IE64 Processor User's Manual

IE64 64-bit RISC Processor Reference Manual

Last modified: 2026-07-07

Table of Contents

---

1. Architecture Overview

The IE64 is a 64-bit RISC load-store CPU. It uses a clean, regular instruction encoding with the following core characteristics:

- **Word size**: 64-bit registers, 64-bit data path
- **Instruction width**: Fixed 8 bytes (64 bits) per instruction
- **Byte order**: Little-endian throughout (instruction encoding, memory access, immediates)
- **Architecture class**: Load-store-style register machine (integer computation is register-register/register-immediate; memory is accessed by explicit memory, stack, FPU-memory, and atomic instructions)

- **Integer condition model**: Compare-and-branch (no integer flags register)
- **Register file**: 32 general-purpose 64-bit registers (R0 hardwired to zero)
- **Address space**: 64-bit effective addresses for instruction fetch, explicit memory operations, stack operations, FPU memory operations, atomics, and MMU translation. IE64 exposes the active CPU-visible RAM size through the read-only `CR_RAM_SIZE_BYTES` control register.
- **Stack**: Full-descending, R31 serves as stack pointer. Hardware enforces 8-byte granularity for PUSH/POP. Software that requires wider call-boundary alignment must maintain it explicitly.
- **Interrupt model**: Legacy single-vector timer interrupt, plus MMU trap/vector registers when the MMU is enabled

---

2. Register File

The IE64 has 32 general-purpose 64-bit registers, addressed by a 5-bit field (0-31).

Register	Alias	Description
R0	--	Hardwired zero. Reads always return 0. Writes are silently discarded.
R1-R30	--	General-purpose registers. 64-bit read/write.
R31	SP	Stack pointer. Used implicitly by PUSH, POP, JSR, RTS, RTI, and interrupt entry. Initialised to `0x9F000` on reset.

Reset clears all general-purpose registers, then sets both R31 and R30 to `STACK_START` (`0x9F000`). R31 is the architectural stack pointer; no native IE64 instruction gives R30 special semantics.

Floating Point Registers (F0-F15): - 16 dedicated 32-bit scalar registers for IEEE-754 single-precision floating point. - Double-precision operations use even/odd register pairs (`f0:f1` through `f14:f15`). - Accessed via dedicated FPU instructions (single-precision opcodes 0x60-0x7C and double-precision opcodes 0x80-0x97). - Initialised to 0.0 on reset.

Program Counter (PC): - 64-bit internal register, not directly addressable. - Full 64-bit PC. The CPU can address the full active visible RAM exposed by `CR_RAM_SIZE_BYTES`. - Initialised to `0x1000` (PROG_START) on reset. - Most sequential instructions advance `PC` by 8 bytes. - Control-transfer, trap, return, fault, interrupt, and stopped-state instructions use the PC behaviour defined by their individual instruction entries.

There is no integer flags register. All integer conditional branches use explicit register-register comparison within the branch instruction itself. FPU condition state is held separately in FPSR.

---

3. Instruction Encoding

Every IE64 instruction is exactly 8 bytes (64 bits), encoded in little-endian byte order.

3.1 Byte-Level Format

```
Byte:    0         1              2              3            4    5    6    7
       +--------+----------+----------+----------+------+------+------+------+
       | Opcode | Rd|Sz|X  | Rs|unused| Rt|unused|       imm32 (LE)         |
       +--------+----------+----------+----------+------+------+------+------+
Bits:   [7:0]    [7:3][2:1][0] [7:3][2:0] [7:3][2:0]    [31:0]
```

3.2 Field Definitions

Field	Byte	Bits	Width	Description
opcode	0	[7:0]	8	Instruction opcode
Rd	1	[7:3]	5	Destination register index (0-31)
Size	1	[2:1]	2	Operand size code
X	1	[0]	1	Operand mode: 0 = register Rt, 1 = immediate imm32
Rs	2	[7:3]	5	First source register index (0-31)
unused	2	[2:0]	3	Reserved; software should encode 0; the CPU ignores these bits
Rt	3	[7:3]	5	Second source register index (0-31)
unused	3	[2:0]	3	Reserved; software should encode 0; the CPU ignores these bits
imm32	4-7	[31:0]	32	32-bit immediate value (little-endian)

3.3 Field Extraction Formulas

```
rd   = byte1 >> 3           // upper 5 bits of byte 1
size = (byte1 >> 1) & 0x03  // bits [2:1] of byte 1
xbit = byte1 & 1            // bit [0] of byte 1
rs   = byte2 >> 3           // upper 5 bits of byte 2
rt   = byte3 >> 3           // upper 5 bits of byte 3
imm32 = bytes[4] | (bytes[5] << 8) | (bytes[6] << 16) | (bytes[7] << 24)
```

3.4 Encoding Helper

```
instr[0] = opcode
instr[1] = (rd << 3) | (size << 1) | xbit
instr[2] = rs << 3
instr[3] = rt << 3
instr[4..7] = imm32 (little-endian)
```

3.5 Size Codes

Code	Suffix	Width	Mask
0	`.B`	8-bit	`val & 0xFF`
1	`.W`	16-bit	`val & 0xFFFF`
2	`.L`	32-bit	`val & 0xFFFFFFFF`
3	`.Q`	64-bit	`val` (no mask)

If no size suffix is specified in assembly, the default is `.Q` (64-bit).

3.6 Third Operand Resolution

When X=0, the third operand is the value of register Rt: `operand3 = regs[rt]`

When X=1, the third operand is the immediate, zero-extended to 64 bits: `operand3 = uint64(imm32)`

---

4. Complete Instruction Reference

4.0 Instruction Entry Schema

Each instruction entry uses the following field schema.

Field	Meaning
Operation	Canonical mnemonic and brief operation class.
Assembler Syntax	Assembly-language form for the instruction encoding.
Attributes	Size, privilege level, memory access class, and fixed-width constraints.
Description	Instruction-specific behaviour, including register, memory, control-register, or trap state changes.
Condition Codes	Integer condition-code effect. IE64 has no integer flags register; entries that write FPU condition state describe that effect explicitly.
Instruction Format	Opcode byte and fixed fields within the 8-byte instruction.
Instruction Fields	Meaning of `Rd`, `Rs`, `Rt`, size, `X`, and `imm32` for the entry.
Exceptions	Faults or traps architecturally raised by the instruction.
Notes	Architectural aliases, operand restrictions, or compatibility details.

Notation used in the `Operation` column is source-neutral pseudocode: `Rd`, `Rs`, and `Rt` are the register fields from bytes 1-3; `imm32` is the little-endian immediate in bytes 4-7; `maskToSize(x, s)` keeps the low bits selected by the size field; and `signExtend32to64(x)` extends a 32-bit signed quantity to the 64-bit register width. Rows describe the architectural effect. Instruction aliases are described only when they name the same encoded CPU behaviour.

4.1 Data Movement

1. MOVE - move.s Rd, Rs

Operation: `Rd = maskToSize(Rs, s)`.

Assembler Syntax: `move.s Rd, Rs`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The selected-size value in `Rs` is copied to `Rd`; bits outside the selected size are cleared.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x01`.

Instruction Fields: Byte 0 holds opcode `0x01`. Byte 1 bits 7-3 select `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`. Byte 3 and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: None.

2. MOVE - move.s Rd, #imm

Operation: `Rd = maskToSize(imm32, s)`.

Assembler Syntax: `move.s Rd, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The immediate field is masked to the selected size and loaded into `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x01`.

Instruction Fields: Byte 0 holds opcode `0x01`. Byte 1 bits 7-3 select `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Bytes 2-3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

3. MOVT - movt Rd, #imm

Operation: `Rd = (Rd & 0x00000000FFFFFFFF) \| (imm32 << 32)`.

Assembler Syntax: `movt Rd, #imm`.

Attributes: Memory: N; Size: --.

Description: The immediate field is placed in the upper 32 bits of `Rd`; the low 32 bits of `Rd` are preserved.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x02`.

Instruction Fields: Byte 0 holds opcode `0x02`. Byte 1 bits 7-3 select destination register `Rd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Bytes 2-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order; this field supplies the new high 32 bits of `Rd`.

Exceptions: None.

Notes: None.

4. MOVEQ - moveq Rd, #imm

Operation: `Rd = signExtend32to64(imm32)`.

Assembler Syntax: `moveq Rd, #imm`.

Attributes: Memory: N; Size: --.

Description: The 32-bit immediate is sign-extended to 64 bits and loaded into `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x03`.

Instruction Fields: Byte 0 holds opcode `0x03`. Byte 1 bits 7-3 select destination register `Rd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Bytes 2-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold `imm32` in little-endian order; the processor sign-extends this field to 64 bits.

Exceptions: None.

Notes: None.

5. LEA - lea Rd, disp(Rs)

Operation: `Rd = Rs + signExtend32to64(imm32)`.

Assembler Syntax: `lea Rd, disp(Rs)`.

Attributes: Memory: N; Size: --.

Description: The sign-extended immediate is added to `Rs`, and the 64-bit result is loaded into `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x04`.

Instruction Fields: Byte 0 holds opcode `0x04`. Byte 1 bits 7-3 select destination register `Rd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select base register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold signed `imm32` displacement in little-endian order.

Exceptions: None.

Notes: None.

4.2 Load/Store

6. LOAD - load.s Rd, (Rs)

Operation: `Rd = zeroExtend(mem[Rs], s)`.

Assembler Syntax: `load.s Rd, (Rs)`.

Attributes: Memory: Read; Size: B/W/L/Q.

Description: The effective address `Rs` is read using the selected size, zero-extended, and loaded into `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x10`.

Instruction Fields: Byte 0 holds opcode `0x10`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); byte 1 bit 0 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select base register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold signed `imm32` displacement in little-endian order; encode zero for `(Rs)`.

Exceptions: If MMU translation is enabled, reads can trap with cause 0 (`FAULT_NOT_PRESENT`), cause 1 (`FAULT_READ_DENIED`), or cause 10 (`FAULT_SKAC`). After optional MMU translation, physical backing is checked; a physical read outside implemented CPU-visible memory raises cause 0.

Notes: None.

7. LOAD - load.s Rd, disp(Rs)

Operation: `Rd = zeroExtend(mem[Rs + signExtend(disp)], s)`.

Assembler Syntax: `load.s Rd, disp(Rs)`.

Attributes: Memory: Read; Size: B/W/L/Q.

Description: The effective address `Rs + signExtend(disp)` is read using the selected size, zero-extended, and loaded into `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x10`.

Instruction Fields: Byte 0 holds opcode `0x10`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); byte 1 bit 0 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select base register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold signed `imm32` displacement in little-endian order; encode zero for `(Rs)`.

Exceptions: If MMU translation is enabled, reads can trap with cause 0 (`FAULT_NOT_PRESENT`), cause 1 (`FAULT_READ_DENIED`), or cause 10 (`FAULT_SKAC`). After optional MMU translation, physical backing is checked; a physical read outside implemented CPU-visible memory raises cause 0.

Notes: None.

8. STORE - store.s Rd, (Rs)

Operation: `mem[Rs] = maskToSize(Rd, s)`.

Assembler Syntax: `store.s Rd, (Rs)`.

Attributes: Memory: Write; Size: B/W/L/Q.

Description: The selected-size value in `Rd` is written to the effective address `Rs`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x11`.

Instruction Fields: Byte 0 holds opcode `0x11`. Byte 1 bits 7-3 select source register `Rd`; bits 2-1 select size (B/W/L/Q); byte 1 bit 0 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select base register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold signed `imm32` displacement in little-endian order; encode zero for `(Rs)`.

Exceptions: If MMU translation is enabled, writes can trap with cause 0 (`FAULT_NOT_PRESENT`), cause 2 (`FAULT_WRITE_DENIED`), or cause 10 (`FAULT_SKAC`). After optional MMU translation, physical backing is checked; a physical write outside implemented CPU-visible memory raises cause 0.

Notes: None.

9. STORE - store.s Rd, disp(Rs)

Operation: `mem[Rs + signExtend(disp)] = maskToSize(Rd, s)`.

Assembler Syntax: `store.s Rd, disp(Rs)`.

Attributes: Memory: Write; Size: B/W/L/Q.

Description: The selected-size value in `Rd` is written to the effective address `Rs + signExtend(disp)`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x11`.

Instruction Fields: Byte 0 holds opcode `0x11`. Byte 1 bits 7-3 select source register `Rd`; bits 2-1 select size (B/W/L/Q); byte 1 bit 0 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select base register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold signed `imm32` displacement in little-endian order; encode zero for `(Rs)`.

Exceptions: If MMU translation is enabled, writes can trap with cause 0 (`FAULT_NOT_PRESENT`), cause 2 (`FAULT_WRITE_DENIED`), or cause 10 (`FAULT_SKAC`). After optional MMU translation, physical backing is checked; a physical write outside implemented CPU-visible memory raises cause 0.

Notes: None.

Memory Access Widths

Size	Transfer Width	Architectural Memory Access
`.B`	1 byte	One 8-bit little-endian access
`.W`	2 bytes	One 16-bit little-endian access
`.L`	4 bytes	One 32-bit little-endian access
`.Q`	8 bytes	One 64-bit little-endian access

> **64-bit memory access**: `.Q` loads and stores transfer one little-endian 64-bit quantity at the effective address. The CPU ISA defines the effective address, transfer width, and register result. Address validity, translation, protection, and fault behaviour are defined by the CPU memory and MMU rules.

4.3 Arithmetic

10. ADD - add.s Rd, Rs, Rt

Operation: `Rd = maskToSize(Rs + Rt, s)`.

Assembler Syntax: `add.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor adds `Rt` and `Rs`, masks the sum to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x20`.

Instruction Fields: Byte 0 holds opcode `0x20`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: None.

11. ADD - add.s Rd, Rs, #imm

Operation: `Rd = maskToSize(Rs + imm32, s)`.

Assembler Syntax: `add.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor adds the immediate field and `Rs`, masks the sum to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x20`.

Instruction Fields: Byte 0 holds opcode `0x20`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

12. SUB - sub.s Rd, Rs, Rt

Operation: `Rd = maskToSize(Rs - Rt, s)`.

Assembler Syntax: `sub.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor subtracts `Rt` and `Rs`, masks the difference to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x21`.

Instruction Fields: Byte 0 holds opcode `0x21`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: None.

13. SUB - sub.s Rd, Rs, #imm

Operation: `Rd = maskToSize(Rs - imm32, s)`.

Assembler Syntax: `sub.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor subtracts the immediate field and `Rs`, masks the difference to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x21`.

Instruction Fields: Byte 0 holds opcode `0x21`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

14. MULU - mulu.s Rd, Rs, Rt

Operation: `Rd = maskToSize(Rs * Rt, s)` (unsigned).

Assembler Syntax: `mulu.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor multiplies `Rs` by `Rt` as unsigned integers, masks the product to the selected size, and writes the result to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x22`.

Instruction Fields: Byte 0 holds opcode `0x22`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: None.

15. MULU - mulu.s Rd, Rs, #imm

Operation: `Rd = maskToSize(Rs * imm32, s)` (unsigned).

Assembler Syntax: `mulu.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor multiplies `Rs` by the zero-extended immediate as unsigned integers, masks the product to the selected size, and writes the result to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x22`.

Instruction Fields: Byte 0 holds opcode `0x22`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

16. MULS - muls.s Rd, Rs, Rt

Operation: `Rd = maskToSize(int64(Rs) * int64(Rt), s)` (signed).

Assembler Syntax: `muls.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor multiplies `Rs` by `Rt` as signed integers, masks the product to the selected size, and writes the result to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x23`.

Instruction Fields: Byte 0 holds opcode `0x23`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: None.

17. MULS - muls.s Rd, Rs, #imm

Operation: `Rd = maskToSize(int64(Rs) * int64(uint64(imm32)), s)` (signed).

Assembler Syntax: `muls.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor multiplies `Rs` by the zero-extended `imm32` converted to `int64`, applies the selected size only when masking the product written to `Rd`, and writes the result to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x23`.

Instruction Fields: Byte 0 holds opcode `0x23`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

18. DIVU - divu.s Rd, Rs, Rt

Operation: If `Rt == 0`, `Rd = 0`; otherwise `Rd = maskToSize(Rs / Rt, s)` (unsigned).

Assembler Syntax: `divu.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor divides `Rs` by `Rt` as unsigned integers, masks the quotient to the selected size, and writes the result to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x24`.

Instruction Fields: Byte 0 holds opcode `0x24`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: If the divisor is zero and `Rd` is not `R0`, `Rd` receives zero.

19. DIVU - divu.s Rd, Rs, #imm

Operation: If `imm32 == 0`, `Rd = 0`; otherwise `Rd = maskToSize(Rs / imm32, s)` (unsigned).

Assembler Syntax: `divu.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor divides `Rs` by the zero-extended immediate as unsigned integers, masks the quotient to the selected size, and writes the result to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x24`.

Instruction Fields: Byte 0 holds opcode `0x24`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: If the divisor is zero and `Rd` is not `R0`, `Rd` receives zero.

20. DIVS - divs.s Rd, Rs, Rt

Operation: If `Rt == 0`, `Rd = 0`; otherwise `Rd = maskToSize(int64(Rs) / int64(Rt), s)` (signed).

Assembler Syntax: `divs.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor divides `Rs` by `Rt` as signed integers, masks the quotient to the selected size, and writes the result to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x25`.

Instruction Fields: Byte 0 holds opcode `0x25`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: If the divisor is zero and `Rd` is not `R0`, `Rd` receives zero. `DIVS` interprets `Rs` and the register divisor as full 64-bit signed integers and applies the selected size only when masking the quotient written to `Rd`. For `.b`, `.w`, and `.l`, the operands are not sign-extended from the selected width before division.

21. DIVS - divs.s Rd, Rs, #imm

Operation: If `imm32 == 0`, `Rd = 0`; otherwise `Rd = maskToSize(int64(Rs) / int64(uint64(imm32)), s)` (signed).

Assembler Syntax: `divs.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor divides `Rs` by the zero-extended `imm32` converted to `int64`, applies the selected size only when masking the quotient written to `Rd`, and writes the result to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x25`.

Instruction Fields: Byte 0 holds opcode `0x25`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: If the divisor is zero and `Rd` is not `R0`, `Rd` receives zero. `DIVS` interprets `Rs` as a full 64-bit signed integer and the immediate divisor as zero-extended `imm32` converted to `int64`; the selected size is applied only when masking the quotient written to `Rd`. For `.b`, `.w`, and `.l`, the operands are not sign-extended from the selected width before division. `MODS` differs: it sign-extends both operands to the selected width before taking the remainder.

22. MOD - mod.s Rd, Rs, Rt

Operation: If `Rt == 0`, `Rd = 0`; otherwise `Rd = maskToSize(Rs % Rt, s)` (unsigned).

Assembler Syntax: `mod.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor divides `Rs` by `Rt` as unsigned integers and writes the selected-size remainder to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x26`.

Instruction Fields: Byte 0 holds opcode `0x26`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: If the divisor is zero and `Rd` is not `R0`, `Rd` receives zero.

