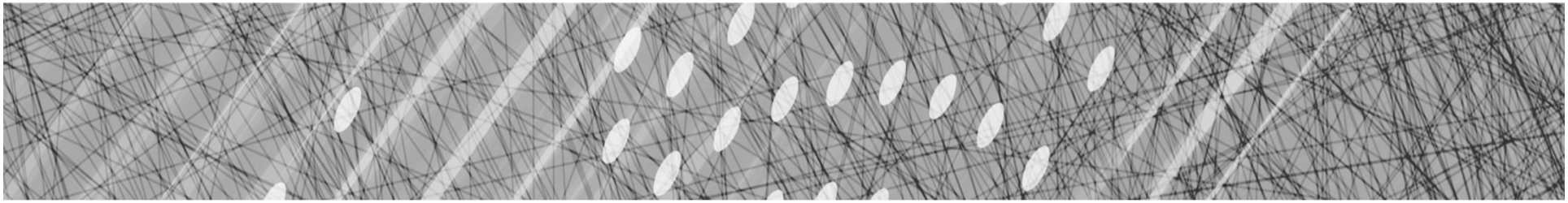


# The RISC-V Processor

**Hakim Weatherspoon**
**CS 3410**
Computer Science
Cornell University


Cornell CIS
COMPUTING AND INFORMATION SCIENCE

[Weatherspoon, Bala, Bracy, and Sirer]

# Announcements

Check online syllabus/schedule

- http://www.cs.cornell.edu/Courses/CS3410/2019sp/schedule
- Slides and Reading for lectures
- Office Hours
- ***Pictures of all TAs***

- **Dates to keep in Mind**
  - **Prelims: Tue Mar 5th and Thur May 2nd**
  - ***Proj 1: Due next Friday, Feb 15th***
  - Proj3: Due before Spring break
  - Final Project: Due when final will be Feb 16th

Schedule is subject to change

# Collaboration, Late, Re-grading Policies

•"White Board" Collaboration Policy

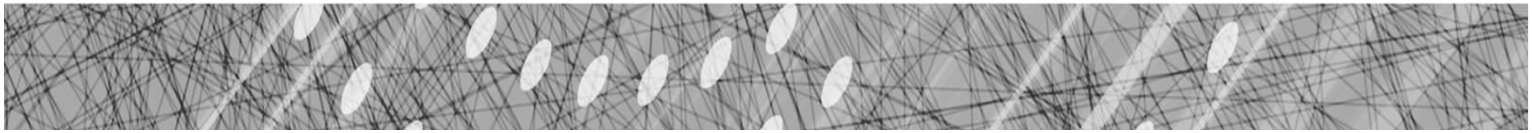
- Can discuss approach together on a "white board"
- Leave, watch a movie such as Stranger Things, then write up solution independently
- Do not copy solutions

## Late Policy

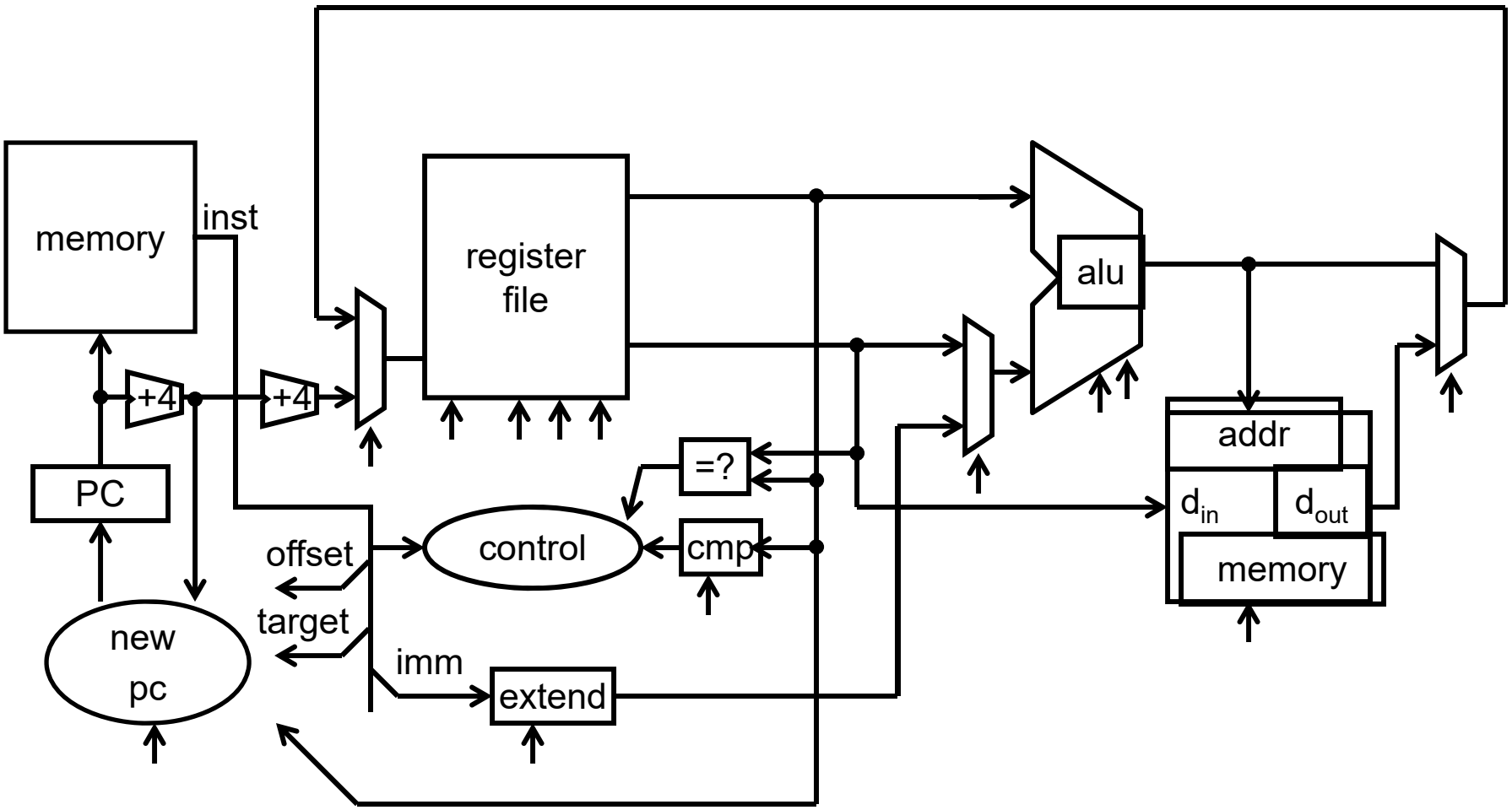
- Each person has a total of ***four*** "slip days"
- Max of ***two*** slip days for any individual assignment
- Slip days deducted first for *any* late assignment, cannot selectively apply slip days
- For projects, slip days are deducted from all partners
- ***25%*** deducted per day late after slip days are exhausted

## Regrade policy

- Submit written request within a week of receiving score



# Big Picture: Building a Processor




A single cycle processor

# Goal for the next 2 lectures

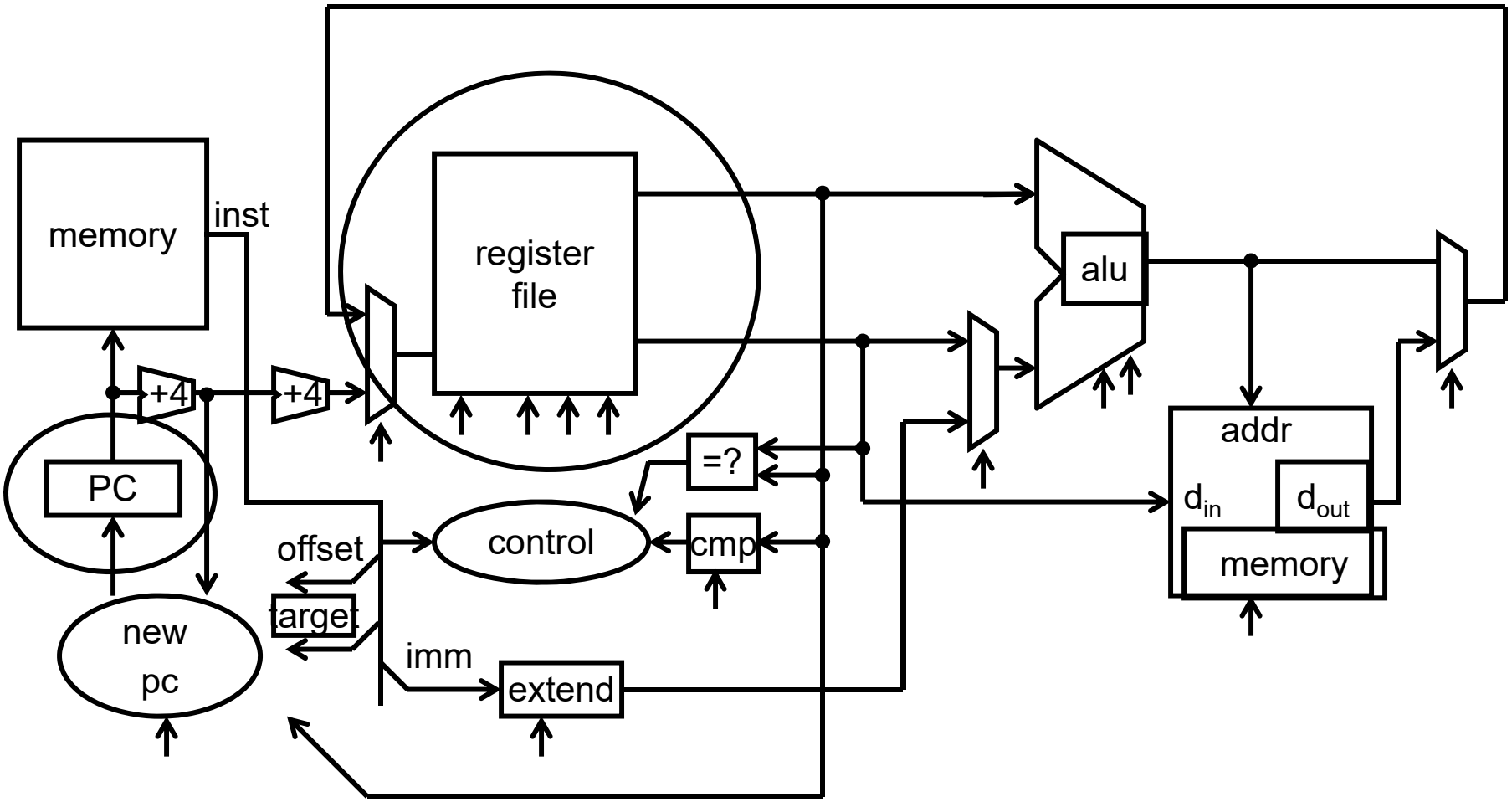
- Understanding the basics of a processor
  - We now have the technology to build a CPU!

- Putting it all together:
  - Arithmetic Logic Unit (ALU)
  - Register File
  - Memory
    - SRAM: cache
    - DRAM: main memory
  - RISC-V Instructions & how they are executed



5

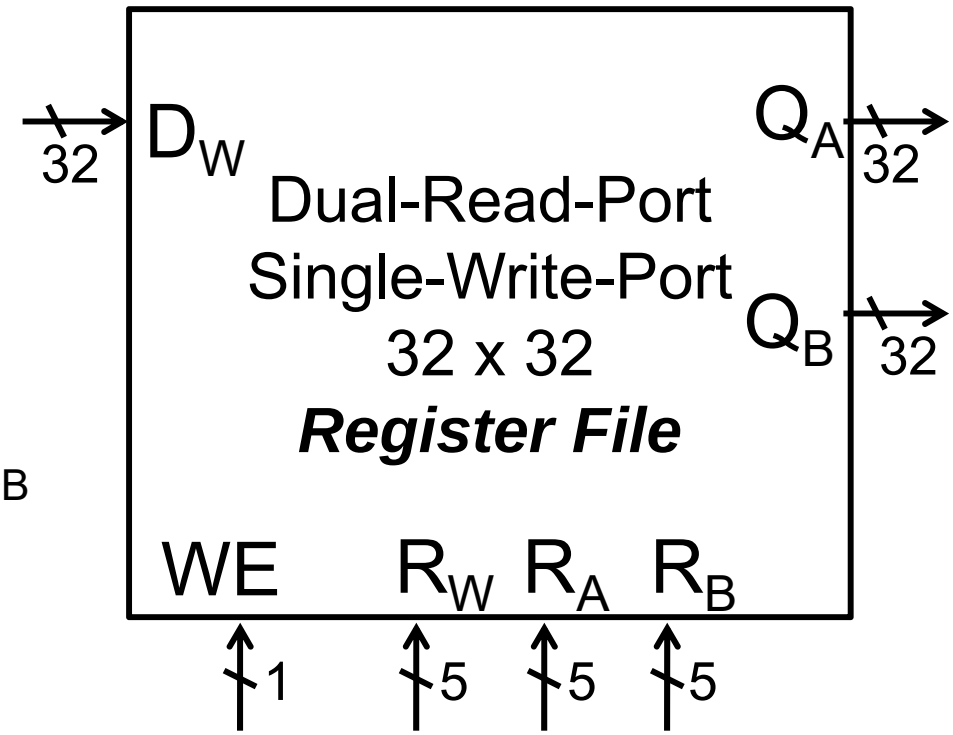
# **RISC-V** Register File




A single cycle processor

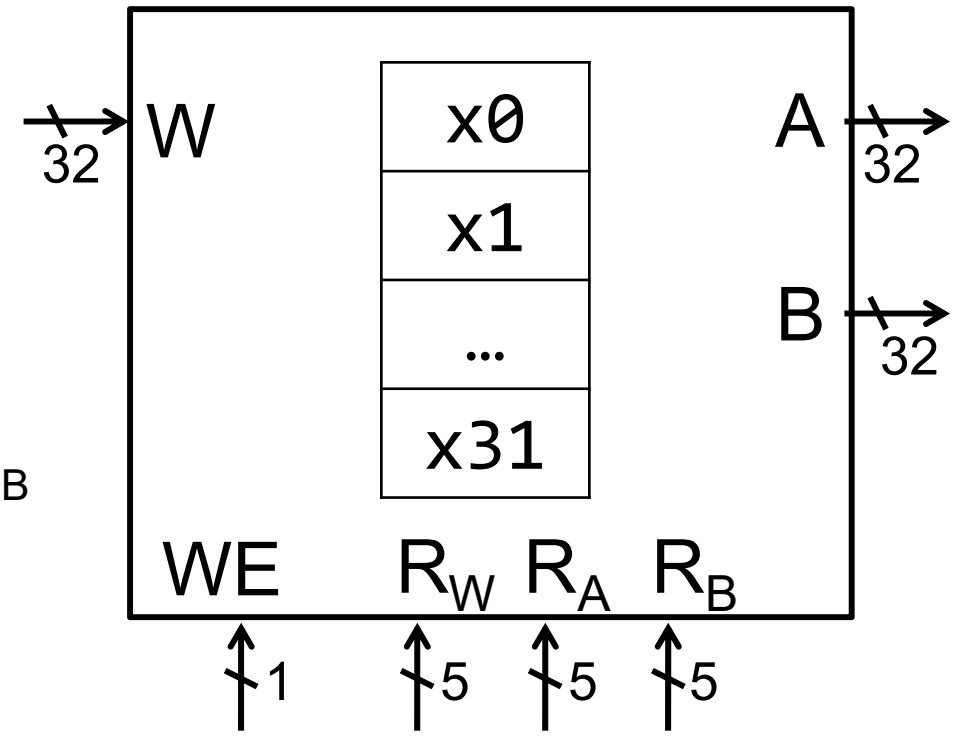
# **RISC-V** Register File

- RISC-V register file
  - 32 registers, 32-bits each
  - x0 wired to zero
  - Write port indexed via $R_W$
    - on falling edge when WE=1
  - Read ports indexed via $R_A$, $R_B$



# **RISC-V** Register File

- RISC-V register file
  - 32 registers, 32-bits each
  - x0 wired to zero
  - Write port indexed via $R_W$
    - on falling edge when WE=1
  - Read ports indexed via $R_A$, $R_B$

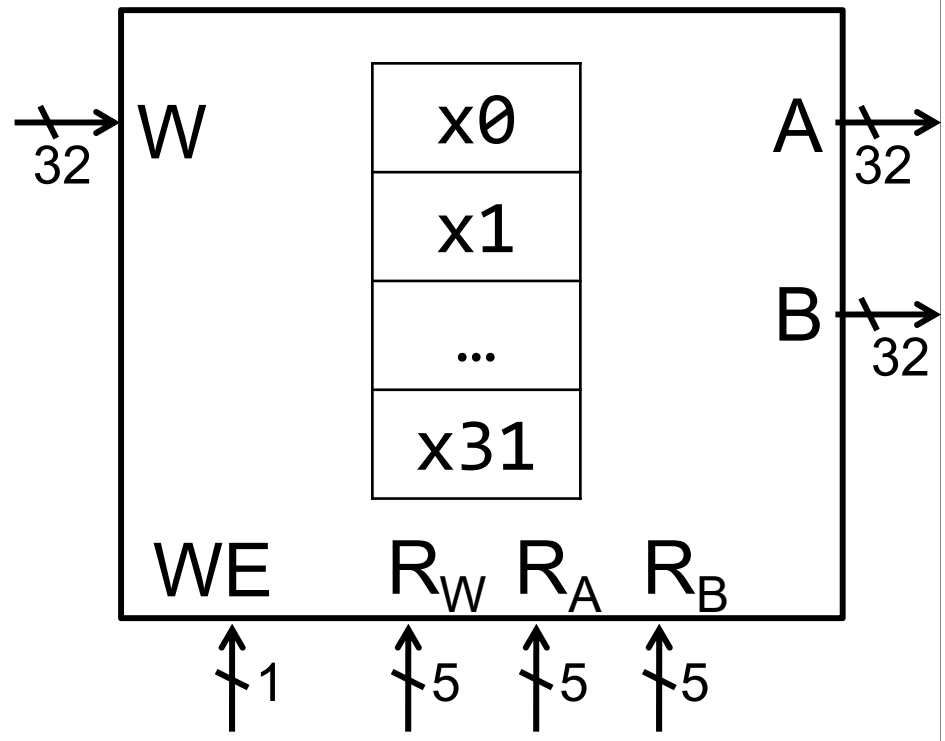



- RISC-V register file
  - Numbered from 0 to 31
  - Can be referred by number: x0, x1, x2, … x31
  - Convention, each register also has a name:
    - x10 – x17 → a0 – a7, x28 – x31 → t3 – t6

# iClicker Question

If we wanted to support 64 registers, what would change?

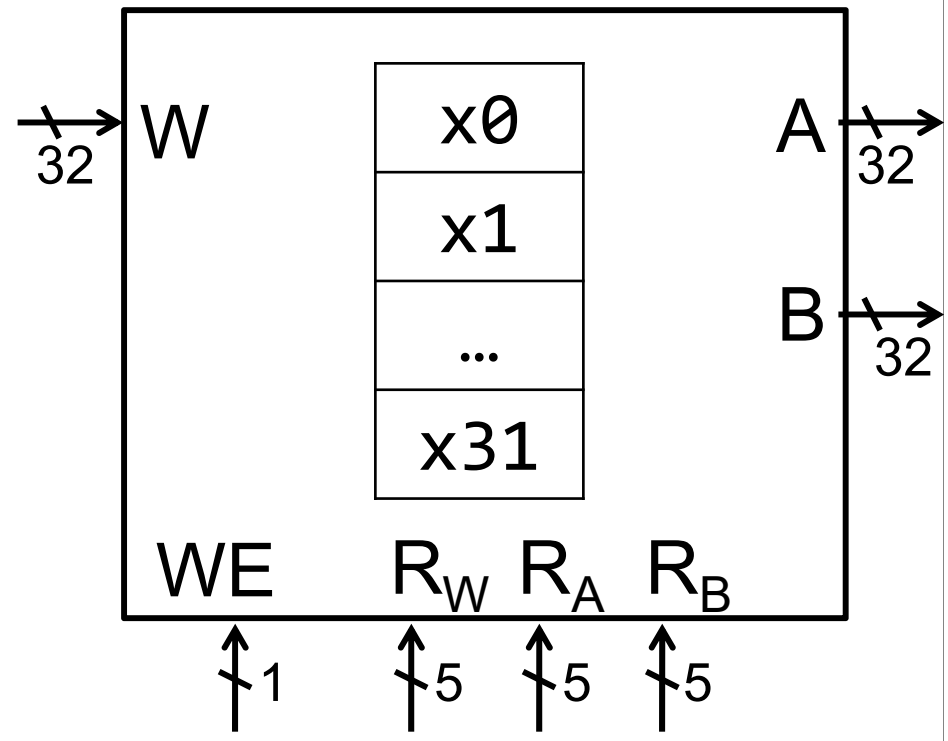
a) W, A, B → 64
b) $R_W$, $R_A$, $R_B$ 5 → 6
c) W 32 → 64, $R_W$ 5 → 6
d) A & B only



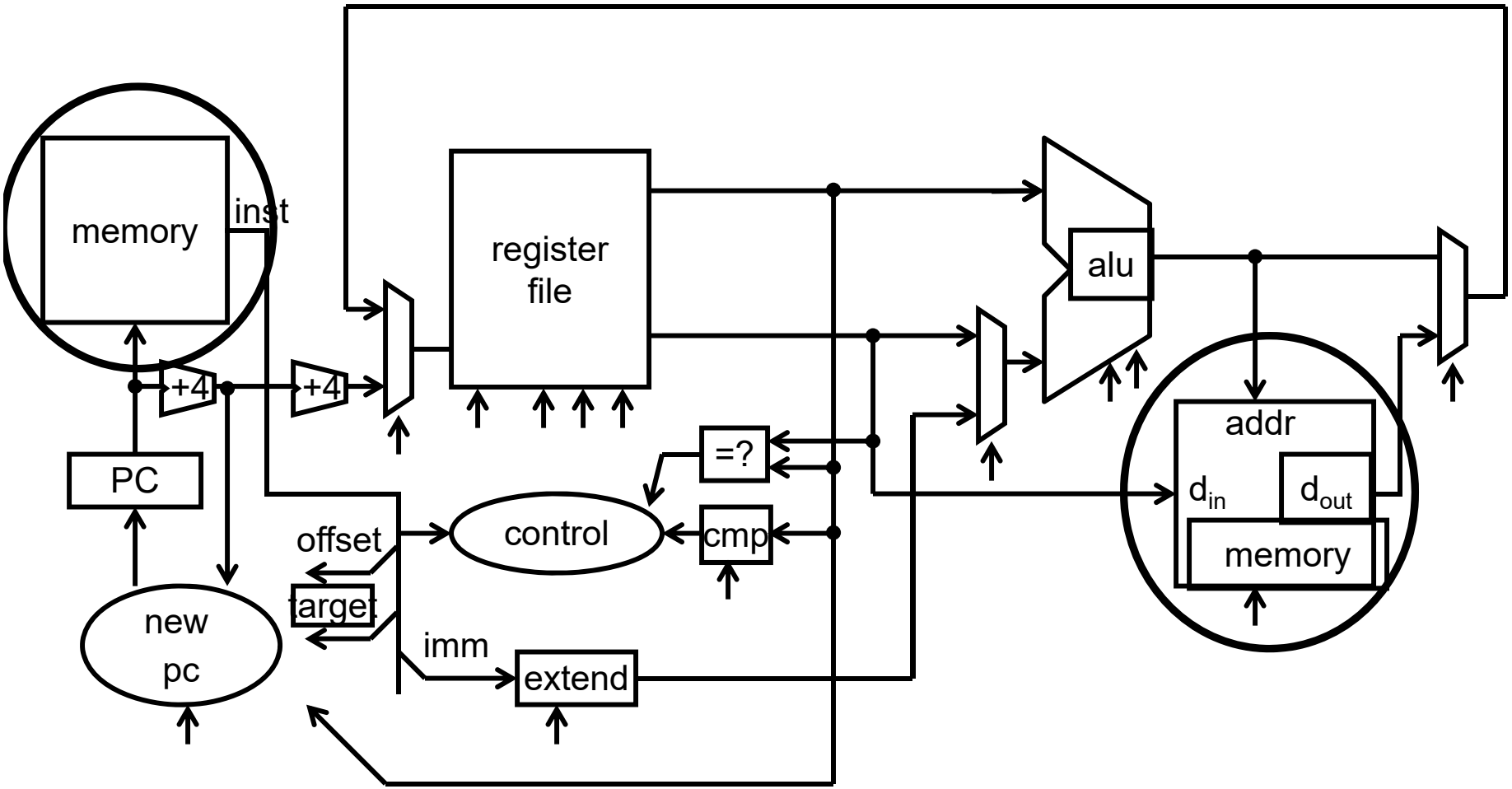
# iClicker Question

If we wanted to support 64 registers, what would change?

a) ~~W, A, B → 64~~
b) $R_W$, $R_A$, $R_B$ 5 → 6
c) ~~W 32 → 64~~, $R_W$ 5 → 6
d) A & B only

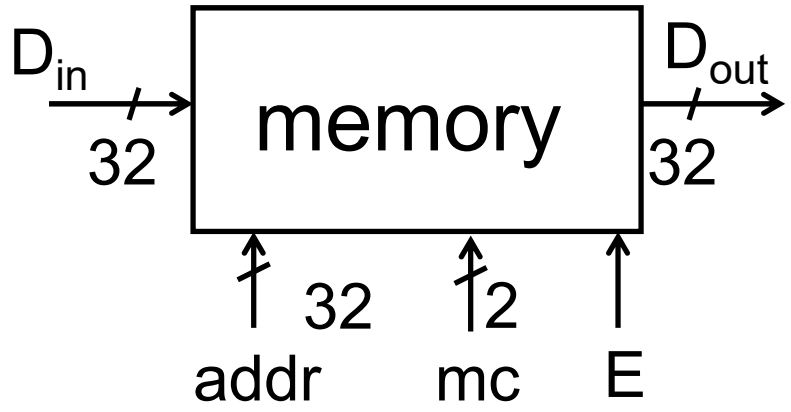


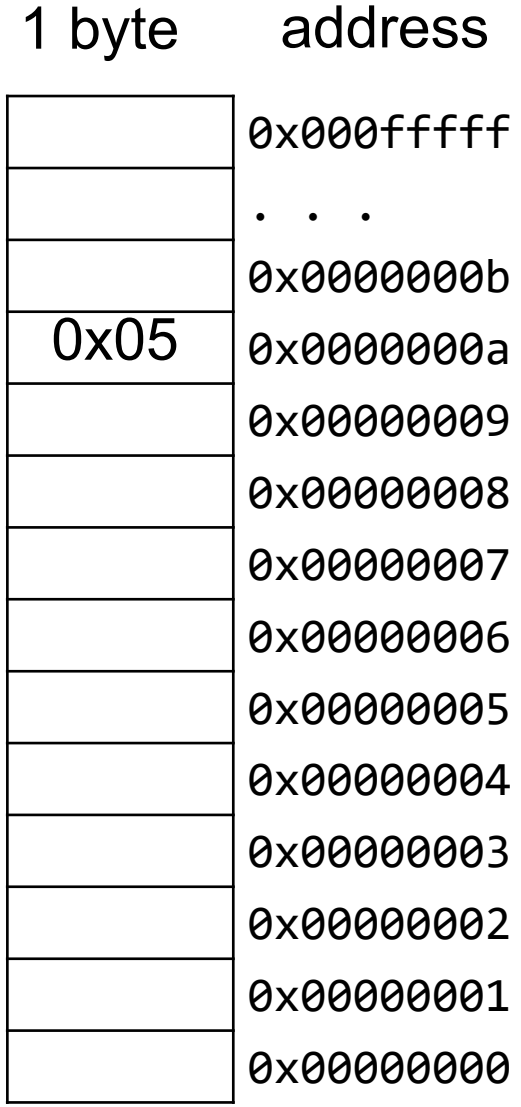
# **RISC-V** Memory




A single cycle processor

# **RISC-V** Memory

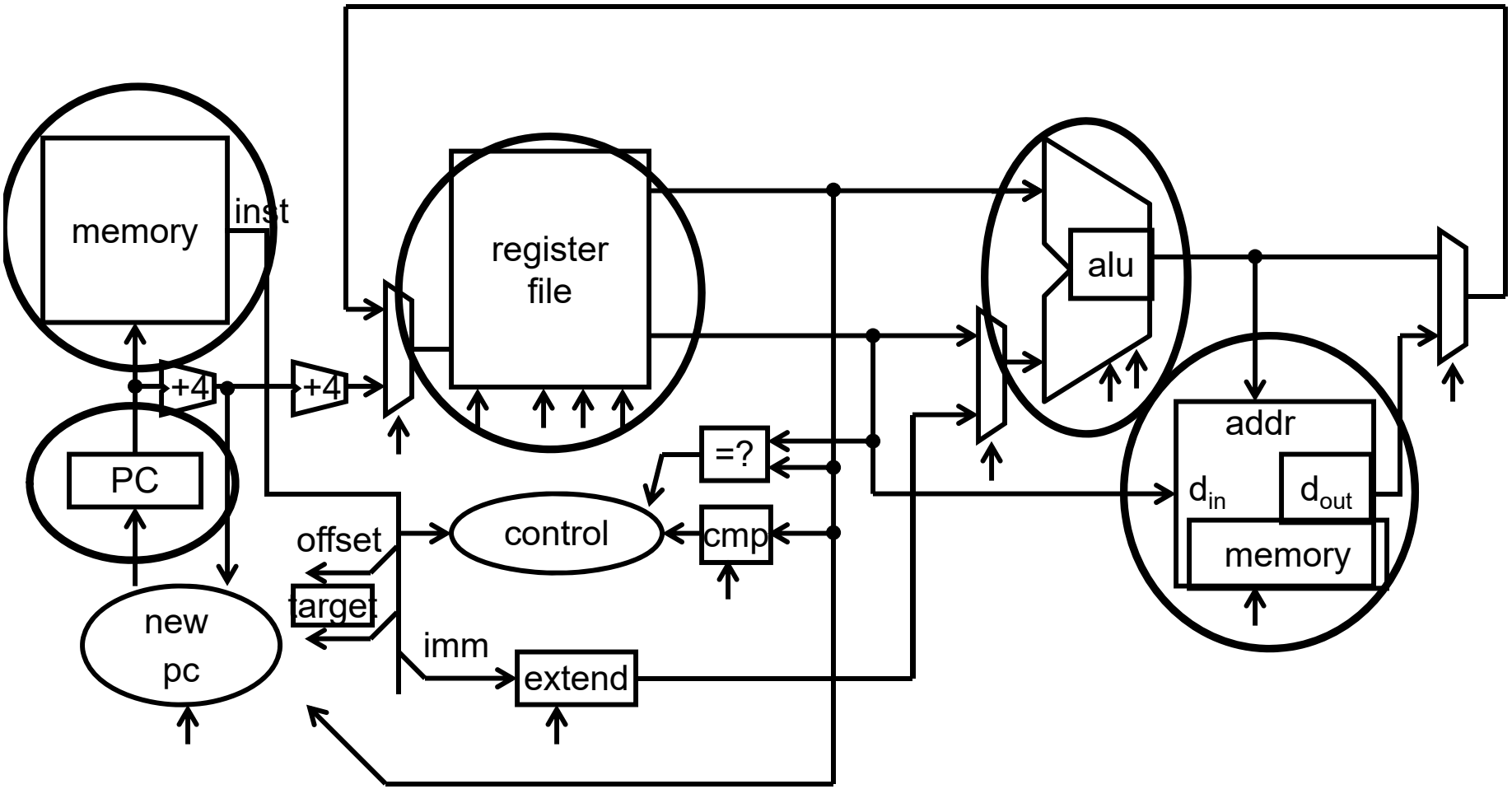



- 32-bit address
- 32-bit data (but byte addressed)
- Enable + 2 bit memory control (mc)

00: read word (4 byte aligned)
01: write byte
10: write halfword (2 byte aligned)
11: write word (4 byte aligned)

# Putting it all together: Basic Processor




A single cycle processor

# To make a computer

Need a program

- Stored program computer

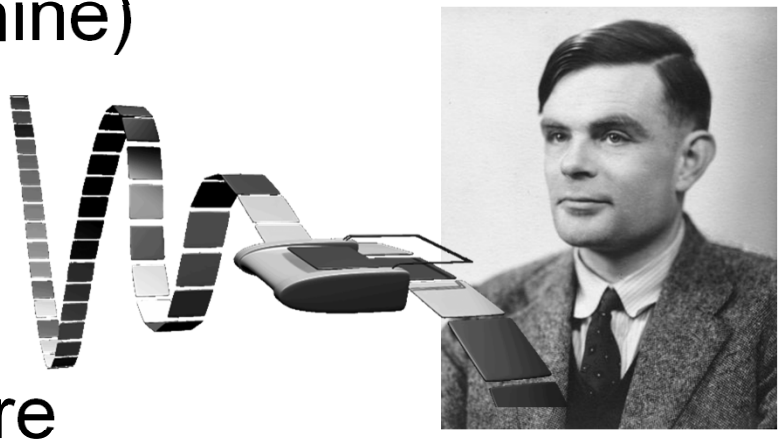
Architectures

- von Neumann architecture
- Harvard (modified) architecture

# To make a computer

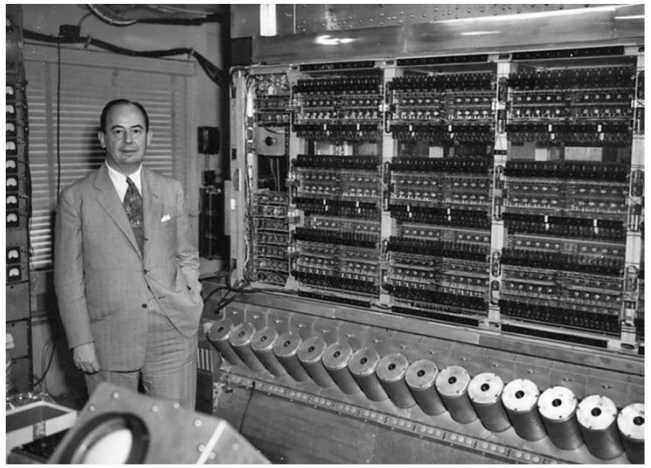
Need a program

- Stored program computer
- (a Universal Turing Machine)



Architectures

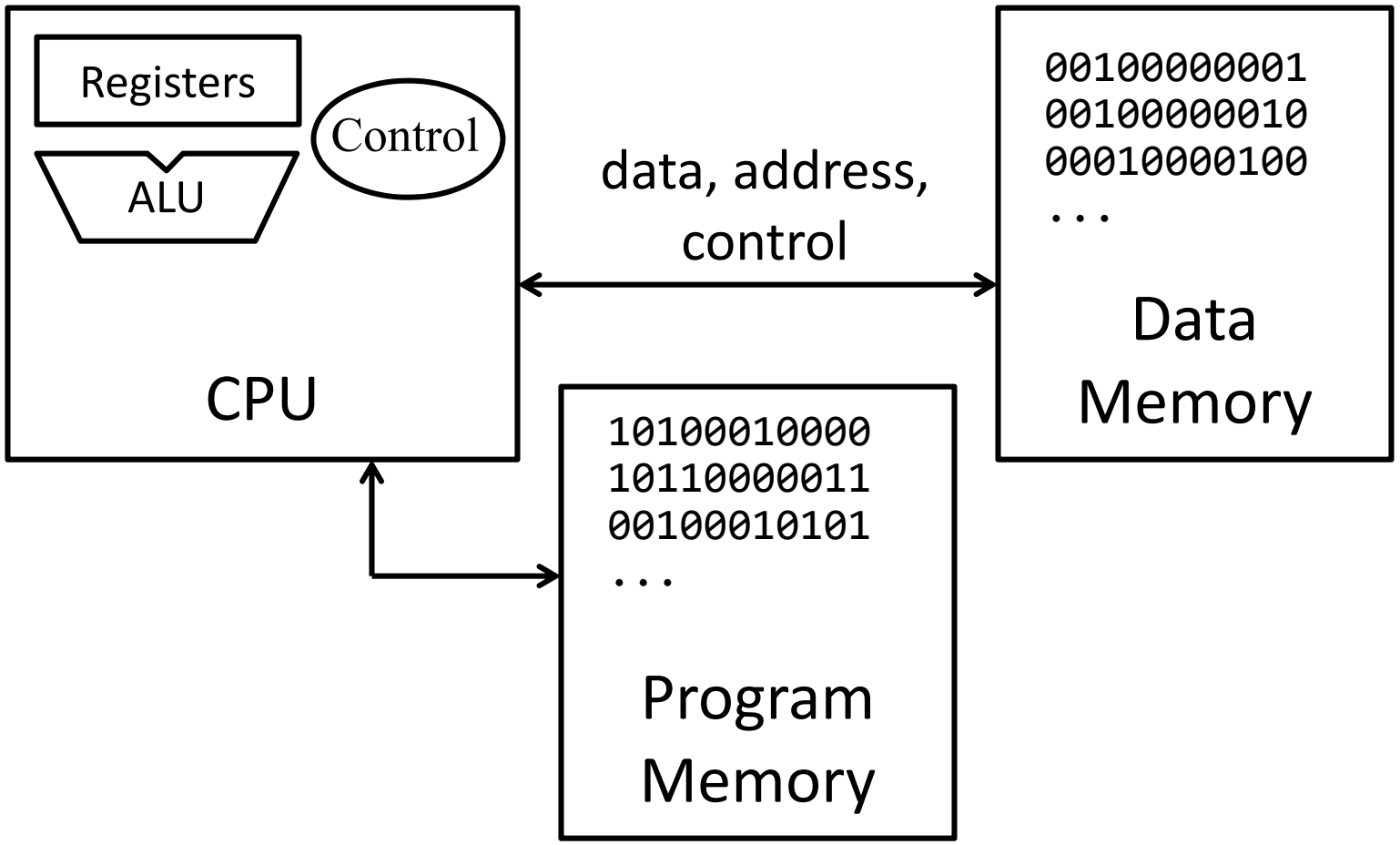
- von Neumann architecture
- Harvard (modified) architecture



# Putting it all together: Basic Processor

## A RISC-V CPU with a (modified) Harvard architecture

- Modified: instructions & data in common address space, separate instr/data caches can be accessed in parallel



# Takeaway

A processor executes instructions

- Processor has some internal state in storage elements (registers)

A memory holds instructions and data

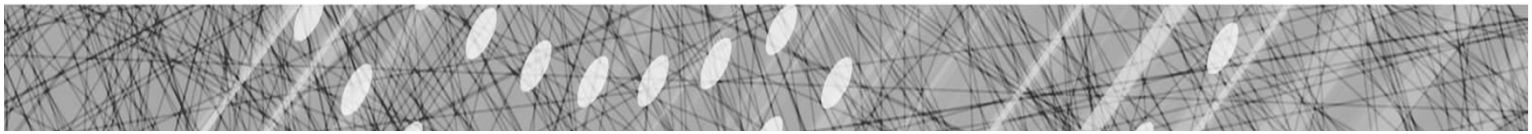
- (modified) Harvard architecture: separate insts and data
- von Neumann architecture: combined inst and data

A bus connects the two

We now have enough building blocks to build machines that can perform non-trivial computational tasks

# Next Goal

- How to program and execute instructions on a RISC-V processor?



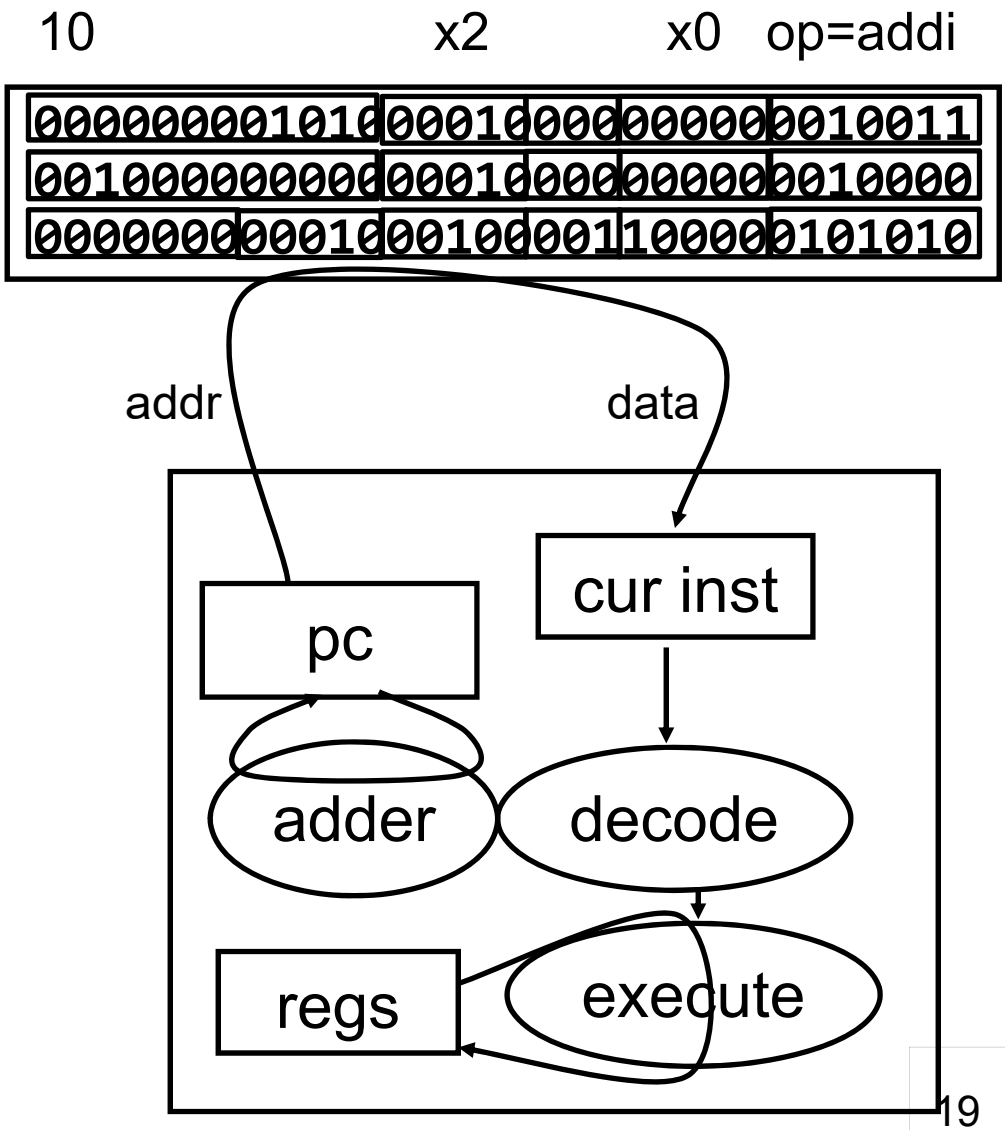
# Instruction Usage

Instructions are stored in memory, encoded in binary

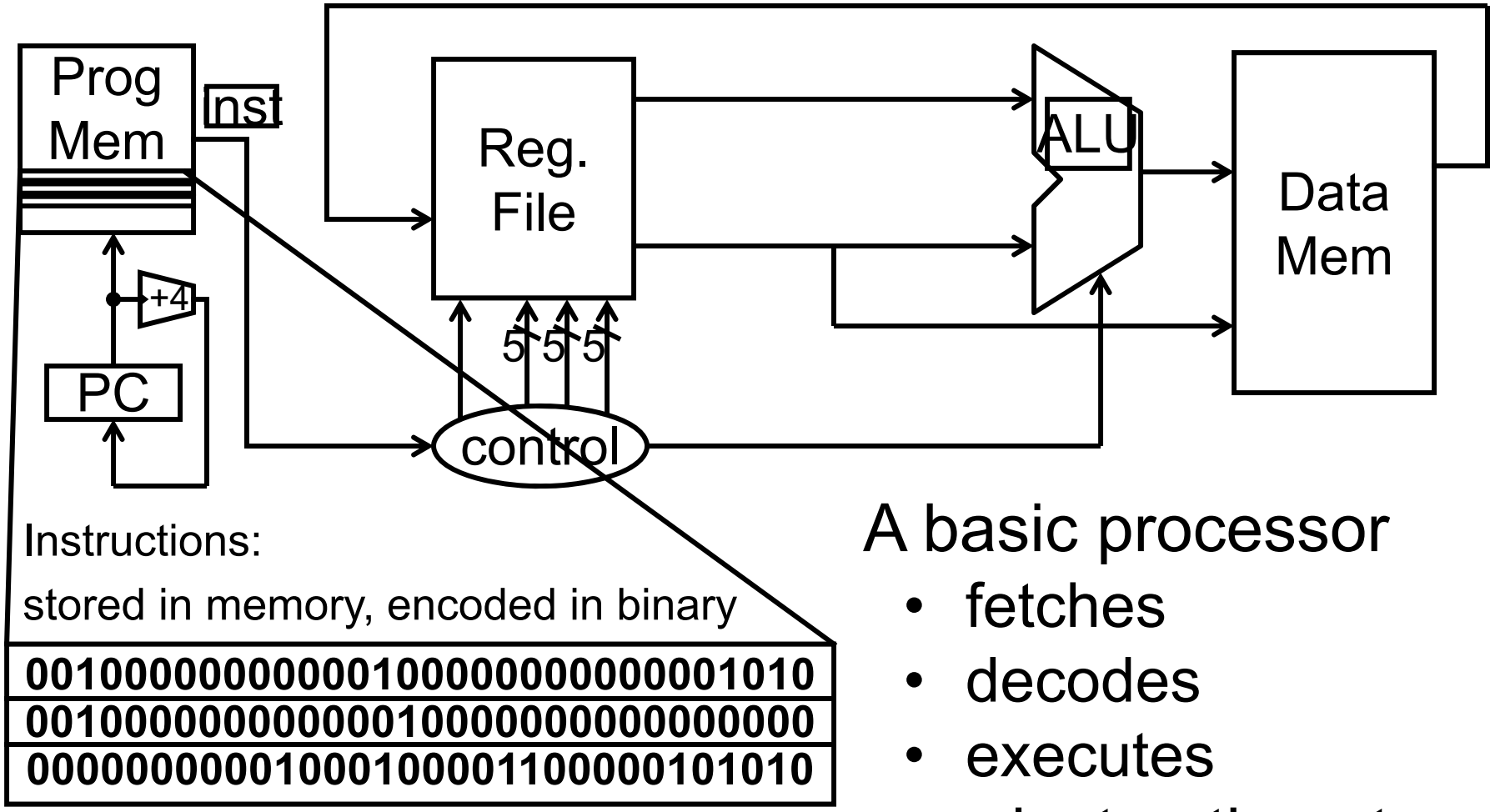
A basic processor

- fetches
- decodes
- executes

one instruction at a time



# Instruction Processing

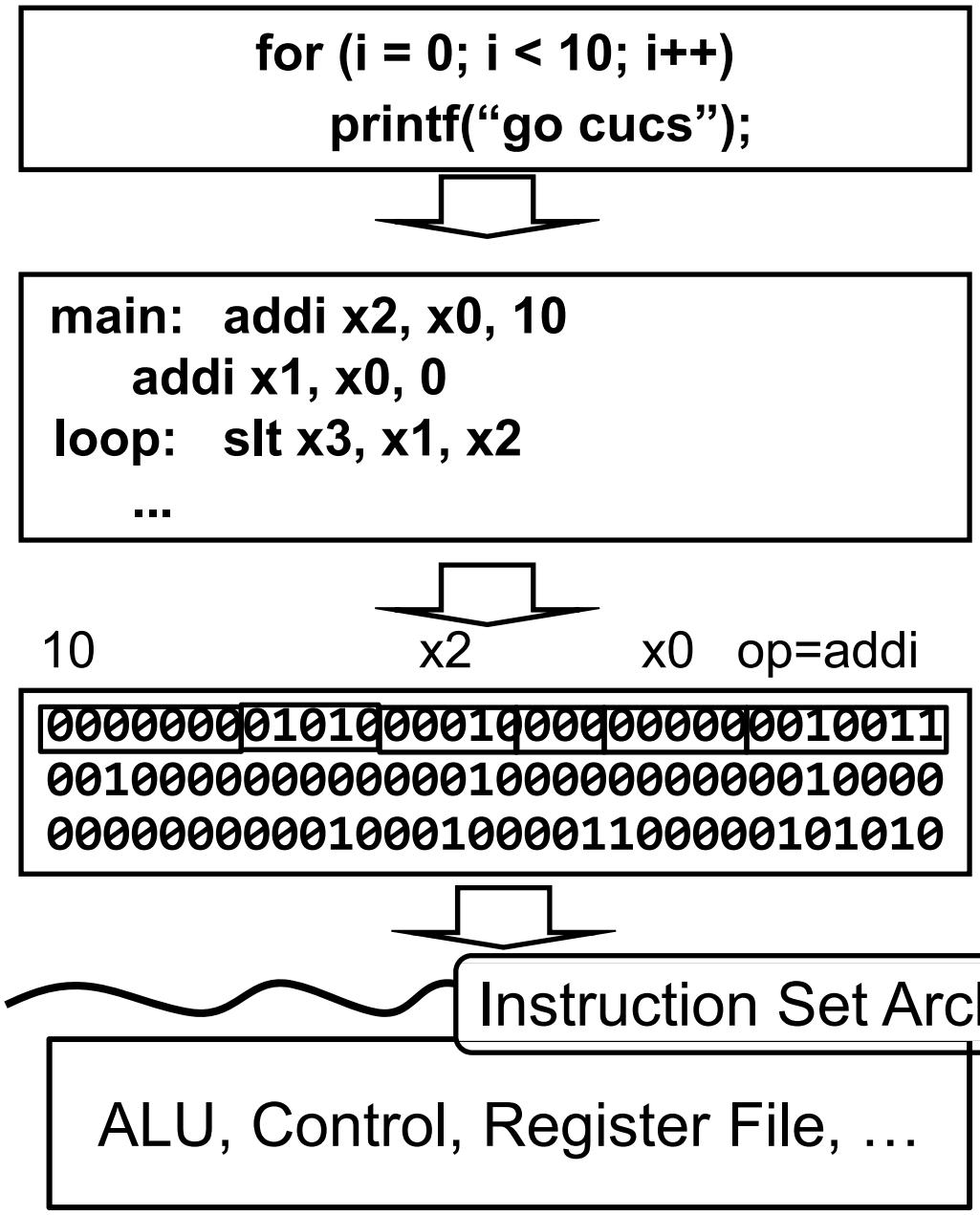



A basic processor

- fetches
- decodes
- executes

one instruction at a time

# Levels of Interpretation: Instructions




High Level Language

- C, Java, Python, ADA, …
- Loops, control flow, variables

Assembly Language

- No symbols (except labels)
- One operation per statement
- "human readable machine language"

Machine Language

- Binary-encoded assembly
- Labels become addresses
- **The language of the CPU**

Machine Implementation (Microarchitecture)

# Instruction Set Architecture (ISA)

Different CPU architectures specify different instructions

Two classes of ISAs

- Reduced Instruction Set Computers (RISC)
  IBM Power PC, Sun Sparc, MIPS, Alpha
- Complex Instruction Set Computers (CISC)
  Intel x86, PDP-11, VAX

Another ISA classification: Load/Store Architecture

- Data must be in registers to be operated on
  For example: array[x] = array[y] + array[z]
  1 add ? OR 2 loads, an add, and a store ?
- Keeps HW simple → many RISC ISAs are load/store

# iClicker Question

What does it mean for an architecture to be called a load/store architecture?

(A) Load and Store instructions are supported by the ISA.
(B) Load and Store instructions can also perform arithmetic instructions on data in memory.
(C) Data must first be loaded into a register before it can be operated on.
(D) Every load must have an accompanying store at some later point in the program.

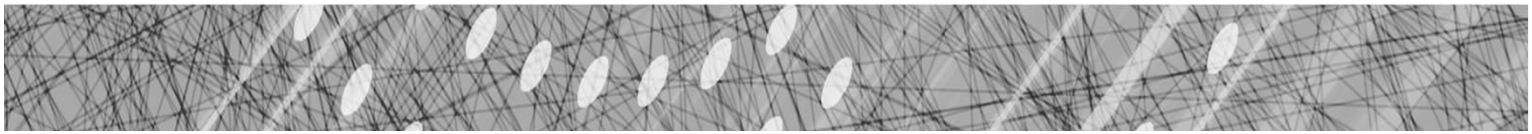
# iClicker Question

What does it mean for an architecture to be called a load/store architecture?

(A) Load and Store instructions are supported by the ISA.
(B) Load and Store instructions can also perform arithmetic instructions on data in memory.
(C) Data must first be loaded into a register before it can be operated on.
(D) Every load must have an accompanying store at some later point in the program.

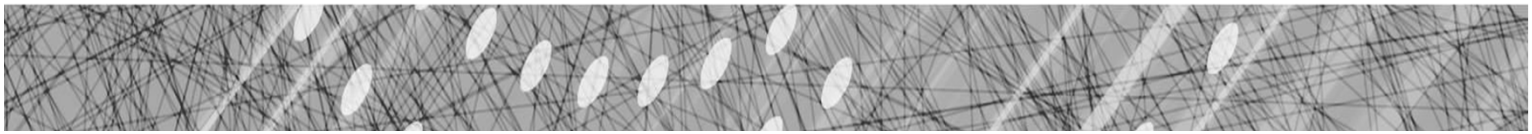
# Takeaway

A RISC-V processor and ISA (instruction set architecture) is an example a Reduced Instruction Set Computers (RISC) where simplicity is key, thus enabling us to build it!!

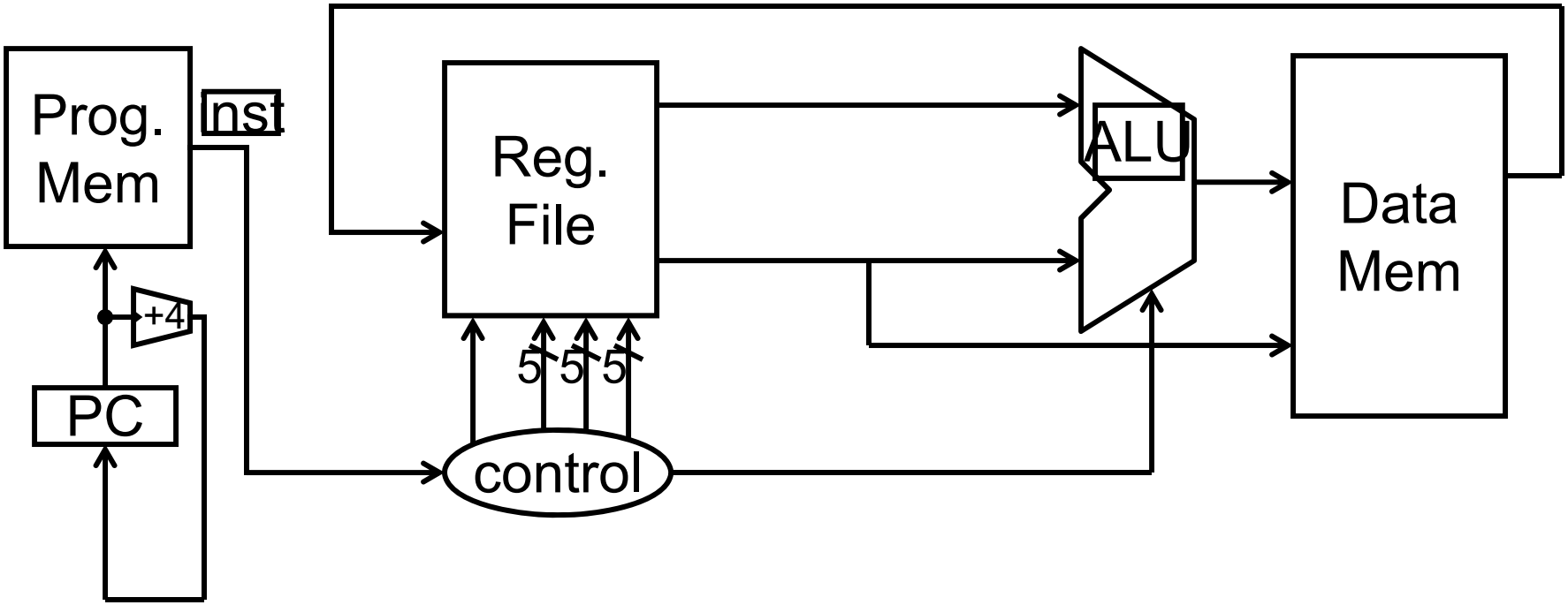


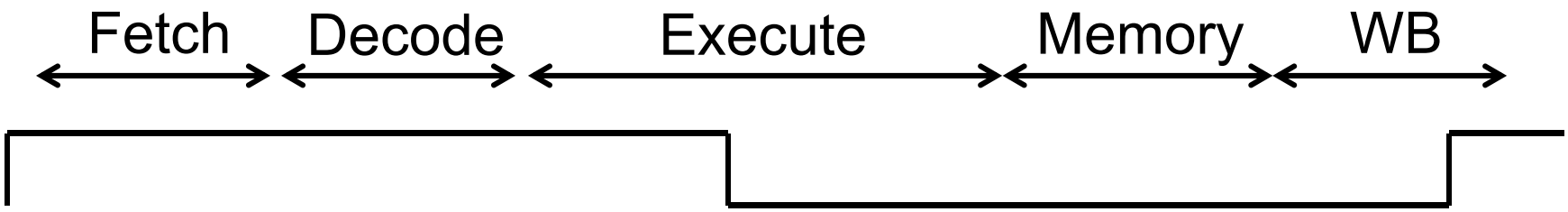
# Next Goal

How are instructions executed?
What is the general datapath to execute an instruction?



# Five Stages of RISC-V Datapath



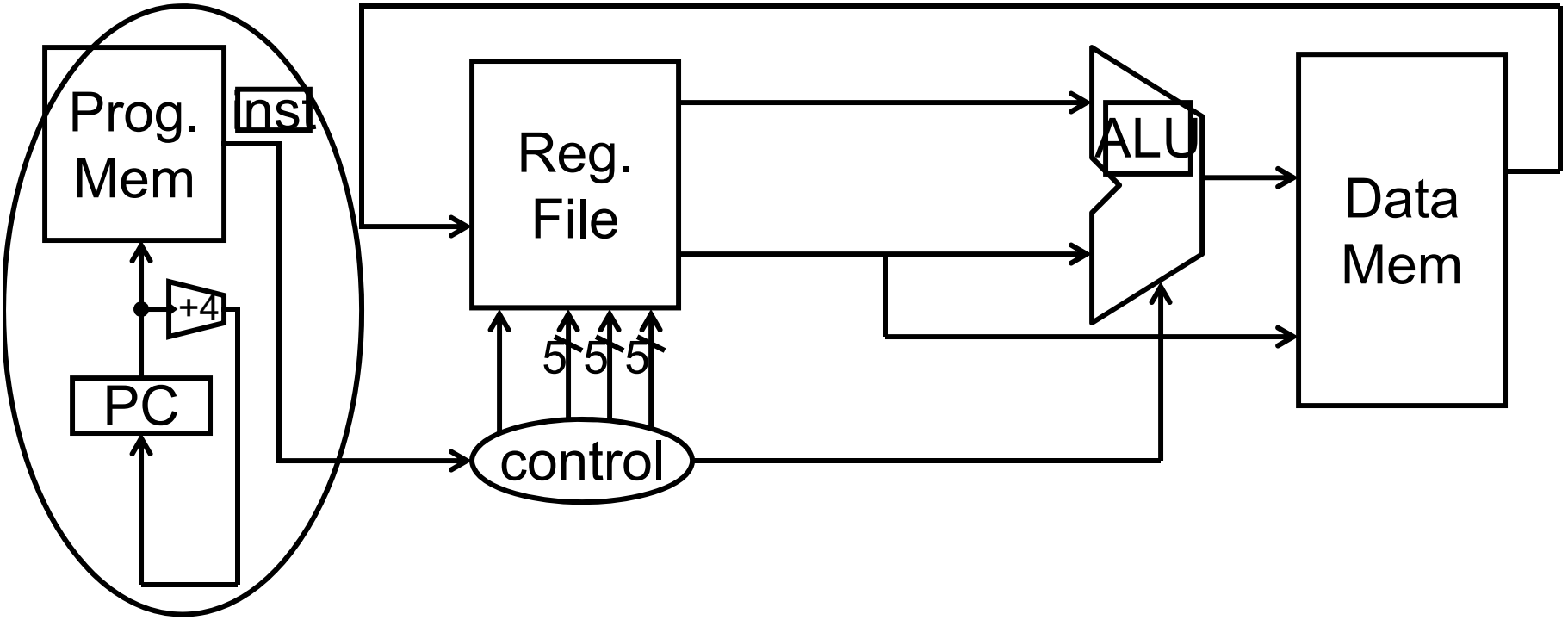

A single cycle processor – this diagram is not 100% spatial

# Five Stages of RISC-V Datapath

Basic CPU execution loop

1. Instruction Fetch
2. Instruction Decode
3. Execution (ALU)
4. Memory Access
5. Register Writeback

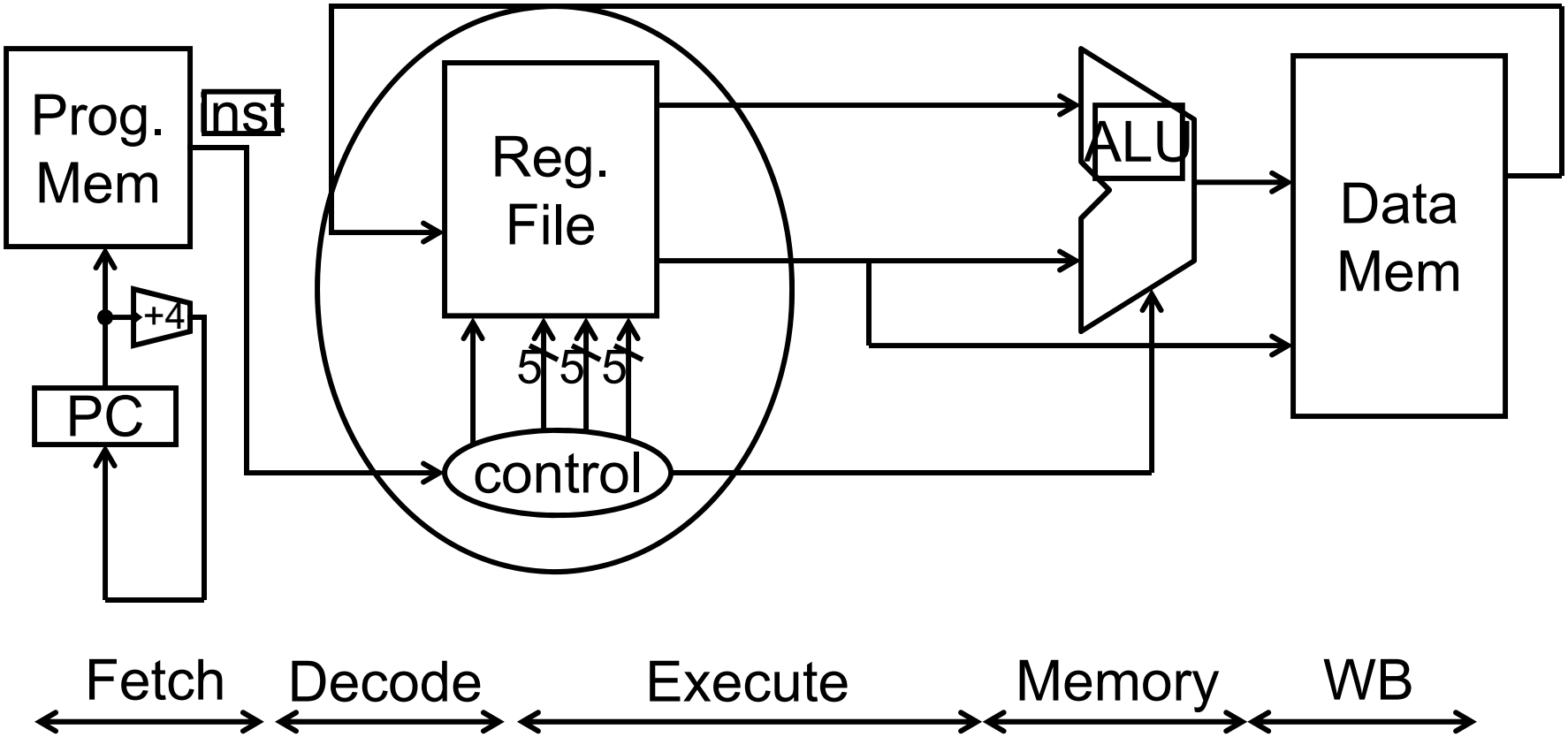
# Stage 1: Instruction Fetch





Fetch 32-bit instruction from memory
Increment PC = PC + 4

# Stage 2: Instruction Decode



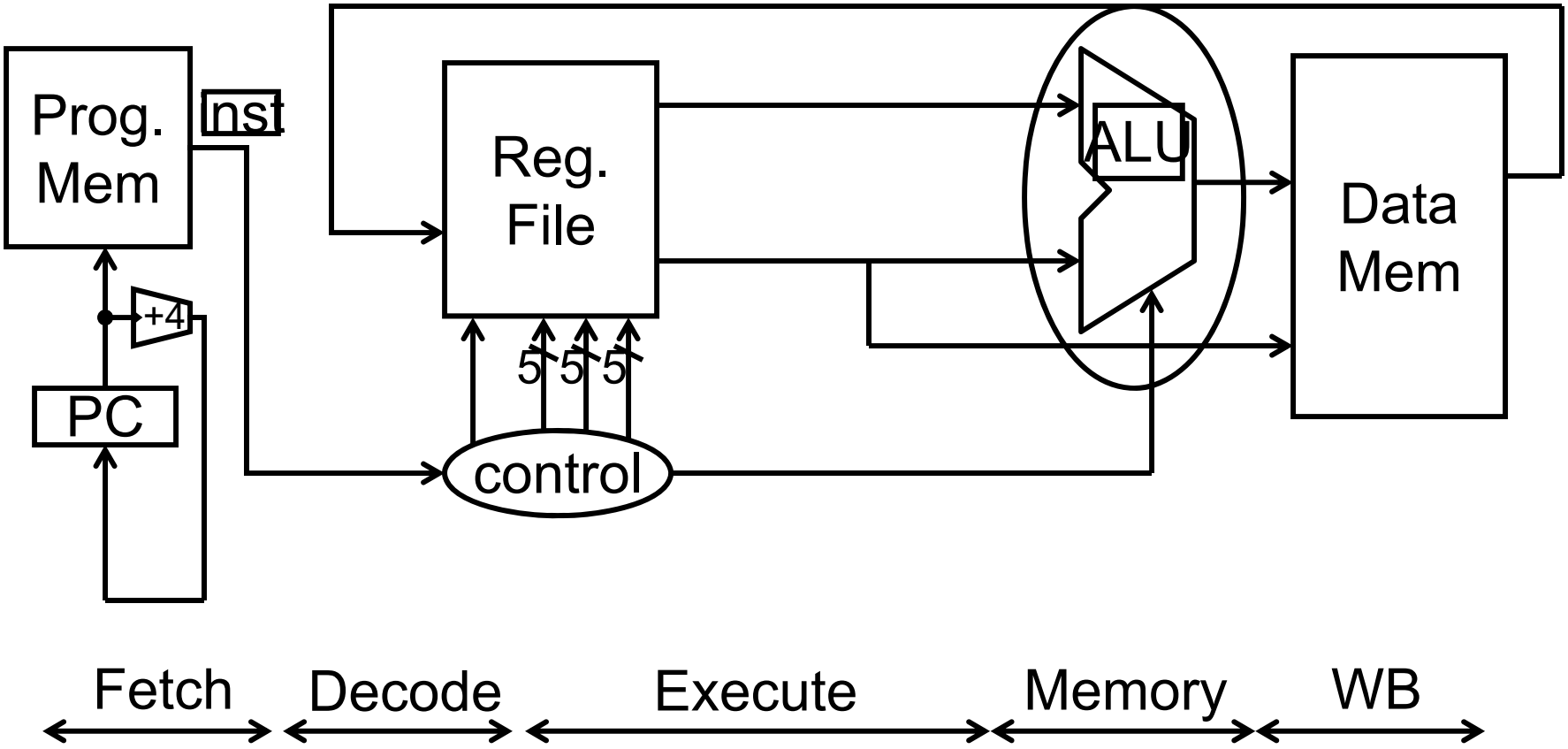

Gather data from the instruction

Read opcode; determine instruction type, field lengths

Read in data from register file

(0, 1, or 2 reads for `jump`, `addi`, or `add`, respectively)

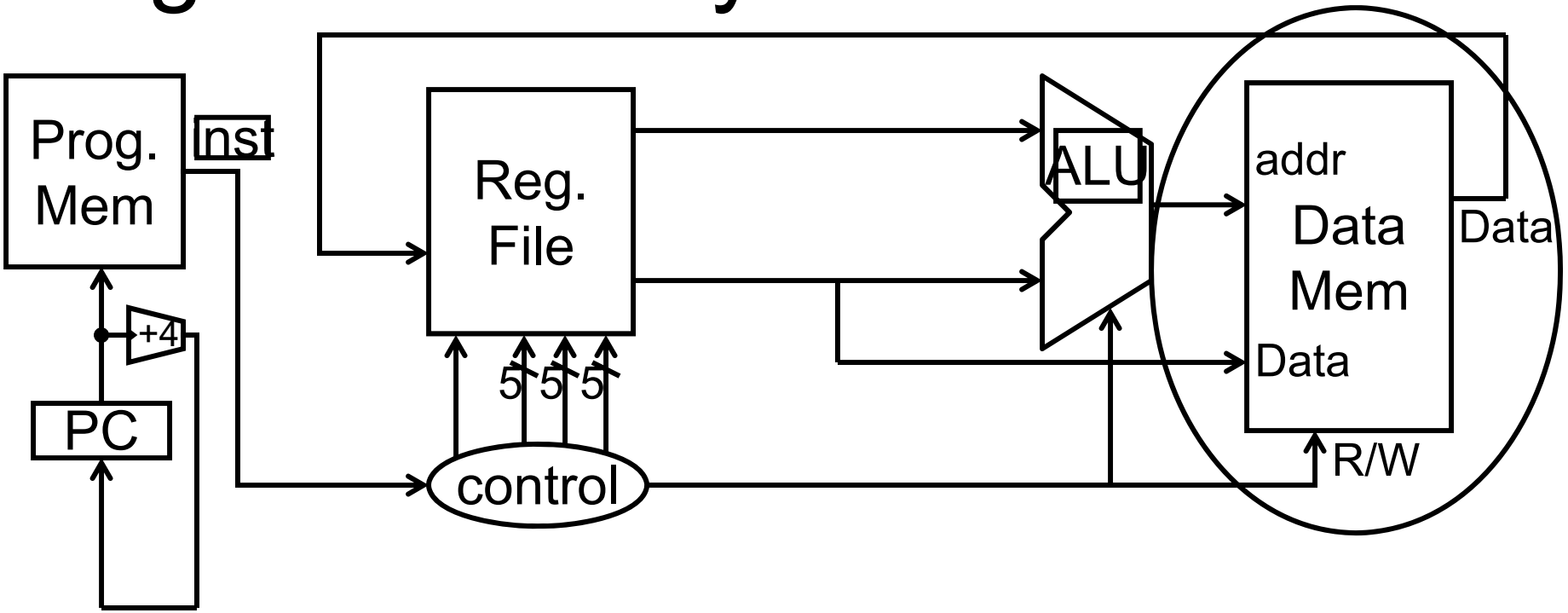
# Stage 3: Execution (ALU)




Useful work done here (+, -, *, /), shift, logic operation, comparison (slt)

Load/Store? lw x2, x3, 32 → Compute address

# Stage 4: Memory Access

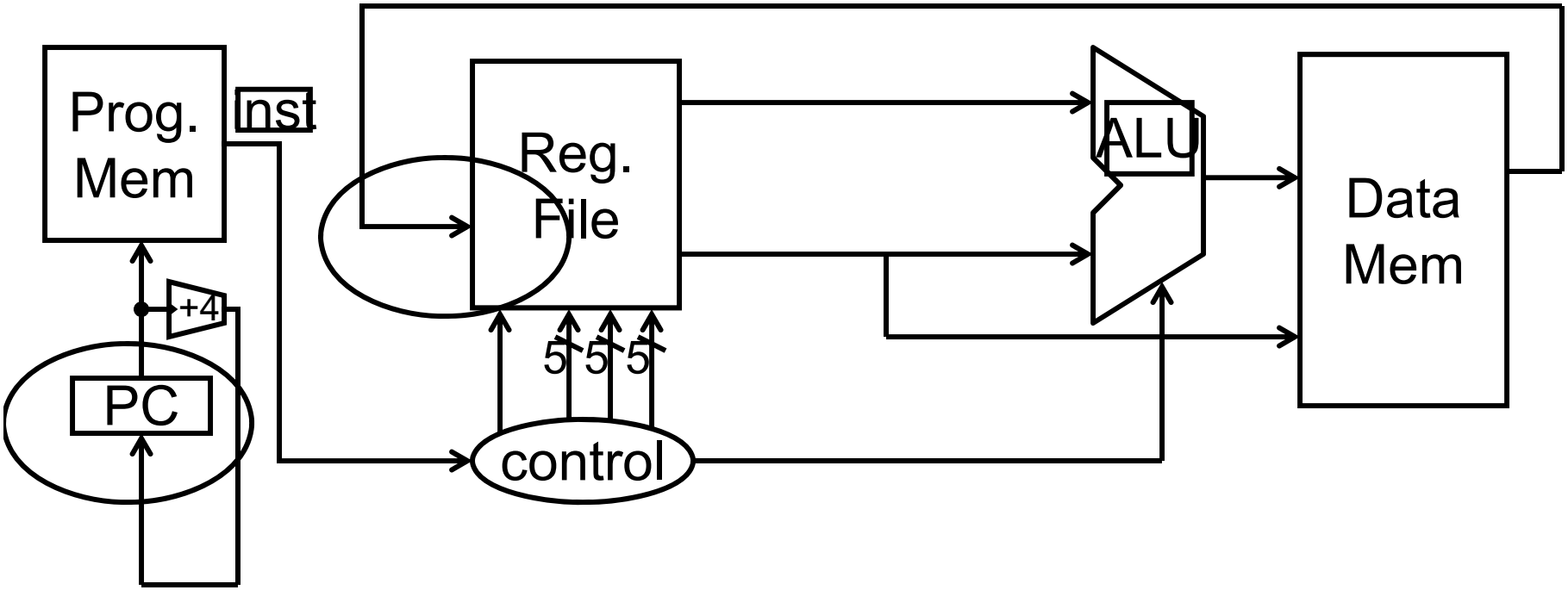




Used by load and store instructions only

Other instructions will skip this stage

# Stage 5: Writeback





Write to register file

- For arithmetic ops, logic, shift, etc, load. What about stores?

Update PC

- For branches, jumps

# iClicker Question

Which of the following statements is true?

(A) All instructions require an access to Program Memory.
(B) All instructions require an access to Data Memory.
(C) All instructions write to the register file.
(D) Some RISC-V instructions are shorter than 32 bits
(E) A & C

# iClicker Question

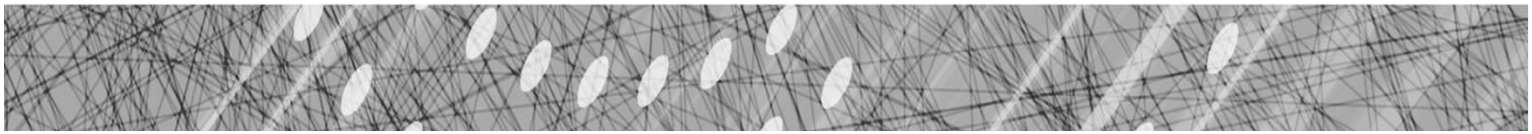
Which of the following statements is true?

(A) All instructions require an access to Program Memory.
(B) All instructions require an access to Data Memory.
(C) All instructions write to the register file.
(D) Some RISC-V instructions are shorter than 32 bits
(E) A & C

# Takeaway

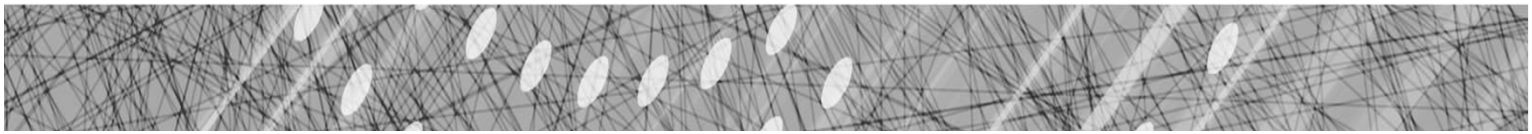
- The datapath for a RISC-V processor has five stages:
  1. Instruction Fetch
  2. Instruction Decode
  3. Execution (ALU)
  4. Memory Access
  5. Register Writeback

- This five stage datapath is used to execute all RISC-V instructions



# Next Goal

- Specific datapaths RISC-V Instructions



# RISC-V Design Principles

## Simplicity favors regularity

- 32 bit instructions

## Smaller is faster

- Small register file

## Make the common case fast

- Include support for constants

## Good design demands good compromises

- Support for different type of interpretations/classes

# Instruction Types

- Arithmetic
  - add, subtract, shift left, shift right, multiply, divide
- Memory
  - load value from memory to a register
  - store value to memory from a register
- Control flow
  - conditional jumps (branches)
  - jump and link (subroutine call)

- Many other instructions are possible
  - vector add/sub/mul/div, string operations
  - manipulate coprocessor
  - I/O

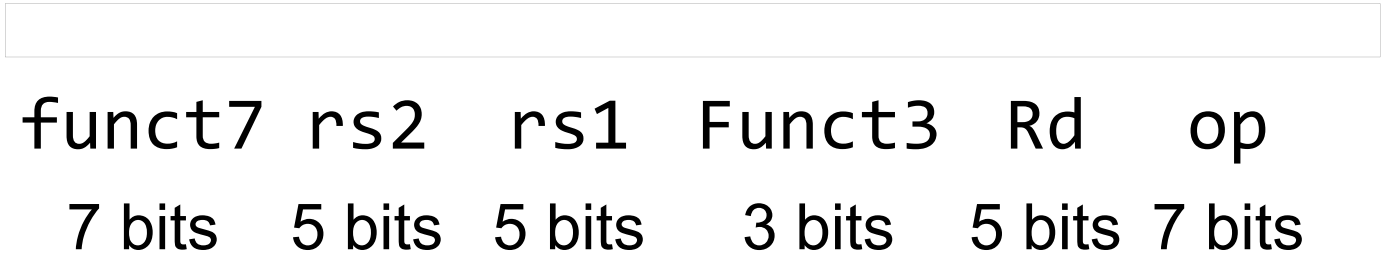
# RISC-V Instruction Types

- Arithmetic/Logical
  - R-type: result and two source registers, shift amount
  - I-type: result and source register, shift amount in 16-bit immediate with sign/zero extension
  - U-type: result register, 16-bit immediate with sign/zero extension
- Memory Access
  - I-type for loads and S-type for stores
  - load/store between registers and memory
  - word, half-word and byte operations
- Control flow
  - UJ-type: jump-and-link
  - I-type: jump-and-link register
  - SB-type: conditional branches: pc-relative addresses

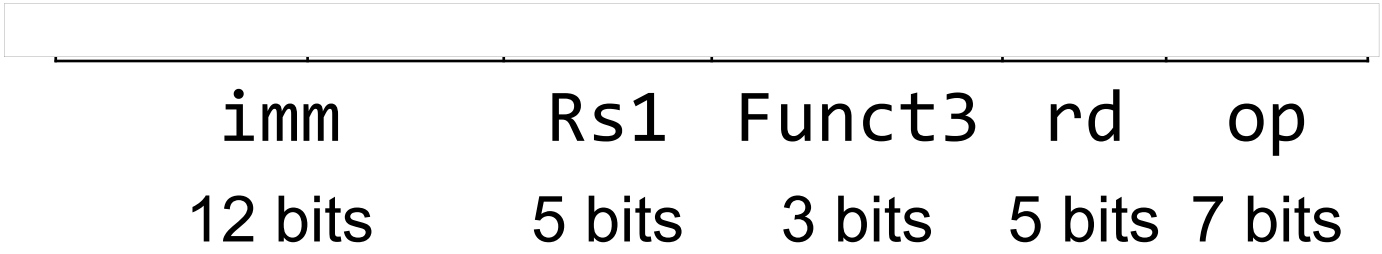
# RISC-V instruction formats

All RISC-V instructions are 32 bits long, have 4 formats

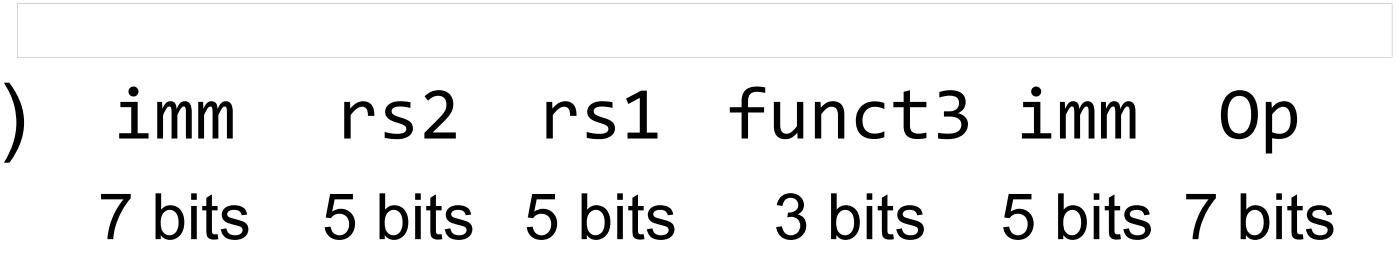
- R-type




- I-type




- S-type (SB-type)




- U-type (UJ-type)

# R-Type (1): Arithmetic and Logic

```
0000000001100100010000100011 0011
31          25 24     20 19      15 14  12 11      7 6           0
funct7 rs2   rs1   Funct3   Rd    op
```

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

| **op** | **funct3** | **mnemonic** | **description** |
|---|---|---|---|
| 0110011 | 000 | ADD rd, rs1, rs2 | R[rd] = R[rs1] + R[rs2] |
| 0110011 | 000 | SUB rd, rs1, rs2 | R[rd] = R[rs1] – R[rs2] |
| 0110011 | 110 | OR rd, rs1, rs2 | R[rd] = R[rs1] \| R[rs2] |
| 0110011 | 100 | XOR rd, rs1, rs2 | R[rd] = R[rs1] ⊕ R[rs2] |

# R-Type (1): Arithmetic and Logic

```
00000000011001000100001000110011
31          25 24    20 19     15 14  12 11     7 6          0
funct7 rs2  rs1  Funct3  Rd   op
```

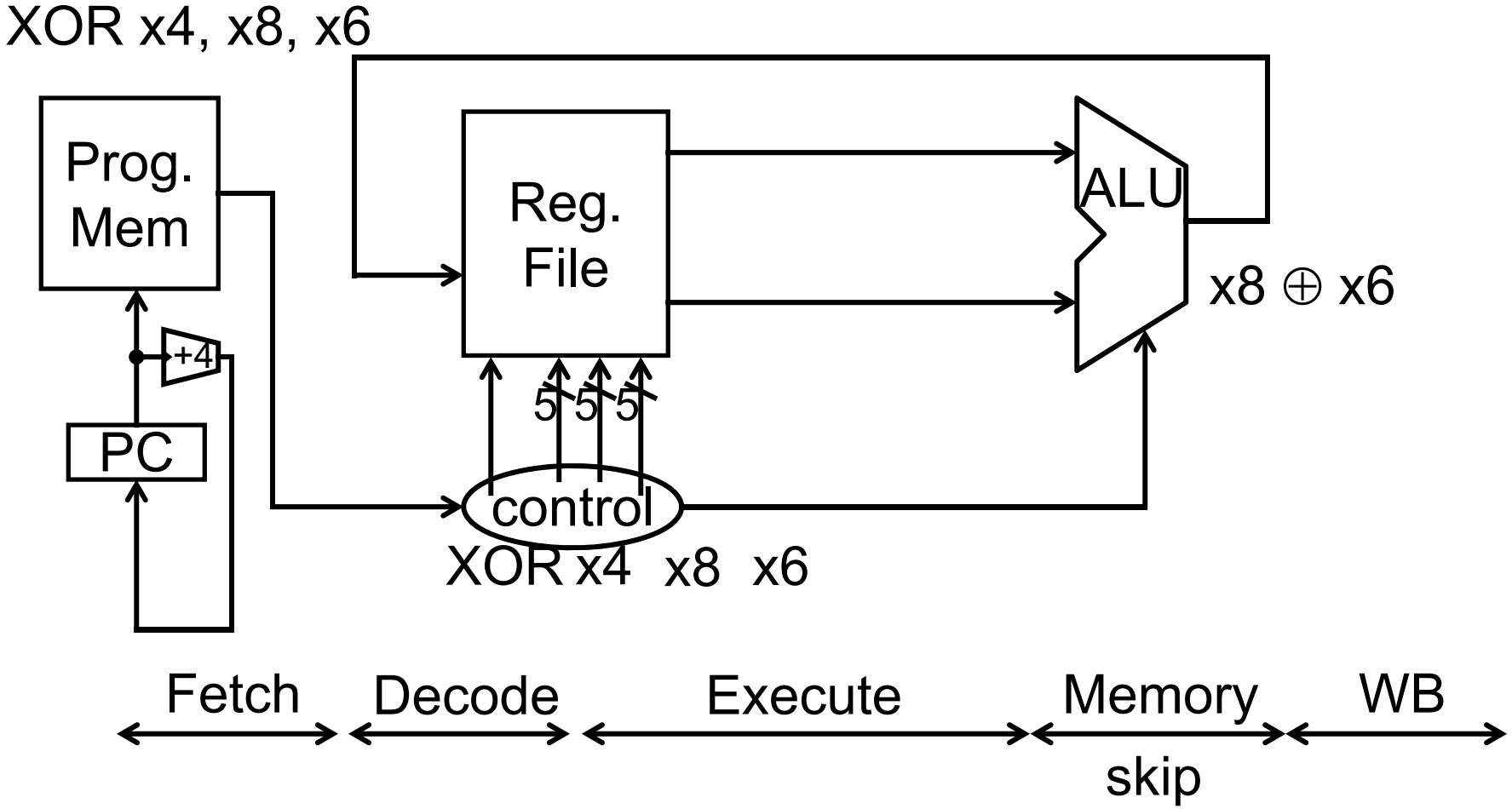
7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

| op | funct3 | mnemonic | description |
|---|---|---|---|
| 0110011 | 000 | ADD rd, rs1, rs2 | R[rd] = R[rs1] + R[rs2] |
| 0110011 | 000 | SUB rd, rs1, rs2 | R[rd] = R[rs1] – R[rs2] |
| 0110011 | 110 | OR rd, rs1, rs2 | R[rd] = R[rs1] \| R[rs2] |
| 0110011 | 100 | XOR rd, rs1, rs2 | R[rd] = R[rs1] ⊕ R[rs2] |

Example: x4 = x8 ⊕ x6 # XOR x4, x8, x6

rd, rs1, rs2

# Arithmetic and Logic




Example: x4 = x8 ⊕ x6 # XOR x4, x8, x6
rd, rs1, rs2

# R-Type (2): Shift Instructions

```
0000000001100010000101000110011
31          25 24     20 19      15 14  12 11      7 6           0
funct7 rs2   rs1  Funct3  Rd   op
```

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

| op | funct3 | mnemonic | description |
|---|---|---|---|
| 0110011 | 001 | SLL rd, rs1, rs2 | R[rd] = R[rs1] << R[rs2] |
| 0110011 | 101 | SRL rd, rs1, rs2 | R[rd] = R[rs1] >>> R[rs2] (zero ext.) |
| 0110011 | 101 | SRA rd, rs1, rs2 | R[rd] = R[rt] >>> R[rs2] (sign ext.) |

# R-Type (2): Shift Instructions

```
0000000001100010000101000110011
31          25 24     20 19      15 14  12 11      7 6           0
funct7 rs2   rs1  Funct3  Rd   op
```

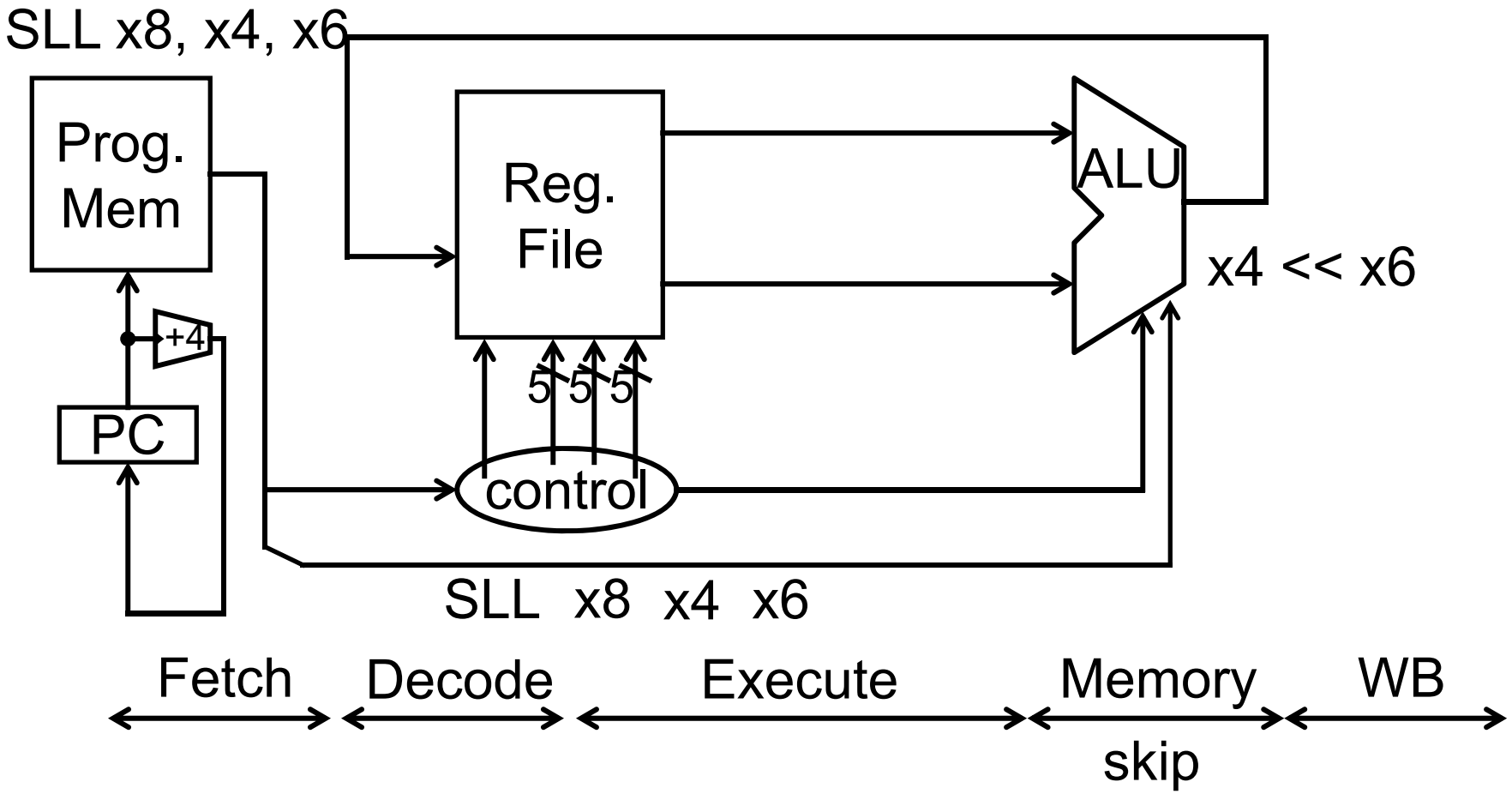
7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

| op | funct3 | mnemonic | description |
|---|---|---|---|
| 0110011 | 001 | SLL rd, rs1, rs2 | R[rd] = R[rs1] << R[rs2] |
| 0110011 | 101 | SRL rd, rs1, rs2 | R[rd] = R[rs1] >>> R[rs2] (zero ext.) |
| 0110011 | 101 | SRA rd, rs1, rs2 | R[rd] = R[rt] >>> R[rs2] (sign ext.) |

Example: x8 = x4 * $2^{x6}$ # SLL x8, x4, x6

x8 = x4 << x6

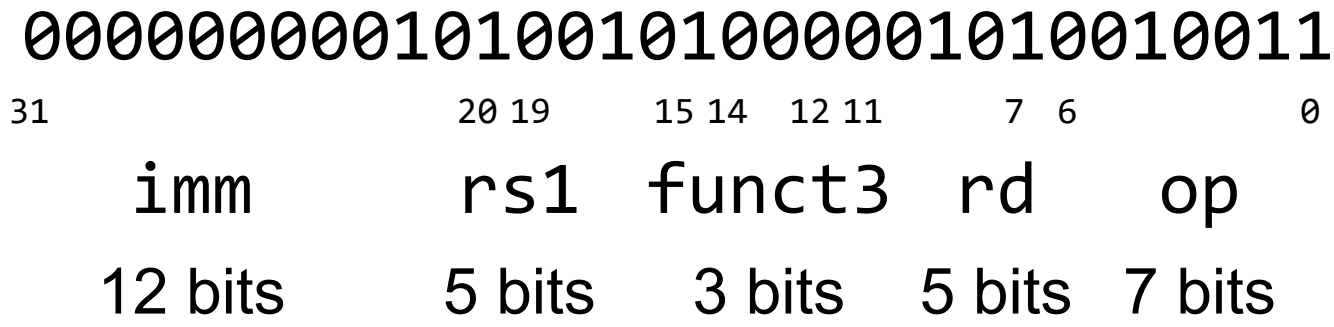
# Shift




Example: $x8 = x4 * 2^{x6}$ # SLL x8, x4, x6

x8 = x4 << x6

# I-Type (1): Arithmetic w/ immediates




| op | funct3 | mnemonic | description |
|---|---|---|---|
| 0010011 | 000 | ADDI rd, rs1, imm | R[rd] = R[rs1] + sign_extend(imm) |
| 0010011 | 111 | ANDI rd, rs1, imm | R[rd] = R[rs1] & sign_extend(imm) |
| 0010011 | 110 | ORI rd, rs1, imm | R[rd] = R[rs1] \| sign_extend(imm) |

# I-Type (1): Arithmetic w/ immediates

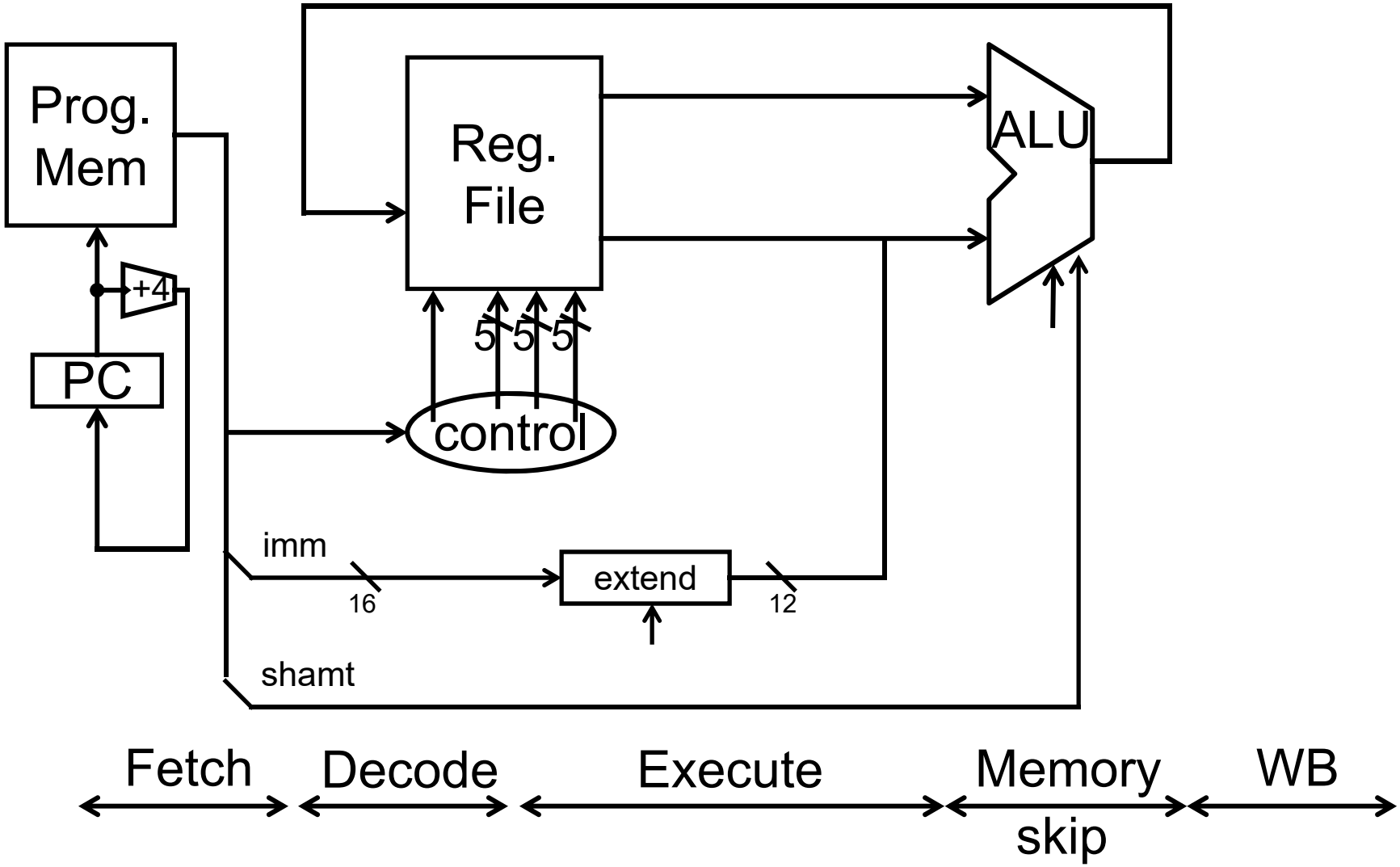
```
00000000010100101000001010010011
31                  20 19     15 14  12 11     7  6          0
    imm              rs1  funct3  rd    op
```

12 bits 5 bits 3 bits 5 bits 7 bits

| op | funct3 | mnemonic | description |
|---|---|---|---|
| 0010011 | 000 | ADDI rd, rs1, imm | R[rd] = R[rs1] + sign_extend(imm) |
| 0010011 | 111 | ANDI rd, rs1, imm | R[rd] = R[rs1] & sign_extend(imm) |
| 0010011 | 110 | ORI rd, rs1, imm | R[rd] = R[rs1] \| sign_extend(imm) |

Example: x5 = x5 + 5 # ADDI x5, x5, 5
x5 += 5

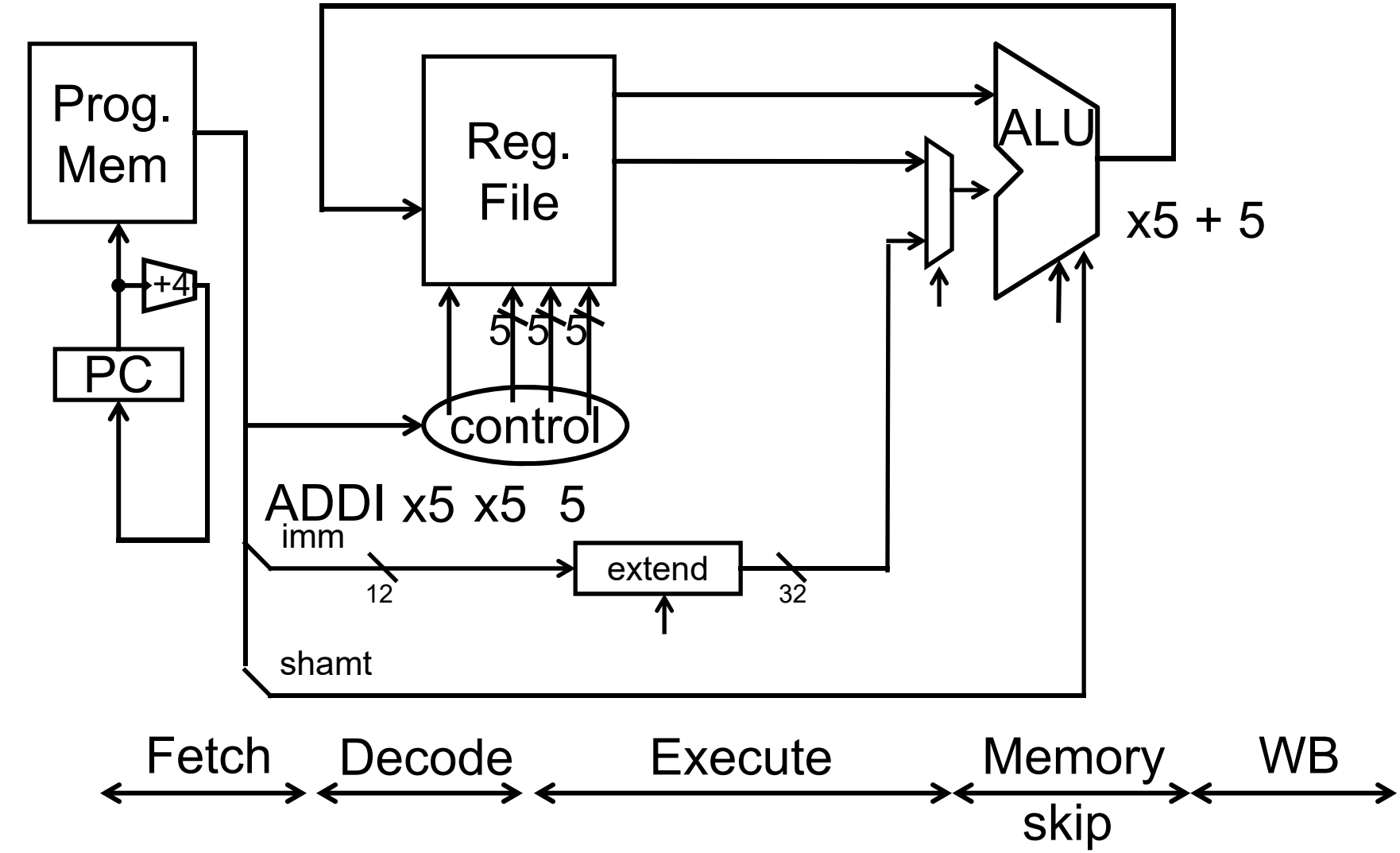
# Arithmetic w/ immediates




Example: x5 = x5 + 5  # ADDI x5, x5, 5

# Arithmetic w/ immediates

ADDI x5, x5, 5




Example: x5 = x5 + 5 # ADDI x5, x5, 5

# iClicker Question

- To compile the code `y = z + 1`, assuming y is stored in `X1` and z is stored in `X2`, you can use the `ADDI` instruction. What is the largest number for which we can continue to use `ADDI`?

(a) 12
(b) $2^{12}$-1 = 4,095
(c) $2^{12-1}$ -1 = 2,047
(d) $2^{16-1}$ = 65,535
(e) $2^{32-1}$ = ~4.3 billion

# iClicker Question

- To compile the code `y = z + 1`, assuming y is stored in `X1` and x is stored in `X2`, you can use the `ADDI` instruction. What is the largest number for which we can continue to use `ADDI`?

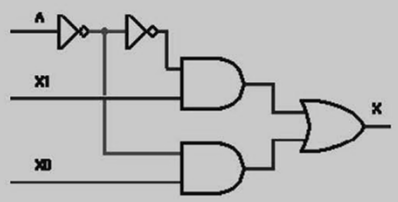
(a) 12

(b) $2^{12}$-1 = 2,047

(c) $2^{12-1}$ = 4,095

(d) $2^{16-1}$ = 65,535

(e) $2^{32-1}$ = ~4.3 billion




to:

from:

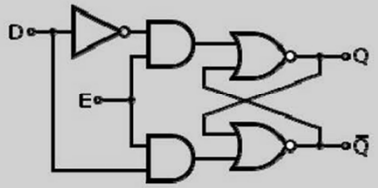
if I were a mux I'd select you


valentine, we two's complement each other perfectly <3

to:

from:

you make my heart flip-flop <3




to:

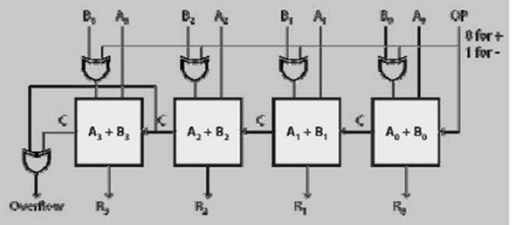
from:


I'm falling (edge) in love with you

to: from:

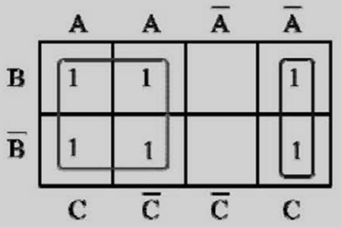
to:

from:



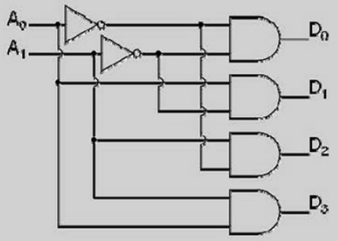

I'm overflowing with love for you

you hold the karnaugh map to my heart




to:

from:




it's not hard to decode my feelings for you <3

to: from:

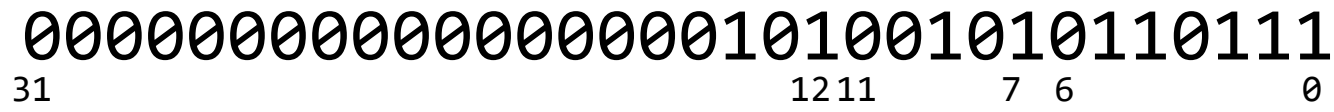
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

are u the leftmost bit? because u are the most significant to me <3

to: from:

THIS IS WHAT LEARNING LOGIC GATES FEELS LIKE

# U-Type (1): "Load" Upper Immediate





| op | mnemonic | description |
|---|---|---|
| 0110111 | LUI rd, imm | R[rd] = sign_ext(imm) << 12 |

# U-Type (1): "Load" Upper Immediate

```
00000000000000001010010110111
31                          12 11    7 6         0
```

| imm | rd | op |
|---|---|---|
| 20 bits | 5 bits | 7 bits |

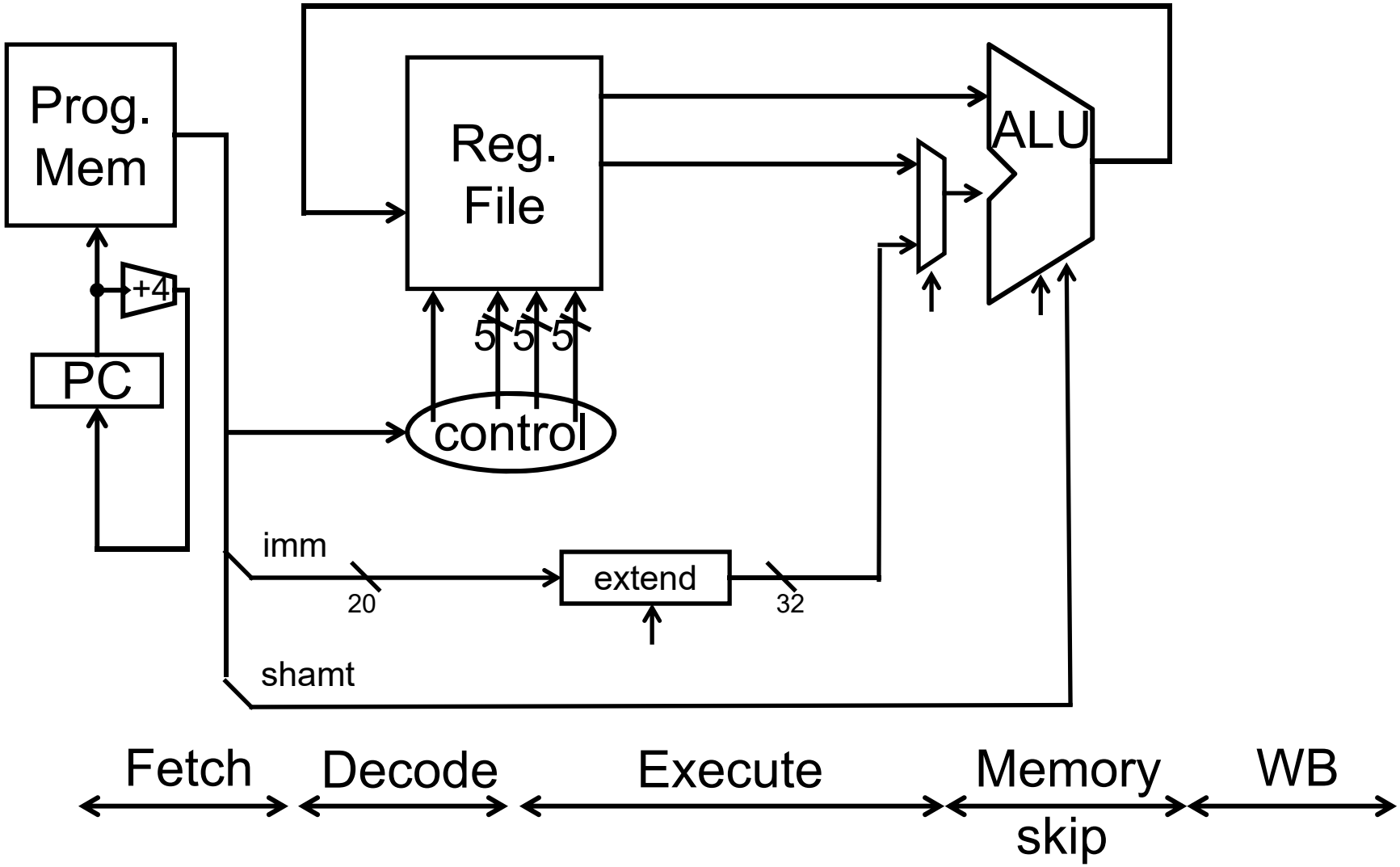
WORST NAME EVER !

| op | mnemonic | description |
|---|---|---|
| 0110111 | LUI rd, imm | R[rd] = sign_ext(imm) << 12 |

Example: x5 = 0x5000 # LUI x5, 5

Example:
```
LUI x5, 0xbeef1
ADDI x5, x5 0x234
```
What does x5 = 0xbeef1234

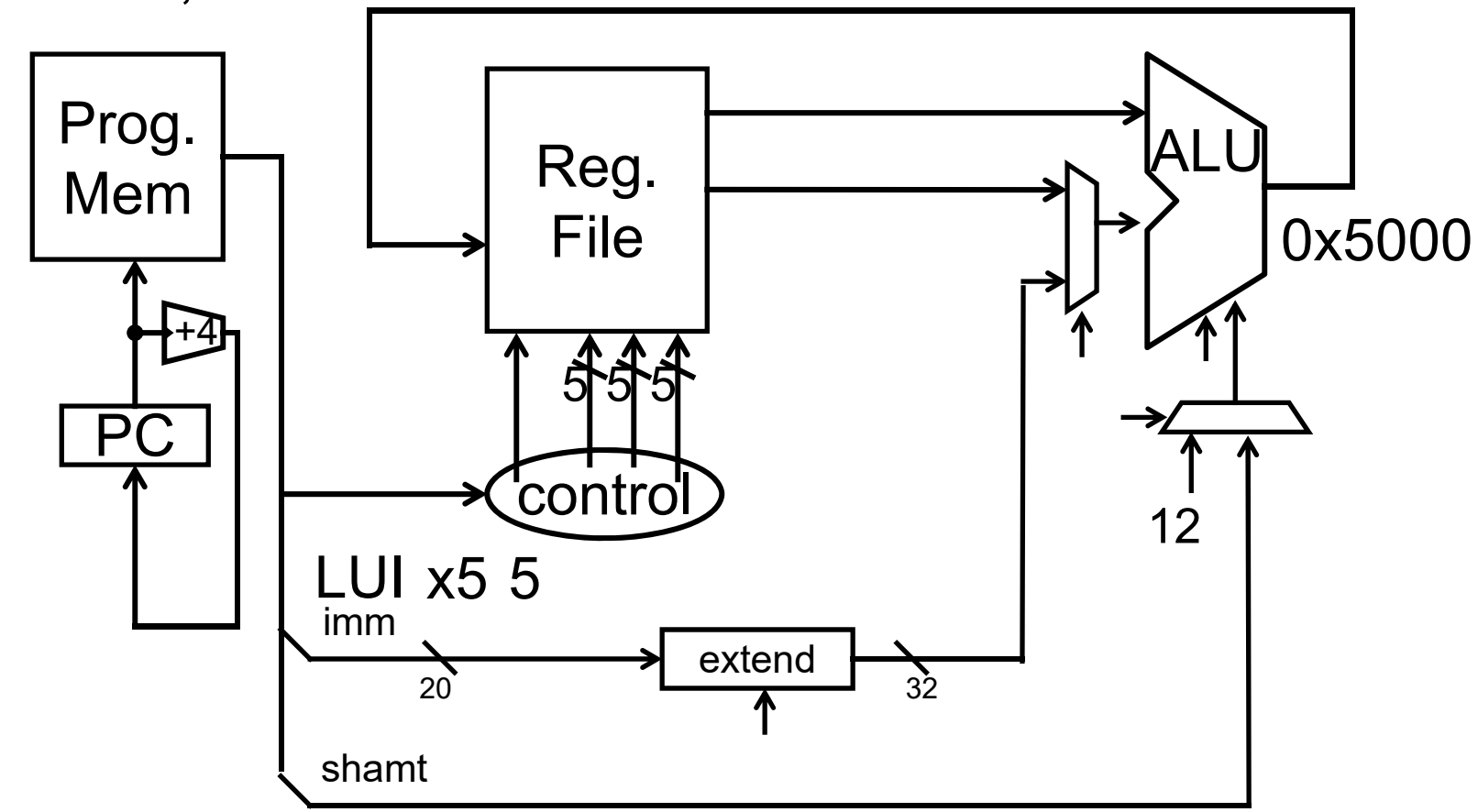
# Load Upper Immediate




Example: x5 = 0x5000 # LUI x5, 5

# Load Upper Immediate

LUI x5, 5





Example: x5 = 0x5000 # LUI x5, 5

# RISC-V Instruction Types

- Arithmetic/Logical
  - R-type: result and two source registers, shift amount
  - I-type: result and source register, shift amount in 16-bit immediate with sign/zero extension
  - U-type: result register, 16-bit immediate with sign/zero extension
- Memory Access
  - I-type for loads and S-type for stores
  - load/store between registers and memory
  - word, half-word and byte operations
- Control flow
  - U-type: jump-and-link
  - I-type: jump-and-link register
  - SB-type: conditional branches: pc-relative addresses

# I-Type (2): Load Instructions

```
00000000010000101010000010000011
31                  20 19    15 14  12 11     7  6          0
```

| imm | rs1 | funct3 | rd | op |
|---|---|---|---|---|
| 12 bits | 5 bits | 3 bits | 5 bits | 7 bits |

base + offset addressing

| op | funct3 | mnemonic | Description |
|---|---|---|---|
| 0000011 | 000 | LB rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 001 | LH rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 010 | LW rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 011 | LD rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 100 | LBU rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 101 | LHU rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 110 | LWU rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |

signed offsets

# I-Type (2): Load Instructions

```
00000000010000101010000010000011
31                  20 19    15 14  12 11     7  6          0
```

| imm | rs1 | funct3 | rd | op |
|---|---|---|---|---|
| 12 bits | 5 bits | 3 bits | 5 bits | 7 bits |

base + offset addressing

| op | funct3 | mnemonic | Description |
|---|---|---|---|
| 0000011 | 000 | LB rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 001 | LH rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 010 | LW rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 011 | LD rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 100 | LBU rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 101 | LHU rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 110 | LWU rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |

signed offsets

Example: x1 = Mem[4+x5] # LW x1, x5, 4
LW x1 4(x5)

# I-Type (2): Load Instructions

```
00000000010000101010000010000011
31                  20 19    15 14  12 11     7  6          0
```

| imm | rs1 | funct3 | rd | op |
|---|---|---|---|---|
| 12 bits | 5 bits | 3 bits | 5 bits | 7 bits |

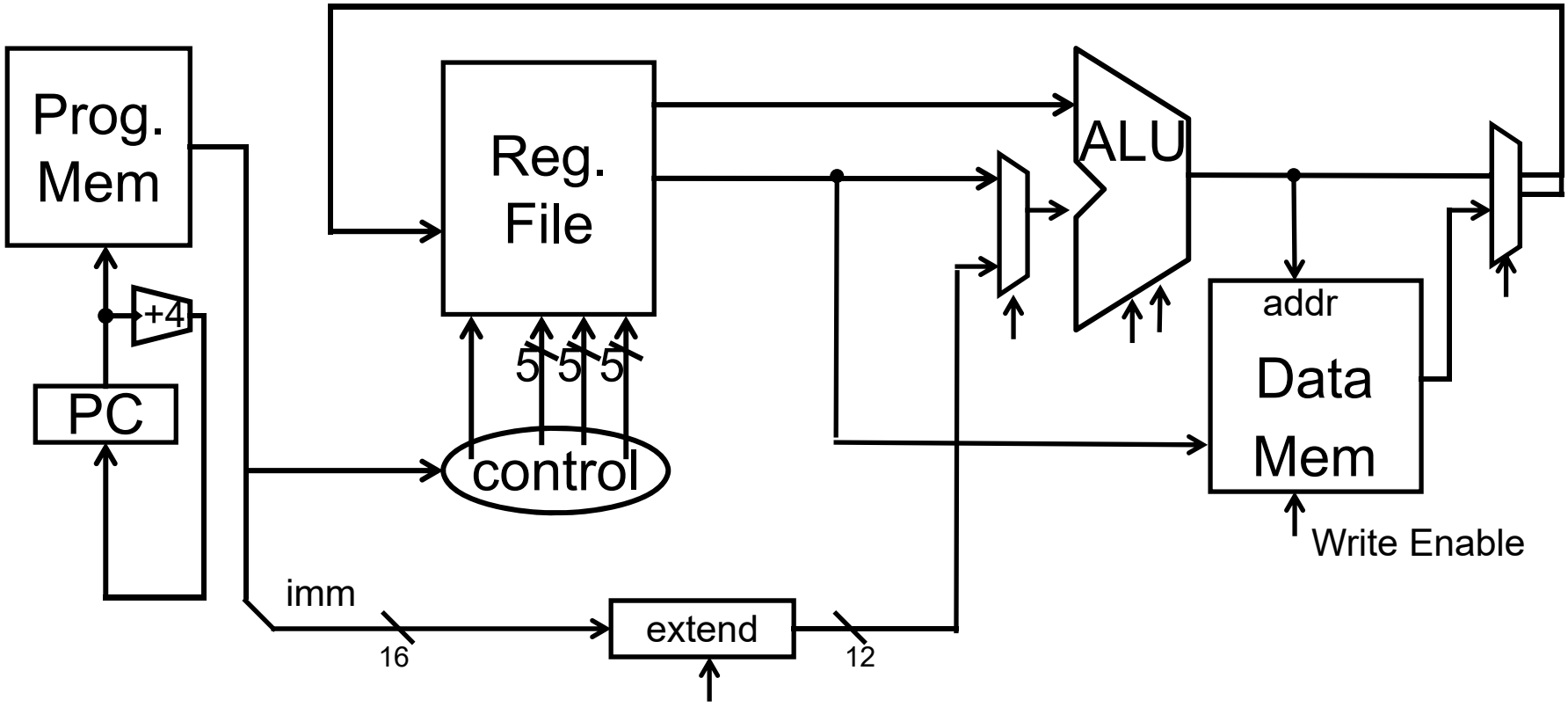
base + offset addressing

| op | funct3 | mnemonic | Description |
|---|---|---|---|
| 0000011 | 000 | LB rd, rs1, imm | R[rd] = sign_ext(Mem[imm+R[rs1]]) |
| 0000011 | 001 | LH rd, rs1, imm | R[rd] = sign_ext(Mem[imm+R[rs1]]) |
| 0000011 | 010 | LW rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 011 | LD rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |
| 0000011 | 100 | LBU rd, rs1, imm | R[rd] = zero_ext(Mem[imm+R[rs1]]) |
| 0000011 | 101 | LHU rd, rs1, imm | R[rd] = zero_ext(Mem[imm+R[rs1]]) |
| 0000011 | 110 | LWU rd, rs1, imm | R[rd] = Mem[imm+R[rs1]] |

signed offsets

Example: x1 = Mem[4+x5] # LW x1, x5, 4
LW x1 4(x5)

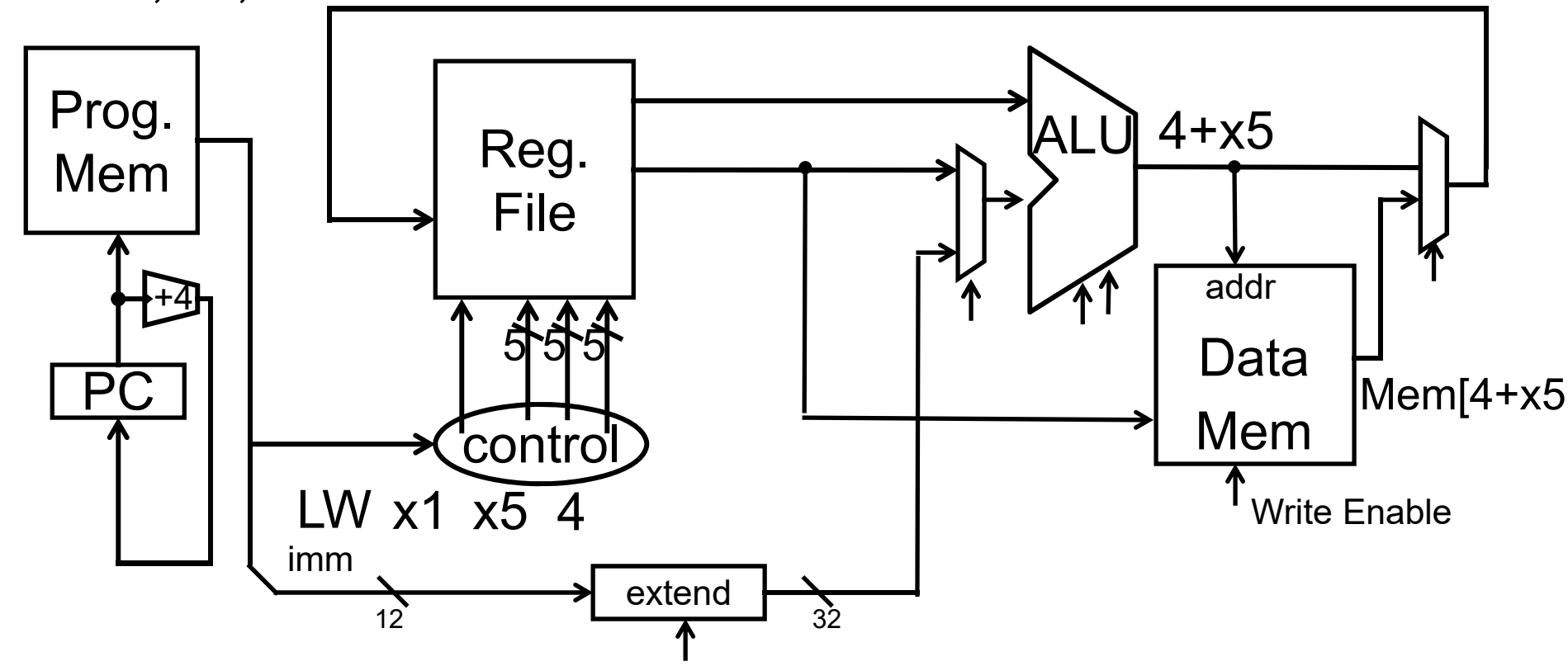
# Memory Operations: Load




Example: x1 = Mem[4+x5] # LW x1, x5, 4
LW x1 4(x5)

# Memory Operations: Load

LW x1, x5, 4




Fetch | Decode | Execute | Memory | WB

Example: x1 = Mem[4+x5] # LW x1, x5, 4

LW x1 4(x5)

# S-Type (1): Store Instructions

```
00001000000000101010000000010011
31          25 24    20 19    15 14  12 11     7 6          0
 imm    rs2   rs1  funct3 imm    Op
```

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

base + offset addressing

| op | funct3 | mnemonic | description |
|---|---|---|---|
| 0100011 | 000 | SB rs2, rs1, imm | Mem[sign_ext(imm)+R[rs1]] = R[rd] |
| 0100011 | 001 | SH rs2, rs1, imm | Mem[sign_ext(imm)+R[rs1]] = R[rd] |
| 0100011 | 010 | SW rs2, rs1, imm | Mem[sign_ext(imm)+R[rs1]] = R[rd] |

signed offsets

# S-Type (1): Store Instructions

```
00001000000000101010000000010011
31          25 24    20 19    15 14  12 11     7 6          0
 imm    rs2   rs1  funct3 imm    Op
```

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

base + offset addressing

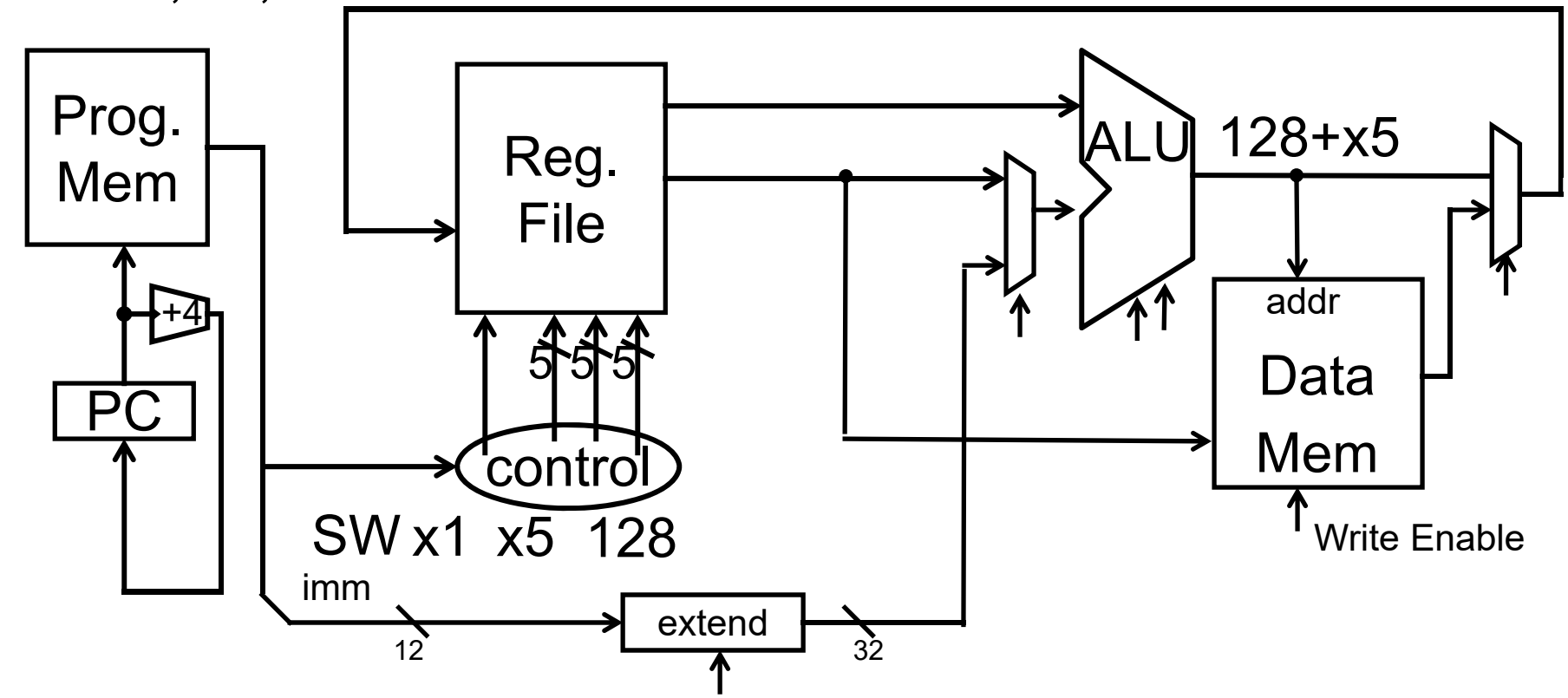
| op | funct3 | mnemonic | description |
|---|---|---|---|
| 0100011 | 000 | SB rs2, rs1, imm | Mem[sign_ext(imm)+R[rs1]] = R[rd] |
| 0100011 | 001 | SH rs2, rs1, imm | Mem[sign_ext(imm)+R[rs1]] = R[rd] |
| 0100011 | 010 | SW rs2, rs1, imm | Mem[sign_ext(imm)+R[rs1]] = R[rd] |

signed offsets

Example: Mem[128+x5] = x1 # SW x1, x5, 128
SW x1 128(x5)

# Memory Operations: Load

SW x1, x5, 128




Fetch | Decode | Execute | Memory | WB

Example: Mem[4+x5] = x1 # SW x1, x5, 128

SW x1 128(x5)

# Memory Layout Options

- # x5 contains 5 (0x00000005)

- SB x5, x0, 0
- SB x5, x0, 2
- SW x5, x0, 8

- Two ways to store a word in memory.

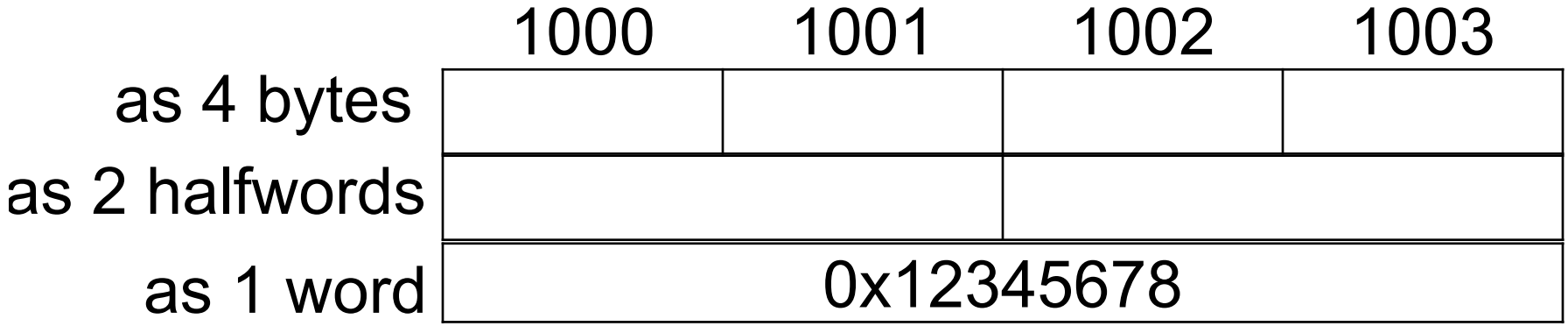
Endianness: ordering of bytes within a memory word

| | |
|---|---|
| | 0x000fffff |
| | ... |
| | 0x0000000b |
| | 0x0000000a |
| | 0x00000009 |
| | 0x00000008 |
| | 0x00000007 |
| | 0x00000006 |
| | 0x00000005 |
| | 0x00000004 |
| | 0x00000003 |
| | 0x00000002 |
| | 0x00000001 |
| | 0x00000000 |

# Little Endian

Endianness: Ordering of bytes within a memory word

Little Endian = least significant part first (RISC-V, x86)




Clicker Question: What values go in the byte-sized boxes with addresses 1000 and 1001?

a) 0x8, 0x7

b) 0x78, 0x56

c) 0x87, 0x65

d) 0x12, 0x34

e) 0x1, 0x2

*THIS IS WHAT YOUR PROJECTS WILL BE*

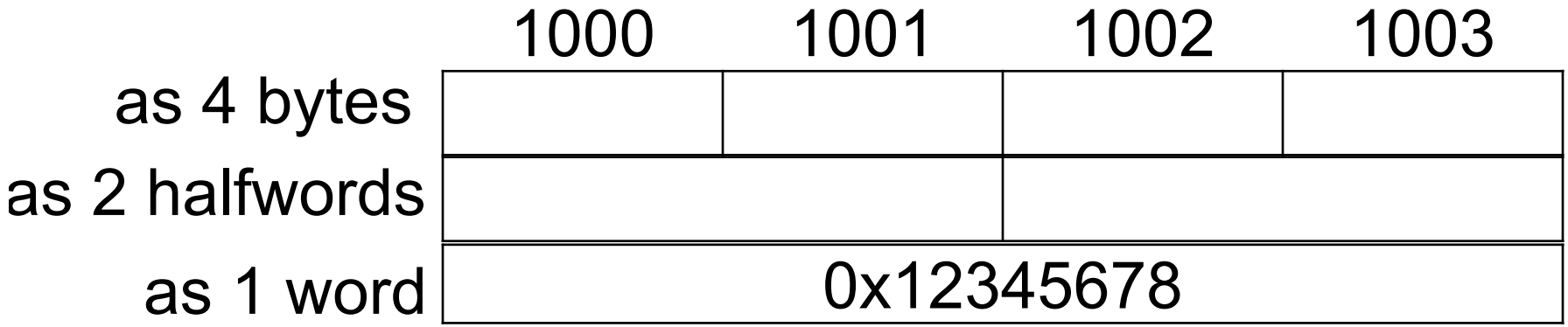


70

# Little Endian

Endianness: Ordering of bytes within a memory word

Little Endian = least significant part first (RISC-V, x86)




Clicker Question: What values go in the byte-sized boxes with addresses 1000 and 1001?

a) 0x8, 0x7

b) 0x78, 0x56

c) 0x87, 0x65

d) 0x12, 0x34

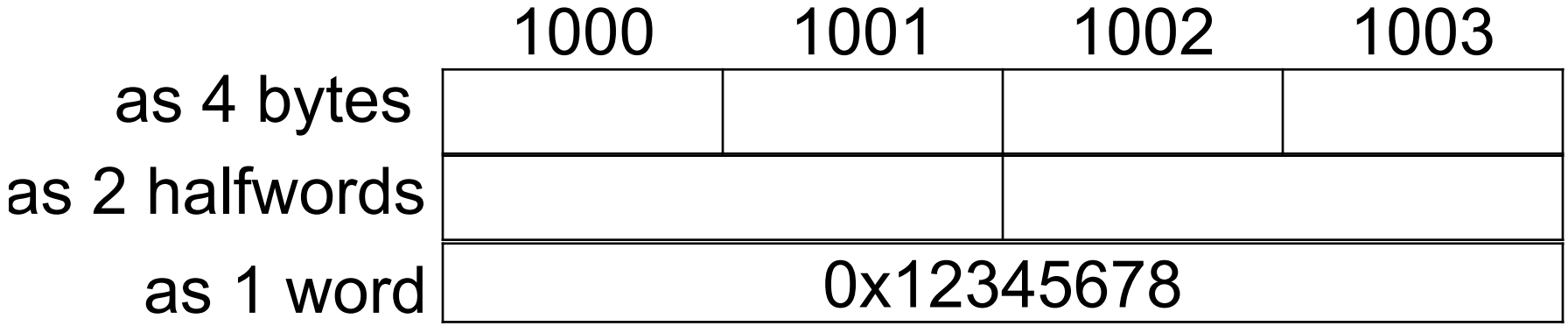
e) 0x1, 0x2

*THIS IS WHAT YOUR PROJECTS 7 WILL BE*

# Little Endian

Endianness: Ordering of bytes within a memory word

Big Endian = most significant part first (MIPS, networks)




Clicker Question: What value goes in the half-word sized box with address 1000?

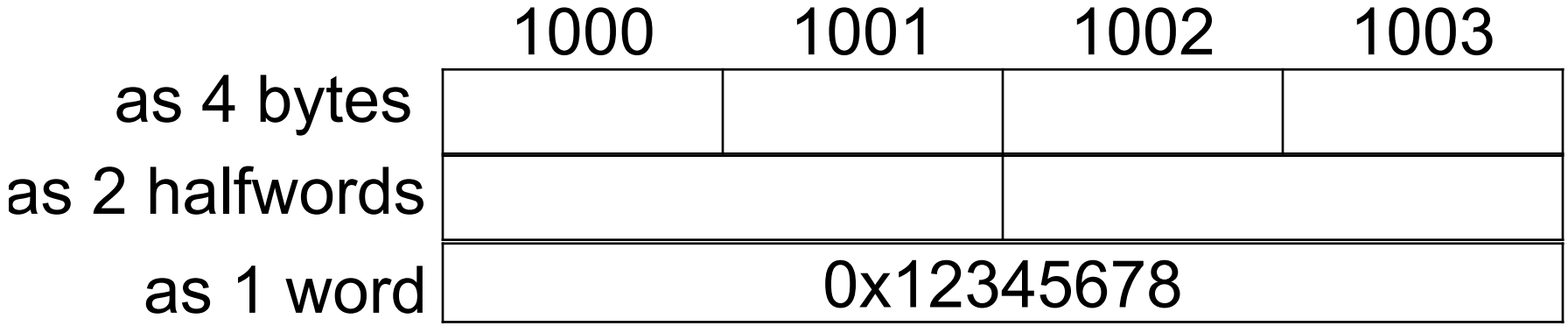
a) 0x1
b) 0x12
c) 0x1234
d) 0x4321
e) 0x5678

*THIS IS WHAT YOUR PROJECTS WILL BE*

# Little Endian

Endianness: Ordering of bytes within a memory word

Big Endian = most significant part first (MIPS, networks)




Clicker Question: What value goes in the half-word sized box with address 1000?

a) 0x1
b) 0x12
c) 0x1234
d) 0x4321
e) 0x5678

*THIS IS WHAT YOUR PROJECTS WILL BE*



73

# Little Endian

*WHAT WE USE IN 3410*

Little Endian = least significant part first (RISC-V, x86)

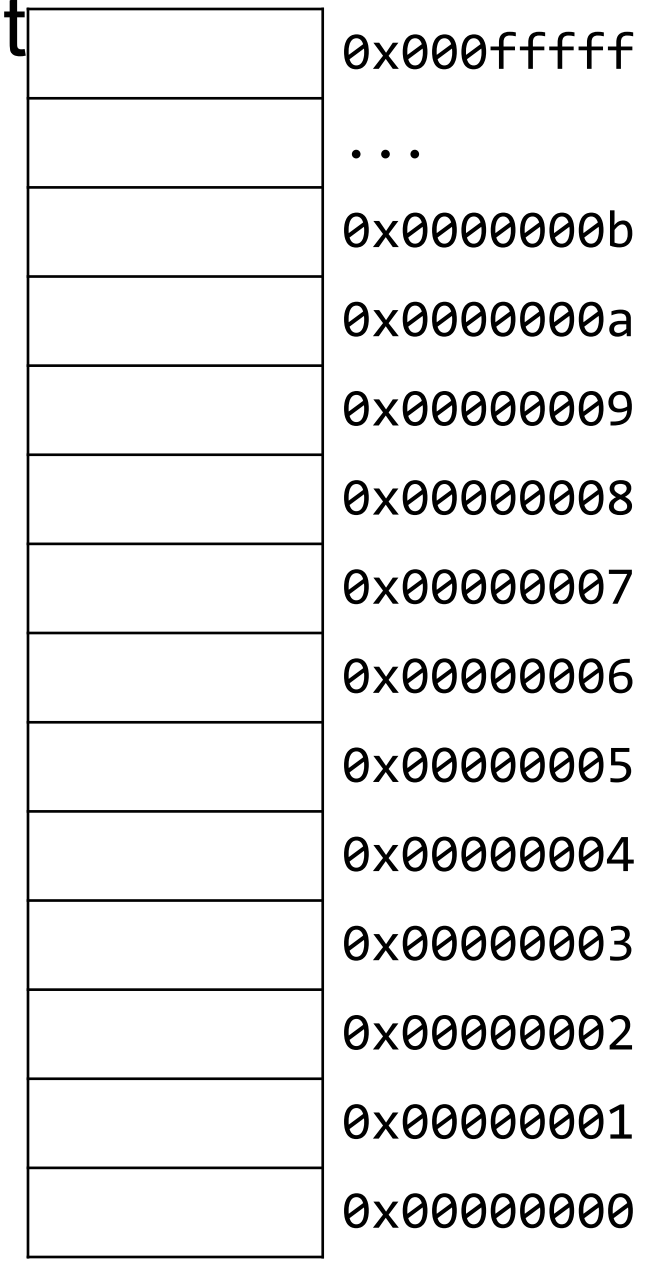
Example:

r5 contains 5 (0x00000005)

`SW r5, 8(r0)`

Clicker Question: After executing the store, which byte address contains the byte 0x05?

a) 0x00000008
b) 0x00000009
c) 0x0000000a
d) 0x0000000b
e) I don't know



# Little Endian

*WHAT WE USE IN 3410*

Little Endian = least significant part first (RISC-V, x86)

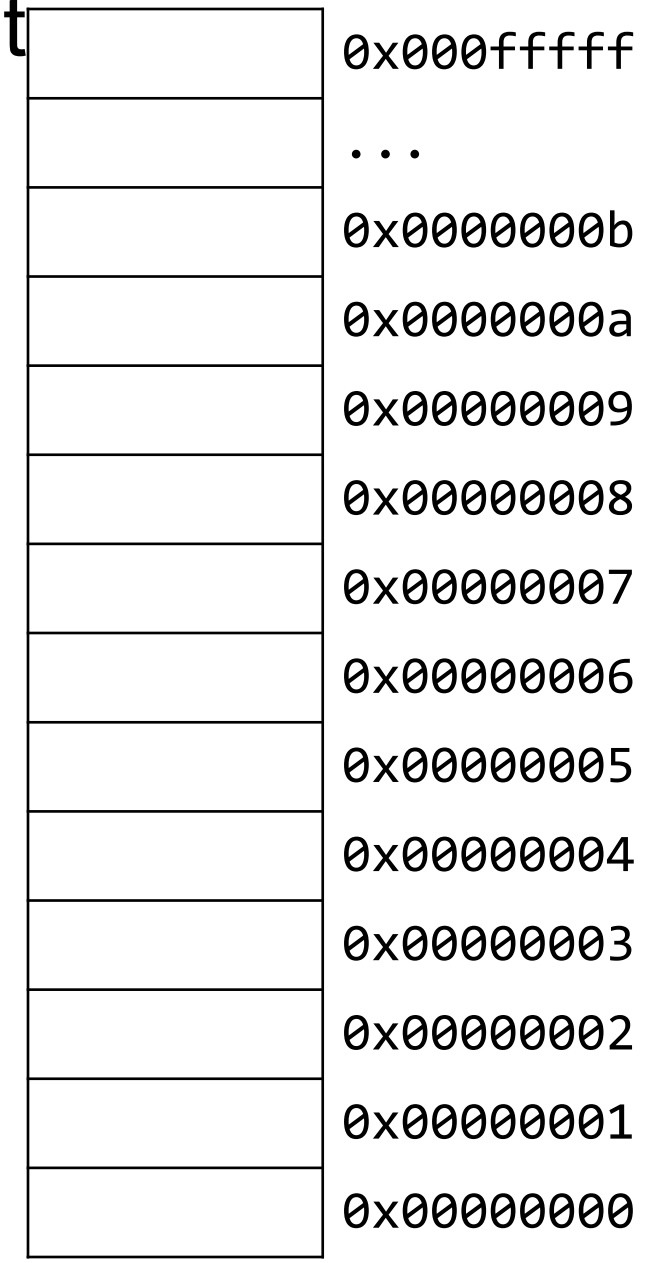
Example:

r5 contains 5 (0x00000005)

`SW r5, 8(r0)`

Clicker Question: After executing the store, which byte address contains the byte 0x05?

a) 0x00000008
b) 0x00000009
c) 0x0000000a
d) 0x0000000b
e) I don't know



# Big Endian

Big Endian = most significant part first (some MIPS, networks)

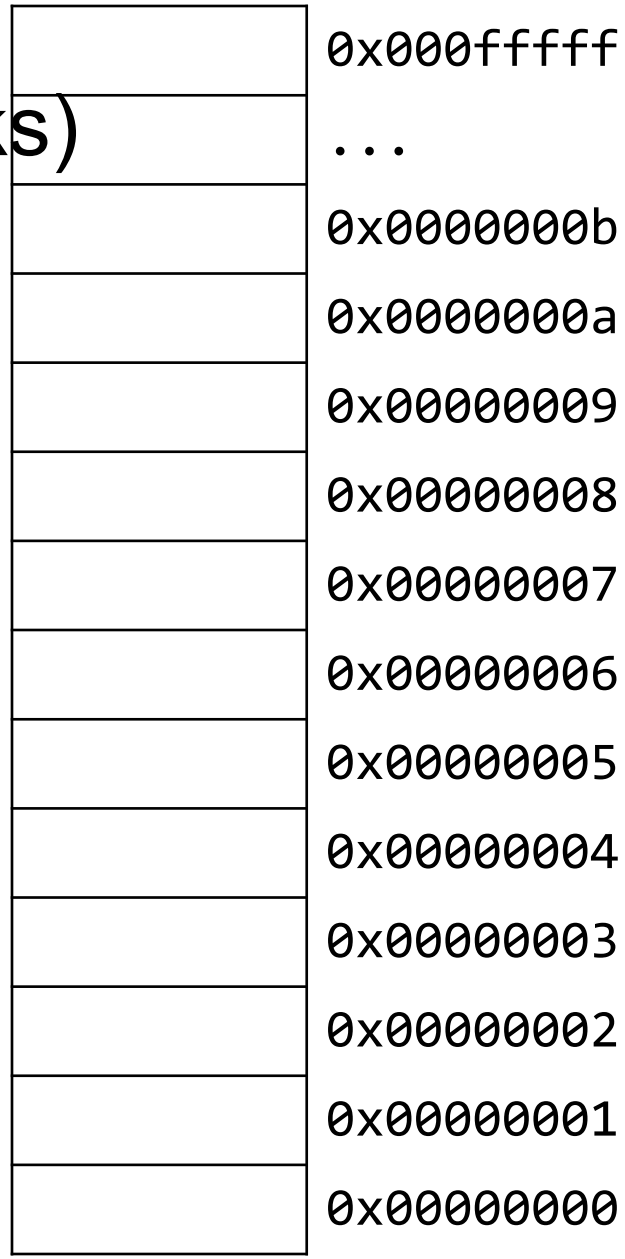
Example:

r5 contains 5 (0x00000005)

`SW r5, 8(r0)`

Clicker Question: After executing the store, which byte address contains the byte 0x05?

a) 0x00000008
b) 0x00000009
c) 0x0000000a
d) 0x0000000b
e) I don't know



# Big Endian

Big Endian = most significant part first (some MIPS, networks)

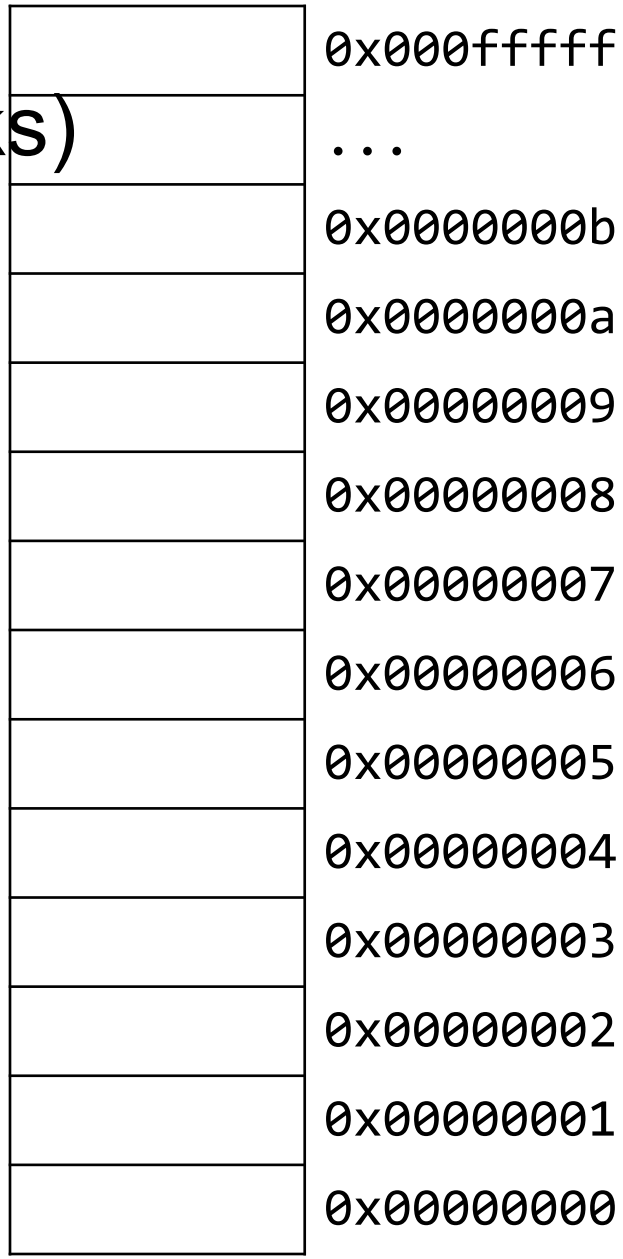
Example:

r5 contains 5 (0x00000005)

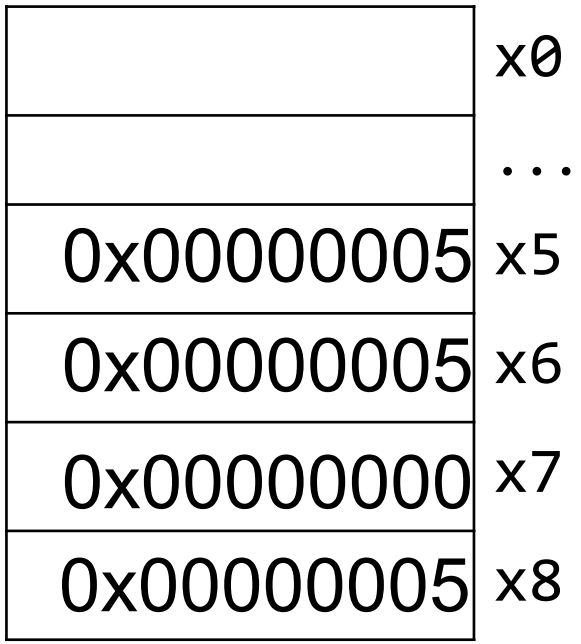
`SW r5, 8(r0)`

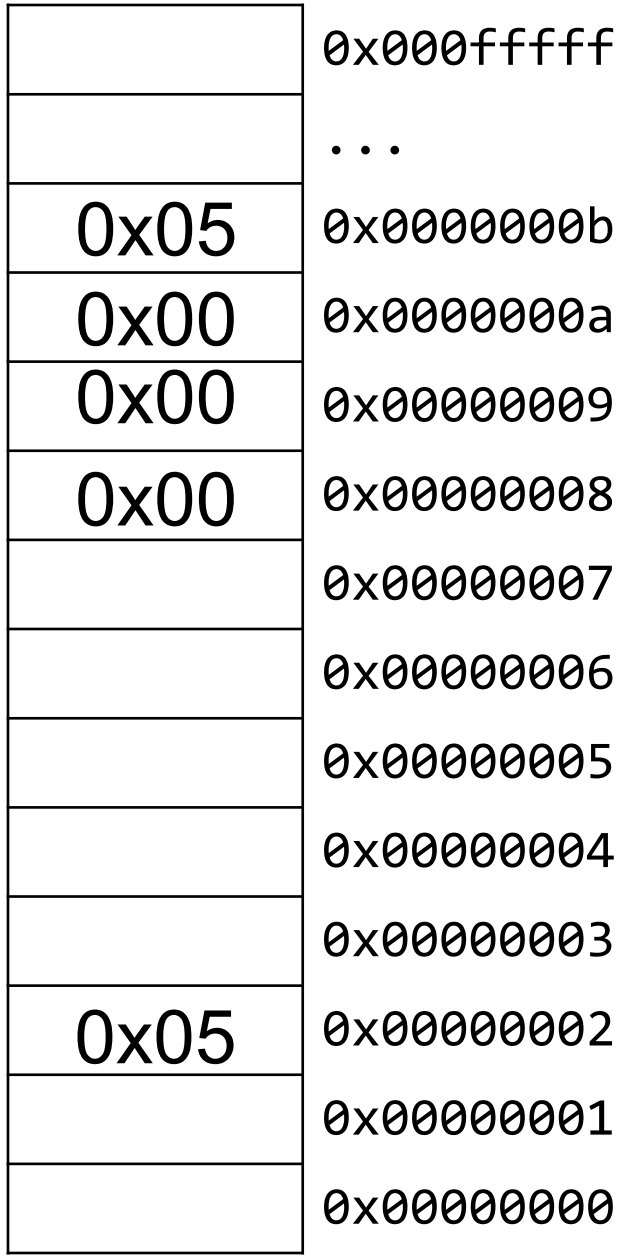
Clicker Question: After executing the store, which byte address contains the byte 0x05?

a) 0x00000008
b) 0x00000009
c) 0x0000000a
d) 0x0000000b
e) I don't know



# Big Endian Memory Layout



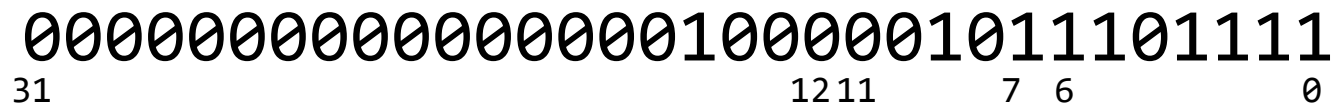




- SB x5, x0, 2
- LB x6, x0, 2
- SW x5, x0, 8
- LB x7, x0, 8
- LB x8, x0, 11

# RISC-V Instruction Types

- Arithmetic/Logical
  - R-type: result and two source registers, shift amount
  - I-type: result and source register, shift amount in 16-bit immediate with sign/zero extension
  - U-type: result register, 16-bit immediate with sign/zero extension
- Memory Access
  - I-type for loads and S-type for stores
  - load/store between registers and memory
  - word, half-word and byte operations
- Control flow
  - U-type: jump-and-link
  - I-type: jump-and-link register
  - S-type: conditional branches: pc-relative addresses

# UJ-Type (2): Jump and Link




| op | Mnemonic | Description |
|---|---|---|
| 1101111 | JAL rd, imm | R[rd] = PC+4;<br>PC=PC + sext(imm) |

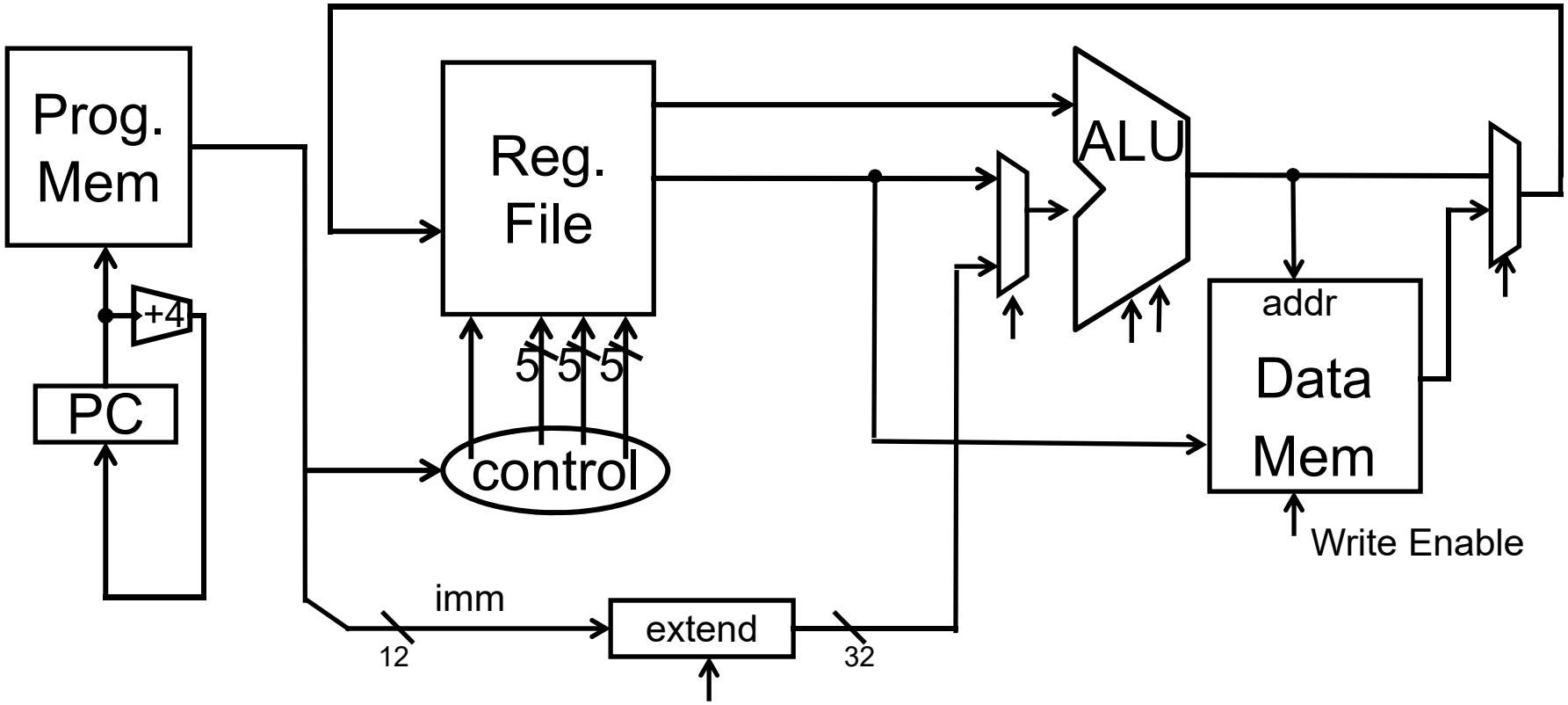
Example: x5 = PC+4 # JAL x5, 16

PC = PC + 16 (i.e. 16 == 8<<1)

Why?

Function/procedure calls

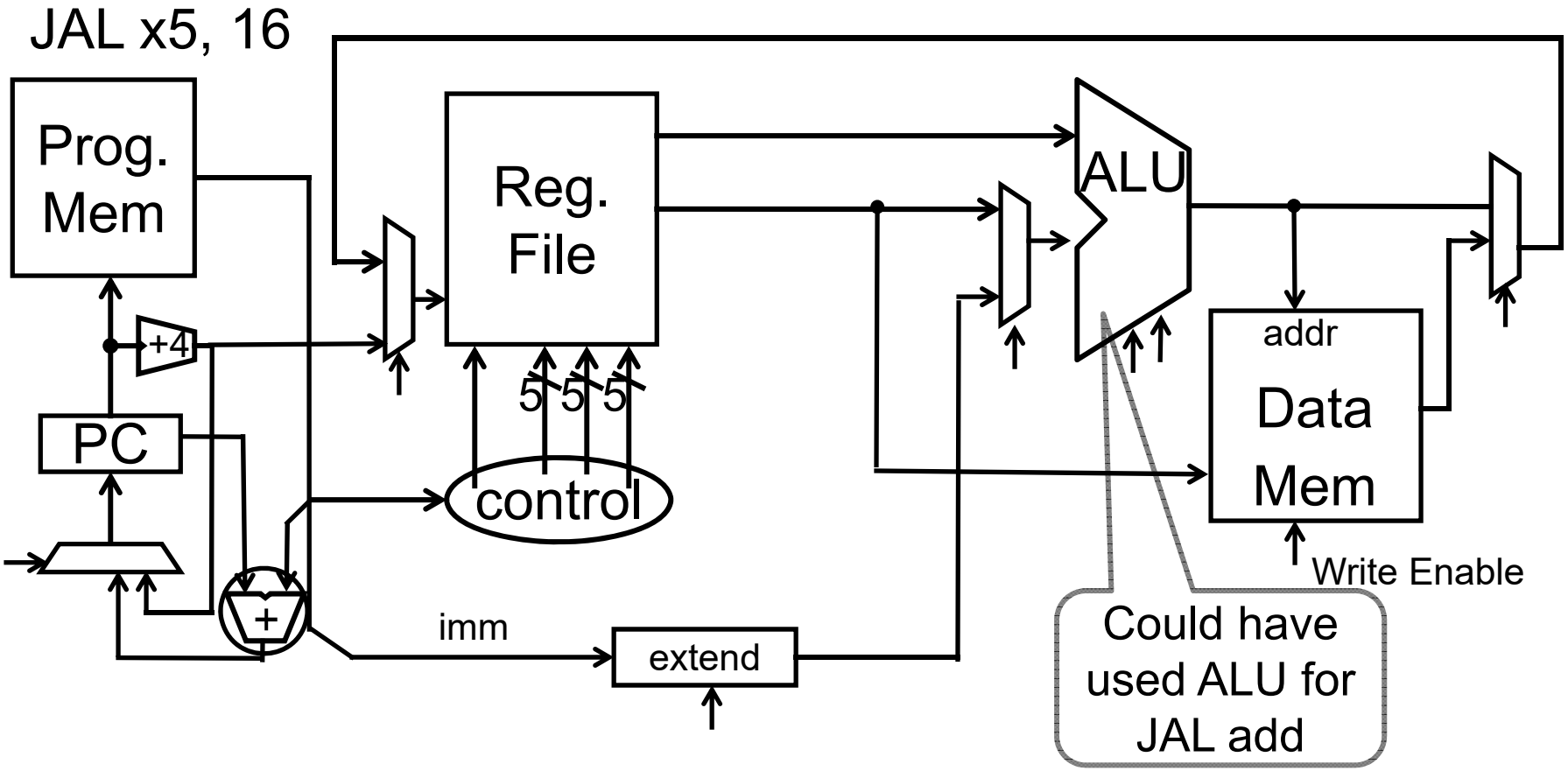
# Jump and Link




Example: x5 = PC+4 # JAL x5, 16

PC = PC + 16 (i.e. 16 == 8<<1)

# Jump and Link




Example: x5 = PC+4 # JAL x5, 16

PC = PC + 16 (i.e. 16 == 8<<1)

# I-Type (3): Jump and Link Register

```
00000001000001000000001011100111
31                  20 19    15 14  12 11     7  6          0
```

| imm | rs1 | funct3 | rd | op |
|---|---|---|---|---|
| 12 bits | 5 bits | 3 bits | 5 bits | 7 bits |

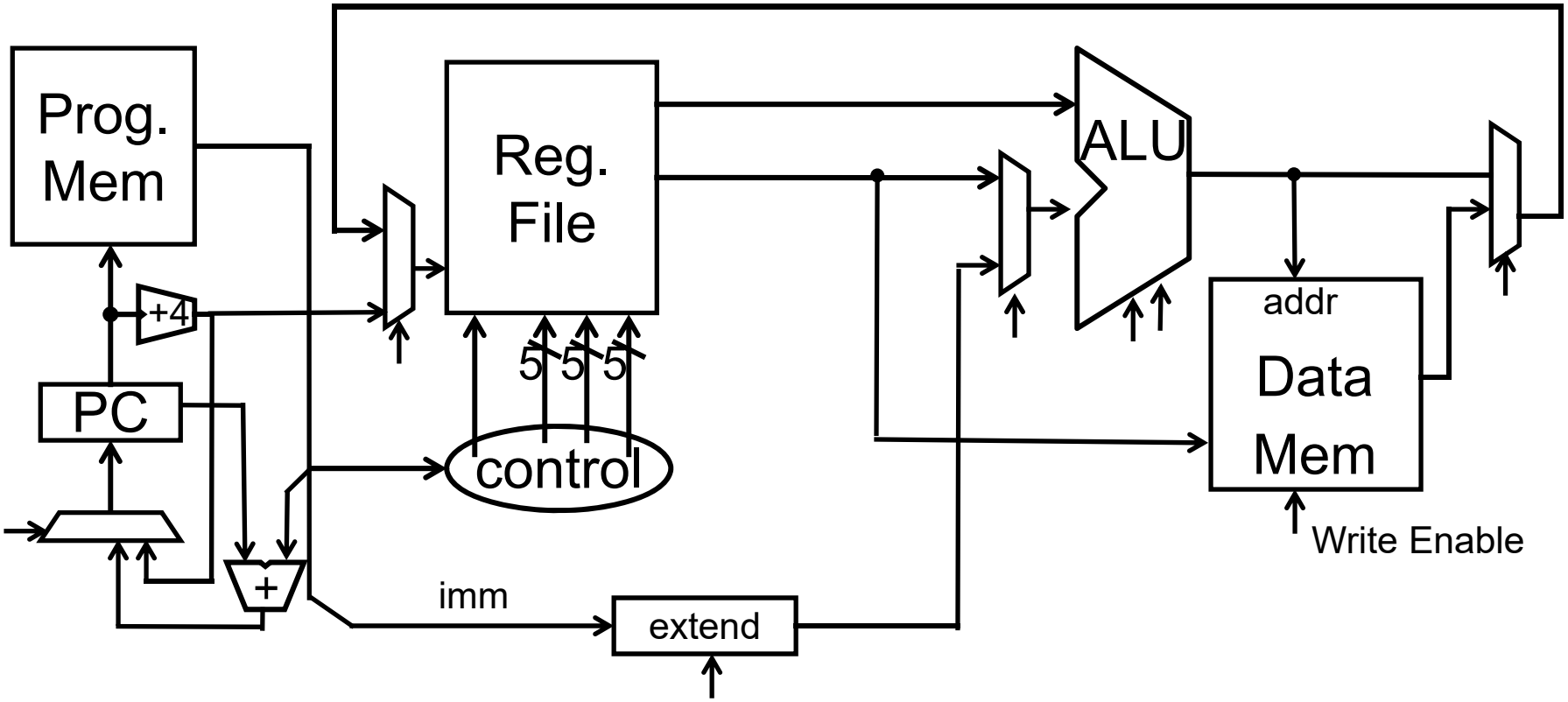
| op | funct3 | Mnemonic | Description |
|---|---|---|---|
| 1100111 | 000 | JALR rd, rs1, imm | R[rd] = PC+4;<br>PC=(R[rs1]+sign_ex(imm))&0xfffffffe |

Example: x5 = PC+4 # JALR x5, x4, 16
PC = x4 + 16

Why?

Function/procedure calls

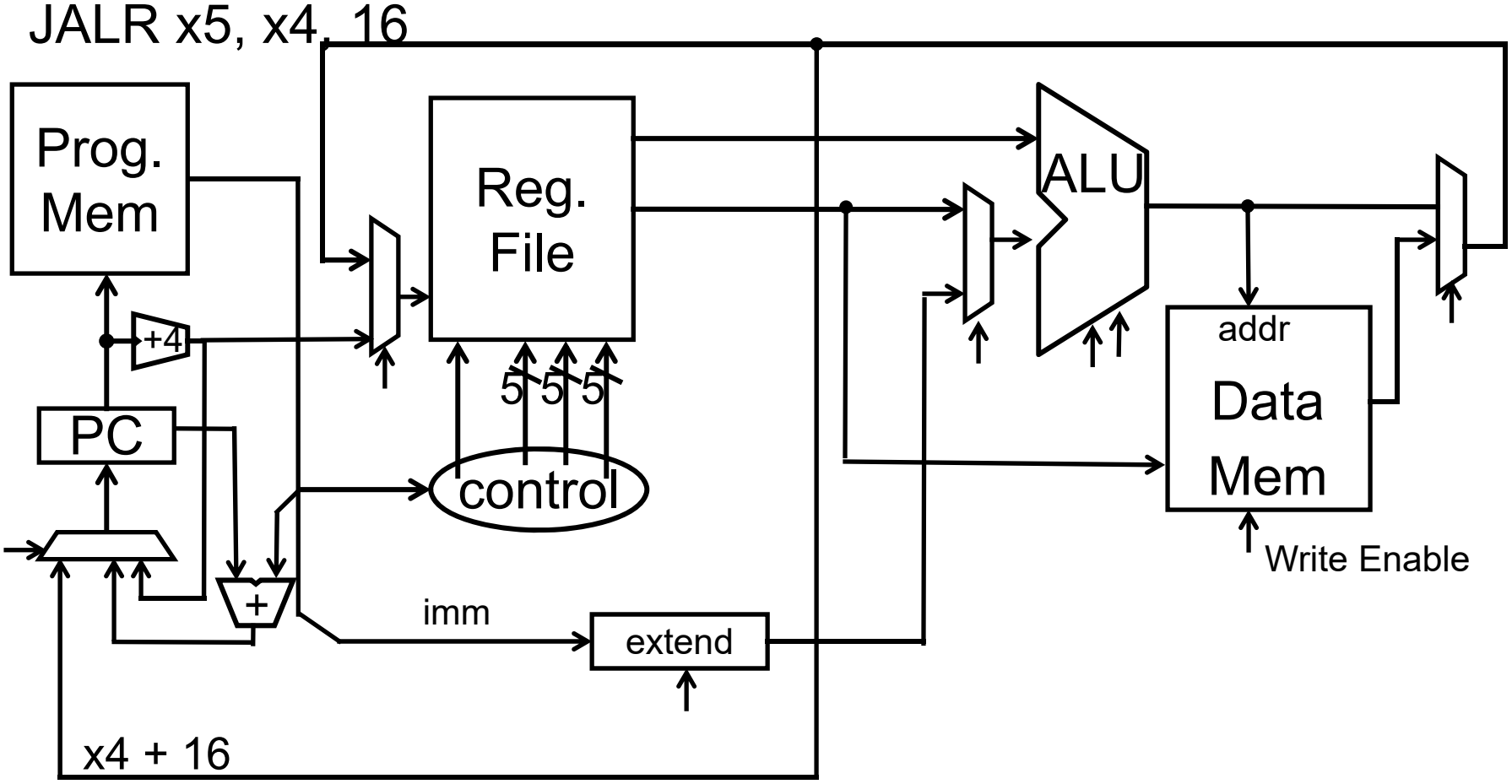
# Jump and Link Register




Example: x5 = PC+4 # JALR x5, x4, 16

PC = x4 + 16

# Jump and Link Register




Example: x5 = PC+4 # JALR x5, x4, 16

PC = x4 + 16

# Moving Beyond Jumps

- Can use Jump and link (JAL) or Jump and Link Register (JALR) instruction to jump to 0xabcd1234

What about a jump based on a condition?

- # assume 0 <= x3 <= 1
- if (x3 == 0) jump to 0xdecafe00
  else jump to 0xabcd1234

# SB-Type (2): Branches

```
00000100000000101000000000010011
31          25 24     20 19     15 14  12 11      7 6           0
 imm    rs2   rs1  funct3 imm    Op
```

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

signed

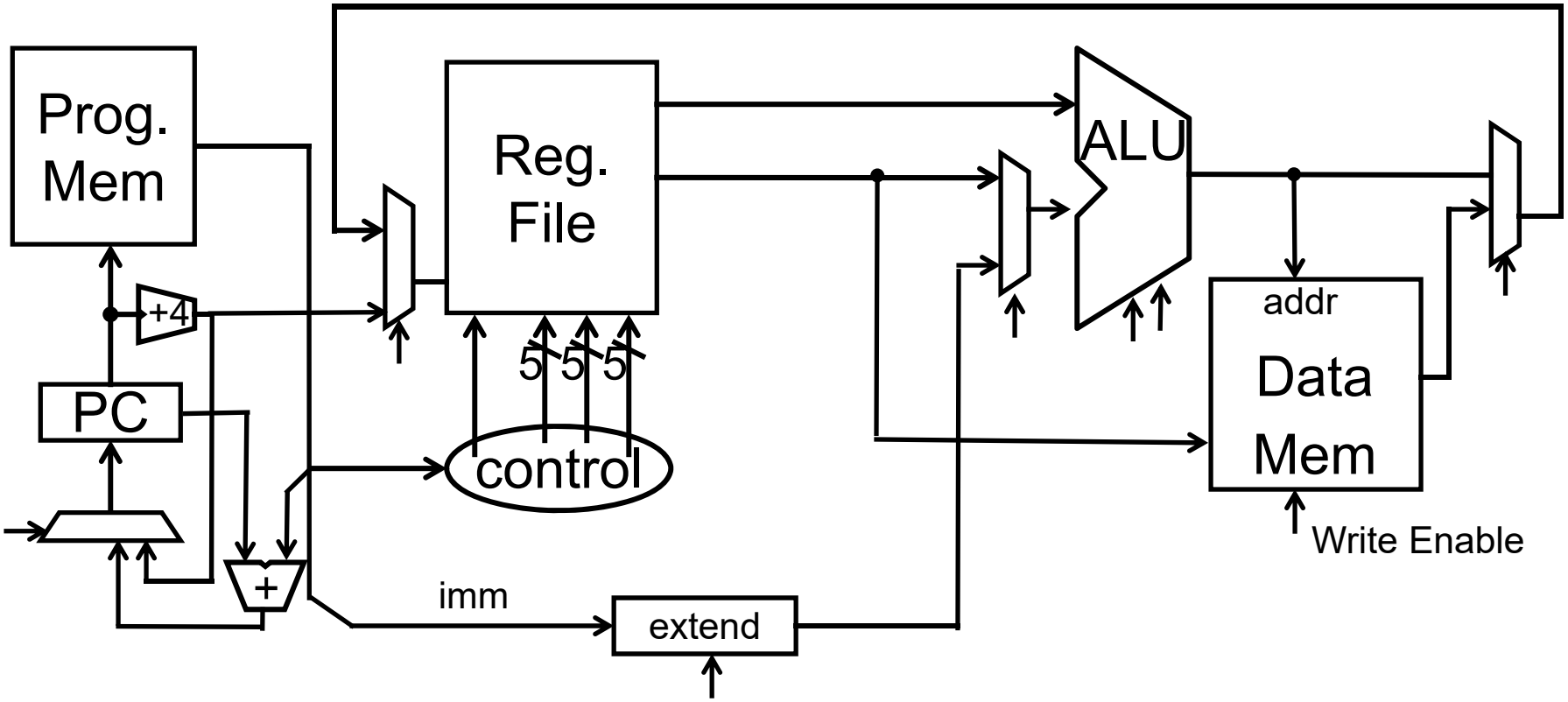
| op | mnemonic | description |
|---|---|---|
| 1100011 | BEQ rs1, rs2, imm | PC=(R[rs1] == R[rs2] ? PC+sext(imm)<<1 : PC+4) |
| 1100011 | BNE rs1, rs2, imm | PC=(R[rs1] != R[rs2] ? PC+sext(imm)<<1 : PC+4) |

Example: BEQ x5, x1, 128

```
if(R[x5]==R[x1])
   PC = PC + 128  (i.e. 128 == 64<<1)
```

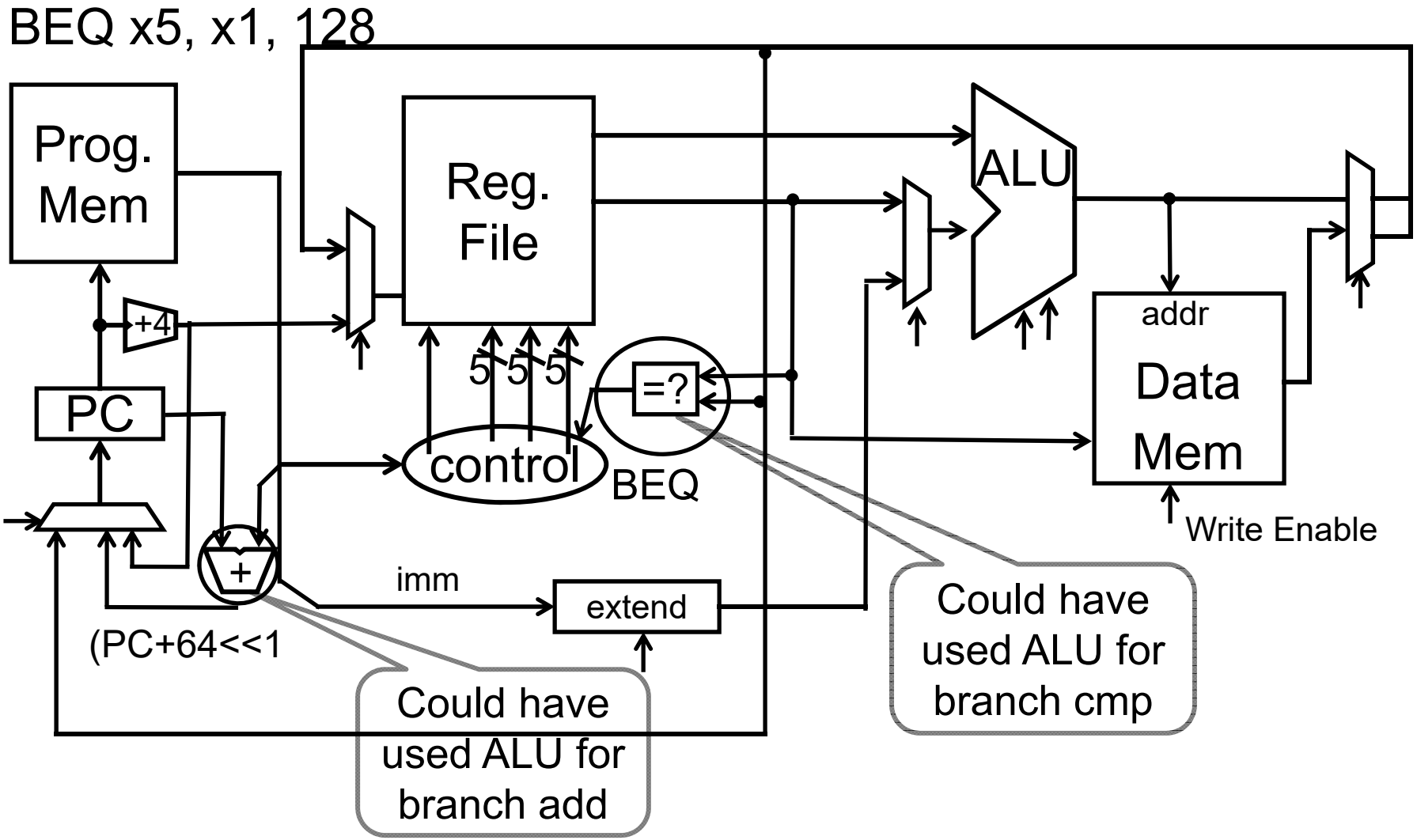
A word about all these +'s…

# Control Flow: Branches




Example: BEQ x5, x1, 128

# Control Flow: Branches




Example: BEQ x5, x1, 128

# SB-Type (3): Conditional Jumps

```
00000000000001010001000000010011
31          25 24    20 19    15 14  12 11     7 6          0
 imm   rs2  rs1 funct3 imm   Op
```

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

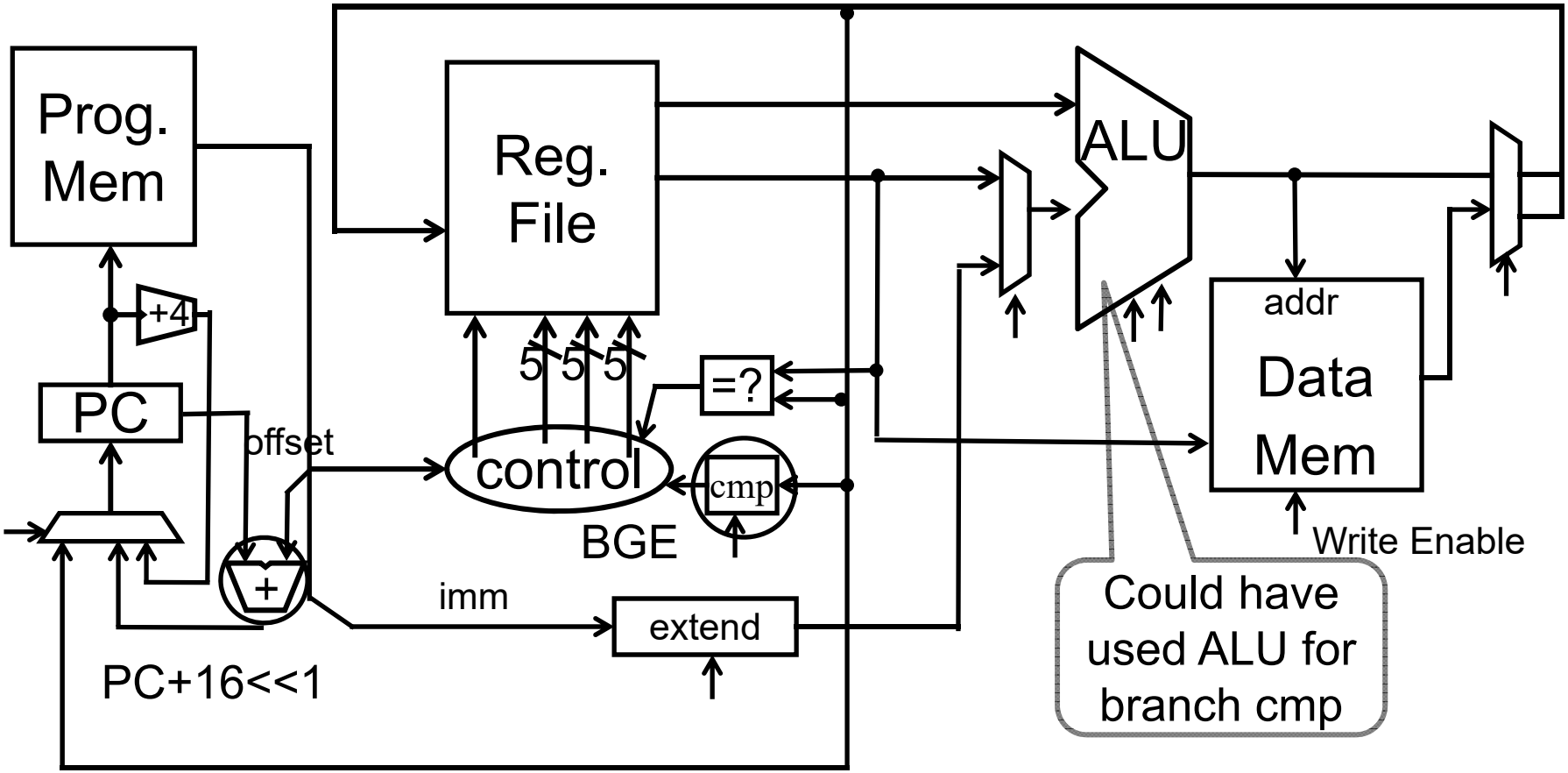
| op | funct3 | mnemonic | description |
|---|---|---|---|
| 1100011 | 100 | BLT rs1, rs2, imm | PC=(R[rs1] <s R[rs2] ? PC + sext(imm)<<1 : PC+4) |
| 1100011 | 101 | BGE rs1, rs2, imm | PC=(R[rs1] >=s R[rs2] ? PC + sext(imm)<<1 : PC+4) |
| 1100011 | 110 | BLTU rs1, rs2 imm | PC=(R[rs1] <u R[rs2] ? PC + sext(imm)<<1 : PC+4) |
| 1100011 | 111 | BGEU rs1, rs2, imm | PC=(R[rs1] >=u R[rs2] ? PC + sext(imm)<<1 : PC+4) |

Example: BGE x5, x0, 32

```
if(R[x5] ≥s R[x0])
  PC = PC + 32  (i.e. 32 == 16<<1)
```

# Control Flow: More Branches

BGE x5, x0, 32




Example: BGE x5, x0, 32

# RISC-V Instruction Types

- Arithmetic/Logical
  - R-type: result and two source registers, shift amount
  - I-type: result and source register, shift amount in 16-bit immediate with sign/zero extension
  - U-type: result register, 16-bit immediate with sign/zero extension

✓

- Memory Access
  - I-type for loads and S-type for stores
  - load/store between registers and memory
  - word, half-word and byte operations

✓

- Control flow
  - U-type: jump-and-link
  - I-type: jump-and-link register
  - S-type: conditional branches: pc-relative addresses

✓

# iClicker Question

What RISC-V instruction would you use for a:

1. For loop?
2. While loop?
3. Function call?
4. If statement?
5. Return statement?

(A)Jump and Link Register (`JALR lr, x2, 0x000FFFF`)

(B) Branch Equals (`BEQ x1, x2, 0xAAAA`)

(C) Branch Less Than (`BLT x1, x2, 0xAAAA`)

(D)Jump and Link (`JAL lr, 0x000FFFF`)

# iClicker Question

- What is the one topic you're most uncertain about at this point in the class?

(A) Gates & Logic
(B) Circuit Simplification
(C) Finite State Machines
(D) RISC-V Processor
(E) RISC-V Assembly

# Summary

We have all that it takes to build a processor!

- Arithmetic Logic Unit (ALU)
- Register File
- Memory

RISC-V processor and ISA is an example of a Reduced Instruction Set Computers (RISC)

- Simplicity is key, thus enabling us to build it!

We now know the data path for the MIPS ISA:

- register, memory and control instructions