System I

Instruction Set Architecture

Bo Feng, Lei Wu, Rui Chang

Zhejiang University

Disclaimer

- **Many images and resources used in this lecture are collected from the Internet, and they are used only for the educational purpose. The copyright belong to the original owners, respectively.**

- **Part of slides credit to**
  - **David A. Patterson and John L. Hennessy. Computer Organization and Design RISC-V Edition: The Hardware Software Interface, 1st Edition.**
  - **John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach, 6th Edition.**
  - **Andrew Waterman and David A. Patterson. The RISC-V Reader: An Open Architecture Atlas.**
  - **CSCE 513, Prof. Yonghong Yan @ University of North Carolina at Charlotte**
  - **CENG3420, Bei Yu @ The Chinese University of Hong Kong**
  - **CS 3410, Prof. Hakim Weatherspoon @ Cornell University**
  - **CS 162, Prof. Sam Kumar @ UC Berkeley**
  - **CSc 453, Prof. Saumya Debray @ University of Arizona**

Overview

- RISC-V ISA
- RISC-V Assembly Language

Overview

- RISC-V ISA
- RISC-V Assembly Language

What is RISC-V?

- RISC-V (pronounced "risk-five") is an ISA standard
  - An open-source implementation of a reduced instruction set computing (RISC) based instruction set architecture (ISA)
  - There was RISC-I, II, III, IV before
- Most ISAs: X86, ARM, Power, MIPS, SPARC
  - Commercially protected by patents
  - Preventing practical efforts to reproduce the computer systems.
- RISC-V is open
  - Permitting any person or group to construct compatible computers
  - Use associated software
- Originated in 2010 by researchers at UC Berkeley
  - Krste Asanović, David Patterson and students
- ISA Specifications
  - Unprivileged specification version 20191213 (v2.2)
  - Privileged specification version 20211203 (v1.11)
  - More on github: https://github.com/riscv/riscv-isa-manual

https://riscv.org/

Goals in Defining RISC-V

- A completely open ISA that is freely available to academia and industry
- A real ISA suitable for direct native hardware implementation, not just simulation nor binary translation
- An ISA that avoids "over-architecting" for
  - A particular microarchitecture style (e.g., microcoded, in-order, decoupled, out-of-order) or
  - Implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these
- RISC-V ISA includes
  - A small base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and
  - Optional standard extensions, to support general-purpose software development
  - Optional customer extensions
- Support for the revised 2008 IEEE-754 floating-point standard

RISC-V Principles

- Generally kept very simple and extendable
  - Whether short, long, or variable
- Separated into multiple specifications
  - User-level ISA spec (compute instructions)
  - Compressed ISA spec (16-bit instructions)
  - Privileged ISA spec (supervisor-mode instructions)
  - More…
- ISA support is given by RV + word-width + extensions supported
  - E.g., RV32I means 32-bit RISC-V with support for the I (integer) instruction set

User-Level ISA

- Defines the normal instructions needed for computation
  - A mandatory **base integer ISA**
    - **I: Integer instructions**:
      - ALU
      - Branches/jumps
      - Loads/stores
  - **Standard extensions**
    - M: Integer Multiplication and Division
    - A: Atomic Instructions
    - F: Single-Precision Floating-Point
    - D: Double-Precision Floating-Point
    - C: Compressed Instructions (16 bit)
    - **G = IMAFD: integer base + four standard extensions**
  - Optional extensions

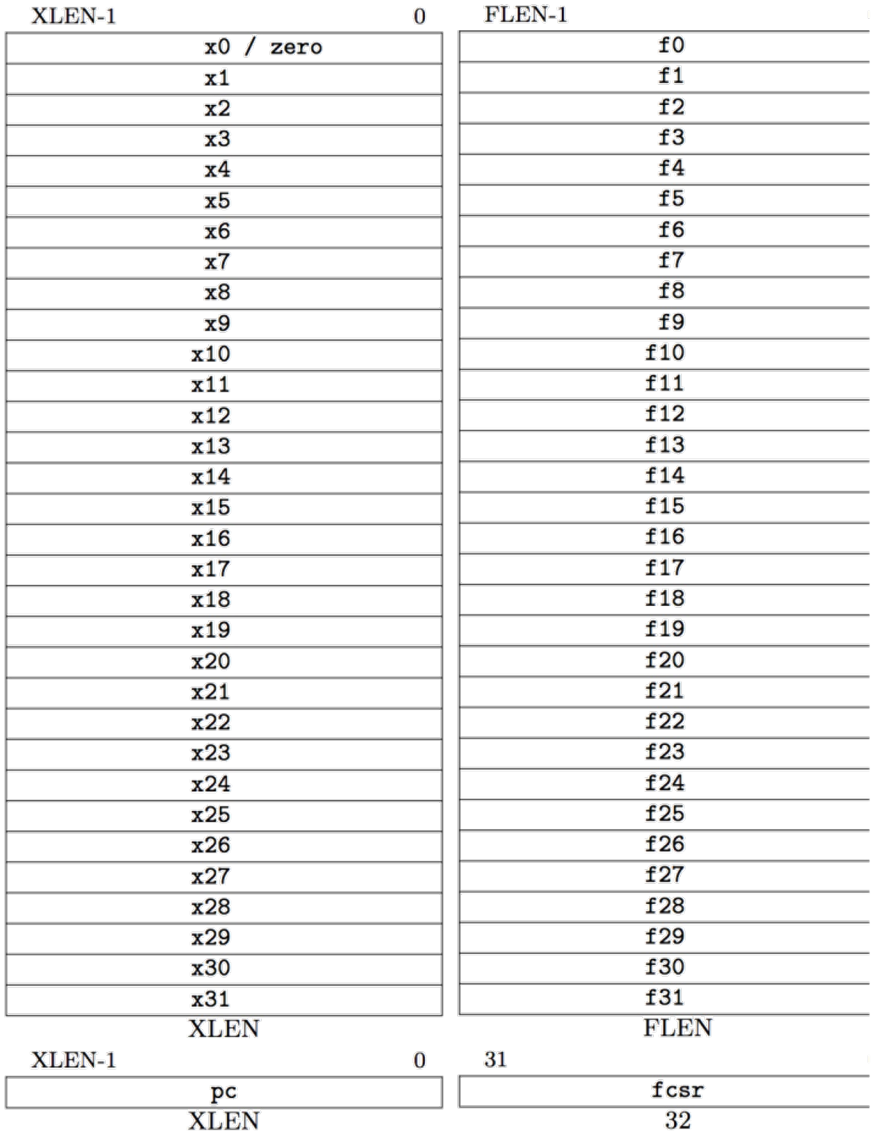
Basic RISC-V ISA

- Both 32-bit and 64-bit address space variants
  - RV32 and RV64
- Easy to subset/extend for education/research
  - RV32IM, RV32IMA, RV32IMAFD, RV32G
- SPEC on the website
  - www.riscv.org

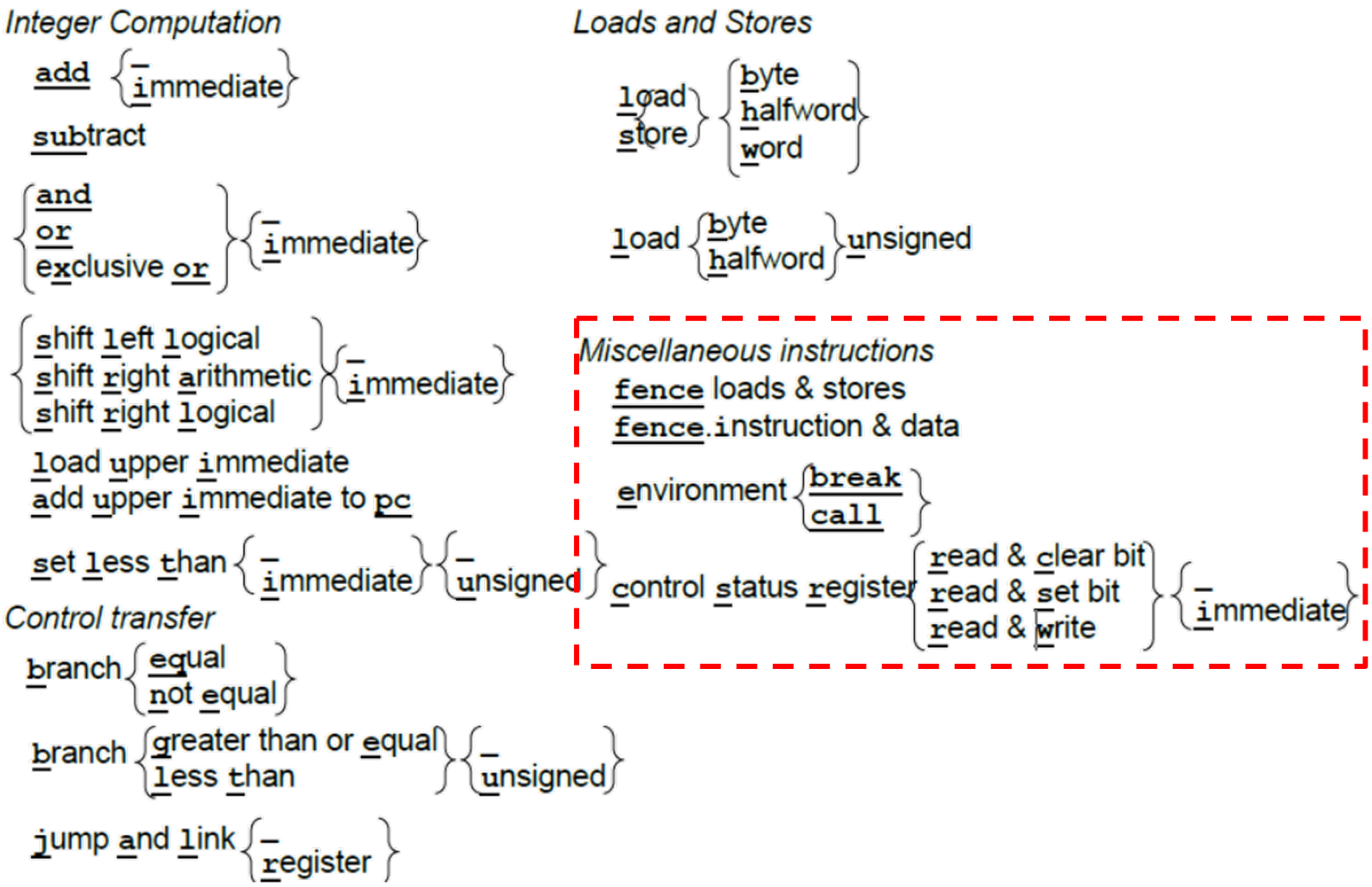
Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing; see Chapter 5
F	Adds single precision (32-bit) IEEE floating point, includes 32 32-bit floating point registers, instructions to load and store those registers and operate on them
D	Extends floating point to double precision, 64-bit, making the registers 64-bits, adding instructions to load, store, and operate on the registers
Q	Further extends floating point to add support for quad precision, adding 128-bit operations
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a compressed version of the instruction set intended for small-memory-sized embedded applications. Defines 16-bit versions of common RV32I instructions
V	A future extension to support vector operations (see Chapter 4)
B	A future extension to support operations on bit fields
T	A future extension to support transactional memory
P	An extension to support packed SIMD instructions: see Chapter 4
RV128I	A future base instruction set providing a 128-bit address space

RISC-V Processor State

- Program counter (**PC**)
- 32 32/64-bit integer registers (**x0-x31**)
  - x0 always contains a 0
  - x1 to hold the return address on a call.
- 32 floating-point (FP) registers (**f0-f31**)
  - Each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)
- FP status register (**fsr**), used for FP rounding mode & exception reporting



RV32I


Integer Computation
add { _ immediate }
subtract
{ and / or / exclusive or } { _ immediate }
{ shift left logical / shift right arithmetic / shift right logical } { _ immediate }
load upper immediate
add upper immediate to pc
set less than { _ immediate } { _ unsigned }
Control transfer
branch { equal / not equal }
branch { greater than or equal / less than } { _ unsigned }
jump and link { _ register }
Loads and Stores
{ load / store } { byte / halfword / word }
load { byte / halfword } unsigned
Miscellaneous instructions
fence loads & stores
fence.instruction & data
environment { break / call }
control status register { read & clear bit / read & set bit / read & write } { _ immediate }

ALU Instructions

Example instrucmtion	Instruction name	Meaning
`add x1,x2,x3`	Add	`Regs[x1]←Regs[x2]+Regs[x3]`
`addi x1,x2,3`	Add immediate unsigned	`Regs[x1]←Regs[x2]+3`
`lui x1,42`	Load upper immediate	`Regs[x1]←`0^{32}`##42##`0^{12}
`sll x1,x2,5`	Shift left logical	`Regs[x1]←Regs[x2]<<5`
`slt x1,x2,x3`	Set less than	`if (Regs[x2]<Regs[x3]) Regs[x1]←1 else Regs[x1]←0`

Figure A.26 The basic ALU instructions in RISC-V are available both with register-register operands and with one immediate operand. LUI uses the U-format that employs the rs1 field as part of the immediate, yielding a 20-bit immediate.

Load/Store Instructions

Example instruction	Instruction name	Meaning
ld x1,80(x2)	Load doubleword	Regs[x1] ← Mem[80+Regs[x2]]
lw x1,60(x2)	Load word	Regs[x1] $\leftarrow_{64}$ Mem[60+Regs[x2]]$_0$)^{32} ## Mem[60+Regs[x2]]
lwu x1,60(x2)	Load word unsigned	Regs[x1] $\leftarrow_{64}$ 0^{32} ## Mem[60+Regs[x2]]
lb x1,40(x3)	Load byte	Regs[x1] $\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{56} ## Mem[40+Regs[x3]]
lbu x1,40(x3)	Load byte unsigned	Regs[x1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[x3]]
lh x1,40(x3)	Load half word	Regs[x1] $\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{48} ## Mem[40+Regs[x3]]
flw f0,50(x3)	Load FP single	Regs[f0] $\leftarrow_{64}$ Mem[50+Regs[x3]] ## 0^{32}
fld f0,50(x2)	Load FP double	Regs[f0] $\leftarrow_{64}$ Mem[50+Regs[x2]]
sd x2,400(x3)	Store double	Mem[400+Regs[x3]] $\leftarrow_{64}$ Regs[x2]
sw x3,500(x4)	Store word	Mem[500+Regs[x4]] $\leftarrow_{32}$ Regs[x3]$_{32..63}$
fsw f0,40(x3)	Store FP single	Mem[40+Regs[x3]] $\leftarrow_{32}$ Regs[f0]$_{0..31}$
fsd f0,40(x3)	Store FP double	Mem[40+Regs[x3]] $\leftarrow_{64}$ Regs[f0]
sh x3,502(x2)	Store half	Mem[502+Regs[x2]] $\leftarrow_{16}$ Regs[x3]$_{48..63}$
sb x2,41(x3)	Store byte	Mem[41+Regs[x3]] $\leftarrow_{8}$ Regs[x2]$_{56..63}$

Figure A.25 The load and store instructions in RISC-V. Loads shorter than 64 bits are available in both sign-extended and zero-extended forms. All memory references use a single addressing mode. Of course, both loads and stores are available for all the data types shown. Because RV64G supports double precision floating point, all single precision floating point loads must be aligned in the FP register, which are 64-bits wide.

Control Transfer Instructions

Example instruction	Instruction name	Meaning
`jal x1,offset`	Jump and link	`Regs[x1]←PC+4; PC←PC + (offset<<1)`
`jalr x1,x2,offset`	Jump and link register	`Regs[x1]←PC+4; PC←Regs[x2]+offset`
`beq x3,x4,offset`	Branch equal zero	`if (Regs[x3]==Regs[x4]) PC←PC + (offset<<1)`
`bgt x3,x4,name`	Branch not equal zero	`if (Regs[x3]>Regs[x4]) PC←PC + (offset<<1)`

Figure A.27 Typical control flow instructions in RISC-V. All control instructions, except jumps to an address in a register, are PC-relative.

RISC-V Dynamic Instruction Mix for SPECint2006

Program	Loads	Stores	Branches	Jumps	ALU operations
astar	28%	6%	18%	2%	46%
bzip	20%	7%	11%	1%	54%
gcc	17%	23%	20%	4%	36%
gobmk	21%	12%	14%	2%	50%
h264ref	33%	14%	5%	2%	45%
hmmer	28%	9%	17%	0%	46%
libquantum	16%	6%	29%	0%	48%
mcf	35%	11%	24%	1%	29%
omnetpp	23%	15%	17%	7%	31%
perlbench	25%	14%	15%	7%	39%
sjeng	19%	7%	15%	3%	56%
xalancbmk	30%	8%	27%	3%	31%

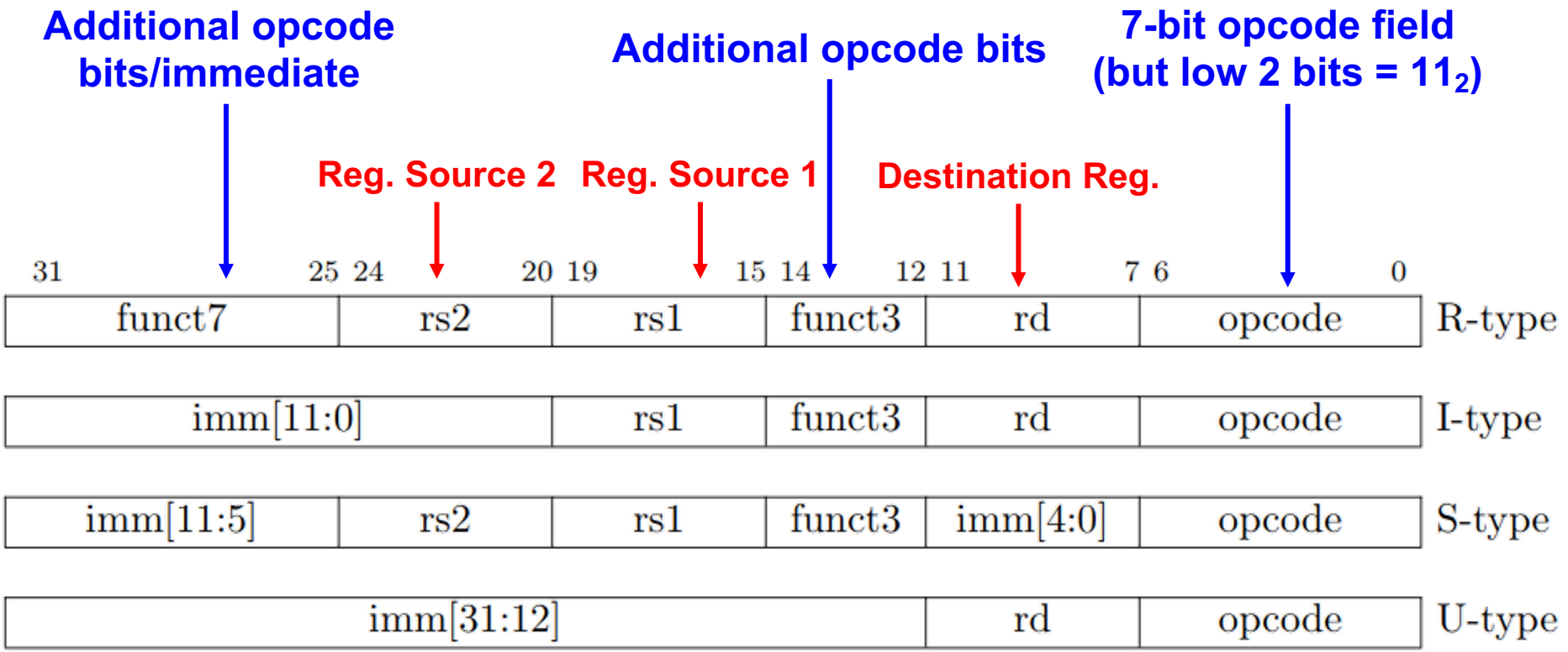
Figure A.29 RISC-V dynamic instruction mix for the SPECint2006 programs. Omnetpp includes 7% of the instructions that are floating point loads, stores, operations, or compares; no other program includes even 1% of other instruction types. A change in gcc in SPECint2006, creates an anomaly in behavior. Typical integer programs have load frequencies that are 1/5 to 3x the store frequency. In gcc, the store frequency is actually higher than the load frequency! This arises because a large fraction of the execution time is spent in a loop that clears memory by storing x0 (not where a compiler like gcc would usually spend most of its execution time!). A store instruction that stores a register pair, which some other RISC ISAs have included, would address this issue.

RISC-V Hybrid Instruction Encoding

- 16, 32, 48, 64, … bits length encoding
- Base instruction set (RV32) always has fixed 32-bit instructions with lowest two bits = 11_2
- All branches and jumps have targets at 16-bit granularity (even in base ISA where all instructions are fixed 32 bits)

Four Core RISC-V Instruction Formats

- https://github.com/riscv/riscv-opcodes/

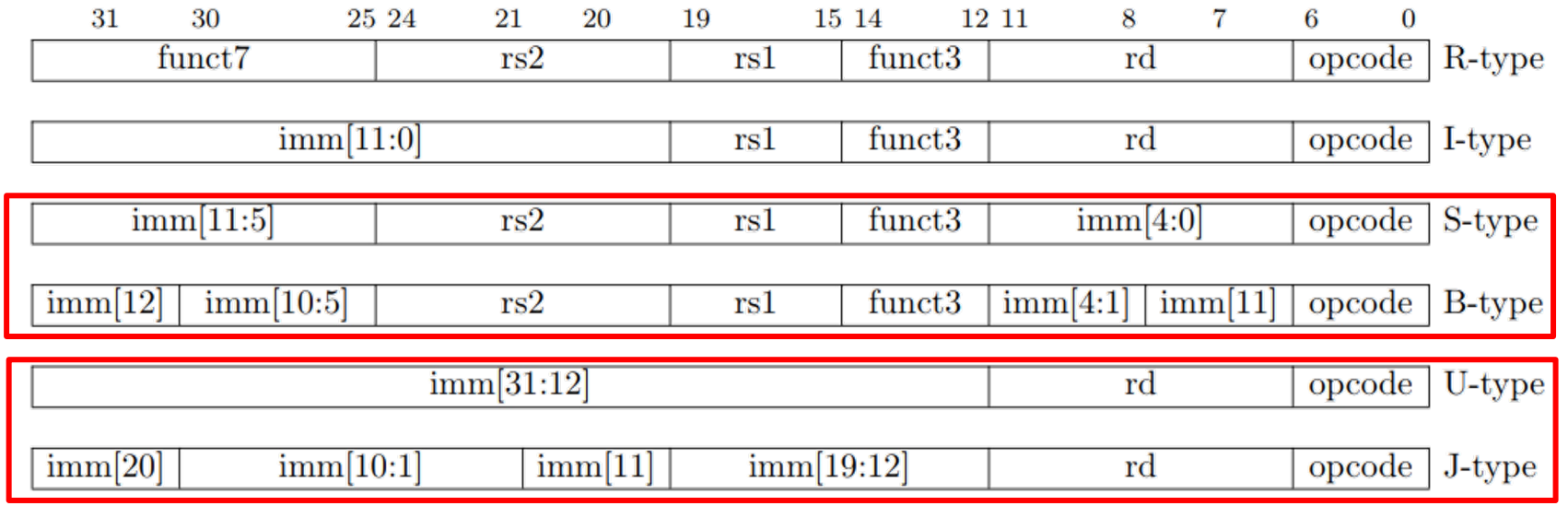


Aligned on a four-byte boundary in memory. There are variants!
Sign bit of immediates always on bit 31 of instruction. Register fields never move.