23. MOD - mod.s Rd, Rs, #imm

Operation: If `imm32 == 0`, `Rd = 0`; otherwise `Rd = maskToSize(Rs % imm32, s)` (unsigned).

Assembler Syntax: `mod.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor divides `Rs` by the zero-extended immediate as unsigned integers and writes the selected-size remainder to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x26`.

Instruction Fields: Byte 0 holds opcode `0x26`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: If the divisor is zero and `Rd` is not `R0`, `Rd` receives zero.

24. NEG - neg.s Rd, Rs

Operation: `Rd = maskToSize(-int64(Rs), s)`.

Assembler Syntax: `neg.s Rd, Rs`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor negates `Rs` as a signed integer, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x27`.

Instruction Fields: Byte 0 holds opcode `0x27`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); byte 1 bit 0 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

25. MODS - mods.s Rd, Rs, Rt/#imm

Operation: Let `D` be `Rt` for the register form or `imm32` for the immediate form, sign-extended to the selected size. If `D == 0`, `Rd = 0`; otherwise `Rd = maskToSize(signExtend(Rs, s) % D, s)`.

Assembler Syntax: `mods.s Rd, Rs, Rt/#imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor computes signed remainder using truncation-toward-zero division semantics and writes the selected-size result to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x28`.

Instruction Fields: Byte 0 holds opcode `0x28`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 selects the third operand source: 0 selects `Rt`, 1 selects `imm32`. Byte 2 bits 7-3 select source register `Rs`. When bit 0 is 0, byte 3 bits 7-3 select source register `Rt` and bytes 4-7 are ignored. When bit 0 is 1, byte 3 is ignored and bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: If the divisor is zero and `Rd` is not `R0`, `Rd` receives zero.

26. MULHU - mulhu Rd, Rs, Rt/#imm

Operation: Upper 64 bits of unsigned `Rs * operand3`.

Assembler Syntax: `mulhu Rd, Rs, Rt/#imm`.

Attributes: Memory: N; Size: Q.

Description: The processor multiplies `Rs` by the third operand as unsigned integers and writes the upper 64 bits of the product to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x29`.

Instruction Fields: Byte 0 holds opcode `0x29`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select quadword size (Q); bit 0 selects the third operand source: 0 selects `Rt`, 1 selects `imm32`. Byte 2 bits 7-3 select source register `Rs`. When bit 0 is 0, byte 3 bits 7-3 select source register `Rt` and bytes 4-7 are ignored. When bit 0 is 1, byte 3 is ignored and bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

27. MULHS - mulhs Rd, Rs, Rt/#imm

Operation: Upper 64 bits of signed `int64(Rs) * int64(operand3)`.

Assembler Syntax: `mulhs Rd, Rs, Rt/#imm`.

Attributes: Memory: N; Size: Q.

Description: The processor multiplies `Rs` by the third operand and writes the upper 64 bits of the signed 128-bit product to `Rd`. Register operands use the full 64-bit value in `Rt`; immediate operands use zero-extended `imm32` converted to `int64`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x2A`.

Instruction Fields: Byte 0 holds opcode `0x2A`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select quadword size (Q); bit 0 selects the third operand source: 0 selects `Rt`, 1 selects `imm32`. Byte 2 bits 7-3 select source register `Rs`. When bit 0 is 0, byte 3 bits 7-3 select source register `Rt` and bytes 4-7 are ignored. When bit 0 is 1, byte 3 is ignored and bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

Immediate operands: ALU immediates are zero-extended to 64 bits before the operation. For signed immediate forms this means `#0xFFFFFFFF` is the positive value 4294967295 for MULS/DIVS `.q`; use MOVEQ or a register operand when a negative signed operand is required. MODS sign-extends the selected operand width after this zero-extension, so `.b`/`.w`/`.l` immediates are interpreted at the selected width.

Division by zero: If the divisor (Rt or imm32) is zero, the result is 0 (no exception raised). This applies to DIVU, DIVS, MOD, and MODS.

NEG is a 2-operand instruction: it reads Rs and writes the two's complement negation to Rd.

4.4 Logical

28. AND - and.s Rd, Rs, Rt

Operation: `Rd = maskToSize(Rs & Rt, s)`.

Assembler Syntax: `and.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor performs a bitwise AND of `Rs` and `Rt`, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x30`.

Instruction Fields: Byte 0 holds opcode `0x30`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: None.

29. AND - and.s Rd, Rs, #imm

Operation: `Rd = maskToSize(Rs & imm32, s)`.

Assembler Syntax: `and.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor performs a bitwise AND of `Rs` and the immediate field, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x30`.

Instruction Fields: Byte 0 holds opcode `0x30`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

30. OR - or.s Rd, Rs, Rt

Operation: `Rd = maskToSize(Rs \| Rt, s)`.

Assembler Syntax: `or.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor performs a bitwise OR of `Rs` and `Rt`, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x31`.

Instruction Fields: Byte 0 holds opcode `0x31`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: None.

31. OR - or.s Rd, Rs, #imm

Operation: `Rd = maskToSize(Rs \| imm32, s)`.

Assembler Syntax: `or.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor performs a bitwise OR of `Rs` and the immediate field, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x31`.

Instruction Fields: Byte 0 holds opcode `0x31`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

32. EOR - eor.s Rd, Rs, Rt

Operation: `Rd = maskToSize(Rs ^ Rt, s)`.

Assembler Syntax: `eor.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor performs a bitwise exclusive-OR of `Rs` and `Rt`, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x32`.

Instruction Fields: Byte 0 holds opcode `0x32`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: None.

33. EOR - eor.s Rd, Rs, #imm

Operation: `Rd = maskToSize(Rs ^ imm32, s)`.

Assembler Syntax: `eor.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor performs a bitwise exclusive-OR of `Rs` and the immediate field, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x32`.

Instruction Fields: Byte 0 holds opcode `0x32`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

34. NOT - not.s Rd, Rs

Operation: `Rd = maskToSize(~Rs, s)`.

Assembler Syntax: `not.s Rd, Rs`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The processor inverts `Rs`, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x33`.

Instruction Fields: Byte 0 holds opcode `0x33`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); byte 1 bit 0 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

NOT is a 2-operand instruction. It performs bitwise complement of Rs, masked to the specified size, and stores the result in Rd.

4.5 Shifts

35. LSL - lsl.s Rd, Rs, Rt

Operation: `Rd = maskToSize(Rs << (Rt & 63), s)`.

Assembler Syntax: `lsl.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The shift count is taken from `Rt` masked to six bits. The processor shifts `Rs` left, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x34`.

Instruction Fields: Byte 0 holds opcode `0x34`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: None.

36. LSL - lsl.s Rd, Rs, #imm

Operation: `Rd = maskToSize(Rs << (imm32 & 63), s).`

Assembler Syntax: `lsl.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The shift count is taken from the immediate field masked to six bits. The processor shifts `Rs` left, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x34`.

Instruction Fields: Byte 0 holds opcode `0x34`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

37. LSR - lsr.s Rd, Rs, Rt

Operation: `Rd = maskToSize(Rs >> (Rt & 63), s).`

Assembler Syntax: `lsr.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The shift count is taken from `Rt` masked to six bits. The processor shifts `Rs` right logically, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x35`.

Instruction Fields: Byte 0 holds opcode `0x35`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: None.

38. LSR - lsr.s Rd, Rs, #imm

Operation: `Rd = maskToSize(Rs >> (imm32 & 63), s)`.

Assembler Syntax: `lsr.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The shift count is taken from the immediate field masked to six bits. The processor shifts `Rs` right logically, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x35`.

Instruction Fields: Byte 0 holds opcode `0x35`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

39. ASR - asr.s Rd, Rs, Rt

Operation: `Rd = maskToSize(signedRs >> (Rt & 63), s)`.

Assembler Syntax: `asr.s Rd, Rs, Rt`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The shift count is taken from `Rt` masked to six bits. The processor shifts `Rs` right arithmetically, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x36`.

Instruction Fields: Byte 0 holds opcode `0x36`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 0 for the register form. Byte 2 bits 7-3 select source register `Rs`; byte 3 bits 7-3 select source register `Rt`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this form and ignored by the processor.

Exceptions: None.

Notes: None.

40. ASR - asr.s Rd, Rs, #imm

Operation: `Rd = maskToSize(signedRs >> (imm32 & 63), s)`.

Assembler Syntax: `asr.s Rd, Rs, #imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The shift count is taken from the immediate field masked to six bits. The processor shifts `Rs` right arithmetically, masks the result to the selected size, and writes it to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x36`.

Instruction Fields: Byte 0 holds opcode `0x36`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 is encoded as 1 for the immediate form. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 and byte

3 are reserved by this form and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

41. CLZ - clz.l Rd, Rs

Operation: `Rd = LeadingZeros32(uint32(Rs)).`

Assembler Syntax: `clz.l Rd, Rs`.

Attributes: Memory: N; Size: L.

Description: The processor counts leading zero bits in the low 32 bits of `Rs` and writes the count to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x37`.

Instruction Fields: Byte 0 holds opcode `0x37`. Byte 1 bits 7-3 select destination register `Rd`. Byte 1 bits 2-1 are reserved by this instruction and ignored by the processor; the operation always uses the low 32 bits of `Rs`. Byte 1 bit 0 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

42. SEXT - sext.s Rd, Rs

Operation: Sign-extend byte/word/long source to 64 bits.

Assembler Syntax: `sext.s Rd, Rs`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The selected byte, word, or long source value is sign-extended to 64 bits and written to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x38`.

Instruction Fields: Byte 0 holds opcode `0x38`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); byte 1 bit 0 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

43. ROL - rol.s Rd, Rs, Rt/#imm

Operation: Rotate left within the selected width.

Assembler Syntax: `rol.s Rd, Rs, Rt/#imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The selected-width value is rotated left and written to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x39`.

Instruction Fields: Byte 0 holds opcode `0x39`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 selects the third operand source: 0 selects `Rt`, 1 selects `imm32`. Byte 2 bits 7-3 select source register `Rs`. When bit 0 is 0, byte 3 bits 7-3 select source register `Rt` and bytes 4-7 are ignored. When bit 0 is 1, byte 3 is ignored and bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

44. ROR - ror.s Rd, Rs, Rt/#imm

Operation: Rotate right within the selected width.

Assembler Syntax: `ror.s Rd, Rs, Rt/#imm`.

Attributes: Memory: N; Size: B/W/L/Q.

Description: The selected-width value is rotated right and written to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x3A`.

Instruction Fields: Byte 0 holds opcode `0x3A`. Byte 1 bits 7-3 select destination register `Rd`; bits 2-1 select size (B/W/L/Q); bit 0 selects the third operand source: 0 selects `Rt`, 1 selects `imm32`. Byte 2 bits 7-3 select source register `Rs`. When bit 0 is 0, byte 3 bits 7-3 select source register `Rt` and bytes 4-7 are ignored. When bit 0 is 1, byte 3 is ignored and bytes 4-7 hold unsigned `imm32` in little-endian order.

Exceptions: None.

Notes: None.

45. CTZ - ctz.l Rd, Rs

Operation: `Rd = TrailingZeros32(uint32(Rs))`.

Assembler Syntax: `ctz.l Rd, Rs`.

Attributes: Memory: N; Size: L.

Description: The processor counts trailing zero bits in the low 32 bits of `Rs` and writes the count to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x3B`.

Instruction Fields: Byte 0 holds opcode `0x3B`. Byte 1 bits 7-3 select destination register `Rd`. Byte 1 bits 2-1 are reserved by this instruction and ignored by the processor; the operation always uses the low 32 bits of `Rs`. Byte 1 bit 0 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

46. POPCNT - popcnt.l Rd, Rs

Operation: `Rd = OnesCount32(uint32(Rs))`.

Assembler Syntax: `popcnt.l Rd, Rs`.

Attributes: Memory: N; Size: L.

Description: The processor counts one bits in the low 32 bits of `Rs` and writes the count to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x3C`.

Instruction Fields: Byte 0 holds opcode `0x3C`. Byte 1 bits 7-3 select destination register `Rd`. Byte 1 bits 2-1 are reserved by this instruction and ignored by the processor; the operation always uses the low 32 bits of `Rs`. Byte 1 bit 0 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

47. BSWAP - bswap.l Rd, Rs

Operation: `Rd = ReverseBytes32(uint32(Rs)).`

Assembler Syntax: `bswap.l Rd, Rs`.

Attributes: Memory: N; Size: L.

Description: The processor reverses byte order in the low 32 bits of `Rs` and writes the result to `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x3D`.

Instruction Fields: Byte 0 holds opcode `0x3D`. Byte 1 bits 7-3 select destination register `Rd`. Byte 1 bits 2-1 are reserved by this instruction and ignored by the processor; the operation always uses the low 32 bits of `Rs`. Byte 1 bit 0 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

Shift amount masking: The shift count is always masked to 6 bits (`& 63`), limiting the effective shift range to 0-63.

CLZ (Count Leading Zeros): A 2-operand instruction that counts the number of leading zero bits in the low 32 bits of Rs and stores the result in Rd. The result is an integer in the range 0-32: zero if the most-significant bit is set, 32 if the input is zero. Only the `.l` size suffix is supported. Writing to R0 is silently discarded (as with all instructions).

This instruction is useful for O(1) floating-point normalisation, integer $\log_2$ computation, and highest-set-bit detection.

ASR sign extension: Before performing the arithmetic right shift, the source value is sign-extended according to the current size:

Size	Sign-extension
`.B`	`int64(int8(Rs))`
`.W`	`int64(int16(Rs))`
`.L`	`int64(int32(Rs))`
`.Q`	`int64(Rs)`

The result is then masked to the specified size after the shift.

LSL and LSR operate on the unsigned 64-bit value in Rs. The result is masked to the specified size.

ROL and ROR first mask the source to the selected size, rotate within that width, then mask the result again. The effective rotate count is masked to 3 bits for `.B`, 4 bits for `.W`, 5 bits for `.L`, and 6 bits for `.Q`.

CTZ, POPCNT, and BSWAP are 2-operand `.l`-only instructions. CTZ and POPCNT operate on `uint32(Rs)`. BSWAP stores `bits.ReverseBytes32(uint32(Rs))` zero-extended to 64 bits.

4.6 Floating Point (FPU)

The IE64 FPU provides native single-precision (`f*`) and double-precision (`d*`) IEEE-754 operations. Single-precision values use the 16 scalar registers `f0-f15`. Double-precision instruction encodings name the even-numbered F register of the even-odd register pair (`f0`, `f2`, ..., `f14`). The conceptual pair names below describe storage:

- `d0` = `f0:f1`
- `d1` = `f2:f3`
- `d2` = `f4:f5`
- `d3` = `f6:f7`
- `d4` = `f8:f9`
- `d5` = `f10:f11`
- `d6` = `f12:f13`
- `d7` = `f14:f15`

All `d*` mnemonics require even-numbered FP operands. Odd FP operand encodings are invalid. Writing a double clobbers both halves of the pair.

4.6.1 FPU Data Movement

48. FMOV - fmov fd, fs

Operation: `fd = fs` (FP copy).

Assembler Syntax: `fmov fd, fs`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd` and `fs` encode `f0-f15`. FPSR/FPCR: neither FPSR nor FPCR is read or written.

Description: The processor copies the single-precision value in `fs` to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x60`.

Instruction Fields: Byte 0 holds opcode `0x60`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

49. FLOAD - fload fd, disp(rs)

Operation: `fd = mem32[rs + disp]`.

Assembler Syntax: `fload fd, disp(rs)`.

Attributes: Memory: 32-bit read. Operand size: single precision. Privilege: unprivileged. FP operands: `fd` encodes `f0-f15`. Integer operand: `rs` supplies the base address. FPSR/FPCR: writes FPSR condition-code bits from the loaded value; FPCR is not read.

Description: The processor evaluates `mem32[rs + disp]` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x61`.

Instruction Fields: Byte 0 holds opcode `0x61`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select integer base register `rs`; byte 2 bits 2-0 and byte 3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold signed `imm32` displacement in little-endian order for the 32-bit memory transfer.

Exceptions: An invalid `fd` encoding enters the stopped processor state with PC unchanged. Read faults are those defined by the CPU memory and MMU rules.

Notes: None.

50. FSTORE - fstore fs, disp(rs)

Operation: `mem32[rs + disp] = fs`.

Assembler Syntax: `fstore fs, disp(rs)`.

Attributes: Memory: 32-bit write. Operand size: single precision. Privilege: unprivileged. FP operands: `fs` encodes `f0-f15`. Integer operand: `rs` supplies the base address. FPSR/FPCR: neither FPSR nor FPCR is read or written.

Description: The processor evaluates `fs` and writes the result to `mem32[rs + disp]`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x62`.

Instruction Fields: Byte 0 holds opcode `0x62`. Byte 1 bits 7-3 select source FP register `fs`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select integer base register `rs`; byte 2 bits 2-0 and byte 3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold signed `imm32` displacement in little-endian order for the 32-bit memory transfer.

Exceptions: An invalid `fs` encoding enters the stopped processor state with PC unchanged. Write faults are those defined by the CPU memory and MMU rules.

Notes: None.

51. FMOVECR - fmovecr fd, #idx

Operation: `fd = ROM_Constant[idx]`.

Assembler Syntax: `fmovecr fd, #idx`.

Attributes: Memory: none. Operand size: single precision result from an 8-bit constant selector. Privilege: unprivileged. FP operands: `fd` encodes `f0-f15`. FPSR/FPCR: writes FPSR condition-code bits from the constant; FPCR is not read.

Description: The processor evaluates `ROM_Constant[idx]` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x78`.

Instruction Fields: Byte 0 holds opcode `0x78`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 and bytes 2-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order; the low 8 bits select the FPU constant index.

Exceptions: An invalid `fd` encoding enters the stopped processor state with PC unchanged.

Notes: None.

FLOAD/FSTORE always transfer 4 bytes (32 bits) between memory and an FP register. The `disp` is a signed 32-bit immediate. **FSTORE** uses `fs` as the source floating-point register and `rs` as the base integer register for the effective address.

FMOVECR loads a constant from the FPU ROM (indices 0-15). Indices outside this range load 0.0 and set the Z condition code.

Index	Constant	Index	Constant
0	Pi	8	1.0
1	e	9	2.0
2	log2(e)	10	10.0
3	log10(e)	11	100.0
4	ln(2)	12	1000.0
5	ln(10)	13	0.5
6	log10(2)	14	Smallest positive FP32 subnormal (`0x00000001`, approximately `1.40129846e-45`)
7	0.0	15	FLT_MAX

4.6.2 FPU Arithmetic

52. FADD - fadd fd, fs, ft

Operation: `fd = fs + ft`.

Assembler Syntax: `fadd fd, fs, ft`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd`, `fs`, and `ft` encode `f0-f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `fs + ft` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x63`.

Instruction Fields: Byte 0 holds opcode `0x63`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd`, `fs`, or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: None.

53. FSUB - fsub fd, fs, ft

Operation: `fd = fs - ft`.

Assembler Syntax: `fsub fd, fs, ft`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd`, `fs`, and `ft` encode `f0-f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `fs - ft` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x64`.

Instruction Fields: Byte 0 holds opcode `0x64`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd`, `fs`, or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: None.

54. FMUL - fmul fd, fs, ft

Operation: `fd = fs * ft`.

Assembler Syntax: `fmul fd, fs, ft`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd`, `fs`, and `ft` encode `f0-f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `fs * ft` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x65`.

Instruction Fields: Byte 0 holds opcode `0x65`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd`, `fs`, or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: None.

55. FDIV - fdiv fd, fs, ft

Operation: `fd = fs / ft`.

Assembler Syntax: `fdiv fd, fs, ft`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd`, `fs`, and `ft` encode `f0-f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `fs / ft` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x66`.

Instruction Fields: Byte 0 holds opcode `0x66`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd`, `fs`, or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: None.

56. FMOD - fmod fd, fs, ft

Operation: `fd = fs % ft`.

Assembler Syntax: `fmod fd, fs, ft`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd`, `fs`, and `ft` encode `f0-f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `fs % ft` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x67`.

Instruction Fields: Byte 0 holds opcode `0x67`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd`, `fs`, or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: None.

57. FABS - fabs fd, fs

Operation: `fd = \|fs\|`.

Assembler Syntax: `fabs fd, fs`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd` and `fs` encode `f0-f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and does not set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `\|fs\|` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x68`.

Instruction Fields: Byte 0 holds opcode `0x68`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

58. FNEG - fneg fd, fs

Operation: `fd = -fs`.

Assembler Syntax: `fneg fd, fs`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd` and `fs` encode `f0`-`f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and does not set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `-fs` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x69`.

Instruction Fields: Byte 0 holds opcode `0x69`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

59. FSQRT - fsqrt fd, fs

Operation: `fd = sqrt(fs)`.

Assembler Syntax: `fsqrt fd, fs`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd` and `fs` encode `f0`-`f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `sqrt(fs)` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x6A`.

Instruction Fields: Byte 0 holds opcode `0x6A`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

60. FINT - fint fd, fs

Operation: `fd = round(fs)` (uses FPCR mode).

Assembler Syntax: `fint fd, fs`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd` and `fs` encode `f0`-`f15`. FPSR/FPCR: reads FPCR rounding bits, writes FPSR condition-code bits from the rounded result, and does not set FPSR sticky exception flags.

Description: The processor rounds `fs` according to the rounding mode selected by `FPCR` and writes the single-precision result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x6B`.

Instruction Fields: Byte 0 holds opcode `0x6B`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and

bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

4.6.3 FPU Transcendentals

61. FSIN - fsin fd, fs

Operation: `fd = sin(fs)`.

Assembler Syntax: `fsin fd, fs`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd` and `fs` encode `f0`-`f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and does not set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `sin(fs)` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x71`.

Instruction Fields: Byte 0 holds opcode `0x71`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

62. FCOS - fcos fd, fs

Operation: `fd = cos(fs)`.

Assembler Syntax: `fcos fd, fs`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd` and `fs` encode `f0`-`f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and does not set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `cos(fs)` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x72`.

Instruction Fields: Byte 0 holds opcode `0x72`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

63. FTAN - ftan fd, fs

Operation: `fd = tan(fs)`.

Assembler Syntax: `ftan fd, fs`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd` and `fs` encode `f0`-`f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and does not set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `tan(fs)` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x73`.

Instruction Fields: Byte 0 holds opcode `0x73`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

64. FATAN - fatan fd, fs

Operation: `fd = atan(fs)`.

Assembler Syntax: `fatan fd, fs`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd` and `fs` encode `f0`-`f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and does not set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `atan(fs)` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x74`.

Instruction Fields: Byte 0 holds opcode `0x74`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

65. FLOG - flog fd, fs

Operation: `fd = ln(fs)`.

Assembler Syntax: `flog fd, fs`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd` and `fs` encode `f0`-`f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `ln(fs)` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x75`.

Instruction Fields: Byte 0 holds opcode `0x75`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

66. FEXP - fexp fd, fs

Operation: `fd = e^fs`.

Assembler Syntax: `fexp fd, fs`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd` and `fs` encode `f0-f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `e^fs` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x76`.

Instruction Fields: Byte 0 holds opcode `0x76`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

67. FPOW - fpow fd, fs, ft

Operation: `fd = fs^ft`.

Assembler Syntax: `fpow fd, fs, ft`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fd`, `fs`, and `ft` encode `f0-f15`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `fs^ft` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x77`.

Instruction Fields: Byte 0 holds opcode `0x77`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd`, `fs`, or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: None.

4.6.4 FPU Comparison and Conversion

68. FCMP - fcmp rd, fs, ft

Operation: `rd = (fs < ft ? -1 : (fs > ft ? 1 : 0))`.

Assembler Syntax: `fcmp rd, fs, ft`.

Attributes: Memory: none. Operand size: single precision. Privilege: unprivileged. FP operands: `fs` and `ft` encode `f0-f15`; `rd` receives the integer comparison result. FPSR/FPCR: writes FPSR condition-code bits and may set the IO sticky flag for unordered operands; FPCR is not read.

Description: The processor compares the single-precision values in `fs` and `ft`, writes `-1`, `0`, or `1` to `rd`, and updates FPSR condition state. If either operand is NaN, the comparison is unordered: `rd` receives `0`, the NaN condition code is set, and the IO sticky exception flag is set.

Condition Codes: See the FPU condition-state description for this instruction class.

Instruction Format: Fixed 8-byte instruction; opcode = `0x6C`.

Instruction Fields: Byte 0 holds opcode `0x6C`. Byte 1 bits 7-3 select destination integer register `rd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fs` or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: `+Inf` compares equal to `+Inf` and greater than finite values.

69. FCVTIF - fcvtif fd, rs

Operation: `fd = float32(int32(rs))`.

Assembler Syntax: `fcvtif fd, rs`.

Attributes: Memory: none. Operand size: single precision result from a 32-bit signed integer source. Privilege: unprivileged. FP operands: `fd` encodes `f0-f15`; `rs` is an integer source register. FPSR/FPCR: writes FPSR condition-code bits from the FP result; FPCR is not read.

Description: The processor evaluates `float32(int32(rs))` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x6D`.

Instruction Fields: Byte 0 holds opcode `0x6D`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source integer register `rs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` encoding enters the stopped processor state with PC unchanged.

Notes: None.

70. FCVTFI - fcvtfi rd, fs

Operation: `rd = int32(fs)` (saturating).

Assembler Syntax: `fcvtfi rd, fs`.

Attributes: Memory: none. Operand size: single precision source to 32-bit signed integer result. Privilege: unprivileged. FP operands: `fs` encodes `f0-f15`; `rd` receives the integer result. FPSR/FPCR: may set the FPSR IO sticky flag for NaN or saturation; FPCR is not read.

Description: The processor converts `fs` to a signed 32-bit integer with saturation and writes the result to `rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x6E`.

Instruction Fields: Byte 0 holds opcode `0x6E`. Byte 1 bits 7-3 select destination integer register `rd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

71. FMOVI - fmovi fd, rs

Operation: `fd = bits_to_float(uint32(rs))`.

Assembler Syntax: `fmovi fd, rs`.

Attributes: Memory: none. Operand size: 32-bit bit pattern moved from integer to single precision FP storage. Privilege: unprivileged. FP operands: `fd` encodes `f0-f15`; `rs` is an integer source register. FPSR/FPCR: writes FPSR condition-code bits from the resulting FP bit pattern; FPCR is not read.

Description: The processor evaluates `bits_to_float(uint32(rs))` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x6F`.

Instruction Fields: Byte 0 holds opcode `0x6F`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source integer register `rs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` encoding enters the stopped processor state with PC unchanged.

Notes: None.

72. FMOVO - fmovo rd, fs

Operation: `rd = uint64(float_to_bits(fs))`.

Assembler Syntax: `fmovo rd, fs`.

Attributes: Memory: none. Operand size: 32-bit FP bit pattern zero-extended to an integer result. Privilege: unprivileged. FP operands: `fs` encodes `f0-f15`; `rd` receives the integer result. FPSR/FPCR: neither FPSR nor FPCR is read or written.

Description: The processor evaluates `uint64(float_to_bits(fs))` and writes the result to `rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x70`.

Instruction Fields: Byte 0 holds opcode `0x70`. Byte 1 bits 7-3 select destination integer register `rd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

FCVTFI saturates to `INT32_MAX` or `INT32_MIN` on overflow. NaNs return 0 and set the IO exception flag.

4.6.5 FPU Status and Control

73. FMOVSR - fmovsr rd

Operation: `rd = FPSR`.

Assembler Syntax: `fmovsr rd`.

Attributes: Memory: none. Operand size: 32-bit status word. Privilege: unprivileged. Integer operand: `rd` receives FPSR, with writes to `R0` discarded. FPSR/FPCR: reads FPSR; FPCR is not read or written.

Description: The processor evaluates `FPSR` and writes the result to `rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x79`.

Instruction Fields: Byte 0 holds opcode `0x79`. Byte 1 bits 7-3 select destination integer register `rd`; byte 1 bits 2-0 and bytes 2-7 are reserved by this instruction and ignored by the processor. The value read is `FPSR`.

Exceptions: None.

Notes: None.

74. FMOVCR - fmovcr rd

Operation: `rd = FPCR`.

Assembler Syntax: `fmovcr rd`.

Attributes: Memory: none. Operand size: 32-bit control word. Privilege: unprivileged. Integer operand: `rd` receives FPCR, with writes to `R0` discarded. FPSR/FPCR: reads FPCR; FPSR is not read or written.

Description: The processor evaluates `FPCR` and writes the result to `rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x7A`.

Instruction Fields: Byte 0 holds opcode `0x7A`. Byte 1 bits 7-3 select destination integer register `rd`; byte 1 bits 2-0 and bytes 2-7 are reserved by this instruction and ignored by the processor. The value read is `FPCR`.

Exceptions: None.

Notes: None.

75. FMOVSC - fmovsc rs

Operation: `FPSR = rs` (masked).

Assembler Syntax: `fmovsc rs`.

Attributes: Memory: none. Operand size: 32-bit status word. Privilege: unprivileged. Integer operand: `rs` supplies the new FPSR value. FPSR/FPCR: writes FPSR after applying the FPSR writable-bit mask; FPCR is not read or written.

Description: The processor writes the architecturally writable status bits from `rs` into `FPSR`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x7B`.

Instruction Fields: Byte 0 holds opcode `0x7B`. Byte 2 bits 7-3 select source integer register `rs`; bytes 1, byte 2 bits 2-0, and bytes 3-7 are reserved by this instruction and ignored by the processor. The destination is `FPSR`.

Exceptions: None.

Notes: None.

76. FMOVCC - fmovcc rs

Operation: `FPCR = rs`.

Assembler Syntax: `fmovcc rs`.

Attributes: Memory: none. Operand size: 32-bit control word. Privilege: unprivileged. Integer operand: `rs` supplies the new FPCR value. FPSR/FPCR: writes FPCR; FPSR is not read or written.

Description: The processor writes `rs` to `FPCR` subject to that control register's architectural write rules.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x7C`.

Instruction Fields: Byte 0 holds opcode `0x7C`. Byte 2 bits 7-3 select source integer register `rs`; bytes 1, byte 2 bits 2-0, and bytes 3-7 are reserved by this instruction and ignored by the processor. The destination is `FPCR`.

Exceptions: None.

Notes: None.

FPSR (Status Register): - Bits 27:24 - Condition Codes (N, Z, I, NaN). Overwritten per instruction. - Bits 3:0 - Exception Flags (UE, OE, DZ, IO). Sticky (IEEE-754). - **FMOVSC** masks the input value to preserve only these valid bits; bits 23:4 are reserved and always read as zero.

FPCR (Control Register): - Bits 1:0 - Rounding Mode: - 00: Nearest (default) - 01: Toward Zero (truncate) - 10: Toward -Inf (floor) - 11: Toward +Inf (ceil) - **FMOVCC** stores the full 32-bit source value in FPCR. FPU arithmetic interprets only bits 1:0 as the rounding mode; bits 31:2 are preserved and have no defined effect.

4.6.6 Double-Precision (Register Pairs)

77. DMOV - dmov fd, fs

Operation: `fd = fs`.

Assembler Syntax: `dmov fd, fs`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd` and `fs` must encode even registers from `f0` through `f14`. FPSR/FPCR: neither FPSR nor FPCR is read or written.

Description: The processor evaluates `fs` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x80`.

Instruction Fields: Byte 0 holds opcode `0x80`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

78. DLOAD - dload fd, disp(rs)

Operation: `fd = mem64[rs + disp]`.

Assembler Syntax: `dload fd, disp(rs)`.

Attributes: Memory: 64-bit read. Operand size: double precision. Privilege: unprivileged. FP operands: `fd` must encode an even register from `f0` through `f14`. Integer operand: `rs` supplies the base address. FPSR/FPCR: writes FPSR condition-code bits from the loaded value; FPCR is not read.

Description: The processor evaluates `mem64[rs + disp]` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x81`.

Instruction Fields: Byte 0 holds opcode `0x81`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select integer base register `rs`; byte 2 bits 2-0 and byte 3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold signed `imm32` displacement in little-endian order for the 64-bit memory transfer.

Exceptions: An invalid or odd `fd` encoding enters the stopped processor state with PC unchanged. Read faults are those defined by the CPU memory and MMU rules.

Notes: None.

79. DSTORE - dstore fs, disp(rs)

Operation: `mem64[rs + disp] = fs`.

Assembler Syntax: `dstore fs, disp(rs)`.

Attributes: Memory: 64-bit write. Operand size: double precision. Privilege: unprivileged. FP operands: `fs` must encode an even register from `f0` through `f14`. Integer operand: `rs` supplies the base address. FPSR/FPCR: neither FPSR nor FPCR is read or written.

Description: The processor evaluates `fs` and writes the result to `mem64[rs + disp]`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x82`.

Instruction Fields: Byte 0 holds opcode `0x82`. Byte 1 bits 7-3 select source FP register `fs`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select integer base register `rs`; byte 2 bits 2-0 and byte 3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold signed `imm32` displacement in little-endian order for the 64-bit memory transfer.

Exceptions: An invalid or odd `fs` encoding enters the stopped processor state with PC unchanged. Write faults are those defined by the CPU memory and MMU rules.

Notes: None.

80. DADD - dadd fd, fs, ft

Operation: `fd = fs + ft`.

Assembler Syntax: `dadd fd, fs, ft`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd`, `fs`, and `ft` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set

FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `fs + ft` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x83`.

Instruction Fields: Byte 0 holds opcode `0x83`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd`, `fs`, or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: None.

81. DSUB - dsub fd, fs, ft

Operation: `fd = fs - ft`.

Assembler Syntax: `dsub fd, fs, ft`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd`, `fs`, and `ft` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `fs - ft` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x84`.

Instruction Fields: Byte 0 holds opcode `0x84`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd`, `fs`, or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: None.

82. DMUL - dmul fd, fs, ft

Operation: `fd = fs * ft`.

Assembler Syntax: `dmul fd, fs, ft`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd`, `fs`, and `ft` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `fs * ft` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x85`.

Instruction Fields: Byte 0 holds opcode `0x85`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd`, `fs`, or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: None.

83. DDIV - ddiv fd, fs, ft

Operation: `fd = fs / ft`.

Assembler Syntax: `ddiv fd, fs, ft`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd`, `fs`, and `ft` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `fs / ft` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x86`.

Instruction Fields: Byte 0 holds opcode `0x86`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd`, `fs`, or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: None.

84. DMOD - dmod fd, fs, ft

Operation: `fd = fmod(fs, ft)`.

Assembler Syntax: `dmod fd, fs, ft`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd`, `fs`, and `ft` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `fmod(fs, ft)` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x87`.

Instruction Fields: Byte 0 holds opcode `0x87`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd`, `fs`, or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: None.

85. DABS - dabs fd, fs

Operation: `fd = \|fs\|`.

Assembler Syntax: `dabs fd, fs`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd` and `fs` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and does not set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `\|fs\|` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x88`.

Instruction Fields: Byte 0 holds opcode `0x88`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

86. DNEG - dneg fd, fs

Operation: `fd = -fs`.

Assembler Syntax: `dneg fd, fs`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd` and `fs` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and does not set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `-fs` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x89`.

Instruction Fields: Byte 0 holds opcode `0x89`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

87. DSQRT - dsqrt fd, fs

Operation: `fd = sqrt(fs)`.

Assembler Syntax: `dsqrt fd, fs`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd` and `fs` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `sqrt(fs)` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x8A`.

Instruction Fields: Byte 0 holds opcode `0x8A`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

88. DINT - dint fd, fs

Operation: `fd = round(fs)`.

Assembler Syntax: `dint fd, fs`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd` and `fs` must encode even registers from `f0` through `f14`. FPSR/FPCR: reads FPCR rounding bits, writes FPSR condition-code bits from the rounded result, and does not set FPSR sticky exception flags.

Description: The processor evaluates `round(fs)` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x8B`.

Instruction Fields: Byte 0 holds opcode `0x8B`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

89. DCMP - dcmp rd, fs, ft

Operation: `rd = -1/0/1`.

Assembler Syntax: `dcmp rd, fs, ft`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fs` and `ft` must encode even registers from `f0` through `f14`; `rd` receives the integer comparison result. FPSR/FPCR: writes FPSR condition-code bits and may set the IO sticky flag for unordered operands; FPCR is not read.

Description: The processor compares the double-precision values in `fs:fs+1` and `ft:ft+1`, writes `-1`, `0`, or `1` to `rd`, and updates FPSR condition state. If either operand is NaN, the comparison is unordered: `rd` receives `0`, the NaN condition code is set, and the IO sticky exception flag is set.

Condition Codes: See the FPU condition-state description for this instruction class.

Instruction Format: Fixed 8-byte instruction; opcode = `0x8C`.

Instruction Fields: Byte 0 holds opcode `0x8C`. Byte 1 bits 7-3 select destination integer register `rd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fs` or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: `+Inf` compares equal to `+Inf` and greater than finite values.

90. DCVTIF - dcvtif fd, rs

Operation: `fd = float64(int64(rs))`.

Assembler Syntax: `dcvtif fd, rs`.

Attributes: Memory: none. Operand size: double precision result from a 64-bit signed integer source. Privilege: unprivileged. FP operands: `fd` must encode an even register from `f0` through `f14`; `rs` is an integer source register. FPSR/FPCR: writes FPSR condition-code bits from the FP result; FPCR is not read.

Description: The processor evaluates `float64(int64(rs))` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x8D`.

Instruction Fields: Byte 0 holds opcode `0x8D`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source integer register `rs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd` encoding enters the stopped processor state with PC unchanged.

Notes: None.

91. DCVTFI - dcvtfi rd, fs

Operation: `rd = int64(fs)` (saturating).

Assembler Syntax: `dcvtfi rd, fs`.

Attributes: Memory: none. Operand size: double precision source to 64-bit signed integer result. Privilege: unprivileged. FP operands: `fs` must encode an even register from `f0` through `f14`; `rd` receives the integer result. FPSR/FPCR: may set FPSR IO for NaN or saturation; FPCR is not read.

Description: The processor converts `fs` to a signed 64-bit integer with saturation and writes the result to `rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x8E`.

Instruction Fields: Byte 0 holds opcode `0x8E`. Byte 1 bits 7-3 select destination integer register `rd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fs` encoding enters the stopped processor state with PC unchanged.

Notes: None.

92. FCVTSD - fcvtsd fd, fs

Operation: `fd = float64(float32(fs))`.

Assembler Syntax: `fcvtsd fd, fs`.

Attributes: Memory: none. Operand sizes: single precision source, double precision result. Privilege: unprivileged. FP operands: `fs` encodes `f0-f15`; `fd` must encode an even register from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result; FPCR is not read.

Description: The processor evaluates `float64(float32(fs))` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x8F`.

Instruction Fields: Byte 0 holds opcode `0x8F`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fs` encoding, or an invalid or odd `fd` encoding, enters the stopped processor state with PC unchanged.

Notes: None.

93. FCVTDS - fcvtds fd, fs

Operation: `fd = float32(float64(fs))`.

Assembler Syntax: `fcvtds fd, fs`.

Attributes: Memory: none. Operand sizes: double precision source, single precision result. Privilege: unprivileged. FP operands: `fs` must encode an even register from `f0` through `f14`; `fd` encodes `f0-f15`. FPSR/FPCR: writes FPSR condition-code bits from the result; FPCR is not read.

Description: The processor evaluates `float32(float64(fs))` and writes the result to `fd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x90`.

Instruction Fields: Byte 0 holds opcode `0x90`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid `fd` encoding, or an invalid or odd `fs` encoding, enters the stopped processor state with PC unchanged.

Notes: None.

94. DSIN - dsin fd, fs

Operation: `fd = sin(fs)`.

Assembler Syntax: `dsin fd, fs`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd` and `fs` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and does not set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `sin(fs)` and writes the double-precision result to `fd`.

Condition Codes: See the FPU condition-state description for this instruction class.

Instruction Format: Fixed 8-byte instruction; opcode = `0x91`.

Instruction Fields: Byte 0 holds opcode `0x91`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: NaN inputs propagate through the canonical FPU helper and set the NaN condition code without creating a new sticky invalid-operation flag.

95. DCOS - dcos fd, fs

Operation: `fd = cos(fs)`.

Assembler Syntax: `dcos fd, fs`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd` and `fs` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and does not set FPSR

sticky exception flags; FPCR is not read.

Description: The processor evaluates `cos(fs)` and writes the double-precision result to `fd`.

Condition Codes: See the FPU condition-state description for this instruction class.

Instruction Format: Fixed 8-byte instruction; opcode = `0x92`.

Instruction Fields: Byte 0 holds opcode `0x92`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: NaN inputs propagate through the canonical FPU helper and set the NaN condition code without creating a new sticky invalid-operation flag.

96. DTAN - dtan fd, fs

Operation: `fd = tan(fs)`.

Assembler Syntax: `dtan fd, fs`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd` and `fs` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and does not set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `tan(fs)` and writes the double-precision result to `fd`.

Condition Codes: See the FPU condition-state description for this instruction class.

Instruction Format: Fixed 8-byte instruction; opcode = `0x93`.

Instruction Fields: Byte 0 holds opcode `0x93`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: NaN inputs propagate through the canonical FPU helper and set the NaN condition code without creating a new sticky invalid-operation flag.

97. DATAN - datan fd, fs

Operation: `fd = atan(fs)`.

Assembler Syntax: `datan fd, fs`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd` and `fs` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and does not set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `atan(fs)` and writes the double-precision result to `fd`.

Condition Codes: See the FPU condition-state description for this instruction class.

Instruction Format: Fixed 8-byte instruction; opcode = `0x94`.

Instruction Fields: Byte 0 holds opcode `0x94`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: NaN inputs propagate through the canonical FPU helper and set the NaN condition code without creating a new sticky invalid-operation flag.

98. DLOG - dlog fd, fs

Operation: `fd = ln(fs)`.

Assembler Syntax: `dlog fd, fs`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd` and `fs` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `ln(fs)` and writes the double-precision result to `fd`. A zero input sets divide-by-zero. A negative non-zero non-NaN input, including negative infinity, sets invalid operation.

Condition Codes: See the FPU condition-state description for this instruction class.

Instruction Format: Fixed 8-byte instruction; opcode = `0x95`.

Instruction Fields: Byte 0 holds opcode `0x95`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: NaN inputs propagate through the canonical FPU helper and set the NaN condition code without creating a new sticky invalid-operation flag.

99. DEXP - dexp fd, fs

Operation: `fd = e^fs`.

Assembler Syntax: `dexp fd, fs`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd` and `fs` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `e^fs` and writes the double-precision result to `fd`. A finite input that produces infinity sets overflow. A finite non-zero input that produces zero sets underflow.

Condition Codes: See the FPU condition-state description for this instruction class.

Instruction Format: Fixed 8-byte instruction; opcode = `0x96`.

Instruction Fields: Byte 0 holds opcode `0x96`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 2 bits 2-0, byte 3, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd` or `fs` encoding enters the stopped processor state with PC unchanged.

Notes: NaN inputs propagate through the canonical FPU helper and set the NaN condition code without creating a new sticky invalid-operation flag.

100. DPOW - dpow fd, fs, ft

Operation: `fd = fs^ft`.

Assembler Syntax: `dpow fd, fs, ft`.

Attributes: Memory: none. Operand size: double precision. Privilege: unprivileged. FP operands: `fd`, `fs`, and `ft` must encode even registers from `f0` through `f14`. FPSR/FPCR: writes FPSR condition-code bits from the result and may set FPSR sticky exception flags; FPCR is not read.

Description: The processor evaluates `fs^ft` and writes the double-precision result to `fd`. Finite inputs that produce infinity set overflow; finite non-zero inputs that produce zero set underflow; non-NaN inputs that produce NaN set invalid operation.

Condition Codes: See the FPU condition-state description for this instruction class.

Instruction Format: Fixed 8-byte instruction; opcode = `0x97`.

Instruction Fields: Byte 0 holds opcode `0x97`. Byte 1 bits 7-3 select destination FP register `fd`; byte 1 bits 2-0 are reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select source FP register `fs`; byte 3 bits 7-3 select source FP register `ft`. Byte 2 bits 2-0, byte 3 bits 2-0, and bytes 4-7 are reserved by this instruction and ignored by the processor.

Exceptions: An invalid or odd `fd`, `fs`, or `ft` encoding enters the stopped processor state with PC unchanged.

Notes: NaN inputs propagate through the canonical FPU helper and set the NaN condition code without creating a new sticky invalid-operation flag unless the operation creates NaN from non-NaN inputs.

Notes: - `dload`/`dstore` always transfer 8 bytes. - `dcvtfi` saturates to `INT64_MAX`/`INT64_MIN` on overflow and sets IO. - `fcvtsd` requires an even destination. `fcvtds` requires an even source. - Double-precision opcodes are unsized; size suffixes are not used. - `dsin`, `dcos`, `dtan`, `datan`, `dlog`, `dexp`, and `dpow` are unsized double-precision FP64 opcodes.

4.7 Branches

All branches are PC-relative. The branch offset is stored as a signed 32-bit value in the imm32 field. The new PC is calculated as:

```
PC_new = PC_current + signExtend32to64(offset)
```

If the branch is not taken, PC advances by 8 (one instruction).

101. BRA - bra label

Operation: `PC = PC + signExtend32to64(imm32)`.

Assembler Syntax: `bra label`.

Attributes: Condition: Always; Comparison: --.

Description: Unconditionally branches to the signed 32-bit PC-relative target encoded in imm32.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x40`.

Instruction Fields: Byte 0 holds opcode `0x40`. Bytes 1-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold signed `imm32` PC-relative branch offset in little-endian order.

Exceptions: None.

Notes: None.

102. BEQ - beq Rs, Rt, label

Operation: `if Rs == Rt then PC = PC + signExtend32to64(imm32) else PC += 8`.

Assembler Syntax: `beq Rs, Rt, label`.

Attributes: Condition: `Rs == Rt`; Comparison: Unsigned (equality).

Description: Compares Rs and Rt for equality and branches to the signed 32-bit PC-relative target when the condition is true.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x41`.

Instruction Fields: Byte 0 holds opcode `0x41`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select comparison register `Rs`; byte 3 bits 7-3 select comparison register `Rt`. Byte 2 bits 2-0 and byte 3 bits 2-0 are reserved. Bytes 4-7 hold signed `imm32` PC-relative branch offset in little-endian order.

Exceptions: None.

Notes: None.

103. BNE - bne Rs, Rt, label

Operation: `if Rs != Rt then PC = PC + signExtend32to64(imm32) else PC += 8`.

Assembler Syntax: `bne Rs, Rt, label`.

Attributes: Condition: `Rs != Rt`; Comparison: Unsigned (equality).

Description: Compares Rs and Rt for equality and branches to the signed 32-bit PC-relative target when the condition is true.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x42`.

Instruction Fields: Byte 0 holds opcode `0x42`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select comparison register `Rs`; byte 3 bits 7-3 select comparison register `Rt`. Byte 2 bits 2-0 and byte 3 bits 2-0 are reserved. Bytes 4-7 hold signed `imm32` PC-relative branch offset in little-endian order.

Exceptions: None.

Notes: None.

104. BLT - blt Rs, Rt, label

Operation: `if int64(Rs) < int64(Rt) then PC = PC + signExtend32to64(imm32) else PC += 8`.

Assembler Syntax: `blt Rs, Rt, label`.

Attributes: Condition: `int64(Rs) < int64(Rt)`; Comparison: Signed.

Description: Interprets Rs and Rt as signed 64-bit integers and branches to the signed 32-bit PC-relative target when the condition is true.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x43`.

Instruction Fields: Byte 0 holds opcode `0x43`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select comparison register `Rs`; byte 3 bits 7-3 select comparison register `Rt`. Byte 2 bits 2-0 and byte 3 bits 2-0 are

reserved. Bytes 4-7 hold signed `imm32` PC-relative branch offset in little-endian order.

Exceptions: None.

Notes: None.

105. BGE - bge Rs, Rt, label

Operation: `if int64(Rs) >= int64(Rt) then PC = PC + signExtend32to64(imm32) else PC += 8`.

Assembler Syntax: `bge Rs, Rt, label`.

Attributes: Condition: `int64(Rs) >= int64(Rt)`; Comparison: Signed.

Description: Interprets Rs and Rt as signed 64-bit integers and branches to the signed 32-bit PC-relative target when the condition is true.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x44`.

Instruction Fields: Byte 0 holds opcode `0x44`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select comparison register `Rs`; byte 3 bits 7-3 select comparison register `Rt`. Byte 2 bits 2-0 and byte 3 bits 2-0 are reserved. Bytes 4-7 hold signed `imm32` PC-relative branch offset in little-endian order.

Exceptions: None.

Notes: None.

106. BGT - bgt Rs, Rt, label

Operation: `if int64(Rs) > int64(Rt) then PC = PC + signExtend32to64(imm32) else PC += 8`.

Assembler Syntax: `bgt Rs, Rt, label`.

Attributes: Condition: `int64(Rs) > int64(Rt)`; Comparison: Signed.

Description: Interprets Rs and Rt as signed 64-bit integers and branches to the signed 32-bit PC-relative target when the condition is true.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x45`.

Instruction Fields: Byte 0 holds opcode `0x45`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select comparison register `Rs`; byte 3 bits 7-3 select comparison register `Rt`. Byte 2 bits 2-0 and byte 3 bits 2-0 are reserved. Bytes 4-7 hold signed `imm32` PC-relative branch offset in little-endian order.

Exceptions: None.

Notes: None.

107. BLE - ble Rs, Rt, label

Operation: `if int64(Rs) <= int64(Rt) then PC = PC + signExtend32to64(imm32) else PC += 8`.

Assembler Syntax: `ble Rs, Rt, label`.

Attributes: Condition: `int64(Rs) <= int64(Rt)`; Comparison: Signed.

Description: Interprets Rs and Rt as signed 64-bit integers and branches to the signed 32-bit PC-relative target when the condition is true.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x46`.

Instruction Fields: Byte 0 holds opcode `0x46`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select comparison register `Rs`; byte 3 bits 7-3 select comparison register `Rt`. Byte 2 bits 2-0 and byte 3 bits 2-0 are reserved. Bytes 4-7 hold signed `imm32` PC-relative branch offset in little-endian order.

Exceptions: None.

Notes: None.

108. BHI - bhi Rs, Rt, label

Operation: `if Rs > Rt then PC = PC + signExtend32to64(imm32) else PC += 8`.

Assembler Syntax: `bhi Rs, Rt, label`.

Attributes: Condition: `Rs > Rt`; Comparison: Unsigned.

Description: Interprets Rs and Rt as unsigned 64-bit integers and branches to the signed 32-bit PC-relative target when the condition is true.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x47`.

Instruction Fields: Byte 0 holds opcode `0x47`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select comparison register `Rs`; byte 3 bits 7-3 select comparison register `Rt`. Byte 2 bits 2-0 and byte 3 bits 2-0 are reserved. Bytes 4-7 hold signed `imm32` PC-relative branch offset in little-endian order.

Exceptions: None.

Notes: None.

109. BLS - bls Rs, Rt, label

Operation: `if Rs <= Rt then PC = PC + signExtend32to64(imm32) else PC += 8`.

Assembler Syntax: `bls Rs, Rt, label`.

Attributes: Condition: `Rs <= Rt`; Comparison: Unsigned.

Description: Interprets Rs and Rt as unsigned 64-bit integers and branches to the signed 32-bit PC-relative target when the condition is true.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x48`.

Instruction Fields: Byte 0 holds opcode `0x48`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select comparison register `Rs`; byte 3 bits 7-3 select comparison register `Rt`. Byte 2 bits 2-0 and byte 3 bits 2-0 are reserved. Bytes 4-7 hold signed `imm32` PC-relative branch offset in little-endian order.

Exceptions: None.

Notes: None.

110. JMP - `jmp (Rs)` / `jmp disp(Rs)`

Operation: `PC = Rs + signExtend32to64(imm32)`.

Assembler Syntax: `jmp (Rs)` / `jmp disp(Rs)`.

Attributes: Condition: `PC = Rs + signExtend(disp)`; Comparison: Register-indirect.

Description: Transfers control to the effective address computed from Rs plus the signed 32-bit displacement.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x49`.

Instruction Fields: Byte 0 holds opcode `0x49`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select base register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold signed `imm32` displacement in little-endian order.

Exceptions: None.

Notes: None.

Encoding note for conditional branches: Rs is in byte 2[7:3], Rt is in byte 3[7:3], and the branch offset is in bytes 4-7 (imm32). The Rd field (byte 1[7:3]) is unused and should be encoded as 0. Branch targets are limited to the signed 32-bit PC-relative range described in section 7.3.

BRA uses only the imm32 field. Rs and Rt fields are unused.

JMP (opcode `0x49`): - Computes the effective address as `uint64(int64(Rs) + int64(int32(imm32)))`. - Transfers control to the effective address. - Does not modify the stack. No return address is saved. - Rs is in byte 2[7:3], the optional displacement is in bytes 4-7 (imm32). - Enables computed jumps, jump tables, and register-indirect branching.

4.8 Subroutine / Stack

111. JSR - jsr label

Operation: `SP -= 8; mem[SP] = PC + 8; PC = PC + offset`.

Assembler Syntax: `jsr label`.

Attributes: Memory: Write; Size: Q.

Description: The processor pushes `PC + 8` on the stack and loads `PC + offset` into `PC`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x50`.

Instruction Fields: Byte 0 holds opcode `0x50`. Bytes 1-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold signed `imm32` PC-relative subroutine offset in little-endian order.

Exceptions: If MMU translation is enabled, the stack write can trap with cause 0 (`FAULT_NOT_PRESENT`), cause 2 (`FAULT_WRITE_DENIED`), or cause 10 (`FAULT_SKAC`). A physical 8-byte stack write outside implemented CPU-visible memory enters the stopped processor state and does not create a trap frame.

Notes: None.

112. RTS - rts

Operation: `PC = mem[SP]; SP += 8`.

Assembler Syntax: `rts`.

Attributes: Memory: Read; Size: Q.

Description: The processor pops an address from the stack into `PC` and increments `SP` by eight bytes.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x51`.

Instruction Fields: Byte 0 holds opcode `0x51`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: If MMU translation is enabled, the stack read can trap with cause 0 (`FAULT_NOT_PRESENT`), cause 1 (`FAULT_READ_DENIED`), or cause 10 (`FAULT_SKAC`). A physical 8-byte stack read outside implemented CPU-visible memory enters the stopped processor state and does not create a trap frame.

Notes: None.

113. PUSH - push Rs

Operation: `SP -= 8; mem[SP] = Rs`.

Assembler Syntax: `push Rs`.

Attributes: Memory: Write; Size: Q.

Description: The processor decrements `SP` by eight bytes and stores `Rs` at the new stack top.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x52`.

Instruction Fields: Byte 0 holds opcode `0x52`. Byte 2 bits 7-3 select source register `Rs`; bytes 1, byte 2 bits 2-0, and bytes 3-7 are reserved by this instruction and ignored by the processor.

Exceptions: If MMU translation is enabled, the stack write can trap with cause 0 (`FAULT_NOT_PRESENT`), cause 2 (`FAULT_WRITE_DENIED`), or cause 10 (`FAULT_SKAC`). A physical 8-byte stack write outside implemented CPU-visible memory enters the stopped processor state and does not create a trap frame.

Notes: None.

114. POP - pop Rd

Operation: `Rd = mem[SP]; SP += 8`.

Assembler Syntax: `pop Rd`.

Attributes: Memory: Read; Size: Q.

Description: The processor loads a 64-bit value from the stack top into `Rd` and increments `SP` by eight bytes.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x53`.

Instruction Fields: Byte 0 holds opcode `0x53`. Byte 1 bits 7-3 select destination register `Rd`; byte 1 bits 2-0 and bytes 2-7 are reserved by this instruction and ignored by the processor.

Exceptions: If MMU translation is enabled, the stack read can trap with cause 0 (`FAULT_NOT_PRESENT`), cause 1 (`FAULT_READ_DENIED`), or cause 10 (`FAULT_SKAC`). A physical 8-byte stack read outside implemented CPU-visible memory enters the stopped processor state and does not create a trap frame.

Notes: None.

115. JSR - `jsr (Rs)` / `jsr disp(Rs)`

Operation: `SP -= 8; mem[SP] = PC + 8; PC = Rs + signExtend(disp)`.

Assembler Syntax: `jsr (Rs)` / `jsr disp(Rs)`.

Attributes: Memory: Write; Size: Q.

Description: The processor pushes `PC + 8` on the stack and loads `Rs + signExtend(disp)` into `PC`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x54`.

Instruction Fields: Byte 0 holds opcode `0x54`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 bits 7-3 select the base register `Rs`; byte 2 bits 2-0 and byte 3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold the signed displacement in little-endian order. Memory access class: Write.

Exceptions: If MMU translation is enabled, the stack write can trap with cause 0 (`FAULT_NOT_PRESENT`), cause 2 (`FAULT_WRITE_DENIED`), or cause 10 (`FAULT_SKAC`). A physical 8-byte stack write outside implemented CPU-visible memory enters the stopped processor state and does not create a trap frame.

Notes: None.

4.9 System

116. NOP - nop

Operation: No operation; PC += 8.

Assembler Syntax: `nop`.

Attributes: Memory: N; Size: --.

Description: No architectural register or memory state is modified; execution continues at the next instruction.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xE0`.

Instruction Fields: Byte 0 holds opcode `0xE0`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

117. HALT - halt

Operation: Stops execution.

Assembler Syntax: `halt`.

Attributes: Memory: N; Size: --.

Description: `HALT` enters the stopped processor state. The program counter is not advanced after the `HALT` word is fetched.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xE1`.

Instruction Fields: Byte 0 holds opcode `0xE1`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: None. No trap is generated.

Notes: `HALT` does not advance `PC`.

118. SEI - sei

Operation: Enable interrupts (set TIMER_CTRL bit 1).

Assembler Syntax: `sei`.

Attributes: Memory: N; Size: --.

Description: The processor enables interrupt delivery by setting the interrupt-enable bit in the timer control state.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xE2`.

Instruction Fields: Byte 0 holds opcode `0xE2`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

119. CLI - cli

Operation: Disable interrupts (clear TIMER_CTRL bit 1).

Assembler Syntax: `cli`.

Attributes: Memory: N; Size: --.

Description: The processor disables interrupt delivery by clearing the interrupt-enable bit in the timer control state.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xE3`.

Instruction Fields: Byte 0 holds opcode `0xE3`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

120. RTI - rti

Operation: Return from interrupt.

Assembler Syntax: `rti`.

Attributes: Memory: Read; Size: Q.

Description: The processor restores interrupt return state and resumes execution at the interrupted address.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xE4`.

Instruction Fields: Byte 0 holds opcode `0xE4`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: If MMU translation is enabled, the stack read can trap with cause 0 (`FAULT_NOT_PRESENT`), cause 1 (`FAULT_READ_DENIED`), or cause 10 (`FAULT_SKAC`). A physical 8-byte stack read outside implemented CPU-visible memory enters the stopped processor state and does not create a trap frame.

Notes: None.

121. WAIT - wait #usec

Operation: Sleep for `imm32` microseconds; PC += 8.

Assembler Syntax: `wait #usec`.

Attributes: Memory: N; Size: --.

Description: The processor waits for the number of microseconds encoded in `imm32` and then resumes at the next instruction.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xE5`.

Instruction Fields: Byte 0 holds opcode `0xE5`. Bytes 1-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold unsigned `imm32` in little-endian order; this field is the requested wait interval in microseconds.

Exceptions: None.

Notes: None.

4.10 MMU, Privilege, and Atomic Instructions

122. MTCR - mtcr CRn, Rs

Operation: `CRn = Rs`.

Assembler Syntax: `mtcr CRn, Rs`.

Attributes: Privileged; Memory: N; Size: Q.

Description: The processor writes the 64-bit value in `Rs` to the selected control register. `CRn` is a 5-bit encoded field; assigned control registers are listed in section 8.1.1. Writing `CR0` (PTBR) or changing `CR5` bit 0 (MMU enable) invalidates all TLB entries. Writes to `CR5` bit 1 (supervisor mode) and bit 4 (SUA latch) are ignored; those state bits are changed only by trap entry, `ERET`, `SUAEN`, and `SUADIS`. Supervisor-mode writes to unassigned control-register numbers have no architectural effect.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xE6`.

Instruction Fields: Byte 0 holds opcode `0xE6`. Byte 1 bits 7-3 hold the control-register number `CRn`; byte 1 bits 2-0 are reserved. Byte 2 bits 7-3 select source register `Rs`; byte 2 bits 2-0 are reserved. Bytes 3-7 are reserved by this instruction and ignored by the processor.

Exceptions: User-mode execution raises `FAULT_PRIV` (cause 5). Writing `CR15` (`RAM_SIZE_BYTES`) raises `FAULT_ILLEGAL_INSTRUCTION` (cause 11).

Notes: `CR13` (`PREV_MODE`) is read-only by effect and ignores writes. `CR3` (`FAULT_PC`) is writable so a trap handler can redirect the return address before `ERET`. Encodings `CR16` through `CR31` are reserved; `MTCR` to those encodings is ignored after the privilege check succeeds.

123. MFCR - mfcr Rd, CRn

Operation: `Rd = CRn`.

Assembler Syntax: `mfcr Rd, CRn`.

Attributes: Privileged except `CR6`; Memory: N; Size: Q.

Description: The processor reads the selected control register and writes the 64-bit value to `Rd`. `CRn` is a 5-bit encoded field; assigned control registers are listed in section 8.1.1. Register `R0` remains hardwired to zero; an `MFCR` with `Rd = R0` discards the value. Supervisor-mode reads of unassigned control-register numbers return zero.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xE7`.

Instruction Fields: Byte 0 holds opcode `0xE7`. Byte 1 bits 7-3 select destination register `Rd`; byte 1 bits 2-0 are reserved. Byte 2 bits 7-3 hold the control-register number `CRn`; byte 2 bits 2-0 are reserved. Bytes 3-7 are reserved by this instruction and ignored by the processor.

Exceptions: User-mode execution raises `FAULT_PRIV` (cause 5) unless `CRn` is `CR6` (`TP`). Reading `CR6` is permitted in user mode.

Notes: User-mode exception: MFCR is normally supervisor-only, but reading CR6 (TP) is permitted in user mode. Reading `CR5` composes the live MMU-enable, supervisor, SKEF, SKAC, and SUA bits. Reading `CR15` returns the active CPU-visible RAM size. Encodings `CR16` through `CR31` are reserved; `MFCR` from those encodings returns zero after the privilege check succeeds.

124. ERET - eret

Operation: `PC = CR3`; restore the saved privilege and trap-frame state.

Assembler Syntax: `eret`.

Attributes: Privileged; Memory: N; Size: --.

Description: The processor consumes the active trap frame, sets `PC` to `CR3` (`FAULT_PC`), and restores privilege state. If the interrupted context was user mode, `ERET` saves the current `R31` to `CR8` (`KSP`), loads `R31` from `CR12` (`USP`), enters user mode, enables interrupts, and clears the live `SUA` latch. If the interrupted context was supervisor mode, the processor remains supervisor and restores the live `SUA` latch from `CR14` (`SAVED_SUA`).

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xE8`.

Instruction Fields: Byte 0 holds opcode `0xE8`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: User-mode execution raises `FAULT_PRIV` (cause 5).

Notes: `ERET` does not pop the data stack. The pop described here is the architectural trap-frame stack in section 11.14.

125. TLBFLUSH - tlbflush

Operation: Invalidate every TLB entry.

Assembler Syntax: `tlbflush`.

Attributes: Privileged; Memory: N; Size: --.

Description: The processor invalidates all entries in the 64-entry translation lookaside buffer. Subsequent translated fetches and data accesses perform page-table walks until new entries are cached.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xE9`.

Instruction Fields: Byte 0 holds opcode `0xE9`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: User-mode execution raises `FAULT_PRIV` (cause 5).

Notes: Execute `TLBFLUSH` after bulk page-table modifications. `MTCR` writes that change PTBR or MMU enable also invalidate all TLB entries.

126. TLBINVAL - tlbinval Rs

Operation: Invalidate the TLB entry selected by `Rs >> 12`.

Assembler Syntax: `tlbinval Rs`.

Attributes: Privileged; Memory: N; Size: Q.

Description: The processor treats `Rs` as a virtual address, derives the virtual page number by shifting it right by 12, and invalidates the matching TLB entry. Bits 11-0 of the address are ignored for invalidation.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xEA`.

Instruction Fields: Byte 0 holds opcode `0xEA`. Byte 2 bits 7-3 select source register `Rs`; all other bits in bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: User-mode execution raises `FAULT_PRIV` (cause 5).

Notes: `Rs` contains an address within the affected virtual page, not a pre-shifted page number.

127. SYSCALL - syscall #imm32

Operation: `CR1 = imm32; CR2 = 6; CR3 = PC + 8; PC = CR4`.

Assembler Syntax: `syscall #imm32`.

Attributes: Any privilege level; Memory: N; Size: --.

Description: The processor enters the trap mechanism synchronously. It saves the syscall number from the immediate field in `CR1` (`FAULT_ADDR`), saves cause code 6 in `CR2` (`FAULT_CAUSE`), saves the address of the following instruction in `CR3` (`FAULT_PC`), switches to supervisor mode, and transfers control to `CR4` (`TRAP_VEC`).

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xEB`.

Instruction Fields: Byte 0 holds opcode `0xEB`. Bytes 4-7 hold unsigned `imm32` in little-endian order. Bytes 1-3 are reserved by this instruction and ignored by the processor.

Exceptions: None. `SYSCALL` is itself a trap source and records `FAULT_SYSCALL` (cause 6).

Notes: `ERET` from a syscall trap resumes at the instruction after `SYSCALL` unless the handler rewrites `CR3`.

128. SMODE - smode Rd

Operation: `Rd = supervisor ? 1 : 0`.

Assembler Syntax: `smode Rd`.

Attributes: Any privilege level; Memory: N; Size: Q.

Description: The processor writes `1` to `Rd` when the current privilege level is supervisor and `0` when the current privilege level is user. Register `R0` remains hardwired to zero; `SMODE R0` discards the result.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xEC`.

Instruction Fields: Byte 0 holds opcode `0xEC`. Byte 1 bits 7-3 select destination register `Rd`; byte 1 bits 2-0 are reserved. Bytes 2-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: `SMODE` reads the current mode only; privilege changes occur on trap entry and `ERET`.

129. CAS - cas Rd, disp(Rs), Rt

Operation: `old = mem64[addr]; if old == Rd then mem64[addr] = Rt; Rd = old`.

Assembler Syntax: `cas Rd, disp(Rs), Rt` or `cas Rd, (Rs), Rt`.

Attributes: Atomic 64-bit read-modify-write; sequentially consistent; full memory barrier.

Description: The processor compares the 64-bit memory value at `addr` with the original value of `Rd`. If they match, `Rt` is written to memory. `Rd` receives the old memory value regardless of whether the swap occurs.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Atomic memory RMW form; opcode = `0xED`.

Instruction Fields: Byte 0 holds opcode `0xED`. Byte 1 bits 7-3 select `Rd`, which supplies the comparison value before the operation and receives the old memory value after it. Byte 2 bits 7-3 select base register `Rs`. Byte 3 bits 7-3 select operand register `Rt`, the replacement value. Bytes 4-7 hold signed `imm32` displacement in little-endian order; `addr = Rs + signExtend32to64(imm32)`.

Exceptions: `FAULT_MISALIGNED` for an unaligned effective address or for an address in a non-atomic CPU address aperture. MMU translation faults use their normal cause codes. After optional MMU translation, the physical 8-byte word must lie entirely inside the processor's atomic RAM aperture; a translated address outside that aperture raises `FAULT_NOT_PRESENT`.

Notes: Use the returned old value in `Rd` to test whether the compare-and-swap succeeded.

130. XCHG - xchg Rd, disp(Rs), Rt

Operation: `old = mem64[addr]; mem64[addr] = Rt; Rd = old`.

Assembler Syntax: `xchg Rd, disp(Rs), Rt` or `xchg Rd, (Rs), Rt`.

Attributes: Atomic 64-bit read-modify-write; sequentially consistent; full memory barrier.

Description: The processor atomically replaces the 64-bit memory value at `addr` with `Rt` and returns the old memory value in `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Atomic memory RMW form; opcode = `0xEE`.

Instruction Fields: Byte 0 holds opcode `0xEE`. Byte 1 bits 7-3 select `Rd`, which receives the old memory value. Byte 2 bits 7-3 select base register `Rs`. Byte 3 bits 7-3 select operand register `Rt`, the replacement value. Bytes 4-7 hold signed `imm32` displacement in little-endian order; `addr = Rs + signExtend32to64(imm32)`.

Exceptions: `FAULT_MISALIGNED` for an unaligned effective address or for an address in a non-atomic CPU address aperture. MMU translation faults use their normal cause codes. After optional MMU translation, the physical 8-byte word must lie entirely inside the processor's atomic RAM aperture; a translated address outside that aperture raises `FAULT_NOT_PRESENT`.

Notes: No size suffix is defined; the operation is always 64-bit.

131. FAA - faa Rd, disp(Rs), Rt

Operation: `old = mem64[addr]; mem64[addr] = old + Rt; Rd = old`.

Assembler Syntax: `faa Rd, disp(Rs), Rt` or `faa Rd, (Rs), Rt`.

Attributes: Atomic 64-bit read-modify-write; sequentially consistent; full memory barrier.

Description: The processor atomically adds `Rt` to the 64-bit memory value at `addr` and returns the old memory value in `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Atomic memory RMW form; opcode = `0xEF`.

Instruction Fields: Byte 0 holds opcode `0xEF`. Byte 1 bits 7-3 select `Rd`, which receives the old memory value. Byte 2 bits 7-3 select base register `Rs`. Byte 3 bits 7-3 select operand register `Rt`, the addend. Bytes 4-7 hold signed `imm32` displacement in little-endian order; `addr = Rs + signExtend32to64(imm32)`.

Exceptions: `FAULT_MISALIGNED` for an unaligned effective address or for an address in a non-atomic CPU address aperture. MMU translation faults use their normal cause codes. After optional MMU translation, the physical 8-byte word must lie entirely inside the processor's atomic RAM aperture; a translated address outside that aperture raises `FAULT_NOT_PRESENT`.

Notes: Arithmetic wraps modulo 64 bits.

132. FAND - fand Rd, disp(Rs), Rt

Operation: `old = mem64[addr]; mem64[addr] = old & Rt; Rd = old`.

Assembler Syntax: `fand Rd, disp(Rs), Rt` or `fand Rd, (Rs), Rt`.

Attributes: Atomic 64-bit read-modify-write; sequentially consistent; full memory barrier.

Description: The processor atomically ANDs `Rt` with the 64-bit memory value at `addr` and returns the old memory value in `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Atomic memory RMW form; opcode = `0xF0`.

Instruction Fields: Byte 0 holds opcode `0xF0`. Byte 1 bits 7-3 select `Rd`, which receives the old memory value. Byte 2 bits 7-3 select base register `Rs`. Byte 3 bits 7-3 select operand register `Rt`, the bit mask. Bytes 4-7 hold signed `imm32` displacement in little-endian order; `addr = Rs + signExtend32to64(imm32)`.

Exceptions: `FAULT_MISALIGNED` for an unaligned effective address or for an address in a non-atomic CPU address aperture. MMU translation faults use their normal cause codes. After optional MMU translation, the physical 8-byte word must lie entirely inside the processor's atomic RAM aperture; a translated address outside that aperture raises `FAULT_NOT_PRESENT`.

Notes: No size suffix is defined; the operation is always 64-bit.

133. FOR - for Rd, disp(Rs), Rt

Operation: `old = mem64[addr]; mem64[addr] = old | Rt; Rd = old`.

Assembler Syntax: `for Rd, disp(Rs), Rt` or `for Rd, (Rs), Rt`.

Attributes: Atomic 64-bit read-modify-write; sequentially consistent; full memory barrier.

Description: The processor atomically ORs `Rt` with the 64-bit memory value at `addr` and returns the old memory value in `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Atomic memory RMW form; opcode = `0xF1`.

Instruction Fields: Byte 0 holds opcode `0xF1`. Byte 1 bits 7-3 select `Rd`, which receives the old memory value. Byte 2 bits 7-3 select base register `Rs`. Byte 3 bits 7-3 select operand register `Rt`, the bit mask. Bytes 4-7 hold signed `imm32` displacement in little-endian order; `addr = Rs + signExtend32to64(imm32)`.

Exceptions: `FAULT_MISALIGNED` for an unaligned effective address or for an address in a non-atomic CPU address aperture. MMU translation faults use their normal cause codes. After optional MMU translation, the physical 8-byte word must lie entirely inside the processor's atomic RAM aperture; a translated address outside that aperture raises `FAULT_NOT_PRESENT`.

Notes: The mnemonic is `FOR`; it is unrelated to structured-language loop syntax.

134. FXOR - fxor Rd, disp(Rs), Rt

Operation: `old = mem64[addr]; mem64[addr] = old ^ Rt; Rd = old`.

Assembler Syntax: `fxor Rd, disp(Rs), Rt` or `fxor Rd, (Rs), Rt`.

Attributes: Atomic 64-bit read-modify-write; sequentially consistent; full memory barrier.

Description: The processor atomically XORs `Rt` with the 64-bit memory value at `addr` and returns the old memory value in `Rd`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Atomic memory RMW form; opcode = `0xF2`.

Instruction Fields: Byte 0 holds opcode `0xF2`. Byte 1 bits 7-3 select `Rd`, which receives the old memory value. Byte 2 bits 7-3 select base register `Rs`. Byte 3 bits 7-3 select operand register `Rt`, the bit mask. Bytes 4-7 hold signed `imm32` displacement in little-endian order; `addr = Rs + signExtend32to64(imm32)`.

Exceptions: `FAULT_MISALIGNED` for an unaligned effective address or for an address in a non-atomic CPU address aperture. MMU translation faults use their normal cause codes. After optional MMU translation, the physical 8-byte word must lie entirely inside the processor's atomic RAM aperture; a translated address outside that aperture raises `FAULT_NOT_PRESENT`.

Notes: No size suffix is defined; the operation is always 64-bit.

135. SUAEN - suaen

Operation: `SUA = 1`.

Assembler Syntax: `suaen`.

Attributes: Privileged; Memory: N; Size: --.

Description: The processor sets the supervisor-user-access latch, `CR5` bit 4. When `SKAC` is enabled, this latch permits supervisor data accesses to pages marked user-accessible until the latch is cleared.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xF3`.

Instruction Fields: Byte 0 holds opcode `0xF3`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: User-mode execution raises `FAULT_PRIV` (cause 5).

Notes: Use `SUADIS` to close the same supervisor-user-access window.

136. SUADIS - suadis

Operation: `SUA = 0`.

Assembler Syntax: `suadis`.

Attributes: Privileged; Memory: N; Size: --.

Description: The processor clears the supervisor-user-access latch, `CR5` bit 4. Clearing an already-clear latch leaves architectural state unchanged.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xF4`.

Instruction Fields: Byte 0 holds opcode `0xF4`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: User-mode execution raises `FAULT_PRIV` (cause 5).

Notes: `SUADIS` does not change `SKEF` or `SKAC`; it only clears the live `SUA` latch.

---

5. Architectural Instruction Idioms

The following idioms use only real IE64 instructions and architecturally defined register behaviour.

5.1 Absolute Address Formation

Since R0 is hardwired to zero, `lea Rd, disp(r0)` loads the sign-extended 32-bit displacement as an address value.

```
lea r5, $A0000(r0)     ; r5 = $A0000
```

5.2 Full-Width Constant Formation

A full 64-bit constant is formed with `MOVE.L` for the low half followed by `MOVT` for the high half.

```
move.l r3, #$CAFEBABE
movt   r3, #$DEADBEEF
```

5.3 Zero-Comparison Branches

R0 can be used as the zero operand in all compare-and-branch instructions.

```
beq r5, r0, loop       ; branch if r5 == 0
bne r5, r0, loop       ; branch if r5 != 0
blt r5, r0, loop       ; branch if int64(r5) < 0
```

---

6. Addressing Modes

The IE64 supports the following addressing modes:

Mode	Syntax	Description	Used By
Immediate	`#imm`	32-bit immediate value, zero-extended to 64 bits	MOVE, ALU ops, FMOVECR, SYSCALL, WAIT
Register	`Rs` or `Rt`	Register contents (64-bit)	MOVE, ALU ops, branches, control-register moves
Register-indirect (data)	`(Rs)`	Memory at address in Rs	LOAD, STORE, FLOAD, FSTORE, DLOAD, DSTORE, atomics
Register-indirect (control)	`(Rs)`	Transfer control to address in Rs	JMP, JSR
Displacement	`disp(Rs)`	Address `Rs + signExtend(disp)`	LOAD, STORE, LEA, JMP, JSR, FLOAD, FSTORE, DLOAD, DSTORE, atomics
PC-relative	encoded signed `imm32` offset	`PC + signExtend(imm32)`	BRA, Bcc, JSR

6.1 Immediate Addressing

The 32-bit immediate (imm32) is zero-extended to 64 bits when used as an operand (`operand3 = uint64(imm32)`). This means:

- Unsigned values 0 to 0xFFFFFFFF can be loaded directly.
- Negative 32-bit values can be formed with MOVEQ, which sign-extends imm32.
- The X bit (byte 1, bit 0) must be 1 to select immediate mode.

6.2 Displacement Addressing

Used by LOAD, STORE, LEA, JMP, JSR_IND, FLOAD, FSTORE, DLOAD, DSTORE, and atomic RMW instructions. The displacement is stored in imm32 and sign-extended to 64 bits before being added to the base register:

```
addr = uint64(int64(Rs) + int64(int32(imm32)))
```

This provides a +/- 2 GiB displacement range around the base register. The CPU keeps the effective virtual address at full 64-bit width through optional MMU translation and physical memory dispatch.

When `imm32` is zero, the effective address is exactly the value in `Rs`. Control-transfer and memory instructions that encode a displacement consume the signed `imm32` field directly; the CPU does not inspect the written source form.

---

7. Branch Architecture

7.1 Compare-and-Branch

The IE64 uses compare-and-branch instructions instead of a separate flags register. Each conditional branch instruction encodes:

- Two source registers (Rs, Rt) for comparison
- A signed 32-bit PC-relative offset

The comparison and branch are performed atomically in a single instruction. This eliminates the need for separate compare (CMP) and branch instructions, and avoids hazards associated with flag registers in pipelined CPUs.

7.2 Register-Indirect Transfer

JMP (opcode `0x49`) provides register-indirect unconditional transfer. The target address is computed from a register plus an optional signed 32-bit displacement. This enables computed jumps, jump tables, and dispatch through register-held addresses.

JSR (opcode `0x54`) provides the same register-indirect addressing for subroutine calls. It pushes the return address before transferring control, so a standard `rts` returns to the caller.

7.3 PC-Relative Offsets

All branch offsets are signed 32-bit values stored in the imm32 field. The effective target address is:

```
target = PC + signExtend32to64(offset)
```

Where `PC` is the address of the branch instruction itself (not PC+8).

The instruction encoding stores a signed 32-bit offset. Targets outside the signed 32-bit PC-relative range are not representable by the branch encoding.

7.4 R0-as-Zero Idioms

Since R0 is hardwired to zero, comparisons against zero are natural:

Idiom	Instruction	Meaning
Branch if zero	`beq Rs, r0, label`	Rs == 0
Branch if nonzero	`bne Rs, r0, label`	Rs != 0
Branch if negative	`blt Rs, r0, label`	int64(Rs) < 0
Branch if non-negative	`bge Rs, r0, label`	int64(Rs) >= 0
Branch if positive	`bgt Rs, r0, label`	int64(Rs) > 0
Branch if non-positive	`ble Rs, r0, label`	int64(Rs) <= 0
Move zero to Rd	`move.q Rd, r0`	Rd = 0
Test equal to value	`move.q Rt, #val` then `beq Rs, Rt, label`	Rs == val

No condition-code register is required for these patterns.

7.5 Signed vs. Unsigned Comparisons

Branch	Comparison Type	Condition
BEQ	Equality (unsigned/signed irrelevant)	Rs == Rt
BNE	Equality (unsigned/signed irrelevant)	Rs != Rt
BLT	Signed	int64(Rs) < int64(Rt)
BGE	Signed	int64(Rs) >= int64(Rt)
BGT	Signed	int64(Rs) > int64(Rt)
BLE	Signed	int64(Rs) <= int64(Rt)
BHI	Unsigned	uint64(Rs) > uint64(Rt)

Branch	Comparison Type	Condition
BLS	Unsigned	uint64(Rs) <= uint64(Rt)

Note: For unsigned "greater than or equal" and "less than", use the complementary conditions: - Unsigned `Rs >= Rt`: Use `bls Rt, Rs, label` (swap operands and use BLS) or check with BHI + BEQ. - Unsigned `Rs < Rt`: Use `bhi Rt, Rs, label` (swap operands and use BHI).

---

8. Address Space and Reset State

8.1 CPU Address Space

IE64 defines 64-bit effective addresses. The programme counter, explicit LOAD/STORE effective addresses, stack addresses, FPU memory-operation addresses, atomic-operation addresses, and MMU translation inputs are all 64-bit quantities.

Property	Architectural definition
Effective-address width	64 bits
Data byte order	Little-endian
Instruction width	8 bytes
Integer memory widths	`.B`, `.W`, `.L`, `.Q`
Stack transfer width	8 bytes
Reset PC	`PROG_START` (`0x1000`)
Reset stack pointer	`STACK_START` (`0x9F000`)
Active RAM-size register	`CR_RAM_SIZE_BYTES` (`CR15`), read by `MFCR`

The IE64 CPU ISA defines effective-address behaviour and CPU-visible control state only. Addresses outside those CPU-visible rules have no additional meaning in this processor manual.

8.1.1 Control Register Numbers

Register	Number
`CR_PTBR`	CR0
`CR_FAULT_ADDR`	CR1
`CR_FAULT_CAUSE`	CR2
`CR_FAULT_PC`	CR3
`CR_TRAP_VEC`	CR4
`CR_MMU_CTRL`	CR5
`CR_TP`	CR6
`CR_INTR_VEC`	CR7
`CR_KSP`	CR8
`CR_TIMER_PERIOD`	CR9

Register	Number
`CR_TIMER_COUNT`	CR10
`CR_TIMER_CTRL`	CR11
`CR_USP`	CR12
`CR_PREV_MODE`	CR13
`CR_SAVED_SUA`	CR14
`CR_RAM_SIZE_BYTES`	CR15
Reserved	CR16-CR31

Unassigned control-register encodings are reserved. In supervisor mode, `MFCR` from a reserved encoding returns zero and `MTCR` to a reserved encoding has no effect. User-mode access still follows the privilege rule for the instruction: only `MFCR` `CR6` is user-readable.

8.2 Initial State After Reset

Register/State	Value
PC	`$1000` (PROG_START)
R0	`0` (hardwired)
R1-R29	`0`
R30	`$9F000`
R31 (SP)	`$9F000` (STACK_START)
Interrupt enabled	`false`
In-interrupt flag	`false`
Timer enabled	`false`
Timer state	TIMER_STOPPED (0)
Timer count	0
Timer period	0

8.3 Stack

The stack grows downward from `$9F000` toward lower addresses. All stack operations (PUSH, POP, JSR, RTS, RTI, interrupt entry) use 8-byte (64-bit) transfers.

```
High memory:
  $9F000  <-- Initial SP (STACK_START)
  $9EFF8  <-- SP after first PUSH
  $9EFF0  <-- SP after second PUSH
  ...
Low memory:
  $01000  <-- PROG_START (stack must not grow past programme area)
```

---

9. Interrupt/Timer System

9.1 Non-MMU Fixed Interrupt Vector

In the non-MMU timer-interrupt path, IE64 uses a single fixed interrupt vector. This vector is initialised to `0` on reset. No instruction or externally addressable register changes the fixed vector. When the MMU is enabled and CR7 (`INTR_VEC`) is nonzero, timer interrupts use the CR7/ERET model described in section 9.4 and section 11.12 instead.

The vector table area at `$0000` is reserved.

9.2 Timer Registers

The architectural IE64 timer contract is the CR9/CR10/CR11 control-register interface:

Control register	Name	Architectural role
CR9	TIMER_PERIOD	Auto-reload value in timer-step units
CR10	TIMER_COUNT	Current countdown value in timer-step units
CR11	TIMER_CTRL	Enable and interrupt-enable bits

The IE64 timer is CPU-integrated. The architectural programming contract is the control-register interface above.

9.3 Timer Operation

The timer countdown advances once per decoded instruction step, after the instruction word and operand fields have been fetched and before the decoded instruction body is executed:

1. When the timer is enabled and TIMER_COUNT is nonzero, the CPU decrements the count by one for each decoded IE64 instruction step.
2. When the count reaches zero, timer state becomes expired.
3. If the timer is still enabled, the count is reloaded from TIMER_PERIOD.
4. If interrupts are enabled and the CPU is not already servicing an interrupt, the interrupt handler fires. A handler that reads TIMER_COUNT after entry observes the reloaded value when the timer remained enabled.
5. If TIMER_COUNT is zero and TIMER_PERIOD is nonzero at the next timer step, the count is loaded from TIMER_PERIOD and the timer returns to the running state.

9.4 Non-MMU Interrupt Flow

The following describes timer-interrupt entry when the MMU is disabled, or when INTR_VEC is zero:

1. The timer interrupt is delivered only when interrupt delivery is enabled and the interrupt-active latch is clear.
2. The interrupt-active latch is set, preventing nested delivery.
3. Push the current PC onto the stack: `SP -= 8; mem[SP] = PC`.
4. Set `PC = interruptVector`.
5. The ISR executes. It must end with `RTI`.
6. `RTI` pops the return address: `PC = mem[SP]; SP += 8`.
7. `RTI` clears the interrupt-active latch, re-enabling interrupt delivery.

When the MMU is enabled and INTR_VEC (CR7) is nonzero, the unified timer interrupt model is used (see section 11.12). Timer interrupts save PC to FAULT_PC, set FAULT_CAUSE=8, perform automatic stack switching only when the interrupted code was in user mode, and jump to INTR_VEC. The handler returns via ERET. When the MMU is off or

INTR_VEC is zero, the fixed-vector model is used: the CPU pushes PC and jumps to `interruptVector`, returning via RTI.

9.5 Interrupt Programming Patterns

9.5.1 Unified Model (MMU enabled)

When the MMU is enabled, timer interrupts can be delivered through INTR_VEC (CR7) using the ERET-based trap model. The handler uses the same return path as syscalls and faults:

```
    ; Kernel initialisation (supervisor mode)
    move.l r1, #kernel_stack_top
    mtcr cr8, r1            ; KSP = kernel stack
    move.l r1, #timer_handler
    mtcr cr7, r1            ; INTR_VEC = handler address
    move.l r1, #44100
    mtcr cr9, r1            ; TIMER_PERIOD = 44100 timer steps
    move.l r1, #3
    mtcr cr11, r1           ; TIMER_CTRL = enable + interrupt enable
    ; ... set up page table, enable MMU, ERET to user mode ...

timer_handler:
    push r1
    push r2
    ; ... handle timer interrupt ...
    pop r2
    pop r1
    eret                    ; return to interrupted code (stack auto-restored)
```

9.5.2 Non-Programmable Fixed-Vector Model (MMU disabled)

The non-MMU push-PC/RTI delivery path uses the fixed reset vector and is not a programmable interrupt-vector ABI. On reset, `interruptVector` is zero, and there is no instruction that sets it. If a timer expires in this mode, delivery jumps to the fixed vector. Ordinary software cannot configure that vector.

Supervisor software that needs programmable IE64 timer interrupts should use the unified CR7/ERET model above: set KSP (CR8), INTR_VEC (CR7), TIMER_PERIOD (CR9), and TIMER_CTRL (CR11), then return from the handler with ERET.

9.6 SEI/CLI Semantics

Instruction	Effect
SEI	Sets TIMER_CTRL (CR11) bit 1, enabling interrupt delivery. The timer continues running; the next expiration after SEI will trigger an interrupt. Note: expirations that occurred while interrupts were disabled are not queued or replayed.
CLI	Clears TIMER_CTRL (CR11) bit 1, disabling interrupt delivery. Timer continues running and counting, but interrupts are not delivered.

Interrupts do not nest: while the interrupt-active latch is set, no new interrupt is delivered regardless of the interrupt-enable latch.

---

10. 64-bit Constant Loading

Because the imm32 field is only 32 bits wide, loading a full 64-bit constant requires two instructions.

10.1 Pattern: MOVE.L + MOVT

```
; Load 0xDEADBEEF_CAFEBABE into R5
move.l r5, #$CAFEBABE      ; R5 = 0x00000000_CAFEBABE (lower 32 bits)
movt   r5, #$DEADBEEF      ; R5 = 0xDEADBEEF_CAFEBABE (upper 32 bits set)
```

Step by step: 1. `move.l r5, #$CAFEBABE` -- loads the 32-bit immediate `$CAFEBABE` into R5. Since this is a `.L` (32-bit) operation, the result is `0x00000000_CAFEBABE`. 2. `movt r5, #$DEADBEEF` -- takes the current value of R5, clears the upper 32 bits, and ORs in `$DEADBEEF << 32`. Result: `0xDEADBEEF_CAFEBABE`.

MOVT operation: `Rd = (Rd & 0x00000000FFFFFFFF) | (imm32 << 32)`

10.2 MOVEQ Alternative

For signed 32-bit values that need sign-extension to 64 bits:

```
moveq r5, #-1      ; R5 = 0xFFFFFFFF_FFFFFFFF (sign-extended from 32-bit -1)
moveq r5, #$80000000  ; R5 = 0xFFFFFFFF_80000000 (sign-extended)
```

MOVEQ interprets imm32 as a signed 32-bit integer and sign-extends it: `Rd = int64(int32(imm32))`.

---

11. Memory Management Unit

The IE64 includes an optional 6-level sparse radix paged MMU that provides virtual-to-physical address translation, page-level access control, and a supervisor/user privilege model. When enabled, instruction fetches and explicit memory operations are translated through a software-managed page table. The MMU is disabled on reset; supervisor code must build a page table and explicitly enable translation.

11.1 Privilege Levels

The IE64 operates in one of two privilege levels:

Level	MMU_CTRL.1	Description
Supervisor	1	Privileged execution mode. Can execute MTCR, MFCR, ERET, TLBFLUSH, TLBINVAL, SUAEN, and SUADIS. Page U-bit checks do not restrict supervisor access unless SKEF or SKAC is enabled.
User	0	Restricted. Privileged instructions cause a fault (cause code 5), except `MFCR Rd, CR6` which is user-readable. Can only access pages with U=1.

On reset the CPU is in supervisor mode. Transitioning to user mode is done via ERET (which clears supervisor mode and jumps to FAULT_PC). Returning to supervisor mode occurs only via a trap (fault or SYSCALL), which implicitly sets supervisor mode before jumping to the trap vector. The SMODE instruction reads the current mode into a register for introspection.

11.2 Control Registers

Sixteen control registers manage the MMU, thread state, timer, stack switching, trap state, and the live RAM-size discovery slot. They are accessed via MTCR (write) and MFCR (read).

CR	Name	R/W	Description
CR0	PTBR	RW	Page Table Base Register. Physical address of the page table.
CR1	FAULT_ADDR	RW	Virtual address that caused the most recent fault, or the syscall number (imm32) for SYSCALL. Writable so handlers can communicate information back. See 11.14 for the trap-stack semantics.
CR2	FAULT_CAUSE	RW	Cause code of the most recent fault (see 11.7). Writable for handler flexibility. See 11.14 for the trap-stack semantics.
CR3	FAULT_PC	RW	PC saved at trap entry. Used by ERET to resume. **Writable**: trap handlers must be able to modify this before ERET (e.g., to skip a faulting instruction or redirect execution). See 11.14 for the trap-stack semantics.
CR4	TRAP_VEC	RW	Trap handler entry PC. Jumped to on any fault or SYSCALL. When the MMU is enabled this PC is translated like any other instruction fetch.
CR5	MMU_CTRL	Special	Bit 0: MMU enable (RW). Bit 1: supervisor mode (RO). Bit 2: SKEF enable (RW). Bit 3: SKAC enable (RW). Bit 4: SUA latch (RO via MTCR; mutated only by SUAEN/SUADIS). See 11.2.1.
CR6	TP	RW	Thread Pointer. User-readable via MFCR (exception to the normal supervisor-only rule). Writable only in supervisor mode via MTCR. Intended for thread-local storage (TLS) base address.
CR7	INTR_VEC	RW	Timer interrupt handler PC. When MMU is enabled and INTR_VEC is nonzero, timer interrupts use the unified ERET-based entry model instead of the non-MMU push-PC/RTI model. The PC is translated like any other instruction fetch. Supervisor-only.
CR8	KSP	RW	Kernel Stack Pointer. Automatically swapped with R31 on user-to-supervisor transitions. Supervisor-only.
CR9	TIMER_PERIOD	RW	Timer reload period in decoded-instruction timer-step units. Supervisor-only.
CR10	TIMER_COUNT	RW	Current timer countdown value. Supervisor-only.
CR11	TIMER_CTRL	RW	Bit 0 = timer enable, Bit 1 = interrupt enable. SEI/CLI are aliases for setting/clearing bit 1. Supervisor-only.
CR12	USP	RW	Saved User Stack Pointer. Readable/writable in supervisor mode for context switch. Set automatically on user-to-supervisor transition. Supervisor-only.
CR13	PREV_MODE	RO	Previous privilege mode saved by `trapEntry`: 0 = trap came from user mode, 1 = trap came from supervisor mode. Read-only; set automatically on any trap/interrupt entry. Used by fault handlers to distinguish user faults (kill task) from kernel faults (panic). See 11.14 for the trap-stack semantics.
CR14	SAVED_SUA	RW	SUA latch snapshot taken on trap entry and consumed on ERET. Readable by kernel handlers that observe the interrupted code path's SUA state; writable so handlers can stage a custom value before ERET. See 11.2.1 and 11.14. Supervisor-only.
CR15	RAM_SIZE_BYTES	RO	Read-only active CPU-visible RAM size in bytes. Every MFCR observes the current architectural value. MTCR to CR15 raises `FAULT_ILLEGAL_INSTRUCTION` (cause 11). Supervisor-only.

PTBR must point to the physical address of the level-0 page table. The architecture does not require a special PTBR alignment beyond the byte addressability of physical memory, but supervisor software should allocate the level-0 table on a natural boundary and zero-initialise it before installing entries.

TRAP_VEC must be set before enabling the MMU, or faults will jump to address 0.

MMU_CTRL bit 0 is the master enable. Writing 1 activates translation for all subsequent memory accesses. Bit 1 (supervisor mode) is read-only; it reflects the current privilege level and cannot be written by MTCR. Bits 2-4 are the supervisor execute/access guard controls described below.

11.2.1 SKEF / SKAC / SUA (MMU_CTRL bits 2-4)

These bits are the IE64 supervisor execute/access guards.

Bit	Name	MTCR writable?	Description
2	SKEF	Yes	Supervisor-Kernel-Execute-Fault. When set, a supervisor instruction fetch from a page with `PTE_U==1` raises `FAULT_SKEF` (cause 9).
3	SKAC	Yes	Supervisor-Kernel-Access-Check. When set, a supervisor read or write on a page with `PTE_U==1` raises `FAULT_SKAC` (cause 10), **unless** the `SUA` latch is also set.
4	SUA	No (RO via MTCR)	Supervisor-User-Access latch. Mutated only by the privileged `SUAEN` and `SUADIS` opcodes. When set and `SKAC` is enabled, kernel data accesses to user pages succeed; when clear they fault with `FAULT_SKAC`.

Trap-entry / ERET discipline. On trap entry the `SUA` latch is snapshotted into `CR_SAVED_SUA` (CR14) and then forcibly cleared so a kernel handler cannot inherit an open supervisor-user-access window from the interrupted code path. On ERET the saved value is restored into the live latch when returning to supervisor mode (nested return); user-mode ERET clears the live latch unconditionally. See 11.14 for how the trap stack preserves `CR_SAVED_SUA` across nested traps automatically.

Helper idiom. Kernel user-memory access must bracket every supervisor load/store against a user pointer with `SUAEN` and `SUADIS`:

```
    suaen                       ; open the access window
    load.b  r3, (r1)            ; user load
    store.b r3, (r2)            ; kernel store
    suadis                      ; close the access window
```

Kernel user-memory helpers must enforce this pattern around every supervisor access to user pages.

11.3 Page Table Format

The MMU uses a 6-level sparse radix page table for full 64-bit virtual addresses. Each page is 4 KiB, so virtual address bits 63:12 form a 52-bit VPN and bits 11:0 form the page offset.

Level layout:

Level	VPN Bits	Index Bits	Entries	Table Size
0 (top)	51..45	7	128	1 KiB
1	44..36	9	512	4 KiB
2	35..27	9	512	4 KiB
3	26..18	9	512	4 KiB
4	17..9	9	512	4 KiB
5 (leaf)	8..0	9	512	4 KiB

`CR_PTBR` points to the physical address of the level-0 table. Each entry is 8 bytes. Non-leaf entries use the same PTE encoding as leaf entries: the P bit marks the next-level table present, and the PPN field points to the next-level table.

Permission and A/D bits in non-leaf entries are ignored for non-leaf entries; the leaf entry controls access permissions.

11.4 Page Table Entry (PTE) Format

Bit(s)	Name	Description
0	P (Present)	Page is mapped. If P=0, any access faults with cause 0.
1	R (Read)	Read permission. If R=0, loads fault with cause 1.
2	W (Write)	Write permission. If W=0, stores fault with cause 2.
3	X (Execute)	Execute permission. If X=0, instruction fetch faults with cause 3.
4	U (User)	User-accessible. If U=0, user-mode access faults with cause 4.
5	A (Accessed)	Set by hardware on any successful translation (read, write, or execute).
6	D (Dirty)	Set by hardware on write access. Only set when the access is a store; reads and fetches do not set D.
11:7	--	Reserved; software should write 0. The MMU ignores these bits.
63:12	PPN	Physical Page Number (52 bits). Physical address = `(PPN << 12) | offset`.

A/D bit semantics:

- The A and D bits are set by the MMU after all permission checks have passed. Both TLB-hit and TLB-miss paths perform A/D updates.
- A is set on every successful translation regardless of access type (read, write, execute).
- D is set only on write accesses.
- The bits are written back to the page table entry in memory only when they change (i.e., when the bit was previously 0). This avoids unnecessary memory writes on repeated accesses to the same page.
- **Architectural constraint**: Page tables must reside in normal writable physical memory. The A/D write-back performs a physical store to the PTE; if the page table were placed over non-RAM storage, the write-back would not be a valid page-table update.
- Kernel software clears A/D bits by rewriting the PTE directly in memory and then flushing the TLB (via `TLBFLUSH` or `TLBINVAL`) to ensure the cached TLB entry is also updated. This is the basis for page reclamation and working-set estimation algorithms.

11.5 Virtual Address Format

Page count is no longer a fixed `MMU_NUM_PAGES = 8192`; software should derive relevant bounds from active visible RAM queried via `CR_RAM_SIZE_BYTES`.

11.6 MMU Instructions

Nine privilege and MMU opcodes are defined. All except SYSCALL and SMODE are privileged (supervisor-only); executing them in user mode faults with cause code 5 (privilege violation). MFCR has a special exception: reading CR6 (TP) is permitted in user mode.

Their instruction-by-instruction entries are part of the Complete Instruction Reference in section 4.9. This section defines the control-register and MMU state those instructions manipulate.

11.7 Trap Model

Traps are raised by faults (translation errors, permission violations) and by the SYSCALL instruction. On trap entry:

1. CR1 (FAULT_ADDR) is set to the faulting virtual address (for faults) or the syscall number (for SYSCALL).
2. CR2 (FAULT_CAUSE) is set to the cause code.
3. CR3 (FAULT_PC) is set to the relevant PC (see below).
4. If the trap came from user mode, automatic stack switching occurs: user R31 is saved to USP (CR12), R31 is loaded from KSP (CR8). Supervisor-origin traps do not swap stacks. See section 11.11.
5. The CPU switches to supervisor mode, or remains in supervisor mode for supervisor-origin traps.
6. PC is set to CR4 (TRAP_VEC).

Differentiated PC save: - **SYSCALL**: CR3 = PC + 8 (address of the instruction *after* SYSCALL). ERET resumes execution past the syscall. - **Faults**: CR3 = faulting PC (address of the instruction that caused the fault). ERET re-executes the faulting instruction after the handler fixes the page table.

This distinction means trap handlers do not need to adjust the return address; ERET always restores CR3 directly.

11.8 Fault Cause Codes

Code	Name	Trigger
0	Page Not Present	Absent PTE mapping or unavailable physical/atomic backing. Access to a page with P=0, access whose physical memory backing is unavailable, and an atomic operation outside the atomic RAM aperture all raise cause 0.
1	Read Denied	Load from a page with R=0.
2	Write Denied	Store to a page with W=0.
3	Execute Denied	Instruction fetch from a page with X=0.
4	User/Supervisor	User-mode access to a page with U=0.
5	Privilege Violation	User-mode execution of a privileged instruction (MTCR, MFCR except `MFCR Rd, CR6`, ERET, TLBFLUSH, TLBINVAL, SUAEN, SUADIS).
6	Syscall	SYSCALL instruction executed.
7	Misaligned	Atomic memory operation (CAS, XCHG, FAA, FAND, FOR, FXOR) with address not 8-byte aligned.
8	Timer Interrupt	Timer interrupt (delivered via INTR_VEC when MMU enabled).
9	SKEF	Supervisor instruction fetch from a page with `PTE_U==1` while `MMU_CTRL.SKEF` is set.
10	SKAC	Supervisor data read or write on a page with `PTE_U==1` while `MMU_CTRL.SKAC` is set and `MMU_CTRL.SUA` is clear.
11	Illegal Instruction	Opcode-level invariants the CPU cannot otherwise enforce. Currently raised only by `MTCR RAM_SIZE_BYTES, Rs` (the CR is read-only - see 11.2).

11.9 Translation Lookaside Buffer (TLB)

The MMU maintains a 64-entry direct-mapped translation lookaside buffer to cache page table lookups.

- **Indexing**: `TLB[VPN & 63]`. Each entry stores the VPN tag, physical page number, leaf-PTE physical address, and translation permission flags.
- **Lookup**: On every translated access, the TLB is checked first. On a hit (VPN tag matches), the cached translation and permission flags are used. On a miss, the page table is walked, the PTE is loaded, permission checks are performed, and the translation result is cached in the TLB.
- **Invalidation**: TLBFLUSH clears all 64 entries. TLBINVAL clears only the entry matching the given VA's VPN. Writing PTBR or MMU_CTRL via MTCR also flushes the entire TLB.
- **Coherency**: The TLB is not automatically coherent with page table memory. After modifying a PTE in memory, software must execute TLBINVAL for the affected page (or TLBFLUSH for bulk changes) before the new mapping takes effect.

11.10 W^X Page Policy

The IE64 page-table format supports a Write XOR Execute page policy by giving each page independent W and X permission bits:

- **Code pages**: P=1, X=1, W=0, with R optional. Execute-only user text (`P=1, R=0, X=1`) is a first-class contract.
- **Data/stack pages**: P=1, R=1, W=1, X=0 (readable + writable, not executable).

Supervisor page-table code should enforce the policy by avoiding PTEs that set both W and X. The MMU checks the requested access against the relevant bit but does not reject a PTE merely because both W and X are set. To load new code under a W^X policy, supervisor software maps the target pages as writable, writes the code, then remaps them as executable with appropriate TLB invalidation between steps.

11.11 Automatic Stack Switching

The IE64 provides automatic kernel/user stack separation via KSP (CR8) and USP (CR12). On any user-to-supervisor transition (SYSCALL, fault, or timer interrupt):

1. The previous privilege mode is saved in CPU trap state.
2. The current user R31 (stack pointer) is saved to USP (CR12).
3. R31 is loaded from KSP (CR8), giving the handler a kernel stack.
4. The CPU switches to supervisor mode.

On ERET:

1. If the previous mode was user: R31 is saved to KSP (CR8), R31 is restored from USP (CR12), and the CPU returns to user mode.
2. If the previous mode was supervisor: the CPU stays in supervisor mode with no stack swap.

This allows trap handlers to execute on a dedicated kernel stack without any software stack-switching prologue. The kernel must initialise KSP (via MTCR) before entering user mode for the first time.

11.12 Unified Timer Interrupt Model

The IE64 supports two timer interrupt models, selected automatically based on the MMU state and INTR_VEC (CR7):

Unified model (MMU enabled, INTR_VEC nonzero):

Timer interrupts are delivered through the same trap mechanism as faults and SYSCALLs:

1. PC is saved to CR3 (FAULT_PC).

2. CR2 (FAULT_CAUSE) is set to 8 (FAULT_TIMER).
3. If the interrupt interrupted user mode, automatic stack switching occurs (user R31 saved to USP, R31 loaded from KSP). Supervisor-origin timer interrupts do not swap stacks.
4. The CPU switches to supervisor mode, or remains in supervisor mode for supervisor-origin interrupts.
5. PC is set to INTR_VEC (CR7).

The handler returns via ERET, which restores the stack and privilege mode automatically. This model eliminates the need for RTI and provides a consistent trap-return path for all supervisor entry points.

Fixed-vector model (MMU disabled, or INTR_VEC is zero):

Timer interrupts use the classic push-PC/RTI mechanism:

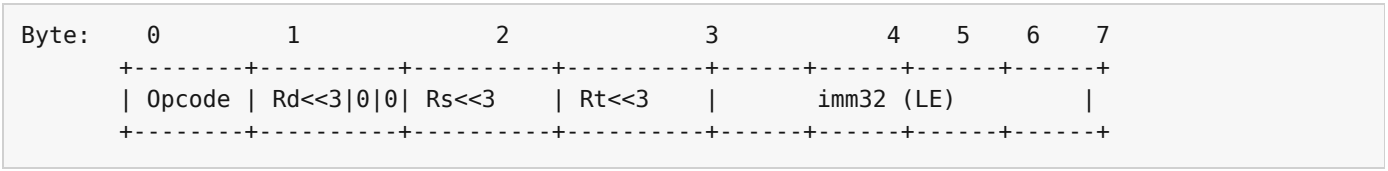
1. The current PC is pushed onto the stack: `SP -= 8; mem[SP] = PC`.
2. The interrupt-active latch is set, preventing nested delivery.
3. PC is set to `interruptVector`.
4. The handler returns via RTI, which pops PC and clears the interrupt-active latch.

This model is the defined timer-interrupt path when MMU vectoring is not active.

11.13 Atomic Memory Operations

The IE64 provides six atomic read-modify-write (RMW) instructions for lock-free synchronisation. These instructions use a dedicated encoding form that repurposes all three register fields and the immediate field:

Encoding: Memory RMW Form



- **Rd**: Destination register (receives the old value read from memory)
- **Rs**: Base register (memory address source)
- **Rt**: Operand register (value to swap, add, or use in bitwise operation)
- **imm32**: Signed displacement added to Rs to form the effective address

Effective address: `addr = uint64(int64(Rs) + int64(int32(imm32)))`

Instruction-by-instruction entries for CAS, XCHG, FAA, FAND, FOR, and FXOR are part of the Complete Instruction Reference in section 4.9.

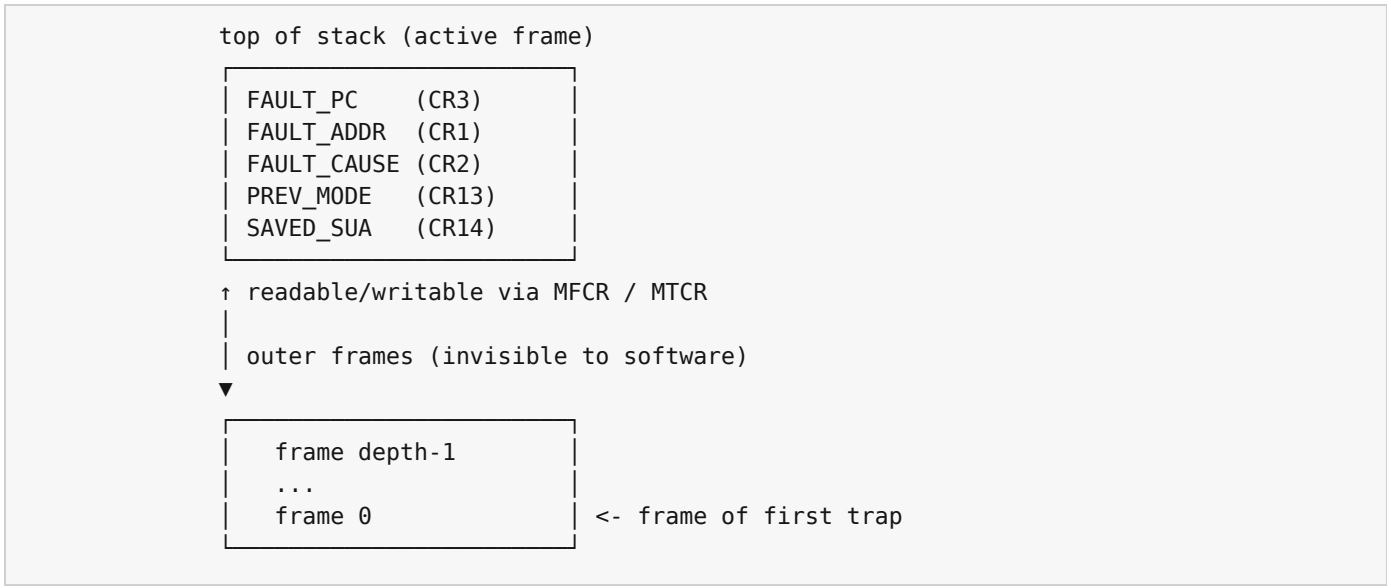
11.13.1 Shared Atomic Semantics

- **Size**: Always 64-bit (`.Q`). No size suffix is accepted; atomic operations operate on naturally-aligned 64-bit words only.
- **Alignment**: The effective address must be 8-byte aligned (`addr & 7 == 0`). A misaligned address causes a trap with FAULT_MISALIGNED (cause code 7).
- **Atomic RAM aperture only**: Atomic operations are valid only for aligned 64-bit words in the processor's atomic RAM aperture. An unaligned effective address or an address in a non-atomic CPU address aperture raises `FAULT_MISALIGNED` (cause 7). After optional MMU translation, the physical 8-byte word must lie entirely inside the processor's atomic RAM aperture. A translated address outside that aperture raises `FAULT_NOT_PRESENT` (cause 0).
- **Ordering**: All atomic operations are sequentially consistent. They act as full memory barriers.

- **MMU**: When the MMU is enabled, the effective address is translated as a write operation through the normal page-table translation path. A/D bits are set accordingly. The resulting physical word must lie inside the atomic RAM aperture; non-contiguous physical memory beyond that aperture is not a target for these instructions.
- **CAS (Compare-And-Swap)**: Reads the 64-bit value at `[addr]` into a temporary. If the temporary equals the current value of Rd, the value of Rt is written to `[addr]`. Regardless of whether the swap occurred, Rd receives the old value from memory. This allows the caller to detect success by comparing the returned old value against the expected value.

11.14 Trap-Frame Stack

Nested-trap state preservation is architectural rather than kernel-managed. The CPU owns a fixed-depth trap-frame stack that holds the outer trap's `FAULT_PC`, `PREV_MODE`, `CR_SAVED_SUA`, `FAULT_ADDR`, and `FAULT_CAUSE`. The live CR fields remain the canonical "top of stack" accessed through `MFCR` / `MTCR`; the stack is not directly visible to software.



Push. On trap entry (fault, SYSCALL, timer interrupt) the CPU snapshots the active frame (all five fields) onto the stack before overwriting them. The snapshot happens first; subsequent trap-entry bookkeeping (setting `PREV_MODE`, saving and clearing the `SUA` latch into `SAVED_SUA`) then writes the new active-frame values.

Pop. On `ERET` the CPU consumes the active frame (uses `FAULT_PC` as the new PC, restores the `SUA` latch from `SAVED_SUA` or clears it on user return, and swaps stacks on user return) and then pops the previous frame off the stack into the active fields. When the stack is empty the active fields are cleared to zero, matching the fresh-boot state.

Overflow. The stack depth is fixed. Exceeding it halts the CPU with a trap-stack overflow condition: runaway nested faults are always a kernel bug and must be visible, not silently dropped.

Implications for kernel handlers. Handlers do **not** need to save and restore `CR_FAULT_PC` or `CR_SAVED_SUA` on the kernel stack to survive a nested synchronous trap. The trap stack preserves the outer context automatically. Existing kernel code that performs a manual `MFCR CR_FAULT_PC` / `MTCR CR_FAULT_PC` prologue around a possibly-faulting region still works - such save/restore now writes the active frame, so the restore writes back the same value already preserved. The older handler pattern is thus redundant but harmless; new handlers should omit it.

Reset. Processor reset clears the trap stack to depth 0 and clears all trap-frame slots. After reset, no saved trap frame is architecturally visible.

Appendix A: Opcode Map

A.1 Instruction Set Summary

Opcode	Operation	Syntax
MOVE	Data Movement	`MOVE Rd, Rs / Rd, #imm`
MOVT	Data Movement	`MOVT Rd, #imm`
MOVEQ	Data Movement	`MOVEQ Rd, #imm`
LEA	Data Movement	`LEA Rd, disp(Rs)`
LOAD	Memory Access	`LOAD Rd, disp(Rs)`
STORE	Memory Access	`STORE Rd, disp(Rs)`
ADD	Arithmetic	`ADD Rd, Rs, Rt/#imm`
SUB	Arithmetic	`SUB Rd, Rs, Rt/#imm`
MULU	Arithmetic	`MULU Rd, Rs, Rt/#imm`
MULS	Arithmetic	`MULS Rd, Rs, Rt/#imm`
DIVU	Arithmetic	`DIVU Rd, Rs, Rt/#imm`
DIVS	Arithmetic	`DIVS Rd, Rs, Rt/#imm`
MOD	Arithmetic	`MOD Rd, Rs, Rt/#imm`
NEG	Arithmetic	`NEG Rd, Rs`
MODS	Arithmetic	`MODS Rd, Rs, Rt/#imm`
MULHU	Arithmetic	`MULHU Rd, Rs, Rt/#imm`
MULHS	Arithmetic	`MULHS Rd, Rs, Rt/#imm`
AND	Logical	`AND Rd, Rs, Rt/#imm`
OR	Logical	`OR Rd, Rs, Rt/#imm`
EOR	Logical	`EOR Rd, Rs, Rt/#imm`
NOT	Logical	`NOT Rd, Rs`
LSL	Shift	`LSL Rd, Rs, Rt/#imm`
LSR	Shift	`LSR Rd, Rs, Rt/#imm`
ASR	Shift	`ASR Rd, Rs, Rt/#imm`
CLZ	Shift	`CLZ.l Rd, Rs`
SEXT	Shift	`SEXT Rd, Rs`
ROL	Shift	`ROL Rd, Rs, Rt/#imm`
ROR	Shift	`ROR Rd, Rs, Rt/#imm`
CTZ	Shift	`CTZ.l Rd, Rs`
POPCNT	Shift	`POPCNT.l Rd, Rs`
BSWAP	Shift	`BSWAP.l Rd, Rs`

Opcode	Operation	Syntax
BRA	Branch	`BRA label`
BEQ	Branch	`BEQ Rs, Rt, label`
BNE	Branch	`BNE Rs, Rt, label`
BLT	Branch	`BLT Rs, Rt, label`
BGE	Branch	`BGE Rs, Rt, label`
BGT	Branch	`BGT Rs, Rt, label`
BLE	Branch	`BLE Rs, Rt, label`
BHI	Branch	`BHI Rs, Rt, label`
BLS	Branch	`BLS Rs, Rt, label`
JMP	Branch	`JMP (Rs) / disp(Rs)`
JSR	Subroutine	`JSR label`
RTS	Subroutine	`RTS`
PUSH	Stack	`PUSH Rs`
POP	Stack	`POP Rd`
JSR	Subroutine	`JSR (Rs) / disp(Rs)`
FMOV	FPU	`FMOV fd, fs`
FLOAD	FPU	`FLOAD fd, disp(rs)`
FSTORE	FPU	`FSTORE fs, disp(rs)`
FADD	FPU	`FADD fd, fs, ft`
FSUB	FPU	`FSUB fd, fs, ft`
FMUL	FPU	`FMUL fd, fs, ft`
FDIV	FPU	`FDIV fd, fs, ft`
FMOD	FPU	`FMOD fd, fs, ft`
FABS	FPU	`FABS fd, fs`
FNEG	FPU	`FNEG fd, fs`
FSQRT	FPU	`FSQRT fd, fs`
FINT	FPU	`FINT fd, fs`
FCMP	FPU	`FCMP rd, fs, ft`
FCVTIF	FPU	`FCVTIF fd, rs`
FCVTFI	FPU	`FCVTFI rd, fs`
FMOVI	FPU	`FMOVI fd, rs`
FMOVO	FPU	`FMOVO rd, fs`
FSIN	FPU	`FSIN fd, fs`
FCOS	FPU	`FCOS fd, fs`

Opcode	Operation	Syntax
FTAN	FPU	`FTAN fd, fs`
FATAN	FPU	`FATAN fd, fs`
FLOG	FPU	`FLOG fd, fs`
FEXP	FPU	`FEXP fd, fs`
FPOW	FPU	`FPOW fd, fs, ft`
FMOVECR	FPU	`FMOVECR fd, #idx`
FMOVSR	FPU	`FMOVSR rd`
FMOVCR	FPU	`FMOVCR rd`
FMOVSC	FPU	`FMOVSC rs`
FMOVCC	FPU	`FMOVCC rs`
DMOV	FPU64	`DMOV fd, fs`
DLOAD	FPU64	`DLOAD fd, disp(rs)`
DSTORE	FPU64	`DSTORE fs, disp(rs)`
DADD	FPU64	`DADD fd, fs, ft`
DSUB	FPU64	`DSUB fd, fs, ft`
DMUL	FPU64	`DMUL fd, fs, ft`
DDIV	FPU64	`DDIV fd, fs, ft`
DMOD	FPU64	`DMOD fd, fs, ft`
DABS	FPU64	`DABS fd, fs`
DNEG	FPU64	`DNEG fd, fs`
DSQRT	FPU64	`DSQRT fd, fs`
DINT	FPU64	`DINT fd, fs`
DCMP	FPU64	`DCMP rd, fs, ft`
DCVTIF	FPU64	`DCVTIF fd, rs`
DCVTFI	FPU64	`DCVTFI rd, fs`
FCVTSD	FPU64	`FCVTSD fd, fs`
FCVTDS	FPU64	`FCVTDS fd, fs`
DSIN	FPU64	`DSIN fd, fs`
DCOS	FPU64	`DCOS fd, fs`
DTAN	FPU64	`DTAN fd, fs`
DATAN	FPU64	`DATAN fd, fs`
DLOG	FPU64	`DLOG fd, fs`
DEXP	FPU64	`DEXP fd, fs`
DPOW	FPU64	`DPOW fd, fs, ft`

Opcode	Operation	Syntax
NOP	System	`NOP`
HALT	System	`HALT`
SEI	System	`SEI`
CLI	System	`CLI`
RTI	System	`RTI`
WAIT	System	`WAIT #usec`
MTCR	MMU	`MTCR CRn, Rs`
MFCR	MMU	`MFCR Rd, CRn`
ERET	MMU	`ERET`
TLBFLUSH	MMU	`TLBFLUSH`
TLBINVAL	MMU	`TLBINVAL Rs`
SYSCALL	MMU	`SYSCALL #imm32`
SMODE	MMU	`SMODE Rd`
CAS	Atomic	`CAS Rd, disp(Rs), Rt`
XCHG	Atomic	`XCHG Rd, disp(Rs), Rt`
FAA	Atomic	`FAA Rd, disp(Rs), Rt`
FAND	Atomic	`FAND Rd, disp(Rs), Rt`
FOR	Atomic	`FOR Rd, disp(Rs), Rt`
FXOR	Atomic	`FXOR Rd, disp(Rs), Rt`
SUAEN	System	`SUAEN`
SUADIS	System	`SUADIS`

A.2 Machine Opcode Encoding Map

Opcode Byte	Mnemonic	Category	Operands
`0x01`	MOVE	Data Movement	Rd, Rs / Rd, #imm
`0x02`	MOVT	Data Movement	Rd, #imm
`0x03`	MOVEQ	Data Movement	Rd, #imm
`0x04`	LEA	Data Movement	Rd, disp(Rs)
`0x10`	LOAD	Memory Access	Rd, disp(Rs)
`0x11`	STORE	Memory Access	Rd, disp(Rs)
`0x20`	ADD	Arithmetic	Rd, Rs, Rt/#imm
`0x21`	SUB	Arithmetic	Rd, Rs, Rt/#imm
`0x22`	MULU	Arithmetic	Rd, Rs, Rt/#imm
`0x23`	MULS	Arithmetic	Rd, Rs, Rt/#imm

Opcode Byte	Mnemonic	Category	Operands
0x24	DIVU	Arithmetic	Rd, Rs, Rt/#imm
0x25	DIVS	Arithmetic	Rd, Rs, Rt/#imm
0x26	MOD	Arithmetic	Rd, Rs, Rt/#imm
0x27	NEG	Arithmetic	Rd, Rs
0x28	MODS	Arithmetic	Rd, Rs, Rt/#imm
0x29	MULHU	Arithmetic	Rd, Rs, Rt/#imm
0x2A	MULHS	Arithmetic	Rd, Rs, Rt/#imm
0x30	AND	Logical	Rd, Rs, Rt/#imm
0x31	OR	Logical	Rd, Rs, Rt/#imm
0x32	EOR	Logical	Rd, Rs, Rt/#imm
0x33	NOT	Logical	Rd, Rs
0x34	LSL	Shift	Rd, Rs, Rt/#imm
0x35	LSR	Shift	Rd, Rs, Rt/#imm
0x36	ASR	Shift	Rd, Rs, Rt/#imm
0x37	CLZ	Shift	Rd, Rs
0x38	SEXT	Shift	Rd, Rs
0x39	ROL	Shift	Rd, Rs, Rt/#imm
0x3A	ROR	Shift	Rd, Rs, Rt/#imm
0x3B	CTZ	Shift	Rd, Rs
0x3C	POPCNT	Shift	Rd, Rs
0x3D	BSWAP	Shift	Rd, Rs
0x40	BRA	Branch	label
0x41	BEQ	Branch	Rs, Rt, label
0x42	BNE	Branch	Rs, Rt, label
0x43	BLT	Branch	Rs, Rt, label
0x44	BGE	Branch	Rs, Rt, label
0x45	BGT	Branch	Rs, Rt, label
0x46	BLE	Branch	Rs, Rt, label
0x47	BHI	Branch	Rs, Rt, label
0x48	BLS	Branch	Rs, Rt, label
0x49	JMP	Branch	(Rs) / disp(Rs)
0x50	JSR	Subroutine	label
0x51	RTS	Subroutine	(none)
0x52	PUSH	Stack	Rs

Opcode Byte	Mnemonic	Category	Operands
0x53	POP	Stack	Rd
0x54	JSR	Subroutine	(Rs) / disp(Rs)
0x60	FMOV	FPU	fd, fs
0x61	FLOAD	FPU	fd, disp(rs)
0x62	FSTORE	FPU	fs, disp(rs)
0x63	FADD	FPU	fd, fs, ft
0x64	FSUB	FPU	fd, fs, ft
0x65	FMUL	FPU	fd, fs, ft
0x66	FDIV	FPU	fd, fs, ft
0x67	FMOD	FPU	fd, fs, ft
0x68	FABS	FPU	fd, fs
0x69	FNEG	FPU	fd, fs
0x6A	FSQRT	FPU	fd, fs
0x6B	FINT	FPU	fd, fs
0x6C	FCMP	FPU	rd, fs, ft
0x6D	FCVTIF	FPU	fd, rs
0x6E	FCVTFI	FPU	rd, fs
0x6F	FMOVI	FPU	fd, rs
0x70	FMOVO	FPU	rd, fs
0x71	FSIN	FPU	fd, fs
0x72	FCOS	FPU	fd, fs
0x73	FTAN	FPU	fd, fs
0x74	FATAN	FPU	fd, fs
0x75	FLOG	FPU	fd, fs
0x76	FEXP	FPU	fd, fs
0x77	FPOW	FPU	fd, fs, ft
0x78	FMOVECR	FPU	fd, #idx
0x79	FMOVSR	FPU	rd
0x7A	FMOVCR	FPU	rd
0x7B	FMOVSC	FPU	rs
0x7C	FMOVCC	FPU	rs
0x80	DMOV	FPU64	fd, fs
0x81	DLOAD	FPU64	fd, disp(rs)
0x82	DSTORE	FPU64	fs, disp(rs)

Opcode Byte	Mnemonic	Category	Operands
0x83	DADD	FPU64	fd, fs, ft
0x84	DSUB	FPU64	fd, fs, ft
0x85	DMUL	FPU64	fd, fs, ft
0x86	DDIV	FPU64	fd, fs, ft
0x87	DMOD	FPU64	fd, fs, ft
0x88	DABS	FPU64	fd, fs
0x89	DNEG	FPU64	fd, fs
0x8A	DSQRT	FPU64	fd, fs
0x8B	DINT	FPU64	fd, fs
0x8C	DCMP	FPU64	rd, fs, ft
0x8D	DCVTIF	FPU64	fd, rs
0x8E	DCVTFI	FPU64	rd, fs
0x8F	FCVTSD	FPU64	fd, fs
0x90	FCVTDS	FPU64	fd, fs
0x91	DSIN	FPU64	fd, fs
0x92	DCOS	FPU64	fd, fs
0x93	DTAN	FPU64	fd, fs
0x94	DATAN	FPU64	fd, fs
0x95	DLOG	FPU64	fd, fs
0x96	DEXP	FPU64	fd, fs
0x97	DPOW	FPU64	fd, fs, ft
0xE0	NOP	System	(none)
0xE1	HALT	System	(none)
0xE2	SEI	System	(none)
0xE3	CLI	System	(none)
0xE4	RTI	System	(none)
0xE5	WAIT	System	#usec
0xE6	MTCR	MMU	CRn, Rs
0xE7	MFCR	MMU	Rd, CRn
0xE8	ERET	MMU	(none)
0xE9	TLBFLUSH	MMU	(none)
0xEA	TLBINVAL	MMU	Rs
0xEB	SYSCALL	MMU	#imm32
0xEC	SMODE	MMU	Rd

Opcode Byte	Mnemonic	Category	Operands
0xED	CAS	Atomic	Rd, disp(Rs), Rt
0xEE	XCHG	Atomic	Rd, disp(Rs), Rt
0xEF	FAA	Atomic	Rd, disp(Rs), Rt
0xF0	FAND	Atomic	Rd, disp(Rs), Rt
0xF1	FOR	Atomic	Rd, disp(Rs), Rt
0xF2	FXOR	Atomic	Rd, disp(Rs), Rt
0xF3	SUAEN	System	(none)
0xF4	SUADIS	System	(none)

A.3 Opcode Ranges

Range	Category
$01-$04	Data Movement
$10-$11	Memory Access
$20-$2A	Arithmetic
$30-$3D	Logical / Shift
$40-$49	Branches
$50-$54	Subroutine / Stack
$60-$7C	Floating Point (FPU)
$80-$97	Double-precision Floating Point (FPU64)
$E0-$E5	System
$E6-$EC	MMU
$ED-$F2	Atomic Memory Operations
$F3-$F4	Supervisor-user-access controls

Opcodes not listed above are reserved. Executing a reserved opcode enters the stopped processor state without architecturally advancing PC.

Appendix B: Encoding Examples

B.1 `move.l r5, #$CAFEBABE`

```
Opcode = 0x01 (MOVE)
Rd = 5, Size = 2 (.L), X = 1 (immediate)
Rs = 0, Rt = 0
imm32 = 0xCAFEBABE

Byte 0: 0x01
Byte 1: (5 << 3) | (2 << 1) | 1 = 0x28 | 0x04 | 0x01 = 0x2D
Byte 2: 0 << 3 = 0x00
Byte 3: 0 << 3 = 0x00
Bytes 4-7: 0xBE 0xBA 0xFE 0xCA (little-endian)

Binary: 01 2D 00 00 BE BA FE CA
```

B.2 `add.q r3, r1, r2`

```
Opcode = 0x20 (ADD)
Rd = 3, Size = 3 (.Q), X = 0 (register)
Rs = 1, Rt = 2
imm32 = 0

Byte 0: 0x20
Byte 1: (3 << 3) | (3 << 1) | 0 = 0x18 | 0x06 | 0x00 = 0x1E
Byte 2: 1 << 3 = 0x08
Byte 3: 2 << 3 = 0x10
Bytes 4-7: 0x00 0x00 0x00 0x00

Binary: 20 1E 08 10 00 00 00 00
```

B.3 `beq r1, r2, target` (target 24 bytes ahead)

```
Opcode = 0x41 (BEQ)
Rd = 0 (unused), Size = 3 (.Q), X = 0
Rs = 1, Rt = 2
imm32 = 24 (signed offset = +24)

Byte 0: 0x41
Byte 1: (0 << 3) | (3 << 1) | 0 = 0x06
Byte 2: 1 << 3 = 0x08
Byte 3: 2 << 3 = 0x10
Bytes 4-7: 0x18 0x00 0x00 0x00

Binary: 41 06 08 10 18 00 00 00
```

B.4 `store.b r7, 4(r10)`

```
Opcode = 0x11 (STORE)
Rd = 7, Size = 0 (.B), X = 1 (displacement non-zero)
Rs = 10, Rt = 0
imm32 = 4

Byte 0: 0x11
Byte 1: (7 << 3) | (0 << 1) | 1 = 0x38 | 0x00 | 0x01 = 0x39
Byte 2: 10 << 3 = 0x50
Byte 3: 0x00
Bytes 4-7: 0x04 0x00 0x00 0x00

Binary: 11 39 50 00 04 00 00 00
```

B.5 `push r15`

```
Opcode = 0x52 (PUSH)
Rd = 0 (unused), Size = 3 (.Q), X = 0
Rs = 15, Rt = 0
imm32 = 0

Byte 0: 0x52
Byte 1: (0 << 3) | (3 << 1) | 0 = 0x06
Byte 2: 15 << 3 = 0x78
Byte 3: 0x00
Bytes 4-7: 0x00 0x00 0x00 0x00

Binary: 52 06 78 00 00 00 00 00
```

B.6 `jmp (r5)`

```
Opcode = 0x49 (JMP)
Rd = 0 (unused), Size = 0, X = 0
Rs = 5, Rt = 0
imm32 = 0 (no displacement)

Byte 0: 0x49
Byte 1: 0x00
Byte 2: 5 << 3 = 0x28
Byte 3: 0x00
Bytes 4-7: 0x00 0x00 0x00 0x00

Binary: 49 00 28 00 00 00 00 00
```

B.7 `jsr 16(r3)`

```
Opcode = 0x54 (JSR indirect)
Rd = 0 (unused), Size = 0, X = 0
Rs = 3, Rt = 0
imm32 = 16 (displacement)

Byte 0: 0x54
Byte 1: 0x00
Byte 2: 3 << 3 = 0x18
Byte 3: 0x00
Bytes 4-7: 0x10 0x00 0x00 0x00

Binary: 54 00 18 00 10 00 00 00
```