Lecture 04 RISC-V ISA

ITSC 3181 Intro to Computer Architecture

Department of Computer Science
Yonghong Yan
yyan7@uncc.edu
https://passlab.github.io/yanyh/

Acknowledgement

- Slides adapted from
  - Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UCB

Review: ISA Principles -- Iron-code Summary

- *Section A.2*—Use general-purpose registers with a load-store architecture.
- *Section A.3*—Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register indirect.
- *Section A.4*—Support these data sizes and types: 8-, 16-, 32-, and 64-bit integers and 64-bit IEEE 754 floating-point numbers.
  - Now we see 16-bit FP for deep learning in GPU
    - http://www.nextplatform.com/2016/09/13/nvidia-pushes-deep-learning-inference-new-pascal-gpus/
- *Section A.5*—Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register- register, and shift.
- *Section A.6*—Compare equal, compare not equal, compare less, branch (with a PC-relative address at least 8 bits long), jump, call, and return.
- *Section A.7*—Use fixed instruction encoding if interested in performance, and use variable instruction encoding if interested in code size.
- *Section A.8*—Provide at least 16 general-purpose registers, be sure all addressing modes apply to all data transfer instructions, and aim for a minimalist IS
  - Often use separate floating-point registers.
  - The justification is to increase the total number of registers without raising problems in the instruction format or in the speed of the general-purpose register file. This compromise, however, is not orthogonal.

What is RISC-V

- RISC-V (pronounced "risk-five") is a 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
- 2017 version 2 of the userspace ISA is fixed
  - User-Level ISA Specification v2.2
  - Draft Compressed ISA Specification v1.79
  - Draft Privileged ISA Specification v1.10

https://riscv.org/
https://en.wikipedia.org/wiki/RISC-V

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 or 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 ISA Principles

- Generally kept very simple and extendable
- 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(nteger) 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

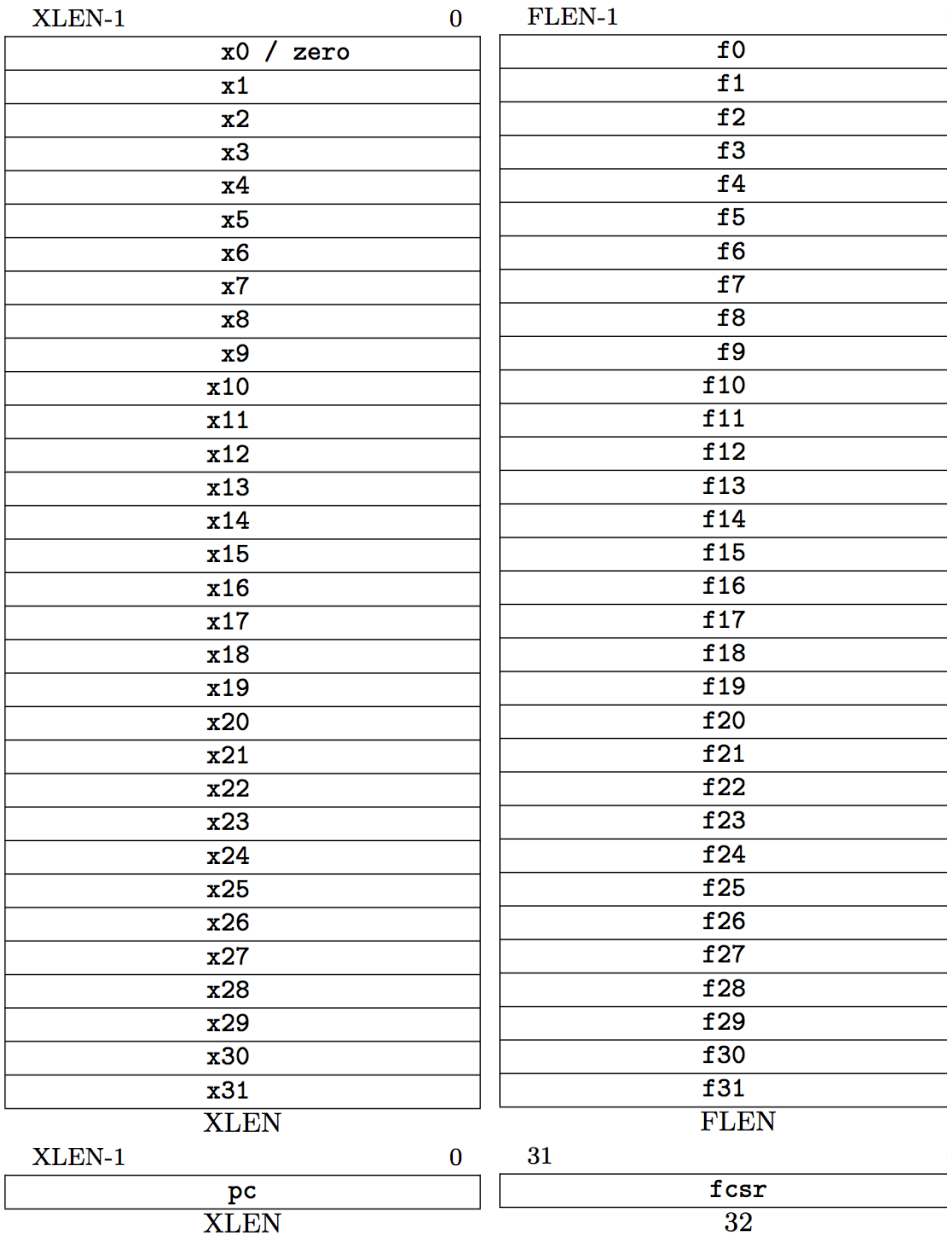
RISC-V ISA

- 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

RV32/64 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



RV64G In One Table

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP; only memory address mode is 12-bit displacement+contents of a GPR*
`lb, lbu, sb`	Load byte, load byte unsigned, store byte (to/from integer registers)
`lh, lhu, sh`	Load half word, load half word unsigned, store half word (to/from integer registers)
`lw, lwu, sw`	Load word, store word (to/from integer registers)
`ld, sd`	Load doubleword, store doubleword
Arithmetic/logical	*Operations on data in GPRs. Word versions ignore upper 32 bits*
`add, addi, addw, addiw, sub, subi, subw, subiw`	Add and subtract, with both word and immediate versions
`slt, sltu, slti, sltiu`	set-less-than with signed and unsigned, and immediate
`and, or, xor, andi, ori, xori`	and, or, xor, both register-register and register-immediate
`lui`	Load upper immediate: loads bits 31..12 of a register with the immediate value. Upper 32 bits are set to 0
`auipc`	Sums an immediate and the upper 20-bits of the PC into a register; used for building a branch to any 32-bit address
`sll, srl, sra, slli, srli, srai, sllw,slliw, srli, srliw, srai, sraiw`	Shifts: logical shift left and right and arithmetic shift right, both immediate and word versions (word versions leave the upper 32 bit untouched)
`mul, mulw, mulh, mulhsu, mulhu, div,divw, divu, rem, remu, remw, remuw`	Integer multiply, divide, and remainder, signed and unsigned with support for 64-bit products in two instructions. Also word versions
Control	*Conditional branches and jumps; PC-relative or through register*
`beq, bne, blt, bge, bltu, bgeu`	Branch based on compare of two registers, equal, not equal, less than, greater or equal, signed and unsigned
`jal,jalr`	Jump and link address relative to a register or the PC
Floating point	*All FP operation appear in double precision (.d) and single (.s)*
`flw, fld, fsw, fsd`	Load, store, word (single precision), doubleword (double precision)
`fadd, fsub, fmult, fiv, fsqrt, fmadd, fmsub, fnmadd, fnmsub, fmin, fmax, fsgn, fsgnj, fsjnx`	Add, subtract, multiply, divide, square root, multiply-add, multiply-subtract, negate multiply-add, negate multiply-subtract, maximum, minimum, and instructions to replace the sign bit. For single precision, the opcode is followed by: .s, for double precision: .d. Thus fadd.s, fadd.d
`feq, flt, fle`	Compare two floating point registers; result is 0 or 1 stored into a GPR
`fmv.x.*, fmv.*.x`	Move between the FP register abd GPR, "*" is s or d
`fcvt.*.l, fcvt.l.*, fcvt.*.lu, fcvt.lu.*, fcvt.*.w, fcvt.w.*, fcvt.*.wu, fcvt.wu.*`	Converts between a FP register and integer register, where "*" is S or D for single or double precision. Signed and unsigned versions and word, doubleword versions

Figure A.28 A list of the vast majority of instructions in RV64G. This list can also be found on the back inside cover. This table omits system instructions, synchronization and atomic instructions, configuration instructions, instructions to reset and access performance counters, about 10 instructions in total.

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.

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	$\texttt{Regs[x1]} \leftarrow 0^{32} \# \# 42 \# \# 0^{12}$
`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.

Control Flow 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.

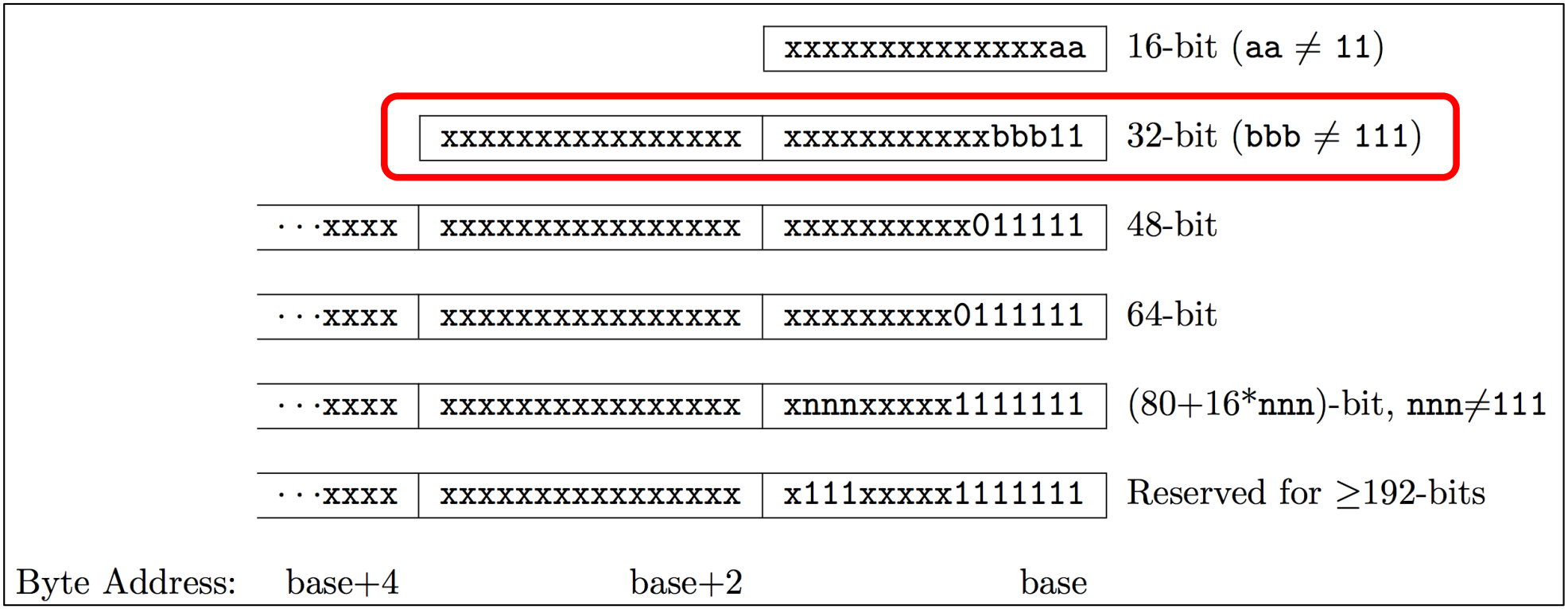
RISC-V Dynamic Instruction Mix for SPECint2006

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.

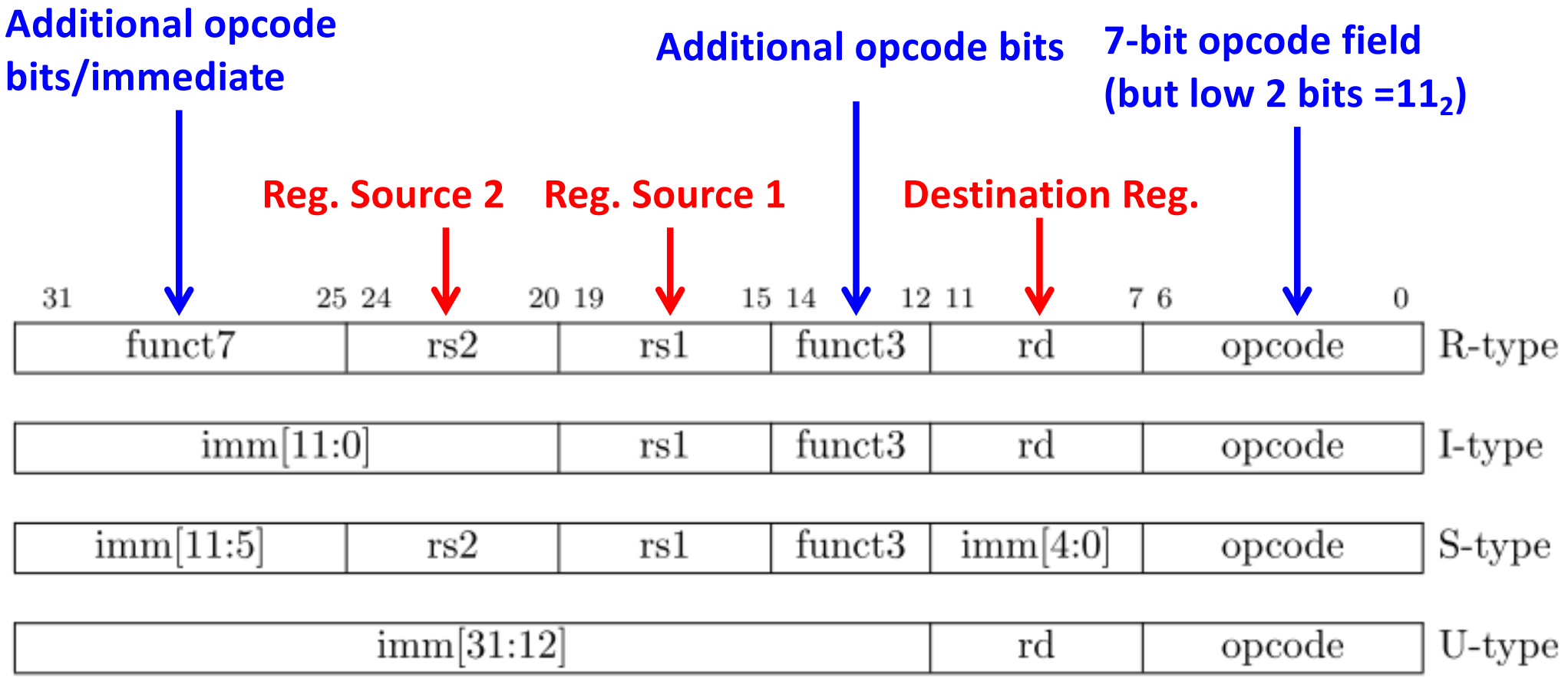
RISC-V Hybrid Instruction Encoding

- 16, 32, 48, 64 … bits length encoding
- Base instruction set (RV32) always has fixed 32-bit instructions 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

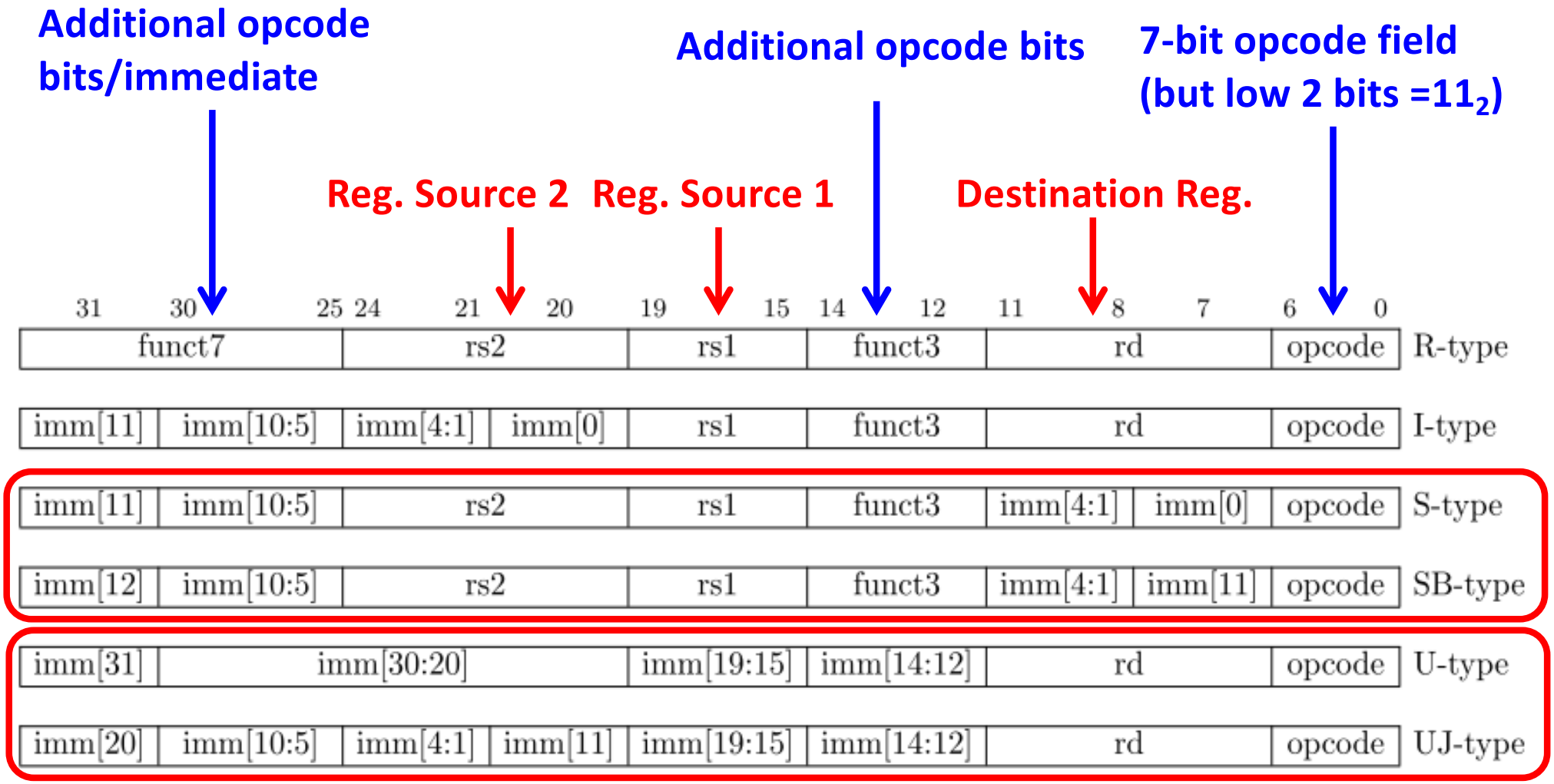
https://github.com/riscv/riscv-opcodes/blob/master/opcodes



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.

With Variants



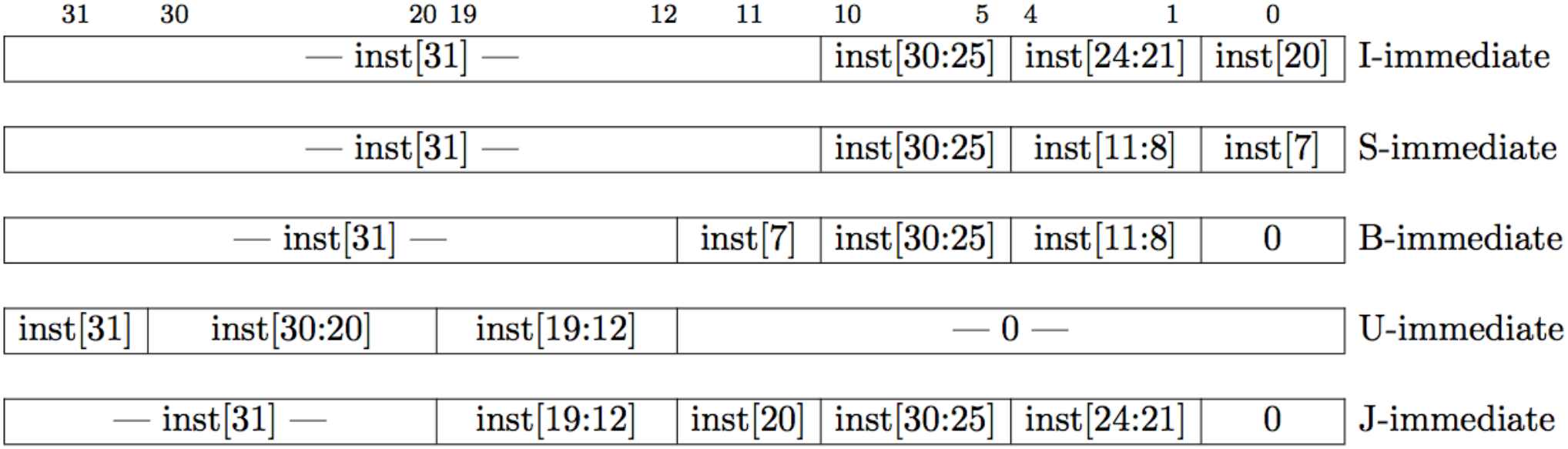
Based on the handling of the immediates

RISC-V Encoding Summary

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

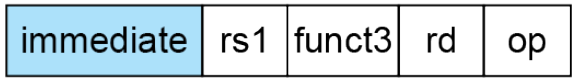
Immediate Encoding Variants

- 32-bit Immediate produced by each base instruction format
  - Instruction bit: inst[y]

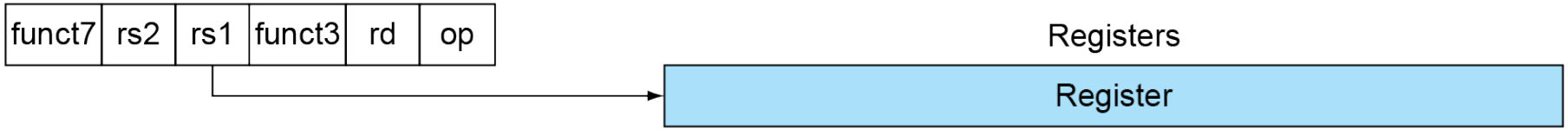


RISC-V Addressing Summary

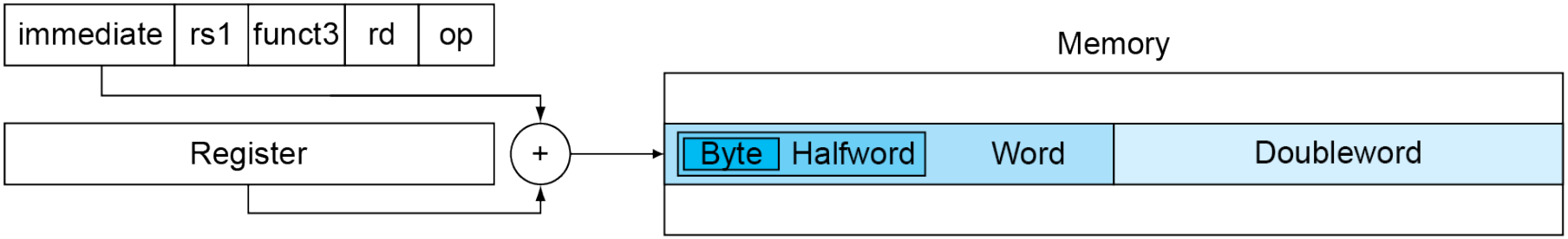
1. Immediate addressing



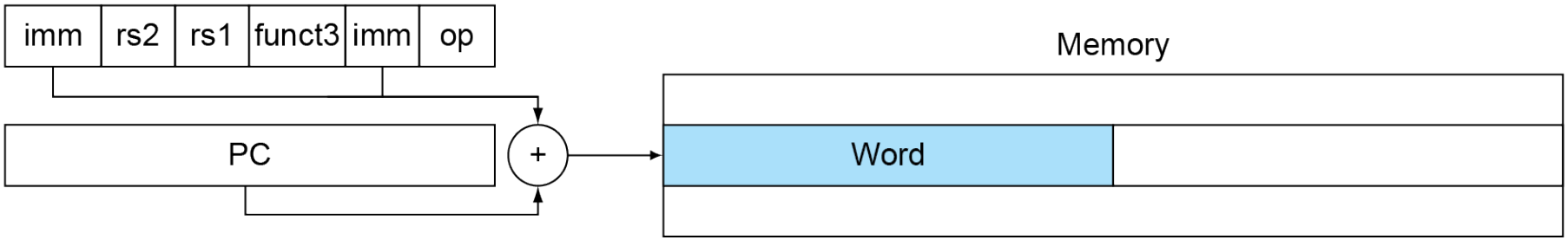
2. Register addressing



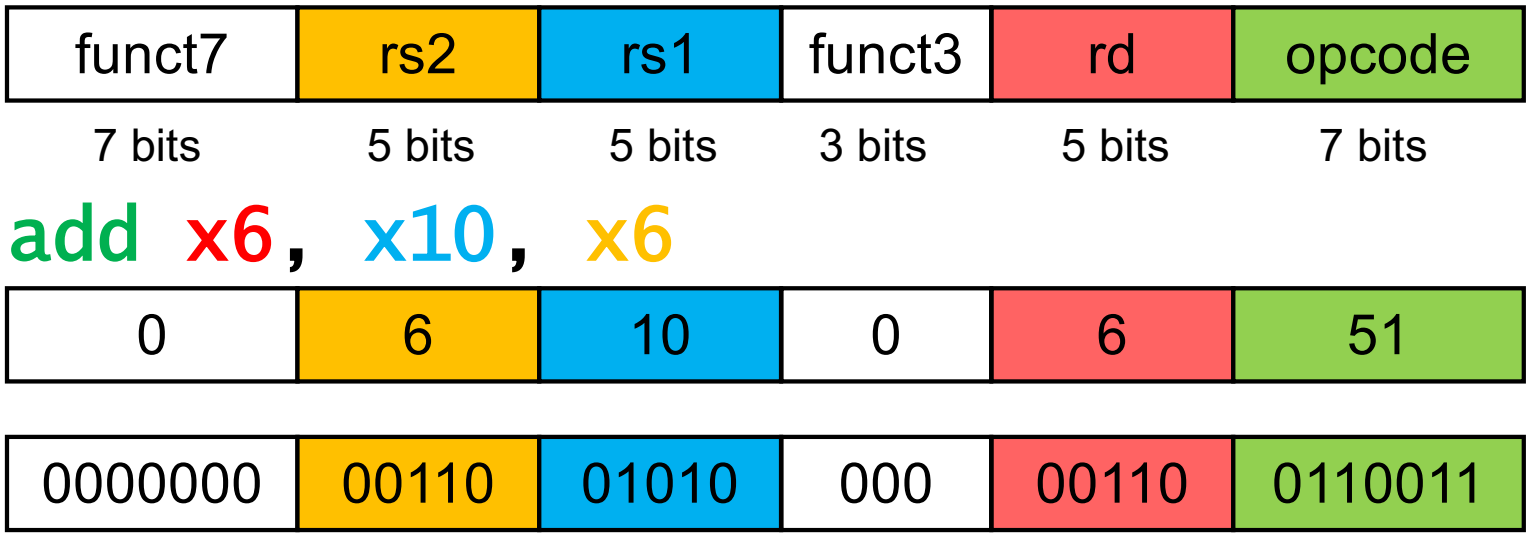
3. Base addressing, i.e., displacement addressing



4. PC-relative addressing



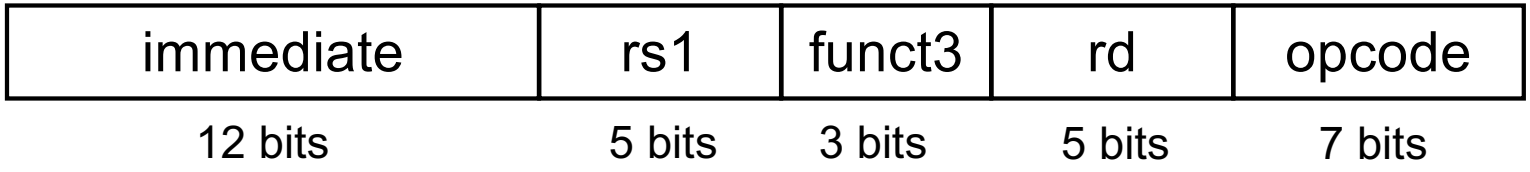
R-Format Encoding Example



$0000\ 0000\ 0110\ 0101\ 0000\ 0011\ 0011\ 0011_{two} =$

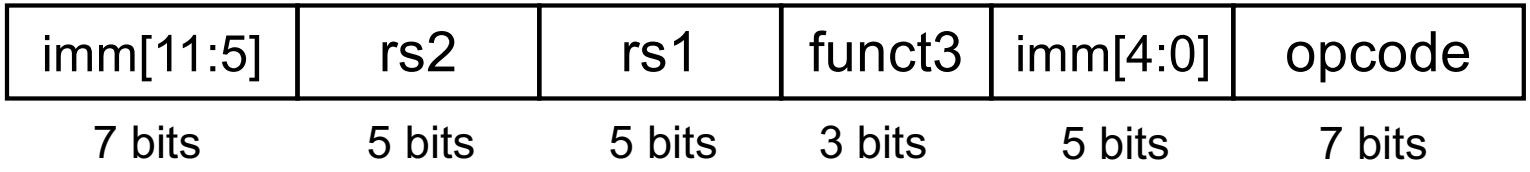
00650333_{16}

RISC-V I-Format Instructions



- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended
- *Design Principle:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

RISC-V S-Format Instructions



- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

Integer Computational Instructions (ALU)

