What are the sliderules?

The sliderules are information-dense spreadsheets about the RISC-V instruction set (RV32IMAC for now).
They have been designed as didactic tools with one extremely specific usecase in mind.
Enabling a student to manually:
- quiclky encode assembly into binary machine language **(a2b "Encoder" Sliderule)**
- quickly decode binary machine language into assembly **(b2a "Decoder" Sliderule)**

How to read the a2b "Encoder" Sliderule?

The a2b "Encoder" Sliderule quickly retrieves the binary form of any assembly instruction at hand.

The Sliderule is read starting from **column K**, where all the instructions and pseudoinstructions can be found in alphabetical order.
- On the immediate right is the assembly syntax rule.
- On the left is some description and classification of the instruction.
- On the far right there is the actual encoding bit by bit, with immediate encoding specified under the instruction encoding.

How to read the b2a "Decoder" Sliderule?

The b2a "Decoder" Sliderule quickly retrieves the exact assembly instruction (or just its type and format) from any valid machine instruction in binary form.

The binary is intended to be read from lsb to msb and the Sliderule is read starting from **column AF** (corresponding to the lsb of the instruction).
Here is what can be retreived:

OPTION 1 - Retrieve the instruction type and format from the "Simple View"
This is done in the first section of the sheet called "Simple View" and uses only the **opcode** of the instructions.
The instructions are ordered with a precise rationale: you will find all instructions with lsb equal to 0 (column AF) and below all the instructions with lsb==1.
Within each block, you will find first all instruction with the second bit equal to 0 (next column on the left, column AE) and below all instructions with the second bit equal to 1.
So on for all the opcode bits.
Once the matching opcode is found, horizontally the following information can be seen:
- on the left, the structure of the instruction type and immediate encoding
- on the right, extension, quadrant, type and description of the opcode

ATTENTION: for the C extension, the ordering of the instructions follows the first two bits and the last three instead of the bits in order:
that is instead of checking columns sequentially to the left (AF, AE, AD, AC...) the order should be AF, AE, S, R, Q.

OPTION 2 - Retrieve the exact instruction from the "Exploded View"
This is done in the second section of the sheet called "Exploded View" and uses **all the bits** of the instruction.
The instructions are ordered with a precise rationale: you will find all instructions with lsb equal to 0 (column AF) and below all the instructions with lsb==1.
Within each block, you will find first all instruction with the second bit equal to 0 (next column on the left, column AE) and below all instructions with the second bit equal to 1.
So on for all the bits within the limit of the structure of the instructions. This is so to reach the correct approximate area of the sheet quickly for any given researched instruction.
Once the matching instruction is found, horizontally the following information can be seen:
- on the left, the structure of the instruction type and immediate encoding
- on the right, type, assembly format and description of the instruction

ATTENTION: for the C extension, the ordering of the instructions follows the first two bits and the last three instead of the bits in order:
that is instead of checking columns sequentially to the left (AF, AE, AD, AC...) the order should be AF, AE, S, R, Q.

How can one contribute?

OPTION 1 - Directly modify
Just make a modification or fix an existing error (this requires write access, see option 3),
update the version number (top right of each sliderule)
and feel free to add yourself to the contributors list (left of version number).

OPTION 2 - Report an Error
Just use the "Error Reporting" sheet to report an error, I will fix it as soon as possible. (this requires write access, see option 3)

OPTION 3 - Email
For anything else (write access request included), just drop me an email at magi.wanders@gmail.com (Simone Shawn Cazzaniga)

What is the future vision?

Doing things in Google Sheets is rather cumbersome and time consuming, even if with big advantages.
The future of this project, aside than containing the full RISC-V instruction set is to:
- Become version controllable (git/github)
- Have a unified encoded database from which to render directly in latex all the possibly needed tables and sliderules.

Please message me at magi.wanders@gmail.com if you have the knowledge to work on such things or even just for advice on how to do it.

5-bit Encoding (rx)	3-bit Compressed Encoding (rx')	Register Name	ABI Name	Description	Saved by calle-
0	-	x0	0	hardwired zero	-
1	-	x1	ra	return address	-R
2	-	x2	sp	stack pointer	-E
3	-	x3	gp	global pointer	-
4	-	x4	tp	thread pointer	-
5	-	x5	t0	temporary register 0	-R
6	-	x6	t1	temporary register 1	-R
7	-	x7	t2	temporary register 2	-R
8	0	x8	s0 / fp	saved register 0 / frame pointer	-E
9	1	x9	s1	saved register 1	-E
10	2	x10	a0	function argument 0 / return value 0	-R
11	3	x11	a1	function argument 1 / return value 1	-R
12	4	x12	a2	function argument 2	-R
13	5	x13	a3	function argument 3	-R
14	6	x14	a4	function argument 4	-R
15	7	x15	a5	function argument 5	-R
16	-	x16	a6	function argument 6	-R
17	-	x17	a7	function argument 7	-R
18	-	x18	s2	saved register 2	-E
19	-	x19	s3	saved register 3	-E
20	-	x20	s4	saved register 4	-E
21	-	x21	s5	saved register 5	-E
22	-	x22	s6	saved register 6	-E
23	-	x23	s7	saved register 7	-E
24	-	x24	s8	saved register 8	-E
25	-	x25	s9	saved register 9	-E
26	-	x26	s10	saved register 10	-E
27	-	x27	s11	saved register 11	-E
28	-	x28	t3	temporary register 3	-R
29	-	x29	t4	temporary register 4	-R
30	-	x30	t5	temporary register 5	-R
31	-	x31	t6	temporary register 6	-R

RV32IMAC
Instruction Set "Sliderule" Encoder

Designed to be printed on A3

Contributors: Simone Shawn Cazzaniga (magiwanders@github | www.magiwanders.com), v0.0.3

