

Processor

Prof. Hakim Weatherspoon

CS 3410, Spring 2015

Computer Science

Cornell University

See P&H Chapter: 2.16-2.20, 4.1-4.4, Appendix B

Announcements

Project Partner finding assignment on CMS

Office hours over break

Announcements

Make sure to go to **your** Lab Section this week

Lab2 due in class this week (it is **not** homework)

Lab1: Completed Lab1 due **this** Friday, Feb 13th, **before** winter break

Note, a **Design Document** is due when you submit Lab1 final circuit

Work **alone**

Save your work!

- **Save often.** Verify file is non-zero. Periodically save to Dropbox, email.
- Beware of MacOSX 10.5 (leopard) and 10.6 (snow-leopard)

Homework1 is out

Due a week before prelim1, Monday, February 23rd

Work on problems incrementally, as we cover them in lecture (i.e. part 1)

Office Hours for help

Work **alone**

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/CS3410/2015sp/schedule.html>
- Slides and Reading for lectures
- Office Hours
- ***Pictures of all TAs***
- Homework and Programming Assignments
- **Dates to keep in Mind**
 - **Prelims: Tue Mar 3rd and Thur April 30th**
 - ***Lab 1: Due this Friday, Feb 13th before Winter break***
 - Proj2: Due Thur Mar 26th before Spring break
 - Final Project: Due when final would be (not known until Feb 14t

Schedule is subject to change

Collaboration, Late, Re-grading Policies

“Black Board” Collaboration Policy

- Can discuss approach together on a “black board”
- Leave and write up solution independently
- Do not copy solutions

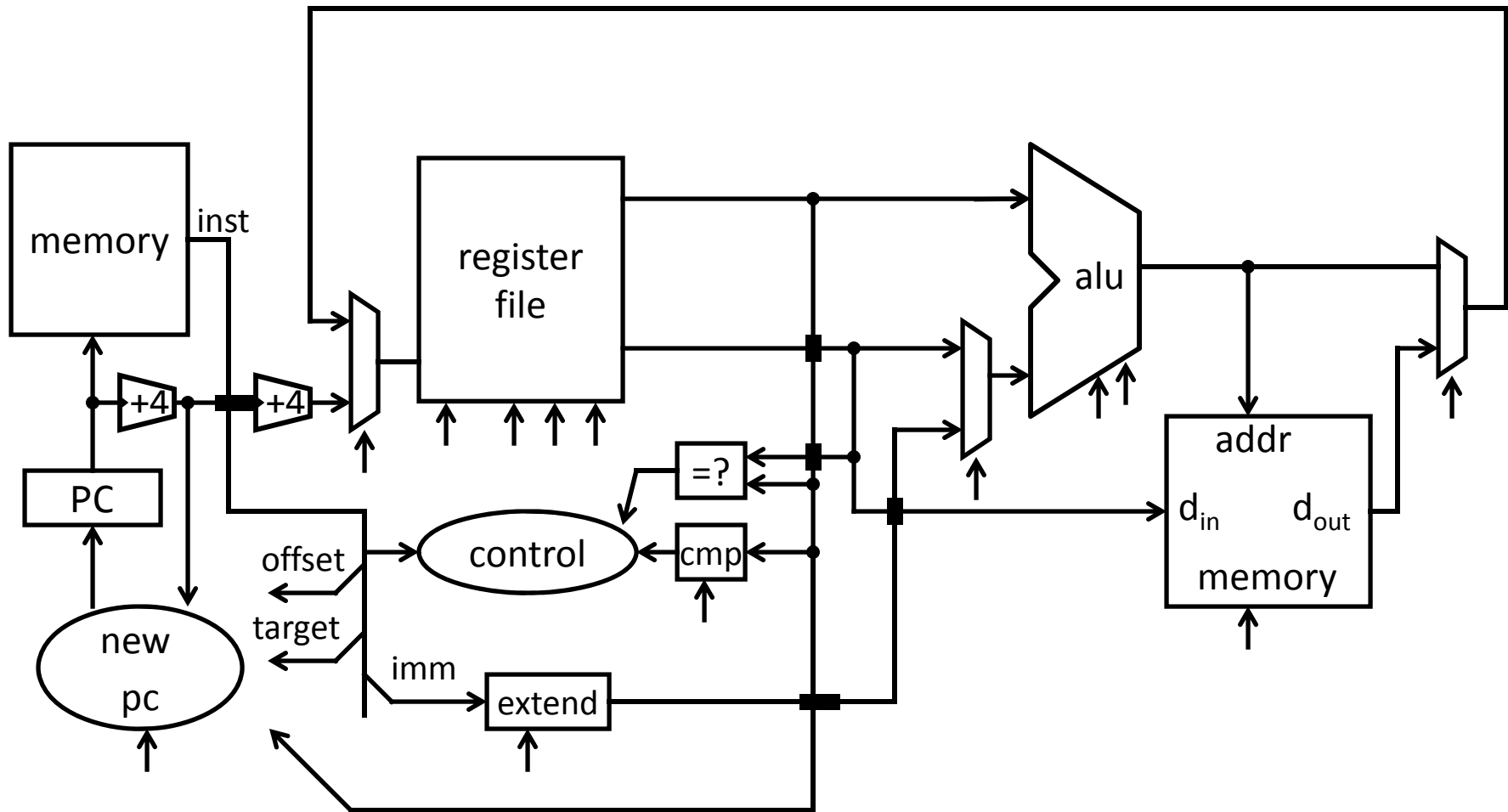
Late Policy

- 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 to lead TA,
and lead TA will pick a different grader
- Submit another written request,
lead TA will regrade directly
- Submit yet another written request for professor to regrade.

Big Picture: Building a Processor



A Single cycle processor

Goal for Today

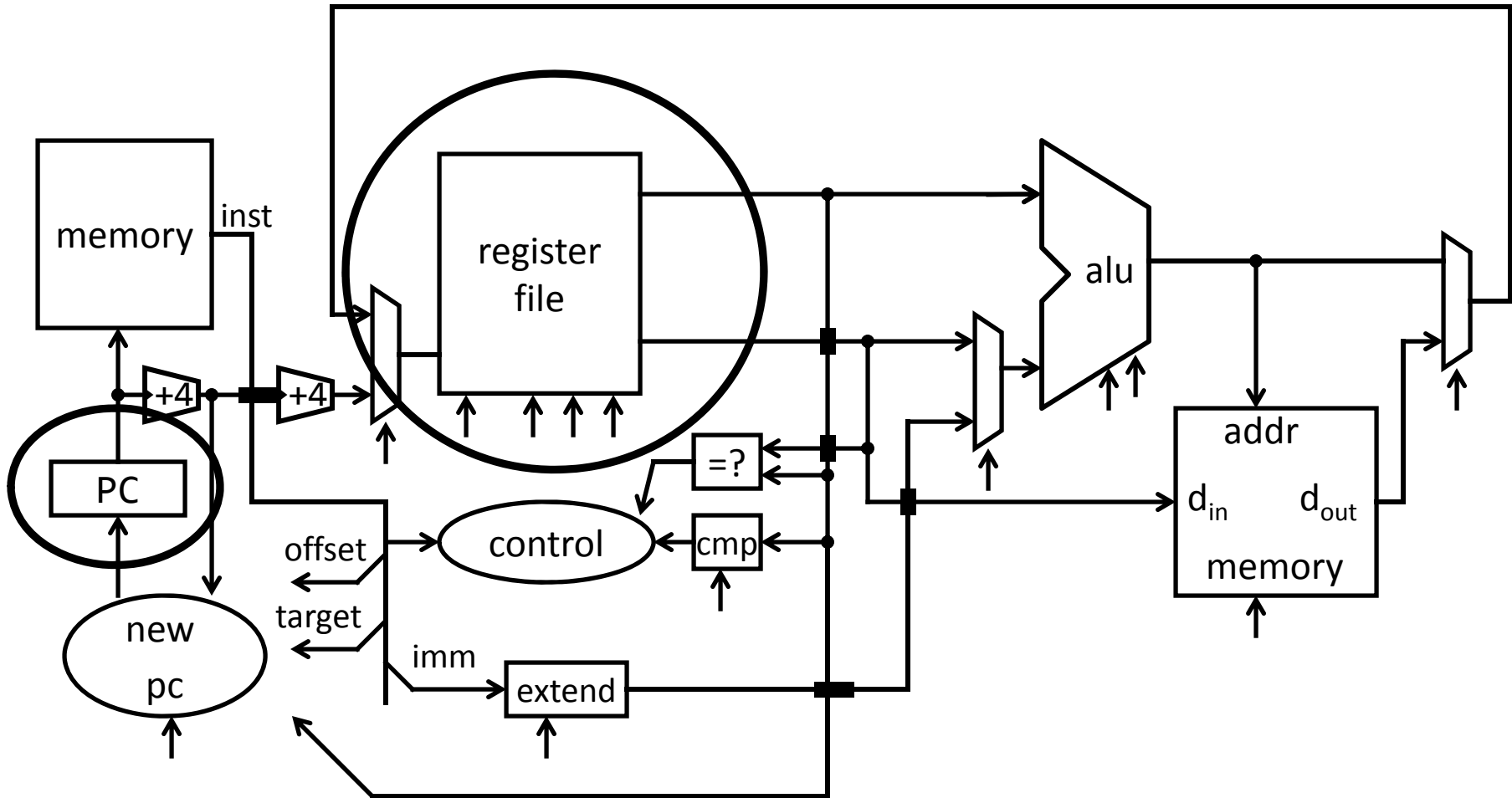
Understanding the basics of a processor

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

Putting it all together:

- Arithmetic Logic Unit (ALU)—Lab0 & 1, Lecture 2 & 3
- Register File—Lecture 4 and 5
- Memory—Lecture 5
 - SRAM: cache
 - DRAM: main memory
- Instruction-types
- Instruction Datapaths

