Slicudis RISC Machine

The SRM architecture is a flagless RISC ISA that was created 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 and reading/writing 4 bytes of data in a single clock cycle. Instructions have the size of 32 bits, making the fetch process faster and more efficient.

Instructions have an opcode size of 5 bits (32 opcodes). 9 of the opcodes are reserved for the base instructions, leaving the 23 remaining spaces for extensions.

Index

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	-
r2	IP	Instruction Pointer (Read-Only)	-
r3	IR	Interrupt Return Pointer	-
r4	RP	Subroutine Return Pointer	Caller
r5	SP	Stack Pointer	Callee
r6	FP	Frame Pointer	Callee
r7	GP	Global Pointer	-
r8	A0	Function Argument/Output 0	Caller
r9	A1	Function Argument/Output 1	Caller
r10	A2	Function Argument/Output 2	Caller
r11	S0	Saved register 0	Callee
r12	S1	Saved register 1	Callee
r13	T0	Temporary register 0	Caller
r14	T1	Temporary register 1	Caller
r15	T2	Temporary register 2	Caller
r16	T3	Temporary register 3	Caller
r17	T4	Temporary register 4	Caller
r18	T5	Temporary register 5	Caller
r19	T6	Temporary register 6	Caller
r20	T7	Temporary register 7	Caller
r21	S2	Saved register 2	Callee
r22	S3	Saved register 3	Callee
r23	S4	Saved register 4	Callee
r24	S5	Saved register 5	Callee
r25	S6	Saved register 6	Callee
r26	S7	Saved register 7	Callee
r27	S8	Saved register 8	Callee
r28	S9	Saved register 9	Callee
r29	A3	Function Argument/Output 3	Caller
r30	A4	Function Argument/Output 4	Caller
r31	A5	Function Argument/Output 5	Caller

ZR: Constant 0

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.

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

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

RP: Holds the return address for a function call.

IR: Holds the return address for an interrupt subroutine.

IP: Holds the current value of the instruction pointer.

SR: The processor's status register, which contains the status flags and it can only be modified when Kernel mode is enabled.

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

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
0 - Kernel mode (K)
1 - Enable hardware interrupts (I)
2 - Protected memory mode (P)
3 - 64-bit mode (Q)

- **Kernel mode:** Is set by interrupts (Software, Hardware and Reset) and enables functions that can only be executed by the Kernel.
- **Enable hardware interrupts:** Enables the execution of interrupt subroutines.
- **Protected mode:** Programs use virtual memory when this bit is active. (Only applied if the processor has the protected mode extension).
- **64-bit mode:** ALU operations use 64 bits when this bit is set. (Only applied if the processor has the 64-bit extension).

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 8 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
RI	opcode					rd					imm [21:0]																					
RI2	opcode					imm [21:17]					rs1					imm [16:0]															
I	opcode					imm [26:0]																										
RRR	opcode					rd					rs1					fn1				rs2											
RRI	opcode					rd					rs1					fn1				imm [12:0]											
RRI2	opcode					imm [12:8]					rs1					fn1				rs2				imm [7:0]							
RR	opcode					rd					rs1																					
N	opcode																															

Assembly syntax:

- Arithmetic/Logic: {Mnemonic}, {Destination}, {Source A}, {Source B}
- Arithmetic/Logic (Imm): {Mnemonic}, {Destination}, {Source A}, {Immediate}
- Memory: {Mnemonic} {Data source/destination}, {Address source}, {Immediate}
- Jumping: {Mnemonic} {Destination address} //The assembler converts the destination to the relative address for the instruction
- Indirect jumping: {Mnemonic} {Source register}, {Immediate}
- Conditional jumping: {Mnemonic} {Source A}, {Source B}, {Destination address} //The assembler converts the destination to the relative address for the instruction
- Syscall: {Mnemonic}