Extension	Type	Description	Explanation	Assembly			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
							instuction / opcode
I	R	Add	rd = rs1 + rs2	add	rd, rs1, rs2		- 0 - - - - - rs2 rs1 0 0 0 rd 0 1 1 0 0 1 1
I	I	Add Immediate	rd = rs + imm	addi	rd, rs,	imm	m1 imm1 rs 0 0 0 rd 0 0 1 0 0 1 1 / - m1 - imm1
A	R	Atomic Add	rd <- M[rs1] + rs2 ; rd -> M[rs1]	amoadd.w	rd, rs1, rs2		0 0 0 0 0 aq rl rs2 rs1 0 1 0 rd 0 1 0 1 1 1 1
A	R	Atomic And	rd <- M[rs1] & rs2 ; rd -> M[rs1]	amoand.w	rd, rs1, rs2		0 1 1 0 0 aq rl rs2 rs1 0 1 0 rd 0 1 0 1 1 1 1
A	R	Atomic Max	rd <- max(M[rs1], rs2) ; rd -> M[rs1]	amomax.w	rd, rs1, rs2		1 0 1 0 0 aq rl rs2 rs1 0 1 0 rd 0 1 0 1 1 1 1
A	R	Atomic Min	rd <- min(M[rs1], rs2) ; rd -> M[rs1]	amomin.w	rd, rs1, rs2		1 0 0 0 0 aq rl rs2 rs1 0 1 0 rd 0 1 0 1 1 1 1
A	R	Atomic Or	rd <- M[rs1] | rs2 ; rd -> M[rs1]	amoor.w	rd, rs1, rs2		0 1 0 1 0 aq rl rs2 rs1 0 1 0 rd 0 1 0 1 1 1 1
A	R	Atomic Swap	rd <- M[rs1] ; swap(rd, rs2) ; rd -> M[rs1]	amoswap.w	rd, rs1, rs2		0 0 0 0 1 aq rl rs2 rs1 0 1 0 rd 0 1 0 1 1 1 1
A	R	Atomic Xor	rd <- M[rs1] ^ rs2 ; rd -> M[rs1]	amoxor.w	rd, rs1, rs2		0 0 1 0 0 aq rl rs2 rs1 0 1 0 rd 0 1 0 1 1 1 1
I	R	And	rd = rs1 & rs2	and	rd, rs1, rs2		- 0 - - - - - rs2 rs1 1 1 1 rd 0 1 1 0 0 1 1
I	I	And Immediate	rd = rs1 & imm	andi	rd, rs1	imm	m1 imm1 rs1 1 1 1 rd 0 0 1 0 0 1 1 / - m1 - imm1
I	U	Add Upper Immediate to PC	rd = PC + imm << 12	auipc	rd,	imm	imm1 rd 0 0 1 0 1 1 1 / imm1 - 0 -
I	B	Branch if Equal	if rs1 == rs2: PC+=imm	beq	rs1, rs2	imm	m2 imm2 rs2 rs1 0 0 0 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
PSEUDO	B	Branch if Zero	beq rs, x0, imm	beqz	rs,	imm	m2 imm2 0 0 0 0 0 rs 0 0 0 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
I	B	Branch if Greater or Equal	if (rs1 >= rs2) ^ (rs1[31] != rs2[31]): PC+=imm	bge	rs1, rs2	imm	m2 imm2 rs2 rs1 1 0 1 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
I	B	Branch if Unsigned is Greater or Equal	if rs1 >= rs2: PC+=imm	bgeu	rs1, rs2	imm	m2 imm2 rs2 rs1 1 1 1 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
PSEUDO	B	Branch if Zero or Greater	bge rs, x0, imm	bgez	rs,	imm	m2 imm2 0 0 0 0 0 rs1 1 0 1 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
PSEUDO	B	Branch if Greater	blt rs2, rs1, imm	bgt	rs1, rs2,	imm	m2 imm2 rs1 rs2 1 0 0 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
PSEUDO	B	Branch if Unsigned is Greater	bltu rs2, rs1, imm	bgtu	rs1, rs2,	imm	m2 imm2 rs1 rs2 1 1 0 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
PSEUDO	B	Branch if Greater Than Zero	blt x0, rs, imm	bgtz	rs,	imm	m2 imm2 0 0 0 0 0 rs 1 0 0 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
PSEUDO	B	Branch if Less or Equal	bge rs2, rs1, imm	ble	rs1, rs2,	imm	m2 imm2 rs1 rs2 1 0 1 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
PSEUDO	B	Branch if Unsigned is Less or Equal	bgeu rs2, rs1, imm	bleu	rs1, rs2,	imm	m2 imm2 rs1 rs2 1 1 1 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
PSEUDO	B	Branch if Zero or Less	bge x0, rs, imm	blez	rs,	imm	m2 imm2 rs 0 0 0 0 0 1 0 1 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
I	B	Branch if Less Than	if (rs1 < rs2) ^ (rs1[31] != rs2[31]): PC+=imm	blt	rs1, rs2	imm	m2 imm2 rs2 rs1 1 0 0 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
PSEUDO	B	Branch if Less Than Zero	blt rs, x0, imm	bltz	rs,	imm	m2 imm2 0 0 0 0 0 rs 1 0 0 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
I	B	Branch if Unsigned is Less Than	if rs1 < rs2: PC+=imm	bltu	rs1, rs2	imm	m2 imm2 rs2 rs1 1 1 0 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
I	B	Branch if Not Equal	if rs1 != rs2: PC+=imm	bne	rs1, rs2	imm	m2 imm2 rs2 rs1 0 0 1 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
PSEUDO	B	Branch if Not Zero	bne rs, x0, imm	bnez	rs,	imm	m2 imm2 0 0 0 0 0 rs 0 0 1 imm1 m1 1 1 0 0 0 1 1 / - m2 - m1 imm2 imm1 0
C	CR	Compressed Add	add rd, rd, rs (rd += rs)	c.add	rd, rs		1 0 0 1 rd != 0 rs != 0 1 0
C	CI	Compressed Add Immediate	addi rd, rd, nzimm (rd += nzimm)	c.addi	rd	(sign extend) imm = nzimm	0 0 0 m1 rd != 0 imm1 0 1 / - m1 - imm m1 imm1
C	CIW	Compressed Add imm*4 to SP	addi rd', x2, 4*imm (rd' = sp + nzuimm = sp + 4*imm)	c.addi4spn	rd'	4 * imm = nzuimm	0 0 0 imm2 imm1 m2 m1 rd' 0 0 / - 0 - imm imm1 imm2 m1 m2 - 0 -
C	CA	Compressed And	and rd', rd', rs2' (rd' &= rs2')	c.and	rd,rs1		1 0 0 0 1 1 rd' 1 1 rs2' 0 1
C	CB	Compressed And Immediate	andi rd', rd', imm (rd &= imm)	c.andi	rd	imm	1 0 0 m1 1 0 rd' imm1 0 1 / - m1 - imm1
C	CB	Comrpessed Branch if EQual to Zero	beq rs1', x0, offset (if (rs1' == x0) PC += offset)	c.beqz	rs1'	(msb extended) imm << 1 = offset	1 1 0 m1 imm3 rs1' imm2 imm1 m2 0 1 / - m1 - imm m1 imm2 m2 imm3 imm1 0
C	CB	Compressed Branch if Not Equal to Zero	bne rs1', x0, offset (if (rs1' != x0) PC += offset)	c.bnez	rs1'	(msb extended) imm << 1 = offset	1 1 1 m1 imm3 rs1' imm2 imm1 m2 0 1 / - m1 - imm m1 imm2 m2 imm3 imm1 0

RV32IMAC

Instruction Set "Sliderule" Encoder

Designed to be printed on A3

Contributors: Simone Shawn Cazzaniga (magiwanders@github | www.magiwanders.com),

v0.0.3

Instruction bit fields are listed from bit 31 (left) to bit 0 (right), separated by " · ". Bits 1–0 are the opcode's lowest bits. For compressed instructions, bits 31–16 are shown as a black (unused) area and only bits 15–0 are listed.