MIPS Register File

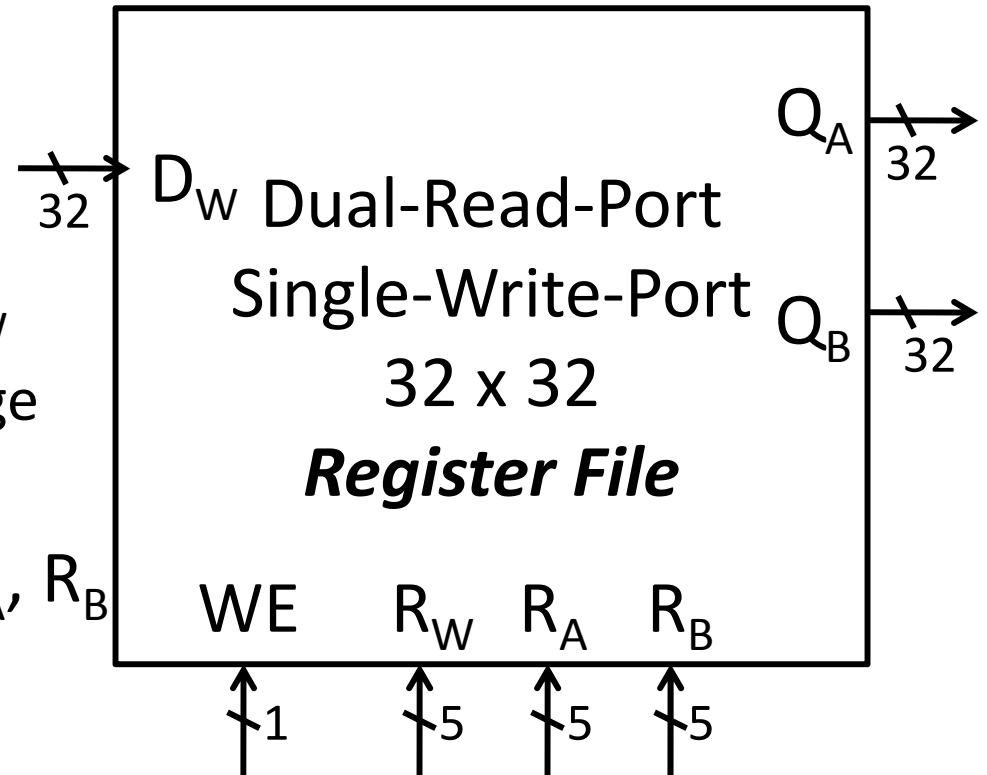


A Single cycle processor

MIPS Register file

MIPS register file

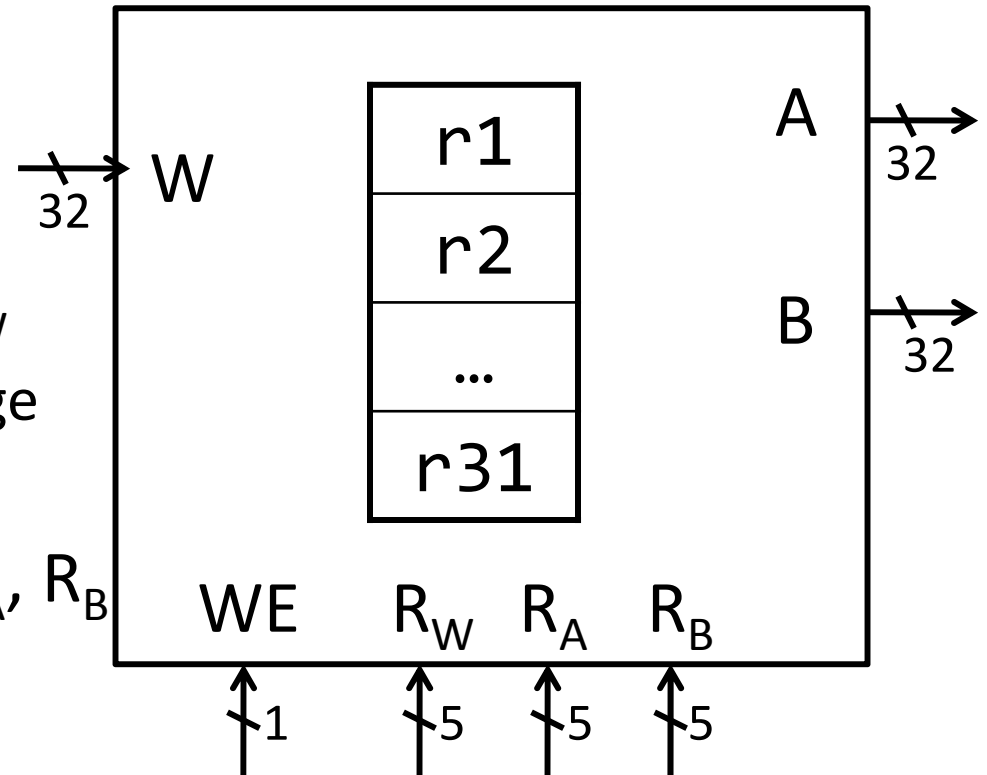
- 32 registers, 32-bits each (with r0 wired to zero)
- Write port indexed via R_W
 - Writes occur on falling edge but only if WE is high
- Read ports indexed via R_A, R_B



MIPS Register file

MIPS register file

- 32 registers, 32-bits each (with r0 wired to zero)
- Write port indexed via R_W
 - Writes occur on falling edge but only if WE is high
- Read ports indexed via R_A, R_B

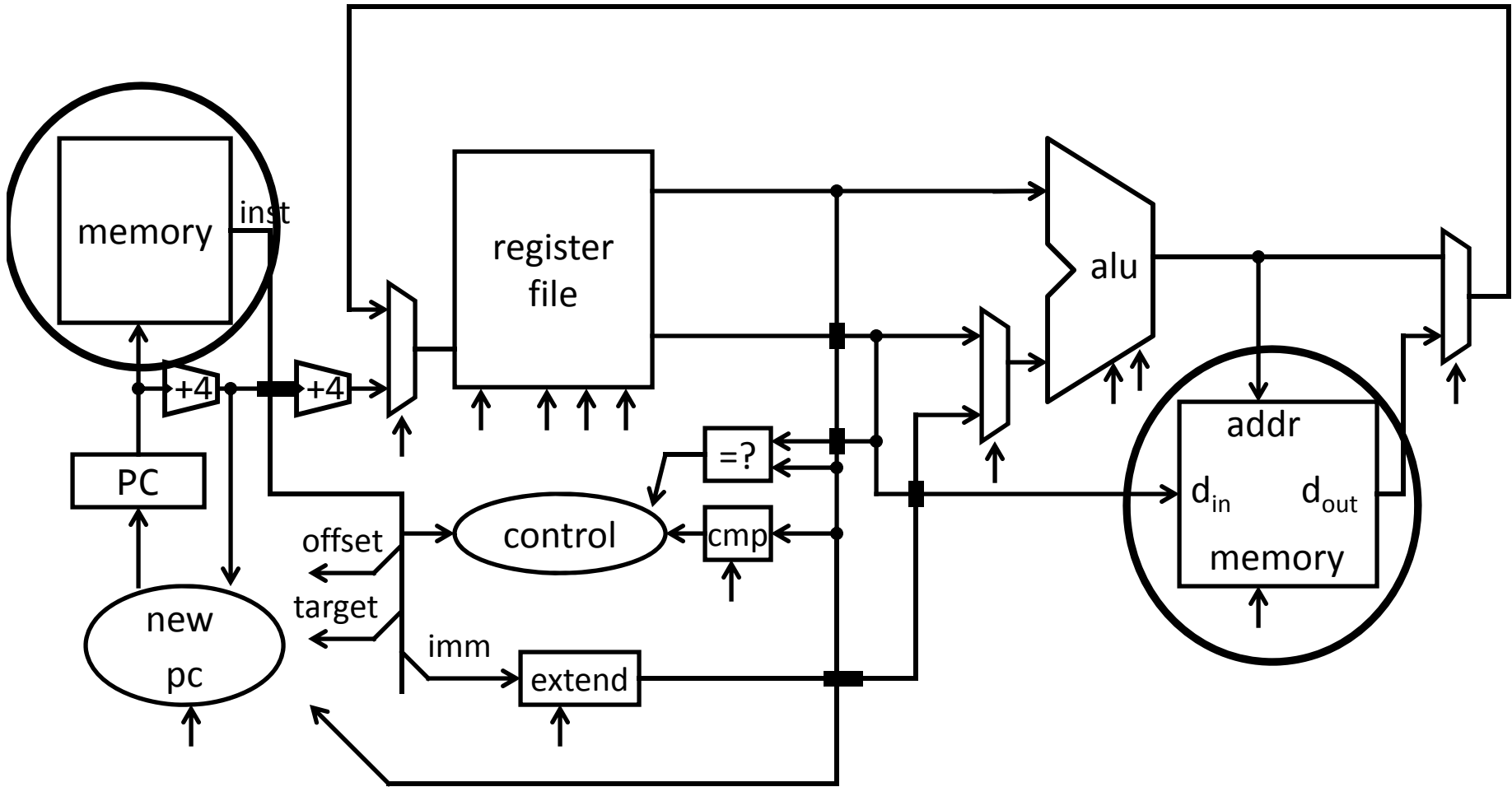


MIPS Register file

Registers

- Numbered from 0 to 31.
- Each register can be referred by number or name.
- \$0, \$1, \$2, \$3 ... \$31
- Or, by convention, each register has a name.
 - \$16 - \$23 → \$s0 - \$s7
 - \$8 - \$15 → \$t0 - \$t7
 - \$0 is always \$zero.
 - Patterson and Hennessy p105.

