Slicudis RISC Machine

(SRM)

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 (not all of them can be written to), each of 32 bits. The sizes of the address bus and the data bus are 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 32 registers that are used for arithmetic operations, holding data, addresses, etc. These registers can be used for general purpose or for specific operations.

Address	Register	Function
0x00	ZR	Constant 0 (Read-Only)
0x01	G1	Caller (Conventional)
0x02	G2	Caller (Conventional)
0x03	G3	Caller (Conventional)
0x04	G4	Caller (Conventional)
0x05	G5	Caller (Conventional)
0x06	G6	Caller (Conventional)
0x07	G7	Callee (Conventional)
0x08	G8	Callee (Conventional)
0x09	G9	Callee (Conventional)
0x0A	G10	Callee (Conventional)
0x0B	G11	Callee (Conventional)
0x0C	G12	Callee (Conventional)
0x0D	G13	Global use
0x0E	G14	Global use
0x0F	G15	Global use
0x10	G16	Global use
0x11	G17	Global use
0x12	G18	Global use
0x13	G19	Global use
0x14	G20	Global use
0x15	G21	Global use
0x16	G22	Global use
0x17	G23	Global use
0x18	G24	Global use
0x19	G25	Global use
0x1A	G26	Global use
0x1B	SP	Stack Pointer (Conventional)
0x1C	SRD	Subroutine return dest. (Conventional)
0x1D	IRD	Interrupt return destination
0x1E	PC	Program Counter (Read-Only)
0x1F	SR	Status Register (Read-Only)

ZR: Always holds 0x0 and literally can't be modified.

G1-G6: General purpose registers that are used by the user.

G7-G12: General purpose registers that the kernel uses for subroutines.

G13-G24: Global general purpose registers

SP: Used in operations related to the stack

SDR: Used to hold the return address for subroutines

IRD: Used to hold the return address for interrupts

SR: Shows the contents of the status register

PC: Shows the contents of the P.C

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

- **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)
- **Kernel mode:** Is set by interrupts or the INT instruction and reset by the URT instruction

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
0xFFFFFFFC	$rst_vec	Reset vector	4 bytes
0xFFFFFFF8	$s_int	Software INT vector	4 bytes
0xFFFFFFF4	$h_int	Hardware INT vector	4 bytes

- **Reset vector:** Contains the starting point location of the P.C
- **Software INT vector:** Contains the subroutine location of system calls
- **Hardware INT vector:** Contains the subroutine location of hardware interrupts

In this example code, we're trying to make a SYSCAL subroutine to add G1 and G2 together and store the result in G7:

```
ORG 0xfffffffc
DB 0xff, 0xfd, 0x00, 0x00 //Reset location
DB 0xff, 0xfe, 0x00, 0x00 //Location of SYSCAL subroutines
DB 0xff, 0xff, 0x00, 0x00 //Location of the hardware interrupt subroutines

ORG 0xfffe0000 //Syscal subroutine
.Code:
ADD G7, G1, G2
URT

ORG 0xfffd0000 //Start
.Code:
ADDI G1, ZR, 5
ADDI G2, ZR, 14
SYSCAL
.End:
JMP .End
```

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 8 formats to allocate the parameters and operands.

FORMAT	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					C					IMMEDIATE [21:0]																					
I	OPCODE					IMMEDIATE [26:0]																											
RRR	OPCODE					C					A					FN 1			B													
RRI	OPCODE					C					A					FN 1			IMMEDIATE [12:0]													
RRI2	OPCODE					C					A					FN 1	IMMEDIATE [14:0]															
RR	OPCODE					C					A																					
N	OPCODE																															
PB	OPCODE					FN 2				F1																						

BASE INSTRUCTIONS:

OPCODE	Instruction	Mnemonic	Definition	Func. 1	Format
0x08	System Call Interrupt	SYSCAL	Trigger an interrupt and enable kernel mode	-	N
0x02	Move high immediate	MHI	C [31:10] = IMMEDIATE	-	RI
0x03	Store Double-word	STD	M [A + IMM (Signed)] = C [31:0] //Ignores ADDR[0]	0x2	
0x03	Store Word	STW	M [A + IMM (Signed)] = C [15:0] //Ignores ADDR[1:0]	0x1	RRI2
0x03	Store Byte	STB	M [A + IMM (Signed)] = C [7:0] //Ignores ADDR[2:0]	0x0	
0x04	Load Double-word	LDD	C[31:0] = M [A + IMM] //Ignores ADDR[0]	0x2	
0x04	Load word	LDW	C[15:0] = M [A + IMM] //Ignores ADDR[1:0]	0x1	RRI2
0x04	Load byte	LDB	C[7:0] = M [A + IMM] //Ignores ADDR[2:0]	0x0	
0x05	Indirect Jump	IJR	PC = C + IMM //Signed Immediate, Ignores ADDR[2:0]	-	RI
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	RRI
0x07	Jump if not equal	JNE	If !Z in A-B: PC = PC +- IMM //Signed Immediate, Ignores ADDR[2:0]	0x4	
0x07	Jump if higher or equal	JHE	If !C in A-B: PC = PC +- IMM //Signed Immediate, Ignores ADDR[2:0]	0x5	
0x07	Jump if signed higher or equal	JSHE	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	Aritmethic Right Shift	ASL	C = A <<< B	0x8	
0x00	Addition Carry Check	CCH	C = Cout of A + B	0x9	
0x00	Subtraction Borrow Check	BCH	C = Borrow of A - B	0xA	
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	Aritmethic Right Shift (Immediate)	ASLI	C = A <<< IMM	0x8	
0x01	Addition Carry Check (Immediate)	CCHI	C = Cout of (A + IMM)	0x9	
0x01	Subtraction Borrow Check (Immediate)	BCHI	C = Borrow of (A - IMM)	0xA	
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	User mode return	URT	Return from an interrupt in user mode	-	N

PSEUDO-INSTRUCTIONS

Pseudo-instructions are groups of instructions to make the equivalent of an instruction. These are some of the most used pseudoinstructions.

Pseudoinstructions	Equivalents	Definition
MVI (IMM)	MHI (C), (IMM[31:10])	C = 32b IMM
	ADDI (C), (DEST), (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)	MOV srd, pc	Jump and Link
	JMP (IMM)	
RET	IJR srd, 0	PC = SRD
NOP	ADD zr, zr, zr	No Operation

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 flag). 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] = C [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	RRI2
0x0b	Move high immediate (High: High)	MHI.HH	C [63:42] = IMM	-	RI
0x0c	Move high immediate (High: Low)	MHI.HL	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	
0x00	Division	DIV	C = A / B	0XC	RRR
0x00	Modulo / Remainder	MOD	C = A % B	0XD	
0x01	Multiplication (Immediate)	MULI	C = A * IMM	0xB	
0x01	Division (Immediate)	DIVI	C = A / IMM	0xC	RRI
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: