Decoding RISCV instructions

Louis Geoffroy

September 2022

1 Introduction

This document has for purpose to synthesise the syntax and decoding of RISCV RV32IZiscr ISA.

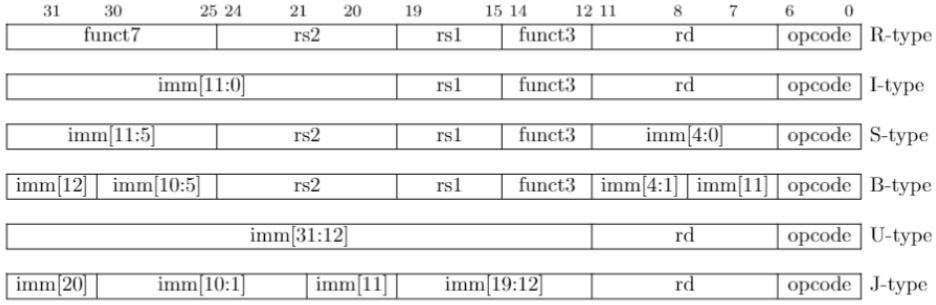


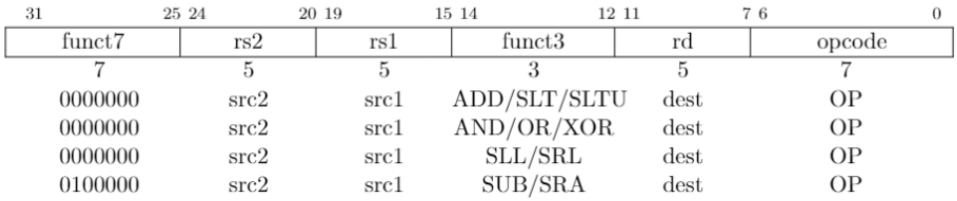
Figure 1: RISC-V inscrutions de base sans extension

To decode an instruction, the 32 bits of an instruction are split in several part, **funct3**, **funct7** and **opcode**. These numbers provide the information needed to decode the type of instruction.

1.0.1 R-type :

1.0.1.1 Parameters

These instruction are arithmetic instructions using 2 sources registers :



31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Figure 2: RISC-V inscrutions de type R

- **rs2** : source register number 2
- **rs1** : source register number 1
- **rd** : destination register
- **opcode** = 0110011 : This value indicate that the instruction is an R-type

1.0.1.2 Operations encoding

Operations :

- **add rd, rs1, rs2** : rd = rs1 + rs2, overflow ignored
- **sub rd, rs1, rs2** : rd = rs2-rs1, overflow ignored
- **slt rd, rs1, rs2** : rd = 1 if rs1 < rs2 else 0. Signed comparaison
- **sltu rd, rs1, rs2** : rd = 1 if rs1 < rs2 else 0. Unsigned comparaison
- **and rd, rs1, rs2** : rd = rs1 & rs2
- **or rd, rs1, rs2** : rd = rs1 $\|$ rs2
- **xor rd, rs1, rs2** : rd = rs1 $\wedge$ rs2
- **sll rd, rs1, rs2** : shift logic left on rs1. Shift value are the lsb of rs2
- **srl rd, rs1, rs2** : shift logic right on rs1. Shift value are the lsb of rs2
- **sra rd, rs1, rs2** : shift arithmetic right on rs1. Shift value are the lsb of rs2

1.0.1.3 Opcode value

Instruction	funct7	funct3	opcode
ADD	0000000	000	0110011
SLT	0000000	010	0110011
SLTU	0000000	011	0110011
AND	0000000	111	0110011
OR	0000000	110	0110011
XOR	0000000	100	0110011
SLL	0000000	001	0110011
SRL	0000000	101	0110011
SUB	0100000	000	0110011
SRA	0100000	101	0110011

1.0.2 I-type :

1.0.2.1 Parameters

These instruction are very similar to the R-type, except rs2 is a 12 bits immediat value.
This immediat is always a 32 sign-extended value.
I-type are also used to encode load instruction which will be detailled in the section 1.0.7.1.

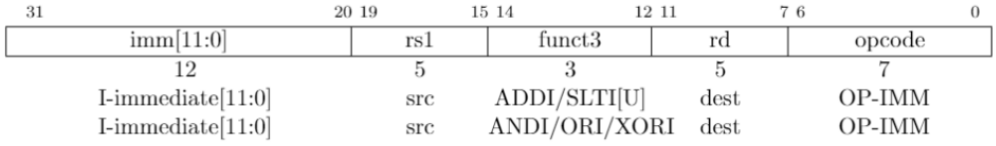


Figure 3: RISC-V I-type instruction

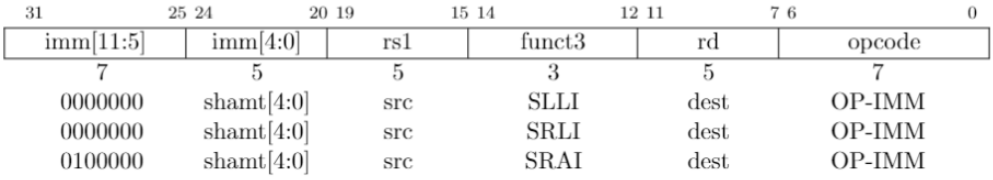


Figure 4: RISC-V shift with immediat

- **imm[11:0]** : 12 bits immediat that must be loaded in the second operand

- **rs1** : source register
- **rd** : destination register

1.0.2.2 Operations encoding

Operations :

- **addi rd, rs1, imm** : rd = rs1 + imm, overflow ignored
- **slti rd, rs1, imm** : rd = 1 if (rs1 $<$ $imm[11:0]$) else 0. Signed comparaison
- **sltiu rd, rs1, imm** : rd = 1 if (rs1 $<$ $imm[11:0]$) else 0. Unsigned comparaison
- **andi rd, rs1, imm** : rd = rs1 & imm
- **ori rd, rs1, imm** : rd = rs1 $\|$ imm
- **xori rd, rs1, imm** : rd = rs1 $\wedge$ imm
- **slli rd, rs1, imm** : Shift logic left. Shift value are the immediat 5 lsb
- **srli rd, rs1, imm** : Shift logic right. Shift value are the immediat 5 lsb
- **srai rd, rs1, imm** : Shift arithmetic right. Shift value are the immediat 5 lsb

1.0.2.3 Opcode value

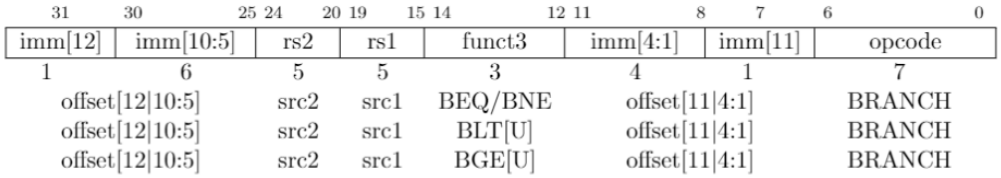
Instruction	funct3	opcode
ADDI	000	0010011
SLTI	010	0010011
SLTIU	011	0010011
ANDI	111	0010011
ORI	110	0010011
XORI	100	0010011
SLLI	001	0010011
SRLI	101	0010011
SRAI	101	0010011

Remarque : nop instruction is encoded as a **addi x0,x0,0**

1.0.3 B-type :

1.0.3.1 Parameters

B-type instruction are conditional branch instructions. Please notice there is no delay slot is the RISCV architecture.
All of these instruction does a comparaison between the source operand rs1 and the source operand rs2. If the comparaison succeed the program will jump.



31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

Figure 5: RISC-V inscrutions de type B

- **imm[11:0]** : 12 bits immediat that must be loaded in the second operand
- **rs1** : source register
- **rs2** : destination register

1.0.3.2 Operations encoding

Operations :

- **beq rs1, rs2** : jump if rs1 = rs2
- **bne rs1, rs2** : jump if rs1 ! = rs2
- **blt rs1, rs2** : jump if rs1 < rs2
- **bltu rs1, rs2** : jump if rs1 < rs2, with rs1 and rs2 unsigned integer
- **bge rs1, rs2** : jump if rs1 >= rs2
- **bgeu rs1, rs2** : jump if rs1 >=, rs2 with rs1 and rs2 unsigned integer

bgt/bgtu, ble/bleu are the same than blt/bltu and bge/bgeu with operand reversed.

1.0.3.3 Opcode value

Instruction	funct3	opcode
BEQ	000	1100011
BNE	001	1100011
BLT	100	1100011
BGE	101	1100011
BLTU	110	1100011
BGEU	111	1100011

1.0.4 U-type :

1.0.4.1 Parameters

U-type instructions allow to load an immediate value in the most significant bits of a register

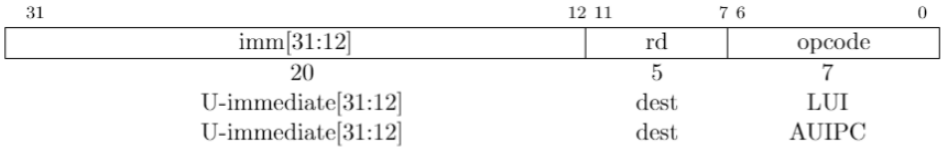


Figure 6: RISC-V inscrutions de type U

- **imm[31:12]** : 20 bits immediat that is loaded in the most significant bits of the destination register
- **rd** : destination register

1.0.4.2 Operations encoding

Operations :

- **lui rd** : rd = imm[31:12], the immediat value is loaded in the bits 31 to 12 and 0 is put in bits 11 to 0
- **auipc rd** : rd = PC + imm[31:12], the value of PC is added to the immediate value

1.0.4.3 Opcode value

Instruction	opcode
LUI	0110111
AUIPC	0010111

1.0.5 J-type :

1.0.5.1 Parameters

J-type are unconditionnal branch.

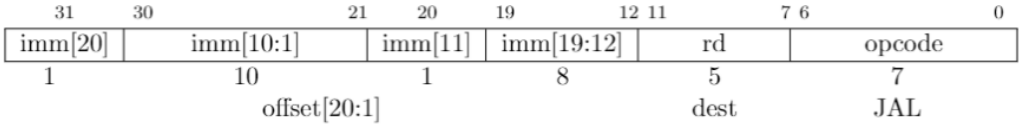


Figure 7: RISC-V inscrutions de type J

- **imm[20:1]** : 20 bits immediat that is added to PC value
- **rd** : destination register

1.0.5.2 Operations encoding

- **jal rd** : PC = PC + imm — rd = current PC +4, the immediat value is added to PC and rd save the return adress.

1.0.5.3 Opcode value

Instruction	opcode
JAL	1101111

1.0.6 Jalr-type :

1.0.6.1 Parameters

The jalr instruction is almost the same than the jal one excepting the fact that it can jump to relative adresses using a source register. Because of this, this operand is encoded like a I-instruction.

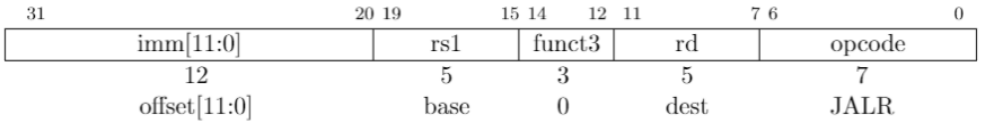


Figure 8: RISC-V inscrution jalr de type I

- **imm[11:0]** : 12 bits immediat that is added to PC value
- **rs1** : source registrer
- **rd** : destination register

1.0.6.2 Operations encoding

- **jalr rd, imm,(rs1)** : PC = PC + imm + rs1 — rd = current PC +4

1.0.6.3 Opcode value

Instruction	funct3	opcode
JALR	000	1100111

The standard software calling convention uses x1 as the return address register and x5 as an alternate link register.

1.0.7 Memory access :

1.0.7.1 Parameters

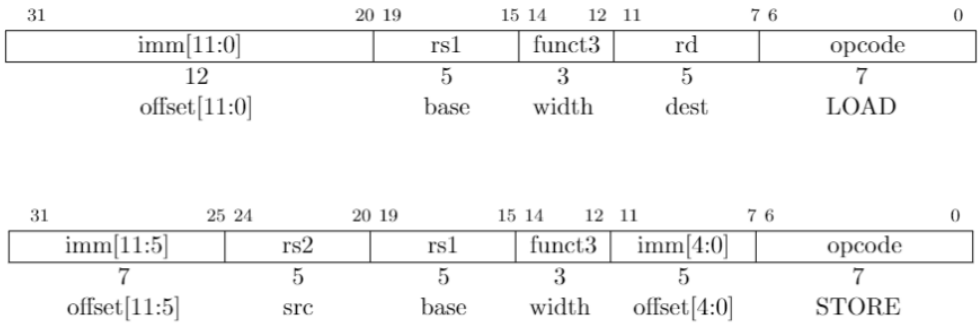


Figure 9: RISC-V inscrutions de type accès mémoire

Load instructions are encoded with the I format while store instructions are encoded as S-type.

1.0.7.2 Operations encoding

[data] will be the data contained at the memory adress data while data is just a data. So for instance if i write :
$rd = [0x004]$ it means rd will receive the value contain at the adress 0x004. And if you put a register instead of a number, like $[rs1]$ it means the value contain at the adress pointed by the value inside rs1.

- **lw rd, imm(rs1)** : rd = [imm + rs1]
- **lh rd, imm(rs1)** : rd = [imm + rs1] 0x0000FFFF, we load a half word considered as signed-integer (need to sign extend)
- **lhu rd, imm(rs1)** : rd = [imm + rs1] 0x0000FFFF, we load a half word considered as unsigned-integer (no need to sign extend)
- **lb rd, imm(rs1)** : rd = [imm + rs1] 0x000000FF, we load a byte word considered as signed-integer (need to sign extend)
- **lbu rd, imm(rs1)** : rd = [imm + rs1] 0x000000FF, we load a byte word considered as unsigned-integer (no need to sign extend)
- **sw rs2, imm(rs1)** : [imm + rs1] = rs2, we store a word, so adress must be word-aligned
- **sh rs2, imm(rs1)** : [imm + rs1] = rs2, we store a half-word, so adress must be half-word-aligned
- **sb rs2, imm(rs1)** : [imm + rs1] = rs2, we store a byte, so adress must be byte-aligned

1.0.7.3 Opcode value

Instruction	funct3	opcode
LW	010	0000011
LH	001	0000011
LHU	101	0000011
LB	000	0000011
LBU	100	0000011
SW	010	0100011
SH	001	0100011
SB	000	0100011

1.1 Ecall Ebreak

1.1.0.1 Parameters

These instructions are systems instructions allowing to do a system call and to execute a specific task. It is both encoded as I-type.

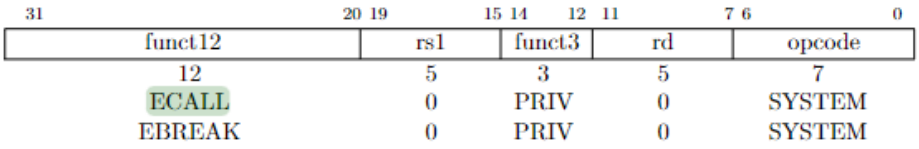


Figure 10: RISC-V systeme call

- **rs1** : source registrer
- **rd** : destination register

1.1.0.2 Operations encoding

- **ecall rd, rs1** : rd = rs1
- **ebreak rd, rs1** : rd = rs1

1.1.0.3 Opcode value

Instruction	funct12	rs1	funct3	rd	opcode
ecall	000000000000	00000	000	00000	1110011
ebreak	000000000001	00000	000	00000	1110011

1.2 CSR Instructions

1.2.0.1 Parameters

These are the instructions for accessing the CSR registers, the equivalent of CP0 in MIPS. Their encoding is as follows :

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

Figure 11: RISC-V CSR Instruction

- **rd** : destination register
- **source[11,0]** : adresse of a csr register
- **rs1**/**imm** : source registrer or immediat value

1.2.0.2 Operations encoding

- **csrrw rd, @csr, rs1** :
  rd = [csr] and [csr] = rs1
  The destination register will receive the value in the CSR, the CSR will be written with the value in rs1,
- **csrrc rd, @csr, rs1** :
  rd = [csr] and [csr] = ([csr] & !rs1)
  Allow to clear bits in the csr register,
- **csrrs rd, @csr, rs1** :
  rd = [csr] and [csr] = ([csr] || rs1)
  Allow to set the bits in the csr register,
- **csrrw rd, @csr, imm** :
  rd = [csr] and [csr] = imm

- **csrrc rd, @csr, imm** :
  rd = [csr] and [csr] = ([csr] & !imm)
- **csrrs rd, @csr, imm** :
  rd = [csr] and [csr] = ([csr] || imm)

1.2.0.3 Opcode value

Instruction	funct3	opcode
CSRRW	001	1110011
CSRRS	010	1110011
CSRRC	011	1110011
CSRRWI	101	1110011
CSRRSI	110	1110011
CSRRCI	111	1110011

1.3 Conclusion :

Instruction type	opcode
R-Type	0110011
I-Type	0010011
S-Type	0100011
B-Type	1100011
U-Type	0110111
J-Type	1101111
Privilege	1110011

Attention for memory accesses although the encoding is the same as for I and S, the opcodes are different.