RISC-V Encoding Summary

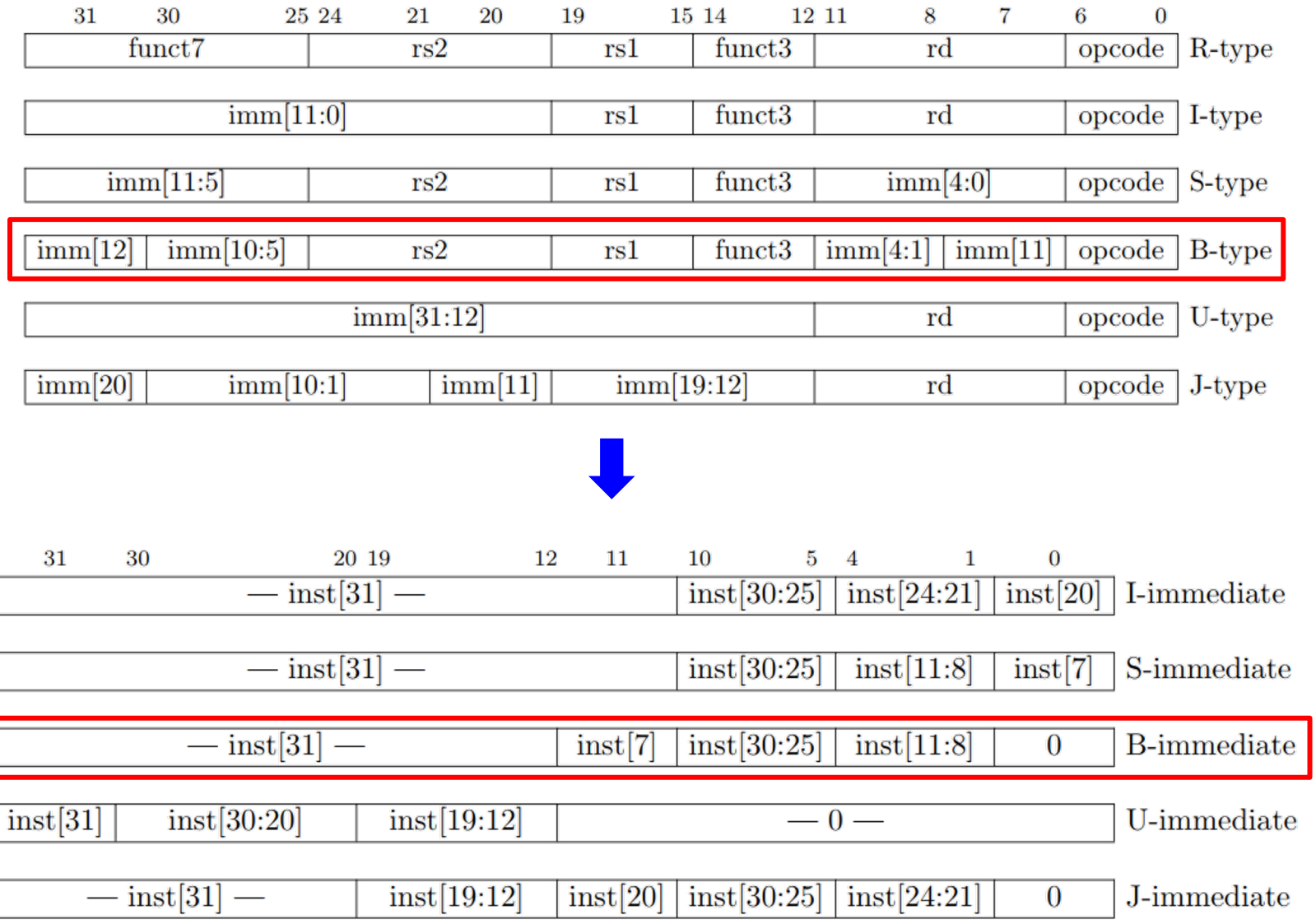
Name	Field						Comments
Field size	**7 bits**	**5 bits**	**5 bits**	**3 bits**	**5 bits**	**7 bits**	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	imm[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	Stores
B-type	imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode	Conditional branch format
J-type	imm[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	imm[31:12]				rd	opcode	Upper immediate format

Immediate Encoding Variants

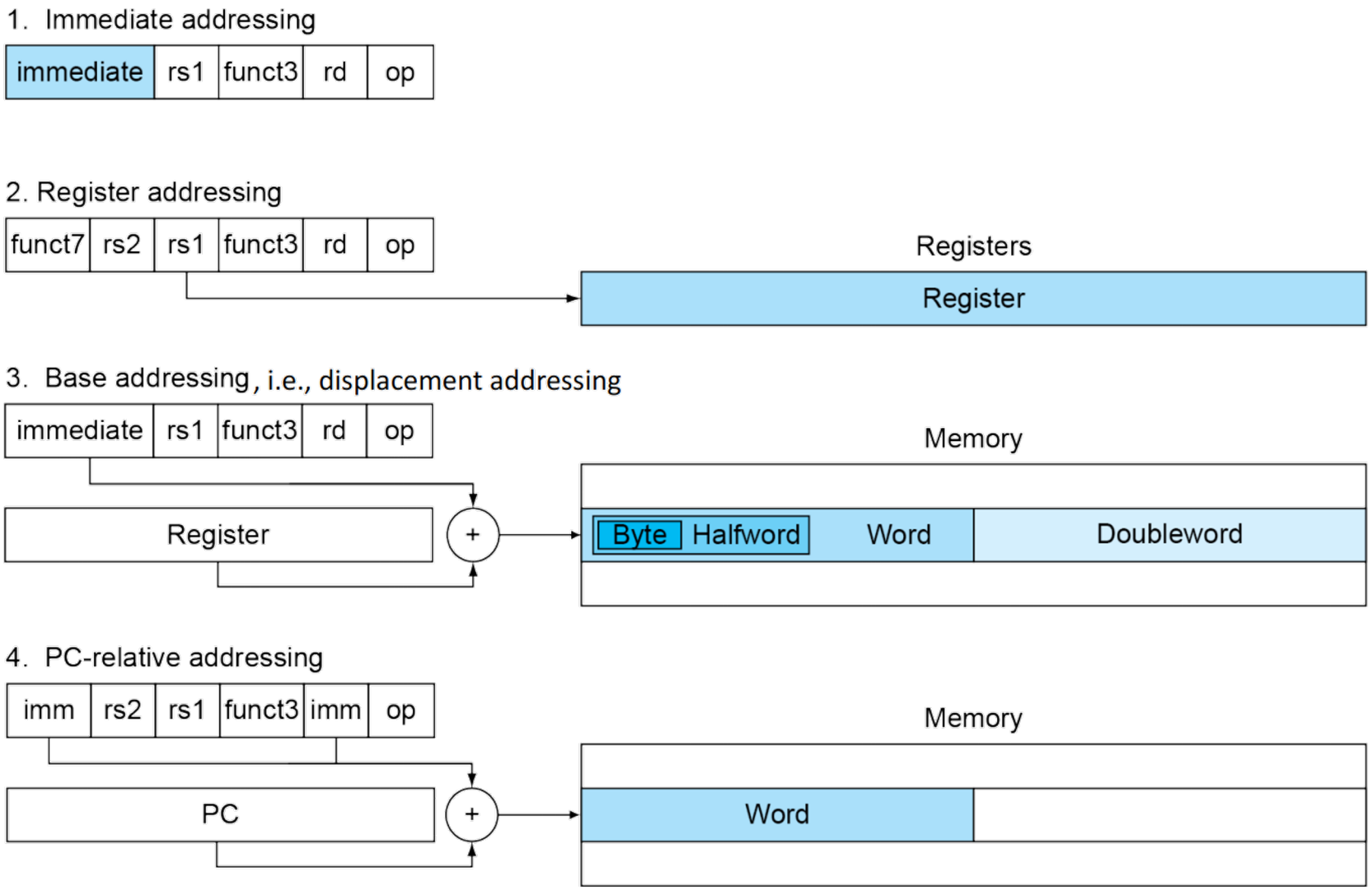


- S-type vs. B-type
  - The 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits (imm[10:1]) and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in B format.
- U-type vs. J-type
  - Similarly, the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates. The location of instruction bits in the U and J format immediates is chosen to maximize overlap with the other formats and with each other.

