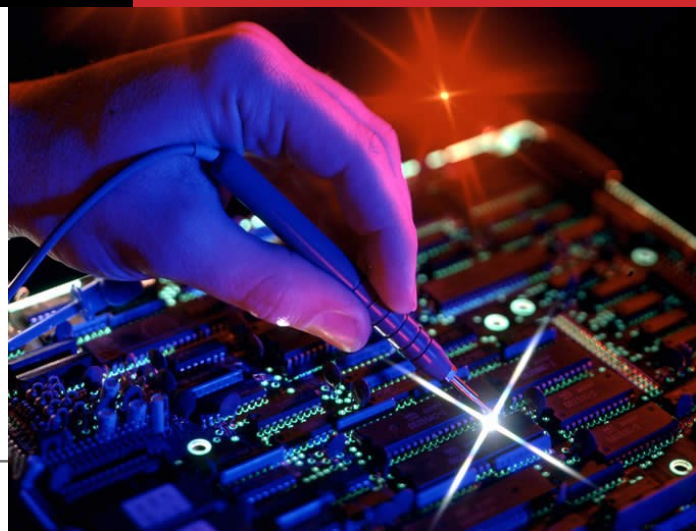
Advanced Computer Architecture

INTRODUCTION TO RISC-V

Dennis A. N. Gookyi

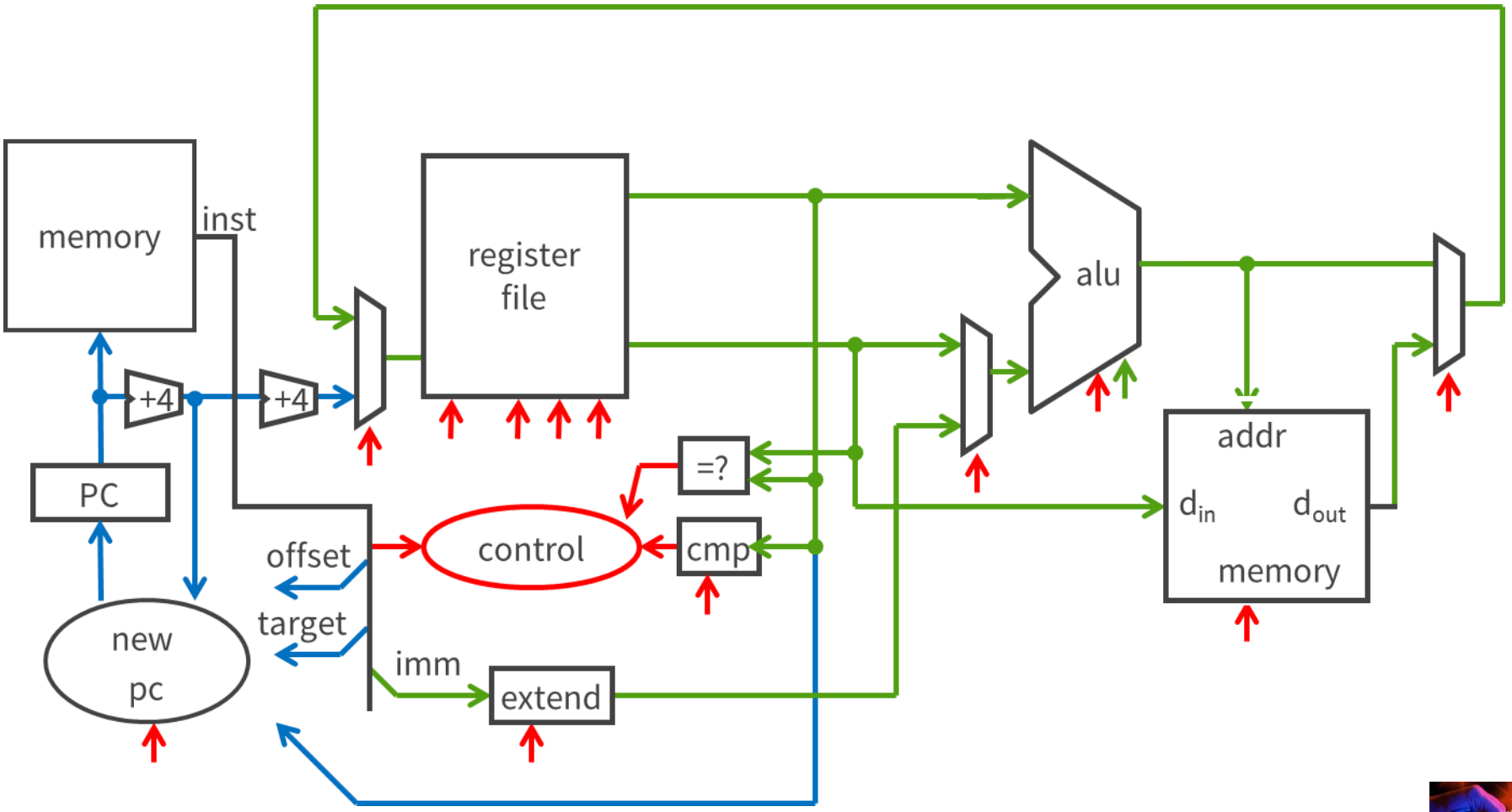


CONTENTS

- Introduction to RISC-V

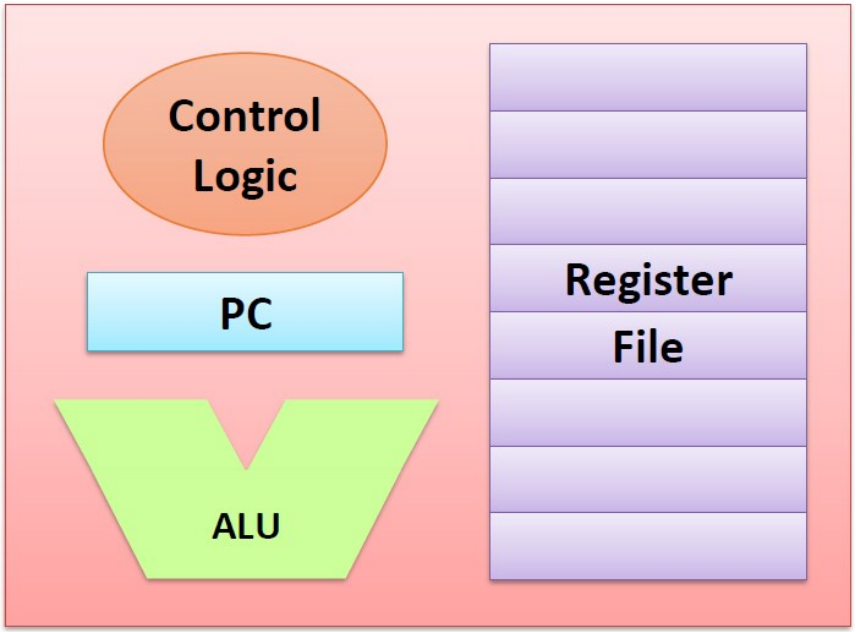
BIG PICTURE: BUILDING A PROCESSOR

- Single cycle processor



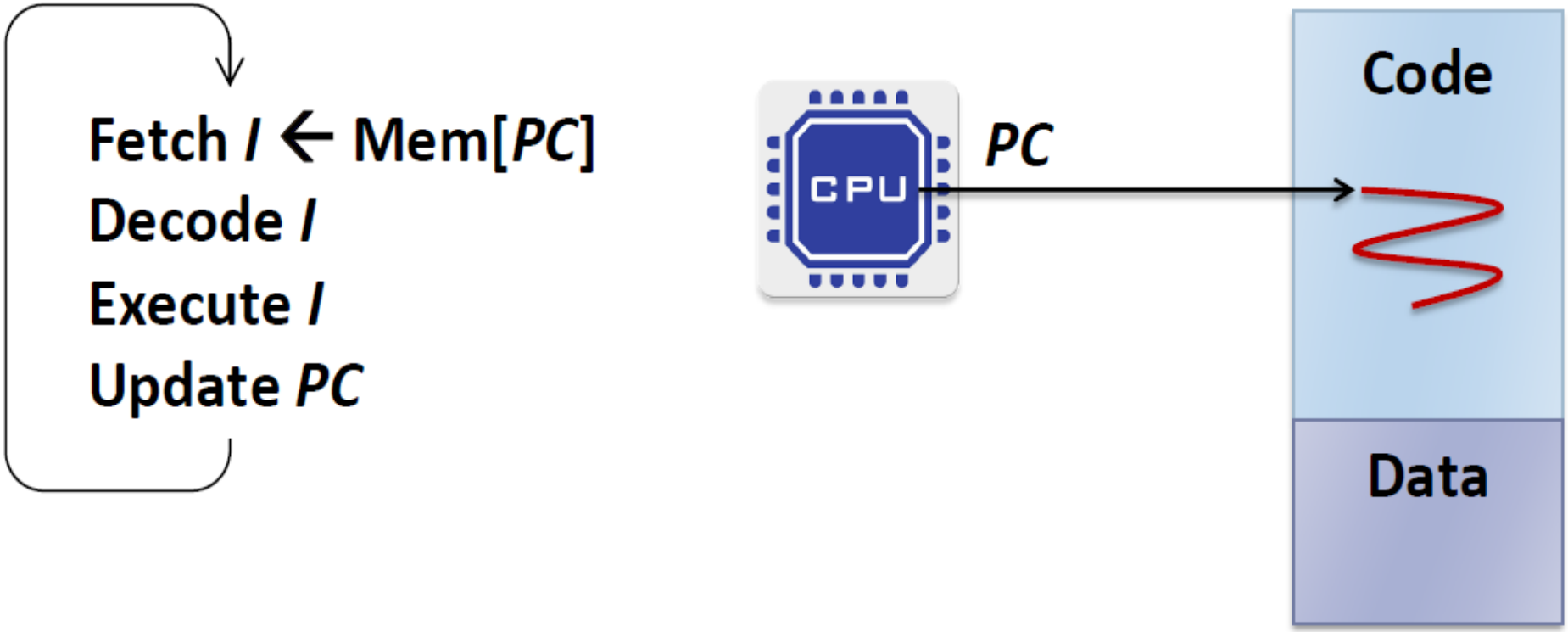
CPU

- Central Processing Unit
  - PC (Program Counter)
    - Address of next instruction
  - Register file
    - Heavily used program data
  - ALU (Arithmetic and Logic Unit)
    - Arithmetic operations
    - Logical operations
    - Control logic
    - Control instruction fetch, decoding, and execution