MIPS Memory

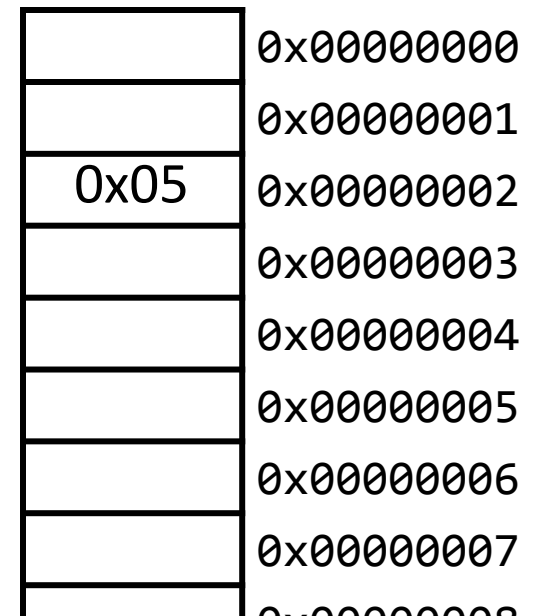
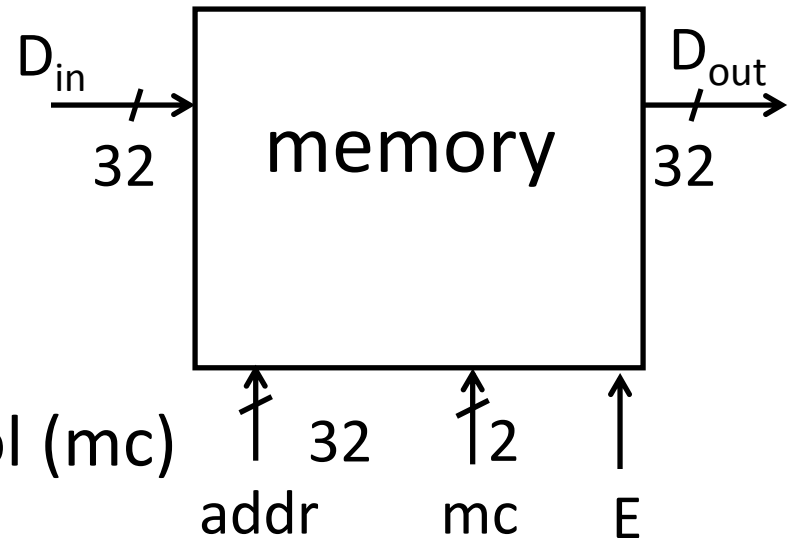