Immediate Encoding Variants

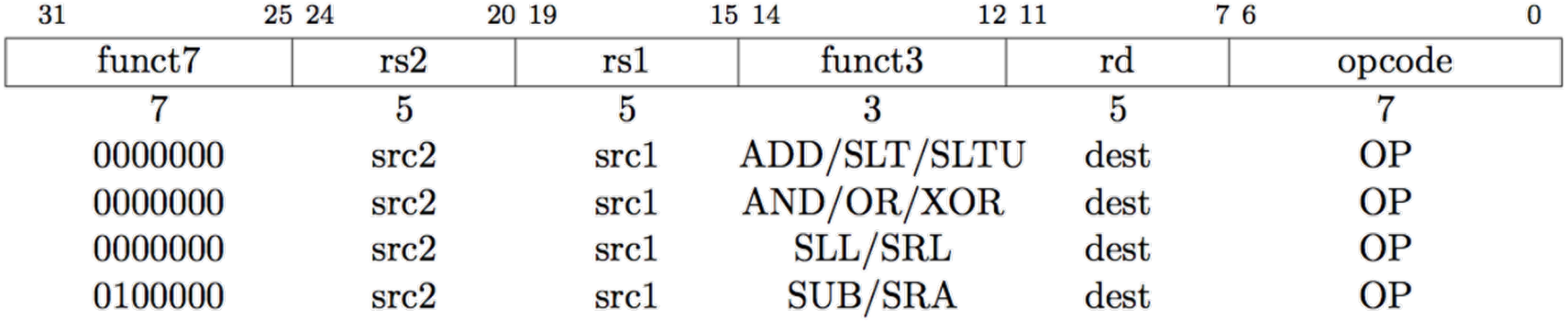


RISC-V Addressing Modes



ALU Instructions: R-Type

- R-type (Register)
  - rs1 and rs2 are the source register, rd is the destination
  - ADD/SUB
  - SLT, SLTU: set less than
  - SRL, SLL, SRA: shift logic or arithmetic left or right



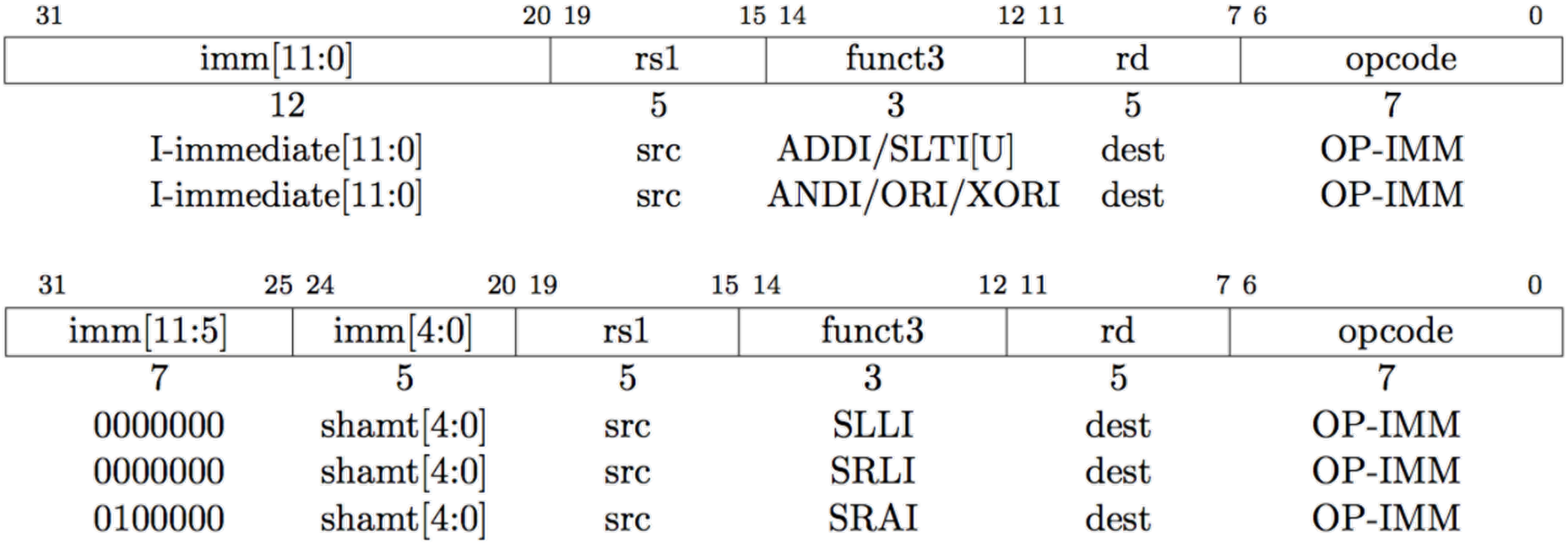
31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends

ALU Instructions: I-Type

- I-type (immediate), all immediates in all instructions are sign extended
  - ADDI: adds sign extended 12-bit immediate to rs1
  - SLTI(U): set less than immediate
  - ANDI/ORI/XORI: logical operations
  - SLLI/SRLI/SRAI: shifts by constants

I-type instructions end with I

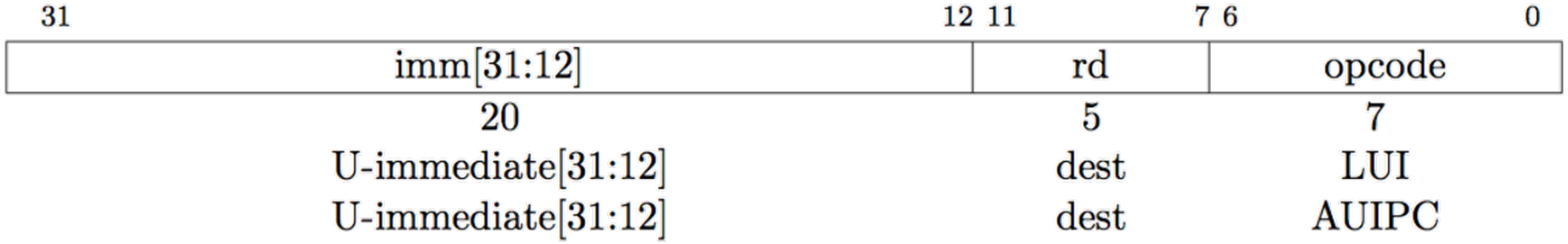


31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

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

ALU Instructions: U-Type

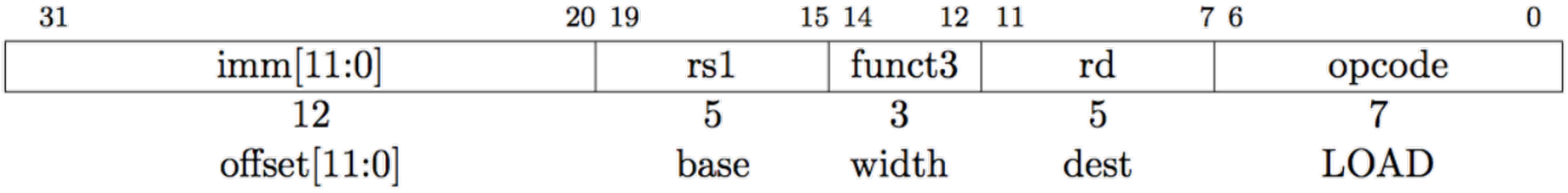
- LUI/AUIPC: load upper immediate/add upper immediate to PC

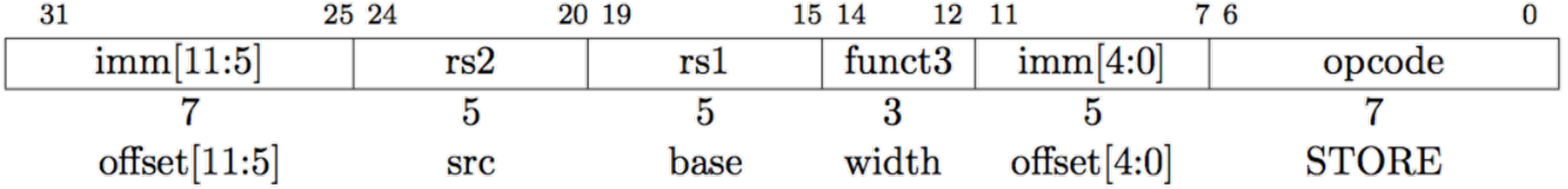


- Writes 20-bit immediate to top of destination register
- Used to build large immediates
- 12-bit immediates are signed, so have to account for sign when building 32-bit immediates in 2-instruction sequence (LUI high-20 bits, ADDI low-12 bits)

Load/Store Instructions: I/S-Type

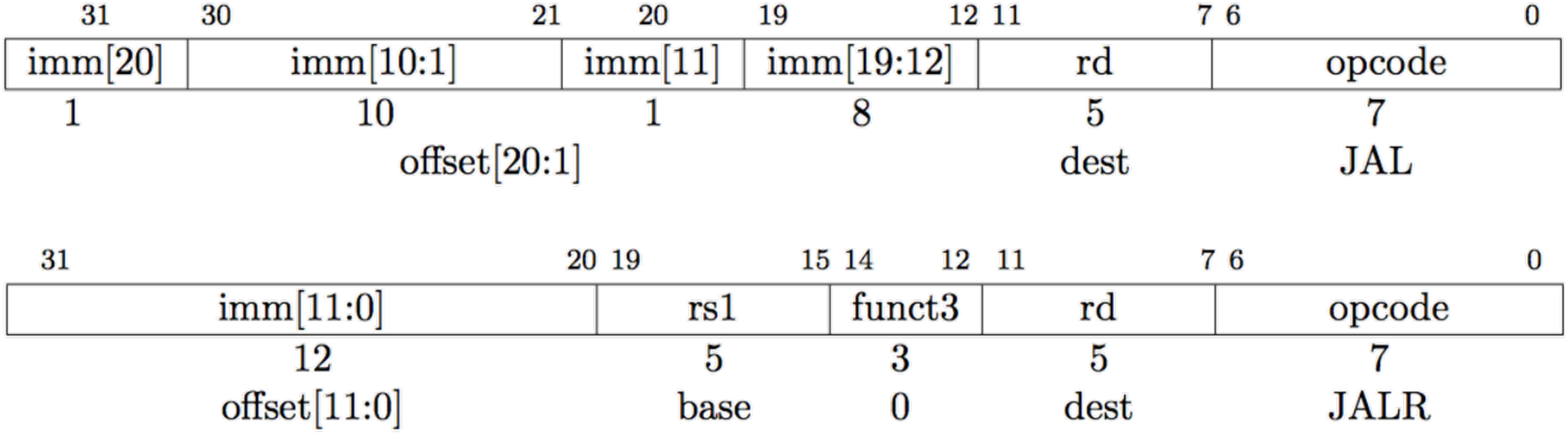
- Load instruction (I-type)
  - rd = MEM (rs1 + imm)
- Store instruction (S-type)
  - MEM (rs1 + imm) = rs2





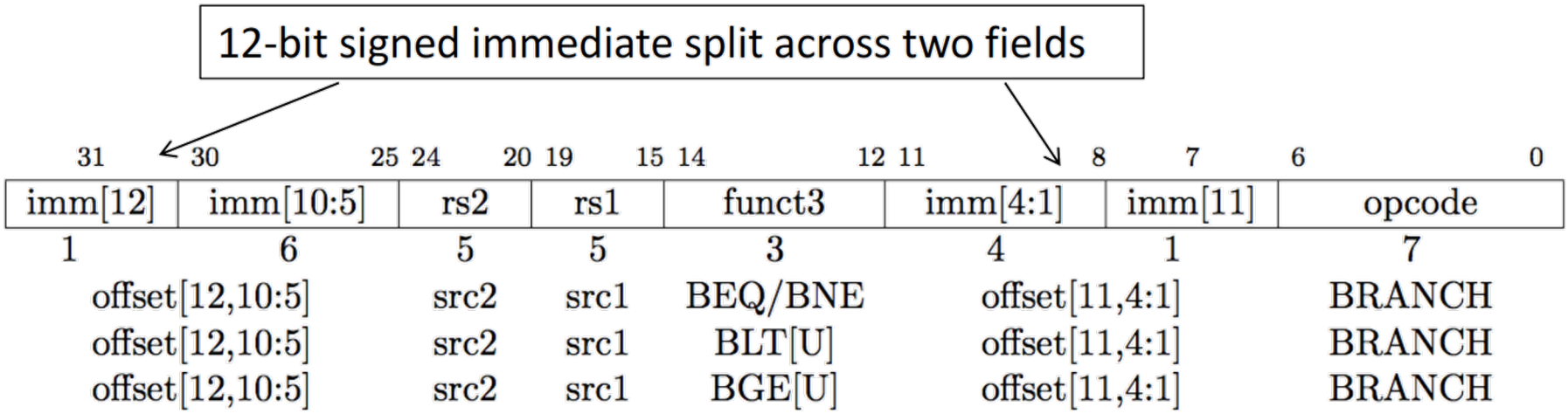
Control Transfer Instructions: J-Type

- No architecturally visible delay slots
- Unconditional jumps: PC + offset target
  - JAL: jump and link, also writes PC + 4 to x1, J-type
    - Offset scaled by 1-bit left shift – can jump to 16-bit instruction boundary (same for branches)
  - JLAR: jump and link register where imm (12bits) + rd1 = target



Control Transfer Instructions: B-Type

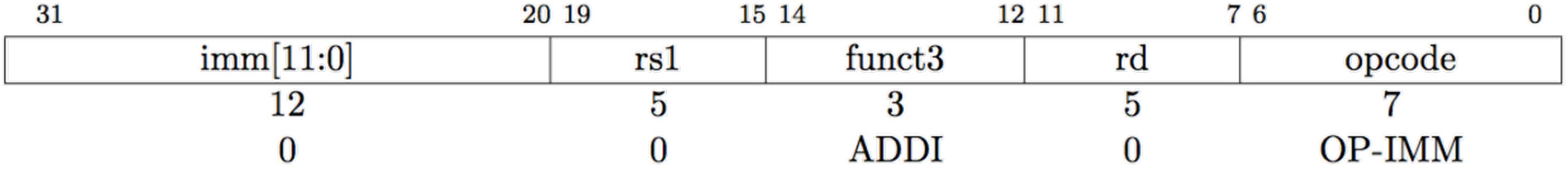
- No architecturally visible delay slots
- Conditional branches: B-type and PC + offset target



Branches, compare two registers, PC + (immediate << 1) target (signed offset in multiples of two). Branches do not have delay slot.

Where is NOP?

ADDI x0, x0, 0



Privileged ISA: Modes

- RISC-V privileged spec defines 3 levels of privilege, called modes
  - Machine mode is the highest privileged mode and the only required mode

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

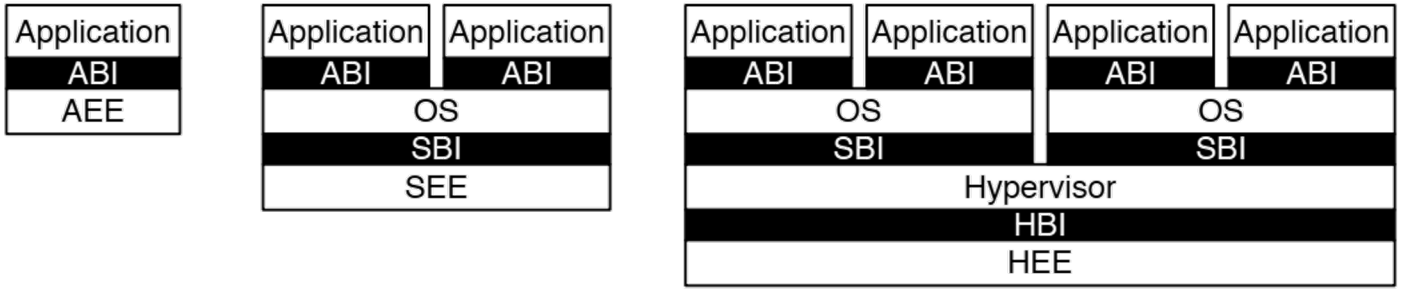
- More-privileged modes generally have access to all of the features of less-privileged modes, and they add additional functionality not available to less-privileged modes, such as the ability to handle interrupts and perform I/O. Processors typically spend most of their execution time in their least-privileged mode; interrupts and exceptions transfer control to more-privileged modes.

Software Stack and Instructions

- Implementations might provide anywhere from 1 to 4 privilege modes trading off reduced isolation for lower implementation cost

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

- RISC-V privileged software stack



- RV32/64 privileged instructions

machine-mode
supervisor-mode
} trap return

supervisor-mode fence.virtual memory address

wait for interrupt

RISC-V Reference Card

Open RISC-V Reference Card ①

Base Integer Instructions: RV32I and RV64I

Category	Name	Fmt	RV32I Base		+RV64I	
Shifts	Shift Left Logical	R	SLL	rd,rs1,rs2	SLLW	rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI	rd,rs1,shamt	SLLIW	rd,rs1,shamt
	Shift Right Logical	R	SRL	rd,rs1,rs2	SRLW	rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt	SRLIW	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2	SRAW	rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt	SRAIW	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2	ADDW	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm	ADDIW	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2	SUBW	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm		
	Add Upper Imm to PC	U	AUIPC	rd,imm		
Logical	XOR	R	XOR	rd,rs1,rs2		
	XOR Immediate	I	XORI	rd,rs1,imm		
	OR	R	OR	rd,rs1,rs2		
	OR Immediate	I	ORI	rd,rs1,imm		
	AND	R	AND	rd,rs1,rs2		
	AND Immediate	I	ANDI	rd,rs1,imm		
Compare	Set <	R	SLT	rd,rs1,rs2		
	Set < Immediate	I	SLTI	rd,rs1,imm		
	Set < Unsigned	R	SLTU	rd,rs1,rs2		
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm		
Branches	Branch =	B	BEQ	rs1,rs2,imm		
	Branch ≠	B	BNE	rs1,rs2,imm		
	Branch <	B	BLT	rs1,rs2,imm		
	Branch ≥	B	BGE	rs1,rs2,imm		
	Branch < Unsigned	B	BLTU	rs1,rs2,imm		
	Branch ≥ Unsigned	B	BGEU	rs1,rs2,imm		
Jump & Link	J&L	J	JAL	rd,imm		
	Jump & Link Register	I	JALR	rd,rs1,imm		
Synch	Synch thread	I	FENCE			
	Synch Instr & Data	I	FENCE.I			
Environment	CALL	I	ECALL			
	BREAK	I	EBREAK			

Control Status Register (CSR)

Read/Write	I	CSRRW	rd,csr,rs1
Read & Set Bit	I	CSRRS	rd,csr,rs1
Read & Clear Bit	I	CSRRC	rd,csr,rs1
Read/Write Imm	I	CSRRWI	rd,csr,imm
Read & Set Bit Imm	I	CSRRSI	rd,csr,imm
Read & Clear Bit Imm	I	CSRRCI	rd,csr,imm

Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm
	Load Half Unsigned	I	LHU	rd,rs1,imm
	Load Word	I	LW	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm
	Store Halfword	S	SH	rs1,rs2,imm
	Store Word	S	SW	rs1,rs2,imm

+RV64I			
LWU	rd,rs1,imm		
LD	rd,rs1,imm		
SD	rs1,rs2,imm		

RV Privileged Instructions

Category	Name	Fmt	RV mnemonic
Trap	Mach-mode trap return	R	MRET
	Supervisor-mode trap return	R	SRET
Interrupt	Wait for Interrupt	R	WFI
MMU	Virtual Memory FENCE	R	SFENCE.VMA rs1,rs2

Examples of the 60 RV Pseudoinstructions

Branch = 0 (BEQ rs,x0,imm)	J	BEQZ rs,imm
Jump (uses JAL x0,imm)	J	J imm
MoVe (uses ADDI rd,rs,0)	R	MV rd,rs
RETurn (uses JALR x0,0,ra)	I	RET

Optional Compressed (16-bit) Instruction Extension: RV32C

Category	Name	Fmt	RVC		RISC-V equivalent	
Loads	Load Word	CL	C.LW	rd',rs1',imm	LW	rd',rs1',imm*4
	Load Word SP	CI	C.LWSP	rd,imm	LW	rd,sp,imm*4
	Float Load Word SP	CL	C.FLW	rd',rs1',imm	FLW	rd',rs1',imm*8
	Float Load Word	CI	C.FLWSP	rd,imm	FLW	rd,sp,imm*8
	Float Load Double	CL	C.FLD	rd',rs1',imm	FLD	rd',rs1',imm*16
	Float Load Double SP	CI	C.FLDSP	rd,imm	FLD	rd,sp,imm*16
Stores	Store Word	CS	C.SW	rs1',rs2',imm	SW	rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP	rs2,imm	SW	rs2,sp,imm*4
	Float Store Word	CS	C.FSW	rs1',rs2',imm	FSW	rs1',rs2',imm*8
	Float Store Word SP	CSS	C.FSWSP	rs2,imm	FSW	rs2,sp,imm*8
	Float Store Double	CS	C.FSD	rs1',rs2',imm	FSD	rs1',rs2',imm*16
	Float Store Double SP	CSS	C.FSDSP	rs2,imm	FSD	rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD	rd,rs1	ADD	rd,rd,rs1
	ADD Immediate	CI	C.ADDI	rd,imm	ADDI	rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP	x0,imm	ADDI	sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN	rd',imm	ADDI	rd',sp,imm*4
	SUB	CR	C.SUB	rd,rs1	SUB	rd,rd,rs1
	AND	CR	C.AND	rd,rs1	AND	rd,rd,rs1
	AND Immediate	CI	C.ANDI	rd,imm	ANDI	rd,rd,imm
	OR	CR	C.OR	rd,rs1	OR	rd,rd,rs1
	eXclusive OR	CR	C.XOR	rd,rs1	AND	rd,rd,rs1
	MoVe	CR	C.MV	rd,rs1	ADD	rd,rs1,x0
	Load Immediate	CI	C.LI	rd,imm	ADDI	rd,x0,imm
	Load Upper Imm	CI	C.LUI	rd,imm	LUI	rd,imm
Shifts	Shift Left Imm	CI	C.SLLI	rd,imm	SLLI	rd,rd,imm
	Shift Right Ari. Imm.	CI	C.SRAI	rd,imm	SRAI	rd,rd,imm
	Shift Right Log. Imm.	CI	C.SRLI	rd,imm	SRLI	rd,rd,imm
Branches	Branch=0	CB	C.BEQZ	rs1',imm	BEQ	rs1',x0,imm
	Branch≠0	CB	C.BNEZ	rs1',imm	BNE	rs1',x0,imm
Jump	Jump	CJ	C.J	imm	JAL	x0,imm
	Jump Register	CR	C.JR	rd,rs1	JALR	x0,rs1,0
Jump & Link	J&L	CJ	C.JAL	imm	JAL	ra,imm
	Jump & Link Register	CR	C.JALR	rs1	JALR	ra,rs1,0
System	Env. BREAK	CI	C.EBREAK		EBREAK	

Optional Compressed Extention: RV64C

All RV32C (except C.JAL, 4 word loads, 4 word stores) plus:

ADD Word (C.ADDW)	Load Doubleword (C.LD)
ADD Imm. Word (C.ADDIW)	Load Doubleword SP (C.LDSP)
SUBtract Word (C.SUBW)	Store Doubleword (C.SD)
	Store Doubleword SP (C.SDSP)

32-bit Instruction Formats

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

16-bit (RVC) Instruction Formats

	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	funct4		rd/rs1		rs2		op
CI	funct3	imm	rd/rs1		imm		op
CSS	funct3	imm			rs2		op
CIW	funct3	imm			rd'		op
CL	funct3	imm	rs1'	imm	rd'		op
CS	funct3	imm	rs1'	imm	rs2'		op
CB	funct3	offset	rs1'	offset			op
CJ	funct3	jump target					op

RISC-V Integer Base (RV32I/64I), privileged, and optional RV32/64C. Registers x1-x31 and the PC are 32 bits wide in RV32I and 64 in RV64I (x0=0). RV64I adds 12 instructions for the wider data. Every 16-bit RVC instruction maps to an existing 32-bit RISC-V instruction.

