero and will be ignored by the processor.

The LSID is used to check for potential dependences on prior stores within the same block.

LHS

Load Halfword Signed

Format:

31 25	24 23	22 18	17 9	8 0	
37	PR	LSID	IMM	T0	L : 1

Description:

An address is computed and used to load a halfword from memory. The halfword is sign-extended and written to a target.

Operation:

address ← OP0 + *SEXT*(IMM, 9)

value ← *MEM*(address, 16, LSID)

T0 ← *SEXT*(value, 16)

Exceptions:

Execute Exception – Misaligned Load Error

Execute Exception – DTLB Translation Error

Execute Exception – DTLB Protection Error

Execute Exception – External Load Error

Notes:

The address must be naturally aligned. If the least-significant bit of the address is non-zero, an Execute Exception occurs.

Although a 64-bit address is computed, the upper 24 bits should always be zero and will be ignored by the processor.

The LSID is used to check for potential dependences on prior stores within the same block.

LOCK

Load and Lock

Format:

31 25	24 23	22 18	17 9	8 0
39	PR	LSID	IMM	T0

L : 1

Description:

An address is computed and used to load a byte from memory. The byte is zero-extended and written to a target. Just after the value is loaded from memory (and prior to any other loads or stores to that location), a one is written into that same memory location.

Operation:

address ← OP0 + *SEXT*(IMM, 9)

value ← *MEM*(address, 8, LSID)

L2_MEM(address, 8) ← 1

T0 ← *ZEXT*(value, 8)

Exceptions:

Execute Exception – Misaligned Load Error

Execute Exception – DTLB Translation Error

Execute Exception – DTLB Protection Error

Execute Exception – Lock Error

Execute Exception – External Load Error

Notes:

Although a 64-bit address is computed, the upper 24 bits should always be zero and will be ignored by the processor.

The LSID is used to check for potential dependences on prior stores within the same block (but matching stores are prohibited).

The LOCK instruction is intended to support mutexes in the prototype TRIPS system. The instruction may be used to perform an atomic swap, but its use is subject to several restrictions.

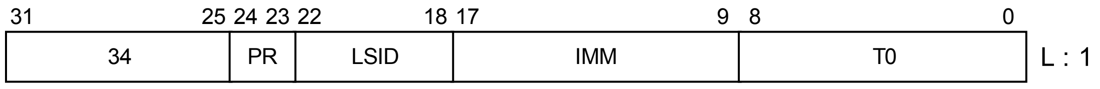
- The processor does not support atomic access to locks in its Data Cache. The LOCK instruction must address a non-cacheable memory segment. Otherwise, an exception will occur.
- A program block should include at most one LOCK instruction.
- The *Block Synchronization Required* flag should be set for the block that holds the LOCK instruction. This guarantees that the block will not be speculatively executed.
- The block should be formulated such that no exceptions are possible after the LOCK instruction has been executed. Otherwise, the lock may be left in an undefined state.
- The lock variable stored in memory must be one byte wide and must represent the locked state with a one.

Improper use of the LOCK instruction can result in undefined behavior for both the software and the system.

LW

Load Word

Format:



Description:

An address is computed and used to load a word from memory. The word is zero-extended and written to a target.

Operation:

address ← OP0 + *SEXT*(IMM, 9)

value ← *MEM*(address, 32, LSID)

T0 ← *ZEXT*(value, 32)

Exceptions:

Execute Exception – Misaligned Load Error

Execute Exception – DTLB Translation Error

Execute Exception – DTLB Protection Error

Execute Exception – External Load Error

Notes:

The address must be naturally aligned. If any of the two least-significant bits of the address are non-zero, an Execute Exception occurs.

Although a 64-bit address is computed, the upper 24 bits should always be zero and will be ignored by the processor.

The LSID is used to check for potential dependences on prior stores within the same block.

LWS

Load Word Signed

Format:

31 25	24 23	22 18	17 9	8 0	
38	PR	LSID	IMM	T0	L : 1

Description:

An address is computed and used to load a word from memory. The word is sign-extended and written to a target.

Operation:

```
address ← OP0 + SEXT( IMM, 9 )
value ← MEM( address, 32, LSID )
T0 ← SEXT( value, 32 )
```

Exceptions:

Execute Exception – Misaligned Load Error

Execute Exception – DTLB Translation Error

Execute Exception – DTLB Protection Error

Execute Exception – External Load Error

Notes:

The address must be naturally aligned. If any of the two least-significant bits of the address are non-zero, an Execute Exception occurs.

