Load Scheduling

When load scheduling is enabled, the BIU determines whether a load is implemented immediately or is delayed, if the load data is not yet required. The LR333x0 does not stall in the MEM stage of the load operation unless there is a data dependency in the CPU pipeline. The BIU initiates the fetch and signals the CPU to latch the data when it appears on the data bus. The CPU then updates the register file if the load instruction has reached its WB stage without being cancelled.

The LR333x0 supports a single scheduled load. The LR333x0 does not wait for the load operation's data fetch to complete unless there are any SW or LW instructions in the ALU stage or if there is a data dependency in the RD or ALU stage.

Load scheduling saves at least one cycle as shown in Figure 3.7.

Figure 3.7
Load Scheduling Example

```
#Four-clock memory system
#load non-cacheable word (ID stage)
#without load scheduling

lw r1, (r3)

.
.

stall
stall
stall
stall
stall      #fixup cycle

.
.

addiu r1, constant

.
.
```

```
#load non-cacheable word (ID stage)
#with load scheduling

lw r1, (r3)

.
.
.

stall      #writeback to register file
stall      #fixup cycle

.
.

addiu r1, constant

.
.
```

The LDSCH bit in the BIU/Cache Control Register determines whether load scheduling is enabled or not. Refer to Section 6.2, "BIU/Cache Configuration Register," for more information.

Chapter 4
Exception Processing

This chapter describes the LR333x0's exception handling mechanisms, the system control coprocessor (CP0) registers, and all events that cause exceptions. This chapter includes the following sections:

- Exception Handling Operation
- Exception Handling Registers
- Exception Description Details
- Interrupt Exception Handling

The LR333x0 is always in one of two operating modes: *normal* or *exception*. In the normal operating mode, the LR333x0 executes the program-specified sequence of instructions. In the exception mode, the normal sequence of instruction execution is suspended to allow the LR333x0 to respond to abnormal or asynchronous events. The LR333x0's exception-handling system efficiently manages machine exceptions, including arithmetic overflows, I/O interrupts, and system calls.

Table 4.1 lists the exceptions that the LR333x0 recognizes.

Table 4.1
LR333x0 Exceptions

Exception	*Mnemonic*	*Cause*
Reset	Reset	Deassertion of the LR333x0's $\overline{\text{RESET}}$ signal causes an exception that transfers control to the reset vector.
Bus Error	IBE (instr.) DBE (data)	Assertion of the LR333x0's $\overline{\text{BERR}}$ signal during an instruction fetch (IBE) or data load or store (DBE).
Address Error	AdEL (load) AdES (store)	Attempt to load, fetch, or store an unaligned word—that is, a word or halfword at an address not evenly divisible by four or two, respectively. Also caused by reference to an address with most-significant bit set while in user mode.
Overflow	Ovf	Two's complement overflow during add or subtract.
System Call	Sys	Execution of the SYSCALL instruction.
Breakpoint	Bp	Execution of the BREAK instruction.
Reserved Instruction	RI	Execution of an instruction with an undefined or reserved major operation code (bits 31:26), or a SPECIAL instruction whose minor opcode (bits 5:0) is undefined.
Coprocessor Unusable	CpU	Execution of a coprocessor instruction where the CU (Coprocessor Usable) bit is not set for the target coprocessor.
Interrupt	Int	Assertion of one of the LR333x0's six hardware interrupt inputs (including Counter/Timers), or setting one of the two software interrupt bits in the Cause Register. Interrupts must be enabled.
Debug	Debug	Detection of a program counter breakpoint, data address breakpoint, or trace condition.

4.1 Exception Handling Operation

When an exception occurs, the LR333x0 aborts the current instruction and all instructions following in the pipeline that have already begun execution. The occurrence of the exception puts the system in kernel mode. The LR333x0 jumps directly into a designated exception-handler routine. The LR333x0 loads the Exception Program Counter (EPC) Register with an appropriate restart location where execution may resume after the exception is serviced. The restart location in the EPC is the address of the instruction that caused the exception or, if the instruction was executing in a branch delay slot, the address of the branch instruction immediately preceding the delay slot.

Even though the processor is pipelined, exceptions are reported synchronously, so that all exceptions for an instruction are reported prior to exceptions for successive instructions. The characteristics of the machine's pipeline staging, however, cannot guarantee that all processor and associated system states will remain completely unchanged as a result of the (possibly incomplete) execution of the instruction immediately following an instruction that causes an exception. Examples of these state changes include:

- Instructions may have been read from memory and loaded into the I-Cache.
- The multiply/divide register may have been altered by a MULT/MULTU, DIV/DIVU, or MTHI/MTLO instruction.
- The cache may have been updated in response to a bus error on a cacheable, memory write operation.

The above events can normally be ignored because enough of the state of the machine is restored so that execution always properly resumes after servicing the exception. In the case of the multiply/divide and CP0 registers, so called "reorganization constraints" prevent the use of instructions in an organization for which the hardware cannot guarantee interruptibility.

4.2 Exception Handling Registers

CP0 contains eleven registers that relate to exception processing:

- Status
- Cause
- Bad Address (BadA)
- Target Address (TAR)
- Exception Program Counter (EPC)
- Processor Revision Identifier (PRId)
- Debug and Cache Invalidate Control (DCIC)
- Breakpoint Program Counter (BPC)
- Breakpoint Program Counter Mask (BPCM)
- Breakpoint Data Address (BDA)
- Breakpoint Data Address Mask (BDAM)

During exception processing, software can examine these registers to determine the cause of an exception and the state of the CPU at the time of an exception.

The exception-processing registers are numbered as shown in Table 4.2. Each of these registers is described in detail in the pages that follow.

Table 4.2 Exception-Processing Register Numbering

Number	*Mnemonic*	*Name*
3	BPC	Breakpoint Program Counter
5	BDA	Breakpoint Data Address
6	TAR	Target Address
7	DCIC	Debug and Cache Invalidate Control
8	BadA	Bad Address (read only)
9	BDAM	Breakpoint Data Address Mask
11	BPCM	Breakpoint Program Counter Mask
12	SR	Status
13	Cause	Cause
14	EPC	Exception Program Counter
15	PRId	Processor Revision Identifier

Compatibility Note

The LR333x0 does not implement the CP0 registers that relate to the R3000's Translation Lookaside Buffer (TLB): EntryHi (10), EntryLo (2), Index (0), Random (1), and Context Registers (4). Attempts to access these registers cause Reserved Instruction (RI) Exceptions.

The LR333x0 treats accesses to unused CP0 registers 16 through 31 as NOPs.

When in user mode with Cu0 set to zero, the LR333x0 takes a Reserved Instruction (RI) Exception if it decodes a RFE, MTC0, or MFC0 instruction. In like conditions, the R3000 takes a Coprocessor Unusable Exception (CpU).

Status Register

The Status Register contains all major status bits for exception conditions. All bits in the Status Register, with the exception of the TS (TLB Shutdown) bit, are readable and writable; the TS bit is read-only. The format of the 32-bit Status Register is shown below. Additional details on the function of each Status Register bit are provided in the paragraphs that follow.

Location: CP0
Address: 12

Cold Reset Initial Value: 0x0040.0000
Warm Reset Initial Value: 0x0040.0000

31 28	27 23	22	21	20	19	18	17	16	15 10	9 8	7 6	5	4	3	2	1	0
Cu[3:0]	R	BEV	TS	PE	R	PZ	R	IsC	Intr[5:0]	Sw[1:0]	R	KUo	IEo	KUp	IEp	KUc	IEc

Cu[3:0]

Coprocessor Usability [3:0] **[31:28]**
Software sets Cu[3:0] to one to indicate that the associated coprocessor is usable. Bit 31 corresponds to Coprocessor 3 and bit 28 corresponds to Coprocessor 0. Because the LR333x0 does not support external coprocessors, Cu[3:1] should be set to zero, unless you intend to use the BCzF or BCzT instructions to test the CPC[3:0] signals. When Cu[3:1] are zero, a coprocessor instruction causes a Coprocessor Unusable Exception (CpU). Note that the system control coprocessor (CP0) is always considered usable when the LR333x0 is operating in kernel mode, regardless of the setting of the Cu0 bit.

R

Reserved **[27: 23], 19, 17, [7:6]**
These bits are reserved and read as zero. The LR333x0 ignores attempts to set these bits; however, software should write these bits as zero to ensure compatibility with future versions of hardware.

BEV

Bootstrap Exception Vectors **22**
The BEV bit controls the location of general exception vectors during bootstrap (immediately following reset). When this bit is set to zero, the normal exception vectors are used; when the bit is set to one, bootstrap vector locations are used.

BEV Set to Zero – The debug exception vector is located at 0x8000.0040, and the general exception vector is located at 0x8000.0080.

BEV Set to One – The debug exception vector is relocated to an address of 0xBFC0.0140, and the general exception vector is relocated to 0xBFC0.0180. This alternate set of vectors can be used when diagnostic tests cause exceptions to occur prior to verification of proper operation of the cache and main memory system. The LR333x0 sets this bit to one upon deassertion of $\overline{\text{RESET}}$. (Refer to Section 4.3, "Exception Description Details," later in this chapter for a description of the exception vectors.)

TS

TLB Shutdown **21**
In the LR333x0, this bit is permanently set to zero. This bit is read-only.

PE

Parity Error **20**
The LR333x0 sets the PE bit to one if it detects an external memory parity error. Software may use the PE bit to log external memory parity errors. The parity error condition is reset by

writing a one into PE; writing a zero into this bit does not affect its value.

PZ — **Parity Zero** — **18**

Software sets PZ to one to force the LR333x0 to generate zeros on DP[3:0] during store transactions.

IsC — **Isolate Cache** — **16**

When this bit is set to one, store operations do not propagate through the LR333x0 to the external memory system. This bit must be set to one for cache testing. Refer to Section 6.4, "Cache Maintenance and Testing," for more information.

Intr[5:0] — **Hardware Interrupt Mask [5:0]** — **[15:10]**

Software sets these six bits to one to enable the corresponding hardware interrupts. Bit 15 corresponds to INT5, and bit 10 corresponds to INT0. All interrupts can be disabled by clearing the Interrupt Enable bits (IEo/IEp/IEc) described below.

Sw[1:0] — **Software Interrupt Mask [1:0]** — **[9:8]**

Software sets these two bits to one to enable the corresponding software interrupts. Bit 9 corresponds to Sw1, and bit 8 corresponds to Sw0. All interrupts can be disabled by clearing the Interrupt Enable bits (IEo/IEp/IEc) described below.

KUo, p, c — **Kernel-User Mode, Old/Previous/Current** — **5, 3, 1**

The KUo, KUp, and KUc bits comprise a three-level stack showing the old/previous/current mode (0 means kernel; 1 means user). The occurrence of an exception automatically puts the system in kernel mode. Manipulation and use of these bits during exception processing is described in the following section.

IEo, p, c — **Interrupt Enable, Old/Previous/Current** — **4, 2, 0**

The IEo, IEp, and IEc bits comprise a three-level stack showing the old/ previous/current interrupt enable settings (0 means disabled; 1 means enabled). Manipulation and use of these bits during exception processing is described in the following section.

Status Register Mode Bits and Exception Processing

When the LR333x0 responds to an exception, it saves the *current* kernel/user mode (KUc) and *current* interrupt enable mode (IEc) bits of the Status Register into the *previous* mode bits (KUp and IEp). The *previous* mode bits (KUp and IEp) are saved into the old mode bits (KUo and IEo). The current mode bits (KUc and IEc) are cleared to cause the processor to enter the kernel operating mode and to disable all interrupts.

This three-level set of mode bits lets the LR333x0 respond to two levels of exceptions before software must save the contents of the Status Register. Figure 4.1 shows how the LR333x0 manipulates the Status Register during exception recognition.

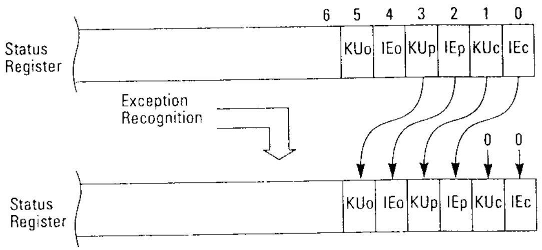


Figure 4.1
The Status Register and Exception Recognition

After an exception handler has completed execution, the LR333x0 must return to the system context that existed prior to the exception (if possible). The Restore From Exception (RFE) instruction provides the mechanism for this return.

The RFE instruction restores control to a process that was preempted by an exception. When the RFE instruction is executed, it restores the *previous* interrupt mask (IEp) bit and kernel/user mode (KUp) bit in the Status Register into the corresponding *current* status bits (IEc and KUc). It also restores the *old* status bits (IEo and KUo) into the corresponding previous status bits (IEp and KUp). The old status bits (IEo and KUo) remain unchanged. The actions of the RFE instruction are illustrated in Figure 4.2.

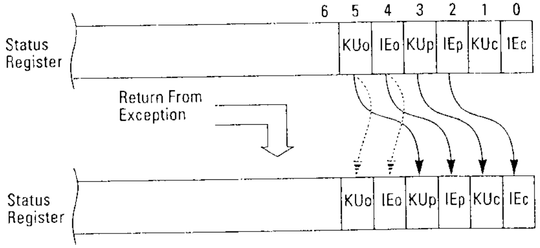


Figure 4.2
Restoring from Exceptions

Cause Register

The contents of the Cause Register describe the last occurring exception. A four-bit exception code field (ExcCode) indicates the cause of the exception. The remaining fields contain detailed information specific to certain exceptions.

With the exception of the Sw bits, all bits in the Cause Register are read-only. Writes to the Sw bits set or reset software interrupts. The format of the 32-bit Cause Register is shown below:

Location:	CP0	Cold Reset Initial Value:	0x0000.0000
Address:	13	Warm Reset Initial Value:	0x0000.0000

31	30	29 28	27 16	15 10	9 8	7 6	5 2	1 0
BD	BT	CE	R	IP[5:0]	Sw[1:0]	R	ExcCode	R

BD — **Branch Delay** 31
The LR333x0 sets this bit to one to indicate that the last exception was taken while executing in a branch delay slot.

BT — **Branch Taken** 30
When the BD bit is set, the BT bit determines whether or not the branch is taken. A value of one in BT indicates that the branch is taken. The Target Address Register holds the return address.

CE — **Coprocessor Error** [29:28]
When taking a Coprocessor Unusable exception, the LR333x0 writes the references coprocessor number in this field. This field is otherwise undefined.

R — **Reserved** [27:16], [7:6], [1:0]
These bits are reserved and read as zero. The LR333x0 ignores attempts to set these bits; however, software should write these bits as zero to ensure compatibility with future versions of hardware.

IP[5:0] — **Interrupt Pending [5:0]** [15:10]
The LR333x0 sets these bits to indicate that an external interrupt is pending on INT[5:0]. Bit 15 corresponds to INT5 and bit 10 corresponds to INT0.

Sw[1:0] — **Software Interrupts [1:0]** [9:8]
By setting either of these bits to one, software causes the LR333x0 to transfer control to the general exception routine. Bit 9 corresponds to Sw1. The exception routine can tell which software interrupt bit is set by reading this field. The exception routine must reset the Sw bits to zero before returning control to the interrupting software.

ExcCode — **Exception Code** [5:2]
The LR333x0 sets this field to indicate the type of event that caused the last general exception. The four bits are encoded as described in the table below.

Value	*Mnemonic*	*Description*
0	Int	External Interrupt
1	—	Reserved
2	—	Reserved
3	—	Reserved
4	AdEL	Address Error Exception (load or instruction)
5	AdES	Address Error Exception (store)
6	IBE	Bus Error Exception (for an instruction fetch)
7	DBE	Bus Error Exception (for a data load or store)
8	Sys	SYSCALL Exception
9	Bp	Breakpoint Exception
A	RI	Reserved Instruction Exception
B	CpU	Coprocessor Unusable Exception
C	Ovf	Arithmetic Overflow Exception
D-F	—	Reserved

Bad Address Register

The Bad Address (BadA) Register is a read-only register that saves the address associated with an illegal access. This register saves only addressing errors (*AdEL* or *AdES*), not bus errors. The format of the 32-bit BadA Register is shown below:

Location:	CP0	Cold Reset Initial Value:	Undefined
Address:	8	Warm Reset Initial Value:	Unchanged

31 0
Bad Address

Target Address Register

The Target Address (TAR) Register is a read-only register that holds the return address for a branch. When the cause of an exception is in the branch delay slot (the LR333x0 sets the BD bit in the Cause Register to one), execution resumes either at the target of the branch or at the Exception Program Counter [EPC] + 8. If the branch was taken, the LR333x0 sets the BT bit in the Cause Register to one and loads the branch target address in the TAR Register. The exception handler needs only to load this address into a register and jump to that location. The format of the 32-bit TAR Register is shown below:

Location:	CP0	Cold Reset Initial Value:	Undefined
Address:	6	Warm Reset Initial Value:	Unchanged

31 … 0
Target Address

Exception Program Counter Register

The 32-bit Exception Program Counter (EPC) Register contains the address where processing resumes after an exception is serviced. In most cases, the EPC Register contains the address of the instruction that caused the exception. However, when the exception instruction resides in a branch delay slot, the Cause Register's BD bit is set to one to indicate that the EPC Register contains the address of the immediately preceding branch or jump instruction. The format of the EPC Register is shown below:

Location:	CP0	Cold Reset Initial Value:	Undefined
Address:	14	Warm Reset Initial Value:	Unchanged

31 … 0
EPC

Processor Revision Identifier Register

The Processor Revision Identifier (PRId) Register contains information that identifies the implementation and revision level of the processor and system control coprocessor.

The revision number can distinguish some chip revisions. However, LSI Logic is free to change this field at any time and does not guarantee that changes to its chips necessarily change the revision number or that changes to the revision number necessarily reflect real chip changes. For this reason, software should not rely on the revision number to characterize the chip.

The format of this 32-bit, read-only register is shown below.

Location:	CP0	LR33300 Reset Initial Value:	0x0000.0AXX
Address:	15	LR33310 Reset Initial Value:	0x0000.0BXX

31 … 16	15 … 8	7 … 0
R	Imp	Rev

R — **Reserved** **[31:16]**
These bits are reserved and read as zero. The LR333x0 ignores attempts to set these bits; however, software should write these bits as zero to ensure compatibility with future versions of hardware.

Imp — **Implementation** **[15:8]**
This eight-bit field contains the LR333x0's implementation number.

Part Number	*Imp[15:8]*
LR33300	0x0A
LR33310	0x0B

Rev — **Revision** **[7:0]**
This eight-bit field contains the LR333x0's revision number.

Part Number	*Rev[7:0]*
LR33300	0x0A
LR33310	0x0A

Debug and Cache Invalidate Control Register

The Debug and Cache Invalidate Control (DCIC) Register contains the enable and status bits for the LR333x0's breakpoint mechanism and the control bits for the Cache Controller's invalidate mechanism. All bits in the DCIC Register are readable and writable. The format of the DCIC Register is shown below. Additional details on the function of each DCIC Register bit are provided in the paragraphs that follow.

Location: CP0 Cold Reset Initial Value: 0x0000.0000
Address: 7 Warm Reset Initial Value: 0x0000.0000

31	30	29	28	27	26	25	24	23	22	6	5	4	3	2	1	0
TR	UD	KD	TE	DW	DR	DAE	PCE	DE	R		T	W	R	DA	PC	DB

Breakpoint Control Bits – There are three types of breakpoint exceptions: those caused by instruction fetches (break on program counter), those caused by data accesses (break on data address), and those caused by non-sequential instruction fetches (trace). To make the debug mechanism precise, it can be configured to detect breakpoints in user mode or kernel mode or both. It can also be configured to discriminate between read and write accesses. All breakpoint and trace conditions are posted as Breakpoint Exceptions in the Cause Register.

TR **Trap Enable** 31
When software sets TR to one, the LR333x0 vectors to the Debug Exception address when it encounters a debug condition. If this bit is zero, the LR333x0 will not trap, but it will set the appropriate debug status bits for any debug condition that software has enabled.

UD **User Debug Enable** 30
When software sets UD to one, the LR333x0 detects debug conditions when running in user mode.

KD **Kernel Debug Enable** 29
When software sets KD to one, the LR333x0 detects debug conditions when running in kernel mode.

TE **Trace Enable** 28
When software sets TE to one and the LR333x0 fetches any non-sequential instruction, the LR333x0 vectors to the Debug Exception address.

Three bits control the LR333x0's Breakpoint Data Address feature. DAE enables or disables the breakpoint function. DW and DR control whether read and/or write accesses are trapped.

DW **Data Write Enable** 27
When software sets DW and DAE to one and the LR333x0 writes to the address specified in the Breakpoint Data Address Register, the LR333x0 vectors to the Debug Exception address.

DR **Data Read Enable** 26
When software sets DR and DAE to one and the LR333x0 reads from the address specified in the Breakpoint Data Address Register, the LR333x0 vectors to the Debug Exception address.

DAE **Data Address Breakpoint Enable** 25
When software sets DAE to one and the LR333x0 accesses the address specified in the Breakpoint Data Address Register, the LR333x0 vectors to the Debug Exception address.

PCE **Program Counter Breakpoint Enable** 24
When software sets PCE to one and the LR333x0 fetches an instruction from the address specified in the Breakpoint Program Counter Register, the LR333x0 vectors to the Debug Exception address.

DE **Debug Enable** 23
Software sets DE to one to enable the LR333x0's debug facility. If this bit is zero, the LR333x0 will not detect the breakpoint or trace conditions specified in the DCIC Register, bits [31:24].

R **Reserved** [22:6]
These bits are reserved and read as zero. The LR333x0 ignores attempts to set these bits; however, software should write these bits as zero to ensure compatibility with future versions of hardware.

Breakpoint Status Bits – The breakpoint mechanism posts the cause of a Debug Exception with these six status bits.

T **Trace** 5
The LR333x0 sets T to one when it detects a trace condition.

W **Write Reference** 4
The LR333x0 sets W to one when it detects a write reference to the address specified in the Breakpoint Address Register.

R **Read Reference** 3
The LR333x0 sets R to one when it detects a read reference to the address specified in the Breakpoint Data Address Register.

DA **Data Address** 2
The LR333x0 sets DA to one when it detects a data address debug condition.

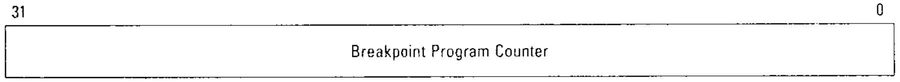
PC **Program Counter** 1
The LR333x0 sets PC to one when it detects a program counter debug condition.

DB **Debug** 0
The LR333x0 sets DB to one when it detects any debug condition.

Breakpoint Program Counter Register

The Breakpoint Program Counter (BPC) Register is a read/write register that software uses to specify a program counter breakpoint. The format of the 32-bit BPC Register is shown below:

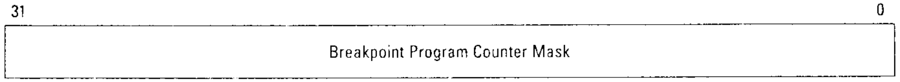
Location: CP0 Cold Reset Initial Value: Undefined
Address: 3 Warm Reset Initial Value: Unchanged



Breakpoint Program Counter Mask Register

The Breakpoint Program Counter Mask (BPCM) Register is a read/write register that masks bits in the BPC Register. A one in any bit in the BPCM Register indicates that the LR333x0 compares the corresponding bit in the BPC Register for program counter exceptions. Values of zero in the mask indicate that the LR333x0 does not check the corresponding bits in the BPC Register for exceptions. The format of the 32-bit BPCM Register is shown below:

Location: CP0 Cold Reset Initial Value: 0xFFFF.FFFF
Address: 11 Warm Reset Initial Value: 0xFFFF.FFFF

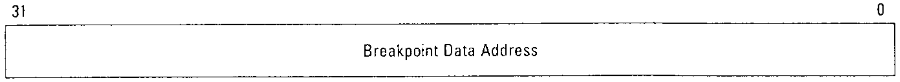


For example, if the BPCM Register is set to 0xFFFF.0000, the LR333x0 compares bits [31:16] of the Program Counter with the corresponding bits in the BPC Register for program counter exceptions.

