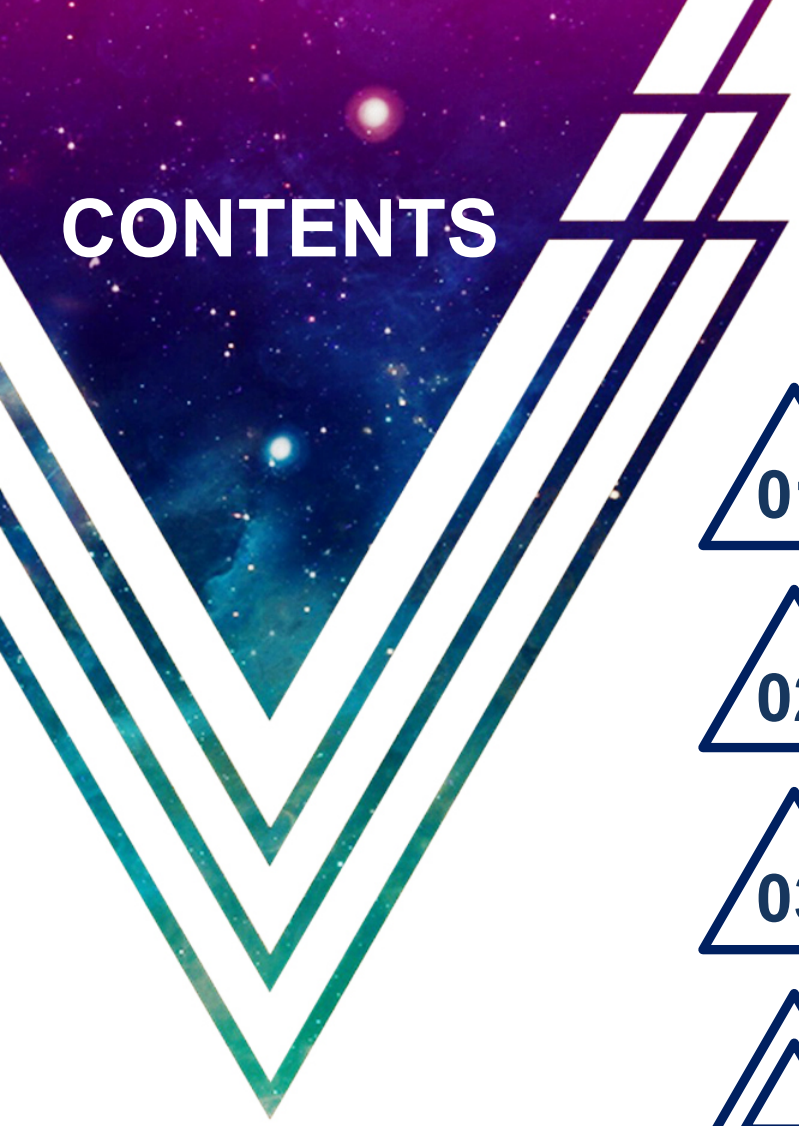
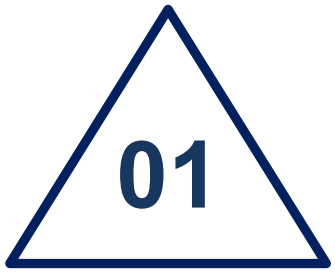
RISC-V ISA