- **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

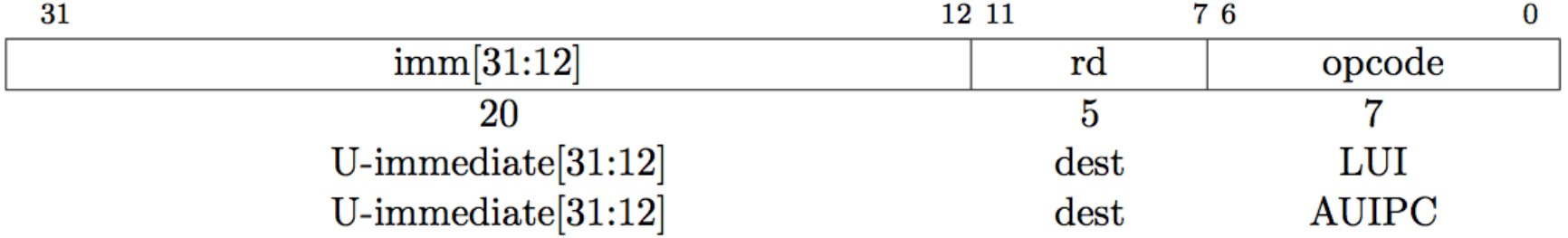
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

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

Integer Computational Instructions (ALU)

- **I-type (Immediate), all immediates in all instructions are sign extended**
  - **LUI/AUIPC: load upper immediate/add upper immediate to pc**

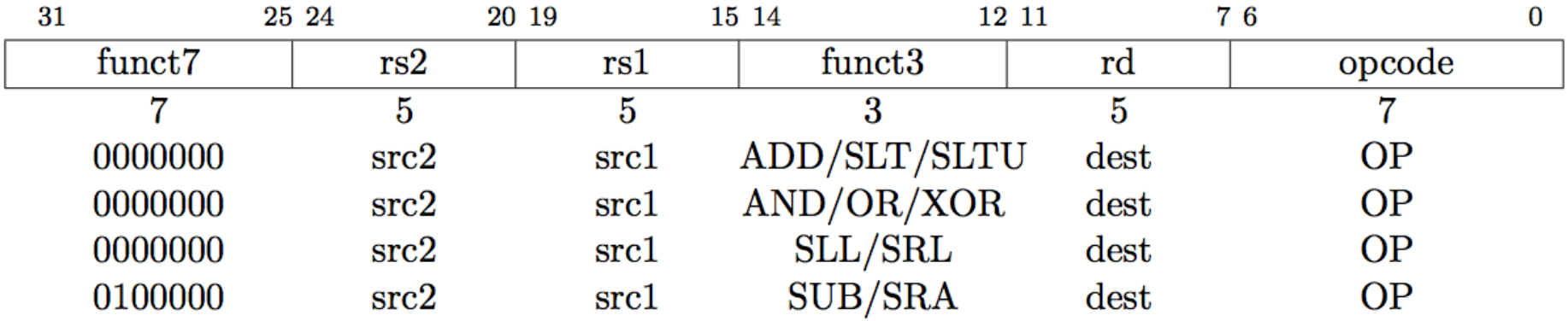
I-type instructions end with I



- 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-20b, ADDI low-12b)

Integer Computational Instructions

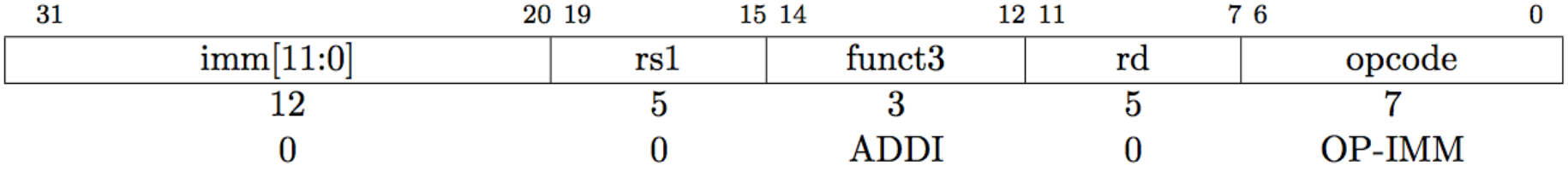
- **R-type (Register)**
  - rs1 and rs2 are the source registers. rd the destination
  - ADD/SUB:
  - SLT, SLTU: set less than
  - SRL, SLL, SRA: shift logical or arithmetic left or right



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

NOP Instruction

ADDI x0, x0, 0

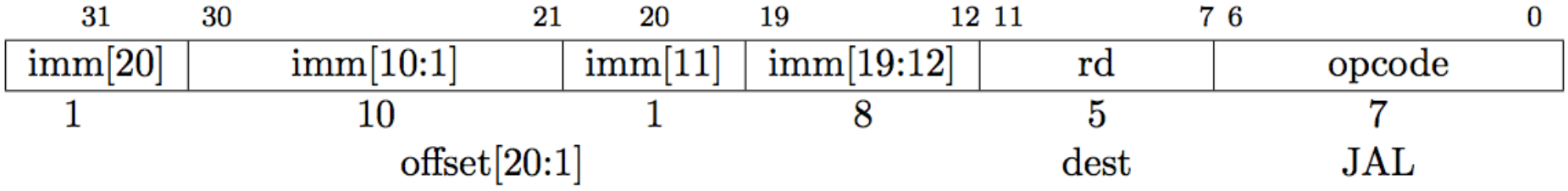


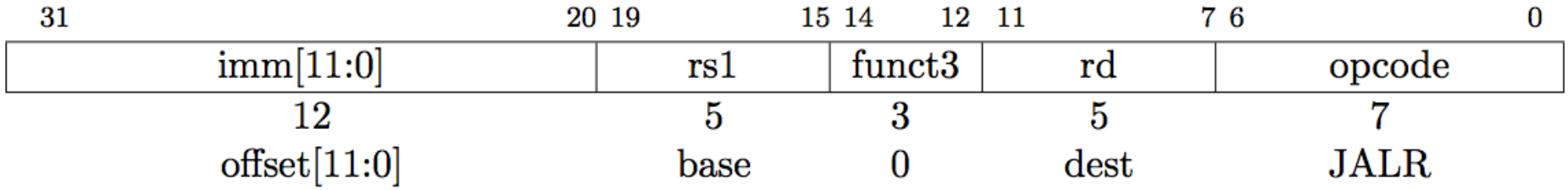
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

Control Transfer Instructions

NO architecturally visible delay slots

- Unconditional Jumps: PC+offset target
  - JAL: Jump and link, also writes PC+4 to x1, UJ-type
    - Offset scaled by 1-bit left shift – can jump to 16-bit instruction boundary (Same for branches)
  - JALR: Jump and link register where Imm (12 bits) + rd1 = target

