OSOROM Instruction Set Reference

The Moroso Project

July 19, 2015

Contents

1 CPU Architecture **3**
- 1.1 Instructions . . . 3
- 1.2 Registers . . . 3
- 1.3 ALU Instruction Formats . . . 3
- 1.4 VLIW . . . 4
- 1.5 Virtual Memory . . . 4
- 1.6 Coprocessor Registers . . . 5
- 1.7 Exceptions and Interrupts . . . 6

2 Instruction Set Summary **8**
- 2.1 Instruction Set Encoding . . . 8
- 2.2 ALU Opcodes . . . 9

3 Instruction Listing **10**
- 3.1 ADD - Addition . . . 11
- 3.2 AND - Bitwise And . . . 12
- 3.3 B - Branch . . . 13
- 3.4 BL - Branch with Link . . . 14
- 3.5 BREAK . . . 15
- 3.6 CMPBC - Compare Bits Clear . . . 16
- 3.7 CMPBS - Compare Bits Set . . . 17
- 3.8 CMPEQ - Compare Equal . . . 18
- 3.9 CMPLES - Compare Less Than or Equal (Signed) . . . 19
- 3.10 CMPLEU - Compare Less Than or Equal (Unsigned) . . . 20
- 3.11 CMPLTS - Compare Less Than (Signed) . . . 21
- 3.12 CMPLTU - Compare Less Than (Unsigned) . . . 22
- 3.13 DIV - Integer Division . . . 23
- 3.14 ERET - Return from Exception . . . 24
- 3.15 FENCE . . . 25
- 3.16 FLUSH - Flush L1 Caches . . . 26
- 3.17 LB - Load Byte . . . 27
- 3.18 LH - Load Half-Word . . . 28
- 3.19 LL - Load Linked . . . 29
- 3.20 LW - Load Word . . . 30
- 3.21 MFC - Move From Coprocessor Register . . . 31
- 3.22 MFHI - Move From Multiply/Division Overflow Register . . . 32
- 3.23 MOV - Move . . . 33

3.24 MTC - Move To Coprocessor Register 34
3.25 MTHI - Move To Multiply/Division Overflow Register 35
3.26 MVN - Move and Invert 36
3.27 MULT - Integer Multiplication 37
3.28 NOR - Bitwise Logical NOR 38
3.29 OR - Bitwise Logical OR 39
3.30 RSB - Reverse Subtraction 40
3.31 SB - Store Byte 41
3.32 SC - Store Conditional 42
3.33 SH - Store Half-Word 43
3.34 SUB - Subtraction 44
3.35 SW - Store Word 45
3.36 SXB - Sign-Extend Byte 46
3.37 SXH - Sign-Extend Half-Word 47
3.38 SYSCALL - System Call Exception 48
3.39 XOR - Bitwise Exclusive OR 49

Chapter 1

CPU Architecture

1.1 Instructions

Each instruction is a 32-bit word, stored in little-endian order. The top 2 bits of the instruction select a predicate register, and the third bit from the top optionally inverts it. The instruction is only executed if the resulting predicate is true.

1.2 Registers

There are 32 32-bit general-purpose registers, R0 through R31. R31 is used as the link register for BL instructions. The program counter is stored separately, and may only be modified through branch instructions and read through BL instructions.

There are three one-bit predicate registers, P0 through P2, that may be written to by compare instructions. A fourth predicate register, P3, is always true. Instructions predicated on P3 will always execute, and writes to P3 are ignored.

1.3 ALU Instruction Formats

ALU instructions may accept immediate values for one of their input operands, which come in two forms: Short immediates are embedded in the instruction and consist of a 10-bit constant and a 4-bit rotate amount. To achieve any even rotation between 0 and 30, the rotate amount is first multiplied by 2, and the constant is rotated right by the result. ALU instructions that accept a single operand use the portion of the instruction normally used for the other operand register to extend the constant to 15 bits. Long immediate operands are 32 bit values, and occupy the memory word following the instruction.

If a register is used as the last source operand of an ALU instruction, it may optionally be shifted by a 5-bit immediate value, as specified with the `SHF` and `SHIFTAMT` fields of the instruction. The `SHIFTAMT` field is the amount to shift, between 0 and 31, and the `SHF` field specifies the shift type, as follows:

Encoding	Mnemonic	Shift Type
0 0	`LSL`	Logical Shift Left
0 1	`LSR`	Logical Shift Right
1 0	`ASR`	Arithmetic Shift Right
1 1	`ROR`	Rotate Right

ALU instructions taking a single operand may also specify a register shifted by an unsigned value contained in another register `Rt`. In this case, the same `SHF` types are available. If the value of the shift register is greater than 31, the following behavior is used: Logical shifts zero the result, arithmetic right shift extends the sign bit of the operand register across the entire result, and rotate rotates the value by `Rt % 32`.

1.4 VLIW

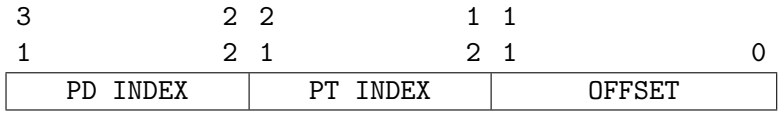
At each time step, the CPU fetches a "packet" of 4 instructions located contiguously in memory and executes them in parallel. These packets must be aligned to their 16-byte size, and branch targets must also be so aligned.

There are three types of instructions: Control, Memory, and ALU instructions. Only one control instruction may be executed in any given packet; it must occupy the first (lowest-address) slot in its packet. Likewise, only the first two slots in a packet may execute memory instructions. ALU instructions may be located in any slot.

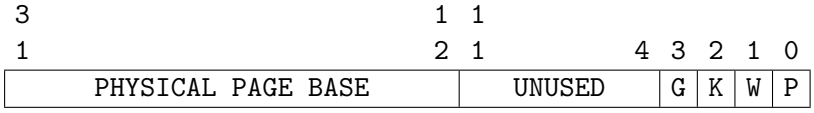
If a long immediate operand is specified for an ALU instruction, the following slot is interpreted as an operand and no operation is issued for that slot. The final slot in a packet may not specify a long immediate operand.

