Slicudis. RISC. Machine. Manual

ISA version 5.2

Written by Santiago Licudis

Contents

Recent changelog

- Removed INT, re-implemented **SYSCALL** and implemented **SYSBREAK** as part of the base System extension.
- Specified new interrupt/exception cause IDs

Introduction

SRM is an open source 32-bit big endian reduced instruction set architecture designed by Santiago Licudis with educational purposes and the potential of being used in real life applications.

Syntax specifications

This manual uses **System Verilog** and **C-like** syntax for the symbology, operators and concatenations.

The operand “**rd**” is defined as the destination register used by an instruction. “**rs1**” and “**rs2**” are defined as the source registers.

The operand “**imm**” is defined as an immediate value. All the immediate values are **sign-extended** unless it’s specified that it’s unsigned in the instruction definitions / notes.

The operand “**opcode**” indicates the main opcode of an instruction.

The operands “**fn4**” and “**fn7**” are **secondary opcodes** used by the instructions.

The symbol “$” is the position of the instruction/label that uses it.

Instruction Set naming convention

A specific syntax system is used for naming variations of the instruction set. The names start with “SRM”. Then the register size is specified (Example: SRM32). After that the extensions are specified (Example: SRM32SMA = 32-bit SRM with the Multiplication/Division and Atomic extensions)

Instruction extensions:

S - Base system extension
M - Multiplication and Division
A - Atomic extension (Work in progress)
H - Register reduction extension
L - Hypervisor extension (Work in progress)
F- Single precision Floating point extension (Work in progress)
D - Double precision Floating point extension (Work in progress)
Zmo - Memory ordering extension (Work in progress)
Zvm - Protected/Virtual memory extension (Work in progress)

Bit size extensions:

SRM32 - Standard 32-bit SRM
SRM64 - 64-bit extension

Examples:

Useful for small microcontrollers: SRM32H
Useful for a graphing calculator: SRM32MF
Technically most advanced SRM CPU: SRM64SMALFZmoZvm

Base Instructions

Formats

FMT	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DI	opcode						rd					imm [20:0]																				
DSS	opcode						rd					rs1					fn4				rs2					fn7						
DSI	opcode						rd					rs1					fn4				imm [11:0]											
SSI	opcode						imm [11:7]					rs1					fn4				rs2					imm [6:0]						

Arithmetic instructions

Opcode: 0x0 / 0b000000
Format: DSS
FN7: -

ADD: (Add)
Description: Adds registers **rs1** and **rs2**, and stores the result in register **rd**.
Syntax: add {rd}, {rs1}, {rs2}
FN4: 0x0

SUB: (Subtract)
Description: Subtracts register **rs1** by register **rs2**, and stores the result in register **rd**.
Syntax: sub {rd}, {rs1}, {rs2}
FN4: 0x1

AND: (Bitwise AND)
Description: Bitwise AND between registers **rs1** and **rs2**, and stores the result in register **rd**.
Syntax: and {rd}, {rs1}, {rs2}
FN4: 0x2

OR: (Bitwise OR)
Description: Bitwise OR between registers **rs1** and **rs2**, and stores the result in register **rd**.
Syntax: or {rd}, {rs1}, {rs2}
FN4: 0x3

XOR: (Bitwise XOR)
Description: Bitwise XOR between registers **rs1** and **rs2**, and stores the result in register **rd**.
Syntax: xor {rd}, {rs1}, {rs2}
FN4: 0x4

SHR: (Logical right shift)
Description: Shifts register **rs1** by register **rs2** to the right, and stores the result in register **rd**.
Syntax: shr {rd}, {rs1}, {rs2}
FN4: 0x5

ASR: (Arithmetic right shift)
Description: Shifts register **rs1** by register **rs2** to the right. The MSBs are set to the sign of register **rs1**. The result is stored in register **rd**.
Syntax: asr {rd}, {rs1}, {rs2}
FN4: 0x6

SHL: (Logical left shift)
Description: Shifts register **rs1** by register **rs2** to the left, and stores the result in register **rd**.
Syntax: shl {rd}, {rs1}, {rs2}
FN4: 0x7

CCH: (Addition carry check)
Description: Sets register **rd** to the carry out of register **rs1** plus register **rs2**
Syntax: cch {rd}, {rs1}, {rs2}
FN4: 0x8

BCH: (Subtraction borrow check)
Description: Sets register **rd** to the borrow out of register **rs1** minus register **rs2**
Syntax: bch {rd}, {rs1}, {rs2}
FN4: 0x9

Arithmetic instructions with immediate values

Opcode: 0x1 / 0b000001
Format: DSI

ADDI: (Add immediate)
Description: Adds registers **rs1** by **imm**, and stores the result in register **rd**.
Syntax: addi {rd}, {rs1}, {imm}
FN4: 0x0

ANDI: (Bitwise AND immediate)
Description: Bitwise AND between register **rs1** and **imm**, and stores the result in register **rd**.
Syntax: andi {rd}, {rs1}, {imm}
FN4: 0x2

ORI: (Bitwise OR immediate)
Description: Bitwise OR between register **rs1** and **imm**, and stores the result in register **rd**.
Syntax: ori {rd}, {rs1}, {imm}
FN4: 0x3

XORI: (Bitwise XOR immediate)
Description: Bitwise XOR between register **rs1** and **imm**, and stores the result in register **rd**.
Syntax: xori {rd}, {rs1}, {imm}
FN4: 0x4

SHRI: (Logical right shift immediate)
Description: Shifts register **rs1** by **imm** to the right, and stores the result in register **rd**.
Syntax: shri {rd}, {rs1}, {imm}
FN4: 0x5

ASRI: (Arithmetic right shift immediate)
Description: Shifts register **rs1** by **imm** to the right. The MSBs are set to the sign of register **rs1**. The result is stored in register **rd**.
Syntax: asri {rd}, {rs1}, {imm}
FN4: 0x6

SHLI: (Logical left shift immediate)
Description: Shifts register **rs1** by **imm** to the left, and stores the result in register **rd**.
Syntax: shli {rd}, {rs1}, {imm}
FN4: 0x7

CCHI: (Addition carry check immediate)
Description: Sets register **rd** to the carry out of register **rs1** plus **imm**.
Syntax: cchi {rd}, {rs1}, {imm}
FN4: 0x8

BCHI: (Subtraction borrow check immediate)
Description: Sets register **rd** to the borrow out of register **rs1** minus **imm.**
Syntax: bchi {rd}, {rs1}, {imm}
FN4: 0x9

LUI (Load upper immediate)

Opcode: 0x2 / 0b000010
Format: DI
Description: Register **rd** is set to (**imm** << 11).
Syntax: lui {rd}, {imm}

Memory storing

Opcode: 0x3 / 0b000011
Format: SSI
Note: The data format is big endian.

STB: (Store byte)
Description: memory**[rs1** + **imm**] = **rs2**[7:0].
Syntax: stb {rs2}, [{rs1}, {imm}]
FN4: 0x0

STW: (Store word)
Description: memory**[rs1** + **imm**] = **rs2**[15:0]. Address[0] is padded with 0.
Syntax: stw {rs2}, [{rs1}, {imm}]
FN4: 0x1

STD: (Store dword)
Description: memory**[rs1** + **imm**] = **rs2**[31:0]. Address[1:0] is padded with 0s.
Syntax: std {rs2}, [{rs1}, {imm}]
FN4: 0x2

Memory loading

Opcode: 0x4 / 0b000100
Format: DSI
Note: The data format is big endian.

LDB: (Load byte)
Description: **rd** = memory[**rs1** + **imm**][7:0]. Zero-extends
Syntax: ldb {rd}, [{rs1}, {imm}]
FN4: 0x0

LDSB: (Load signed byte)
Description: **rd** = memory[**rs1** + **imm**][7:0]. Sign-extends.
Syntax: ldsb {rd}, [{rs1}, {imm}]
FN4: 0x4

LDW: (Load word)
Description: **rd** = memory[**rs1** + **imm**][15:0]. Zero-extends and address[0] is padded with 0.
Syntax: ldw {rd}, [{rs1}, {imm}]
FN4: 0x1

LDSW: (Load signed word)
Description: **rd** = memory[**rs1** + **imm**][15:0]. Sign-extends and address[0] is padded with 0.
Syntax: ldsw {rd}, [{rs1}, {imm}]
FN4: 0x5

LDD: (Load dword)
Description: **rd** = memory[**rs1** + **imm**][31:0]. Zero-extends and address[1:0] is padded with 0s.
Syntax: ldw {rd}, [{rs1}, {imm}]
FN4: 0x2

JAL (Jump and link)

Opcode: 0x5 / 0b000101
Format: DI
Description: IP is set to (IP + **imm**) and the old value of IP+4 is stored to register **rd**.
Syntax: jal {rd}, {label} (**imm** is the relative position between IP and the label's address).

JALR (Jump to register and link)

Opcode: 0x6 / 0b000110
Format: DSI
FN4: 0x0
Description: IP is set to (**rs1** + **imm**) and the old value of IP+4 is stored to register **rd**.
Syntax: jalr {rd}, {rs1}, {imm}

Conditional jumping

Opcode: 0x7 / 0b000111
Format: SSI

JEQ: (Jump if equal)
Description: IP is set to (IP + **imm**) only if (**rs1** == **rs2)**. If not, no operation happens.
Syntax: jeq {rs1}, {rs2}, {label} (**imm** is the relative position between IP and the label's address).
FN1: 0x0

JLT: (Jump if less than)
Description: IP is set to (IP + **imm**) only if (**rs1** < **rs2)**. If not, no operation happens.
Syntax: jlt {rs1}, {rs2}, {label} (**imm** is the relative position between IP and the label's address).
FN1: 0x1

JSLT: (Jump if signed less than)
Description: IP is set to (IP + **imm**) only if signed (**rs1** < **rs2)**. If not, no operation happens.
Syntax: jslt {rs1}, {rs2}, {label} (**imm** is the relative position between IP and the label's address).
FN1: 0x2

JNE: (Jump if not equal)
Description: IP is set to (IP + **imm**) only if (**rs1** != **rs2)**. If not, no operation happens.
Syntax: jeq {rs1}, {rs2}, {label} (**imm** is the relative position between IP and the label's address).
FN1: 0x4

JGE: (Jump if greater of equal than)
Description: IP is set to (IP + **imm**) only if (**rs1** >= **rs2)**. If not, no operation happens.
Syntax: jlt {rs1}, {rs2}, {label} (**imm** is the relative position between IP and the label's address).
FN1: 0x5

JSGE: (Jump if signed greater or equal than)
Description: IP is set to (IP + **imm**) only if signed (**rs1** >= **rs2)**. If not, no operation happens.
Syntax: jslt {rs1}, {rs2}, {label} (**imm** is the relative position between IP and the label's address).
FN1: 0x6

Pseudo-instructions

Pseudo-instructions are virtual instructions (used by assemblers) that can be represented by one real instruction.

NOP: (No operation)
Description: No operation. | Conversion: add zr, zr, zr | Syntax: nop

NOT: (Bitwise NOT)
Description: **rd** = ~**rs1** | Conversion: xori {rd}, {rs1}, -1| Syntax: not {rd}, {rs1}

INC: (Increment)
Description: **rd** = **rs1**++ | Conversion: addi {rd}, {rs1}, 1 | Syntax: inc {rd}, {rs1}

DEC: (Decrement)
Description: **rd** = **rs1**- - | Conversion: addi {rd}, {rs1}, -1 | Syntax: dec {rd}, {rs1}

MOV: (Move)
Description: Move **rs1** to **rd** | Conversion: add {rd}, {rs1}, zr | Syntax: mov {rd}, {rs1}

Move IP:
Description: Move IP + 4 to **rd** | Conversion: jalr zr, {rd}, $+4 | Syntax: mov {rd}, ip

RET: (Return)
Description: Return from a function call | Conversion: jalr zr, rp, 0 | Syntax: mov {rd}, ip

CLR: (Clear)
Description: Clear **rd** | Conversion: xor {rd}, {rd}, {rd} | Syntax: clr {rd}

NEG: (Negate)
Description: Negate **rd** | Conversion: sub {rd}, zr, {rd} | Syntax: clr {rd}

SLT: (Set less than)
Description: **rd** = (**rs1** < **rs2**) ? 1 : 0 | Conversion: bch {rd}, {rs1}, {rs2}
Syntax: slt {rd}, {rs1}, {rs2}

JLE: (Jump if less or equal)
Description: Jump if (**rs1** <= **rs2**) | Conversion: jge {rs2}, {rs1}, {label}
Syntax: jle {rs1}, {rs2}, {label}

JGT: (Jump if greater than)
Description: Jump if (**rs1** > **rs2**) | Conversion: jlt {rs2}, {rs1}, {label}
Syntax: jgt {rs1}, {rs2}, {label}

JMP: (Jump)
Description: Jump | Conversion: jal zr, {label} | Syntax: jmp {label}

SGT: (Set if greater than)
Description: **rd** = (**rs1** > **rs2**) ? 1 : 0 | Conversion: slt {rd}, {rs2}, {rs1}
Syntax: sgt {rd}, {rs1}, {rs2}

SSGT: (Set if signed greater than)
Description: **rd** = (**rs1** > **rs2**) ? 1 : 0 | Conversion: sslt {rd}, {rs2}, {rs1}
Syntax: ssgt {rd}, {rs1}, {rs2}

JZ: (Jump if zero)
Description: Jump if (**rs1** == 0) | Conversion: jeq {rs1}, zr, {label}
Syntax: jz {rs1}, {label}

JNZ: (Jump if not zero)
Description: Jump if (**rs1** != 0) | Conversion: jne {rs1}, zr, {label}
Syntax: jnz {rs1}, {label}

JLZ: (Jump if less than zero)
Description: Jump if (**rs1** < 0) | Conversion: jslt {rs1}, zr, {label}
Syntax: jlz {rs1}, {label}

JGZ: (Jump if greater than zero)
Description: Jump if (**rs1** > 0) | Conversion: jslt zr, {rs1}, {label}
Syntax: jgz {rs1}, {label}

LEA: (Load effective address)
Description: Load **rd** with the effective address of [**rs1**, **imm**]
Conversion: addi {rd}, {rs1}, {imm}
Syntax: lea {rd}, [{rs1}, {imm}]

Macro-instructions

Macro-instructions are virtual instructions (used by assemblers) that can be represented by one real instruction. They take more than one instruction to recreate and sometimes require conditional compilation systems, like LDI.

LDI: (Load immediate)
Description: Load **rd** with an immediate value
Syntax: ldi {rd}, {imm}
Conversion:

```
if (imm is 0):
        place: clr {rd}
else if (imm is positive):
        if (imm[10:0] != 0):
                place: addi {rd}, zr, {imm}[10:0]
        if (imm[31:11] != 0):
                place: lui {rd}, zr, {imm}[31:11]
else if (imm is negative):
        if (imm[10:0] != 0x7ff):
                place: addi {rd}, zr, -1*{imm[10:0] (positive)}
        if (imm[31:11] != 0x1fffff):
                place: lui {rd}, zr. {imm}[31:11]
```

PUSH: (Push)
Description: Push **rs1** onto the stack
Syntax: push {rs1}
Conversion:

```
//p is the list of push instructions in a row
//y is the number of those pushes in a row
for (x = 0; x < y; x++):
        place: std {rs1 of p[x]}, [sp, x*-4]
place: addi sp, sp, y*-4
```

POP: (Pop)
Description: Pop to **rd** from the stack
Syntax: pop {rd}
Conversion:

```
//p is the list of pop instructions in a row
//y is the number of those pushes in a row
for (x = 0; x =< y; x = x++):
        place: ldb {rd of ps[x]}, [sp, x*4]
place: addi sp, sp, y*4
```

XCHG: (Exchange)
Description: Exchange between **rs1** and **rs2**
Syntax: xchg {rs1}, {rs2}
Conversion:
xor {rs1}, {rs1}, {rs2}
xor {rs2}, {rs1}, {rs2}
xor {rs1}, {rs1}, {rs2}

LD(B/SB/W/SW/D)UPDT: (Load byte/signed byte/word/signed word/dword)
Description: Load **rd** and load **rs1** with the effective address
Syntax: ld(b/sb/w/sw/d)updt {rd}, [{rs1}, {imm}] (Example: ldbupdt s0, [t0, 1])
Conversion:
ld(b/sb/w/sw/d) {rd}, [{rs1}, {imm}]
addi {rs1}, {rs1}, {imm}

ST(B/SB/W/SW/D)UPDT: (Store byte/word/dword)
Description: Store **rs2** and load **rs1** with the effective address
Syntax: st(b/sb/w/sw/d)updt {rs2}, [{rs1}, {imm}] (Example: stwupdt s0, [t0, 1])
Conversion:
st(b/sb/w/sw/d) {rs2}, [{rs1}, {imm}]
addi {rs1}, {rs1}, {imm}

SSLT: (Set if signed less than)
Description: rd = signed (rs1 < rs2) ? 1 : 0
Syntax: sslt {rd}, {rs1}, {rs2}
Conversion:
sub {rd}, {rs1}, {rs2}
shri {rd}, {rd}, 31

BST: (Bit set)
Description: rd[imm] = 1
Syntax: bst {rd}, {imm}
Conversion:
if (imm < 12):
    place: ori {rd}, {rd}, 1 << {imm}
else:
    place: ldi at, 1 << {imm}
    place: or {rd}, {rd}, at

BCL: (Bit clear)
Description: rd[imm] = 0
Syntax: bcl {rd}, {imm}
Conversion:
if (imm < 12):
    place: andi {rd}, {rd}, -1*(1 << {imm})
else:
    place: ldi at, -1*(1 << {imm})
    place: and {rd}, {rd}, at

BFL: (Bit flip)
Description: rd[imm] = 0
Syntax: bfl {rd}, {imm}
Conversion:
if (imm < 12):
    place: xori {rd}, {rd}, 1 << {imm}
else:
    place: ldi at, 1 << {imm}
    place: xor{rd}, {rd}, at

Registers

Register file

The register files in SRM cores consist of 3 special purpose registers and 29 general purpose registers. The H extension removes r16-31.

Register	Name	Function	Saver
r0	zr	Constant 0	-
r1	at	Assembler temporary	Caller
r2	gp	Global pointer	-
r3	tp	Thread pointer	-
r4	rp	Return pointer	Callee
r5	sp	Stack pointer	Callee
r6	fp	Frame pointer	Callee
r7-9	a0-2	Function arguments/return values	Caller
r10-12	s0-2	Saved registers	Callee
r13-20	t0-7	Temporaries	Caller
r21-28	s3-8	Saved registers	Callee
r28-31	a3-a7	Function arguments	Caller

Each register has their assigned application for function calls. **ZR** is always 0 and can't be modified, **AT** is a temporary used by assemblers for macro-instructions, **RP** contains the return address for function calls, **SP** points at the top-most value of the stack, FP is used to delineate the boundary between two stack frames, **GP** is used for fast access to global variables and data structures, **TP** points at the thread local memory, **A0-5** are used as function arguments and their return values, **S0-8** are used for local variables that are expected to keep their values after function calls, and **T0-7** are used for local variables that become garbage after function calls. The conventions for r1-31 aren't enforced but are highly recommended to follow.

Special purpose internal registers

These registers are outside of the register file and are used for specific applications.

IP: Instruction pointer register; points at the memory location of the current instruction being executed and its reset value is 0x0.

CSRs: Control and Status registers; they contain control and status values/flags. They are from the S (System) extension.

Standard extensions

These extensions expand the instruction set and the new instructions are assigned to specific opcodes to maintain global compatibility.

Base system extension

This extension includes system instructions, privilege modes (User, Kernel and Machine), control, status registers (CSRs) mainly for handling interrupts, and instructions for IO.
The System extension supports up to 4096 CSRs.

Register	Label	Function	Read privilege	Write privilege
csr0	mstatus	Machine status register	Machine	Machine
csr1	miva	Machine interrupt vector address	Machine	Machine
csr2	mra	Machine interrupt return address	Machine	Machine
csr3	mcause	Machine interrupt cause	Machine	Machine
csr4	mrpiv	Machine interrupt saved privilege mode	Machine	Machine
csr5-1023	*reserved*	*reserved*	Machine	Machine
csr6-2047	*reserved*	*reserved*	*reserved*	*reserved*
csr2048	kstatus	Kernel status register	Kernel	Kernel
csr2049	kiva	Kernel interrupt vector address	Kernel	Kernel
csr2050	kra	Kernel interrupt return address	Kernel	Kernel
csr2051	kcause	Kernel interrupt cause	Kernel	Kernel
csr2052	kpriv	Kernel interrupt saved privilege mode	Kernel	Kernel
csr2053-3071	*reserved*	*reserved*	Kernel	Kernel
csr3072-4095	*reserved*	*reserved*	User	User

MSTATUS: (Machine status register)

Bit position	Function
0	Enable external interrupts at machine level
1 - 2	Current privilege mode

KSTATUS: (Kernel status register)

Bit position	Function
0	Enable external interrupts at kernel level

PRIVILEGE LEVELS:

0 - USER (Least privileged): Restricted hardware and memory access
1 - KERNEL: Unrestricted access
2 - *RESERVED*
3 - MACHINE (Most privileged)

SYSCALL: (System call)
Opcode: 0x8 / 0b001000
Format: DSI
FN4: 0x0
Description: Transfers control to a higher privilege level (see the **Interrupt and exception handling** section for more details).
Syntax: syscall

SYSBREAK: (System break)
Opcode: 0x8 / 0b001000
Format: DSI
FN4: 0x1
Description: Transfers control to a higher privilege level for debugging purposes (see the **Interrupt and exception handling** section for more details).
Syntax: syscall

MRET: (Machine return)
Opcode: 0x9 / 0b001001
Format: DSS
FN4: 0x3
FN7: 0x0
Description: IP is set to the contents of the **mra** CSR and the privilege mode is set to the contents of **mpriv**. This instruction can only be executed on privilege level 3.
Syntax: mret

SYSRET: (System return)
Opcode: 0x9 / 0b001001
Format: DSS
FN4: 0x1
FN7: 0x0
Description: IP is set to the contents of the **kra** CSR and the privilege mode is set to the contents of **kpriv**. This instruction can only be executed on privilege level 1+.
Syntax: sysret

CSRW: (CSR write)
Opcode: 0xA / 0b001010
Format: DSI
FN4: 0x0
Description: csr[imm] is set to register **rs1** if the privilege mode allows the program to write to the selected CSR.
Syntax: csrr {rs1}, {csr}

CSRR: (CSR read)
Opcode: 0xA / 0b001010
Format: DSI
FN4: 0x1
Description: Register **rd** is set to csr[imm] if the privilege mode allows the program to write to the selected CSR (If not, the value returned is 0).
Syntax: csrr {rd}, {csr}

CSRS: (CSR bit set)
Opcode: 0xA / 0b001010
Format: DSI
FN4: 0x2
Description: csr[**imm**][{**rd**[0], **rs1**}] is set to 1 (rd and rs1 are just the operands, not the selected registers by them).
Syntax: csrs {csr}, {position}

CSRC: (CSR bit clear)
Opcode: 0xA / 0b001010
Format: DSI
FN4: 0x3
Description: csr[**imm**][{**rd**[0], **rs1**}] is set to 0 (rd and rs1 are just the operands, not the selected registers by them).
Syntax: csrs {csr}, {position}

INB: (Input byte):
Opcode: 0xB / 0b001011
Format: DSI
FN4: 0x0
Description: Register **rd** is set to the input data of port [**rs1** + **imm**][7:0]. Zero-extends. This instruction can only be executed on a privilege level equal or higher than level 1.
Syntax: inb {rd}, [{rs1}, {imm}]

INW: (Input word):
Opcode: 0xB / 0b001011
Format: DSI
FN4: 0x1
Description: Register **rd** is set to the input data of port [**rs1** + **imm**][15:0]. Zero-extends and the lowest bit of the address is padded with a 0. This instruction can only be executed on a privilege level equal or higher than level 1.
Syntax: inw {rd}, [{rs1}, {imm}]

IND: (Input dword):
Opcode: 0xB / 0b001011
Format: DSI
FN4: 0x2
Description: Register **rd** is set to the input data of port [**rs1** + **imm**][31:0]. Zero-extends and the lowest 2 bits of the address are padded with a 0s. This instruction can only be executed on a privilege level equal or higher than level 1.
Syntax: ind {rd}, [{rs1}, {imm}]

INSB: (Input signed byte):
Opcode: 0xB / 0b001011
Format: DSI
FN4: 0x4
Description: Register **rd** is set to the input data of port [**rs1** + **imm**][7:0]. Sign-extends and the lowest 2 bits of the address are padded with a 0s. This instruction can only be executed on a privilege level equal or higher than level 1.
Syntax: insb {rd}, [{rs1}, {imm}]

INSW: (Input signed word):
Opcode: 0xB / 0b001011
Format: DSI
FN4: 0x4
Description: Register **rd** is set to the input data of port [**rs1** + **imm**][15:0]. Sign-extends and the lowest 2 bits of the address are padded with a 0s. This instruction can only be executed on a privilege level equal or higher than level 1.
Syntax: insw {rd}, [{rs1}, {imm}]

OUTB: (Output byte):
Opcode: 0xC / 0b001100
Format: SSI
FN4: 0x4
Description: port [**rs1** + **imm**][7:0] is set to the contents of register **rs2** [7:0]. This instruction can only be executed on a privilege level equal or higher than level 1.
Syntax: outb {rs2}, [{rs1}, {imm}]

OUTW: (Output word):
Opcode: 0xC / 0b001100
Format: SSI
FN4: 0x4
Description: port [**rs1** + **imm**][15:0] is set to the contents of register **rs2** [15:0]. This instruction can only be executed on a privilege level equal or higher than level 1.
Syntax: outw {rs2}, [{rs1}, {imm}]

OUTD: (Output dword):
Opcode: 0xC / 0b001100
Format: SSI
FN4: 0x4
Description: port [**rs1** + **imm**][31:0] is set to the contents of register **rs2** [31:0]. This instruction can only be executed on a privilege level equal or higher than level 1.
Syntax: outd {rs2}, [{rs1}, {imm}]

64-bit extension (SRM64)

The 64-bit extension includes a control flag for the **msr** CSR called “64-bit mode” and 3 new instructions. The Q flag makes the core use 64-bit arithmetic/logic instructions and adjusts **cch, bch, mulh** and **smulh** to use the 64-bit carry/borrow out. When the Q flag is disabled, the upper 32 bits of the registers will be ignored during arithmetic operations

Memory storing:

Opcode: 0x3 / 0b000011
Format: SSI

STQ: (Store qword)
Description: memory[**rs1** + **imm**] = **rs2**[63:0].
Syntax: stq {rs2}, [{rs1}, {imm}]
FN4: 0x3

Memory loading:

Opcode: 0x4 / 0b000100

LDSD: (Load signed dword)
Description: **rd** = memory[**rs1** + **imm**][31:0]. Sign-extends.
Syntax: ldsd {rd}, [{rs1}, {imm}]
FN4: 0x6

LDSD: (Load qword)
Description: **rd** = memory[**rs1** + **imm**][63:0]. Zero-extends.
Syntax: ldq {rd}, [{rs1}, {imm}]
FN4: 0x3

Register reduction extension (H)

This extension removes registers r16-31. SRM cores without this extension are software compatible with the software for SRM CPUs with only 16 registers.

Multiplication/Division extension (M)

This extension has 8 sequential arithmetic instructions that allow multiplication and division in one single main opcode.

Opcode: 0xD / 0b001101
Format: DSS

MUL: (Multiply)
Description: Register **rs1** is multiplied by **rs2** and the result is stored in **rs2**.
Syntax: mul {rd}, {rs1}, {rs2}
FN4: 0x0

SMUL: (Signed multiply)
Description: **rd** = signed **rs1** * **rs2**
Syntax: smul {rd}, {rs1}, {rs2}
FN4: 0x1

MULH: (Multiply high)
Description: Register **rs1** is multiplied by **rs2** and the carry out of the result is stored in **rs2**.
Syntax: mulh {rd}, {rs1}, {rs2}
FN4: 0x2

SMUL: (Signed multiply high)
Description: **rd** = carry out of signed **rs1** * **rs2**
Syntax: smulh {rd}, {rs1}, {rs2}
FN4: 0x3

DIV: (Divide)
Description: Register **rs1** is divided by **rs2** and the quotient of the result is stored in **rs2**.
Syntax: div {rd}, {rs1}, {rs2}
FN4: 0x4

SDIV: (Signed divide)
Description: **rd** = signed **rs1** / **rs2**
Syntax: sdiv {rd}, {rs1}, {rs2}
FN4: 0x5

MOD: (Modulo)
Description: Register **rs1** is divided by **rs2** and the remainder of the result is stored in **rs2**.
Syntax: mod {rd}, {rs1}, {rs2}
FN4: 0x6

SMOD: (Signed modulo)
Description: **rd** = signed **rs1** % **rs2**
Syntax: smod {rd}, {rs1}, {rs2}
FN4: 0x7

M macro-instructions:

(MUL/SMUL/MULH/SMULH/DIV/SDIV/MOD/SMOD)I: ([...] Immediate)
Description: **rd** = **rs1** [...] **imm**
Syntax: (mul/smul/mulh/smulh/div/sdiv/mod/smod)i {rd}, {rs1}, {imm}
Convertion:
ldi at, {imm}
(mul/smul/mulh/smulh/div/sdiv/mod/smod) {rd}, {rs1}, at

MADD/SMADD: (Multiply / Signed multiply and accumulate)
Description: **rd** += **rs1** * **rs2**
Syntax: madd/smadd {rd}, {rs1}, {imm}
Convertion:
mul at, {rs1}, {rs2}
add {rd}, {rs}, at

(MADD/SMADD)I: (Multiply / Signed multiply immediate and accumulate)
Description: **rd** += **rs1** * **rs2**
Syntax: (madd/smadd)i {rd}, {rs1}, {imm}
Convertion:
ldi at, {imm}
(s)mul at, {rs1}, at
add {rd}, {rs}, at

Interrupt and exception handling

Definition of interrupt: *An interrupt is a request for the core to interrupt currently executing code (when permitted), so that the event can be processed in a timely manner. If the request is accepted, the processor will suspend its current activities, save its state, and execute a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event. This interruption is often temporary, allowing the software to resume normal activities after the interrupt handler finishes.*

Definition of exception: *Type of interrupt triggered when an internal error is detected inside the core.*

The process of triggering interrupts varies depending on the privilege mode that the core was in when the interrupt was triggered.

MACHINE LEVEL (LEVEL 3):

1. If **mstatus**[0] is 0, the external interrupt is ignored or placed in a queue (chosen by the manufacturer). In case of an internal interrupt, the checking process is skipped.
2. If **mstatus**[0] (Unless it's an internal interrupt) is 1, the following processes happen:
3. IP is copied to the **mra** CSR.
4. IP is set to the contents of the **miva** CSR

KERNEL LEVEL (LEVEL 1):

1. If **kstatus**[0] is 0, the external interrupt is ignored or placed in a queue (chosen by the manufacturer). In case of an internal interrupt, the checking process is skipped.
2. If **kstatus**[0] (Unless it's an internal interrupt) is 1, the following processes happen:
3. IP is copied to the **mra** CSR.
4. IP is set to the contents of the **miva** CSR.
5. The privilege mode is saved to the **mpriv** CSR.
6. The privilege mode is set to level 3 (Machine mode).

USER LEVEL (LEVEL 0):

1. IP is copied to the **kra** CSR.
2. IP is set to the contents of the **kiva** CSR..
3. The privilege mode is saved to the **kpriv** CSR.
4. The privilege mode is set to level 1 (Kernel mode).

Interrupt cause IDs

When interrupts are triggered, **mcause** or **kcause** (depending on the level on which the interrupt was triggered) are updated with the type of interrupt that was triggered.

The standard IDs used are:

0x0: System call from user mode
0x1: System call from kernel mode
0x2: *Reserved*
0x3: System call from machine mode
0x4: System break

EXAMPLE:

Using a table of addresses for each type of interrupt

```
Handling_section:
 ;Context switch logic here…
 csrr kcause, t0              ;Get the interrupt cause
 shli t0, t0, 2               ;Adjust
 ldi t1, TableBaseAddress     ;Get the interrupt table address
 add t0, t0, t1               ;Calculate the address
 ldd t0, [t0, 0]              ;Fetch the vector
 jalr zr, t0, 0               ;Jump to the selected routine
;...
```