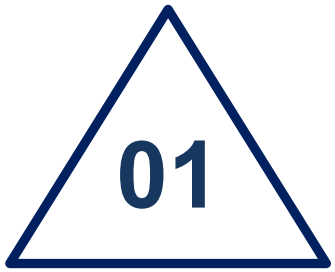
RISC-V ISA

Chapter 3
Standard Extensions

CONTENTS



Introduction

Definitions for standard extension

3.1 Introduction

RISC-V has defined base integer instructions and some privilege instructions, another goal of RISC-V is to provide a basis for more specialised instruction-set extensions or more customised accelerators.

- **Standard Extension**:

Designed for general usages, which is not conflict with another standard extensions.

- **Non-standard Extension**:

Usually designed highly specialised, which may cause conflict with other extensions. Not covered here.

3.1 Introduction

Standard Extension	Description
M	Integer Multiply/Divide extension
F	Single-precision floating-point extension
D	Double-precision floating-point extension
Q	Quad-precision floating-point extension
A	Atomic extension
C	Compressed extension

02

Standard Extensions

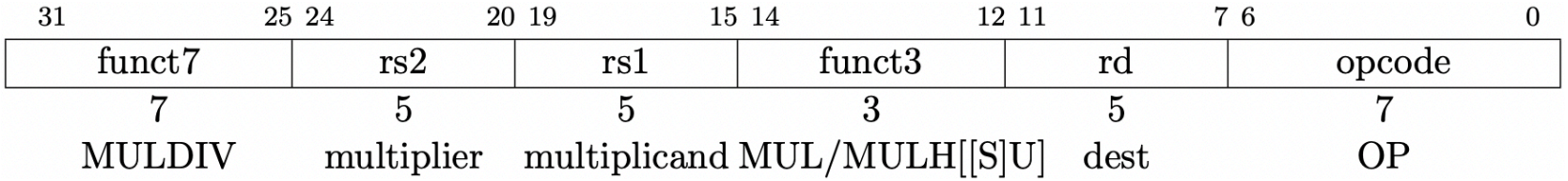
Cover extensions including “M”, ”F”, “A”, and “C” extensions

1. “M” Standard Extension

2. “F”, ”D”, ”Q” Standard Extension
3. “A” Standard Extension
4. “C” Standard Extension

3.2 Standard Extensions

3.2.1 “M” Standard Extension



MUL performs an *XLEN-bit* × *XLEN-bit* multiplication of rs_1 *by* rs_2, then places lower *XLEN-bit* of the product into destination register *rd*.

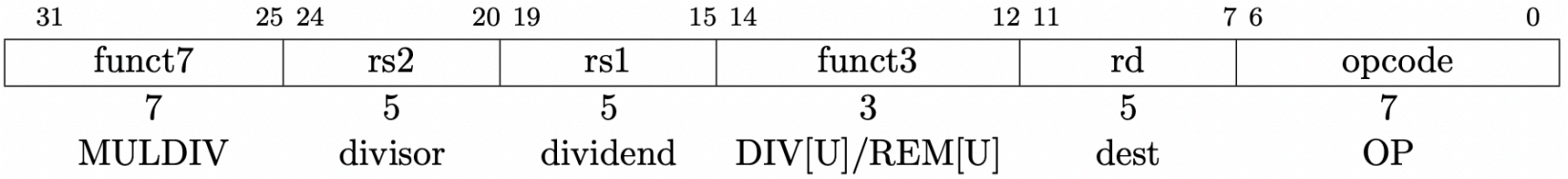
$$rd = lower(rs_1 \times rs_2)$$

MULH performs the similar computation but places upper *XLEN-bit* of the product into destination register *rd*.