Chapter 2

Base Integer Instruction Set



CONTENTS



Introduction

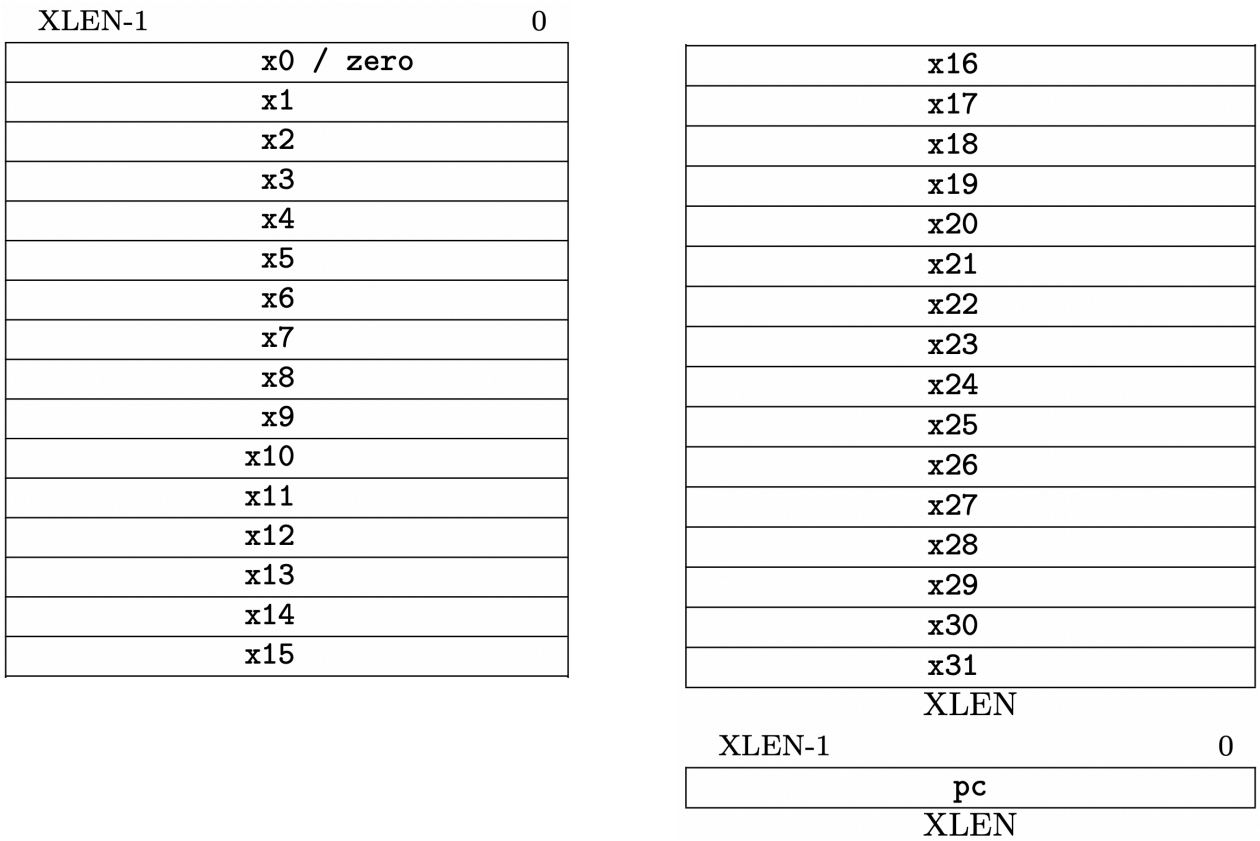
General purpose registers, and encoding formats

1. Registers

2. Formats and Encoding Variants

2.1 Introduction

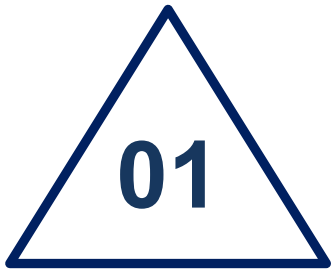
2.1.1 Registers



2.1 Introduction

2.1.1 Registers

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller



Introduction

General purpose registers, and encoding formats

1. Registers

2. Formats and Encoding Variants

2.1 Introduction

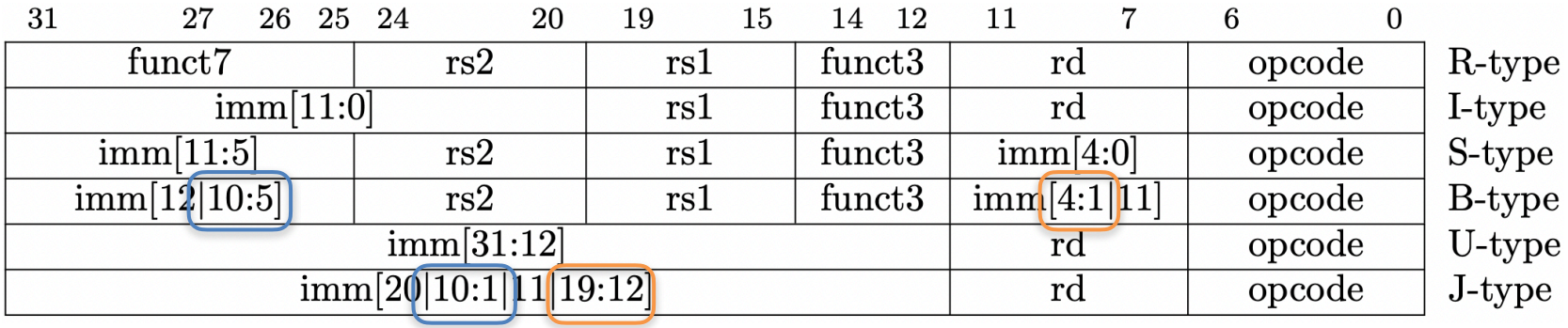
2.1.2 Formates and Encoding Variants

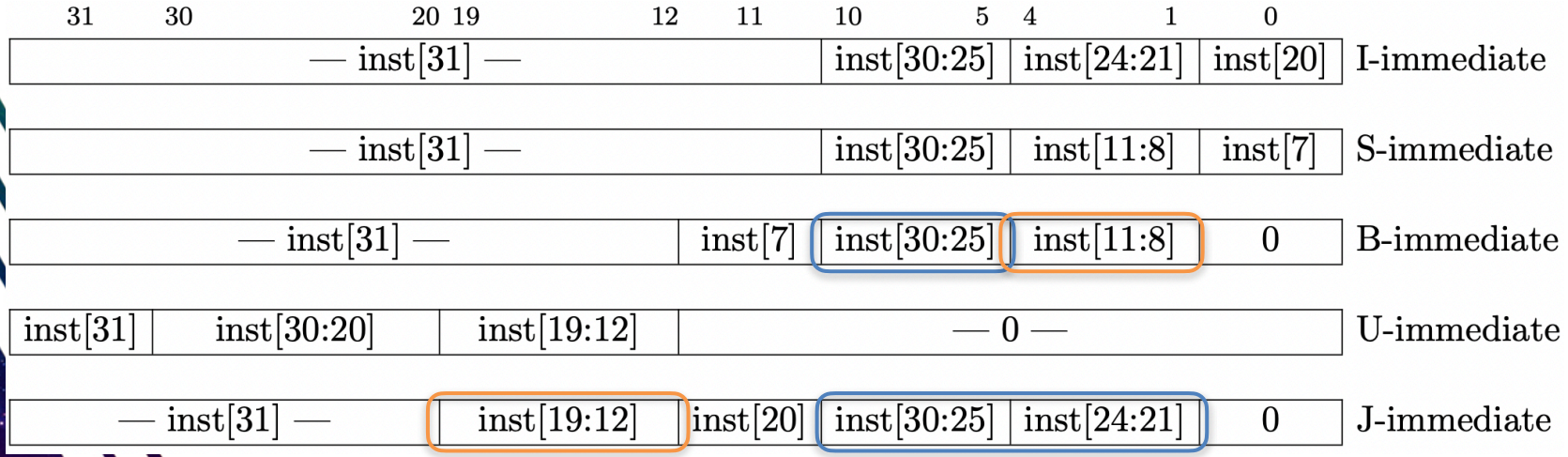
31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

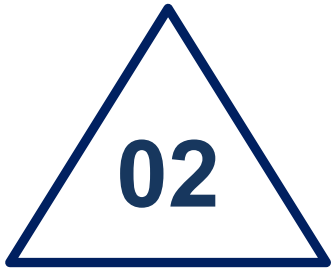
RISC-V ISA keeps the source registers (rs_1, rs_2) and destination register (rd) at the same position to simplify decoding.

2.1 Introduction

2.1.2 Formates and Encoding Variants







Instructions

Base-integer instruction including from operations to hints

1. Integer Register Operations

2. Control Transfer Instructions

3. Load and Store Instructions

4. Memory Ordering Instructions

5. Environment Call and Breakpoints

6. Hint Instructions

2.2 Instructions

2.2.1 Integer Register Operations

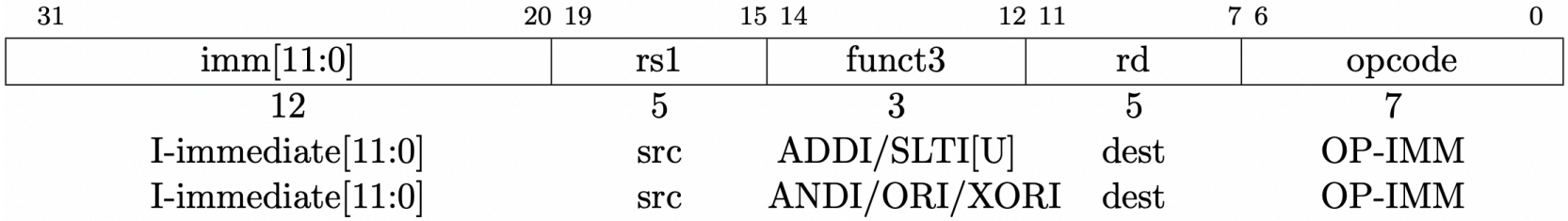
31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

RV32I defines several arithmetic R-type operations. Read the value in registers rs_1 and rs_2 as sources and write the arithmetic result into register rd.

$$rd = rs_1 \odot rs_2$$

2.2 Instructions

2.2.1 Integer Register Operations



31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

Similar to R-type arithmetic instructions, but I-type instructions read the *immediate* value and rs_1 as source then write the result into register *rd*.

$$rd = rs_1 \odot imm$$

2.2 Instructions

2.2.1 Integer Register Operations

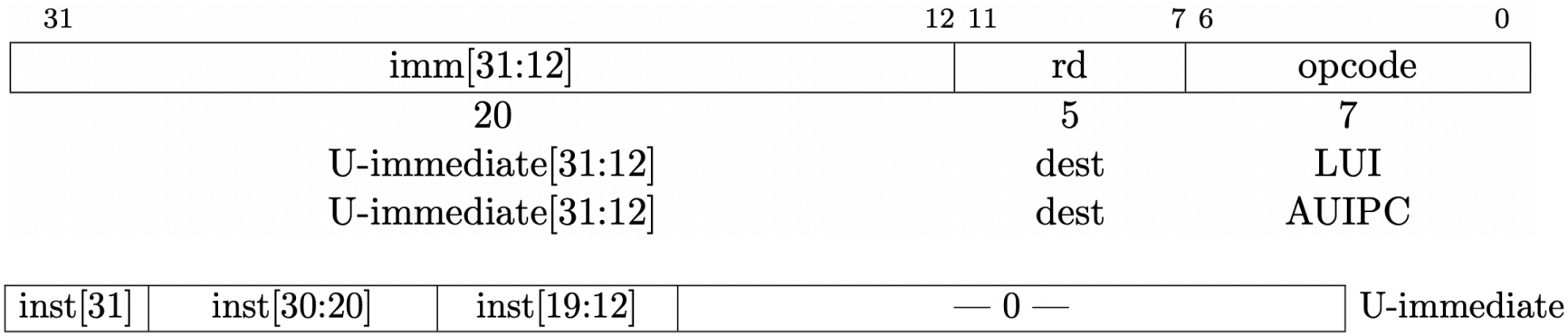
31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

Shift instructions are encoded as I-type format. Read the value in register rs_1 as source and shift number of bits as the value in shift amount *shamt* field, then write the result into register *rd*.

$$rd = rs_1 \odot shamt$$

2.2 Instructions

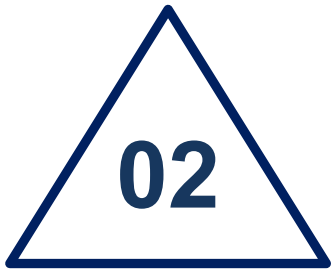
2.2.1 Integer Register Operations



LUI (Load upper immediate) and **AUIPC** (Add upper immediate to PC) are encoded as U-type instructions. They both read 20-bit immediate field and fill the lowest 12 bits with zeros, then write into register *rd*.

- **LUI**:
  Used to build 32-bit constant

- **AUIPC**:
  Used to build pc-relative address; it sums U-immediate and current PC, then write in to register *rd*.



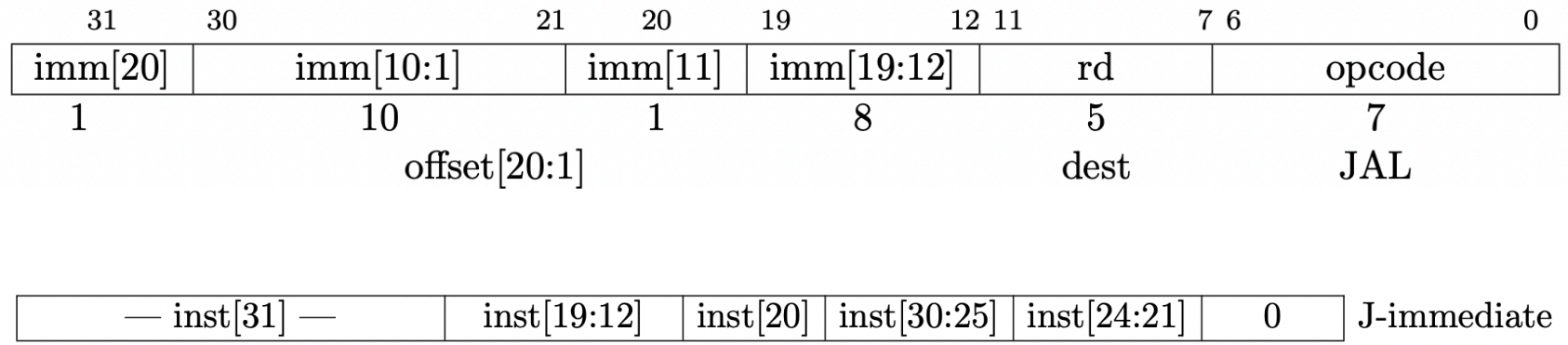
Instructions

Base-integer instruction including from operations to hints

1. Integer Register Operations
2. ***Control Transfer Instructions***
3. Load and Store Instructions
4. Memory Ordering Instructions
5. Environment Call and Breakpoints
6. Hint Instructions

2.2 Instructions

2.2.2 Control Transfer Instructions



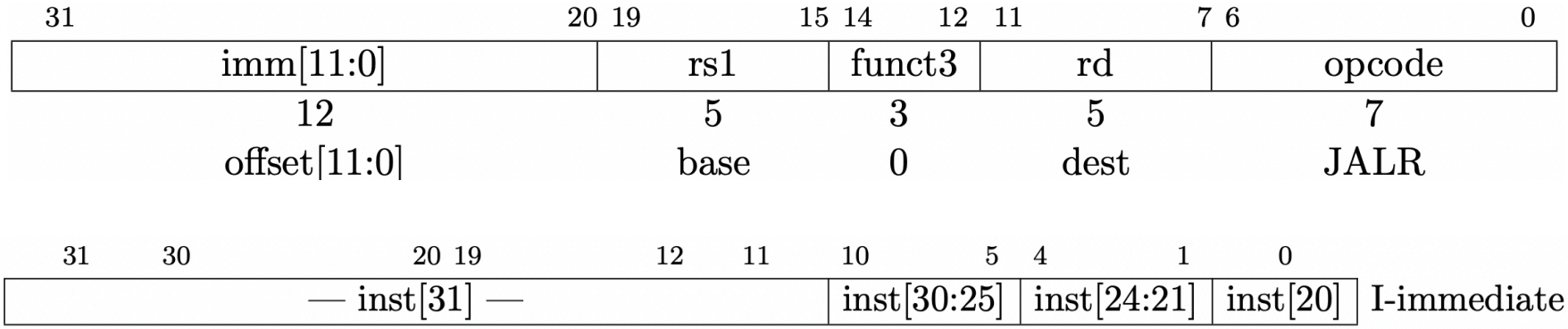
JAL (jump and link) instruction is encoded as J-type. Because of the format of J-immediate, it can target a range of $\pm 1MB$.

$$rd = PC$$
$$PC = PC + imm\_20$$

* The minimal range of J-type instruction is only 2 bytes. SInce base integer restricts 4-byte alignment, a jump with less than 4-byte causes an instruction-address-misaligned exception. Except that compressed (C) extension is supported.

2.2 Instructions

2.2.2 Control Transfer Instructions



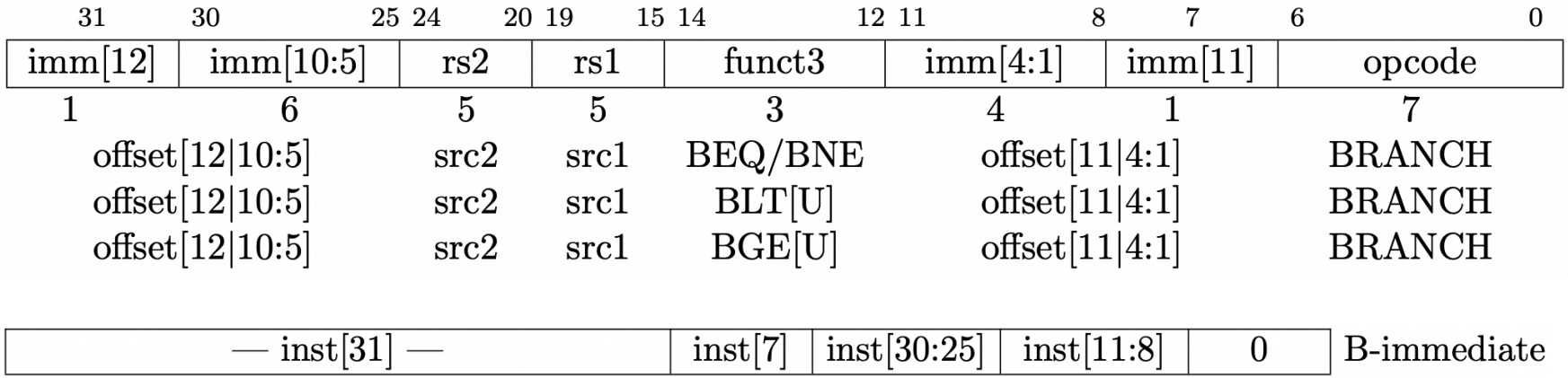
JALR (jump and link register) instruction uses I-type. Unlike J-type instruction, it targets $\pm 2KB$ plus the value in rs_1.

$$rd = PC$$

$$PC = rs_1 + imm\_12$$

2.2 Instructions

2.2.2 Control Transfer Instructions

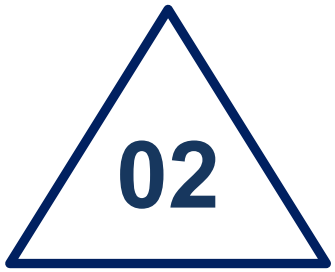


31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

— inst[31] —	inst[7]	inst[30:25]	inst[11:8]	0	B-immediate

Conditional branch typically uses B-type. Compare the value in register rs_1 to the value in register rs_2, if the condition is satisfied then take the branch relatively to *immediate*. As the B-type instruction, the range of conditional branch is $\pm 4KB$.

$$PC = rs_1 + imm\_12$$



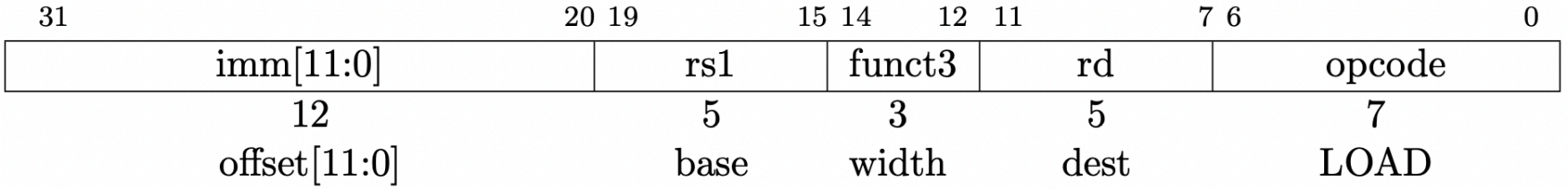
Instructions

Base-integer instruction including from operations to hints

1. Integer Register Operations
2. Control Transfer Instructions
3. ***Load and Store Instructions***
4. Memory Ordering Instructions
5. Environment Call and Breakpoints
6. Hint Instructions