Breakpoint Data Address Register

The Breakpoint Data Address (BDA) Register is a read/write register that software uses to specify a data address breakpoint. The format of the 32-bit BDA Register is shown below:

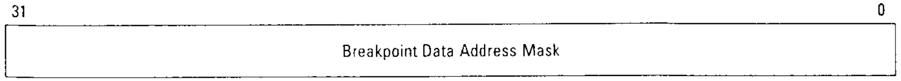
Location: CP0 Cold Reset Initial Value: Undefined
Address: 5 Warm Reset Initial Value: Unchanged



Breakpoint Data Address Mask Register

The Breakpoint Data Address Mask (BDAM) Register is a read/write register that masks bits in the BDA Register. A one in any bit in the BDAM Register indicates that the LR333x0 compares the corresponding bit in the BDA Register for debug exceptions. Values of zero in the mask indicate that the LR333x0 does not check the corresponding bits in the BDA Register for exceptions. The format of the 32-bit BDAM Register is shown below:

Location: CP0 Cold Reset Initial Value: 0xFFFF.FFFF
Address: 9 Warm Reset Initial Value: 0xFFFF.FFFF



For example, if the BDAM Register is set to 0xFFFF.0000, the LR333x0 compares bits [31:16] of the BDA Register for debug exceptions.

4.3 Exception Description Details

This section describes each exception in detail. The exceptions are discussed according to cause, handling, and servicing. Note that machine exceptions cannot be masked.

Exception Vector Locations

The LR333x0 uses three different addresses for exception vectors:

- The reset exception vector is at address 0xBFC0.0000.
- The debug exception vector is at address 0x8000.0040.
- The general exception vector, which is used for all other types of exceptions, is at address 0x8000.0080.

If the BEV (Bootstrap Exception Vector) bit in the Status Register is set to one, the debug exception vector address is changed to 0xBFC0.0140, and the general exception vector is changed to 0xBFC0.0180.

Address Error Exception

Cause – The address error exception occurs when an attempt is made to load, fetch, or store a word that is not aligned on a word boundary. Attempts to load or store a halfword that is not aligned on a halfword boundary also cause this exception. The exception also occurs in user mode if a reference is made to an address whose most-significant bit is set, indicating a kernel mode address. This exception is not maskable.

Handling – When an address error exception occurs, the LR333x0 branches to the general exception vector (0x8000.0080 or 0xBFC0.0180). The LR333x0 sets the *AdEL* or *AdES* exception code in the Cause Register's ExcCode field to indicate whether the address error occurred during an instruction fetch or a load operation (*AdEL*) or a store operation (*AdES*). The LR333x0 saves the KUp, IEp, KUc, and IEc bits of the Status Register into the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

The EPC Register points at the instruction that caused the exception, unless the instruction is in a branch delay slot. In that case, the EPC Register points at the branch instruction preceding the exception-causing instruction and sets the BD bit of the Cause Register.

When this exception occurs, the BadA Register contains the address that was improperly aligned or that improperly addressed kernel data while in user mode.

Servicing – A kernel should hand the executing process a segmentation violation signal. Such an error is usually fatal, although an alignment error might be handled by simulating the instruction that caused the error.

Breakpoint Exception

Cause – The breakpoint exception occurs when the LR333x0 executes the BREAK instruction. This exception is not maskable.

Handling – When the breakpoint exception occurs, the LR333x0 branches to the general exception vector (0x8000.0080 or 0xBFC0.0180) and sets the BP code in the Cause Register's ExcCode field. The LR333x0 saves the KUp, IEp, KUc, and IEc bits of the Status Register into the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

The EPC Register points at the BREAK instruction that caused the exception, unless the instruction is in a branch delay slot. In that case, the EPC Register points at the branch instruction preceding the BREAK and sets the BD bit of the Cause Register.

Servicing – Control is transferred to the applicable system routine. Unused bits of the BREAK instruction (bits [25:6]) can be used to pass additional information. These bits can be examined by loading the contents of the instruction pointed at by the EPC Register. If the instruction resides in the branch delay slot, a value of four must be added to the contents of the EPC Register to locate the instruction.

To resume execution, the EPC Register must be changed so that the LR333x0 does not execute the BREAK instruction again. A value of four must be added to the contents of the EPC Register before returning. If a BREAK instruction is in the branch delay slot, the branch instruction must be interpreted in order to resume execution.

Bus Error Exception

Cause – The bus error exception occurs when the external logic asserts the $\overline{\text{BERR}}$ input to end an external memory transaction such as an instruction fetch or store operation. For example, events like a bus time-out, backplane bus parity errors, and invalid physical memory addresses or access types should cause external logic to signal this exception.

Except for stores, this exception is synchronous. For store transactions, the delay caused by the write buffer prevents the exception from being synchronous with the instruction stream. This exception is not maskable.

Handling – When a bus error exception occurs, the LR333x0 branches to the general exception vector (0x8000.0080 or 0xBFC0.0180). The LR333x0 sets the IBE or DBE code in the Cause Register's ExcCode field to indicate whether the error occurred during an instruction-fetch reference

(IBE) or during a data load or store reference (DBE). The LR333x0 saves the KUp, IEp, KUc, and IEc bits of the Status Register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

The EPC Register points at the instruction that caused the exception, unless the instruction is in a branch delay slot. In that case, the EPC Register points at the branch instruction preceding the exception-causing instruction and sets the BD bit of the Cause Register.

