EE Core User's Manual

Publication date: April 2002

Sony Computer Entertainment Inc.
1-1, Akasaka 7-chome, Minato-ku
Tokyo 107-0052 Japan

Sony Computer Entertainment America
919 East Hillsdale Blvd.
Foster City, CA 94404, U.S.A.

Sony Computer Entertainment Europe
30 Golden Square
London W1F 9LD, U.K.

About This Manual

The "EE Core User's Manual" describes the EE Core (the CPU core unit), which controls the entire Emotion Engine, and its general operations. It provides overviews of the configuration and instruction set and the functions of the main blocks. For details of the instruction set, refer to the "EE Core Instruction Set Manual".

- Chapter 1 "Architecture Overview" describes the EE Core's features and configuration, pipeline operation, and the main blocks and functions.
- Chapter 2 "Instruction Set Overview" provides an overview of the instruction set, and describes data alignment and characteristic operations such as branch delay.
- Chapter 3 "Registers" describes the registers of the EE Core and the System Control Coprocessor (COP0).
- Chapter 4 "Exception Processing" describes exceptions, the events that suspend execution of instructions. This chapter also describes preprocessor resetting as one of the exceptions.
- Chapter 5 "Memory Management" describes the virtual and physical spaces, conversion from the virtual address to the physical address, cache mode, and the System Control Coprocessor (COP0) that governs them.
- Chapter 6 "Caches" describes the EE Core's instruction cache, data cache, and scratchpad RAM.
- Chapter 7 "Performance Counters and Instruction Stepping" describes the functions and means for monitoring and counting the internal events of the EE Core.
- Chapter 8 "Floating-Point Unit (FPU)" describes the floating-point operation unit connected as a coprocessor.
- Chapter 9 "Hardware Breakpoint" describes breakpoint function, a part of the COP0 functions.

Changes Since Release of 5th Edition

Since release of the 5th Edition of the EE Core User’s Manual, the following changes have been made. Note that each of these changes is indicated by a revision bar in the margin of the affected page.

Ch. 1: Architecture Overview

- A correction has been made to section 1.1. Features of the EE Core, on page 14.
- A correction has been made to “C2 Pipe” in section 1.2.6. Physical Pipes, on page 16.

Ch. 2: Instruction Set Overview

- A correction has been made to the “JALR” row in the Branch/Jump Instructions table on page 37.
- A correction has been made to section 2.3. Load/Store Instructions, on page 39.
- A correction has been made to the “MTC1” row in Table 2-7 Coprocessor Instructions, on page 50.

Ch. 3: Registers

- A correction has made to section 3.2. System Control Coprocessor (COP0) Registers, on page 62.
- A correction has been made to the description of the Context register on page 66.
- A correction has been made to Figure 3-2 Wired TLB Entries on page 68.
- A correction has been made to the “CU (CU[3:01])” row in the COP0 Status table on page 73.
- A correction has been made to the figure in the Cause register description on page 75.
- Corrections have been made to the description of the DAB/DABM register on page 82.

Ch. 4: Exception Processing

- A correction has been made to the table in section 4.2.2. NMI Exception, on page 96.
- A correction has been made to “Operation” in section 4.2.7. TLB Refill Exception, on page 101.
- A correction has been made to “Operation” in section 4.2.8. TLB Invalid Exception, on page 102.

- A correction has been made to “Programming Notes” in section 4.2.13. Reserved Instruction Exception, on page 107.

Ch. 5: Memory Management

- Corrections have been made to “seeg (Supervisor Mode – Supervisor Space)” in section 5.1.5. Supervisor Mode Address Space, on page 115.
- A correction has been made to the “MASK” row of the table in section 5.2.4. TLB Entry, on page 124.
- Corrections have been made to the “Move Instruction” rows of Table 8-3 FPU Instructions, on page 161.

Ch. 9: Hardware Breakpoint

- A correction has been made to the example code in section 9.3.4. Setting a Data Value Breakpoint, on page 177.

Glossary

Term	Definition
EE	Emotion Engine. CPU of the PlayStation 2.
EE Core	Generalized computation and control unit of EE. Core of the CPU.
COP0	EE Core system control coprocessor.
COP1	EE Core floating-point operation coprocessor. Also referred to as FPU.
COP2	Vector operation unit coupled as a coprocessor of EE Core. VPU0.
GS	Graphics Synthesizer. Graphics processor connected to EE.
GIF	EE Interface unit to GS.
IOP	Processor connected to EE for controlling input/output devices.
SBUS	Bus connecting EE to IOP.
VPU (VPU0/VPU1)	Vector operation unit. EE contains 2 VPUs: VPU0 and VPU1.
VU (VU0/VU1)	VPU core operation unit.
VIF (VIF0/VIF1)	VPU data decompression unit.
VIFcode	Instruction code for VIF.
SPR	Quick-access data memory built into EE Core (Scratchpad memory).
IPU	EE Image processor unit.
word	Unit of data length: 32 bits
qword	Unit of data length: 128 bits
Slice	Physical unit of DMA transfer: 8 qwords or less
Packet	Data to be handled as a logical unit for transfer processing.
Transfer list	A group of packets transferred in serial DMA transfer processing.
Tag	Additional data indicating data size and other attributes of packets.
DMAtag	Tag positioned first in DMA packet to indicate address/size of data and address of the following packet.
GS primitive	Data to indicate image elements such as point and triangle.
Context	A set of drawing information (e.g. texture, distant fog color, and dither matrix) applied to two or more primitives uniformly. Also referred to as the drawing environment.
GIFtag	Additional data to indicate attributes of GS primitives.
Display list	A group of GS primitives to indicate batches of images.

(This page is left blank intentionally)

Contents

1. Architecture Overview 13
1.1. Features of the EE Core 14
1.2. Block Diagram and Functional Block Description 15
1.2.1. PC Unit 16
1.2.2. MMU 16
1.2.3. Caches and Scratchpad RAM 16
1.2.4. Issue Logic and Staging Registers 16
1.2.5. GPR (General Purpose Registers) and FPR (Floating-Point Registers) 16
1.2.6. Physical Pipes 16
1.2.7. Operand/Bypass logic 17
1.2.8. Writeback Buffer 17
1.2.9. UCAB 17
1.2.10. Result and Move Buses 17
1.2.11. Bus Interface Unit 17
1.3. Superscalar Pipeline Operation 18
1.3.1. Interlock by Data Hazards 18
1.3.2. Integer Instruction Pipeline Stages 18
1.3.3. COP1 Pipeline 21
1.3.4. COP2 Pipeline 23
1.3.5. Classification and Routing of Instructions 23
1.3.6. Instruction Issue Combinations 24
1.4. Registers 27
1.4.1. CPU Registers 27
1.4.2. FPU Registers 27
1.4.3. COP0 Registers 27
1.5. Memory Management 28
1.6. Cache Memory and Scratchpad RAM 29
1.7. Bus Interface 30
1.8. Floating-Point Unit 31
1.9. Debug Support Functions 32
2. Instruction Set Overview 33
2.1. Binary Formats 34
2.2. Instruction Set Summary 35
2.2.1. Instruction Set List 35
2.2.2. MIPS III Instructions not Supported by EE Core 38
2.3. Load/Store Instructions 39
2.3.1. Data Formats and Alignment 39
2.3.2. Load Delay 41
2.4. Computational Instructions 42
2.5. Branch/Jump Instructions 44

2.5.1. Branch Delay Slot 44
2.5.2. Overview of Jump Instructions 45
2.5.3. Overview of Branch Instructions 45
2.6. Exception Instructions 46
2.7. Serialization Instruction 47
2.8. MIPS IV Instructions 48
2.9. System Control Coprocessor (COP0) Instructions 49
2.10. Coprocessor Instructions (COP1/COP2) 50
2.10.1. COP1(FPU) Instructions 50
2.10.2. COP2 Instructions 51
2.10.3. VU Macro instructions 51
2.11. EE Core-Specific Instructions 54
2.11.1. EE Core-Specific Multiply / Divide Instructions 54
2.11.2. Multimedia Instructions 55
2.12. Latency 58
3. Registers 59
3.1. CPU Registers 60
3.1.1. General Purpose Registers 61
3.1.2. HI and LO Registers 61
3.1.3. SA Register 61
3.1.4. Program Counter (PC) 61
3.2. System Control Coprocessor (COP0) Registers 62
Index : Index that specifies TLB entry for reading or writing 63
Random : Index that specifies TLB entry for the TLBWR instruction 64
EntryLo0 / EntryLo1 : Lower part of the TLB entry 65
Context : TLB miss handling information 66
PageMask : Page size comparison mask 67
Wired : The number of Wired TLB entries 68
BadVAddr : Virtual address that causes an error 69
Count : Timer count value 70
EntryHi : Upper parts of a TLB entry 71
Compare : Timer stable value 72
Status : COP0 Status 73
Cause : Cause of the most recent exception 75
EPC : Address generating exceptions 76
PRId : Processor Revision 77
Config : Processor Configuration 78
BadPAddr : Physical address that caused an error 79
BPC : Control of the breakpoint function 80
IAB / IABM : Instruction address breakpoint settings 81
DAB / DABM : Data address breakpoint settings 82
DVB / DVBM : Data value breakpoint settings 83
PCCR : Performance Counter Control 84
PCR0 / PCR1 : Performance Counter 85
TagLo : Lower parts of a cache tag 86
TagHi : Upper parts of a cache tag 87

ErrorEPC : Address generating a level 2 exception 88
4. Exception Processing 89
4.1. Exception Handling Process 90
4.1.1. Exception Vector 90
4.1.2. Level 1 Exception Handling 90
4.1.3. Level 2 Exception Handling 92
4.1.4. Exception Priority 93
4.2. Exception Reference 94
4.2.1. Reset Exception 95
4.2.2. NMI Exception 96
4.2.3. Performance Counter Exception 97
4.2.4. Debug Exception 98
4.2.5. Interrupt Exception 99
4.2.6. TLB Modified Exception 100
4.2.7. TLB Refill Exception 101
4.2.8. TLB Invalid Exception 102
4.2.9. Address Error Exception 103
4.2.10. Bus Error Exception 104
4.2.11. System Call Exception 105
4.2.12. Break Exception 106
4.2.13. Reserved Instruction Exception 107
4.2.14. Coprocessor Unusable Exception 108
4.2.15. Trap Exception 109
4.2.16. Overflow Exception 110
5. Memory Management 111
5.1. Address Space 112
5.1.1. Physical Address Space 112
5.1.2. Virtual Address Space 112
5.1.3. Operating Modes and Address Space 113
5.1.4. User Mode Address Space 114
5.1.5. Supervisor Mode Address Space 115
5.1.6. Kernel Mode Address Space 116
5.2. Address Translation 118
5.2.1. Overview of Address Translation 118
5.2.2. Address Translation Look-aside Buffer (TLB) 120
5.2.3. Address Translation Process Flow 121
5.2.4. TLB Entry 123
5.2.5. Scratchpad RAM Mapping 125
5.2.6. Cache Mode 126
5.3. System Control Coprocessor 127
5.3.1. TLB Related Register 127
5.3.2. TLB Operation Instructions 127
6. Caches 129
6.1. Cache and SPRAM Features 130
6.2. Organization of the Caches 131
6.2.1. Organization of the Data Cache 131

6.2.2. Organization of the Instruction Cache 131
6.2.3. Tag Structure 132
6.3. Cache Operations 134
6.3.1. Line Replacement Algorithm 134
6.3.2. Non-blocking Loads and Hit Under Miss 134
6.3.3. Cache Hit and Miss Operations 135
6.3.4. Data Cache Writeback 135
6.3.5. Data Cache State Transitions 136
6.3.6. Instruction Cache State Transitions 137
6.3.7. Data Cache Lock Function 137
6.3.8. Relationship between Cached and Uncached Operations 138
6.3.9. Data Consistency between Cache and SPRAM 138
6.4. Scratchpad RAM (SPRAM) 139
6.4.1. SPRAM Overview 139
6.4.2. DMA Access to SPRAM 139
6.4.3. SPRAM Mapping 140
6.5. Uncached Accelerated Buffer (UCAB) 141
6.5.1. UCAB Overview 141
6.5.2. Non-Blocking Loads and Hit Under Miss 141
6.6. Cache Control Registers 142
7. Performance Counters and Instruction Stepping 143
7.1. Configuration of Performance Counter 144
7.1.1. Performance Counter Control Registers (PCCR) 144
7.1.2. Counter Registers (PCR0 / PCR1) 145
7.1.3. Access to the Performance Counter Registers 145
7.1.4. Initial Value of the Performance Counter Registers 145
7.2. Performance Counter Operation Details 146
7.2.1. Counter Increment 146
7.2.2. Counter Event 146
7.2.3. Counter Event Descriptions 147
7.2.4. Occurrence of Counter Exceptions 150
7.2.5. Priority of Counter Exceptions 150
7.2.6. Initializing Performance Counters 151
7.2.7. Notes on Pipelining 151
7.2.8. Notes on Instruction Stepping 152
8. Floating-Point Unit (FPU) 155
8.1. Data Formats 156
8.1.1. Floating-Point Format 156
8.1.2. Fixed-Point Format 156
8.2. FPU Registers 157
8.3. FPU Control Registers 158
8.3.1. Implementation and Revision Register (FCR0) 158
8.3.2. Control/Status Register (FCR31) 158
8.4. Instruction Set Overview 160
8.5. Results of Abnormal Computation 162
8.6. Sign of Zero 163

8.7. Rounding....164
8.8. IEEE 754 Compatibility....165
9. Hardware Breakpoint....167
9.1. Overview of Hardware Breakpoint....168
9.2. Breakpoint Registers....169
9.2.1. Breakpoint Control (BPC) Register....169
9.2.2. Instruction Address Registers (IAB / IABM)....170
9.2.3. Data Address Registers (DAB / DABM)....170
9.2.4. Data Value Registers (DVB / DVBM)....171
9.2.5. Establishment of Breakpoint and Operation of Exception Generation....171
9.3. Setting Breakpoints....174
9.3.1. Procedure for Setting Breakpoints....174
9.3.2. Setting an Instruction Breakpoint....175
9.3.3. Setting a Data Address Breakpoint....176
9.3.4. Setting a Data Value Breakpoint....177
9.4. Outputting Trigger Signals....179
9.5. Notes on Breakpoint Processing....180

(This page is left blank intentionally)

1. Architecture Overview

This chapter provides an overview of the EE Core architecture, focusing on the following items:

- Block diagram and functional block
- Superscalar pipeline operation
- Instruction set
- Registers
- Memory Management
- Cache Memory and Scratchpad RAM
- Bus interface
- Floating-Point Unit
- Performance Monitors
- Debug Support

1.1. Features of the EE Core

The EE Core is a superscalar processor with a subset of 64-bit MIPS IV Instruction Set Architecture. It implements a large extension to the instruction set tailored for multimedia applications.
It contains a CPU, a floating-point execution unit (FPU = Coprocessor 1), instruction and data caches, a scratchpad RAM and a vector operation coprocessor (VPU0= Coprocessor 2).
It has two pipelines. Two instructions can be decoded each cycle. These instructions are executed and completed in order. However, since Data Cache misses are non-blocking and a single cache miss does not stall the pipeline, load misses or uncached loads may be retired out-of-order. Multiply, Multiply-Accumulate, Divide, Prefetch and Coprocessor instructions are also retired out-of-order.

These features of the EE Core are summarized as follows:

- 2-way superscalar pipeline
- 128-bit (two 64-bit) data path and 128-bit system bus
- Instruction set
  - 64-bit MIPS III instruction set (excluding some instructions)
  - Selected MIPS IV instructions (Prefetch and Move conditional instructions)
  - Non-blocking load instructions
  - Three-operand Multiply and Multiply-Accumulate instructions
  - 128-bit multimedia instructions which configure the 128-bit data path as two 64-bit, four 32-bit, eight 16-bit or sixteen 8-bit paths
  - Little endian
- Branch predictions by Branch History Table (BHT) and Branch Target Address Cache (BTAC)
- On-chip caches and scratchpad RAM
  - Instruction cache: 16 KB, 2-way set associative
  - Data cache: 8 KB, 2-way set associative (with write-back protocol)
  - Data scratchpad RAM: 16 KB
  - Cache line: 64 bytes
  - Data cache line locking
  - Prefetch functions
- Fast integer Multiply and Multiply-Accumulate operations
- Memory management unit
  - 32-bit physical address space
  - 32-bit virtual address space
  - 48-entry (96-page) full set associative address translation look-aside buffer (TLB)
- Performance counter features
- Debug support features

1.2. Block Diagram and Functional Block Description

A block diagram of the EE Core is shown below.

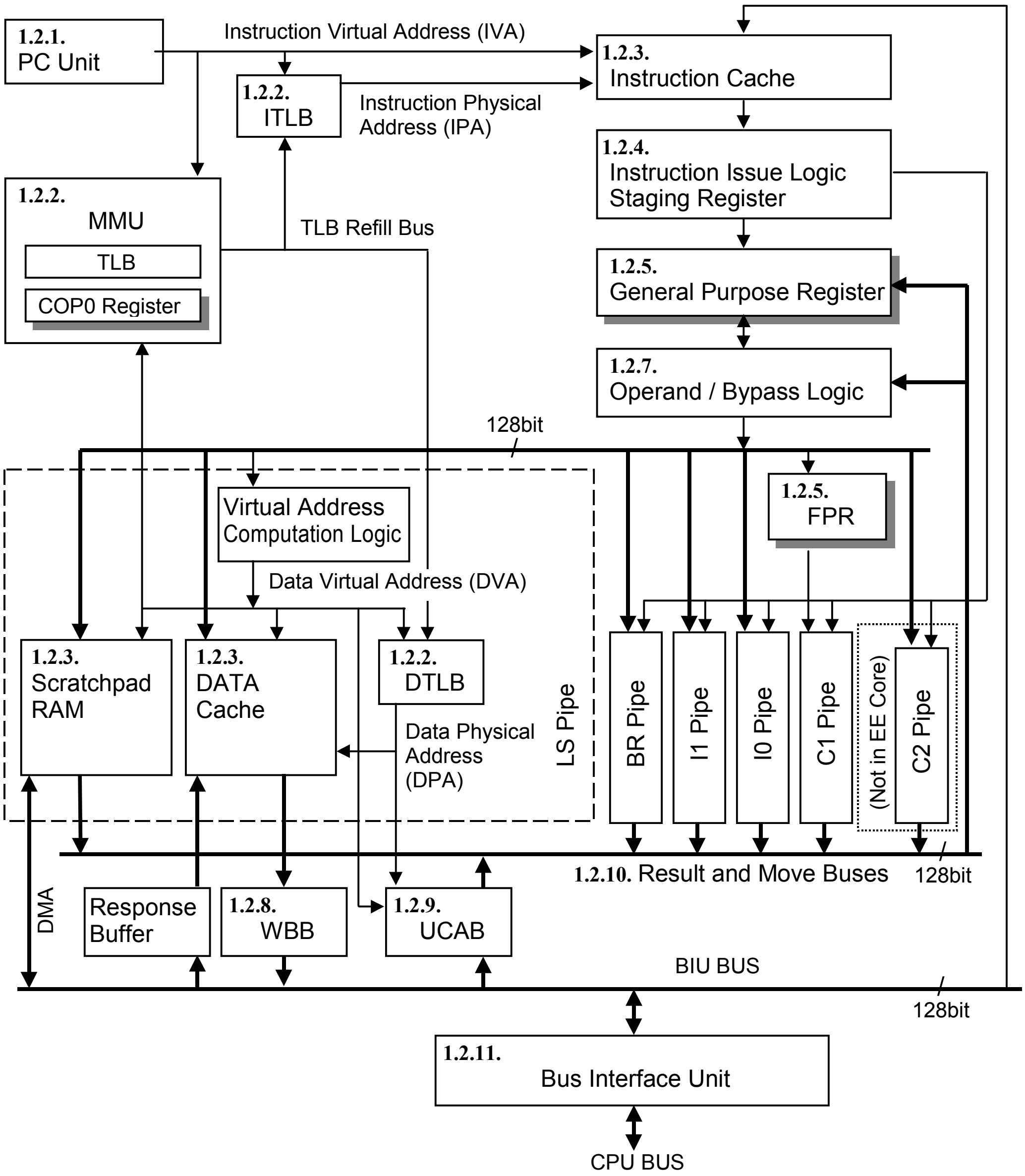


Figure 1-1 EE Core Block Diagram

1.2.1. PC Unit

The 32-bit Program Counter (PC) holds the address of the instruction that is being executed. It contains a 64-entry Branch Target Address Cache (BTAC), which is used for branch predictions.

1.2.2. MMU

The Memory Management Unit (MMU) supports the address translation functions of the CPU. It sends data to the DTLB (Data address Translation Lookaside Buffer) and ITLB (Instruction address Translation Lookaside Buffer) via the TLB Refill Bus. For details of the MMU, refer to "5. Memory Management".

1.2.3. Caches and Scratchpad RAM

The instruction cache, the data cache and Scratchpad RAM are described in "6. Caches". For each branch instruction present in the Instruction Cache, two bits of branch history are stored in the Branch History Table (BHT). Data is transferable via DMA between scratchpad RAM and main memory.

1.2.4. Issue Logic and Staging Registers

The issue logic decides which pipes to execute instructions. This unit places a maximum of two instructions in appropriate pipes each cycle. For details, refer to "1.3. Superscalar Pipeline Operation".

1.2.5. GPR (General Purpose Registers) and FPR (Floating-Point Registers)

The General Purpose Registers and the Floating-Point Registers are described in Section "1.4. Registers".

1.2.6. Physical Pipes

The physical pipes execute operations of instructions. The EE Core has 6 physical pipes, described below.

I0 and I1 Pipes

The I0 and I1 pipes contain logic to support integer arithmetic. Both are composed of a complete 64-bit ALU, Shifter and Multiply-Accumulate unit. The I0 pipeline contains the SA register used for funnel shift operations. The I1 pipeline contains a LZC (leading zero counting) unit. Furthermore, the two pipelines share a single 128-bit multimedia shifter.
These are configured dynamically into a single 128-bit execution pipe per instruction to execute the 128-bit Multimedia ALU, Shift and MAC instructions.

LS Pipe

The LS Pipe (Load/Store Pipe) contains logic to support 128-bit Load and Store instructions.

BR Pipe

The BR Pipe (Branch Pipe) contains logic to execute a Branch instruction.

C1 Pipe

The C1 Pipe contains logic to support a Floating-Point coprocessor unit (COP1=FPU).

C2 Pipe

The C2 Pipe contains logic to support a customer-specific coprocessor unit (COP2=VPU0).

1.2.7. Operand/Bypass logic

The Operand/Bypass logic is a unit that takes data from the GPRs, Result Bus and Move Bus and routes the data to the pipelines and Scratchpad RAM.

1.2.8. Writeback Buffer

The Writeback Buffer (WBB) is an 8-entry by 16-byte (1-qword) FIFO storing data prior to accessing the CPU bus. It increases performance by decoupling the processor from the latencies of the CPU bus. The WBB also has a function for gathering uncached accelerated stores, that is, for gathering sequential stores up to 1 qword.

1.2.9. UCAB

The Uncached Accelerated Buffer (UCAB) is a 8-qword buffer. It caches 128 sequential bytes of data during an uncached accelerated load miss. If the address hits in the UCAB, the loads from the uncached accelerated space get the data from this buffer.

1.2.10. Result and Move Buses

The Result and Move Buses convey data between execution units, scratchpad RAM, the Data Cache and the Operand/Bypass logic.

1.2.11. Bus Interface Unit

The Bus Interface Unit (BIU) connects the EE Core to the rest of the system. It combines the Core's internal bus signals with the CPU Bus.

1.3. Superscalar Pipeline Operation

The EE Core has a six-stage superscalar pipeline. It can fetch, decode and execute a maximum of two instructions in parallel each cycle.
This section discusses in more detail the six physical pipelines described above. It also discusses how instructions are routed among pipes.

1.3.1. Interlock by Data Hazards

The following pipelines are interlocked when a register-related data hazard occurs. If the succeeding instruction attempts to read the same register while executing an instruction to write any of the general-purpose, HI, LO, SA, program counter, or coprocessor registers, the pipeline stalls until the write operation by the preceding instruction finishes.

1.3.2. Integer Instruction Pipeline Stages

The EE Core contains four integer pipelines: the I0 and I1 pipes, and the LS and BR pipes. Each pipe consists of the following six stages, with each stage having 2 phases:

Symbol	Stage	Phase
1I	Instruction Fetch	Phase 1
2I	Instruction Fetch	Phase 2
1Q	Instruction Queue	Phase 1
2Q	Instruction Queue	Phase 2
1R	Register Fetch	Phase 1
2R	Register Fetch	Phase 2
1A	Execution	Phase 1
2A	Execution	Phase 2
1D	Data Fetch	Phase 1
2D	Data Fetch	Phase 2
1W	Write-back	Phase 1
2W	Write-back	Phase 2

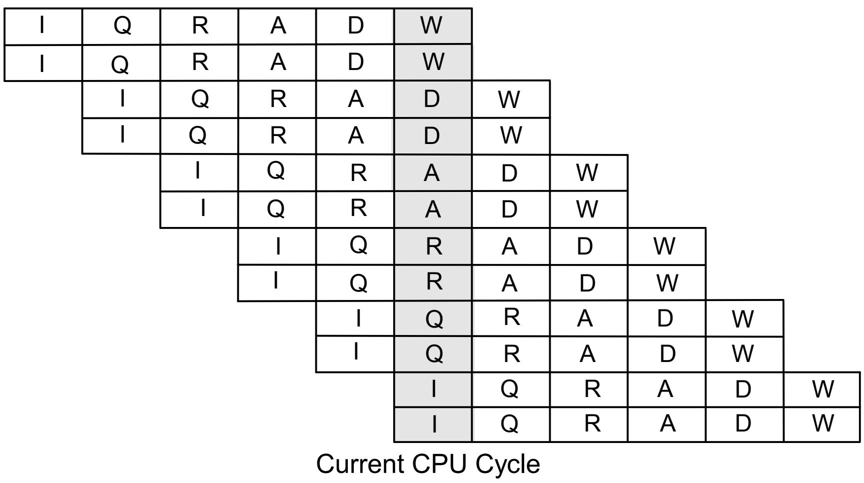


Figure 1-2 EE Core Integer Instruction Pipeline

Operations performed in each stage and phase are described as follows.

1I: Instruction Fetch, Phase 1

- The sequential address is calculated
- The branch address is calculated

2I: Instruction Fetch, Phase 2

- Selection of instruction addresses. Selects one of the following as the instruction address to be executed:
  - Sequential address
  - Branch/Jump address
  - Predicted Branch Target address from the BTAC
  - Exception vector address
  - EPC or ErrorEPC

1Q: Instruction Queue, Phase 1

- The instruction address translation look-aside buffer (ITLB) performs the virtual-to-physical address translation
- The Instruction Cache (data, Tag, S bits and BHT) fetch begins
- The BTAC read begins
- TLB read for instruction fetch starts

2Q: Instruction Queue, Phase 2

- The Instruction Cache fetch is completed
- TLB read completes
- The Instruction Cache Tag hit check is determined and either the cache or memory is selected
- The instructions are selected based on the S bit (SPRAM selection bit)
- The BTAC read is completed and, if there is a hit, an appropriate predicted target address is output

1R: Register Fetch, Phase 1

- Instructions are passed to the appropriate execution units
- The register file read is started
- Execution unit structural hazards are determined

2R: Register Fetch, Phase 2

- Instructions are decoded, data dependencies are determined and the appropriate instructions are issued
- The register file read is completed

1A: Execution, Phase 1

- Bypassing from D or W stage

2A: Execution, Phase 2

- The execution unit starts executing an instruction
- The integer arithmetic, logical, shift and multimedia instructions are completed
- The iterative steps of the Multiply, Multiply-Accumulate, or Divide instructions are executed
- The virtual address for load and store instructions is calculated
- The branch condition is determined
- The DTLB read starts
- The Data Cache, Scratchpad RAM, or UCAB read starts

1D: Data Fetch, Phase 1

- The DTLB read finishes
- The TLB read for a data access starts
- The Data Cache, Scratchpad RAM, or UCAB read is completed
- Data Cache Tag checking is completed
- Load or register data is obtained from COP1 and COP2
- COP0 registers are read

2D: Data Fetch, Phase 2

- The TLB read for a data access finishes
- Data alignment and way are selected for reading from the Data Cache
- Data alignment is done for reading from the Scratchpad RAM
- Data sign extension is done
- BHT bits and BTAC are updated
- Exceptions are detected

1W: Writeback, Phase 1

- The Data Cache or scratchpad RAM is written.
- Data is transferred to COP1 and COP2

2W: Writeback, Phase 2

- Results are written to the register file
- The COP0, COP1 and COP2 registers are written

1.3.3. COP1 Pipeline

The COP1 pipeline consists of eight stages. Each stage has two phases, as shown in Figure 1-3.
COP1 instructions execute simultaneously in the integer pipeline I0 and the COP1 pipeline. With regard to descriptions such as "A/T", "D/X" and so on, the first letter identifies the main integer pipeline stage and the second letter identifies the COP1 pipeline stage.

Symbol	Stage	Phase
1I	Instruction Fetch	Phase 1
2I	Instruction Fetch	Phase 2
1Q	Instruction Queue	Phase 1
2Q	Instruction Queue	Phase 2
1R	Register Fetch	Phase 1
2R	Register Fetch	Phase 2
1T	COP1 Register Fetch	Phase 1
2T	COP1 Register Fetch	Phase 2
X	Execution 1	
Y	Arithmetic/ALU 1	
Z	Arithmetic/ALU 2	
1S	Result writing	Phase 1
2S	Result writing	Phase 2

I	Q	R	A/T	D/X	W/Y	Z	S							
	I	Q	R	A/T	D/X	W/Y	Z	S						
		I	Q	R	A/T	D/X	W/Y	Z	S					
			I	Q	R	A/T	D/X	W/Y	Z	S				
				I	Q	R	A/T	D/X	W/Y	Z	S			
					I	Q	R	A/T	D/X	W/Y	Z	S		
						I	Q	R	A/T	D/X	W/Y	Z	S	
							I	Q	R	A/T	D/X	W/Y	Z	S

Figure 1-3 COP1 Pipeline

The operations of the I, Q and R stages are the same as those of the integer pipeline. The following describes operations in the stages specific to the COP1 pipeline.

1T: COP1 Register Fetch, Phase 1

- Operand register read

2T: COP1 Register Fetch, Phase 2

- Bypasses from the S Stage/W Stage for S/T overlap.

X: Execution 1

As the first phase for Multiply Operations, the following occurs:

- For sign, exclusive-OR is performed
- For exponent, biasing is performed
- For significand, the Booth function/Wallace multiplication is performed

ALU, Conversion and Min/Max instructions have no operations.

Y: Arithmetic / ALU Processing 1

As the second phase for Multiply Operations, the following occurs:

- Overflow/underflow on exponent is tested
- Normalization for multiplication is done

Also, as the first stage for ALU operations, the following occurs:

- Exponents are compared to determine the alignment
- For floating-point to integer conversion, the alignment step is performed
- For integer to floating-point conversion, the shift amount is determined

No operations are executed for Compare instructions.

Z: Arithmetic / ALU Processing 2

As the second phase of ALU operations, the following occurs:

- Overflow/underflow detection
- Exponent readjustment
- Significand addition
- Exponent renormalization
- For floating-point to integer conversion and 2's complement instructions, an overflow test is performed
- For integer to floating-point conversion and shift instructions, an exponent adjustment is performed
- For the Compare instruction, the comparison is performed

For Multiply Operations, this stage is no-op.

1S: Result Writing, Phase 1

During the 1S Stage, the results are available for computations.

2S: Result Writing, Phase 2

- FPR registers are written
- FCR31 is updated
- Bypass values are passed to the 2T stage

1.3.4. COP2 Pipeline

For COP2 pipeline operations, refer to the supplementary volume, "VU User's Manual". COP2 instructions execute simultaneously in the main integer pipeline I1 and the COP2 pipeline.

1.3.5. Classification and Routing of Instructions

Instruction routing, or what physical pipes are used for instructions of each category, is shown in Table 1-1. Instructions which require more than one execution pipeline are identified with *. For example, COP1 Move is executed in both the LS and the C1 pipelines. On the other hand, the ALU instructions are executed in either the I0 or the I1 pipeline. Figure 1-4 illustrates the contents of Table 1-1, adding the relation with logical pipes.

Instruction Categories	Instructions	Physical Pipeline					
		I0	I1	LS	BR	C1	C2
Load/Store	Load/Store, 128-bit Load/Store, Prefetch, CACHE			O			
SYNC	Synchronization		O				
LZC	Leading Zero Count		O				
ERET	Exception return		O				
SA Operate	Move to/from SA register	O					
COP0	COP0 move, COP0 operation			O			
COP1 Move	COP1 move, COP1 Load/Store			*		*	
COP2 Move	COP2 move, COP2 Load/Store			*			*
COP1 Operate	COP1 operation	*				*	
COP2 Operate	COP2 operation						*
ALU	Arithmetic, Shift, Logical, Trap, SYSCALL, BREAK	O	O				
MAC0	Multiply and Multiply-Accumulate for HI/LO register, Move to/from HI/LO	O					
MAC1	Multiply and Multiply-Accumulate for HI1/LO1 register, Move to/from HI1/LO1		O				
Branch	Branch and Jump				O		
Wide Operate	128-bit multimedia instructions	*	*				

Table 1-1 Categories of Instructions and Routing to Physical Pipes

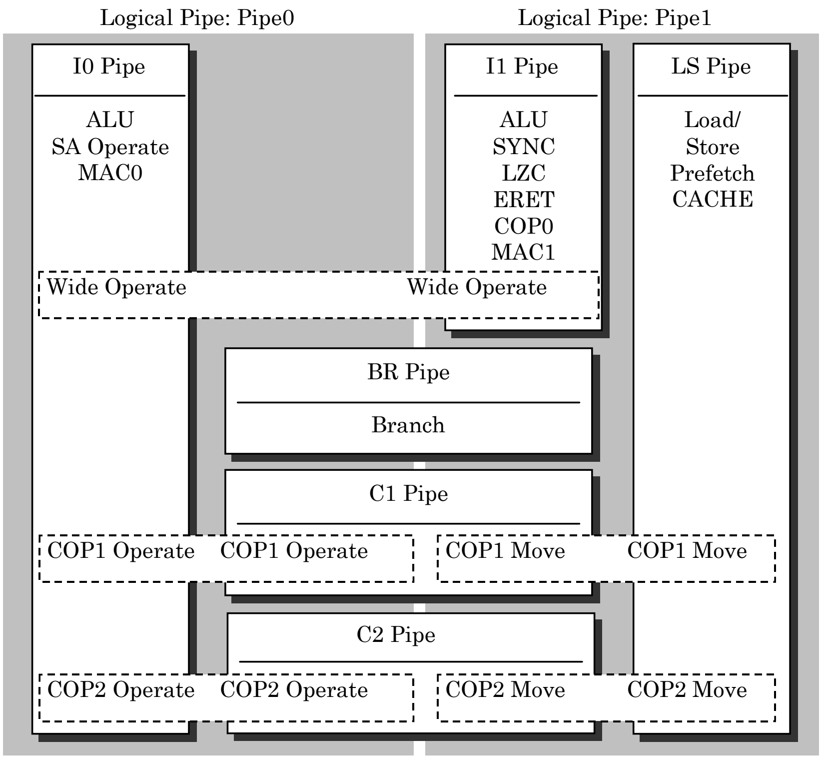


Figure 1-4 Instruction Routing in Logical and Physical Pipes

1.3.6. Instruction Issue Combinations

The EE Core always fetches two instructions and tries to issue two instructions in each cycle whenever possible. If an instruction can't be issued because of data dependency or other reasons, a pair of staging registers is used as a buffer to connect between the Q and the R stage. If two instructions can't be issued in a particular cycle, one instruction is saved in the staging registers. In the next cycle, the EE Core again fetches two instructions and tries to issue two instructions. The first one is what is left over in the staging register from the previous cycle; the other is what is newly fetched.

