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). 11 of the opcodes are reserved for the base instructions, leaving the 15 remaining spaces for extensions.

Index

Registers

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

Register	Label	Function	Saver
r0	ZR	Constant 0 (Read-Only)	-
r1	SR	Status Register (Read-Only)	-
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: Holds the current value of the Status Register

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

Status Register

The Status Register contains the flags of the last ALU operations and control bits used by the processor. The status bits can only be set/reset by the kernel (except for K, which is only set/reset by interrupts and URT)

Status Register	Function
0 - Kernel mode (K)	Enable the control over the Status Register
1 - Enable hardware interrupts (I)	Enable hardware interrupts
2 - Protected memory mode (P)	Set the memory controller to protected mode
3 - 64-bit mode (Q)	Set the aritmethic operations to 64-bit mode

- **Kernel mode:** Is set by interrupts or the SYSCAL instruction and reset by the URT instruction.

- **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 ext.)

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

Base instruction set

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

INSTRUCTION FORMATS: The base instruction set uses 9 formats to allocate the parameters and operands.

<table>
<tr><th>FORMAT</th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>RI</td><td colspan="5">OPCODE</td><td colspan="5">C</td><td colspan="22">IMMEDIATE [21:0]</td></tr>
<tr><td>RI2</td><td colspan="5">OPCODE</td><td colspan="5">IMM [21:17]</td><td colspan="5">A</td><td colspan="17">IMMEDIATE [16:0]</td></tr>
<tr><td>I</td><td colspan="5">OPCODE</td><td colspan="27">IMMEDIATE [26:0]</td></tr>
<tr><td>RRR</td><td colspan="5">OPCODE</td><td colspan="5">C</td><td colspan="5">A</td><td colspan="4">FN 1</td><td colspan="5">B</td><td colspan="8"></td></tr>
<tr><td>RRI</td><td colspan="5">OPCODE</td><td colspan="5">C</td><td colspan="5">A</td><td colspan="4">FN 1</td><td colspan="13">IMMEDIATE [12:0]</td></tr>
<tr><td>RRI2</td><td colspan="5">OPCODE</td><td colspan="5">IMM [12:8]</td><td colspan="5">A</td><td colspan="4">FN 1</td><td colspan="5">B</td><td colspan="8">IMMEDIATE [7:0]</td></tr>
<tr><td>RR</td><td colspan="5">OPCODE</td><td colspan="5">C</td><td colspan="5">A</td><td colspan="17"></td></tr>
<tr><td>N</td><td colspan="5">OPCODE</td><td colspan="27"></td></tr>
<tr><td>PB</td><td colspan="5">OPCODE</td><td colspan="4">FN 2</td><td>F1</td><td colspan="22"></td></tr>
</table>

BASE INSTRUCTIONS:

OPCODE	Instruction	Mnemonic	Definition	Func. 1	Format
0x08	System Call Interrupt	SYSCALL	Trigger an interrupt, enable kernel mode and disable interrupt requests	-	N
0x02	Load Upper Immediate	LUI	C = IMM << 10 //Lower bits are reset	-	RI
0x03	Store Byte	STB	M [A +- IMM][7:0] = B [7:0] //Signed immediate	0x0	
0x03	Store Word	STW	M [A +- IMM][15:0] = B [15:0] //Ignores ADDR[0], signed immediate	0x1	RRI2
0x03	Store Double-word	STD	M [A +- IMM][31:0] = B [31:0] //Ignores ADDR[1:0], signed immediate	0x2	
0x04	Load byte	LDB	C = M [A +- IMM][7:0] //Signed immediate	0x0	
0x04	Load word	LDW	C = M [A +- IMM][15:0] //Ignores ADDR[0], signed immediate	0x1	RRI
0x04	Load Double-word	LDD	C = M [A +- IMM][31:0] //Ignores ADDR[1:0], signed immediate	0x2	
0x05	Indirect Jump to [Register]	IRJ	PC = C + IMM //Signed Immediate, Ignores ADDR[2:0]	-	RI2
0x06	Jump	JMP	PC = PC +- IMM //Signed Immediate, Ignores ADDR[2:0]	-	I
0x07	Jump if equal	JEQ	If Z in A-B: PC = PC +- IMM //Signed Immediate, Ignores ADDR[2:0]	0x0	
0x07	Jump if lower than	JLT	If C in A-B: PC = PC +- IMM //Signed Immediate, Ignores ADDR[2:0]	0x1	
0x07	Jump if signed lower than	JSLT	If S in A-B: PC = PC +- IMM //Signed Immediate, Ignores ADDR[2:0]	0x2	
0x07	Jump if subtraction overflow	JSV	If V in A-B: PC = PC +- IMM //Signed Immediate, Ignores ADDR[2:0]	0x3	RRI2
0x07	Jump if not equal	JNE	If !Z in A-B: PC = PC +- IMM //Signed Immediate, Ignores ADDR[2:0]	0x4	
0x07	Jump if greater or equal	JGE	If !C in A-B: PC = PC +- IMM //Signed Immediate, Ignores ADDR[2:0]	0x5	
0x07	Jump if signed higher or equal	JSGE	If !S in A-B: PC = PC +- IMM //Signed Immediate, Ignores ADDR[2:0]	0x6	
0x07	Jump if not subtraction overflow	JNSV	If !V in A-B: PC = PC +- IMM //Signed Immediate, Ignores ADDR[2:0]	0x7	
0x00	Addition	ADD	C = A + B	0x0	
0x00	Subtraction	SUB	C = A - B	0x1	
0x00	Bitwise AND	AND	C = A & B	0x2	
0x00	Bitwise OR	OR	C = A | B	0x3	
0x00	Bitwise XOR	XOR	C = A ^ B	0x4	
0x00	Logical Right Shift	SHR	C = A >> B	0x5	RRR
0x00	Aritmethic Right Shift	ASR	C = A >>> B	0x6	
0x00	Logical Left Shift	SHL	C = A << B	0x7	
0x00	Addition Carry Check	CCH	C = Cout of A + B	0x8	
0x00	Subtraction Borrow Check	BCH	C = Borrow of A - B	0x9	
0x01	Addition (Immediate)	ADDI	C = A + IMM	0x0	
0x01	Subtraction (Immediate)	SUBI	C = A - IMM	0x1	
0x01	Bitwise AND (Immediate)	ANDI	C = A & IMM	0x2	
0x01	Bitwise OR (Immediate)	ORI	C = A | IMM	0x3	
0x01	Bitwise XOR (Immediate)	XORI	C = A ^ IMM	0x4	
0x01	Logical Right Shift (Immediate)	SHRI	C = A >> IMM	0x5	RRI
0x01	Aritmethic Right Shift (Immediate)	ASRI	C = A >>> IMM	0x6	
0x01	Logical Left Shift (Immediate)	SHLI	C = A << IMM	0x7	
0x01	Addition Carry Check (Immediate)	CCHI	C = Cout of (A + IMM)	0x8	
0x01	Subtraction Borrow Check (Immediate)	BCHI	C = Borrow of (A - IMM)	0x9	
0x0A	Set status bit	SSB	Status bit [FN 2] = 1	0x0	PB
0x0A	Clear status bit	CSB	Status bit [FN 2] = 0	0x1	PB
0x09	System Return	SYSRET	Return from an interrupt and re-enable interrupt requests	-	N

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	Equivalents	Definition
LDI (C), (IMM)	LUI (C), (IMM[31:10])	C = 32b IMM
	ADDI (C), (C), (IMM[9:0])	
INC (C)	ADDI (C), (C), 1	C = C + 1
DEC (C)	SUBI (C), (C), 1	C = C - 1
MOV (C), (A)	ADDI (C), zr, (A)	C = A
JAL (IMM)	ADDI rp, ip, 8	Jump and Link
	JMP (IMM)	
RET	IJR rp, 0	IP = RP
NOP	ADD zr, zr, zr	No Operation
CLR (C)	XOR (C), (C), (C)	C = 0
WFI	IJR ip, 0	Wait for an interrupt
PSH (B)	SUBI sp, sp, 4	Push a register to the stack
	STD (B), sp	
POP (C)	LDD (C), sp	Pop to a register from the stack
	ADDI sp, sp, 4	

64-BIT EXTENSION

This extension adds 64-bit variants for 2 opcodes and extends 2 instructions (giving 4 new instructions in total) 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)

OPCODE	Instruction	Mnemonic	Definition	Func. 1	Format
0x03	Store Quad-Word	STQ	[A + IMM] = B [63:0] //Ignores ADDR[3:0]	0x3	RRI2
0x04	Load Quad-Word (Only for the 64b ext.)	LDQ	C[63:0] = [A + IMM] //Ignores ADDR[3:0]	0x3	RRI
0x0b	Load Upper Immediate (Upper: Upper)	LUI.UU	C [63:42] = IMM	-	RI
0x0c	Move high immediate (Upper: Low)	LUI.LU	C [53:32] = IMM	-	RI

MUL/DIV EXTENSION

This extension implements signed multiplication, division and modulo. No new opcodes are implemented. The extension implements more operations to the ALU.

OPCODE	Instruction	Mnemonic	Definition	Func. 1	Format
0x00	Multiplication	MUL	C = A * B	0xB	RRR
0x00	Division	DIV	C = A / B	0XC	
0x00	Modulo / Remainder	MOD	C = A % B	0XD	
0x01	Multiplication (Immediate)	MULI	C = A * IMM	0xB	RRI
0x01	Division (Immediate)	DIVI	C = A / IMM	0xC	
0x01	Modulo / Remainder (Immediate)	MODI	C = A % IMM	0xD	

LEGAL

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