A Single cycle processor

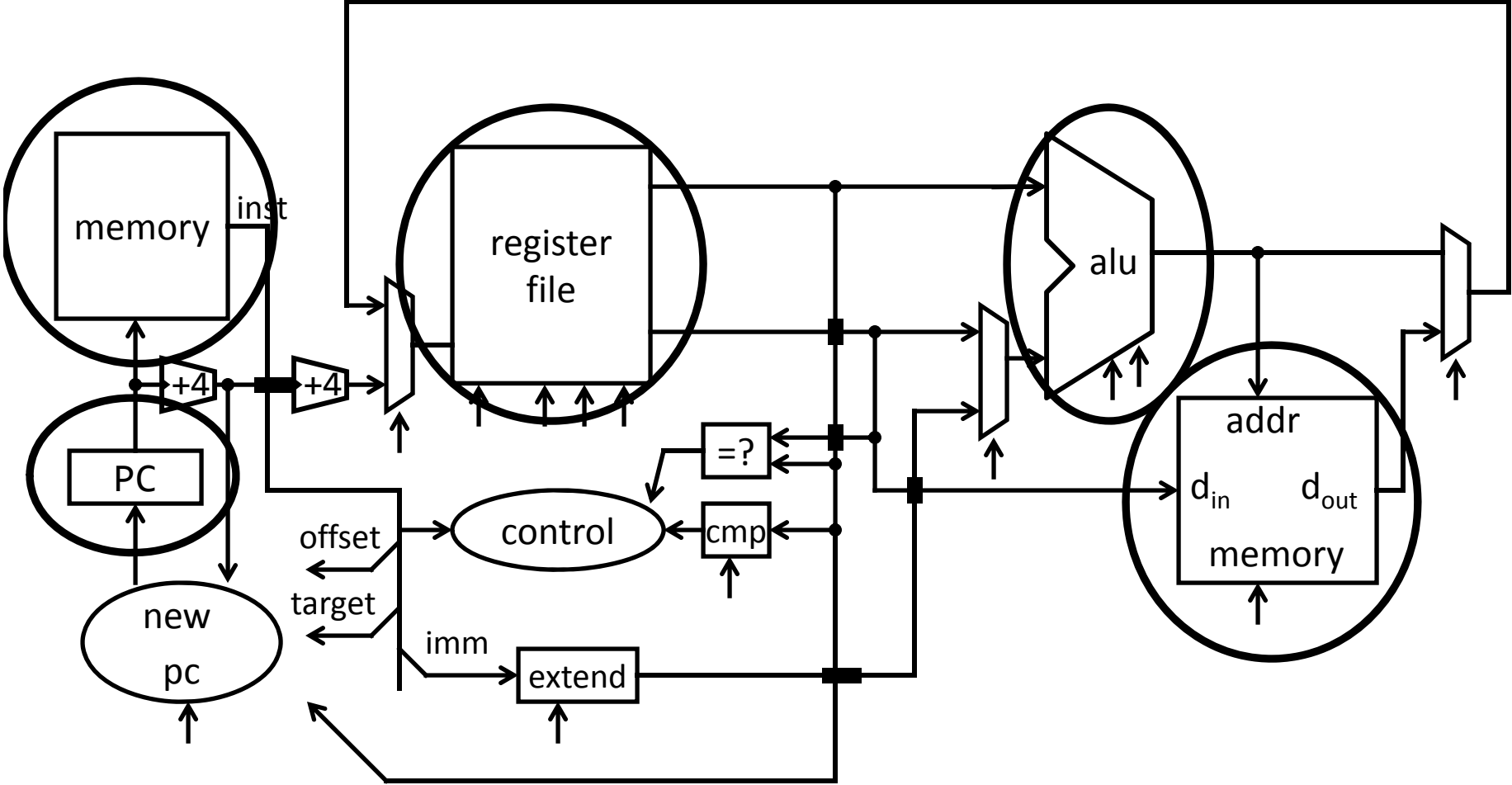
MIPS Memory

MIPS 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

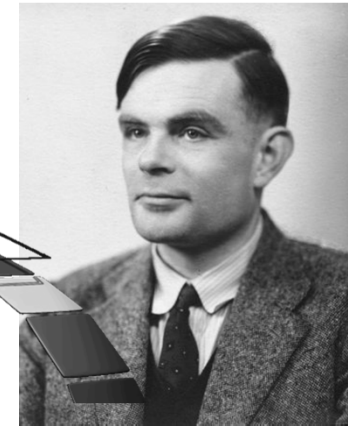
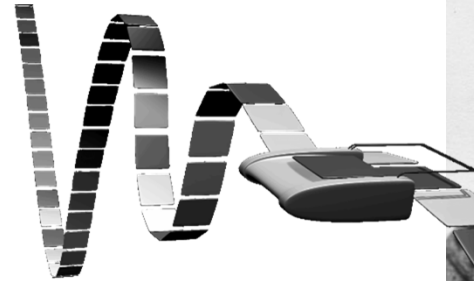
Harvard (modified) architecture

To make a computer

Need a program

Stored program computer

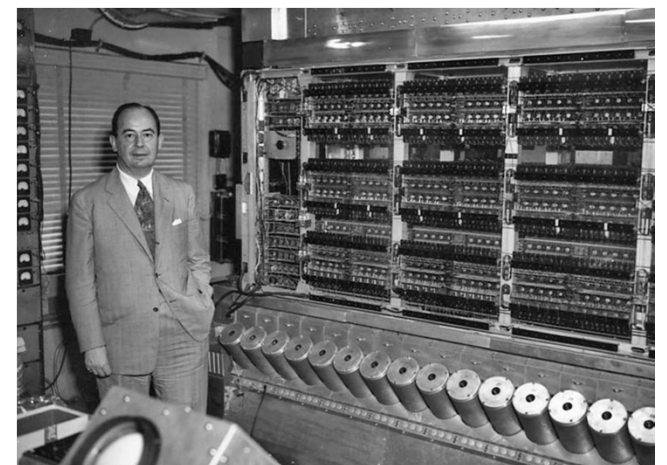
(a Universal Turing Machine)