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	fn1	Note
syscall	System call	ir = ip; ip = ir; k = 1	N	0x08	-	-
lui	Load upper imm.	rd = imm << 10	RI	0x02	-	Clears lower bits
stb	Store byte	m[rs1 + imm][7:0] = rs2[7:0]	RRI2	0x03	0x0	Signed imm
stw	Store word	m[rs1 + imm][15:0] = rs2[15:0]	RRI2	0x03	0x1	Signed imm
std	Store dword	m[rs1 + imm][31:0] = rs2[31:0]	RRI2	0x03	0x2	Signed imm
ldb	Load byte	rd = m[rs1 + imm][7:0]	RRI	0x04	0x0	Zero-extends, signed imm
ldw	Load word	rd = m[rs1 + imm][15:0]	RRI	0x04	0x1	Zero-extends, signed imm
ldd	Load dword	rd = m[rs1 + imm][31:0]	RRI	0x04	0x2	Zero-extends, signed imm
ldsb	Load signed byte	rd =m[rs1 + imm][7:0]	RRI	0x04	0x4	Sign-extends, signed imm
ldsw	Load signed word	rd = m[rs1 + imm][15:0]	RRI	0x04	0x5	Sign-extends, signed imm
irj	Indirect register jump	ip = rs1 + imm	RI2	0x05	-	Signed imm
jmp	Jump	ip += imm	I	0x06	-	Signed imm
jeq	Jump if equal	If Z in rs1-rs2: ip += imm	RRI2	0x07	0x0	Signed imm
jlt	Jump if lower than	If C in rs1-rs2: ip += imm	RRI2	0x07	0x1	Signed imm
jslt	Jump if signed lower than	If S in rs1-rs2: ip += imm	RRI2	0x07	0x2	Signed imm
jsv	Jump if subtraction overflow	If V in rs1-rs2: ip += imm	RRI2	0x07	0x3	Signed imm
jne	Jump if not equal	If !Z in rs1-rs2: ip += imm	RRI2	0x07	0x4	Signed imm
jge	Jump if greater or equal	If !C in rs1-rs2: ip += imm	RRI2	0x07	0x5	Signed imm
jsge	Jump if signed higher or equal	If !S in rs1-rs2: ip += imm	RRI2	0x07	0x6	Signed imm
jnsv	Jump if not subtraction overflow	If !V in rs1-rs2: ip += imm	RRI2	0x07	0x7	Signed imm
add	Add	rd = rs1 + rs2	RRR	0x00	0x0	-
sub	Subtract	rd = rs1 - rs2	RRR	0x00	0x1	-
and	AND	rd = rs1 & rs2	RRR	0x00	0x2	-
or	OR	rd = rs1 | rs2	RRR	0x00	0x3	-
xor	XOR	rd = rs1 ^ rs2	RRR	0x00	0x4	-
shr	Logical right shift	rd = rs1 >> rs2	RRR	0x00	0x5	-
asr	Aritmethic right shift	rd = rs1 >>> rs2	RRR	0x00	0x6	Sign-extends
shl	Logical left shift	rd = rs1 << rs2	RRR	0x00	0x7	-
cch	Addition carry check	rd = cout of (rs1 + rs2)	RRR	0x00	0x8	-
bch	Subtraction borrow check	rd = borrow of (rs1 - rs2)	RRR	0x00	0x9	-
addi	Add imm.	rd = rs1 + imm	RRI	0x01	0x0	-
subi	Subtract imm.	rd = rs1 - imm	RRI	0x01	0x1	-
andi	AND imm.	rd = rs1 & imm	RRI	0x01	0x2	-
ori	OR imm.	rd = rs1 | imm	RRI	0x01	0x3	-
xori	XOR imm.	rd = rs1 ^ imm	RRI	0x01	0x4	-
shri	Logical right shift imm.	rd = rs1 >> imm	RRI	0x01	0x5	-
asri	Aritmethic right shift imm.	rd = rs1 >>> imm	RRI	0x01	0x6	-
shli	Logical left shift imm.	rd = rs1 << imm	RRI	0x01	0x7	-
cchi	Addition carry check imm.	rd = cout of (rs1 + imm)	RRI	0x01	0x8	-
bchi	Subtraction borrow check imm.	rd = borrow of (rs1 - imm)	RRI	0x01	0x9	-

Memory misalignments (without extensions for handling them) are handled by ignoring the lower bits of an address, depending on the size of the manipulated data. The first 2 bits of the address are ignored in LDD and STD, the first bit of the address is ignored in LDW and STW and no bit of the address is ignored in LDB and STB. The ignored bits are replaced with “0”. The base storing and loading instructions use the big endian data format.

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 doesn’t have an LDI instruction, because of the small fixed size of the instructions. That’s why LUI is used for loading an upper immediate to a register and ADDI for the lower immediate, but you can type LDI in the SRM assembly language, and the assembler will convert it to the real instructions.

Pseudoinstructions	Equivalent	Description
ldi (rd), (imm)	if(imm[31:0] != 0): lui (rd), (imm[31:10]) if(imm[9:0] != 0): addi (rd), (rd), (imm[9:0])	rd = imm
inc (rd)	addi (rd), (rd), 1	rd++
dec (rd)	subi (rd), (rd), 1	rd--
mov (rd), (rs)	addi (rd), zr, (rs)	rd = rs
call (imm)	addi rp, ip, 8 jmp (imm)	rp = ip + 8; ip += imm
ret	irj rp, 0	ip = rp
nop	add zr, zr, zr	No operation
clr (rd)	xor (rd), (rd), (rd)	rd = 0
psh (rs)	subi sp, sp, 4 std (rs), sp	sp -= 4; m[sp] = rs
pop (rd)	ldd (rd), sp addi sp, sp, 4	rd = m[sp]; sp += 4
sysret	irj ir, 0	ip = ir
neg (rd), (rs)	sub (rd), zr, (rs)	rd = 0 - rs

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)

Inst.	Name	Description	Type	opcode	fn1	Note
stq	Store qword	m[rs1 + imm][63:0] = rs2[63:0]	RRI2	0x03	0x3	Signed imm
ldq	Load qword	rd = m[rs1 + imm][63:0]	RRI	0x04	0x3	Zero-extends, signed imm
ldsd	Load signed dword	rd = m[rs1 + imm][31:0]	RRI	0x04	0x6	Sign-extends, signed imm

MUL/DIV EXTENSION

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

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

LEGAL

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