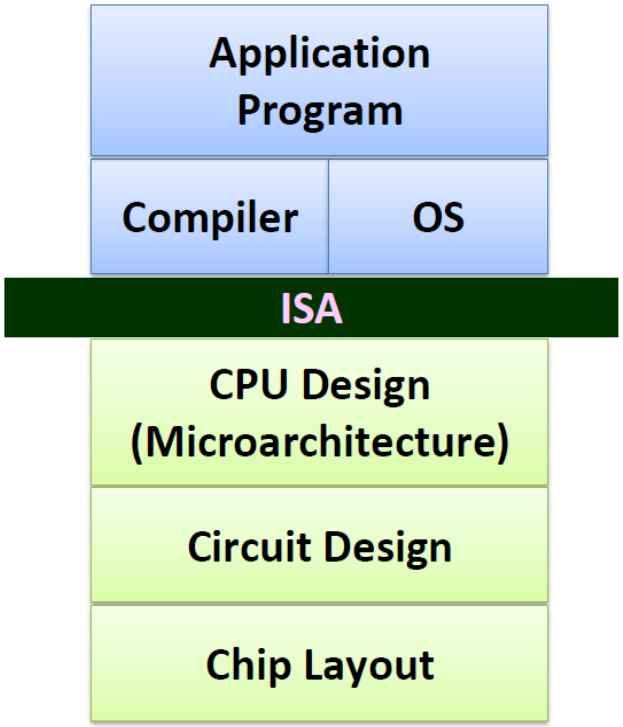
CPU

- The life of a CPU



INSTRUCTION SET ARCHITECTURE (ISA)

- Above: How to program a machine
  - Processors execute instructions in sequence
- Below: What needs to be built
  - Use a variety of tricks to make it run fast
- Instruction set
- Processor registers
- Memory addressing modes
- Data types and representations
- Byte ordering



INSTRUCTION SET ARCHITECTURE (ISA)

- Mainstream ISAs

x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

Macbooks & PCs
(Core i3, i5, i7, M)
x86 Instruction Set

ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)

Smartphone-like devices
(iPhone, Android), Raspberry Pi, Embedded systems
ARM Instruction Set

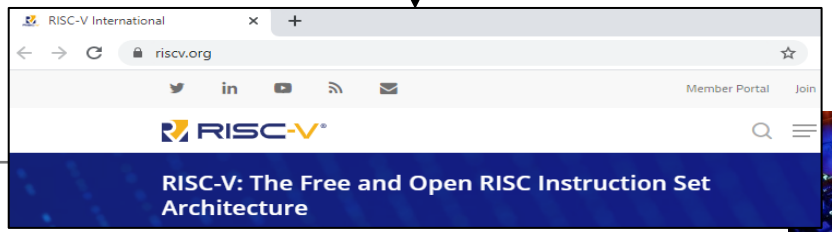
RISC-V

Designer	University of California, Berkeley
Bits	32, 64, 128
Introduced	2010
Version	2.2
Design	RISC
Type	Load-store
Encoding	Variable
Branching	Compare-and-branch
Endianness	Little

Versatile and open-source
Relatively new, designed for cloud computing, embedded systems, academic use
RISC V Instruction Set

THE RISC-V INSTRUCTION SET

- A completely open ISA that is freely available to academia and industry
- Fifth RISC ISA design developed at UC Berkeley
  - RISC-I (1981), RISC-II (1983), SOAR (1984), SPUR (1989), and RISC-V (2010)
- Now managed by the RISC-V Foundation (http://riscv.org)
- Typical of many modern ISAs
  - See RISC-V Reference Card (or Green Card)
- Similar ISAs have a large share of the embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers



FREE OPEN ISA ADVANTAGES

- Greater innovation via free-market competition
  - From many core designers, closed-source and open-source
- Shared open-core designs
  - Shorter time to market, lower cost from reuse, fewer errors given more eyeballs, transparency makes it difficult for government agencies to add secret trap doors
- Processors becoming affordable for more devices
  - Help expand the Internet of Things (IoTs), which could cost as little as $1
- Software stack survives for a long time
- Make architectural research and education more real
  - Fully open hardware and software stacks

RISC-V ISAS

- Three base integer ISAs, one per address width
  - RV32I, RV64I, RV128I
  - RV32I: Only 40 instructions defined
  - RV32E: Reduced version of RV32I with 16 registers for embedded systems
- Standard extensions
- Standard RISC encoding in a fixed 32-bit instruction format
- C extension offers shorter 16-bit versions of common 32-bit RISC-V instructions (can be intermixed with 32-bit instructions)

Name	Extension
M	Integer Multiply/Divide
A	Atomic Instructions
F	Single-precision FP
D	Double-precision FP
G	General-purpose (= IMAFD)
Q	Quad-precision FP
C	Compressed Instructions

RISC-V ISA

- The RISC V "Green Card"

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	`LB rd,rs1,imm`	
	Load Halfword	I	`LH rd,rs1,imm`	
	Load Word	I	`LW rd,rs1,imm`	`L{D|Q} rd,rs1,imm`
	Load Byte Unsigned	I	`LBU rd,rs1,imm`	
	Load Half Unsigned	I	`LHU rd,rs1,imm`	`L{W|D}U 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`	`S{D|Q} rs1,rs2,imm`
Shifts	Shift Left	R	`SLL rd,rs1,rs2`	`SLL{W|D} rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI rd,rs1,shamt`	`SLLI{W|D} rd,rs1,shamt`
	Shift Right	R	`SRL rd,rs1,rs2`	`SRL{W|D} rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI rd,rs1,shamt`	`SRLI{W|D} rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA rd,rs1,rs2`	`SRA{W|D} rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI rd,rs1,shamt`	`SRAI{W|D} rd,rs1,shamt`
Arithmetic	ADD	R	`ADD rd,rs1,rs2`	`ADD{W|D} rd,rs1,rs2`
	ADD Immediate	I	`ADDI rd,rs1,imm`	`ADDI{W|D} rd,rs1,imm`
	SUBtract	R	`SUB rd,rs1,rs2`	`SUB{W|D} 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 =	SB	`BEQ rs1,rs2,imm`	
	Branch ≠	SB	`BNE rs1,rs2,imm`	
	Branch <	SB	`BLT rs1,rs2,imm`	
	Branch ≥	SB	`BGE rs1,rs2,imm`	
	Branch < Unsigned	SB	`BLTU rs1,rs2,imm`	
	Branch ≥ Unsigned	SB	`BGEU rs1,rs2,imm`	
Jump & Link	J&L	UJ	`JAL rd,imm`	
	Jump & Link Register	UJ	`JALR rd,rs1,imm`	
Synch	Synch thread	I	`FENCE`	
	Synch Instr & Data	I	`FENCE.I`	
System	System CALL	I	`SCALL`	
	System BREAK	I	`SBREAK`	
Counters	ReaD CYCLE	I	`RDCYCLE rd`	
	ReaD CYCLE upper Half	I	`RDCYCLEH rd`	
	ReaD TIME	I	`RDTIME rd`	
	ReaD TIME upper Half	I	`RDTIMEH rd`	
	ReaD INSTR RETired	I	`RDINSTRET rd`	
	ReaD INSTR upper Half	I	`RDINSTRETH rd`	

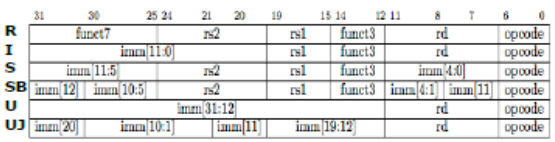
RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	`CSRRW rd,csr,rs1`
	Atomic Read & Set Bit	`CSRRS rd,csr,rs1`
	Atomic Read & Clear Bit	`CSRRC rd,csr,rs1`
	Atomic R/W Imm	`CSRRWI rd,csr,imm`
	Atomic Read & Set Bit Imm	`CSRRSI rd,csr,imm`
	Atomic Read & Clear Bit Imm	`CSRRCI rd,csr,imm`
Change Level	Env. Call	`ECALL`
	Environment Breakpoint	`EBREAK`
	Environment Return	`ERET`
Trap Redirect	to Supervisor	`MRTS`
	Redirect Trap to Hypervisor	`MRTH`
	Hypervisor Trap to Supervisor	`HRTS`
Interrupt	Wait for Interrupt	`WFI`
MMU	Supervisor FENCE	`SFENCE.VM rs1`

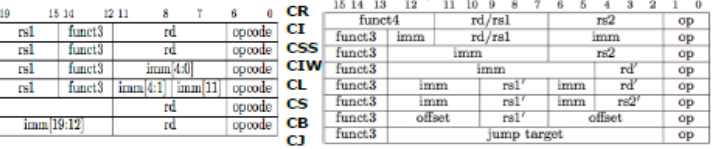
Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI 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`
	Load Double	CL	`C.LD rd',rs1',imm`	`LD rd',rs1',imm*8`
	Load Double SP	CI	`C.LDSP rd,imm`	`LD rd,sp,imm*8`
	Load Quad	CL	`C.LQ rd',rs1',imm`	`LQ rd',rs1',imm*16`
	Load Quad SP	CI	`C.LQSP rd,imm`	`LQ 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`
	Store Double	CS	`C.SD rs1',rs2',imm`	`SD rs1',rs2',imm*8`
	Store Double SP	CSS	`C.SDSP rs2,imm`	`SD rs2,sp,imm*8`
	Store Quad	CS	`C.SQ rs1',rs2',imm`	`SQ rs1',rs2',imm*16`
	Store Quad SP	CSS	`C.SQSP rs2,imm`	`SQ rs2,sp,imm*16`
Arithmetic	ADD	CR	`C.ADD rd,rs1`	`ADD rd,rd,rs1`
	ADD Word	CR	`C.ADDW rd,rs1`	`ADDW rd,rd,imm`
	ADD Immediate	CI	`C.ADDI rd,imm`	`ADDI rd,rd,imm`
	ADD Word Imm	CI	`C.ADDIW rd,imm`	`ADDIW 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`
	Load Immediate	CI	`C.LI rd,imm`	`ADDI rd,x0,imm`
	Load Upper Imm	CI	`C.LUI rd,imm`	`LUI rd,imm`
	MoVe	CR	`C.MV rd,rs1`	`ADD rd,rs1,x0`
	SUB	CR	`C.SUB rd,rs1`	`SUB rd,rd,rs1`
Shifts	Shift Left Imm	CI	`C.SLLI rd,imm`	`SLLI 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`

32-bit Instruction Formats



16-bit (RVC) Instruction Formats



RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

RISC-V ISA

- RV32I Base ISA

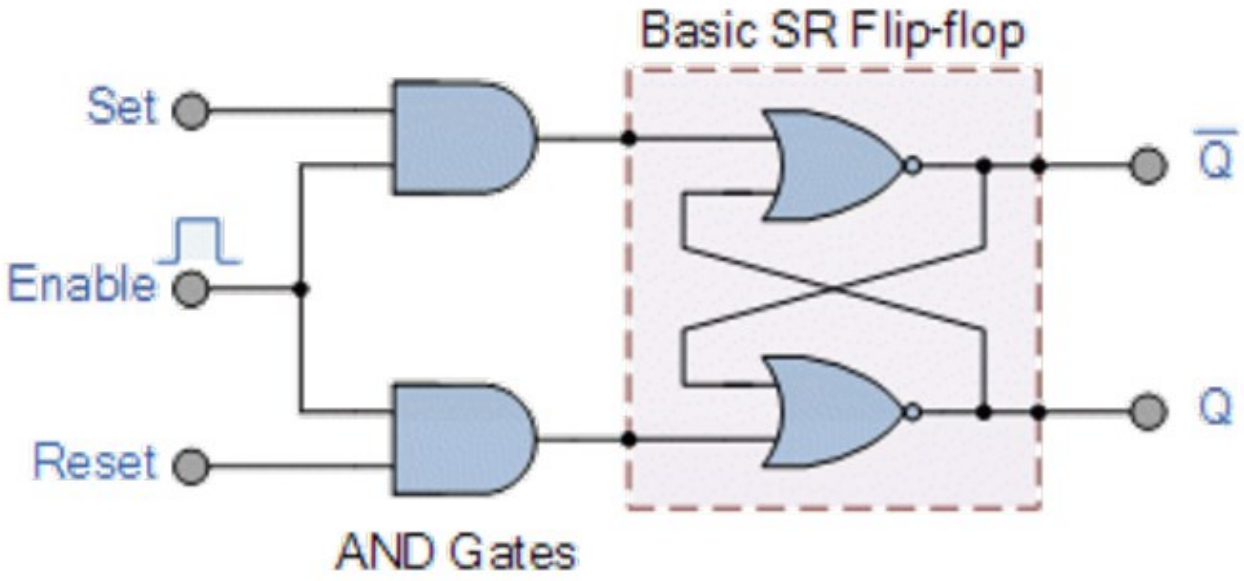
31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]				rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2		rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2		rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2		rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2		rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2		rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2		rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]				rs1	000	rd	0000011	LB
imm[11:0]				rs1	001	rd	0000011	LH
imm[11:0]				rs1	010	rd	0000011	LW
imm[11:0]				rs1	100	rd	0000011	LBU
imm[11:0]				rs1	101	rd	0000011	LHU
imm[11:5]		rs2		rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2		rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2		rs1	010	imm[4:0]	0100011	SW
imm[11:0]				rs1	000	rd	0010011	ADDI
imm[11:0]				rs1	010	rd	0010011	SLTI
imm[11:0]				rs1	011	rd	0010011	SLTIU
imm[11:0]				rs1	100	rd	0010011	XORI
imm[11:0]				rs1	110	rd	0010011	ORI
imm[11:0]				rs1	111	rd	0010011	ANDI
0000000		shamt		rs1	001	rd	0010011	SLLI
0000000		shamt		rs1	101	rd	0010011	SRLI
0100000		shamt		rs1	101	rd	0010011	SRAI
0000000		rs2		rs1	000	rd	0110011	ADD
0100000		rs2		rs1	000	rd	0110011	SUB
0000000		rs2		rs1	001	rd	0110011	SLL
0000000		rs2		rs1	010	rd	0110011	SLT
0000000		rs2		rs1	011	rd	0110011	SLTU
0000000		rs2		rs1	100	rd	0110011	XOR
0000000		rs2		rs1	101	rd	0110011	SRL
0100000		rs2		rs1	101	rd	0110011	SRA
0000000		rs2		rs1	110	rd	0110011	OR
0000000		rs2		rs1	111	rd	0110011	AND
0000	pred	succ		00000	000	00000	0001111	FENCE
0000	0000	0000		00000	001	00000	0001111	FENCE.I
000000000000				00000	000	00000	1110011	ECALL
000000000001				00000	000	00000	1110011	EBREAK
csr				rs1	001	rd	1110011	CSRRW
csr				rs1	010	rd	1110011	CSRRS
csr				rs1	011	rd	1110011	CSRRC
csr				zimm	101	rd	1110011	CSRRWI
csr				zimm	110	rd	1110011	CSRRSI
csr				zimm	111	rd	1110011	CSRRCI

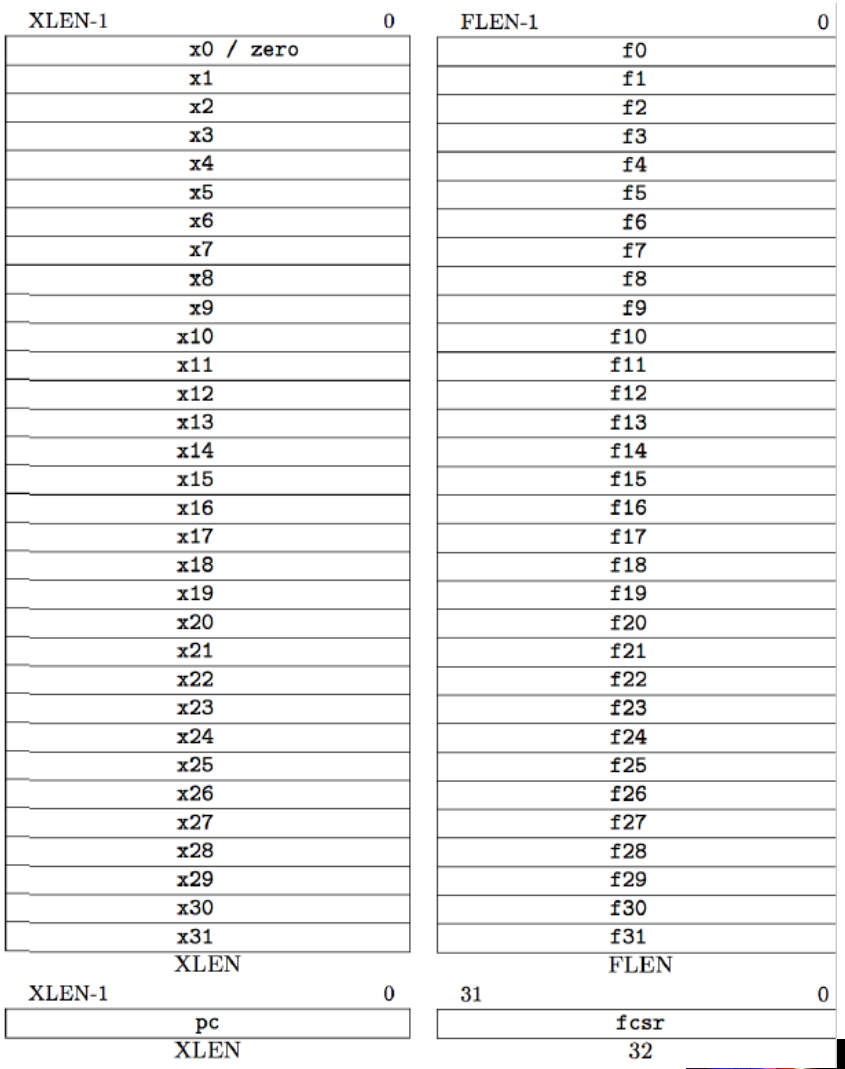
RISC-V REGISTERS

- Hardware uses registers for variables
- Registers are:
  - Small memories of a fixed size (32-bit in RV32I)
  - Can be read or written
  - Limited in number (32 registers in RISC V)
  - Very fast and low power to access



RISC-V REGISTERS

- Program counter (pc)
- 32 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)
  - Is an extension
- FP status register (fsr), used for FP rounding mode and exception reporting



RISC-V REGISTERS

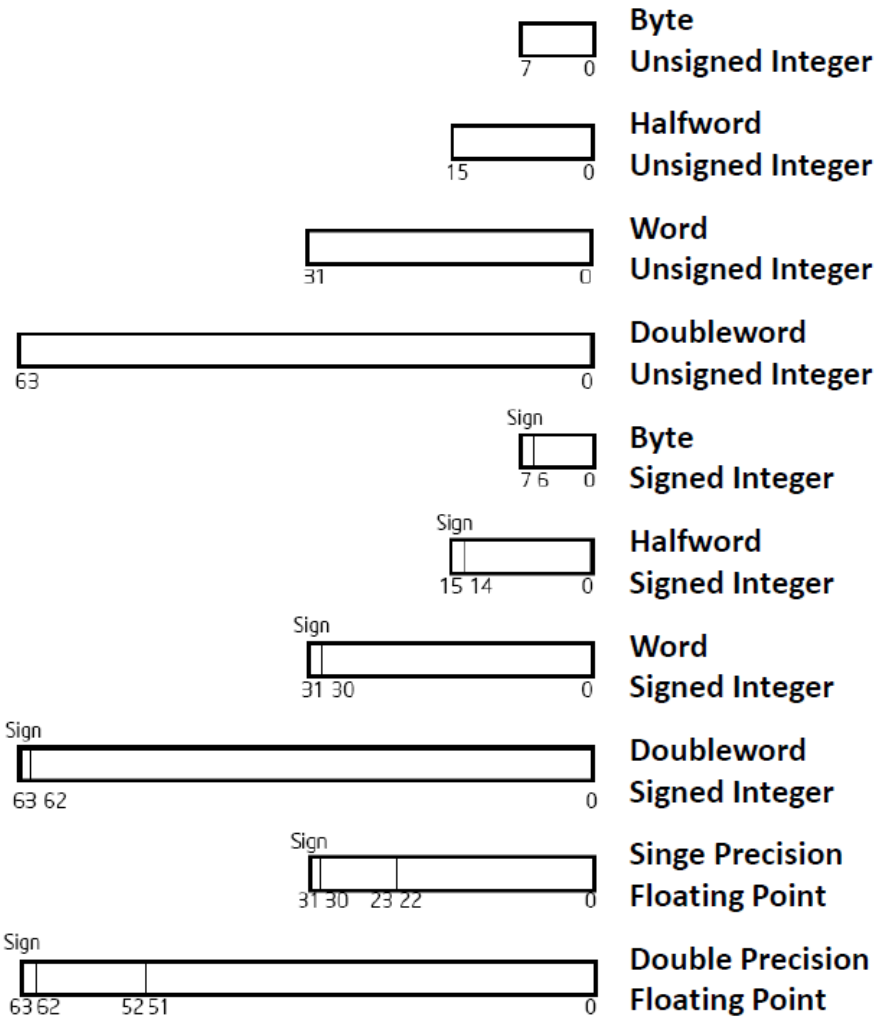
- Registers description

#	Name	Usage
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporaries (Caller-save registers)
x6	t1	
x7	t2	
x8	s0/fp	Saved register / Frame pointer
x9	s1	Saved register
x10	a0	Function arguments / Return values
x11	a1	
x12	a2	Function arguments
x13	a3	
x14	a4	
x15	a5	

#	Name	Usage
x16	a6	Function arguments
x17	a7	
x18	s2	Saved registers (Callee-save registers)
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s8	
x25	s9	
x26	s10	
x27	s11	
x28	t3	Temporaries (Caller-save registers)
x29	t4	
x30	t5	
x31	t6	
	pc	Program counter

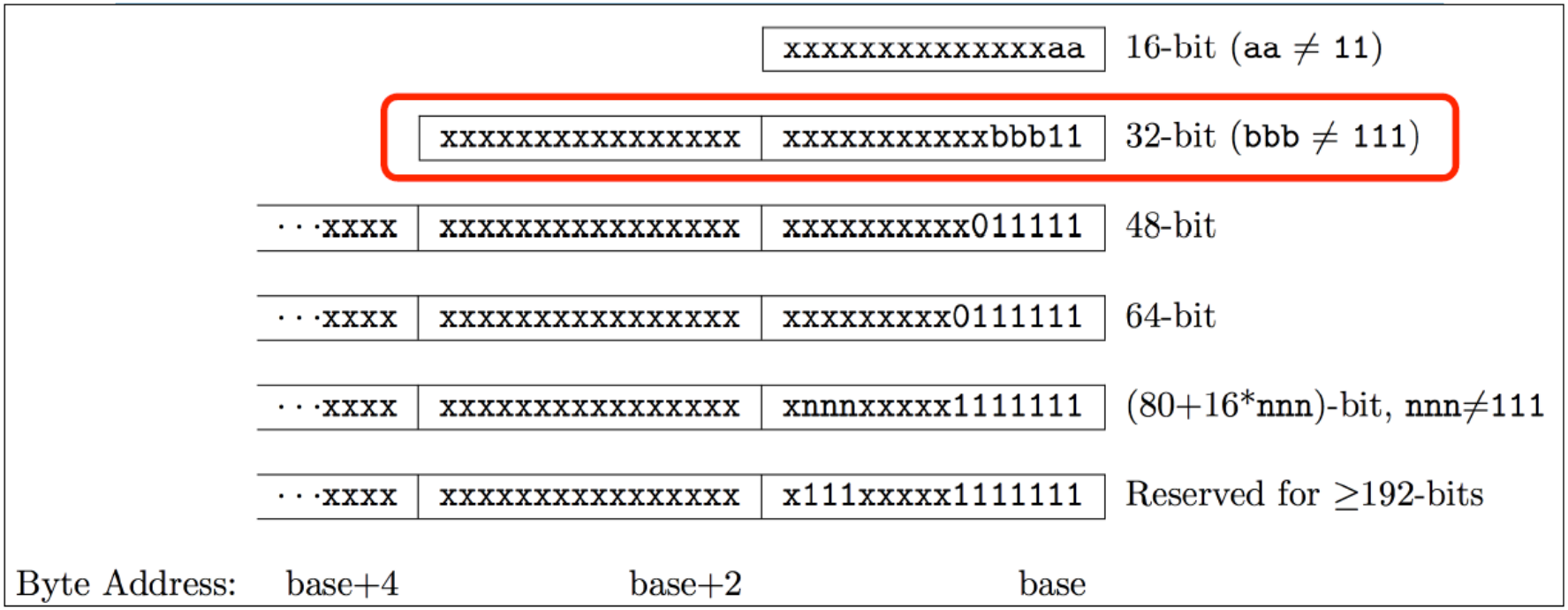
RISC-V DATA TYPES

- Integer data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4 or 8 bytes (with F or D extension)
- No aggregated types such as arrays or structures
  - Just contiguously allocated bytes in memory



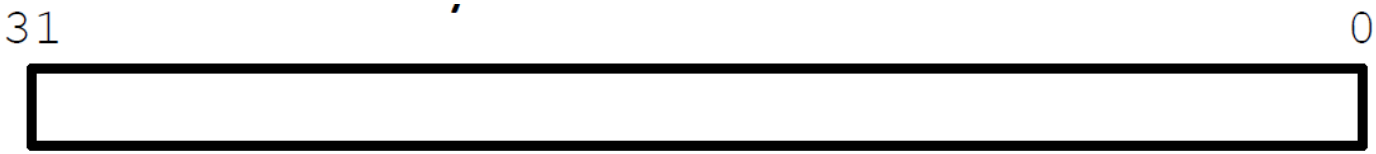
RISC-V INSTRUCTION ENCODING

- 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)



RISC-V INSTRUCTION ENCODING

- By convention, RISC-V instructions are each
  - 1 word = 4 bytes = 32 bits



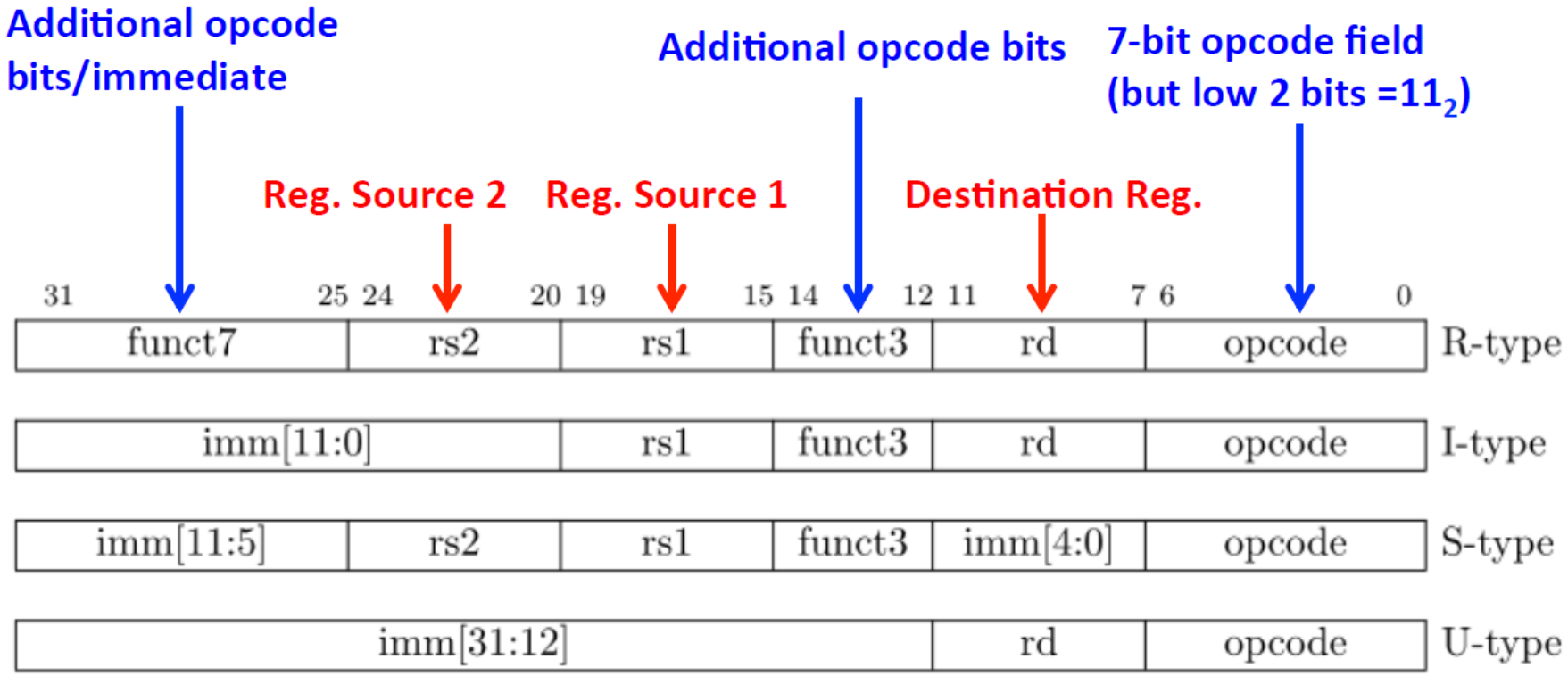
- Divide the 32 bits of instruction into "fields"
  - Regular field sizes → simpler hardware
  - Will need some variation
- Define 6 types of instruction formats:
  - R-Format
  - I-Format
  - S-Format
  - U-Format
  - SB-Format
  - UJ-Format

THE 6 INSTRUCTION FORMATS

- R-Format: instructions using 3 register inputs
  - Eg. add, xor, mul, arithmetic/logical ops
- I-Format: instructions with immediates, loads
  - Eg. addi, lw, jalr, slli
- S-Format: store instructions
  - Eg. sw, sb
- SB-Format: branch instructions
  - Eg. beq, bge
- U-Format: instructions with upper immediates
  - Eg. lui, auipc
  - upper immediate is 20-bits
- UJ-Format: the jump instruction
  - Eg. jal

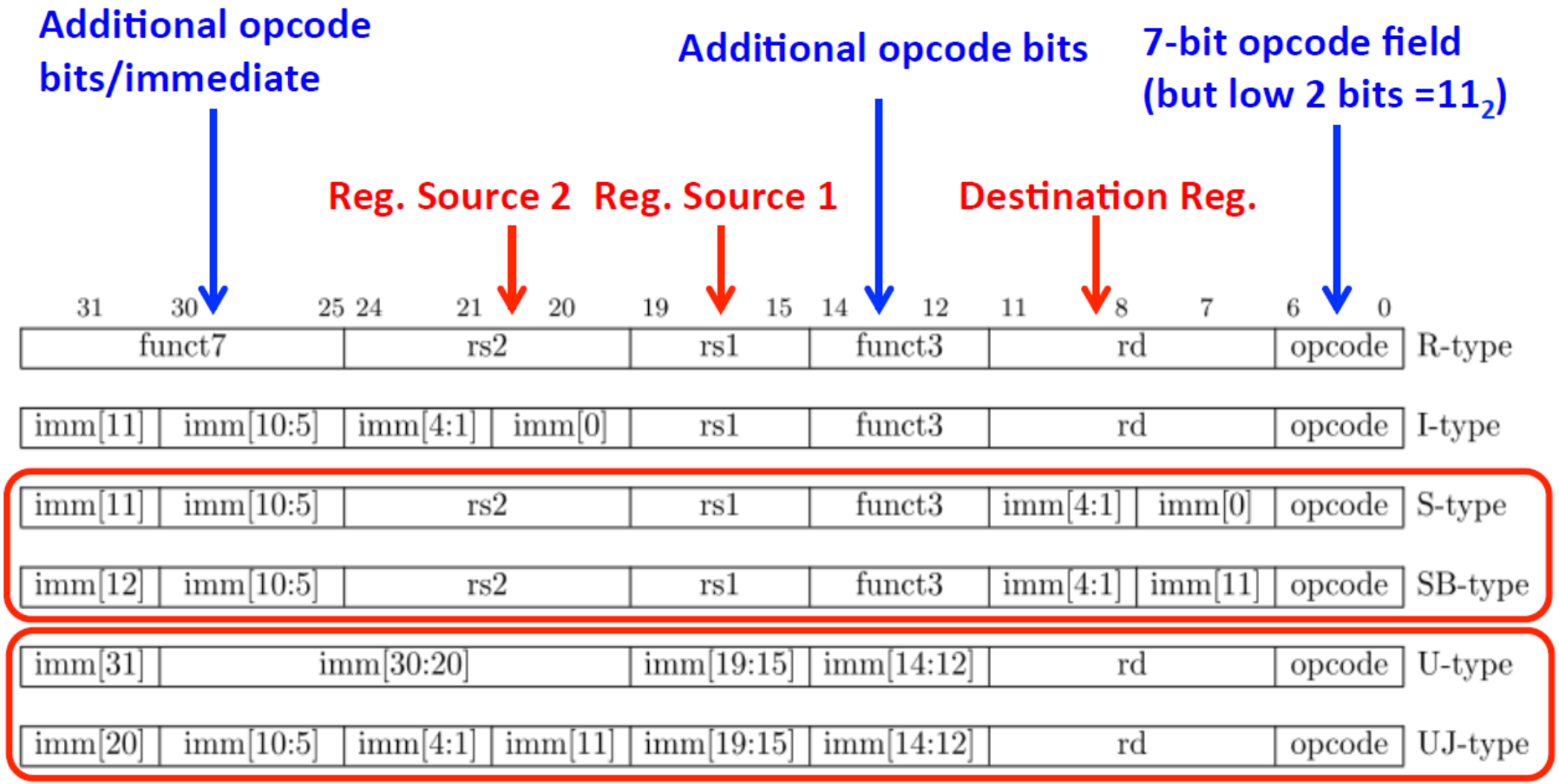
4 CORE RISC-V INSTRUCTION FORMATS

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



6 RISC-V INSTRUCTION FORMATS

- Variants



Based on the handling of the immediates

IMMEDIATE ENCODING VARIANTS

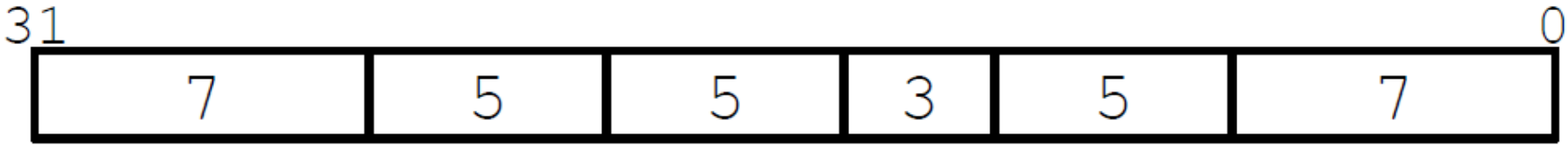
- Immediate produced by each base instruction format
  - Instruction bit (inst[y])

31 — 30 — 20	19 — 12	11	10 — 5	4 — 1	0	
— inst[31] —			inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —			inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —		inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]		— 0 —		U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

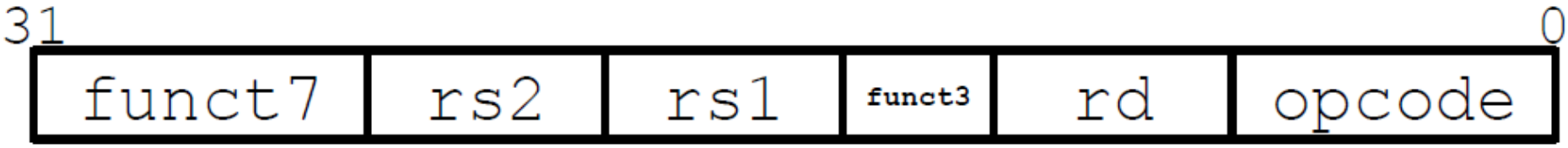
31 — 30	25 — 24	21 — 20	19 — 15	14 — 12	11 — 8	7	6 — 0	
funct7	rs2		rs1	funct3	rd		opcode	R-type
imm[11]	imm[10:5]	imm[4:1]	imm[0]	rs1	funct3	rd	opcode	I-type
imm[11]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[0]	opcode	S-type
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	SB-type
imm[31]	imm[30:20]		imm[19:15]	imm[14:12]	rd		opcode	U-type
imm[20]	imm[10:5]	imm[4:1]	imm[11]	imm[19:15]	imm[14:12]	rd	opcode	UJ-type

R-FORMAT INSTRUCTIONS

- Define "fields" of the following number of bits each:
  - 7 + 5 + 5 + 3 + 5 + 7 = 32

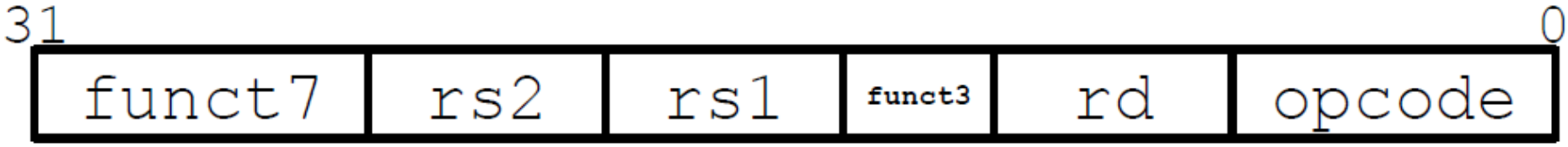


- Each field has a name:



- Each field is viewed as its own unsigned int
  - 5-bit fields can represent any number 0-31, while 7-bit fields can represent any number 0-128, etc

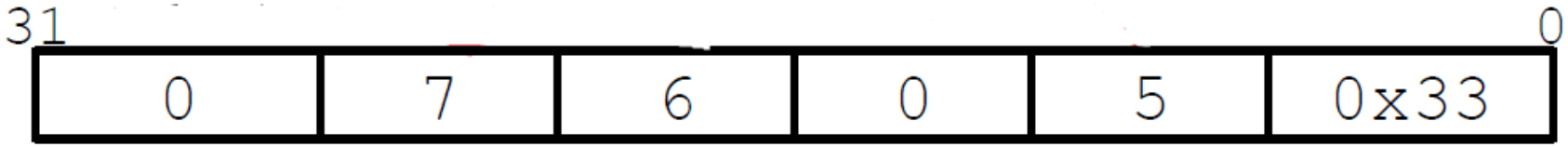
R-FORMAT INSTRUCTIONS



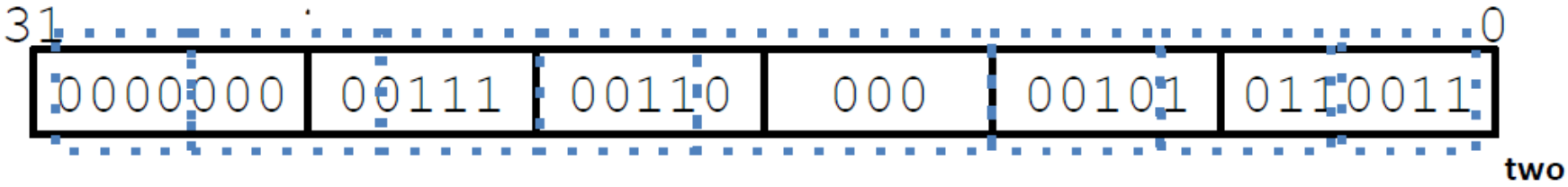
- opcode (7): partially specifies operation
- funct7+funct3 (10): combined with opcode, these two fields describe what operation to perform
- rs1 (5): 1st operand ("source register 1")
- rs2 (5): 2nd operand (second source register)
- rd (5): "destination register" — receives the result of the computation
- Recall: RISCV has 32 registers
  - A 5-bit field can represent exactly $2^5 = 32$ things (interpret as the register numbers x0-x31)

R-FORMAT INSTRUCTIONS

- R-Format example
  - RISC V Instructions: add x5, x6, x7
  - Field representation (decimal):



  - Field representation (binary):

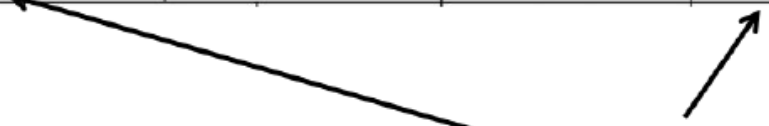


  - Hex representation: 0x 0073 02B3
  - Decimal representation: 7,537,331
    - Called a Machine Language Instruction

R-FORMAT INSTRUCTIONS

- All RV32 R-Format instructions

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND



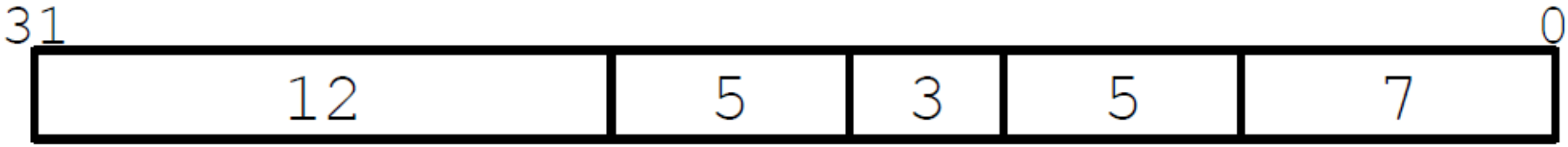
Different encoding in funct7 + funct3 selects different operations

I-FORMAT INSTRUCTIONS

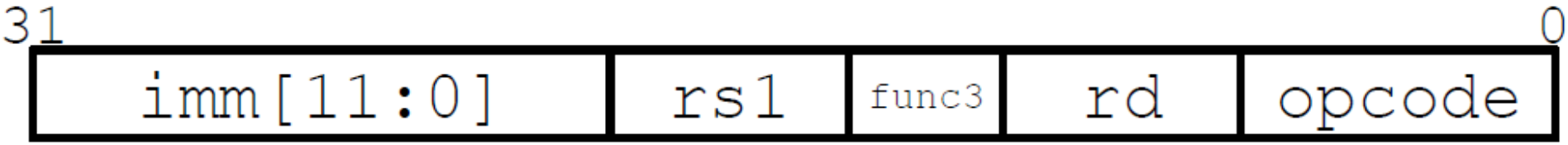
- What about instructions with immediates?
  - 5-bit field too small for most immediates
- Ideally, RISCV would have only one instruction format (for simplicity)
  - Unfortunately here we need to compromise
- Define new instruction format that is mostly consistent with R-Format
  - First notice that, if instruction has immediate, then it uses at most 2 registers (1 src, 1 dst)

I-FORMAT INSTRUCTIONS

- Define "fields" of the following number of bits each:
  - 12 + 5 + 3 + 5 + 7 = 32 bits

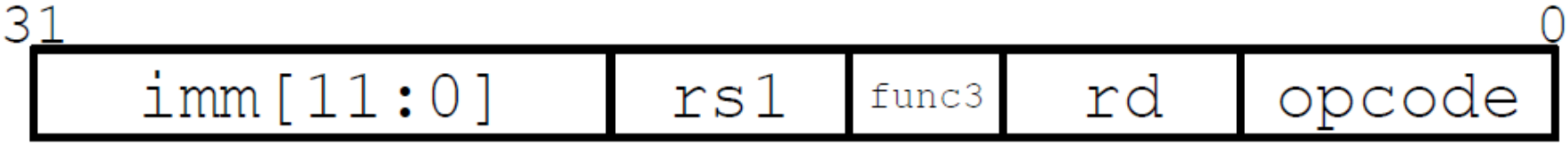


- Field names:



- Key Concept: Only imm field is different from R-format:
  - rs2 and funct7 replaced by 12-bit signed immediate, imm[11:0]

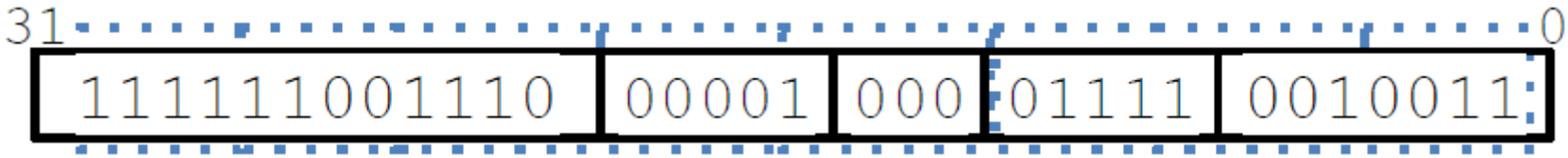
I-FORMAT INSTRUCTIONS



- opcode (7): uniquely specifies the instruction
- rs1 (5): specifies a register operand
- rd (5): specifies destination register that receives the result of the computation
- immediate (12): 12-bit number
  - All computations are done in words, so 12-bit immediate must be extended to 32-bits
  - Always sign-extended to 32-bits before use in an arithmetic operation
  - Can represent 2^{12} different immediates
  - imm[11:0] can hold values in range $[-2^{11}, +2^{11}]$

I-FORMAT INSTRUCTIONS

- I-Format example
  - RISCV Instruction: addi x15, x1, -50
  - Field representation (binary):



  - Hex representation: 0xFCE0 8793
  - Decimal representation: 4,242,573,203

I-FORMAT INSTRUCTIONS

- All RV32 I-Format instructions

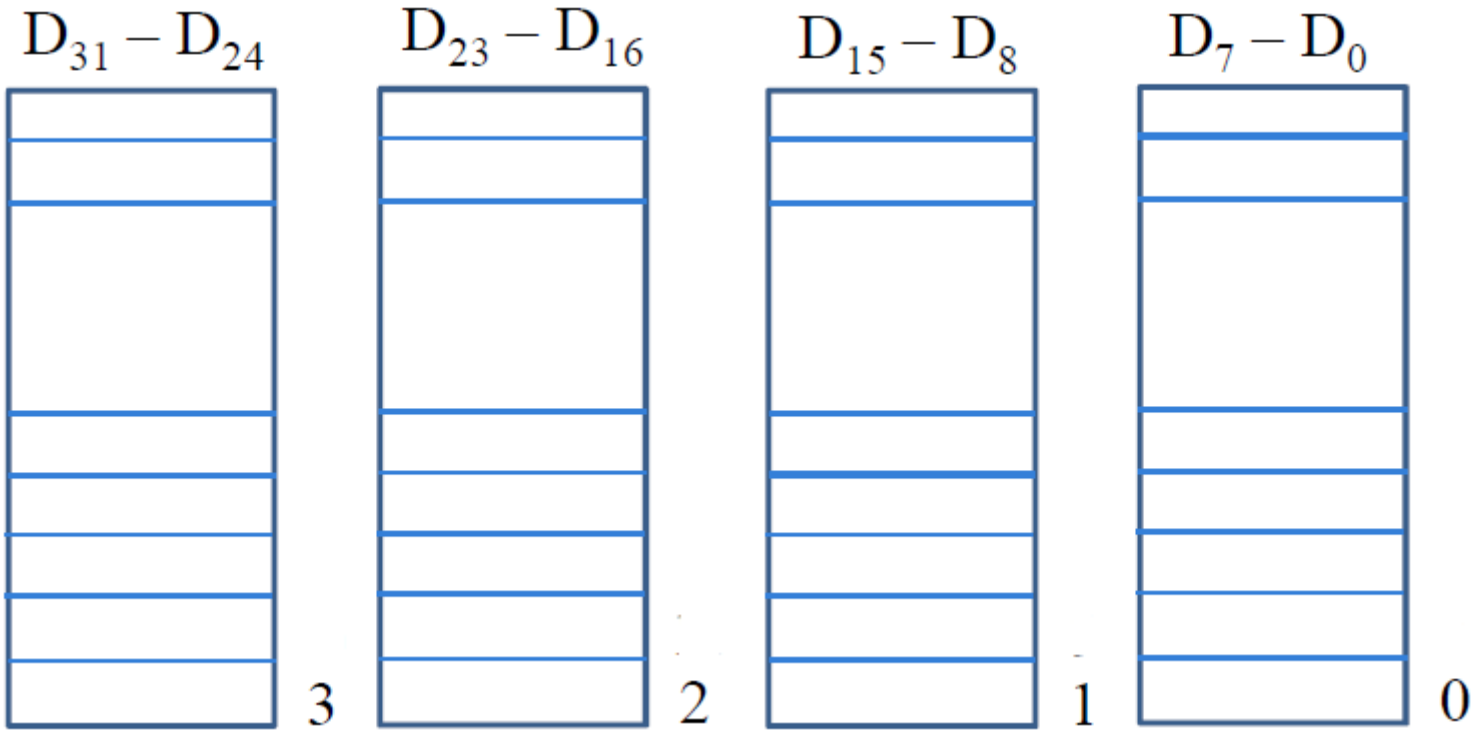
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

One of the higher-order immediate bits is used to distinguish "shift right logical" (SRLI) from "shift right arithmetic" (SRAI)

"Shift-by-immediate" instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

MEMORY OPERANDS

- RISC V uses byte addressing which means that each word requires 4 bytes
- When addressing consecutive words, memory address increments by 4

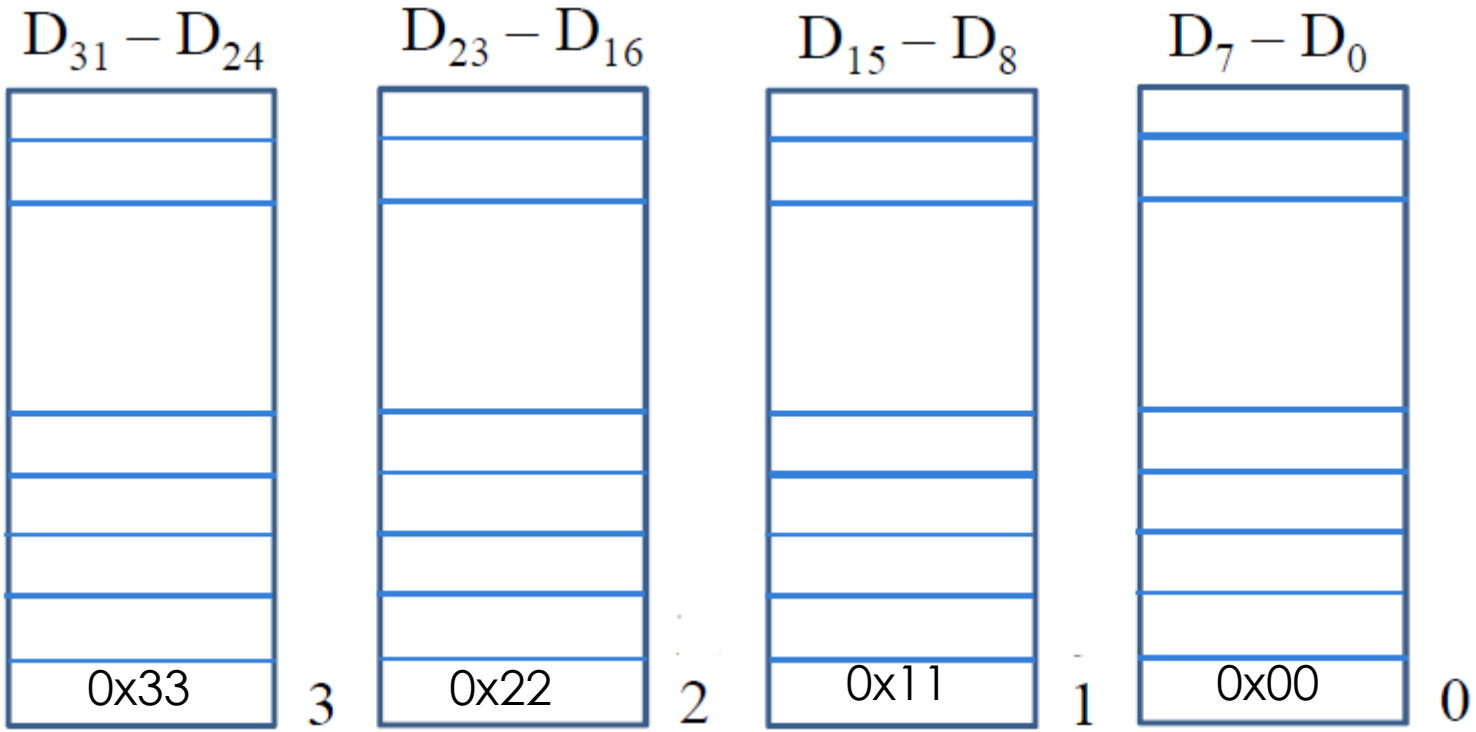


MEMORY OPERANDS (LITTLE VS BIG ENDIAN)

- Little Endian Byte Order:
  - The LSB of the data is placed at the byte with the lowest address
- Big Endian Byte Order:
  - The MSB of the data is placed at the byte with the lowest address
- RISC V is Little Endian

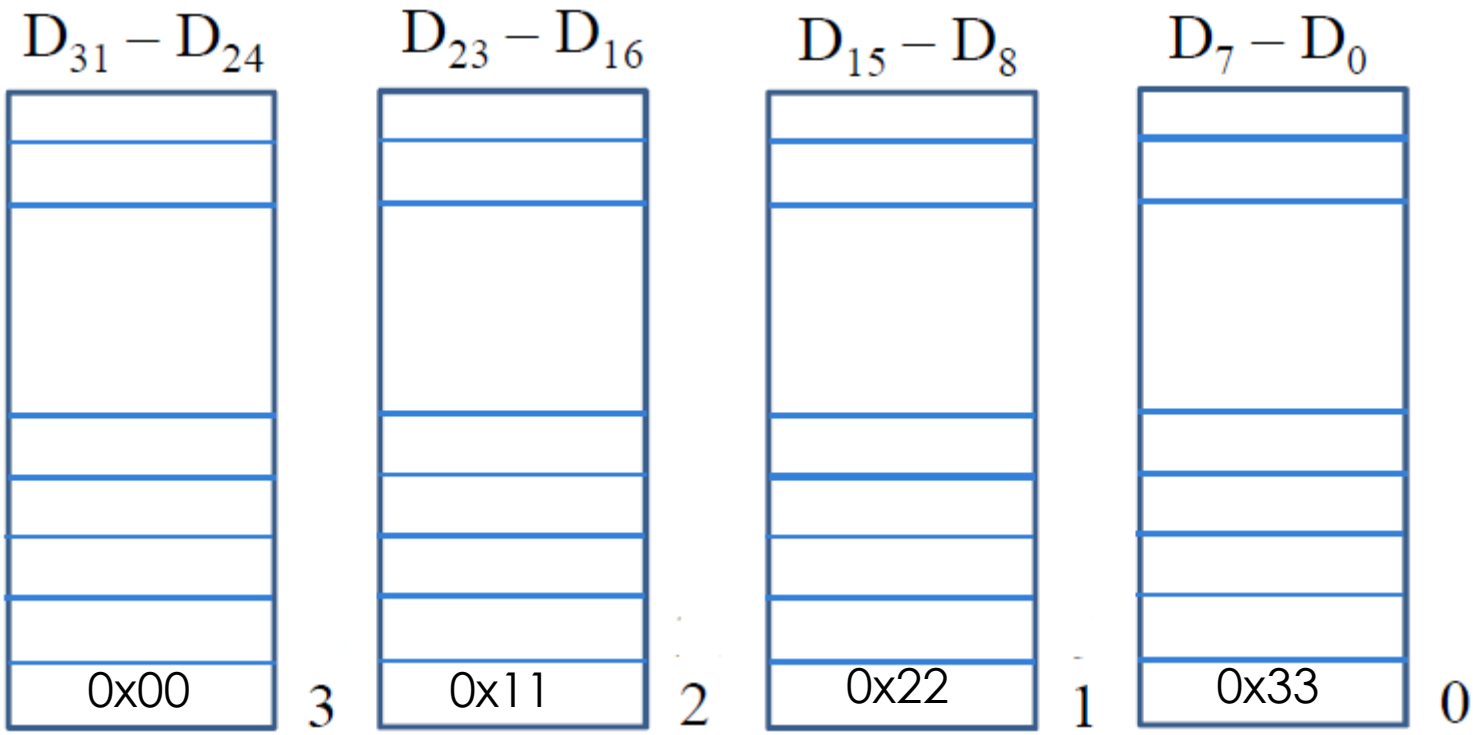
MEMORY OPERANDS (LITTLE VS BIG ENDIAN)

- Little Endian example
  - Data: 0x33221100



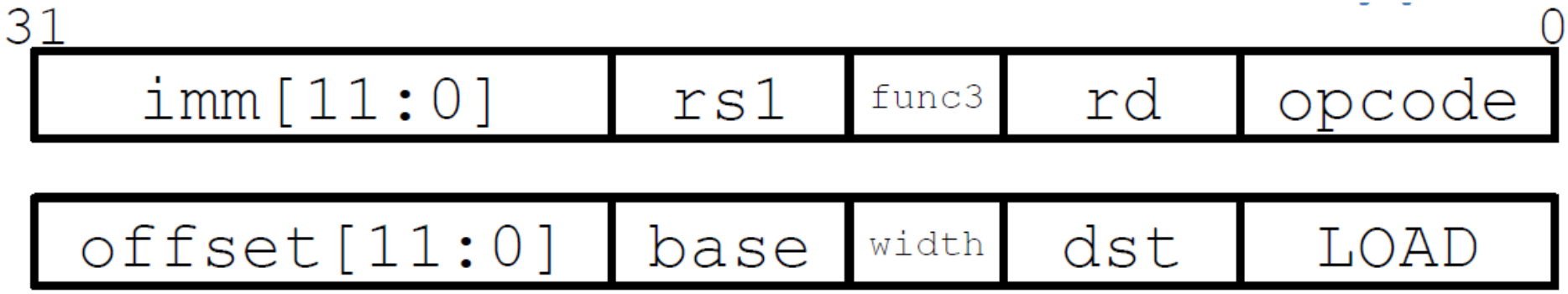
MEMORY OPERANDS (LITTLE VS BIG ENDIAN)

- Big Endian example
  - Data: 0x33221100



I-FORMAT (LOAD) INSTRUCTIONS

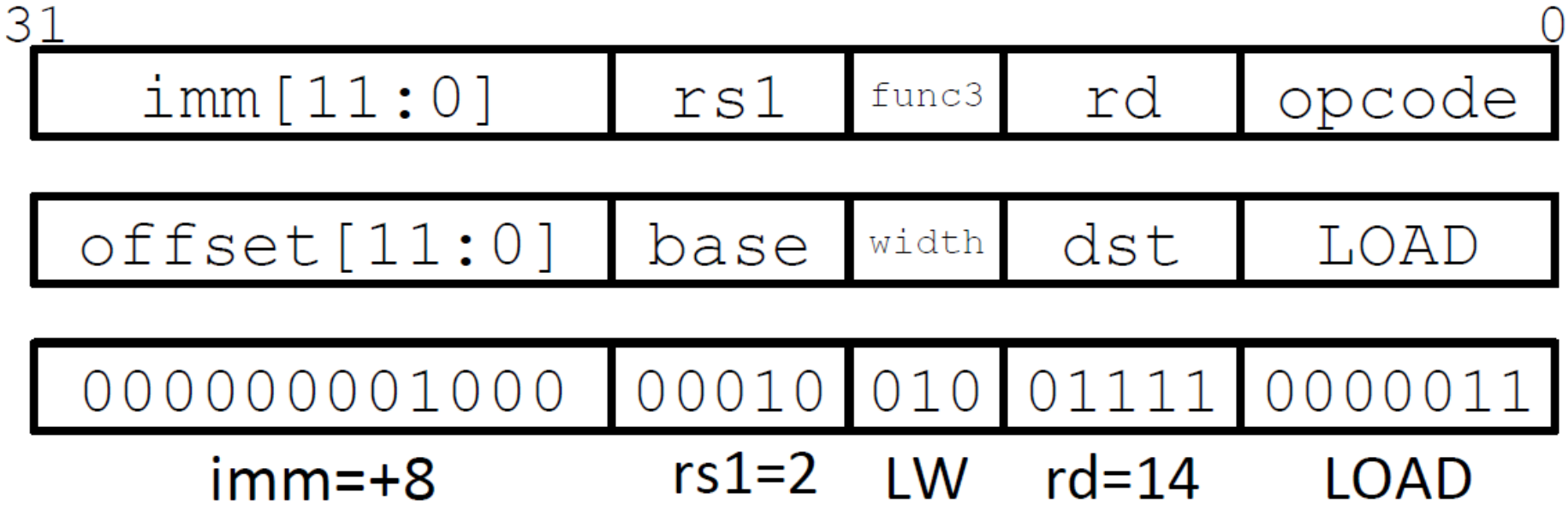
- Load instructions are also I-Format



- The 12-bit signed immediate is added to the base address in register rs1 to form the memory address
  - This is very similar to the add-immediate operation but used to create address, not to create the final result
- Value loaded from memory is stored in rd

I-FORMAT (LOAD) INSTRUCTIONS

- I-Format (Load) instruction Example
  - RISC V instruction: lw x14, 8(x2)



I-FORMAT (LOAD) INSTRUCTIONS

- All RV32 I-Format (Load) instruction

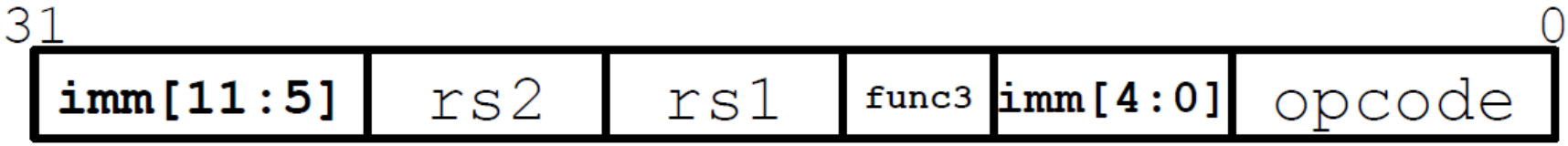
imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

funct3 field encodes size and signedness of load data

- LBU is "load unsigned byte"
- LH is "load halfword", which loads 16 bits (2 bytes) and sign-extends to fill the destination 32-bit register
- LHU is "load unsigned halfword", which zero-extends 16 bits to fill the destination 32-bit register
- There is no LWU in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32-bit register

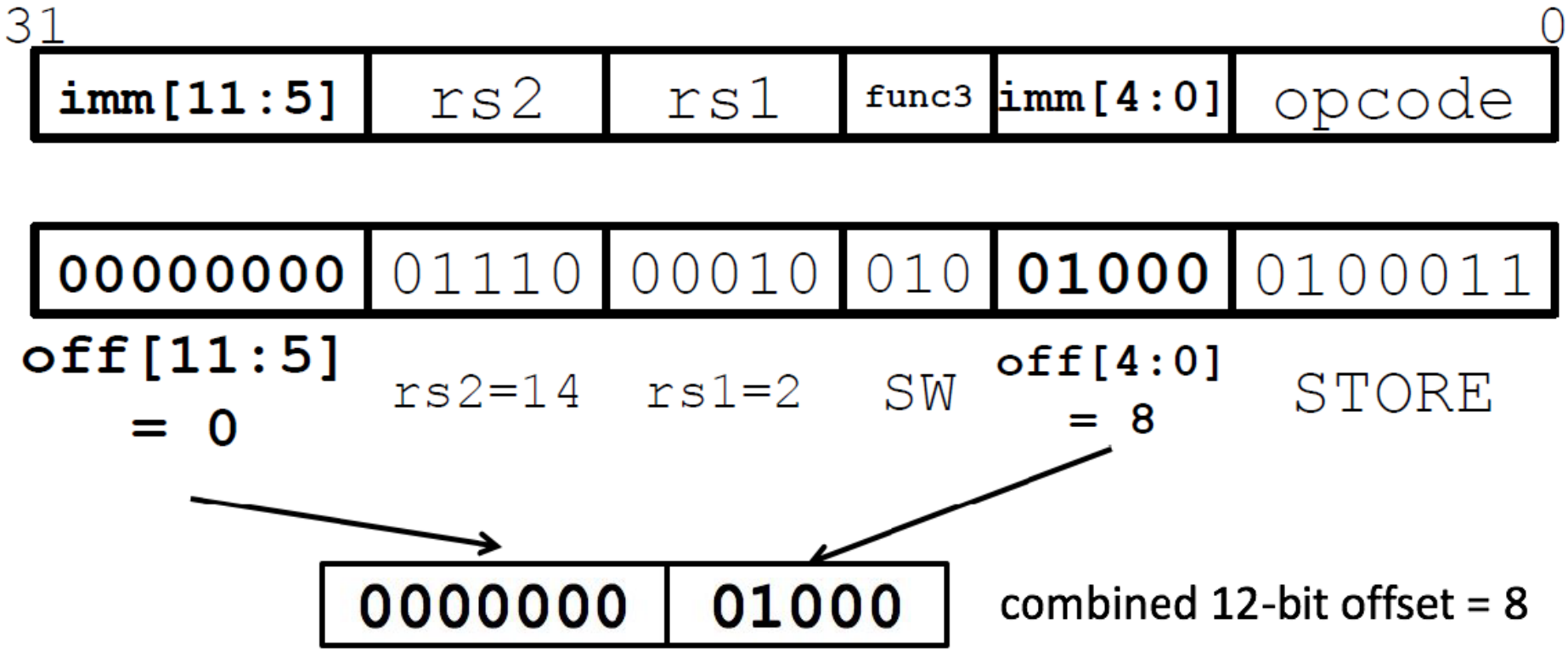
S-FORMAT INSTRUCTIONS

- Store needs to read two registers, rs1 for base memory address, and rs2 for data to be stored, as well as needs immediate offset
- Can't have both rs2 and immediate in the same place as other instructions
- Note: stores don't write a value to the register file, no rd
- RISC-V design decision is to move low 5 bits of immediate to where rd field was in other instructions – keep rs1/rs2 fields in same place
  - Register names more critical than immediate bits in hardware design



S-FORMAT INSTRUCTIONS

- S-Format instruction example
  - RISC V instruction: sw x14, 8(x2)



S-FORMAT INSTRUCTIONS

- All RV32 S-Format instruction

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

C TO RISC-V

- C:
  - a = b + c (a -> x1, b -> x2, c -> x3)
- RISC V:
  - add x1, x2, x3
- C:
  - d = e – f (d -> x3, e -> x4, f -> x5)
- RISC V:
  - sub x3, x4, x5

C TO RISC-V

- C:
  - a = b + c + d – e (x10 -> a, x1 -> b, x2 -> c, x3 -> d, x4 -> e)

- RISC V:
  - add x10, x1, x2 # a_temp = b + c
  - add x10, x10, x3 # a_temp = a_temp + d
  - sub x10, x10, x4 # a = a_temp - e

C TO RISC-V

- C:
  - f = g – 10 (x3 -> f, x4 -> g)
- RISC V:
  - addi x3, x4, -10
- C:
  - f = g (x3 -> f, x4 -> g)
- RISC V:
  - add x3, x4, x0