1.5 Virtual Memory

Virtual memory is accomplished through a hardware-filled TLB, with 4KB pages and a 2-level page-table structure. Page directories and page tables are one page in size, containing 1024 32-bit entries. Virtual addresses are 32 bits and are broken up into 3 parts: Page Directory Index, Page Table Index, and Page Offset, as follows:



The page directory base is stored in a coprocessor register, `PTBR`. Writes to this register will flush all entries not marked global from the TLB. Page table entries and page directory entries are each 32 bits and have the same format:



The four flags in each entry have the following meanings:

- `G`: Global page. Mappings with this bit set in either level will not be flushed from the TLB when the page table base register is modified.
- `K`: Kernel-only page. Mappings with this bit set in either level may only be read or written to while the processor is in kernel mode. Attempting to access such mappings from user mode will generate a page fault exception.
- `W`: Writeable page. Attempting to write to a page without this bit set in both levels will result in a page fault.

- `P`: Present. Attempting to access a page without this bit set in both levels will result in a page fault.

Since our FPGA has 512MB of physical memory, 17 bits for base addresses 12 bits for offsets are sufficient to address any phyiscal memory. However, peripherals are mapped into the same physical address space (staring at `0x80000000`), so the entire 20 bits available for the base are needed.

1.6 Coprocessor Registers

The coprocessor registers are control registers accessible only in kernel mode via the MFC and MTC instructions. They provide information about exceptions to the kernel and are used to configure the processor.

1.6.1 List of Coprocessor Registers

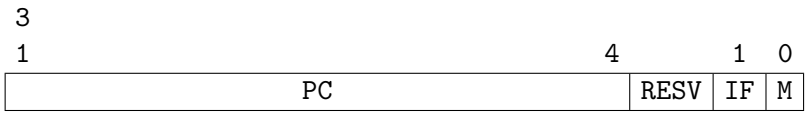
Encoding	Mnemonic	Purpose
`00000`	PFLAGS	Processor control flags. See below.
`00001`	PTB	Page Table Base Register. Specifies the start of the page directory for virtual address translation. Writes to this register flush the TLB.
`00010`	EHA	Exception Handler Address. Specifies a virtual address to which to transfer control when an exception or interrupt is encountered.
`00011`	EPC	Error PC. The virtual address of the last instruction packet to cause an exception, or the first instruction not executed due to an interrupt. LSBs contain system state when the exception occurred.
`00100`	EC0	Error Cause 0. The cause of an exception occurring on lane 0, or the type of a received interrupt. See Section 1.7 for a list of causes.
`00101`	EC1	Error Cause 1. The cause of an exception occurring on lane 1.
`00110`	EC2	Error Cause 2. The cause of an exception occurring on lane 2.
`00111`	EC3	Error Cause 3. The cause of an exception occurring on lane 3.
`01000`	EA0	Error Address 0. If lane 0 makes an invalid memory access, the virtual address it attempted to access will be stored in this register.
`01001`	EA1	Error Address 1. If lane 1 makes an invalid memory access, the virtual address it attempted to access will be stored in this register.
`10000`	SP0	Scratchpad register for system software usage.
`10001`	SP1	Scratchpad register for system software usage.
`10010`	SP2	Scratchpad register for system software usage.
`10011`	SP3	Scratchpad register for system software usage.

1.6.2 The PFLAGS Register

At present, the PFLAGS register contains 2 flags: Bit 0 is 1 if interrupts are enabled, and bit 1 is 1 if paging is enabled. More control flags may be added here if needed.

1.6.3 The EPC Register

The `EPC` register's MSBs contain the PC at which an exception occurred. (If an instruction fetch failed because of a paging exception, `EPC[PC]` contains the PC for which the fetch attempt failed.)



The `M` field contains the system mode at the time that the exception occurred: if `EPC[M]` is set, then the system was in kernel mode at the time, and otherwise, the system was in user mode. The `EPC[IF]` field contains the interrupt flag from `PFLAGS` at the time; if it is set, then interrupts were enabled, and if it is clear, than interrupts were disabled.

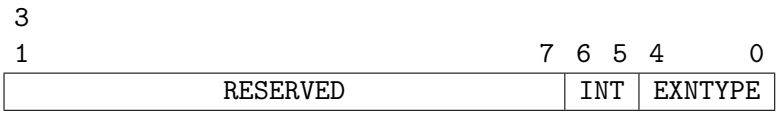
1.6.4 Scratchpad registers

Scratchpad registers are provided for the convenience of system software, should it wish to save and restore user registers (or other data) without reserving user-architectural registers for that purpose. Their value is defined not to affect system state other than through reads and writes.

1.7 Exceptions and Interrupts

Exceptions and interrupts cause the processor to enter kernel mode and transfer control to the virtual address specified in the EHA register. If an interrupt occurs, the code corresponding to that interrupt will be written into the EC0 register. If one or more instructions in a packet cause exceptions, a code corresponding to the exception type will be written into the EC register corresponding to that instruction's location in the packet. If an instruction on lane 0 or 1 attempts to access an invalid memory address, the address will be written into the EA register for that lane. In all cases, the virtual address of the program counter where the exception occurred is written into the EPC register. For exceptions, this is the address of the instruction packet that caused the exception. For interrupts, this is the address of the first instruction packet not executed due to the interrupt. In all cases, interrupts are disabled. This is necessary in order to ensure the kernel can save the coprocessor registers before they are overwritten by another interrupt.

1.7.1 Error Code Format



- `EXNTYPE` corresponds to a value in Table 1.1 and indicates the type of exception that occurred.
- If `EXNTYPE` indicates that an interrupt occurred, `INT` will correspond to a value in Table 1.2 and signify the type of interrupt.

Error Code	Type
00000	No Error
00001	Page Fault on Instruction Fetch
00010	Illegal Instruction
00011	Insufficient Permissions
00100	Duplicate Destination
00101	Page Fault on Data Access
00110	Invalid Physical Address
00111	Divide by Zero (TODO are we doing other arithmetic exceptions??)
01000	Interrupt
01001	SYSCALL
01010	BREAK