Extension	Type	Description	Explanation			Assembly			Instruction (31 … 0)
C	CR	Compressed Environment Break	Transfer Control back to Debug Environment			c.ebreak			1 · 0 · 0 · 1 · - 0 - · - 0 - · 1 · 0
C	CJ	Compressed Jump	jal	x0, offset	PC += offset	c.j			1 · 0 · 1 · m1 · m2 · imm2 · m3 · m4 · m5 · imm1 · m6 · 0 · 1
							offset		- m1 - · m3 · imm2 · m5 · m4 · m6 · m2 · m2 · 0
C	CJ	Compressed Jump and Link	jal	x1, offset	rd = PC + 2 ; PC += offset	c.jal			0 · 0 · 1 · m1 · m2 · imm2 · m3 · m4 · m5 · imm1 · m6 · 0 · 1
							offset		- m1 - · m3 · imm2 · m5 · m4 · m6 · m2 · imm1 · 0
C	CR	Compressed Jump and Link Register	jalr	x1, 0(rs1)	ra = PC + 2; PC = rs1	c.jalr	rs1		1 · 0 · 0 · 1 · rs1 != 0 · - 0 - · 1 · 0
C	CR	Compressed Jump Register	jalr	x0, 0(rs1)	PC = rs1	c.jr	rd,rs1		1 · 0 · 0 · 0 · rs1 != 0 · - 0 - · 1 · 0
C	CI	Compressed Load Immediate	addi	rd, x0, imm	rd = imm	c.li	rd, imm		0 · 1 · 0 · m1 · rd != 0 · imm1 · 0 · 1
								imm	- m1 - · imm1
C	CI	Compressed Load Upper Immediate	lui	rd, nzimm	rd = imm << 12 (zero extends)	c.lui	rd, imm		0 · 1 · 1 · m1 · rd != 0, 2 · imm1 · 0 · 1
								(msb extend) imm<<12 = nzimm	- m1 - · imm (m1 · imm1) · - 0 -
C	CL	Load Word	lw	rd, offset(rs1')	rd' <- M[rs1' + offset]	c.lw	rd'	(rs1')	0 · 1 · 0 · imm · rs1' · m2 · m1 · rd' · 0 · 0
							4 * imm = offset		- 0 - · imm (m1 · imm · m2) · - 0 -
C	CI	Compressed Load Word from SP	lw	rd, offset(x2)	rd <- M[sp + offset]	c.lwsp	rd,		0 · 1 · 0 · m1 · rd · imm2 · imm1 · 1 · 0
								4 * imm = offset	- 0 - · imm (imm1 · m1 · imm2) · - 0 -
C	CR	Compressed Move	add	rd, x0, rs	rd = rs	c.mv	rd,rs		1 · 0 · 0 · 0 · rd != 0 · rs != 0 · 1 · 0
C	CI	Compressed No OPeration	Literally do nothing.			c.nop			0 · 0 · 0 · 0 · - 0 - · - 0 - · 0 · 1
C	CA	Compressed Or	or	rd', rd', rs2'	rd' |= rs2'	c.or	rd,rs1		1 · 0 · 0 · 0 · 1 · 1 · rd' · 1 · 0 · rs2' · 0 · 1
C	CI	Compressed Shift Left Logical Immediate	slli	rd, rd, shamt	rd = rd <<0 shamt	c.slli	rd,		0 · 0 · 0 · 0 · rd != 0 · imm · 1 · 0
								shamt	- 0 - · 0 · imm
C	CB	Compressed Shift Right Arithmetic Immediate	srai	rd', rd'. shamt	rd = rd (msb)>> shamt	c.srai	rd		1 · 0 · 0 · 0 · 0 · 1 · rd' · imm · 0 · 1
								shamt	- 0 - · 0 · imm
C	CB	Compressed Shift Right Logical Immediate	srli	rd', rd'. shamt	rd = rd (zero)>> shamt	c.srli	rd		1 · 0 · 0 · 0 · 0 · 0 · rd' · imm · 0 · 1
								shamt	- 0 - · 0 · imm
C	CA	Compressed Subtract	sub	rd', rd', rs2'	rd' -= rs2'	c.sub	rd,rs1		1 · 0 · 0 · 0 · 1 · 1 · rd' · 0 · 0 · rs2' · 0 · 1
C	CSS	Compressed Store Word	sw	rs1', offset(rs2')	rs2' -> M[rs1' + offset]	c.sw	rs1',	(rs2')	1 · 1 · 0 · imm1 · rs1' · m2 · m1 · rs2' · 0 · 0
							4 * imm = offset		- 0 - · imm (m1 · imm1 · m2) · - 0 -
C	CSS	Compressed Store Word to SP	sw	rd, offset(x2)	rs2' -> M[sp + offset]	c.swsp	rd,		1 · 1 · 0 · imm2 · imm1 · rd · 1 · 0
								4 * imm = offset	- 0 - · imm (imm1 · imm2) · - 0 -