$$rd = upper(rs_1 \times rs_2)$$

3.2 Standard Extensions

3.2.1 “M” Standard Extension



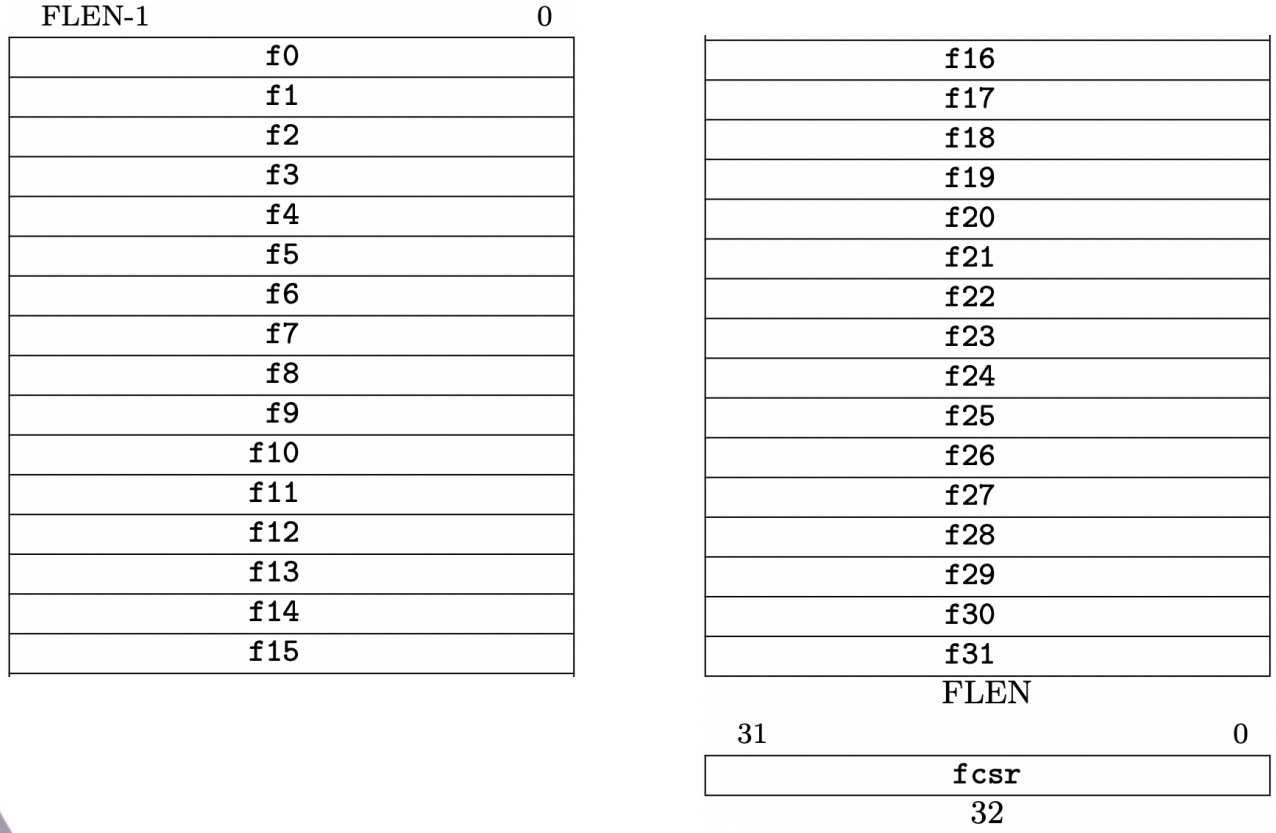
DIV performs an *XLEN-bit* by *XLEN-bit* signed integer division of rs_1 *by* rs_2, then places the quotient into destination register *rd*.

REM performs the similar computation, then places remainder into destination register *rd*.

3.2 Standard Extensions

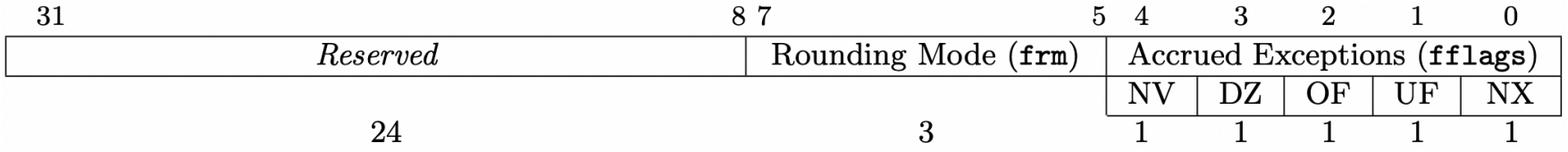
3.2.2 “F”, “D”, “Q” Standard Extension

“F”, ”D”, ”Q” represent the single-precision, double-precision, and quad-precision respectively.



