RISC-V: Data Memory & Control Flow

CS 3410: Computer System Organization and Programming

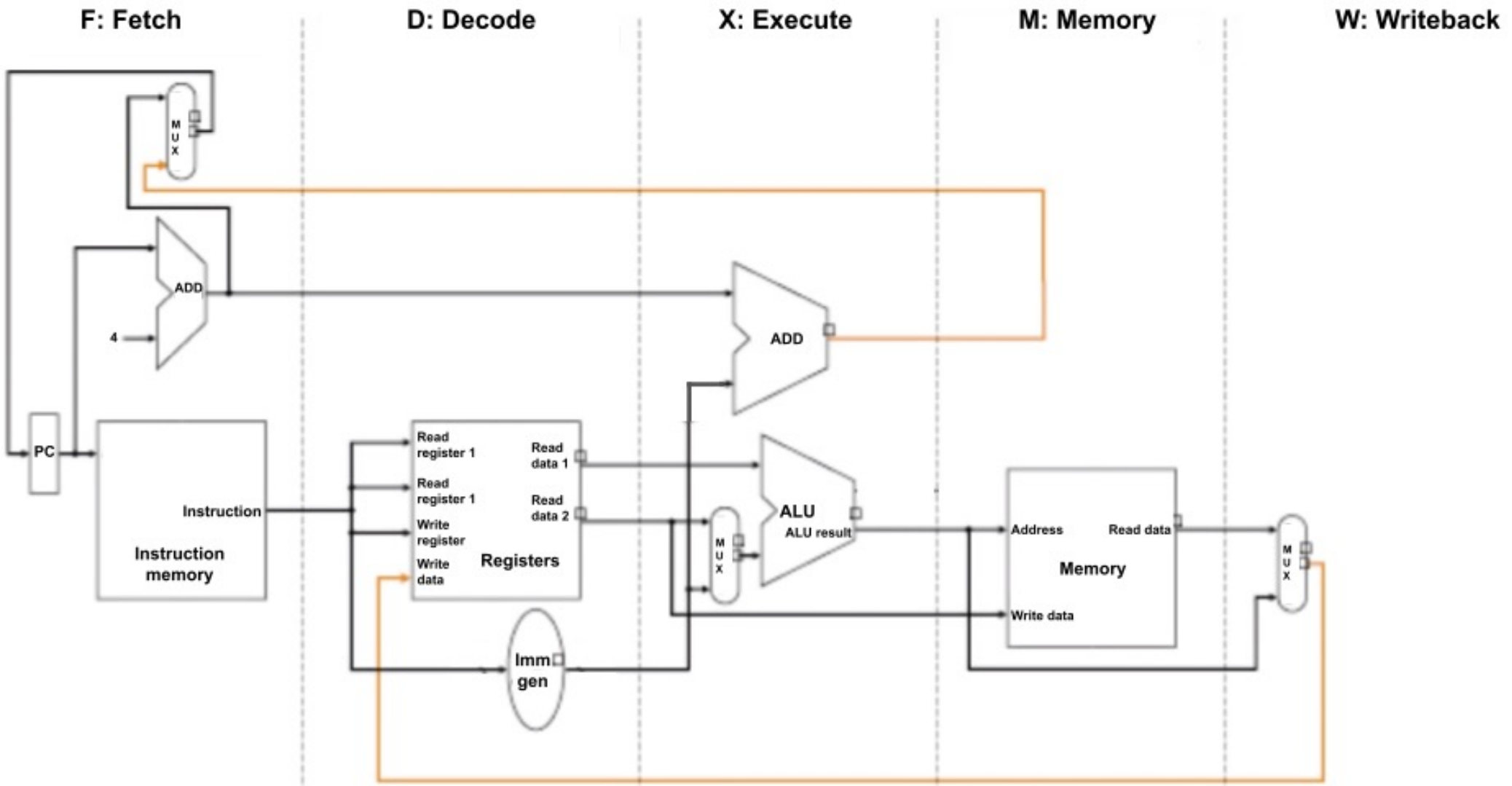
Spring 2025

[K. Bala, A. Bracy, G. Guidi, E. Sirer, A. Sampson, Z. Susag, and H. Weatherspoon]

Today's Plan

- Memory Instructions
  - Memory Hierarchy
  - Loading & Storing a byte
  - Loading & Storing multiple bytes
- Control Flow Instructions
  - Additional Branch Instructions
  - Loops

Last Time


F: Fetch
D: Decode
X: Execute
M: Memory
W: Writeback
MUX
ADD
4
ADD
PC
Instruction
Instruction memory
Read register 1
Read register 1
Write register
Write data
Read data 1
Read data 2
Registers
Imm gen
MUX
ALU
ALU result
Address
Read data
Memory
Write data
MUX

Which of the following statements are true?

All instructions require an access to program memory

All instructions require an access to data memory

All instructions write to the register file

Some RISC-V instructions are shorter than 32 bits

A & C

Which of the following statements are true?

All instructions require an access to program memory

0%

All instructions require an access to data memory

0%

All instructions write to the register file

0%

Some RISC-V instructions are shorter than 32 bits

0%

A & C

0%

Which of the following statements are true?

All instructions require an access to program memory

0%

All instructions require an access to data memory

0%

All instructions write to the register file

0%

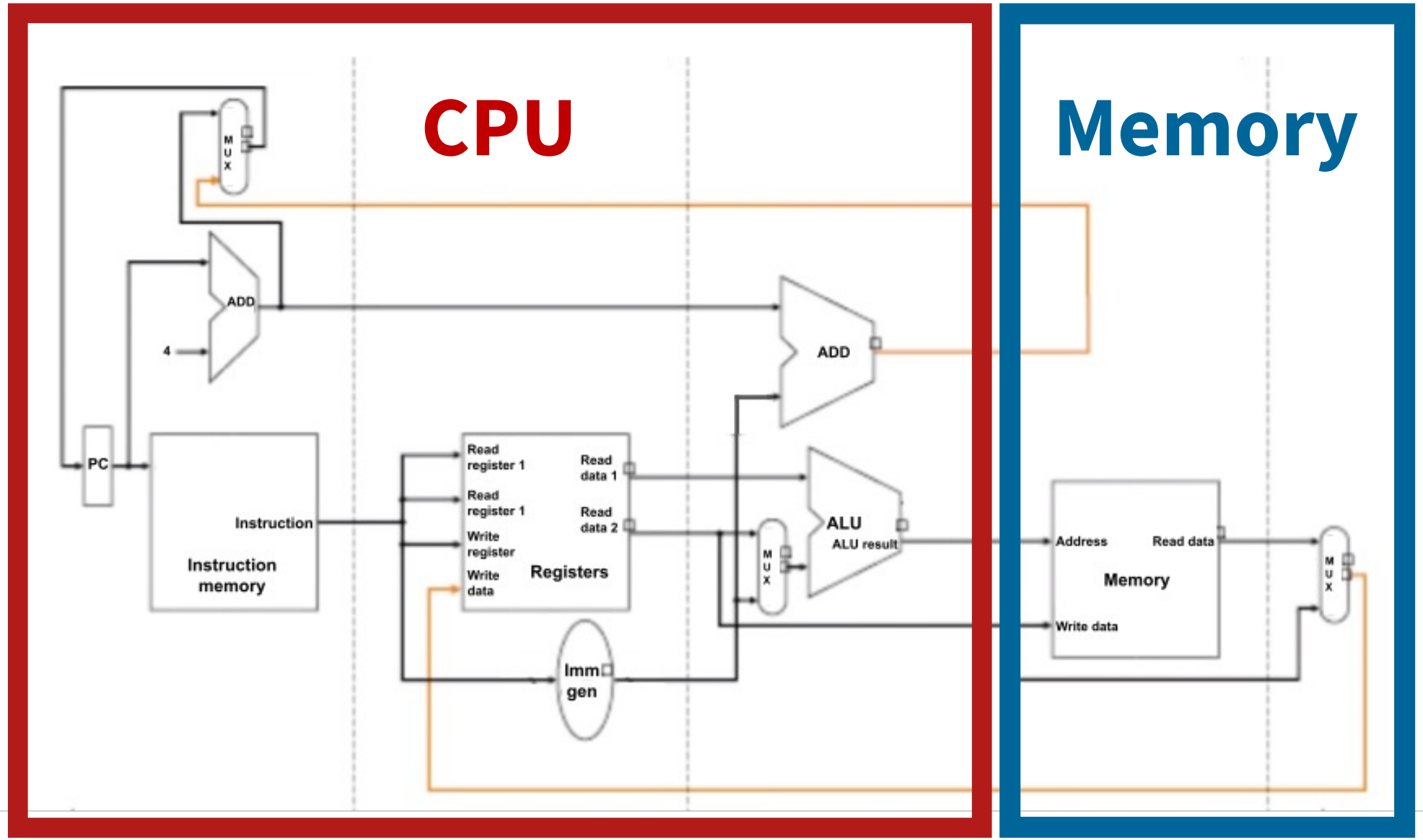
Some RISC-V instructions are shorter than 32 bits

0%

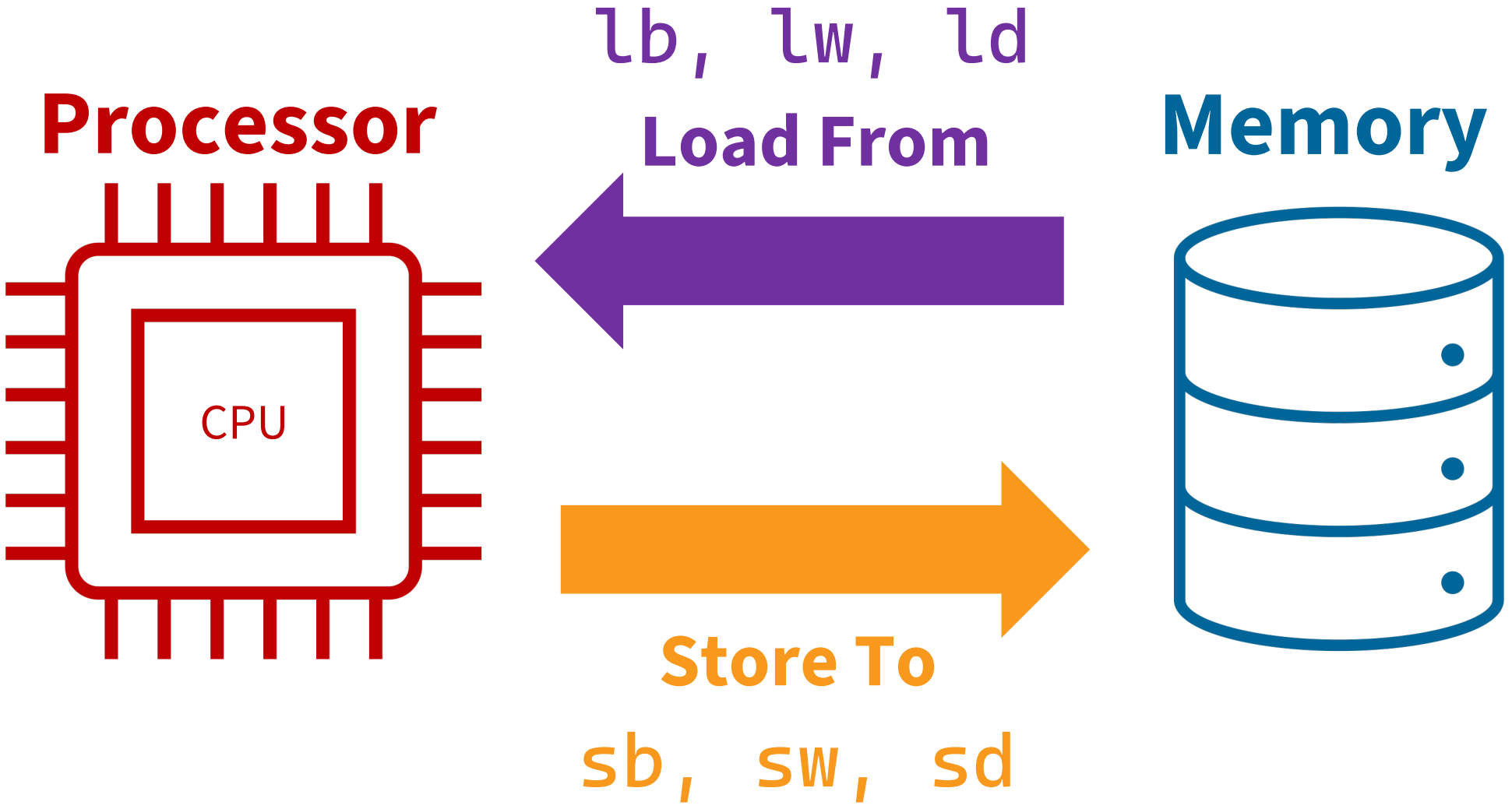
A & C

0%

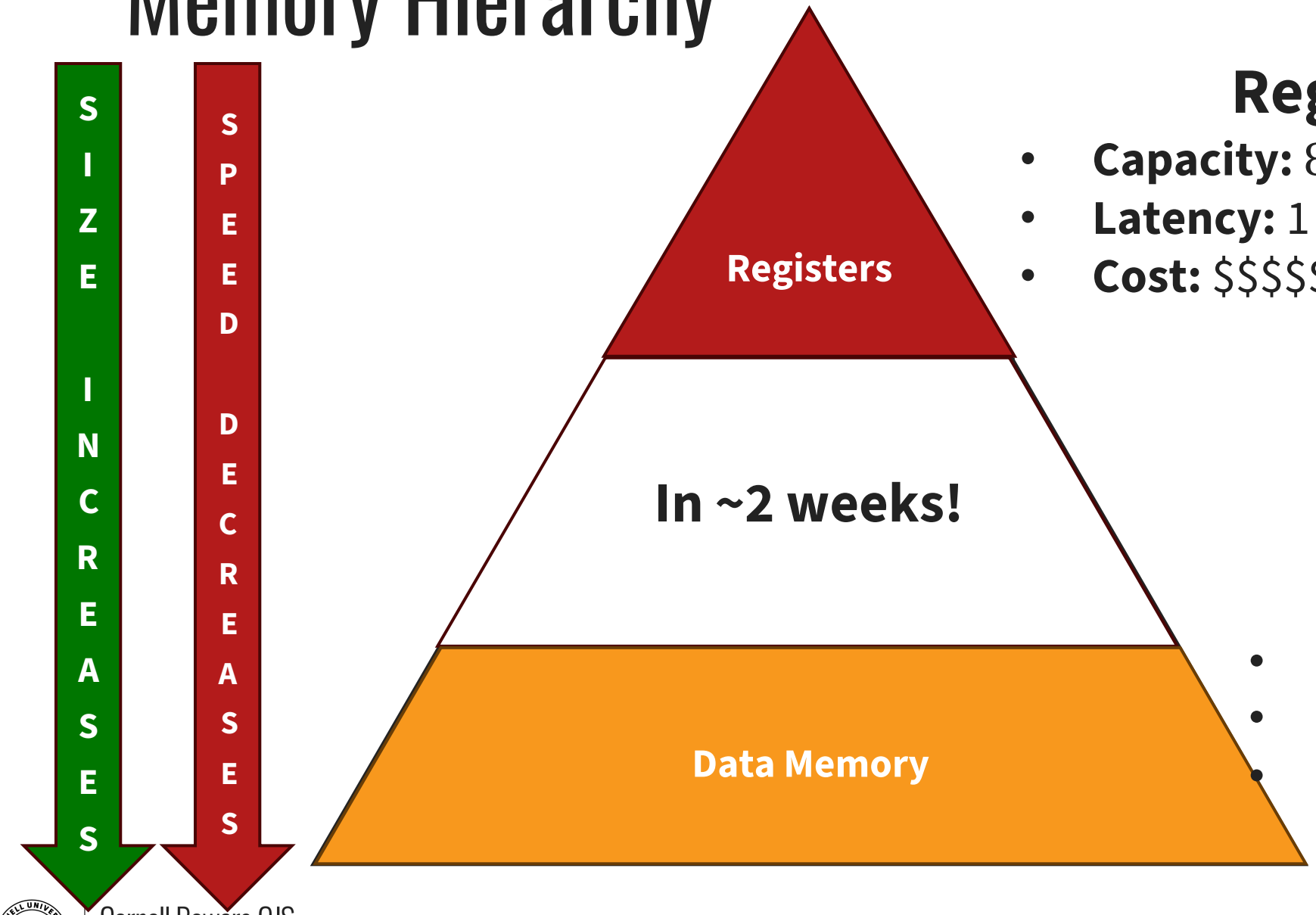
Last Time


CPU
Memory
M
U
X
ADD
4
ADD
PC
Instruction
Instruction memory
Read register 1
Read register 1
Write register
Write data
Registers
Read data 1
Read data 2
ALU
ALU result
Imm gen
Address
Read data
Memory
Write data

Last Time



Memory Hierarchy



Registers

- **Capacity:** $8 \times 31 = 248$ bytes
- **Latency:** 1 cycle ~ 1ns
- **Cost:** $$$$$

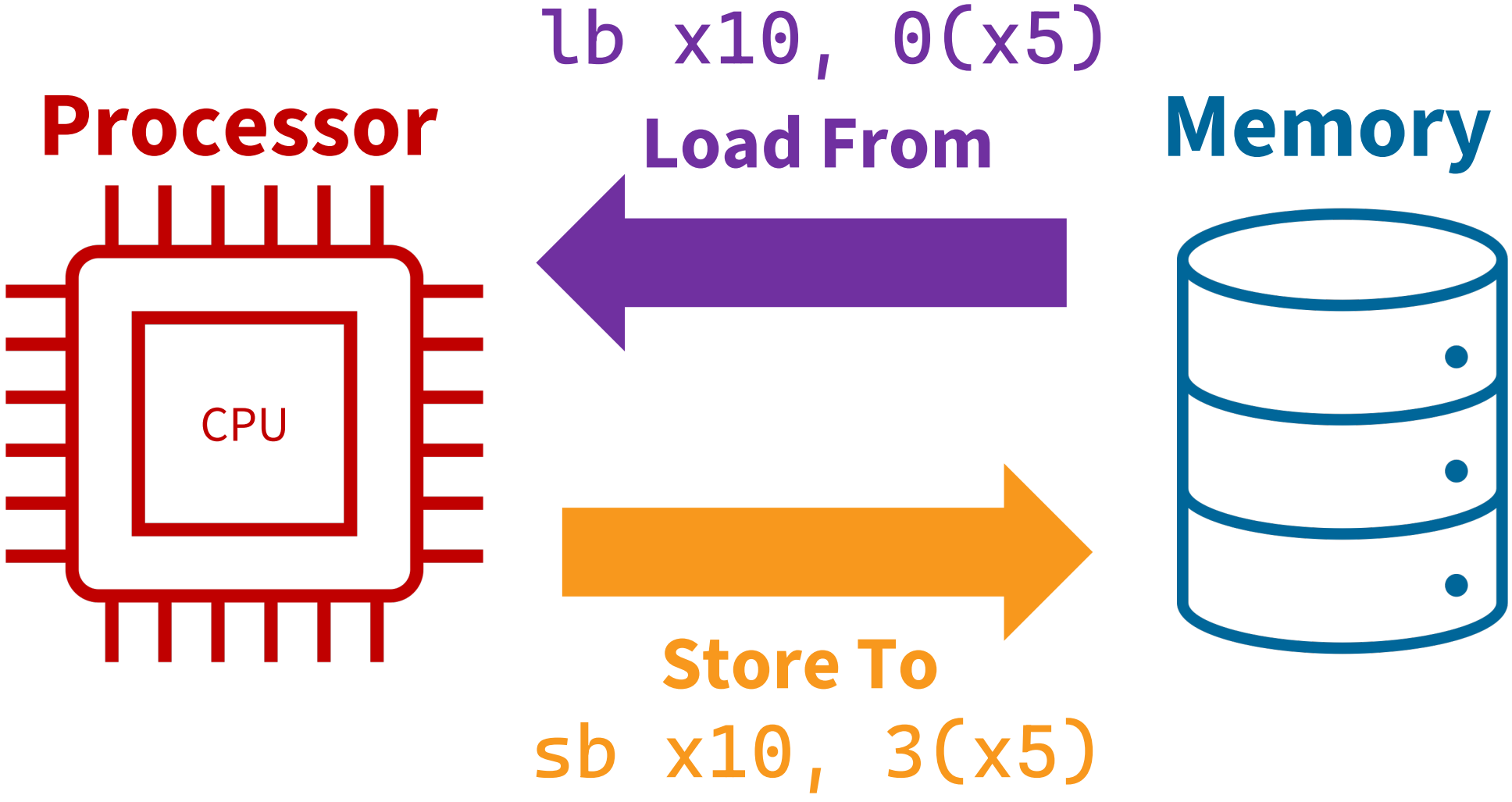
Data Memory

- **Capacity:** $2^{64} = 16$ *exabytes*
- **Latency:** 100 cycles ~ 100ns
- **Cost:** $$

Loading & Storing a Byte

8 bits. No more. No less.

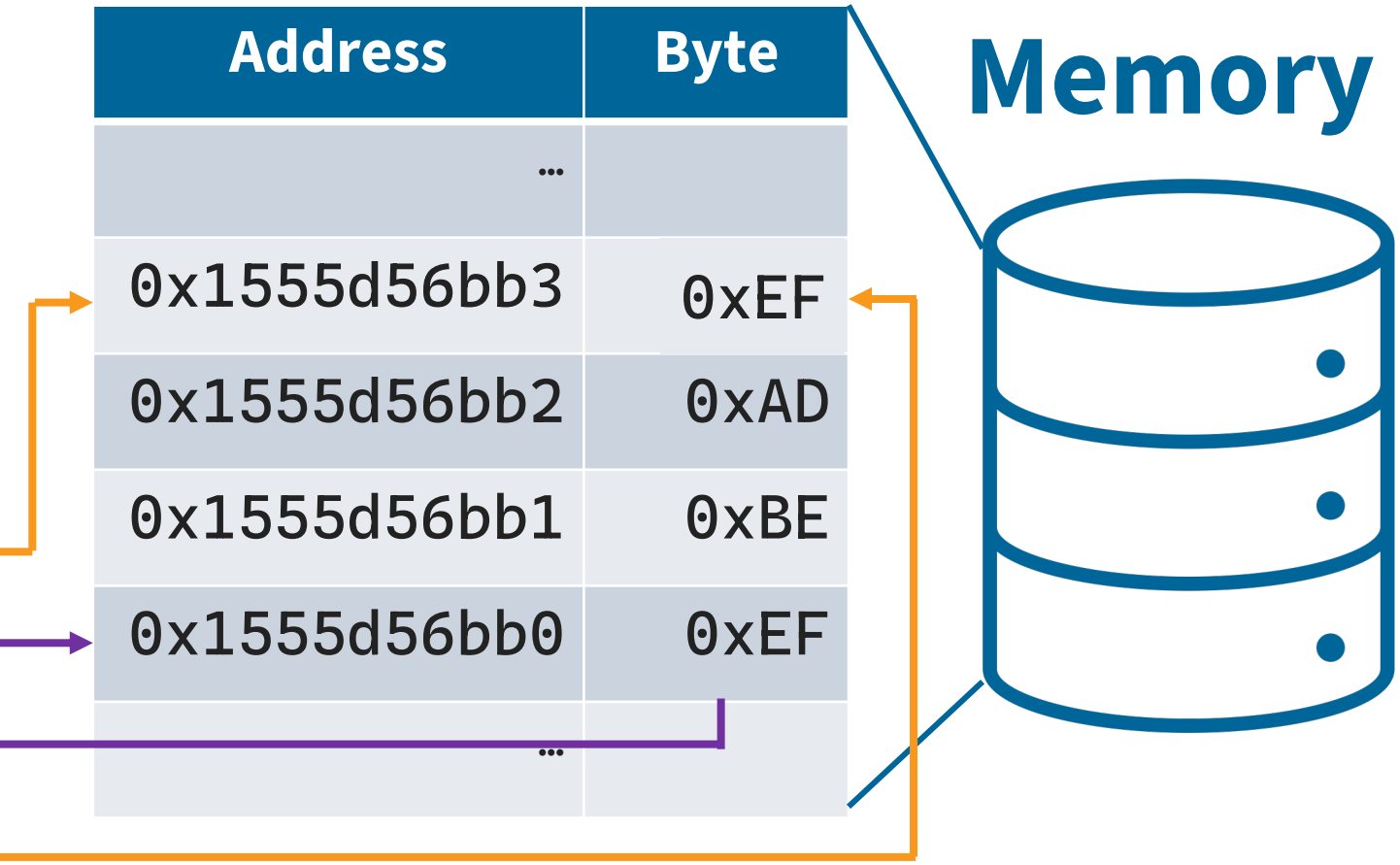
Loading & Storing a Byte



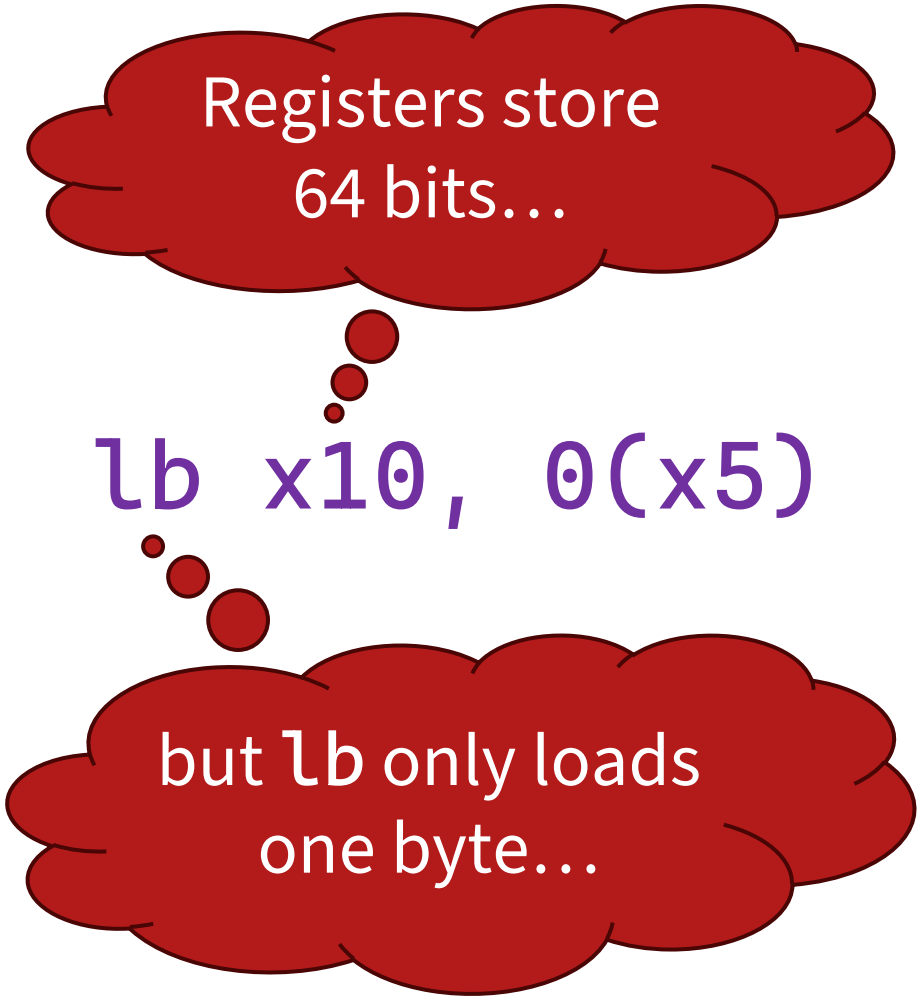
Loading & Storing a Byte

```
x5 = 0x1555d56bb0
```

```
lb x10, 0(x5)
sb x10, 3(x5)
```



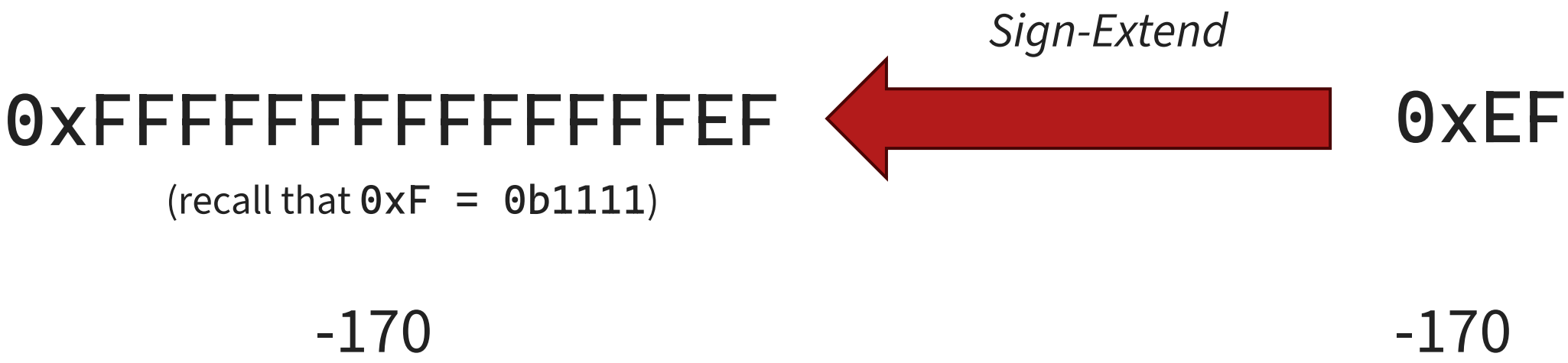
Extension



Sign-Extension

Fill upper bits with the MSB

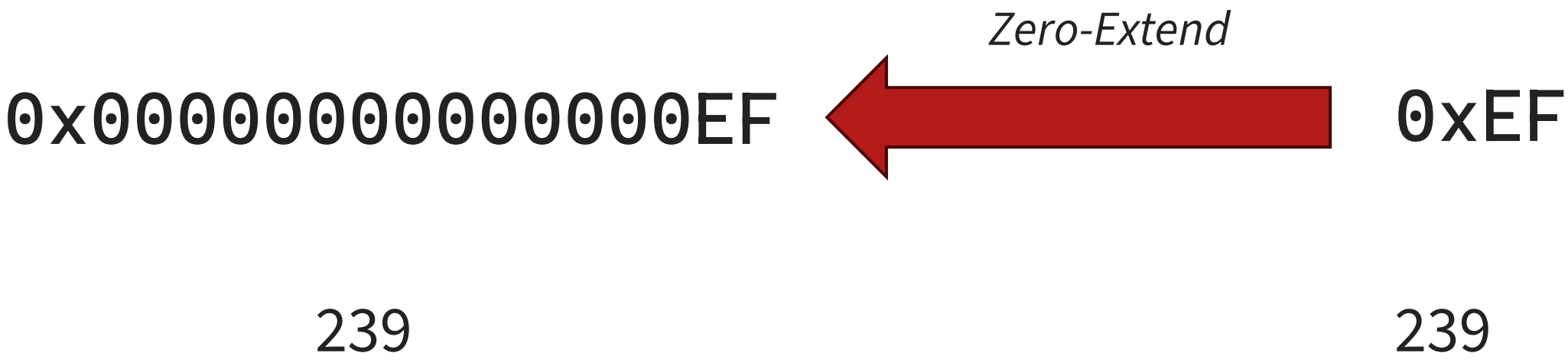
Ex. `lb x10, 0(x5)`



Zero-Extension

Fill upper bits with **0**

Ex. `lbu x10, 0(x5)`



Loading & Storing Multiple Bytes

8 bits. Often more. Still no less.

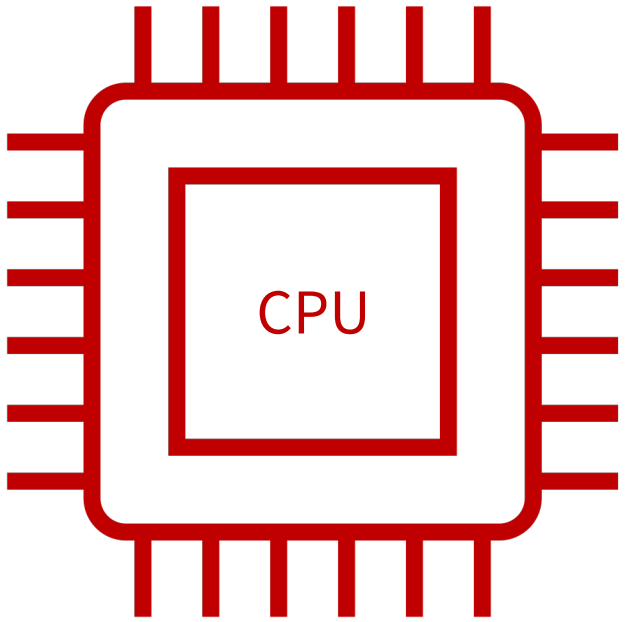
Loading & Storing Multiple Bytes

A word is 4 bytes (32 bits)

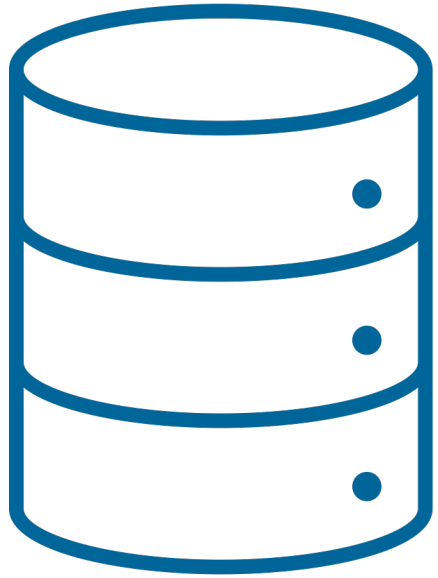
```
lw x10, 0(x5)
```

Load From

Processor



Memory



Store To

```
sw x10, 3(x5)
```

Endianness

*The order in which **bytes** are stored in memory*

What we use!