The instructions that get issued go to the R-stage of the pipeline and get associated with one of two logical pipes: Pipe 0 or Pipe 1. The instructions are then routed to an appropriate physical pipe for processing.

Instruction categories that can get issued to Pipe 0 and Pipe 1 are shown in Table 1-2.

Instruction	Logical Pipes	
	Pipe 0	Pipe 1
Load/Store		O
SYNC		O
LZC		O
ERET		O
SA Operate	O	
COP0		O
COP1 Move		O
COP2 Move		O
COP1 Operate	O	
COP2 Operate	O	
ALU	O	O
MAC0	O	
MAC1		O
Branch	O	O
Wide Operate	O	

Table 1-2 Instruction Categories and Routing to Logical Pipes

The ALU and Branch instruction categories can get issued to either Pipe 0 or Pipe 1. The binding of these 2 instructions is determined at instruction issue time.
In the MIPS ISA, placing an instruction from either the Branch or ERET category in the branch delay slot of the Branch category instruction is not allowed. That is, the following sequences are illegal and should not be issued.

- Branch – Branch
- Branch – ERET

The following sequences of instructions are also not allowed in the EE Core: (Though the Branch-Likely instruction category is a subset of the Branch instruction category, the following limitations are restricted to the Branch-Likely instruction category.)

- Branch – SYNC.P
- Branch – SYNC.L
- Branch – CACHE (CACHE instructions must be directly preceded and followed by a SYNC instruction.)
- Branch-Likely – MTSA
- Branch-Likely – MTSAB
- Branch-Likely – MTSAH
- Branch-Likely – TLBR
- Branch-Likely – TLBWI
- Branch-Likely – TLBWR

Table 1-3 shows the instruction categories that can be issued concurrently to the 2 logical pipes.
"X" indicates a combination in which an instruction can not be issued concurrently. "Y" indicates a combination in which an instruction can be issued concurrently (i.e. enter the R stage), but it stalls for a single cycle in the A stage because of a resource hazard in the previous instruction.

		Pipe0						
		SA Oper.	COP1 Oper.	COP2 Oper.	ALU	MAC0	Branch	Wide Oper.
Pipe1	Load / Store	O	O	O	O	O	O	O
	ERET	O	O	O	O	O	X	O
	SYNC	O	O	O	O	O	O	O
	LZC	O	O	O	O	O	O	Y
	COP1 Move	O	Y	O	O	O	O	O
	COP2 Move	O	O	Y	O	O	O	O
	ALU	O	O	O	O	O	O	Y
	MAC1	O	O	O	O	O	O	Y
	Branch	O	O	O	O	O	X	O
	COP0	O	O	O	O	O	O	O

Table 1-3 Concurrently Issued Instruction Categories

1.4. Registers

The EE Core contains a register set that extends the normal MIPS-compatible register set.
General purpose registers (GPRs), are extended from 64 bits to 128 bits, and a pair of HI/LO registers for the I1 pipe and the SA register for the funnel shift instructions are added.

1.4.1. CPU Registers

The EE Core has 128-bit wide general-purpose registers (GPRs). The upper 64 bits of the GPRs are only used by the EE Core-specific 128-bit Multimedia instructions (Parallel instructions).
HI1 and LO1 (which are the upper 64 bits of the 128-bit HI/LO registers respectively) are also used by the EE Core-specific multiply and divide instructions, such as MULT1, MULTU1, DIV1, DIVU1, MADD1, MADDU1, MFHI1, MFLO1, MTHI1 and MTLO1. These are not parallel instructions, but I1 pipe-specific instructions. The SA register holds the shift amount in funnel shift instructions that shift 128-bit registers.

1.4.2. FPU Registers

The floating-point unit (FPU=COP1) has 32-bit wide floating-point registers, 2 floating-point control registers and a single 32-bit accumulator.

1.4.3. COP0 Registers

Coprocessor 0 (COP0) registers have registers related to address translation, exception handling, debugging, and so forth.

1.5. Memory Management

The EE Core processor provides a memory management unit (MMU) that uses an on-chip translation look-aside buffer (TLB) to translate virtual addresses into physical addresses.
The EE Core supports the MIPS-compatible 32-bit address and 64-bit data mode. Only 32-bit virtual and physical addresses have been implemented. There is no requirement for address sign extension. Address error exception checking will not be done on the upper 32 bits (which are ignored). The only condition that will generate the address error exception will be address alignment errors and segment protection errors.
The address error exception will not occur even when a PC wraps around from kseg3 to kuseg in the kernel mode.
The reserved instruction exception will never occur to an instruction disabled by a processor mode, since there is only one addressing mode.
Features of the memory management of the EE Core are as follows:

- MIPS III-compatible 32-bit MMU (with special bit defined for scratchpad RAM)
- Operation Modes: User, Supervisor and Kernel
- TLB: 48 entries of even/odd page pairs (96 pages)
  Full set associative
- Page Size: 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, 16 MB
- ITLB: 2 entries
- DTLB: 4 entries
- Address Sizes: Virtual Address Size = 32 bits, 2 GB for each user process
  Physical Address Size = 32 bits, 4 GB

1.6. Cache Memory and Scratchpad RAM

The EE Core has an instruction cache and a separate data cache. It also has a scratchpad RAM for fast manipulation of large data structures.
The following are the main features of the caches:

- Separate Instruction Cache and Data Cache
- Virtually addressed index and physically addressed tag
- Write-back policy for the Data Cache
- Data Cache and Instruction Cache burst read sequential ordering
- Cache Size: Instruction Cache: 16 KB
  Data Cache: 8 KB
  Scratchpad RAM: 16 KB
- Line size: 64 bytes
- Refill size: 64 bytes
- The number of way: 2-way set-associative
- Write Policy: Write-back, write allocate
- Data order for block reads: Sequential ordering
- Data order for block writes: Sequential ordering
- Instruction cache miss restart: After all data received
- Data cache miss restart: Early restart on first quadword
- Cache parity: No
- Cache Locking: Data Cache Line Lock.
  Controlled by Cache instruction
- Cache Snooping: No
- Scratchpad RAM snooping: No
- Non-blocking load: Yes
- Hit Under Miss support: Yes
- Data Cache Prefetch: Yes

The following are the features of the Scratchpad RAM (SPRAM):

- 1024 x 128 bits (16 KB) static RAM
- External DMA read and write capability
- Accessible to software through load/store instructions

1.7. Bus Interface

The EE Core is connected to the rest of the system and external devices through the group of on-chip system bus signals called the CPU Bus. Features of the CPU Bus include the following:

- Separate data and address buses (Demultiplexed operation)
- 128-bit data bus
- Clocked synchronous operations
- 8/16/32/64/128-bit burst access
- Multimaster capability
- Pipelined operations
- No turn-around or dead cycles between transfers

Note that cache coherency support and split transactions are not provided.

1.8. Floating-Point Unit

The floating-point unit (FPU = COP1) is a unit implementing a single-precision floating-point operation. This unit is not IEEE 754 compatible. The features are as follows:

- High-performance single-precision floating-point unit tightly coupled to the EE Core
- Supports single-precision format as defined in the IEEE 754 specification
- Plus/Minus "0" in line with the IEEE 754 specification are supported
- NaNs and plus/minus infinities are not supported
- No hardware exception mechanism to affect instruction execution
- Supports ADD, SUB, MUL, DIV, ABS, NEG, SQRT, RSQRT, MAX, MIN, Compare, and Conversion instructions

For details of the floating-point unit, refer to "8. Floating-Point Unit (FPU)".

1.9. Debug Support Functions

The EE Core has the following debug support features:

- Instruction Address Breakpoint
- Data Address Breakpoint
- Data Value Breakpoint
- Each breakpoint is individually available and can set Mask
- Breakpoints can be available in different processor modes (User, Supervisor, Kernel and Exception modes)

2. Instruction Set Overview

This chapter provides an overview of the EE Core instruction set. Refer to the supplementary volume "EE Core Instruction Set Manual" for detailed descriptions of individual instructions.
The EE Core supports all MIPS III instructions with the exception of 64-bit multiply, 64-bit divide, Load-Linked and Store Conditional instructions. It also implements additional EE Core-specific instructions, such as selected MIPS IV, Multiply/Add, and multimedia instructions.
The instruction set of the EE Core can be divided into the following groups:

- Load and Store instructions
- Computational instructions
- Branch and Jump instructions
- Exception instructions
- Serialization instructions
- MIPS IV instructions
- System Control Coprocessor (COP0) instructions
- Coprocessor (COP1 / COP2) instructions
- EE Core-specific instructions

2.1. Binary Formats

There are three instruction formats in the EE Core: I-type (immediate), J-type (Jump), R-type (register), as shown below.

I-type (immediate)

31 26	25 21	20 16	15 0
op	rs	rt	immediate

J-type (jump)

31 26	25 0
op	target

R-type (register)

31 26	25 21	20 16	15 11	10 6	5 0
op	rs	rt	rd	sa	function

The contents of each field are as follows.

Field Name	Width	Contents
op	6 bits	Opcode
rs	5 bits	Source register specifier
rt	5 bits	Specifies target (source/destination) register or branch condition
immediate	16 bits	Immediate value, branch instruction offset or offset address
target	26 bits	Jump target address
rd	5 bits	Destination register specifier
sa	5 bits	Shift amount
function	6 bits	Function field

Refer to the supplementary volume "EE Core Instruction Set Manual" for the actual values.

2.2. Instruction Set Summary

The EE Core supports most of the MIPS III instructions and some of the MIPS IV instructions. In addition, EE Core-specific instructions (e.g. Multiply-Add) are implemented.

2.2.1. Instruction Set List

The EE Core instruction set is listed below.

Load/Store Instructions

Mnemonic	Description	MIPS ISA
LB	Load Byte	MIPS I
LBU	Load Byte Unsigned	MIPS I
LD	Load Doubleword	MIPS III
LDL	Load Doubleword Left	MIPS III
LDR	Load Doubleword Right	MIPS III
LH	Load Halfword	MIPS I
LHU	Load Halfword Unsigned	MIPS I
LW	Load Word	MIPS I
LWL	Load Word Left	MIPS I
LWR	Load Word Right	MIPS I
LWU	Load Word Unsigned	MIPS III
SB	Store Byte	MIPS I
SD	Store Doubleword	MIPS III
SDL	Store Doubleword Left	MIPS III
SDR	Store Doubleword Right	MIPS III
SH	Store Halfword	MIPS I
SW	Store Word	MIPS I
SWL	Store Word Left	MIPS I
SWR	Store Word Right	MIPS I

Computational Instructions: ALU Immediate Operations

Mnemonic	Description	MIPS ISA
ADDI	Add Immediate	MIPS I
ADDIU	Add Immediate Unsigned	MIPS I
ANDI	AND Immediate	MIPS I
DADDI	Doubleword Add Immediate	MIPS III
DADDIU	Doubleword Add Immediate (Unsigned)	MIPS III
LUI	Load Upper Immediate	MIPS I
ORI	OR Immediate	MIPS I
SLTI	Set on Less Than Immediate	MIPS I
SLTIU	Set on Less Than Immediate Unsigned	MIPS I
XORI	Exclusive OR Immediate	MIPS I

Computational Instructions: Three-Operand Register-Type Operations

Mnemonic	Description	MIPS ISA
ADD	Add	MIPS I
ADDU	Add Unsigned	MIPS I
AND	AND	MIPS I
DADD	Doubleword Add	MIPS III
DADDU	Doubleword Add Unsigned	MIPS III
DSUB	Doubleword Subtract	MIPS III
DSUBU	Doubleword Subtract Unsigned	MIPS III
NOR	NOR	MIPS I
OR	OR	MIPS I
SLT	Set on Less Than	MIPS I
SLTU	Set on Less Than Unsigned	MIPS I
SUB	Subtract	MIPS I
SUBU	Subtract Unsigned	MIPS I
XOR	Exclusive OR	MIPS I

Computational Instructions: Shift Operations

Mnemonic	Description	MIPS ISA
DSLL	Doubleword Shift Left Logical	MIPS III
DSLL32	Doubleword Shift Left Logical +32	MIPS III
DSLLV	Doubleword Shift Left Logical Variable	MIPS III
DSRA	Doubleword Shift Right Arithmetic	MIPS III
DSRA32	Doubleword Shift Right Arithmetic +32	MIPS III
DSRAV	Doubleword Shift Right Arithmetic Variable	MIPS III
DSRL	Doubleword Shift Right Logical	MIPS III
DSRL32	Doubleword Shift Right Logical +32	MIPS III
DSRLV	Doubleword Shift Right Logical Variable	MIPS III
SLL	Shift Left Logical	MIPS I
SLLV	Shift Left Logical Variable	MIPS I
SRA	Shift Right Arithmetic	MIPS I
SRAV	Shift Right Arithmetic Variable	MIPS I
SRL	Shift Right Logical	MIPS I
SRLV	Shift Right Logical Variable	MIPS I

Computational Instructions: Multiply and Divide

Mnemonic	Description	MIPS ISA
DIV	Divide	MIPS I
DIVU	Divide Unsigned	MIPS I
MFHI	Move From HI	MIPS I
MFLO	Move From LO	MIPS I
MTHI	Move To HI	MIPS I
MTLO	Move To LO	MIPS I
MULT	Multiply	MIPS I
MULTU	Multiply Unsigned	MIPS I

In addition, there are EE Core-specific Multiply and Divide instructions (which use the I1 pipe explicitly.)

Branch/Jump Instructions

Mnemonic	Description	MIPS ISA
J	Jump	MIPS I
JAL	Jump And Link	MIPS I
JALR	Jump And Link	MIPS I
JR	Jump Register	MIPS I
BEQ	Branch on Equal	MIPS I
BEQL	Branch on Equal Likely	MIPS II
BGEZ	Branch on Greater Than or Equal to Zero	MIPS I
BGEZAL	Branch on Greater Than or Equal to Zero And Link	MIPS I
BGEZALL	Branch on Greater Than or Equal to Zero And Link Likely	MIPS II
BGEZL	Branch on Greater Than or Equal to Zero Likely	MIPS II
BGTZ	Branch on Greater Than Zero	MIPS I
BGTZL	Branch on Greater Than Zero Likely	MIPS II
BLEZ	Branch on Less Than or Equal to Zero	MIPS I
BLEZL	Branch on Less Than or Equal to Zero Likely	MIPS II
BLTZ	Branch on Less Than Zero	MIPS I
BLTZAL	Branch on Less Than Zero and Link	MIPS I
BLTZALL	Branch on Less Than Zero And Link Likely	MIPS II
BLTZL	Branch on Less Than Zero Likely	MIPS II
BNE	Branch on Not Equal	MIPS I
BNEL	Branch on Not Equal Likely	MIPS II

Exception Instructions

Mnemonic	Description	MIPS ISA
SYSCALL	System Call	MIPS I
BREAK	Break	MIPS I
TGE	Trap if Greater Than or Equal	MIPS II
TGEU	Trap if Greater Than or Equal Unsigned	MIPS II
TLT	Trap if Less Than	MIPS II
TLTU	Trap if Less Than Unsigned	MIPS II
TEQ	Trap if Equal	MIPS II
TNE	Trap if Not Equal	MIPS II
TGEI	Trap if Greater Than or Equal Immediate	MIPS II
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	MIPS II
TLTI	Trap if Less Than Immediate	MIPS II
TLTIU	Trap if Less Than Immediate Unsigned	MIPS II
TEQI	Trap if Equal Immediate	MIPS II
TNEI	Trap if Not Equal Immediate	MIPS II

Serialization Instruction

Mnemonic	Description	MIPS ISA
SYNC	Synchronization	MIPS II

MIPS IV Instructions

Mnemonic	Description	MIPS ISA
MOVN	Move on Register Not Equal to Zero	MIPS IV
MOVZ	Move on Register Equal to Zero	MIPS IV
PREF	Prefetch	MIPS IV

2.2.2. MIPS III Instructions not Supported by EE Core

The EE Core does not support the following MIPS III instructions:

Mnemonic	Description	MIPS ISA
DDIV	64-bit Divide	MIPS III
DDIVU	64-bit Divide (unsigned)	MIPS III
DMULT	64-bit Multiply	MIPS III
DMULTU	64-bit Multiply (unsigned)	MIPS III
LL	Semaphore	MIPS III
LLD	Semaphore	MIPS III
SC	Semaphore	MIPS III
SCD	Semaphore	MIPS III

2.3. Load/Store Instructions

Load/Store instructions transfer different sizes of data—bytes, halfwords, words and doublewords—between memory and registers. Load instructions include sign-extended and zero-extended instructions. They support signed and unsigned integers.

The binary formats of load and store instructions are I-type (Immediate) and the only addressing mode is "base register plus 16-bit signed immediate offset" mode.

Note that the EE Core does not support Load-Linked and Store Conditional instructions, LL, LLD, SC and SCD in MIPS III instructions.

The list of load/store instructions is shown below. In addition, refer to "2.10. Coprocessor Instructions (COP1/COP2) " about load/store instructions related to the coprocessor.

Mnemonic	Description	MIPS ISA
LB	Load Byte	MIPS I
LBU	Load Byte Unsigned	MIPS I
LD	Load Doubleword	MIPS III
LDL	Load Doubleword Left	MIPS III
LDR	Load Doubleword Right	MIPS III
LH	Load Halfword	MIPS I
LHU	Load Halfword Unsigned	MIPS I
LW	Load Word	MIPS I
LWL	Load Word Left	MIPS I
LWR	Load Word Right	MIPS I
LWU	Load Word Unsigned	MIPS III
SB	Store Byte	MIPS I
SD	Store Doubleword	MIPS III
SDL	Store Doubleword Left	MIPS III
SDR	Store Doubleword Right	MIPS III
SH	Store Halfword	MIPS I
SW	Store Word	MIPS I
SWL	Store Word Left	MIPS I
SWR	Store Word Right	MIPS I

Table 2-1 Load/Store Instructions

2.3.1. Data Formats and Alignment

The EE Core uses the following five data formats:

- 128-bit quadword
- 64-bit doubleword
- 32-bit word
- 16-bit halfword
- 8-bit byte

Byte ordering in data formats of halfword or greater is configured in little-endian order . Byte 0 is always the least significant (rightmost) byte, which is compatible with iAPX® x86 and DEC VAX® conventions.

Bit ordering is configured in little endian and bit 0 always indicates the least significant (rightmost) bit (although no instructions explicitly designate bit positions within words).

Refer to Figure 2-1and Figure 2-2 about the word/byte ordering and bit ordering respectively.

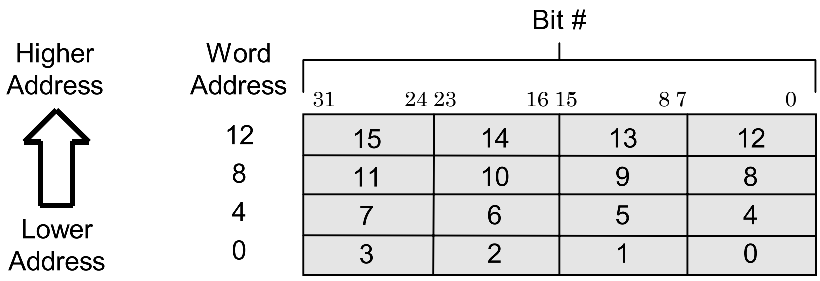


Figure 2-1 Little-Endian Byte Ordering

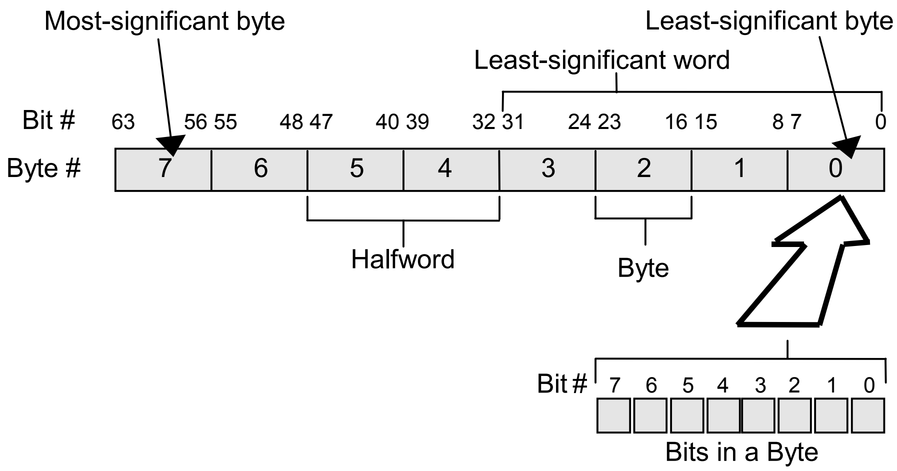


Figure 2-2 Little-Endian Data in a Doubleword

The EE Core uses byte addressing for all data access. Data of a halfword or greater has the following alignment constraints. Any access that does not satisfy these constraints will generate address error exceptions.

Data Formats	Condition
Halfword	Even address (0, 2, 4...)
Word	Address divisible by four (0, 4, 8...)
Doubleword	Address divisible by eight (0, 8, 16...)
Quadword	Address divisible by sixteen (0, 16, 32...)

The following special instructions load and store words, doublewords or quadwords that are not aligned according to the above constraints.

Mnemonic	Description
LWL	Load Word Left
LWR	Load Word Right
SWL	Store Word Left
SWR	Store Word Right
LDL	Load Doubleword Left
LDR	Load Doubleword Right
SDL	Store Doubleword Left
SDR	Store Dobuleword Right

In order to load/store misaligned data, these instructions have to be used in pairs. Therefore, misaligned data requires one more instruction cycle than aligned data. Refer to Figure 2-3.

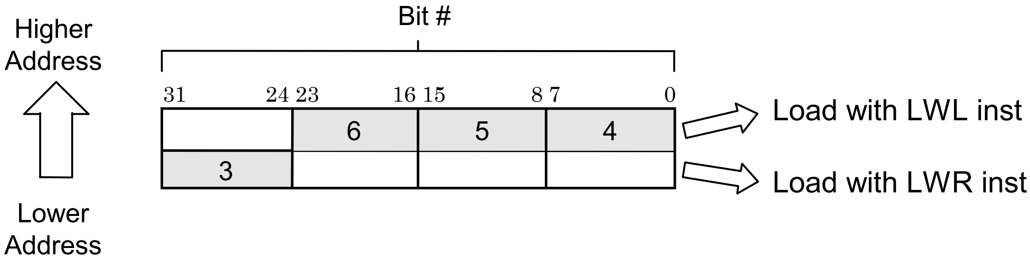


Figure 2-3 Little-Endian Misaligned Word Addressing

If LQ and SQ instructions are combined with a QFSRV instruction (funnel shift), extracting aligned data from a misaligned quadword is possible.

2.3.2. Load Delay

In general, when the data loaded by an instruction is not allowed to be used by the immediately following instruction, the load instruction is called a delayed load instruction. The delay until the data can be used is called the load delay slot.

In the EE Core, there is no absolute delayed load instruction. Loaded data is allowed to be used in the instruction following a load instruction. In such cases, however, hardware interlocks insert additional clock cycles. Consequently, taking a load delay into account in programming is not required. However, for software performance, it is desirable to place an instruction that does not use the data immediately following a load instruction.

2.4. Computational Instructions

EE Core computational instructions can be of either the R-type or I-type format. Both operands are registers in the R-type format. One operand is a 16-bit immediate in the I-type format.
Computational instructions perform the following operations on register values:

- Arithmetic
- Logical
- Shift
- Multiply
- Divide

Computational instructions are divided into the following four categories:

- ALU immediate instructions
- Three-Operand Register-Type instructions
- Shift instructions
- Multiply and Divide instructions

The EE Core does not support the 64-bit Multiply and Divide instructions (DMULT, DMULTU, DDIV and DDIVU) in the MIPS III instruction set. In 64-bit operations, sign-extended or zero-extended 32-bit values are required. The result is unpredictable if an incorrect 64-bit value is used.

Computational Instructions: ALU Immediate Operations

Mnemonic	Description	MIPS ISA
ADDI	Add Immediate	MIPS I
ADDIU	Add Immediate Unsigned	MIPS I
ANDI	AND Immediate	MIPS I
DADDI	Doubleword Add Immediate	MIPS III
DADDIU	Doubleword Add Immediate (Unsigned)	MIPS III
LUI	Load Upper Immediate	MIPS I
ORI	OR Immediate	MIPS I
SLTI	Set on Less Than Immediate	MIPS I
SLTIU	Set on Less Than Immediate Unsigned	MIPS I
XORI	Exclusive OR Immediate	MIPS I

Computational Instructions: Three-Operand Register-Type Operations

Mnemonic	Description	MIPS ISA
ADD	Add	MIPS I
ADDU	Add Unsigned	MIPS I
AND	AND	MIPS I
DADD	Doubleword Add	MIPS III
DADDU	Doubleword Add Unsigned	MIPS III
DSUB	Doubleword Subtract	MIPS III
DSUBU	Doubleword Subtract Unsigned	MIPS III
NOR	NOR	MIPS I
OR	OR	MIPS I
SLT	Set on Less Than	MIPS I
SLTU	Set on Less Than Unsigned	MIPS I
SUB	Subtract	MIPS I
SUBU	Subtract Unsigned	MIPS I
XOR	Exclusive OR	MIPS I

Computational Instructions: Shift Operations

Mnemonic	Description	MIPS ISA
DSLL	Doubleword Shift Left Logical	MIPS III
DSLL32	Doubleword Shift Left Logical +32	MIPS III
DSLLV	Doubleword Shift Left Logical Variable	MIPS III
DSRA	Doubleword Shift Right Arithmetic	MIPS III
DSRA32	Doubleword Shift Right Arithmetic +32	MIPS III
DSRAV	Doubleword Shift Right Arithmetic Variable	MIPS III
DSRL	Doubleword Shift Right Logical	MIPS III
DSRL32	Doubleword Shift Right Logical +32	MIPS III
DSRLV	Doubleword Shift Right Logical Variable	MIPS III
SLL	Shift Left Logical	MIPS I
SLLV	Shift Left Logical Variable	MIPS I
SRA	Shift Right Arithmetic	MIPS I
SRAV	Shift Right Arithmetic Variable	MIPS I
SRL	Shift Right Logical	MIPS I
SRLV	Shift Right Logical Variable	MIPS I

Computational Instructions: Multiply and Divide

Mnemonic	Description	MIPS ISA
DIV	Divide	MIPS I
DIVU	Divide Unsigned	MIPS I
MFHI	Move From HI	MIPS I
MFLO	Move From LO	MIPS I
MTHI	Move To HI	MIPS I
MTLO	Move To LO	MIPS I
MULT	Multiply	MIPS I
MULTU	Multiply Unsigned	MIPS I

In addition, there are EE Core-specific Multiply and Divide instructions (which use the I1 pipe explicitly.)

Table 2-2 Computational Instructions

2.5. Branch/Jump Instructions

The EE Core instruction set defines the following instructions to change the flow of control in programming: PC-relative conditional branches, a PC-region unconditional jump, an absolute register unconditional jump, and a corresponding subroutine call (which records the return link address in a general-purpose register). There are two different conditional branch instructions: Branch and Branch-Likely, as described below.

Mnemonic	Description	MIPS ISA
J	Jump	MIPS I
JAL	Jump And Link	MIPS I
JALR	Jump And Link Register	MIPS I
JR	Jump Register	MIPS I
BEQ	Branch on Equal	MIPS I
BEQL	Branch on Equal Likely	MIPS II
BGEZ	Branch on Greater Than or Equal to Zero	MIPS I
BGEZAL	Branch on Greater Than or Equal to Zero And Link	MIPS I
BGEZALL	Branch on Greater Than or Equal to Zero And Link Likely	MIPS II
BGEZL	Branch on Greater Than or Equal to Zero Likely	MIPS II
BGTZ	Branch on Greater Than Zero	MIPS I
BGTZL	Branch on Greater Than Zero Likely	MIPS II
BLEZ	Branch on Less Than or Equal to Zero	MIPS I
BLEZL	Branch on Less Than or Equal to Zero Likely	MIPS II
BLTZ	Branch on Less Than Zero	MIPS I
BLTZAL	Branch on Less Than Zero and Link	MIPS I
BLTZALL	Branch on Less Than Zero and Link Likely	MIPS II
BLTZL	Branch on Less Than Zero Likely	MIPS II
BNE	Branch on Not Equal	MIPS I
BNEL	Branch on Not Equal Likely	MIPS II

Table 2-3 Branch and Jump Instructions

2.5.1. Branch Delay Slot

All branch instructions have a delay of one instruction (the branch delay slot). That is, the instruction immediately following the branch instruction is executed while the target instruction is fetched.

If a branch is taken when an unconditional or conditional branch is established, the instruction in the branch delay slot (immediately following the branch instruction) is executed before the branch operation.

If the branch is not taken and execution falls through, branch instructions execute the instruction in the delay slot, but the branch-likely instructions do not. (They are said to nullify it.)

By convention, if an instruction in the branch delay slot is suspended by an exception or interrupt, the program can be continued by re-executing the immediately preceding branch instruction. To permit this, branches must be restartable. That is, in procedure calls, the register that stores the return link (usually register 31) must not determine the branch target.

2.5.2. Overview of Jump Instructions

Subroutine calls in high-level languages are usually implemented with Jump or Jump-Link instructions, both of which are J-type instructions. The 26-bit target address shifts left 2 bits and combines with the high-order 4 bits of the program counter to form the branch target address.
Returns, dispatches and cross-page jumps are usually implemented with the Jump-Register or Jump-Link-Register instructions. Both are R-type instructions, in which the value of one of the general-purpose registers is the branch target address.

2.5.3. Overview of Branch Instructions

Conditional statements in high-level languages are usually implemented with branch or branch-likely instructions, both of which are I-type instructions. The 16-bit offset shifts left 2 bits and is sign-extended to 32 bits, which is added to the instruction address of the branch delay slot, to form the branch target address.
Branch instructions are different from branch-likely instructions, as they execute the instruction in the branch delay slot when a condition is not taken. Branch-likely instructions nullify the instruction in the branch delay slot when a condition is not taken.

2.6. Exception Instructions

Exception instructions allow the software to cause traps. The list is shown below. Refer to the "EE Core Instruction Set Manual" for further information.

Mnemonic	Description	MIPS ISA
SYSCALL	System Call	MIPS I
BREAK	Break	MIPS I
TGE	Trap if Greater Than or Equal	MIPS II
TGEU	Trap if Greater Than or Equal Unsigned	MIPS II
TLT	Trap if Less Than	MIPS II
TLTU	Trap if Less Than Unsigned	MIPS II
TEQ	Trap if Equal	MIPS II
TNE	Trap if Not Equal	MIPS II
TGEI	Trap if Greater Than or Equal Immediate	MIPS II
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	MIPS II
TLTI	Trap if Less Than Immediate	MIPS II
TLTIU	Trap if Less Than Immediate Unsigned	MIPS II
TEQI	Trap if Equal Immediate	MIPS II
TNEI	Trap if Not Equal Immediate	MIPS II

Table 2-4 Exception Instructions

2.7. Serialization Instruction

The order in which memory accesses from load and store instructions appear outside the EE Core is not specified by the EE Core architecture.

The order of loads and stores can be guaranteed at a point in a program by using a SYNC or SYNC.L instruction. That is, load and store instructions executed before the SYNC or SYNC.L instruction are guaranteed to retire before load and store instructions following the SYNC or SYNC.L instruction are executed. In addition to load and store instructions, a SYNC.P instruction can be used to guarantee the completion of an instruction. Instructions executed before a SYNC.P instruction are completed before an instruction following SYNC.P is executed.

Mnemonic	Description	MIPS ISA
SYNC	Synchronization with load/store operation	MIPS II
SYNC.L	Synchronization with load/store operation	MIPS II
SYNC.P	Synchronization	MIPS II

2.8. MIPS IV Instructions

The EE Core supports the following parts of the MIPS IV instruction set:

- Conditional move instructions
- The Prefetch instruction

Mnemonic	Description	MIPS ISA
MOVN	Move on Register Not Equal to Zero	MIPS IV
MOVZ	Move on Register Equal to Zero	MIPS IV
PREF	Prefetch	MIPS IV

Table 2-5 MIPS IV Instruction Supported in EE Core

Conditional Move operations allow "IF" statements to be described without branches. "THEN" and "ELSE" clauses are always computed and the results are placed in a temporary register. Then, appropriate results are transferred to the target register depending on the conditions.
The Prefetch instruction fetches data expected to be used in the near future and places it in the Data Cache.

2.9. System Control Coprocessor (COP0) Instructions

COP0 instructions perform operations specifically on the System Control Coprocessor to manipulate the memory management and exception handling facilities of the processor.
COP0 instructions are enabled when the processor is in Kernel mode or when bit 28 (CU [0]) is set in the STATUS register. Otherwise, executing the following instructions generates a "Coprocessor Unusable" exception.
Refer to the "EE Core Instruction Set Manual" for details of each COP0 instruction.

Mnemonic	Description
BC0F	Branch on COP0 False
BC0FL	Branch on COP0 False Likely
BC0T	Branch on COP0 True
BC0TL	Branch on COP0 True Likely
CACHE.op	Cache
DI	Disable Interrupt
EI	Enable Interrupt
ERET	Exception Return
MFBPC	Move From Breakpoint Control
MFC0	Move From COP0
MFDAB	Move From Data Address Breakpoint
MFDABM	Move From Data Address Breakpoint Mask
MFDVB	Move From Data Value Breakpoint
MFDVBM	Move From Data Value Breakpoint Mask
MFIAB	Move From Instruction Address Breakpoint
MFIABM	Move From Instruction Address Breakpoint Mask
MFPC	Move From Performance Counter
MFPS	Move From Performance Event Specifier
MTBPC	Move To Breakpoint Control
MTC0	Move To COP0
MTDAB	Move To Data Address Breakpoint
MTDABM	Move To Data Address Breakpoint Mask
MTDVB	Move To Data Value Breakpoint
MTDVBM	Move To Data Value Breakpoint Mask
MTIAB	Move To Instruction Address Breakpoint
MTIABM	Move To Instruction Address Breakpoint Mask
MTPC	Move To Performance Counter
MTPS	Move To Performance Event Specifier
TLBR	Read Indexed TLB Entry
TLBWI	Write Index TLB Entry
TLBWR	Write Random TLB Entry

Table 2-6 Coprocessor 0 Instructions

2.10. Coprocessor Instructions (COP1/COP2)

COP1 and COP2 instructions perform operations in their respective coprocessors. Loads and stores I-type. Other instructions have coprocessor-dependent formats.

