

# Slicudis RISC Machine



The SRM architecture is a flagless RISC ISA designed by Santiago Licudis with the aim of surpassing current RISC architectures.

The base architecture consists of a processor with a bank of 32 registers, each of 32 bits. The sizes of the address bus and the data bus (in the base architecture) are at least 32 bits each, giving access to 4 GiB of memory. Instructions have a fixed size of 32 bits, making the fetch process faster.

Instructions have an opcode size of 6 bits (64 opcodes). 9 of the opcodes are reserved for the base instructions, leaving the 55 remaining spaces for extensions.

## Index

- 2 - Registers.
- 3 - Status Register.
- 4 - Address locations and Instruction formats.
- 5 - Base instruction set.
- 6 - Pseudo-Instructions.
- 7 - 64-bit extension.
- 7 - Multiplication and Division extension.
- 8 - Legal.

# Registers

SRM processors contain thirty-two registers that are used for arithmetic operations, holding data, addresses, etc. These registers are used for general purpose or for specific operations.

Register	Label	Function	Saver
r0	zr	Constant 0 (Read-Only)	-
r1	sr	Status Register (Kernel read/write only)	-
r2	ir	Interrupt Return Pointer (Kernel read/write only)	-
r3	at	Assembler temporary	Caller
r4	rp	Subroutine Return Pointer	Caller
r5	sp	Stack Pointer	Callee
r6	fp	Frame Pointer	Callee
r7	gp	Global Pointer	-
r8-10	a0-a2	Function Arguments/Return	Caller
r11-12	s0-1	Saved registers	Callee
r13-20	t0-7	Temporaries	Caller
r21-28	s2-9	Saved registers	Callee
r29-31	a3-a5	Function Arguments	Caller

**ZR:** Constant 0

**SR:** The processor's status register, which contains the status flags. It can only be read and modified in Kernel mode.

**IR:** Holds the return address for an interrupt subroutine. It can only be read and modified in Kernel mode

**AT:** Used by compilers to hold temporal values for complex pseudoinstructions

**RP:** Holds the return address for a function call.

**SP:** Stack pointer; Points to the top-most value of the stack.

**FP:** Frame pointer; Used to delineate the boundary between two stack frames.

**GP:** Used for fast access to global variables and data structures.

**A0-A5:** Used as arguments and return values in function calls.

**S0-S9:** Used for local variables that are expected to keep their values after function calls.

**T0-T7:** Used for local variables that become garbage after function calls.

# Status Register

The Status Register contains the control flags used by the processor. This register can only be modified in Kernel mode, which is enabled by interrupts (Software, Hardware and Reset).

Status Register	Extension
0 - Kernel mode (K)	Base ISA
1 - Saved privilege level (S)	Base ISA
2 - Enable hardware interrupts (EI)	Base ISA
3 - 64-bit mode (Q)	64-bit extension
4 - Virtual mode (VM)	Virtual memory ext.

- **Kernel mode:** Is set by interrupts (Software, Hardware and Resets) and enables functions that can only be executed by the Kernel.
- **Saved privilege level:** Contains the privilege level that the CPU had before an interrupt was triggered
- **Enable hardware interrupts:** Enables external hardware trigger interrupt subroutines.
- **Virtual mode:** Programs will use virtual memory when this flag is set.
- **64-bit mode:** ALU operations use 64 bits when this flag is set.

# Address locations

SRM processors must use these locations for defining the code for jumping to the reset location or the interrupt subroutines.

Address	Label	Name	Size
0x0	s_int	Software INT vector	4 bytes
0x4	h_int	Hardware INT vector	4 bytes
0x8	rst_loc	Reset Location	-

- **Software INT vector:** Contains the subroutine address of system calls
- **Hardware INT vector:** Contains the subroutine address of hardware interrupts
- **Reset Location:** Its address is the reset value of the program counter (0x8)

## Base instruction formats

The base instruction set uses 5 formats to allocate the parameters and operands.

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]																					
I	opcode						imm [25:0]																									
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]						

General Assembly syntax:

- DI: {Mnemonic} {rd}, {imm} //Example: LUI t5, 0x20
- I: {Mnemonic} {imm} //Example: JMP label
- DSS: {Mnemonic} {rd}, {rs1}, {rs2} //Example: ADD s0, s1, s2
- DSI: {Mnemonic} {rd}, {rs1}, {imm} //Example: ADDI s0, s1, 0x40
- SSI: {Mnemonic} {rs1}, {rs2}, {imm} //Example: JEQ a0, a1, label

Specific Assembly syntax:

- STB/STW/STD {rs2}, [{rs1}, {imm}] //Example: STD s0, [t0, 20]
- LDB/LDW/LDD {rd}, [{rs1}, {imm}] //Example: LDB a2, [gp, -64]
- SYSCALL/SYSRET //Example: SYSCALL

# Base instruction set

Every SRM processor must be able to execute the base instructions to keep global compatibility between all SRM processors.