Table 1.1: Exception Codes

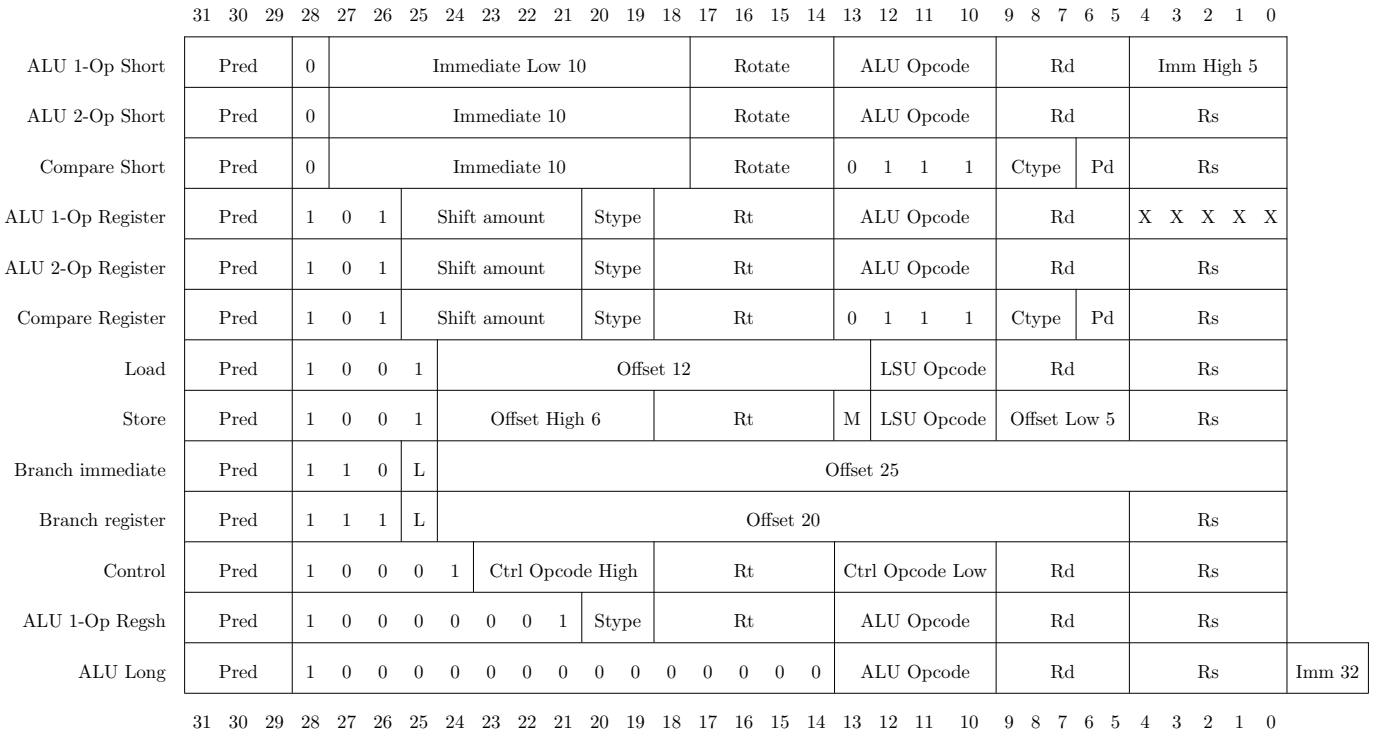
Interrupt Code	Type
00	Timer
01	USB
10	Framebuffer
11	SD Card

Table 1.2: Interrupt Codes

Chapter 2

Instruction Set Summary

2.1 Instruction Set Encoding



2.2 ALU Opcodes

Encoding	Mnemonic	Instruction	Action	Section(s)
0 0 0 0	`ADD`	Addition	`Rd := Rs + Op2`	3.1
0 0 0 1	`AND`	Bitwise And	`Rd := Rs & Op2`	3.2
0 0 1 0	`NOR`	Bitwise Logical NOR	`Rd := ~(Rs | Op2)`	3.28
0 0 1 1	`OR`	Bitwise Logical OR	`Rd := Rs | Op2`	3.29
0 1 0 0	`SUB`	Subtraction	`Rd := Rs - Op2`	3.34
0 1 0 1	`RSB`	Reverse Subtraction	`Rd := Op2 - Rs`	3.30
0 1 1 0	`XOR`	Bitwise Exclusive OR	`Rd := Rs ^ Op2`	3.39
0 1 1 1	`COMPARE`	Compare Instructions	`Pd := Rs CMP Op2`	3.6, 3.7, 3.8, 3.9, 3.10, 3.11, 3.12
1 0 0 0	`MOV`	Move	`Rd := Op2`	3.23
1 0 0 1	`MVN`	Move and Invert	`Rd := ~Op2`	3.26
1 0 1 0	`SXB`	Sign-Extend Byte	`Rd := SX Op2[7:0]`	3.36
1 0 1 1	`SXH`	Sign-Extend Half-Word	`Rd := SX Op2[15:0]`	3.37
1 1 0 0	reserved	n/a	n/a	n/a
1 1 0 1	reserved	n/a	n/a	n/a
1 1 1 0	reserved	n/a	n/a	n/a
1 1 1 1	reserved	n/a	n/a	n/a

Chapter 3

Instruction Listing

3.1 ADD - Addition

3.1.1 Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
PRED			0	CONSTANT														ROTATE				0	0	0	0	RD				RS		
PRED			1	0	1	SHIFTAMT					SHF		RT									0	0	0	0	RD				RS		
PRED			1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	RD				RS		LONG

3.1.2 Syntax

- `[[!]Pn ->] Rd <- Rs + Imm`
- `[[!]Pn ->] Rd <- Rs + Rt`
- `[[!]Pn ->] Rd <- Rs + (Rt SHF Imm)`

3.1.3 Type

ADD is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.1.4 Behavior

Adds the two operands together into the destination register. If two register operands are specified, the second may optionally be shifted by an immediate value before the addition is performed.

TODO optional overflow exceptions???

3.2 AND - Bitwise And

3.2.1 Encoding

31–28	27	26–21	20–17	16–15	14	13–8	7–4	3–0	
PRED	0	CONSTANT		ROTATE	0 0 0 1	RD	RS		
PRED	1 0 1	SHIFTAMT		SHF	RT	0 0 0 1	RD	RS	
PRED	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					0 0 0 1	RD	RS	LONG

3.2.2 Syntax

- `[[!]Pn ->] Rd <- Rs & Imm`
- `[[!]Pn ->] Rd <- Rs & Rt`
- `[[!]Pn ->] Rd <- Rs & (Rt SHF Imm)`

3.2.3 Type

AND is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.2.4 Behavior

Computes the bitwise logical AND of the two operands, storing the result in the destination register. If two register operands are specified, the second may optionally be shifted by an immediate value before the computation is performed.

3.3 B - Branch

3.3.1 Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PRED			1	1	0	0	OFFSET																								
PRED			1	1	1	0	OFFSET																					RS				

3.3.2 Syntax

- `[[!]Pn ->] B <label>`
- `[[!]Pn ->] B Rs + OFF`

3.3.3 Type

Branch is a Control operation. It may only be placed as the first instruction in a packet.

3.3.4 Behavior

Transfers control to the address specified. The `OFFSET` field is shifted left by 4 places and sign-extended to 32 bits, then added to either the source register or, if none is specified, the address of the branch instruction. If the specified base register is not aligned to a 16-byte value, the target is rounded down to the nearest aligned value before control is transferred. Other instructions in the same packet as the branch instruction are executed before control is transferred.

3.4 BL - Branch with Link

3.4.1 Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PRED			1	1	0	1	OFFSET																								
PRED			1	1	1	1	OFFSET																					RS			

3.4.2 Syntax

- `[[!]Pn ->] BL <label>`
- `[[!]Pn ->] BL Rs + OFF`

3.4.3 Type

Branch with Link is a Control operation. It may only be placed as the first instruction in a packet.

3.4.4 Behavior

BL functions identically to the Branch instruction, except that the address of the BL instruction is placed in R31.

3.5 BREAK

3.5.1 Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PRED			1	0	0	0	1	0	0	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

3.5.2 Syntax

- `[[!]Pn ->] BREAK`
- `[[!]Pn ->] BREAK Imm`

3.5.3 Behavior

Equivalent to SYSCALL, except that the exception code stored in EC0 is different - See Section 1.7.

3.6 CMPBC - Compare Bits Clear

3.6.1 Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
PRED			0	CONSTANT														ROTATE				0	1	1	1	1	1	1	PD		RS		
PRED			1	0	1	SHIFTAMT								SHF		RT						0	1	1	1	1	1	1	PD		RS		
PRED			1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	PD		RS		LONG

3.6.2 Syntax

- `[[!]Pn ->] Pd <- Rs BC Imm`
- `[[!]Pn ->] Pd <- Rs BC (Rt SHF Imm)`

3.6.3 Type

CMPBC is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.6.4 Behavior

CMPBC checks each bit of Rs corresponding to a 1 bit in the second operand, writing 1 into the destination predicate if all such bits are 0, and writing 0 otherwise – that is, it computes the unary OR of the bits in `(~Rs & OP2)` and stores the result in the desired predicate.
P3 is pinned to 1. Writes into P3 from compare instructions are ignored.

3.7 CMPBS - Compare Bits Set

3.7.1 Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
PRED				0	CONSTANT														ROTATE			0	1	1	1	1	1	0	PD		RS		
PRED				1	0	1	SHIFTAMT									SHF		RT				0	1	1	1	1	1	0	PD		RS		
PRED				1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	PD		RS		LONG

3.7.2 Syntax

- `[[!]Pn ->] Pd <- Rs BS Imm`
- `[[!]Pn ->] Pd <- Rs & Imm`
- `[[!]Pn ->] Pd <- Rs BS (Rt SHF Imm)`
- `[[!]Pn ->] Pd <- Rs & (Rt SHF Imm)`

3.7.3 Type

CMPBS is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.7.4 Behavior

CMPBS checks each bit of Rs corresponding to a 1 bit in the second operand, writing 1 into the destination predicate if any such bit is 1, and writing 0 otherwise – that is, it computes the unary OR of the bits in `(Rs & OP2)` and stores the result in the desired predicate.
P3 is pinned to 1. Writes into P3 from compare instructions are ignored.

3.8 CMPEQ - Compare Equal

3.8.1 Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
PRED			0	CONSTANT															ROTATE			0	1	1	1	0	1	0	PD		RS		
PRED			1	0	1	SHIFTAMT							SHF	RT								0	1	1	1	0	1	0	PD		RS		
PRED			1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	PD		RS		LONG

3.8.2 Syntax

- `[[!]Pn ->] Pd <- Rs == Imm`
- `[[!]Pn ->] Pd <- Rs == (Rt SHF Imm)`

3.8.3 Type

CMPEQ is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.8.4 Behavior

CMPEQ tests whether its two operands are equal, writing 1 into the destination predicate if they are and 0 otherwise.
P3 is pinned to 1. Writes into P3 from compare instructions are ignored.

3.9 CMPLES - Compare Less Than or Equal (Signed)

3.9.1 Encoding

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0	
PRED			0	CONSTANT														ROTATE				0	1	1	1	1	0	1	PD		RS		
PRED			1	0	1	SHIFTAMT								SHF		RT						0	1	1	1	1	0	1	PD		RS		
PRED			1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	PD		RS		LONG

3.9.2 Syntax

- `[[!]Pn ->] Pd <- Rs LES Imm`
- `[[!]Pn ->] Pd <- Rs <=s Imm`
- `[[!]Pn ->] Pd <- Rs LES (Rt SHF Imm)`
- `[[!]Pn ->] Pd <- Rs <=s (Rt SHF Imm)`

3.9.3 Type

CMPLES is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.9.4 Behavior

CMPLES interprets its operands as 32-bit two's-complement signed integers, and writes 1 into the specified predicate register if the value of `Rs` is less than or equal to the value of the second operand, and 0 otherwise.
P3 is pinned to 1. Writes into P3 from compare instructions are ignored.

3.10 CMPLEU - Compare Less Than or Equal (Unsigned)

3.10.1 Encoding

31–28	27	26–23	22–18	17–14	13	12–9	8–7	6–5	4–0	
PRED	0	CONSTANT			ROTATE		0 1 1 1	0 0 1	PD	RS	
PRED	1 0 1	SHIFTAMT			SHF	RT		0 1 1 1	0 0 1	PD	RS	
PRED	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0							0 1 1 1	0 0 1	PD	RS	LONG

3.10.2 Syntax

- `[[!]Pn ->] Pd <- Rs LEU Imm`
- `[[!]Pn ->] Pd <- Rs <= Imm`
- `[[!]Pn ->] Pd <- Rs <=u Imm`
- `[[!]Pn ->] Pd <- Rs LEU (Rt SHF Imm)`
- `[[!]Pn ->] Pd <- Rs <= (Rt SHF Imm)`
- `[[!]Pn ->] Pd <- Rs <=u (Rt SHF Imm)`

3.10.3 Type

CMPLEU is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.10.4 Behavior

CMPLEU interprets its operands as 32-bit unsigned integers, and writes 1 into the specified predicate register if the value of `Rs` is less than or equal to the value of the second operand, and 0 otherwise.
P3 is pinned to 1. Writes into P3 from compare instructions are ignored.