Servicing – The physical address where the fault occurred can be computed from the information in the CP0 registers:

- If the Cause Register's IBE code is set (showing an instruction fetch), the address is contained in the EPC Register.
- If the Cause Register's DBE exception code is set, a load or store instruction caused the exception. For load instructions, the address of the *instruction* that caused the exception is contained in the EPC Register (if the BD bit of the Cause Register is set, add four to the contents of the EPC Register). The address of the load reference can then be obtained by interpreting the instruction.

  For store transactions, the EPC Register contains the address of the interrupted instruction, which may not be related to the store instruction that caused the exception.

Coprocessor Unusable Exception

Cause – The coprocessor unusable exception occurs when an instruction is sent to a coprocessor unit that has not been marked usable (the appropriate CU bit in the Status Register has not been set). For CP0 instructions, this exception occurs when the unit has not been marked usable, and the process is executing in user mode. CP0 is always usable from kernel mode regardless of the setting of the Cu0 bit in the Status Register. This exception is not maskable.

Handling – When a coprocessor unusable exception occurs, the LR333x0 branches to the general exception vector (0x8000.0080 or 0xBFC0.0180) and sets the CpU code in the Cause Register's ExcCode field. The LR333x0 saves the KUp, IEp, KUc, and IEc bits of the Status Register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Only one coprocessor can fail at a time. The contents of the Cause Register's CE (Coprocessor Error) field show which of the four coprocessors (0, 1, 2, or 3) the LR333x0 referenced when the exception occurred.

The EPC Register points at the coprocessor instruction that caused the exception unless the instruction is in a branch delay slot. In that case, the EPC Register points at the branch instruction that preceded the coprocessor instruction and sets the BD bit of the Cause Register.

Servicing – The coprocessor unit that was referenced is identified by examining the contents of the Cause Register's CE field. If the process is entitled access to the coprocessor, the coprocessor is marked usable, and the corresponding user state is restored to the coprocessor.

If the process is entitled access to the coprocessor, but the coprocessor is known not to exist or to have failed, the system could interpret the coprocessor instruction. If the BD bit is set in the Cause Register, the branch instruction must be interpreted; then the coprocessor instruction could be emulated with the EPC Register advanced past the coprocessor instruction.

If the process is not entitled access to the coprocessor, the process executing at the time should be handed an illegal instruction/privileged instruction fault signal. Such an error is usually fatal.

Debug Exception

Cause – The debug exception occurs when the LR333x0 detects a program counter breakpoint, data address breakpoint, or trace condition. The LR333x0 detects debug exceptions only if one or more of the debug conditions is enabled as described in the subsection entitled "Debug and Cache Invalidate Control Register" on page 4-11.

Handling – When the debug exception occurs, the LR333x0 branches to the debug exception vector (0x8000.0040 or 0xBFC0.0140) and sets the BP code in the Cause Register's ExcCode field. The LR333x0 saves the KUp, IEp, KUc, and IEc bits of the Status Register into the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

The EPC Register points at the instruction that caused the exception, unless the instruction is in a branch delay slot. In that case, the EPC Register points at the branch instruction preceding the delay slot instruction and sets the BD bit of the Cause Register.

Servicing – Control is transferred to the debug exception handling routine. The breakpoint handler must examine the debug status bits of the DCIC Register to determine the cause of the exception.

Interrupt Exception

Cause – The interrupt exception occurs when one of eight interrupt conditions (software generates two, hardware generates six) is asserted. The significance of these interrupts is implementation-dependent.

Each of the eight external interrupts can be individually masked by clearing the corresponding bit in the IntMask field of the Status Register. All eight of the interrupts can be masked at once by clearing the IEc bit in the Status Register. The six hardware interrupts are masked when the INTMASK input is asserted.

Handling – When an interrupt exception occurs, the LR333x0 branches to the general exception vector (0x8000.0080 or 0xBFC0.0180) and sets the Int code in the Cause Register's ExcCode field. The LR333x0 saves the KUp, IEp, KUc, and IEc bits of the Status Register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

The IP field in the Cause Register shows which of six external interrupts are pending; the Sw field in the Cause Register shows which of two software interrupts is pending. More than one interrupt can be pending at a time.

Servicing – If software generated the interrupt, the interrupt condition is cleared by setting the corresponding Cause Register bit Sw[1:0] to zero.

If external hardware generated the interrupt, the interrupt condition is cleared by alleviating the condition that caused the assertion of the interrupt signal.

Overflow Exception

Cause – The overflow exception occurs when an ADD, ADDI, SUB, or SUBI instruction results in a two's complement overflow. This exception is not maskable.

Handling – When an overflow exception occurs, the LR333x0 branches to the general exception vector (0x8000.0080 or 0xBFC0.0180) and sets the Ovf code of the Cause Register. The LR333x0 saves the KUp, IEp, KUc, and IEc bits of the Status Register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

The EPC Register points at the instruction that caused the exception, unless the instruction is in a branch delay slot. In that case, the EPC Register points at the branch instruction that preceded the exception-causing instruction and sets the BD bit of the Cause Register.

Servicing – A kernel should hand the executing process a floating-point exception or integer overflow error when this exception occurs. Such an error is usually fatal.

Reserved Instruction Exception

Cause – The reserved instruction exception occurs when the LR333x0 executes an instruction whose major opcode (bits 31:26) is undefined or a SPECIAL instruction whose minor opcode (bits 5:0) is undefined.

This exception provides a way to interpret instructions that might be added to or removed from the processor architecture. This exception is not maskable.

Handling – When a reserved instruction exception occurs, the LR333x0 branches to the general exception vector (0x8000.0080 or 0xBFC0.0180) and sets the RI code of the Cause Register's ExcCode field. The LR333x0 saves the KUp, IEp, KUc, and IEc bits of the Status Register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

The EPC Register points at the reserved instruction that caused the exception, unless the instruction is in a branch delay slot. In that case, the EPC Register points at the branch instruction that preceded the reserved instruction and sets the BD bit of the Cause Register.

Servicing – If instruction interpretation is not implemented, the kernel should hand the executing process an illegal instruction/reserved operand fault signal. Such an error is usually fatal.

An operating system can interpret the undefined instruction and pass control to a routine that implements the instruction in software. If the undefined instruction is in the branch delay slot, the routine that implements the instruction is responsible for simulating the branch instruction after the undefined instruction has been "executed." Simulation of the branch instruction includes determining whether the conditions of the branch were met and then transferring control to the branch target address (if required) or to the instruction following the delay slot if the branch is not taken. If the branch is not taken, the next instruction's address is [EPC] + 8. If the branch is taken, the BT bit in the Cause Register is set to one and the branch target address is in the Target Address Register.

Reset Exception

Cause – The reset exception occurs upon deassertion of the LR333x0's $\overline{\text{RESET}}$ signal. This exception is not maskable. Refer to the descriptions of the $\overline{\text{HIGHZ}}$ and $\overline{\text{RESET}}$ signals on page 8-14 and to the two figures entitled "Cold Reset Timing" and "Warm Reset Timing" in the *LR33300 and LR33310 Enhanced Self-Embedding™ Processors Technical Manual Addendum* for more specific information on warm and cold resets.

Handling – When a reset exception occurs, the LR333x0 provides a special exception vector (0xBFC0.0000). The vector resides in the LR333x0's non-cacheable address space; therefore the hardware does not need to initialize the cache to handle this exception. The processor can fetch and execute instructions while the caches are in an undefined state. Software can initialize the caches using the cache invalidate mechanism described in Section 6.4, "Cache Maintenance and Testing."

The contents of all registers in the LR333x0 are undefined when the reset exception occurs, except for the following:

- All bits in the Status Register are set to zero except BEV, which is set to one.
- All bits in the Cause Register are set to zero.
- All bits in the DCIC Register are set to zero.

Servicing – The reset exception is serviced by initializing all processor registers, coprocessor registers, and the memory system. Typically, diagnostics would then be executed, and the operating system bootstrapped. The reset exception vector is selected to appear within the non-cacheable, unmapped memory space of the machine so that instructions can be fetched and executed while the cache and memory system are still in an undefined state.

System Call Exception

Cause – The system call exception occurs when the LR333x0 executes a SYSCALL instruction. This exception is not maskable.

Handling – When the system call exception occurs, the LR333x0 branches to the general exception vector (0x8000.0080 or 0xBFC0.0180) and sets the Sys code in the Cause Register's ExcCode field. The LR333x0 saves the KUp, IEp, KUc, and IEc bits of the Status Register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

The EPC Register points at the SYSCALL instruction that caused the exception, unless the SYSCALL instruction is in a branch delay slot. In that case, the EPC Register points at the branch instruction that preceded the SYSCALL instruction and sets the BD bit of the Cause Register.

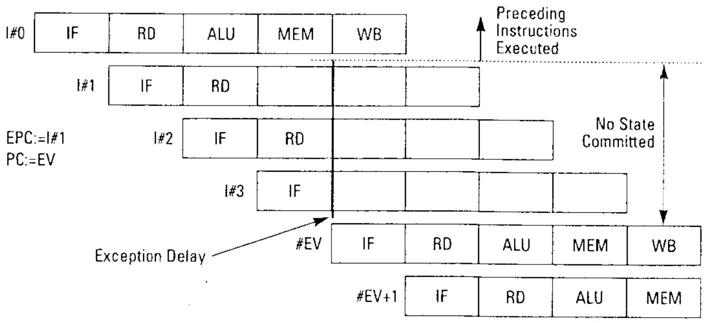
Servicing – The operating system transfers control to the applicable system routine. To resume execution, the EPC Register must be altered so that the SYSCALL instruction does not execute again. A value of four is added to the EPC Register before returning to avoid re-execution of the SYSCALL instruction. If the SYSCALL instruction is in a branch delay slot, the branch must be interpreted.

4.4 Interrupt Exception Handling

The interrupt exception occurs when one of the eight interrupt conditions is asserted. Figure 4.3 illustrates the delay for an exception. Instruction #1 was interrupted during its ALU pipeline stage. The instructions in the MEM and WB pipeline stages will be completed (these are the two instructions previous to Instruction #1). Instruction #1, Instruction #2, and Instruction #3 are cancelled. The next instruction is located at the exception vector (#EV).

Because other exceptions can be detected in stages before the ALU pipeline stage, an interrupt exception may not be taken if another such exception occurs at the same time. Therefore the interrupt signal must be held asserted until the exception-handling software disables the corresponding interrupt bit.

Figure 4.3 Interrupt Exception Delay Example



The following software routine is an example of a multipurpose interrupt handler routine. Most operating systems do not need numerous distinct exception vector addresses; instead, they first execute a common interrupt handler that determines the specific processing required to handle the exception. The operating system itself may then determine what state information, if any, needs to be saved, thus saving hardware expense and letting the operating system supply the appropriate complexity.

In the following example, control is passed to the appropriate interrupt handler via a vector table, but only after higher priority interrupts have been re-enabled. The vector scheme permits handlers to be written in C or assembler. An assembler handler is indicated by setting the least significant bit of the address in the vector table.

Note that this routine is just an example—the code shown may not be the most efficient.

```
#include "mips.h"                              /* registers that are saved for all ints */
#define E_SR         0
#define E_EPC        1
#define E_AT         2
#define E_RA         3
#define E_SIZE       4                         /* must be even, double word aligned */
                                               /* registers that are saved for C handlers */
#define C_V0         0
#define C_V1         1
#define C_A0         2
#define C_A1         3
#define C_A2         4
#define C_A3         5
#define C_T0         6
#define C_T1         7
#define C_T2         8
#define C_T3         9
#define C_T4         10
#define C_T5         11
#define C_T6         12
#define C_T7         13
#define C_T8         14
#define C_T9         15
#define C_LO         16
#define C_HI         17
#define C_SIZE       18                        /* must be even, double word aligned */
     .globl isr
     .ent isr
     .set noat
isr:
     .set noreorder
     mfc0            k0,C0_CAUSE
     nop
     .set reorder
                                               # first see if EXCCODE=0
```

```
     and             k1,k0,CAUSE_EXCMASK
     beq             k1,zero,ext_int
goPmon:                                        # transfer control to PMON
     li              k0,0x9fc00200
     lw              k0,12*4(k0)
     j               k0

ext_int:                                       # save AT, ra, EPC & SR
     subu            sp,E_SIZE*4               # allocate stack space
     sw              AT,E_AT*4(sp)             # save AT
     sw              ra,E_RA*4(sp)             # save RA
     .set noreorder
     mfc0            k1,C0_EPC
     nop
     sw              k1,E_EPC*4(sp)            # save EPC
     mtc0            k1,C0_SR
     nop
     sw              k1,E_SR*4(sp)             # save SR
     .set reorder
                                               # compute possible requestors (those not masked)
     and             k0,k1                     # CAUSE is still in k0
     and             k0,SR_IMASK

                                               # determine highest priority requester
     sll             k0,31-16                  # move bit 16 into MS position
     li              ra,9                      # initial int#
1:   sll             k0,1                      # move next bit into MS position
     subu            ra,1                      # next int#
     bgez            k0,1b                     # not found, loop back
                                               # int# of highest priority int is now in ra
                                               # convert int# to mask

     li              k0,-1
     sll             k0,ra
     sll             k0,8                      # move it into position
     and             k0,SR_IMASK               # clear other bits
                                               # mask now in k0
                                               # load new mask into SR and enable ints
     lw              k1,E_SR*4(sp)             # get current SR value
     li              AT,~SR_IMASK
     and             k1,AT                     # clear IM field
     or              k1,k0                     # load new mask value
     .set noreorder
     mtc0            k1,C0_SR                  # update SR
     nop                                       # give it time
     nop                                       # ...
     or              k1,SR_IEC                 # now enable ints
     mtc0            k1,C0_SR                  # update SR
     nop
     .set reorder
                                               # ints now enabled

                                               # use int# in ra to index into vector table
     la              AT,vect_table
```

```
        subu        ra,1                # zero based table
        sll         ra,2                # word sized entries
        addu        AT,ra               # compute entry addr
        lw          AT,(AT)             # pick up addr from table
        bne         AT,zero,1f          # return to Pmon if addr is zero
                                        # restore registers and return to Pmon
                                        # restore SR, which disables ints
        .set noreorder
        lw          AT,E_SR*4(sp)
        nop
        mtc0        AT,C0_SR
        nop
        .set reorder
                                        # now restore other regs
        lw          AT,E_AT*4(sp)
        lw          ra,E_RA*4(sp)
        lw          k0,E_EPC*4(sp)
        addu        sp,E_SIZE*4

        b           goPmon
1:                                      # determine if handler is in C or ASM
        and         ra,AT,1             # test lsb
        .set noreorder
        beq         ra,zero,c_isr       # branch if in C
        nop                             # nop necessary to stop ASM from
                                        # putting srl in delay slot
        .set reorder
        srl         AT,1                # clear lsb
        sll         AT,1
                                        # transfer control to ASM handler
        jal         AT                  # saves return address in ra

done:                                   # restore SR, which disables ints
        .set noreorder
        lw          AT,E_SR*4(sp)
        nop
        mtc0        AT,C0_SR
        nop
        .set reorder                    # restore other regs and deallocate stack space
        lw          AT,E_AT*4(sp)
        lw          ra,E_RA*4(sp)
        lw          k0,E_EPC*4(sp)
        addu        sp,E_SIZE*4
        .set noreorder
        j           k0                  # return to interrupted program
        rfe
        .set reorder

c_isr:                                  # save other registers required for C
        subu        sp,C_SIZE*4         # allocate stack space
        sw          v0,C_V0*4(sp)
        sw          v1,C_V1*4(sp)
        sw          a0,C_A0*4(sp)
```

```
        sw          a1,C_A1*4(sp)
        sw          a2,C_A2*4(sp)
        sw          a3,C_A3*4(sp)
        sw          t0,C_T0*4(sp)
        sw          t1,C_T1*4(sp)
        sw          t2,C_T2*4(sp)
        sw          t3,C_T3*4(sp)
        sw          t4,C_T4*4(sp)
        sw          t5,C_T5*4(sp)
        sw          t6,C_T6*4(sp)
        sw          t7,C_T7*4(sp)       # 16..23 (s0..s7) don't need to be saved
        sw          t8,C_T8*4(sp)
        sw          t9,C_T9*4(sp)       # 26..27 (k0..k1) don't need to be saved
                                        # 28 (gp) is usually the same for ISR
                                        # and interrupted program
                                        # 29 (sp) doesn't need to be saved
                                        # 30 (s8) doesn't need to be saved
        mflo        ra
        sw          ra,C_LO*4(sp)
        mfhi        ra
        sw          ra,C_HI*4(sp)
                                        # call the C routine
        subu        sp,24               # allocate min context
        jal         AT                  # call C handler
        addu        sp,24               # deallocate context
                                        # restore the machine state
        lw          v0,C_V0*4(sp)
        lw          v1,C_V1*4(sp)
        lw          a0,C_A0*4(sp)
        lw          a1,C_A1*4(sp)
        lw          a2,C_A2*4(sp)
        lw          a3,C_A3*4(sp)
        lw          t0,C_T0*4(sp)
        lw          t1,C_T1*4(sp)
        lw          t2,C_T2*4(sp)
        lw          t3,C_T3*4(sp)
        lw          t4,C_T4*4(sp)
        lw          t5,C_T5*4(sp)
        lw          t6,C_T6*4(sp)
        lw          t7,C_T7*4(sp)
        lw          t8,C_T8*4(sp)
        lw          t9,C_T9*4(sp)
        lw          ra,C_LO*4(sp)
        mtlo        ra
        lw          ra,C_HI*4(sp)
        mthi        ra
        addu        sp,C_SIZE*4         # deallocate stack space
        b           done                # branch to code to restore regs
                                        # and return to interrupted program
        .set reorder
        .set at
        .end isr
```