Ex: 0xDEADBEEF

Little Endian

*Least significant **byte** at lowest memory address*

Address	Byte
a+3	0xDE
a+2	0xAD
a+1	0xBE
a	0XEF

Big Endian

*Most significant **byte** at lowest memory address*

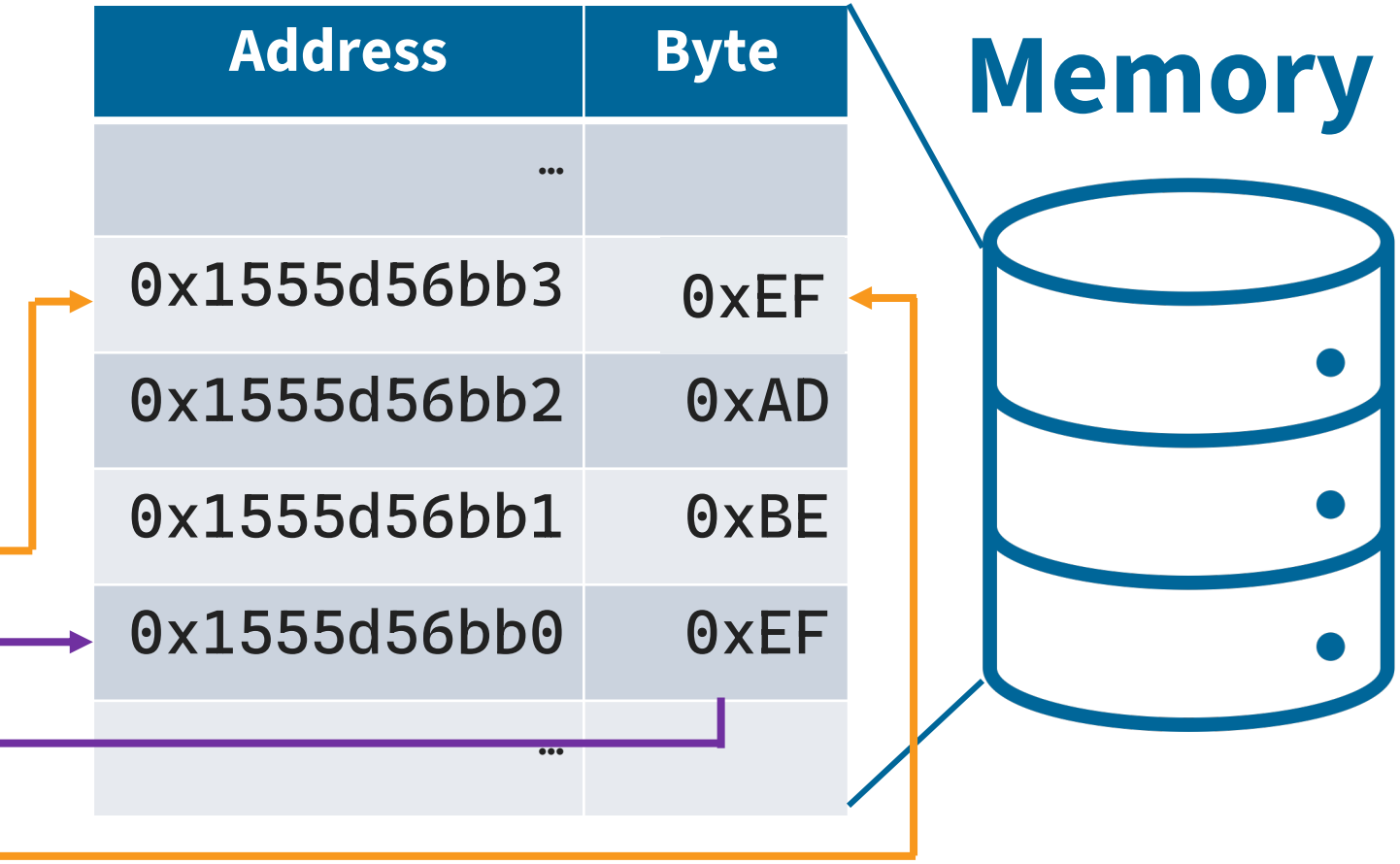
Address	Byte
a+3	0xEF
a+2	0xBE
a+1	0xAD
a	0XDE

Loading & Storing Multiple Bytes

```
x5 = 0x1555d56bb0
```

```
lw x10, 0(x5)
```

```
sw x10, 3(x5)
```



Poll Everywhere: Translate *x = *y

Goal: translate *x = *y into RISC-V

```
1: add x3,    x5, zero
2: add x5,    x3, zero
3: lw  x3, 0(x5)
4: lw  x5, 0(x3)
5: lw  x8, 0(x5)
6: sw  x8, 0(x3)
7: lw  x5, 0(x8)
8: sw  x3, 0(x8)
```

Multiple Choice:

A. 1

B. 2

C. 3

D. 4

E. First 5, then 6

F. First 6, then 5

G. First 7, then 8

Translate *x = *y

1 0%

2 0%

3 0%

4 0%

First 5, then 6 0%

First 6, then 5 0%

First 7, then 8 0%

Control Flow Instructions

Computers Need to Make Decisions

Programs frequently need to perform actions based on certain *conditions*

Conditional Branch Instructions:

```
beq rs1, rs2, label # branch if equal
bne rs1, rs2, label # branch if not equal

blt rs1, rs2, label # branch if less than
bge rs1, rs2, label # branch if greater than or equal to

bltu rs1, rs2, label # (unsigned) branch if less than
bltu rs1, rs2, label # (unsigned) branch if greater than…
```

Example: `if` condition

C

```c
if (x1 == x2) {
   x3 += 42;
}
x3 += 27;
```

RISC-V Assembly

```
   bne x1, x2, EXIT


EXIT:
```

Example: `if` condition

C

```
if (x1 == x2) {
   x3 += 42;
}
x3 += 27;
```

RISC-V Assembly

```
   bne x1, x2, EXIT
   addi x3, x3, 42
EXIT:
```

Example: `if` condition

C

```
if (x1 == x2) {
   x3 += 42;
}
x3 += 27;
```

RISC-V Assembly

```
   bne x1, x2, EXIT
   addi x3, x3, 42
EXIT:
   addi x3, x3, 27
```

Unconditional Branch

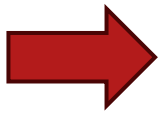
Unconditional branch: always branch (jump)

```
j label # jumps to label
```

Example: `if-else`

Assuming the below translations, **compile** *if-else* block:

```
f → x10, g → x11, h → x12, i → x13, j → x14
```



```
if (i ≠ j) {
   f = g + h;
} else {
   f = g - h;
}
```

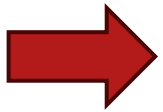
```
beq x13,x14,Else


Else:
```

Example: `if-else`

Assuming the below translations, **compile** *if-else* block:

```
f → x10, g → x11, h → x12, i → x13, j → x14
```

```
if (i ≠ j) {
   f = g + h;
} else {
   f = g - h;
}
```



```
   bne x13,x14,Else


Else:
   sub x10, x11, x12
```

Example: `if-else`

Assuming the below translations, **compile** *if-else* block:

```
f → x10, g → x11, h → x12, i → x13, j → x14
```

```
if (i ≠ j) {
  f = g + h;
} else {
  f = g - h;
}
```

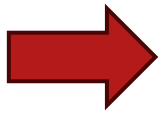
```
  bne x13,x14,Else
  add x10,x11,x12

Else:
  sub x10, x11, x12
```

Example: `if-else`

Assuming the below translations, **compile** *if-else* block:

```
f → x10, g → x11, h → x12, i → x13, j → x14
```

```
if (i ≠ j) {
   f = g + h;
} else {
   f = g - h;
}
```



```
   bne x13,x14,Else
   add x10,x11,x12
   j Exit
Else:
   sub x10, x11, x12
Exit:
```

Labels in Machine Code

```
   bne x13,x14,Else
   add x10,x11,x12
   j Exit
Else:
   sub x10, x11, x12
Exit:
```

```
0:    bne x13,x14,12
4:    add x10,x11,x12
8:    j 8
12:   sub x10, x11, x12
16:
```

*Labels are replaced with **offsets** in the assembly code.*

Labels in Machine Code

```
 bne x13,x14,Else
  add x10,x11,x12
  j Exit
Else:
  sub x10, x11, x12
Exit:
```

Loops in C & Assembly

Types of Loops

There are **three** types of loops in C:

For

```c
for (int i = 0; i < 10; i++) {
  printf("%d\n", i);
}
```

While

```c
int i = 0;
while (i < 10) {
    printf("%d\n", i);
    i++;
}
```

Do-While

```c
int i = 0;
do {
    printf("%d\n", i);
    i++;
} while (i < 10);
```

Example: Compile `while` loop

```
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
add  x9,  x8, x0   # x9  = &A[0]
```

Example: Compile `while` loop

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
add  x9,  x8, x0    # x9  = &A[0]
add  x10, x0, x0    # sum = 0
```

Example: Compile `while` loop

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
add  x9,  x8, x0    # x9  = &A[0]
add  x10, x0, x0    # sum = 0
add  x11, x0, x0    # i   = 0
```

Example: Compile `while` loop

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
add  x9,  x8, x0    # x9  = &A[0]
add  x10, x0, x0    # sum = 0
add  x11, x0, x0    # i   = 0
addi x13, x0, 20    # x13 = 20
```

Example: Compile `while` loop

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
```

Example: Compile `while` loop

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
```

Example: Compile `while` loop

```
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
    lw   x12, 0(x9)     # x12  = A[i]
```

Example: Compile `while` loop

```
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
    lw   x12, 0(x9)     # x12  = A[i]
    add  x10, x10, x12  # sum += x12
```

Example: Compile `while` loop

```
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
    lw   x12, 0(x9)     # x12  = A[i]
    add  x10, x10, x12  # sum += x12
    addi x9,  x9,  4    # &A[i+1]
```

Example: Compile `while` loop

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
    lw   x12, 0(x9)     # x12  = A[i]
    add  x10, x10, x12  # sum += x12
    addi x9,  x9,  4    # &A[i+1]
    addi x11, x11, 1    # i++
```

Example: Compile `while` loop

```
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
    lw   x12, 0(x9)     # x12  = A[i]
    add  x10, x10, x12  # sum += x12
    addi x9,  x9,  4    # &A[i+1]
    addi x11, x11, 1    # i++
    j Loop
```

Example: Compile `while` loop

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
    lw   x12, 0(x9)     # x12  = A[i]
    add  x10, x10, x12  # sum += x12
    addi x9,  x9,  4    # &A[i+1]
    addi x11, x11, 1    # i++
    j Loop
Done:
```