3.2 Standard Extensions

3.2.2 “F”, “D”, “Q” Standard Extension



Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101		*Invalid. Reserved for future use.*
110		*Invalid. Reserved for future use.*
111	DYN	In instruction’s *rm* field, selects dynamic rounding mode; In Rounding Mode register, *Invalid.*

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

3.2 Standard Extensions

3.2.2 “F”, “D”, “Q” Standard Extension

Floating-point operations can use either static rounding mode or dynamic rounding mode:

- **Static rounding mode**:

Select the *rm* field from 000 *to* 100. The specific instruction utilises the static rounding mode corresponding to *rm* field.

- **Dynamic rounding mode**:

Select the *rm* field as 111. The specific instruction utilises the dynamic rounding mode corresponding to *frm* field in *fcsr* status register.

3.2 Standard Extensions

3.2.2 “F”, “D”, “Q” Standard Extension

The *fcsr* can be read, written or copied by the CSR instructions:

FRCSR reads *fcsr* into integer register *rd.*

$$rd = fcsr$$

FSCSR swaps *fcsr* by coping into integer register *rd,* then writing a value from integer register rs_1 into *fcsr.*

$$rd = fcsr$$

$$fcsr = rs_1$$

The field within *fcsr* can be accessed individually through CSR addresses:

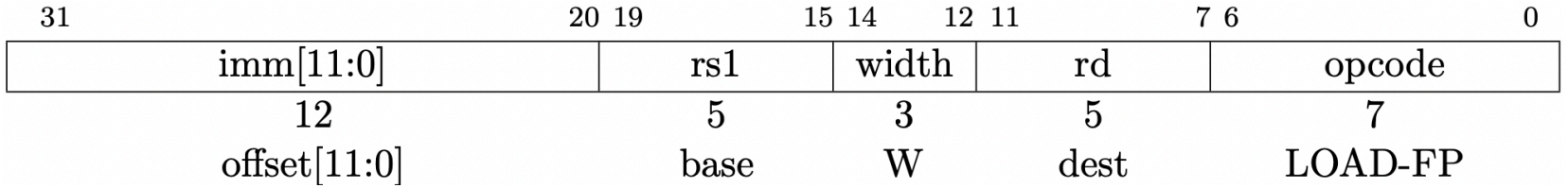
FRRM reads *frm* field and copies it into 3 LSBs of integer register *rd* with zeros in other bits.

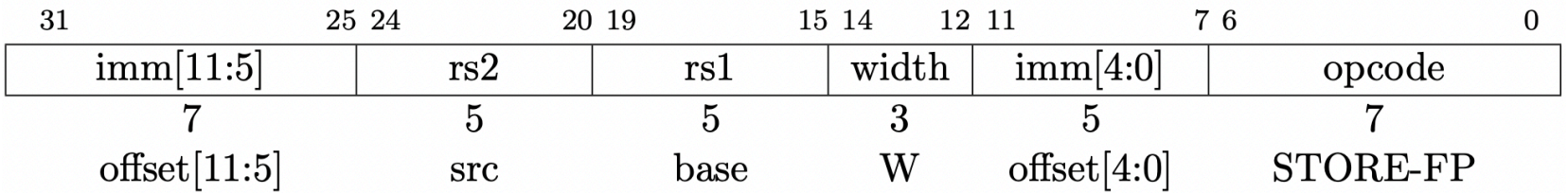
$$rd = zero\,.\,ext(frm)$$

FSRM swaps *frm* by coping into integer register *rd,* then writing a value from the 3 LSBs of integer register rs_1 into *frm* with zeros in other bits.

$$rd = frm$$
$$frm = rs_1[2:0]$$

Load and Store Instruction





FLW loads a single-precision floating-point value from memory into floating-point register *rd*.

$$rd = mem(rs_1 + offset)$$

FSW stores a single-precision floating-point value from floating-point register rs_2 into memory.

$$mem(rs_1 + offset) = rs_2$$

Computational Instructions

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	S	src2	src1	RM	dest	OP-FP
FMUL/FDIV	S	src2	src1	RM	dest	OP-FP
FSQRT	S	0	src	RM	dest	OP-FP
FMIN-MAX	S	src2	src1	MIN/MAX	dest	OP-FP

FADD/FMUL performs addition/multiplication between rs_1 *and* rs_2, then write the result into register *rd*.

$$rd = rs_1 \odot rs_2$$

FSUB/FDIV performs subtraction/division of rs_1 *by* rs_2, then write the result into register *rd*.

$$rd = rs_1 \odot rs_2$$

Computational Instructions

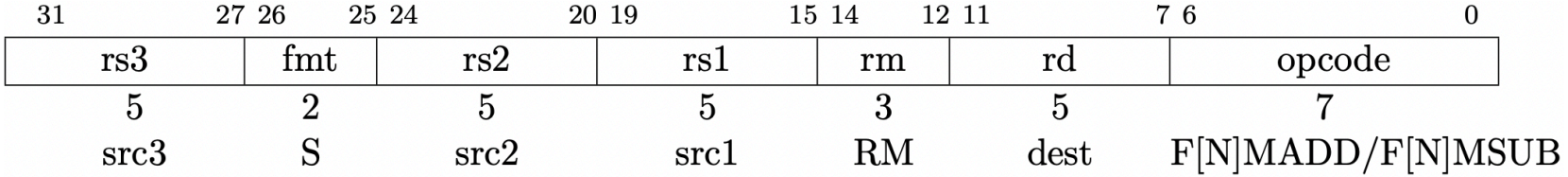
31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	S	src2	src1	RM	dest	OP-FP
FMUL/FDIV	S	src2	src1	RM	dest	OP-FP
FSQRT	S	0	src	RM	dest	OP-FP
FMIN-MAX	S	src2	src1	MIN/MAX	dest	OP-FP

FSQRT computes the square root of rs_1, then write the result into register *rd*.

FMIN/FMAX write the minimum/maximum of rs_1 and rs_2 into register *rd*.

* The value -0.0 is considered to be less than $+0.0$. If the both inputs are NaN, the result is canonical NaN.

Computational Instructions



Floating-point fused multiply-add instructions requires a new standard instruction format. A R4-type specifies three source registers, rs_1, rs_2, rs_3, and a destination register *rd*.

FMADD/FMSUB multiplies the value in rs_1 *and* rs_2, adds/subtracts the value in rs_3, then write the result into register *rd*.

$$rd = (rs_1 \times rs_2) \pm rs_3$$

FNMADD/FNMSUB multiplies the value in rs_1 *and* rs_2, negates the product, subtracts/adds the value in rs_3, then writes the result into register *rd*.

$$rd = -(rs_1 \times rs_2) \mp rs_3$$

3.2 Standard Extensions

3.2.3 “A” Standard Extension

The memory consistency model in RISC-V architecture uses a model called RVWMO which uses a type of weak memory ordering model. To restrict memory ordering and maintain memory consistency, atomic extension is supported.

RISC-V ISA supports three types of memory ordering instruction,

- **FENCE**:

Simply restrict memory ordering by blocking memory operations as predecessor set and successor set. FENCE is already supported in base integer instruction.

3.2 Standard Extensions

3.2.3 “A” Standard Extension

The memory consistency model in RISC-V architecture uses a model called RVWMO which uses a type of weak memory ordering model. To restrict memory ordering and maintain memory consistency, atomic extension is supported.

RISC-V ISA supports three types of memory ordering instruction,

- **Dynamic rounding mode**:

Perform read-modify-write atomic operation, the critical operations between acquire and release can be locked in order.

- **Load-reserved / Store-conditional:**

Reserved an address space for its thread, this reservation can be checked whether another store intervenes it.

Acquire-Release Semantic Operations

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
LR.W/D	ordering		0	addr	width	dest	AMO
SC.W/D	ordering		src	addr	width	dest	AMO

- **Load-reserved (LR):**

Load data from the address rs_1, place the sign-extended into *rd*, and register a reservation set.

$$rd = sign\,.\,ext(mem(rs_1))$$

$$register\ reservation\ set\ \phi$$

Acquire-Release Semantic Operations

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
LR.W/D	ordering		0	addr	width	dest	AMO
SC.W/D	ordering		src	addr	width	dest	AMO

- **Store-conditional *(SC):***

Write data from the source address rs_2 to the address rs_1.

1. If SC succeeds: write data in rs_2 into memory, then write zero into *rd*.
2. If SC fails: do not write any data into memory, then write nonzero into *rd*.

$$if\ succeeds:\ mem(rs_1) = rs_2,\ rd = zero$$

$$if\ fails:\ rd = nonzero$$

Acquire-Release Semantic Operations

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
LR.W/D	ordering		0	addr	width	dest	AMO
SC.W/D	ordering		src	addr	width	dest	AMO

- **Acquire access*(aq)*:**

Treat AMO instructions as acquire access, and the following memory operations can’t be observed before the acquire operation takes place.

- **Release access*(rl):***

Treat the AMO instructions as release access; the release memory operations can’t be observed before the earlier memory operation takes place.

Acquire-Release Semantic Operations

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
LR.W/D	ordering		0	addr	width	dest	AMO
SC.W/D	ordering		src	addr	width	dest	AMO

- **Sequentially Consistent:**

When an instruction with acquire access and release access is asserted, such an instruction is Sequentially Consistent.

Atomic Memory Operations (AMOs)

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
AMOSWAP.W/D	ordering		src	addr	width	dest	AMO
AMOADD.W/D	ordering		src	addr	width	dest	AMO
AMOAND.W/D	ordering		src	addr	width	dest	AMO
AMOOR.W/D	ordering		src	addr	width	dest	AMO
AMOXOR.W/D	ordering		src	addr	width	dest	AMO
AMOMAX[U].W/D	ordering		src	addr	width	dest	AMO
AMOMIN[U].W/D	ordering		src	addr	width	dest	AMO

Load data from the rs_1, place the value into rd, then apply a binary operator to the loaded value and the value in rs_2. Store the result back to the address rs_1.

$$rd = mem(rs_1)$$
$$mem(rs_1) = mem(rs_1) \odot rs_2$$

3.2 Standard Extensions

3.2.4 “C” Standard Extension

Compression extension is used to reduce code size by adding only 16-bit instructions for common operations.

The compression scheme offers the shorter 16-bit version of common instructions when any following condition is met.

- The immediate or offset field is small.
- One of the registers is a zero register $x0$, a link register $x1$, or a stack register $x2$.
- The destination register and the first source register are identical.
- The registers are the most popular used, i.e. $x8$ to $x15$.

3.2 Standard Extensions

3.2.4 “C” Standard Extension

Compressed Instruction Formats

Format	Meaning	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd′	op
CL	Load	funct3	imm		rs1′	imm	rd′	op
CS	Store	funct3	imm		rs1′	imm	rs2′	op
CA	Arithmetic	funct6			rd′/rs1′	funct2	rs2′	op
CB	Branch	funct3	offset		rs1′	offset		op
CJ	Jump	funct3	jump target					op

RVC Register Number	000	001	010	011	100	101	110	111
Integer Register Number	x8	x9	x10	x11	x12	x13	x14	x15
Integer Register ABI Name	s0	s1	a0	a1	a2	a3	a4	a5
Floating-Point Register Number	f8	f9	f10	f11	f12	f13	f14	f15
Floating-Point Register ABI Name	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

Load and Store Instructions

Stack-Pointer-Based Loads and Stores

CI format

15 13	12	11 7	6 2	1 0
funct3	imm	rd	imm	op
3	1	5	5	2
C.LWSP	offset[5]	dest≠0	offset[4:2|7:6]	C2
C.LDSP	offset[5]	dest≠0	offset[4:3|8:6]	C2
C.LQSP	offset[5]	dest≠0	offset[4|9:6]	C2
C.FLWSP	offset[5]	dest	offset[4:2|7:6]	C2
C.FLDSP	offset[5]	dest	offset[4:3|8:6]	C2

$$rd = mem(x2\_scale + offset)$$

Load and Store Instructions

Stack-Pointer-Based Loads and Stores

CSS format

15 13	12 7	6 2	1 0
funct3	imm	rs2	op
3	6	5	2
C.SWSP	offset[5:2|7:6]	src	C2
C.SDSP	offset[5:3|8:6]	src	C2
C.SQSP	offset[5:4|9:6]	src	C2
C.FSWSP	offset[5:2|7:6]	src	C2
C.FSDSP	offset[5:3|8:6]	src	C2

$$mem(x2\_scaled + offset) = rs_2$$

Load and Store Instructions

Register-Based Loads and Stores

CL format

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1′	imm	rd′	op
3	3	3	2	3	2
C.LW	offset[5:3]	base	offset[2|6]	dest	C0
C.LD	offset[5:3]	base	offset[7:6]	dest	C0
C.LQ	offset[5|4|8]	base	offset[7:6]	dest	C0
C.FLW	offset[5:3]	base	offset[2|6]	dest	C0
C.FLD	offset[5:3]	base	offset[7:6]	dest	C0

$$rd = mem(rs'_1\_scaled + offset)$$

Load and Store Instructions

Register-Based Loads and Stores

CS format

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1′	imm	rs2′	op
3	3	3	2	3	2
C.SW	offset[5:3]	base	offset[2|6]	src	C0
C.SD	offset[5:3]	base	offset[7:6]	src	C0
C.SQ	offset[5|4|8]	base	offset[7:6]	src	C0
C.FSW	offset[5:3]	base	offset[2|6]	src	C0
C.FSD	offset[5:3]	base	offset[7:6]	src	C0

$$mem(rs'_1\_scaled + offset) = rs'_2$$

Control Transfer Instructions

CJ format

15	13 12	2 1	0
funct3	imm		op
3	11		2
C.J	offset[11|4|9:8|10|6|7|3:1|5]		C1
C.JAL	offset[11|4|9:8|10|6|7|3:1|5]		C1

C.J and **C.JAL** perform a transfer to the target address that is formed as the immediate in multiple of $2\,Byte$, i.e. a $\pm 2\,KB$ range.

CB format

15	13 12	10 9	7 6	2 1	0
funct3	imm	rs1′	imm		op
3	3	3	5		2
C.BEQZ	offset[8|4:3]	src	offset[7:6|2:1|5]		C1
C.BNEZ	offset[8|4:3]	src	offset[7:6|2:1|5]		C1

C.BNQZ and **C.BNEZ** perform a conditional transfer to the target address that is formed by *offset*, i.e. a $\pm 256\,B$ range.

Integer Computational Instruction

CI format

15 13	12	11 7	6 2	1 0
funct3	imm[5]	rd/rs1	imm[4:0]	op
3	1	5	5	2
C.ADDI	nzimm[5]	dest≠0	nzimm[4:0]	C1
C.ADDIW	imm[5]	dest≠0	imm[4:0]	C1
C.ADDI16SP	nzimm[9]	2	nzimm[4|6|8:7|5]	C1

CA format

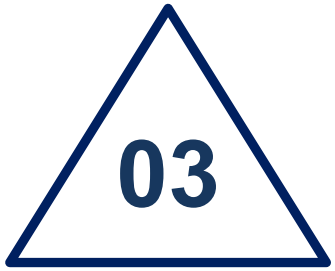
15 10	9 7	6 5	4 2	1 0
funct6	rd′/rs1′	funct2	rs2′	op
6	3	2	3	2
C.AND	dest	C.AND	src	C1
C.OR	dest	C.OR	src	C1
C.XOR	dest	C.XOR	src	C1
C.SUB	dest	C.SUB	src	C1
C.ADDW	dest	C.ADDW	src	C1
C.SUBW	dest	C.SUBW	src	C1

Integer Computational Instruction

CI format

15	13	12	11	7	6	2	1	0
funct3		imm[5]	rd		imm[4:0]		op	
3		1	5		5		2	
C.LI		imm[5]	dest≠0		imm[4:0]		C1	
C.LUI		nzimm[17]	dest≠{0, 2}		nzimm[16:12]		C1	

C.LI and **C.LUI** perform a constant value according to sign-extended immediate, store it into *rd*.



Extension Not Covered

Brief introduce other extensions.

3.2 Extension Not Covered

Standard Extension	Description
L	Decimal floating-point extension
B	Bit manipulation extension
T	Transactional memory extension
P	Packed-SIMD extension
V	Vector operation extension
N	User-level interrupt extension

References

[1] (2020) The RISC-V Instruction Set Manual, Volume I - Unprivileged ISA.

[2] (2020) The RISC-V Instruction Set Manual, Volume II - Privileged ISA.

[3] (2018) RISCV - An Overview of the Instruction Set Architecture.

THANKS
for your
ATTENTION