

CS 110

Computer Architecture

Lecture 7:

Multiplication and Floating Point

Instructors:

Sören Schwertfeger & Chundong Wang

<https://robotics.shanghaitech.edu.cn/courses/ca/20s/>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Summary of RISC-V Instruction Formats

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0			
funct7				rs2			rs1			funct3			rd			opcode	R-type
imm[11:0]						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]]						imm[19:12]						rd			opcode	J-type	

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

All RV32 I-format Arithmetic 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)

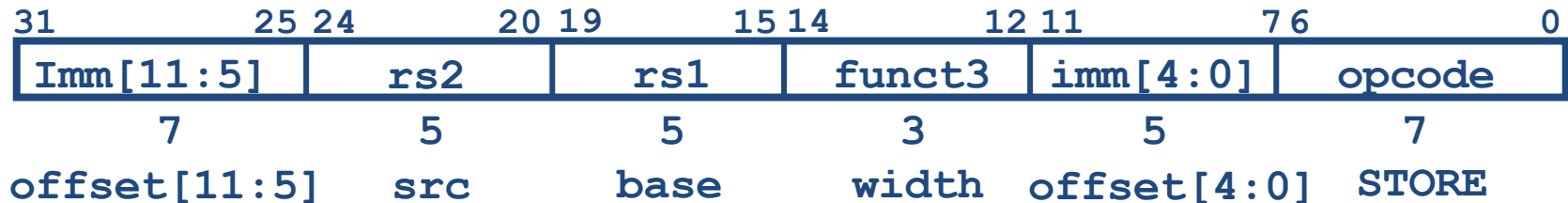
All RV32 Load Instructions

<code>imm[11:0]</code>	<code>rs1</code>	000	<code>rd</code>	0000011	<code>lb</code>
<code>imm[11:0]</code>	<code>rs1</code>	001	<code>rd</code>	0000011	<code>lh</code>
<code>imm[11:0]</code>	<code>rs1</code>	010	<code>rd</code>	0000011	<code>lw</code>
<code>imm[11:0]</code>	<code>rs1</code>	100	<code>rd</code>	0000011	<code>lbu</code>
<code>imm[11:0]</code>	<code>rs1</code>	101	<code>rd</code>	0000011	<code>lhu</code>

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 destination 32-bit register
- LHU is “load unsigned halfword”, which zero-extends 16 bits to fill 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 Used for Stores



- Store needs to read two registers, rs1 for base memory address, and rs2 for data to be stored, as well immediate offset!
- Can't have both rs2 and immediate in same place as other instructions!
- Note that stores don't write a value to the register file, ***no rd!***
- RISC-V design decision is 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

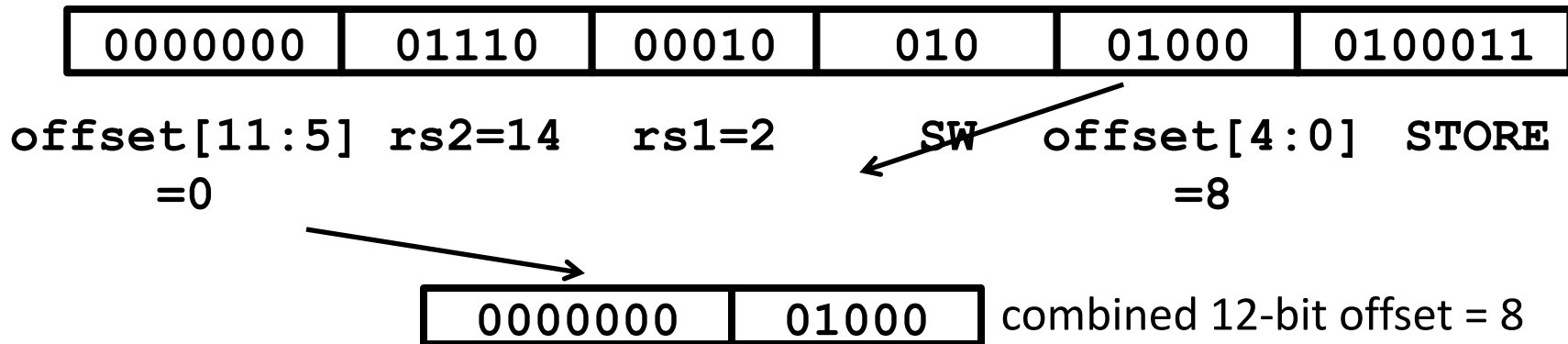
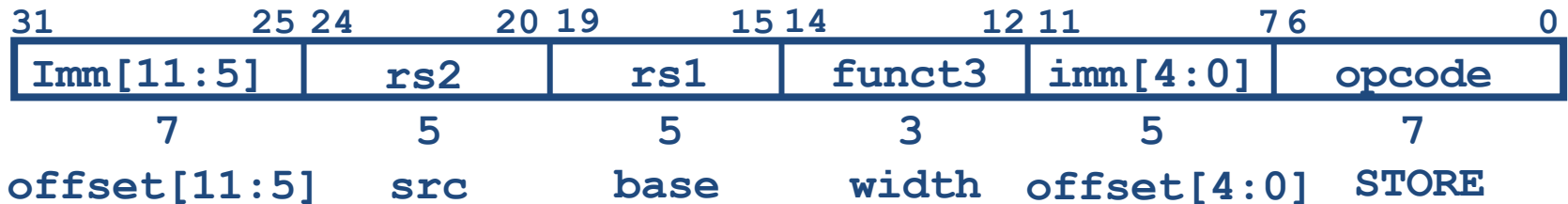
Keeping Registers always in the Same Place...

- The critical path for ***all operations*** includes fetching values from the registers
- By always placing the read sources in the same place, the register file can read without hesitation
 - If the data ends up being unnecessary (e.g. I-Type), it can be ignored
- Other RISCs have had slightly different encodings
 - Necessitating the logic to look at the instruction to determine which registers to read
- Example of one of the (many) little tweaks done in RISC-V to make things work better

S-Format Example

- RISC-V Assembly Instruction:

sw x14, 8(x2)



All RV32 Store Instructions

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

width

- Store byte, halfword, word

RISC-V Conditional Branches

- E.g., **BEQ x1, x2, Label**
- Branches read two registers but don't write a register (similar to stores)
- How to encode label, i.e., where to branch to?

Branching Instruction Usage

- Branches typically used for loops (`if-else`, `while`, `for`)
 - Loops are generally small (< 50 instructions)
 - Function calls and unconditional jumps handled with jump instructions (J-Format)
- **Recall:** Instructions stored in a localized area of memory (Code/Text)
 - Largest branch distance limited by size of code
 - Address of current instruction stored in the program counter (PC)

PC-Relative Addressing

- **PC-Relative Addressing:** Use the `immediate` field as a two's-complement offset to PC
 - Branches generally change the PC by a small amount
 - Can specify $\pm 2^{11}$ 'unit' addresses from the PC
- Why not use byte as a unit of offset from PC?
 - Because instructions are 32-bits (4-bytes)
 - We don't branch into middle of instruction

Scaling Branch Offset

- One idea: To improve the reach of a single branch instruction, multiply the offset by four bytes before adding to PC
- This would allow one branch instruction to reach $\pm 2^{11} \times 32$ -bit instructions either side of PC
 - Four times greater reach than using byte offset

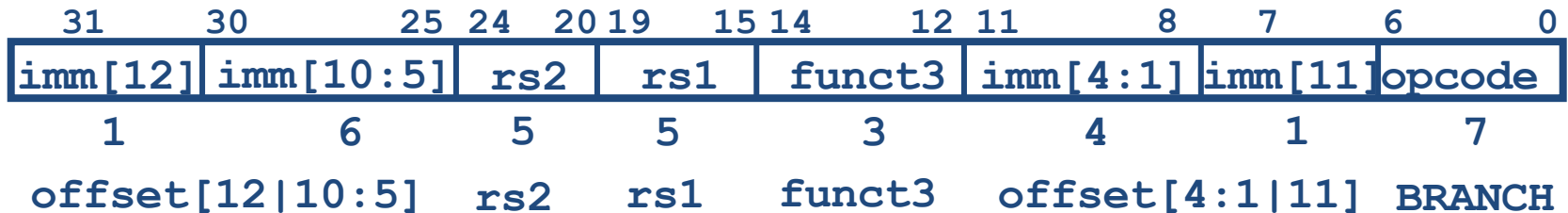
RISC-V Feature, $n \times 16$ -bit instructions

- Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 16-bits in length
- To enable this, RISC-V scales the branch offset by 2 bytes even when there are no 16-bit instructions
- Reduces branch reach by half and means that $\frac{1}{2}$ of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class)
- RISC-V conditional branches can only reach $\pm 2^{10} \times$ 32-bit instructions on either side of PC

Branch Calculation

- If we **don't** take the branch:
$$PC = PC + 4 \quad (\text{i.e., next instruction})$$
- If we **do** take the branch:
$$PC = PC + \text{immediate} * 2$$
- **Observations:**
 - `immediate` is number of (1/2-) instructions to jump (remember, specifies words) either forward (+) or backwards (–)

RISC-V B-Format for Branches




- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate imm[12:1]
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode *even* 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

Branch Example, Determine Offset

- RISC-V Code:

```
Loop: beq    x19, x10, End
      add    x18, x18, x10
      addi   x19, x19, -1
      j      Loop
End:  # target instruction
```



A wavy line with an arrow at the bottom points from the 'beq' instruction to a vertical list of numbers 0, 1, 2, 3, 4. The numbers 1, 2, 3, and 4 are aligned with the four instructions following the branch.


Count instructions from branch

- Branch offset = **4×32-bit instructions = 16 bytes**
- (Branch with offset of 0, branches to itself)

Branch Example, Determine Offset

- RISC-V Code:

```
Loop: beq    x19, x10, End
      add    x18, x18, x10
      addi   x19, x19, -1
      j      Loop
End:  # target instruction
```



Count
instructions
from branch

???????	01010	10011	000	?????	1100011
imm	rs2=10	rs1=19	BEQ	imm	BRANCH

Branch Example, Encode Offset

- RISC-V Code:

```
Loop: beq    x19, x10, End
```

```
    add     x18, x18, x10
```

```
    addi    x19, x19, -1
```

```
    j       Loop
```

```
End: # target instruction
```

offset = 16 bytes = 8x2 bytes

???????	01010	10011	000	?????	1100011
imm	rs2=10	rs1=19	BEQ	imm	BRANCH

RISC-V Immediate Encoding

Instruction encodings, inst[31:0]

31	30	25	24	20	19	15	14	12	11	8	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

32-bit immediates produced, imm[31:0]

31	25	24	12	11	10	5	4	1	0			
-inst[31]-					inst[30:25]		inst[24:21]		inst[20]		I-imm.	
-inst[31]-					inst[30:25]		inst[11:8]		inst[7]		S-imm.	
-inst[31]-					inst[7]		inst[30:25]		inst[11:8]		0	B-imm.

Upper bits sign-extended from inst[31] always

Only bit 7 of instruction changes role in immediate between S and B

Branch Example, complete encoding

beq **x19,x10**, offset = 16 bytes

13-bit immediate, imm[12:0], with value 16

0000000010000

imm[0] discarded,
always zero

imm[12]

imm[11]

0	000000	01010	10011	000	1000	0	1100011
---	--------	-------	-------	-----	------	---	---------

imm[10:5] rs2=10 rs1=19 BEQ imm[4:1] BRANCH

All RISC-V Branch Instructions

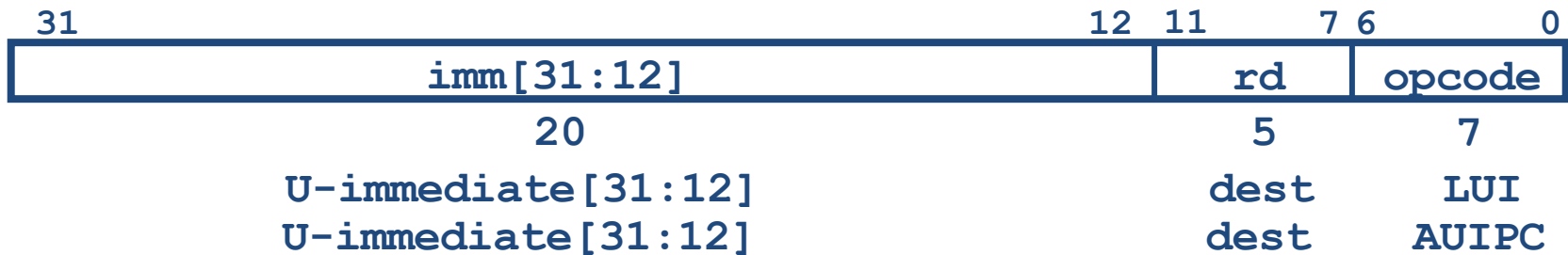
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU

Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
 - If moving individual lines of code, then yes
 - If moving all of code, then no ('position-independent code')
- What do we do if destination is $> 2^{10}$ instructions away from branch?
 - Other instructions save us

```
beq x10,x0,far          bne x10,x0,next
# next instr           j    far
                        next: # next instr
```

U-Format for “Upper Immediate” Instructions



- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
 - LUI – Load Upper Immediate
 - AUIPC – Add Upper Immediate to PC

LUI to Create Long Immediates

- LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
- Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (LUI/ADDI).

```
LUI x10, 0x87654      # x10 = 0x87654000
```

```
ADDI x10, x10, 0x321  # x10 = 0x87654321
```

One Corner Case

How to set 0xDEADBEEF?

LUI x10, 0xDEADB # x10 = 0xDEADB000

ADDI x10, x10, 0xEEF # x10 = 0xDEAD**A**EEF

ADDI 12-bit immediate is always sign-extended, if top bit is set, will subtract 1 from upper 20 bits

Solution

How to set 0xDEADBEEF?

```
LUI x10, 0xDEADC          # x10 = 0xDEADC000
```

```
ADDI x10, x10, 0xEEF      # x10 = 0xDEADBEEF
```

Pre-increment value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.

Assembler pseudo-op handles all of this:

```
li x10, 0xDEADBEEF        # Creates two instructions
```

Actually: Important!

The assembler treats the provided number for ADDI as signed number. So in order to get 0xEEF, we have to provide the according negative number! So actually, only this works:

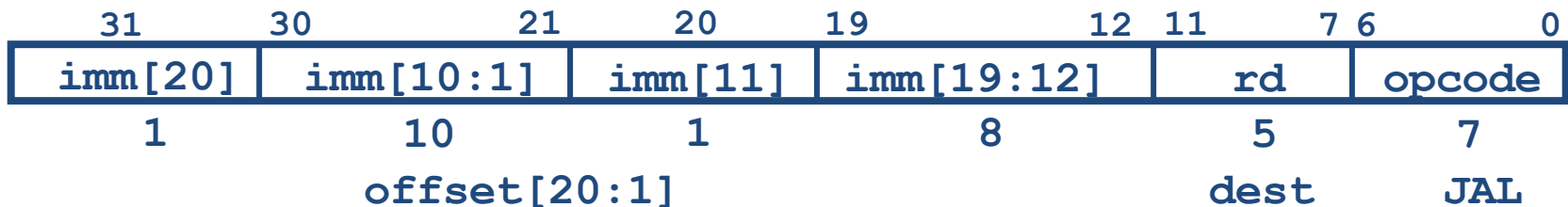
```
ADDI x10, x10, -273       # -273 = 0xFFFFFFFFEEF
```

AUIPC

- Adds upper immediate value to PC and places result in destination register
- Used for PC-relative addressing

```
Label: AUIPC x10, 0 # Puts address of label in x10
```

J-Format for Jump Instructions



- JAL saves PC+4 in register rd (the return address)
 - Assembler “j” jump is pseudo-instruction, uses JAL but sets rd=x0 to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
 - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

Uses of JAL

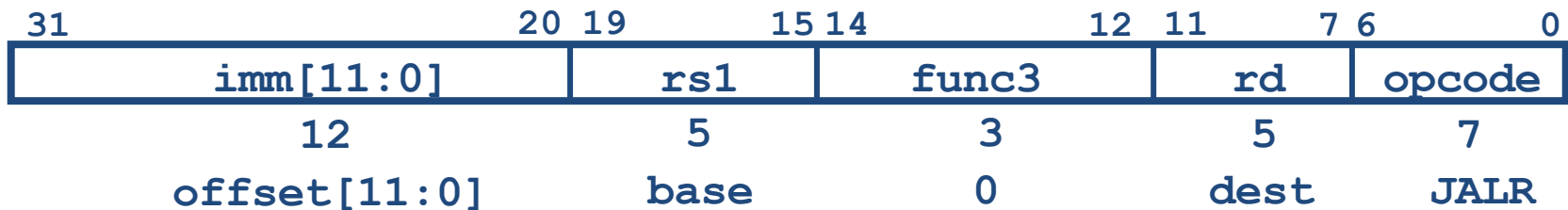
j pseudo-instruction

j Label = jal x0, Label # Discard return address

Call function within 2^{18} instructions of PC

jal ra, FuncName

JALR Instruction (I-Format)



- JALR rd, rs, immediate
 - Writes PC+4 to rd (return address)
 - Sets PC = rs + immediate
 - Uses same immediates as arithmetic and loads
 - **no** multiplication by 2 bytes
 - In contrast to branches and JAL

Uses of JALR

ret and jr psuedo-instructions

```
ret = jr ra = jalr x0, ra, 0
```

Call function at any 32-bit absolute address

```
lui x1, <hi20bits>
```

```
jalr ra, x1, <lo12bits>
```

Jump PC-relative with 32-bit offset

```
auipc x1, <hi20bits>
```

```
jalr x0, x1, <lo12bits>
```

Summary of RISC-V Instruction Formats

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
funct7					rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						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]]						imm[19:12]				rd		opcode		J-type	

CORE 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	
SB	imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]		opcode	
U	imm[31:12]										rd		opcode	
UJ	imm[20 10:1 11 19:12]										rd		opcode	

Complete RV32I ISA

imm[31:12]				rd	0110111
imm[31:12]				rd	0010111
imm[20 10:1 11 19:12]				rd	1101111
imm[11:0]		rs1	000	rd	1100111
imm[12 10:5]		rs2	000	imm[4:1 11]	1100011
imm[12 10:5]		rs2	001	imm[4:1 11]	1100011
imm[12 10:5]		rs2	100	imm[4:1 11]	1100011
imm[12 10:5]		rs2	101	imm[4:1 11]	1100011
imm[12 10:5]		rs2	110	imm[4:1 11]	1100011
imm[12 10:5]		rs2	111	imm[4:1 11]	1100011
imm[11:0]		rs1	000	rd	0000011
imm[11:0]		rs1	001	rd	0000011
imm[11:0]		rs1	010	rd	0000011
imm[11:0]		rs1	100	rd	0000011
imm[11:0]		rs1	101	rd	0000011
imm[11:5]		rs2	000	imm[4:0]	0100011
imm[11:5]		rs2	001	imm[4:0]	0100011
imm[11:5]		rs2	010	imm[4:0]	0100011
imm[11:0]		rs1	000	rd	0010011
imm[11:0]		rs1	010	rd	0010011
imm[11:0]		rs1	011	rd	0010011
imm[11:0]		rs1	100	rd	0010011
imm[11:0]		rs1	110	rd	0010011
imm[11:0]		rs1	111	rd	0010011

LUI
 AUIPC
 JAL
 JALR
 BEQ
 BNE
 BLT
 BGE
 BLTU
 BGEU
 LB
 LH
 LW
 LBU
 LHU
 SB
 SH
 SW
 ADDI
 SLTI
 SLTIU
 XORI
 ORI
 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
0000000000000			00000	000	00000	1110011	ECALL
0000000000001			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	

Not in CA lectures

ADMIN

HW

- HW2 is due this Friday
- HW3 will be published this week
 - RISC-V programming
 - In Venus simulator
 - Venus tutorial on the website

RISC-V ISA Specification

- Different modules
- Class covers RV32I Base Integer Instruction Set
 - RV64I (used in textbook) and RV128I also available
 - RV32E: Embedded Systems (only 16 registers)
- Various Extensions, named with leeters
- The RISC-V Instruction Set Manual; Volume II: Privileged Architecture
 - For Operating System

RISC-V Specifications

- <https://riscv.org/technical/specifications/>
 - ISA Specification
 - Debug Specification
 - Trace Specification
 - Compliance Framework
- <https://five-embeddev.com/riscv-isa-manual/latest/intro.html>
 - Manual
- <https://github.com/riscv/riscv-isa-manual/releases/latest>
 - Latest draft document

Editors: Andrew Waterman¹, Krste Asanović^{1,2}

¹SiFive Inc.,

²CS Division, EECS Department, University of California, Berkeley

andrew@sifive.com, krste@berkeley.edu

March 5, 2021

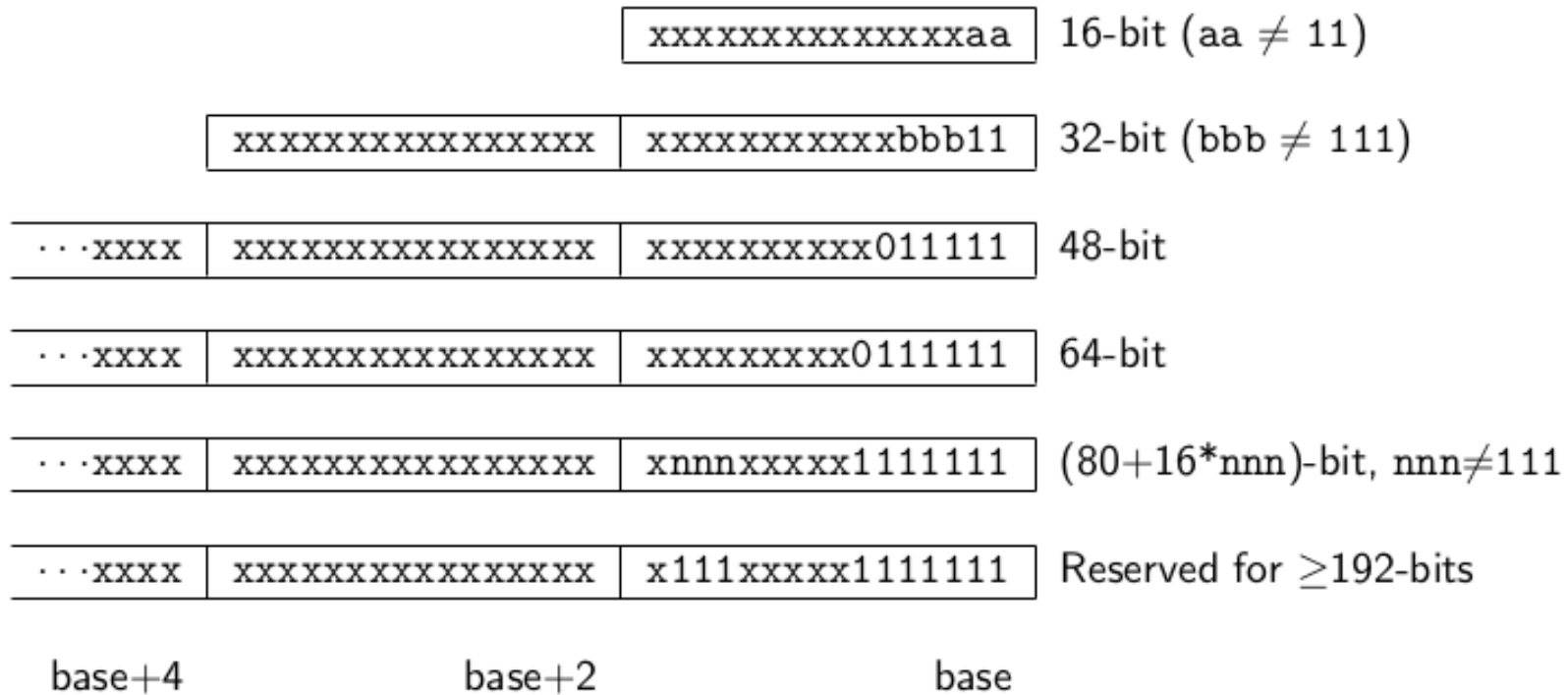
Subset	Name	Implies
Base ISA		
Integer	I	
Reduced Integer	E	
Standard Unprivileged Extensions		
Integer Multiplication and Division	M	Zicsr F
Atomics	A	
Single-Precision Floating-Point	F	
Double-Precision Floating-Point	D	
General	G	IMADZifencei
Quad-Precision Floating-Point	Q	D
Decimal Floating-Point	L	
16-bit Compressed Instructions	C	
Bit Manipulation	B	
Dynamic Languages	J	
Transactional Memory	T	A
Packed-SIMD Extensions	P	
Vector Extensions	V	
User-Level Interrupts	N	
Control and Status Register Access	Zicsr	
Instruction-Fetch Fence	Zifencei	
Misaligned Atomics	Zam	
Total Store Ordering	Ztso	
Standard Supervisor-Level Extensions		
Supervisor-level extension “def”	Sdef	
Standard Hypervisor-Level Extensions		
Hypervisor-level extension “ghi”	Hghi	
Standard Machine-Level Extensions		
Machine-level extension “jkl”	Zxmjkl	
Non-Standard Extensions		
Non-standard extension “mno”	Xmno	

Base	Version	Status
RVWMO	2.0	Ratified
RV32I	2.1	Ratified
RV64I	2.1	Ratified
<i>RV32E</i>	<i>1.9</i>	<i>Draft</i>
<i>RV128I</i>	<i>1.7</i>	<i>Draft</i>
Extension	Version	Status
M	2.0	Ratified
A	2.1	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
<i>Counters</i>	<i>2.0</i>	<i>Draft</i>
<i>L</i>	<i>0.0</i>	<i>Draft</i>
<i>B</i>	<i>0.0</i>	<i>Draft</i>
<i>J</i>	<i>0.0</i>	<i>Draft</i>
<i>T</i>	<i>0.0</i>	<i>Draft</i>
<i>P</i>	<i>0.2</i>	<i>Draft</i>
<i>V</i>	<i>0.7</i>	<i>Draft</i>
Zicsr	2.0	Ratified
Zifencei	2.0	Ratified
<i>Zam</i>	<i>0.1</i>	<i>Draft</i>
<i>Ztso</i>	<i>0.1</i>	<i>Frozen</i>

Clarifications

- RISC-V ISA Spec: Does NOT define Assembly Syntax
 - Defines Binary Machine Instructions and their behavior
 - Different Assemblers could have different syntax (i.e. allow commas or not)
- Project 1 RISC-V emulator: behave exactly like Venus!
- ALL I-Type instructions (including sltiu):
 - do sign-extension
 - (in Venus): input number is signed, even if hex

RISC-V instruction sizes



RV32M

- Multiplication and Division for RV32I

MUL
MULH
MULHSU
MULHU
DIV
DIVU
REM
REMU

Source Registers	Destination Registers
<i>rs1, rs2</i>	<i>rd</i>
<i>rs1, rs2</i>	<i>rd</i>
<i>rs1, rs2</i>	<i>rd</i>
<i>rs1, rs2</i>	<i>rd</i>
<i>rs1, rs2</i>	<i>rd</i>
<i>rs1, rs2</i>	<i>rd</i>
<i>rs1, rs2</i>	<i>rd</i>

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

Multiplicand	1000	8
Multiplier	<u>x1001</u>	9
	1000	
	0000	
	0000	
	+1000	
	<u>01001000</u>	72

- m bits \times n bits = $m + n$ bit product

Integer Multiplication (2/3)

- In RISC-V, we multiply registers, so:
 - 32-bit value x 32-bit value = 64-bit value
- Multiplication is **not** part of standard RISC-V because:
 - It requires a more complicated ALU
 - The compiler can use a series of shifts and adds if the multiplier isn't present
- Syntax of Multiplication (signed):
 - **mul rd, rs1, rs2**
 - **mulh rd, rs1, rs2**
 - Multiplies 32-bit values in those registers and returns either the lower or upper 32b result
 - If you do mulh/mul back to back, the architecture can fuse them
 - Also unsigned versions of the above

Integer Multiplication (3/3)

- Example:
 - in C: **a = b * c;**
 - **int64_t a; int32_t b, c;**
 - These types are defined in C99, in `stdint.h`
- in RISC-V:
 - let **b** be **s2**; let **c** be **s3**; and let **a** be **s0** and **s1**
(since it may be up to 64 bits)
 - **mulh s1, s2, s3**
mul s0, s2, s3

Integer Division (1/2)

- Paper and pencil example (unsigned):

– Quotient = $1001010 / 1000$

– Remainder = $1001010 \% 1000$

		<u>1001</u>	Quotient
Divisor	1000	1001010	Dividend
		<u>-1000</u>	
		10	
		101	
		1010	
		<u>-1000</u>	
		10	Remainder
			(or Modulo result)

Dividend = Quotient x Divisor + Remainder

Integer Division (2/2)

- Syntax of Division (signed):
 - **div rd, rs1, rs2**
rem rd, rs1, rs2
 - Divides 32-bit rs1 by 32-bit rs2, returns the quotient (/) for div, remainder (%) for rem
 - Again, can fuse two adjacent instructions
- Example in C: `a = c / d; b = c % d;`
- RISC-V:
 - `a↔s0; b↔s1; c↔s2; d↔s3`
 - **div s0, s2, s3**
rem s1, s2, s3

Note Optimization...

- A recommended convention
 - **mulh** s1 s2 s3
 mul s0 s2 s3
 - **div** s0 s2 s3
 rem s1 s2 s3
- Not a *requirement but*...
 - RISC-V says "if you do it this way, *and* the microarchitecture supports it, it can fuse the two operations into one"
 - Same logic behind much of the 16b ISA design: If you follow the convention you can get significant optimizations

Review of Integer Numbers

- Computers are made to deal with numbers
- What can we represent in N bits?
 - 2^N things, and no more! They could be...
 - Unsigned integers:
 0 to $2^N - 1$
(for $N=32$, $2^N - 1 = 4,294,967,295$)
 - Signed Integers (Two's Complement)
 $-2^{(N-1)}$ to $2^{(N-1)} - 1$
(for $N=32$, $2^{(N-1)} = 2,147,483,648$)

What about other numbers?

1. Very large numbers? (seconds/millennium)
 $\Rightarrow 31,556,926,000_{10} (3.1556926_{10} \times 10^{10})$
2. Very small numbers? (Bohr radius)
 $\Rightarrow 0.0000000000529177_{10}\text{m} (5.29177_{10} \times 10^{-11})$
3. Numbers with both integer & fractional parts?
 $\Rightarrow 1.5$

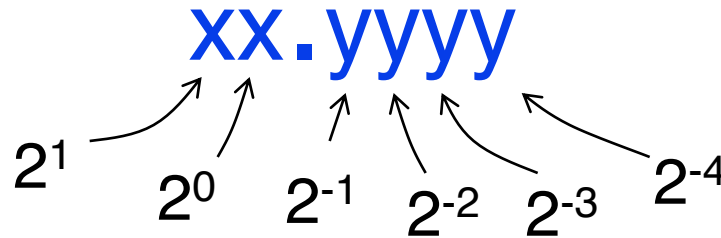
First consider #3.

...our solution will also help with #1 and #2.

Representation of Fractions

“Binary Point” like decimal point signifies boundary between integer and fractional parts:

Example 6-bit representation:



$$10.1010_{\text{two}} = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{\text{ten}}$$

If we assume “fixed binary point”, range of 6-bit representations with this format:

0 to 3.9375 (almost 4)

Fractional Powers of 2

i	2^{-i}	
0	1.0	1
1	0.5	1/2
2	0.25	1/4
3	0.125	1/8
4	0.0625	1/16
5	0.03125	1/32
6	0.015625	
7	0.0078125	
8	0.00390625	
9	0.001953125	
10	0.0009765625	
11	0.00048828125	
12	0.000244140625	
13	0.0001220703125	
14	0.00006103515625	
15	0.000030517578125	

Representation of Fractions with Fixed Pt.

What about addition and multiplication?

Addition is
straightforward:

$$\begin{array}{r} 01.100 \\ + 00.100 \\ \hline 10.000 \end{array} \quad \begin{array}{r} 1.5_{\text{ten}} \\ 0.5_{\text{ten}} \\ \hline 2.0_{\text{ten}} \end{array} \quad \begin{array}{r} 01.100 \\ 00.100 \\ \hline 00.000 \end{array} \quad \begin{array}{r} 1.5_{\text{ten}} \\ 0.5_{\text{ten}} \\ \hline 0.0_{\text{ten}} \end{array}$$

Multiplication a bit more complex:

$$\begin{array}{r} 00 \ 000 \\ 000 \ 00 \\ 0110 \ 0 \\ 00000 \\ 00000 \\ \hline 0000110000 \end{array}$$

Where's the answer, 0.11? (need to remember where point is)

Representation of Fractions

So far, in our examples we used a “fixed” binary point. What we really want is to “float” the binary point. Why?

Floating binary point most effective use of our limited bits (and thus more accuracy in our number representation):

example: put 0.1640625_{ten} into binary. Represent with 5-bits choosing where to put the binary point.

... 000000.001010100000...



Store these bits and keep track of the binary point 2 places to the left of the MSB

Any other solution would lose accuracy!

With floating-point rep., each numeral carries an exponent field recording the whereabouts of its binary point.

The binary point **can be outside** the stored bits, so very large and small numbers can be represented.

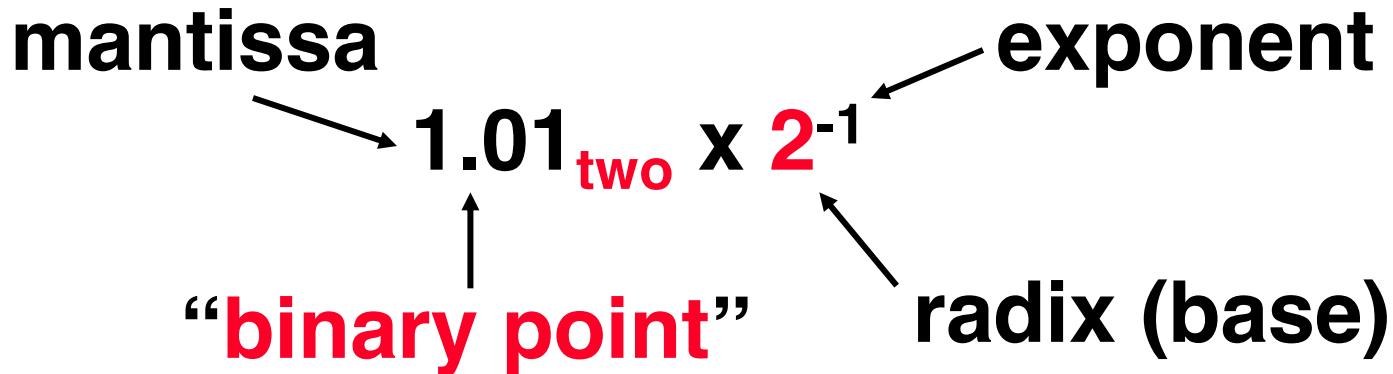
Scientific Notation (in Decimal)

The diagram illustrates the components of scientific notation using the example $6.02_{\text{ten}} \times 10^{23}$. Arrows point from labels to specific parts of the expression:

- mantissa**: Points to the number 6.02.
- decimal point**: Points to the dot between 6 and 02.
- exponent**: Points to the superscript 23.
- radix (base)**: Points to the base 10.

- Normalized form: no leading 0s
(exactly one digit to left of decimal point)
- Alternatives to representing $1/1,000,000,000$
 - Normalized: 1.0×10^{-9}
 - Not normalized: $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$

Scientific Notation (in Binary)



- Computer arithmetic that supports it called floating point, because it represents numbers where the binary point is not fixed, as it is for integers
 - Declare such variable in C as `float`
 - `double` for double precision.

Floating-Point Representation (1/2)

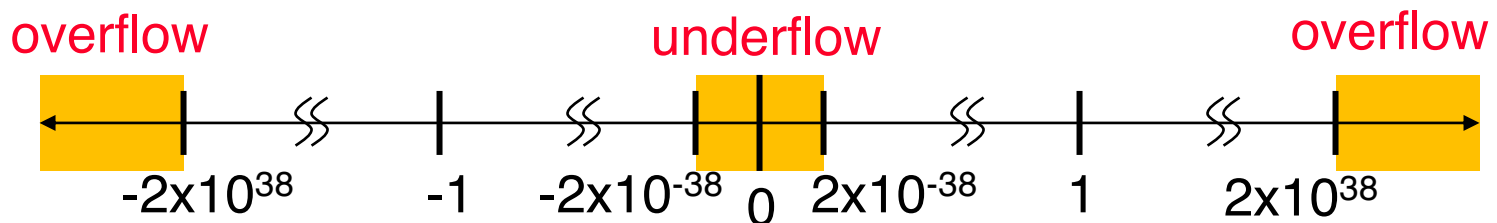
- Normal format: $+1.\text{xxx...x}_{\text{two}} * 2^{\text{yyy...y}_{\text{two}}}$
- Multiple of Word Size (32 bits)



- S represents Sign
Exponent represents y's
Significand represents x's
- Represent numbers as small as $2.0_{\text{ten}} \times 2^{-126}$ to as large as $2.0_{\text{ten}} \times 2^{127}$
- $2^{126} = 8.507059173023462 \text{ e}37 \approx 10^{38}$

Floating-Point Representation (2/2)

- What if result too large?
($> 2.0 \times 10^{38}$, $< -2.0 \times 10^{38}$)
 - **Overflow!** \Rightarrow Exponent larger than represented in 8-bit Exponent field
- What if result too small?
(>0 & $< 2.0 \times 10^{-38}$, <0 & $> -2.0 \times 10^{-38}$)
 - **Underflow!** \Rightarrow Negative **exponent** larger than represented in 8-bit Exponent field



- What would help reduce chances of overflow and/or underflow?

IEEE 754 Floating Point Standard (1/3)

Single Precision (Double Precision similar):



- **Sign** bit: 1 means negative 0 means positive
- **Significand** in *sign-magnitude* format (not 2's complement)
 - To pack more bits, leading 1 implicit for normalized numbers
 - 1 + 23 bits single, 1 + 52 bits double
 - always true: $0 < \text{Significand} < 1$ (for normalized numbers)
- Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

IEEE 754 Floating Point Standard (2/3)

- IEEE 754 uses “biased exponent” representation
 - Designers wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
 - Wanted bigger (integer) exponent field to represent bigger numbers
 - 2’s complement poses a problem (because negative numbers look bigger)
 - Use just magnitude and offset by half the range

IEEE 754 Floating Point Standard (3/3)

- Called Biased Notation, where bias is number subtracted to get final number
 - IEEE 754 uses bias of 127 for single prec.
 - Subtract 127 from Exponent field to get actual value for exponent

- **Summary (single precision):**



- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

- Double precision identical, except with exponent bias of 1023 (half, quad similar)