Open RISC-V Reference Card ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV64M	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MULW	rd,rs1,rs2
	MULtiply High	R	MULH	rd,rs1,rs2		
	MULtiply High Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply High Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIVW	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REMW	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMUW	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV64A	
Load	Load Reserved	R	LR.W	rd,rs1	LR.D	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.D	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.D	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.D	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.D	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.D	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.D	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.D	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.D	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.D	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.D	rd,rs1,rs2

Two Optional Floating-Point Instruction Extensions: RVF & RVD

Category	Name	Fmt	RV32{F|D} (SP,DP Fl. Pt.)		+RV64{F|D}	
Move	Move from Integer	R	FMV.W.X	rd,rs1	FMV.D.X	rd,rs1
	Move to Integer	R	FMV.X.W	rd,rs1	FMV.X.D	rd,rs1
Convert	ConVerT from Int	R	FCVT.{S|D}.W	rd,rs1	FCVT.{S|D}.L	rd,rs1
	ConVerT from Int Unsigned	R	FCVT.{S|D}.WU	rd,rs1	FCVT.{S|D}.LU	rd,rs1
	ConVerT to Int	R	FCVT.W.{S|D}	rd,rs1	FCVT.L.{S|D}	rd,rs1
	ConVerT to Int Unsigned	R	FCVT.WU.{S|D}	rd,rs1	FCVT.LU.{S|D}	rd,rs1
Load	Load	I	FL{W,D}	rd,rs1,imm		
Store	Store	S	FS{W,D}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{S|D}	rd,rs1,rs2		
	SUBtract	R	FSUB.{S|D}	rd,rs1,rs2		
	MULtiply	R	FMUL.{S|D}	rd,rs1,rs2		
	DIVide	R	FDIV.{S|D}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{S|D}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D}	rd,rs1,rs2,rs3		
	Multiply-SUBtract	R	FMSUB.{S|D}	rd,rs1,rs2,rs3		
	Negative Multiply-SUBtract	R	FNMSUB.{S|D}	rd,rs1,rs2,rs3		
	Negative Multiply-ADD	R	FNMADD.{S|D}	rd,rs1,rs2,rs3		
Sign Inject	SiGN source	R	FSGNJ.{S|D}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{S|D}	rd,rs1,rs2		
	MAXimum	R	FMAX.{S|D}	rd,rs1,rs2		
Compare	compare Float =	R	FEQ.{S|D}	rd,rs1,rs2		
	compare Float <	R	FLT.{S|D}	rd,rs1,rs2		
	compare Float ≤	R	FLE.{S|D}	rd,rs1,rs2		
Categorize	CLASSify type	R	FCLASS.{S|D}	rd,rs1		
Configure	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

Calling Convention

Register	ABI Name	Saver
x0	zero	---
x1	ra	Caller
x2	sp	Callee
x3	gp	---
x4	tp	---
x5-7	t0-2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10-11	a0-1	Caller
x12-17	a2-7	Caller
x18-27	s2-11	Callee
x28-31	t3-t6	Caller
f0-7	ft0-7	Caller
f8-9	fs0-1	Callee
f10-11	fa0-1	Caller
f12-17	fa2-7	Caller
f18-27	fs2-11	Callee
f28-31	ft8-11	Caller

Hardwired zero	
Return address	
Stack pointer	
Global pointer	
Thread pointer	
Temporaries	t0-0,ft0-7
Saved registers	s0-11,fs0-11
Function args	a0-7,fa0-7

Optional Vector Extension: RVV

Name	Fmt	RV32V/R64V	
SET Vector Len.	R	SETVL	rd,rs1
MULtiply High	R	VMULH	rd,rs1,rs2
REMainder	R	VREM	rd,rs1,rs2
Shift Left Log.	R	VSLL	rd,rs1,rs2
Shift Right Log.	R	VSRL	rd,rs1,rs2
Shift R. Arith.	R	VSRA	rd,rs1,rs2
LoaD	I	VLD	rd,rs1,imm
LoaD Strided	R	VLDS	rd,rs1,rs2
LoaD indeXed	R	VLDX	rd,rs1,rs2
STore	S	VST	rd,rs1,imm
STore Strided	R	VSTS	rd,rs1,rs2
STore indeXed	R	VSTX	rd,rs1,rs2
AMO SWAP	R	AMOSWAP	rd,rs1,rs2
AMO ADD	R	AMOADD	rd,rs1,rs2
AMO XOR	R	AMOXOR	rd,rs1,rs2
AMO AND	R	AMOAND	rd,rs1,rs2
AMO OR	R	AMOOR	rd,rs1,rs2
AMO MINimum	R	AMOMIN	rd,rs1,rs2
AMO MAXimum	R	AMOMAX	rd,rs1,rs2
Predicate =	R	VPEQ	rd,rs1,rs2
Predicate ≠	R	VPNE	rd,rs1,rs2
Predicate <	R	VPLT	rd,rs1,rs2
Predicate ≥	R	VPGE	rd,rs1,rs2
Predicate AND	R	VPAND	rd,rs1,rs2
Pred. AND NOT	R	VPANDN	rd,rs1,rs2
Predicate OR	R	VPOR	rd,rs1,rs2
Predicate XOR	R	VPXOR	rd,rs1,rs2
Predicate NOT	R	VPNOT	rd,rs1
Pred. SWAP	R	VPSWAP	rd,rs1
MOVe	R	VMOV	rd,rs1
ConVerT	R	VCVT	rd,rs1
ADD	R	VADD	rd,rs1,rs2
SUBtract	R	VSUB	rd,rs1,rs2
MULtiply	R	VMUL	rd,rs1,rs2
DIVide	R	VDIV	rd,rs1,rs2
SQuare RooT	R	VSQRT	rd,rs1,rs2
Multiply-ADD	R	VFMADD	rd,rs1,rs2,rs3
Multiply-SUB	R	VFMSUB	rd,rs1,rs2,rs3
Neg. Mul.-SUB	R	VFNMSUB	rd,rs1,rs2,rs3
Neg. Mul.-ADD	R	VFNMADD	rd,rs1,rs2,rs3
SiGN inJect	R	VSGNJ	rd,rs1,rs2
Neg SiGN inJect	R	VSGNJN	rd,rs1,rs2
Xor SiGN inJect	R	VSGNJX	rd,rs1,rs2
MINimum	R	VMIN	rd,rs1,rs2
MAXimum	R	VMAX	rd,rs1,rs2
XOR	R	VXOR	rd,rs1,rs2
OR	R	VOR	rd,rs1,rs2
AND	R	VAND	rd,rs1,rs2
CLASS	R	VCLASS	rd,rs1
SET Data Conf.	R	VSETDCFG	rd,rs1
EXTRACT	R	VEXTRACT	rd,rs1,rs2
MERGE	R	VMERGE	rd,rs1,rs2
SELECT	R	VSELECT	rd,rs1,rs2

RISC-V calling convention and five optional extensions: 8 RV32M; 11 RV32A; 34 floating-point instructions each for 32- and 64-bit data (RV32F, RV32D); and 53 RV32V. Using regex notation, {} means set, so FADD.{F|D} is both FADD.F and FADD.D. RV32{F|D} adds registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. RV32V adds vector registers v0-v31, vector predicate registers vp0-vp7, and vector length register vl. RV64 adds a few instructions: RVM gets 4, RVA 11, RVF 6, RVD 6, and RVV 0.

Specifications and Software

- Specification from RISC-V website
  - https://riscv.org/specifications/
- RISC-V software includes
  - Toolchain projects
    - https://wiki.riscv.org/display/HOME/Toolchain+Projects
  - A simulator (“spike”)
    - https://github.com/riscv-software-src/riscv-isa-sim
  - Standard simulator QEMU (Upstream now)
    - https://github.com/riscv/riscv-qemu
- Operating systems support exists for Linux (Upstream now)
  - https://github.com/riscv/riscv-linux
- A javascript ISA simulator to run a RISC-V Linux system on a web browser
  - https://github.com/riscv/riscv-angel