Inst.	Name	Description	FMT	opcode	fn4	fn7	Note
add	Add	$rd = rs1 + rs2$	DSS	0x00	0x0	0x0	-
sub	Subtract	$rd = rs1 - rs2$	DSS	0x00	0x1	0x0	-
and	AND	$rd = rs1 \& rs2$	DSS	0x00	0x2	0x0	-
or	OR	$rd = rs1   rs2$	DSS	0x00	0x3	0x0	-
xor	XOR	$rd = rs1 \wedge rs2$	DSS	0x00	0x4	0x0	-
shr	Logical right shift	$rd = rs1 \gg rs2$	DSS	0x00	0x5	0x0	-
asr	Aritmethic right shift	$rd = rs1 \ggg rs2$	DSS	0x00	0x6	0x0	Sign-extends
shl	Logical left shift	$rd = rs1 \ll rs2$	DSS	0x00	0x7	0x0	-
cch	Addition carry check	$rd = \text{cout of } (rs1 + rs2)$	DSS	0x00	0x8	0x0	-
bch	Subtraction borrow check	$rd = \text{borrow of } (rs1 - rs2)$	DSS	0x00	0x9	0x0	-
addi	Add imm.	$rd = rs1 + \text{imm}$	DSI	0x01	0x0	-	Unsigned imm
subi	Subtract imm.	$rd = rs1 - \text{imm}$	DSI	0x01	0x1	-	Unsigned imm
andi	AND imm.	$rd = rs1 \& \text{imm}$	DSI	0x01	0x2	-	Unsigned imm
ori	OR imm.	$rd = rs1   \text{imm}$	DSI	0x01	0x3	-	Unsigned imm
xori	XOR imm.	$rd = rs1 \wedge \text{imm}$	DSI	0x01	0x4	-	Unsigned imm
shri	Logical right shift imm.	$rd = rs1 \gg \text{imm}$	DSI	0x01	0x5	-	Unsigned imm
asri	Aritmethic right shift imm.	$rd = rs1 \ggg \text{imm}$	DSI	0x01	0x6	-	Unsigned imm
shli	Logical left shift imm.	$rd = rs1 \ll \text{imm}$	DSI	0x01	0x7	-	Unsigned imm
cchi	Addition carry check imm.	$rd = \text{cout of } (rs1 + \text{imm})$	DSI	0x01	0x8	-	Unsigned imm
bchi	Subtraction borrow check imm.	$rd = \text{borrow of } (rs1 - \text{imm})$	DSI	0x01	0x9	-	Unsigned imm
lui	Load upper imm.	$rd = \text{imm} \ll 11$	DI	0x02	-	-	Unsigned imm
stb	Store byte	$m[rs1 + \text{imm}][7:0] = rs2[7:0]$	SSI	0x03	0x0	-	-
stw	Store word	$m[rs1 + \text{imm}][15:0] = rs2[15:0]$	SSI	0x03	0x1	-	-
std	Store dword	$m[rs1 + \text{imm}][31:0] = rs2[31:0]$	SSI	0x03	0x2	-	-
ldb	Load byte	$rd = m[rs1 + \text{imm}][7:0]$	DSI	0x04	0x0	-	Zero-extends
ldw	Load word	$rd = m[rs1 + \text{imm}][15:0]$	DSI	0x04	0x1	-	Zero-extends
ldd	Load dword	$rd = m[rs1 + \text{imm}][31:0]$	DSI	0x04	0x2	-	Zero-extends
ldsb	Load signed byte	$rd = m[rs1 + \text{imm}][7:0]$	DSI	0x04	0x4	-	Sign-extends
ldsw	Load signed word	$rd = m[rs1 + \text{imm}][15:0]$	DSI	0x04	0x5	-	Sign-extends
jal	Jump and link	$rd = ip + 4; ip += \text{imm}$	DI	0x05	-	-	Signed imm
jalr	Jump and link reg.	$rd = ip + 4; ip = rs1 + \text{imm}$	DSI	0x06	-	-	Signed imm
jeq	Jump if ==	If $(rs1 == rs2): ip += \text{imm}$	SSI	0x07	0x0	-	-
jlt	Jump if <	If $(rs1 < rs2): ip += \text{imm}$	SSI	0x07	0x1	-	-
jslt	Jump if signed <	If signed $(rs1 < rs2): ip += \text{imm}$	SSI	0x07	0x2	-	-
jsv	Jump if subtraction overflow	If V in $(rs1 - rs2): ip += \text{imm}$	SSI	0x07	0x3	-	-
jne	Jump if !=	If $(rs1 != rs2): ip += \text{imm}$	SSI	0x07	0x4	-	-
jge	Jump if >=	If $(rs1 \geq rs2): ip += \text{imm}$	SSI	0x07	0x5	-	-
jsge	Jump if signed >=	If signed $(rs1 \geq rs2): ip += \text{imm}$	SSI	0x07	0x6	-	-
jnsv	Jump if not subtraction overflow	If !V in $(rs1 - rs2): ip += \text{imm}$	SSI	0x07	0x7	-	-
syscall	System call	$ir = ip; ip = ir; s = k; k = 1$	DSS	0x08	0x0	0x0	-
sysret	System return	$k = s; ip = ir$	DSS	0x08	0x1	0x0	Kernel-only

