Cornell Bowers CIS
Computer Science

CS3410: Computer Systems and Organization

LEC10: RISC-V Data Transfer

Professor Giulia Guidi
Monday, September 29, 2025

Credits: Bala, Bracy, Garcia, Guidi, Kao, Sampson, Sirer, Weatherspoon

Quick Introduction

Meet the Course Staff

Instructors

Giulia Guidi (she/her)
Professor
Hometown: Mantua, Italy
Ask me about: research, dogs, pottery, hiking
OH: Book here

Mahler

Nina

Plan for Today

- Review
- RISC-V Data Transfer
- **Possibly** RISC-V Control Flow or Decision Making (likely on Wednesday)

PSA: Prelim & Prelim Survey

Review of memory basics and Endianess

Zero-Extension

Unsigned number **extension** from 4 to 8 bit:

E.g., 1 ➡ 0001 ➡ 0000 0001

append additiona bits and set them to **zero**

Sign-Extension

Two's complement: To negate any number, flip the bits, *and add one*

E.g., -1: 1 ➡ 0001 ➡ 1110 ➡ 1111

basically, smear the MSB like butter on bread

Ok, what if I want to represent -1 on **8 bits** instead of 4?

-1: **1**111 ➡ 1111 **1**111

look at the most significant bit (MSB), if 0 ➡ then append 0s
if 1 ➡ then append 1s

7: **0**111 ➡ 0000 **0**111

Ok, what about hex? E.g., -1 = FFFF

If MSB = 0-7 ➡ then append 0s

Otherwise append Fs

Bit Truncation

Ok, so what if we need to move **from 8 to 4 bits**?

E.g., 0000 0111 = 7

E.g., ~~0000~~ 0111 = 7 😀

E.g., 0000 1111 = 15

E.g., ~~0000~~ 1111 = -1 😥
if *signed*

Can’t fix that—Can’t represent 15 on 4 bits *if signed*

Memory is Byte-Addressable

The memory is like an array of bytes—think of memory as a **long sequence of numbered boxes**

- Each **address** is like the **index** of a box
- Each **box stores** 8 bits = **1 byte** of data

address	value (hex)	value (8-bit binary)
1000	0xAA	10101010
1001	0xBB	10111011
1002	0xCC	11001100
1003	0xDD	11011101

Memory is Byte-Addressable

- 4 **bytes** = 1 **word**, thus the **word** addresses are **4 bytes apart**
- So, larger data (e.g., 32-bit word) is stored across **multiple consecutive addresses**
    - A word must start at an **address divisible by 4** (alignment)

```
address  value (hex)  value (8-bit binary)
-------  -----------  --------------------
 1000    0xAA         10101010
 1001    0xBB         10111011
 1002    0xCC         11001100
 1003    0xDD         11011101    <- 1 word (bytes 1000-1003)
 1004    0x11         00010001
 1005    0x22         00100010
 1006    0x33         00110011
 1007    0x44         01000100    <- 1 word (bytes 1004-1007)
```

Big Endian and Little Endian

Endianness = how those 4 **bytes** are ordered in memory

- **Bits** are always stored as usual within a **byte**

`int x = 0xDDCCBBAA (1 word)`

Little endian (RISC-V default): lowest-addressed byte = least significant byte (LSB)

```
address  value (hex)  value (8-bit binary)
-------  -----------  --------------------
 1000    0xAA   (LSB) 10101010
 1001    0xBB         10111011
 1002    0xCC         11001100
 1003    0xDD   (MSB) 11011101 <- 1 word (bytes 1000-1003)
```

Big Endian and Little Endian

Endianness = how those 4 **bytes** are ordered in memory

- **Bits** are always stored as usual within a **byte**

```
int x = 0xDDCCBBAA (1 word)
```

Big endian: lowest-addressed byte = most significant byte (MSB)

```
address  value (hex)  value (8-bit binary)
-------  -----------  --------------------
 1000    0xDD   (MSB) 10101010
 1001    0xCC         10111011
 1002    0xBB         11001100
 1003    0xAA   (LSB) 11011101 <- 1 word (bytes 1000-1003)
```

Poll

Q: If we store the 32-bit value `0xF0CACC1A` at address 1000 — what is stored at address 1000 in little endian?

PollEv.com /gguidi

Or send **gguidi** to **22333**

Poll

Q: If we store the 32-bit value `0xF0CACC1A` at address 1000 — what is stored at address 1000 in little endian?

Little endian (RISC-V default): lowest-addressed byte = least significant byte (LSB)

address	value (hex)	
1000	0x1A	(LSB)
1001	0xCC	
1002	0xCA	
1003	0xF0	(MSB)

Ok, moving on to RISC-V assembly

RISC-V Overview

Central Processing Unit

Basic job of a CPU: execute **instructions**!

the primitive operations that the CPU may execute

Instruction Set Architecture

An **ISA** defines what operations a particular CPU supports, and how it implements them

- The **assembly** language: the low-level CPU instructions
- The **machine** language: how the instructions are represented, in bits

RISC-V ISA defines instructions for the CPU down to the bit level:

```
add  x18, x19, x10
```

assembly code

```
0000000 01010 10011 000 10010 0110011
```

machine code

RISC-V Overview

High Level Language Program (e.g. C)

Compiler

Assembly Language Program (e.g., RISC-V)

Assembler

Machine Language Program (e.g., RISC-V)

Hardware

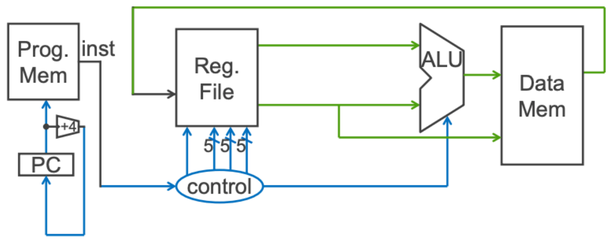
Hardware Architecture Description (e.g., block diagram)

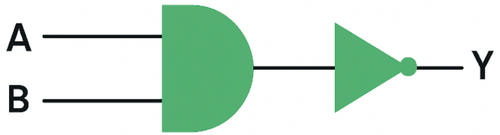
Logic Circuit Description (e.g., circuit schematic diagram)

```
temp = v[k];
v[k] = v[k+1];
```

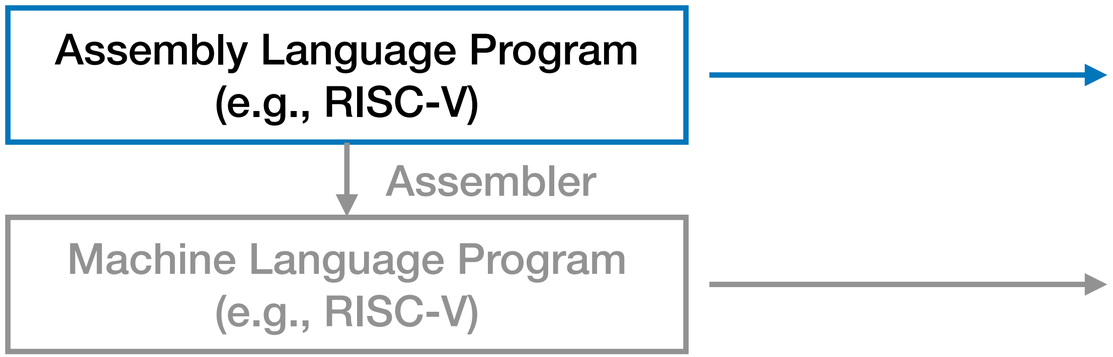
```
lw x13, 0(x12) # temp = v[k]
lw x14, 4(x12) # x14 = v[k+1]
sw x14, 0(x12) # v[k] = v[k+1]
```

```
0000 0000 0000 0110 0010 0110 1000 0011
0000 0000 0100 0110 0010 0111 0000 0011
0000 0000 1110 0110 0010 0000 0010 0011
```





RISC-V Overview




```
lw x13, 0(x12) # temp = v[k]
lw x14, 4(x12) # x14 = v[k+1]
sw x14, 0(x12) # v[k] = v[k+1]
```

```
0000 0000 0000 0110 0010 0110 1000 0011
0000 0000 0100 0110 0010 0111 0000 0011
0000 0000 1110 0110 0010 0000 0010 0011
```

RISC-V Assembly

RISC-V addition and subtraction

destination register

```
add rd, rs1, rs2   source registers
   R[rd] = R[rs1] + R[rs2]
```

```
sub rd, rs1, rs2
   R[rd] = R[rs1] - R[rs2]
```

RISC-V add immediate

```
addi rd, rs1, imm
   R[rd] = R[rs1] + imm
```

RISC-V shift right logical immediate

```
srli rd, rd, 1
   R[rd] = R[rd] >> 1 (eq. to /2)
```

RISC-V Assembly

Disassembled assembly

function address

function name

```
0000000000000000 <mean>:
   0:    00b50533          add     a0, a0, a1
   4:    00155513          srli    a0, a0, 0x1
   8:    00008067          ret
```

offset (word)

instruction

destination register

source registers

The instructions are sitting on “function address” + “offset”

In RISC-V, every instruction is **exactly 4 bytes long**, so the next instruction starts at address 4

RISC-V Assembly

Disassembled assembly

function address function name

```
0000000000000000 <mean>:
   0:   00b50533
   4:   00155513
   8:   00008067
```

instruction

offset (word)

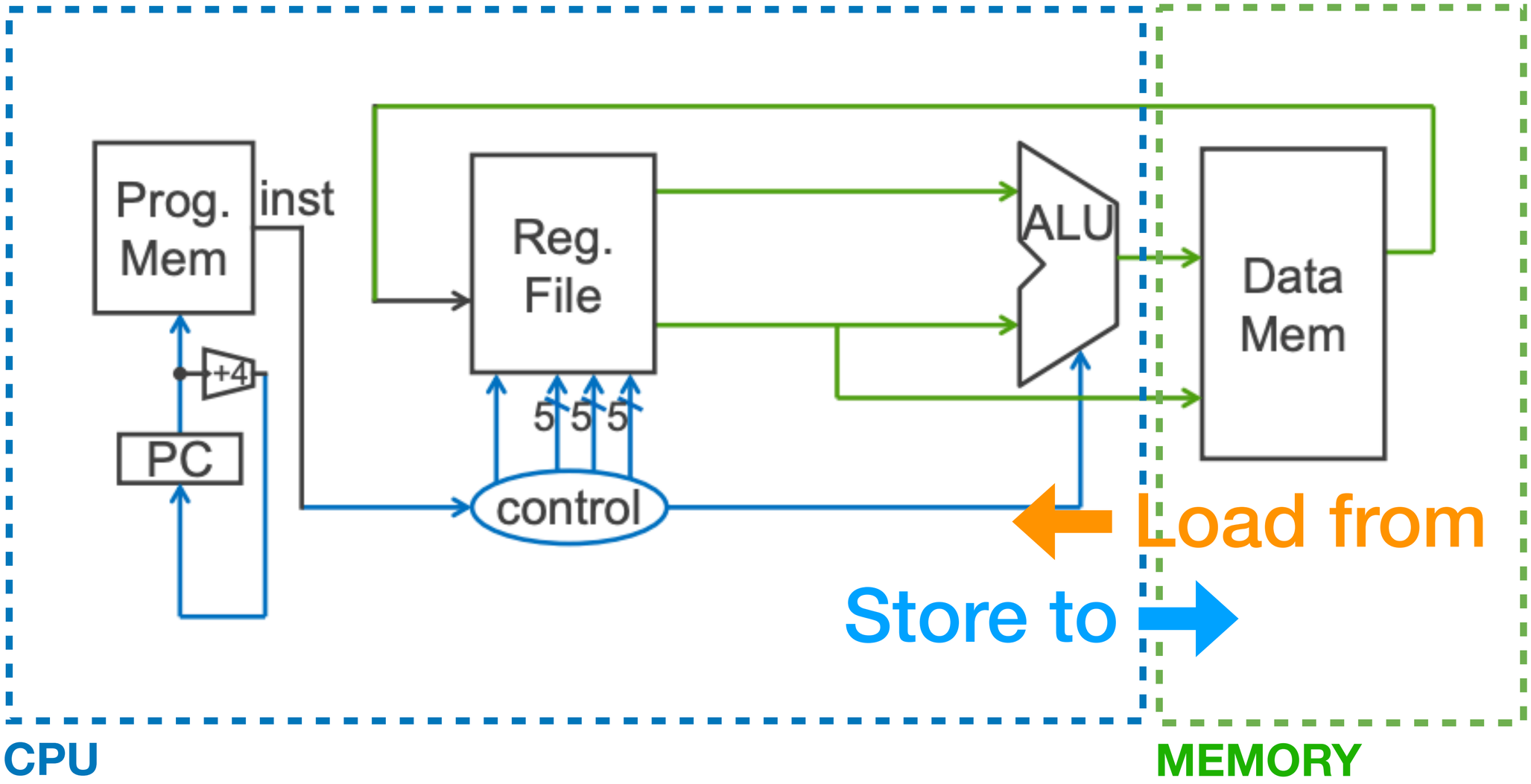
The instructions are sitting on "function addr

In RISC-V, every instruction is **exac**
starts at address 4

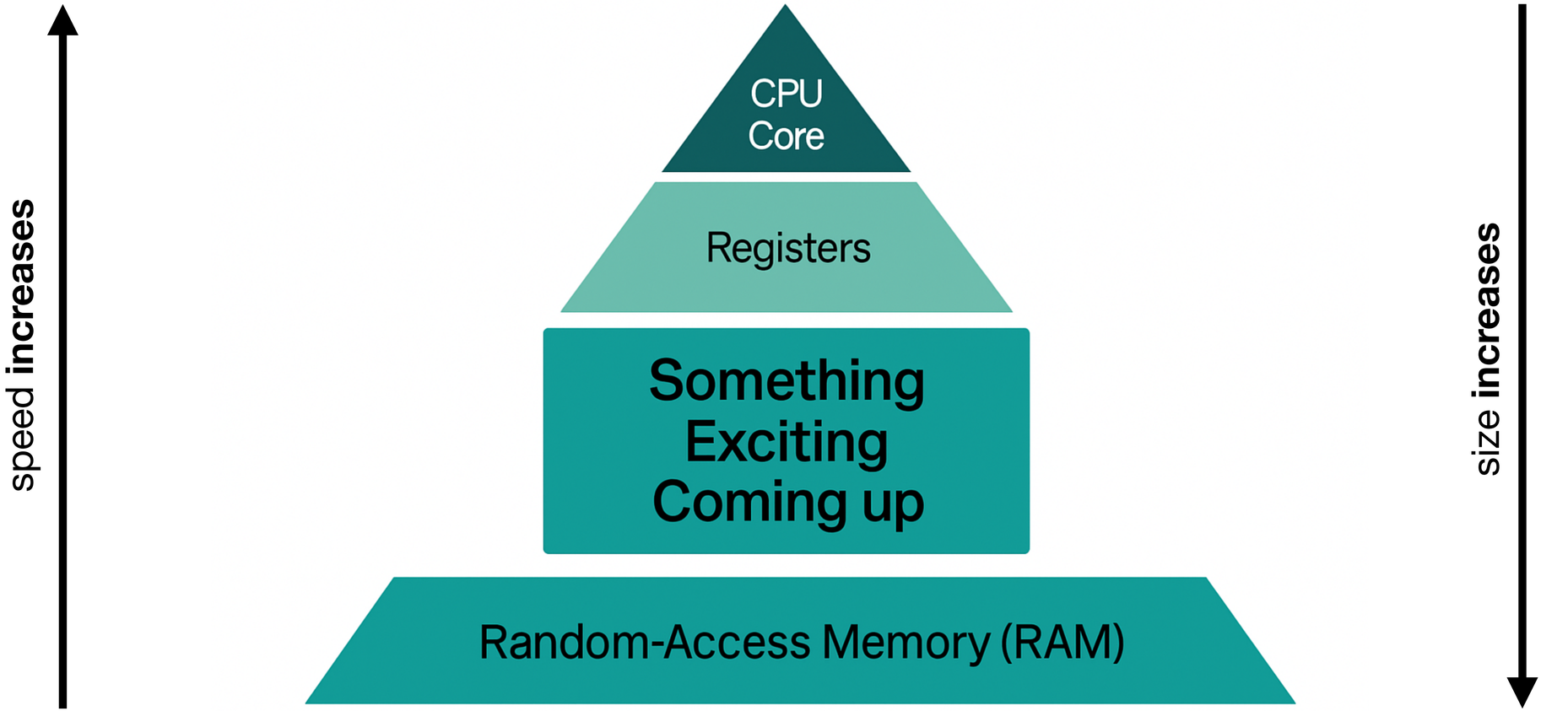
```
address   value (hex)
-------   -----------
 0000     0x33  (LSB)
 0001     0x05
 0002     0xB5
 0003     0x00  (MSB) <- 1 word (bytes 0000–0003)
 0004     0x13  (LSB)
 0005     0x55
 0006     0x15
 0007     0x00  (MSB) <- 1 word (bytes 0004–0007)
 0008     0x67  (LSB)
 0009     0x80
 0010     0x00
 0011     0x00  (MSB) <- 1 word (bytes 0008–0011)
```

RISC-V Data Transfer (Memory Op)

Data Transfer: Load from and Store to Memory



Principle of Locality and Memory Hierarchy



Register versus Memory

Given that:

- Registers: 32 words (256 bytes is RV64 or 128 bytes if RV32)
- DRAM (data memory): billions of bytes (2-96 GB on a typical laptop)

Physics dictates that **smaller is faster**

Registers are **50-500 times faster** than DRAM (one access latency, tens of ns)!

Register versus Memory

10^9 tape/optical robot	Andromeda	2,000 years
10^6 disk	Pluto	2 years
100 memory	New York City	4 hrs
2 “something great coming up”	this campus	15 min
1 register	my head	1 min
[ns]		

Poll

Q: Given int A[100]; in the slide, where does A sit?

C code:

```
int A[100];
g = h + A[3];
```

PollEv.com /gguidi

Or send **gguidi** to **22333**

Load from Memory to Register

C code:

```
int A[100];        # A sits on the stack
g = h + A[3];
```

Load from Memory to Register

C code:

```
int A[100];
g = h + A[3];
```

⬅ Load from (Data flow)

1 word = 4 bytes

Using “load word” (`lw`) in RISC-V:

```
lw  x10, 12(x15)  # Reg x10 gets A[3]
add x11, x12, x10 # g = h + A[3]
```

`x15:` address in memory (pointer to A[0])

`12:` offset in bytes but we load one **word** at a time

→ `lw` updates `x10` in the register file

Load from Memory to Register

C code:

```
int A[100];
g = h + A[3];
```

1 word = 4 bytes

Using “load word” (lw) in RISC-V:

```
lw  x10, 12(x15)  # Reg x10 gets
add x11, x12, x10 # g = h + A[3]
```

→ lw updates x10 in the register file

E: Check the hex for this lw and add

```
address  value (hex)
-------  -----------
 1000    0x03  (LSB)
 1001    0x87
 1002    0xC7
 1003    0x00  (MSB) <- 1 word (bytes 1000–1003)
 1004    0x33  (LSB)
 1005    0x06
 1006    0xA6
 1007    0x00  (MSB) <- 1 word (bytes 1004–1007)
```

Store from Register to Memory

C code:

```
int A[100];
A[10] = h + A[3];
```

1 word = 4 bytes

E: Do the hex translation for the `sw`, and draw the corresponding memory view (i.e., like in the previous slide)

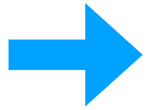
Using “store word” (sw) in RISC-V:

```
lw  x10, 12(x15)  # Temp reg x10 gets A[3]
add x11, x12, x10 # Temp reg x11 gets h + A[3]
sw  x11, 40(x15)  # A[10] = h + A[3]
```

`x15 + 12`

Offset must be a multiple of 4

`x15 + 40`



Store to (Data flow)

→ `sw` does **not** update any register in the register file; only memory `40(x15)` is updated

Poll

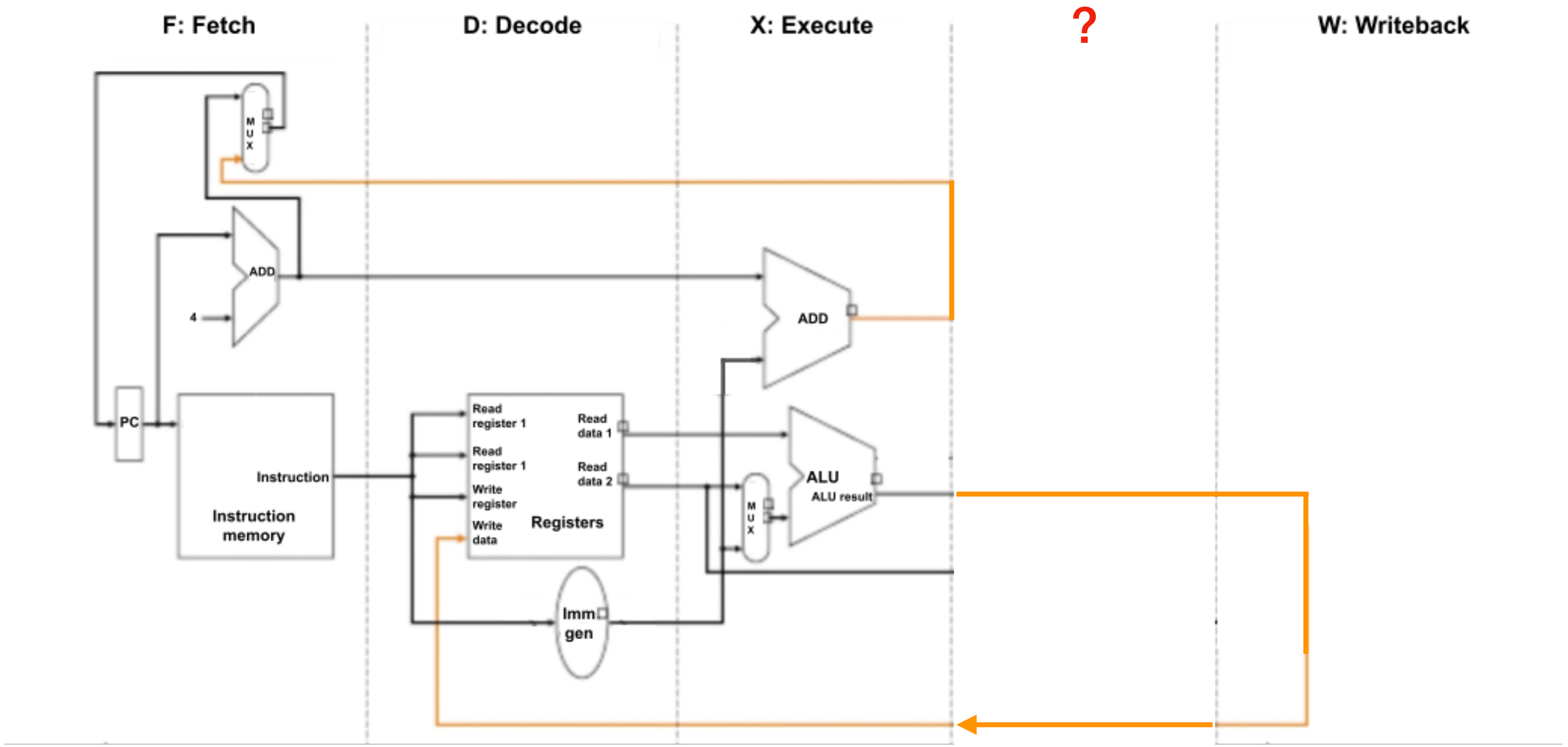
True or False: whether a compiler puts a local variable in a register or on the stack doesn't impact performance

PollEv.com /gguidi

Or send **gguidi** to **22333**

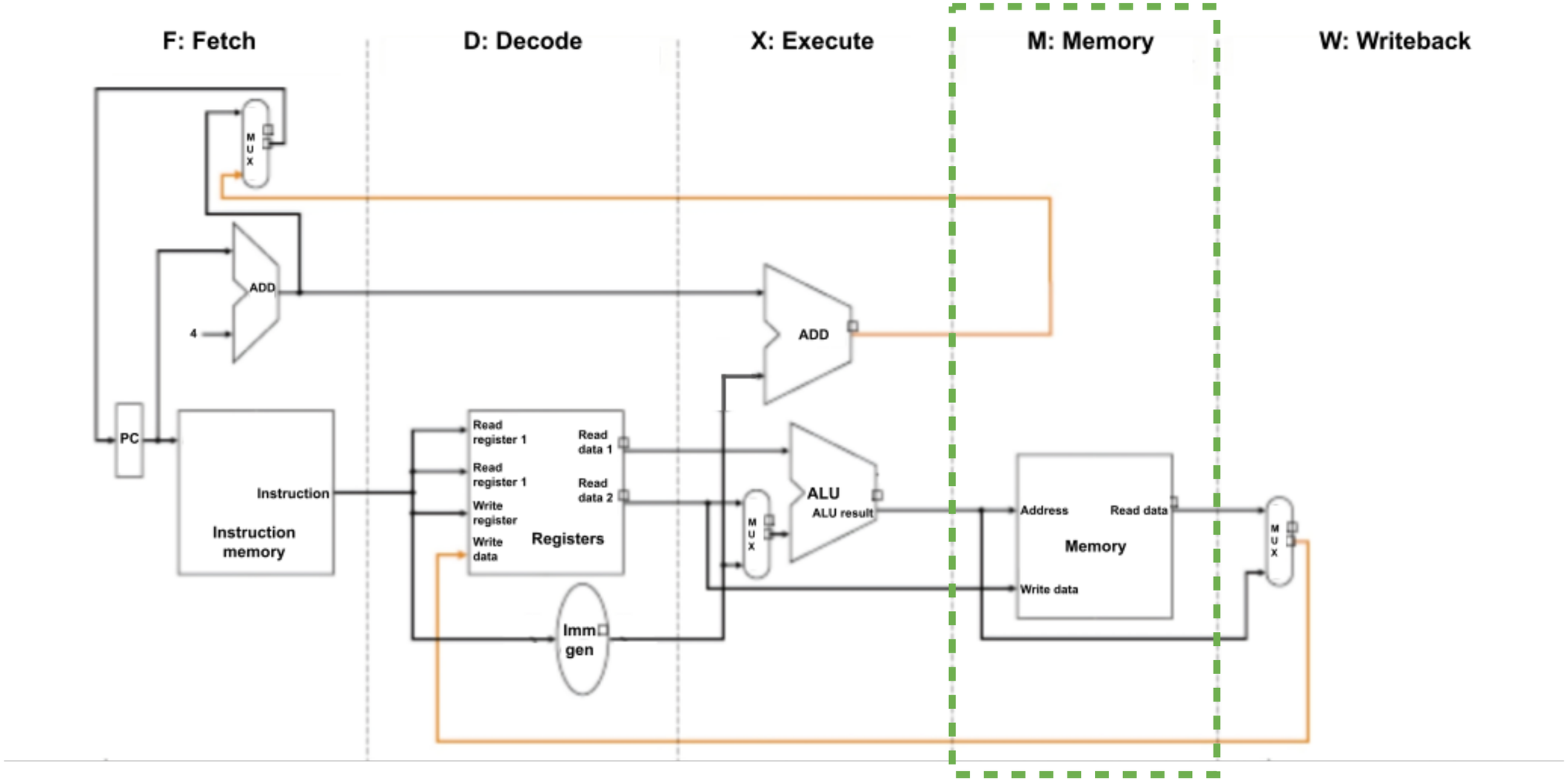
Good time to review the CPU stages

CPU 5 Stages in RISC-V

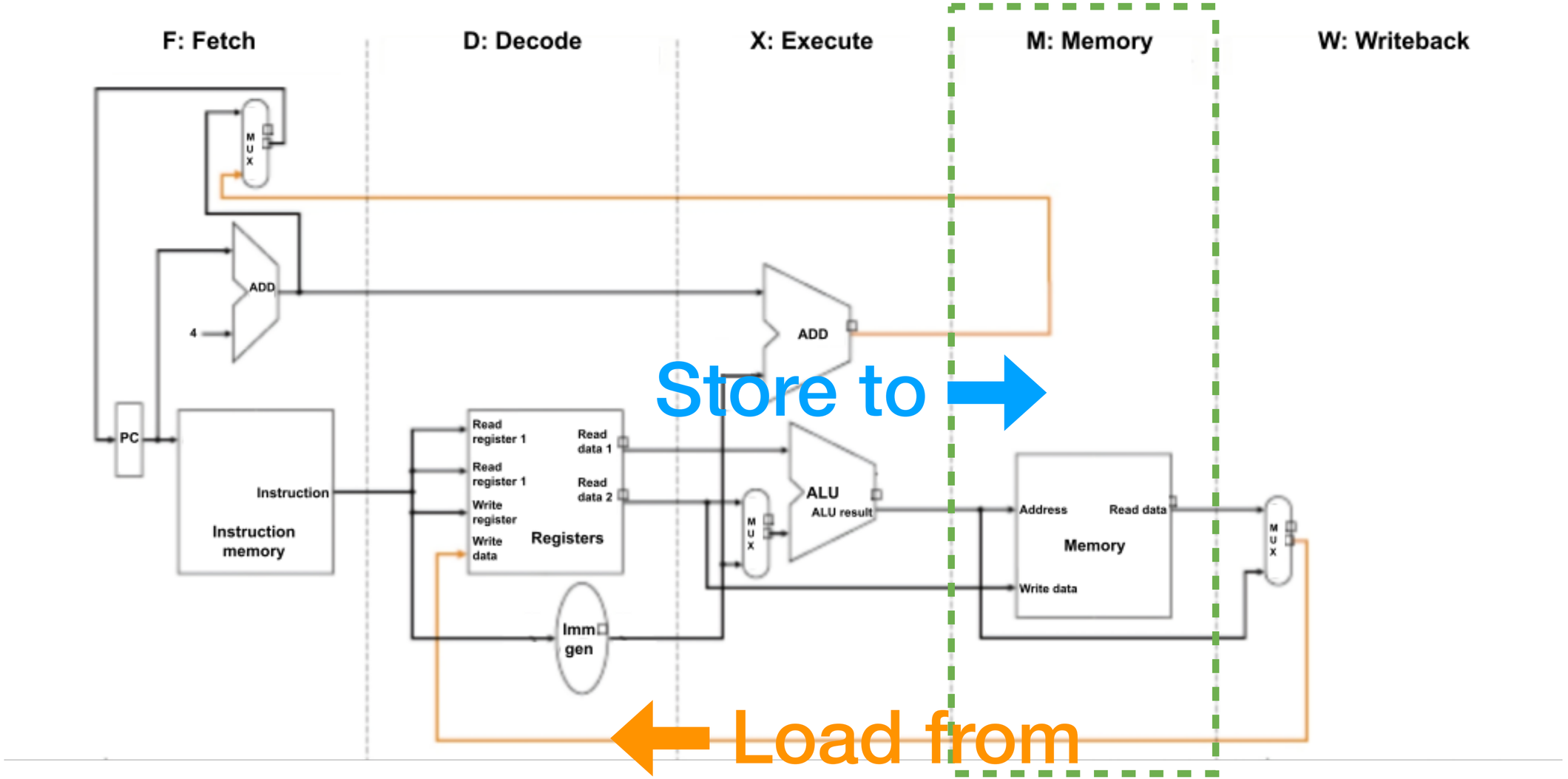


https://www.cs.cornell.edu/courses/cs3410/2025sp/notes/cpu_stages.html

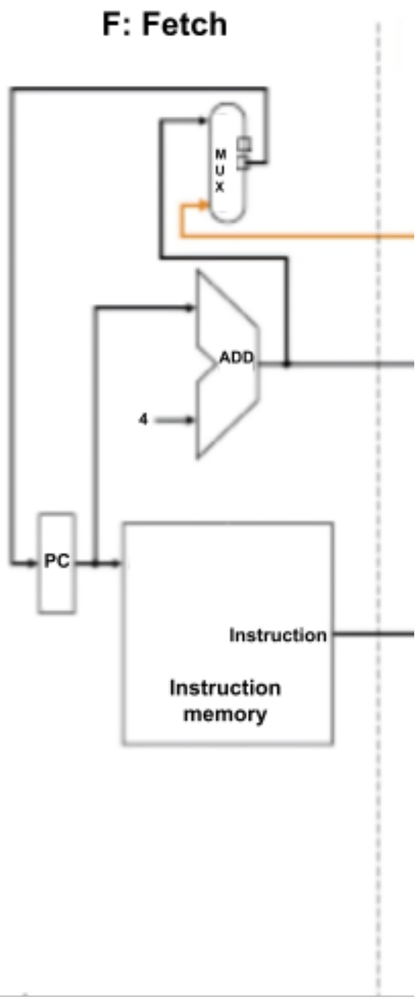
CPU 5 Stages in RISC-V



Data Transfer: Load from and Store to Memory



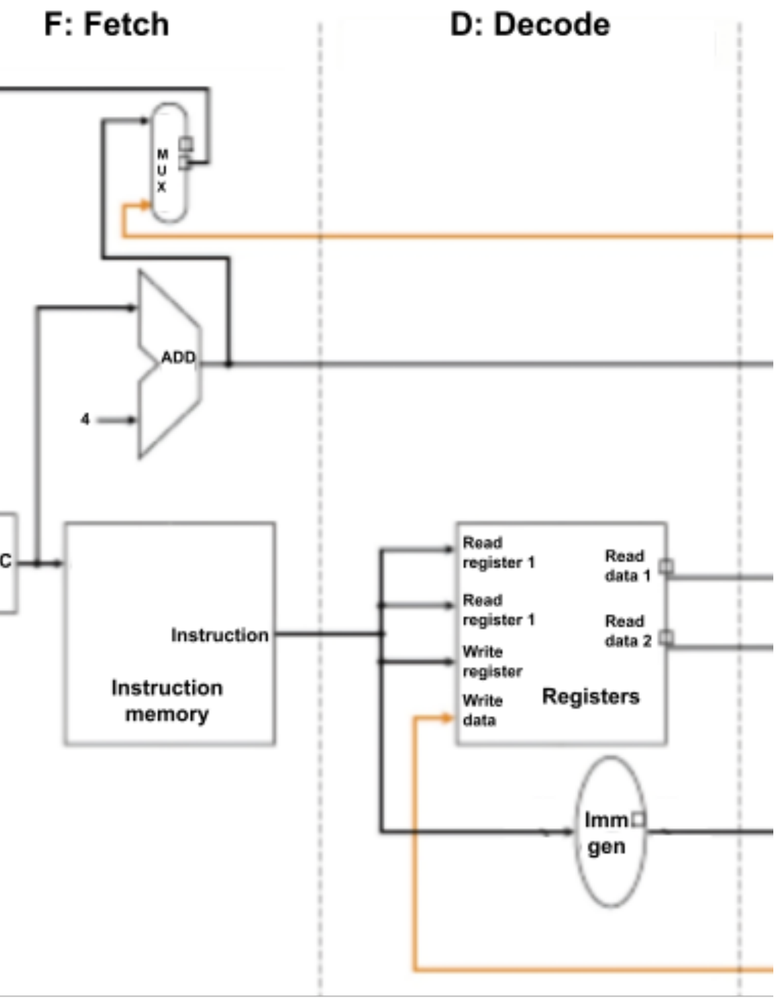
CPU 5 Stages in RISC-V



- **F: fetch** instruction from instruction memory
- Update Program Counter (PC), normally PC + 4, unless a branch/jump (e.g., bne, beq)

Covered in more detailed on Wednesday

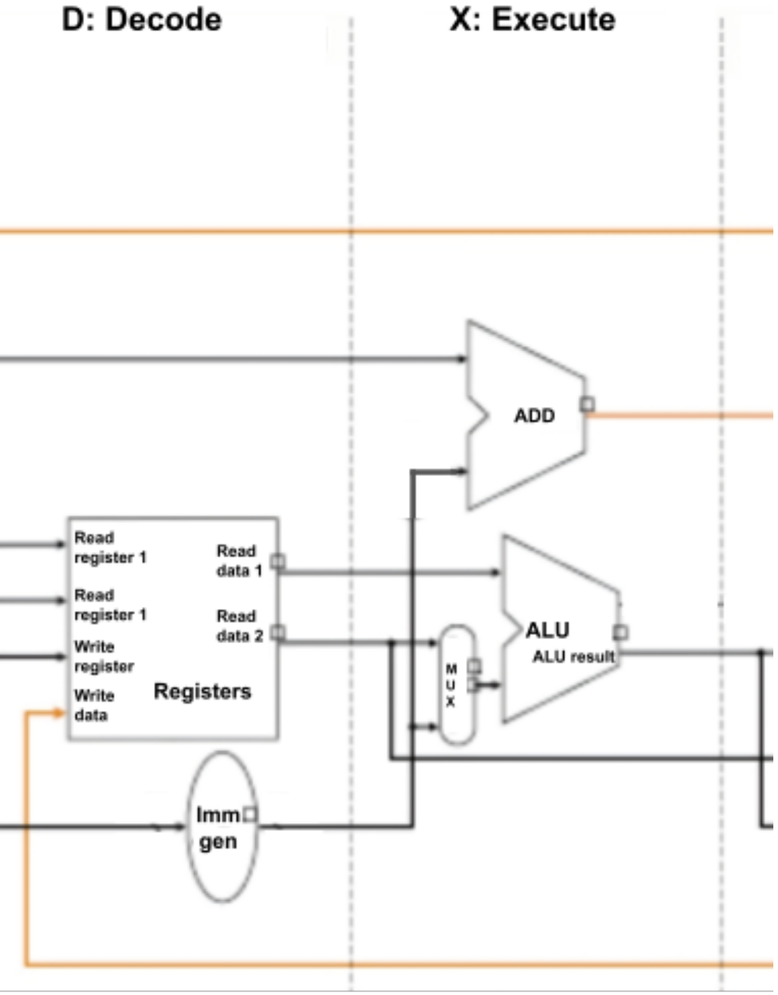
CPU 5 Stages in RISC-V



- **D: decode** opcode, figure out instruction type
- Read registers from the register file
- Generate control signals (ALU operation, memory access, writeback control)

https://www.cs.cornell.edu/courses/cs3410/2025sp/notes/cpu_stages.html

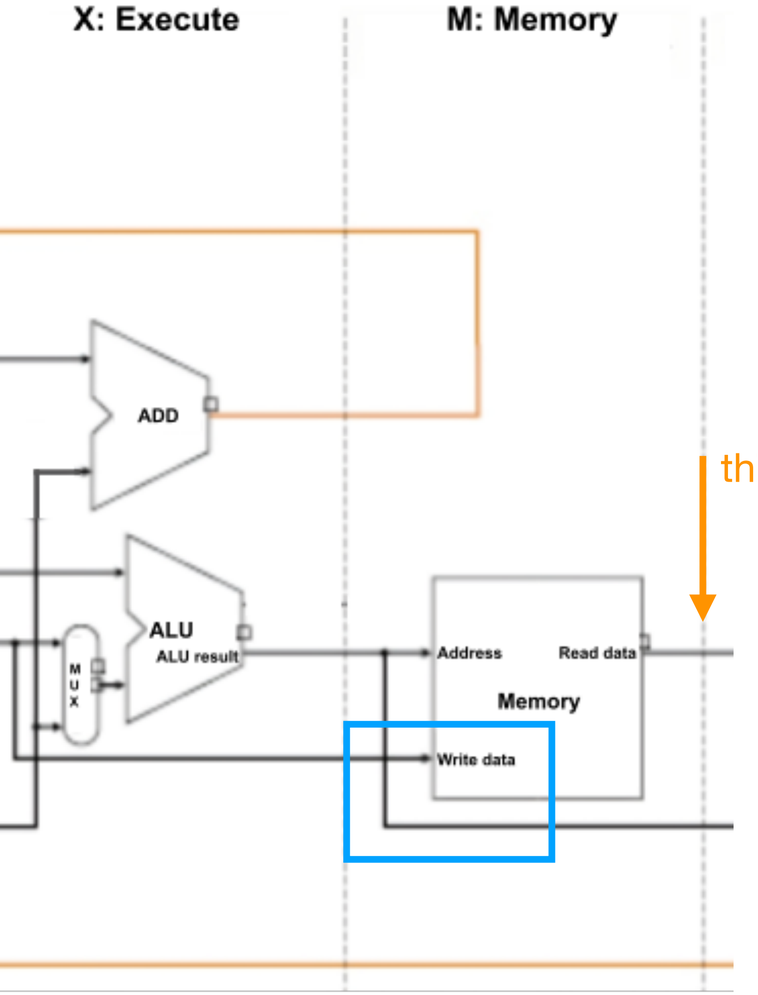
CPU 5 Stages in RISC-V



- **X: execute** — perform ALU operation (add, xor, shift, compare)
- Compute memory address for load (e.g., `lw`) and store (e.g., `sw`)
- Compute branch target address
- Compare registers for branch decisions (e.g., `bne`, `beq`)

Covered in more detailed on Wednesday

CPU 5 Stages in RISC-V



- **M: memory** access
- For load (e.g., `lw`): read data memory

  load updates the register file

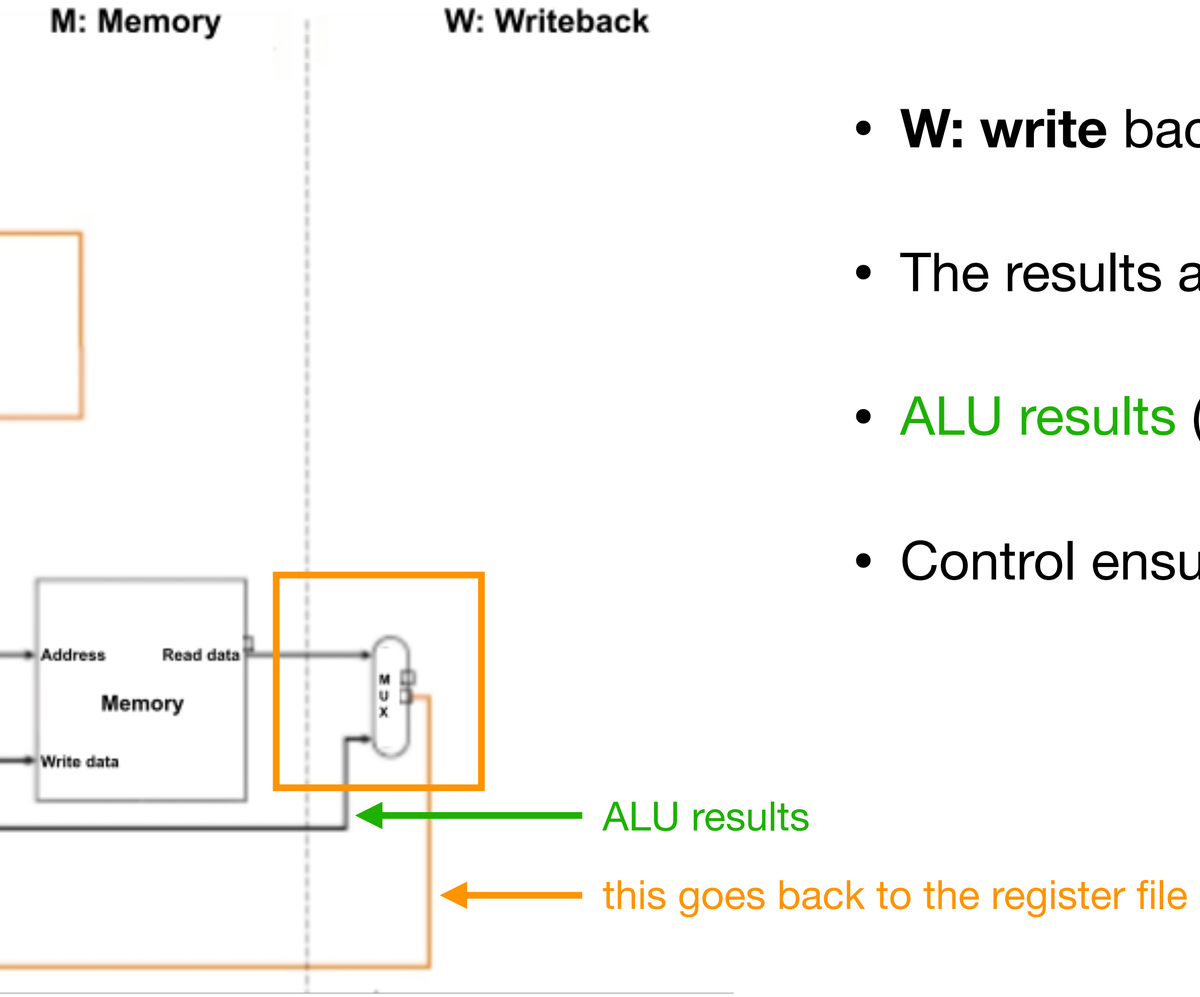
this goes back to the register file

- For store (e.g., `sw`): write register value to data memory

  store only updates data memory

- For ALU-only instructions: nothing happens here

CPU 5 Stages in RISC-V



- **W: write** back
- The results are written back into the register file
- ALU results (e.g., `add`) or memory data (from `lw`)
- Control ensures only the right instructions are written back

CPU 5 Stages in RISC-V

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`add rd, rs1, rs2`	✓	✓	✓		✓

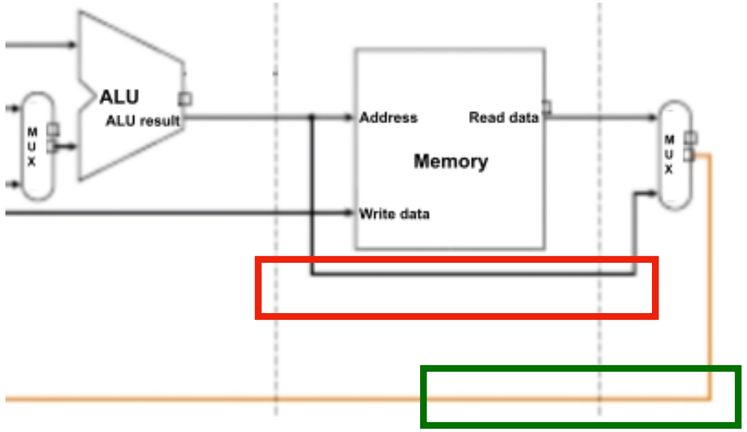
The instruction is fetched from the instruction memory

The registers `rs1` and `rs2` read, control signals set

ALU adds `rs1` + `rs2`

Don't need it, **no** memory access

ALU result written back to the destination register `rd`



CPU 5 Stages in RISC-V

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`srli rd, rs1, imm`	✓	✓			

The instruction is fetched from the instruction memory

Decode instruction, read register `rs1`, extract the **immediate**

`srli x5, x6, 10`

I-type RV32:

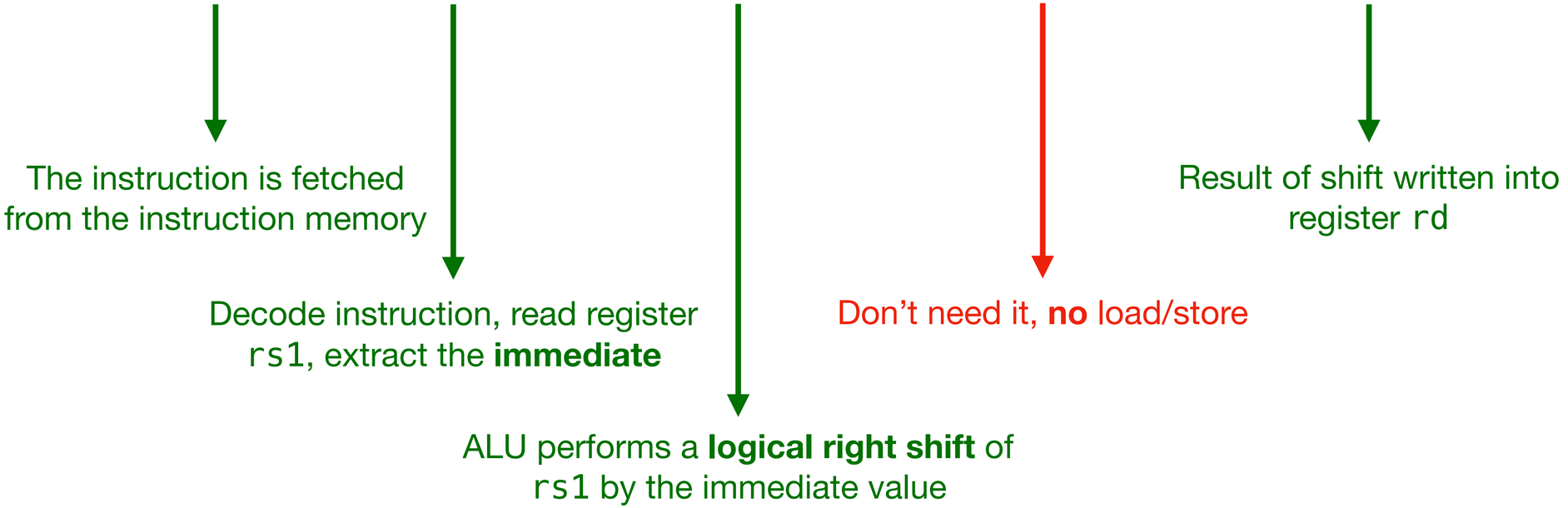
funct7 | shamt[4:0] | rs1 | funct3 | rd | opcode

0000000	01010	00110	101	00101	0010011
n/a	10_{10}	6_{10}		5_{10}	

- The immediate (imm) is **encoded directly inside the 32-bit instruction** (not stored in a register)
- In the **Decode** stage, the control logic and immediate generator unit take the 32-bit instruction, extract the right field, and sign-extend or zero-extend it as required

CPU 5 Stages in RISC-V

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`srli rd, rs1, imm`	✓	✓	✓		✓



CPU 5 Stages in RISC-V

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`lw rd, offset(rs1)`	✓	✓	✓	✓	✓

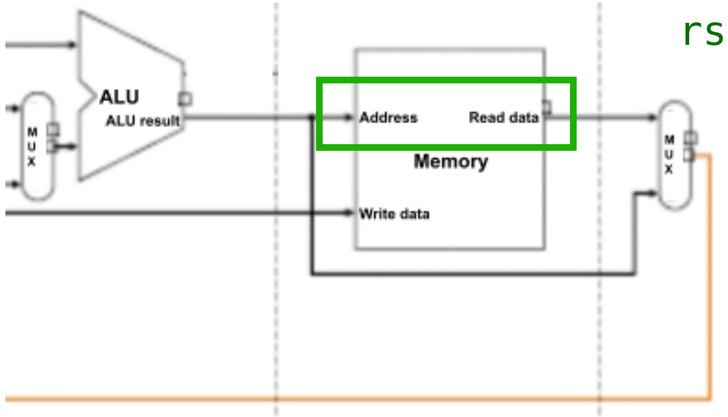
The instruction is fetched from the instruction memory

Decode instruction, read register `rs1`, extract the **offset (imm)**

ALU computes effective **address** = `rs1 base + offset`

Data memory **accessed**, value loaded

The loaded value is written back into destination register `rd`



CPU 5 Stages in RISC-V

Instruction	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
`sw rs2, offset(rs1)`	✓	✓	✓	✓	

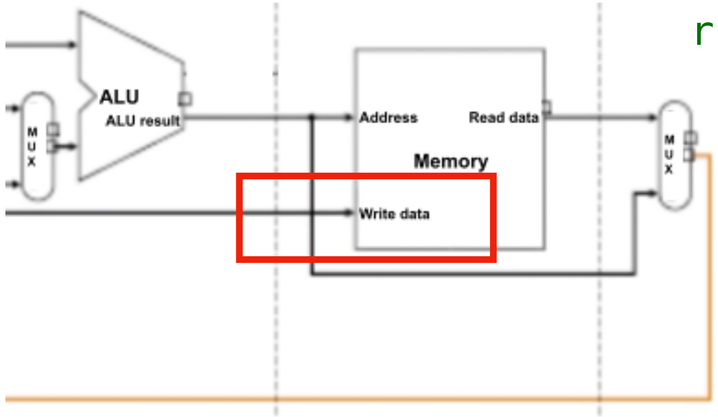
The instruction is fetched from the instruction memory

Decode instruction, read register `rs2`, `rs1`, extract the **offset**

ALU computes effective **address** = `rs1 base + offset`

Data memory **accessed**, `rs2` value **stored**

There's nothing to write back into registers



Ok, back to data transfer

Loading and Storing Bytes

"load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

Loading and Storing Bytes

"load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

"load word"

Uses same format as `lw` and `sw:` "store word"

- E.g., `lb x10, 3(x11)` pointer to memory
  offset in bytes (doesn't have to be multiple of 4)

1. Compute the effective address = content of `x11` + `3`
   - Let us assume `3(x11)` contains the value `4` (decimal) and we use **16-bit** register
   - `4` (decimal) = `0000 0000 0000 0100` (16-bit binary)

Loading and Storing Bytes

"load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

"load word"

Uses same format as `lw` and `sw:` "store word"

- E.g., `lb x10, 3(x11)` pointer to memory
  offset in bytes (doesn't have to be multiple of 4)

1. Compute the effective address = content of `x11` + `3`
   - Let us assume `3(x11)` contains the value `4` (decimal) and we use **16-bit** register
   - `4` (decimal) = `0000 0000 0000 0100` (16-bit binary)
2. Then, **load 1 byte** from memory at that address
   - The loaded **byte** is `0000 0100` (8-bit binary)

Loading and Storing Bytes

"load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

"load word"

Uses same format as `lw` and `sw:` "store word"

- E.g., `lb x10, 3(x11)` — `3(x11)`: pointer to memory; `3`: offset in bytes (doesn't have to be multiple of 4)

2. Then, **load 1 byte** from memory at that address
   - The loaded **byte** is `0000 0100` (8-bit binary)
3. Finally, **sign-extend** the byte to 16 bits (we assumed `x10` is a **16-bit** register)
   - The final value in `x10` is `0000 0000 0000 0100` (16-bit binary) = `4` (decimal)

Loading and Storing Bytes

"load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

"load word"

Uses same format as `lw` and `sw:` "store word"

- E.g., `lb x10, 3(x11)` pointer to memory
  offset in bytes (doesn't have to be multiple of 4)

1. Compute the effective address = content of `x11` + `3`
   - Ok **but what if** 3(x11) contains the value `3410` (decimal) and we use **16-bit** register
   - `3410` (decimal) = `0000 1101 0101 0010` (16-bit binary)
2. Then, **load 1 byte** from memory at that address
   - The loaded **byte** is `0101 0010` (8-bit binary)

Loading and Storing Bytes

"load byte"

In addition to `lw` and `sw`, RISC-V has `lb` and `sb` "store byte"

"load word"

Uses same format as `lw` and `sw:` "store word"

- E.g., `lb x10, 3(x11)` — `3(x11)`: pointer to memory; `3`: offset in bytes (doesn't have to be multiple of 4)

2. Then, **load 1 byte** from memory at that address
   - The loaded **byte** is `0101 0010` (8-bit binary)
3. Finally, **sign-extend** the byte to 16 bits (we assumed `x10` is a **16-bit** register)
   - The final value in `x10` is `0000 0000 0101 0010` (16-bit binary)
   - Ops! `0000 0000 0101 0010` (16-bit binary) = `82` (decimal) != `3410` (decimal)

`lbu`

In addition to `lw` and `sw`, RISC-V has `lb` and `sb`

"load word"

Uses same format as `lw` and `sw:` "store word"

- E.g., `lb x10, 3(x11)` pointer to memory

  offset in bytes (doesn't have to be multiple of 4)

`lbu` = unsigned load byte It doesn't need to preserve the sign

`lbu`

In addition to `lw` and `sw`, RISC-V has `lb` and `sb`

"load word"

Uses same format as `lw` and `sw:` "store word"

- E.g., `lb x10, 3(x11)` pointer to memory

offset in bytes (doesn't have to be multiple of 4)

`lbu` = unsigned load byte — It doesn't need to preserve the sign: **zero extension**

But no `sbu`, why? — It doesn't matter! You're just writing the low 8 bits of a register directly to memory, so **no** extension

addi

The following two instructions:

```
lw  x10, 12(x15)  # temp reg x10 gets A[3]
add x12, x12, x10 # reg x12 = reg x12 + A[3]
```

Replace `addi`:

```
addi x12, value   # put value in A[3]
```

This involves going to New York City (load from memory)

The add immediate is so common that is deserves its own instruction