Although a 64-bit address is computed, the upper 24 bits should always be zero and will be ignored by the processor.

The LSID is used to check for potential dependences on prior stores within the same block.

MFPC

Move From PC

Format:

31 25	24 23	22 18	17 9	8 0	
24	PR	1	0	T0	I : 0

Description:

The address stored in the Program Counter is copied (or moved) to a target. The Program Counter holds the virtual address of the current program block.

Operation:

T0 $\leftarrow$ PC

Exceptions:

None

Notes:

None

MOV

Move

Format:

31 25	24 23	22 18	17 9	8 0	
19	PR	0	T1	T0	G : 1

Description:

The OP0 value is copied (or moved) to one or more targets.

Operation:

T0, T1 $\leftarrow$ OP0

Exceptions:

None

Notes:

None

MOV3

Move To 3 Targets

Format:

31 25	24 23	22 21	20 14	13 7	6 0	
6	PR	M3TX	M3T0	M3T1	M3T2	M3 : 1

Description:

The OP0 value is copied (or moved) to three targets. Target specifiers are formed by concatenating 2 common upper bits (M3TX) with 7 unique lower bits for each target (M3T0, M3T1, M3T2).

Operation:

TARGET(M3TX, M3T0) ← OP0

TARGET(M3TX, M3T1) ← OP0

TARGET(M3TX, M3T2) ← OP0

Exceptions:

None

Notes:

Since all targets share 2 common upper bits, they are constrained to target the same type of operand slot.

MOV4

Move To 4 Targets

Format:

31 25	24	23 20	19 15	14 10	9 5	4 0	
7	0	M4TX	M4T0	M4T1	M4T2	M4T3	M4 : 1

Description:

The OP0 value is copied (or moved) to four targets. Target specifiers are formed by concatenating 4 common upper bits (M4TX) with 5 unique lower bits for each target (M4T0, M4T1, M4T2, M4T3).

Operation:

TARGET(M4TX, M4T0) $\leftarrow$ OP0

TARGET(M4TX, M4T1) $\leftarrow$ OP0

TARGET(M4TX, M4T2) $\leftarrow$ OP0

TARGET(M4TX, M4T3) $\leftarrow$ OP0

Exceptions:

None

Notes:

Since all targets share 4 common upper bits, they are constrained to target the same type of operand slot and the same subset of instruction identifiers.

Because this instruction uses the M4 format, it may not be predicated.

MOVI

Move Immediate

Format:

31 … 25	24 23	22 … 18	17 … 9	8 … 0	
24	PR	0	IMM	T0	I : 0

Description:

A sign-extended immediate value is copied (or moved) to a target.

Operation:

T0 ← *SEXT*(IMM, 9)

Exceptions:

None

Notes:

None

MUL

Multiply

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	2	T1	T0	G : 2

Description:

The OP0 value is multiplied with the OP1 value to form a 64-bit product. Multiplication is performed using 64-bit 2's complement arithmetic. Arithmetic overflows are ignored and only the lowest 64 bits of the result are saved.

Operation:

T0, T1 $\leftarrow$ OP0 * OP1

Exceptions:

None

Notes:

The MUL instruction may be used to perform both signed and unsigned multiplication.

MULI

Multiply Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	2	IMM	T0	I : 1

Description:

The OP0 value is multiplied with a sign-extended immediate value to form a 64-bit product. Multiplication is performed using 64-bit 2’s complement arithmetic. Arithmetic overflows are ignored and only the lowest 64 bits of the result are saved.

Operation:

T0 ← OP0 * *SEXT*(IMM, 9)

Exceptions:

None

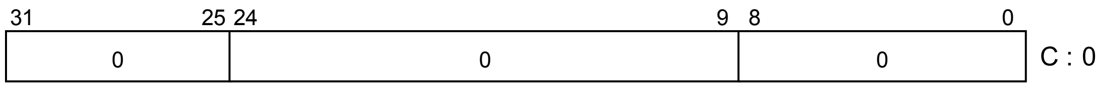
Notes:

The MULI instruction may be used to perform both signed and unsigned multiplication.

NOP

No Operation

Format:



Description:

The opcode zero designates an empty instruction slot (or an instruction that performs no operation).

Operation:

None

Exceptions:

None

Notes:

None

NULL

Nullify Output

Format:

31 25	24 23	22 18	17 9	8 0	
16	PR	0	T1	T0	G : 0

Description:

Null values are sent to each target. Write instructions that receive a null operand are considered nullified (cancelled) writes. Store instructions that receive a null operand are considered nullified (cancelled) stores. Other instructions that receive null operands will propagate them (in most cases).

Operation:

T0, T1 $\leftarrow$ NULL

Exceptions:

None

Notes:

The NULL instruction should usually be predicated (as its only function is to predicate block outputs).

Output nullification should not be confused with general instruction predication. Instructions that receive a null operand will still execute, but will produce a special result. Instructions must still receive all of their expected operands and enabling predicates before they can execute.

OR

Bitwise OR

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	9	T1	T0	G : 2

Description:

A bitwise logical OR operation is performed using the OP0 and OP1 values.

Operation:

T0, T1 ← OP0 | OP1

Exceptions:

None

Notes:

None

ORI

Bitwise OR Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	9	IMM	T0	I : 1

Description:

A bitwise logical OR operation is performed using the OP0 value and a sign-extended immediate value.

Operation:

T0 $\leftarrow$ OP0 | *SEXT*(IMM, 9)

Exceptions:

None

Notes:

None

READ

Read General Register

Format:

21	20 … 16	15 … 8	7 … 0	
1	GR	RT0	RT1	R : 0

Description:

The specified General Register value is retrieved and written to one or more targets.

Operation:

```
reg_bank ← READ_ID % 4
reg_number ← 4 * GR + reg_bank
TARGET( 1, RT0 ) ← REG( reg_number )
if ( RT1 != RT0 ) READ_TARGET( 1, RT1 ) ← REG( reg_number )
```

Exceptions:

None

Notes:

This instruction may only be executed from the Read Queue.

The READ instruction may only retrieve a General Register from the corresponding General Register bank.

RET

Return

Format:

31 25	24 23	22 20	19 0	
14	PR	EXIT	0	B : 1

Description:

The OP0 value is treated as a program address and used to direct the processor to the next program block. The processor will first finish executing the current program block. The Program Counter is an implicit target.

Operation:

address ← OP0

PC ← address

Exceptions:

Execute Exception – Misaligned Branch Error

Notes:

The address must be aligned to a 128-byte (chunk) boundary. If any of the seven least-significant bits of the address are non-zero, an Alignment Exception occurs.

Any block that executes a RET instruction must not execute any other *branch instruction*.

This instruction behaves just like the BR instruction, but informs the processor that a subroutine is being returned from. This may allow the hardware to accurately predict return addresses when needed.

SB

Store Byte

Format:

31 25	24 23	22 18	17 9	8 0	
40	PR	LSID	IMM	0	S : 2

Description:

An address is computed and used to store a byte to memory. The low-order byte of the OP1 value is stored.

Operation:

address ← OP0 + *SEXT*(IMM, 9)

MEM(address, 8, LSID) ← OP1

Exceptions:

Execute Exception – Misaligned Store Error

Execute Exception – DTLB Translation Error

Execute Exception – DTLB Protection Error

Store Exception – External Store Error

Notes:

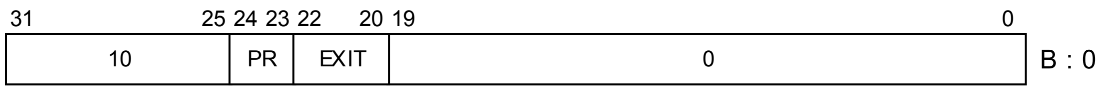
Although a 64-bit address is computed, the upper 24 bits should always be zero and will be ignored by the processor.

The LSID is used to check for subsequent dependent loads within the same block.

SCALL

System Call

Format:



Description:

This instruction may be used to request a special service from the operating system. The processor will first finish executing and committing the current program block, then initiate a System Call Exception. The Program Counter is an implicit target.

Operation:

PC $\leftarrow$ 0

Exceptions:

System Call Exception

Notes:

Software (TRIPS ABI) conventions define how to identify the type of system call and how to save the return address.

Any block that executes an SCALL instruction must not execute other *branch instruction*.

SD

Store Doubleword

Format:

31 25	24 23	22 18	17 9	8 0	
43	PR	LSID	IMM	0	S : 2

Description:

An address is computed and used to store a doubleword to memory.

Operation:

address ← OP0 + *SEXT*(IMM, 9)

MEM(address, 64, LSID) ← OP1

Exceptions:

Execute Exception – Misaligned Store Error

Execute Exception – DTLB Translation Error

Execute Exception – DTLB Protection Error

Store Exception – External Store Error

Notes:

The address must be naturally aligned. If any of the three least-significant bits of the address are non-zero, an Execute Exception occurs.

Although a 64-bit address is computed, the upper 24 bits should always be zero and will be ignored by the processor.

The LSID is used to check for subsequent dependent loads within the same block.

SH

Store Halfword

Format:

31 25	24 23	22 18	17 9	8 0	
41	PR	LSID	IMM	0	S : 2

Description:

An address is computed and used to store a halfword to memory. The low-order halfword of the OP1 value is stored.

Operation:

address $\leftarrow$ OP0 + *SEXT*(IMM, 9)

MEM(address, 16, LSID) $\leftarrow$ OP1

Exceptions:

Execute Exception – Misaligned Store Error

Execute Exception – DTLB Translation Error

Execute Exception – DTLB Protection Error

Store Exception – External Store Error

Notes:

The address must be naturally aligned. If the least-significant bit of the address is non-zero, an Execute Exception occurs.

Although a 64-bit address is computed, the upper 24 bits should always be zero and will be ignored by the processor.

The LSID is used to check for subsequent dependent loads within the same block.

SLL

Shift Left Logical

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	12	T1	T0	G : 2

Description:

The OP0 value is shifted left by an amount specified by the OP1 value. Zero bits are propagated into the vacated bit positions. The six low-order bits of the OP1 value are used to form an unsigned shift amount ranging from 0 to 63.

Operation:

shift_amount ← OP1 & 0x3F

T0, T1 ← *SLL*(OP0, shift_amount)

Exceptions:

None

Notes:

None

SLLI

Shift Left Logical Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	12	IMM	T0	I : 1

Description:

The OP0 value is shifted left by an amount specified by an immediate value. Zero bits are propagated into the vacated bit positions. The six low-order bits of the immediate value are used to form an unsigned shift amount ranging from 0 to 63.

Operation:

shift_amount ← *SEXT*(IMM, 9) & 0x3F

T0 ← *SLL*(OP0, shift_amount)

Exceptions:

None

Notes:

None

SRA

Shift Right Arithmetic

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	14	T1	T0	G : 2

Description:

The OP0 value is shifted right by an amount specified by the OP1 value. The OP0 value is treated as a *signed* integer and its most significant bit (the sign bit) is propagated into the vacated bit positions. The six low-order bits of the OP1 value are used to form an unsigned shift amount ranging from 0 to 63.

Operation:

shift_amount $\leftarrow$ OP1 & 0x3F

T0, T1 $\leftarrow$ *SRA*(OP0, shift_amount)

Exceptions:

None

Notes:

None

SRAI

Shift Right Arithmetic Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	14	IMM	T0	I : 1

Description:

The OP0 value is shifted right by an amount specified by an immediate value. The OP0 value is treated as a *signed* integer and its most significant bit (the sign bit) is propagated into the vacated bit positions. The six low-order bits of the immediate value are used to form an unsigned shift amount ranging from 0 to 63.

Operation:

shift_amount ← *SEXT*(IMM, 9) & 0x3F

T0 ← *SRA*(OP0, shift_amount)

Exceptions:

None

Notes:

None

SRL

Shift Right Logical

Format:

31 25	24 23	22 18	17 9	8 0
23	PR	13	T1	T0

G : 2

Description:

The OP0 value is shifted right by an amount specified by the OP1 value. The OP0 value is treated as an *unsigned* integer and zero bits are propagated into the vacated bit positions. The six low-order bits of the OP1 value are used to form an unsigned shift amount ranging from 0 to 63.

Operation:

shift_amount ← OP1 & 0x3F

T0, T1 ← *SRL*(OP0, shift_amount)

Exceptions:

None

Notes:

None

SRLI

Shift Right Logical Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	13	IMM	T0	I : 1

Description:

The OP0 value is shifted right by an amount specified by an immediate value. The OP0 value is treated as an *unsigned* integer and zero bits are propagated into the vacated bit positions. The six low-order bits of the immediate value are used to form an unsigned shift amount ranging from 0 to 63.

Operation:

```
shift_amount ← SEXT( IMM, 9 ) & 0x3F
T0 ← SRL( OP0, shift_amount )
```

Exceptions:

None

Notes:

None

SUB

Subtract

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	1	T1	T0	G : 2

Description:

> The OP1 value is subtracted from the OP0 value to produce a 64-bit difference. Subtraction is performed using 64-bit 2’s complement arithmetic. Arithmetic overflows are ignored and only the lowest 64 bits of the result are saved.

Operation:

> T0, T1 $\leftarrow$ OP0 – OP1

Exceptions:

> None

Notes:

> The SUB instruction may be used to perform both signed and unsigned subtraction.

SUBI

Sub Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	1	IMM	T0	I : 1

Description:

A sign-extended immediate value is subtracted from the OP0 value to produce a 64-bit difference. Subtraction is performed using 64-bit 2's complement arithmetic. Arithmetic overflows are ignored and only the lowest 64 bits of the result are saved.

Operation:

T0 $\leftarrow$ OP0 – *SEXT*(IMM, 9)

Exceptions:

None

Notes:

The SUBI instruction may be used to perform both signed and unsigned addition.

SW

Store Word

Format:

31 25	24 23	22 18	17 9	8 0	
42	PR	LSID	IMM	0	S : 2

Description:

An address is computed and used to store a word to memory. The low-order word of the OP1 value is stored.

Operation:

address ← OP0 + *SEXT*(IMM, 9)

MEM(address, 32, LSID) ← OP1

Exceptions:

Execute Exception – Misaligned Store Error

Execute Exception – DTLB Translation Error

Execute Exception – DTLB Protection Error

Store Exception – External Store Error

Notes:

The address must be naturally aligned. If any of the two least-significant bits of the address are non-zero, an Execute Exception occurs.

Although a 64-bit address is computed, the upper 24 bits should always be zero and will be ignored by the processor.

The LSID is used to check for subsequent dependent loads within the same block.

TEQ

Test EQ

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	16	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. If the OP0 value is equal to the OP1 value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0, T1 ← *EQ*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

The TEQ instruction may be used to perform both signed and unsigned tests.

TEQI

Test EQ Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	16	IMM	T0	I : 1

Description:

The OP0 value is compared with a sign-extended immediate value. If the OP0 value is equal to the immediate value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0 ← *EQ*(OP0, *SEXT*(IMM, 9)) ? 1 : 0

Exceptions:

None

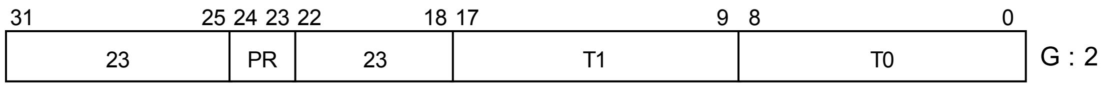
Notes:

The TEQI instruction may be used to perform both signed and unsigned tests.

TGE

Test GE

Format:



Description:

The OP0 value is compared with the OP1 value. The two values are compared as *signed* integers. If the OP0 value is greater than or equal to the OP1 value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0, T1 ← *GE*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

None

TGEI

Test GE Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	23	IMM	T0	I : 1

Description:

The OP0 value is compared with a sign-extended immediate value. The two values are compared as *signed* integers. If the OP0 value is greater than or equal to the immediate value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0 ← *GE*(OP0, *SEXT*(IMM, 9)) ? 1 : 0

Exceptions:

None

Notes:

None

TGEU

Test GE Unsigned

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	25	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. The two values are compared as *unsigned* integers. If the OP0 value is greater than or equal to the OP1 value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0, T1 ← *GEU*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

None

TGEUI

Test GE Unsigned Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	25	IMM	T0	I : 1

Description:

The OP0 value is compared with a sign-extended immediate value. The two values are compared as *unsigned* integers. If the OP0 value is greater than or equal to the immediate value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0 ← *GEU*(OP0, *SEXT*(IMM, 9)) ? 1 : 0

Exceptions:

None

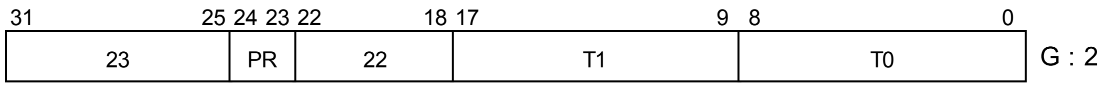
Notes:

Even though the test is unsigned, the IMM field will still be sign-extended. For this instruction, the IMM field may encode an unsigned value between 0 and 255.

TGT

Test GT

Format:



Description:

The OP0 value is compared with the OP1 value. The two values are compared as *signed* integers. If the OP0 value is greater than the OP1 value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0, T1 ← *GT*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

None

TGTI

Test GT Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	22	IMM	T0	I : 1

Description:

The OP0 value is compared with a sign-extended immediate value. The two values are compared as *signed* integers. If the OP0 value is greater than the immediate value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0 ← *GT*(OP0, *SEXT*(IMM, 9)) ? 1 : 0

Exceptions:

None

Notes:

None

TGTU

Test GT Unsigned

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	24	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. The two values are compared as *unsigned* integers. If the OP0 value is greater than the OP1 value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0, T1 $\leftarrow$ *GTU*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

None

TGTUI

Test GT Unsigned Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	24	IMM	T0	I : 1

Description:

The OP0 value is compared with a sign-extended immediate value. The two values are compared as *unsigned* integers. If the OP0 value is greater than the immediate value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0 ← *GTU*(OP0, *SEXT*(IMM, 9)) ? 1 : 0

Exceptions:

None

Notes:

Even though the test is unsigned, the IMM field will still be sign-extended. For this instruction, the IMM field may encode an unsigned value between 0 and 255.

TLE

Test LE

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	17	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. The two values are compared as *signed* integers. If the OP0 value is less than or equal to the OP1 value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0, T1 $\leftarrow$ *LE*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

None

TLEI

Test LE Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	17	IMM	T0	I : 1

Description:

The OP0 value is compared with a sign-extended immediate value. The two values are compared as *signed* integers. If the OP0 value is less than or equal to the immediate value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0 ← *LE*(OP0, *SEXT*(IMM, 9)) ? 1 : 0

Exceptions:

None

Notes:

None

TLEU

Test LE Unsigned

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	19	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. The two values are compared as *unsigned* integers. If the OP0 value is less than or equal to the OP1 value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0, T1 ← *LEU*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

None

TLEUI

Test LE Unsigned Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	19	IMM	T0	I : 1

Description:

The OP0 value is compared with a sign-extended immediate value. The two values are compared as *unsigned* integers. If the OP0 value is less than or equal to the immediate value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0 $\leftarrow$ *LEU*(OP0, *SEXT*(IMM, 9)) ? 1 : 0

Exceptions:

None

Notes:

Even though the test is unsigned, the IMM field will still be sign-extended. For this instruction, the IMM field may encode an unsigned value between 0 and 255.

TLT

Test LT

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	18	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. The two values are compared as *signed* integers. If the OP0 value is less than the OP1 value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0, T1 ← *LT*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

None

TLTI

Test LT Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	18	IMM	T0	I : 1

Description:

The OP0 value is compared with a sign-extended immediate value. The two values are compared as *signed* integers. If the OP0 value is less than the immediate value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0 ← *LT*(OP0, *SEXT*(IMM, 9)) ? 1 : 0

Exceptions:

None

Notes:

None

TLTU

Test LT Unsigned

Format:

31 – 25	24 23	22 – 18	17 – 9	8 – 0	
23	PR	20	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. The two values are compared as *unsigned* integers. If the OP0 value is less than the OP1 value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0, T1 ← *LTU*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

None

TLTUI

Test LT Unsigned Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	20	IMM	T0	I : 1

Description:

The OP0 value is compared with a sign-extended immediate value. The two values are compared as *unsigned* integers. If the OP0 value is less than the immediate value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0 ← *LTU*(OP0, *SEXT*(IMM, 9)) ? 1 : 0

Exceptions:

None

Notes:

Even though the test is unsigned, the IMM field will still be sign-extended. For this instruction, the IMM field may encode an unsigned value between 0 and 255.

TNE

Test NE

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	21	T1	T0	G : 2

Description:

The OP0 value is compared with the OP1 value. If the OP0 value is NOT equal to the OP1 value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0, T1 ← *NE*(OP0, OP1) ? 1 : 0

Exceptions:

None

Notes:

None

TNEI

Test NE Immediate

Format:

31 25	24 23	22 18	17 9	8 0
31	PR	21	IMM	T0

I : 1

Description:

The OP0 value is compared with a sign-extended immediate value. If the OP0 value is NOT equal to the immediate value, then a 1 (true) is produced. Otherwise, a 0 (false) is produced.

Operation:

T0 $\leftarrow$ *NE*(OP0, *SEXT*(IMM, 9)) ? 1 : 0

Exceptions:

None

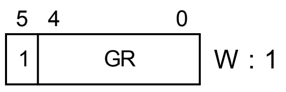
Notes:

None

WRITE

Write General Register

Format:



Description:

The specified General Register is written.

Operation:

```
reg_bank ← WRITE_ID % 4
reg_number ← 4 * GR + reg_bank
REG( reg_number ) ← DATA
```

Exceptions:

None

Notes:

This instruction may only be executed from the Write Queue.

The WRITE instruction may only modify a General Register from the corresponding General Register bank.

XOR

Bitwise XOR

Format:

31 25	24 23	22 18	17 9	8 0	
23	PR	10	T1	T0	G : 2

Description:

A bitwise logical XOR operation is performed using the OP0 and OP1 values.

Operation:

T0, T1 ← OP0 ^ OP1

Exceptions:

None

Notes:

None

XORI

Bitwise XOR Immediate

Format:

31 25	24 23	22 18	17 9	8 0	
31	PR	10	IMM	T0	I : 1

Description:

A bitwise logical XOR operation is performed using the OP0 value and a sign-extended immediate value.

Operation:

T0 ← OP0 ^ *SEXT*(IMM, 9)

Exceptions:

None

Notes:

An XORI with an immediate value of -1 may be used to perform an bitwise complement (the NOT operation).

Chapter 9 - Performance Monitor

9.1 Overview

The TRIPS processor includes a built-in Performance Monitor to help track a small set of performance-critical events. Several count registers are implemented, as well as a handful of special address registers. The Performance Monitor is always enabled and does not affect normal processing operations. It's registers may be accessed just like the other registers described in Chapter 4.

9.2 Register Map

The following table summarizes the Performance Monitor registers. Each table entry shows the register's address offset , size, name, and associated information. The registers are grouped based upon where they are implemented within the processor.

Offset	Bytes	Registers	Information
0x2000	4	PMON_DT0_CTR0	Data Cache Load Accesses
0x2004	4	PMON_DT0_CTR1	Data Cache Load Misses
0x2008	4	PMON_DT0_CTR2	Data Cache Store Accesses
0x200C	4	PMON_DT0_CTR3	Data Cache Store Misses
0x2010	4	PMON_DT0_CTR4	Data Cache Line Fills
0x2014	4	PMON_DT0_CTR5	Data Cache Line Spills
0x2018	4	PMON_DT0_CTR6	Load Predictor Load Deferrals
0x2040	4	PMON_DT1_CTR0	Data Cache Load Accesses
0x2044	4	PMON_DT1_CTR1	Data Cache Load Misses
0x2048	4	PMON_DT1_CTR2	Data Cache Store Accesses

Offset	Bytes	Registers	Information
0x204C	4	PMON_DT1_CTR3	Data Cache Store Misses
0x2050	4	PMON_DT1_CTR4	Data Cache Line Fills
0x2054	4	PMON_DT1_CTR5	Data Cache Line Spills
0x2058	4	PMON_DT1_CTR6	Load Predictor Load Deferrals
0x2080	4	PMON_DT2_CTR0	Data Cache Load Accesses
0x2084	4	PMON_DT2_CTR1	Data Cache Load Misses
0x2088	4	PMON_DT2_CTR2	Data Cache Store Accesses
0x208C	4	PMON_DT2_CTR3	Data Cache Store Misses
0x2090	4	PMON_DT2_CTR4	Data Cache Line Fills
0x2094	4	PMON_DT2_CTR5	Data Cache Line Spills
0x2098	4	PMON_DT2_CTR6	Load Predictor Load Deferrals
0x20C0	4	PMON_DT3_CTR0	Data Cache Load Accesses
0x20C4	4	PMON_DT3_CTR1	Data Cache Load Misses
0x20C8	4	PMON_DT3_CTR2	Data Cache Store Accesses
0x20CC	4	PMON_DT3_CTR3	Data Cache Store Misses
0x20D0	4	PMON_DT3_CTR4	Data Cache Line Fills
0x20D4	4	PMON_DT3_CTR5	Data Cache Line Spills
0x20D8	4	PMON_DT3_CTR6	Load Predictor Load Deferrals
0x2100	4	PMON_GT_CTR0	Active Cycles
0x2104	4	PMON_GT_CTR1	Block Fetches
0x2108	4	PMON_GT_CTR2	Block Commits
0x210C	4	PMON_GT_CTR3	Block Flushes
0x2110	4	PMON_GT_CTR4	Inst Cache Misses
0x2114	4	PMON_GT_CTR5	Branch Predictor Guesses

Offset	Bytes	Registers	Information
0x2118	4	PMON_GT_CTR6	Branch Predictor Mispredicts
0x211C	4	PMON_GT_CTR7	Load Misordering Flushes
0x2120	8	PMON_GT_ADDR0	Inst Cache Miss Address
0x2128	8	PMON_GT_ADDR1	Branch Mispredict Address
0x2130	8	PMON_GT_ADDR2	Load Misordering Address

9.3 Register Descriptions

The Performance Monitor includes two types of registers – count registers and address registers. These are described in separate sections below.

9.3.1 Count Registers

Each count register is 32 bits wide and its value increments by one when the corresponding event occurs. If a count value reaches the maximum value (0xFFFFFFFF), it will roll over to zero during the next increment. (The system software should halt the processor and gather the counts frequently enough to prevent this.) Count registers are read-only, but their values are automatically reset to zero after a read (and during power-on).

Each count is described (in alphabetical order) below. Some of them (those collected in the Data Tile) are tracked separately for each Data Cache bank.

- Active Cycles – Incremented for every cycle that the processor spends active (not halted).
- Block Commits – The number of program blocks that were executed and committed.
- Block Fetches – The number of program blocks that were fetched for execution.
- Block Flushes – The number of internal block flush events that occurred due to a branch misprediction, load misordering, or exception. (This may be less than the number of flushed blocks.)

- Branch Predictor Guesses – The number of committed blocks for which a branch prediction was made. This excludes committed blocks for which the branch predictor was inhibited due to XFLAGS.
- Branch Predictor Mispredicts – The number of branch predictor guesses (as defined above) that were incorrect.
- Data Cache Line Fills – The number of cache lines written into the associated Data Cache bank (result from a load).
- Data Cache Line Spills – The number of dirty cache lines written back to memory from the associated Data Cache bank (resulting from a cache line replacement). This excludes spills due to a cache flush operation.
- Data Cache Load Accesses – The number of load operations that hit or missed in the associated Data Cache bank. This excludes loads that arrive with null or exception tokens. It includes speculative loads.
- Data Cache Load Misses – The number of load accesses (as defined above) that missed in the associated Data Cache bank. This includes accesses to uncacheable memory segments. (This may be less than the number of external load requests due to request merging.)
- Data Cache Store Accesses – The number of store operations that hit or missed in the associated Data Cache bank. Since store operations are not performed speculatively, this includes only unnullified stores from committed blocks.
- Data Cache Store Misses – The number of store accesses (as defined above) that missed in the associated Data Cache bank. (This may be less than the number of external store requests due to request merging.)
- Inst Cache Misses – The number of program block fetches (as defined above) that missed in the Instruction Cache. This includes fetches from uncacheable memory segments. It excludes fetches that resulted in fetch exceptions.
- Load Misordering Flushes – The number of committed blocks for which a load misordering flush occurred.

- Load Predictor Load Deferrals – The number of load operations to the associated Data Cache bank that were deferred (delayed) due to a predicted store dependence. This excludes loads for which the load predictor was inhibited due to XFLAGS. It also excludes loads that are being reexecuted due to a misordering flush.

9.3.2 Address Registers

The address registers are loaded with address information whenever the corresponding event occurs. When the processor halts, they hold sampled information associated with the most recent event. Address registers are read-only, but their values are automatically reset to zero after a read (and during power-on).

Each sampled address is described (in alphabetical order) below.

- Branch Mispredict Address – The threadslot identifier plus virtual address of the program block that caused the most recent branch predictor mispredict event (as defined above). (This is not the mispredicted target address.)
- Instruction Cache Miss Address – The threadslot identifier plus virtual address of the program block that caused the most recent Instruction Cache miss event (as defined above).
- Load Misordering Address – The threadslot identifier plus virtual address of the program block that caused the most recent load misordering flush (as defined above). (This is not the target address of the misordered load.)

All address registers use the following format. Bits [39:0] hold the virtual address. Bits [41:40] hold the threadslot identifier. The remaining bits are reserved.