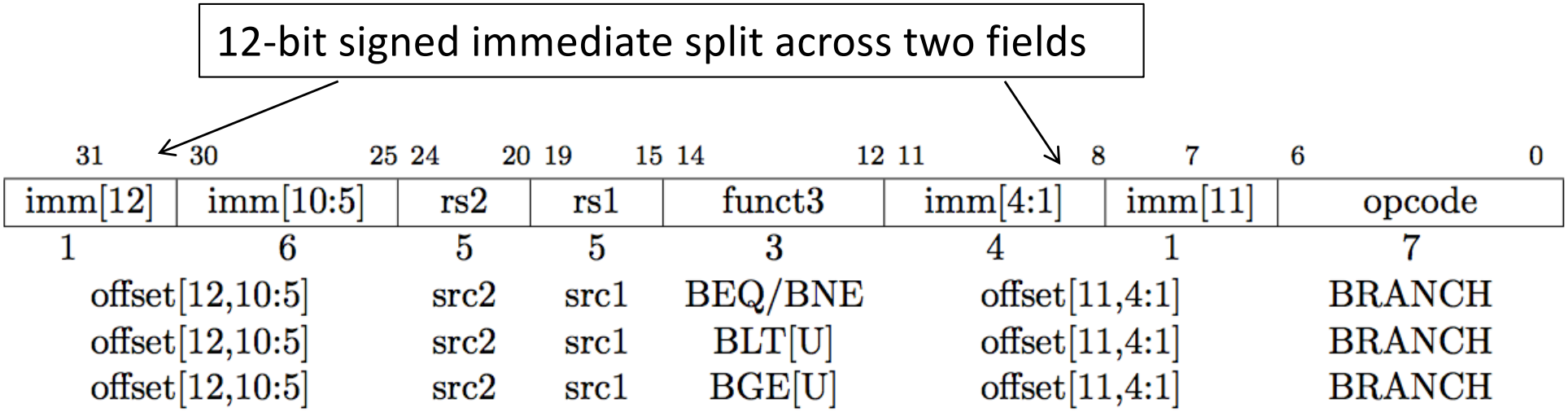




Control Transfer Instructions

NO architecturally visible delay slots

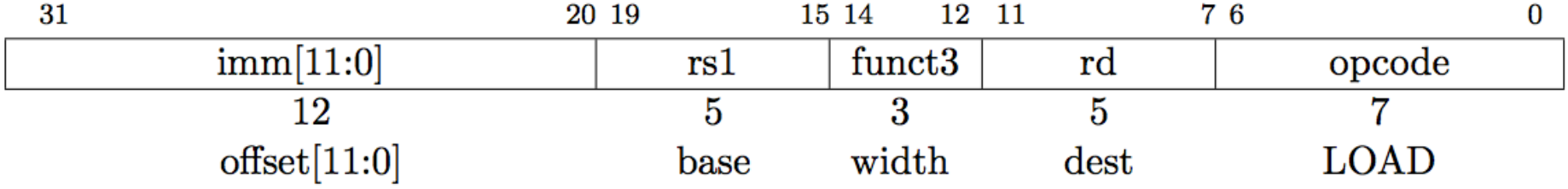
- Conditional Branches: SB-type and PC+offset target

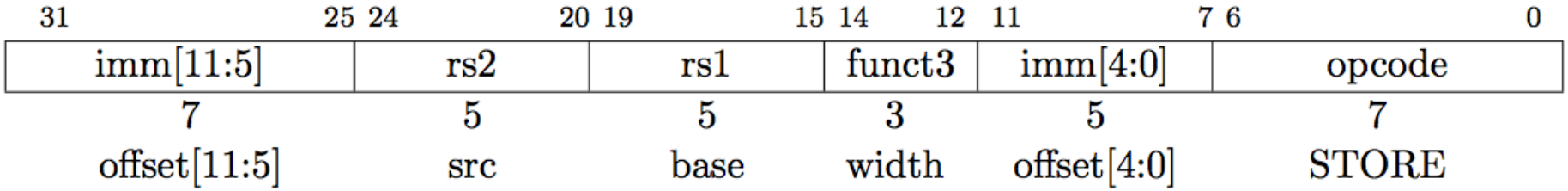


Branches, compare two registers, PC+(immediate<<1) target
(Signed offset in multiples of two). Branches do not have delay slot

Loads and Stores

- Store instructions (S-type)
  - MEM(rs1+imm) = rs2
- Loads (I-type)
  - Rd = MEM(rs1 + imm)





Specifications and Software From riscv.org and github.com/riscv

- Specification from RISC-V website
  - https://riscv.org/specifications/

- RISC-V software includes
  - GNU Compiler Collection (GCC) toolchain (with GDB, the debugger)
    - https://github.com/riscv/riscv-tools
  - LLVM toolchain
  - A simulator ("Spike")
    - https://github.com/riscv/riscv-isa-sim
  - Standard simulator QEMU
    - https://github.com/riscv/riscv-qemu
- Operating systems support exists for Linux
  - 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

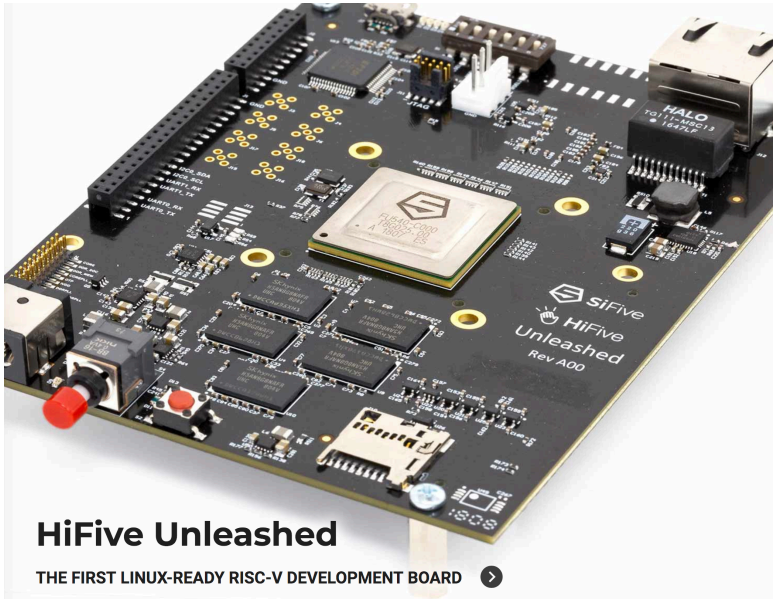
RISC-V Implementations

- For RISC-V implementation, the UCB created **Chisel**, an open-source hardware construction language that is a specialized dialect of Scala.
  - **Chisel: Constructing Hardware In a Scala Embedded Language**
  - https://chisel.eecs.berkeley.edu/
- In-order Rocket core and chip generator
  - https://github.com/freechipsproject/rocket-chip
- Out-of-order BOOM core
  - https://github.com/ucb-bar/riscv-boom
- UCB Sodor cores for education (single cycle, and 1-5 stages pipeline)
  - https://github.com/ucb-bar/riscv-sodor

RISC-V Implementations

- A list from
  - https://riscv.org/risc-v-cores/
- The Indian IIT-Madras is developing six RISC-V open-source CPU designs (SHAKTI) for six distinct usages
  - https://shaktiproject.bitbucket.io/index.html

- SiFive HiFive Unleashed
  - First Linux RISC-V Board
    - First shipment: June 2018
  - https://www.sifive.com/
  - https://github.com/sifive/freedom



Additional Information

Calling Convention

- C Datatypes and Alignment
  - RV32 employs an ILP32 integer model, while RV64 is LP64
  - Floating-point types are IEEE 754-2008 compatible
  - All of the data types are keeped naturally aligned when stored in memory
  - char is implicitly unsigned
  - In RV64, 32-bit types, such as int, are stored in integer registers as proper sign extensions of their 32-bit values; that is, bits 63..31 are all equal
    - This restriction holds even for unsigned 32-bit types

C type	Description	Bytes in RV32	Bytes in RV64
`char`	Character value/byte	1	1
`short`	Short integer	2	2
`int`	Integer	4	4
`long`	Long integer	4	8
`long long`	Long long integer	8	8
`void*`	Pointer	4	8
`float`	Single-precision float	4	4
`double`	Double-precision float	8	8
`long double`	Extended-precision float	16	16

Calling Convention

- RVG Calling Convention
  - If the arguments to a function are conceptualized as fields of a C struct, each with pointer alignment, the argument registers are a shadow of the first eight pointer-words of that struct
    - Floating-point arguments that are part of unions or array fields of structures are passed in integer registers
    - Floating-point arguments to variadic functions (except those that are explicitly named in the parameter list) are passed in integer registers
  - The portion of the conceptual struct that is not passed in argument registers is passed on the stack
    - The stack pointer sp points to the first argument not passed in a register
  - Arguments smaller than a pointer-word are passed in the least-significant bits of argument registers
  - When primitive arguments twice the size of a pointer-word are passed on the stack, they are naturally aligned
    - When they are passed in the integer registers, they reside in an aligned even-odd register pair, with the even register holding the least-significant bits
  - Arguments more than twice the size of a pointer-word are passed by reference

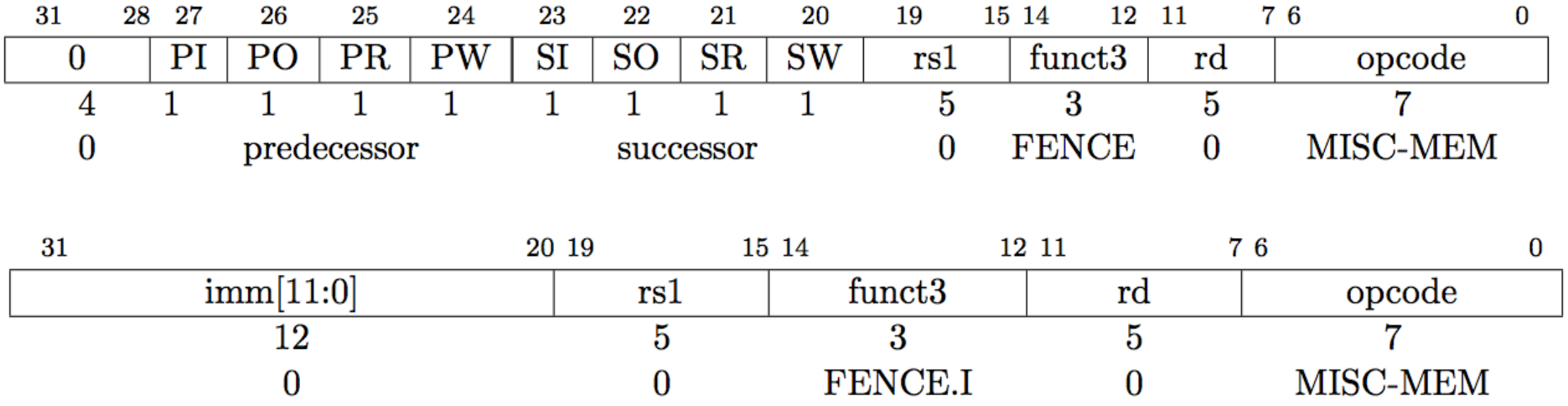
Calling Convention

- The stack grows downward and the stack pointer is always kept 16-byte aligned
- Values are returned from functions in integer registers v0 and v1 and floating-point registers fv0 and fv1
  - Floating-point values are returned in floating-point registers only if they are primitives or members of a struct consisting of only one or two floating-point values
  - Other return values that fit into two pointer-words are returned in v0 and v1
  - Larger return values are passed entirely in memory; the caller allocates this memory region and passes a pointer to it as an implicit first parameter to the callee

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	s0/fp	Saved register/frame pointer	Callee
x3–13	s1–11	Saved registers	Callee
x14	sp	Stack pointer	Callee
x15	tp	Thread pointer	Callee
x16–17	v0–1	Return values	Caller
x18–25	a0–7	Function arguments	Caller
x26–30	t0–4	Temporaries	Caller
x31	gp	Global pointer	—
f0–15	fs0–15	FP saved registers	Callee
f16–17	fv0–1	FP return values	Caller
f18–25	fa0–7	FP arguments	Caller
f26–31	ft0–5	FP temporaries	Caller

Memory Model

- RISC-V: Relaxed memory model



Control and Status Register (CSR) Instructions

- CSR Instructions

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	zimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	zimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	zimm[4:0]	CSRRCI	dest	SYSTEM

- Timer and counters

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
RDCYCLE[H]	0	CSRRS	dest	SYSTEM
RDTIME[H]	0	CSRRS	dest	SYSTEM
RDINSTRET[H]	0	CSRRS	dest	SYSTEM

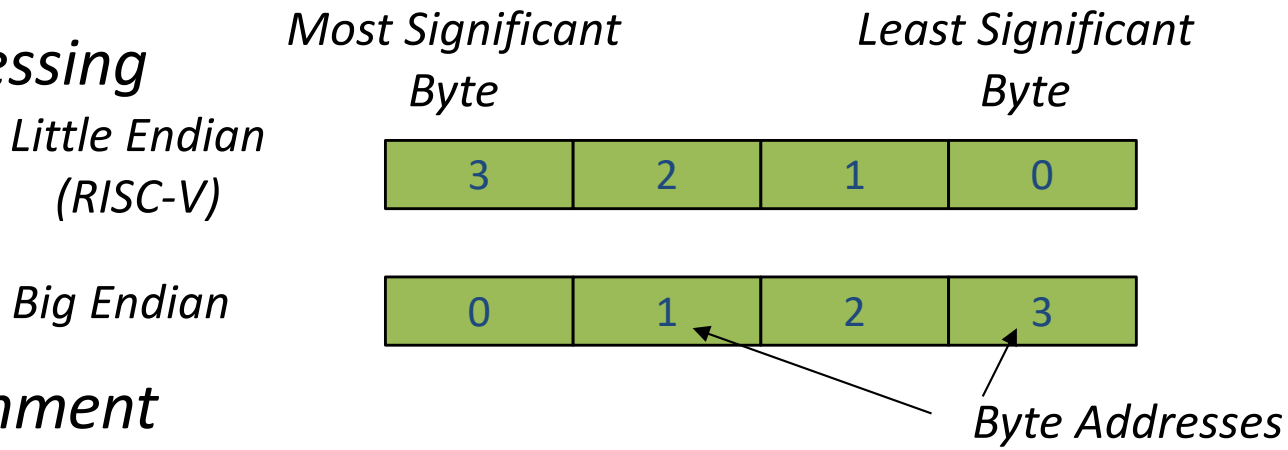
Data Formats and Memory Addresses

Data formats:

8-b Bytes, 16-b Half words, 32-b words and 64-b double words

Some issues

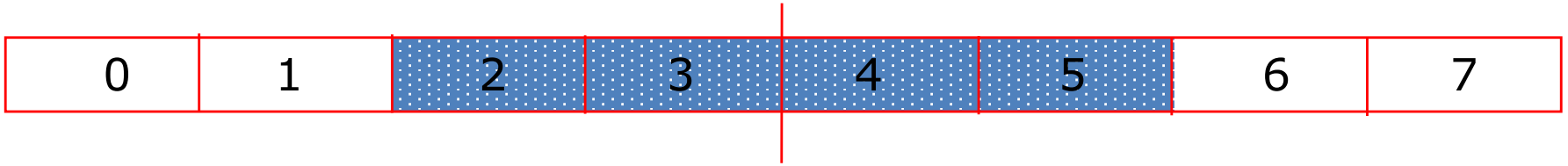
- *Byte addressing*



- *Word alignment*

  Suppose the memory is organized in 32-bit words.

  Can a word address begin only at 0, 4, 8, ... ?