2.10.1. COP1(FPU) Instructions

The following are COP1 instructions.

Mnemonic	Description
ABS.S	Single Floating-Point Absolute
ADD.S	Single Floating-Point Add
ADDA.S	Single Floating-Point Add to Accumulator
BC1F	Branch on FPU False
BC1FL	Branch on FPU False Likely
BC1T	Branch on FPU True
BC1TL	Branch on FPU True Likely
C.cond.S	Single Floating-Point Compare
CFC1	Move Control Word from FCR
CTC1	Move Control Word to FCR
CVT.S.W	32-bit Fixed Point Floating-Point Convert to Single Floating-Point
CVT.W.S	Single Floating-Point Convert to 32-bit Fixed Point
DIV.S	Single Floating-Point Divide
LWC1	Load Word to FPR
MADD.S	Single Floating-Point Multiply and Add
MADDA.S	Single Floating-Point Multiply and Add to Accumulator
MAX.S	Single Floating-Point Maximum
MFC1	Move Word from FPR
MIN.S	Single Floating-Point Minimum
MOV.S	Single Floating-Point Move
MSUB.S	Single Floating-Point Multiply and Subtract
MSUBA.S	Single Floating-Point Multiply / Subtract from Accumulator
MTC1	Move Word to FPR
MUL.S	Single Floating-Point Multiply
MULA.S	Single Floating-Point Multiply to Accumulator
NEG.S	Single Floating-Point Negate
RSQRT.S	Single Floating-Point Reciprocal Square Root
SQRT.S	Single Floating-Point Square Root
SUB.S	Single Floating-Point Subtract
SUBA.S	Single Floating-Point Subtract to Accumulator
SWC1	Store Word from FPR

Table 2-7 Coprocessor 1 Instructions

2.10.2. COP2 Instructions

The following are COP2 instructions. Refer to the "VU User's Manual" for the details of COP2 instructions.

Mnemonic	Description
BC2F	Branch on COP2 False
BC2FL	Branch on COP2 False Likely
BC2T	Branch on COP2 True
BC2TL	Branch on COP2 True Likely
CALLMS	Call Micro Subroutine
CALLMSR	Call Micro Subroutine Register
CFC2	Move Control From COP2
CTC2	Move Control To COP2
LQC2	Load Quadword to COP2
SQC2	Store Quadword from COP2
QMFC2	Quadword Move From COP2
QMTC2	Quadword Move To COP2
WAITQ	Wait Q Register

Table 2-8 COP2 Instructions

2.10.3. VU Macro instructions

COP2 (VPU0) provides EE Core programs with a macro instruction set, with almost the same functionality as the VU-specific instruction set (micro instructions). The list of macro instructions is shown below. Refer to the "VU User's Manual" for the details of each instruction.

VU Macro Instructions: Floating-Point Operations

Mnemonic	Description
VABS	Absolute
VADD	Addition
VADDi	ADD broadcast I register
VADDq	ADD broadcast Q register
VADDbc	ADD broadcast bc field
VADDA	ADD output to ACC
VADDAi	ADD output to ACC broadcast I register
VADDAq	ADD output to ACC broadcast Q register
VADDAbc	ADD output to ACC broadcast bc field
VSUB	Subtraction
VSUBi	SUB broadcast I register
VSUBq	SUB broadcast Q register
VSUBbc	SUB broadcast bc field
VSUBA	SUB output to ACC
VSUBAi	SUB output to ACC broadcast I register
VSUBAq	SUB output to ACC broadcast Q register
VSUBAbc	SUB output to ACC broadcast bc field
VMU	Multiply
VMULi	MUL broadcast I register
VMULq	MUL broadcast Q register
VMULbc	MUL broadcast bc field
VMULA	MUL output to ACC
VMULAi	MUL output to ACC broadcast I register

Mnemonic	Description
VMULAq	MUL output to ACC broadcast Q register
VMULAbc	MUL output to ACC broadcast bc field
VMADD	MUL and ADD (SUB)
VMADDi	MUL and ADD (SUB) broadcast I register
VMADDq	MUL and ADD (SUB) broadcast Q register
VMADDbc	MUL and ADD (SUB) broadcast bc field
VMADDA	MUL and ADD (SUB) output to ACC
VMADDAi	MUL and ADD (SUB) output to ACC broadcast I register
VMADDAq	MUL and ADD (SUB) output to ACC broadcast Q register
VMADDAbc	MUL and ADD (SUB) output to ACC broadcast bc field
VMSUB	Multiply and SUB
VMSUBi	Multiply and SUB broadcast I register
VMSUBq	Multiply and SUB broadcast Q register
VMSUBbc	Multiply and SUB broadcast bc field
VMSUBA	Multiply and SUB output to ACC
VMSUBAi	Multiply and SUB output to ACC broadcast I register
VMSUBAq	Multiply and SUB output to ACC broadcast Q register
VMSUBAbc	Multiply and SUB output to ACC broadcast bc field
VMAX	Maximum
VMAXi	Maximum broadcast I register
VMAXbc	Maximum broadcast bc field
VMINI	Minimum
VMINIi	Minimum broadcast I register
VMINIbc	Minimum broadcast bc field
VOPMULA	Outer product MULA
VOPMSUB	Outer product MSUB
VDIV	Floating Divide
VSQRT	Floating Square-root
VRSQRT	Floating reciprocal Square-root

VU Macro Instructions: Integer Operations

Mnemonic	Description
VIADD	Integer ADD
VIADDI	Integer ADD immediate
VIAND	Integer AND
VIOR	Integer OR
VISUB	Integer SUB

VU Macro Instructions: Convert/Move

Mnemonic	Description
VFTOI0	Float to Integer, fixed point 0-bit
VFTOI4	Float to Integer, fixed point 4-bit
VFTOI12	Float to Integer, fixed point 12-bit
VFTOI15	Float to Integer, fixed point 15-bit
VITOF0	Integer to Float, fixed point 0-bit
VITOF4	Integer to Float, fixed point 4-bit
VITOF12	Integer to Float, fixed point 12-bit
VITOF15	Integer to Float, fixed point 15-bit
VMOVE	Move Floating register
VMFIR	Move From integer register
VMTIR	Move To integer register
VMR32	Rotate right 32 bits

VU Macro Instructions: Random Numbers

Mnemonic	Description
VRINIT	Random-unit init R register
VRGET	Random-unit get R register
VRNEXT	Random-unit next M sequence
VRXOR	Random-unit XOR R register

VU Macro Instructions: Load/Store

Mnemonic	Description
VLQD	Load Quadword with pre-decrement
VLQI	Load Quadword with post-increment
VSQD	Store Quadword with pre-decrement
VSQI	Store Quadword with post-increment
VILWR	Integer load word register
VISWR	Integer store word register

VU Macro Instructions: Others

Mnemonic	Description
VNOP	No Operation
VCLIP	Clipping
VWAITQ	Wait Q Register

Table 2-9 VU Macro Instructions

2.11. EE Core-Specific Instructions

The EE Core extends its instruction set from the original MIPS architecture. The following instructions are supported in particular:

- Three-operand Multiply and Multiply-Add instructions
- Multiply and divide instruction for Pipeline I1
- Multimedia instructions
- Enable interrupt and Disable interrupt instructions

Refer to the "EE Core Instruction Set Manual" for more information about each instruction.

2.11.1. EE Core-Specific Multiply / Divide Instructions

The standard MIPS instructions for multiply, divide and move to / from HI / LO registers execute on the I0 pipeline's MAC unit. A set of new instructions has also been defined to execute on the I1 pipeline's MAC unit:

Mnemonic	Description
MADD	Multiply/Add
MADDU	Multiply/Add Unsigned
MULT	Multiply (3-operand)
MULTU	Multiply Unsigned (3-operand)
MULT1	Multiply 1
MULTU1	Multiply Unsigned 1
DIV1	Divide 1
DIVU1	Divide Unsigned 1
MADD1	Multiply/Add 1
MADDU1	Multiply/Add Unsigned 1
MFHI1	Move From HI1
MFLO1	Move From LO1
MTHI1	Move To HI1
MTLO1	Move To LO1

Table 2-10 EE Core-Specific Multiply / Divide Instructions

The EE Core supports three-operand multiply instructions that store the multiply result to a general-purpose register in addition to the LO register. These instructions don't have to use the MFLO instruction to move data from the LO register to a general-purpose register.

- MULT rd, rs, rt HI || LO = rs x rt (signed)
  rd = new LO contents
- MADDU rd, rs, rt HI || LO += rs x rt (unsigned)
  rd = new LO contents

The EE Core also supports new multiply-add instructions, MADD and MADDU. These instructions execute multiply-accumulate operations using the HI and LO registers as accumulators.

- MADD rd, rs, rt HI || LO += rs x rt (signed)
  rd = new LO contents
- MADDU rd, rs, rt HI || LO += rs x rt (unsigned)
  rd = new LO contents

2.11.2. Multimedia Instructions

The EE Core implements a new set of instructions to support multimedia applications. (See Table 2-11) Most of these instructions perform parallel operations by combining the I0 and I1 pipelines. Instructions perform parallel operations on either two 64-bit data items, four 32-bit data items, eight 16-bit data items or sixteen 8-bit data items to form a 128-bit path.

In order to support the 128-bit datapath, 128-bit load/store instructions are also implemented.

Multimedia Instructions: Arithmetic

Mnemonic	Description
PADDB	Parallel Add Byte
PSUBB	Parallel Subtract Byte
PADDH	Parallel Add Halfword
PSUBH	Parallel Subtract Halfword
PADDW	Parallel Add Word
PSUBW	Parallel Subtract Word
PADSBH	Parallel Add/Subtract Halfword
PADDSB	Parallel Add with Signed Saturation Byte
PSUBSB	Parallel Subtract with Signed Saturation Byte
PADDSH	Parallel Add with Signed Saturation Halfword
PSUBSH	Parallel Subtract with Signed Saturation Halfword
PADDSW	Parallel Add with Signed Saturation Word
PSUBSW	Parallel Subtract with Signed Saturation Word
PADDUB	Parallel Add with Signed Saturation Byte
PSUBUB	Parallel Subtract with Signed Saturation Byte
PADDUH	Parallel Add with Unsigned Saturation Halfword
PSUBUH	Parallel Subtract with Unsigned Saturation Halfword
PADDUW	Parallel Add with Unsigned Saturation Word
PSUBUW	Parallel Subtract with Unsigned Saturation Word

Multimedia Instructions: Multiply and Divide

Mnemonic	Description
PMULTW	Parallel Multiply Word
PMULTUW	Parallel Multiply Unsigned Word
PDIVW	Parallel Divide Word
PDIVUW	Parallel Divide Unsigned Word
PMADDW	Parallel Multiply/Add Word
PMADDUW	Parallel Multiply/Add Unsigned Word
PMSUBW	Parallel Multiply/Subtract Word
PMFHI	Parallel Move From HI
PMFLO	Parallel Move From LO
PMTHI	Parallel Move To HI
PMTLO	Parallel Move To LO
PMULTH	Parallel Multiply Halfword
PMADDH	Parallel Multiply/Add Halfword
PMSUBH	Parallel Multiply/Subtract Halfword
PMFHL	Parallel Move From HI/LO
PMTHL	Parallel Move To HI/LO
PHMADH	Parallel Horizontal Multiply/Add Halfword
PHMSBH	Parallel Horizontal Multiply/Subtract Halfword
PDIVBW	Parallel Divide Broadcast Word

Multimedia Instructions: Shift Operations

Mnemonic	Description
MFSA	Move From SA Register
MTSA	Move To SA Register
MTSAB	Move Byte Count to SA Register
MTSAH	Move Halfword Count to SA Register
PSLLH	Parallel Shift Left Logical Halfword
PSRLH	Parallel Shift Right Logical Halfword
PSRAH	Parallel Shift Right Arithmetic Halfword
PSLLW	Parallel Shift Left Logical Word
PSLLVW	Parallel Shift Left Logical Variable Word
PSRLW	Parallel Shift Right Logical Word
PSRLVW	Parallel Shift Right Logical Variable Word
PSRAW	Parallel Shift Right Arithmetic Word
PSRAVW	Parallel Shift Right Arithmetic Variable Word
QFSRV	Quadword Funnel Shift Right Variable

Multimedia Instructions: Others

Mnemonic	Description
PABSH	Parallel Absolute Halfword
PABSW	Parallel Absolute Word
PMAXH	Parallel Maximum Halfword
PMINH	Parallel Minimum Halfword
PMAXW	Parallel Maximum Word
PMINW	Parallel Minimum Word
PAND	Parallel AND
POR	Parallel OR
PXOR	Parallel XOR
PNOR	Parallel NOR

Multimedia Instructions: Compare

Mnemonic	Description
PCGTB	Parallel Compare for Greater Than Byte
PCEQB	Parallel Compare for Equal Byte
PCGTH	Parallel Compare for Greater Than Halfword
PCEQH	Parallel Compare for Equal Halfword
PCGTW	Parallel Compare for Greater Than Word
PCEQW	Parallel Compare for Equal Word
PLZCW	Parallel Leading Zero Count Word

Multimedia Instructions: Load/Store

Mnemonic	Description
LQ	Load Quadword
SQ	Store Quadword

Multimedia Instructions: Data Rearrangement

Mnemonic	Description
PPACB	Parallel Pack To Byte
PPACH	Parallel Pack To Halfword
PINTEH	Parallel Interleave Even Halfword
PPACW	Parallel Pack To Word
PEXTUB	Parallel Extend Upper From Byte
PEXTLB	Parallel Extend Lower From Byte
PEXTUH	Parallel Extend Upper From Halfword
PEXTLH	Parallel Extend Lower From Halfword
PEXTUW	Parallel Extend Upper From Word
PEXTLW	Parallel Extend Lower From Word
PEXT5	Parallel Extend from 5 bits
PPAC5	Parallel Pack to 5 bits
PCPYH	Parallel Copy Halfword
PCPYLD	Parallel Copy Lower Doubleword
PCPYUD	Parallel Copy Upper Doubleword
PREVH	Parallel Reverse Halfword
PINTH	Parallel Interleave Halfword
PEXEH	Parallel Exchange Even Halfword
PEXCH	Parallel Exchange Center Halfword
PEXEW	Parallel Exchange Even Word
PEXCW	Parallel Exchange Center Word
PROT3W	Parallel Rotate 3 Word

Table 2-11 Multimedia Instructions

2.12. Latency

The execution cycles of the EE Core instructions are listed below.
Obtaining cycles actually required for the program is difficult, for the following reasons:

- Determination if two instructions are issued simultaneously is required
- It is impossible to predict the latency at the time of a cache miss

Integer Instructions

Instruction Category	Execution Pipe	Latency	Throughput
Add/Sub, and logical operations	I0/I1	1	1
Transfer to HI / LO registers	I0/I1	1	1
Shift / LUI	I0/I1	1	1
Branch / Jump	BR	1	1
Conditional Move	I0/I1	1	1
MULT / MULTU	I0	4*	2
MULT1 / MULTU1	I1	4*	2
DIV / DIVU	I0	37*	37
DIV1 / DIVU1	I1	37*	37
MADD / MADDU	I0	4*	2
MADD1 / MADDU1	I1	4*	2
Load**	LS	1	1
Store**	LS	-	1
Multimedia Multiply	I0+I1	4	2
Multimedia Divide	I0+I1	37	37

* The latency for HI, LO, HI1, LO1, and GPR that store operation results

** When there is no cache miss.

Floating-Point Instructions

Instruction Category	Execution Pipe	Latency	Throughput
MTC1	C1+LS	2	1
Add / Sub / Abs / Neg / C.cond	C1	4	1
CVT	C1	4	1
Mul	C1	4	1
MFC1	C1+LS	2	1
Move	C1	4	1
DIV.S	C1	8	7
SQRT.S	C1	8	7
RSQRT.S	C1	14	13
MADD	C1	4	1
LWC1*	C1+LS	2	1

* When there is no cache miss.

3. Registers

This chapter describes registers in the CPU and the System Control Coprocessor (COP0). Refer to Chapter 8 and the "VU User’s Manual" for the FPU (COP1) registers and the VPU (COP2) registers, respectively.

The CPU registers group consists of the following:

- General-Purpose Registers (GPRs)
- HI and LO registers that hold the results of integer multiply and divide operations
- The SA register that is used by the funnel shift instructions
- The Program Counter (PC) register

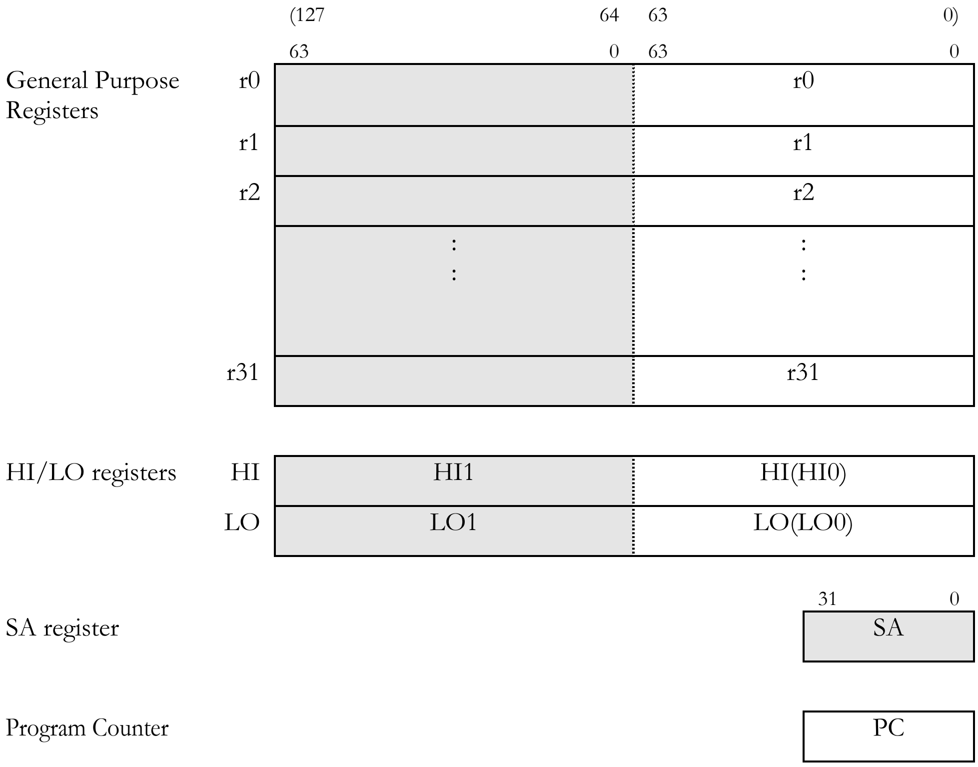
The COP0 registers control the processor state and hold its status. These registers can be read using the MFC0 instruction and written using the MTC0 instruction.

3.1. CPU Registers

The EE Core provides the following registers:

- 32 128-bit General Purpose Registers (GPRs)
- Four registers that hold the results of integer multiply and divide operations (HI0, LO0, HI1, and LO1)
- Shift Amount (SA) register
- Program Counter (PC)

The EE Core has 128-bit-wide General Purpose Registers (GPRs). The upper 64-bits of the GPRs are only used by the EE Core-specific instructions (The "Load/Store Quadword" and "Multimedia" instructions).
The HI and LO registers have also been extended to 128-bit-wide. The lower 64 bits are HI0/LO0 registers, which correspond to the standard 64-bit HI and LO registers. The upper 64 bits, as HI1/LO1 registers, are only used by the newly defined multiply and divide instructions. These instructions are MULT1, MULTU1, DIV1, DIVU1, MADD1, MADDU1, MFHI1, MFLO1, MTHI1, MTLO1 and so on. All these instructions are equivalent to existing instructions that use HI0 and LO0 registers. The SA register specifies the shift amount used by the funnel shift (QFSRV) instruction.



* The EE Core specific instructions are grayed.

Figure 3-1 CPU Registers

3.1.1. General Purpose Registers

The standard 64-bit general-purpose registers have been extended to 128-bit registers. New instructions have been defined to use the upper 64-bits of these registers.
Two of the general-purpose registers, r0 and r31, have the following special features:

- r0 is hardwired to a value of zero. When a zero value is needed, r0 can be used as a source, and a destination of an instruction whose result is not necessary.
- r31 is the link register used by the Jump and Link instructions. It should not used by other instructions.

3.1.2. HI and LO Registers

The standard 64-bit HI and LO registers have been expanded to 128-bit registers. New instructions have been defined to use the upper 64-bits of these registers.
The HI and LO registers consist of the upper 64 bits (HI1/LO1) and the lower 64 bits (HI0/LO0), and can be used separately. HI0 and LO0 are equivalent to the standard 64-bit HI and LO registers.
HI and LO registers hold the results of integer multiply, integer multiply-accumulate, and integer divide operations. In integer divide operations, the quotient and remainder are stored in LO0 and LO1, and HI0 and HI1, respectively.

3.1.3. SA Register

The SA register specifies the shift amount when shifting or rotating (funnel shifting) a 128-bit data using the QFSRV instruction. The register is EE Core-specific, but it needs to be saved and restored as a part of the processor state.
New instructions have been defined for a data transfer between the SA register and the general-purpose registers.

3.1.4. Program Counter (PC)

The Program Counter (PC) holds the address of the instruction that is being executed. The PC is incremented automatically by 4 when a normal instruction is executed. When the jump or branch instruction is executed, the PC is changed to the specified target address. When an exception occurs, the value of the PC is changed to an exception vector address.

3.2. System Control Coprocessor (COP0) Registers

As shown in Table 3-1, the system control coprocessor (COP0) has registers related to memory management and exception handling. Details of each register are described later. For details, refer to "4. Exception Processing" and "5. Memory Management".

No.	Register Name	Description	Purpose
0	Index	Index that specifies TLB entry for reading or writing	MMU
1	Random	Pseudo-random index for TLB replacement	MMU
2	EntryLo0	Low half of TLB entry (for even PFN)	MMU
3	EntryLo1	Low half of TLB entry (for odd PFN)	MMU
4	Context	Pointer to PTE table	MMU
5	PageMask	Most significant part of the TLB entry (page size mask)	MMU
6	Wired	Number of wired TLB entries	MMU
7	(Reserved)	Undefined	-
8	BadVAddr	Bad virtual address	Exception
9	Count	Timer compare	Exception
10	EntryHi	High half (Virtual page number and ASID) of TLB entry	MMU
11	Compare	Timer reference value	Exception
12	Status	Processor Status Register	Exception
13	Cause	Result of the last exception taken	Exception
14	EPC	Exception Program Counter	Exception
15	PRId	Processor Revision Identifier	MMU
16	Config	Configuration Register	MMU
17 - 22	(Reserved)	Undefined	-
23	BadPAddr	Bad physical address	Exception
24	Debug	Registers related to debug function	Debug
25	Perf	Performance Counter and Control Register	Exception
26 - 27	(Reserved)	Undefined	-
28	TagLo	Low bits of the Cache Tag	Cache
29	TagHi	High bits of the Cache Tag	Cache
30	ErrorEPC	Error Exception Program Counter	Exception
31	(Reserved)	Undefined	-

Table 3-1 COP0 Registers

There are 7 registers related to the debug function and 3 registers related to the performance counter. They are assigned to Register Nos. 24 and 25 respectively, and accessed via specific instructions (MFC0/MTC0 instruction variations).

Index : Index that specifies TLB entry for reading or writing

CPR[0,00]

31	30 … 6	5 … 0
P	0	Index
1	25	6

Name	Pos.	Contents	r/w	Initial Value
Index	5:0	Index that points to the TLB entry for TLB reading or writing	r/w	Undefined

The Index register points to the TLB entry for the TLB Read (TLBR) or TLB Write (TLBWI) Instructions.

Random : Index that specifies TLB entry for the TLBWR instruction

CPR[0,01]

31 … 6	5 … 0
0	Random
26	6

Name	Pos.	Contents	r/w	Initial Value
Random	5:0	TLB Random Index	r/-	47

The Random register specifies the TLB entry for the TLB Write Random (TLBWR) instruction.
The value of the Random register decrements every cycle in which an instruction is executed. Its value ranges between an upper bound and a lower bound. The lower bound is the value the Wired register indicates. The upper bound is set by the total number of TLB entries (47).
The Random register is set to the value of the upper bound upon system reset and when the Wired register is written.

EntryLo0 / EntryLo1 : Lower part of the TLB entry

CPR[0,02] / CPR[0,03]

EntryLo0

31	30 26	25 6	5 3	2	1	0
S	0	PFN	C	D	V	G
1	5	20	3	1	1	1

EntryLo1

31 26	25 6	5 3	2	1	0
0	PFN	C	D	V	G
6	20	3	1	1	1

Name	Pos.	Description	r/w	Initial Value
S	31 (EntryLo0)	Memory type 0 Main memory 1 Scratchpad RAM	r/w	Undefined
PFN	25:6	Page frame number (the upper bits of the physical address)	r/w	Undefined
C	5:3	TLB page coherency attribute (Cache mode) 000(0) Reserved 001(1) Reserved 010(2) Uncached 011(3) Cached, write-back, with write allocate 100(4) Reserved 101(5) Reserved 110(6) Reserved 111(7) Uncached Accelerated	r/w	Undefined
D	2	Dirty bit (The bit is set to 1 if writable)	r/w	Undefined
V	1	Valid bit (The bit is set to 1 if the TLB entry is enabled)	r/w	Undefined
G	0	Global bit	r/w	Undefined

The EntryLo0 and EntryLo1 registers correspond to the lower part of each TLB entry. EntryLo0 and EntryLo1 are used for even and odd virtual pages, respectively. The C bit indicates cache modes for the virtual pages. When the reserved values in the above table are set, TLB entries may not be set correctly. The Dirty bit indicates whether or not a write to virtual pages is possible. If the bit is cleared to 0, it can be write-protected.
The Global bit indicates whether or not ASID is used during TLB look-up. If both the Global bit of EntryLo0 and EntryLo1 are set to 1, the processor ignores the ASID during TLB lookup.

Context : TLB miss handling information

CPR[0,04]

31 23	22 4	3 0
PTEBase	BadVPN2	0
9	19	4

Name	Pos.	Description	r/w	Initial Value
PTEBase	31:23	Page table address	r/w	Undefined
BadVPN2	22:4	Virtual page address that caused TLB miss	r/-	Undefined

The OS uses the Context register as a pointer to a page table when handling a TLB miss.
The page table is an OS-managed data structure that holds the corresponding information of virtual and physical addresses. In the case of address translation, the corresponding information is first searched for in TLB. If there is no information in TLB, a TLB miss (TLB Refill exception) occurs. Then the OS refers to the Context register and reads the corresponding information not in TLB from the page table. For details, refer to the description in "5.2. Address Translation".
The PTEBase field sets a page address of the page table. Normally, kseg3 (Kernel mode / Kernel space 3) is used.
In the BadVPN2 field, bits 31:13 of the virtual address, which has caused the TLB miss, are set automatically. (The virtual address, bit 12, is excluded because a single TLB entry corresponds to an even-odd page pair).
If a page size is 4KB, the value of the Context register configured as above becomes directly the address of the applicable entry of a page table (1 entry = 8 bytes). For other page and PTE sizes, shifting and masking this value produces the appropriate address.

PageMask : Page size comparison mask

CPR[0,05]

31 25	24 13	12 0
0	MASK	0
7	12	13

Name	Pos.	Description	r/w	Initial Value
MASK	24:13	Page size comparison mask 0000 0000 0000 4KB 0000 0000 0011 16KB 0000 0000 1111 64KB 0000 0011 1111 256KB 0000 1111 1111 1MB 0011 1111 1111 4MB 1111 1111 1111 16MB	r/w	Undefined

The PageMask register corresponds to the MASK field of each TLB entry. It holds a comparison mask that indicates a page size.
During TLB look-up for translating virtual addresses into physical addresses, the value of the MASK field in TLB identifies the effective bits among bits 24:13. When the value of the MASK field is not one of the values shown in the table above, the operation of TLB is undefined. For details, refer to the description in "5.2. Address Translation".
The TLB read (TLBR) instruction uses the PageMask register as a destination, and the TLB write (TLBWI / TLBWR) instruction uses it as a source.
If the S bit of EntryLo0 is 1 (the page that corresponds to scratchpad RAM) on the TLB write operation, the contents of the PageMask register are all zeros. When reading the TLB entry that corresponds to scratchpad RAM (the S bit of EntryLo0 register is 1), the values of the PageMask register are undefined.

Wired : The number of Wired TLB entries

CPR[0,06]

Name	Pos.	Description	r/w	Initial Value
Wired	5:0	The number of wired TLB entries	r/w	0

The Wired register specifies the boundary between the wired and random entries of the TLB. The value of the Random register, which becomes the index for the TLB Random write (TLBWR), ranges from the largest number of the TLB entry (47) to the value of the Wired register. Therefore, the TLB entry whose number is smaller than the value of the Wired register cannot be overwritten by the TLBWR instruction.
The Wired register is set to 0 upon system reset. When setting the value in the Wired register, the upper limit (47) is set in the Random register. Writing a value greater than 47 into the Wired register produces undefined results.

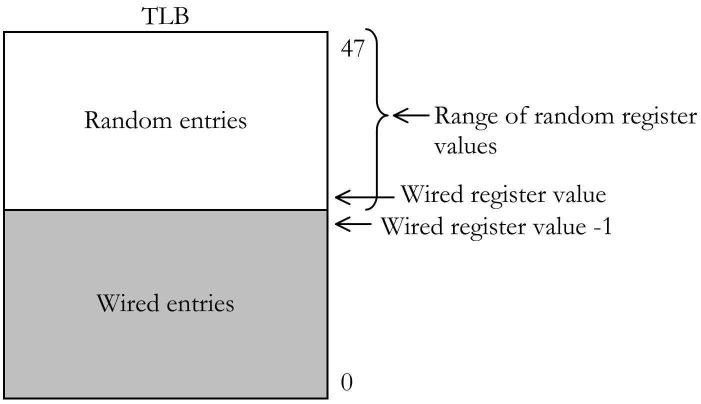


Figure 3-2 Wired TLB Entries

BadVAddr : Virtual address that causes an error

CPR[0,08]

31 0
BadVAddr
32

Name	Pos.	Description	r/w	Initial Value
BadVAddr	31:0	Virtual address that has caused the most recent TLB Invalid, TLB Modified, TLB Refill, or Address Error exception.	r/-	Undefined

In the BadVAddr register, when the TLB Invalid, TLB Modified, TLB Refill, or Address Error exception occurs, its virtual address is stored.
Since bus errors are not addressing errors, the BadVAddr register does not have any information about bus errors.

Count : Timer count value

CPR[0,09]

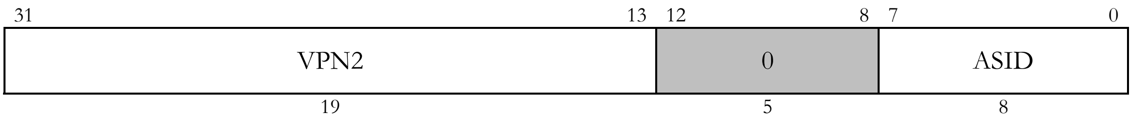
31	0
Count	
32	

Name	Pos.	Description	r/w	Initial Value
Count	31:0	Timer count value	r/w	Undefined

The Count register is a real-time timer register that is incremented every CPU clock cycle. When the value is equal to the value of the Compare register, the timer interrupt is signaled through IP[7]. (This interrupt can be disabled through the interrupt mask bit, IM[7].)

EntryHi : Upper parts of a TLB entry

CPR[0,10]



Name	Pos.	Description	r/w	Initial Value
VPN2	31:13	Virtual page number divided by two	r/w	Undefined
ASID	7:0	Address space ID field	r/w	Undefined

The EntryHi register corresponds to the upper parts of each TLB entry.
When mapping a single virtual address to different physical addresses per process is desirable, the ASID field can be used as an additional part of the virtual address.
When a TLB Refill, TLB Invalid, or TLB Modified exception occurs, the virtual page number (VPN2) and ASID that do not match a TLB entry are stored in the EntryHi register.

Compare : Timer stable value

CPR[0,11]

31   0
Compare
32

Name	Pos.	Description	r/w	Initial Value
Compare	31:0	Timer stable value	r/w	Undefined

The Compare register acts as a timer, in addition to the Count register. It maintains a stable value. When the value of the Count register incremented every CPU cycle equals the stable value, a timer interrupt occurs. (To be more precise, when the value of the Compare register equals the value of the Count register, the interrupt bit IP[7] of the Cause register is set. If an interrupt is enabled, a timer interrupt occurs.)
As a side effect, writing to the Compare register clears the timer interrupt.
The Compare register is a read/write register. In normal use, however, the Compare register is write-only.

Status : COP0 Status

COP[0,12]

31 28	27 24	23	22	21 19	18	17	16	15	14 13	12	11 10	9 5	04 03	02	01	00
CU	0	DEV	BEV	0	CH	EDI	EIE	IM7	0	BEM	IM	0	KSU	ERL	EXL	IE
4	4	1	1	3	1	1	1	1	2	1	2	5	2	1	1	1

Name	Pos.	Description	r/w	Initial Value
CU (CU[3:0])	31:28	Usability of each of coprocessor units. 0 Unusable 1 Usable	r/w	Undefined
DEV	23	Address of Performance counter and debug exception vectors (1→bootstrap).	r/w	Undefined
BEV	22	Address of the TLB Refill exception or general exception vectors (1→bootstrap).	r/w	1
CH	18	Status of the most recent Cache instructions (Cache Hit Invalidate / Cache Hit Write-back Invalidate) for the data cache. 0 miss 1 hit	r/w	Undefined
EDI	17	EI/DI instruction enable 0 enabled only in Kernel mode 1 enabled in all modes	r/w	Undefined
EIE	16	Enable IE bit 0 disables the IE bit (disables all interrupts regardless of the value of the IE bit) 1 enables the IE bit		Undefined
IM[7,3:2]	15, 11:10	Interrupt Mask 0 disables interrupts 1 enables interrupts	r/w	Undefined
BEM	12	Bus Error Mask 0 signals a bus error 1 masks a bus error	r/w	Undefined
KSU	4:3	Operation modes 00 Kernel mode 01 Supervisor mode 10 User mode 11 (Reserved)	r/w	Undefined
ERL	2	Error Level (set by the processor when Reset, NMI, performance counter, or debug exception is taken).	r/w	1
EXL	1	Exception Level: (set by the processor when any exception other than Reset, NMI, performance counter, or debug exception is taken).	r/w	Undefined
IE	0	Interrupt Enable flag 0 disables all interrupts 1 enables all interrupts	r/w	Undefined

The Status register indicates operating mode, interrupt enabling, and COP0 processor status. The following paragraphs describe some of the important fields.
The CU field controls the usability of COP0 to COP3. Regardless of the setting of the CU[0] (bit 28), COP0 is always usable in Kernel mode. For other cases, an access to an unusable coprocessor causes an exception. In the EE Core, COP3 is always unusable.
The EDI bit controls the EI and DI instructions, which enable and disable interrupts, except NMI, respectively. If this bit is 1, the EI and DI instructions are usable in User, Supervisor, and Kernel modes. If this bit is 0, the EI and DI instructions are usable only in Kernel mode, and operate as NOP in the User and Supervisor modes.
The EIE bit and IE bit control interrupt enabling. Only when both bits are set to 1 (in addition, when the ERL and EXL bits are 0), interrupts are enabled.
The IM field controls the usability of three interrupt signals. IM[7] (bit 15) and IM[3:2] (bit 11:10) correspond to the internal timer interrupt and Int[1:0] signals, respectively.
The processor recognizes an interrupt only when the corresponding IM bits, the IP bit of the Cause register, and the IE field of the Status register are set to 1. Note that the EE Core does not support software interrupts.
The BEM bit controls an update of the BadPAddr register (COP[0,23]) and a signal of the bus error exception. When the bit is 0, updating the BadPAddr register and signaling the bus error exception takes place. (At the same time, this bit is set to 1.) When the bit is 1, updating the BadPAddr register and signaling the bus error exception does not occur.
The KSU field (bit 4:3), along with the ERL and EXL bits, indicates the operation mode of the processor.

KSU	ERL	EXL	Operation mode
10	0	0	User mode
01	0	0	Supervisor mode
00	0	0	Kernel mode
(any)	0	1	Kernel mode (level 1 exception handler)
(any)	1	(any)	Kernel mode (level 2 exception handler)

According to the operation mode, accessible address spaces are restricted as follows:

Operation mode	User address space	Supervisor address space	Kernel address space
User mode	Yes	No	No
Supervisor mode	Yes	Yes	No
Kernel mode	Yes	Yes	Yes

Cause : Cause of the most recent exception

COP [0,13]

31	30	29 28		18 16	15		11 10		06 02	00
BD	BD2	CE	0	EXC2	IP[7]	0	IP[3:2]	0	ExcCode	0
1	1	2	9	3	1	3	2	3	5	2

Name	Pos.	Description	r/w	Initial Value
BD	31	Set by the processor when any exception other than Reset, NMI, performance counter, or debug exception occurs in a branch delay slot.	r/-	Undefined
BD2	30	Set by the processor when NMI, performance counter, or debug exception occurs in a branch delay slot.	r/-	Undefined
CE	29:28	Coprocessor number when a Coprocessor Unusable exception is taken.	r/-	Undefined
EXC2	18:16	Exception codes for level 2 exceptions 000 (0) : Res (Reset) 001 (1) : NMI (Non-maskable Interrupt) 010 (2) : PerfC (Performance Counter) 011 (3) : Dbg (Debug) 1xx (4-7): (Reserved)	r/-	Undefined
IP[7]	15	Set when a timer interrupt is pending.	r/-	Undefined
IP[3]	11	Set when the Int[0] interrupt is pending.	r/-	Undefined
IP[2]	10	Set when the Int[1] interrupt is pending.	r/-	Undefined
ExcCode	6:2	Exception codes 00000 (0): Int (Interrupt) 00001 (1): Mod (TLB Modified) 00010 (2): TLBL (TLB Refill(instruction fetch or load)) 00011 (3): TLBS (TLB Refill(store)) 00100 (4): AdEL (Address error(instruction fetch or load)) 00101 (5): AdES (Address error(store)) 00110 (6): IBE (Bus error(instruction)) 00111 (7): DBE (Bus error(data)) 01000 (8): Sys (System call) 01001 (9): Bp (Breakpoint) 01010 (10): RI (Reserved instruction) 01011 (11): CpU (Coprocessor Unusable) 01100 (12): Ov (Overflow) 01101 (13): Tr (Trap) (14-31): (Reserved)	r/-	Undefined

The Cause register indicates information about the cause of the most recent exception.

EPC : Address generating exceptions

CPR[0,14]

31	0
EPC	
32	

Name	Pos.	Description	r/w	Initial Value
EPC	31:0	Address that is to resume after an exception has been serviced.	r/w	Undefined

The EPC register is set automatically when an exception occurs, and indicates the address that is to be restored from the exception handler. If an exception is precise, the value of the EPC register will be one of the following:

- The virtual address of the instruction that was the direct cause of the exception
- The virtual address of the immediately preceding branch or jump instruction (the BD bit in the Cause register is set then)

Note that if the EXL bit in the Status register is set to 1, the value of the EPC register is not updated even when an exception occurs.

PRId : Processor Revision

CPR[0,15]

31 – 16	15 – 8	7 – 0
0	Imp	Rev
16	8	8

Name	Pos.	Description	r/w	Initial Value
Imp	15:8	Implementation number	r/-	0x2E
Rev	7:0	Revision number	r/-	Revision number

The PRId register contains the implementation and revision information of the EE Core and COP0.
The value of the Imp field is fixed to 0x2e, which indicates the EE Core.
The Rev field is a revision number of a chip architecture. Hi-order 4 bits (bits 7:4) and low-order 4 bits (bits 3:0) indicate a major and minor revision number, respectively.
Note that there is no guarantee that changes to the chip will be reflected in the PRId register, or that changes to the revision number will indicate chip changes. For this reason, the software should not rely on the value of the PRId register.

Config : Processor Configuration

CPR[0,16]

31	30 – 28	27 – 19	18	17	16	15 – 14	13	12	11 – 09	08 – 06	05 – 03	02 – 00
0	EC	0	DIE	ICE	DCE	0	NBE	BPE	IC	DC	0	K0
1	3	9	1	1	1	2	1	1	3	3	3	3

Name	Pos.	Description	r/w	Initial Value
EC	30:28	Bus clock ratio 000 Value from dividing the processor clock frequency by 2	r/-	0
DIE	18	Setting this bit to 1 enables the pipeline parallel issue. 0 Single issue 1 Double issue	r/w	0
ICE	17	Setting this bit to 1 enables the instruction cache. 0 Instruction cache disable 1 Instruction cache enable	r/w	0
DCE	16	Setting this bit to 1 enables the data cache 0 Data cache disable 1 Data cache enable	r/w	0
NBE	13	Setting this bit to 1 enables non-blocking load 0 Enables Blocking loads 1 Enables Non-blocking loads and hit under miss	r/w	0
BPE	12	Setting this bit to 1 enables branch prediction 0 Disables Branch Prediction 1 Enables Branch Prediction	r/w	0
IC	11:9	Instruction cache size 010 16KB	r/-	010
DC	8:6	Data cache size 001 8KB	r/-	001
K0	2:0	kseg0 cache mode 000 Cached without writeback and write allocate 010 Uncached 011 Cached with writeback and write allocate 111 Uncached Accelerated (Others: Reserved)	r/w	Undefined

The Config register sets various options concerning processor operation and configuration. The EC, IC, and DC fields are related to the hardware configuration, and are set by hardware on reset. Software cannot change the settings.

When the DIE bit is cleared to 0, the EE Core uses one of two pipelines, and issues one instruction per cycle.

The ICE and DCE bits specify if the instruction cache and data cache are enabled or disabled, respectively. Specifying them is given higher priority than specifying a cache mode in the K0 filed or TLB entry.

BadPAddr : Physical address that caused an error

CPR[0,23]

31 ... 4	0
BdPAddr	
28	4

Name	Pos.	Description	r/w	Initial Value
BdPAddr	31:4	Physical address value	r/w	Undefined

The BadPAddr register contains the most recent physical address that caused a bus error.
However, the error address is set only when Status.BEM is cleared to 0. When Status.BEM is set to 1, a bus error is masked, and the value of this register cannot be changed.

BPC：Control of the breakpoint function

CCR[0,24]-0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14 … 03	02	01	00
IAE	DRE	DWE	DVE	0	IUE	ISE	IKE	IXE	0	DUE	DSE	DKE	DXE	ITE	DTE	BED	0	DWB	DRB	IAB
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	12	1	1	1

Name	Pos.	Description	r/w	Initial Value
IAE	31	Instruction Address breakpoint Enable	r/w	0
DRE	30	Data Read breakpoint Enable	r/w	0
DWE	29	Data Write breakpoint Enable	r/w	0
DVE	28	Data Value breakpoint Enable	r/w	Undefined
IUE	26	Instruction address breakpoint - User mode Enable	r/w	Undefined
ISE	25	Instruction address breakpoint - Supervisor mode Enable	r/w	Undefined
IKE	24	Instruction address breakpoint - Kernel mode Enable	r/w	Undefined
IXE	23	Instruction address breakpoint - EXL mode Enable	r/w	Undefined
DUE	21	Data breakpoint - User mode Enable	r/w	Undefined
DSE	20	Data breakpoint - Supervisor mode Enable	r/w	Undefined
DKE	19	Data breakpoint - Kernel mode Enable	r/w	Undefined
DXE	18	Data breakpoint - EXL mode Enable	r/w	Undefined
ITE	17	Instruction address breakpoint - Trigger generation Enable	r/w	Undefined
DTE	16	Data breakpoint - Trigger generation Enable	r/w	Undefined
BED	15	Breakpoint Exception Disable	r/w	Undefined
DWB	2	Data Write Breakpoint establishment flag	r/w	Undefined
DRB	1	Data Read Breakpoint establishment flag	r/w	Undefined
IAB	0	Instruction Address Breakpoint	r/w	Undefined

The BPC register controls the breakpoint functions, and has a flag that indicates the establishment of a breakpoint. BPC is assigned to COP0 register 24 along with other breakpoint-related registers. Instead of MFC0 and MTC0 instructions, MFBPC and MTBPC are used to read or write the contents.

The IAE, IUE, ISE, IKE, or IXE bit determines whether or not to judge the establishment of the instruction address breakpoint.

The DWE, DRE, DUE, DSE, DKE, or DXE bit determines whether or not to judge the establishment of the data address breakpoint. Also, the DVE bit determines whether or not to judge the establishment of the data value breakpoint.

The ITE, DTE, or BED bit sets the operation when the breakpoint is established.

The DWB, DRB, or IAB bit is a flag that indicates that a breakpoint has been established. Bits 27, 22 and 14 are reserved and must be set to 0.

IAB / IABM : Instruction address breakpoint settings

CCR[0,24]-1/-2

31 … 0
32

The IAB and IABM registers indicate the establishment condition of the instruction address breakpoint. The IAB and IABM registers specify a break address and the mask that indicates valid bits among the break address, respectively.

The IAB and IABM registers, along with other breakpoint-related registers, are assigned to COP0 register 24 in duplicate. To read/write the contents, the MFIAB / MTIAB and MFIABM / MTIABM instructions are used instead of the MFC0 / MTC0 instruction.

DAB / DABM : Data address breakpoint settings

CCR[0,24]-4/-5

31 0
32

The DAB and DABM registers indicate the establishment condition of the data address breakpoint. The DAB and DABM registers specify a break address and the mask that indicates the valid bits among the break address, respectively.

The DAB and DABM registers, along with other breakpoint-related registers, are assigned to COP0 register 24 in duplicate. To read/write the contents, the MFDAB / MTDAB and MFDABM / MTDABM instructions are used instead of the MFC0 / MTC0 instruction.

DVB / DVBM : Data value breakpoint settings

CCR[0,24]-6/-7

31	0
32	

The DVB and DVBM registers indicate the establishment condition of the data value breakpoint. The DVB and DVBM registers specify a break value and the mask that indicates valid bits among the break value.
The DVB and DVBM registers, along with other breakpoint related registers, are assigned to COP0 register 24 in duplicate. To read/write the contents, the MFDVB / MTDVB and MFDVBM / MTDVBM instructions are used instead of the MFC0 / MTC0 instruction.

PCCR : Performance Counter Control

CCR[0,25]

31		19 15	14	13	12	11		9 5	4	3	2	1	
CTE	0	EVENT1	U1	S1	K1	EXL1	0	EVENT0	U0	S0	K0	EXL0	0
1	11	5	1	1	1	1	1	5	1	1	1	1	1

Name	Pos.	Description	r/w	Initial Value
CTE	31	Enables counter function	r/w	0
EVENT1	19:15	Events that counted in CTR1 (Refer to the table below)	r/w	Undefined
U1	14	Enables a counting operation in CTR1 in the User mode.	r/w	Undefined
S1	13	Enables a counting operation in CTR1 in the Supervisor mode.	r/w	Undefined
K1	12	Enables a counting operation in CTR1 in the Kernel mode.	r/w	Undefined
EXL1	11	Enables a counting operation in CTR1 when executing the level 1 exception handler.	r/w	Undefined
EVENT0	9:5	Events that counted in CTR0 (Refer to the table below)	r/w	Undefined
U0	4	Enables a counting operation in CTR0 in the User mode.	r/w	Undefined
S0	3	Enables a counting operation in CTR0 in the Supervisor mode.	r/w	Undefined
K0	2	Enables a counting operation in CTR0 in the Kernel mode.	r/w	Undefined
EXL0	1	Enables a counting operation in CTR0 when executing the level 1 exception handler.	r/w	Undefined

EVENT0 / EVENT1	Events to be counted	
	CTR0	CTR1
0	(reserved)	Issues the low-order branch
1	Processor cycle	Processor cycle
2	Issues a single instruction	Issues a double instruction
3	Issues branch	Branch prediction miss
4	BTAC miss	TLB miss
5	TLB miss	DTLB miss
6	Instruction cache (I$) miss	Data cache (D$) miss
7	Access to DTLB	WBB single request unusable
8	Non-blocking load	WBB burst request unusable
9	WBB single request	WBB burst request almost full
10	WBB burst request	WBB burst request full
11	CPU address bus busy	CPU data bus busy
12	Completes instruction	Completes instruction
13	Completes non-BDS instruction	Completes non-BDS instruction
14	Completes the COP2 instruction	Completes the COP1 instruction
15	Completes loads	Completes stores
16	No event	No event
17-31	(reserved)	(reserved)

The PCCR register controls the function of the performance counter. It determines which events are counted, and which operation modes are used for counting, for counter registers PCR0 and PCR1.

PCR0 / PCR1 : Performance Counter

CCR[0,25]

31	30 … 0
OVFL	VALUE
1	31

Name	Pos.	Description	r/w	Initial Value
OVFL	31	Overflow flag	r/w	Undefined
VALUE	30:0	Counter	r/w	Undefined

PCR0 and PCR1 are performance counter registers. They execute a count operation separately, according to the specification of the CCR register. When the VALUE field overflows, the OVFL bit is set. When the CCR.CTE bit is set to 1, the debug exception occurs.

TagLo : Lower parts of a cache tag

CCR[0,28]

The TagLo register, when operating a cache by the CACHE instruction, is used as follows:

Instruction		Processing		
CACHE BXLBT	Index Load BTAC	BTAC fetch address	->	TagLo[31:3]
		BTAC.V bit	->	TagLo[0]
CACHE BXSBT	Index Store BTAC	BTAC fetch address	<-	TagLo[31:3]
		BTAC.V bit	<-	TagLo[0]
CACHE DXLDT	Index Load Data	Data cache data	->	TagLo[31:0]
CACHE DXLTG	Index Load Tag	Data cache tag	->	TagLo
CACHE DXSDT	Index Store Data	Data cache data	<-	TagLo[31:0]
CACHE DXSTG	Index Store Tag	Data cache tag	<-	TagLo*
CACHE IXLDT	Index Load Data	Instruction cache data (instruction word)	->	TagLo[31:0]
CACHE IXLTG	Index Load Tag	Instruction cache tag	->	TagLo**
CACHE IXSDT	Index Store Data	Instruction cache data (instruction word)	<-	TagLo[31:0]
CACHE IXSTG	Index Store Tag	Instruction cache tag	<-	TagLo**

TagLo* indicates the following field structure which corresponds to the cache tag.
TagLo** has this field structure, except the D/L bits.

31 12	11 7	6	5	4	3	2 0
PTagLo	-	D	V	R	L	-
20	5	1	1	1	1	3

Name	Pos.	Description	r/w	Initial Value
PTagLo	31:12	Physical address tag cache	r/w	Undefined
D	6	Dirty bit	r/w	Undefined
V	5	Valid bit	r/w	Undefined
R	4	LRF bit	r/w	Undefined
L	3	Lock bit	r/w	Undefined

TagHi : Upper parts of a cache tag

CCR[0,29]

The TagHi register, when operating a cache by the CACHE instruction, is used as follows:

Instruction		Processing		
CACHE BXLBT	Index Load BTAC	BTAC target address	->	TagHi[31:2]
CACHE BXSBT	Index Store BTAC	BTAC target address	<-	TagHi[31:2]
CACHE IXLDT	Index Load Data	Instruction cache BHT	->	TagHi[5:4]
		Instruction cache steering bit	->	TagHi[3:0]
CACHE IXSDT	Index Store Data	Instruction cache BHT	<-	TagHi[5:4]
		Instruction cache steering bit	<-	TagHi[3:0]

ErrorEPC : Address generating a level 2 exception

CCR[0,30]

31 — 0
ErrorEPC
32

Name	Pos.	Description	r/w	Initial Value
ErrorEPC	31:0	Restart address after servicing a level 2 error	r/w	Undefined

The ErrorEPC register contains a return address from an exception handler when NMI, a debug, or a counter exception occurs.

The value of the ErrorEPC register normally becomes the address of the instruction in which an exception is generated. However, when an exception occurs to the instruction in the branch delay slot, the value becomes the address of the branch instruction immediately preceding the instruction in which an exception is generated. In order to identify this, the Cause.BD2 bit is set to 1.

4. Exception Processing

Any event that causes an instruction's execution to be interrupted is called an exception. This chapter discusses how the processor handles exceptions.
Because processor reset is one of the exceptions, reset semantics are also discussed in this chapter.

4.1. Exception Handling Process

This section describes the processor handling process when exceptions are recognized. Exceptions are divided into levels 1 and 2. The processing of level 1 differs from that of level 2 in details. Refer to "4.2. Exception Reference" for further information about each exception.

4.1.1. Exception Vector

The entry address of an exception handler (exception vector) is fixed at a particular address in memory. An exception vector has two types: one is normally used, and the other is used when bootstrapping. The usage depends on the value of the Status.BEV or Status.DEV bit (except Reset/NMI).
The list of exception vectors is shown below. Address 0 is normally used, and Address 1 is used when bootstrapping.

Exception Vector	Exception	Level	BEV	DEV	Address 0	Address 1
V_RESET_NMI	Reset / NMI	2	-	-	0xBFC00000	0xBFC00000
V_TLB_REFILL	TLB Refill *	1	0	-	0x80000000	-
			1	-	-	0xBFC00200
V_COUNTER	Performance Counter	2	-	0	0x80000080	-
			-	1	-	0xBFC00280
V_DEBUG	Debug	2	-	0	0x80000100	-
			-	1	-	0xBFC00300
V_COMMON	All other exceptions	1	0	-	0x80000180	-
			1	-	-	0xBFC00380
V_INTERRUPT	Interrupt	1	0	-	0x80000200	-
			1	-	-	0xBFC00400

* For the TLB Refill exception that is recognized when Status.EXL = 1 (i.e. in an exception handler), the V_COMMON vector is used.

Table 4-1 Exception Vector Types

4.1.2. Level 1 Exception Handling

When a level 1 exception is recognized, the following processes are executed:

- Switches to Kernel mode (Status.EXL <- 1)
- Saves addresses (sets EPC, Cause.BD)
- Sets exception cause codes (sets Cause.ExcCode, etc.)
- Jumps to the specified vector address

When a level 1 exception is recognized in an exception handler, an address is not saved. Also, in this case, the V_COMMON vector is applied to the TLB Refill exception.
Exceptions occur and are recognized in all modes; User, Supervisor, and Kernel. However, the processor is switched to Kernel mode before executing the first instruction of an exception handler. This switching takes place by setting the Status.EXL bit, not the Status.KSU bit, to 1. When Status.EXL is 1, the operating mode is Kernel mode, regardless of the setting of Status.KSU. If executing the ERET instruction when an exception handler finishes, the EXL is cleared to zero, and the processor restores the original mode.
In addition to switching operating modes, the virtual address of the instruction cancelled by the exception is saved in the EPC register as a restart address, and the Cause.BD bit is cleared to 0. However, if the cancelled instruction is in the delay slot of a branch instruction, the virtual address of the branch instruction, not the

cancelled instruction, is saved in the EPC register. The Cause.BD bit is set to 1. For this reason, when the exception handler examines the instruction address for an exception cause, it should consider the Cause.BD bit.
The code indicating an exception cause is determined in the 5-bit Cause.ExcCode field. In addition, with the coprocessor exception, the coprocessor number for the exception cause is set in the Cause.CE field.
Finally, it jumps to the vector address specified by the exception cause and the Status.BEV bit (see Table 4-1).
The operation of the exception handling is described in pseudo-C code as follows:

```
Level1_exception_base (int cause, int in_branch_delay)
{
 Cause.ExcCode = cause;   // set Level 1 exception cause

 // if already in exception handler (i.e. EXL==1),
 // do not update EPC and Cause.BD.
 // Furthermore, use general vector in this case.
 if ( Status.EXL ) {
   vector = V_COMMON;  // use general vector
 }
 else {  // normal Level 1 exception processing
   if ( in_branch_delay ) {    // Check for branch delay slot
     EPC = PC-4;
     Cause.BD = 1;
   }
   else {
     EPC = PC;
     Cause.BD = 0;
   }

   // Set to kernel mode, and disable interrupts
   Status.EXL = 1;

   // Select vector
   if ( cause == TLB_REFILL )
     vector = V_TLB_REFILL;
   else if ( cause == INTERRUPT )
     vector = V_INTERRUPT;
   else
     vector = V_COMMON;
 }

 // Select vector base according to Status.BEV bit.
 if ( Status.BEV )
   PC = 0xBFC00200 + vector;
 else
   PC = 0x80000000 + vector;
}
```

Once an exception service routine is entered, external interrupts (interrupt exceptions) other than NMI are disabled. An internal interrupt is generated and recognized if other level 1 exceptions are enabled in addition to level 2 exceptions (reset, NMI, performance counter, and debug). Note that the EPC register and Cause.BD bit are overwritten at this time.

4.1.3. Level 2 Exception Handling

Reset, NMI, performance counter, and debug exceptions, which may be generated during execution of the level 1 exception hander, are handled as level 2 exceptions. The contents of handling are similar to those of level 1, except that different registers are used.

First, switching to Kernel mode is executed by setting Status.ERL to 1. The instruction address, which is cancelled by an exception, is stored in the ErrorEPC register as a restart address, and the Cause.BD2 bit is cleared to 0. However, if the cancelled instruction is in the delay slot of a branch instruction, the Cause.BD2 bit is set to 1 and the address of the branch instruction is stored in the Error EPC register.

The cause code for an exception is stored in the Cause.EXC2 field.

The operation of the level 2 handling is described in pseudo-C code as follows:

```
Level2_exception_base (int cause, int in_branch_delay)
{
 Cause.EXC2 = cause; // set Level 2 exception cause

 if ( in_branch_delay ) {  // Check for branch delay slot
  ErrorEPC = PC-4;
  Cause.BD2 = 1;
 }
 else {
  ErrorEPC = PC;
  Cause.BD2 = 0;
 }

 // Set to Level 2 kernel mode
 Status.ERL = 1;

 // Jump to appropriate address
 if ( cause == RESET || cause == NMI )
  PC = 0xBFC00000;
 else {
  // Select non NMI/reset vector
  if ( cause == COUNTER )
   vector = V_COUNTER;
  else if ( cause == DEBUG )
   vector = V_DEBUG;

  // Select vector base according to Status.DEV bit.
  if ( Status.DEV )
   PC = 0xBFC00200 + vector;
  else
   PC = 0x80000000 + vector;
 }
}
```

If the level 2 exception handler is entered, NMI, interrupt, bus error, debug, and performance counter exceptions are disabled. (They are stored until the exception 2 handler finishes, and are recognized when finishing.) Programmers must set the program not to generate internal exceptions. When internal exceptions such as overflow occur during execution of the level 2 exception handler, the control is transferred to the corresponding level 1 handler. Both Status.EXL and Status.ERL will be set and the ERET instruction cannot operate properly.

4.1.4. Exception Priority

Exception priority rules determine which exception is taken first, if multiple internal exceptions occur simultaneously. Priority for internal and external exceptions in the EE Core is shown in Table 4-2. Note that this priority is from the pipeline's perspective. Since external exceptions occur regardless of an instruction execution, the priority between external and internal exceptions has little meaning.

Priority	Exception	Internal / External
Highest	Reset	External
	NMI	External
	Performance Counter	Internal
	Debug (Instruction Breakpoint)	Internal
	Address Error (Instruction Fetch)	Internal
	TLB Refill (Instruction Fetch)	Internal
	TLB Invalid (Instruction Fetch)	Internal
	Bus Error (Instruction Fetch)	Internal
	SYSTEMCALL, BREAK, Reserved Instruction, or Coprocessor Unusable	Internal
	Interrupt	External
	Debug (Data address/ Data value breakpoint)	Internal
	Overflow, Trap	Internal
	Address Error (Load / Store)	Internal
	TLB Refill (Load / Store)	Internal
	TLB Invalid (Load / Store)	Internal
	TLB Modified (Load / Store)	Internal
Lowest	Bus error (Load / Store)	Internal

Table 4-2 Exception Priority Order

4.2. Exception Reference

Exceptions for the EE Core are listed in the table below.

Exception	Level	Cause of Exception Code		Vector
		ExcCode	EXC2	
Reset	2	–	0	V_RESET_NMI
NMI	2	–	1	V_RESET_NMI
Performance Counter	2	–	2	V_COUNTER
Debug	2	–	4	V_DEBUG
Interrupt	1	0	–	V_INTERRUPT
TLB Modified	1	1	–	V_COMMON
TLB Refill (Instruction Fetch / Load)	1	2	–	V_TLB_REFILL
TLB Refill (Store)	1	3	–	V_TLB_REFILL
TLB Invalid (Instruction Fetch / Load)	1	2	–	V_COMMON
TLB Invalid (Store)	1	3	–	V_COMMON
Address error (Instruction Fetch / Load)	1	4	–	V_COMMON
Address Error (Store)	1	5	–	V_COMMON
Bus Error (Instruction Fetch)	1	6	–	V_COMMON
Bus Error (Load / Store)	1	7	–	V_COMMON
SYSTEMCALL	1	8	–	V_COMMON
BREAK	1	9	–	V_COMMON
Reserved Instruction	1	10	–	V_COMMON
Coprocessor Unusable	1	11	–	V_COMMON
Overflow	1	12	–	V_COMMON
Trap	1	13	–	V_COMMON

Table 4-3 Exception List

4.2.1. Reset Exception

Exception Level	Exception Vector Address	Cause Code
Level 2	V_RESET_NMI 0xBFC00000	Cause.EXC2 = 0

Cause of Exception

The Reset exception occurs when the Reset* signal is asserted and then deasserted. The exception is not maskable.

Operation

When the Reset exception occurs, registers in the COP0 are initialized as follows and the control is transferred to the V_RESET_NMI exception vector. The value of the bits and registers, not shown below, is undefined other than the bits fixed to 0.

- Status Register : Status.ERL=Status.BEV=1, Status.BEM=0
- Cause Register : Cause.EXC2=0
- Config Register : Config.DIE=Config.ICE=Config.DCE=Config.NBE=Config.BPE=0
- Random Register : Value of its upper bound (47)
- Wired Register : 0
- CCR Register : CCR.CTE=0
- BPC Register : BPC.IAE=BPC.DRE=BPC.DWE=0

The Valid, Dirty, LRF, and Lock bits of the data cache and the Valid and LRF bits of the instruction cache are all initialized to 0.

Handler Processing

The following should be handled in the Reset exception handler:

- Initializing the registers of the CPU and coprocessor, caches, and the memory system
- Performing diagnostic tests
- Bootstrapping the operating system

The Reset exception vector is located within uncached and unmapped address space. Therefore, the cache and TLB need not be initialized in order to process the exception handler.

4.2.2. NMI Exception

Exception Level	Exception Vector	Cause Code
Level 2	V_RESET_NMI 0xBFC00000	Cause.EXC2 = 1

Cause of Exception

The NMI (Non-Maskable Interrupt) exception is external, and occurs in the falling edge of the NMI* signal. The NMI exception is masked only when the level 2 exception handler is in process. It is recognized regardless of the settings of the Status.EXL and Status.IE bits.

Operation

When the NMI exception is recognized, the following registers are set, and the control is transferred to the V_RESET_NMI exception vector.

- Cause.EXC2 : 1
- ErrorEPC Register : Restart Address
- Cause.BD2 : When the exception occurs in the instruction of a branch delay slot, it is set to 1, and otherwise, 0
- Status Register : Status.ERL=Status.BEV=1

The contents of all registers, other than these above, are not modified.

Handler Processing

The NMI and Reset exceptions share the same exception vector. This vector is located within uncached and unmapped address space. Therefore, the cache and TLB need not be initialized in order to process the exception handler.

4.2.3. Performance Counter Exception

Exception Level	Exception Vector	Cause Code
Level 2	V_COUNTER DEV=0: 0x80000080 DEV=1: 0xBFC00280	Cause.EXC2 = 2

Cause of Exception

The Performance Counter exception occurs when the performance counter overflows or meets specified conditions. This exception is not maskable.

Operation

When the Performance Counter exception is recognized, each of the registers is set as follows, and the control is transferred to the V_ COUNTER exception vector:

- Cause.EXC2 : 2
- Cause.BD2 : When the exception is caused in the branch delay slot, it is set to 1, otherwise, 0
- ErrorEPC Register : Restart Address (the address of the instruction causing exception. However, the address of the preceding conditional branch instruction if Cause.BD2 is 1.)

Handler Processing

No special attention is needed.

4.2.4. Debug Exception

Exception Level	Exception Vector	Cause Code
Level 2	V_DEBUG DEV=0: 0x80000100 DEV=1: 0xBFC00300	Cause.EXC2 = 3

Cause of Exception

The Debug exception occurs when the breakpoint conditions are met. This exception is masked when the Status.ERL bit is set to 1, that is, when the level 2 exception handler is in process.

Operation

When the Debug exception is recognized, each of the registers is set as follows, and the control is transferred to the V_ DEBUG exception vector:

- Cause.EXC2 : 3
- Cause.BD2 : When the exception is caused in the branch delay slot, it is set to 1, otherwise, 0
- ErrorEPC Register : Restart Address (The address of the instruction causing exception. However, if Cause.BD2 is 1, it is the address of the preceding conditional branch instruction.)

Handler Processing

For breakpoints, refer to the description in "3.2. System Control Coprocessor (COP0) Registers".
A data value breakpoint becomes an inaccurate exception in load instructions due to memory latency. If ASID is changed while the control is transferred from a load instruction that has caused an exception to an exception handler, the memory mapping referred to by the exception handler differs from the memory mapping at the occurrence of the exception.

4.2.5. Interrupt Exception

Exception Level	Exception Vector	Cause Code
Level 1	V_INTERRUPT BEV=0: 0x80000200 BEV=1: 0xBFC00400	Cause.ExcCode = 0

Cause of Exception

The interrupt exception occurs if one of the three interrupt signals is asserted.
Each of the three interrupts can be masked by clearing the Status.IM[7], Status.IM[3], or Status.IM[2] bit to 0. Also, all of three interrupts can be masked at once by clearing the Status.IE or Status.EIE bit to 0.
The three interrupts are masked when the Status.EXL or Status.ERL bit is set to 1, that is, when the exception handler is in process.

Operation

When the Interrupt exception is recognized, each of the registers is set as follows, and the control is transferred to the V_ INTERRUPT exception vector:

- Cause.ExcCode : 0
- Cause.IP[7] / [3] / [2] : When the corresponding interrupt is caused, it is set to 1, otherwise, 0
- Cause.BD : When the exception is caused in the branch delay slot, it is set to 1, otherwise, 0
- EPC Register : Restart Address (The address of the instruction causing exception. However, if Cause.BD is 1, it is the address of the preceding conditional branch instruction.)

If the interrupt signal is asserted and deasserted in a very short time, the Cause.IP[7], [3], or [2] bit may not reflect the cause of an interruption correctly.

Handler Processing

Interruptions, generated by external devices, are cleared by issuing the appropriate instruction, and then removing the cause of the interruption. Note that, because of buffering, the instruction to the external device occurs after other instructions finish. If the ERET instruction is executed before the interrupt signal is deasserted, the same interrupt exception is caused again. To avoid this, execute the SYNC instruction before the ERET instruction.

4.2.6. TLB Modified Exception

Exception Level	Exception Vector	Cause Code
Level 1	V_COMMON BEV=0: 0x80000180 BEV=1: 0xBFC00380	Cause.ExcCode = 1

Cause of Exception

The TLB Modified exception occurs when the virtual address of a store operation matches a TLB entry that is Valid, not Dirty (i.e. not writable). This exception is not maskable.

Operation

When the TLB Modified exception is recognized, the values of the registers are set as follows, and the control is transferred to the V_COMMON exception vector:

- Cause.ExcCode : 1
- EPC Register : Restart Address
- Cause.BD : When the exception is caused in the branch delay slot, it is set to 1, otherwise, 0.
- BadVAddr Register : The virtual address that failed address translation.
- Context Register : The address of the page table and high-order 19 bits of the virtual address that failed address translation
- EntryHi Register : High-order 19 bits of the virtual address that failed address translation and ASID
- EntryLo Register : Undefined

Handler Processing

The kernel uses the failed virtual address or virtual page number to identify the access control information. If write operations are not permitted, a write protection violation occurs. If write accesses are permitted, the page frame is marked dirty/writable by the kernel in its own data structures.

4.2.7. TLB Refill Exception

Exception Level	Exception Vector	Cause Code
Level 1	V_TLB_REFILL * BEV=0: 0x80000000 BEV=1: 0xBFC00200	Cause.ExcCode = 2(Load) Cause.ExcCode = 3(Store)

* When it occurs in the TLB Refill exception handler, the vector is V_COMMON (0x80000180 / 0xBFC00380).

Cause of Exception

The TLB Refill exception occurs if there is no TLB entry that matches a virtual address when referring to a mapped address space. This exception is not maskable.

Operation

When the TLB Refill exception is recognized, the values of the registers are set as follows, and the control is transferred to the V_TLB_REFILL exception vector:

- Cause.ExcCode: When the exception is caused due to a load operation, it is set to 2, and when it is caused due to a store operation, it is set to 3
- EPC Register : Restart Address
- Cause.BD : When the exception is caused due to the instruction of a branch delay slot, it is set to 1, otherwise, 0
- BadVAddr Register : The virtual address that failed address translation
- Context Register : The address of the page table and high-order 19 bits of the virtual address that failed address translation
- EntryHi Register : High-order 19 bits of the virtual address that failed address translation and ASID
- EntryLo Regiseter : Undefined
- Random Register : Place that stores new TLB entry (TLB index)

When the TLB Refill exception occurs within the TLB Refill exception handler (since the Status.EXL bit has been set), the control is transferred to the V_COMMON exception vector. Note that, in this situation, the EPC register and Status.BD bit that indicate the position where the exception is caused are not modified.

Handler Processing

The handler of this exception reads physical frames and access control bits from the page table, regarding the contents of the Context register as a virtual address, and stores them in the EntryLo0 and EntryLo1 registers. Then, it writes the EntryHi and EntryLo registers in TLB.

4.2.8. TLB Invalid Exception

Exception Level	Exception Vector	Cause Code
Level 1	V_COMMON BEV=0: 0x80000180 BEV=1: 0xBFC00380	Cause.ExcCode = 2 (Instruction Fetch / Load) Cause.ExcCode = 3 (Store)

Cause of Exception

The TLB Invalid Exception occurs when the TLB entry that matches a virtual address is invalid (V=0). This exception is not maskable.

Operation

When the TLB Invalid exception is recognized, the values of the registers are set as follows, and the control is transferred to the V_COMMON exception vector:

- Cause.ExcCode: When the exception is caused due to an instruction fetch or a load operation, it is set to 2, and when it is caused due to a store operation, it is set to 3
- EPC Register : Restart Address
- Cause.BD : When the exception is caused in the branch delay slot, it is set to 1, otherwise, 0
- BadVAddr Register : The virtual address that failed address translation
- Context Register : The address of the page table and high-order 19 bits of the virtual address that failed address translation
- EntryHi Register : High-order 19 bits of the virtual address that failed address translation and ASID
- EntryLo Register : Undefined
- Random Register : Position that stores new TLB entry (TLB index)

Handler Processing

A TLB entry is, typically, marked invalid when one of the following is true, and the V bit is cleared to 0.

- A virtual address does not exist
- The virtual address exists, but not in main memory (a page fault)
- Desired trap (e.g. to maintain a reference bit)

After servicing the cause of a TLB Invalid exception, the TLB entry is probed with TLBP (TLB Probe), and replaced by an entry with its Valid bit set.

4.2.9. Address Error Exception

Exception Level	Exception Vector	Cause Code
Level 1	V_COMMON BEV=0: 0x80000180 BEV=1: 0xBFC00380	Cause.ExcCode = 4 (Instruction Fetch / Load) Cause.ExcCode = 5 (Store)

Cause of Exception

The Address Error exception occurs in one of these situations:

- Attempting to load or store a doubleword, word, or halfword data on an unaligned address
- Attempting to fetch an instruction that is not aligned on a word boundary
- Attempting to refer to the kernel address space in User or Supervisor mode
- Attempting to refer to the supervisor address space in User mode

This exception is not maskable.

Operation

When the Address Error exception is recognized, the values in each register are set as follows, and the control is transferred to the V_COMMON exception vector:

- Cause.ExcCode : When the exception is caused due to a load or the instruction fetch, it is set to 4, and when it is caused due to a store, it is set to 5
- EPC Register : Restart Address
- Cause.BD : When the exception is caused due to the instruction of a branch delay slot, it is set to 1, and otherwise, 0
- BadVAddr Register : Bad Virtual address that is the direct cause for an exception generation
- Context Register : Undefined
- EntryHi.VPN : Undefined
- EntryLo Register : Undefined

Handler Processing

No special attention is required for the exception handler.

4.2.10. Bus Error Exception

Exception Level	Exception Vector	Cause Code
Level 1	V_COMMON BEV=0: 0x80000180 BEV=1: 0xBFC00380	Cause.ExcCode = 6 (Instruction Fetch) Cause.ExcCode = 7 (Load / Store)

Cause of Exception

The Bus Error is an external exception, and is caused by events such as bus time-out, external bus parity errors, invalid physical memory addresses or invalid access types. This exception is masked when Status.EXL or Status.ERL is set to 1 (when processing an exception handler).

Operation

When the Bus Error exception is recognized, each of the registers is set as follows, and the control is transferred to the V_COMMON exception vector:

- Cause.ExcCode : When the exception is caused due to an instruction fetch operation, it is set to 6, and when it is caused due to a load/store operation, it is set to 7
- EPC Register : The instruction that the processor is executing
- Cause.BD : When the exception is caused in the branch delay slot, it is set to 1, otherwise, 0

If the Bus Error exception is caused by a load or store instruction, the instruction is retired. At that time, the value that is loaded in the registers is undefined, and how the contents of memory are updated depends on the memory subsystem's design. If a data value breakpoint is set at the address, breakpoint recognition depends on the implementation.

Handler Processing

The Bus Error exception is imprecise and has no special relation to a currently executing instruction. The Bus Error may occur due to an instruction prefetch, the operation to the data cache line that is not related to the instruction, or the load / store instruction issued several steps before. So restoring and continuing the processing is difficult.

If the Bus Error occurs when reading an instruction fetch or data cache, the value of the loaded cache line is undefined. Since it is not possible, in general, to determine the offending address, the entire data and instruction cache should be invalidated.

4.2.11. System Call Exception

Exception Level	Exception Vector	Cause Code
Level 1	V_COMMON BEV=0: 0x80000180 BEV=1: 0xBFC00380	Cause.ExcCode = 8

Cause of Exception

The System Call exception occurs when executing the SYSCALL instruction. This exception is not maskable.

Operation

When the System call exception is recognized, each of the registers is set as follows, and the control is transferred to the V_COMMON exception vector:

- Cause.ExcCode : 8
- Cause.BD : When the exception is caused in the branch delay slot, it is set to 1, otherwise, 0
- EPC Register : Restart Address (The address of the instruction causing exception. However, if Cause.BD is 1, it is the address of the preceding conditional branch instruction.)

Handler Processing

With the SYSCALL instruction, bits 25:6 of the instruction code are an optional code field. Obtain the parameter for the exception by calculating the address of the SYSCALL instruction with the Cause.BD and EPC in the handler, and reading the code field.
When resuming the execution after finishing the handler processing, the SYSCALL instruction that caused an exception should not be re-executed. For this reason, add 4 to the EPC register before returning with ERET. Note that if the SYSCALL instruction is in a branch delay slot, complicated processing (i.e. calculating the re-executing address by considering if the conditional branch can be generated or not) may be required.

4.2.12. Break Exception

Exception Level	Exception Vector	Cause Code
Level 1	V_COMMON BEV=0: 0x80000180 BEV=1: 0xBFC00380	Cause.ExcCode = 9

Cause of Exception

The Break exception occurs when executing the BREAK instruction. This exception is not maskable.

Operation

When the Break exception is recognized, each of the registers is set as follows, and the control is transferred to the V_COMMON exception vector:

- Cause.ExcCode : 9
- Cause.BD : When the exception is caused in the branch delay slot, it is set to 1, otherwise, 0
- EPC Register : Restart Address (The address of the instruction causing exception. However, if Cause.BD is 1, it is the address of the preceding conditional branch instruction.)

Handler Processing

With the BREAK instruction, bits 25:6 of the instruction code are an optional code field. Obtain the parameter for the exception by calculating the address of the BREAK instruction with the Cause.BD and EPC in the handler, and reading the code field.

When resuming the execution after finishing the handler processing, the BREAK instruction that caused an exception should not be re-executed. For this reason, add 4 to the EPC register before returning with ERET. Note that if the BREAK instruction is caused in a branch delay slot, complicated processing (i.e. calculating the re-executing address by considering if the conditional branch can be generated or not) may be required.

4.2.13. Reserved Instruction Exception

Exception Level	Exception Vector	Cause Code
Level 1	V_COMMON BEV=0: 0x80000180 BEV=1: 0xBFC00380	Cause.ExcCode = 10

Cause of Exception

The Reserved Instruction exception occurs when attempting to execute an undefined or unsupported instruction code. This exception is not maskable.

Operation

When the Reserved Instruction exception is recognized, each of the registers is set as follows, and the control is transferred to the V_ COMMON exception vector:

- Cause.ExcCode : 10
- Cause.BD : When the exception is caused in the branch delay slot, it is set to 1, otherwise, 0
- EPC Register : Restart Address (The address of the instruction causing exception. However, if Cause.BD is 1, it is the address of the preceding conditional branch instruction.)

Handler Processing

When resuming the execution after finishing the handler processing, the Reserved Instruction that caused an exception should not be re-executed. For this reason, add 4 to the EPC register before returning with ERET. Note that if the exception is caused in a branch delay slot, complicated processing (i.e. calculating the re-executing address by considering if the conditional branch can be generated or not) may be required.

Programming Notes

In other MIPS ISA implementations, attempting to execute 64-bit operations in 32-bit User or Supervisor mode may cause the Reserved Instruction exception. In the EE Core, however, 64-bit operations are always valid, regardless of the operation mode.

4.2.14. Coprocessor Unusable Exception

Exception Level	Exception Vector	Cause Code
Level 1	V_COMMON BEV=0: 0x80000180 BEV=1: 0xBFC00380	Cause.ExcCode = 11

Cause of Exception

The Coprocessor Unusable exception occurs when attempting to execute the instruction of a coprocessor whose CUx bit of the Status register is 0 (unusable). (In Kernel mode, however, the COP0 instruction can execute regardless of Status.CU0.) This instruction is not maskable.

Operation

When the Coprocessor Unusable exception is recognized, each of the registers is set as follows, and the control is transferred to the V_COMMON exception vector:

- Cause.ExcCode : 11
- Cause.CE : The number of the coprocessor causing exceptions
- Cause.BD : When the exception is caused in the branch delay slot, it is set to 1, otherwise, 0
- EPC Register : Restart Address (the address of the instruction causing exception. However, if Cause.BD is 1, it is the address of the preceding conditional branch instruction.)

Handler Processing

If the process that causes exceptions has access to the coprocessor, the process can be re-executed by setting the Status.CUx bit to 1 (usable) and restoring from the exception handler.
Even if the process has access, when the coprocessor does not exist or has failed, resuming the process is possible by emulating the interpretation of the coprocessor instruction in the handler and advancing the EPC register to the next address. Note that if the Cause.BD is 1, the conditional branch instruction should be interpreted before the processing.

4.2.15. Trap Exception

Exception Level	Exception Vector	Cause Code
Level 1	V_COMMON BEV=0: 0x80000180 BEV=1: 0xBFC00380	Cause.ExcCode = 13

Cause of Exception

The Trap exception occurs when the TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEIU, TLTI, TLTIU, TEQI, or TNEI instruction (Trap Instruction) results in a true condition. This exception is not maskable.

Operation

When the Trap exception is recognized, each of the registers is set as follows, and the control is transferred to the V_COMMON exception vector:

- Cause.ExcCode : 12
- Cause.BD : When the exception is caused in the branch delay slot, it is set to 1, otherwise, 0
- EPC Register : Restart Address (the address of the instruction causing exception. If Cause.BD is 1, however, the address of the preceding conditional branch instruction.)

Note that the trap instruction is considered complete regardless of whether or not the Trap exception occurs.

Handler Processing

With the TGE, TGEU, TLT, TLTU, TEQ, or TNE instruction, bits 15:6 of the instruction code are an optional code field. Obtain the parameter for the exception by calculating the address of the trap instruction with Cause.BD and EPC in the handler and, reading the code field. If the trap instruction has the code field or not depends on bits 31:26; 000000 or 000001.

4.2.16. Overflow Exception

Exception Level	Exception Vector	Cause Code
Level 1	V_COMMON BEV=0: 0x80000180 BEV=1: 0xBFC00380	Cause.ExcCode = 12

Cause of Exception

The Overflow exception occurs when the ADD, ADDI, SUB, DADD, DADDI, or DSUB instruction results in a 2's complement overflow. This exception is not maskable.

Operation

When the Overflow exception is recognized, each of the registers is set as follows, and the control is transferred to the V_COMMON exception vector:

- Cause.ExcCode : 12
- EPC Register : Restart Address
- Cause.BD : When the exception is caused in the branch delay slot, it is set to 1, otherwise, 0

Handler Processing

No special attention is needed.

5. Memory Management

The EE Core processor provides a memory control unit (MMU) which uses an on-chip address translation look-aside buffer (TLB) to translate virtual addresses into physical addresses.
This chapter describes the virtual and physical address spaces, the virtual-to-physical address translation, the cache mode, and the System Control Coprocessor (COP0) that provides the software interface to the TLB.

5.1. Address Space

This section describes virtual and physical spaces.

5.1.1. Physical Address Space

The EE Core has a 32-bit physical address. The physical address space corresponds to 4 GB.

5.1.2. Virtual Address Space

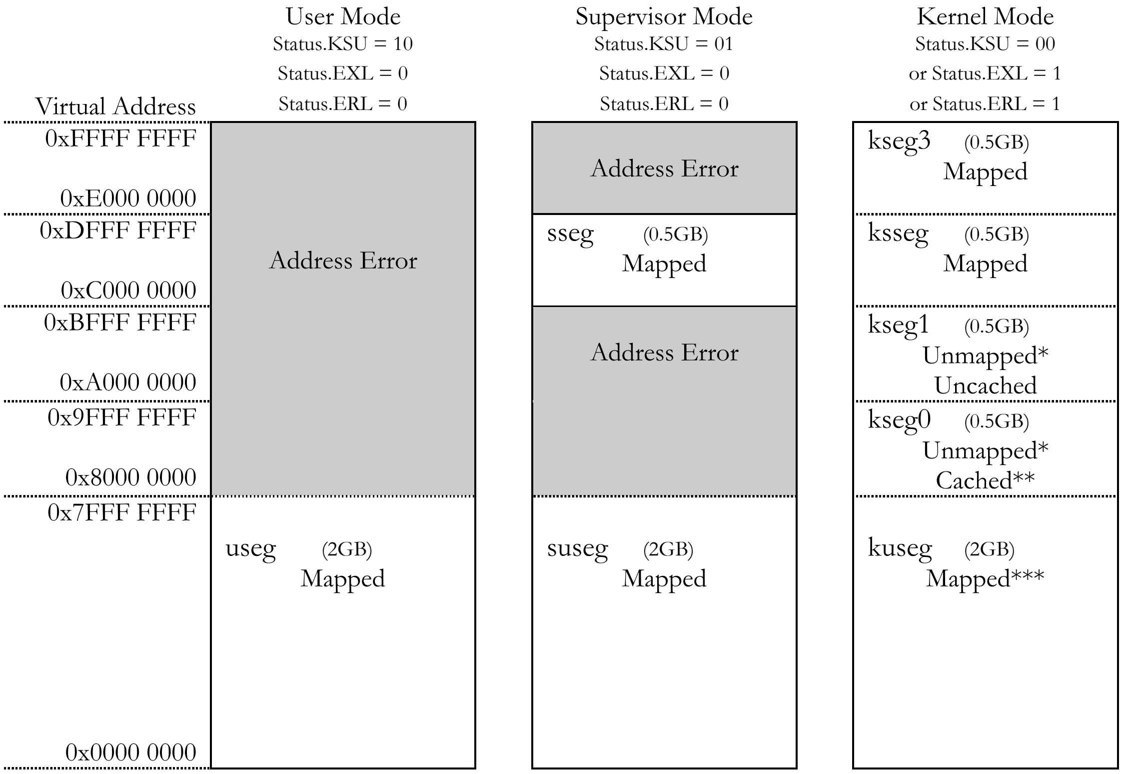
The EE Core implements only a 32-bit virtual address space. There is no requirement for address sign extension, and no checking for the address error exception will be done on the upper 32 bits of the address.

The virtual address is 32-bit, and consists of the virtual page number (VPN) and offset. The upper 3 bits of the VPN is used for identifying the operation mode. The page size is variable; either 4KB, 16KB, 64KB, 256KB, 1MB, 4MB, or 16MB. The width of the VPN and offset depend on the page size.

Also, the virtual address space has an 8-bit address space identifier (ASID), which reduces the frequency of the TLB flushing when switching contexts.

5.1.3. Operating Modes and Address Space

The EE Core has the three standard MIPS operating modes: User (user program), Supervisor (OS), and Kernel (exception handler). The address spaces of each operating mode are shown in Figure 5-1.



*Note: Virtual addresses of Kernel segments, kseg0 and kseg1, are not mapped through the TLB, and are always translated into physical addresses from 0x0000 0000 to 0x1FFF FFFF.

**Note: The kseg0 cache mode is controlled by the K0 field in the Config register.

***Note: The Kernel mode user space, or kuseg, is unmapped when Status.ERL=1 (when the level 2 exception handler is executing).

Figure 5-1 Address Space of Each Operating Mode

5.1.4. User Mode Address Space

In the User mode, a user program is executed. The processor operates in the User mode when KSU = 10, ERL =0, and EXL = 0 in the Status register.
In the User mode, a uniform virtual address space of 2 GB, useg, is available (Fig. 5-2). The most significant bit of the virtual address available in User mode is 0. Attempting to access the address whose most significant bit is 1 will cause the address error exception.

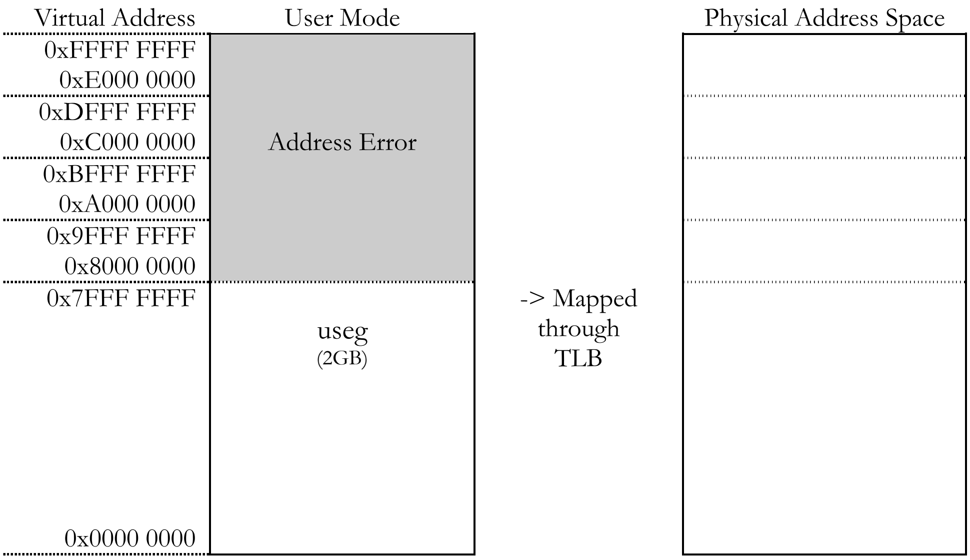


Figure 5-2 User Mode Address Space

useg (User Mode – User Space)

The user space, or useg, is allocated to the virtual address 0x00000000 to 0x7FFFFFFF, and is mapped into the physical address by the TLB. (At that time, the virtual address is extended by adding 8 bits ASID.) The cache mode is also controlled by the TLB entry.

5.1.5. Supervisor Mode Address Space

The Supervisor mode is the operation mode in which an OS routine is executed. The processor operates in the Supervisor mode when KSU = 01, ERL =0, and EXL = 0 in the Status register.
In the Supervisor mode, in addition to the address space available in User mode (0x00000000 to 0x7FFFFFFF), the address space in which the most significant 3 bits of the virtual address is 110 (0xC0000000 to 0xDFFFFFFF) is available. Attempting to access other than these addresses will cause the address error exception.

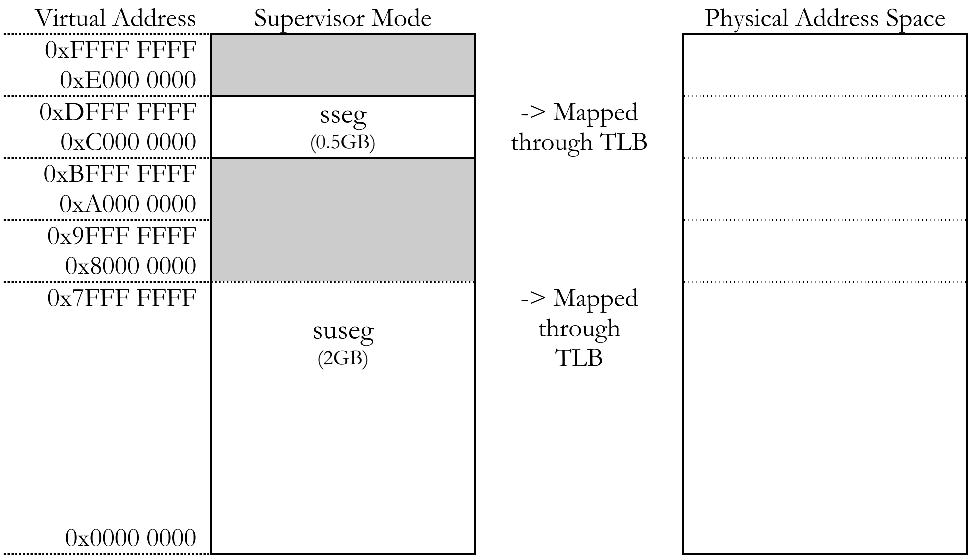


Figure 5-3 Supervisor Mode Virtual Address Space

suseg (Supervisor Mode – User Space)

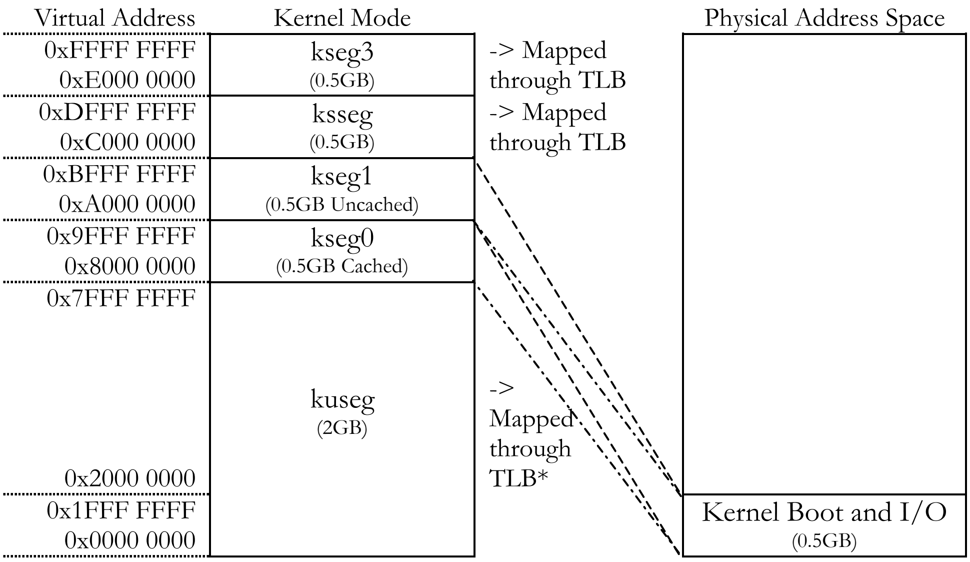
Supervisor mode user space, or suseg, is allocated to the virtual address 0x00000000 to 0x7FFFFFFF, and is mapped into the physical address by the TLB. (At that time, the virtual address is extended by adding 8 bits ASID.) The cache mode is also controlled by the TLB entry.

sseg (Supervisor Mode – Supervisor Space)

Supervisor space, or sseg, is allocated to the virtual address0xC0000000 to 0xDFFFFFFF, and is mapped into the physical address by the TLB. (At that time, the virtual address is extended by adding 8 bits ASID.) The cache mode is also controlled by the TLB entry.

5.1.6. Kernel Mode Address Space

The Kernel mode is the operation mode in which the exception handler is executed. When an exception is recognized, the processor becomes Kernel mode, and returns to the original mode by the ERET instruction. The processor operates in the Kernel mode when KSU = 00, EXL = 1, or ERL=1 in the Status register.
The virtual address space of the Kernel mode is divided into regions differentiated by the most-significant 3 bits of the virtual address, as shown in Fig. 5-4.



*Note: Kernel mode user space, or kuseg, is unmapped when Status.ERL=1 (when the level 2 exception handler is executing).

Figure 5-4 Kernel Mode Virtual Address Space

kuseg (Kernel Mode – User Space)

Kernel mode user space, or kuseg, is allocated to the virtual address 0x00000000 to 0x7FFFFFFF. When Status.ERL=0, (when the level 1 exception handler is executing), as in the cases of User and Supervisor mode, the virtual address is mapped into the physical address by adding 8 bits ASID by the TLB.
When Status.ERL = 1 (when the level 2 exception handler is executing), kuseg becomes unmapped and uncached, and mapped to the physical address 0x00000000 to 0x7FFFFFFF directly.

kseg0 (Kernel Mode – Kernel Space 0)

In Kernel mode, the space whose most significant 3 bits of the virtual address is 100 (0x80000000 - 0x9FFFFFFF) is the Kernel space, kseg0.
kseg0 is not affected by the address translation by the TLB, and is mapped to the physical address, defined by subtracting 0x80000000 from the virtual address, that is 0x00000000 to 0x1FFFFFFF directly. The cache mode is controlled by the k0 field of the Config register.

kseg1 (Kernel Mode – Kernel Space 1)

In Kernel mode, the space whose most significant 3 bits of the virtual address is 101 (0xA0000000 to 0xBFFFFFFF) is the Kernel space, kseg1.
kseg1 is not affected by the address translation by the TLB, and is mapped to the physical address, defined by subtracting 0xA0000000 from the virtual address, that is 0x00000000 to 0x1FFFFFFF directly. Caches are disabled, and the physical memory (or memory-mapped I/O device register) is accessed directly.

ksseg (Kernel Mode – Supervisor Mode)

In Kernel mode, the space whose most significant 3 bits of the virtual address is 110 (0xC0000000 to 0xDFFFFFFF) is the Supervisor space, ksseg. As in the case of Supervisor mode, the virtual address is extended by adding 8 bits ASID, and is mapped into the physical address by the TLB.

kseg3 (Kernel Mode – Kernel Space 3)

In Kernel mode, the space whose most significant 3 bits of the virtual address is 111 (0xE0000000 to 0xFFFFFFFF) is the Kernel space, kseg3. The virtual address is extended by adding 8 bits ASID, and is mapped into the physical address by the TLB. The cache mode is also controlled by the TLB.

Note that even if the program counter wraps around from kseg3 to kuseg, no address error exception occurs.

5.2. Address Translation

A virtual address to physical address translation is done by calculating the physical frame number (PFN) that corresponds to a virtual page number (VPN) from the corresponding table (page table). The page table is placed in memory, and is controlled by the OS. In order to accelerate the address translation, the address translation look-aside buffer (TLB) is provided. The following sections describe the address translation that uses the TLB.

5.2.1. Overview of Address Translation

The virtual address space is divided into particular size pages and is mapped into the physical address space in pages (Fig 5-5). The address translation process calculates the page number of the physical address space from the page number of the virtual address space using the page table.

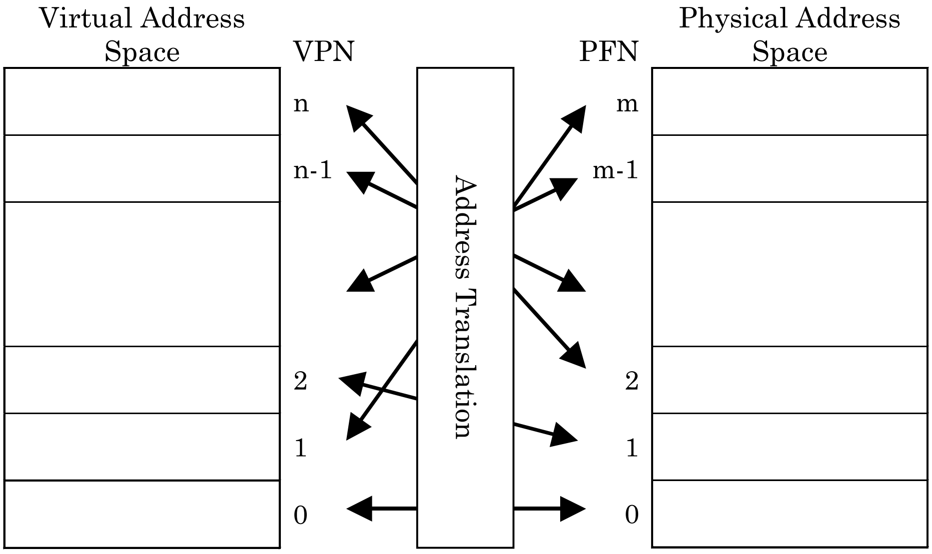


Figure 5-5 Mapping in Page Units

Page sizes can be chosen from 4KB, 16KB, 64KB, 256KB, 1MB, 4MB, or 16MB. The upper 20 to 8 bits of the virtual address become the virtual page number (VPN), depending on these sizes.
Fig. 5-6 and Fig 5-7 illustrate the corresponding conception figures between virtual and physical addresses, when the page size is 4KB, or the VPN is 20-bit width, and when the page size is 16MB, or the VPN is 8-bit width, respectively.

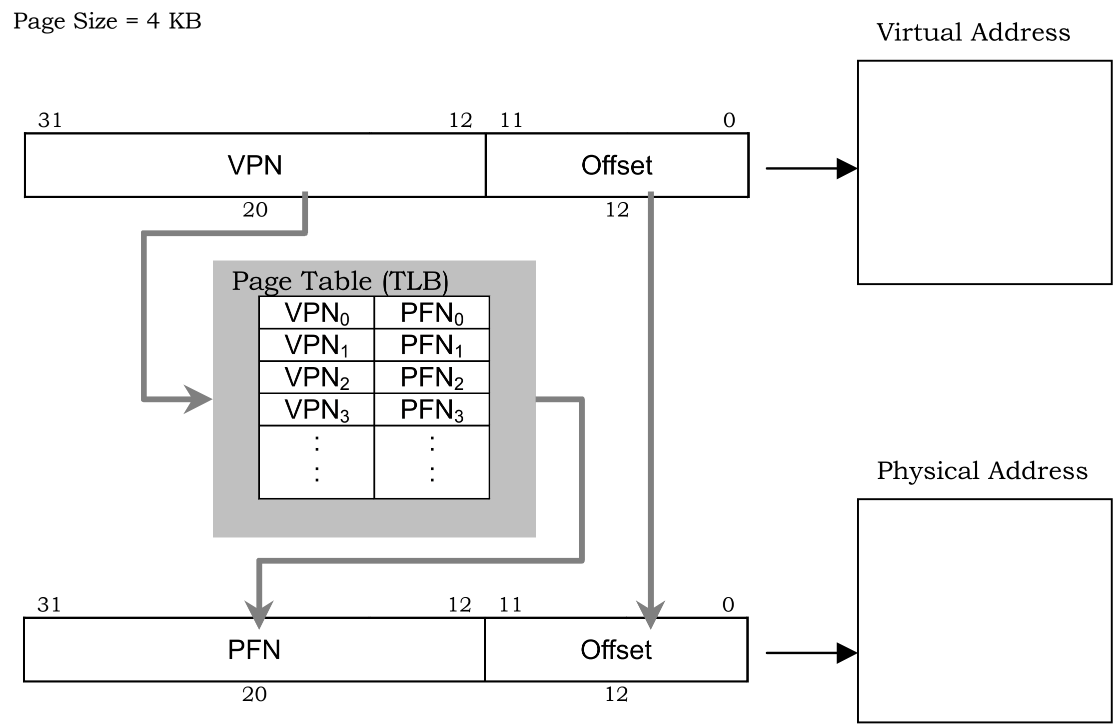


Figure 5-6 Address Translation with 4 KB Pages

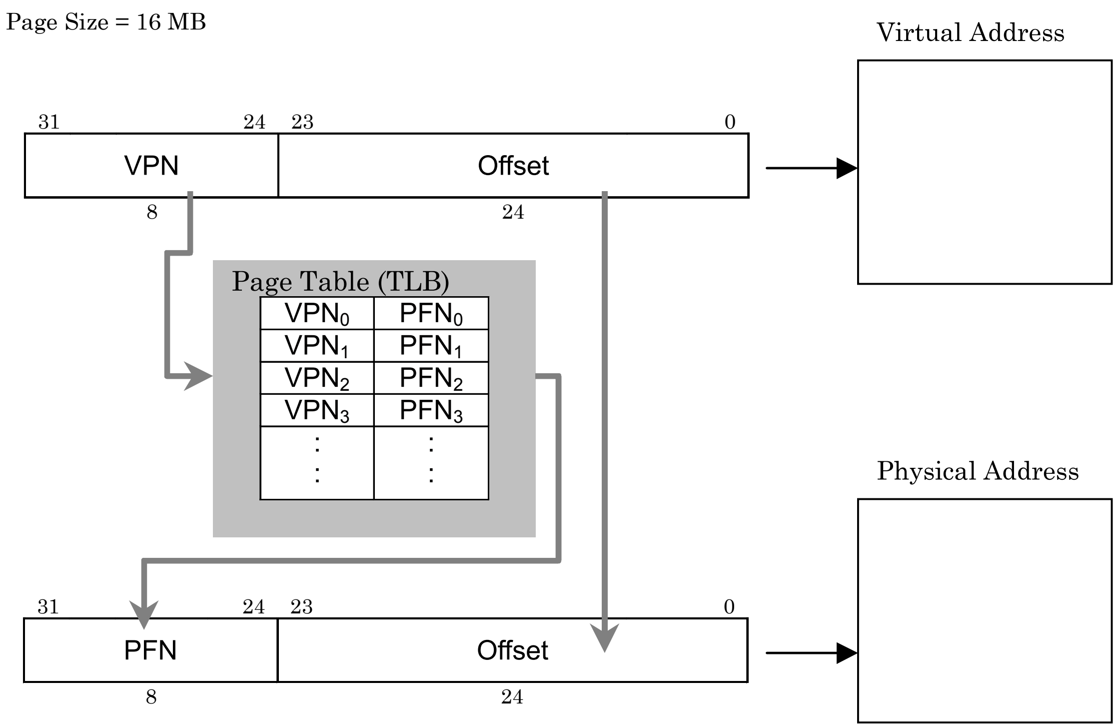


Figure 5-7 Address Translation with 16 MB Pages

5.2.2. Address Translation Look-aside Buffer (TLB)

Since the page table that indicates the relation between the virtual and physical addresses must be read and written at high speed, an on-chip address translation look-aside buffer (TLB) is provided. The TLB is a 48-entry full associative memory. It holds the access limitation flag of each page, cache mode, page size information, ASID, as well as VPN-PFN corresponding information 2 pages per 1 entry. (The contents of a TLB entry are described in detail later.)

During the address translation, the TLB takes the VPN from the virtual address that is to be accessed, and the TLB entry that has the VPN2 field that matches the VPN is searched. If the matched TLB entry is searched (TLB hit), it takes the PFN from the TLB entry, and a physical address is obtained by combining it with the lower Offset of the virtual address. (ASID and the G bit are related to judge the TLB hit. Details are described later.)

Since the number of the TLB entry is limited, holding information about the entire virtual address is not possible. If there is no TLB entry that matches the VPN, a TLB miss occurs and a TLB Refill exception is generated. Then the software (OS) reads the proper information from the page table in memory into the TLB. The corresponding information newly read by the TLB miss can be written in the TLB entry that is selected by the hardware at random or the TLB entry specified in software.

Fixed Entry

A part of the TLB entry can be fixed to protect it from rewriting when newly corresponding information is read due to a TLB miss. Refer to the description of the Wired register and the TLBWR instruction for more details.

TLB Invalid

The V bit of the TLB entry indicates validity of the entry. When the V bit of the TLB entry that matches the given virtual address is 0, a TLB Invalid exception occurs.

Multiple Match Protection

The EE Core does not support multiple TLB entry matches to a given virtual address. In this case, the operation is undefined. The software is expected never to allow the multiple matches to occur.

G bit and ASID

When the G bit of the TLB entry is set to 1, the TLB hit or miss is judged only by the VPN match for the TLB entry. When the G bit is 0, the TLB hit does not occur unless, in addition to the VPN match, ASID corresponds to the ASID field of the EntryHi register. Therefore, if allocating different ASIDs (EntryHi.ASID holds different values) during each process (by providing proper page tables), different physical memory regions are available for every process in the same virtual address.

5.2.3. Address Translation Process Flow

Fig. 5-8 illustrates the flow chart for the address translation process, which obtains the physical address from the virtual address.
The address translation process is largely divided into four stages:

- At the first stage, according to the processor's operating mode, the logic judges if the input virtual address is valid or not. If it is not valid, the address error exception occurs. Since the address that corresponds to kseg0 and kseg1 in Kernel mode is unmapped, the physical address is obtained at this stage.
- At the second stage, the logic searches the VPN, the upper 8 to 20 bits (depending on the page size) of the virtual address, and the TLB entry that matches the ASID field of the EntryHi register (only when the Global bit is 1). If there is no TLB entry to match, the TLB Refill exception is generated.
- At the third stage, the logic examines the control bit of the matched TLB entry. If the V bit is 0, the TLB Invalid exception occurs. If the D bit is 0 and there is a write access, the TLB Modified exception occurs.
- At the final stage, the access target is determined according to the contents of the S bit and the C field. The physical address is obtained from the physical page number (PFN) taken from the TLB entry and Offset in the lower 24 to 12 bits of the virtual address.

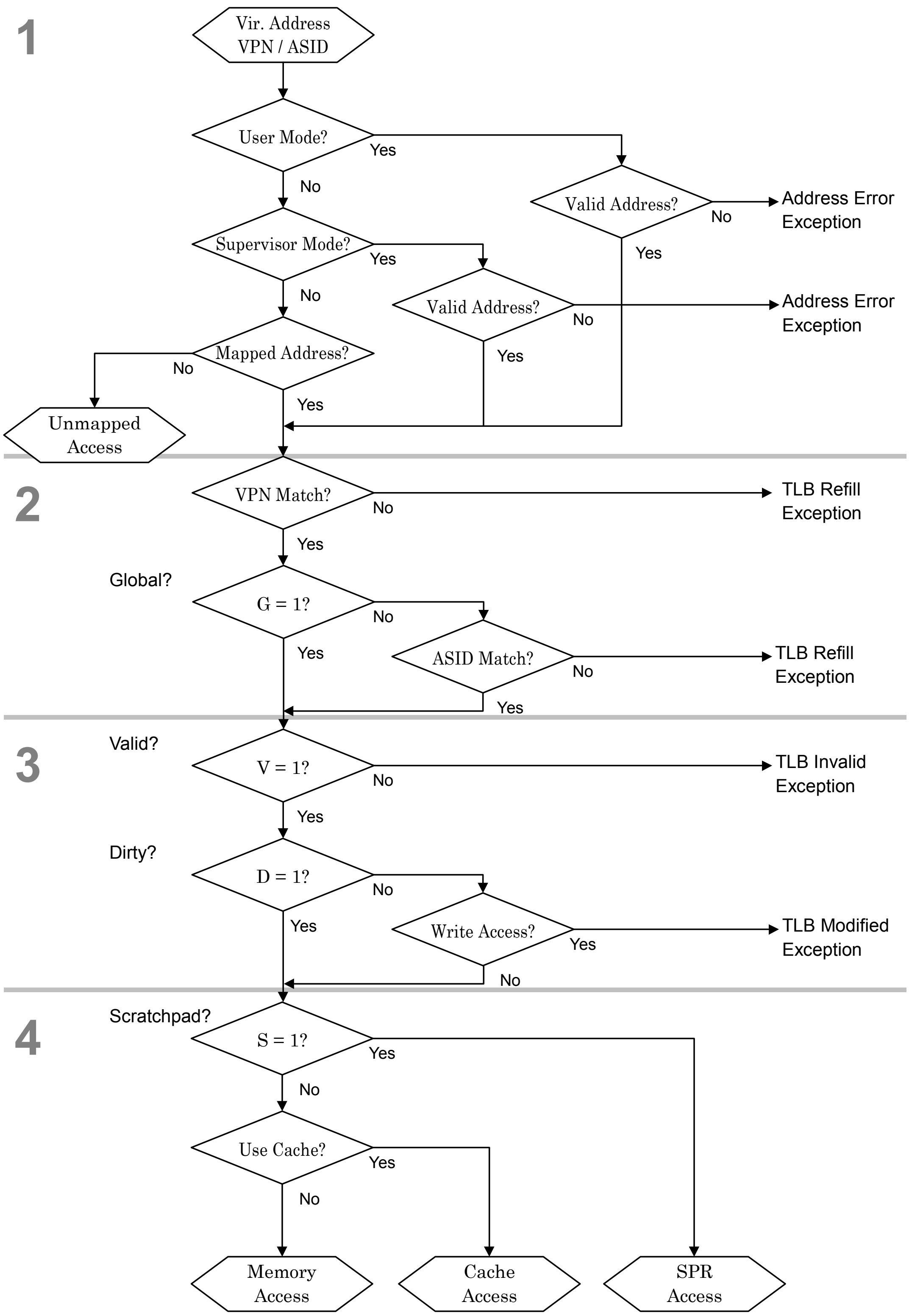


Figure 5-8 Address Translation Process Flow

5.2.4. TLB Entry

The TLB entry is a 128-bit data, and has the structure illustrated in Fig. 5-9. The contents of each field are shown in the table below.

127–121	120–109	108–96
0	MASK	0
7	12	13

95–77	76	75–72	71–64
VPN2	G	0	ASID
19	1	4	8

63	62–58	57–38	37–35	34	33	32
S	0	PFN	C	D	V	0
1	5	20	3	1	1	1

31–26	25–6	5–3	2	1	0
0	PFN	C	D	V	0
6	20	3	1	1	1

Figure 5-9 Structure of a TLB Entry

Name	Pos.	Contents
V	1	Valid 0 Mapping even pages is invalid. 1 Mapping even pages is valid.
D	2	Dirty 0 Disables a write to even pages. 1 Enables a write to even pages.
C	5:3	Cache modes in even pages 2 Uncached 3 Cached with write-back and write-allocate 7 Uncached accelerated (Other values are reserved)
PFN	25:6	Page Frame Number Page frame number of even pages.
V	33	Valid 0 Mapping odd pages is invalid 1 Mapping odd pages is valid
D	34	Dirty 0 Disables a write to odd pages 1 Enables a write to odd pages
C	37:35	Cache modes in odd pages 2 Uncached 3 Cached with write-back and write-allocate 7 Uncached accelerated (Other values are reserved)
PFN	57:38	Page Frame Number Page frame number of odd pages.
S	63	Scratchpad RAM 0 Main memory 1 Scratchpad RAM
ASID	71:64	Address Space ID Address space identifier.
G	76	Global 0 Includes ASID for judgement condition of the TLB hit. 1 Ignores ASID.
VPN2	95:77	Virtual Page Number / 2 The value from dividing the virtual number by 2. (If concatenated with 0, it becomes an even page, and if concatenated with 1, it becomes the virtual page number corresponding to odd pages.)
MASK	120:109	Page Comparison Mask The mask that indicates valid parts in VPN2 (indicates the page size).

5.2.5. Scratchpad RAM Mapping

The scratchpad RAM (SPRAM) has a special restriction for allocating the virtual address, because it is physically different from normal memories. It must be mapped into a contiguous 16 KB of virtual address space that is aligned on a 16KB boundary. Results are not guaranteed if this restriction is not followed.

The TLB entry corresponding to SPRAM is indicated by setting the S bit to 1. The MASK field must be all zeros, and two D bits and two V bits must be the same value, respectively. The PFN and C field values are disregarded. (When read, the C field value is 2, which indicates uncached mode.)

The virtual address region 16KB that is to be a pair of even and odd pages to the virtual address allocated in SPRAM will not be able to map as a 16KB page, since a multiple match of the TLB is not allowed. To use the virtual address or keep it unused, it must be mapped as 4KB by 4 pages. (It can also be allocated to SPRAM using another TLB entry.)

Fig. 5-10 illustrates the allocation of SPRAM to 16KB from the virtual address 0x00010000, and the mapping of the corresponding odd pages (16KB from 0x00014000) as a 4KB page.

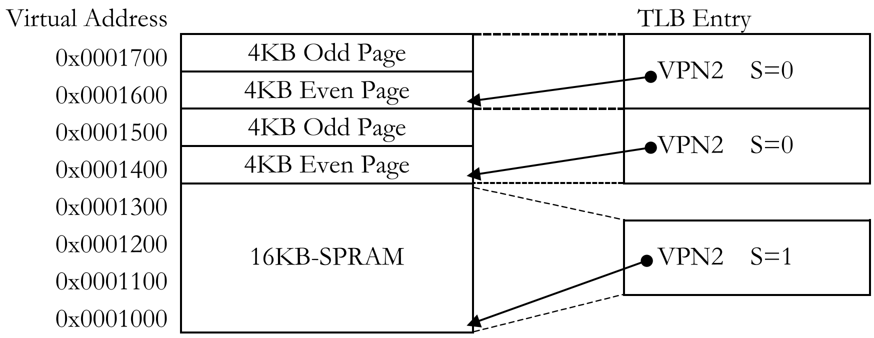


Figure 5-10 Example of SPRAM Mapping

5.2.6. Cache Mode

One of the following cache modes can be specified for each page by the C field of the TLB entry.

C Field Setting Value	Cache Mode
2	Uncached
3	Cached with write-back and write-allocate
7	Uncached accelerated

In the cached mode, when a cache miss occurs in a store operation, the missed data is read from memory to a cache line. Then the data is stored, to perform a write allocate.
In the uncached accelerated mode, a special buffer (UCAB) is used. When loading data, the EE Core fetches 128 bytes (8 quadwords) from memory at one time and places them in UCAB. In the following load operation, taking as much data from the UCAB as possible can reduce the bus traffic. For details of the UCAB, see "6.5. Uncached Accelerated Buffer (UCAB)".
The store operation in the uncached accelerated mode uses a write-back buffer (WBB) of 8 qwords (128 bits x 8 entries). Data is stacked up in the same WBB entry to reduce bus traffic, as long as the destination memory address is written in a store operation and the next store operation is performed within the same qword boundary. Data is stacked up until any of the following occurs:

- Attempting to store data having a different attribute or an address
- Executing a load operation
- Executing the SYNC or SYNC.L instruction
- An exception

5.3. System Control Coprocessor

The System Control Coprocessor (COP0) is the unit that supports privileged operations such as memory management, address translation, cache control, and exception handling. TLB related registers and TLB operating instructions of the COP0 are described in this section.

5.3.1. TLB Related Register

The registers directly related to the TLB entry, PageMask, EntryHi, EntryLo0, and EntryLo1, are provided. Except for the G Bit position, the contents of these four registers correspond to the TLB entry. The registers for controlling the use of each TLB entry, Index, Random, and Wired are provided. Also, the registers related to the TLB exception, Context and BadVAddr are provided.

Register Name	Register Number	Contents	
PageMask	5	Page Comparison Mask (Page Size)	
EntryHi	10	VPN2 ASID	Virtual page number divided by two Address Space Identifier
EntryLo0	2	S PFN C D G	Scratchpad RAM flag Page frame number of odd pages Cache mode of odd pages Write permission flag of add pages Global (disregard of ASID) flag
EntryLo1	3	PFN C D G	Page frame number of even pages Cache mode of even pages Write permission flag of even pages Global (disregard of ASID) flag
Index	0	Index	TLB entry index
Random	1	Random	TLB entry index (Updated automatically)
Wired	6	Wired	The number of wired TLB entries
Context	4	PTEBase BadVPN2	Page table address Virtual page number of an exception cause
BadVAddr	8	Virtual address of an exception cause	

5.3.2. TLB Operation Instructions

The following instructions are provided in order to operate the TLB.

Mnemonic	Function
TLBR	Translation Look-aside Buffer Read Reads the contents of the TLB entry that is indicated by the Index register, and stores them in PageMask, EntryHi, EntryLo0, and EntryLo1 registers.
TLBWI	Translation Look-aside Buffer Write Index Stores the contents of PageMask, EntryHi, EntryLo0, and EntryLo1 registers in the TLB entry that the Index register indicates.
TLBWR	Translation Look-aside Buffer Write Random Stores the contents of PageMask, EntryHi, EntryLo0, and EntryLo1 registers in the TLB entry that the Random register indicates. (The Random register is updated automatically.)

(This page is left blank intentionally)

6. Caches

The EE Core contains both an instruction cache and a separate data cache. The processor also contains an embedded scratchpad RAM (SPRAM) for fast manipulation of large data structures and an embedded Uncached Accelerated Buffer (UCAB), which operates as a read buffer for the uncached accelerated space.

This chapter describes the cache structures, operations and control.

6.1. Cache and SPRAM Features

The two caches of the EE Core are configured as shown in Table 6-1:

Cache	Size	Organization	Line Size	Refill Size
Instruction Cache	16 KB	2-Way	64 bytes	64 bytes
Data Cache	8 KB	2-Way	64 bytes	64 bytes

Table 6-1 Cache Configuration

The following are the main features of the caches:

- Separate Instruction Cache and Data Cache
- Virtually indexed and physically tagged caches
- 64-byte line size
- 64-byte refill size
- 2-way set-associative cache
- Write-back policy (Data Cache)
- Missed quadword first sequential order refills (Data Cache)
- Line Locking (Data Cache)
- Non-Blocking Loads
- Supports hits under miss (Data Cache)
- No bus snooping (CACHE instructions are used to keep coherency with memory)

The following are the features of the Scratchpad RAM (SPRAM):

- 16 KB static RAM organized as 1K x 128 bits
- External DMA read and write capability
- Accessible to software through load/store instructions
- Can be mapped to the virtual address space via software

6.2. Organization of the Caches

Organization of the Instruction and Data Caches is described below. Both are 2-way set-associative, composed of two sets of tags and 64-byte data.

Cache	Size	Organization	Tag Index
Data	8KB	64 bytes x 64 entries x 2-way	Bits 11: 6 in the virtual address
Instruction	16KB	64 bytes x 128 entries x 2-way	Bits 12: 6 in the virtual address

Table 6-2 Cache Size and Address Bits

While the caches are indexed by the virtual address, a hit or miss is determined in a physical address. This is possible because the caches and the TLB are accessed in parallel. When the tags have been accessed, the translated physical address is compared with the frame number and a cache hit or miss is determined.

6.2.1. Organization of the Data Cache

The Data Cache is connected to the CPU via a 128-bit bus. Therefore, the Data Cache can supply to the CPU or the coprocessors up to a quadword of data per access.
Organization of the Data Cache is illustrated in Figure 6-1. Tags are discussed in detail in a later section.

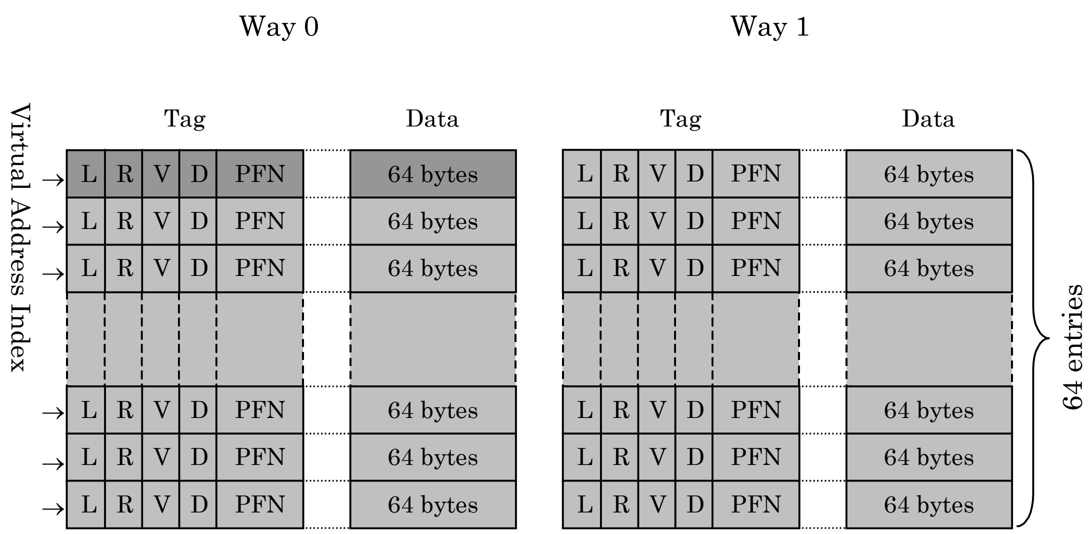


Figure 6-1 Organization of Data Cache

6.2.2. Organization of the Instruction Cache

The Instruction Cache is connected to the CPU pipeline via a 64-bit bus. This enables the CPU to fetch two instructions per cycle. Organization of the Instruction Cache is shown in Figure 6-2. Tags are discussed in detail in a later section.

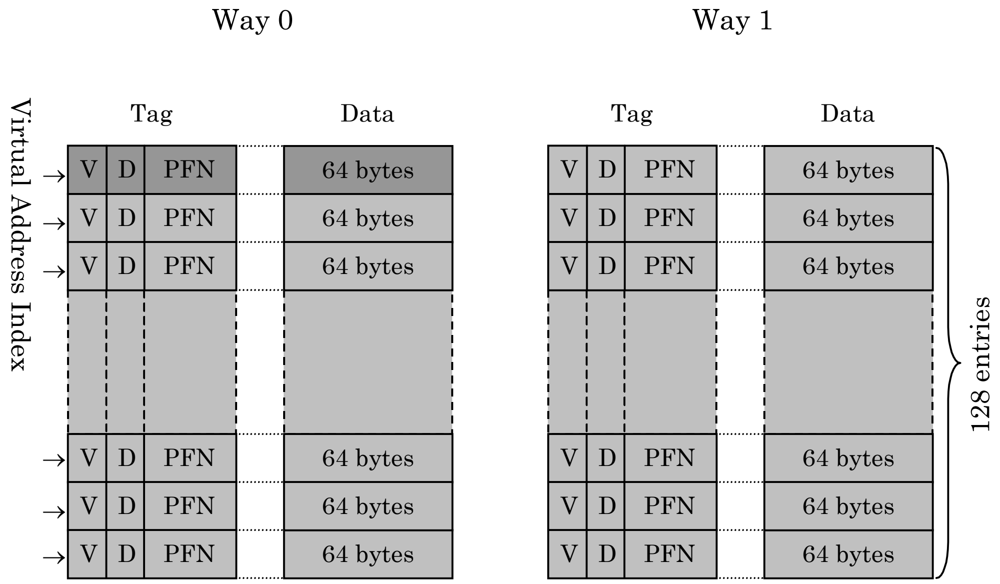


Figure 6-2 Organization of Instruction Cache

6.2.3. Tag Structure

The cache tag consists of a set of state bits and a physical page frame number or PFN field. The data and instruction caches have different numbers of state bits.

Data Cache Tag Structure

Each Data Cache tag entry, as shown below, contains four state bits in addition to the physical page frame number (PFN).

Dirty (D)	Valid (V)	LRF (R)	Lock (L)	PFN

Figure 6-3 Data Cache Tag Fields

The Dirty bit and the Valid bit together identify the three states of the Data Cache Line (Valid Clean, Valid Dirty or Invalid).

Dirty (D)	Valid (V)	Cache Line State
0	0	Invalid
0	1	Valid Clean
1	1	Valid Dirty

* The combination of D=1 and V=0 does not occur. When the V bit is set to 0, the D bit is also set to 0.

Table 6-3 Data Cache Line States

The LRF bits indicate the Least-Recently-Filled line and control a replacement between the two ways on the same line. A refill access to a cache line in a way will flip the LRF bit to point to the other way as the least recently filled. For details of the LRF line update operation, refer to Section "6.3.1. Line Replacement Algorithm".

The lock bit is a flag which locks lines to keep data from being replaced. Refer to "6.3.7. Data Cache Lock Function" for more details.

Instruction Cache Tag Structure

Each Instruction Cache tag entry, as shown below, contains two state bits in addition to the physical page frame number (PFN).

Valid (V)	LRF (R)	PFN

Figure 6-4 Instruction Cache Tag Fields

The Valid bit indicates if each line is in the Valid or Invalid states. The LRF bits, like those of the Data Cache tag, indicate the Least-Recently-Filled line and control a replacement. Refer to Section "6.3.1. Line Replacement Algorithm" for more information.

Initial Value of Cache Tags

Status bits of all Data and Instruction Cache tags are initialized to 0 and the values of PFN fields are undefined upon reset.

6.3. Cache Operations

This section describes cache operations in regard to read/write policies, coherency, write-back and the lock function.

6.3.1. Line Replacement Algorithm

Based on the LRF algorithm for line replacement of the Instruction Cache and Data Cache, one of the 2 ways that was least recently refilled is replaced. For example, when a read from memory to each line occurs, the LRF bit is flipped. (Load and store accesses to the Data Cache do not modify the LRF bit.) XOR of the LRF bits indicate which way is the least recently filled and that result determines which way could be refilled. Refer to the following table.

Way0 LRF	Way1 LRF	XOR		Refill Way	New Way0 LRF	New Way1 LRF
0	0	0	->	0	1	0
1	0	1		1	1	1
1	1	0		0	0	1
0	1	1		1	0	0

Table 6-4 LRF Line Replacement Algorithm

If the cache line is locked, regardless of the state of the LRF bits, the data is refilled into the unlocked way. And when one of the ways is Invalid in the Instruction Cache, regardless of the state of the LRF bits, the data is refilled into the Invalidated way.

6.3.2. Non-blocking Loads and Hit Under Miss

The Data Cache supports non-blocking load and hit under miss to improve performance. Support for a non-blocking load allows the pipeline to continue instruction execution until one of the following occurs even when load instructions are pending due to a cache miss or uncached loads are in the process of execution:

- An instruction which has data dependency with a load instruction that is pending (except for a load, store, or prefetch instruction) is issued
- Pipe 0 stalls

Loads to GPRs are non-blocking and loads to COP1 and COP2 are always blocking.
Support for hit under miss enables access to the Data Cache and continued execution of instructions in the pipeline, even when load, store, or prefetch instructions are pending due to a cache miss.
Uncached loads also do not stall the pipeline. The pipeline continues instruction execution until one of the following occurs:

- A load, store, or prefetch instruction which has data dependency with the preceeding uncached load is issued
- A Data Cache miss occurs
- An uncached accelerated buffer miss occurs
- An uncached load instruction is issued

Support for a blocking load and hit under miss allows the pipeline to continue instruction execution until one of the following occurs, even when memory is accessed due to a data cache miss or uncached loads:

- An instruction that has data dependency with the load instruction in the process of accessing memory is issued
- A second Data Cache miss occurs or an uncached accelerated buffer miss occurs
- An uncached load instruction is issued
- Pipe 0 stalls

6.3.3. Cache Hit and Miss Operations

With a Data Cache hit, the cache sends data to the CPU in 128-bit (1-qword) units. In case of an Instruction Cache hit, the cache sends data (instruction code) in 64-bit units. To read or write data less than 128 bits is specified by the least significant 4 bits (bits 3:0) of the address.
With cache misses, cache refill is performed in one cache line (64-byte = 4-qword). Since the caches are connected to the system bus via a 128-bit bus, cache refill takes a burst of 4 bus cycles (8 CPU cycles). (Actual transfer time can be more due to bus arbitration, etc).
With a cache refill, both the Instruction and Data Cache always fetch first a missed quadword of a burst of four quadwords. The sequence in which four quadwords are read depending on the least significant 2 bits of the missed quadword is shown in the table below. Figure 6-5 illustrates the sequence in which the Data Cache is read from the memory when the second quadword misses.

Missed Address (PA[5:4])	Read Order (PA[5:4])
00	00 → 01 → 10 → 11
01	01 → 10 → 11 → 00
10	10 → 11 → 00 → 01
11	11 → 00 → 01 → 10

Table 6-5 Reread Order in Case of Cache Miss

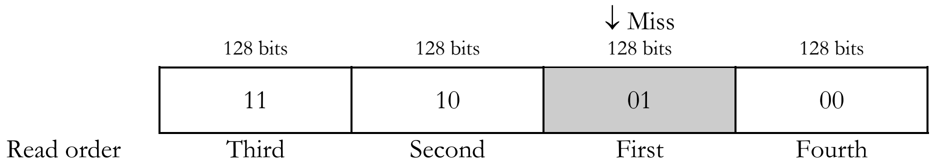


Figure 6-5 Reread Processing in Case of Cache Miss

With a write miss to the Data Cache, the data is read from main memory in sequential order.
With cache misses in the Instruction Cache, just like with the Data Cache, a reread is performed in 4 quadwords and the pipeline starts in the same cycle the final qword is stored into the Instruction Cache.

6.3.4. Data Cache Writeback

Data cache lines are written back to memory before the missed data are read when the line data are dirty and a read or write miss occurs.
In addition, when the CACHE DXWBIN instruction has been executed to a dirty cache line or the CACHE DHWBIN or CACHE DHWOIN instruction hits on a dirty cache line, the cache line is written back.

6.3.5. Data Cache State Transitions

As discussed previously, lines in the Data Cache can be in one of several states: Invalid, Valid Clean or Valid Dirty.
Invalid means the Data Cache line does not contain valid data. When a miss occurs, the data can be read in to the line immediately.
Valid Clean indicates that there is valid data in the Data Cache line and it is the same data as in memory.
Valid Dirty indicates that there is valid data in the Data Cache line and it is not the same data as in memory. That is, the data written into the cache has not been reflected yet in memory. The line has to be written back before reading the data.
The transition of the Data Cache is shown in Figure 6-6.

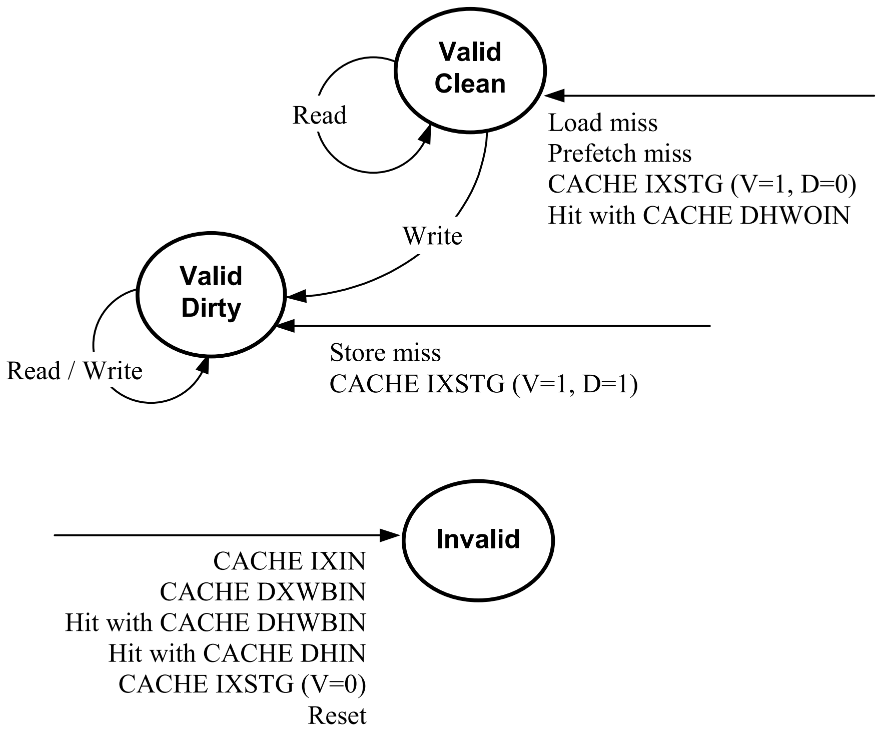


Figure 6-6 Data Cache Transition Diagram

6.3.6. Instruction Cache State Transitions

Cache lines in the Instruction Cache can be in either of two states: Invalid or Valid.
Invalid means the Instruction Cache line does not contain valid instructions. When a miss occurs, the line can read the instruction immediately.
Valid state indicates that there are valid instructions in the cache line.
The transition of the Data Cache is shown in Figure 6-7.

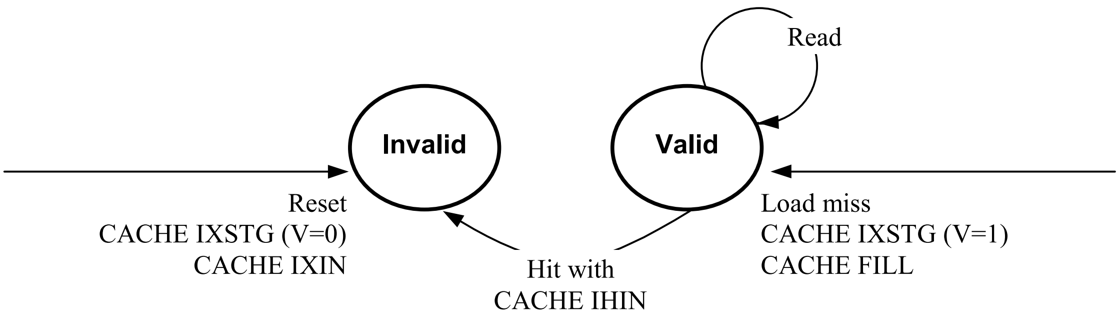


Figure 6-7 Instruction Cache Transition Diagram

6.3.7. Data Cache Lock Function

The contents of the Data Cache are replaced dynamically using the LRF algorithm. However, a Data Cache lock function has been provided to retain important data in the cache.
When the Lock bit of the data cache tag is set, the LRF bit on the line is no longer meaningful and the other way is the only way available for cache miss processing (this cache miss is blocking). A write access to a locked line is performed only to the cache, not to memory. Also, the Dirty bit is not set. To lock the Data Cache, the following two instructions can be used:

CACHE DXSTG(DCACHE Index Store Tag)
CACHE DXSDT(DCACHE Index Store Data)

The sample code for locking the data cache is as follows:

```
li t0,0x00010068     // PTagLo = 0x00010, D=V=L=1, R=0
mtc0    t0,$28              // Transfers t0 to the TagLo register
sync.l
cache   dxstg,0(r0)         // Sets (locks) the cache tag via the TagLo register
sync.l
la s0,0x00010000
sw      t1,0(s0)   // Writes to the locked line
```

The sample code for unlocking the data chache is as follows:

```
li t0,0x00010060     // D=V=1, L=R=0
mtc0    t0,$28              // Transfers t0 to the TagLo register
sync.l
cache   dxstg,0(r0)         // Sets (unlocks) the cache tag via the TagLo register
sync.l
```

The following restrictions apply to line locking:

- The result of re-locking a locked line is undefined
- The results of locking both ways of a cache line are undefined

6.3.8. Relationship between Cached and Uncached Operations

Uncached or Uncached Accelerated load and store operations are always executed in order on the CPU bus. On the other hand, cached load operations can precede earlier stored data on the CPU bus while the data is kept in the buffer. In order to avoid this, wait until the stored data has been sent to the Data Cache, SPRAM or CPU bus, using the SYNC or SYNC.L instructions.
Uncached or Uncached Accelerated store operations bypass the Data Cache completely.

6.3.9. Data Consistency between Cache and SPRAM

The capability to retain data consistency between address spaces, which are mapped into the instruction cache and the data cache, and SPRAM is not provided.

6.4. Scratchpad RAM (SPRAM)

Certain applications require high-speed on-chip RAM that can be accessed by normal load and store instructions to handle data structures efficiently. To achieve this capability, a Scratchpad RAM (SPRAM) of 16 KB is provided, in addition to the locked Data Cache capability. The DMA controller, as well as the CPU, can access SPRAM; the DMA controller has access priority.

6.4.1. SPRAM Overview

The SPRAM is similar to a tag-free Data Cache configured as 1024 x 128 bits. The SPRAM and the Data Cache use the same access paths, which means the CPU can access only either SPRAM or the Data Cache in any given CPU cycle. The SPRAM can be mapped into the virtual address.

SPRAM space pages are 16 KB in size. The least significant 14 bits of the virtual address indicate addresses in SPRAM. The upper 18 bits of the virtual address are used to access the TLB to determine if that particular 16 KB block is mapped into SPRAM or not. To differentiate between the memory spaces (between the Data Cache and SPRAM), the S bit in the TLB entry is used.

6.4.2. DMA Access to SPRAM

To read data from and write data to the SPRAM, a special DMA protocol is provided, in addition to load/store instructions. The DMA transfer between SPRAM and memory is performed as shown in Figure 6-8.

For DMA writes to SPRAM, a special SPRAM write signal is provided to the CPU along with a 10-bit SPRAM address (bits 13: 4). Data is placed on the CPU Bus. The CPU samples the data from the CPU Bus and writes it into the indexed address in the SPRAM.

For DMA reads from SPRAM, the external DMA controller reads the contents of the SPRAM into memory. The CPU Bus address is used to index the SPRAM and the corresponding data is placed on the CPU Bus. Thus, by reading / writing with DMA, the CPU can execute programs concurrently, which can result in higher performance.

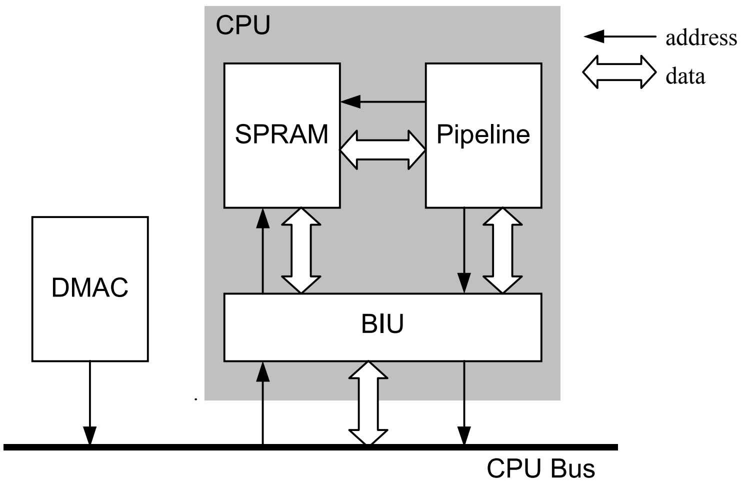


Figure 6-8 SPRAM Data Paths

Figure 6-8 illustrates how the SPRAM is embedded in the CPU and accessed by the DMA. Simultaneous access to the SPRAM by the CPU and the DMA controller results in alternate cycle accesses, with the DMA controller having the highest priority.

6.4.3. SPRAM Mapping

The SPRAM can be mapped into the virtual address space as one 16-KB page. The sample code for setting TLB is as follows:

```
li t0,    Lo0      // Lo0=0x80000018 or (D<<2)31..0 or (V<<1)31..0
MTC0      t0,      $2      // EntryLo0($2)=Lo0
li t1,    Lo1      // Lo1=0x00000018 or (D<<2)31..0 or (V<<1)31..0
MTC0      t1,      $3      // EntryLo0($3)=Lo1
MTC0      $0,      $5      // PageMask($5)=0x00000000
li t2,    VA_ASID          // VA_ASID=(VA>>1)31..13 || 05 || (ASID)7..0
MTC0      t2,      $10     // EntryHi($10)=VA_ASID
TLBWI                      // Can be replaced with TLBWR
SYNC.P
```

One of the following three values is given to EntryLo0/EntryLo1. In the SPRAM, the D and V bits of EntryLo0 must be the same as those of EntryLo1. In addition, setting V to 0 and D to 1 is meaningless.

- Invalid entry (A TLB Invalid exception occurs to the first access.)

  EntryLo0($2)==0x8000_0018 (S=1, D=0, V=0)

  EntryLo1($3)==0x0000_0018 (D=0, V=0)

- Valid clean entry (A TLB Modified exception occurs to the first write operation.)

  EntryLo0($2)==0x8000_001a (S=1, D=0, V=1)

  EntryLo1($3)==0x0000_001a (D=0, V=1)

- Valid dirty entry (Normal setting that indicates writable area.)

  EntryLo0($2)==0x8000_001e (S=1, D=1, V=1)

  EntryLo1($3)==0x0000_001e (D=1, V=1)

The SPRAM has a restriction for mapping the page that pairs up with it. For details, refer to "5.2.5. Scratchpad RAM Mapping".

6.5. Uncached Accelerated Buffer (UCAB)

The EE Core has a small-capacity read-only cache for the uncached accelerated space to reduce the bus traffic. This cache is called the Uncached Accelerated Buffer (UCAB).

6.5.1. UCAB Overview

The UCAB is a Direct Map cache of 128 bytes x 1 line. The UCAB is a read-only cache, and only a refill access by a UCAB miss can write data to it.

Cache	Size	Organization	Tag Index
UCAB	128 bytes	128 bytes x 1 Direct Map	-

Table 6-6 UCAB Configuration

The UCAB tag holds the upper 25 bits of the physical address (5 bits from bit 11 to bit 7 are the same as those of the virtual address). In an uncached accelerated load, if the upper 25 bits of the physical address match the UCAB tag address, that is, if there is a hit in the UCAB, then data is provided from the UCAB.
The UCAB is disabled by one of the following:

- Load operation resulting in no hit in the UCAB
- Store operation
- SYNC or SYNC.L instruction
- Some exception

Bus snooping is not supported.
The UCAB tag has the Valid bit (V), but does not have the Dirty, LRF, and Lock bits.
The Valid bit is initialized to 0 when reset.

6.5.2. Non-Blocking Loads and Hit Under Miss

The UCAB supports non-blocking load and hit under miss as well as the Data Cache. Support for a non-blocking load allows the pipeline to continue instruction execution until one of the following occurs even when a refill operation is in the process of execution due to a load instruction resulted in a UCAB miss:

- An instruction which has data dependency with the load instruction that has resulted in a UCAB miss is issued
- A Data Cache miss occurs or a second UCAB miss occurs
- An uncached load instruction is issued
- Pipe 0 stalls

6.6. Cache Control Registers

The operations of the caches are controlled by bits in the Config register. For details, refer to "3.2. System Control Coprocessor (COP0) Registers".

Bit	Description
ICE	Instruction Cache Enable
DCE	Data Cache Enable
IC	Instruction Cache Size (fixed)
DC	Data Cache Size (fixed)

The two cache tag registers TagLo and TagHi are 32-bit read/write registers that are used for setting the cache tag and running diagnostic checks on it.

TagLo Register

31 12	11 7	6	5	4	3	2 0
PTagLo	0	D	V	R	L	0

TagHi Register

31 0

Name	Contents
PTagLo	Physical address bits 31:12
D	Dirty bit (Not used for the Instruction Cache)
V	Valid bit
L	Lock bit (Not used for the Instruction Cache)
R	LRF bit

The contents of the TagHi register have a variety of meanings depending on instructions.

These Tag registers are manipulated by MTC0 and CACHE instructions.

7. Performance Counters and Instruction Stepping

The performance counter provides the means for monitoring and counting the internal events of the CPU and the pipeline during program execution. It is used for tuning the software and hardware. Debuggers can also use the performance counter to provide instruction stepping.

7.1. Configuration of Performance Counter

The performance counter consists of one control register and two counter registers. These three registers are mapped to COP0 register 25 and can be accessed by the dedicated COP0 move instruction.

7.1.1. Performance Counter Control Registers (PCCR)

The Performance Counter Control Register, or PCCR, controls the functions of the performance counter.

31	30 20	19 15	14	13	12	11	10	9 5	4	3	2	1	0
CTE	0	EVENT1	U1	S1	K1	EXL1	0	EVENT0	U0	S0	K0	EXL0	0
1		5	1	1	1	1	1	5	1	1	1	1	1

Name	Pos.	Contents
EXL0	1	PCR0 operation in Level 1 exception handler 0 Not counted 1 Counted
K0	2	PCR0 operation in Kernel mode (except in an exception handler) 0 Not Counted 1 Counted
S0	3	PCR0 operation in Supervisor mode 0 Not Counted 1 Counted
U0	4	PCR0 operation in User mode 0 Not Counted 1 Counted
EVENT0	9:5	Event specification counted by PCR0 (See Table 7-1)
EXL1	11	PCR1 operation in Level 1 exception handler 0 Not Counted 1 Counted
K1	12	PCR1 operation in Kernel mode (except in an exception handler) 0 Not Counted 1 Counted
S1	13	PCR1 operation in Supervisor mode 0 Not Counted 1 Counted
U1	14	PCR1 operation in User mode 0 Not Counted 1 Counted
EVENT1	19:15	Event specification counted by PCR1 (See Table 7-1)
CTE	31	Counter Enable If 1, PCR0 and PCR1 counting and exception generation are enabled.

The PCCR register bits are initially undefined, except for the CTE bit, which is initialized to 0.

7.1.2. Counter Registers (PCR0 / PCR1)

There are two counter registers; PCR0 and PCR1. PCR0 and PCR1 can independently count one event, according to the specifications in the control register.

31	30 ... 0
OVFL	VALUE
1	31

Name	Pos.	Contents
VALUE	30:0	Counter Value
OVFL	31	Counter Overflow Flag

7.1.3. Access to the Performance Counter Registers

Performance counter control register PCCR and counter registers PCR0 and PCR1 are accessed by using the following MFC0 and MTC0 instruction variations, respectively.

Mnemonic	Function
MFPC rt, 0	GPR[rt] ← PCR0 Transfer from counter PCR0 to GPR.
MFPC rt, 1	GPR[rt] ← PCR1 Transfer from counter PCR1 to GPR.
MFPS rt, 0	GPR[rt] ← PCCR Transfer from performance counter control register to GPR.
MTPC rt, 0	PCR0 ← GPR[rt] Transfer from GPR to counter PCR0.
MTPC rt, 1	PCR1 ← GPR[rt] Transfer from GPR to counter PCR1.
MTPS rt, 0	PCCR ← GPR[rt] Transfer from GPR to performance counter control register.

7.1.4. Initial Value of the Performance Counter Registers

The CTE bit of the Performance Counter Control Register PCCR is initialized to 0 upon reset. This prevents event counting and exception generation immediately after reset.
The remaining bits of PCCR, and counter registers PCR0 and PCR1 must be initialized by software.

7.2. Performance Counter Operation Details

7.2.1. Counter Increment

Counters PCR0 and PCR1 increment by 1 whenever the events specified in PCCR.EVENT0 and PCCR.EVENT1 are generated. However, the following three conditions must be met:

1. Counter enable flag PCCR.CTE is set to 1.
2. The processor's operation mode matches the operation mode specified in PCCR.U0 / PCCR.S0 / PCCR.K0 / PCCR.EXL0 or PCCR.U1 / PCCR.S1 / PCCR.K1 / PCCR.EXL1.
3. Level 2 exception handlers are not being executed.

7.2.2. Counter Event

The following table lists the events performance counters PCR0 and PCR1 can detect respectively and the values that are specified in PCCR.EVENT0 and PCCR.EVENT1.

EVENT0/1	Counter 0 (PCR0)	Counter 1 (PCR1)
0	(reserved)	Low-order branch issued
1	Processor cycle	Processor cycle
2	Single instruction issue	Dual instruction issue
3	Branch issued	Branch mispredicted
4	BTAC miss	TLB miss
5	ITLB miss	DTLB miss
6	Instruction cache (I$) miss	Data cache (D$) miss
7	Access to DTLB	WBB single request unavailable
8	Non-blocking load	WBB burst request unavailable
9	WBB single request	WBB burst request almost full
10	WBB burst request	WBB burst request full
11	CPU address bus busy	CPU data bus busy
12	Instruction completed	Instruction completed
13	Non-BDS instruction completed	Non-BDS instruction completed
14	COP2 instruction completed	COP1 instruction completed
15	Load completed	Store completed
16	No event	No event
17-31	(reserved)	(reserved)

Table 7-1 Events that the Performance Counters Support

7.2.3. Counter Event Descriptions

The following are detailed descriptions of the events that the performance counter can count.

Low-order branch issued [PCR1, PCCR.EVENT1=0]

This event occurs whenever a branch is issued in the low-order (even address) position. The feature to count this event is required, since only these branches are subject to BTAC lookup.
Note that a branch in this case is accompanied by the branch prediction (i.e. conditional branches and J/JAL instruction). The JR, JALR, ERET, SYSCALL, BREAK, or TRAP instructions do not generate this event.

Processor cycle [PCR0, PCCR.EVENT0=1 / PCR1, PCCR.EVENT1=1]

This event occurs every CPU clock cycle.

Single instruction issue [PCR0, PCCR.EVENT0=2]

This event occurs when an instruction is issued in only one of the two EE Core logical pipelines.

Dual instruction issued [PCR1, PCCR.EVENT1=2]

This event occurs when an instruction is issued in the two EE Core logical pipelines. The counter value increments by 1 at this point. Therefore, the number of instructions issued in a certain period can be obtained from the following expression.
(Dual instruction issued) x 2 + (Single instruction issued)

Branch issued [PCR0, PCCR.EVENT0=3]

This event occurs when a branch is issued. If the instruction prior to the branch instruction generates an exception, the branch instruction may be cancelled even when this event occurs.
Note that a branch in this case is accompanied by the branch prediction (i.e. conditional branches and J/JAL instruction). The JR, JALR, ERET, SYSCALL, BREAK, or TRAP instructions do not generate this event.

Branch mispredicted [PCR1, PCCR.EVENT1=3]

This event occurs when the prediction of a branch address is incorrect in conditional branches. The TRAP instruction does not generate this event. Note that a branch may get cancelled if the instruction prior to it signals an exception.

BTAC miss [PCR0, PCCR.EVENT0=4]

This event occurs when the lookup to BTAC (Branch Target Address Cache) fails.
This enables counting of the low-order (even address) branch instructions that hit the BTAC. Note that high-order (odd address) branch instructions do not refer to the BTAC.

TLB miss [PCR1, PCCR.EVENT1=4]

This event occurs when a TLB miss occurs.

ITLB miss [PCR0, PCCR.EVENT0=5]

This event occurs when an ITLB miss occurs.

DTLB miss [PCR1, PCCR.EVENT1=5]

This event occurs when a DTLB miss occurs.
DTLB is accessed even when an unmapped virtual address is accessed.

Instruction cache miss [PCR0, PCCR.EVENT0=6]

This event occurs when an instruction cache miss occurs. An uncached instruction fetch will not be counted.

Data cache miss [PCR1, PCCR.EVENT1=6]

This event occurs when a bus read access occurs during a load, store, or prefetch instruction. It occurs if a load, store, or prefetch instruction to the cached area results in a cache miss. This event also occurs during a load operation from the cached area, when the cache is disabled (Config.DCE=0), and during a load operation from an uncached or an uncached accelerated area.

Access to DTLB [PCR0, PCCR.EVENT0=7]

This event occurs when an access to DTLB occurs.
This event counts the total number of loads and stores executed to the cached area (including cancelled ones). If there are no uncached loads and stores, dividing the counter value of the "data cache miss" event by that of the "access to DTLB" event provides a good estimate of the data cache miss rate.
DTLB is accessed even when an unmapped virtual address is accessed.

WBB single request unavailable [PCR1, PCCR.EVENT1=7]

This event occurs when a single request is issued to a WBB having insufficient free entries (i.e. all eight entries have already been used).

Non-blocking load [PCR0, PCCR.EVENT0=8]

This event occurs when a non-blocking cache miss (first cache miss) occurs due to a load instruction. It also occurs when a UCAB miss occurs or a load instruction from the uncached area is executed.

WBB burst request unavailable [PCR1, PCCR.EVENT1=8]

This event occurs when a burst request is issued to a WBB having insufficient free entries (i.e. five or more entries have already been used).

WBB single request [PCR0, PCCR.EVENT0=9]

This event occurs when a single request is issued to the WBB.

WBB burst request almost full [PCR1, PCCR.EVENT1=9]

This event occurs when a burst request is issued to a WBB having insufficient free entries (i.e. five to seven entries have already been used).

WBB burst request [PCR0, PCCR.EVENT0=10]

This event occurs when a burst request is issued to the WBB.

WBB burst request full [PCR1, PCCR.EVENT1=10]

This event occurs when a burst request is issued to the WBB with no free entries (i.e. all eight entries have already been used).

CPU address bus busy [PCR0, PCCR.EVENT0=11]

This event occurs every BUSCLK (not CPU clock) when the CPU address bus is unavailable.
The CPU address bus is considered unavailable if it is busy or data for the first address (out of two addresses issued) has not yet been returned.

CPU data bus busy [PCR1, PCCR.EVENT1=11]

This event occurs every BUSCLK (not CPU clock) when the CPU data bus is unavailable.

Instruction completed [PCR0, PCCR.EVENT0=12 / PCR1, PCCR.EVENT1=12]

This event occurs when an instruction is completed (reaches the stage in which the instruction is sure to be ended).
Subtracting the count value of the "instruction completed" event from the number of executed instructions provides the number of cancelled instructions.
Some instructions including SYSCALL and TEQ generate exceptions as part of their operations. It is considered that such instructions should complete, regardless of the occurrence of the exceptions. Even if the condition of the TEQ instruction succeeds and causes a Trap exception, an instruction complete event occurs. However, if an exception due to another cause occurs at this time, the instruction is cancelled, and no instruction complete event occurs.
Even if an instruction in the branch delay slot (BDS) of a branch-likely instruction does not meet the branch conditions, and is nullified, an instruction complete event occurs.
Note that up to two instructions can be executed every CPU cycle in the EE Core. Other instruction completion type events are in like manner, but the event counter is incremented by two when two instructions are executed and completed. Therefore, it is ambiguous which instruction causes counter exceptions. When it is inconvenient, a dual instruction issue must be prohibited.

Non-BDS instruction completed [PCR0, PCCR.EVENT0=13 / PCR1, PCCR.EVENT1=13]

This event occurs when an instruction that does not have a branch delay slot completes (or reaches the stage in which the instruction is sure to be finished).
This event does not occur when the branch instructions or jump instructions complete, but does occur when the instruction in the branch delay slot completes. This event also occurs when an instruction in the branch delay slot of a branch-likely instruction is nullified (the branch condition is not met).

COP2 instruction completed [PCR0, PCCR.EVENT0=14]

This event occurs when a COP2 instruction completes. This event also occurs when a COP2 instruction in the branch delay slot of a branch-likely instruction is nullified (the branch condition is not met).

COP1 instruction completed [PCR1, PCCR.EVENT1=14]

This event occurs when a COP1 instruction completes. This event also occurs when a COP1 instruction in the branch delay slot of a branch-likely instruction is nullified (the branch condition is not met).

Load completed [PCR0, PCCR.EVENT0=15]

This event occurs when a load instruction completes. This event even occurs when a load instruction is in the branch delay slot of the branch-likely instruction, and the branch condition is not met and nullified. In addition, this event occurs even when a bus error occurs.

Store completed [PCR1, PCCR.EVENT1=15]

This event occurs when a store instruction completes. This event even occurs when a store instruction is in the branch delay slot of the branch-likely instruction, and the branch condition is not met and nullified. In addition, this event occurs even when a bus error occurs.

No event [PCR0, PCCR.EVENT0=16 / PCR1, PCCR.EVENT1=13]

This event is a virtual event, and effectively disables the corresponding counter from counting up. It is available if one of the two counters needs to be activated.

7.2.4. Occurrence of Counter Exceptions

A counter exception occurs when one of the performance counters overflows. The condition to generate the counter exception is shown in the following expression.

```
STATUS.ERL && PCCR.CTE && (PCR0.OVFL || PCR1.OVFL)
```

When a counter exception occurs, after the instruction being executed is cancelled (see notes provided later in this chapter), control is transferred to the counter exception handler using the following processing.

```
if ( in branch delay slot ) {
  ErrorEPC = PC - 4;
  CAUSE.BD2 = 1;
}
else {
  ErrorEPC = PC;
  CAUSE.BD2 = 0;
}
if ( STATUS.DEV )
  PC = 0xBFC00280;  // Entry point for bootstrap (Uncached)
else
  PC = 0x80000080;  // Normal entry point
STATUS.ERL = 1;
CAUSE.EXC2 = 2;    // Counter exception
```

Since the normal exception entry point is in kseg0 space, the address is unmapped in the counter exception handler. Config.K0 determines the caching policy. If the cache must be saved at the time of the occurrence of a counter exception, kseg0 should be configured in uncached mode. If the processing performance is more important than caching, kseg0 should be configured in cached mode.

7.2.5. Priority of Counter Exceptions

Counter exceptions have the highest priority after the Reset and NMI exceptions.
If the Reset exception occurs, the program is initialized, and a simultaneous counter exception is ignored.
If the NMI and counter exceptions occur at the same time, the control is transferred to the NMI exception handler with the OVFL bit of the counter set to 1 and the ErrorEPC register pointing at the instruction causing the counter overflow. If the NMI exception handler exits, the instruction that caused the overflow is re-executed. However, since the OVFL bit is set, the instruction is cancelled once more, and the control is transferred to the counter exception handler.

7.2.6. Initializing Performance Counters

The following pseudo code sequence is needed to initialize and activate the performance counters. In the example below, PCR0 is set up to count CPU clocks in all operating modes and generate a counter exception after the count exceeds 2^{31}. PCR1 counts stores while in supervisor mode and generates a counter exception after 256 such stores. The code must be executed while in kernel mode.

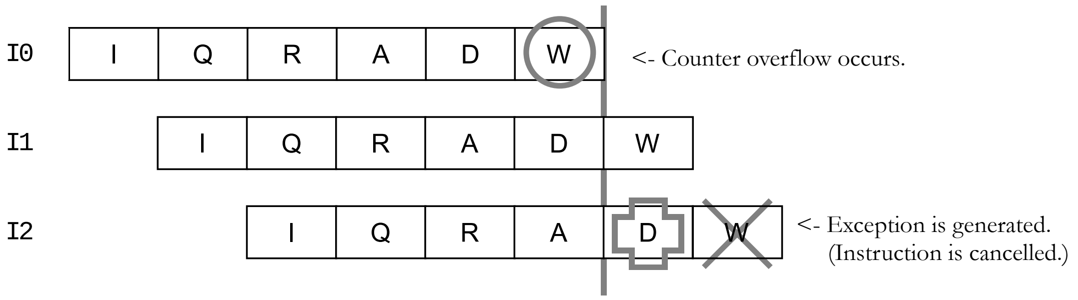
```
STATUS.ERL = 1;            // Set ERL (to inhibit counting)
ErrorEPC = <target address where counting is to start>

PCR0 = 0;                  // Initialize PCR0 and …
PCCR.EVENT0 = 1;           // … set up to count CPU clocks …
PCCR.U0 = 1;               //  in all operation modes
PCCR.S0 = 1;
PCCR.K0 = 1;
PCCR.EXL0 = 1;

PCR1 = 0x7FFFFF00;         //  Initialize PCR1 to 2^31 – 256, and …
PCCR.EVENT1 = 15;          // … set up to count completed stores …
PCCR.U1 = 0;               //  … while in super visor mode
PCCR.S1 = 1;
PCCR.K1 = 0;
PCCR.EXL1 = 0;

PCCR.CTE = 1;   //  Enable counter operations
ERET            //  Clear ERL to jump to the target address
                //  (Also guarantee that the COP0 registers are updated.)
```

7.2.7. Notes on Pipelining

Counters are incremented immediately after an event occurs. The occurrence of an event does not correspond to the execution of an instruction, except for an event that specifies instruction completion. Even when an instruction is nullified and leaves no results, the events generated so far are counted.

An event that specifies instruction completion (e.g. "load completed") is generated in the 2W stage where it is guaranteed to complete. (Even if a bus error occurs in this stage, the load is considered to have been completed and an event is generated.) Instruction completion is always in-order. Even if an instruction features out-of-order completion, it must generate "instruction completed" events in order.

The general rule for internal exception handling is that only the instruction that has caused the exception and the following instruction in process in the pipeline are cancelled. For counter exceptions, however, instructions in process but which are older than the one causing the exception may also be cancelled. It is permitted only when the event being counted is not an instruction completion event. If the performance counter counts ICache misses that cause an exception, instructions in the pipe prior to the one causing the ICache miss may get cancelled. If the exception handler examines the instruction word at ErrorEPC, it may find the word to be free from the ICache miss. The instruction-completion-type events generated at stage 2W are detected as exceptions by the EE Core at stage 2D. Therefore, there is a difference between the instructions that cause exceptions and the instructions where the exceptions are generated. An example is shown in the figure below.



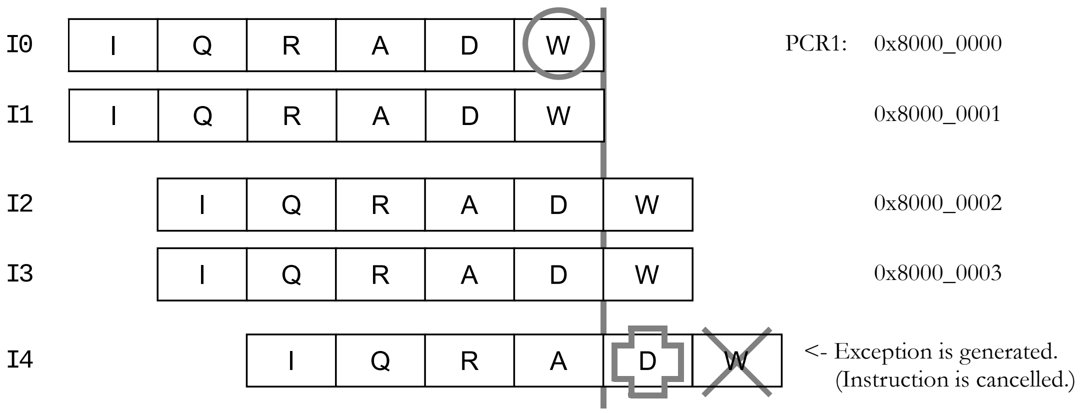
7.2.8. Notes on Instruction Stepping

The following setting causes the program to be trapped after executing *k* steps.

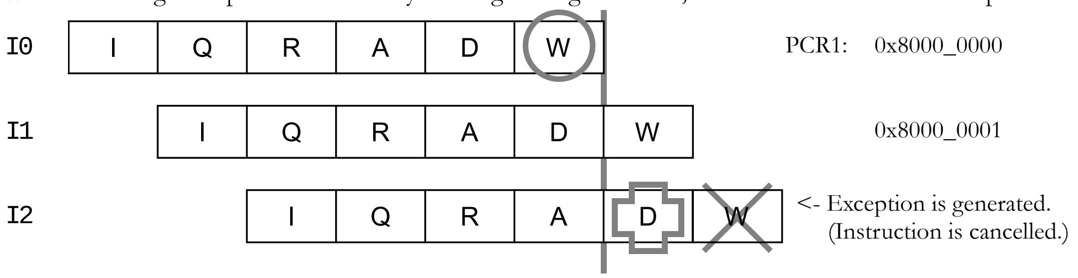
PCCR.EVENT1=13

PCR1=0x8000_0000 - *k*

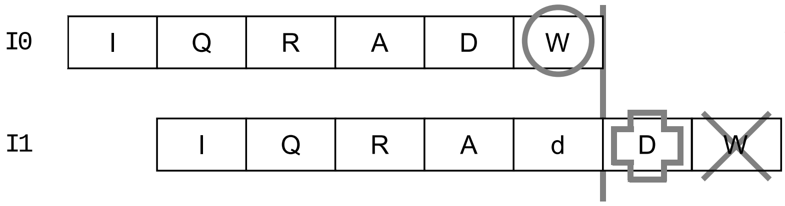
However, as described in "7.2.7. Notes on Pipelining", a counter exception might occur a few instructions after the counter overflows. In the following example, the counter overflows when Instruction I0 is completed, but an exception occurs in Instruction I4. The PCR1 value indicates three more instructions have been completed in between.



When canceling multiple instructions by clearing Config.DIE to 0, one more instruction is completed.



If I1 has a stall condition, there are no more instructions to be executed.



PCR1: 0x8000_0000

<- Exception is generated.
(Instruction is cancelled.)

Note that inaccuracy is unavoidable in instruction stepping.

(This page is left blank intentionally)

8. Floating-Point Unit (FPU)

This chapter describes the floating-point unit (FPU=COP1) of the EE Core. The following is an overview of the FPU's features :

- High performance single-precision floating-point unit tightly coupled to the EE Core
- Supports single-precision format, as defined in the IEEE 754 specification
- No support for NaNs and plus/minus infinite is provided. Plus/minus "0" support is provided
- No hardware exception mechanism to affect the instruction stream
- Compatible with coprocessor 2 (VPU)

8.1. Data Formats

8.1.1. Floating-Point Format

The FPU only supports 32-bit single-precision for floating-point numbers. The numeric value format is based on the IEEE754 standard, and has a 24-bit signed fraction and an 8-bit exponent.

31	30 — 23	22 — 0
s Sign	e Exponent	f Fraction
1	8	23

Specification	
Size (Width)	32 bits
Exponent	8 bits
Integer bit	hidden
Fraction	24 bits
Emax (Maximum Exponent)	+128
Emin (Minimum Exponent)	-126
Biased Exponent	+127

Figure 8-1 Single-Precision Floating-Point Format

The true exponent E is calculated by subtracting the biased value from the exponent e. The range of E can be every integer value between Emin and Emax. The value of a single-precision floating-point number is shown below.

IF $0 < e <= 255$, then $(-1)^s \bullet 2^{e-127}(1.f)$

IF $e = 0$, then $(-1)^s \bullet 0$

Note that FPU does not support denormalized numbers, plus and minus infinities, and NaN (Not a Number) as defined in IEEE754.

8.1.2. Fixed-Point Format

Fixed-point values are in 2's complement format, shown in Figure 8-2, and unsigned fixed-point values are not directly supported.

31 — 0
Fixed-point (2's complement)
32

Figure 8-2 Fixed-Point Format

8.2. FPU Registers

The FPU has 32 general-purpose registers (FPRs), each of which is 32 bits wide. The CPU can access these registers through move (MFC1 and MTC1), load (LWC1) and store (SWC1) instructions.
There are 32 floating-point control registers (FCR), of which only two (FCR0 and FCR31) are implemented. Details of these are described later.
In addition, a 32-bit accumulator is used in the multiply-accumulate type instructions.

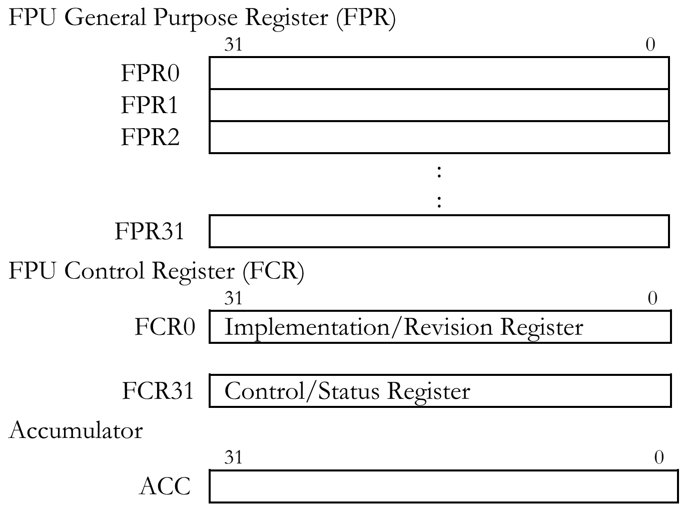


Figure 8-3 FPU Registers

8.3. FPU Control Registers

The FPU has 32 control registers (FCRs), which can only be accessed by move instructions (CFC1 and CTC1). However, FCR1 to FCR30 are reserved registers, and only FCR0 (Implementation/Revision register) and FCR31(Control/Status register) are implemented.

8.3.1. Implementation and Revision Register (FCR0)

The Implementation and Revision register (FCR0) is read-only and contains the implementation number that indicates specifications of FPU, and the revision number that indicates changes made to design and production process of the chip. This information can determine the FPU capability and performance level, and can be used for diagnostic software.

31 16	15 8	7 0
0	Imp	Rev
16	8	8

Figure 8-4 Implementation/Revision Register

Field	Pos.	Description	r/w	Initial Value
0	31:16	Fixed as zero (reserved)	r /-	0
Imp	15:8	Implementation number	r /-	0x2E
Rev	7:0	Revision number bits 7:4 and bits 3:0 indicate a major and minor revision respectively.	r /-	Revision Number

Table 8-1 Implementation/Revision Register Fields

8.3.2. Control/Status Register (FCR31)

The Control/Status register (FCR31) contains FPU status information, such as results of arithmetic operations.

				Cause Flags					Sticky Flags						
31 25	24	23		17	16	15	14		6	5	4	3	2	1	0
0	1	C	0	I	D	O	U	0	SI	SD	SO	SU	0	0	1
7	1	1	5	1	1	1	1	7	1	1	1	1	1	1	1

Figure 8-5 Control/Status Register

Four flags—I, D, O, and U—are collectively referred to as the Cause flags. Likewise, four flags—SI, SD, SO, and SU—are collectively referred to as the Sticky flags (Accumulation flags).
Cause flags indicate the result of the immediately prior arithmetic instruction. Sticky flags are set to 1 when the corresponding Cause flags become 1, and are never set to 0 unless software explicitly clears them. That is, part of the program indicates the result of arithmetic instructions.

Field	Pos.	Description	r/w	Initial Value
C	23	Condition bit The bit is set to 1 when the result of a floating-point Compare operation is true and the bit is cleared to 0 when the result is false.	r/w	0
I	17	Invalid Operation flag The bit is set to 1 when attempting to execute 0/0 division, square root of a negative number or reciprocal square root of a negative number. Otherwise, the bit is cleared to 0.	r/w	0
D	16	Division by Zero flag The bit is set to 1 when attempting to execute division by zero. Otherwise, the bit is cleared to 0.	r/w	0
O	15	Overflow flag The bit is set to 1 when the exponent of the computational result overflows. Otherwise, the bit is cleared to 0.	r/w	0
U	14	Underflow flag The bit is set to 1 when the exponent of the computational result underflows. Otherwise, the bit is cleared to 0	r/w	0
SI	6	Invalid Operation cumulative flag The bit is set to 1 when attempting to execute 0/0 division, square root of the negative number or reciprocal square root of the negative number.	r/w	0
SD	5	Division by Zero cumulative flag The bit is set to 1 when attempting to execute division by zero.	r/w	0
SO	4	Overflow cumulative flag The bit is set to 1 when the exponent of the computational result overflows.	r/w	0
SU	3	Underflow cumulative flag The bit is set to 1 when the exponent of the computational result underflows.	r/w	0

Table 8-2 Control/Status Register Fields

8.4. Instruction Set Overview

All FPU instructions are 32 bits long and aligned on a word boundary.
FPU instructions can be divided into the following categories:

- Move instructions: Instructions which move data between memory and the main processor and between FPU general-purpose registers and FPU control registers
- Conversion instructions: Instructions that convert data formats of numeric data
- Computational instructions: Instructions that perform arithmetic operations on floating-point data in the FPU general-purpose register
- Compare instructions: Instructions that compare the contents of FPU general-purpose registers and reflect the result in the C bit of the status register
- Branch on FPU condition instructions: Instructions that branch according to the C bit of the status register

The list of FPU instructions is shown below. For details of each instruction, refer to the "EE Core Instruction Set Manual".

Category	Instruct.	Description
Move Instruction	LWC1	Load Word to FPR
	SWC1	Store Word from FPR
	MTC1	Move Word to FPR
	MFC1	Move Word from FPR
	MOV.S	Single Floating-point Move
	CTC1	Move Control Word to FCR
	CFC1	Move Control Word from FCR
Conversion Instruction	CVT.S.W	32-bit Fixed Point Convert to Single Floating-point
	CVT.W.S	Single Floating-point Convert to 32-bit Fixed Point
Computational Instruction	ADD.S	Single Floating-point Add
	SUB.S	Single Floating-point Subtract
	MUL.S	Single Floating-point Multiply
	DIV.S	Single Floating-point Divide
	ABS.S	Single Floating-point Absolute
	NEG.S	Single Floating-point Negate
	SQRT.S	Single Floating-point Square Root
	ADDA.S	Single Floating-point Add to Accumulator
	SUBA.S	Single Floating-point Subtract to Accumulator
	MULA.S	Single Floating-point Multiply to Accumulator
	MADD.S	Single Floating-point Multiply and Add
	MADDA.S	Single Floating-point Multiply and Add to Accumulator
	MSUB.S	Single Floating-point Multiply and Subtract
	MSUBA.S	Single Floating-point Multiply/Subtract from Accumulator
	RSQRT.S	Single Floating-point Reciprocal Square Root
Computational Instruction	MAX.S	Single Floating-point Maximum
	MIN.S	Single Floating-point Minimum
Comparison	C.cond.S	Single Floating-point Compare
Branch on FPU Condition	BC1T	Branch on FPU True
	BC1F	Branch on FPU False
	BC1TL	Branch on FPU True Likely
	BC1FL	Branch on FPU False Likely

Table 8-3 FPU Instructions

8.5. Results of Abnormal Computation

If abnormal computations such as "Divide by Zero" are performed, or an overflow or underflow occurs, the resulting values and flags set in the status register are as shown in the table below.

Computational Exception	Result Value	Cause Flag
Divide by Zero	+Fmax or –Fmax	D=1
0 / 0	+Fmax or –Fmax	I=1
Square root of a negative number	Square root of the absolute value of the parameter	I=1
Exponent Overflow	+Fmax or –Fmax	O=1
Exponent Underflow	+0 or –0	U=1
Conversion Overflow	+Fmax or –Fmax	None

*Fmax is the maximum number in a single-precision floating-point format.

Table 8-4 Results of Abnormal Computation

8.6. Sign of Zero

In a single-precision floating-point format, two zeros, +0 and –0 are present. The results when using signed 0 as both of the operands are shown in the table below. This is compatible with the IEEE 754 specification.

Operation	Result	I/SI flag	D/SD flag
(+0)+(+0)	+0	-/-	-/-
(+0)+(-0)	+0	-/-	-/-
(-0)+(+0)	+0	-/-	-/-
(-0)+(-0)	-0	-/-	-/-
(+0)-(+0)	+0	-/-	-/-
(+0)-(-0)	+0	-/-	-/-
(-0)-(+0)	-0	-/-	-/-
(-0)-(-0)	+0	-/-	-/-
(+0)×(+0)	+0	-/-	-/-
(+0)×(-0)	-0	-/-	-/-
(-0)×(+0)	-0	-/-	-/-
(-0)×(-0)	+0	-/-	-/-
(+0)/(+0)	7FFFFFFF	1/1	0/0
(+0)/(-0)	FFFFFFFF	1/1	0/0
(-0)/(+0)	FFFFFFFF	1/1	0/0
(-0)/(-0)	7FFFFFFF	1/1	0/0
Max(+0,+0)	+0	-/-	-/-
Max(+0,-0)	+0	-/-	-/-
Max(-0,+0)	+0	-/-	-/-
Max(-0,-0)	-0	-/-	-/-
Min(+0,+0)	+0	-/-	-/-
Min(+0,-0)	-0	-/-	-/-
Min(-0,+0)	-0	-/-	-/-
Min(-0,-0)	-0	-/-	-/-
(+0)/SQRT(+0)	7FFFFFFF	1/1	0/0
(+0)/SQRT(-0)	FFFFFFFF	1/1	0/0
(-0)/SQRT(+0)	FFFFFFFF	1/1	0/0
(-0)/SQRT(-0)	7FFFFFFF	1/1	0/0
(+fs)/(+0)	7FFFFFFF	0/0	1/1
(+fs)/(-0)	FFFFFFFF	0/0	1/1
(-fs)/(+0)	FFFFFFFF	0/0	1/1
(-fs)/(-0)	7FFFFFFF	0/0	1/1

Table 8-5 Computation of Signed Zero

8.7. Rounding

The FPU only supports "Rounding towards 0". "Rounding towards Nearest" and "Rounding towards +/- infinities" as defined by IEEE 754 are not supported.
Since "Rounding towards Nearest" is not supported, the FPU does not use the Guard, Round and Sticky bits during rounding. Also, since "Rounding toward 0" does not require full value prior to rounding, unlike the definition of IEEE 754, the results may differ from the IEEE 754 Rounding to 0. This difference is usually restricted to the least significant bit only.
Since the rounding mode is not programmable in this FPU, the two least significant bits of the Control and Status registers (FCR31) are hardwired to "01".

8.8. IEEE 754 Compatibility

The FPU is not compatible with the IEEE 754 Floating-Point standard. Only single-precision operations are supported. Overflow and Underflow are detected only for overflow or underflow of the exponent. While the IEEE standard recommends trapping a computational exception, in the FPU, processing continues by just setting a flag. Since these flags can be sampled on an instruction by instruction basis, emulating this trap is possible if necessary.
In addition, the following shows the differences from the IEEE 754 specification. When computational results are different, refer to Table 8-6.

- NaN (Not a Number), +∞, -∞ and denormalized numbers are not supported.
- The FPU does not use the Guard, Round and Sticky bits during computations. The computed result usually differs from IEEE 754 only in the least-significant bit. For saturating instructions, bits other than the least-significant bit can be different.
- Only "Rounding towards 0" is supported and "Rounding towards Nearest", "Rounding towards +/-∞" are not supported. The result of "Rounding towards 0" can differ from IEEE 754 in the least significant bit.
- IEEE 754-defined exceptions are not fully supported. In particular, Invalid Operation exceptions due to NaN, +/-∞ and Inexact exceptions are not supported.

Operation	IEEE 754	FPU
0/0	Result is a "NaN". Invalid Operation exception is taken.	Result is +Fmax or -Fmax. I bit (SI bit) is set.
Sqrt (negative number)	Result is a "NaN". Invalid Operation exception is taken.	Result is Sqrt (|x|). I bit (SI bit) is set.
Division by zero	Result is +∞ or -∞ Division by Zero exception is taken	Result is +Fmax or -Fmax. D bit (SD bit) is set.
Exponent overflow	Result is +∞, -∞, +Fmax or -Fmax (determined by the rounding mode) Overflow exception is taken	Result is +Fmax or -Fmax. O bit (SO bit) is set.
Exponent underflow	Result is Denormalized value. Underflow exception is taken	Result is +0 or -0. U bit (SU bit) is set.
Conversion of Floating-point to Integer Overflow	Result is not defined. Invalid Operation exception is taken.	Result is 2^{31}-1 or -2^{31} I bit (SI bit) is set.

Table 8-6 Differences of the Computational Results between IEEE 754 and FPU

(This page is left blank intentionally)

9. Hardware Breakpoint

This chapter describes hardware breakpoint function, one of the debugging support functions provided by the EE Core.

9.1. Overview of Hardware Breakpoint

The EE Core has a hardware breakpoint function for debugging purposes. The main features are as follows:

- Provision of both instruction and data breakpoints in the virtual address
- Instruction address breakpoint with address masking
- Data breakpoint by three events: address with masking, data value with masking, and directions of reading and writing
- Independent breakpoint control for instruction and data
- Breakpoint function control by the processor's operating mode
- Provision of an external signal when the breakpoint conditions are met

When the breakpoint conditions are met, a debugging exception, one of the level 2 exceptions, can be generated. This exception is masked only when level 2 exceptions are in process. The occurrence of this exception is controllable by the breakpoint control register.
Note that some data value breakpoint exceptions are imprecise, because some instructions are completed before data is read from memory. The following summarizes imprecise cases:

- Data value breakpoint in a load instruction
- Data value breakpoint in a SWC1 instruction
- Data value breakpoint in a SQC2 instruction

The hardware breakpoint is implemented as a part of the COP0 functions, and controlled by specifying values to seven registers using dedicated transfer instructions.
Figure 9-1 shows an overview of breakpoint operations.

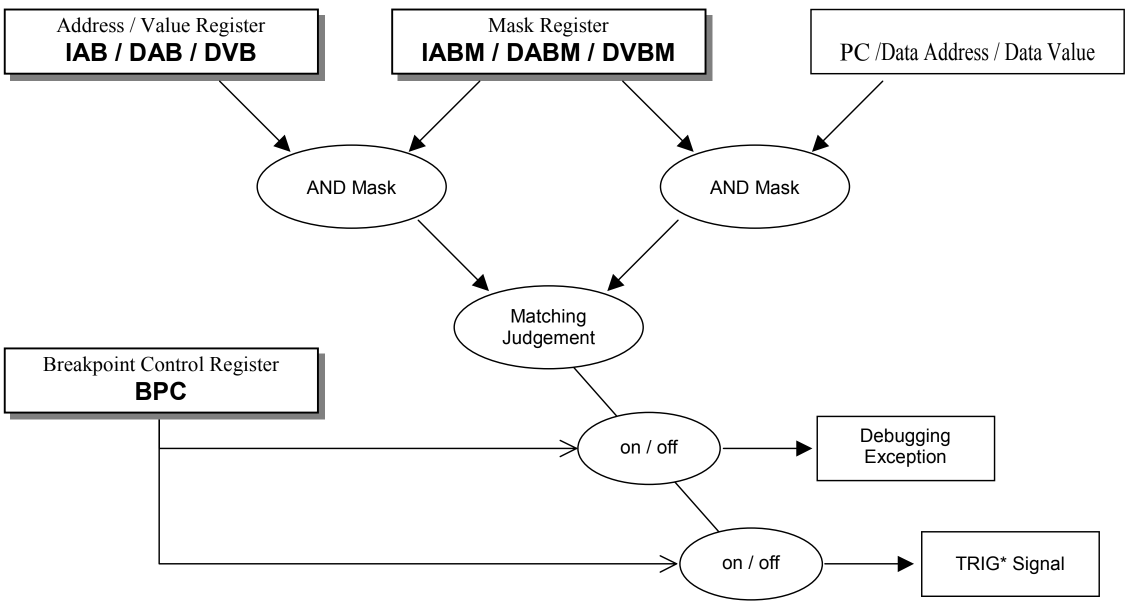


Figure 9-1 Overview of Breakpoint Operations

9.2. Breakpoint Registers

The hardware breakpoint uses one breakpoint control register and three pairs of breakpoint registers (seven registers in total).

Register Name	Function
BPC	Breakpoint control register
IAB	Instruction address breakpoint register
IABM	Instruction address breakpoint mask register
DAB	Data address breakpoint register
DABM	Data address breakpoint mask register
DVB	Data value breakpoint register
DVBM	Data value breakpoint mask register

All of these registers are writable/readable in 32-bit width, and mapped to COP0 register 24. The following special instructions are provided for transferring data between these registers and the general-purpose registers.

Register Name	Read Instruction	Write Instruction
BPC	MFBPC	MTBPC
IAB	MFIAB	MTIAB
IABM	MFIABM	MTIABM
DAB	MFDAB	MTDAB
DABM	MFDABM	MTDABM
DVB	MFDVB	MTDVB
DVBM	MFDVBM	MTDVBM

The function and usage of each register are described as follows:

9.2.1. Breakpoint Control (BPC) Register

The Breakpoint control register (BPC register) has the control bit for the breakpoint function and the status flag.

31	30	29	28		26	25	24	23		21	20	19	18	17	16	15		02	01	00
IAE	DRE	DWE	DVE	0	IUE	ISE	IKE	IXE	0	DUE	DSE	DKE	DXE	ITE	DTE	BED	0	DWB	DRB	IAB

IAE / DRE / DWE / DVE: Control of entire breakpoint functions

These bits enable/disable these functions: the instruction address breakpoint, the data address breakpoint (when reading), the data address breakpoint (when writing), and the data value breakpoint. The corresponding breakpoint function is valid when these bits are 1.

IUE / ISE / IKE / IXE: Control of instruction breakpoint in each processor mode

These bits enable/disable the instruction address breakpoint function in the User, Supervisor, Kernel, and Level 1 exception handler execution (Kernel) modes. The corresponding instruction address breakpoint function is valid when these bits are 1. When the IAE bit is 0, the instruction address breakpoint function is disabled, and these bits are meaningless.

DUE / DSE / DKE / DXE: Control of data breakpoint in each processor mode

These bits enable/disable the data address breakpoint function in the User, Supervisor, Kernel, and Level 1 exception handler execution (Kernel) modes. The corresponding data address breakpoint function is valid when these bits are 1. When both the DRE and DWE bits are 0, the data address breakpoint function is disabled, and these bits are meaningless.

ITE / DTE: Control of Trigger Signal Output

These bits enable/disable trigger signal generation when the condition of the instruction address breakpoint or data breakpoint is established. If the corresponding breakpoint condition is established when these bits are 1, the TRIG* signal is asserted.

BED: Control of Debug Exceptions

This bit enables/disables debug exception generation when the condition of the instruction address breakpoint or data breakpoint is established. When the bit is 1, a debug exception does not occur even though the breakpoint conditions are met.
Note that the setting of this bit does not affect trigger signal generation set by the ITE or DTE bit.

DWB / DRB / IAB: Breakpoint Condition Establishment Flag

These bits become 1 when the conditions of the data breakpoint (when writing), the data breakpoint (when reading), or the instruction address breakpoint are established.

9.2.2. Instruction Address Registers (IAB / IABM)

The Instruction Address Breakpoint register (IAB register) and the Instruction Address Breakpoint Mask register (IABM register), as a pair, specify the range of instruction addresses that are to be the breakpoint condition.
Virtual addresses are specified in the IAB register, and a mask that indicates valid bits in the virtual addresses is specified in the IABM register. If the AND between the IAB and IABM registers matches the AND between the program counter and the IABM register, the condition of the instruction address breakpoint is established. Since the lower two bits of the program counter are always set to 0, the lower two bits of these registers are also fixed to 0.

9.2.3. Data Address Registers (DAB / DABM)

The Data Address Breakpoint register (DAB register) and the Data Address Breakpoint Mask register (DABM register), as a pair, specify the range of data addresses that are to be the breakpoint condition.
Virtual addresses are specified in the DAB register, and a mask that indicates valid bits in the virtual addresses is specified in the DABM register. If the AND between the DAB and DABM registers matches the AND between the effective address of the load/store instruction and the DABM register, the condition of the data address breakpoint is established.

9.2.4. Data Value Registers (DVB / DVBM)

The Data Value Breakpoint register (DVB register) and the Data Value Breakpoint Mask register (DVBM register), as a pair, specify the data values that are to be the breakpoint condition. Values are specified in the DVB register, and a mask that indicates valid bits in the values is specified in the DVBM register.
If the data address breakpoint conditions are met and the AND between the DVB and DVBM registers matches the AND between the data to be loaded/stored and the DVBM register, the conditions of the data value breakpoint are established.
Since the data value register has a 32-bit width, the data to be loaded/stored is treated as follows:

- Only the lower 32 bits of GPR in store instructions
- Only the lower 32 bits of loaded data in LQ/LD instructions
- The value obtained by sign-extending the loaded data to 32 bits in LH/LB instructions
- The lower 32 bits of the value loaded and merged with the GPR value in load instructions that disregard alignments (e.g. LWL, LDL)

9.2.5. Establishment of Breakpoint and Operation of Exception Generation

Figure 9-2 is a flowchart indicating the first part of the operations that determine the establishment of the hardware breakpoint.

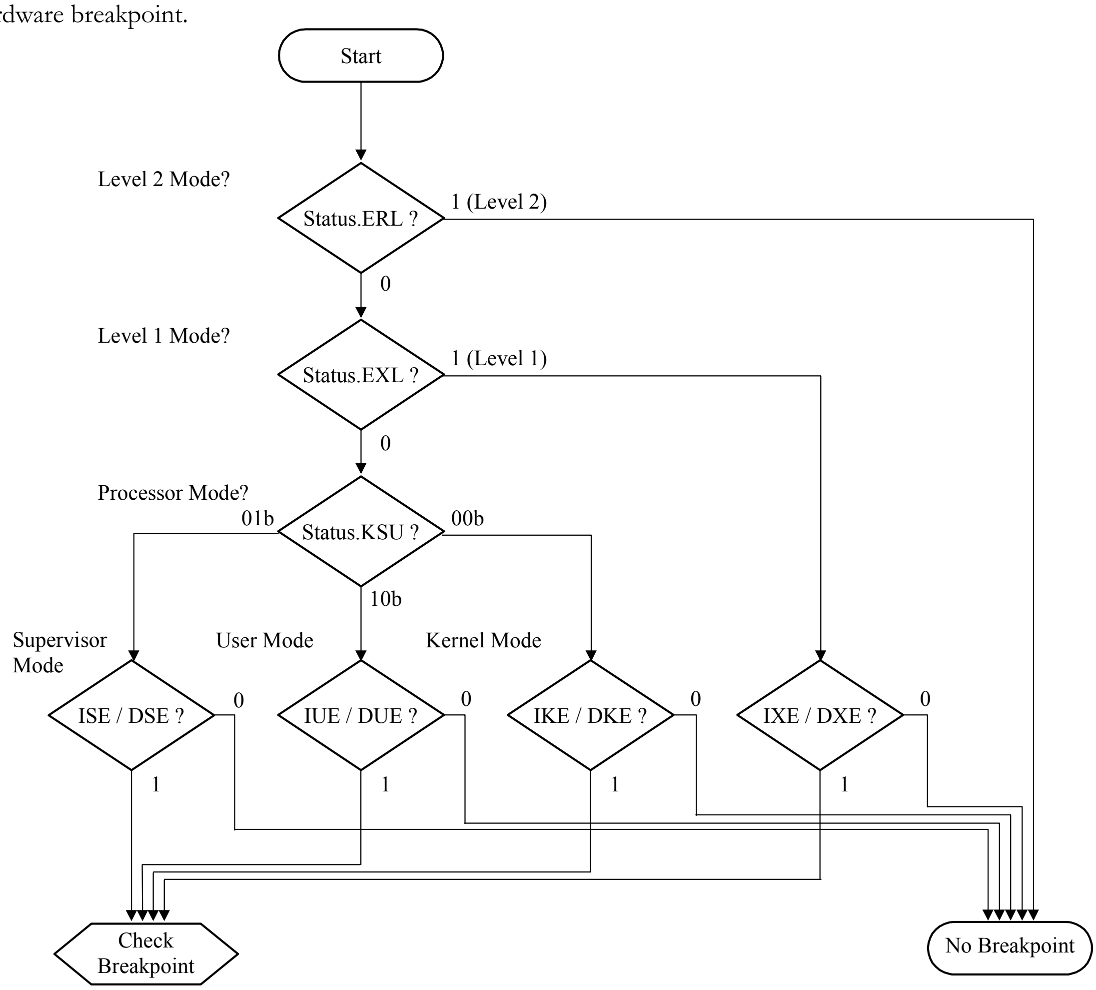


Figure 9-2 Breakpoint Operation (1)

The following flowchart shows the establishment of an instruction breakpoint in response to Figure 9-2 and generation of an exception or a trigger signal.

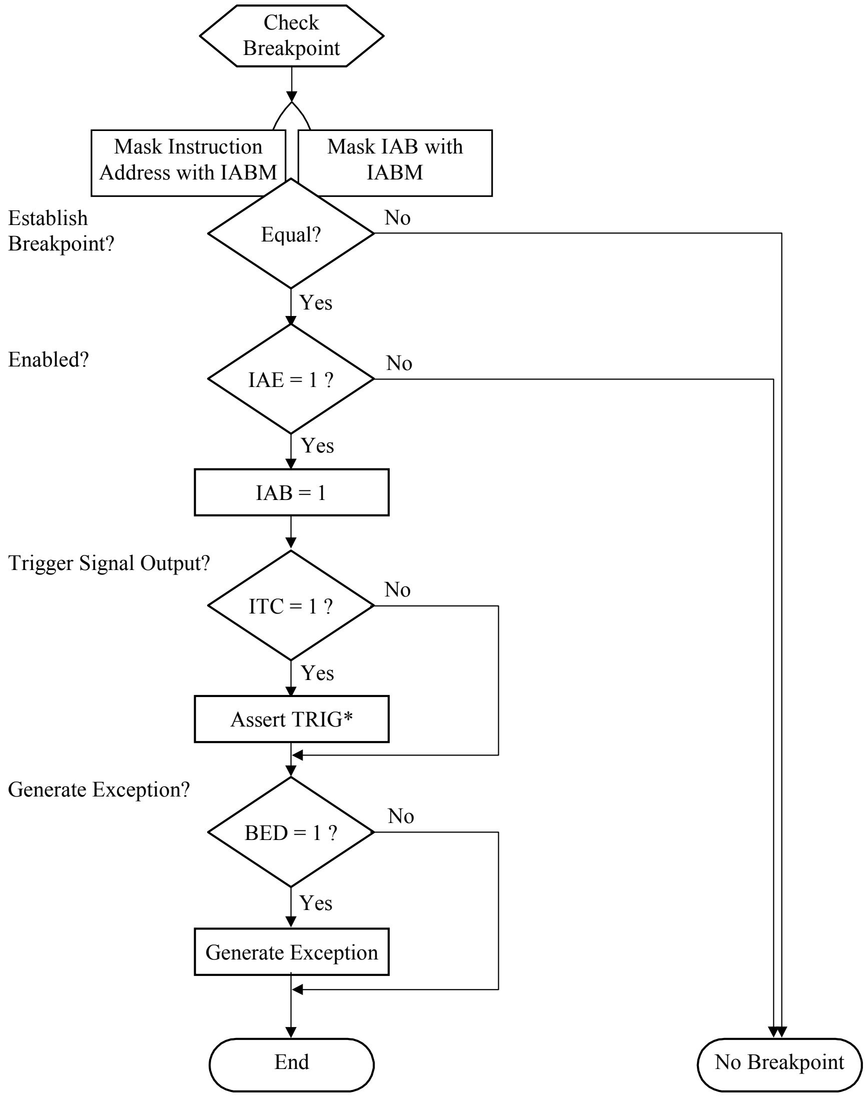


Figure 9-3 Breakpoint Operation (2): Instruction Breakpoint

The following flowchart shows the establishment of a data breakpoint in response to Figure 9-2 and generation of an exception or a trigger signal.

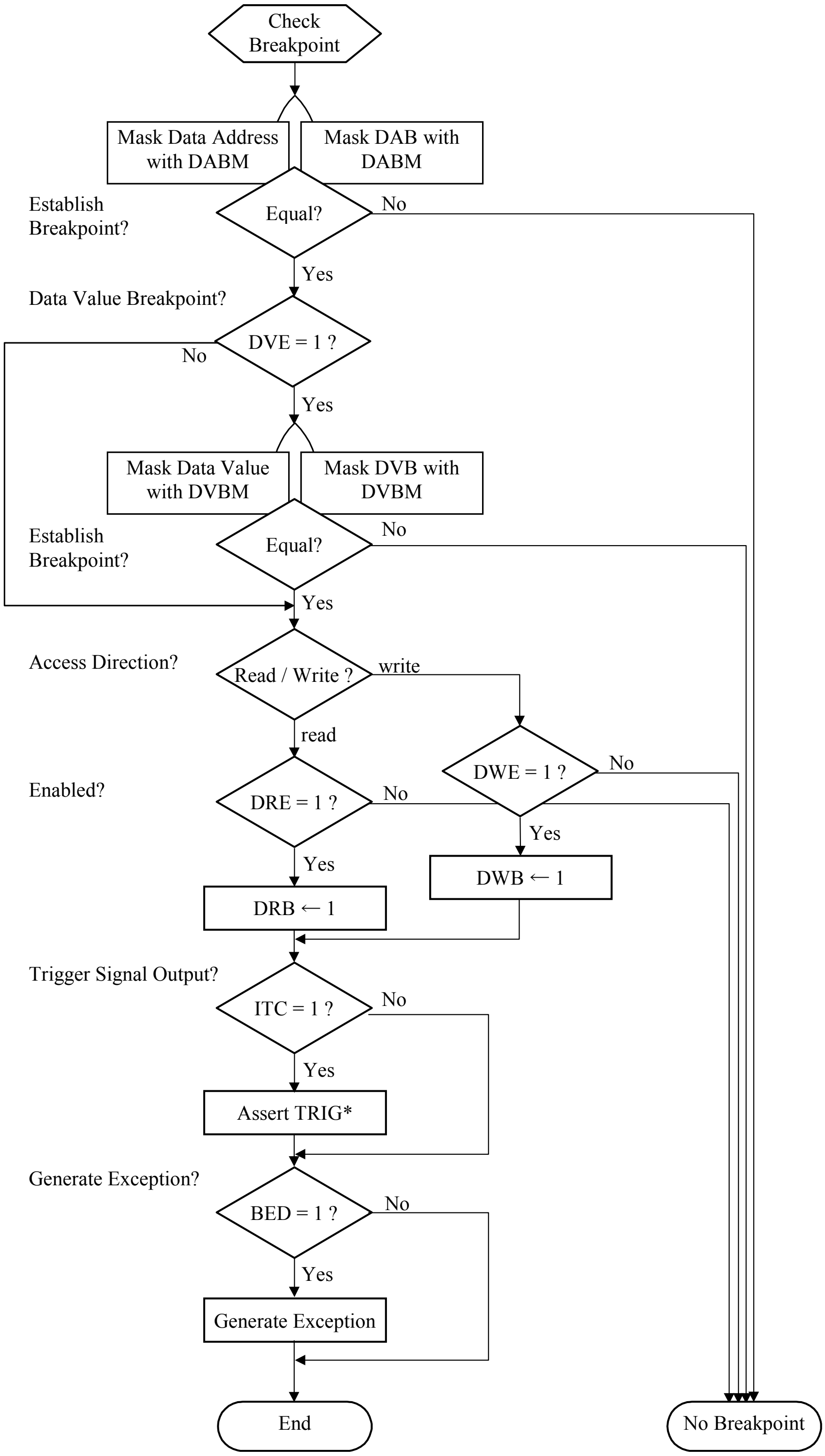


Figure 9-4 Breakpoint Operation (3): Data Breakpoint

9.3. Setting Breakpoints

It is necessary to follow several restrictions to set a breakpoint properly under the pipeline operation. There are two main issues:

- In setting a breakpoint, a group of three or more register values must be changed. Note that this change is made via a pipeline operation, and may create a hazardous area when generating an exception carelessly.
- A load instruction may be completed before data is obtained. The occurrence of a breakpointing event may be behind the instruction completion. This makes generated breakpoints imprecise. In addition, recognition of a debugging exception will be delayed until the processor returns from the level 2 exception handler in the case where another level 2 exception is generated immediately after the instruction in which a data value breakpoint is established.

These restrictions are described below, using sample programs.

9.3.1. Procedure for Setting Breakpoints

It is necessary to control the on/off settings of the breakpoint functions properly to avoid establishing a breakpoint under unsatisfactory conditions while a breakpoint setting is being changed. One easy way is to change the processor mode into Level 2 to mask debugging exceptions temporarily. However, there is a side effect: the user segment becomes unmapped. Therefore, the following sections describe procedures for setting breakpoints according to the steps below without changing the processor mode.

(1) Synchronize the pipeline
(2) Disable the breakpoint that is going to be set
(3) Synchronize the pipeline
(4) Set the breakpoint register pairs
(5) Set the breakpoint control register (enabling the specified breakpoints)
(6) Synchronize the pipeline

Step (1) is to ensure that there is no pending debugging exception at this point. This is to avoid inconsistency in the debugging exception handler encountered when a debugging exception originating from a preceding instruction occurs at the start of changing the breakpoint setting. This synchronized operation uses a SYNC.P instruction for setting an instruction breakpoint and a SYNC.L instruction for setting a data breakpoint.
Step (3) is a process to wait until the breakpoint control register value change made by disabling the breakpoint in Step (2) is written. A SYNC.P instruction is used.
Breakpoint conditions are specified to the two registers in Step (4), and then the breakpoint is enabled in Step (5). These two stages of operations are always written to the registers in this order, so a breakpoint is never established in unsatisfactory conditions. Step (6) is to guarantee that the process moves to the next stage after the breakpoint set in (4) and (5) is enabled. A SYNC.P instruction is used.

9.3.2. Setting an Instruction Breakpoint

The following sample code sets a breakpoint which generates a debugging exception when performing a program within the range from 0x1234_5600 to 0x1234_56ff in user mode or supervisor mode.

```
# -----------------------------------------------------------------------------------------------------------------------
#
# (1) Separation from the breakpoint originating from the preceding instructions
sync.p

# (2) Prohibition of instruction breakpoint
# (Settings related to data breakpoint are to be stored.)
mfbpc $4   # Acquisition of the breakpoint control register value
bgez $4, 1f # Skip (2) if the IAE bit is 0. It has already been prohibited.
nop         # (bds)
li $5, (1 << 31)        # The IAE bit is …
xor $4, $5, $4          # … set to 0, then …
mtbpc $4   # … the breakpoint control register is written.

# (3) Synchronization for guaranteeing that breakpoints has been prohibited.
sync.p

1:

# (4) Setting breakpoint conditions
li $4, 0x12345678     # A breakpoint address is …
mtiab $4    # … specified to the IAB register (the lower 8 bits are arbitrary because they are to be masked).

li $5, 0xffffff00         # Masking is …
mtiabm $5 # … specified to the IABM register.

# (5) Setting the breakpoint control register
mfbpc $4   # Obtains the current value, and …
li $5, ~(                \
    ( 1 << 26 )          # IUE  \
   | ( 1 << 25 )        # ISE  \
   | ( 1 << 24 )        # IKE  \
   | ( 1 << 23 )        # IXE  \
   | ( 1 << 17 )        # ITE  \
   | ( 1 <<  0 )        # IAB  \
  )
and $4, $4, $5          # … clears the flags related to the instruction breakpoint, then …
li $6,                   \
  (                      \
    ( 1 << 31 )         # IAE = 1:  instruction breakpoint is enabled \
   | ( 1 << 26 )        # IUE = 1: enabled in user mode \
   | ( 1 << 20 )        # IUE = 1: enabled in supervisor mode \
   | ( 1 << 15 )        # BED = 1: debugging exception occurs \
  )
or $5, $4, $6
mtbpc $5   # … makes settings again.

# (6) Synchronization for guaranteeing that breakpoint has been set.
sync.p
#
# -----------------------------------------------------------------------------------------------------------------------
```

9.3.3. Setting a Data Address Breakpoint

The following sample code sets a breakpoint which generates a debugging exception when reading/writing an address from 0x1230_0000 to 0x1233_ffff in kernel mode or while executing a Level 1 exception handler.

```
# -------------------------------------------------------------------------
#
# (1) Separation from the debug exception originating from the preceding instructions
sync.l

# (2) Prohibition of data breakpoint
mfbpc $4   # Acquisition of the breakpoint control register value
li $5, ~(                \
    ( 1 << 30 )          # DRE  \
  | ( 1 << 29 )          # DWE  \
  | ( 1 << 28 )          # DVE  \
  | ( 1 << 21 )          # DUE  \
  | ( 1 << 20 )          # DSE  \
  | ( 1 << 19 )          # DKE  \
  | ( 1 << 18 )          # DXE  \
  | ( 1 << 16 )          # DTE  \
  | ( 1 << 2 )           # DWB  \
  | ( 1 << 1 )           # DRB  \
  )
and $4, $4, $5           # Clears all the flags related to the data breakpoint, then …
mtbpc $4   # … makes settings again.

# (3) Synchronization for guaranteeing that data breakpoint has been prohibited.
sync.p

# (4) Setting breakpoint registers
li $6, 0x12305678
mtdab  $6  # Specifies the address to the DAB register (the lower 18 bits are arbitrary because they are to be masked).

li $5, 0xfffc0000
mtdabm $5                #  Masking is specified to the DABM register.

# (5) Setting the breakpoint control register.
li $6,                   \
  (                      \
    ( 1 << 30 )          # DRE = 1: enabled when reading \
  | ( 1 << 29 )          # DWE = 1: enabled when writing \
  | ( 1 << 19 )          # DKE = 1: enabled in kernel mode \
  | ( 1 << 18 )          # DXE = 1: enabled in level 1 exception handler \
  | ( 1 << 15 )          # BED = 1: debugging exception occurs \
  )
or $5, $4, $6
mtbpc $5

# (6) Synchronization for guaranteeing that breakpoint has been set.
sync.p
# -------------------------------------------------------------------------
```

9.3.4. Setting a Data Value Breakpoint

The following sample code sets a breakpoint which generates a debugging exception when reading data containing 0xcafe in the lower 16 bits in an address from 0x1230_0000 to 0x1233_ffff in kernel mode or while executing a Level 1 exception handler.

```
# --------------------------------------------------------------------------
#
# (1) Separation from the debug exception originating from the preceding instructions
sync.l

# (2) Prohibition of data breakpoint
mfbpc $4   # Acquisition of the breakpoint control register value
li $5, ~(                \
    ( 1 << 30 )          # DRE  \
   | ( 1 << 29 )         # DWE  \
   | ( 1 << 28 )         # DVE  \
   | ( 1 << 21 )         # DUE  \
   | ( 1 << 20 )         # DSE  \
   | ( 1 << 19 )         # DKE  \
   | ( 1 << 18 )         # DXE  \
   | ( 1 << 16 )         # DTE  \
   | ( 1 << 2 )          # DWB  \
   | ( 1 << 1 )          # DRB  \
   )
and $4, $4, $5           # Clears all the flags related to the data breakpoint, then …
mtbpc $4   # … makes settings again.

# (3) Synchronization for guaranteeing that data breakpoint has been prohibited.
sync.p

# (4) Setting breakpoint registers
li $6, 0x12305678
mtdab  $6  # Specifies the address to the DAB register (the lower 18 bits are arbitrary because they are to be masked).

li $5, 0xfffc0000
mtdabm $5                # Masking is specified to the DABM register.

li $6, 0xbabecafe
mtdvb $6  # Specifies the data value to the DVB register (the upper 16 bits are arbitrary because they are to be masked).

li $5, 0x0000ffff
mtdvbm $5                # Masking is specified to the DVBM register.

# (5) Setting the breakpoint control register.
li $6,                   \
   (                     \
    ( 1 << 30 )          # DRE = 1: enabled when reading \
   | ( 1 << 28 )         # DVE = 1: determines the data value as well \
   | ( 1 << 19 )         # DKE = 1: enabled in kernel mode \
   | ( 1 << 18 )         # DXE = 1: enabled in level 1 exception handler \
   | ( 1 << 15 )         # BED = 1: debugging exception occurs \
   )
or $5, $4, $6
mtbpc $5
```

```
# (6) Synchronization for guaranteeing that breakpoint has been set.
sync.p
# --------------------------------------------------------------------------
```

A data value breakpoint can be set by adding conditions to a data address breakpoint. By setting the DABM register to 0x00000000, the entire address can be masked, and a breakpoint can be established only with the data values to be read/written as conditions.

9.4. Outputting Trigger Signals

A TRIG* signal can be asserted to make the establishment of a breakpoint visible outside of the processor. Setting the ITE bit of the breakpoint control register (BPC. ITE) to 1 causes a TRIG* signal to be asserted for two bus clocks when an instruction address breakpoint is established. Setting BPC.DTE to 1 causes a TRIG* signal to be asserted in the same manner, when a data breakpoint is established.

A TRIG* signal is directly connected to the breakpoint establishment logic while exceptions (including a debugging exception) always occur with the completion of an instruction. Therefore, the timing of the assertion of a TRIG* signal and the timing of an occurrence of an exception may differ. If the breakpoint established right before the control moves to the Level 2 exception handler results in an imprecise debugging exception, the debugging exception may not be generated although a TRIG* signal has already been asserted, because the debugging exception is held while the handler is in process.

9.5. Notes on Breakpoint Processing

As previously described, imprecise exceptions are sometimes generated due to the establishment of breakpoints. Please note the following issues regarding breakpoint processing.

Consistency in ASID

ASID is not addressed in a breakpoint operation. Therefore, it is necessary for software to take care of this when applying breakpoints only to a specific process. It is also necessary to consider the possibility that an imprecise debugging exception (originating from an instruction in the yet-to-be-switched context) may be detected in the middle of or immediately after context switching. This means that the ASID may different values between when the process moves to the exception handler and when the instruction causes the exception.

To avoid this, recognition of a breakpoint (originating from the instructions so far) must be waited for (if there is one), by executing a SYNC.L instruction right before changing the ASID. (Since imprecise debugging exceptions relate to load/store instructions, a SYNC.L instruction is used for synchronization purposes.)

Overlapping with Level 2 Exception

A debugging exception caused by an instruction may be suspended and occur after another Level 2 exception caused by the following instruction has occurred. In such a case, debugging exception handling is performed after the control returns from another Level 2 exception handler.

The debugging exception will always eventually be issued. If a program needs to ensure the order in which exceptions are handled, then the program must check whether or not there is an established breakpoint pending at the start of all Level 2 exception handlers.

In addition, if a Level 2 exception handler does not return to the place where the exception has been detected, the breakpoint conditions must be reset.