SCAT Instruction Set Architecture

Working draft 2022-09-22

CPU Registers

The CPU has 16 registers named R0 to R15, each 32 bits wide. Register R0 (aka `zero`) is hardwired with all bits equal to 0. All writes to R0 are discarded. Register R15 (aka `PC`) is the *Program Counter*, i.e. it holds the address of the current instruction. Other registers (R1 to R14) are general purpose, i.e. they are freely available for software. Typically though, register R13 is used as the *Stack Pointer* `SP` and R14 as the *Link Register* `LR`.

Instruction Format

All instructions are 32 bits in length, and follow a similar format as illustrated below. Bits bits 28 to 31 are the *Type* field ; they indicate how the rest of the instruction bits should be interpreted. Bits 24 to 27 are specific to each instruction type: opcode for ALU instructions, comparison code for conditional branches, etc. The next fields are the *destination* (rd) and *source* (rs1, rs2) registers. Most instruction types use just one source register. In that case, bits 0 to 15 encode a 16-bits *immediate value* that serves as an additional operand to the instruction.

	31 28	27 24	23 20	19 16	15 12	11 0
ALU reg-reg	type 1	op	rd	rs1	rs2	ignored
ALU reg-imm	type 2	op	rd	rs1	imm	
Cond. jump	type 3	comp	rd	rs1	imm	
Memory	type 4	trans	rd	rs1	imm	
Subroutines	type 5	ignored	rd	rs1	imm	

Type 1: Arithmetic and Logic Register-Register Operations

asm	Name	opcode	Description	Notes
add	Addition	0000	rd = rs1 + rs2	
sub	Subtraction	0001	rd = rs1 - rs2	
mul	Multiplication	0010	rd = rs1 * rs2	
div	Integer Division	0011	rd = rs1 / rs2	quotient of floored division
mod	Modulo	0100	rd = rs1 % rs2	remainder of floored division
or	Bitwise OR	0101	rd = rs1 | rs2	
and	Bitwise AND	0110	rd = rs1 & rs2	
xor	Bitwise exclusive OR	0111	rd = rs1 ^ rs2	
lsl	Logical Shift Left	1000	rd = rs1 << rs2	
lsr	Logical Shift Right	1001	rd = rs1 >> rs2	
asr	Arithmetic Shift Right	1010	rd = rs1 >> rs2	rs1 as a signed integer
slt	Set Less Than	1011	rd = (rs1 < rs2)?1:0	rs1/rs2 as signed integers
sltu	Set Less Than, Unsigned	1100	rd = (rs1 < rs2)?1:0	rs1/rs2 as unsigned integers

Notes:
- In shift instructions, `rs2` is always interpreted as a positive (unsigned) number. However, shifts only make sense when $0 < rs2 < 32$.
- Multiplication discards the upper bits of the result.
- The `div` and `mod` instructions implement *floored division* semantics: the mathematical result is always rounded towards negative infinity. For example $8 \div -3$ gives -3 with a remainder of -1.

Type 2: Arithmetic and Logic Register-Immediate Operations

These instructions are similar to type 1 but the second operand is a sign-extended 16-bits immediate value.

asm	Name	op code	Description
addi	Addition	0000	rd = rs1 + sxt(imm)
subi	Subtraction	0001	rd = rs1 - sxt(imm)
muli	Multiplication	0010	rd = rs1 * sxt(imm)
divi	Integer Division	0011	rd = rs1 / sxt(imm)
modi	Modulo	0100	rd = rs1 % sxt(imm)
ori	Bitwise OR	0101	rd = rs1 | sxt(imm)
andi	Bitwise AND	0110	rd = rs1 & sxt(imm)
xori	Bitwise exclusive OR	0111	rd = rs1 ^ sxt(imm)
lsli	Logical Shift Left	1000	rd = rs1 << imm
lsri	Logical Shift Right	1001	rd = rs1 >> imm
asri	Arithmetic Shift Right	1010	rd = rs1 >> imm
slti	Set Less Than	1011	rd = (rs1 < sxt(imm))?1:0
sltiu	Set Less Than, Unsigned	1100	rd = (rs1 < imm)?1:0

Note: Shift instructions only make sense with $0 < $ `imm` $ < 32$.

Type 3: Conditional Jumps aka Compare-And-Branch

These instructions compare the values of two registers and take a jump (i.e. change the value of `PC`) when the condition is verified.

asm	Name	comp code	Description
beq	Branch if equal	0000	if (rd == rs1) PC += sxt(imm)
bne	Branch if not equal	0001	if (rd != rs1) PC += sxt(imm)
blt	Branch if lower than	0010	if (rd < rs1) PC += sxt(imm)
bge	Branch if greater or equal	0011	if (rd >= rs1) PC += sxt(imm)
bltu	Branch if lower than, Unsigned	0100	if (rd < rs1) PC += sxt(imm)
bgeu	Branch if greater or equal, Unsigned	0101	if (rd >= rs1) PC += sxt(imm)

Note: `blt`/`bge` perform a signed comparison, whereas `bltu`/`bgeu` interpret both operands as unsigned numbers.

Type 4: Memory transfers

asm	Name	trans code	Description
load rd, [rs1 + offset]	Load Word	0000	rd = MEM[rs1 + sxt(imm)]
store [rd + offset], rs	Store Word	0001	MEM[rd + sxt(imm)] = rs1

Type 5: Subroutine calls

asm	Name	trans code	Description
jal	Jump-And-Link	0000	rd = PC+4 ; PC = rs1 + sxt(imm)

Typically, `jal` is used with R14 (aka `LR`) as destination register. Returning from subroutine can be achieved with `jal R0, LR`.

Pseudo Instructions

asm	Base Instruction(s)	Description
nop	addi r0, r0, 0	Preferred encoding for NOP
mov rd, rs	addi rd, rs, 0	Copy a value from one register to another
leti rd, imm	addi rd, zero, imm	Assign constant (see below for large values)
not rd, rs	xori rd, rs, -1	One's complement aka bitwise NOT
neg rd, rs	sub rd, zero, rs	Two's complement aka sign change
seqz rd, rs	sltiu rd, rs, 0	Set rd if $rs = 0$
snez rd, rs	slt rd, zero, rs	Set rd if $rs \neq 0$
sltz rd, rs	slt rd, rs, zero	Set rd if $rs < 0$
sgtz rd, rs	slt rd, zero, rs	Set rd if $rs > 0$
bra offset	addi R15, R15, offset	Branch always
beqz r, offset	beq r, zero, offset	Branch if $r = 0$
bnez r, offset	bne r, zero, offset	Branch if $r \neq 0$
blez r, offset	bge zero, r, offset	Branch if $r \leqslant 0$
bgez r, offset	bge r, zero, offset	Branch if $r \geqslant 0$
bltz r, offset	blt r, zero, offset	Branch if $r < 0$
bgtz r, offset	blt zero, r, offset	Branch if $r > 0$
bgt r1, r2, offset	blt r2, r1, offset	Branch if $r1 > r2$
ble r1, r2, offset	bge r2, r1, offset	Branch if $r1 \leqslant r2$
bgtu r1, r2, offset	bltu r2, r1, offset	Branch if $r1 > r2$, both unsigned
bleu r1, r2, offset	bgeu r2, r1, offset	Branch if $r1 \leqslant r2$, both unsigned
push reg	subi r13, r13, 4	
	store [r13], reg	Push onto stack (r13 is our stack pointer)
pop reg	load reg, [r13]	
	addi r13, r13, 4	Pop from stack
jmp offset	jal zero, R15, offset	Relative jump, return address discarded
call offset	jal R14, R15, offset	Relative jump, return address saved to LR
ret	jal zero, R14	Return from subroutine

Immediates above 16 bits

The `leti` pseudo-instruction assigns a literal value to a register. When the number is between -32768 and 32767 (both included) it will simply be encoded in 16-bits two's complement, and used as the immediate operand in an `addi` instruction.

For larger values, the assembler will have to generate several instructions in a row. For example, writing `leti r1, 0x12345678` will produce three instructions:

```
addi r1, zero, 0x1234
lsli r1, r1, 16 ; shift into upper half
addi r1, r1, 0x5678
```

This is often enough, but because `addi` always performs a sign-extension, some (rare) values are trickier to compute correctly. For example `leti r1, 0x43218765` translates into no less than five instructions.

An alternative (manual) approach is to have the value somewhere as a named `.word` and load it through PC-indirect addressing, as illustrated below. Note: be sure to choose a "safe" place (e.g. just after an unconditional jump) where the CPU will not attempt to execute a value as an instruction.

```
        load r1, [myvalue] ; assembled as [r15+8]
        jmp +0
myvalue: .word 0x43218765
```