C	CA	Compressed Xor	xor	rd', rd', rs2'	rd' ^= rs2'	c.xor	rd,rs1		1 · 0 · 0 · 0 · 1 · 1 · rd' · 0 · 1 · rs2' · 0 · 1
PSEUDO (multiple)	/	Call Far-Away Subroutine	auipc	x1, imm[31:12]		call	imm		imm[31:12] · 0 · 0 · 0 · 0 · 1 · 0 · 0 · 1 · 0 · 1 · 1 · 1
			jalr	x1, x1, imm[11:0]					imm[11:0] · 0 · 0 · 0 · 0 · 1 · 0 · 0 · 0 · 0 · 0 · 0 · 0 · 1 · 1 · 1 · 0 · 0 · 1 · 1 · 1
M	R	Divide	rd = rs1 / rs2			div	rd, rs1, rs2		- · - · - · - · - · - · 1 · rs2 · rs1 · 1 · 0 · 0 · rd · 0 · 1 · 1 · 0 · 0 · 1 · 1
M	R	Divide Unsigned	rd = rs1 / (unsigned) rs2			divu	rd, rs1, rs2		- · - · - · - · - · - · 1 · rs2 · rs1 · 1 · 0 · 1 · rd · 0 · 1 · 1 · 0 · 0 · 1 · 1
I	I	Environment Break	Transfer Control back to Debug Environment			ebreak			- · - · - · - · - · - · - · - · - · - · - · 1 · rs1 · 0 · 0 · 0 · rd · 1 · 1 · 1 · 0 · 0 · 1 · 1
I	I	Environment Call	Transfer Control to Execution Environment			ecall			- · - · - · - · - · - · - · - · - · - · - · 0 · rs1 · 0 · 0 · 0 · rd · 1 · 1 · 1 · 0 · 0 · 1 · 1
PSEUDO	J	Jump	jal	x0, imm		j			m2 · imm2 · m1 · imm1 · 0 · 0 · 0 · 0 · 0 · 1 · 1 · 0 · 1 · 1 · 1 · 1
							imm		- m2 - · imm1 · m1 · imm2 · 0
I	J	Jump and Link	rd = PC + 4 ; PC += imm			jal	rd,		m2 · imm2 · m1 · imm1 · rd · 1 · 1 · 0 · 1 · 1 · 1 · 1
								imm	- m2 - · imm1 · m1 · imm2 · 0
PSEUDO	J	Jump and Link	jal	x1, imm		jal			m2 · imm2 · m1 · imm1 · 0 · 0 · 0 · 0 · 1 · 1 · 1 · 0 · 1 · 1 · 1 · 1
							imm		- m2 - · imm1 · m1 · imm2 · 0
I	I	Jump and Link Register	rd = PC + 4 ; PC = rs1 + imm			jalr	rd, rs1,		m · imm1 · rs1 · 0 · 0 · 0 · rd · 1 · 1 · 0 · 0 · 1 · 1 · 1
								imm	- m - · imm1
PSEUDO	I	Jump and Link Register	jalr	x1, rs, 0		jalr	rs		- 0 - · rs · 0 · 0 · 0 · 0 · 0 · 0 · 0 · 1 · 1 · 1 · 0 · 0 · 1 · 1 · 1
PSEUDO	I	Jump Register	jalr	x0, rs, 0		jr	rs		- 0 - · rs · 0 · 0 · 0 · 0 · 0 · 0 · 0 · 0 · 1 · 1 · 0 · 0 · 1 · 1 · 1
PSEUDO (multiple)	/	Load Address	auipc	rd, imm[31:12]		la	rd, imm		imm[31:12] · rd · 0 · 0 · 1 · 0 · 1 · 1 · 1
			addi	rd, rd, imm[11:0]					imm[11:0] · rd · 0 · 0 · 0 · rd · 0 · 0 · 1 · 0 · 0 · 1 · 1
I	I	Load Byte	rd[7:0] <- M[rs+imm][7:0]			lb	rd,	(rs)	m1 · imm1 · rs1 · 0 · 0 · 0 · rd · 0 · 0 · 0 · 0 · 0 · 1 · 1
							imm		- m1 - · imm1
PSEUDO (multiple)	/	Load Global Byte (imm more than 12bit)	auipc	rd, imm[31:12]		lb	rd, imm		imm[31:12] · rd · 0 · 0 · 1 · 0 · 1 · 1 · 1
			lb	rd, imm[11:0](rd)					imm[11:0] · rs1 · 0 · 0 · 0 · rd · 0 · 0 · 0 · 0 · 0 · 1 · 1
I	I	Load Unsigned Byte	rd[7:0] <- M[rs+uimm][7:0]			lbu	rd,	(rs)	m1 · imm1 · rs1 · 1 · 0 · 0 · rd · 0 · 0 · 0 · 0 · 0 · 1 · 1
							imm		- m1 - · imm1
I	I	Load Half Word	rd[15:0] <- M[rs+imm][15:0]			lh	rd,	(rs)	m1 · imm1 · rs1 · 0 · 0 · 1 · rd · 0 · 0 · 0 · 0 · 0 · 1 · 1
							imm		- m1 - · imm1
PSEUDO (multiple)	/	Load Global Half Word (imm more than 12bit)	auipc	rd, imm[31:12]		lh	rd, imm		imm[31:12] · rd · 0 · 0 · 1 · 0 · 1 · 1 · 1
			lh	rd, imm[11:0](rd)					imm[11:0] · rs1 · 0 · 0 · 1 · rd · 0 · 0 · 0 · 0 · 0 · 1 · 1

RV32IMAC
Instruction Set "Sliderule" Encoder

Designed to be printed on A3

Contributors: Simone Shawn Cazzaniga (magiwanders@github | www.magiwanders.com), v0.0.3

Extension	Type	Description	Explanation	Assembly	instuction 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	opcode 6 5 4 3 2 1 0
I	I	Load Unsigned Half Word	rd[15:0] <- M[rs+uimm][15:0]	lhu rd, imm (rs)	m1 imm1 rs1 1 0 1 rd	0 0 0 0 0 1 1
					- m1 - imm1	
A	R	Load Reserved	rd <- M[rs1] ; Reserve M[rs1]	lr.w rd, rs1	0 0 0 1 0 aq rl rs2 rs1 0 1 0 rd	0 1 0 1 1 1 1
I	U	Load Upper Immediate	rd = imm << 12	lui rd, imm	imm1 rd	0 1 1 0 1 1 1
					imm1 - 0 -	
I	I	Load Word	rd[31:0] <- M[rs+imm][31:0]	lw rd, imm (rs)	m1 imm1 rs 0 1 0 rd	0 0 0 0 0 1 1
					- m1 - imm1	
PSEUDO (multiple)	/	Load Global Word (imm more than 12bit)	auipc rd, imm[31:12]	lw rd, imm	imm[31:12] rd	0 0 1 0 1 1 1
			lw rd, imm[11:0](rd)		imm[11:0] rs 0 1 0 rd	0 0 0 0 0 1 1
I	R	Or	rd = rs1 | rs2	or rd, rs1, rs2	- 0 - - - - - - rs2 rs1 1 1 0 rd	0 1 1 0 0 1 1
I	I	Or Immediate	rd = rs1 | imm	ori rd, rs1 imm	m1 imm1 rs1 1 1 0 rd	0 0 1 0 0 1 1
					- m1 - imm1	
M	R	Multiply Signed x Signed	rd = (signed)rs1 * (signed)rs2	mul rd, rs1, rs2	- - - - - - 1 rs2 rs1 0 0 0 rd	0 1 1 0 0 1 1
M	R	Multiply High Signed x Signed	rd = { (signed)rs1 * (signed)rs2 }[63:32]	mulh rd, rs1, rs2	- - - - - - 1 rs2 rs1 0 0 1 rd	0 1 1 0 0 1 1
M	R	Multiply High Signed x Unsigned	rd = (signed)rs1 * (unsigned)rs2	mulhsu rd, rs1, rs2	- - - - - - 1 rs2 rs1 0 1 0 rd	0 1 1 0 0 1 1
M	R	Multiply High Unsigned x Unsigned	rd = rs1 % rs2	mulhu rd, rs1, rs2	- - - - - - 1 rs2 rs1 0 1 1 rd	0 1 1 0 0 1 1
PSEUDO	I	Move Register	addi rd, rs, 0	mv rd, rs	- 0 - rs 0 0 0 rd	0 0 1 0 0 1 1
PSEUDO	R	2's Complement	sub rd, x0, rs	neg rd, rs	- 1 - - - - - rs - 0 - 0 0 0 rd	0 1 1 0 0 1 1
PSEUDO	I	No OPeration	addi x0, x0, 0	nop	- 0 - - 0 - 0 0 0 - 0 -	0 0 1 0 0 1 1
PSEUDO	I	1's Complement	xori rd, rs, -1	not rd, rs	- 1 - rs 1 0 0 rd	0 0 1 0 0 1 1
M	R	Remainder	rd = rs1 % rs2	rem rd, rs1, rs2	- - - - - - 1 rs2 rs1 1 1 0 rd	0 1 1 0 0 1 1
M	R	Reminder Unsigned	rd = rs1 % (unsigned) rs2	remu rd, rs1, rs2	- - - - - - 1 rs2 rs1 1 1 1 rd	0 1 1 0 0 1 1
PSEUDO	I	Return from Subroutine	jalr x0, x1, 0	ret	- 0 - 0 0 0 0 1 0 0 0 0 0 0 0 0 0	1 1 0 0 1 1 1
I	S	Store Byte	rs2[7:0] -> M[rs1+imm][7:0]	sb rs1, imm (rs2)	m2 imm2 rs2 rs1 0 0 0 imm1	0 1 0 0 0 1 1
					- m2 - imm2	imm1
PSEUDO (multiple)	/	Store Global Byte (imm more than 12bit)	auipc rt, imm[31:12]	sb rd, imm(rt)	imm[31:12] rt	0 0 1 0 1 1 1
			sb rd, imm[11:0](rt)		imm[11:5] rt rd 0 0 0 imm[4:0]	0 1 0 0 0 1 1
A	R	Store Conditional	if reserved { M[rs1] = rs2; rd=0} else {rd=1}	sc.w rd, rs1, rs2	0 0 0 1 1 aq rl rs2 rs1 0 1 0 rd	0 1 0 1 1 1 1
PSEUDO	I	Set if = zero	sltiu rd, rs, 1	seqz rd, rs	- 0 - 1 rs 0 1 1 rd	0 0 1 0 0 1 1
PSEUDO	R	Set if > zero	slt rd, x0, rs	sgtz rd, rs	- 0 - - - - - rs - 0 - 0 1 0 rd	0 1 1 0 0 1 1
I	S	Store Half Word	rs2[15:0] -> M[rs1+imm][15:0]	sh rs1, imm (rs2)	m2 imm2 rs2 rs1 0 0 1 imm1	0 1 0 0 0 1 1
					- m2 - imm2	imm1
PSEUDO (multiple)	/	Store Global Half Word (imm more than 12bit)	auipc rt, imm[31:12]	sh rd, imm(rt)	imm[31:12] rt	0 0 1 0 1 1 1
			sh rd, imm[11:0](rt)		imm[11:5] rt rd 0 0 1 imm[4:0]	0 1 0 0 0 1 1
I	R	Shift Left Logical	rd = rs1 << (zero extends) rs2	sll rd, rs1, rs2	- 0 - - - - - rs2 rs1 0 0 1 rd	0 1 1 0 0 1 1
I	I	Shift Left Logical Immediate	rd = rs1 << (zero extends) imm_sh	slli rd, rs1, imm_sh	- 0 - - - - - imm_sh rs1 0 0 1 rd	0 0 1 0 0 1 1
I	R	Set Less Than	{ 31'b0, (rs1[31] == rs2[31]) ? (rs1<rs2) : (rs1[31]) }	slt rd, rs1, rs2	- 0 - - - - - rs2 rs1 0 1 0 rd	0 1 1 0 0 1 1
I	I	Set Less Than Immediate	{ 31'b0, (rs1[31] == imm[31]) ? (rs1<imm) : (rs1[31]) }	slti rd, rs1 imm	m1 imm1 rs1 0 1 0 rd	0 0 1 0 0 1 1
					- m1 - imm1	
I	I	Set Less Than Unsigned Immediate	{ 31'b0, rs1 < imm }	sltiu rd, rs1 imm	m1 imm1 rs1 0 1 1 rd	0 0 1 0 0 1 1
					- m1 - imm1	
I	R	Set Less Than Unsigned	{ 31'b0, rs1 < rs2 }	sltu rd, rs1, rs2	- 0 - - - - - rs2 rs1 0 1 1 rd	0 1 1 0 0 1 1
PSEUDO	R	Set if < zero	slt rd, rs, x0	sltz rd, rs	- 0 - - - - - - 0 - rs 0 1 0 rd	0 1 1 0 0 1 1
PSEUDO	R	Set if not = zero	sltu rd, x0, rs	snez rd, rs	- 0 - - - - - rs - 0 - 0 1 1 rd	0 1 1 0 0 1 1
I	R	Shift Right Arithmetic	rd = rs1 (msb extends) >> rs2	sra rd, rs1, rs2	- 1 - - - - - rs2 rs1 1 0 1 rd	0 1 1 0 0 1 1
I	I	Shift Right Arithmetic Immediate	rd = rs1 (msb extends) >> imm_sh	srai rd, rs1, imm_sh	- 1 - - - - - imm_sh rs1 1 0 1 rd	0 0 1 0 0 1 1
I	R	Shift Right Logical	rd = rs1 (zero extends) >> rs2	srl rd, rs1, rs2	- 0 - - - - - rs2 rs1 1 0 1 rd	0 1 1 0 0 1 1
I	I	Shift Right Logical Immediate	rd = rs1 (zero extends) >> imm_sh	srli rd, rs1, imm_sh	- 0 - - - - - imm_sh rs1 1 0 1 rd	0 0 1 0 0 1 1
I	R	Subtract	rd = rs1 - rs2	sub rd, rs1, rs2	- 1 - - - - - rs2 rs1 0 0 0 rd	0 1 1 0 0 1 1
I	S	Store Word	rs2[31:0] -> M[rs1+imm][31:0]	sw rs1, imm (rs2)	m2 imm2 rs2 rs1 0 1 0 imm1	0 1 0 0 0 1 1
					- m2 - imm2	imm1
PSEUDO (multiple)	/	Store Global Word (imm more than 12bit)	auipc rt, imm[31:12]	sw rd, imm(rt)	imm[31:12] rt	0 0 1 0 1 1 1
			sw rd, imm[11:0](rt)		imm[11:5] rt rd 0 1 0 imm[4:0]	0 1 0 0 0 1 1
PSEUDO (multiple)	/	Tail Call Far-Away Subroutine	auipc x6, imm[31:12]	tail imm	imm[31:12] 0 0 1 1 0	0 0 1 0 1 1 1
			jalr x0, x6, imm[11:0]		imm[11:0] 0 0 1 1 0 0 0 0 0 0 0 0 0	1 1 0 0 1 1 1
I	R	Xor	rd = rs1 & rs2	xor rd, rs1, rs2	- 0 - - - - - rs2 rs1 1 0 0 rd	0 1 1 0 0 1 1
I	I	Xor Immediate	rd = rs1 ^ imm	xori rd, rs1, imm	m1 imm1 rs1 1 0 0 rd	0 0 1 0 0 1 1
					- m1 - imm1	

RV32IMAC

Instruction Set "Sliderule" Decoder

Designed to be printed on A3

Contributors: Simone Shawn Cazzaniga (magiwanders@github | www.magiwanders.com),

v0.1.6

Bitwise Breakdown - inst[31:0]

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	Meaning: Mnemonics	Meaning: Description
'I' Base + 'A' 'C' Extensions General Structure																																	
[redacted]																instr[15:2]														0	0	General / Quadrant 0	Compressed Instruction Structure ('C' Extension)
																														0	1	General / Quadrant 1	Compressed Instruction Structure ('C' Extension)
																														1	0	General / Quadrant 2	Compressed Instruction Structure ('C' Extension)
instr[31:7]																						opcode[6:2]								1	1	General / Quadrant 3	RV32IA Instruction Structure (non-compressed)
'I' Base + 'A' 'C' Extensions Immediate View																																	
[redacted]																0 0 0 imm2				imm1		m2	m1	rd'						0	0	**Put SP+nzuimm in rd'**	rd' = sp + nzuimm	**CIW - Type** "Wide Immediate"	**Quadrant 0**	**'C' Extension**
																- 0 -				imm1			imm2	m1	m2					- 0 -		nzuimm	CIW non zero unsigned immediate			
																0 1 0 imm				rs1'		m2	m1	rd'						0	0	**Load Word**	rd' <- M[rs1' + offset]	**CL - Type** "Load"		
																- 0 -						m1			imm		m2			- 0 -		offset	CL Load word offset			
																1 1 0 imm				rs1'		m2	m1	rs2'						0	0	**Store Word**	rs2' -> M[rs1' + offset]	**CS - Type** "Store"		
																- 0 -						m1			imm		m2			- 0 -		offset	CS Store word offset			
																1 0 0 0 1 1				rd'		f2		rs2'						0	1	**CA Register Operations**	rd' = rd' (f2) rs2'	**CA** "Arithmetic" **Type**		
																funct3				m1	imm3			rs1'		imm2		imm1	m2	0	1	**Branch**	if (rs1' (funct3) x0) PC += offset	**CB - Type** "Branch / Arithmetic"		
																- m1 -						m1	imm2	m2	imm3		imm1			0		offset	CB branch offset			
																1 0 0 m1 f2				rd'					imm1					0	1	**Shift Right**	rd = rd >> shamt (f2)			
																- 0 -								m1		imm1						shamt	CB non zero shift immediate			
																1 0 0 m1 1 0				rd'					imm1					0	1	**And Immediate**	rd &= imm			
																- m1 -									imm1							imm	CB And immediate			
																funct3			m1	m2	imm2	m3	m4	m5		imm1		m6		0	1	**Jump (and Link)**	(rd = PC + 2 ;) PC += offset	**CJ - Type** "Jump"		
																- m1 -					m3	imm2	m5	m4	m6	m2		imm1		0		offset	CJ jump offset			
																0 0 0 0				- 0 -					- 0 -					0	1	**No OPeration**	Litterally do nothing			
																0 0 0 m1				rd != 0					imm1					0	1	**Add Immediate**	rd += nzimm			
																- 0 -								m1		imm1						nzimm	CI add non-zero immediate			
																0 1 0 m1				rd != 0					imm1					0	1	**Load Immediate**	rd = imm			
																- m1 -									imm1							imm	CI load immediate			
																0 1 1 m1				rd != 0, 2					imm1					0	1	**Load Upper Immediate**	rd = imm << 12 (zero extends)	**CI - Type** "Immediate"		
- m1 -											imm1					- 0 -									imm							imm	CI load upper immediate			
[redacted]																0 1 1 m1				rd != 0		m2	m3	rd'						0	1	**Add nzimm to SP**	sp += nzimm			
																- 0 -					m1	imm1		m3	m4	m2			- 0 -		nzimm	CI add-to-SP non-zero immediate			
																0 0 0 m1				rd != 0					imm2		imm1			1	0	**Logical Left Shift Immediate**	rd = rd << shamt (zero extends)			
																- 0 -									0		imm2		imm1			shamt	CI non zero shift immediate			
																0 1 0 m1				rd != 0					imm2		imm1			1	0	**Load Word from SP**	rd <- M[sp + offset]			
																- 0 -						imm1		m1		imm2				- 0 -		offset	CI memory load offset immediate			
																1 1 0 imm2				imm1				rs						1	0	**Store Word to SP**	rs' -> M[sp + offset]	**CSS - Type** "Stack-relative Store"		
																- 0 -						imm1				imm2				- 0 -		offset	CSS memory store offset immediate			
																1 0 0 1				- 0 -					- 0 -					1	0	**Environment Break**	Control Back to Debug Environment	**CR - Type** "Register"		
																1 0 0 f				rs1 != 0					- 0 -							**CR Control Transfer Instructions**	(ra = PC + 2;) PC = rs1			
																				rd != 0					rs != 0							**CR Register Operations**	rd = rd (funct) rs1			
m2		imm2			rs2			rs1			funct3					imm1									0 1 0 0 0						1	1	**Store**	**funct(M[rs1+imm]) <- funct(rs2)**	**S - type** "Store"		
																									0 1 0 0 1								**?**				
- m2 -																imm2									imm1								**imm**	**S - immediate**			
m2		imm2			rs2			rs1			funct3				m1	imm1								1 1 0 0 0						1	1	**Branch**	**if (rs1 (funct) rs2) PC += imm**	**B - type** "Branch"	**Quadrant 3**	**'I' Base**
																																imm	**B - immediate**			
																funct7									1 0 1 0 0								**?**		**R - type** "Register"		
																- f - - - - -				rs2				rs1		funct3				rd					0 1 1 0 0						1	1	**Operations with Registers**	**rd = rs1 (funct) rs2**			**'M' Extension**
																- - - - - - 1																**'M' Extension**	**Multiplication and Division**			
																funct7									0 1 1 1 0								**?**				
																funct5		aq	rl						0 1 0 1 1								**'A' Extension**	**Atomic Operations**			**'A' Extension**
m1											imm1														0 0 0 0 0								**Load**	**rd = funct(M[rs1+imm])**	**I - type** "Immediate"		
m1											imm1														0 0 1 0 0								**Operations with Immediate**	**rd = rs1 (funct) imm**			
-	f	-	-	-	-	-					imm_sh																						**Shift Operations with Immediate**				
-	-	-	-	-	-	-	-	-	-	f						rs1				funct3				rd					1 1 0 0 0				**Environment**	**Control to Debug / OS**			
m1											imm1														0 0 1 1 0								**?**				
m1											imm1														0 0 0 0 1												
m1											imm1														1 1 0 0 1								**Jump and Link Register**	**rd = PC + 4 ; PC = rs1 + imm**			**'I' Base** cont'd
- m1 -																						imm1											**imm / uimm**	**I - immediate**			
imm1																				rd					0 0 1 0 1								**Add Upper Immediate to PC**	**rd = PC + imm**	**U - type** "Upper Immediate"		
																									0 1 1 0 1								**Load Upper Immediate**	**rd = imm**			
imm1																						- 0 -											**imm**	**U - immediate**			
m2		imm2							m1		imm1														1 1 0 1 1						1	1	**Jump and Link**	**rd = PC + 4 ; PC += imm**	**J - type** "Jump"		
- m2 -											imm1																						**imm**	**J - immediate**			
Exploded View																																	
[redacted]																funct3														0	0	**Quadrant 0 Compressed Instruction Opcode**				
																0 0 0 0 0 0 0 0 0 0 0				rd'										0	0	RESERVED				
																0 0 0 imm2				imm1				m2	m1					0	0	Add imm*4 to SP and put it in rd'	**c.addi4spn**	**rd', imm**	addi rd', x2, 4*imm	rd' = sp + nzuimm = sp + 4*imm	**CIW - Type** "Wide Immediate"
																- 0 -							imm						- 0 -		4 * imm = nzuimm	CIW non zero unsigned immediate			
																0 0 1																fld ?				
																0 1 0 imm				rs1'		m2	m1	rd'						0	0	Load Word	**c.lw**	**rd', imm(rs1')**	lw rd, offset(rs1')	rd' <- M[rs1' + offset]	**CL - Type** "Load"
																- 0 -								m1	imm		m2			- 0 -		4 * imm = offset	CL Load word offset			
																0 1 1																flw ?				
																1 0 0																RESERVED				
																1 0 1																fsd ?				
																1 1 0 imm1				rs1'		m2	m1	rs2'						0	0	Store Word	**c.sw**	**rs1', imm(rs2')**	sw rs1', offset(rs2')	rd2' -> M[rs1' + offset]	**CS - Type** "Store"
																- 0 -								m1	imm	imm1	m2			- 0 -		4 * imm = offset	CS Store word offset			
																1 1 1																fsw ?				
																funct3														1	0	**Quadrant 2 Compressed Instruction Opcode**				
																0 0 0 0 0 0 0 0 0 0 0				rd										1	0	RESERVED				
																- 0 -								0	imm							Shift Left Logical Imm	**c.slli**	**rd, imm**	slli rd, rd, shamt	rd = rd <<0 shamt	**CI - Type** "Immediate"
																0 0 1																imm = shamt	CI non zero immediate			
																0 1 0 m1				rd != 0					imm2		imm1			1	0	RESERVED				
																- 0 -									imm							Load Word from SP	**c.lwsp**	**rd, imm**	lw rd, offset(x2)	rd <- M[sp + offset]	**CI - Type** "Immediate"
																0 1 1																4 * imm = offset	CI memory load offset immediate			
																1 0 0																				
																			1	- 0 -					- 0 -								Environment Break	**c.ebreak**		Control Back to Debug Environment		**CR - Type** "Register"
																				rs1 != 0					- 0 -							**CR Control Transfer Instructions**	(ra = PC + 2;) PC = rs1			
																			0														Jump to Register	**c.jr**	**rd,rs1**	jalr x0, 0(rs1)	just	PC = rs1
																			1														Jump and Link Register	**c.jalr**	**rs1**	jalr x1, 0(rs1)	ra = PC + 2	
																				- 0 -					rs != 0							HINTs	PC += 4 and increment applicable performance counters			
																				rd != 0					rs != 0							**CR Register Operations**	rd = rd (funct) rs1			
																			0														MoVe register	**c.mv**	**rd,rs**	add rd, x0, rs	rd = rs	
																			1														Add	**c.add**		add rd, rd, rs	rd += rs	
																1 0 1																				
																1 1 0 imm2				imm1				rd						1	0	Store Word to SP	**c.swsp**	**rd, offset**	sw rd, offset(x2)	rs2' -> M[sp + offset]	**CSS - Type** "Stack-relative Store"
																- 0 -							imm						- 0 -		4 * imm = offset	CSS memory store offset immediate			
																1 1 1																				
																funct3														0	1	**Quadrant 1 Compressed Instruction Opcode**				
																0 0 0 0 - 0 -									- 0 -					0	1	**No OPeration**	**c.nop**	/	Literally do nothing.		**CI - Type** "Immediate"
																0 0 0 m1									imm1					0	1	HINTs	PC += 4 and increment applicable performance counters			
																0 0 0 m1				rd != 0					imm1					0	1	**Add Immediate**	**c.addi**	**rd, imm**	addi rd, rd, nzimm	rd += nzimm	
																- m1 -										imm1							(sign extend) imm = nzimm	CI add non-zero immediate			
																0 0 1 m1	m2	imm2	m3	m4	m5		imm1		m6		0	1	Jump and Link	**c.jal**	**offset**	jal x1, offset	rd = PC + 2 ; PC += offset	**CJ - Type** "Jump"
																- m1 -					m3	imm2	m5	m4	m6	m2		imm1		0		offset	CJ jump offset			
																0 1 0 m1				rd != 0					imm1					0	1	Load Immediate	**c.li**	**rd, imm**	addi rd, x0, imm	rd = imm	**CI - Type** "Immediate"
																- m1 -										imm1							imm	CI load immediate			
																0 1 1 0 rd != 0, 2					0 0 0 0 0									0	1	RESERVED				
																0 m1									imm1							Load Upper Immediate	**c.lui**	**rd, imm**	lui rd, nzimm	rd = imm << 12 (zero extends)	
- m1 -			imm													- 0 -																(msb extend) imm << 12 = nzimm	CI load upper immediate			
		m1	imm1																																
[redacted]																1 0 0																**CA, CB instructions**				
																1 0 0 0 f2				rd'					imm					0	1	**Shift Right**	rd = rd >> shamt (f2)	**CB - Type** "Branch / Arithmetic"		
																- 0 -								0	imm							shamt	CB non zero shift immediate			
																- 0 -																HINTs	PC += 4 and increment applicable performance counters			
																0 0																Shift Logical Right Immediate	**c.srli**	**rd, shamt**	srli rd', rd'. shamt	rd = rd (zero)>> shamt	
																0 1																Shift Arithmetic Right Immediate	**c.srai**	**rd, shamt**	srai rd', rd'. shamt	rd = rd (msb)>> shamt	
																1 0 0 m1	1	0		rd'					imm1					0	1	And Immediate	**c.andi**	**rd, imm**	andi rd', rd', imm	rd &= imm	
																- m1 -										imm1							imm	CB And immediate			
																1 0 0 0 1 1				rd'		f2			rs2'					0	1	**CA Register Operations**	rd' = rd' (f2) rs2'	**CA - Type** "Arithmetic"		
																								0 0								Sub	**c.sub**	**rd,rs1**	sub rd', rd', rs2'	rd' -= rs2'	
																								1 0								Or	**c.or**	**rd,rs1**	or rd', rd', rs2'	rd' |= rs2'	
																								0 1								Xor	**c.xor**	**rd,rs1**	xor rd', rd', rs2'	rd' ^= rs2'	
																								1 1								And	**c.and**	**rd,rs1**	and rd', rd', rs2'	rd' &= rs2'	
																1 0 1 m1	m2	imm2	m3	m4	m5		imm1		m6		0	1	Jump	**c.j**	**offset**	jal x0, offset	PC += offset	**CJ - Type** "Jump"
																- m1 -					m3	imm2	m5	m4	m6	m2		m2		0		offset	CJ jump offset			
																funct3			m1	imm3		rs1'			imm2		imm1	m2	0	1	**Branch**	if (rs1' (funct3) x0) PC += offset	**CB - Type** "Branch / Arithmetic"		
																- m1 -							imm								(msb extended) imm << 1 = offset	CB branch offset			
																1 1 0									m1	imm2	m2	imm3	imm1	0		Branch if EQual to Zero	**c.beqz**	**rs1', imm**	beq rs1', x0, offset	if (rs1' == x0) PC += offset	
																1 1 1																Branch if Not Equal to Zero	**c.bnez**		bne rs1', x0, offset	if (rs1' != x0) PC += offset	
																														1	1	**Quadrant 3 Non-Compressed Instructions Opcode**				
																						opcode[6:2]										Rest of the opcode				
m1		imm1					rs1			funct3				rd					0	0	0	0	0	1	1	**Load**	**rd = M[rs1 + (u)imm] [funct]**	**I - type** "Immediate"		
- m1 -																imm1																**imm / uimm**	**I - immediate**			
																									0 0 0								**Load**		**Byte**	**lb**	rd, imm(rs1)	rd	[7:0]	<-	M[rs1 + imm]	[7:0]
																									1 0 0								**Load**	**Unsigned**		**lbu**					M[rs1 + uimm]	
																									0 1 0								**Load**		**Word**	**lw**			[31:0]		M[rs1 + imm]	[31:0]
																									1 1 0																	
																									0 0 1								**Load**		**Half-Word**	**lh**			[15:0]		M[rs1 + imm]	[15:0]

RV32IMAC

Instruction Set "Sliderule" Decoder

Designed to be printed on A3

Contributors: Simone Shawn Cazzaniga (magiwanders@github | www.magiwanders.com),

v0.1.6

Bitwise Breakdown - inst[31:0]

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	Meaning: Mnemonics			Description					
																	1	0	1													Load	Unsigned	Half-Word	lhu			[15:0]		M[rs1 + uimm]	[15:0]
																	0	1	1																					
																	1	1	1																					
																									1	0	0	0	0			R4 - type								
m2	imm2						rs2				rs1						funct3			imm1					0	1	0	0	0	1	1	Store			funct(M[rs1+imm]) <- funct(rs2)					S - type "Store"
- m2 -																				imm2					imm1							imm			S - immediate					
																	0	0	0													Store	Byte		sb	rs1, imm(rs2)	rs2	[7:0]	->	M[rs1+imm]	[7:0]
																	1	0	0																					
																	0	1	0														Word		sw			[31:0]			[31:0]
																	1	1	0																					
																	0	0	1														Half-Word		sh			[15:0]			[15:0]
																	1	0	1																					
																	0	1	1																					
																	1	1	1																					
m2	imm2						rs2				rs1						funct3			imm1			m1		1	1	0	0	0	1	1	Branch			if (rs1 (funct) rs2) PC += imm					B - type "Branch"
- m2 -																				m1	imm2				imm1					0		imm			B - immediate					
																	0	0	0													Branch if	EQual		beq	rs1, rs2, imm	if	rs1 == rs2		PC += imm	
																	1	0	0														Less Than		blt			(rs1 < rs2) ^ (rs1[31] != rs2[31])			
																	0	1	0																					
																	1	1	0														Less Than	Unsigned	bltu			rs1 < rs2			
																	0	0	1														Not Equal		bne			rs1 != rs2			
																	1	0	1														Greater or Equal		bge			(rs1 >= rs2) ^ (rs1[31] != rs2[31])			
																	0	1	1																					
																	1	1	1														Greater or Equal	Unsigned	bgeu			rs1 >= rs2			
m1	imm1																															Operations with Immediate			rd = rs1 (funct) imm					I - type "Immediate"
-	f	-	-	-	-	-	imm_sh				rs1						funct3			rd					0	0	1	0	0	1	1									
- m1 -																				imm1												imm			I - immediate					
																	0	0	0														Add		addi	rd, rs1, imm	rd =	rs1	+		imm
																	1	0	0														Xor		xori	rd, rs1, imm	rd =	rs1	^		imm
																	0	1	0														Set Less Than		slti	rd, rs1, imm	rd =	{ 31'b0, (rs1[31] == imm[31]) ? (rs1<imm) : (rs1[31]) }			
																	1	1	0														Or		ori	rd, rs1, imm	rd =	rs1	|		imm
	0						imm_sh										0	0	1													Immediate	Shift Logical	Left	slli	rd, rs1, imm_sh	rd =	rs1	<<	zero extends	imm_sh
	0						imm_sh										1	0	1														Shift Logical	Right	srli		rd =	rs1	zero extends	>>	imm_sh
	1						imm_sh																										Shift Arithmetic		srai		rd =	rs1	msb extends	>>	imm_sh[4:0]
																	0	1	1														Set Less Than	Unsigned	sltiu	rd, rs1, imm	rd =	{ 31'b0, rs1 < imm }			
																	1	1	1														And		andi	rd, rs1, imm	rd =	rs1	&		imm
funct7							rs2				rs1						funct3			rd					1	0	1	0	0			R - type								
-	f	-	-	-	-	-	rs2				rs1						funct3			rd					0	1	1	0	0	1	1	Operations with Registers			rd = rs1 (funct) rs2					R - type
	0																0	0	0														Add		add	rd, rs1, rs2	rd =	rs1	+		rs2
	1																																Subtract		sub				-		
	0																1	0	0														Xor		xor	rd, rs1, rs2	rd =	rs1	^		rs2
	0																0	1	0														Set Less Than		slt	rd, rs1, rs2	rd =	{ 31'b0, (rs1[31] == rs2[31]) ? (rs1<rs2) : (rs1[31]) }			
	0																1	1	0														Or		or	rd, rs1, rs2	rd =	rs1	|		rs2
	0																0	0	1													Shift	Left	Logical	sll	rd, rs1, rs2	rd =	rs1	<<	zero extends	rs2[4:0]
	0																1	0	1														Right		srl			rs1	zero extends	>>	rs2[4:0]
	1																																	Arithmetic	sra				msb extends	>>	rs2[4:0]
	0																0	1	1														Set Less Than Unsigned		sltu	rd, rs1, rs2	rd =	{ 31'b0, rs1 < rs2 }			
	0																1	1	1														And		and	rd, rs1, rs2	rd =	rs1	&		rs2
-	-	-	-	-	-	1	rs2				rs1						funct3			rd					0	1	1	0	0	1	1	'M' Extension			Multiplication and division					R - type
																	0	0	0														Mul SignedxSigned		mul	rd, rs1, rs2	rd =	rs1	s * s		rs2
																	1	0	0														Div		div				/		
																	0	1	0														Mul High SignedxUnsigned		mulhsu				s * u [63:32]		
																	1	1	0														Remainder		rem				%		
																	0	0	1														Mul High SignedxSigned		mulh				s * s [63:32]		
																	1	0	1														Div Unsigned		divu				/ unsigned		
																	0	1	1														Mul High UnsignedxUnsigned		mulhu				u * u [63:32]		
																	1	1	1														Remainder Unsigned		remu				% unsigned		
-	-	-	-	-	-	-	-	-	-	-	f		rs1				funct3			rd					1	1	1	0	0	1	1	Environment			Control to Debug / OS					I - type
											0						0	0	0													Environment	Call		ecall						
											1																							Break		ebreak						
																									0	0	0	1	0												
																									1	0	0	1	0			R4 - type								
																									0	1	0	1	0						-					
																									1	1	0	1	0						-					
imm[11:0]													rs1				funct3			rd					0	0	1	1	0			I - type								
																									1	0	1	1	0						-					
funct7							rs2				rs1						funct3			rd					0	1	1	1	0			R - type								
																									1	1	0	1	0						-					
imm[11:0]													rs1				funct3			rd					0	0	0	0	1			I - type								
																									1	0	0	0	1			R4 - type								
imm[11:5]							rs2				rs1						funct3			imm[4:0]					0	1	0	0	1			S - type								
m	imm1										rs1						0	0	0	rd					1	1	0	0	1	1	1	Jump and Link Register		jalr	rd, rs1, imm	rd = PC + 4 ; PC = rs1 + imm				I - type "Immediate"
- m -																				imm1												imm			I - immediate					
imm1																				rd					0	0	1	0	1	1	1	Add Upper Immediate to PC		auipc	rd, imm	rd = PC + imm << 12				U - type "Upper Immediate"
imm1																				- 0 -												imm			U - immediate					
																									1	0	1	0	1						-					
imm1																				rd					0	1	1	0	1	1	1	Load Upper Immediate		lui	rd, imm	rd = imm << 12				U - type "Upper Immediate"
imm1																				- 0 -												imm			U - immediate					
																									1	1	1	0	1						-					
																									0	0	0	1	1						-					
																									1	0	0	1	1			R4 - type								
funct5					aq	rl					rs2				rs1		0	1	0	rd					0	1	0	1	1	1	1	'A' Extension								R - type
					0	0																										NO ORDERING CONSTRAINTS								
					1	0																										ACQUIRE ACCESS								
					0	1																										RELEASE ACCESS								
					1	1																										SEQUENTIALLY CONSISTENT ACCESS								
funct5																																								
0	0	0	0	0																												Atomic Add		amoadd.w	rd, rs1, rs2	rd <-	M[rs1]	+rs2		M[rs1] = rd
1	0	0	0	0																												Atomic Min		amomin.w	rd, rs1, rs2	rd <-	min(M[rs1], rs2)			M[rs1] = rd
0	0	1	0	0																												Atomic Xor		amoxor.w	rd, rs1, rs2	rd <-	M[rs1]	^ rs2		M[rs1] = rd
1	0	1	0	0																												Atomic Max		amomax.w	rd, rs1, rs2	rd <-	max(M[rs1], rs2)			M[rs1] = rd
0	1	1	0	0																												Atomic And		amoand.w	rd, rs1, rs2	rd <-	M[rs1]	& rs2		M[rs1] = rd
0	0	0	1	0																												Load Reserved		lr.w	rd, rs1	rd <-	M[rs1]			Reserve M[rs1]
0	1	0	1	0																												Atomic Or		amoor.w	rd, rs1, rs2	rd <-	M[rs1]	| rs2		M[rs1] = rd
0	0	0	0	1																												Atomic Swap		amoswap.w	rd, rs1, rs2	rd <-	M[rs1]	swap(rd, rs2)		M[rs1] = rd
0	0	0	1	1																												Store Conditional		sc.w	rd, rs1, rs2	if reserved { M[rs1] = rs2; rd=0} else {rd=1}				
m2	imm2					m1	imm1													rd					1	1	0	1	1	1	1	Jump and Link		jal	rd, imm	rd = PC + 4 ; PC += imm				J - type "Jump"
- m2 -							imm1													m1	imm2									0		imm			J - immediate					
																									0	0	1	1	1						-					
																									1	0	1	1	1						-					
																									0	1	1	1	1						-					
																									1	1	1	1	1						-					