3.11 CMPLTS - Compare Less Than (Signed)

3.11.1 Encoding

31–28	27	26–23	22–18	17–13	12–9	8–5	4–0	
PRED	0	CONSTANT		ROTATE	0111	100	PD	RS	
PRED	101	SHIFTAMT		SHF	RT	0111	100	PD	RS	
PRED	100	0000	0000	0000	0000	0111	100	PD	RS	LONG

3.11.2 Syntax

- `[[!]Pn ->] Pd <- Rs LTS Imm`
- `[[!]Pn ->] Pd <- Rs <s Imm`
- `[[!]Pn ->] Pd <- Rs LTS (Rt SHF Imm)`
- `[[!]Pn ->] Pd <- Rs <s (Rt SHF Imm)`

3.11.3 Type

CMPLTS is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.11.4 Behavior

CMPLTS interprets its operands as 32-bit two's-complement signed integers, and writes 1 into the specified predicate register if the value of `Rs` is less than to the value of the second operand, and 0 otherwise.
P3 is pinned to 1. Writes into P3 from compare instructions are ignored.

3.12 CMPLTU - Compare Less Than (Unsigned)

3.12.1 Encoding

31–28	27	26–20	19–15	14–13	12–10	9–7	6–5	4–0	
PRED	0	CONSTANT		ROTATE	0111	000	PD	RS	
PRED	101	SHIFTAMT		SHF	RT	0111	000	PD	RS	
PRED	1000000000000000					0111	000	PD	RS	LONG

3.12.2 Syntax

- `[[!]Pn ->] Pd <- Rs LTU Imm`
- `[[!]Pn ->] Pd <- Rs < Imm`
- `[[!]Pn ->] Pd <- Rs <u Imm`
- `[[!]Pn ->] Pd <- Rs LTU (Rt SHF Imm)`
- `[[!]Pn ->] Pd <- Rs < (Rt SHF Imm)`
- `[[!]Pn ->] Pd <- Rs <u (Rt SHF Imm)`

3.12.3 Type

CMPLTU is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.12.4 Behavior

CMPLTU interprets its operands as 32-bit unsigned integers, and writes 1 into the specified predicate register if the value of `Rs` is less than to the value of the second operand, and 0 otherwise. P3 is pinned to 1. Writes into P3 from compare instructions are ignored.

3.13 DIV - Integer Division

3.13.1 Encoding

31–29	28	27	26	25	24	23	22	21	20	19–15	14	13	12	11	10–6	5–0
PRED	1	0	0	0	1	1	0	0	1	X	RT	X	X	X	X	RD	RS

3.13.2 Syntax

- `[[!]Pn ->] Rd <- Rs / Rt` (defaults to unsigned)
- `[[!]Pn ->] Rd <- Rs /s Rt`
- `[[!]Pn ->] Rd <- Rs /u Rt`

3.13.3 Type

DIV is a Control operation. It may only be placed as the first instruction in a packet.

3.13.4 Behavior

Performs the integer division `Rs / Rt`, writing the quotient into `Rd` and the remainder into the multiply/divide overflow register.

3.14 ERET - Return from Exception

3.14.1 Encoding

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
PRED			1	0	0	0	1	0	1	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

3.14.2 Syntax

- `[[!]Pn ->] ERET`

3.14.3 Type

ERET is a Control operation. It may only be placed as the first instruction in a packet. It may only be used when the processor is in kernel mode.

3.14.4 Behavior

ERET returns the processor to the state specified by the `EPC` coprocessor register. Specifically, the program counter takes on the value from `EPC[PC]`, the system mode flag takes on the value from `EPC[M]` (kernel mode if `M` is 1, otherwise user mode), and the interrupt enable flag takes on the value from `EPC[IF]` (interrupts enabled if `IF` is 1, otherwise disabled). Other instructions in the same packet as the ERET instruction are executed in kernel mode before control is transferred. This instruction also clears the memory link bit.

3.15 FENCE

3.15.1 Encoding

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
PRED			1	0	0	0	1	0	0	1	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

3.15.2 Syntax

- `[[!]Pn ->] FENCE`

3.15.3 Type

FENCE is a Control operation. It may only be placed as the first instruction in a packet.

3.15.4 Behavior

Our architecture doesn't reorder requests, so right now FENCE is a no-op. If we have to add behavior to it in the future, memory instructions in the same packet as the FENCE will probably go after it, but please don't count on that right now.

3.16 FLUSH - Flush L1 Caches

3.16.1 Encoding

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0
PRED			1	0	0	0	1	0	1	0	1	C	X	X	X	X	X	X	X	TYPE			X	X	X	X	X	RS		

3.16.2 Syntax

- `[[!]Pn ->] FLUSH.<TYPE> Rs`

3.16.3 Type

FLUSH is a Control operation. It may only be placed as the first instruction in a packet.

3.16.4 Behavior

FLUSH invalidates the line in the specified L1 cache corresponding to the virtual address contained in `Rs`. For instruction or data cache flushes, attempting to flush a virtual address whose access would cause a page fault will also cause a page fault. If a dirty data cache line is flushed, the new value is immediately written out to the L2 cache, where it will be visible to DMA peripherals. Values for the `TYPE` field are as follows:

Value	Mnemonic	Type
0 0	`DATA`	L1 Data Cache
0 1	`INST`	L1 Instruction Cache
1 0	`DTLB`	Data TLB
1 1	`ITLB`	Instruction TLB

3.17 LB - Load Byte

3.17.1 Encoding

31 30 29	28 27 26 25	24 23 22 21 20 19 18 17 16	15 14 13	12 11 10 9 8	7 6 5 4 3 2 1 0
PRED	1 0 0 1	OFFSET	0 0 0	RD	RS

3.17.2 Syntax

- `[[!]Pn ->] Rd <- *b(Rs [+ OFFSET])`

3.17.3 Type

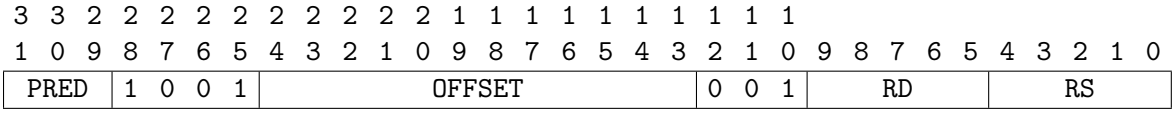
LB is a Memory operation. It may only be placed in the first or second slot in an instruction packet.

3.17.4 Behavior

LB loads a byte from the specified address into `Rd`. The address is computed by sign-extending the `OFFSET` field to 32 bits and adding it to the value of `Rs`. The high 24 bits of `Rd` are zeroed.

3.18 LH - Load Half-Word

3.18.1 Encoding



3.18.2 Syntax

- `[[!]Pn ->] Rd <- *h(Rs [+ OFFSET])`

3.18.3 Type

LH is a Memory operation. It may only be placed in the first or second slot in an instruction packet.

3.18.4 Behavior

LH loads two bytes from the specified address into `Rd`. The address is computed by sign-extending the `OFFSET` field to 32 bits and adding it to the value of `Rs`. Only 2-byte aligned addresses may be accessed - the low bit of the address is ignored. The high 16 bits of `Rd` are zeroed.

3.19 LL - Load Linked

3.19.1 Encoding

31 30 29	28 27 26 25	24 23 22 21 20 19 18 17 16 15 14 13 12 11 10	9 8 7	6 5 4	3 2 1 0
PRED	1 0 0 1	OFFSET	0 1 1	RD	RS

3.19.2 Syntax

- `[[!]Pn ->] Rd <- *ll(Rs [+ OFFSET])`

3.19.3 Type

LL is a Memory operation. It may only be placed in the first or second slot in an instruction packet.

3.19.4 Behavior

LL loads a word from the specified address into `Rd`. The address is computed by sign-extending the `OFFSET` field to 32 bits and adding it to the value of `Rs`. Only 4-byte aligned addresses may be accessed - the low 2 bits of the address are ignored.
Additionally, LL sets the link bit. A future SC operation will only complete if no interrupt, exception, or ERET clears the link bit in the intervening period.

3.20 LW - Load Word

3.20.1 Encoding

31 30 29	28 27 26 25	24 23 22 21 20 19 18 17 16	15 14 13	12 11 10	9 8 7 6 5
PRED	1 0 0 1	OFFSET	0 1 0	RD	RS

3.20.2 Syntax

- `[[!]Pn ->] Rd <- *l(Rs [+ OFFSET])`
- `[[!]Pn ->] Rd <- *w(Rs [+ OFFSET])`

3.20.3 Type

LW is a Memory operation. It may only be placed in the first or second slot in an instruction packet.

3.20.4 Behavior

LW loads a word from the specified address into `Rd`. The address is computed by sign-extending the `OFFSET` field to 32 bits and adding it to the value of `Rs`. Only 4-byte aligned addresses may be accessed - the low 2 bits of the address are ignored.

3.21 MFC - Move From Coprocessor Register

3.21.1 Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PRED			1	0	0	0	1	0	1	1	0	X	X	X	X	X	X	X	X	X	X	RD					CPRS				

3.21.2 Syntax

- `[[!]Pn ->] Rd <- CPRs`

3.21.3 Type

MFC is a Control operation. It may only be placed as the first instruction in a packet. It may only be used when the processor is in kernel mode.

3.21.4 Behavior

MFC moves the contents of the specfied coprocessor register into the specified general-purpose register. See Section 1.6for a description of the available coprocessor registers and their mnemonics.

3.22 MFHI - Move From Multiply/Division Overflow Register

3.22.1 Encoding

31 30 29	28 27 26 25 24 23 22 21 20 19	18 17 16 15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
PRED	1 0 0 0 1 1 0 1 0	X X X X X X X X X	RD	X X X X X

3.22.2 Syntax

- `[[!]Pn ->] Rd <- OVF`

3.22.3 Type

MFHI is a Control operation. It may only be placed as the first instruction in a packet.

3.22.4 Behavior

MFHI moves the contents of the multiply/division overflow register into the specified general-purpose register. This register stores the high bits of results of multiplication operations, or the remainders generated by division operations.

3.23 MOV - Move

3.23.1 Encoding

31–28	27	26–25	24–20	19–14	13–9	8–5	4–0	
PRED	0	CONSTLOW		ROTATE	1000	RD	CONSTHI	
PRED	101	SHIFTAMT		SHF	RT	1000	RD	XXXXX
PRED	1000001			SHF	RT	1000	RD	RS
PRED	1000000000000000					1000	RD	XXXXX	LONG

3.23.2 Syntax

- `[[!]Pn ->] Rd <- Imm`
- `[[!]Pn ->] Rd <- Rt`
- `[[!]Pn ->] Rd <- (Rt SHF Imm)`
- `[[!]Pn ->] Rd <- Rt SHF Rs`

3.23.3 Type

MOV is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.23.4 Behavior

MOV moves the value of its source operand into the specified destination register. If the second operand is a register, it may be shifted by an immediate value or by the contents of a register before being stored.

3.24 MTC - Move To Coprocessor Register

3.24.1 Encoding

31 30 29	28 27 26 25 24 23 22 21	20 19 18 17 16 15 14 13 12 11	10 9 8 7 6	4 3 2 1 0
PRED	1 0 0 0 1 0 1 1 1	X X X X X X X X X X	CPRD	RS

3.24.2 Syntax

- `[[!]Pn ->] CPRd <- Rs`

3.24.3 Type

MTC is a Control operation. It may only be placed as the first instruction in a packet. It may only be used when the processor is in kernel mode.

3.24.4 Behavior

MTC moves the contents of the selected general-purpose register into the specified coprocessor register. See Section 1.6 for a description of the available coprocessor registers and their mnemonics.

3.25 MTHI - Move To Multiply/Division Overflow Register

3.25.1 Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PRED			1	0	0	0	1	1	0	1	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	RS				

3.25.2 Syntax

- `[[!]Pn ->] OVF <- Rs`

3.25.3 Type

MTHI is a Control operation. It may only be placed as the first instruction in a packet.

3.25.4 Behavior

MTHI moves the value of the specified general-purpose register into the multiply/division overflow register.

3.26 MVN - Move and Invert

3.26.1 Encoding

31–28	27	26–21	20–15	14	13–12	11–8	7–4	3–0	
PRED	0	CONSTLOW			ROTATE	1001	RD	CONSTHI	
PRED	101	SHIFTAMT		SHF	RT	1001	RD	XXXXX	
PRED	1000001			SHF	RT	1001	RD	RS	
PRED	1000000000000000					1001	RD	XXXXX	LONG

3.26.2 Syntax

- `[[!]Pn ->] Rd <- ~Imm`
- `[[!]Pn ->] Rd <- ~Rt`
- `[[!]Pn ->] Rd <- ~(Rt SHF Imm)`
- `[[!]Pn ->] Rd <- ~(Rt SHF Rs)`

3.26.3 Type

MVN is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.26.4 Behavior

MVN takes the bitwise complement of its source operand, storing the result in the specified destination register. If the second operand is a register, it may be shifted by an immediate value or by the contents of a register before being inverted and stored.

3.27 MULT - Integer Multiplication

3.27.1 Encoding

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0
PRED			1	0	0	0	1	1	0	0	0	S	RT					X	X	X	X	RD					RS				

3.27.2 Syntax

- `[[!]Pn ->] Rd <- Rs * Rt` (defaults to unsigned)
- `[[!]Pn ->] Rd <- Rs *s Rt`
- `[[!]Pn ->] Rd <- Rs *u Rt`

3.27.3 Type

MULT is a Control operation. It may only be placed as the first instruction in a packet.

3.27.4 Behavior

MULT performs an integer multiplication on its two register operands. If the S bit is set, the operands are interpreted as two's complement signed integers; otherwise they are interpreted as unsigned. The low 32 bits of the 64 bit result are placed in the specified destination register, and the high 32 bits are placed in the special-purpose multiply/division overflow register, where they are accessible via the MFHI instruction.

3.28 NOR - Bitwise Logical NOR

3.28.1 Encoding

31–28	27	26–21	20–16	15–14	13–12	11–10	9–5	4–0	
PRED	0	CONSTANT		ROTATE	0 0 1 0		RD	RS	
PRED	1 0 1	SHIFTAMT		SHF	RT		0 0 1 0	RD	RS
PRED	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0						0 0 1 0	RD	RS	LONG

3.28.2 Syntax

- `[[!]Pn ->] Rd <- Rs ~| Imm`
- `[[!]Pn ->] Rd <- Rs NOR Imm`
- `[[!]Pn ->] Rd <- Rs ~| Rt`
- `[[!]Pn ->] Rd <- Rs NOR Rt`
- `[[!]Pn ->] Rd <- Rs ~| (Rt SHF Imm)`
- `[[!]Pn ->] Rd <- Rs NOR (Rt SHF Imm)`

3.28.3 Type

NOR is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.28.4 Behavior

Computes the bitwise logical NOR of the two operands, storing the result in the destination register. If two register operands are specified, the second may optionally be shifted by an immediate value before the computation is performed.

3.29 OR - Bitwise Logical OR

3.29.1 Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
PRED			0	CONSTANT														ROTATE		0	0	1	1	RD				RS				
PRED			1	0	1	SHIFTAMT					SHF	RT								0	0	1	1	RD				RS				
PRED			1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	RD				RS				LONG

3.29.2 Syntax

- `[[!]Pn ->] Rd <- Rs | Imm`
- `[[!]Pn ->] Rd <- Rs OR Imm`
- `[[!]Pn ->] Rd <- Rs | Rt`
- `[[!]Pn ->] Rd <- Rs OR Rt`
- `[[!]Pn ->] Rd <- Rs | (Rt SHF Imm)`
- `[[!]Pn ->] Rd <- Rs OR (Rt SHF Imm)`

3.29.3 Type

OR is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.29.4 Behavior

Computes the bitwise logical OR of the two operands, storing the result in the destination register. If two register operands are specified, the second may optionally be shifted by an immediate value before the computation is performed.

3.30 RSB - Reverse Subtraction

3.30.1 Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
PRED			0	CONSTANT															ROTATE			0	1	0	1	RD				RS		
PRED			1	0	1	SHIFTAMT					SHF	RT										0	1	0	1	RD				RS		
PRED			1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	RD				RS		LONG

3.30.2 Syntax

- `[[!]Pn ->] Rd <- Rs -: Imm`
- `[[!]Pn ->] Rd <- Rs RSB Imm`
- `[[!]Pn ->] Rd <- Rs -: Rt`
- `[[!]Pn ->] Rd <- Rs RSB Rt`
- `[[!]Pn ->] Rd <- Rs -: (Rt SHF Imm)`
- `[[!]Pn ->] Rd <- Rs RSB (Rt SHF Imm)`

3.30.3 Type

RSB is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.30.4 Behavior

Subtracts the value stored in RS from the value of the second operand, storing the result in the destination register. If two register operands are specified, the second may optionally be shifted by an immediate value before the computation is performed.

3.31 SB - Store Byte

3.31.1 Encoding

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0
PRED			1	0	0	1	OFFSETHIGH											RT				M	1	0	0	OFFSETLOW						

3.31.2 Syntax

- `[[!]Pn ->] *b(Rs [+ OFFSET]) <- Rt`

3.31.3 Type

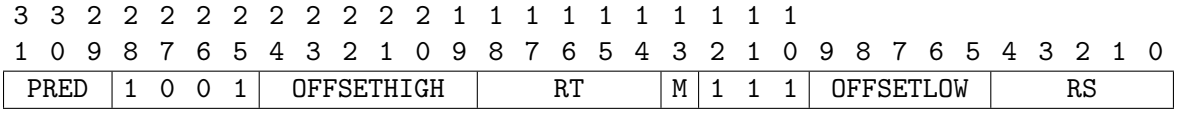
SB is a Memory operation. It may only be placed in the first or second slot in an instruction packet.

3.31.4 Behavior

SB stores the low byte of `Rt` at the provided address. The address is computed by sign-extending the `OFFSET` field to 32 bits and adding it to the value of `Rs`.
The `OFFSET` field for stores is broken up into 3 parts: `{inst[24:19], inst[13], inst[9:5]}`.

3.32 SC - Store Conditional

3.32.1 Encoding



31–28	27	26	25	24	23–19	18–13	12	11	10	9	8–3	2–0
PRED	1	0	0	1	OFFSETHIGH	RT	M	1	1	1	OFFSETLOW	RS

3.32.2 Syntax

- `[[!]Pn ->] *sc(Rs [+ OFFSET]) <- Rt`

3.32.3 Type

SC is a Memory operation. It may only be placed in the first or second slot in an instruction packet.

3.32.4 Behavior

SC stores the value in `Rt` at the provided address, if the processor's link bit is set. The address is computed by sign-extending the `OFFSET` field to 32 bits and adding it to the value of `Rs`. Only 4-byte aligned addresses may be accessed - the low 2 bits of the address are ignored.

The link bit is set by executing the LL instruction, and is cleared by the SC or ERET instructions, or any time an exception or interrupt is dispatched. SC only completes if the link bit is set when it is executed - thus, if it completes, all instructions since the last LL instruction are known to have been executed atomically. The SC instruction writes a 1 into predicate register 0 if the store completes successfully, or a 0 if it fails.

The `OFFSET` field for stores is broken up into 3 parts: `{inst[24:19], inst[13], inst[9:5]}`.

3.33 SH - Store Half-Word

3.33.1 Encoding

31–28	27	26	25	24	23–19	18–14	13	12	11	10	9–5	4–0
PRED	1	0	0	1	OFFSETHIGH	RT	M	1	0	1	OFFSETLOW	RS

3.33.2 Syntax

- `[[!]Pn ->] *h(Rs [+ OFFSET]) <- Rt`

3.33.3 Type

SH is a Memory operation. It may only be placed in the first or second slot in an instruction packet.

3.33.4 Behavior

SH stores the low two bytes of `Rt` at the provided address. The address is computed by sign-extending the `OFFSET` field to 32 bits and adding it to the value of `Rs`. The address must be aligned to a multiple of 2; the low bit of the address is ignored.
The `OFFSET` field for stores is broken up into 3 parts: `{inst[24:19], inst[13], inst[9:5]}`.

3.34 SUB - Subtraction

3.34.1 Encoding

31–28	27	26–25	24–23	22–19	18–13	12–7	6–5	4–0	
PRED	0	CONSTANT			ROTATE	0100	RD	RS	
PRED	101	SHIFTAMT		SHF	RT	0100	RD	RS	
PRED	1000	0000	0000	0000	0000	0100	RD	RS	LONG

3.34.2 Syntax

- `[[!]Pn ->] Rd <- Rs - Imm`
- `[[!]Pn ->] Rd <- Rs SUB Imm`
- `[[!]Pn ->] Rd <- Rs - Rt`
- `[[!]Pn ->] Rd <- Rs SUB Rt`
- `[[!]Pn ->] Rd <- Rs - (Rt SHF Imm)`
- `[[!]Pn ->] Rd <- Rs SUB (Rt SHF Imm)`

3.34.3 Type

SUB is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.34.4 Behavior

Subtracts the value of the second operand from the value in `Rs`, storing the result in the destination register. If two register operands are specified, the second may optionally be shifted by an immediate value before the computation is performed.

3.35 SW - Store Word

3.35.1 Encoding

31–28	27	26	25	24	23–19	18–14	13	12	11	10	9–5	4–0
PRED	1	0	0	1	OFFSETHIGH	RT	M	1	1	0	OFFSETLOW	RS

3.35.2 Syntax

- `[[!]Pn ->] *l(Rs [+ OFFSET]) <- Rt`
- `[[!]Pn ->] *w(Rs [+ OFFSET]) <- Rt`

3.35.3 Type

SW is a Memory operation. It may only be placed in the first or second slot in an instruction packet.

3.35.4 Behavior

SW stores the contents of `Rt` at the provided address. The address is computed by sign-extending the `OFFSET` field to 32 bits and adding it to the value of `Rs`. The address must be aligned to a multiple of 4; the low two bits of the address are ignored.
The `OFFSET` field for stores is broken up into 3 parts: `{inst[24:19], inst[13], inst[9:5]}`.

3.36 SXB - Sign-Extend Byte

3.36.1 Encoding

31–28	27	26–21	20–15	14–13	12–9	8–5	4–0	
PRED	0	CONSTLOW		ROTATE	1010	RD	CONSTHI	
PRED	101	SHIFTAMT		SHF	RT	1010	RD	XXXXX
PRED	1000001			SHF	RT	1010	RD	RS
PRED	1000000000000000					1010	RD	XXXXX	LONG

3.36.2 Syntax

- `[[!]Pn ->] Rd <- SXB Imm`
- `[[!]Pn ->] Rd <- SXB Rt`
- `[[!]Pn ->] Rd <- SXB (Rt SHF Imm)`
- `[[!]Pn ->] Rd <- SXB (Rt SHF Rs)`

3.36.3 Type

SXB is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.36.4 Behavior

SXB moves the low byte of its source operand into the specified destination register, sign-extending it across the rest of the register. If the second operand is a register, it may be shifted by an immediate value or by the contents of a register before being stored.

3.37 SXH - Sign-Extend Half-Word

3.37.1 Encoding

31–28	27	26–21	20–15	14	13–12	11–8	7–4	3–0	
PRED	0	CONSTLOW		ROTATE	1011	RD	CONSTHI		
PRED	101	SHIFTAMT		SHF	RT	1011	RD	XXXXX	
PRED	1000001			SHF	RT	1011	RD	RS	
PRED	1000000000000000					1011	RD	XXXXX	LONG

3.37.2 Syntax

- `[[!]Pn ->] Rd <- SXH Imm`
- `[[!]Pn ->] Rd <- SXH Rt`
- `[[!]Pn ->] Rd <- SXH (Rt SHF Imm)`
- `[[!]Pn ->] Rd <- SXH (Rt SHF Rs)`

3.37.3 Type

SXH is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.37.4 Behavior

SXH moves the low two bytes of its source operand into the specified destination register, sign-extending it across the rest of the register. If the second operand is a register, it may be shifted by an immediate value or by the contents of a register before being stored.

3.38 SYSCALL - System Call Exception

3.38.1 Encoding

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0
PRED			1	0	0	0	1	0	0	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

3.38.2 Syntax

- `[[!]Pn ->] SYSCALL`
- `[[!]Pn ->] SYSCALL Imm`

3.38.3 Behavior

SYSCALL causes an exception to occur on the instruction packet of which it is a part. Its corresponding error code is placed in the EC0 coprocessor register - see Section 1.7 for details. Note that this causes other instructions in the packet to be canceled.

3.39 XOR - Bitwise Exclusive OR

3.39.1 Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
PRED			0	CONSTANT														ROTATE			0	1	1	0	RD				RS				
PRED			1	0	1	SHIFTAMT					SHF	RT									0	1	1	0	RD				RS				
PRED			1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	RD				RS				LONG

3.39.2 Syntax

- `[[!]Pn ->] Rd <- Rs ^ Imm`
- `[[!]Pn ->] Rd <- Rs XOR Imm`
- `[[!]Pn ->] Rd <- Rs ^ Rt`
- `[[!]Pn ->] Rd <- Rs XOR Rt`
- `[[!]Pn ->] Rd <- Rs ^ (Rt SHF Imm)`
- `[[!]Pn ->] Rd <- Rs XOR (Rt SHF Imm)`

3.39.3 Type

XOR is an ALU operation. It may be placed in any slot of a packet. If the long immediate form is used it must not be located in the last slot in a packet, and no instruction may be specified in the following slot.

3.39.4 Behavior

Computes the bitwise logical XOR of the two operands, storing the result in the destination register. If two register operands are specified, the second may optionally be shifted by an immediate value before the computation is performed.