# PSEUDO-INSTRUCTIONS

Pseudo-instructions are “instructions” that are actually the equivalent of specific instructions, but are used to program in a less complicated way in assembly language. For example, the SRM ISA does not have an LDI instruction, due to the small fixed size of the instructions. That is why LUI is used to load an instruction immediately above into a register and ADDI for the instruction immediately below, but LDI can be written in SRM assembly language and the assembler will convert it into the actual instructions.

Pseudoinstruction	Name	Equivalent	Description
ldi (rd), (imm)	Load Imm.	if(imm[31:10] != 0): lui (rd), (imm[31:10]) if(imm[9:0] != 0): addi (rd), (rd), (imm[9:0])	rd = imm
inc (rd)	Increment	addi (rd), (rd), 1	rd++
dec (rd)	Decrement	subi (rd), (rd), 1	rd--
mov (rd), (rs)	Move	addi (rd), zr, (rs)	rd = rs
mov (rd), ip	Move IP reg.	jal (rd), 4	rd = ip
ret	Return	jalr zr, rp, 0	ip = rp
nop	No operation	add zr, zr, zr	No operation
clr (rd)	Clear	xor (rd), (rd), (rd)	rd = 0
psh (rs)	Push	subi sp, sp, 4 std (rs), sp, 0	sp -= 4 m[sp] = rs
pop (rd)	Pop	ldd (rd), sp addi sp, sp, 4	rd = m[sp] sp += 4
neg (rd), (rs)	Negate	sub (rd), zr, (rs)	rd = 0 - rs
xchg (rs1), (rs2)	Exchange	mov at, (rs1) mov (rs1), (rs2) mov (rs2), at	rs1 = rs2 rs2 = rs1
slt (rd), (rs1), (rs2)	Set less than	bch (rd), (rs1), (rs2)	rd = (rs1 < rs2) ? 1 : 0
jle (rs1), (rs2), (imm)	Jump if <=	jge (rs2), (rs1), (imm)	if (rs1 <= rs2): ip += imm
jgt (rs1), (rs2), (imm)	Jump if >	jlt (rs2), (rs1), (imm)	if (rs1 > rs2): ip += imm
jmp	Jump	jal zr, (imm)	ip = imm

# 64-BIT EXTENSION

This extension implements 64-bit memory instructions (LDQ and STQ) and support for 64-bit arithmetic operations (which are enabled by the Q status bit). SRM processors with this extension still support the 32-bit instructions (If Q = 0). If Q is disabled, the upper 32 bits of the registers will be padded with 0x0000.

Inst.	Name	Description	Type	opcode	fn1	fn7	Note
stq	Store qword	$m[rs1 + imm][63:0] = rs2[63:0]$	SSI	0x03	0x3	-	-
ldq	Load qword	$rd = m[rs1 + imm][63:0]$	DSI	0x04	0x3	-	Zero-extends
lds	Load signed dword	$rd = m[rs1 + imm][31:0]$	DSI	0x04	0x6	-	Sign-extends

# MUL/DIV EXTENSION

This extension implements signed and unsigned multiplication, division and modulo.

Inst.	Name	Description	Type	opcode	fn1	fn8	Note
mul	Multiply	$rd = rs1 * rs2$	DSS	0x09	0x0	-	-
smul	Multiply (Signed)	$rd = rs1 * rs2$	DSS	0x09	0x1	-	Signed operands
mlu	Multiply (Upper)	$rd = \text{cout of } (rs1 * rs2)$	DSS	0x09	0x2	-	-
smlu	Multiply (Signed) (Upper)	$rd = \text{cout of } (rs1 * rs2)$	DSS	0x09	0x3	-	Signed operands
div	Divide	$rd = rs1 / rs2$	DSS	0x09	0x4	-	-
sdiv	Divide (Signed)	$rd = rs1 / rs2$	DSS	0x09	0x5	-	Signed operands
mod	Modulo	$rd = rs1 \% rs2$	DSS	0x09	0x6	-	-
smod	Signed Modulo	$rd = rs1 \% rs2$	DSS	0x09	0x7	-	Signed operands

# LEGAL

This project is under the MIT license, meaning that companies/people have to follow the terms the license:

MIT License

Copyright (c) 2024 Santiago Licudis

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.