2.2 Instructions

2.2.3 Load and Store Instructions



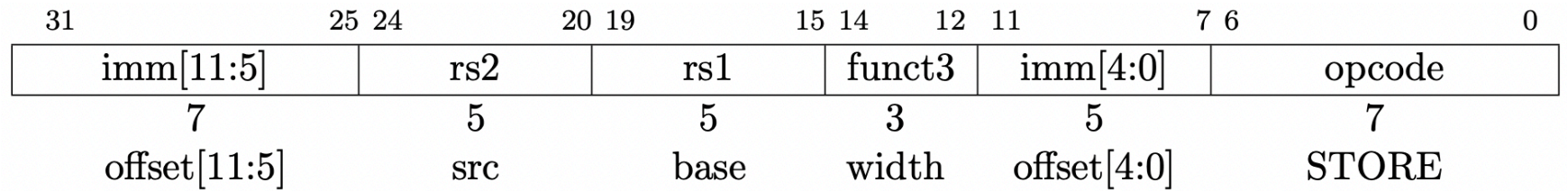
Loads are encoded as I-type, load a data from memory with the address in register rs_1 plus offset into destination register *rd*.

$$rd = mem[rs_1 + offset]$$

* Loads with a destination of x0 must raise an exception.

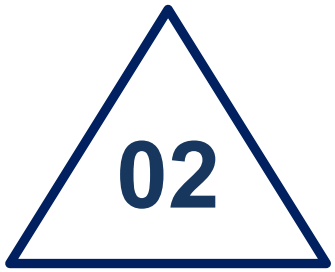
2.2 Instructions

2.2.3 Load and Store Instructions



Stores are encoded as S-type, store a data in register rs_2 into the memory with address in rs_1 plus offset.

$$mem[rs_1 + offset] = rs_2$$



Instructions

Base-integer instruction including from operations to hints

1. Integer Register Operations

2. Control Transfer Instructions

3. Load and Store Instructions

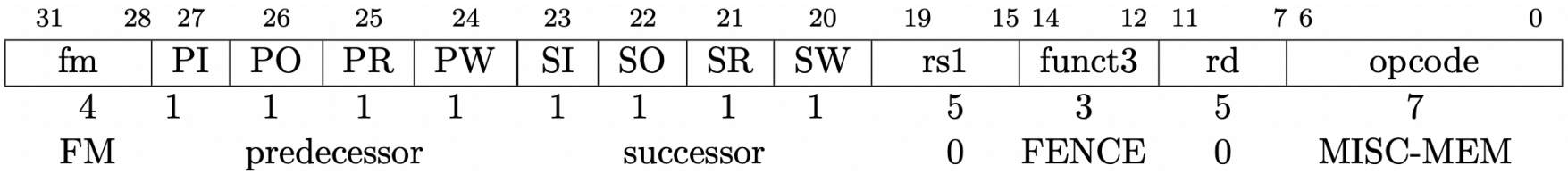
4. Memory Ordering Instructions

5. Environment Call and Breakpoints

6. Hint Instructions

2.2 Instructions

2.2.4 Memory Ordering Instructions



31 28	27	26	25	24	23	22	21	20	19 15	14 12	11 7	6 0
fm	PI	PO	PR	PW	SI	SO	SR	SW	rs1	funct3	rd	opcode
4	1	1	1	1	1	1	1	1	5	3	5	7
FM	predecessor				successor				0	FENCE	0	MISC-MEM

FENCE instruction is used to order device I/O accesses and memory accesses from multithreads coprocessors.

It uses several fields to indicate the corresponding attributes.

P: predecessor I: device input R: memory read

S: successor O: device output W: memory write

2.2 Instructions

2.2.4 Memory Ordering Instructions

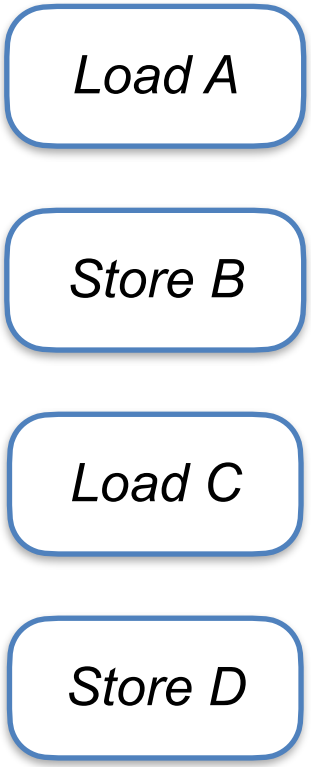
fm field	Mnemonic	Meaning
0000	*none*	Normal Fence
1000	TSO	With FENCE RW,RW: exclude write-to-read ordering Otherwise: *Reserved for future use.*
other		*Reserved for future use.*

- Normal **FENCE** orders all memory operations in predecessor set before in successor set.
- **FENCE.TSO** is weaker than normal fence instruction. This instruction orders all load operations in predecessor set before other memory operations in successor set. And, it orders all store operations in predecessor set before all store operations in successor set.

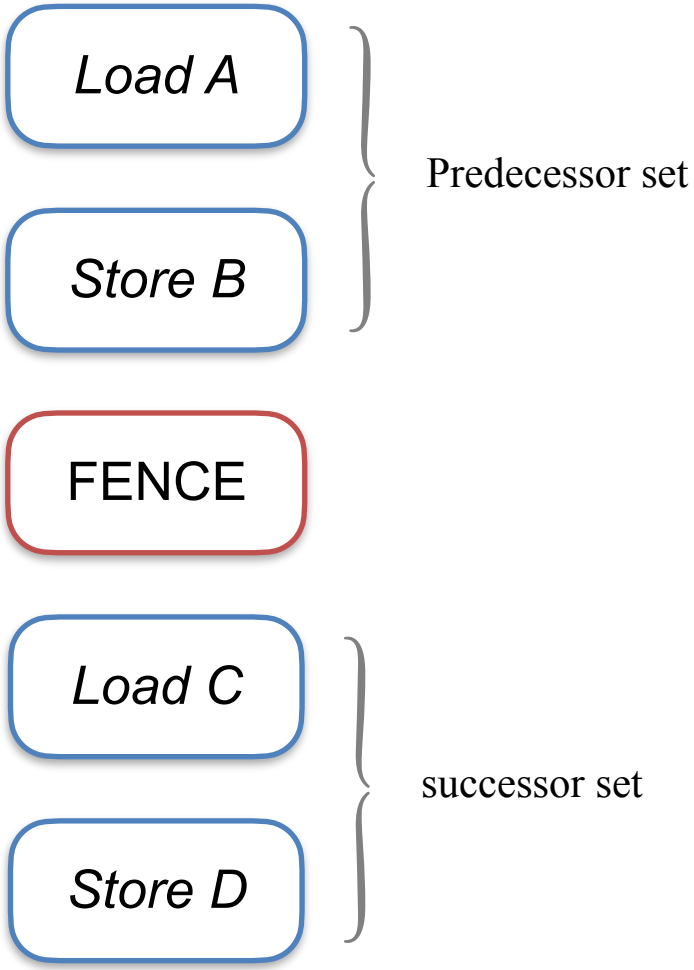
2.2 Instructions

2.2.4 Memory Ordering Instructions

Example:



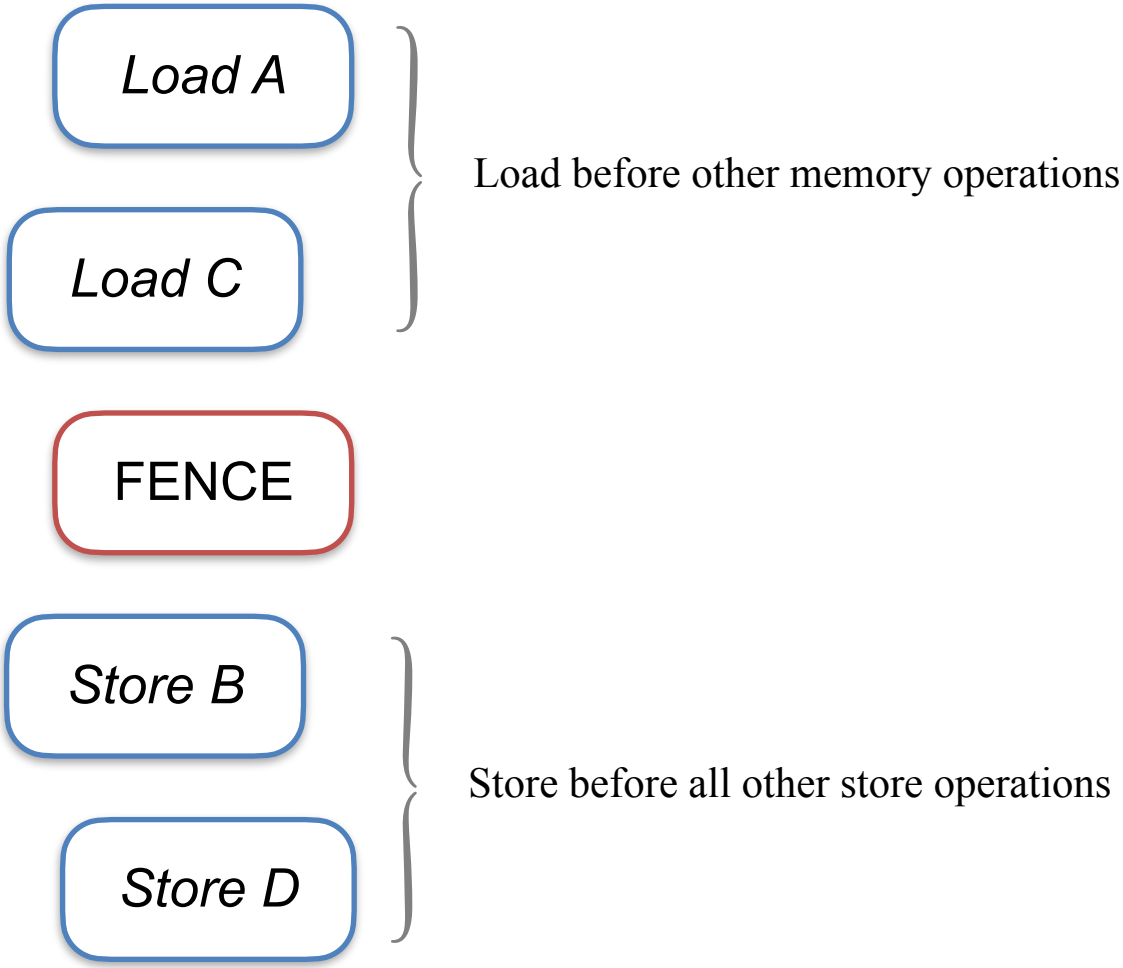
Example: **FENCE**

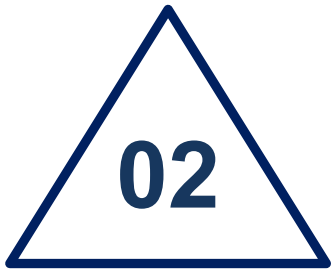


2.2 Instructions

2.2.4 Memory Ordering Instructions

Example: **FENCE.TSO**





Instructions

Base-integer instruction including from operations to hints

1. Integer Register Operations
2. Control Transfer Instructions
3. Load and Store Instructions
4. Memory Ordering Instructions
5. ***Environment Call and Breakpoints***
6. Hint Instructions

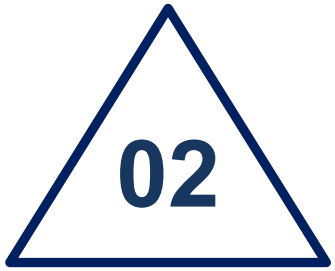
2.2 Instructions

2.2.5 Environment Call and Breakpoints

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

ECALL is used to make a service request to the exception environment. EEI will define how parameters are passed.

EBREAK is used to return to debugging environment.



Instructions

Base-integer instruction including from operations to hints

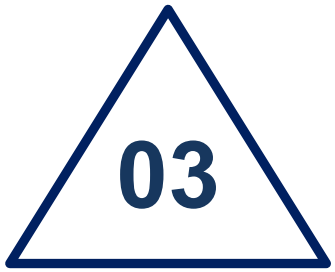
1. Integer Register Operations
2. Control Transfer Instructions
3. Load and Store Instructions
4. Memory Ordering Instructions
5. Environment Call and Breakpoints
6. ***Hint Instructions***

2.2 Instructions

2.2.6 Hint instructions

Instruction	Constraints	Code Points	Purpose
LUI	*rd*=x0	2^{20}	*Reserved for future standard use*
AUIPC	*rd*=x0	2^{20}	
ADDI	*rd*=x0, and either *rs1*≠x0 or *imm*≠0	$2^{17}-1$	
ANDI	*rd*=x0	2^{17}	
ORI	*rd*=x0	2^{17}	
XORI	*rd*=x0	2^{17}	
ADD	*rd*=x0	2^{10}	
SUB	*rd*=x0	2^{10}	
AND	*rd*=x0	2^{10}	
OR	*rd*=x0	2^{10}	
XOR	*rd*=x0	2^{10}	
SLL	*rd*=x0	2^{10}	
SRL	*rd*=x0	2^{10}	
SRA	*rd*=x0	2^{10}	
FENCE	*pred*=0 or *succ*=0	$2^{5}-1$	
SLTI	*rd*=x0	2^{17}	*Reserved for custom use*
SLTIU	*rd*=x0	2^{17}	
SLLI	*rd*=x0	2^{10}	
SRLI	*rd*=x0	2^{10}	
SRAI	*rd*=x0	2^{10}	
SLT	*rd*=x0	2^{10}	
SLTU	*rd*=x0	2^{10}	

Hint instructions are used to indicate performance hints, it doesn't change any visible state except for performance counter.



Variants

Other base-integer variants

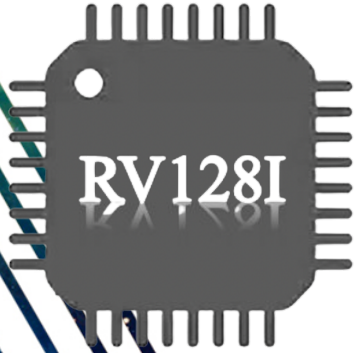
2.3 Variants

RV32E is a reduced version of RV32I, where the only difference is the number of integer registers.

XLEN-1	0
x0 / zero	
x1	
x2	
x3	
x4	
x5	
x6	
x7	
x8	
x9	
x10	
x11	
x12	
x13	
x14	
x15	

2.3 Variants

RV64I widen integer registers to 64-bit, and supports address space to 64 bits ($XLEN = 64$)



RV128I widen integer registers to 128-bit, and supports address space to 64 bits ($XLEN = 128$)

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