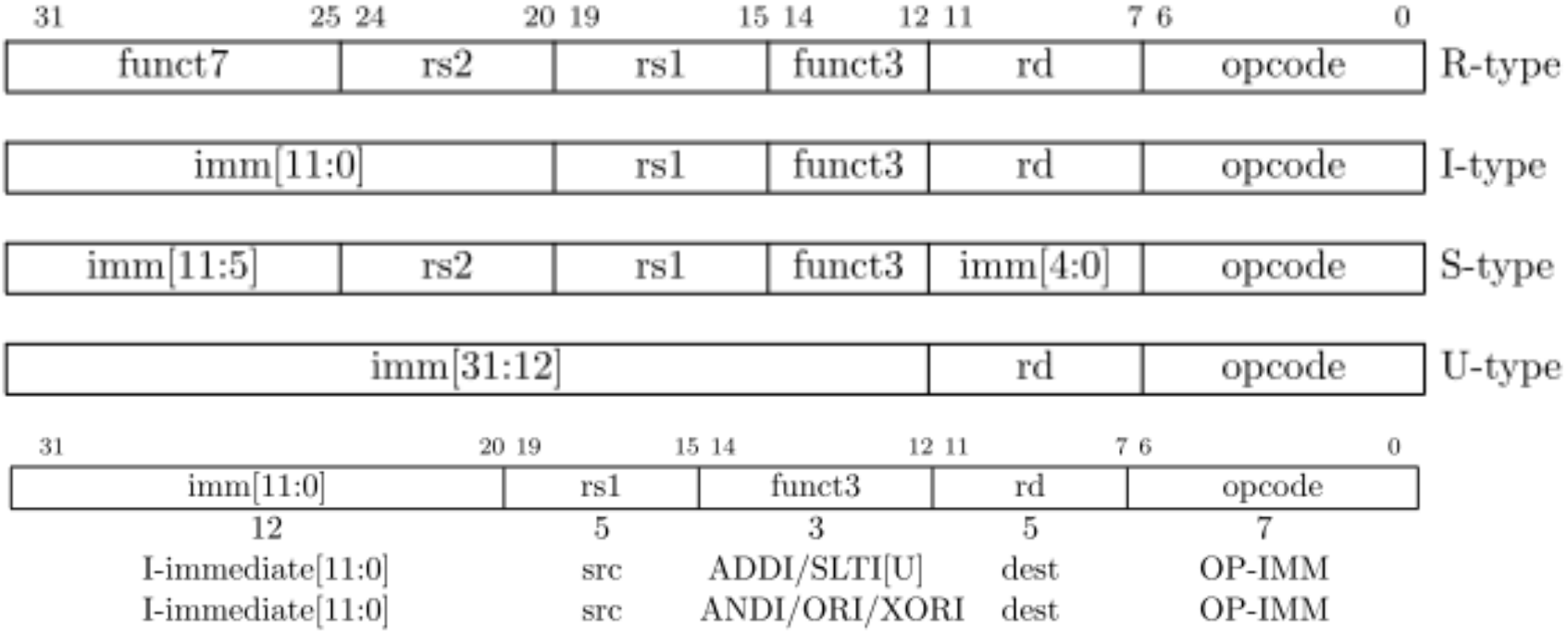
ISA Design

- RISC-V has 32 integer registers and can have 32 floating-point registers
  - Register number 0 is a constant 0
  - Register number 1 is the return address (link register)
- The memory is addressed by 8-bit bytes
- The instructions must be aligned to 32-bit addresses
- Like many RISC designs, it is a "load-store" machine
  - The only instructions that access main memory are loads and stores
  - All arithmetic and logic operations occur between registers
- RISC-V can load and store 8 and 16-bit items, but it lacks 8 and 16-bit arithmetic, including comparison-and-branch instructions
- The 64-bit instruction set includes 32-bit arithmetic

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

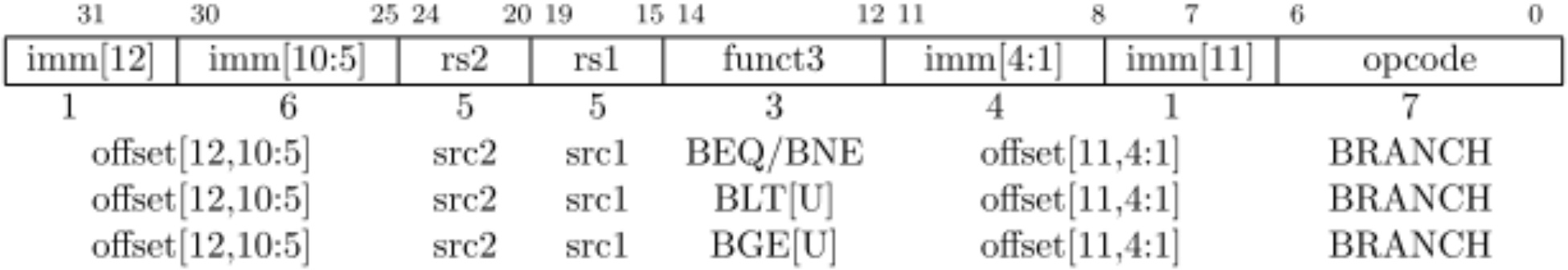
ISA Design for Performance

- Features to increase a computer's speed, while reducing its cost and power usage
  - placing most-significant bits at a fixed location to speed sign-extension, and a bit-arrangement designed to reduce the number of multiplexers in a CPU



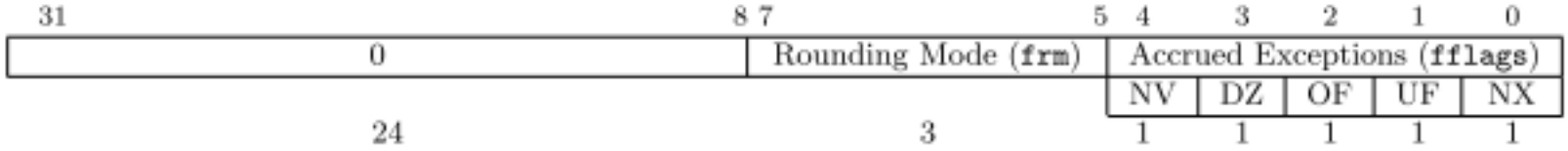
ISA Design

- Intentionally lacks condition codes, and even lacks a carry bit
  - To simplify CPU designs by minimizing interactions between instructions
- Builds comparison operations into its conditional-jumps



ISA Design

- The lack of a carry bit complicates multiple-precision arithmetic
  - GMP, MPFR
- Does not detect or flag most arithmetic errors, including overflow, underflow and divide by zero
  - No special instruction set support for overflow checks on integer arithmetic operations.
    - Most popular programming languages do not support checks for integer overflow, partly because most architectures impose a significant runtime penalty to check for overflow on integer arithmetic and partly because modulo arithmetic is sometimes the desired behavior
  - Floating-Point Control and Status Register



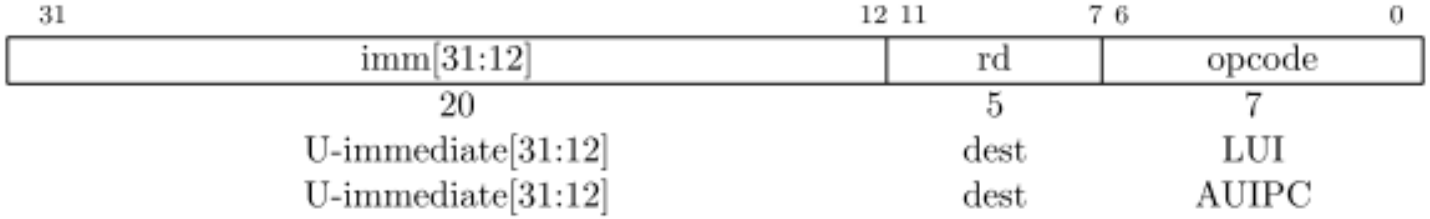
Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

ISA Design

- Lacks the "count leading zero" and bit-field operations normally used to speed software floating-point in a pure-integer processor
- No branch delay slot, a position after a branch instruction that can be filled with an instruction which is executed regardless of whether the branch is taken or not
  - This feature can improve performance of pipelined processors,
  - Omitted in RISC-V because it complicates both multicycle CPUs and superscalar CPUs
- Lacks address-modes that "write back" to the registers
  - For example, it does not do auto-incrementing

ISA Design

- A load or store can add a twelve-bit signed offset to a register that contains an address. A further 20 bits (yielding a 32-bit address) can be generated at an absolute address
  - RISC-V was designed to permit position-independent code. It has a special instruction to generate 20 upper address bits that are relative to the program counter. The lower twelve bits are provided by normal loads, stores and jumps



  - *LUI (load upper immediate) places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros*
  - *AUIPC (add upper immediate to pc) is used to build pc-relative addresses, forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd*