MIPS32® M6200 Processor Core Family Programmer's Guide

Document Number: MD01093
Revision 01.00
January 12, 2016

Table of Contents

List of Tables

List of Figures

Chapter 1

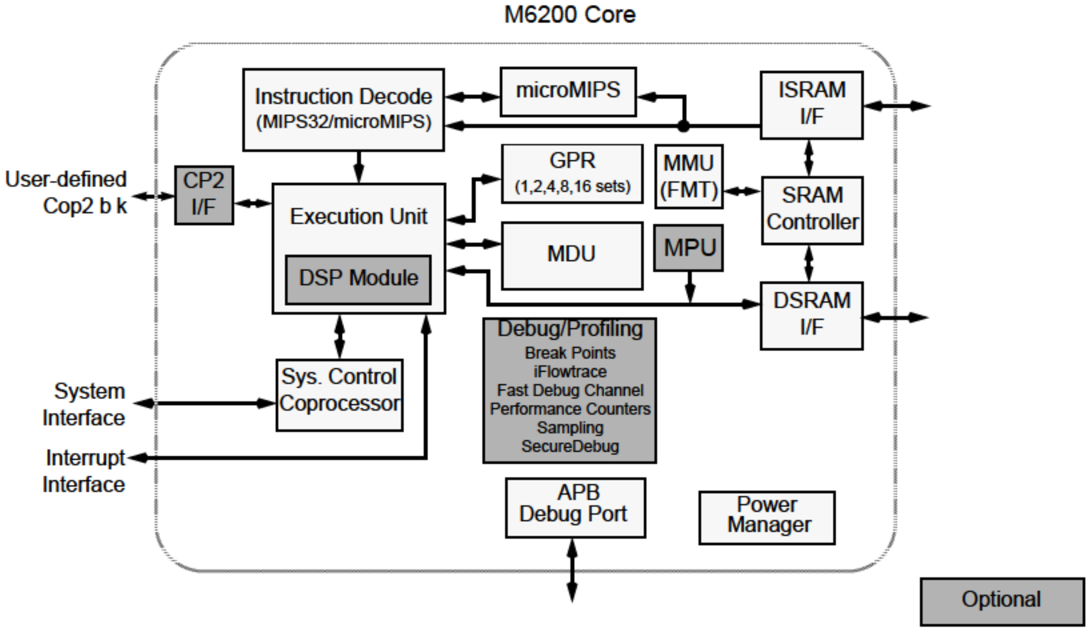
Introduction

The M6200 core is a member of the MIPS32 processor core family.
It is a configurable and synthesizable solution for use in microcontroller and real-time embedded applications requiring a high level of performance efficiency, small silicon area, and low power.

The M6200 processor is a 6-stage pipeline design that implements the MIPS32 Release 6 Architecture. It also includes support for the microMIPS32™ ISA, an Instruction Set Architecture with optimized MIPS32 16-bit and 32-bit instructions that provides a significant reduction in code size with a performance equivalent to MIPS32. The M6200 core is an enhancement of the microAptiv™ UC core, capable of operating up to 30% higher frequency. The M6200 core includes the MIPS Architecture DSP Module Revision 3 as a configurable option, providing a high level of digital signal processing capabilities and Single Instruction Multiple Data (SIMD) support.

Figure 1.1 shows a block diagram of the M6200 core. Optional blocks are shown as shaded.

Figure 1.1 MIPS32® M6200 Core Block Diagram



1.1 Features

- The M6200 builds on the feature set supported by the MIPS32® microAptiv™ UC core as outlined below, but adds the following key capabilities:
  - 6-stage pipeline implementation, enabling up to 30% higher frequencies

  - Support for MIPS32® R6 architecture
  - Support for microMIPS™ R6 ISA
  - 32-bit Address and 64-bit wide Data Bus
  - Data integrity features:
    - Optional ECC on all instruction and data RAM structures
    - Optional parity on data transmission busses

- The M6200 retains features from the microAptiv UC, including:
  - 32-bit General Purpose Registers (GPR)
  - Instruction and Data SRAM interfaces
  - Memory Protection Unit
  - Microcontroller Application Specific Extension (MCU ASE)
  - Multiply/Divide Unit
  - DSP Module (optional)
  - Debug and profiling support
  - Secure Debug
  - Coprocessor interface
  - Power management

- MIPS32 Architecture Features
  - Vectored interrupts and support for external interrupt controller
  - Programmable exception vector base
  - Simple boot exception vector relocation via 2 externally controlled pins
  - GPR shadow registers (1, 2, 4, 8, or 16 additional shadows can be optionally added to minimize latency for interrupt handlers)
  - Bit field manipulation instructions

- microMIPS32 Instruction Set Architecture Release 6
  - microMIPS ISA reduces code size over MIPS32, while maintaining MIPS32 performance.
  - Combining both 16-bit and 32-bit opcodes, microMIPS supports all MIPS32 instructions, with new optimized encoding. Frequently used MIPS32 instructions are available as 16-bit instructions.
  - Stack pointer implicit in instruction.
  - MIPS32 assembly and ABI-compatible.
  - Supports MIPS architecture Modules and ASEs.

- MCU™ ASE

  - Increases the number of interrupt hardware inputs from 6 to 8 for Vectored Interrupt (VI) mode, and from 63 to 255 for External Interrupt Controller (EIC) mode.
  - Separate priority and vector generation. 16-bit vector address is provided.
  - Hardware assist combined with the use of Shadow Register Sets to reduce interrupt latency during an interrupt's prologue and epilogue.
  - An interrupt return with automated interrupt epilogue handling instruction (IRET) improves interrupt latency.
  - Supports optional interrupt chaining.
  - Two memory-to-memory atomic read-modify-write instructions (ASET and ACLR) eases commonly used semaphore manipulation in microcontroller applications. Interrupts are automatically disabled during the operation to maintain coherency.

- Memory Management Unit
  - Simple Fixed Mapping Translation (FMT) mechanism
- ECC Support
  - The ISRAM and DSRAM support optional single error correction and double error detection (SECDED) with correction in software.
- Transmission Parity Support
  - The ISRAM and DSRAM interfaces supports optional parity detection on transactions between master and slave
- Memory Protection Unit
  - Optional feature that improves system security by restricting access, execution, and trace capabilities from untrusted code in predefined memory regions.
- Simple SRAM-Style Interface
  - 32-bit address and 64-bit data
  - Single or multi-cycle latencies
  - Dual instruction and data interfaces
  - Partially registered interface
- MIPS DSP Module (Revision 3.0)
  - Support for MAC operations with four additional pairs of HI/LO accumulator registers (Ac0 - Ac3)
  - Fractional data types (Q15, Q31) with rounding support
  - Saturating arithmetic with overflow handling
  - SIMD instructions operate on 2x16-bit or 4x8-bit operands simultaneously
  - Separate MDU pipeline with full-sized hardware multiplier to support back-to-back operations
  - The DSP Module is build-time configurable.

- Multiply/Divide Unit (without DSP)
    - Maximum issue rate of one 32x16 multiply per clock via on-chip 32x16 hardware multiplier array.
    - Maximum issue rate of one 32x32 multiply every other clock
    - Early-in iterative divide. Minimum 11 and maximum 34 clock latency (dividend (*rs*) sign extension-dependent)
- Multiply/Divide Unit (with DSP configuration)
    - Maximum issue rate of one 32x32 multiply per clock via on-chip 32x32 hardware multiplier array
    - Maximum issue rate of one 32x32 multiply every clock
    - Early-in iterative divide. Minimum 12 and maximum 38 clock latency (dividend (*rs*) sign extension-dependent)
- Multi-Core Support
    - External lock indication enables multi-processor semaphores based on LL/SC instructions
    - External sync indication allows memory ordering
- Coprocessor 2 interface
    - 64b data width interface to an external coprocessor
- Interrupt Controller Unit
    - An optional feature that provides supports for up to 256 interrupts, configurable at build-time in options of 8, 16, 32, 64, 128, and 256 sources
    - Interrupts are configurable as to polarity, level or edge sensitivity, and dual- or single-edge sampling
    - Support for MCU ISA
    - Includes 32-bit Watchdog timer
- Power Control
    - Minimum frequency: 0 MHz
    - Power-down mode (triggered by WAIT instruction)
- Debug/Profiling and iFlowtrace™ Mechanism
    - CPU control with start, stop, and single stepping
    - Virtual instruction and data address/value breakpoints
    - Hardware breakpoint supports both address match and address range triggering
    - Optional simple hardware breakpoints on virtual addresses; 8I/4D or 4I/2D breakpoints, or no breakpoints
    - Optional complex hardware breakpoints with 8I/4D simple breakpoints
    - iFlowtrace support for real-time instruction PC and special events
    - PC and/or load/store address sampling for profiling

  - Performance Counters
  - Support for Fast Debug Channel (FDC)
  - Optional trace conversion block (PDT2ATB) converts iFlowtrace signals to ATB interface signals

- Secure Debug
  - An optional feature that disables access via the APB in an untrusted environment

- Testability
  - Full scan design achieves test coverage in excess of 99% (dependent on library and configuration options)

1.2 Architecture Overview

The M6200 core contains both required and optional blocks, as shown in Figure 1.1. Required blocks must be implemented to remain MIPS-compliant. Optional blocks can be added to the M6200 core based on the needs of the implementation.

The required blocks are as follows:

- Instruction Decode
- Execution Unit
- General Purpose Registers (GPR)
- Multiply/Divide Unit (MDU)
- System Control Coprocessor (CP0)
- Memory Management Unit (MMU)
- I/D SRAM Interfaces
- Power Management

Optional or configurable blocks include:

- DSP (integrated with MDU)
- Memory Protection Unit (MPU)Coprocessor 2 interface
- Interrupt Controller Unit (ICU)
- Debug/Trace/Profiling with optional APB Debug, Hardware Breakpoints, PC Sampling, Performance Counters, Fast Debug Channel, and iFlowtrace

1.3 Pipeline Flow

The M6200 core implements a 6-stage pipeline. The pipeline allows the processor to achieve high frequency while minimizing device complexity, reducing both cost and power consumption.

The M6200 core pipeline consists of six stages:

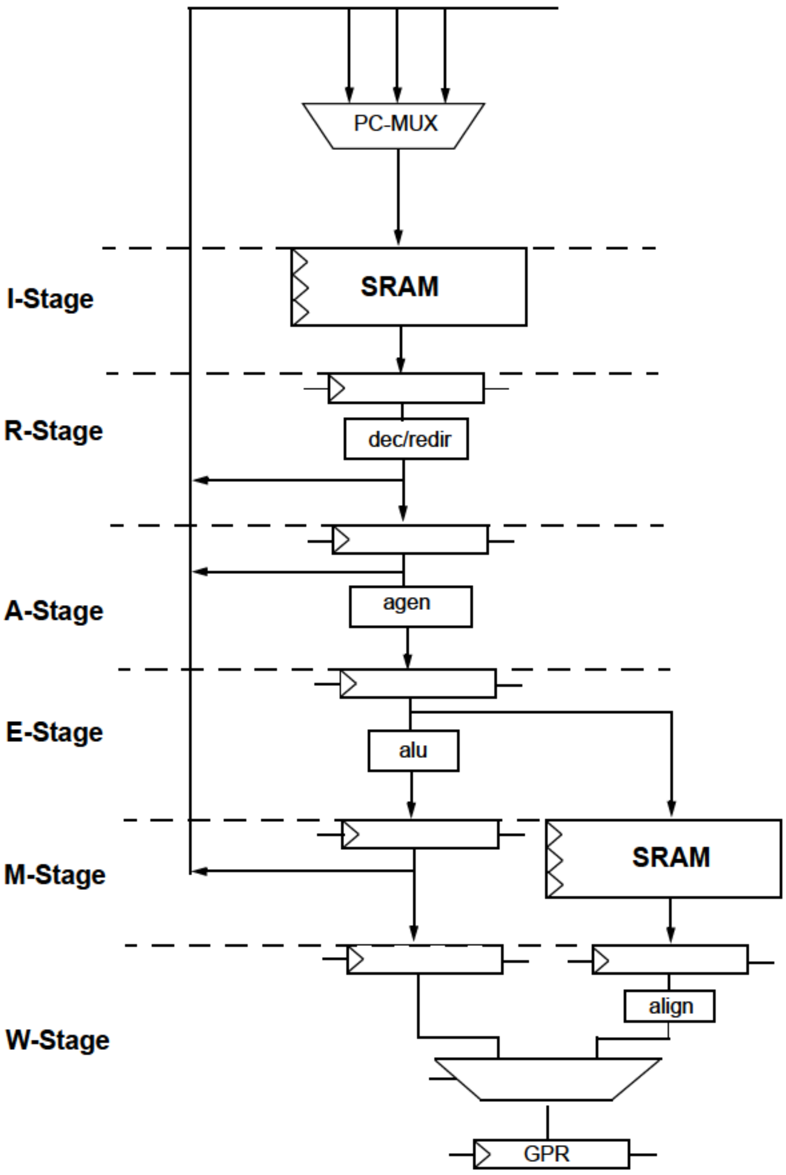
- Instruction (I Stage)

- Register (R Stage)
- Align (A Stage)
- Execution (E Stage)
- Memory (M Stage)
- Write (W stage)

The M6200 core implements a bypass mechanism that allows the result of an operation to be forwarded directly to the instruction that needs it without having to write the result to the register and then read it back.

Figure 1.2 shows a diagram of the M6200 core pipeline .

Figure 1.2 MIPS32® M6200 Core Pipeline



1.3.1 I Stage: Instruction Fetch

- An instruction is fetched from the instructionSRAM.

1.3.2 R Stage: Register File Access

- Instructions are partially decoded.
- Register file is read.
- Jump instructions are decoded. If conditions permit, the M6200 core will improve performance by performing early redirection and fetch the next target instruction before the jump-instruction has graduated.

1.3.3 A Stage: Address Generation

- Data access addresses are calculated.
- Instructions are fully decoded
- Multiplication Booth recode is performed.

1.3.4 E Stage: Execution

- Branch evaluation is performed, causing a redirect if the instruction branches.
- Arithmetic, logic, and multiplication operations are performed. Depending on the size of the operands, the multiplication operation may be double pumped.
- Division uses an iterative sequence, and is non-blocking until an instruction uses the division result.
- Fixed mapping translation is used for cores without TLB support

1.3.5 M Stage: Memory Access

- Data access is performed for load and store instructions.
- On cacheable accesses, store instructions are written to internal write buffers, where the data will be written to external memory as soon as the interface is free or idle.
- On uncacheable accesses, store instructions obeys strongly-ordered memory consistency rules by stalling until the write is the oldest instruction in the pipeline before writing to the external memory.

1.3.6 W Stage: Write

- Exceptions are prioritized and flagged.
- Data reads are aligned before writing to the register file.

1.4 M6200 Required Logic Blocks

The required logic blocks of the M6200 core (Figure 1.1) are defined in the following subsections.

1.4.1 Execution Unit

The M6200 core execution unit implements a load/store architecture with single-cycle ALU operations (logical, shift, add, subtract) and an autonomous multiply/divide unit.

The execution unit includes:

- Single cycle Arithmetic Logic Unit (ALU) for performing arithmetic, bitwise logical operations and branch target calculation.
- Adder for load/store address calculation

- Address unit for calculating the next PC and next fetch address selection muxes.
- Load Aligner.
- Shifter and Store Aligner.
- Branch condition comparator.
- Bypass muxes to advance result between two instructions with data dependency.
- Leading Zero/One detect unit for implementing the CLZ and CLO instructions.
- Read-modify-write control logic implementing atomic instructions defined in the MCU ASE.
- Actual execution of the Atomic Instructions defined in the MCU ASE.
- A separate DSP ALU and Logic block for performing part of DSP Module instructions, such as arithmetic/shift/compare operations when the DSP function is configured.

1.4.2 General Purpose Registers

The M6200 core contains thirty-two 32-bit general-purpose registers used for integer operations and address calculation. Optionally, 1, 2, 4, 8, or 16 additional register file shadow sets (each containing thirty-two registers) can be added to minimize context switching overhead during interrupt/exception processing. The register file is flop-based and is fully bypassed to minimize operation latency in the pipeline.

1.4.3 Multiply/Divide Unit (MDU)

The M6200 core includes a multiply/divide unit (MDU) that contains a separate, dedicated pipeline for integer multiply/divide operations and DSP Module multiply instructions (with DSP option). This pipeline operates in parallel with the integer unit (IU) pipeline and does not stall when the IU pipeline stalls. This allows the long-running MDU operations to be partially masked by system stalls and/or other integer unit instructions.

The MIPS architecture defines that the result of a multiply or divide operation be placed in general-purpose registers (without DSP option) or one of four pairs of *HI* and *LO* registers (with DSP enabled).

1.4.4 MDU with 32x32 DSP Multiplier with DSP Option

With the DSP configuration option enabled, the MDU supports execution of one 16x16, 32x16, or 32x32 multiply or multiply-accumulate operation every clock cycle with the built in 32x32 multiplier array. The multiplier is shared with DSP Module operations.

The MDU also implements various shift instructions operating on the HI/LO register and multiply instructions as defined in the DSP Module. It supports all the data types required for this purpose and includes four extra HI/LO registers as defined by the Module.

Table 1.1 lists the latencies (number of cycles for an instruction to propagate from the beginning to the end of the core's pipeline) and repeat rates (throughput without data dependency) for the DSP multiply and dot-product operations. The approximate latencies and repeat rates are listed in terms of pipeline clocks.

Table 1.1 DSP-related Latencies and Repeat Rates

Opcode	Latency	Repeat Rate
Multiple and dot-product without saturation after accumulation	6	1
Multiple and dot-product with saturation after accumulation (word)	6	1
Multiply and dot-product with saturation after accumulation (doubleword)	7	1
Accumulator shifter uses immediately previous multiply result.	7	2
Multiply without accumulation	6	1

1.4.5 MDU with 32x16 High-Performance Multiplier

Without the DSP option, the high-performance MDU consists of a 32x16 Booth-recoded multiplier, a divide state machine, and the necessary multiplexers and control logic. The first number shown ('32' of 32x16) represents the *rs* operand. The second number ('16' of 32x16) represents the *rt* operand. The M6200 core only checks the value of the *rt* operand to determine how many times the operation must pass through the multiplier. The 16x16 and 32x16 operations pass through the multiplier once. A 32x32 operation passes through the multiplier twice.

The MDU supports execution of one 16x16 or 32x16 multiply or multiply-accumulate operation every clock cycle; 32x32 multiply operations can be issued every other clock cycle. Appropriate interlocks are implemented to stall the issuance of back-to-back 32x32 multiply operations. The multiply operand size is automatically determined by logic built into the MDU.

Table 1.2 and Table 1.3 list the repeat rate (how often the operation can be reissued when there is no data dependency) and latency (number of cycles until a result is available) for the multiply and divide instructions. The approximate latency and repeat rates are listed in terms of pipeline clocks.

Table 1.2 High-Performance Integer Multiply/Divide Unit Latencies and Repeat Rates with DSP

Opcode	Operand Size (mul rt) (div rs)	Latency	Repeat Rate
MUL, MUH, MULU, MUHU (GPR destination)	16 bits	6	1
	32 bits	6	1

Table 1.2 High-Performance Integer Multiply/Divide Unit Latencies and Repeat Rates with DSP

Opcode	Operand Size (mul rt) (div rs)	Latency	Repeat Rate
DIV, MOD/ DIVU, MODU (GPR destination)	4 bits	11/10	8/7
	8 bits	15/14	12/11
	16 bits	23/22	20/19
	24 bits	31/30	28/27
	32 bits	39/38	36/35
MADD/MADDU, MSUB/MSUBU (with DSP)	GPR is 32-bit Accumulator is 64-bit	6	1

Table 1.3 High-Performance Integer Multiply/Divide Unit Latencies and Repeat Rates without DSP

Opcode	Operand Size (mul rt) (div rs)	Latency	Repeat Rate
MUL, MUH, MULU, MUHU (GPR destination)	16 bits	6 (I-R-A-E-M-W)	1
	32 bits	7 (I-R-A-E-E-M-W, 32x16 array is in E)	2
DIV, MOD/DIVU, MODU (GPR destination)	4	11/10	8/7
	8 bits	15/14	12/11
	16 bits	23/22	20/19
	24 bits	31/30	28/27
	32 bits	39/38	36/35

1.4.6 System Control Coprocessor (CP0)

In the MIPS architecture, CP0 is responsible for the virtual-to-physical address translation, the exception control system, the processor's diagnostics capability, the operating modes (kernel, user, and debug), and whether interrupts are enabled or disabled. Configuration information, such as presence of build-time options, such as microMIPS or Coprocessor 2 interface, is also available by accessing the CP0 registers.

Coprocessor 0 also contains the logic for identifying and managing exceptions. Exceptions can be caused by a variety of sources, including boundary cases in data, external events, or program errors.

1.4.7 Interrupt Handling

The M6200 core includes support for eight hardware interrupt pins, two software interrupts, and a timer interrupt. These interrupts can be used in any of three interrupt modes, as defined by Release 2 of the MIPS32 Architecture:

- Interrupt compatibility mode, which acts identically to that in an implementation of Release 1 of the Architecture.
- Vectored Interrupt (VI) mode, which adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt, and to assign a GPR shadow set for use during interrupt processing. The presence of this mode is denoted by the *VInt* bit in the *Config3* register. This mode is architecturally optional; but it is always present on the M6200 core, so the *VInt* bit will always read as a 1 for the M6200 core.
- External Interrupt Controller (EIC) mode, which redefines the way in which interrupts are handled to provide full support for an external interrupt controller handling prioritization and vectoring of interrupts. The presence of this mode is indicated by the *VEIC* bit in the *Config3* register. Again, this mode is architecturally optional. On the M6200 core, the *VEIC* bit is set externally by the static input *SI_EICPresent*, which allows system logic to indicate the presence of an external interrupt controller.

The reset state of the processor is interrupt compatibility mode, such that processors supporting Release 6 of the Architecture (such as the M6200 core) are fully compatible with implementations of Release 1 of the Architecture.

VI or EIC interrupt modes can be combined with the optional shadow registers to specify which shadow set should be used on entry to a particular vector. The shadow registers improve interrupt latency by avoiding the need to save context when invoking an interrupt handler.

In the M6200 core, interrupt latency is reduced by:

- Speculative interrupt-vector prefetching during the pipeline flush.
- Interrupt Automated Prologue (IAP) in hardware: Shadow Register Sets remove the need to save GPRs, and IAP removes the need to save specific Control Registers when handling an interrupt.
- Interrupt Automated Epilogue (IAE) in hardware: Shadow Register Sets remove the need to restore GPRs, and IAE removes the need to restore specific Control Registers when returning from an interrupt.
- Allow interrupt chaining. When servicing an interrupt and interrupt chaining is enabled, there is no need to return from the current Interrupt Service Routine (ISR) if there is another valid interrupt pending to be serviced. The control of the processor can jump directly from the current ISR to the next ISR without IAE and IAP.
- Simple exception vector relocation via the externally controlled pin *SI_Offset*.

1.4.8 GPR Shadow Registers

The MIPS32 Architecture optionally removes the need to save and restore GPRs on entry to high-priority interrupts or exceptions, and to provide specified processor modes with the same capability. This is done by introducing multiple copies of the GPRs, called *shadow sets*, and allowing privileged software to associate a shadow set with entry to kernel mode via an interrupt vector or exception. The normal GPRs are logically considered shadow set zero.

The number of GPR shadow sets is a build-time option. The M6200 core allows 1 (the normal GPRs), 2, 4, 8, or 16 shadow sets. The highest number actually implemented is indicated by the $SRSCtl_{HSS}$ field. If this field is zero, only the normal GPRs are implemented.

Shadow sets are new copies of the GPRs that can be substituted for the normal GPRs on entry to kernel mode via an interrupt or exception. Once a shadow set is bound to a kernel-mode entry condition, references to GPRs operate exactly as one would expect, but they are redirected to registers that are dedicated to that condition. Privileged software may need to reference all GPRs in the register file, even specific shadow registers that are not visible in the current mode, and the RDPGPR and WRPGPR instructions are used for this purpose. The *CSS* field of the *SRSCtl* register provides the number of the current shadow register set, and the *PSS* field of the *SRSCtl* register provides the number of the previous shadow register set that was current before the last exception or interrupt occurred.

If the processor is operating in VI interrupt mode, binding of a vectored interrupt to a shadow set is done by writing to the *SRSMap* register. If the processor is operating in EIC interrupt mode, the binding of the interrupt to a specific shadow set is provided by the external interrupt controller and is configured in an implementation-dependent way. Binding of an exception or non-vectored interrupt to a shadow set is done by writing to the *ESS* field of the *SRSCtl* register. When an exception or interrupt occurs, the value of $SRSCtl_{CSS}$ is copied to $SRSCtl_{PSS}$, and $SRSCtl_{CSS}$ is set to the value taken from the appropriate source. On an ERET, the value of $SRSCtl_{PSS}$ is copied back into $SRSCtl_{CSS}$ to restore the shadow set of the mode to which control returns.

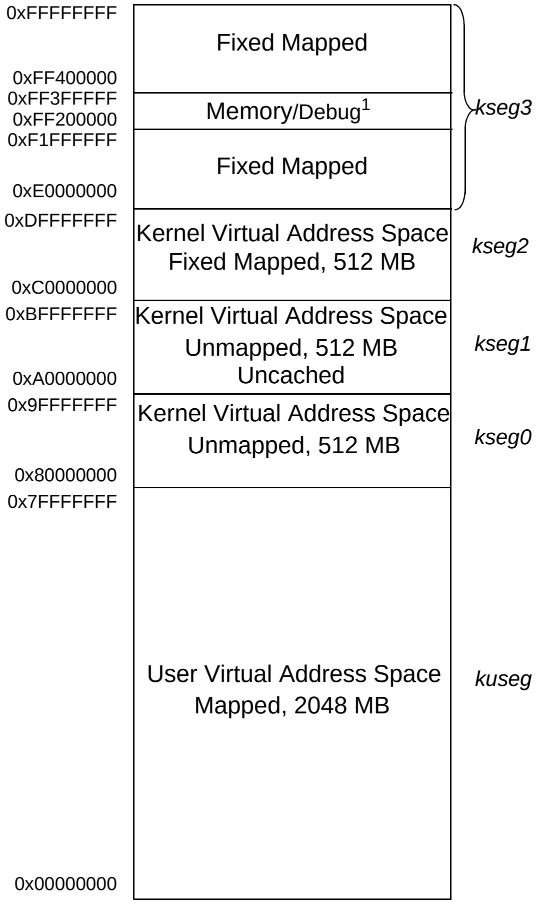
1.4.9 Modes of Operation

The M6200 core implements three modes of operation:

- *User mode* is most often used for applications programs.
- *Kernel mode* is typically used for handling exceptions and operating-system kernel functions, including CP0 management and I/O device accesses.
- *Debug mode* is used during system bring-up and software development. Refer to the Chapter 8, “Debug Support in the M6200 Core” on page 152 for more information on debug mode.

Figure 1.3 shows the virtual address map of the MIPS Architecture.

Figure 1.3 M6200 Core Virtual Address Map



1. This space is mapped to memory in user or kernel mode, and by the Debug module in debug mode.

1.4.10 Memory Management Unit (MMU)

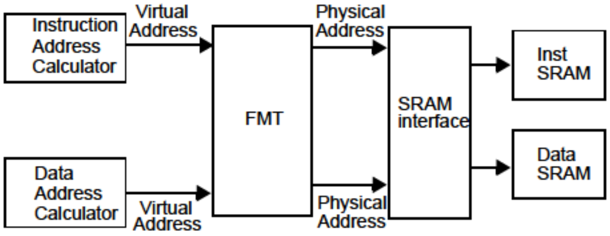
The M6200 core contains a simple Fixed Mapping Translation (FMT) MMU that interfaces between the execution unit and the SRAM Controller.

1.4.11 Fixed Mapping Translation (FMT)

A FMT is smaller and simpler than the full Translation Lookaside Buffer (TLB) style MMU found in other MIPS cores. Like a TLB, the FMT performs virtual-to-physical address translation and provides attributes for the different segments. Those segments that are unmapped in a TLB implementation (kseg0 and kseg1) are translated identically by the FMT.

Figure 1.4 shows how the FMT is implemented in the M6200 core.

Figure 1.4 Address Translation During SRAM Access with FMT Implementation



1.4.12 SRAM Interface Controller

The M6200 core contains an interface to SRAM-style memories that is lightly-coupled to the core (an unregistered interface). This permits deterministic response time with less area than is typically required for caches. The SRAM interface includes separate uni-directional 32-bit busses for address, and 64-bit wide buses for read data and write data. All writes are non-posted.

The SRAM interface is a dual interface that enables independent connection to instruction and data devices. It generally yields the highest performance, because the pipeline can generate simultaneous I and D requests, which are then serviced in parallel.

1.4.12.1 Back-stalling

Typically, read and write transactions will complete in a single cycle. However, if multi-cycle latency is desired, the interface can be stalled to allow connection to slower devices.

1.4.12.2 Lock Mechanism

The SRAM interface includes two protocols to identify a locked sequence. One is used in conjunction with the LL/SC atomic read-modify-write semaphore instructions, and the other is used in conjunction with the MCU ASE Atomic instructions, namely, ASET and ACLR. In the former, the interface includes a protocol that externalizes the execution of the SYNC instruction. External logic may choose to use this information to enforce memory ordering between various elements in the system.

1.4.12.3 Sync Mechanism

The interface includes a protocol that externalizes the execution of the SYNC instruction. External logic might choose to use this information to enforce memory ordering between various elements in the system.

1.4.13 Hardware Reset

The M6200 core has two reset input signals: *SI_WarmResetN* and *SI_ColdResetN*. These two signals are used to initialize critical hardware state.

Both reset signals are active Low and can be asserted synchronously or asynchronously to the core clock, *SI_ClkIn*, and will trigger a Reset exception. The CPU will internally generate a reset pulse of 6 clock cycles, whereupon the core will exit reset synchronously.

The primary difference between the two reset signals is that *SI_WarmResetN* allows the core to enter debug mode at the end of reset through the DbgBOOT mechanism. With *SI_ColdResetN*, that bootmode is not allowed, and the core will begin fetching code in Kernel mode at the end of reset. The reset behavior is summarized in Table 1.4.

Table 1.4 Reset Types

SI_WarmResetN	SI_ColdResetN	Action
1	1	Normal operation, no reset.
0	1	Reset exception; sets *Status*$_{SR}$ bit.
X	0	Reset exception.

One or both of the reset signals must be asserted at power-on or whenever hardware initialization of the core is desired. A cold reset typically occurs when the machine is first turned on. A warm reset usually occurs when the machine is already on, and the system is rebooted. *SI_WarmResetN* sets a bit in the Status register; this bit could be used by software to distinguish between the two reset signals, if desired.

1.4.14 Power Management

The M6200 core offers a number of power management features, including low-power design, active power management, and power-down modes of operation. The core is a static design that supports slowing or halting the clocks, which reduces system power consumption during idle periods.

The M6200 core provides two mechanisms for system-level low-power support:

- Register-controlled power management
- Instruction-controlled power management

1.4.14.1 Register-Controlled Power Management

The power-management function is supported by three bits,*Status*$_{EXL}$, *Status*$_{ERL}$, and *Debug*$_{DM}$ support the power-management function, which allows the user to change the power state if an exception or error occurs while the core is in a low-power state. Depending on what type of exception is taken, one of these three bits will be asserted and reflected on the *SI_EXL*, *SI_ERL,* or *EJ_DebugM* outputs. The external agent can look at these signals and determine whether or not to leave the low-power state to service the exception.

Three bits, *Status*$_{EXL}$, *Status*$_{ERL}$, and *Debug*$_{DM}$, support the power-management function by allowing the user to change the power state if an exception or error occurs while the core is in a low-power state. Depending on what type of exception is taken, one of these three bits will be asserted and reflected on the *SI_EXL*, *SI_ERL,* or *EJ_DebugM* outputs. The external agent can look at these signals and determine whether to leave the low-power state to service the exception.

The following four power-down signals are part of the system interface and change state as the corresponding bits in the CP0 registers are set or cleared:

- The *SI_EXL* signal represents the state of the *EXL* bit (1) in the CP0 *Status* register.
- The *SI_ERL* signal represents the state of the *ERL* bit (2) in the CP0 *Status* register.
- The *EJ_DebugM* signal represents the state of the DM bit (30) in the CP0 *Debug* register.

1.4.14.2 Instruction-Controlled Power Management

The second mechanism for invoking power-down mode is by executing the WAIT instruction. When the WAIT instruction is executed, the internal clock is suspended; however, the internal timer and some of the input pins (*SI_Int[7:0]*, *SI_NMI*, *SI_WarmResetN*, and *SI_ColdResetN*) continue to run. Once the CPU is in instruction-controlled power management mode, any interrupt, NMI, or reset condition causes the CPU to exit this mode and resume normal operation.

The M6200 core asserts the *SI_Sleep* signal, which is part of the system interface bus, whenever the WAIT instruction is executed. The assertion of *SI_Sleep* indicates that the clock has stopped and the M6200 core is waiting for an interrupt.

1.5 Optional or Configurable Logic Blocks

The M6200 core contains several optional or configurable logic blocks, shown as shaded in the block diagram in Figure 1.1.

1.5.1 Data Integrity

The M6200 core optionally supports single-error correction and double-error detection (SECDED). Errors are reported in CP0 registers and rely on software correction. ECC is generated or checked for valid byte lanes.

In addition to ECC protection, the M6200 core also offers data integrity protection on uncached data transmissions. Parity is generated and checked for every 8 bits of data transferred, and for every 32 bits of address transferred.

1.5.2 Memory Protection Unit

The Memory Protection Unit can be configured to have from 1 to 16 memory protection regions. Each region is enabled by registers that define the address, size, and protection of each memory region. The Memory Protection Unit control registers are implemented by CDMM (Common Device Memory Map) registers. After they have been programmed, these control registers can be locked to prohibit later modifications. Once programmed, a Protection Exception will be triggered when an Instruction Fetch or Data Access matches the address of the protected memory region or any modification of the *EBase* (base address of exception vectors) register was attempted. Each protected region can also disable the iFlowtrace capability. Typically, the Memory Protection Unit improves system security by disabling access to bootcode and preventing execution of non-trusted kernel mode code.

1.5.3 DSP Module

The M6200 core implements an optional DSP Module to benefit a wide range of DSP, Media, and DSP-like algorithms. The DSP module is highly integrated with the Execution Unit and the MDU in order to share common logic and to include support for operations on fractional data types, saturating arithmetic, and register SIMD operations. Fractional data types Q15 and Q31 are supported. Register SIMD operations can perform up to four simultaneous add, subtract, or shift operations and two simultaneous multiply operations.

In addition, the DSP Module includes some key features that efficiently address specific problems often encountered in DSP applications. These include, for example, support for complex multiply, variable-bit insert and extract, and implementation and use of virtual circular buffers. The extension also makes available four additional sets of HI-LO accumulators to better facilitate common accumulate functions such as filter operation and convolutions.

1.5.4 Coprocessor 2 Interface

The M6200 core can be configured to have an interface for an on-chip coprocessor. This coprocessor can be tightly coupled to the processor core, allowing high-performance solutions integrating a graphics accelerator or DSP, for example.

The coprocessor interface is extensible and standardized on MIPS cores, allowing for design reuse. The M6200 core supports a subset of the full coprocessor interface standard: 64b data transfer, no Coprocessor 1 support, and single issue in-order data transfer to coprocessor.

The coprocessor interface is designed to ease integration with customer IP. The interface allows high-performance communication between the core and coprocessor. There are no late or critical signals on the interface.

1.5.5 Debug Support

The M6200 core provides for an optional Debug interface via the MIPS Debug Hub (MDH) and Advanced Peripheral Bus (APB) interface. MDH provides the capability for connection to a JTAG or APB compatible debug system for improved debug performance and support for multi-core systems. For more information, refer to the *MIPS® Debug Hub Technical Reference Manual* [6].

The M6200 core also provides a special Debug mode of operation, in addition to the standard User mode and Kernel modes of operation. Debug mode is entered after a debug exception is taken and continues until a debug exception return (DERET) instruction is executed. During this time, the processor executes the debug exception handler routine.

1.5.6 Interrupt Controller Unit (ICU)

This release includes the Interrupt Controller Unit (ICU). The ICU is a configurable IP that supports the following features:

- Accepts up to 256 interrupt sources configurable at build-time in options of 8, 16, 32, 64, 128 and 256 sources.
- Supports MCU ASE where output drives a requested interrupt priority level.
- Distributes (hardwired) interrupt sources to the core.
- Backward compatibility with pre-defined MIPS Technologies interrupt modes, configurable by software.
- Supports interrupt source sensitivity (level-positive, level-negative, edge-positive, edge-negative, dual-edge-sensitive) at build-time. All sources are normalized to positive, level-sensitive signals.
- Interrupt Pending mask feature
- Interrupt sources are mapped to *SI_Int[7:0]* or NMI. Mapping is software-controlled, and control registers are extended to reflect the widening of the *SI_Int* output bus.
- A single 32-bit watch-dog timer
- Supports EIC Shadow set use in EIC interrupt mode capability. Shadow set values are hardwired (a build-time option)

The ICU is described in detail in the *MIPS® Interrupt Controller User's Guide* [17].

1.6 Testability

Testability for production testing of the core is supported through the use of internal scan and memory BIST.

1.6.1 Internal Scan

Full mux-based scan for maximum test coverage is supported, with a configurable number of scan chains. ATPG test coverage can exceed 99%, depending on standard cell libraries and configuration options.

1.6.2 User-specified Memory BIST

Memory BIST can be inserted with a CAD tool or other user-specified method. Wrapper modules and special side-band signal buses of configurable width are provided within the core to facilitate this approach.

1.7 Build-Time Configuration Options

The M6200 core allows a number of features to be customized based on the intended application. Table 1.5 summarizes the key configuration options that can be selected when the core is synthesized and implemented.

For a core that has already been built, software can determine the value of many of these options by checking an appropriate register field. Refer to the *MIPS32® M6200 Processor Core Family Programmer's Guide* for a more complete description of these fields. The value of some options that do not have a functional effect on the core are not visible to software.

Table 1.5 CPU Build-time Configuration Options

Feature	Options	Software Visibility
Integer register file sets	1, 2, 4, 8 or 16	*SRSCtl*$_{HSS}$
DSP Module	Present or not	*Config3*$_{DSPP}$ and *Config3*$_{DSP2P}$
Memory Protection Unit	Present or not. If present 1 - 16 regions	N/A
Debug Controller via APB Port	Present or not	N/A
Fast Debug Channel (FDC)	Present or not	DCR$_{\mathrm{FDCI}}$
Instruction/data hardware breakpoints	0/0, 4/2, or 8/4	DCR$_{\mathrm{InstBrk}}$, IBS$_{\mathrm{BCN}}$ DCR$_{\mathrm{DataBrk}}$, DBS$_{\mathrm{BCN}}$
Hardware breakpoint trigger	By address match, or address match and address range	BCn$_{\mathrm{hwart}}$ and DBCn$_{\mathrm{hwart}}$
Complex breakpoints	0/0 or 8/4	DCR$_{\mathrm{CBT}}$
Performance Counters	Present or not	*Config1*$_{PC}$
iFlowtrace hardware	Present or not	*Config3*$_{ITL}$
iFlowtrace on-chip trace memory size	256B - 8MB	ITCBRDP
* These bits indicate the presence of an external block. Bits will not be set if interface is present, but block is not.		

Table 1.5 CPU Build-time Configuration Options (Continued)

Feature	Options	Software Visibility
iFlowtrace off-chip PIB	Present or not	$\mathrm{IFCTL}_{\mathrm{OfC}}$
Coprocessor2 interface	Present or not	$\mathit{Config1}_{C2*}$
Interrupt Controller (ICU)	Present or not. If present, 8, 16, 32, 64, 128, 256 interrupts, interrupt polarity and edge/level sensitivity, and mapped shadow set value.	N/A
SRAM ECC	Present or not	ErrCtl_{EE}
Parity on data and address bus	Present or not	ErrCtl_{PE}
Interrupt synchronizers	Present or not	N/A
Interrupt Vector Offset	Compute from Vector Input or Immediate Offset	N/A
* These bits indicate the presence of an external block. Bits will not be set if interface is present, but block is not.		

Chapter 2

The MIPS® DSP Module

The M6200 includes support for the MIPS DSP Module Revision 3 that provides enhanced performance capabilities for a wide range of signal-processing applications, with computational support for fractional data types, SIMD, saturation, and other operations that are commonly used in these applications.

Refer to *MIPS® Architecture For Programmers Volume IV-e* [15] *or* [15] for a general description of the DSP Module and detailed descriptions of the DSP instructions. Additional programming information is contained in *Five Methods of Utilizing the MIPS® DSP Module* [18], and *Efficient DSP Module Programming in C: Tips* and *Tricks* [18].

2.1 Additional Register State for the DSP Module

The DSP Module defines four accumulator registers and one additional control/status register, as described below. These registers require the operating system to recognize the presence of the DSP Module and to include these additional registers in the context save and restore operations.

2.1.1 HI/LO Registers

The DSP Module includes four HI/LO accumulator registe1fcr pairs (ac0, ac1, ac2, and ac3). These registers improve the parallelization of independent accumulation routines—for example, filter operations, convolutions, etc. DSP instructions that target the accumulators use two instruction bits to specify the destination accumulator.

2.1.2 DSPControl Register

The *DSPControl* register contains control and status information used by DSP instructions. Figure 2.1 illustrates the bits in this register, and Table 2.1 describes their usage.

Figure 2.1 MIPS32® DSP Module Control Register (DSPControl) Format

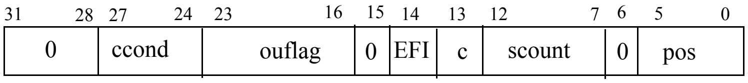


Table 2.1 MIPS® DSP Module Control Register (DSPControl) Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
0	31:28	Reserved. Used in the MIPS64 architecture but not used in the MIPS32 architecture. Must be written as zero; returns zero on read.	0	0	Required

Table 2.1 MIPS® DSP Module Control Register (DSPControl) Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
ccond	27:24	Condition code bits set by compare instructions. The compare instruction sets the right-most bits as required by the number of elements in the vector compare. Bits not set by the instruction remain unchanged.	R/W	0	Required
ouflag	23:16	This field is written by hardware when certain instructions overflow or underflow and may have been saturated. See Table 2.2 for a full list of which bits are set by what instructions.	R/W	0	Required
EFI	14	Extract Fail Indicator. This bit is set to 1 when an EXTP, EXTPV, EXTPDP, or EXTPDP instruction fails. These instructions fail when there are insufficient bits to extract, that is, when the value of pos in *DSPControl* is less than the value of size specified in the instruction. This bit is not sticky, so each invocation of one of the four instructions will reset the bit depending on whether or not the instruction failed.	R/W	0	Required
c	13	Carry bit. This bit is set and used by special add instructions that implement a 64-bit add across two GPRs. The ADDSC instruction sets the bit and the ADDWC instruction uses this bit.	R/W	0	Required
scount	12:7	This field is for use by the INSV instruction. The value of this field is used to specify the size of the bit field to be inserted.	R/W	0	Required
pos	5:0	This field is used by the variable insert instructions INSV to specify the insert position. It is also used to indicate the extract position for the EXTP, EXTPV, EXTPDP, and EXTPDPV instructions. The decrement pos (DP) variants of these instructions on completion will have decremented the value of pos (by the size amount). The MTHLIP instruction will increment the pos value by 32 after copying the value of *LO* to *HI*.	R/W	0	Required
0	15:13	Must be written as zero; returns zero on read.	0	0	Reserved

The bits of the overflow flag ouflag field in the *DSPControl* register are set by a number of instructions, as described in Table 2.2. These bits are sticky and can be reset only by an explicit write to these bits in the register (using the WRDSP instruction).

Table 2.2 DSPControl ouflag Bits

Bit Number	Description
16	This bit is set when the destination is accumulator (*HI-LO* pair) zero, and an operation overflow or underflow occurs. These instructions are: DPAQ_S, DPAQ_SA, DPSQ_S, DPSQ_SA, DPAQX_S, DPAQX_SA, DPSQX_S, DPSQX_SA, MAQ_S, MAQ_SA and MULSAQ_S.
17	Same instructions as above, when the destination is accumulator (*HI-LO* pair) one.

Table 2.2 DSPControl (Continued) ouflag Bits

Bit Number	Description
18	Same instructions as above, when the destination is accumulator (*HI-LO* pair) two.
19	Same instructions as above, when the destination is accumulator (*HI-LO* pair) three.
20	Instructions that set this bit on an overflow/underflow: ABSQ_S, ADDQ, ADDQ_S, ADDU, ADDU_S, ADDWC, SUBQ, SUBQ_S, SUBU and SUBU_S.
21	Instructions that set this bit on an overflow/underflow: MUL, MUL_S, MULEQ_S, MULEU_S, MULQ_RS, and MULQ_S.
22	Instructions that set this bit on an overflow/underflow: PRECRQ_RS, SHLL, SHLL_S, SHLLV, and SHLLV_S.
23	Instructions that set this bit on an overflow/underflow: EXTR, EXTR_S, EXTR_RS, EXTRV, and EXTRV_RS.

2.2 Software Detection of the DSP Module

The presence of the MIPS DSP Module Revision 3 in the M6200 core is indicated by five static CP0 register bits. In the *Config* register, the three *AR* (*Architecture Revision*) bits indicate the implementation of Revision 6 of the MIPS32 architecture; in the *Config3* register, the *DSPP* (*DSP Present*) bit indicates the presence of the DSP Module, and the *DSP2P* (*DSP Rev2 Present*) bit indicates the presence of the MIPS DSP Module Revision 2 or higher. Because the DSP Module is always supplied with the M6200 processor core and is configurable, the *DSPP* and *DSP2P* are always preset to 0’s or 1’s.

The *MX* (*DSP Module Enable*) read/write bit in the CP0 *Status* register must be set to enable access to the additional instructions defined by the DSP Module, as well as to the MTLO/HI, MFLO/HI instructions that access accumulators ac0, ac1, ac2, and ac3. Executing a DSP Module instruction or the MTLO/HI, MFLO/HI instructions with this bit set to zero causes a DSP State Disabled Exception (exception code 26 in the CP0 *Cause* register). This exception can be used by system software to do lazy context switching.

Chapter 3

Memory Management of the M6200 Core

The M6200 processor core includes a Memory Management Unit (MMU) that interfaces between the execution unit and the SRAM controller. The core implements a simple Fixed Mapping Translation (FMT) style MMU.

3.1 Introduction

The MMU in a M6200 processor core translates a virtual address to a physical address beforethe request is sent to the SRAM interface for an external memory reference.

In the M6200 processor core, the MMU is based on a simple algorithm to translate virtual addresses to physical addresses via a Fixed Mapping Translation (FMT) mechanism. These translations are different for various regions of the virtual address space (useg/kuseg, kseg0, kseg1, kseg2/3).

3.1.1 Memory Management Unit (MMU)

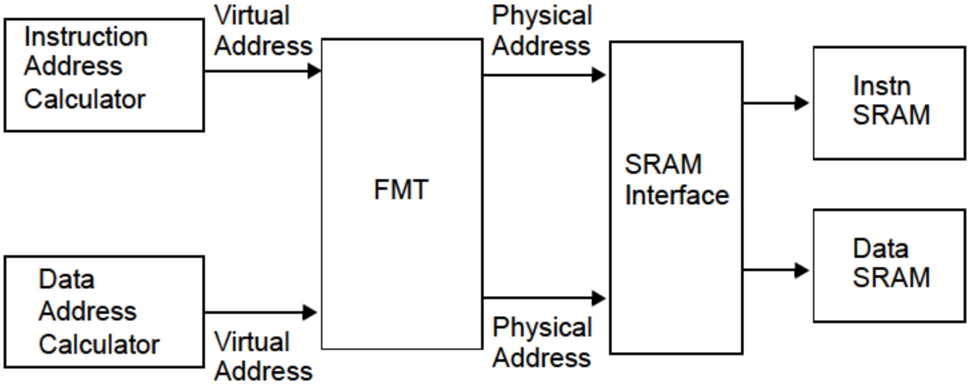
The M6200 core contains a simple Fixed Mapping Translation (FMT) MMU that interfaces between the execution unit and the SRAM controller.

3.1.1.1 Fixed Mapping Translation (FMT)

An FMT is smaller and simpler than the full Translation Lookaside Buffer (TLB) style MMU found in other MIPS cores. Like a TLB, the FMT performs virtual-to-physical address translation and provides attributes for the different segments. Those segments that are unmapped in a TLB implementation (kseg0 and kseg1) are translated identically by the FMT.

Figure 3.1 shows how the memory management unit interacts with the SRAM access in the M6200 core.

Figure 3.1 Address Translation During SRAM Access



3.2 Modes of Operation

The M6200 core implements three modes of operation:

- *User mode* is most often used for applications programs.
- *Kernel mode* is typically used for handling exceptions and operating-system kernel functions, including CP0 management and I/O device accesses.
- *Debug mode* is used during system bring-up and software development.

User mode is most often used for application programs. Kernel mode is typically used for handling exceptions and privileged operating system functions, including CP0 management and I/O device accesses. Debug mode is used for software debugging and most likely occurs within a software development tool.

The address translation performed by the MMU depends on the mode in which the processor is operating.

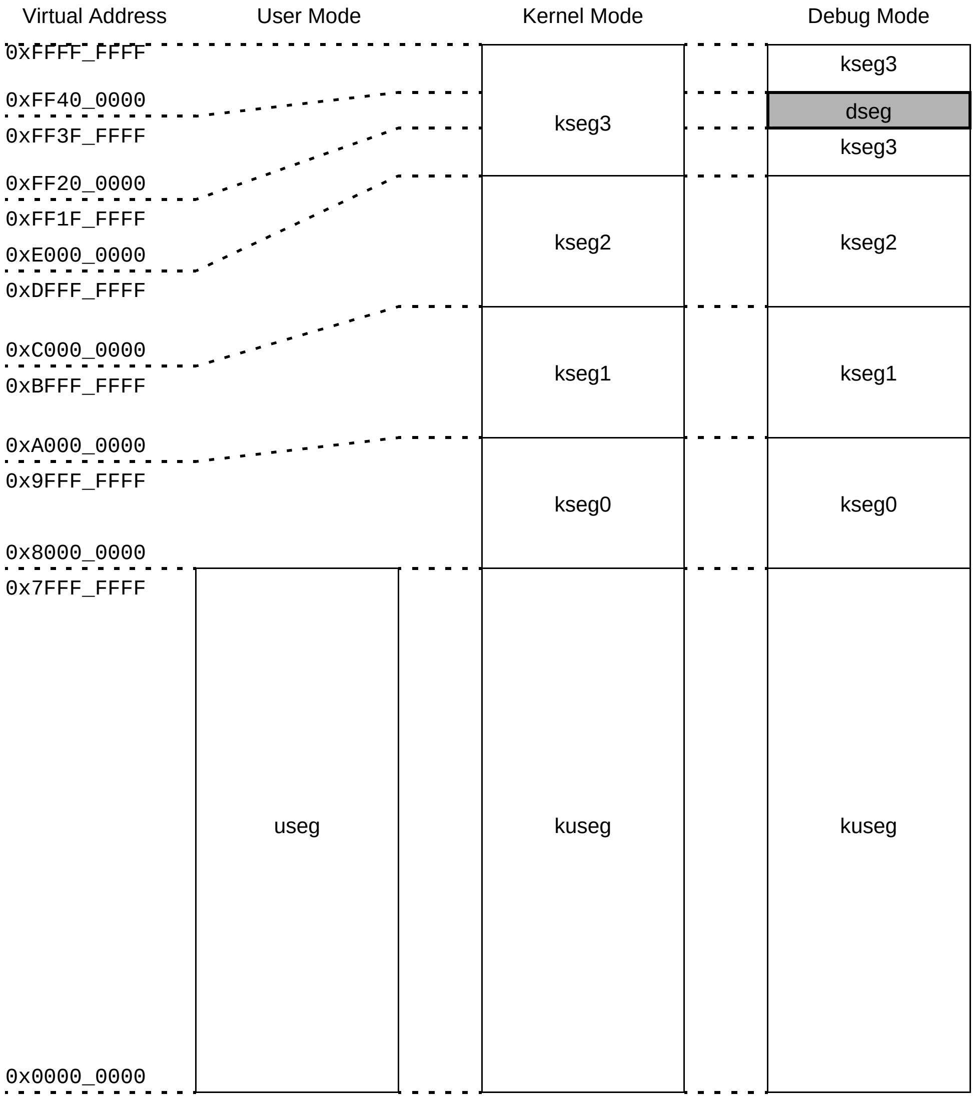
3.2.1 Virtual Memory Segments

The Virtual memory segments differ depending on the mode of operation. Figure 3.2 shows the segmentation for the 4 GByte (2^{32} bytes) virtual memory space addressed by a 32-bit virtual address, for the three modes of operation.

The core enters Kernel mode both at reset and when an exception is recognized. While in Kernel mode, software has access to the entire address space, as well as all CP0 registers. User mode accesses are limited to a subset of the virtual address space (0x0000_0000 to 0x7FFF_FFFF) and can be inhibited from accessing CP0 functions. In User mode, virtual addresses 0x8000_0000 to 0xFFFF_FFFF are invalid and cause an exception if accessed.

Debug mode is entered on a debug exception. While in Debug mode, the debug software has access to the same address space and CP0 registers as for Kernel mode. In addition, while in Debug mode the core has access to the debug segment dseg. This area overlays part of the kernel segment kseg3. dseg access in Debug mode can be turned on or off, allowing full access to the entire kseg3 in Debug mode, if so desired.

Figure 3.2 M6200 processor core Virtual Memory Map



Each of the segments shown in Figure 3.2 are either mapped or unmapped. The following two sub-sections explain the distinction. Then sections 3.2.2 “User Mode”, 3.2.3 “Kernel Mode” and 3.2.4 “Debug Mode” specify which segments are actually mapped and unmapped.

3.2.1.1 Unmapped Segments

An unmapped segment does not use the FMT to translate from virtual-to-physical addresses.

Unmapped segments have a fixed simple translation from virtual to physical address. This is much like the translations the FMT provides for the M6200 core, but we will still make the distinction.

All segments are treated as uncached within the M6200 core. Cache coherency attributes of cached or uncached can be specified and this information will be sent with the request to allow the system to make a distinction between the two.

3.2.1.2 Mapped Segments

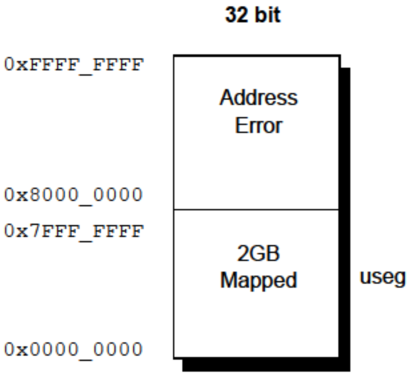
A mapped segment does use the FMT to translate from virtual-to-physical addresses.

For the M6200 core, the mapped segments have a fixed translation from virtual to physical address. The cacheability of the segment is defined in the CP0 *Config* register fields *K23* and *KU*. Write protection of segments is not possible during FMT translation.

3.2.2 User Mode

In user mode, a single 2 GByte (2^{31} bytes) uniform virtual address space called the user segment (useg) is available. Figure 3.3 shows the location of user mode virtual address space.

Figure 3.3 User Mode Virtual Address Space



The user segment starts at address 0x0000_0000 and ends at address 0x7FFF_FFFF. Accesses to all other addresses cause an address error exception.

The processor operates in User mode when the *Status* register contains the following bit values:

- *UM* = 1
- *EXL* = 0
- *ERL* = 0

In addition to the above values, the DM bit in the *Debug* register must be 0.

Table 3.1 lists the characteristics of the useg User mode segments.

Table 3.1 User Mode Segments

Address Bit Value	Status Register Bit Value EXL	 ERL	 UM	Segment Name	Address Range	Segment Size
32-bit A(31) = 0	0	0	1	useg	0x0000_0000 --> 0x7FFF_FFFF	2 GByte (2^{31} bytes)

All valid user mode virtual addresses have their most significant bit cleared to 0, indicating that user mode can only access the lower half of the virtual memory map. Any attempt to reference an address with the most significant bit set while in user mode causes an address error exception.

The system maps all references to *useg* through the FMT.

3.2.3 Kernel Mode

The processor operates in Kernel mode when the *DM* bit in the *Debug* register is 0 and the *Status* register contains one or more of the following values:

- *UM* = 0
- *ERL* = 1
- *EXL* = 1

When a non-debug exception is detected, *EXL* or *ERL* will be set and the processor will enter Kernel mode. At the end of the exception handler routine, an Exception Return (ERET) instruction is generally executed. The ERET instruction jumps to the Exception PC, clears *ERL*, and clears *EXL* if *ERL*=0. This may return the processor to User mode.

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 3.4. Also, Table 3.2 lists the characteristics of the Kernel mode segments.

Figure 3.4 Kernel Mode Virtual Address Space

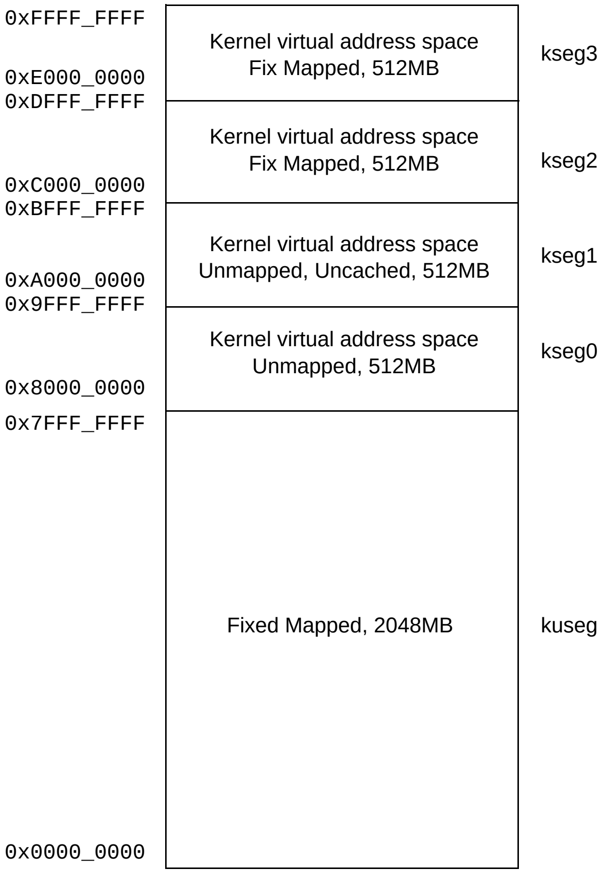


Table 3.2 Kernel Mode Segments

Address Bit Values	Status Register Is One of These Values			Segment Name	Address Range	Segment Size
	UM	EXL	ERL			
A(31) = 0	(UM = 0 or EXL = 1 or ERL = 1) and DM = 0			kuseg	0x0000_0000 through 0x7FFF_FFFF	2 GBytes (2^{31} bytes)
A(31:29) = 100_2				kseg0	0x8000_0000 through 0x9FFF_FFFF	512 MBytes (2^{29} bytes)
A(31:29) = 101_2				kseg1	0xA000_0000 through 0xBFFF_FFFF	512 MBytes (2^{29} bytes)
A(31:29) = 110_2				kseg2	0xC000_0000 through 0xDFFF_FFFF	512 MBytes (2^{29} bytes)
A(31:29) = 111_2				kseg3	0xE000_0000 through 0xFFFF_FFFF	512 MBytes (2^{29} bytes)

3.2.3.1 Kernel Mode, User Space (kuseg)

In Kernel mode, when the most-significant bit of the virtual address (A31) is cleared, the 32-bit kuseg virtual address space is selected and covers the full 2^{31} bytes (2 GBytes) of the current user address space mapped to addresses 0x0000_0000 - 0x7FFF_FFFF.

When the *Status* register's *ERL* = 1, the user address region becomes a 2^{29}-byte unmapped and uncached address space. While in this setting, the kuseg virtual address maps directly to the same physical address.

3.2.3.2 Kernel Mode, Kernel Space 0 (kseg0)

In Kernel mode, when the most-significant three bits of the virtual address are 100_2, 32-bit kseg0 virtual address space is selected; it is the 2^{29}-byte (512-MByte) kernel virtual space located at addresses 0x8000_0000 - 0x9FFF_FFFF. References to kseg0 are unmapped; the physical address selected is defined by subtracting 0x8000_0000 from the virtual address. The *K0* field of the *Config* register controls cacheability.

3.2.3.3 Kernel Mode, Kernel Space 1 (kseg1)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are 101_2, 32-bit kseg1 virtual address space is selected. kseg1 is the 2^{29}-byte (512-MByte) kernel virtual space located at addresses 0xA000_0000 - 0xBFFF_FFFF. References to kseg1 are unmapped; the physical address selected is defined by subtracting 0xA000_0000 from the virtual address.

3.2.3.4 Kernel Mode, Kernel Space 2 (kseg2)

In Kernel mode, when UM = 0, *ERL* = 1, or *EXL* = 1 in the *Status* register, and *DM* = 0 in the *Debug* register, and the most-significant three bits of the 32-bit virtual address are 110_2, 32-bit kseg2 virtual address space is selected. In the M6200 core, this 2^{29}-byte (512-MByte) kernel virtual space is located at physical addresses 0xC000_0000 - 0xDFFF_FFFF.

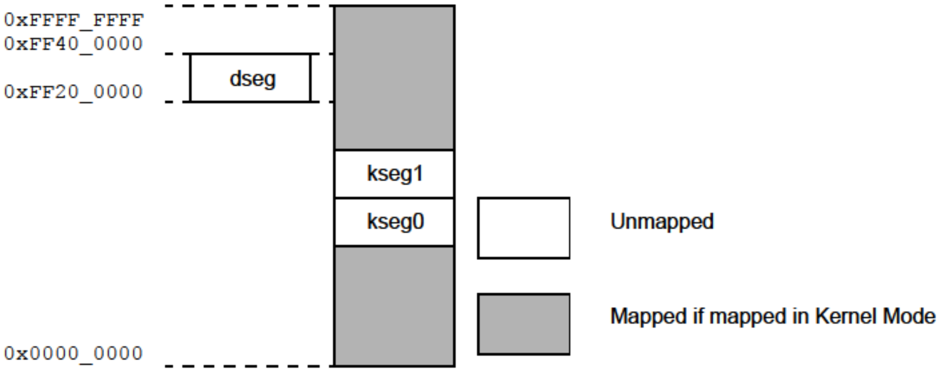
3.2.3.5 Kernel Mode, Kernel Space 3 (kseg3)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are $111_{2\,,}$ the kseg3 virtual address space is selected. In the M6200 core, this 2^{29}-byte (512-MByte) kernel virtual space is located at physical addresses 0xE000_0000 - 0xFFFF_FFFF.

3.2.4 Debug Mode

Debug mode address space is identical to Kernel mode address space with respect to mapped and unmapped areas, except for *kseg3*. In *kseg3,* a debug segment *dseg* co-exists in the virtual address range 0xFF20_0000 to 0xFF3F_FFFF. The layout is shown in Figure 3.5.

Figure 3.5 Debug Mode Virtual Address Space



The dseg is sub-divided into the dmseg segment at 0xFF20_0000 to 0xFF2F_FFFF which is used when the probe services the memory segment, and the drseg segment at 0xFF30_0000 to 0xFF3F_FFFF which is used when memory-mapped debug registers are accessed. The subdivision and attributes for the segments are shown in Table 3.3.

Accesses to memory that would normally cause an exception if tried from kernel mode cause the core to re-enter debug mode via a debug mode exception.

The unmapped kseg0 and kseg1 segments from kernel mode address space are available from debug mode, which allows the debug handler to be executed from uncached and unmapped memory.

Table 3.3 Physical Address and Cache Attributes for dseg, dmseg, and drseg Address Spaces

Segment Name	Sub-Segment Name	Virtual Address	Generates Physical Address	Cache Attribute
dseg	dmseg	0xFF20_0000 through 0xFF2F_FFFF	dmseg maps to addresses 0x0_0000 - 0xF_FFFF in Debug probe memory space.	Uncached
	drseg	0xFF30_0000 through 0xFF3F_FFFF	drseg maps to the breakpoint registers 0x0_0000 - 0xF_FFFF	

3.2.4.1 Conditions and Behavior for Access to drseg, Debug Registers

The behavior of CPU access to the drseg address range at 0xFF30_0000 to 0xFF3F_FFFF is determined as shown in Table 3.4

Table 3.4 CPU Access to drseg Address Range

Transaction	LSNM Bit in Debug Register	Access
Load / Store	1	Kernel mode address space (kseg3)
Fetch	Don't care	drseg, see comments below
Load / Store	0	

Debug software is expected to read the Debug Control Register (*DCR*) to determine which other memory mapped registers exist in drseg. The value returned in response to a read of any unimplemented memory mapped register is unpredictable, and writes are ignored to any unimplemented register in the drseg. Refer to Chapter 8, “Debug Support in the M6200 Core” on page 152 for more information on the *DCR*.

The allowed access size is limited for the drseg. Only word size transactions are allowed. Operation of the processor is undefined for other transaction sizes.

3.2.4.2 Conditions and Behavior for Access to dmseg, Debug Memory

The behavior of CPU access to the dmseg address range at 0xFF20_0000 to 0xFF2F_FFFF is determined by the table shown in Table 3.5.

Table 3.5 CPU Access to dmseg Address Range

Transaction	ProbEn bit in DCR register	LSNM bit in Debug register	Access
Load / Store	Don’t care	1	Kernel mode address space (kseg3)
Fetch	1	Don’t care	dmseg
Load / Store	1	0	
Fetch	0	Don’t care	See comments below
Load / Store	0	0	

The case with access to the dmseg when the *ProbEn* bit in the *DCR* register is 0 is not expected to happen. Debug software is expected to check the state of the *ProbEn* bit in *DCR* register before attempting to reference dmseg. If such a reference does happen, the reference hangs until it is satisfied by the probe. The probe can not assume that there will never be a reference to dmseg if the *ProbEn* bit in the *DCR* register is 0 because there is an inherent race between the debug software sampling the *ProbEn* bit as 1 and the probe clearing it to 0.

3.3 Fixed Mapping MMU

The M6200 core implements a simple Fixed Mapping (FM) memory management unit that is smaller than the a full translation lookaside buffer (TLB) and more easily synthesized. Like a TLB, the FMT performs virtual-to-physical address translation and provides attributes for the different memory segments. Those memory segments which are unmapped in a TLB implementation (kseg0 and kseg1) are translated identically by the FMT MMU.

The FMT also determines the cacheability of each segment. These attributes are controlled via bits in the *Config* register. Table 3.6 shows the encoding for the K23 (bits 30:28), KU (bits 27:25) and K0 (bits 2:0) of the *Config* register.

The M6200 core does not contain caches and will treat all references as uncached, but these *Config* fields will be sent out to the system with the request and it can choose to use them to control any external caching that may be present.

Table 3.6 Cacheability of Segments with Block Address Translation

Segment	Virtual Address Range	Cacheability
useg/kuseg	0x0000_0000- 0x7FFF_FFFF	Controlled by the KU field (bits 27:25) of the *Config* register.
kseg0	0x8000_0000- 0x9FFF_FFFF	Controlled by the K0 field (bits 2:0) of the *Config* register.

Table 3.6 Cacheability of Segments with Block Address Translation (Continued)

Segment	Virtual Address Range	Cacheability
kseg1	0xA000_0000- 0xBFFF_FFFF	Always uncacheable.
kseg2	0xC000_0000- 0xDFFF_FFFF	Controlled by the K23 field (bits 30:28) of the *Config* register.
kseg3	0xE000_0000- 0xFFFF_FFFF	Controlled by K23 field (bits 30:28) of the *Config* register.

The FMT performs a simple translation to map from virtual addresses to physical addresses. This mapping is shown in Figure 3.6. When *ERL*=1, useg and kuseg become unmapped and uncached. The *ERL* behavior is the same as if there was a TLB. The *ERL* mapping is shown in Figure 3.7.

The *ERL* bit is usually never asserted by software. It is asserted by hardware after a Reset, SoftReset or NMI. See Chapter 4, “Exceptions and Interrupts in the M6200 Core” on page 49 for further information on exceptions.

Figure 3.6 FMT Memory Map (ERL=0) in the M6200 Processor Core

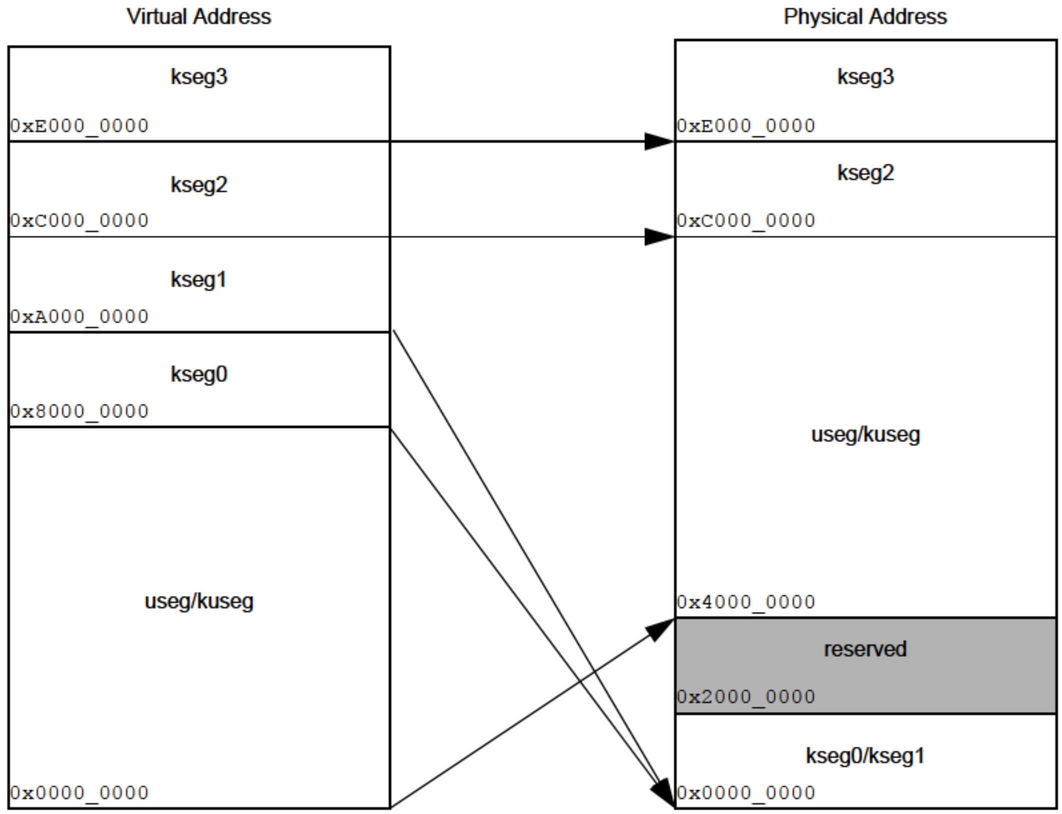
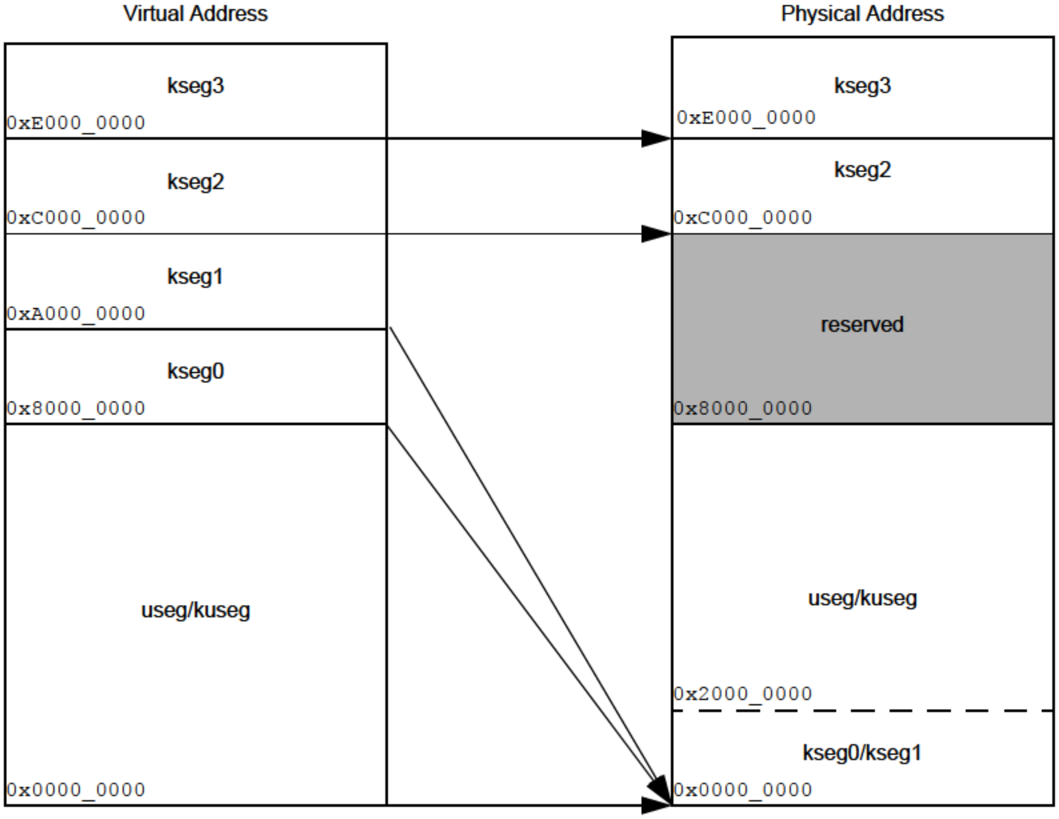


Figure 3.7 FMT Memory Map (ERL=1) in the M6200 Processor Core



3.4 System Control Coprocessor

The System Control Coprocessor (CP0) is implemented as an integral part of M6200 processor core and supports memory management, address translation, exception handling, and other privileged operations. Certain CP0 registers are used to support memory management.

Chapter 4

Exceptions and Interrupts in the M6200 Core

The M6200 processor core receives exceptions from a number of sources, including arithmetic overflows, I/O interrupts, and system calls. When the CPU detects one of these exceptions, the normal sequence of instruction execution is suspended and the processor enters kernel mode. In kernel mode, the core disables interrupts and forces execution of a software exception processor (called a *handler*) located at a specific address. The handler saves the context of the processor, including the contents of the program counter, the current operating mode, and the status of the interrupts (enabled or disabled). This context is saved so it can be restored when the exception has been serviced.

When an exception occurs, the core loads the *Exception Program Counter* (*EPC*) register with a location where execution can restart after the exception has been serviced. Most exceptions are *precise,* which mean that *EPC* can be used to identify the instruction that caused the exception. For precise exceptions, the restart location in the *EPC* register is the address of the instruction that caused the exception or, if the instruction was executing in a branch delay slot or forbidden slot, the address of the branch instruction immediately preceding the delay slot. To distinguish between the two, software must read the *BD* bit in the CP0 *Cause* register. For *imprecise* exceptions, the instruction that caused the exception cannot be identified. Bus error exceptions, CP2 exceptions, debug breakpoint exceptions, and SRAM Interface Parity or ECC errors may be imprecise. For more information on the behavior of the core in response to imprecise exceptions, refer to the descriptions of the individual exceptions in subsequent sections of this chapter.

4.1 Exception Conditions

When an exception condition occurs, the instruction causing the exception and all those that follow it in the pipeline are cancelled (“flushed”). Accordingly, any stall conditions and any later exception conditions that might have referenced this instruction are inhibited—obviously there is no benefit in servicing stalls for a cancelled instruction.

When an exception condition is detected on an instruction fetch, the core aborts that instruction and all instructions that follow. When this instruction reaches the W stage, various CP0 registers are written with the exception state, change the current program counter (PC) to the appropriate exception vector address, and clearing the exception bits of earlier pipeline stages.

This implementation allows all preceding instructions to complete execution and prevents all subsequent instructions from completing. Thus, the value in the *EPC* (*ErrorEPC* for errors, or *DEPC* for debug exceptions) is sufficient to restart execution. It also ensures that exceptions are taken in the order of execution; an instruction taking an exception may itself be killed by an instruction further down the pipeline that takes an exception in a later cycle.

4.2 Exception Priority

Table 4.1 contains a list and a brief description of all exception conditions, The exceptions are listed in the order of their relative priority, from highest priority (Reset) to lowest priority. When several exceptions occur simultaneously, the exception with the highest priority is taken.

Table 4.1 Priority of Exceptions

Exception	Description
Reset	Assertion of *SI_ColdResetN* signal.
Soft Reset	Assertion of *SI_WarmResetN* signal.
DSS	Debug Single Step.
DINT	Debug Interrupt. Caused by the assertion of the external *EJ_DINT* input, or by setting the DbgBrk bit in the OCR register.
NMI	Asserting edge of *SI_NMI* signal.
Interrupt	Assertion of unmasked hardware or software interrupt signal.
Protection - Instruction fetch	Instruction fetch access to a protected memory region was attempted.
DIB	Debug hardware instruction break matched.
AdEL (Instruction)	Fetch address alignment error. User-mode fetch reference to kernel address.
ISRAM Parity Error	Parity error on ISRAM transmission.
ISRAM ECC Error	ECC error on ISRAM access.
IBE	Instruction fetch bus error.
Execution Exceptions	An instruction could not be completed because it was not allowed access to the required resources (Coprocessor Unusable) or was illegal (Reserved Instruction). If both exceptions occur on the same instruction, the Coprocessor Unusable Exception may take priority over the Reserved Instruction Exception. (See the description of the RI exceptions for more details.) In the M6200 and M6250 family cores, a DSP-disabled (MDMX) exception is treated as the same class and priority as the Coprocessor Unusable Exception.
Protection - Instr Execution	Occurs when an attempt to write *EBase* is not allowed by MPU.
Protection - Data access	Data access to a protected memory region was attempted.
DDBL / DDBS	Debug Data Address Break (address only) or Debug Data Value Break on Store (address and value).
AdEL (Data)	Load address alignment error. User mode load reference to kernel address.
AdES (Data)	Store address alignment error. User mode store to kernel address.
DSRAM Parity Error	Parity error on DSRAM transmission.
DSRAM ECC Error	ECC error on DSRAM access.
DBE	Load or store bus error.
DDBL	Debug data hardware breakpoint matched in load data compare.
CBrk	Debug complex breakpoint.

4.3 Interrupts

In the MIPS32® Release 1 architecture, support for exceptions included two software interrupts, six hardware interrupts, and a special-purpose timer interrupt. The timer interrupt was provided external to the core and was typically combined with hardware interrupt 5 in a system-dependent manner. Interrupts were handled either through the general exception vector (offset 0x180) or the special interrupt vector (0x200), based on the value of *CauseIV*. Software was required to prioritize interrupts as a function of the *CauseIV* bits in the interrupt handler prologue.

Release 2 of the Architecture, implemented by the M6200 core, adds a number of upward-compatible extensions to the Release 1 interrupt architecture, including support for vectored interrupts and the implementation of a new interrupt mode that permits the use of an external interrupt controller.

The M6200 core also includes the Microcontroller Application-Specific Extension (MCU ASE) that provides enhanced interrupt delivery and reduction of interrupt latency.

4.3.1 Interrupt Modes

The M6200 core includes support for three interrupt modes, as defined by Release 2 of the Architecture:

- Interrupt Compatibility mode, in which the behavior of the M6200 is identical to the behavior of a Release 1 implementations.

- Vectored Interrupt (VI) mode, which adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt, and to assign a GPR shadow set for use during interrupt processing. The presence of this mode is denoted by the *VInt* bit in the *Config3* register. Although this mode is architecturally optional, it is always present on the M6200 processor, so the *VInt* bit will always read as a 1.

- External Interrupt Controller (EIC) mode, which redefines the way interrupts are handled to provide full support for an external interrupt controller that handles prioritization and vectoring of interrupts. As with VI mode, this mode is architecturally optional. The presence of this mode is denoted by the *VEIC* bit in the *Config3* register. On the M6200 core, the *VEIC* bit is set externally by the static input, *SI_EICPresent*, to allow system logic to indicate the presence of an external interrupt controller.

Following reset, the M6200 processor defaults to Compatibility mode, which is fully compatible with all implementations of Release 1 of the Architecture.

Table 4.2 shows the current interrupt mode of the procœssor as a function of the Coprocessor 0 register fields that can affect the mode.

Table 4.2 Interrupt Modes

Status$_{BEV}$	Cause$_{IV}$	IntCtl$_{VS}$	Config3$_{VINT}$	Config3$_{VEIC}$	Interrupt Mode
1	x	x	x	x	Compatibly
x	0	x	x	x	Compatibility
x	x	=0	x	x	Compatibility
0	1	≠0	1	0	Vectored Interrupt
0	1	≠0	x	1	External Interrupt Controller

Table 4.2 Interrupt Modes (Continued)

Status_{BEV}	Cause_{IV}	IntCtl_{VS}	Config3_{VINT}	Config3_{VEIC}	Interrupt Mode
0	1	≠0	0	0	Can’t happen - IntCtl_{VS} can not be non-zero if neither Vectored Interrupt nor External Interrupt Controller mode is implemented.
“x” denotes don’t care					

4.3.1.1 Interrupt Compatibility Mode

This is the default interrupt mode for the processor and is entered when a Reset exception occurs. In this mode, interrupts are non-vectored and dispatched though exception vector offset 0x180 (if Cause_{IV} = 0) or vector offset 0x200 (if Cause_{IV} = 1). This mode is in effect if any of the following conditions are true:

- Cause_{IV} = 0
- Status_{BEV} = 1
- IntCtl_{VS} = 0, which would be the case if vectored interrupts are not implemented, or have been disabled.

Here is a typical software handler for interrupt compatibility mode:

```
/*
 * Assumptions:
 *  - CauseIV = 1 (if it were zero, the interrupt exception would have to
 *                 be isolated from the general exception vector before getting
 *                 here)
 *  - GPRs k0 and k1 are available (no shadow register switches invoked in
 *                                  compatibility mode)
 *  - The software priority is IP9..IP0 (HW7..HW0, SW1..SW0)
 *
 * Location: Offset 0x200 from exception base
 */

IVexception:
   mfc0   k0, C0_Cause        /* Read Cause register for IP bits */
   mfc0   k1, C0_Status       /* and Status register for IM bits */
   andi   k0, k0, M_CauseIM   /* Keep only IP bits from Cause */
   and    k0, k0, k1          /* and mask with IM bits */
   beq    k0, zero, Dismiss   /* no bits set - spurious interrupt */
   clz    k0, k0              /* Find first bit set, IP9..IP0; k0 = 14..23 */
   xori   k0, k0, 0x17        /* 14..23 => 9..0 */
   sll    k0, k0, VS          /* Shift to emulate software IntCtlVS */
   la     k1, VectorBase      /* Get base of 10 interrupt vectors */
   addu   k0, k0, k1          /* Compute target from base and offset */
   jr     k0                  /* Jump to specific exception routine */
   nop

/*
 * Each interrupt processing routine processes a specific interrupt, analogous
 * to those reached in VI or EIC interrupt mode. Since each processing routine
 * is dedicated to a particular interrupt line, it has the context to know
 * which line was asserted.  Each processing routine may need to look further
 * to determine the actual source of the interrupt if multiple interrupt requests
 * are ORed together on a single IP line. Once that task is performed, the
 * interrupt may be processed in one of two ways:
 *
 * - Completely at interrupt level (e.g., a simply UART interrupt). The
 *   SimpleInterrupt routine below is an example of this type.
 * - By saving sufficient state and re-enabling other interrupts. In this
 *   case the software model determines which interrupts are disabled during
 *   the processing of this interrupt. Typically, this is either the single
 *   StatusIM bit that corresponds to the interrupt being processed, or some
 *   collection of other StatusIM bits so that “lower” priority interrupts are
 *    also disabled. The NestedInterrupt routine below is an example of this type.
 */

SimpleInterrupt:
/*
 * Process the device interrupt here and clear the interupt request
 * at the device. In order to do this, some registers may need to be
 * saved and restored. The coprocessor 0 state is such that an ERET
 * will simple return to the interrupted code.
 */
   eret                       /* Return to interrupted code */

NestedException:
/*
 * Nested exceptions typically require saving the EPC and Status registers,
 * any GPRs that may be modified by the nested exception routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

   /* Save GPRs here, and setup software context */
   mfc0   k0, C0_EPC          /* Get restart address */
   sw     k0, EPCSave         /* Save in memory */
   mfc0   k0, C0_Status       /* Get Status value */
   sw     k0, StatusSave      /* Save in memory */
   li     k1, ~IMbitsToClear  /* Get Im bits to clear for this interrupt */
                              /*   this must include at least the IM bit */
                              /*   for the current interrupt, and may include */
                              /*   others */
   and    k0, k0, k1             /* Clear bits in copy of Status */
   ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                 /* Clear KSU, ERL, EXL bits in k0 */
   mtc0   k0, C0_Status          /* Modify mask, switch to kernel mode, */
                                 /*   re-enable interrupts */

   /*
    * Process interrupt here, including clearing device interrupt.
    * In some environments this may be done with a thread running in
    * kernel or user mode. Such an environment is well beyond the scope of
    * this example.
    */

/*
```

```
    * To complete interrupt processing, the saved values must be restored
    * and the original interrupted code restarted.
    */

      di                           /* Disable interrupts - may not be required */
      lw     k0, StatusSave        /* Get saved Status (including EXL set) */
      lw     k1, EPCSave           /*   and EPC */
      mtc0   k0, C0_Status         /* Restore the original value */
      mtc0   k1, C0_EPC            /*  and EPC */
      /* Restore GPRs and software state */
      eret                         /* Dismiss the interrupt */
```

4.3.1.2 Vectored Interrupt (VI) Mode

In Vectored Interrupt (VI) mode, a priority encoder prioritizes pending interrupts and generates a vector which can be used to direct each interrupt to a dedicated handler routine. This mode also allows each interrupt to be mapped to a GPR shadow register set for use by the interrupt handler. VI mode is in effect when all the following conditions are true:

- $Config3_{VInt} = 1$
- $Config3_{VEIC} = 0$
- $IntCtl_{VS} \neq 0$
- $Cause_{IV} = 1$
- $Status_{BEV} = 0$

In VI interrupt mode, the eight hardware interrupts are interpreted as individual hardware interrupt requests. The timer interrupt is combined in a system-dependent way (external to the core) with the hardware interrupts (the interrupt with which they are combined is indicated by the *PTI* field in *IntCtlI*) to provide the appropriate relative priority of the timer interrupt with that of the hardware interrupts. The processor interrupt logic ANDs each of the $Cause_{IP}$ bits with the corresponding $Status_{IM}$ bits. If any of these values is 1, and if interrupts are enabled ($Status_{IE} = 1$, $Status_{EXL} = 0$, and $Status_{ERL} = 0$), an interrupt is signaled and a priority encoder scans the values in the order shown in Table 4.3.

Table 4.3 Relative Interrupt Priority for Vectored Interrupt Mode

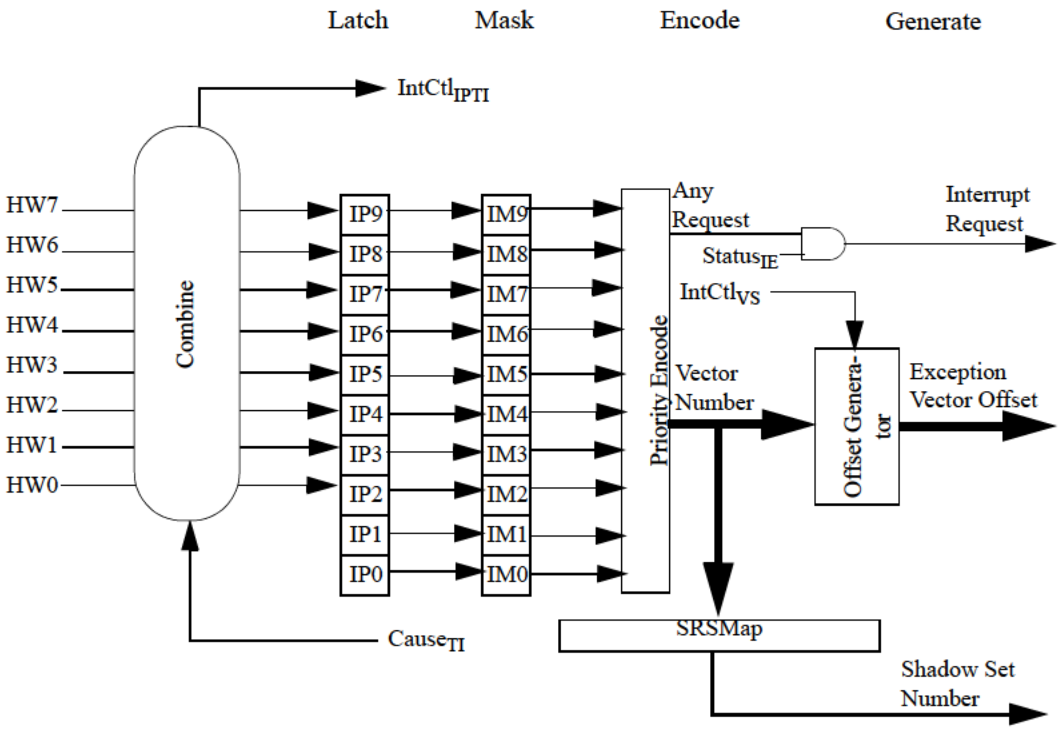
Relative Priority	Interrupt Type	Interrupt Source	Interrupt Request Calculated From	Vector Number Generated by Priority Encoder
Highest Priority	Hardware	HW7	IP9 and IM9	9
		HW6	IP8 and IM8	8
		HW5	IP7 and IM7	7
		HW4	IP6 and IM6	6
		HW3	IP5 and IM5	5
		HW2	IP4 and IM4	4
		HW1	IP3 and IM3	3
		HW0	IP2 and IM2	2

Table 4.3 Relative Interrupt Priority for Vectored Interrupt Mode

Relative Priority	Interrupt Type	Interrupt Source	Interrupt Request Calculated From	Vector Number Generated by Priority Encoder
	Software	SW1	IP1 and IM1	1
Lowest Priority		SW0	IP0 and IM0	0

The priority order places a relative priority on each hardware interrupt and places the software interrupts at a priority lower than all hardware interrupts. When the priority encoder finds the highest priority pending interrupt, it outputs an encoded vector number that is used in the calculation of the handler for that interrupt, as described below. This is shown pictorially in Figure 4.1.

Figure 4.1 Interrupt Generation for Vectored Interrupt Mode



Note that an interrupt request may be deasserted between the time the processor detects the interrupt request and the time that the software interrupt handler runs. The interrupt logic must latch the exception vector offset when making an interrupt request to the processor pipeline so that the exception vector offset does not change if the interrupt request is deasserted. The software interrupt handler must be prepared to handle this condition by simply returning from the interrupt via ERET.

A typical software handler for vectored interrupt mode bypasses the entire sequence of code following the IV exception label shown for the compatibility mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine. Unlike the compatibility mode examples, a vectored interrupt handler may take advantage of a dedicated GPR shadow set to avoid saving any registers. As such, the Simple Interrupt code shown above need not save the GPRs.

A nested interrupt is similar to that shown for compatibility mode, but may also take advantage of running the nested exception routine in the GPR shadow set dedicated to the interrupt or in another shadow set. Such a routine might look as follows:

```
NestedException:
/*
 * Nested exceptions typically require saving the EPC, Status and SRSCtl registers,
 * setting up the appropriate GPR shadow set for the routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

   /* Use the current GPR shadow set, and setup software context */
   mfc0   k0, C0_EPC          /* Get restart address */
   sw     k0, EPCSave         /* Save in memory */
   mfc0   k0, C0_Status       /* Get Status value */
   sw     k0, StatusSave      /* Save in memory */
   mfc0   k0, C0_SRSCtl       /* Save SRSCtl if changing shadow sets */
   sw     k0, SRSCtlSave
   li     k1, ~IMbitsToClear  /* Get Im bits to clear for this interrupt */
                              /*   this must include at least the IM bit */
                              /*   for the current interrupt, and may include */
                              /*   others */
   and    k0, k0, k1             /* Clear bits in copy of Status */
   /* If switching shadow sets, write new value to SRSCtlPSS here */
   ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                 /* Clear KSU, ERL, EXL bits in k0 */
   mtc0   k0, C0_Status          /* Modify mask, switch to kernel mode, */
                                 /*   re-enable interrupts */
   /*
    * If switching shadow sets, clear only KSU above, write target
    * address to EPC, and do execute an eret to clear EXL, switch
    * shadow sets, and jump to routine
    */

   /* Process interrupt here, including clearing device interrupt */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

   di                         /* Disable interrupts - may not be required */
   lw     k0, StatusSave      /* Get saved Status (including EXL set) */
   lw     k1, EPCSave         /*   and EPC */
   mtc0   k0, C0_Status       /* Restore the original value */
   lw     k0, SRSCtlSave      /* Get saved SRSCtl */
   mtc0   k1, C0_EPC          /*  and EPC */
   mtc0   k0, C0_SRSCtl       /* Restore shadow sets */
   ehb                        /* Clear hazard */
   eret                       /* Dismiss the interrupt */
```

4.3.1.3 External Interrupt Controller Mode

External Internal Interrupt Controller Mode redefines the way that the processor interrupt logic is configured to provide support for an external interrupt controller. The interrupt controller is responsible for prioritizing all interrupts, including hardware, software, timer, and performance counter interrupts, and directly supplying to the processor the priority level and vector number of the highest priority interrupt. EIC interrupt mode is in effect if all of the following conditions are true:

- $Config3_{VEIC} = 1$
- $IntCtl_{VS} \neq 0$
- $Cause_{IV} = 1$
- $Status_{BEV} = 0$

In EIC interrupt mode, the processor sends the state of the software interrupt requests ($Cause_{IP1..IP0}$), the timer interrupt request ($Cause_{TI}$), the performance counter interrupt request ($Cause_{PCI}$) and Fast Debug Channel Interrupt ($Cause_{FDCI}$) to the external interrupt controller, where it prioritizes these interrupts in a system-dependent way with other hardware interrupts. The interrupt controller can be a hard-wired logic block, or it can be configurable based on control and status registers. This allows the interrupt controller to be more specific or more general as a function of the system environment and needs.

The external interrupt controller prioritizes its interrupt requests and produces the priority level and the vector number of the highest priority interrupt to be serviced. The priority level, called the Requested Interrupt Priority Level (RIPL), is an 8-bit encoded value in the range 0..255, inclusive. A value of 0 indicates that no interrupt requests are pending. The values 1..255 represent the lowest (1) to highest (255) RIPL for the interrupt to be serviced. The interrupt controller passes this value on the 8 hardware interrupt lines, which are treated as an encoded value in EIC interrupt mode. There are two implementation options available for the vector offset:

1. The first option is to send a separate vector number along with the RIPL to the processor.
2. A second option is to send an entire vector offset along with the RIPL to the processor. This option is enabled through the core’s configuration GUI, and it is not affected by software.

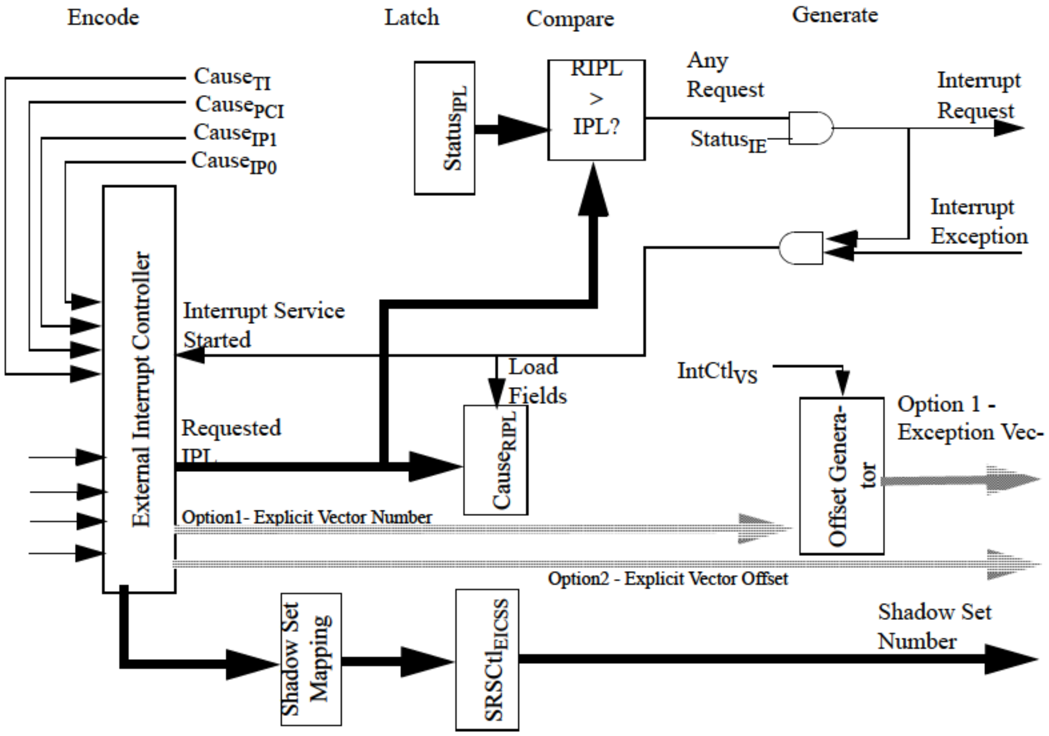
The M6200 core does not support the option to treat the RIPL value as the vector number for the processor.

$Status_{IPL}$ (which overlays $StatusI_{IM9..IM2}$) is interpreted as the Interrupt Priority Level (IPL) at which the processor is currently operating (with a value of zero indicating that no interrupt is currently being serviced). When the interrupt controller requests service for an interrupt, the processor compares *RIPL* with $Status_{IPL}$ to determine if the requested interrupt has higher priority than the current *IPL*. If RIPL is strictly greater than $Status_{IPL}$, and interrupts are enabled ($Status_{IE} = 1$, $Status_{EXL} = 0$, and $Status_{ERL} = 0$) an interrupt request is signaled to the pipeline. When the processor starts the interrupt exception, it loads RIPL into $Cause_{RIPL}$ (which overlays $Cause_{IP9..IP2}$) and signals the external interrupt controller to notify it that the request is being serviced. Because $Cause_{RIPL}$ is only loaded by the processor when an interrupt exception issignaled, it is available to software duringinterrupt processing. The vector number that the EIC passes to the core is combined with the $IntCtl_{VS}$ to determine where the interrupt service routine is located. The vector number is not stored in any software-visible registers.

In EIC interrupt mode, the external interrupt controller is also responsible for supplying the GPR shadow set number to use when servicing the interrupt. As such, the *SRSMap* register is not used in this mode, and the mapping of the vectored interrupt to a GPR shadow set is done by programming (or designing) the interrupt controller to provide the correct GPR shadow set number when an interrupt is requested. When the processor loads an interrupt request into *$Cause_{RIPL}$*, it also loads the GPR shadow set number into *$SRSCtl_{EICSS}$*, which is copied to *$SRSCtl_{CSS}$* when the interrupt is serviced.

The operation of EIC interrupt mode is shown pictorially in Figure 4.2.

Figure 4.2 Interrupt Generation for External Interrupt Controller Interrupt Mode



A typical software handler for EIC interrupt mode bypasses the entire sequence of code following the IV exception label shown for the compatibility-mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine. Unlike the compatibility mode examples, an EIC interrupt handler may take advantage of a dedicated GPR shadow set to avoid saving any registers. As such, the Simple Interrupt code shown above need not save the GPRs.

A nested interrupt is similar to that shown for compatibility mode, but may also take advantage of running the nested exception routine in the GPR shadow set dedicated to the interrupt or in another shadow set. It also need only copy *$Cause_{RIPL}$* to *$Status_{IPL}$* to prevent lower priority interrupts from interrupting the handler. Such a routine might look as follows:

```
NestedException:
/*
 * Nested exceptions typically require saving the EPC, Status,and SRSCtl registers,
 * setting up the appropriate GPR shadow set for the routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */
```

```
   /* Use the current GPR shadow set, and setup software context */
   mfc0   k1, C0_Cause        /* Read Cause to get RIPL value */
   mfc0   k0, C0_EPC          /* Get restart address */
   srl    k1, k1, S_CauseRIPL /* Right justify RIPL field */
   sw     k0, EPCSave         /* Save in memory */
   mfc0   k0, C0_Status       /* Get Status value */
   sw     k0, StatusSave      /* Save in memory */
   ins    k0, k1, S_StatusIPL, 6 /* Set IPL to RIPL in copy of Status */
   mfc0   k1, C0_SRSCtl       /* Save SRSCtl if changing shadow sets */
   sw     k1, SRSCtlSave
   /* If switching shadow sets, write new value to SRSCtlPSS here */
   ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                              /* Clear KSU, ERL, EXL bits in k0 */
   mtc0   k0, C0_Status          /* Modify IPL, switch to kernel mode, */
                              /*   re-enable interrupts */
   /*
    * If switching shadow sets, clear only KSU above, write target
    * address to EPC, and do execute an eret to clear EXL, switch
    * shadow sets, and jump to routine
    */

   /* Process interrupt here, including clearing device interrupt */

/*
 * The interrupt completion code is identical to that shown for VI mode above.
 */
```

4.3.2 Generation of Exception Vector Offsets for Vectored Interrupts

For vectored interrupts (in either VI or EIC interrupt mode), a vector number is produced by the interrupt control logic. This number is combined with *IntCtlVS* to create the interrupt offset, which is added to 0x200 to create the exception vector offset. For VI interrupt mode, the vector number is in the range 0..9, inclusive. For EIC interrupt mode, the vector number is in the range 0..63, inclusive. The *IntCtlVS* field specifies the spacing between vector locations. If this value is zero (the default reset state), the vector spacing is zero and the processor reverts to Interrupt Compatibility Mode. A non-zero value enables vectored interrupts, and Table 4.4 shows the exception vector offset for a representative subset of the vector numbers and values of the *IntCtlVS* field.

Table 4.4 Exception Vector Offsets for Vectored Interrupts

Vector Number	Value of IntCtl_{VS} Field				
	2#00001	**2#00010**	**2#00100**	**2#01000**	**2#10000**
0	16#0200	16#0200	16#0200	16#0200	16#0200
1	16#0220	16#0240	16#0280	16#0300	16#0400
2	16#0240	16#0280	16#0300	16#0400	16#0600
3	16#0260	16#02C0	16#0380	16#0500	16#0800
4	16#0280	16#0300	16#0400	16#0600	16#0A00
5	16#02A0	16#0340	16#0480	16#0700	16#0C00
6	16#02C0	16#0380	16#0500	16#0800	16#0E00
7	16#02E0	16#03C0	16#0580	16#0900	16#1000

Table 4.4 Exception Vector Offsets for Vectored Interrupts

Vector Number	Value of IntCtl$_{VS}$ Field				
	2#00001	2#00010	2#00100	2#01000	2#10000
• • •					
61	16#09A0	16#1140	16#2080	16#3F00	16#7C00
62	16#09C0	16#1180	16#2100	16#4000	16#7E00
63	16#09E0	16#11C0	16#2180	16#4100	16#8000

The general equation for the exception vector offset for a vectored interrupt is:

$$\texttt{vectorOffset} \leftarrow \texttt{16\#200} + (\texttt{vectorNumber} \times (\texttt{IntCtl}_{\texttt{VS}} \parallel \texttt{2\#00000}))$$

When using large vector spacing and EIC mode, the offset value can overlap with bits that are specified in the *EBase* register. Software must ensure that any overlapping bits are specified as 0 in *EBase*. This implementation ORs together the offset and base registers, but it is architecturally undefined and software should not rely on this behavior.

Although there are 255 EIC priority interrupts, only 64 vectors are provided. There is no one-to-one mapping for each EIC interrupt to its interrupt vector. The 255 priority interrupts will share the 64 interrupt vectors as specified by the *SI_EICVector*[5:0] input pins. However, as mentioned in option 2 of Section 4.3.1.3 "External Interrupt Controller Mode", the *SI_Offset*[17:1] input pins can be used to provide each EIC interrupt with a unique interrupt handler location.

4.3.3 MCU ASE Enhancement for Interrupt Handling

The MCU ASE extends the MIPS/microMIPS**32** Architecture with a set of new features designed for the microcontroller market. The MCU ASE contains enhancements in two key areas: interrupt delivery and interrupt latency. For more details, refer to the *The MCU Privileged Resource Architecture* chapter of the *MIPS® Architecture for Programmers Volume IV-h: The MCU Application-Specific Extension to the MIPS32 Architecture* [13] or *MIPS® Architecture for Programmers Volume IV-h: The MCU Application-Specific Extension to the microMIPS32™ Architecture* [14].

4.3.3.1 Interrupt Delivery

The MCU ASE extends the number of hardware interrupt sources from 6 to 8. For legacy and vectored-interrupt mode, this represents 8 external interrupt sources. For EIC mode, the widened *IPL* and *RIPL* fields can now represent 256 external interrupt sources.

4.3.3.2 Interrupt Latency Reduction

The MCU ASE includes a package of extensions to MIPS32/microMIPS32 that decrease the latency of the processor's response to a signalled interrupt.

4.3.3.2.1 Interrupt Vector Prefetching

Normally on MIPS architecture processors, when an interrupt or exception is signalled, execution pipelines must be flushed before the interrupt/exception handler is fetched. This is necessary to avoid mixing the contexts of the interrupted/faulting program and the exception handler. The MCU ASE introduces a hardware mechanism in which the interrupt exception vector is prefetched whenever the interrupt input signals change. The prefetch memory transaction occurs in parallel with the pipeline flush and exception prioritization. This decreases the overall latency of the execution of the interrupt handler's first instruction.

4.3.3.2.2 Automated Interrupt Prologue

The use of Shadow Register Sets avoids the software steps of having to save general-purpose registers before handling an interrupt.

The MCU ASE adds additional hardware logic that automatically saves some of the CP0 state in the stack and automatically updates some of the CP0 registers in preparation for interrupt handling.

4.3.3.2.3 Automated Interrupt Epilogue

A mirror to the Automated Prologue, this features automates the restoration of some of the CP0 registers from the stack and the preparation of some of the CP0 registers for returning to non-exception mode. This feature is implemented by the IRET instruction, which is introduced in this ASE.

4.3.3.2.4 Interrupt Chaining

An optional feature of the Automated Interrupt Epilogue, this feature allows handling a second interrupt after a primary interrupt is handled, without returning to non-exception mode (and the related pipeline flushes that would normally be necessary).

4.4 GPR Shadow Registers

Release 2 of the Architecture optionally removes the need to save and restore GPRs on entry to high priority interrupts or exceptions, and to provide specified processor modes with the same capability. This is done by introducing multiple copies of the GPRs, called *shadow sets*, and allowing privileged software to associate a shadow set with entry to kernel mode via an interrupt vector or exception. The normal GPRs are logically considered shadow set zero.

The number of GPR shadow sets is a build-time option on the M6200 core. The core allows one (the normal GPRs), two, four, eight or sixteen shadow sets. The highest number actually implemented is indicated by the $SRSCtl_{HSS}$ field. If this field is zero, only the normal GPRs are implemented.

Shadow sets are new copies of the GPRs that can be substituted for the normal GPRs on entry to kernel mode via an interrupt or exception. When a shadow set is bound to a kernel mode entry condition, reference to GPRs work exactly as one would expect, but they are redirected to registers that are dedicated to that condition. Privileged software may need to reference all GPRs in the register file, even specific shadow registers that are not visible in the current mode. The RDPGPR and WRPGPR instructions are used for this purpose. The *CSS* field of the *SRSCtl* register provides the number of the current shadow register set, and the *PSS* field of the *SRSCtl* register provides the number of the previous shadow register set (that which was current before the last exception or interrupt occurred).

If the processor is operating in VI interrupt mode, binding of a vectored interrupt to a shadow set is done by writing to the *SRSMap* register. If the processor is operating in EIC interrupt mode, the binding of the interrupt to a specific shadow set is provided by the external interrupt controller, and is configured in an implementation-dependent way. Binding of an exception or non-vectored interrupt to a shadow set is done by writing to the *ESS* field of the *SRSCtl* register. When an exception or interrupt occurs, the value of $SRSCtl_{CSS}$ is copied to $SRSCtl_{PSS}$, and $SRSCtl_{CSS}$ is set to the value taken from the appropriate source. On an ERET, the value of $SRSCtl_{PSS}$ is copied back into $SRSCtl_{CSS}$ to restore the shadow set of the mode to which control returns. More precisely, the rules for updating the fields in the *SRSCtl* register on an interrupt or exception are as follows:

1. No field in the *SRSCtl* register is updated if any of the following conditions is true. In this case, steps 2 and 3 are skipped.

   - The exception is one that sets *Status*$_{ERL}$: Reset, Soft Reset, or NMI.

   - The exception causes entry into Debug Mode.

   - *Status*$_{BEV}$ = 1

   - *Status*$_{EXL}$ = 1

2. *SRSCtl*$_{CSS}$ is copied to *SRSCtl*$_{PSS}$.

3. *SRSCtl*$_{CSS}$ is updated from one of the following sources:

   - The appropriate field of the *SRSMap* register, based on IPL, if the exception is an interrupt, *Cause*$_{IV}$ = 1, *Config3*$_{VEIC}$ = 0, and *Config3*$_{VInt}$ = 1. These are the conditions for a vectored interrupt.

   - The *EICSS* field of the *SRSCtl* register if the exception is an interrupt, *Cause*$_{IV}$ = 1, and *Config3*$_{VEIC}$ = 1. These are the conditions for a vectored EIC interrupt.

   - The *ESS* field of the *SRSCtl* register in any other case. This is the condition for a non-interrupt exception, or a non-vectored interrupt.

Similarly, the rules for updating the fields in the *SRSCtl* register at the end of an exception or interrupt are as follows:

1. No field in the *SRSCtl* register is updated if any of the following conditions is true. In this case, step 2 is skipped.

   - A DERET is executed.

   - An ERET is executed with *Status*$_{ERL}$ = 1.

2. *SRSCtl*$_{PSS}$ is copied to *SRSCtl*$_{CSS}$.

These rules have the effect of preserving the *SRSCtl* register in any case of a nested exception or one which occurs before the processor has been fully initialize (*Status*$_{BEV}$ = 1).

Privileged software may switch the current shadow set by writing a new value into *SRSCtl*$_{PSS}$, loading *EPC* with a target address, and doing an ERET.

4.5 Exception Vector Locations

The Reset, Soft Reset, NMI and Debug exceptions are vectored to a specific location as shown in Table 4.5 and Table 4.6. Addresses for all other exceptions are a combination of a vector offset and a vector base address. In Release 1 of the architecture, the vector base address was fixed. In Release 2 of the architecture, software is allowed to specify the vector base address via the *EBase* register for exceptions that occur when *StatusBEV* equals 0. Another degree of flexibility in the selection of the vector base address, for use when *StatusBEV* equals 1, is provided via a set of input pins, *SI_UseExceptionBase* and *SI_ExceptionBase[29:12]*. Table 4.5 gives the vector base address when *SI_UseExceptionBase* equals 0, as a function of the exception and whether the *BEV* bit is set in the *Status* register. Table 4.6 gives the vector base addresses when *SI_UseExceptionBase* equals 1. As can be seen in Table 4.6, when *SI_UseExceptionBase* equals 1, the exception vectors for cases where *StatusBEV* equals 0 are not affected.

Table 4.7 gives the offsets from the vector base address as a function of the exception. Note that the IV bit in the *Cause* register causes Interrupts to use a dedicated exception vector offset, rather than the general exception vector. Table 4.4 gives the offset from the base address in the case where *StatusBEV* = 0 and *CauseIV* = 1. Table 4.8 combines these two tables into one that contains all possible vector addresses as a function of the state that can affect the vector selection. To avoid complexity in the table, it is assumed that *IntCtlVS* is 0.

.

Table 4.5 Exception Vector Base Addresses when SI_UseExceptionBase = 0

Exception	Status$_{BEV}$	
	0	**1**
Reset, Soft Reset, NMI	16#BFC0.0000	
Debug (with ProbEn = 0 in the OCI CONTROL Register)	16#BFC0.0480	
Debug (with ProbEn = 1 in the OCI CONTROL Register)	16#FF20.0200	
SRAM Interface Parity Error	EBase$_{31\ 30}$ || 1 || EBase$_{28\ 12}$ || 16#000 Note that EBase$_{31\ 30}$ have the fixed value 2#10	16#BFC0.0300
SRAM ECC Error	EBase$_{31\ 30}$ || 1 || EBase$_{28\ 12}$ || 16#000 Note that EBase$_{31\ 30}$ have the fixed value 2#10	16#BFC0.0300
Other	EBase$_{31\ 12}$ || 16#000 Note that EBase$_{31\ 30}$ have the fixed value 2#10	16#BFC0.0200

Table 4.6 Exception Vector Base Addresses when SI_UseExceptionBase = 1

Exception	Status$_{BEV}$	
	0	**1**
Reset, Soft Reset, NMI	2#10 || SI_ExceptionBase[29:12] || 16#000	
Debug with OCR$_{ProbEn}$ = 1	16#FF20.0200	
Debug with OCR$_{ProbEn}$ = 0 and DCR$_{RdVec}$=1	DebugVectorAddr[31:0] Note that DebugVectorAddr[31:30] have the fixed value 2#10	
Debug with OCR$_{ProbEn}$ = 0 and DCR$_{RdVec}$ = 0	2#10 ||SI_ExceptionBase[29:12] || 16#480	

Table 4.6 Exception Vector Base Addresses when SI_UseExceptionBase = 1 (Continued)

	Status$_{BEV}$	
Exception	**0**	**1**
SRAM Interface Parity Error	EBase$_{31\ 30}$ || 1 || EBase$_{28\ 12}$ || `16#000` Note that EBase$_{31\ 30}$ have the fixed value `2#10`	`16#BFC0.0300`
SRAM ECC Error	EBase$_{31\ 30}$ || 1 || EBase$_{28\ 12}$ || `16#000` Note that EBase$_{31\ 30}$ have the fixed value `2#10`	`16#BFC0.0300`
Other	EBase$_{31\ 12}$ || `16#000` Note that EBase$_{31\ 30}$ have the fixed value `2#10`	`16#BFC0.0200`

Table 4.7 Exception Vector Offsets

Exception	Vector Offset
Cache Error	`16#100`
General Exception	`16#180`
Interrupt, *Cause*$_{IV}$ = 1	`16#200`
Reset, Soft Reset, NMI	None (Uses Reset Base Address)

Table 4.8 Exception Vectors

Exception	SI_Use Exception Base	Status$_{BEV}$	Status$_{EXL}$	Cause$_{IV}$	Debug ProbEn	DCR RdVec	Vector Assumes that EBase retains its reset state and that IntCtl$_{VS}$ = 0
Reset, Soft Reset, NMI	0	x	x	x	x	x	`16#BFC0.0000`

Reset, Soft Reset, NMI	1	x	x	x	x	x	`2#10 ||`
							`SI_ExceptionBase[29:12] ||`
							`16#000`

Debug	0	x	x	x	0	0	`2#10 ||`
							`DebugVectorAddr[29:7] ||`
							`16#00`

Debug	1	x	x	x	1	0	`2#10 ||`
							`SI_ExceptionBase[29:12] ||`
							`16#480`

Table 4.8 Exception Vectors (Continued)

Exception	SI_Use Exception Base	Status$_{BEV}$	Status$_{EXL}$	Cause$_{IV}$	Debug ProbEn	DCR RdVec	Vector Assumes that EBase retains its reset state and that IntCtl$_{VS}$ = 0
SRAM Interface Parity Error	x	0	x	x	x	x	`16#EBase[31:30] || 2#1 || EBase[28:12] || 16#000`
SRAM Interface Parity Error	x	1	x	x	x	x	`16#BFC0.0300`
SRAM ECC Error	x	0	x	x	x	x	`16#EBase[31:30] || 2#1 || EBase[28:12] || 16#000`
SRAM ECC Error	x	1	x	x	x	x	`16#BFC0.0300`
Interrupt	x	0	0	0	x	x	`16#8000.0180`
Interrupt	x	0	0	1	x	x	`16#8000.0200`
Interrupt	x	1	0	0	x	x	`16#BFC0.0380`
Interrupt	x	1	0	1	x	x	`16#BFC0.0400`
All others	x	0	x	x	x	x	`16#8000.0180`
All others		1	x	x	x	x	`16#BFC0.0380`
‘x’ denotes don’t care							

4.6 General Exception Processing

With the exception of Reset, Soft Reset, NMI, SRAM ECC error, SRAM Parity error, and Debug exceptions, which have their own special processing as described below, exceptions have the same basic processing flow:

- If the *EXL* bit in the *Status* register is zero, the *EPC* register is loaded with the PC at which execution will be restarted and the *BD* bit is set appropriately in the *Cause* register (see Table 5.17). The value loaded into the *EPC* register is dependent on whether the instruction is in the delay slot of a branch or jump which has delay slots. Table 4.9 shows the value stored in each of the CP0 PC registers, including *EPC*. If *Status*$_{BEV}$ = 0, the *CSS* field in the *SRSCtl* register is copied to the *PSS* field, and the *CSS* value is loaded from the appropriate source.

  If the *EXL* bit in the *Status* register is set, the *EPC* register is not loaded and the *BD* bit is not changed in the *Cause* register.

Table 4.9 Value Stored in EPC, ErrorEPC, or DEPC on an Exception

In Branch/Jump Delay Slot?	Value stored in EPC/ErrorEPC/DEPC
No	Address of the instruction
Yes	Address of the branch or jump instruction (PC-4)

Table 4.9 Value Stored in EPC, ErrorEPC, or DEPC on an Exception

In Branch/Jump Delay Slot?	Value stored in EPC/ErrorEPC/DEPC
No	Upper 31 bits of the address of the instruction, combined with the *ISA Mode* bit
Yes	Upper 31 bits of the branch or jump instruction (PC-2 or PC-4 depending on size of the instruction in the microMIPS ISA Mode and PC-4 in the 32-bit ISA Mode), combined with the *ISA Mode* bit

- The *CE* and *ExcCode* fields of the *Cause* registers are loaded with the values appropriate to the exception. The *CE* field is loaded, but not defined, for any exception type other than a coprocessor unusable exception.

- The *EXL* bit is set in the *Status* register.

- The processor is started at the exception vector.

The value loaded into *EPC* represents the restart address for the exception and need not be modified by exception handler software in the normal case. Software need not look at the *BD* bit in the *Cause* register unless it wishes to identify the address of the instruction that actually caused the exception.

Note that individual exception types may load additional information into other registers. This is noted in the description of each exception type below.

Operation:

```
/* If Status$_{EXL}$ is 1, all exceptions go through the general exception vector */
/* and neither EPC nor Cause$_{BD}$ nor SRSCtl are modified */
if Status$_{EXL}$ = 1 then
    vectorOffset ← 0x180
else
/* For implementations that include the MIPS16e ASE, calculate potential */
    /* PC adjustment for exceptions in the delay slot */
    if (Config1$_{CA}$ = 0 & Config3$_{ISA}$ = 0 ) then
        restartPC ← PC
        branchAdjust ← 4        /* Possible adjustment for delay slot */
    elseif (Config1$_{CA}$ = 1) /* MIPS16 is implemented */
        restartPC ← PC$_{..1}$ || ISAMode
        if (ISAMode = 0) or ExtendedMIPS16Instruction
            branchAdjust ← 4    /* Possible adjustment for 32-bit MIPS delay slot */
        else
            branchAdjust ← 2    /* Possible adjustment for MIPS16 delay slot */
        endif
    elseif (Config3$_{ISA}$ = 1) /* only microMIPS is implemented */
        restartPC ← PC
        branchAdjust ← BDSlotInstrSize/* Adjust for microMIPS delay slot */
    elseif (Config3$_{ISA}$ > 1) /* both MIPS32/64 & microMIPS are implemented */
        restartPC ← PC$_{..1}$ || ISAMode
        if (ISAMode = 0)
            branchAdjust ← 4    /* Possible adjustment for 32-bit MIPS delay slot */
        else
            branchAdjust ← BDSlotInstrSize/* Adjust for microMIPS delay slot */
        endif
    endif
    if InstructionInBranchDelaySlot then
        EPC ← restartPC/* PC of branch/jump */
        $Cause_{BD}$ ← 1
    else
        EPC ← restartPC              /* PC of instruction */
        $Cause_{BD}$ ← 0
    endif

    /* Compute vector offsets as a function of the type of exception */
    NewShadowSet ← $SRSCtl_{ESS}$        /* Assume exception, Release 2 only */
    if ExceptionType = TLBRefill then
        vectorOffset ← 0x000
    elseif (ExceptionType = Interrupt) then
        if ($Cause_{IV}$ = 0) then
            vectorOffset ← 0x180
        else
            if ($Status_{BEV}$ = 1) or ($IntCtl_{VS}$ = 0) then
                vectorOffset ← 0x200
            else
                if $Config3_{VEIC}$ = 1 then
                    if (EIC_option1)
                        VecNum ← $Cause_{RIPL}$
                    elseif (EIC_option2)
                        VecNum ← EIC_VecNum_Signal
                    endif
                    NewShadowSet ← $SRSCtl_{EICSS}$
                else
                    VecNum ← VIntPriorityEncoder()
                    NewShadowSet ← $SRSMap_{IPL \times 4+3..IPL \times 4}$
                endif
                if (EIC_option3)
                    vectorOffset ← EIC_VectorOffset_Signal
                else
                    vectorOffset ← 0x200 + (VecNum × ($IntCtl_{VS}$ ∥ 0b00000))
                endif
            endif /* if ($Status_{BEV}$ = 1) or ($IntCtl_{VS}$ = 0) then */
        endif /* if ($Cause_{IV}$ = 0) then */
    endif /* elseif (ExceptionType = Interrupt) then */

    /* Update the shadow set information for an implementation of */
    /* Release 2 of the architecture */
    if (ArchitectureRevision ≥ 2) and ($SRSCtl_{HSS}$ > 0) and ($Status_{BEV}$ = 0) then
        /* It is implementation-dependent whether this update occurs */
        /* if $Status_{ERL}$ = 1. */
        $SRSCtl_{PSS}$ ← $SRSCtl_{CSS}$
        $SRSCtl_{CSS}$ ← NewShadowSet
    endif
endif /* if $Status_{EXL}$ = 1 then */

$Cause_{CE}$ ← FaultingCoprocessorNumber
$Cause_{ExcCode}$ ← ExceptionType
$Status_{EXL}$ ← 1

if $Config1_{CA}$ = 1 then
    ISAMode ← 0
endif
if $Config3_{ISA}$ > 1 then
    ISAMode ← $Config3_{ISAOnExc}$
```

```
endif

/* Calculate the vector base address */
if $Status_{BEV}$ = 1 then
   vectorBase ← 0xBFC0.0200
else
   if ArchitectureRevision ≥ 2 then
      /* The fixed value of $EBase_{31..30}$ forces the base to be in kseg0 or kseg1 */
      vectorBase ← $EBase_{31..12}$ || 0x000
   else
      vectorBase ← 0x8000.0000
   endif
endif

/* Exception PC is the sum of vectorBase and vectorOffset. Vector */
/* offsets > 0xFFF (vectored or EIC interrupts only), require */
/* that $EBase_{15..12}$ have zeros in each bit position less than or */
/* equal to the most significant bit position of the vector offset */
PC ← $vectorBase_{31..30}$ || ($vectorBase_{29..0}$ + $vectorOffset_{29..0}$)
                           /* No carry between bits 29 and 30 */
```

4.7 Debug Exception Processing

All debug exceptions have the same basic processing flow:

- The *DEPC* register is loaded with the program counter (PC) value at which execution will be restarted and the *DBD* bit is set appropriately in the *Debug* register. The value loaded into the *DEPC* register is the current PC if the instruction is not in the delay slot of a branch, or the PC-4 of the branch if the instruction is in the delay slot of a branch.

- The *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB*, *DINT*, *DIBImpr*, *DDBLImpr*, and *DDBSImpr* bits in the *Debug* register are updated appropriately depending on the debug exception type.

- The *Debug2* register is updated with additional information for complex breakpoints.

- *Halt* and *Doze* bits in the *Debug* register are updated appropriately.

- *DM* bit in the *Debug* register is set to 1.

- The processor is started at the debug exception vector.

The value loaded into *DEPC* represents the restart address for the debug exception and need not be modified by the debug exception handler software in the usual case. Debug software need not look at the *DBD* bit in the *Debug* register unless it wishes to identify the address of the instruction that actually caused the debug exception.

A unique debug exception is indicated through the *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB*, *DINT*, *DIBImpr*, *DDBLImpr*, and *DDBSImpr* bits in the *Debug* register.

No other CP0 registers or fields are changed due to the debug exception, thus no additional state is saved.

Operation:

```
if InstructionInBranchDelaySlot then
   DEPC ← PC-4
```

```
    Debug$_{DBD}$ ← 1
else
    DEPC ← PC
    Debug$_{DBD}$ ← 0
endif
Debug$_{D* bits}$ ← DebugExceptionType
Debug$_{Halt}$ ← HaltStatusAtDebugException
Debug$_{Doze}$ ← DozeStatusAtDebugException
Debug$_{DM}$ ← 1
if OCI CONTROLRegister$_{ProbTrap}$ = 1 then
    PC ← 0xFF20_0200
else
    PC ← 0xBFC0_0480
endif
```

The same debug exception vector location is used for all debug exceptions. The location is determined by the ProbTrap bit in the OCI CONTROL Register (OCR), as shown in Table 4.10

Table 4.10 Debug Exception Vector Location

OCR$_{ProbEn}$	OCR$_{ProbTrap}$	OCR$_{RDVec}$	Debug Exception Vector Address
x	0	0	0xFFFF FFFF BFC0 0480
x	0	1	0xFFFF FFFF 0000 0000 + (DebugVectorAddr$_{31\ 1}$ || 0)
1	1	0	0xFFFF FFFF FF20 0200 in dmseg
1	1	1	

4.8 Exception Descriptions

The following subsections describe each of the exceptions listed in the same order as shown in Table 4.1.

4.8.1 Reset Exception

A reset exception occurs when the *SI_ColdResetN* signal is asserted to the processor. This exception is not maskable. When this exception occurs, the processor performs a full reset initialization, including aborting state machines, establishing critical state, and generally placing the processor in a state in which it can execute instructions from uncached, unmapped address space. On a Reset/WarmReset exception, the state of the processor is not defined, with the following exceptions:

- The *Config, Config1*, *Config2*, and *Config3* registers are initialized with their boot state.
- The *RP*, *BEV*, *TS*, *SR*, *NMI*, and *ERL* fields of the *Status* register are initialized to a specified state.
- *Watch* register enables and *Performance Counter* register interrupt enables are cleared.
- The *ErrorEPC* register is loaded with the restart PC, as described in Table 4.9. Note that this value may or may not be predictable if the Reset Exception was taken as the result of power being applied to the processor because PC may not have a valid value in that case. In some implementations, the value loaded into *ErrorEPC* register may not be predictable on either a Reset or Soft Reset Exception.
- PC is loaded with 0xBFC0_0000.

Cause Register ExcCode Value:

None

Additional State Saved:

None

Entry Vector Used:

Reset (0xBFC0_0000)

Operation:

```
Wired ← 0
HWREna ← 0
Status$_{BEV}$ ← 1
Status$_{TS}$ ← 0
Status$_{SR}$ ← 0
Status$_{NMI}$ ← 0
Status$_{ERL}$ ← 1
IntCtl$_{VS}$ ← 0
SRSCtl$_{HSS}$ ← HighestImplementedShadowSet
SRSCtl$_{ESS}$ ← 0
SRSCtl$_{PSS}$ ← 0
SRSCtl$_{CSS}$ ← 0
SRSMap ← 0
Cause$_{DC}$ ← 0
EBase$_{ExceptionBase}$ ← 0
Config ← ConfigurationState
Config$_{K0}$ ← 2                  # Suggested - see Config register description
Config1 ← ConfigurationState
Config2 ← ConfigurationState
Config3 ← ConfigurationState
WatchLo[n]$_{I}$ ← 0               # For all implemented Watch registers
WatchLo[n]$_{R}$ ← 0               # For all implemented Watch registers
WatchLo[n]$_{W}$ ← 0               # For all implemented Watch registers
PerfCnt.Control[n]$_{IE}$ ← 0      # For all implemented PerfCnt registers
if ( Config1$_{CA}$ = 0 & Config3$_{ISA}$ = 0$_{)}$ then
   restartPC ← PC
   branchAdjust ← 4       # Possible adjustment for delay slot
elseif ( Config1$_{CA}$ = 1 ) then /* MIPS16 implemented */
   restartPC ← PC$_{31..1}$ ∥ ISAMode
   if (ISAMode = 0) or ExtendedMIPS16Instruction
      branchAdjust ← 4    # Possible adjustment for 32-bit MIPS delay slot
   else
      branchAdjust ← 2    # Possible adjustment for MIPS16e delay slot
   endif
   ISAMode ← 0
elseif (Config3$_{ISA}$ = 1) /* only microMIPS is implemented */
   restartPC ← PC
   branchAdjust ← BDSlotInstrSize/* Adjust for microMIPS delay slot */
elseif (Config3$_{ISA}$ > 1) /* both MIPS32/64 & microMIPS are implemented */
   restartPC ← PC$_{31..1}$ ∥ ISAMode
   if (ISAMode = 0)
      branchAdjust ← 4    /* Possible adjustment for 32-bit MIPS delay slot */
   else
      branchAdjust ← BDSlotInstrSize/* Adjust for microMIPS delay slot */
   endif
   ISAMode ← Config3$_{ISA}$ ==3
```

```
endif
if InstructionInBranchDelaySlot then
    ErrorEPC ← restartPC - branchAdjust # PC of branch/jump
else
    ErrorEPC ← restartPC # PC of instruction
endif
PC ← 0xBFC0 0000
```

4.8.2 Soft Reset Exception

A Soft Reset Exception occurs when the *SI_WarmResetN* signal is asserted. This exceptions is not maskable. When a Soft Reset Exception occurs, the processor performs a subset of the full reset initialization. Although a Soft Reset Exception does not unnecessarily change the state of the processor, it may be forced to do so in order to place the processor in a state in which it can execute instructions from uncached, unmapped address space. Since bus, cache, or other operations may be interrupted, portions of the cache, memory, or other processor state may be inconsistent.

The primary difference between the Reset and Soft Reset Exceptions is in actual use. The Reset Exception is typically used to initialize the processor on power-up, while the Soft Reset Exception is typically used to recover from a non-responsive (hung) processor. The semantic difference is provided to allow boot software to save critical coprocessor 0 or other register state to assist in debugging the potential problem. As such, the processor may reset the same state when either reset signal is asserted, but the interpretation of any state saved by software may be very different.

In addition to any hardware initialization required, the following state is established on a Soft Reset Exception:

- The *BEV*, *TS*, *SR*, *NMI*, and *ERL* fields of the *Status* register are initialized to a specified state.
- The *ErrorEPC* register is loaded with the restart PC, as described in Table 4.9.
- PC is loaded with `0xBFC0 0000`.

Cause Register ExcCode Value

None

Additional State Saved

None

Entry Vector Used

Reset (`0xBFC0 0000`)

Operation

```
$Config_{K0}$ ← 2                    # Suggested - see Config register description
$Status_{BEV}$ ← 1
$Status_{TS}$ ← 0
$Status_{SR}$ ← 1
$Status_{NMI}$ ← 0
$Status_{ERL}$ ← 1
$PerfCnt.Control[n]_{IE}$ ← 0      # For all implemented PerfCnt registers
if ( $Config1_{CA}$ = 0 & $Config3_{ISA}$ = 0$_{)}$then
    restartPC ← PC
    branchAdjust ← 4       # Possible adjustment for delay slot
elseif ( $Config1_{CA}$ = 1) then
    restartPC ← $PC_{31..1}$ || ISAMode
    if (ISAMode = 0) or ExtendedMIPS16Instruction
```

```
      branchAdjust ← 4   # Possible adjustment for 32-bit MIPS delay slot
   else
      branchAdjust ← 2   # Possible adjustment for MIPS16e delay slot
   endif
   ISAMode ← 0
elseif ($Config3_{ISA}$ = 1) /* only microMIPS is implemented */
   restartPC ← PC
   branchAdjust ← BDSlotInstrSize/* Adjust for microMIPS delay slot */
elseif ($Config3_{ISA}$ > 1) /* both MIPS32/64 & microMIPS are implemented */
   restartPC ← $PC_{31..1}$ ∥ ISAMode
   if (ISAMode = 0)
      branchAdjust ← 4   /* Possible adjustment for 32-bit MIPS delay slot */
   else
      branchAdjust ← BDSlotInstrSize/* Adjust for microMIPS delay slot */
   endif
   ISAMode ← $Config3_{ISA}$ ==3
endif
if InstructionInBranchDelaySlot then
   ErrorEPC ← restartPC - branchAdjust # PC of branch/jump
else
   ErrorEPC ← restartPC # PC of instruction
endif
PC ← 0xBFC0 0000
```

4.8.3 Debug Single Step Exception

When single-step mode is enabled, a Debug Single Step exception occurs each time the processor has taken a single execution step in Non-Debug Mode. An execution step is a single instruction, or an instruction pair consisting of a jump/branch instruction and the instruction in the associated delay slot. The *SSt* bit in the *Debug* register enables Debug Single Step exceptions. They are disabled on the first execution step after a DERET.

The *DEPC* register points to the instruction on which the Debug Single Step Exception occurred, which is also the next instruction to execute when returning from Debug Mode. The debug software can examine the system state before this instruction is executed. Thus the DEPC will not point to the instruction(s) that have just executed in the execution step, but rather the instruction following the execution step. The Debug Single Step Exception never occurs on an instruction in a jump/branch delay slot, because the jump/branch and the instruction in the delay slot are always executed in one execution step; thus the DBD bit in the Debug register is never set for a Debug Single Step Exception.

Exceptions occurring on the instruction(s) executed with debug single step exception enabled are taken even though debug single step was enabled. For a normal exception (other than reset), a debug single step exception is then taken on the first instruction in the normal exception handler. Debug exceptions are unaffected by single-step mode, e.g. returning to a SDBBP instruction with debug single step exceptions enabled causes a debug software breakpoint exception, and *DEPC* points to the SDBBP instruction. However, returning to an instruction (not jump/branch) just before the SDBBP instruction, causes a debug single step exception with the *DEPC* pointing to the SDBBP instruction.

To ensure proper functionality of single step, the Debug Single Step Exception has priority over all other exceptions except reset and soft reset.

Note that the Debug Single Step Exception is only possible when the *NoSSt* bit in the *Debug* register is 0.

Debug Register Debug Status Bit Set

DSS

Additional State Saved

None

Entry Vector Used

Debug exception vector

4.8.4 Debug Interrupt Exception

A debug interrupt exception is either caused by the DbgBrk bit in the OCI CONTROL register, or caused by the debug interrupt request signal to the CPU.

The debug interrupt exception is an asynchronous debug exception that is taken as soon as possible, but with no specific relation to the executed instructions. The*DEPC* register is set to the instruction where execution should continue after the debug handler is through. The *DBD* bit is set based on whether the interrupted instruction was executing in the delay slot of a branch.

Debug Register Debug Status Bit Set

DINT

Additional State Saved

None

Entry Vector Used

Debug exception vector

4.8.5 Non-Maskable Interrupt (NMI) Exception/

A non maskable interrupt exception occurs when the *SI_NMI* signal is asserted to the processor. *SI_NMI* is an edge sensitive signal - only one NMI exception will be taken each time it is asserted. An NMI exception occurs only at instruction boundaries, so it does not cause any reset or other hardware initialization. The state of the cache, memory, and other processor states are consistent and all registers are preserved, with the following exceptions:

- The *BEV*, *TS*, *SR*, *NMI*, and *ERL* fields of the *Status* register are initialized to a specified state.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC.
- PC is loaded with 0xBFC0_0000.

Cause Register ExcCode Value:

None

Additional State Saved:

None

Entry Vector Used:

Reset (0xBFC0_0000)

Operation:

```
    $\mathrm{Status}_{\mathrm{BEV}} \leftarrow 1$
```

```
$Status_{TS}$ ← 0
$Status_{SR}$ ← 0
$Status_{NMI}$ ← 1
$Status_{ERL}$ ← 1
if ( $Config1_{CA}$ = 0 & $Config3_{ISA}$ = 0) then
   restartPC ← PC
   branchAdjust ← 4       # Possible adjustment for delay slot
elseif ( $Config1_{CA}$ = 1) then /* MIPS16 is implemented */
   restartPC ← $PC_{31..1}$ || ISAMode
   if (ISAMode = 0) or ExtendedMIPS16Instruction
      branchAdjust ← 4    # Possible adjustment for 32-bit MIPS delay slot
   else
      branchAdjust ← 2    # Possible adjustment for MIPS16e delay slot
   endif
   ISAMode ← 0
elseif ($Config3_{ISA}$ = 1) /* only microMIPS is implemented */
   restartPC ← PC
   branchAdjust ← BDSlotInstrSize/* Adjust for microMIPS delay slot */
elseif ($Config3_{ISA}$ > 1) /* both MIPS32/64 & microMIPS are implemented */
   restartPC ← $PC_{33..1}$ ‖ ISAMode
   if (ISAMode = 0)
      branchAdjust ← 4    /* Possible adjustment for 32-bit MIPS delay slot */
   else
      branchAdjust ← BDSlotInstrSize/* Adjust for microMIPS delay slot */
   endif
   ISAMode ← $Config3_{ISA}$ ==3
endif
if InstructionInBranchDelaySlot then
   ErrorEPC ← restartPC # PC of branch/jump
else
   ErrorEPC ← restartPC # PC of instruction
endif
PC ← 0xBFC0 0000
```

4.8.6 Interrupt Exception

The interrupt exception occurs when one or more of the eight hardware, two software, or timer interrupt requests is enabled by the *Status* register, and the interrupt input is asserted. See 4.3 “Interrupts” on page 51 for more details about the processing of interrupts.

Register ExcCode Value:

Int

Additional State Saved:

Table 4.11 Register States an Interrupt Exception

Register State	Value
CauseIP	indicates the interrupts that are pending.

Entry Vector Used:

General exception vector (offset 0x180) if the *IV* bit in the *Cause* register is zero.

Interrupt vector (offset 0x200) if the *IV* bit in the *Cause* register is one.

See 4.3.2 “Generation of Exception Vector Offsets for Vectored Interrupts” on page 59 for the entry vector used, depending on the interrupt mode the processor is operating in.

4.8.7 Debug Instruction Break Exception

A debug instruction break exception occurs when an instruction hardware breakpoint matches an executed instruction. The *DEPC* register and *DBD* bit in the *Debug* register indicate the instruction that caused the instruction hardware breakpoint to match. This exception can only occur if instruction hardware breakpoints are implemented.

Debug Register Debug Status Bit Set:

DIB

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

4.8.8 Address Error Exception — Instruction Fetch/Data Access

An address error exception occurs under the following circumstances:

- An instruction is fetched from an address that is not aligned on a word boundary.
- A load or store word instruction is executed in which the address is not aligned on a word boundary.
- A load or store halfword instruction is executed in which the address is not aligned on a halfword boundary.
- A reference is made to a kernel address space from User Mode.
- A reference is made to a supervisor address space from User Mode.

Note that in the case of an instruction fetch that is not aligned on a word boundary, PC is updated before the condition is detected. Therefore, both *EPC* and *BadVAddr* point to the unaligned instruction address. In the case of a data access the exception is taken if either an unaligned address or an address that was inaccessible in the current processor mode was referenced by a load or store instruction.

Cause Register ExcCode Value:

AdEL: Reference was a load or an instruction fetch

AdES: Reference was a store

Additional State Saved:

Table 4.12 CP0 Register States on an Address Exception Error

Register State	Value
BadVAddr	Failing address

Entry Vector Used:

General exception vector (offset 0x180)

4.8.9 SRAM Interface Parity Error Exception

An SRAM interface error exception occurs when an instruction or data reference detects a data parity error. This exception is not maskable. The exception vector is to an unmapped, uncached address. When detected internally, this exception is imprecise on the M6200.

Cause Register ExcCode Value

CacheErr: The CacheErr code is used to represent that an error in transmis sion was detected. For details, refer to the *CacheErr* and *CacheErrAddr* registers.

Additional State Saved

Table 4.13 CP0 Register States on a SRAM Interface Parity Error Exception

Register State	Value
CacheErr	Address of error detected
ErrorEPC	Restart PC

Entry Vector Used

SRAM interface parity error vector (offset 0x100)

4.8.10 SRAM ECC Error Exception

An SRAM ECC Error exception occurs when an instruction or data reference detects an ECC error. This exception is not maskable. The exception vector is to an unmapped, uncached address. This exception is imprecise on the M6200.

Cause Register ExcCode Value

CacheErr: The CacheErr code is used to represent that an error in transmis sion was detected. For details, refer to the *CacheErr* and *CacheErrAddr* registers.

Additional State Saved

Table 4.14 CP0 Register States on an SRAM ECC Error Exception

Register State	Value
CacheErr	Address of error detected
ErrorEPC	Restart PC

Entry Vector Used

SRAM ECC error vector (offset 0x100)

4.8.11 Bus Error Exception — Instruction Fetch or Data Access

A bus error exception occurs when an instruction or data access makes a bus request and that request terminates in an error. The bus error exception can occur on either an instruction fetch or a data access. Bus error exceptions that occur on an instruction fetch have a higher priority than bus error exceptions that occur on a data access.

By default, bus errors on instruction accesses are returned immediately with the read data (if available). When fetching speculatively, the M6200 core will save these errors until the fetch becomes critical and will then be executed.

For bus error exceptions on data read accesses, all data reads are considered critical and bus error exceptions are taken immediately.

For bus error exceptions on data write accesses, the write response is typically expected to return on the next cycle. However, if the response is delayed, it can be sent later on the port *DS_BError*. These error responses will cause the core to take an imprecise bus error exception; however, it does not accurately update the ErrorEPC value, and software is responsible for verifying the state of the CPU caused by the imprecise error exception.

Cause Register ExcCode Value:

IBE:	Error on an instruction reference
DBE:	Error on a data reference

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.12 Protection Exception

The protection exception occurs when an access to memory that has been protected by the Memory Protection Unit has been attempted. Or under certain circumstances, attempted write to the *EBase* register. See the *Security Features of the M6200 Processor Family* [4] for more information.

Register ExcCode Value:

Prot (Cause Code 29)

Additional State Saved:

MPU Config Register, Triggered Field

MPU StatusN Register, Cause* Fields

BadVAddr, triggered address

BadInstr (data-triggered protection exceptions only)

BadInstrP

Entry Vector Used

General exception vector (offset 0x180)

4.8.13 Debug Software Breakpoint Exception

A debug software breakpoint exception occurs when an SDBBP instruction is executed. The *DEPC* register and *DBD* bit in the *Debug* register will indicate the SDBBP instruction that caused the debug exception.

Debug Register Debug Status Bit Set:

DBp

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

4.8.14 Execution Exception — System Call

The system call exception is one of the execution exceptions. A system call exception occurs when a SYSCALL instruction is executed.

Cause Register ExcCode Value:

Sys

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.15 Execution Exception — Breakpoint

The breakpoint exception is one of the execution exceptions. All of these exceptions have the same priority. A breakpoint exception occurs when a BREAK instruction is executed.

Cause Register ExcCode Value:

Bp

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.16 Execution Exception — Reserved Instruction

The reserved instruction exception is one of the execution exceptions. All of these exceptions have the same priority. A reserved instruction exception occurs when a reserved or undefined major opcode or function field is executed. This includes Coprocessor 2 instructions which are decoded reserved in the Coprocessor 2. On the M6200 core, the priority between RI exceptions and other execution exceptions is dependent on the ISA mode. If the core is operating in microMIPS mode, an RI exception is flagged if the opcode does not refer to a defined instruction, or if the hardware resources for this instruction are not configured/present. If the core is operating in MIPS32 mode, and the instruction falls in one of the privileged resource groups identified by the major opcode CP0, COP1, COP2, or DSP minor opcodes, then the CPU flags a Coprocessor Unusable exception; else, an undefined instruction will cause an RI exception.

Cause Register ExcCode Value:

RI

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.17 Execution Exception — Coprocessor Unusable

The coprocessor unusable exception is one of the execution exceptions. A coprocessor unusable exception occurs when an attempt is made to execute a coprocessor instruction for one of the following:

- A corresponding coprocessor unit that has not been marked usable by setting its CU bit in the *Status* register
- CP0 instructions when the processor is executing in User Mode

Cause Register ExcCode Value:

CpU

Additional State Saved:

Table 4.15 Register States on a Coprocessor Unusable Exception

Register State	Value
$Cause_{CE}$	Unit number of the coprocessor being referenced

Entry Vector Used:

General exception vector (offset 0x180)

4.8.18 Execution Exception — DSP Module State Disabled

The DSP Module State Disabled exception is an execution exception. It occurs when an attempt is made to execute a DSP Module instruction when the MX bit in the *Status* register is not set. This allows an OS to do “lazy” context switching.

Cause Register ExcCode Value:

DSPDis

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.19 Execution Exception — Coprocessor 2 Exception

The Coprocessor 2 exception is one of the execution exceptions. A Coprocessor 2 exception occurs when a valid Coprocessor 2 instruction cause a general exception in the Coprocessor 2.

Cause Register ExcCode Value:

C2E

Additional State Saved:

Depending on the Coprocessor 2 implementation, additional state information of the exception can be saved in a Coprocessor 2 control register.

Entry Vector Used:

General exception vector (offset 0x180)

4.8.20 Execution Exception — Implementation-Specific 1 Exception

The Implementation-Specific 1 exception is one of the execution exceptions. An implementation-specific 1 exception occurs when a valid coprocessor 2 instruction cause an implementation-specific 1 exception in the Coprocessor 2.

Cause Register ExcCode Value:

IS1

Additional State Saved:

Depending on the coprocessor 2 implementation, additional state information of the exception can be saved in a coprocessor 2 control register.

Entry Vector Used:

General exception vector (offset 0x180)

4.8.21 Execution Exception — Integer Overflow

The integer overflow exception is one of the execution exceptions. An integer overflow exception occurs when selected integer instructions result in a 2's complement overflow.

Cause Register ExcCode Value:

Ov

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.22 Debug Data Break Exception

A debug data break exception occurs when a data hardware breakpoint matches the load/store transaction of an executed load/store instruction. The *DEPC* register and *DBD* bit in the *Debug* register will indicate the load/store instruction that caused the data hardware breakpoint to match. The load/store instruction that caused the debug exception has not completed e.g. not updated the register file, and the instruction can be re-executed after returning from the debug handler.

Debug Register Debug Status Bit Set:

DDBL for a load instruction or DDBS for a store instruction

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

4.8.23 Complex Break Exception

A complex data break exception occurs when the complex hardware breakpoint detects an enabled breakpoint. Complex breaks are taken imprecisely—the instruction that actually caused the exception is allowed to complete and the *DEPC* register and *DBD* bit in the *Debug* register point to a following instruction.

Debug Register Debug Status Bit Set:

DIBImpr, *DDBLImpr*, and/or *DDBSImpr*

Additional State Saved:

Debug2 fields indicate which type(s) of complex breakpoints were detected.

Entry Vector Used:

Debug exception vector

4.9 Exception Handling and Servicing Flowcharts

The remainder of this chapter contains flowcharts for the following exceptions and guidelines for their handlers:

- General exceptions and their exception handler
- Reset, soft reset and NMI exceptions, and a guideline to their handler
- Debug exceptions

Figure 4.3 General Exception Handler (HW)

Exceptions other than Reset, Soft Reset, NMI, Debug, SRAM ECC or SRAM Parity error.
Note: Interrupts can be masked by IE or IMs is masked if EXL = 1

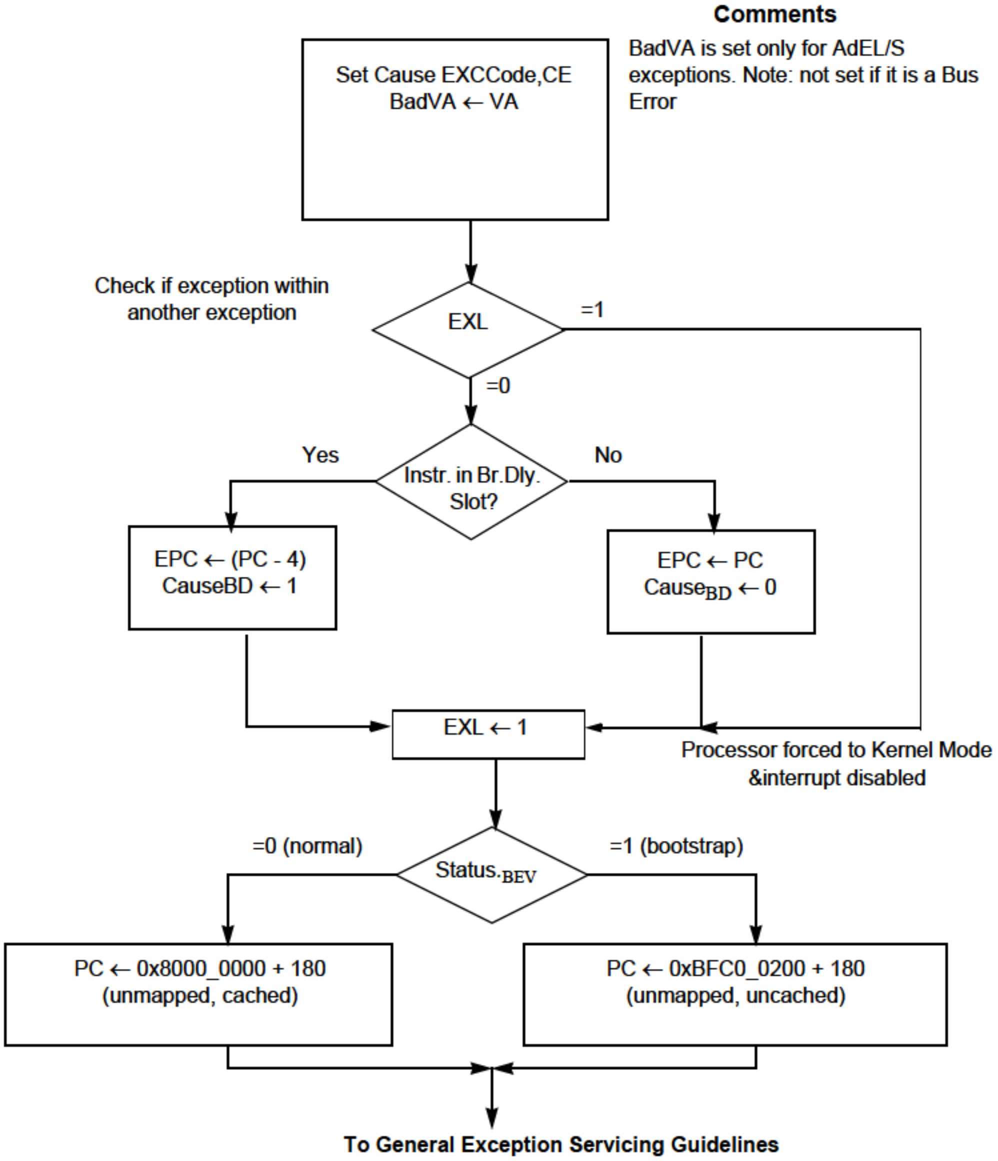


Figure 4.4 General Exception Servicing Guidelines (SW)

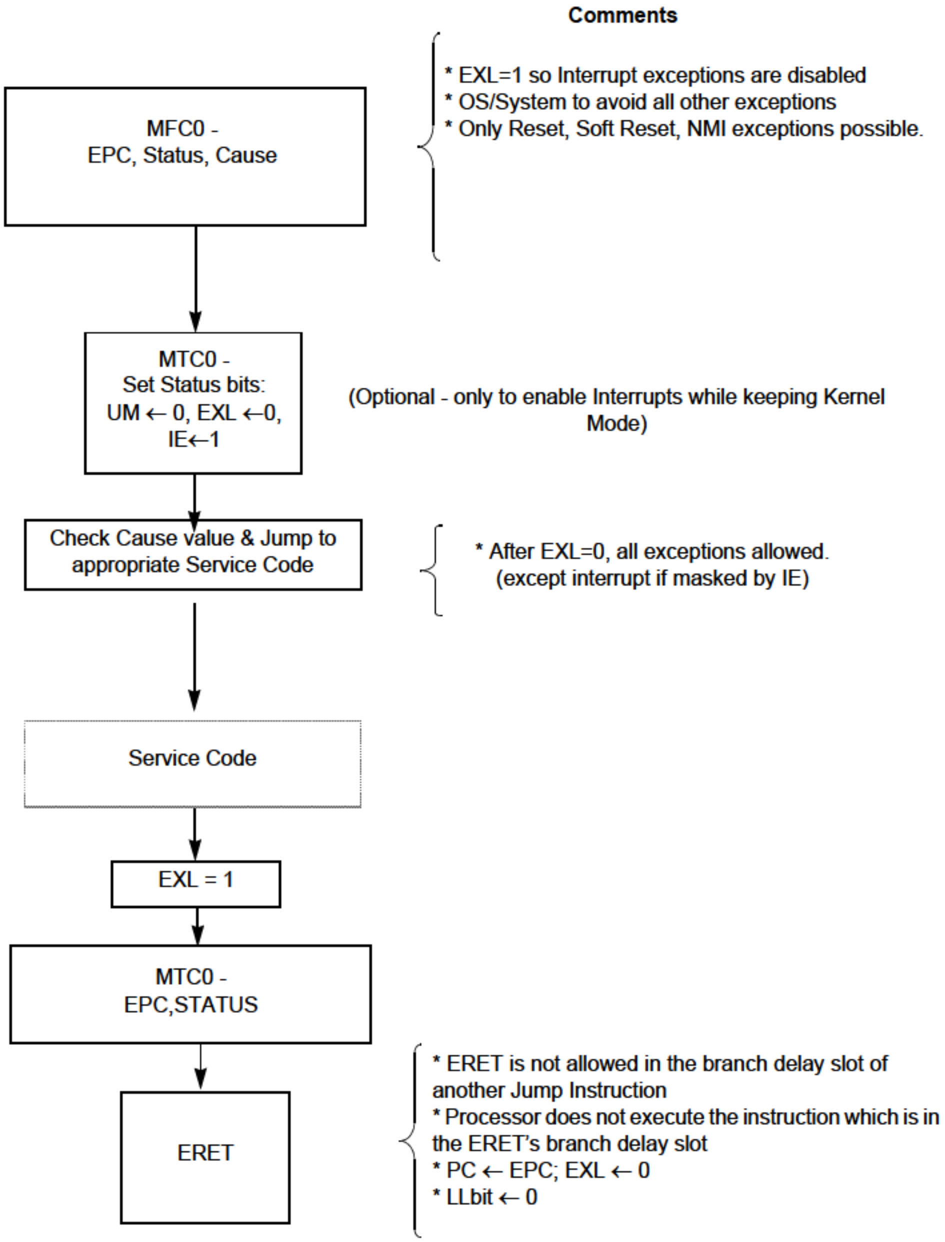
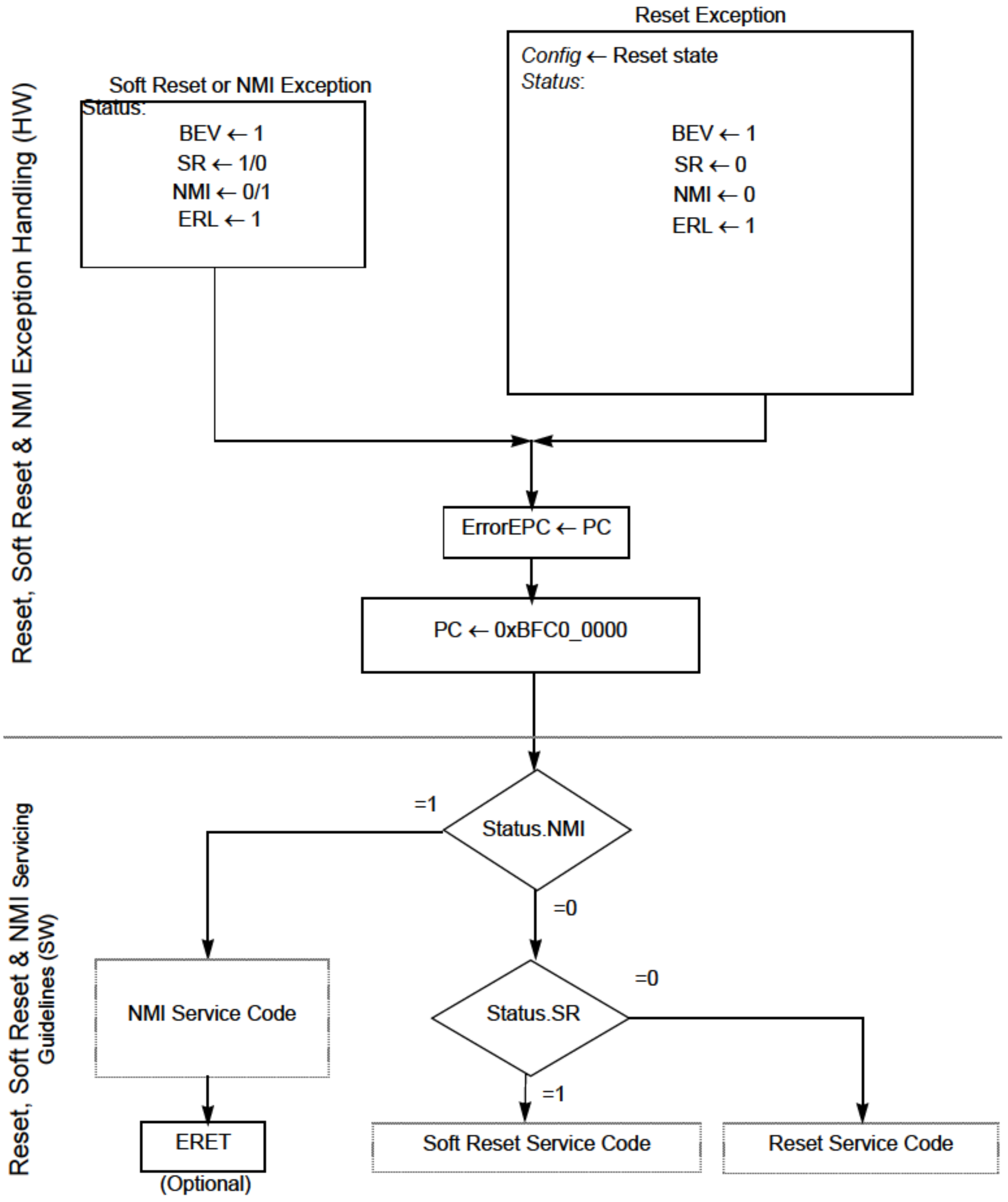

Comments
MFC0 -
EPC, Status, Cause
* EXL=1 so Interrupt exceptions are disabled
* OS/System to avoid all other exceptions
* Only Reset, Soft Reset, NMI exceptions possible.
MTC0 -
Set Status bits:
UM ← 0, EXL ←0,
IE←1
(Optional - only to enable Interrupts while keeping Kernel Mode)
Check Cause value & Jump to appropriate Service Code
* After EXL=0, all exceptions allowed.
(except interrupt if masked by IE)
Service Code
EXL = 1
MTC0 -
EPC,STATUS
ERET
* ERET is not allowed in the branch delay slot of another Jump Instruction
* Processor does not execute the instruction which is in the ERET's branch delay slot
* PC ← EPC; EXL ← 0
* LLbit ← 0

Figure 4.5 Reset, Soft Reset and NMI Exception Handling and Servicing Guidelines



Chapter 5

CP0 Registers of the M6200 Core

The System Control Coprocessor (CP0) provides the register interface to the M6200 processor core for the support of memory management, address translation, exception handling, and other privileged operations. Each CP0 register is identified by a *Register Number*, from 0 through 31, and a *Select Number* that is used as the value in the *sel* field of the MFC0 (Move From Coprocessor 0) and MTC0 (Move To Coprocessor 0) instructions. For example, the *EBase* register is Register Number 15, Select 1. After updating a CP0 register, there is a hazard period of zero or more instructions from the update by the MTC0 instruction until the update has taken effect in the core.

The Debug registers are described in Chapter 8, “Debug Support in the M6200 Core” on page 152.

5.1 CP0 Register Summary

Table 5.1 lists the CP0 registers in numerical order. Individual registers are described in Section 5.2 “CP0 Register Descriptions”.

Table 5.1 CP0 Registers

Register Number	Select Number	Register Name	Function
0-3		Reserved	Reserved in the M6200 core
4	2	*UserLocal*	User information that can be written by privileged software and read via *RDHWR* register 29
5-6		Reserved	Reserved in the M6200 core
7	0	*HWREna*	Enables access via the RDHWR instruction to selected hardware registers in non-privileged mode
8	0 1 2	*BadVAddr* *BadInstr* *BadInstrP*	Reports the address for the most recent address-related exception Reports the instruction that caused the most recent exception Reports the branch instruction if a delay slot or forbidden slot caused the most recent exception
9	0	*Count*	Processor cycle count
10	0	Reserved	Reserved in the M6200 core
11	0	*Compare*	Timer interrupt control
12	0 1 2 3 4 5	*Status* *IntCtl* *SRSCtl* *SRSMap1* *View_IPL* *SRSMAP2*	Processor status and control Interrupt system status and control Shadow Register Sets status and control Shadow set IPL mapping Contiguous view of IM and IPL fields Shadow set IPL mapping

Table 5.1 CP0 Registers (Continued)

Register Number	Select Number	Register Name	Function
13	0 4 5	*Cause* *View_RIPL* *NestedExc*	Cause of last exception Read access to *IP* ot *RIPL* field in *Cause* register Supports nested fault feature
14	0 2	*EPC* *NestedEPc*	Program counter at last exception Supports nested fault feature
15	0 1 2	*PRId*/ *EBase* *CDMMBase*	Processor identification and revision Exception base address Common Device Memory Map Base register
16	0 1 2 3 4 5 7	*Config* *Config1* *Config2* *Config3* *Config4* *Config5* *Config7*	Configuration registers
17-22		Reserved	Reserved in the M6200 core
17	0	*LLAddr*	Load linked address
23	0 3 6	*Debug* *UserTraceData1* *Debug2*	Debug register User Trace Data1 register Debug register 2
24	0 3	*DEPC* *UserTraceData2*	Program counter at last debug exception User Trace Data2 register
25	0 1 2 3	*PerfCtl0* *PerfCnt0* *PerfCtl1* *PerfCnt1*	Performance counter 0 control Performance counter 0 Performance counter 1control Performance counter 1
26	0	*ErrCtl*	Software parity check enable
27	0	*CacheErr*	Records information about SRAM ECC errors
27	1	*CacheErrAddr*	Records information about SRAM address of ECC errors
28-29		Reserved	Reserved in the M6200 core
30	0	*ErrorEPC*	Program counter at last error
31	0 2 3 4 5 6 7	*DeSAVE* *KScratch1* *Kscratch2* *Kscratch3* *Kscratch4* *Kscratch5* *Kscratch6*	Debug handler scratchpad register Scratch Register for Kernel Mode Scratch Register for Kernel Mode Scratch Register for Kernel Mode Scratch Register for Kernel Mode Scratch Register for Kernel Mode Scratch Register for Kernel Mode

5.2 CP0 Register Descriptions

This section contains descriptions of each CP0 register. The registers are listed in numerical order, first by Register Number, then by Select Number.

For each register described below, field descriptions include the read/write properties of the field (shown in Table 5.2) and the reset state of the field.

Table 5.2 CP0 Register R/W Field Types

<table>
<tr><th>Read/Write Notation</th><th>Hardware Interpretation</th><th>Software Interpretation</th></tr>
<tr><td>R/W</td><td colspan="2">A field in which all bits are readable and writable by software and, potentially, by hardware.
Hardware updates of this field are visible by software reads. Software updates of this field are visible by hardware reads.
If the reset state of this field is “Undefined,” either software or hardware must initialize the value before the first read will return a predictable value. This should not be confused with the formal definition of UNDEFINED behavior.</td></tr>
<tr><td>R</td><td>A field that is either static or is updated only by hardware.
If the Reset State of this field is either “0” or “Preset”, hardware initializes this field to zero or to the appropriate state, respectively, on powerup.
If the Reset State of this field is “Undefined”, hardware updates this field only under those conditions specified in the description of the field.</td><td>A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware.
If the Reset State of this field is “Undefined,” software reads of this field result in an UNPREDICTABLE value except after a hardware update done under the conditions specified in the description of the field.</td></tr>
<tr><td>W</td><td colspan="2">A field that can be written by software but which can not be read by software.
Software reads of this field will return an UNDEFINED value.</td></tr>
<tr><td>0</td><td>A field that hardware does not update, and for which hardware can assume a zero value.</td><td>Software reads of this field return zero. Software writes of non-zero values to this field are ignored.</td></tr>
</table>

5.2.1 UserLocal Register (CP0 Register 4, Select 2)

The *UserLocal* register is a read-write register that is not interpreted by the hardware and conditionally readable via the RDHWR instruction.

Figure 5.1 shows the format of the *UserLocal* register; Table 5.3 describes the *UserLocal* register fields.

Figure 5.1 UserLocal Register Format

31 0
UserLocal

Table 5.3 UserLocal Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
UserLocal	31:0	This field contains software information that is not interpreted by hardware.	R/W	Undefined

Programming Notes

Privileged software may write this register with arbitrary information and make it accessible to unprivileged software via register 29 (*ULR*) of the RDHWR instruction. To do so, bit 29 of the *HWREna* register must be set to 1 to enable unprivileged access to the register. In some operating environments, the *UserLocal* register contains a pointer to a thread-specific storage block that is obtained via the *RDHWR* register.

5.2.2 HWREna Register (CP0 Register 7, Select 0)

The *HWREna* register contains a bit mask that determines which hardware registers are accessible via the RDHWR instruction when that instruction is executed in a mode in which Coprocessor 0 is not enabled.

Figure 5.2 shows the format of the *HWREna* Register; Table 5.4 describes the *HWREna* register fields.

Figure 5.2 HWREna Register Format

31 30	29	28 4	3 0
0	ULR	0	Mask

Table 5.4 HWREna Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:30	Must be written with zero; returns zero on read	0	0
0	28:4	Must be written with zero; returns zero on read	0	0
ULR	29	User Local Register. This register provides read access to the coprocessor 0 *UserLocal* register. In some operating environments, the *UserLocal* register is a pointer to a thread-specific storage block.	R/W	0
Mask	3:0	Each bit in this field enables access by the RDHWR instruction to a particular hardware register (which may not be an actual register). If bit ‘n’ in this field is a 1, access is enabled to hardware register ‘n’. If bit ‘n’ of this field is a 0, access is disabled. See the RDHWR instruction for a list of valid hardware registers.	R/W	0

Privileged software may determine which of the hardware registers are accessible by the RDHWR instruction. In doing so, a register may be virtualized at the cost of handling a Reserved Instruction Exception, interpreting the instruction, and returning the virtualized value. For example, if it is not desirable to provide direct access to the *Count* register, access to that register may be individually disabled and the return value can be virtualized by the operating system.

5.2.3 BadVAddr Register (CP0 Register 8, Select 0)

The *BadVAddr* register is a read-only register that captures the most recent virtual address that caused the following exception:

- Address error (AdEL or AdES)
- Memory Protection exception

The *BadVAddr* register does not capture address information for bus errors, because they are not addressing errors.

Figure 5.3 BadVAddr Register Format

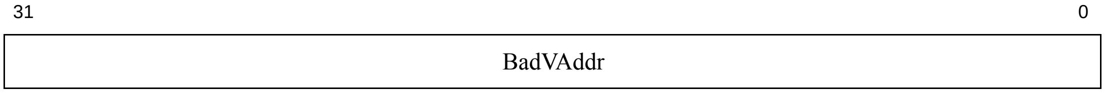


Table 5.5 BadVAddr Register Field Description

Fields				
Name	**Bits**	**Description**	**Read/Write**	**Reset State**
BadVAddr	31:0	Bad virtual address.	R	Undefined

5.2.4 BadInstr Register (CP0 Register 8, Select 1)

The *BadInstr* register is a read-only register that captures the most recent instruction that caused one of the following exceptions:

- Execution Exception:

  Integer Overflow, Trap, System Call, Breakpoint, Floating-point, Coprocessor 2 exception, Memory Protection exception (applicable to data-triggered memory protection exceptions only), Coprocessor Unusable, Reserved Instruction

- Addressing:

  Address Error, TLB Refill, TLB Invalid, TLB Read Inhibit, TLB Execute Inhibit, TLB Modified

The *BadInstr* register is provided to allow acceleration of instruction emulation. The *BadInstr* register is only set by exceptions that are synchronous to an instruction. The *BadInstr* register is not set by interrupts or by NMI, Machine check, Bus Error, SRAM ECC error, SRAM Parity error, or Debug exceptions.

When a synchronous exception occurs for which there is no valid instruction word (for example TLB Refill - Instruction Fetch), the value stored in *BadInstr* is **UNPREDICTABLE**.

Presence of the *BadInstr* register is indicated by the $Config3_{BI}$ bit.

Figure 5.4 shows the proposed format of the *BadInstr* register; Table 5.6 describes the *BadInstr* register fields.

Figure 5.4 BadInstr Register Format

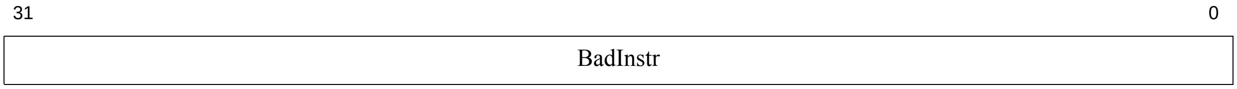


Table 5.6 BadInstr Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
BadInstr	31:0	Faulting instruction word. Instruction words smaller than 32 bits are placed in bits 15:0, with bits 31:16 containing zero.	R	Undefined

5.2.5 BadInstrP Register (CP0 Register 8, Select 2)

The *BadInstrP* register is an optional register that is used in conjunction with the *BadInstr* register. The *BadInstrP* register contains the prior branch instruction when the faulting instruction is in a branch delay slot or forbidden slot.

The *BadInstrP* register is updated for these exceptions:

- Execution Exception

  Integer Overflow, Trap, System Call, Breakpoint, Floating-point, Coprocessor 2 exception, Memory Protection exception (applicable to data-triggered memory protection exceptions only), Coprocessor Unusable, Reserved Instruction

- Addressing

  Address Error, TLB Refill, TLB Invalid, TLB Read Inhibit, TLB Execute Inhibit, TLB Modified

The *BadInstrP* register is provided to allow acceleration of instruction emulation. The *BadInstrP* register is only set by exceptions that are synchronous to an instruction. The *BadInstrP* register is not set by Interrupts or by NMI, Machine check, Bus Error, SRAM ECC error, SRAM Parity error, or Debug exceptions. When a synchronous exception occurs, and the faulting instruction is not in a branch delay slot or forbidden slot, then the value stored in *BadInstrP* is **UNPREDICTABLE**.

Presence of the *BadInstrP* register is indicated by the $Config3_{BP}$ bit. The *BadInstrP* register is instantiated per-VPE in an MT ASE processor.

Figure 5.5 shows the proposed format of the *BadInstrP* register; Table 5.7 describes the *BadInstrP* register fields.

Figure 5.5 BadInstrP Register Format

31	0
BadInstrP	

Table 5.7 BadInstrP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
BadInstrP	31:0	Prior branch instruction. Instruction words smaller than 32 bits are placed in bits 15:0, with bits 31:16 containing zero.	R	Undefined

5.2.6 Count Register (CP0 Register 9, Select 0)

The *Count* register acts as a timer, incrementing at a constant rate, whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. The counter increments every other clock if the *DC* bit in the *Cause* register is 0.

The *Count* register can be written for functional or diagnostic purposes, including at reset or to synchronize processors.

By writing the *CountDM* bit in the *Debug* register, it is possible to control whether the *Count* register continues incrementing while the processor is in debug mode.

Figure 5.6 Count Register Format

31	0
Count	

Table 5.8 Count Register Field Description

Fields		Description	Read/Write	Reset State
Name	Bits			
Count	31:0	Interval counter.	R/W	Undefined

5.2.7 Compare Register (CP0 Register 11, Select 0)

The *Compare* register acts in conjunction with the *Count* register to implement a timer and timer interrupt function. The timer interrupt is an output of the cores. The *Compare* register maintains a stable value and does not change on its own.

When the value of the *Count* register equals the value of the *Compare* register, the *SI_TimerInt* pin is asserted. This pin will remain asserted until the *Compare* register is written. The *SI_TimerInt* pin can be fed back into the core on one of the interrupt pins to generate an interrupt.

For diagnostic purposes, the *Compare* register is a read/write register. In normal use, however, the *Compare* register is write-only. Writing a value to the *Compare* register, as a side effect, clears the timer interrupt.

Figure 5.7 Compare Register Format

31 0
Compare

Table 5.9 Compare Register Field Description

Fields				
Name	**Bit(s)**	**Description**	**Read/Write**	**Reset State**
Compare	31:0	Interval count compare value.	R/W	Undefined

5.2.8 Status Register (CP0 Register 12, Select 0)

The *Status* register is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. Fields of this register combine to create operating modes for the processor. Refer to 3.2 "Modes of Operation" on page 38 for a discussion of operating modes, and 4.3 "Interrupts" on page 51 for a discussion of interrupt modes.

Interrupt Enable: Interrupts are enabled when all of the following conditions are true:

- *IE* = 1
- *EXL* = 0
- *ERL* = 0
- *DM* = 0

If these conditions are met, then the settings of the *IM* and *IE* bits enable the interrupts.

Operating Modes: If the *DM* bit in the *Debug* register is 1, then the processor is in debug mode; otherwise the processor is in either kernel or user mode. The following CPU *Status* register bit settings determine user or kernel mode:

- User mode: *UM* = 1, *EXL* = 0, and *ERL* = 0
- Kernel mode: *UM* = 0, or *EXL* = 1, or *ERL* = 1

Coprocessor Accessibility: The *Status* register *CU* bits control coprocessor accessibility. If any coprocessor is unusable, then an instruction that accesses it generates an exception.

Figure 5.8 shows the format of the *Status* register; Table 5.10 describes the *Status* register fields.

Figure 5.8 Status Register Format

31 29	28	27 25	24	23	22	21	20	19	18	17	16 10	9 8	7 6 5	4	3	2	1	0
CU3..CU1	RW	0	MX	R	BEV	R	SR	NMI	IM9	R	IM8..IM2	IM1..IM0	R	UM	R	ERL	EXL	IE
									IPL		IPL							

Table 5.10 Status Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>CU3</td><td>31</td><td>Controls access to Coprocessor 3. COP3 is not supported. This bit cannot be written and will read as 0.</td><td>R</td><td>0</td></tr>
<tr><td>CU2</td><td>30</td><td>Controls access to Coprocessor 2. This bit can only be written if coprocessor is attached to the COP2 interface. (<i>C2</i> bit in Config1 is set). This bit will read as 0 if no coprocessor is present.</td><td>R/W</td><td>0</td></tr>
<tr><td>CU1</td><td>29</td><td>Controls access to Coprocessor 1. This bit cannot be written and always reads as 0, because an FPU is not supported.</td><td>R</td><td>0</td></tr>
<tr><td>RW</td><td>28</td><td>Read/write field. This bit can be written by software without side-effects. For example, the kernel can set this bit to signify that the exception condition is due to user code, before saving <i>Status</i> to the stack in memory.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>27:25</td><td>Reserved. This bit must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>MX</td><td>24</td><td>MIPS DSP Extension. Enables access to DSP Module resources:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Access not allowed</td></tr>
<tr><td>1</td><td>Access allowed</td></tr>
</table>
An attempt to execute any DSP Module instruction before this bit has been set to 1 will cause a DSP State Disabled exception. The state of this bit is reflected in $Config3_{DSPP}$.</td><td>R/W</td><td>0</td></tr>
<tr><td>R</td><td>23</td><td>Reserved. This field is ignored on writes and reads as 0.</td><td>0</td><td>0</td></tr>
<tr><td>BEV</td><td>22</td><td>Controls the location of exception vectors:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Normal</td></tr>
<tr><td>1</td><td>Bootstrap</td></tr>
</table></td><td>R/W</td><td>1</td></tr>
<tr><td>R</td><td>21</td><td>Reserved. This bit must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
<tr><td>SR</td><td>20</td><td>Indicates that the entry through the reset exception vector was due to a Soft Reset:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Not Soft Reset (NMI or Reset)</td></tr>
<tr><td>1</td><td>Soft Reset</td></tr>
</table>
Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.</td><td>R/W</td><td>1 for Soft Reset; 0 otherwise</td></tr>
</table>

Table 5.10 Status Register Field Descriptions (Continued)

<table>
<thead>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
</thead>
<tbody>
<tr><td>NMI</td><td>19</td><td>Indicates that the entry through the reset exception vector was due to an NMI:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Not NMI (Soft Reset or Reset)</td></tr><tr><td>1</td><td>NMI</td></tr></table>
Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.</td><td>R/W</td><td>1 for NMI; 0 otherwise</td></tr>
<tr><td>R</td><td>17</td><td>Reserved. This field is ignored on writes and reads as 0.</td><td>0</td><td>0</td></tr>
<tr><td>IM9:IM2</td><td>18,
16:10</td><td>Interrupt Mask: Controls the enabling of each of the hardware interrupts. Refer to 4.3 “Interrupts” on page 51 for a complete discussion of enabled interrupts.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupt request disabled</td></tr><tr><td>1</td><td>Interrupt request enabled</td></tr></table>
In implementations in which EIC interrupt mode is enabled ($\mathit{Config3}_{VEIC}$ = 1), these bits have a different meaning and are interpreted as the <i>IPL</i> field, described below.</td><td>R/W</td><td>Undefined for IM7:IM2
0 for IM9:IM8</td></tr>
<tr><td>IPL</td><td>18,
16:10</td><td>Interrupt Priority Level.
In implementations in which EIC interrupt mode is enabled ($\mathit{Config3}_{VEIC}$ = 1), this field is the encoded (0:255) value of the current IPL. An interrupt will be signaled only if the requested IPL is higher than this value.
If EIC interrupt mode is not enabled ($\mathit{Config3}_{VEIC}$ = 0), these bits have a different meaning and are interpreted as the <i>IM7</i>..<i>IM2</i> bits, described above.</td><td>R/W</td><td>Undefined for IPL15:IPL10
0 for IPL18:IPL17</td></tr>
<tr><td>IM1:IM0</td><td>9:8</td><td>Interrupt Mask: Controls the enabling of each of the software interrupts. Refer to Section 4.3 “Interrupts”for a complete discussion of enabled interrupts.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupt request disabled</td></tr><tr><td>1</td><td>Interrupt request enabled</td></tr></table>
In implementations of in which EIC interrupt mode is enabled, these bits are writable, but have no effect on the interrupt system.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>7:5</td><td>Reserved. This field is ignored on writes and reads as 0.</td><td>R</td><td>0</td></tr>
</tbody>
</table>

Table 5.10 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>UM</td><td>4</td><td>This bit denotes the base operating mode of the processor. See Section 3.2 “Modes of Operation” for a full discussion of operating modes. The encoding of this bit is:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Base mode is Kernel Mode</td></tr>
<tr><td>1</td><td>Base mode is User Mode</td></tr>
</table>
Note that the processor can also be in kernel mode if *ERL* or *EXL* is set, regardless of the state of the *UM* bit.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>3</td><td>This bit is reserved. This bit is ignored on writes and reads as zero.</td><td>R</td><td>0</td></tr>
<tr><td>ERL</td><td>2</td><td>Error Level; Set by the processor when a Reset, Soft Reset, NMI or SRAM ECC error, SRAM parity error exception is taken.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Normal level</td></tr>
<tr><td>1</td><td>Error level</td></tr>
</table>
When *ERL* is set:
• The processor is running in kernel mode
• Interrupts are disabled
• The ERET instruction will use the return address held in *ErrorEPC* instead of *EPC*
• The lower 2^{29} bytes of kuseg are treated as an unmapped and uncached region. See Chapter 3, “Memory Management of the M6200 Core” on page 37. This allows main memory to be accessed in the presence of SRAM Interface Parity or ECC errors. The operation of the processor is **UNDEFINED** if the *ERL* bit is set while the processor is executing instructions from kuseg.</td><td>R/W</td><td>1</td></tr>
<tr><td>EXL</td><td>1</td><td>Exception Level; Set by the processor when any exception other than Reset, Soft Reset, NMI, and parity or ECC error exceptions is taken.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Normal level</td></tr>
<tr><td>1</td><td>Exception level</td></tr>
</table>
When *EXL* is set:
• The processor is running in Kernel Mode
• Interrupts are disabled.
• *EPC*, $Cause_{BD}$ and *SRSCtl* will not be updated if another exception is taken.</td><td>R/W</td><td>Undefined</td></tr>
</table>

Table 5.10 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IE</td><td>0</td><td>Interrupt Enable: Acts as the master enable for software and hardware interrupts:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupts are disabled</td></tr>
<tr><td>1</td><td>Interrupts are enabled</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
</table>

5.2.9 IntCtl Register (CP0 Register 12, Select 1)

The *IntCtl* register controls the interrupt capabilities of the M6200 core, including vectored interrupts and support for an external interrupt controller.

If vectored interrupts are not implemented (*Config3*$_{VInt}$ = 0 and *Config3*$_{VEIC}$ = 0), the *IPTI* and *IPPCI* fields must be implemented as read-only value, but the remaining bits of this register may be implemented as an ignore-revision-on-write, read-as-zeros register.

Figure 5.9 shows the format of the *IntCtl* register; Table 5.11 describes the *IntCtl* register fields.

Figure 5.9 IntCtl Register Format

31 29	28 26	25 23	22	21	20 16	15	14	13	12 10	9 5	4 0
IPTI	IPPCI	IPFDC	PF	ICE	StkDec	Clr EXL	APE	Use KStk	0	VS	0

Table 5.11 IntCtl Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IPTI</td><td>31:29</td><td>For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Timer Interrupt request is merged, and allows software to determine whether to consider $Cause_{TI}$ for a potential interrupt.
<table>
<tr><th>Encoding</th><th>IP bit</th><th>Hardware Interrupt Source</th></tr>
<tr><td>2</td><td>2</td><td>HW0</td></tr>
<tr><td>3</td><td>3</td><td>HW1</td></tr>
<tr><td>4</td><td>4</td><td>HW2</td></tr>
<tr><td>5</td><td>5</td><td>HW3</td></tr>
<tr><td>6</td><td>6</td><td>HW4</td></tr>
<tr><td>7</td><td>7</td><td>HW5</td></tr>
</table>
The value of this bit is set by the static input, *SI_IPTI[2:0]*. This allows external logic to communicate the specific *SI_Int* hardware interrupt.
The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode.</td><td>R</td><td>Externally Set</td></tr>
<tr><td>IPPCI</td><td>28:26</td><td>For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Performance Counter Interrupt request is merged, and allows software to determine whether to consider $Cause_{PCI}$ for a potential interrupt.
<table>
<tr><th>Encoding</th><th>IP bit</th><th>Hardware Interrupt Source</th></tr>
<tr><td>2</td><td>2</td><td>HW0</td></tr>
<tr><td>3</td><td>3</td><td>HW1</td></tr>
<tr><td>4</td><td>4</td><td>HW2</td></tr>
<tr><td>5</td><td>5</td><td>HW3</td></tr>
<tr><td>6</td><td>6</td><td>HW4</td></tr>
<tr><td>7</td><td>7</td><td>HW5</td></tr>
</table>
The value of this bit is set by the static input, *SI_IPPCI[2:0]*. This allows external logic to communicate the specific *SI_Int* hardware interrupt.
The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode.</td><td>R</td><td>Preset or Externally Set</td></tr>
</table>

Table 5.11 IntCtl Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IPFDC</td><td>25:23</td><td>For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Fast Debug Channel Interrupt request is merged, and allows software to determine whether to consider $Cause_{FDC}$ for a potential interrupt.
<table>
<tr><th>Encoding</th><th>IP bit</th><th>Hardware Interrupt Source</th></tr>
<tr><td>2</td><td>2</td><td>HW0</td></tr>
<tr><td>3</td><td>3</td><td>HW1</td></tr>
<tr><td>4</td><td>4</td><td>HW2</td></tr>
<tr><td>5</td><td>5</td><td>HW3</td></tr>
<tr><td>6</td><td>6</td><td>HW4</td></tr>
<tr><td>7</td><td>7</td><td>HW5</td></tr>
</table>
The value of this field is **UNPREDICTABLE** if External Interrupt Controller Mode is both implemented and enabled. The external interrupt controller is expected to provide this information for that interrupt mode.
If FDC is not implemented, this field returns zero on read.</td><td>R</td><td>Preset or Externally Set</td></tr>
<tr><td>PF</td><td>22</td><td>Enables Vector Prefetching Feature.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Vector Prefetching disabled.</td></tr>
<tr><td>1</td><td>Vector Prefetching enabled.</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>ICE</td><td>21</td><td>For IRET instruction. Enables Interrupt Chaining.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupt Chaining disabled</td></tr>
<tr><td>1</td><td>Interrupt Chaining enabled</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>StkDec</td><td>20:16</td><td>For Auto-Prologue feature. This is the number of 4-byte words that is decremented from the value of GPR29.
<table>
<tr><th>Encoding</th><th>Decrement Amount in Words</th><th>Decrement Amount in Bytes</th></tr>
<tr><td>0-3</td><td>3</td><td>12</td></tr>
<tr><td>Others</td><td>As encoded, e.g. 0x5 means 5 words</td><td>4 * encoded value
e.g. 0x5 means 20 bytes</td></tr>
</table></td><td>R/W</td><td>0x3</td></tr>
</table>

Table 5.11 IntCtl Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>ClrEXL</td><td>15</td><td>For Auto-Prologue feature and IRET instruction.
If set, during Auto-Prologue and IRET interrupt chaining, the <i>KSU</i>/<i>ERL</i>/<i>EXL</i> fields are cleared.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Fields are not cleared by these operations.</td></tr><tr><td>1</td><td>Fields are cleared by these operations.</td></tr></table></td><td>R/W</td><td>0</td></tr>
<tr><td>APE</td><td>14</td><td>Enables Auto-Prologue feature.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Auto-Prologue disabled</td></tr><tr><td>1</td><td>Auto-Prologue enabled</td></tr></table></td><td>R/W</td><td>0</td></tr>
<tr><td>UseKStk</td><td>13</td><td>Chooses which Stack to use during Interrupt Automated Prologue.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Copy $29 of the Previous SRS to the Current SRS at the beginning of IAP.

This is used for Bare-Iron environments with only one stack.</td></tr><tr><td>1</td><td>Use $29 of the Current SRS at the beginning of IAP.
This is used for environments where there are separate User-mode and Kernel mode stacks. In this case, $29 of the SRS used during IAP must be pre-initialized by software to hold the Kernel mode stack pointer.</td></tr></table></td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>12:10</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
</table>

Table 5.11 IntCtl Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>VS</td><td>9:5</td><td>Vector Spacing. If vectored interrupts are implemented (as denoted by $Config3_{VInt}$ or $Config3_{VEIC}$), this field specifies the spacing between vectored interrupts.
<table>
<tr><th>Encoding</th><th>Spacing Between Vectors (hex)</th><th>Spacing Between Vectors (decimal)</th></tr>
<tr><td>16#00</td><td>16#000</td><td>0</td></tr>
<tr><td>16#01</td><td>16#020</td><td>32</td></tr>
<tr><td>16#02</td><td>16#040</td><td>64</td></tr>
<tr><td>16#04</td><td>16#080</td><td>128</td></tr>
<tr><td>16#08</td><td>16#100</td><td>256</td></tr>
<tr><td>16#10</td><td>16#200</td><td>512</td></tr>
</table>
All other values are reserved. The operation of the processor is <b>UNDEFINED</b> if a reserved value is written to this field.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>4:0</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
</table>

5.2.10 SRSCtl Register (CP0 Register 12, Select 2)

The *SRSCtl* register controls the operation of GPR shadow sets in the processor.

Figure 5.10 shows the format of the *SRSCtl* register; Table 5.12 describes the *SRSCtl* register fields.

Figure 5.10 SRSCtl Register Format

31 30	29 26	25 22	21 18	17 16	15 12	11 10	9 6	5 4	3 0
0	HSS	0	EICSS	0	ESS	0	PSS	0	CSS

Table 5.12 SRSCtl Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:30	Must be written as zeros; returns zero on read.	0	0

Table 5.12 SRSCtl Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>HSS</td><td>29:26</td><td>Highest Shadow Set. This field contains the highest shadow set number that is implemented by this processor. A value of zero in this field indicates that only the normal GPRs are implemented.
Possible values of this field for the M6200 processor are:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>One shadow set (normal GPR set) is present.</td></tr>
<tr><td>1</td><td>Two shadow sets are present.</td></tr>
<tr><td>3</td><td>Four shadow sets are present.</td></tr>
<tr><td>7</td><td>Eight shadow sets are present</td></tr>
<tr><td>15</td><td>Sixteen shadow sets are present</td></tr>
<tr><td>2, 4-6, 8-14</td><td>Reserved</td></tr>
</table>
The value in this field also represents the highest value that can be written to the *ESS*, *EICSS*, *PSS*, and *CSS* fields of this register, or to any of the fields of the *SRSMap* register. The operation of the processor is **UNDEFINED** if a value larger than the one in this field is written to any of these other fields.</td><td>R</td><td>Preset</td></tr>
<tr><td>0</td><td>25:22</td><td>Must be written as zeros; returns zero on read.</td><td>0</td><td>0</td></tr>
<tr><td>EICSS</td><td>21:18</td><td>EIC interrupt mode shadow set. If $Config3_{VEIC}$ is 1 (EIC interrupt mode is enabled), this field is loaded from the external interrupt controller for each interrupt request and is used in place of the *SRSMap* register to select the current shadow set for the interrupt.
See Section 4.3.1 “Interrupt Modes” for a discussion of EIC interrupt mode. If $Config3_{VEIC}$ is 0, this field must be written as zero, and returns zero on read.</td><td>R</td><td>Undefined</td></tr>
<tr><td>0</td><td>17:16</td><td>Must be written as zeros; returns zero on read.</td><td>0</td><td>0</td></tr>
<tr><td>ESS</td><td>15:12</td><td>Exception Shadow Set. This field specifies the shadow set to use on entry to Kernel Mode caused by any exception other than a vectored interrupt.
The operation of the processor is <b>UNDEFINED</b> if software writes a value into this field that is greater than the value in the <i>HSS</i> field.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>11:10</td><td>Must be written as zeros; returns zero on read.</td><td>0</td><td>0</td></tr>
</table>

Table 5.12 SRSCtl Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
PSS	9:6	Previous Shadow Set. If GPR shadow registers are implemented, and with the exclusions noted in the next paragraph, this field is copied from the *CSS* field when an exception or interrupt occurs. An ERET instruction copies this value back into the *CSS* field if $Status_{BEV}$ = 0. This field is not updated on any exception which sets $Status_{ERL}$ to 1 (Reset, Soft Reset, NMI, SRAM Interface Parity or ECC error), an entry into Debug mode, or any exception or interrupt that occurs with $Status_{EXL}$ = 1, or $Status_{BEV}$ = 1. This field is not updated on an exception that occurs while $Status_{ERL}$ = 1. The operation of the processor is **UNDEFINED** if software writes a value into this field that is greater than the value in the *HSS* field.	R/W	0
0	5:4	Must be written as zeros; returns zero on read.	0	0
CSS	3:0	Current Shadow Set. If GPR shadow registers are implemented, this field is the number of the current GPR set. With the exclusions noted in the next paragraph, this field is updated with a new value on any interrupt or exception, and restored from the *PSS* field on an ERET. Table 5.13 describes the various sources from which the *CSS* field is updated on an exception or interrupt. This field is not updated on any exception which sets $Status_{ERL}$ to 1 (Reset, Soft Reset, NMI, SRAM Interface Parity or ECC error), an entry into Debug mode, or any exception or interrupt that occurs with $Status_{EXL}$ = 1, or $Status_{BEV}$ = 1. Neither is it updated on an ERET with $Status_{ERL}$ = 1 or $Status_{BEV}$ = 1. This field is not updated on an exception that occurs while $Status_{ERL}$ = 1. The value of *CSS* can be changed directly by software only by writing the *PSS* field and executing an ERET instruction.	R	0

5.2.11 SRSMap1 Register (CP0 Register 12, Select 3)

The *SRSMap* register contains 8, 4-bit fields that provide the mapping from a vector number to the shadow set number to use when servicing such an interrupt. The values from this register are not used for a non-interrupt exception, or a non-vectored interrupt ($Cause_{IV}$ = 0 or $IntCtl_{VS}$ = 0). In such cases, the shadow set number comes from $SRSCtl_{ESS}$.

If $SRSCtl_{HSS}$ is zero, the results of a software read or write of this register are **UNPREDICTABLE**.

The operation of the processor is **UNDEFINED** if a value is written to any field in this register that is greater than the value of $SRSCtl_{HSS}$.

The *SRSMap* register contains the shadow register set numbers for vector numbers 7:0. The same shadow set number can be established for multiple interrupt vectors, creating a many-to-one mapping from a vector to a single shadow register set number.

Table 5.13 Sources for RSCtl$_{CSS}$ on an Exception or Interrupt

Exception Type	Condition	SRSCtl$_{CSS}$ Source	Comment
Exception	All	*SRSCtl*$_{ESS}$	
Non-Vectored Interrupt	*Cause*$_{IV}$ = 0	*SRSCtl*$_{ESS}$	Treat as exception
Vectored Interrupt	*Cause*$_{IV}$ = 1 and *Config3*$_{VEIC}$ = 0 and *Config3*$_{VInt}$ = 1	*SRSMap*$_{VECTNUM}$	Source is internal map register. (for `VECTNUM` see Table 4.3)
Vectored EIC Interrupt	*Cause*$_{IV}$ = 1 and *Config3*$_{VEIC}$ = 1	*SRSCtl*$_{EICSS}$	Source is external interrupt controller.

Figure 5.11 shows the format of the *SRSMap* register; Table 5.14 describes the *SRSMap* register fields.

Figure 5.11 SRSMap Register Format

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
SSV7	SSV6	SSV5	SSV4	SSV3	SSV2	SSV1	SSV0

Table 5.14 SRSMap Register Field Descriptions

Fields				
Name	Bits	Description	Read/Write	Reset State
SSV7	31:28	Shadow register set number for Vector Number 7	R/W	0
SSV6	27:24	Shadow register set number for Vector Number 6	R/W	0
SSV5	23:20	Shadow register set number for Vector Number 5	R/W	0
SSV4	19:16	Shadow register set number for Vector Number 4	R/W	0
SSV3	15:12	Shadow register set number for Vector Number 3	R/W	0
SSV2	11:8	Shadow register set number for Vector Number 2	R/W	0
SSV1	7:4	Shadow register set number for Vector Number 1	R/W	0
SSV0	3:0	Shadow register set number for Vector Number 0	R/W	0

5.2.12 View_IPL Register (CP0 Register 12, Select 4)

Figure 5.12 View_IPL Register Format

31 10	9 0
0	IM
	IPL

Table 5.15 View_IPL Register Field Descriptions

Fields				
Name	Bits	Description	Read / Write	Reset State
IM	9:0	Interrupt Mask. If EIC interrupt mode is not enabled, controls which interrupts are enabled.	R/W	Undefined for IM7:IM2 0 for IM9:IM8
IPL	9:2	Interrupt Priority Level. If EIC interrupt mode is enabled, this field is the encoded value of the current *IPL*.	R/W	Undefined
0	31:10, 1:0	Must be written as zero; returns zero on read.	0	0

This register gives read and write access to the *IM* or *IPL* field that is also available in the *Status* Register. The use of this register allows the Interrupt Mask or the Priority Level to be read/written without extracting/inserting that bit field from/to the *Status* Register.

The *IPL* field might be located in non-contiguous bits within the *Status* Register. All of the *IPL* bits are presented as a contiguous field within this register.

5.2.13 SRSMap2 Register (CP0 Register 12, Select 5)

The *SRSMap2* register contains 2 4-bit fields that provide the mapping from an vector number to the shadow set number to use when servicing such an interrupt. The values from this register are not used for a non-interrupt exception, or a non-vectored interrupt ($Cause_{IV}$ = 0 or $IntCtl_{VS}$ = 0). In such cases, the shadow set number comes from $SRSCtl_{ESS}$.

If $SRSCtl_{HSS}$ is zero, the results of a software read or write of this register are **UNPREDICTABLE**.

The operation of the processor is **UNDEFINED** if a value is written to any field in this register that is greater than the value of $SRSCtl_{HSS}$.

The *SRSMap2* register contains the shadow register set numbers for vector numbers 9:8. The same shadow set number can be established for multiple interrupt vectors, creating a many-to-one mapping from a vector to a single shadow register set number.

Figure 5.13 shows the format of the *SRSMap2* register; Table 5.16 describes the *SRSMap2* register fields.

Figure 5.13 SRSMap Register Format

31 8	7 4	3 0
0	SSV9	SSV8

Table 5.16 SRSMap Register Field Descriptions

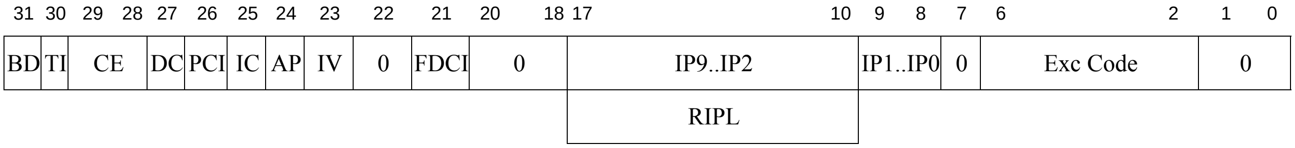
Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:8	Must be written as zero; returns zero on read.	R	0
SSV9	7:4	Shadow register set number for Vector Number 9	R/W	0
SSV8	3:0	Shadow register set number for Vector Number 8	R/W	0

5.2.14 Cause Register (CP0 Register 13, Select 0)

The *Cause* register primarily describes the cause of the most recent exception. In addition, fields also control software interrupt requests and the vector through which interrupts are dispatched. With the exception of the *IP1..0*, *DC*, *IV*, and *WP* fields, all fields in the *Cause* register are read-only. In the optional External Interrupt Controller (EIC) interrupt mode, in which *IP7..2* are interpreted as the Requested Interrupt Priority Level (RIPL).

Figure 5.14 shows the format of the *Cause* register; Table 5.17 describes the *Cause* register fields.

Figure 5.14 Cause Register Format



31	30	29 28	27	26	25	24	23	22	21	20 18	17 10	9 8	7	6 2	1 0
BD	TI	CE	DC	PCI	IC	AP	IV	0	FDCI	0	IP9..IP2	IP1..IP0	0	Exc Code	0
											RIPL				

Table 5.17 Cause Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
BD	31	Indicates whether the last exception taken occurred in a branch delay slot or forbidden slot: **Encoding** / **Meaning** 0 — Not in delay slot 1 — In delay slot The processor updates *BD* only if $\mathit{Status}_{\mathit{EXL}}$ was zero when the exception occurred. If the exception occurred in a branch delay slot or forbidden slot, the exception program counter (EPC) is set to restart execution at the branch.	R	Undefined

Table 5.17 Cause Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>TI</td><td>30</td><td>Timer Interrupt. This bit denotes whether a timer interrupt is pending (analogous to the *IP* bits for other interrupt types):
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No timer interrupt is pending</td></tr>
<tr><td>1</td><td>Timer interrupt is pending</td></tr>
</table>
The state of the *TI* bit is available on the external core interface as the *SI_TimerInt* signal</td><td>R</td><td>Undefined</td></tr>
<tr><td>CE</td><td>29:28</td><td>Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This field is loaded by hardware on every exception, but is **UNPREDICTABLE** for all exceptions except for Coprocessor Unusable.</td><td>R</td><td>Undefined</td></tr>
<tr><td>DC</td><td>27</td><td>Disable Count register. In some power-sensitive applications, the *Count* register is not used and is the source of meaningful power dissipation. This bit allows the *Count* register to be stopped in such situations.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Enable counting of *Count* register</td></tr>
<tr><td>1</td><td>Disable counting of *Count* register</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>PCI</td><td>26</td><td>Performance Counter Interrupt. This bit denotes whether a performance counter interrupt is pending (analogous to the IP bits for other interrupt types):
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No timer interrupt is pending</td></tr>
<tr><td>1</td><td>Timer interrupt is pending</td></tr>
</table>
The state of the *PCI* bit is available on the external M6200 interface as the *SI_PCInt* signal.</td><td>R</td><td>0</td></tr>
<tr><td>IC</td><td>25</td><td>Indicates if Interrupt Chaining occurred on the last IRET instruction.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupt Chaining did not happen on last IRET</td></tr>
<tr><td>1</td><td>Interrupt Chaining occurred during last IRET</td></tr>
</table></td><td>R</td><td>Undefined</td></tr>
</table>

Table 5.17 Cause Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>AP</td><td>24</td><td>Indicates whether an exception occurred during Interrupt Auto-Prologue.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Exception did not occur during Auto-Prologue operation.</td></tr><tr><td>1</td><td>Exception occurred during Auto-Prologue operation.</td></tr></table></td><td>R</td><td>Undefined</td></tr>
<tr><td>IV</td><td>23</td><td>Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Use the general exception vector (16#180)</td></tr><tr><td>1</td><td>Use the special interrupt vector (16#200)</td></tr></table>If the $Cause_{IV}$ is 1 and $Status_{BEV}$ is 0, the special interrupt vector represents the base of the vectored interrupt table.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>0</td><td>22</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
<tr><td>FDCI</td><td>21</td><td>Fast Debug Channel Interrupt. This bit denotes whether a FDC Interrupt is pending (analogous to the <i>IP</i> bits for other interrupt types):<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No Fast Debug Channel interrupt is pending</td></tr><tr><td>1</td><td>Fast Debug Channel interrupt is pending</td></tr></table></td><td>R</td><td>0</td></tr>
</table>

Table 5.17 Cause Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IP9:IP2</td><td>17:10</td><td>Indicates an interrupt is pending:
<table>
<tr><th>Bit</th><th>Name</th><th>Meaning</th></tr>
<tr><td>17</td><td>IP9</td><td>Hardware Interrupt 7</td></tr>
<tr><td>16</td><td>IP8</td><td>Hardware Interrupt 6</td></tr>
<tr><td>15</td><td>IP7</td><td>Hardware interrupt 5</td></tr>
<tr><td>14</td><td>IP6</td><td>Hardware interrupt 4</td></tr>
<tr><td>13</td><td>IP5</td><td>Hardware interrupt 3</td></tr>
<tr><td>12</td><td>IP4</td><td>Hardware interrupt 2</td></tr>
<tr><td>11</td><td>IP3</td><td>Hardware interrupt 1</td></tr>
<tr><td>10</td><td>IP2</td><td>Hardware interrupt 0</td></tr>
</table>
In implementations of Release 1 of the Architecture, timer and performance counter interrupts are combined in an implementation-dependent way with hardware interrupt 5.
In implementations in which EIC interrupt mode is not enabled ($Config3_{VEIC}$ = 0), timer and performance counter interrupts are combined in an implementation-dependent way with any hardware interrupt. If EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), these bits have a different meaning, and are interpreted as the *RIPL* field, described below.</td><td>R</td><td>Undefined for IP7:IP2
0 for IP9:IP8</td></tr>
<tr><td>RIPL</td><td>17:10</td><td>Requested Interrupt Priority Level.
In implementations in which EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), this field is the encoded (0..255) value of the requested interrupt. A value of zero indicates that no interrupt is requested.
If EIC interrupt mode is not enabled ($Config3_{VEIC}$ = 0), these bits have a different meaning and are interpreted as the *IP7..IP2* bits, described above.</td><td>R</td><td>Undefined for bits 15:10
0 for bits 17:16</td></tr>
<tr><td>IP1:IP0</td><td>9:8</td><td>Controls the request for software interrupts:
<table>
<tr><th>Bit</th><th>Name</th><th>Meaning</th></tr>
<tr><td>9</td><td>IP1</td><td>Request software interrupt 1</td></tr>
<tr><td>8</td><td>IP0</td><td>Request software interrupt 0</td></tr>
</table>
These bits are exported to an external interrupt controller for prioritization in EIC interrupt mode with other interrupt sources. The state of these bits is available on the external core interface as the *SI_SWInt[1:0]* bus.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>ExcCode</td><td>6:2</td><td>Exception code - see Table 5.18</td><td>R</td><td>Undefined</td></tr>
<tr><td>0</td><td>20:18, 7, 1:0</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
</table>

Table 5.18 Cause Register ExcCode Field

Exception Code Value		Mnemonic	Description
Decimal	Hexadecimal		
0	16#00	Int	Interrupt
1-3	16#01-16#03	-	Reserved
4	16#04	AdEL	Address error exception (load or instruction fetch)
5	16#05	AdES	Address error exception (store)
6	16#06	IBE	Bus error exception (instruction fetch)
7	16#07	DBE	Bus error exception (data reference: load or store)
8	16#08	Sys	Syscall exception
9	16#09	Bp	Breakpoint exception
10	16#0a	RI	Reserved instruction exception
11	16#0b	CpU	Coprocessor Unusable exception
12	16#0c	Ov	Arithmetic Overflow exception
13	16#0d	Tr	Trap exception
16	16#10	IS1	Implementation-Specific Exception 1 (COP2)
17	16#11	CEU	CorExtend Unusable
18	16#12	C2E	Coprocessor 2 exceptions
19-25	16#13-16#19	-	Reserved
26	16#1a	DSPDis	DSP Module State Disabled exception
27-28	16#1b-16#1c	-	Reserved
29	16#1d	MPU	MPU Exception
30	16#1e	SRAM Interface Parity or ECC error	SRAM Interface Parity or ECC error. In normal mode, an SRAM error exception has a dedicated vector and the *Cause* register is not updated. If an SRAM error occurs in Debug Mode, this code is written to the $Debug_{DExcCode}$ field to indicate that re-entry to Debug Mode was caused by an SRAM error.
31	16#1f	-	Reserved

5.2.15 View_RIPL Register (CP0 Register 13, Select 4)

This register gives read access to the *IP* or *RIPL* field that is also available in the *Cause* Register. The use of this register allows the Interrupt Pending or the Requested Priority Level to be read without extracting that bit field from the *Cause* Register.

Figure 5.15 View_RIPL Register Format

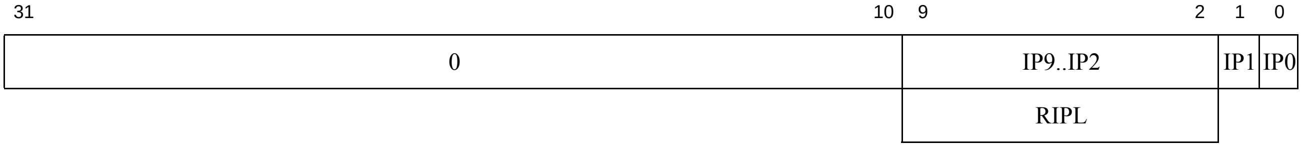


Table 5.19 View_RIPL Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:10	Must be written as zero; returns zero on read.	0	0
IP9:IP2	9:2	HW Interrupt Pending. If EIC interrupt mode is not enabled, indicates which HW interrupts are pending.	R	Undefined for IP7:IP2 0 for IP9:IP8
RIPL	9:2	Interrupt Priority Level. If EIC interrupt mode is enabled, this field indicates the Requested Priority Level of the pending interrupt.	R	Undefined
IP1:IP0	1:0	SW Interrupt Pending. If EIC interrupt mode is not enabled, controls which SW interrupts are pending.	R/W	Undefined

5.2.16 NestedExc (CP0 Register 13, Select 5)

The *Nested Exception* (*NestedExc*) register is an optional read-only register containing the values of $Status_{EXL}$ and $Status_{ERL}$ prior to acceptance of the current exception.

This register is part of the Nested Fault feature. The existence of the register can be determined by reading the $Config5_{NFExists}$ bit.

Figure 5.16 shows the format of the *NestedExc* register; Table 5.20 describes the *NestedExc* register fields.

Figure 5.16 NestedExc Register Format

31	3	2	1	0
0		ERL	EXL	0

Table 5.20 NestedExc Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:3	Reserved, read as 0.	R0	0
ERL	2	Value of $Status_{ERL}$ prior to acceptance of current exception. Updated by all exceptions that would set either $Status_{EXL}$ or $Status_{ERL}$. Not updated by Debug exceptions.	R	Undefined

Table 5.20 NestedExc Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
EXL	1	Value of *Status*$_{EXL}$ prior to acceptance of current exception. Updated by exceptions which would update EPC if *Status*$_{EXL}$ is not set (Interrupt, Address Error, all TLB exceptions, Bus Error, CopUnusable, Reserved Instruction, Overflow, Trap, Syscall, etc.). For these exception types, this register field is updated regardless of the value of *Status*$_{EXL}$. Not updated by exception types which update *ErrorEPC* (Reset, Soft Reset, NMI, SRAM Interface Parity or ECC error). Not updated by Debug exceptions.	R	Undefined
0	0	Reserved, read as 0.	R0	0

5.2.17 Exception Program Counter (CP0 Register 14, Select 0)

The *Exception Program Counter* (*EPC*) is a read/write register that contains the address at which processing resumes after an exception has been serviced. All bits of the *EPC* register are significant and must be writable.

For synchronous (precise) exceptions, the *EPC* contains 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, when the exception-causing instruction is in a branch delay slot or forbidden slot, and the *Branch Delay* bit in the *Cause* register is set.

On new exceptions, the processor does not write to the *EPC* register when the *EXL* bit in the *Status* register is set; however, the register can still be written via the MTC0 instruction.

A read of the *EPC* register (via MFC0) returns the following value in the destination GPR:

$\texttt{GPR[rt]} \leftarrow \texttt{ExceptionPC}_{31..1} \mid\mid \texttt{ISAMode}_0$

That is, the upper 31 bits of the exception PC are combined with the lower bit of the *ISAMode* field and written to the GPR.

Similarly, a write to the *EPC* register (via MTC0) takes the value from the GPR and distributes that value to the exception PC and the *ISAMode* field, as follows:

$\texttt{ExceptionPC} \leftarrow \texttt{GPR[rt]}_{31..1} \mid\mid 0$
$\texttt{ISAMode} \leftarrow \texttt{2\#0} \mid\mid \texttt{GPR[rt]}_0$

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the exception PC, and the lower bit of the exception PC is cleared. The upper bit of the *ISAMode* field is cleared, and the lower bit is loaded from the lower bit of the GPR.

Figure 5.17 EPC Register Format

31 … 0
EPC

Table 5.21 EPC Register Field Description

Fields				
Name	**Bit(s)**	**Description**	**Read/Write**	**Reset State**
EPC	31:0	Exception Program Counter.	R/W	Undefined

5.2.18 NestedEPC (CP0 Register 14, Select 2)

The *Nested Exception Program Counter* (*NestedEPC*) is an optional read/write register with the same behavior as the *EPC* register, except that:

- The *NestedEPC* register ignores the value of $Status_{EXL}$ and is therefore updated on the occurrence of any exception, including nested exceptions.
- The *NestedEPC* register is not used by the ERET/DERET/IRET instructions. To return to the address stored in *NestedEPC,* software must copy the value of the *NestedEPC* register to the *EPC* register.

This register is part of the Nested Fault feature. The existence of the register can be determined by reading the *Config5NFExists* bit.

Figure 5.16 shows the format of the *NestedEPC* register; Table 5.20 describes the *NestedEPC* register fields.

Figure 5.18 NestedEPC Register Format

31 … 0
NestedEPC

Table 5.22 NestedEPC Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
NestedEPC	31:0	Nested Exception Program Counter. Updated by exceptions which would update EPC if *$Status_{EXL}$* is not set (Interrupt, Address Error, all TLB exceptions, Bus Error, CopUnusable, Reserved Instruction, Overflow, Trap, Syscall, etc.). For these exception types, this register field is updated regardless of the value of *$Status_{EXL}$*. Not updated by exception types which update *ErrorEPC* (Reset, Soft Reset, NMI, and SRAM Interface Parity or ECC error). Not updated by Debug exceptions.	R/W	Undefined

5.2.19 Processor Identification (CP0 Register 15, Select 0)

The *Processor Identification* (*PRId*) register is a 32-bit, read-only register that contains information identifying the manufacturer, manufacturer options, processor identification, and revision level of the processor.

Figure 5.19 PRId Register Format

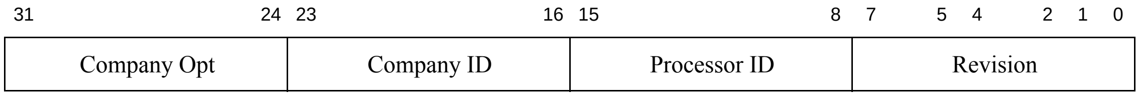


Table 5.23 PRId Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>Company Opt</td><td>31:24</td><td>Company Option. Whatever name is specified by the SoC builder who synthesizes the M6200 core; refer to your SoC manual. This field should be preset by the configuration GUI with a number between 0x00 and 0x7F; higher values (0x80-0xFF) are reserved by MIPS Technologies.</td><td>R</td><td>0</td></tr>
<tr><td>Company ID</td><td>23:16</td><td>Company Identifier. Identifies the company that designed or manufactured the processor. In the M6200 this field contains a value of 1 to indicate MIPS.
.</td><td>R</td><td>1</td></tr>
<tr><td>Processor ID</td><td>15:8</td><td>Processor Identifier. Identifies the type of processor. This field allows software to distinguish between the various types of MIPS processors.</td><td>R</td><td>0xaa</td></tr>
<tr><td>Revision</td><td>7:0</td><td>Processor Revision. Specifies the revision number of the processor. This field allows software to distinguish between different revisions of the same processor type. This field contains the following three subfields:
<table>
<tr><th>Bits</th><th>Name</th><th>Meaning</th><th>Read/ Write</th><th>Reset</th></tr>
<tr><td>7:5</td><td>Major Revision</td><td>This number is increased on major revisions of the processor core.</td><td>R</td><td>Preset</td></tr>
<tr><td>4:2</td><td>Minor Revision</td><td>This number is increased on each incremental revision of the processor and reset on each new major revision.</td><td>R</td><td>Preset</td></tr>
<tr><td>1:0</td><td>Patch Level</td><td>If a patch is made to modify an older revision of the processor, this field is incremented.</td><td>R</td><td>Preset</td></tr>
</table>
</td><td>R</td><td>0x20
(0b001_000_00)</td></tr>
</table>

5.2.20 EBase Register (CP0 Register 15, Select 1)

The *EBase* register is a read/write register containing the base address of the exception vectors used when $Status_{BEV}$ equals 0, and a read-only CPU number value that may be used by software to distinguish different processors in a multiprocessor system.

The *EBase* register provides the ability for software to identify the specific processor within a multiprocessor system, and allows the exception vectors for each processor to be different, especially in systems composed of heterogeneous processors. Bits 31:12 of the *EBase* register are concatenated with zeros to form the base of the exception vectors when $Status_{BEV}$ is 0. The exception vector base address comes from the fixed defaults (see Section 4.5 “Exception Vector Locations”) when $Status_{BEV}$ is 1, or for any Debug exception. The reset state of bits 31:12 of the *EBase* register initialize the exception base register to `16#8000.0000`, providing backward compatibility with Release 1 implementations.

Bits 31:30 of the *EBase* Register are fixed with the value `2#10` to force the exception base address to be in the kseg0 or kseg1 unmapped virtual address segments.

If the value of the exception base register is to be changed, this must be done with $Status_{BEV}$ equal 1. The operation of the processor is **UNDEFINED** if the Exception Base field is written with a different value when $Status_{BEV}$ is 0.

Combining bits 31:12 with the *Exception Base* field allows the base address of the exception vectors to be placed at any 4KByte page boundary. If vectored interrupts are used, a vector offset greater than 4KBytes can be generated. In this case, bit 12 of the *Exception Base* field must be zero. The operation of the processor is **UNDEFINED** if software writes bit 12 of the *Exception Base* field with a 1 and enables the use of a vectored interrupt whose offset is greater than 4KBytes from the exception base address.

Figure 5.20 shows the format of the *EBase* Register; Table 5.24 describes the *EBase* register fields.

Figure 5.20 EBase Register Format

31	30	29 12	11 10	9 0
1	0	Exception Base	0 0	CPUNum

Table 5.24 EBase Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
1	31	This bit is ignored on writes and returns one on reads.	R	1
0	30	This bit is ignored on writes and returns zero on reads.	R	0
Exception Base	29:12	In conjunction with bits 31:30, this field specifies the base address of the exception vectors when $Status_{BEV}$ is zero.	R/W	0
0	11:10	Must be written as zero; returns zero on reads.	0	0
CPUNum	9:0	This field specifies the number of the CPU in a multiprocessor system and can be used by software to distinguish a particular processor from the others. The value in this field is set by the *SI_CPUNum[9:0]* static input pins to the core. In a single processor system, this value should be set to zero.	R	Externally Set

5.2.21 CDMMBase Register (CP0 Register 15, Select 2)

The 36-bit physical base address for the Common Device Memory Map facility is defined by this register. This register only exists if $Config3_{CDMM}$ is set to one.

Figure 5.21 shows the format of the *CDMMBase* register, and Table 5.25 describes the register fields.

Figure 5.21 CDMMBase Register Format

31 11	10	9	8 0
CDMM_UPPER_ADDR	EN	CI	CDMMSize

Table 5.25 CDMMBase Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>CDMM_UPPER_ADDR</td><td>31:11</td><td>Bits 35:15 of the base physical address of the memory mapped registers.
The number of implemented physical address bits is implementation-specific. For the unimplemented address bits, writes are ignored and reads return zero.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>EN</td><td>10</td><td>Enables the CDMM region.
If this bit is cleared, memory requests to this address region access regular system memory. If this bit is set, memory requests to this region access the CDMM logic<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>CDMM Region is disabled.</td></tr><tr><td>1</td><td>CDMM Region is enabled.</td></tr></table></td><td>R/W</td><td>0</td></tr>
<tr><td>CI</td><td>9</td><td>If set to 1, this indicates that the first 64-byte Device Register Block of the CDMM is reserved for additional registers that manage CDMM region behavior and are not IO device registers.</td><td>R</td><td>0</td></tr>
<tr><td>CDMMSize</td><td>8:0</td><td>This field represents the number of 64-byte Device Register Blocks instantiated in the core.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>1 DRB</td></tr><tr><td>1</td><td>2 DRBs</td></tr><tr><td>2</td><td>3 DRBs</td></tr><tr><td>...</td><td>...</td></tr><tr><td>511</td><td>512 DRBs</td></tr></table></td><td>R</td><td>Preset</td></tr>
</table>

5.2.22 Config Register (CP0 Register 16, Select 0)

The *Config* register specifies various configuration and capabilities information. Most of the fields in the *Config* register are initialized by hardware during the Reset exception process, or are constant.Figure 5.22 shows the format of the Config Register format, and Table 5.26 describes the register fields.

Figure 5.22 Config Register Format

31	30 28	27 25	24 23	22	21	20	19 16	15	14 13	12 10	9 7	6 4 3	2 0
M	K23	KU	0	UDI	0	MDU	0	BE	AT	AR	MT	0	K0

Table 5.26 Config Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>M</td><td>31</td><td>This bit is hardwired to ‘1’ to indicate the presence of the <i>Config1</i> register.</td><td>R</td><td>1</td></tr>
<tr><td>K23</td><td>30:28</td><td>This field controls the cacheability of the kseg2 and kseg3 address segments in FM implementations.
Refer to Table 5.27 for the field encoding.</td><td>FM: R/W</td><td>FM: 010</td></tr>
<tr><td>KU</td><td>27:25</td><td>This field controls the cacheability of the kuseg and useg address segments in FM implementations.
Refer to Table 5.27 for the field encoding.</td><td>FM: R/W</td><td>FM: 010</td></tr>
<tr><td>0</td><td>24:23</td><td>Must be written as 0. Returns zero on reads.</td><td>0</td><td>0</td></tr>
<tr><td>UDI</td><td>22</td><td>This bit indicates if CorExtend User Defined Instructions have been implemented.
This bit is hardwired to 0 to indicate that user-defined instructions are not implemented.</td><td>R</td><td>0</td></tr>
<tr><td>0</td><td>21</td><td>Must be written as 0. Returns zero on reads.</td><td>0</td><td>0</td></tr>
<tr><td>MDU</td><td>20</td><td>This bit indicates the type of Multiply/Divide Unit present. This bit is preset to 0 to indicate high-performance MDU.</td><td>R</td><td>0</td></tr>
<tr><td>0</td><td>19:16</td><td>Must be written as 0. Returns zero on reads.</td><td>0</td><td>0</td></tr>
<tr><td>BE</td><td>15</td><td>Indicates the endian mode in which the processor is running. Set via <i>SI_Endian</i> input pin.
0: Little endian
1: Big endian</td><td>R</td><td>0</td></tr>
<tr><td>AT</td><td>14:13</td><td>Architecture type implemented by the processor. This field is always 00 to indicate the MIPS32 architecture.</td><td>R</td><td>00</td></tr>
<tr><td>AR</td><td>12:10</td><td>Architecture revision level. This field is always 010 to indicate MIPS32 Release 6.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Release 1</td></tr><tr><td>1</td><td>Release 2</td></tr><tr><td>2</td><td>Release 6</td></tr><tr><td>3-7</td><td>Reserved</td></tr></table></td><td>R</td><td>010</td></tr>
<tr><td>MT</td><td>9:7</td><td>MMU Type:
3: Fixed Mapping
0-2, 4-7: Reserved</td><td>R</td><td>3</td></tr>
<tr><td>0</td><td>6:4</td><td>Must be written as zeros; returns zeros on reads.</td><td>0</td><td>0</td></tr>
<tr><td>K0</td><td>2:0</td><td>Kseg0 coherency algorithm. Refer to Table 5.27 for the field encoding.</td><td>R/W</td><td>010</td></tr>
</table>

Table 5.27 Cache Coherency Attributes

C(2:0) Value	Cache Coherency Attribute
2	Uncached.
3	Cached Core does not support caches, but passes this attribute to the system for use with any external caching mechanisms. In addition, when fetching from cacheable space, the core will attempt to improve performance by anticipating where the next instruction is located and pre-fetch/speculatively-fetch the next instruction. In contrast, fetching from uncacheable space will cause the core to stall until the current outstanding instruction is completed and graduated from the pipeline. Similarly, when reading or writing to cacheable space, the core will attempt to optimize the performance by using its internal buffers. This may cause data/address-independent cacheable accesses to appear out-of-order. Performing the SYNC instruction or accessing uncacheable space will prevent this behavior but potentially reduce performance.
All other values	Cause the CCA value to be set to 3.

5.2.23 Config1 Register (CP0 Register 16, Select 1)

The *Config1* register is an adjunct to the *Config* register and encodes additional information about capabilities present on the core. All fields in the *Config1* register are read-only.

Figure 5.23 Config1 Register Format — Select 1

31	30 25	24 22	21 19	18 16	15 13	12 10	9 7	6	5	4	3	2	1	0
M	MMU Size	IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP

Table 5.28 Config1 Register Field Descriptions — Select 1

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
M	31	This bit is hardwired to ‘1’ to indicate the presence of the *Config2* register.	R	1
MMU Size	30:25	This field contains the number of entries in the TLB minus one.	R	0
IS	24:22	This field contains the number of instruction cache sets per way. Because the M6200 core does not include caches, this field is always read as 0.	R	0
IL	21:19	This field contains the instruction cache line size. Because the M6200 core does not include caches, this field is always read as 0.	R	0
IA	18:16	This field contains the level of instruction cache associativity. Because the M6200 core does not include caches, this field is always read as 0.	R	0
DS	15:13	This field contains the number of data cache sets per way. Because the M6200 core does not include caches, this field is always read as 0.	R	0
DL	12:10	This field contains the data cache line size. Because the M6200 core does not include caches, this field is always read as 0.	R	0
DA	9:7	This field contains the type of set associativity for the data cache. Because the M6200 core does not include caches, this field is always read as 0.	R	0
C2	6	Coprocessor 2 present. 0: No coprocessor is attached to the COP2 interface 1: A coprocessor is attached to the COP2 interface If the Cop2 interface logic is not implemented, this bit will read 0.	R	Preset
MD	5	MDMX implemented. This bit always reads as 0 because MDMX is not supported.	R	0
PC	4	Performance Counter registers implemented. 0: No Performance Counter registers are implemented 1: Performance Counter registers are implemented	R	Preset
WR	3	Watch Registers implemented. This bit is always read as 0, because the M6200 core does not contain Watch registers.	R	0
CA	2	Code compression (MIPS16e) implemented. 0: MIPS16e is not implemented 1: MIPS16e is implemented Because the M6200 does not support the MIPS16e ASE, this bit is always read as 0.	R	0

Table 5.28 Config1 Register Field Descriptions — Select 1 (Continued)

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
EP	1	Debug interface present: This bit is always set to indicate that the core implements the Debug interface.	R	1
FP	0	FPU implemented. 0: No FPU 1: FPU is implemented This bit always reads as 0 because an FPU is not supported.	R	0

5.2.24 Config2 Register (CP0 Register 16, Select 2)

The *Config2* register is an adjunct to the *Config* register and is reserved to encode additional capabilities information. *Config2* is allocated for showing the configuration of level 2/3 caches. These fields are reset to 0 because L2/L3 caches are not supported by the M6200 core. All fields in the *Config2* register are read-only.

Figure 5.24 Config2 Register Format — Select 2

31	30 0
M	0

Table 5.29 Config2 Register Field Descriptions — Select 1

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
M	31	This bit is hardwired to ‘1’ to indicate the presence of the *Config3* register.	R	1
0	30:0	These bits are reserved.	R	0

5.2.25 Config3 Register (CP0 Register 16, Select 3)

The *Config3* register encodes additional capabilities. All fields in the *Config3* register are read-only.

Figure 5.25 shows the format of the *Config3* register; Table 5.30 describes the *Config3* register fields.

Figure 5.25 Config3 Register Format

31	30 28	27	26	25 23	22 21	20 18	17	16	15 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	BP	BI	0	IPLW	MMAR	MCU	ISA On Exc	ISA	ULRI	RXI	DSP2P	DSPP	0	ITL	LPA	VEIC	VInt	SP	CDMM	0	SM	TL

Table 5.30 Config3 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>M</td><td>31</td><td>This bit is reserved to indicate that a <i>Config4</i> register is present.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td><i>Config4</i> is not implemented.</td></tr>
<tr><td>1</td><td><i>Config4</i> is implemented.</td></tr>
</table></td><td>R</td><td>1</td></tr>
<tr><td>0</td><td>30:28,
25:23,9,2</td><td>Must be written as zeros; returns zeros on read.</td><td>0</td><td>0</td></tr>
<tr><td>BP</td><td>27</td><td><i>BadInstrP</i> register implemented. This bit indicates whether the faulting prior branch instruction word register is present.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td><i>BadInstrP</i> is not implemented.</td></tr>
<tr><td>1</td><td><i>BadInstrP</i> is implemented.</td></tr>
</table></td><td>R</td><td>1</td></tr>
<tr><td>BI</td><td>26</td><td><i>BadInstr</i> register implemented. This bit indicates whether the faulting instruction word register is present.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td><i>BadInstr</i> is not implemented.</td></tr>
<tr><td>1</td><td><i>BadInstr</i> is implemented.</td></tr>
</table></td><td>R</td><td>1</td></tr>
<tr><td>IPLW</td><td>22:21</td><td>Width of the $Status_{IPL}$ and $Cause_{RIPL}$ fields:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td><i>IPL</i> and <i>RIPL</i> fields are 6-bits in width.</td></tr>
<tr><td>1</td><td><i>IPL</i> and <i>RIPL</i> fields are 8-bits in width.</td></tr>
<tr><td>Others</td><td>Reserved.</td></tr>
</table>
If the <i>IPL</i> field is 8-bits in width, bits 18 and 16 of <i>Status</i> are used as the most significant bit and second most significant bit, respectively, of that field.

If the <i>RIPL</i> field is 8-bits in width, bits 17 and 16 of <i>Cause</i> are used as the most significant bit and second most significant bit, respectively, of that field.</td><td>R</td><td>1</td></tr>
</table>

Table 5.30 Config3 Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>MMAR</td><td>20:18</td><td>microMIPS Architecture revision level. This field is always 010 to indicate microMIPS32 Release 6.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Release 1</td></tr>
<tr><td>1</td><td>Release 2</td></tr>
<tr><td>2</td><td>Release 6</td></tr>
<tr><td>3-7</td><td>Reserved</td></tr>
</table></td><td>R</td><td>010</td></tr>
<tr><td>MCU</td><td>17</td><td>MIPS MCU ASE Implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MCU ASE is not implemented.</td></tr>
<tr><td>1</td><td>MCU ASE is implemented.</td></tr>
</table></td><td>R</td><td>1</td></tr>
<tr><td>ISAOnExc</td><td>16</td><td>Reflects the Instruction Set Architecture used when vectoring to an exception. Affects exceptions whose vectors are offsets from <i>EBase</i>.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS32 ISA is used on entrance to an exception vector.</td></tr>
<tr><td>1</td><td>microMIPS is used on entrance to an exception vector.</td></tr>
</table>
Note that the M6200 supports booting only in MIPS32 mode.</td><td>R</td><td>0</td></tr>
<tr><td>ISA</td><td>15:14</td><td>Indicates Instruction Set Availability.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Reserved</td></tr>
<tr><td>1</td><td>Reserved</td></tr>
<tr><td>2</td><td>Both MIPS32 and microMIPS are implemented. MIPS32 ISA used when coming out of reset.</td></tr>
<tr><td>3</td><td>Both MIPS32 and microMIPS are implemented. microMIPS is used when coming out of reset.</td></tr>
</table></td><td>R</td><td>2</td></tr>
</table>

Table 5.30 Config3 Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>ULRI</td><td>13</td><td><i>UserLocal</i> register implemented. This bit indicates whether the <i>UserLocal</i> coprocessor 0 register is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td><i>UserLocal</i> register is not implemented</td></tr>
<tr><td>1</td><td><i>UserLocal</i> register is implemented</td></tr>
</table></td><td>R</td><td>1</td></tr>
<tr><td>RXI</td><td>12</td><td>Indicates whether the <i>RIE</i> and <i>XIE</i> bits exist within the <i>PageGrain</i> register..
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>The <i>RIE</i> and <i>XIE</i> bits are not implemented within the <i>PageGrain</i> register.</td></tr>
<tr><td>1</td><td>The <i>RIE</i> and <i>XIE</i> bits are implemented within the <i>PageGrain</i> register</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>DSP2P</td><td>11</td><td>Reads 1 to indicate that Revision 2 or higher of the MIPS DSP Module is implemented</td><td>R</td><td>Preset</td></tr>
<tr><td>DSPP</td><td>10</td><td>Reads 1 to indicate that the MIPS DSP Module extension is implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td>ITL</td><td>8</td><td>Indicates that iFlowtrace hardware is present.</td><td>R</td><td>Preset</td></tr>
<tr><td>LPA</td><td>7</td><td>Denotes the presence of support for large physical addresses on MIPS64 processors. Not used by MIPS32 processors and returns zero on read.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Large physical address support is not implemented</td></tr>
<tr><td>1</td><td>Large physical address support is implemented</td></tr>
</table>
For implementations of Release 1 of the Architecture, this bit returns zero on read.</td><td>R</td><td>Preset</td></tr>
<tr><td>VEIC</td><td>6</td><td>Indicates support for an external interrupt controller.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Support for EIC interrupt mode is not implemented</td></tr>
<tr><td>1</td><td>Support for EIC interrupt mode is implemented</td></tr>
</table>
The value of this bit is set by the static input, <i>SI_EICPresent</i>. This allows external logic to communicate whether an external interrupt controller is attached to the processor or not.</td><td>R</td><td>Externally Set</td></tr>
</table>

Table 5.30 Config3 Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>VInt</td><td>5</td><td>Indicates implementation of Vectored interrupts.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Vector interrupts are not implemented</td></tr><tr><td>1</td><td>Vectored interrupts are implemented</td></tr></table>On the M6200 core, this bit is always a 1, because vectored interrupts are implemented.</td><td>R</td><td>1</td></tr>
<tr><td>SP</td><td>4</td><td>When set, indicates that Small (1KByte) page support is implemented. This bit is hardwired to ‘0’ to indicate that small page size is not supported.</td><td>R</td><td>0</td></tr>
<tr><td>CDMM</td><td>3</td><td>Common Device Memory Map implemented. This bit indicates whether the CDMM is implemented.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>CDMM is not implemented</td></tr><tr><td>1</td><td>CDMM is implemented</td></tr></table></td><td>R</td><td>Preset</td></tr>
<tr><td>SM</td><td>1</td><td>SmartMIPS™ ASE implemented. This bit indicates whether the SmartMIPS ASE is implemented. Because SmartMIPS is not present on the M6200 core, this bit will always be 0.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>SmartMIPS ASE is not implemented</td></tr><tr><td>1</td><td>SmartMIPS ASE is implemented</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>TL</td><td>0</td><td>Trace Logic implemented. This bit indicates whether PC or data trace is implemented.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Trace logic is not implemented</td></tr><tr><td>1</td><td>Trace logic is implemented</td></tr></table></td><td>R</td><td>0</td></tr>
</table>

5.2.26 Config4 Register (CP0 Register 16, Select 4)

The *Config4* register encodes additional capabilities. This register is required if any optional feature described by this register is implemented and is otherwise optional.

Figure 5.26 shows the format of the *Config4* register; Table 5.31 describes the *Config4* register fields.

Figure 5.26 Config4 Register Format

31	30 29 24	23 16	15 0
M	0	KScr Exist	0

Table 5.31 Config4 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
M	31	This bit is reserved to indicate that a *Config5* register is present.	R	1
0	30:24	Reserved	R	0
KScrExist	23:16	Indicates how many scratch registers are available to kernel-mode software in CP0 Register 31. Each bit represents a select for Coproecessor0 Register 31. Bit 16 represents Select 0, Bit 23 represents Select 7. If the bit is set, the associated scratch register is implemented and available for kernel-mode software. Bits 2-7 are always 1 because KScratch1-6 must be implemented. Scratch registers meant for other purposes are not represented in this field. For example, if the Debug interface is implemented, Bit 16 is preset to zero even though DESAVE register is implemented at Select 0. Select 1 is reserved for future debug purposes and should not be used as a kernel scratch register, so bit 17 is preset to zero.	R	8'b1111_1100
0	15:0	Must be written as zeros; returns zeros on read.	0	0

5.2.27 Config5 Register (CP0 Register 16, Select 5)

The *Config5* register encodes additional capabilities. This register is required if any optional feature described by this register is implemented and is otherwise optional.

Figure 5.27 shows the format of the *Config5* register; Table 5.32 describes the *Config5* register fields.

Figure 5.27 Config5 Register Format

31	30	29 28 14	13	12	11	10 9 7	6	5	4	3 2	1	0
M	0	0	XNP	0	DEC	0	SBRI	MVH	LLB	0	0	NF

Table 5.32 Config5 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
M	31	This bit is reserved and must always read as a 0.	R	0
0	30	Must be written as zeros; returns zeros on read.	0	0

Table 5.32 Config5 Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>0</td><td>29:14</td><td>Must be written as zeros; returns zeros on read.</td><td>0</td><td>0</td></tr>
<tr><td>XNP</td><td>13</td><td>Extended LL/SC family of instructions Not Present.
The LLX/SCX family of instructions is required for Release 6 Double-Width atomic support. This support is provided by extending the capability of legacy LL/SC instructions.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>LLX/SCX instruction family supported</td></tr>
<tr><td>1</td><td>LLX/SCX instruction family not supported</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>0</td><td>12</td><td>Must be written as zeros; returns zeros on read.</td><td>R0</td><td>Preset</td></tr>
<tr><td>DEC</td><td>11</td><td>Dual Endian Capability. Determines endian capability of processor.
If both modes are supported, then the processor will initially boot in little-endian mode always. Thereafter, software can force a change in endian mode by setting a bit in a memory-mapped external register. The endian mode change will only take effect on subsequent reset. For current endian state, software should read <i>ConfigBE</i>.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Only Little-Endian mode supported. Any implementation must support Little-endian mode.</td></tr>
<tr><td>1</td><td>Both Little and Big-Endian modes supported.</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>0</td><td>10:7</td><td>Must be written as zeros; returns zeros on read.</td><td>0</td><td>0</td></tr>
<tr><td>SBRI</td><td>6</td><td>SDBBP instruction Reserved Instruction control.
The purpose of this field is to restrict availability of SDBBP to kernel mode operation. It prevents user (and supervisor) code from entering Debug mode using SDBBP.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>SDBBP instruction executes as defined prior to Release 6</td></tr>
<tr><td>1</td><td>SDBBP instruction can only be executed in kernel mode. User (or supervisor, if supported) execution of SDBBP will cause a Reserved Instruction exception.</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
</table>

Table 5.32 Config5 Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>MVH</td><td>5</td><td>Move To/From High COP0 (MTHC0/MFHC0) instructions are implemented.
Currently these instructions are only required for Extended Physical Addressing (XPA).
.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MTHC0 and MFHC0 are not supported. COP0 extensions do not exist.</td></tr>
<tr><td>1</td><td>MTHC0 and MFHC0 are supported. Extensions to 32-bit COP0 registers exist.</td></tr>
</table></td><td>R</td><td>1</td></tr>
<tr><td>LLB</td><td>4</td><td>Load-Linked Bit (LLB) is present in CP0 <i>LLAddr</i>.</td><td>R</td><td>1</td></tr>
<tr><td>0</td><td>3:1</td><td>Must be written as zeros; returns zeros on read.</td><td>0</td><td>0</td></tr>
<tr><td>NFExists</td><td>0</td><td>Indicates that the Nested Fault feature is present.
The Nested Fault feature allows recognition of faulting behavior within an exception handler.</td><td>R</td><td>1</td></tr>
</table>

5.2.28 Config7 Register (CP0 Register 16, Select 7)

The *Config7* register contains implementation specific configuration information. A number of these bits are writable to disable certain performance enhancing features within the M6200 core.

Figure 5.28 Config7 Register Format

31 19	18	17 9	8	7 0
0	HCI	0	ES	

Table 5.33 Config7 Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read / Write**	**Reset State**
0	31:19,17:0	These bits are unused and should be written as 0.	R	0
HCI	18	Hardware Cache Initialization: Indicates that a cache does not require initialization by software. It is the integrator's responsibility to include a hardware cache initialization module if this bit is configured to 1.	R	0
ES	8	Externalize Sync: When asserted, SYNC instructions are externalized and made visible to the bus. For designs that have slave controllers unable to handle SYNC behaviors, set this bit to 0.	R/W	0

5.2.29 Load Linked Address (CP0 Register 17, Select 0)

The *LLAddr* register contains the physical address read by the most recent Load Linked (LL) instruction.

Figure 5.29 LLAddr Register Format

31 1	0
PAddr	LLB

Table 5.34 LLAddr Register Field Descriptions

Fields				
Name	**Bit(s)**	**Description**	**Read/Write**	**Reset State**
PAddr[31:1]	31:1	Physical Address. This field encodes the physical address read by the most recent Load Linked instruction. *LLAddr[1]* is aligned to *PA[5*], which implies that *PAddr* is always 32-byte aligned	R	Undefined
LLB	0	LL Bit. *LLB* is set when the LL instruction is executed. The SC instructions and other hardware events may clear *LLB*. This field allows the LL Bit to be accessible by software. *LLB* can be cleared by software but cannot be set.	R/W	0

5.2.30 Debug Register (CP0 Register 23, Select 0)

The *Debug* register is used to control the debug exception and provide information about the cause of the debug exception and also when re-entering at the debug exception vector due to a normal exception in debug mode. The read-only information bits are updated every time the debug exception is taken, or when a normal exception is taken when already in debug mode.

Only the *DM* bit and the *Dbgver* field are valid when read from non-debug mode; the values of all other bits and fields are UNPREDICTABLE. Operation of the processor is UNDEFINED if the *Debug* register is written in non-debug mode.

Some of the bits and fields are only updated on debug exceptions and/or exceptions in debug mode, as shown below:

- *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB*, *DINT*, *DIBImpr*, *DDBLImpr*, *DDBSImpr* are updated on both debug exceptions and on exceptions in debug modes.
- *DExcCode* is updated on exceptions in debug mode, and is undefined after a debug exception.
- *Halt* and *Doze* are updated on a debug exception, and are undefined after an exception in debug mode.
- *DBD* is updated on both debug and on exceptions in debug modes.

All bits and fields are undefined when read from normal mode, except those explicitly described to be defined, e.g., *Dbgver* and *DM*.

Figure 5.30 Debug Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19
DBD	DM	NoDCR	LSNM	Doze	Halt	CountDM	IBusEP	MCheckP	CacheEP	DBusEP	IEXI	DDB-SImpr

18	17 15	14 10	9	8	7	6	5	4	3	2	1	0
DDB-LImpr	DbgVer	DExcCode	NoSSt	SSt	R	DIBImpr	DINT	DIB	DDBS	DDBL	DBp	DSS

Table 5.35 Debug Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>DBD</td><td>31</td><td>Indicates whether the last debug exception or exception in debug mode occurred in a branch delay slot or forbidden slot:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Not in delay slot</td></tr>
<tr><td>1</td><td>In delay slot</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>DM</td><td>30</td><td>Indicates that the processor is operating in debug mode:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Processor is operating in non-debug mode</td></tr>
<tr><td>1</td><td>Processor is operating in debug mode</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>NoDCR</td><td>29</td><td>Indicates whether the dseg memory segment is present and the Debug Control Register is accessible:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>dseg is present</td></tr>
<tr><td>1</td><td>No dseg present</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>LSNM</td><td>28</td><td>Controls access of load/store between dseg and main memory:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Load/stores in dseg address range goes to dseg</td></tr>
<tr><td>1</td><td>Load/stores in dseg address range goes to main memory</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
</table>

Table 5.35 Debug Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>Doze</td><td>27</td><td>Indicates that the processor was in any kind of low power mode when a debug exception occurred:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Processor not in low-power mode when debug exception occurred</td></tr><tr><td>1</td><td>Processor in low-power mode when debug exception occurred</td></tr></table></td><td>R/W</td><td>1</td></tr>
<tr><td>Halt</td><td>26</td><td>Indicates that the internal system bus clock was stopped when the debug exception occurred:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Internal system bus clock stopped</td></tr><tr><td>1</td><td>Internal system bus clock running</td></tr></table></td><td>R/W1</td><td>0</td></tr>
<tr><td>CountDM</td><td>25</td><td>Indicates the Count register behavior in debug mode:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Count register stopped in debug mode</td></tr><tr><td>1</td><td>Count register is running in debug mode</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>IBusEP</td><td>24</td><td>Instruction-fetch Bus Error Exception Pending. Set when an instruction-fetch bus error event occurs, or if a 1 is written to the bit by software. Cleared when a Bus Error exception on an instruction fetch is taken by the processor, and by reset. If <i>IBusEP</i> is set when <i>IEXI</i> is cleared, a Bus Error exception on an instruction fetch is taken by the processor, and <i>IBusEP</i> is cleared.</td><td>R/W1</td><td>0</td></tr>
<tr><td>MCheckP</td><td>23</td><td>Indicates that an imprecise Machine Check exception is pending. Machine Check exceptions are not implemented in the M6200 processor, so this bit will always read as 0.</td><td>R</td><td>0</td></tr>
<tr><td>CacheEP</td><td>22</td><td>Indicates that an imprecise SRAM Interface Parity or ECC error is pending.</td><td>R/W1</td><td>0</td></tr>
<tr><td>DBusEP</td><td>21</td><td>Data access Bus Error exception Pending. Covers imprecise bus errors on data access, similar to the behavior of <i>IBusEP</i> for imprecise bus errors on an instruction fetch.</td><td>R/W1</td><td>0</td></tr>
</table>

Table 5.35 Debug Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>IEXI</td><td>20</td><td>Imprecise Error eXception Inhibit controls exceptions taken due to imprecise error indications. Set when the processor takes a debug exception or exception in debug mode. Cleared by execution of the DERET instruction; otherwise modifiable by debug mode software. When <i>IEXI</i> is set, the imprecise error exception from a bus error on an instruction fetch or data access, SRAM Interface Parity or ECC error, or machine check is inhibited and deferred until the bit is cleared.</td><td>R/W</td><td>0</td></tr>
<tr><td>DDBSImpr</td><td>19</td><td>Indicates that an imprecise Debug Data Break Store exception was taken. Imprecise data breaks only occur on complex breakpoints.</td><td>R</td><td>Undefined</td></tr>
<tr><td>DDBLImpr</td><td>18</td><td>Indicates that an imprecise Debug Data Break Load exception was taken. Imprecise data breaks only occur on complex breakpoints.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>Dbgver</td><td>17:15</td><td>Debug interface version.</td><td>R</td><td>0</td></tr>
<tr><td>DExcCode</td><td>14:10</td><td>Indicates the cause of the latest exception in debug mode. The field is encoded as the <i>ExcCode</i> field in the <i>Cause</i> register for those normal exceptions that may occur in debug mode.
Value is undefined after a debug exception.</td><td>R/W</td><td>0</td></tr>
<tr><td>NoSST</td><td>9</td><td>Indicates whether the single-step feature controllable by the <i>SSt</i> bit is available in this implementation:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Single-step feature available</td></tr>
<tr><td>1</td><td>No single-step feature available</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>SSt</td><td>8</td><td>Controls if debug single step exception is enabled:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No debug single-step exception enabled</td></tr>
<tr><td>1</td><td>Debug single step exception enabled</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>R</td><td>7</td><td>Reserved. Must be written as zeros; returns zeros on reads.</td><td>R</td><td>0</td></tr>
<tr><td>DIBImpr</td><td>6</td><td>Indicates that an Imprecise debug instruction break exception occurred (due to a complex breakpoint). Cleared on exception in debug mode.</td><td>R</td><td>Undefined</td></tr>
<tr><td>DINT</td><td>5</td><td>Indicates that a debug interrupt exception occurred. Cleared on exception in debug mode.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No debug interrupt exception</td></tr>
<tr><td>1</td><td>Debug interrupt exception</td></tr>
</table></td><td>R</td><td>Undefined</td></tr>
</table>

Table 5.35 Debug Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>DIB</td><td>4</td><td>Indicates that a debug instruction break exception occurred. Cleared on exception in debug mode.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No debug instruction exception</td></tr><tr><td>1</td><td>Debug instruction exception</td></tr></table></td><td>R</td><td>Undefined</td></tr>
<tr><td>DDBS</td><td>3</td><td>Indicates that a debug data break exception occurred on a store. Cleared on exception in debug mode.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No debug data exception on a store</td></tr><tr><td>1</td><td>Debug instruction exception on a store</td></tr></table></td><td>R</td><td>Undefined</td></tr>
<tr><td>DDBL</td><td>2</td><td>Indicates that a debug data break exception occurred on a load. Cleared on exception in debug mode.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No debug data exception on a load</td></tr><tr><td>1</td><td>Debug instruction exception on a load</td></tr></table></td><td>R</td><td>Undefined</td></tr>
<tr><td>DBp</td><td>1</td><td>Indicates that a debug software breakpoint exception occurred. Cleared on exception in debug mode.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No debug software breakpoint exception</td></tr><tr><td>1</td><td>Debug software breakpoint exception</td></tr></table></td><td>R</td><td>Undefined</td></tr>
<tr><td>DSS</td><td>0</td><td>Indicates that a debug single-step exception occurred. Cleared on exception in debug mode.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No debug single-step exception</td></tr><tr><td>1</td><td>Debug single-step exception</td></tr></table></td><td>R/W</td><td>0</td></tr>
</table>

5.2.31 User Trace Data1 Register (CP0 Register 23, Select 3)/User Trace Data2 Register (CP0 Register 24, Select 3)

A software write to any bits in the *UserTraceData1* or *UserTraceData2* registers will trigger a trace record to be written indicating a type 1 or type 2 user format respectively. The trace output data is **UNPREDICTABLE** if these registers are written in consecutive cycles.

This register is only implemented if the MIPS iFlowtrace capability is present.

Figure 5.31 User Trace Data1/User Trace Data2 Register Format

31 0
Data

Table 5.36 UserTraceData1/UserTraceData2 Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read / Write**	**Reset State**
Data	31:0	Software readable/writable data. When written, this triggers a user format trace record out of the iFlowtrace interface that transmits the Data field to trace memory.	R/W	0

5.2.32 Debug2 Register (CP0 Register 23, Select 6)

This register holds additional information about Complex Breakpoint exceptions.

This register is only implemented if complex hardware breakpoints are present.

Figure 5.32 Debug2 Register Format

31 4	3	2	1	0
0	Prm	DQ	Tup	PaCo

Table 5.37 Debug2 Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read/Write**	**Reset State**
0	31:4	Reserved	0	0
Prm	3	Primed. Indicates whether a complex breakpoint with an active priming condition was seen on the last debug exception.	R	Undefined
DQ	2	Data Qualified. Indicates whether a complex breakpoint with an active data qualifier was seen on the last debug exception.	R	Undefined
Tup	1	Tuple. Indicates whether a tuple breakpoint was seen on the last debug exception.	R	Undefined
PaCo	0	Pass Counter. Indicates whether a complex breakpoint with an active pass counter was seen on the last debug exception	R/W	Undefined

5.2.33 Debug Exception Program Counter Register (CP0 Register 24, Select 0)

The Debug Exception Program Counter (*DEPC*) register is a read/write register that contains the address at which processing resumes after a debug exception or debug mode exception has been serviced.

For synchronous (precise) debug and debug mode exceptions, the *DEPC* contains either:

- The virtual address of the instruction that was the direct cause of the debug exception, or
- The virtual address of the immediately preceding branch or jump instruction, when the debug exception causing instruction is in a branch delay slot or forbidden slot, and the *Debug Branch Delay* (*DBD*) bit in the *Debug* register is set.

For asynchronous debug exceptions (debug interrupt, complex break), the *DEPC* contains the virtual address of the instruction where execution should resume after the debug handler code is executed.

A read of the *DEPC* register (via MFC0) returns the following value in the destination GPR:

```
GPR[rt] $\leftarrow$ DebugExceptionPC$_{31..1}$ || ISAMode$_0$
```

That is, the upper 31 bits of the debug exception PC are combined with the lower bit of the *ISAMode* field and written to the GPR.

Similarly, a write to the *DEPC* register (via MTC0) takes the value from the GPR and distributes that value to the debug exception PC and the *ISAMode* field, as follows

```
DebugExceptionPC $\leftarrow$ GPR[rt]$_{31..1}$ || 0
ISAMode $\leftarrow$ 2#0 || GPR[rt]$_0$
```

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the debug exception PC, and the lower bit of the debug exception PC is cleared. The upper bit of the *ISAMode* field is cleared and the lower bit is loaded from the lower bit of the GPR.

Figure 5.33 DEPC Register Format

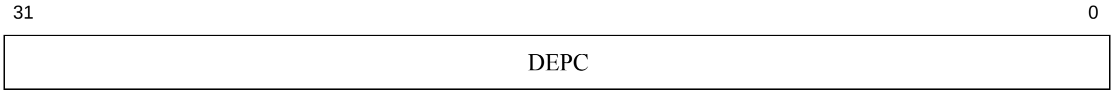


Table 5.38 DEPC Register Field Description

Fields				
Name	Bit(s)	Description	Read/Write	Reset
DEPC	31:0	The *DEPC* register is updated with the virtual address of the instruction that caused the debug exception. If the instruction is in the branch delay slot or forbidden slot, the virtual address of the immediately preceding branch or jump instruction is placed in this register. Execution of the DERET instruction causes a jump to the address in the *DEPC*.	R/W	0

5.2.34 Performance Counter Register (CP0 Register 25, select 0-3)

The M6200 processor defines two performance counters and two associated control registers, which are mapped to CP0 register 25. The select field of the MTC0/MFC0 instructions are used to select the specific register accessed by the instruction, as shown in Table 5.39.

Table 5.39 Performance Counter Register Selects

Select[2:0]	Register
0	Register 0 Control
1	Register 0 Count
2	Register 1 Control
3	Register 1 Count

Each counter is a 32-bit read/write register and is incremented by one each time the countable event, specified in its associated control register, occurs. Each counter can independently count one type of event at a time.

Bit 31 of each of the counters are ANDed with an interrupt enable bit, *IE*, of their respective control register to determine if a performance counter interrupt should be signalled. The two values are then ORed together to create the *SI_PCI* output. This signal is combined with one of the *SI_Int* pins to signal an interrupt to the M6200. Traditionally, this signal is combined with one of the *SI_Int* pins to signal an interrupt to the M6200. However, this is no longer needed as the core will internally route the interrupt to the IP number set by the *IntCtl.IPPCI* field. Counting is not affected by the interrupt indication. This output is cleared when the counter wraps to zero, and may be cleared in software by writing a value with bit 31 = 0 to the *Performance Counter Count* registers.

NOTE: The performance counter registers are connected to a clock that is stopped when the processor is in sleep mode (if the top-level clock gater is present). Most events would not be active during that time, but others would be, notably the cycle count. This behavior should be considered when analyzing measurements taken on a system. Further, note that FPGA implementations of the core would generally not have the clock gater present and thus would have different behavior than a typical ASIC implementation.

Figure 5.34 Performance Counter Control Register

31	30 15	14 11	10 5	4	3	2	1	0
M	0	EventExt	Event	IE	U	0	K	EXL

Table 5.40 Performance Counter Control Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
M	31	If this bit is one, another pair of *Performance Control* and *Counter* registers is implemented at an MTC0 or MFC0 select field value of ‘n+2’ and ‘n+3’.	R	Preset
EventExt	14:11	Event specific to Virtualization Module if supported. Possible events are listed in Table 5.41.	R/W	Undefined
Event	10:5	Counter event enabled for this counter. Possible events are listed in Table 5.41.	R/W	Undefined
IE	4	Counter Interrupt Enable. This bit masks bit 31 of the associated count register from the interrupt exception request output.	R/W	0
U	3	Count in User Mode. When this bit is set, the specified event is counted in User Mode.	R/W	Undefined
K	1	Count in Kernel Mode. When this bit is set, count the event in Kernel Mode when *EXL* and *ERL* both are 0.	R/W	Undefined
EXL	0	Count when *EXL*. When this bit is set, count the event when *EXL* = 1 and *ERL* = 0.	R/W	Undefined
0	30:12, 2	Must be written as zeroes; returns zeroes when read.	0	0

Table 5.41 describes the events countable with the two performance counters. The mode column indicates whether the event counting is influenced by the mode bits (*U*,*K*,*EXL*). The operation of a counter is **UNPREDICTABLE** for events which are specified as Reserved.

Performance counters never count in debug mode or when *ERL* = 1.

The performance counter resets to a low-power state, in which none of the counters will start counting events until software has enabled event counting, using an MTC0 instruction to the Performance Counter Control Registers.

Table 5.41 Performance Counter Events Sorted by Event Number

Event Num	Counter 0	Mode	Counter 1	Mode
0	Cycles	No	Cycles	No
1	Instructions completed	Yes	Instructions completed	Yes
2	Branch instructions	Yes	Branch taken	Yes
3	Jump register r31 (return) instructions	Yes	Reserved	NA
4	Jump register (not r31) instructions	Yes	Reserved	NA
5	Reserved	NA	Reserved	NA
6	Reserved	NA	Reserved	NA
7	Reserved	NA	Reserved	NA
8	Reserved	NA	Reserved	NA

Table 5.41 Performance Counter Events Sorted by Event Number (Continued)

Event Num	Counter 0	Mode	Counter 1	Mode
9	Reserved	NA	Reserved	NA
10	Reserved	NA	Reserved	NA
11	Reserved	NA	Reserved	NA
12	Reserved	NA	Reserved	NA
13	Reserved	NA	Reserved	NA
14	integer instructions completed	Yes	Reserved	NA
15	loads completed	Yes	Stores completed	Yes
16	J/JAL completed	Yes	microMIPS instructions completed	Yes
17	no-ops completed	Yes	Integer multiply/divide completed	Yes
18	Stall cycles	No	Reserved	NA
19	SC instructions completed	Yes	SC instructions failed	Yes
20	Prefetch instructions completed	Yes	Reserved	NA
21	Reserved	NA	Reserved	NA
22	Reserved	NA	Reserved	NA
23	Exceptions taken	Yes	Reserved	NA
24	Reserved	NA	Reserved	NA
25	Reserved	NA	ALU stall cycles	No
26	Instruction Tagram Access	NA	Reserved	NA
27	Instruction Dataram Access	NA	Reserved	NA
28	Instruction WSram Access	NA	Reserved	NA
29	Reserved	NA	Reserved	N/A
30	Data Tagram Access	NA	Reserved	NA
31	Data Dataram Access	NA	Reserved	NA
32	Data WSram Access	NA	Reserved	NA
33	Reserved	NA	Reserved	NA
34	Reserved	NA	Reserved	NA
35	Reserved	NA	CP2 To/From Instructions completed	Yes
36	Reserved	NA	Reserved	NA
37	Reserved	NA	Reserved	NA
38	Reserved	NA	Reserved	NA
39	Reserved	NA	Reserved	NA
40	Uncached stall cycles	Yes	Reserved	NA
41	MDU stall cycles	Yes	Reserved	NA
42	CP2 stall cycles	Yes	Reserved	Yes
43	Reserved	NA	Reserved	NA
44	Reserved	NA	Reserved	NA
45	Load to Use stall cycles	Yes	Reserved	NA

Table 5.41 Performance Counter Events Sorted by Event Number (Continued)

Event Num	Counter 0	Mode	Counter 1	Mode
46	Other interlock stall cycles	Yes	Reserved	NA
47	Reserved	NA	Reserved	NA
48	Reserved	NA	Reserved	NA
49	Debug Instruction Trigger/Breakpoints	Yes	Debug Data Trigger/Breakpoints	Yes
50	Reserved	NA	Reserved	NA
51	Reserved	NA	Reserved	NA
52	Reserved	NA	Reserved	NA
53	Reserved	NA	Reserved	NA
54	Reserved	NA	Reserved	NA
55	Reserved	NA	Reserved	NA
63	User-defined 0	No	User-defined 1	No
56, 64-1023	Reserved	NA	Reserved	NA

Table 5.42 Performance Counter Event Descriptions Sorted by Event Type

Event Name	Counter	Event Number	Description
Cycles	0/1	0	Total number of cycles. The performance counters are clocked by the top-level gated clock. If the M6200 is built with that clock gater present, none of the counters will increment while the clock is stopped, e.g., due to a WAIT instruction.
Instruction Completion: The following events indicate completion of various types of instructions			
Instructions	0/1	1	Total number of instructions completed.
Branch instns	0	2	Counts all branch instructions that completed.
Jump register r31 (return) instns	0	3	Counts all Jump R31 instructions that completed.
Branch taken	1	2	Counts all branch instructions successfully taken
Jump register (not r31)	0	4	Counts all Jump $xx (not $31) and Jump and Link instructions (indirect jumps).
Integer instns	0	14	Non-floating-point, non-Coprocessor 2 instructions.
Loads	0	15	Includes both integer and coprocessor loads.
Stores	1	15	Includes both integer and coprocessor stores.
J/JAL	0	16	Direct Jump (And Link) instruction.
microMIPS	1	16	All microMIPS instructions.
no-ops	0	17	This includes all instructions that normally write to a GPR, but where the destination register was set to r0.
Integer Multiply/Divide	1	17	Counts all Integer Multiply/Divide instructions.
SC	0	19	Counts conditional stores regardless of whether they succeeded.

Table 5.42 Performance Counter Event Descriptions Sorted by Event Type (Continued)

Event Name	Counter	Event Number	Description
SC Fail	1	19	Counts conditional stores that failed, including incorrect programming such as missing LL instruction or incorrect address pairs.
PREF	0	20	Note that this only counts PREFs that are actually attempted. PREFs to uncached addresses or ones with translation errors are not counted.
Cp2 To/From instns	1	35	Includes move to/from, control to/from, and cop2 loads and stores.
Instruction execution events			
SC instructions failed	1	19	SC instruction that did not update memory. Note: While this event and the SC instruction count event can be configured to count in specific operating modes, the timing of the events is much different, and the observed operating mode could change between them, causing some inaccuracy in the measured ratio.
Exceptions Taken	0	23	Any type of exception taken.
Debug instruction triggers	0	49	Number of times an Debug Instruction Trigger Point condition matched (regardless of whether a debug exception occurred).
Debug data triggers	1	49	Number of times an Debug Data Trigger Point condition matched.
General Stalls			
ALU stall cycles	1	25	Counts the number of cycles in which the ALU pipeline cannot advance.
Stall cycles	0	18	Counts the total number of cycles in which no instructions are issued by SRAM to ALU (the RF stage does not advance). This includes both of the previous two events. However, this is different from the sum of them, because cycles when both stalls are active will only be counted once.
Specific stalls - these events will count the number of cycles lost due to this. This will include bubbles introduced by replays within the pipe. If multiple stall sources are active simultaneously, the counters for each of the active events will be incremented.			
Uncached stall cycles	0	40	Cycles in which the processor is stalled on an uncached fetch, load, or store.
MDU stall cycles	0	41	Counts all cycles in which the integer pipeline waits on MDU return data.
Cp2 stall cycles	0	42	Counts all cycles in which the integer pipeline waits on CP2 return data.
Load to Use stall cycles	0	45	Counts all cycles in which the integer pipeline waits on Load return dependent data.
Other interlocks stall cycles	0	46	Counts all cycles in which the integer pipeline waits on return data from MFC0 and RDHWR instructions.
Implementation-specific events - Modules that can be replaced by the customer will have an event signal associated with them.			
User-defined	0/1	63	Two ports are added to the core, which allows the user to connect externally user-defined driven events to be counted using the performance counter. Connected via PM_USER_0 and PM_USER_1.

Figure 5.35 Performance Counter Count Register

31 0
Counter

Table 5.43 Performance Counter Count Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Counter	31:0	Counter	R/W	Undefined

5.2.35 ErrCtl Register (CP0 Register 26, Select 0)

The *ErrCtl* register controls parity/ECC protection of data and instruction SRAM. Parity and ECC protection can be enabled or disabled using the *PE* bit.

Figure 5.36 ErrCtl Register Format

31	30 29	28	27	26 0
EE	Reserved	SPR	PE	Reserved

Table 5.44 Errctl Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>EE</td><td>31</td><td>ECC Enable. If ECC is configured, this bit enables or disables ECC protection for SRAM.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>ECC disabled</td></tr>
<tr><td>1</td><td>ECC enabled</td></tr>
</table>
This field is writable only if the ECC option was configured when the M6200 was built and at least one of the following is true: *IS_ECCPresent* is asserted or *DS_ECCPresent* is asserted. If ECC is not configured, this field is always read as 0. Software can test for ECC support by attempting to write a 1 to this field, then reading back the value.</td><td>R or R/W</td><td>0</td></tr>
<tr><td>SPR</td><td>28</td><td>When asserted, all Store Word (SW) instructions are redirected from the DS interface to the IS interface of the M6200 core.</td><td>R/W</td><td>0</td></tr>
</table>

Table 5.44 Errctl Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>PE</td><td>27</td><td>Parity Enable. If parity is configured, this bit enables or disables parity protection for SRAM.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Parity disabled</td></tr>
<tr><td>1</td><td>Parity enabled</td></tr>
</table>
This field is writable only if the parity option was configured when the M6200 was built and at least one of the following is true: <i>IS_ParPresent</i> is asserted or <i>DS_ParPresent</i> is asserted. If parity not configured, this field is always read as 0. Software can test for parity support by attempting to write a 1 to this field, then reading back the value.</td><td>R/W</td><td>0</td></tr>
<tr><td>R</td><td>26:0</td><td>Reserved</td><td>0</td><td>0</td></tr>
</table>

5.2.36 CacheErr Register (CP0 Register 27, Select 0)

The *CacheErr* register provides an interface with the SRAM Interface Parity or ECC error-detection logic. When an SRAM interface ECC error exception is signaled, the fields of this register are set accordingly.

Figure 5.37 CacheErr Register for Correctable Error

31	30	29	28	27	26	0
ER	EC			BPAR	Error location specifier	

Table 5.45 CacheErr Register Field Descriptions for Correctable Error

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>ER</td><td>31</td><td>This bit indicates whether an uncorrectable instruction error or data error occurred.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Instruction error</td></tr>
<tr><td>1</td><td>Data error</td></tr>
</table></td><td>R</td><td>Undefined</td></tr>
</table>

Table 5.45 CacheErr Register Field Descriptions for Correctable Error (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>EC</td><td>30</td><td>This bit indicates whether an uncorrectable error or correctable error occurred.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Uncorrectable error</td></tr><tr><td>1</td><td>Correctable error</td></tr></table></td><td>R</td><td>Undefined</td></tr>
<tr><td>BPAR</td><td>27</td><td>Bus parity error flag.
This bit indicates whether a parity error occurred.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No parity error detected on the AXI interface</td></tr><tr><td>1</td><td>Parity error detected on DS or ISthe interface</td></tr></table>Software can use this bit to distinguish between a parity error and an ECC uncorrectable error.</td><td>R</td><td>Undefined</td></tr>
<tr><td>ELS</td><td>26:0</td><td>Error Location Specifier. Refer to Table 5.46. If no correctable errors are detected (EC=0), the Error Location Specifier field is not valid and should not be used.</td><td>R</td><td>Undefined</td></tr>
</table>

Table 5.46 Error Location Specifier for SRAM Data Array (DTWS=00)

Fields		Description	Read / Write	Reset State
Name	Bits			
Reserved	29 x+1	Reserved. Must be written as zero; returns zero on reads.	R	Undefined

Table 5.46 Error Location Specifier for SRAM Data Array (DTWS=00) (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Bit Number</td><td>x:0</td><td>If ER=1 (DSRAM Data Array), then x=3, where [3:0] indicates the error bit in the byte or corresponding 5-bit SEC check bits. The ECC code word layout is:
<table>
<tr><td>D7</td><td>D6</td><td>D5</td><td>D4</td><td>P3</td><td>D3</td><td>D2</td><td>D1</td><td>P2</td><td>D0</td><td>P1</td><td>P0</td><td>P</td></tr>
<tr><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>
8-bit byte is {D7, D6, D5, D4, D3, D2, D1, D0};
5-bit ECC checkbit is {P3,P2,P1,P0,P}. For example, if [3:0] is 7, then the error bit is D3; if [3:0] is 4, error bit is P2.

If ER=0 (ISRAM Data Array), then x=6, where [6:0] indicates the error bit in the 32-bit quantity or in the corresponding 7-bit SEC check bits. The ECC code word layout is:
<table>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td>D63</td><td>D62</td><td>D61</td><td>D60</td><td>D59</td><td>D58</td><td>D57</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td>71</td><td>70</td><td>69</td><td>68</td><td>67</td><td>66</td><td>65</td></tr>
<tr><td>P6</td><td>D56</td><td>D55</td><td>D54</td><td>D53</td><td>D52</td><td>D51</td><td>D50</td><td>D49</td><td>D48</td><td>D47</td><td>D46</td><td>D45</td></tr>
<tr><td>64</td><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>54</td><td>53</td><td>52</td><td>51</td></tr>
<tr><td>D44</td><td>D43</td><td>D42</td><td>D41</td><td>D40</td><td>D39</td><td>D38</td><td>D37</td><td>D36</td><td>D35</td><td>D34</td><td>D33</td><td>D32</td></tr>
<tr><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td></tr>
<tr><td>D31</td><td>D30</td><td>D29</td><td>D28</td><td>D27</td><td>D26</td><td>D25</td><td>D24</td><td>D23</td><td>D22</td><td>D21</td><td>D20</td><td>D19</td></tr>
<tr><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td></tr>
<tr><td>D19</td><td>D18</td><td>D17</td><td>D16</td><td>D15</td><td>D14</td><td>D13</td><td>D12</td><td>D11</td><td>D10</td><td>D9</td><td>D8</td><td>D7</td></tr>
<tr><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td></tr>
<tr><td>D7</td><td>D6</td><td>D5</td><td>D4</td><td>P3</td><td>D3</td><td>D2</td><td>D1</td><td>P2</td><td>D0</td><td>P1</td><td>P0</td><td>P</td></tr>
<tr><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>
</td><td></td><td></td></tr>
</table>

5.2.37 CacheErrAddr Register (CP0 Register 27, Select 1)

This register contains the address of the error in the SRAM.

Figure 5.38 CacheErrAddr Register

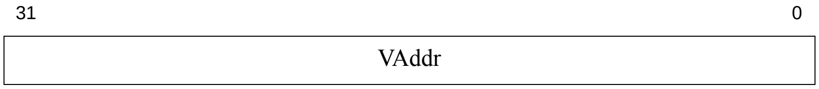


Table 5.47 CacheErrAddr Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
VAddr	31:0	Virtual Address for error data.	R/W	Undefined

5.2.38 ErrorEPC (CP0 Register 30, Select 0)

The *ErrorEPC* register is a read/write register, similar to the *EPC* register, except that *ErrorEPC* is used on error exceptions. All bits of the *ErrorEPC* register are significant and must be writable. It is also used to store the program counter on Reset, Soft Reset, nonmaskable interrupt (NMI) exceptions, and parity and ECC error exceptions.

The *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. This address can be:

- The virtual address of the instruction that caused the exception
- The virtual address of the immediately preceding branch or jump instruction when the error causing instruction is in a branch delay slot or forbidden slot

Unlike the *EPC* register, there is no corresponding branch delay slot or forbidden slot indication for the *ErrorEPC* register.

A read of the *ErrorEPC* register (via MFC0) returns the following value in the destination GPR:

```
GPR[rt] ← ErrorExceptionPC$_{31..1}$ || ISAMode$_0$
```

That is, the upper 31 bits of the error exception PC are combined with the lower bit of the *ISAMode* field and written to the GPR.

Similarly, a write to the *ErrorEPC* register (via MTC0) takes the value from the GPR and distributes that value to the error exception PC and the *ISAMode* field, as follows

```
ErrprExceptionPC ← GPR[rt]$_{31..1}$ || 0
ISAMode ← 2#0 || GPR[rt]$_0$
```

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the error exception PC, and the lower bit of the error exception PC is cleared. The upper bit of the *ISAMode* field is cleared and the lower bit is loaded from the lower bit of the GPR.

Figure 5.39 ErrorEPC Register Format

31	0
ErrorEPC	

Table 5.48 ErrorEPC Register Field Description

Fields				
Name	Bit(s)	Description	Read/Write	Reset State
ErrorEPC	31:0	Error Exception Program Counter.	R/W	Undefined

5.2.39 DeSave Register (CP0 Register 31, Select 0)

The *Debug Exception Save* (*DeSave*) register is a read/write register that functions as a simple memory location. This register is used by the debug exception handler to save one of the GPRs that is then used to save the rest of the context to a pre-determined memory area (such as in the Debug Probe). This register allows the safe debugging of exception handlers and other types of code in which the existence of a valid stack for context saving cannot be assumed.

Figure 5.40 DeSave Register Format

31	0
DESAVE	

Table 5.49 DeSave Register Field Description

Fields				
Name	Bit(s)	Description	Read/Write	Reset State
DESAVE	31:0	Debug exception save contents.	R/W	Undefined

5.2.40 KScratch*n* Registers (CP0 Register 31, Selects 2 to 7)

The *KScratchn* registers are optional read/write registers available for scratchpad storage by kernel-mode software. These registers are 32 bits in width for 32-bit processors and 64 bits for 64-bit processors.

The existence of these registers is indicated by the *KScrExist* field in the *Config4* register. The *KScrExist* field specifies which of the selects are populated with a kernel scratch register.

Debug-mode software should not use these registers, but should instead use the *DeSave* register. If the Debug interface is implemented, select 0 should not be used for a *KScratch* register. Select 1 is being reserved for future debug use and should not be used for a *KScratch* register.

Figure 5.41 KScratch*n* Register Format

31	0
Data	

Table 5.50 KScratch*n* Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Data	31:0	Scratch pad data saved by kernel software.	R/W	Undefined

Chapter 6

Hardware and Software Initialization of the M6200 Core

The M6200 processor core contains only a minimal amount of hardware initialization and relies on software to fully initialize the device.

6.1 Hardware-Initialized Processor State

The M6200 processor core, like most other MIPS processors, is not fully initialized by hardware reset. Only a minimal subset of the processor state is cleared. This is enough to bring the core up while running in unmapped and uncached code space. All other processor state can then be initialized by software. *SI_ColdResetN* is asserted after power-up to bring the device into a known state. Soft reset can be forced by asserting the *SI_WarmResetN* pin. This distinction is made for compatibility with other MIPS processors.

6.1.1 Coprocessor 0 State

Much of the hardware initialization occurs in Coprocessor 0.

- $Status_{BEV}$ - cleared to 1 on Reset/SoftReset
- $Status_{SR}$ - cleared to 0 on Reset, set to 1 on SoftReset
- $Status_{NMI}$ - cleared to 0 on Reset/SoftReset
- $Status_{ERL}$ - set to 1 on Reset/SoftReset
- *Config* fields related to static inputs - set to input value by Reset/SoftReset
- $Config_{K0}$ - set to 010 (uncached) on Reset/SoftReset
- $Debug_{DM}$ - cleared to 0 on Reset/SoftReset (unless DbgBOOT option is used to boot into Debug Mode. Refer to Chapter 8, “Debug Support in the M6200 Core” on page 152 for details.
- $Debug_{LSNM}$ - cleared to 0 on Reset/SoftReset
- $Debug_{IBusEP}$ - cleared to 0 on Reset/SoftReset
- $Debug_{DBusEP}$ - cleared to 0 on Reset/SoftReset
- $Debug_{IEXI}$ - cleared to 0 on Reset/SoftReset
- $Debug_{SSt}$ - cleared to 0 on Reset/SoftReset

6.1.2 Bus State Machines

All pending bus transactions are aborted and the state machines in the SRAM interface unit are reset when a Reset or SoftReset exception is taken.

6.1.3 Static Configuration Inputs

All static configuration inputs should only be changed during Reset.

6.1.4 Fetch Address

Upon Reset/SoftReset, DbgBOOT option, or *SI_UseExceptionBase* option is used, the fetch is directed to VA 0xBFC00000 (PA 0x1FC00000). This address is in KSeg1,which is unmapped and uncached.

6.2 Software Initialized Processor State

Software is required to initialize the following parts of the device.

6.2.1 Register File

The register file powers up in an unknown state with the exception of r0 which is always 0. Initializing the rest of the register file is not required for proper operation in hardware. However, when simulating the operation of the core, unknown values can cause problems. Thus, initializing the register file in the boot code may avoid simulation problems.

6.2.2 Coprocessor 0 State

Miscellaneous CP0 states need to be initialized prior to leaving the boot code. There are various exceptions which are blocked by *ERL*=1 or *EXL*=1 and which are not cleared by Reset. These can be cleared to avoid taking spurious exceptions when leaving the boot code.

- *Cause*: SW0/1 (Software Interrupts) should be cleared.

- *Config*: Typically, the K0, KU and K23 fields should be set to the desired Cache Coherency Algorithm (CCA) value prior to accessing the corresponding memory regions. But in the M6200 core, all CCA values are treated identically, so the hardware reset value of these fields need not be modified.

- *Count*: Should be set to a known value if Timer Interrupts are used.

- *Compare*: Should be set to a known value if Timer Interrupts are used. The write to compare will also clear any pending Timer Interrupts (Thus, *Count* should be set before *Compare* to avoid any unexpected interrupts).

- *Status*: Desired state of the device should be set.

- Other CP0 state: Other registers should be written before they are read. Some registers are not explicitly writeable, and are only updated as a by-product of instruction execution or a taken exception. Uninitialized bits should be masked off after reading these registers.

Chapter 7

Power Management of the M6200 Core

The M6200 processor core offers a number of power management features, including low-power design, active power management and power-down modes of operation. The core is a static design that supports a WAIT instruction designed to signal the rest of the device that execution and clocking should be halted, reducing system power consumption during idle periods.

7.1 Register-Controlled Power Management

Three pins, *SI_EXL*, *SI_ERL*, and *EJ_DebugM*, support the power-management function by allowing the user to change the power state if an exception or error occurs while the core is in a low power state.

If an interrupt is taken while the device is in power down mode, that interrupt may need to be serviced depending on the needs of the application. The interrupt causes an exception which in turn causes the *EXL* bit to be set. The setting of the *EXL* bit causes the assertion of the *SI_EXL* signal on the external bus, indicating to the external agent that an interrupt has occurred. At this time the external agent can choose to either speed up the clocks and service the interrupt or let it be serviced at the lower clock speed.

The setting of the *ERL* bit causes the assertion of the *SI_ERL* signal on the external bus, indicating to the external agent that an error has occurred. At this time the external agent can choose to either speed up the clocks and service the error or let it be serviced at the lower clock speed.

Similarly, the *EJ_DebugM* signal indicates that the processor is in debug mode. Debug mode is entered when the processor takes a debug exception. If fast handling of this is desired, the external agent can speed up the clocks.

The core provides three power-down signals that are part of the system interface. Two of the pins change state as the corresponding bits in the CP0 *Status* register are set or cleared. The third pin indicates that the processor is in debug mode:

- The *SI_EXL* signal represents the state of the *EXL* bit (1) in the CP0 *Status* register.
- The *SI_ERL* signal represents the state of the *ERL* bit (2) in the CP0 *Status* register.
- The *EJ_DebugM* signal indicates that the processor has entered debug mode.

7.2 Instruction-Controlled Power Management

The second mechanism for invoking power down mode is through execution of the WAIT instruction. If the bus is idle at the time the WAIT instruction reaches the E stage of the pipeline the internal clocks are suspended and the pipeline is frozen. However, the internal timer and some of the input pins (*SI_Int[7:0]*, *SI_NMI*, *SI_WarmResetN*, *SI_ColdResetN*, and *EJ_DINT*) continue to run. If the bus is not idle at the time the WAIT instruction reaches the W stage, the pipeline stalls until the bus becomes idle, at which time the clocks are stopped. When the CPU is in instruction controlled power management mode, any enabled interrupt, NMI, debug interrupt, or reset condition causes the

CPU to exit this mode and resume normal operation. While the part is in this low-power mode, the *SI_SLEEP* signal is asserted to indicate to external agents what the state of the chip is.

Chapter 8

Debug Support in the M6200 Core

The M6200 core provides for a optional Debug interface via the MIPS Debug Hub (MDH) and Advanced Peripheral Bus (APB) interface. MDH provides the capability for connection to a JTAG or APB compatible debug system for improved debug performance and support for multi-core systems. The MDH and APB are described in the *MIPS® Debug Hub Technical Reference Manual* [6].

The M6200 core also provides a special Debug mode of operation, in addition to standard User mode and Kernel modes of operation. Debug mode is entered after a debug exception is taken and continues until a debug exception return (DERET) instruction is executed. During this time, the processor executes the debug exception handler routine.

8.1 APB Devices

In order to make APB devices compatible with the ARM CoreSight standard, which includes the APB as the debug bus, the MDH uses the same minimal address mapping of devices. This mapping includes an area for a ROM block that contains the read-only memory words which identify the SoC and each device. A ROM slot occupies a minimum of 4KB (A[11:0]) and starts at address 0 (address range of 0x0-0xFFC). Core debug registers are assigned addresses from 0x000 to 0x07F. Refer to the table below.

Table 8.1 Core Debug Register Address Map

Address	Contents
0x000 - 0xFFC	4KB space as defined by CoreSight
0x000-0xEFC	Device-specific registers (Figure B2-1 in IHI0029D)
0xFCC-0xFCF	DEVTYPE register
0xFD0-0xFEF	Peripheral ID registers
0xFF0-0xFFF	Component ID registers

Values in the ROM table determine the start of the core debug registers (refer to Section 10.2.1 of IHI0031C).

8.2 Core APB Debug Registers

This section describes the APB-accessible debug registers that provide status and control of debug operations via the MDH and APB.

The table below is a list of the Core APB Debug Registers, their offset addresses, names, function, and access types.

The hex values listed in the first column are APB A[6:2] addresses. All registers are accessed as 32 bits thus A1 and A0 are always 0.

Table 8.1 Core APB Debug Registers

Address A[6:2] Register Name	Short Description	Function	R/W
0x01 IDCODE	Device ID	Identifies device's manufacturer, part number, revision, and other device-specific information.	R
0x02 IMPCODE	Implementation code	Identifies main debug features implemented and accessible.	R
0x03 ADDRESS	dmseg address	Address register for processor access.	R
0x04 DATA	Data	Data register for processor read/write access. A write to Data will clear PrAcc. If PrAcc was not set at time probe wrote Data, write cycle will stall (PREADY=0) until core does load or fetch from DMSEG. A read from Data returns value and clears PrAcc. If PrAcc was not set, read will stall (PREADY=0) until core does store to DMSEG.	R/W
0x05 OCI CONTROL	Debug Control	OCI CONTROL Register (OCR) provides access to debug status and control.	R/W
0x06 DRSEG_ADDR	Address of drseg access	Set by probe with address of drseg to read or write. A[30:20] are ignored. A[31] = 0 indicates DRSEG_ADDR does not auto-increment A[31] = 1 indicates DRSEG_ADDR auto-increments after a read or write to DRSEG_DATA	R/W
0x07 DRSEG_DATA	Data value of drseg read or write	DRSEG_DATA register is used to read or write 32 bits of data to the drseg memory. The drseg address is supplied by ADDRESS register (previously set). Read: A read cycle does a read from drseg; 32 bits only Write: A write cycle does a write to drseg; 32 bits only Hardware sets PREADY=0 if there is a need to stretch the read or write cycle which could occur if the drseg access is extended by a memory arbiter and/or clock synchronizer.	R/W
0x08 PCSAMPLE1	PCsample lower	Lower 32 bits of PCsample	R
0x09 PCSAMPLE2	PCsample upper	Upper bits of PCsample, right justified	R
0x0A FDSTATUS	FDC Status & channel number	Refer to FDC section below. LS 4 bits are Tx and Rx FIFO status Read: TxChan field holds channel number when data exists in FIFO Write: RxChan field is written with channel number that probe sends to target. RxChan bits are sticky so only need to be written if they change.	R

Table 8.1 Core APB Debug Registers (Continued)

Address A[6:2] Register Name	Short Description	Function	R/W
0x0B FDDATA	FDC Data	Refer to FDC section below. Read: 32 data bits from FIFO Write: 32 data bits into FIFO	R/W
0X0C DBG_OUT	Debug Output	General-purpose register whose outputs are routed out of the core on the EJ_DBG_OUT bus. A probe can write any pattern to this register. The user can connect any or all signals for his own use. The width is 32 bits.	R/W
0xFC8 DEVID	IMG Block Type. 4=CPU.	Device configuration register used for block type. Value - 32h00000004	R

0xFCC
DEVTYPE
	MAJOR=5 (debug),
	SUBTYPE=1 (core)
		Value - 32h00000015	R

0xFD0 PIDR4	[7:4]size=4’b0000 (block size is 4K bytes) [3:0]DES_2 (4’b0010, MIPS JEDEC continuation code)	Value - 32h00000002	R
0xFD4	Reserved	Value - 32h00000000	R
0xFD8	Reserved	Value - 32h00000000	R
0xFDC	Reserved	Value - 32h00000000	R
0xFE0 PIDR0	[7:0]CPU PRID	Value - 32h0000000X	R
0xFE4 PIDR1	[7:4]DES_0(4’b0111, bits 3:0 of MIPS JEDEC code) [3:0]PART_1 (4’b0000)	Value - 32h00000070	R
0xFE8 PIDR2	[7:4] CPU Major Revision [3]1’b1 indicates JEDEC ID is used [2:0]DES_1 (3’b010, bits 6:4 of MIPS JEDEC code	Value - 32h000000XA	R
0xFEC PIDR3	[7:4] CPU Minor Revision [3:0]CMOD (0=customer has not modified component)	Value - 32h000000X0	R/W
0xFF0 CIDR0	8’h0D	Value - 32h0000000D	R
0xFF4 CIDR1	8’hE0 (Generic IP class)	Value - 32h000000E0	R
0xFF8 CIDR2	8’h05	Value - 32h00000005	R
0xFFC CIDR3	8’hB1	Value - 32h000000B1	R

8.2.1 Device Identification (IDCODE) Register

The Device Identification register is defined by IEEE 1149.1, to identify the device's manufacturer, part number, revision, and other device-specific information. Table 8.2 shows the bit assignments defined for the read-only Device Identification Register, and inputs to the core determine the value of these bits.

Figure 8.1 Device Identification Register Format

31 28	27 12	11 1	0
Version	PartNumber	ManufID	R

Table 8.2 Device Identification Register

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
Version	31:28	**Version** (4 bits) This field identifies the version number of the processor derivative.	R	*EJ_Version[3:0]*
PartNumber	27:12	**Part Number** (16 bits) This field identifies the part number of the processor derivative.	R	*EJ_PartNumber[15:0]*
ManufID	11:1	**Manufacturer Identity** (11 bits) Accordingly to IEEE 1149.1-1990, the manufacturer identity code shall be a compressed form of the JEDEC Publications 106-A.	R	*EJ_ManufID[10:0]*
R	0	Reserved	R	1

8.2.2 Implementation Register (IMPCODE)

The Implementation register is a 32-bit read-only register that identifies features implemented in the M6200

Figure 8.2 shows the format of the Implementation register; Table 8.3 describes the Implementation register fields.

Figure 8.2 Implementation Register Format

	31 29	28 27 25	24	23	22 21	20 17	16 15	14	13 11	10 1	0
32/64-bit Processor	Dbgver	0	DINT sup	0	ASID size	0	0	No DMA	Type	TypeInfo	MIPS 32/64

Table 8.3 Implementation Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Power-up State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Dbgver</td><td>31:29</td><td>Indicates the Debug version:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>OCI Single Core Debug Version 1</td></tr>
<tr><td>1-7</td><td>Reserved</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>DINTsup</td><td>24</td><td>Indicates support for DINT signal from probe:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>DINT signal from the probe is not supported by this processor</td></tr>
<tr><td>1</td><td>Probe can use DINT signal to make debug interrupt on this processor</td></tr>
</table></td><td>R</td><td>Externally driven</td></tr>
<tr><td>ASIDsize</td><td>22:21</td><td>Indicates size of the ASID field:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No ASID in implementation</td></tr>
<tr><td>1</td><td>6-bit ASID</td></tr>
<tr><td>2</td><td>8-bit ASID</td></tr>
<tr><td>3</td><td>Reserved</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>NoDMA</td><td>14</td><td>Indicates no Debug DMA support:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Reserved</td></tr>
<tr><td>1</td><td>No JTAG DMA support</td></tr>
</table></td><td>R</td><td>1</td></tr>
<tr><td>Type</td><td>13:11</td><td>Indicates what type of entity is associated with this TAP or Debug block, and whether the TypeInfo field exists.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Legacy value - probably attached to a CPU. TypeInfo field not implemented.</td></tr>
<tr><td>1</td><td>This Debug block is part of the CPU and the TypeInfo field reflects $\mathit{EBase}_{\text{CPUNUM}}$.</td></tr>
<tr><td>2</td><td>This TAP is attached to a CM and the TypeInfo field is not used.</td></tr>
<tr><td>3</td><td>This TAP is attached to a DBU.</td></tr>
<tr><td>4</td><td>This TAP is attached to an MDH.</td></tr>
<tr><td>Others</td><td>Reserved</td></tr>
</table></td><td>R</td><td>1</td></tr>
</table>

Table 8.3 Implementation Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Power-up State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>TypeInfo</td><td>10:1</td><td>Identifier information specific to the type of entity associated with this TAP or Debug block. The attached entity is specified by the Type field.
<table>
<tr><th>Attached Entity</th><th>Meaning</th></tr>
<tr><td>CPU</td><td>Reflects $EBase_{\text{CPUNUM}}$ of the associated CPU</td></tr>
<tr><td>Others</td><td>Reserved</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>MIPS32/64</td><td>0</td><td>Indicates 32-bit or 64-bit processor:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>32-bit processor</td></tr>
<tr><td>1</td><td>64-bit processor</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>0</td><td>28:25, 23, 20:15</td><td>Ignored on writes; return zeros on reads.</td><td>R</td><td>0</td></tr>
</table>

8.2.3 ADDRESS Register

This read-only Address register provides the 32-bit address for a processor access. The physical address that accompanies the processor access is an untranslated access to DMSEG in Debug mode.

8.2.4 DATA Register

The read/write Data register is used for opcode and data transfers during processor accesses. The operation of memory accesses to dmseg requires a handshake between the PrAcc bit in the OCI CONTROL Register and the APB read/write state sequence.

APB Write: The core fetches an instruction from DMSEG and the probe satisfies that by writing 32 bits to DATA. The completion of the APB write cycle clears PrAcc and supplies DATA to the processor. If PrAcc is not set, or the current processor access is not a load or fetch at the time the probe writes to DATA, the write will stall (PREADY=0) until the core does a load or fetch from DMSEG. The same protocol applies when the core does a (memory) load from dmseg.

APB Read: The core does a store to DMSEG. The probe reads DATA, which returns the data value stored and clears PrAcc at the completion of the APB read cycle. If PrAcc is not set. or the current processor access is not a store, the APB read will stall (PREADY=0) until the core does a store to DMSEG. If the core never does a store, the APB cycle can be aborted.

8.2.5 Fastdata

Fastdata provides an efficient way to upload and download data between target memory and debug memory. A program is run on the target to handshake memory to/from the APB via the DMSEG address space.

The following is a probe sequence that implements Fastdata:

- Load fastdata program into memory (for example, 0x80000000), preferably into cache lines.
- Have debug code jump to the fastdata monitor (while remaining in debug mode).
- Fastdata runs in debug mode, reading from DMSEG and writing to the target.
- Probe now writes 32-bit values into the Data register, one at a time. The core debug hardware maintains PREADY=0 until the core has read the Data register and saved the value in system memory.
- Probe completes a set of downloaded values, then jumps back to a DMSEG location, at which time the probe takes control of core instruction execution, continuing debug-mode operation.

NOTE: In non-fastdata mode, the probe can access the PRAcc bit in the OCI CONTROL Register if needed. Any core instruction fetch or read/write to any address in DMSEG will assert the PRAcc bit and handshake a new DATA value with the probe.

8.2.6 OCI CONTROL Register (OCR)

The 32-bit OCI CONTROL Register (OCR) handles processor reset and soft reset indication, Debug Mode indication, access start, finish, size, and read/write indication. The OCR also:

- Controls debug vector location and indication of serviced processor accesses
- Allows a debug interrupt request
- Indicates processor low-power mode
- Allows implementation-dependent processor and peripheral resets

R/W register bits return their written value on a subsequent read, unless other behavior is defined. Internal synchronization ensures that a written value is updated for an immediate subsequent read.

When first attached, the probe must assert ProbTrap and ProbEn in the OCI CONTROL Register to control the core with a probe.

Figure 8.3 shows the format of the OCI CONTROL Register; Table 8.4 describes the OCI CONTROL register fields.

Figure 8.3 OCI CONTROL Register Format

31	30 29	28 24	23	22	21	20	19	18	17	16	15	14	13	12	11 4	3	2	1	0
Rocc	Psz	0	VPED	Doze	Halt	Per Rst	PRn W	Pr Acc	0	Pr Rst	Prob En	Prob Trap	ISAOn Debug	Dbg Brk	0	DM	SLEEP	DPD	BM

Table 8.4 OCI CONTROL Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Rocc</td><td>31</td><td>Indicates if a processor reset or soft reset has occurred since the bit was cleared:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No reset occurred</td></tr>
<tr><td>1</td><td>Reset occurred</td></tr>
</table>
The Rocc bit stays set as long as reset is applied.
This bit must be cleared to acknowledge that the reset was detected. The Debug Control register is not updated in the Update-DR state unless Rocc is 0 or written to 0 at the same time. This is in order to ensure correct handling of the processor access after reset.</td><td>R/W0</td><td>1</td></tr>
<tr><td>Psz</td><td>30:29</td><td>Indicates the size of a pending processor access, in combination with the Address register:
<table>
<tr><th>Encoding</th><th>32-bit Processor MIPS32/64=0</th><th>64-bit Processor MIPS32/64=1</th></tr>
<tr><td>0</td><td>Byte</td><td>Byte</td></tr>
<tr><td>1</td><td>Halfword</td><td>Halfword</td></tr>
<tr><td>2</td><td>Word</td><td>Word, 5-7 bytes</td></tr>
<tr><td>3</td><td>Triple</td><td>Triple, Doubleword</td></tr>
</table>
This field is valid only when a processor access is pending; otherwise, the read value is undefined.</td><td>R</td><td>Undefined</td></tr>
<tr><td>VPED</td><td>23</td><td>For processors with MIPS MT Module, this bit is a status bit that indicates whether the VPE is currently disabled. A value of 1 indicates that the VPE is disabled and the rest of the Debug state is not valid. If this bit is 0, the processor is either not an MT core or it is an MT core that is currently enabled. Hence, a non-MT core must implement this bit and tie it to zero.</td><td>R</td><td>0 for non-MT cores and 1 for MT cores</td></tr>
<tr><td>Doze</td><td>22</td><td>Indicates if the processor is in low-power mode:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Processor is not in low-power mode</td></tr>
<tr><td>1</td><td>Processor is in low-power mode</td></tr>
</table>
Doze indicates Reduced Power (RP), WAIT, and other implementation-dependent low-power modes.
If the implementation does not support low-power modes, then this bit always reads as 0.</td><td>R</td><td>0</td></tr>
</table>

Table 8.4 OCI CONTROL Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Halt</td><td>21</td><td>Indicates if the internal system bus clock is running:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Internal system bus clock is running</td></tr>
<tr><td>1</td><td>Internal system bus clock is stopped</td></tr>
</table>
Halt indicates WAIT, and other implementation-dependent events that stop the system bus clock.
If the implementation does not support a halt state, this bit always reads as 0.</td><td>R</td><td>0</td></tr>
<tr><td>PerRst</td><td>20</td><td>Controls the peripheral reset with implementation-dependent behavior:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No peripheral reset applied</td></tr>
<tr><td>1</td><td>Peripheral reset applied</td></tr>
</table>
This bit PerRst might not have any effect. There is no inherent indication of whether the PerRst is effective, so the user must consult system documentation. When this bit is changed, then it is only guaranteed that the new value has taken effect when it can be read back here.
This bit is read-only (R) and reads as zero if not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>PRnW</td><td>19</td><td>Indicates read or write of a pending processor access:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Read processor access, for a fetch/load access</td></tr>
<tr><td>1</td><td>Write processor access, for a store access</td></tr>
</table>
This value is defined only when a processor access is pending.</td><td>R</td><td>Undefined</td></tr>
<tr><td>PrAcc</td><td>18</td><td>Indicates a pending processor access and controls finishing of a pending processor access. When read:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No pending processor access</td></tr>
<tr><td>1</td><td>Pending processor access</td></tr>
</table>
This bit is set and cleared in hardware by APB cycles rather than probe firmware. If an APB cycle has to be aborted by the MDH, the probe can then check and even change the value of PrAcc</td><td>R/W0</td><td>0</td></tr>
</table>

Table 8.4 OCI CONTROL Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>PrRst</td><td>16</td><td>Controls the processor reset with implementation-dependent behavior:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No processor reset applied</td></tr>
<tr><td>1</td><td>Processor reset applied</td></tr>
</table>
The PrRst bit might not have any effect. There is no inherent indication of an effective PrRst, so the user must consult system documentation.
If a reset occurs on PrRst, then all parts ofthe system are reset. It is not allowed for only some device to be reset.
When this bit is changed then it is guaranteed that the new value has taken effect when it can be read back here. However, because a processor reset clears this bit, then the effect of setting it can be that the bit is cleared when the reset takes effect. In this case, the Rocc bit should be observed to detect that the reset took effect.
This bit is read-only (R) and reads as zero if not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>ProbEn</td><td>15</td><td>Controls whether the probe handles accesses to the dmseg segment through servicing of processors accesses:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Probe will not serve processor accesses</td></tr>
<tr><td>1</td><td>Probe will service processor accesses</td></tr>
</table>
The ProbEn bit is reflected in a read-only bit in the Debug Control Register (DCR) bit 0 (see Section 8.4 on page 166).
When this bit is changed, it is guaranteed that the new value has taken effect in the DCR when it can be read back here. However, a change of the ProbEn prior to setting the DbgBrk bit will be effective for the debug handler.
Not all combinations of ProbEn and ProbTrap are allowed (see Section 4.7 “Debug Exception Processing”).
When first attached, the probe must assert ProbTrap and ProbEn, located in the Debug Control Register to control the core with a probe.</td><td>R/W</td><td>See Section 4.7 on page 68</td></tr>
</table>

Table 8.4 OCI CONTROL Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>ProbTrap</td><td>14</td><td>Controls location of the debug exception vector:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>See Section 4.7 “Debug Exception Processing”.</td></tr>
<tr><td>1</td><td>0xFFFF FFFF FF20 0200</td></tr>
</table>
When ProbTrap=1, the debug exception vector is relocated to probe-controlled Debug memory, at the fixed location 0xFFFF FFFF FF20 0200.
When this bit is changed, it is guaranteed that the new value is indicated to the processor when it can be read back here. However, a change of the ProbTrap prior to setting the DbgBrk bit will be effective at the debug exception.
Not all combinations of ProbEn and ProbTrap are allowed as described in Section 4.7 on page 68.
When first attached, the probe must assert ProbTrap and ProbEn, located in the Debug Control Register to control the core with a probe.</td><td>R/W</td><td>See Section 4.7 on page 68</td></tr>
<tr><td>ISAOnDebug</td><td>13</td><td>Determines the Instruction Set Architecture to be used on a debug exception when ProbTrap=1:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Use MIPS32/MIPS64 ISA</td></tr>
<tr><td>1</td><td>Use microMIPS ISA</td></tr>
</table>
This bit is read-only and returns 0 if microMIPS is not implemented. This is bit read-only and returns 1 if only microMIPS is implemented.</td><td>R/W</td><td>Bit 0 of Config3 ISA field - 1 if only microMIPS implemented; otherwise 0.</td></tr>
<tr><td>DbgBrk</td><td>12</td><td>Requests a Debug Interrupt exception to the processor when this bit is written as 1. The debug exception request is ignored if the processor is already in debug mode at the time of the request. A write of 0 is ignored.
The debug request restarts the processor clock if the processor was in a low-power mode.
The read value indicates a pending Debug Interrupt exception requested through this bit:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No pending Debug Interrupt exception requested through this bit</td></tr>
<tr><td>1</td><td>Pending Debug Interrupt exception</td></tr>
</table>
The read value can, but is not required to, indicate other pending DINT debug requests (for example, through the DINT signal).
This bit is cleared by hardware when the processor enters Debug Mode.</td><td>R/W1</td><td>See Section 8.5 on page 170</td></tr>
<tr><td>DM</td><td>3</td><td>Indicates if the processor is in Debug Mode:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Processor is not in Debug Mode</td></tr>
<tr><td>1</td><td>Processor is in Debug Mode</td></tr>
</table></td><td>R</td><td>0</td></tr>
</table>

Table 8.4 OCI CONTROL Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>SLEEP</td><td>2</td><td>Indicates if the processor is in Sleep Mode:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>CPU is fetching and executing instructions normally.</td></tr>
<tr><td>1</td><td>CPU has executed a Wait instruction</td></tr>
</table>
The purpose of the bit is for a probe to sample to know if some functions are not accessible because the core is in sleep mode and for power consumption reasons, has turned off certain accesses. One function is DRSEG access; if Sleep=1, the probe will need to wake up the core by generating a debug interrupt before accessing DRSEG using APB accesses.</td><td>R</td><td>0</td></tr>
<tr><td>DPD</td><td>1</td><td>DisableProbeDebug. Reflects the state of the signal *EJ_DisableProbeDebug* and allows a probe to discover if probe debug has been disabled.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Probe not disabled</td></tr>
<tr><td>1</td><td>Probe disabled</td></tr>
</table></td><td>R</td><td></td></tr>
<tr><td>BM</td><td>0</td><td>Indicates processor mode of operation. If BOOTMODE is set to NORMAL and the core receives a warm reset, then DbgBrk, ProbTrap, and ProbEn are reset to 0. If BOOTMODE is set to DbgBOOT and the core receives a warm reset then DbgBrk, ProbTrap, and ProbEn are set to 1.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>NORMAL</td></tr>
<tr><td>1</td><td>DbgBOOT</td></tr>
</table>
If the core receives a cold reset, DbgBrk, ProbTrap, and ProbEn are set to 0, regardless of the setting of this bit.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>28:24,
17, 13,
11:4,</td><td>Must be written as zeros; return zeros on reads.</td><td>0</td><td>0</td></tr>
</table>

8.2.7 DRSEG_ADDR Register

DRSEG_ADDR allows a probe to read or write drseg registers located in 0xFF300000 - 0xFF3FFFFF (word-only accesses). The probe first loads a value into DRSEG_ADDR with the drseg address to be read from or written to. Then the probe reads from or writes to a 32-bit value to the DRSEG_DATA register.

A[30:20] are ignored by a write. A[1:0] are always 0.

A[31] is used to determine if the register is auto-incrementing or not.

A[31] = 0 means DRSEG_ADDR is not auto-incrementing

A[31] = 1 means DRSEG_ADDR is auto-incrementing. Hardware increments the value by 4 (word granular) whenever a read or write occurs to DRSEG_DATA. The purpose is to speed up the reading of iFlowtrace data. Once DRSEG_ADDR is set and A[31]=1, successive reads of DRSEG_DATA will read the next 32 bits of trace data. This will speed up the reading of trace data considerably.

8.2.8 DRSEG_DATA Register

DRSEG_DATA is a new register with a new function. It allows a probe to read or write drseg registers located in 0xFF300000 - 0xFF3FFFFF (word-only accesses). The probe must first load DRSEG_ADDR with the drseg address to be read from or written to. Then the probe reads from or writes to a 32 bit value to this register.

It should be noted that the probe can now access the Debug Control Register (DCR), which is mapped to offset 0 of drseg - directly with an APB write to DRSEG_ADDR with value 0 followed by a DRSEG_DATA read or write.

8.2.9 PCSAMPLE Registers

There are two registers to be read for PCSAMPLE. Core hardware must do the following:

- When a read occurs from PCSAMPLE1 register, the lower 32 bits are put on the APB and the upper bits are stored in a temporary register.
- The next read must be from PCSAMPLE2 register where the hardware places the upper bits from the temporary register onto the APB.

Note that if only the PCSAMPLE1 bits are needed there is no reason to sample PCSAMPLE2.

The layout of the two registers is shown below.

Figure 8.4 PCSAMPLE1

31 1	0
PC Address or Data Address [31:1]	NEW

Figure 8.5 PCSAMPLE2

31 27	26 19	18 11	10	9 0
Reserved	GuestID	TC	K	ASID

Because PCSAMPLE is limited to two 32-bit registers. the following restrictions apply to the M6200 core:

- the DCR.PCSe turns all sampling off (0) or on (1) (no change)
- the DCR.DASe field determines execution (0) or data address (1) sampling. The reset value is 0. This is now PC address or data address sampling; one or the other but not both at the same time.

8.2.10 FDSTATUS and FDDATA Registers

There are two addressable registers, FDSTATUS and FDDATA, used by the probe to send and receive data to and from the core over the FDC. Note that TX and RX (transmit and receive) terms are relative to the core, so from the probe point of view the terms are opposite. From the APB perspective, Read and Write functions access different registers and do different operations.

Figure 8.6 FDSTATUS Debug Register

31 24	23 16	15 13	12	11 8	7 4	3	2	1	0
Tx_Count	Rx_Count	0	GenInt	TxChan	RxChan	RxE	RxF	TxE	TxF

The four flag bits indicate the status of the transmit and receive FIFOs. The two that are of interest are:

- TxE: if 0, transmit FIFO is not empty so 1) the TxChan field is valid and 2) the FDDATA register contains transmitted data.

- RxF: if 0, receive FIFO is not full so if there is data to be sent to the target, the channel number can be written to the RxChan field and the 32 bits of data can be written to the FDDATA register.

FDSTATUS Read:

This register provides status of the Tx and Rx FIFOs and the channel number (TxChan) associated with a word transmitted from the target and channel number (RxChan) sent to the target by the probe. When FDC is active, i.e. supported by probe software to provide a “UART over JTAG”, the FDSTATUS is polled to determine if there is transmitted data from the target and polled to determine if there is room to send data to the target.

Probe software could use the TxCount to know how many words of transmitted data are currently in the FIFO but the probe still has to read FDSTATUS to get the channel number associated with each FDDATA so there may not be much utility to using it.

FDSTATUS Write:

When the Receive FIFO is not full the probe can write an entry into it. This consists of 4 channel bits and 32 data bits. The 4 channel bits are written into the RxChan field of FDSTATUS first followed by a write to the FDDATA which cause the 4+32 bits to be written into the Receive FIFO.

The RxChan bits are sticky meaning if multiple words of data are being sent on the same channel (which will be common), the RxChan only has to be written once.

FDDATA READ:

If FDSTATUS.TxE = 0, the probe extracts the FDSTATUS.TxChan field from FDSTATUS, then reads FDDATA and passes the 32+4 bits to the host.

FDDATA Write:

If FDSTATUS.RxF = 0 and the host has data to send to the target, the probe can write another word of data to FDDATA. FDSTATUS.RxChan field must first be written with the channel being sent on, but only once if the channel stays the same.

8.3 CP0 Debug Registers

Four debug registers (*DEBUG*, *DEBUG2*, *DEPC*, and *DESAVE*) are included in the MIPS Coprocessor 0 (CP0) register set. The *DEBUG* and *DEBUG2* registers show the cause of the debug exception and are used for setting up single-step operations. The *DEPC* (Debug Exception Program Counter) register holds the address on which the debug exception was taken, which is used to resume program execution after the debug operation finishes. Finally, the *DESAVE* (Debug Exception Save) register enables the saving of general-purpose registers used during execution of the debug exception handler. These registers are described in detail in Chapter 5, “CP0 Registers of the M6250 Core” on page 111. The M6200 also includes additional debug registers that provide status and control of debug operations. Those registers are described in the following section.

8.4 Debug Control Register (DCR) Register

The Debug Control Register (DCR) controls and provides information about debug issues, and is always provided with the CPU core. The register is memory-mapped in drseg at offset 0x0.

The DataBrk and InstBrk bits indicate if hardware breakpoints are included in the implementation, and debug software is expected to read hardware breakpoint registers for additional information.

Hardware and software interrupts are maskable for non-debug mode with the INTE bit, which works in addition to the other mechanisms for interrupt masking and enabling. NMI is maskable in non-debug mode with the NMIE bit, and a pending NMI is indicated through the NMIP bit.

The Control register is described in Figure 8.7 and Table 8.5

Figure 8.7 DCR Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DbgBrk_Override	0	ENM			PCIM	0	DASQ	DASe	DAS	0				Data Brk	Inst Brk

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IVM	DVM	0	PCRe	RD Vec	CBT	PCS	PCR			PCSe	IntE	NMIE	NMI pend	0	Prob En

Table 8.5 DCR Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>DbgBrk_Override</td><td>31</td><td>Override DbgBrk and DINT disable.

Re-enable DbgBrk and DINT signal during boot.

Allows DbgBrk to be asserted by a probe (or assertion of DINT signal), resulting in a request for a Debug Interrupt exception from the processor. This provides a means of recovering the CPU from crash, hang, loop or low-power mode.

This feature can allow a Debug Executive to communicate with the probe over the Fast Debug Channel (FDC) and provides a host-based debugger the ability to query the target processor via Debug Executive commands, useful for determining cause of hang.

Software can write this bit and read back to determine if the Secure Debug feature is implemented.</td><td>R/W

If not implemented, must be written as zero; return zeros on reads.</td><td>0</td></tr>
<tr><td>ENM</td><td>29</td><td>Endianess in which the processor is running in kernel and Debug Mode:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Little endian</td></tr><tr><td>1</td><td>Big endian</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>PCIM</td><td>26</td><td>Configure PC Sampling to capture all executed addresses or only those that miss the instruction cache This feature is not supported and this bit will read as 0.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>All PCs captured</td></tr><tr><td>1</td><td>Capture only PCs that miss the cache.</td></tr></table></td><td>R/W</td><td>0</td></tr>
<tr><td>R</td><td>25</td><td>Reserved</td><td>R</td><td>0</td></tr>
</table>

Table 8.5 DCR Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>DASQ</td><td>24</td><td>Qualifies Data Address Sampling using a data breakpoint.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>All data addresses are sampled</td></tr>
<tr><td>1</td><td>Sample matches of data breakpoint 0</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>DASe</td><td>23</td><td>Enables Address Sampling.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Instruction Address sampling enabled.</td></tr>
<tr><td>1</td><td>Data Address sampling enabled.</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>DAS</td><td>22</td><td>Indicates if the Data Address sampling is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Data Address Sampling not implemented</td></tr>
<tr><td>1</td><td>Data Address Sampling implemented</td></tr>
</table></td><td>R</td><td>Preset</td></tr>
<tr><td>DataBrk</td><td>17</td><td>Indicates if data hardware breakpoint is implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No data hardware breakpoint implemented</td></tr>
<tr><td>1</td><td>Data hardware breakpoint implemented</td></tr>
</table></td><td>R</td><td>Preset</td></tr>
<tr><td>InstBrk</td><td>16</td><td>Indicates if instruction hardware breakpoint is implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No instruction hardware breakpoint implemented</td></tr>
<tr><td>1</td><td>Instruction hardware breakpoint implemented</td></tr>
</table></td><td>R</td><td>Preset</td></tr>
<tr><td>IVM</td><td>15</td><td>Indicates if inverted data value match on data hardware breakpoints is implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No inverted data value match on data hardware breakpoints implemented</td></tr>
<tr><td>1</td><td>Inverted data value match on data hardware breakpoints implemented</td></tr>
</table></td><td>R</td><td>Preset</td></tr>
</table>

Table 8.5 DCR Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>DVM</td><td>14</td><td>Indicates if a data value store on a data value breakpoint match is implemented:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No data value store on a data value breakpoint match implemented</td></tr><tr><td>1</td><td>Data value store on a data value breakpoint match implemented</td></tr></table></td><td>R</td><td>Preset</td></tr>
<tr><td>PCRe</td><td>12</td><td>PC Sampling rate extended. Values 1000 to 1100 map to values 2^{13} to 2^{17} cycles, respectively. That is, a PC sample is written out every 8192, 16,384, 32,768, 65,536, or 131,072 cycles respectively. Reserved values are 1101, 1110, and 1111. The external probe or software is allowed to set this value to the desired sample rate.</td><td>R/W</td><td>0</td></tr>
<tr><td>RDVec</td><td>11</td><td>Enables relocation of the debug exception vector. The value in the DebugVectorAddr register is used for Debug exceptions when ProbTrap=0,and RDVec=1.</td><td>R/W</td><td>0</td></tr>
<tr><td>CBT</td><td>10</td><td>Indicates if complex breakpoint block is implemented:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No complex breakpoint block implemented</td></tr><tr><td>1</td><td>Complex breakpoint block implemented</td></tr></table></td><td>R</td><td>Preset</td></tr>
<tr><td>PCS</td><td>9</td><td>Indicates if the PC Sampling feature is implemented.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No PC Sampling implemented</td></tr><tr><td>1</td><td>PC Sampling implemented</td></tr></table></td><td>R</td><td>Preset</td></tr>
<tr><td>PCR</td><td>8:6</td><td>PC Sampling rate. Values 000 to 111 map to values 2^5 to 2^{12} cycles, respectively. That is, a PC sample is written out every 32, 64, 128, 256, 512, 1024, 2048, or 4096 cycles respectively. The external probe or software is allowed to set this value to the desired sample rate.</td><td>R/W</td><td>0</td></tr>
<tr><td>PCSe</td><td>5</td><td>If the PC sampling feature is implemented, then indicates whether PC sampling is initiated or not. That is, a value of 0 indicates that PC sampling is not enabled and when the bit value is 1, then PC sampling is enabled and the counters are operational.</td><td>R/W</td><td>0</td></tr>
</table>

Table 8.5 DCR Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IntE</td><td>4</td><td>Hardware and software interrupt enable for Non-Debug Mode, in conjunction with other disable mechanisms:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupt disabled</td></tr><tr><td>1</td><td>Interrupt enabled depending on other enabling mechanisms</td></tr></table></td><td>R/W</td><td>1</td></tr>
<tr><td>NMIE</td><td>3</td><td>Non-Maskable Interrupt (NMI) enable for Non-Debug Mode:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>NMI disabled</td></tr><tr><td>1</td><td>NMI enabled</td></tr></table></td><td>R/W</td><td>1</td></tr>
<tr><td>NMIpend</td><td>2</td><td>Indication for pending NMI:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No NMI pending</td></tr><tr><td>1</td><td>NMI pending</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>ProbEn</td><td>0</td><td>Probe Enable. This bit reflects the ProbEn bit in the OCI CONTROL register:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No accesses to dmseg allowed</td></tr><tr><td>1</td><td>Accesses to dmseg by Debug probe services allowed</td></tr></table></td><td>R</td><td>Same value as ProbEn in OCR</td></tr>
<tr><td>0</td><td>30, 28:27, 25, 21:18, 13, 1</td><td>Must be written as zeros; return zeros on reads.</td><td>0</td><td>0</td></tr>
</table>

8.5 Hardware Breakpoints

There are several types of *simple* hardware breakpoints. These breakpoints stop the normal operation of the CPU and force the system into debug mode. There are two types of simple hardware breakpoints implemented in the M6200 core: Instruction breakpoints and Data breakpoints. Additionally, *complex* hardware breakpoints are included, which allow detection of more intricate sequences of events.

The M6200 core can be configured with the following breakpoint options:

- No data or instruction
- Two data and four instruction breakpoints

- Four data and eight instruction breakpoints, with complex breakpoints

Instruction breakpoints match on instruction execution operations, and the breakpoint is set on the virtual address. A mask can be applied to the virtual address to set breakpoints on a range of instructions.

Data breakpoints match on load/store transactions, and the breakpoint is set on a virtual address value, with the same single address or address range as the Instruction breakpoint. Data breakpoints can be set on a load, a store, or both. Data breakpoints can also be set to match on the operand value of the load/store operation, with byte-granularity masking. Finally, masks can be applied to both the virtual address and the load/store value.

In addition, the M6200 core has a configurable feature to support data and instruction address-range triggered breakpoints, where a breakpoint can occur when a virtual address is either within or outside a pair of 32-bit addresses. Unlike the traditional address-mask control, address-range triggering is not restricted to a power-of-two boundary.

Complex breakpoints utilize the simple instruction and data breakpoints and break when combinations of events are seen. Complex break features include:

- Pass Counters - Each time a matching condition is seen, a counter is decremented. The break or trigger will only be enabled when the counter has counted down to 0.
- Tuples - A tuple is the pairing of an instruction and a data breakpoint. The tuple will match if both the virtual address of the load or store instruction matches the instruction breakpoint, and the data breakpoint of the resulting load or store address and optional data value matches.
- Priming - This allows a breakpoint to be enabled only after other break conditions have been met. Also called sequential or armed triggering.
- Qualified - This feature uses a data breakpoint to qualify when an instruction breakpoint can be taken. Once a load matches the data address and the data value, the instruction break will be enabled. If a load matches the address, but has mis-matching data, the instruction break will be disabled.

8.5.1 Data Breakpoints

Data breakpoints occur on load/store transactions. Breakpoints are set on virtual address values, similar to the Instruction breakpoint. Data breakpoints can be set on a load, a store, or both. Data breakpoints can also be set based on the value of the load/store operation. Finally, masks can be applied to both the virtual address and the load/store value.

Data breakpoints compare the transaction type (TYPE), which may be load or store, the virtual address of the transaction (ADDR), accessed bytes (BYTELANE) and data value (DATA), with the registers for each data breakpoint including masking or qualification on the transaction properties. When a data breakpoint matches, a debug exception and/or a trigger is generated, and an internal bit in the data breakpoint registers is set to indicate that the match occurred. The match is precise in that the debug exception or trigger occurs on the instruction that caused the breakpoint to match.

8.5.2 Complex Breakpoints

The complex breakpoint unit utilizes the instruction and data breakpoint hardware and looks for more specific matching conditions. There are several different types of enabling that allow more exact breakpoint specification. Tuples add an additional condition to data breakpoints of requiring an instruction breakpoint on the same instructions. Pass counters are counters that decrement each time a matching breakpoint condition is taken. When the counter reaches 0, the break or trigger effect of the breakpoint is enabled. Priming allows a breakpoint to only be enabled when another trigger condition has been detected. Data qualification allows instruction breakpoints to only be enabled when a corresponding load data triggerpoint has matched both address and data. Data qualified breakpoints are also disabled if a load is executed that matches on the address portion of the triggerpoint, but has a mismatching data value. The complex breakpoint features can be combined to create very complex sequences to match on.

In addition to the breakpoint logic, the complex break unit also includes a Stopwatch Timer block. This counter can be used to measure time spent in various sections. It can either be free-running, or it can be set up to start and stop counting based on a trigger from instruction breakpoints.

8.5.3 Conditions for Matching Breakpoints

A number of conditions must be fulfilled in order for a breakpoint to match on an executed instruction or a data transaction, and the conditions for matching instruction and data breakpoints are described below. The breakpoints only match for instructions executed in non-debug mode, thus never on instructions executed in debug mode.

The match of an enabled breakpoint can either generate a debug exception or a trigger indication. The *BE* and/or *TE* bits in the *IBCn* or *DBCn* registers are used to enable the breakpoints.

Debug software should not configure breakpoints to compare on an ASID value unless a TLB is present in the implementation.

8.5.3.1 Conditions for Matching Instruction Breakpoints

There are two methods for matching conditions, Equality and Mask, or Address Range.

Equality and Mask

When an instruction breakpoint is enabled, that breakpoint is evaluated for the address of every executed instruction in non-debug mode, including execution of instructions at an address causing an address error on an instruction fetch. The breakpoint is not evaluated on instructions from a speculative fetch or execution, nor for addresses which are unaligned with an executed instruction.

A breakpoint match depends on the virtual address of the executed instruction (PC) which can be masked at bit level. The registers for each instruction breakpoint have the values and mask used in the compare, and the equation that determines the match is shown below in C-like notation.

```
IB_match =
          ( <all 1's> == ( IBMn_IBM | ~ ( PC ^ IBAn_IBA ) )
```

The match indication for instruction breakpoints is always precise, i.e. indicated on the instruction causing the IB_match to be true.

Address Range

Cores may optionally support the address range triggered instruction breakpoints.When this feature is configured, the following changes are made to the instruction breakpoint registers:

- *IBAn* : represents the upper limit of a address range boundary
- *IBMn* : represents the lower limit of the address range boundary

In addition, the following bits must be supported:

IBCn[6].hwarts : a preset value of 1 indicates that the address range triggered instruction breakpoint feature is supported for this particular instruction breakpoint channel. This bit is read-only.

IBCn[5].excl : a value of 0 indicates that the breakpoint will match for addresses inclusive (within) the range defined by *IBMn* and *IBAn*. A value of 1 indicates that the breakpoint will match for addresses exclusive (outside) to the range defined by *IBMn* and *IBAn*. This bit is writeable.

IBCn[4].hwart : a value of 0 indicates that the breakpoint will match using the "Equality and Mask" equation as found in Section 8.5.3.1 "Conditions for Matching Instruction Breakpoints". A value of 1 indicates that the breakpoint will match using address ranges using the equation below:

```
IB_match =
(!IBCnTCuse || ( TC == IBCnTC ) ) &&
( ! IBCnASIDuse || ( ASID == IBASIDnASID ) ) &&
( ((~IBCnhwarts || ~IBCnhwart) &&
     (( IBMnIBM | ~ ( PC ^ IBAnIBA ) ) == ~0) ||
  (( IBCnhwarts &&  IBCnhwart) &&
     ((~IBCnexcl && (IBM <= PC <= IBA)) ||
      ( IBCnexcl && (IBM > PC || PC > IBA)
)
```

Or if microMIPS is supported:

```
IB_range_match =
(!IBCnTCuse || ( TC == IBCnTC ) ) &&
( ! IBCnASIDuse || ( ASID == IBASIDnASID ) ) &&
( ((~IBCnhwarts || ~IBCnhwart) &&
     (( IBMnIBM | ~ ( ( ( PC[MSB:1] << 1 ) + ISAmode ) ^ IBAnIBA ) ) == ~0 ) ||
  (( IBCnhwarts &&  IBCnhwart) &&
   ( IBMnIBM[0] | ~ ( ISAmode ^ IBAnIBA[0] ) ) == ~0) &&
     ((~IBCnexcl && (IBM[MSB:1] <= PC[MSB:1] <= IBA[MSB:1])) ||
      ( IBCnexcl && (IBM[MSB:1] > PC[MSB:1] || PC[MSB:1] > IBA[MSB:1])
)
```

Also note that addresses that overlap a boundary is considered for both exclusive and inclusive breakpoint matches.

8.5.3.2 Conditions for Matching Data Breakpoints

There are two methods for matching conditions, namely 1) by Equality and Mask or 2) by Address Range:

Equality and Mask

When a data breakpoint is enabled, that breakpoint is evaluated for every data transaction due to a load/store instruction executed in non-debug mode, including load/store for coprocessor, and transactions causing an address error on data access. The breakpoint is not evaluated due to a PREF instruction or other transactions which are not part of explicit load/store transactions in the execution flow, nor for addresses which are not the explicit load/store source or destination address.

A breakpoint match depends on the transaction type (TYPE) as load or store, the address, and optionally the data value of a transaction. The registers for each data breakpoint have the values and mask used in the compare, and the equation that determines the match is shown below in C-like notation.

The overall match equation is the DB_match.

```
DB_match =
          ( ( ( TYPE == load ) && ! DBCn$_{NoLB}$ ) ||
            ( ( TYPE == store ) && ! DBCn$_{NoSB}$ ) ) &&
          DB_addr_match && ( DB_no_value_compare || DB_value_match )
```

The match on the address part, DB_addr_match, depends on the virtual address of the transaction (ADDR) and the accessed bytes (BYTELANE) where BYTELANE[0] is 1 only if the byte at bits [7:0] on the bus is accessed, and BYTELANE[1] is 1 only if the byte at bits [15:8] is accessed, etc. The DB_addr_match is shown below.

```
DB_addr_match =
          ( <all 1's> == ( DBMn$_{DBM}$ | ~ ( ADDR ^ DBAn$_{DBA}$ ) ) ) &&
          ( <all 0's> != ( ~ BAI & BYTELANE ) )
```

The size of *DBCn*$_{BAI}$ and BYTELANE is 4 bits.

Data value compare is included in the match condition for the data breakpoint depending on the bytes (BYTELANE as described above) accessed by the transaction, and the contents of breakpoint registers. The DB_no_value_compare is shown below.

```
DB_no_value_compare =
          ( <all 1's> == ( DBCn$_{BLM}$ | DBCn$_{BAI}$ | ~ BYTELANE ) )
```

The size of DBCn$_{BLM}$, DBCn$_{BAI}$ and BYTELANE is 4 bits.

In case a data value compare is required, DB_no_value_compare is false, then the data value from the data bus (DATA) is compared and masked with the registers for the data breakpoint. The *DBCIVM* bit inverts the sense of the match - if set, the value match term will be high if the data value is not the same as the data in the *DBVn* register. The endianess is not considered in these match equations for value, as the compare uses the data bus value directly, thus debug software is responsible for setup of the breakpoint corresponding with endianess.

```
DB_value_match =
   DBCnIVM ^
   (((DATA[7:0] == DBVn$_{DBV[7:0]}$) || ! BYTELANE[0] || DBCn$_{BLM[0]}$ || DBCn$_{BAI[0]}$ ) &&
    ((DATA[15:8] == DBVn$_{DBV[15:8]}$) || ! BYTELANE[1] || DBCn$_{BLM[1]}$ || DBCn$_{BAI[1]}$ ) &&
    ((DATA[23:16] == DBVn$_{DBV[23:16]}$) || ! BYTELANE[2] || DBCn$_{BLM[2]}$ || DBCn$_{BAI[2]}$) &&
    ((DATA[31:24] == DBVn$_{DBV[31:24]}$) || ! BYTELANE[3] || DBCn$_{BLM[3]}$ || DBCn$_{BAI[3]}$ ))
```

The match for a data breakpoint is always precise, since the match expression is fully evaluated at the time the load/store instruction is executed. A true DB_match can thereby be indicated on the very same instruction causing the DB_match to be true.

Address Range

Cores may optionally support the address range triggered data breakpoints. When this feature is configured, the following changes are made to the data breakpoint registers:

- *DBAn* : represents the upper limit of a address range boundary

- *DBMn* : represents the lower limit of the address range boundary

In addition, the following bits must be supported:

DBCn[10].*hwarts*: a preset value of 1 indicates that the address range triggered data breakpoint feature is supported for this particular data breakpoint channel. This bit is read-only.

DBCn[9].*exc* : a value of 0 indicates that the breakpoint will match for addresses inclusive (within) the range defined by *DBMn* and *DBAn*. A value of 1 indicates that the breakpoint will match for addresses exclusive (outside) to the range defined by *DBMn* and *DBAn*. This bit is writeable.

DBCn[8].*hwart*: a value of 0 indicatesthat the breakpoint will match using the “Equality and Mask” equation as found in Section 8.5.3.2 “Conditions for Matching Data Breakpoints”. A value of 1 indicates thatthe breakpoint will match using address ranges using the equation below:

```
DB_match =
(!DBCnTCuse || ( TC == DBCnTC ) ) &&
( ( ( TYPE == load ) && ! DBCnNoLB ) || ( ( TYPE == store ) && ! DBCnNoSB ) ) &&
DB_addr_range_match && ( DB_no_value_compare || DB_value_match )

DB_addr_range_match =
( ! DBCnASIDuse || ( ASID == DBASIDnASID ) ) &&
( ((~DBCnhwarts || ~DBCnhwart) &&
     (( DBMnDBM | ~ ( ADDR ^ DBAnDBA ) ) == ~0 ) ||
  (( DBCnhwarts &&  DBCnhwart) &&
     ((~DBCnexcl && (DBMn <= ADDR <= DBAn)) ||
      ( DBCnexcl && (DBMn > ADDR || ADDR > DBAn)
)
```

When address range triggered data breakpoints is enabled, *DBCn.BLM*[3:0] must be set to 4'b1111 because value matching is not supported with this feature. Addresses that overlap a boundary is considered for both exclusive and inclusive breakpoint matches.

8.5.4 Debug Exceptions from Breakpoints

Instruction and data breakpoints may be set up to generate a debug exception when the match condition is true, as described below.

8.5.4.1 Debug Exception by Instruction Breakpoint

If the breakpoint is enabled by BE bit in the IBCn register, then a debug instruction break exception occurs if the IB_match equation is true. The corresponding BS[n] bit in the IBS register is set when the breakpoint generates the debug exception.

The debug instruction break exception is always precise, so the *DEPC* register and *DBD* bit in the *Debug* register point to the instruction that caused the IB_match equation to be true.

The instruction receiving the debug exception does not update any registers due to the instruction, nor does any load or store by that instruction occur. Thus a debug exception from a data breakpoint can not occur for instructions receiving a debug instruction break exception.

The debug handler usually returns to the instruction causing the debug instruction break exception, whereby the instruction is executed. Debug software is responsible for disabling the breakpoint when returning to the instruction, otherwise the debug instruction break exception reoccurs.

8.5.4.2 Debug Exception by Data Breakpoint

If the breakpoint is enabled by BE bit in the DBCn register, then a debug exception occurs when the DB_match condition is true. The corresponding BS[n] bit in the DBS register is set when the breakpoint generates the debug exception.

A debug data break exception occurs when a data breakpoint indicates a match. In this case the *DEPC* register and *DBD* bit in the *Debug* register points to the instruction that caused the DB_match equation to be true.

The instruction causing the debug data break exception does not update any registers due to the instruction, and the following applies to the load or store transaction causing the debug exception:

- A store transaction is not allowed to complete the store to the memory system.
- A load transaction with no data value compare, i.e., where the `DB_no_value_compare` is true for the match, is not allowed to complete the load.
- A load transaction for a breakpoint with data value compare must occur from the memory system, since the value is required in order to evaluate the breakpoint.

The result of this is that the load or store instruction causing the debug data break exception appears as not executed, with the exception that a load from the memory system does occur for a breakpoint with data value compare, but the register file is not updated by the load.

If both data breakpoints without and with data value compare would match the same transaction and generate a debug exception, then the following rules apply with respect to updating the BS[n] bits.

- On both a load and store the BS[n] bits are required to be set for all matching breakpoints without a data value compare.
- On a store the BS[n] bits are allowed but not required to be set for all matching breakpoints with a data value compare, but either all or none of the BS[n] bits must be set for these breakpoints.
- On a load then none of the BS[n] bits for breakpoints with data value compare are allowed to be set, since the load is not allowed to occur due to the debug exception from a breakpoint without a data value compare, and a valid data value is therefore not returned.

Any BS[n] bit set prior to the match and debug exception are kept set, since BS[n] bits are only cleared by debug software.

The debug handler usually returns to the instruction causing the debug data break exception, whereby the instruction is re-executed. This re-execution may result in a repeated load from system memory, since the load may have occurred previously in order to evaluate the breakpoint as described above. I/O devices with side effects on loads may not be re-accessible without changing the system behavior. The Load Data Value register was introduced to capture the value that was read and allow debug software to synthesize the load instruction without re-accessing memory. Debug software is responsible for disabling breakpoints when returning to the instruction, otherwise the debug data break exception will reoccur.

8.5.5 Breakpoint Used as Triggerpoint

Both instruction and data hardware breakpoints can be setup by software so that a matching breakpoint does not generate a debug exception, but only an indication through the *BS*[*n*] bit. The *TE* bit in the *IBCn* or *DBCn* register controls if an instruction or data breakpoint is used as a so-called triggerpoint. The triggerpoints are, like breakpoints, only compared for instructions executed in non-debug mode.

The BS[n] bit in the IBS or DBS register is set when the respective IB_match or DB_match bit is true.

The triggerpoint feature can be used to start and stop tracing.

8.5.6 Instruction Breakpoint Registers

The registers for instruction breakpoints are described below. These registers have implementation information and are used to set up the instruction breakpoints. All registers are in drseg, and the addresses are shown in Table 8.6.

Table 8.6 Addresses for Instruction Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x1000	*IBS*	Instruction Breakpoint Status
0x1100 + n * 0x100	*IBAn*	Instruction Breakpoint Address n
0x1108 + n * 0x100	*IBMn*	Instruction Breakpoint Address Mask n
0x1110 + n * 0x100	*IBASIDn*	Instruction Breakpoint ASID n
0x1118 + n * 0x100	*IBCn*	Instruction Breakpoint Control n
0x1120 + n * 0x100	*IBCCn*	Instruction Breakpoint Complex Control n
0x1128 + n * 0x100	*IBPCn*	Instruction Breakpoint Pass Counter n
n is breakpoint number in range 0 to 5 (or 3 or 1, depending on the implemented hardware)		

An example of some of the registers; IBA0 is at offset 0x1100 and IBC2 is at offset 0x1318.

8.5.6.1 Instruction Breakpoint Status (IBS) Register (0x1000)

The Instruction Breakpoint Status (IBS) register holds implementation and status information about the instruction breakpoints. This register is required only if instruction breakpoints are implemented.

Figure 8.8 IBS Register Format

31	30	29 28	27 24	23 6	5 0
Res	ASIDsup	Res	BCN	Res	BS

Table 8.7 IBS Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Res	31	Must be written as zero; returns zero on read.	R	0
ASIDsup	30	Indicates that ASID compare is supported in instruction breakpoints. 0: No ASID compare. 1: ASID compare (*IBASIDn* register implemented).	R	0
Res	29:28	Must be written as zero; returns zero on read.	R	0
BCN	27:24	Number of instruction breakpoints implemented.	R	0, 2, 4, 6 or 8[a]
Res	23:8	Must be written as zero; returns zero on read.	R	0
BS	7:0	Break status for breakpoint n is at *BS[n]*, with n from 0 to 7[b]. The bit is set to 1 when the condition for the corresponding breakpoint has matched and *IBCnTE* or *IBCnBE* are set	R/W	Undefined

Table 8.7 IBS Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			

[a] Based on actual hardware implemented.
[b] In case of fewer than 8 Instruction breakpoints the upper bits become reserved.

8.5.6.2 Instruction Breakpoint Address n (IBAn) Register (0x1100 + n * 0x100)

The Instruction Breakpoint Address n (IBAn) register has the address used in the condition for instruction breakpoint *n*. This register is required only if instruction breakpoints are implemented.

Figure 8.9 IBAn Register Format

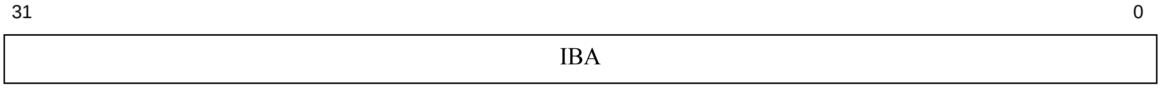


Table 8.8 IBAn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
IBA	31:0	Instruction breakpoint address for condition.	R/W	Undefined

8.5.6.3 Instruction Breakpoint Address Mask n (IBMn) Register (0x1108 + n*0x100)

The Instruction Breakpoint Address Mask n (IBMn) register has the mask for the address compare used in the condition for instruction breakpoint n. A 1 indicates that the corresponding address bit will not be considered in the match. A mask value of all 0’s would require an exact address match, while a mask value of all 1’s would match on any address. This register is required only if instruction breakpoints are implemented.

Figure 8.10 IBMn Register Format

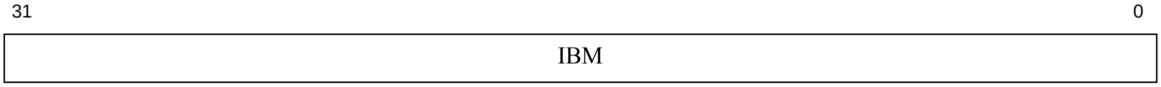


Table 8.9 IBMn Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>IBM</td><td>31:0</td><td>Instruction breakpoint address mask for condition:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Corresponding address bit not masked.</td></tr>
<tr><td>1</td><td>Corresponding address bit masked.</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
</table>

8.5.6.4 Instruction Breakpoint ASID n (IBASIDn) Register (0x1110 + n*0x100)

For processors with a TLB-based MMU, this register is used to define an ASID value to be used in the match expression. On the M6200 processor, this register is reserved and reads as 0. This register is required only if instruction breakpoints are implemented.

Figure 8.11 IBASIDn Register Format

31 8	7 0
Res	ASID

Table 8.10 IBASIDn Register Field Descriptions

Fields				
Name	**Bit(s)**	**Description**	**Read/Write**	**Reset State**
Res	31:8	Must be written as zero; returns zero on read.	R	0
ASID	7:0	Instruction breakpoint ASID value for a compare.	R	0

8.5.6.5 Instruction Breakpoint Control n (IBCn) Register (0x1118 + n*0x100)

The Instruction Breakpoint Control n (IBCn) register controls the setup of instruction breakpoint *n*. This register is required only if instruction breakpoints are implemented.

Figure 8.12 IBCn Register Format

31 24	23	22 7	6	5	4	3	2	1	0
Res	ASIDuse	Res	hwarts	excl	hwart	Res	TE	Res	BE

Table 8.11 IBCn Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read/Write**	**Reset State**
Res	31:24	Must be written as zero; returns zero on read.	R	0
ASIDuse	23	Use ASID value in compare for instruction breakpoint n: **Encoding** / **Meaning** 0 / Don’t use ASID value in compare 1 / Use ASID value in compare	R	0
Res	22:7	Must be written as zero; returns zero on read.	R	0
hwarts	6	A preset value of 1 indicates that the address- range triggered instruction breakpoint feature is supported for this particular instruction breakpoint channel.	R	Preset
excl	5	A value of 0 indicates that the breakpoint will match for addresses within (inclusive of) the range defined by *IBMn* and *IBAn*. A value of 1 indicates that the breakpoint will match for addresses outside (exclusive to) the range defined by *IBMn* and *IBAn*.	R/W	0

Table 8.11 IBCn Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>hwart</td><td>4</td><td>A value of 0 indicates that the breakpoint will match using the “Equality and Mask” equation as found section under 8.5.3.1 “Conditions for Matching Instruction Breakpoints”.
A value of 1 indicates that the breakpoint will match using the “Address Range” equation in section 8.5.3.1 “Conditions for Matching Instruction Breakpoints”</td><td>R/W</td><td>0</td></tr>
<tr><td>Res</td><td>3</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>TE</td><td>2</td><td>Use instruction breakpoint n as triggerpoint:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Don’t use it as triggerpoint</td></tr>
<tr><td>1</td><td>Use it as triggerpoint</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>Res</td><td>1</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>BE</td><td>0</td><td>Use instruction breakpoint n as breakpoint:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Don’t use it as breakpoint</td></tr>
<tr><td>1</td><td>Use it as breakpoint</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
</table>

8.5.6.6 Instruction Breakpoint Complex Control n (IBCCn) Register (0x1120 + n*0x100)

The Instruction Breakpoint Complex Control n (IBCCn) register controls the complex break conditions for instruction breakpoint *n*. This register is required only if complex breakpoints are implemented and only for implemented instruction breakpoints.

Figure 8.13 IBCCn Register Format

31 14	13 10	9	8 5	4	3 2 1 0
Res	PrCnd	CBE	DBrkNum	Q	Res

Table 8.12 IBCCn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
Res	31:14, 3:0	Must be written as zero; returns zero on read.	R	0
PrCnd	13:12	Upper bits of priming condition for instruction breakpoint n. The M6200 core only supports 4 priming conditions, so the upper 2 bits are read as 0.	R	0

Table 8.12 IBCCn Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
PrCnd	11:10	Priming condition for instruction breakpoint n. 00 - Bypass, no priming needed Other - Varies depending on the break number; refer to Table 8.14 for mapping.	R/W	0
CBE	9	Complex Break Enable. Enables this breakpoint for use in a complex sequence as a priming condition for another breakpoint, to start or stop the stopwatch timer, or as part of a tuple breakpoint.	R/W	0
DBrkNum	8:5	Indicates which data breakpoint channel is used to qualify this instruction breakpoint.	R	6I/2D Complex Breakpoint Configuration: IBCC0..2 - 0 IBCC3..6 - 1 8I/4D Complex Breakpoint Configuration: IBCC0..1 - 0 IBCC2..3 - 1 IBCC4..5 - 2 IBCC6..7 - 3
Q	4	Qualify this breakpoint based on the data breakpoint indicated in *DBrkNum*. 0 - Not dependent on qualification 1 - Breakpoint must be qualified to be taken	R/W	0

8.5.6.7 Instruction Breakpoint Pass Counter n (IBPCn) Register (0x1128 + n*0x100)

The Instruction Breakpoint Pass Counter n (IBPCn) register controls the pass counter associated with instruction breakpoint *n*. This register is required only if complex breakpoints are implemented and only for implemented instruction breakpoints.

If complex breakpoints are implemented, there will be an 8b pass counter for each of the instruction breakpoints on the M6200 core.

Figure 8.14 IBPCn Register Format

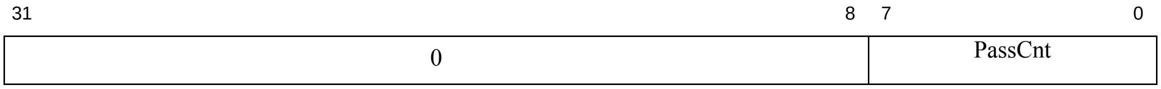


Table 8.13 IBPCn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:8	Ignored on write, returns zero on read.	R	0

Table 8.13 IBPCn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
PassCnt	7:0	Prevents a break/trigger action until the matching conditions on breakpoint n have been seen this number of times. Each time the matching condition is seen, this value will be decremented by 1.When the value reaches 0, subsequent matches will cause a break or trigger as requested and the counter will stay at 0. The break or trigger action is imprecise if the *PassCnt* register was last written to a non-zero value. It will remain imprecise until this register is written to 0 by software. The instruction pass counter should not be set on instruction breakpoints that are being used as part of a tuple breakpoint.	R/W	0

8.5.7 Data Breakpoint Registers

The registers for data breakpoints are described below. These registers have implementation information and are used the setup the data breakpoints. All registers are in drseg, and the addresses are shown in Table 8.14.

Table 8.14 Addresses for Data Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x2000	*DBS*	Data Breakpoint Status
0x2100 + 0x100 * n	*DBAn*	Data Breakpoint Address n
0x2108 + 0x100 * n	*DBMn*	Data Breakpoint Address Mask n
0x2110 + 0x100 * n	*DBASIDn*	Data Breakpoint ASID n
0x2118 + 0x100 * n	*DBCn*	Data Breakpoint Control n
0x2120 + 0x100 * n	*DBVn*	Data Breakpoint Value n
0x2128 + 0x100 * n	*DBCCn*	Data Breakpoint Complex Control n
0x2130 + 0x100 * n	*DBPCn*	Data Breakpoint Pass Counter n
0x2ff0	*DVM*	Data Value Match Register
n is breakpoint number as 0, 1, 2 or 3 (or just 0, depending on the implemented hardware)		

An example of some of the registers; DBM0 is at offset 0x2108 and DBV1 is at offset 0x2220.

8.5.7.1 Data Breakpoint Status (DBS) Register (0x2000)

The Data Breakpoint Status (DBS) register holds implementation and status information about the data breakpoints. This register is required only if data breakpoints are implemented.

Figure 8.15 DBS Register Format

31	30	29 28	27 24	23 2	1 0
Res	ASIDsup	Res	BCN	Res	BS

Table 8.15 DBS Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Res	31	Must be written as zero; returns zero on read.	R	0
ASID	30	Indicates that ASID compares are supported in data breakpoints. 0: Not supported 1: Supported	R	0
Res	29:28	Must be written as zero; returns zero on read.	R	0
BCN	27:24	Number of data breakpoints implemented.	R	4, 2, 1 or 0[a]
Res	23:4	Must be written as zero; returns zero on read.	R	0
BS	3:0	Break status for breakpoint n is at *BS[n]*, with n from 0 to 1[b]. The bit is set to 1 when the condition for the corresponding breakpoint has matched.	R/W0	Undefined

[a] Based on actual hardware implemented.
[b] In case of only 1 data breakpoint bit 1 become reserved.

8.5.7.2 Data Breakpoint Address n (DBAn) Register (0x2100 + 0x100 * n)

The Data Breakpoint Address n (DBAn) register has the address used in the condition for data breakpoint *n*. This register is required only if data breakpoints are implemented.

Figure 8.16 DBAn Register Format

31 0
DBA

Table 8.16 DBAn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
DBA	31:0	Data breakpoint address for condition.	R/W	Undefined

8.5.7.3 Data Breakpoint Address Mask n (DBMn) Register (0x2108 + 0x100 * n)

The Data Breakpoint Address Mask n (DBMn) register has the mask for the address compare used in the condition for data breakpoint n. A 1 indicates that the corresponding address bit will not be considered in the match. A mask value of all 0’s would require an exact address match, while a mask value of all 1’s would match on any address. This register is required only if data breakpoints are implemented.

Figure 8.17 DBMn Register Format

31 … 0
DBM

Table 8.17 DBMn Register Field Descriptions

Fields				
Name	**Bit(s)**	**Description**	**Read/Write**	**Reset State**
DBM	31:0	Data breakpoint address mask for condition: 0: Corresponding address bit not masked 1: Corresponding address bit masked	R/W	Undefined

8.5.7.4 Data Breakpoint ASID n (DBASIDn) Register (0x2110 + 0x100 * n)

For processors with a TLB-based MMU, this register is used to define an ASID value to be used in the match expression. On the M6200 processor, this register is reserved and reads as 0. This register is required only if data breakpoints are implemented.

Figure 8.18 DBASIDn Register Format

31 … 8	7 … 0
Res	ASID

Table 8.18 DBASIDn Register Field Descriptions

Fields				
Name	**Bit(s)**	**Description**	**Read/Write**	**Reset State**
Res	31:8	Must be written as zero; returns zero on read.	R	0
ASID	7:0	Data breakpoint ASID value for compares.	R	0

8.5.7.5 Data Breakpoint Control n (DBCn) Register (0x2118 + 0x100 * n)

The *Data Breakpoint Control n* (*DBCn*) register controls the setup of data breakpoint *n*. This register is required only if data breakpoints are implemented.

Figure 8.19 DBCn Register Format

31 … 24	23	22 … 18	17 … 14	13	12	11	10	9	8	7 … 4	3	2	1	0
Re	ASIDuse	Res	BAI	NoSB	NoLB	Res	hwarts	excl	hwart	BLM	Res	TE	IVM	BE

Table 8.19 DBCn Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Res</td><td>31:24</td><td>Must be written as zero; returns zero on reads.</td><td>R</td><td>0</td></tr>
<tr><td>ASIDuse</td><td>23</td><td>Use ASID value in compare for data breakpoint n:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Don’t use ASID value in compare</td></tr>
<tr><td>1</td><td>Use ASID value in compare</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>Res</td><td>22:18</td><td>Must be written as zero; returns zero on reads.</td><td>R</td><td>0</td></tr>
<tr><td>BAI</td><td>17:14</td><td>Byte access ignore controls ignore of access to a specific byte. <i>BAI[0]</i> ignores access to byte at bits [7:0] of the data bus, <i>BAI[1]</i> ignores access to byte at bits [15:8], etc.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Condition depends on access to corresponding byte</td></tr>
<tr><td>1</td><td>Access for corresponding byte is ignored</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
<tr><td>NoSB</td><td>13</td><td>Controls if condition for data breakpoint is not fulfilled on a store transaction:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Condition may be fulfilled on store transaction</td></tr>
<tr><td>1</td><td>Condition is never fulfilled on store transaction</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
<tr><td>NoLB</td><td>12</td><td>Controls if condition for data breakpoint is not fulfilled on a load transaction:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Condition may be fulfilled on load transaction</td></tr>
<tr><td>1</td><td>Condition is never fulfilled on load transaction</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
<tr><td>Res</td><td>11</td><td>Must be written as zero; returns zero on reads.</td><td>R</td><td>0</td></tr>
<tr><td>hwarts</td><td>10</td><td>A preset value of 1 indicates that the address range triggered data breakpoint feature is supported for this particular data breakpoint channel.</td><td>R</td><td>Preset</td></tr>
<tr><td>excl</td><td>9</td><td>A value of 0 indicates that the breakpoint will match for addresses inclusive (within) the range defined by <i>DBMn</i> and <i>DBAn</i>. A value of 1 indicates that the breakpoint will match for addresses exclusive (outside) of the range defined by <i>DBMn</i> and <i>DBAn</i>.</td><td>R/W</td><td>0</td></tr>
</table>

Table 8.19 DBCn Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>hwart</td><td>8</td><td>A value of 0 indicates that the breakpoint will match using the “Equality and Mask” equation as found section under 8.5.3.2 “Conditions for Matching Data Breakpoints”.
A value of 1 indicates that the breakpoint will match using the “Address Range”equation in section 8.5.3.2 “Conditions for Matching Data Breakpoints”</td><td>R/W</td><td>0</td></tr>
<tr><td>BLM</td><td>7:4</td><td>Byte lane mask for value compare on data breakpoint. <i>BLM[0]</i> masks byte at bits [7:0] of the data bus, <i>BLM[1]</i> masks byte at bits [15:8], etc.:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Compare corresponding byte lane</td></tr><tr><td>1</td><td>Mask corresponding byte lane</td></tr></table></td><td>R/W</td><td>Undefined</td></tr>
<tr><td>Res</td><td>3</td><td>Must be written as zero; returns zero on reads.</td><td>R</td><td>0</td></tr>
<tr><td>TE</td><td>2</td><td>Use data breakpoint n as triggerpoint:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Don’t use it as triggerpoint</td></tr><tr><td>1</td><td>Use it as triggerpoint</td></tr></table></td><td>R/W</td><td>0</td></tr>
<tr><td>IVM</td><td>1</td><td>Invert Value Match. When set, the data value compare will be inverted. i.e., a break or trigger will be taken if the value does not match the specified value</td><td>R/W</td><td>0</td></tr>
<tr><td>BE</td><td>0</td><td>Use data breakpoint n as breakpoint:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Don’t use it as breakpoint</td></tr><tr><td>1</td><td>Use it as breakpoint</td></tr></table></td><td>R/W</td><td>0</td></tr>
</table>

8.5.7.6 Data Breakpoint Value n (DBVn) Register (0x2120 + 0x100 * n)

The Data Breakpoint Value n (DBVn) register has the value used in the condition for data breakpoint n. This register is required only if data breakpoints are implemented.

Figure 8.20 DBVn Register Format

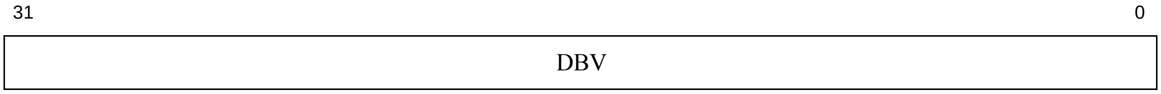


Table 8.20 DBVn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
DBV	31:0	Data breakpoint value for condition.	R/W	Undefined

8.5.7.7 Data Breakpoint Complex Control n (DBCCn) Register (0x2128 + n*0x100)

The Data Breakpoint Complex Control n (DBCCn) register controls the complex break conditions for data breakpoint n. This register is required only if complex breakpoints are implemented and only for implemented data breakpoints.

Figure 8.21 DBCCn Register Format

31 20	19 16	15	14	13 10	9	8 5	4	3 2 1 0
Res	TIBrkNum	TUP	Res	PrCnd	CBE	DBrkNum	Q	Res

Table 8.21 DBCCn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
Res	31:20, 14, 3:0	Must be written as zero; returns zero on read.	R	0
TIBrkNum	19:16	Tuple Instruction Break Number. Indicates which instruction breakpoint will be paired with this data breakpoint to form a tuple breakpoint.	R	6I/2D Complex Breakpoint Configuration: DBCC0 - 0 DBCC1 - 3 8I/4D Complex Breakpoint Configuration: DBCC0 - 0 DBCC1 - 2 DBCC2 - 4 DBCC3 - 6
TUP	15	Tuple Enable. Qualify this data breakpoint with a match on the TIBrkNum instruction breakpoint on the same instruction.	R/W	0
PrCnd	13:12	Upper bits of priming condition for D breakpoint n. M6200 only supports 4 priming conditions so the upper 2 bits are read only as 0.	R	0
PrCnd	11:10	Priming condition for D Breakpoint n. 00 - Bypass, no priming needed Other - Varies depending on the break number, refer to Table 8.24 for mapping.	R/W	0
CBE	9	Complex Break Enable - enables this breakpoint for use as a priming or qualifying condition for another breakpoint.	R/W	0
DQBrkNum	8:5	Indicates which data breakpoint channel is used to qualify this data breakpoint. Data qualification of data breakpoints is not supported on the M6200 core and this field will read as 0 and cannot be written.	R	0
DQ	4	Qualify this breakpoint based on the data breakpoint indicated in *DBrkNum*. Data qualification of data breakpoints is not supported on the M6200 core and this field will read as 0 and cannot be written.	R	0

8.5.7.8 Data Breakpoint Pass Counter n (DBPCn) Register (0x2130 + n*0x100)

The *Data Breakpoint Pass Counter n* (*DBPCn*) register controls the pass counter associated with data breakpoint n. This register is required only if complex breakpoints are implemented and only for implemented data breakpoints.

If complex breakpoints are implemented, there will be an 16b pass counter for each of the data breakpoints on the M6200 core.

Figure 8.22 DBPCn Register Format

31 16	15 0
0	PassCnt

Table 8.22 DBPCn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:16	Ignored on write, returns zero on read.	R	0
PassCnt	15:0	Prevents a break/trigger action until the matching conditions on data breakpoint n have been seen this number of times. Each time the matching condition is seen, this value will be decremented by 1. When the value reaches 0, subsequent matches will cause a break or trigger as requested and the counter will stay at 0. The break or trigger action is imprecise if the *PassCnt* register was last written to a non-zero value. It will remain imprecise until this register is written to 0 by software.	R/W	0

8.5.7.9 Data Value Match (DVM) Register (0x2ffo)

The Data Value Match (DVM) register captures the data value of a load that takes a precise data value breakpoint. This allows debug software to synthesize the load instruction without re-executing it in case it is to a system register that has destructive reads. This register is required only if data breakpoints are implemented.

Figure 8.23 DVM Register Format

31 0
LDV

Table 8.23 DVM Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
LDV	31:0	Load data value for the last precise load data value breakpoint taken.	R	Undefined

8.5.8 Complex Breakpoint Registers

The registers for complex breakpoints are described Table 8.24. These registers have implementation information and are used to setup the data breakpoints. All registers are in drseg.

Table 8.24 Addresses for Complex Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x1120 + 0x100 * n	*IBCCn*	Instruction Breakpoint Complex Control n - described above with instruction breakpoint registers
0x1128 + 0x100 * n	*IBPCn*	Instruction Breakpoint Pass Counter n - described abovewith instruction breakpoint registers
0x2128 + 0x100 * n	*DBCCn*	Data Breakpoint Complex Control n - described above with data breakpoint registers
0x2130 + 0x100 * n	*DBPCn*	Data Breakpoint Pass Counter n - described above with data breakpoint registers
0x8000	*CBTControl*	Complex Break and Triggerpoint Control - indicates which of the complex breakpoint features are implemented
0x8300 + 0x20 * n	*PrCndAIn*	Prime Condition Register A for Instruction breakpoint n
0x84e0 + 0x20 * n	*PrCndADn*	Prime Condition Register A for Data breakpoint n
0x8900	*STCtl*	Stopwatch Timer Control
0x8908	*STCnt*	Stopwatch Timer Count
n is breakpoint number from 0 to 7 (range dependent on implemented hardware)		

8.5.8.1 Complex Break and Trigger Control (CBTC) Register (0x8000)

The *CBTC* register contains configuration bits that indicate which features of complex break are implemented as well as a control bit for the stopwatch timer. On the M6200 core, if complex break is implemented, all of the separate features will be present. This register is required only if complex breakpoints are implemented.

Figure 8.24 CBTC Register Format

31 9	8	7 5	4	3	2	1	0
Res	STMode	Res	STP	PP	DQP	TP	PCP

Table 8.25 CBTC Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
Res	31:9, 7:5	Reserved	R	0
STMode	8	Stopwatch Timer Mode: controls whether the stopwatch timer is free-running or controlled by triggerpoints: 0 - free-running 1 - started and stopped by instruction triggers	R/W	1

Table 8.25 CBTC Register Field Descriptions (Continued)

Fields				
Name	Bits	Description	Read/Write	Reset State
STP	4	Stopwatch Timer Present - indicates whether stopwatch timer is implemented.	R	1
PP	3	Priming Present - indicates whether primed breakpoints are supported	R	1
DQP	2	Data Qualify Present - indicates whether data qualified breakpoints are supported.	R	1
TP	1	Tuple Present - indicates whether any tuple breakpoints are implemented.	R	1
PCP	0	Pass Counters Present - indicates whether any breakpoints have pass counters associated with them.	R	1

8.5.8.2 Priming Condition A (PrCndAI/Dn) Registers

The Prime Condition registers hold implementation specific information about which triggerpoints are used for the priming conditions for each breakpoint register. On the M6200 core, these connections are predetermined and these registers are read-only. This register is required only if complex breakpoints are implemented.

The architecture allows for up to 16 priming conditions to be specified and there can be up to 4 priming condition registers per breakpoint (A/B/C/D). The M6200 core only allows for 4 priming conditions and thus only implements the PrCndA registers. The general description is shown in Table 8.26. The actual priming conditions for each of the breakpoints are shown in Table 8.27.

Figure 8.25 PrCndA Register Format

31 24	23 16	15 8	7 0
Cond3	Cond2	Cond1	Cond0

Table 8.26 PrCndA Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
CondN	31:24 23:16 15:8 7:0	Specifies which triggerpoint is connected to priming condition 3, 2, 1, or 0[a] for the current breakpoint.	R	Preset
	31:30 23:22 15:14 7:6	Reserved	R	0
	29:28 21:20 13:12 5:4	Trigger type 00 - Special/Bypass 01 - Instruction 10 - Data 11 - Reserved	R	Preset
	27:24 19:16 11:8 3:0	Break Number, 0-14	R	Preset

[a] Condition 0 is always Bypass and will read as 8 b0

Table 8.27 Priming Conditions and Register Values for 6I/2D Configuration

Break	Cond0	Cond1	Cond2	Cond3	PrCndA Value	drseg offset
Inst0	Bypass	Data0	Inst1	Inst2	0x1211_2000	0x8300
Inst1	Bypass	Data0	Inst0	Inst2	0x1210_2000	0x8320
Inst2	Bypass	Data0	Inst0	Inst1	0x1110_2000	0x8340
Inst3	Bypass	Data1	Inst4	Inst5	0x1514_2100	0x8360
Inst4	Bypass	Data1	Inst3	Inst5	0x1513_2100	0x8380
Inst5	Bypass	Data1	Inst3	Inst4	0x1413_2100	0x83a0
Data0	Bypass	Inst0	Inst1	Inst2	0x1211_1000	0x84e0
Data1	Bypass	Inst3	Inst4	Inst5	0x1514_1300	0x8500

Table 8.28 Priming Conditions and Register Values for 8I/4D Configuration

Break	Cond0	Cond1	Cond2	Cond3	PrCndA Value	drseg offset
Inst0	Bypass	Data0	Inst1	Inst2	0x1211_2000	0x8300
Inst1	Bypass	Data0	Inst0	Inst2	0x1210_2000	0x8320
Inst2	Bypass	Data1	Inst3	Inst4	0x1413_2100	0x8340
Inst3	Bypass	Data1	Inst2	Inst4	0x1412_2100	0x8360
Inst4	Bypass	Data2	Inst5	Inst6	0x1615_2200	0x8380

Break	Cond0	Cond1	Cond2	Cond3	PrCndA Value	drseg offset
Inst5	Bypass	Data2	Inst4	Inst6	0x1614_2200	0x83a0
Inst6	Bypass	Data3	Inst7	Inst0	0x1017_2300	0x83c0
Inst7	Bypass	Data3	Inst6	Inst0	0x1016_2300	0x83e0
Data0	Bypass	Inst0	Inst1	Data1	0x2111_1000	0x84e0
Data1	Bypass	Inst2	Inst3	Data2	0x2213_1200	0x8500
Data2	Bypass	Inst4	Inst5	Data3	0x2315_1400	0x8520
Data3	Bypass	Inst6	Inst7	Data0	0x2017_1600	0x8540

8.5.8.3 Stopwatch Timer Control (STCtl) Register (0x8900)

The *Stopwatch Timer Control* (*STCtl*) register gives configuration information about how the stopwatch timer register is controlled. On the M6200 core, the break channels that control the stopwatch timer are fixed and this register is read-only. This register is required only if stopwatch timer is implemented.

Figure 8.26 STCtl Register Format

31 18	17 14	13 10	9	8 5	4 1	0
Res	StopChan1	StartChan1	En1	StopChan0	StartChan0	En0

Table 8.29 STCtl Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Res	31:18	Must be written as zero; returns zero on read.	R	0
StopChan1	17:14	Indicates the instruction breakpoint channel that will stop the counter if the timer is under pair1 breakpoint control	R	0
StartChan1	13:10	Indicates the instruction breakpoint channel that will start the counter if the timer is under pair1 breakpoint control	R	0
En1	9	Enables the second pair (pair1) of breakpoint registers to control the timer when under breakpoint control. If the stopwatch timer is configured to be under breakpoint control (by setting *CBTControlSTM*)and this bit is set, the breakpoints indicated in the StartChan1 and StopChan1 fields will control the timer. The M6200 core only supports 1 pair of stopwatch control breakpoints so this field is not writable and will read as 0.	R	0
StopChan0	8:5	Indicates the instruction breakpoint channel that will stop the counter if the timer is under pair0 breakpoint control.	R	0x4
StartChan0	4:1	Indicates the instruction breakpoint channel that will start the counter if the timer is under pair0 breakpoint control.	R	0x1

Table 8.29 STCtl Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
En0	0	Enables the first pair (pair0) of breakpoint registers to control the timer when under breakpoint control. If the stopwatch timer is configured to be under breakpoint control (by setting *CBTControlSTM*)and this bit is set, the breakpoints indicated in the StartChan0 and StopChan0 fields will control the timer. The M6200 core only supports 1 pair of stopwatch control breakpoints so this field is not writable and will read as 1.	R	1

8.5.8.4 Stopwatch Timer Count (STCnt) Register (0x8908)

The Stopwatch Timer Count (STCnt) register is the count value for the stopwatch timer. This register is required only if the stopwatch timer is implemented.

Figure 8.27 STCnt Register Format

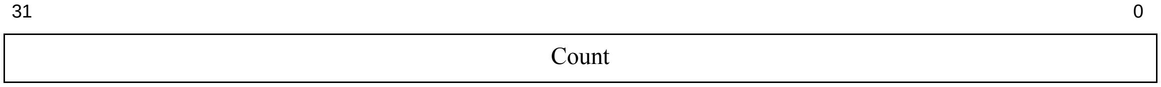


Table 8.30 STCtl Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Count	31:0	Current counter value	R/W	0

8.6 Complex Breakpoint Usage

8.6.1 Checking for Presence of Complex Break Support

Software should verify that the complex breakpoint hardware is implemented prior to attempting to use it. The full sequence of steps is shown below for general use. Spots where the M6200 core has restricted behavior are noted.

1. Read the *Config1EP* bit to check for the presence of Debug logic. Debug logic is always present on the M6200 core.

2. Read the Debug_{NoDCR} bit to check for the presence of the Debug Control Register (DCR). The DCR is always be implemented on the M6200 core.

3. Read the DCR_{CBT} bit to check for the presence of any complex break and trigger features.

4. Read the CBT Control register to check for the presence of each individual feature. If the M6200 core implements any complex break and trigger features, it will implement all of them.

5. If Pass Counters are implemented, they may not be implemented for all break channels and may have different counter sizes. To determine the size and presence of each pass counter, software can write a value of -1 toeach of the IBPCn and DBPCn registers and read back the values to note the bits that were set by the write. If the M6200 core implements pass counters, it will implement an 8-bit counter for each instruction breakpoint and a 16-bit counter for each data breakpoint.

6. If tuples are implemented, they may only be supported on a subset of the data breakpoint channels. This can be checked by seeing if the $DBCCn_{TUP}$ bit can be set to 1. Additionally, some cores may support dynamically changing which instruction breakpoint is associated with a given data breakpoint. This can be checked by attempting to write the $DBCCn_{TIBrkNum}$ field. If the M6200 core implements tuple support, it will support it for all data breakpoint channels and the instruction breakpoint association will be fixed.

7. If Priming Conditions are supported, a core may only support a subset of the possible priming condition values. This can be checked by 4'hf to the $xBCCn_{PrCnd}$ field. If only 1 or 2 bits can be written, the available priming conditions will be described in the PrCndA registers. If 3 bits are writable, PrCndA and PrCndB will describe the conditions, and if all 4 bits are writable, the PrCndA, PrCndB, PrCndC, and PrCndD registers will all exist. Some cores may also support changing the priming conditions and this can be checked by attempting to write to the PrCnd registers. If the M6200 core supports priming conditions, it will support 4 statically mapped priming conditions per breakpoint which will be described in the PrCndA registers.

8. If support for qualified breakpoints is indicated, it may only be supported for some of the breakpoints. Additionally, the data breakpoint used for the qualification may be configurable. Software can check this by writing to the $xBCCn_{PrCnd}$ fields. If the M6200 core support qualified breakpoints, it will only support it on instruction breakpoints and the data break used for qualification will be fixed for each instruction breakpoint.

9. If the stopwatch timer is implemented, either one or two pairs of instruction breakpoints may be available for controlling it and it may be possible to dynamically select which instruction breakpoints are used. This can be tested by writing to the *STCtl* register.

8.6.2 General Complex Break Behavior

There is some general complex break behavior that is common to all complex breakpoints. This behavior is described below:

- Resets to a disabled state - when the core is reset, the complex break functionality will be disabled and debug software that is not aware of complex break should continue to function normally.

- Complex break state is not updated on exceptional instructions

- Complex breakpoints are evaluated at the end of the pipeline and complex breakpoint exceptions are taken imprecisely on the following instruction.

- There is no hazard between enabling and enabled events. When an instruction causes an enabling event, the following instruction sees the enabled state and reacts accordingly.

8.6.3 Usage of Pass Counters

Pass counters specify that the breakpoint conditions must match *N* times before the breakpoint action will be enabled.

- Controlled by writing to the per-breakpoint pass counter register
- Resets to 0
- Writing to a non-zero value enables the pass counter. When enabled, each time the breakpoint conditions match, the counter will be decremented by 1. After the counter value reaches 0, the breakpoint action (breakpoint exception, trigger, or complex break enable) will occur on any subsequent matches and the counter will not decrement further. The action does not occur on the match that causes the 1->0 counter decrement.
- If the breakpoint also has priming conditions and/or data qualified specified, the pass counter will only decrement when the priming and/or qualified conditions have been met.
- If a data breakpoint is configured to be a tuple breakpoint, the data pass counter will only decrement on instructions where both the instruction and data break conditions match. The pass counter for the instruction break involved in a tuple should not be enabled if the tuple is enabled.
- When a pass counter has been enabled, it will be treated as enabled until the pass counter is explicitly written to 0. Namely, breakpoint exceptions will continue to be taken imprecisely until the pass counter is disabled by writing to 0.
- The counter register will be updated as matches are detected. The current count value can be read from the register while operating in debug mode. Note that this behavior is architecturally recommended, but not required.

8.6.4 Usage of Tuple Breakpoints

A tuple breakpoint is the logical AND of a data breakpoint and an instruction breakpoint. Tuple breakpoints are specified as a condition on a data breakpoint. If the $\text{DBCCn}_{\text{TUP}}$ bit is set, the data breakpoint will not match unless there the corresponding instruction breakpoint conditions are also met.

- Uses the data breakpoint resources to specify the break action, break status, pass counters, and priming conditions.
- The instruction breakpoint involved in the tuple should be configured as follows:
  - $\text{IBCCn}_{\text{CBE}} = 1$
  - $\text{IBCCn}_{\text{PrCnd}} = \text{IBCCn}_{\text{DQ}} = \text{IBCn}_{\text{TE}} = \text{IBCn}_{\text{BE}} = \text{IBPCn} = 0$

8.6.5 Usage of Priming Conditions

Priming conditions provide a way to have one breakpoint enabled by another one. Prior to the priming condition being satisfied, any breakpoint matches are ignored.

- Priming condition resets to bypass which specifies that no priming is required

- 3 other priming conditions are available for each breakpoint. These condition vary from breakpoint to breakpoint (since it makes no sense for a breakpoint to prime itself). The conditions for each of the breakpoints are listed in Table 8.27.
- The priming breakpoint must have $xBCn_{TE}$ or $xBCCn_{CBE}$ set.
- When the priming condition has been seen, the primed breakpoint will remain primed until its *xBCCn* register is written
- The primed state is stored with the breakpoint being primed and not with the breakpoint that is doing the priming.
- Each Prime condition is the comparator output after it has been qualified by its own Prime condition, data qualification, and pass counter. Using this, several stages of priming are possible (e.g. data cycle D followed by instruction A followed by instruction B N times followed by instruction C).

8.6.6 Usage of Data Qualified Breakpoints

Each of the instruction breakpoints can be set to be data qualified. In qualified mode, a breakpoint will recognize its conditions only after the specified data breakpoint matches both address and data. If the data breakpoint matches address, but has a mismatch on the data value, the instruction breakpoint will be unqualified and will not match until a subsequent qualifying match.

This feature can be used similarly to the ASID qualification that is available on cores with TLBs. If an RTOS loads a process ID for the current process, that load can be used as the qualifying breakpoint. When a matching process ID is loaded (entering the desired RTOS process), qualified instruction breakpoints will be enabled. When a different process ID is loaded (leaving the desired RTOS process), the qualified instruction breakpoints are disabled. Alternatively, with the InvertValueMatch feature of the data breakpoint, the instruction breakpoints could be enabled on any process ID other than the specified one.

- The qualifying data break must have $DBCn_{TE}$ or $DBCCn_{CBE}$ set.
- The qualifying data break should have data comparison enabled (via settings of $DBCn_{BLM}$ and $DBCn_{BAI}$)
- The qualifying data break should not have pass counters, priming conditions, or tuples enabled.
- The qualifying data access can be either a load or store, depending on the settings of $DBCn_{NoSB}$ and $DBCn_{NoLB}$
- The Qualified/Unqualified state is stored with the instruction breakpoint that is being qualified. Writing its *IBCCn* register will disqualify that breakpoint.
- Qualified instruction breakpoint can also have priming conditions and/or pass counters enabled. The pass counter will only decrement when the priming and qualifying conditions have been met. The instruction breakpoint action (break, trigger, or complex enable) will only occur when all priming, qualifying, and pass counter conditions have been met.
- Qualified instruction breakpoint can be used to prime another breakpoint

8.6.7 Usage of Stopwatch Timers

The stopwatch timer is a drseg memory mapped count register. It can be configured to be free running or controlled by instruction breakpoints. This could be used to measure the amount of time that is spent in a particular function by starting the counter upon function entry and stopping it upon exit.

- Count value is reset to 0
- Reset state has counter stopped and under breakpoint control so that the counter is not running when the core is not being debugged.
- Bit in CBT Control register controls whether the counter is free-running or breakpoint controlled.
- Counter does not count in debug mode
- When breakpoint controlled, the involved instruction breakpoints must have $IBCn_{TE}$ or $IBCCn_{CBE}$ set in order to start or stop the timer.
- It can be asserted by a Debug probe.

8.7 Secure Debug

When *EJ_DisableProbeDebug* is asserted, the following conditions exist:

- IMPCODE, IDCODE: operate as usual
- OCI CONTROL Register (OCR): BOOTMODE fixed at NORMAL, DisableProbeDebug=1, ProbTrap=0, ProbEn=0, PerRst=0, PrRst=0, DbgBrk =0 (except if enabled by the Override bit (DCR[DbgBrk_Override])
- DATA: Since ProbEn=0, the processor will never read/write DMSEG, so any read/write of DATA from the probe will hang the APB.
- DRSEG_DATA: Disabled; a read or write from DRSEG_DATA will never be satisfied and will hang the APB.
- FDC: Operates normally.

When *EJ_DisablePCSamDebug* is asserted, the following conditions exist:

- PCSAMPLE: Disabled. PCSAMPLE1/2 read value is unpredictable as long as NEW is fixed at 0.

8.8 Performance Counters

Performance counters are used to accumulate occurrences of internal predefined events/cycles/conditions for program analysis, debug, or profiling. A few examples of event types are clock cycles, instructions executed, specific instruction types executed, loads, stores, exceptions, and cycles while the CPU is stalled. There are two, 32-bit counters. Each can count one of the 64 internal predefined events, or one of two externally controlled events selected by a corresponding control register. A counter overflow can be programmed to generate an interrupt, where the interrupt handler software can maintain larger total counts.

8.9 iFlowtrace™

The M6200 core has an option for a simple trace mechanism called iFlowtrace. iFlowtrace is a light-weight instruction-only tracing scheme that is sufficient to reconstruct the execution flow in the core, and it can only be controlled by debug software. This tracing scheme has been kept very simple to minimize the impact on die size. iFlowtrace memory can be configured as both on-chip and off-chip.

iFlowtrace also offers special-event trace modes when normal tracing is disabled, namely:

- Function Call/Return and Exception Tracing mode to trace the PC value of function calls and returns and/or exceptions and returns.
- Breakpoint Match mode traces the breakpoint ID of a matching breakpoint and, for data breakpoints, the PC value of the instruction that caused it.
- Filtered Data Tracing mode traces the ID of a matching data breakpoint, the load or store data value, access type and memory access size, and the low-order address bits of the memory access, which is useful when the data breakpoint is set up to match a range of addresses.
- User Trace Messages. The user can instrument their code to add their own 32-bit value messages into the trace by writing to the two Cop0 UTM registers.
- Delta Cycle mode works in combination with the above trace modes to provide a timestamp between stored events. It reports the number of cycles that have elapsed since the last message was generated and put into the trace.

Two new iFlowtrace features:

IFCTL2[UTM_En] (bit 9). This enables (1) UTMs to be output to the trace port or disabled (0).

IFCTL2[DeltaCycle_Divide] (bits 11:10). This sets a divider rate between the CPU clock and delta cycle counter clock tick; the counter value is what is recorded in the trace when DeltaCycle mode is enabled.

Tracing is disabled if the processor enters Debug Mode. This is true for both Normal Trace Mode as well as Special Trace Mode.

The presence of the iFlowtrace mechanism is indicated by the CP0 *Config3*$_{ITL}$ register bit.

For more information, refer to the MIPS® iFlowtrace™ Architecture Specification [7].

8.9.1 A Simple Instruction-Only Tracing Scheme

A trace methodology can often be mostly defined by its inputs and outputs. Hence this basic scheme is described by the inputs to the core tracing logic and by the trace output format from the core. We assume here that the execution flow of the program is traced at the end of the execution path in the core similar to PDtrace.

8.9.1.1 Trace Inputs

1. *In_TraceOn*: When on, legal trace words are coming from the core and at the point when it is turned on, that is for the first traced instruction, a full PC value is output. When off, it cannot be assumed that legal trace words are available at the core interface.

2. *In_Stall*: This says, stall the processor to avoid buffer overflow that can lose trace information. When off, a buffer overflow will simply throw away trace data and start over again. When on, the processor is signalled from the tracing logic to stall until the buffer is sufficiently drained and then the pipeline is restarted.

8.9.1.2 Normal Trace Mode Outputs

1. Stall cycles in the pipe are ignored by the tracing logic and are not traced. This is indicated by the signal *Out_Valid* that is turned off when no valid instruction is being traced. When *Out_Valid* is asserted, instructions are traced out as described in the rest of this section. The traced instruction PC is a virtual address.

2. In the output format, every sequentially executed instruction is traced as 1’b0.

3. Every instruction that is not sequential to the previous one is traced as either a 10 or an 11 (read this as a serial bitstream from left to right). This implies that the target instruction of a branch or jump is traced this way, not the actual branch or jump instruction (this is similar to PDtrace):

4. A 10 instruction implies a taken branch for a conditional branch instruction whose condition is unpredictable statically, but whose branch target can be computed statically and hence the new PC does not need to be traced out. Note that if this branch was not taken, it would have been indicated by a 0 bit, that is sequential flow.

5. A 11 instruction implies a taken branch for an indirect jump-like instruction whose branch target could not be computed statically and hence the taken branch address is now given in the trace. This includes, for example, instructions like jr, jalr, and interrupts:

   - 11 00 - followed by 8 bits of 1-bit shifted offset from the last PC. The bit assignments of this format on the bus between the core tracing logic and the ITCB is:

     [3:0] = 4’b0011
     [11:4] = PCdelta[8:1]

   - 11 01 - followed by 16 bits of 1-bit shifted offset from the last PC. The bit assignments of this format on the bus between the core tracing logic and the ITCB is:

     [3:0] = 4’b1011
     [19:4] = PCdelta[16:1]

   - 11 10 - followed by 31 of the most significant bits of the PC value, followed by a bit (NCC) that indicates no code compression. Note that for a MIPS32 or MIPS64 instruction, NCC=1, and for microMIPS instruction NCC=0. This trace record will appear at all transition points between MIPS32/MIPS64 and microMIPS instruction execution.
     This form is also a special case of the 11 format and it is used when the instruction is not a branch or jump, but nevertheless the full PC value needs to be reconstructed. This is used for synchronization purposes, similar to the Sync in PDtrace. In iFlowtrace rev 2.0 onwards, the sync period is user-defined, and is counted down and when an internal counter runs through all the values, this format is used. The bit assignments of this format on the bus between the core tracing logic and the ITCB is:

     [3:0] = 4’b0111
     [34:4] = PC[31:1]
     [35] = NCC

   - 11 11 - Used to indicate trace resumption after a discontinuity occurred. The next format is a 1110 that sends a full PC value. A discontinuity might happen due to various reasons, for example, an internal buffer overflow, and at trace-on/trace-off trigger action.

8.9.2 Special Trace Modes

iFlowtrace 2.0 adds special trace modes which can only be active when the normal tracing mode is disabled. Software can determine which modes are supported by attempting to write the enable bits in the IFCTL register. Software can check the Illegal bit in the IFCTL register—if an unsupported combination of modes is requested, the bit will be set and the trace contents will be unpredictable. The special trace modes are described below.

8.9.2.1 Mode Descriptions

Delta Cycle Mode

This mode is specified in combination with the other special trace modes. It is enabled via the CYC bit in the Control/Status Register. When delta cycle reporting is enabled, each trace message will include a 10b delta cycle value which reports the number of cycles that have elapsed since the last message was generated. A value of 0 indicates that the two messages were generated in the same cycle. A value of 1 indicates that they were generated in consecutive cycles. If 1023 cycles elapse without an event being traced, a counter rollover message is generated.

Note: If the processor clocks stop due to execution of the WAIT instruction, the delta cycle counter will also stop and will report ‘active’ cycles between events rather than ‘total’ cycles.

Breakpoint Match Mode

This modes uses Debug data and instruction breakpoint hardware to enable a trace of PC values. Instead of starting or stopping trace, a triggerpoint will cause a single breakpoint match trace record. This record indicates that there was a triggerpoint match, the breakpoint ID of the matching breakpoint, and the PC value of an instruction that matched the instruction of data breakpoint.This mode can only be used when normal tracing mode is turned off. This mode can not be used in conjunction with other special trace modes. This mode is enabled or disabled via the BM field in the Control/Status register (see Section 8.9.6 “ITCB Register Interface for Software Configurability”).

The breakpoints used in this mode must have the TE bet set to enable the match condition.

Software should avoid setting up overlapping breakpoints. The behavior when multiple matches occur on the same instruction is to report a Breakpoint ID of 7.

Filtered Data Tracing Mode

This mode uses Debug data breakpoint hardware to enable a trace of data values. Rather than starting or stopping trace as in normal trace mode, a data triggerpoint will cause a filtered data trace record. This record indicates that there was a data triggerpoint match, the breakpoint ID of the matching breakpoint, whether it was a load or store, the size of the request, low order address bits, and the data value. This mode can only be used when normal tracing mode is turned off. This mode can not be used in conjunction with other special trace modes. This mode can be enabled or disabled via the FDT bit in the Control/Status register (see Section 8.9.6 “ITCB Register Interface for Software Configurability”).

The corresponding data breakpoint must have the TE bit set to enable the match condition.

Software should avoid setting up overlapping data breakpoints. The behavior when multiple matches on one load or store are detected is to report a Breakpoint ID of 7.

Extended Filtered Data Tracing Mode

Extends Filtered Data Tracing Mode by adding the virtual address of the load/store instruction to the generated trace information. (see Section “Filtered Data Tracing Mode” above).

This behavior is enabled/disabled by the FDT_CAUSE field in the IFCTL Control/Status register (see Section 8.9.6 “ITCB Register Interface for Software Configurability”). FDT_CAUSE only has effect if the FDT field is also set.

The extended trace sequence is a FDT trace message followed by the Breakpoint Match (BM) trace message. If the $IFCTL_{\text{CYC}}$ field is set, the FDTtrace message will have a DeltaCycle Message value of ‘0’ directly followed by the

Breakpoint Match message. This message sequence (FDT, delta cycle of 0, and BM) indicates to the trace disassembler that Extended Filtered Data Tracing mode is enabled *(*IFCTL$_{FDT\_CAUSE}$=1).

Function Call/Return and Exception Tracing Mode

In this mode, the PC value of function calls and returns and/or exceptions and returns are traced out. This mode can only be used when normal tracing mode is turned off. This mode cannot be used in conjunction with other special trace modes. The function call/return and exception/return are independently enabled or disabled via the FCR and ER bits in the Control//Status register (see Section 8.9.6 "ITCB Register Interface for Software Configurability").

These events are reported for the following instructions:

- MIPS32 function calls: JAL, JALR, JALR.HB, JALX
- microMIPS function calls: JAL, JALR, JALR.HB, JALX, JALR16, JALRS16, JALRS, JALRS.HB, JALS
- MIPS32 function returns: JR, JR.HB
- microMIPS function returns: JR, JR.HB, JRC, JRADDIUSP, JR16
- Exceptions: Reported on the first instruction of the exception handler
- Exception returns: ERET
- MCU ASE Interrupt returns: IRET

Other Trace Messages

In any of the special trace modes, it is possible to embed messages into the trace stream directly from a program. This is done by writing to the *UserTraceData1* or *UseTraceData2* COP0 registers. When *UserTraceData1* register is written, a trace message of type "User Triggered Message 1" (UTM1) is generated. When *UserTraceData2* register is written, a trace message of type "User Triggered Message 2" (UTM2) is generated. Please refer to 5.2.54 "User Trace Data1 Register (CP0 Register 23, Select 3)/User Trace Data2 Register (CP0 Register 24, Select 3)" on page 225.

Overflow messages can also be generated when tracing off-chip if the IO control bit is 0 and trace data is generated faster than it is consumed. No overflow will be generated when using on-chip trace.

8.9.2.2 Special Trace Mode Outputs

The normal and special trace modes cannot be enabled at the same time because the trace message encoding is not unique between the two modes. The software reading the trace stream must be aware of which mode is selected to know how to interpret the bits in the trace stream. The message types for each type of special trace message are unique.

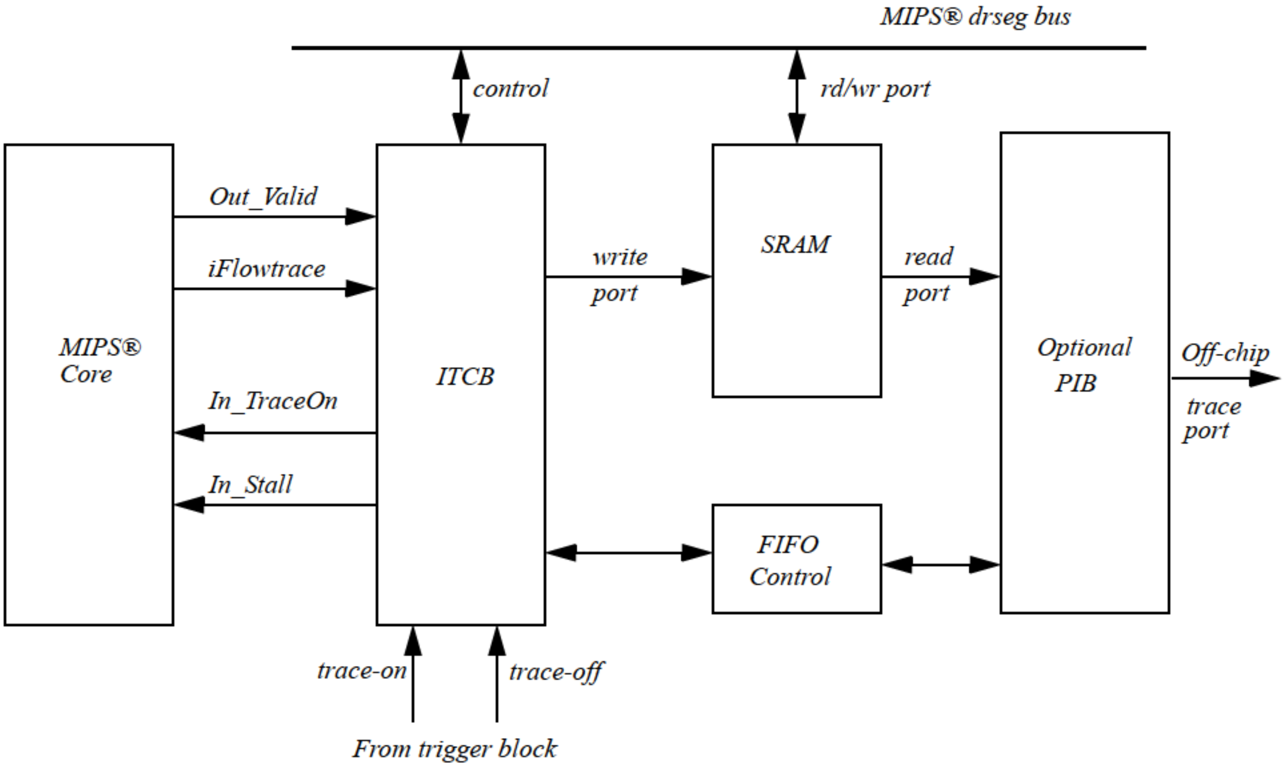
- 00 (as above, read a bitstream from left to right) - Delta Cycle Rollover message. The output format is:
  [1:0] = 2'b00
- 010 - User Trace Message. The format of this type of message is:
  [2:0] = 3'b010
  [34:3] = Data[31:0]
  [35] = UTM2/UTM1 (1=UTM2, 0=UTM1)
  [44:36] = DeltaCycle (if enabled)

- 011 - Reserved
- 10 - Breakpoint Match Message. The output format during this trace mode is:
  [1:0] = 2’b01
  [5:2] = BreakpointID
  [6] = Instruction Breakpoint
  [37:7] = MatchingPC[31:1]
  [38] = NCC
  [48:39] = DeltaCycle (if enabled)
  Note that for a MIPS32 or MIPS64 instruction, NCC=1, and for microMIPS instruction NCC=0.
- 110 - Filtered Data Message. The output format during this trace mode is:
  [2:0] = 3’b011
  [6:3] = BreakpointID
  [7] = Load/Store (1=Load, 0=Store)
  [8] = FullWord (1=32b data, 0= <32b)
  [14:5] = Addr[7:2]
  [46:15] = {32b data value} OR
  [46:15] = {BE[3:0], 4’b0, 24b data value} OR
  [46:15] = {BE[3:0], 12’b0, 16b data value} OR
  [46:15] = {BE[3:0],20’b0, 8b data value}
  [56:47] = DeltaCycle (if enabled)
- 1110 - Function Call/Return/Exception Tracing. The output format during this trace mode is:
  [3:0] = 4’b0111
  [4] = FC
  [5] = Ex
  [6] = R
  [37:8] = PC[31:1]
  [38] = NCC
  [48:39] = Delta Cycle (if enabled)
  Note that for a MIPS32 or MIPS64 instruction, NCC=1, and for microMIPS instruction NCC=0. FC=1 implies a function call, Ex=1 implies the start of an exception handler, and R=1 implies a function or exception return.
- 1111- Overflow message. The format of this type of message is:
  [3:0] = 4’b1111

8.9.3 ITCB Overview

The iFlowtrace Control Block (ITCB) is responsible for accepting trace signals from the CPU core, formatting them, and storing them into an on-chip FIFO. The figure also shows the Probe Interface Block (PIB) which reads the FIFO and outputs the memory contents through a narrow off-chip trace port.

Figure 8.28 Trace Logic Overview



8.9.4 ITCB iFlowtrace Interface

The iFlowtrace interface consists of 57 data signals plus a valid signal. The 57 data signals encode information about what the CPU is doing in each clock cycle. Valid indicates that the CPU is executing an instruction in this cycle and therefore the 57 data signals carry valid execution information. The iFlowtrace data bus is encoded as shown in Table 8.31. Note that all the non-defined upper bits of the bus are zeroes.

Table 8.31 Data Bus Encoding

Valid	Data (LSBs)	Description
0	X	No instructions executed in this cycle
1	0	Normal Mode: Sequential instruction executed
1	01	Normal Mode: Branch executed, destination predictable from code
1	<8>0011	Normal Mode: Discontinuous instruction executed, PC offset is 8 bit signed offset
1	<16>1011	Normal Mode: Discontinuous instruction executed, PC offset is 16 bit signed offset
1	<NCC><31>0111	Normal Mode: Discontinuous instruction or synchronization record, No Code Compression (NCC) bit included as well as 31 MSBs of the PC value
1	00	Special Mode: Delta Cycle Rollover message
1	<10><32>010	Special Mode: User add-in Trace Message. 32 bit user data as well as 10 bit delta cycle if enabled.

Valid	Data (LSBs)	Description
1	<10><NCC><31><1><4>01	Special Mode: Breakpoint Match Message. 4-bit breakpoint ID, 1 bit indicates breakpoint type, 31 MSBs of the PC value, NCC bit included as well as 10-bit delta cycle if enable.
1	<10><32><6><1><1><4>011	Special Mode: Filtered Data Message. 4 bit breakpoint ID, 1 bit load or store indication, 1 bit full word indication, 6 bit of addr[7:2], 32 bit of the data information included as well as 10 bit delta cycle if enabled.
1	<10><NCC><31><R><Ex><FC>011	Special Mode: Function Call/Return/Exception Tracing. 1 bit function call indication, 1 bit exception indication, 1 bit function or exception return indication, 31 MSBs of the PC value, NCC bit included as well as 10 bit delta cycle if enabled.
1	1111	Internal overflow

8.9.5 TCB Storage Representation

Records from iFlowtrace are inserted into a memory stream exactly as they appear in the iFlowtrace data output. Records are concatenated into a continuous stream starting at the LSB. When a trace word is filled, it is written to memory along with some tag bits. Each record consists of a 64-bit word, which comprises 58 message bits and 6 tag bits or header bits that clarify information about the message in that word.

The ITCB includes a 58-bit shift register to accumulate trace messages. When 58 or more bits are accumulated, the 58 bits and 6 tag bits are sent to the memory write interface. Messages may span a trace word boundary; in this case, the 6 tag bits indicate the bit number of the first full trace message in the 58-bit data field.

The tag bits are slightly encoded so they can serve a secondary purpose of indicating to off-chip trace hardware when a valid trace word transmission begins. The encoding ensures that at least one of the 4 LSBs of the tag is always a 1 for a valid trace message. The tag values are shown in Table 8.32. The longest trace message is 57 bits (filtered data trace in special trace mode with delta cycle), so the starting position indicated by the tag bits is always between 0 and 56.

Table 8.32 Tag Bit Encoding

Starting Bit of First Full Trace Message	Encoding (decimal)
0	58
16	59
32	60
48	61
Unused	0,16,32,48
Reserved	62,63
Others	StartingBit

When trace stops (ON set to zero), any partially filled trace words are written to memory. Any unused space above the final message is filled with 1’s. The decoder distinguishes 1111 patterns used for fill in this position from an 1111 overflow message by recognizing that it is the last trace word.

These trace formats are written to a trace memory that is either on-chip or off-chip. No particular size of SRAM is specified; the size is user selectable based on the application needs and area trade-offs. Each trace word can typically store about 20 to 30 instructions in normal trace mode, so a 1 KWord trace memory could store the history of 20K to 30K executed instructions.

The on-chip SRAM or trace memory is written continuously as a circular buffer. It is accessible via drseg address mapped registers. There are registers for the read pointer, write pointer, and trace word. The write pointer register includes a wrap bit that indicates that the pointer has wrapped since the last time the register was written. Before starting trace, the write pointer would typically be set to 0. To read the trace memory, the read pointer should be set to 0 if there has not been a wrap, or to the value of the write pointer if there has been. Reading the trace word register will read the entry pointed to by the read pointer and will automatically increment the read pointer. Software can continue reading until all valid entries have been read out.

8.9.6 ITCB Register Interface for Software Configurability

The ITCB includes a drseg memory interface to allow software to set up tracing and read the current status. If an on-chip trace buffer is also implemented, there are additional registers included for accessing it.

8.9.6.1 iFlowtrace Control/Status (IFCTL) Register (offset 0x3fc0)

The Control/Status register provides the mechanism for turning on the different trace modes. Figure 8.29 has the format of the register and Table 8.33 describes the register fields.

Figure 8.29 Control/Status Register

31	30 16	15	14	13	12	11	10	9	8 5	4	3	2	1	0
Illegal	0	FDT_CAUSE	CYC	FDT	BM	ER	FCR	EST	SyP	OfClk	OfC	IO	En	On

Table 8.33 Control/Status Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
0	30:16	Reserved for future use. Read as zeros, must be written as zeros	R	0	Required
Illegal	31	This bit is set by hardware and indicates if the currently enabled trace output modes are an illegal combination. A value of 1 indicates an unsupported setting. A value of 0 indicates that the currently selected settings are legal.	R	0	Required
FDT_CAUSE	15	Extended Filtered Data Trace mode (FDT). Adds causing load/store virtual address to Filtered Data Trace. FDT_CAUSE only has effect if FDT is set. The extended trace sequence is a FDT trace message followed by the Breakpoint Match (BM) trace message. If CYC is set, the FDT trace message will have a DeltaCycle Message value of ‘0’ directly followed by the Breakpoint match (BM) message. This message sequence (FDT, delta cycle of 0, and BM) indicates to the trace disassembler that Extended Filtered Data Tracing mode is enabled.	R/W	0	Optional for iFlowtrace rev 2.0+

Table 8.33 Control/Status Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
CYC	14	Delta Cycle Mode: This mode can be set in combination with the EST special trace modes. When set, a delta cycle value is included in each of the trace messages and indicates the number of cycles since the last message was generated. If this tracing mode is not implemented, the field is read-only and read as zero.	R/W	0	Optional for iFlowtrace rev 2.0+
FDT	13	Filtered Data Trace mode. If set, on a data breakpoint match, the data value of the matching breakpoint is traced. Normal tracing is inhibited when this mode is active. If this tracing mode is not implemented, the field is read-only and read as zero.	R/W	0	Optional for iFlowtrace rev 2.0+
BM	12	Breakpoint Match. If set, only instructions that match instruction or data breakpoints are traced. Normal tracing is inhibited when this mode is active. If this tracing mode is not implemented, the field is read-only and read as zero.	R/W	0	Optional for iFlowtrace rev 2.0+
ER	11	Trace exceptions and exception returns. If set, trace includes markers for exceptions and exception returns. Can be used in conjunction with the FCR bit. Inhibits normal tracing. If this tracing mode is not implemented, the field is read-only and read as zero.	R/W	0	Optional for iFlowtrace rev 2.0+
FCR	10	Trace Function Calls and Returns. If set, trace includes markers for function calls and returns. Can be used in conjunction with the ER bit. If this tracing mode is not implemented, the field is read-only and read as zero.	R/W	0	Optional for iFlowtrace rev 2.0+
EST	9	Enable Special Tracing Modes. If set, normal tracing is inhibited, allowing the user to choose one of several special tracing modes. Setting this bit inhibits normal trace mode. If no special tracing modes are implemented, this field is read-only, and read as zero.	R/W	0	Optional for iFlowtrace rev 2.0+
SyP	8:5	Synchronization Period. The synchronization period is set to $2^{(SyP+8)}$ instructions. Thus a value of 0x0 implies 256 instructions, and a value of 0xF implies 8M instructions.	R/W	0	Required for iFlowtrace rev 2.0+
OfClk	4	Controls the Off-chip clock ratio. When the bit is set, this implies 1:2, that is, the trace clock is running at 1/2 the core clock, and when the bit is clear, implies 1:4 ratio, that is, the trace clock is at 1/4 the core clock. Ignored unless OfC is also set.	R/W	0	Required
OfC	3	Off-chip. 1 enables the PIB (if present) to unload the trace memory. 0 disables the PIB and would be used when on-chip storage is desired or if a PIB is not present. This bit is settable only if the design supports both on-chip and off-chip modes. Otherwise is a read-only bit indicating which mode is supported.	R/W or R	Preset	Required
IO	2	Inhibit overflow. If set, the CPU is stalled whenever the trace memory is full. Ignored unless OfC is also set.	R/W	0	Required

Table 8.33 Control/Status Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
En	1	Trace enable. This bit may be set by software or by Trace-on/Trace-off action bits from the Complex Trigger block. Software writes EN with the desired initial state of tracing when the ITCB is first turned on and EN is controlled by hardware thereafter. EN turning on and off does not flush partly filled trace words.	R/W	0	Required
On	0	Software control of trace collection. 0 disables all collection and flushes out any partially filled trace words.	R/W	0	Required

8.9.6.2 ITCBTW Register (offset 0x3F80)

The ITCBTW register is used to read Trace Words from the on-chip trace memory. The TW read is the TW pointed to by the ITCBRDP register. A side effect of reading the ITCBTW register is that the ITCBRDP register increments to the next TW in the on-chip trace memory. If ITCBRDP is at the max size of the on-chip trace memory, the increment wraps back to address zero.

Note that this is a 64b register.On a 32b processor, software must read the upper word (offset 0x3F84) first as the address increment takes place on a read of the lower word (0x3F80).

The format of the ITCBTW register is shown below, and the field is described in Table 8.34.

Figure 8.30 ITCBTW Register Format

63	0
Data	

Table 8.34 ITCBTW Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Names	Bits				
Data	63:0	Trace Word	R	Undefined	Required

8.9.6.3 ITCBRDP Register (Offset 0x3f88)

The ITCBRDP register is the address pointer to on-chip trace memory. It points to the TW read when reading the *ITCBTW* register. This value will be automatically incremented after a read of the ITCBTW register.

The format of the ITCBRDP register is shown below, and the field is described in Table 8.35. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

Figure 8.31 ITCBRDP Register Format

31	n+1	n	0
		Address	

Table 8.35 ITCBRDP Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Names	Bits				
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0	Required
Address	n:0	Byte address of on-chip trace memory word.	R/W	Undefined	Required

8.9.6.4 ITCBWRP Register (Offset 0x3f90)

The ITCBWRP register is the address pointer to on-chip trace memory. It points to the location where the next new TW for on-chip trace will be written. The top bit in the register indicates whether the pointer has wrapped. If it has, then the write pointer will also point to the oldest trace word. and the read pointer can be set to that to read the entire array in order. If it is cleared, then the read pointer can be set to 0 to read up to the write pointer position.

The format of the ITCBWRP register is shown below, and the field is described in Table 8.36. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

Figure 8.32 ITCBWRP Register Format

31	30 n+1	n 0
Wrap	0	Address

Table 8.36 ITCBWRP Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Names	Bits				
Wrap	31	Indicates that the entire array has been written at least once	R/W	Undefined	Required
0	30:(n+1)	Reserved. Must be written zero, reads back zero.	0	0	Required
Address	n:0	Byte address of the next on-chip trace memory word to be written	R/W	Undefined	Required

8.9.7 ITCB iFlowtrace Off-Chip Interface

The off-chip interface consists of a 4-bit data port (*TR_DATA*) and a trace clock (*TR_CLK*). *TR_CLK* can be a DDR clock; that is, both edges are significant. *TR_DATA* and *TR_CLK* follow the same timing and have the same output structure as the PDtrace TCB described in MIPS specifications. The trace clock is synchronous to the system clock but running at a divided frequency. The OfClk bit in the Control/Status register indicates the ratio between the trace clock and the core clock. The Trace clock is always 1/2 of the trace port data rate, hence the “full speed” ITCB outputs data at the CPU core clock rate but the trace clock is half that, hence the 1:2 OfClk value is the full speed, and the 1:4 OfClk ratio is half-speed.

When a 64-bit trace word is ready to transmit, the PIB reads it from the FIFO and begins sending it out on *TR_DATA*. It is sent in 4-bit increments starting at the LSBs. In a valid trace word, the 4 LSBs are never all zero, so a probe listening on the *TR_DATA* port can easily determine when the transmission begins and then count 15 additional cycles to collect the whole 64-bit word. Between valid transmissions, *TR_DATA* Is held at zero and *TR_CLK* continues to run.

TR_CLK runs continuously whenever a probe is connected. An optional signal *TR_PROBE_N* may be pulled high when a probe is not connected and could be used to disable the off-chip trace port. If not present, this signal must be tied low at the Probe Interface Block (PIB) input.

The following encoding is used for the 6 tag bits to tell the PIB receiver that a valid transmission is starting:

```
//   if (srcount == 0), EncodedSrCount = 111010 = 58
//   else if (srcount == 16) EncodedSrCount = 111011 = 59
//   else if (srcount == 32) EncodedSrCount = 111100 = 60
//   else if (srcount == 48) EncodedSrCount = 111101 = 61
//   else EncodedSrCount = srcount
```

8.9.8 Breakpoint-Based Enabling of Tracing

Each hardware breakpoint in the Debug block has a control bit associated with it that enables a trigger signal to be generated on a break match condition. In special trace mode, this trigger can be used to insert an event record into the trace stream. In normal trace mode, this trigger signal can be used to turn trace on or off, thus allowing a user to control the trace on/off functionality using breakpoints. Similar to the TraceIBPC and TraceDBPC registers in PDtrace, registers are defined to control the start and stop of iFlowtrace. The details on the actual register names and drseg addresses are shown in Table 8.37.

Table 8.37 drseg Registers that Enable/Disable Trace from Breakpoint-Based Triggers

Register Name	drseg Address	Reset Value	Description
ITrigiFlowTrcEn	0x3FD0	0	Register that controls whether or not hardware instruction breakpoints can trigger iFlowtrace tracing functionality
DTrigiFlowTrcEn	0x3FD8	0	Register that controls whether or not hardware data and tuple breakpoints can trigger iFlowtrace tracing functionality

The bits in each register are defined as follows:

- Bit 28 (IE/DE): Used to specify whether the trigger signal from simple or complex instruction (data or tuple) break should trigger iFlowtrace tracing functions or not. A value of 0 disables trigger signals from instruction breaks, and 1 enables triggers for the same.

- Bits 14:0 (IBrk/DBrk): Used to explicitly specify which instruction (data or tuple) breaks enable or disable iFlowtrace. A value of 0 implies that trace is turned off (unconditional trace stop) and a value of 1 specifies that the trigger enables trace (unconditional trace start).

8.10 PC/Data Address Sampling

It is often useful for program profiling and analysis to periodically sample the value of the PC. This information can be used for statistical profiling akin to gprof, and is also very useful for detecting hot-spots in the code. In a multi-threaded environment, this information can be used to understand thread behavior, and to verify thread scheduling mechanisms in the absence of a full-fledged tracing facility like PDtrace.

The PC sampling feature is optional.When implemented, PC sampling can be turned on or off using an enable bit; when the feature is enabled, the PC value is continually sampled.

The presence or absence of PC Sampling is indicated by the PCS (PC Sample) bit in the Debug Control Register. If PC sampling is implemented, and the PCSe (PC Sample Enable) bit in the Debug Control Register is also set to one, then the PC values are constantly sampled at the defined rate (DCR_{PCR}) and written to a TAP register. The old value in the TAP register is overwritten by the new value, even if this register has not been read out by the debug probe.

The presence or absence of Data Address Sampling is indicated by the DAS (Data Address Sample) bit in the Debug Control Register and enabled by the DASe (Data Address Sampling Enable) bit in the Debug Control Register.

The sample rate is specified by the 3-bit PCR (PC Sample Rate) field (bits 8:6) in the Debug Control Register (DCR). These three bits encode a value 2^5 to 2^{12} in a manner similar to the specification of SyncPeriod. When the implementation allows these bits to be written, the internal PC sample counter will be reset by each write, so that counting for the requested sample rate is immediately restarted.

The sample format includes a New data bit, the sampled value, the ASID of the sampled value (if not disabled by PCnoASID, bit 25 in DCR). Figure 8.5 and Figure 8.4 show the format of the sampled values in the PCSAMPLE register. The New data bit is used by the probe to determine if the sampled data just read out is new or has already been read and must be discarded.

The sampled PC value is the PC of the graduating instruction in the current cycle. If the processor is stalled when the PC sample counter overflows, then the sampled PC is the PC of the next graduating instruction. The processor continues to sample the PC value even when it is in Debug mode.

Note that some of the smaller sample periods can be shorter than the time needed to read out the sampled value. That is, it might take 41 (TCK) clock ticks to read a MIPS32 sample, while the smallest sample period is 32 (processor) clocks. While the sample is being read out, multiple samples may be taken and discarded, needlessly wasting power. To reduce unnecessary overhead, the TAP register includes only those fields that are enabled. If both PC Sampling and Data Sampling are enabled, then both samples are included in the PC Sample scan register. PC Sample is in the least significant bits followed by a Data Address Sample. If either PC Sampling or Data Address Sampling is disabled, then the TAP register does not include that sample. The total scan length is 49 * 2 = 82 bits if all fields are present and enabled.

8.10.1 PC Sampling in Wait State

Note that the processor samples PC even when it is asleep, that is, in a WAIT state. This permits an analysis of the amount of time spent by a processor in WAIT state which may be used for example to revert to a low power mode during the non-execution phase of a real-time application. But counting cycles to update the PC sample value is a waste of power. Hence, when in a WAIT state, the processor must simply switch the New bit to 1 each time it is set to 0 by the probe hardware. Hence, the external agent or probe reading the PC value will detect a WAIT instruction for as long as the processor remains in the WAIT state. When the processor leaves the WAIT state, then counting is resumed as before.

8.10.2 Data Address Sampling

The PC sampling mechanism to allow sampling of data (load and store) addresses. This feature is enabled with DASe, bit 23 in the Debug Control Register. When enabled, the PCSAMPLE scan register includes a data address sample. All load and store addresses can be captured, or they can be qualified using a data breakpoint trigger. DASQ=1 configures data sampling to record a data address only when it triggers data breakpoint 0. To be used for Data Address Sampling qualification, data breakpoint 0 must be enabled using its TE (trigger enable) bit.

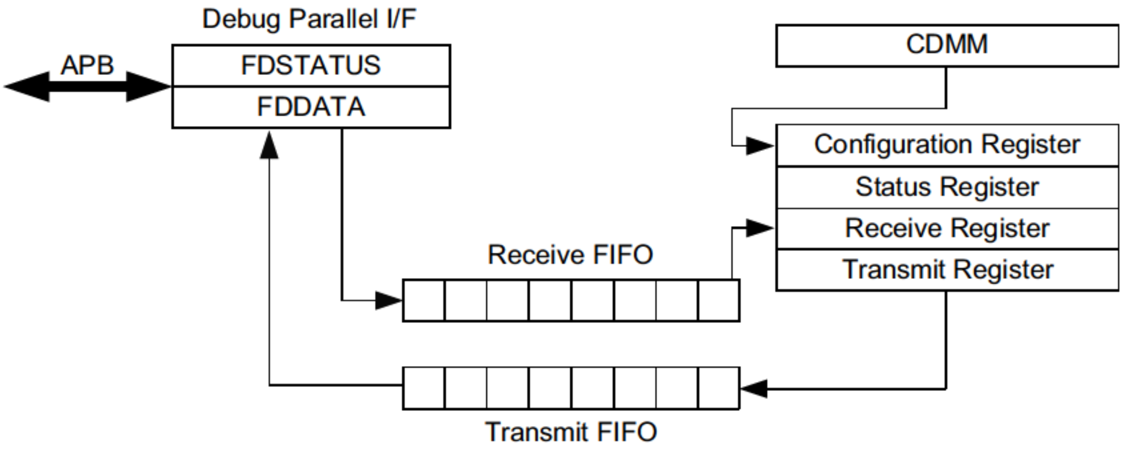
PCSR controls how often data addresses are sampled. When the PCSR counter triggers, the most recent load/store address generated is accepted and made available to shift out through PCSAMPLE.

8.11 Fast Debug Channel (FDC)

The M6200 core includes optional FDC as a mechanism for data transfers between a debug host/probe and a target. FDC provides a FIFO buffering scheme to transfer data, with low CPU overhead and minimized waiting time. The data transfer occurs in the background with hardware FIFOs, and the target CPU can either choose to check the status of the transfer by polling periodically, or it can choose to be interrupted at the end of the transfer. The FDC is shown in Figure 8.9.

Refer to Section 8.2.10 "FDSTATUS and FDDATA Registers" for more information.

Figure 8.33 FDC Overview



FDC utilizes two First In First Out (FIFO) structures to buffer data between the core and probe. The probe uses the FDC TAP instruction to access these FIFOs, while the core itself accesses them using memory accesses. To transfer data out of the core, the core writes one or more pieces of data to the transmit FIFO. At this time, the core can resume doing other work. An external probe would examine the status of the transmit FIFO periodically. If there is data to be read, the probe starts to receive data from the FIFO, one entry at a time. When all data from the FIFO has been drained, the probe goes back to waiting for more data. The core can either choose to be informed of the empty transmit FIFO via an interrupt, or it can choose to periodically check the status. Receiving data works in a similar manner - the probe writes to the receive FIFO. At that time, the core is either interrupted, or finds out via polling a status bit. The core can then do load accesses to the receive FIFO and receive data being sent to it by the probe. The TAP transfer is bidirectional - a single shift can be pulling transmit data and putting receive data at the same time.

The primary advantage of FDC over normal processor accesses or fastdata accesses is that it does not require the core to be blocked when the probe is reading or writing to the data transfer FIFOs. This significantly reduces the core overhead and makes the data transfer far less intrusive to the code executing on the core.

The FDC memory mapped registers are located in the common device memory map (CDMM) region. FDC has a device ID of 0xFD.

8.11.1 Common Device Memory Map

Software on the core accesses FDC through memory-mapped registers, located within the Common Device Memory Map (CDMM). The CDMM is a region of physical address space that is reserved for mapping IO device configuration registers within a MIPS processor. The base address and enabling of this region is controlled by the *CDMMBase* CP0 register, as described in Refer to 5.2.21 "CDMMBase Register (CP0 Register 15, Select 2)" on page 115.

Refer to *MIPS® Architecture For Programmers Volume III* [12] for full details on the CDMM.

8.11.2 Fast Debug Channel Interrupt

The FDC block can generate an interrupt to inform software of incoming data being available or space being available in the outgoing FIFO. This interrupt is handled similarly to the timer or performance counter interrupts. The *CauseFDCI* bit indicates that the interrupt is pending. The interrupt is also sent to the core output *SI_FDCI* where it is combined with one of the *SI_Int* pins. For non-EIC mode, the *SI_IPFDCI* input indicates which interrupt pin is has been combined with and this information is reflected in the *IntCtlIPFDCI* field. Note that this interrupt is a regular interrupt and not a debug interrupt.

The FDC Configuration Register (see Section 8.11.6.2 "FDC Configuration (FDCFG) Register (Offset 0x8)") includes fields for enabling and setting the threshold for generating each interrupt. Receive and transmit interrupt thresholds are specified independently, but they are ORed together to form a single interrupt.

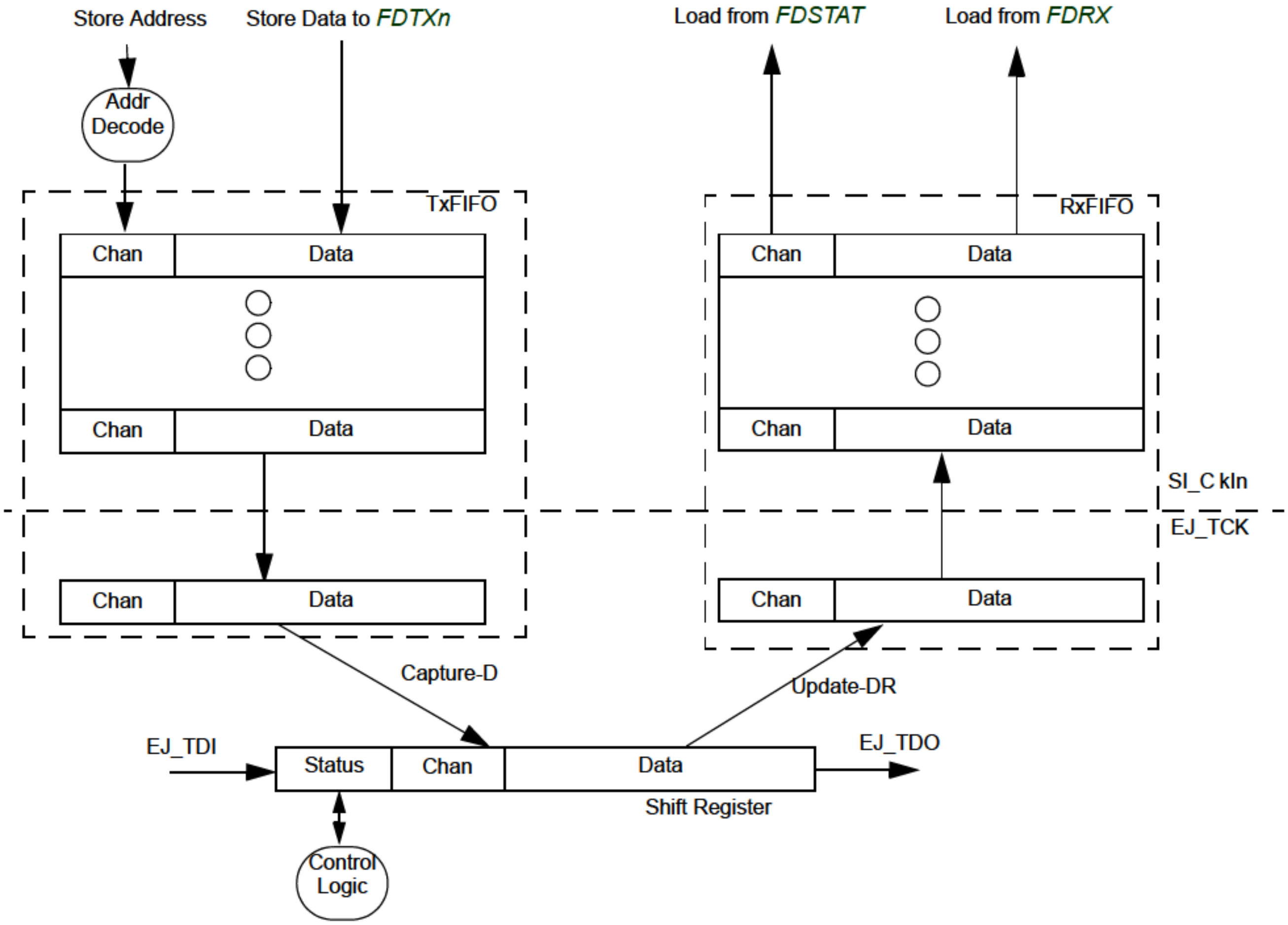
The following interrupt thresholds are supported:

- Interrupts Disabled: No interrupt will be generated and software must poll the status registers to determine if incoming data is available or if there is space for outgoing data.
- Minimum Core Overhead: This setting minimizes the core overhead by not generating an interrupt until the receive FIFO (RxFIFO) is completely full or the transmit FIFO (TxFIFO) is completely empty.
- Minimum latency: To have the core take data as soon as it is available, the receive interrupt can be fired whenever the RxFIFO is not empty. There is a complimentary TxFIFO not full setting although that may not be quite as useful.
- Maximum bandwidth: When configured for minimum core overhead, bandwidth between the probe and core can be wasted if the core does not service the interrupt before the next transfer occurs. To reduce the chances of this happening, the interrupt threshold can be set to almost full or almost empty to generate an interrupt earlier. This setting causes receive interrupts to be generated when there are 0 or 1 unused RxFIFO entries. Transmit interrupts are generated when there are 0 or 1 used TxFIFO entries (see note in following section about this condition)

8.11.3 M6200 Core FDC Buffers

Figure 8.34 shows the general organization of the transmit and receive buffers on the M6200 core.

Figure 8.34 Fast Debug Channel Buffer Organization



One particular thing to note is the asynchronous crossings between the *EJ_TCK* and *SI_ClkIn* clock domains. This crossing is handled with a handshake interface that safely transfers data between the domains. Two data registers are included in this interface, one in the source domain and one in the destination domain. The control logic actively manages these registers so that they can be used as FIFO entries. The fact that one FIFO entry is in the *EJ_TCK* clock domain is normally transparent, but it can create some unexpected behavior:

- TxFIFO availability: Data is first written into the *SI_Clk* FIFO entries, then into the *EJ_TCK* FIFO entry, requiring several *EJ_TCK* cycles to complete the handshake and move the data. *EJ_TCK* is generally much slower than *SI_ClkIn*, and may even be stopped (although that would be uncommon when this feature is in use). This can result in not enough space for new data, even though there are only N-1 data values queued up. To prevent the loss of data, the *TxF* flag in *FDSTAT* is set when all of the *SI_ClkIn* FIFO entries are full. Software writes to the FIFO should always check the *TxF* bit before attempting the write and should not make any assumptions about being able to use all entries arbitrarily. i.e., software seeing the *FxE* bit set should not assume that it can write *TxCnt* data words without checking for full.

- TxFIFO Almost Empty Interrupt: As transmit data moves from *SI_ClkIn* to *EJ_TCK*, both of the flops will temporarily look full. This makes it difficult to determine when just 1 FIFO entry is in use. To enable a simpler condition, the almost empty TxInterrupt condition is set when all of the *SI_ClkIn* FIFO entries are empty. When this condition is met, there will be 0 or 1 valid entries. However, the interrupt will not be asserted when there is only one valid entry if it is an *SI_ClkIn* entry

- The RxFIFO has similar characteristics, but these are even less visible to software since *SI_ClkIn* must be running to access the FDC registers.

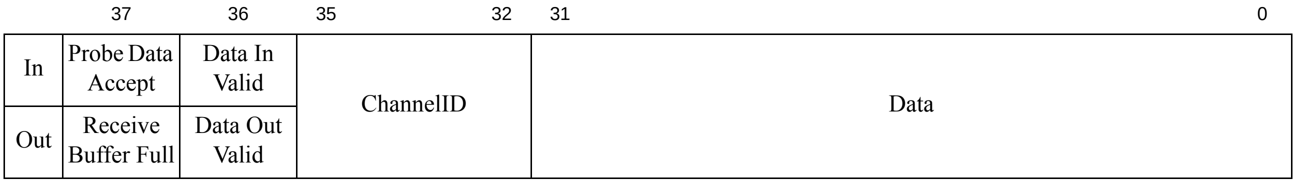
8.11.4 Sleep mode

FDC data transfers do not prevent the core from entering sleep mode and will proceed normally in sleep mode. The FDC block monitors the TAP interface signals with a free-running clock. When new receive data is available or transmit data can be sent, the gated clock will be enabled for a few cycles to transfer the data and then allowed to stop again. If FDC interrupts are enabled, transferring data may cause an interrupt to be generated which can wake the core up.

8.11.5 FDC TAP Register

The FDC TAP instruction performs a 38-bit bidirectional transfer of the FDC TAP register. The register format is shown in Figure 8.35 and the fields are described in Figure 8.38

Figure 8.35 FDC TAP Register Format



	37	36	35 32	31 0
In	Probe Data Accept	Data In Valid	ChannelID	Data
Out	Receive Buffer Full	Data Out Valid		

Table 8.38 FDC TAP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Probe Data Accept	37	Indicates to core that the probe is accepting the data that was scanned out.	W	Undefined
Data In Valid	36	Indicates to core that the probe is sending new data to the receive FIFO.	W	Undefined
Receive Buffer Full	37	Indicates to probe that the receive buffer is full and the core will not accept the data being scanned in. Analogous to ProbeDataAccept, but opposite polarity	R	0x0
Data Out Valid	36	Indicates to probe that the core is sending new data from the transmit FIFO	R	0
ChannelID	35:32	Channel number associated with the data being scanned in or out. This field can be used to indicate the type of data that is being sent and allow independent communication channels Scanning in a value with ChannelID=0xd and Data In Valid = 0 will generate a receive interrupt. This can be used when the probe has completed sending data to the core.	R/W	Undefined

Table 8.38 FDC TAP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Data	31:0	Data value being scanned in or out	R/W	Undefined

8.11.6 Fast Debug Channel Registers

This section describes the Fast Debug Channel registers. CPU access to FDC is via loads and stores to the FDC device in the Common Device Memory Map (CDMM) region. These registers provide access control, configuration and status information, as well as access to the transmit and receive FIFOs. The registers and their respective offsets are shown in Table 8.39

Table 8.39 FDC Register Mapping

Offset in CDMM device block	Register Mnemonic	Register Name and Description
0x0	FDACSR	FDC Access Control and Status Register
0x8	FDCFG	FDC Configuration Register
0x10	FDSTAT	FDC Status Register
0x18	FDRX	FDC Receive Register
0x20 + 0x8* n	FDTXn	FDC Transmit Register n (0 ≤ n ≤ 15)

8.11.6.1 FDC Access Control and Status (FDACSR) Register (Offset 0x0)

This is the general CDMM Access Control and Status register which defines the device type and size and controls user and supervisor access to the remaining FDC registers. The Access Control and Status register itself is only accessible in kernel mode. Figure 8.36 has the format of an Access Control and Status register (shown as a 64-bit register), and Table 8.40 describes the register fields.

Figure 8.36 FDC Access Control and Status Register

63 32	31 24	23 22	21 16	15 12	11 4	3	2	1	0
0	DevID	0	DevSize	DevRev	0	Uw	Ur	Sw	Sr

Table 8.40 FDC Access Control and Status Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
DevType	31:24	This field specifies the type of device.	R	0xfd
DevSize	21:16	This field specifies the number of extra 64-byte blocks allocated to this device. The value 0x2 indicates that this device uses 2 extra, or 3 total blocks.	R	0x2
DevRev	15:12	This field specifies the revision number of the device. The value 0x0 indicates that this is the initial version of FDC	R	0x0

Table 8.40 FDC Access Control and Status Register Field Descriptions (Continued)

Fields				
Name	Bits	Description	Read / Write	Reset State
Uw	3	This bit indicates if user-mode write access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to write to the device while in user mode with access disabled is ignored.	R/W	0
Ur	2	This bit indicates if user-mode read access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to read from the device while in user mode with access disabled will return 0 and not change any state.	R/W	0
Sw	1	This bit indicates if supervisor-mode write access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled.	R/W	0
Sr	0	This bit indicates if supervisor-mode read access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled.	R/W	0
0	11:4	Reserved for future use. Ignored on write; returns zero on read.	R	0

8.11.6.2 FDC Configuration (FDCFG) Register (Offset 0x8)

The FDC configuration register holds information about the current configuration of the Fast Debug Channel mechanism. Figure 8.37 has the format of the FDC Configuration register, and Table 8.41 describes the register fields.

Figure 8.37 FDC Configuration Register

31 20	19 18	17 16	15 8	7 0
0	Tx_IntThresh	Rx_IntThresh	TxFIFOSize	RxFIFOSize

Table 8.41 FDC Configuration Register Field Descriptions

Fields				
Name	Bits	Description	Read / Write	Reset State
0	31:20	Reserved for future use. Read as zeros, must be written as zeros.	R	0

Table 8.41 FDC Configuration Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>TxIntThresh</td><td>19:18</td><td>Controls whether transmit interrupts are enabled and the state of the TxFIFO needed to generate an interrupt.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Transmit Interrupt Disabled</td></tr>
<tr><td>1</td><td>Empty</td></tr>
<tr><td>2</td><td>Not Full</td></tr>
<tr><td>3</td><td>Almost Empty - zero or one entry in use (see 8.11.2 for specifics)</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>RxIntThresh</td><td>17:16</td><td>Controls whether receive interrupts are enabled and the state of the RxFIFO needed to generate an interrupt.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Receive Interrupt Disabled</td></tr>
<tr><td>1</td><td>Full</td></tr>
<tr><td>2</td><td>Not empty</td></tr>
<tr><td>3</td><td>Almost Full - zero or one entry free</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>TxFIFOSize</td><td>15:8</td><td>This field holds the total number of entries in the transmit FIFO.</td><td>R</td><td>Preset</td></tr>
<tr><td>RxFIFOSize</td><td>7:0</td><td>This field holds the total number of entries in the receive FIFO.</td><td>R</td><td>Preset</td></tr>
</table>

8.11.6.3 FDC Status (FDSTAT) Register (Offset 0x10)

The FDC Status register holds up to date state information for the FDC mechanism. Figure 8.38 shows the format of the FDC Status register, and Table 8.42 describes the register fields.

Figure 8.38 FDC Status Register

31 24	23 16	15 8	7 4	3	2	1	0
Tx_Count	Rx_Count	0	RxChan	RxE	RxF	TxE	TxF

Table 8.42 FDC Status Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Tx_Count	31:24	This optional field is not implemented and will read as 0	R	0
Rx_Count	23:16	This optional field is not implemented and will read as 0	R	0
0	15:8	Reserved for future use. Must be written as zeros and read as zeros.	R	0
RxChan	7:4	This field indicates the channel number used by the top item in the receive FIFO. This field is only valid if RxE=0.	R	Undefined

Table 8.42 FDC Status Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
RxE	3	If RxE is set, the receive FIFO is empty. If RxE is not set, the FIFO is not empty.	R	1
RxF	2	If RxF is set, the receive FIFO is full. If RxF is not set, the FIFO is not full.	R	0
TxE	1	If TxE is set, the transmit FIFO is empty. If TxE is not set, the FIFO is not empty.	R	1
TxF	0	If TxF is set, the transmit FIFO is full. If TxF is not set, the FIFO is not full.	R	0

8.11.6.4 FDC Receive (FDRX) Register (Offset 0x18)

This register exposes the top entry in the receive FIFO. A read from this register returns the top item in the FIFO and removes it from the FIFO itself. The result of a write to this register is **UNDEFINED**. The result of a read when the FIFO is empty is also **UNDEFINED** so software must check the RxE flag in FDSTAT prior to reading. Figure 8.39 shows the format of the FDC Receive register, and Table 8.43 describes the register fields.

Figure 8.39 FDC Receive Register

31	0
RxData	

Table 8.43 FDC Receive Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
RxData	31:0	This register holds the top entry in the receive FIFO	R	Undefined

8.11.6.5 FDC Transmit n (FDTXn) Registers (Offset 0x20 + 0x8*n)

These sixteen registers access the bottom entry in the transmit FIFO. The different addresses are used to generate a 4b channel identifier that is attached to the data value. This allows software to track different event types without needing to reserve a portion of the 32b data as a tag. A write to one of these registers results in a write to the transmit FIFO of the data value and channel ID corresponding to the register being written. Reads from these registers are **UNDEFINED**. Attempting to write to the transmit FIFO if it is full has **UNDEFINED** results. Hence, the software running on the core must check the *TxF* flag in FDSTAT to ensure that there is space for the write. Figure 8.40 shows the format of the FDC Transmit register, and Table 8.44 describes the register fields.

Figure 8.40 FDC Transmit Register

31	0
TxData	

Table 8.44 FDC Transmit Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read / Write**	**Reset State**
TxData	31:0	This register holds the bottom entry in the transmit FIFO	W, Undefined value on read	Undefined

Table 8.45 FDTXn Address Decode

Address	Channel	Address	Channel	Address	Channel	Address	Channel
0x20	0x0	0x40	0x4	0x60	0x8	0x80	0xc
0x28	0x1	0x48	0x5	0x68	0x9	0x88	0xd
0x30	0x2	0x50	0x6	0x70	0xa	0x90	0xe
0x38	0x3	0x58	0x7	0x78	0xb	0x98	0xf

8.12 Examples of Debug Operations

Example: Write to OCR, no other operations. The MDH JTAG sequence is:

```
IR: select DEVICEADDR
DR: write 0x0    -> assumes device address of Viking core is 0.
                        Note: A[6:0] are reserved and not writable, return 0
IR: select APBACCESS
DR: scan in: execute=1; RnW=W; Addr=0x05 (OCR reg); data=32 bit value to be written
          scanned out: <don't care>
DR: scan in: execute=0;  <the rest are don't cares>
          scan out: if valid=1 write completed, OK/Fault and error codes are also valid; probe checks these for any errors
                    if valid = 0 probe continues to scan in execute = 0 until OCR= 1 is returned
```

Example: Read OCR register, no other operations.

Assuming DEVICEADDR is already set up and IR has selected APBACCESS

```
DR: scan in: execute=1; RnW=R; Addr=0x05; data=<don't care>
      scanned out: <don't care>
DR: scan in: execute=0  <the remaining fields are don't cares>
      scan out: valid = 1; OK/Fault = 1; data=OCR value
         if valid=1 read completed
         if valid = 0 probe continues to scan in execute = 0 until valid = 1 at which time data = OCR value
```

Example: Cause core to execute a sequence of instructions in debug mode

Assuming DEVICEADDR is already set up and IR has selected APBACCESS

```
  First - check that core is in debug mode by polling OCR
     DR: scan in: execute=1; RnW=R; Addr=0x05 (OCR reg);
          scanned out: <don't care>
     DR: scan in: execute=1; RnW=R; Addr=0x05 (OCR reg);
         scan out: OCR value (from previous scan in).  probe checks: if OCR.DM = 0 continue scans
     DR: scan in: execute=1; RnW=R; Addr=0x05 (OCR reg);
         scan out: OCR value (from previous scan in).  if OCR.DM = 1 move on
       now cause core to execute a series of instructions in debug mode
     DR: scan in: execute=1; RnW=W; Addr=0x04 (DATA reg);  set Data = instr1 opcode
        scan out: ignore
     DR: scan in: execute=1; RnW=W; Addr=0x04 (DATA reg);  set Data = instr2 opcode
        scan out:  if valid=1 write completed so instr1 has executed
              if valid = 0 probe continues to scan in instr2 with execute = 1 until valid = 1 at which time instr1 completed
     DR: scan in: execute=1; RnW=W; Addr=0x04 (DATA reg);  set Data = instr3 opcode
       <etc on polling valid = 1>
```

If the instruction is a load from dmseg then the next DR scan in is a Write to DATA with the value being passed from the probe to the core

If the instruction is a store to dmseg then the next DR scan in is a Read of DATA of the value being picked up by the probe

----

Example: Read the DCR register, which is at offset 0 of drseg

Assuming DEVICEADDR is already set up and IR has selected APBACCESS

```
     DR: scan in: execute=1; RnW=W; Addr=0x06 (DRSEG_ADDR reg);  set Data = 0x7F300000 (A31=0 means no
     auto-incr)
        scan out: ignore
     DR: scan in: execute=1; RnW=R; Addr=0x07 (DRSEG_DATA reg);  Data = <don’t care>
        scan out: check valid=1 that write to DRSEG_ADDR was successful
     DR: scan in: execute=0; <remaining fields are don't cares>
        scan out: check valid=1 that read of DRSEG_DATA was successful; if it wasn’t continue the scan, if it was,
     save data field which is DCR contents.
```

Example: Take multiple pcsamples

Assuming DEVICEADDR is already set up and IR has selected APBACCESS

The sequence is to read PCSAMPLE1 register then PCSAMPLE2 register continuously; probe concatenates the two sample data values into one complete sample set. Note: the data scanned out is always from the previous sample register read

```
     DR: scan in: execute=1; RnW=R; Addr=0x08 (PCSAMPLE1 reg);  Data = <don’t care>
        scan out: ignore
     DR: scan in: execute=1; RnW=R; Addr=0x09 (PCSAMPLE2 reg);  Data = <don’t care>
        scan out: check valid=1 that read of PCSAMPLE1 was successful; if it was, save data as lower 32 bits of first
     sample
     DR: scan in: execute=1; RnW=R; Addr=0x08 (PCSAMPLE1 reg);  Data = <don’t care>    Start 2nd sample
```

scan out: check valid=1 that read of PCSAMPLE2 was successful; if it was, save data as upper 32 bits of first sample; probe extracts the fields, buffers, then sends a set of samples to host the size of a comm packet
<continue>

The probe may need to check for target hitting a breakpoint or being reset periodically. So the probe will need to read the OCR to check DbgBrk and Rocc, then return to pc sampling.

Chapter 9

MIPS32® Instruction Set Architecture

This chapter provides an overview of the M6200 MIPS32 instruction set, including a description of CPU instruction formats, instruction access types, and descriptions of the instructions grouped by function.

9.1 MIPS32® Release 6 ISA

The M6200 processor core supports the MIPS32 Release 6 ISA, which maintains backward-compatibility with previous releases using trap-and-emulate or trap-and-patch; all pre-Release 6 binaries can execute under binary translation. The new R6 unconditional compact branches without a forbidden slot allow single-instruction patching.

9.2 CPU Instruction Formats

All CPU instructions consist of a single 32-bit word, aligned on a word boundary.

There are three basic instruction formats: register (R-type), immediate (I-type), and jump (J-type), shown in Figure 9.1 through Figure 9.8). The fields in the instruction formats are described in Table 9.1.

For MIPS instructions, the layout of the bit fields in instructions is little-endian, regardless of the endianness mode in which the processor is executing. Bit 0 of an instruction is always the right-most bit in the instruction, while bit 31 is always the left-most bit in the instruction. The major opcode is always the left-most 6 bits in the instruction.

Table 9.1 CPU Instruction Format Fields

Field	Description
opcode	6-bit primary operation code. The 6 most-significant bits of the instruction encoding.
function	6-bit function field used to specify functions within the primary opcode SPECIAL. The 6 least-significant bits of the instruction encoding.
rd	5-bit specifier for the destination register.
rs	5-bit specifier for the source register.
rt	5-bit specifier for the target (source/destination) register
sa	5-bit shift amount
rt	5-bit specifier for the target (source/destination) register or used to specify functions within the primary *opcode* REGIMM.
sa	5-bit shift amount.
immediate16	16-bit signed *immediate* used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement.

Table 9.1 CPU Instruction Format Fields (Continued)

Field	Description
immediate	A constant stored inside the instruction (as opposed to a constant separately in memory, that must be accessed using a load instruction). Unless further qualified, *immediate* typically refers to a 16-bit immediate occupying the least significant 16 bits of a 32-bit MIPS instruction. This 16-bit signed immediate is used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacements. Some instructions have other immediate widths, for example, 9-, 10-, 21-, and 26-bit offsets and displacements, and the 26-bit instr_index.
offset	An immediate constant in the instruction, used in forming a memory address or a PC-relative branch target. 16-bit offsets using the 16-bit immediate field are most common, although certain instructions have 9-, 18-, 19-, 21-, and 26-bit offsets.
instr_index	26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address.

Figure 9.1 Register (R-Type) CPU Instruction Format

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

Several different Immediate (I-Type) instruction formats are shown in Figure 9.2. The 16-bit immediate constant inside the first instruction format can be used for both computation and memory/branch offset; the immediates in the other formats are mainly used as memory offset or branch displacement.

Figure 9.2 Immediate (I-Type) CPU Instruction Formats Summary

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	immediate		
opcode	rd	offset			
opcode	offset				
opcode	rs	rt	rd	offset	
opcode	base	rt	offset		function

The most common MIPS Immediate (I-Type) instruction format is the Imm16 format shown in Figure 9.3. The 16-bit signed immediate is used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacements.

Figure 9.3 Immediate (I-Type) Imm16 CPU Instruction Format

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6	5	5	16

The Immediate (I-Type) Off21 CPU instruction format, shown in Figure 9.4, is used by instructions that compare a register against zero and branch (e.g., BLTZC), with larger than the usual 16-bit span.

Certain PC-relative instructions use offsets 18- and 19-bits wide, using the low-order bits of the 21-bit immediate as extra opcode bits.

Figure 9.4 Immediate (I-Type) Off21 CPU Instruction Format

31 26	25 21	20 0
opcode	rd	offset
6	5	21

The Immediate (I-Type) Off26 CPU instruction format, shown in Figure 9.5, is used for PC- relative branches with very large displacements, namely BC and BALC. The 26-bit immediate, shifted left by 2 bits, yields a span of 28-bits, or +/- 128 megabytes of instructions.

Figure 9.5 Immediate (I-Type) Off26 CPU Instruction Format

31 26	25 0
opcode	offset
6	26

The Immediate (I-Type) Off11 CPU instruction format, shown in Figure 9.6, is used for encodings of Coprocessor 2 load and store instructions (LWC2, SWC2, LDC2, SWC2).

Figure 9.6 Immediate (I-Type) Off11 CPU Instruction Format

31 26	25 21	20 16	15 11	10 0
opcode	rs	rt	rd	offset
6	5	5	5	11

The Immediate (I-Type) Off9 CPU Instruction Format format, shown in Figure 9.7, provides a 9-bit memory offset.

Figure 9.7 Immediate (I-Type) Off9 CPU Instruction Format

31 26	25 21	20 16	15 7	6	5 0
opcode	base	rt	offset	0	function
6	5	5	9		6

The Jump (J-Type) CPU instruction format is shown in Figure 9.8. This format is used in the instructions J (jump), JAL (jump-and-link), and JALX (jump and link-exchange), where the *instr_index* bits replaced corresponding bits in the PC.

Figure 9.8 Jump (J-Type) CPU Instruction Format

31 26	25 0
opcode	instr_index
6	26

9.2.0.1 PC-relative Loads

PC-relative loads have a span of +/- 1 Megabytes. The LWPC instruction loads a 32-bit word from a PC-relative address, formed by adding the word-aligned PC to a sign- extended 19-bit immediate shifted left by 2 bits, giving a 21-bit span.

Note that PC-relative load instructions can only generate aligned memory addresses.

9.3 Load and Store Instructions

MIPS processors use a load/store architecture; all operations are performed on operands in processor registers. and main memory is accessed only through load and store instructions. Load and store instructions are of type immediate (I-type).

9.3.1 Scheduling a Load Delay Slot

A load instruction that does not allow its result to be used by the instruction immediately following is called a *delayed load instruction*. The instruction slot immediately following this delayed load instruction is referred to as the *load delay slot*.

In the M6200 core, the instruction immediately following a load instruction can use the contents of the loaded register; however, in such cases hardware interlocks insert additional real cycles. Although not required, the scheduling of load delay slots can be desirable, both for performance and R-Series processor compatibility.

9.3.2 Load and Store Access Types

Access type indicates the size of a core data item to be loaded or stored, set by the load or store instruction opcode. The following data sizes are transferred by CPU load and store instructions:

- Byte
- Halfword
- Word

Signed and unsigned integers of different sizes are supported by loads that either sign-extend or zero-extend the data loaded into the register.

Regardless of access type or byte ordering (endianness), the address given specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the three low-order bits of the address, define the bytes accessed within the addressed word as shown in Table 9.2. Only the combinations shown in the Table are permissible; other combinations cause address error exceptions.

Table 9.2 Byte Access Within a Word

				Bytes Accessed							
	Low Order Address Bits			**Big Endian (31--------------------0)**				**Little Endian (31--------------------0)**			
Access Type	**2**	**1**	**0**	**Byte**				**Byte**			
Word	0	0	0	0	1	2	3	3	2	1	0
Triplebyte	0	0	0	0	1	2	▨	▨	2	1	0
	0	0	1	▨	1	2	3	3	2	1	▨
Halfword	0	0	0	0	1	▨	▨	▨	▨	1	0
	0	1	0	▨	▨	2	3	3	2	▨	▨

Table 9.2 Byte Access Within a Word

				Bytes Accessed							
	Low Order Address Bits			Big Endian (31---------------------0)				Little Endian (31---------------------0)			
Access Type	**2**	**1**	**0**	**Byte**				**Byte**			
Byte	0	0	0	0	▨	▨	▨	▨	▨	▨	0
	0	0	1	▨	1	▨	▨	▨	▨	1	▨
	0	1	0	▨	▨	2	▨	▨	2	▨	▨
	0	1	1	▨	▨	▨	3	3	▨	▨	▨

9.3.3 PC-relative Loads

The M6200 core provides the following PC-relative loads with a span of +/- 1 Megabytes:

- LWPC: Loads a 32-bit word from a PC-relative address, formed by adding the word-aligned PC to a sign-extended 19-bit immediate shifted left by 2 bits, giving a 21-bit span.

- LWUPC: Loads a 32-bit unsigned word from a PC-relative address, formed by adding the word-aligned PC to a sign-extended 19-bit immediate shifted left by 3 bits, giving a 21-bit span.

- LDPC: Loads a 64-bit doubleword from a PC-relative address, formed by adding the PC, aligned to 8 bytes by masking off the low 3 bits, to a sign-extended 18-bit immediate, shifted left by 3 bits, giving a 21-bit span.

Note that PC-relative load instructions can only generate aligned memory addresses.

Table 9.3 PC-relative Loads

Mnemonic	Instruction
LWPC	Load Word, PC-relative
LWUPC	Load Unsigned Word, PC-relative
LDPC	Load Doubleword, PC-relative

9.4 Computational Instructions

Computational instructions can be either in register (R-type) format, in which both operands are registers, or in immediate (I-type) format, in which one operand is a 16-bit immediate.

Computational instructions perform the following operations on register values:

- Arithmetic

- Logical

- Shift

- Multiply

- Divide

These operations fit in the following four categories of computational instructions:

- ALU Immediate
- Three-operand Register-type
- Shift
- Multiply And Divide

9.4.1 Cycle Timing for Multiply and Divide Instructions

Any multiply instruction in the integer pipeline is transferred to the multiplier as remaining instructions continue through the pipeline.

9.4.2 ALU Immediate and Three-Operand Instructions

The *immediate* operand is treated as a signed value for the arithmetic and compare instructions, and treated as a logical value (zero-extended to register length) for the logical instructions.

The M6200 provides the instructions shown in Table 9.4 that are especially suited to address computations and the creation of large constants. Large constants can be formed efficiently using the upper bits with the 16-bit immediates available in most memory access and arithmetic instructions.

Table 9.4 Address Computation and Large Constant Instructions

Mnemonic	Instruction
LSA	Left Shift Add (Word)
AUI	Add Upper Immediate (Word)
ADDIUPC	Add Immediate Unsigned to PC
AUIPC	Add Upper Immediate to PC
ALUIPC	Aligned Add Upper Immediate to PC

- **Left Shift Add:** LSA add two registers, one of which is optionally shifted by a scaling factor from 1 to 4, corresponding to a scaling multiplication, e.g., by element size in an array, by 1, 2, 4, 8, or 16.

- **Add Upper Immediate:** AUI adds an immediate value to a register. The immediate value is sign-extended and shifted by 16 bits.

- **Add Immediate Unsigned PC:** ADDIUPC adds an immediate value to thelower bits of the PC, This instruction performs a PC-relative address calculation. The 19-bit immediate is shifted left by 2 bits, sign-extended, and added to the address of the ADDIUPC instruction. The result is placed in GPR rs.

- **Add Upper Immediate PC:** AUIPC adds the immediate to the upper bits 31:16 of the PC. This instruction performs a PC-relative address calculation. The 16-bit immediate is shifted left by 16 bits and sign-extended, and added to the address of the AUIPC instruction. The result is placed in GPR rs.

- **Add Lower Unsigned Immediate PC:** ALUIPC adds the immediate to the upper bits ofthe PC, zeroing the low 16 bits of the PC. This instruction performs a PC-relative address calculation. The 16-bit immediate is shifted left by 16 bits, sign-extended, and added to the address of the ALUIPC instruction. The low 16 bits of the result are cleared, that is the result is aligned on a 64K boundary. The result is placed in GPR rs.

Note: These instructions sign-extend the 16-bit immediate. The “unsigned” in the name “Add Immediate Unsigned to PC” means that 32-bit overflow detection is not performed. Compare to ADD/ADDU.

9.4.3 Shift Instructions

The BITSWAP instruction reverses the bits in every byte of its operand. BITSWAP corresponds to MIPS DSP Module BITREV.

The ALIGN instruction concatenates its two operands and the same-width contiguous subset from the concatenation at byte granularity. ALIGN is useful for extracted data at a misaligned memory address from two aligned memory load results. ALIGN corresponds to MIPS DSP Module BALIGN.

Table 9.5 Shift Instructions

Mnemonic	Instruction
ALIGN	Extract byte-aligned word from concatenation of two words
BITSWAP	Swap bits in every byte of word operand

9.4.4 Multiply and Divide Instructions

The multiply and divide instructions produce results that are the same width as their operands, using GPRs as both input and output.

- **Multiply-low** instructions (MUL, MULU) produce the low 32-bits of the product.
- **Multiply-high** instructions (MUH, MUHU) produce the high 32-bits of the product.

Note that the low half of a product is the same for signed and unsigned 2’s-complement multiplication, but the upper half differs, for example, MUL and MULU produce the same result, but MUH and MUHU produce different results.

- **Divide** instruction produce a quotient that is loaded into a single GPR destination (DIV, DIVU)
- **Modulus** instructions produce a remainder that is loaded into a single GPR destination (MOD, MODU)

If a full double-width product is desired for a multiplication, or both quotient and remainder are desired for a division, the appropriate same-width instructions should be used in close proximity to permit hardware optimizations. For example, the multiply-low instruction MULU may retain its result, to be provided to the following multiply-high MUHU instruction.

Table 9.6 lists the same-width multiply and divide instructions.

Table 9.6 Multiply/Divide Instructions

Mnemonic	Instruction
MUL	Multiply word, Low part, signed
MUH	Multiply word, High part, signed
MULU	Multiply word, Low part, Unsigned
MUHU	Multiply word, High part, Unsigned

Table 9.6 Multiply/Divide Instructions (Continued)

Mnemonic	Instruction
DIV	Divide words, signed
MOD	Modulus remainder word division, signed
DIVU	Divide words, Unsigned
MODU	Modulus remainder word division, Unsigned

9.5 Jump and Branch Instructions

Jump and branch instructions change the control flow of a program. All jump and branch instructions occur with a delay of one instruction: that is, the instruction immediately following the jump or branch (this is known as the instruction in the *delay slot*) always executes while the target instruction is being fetched from storage.

9.5.1 Overview of Jump Instructions

Subroutine calls in high-level languages are usually implemented with Jump or Jump and Link instructions, both of which are J-type instructions. In J-type format, the 26-bit target address shifts left 2 bits and combines with the high-order 4 bits of the current program counter to form an absolute address.

Returns, dispatches, and large cross-page jumps are usually implemented with the Jump Register or Jump and Link Register instructions. Both are R-type instructions that take the 32-bit byte address contained in one of the general purpose registers.

For more information about jump instructions, refer to the individual instructions in Chapter 10, “M6200 MIPS32® Processor Core Instructions” on page 234.

9.5.2 Branch Instructions

All branch instruction target addresses are computed by adding the address of the instruction in the delay slot to the 16-bit *offset* (shifted left 2 bits and sign-extended to 32 bits). All branches occur with a delay of one instruction.

9.5.2.1 Compact Jump and Compact Branch Instructions

The M6200 core provides conditional and unconditional compact branches and compact jumps, shown in Table 9.7. Conditional compact branches and jumps do not have a delay slot. but have instead a *forbidden slot*. Unconditional compact branches and jumps have neither a delay slot nor a forbidden slot.

The following instructions must not be placed in either a branch delay slot or a forbidden slots: ERET, ERETNC, DERET, WAIT, PAUSE, and any CTI, including branches and jumps,. Their occurrence is required to signal a Reserved Instruction exception.

Table 9.7 Compact Branch and Jump Instructions

Offset	Span	Mnemonic	Instruction
Unconditional Branch and Call			

Offset	Span	Mnemonic	Instruction
26	+/- 128MB	BC	Compact Branch
		BALC	Compact Branch And Link
Indexed Jumps (register + unscaled offset)			
16	+/-32K	JIC	Compact Jump Indexed
		JIALC	Compact Jump Indexed And Link
Compare to Zero			
21	+/- 4MB	BEQZC	Compact Branch if Equal to Zero
		BNEZC	Compact Branch if Not Equal to Zero
16	+/- 128KB	BLEZC	Compact Branch if Less Than or Equal to Zero
		BGEZC	Compact Branch if Greater Than or Equal to Zero
		BGTZC	Compact Branch if Greater Than Zero
		BLTZC	Compact Branch if Less Than Zero
Conditional calls, compare against zero			
16	+/- 128KB	BEQZALC	Compact Branch if Equal to Zero, And Link
		BNEZALC	Compact Branch if Not Equal to Zero, And Link
		BLEZALC	Compact Branch if Less Than or Equal to Zero, And Link
		BGEZALC	Compact Branch if Greater Than or Equal to Zero, And Link
		BGTZALC	Compact Branch if Greater Than Zero, And Link
		BLTZALC	Compact Branch if Less Than Zero, And Link
Compare equality reg-reg			
16	+/- 128KB	BEQC	Compact Branch if Equal
		BNEC	Compact Branch if Not Equal
Compare signed reg-reg			
16	+/- 128KB	BGEC	Compact Branch if Greater than or Equal
		BLTC	Compact Branch if Less Than
Compare Unsigned reg-reg			
16	+/-128KB	BGEUC	Compact Branch if Greater than or Equal, Unsigned
		BLTUC	Compact Branch if Less Than, Unsigned
Aliases Obtained by Reversing Operands			
16	+/- 128KB	BLEC	Compact Branch if Less Than or Equal
		BGTC	Compact Branch if Greater Than
		BLEUC	Compact Branch if Less than or Equal, Unsigned
		BGTUC	Compact Branch if Greater Than, Unsigned
Branch if Overflow			
16	+/-128KB	BOVC	Compact Branch if Overflow (word)
		BNVC	Compact Branch if No overflow, word

9.6 Control Instructions

Control instructions allow the software to initiate traps; they are always R-type. These instructions transfer control to a kernel-mode software exception handler. There are two types of exceptions, *conditional* and *unconditional*. They are caused by the following instructions:

- System call and breakpoint instructions, which cause unconditional exceptions (Table 9.8).
- Trap instructions, which cause conditional exceptions based on the result of a comparison (Table 9.9).

Table 9.8 System Call and Breakpoint Instructions

Mnemonic	Instruction
BREAK	Breakpoint
BREAK16	Breakpoint (16-bit Instruction Size)
SYSCALL	System Call

.

Table 9.9 Trap-on-Condition Instructions Comparing Two Registers

Mnemonic	Instruction
TEQ	Trap if Equal
TGE	Trap if Greater Than or Equal
TGEU	Trap if Greater Than or Equal Unsigned
TLT	Trap if Less Than
TLTU	Trap if Less Than Unsigned
TNE	Trap if Not Equal

9.7 Coprocessor Instructions

CP0 instructions perform operations on the System Control Coprocessor registers to manipulate the memory management and exception handling facilities of the processor. Refer to Chapter 10, “M6200 MIPS32® Processor Core Instructions” on page 234 for a listing of CP0 instructions.

9.8 Miscellaneous Instructions

9.8.1 Conditional Select Instructions

The conditional select instructions test the C- compatible zero/nonzero value of a GPR and select a GPR or 0. These instructions are compatible with the truth values in the C language, and they have only two register inputs (the third input, 0, is implicit). They are listed in Table 9.10.).

Table 9.10 CPU Conditional Select Instructions

Mnemonic	Instruction
SELEQZ	Select GPR rs if GPR rt is Equal to Zero, else select 0
SELNEZ	Select GPR rs if GPR rt is Not Equal to Zero, else select 0

9.8.2 Prefetch Instruction

The PREF instruction is used to indicate that memory is likely to be used in a particular way in the near future and should be prefetched into the cache. A *hint* field may indicate prefetch policies, such as which cache they are fetched into and whether reading or writing is intended. The PREF instruction uses register+offset addressing.

9.8.3 NOP Instructions

The NOP instruction is actually encoded as an all-zero instruction. MIPS processors special-case this encoding as performing no operation, and optimize execution of the instruction.

In addition, the SSNOP instruction takes up one issue cycle on any processor, including super-scalar implementations of the architecture. SSNOP is treated like an ordinary NOP.

Table 9.11 NOP Instructions

Mnemonic	Instruction
NOP	No Operation
SSNOP	Superscalar Inhibit NOP

9.9 MCU ASE Instructions

The MCU ASE includes some new instructions which are particularly useful in microcontroller applications.

9.9.1 ACLR

This instruction allows a bit within an uncached I/O control register to be atomically cleared; that is, the read-modify byte write sequence performed by this instruction cannot be interrupted.

9.9.2 ASET

This instruction allows a bit within an uncached I/O control register to be atomically set; that is, the read-modify byte write sequence performed by this instruction cannot be interrupted.

9.9.3 IRET

This instruction can be used as a replacement for the ERET instruction when returning from an interrupt. This instruction implements the Automated Interrupt Epilogue feature, which automates restoring some of the CP0 registers from the stack and updating the C0_Status register in preparation for returning to non-exception mode. This instruction also implements the optional Interrupt Chaining feature, which allows a subsequent interrupt to be handled without returning to non-exception mode.

9.9.4 ASET/ACLR Unique Behaviors

The ASET and ACLR instructions are atomic read-modify-write operations that typically cannot be interrupted after the operations have begun. This causes some restrictions on the handling of various debug and asynchronous exceptions:

- If a data breakpoint was enabled with data-value matching, the breakpoint exception will occur prior to the read operation, regardless of the data-value comparison result. This is preferable to the breakpoint occurring after the write operation, when the data-value in memory has already been modified.

- If a data breakpoint was enabled with store-only matching, the breakpoint exception will occur prior to the read operation, that is, before the write operation has begun.

- If there is a bus error, parity, or ECC exception initiated by the read or write operation, the ASET/ACLR will be interrupted, and it the responsibility of the SOC to deactivate the bus lock initiated by the ASET/ACLR atomic command.

Chapter 10

M6200 MIPS32® Processor Core Instructions

This chapter supplements the *MIPS32® Architecture Reference Manual, Volume II* with descriptions of instruction behavior that is specific to a M6200 processor core. For complete descriptions of all instructions, refer to *MIPS® Architecture For Programmers, Volume II: The MIPS32® Instruction Set* [12] and *MIPS® Architecture For Programmers, Volume II: microMIPS32® Instruction Set* [13].

The microMIPS instruction set is described in Chapter 11, “microMIPS32™ Instruction Set Architecture” on page 258.

The M6200 processor core also supports the instructions in the MIPS DSP Module Revision 3. The MIPS DSP Module Revision 3 instruction set is described in Chapter 2, “The MIPS® DSP Module” on page 34.

10.1 Understanding the Instruction Descriptions

Refer to *Volume II* of the *MIPS32 Architecture Reference Manual* for detailed information about the instruction descriptions, namely, the instruction fields, definition of terms, and functional notation. This section provides only basic information.

10.2 MIPS32® Instruction Set for the M6200 Core

This section provides a summary of the MIPS32 instructions for M6200 cores. microMIPS32 instructions are described in Chapter 11, “microMIPS32™ Instruction Set Architecture” on page 258.

Table 10.1 lists all M6200 MIPS32 instructions in alphabetical order. Instructions that have implementation-dependent behavior are described individually in subsequent sections; all other MIPS32 instructions are described in the *MIPS® Architecture For Programmers, Volume II: The MIPS32® Instruction Set* [12] and their descriptions are not duplicated here.

Table 10.1 MIPS32 Instruction Set

Instruction	Description	Function
ADD	Integer Add	Rd = Rs + Rt
ADDIU	Unsigned Integer Add Immediate	Rt = Rs + Immed
ADDIUPC	Add Immediate to PC	Rs = (PC + sign_extend (immediate << 2))
ADDU	Unsigned Integer Add	Rt = Rs + Rt
ALIGN	Align Word	Rd = (Rt << (8*bp)) or (rs >> (GPRLEN-8*bp))
ALUIPC	Aligned Add Upper Immediate to PC	-0x0FFFF & (PC + (Immediate << 16))
AND	Logical AND	Rd = Rs & Rt
ANDI	Logical AND Immediate	Rt = Rs AND Immediate

Table 10.1 MIPS32 Instruction Set (Continued)

Instruction	Description	Function
AUI	Add Immediate to Upper Bits	Rt = Rs + (Immediate << 16)
AUIPC	Add Upper Immediate to PC	Rs = (PC + (Immediate << 16))
ACLR	Atomic Bit Clear	See MCU ASE Instructions
ASET	Atomic Bit Set	See MCU ASE Instructions
B	Unconditional Branch (Assembler idiom for: BEQ r0, r0, offset)	PC += (int)offset

BAL	Branch and Link	GPR[31] = PC + 8
		PC += (int)offset

BALC	Branch and Link Compact	Procedure Call (No Delay Slot)
BC	Branch Compact	PC = PC + 4 + Sign_Extend (offset << 2)
BC2EQZ, BC2NEZ	Branch on COP2 Equal/Not Equal to Zero	If COP2 Condition = 0 Then PC += (int)offset If COP2 Condition != 0 Then PC += (int)offset
BEQ	Branch On Equal	If Rs = Rt Then PC += (int)offset
BGEZ	Branch on Greater Than or Equal To Zero	If !Rs[31] Then PC += (int)offset
B(LE,GE,GT,LT,EQ,NE)ZALC	Compact Zero-Compare and Branch-and-Link	If Condition (Rt) Then PC += (int)offset
B<cond>C	Compact Compare and Branch	If Condition (Rs and/or Rt) Then Compact Branch
BGTZ	Branch on Greater Than Zero	if !Rs[31] && Rs != 0 Then PC += (int)offset
BITSWAP	Swap Bits in Byte	Rd.Byte(i) = Reverse_Bits_in_Byte (Rt.Byte(i) for all Bytes i
BLEZ	Branch on Less Than or Equal to Zero	if Rs[31] || Rs == 0 Then PC += (int)offset
BLTZ	Branch on Less Than Zero	if Rs[31] Then PC += (int)offset
BNE	Branch on Not Equal	if Rs != Rt Then PC += (int)offset
BOVC, BNVC	Branch on Overflow/No Overflow, Compact	Branch if/if-not NotWordValue(Rs + Rt)
BREAK	Breakpoint	Break Exception
CACHE	Cache Operation	NOP
CFC2	Move Control Word From Coprocessor 2	Rt = CCR[2, n]
CLO	Count Leading Ones	Rd = NumLeadingOnes(Rs)
CLZ	Count Leading Zeroes	Rd = NumLeadingZeroes(Rs)
COP2	Coprocessor 2 Operation	See Coprocessor 2 Description
CTC2	Move Control Word To Coprocessor 2	CCR[2, n] = Rt
DERET	Return from Debug Exception	PC = DEPC Exit Debug Mode
DI	Disable Interrupts	Rt=Status $\text{Status}_{\text{IE}}$=0
DIV MOD, DIVU MODU	Divide Integers	DIV: Rd = (Divide.signed(Rs div Rt) MOD: Rd = Moduleo.signed(Rs mod Rt) DIVU: Rd = (Divide.unsigned(Rs div Rt) MODU: Rd = Moduleo.unsigned(Rs mod Rt)
EHB	Execution Hazard Barrier	Stall until execution hazards are cleared

Table 10.1 MIPS32 Instruction Set (Continued)

Instruction	Description	Function
EI	Enable Interrupts	Rt=Status $Status_{IE}$=1

ERET	Return from Exception	If SR[2] Then PC = ErrorEPC
		Else
		PC = EPC
		SR[1] = 0
		SR[2] = 0
		LL = 0

ERETNC	Exception Return No Clear	If SR[2] Then PC = ErrorEPC
		Else
		PC = EPC
		SR[1] = 0
		SR[2] = 0

EXT	Extract Bit Field	Rt=ExtractField(Rs,msbd,lsb)
INS	Insert Bit Field	Rt=InsertField(Rt,Rs,msb,lsb)
IRET	Return from Exception	See MCU ASE Instructions
J	Unconditional Jump	PC = PC[31:28] || offset<<2

JAL	Jump and Link	GPR[31] = PC + 8
		PC = PC[31:28] || offset<<2

JALR	Jump and Link Register	Rd = PC + 8
		PC = Rs

JALR.HB	Jump and Link Register with Hazard Barrier	Rd = PC + 8
		PC = Rs
		Stall until all execution and instruction hazards are cleared

JIALC	Jump Indexed and Link Compact	GPR[31] = P C+ 4, PC = (Rt + Sign_extend(offset))
JIC	Jump Indexed Compact	PC = (Rt + Sign_extend(offset))
JR	Jump Register Assembler Idiom	PC = Rs
JR.HB	Jump Register with Hazard Barrier Assembler Idion	PC = Rs Stall until all execution and instruction hazards are cleared
LB	Load Byte	Rt = (byte)Mem[Rs+offset]
LBU	Unsigned Load Byte	Rt = (ubyte))Mem[Rs+offset]
LH	Load Halfword	Rt = (half)Mem[Rs+offset]
LHU	Unsigned Load Halfword	Rt = (uhalf)Mem[Rs+offset]

LL	Load Linked Word	Rt = Mem[Rs+offset]
		LL = 1
		LLAdr = Rs + offset

LSA	Load Scaled Address	Rd = Sign_extend.32((Rs<<(sa+1)) + Rt))
LUI	Load Upper Immediate Assembler Idiom	Rt = immediate << 16
LW	Load Word	Rt = Mem[Rs+offset]
LWC2	Load Word To Coprocessor 2	CPR[2, n, 0] = Mem[Rs+offset]
LWPC	Load Word PC-relative	

Table 10.1 MIPS32 Instruction Set (Continued)

Instruction	Description	Function
MFC0	Move From Coprocessor 0	Rt = CPR[0, n, sel]
MFC2	Move From Coprocessor 2	Rt = CPR[2, n, sel$_{31\ 0}$]
MFHC0	Move From High Coprocessor 0	Rs = Mem(PC+Sign_extend(Offset <<2)
MFHC2	Move From High Word Coprocessor2	Rt= CPR[2,n,sel]$_{63\ 32}$
MTC0	Move To Coprocessor 0	CPR[0, n, sel] = Rt
MTC2	Move To Coprocessor 2	CPR[2, n, sel]$_{31\ 0}$ = Rt
MTCH0	Move To High Word Coprocessor 0	CPR[2, n, sel]$_{63\ 32}$ = Rt
MTHC2	Move To High Word Coprocessor 2	CPR[2, n, sel]$_{63\ 32}$ = Rt
MUL, MUH, MULU, MUHU	Multiply Integers	MUL: Rd = Sign_extend.32(lo_word(multiply.signed(Rs x Rt))) MUH: Rd = Sign_extend.32(hi_word(multiply.signed(Rs x Rt))) MULU: Rd = Sign_extend.32(lo_word(multiply.unsigned(Rs x Rt))) MUHU: Rd = Sign_extend.32(hi_word(multiply.unsigned(Rs x Rt)))
NAL	No-op and Link	GPR[31] = PC + 8
NOP	No Operation (Assembler idiom for: SLL r0, r0, r0)	No Operation
NOR	Logical NOR	Rd = ~(Rs | Rt)
OR	Logical OR	Rd = Rs | Rt
ORI	Logical OR Immediate	Rt = Rs | Immed
PAUSE	Wait For LLBit to Clear	No Operation
PREF	Prefetch	NOP
RDHWR	Read Hardware Register	Rt=HWR[Rd]
RDPGPR	Read GPR from Previous Shadow Set	Rd=SGPR[SRSCtl$_{PSS}$, Rt]
ROTR	Rotate Word Right	Rd = Rt$_{sa-1\ 0}$ || Rt$_{31\ sa}$
ROTRV	Rotate Word Right Variable	Rd = Rt$_{Rs-1\ 0}$ || Rt$_{31\ Rs}$
SB	Store Byte	(byte)Mem[Rs+offset] = Rt

SC	Store Conditional Word	if LL =1
		mem[Rxoffs] = Rt
		Rt = LL

SDBBP	Software Debug Breakpoint	Trap to SW Debug Handler
SEB	Sign Extend Byte	Rd=SignExtend(Rt$_{7\ 0}$)
SEH	Sign Extend Half	Rd=SignExtend(Rt$_{15\ 0}$)
SELEQZ, SELNEZ	Select Integer GPR Value or Zero	SELEQZ: Rd = Rd := Rt = 0?0 Rd:0 SELNEZ: Rd = Rt !:= 0?Rs : 0
SH	Store Halfword	(half)Mem[Rs+offset] = Rt
SIGRIE	Signal Reserved Instruction Exception	SignalException(ReservedInstruction
SLL	Shift Left Logical	Rd = Rt << sa

Table 10.1 MIPS32 Instruction Set (Continued)

Instruction	Description	Function
SLLV	Shift Left Logical Variable	Rd = Rt << Rs[4:0]

SLT	Set on Less Than	if (int)Rs < (int)Rt
		Rd = 1
		else
		Rd = 0

SLTI	Set on Less Than Immediate	if (int)Rs < (int)Immed
		Rt = 1
		else
		Rt = 0

SLTIU	Set on Less Than Immediate Unsigned	if (uns)Rs < (uns)Immed
		Rt = 1
		else
		Rt = 0

SLTU	Set on Less Than Unsigned	if (uns)Rs < (uns)Immed
		Rd = 1
		else
		Rd = 0

SRA	Shift Right Arithmetic	Rd = (int)Rt >> sa
SRAV	Shift Right Arithmetic Variable	Rd = (int)Rt >> Rs[4:0]
SRL	Shift Right Logical	Rd = (uns)Rt >> sa
SRLV	Shift Right Logical Variable	Rd = (uns)Rt >> Rs[4:0]
SSNOP	Superscalar Inhibit No Operation	Nop
SUB	Integer Subtract	Rt = (int)Rs - (int)Rd
SUBU	Unsigned Subtract	Rt = (uns)Rs - (uns)Rd
SW	Store Word	Mem[Rs+offset] = Rt
SWC2	Store Word From Coprocessor 2	Mem[Rs+offset] = CPR[2, n, 0]
SYNC	Synchronize	See SYNC instruction below.
SYNCI	Synchronize Caches to Make Instruction Writes Effective	NOP
SYSCALL	System Call	SystemCallException
TEQ	Trap if Equal	if Rs == Rt TrapException
TGE	Trap if Greater Than or Equal	if (int)Rs >= (int)Rt TrapException
TGEU	Trap if Greater Than or Equal Unsigned	if (uns)Rs >= (uns)Rt TrapException
TLT	Trap if Less Than	if (int)Rs < (int)Rt TrapException
TLTU	Trap if Less Than Unsigned	if (uns)Rs < (uns)Rt TrapException
TNE	Trap if Not Equal	if Rs != Rt TrapException
WAIT	Wait for Interrupts	Stall until interrupt occurs

Table 10.1 MIPS32 Instruction Set (Continued)

Instruction	Description	Function
WRPGPR	Write to GPR in Previous Shadow Set	SGPR[SRSCtl$_{PSS}$,Rd]=Rt
WSBH	Word Swap Bytes within Halfwords	Rd=SwapBytesWithinHalfs(Rt)
XOR	Exclusive OR	Rd = Rs XOR Rt
XORI	Exclusive OR Immediate	Rt = Rs XOR (uns)Immed

Atomically Clear Bit within Byte ACLR

31 26	25 21	20 16	15	14 12	11 4 3 0
REGIMM 000001	base	ATOMIC 00111	0	Bit	offset
6	5	5	1	3	12

Format: `ACLR bit, offset(base)` **MCU ASE**

Purpose: Atomically Clear Bit within Byte

Description: `Disable interrupts; temp ♦ memory[GPR[base] + offset]; temp ♦ (temp and ~(1 << bit)) ; memory[GPR[base] + offset] ♦ temp; Enable Interrupts`

The contents of the 8-bit byte at the memory location specified by the effective address are fetched. The specified bit within the byte is cleared to zero. The modified byte is stored in memory at the location specified by the effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address. The read-modify-write sequence cannot be interrupted.

Transactions with locking semantics occur in some memory interconnects/busses. It is implementation-specific whether this instruction uses such locking transactions.

The execution of this instruction may be disabled by an externally controlled pin, such that attempted execution of this instruction causes a Reserved Instruction exception

Restrictions:

The operation of the processor is **UNDEFINED** if an ACLR instruction is executed in the delay slot of a branch or jump instruction.

Operation:

```
vAddr ♦ sign_extend(offset) + GPR[base]
(pAddr, CCA) ♦ AddressTranslation (vAddr, DATA, STORE)
pAddr ♦ pAddr$_{PSIZE-1..2}$ || (pAddr$_{1..0}$ xor ReverseEndian$^2$)
TempIE ♦ Status$_{IE}$
Status$_{IE}$ ♦ 0
memword ♦ LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ♦ vAddr$_{1..0}$ xor BigEndianCPU$^2$
temp ♦ memword$_{7+8*byte..8*byte}$
temp ♦ temp and (( 1 || 0$^{bit}$) xor 0xFF))
dataword ♦ temp || 0$^{8*byte}$
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
Status$_{IE}$ ♦ TempIE
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error

Programming Notes:

ACLR is treated as a load operation followed by a store operation. All exceptions including TLB, Address Error, Debug Breakpoint, Watch, and Memory Protection exceptions that can be taken on either or both memory access operations must be taken.

31 26	25 21	20 16	15	14 12	11 4 3 0
REGIMM 000001	base	ATOMIC 00111	1	Bit	offset
6	5	5	1	3	12

Format: `ASET bit, offset(base)` **MIPS32 and MCU ASE**

Purpose: Atomically Set Bit within Byte

Description: `Disable interrupts;temp ♦ memory[GPR[base] + offset]; temp ♦ (temp or (1 << bit)) ; memory[GPR[base] + offset] ♦ temp; Enable Interrupts`

The contents of the 8-bit byte at the memory location specified by the effective address are fetched. The specified bit within the byte is set to one. The modified byte is stored in memory at the location specified by the effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address. The read-modify-write sequence cannot be interrupted.

Transactions with locking semantics occur in some memory interconnects/busses. It is implementation-specific whether this instruction uses such locking transactions.

The execution of this instruction may be disabled by an externally controlled pin, such that attempted execution of this instruction causes a Reserved Instruction exception

Restrictions:

The operation of the processor is **UNDEFINED** if an ASET instruction is executed in the delay slot of a branch or jump instruction.

Operation:

```
vAddr ♦ sign_extend(offset) + GPR[base]
(pAddr, CCA) ♦ AddressTranslation (vAddr, DATA, STORE)
pAddr ♦ $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor ReverseEndian$^2$)
TempIE ♦ $Status_{IE}$
$Status_{IE}$ ♦ 0
memword ♦ LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ♦ $vAddr_{1..0}$ xor BigEndianCPU$^2$
temp ♦ $memword_{7+8*byte..8*byte}$
temp ♦ temp or ( 1 || $0^{bit}$)
dataword ♦ temp || $0^{8*byte}$
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
$Status_{IE}$ ♦ TempIE
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error

Programming Notes:

ASET is treated as a load operation followed by a store operation. All exceptions including TLB, Address Error, Debug Breakpoint, Watch, and Memory Protection exceptions that can be taken on either or both memory access operations must be taken.

31 26	25 24	23 21	20 16	15 12	11 0
POOL32B 001000	A0 0	bit	base	ACLR 1011	offset
6	2	3	5	4	12

Format: `ACLR bit, offset(base)` **microMIPS and MCU ASE**

Purpose: Atomically Clear Bit within Byte

Description: `Disable interrupts; temp ♦ memory[GPR[base] + offset]; temp ♦ (temp and ~(1 << bit)) ; memory[GPR[base] + offset] ♦ temp; Enable Interrupts`

The contents of the byte at the memory location specified by the effective address are fetched. The specified bit within the byte is cleared to zero. The modified byte is stored in memory at the location specified by the effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address. The read-modify-write sequence cannot be interrupted.

Transactions with locking semantics occur in some memory interconnects/busses. It is implementation-specific whether this instruction uses such locking transactions.

The execution of this instruction may be disabled by an externally controlled pin, such that attempted execution of this instruction causes a Reserved Instruction exception.

Restrictions:

The operation of the processor is **UNDEFINED** if an ACLR instruction is executed in the delay slot of a branch or jump instruction.

Operation:

```
vAddr ♦ sign_extend(offset) + GPR[base]
(pAddr, CCA) ♦ AddressTranslation (vAddr, DATA, STORE)
pAddr ♦ $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor $ReverseEndian^{2}$)
TempIE ♦ $Status_{IE}$
$Status_{IE}$ ♦ 0
memword ♦ LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ♦ $vAddr_{1..0}$ xor $BigEndianCPU^{2}$
temp ♦ $memword_{7+8*byte..8*byte}$
temp ♦ temp and (( 1 || $0^{bit}$) xor 0xFF))
dataword ♦ temp || $0^{8*byte}$
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
$Status_{IE}$ ♦ TempIE
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error

Programming Notes:

ACLR is treated as a load operation followed by a store operation. All exceptions including TLB, Address Error, Debug Breakpoint, Watch, and Memory Protection exceptions that can be taken on either or both memory access operations must be taken.

31 26	25 24	23 21	20 16	15 12	11 0
POOL32B 001000	A0 0	bit	base	ASET 0011	offset
6	2	3	5	4	12

Format: `ASET bit, offset(base)` **microMIPS AND MCU ASE**

Purpose: Atomically Set Bit within Byte

Description: `Disable interrupts;temp ♦ memory[GPR[base] + offset]; temp ♦ (temp or (1 << bit)) ; memory[GPR[base] + offset] ♦ temp; Enable Interrupts`

The contents of the byte at the memory location specified by the effective address are fetched. The specified bit within the byte is set to one. The modified byte is stored in memory at the location specified by the effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address. The read-modify-write sequence cannot be interrupted.

Transactions with locking semantics occur in some memory interconnects/busses. It is implementation-specific whether this instruction uses such locking transactions.

The execution of this instruction may be disabled by an externally controlled pin, such that attempted execution of this instruction causes a Reserved Instruction exception.

Restrictions:

The operation of the processor is **UNDEFINED** if an ASET instruction is executed in the delay slot of a branch or jump instruction.

Operation:

```
vAddr ♦ sign_extend(offset) + GPR[base]
(pAddr, CCA) ♦ AddressTranslation (vAddr, DATA, STORE)
pAddr ♦ pAddr$_{PSIZE-1..2}$ || (pAddr$_{1..0}$ xor ReverseEndian$^2$)
TempIE ♦ Status$_{IE}$
Status$_{IE}$ ♦ 0
memword ♦ LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ♦ vAddr$_{1..0}$ xor BigEndianCPU$^2$
temp ♦ memword$_{7+8*byte..8*byte}$
temp ♦ temp or ( 1 || 0$^{bit}$)
dataword ♦ temp || 0$^{8*byte}$
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
Status$_{IE}$ ♦ TempIE
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error

Programming Notes:

ASET is treated as a load operation followed by a store operation. All exceptions including TLB, Address Error, Debug Breakpoint, Watch, and Memory Protection exceptions that can be taken on either or both memory access operations must be taken.

31 26	25	6	5 0
CP0 010000	C0 1	0 00 0000 0000 0000 0000	IRET 111000
6	1	20	6

Format: `IRET` **MIPS32 and MCU ASE**

Purpose: Interrupt Return with Automated Interrupt Epilogue

Optionally jump directly to another interrupt vector without returning to original return address.

Description:

IRET is used to automate some of the operations that are required when returning from an interrupt handler. It can be used in place of the ERET instruction at the end of interrupt handlers. The IRET instruction is only appropriate when using Shadow Register Sets and EIC Interrupt mode. The automated operations of this instruction can be used to reverse the effects of the automated operations of the Auto-Prologue feature.

If the EIC mode of interrupts and the Interrupt Chaining feature are used, the IRET instruction can be used to shorten the time between returning from the current interrupt handler and handling the next requested interrupt.

If Automated Prologue feature is disabled, then IRET behaves exactly as ERET.

If either $Status_{ERL}$ or $Status_{BEV}$ bits are set, then IRET behaves exactly as ERET.

Release 6: IRET is only executable in non-user modes. Attempting to execute IRET in user mode will cause a Coprocessor Unusable exception.

If Interrupt Chaining is disabled:

- Interrupts are disabled. CP0 *Status*, *SRSCtl*, and *EPC* registers are restored from the stack. GPR 29 is incremented for the stack frame size. IRET then clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$, and returns to the interrupted instruction pointed by the *EPC* register at the completion of interrupt processing.

If Interrupt Chaining is enabled:

- Interrupts are disabled. CP0 *Status* register is restored from the stack. The priority output of the External Interrupt Controller is compared with the *IPL* field of the *Status* register.

- If $Status_{IPL}$ has a higher priority than that of the External Interrupt Controller value:

  CP0 *SRSCtl* and *EPC* registers are restored from the stack. GPR 29 is incremented for the stack frame size. IRET then clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$, and returns to the interrupted instruction pointed by the *EPC* register at the completion of interrupt processing.

- If $Status_{IPL}$ field has a lower priority than that of the External Interrupt Controller value:

  The value of GPR 29 is first saved to a temporary register then GPR 29 is incremented for the stack frame size. The EIC is signalled that the next pending interrupt has been accepted. This signalling will update the $Cause_{RIPL}$ and $SRSCtl_{EICSS}$ fields from the EIC output values. The $SRSCtl_{EICSS}$ field is copied to the $SRSCtl_{CSS}$ field while the $Cause_{RIPL}$ field is copied to the $Status_{IPL}$ field. The saved temporary register is copied to the GPR 29 of the current SRS. The *KSU*, *ERL* and *EXL* fields of the *Status* register are optionally set to zero. No barrier for execution hazards nor instruction hazards is created. IRET finishes by jumping to the interrupt vector driven by the EIC.

IRET does not execute the next instruction (i.e., it has no delay slot).

Restrictions:

In pre-Release 6, the operation of the processor is **UNDEFINED** if IRET is executed in the delay slot of a branch or jump instruction.

Release 6: Implementations are required to signal a Reserved Instruction Exception if IRET is executed in the delay slot or forbidden slot of a branch or jump instruction.

The operation of the processor is **UNDEFINED** if an IRET is executed when either Shadow Register Sets are not enabled or when EIC interrupt mode is not enabled.

An IRET placed between an LL and SC instruction will always cause the SC to fail.

The effective addresses used for the stack memory transactions must be naturally-aligned. If either of the two least-significant bits of the address is non-zero, an Address Error exception occurs.

IRET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes. The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the IRET returns.

The stack memory transactions behave as individual LW operations with respect to exception reporting. *BadVAddr* would report the faulting address for unaligned access and the faulting word address for unprivileged access, TLB Refill and TLB Invalid exceptions. For TLB exceptions, the faulting word address would be reflected in the *Context*, and *EntryHi* registers. The *CacheError* register would reflect the faulting word address for SRAM Interface Parity or ECC Errors.

Operation:

```
if (( $IntCtl_{APE}$ == 0) | ($Status_{ERL}$ == 1) | ($Status_{BEV}$== 1))
    Act as ERET // read Operation section of ERET description
else
    if (ISAMode)
        EPC ♦ $PC_{31..1}$ || 1 // in case of memory exception
    else
        EPC ♦ PC  // in case of memory exception
    endif
    temp ♦ 0x4 + GPR[29]
    tempStatus ♦ LoadStackWord(temp)
    ClearHazards()
    if ( ($IntCtl_{ICE}$ == 0) | (($IntCtl_{ICE}$ == 1) &
    ($tempStatus_{IPL}$ > $EIC_{RIPL}$)) )
        temp ♦ 0x8 + GPR[29]
        tempSRSCtl ♦ LoadStackWord(temp)
        temp ♦ 0x0 + GPR[29]
        tempEPC ♦ LoadStackWord(temp)
    endif
    Status ♦ tempStatus
    if ( ($IntCtl_{ICE}$ == 0) | (($IntCtl_{ICE}$ == 1) &
        ($tempStatus_{IPL}$ > $EIC_{RIPL}$)) )
        GPR[29] ♦ GPR[29] + DecodedValue($IntCtl_{StkDec}$)
        SRSCtl ♦ tempSRSCtl
        EPC ♦ tempEPC
        temp ♦ EPC
        $Status_{EXL}$ ♦ 0
        if (ArchitectureRevision □ 2) and ($SRSCtl_{HSS}$ > 0)
        and ($Status_{BEV}$ = 0) then
            $SRSCtl_{CSS}$ ♦ $SRSCtl_{PSS}$
        endif
```

```
      if IsMicroMIPSImplemented() then
         PC ♦ $temp_{31..1}$ || 0
         ISAMode ♦ $temp_0$
      else
         PC ♦ temp
      endif
      LLbit ♦ 0
      $Cause_{IC}$ ♦ 0
      ClearHazards()
   else
      Signal_EIC_for_Next_Interrupt()
      (wait for EIC outputs to update)
      $Cause_{RIPL}$ ♦ $EIC_{RIPL}$
      $SRSCtl_{EICSS}$ ♦ $EIC_{SS}$
      temp29 ♦ GPR[29]
      GPR[29] ♦ GPR[29] + DecodedValue($IntCtl_{StkDec}$)
      $Status_{IPL}$ ♦ $Cause_{RIPL}$
      $SRSCtl_{CSS}$ ♦ $SRSCtl_{EICSS}$
      NewShadowSet ♦ $SRSCtl_{EICSS}$
      GPR[29] ♦ temp29
      if ($IntCtl_{ClrEXL}$ == 1)
         $Status_{EXL}$ ♦ 0
         $Status_{KSU}$ ♦ 0
      endif
      $Cause_{IC}$ ♦ 1
      ClearHazards()
      PC ♦ CalcIntrptAddress()
   endif
endif


function LoadStackWord(vaddr)
   if $vAddr_{1..0} \neq 0^2$ then
      SignalException(AddressError)
   endif
   (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
   memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
   LoadStackWord ← memword
endfunction LoadStackWord


function CalcIntrptAddress()
   if $Status_{BEV}$ = 1
      vectorBase ←0xBFC0.0200
   else
      if ( ArchitectureRevision □ 2)
         vectorBase ←$EBase_{31..12} \| 0^{11}$)
      else
         vectorBase ←0x8000.0000
      endif
   endif
   if ($Cause_{IV}$ = 0)
      vectorOffset = 0x180
   else
      if ($Status_{BEV}$ = 1) or ($IntCtl_{VS}$ = 0)
         vectorOffset = 0x200
      else
```

```
         if ( $Config3_{VEIC}$ = 1 and EIC_Option=1)
            VectorNum = $Cause_{RIPL}$
         elseif ($Config3_{VEIC}$ = 1 and EIC_Option=2)
            VectorNum = EIC_VectorNum
         elseif ($Config3_{VEIC}$ = 0 )
            VectorNum = VIntPriorityEncoder()
         endif
         if ($Config3_{VEIC}$ = 1 and EIC_Option=3)
            vectorOffset = EIC_VectorOffset
         else
            vectorOffset = 0x200 + (VectorNum x ($IntCtl_{VS} \| 0^5$))
         endif
      endif

      endif
   CalcIntrptAddress = vectorBase | vectorOffset
endfunction CalcIntrptAddress
```

Exceptions:

Coprocessor Unusable Exception, Reserved Instruction Exception, TLB Refill, TLB Invalid, Address Error, SRAM Interface Parity or ECC Error, Bus Error Exceptions

31 26	25 6	5 0
POOL32A 000000	000 0000 0011 0100 1101	POOL32AXf 111100
6	20	6

Format: `IRET` **microMIPS and MCU ASE**

Purpose: Interrupt Return with Automated Interrupt Epilogue

Optionally jump directly to another interrupt vector without returning to original return address.

Description:

IRET automates some of the operations that are required when returning from an interrupt handler and can be used in place of the ERET instruction at the end of interrupt handlers. IRET is only appropriate when using Shadow Register Sets and the EIC Interrupt mode. The automated operations of this instruction can be used to reverse the effects of the automated operations of the Auto-Prologue feature.

If the EIC interrupt mode and the Interrupt Chaining feature are used, the IRET instruction can be used to shorten the time between returning from the current interrupt handler and handling the next requested interrupt.

If the Automated Prologue feature is disabled, then IRET behaves exactly like ERET.

If either the Status_{ERL} or Status_{BEV} bits are set, then IRET behaves exactly like ERET.

Release 6: IRET is only executable in non-user modes. Attempting to execute IRET in user mode will cause a Coprocessor Unusable exception.

If Interrupt Chaining is disabled:

- Interrupts are disabled. CP0 *Status*, *SRSCtl*, and *EPC* registers are restored from the stack. GPR 29 is incremented for the stack frame size. IRET then clears execution and instruction hazards, conditionally restores SRSCtl_{CSS} from SRSCtl_{PSS}, and returns at the completion of interrupt processing to the interrupted instruction pointed to by the *EPC* register.

If Interrupt Chaining is enabled:

- Interrupts are disabled. CP0 *Status* register is restored from the stack. The priority output of the External Interrupt Controller is compared with the IPL field of the *Status* register.
- If Status_{IPL} has a higher priority than the External Interrupt Controller value:

  CP0 *SRSCtl* and *EPC* registers are restored from the stack. GPR 29 is incremented for the stack frame size. IRET then clears execution and instruction hazards, conditionally restores SRSCtl_{CSS} from SRSCtl_{PSS}, and returns to the interrupted instruction pointed to by the *EPC* register at the completion of interrupt processing.

- If Status_{IPL} has a lower priority than the External Interrupt Controller value:

  The value of GPR 29 is first saved to a temporary register and then GPR 29 is incremented for the stack frame size. The EIC is signalled that the next pending interrupt has been accepted. This signalling will update the Cause_{RIPL} and SRSCtl_{EICSS} fields from the EIC output values. The SRSCtl_{EICSS} field is copied to the SRSCtl_{CSS} field, while the Cause_{RIPL} field is copied to the Status_{IPL} field. The saved temporary register is copied to the GPR 29 of the current SRS. The KSU and EXL fields of the *Status* register are optionally set to zero. No barrier for execution hazards or instruction hazards is created. IRET finishes by jumping to the interrupt vector driven by the EIC.

IRET does not execute the next instruction (i.e., it has no delay slot).

Restrictions:

In pre-Release 6, the operation of the processor is **UNDEFINED** if IRET is executed in the delay slot of a branch or jump instruction.

Release 6: Implementations are required to signal a Reserved Instruction Exception if IRET is executed in the delay slot or forbidden slot of a branch or jump instruction.

The operation of the processor is **UNDEFINED** if IRET is executed when either Shadow Register Sets are not enabled, or the EIC interrupt mode is not enabled.

An IRET placed between an LL and SC instruction will always cause the SC to fail.

The effective addresses used for stack transactions must be naturally-aligned. If either of the two least-significant bits of the address is non-zero, an Address Error exception occurs.

IRET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier begin with the instruction fetch and decode of the instruction at the PC to which the IRET returns.

The stack transactions behave as individual LW operations with respect to exception reporting. BadVAddr would report the faulting address for an unaligned access, and the faulting word address for unprivileged access, TLB Refill, and TLB Invalid exceptions. For TLB exceptions, the faulting word address would be reflected in the *Context* and *EntryHi* registers. The *CacheError* register would reflect the faulting word address for SRAMs.

Operation:

```
if (( IntCtl$_{APE}$ == 0) | (Status$_{ERL}$ == 1) | (Status$_{BEV}$== 1))
    Act as ERET // read Operation section of ERET description
else
    if (ISAMode)
        EPC ♦ PC$_{31..1}$ || 1 // in case of memory exception
    else
        EPC ♦ PC  // in case of memory exception
    endif
    temp ♦ 0x4 + GPR[29]
    tempStatus ♦ LoadStackWord(temp)
    ClearHazards()
    if ( (IntCtl$_{ICE}$ == 0) |  ((IntCtl$_{ICE}$ == 1) &
    (tempStatus$_{IPL}$ > EIC$_{RIPL}$)) )
        temp ♦ 0x8 + GPR[29]
        tempSRSCtl ♦ LoadStackWord(temp)
        temp ♦ 0x0 + GPR[29]
        tempEPC ♦ LoadStackWord(temp)
    endif
    Status ♦ tempStatus
    if ( (IntCtl$_{ICE}$ == 0) | ((IntCtl$_{ICE}$ == 1) &
        (tempStatus$_{IPL}$ > EIC$_{RIPL}$)) )
        GPR[29] ♦ GPR[29] + DecodedValue(IntCtl$_{StkDec}$)
        SRSCtl ♦ tempSRSCtl
        EPC ♦ tempEPC
        temp ♦ EPC
        Status$_{EXL}$ ♦ 0
        if (ArchitectureRevision □ 2) and (SRSCtl$_{HSS}$ > 0) and (Status$_{BEV}$ = 0) then
            SRSCtl$_{CSS}$ ♦ SRSCtl$_{PSS}$
        endif
        if IsMicroMIPSImplemented() then
```

```
         PC ♦ $temp_{31..1}$ || 0
         ISAMode ♦ $temp_0$
      else
         PC ♦ temp
      endif
      LLbit ♦ 0
      $Cause_{IC}$ ♦ 0
      ClearHazards()
   else
      Signal_EIC_for_Next_Interrupt()
      (wait for EIC outputs to update)
      $Cause_{RIPL}$ ♦ $EIC_{RIPL}$
      $SRSCtl_{EICSS}$ ♦ $EIC_{SS}$
      temp29 ♦ GPR[29]
      GPR[29] ♦ GPR[29] + DecodedValue($IntCtl_{StkDec}$)
      $Status_{IPL}$ ♦ $Cause_{RIPL}$
      $SRSCtl_{CSS}$ ♦ $SRSCtl_{EICSS}$
      NewShadowSet ♦ $SRSCtl_{EICSS}$
      GPR[29] ♦ temp29
      if ($IntCtl_{ClrEXL}$ == 1)
         $Status_{EXL}$ ♦ 0
         $Status_{KSU}$ ♦ 0
      endif
      $Cause_{IC}$ ♦ 1
      ClearHazards()
      PC ♦ CalcIntrptAddress()
   endif
endif


function LoadStackWord(vaddr)
   if $vAddr_{1..0} \neq 0^2$ then
      SignalException(AddressError)
   endif
   (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
   memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
   LoadStackWord ← memword
endfunction LoadStackWord


function CalcIntrptAddress()
   if StatusBEV = 1
      vectorBase ←0xBFC0.0200
   else
      if ( ArchitectureRevision □ 2)
         vectorBase ←$EBase_{31..12} \| 0^{11}$)
      else
         vectorBase ←0x8000.0000
      endif
   endif
   if ($Cause_{IV}$ = 0)
      vectorOffset = 0x180
   else
      if ($Status_{BEV}$ = 1) or ($IntCtl_{VS}$ = 0)
         vectorOffset = 0x200
      else
         if ( $Config3_{VEIC}$ = 1 and EIC_Option=1)
```

```
            VectorNum = $\text{Cause}_{\text{RIPL}}$
         elseif ($\text{Config3}_{\text{VEIC}}$ = 1 and EIC_Option=2)
            VectorNum = EIC_VectorNum
         elseif ($\text{Config3}_{\text{VEIC}}$ = 0 )
            VectorNum = VIntPriorityEncoder()
         endif
         if ($\text{Config3}_{\text{VEIC}}$ = 1 and EIC_Option=3)
            vectorOffset = EIC_VectorOffset
         else
            vectorOffset = 0x200 + (VectorNum x ($\text{IntCtl}_{\text{VS}} \| 0^5$))
         endif
      endif
   endif
   CalcIntrptAddress = vectorBase | vectorOffset
endfunction CalcIntrptAddress
```

Exceptions:

Coprocessor Unusable Exception, Reserved Instruction Exception, TLB Refill, TLB Invalid, Address Error, SRAM Interface Parity or ECC Error, Bus Error Exceptions

31 26	25 21	20 16	15 7	6	5 0
SPECIAL 011111	base	rt	offset	0	LL 110110
6	5	5	16		

`LL rt, offset(base)` **MIPS32**

Purpose: Load Linked Word

To load a word from memory for an atomic read-modify-write

Description: `GPR[rt] ♦ memory[GPR[base] + offset]`

The LL and SC instructions provide the primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and written into GPR *rt*. The 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

This begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor. When an LL is executed it starts an active RMW sequence replacing any other sequence that was active. The RMW sequence is completed by a subsequent SC instruction that either completes the RMW sequence atomically and succeeds, or does not and fails.

Executing LL on one processor does not cause an action that, by itself, causes an SC for the same block to fail on another processor.

An execution of LL does not have to be followed by execution of SC; a program is free to abandon the RMW sequence without attempting a write.

Restrictions:

The addressed location must be synchronizable by all processors and I/O devices sharing the location; if it is not, the result in **UNPREDICTABLE**. Which storage is synchronizable is a function of both CPU and system implementations. See the documentation of the SC instruction for the formal definition. The addressed location may be uncached for the M6200 core.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

Release 6 requires that systems provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an Address Error exception on a misaligned access is required.

Operation:

```
vAddr ♦ sign_extend(offset) + GPR[base]
if $vAddr_{1..0}$ … $0^2$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ♦ AddressTranslation (vAddr, DATA, LOAD)
memword ♦ LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ♦ memword
LLbit ♦ 1
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Reserved Instruction

Programming Notes:

There is no Load Linked Word Unsigned operation corresponding to Load Word Unsigned.

Implementation Notes:

An LL on one processor must not take action that, by itself, causes an SC for the same block on another processor to fail. If an implementation depends on retaining the data in the cache during the RMW sequence, cache misses caused by LL must not fetch data in the *exclusive* state, which removes it from the cache if it were present in another cache.

31 26	25 21 20 16 15 11	10 6	5 0
SPECIAL 000000	0 00 0000 0000 0000 0	stype	SYNC 001111
6	15	5	6

`SYNC (stype = 0 implied)` **MIPS32**

Purpose: Synchronize Shared Memory

To order loads and stores.

Description:

Simple Description:

- SYNC affects only *uncached* and *cached coherent* loads and stores. The loads and stores that occur before the SYNC must be completed before the loads and stores after the SYNC are allowed to start.

- Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.

- SYNC is required, potentially in conjunction with EHB, to guarantee that memory reference results are visible across operating mode changes. For example, a SYNC is required on entry to and exit from Debug Mode to guarantee that memory affects are handled correctly.

Detailed Description:

- SYNC does not guarantee the order in which instruction fetches are performed. The *stype* values 1-31 are reserved for future extensions to the architecture. A value of zero will always be defined such that it performs all defined synchronization operations. Non-zero values may be defined to remove some synchronization operations. As such, software should never use a non-zero value of the *stype* field, as this may inadvertently cause future failures if non-zero values remove synchronization operations.

- The SYNC instruction is externalized on the SRAM interface of the M6200 core. External logic can use this information in a system-dependent manner to enforce memory ordering between various memory elements in the system.

Restrictions:

The effect of SYNC on the global order of loads and stores for memory access types other than *uncached* and *cached coherent* is **UNPREDICTABLE**.

Operation:

```
SyncOperation(stype)
```

Exceptions:

None

31 26	25 21	20 16	15 7	6	5 0
SPRCIAL3 0111111	base	rt	offset	0	SC 100110
6	5	5	16	1	6

`SC rt, offset(base)` **MIPS32**

Purpose: Store Conditional Word

To store a word to memory to complete an atomic read-modify-write

Description: `if atomic_update then memory[GPR[base] + offset] ♦ GPR[rt], GPR[rt] ♦ 1 else GPR[rt] ♦ 0`

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The32-bit word in GPR *rt* is conditionally stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The SC completes the RMW sequence begun by the preceding LL instruction executed on the processor. To complete the RMW sequence atomically, the following occur:

- The32-bit word of GPR *rt* is stored into memory at the location specified by the aligned effective address.
- A 1, indicating success, is written into GPR *rt*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *rt*. On the M6200 core, the SRAM interface supports a lock protocol and the success or failure can be indicated by external hardware.

If the following event occurs between the execution of LL and SC, the SC fails:

- An ERET instruction is executed.

If either of the following events occurs between the execution of LL and SC, the SC may succeed or it may fail; the success or failure is not predictable. Portable programs should not cause one of these events.

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LL/SC.
- The instructions executed starting with the LL and ending with the SC do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

The following conditions must be true or the result of the SC is **UNPREDICTABLE**:

- Execution of SC must have been preceded by execution of an LL instruction.
- An RMW sequence executed without intervening events that would cause the SC to fail must use the same address in the LL and SC. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (Address Error) on a misaligned access is required.

Operation:

```
vAddr ♦ sign_extend(offset) + GPR[base]
if $vAddr_{1..0}$ … $0^2$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ♦ AddressTranslation (vAddr, DATA, STORE)
dataword ♦ GPR[rt]
if LLbit then
   StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
endif
GPR[rt] ♦ $0^{31}$ || LLbit
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error

Programming Notes:

LL and SC are used to atomically update memory locations, as shown below.

```
L1:
   LL     T1, (T0)  # load counter
   ADDI   T2, T1, 1 # increment
   SC     T2, (T0)  # try to store, checking for atomicity
   BEQ    T2, 0, L1 # if not atomic (0), try again
   NOP              # branch-delay slot
```

Exceptions between the LL and SC cause SC to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LL and SC function on a single processor for *cached noncoherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.

Implementation Notes:

The block of memory that is locked for LL/SC is typically the largest cache line in use.

31 26	25	24 6	5 0
CP0 010000	CO 1	Implementation-Dependent Code	WAIT 100000
6	1	19	6

`WAIT` **MIPS32**

Purpose: Enter Standby Mode

Wait for Event

Description:

The WAIT instruction forces the core into low power mode. The pipeline is stalled and when all external requests are completed, the processor's main clock is sto pped. The p rocessor will restart when rese t (*SI_WarmResetN* or *SI_ColdResetN*) is signaled, or a non-masked interrupt is taken (*SI_NMI*, *SI_Int*, or *EJ_DINT*). Note that theM6200 core does not use the code field in this instruction.

If the pipeline restarts as the result of an enabled interrupt, that interrupt is taken between the WAIT instruction and the following instruction (*EPC* for the interrupt points at the instruction following the WAIT instruction).

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Release 6 implementations are required to signal a Reserved Instruction Exception if a WAIT instruction is encountered in the delay slot or forbidden slot of a branch or jump instruction.

Operation:

```
I:  if IsCoprocessorEnabled(0) then
       Enter lower power mode
    else
       SignalException(CoprocessorUnusable, 0)
    endif
I+1:/* Potential interrupt taken here */
```

Exceptions:

Coprocessor Unusable Exception

Chapter 11

microMIPS32™ Instruction Set Architecture

The M6200 core supports the microMIPS32 ISA, which contains all MIPS32 ISA instructions in a new 32-bit encoding scheme, with some of the most commonly used instructions also available in 16-bit encoded format. This ISA improves code density through the additional 16-bit instructions, while maintaining a performance similar to MIPS32 mode. In microMIPS mode, 16-bit or 32-bit instructions will be fetched and recoded to legacy MIPS32 instruction opcodes in the pipeline’s I stage, so that the M6200 core can have the same microarchitecture. Because the microMIPS instruction stream can be intermixed with 16-bit halfword or 32-bit word size instructions on halfword or word boundaries, additional logic is in place to address the word misalignment issues, thus minimizing performance loss.

The microMIPS™ architecture minimizes the code footprint of applications, thus reducing the cost of memory, which is particularly high for embedded memory. Customers can generate best results without spending time to profile its application. The smaller code footprint typically leads to reduced power consumption per executed task because of the smaller number of memory accesses.

11.1 ISA Modes

The ISA mode in which the processor is executing is determined by the single-bit ISA Mode register. An ISA Mode bit value of zero selects MIPS32, and a value of 1 selects microMIPS32 mode. The ISA Mode value is not directly visible to user software. Its value is automatically saved to any GPR used as a jump target address, such as GPR31 when written by a JAL instruction, and saved in the source register of a jump instruction.

The ISA mode following reset is determined by the setting of the *Config3*$_{ISA}$ register field, which is a read-only field set by a hardware signal external to the processor core.

The ISA mode of an exception handler is determined by the setting of the *Config3*$_{ISAOnExc}$ register field (bit 16). The *Config3*$_{ISAOnExc}$ register field is writable by software and has a reset value that is set by a hardware signal external to the processor core. This register field allows privileged software to change the ISA mode to be used for subsequent exceptions. All exception types whose vectors are offsets of the *EBase* register have this capability.

The selected ISA mode of a debug exception is determined by the setting of the ISAonDebug register field in the OCI CONTROL Register. This register field is writable by probe software and has a reset value that is set by a hardware signal external to the processor core.

11.2 Mode Switch

Mode switching between MIPS and microMIPS uses the Jump-and-Link-Register and Jump-Register instructions. These instructions interpret bit 0 of the source registers as the target ISA mode and set the ISA Mode bit to its contents.

When exceptions or interrupts occur, the ISA Mode bit is saved in bit 0 of the *EPC*, *DEPC*, or *ErrorEPC*. The ISA Mode bit is then set according to the *Config3*$_{ISA}$ register field. On return from an exception, the processor loads the saved ISA Mode bit from either *EPC*, *DEPC*, or *ErrorEPC*.

11.3 microMIPS Instructions

The reader is referred to the *MIPS® Architecture Reference Manual Volume II-B: microMIPS32™ Instruction Set* [11] for the complete description of all microMIPS instructions.

Appendix A

References

This appendix lists other publications available from MIPS Technologies, Inc. that are referenced in this document. These documents may be included in the `$MIPS_PROJECT/doc` area of a typical M6200 soft or hard core release, or in some cases may be available on the MIPS web site http://www.mips.com.

1. MIPS32® M6200 Processor Core Family Data Sheet
   MIPS Document: MD01092

2. MIPS32® M6200 Processor Core Family Integrator's Guide
   MIPS Document: MD01094

3. MIPS32® M6200 Processor Core Family Getting Started Guide
   MIPS Document: MD01110

4. Security Features of the M6200 Processor Core Family
   MIPS Document: MD01058

5. MIPS32® Interrupt Controller User's Guide
   MIPS Document: MD01146

6. MIPS® Debug Hub Technical Reference Manual
   MIPS Document: MD01070

7. MIPS® iFlowtrace™ Architecture Specification
   MIPS Document: MD00526

8. MIPS® Architecture For Programmers, Volume I: Introduction to the MIPS32® Architecture
   MIPS Document: MD0082

9. MIPS® Architecture For Programmers, Volume I: Introduction to the microMIPS32™ Architecture
   MIPS Document: MD0741

10. MIPS® Architecture For Programmers, Volume II: The MIPS32® Instruction Set
    MIPS Document: MD0086

11. MIPS® Architecture For Programmers, Volume II: The microMIPS32™ Instruction Set
    MIPS Document: MD0582

12. MIPS® Architecture For Programmers Volume III: The MIPS32® and microMIPS32™ Privileged Resource Architecture
    MIPS Document: MD00090

13. MIPS® Architecture for Programmers Volume IV-h: The MCU Application-Specific Extension to the MIPS32® Architectures
    MIPS Document: MD00834

14. MIPS® Architecture for Programmers Volume IV-h: The MCU Application-Specific Extension to the microMIPS32™ Architectures
    MIPS Document: MD00838

15. MIPS® Architecture Reference Manual Volume IV-e: The MIPS® DSP Module of the MIPS32® Architecture
    MIPS Document: MD00372

16. MIPS® Architecture Reference Manual Volume IV-e: The MIPS® DSP Module of the microMIPS32® Architecture
    MIPS Document: MD00762

17. Five Methods of Utilizing the MIPS® DSP Module
    MIPS Document: MD00783

18. Efficient DSP Module Programming in C: Tips and Tricks
    MIPS Document: MD00485

Appendix B

Revision History

Change bars in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

This document may refer to Architecture specifications (for example, instruction set descriptions), and change bars in these sections indicate changes since the previous version of the relevant Architecture document.

.

Revision	Date	Description
00.01	March 30, 2015	• Preliminary version for RC 0.0
01.00	January 12, 2016	• Release of document for RC 1.0