The RISC-V Processor

Hakim Weatherspoon
CS 3410
Computer Science
Cornell University

[Weatherspoon, Bala, Bracy, and Sirer]

Announcements

- Make sure to go to ***your*** Lab Section this week
- Completed **Proj1** due Friday, Feb 15th
- Note, a Design Document is due when you submit Proj1 final circuit
- Work **alone**

BUT use your resources

- Lab Section, Piazza.com, Office Hours
- Class notes, book, Sections, CSUGLab

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***
- Project and Reading Assignments
- **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 Black Lightening, then write up solution independently
- Do not copy solutions

Late Policy

- Each person has a total of ***five*** "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

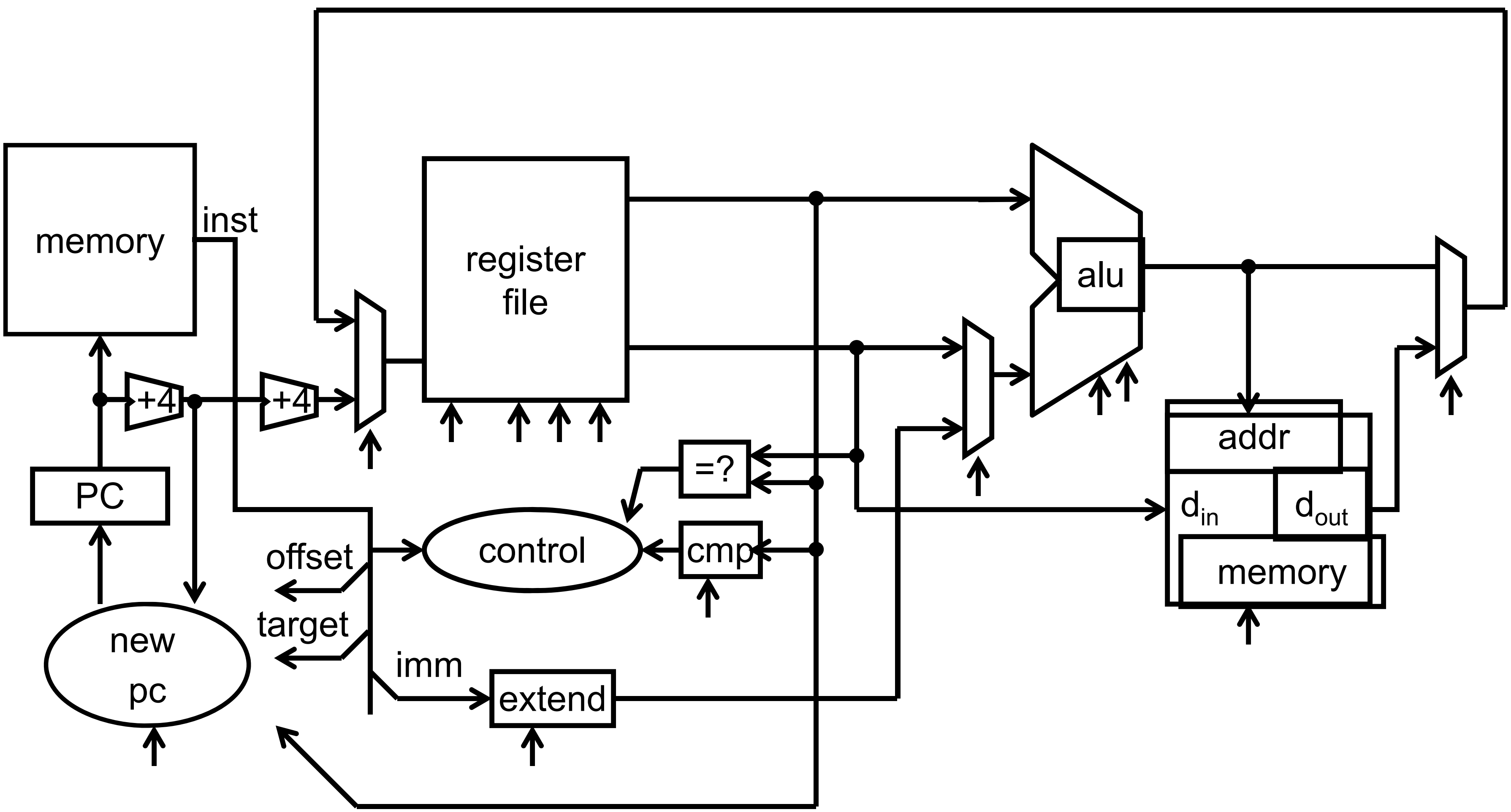
Regrade policy

- Submit written request within a week of receiving score

Announcements

- Level Up (optional enrichment)
  - Teaches CS students tools and skills needed in their coursework as well as their career, such as Git, Bash Programming, study strategies, ethics in CS, and even applying to graduate school.
  - Thursdays at 7-8pm in 310 Gates Hall, starting this week
  - http://www.cs.cornell.edu/courses/cs3110/2019sp/levelup/

Big Picture: Building a Processor



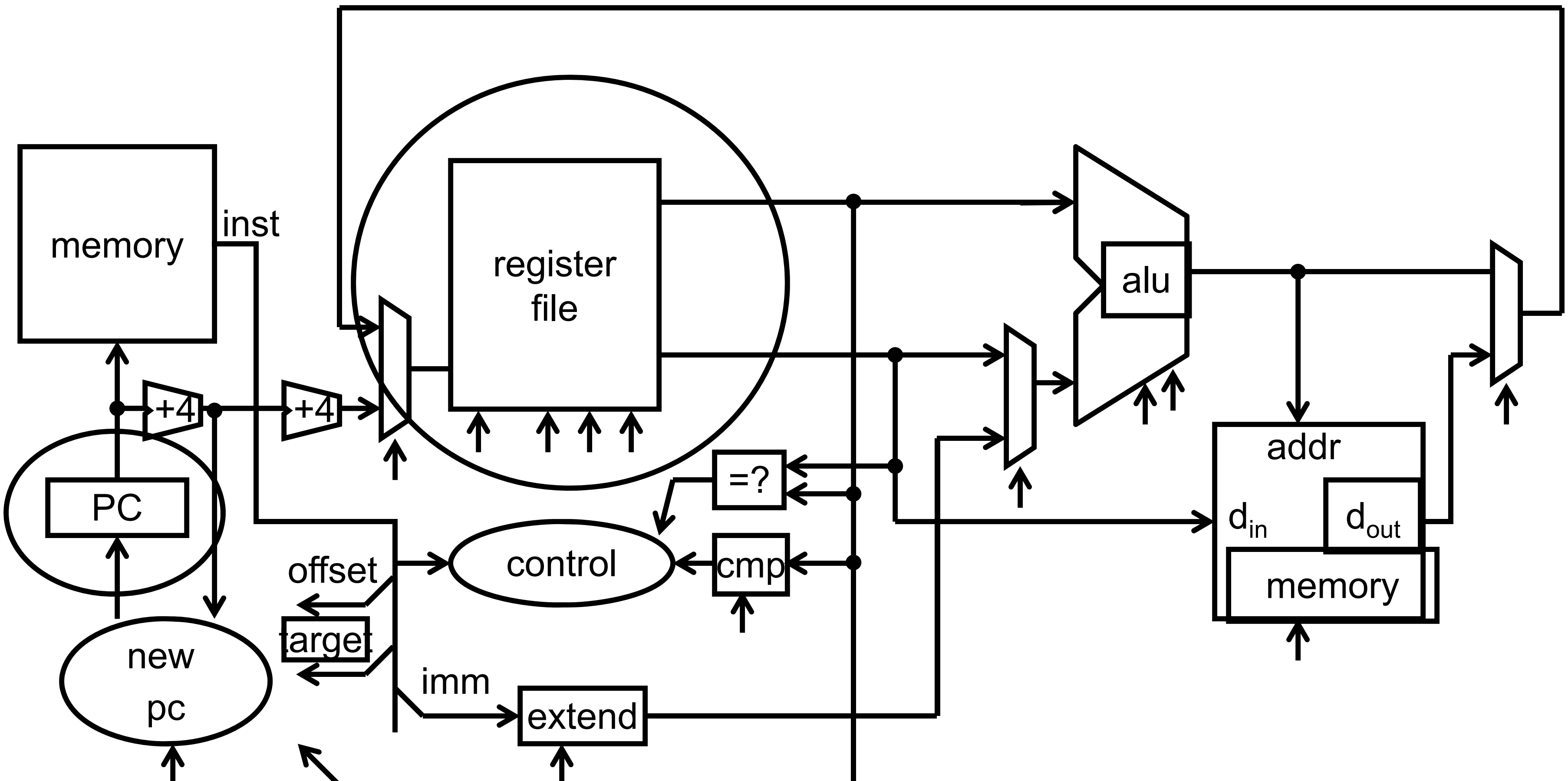
A single cycle processor

Goal for the next few 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

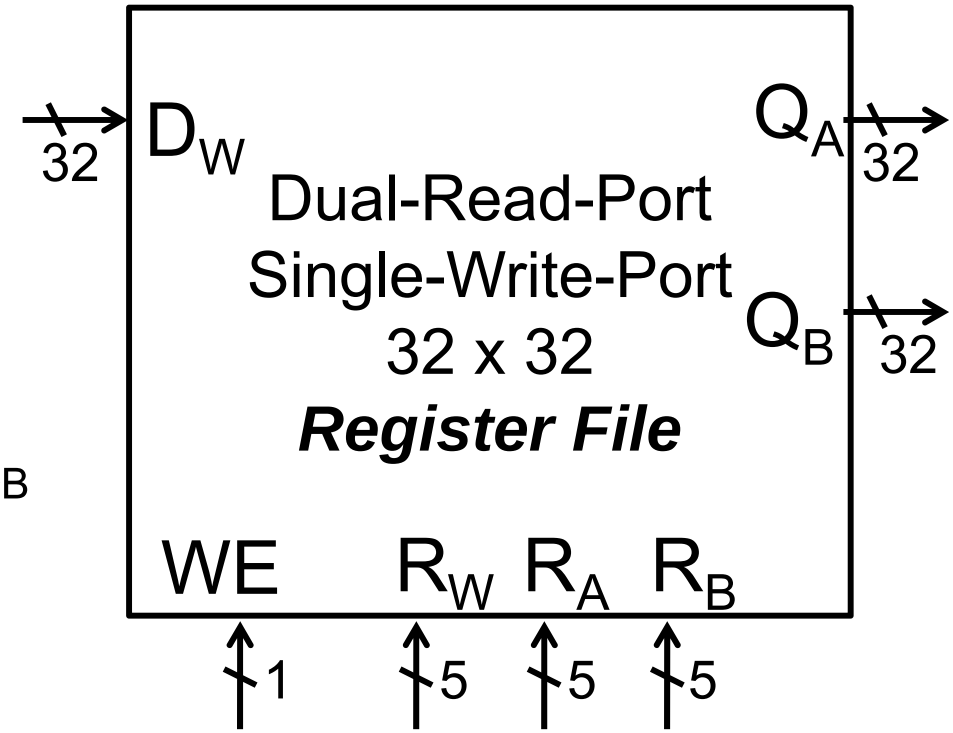
RISC-V Register File



A single cycle processor

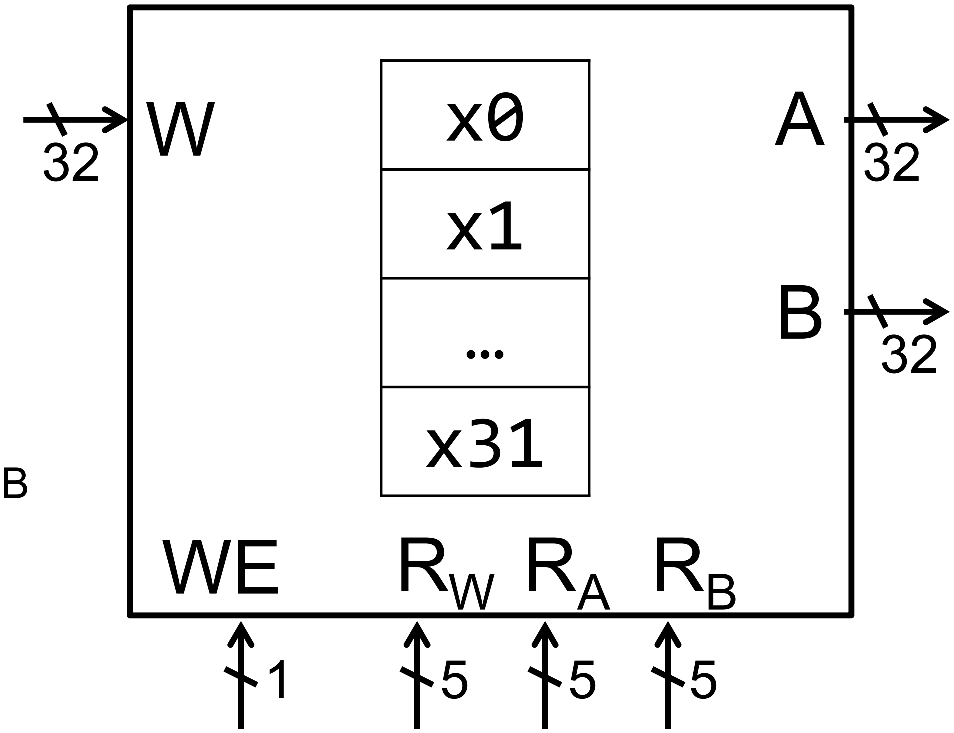
RISC-V Register File

- 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



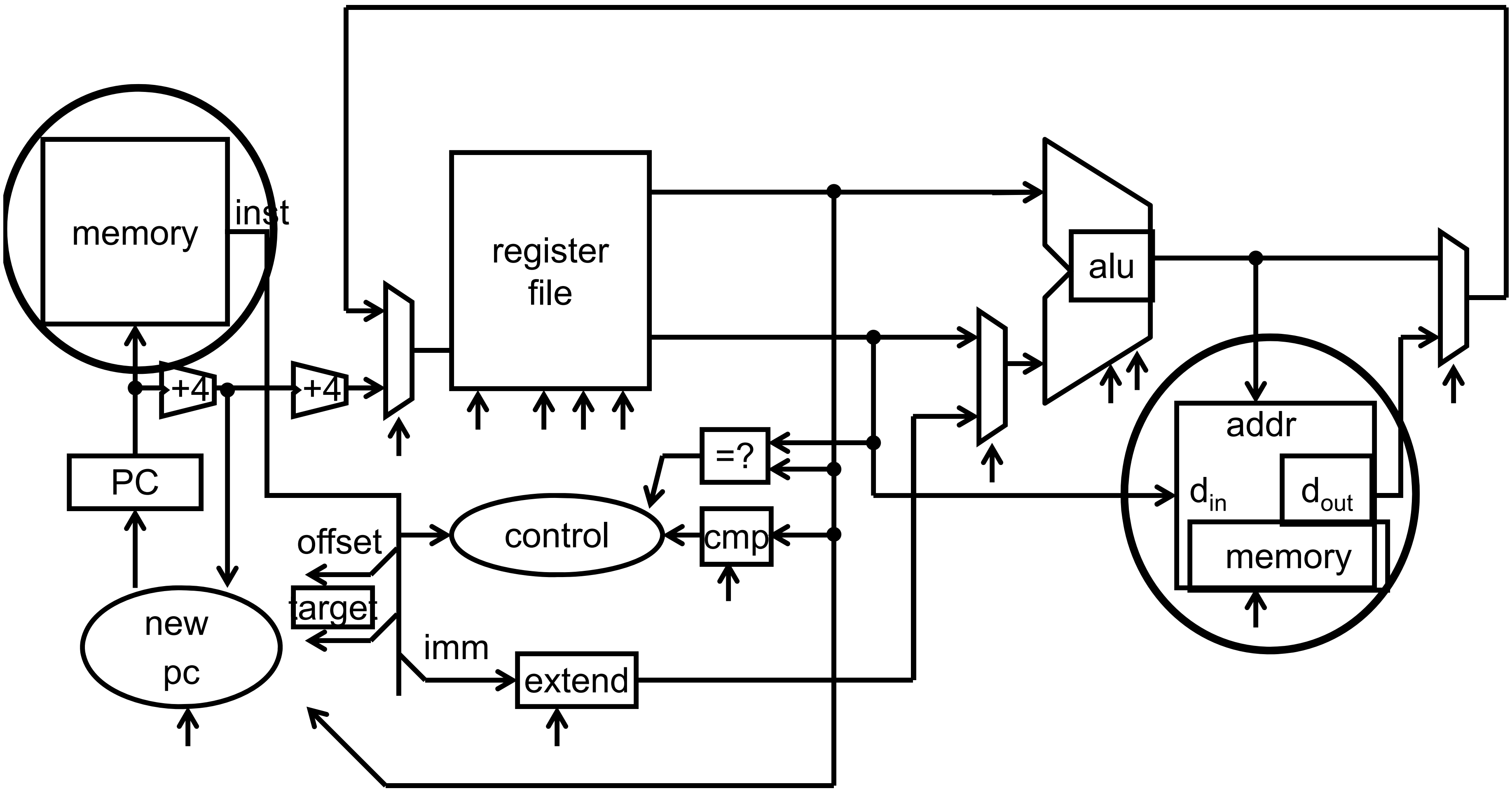
RISC-V Register File

- 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



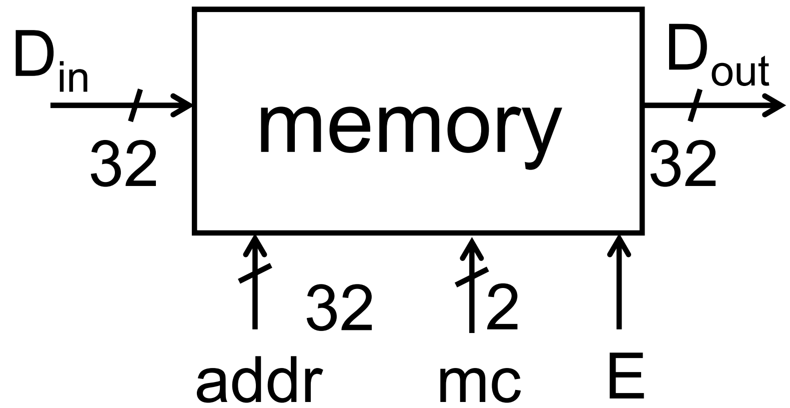
- RISC-V register file
  - Numbered from 0 to 31
  - Can be referred by number: x0, x1, x2, … x31
  - Convention, each register also has a name:
    - x10 – x17 → a0 – a7, x28 – x31 → t3 – t6

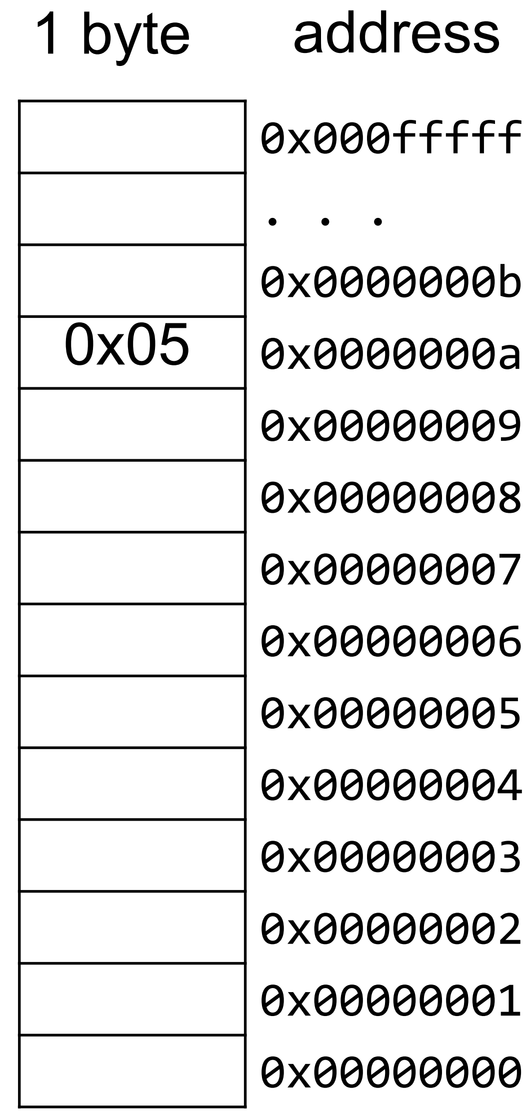
RISC-V Memory



A single cycle processor

RISC-V Memory

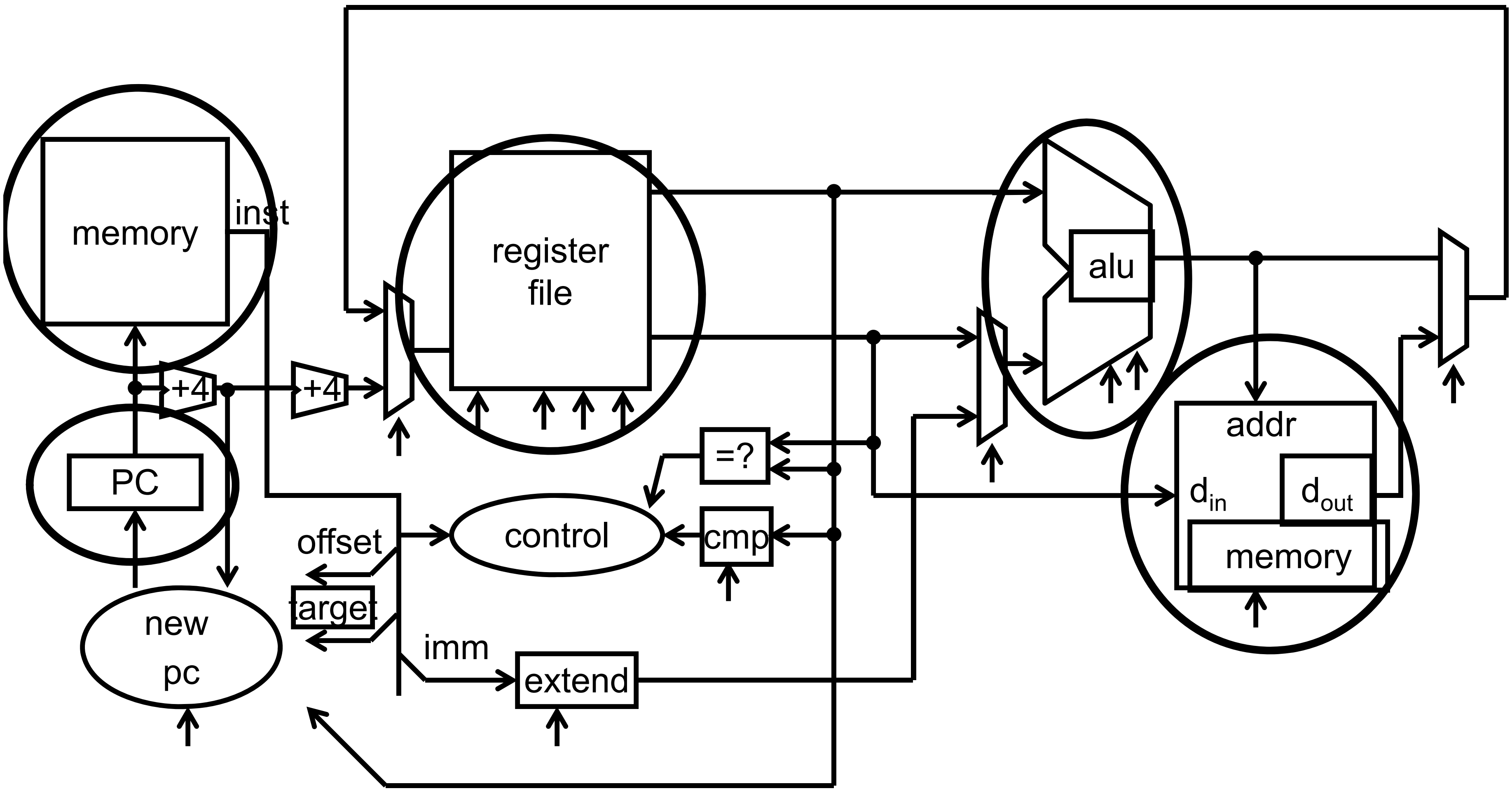




- 32-bit address
- 32-bit data (but byte addressed)
- Enable + 2 bit memory control (mc)

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

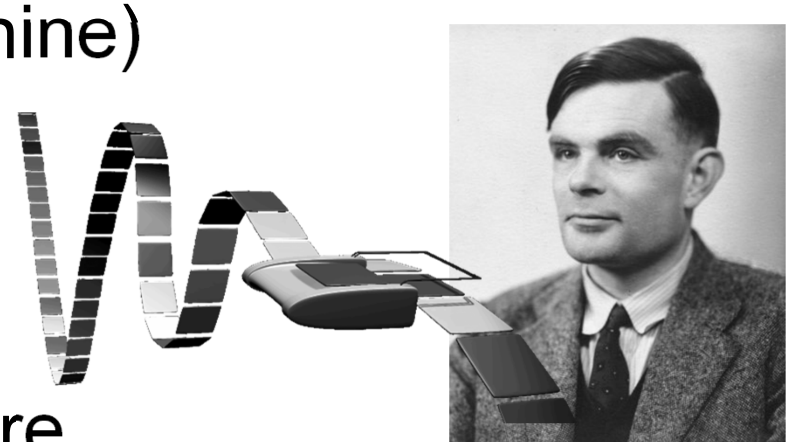
Architectures

- von Neumann architecture
- Harvard (modified) architecture

To make a computer

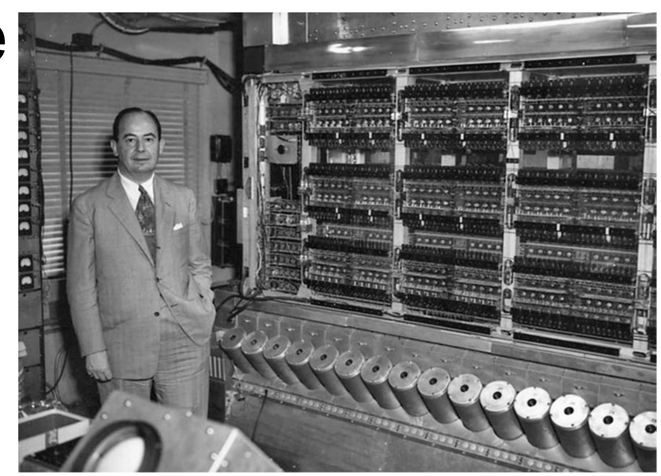
Need a program

- Stored program computer
- (a Universal Turing Machine)



Architectures

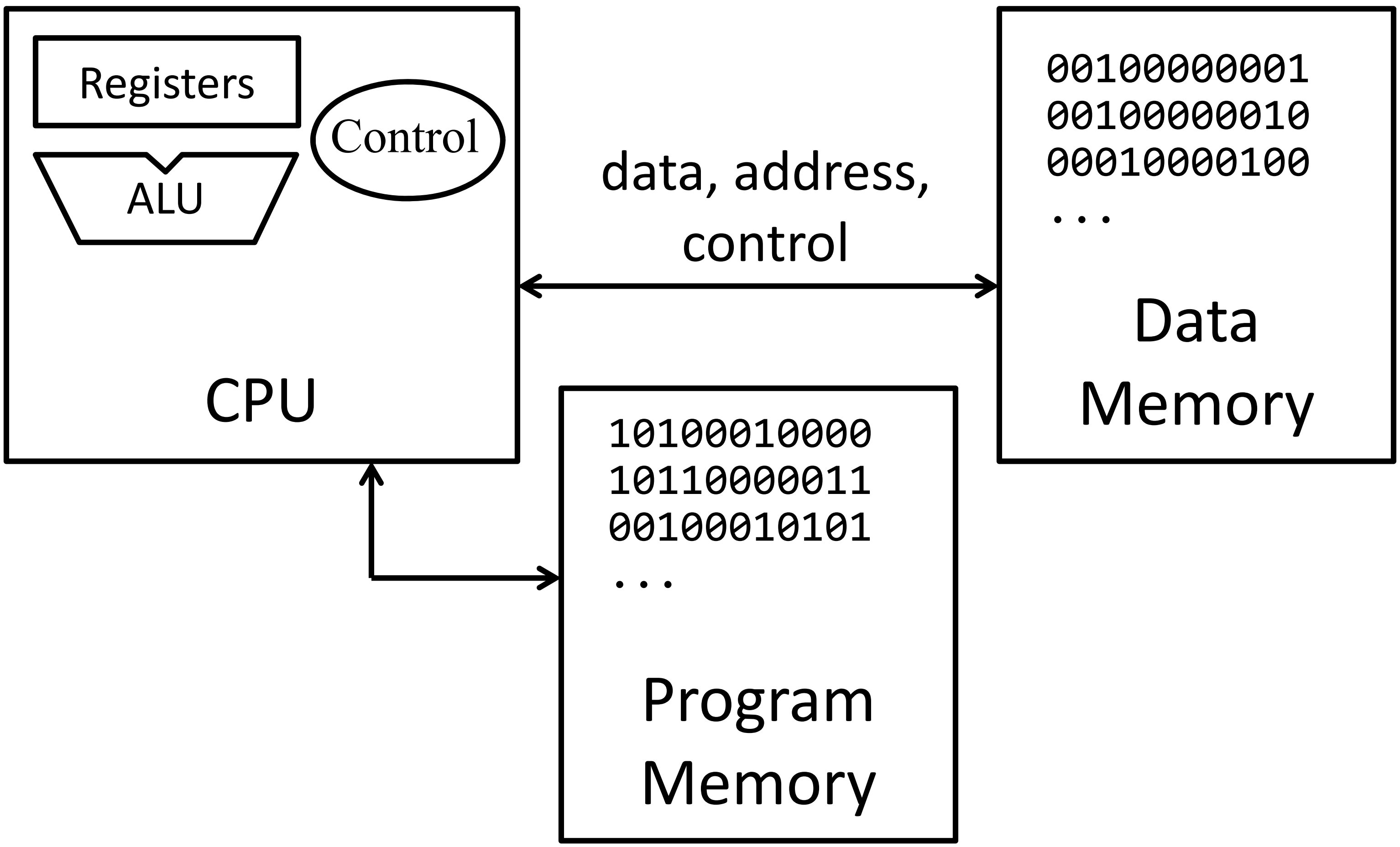
- von Neumann architecture
- Harvard (modified) architecture



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

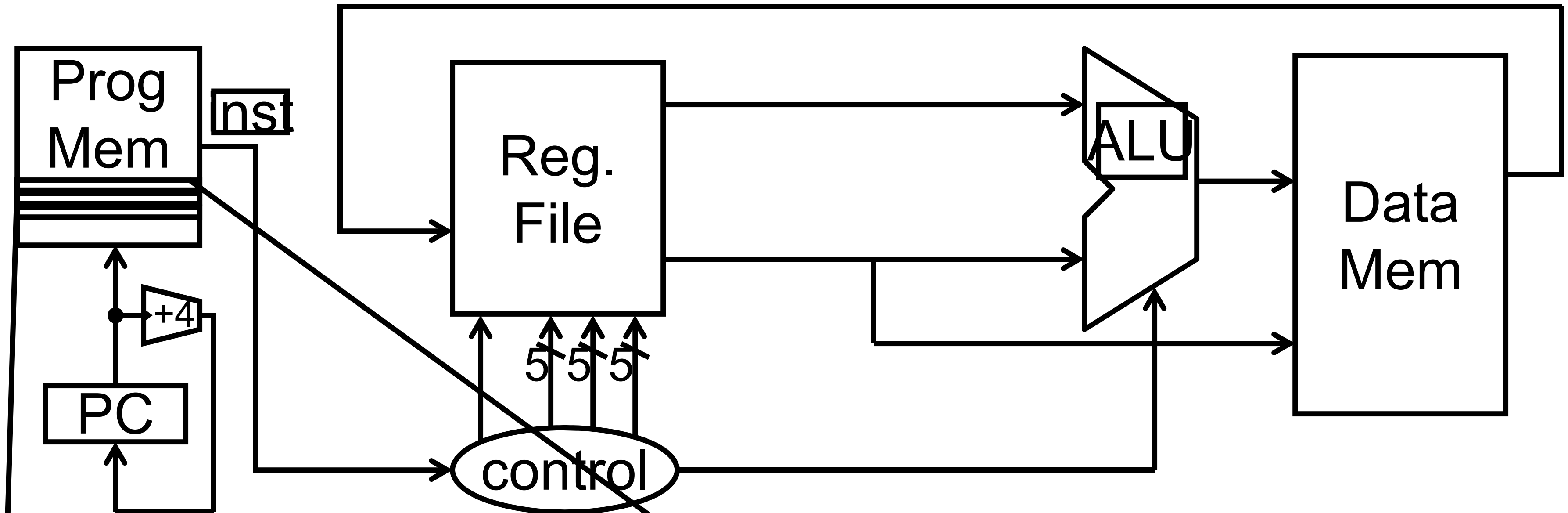
A bus connects the two

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 Processing



Instructions:

stored in memory, encoded in binary

00100000000000010000000000001010
00100000000000010000000000000000
00000000001000100001100000101010

A basic processor
- fetches
- decodes
- executes

one instruction at a time

Levels of Interpretation: Instructions

```
for (i = 0; i < 10; i++)
    printf("go cucs");
```

High Level Language

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

```
main:  addi x2, x0, 10
       addi x1, x0, 0
loop:  slt x3, x1, x2
       ...
```

Assembly Language

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

```
10                x2          x0   op=addi
0000000 01010 00010 000 00000 0010011
00100000000000010000000000010000
00000000001000100001100000101010
```

Machine Language

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

Instruction Set Architecture

ALU, Control, Register File, …

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

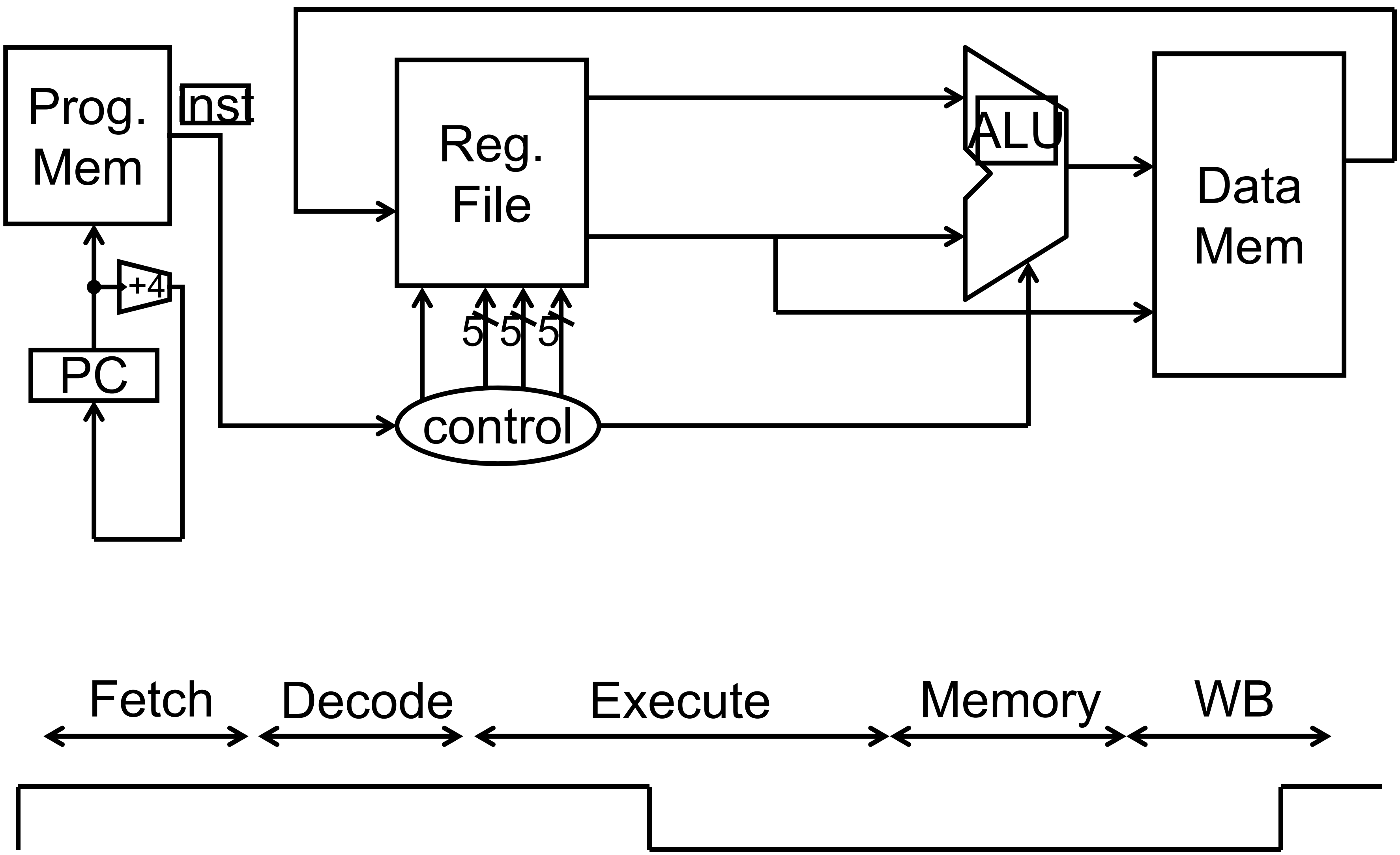
Takeaway

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



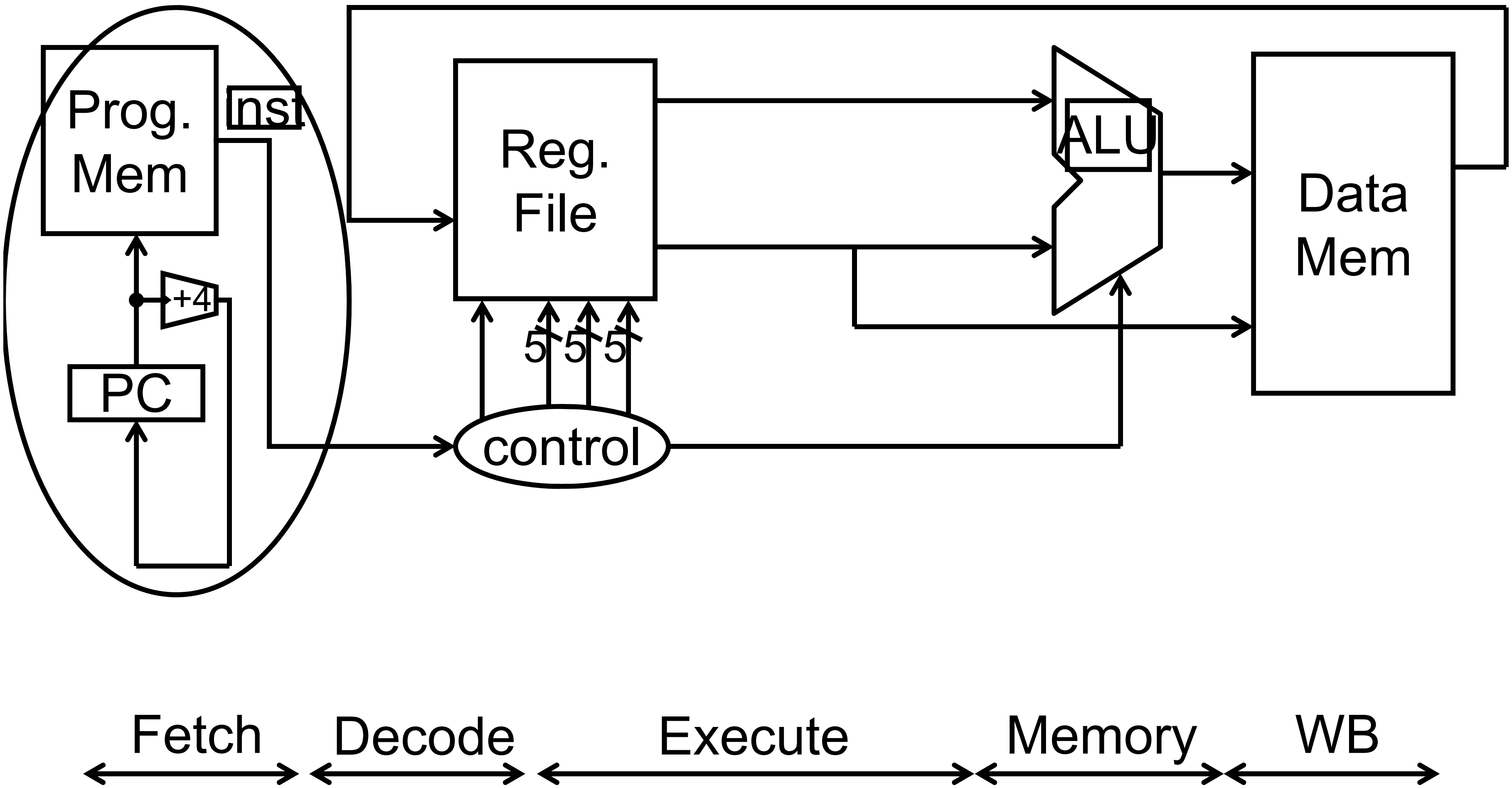
A single cycle processor – this diagram is not 100% spatial

Five Stages of RISC-V Datapath

Basic CPU execution loop

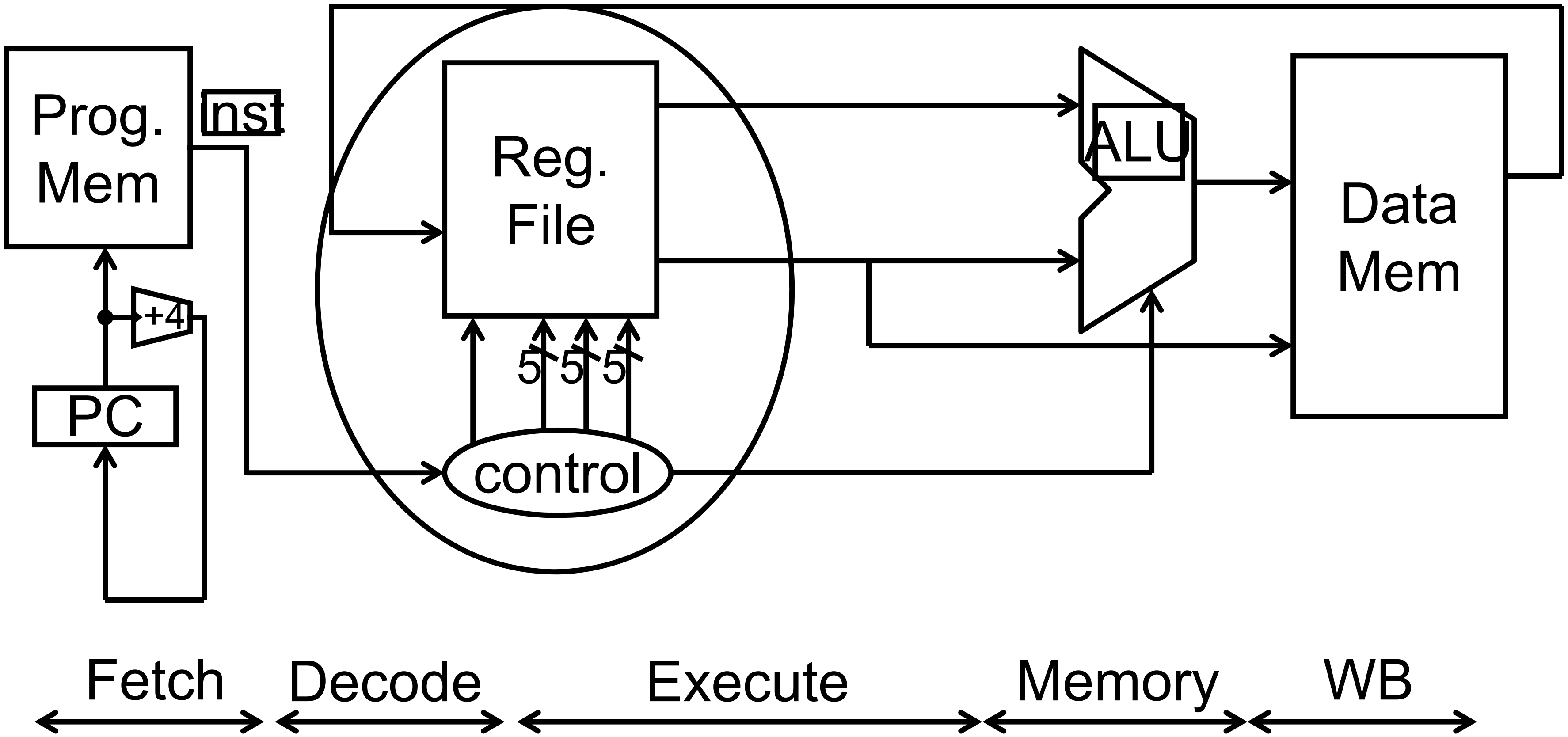
1. Instruction Fetch
2. Instruction Decode
3. Execution (ALU)
4. Memory Access
5. Register Writeback

Stage 1: Instruction Fetch



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

Stage 2: Instruction Decode



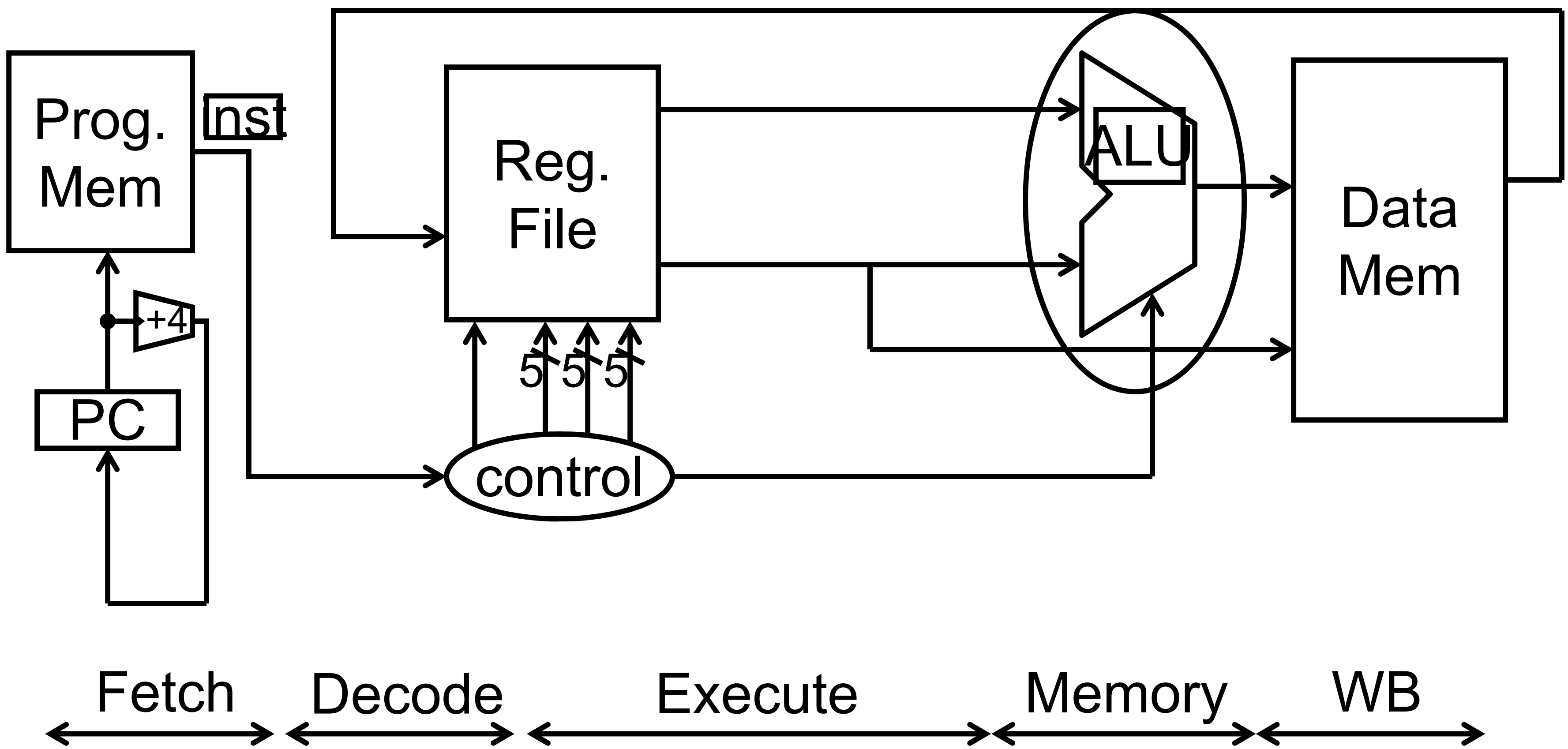
Gather data from the instruction

Read opcode; determine instruction type, field lengths

Read in data from register file

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

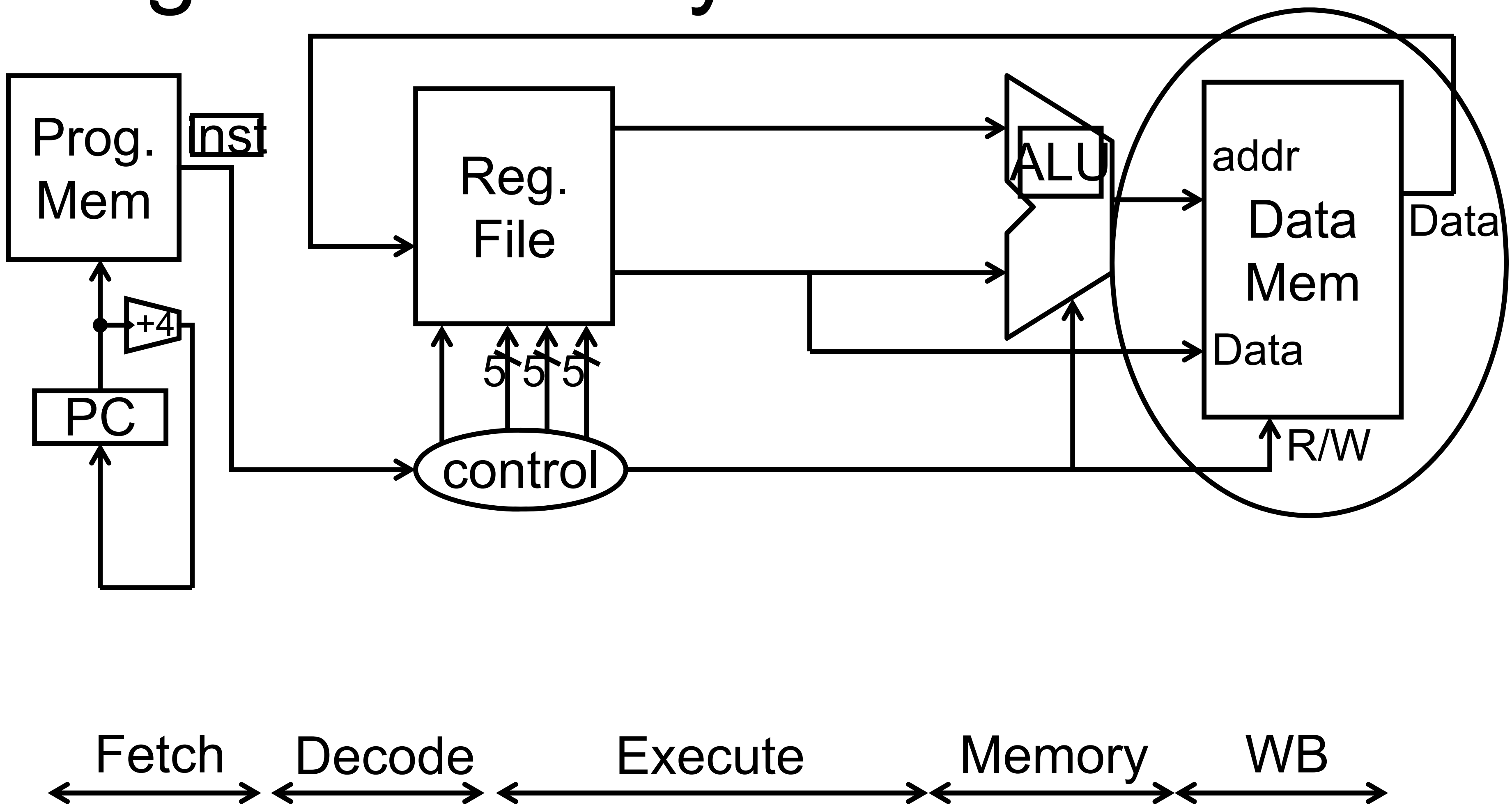
Stage 3: Execution (ALU)



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

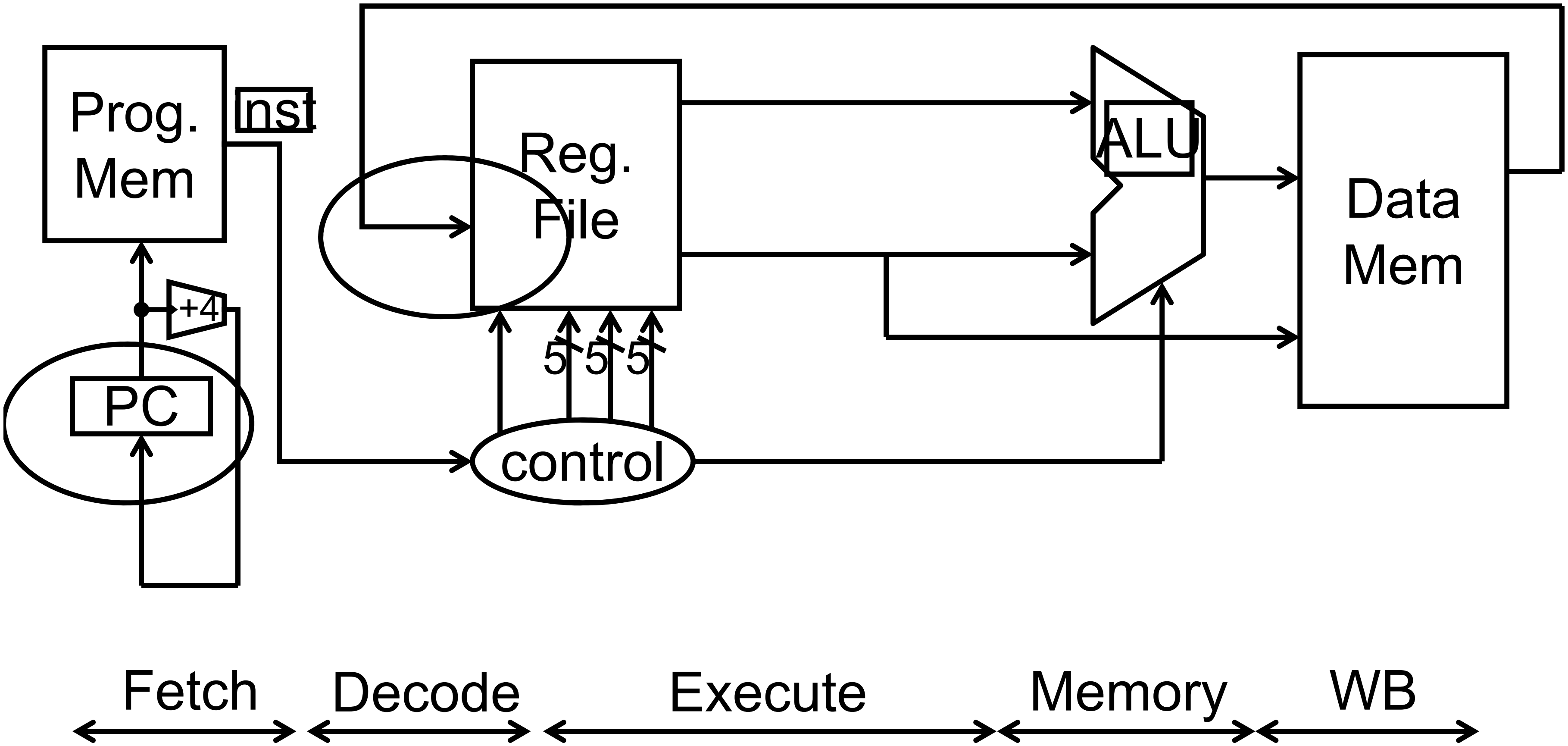
Load/Store? lw x2, x3, 32 → Compute address

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

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

RISC-V instruction formats

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

- R-type

funct7	rs2	rs1	funct3	rd	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- I-type

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

- S-type

imm	rs2	rs1	funct3	imm	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- U-type

imm	rd	op
20 bits	5 bits	7 bits

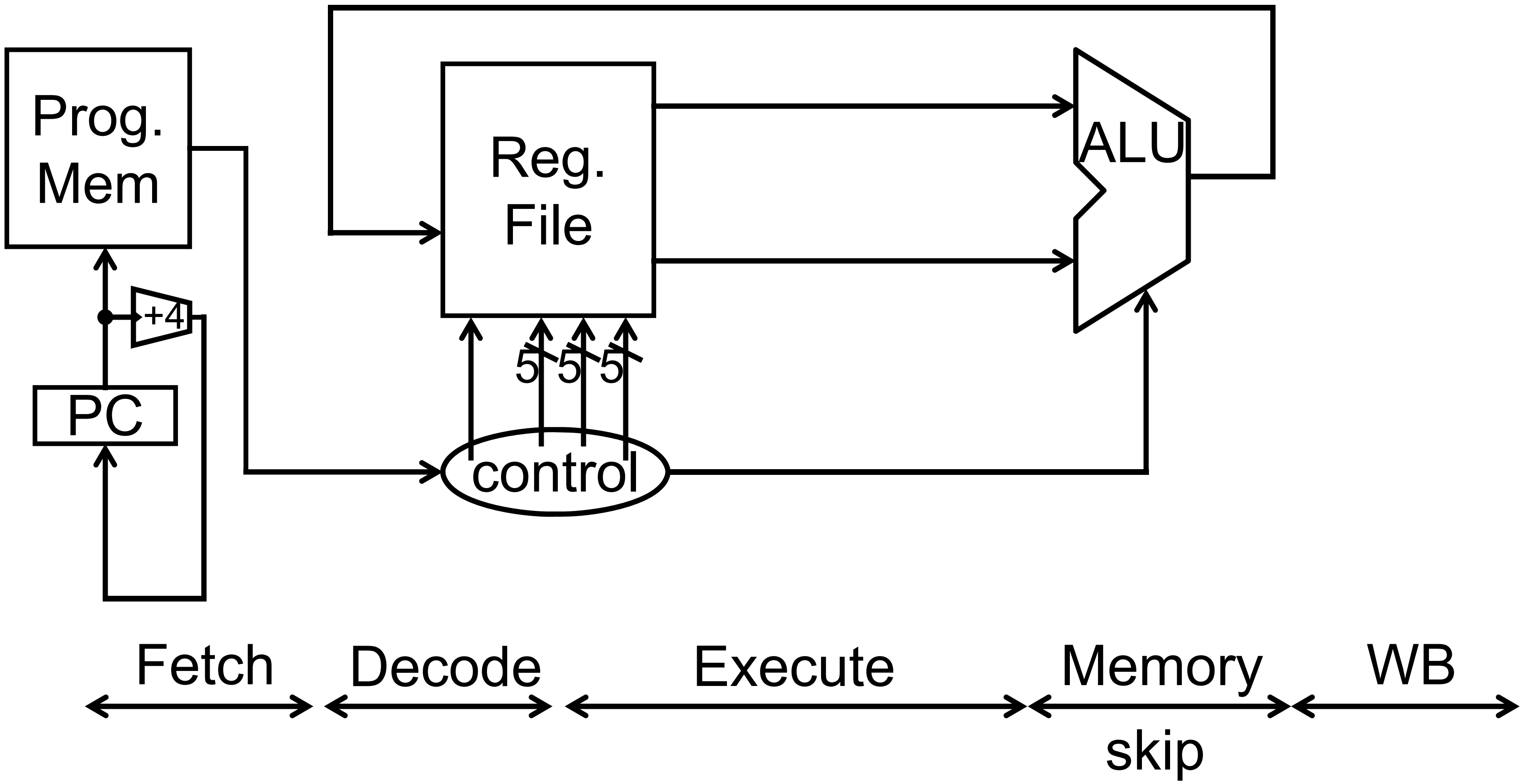
R-Type (1): Arithmetic and Logic

`00000000011001000100001000110011`

funct7	$\boxed{\text{rs2}}$	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]

Arithmetic and Logic

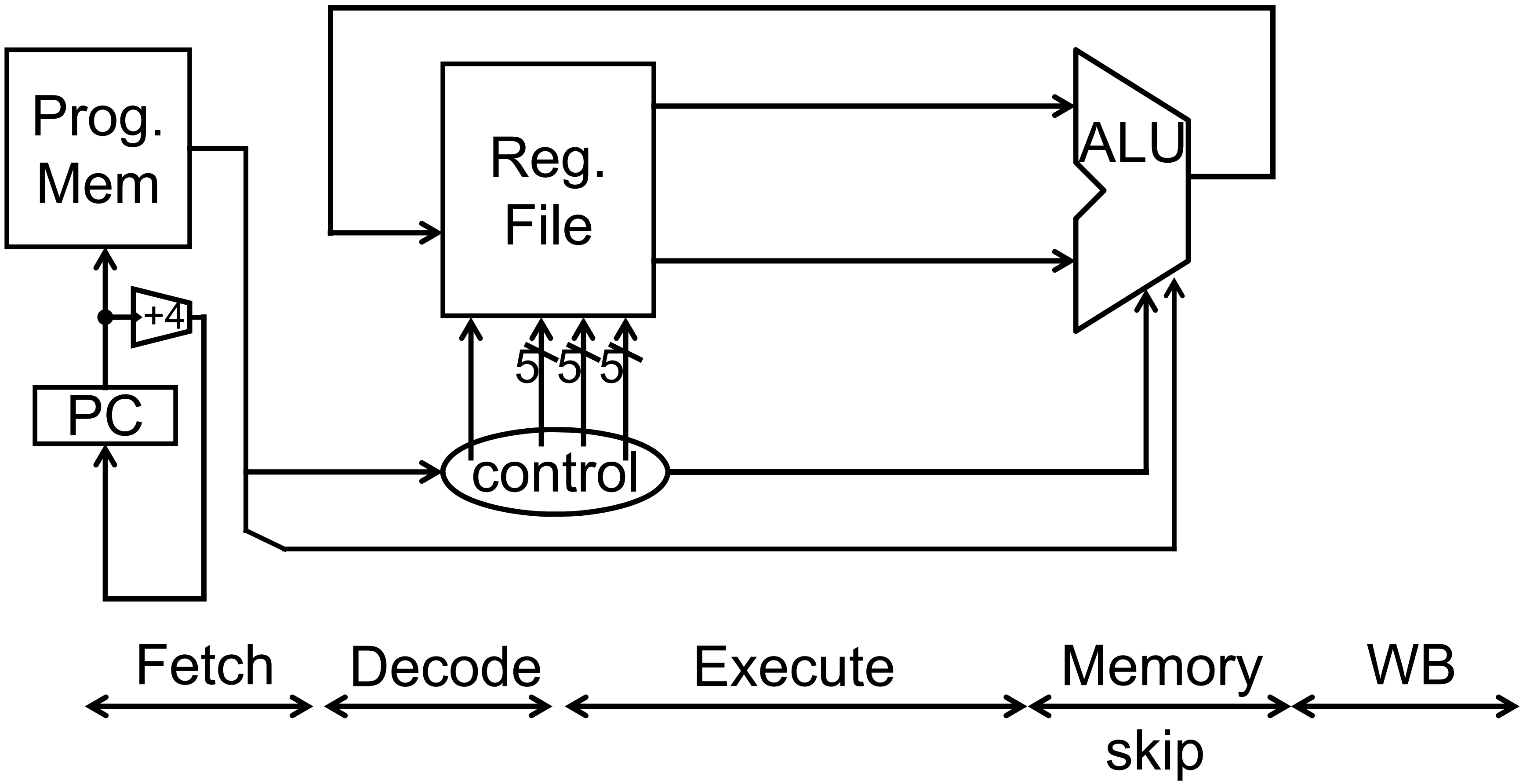


R-Type (2): Shift Instructions

0000000	00011	00010	000	01010	0011011
funct7	$\boxed{\text{rs2}}$	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.)

Shift


Prog.
Mem
+4
PC
Reg.
File
5
5
5
control
ALU
Fetch
Decode
Execute
Memory
skip
WB

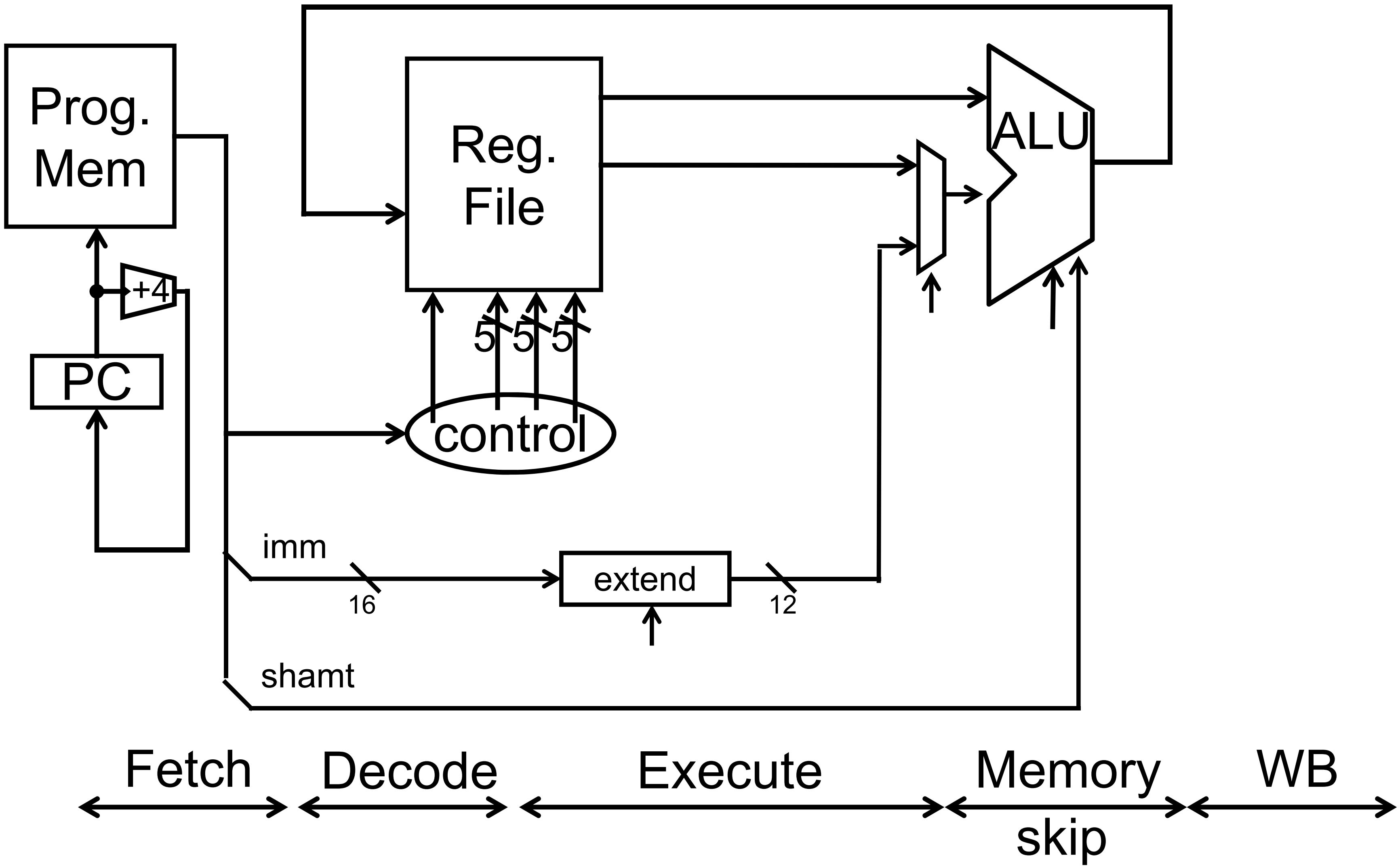
I-Type (1): Arithmetic w/ immediates

```
 00000000010100101000001010010011
     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] + imm
0010011	111	ANDI rd, rs1, imm	R[rd] = R[rs1] & zero_extend(imm)
0010011	110	ORI rd, rs1, imm	R[rd] = R[rs1] | zero_extend(imm)

Arithmetic w/ immediates



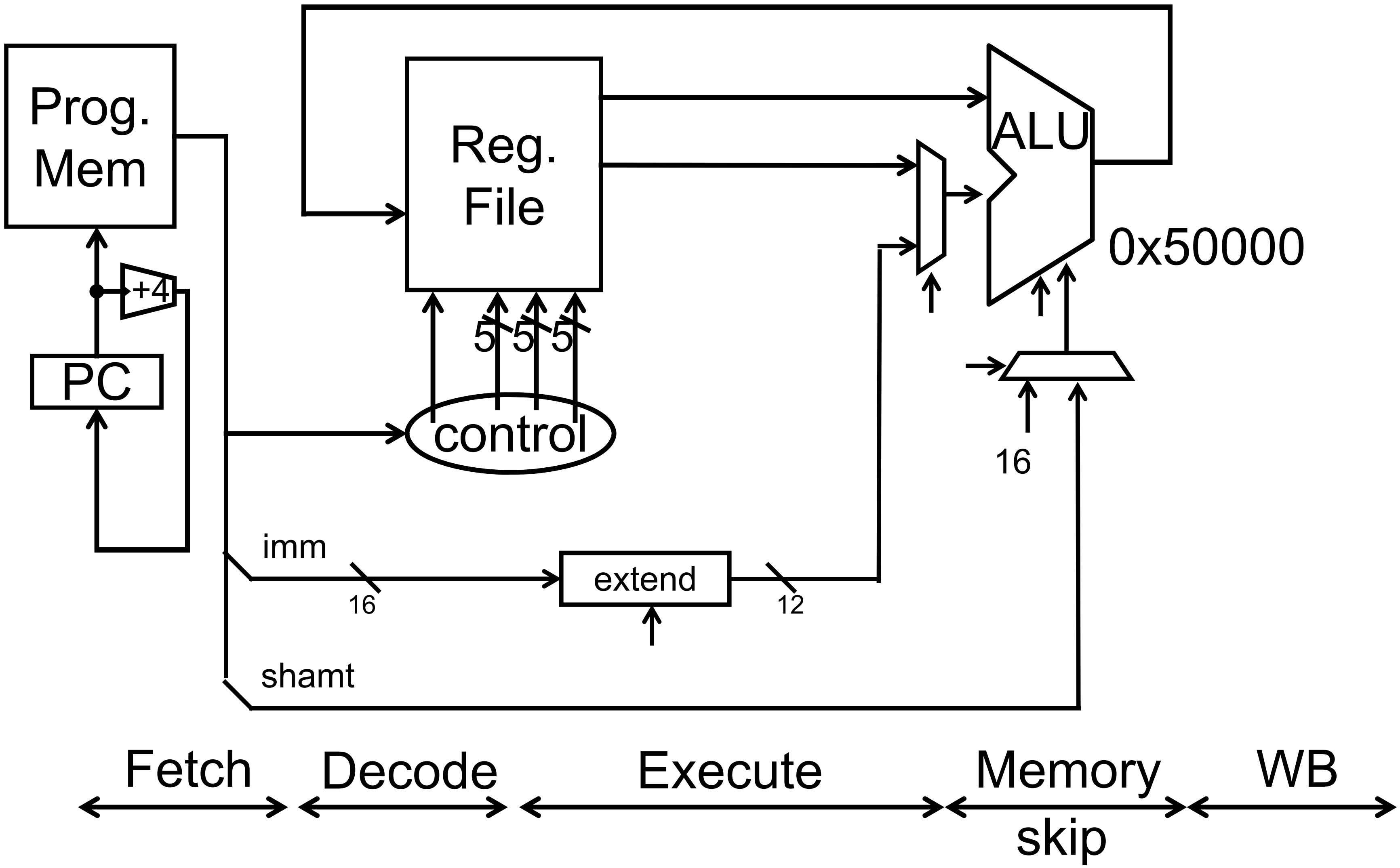
U-Type (1): “Load” Upper Immediate

```
000000000000000001010010110110111
```

imm	rd	op
20 bits	5 bits	7 bits

op	**mnemonic**	**description**
0110111	LUI rd, imm	R[rd] = imm << 16

Load Upper Immediate



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

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

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