Architectures

von Neumann architecture

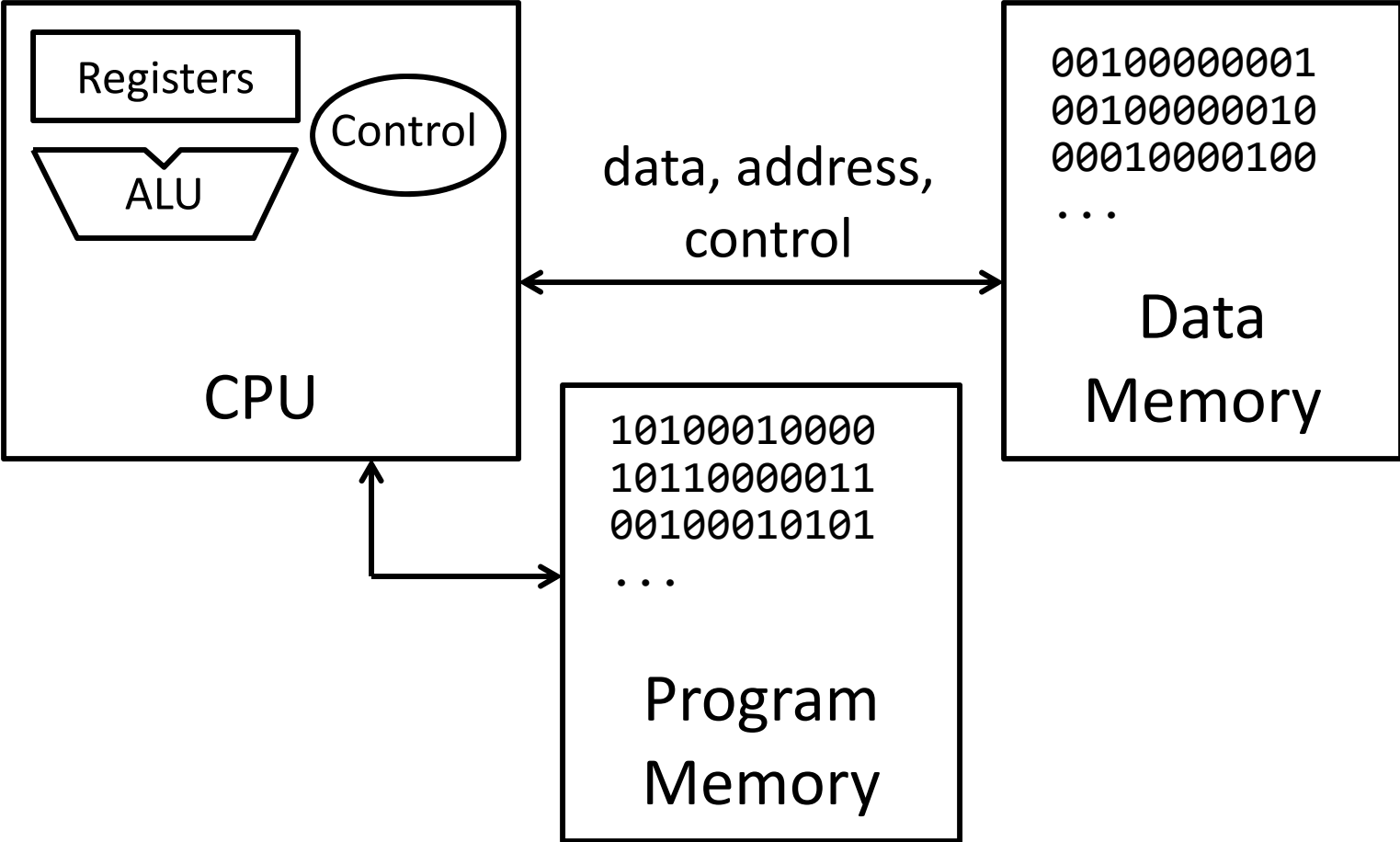
Harvard (modified) architecture



Putting it all together: Basic Processor

Let's build a MIPS CPU

- ...but using (modified) Harvard architecture



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 MIPS processor?

Levels of Interpretation: Instructions

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

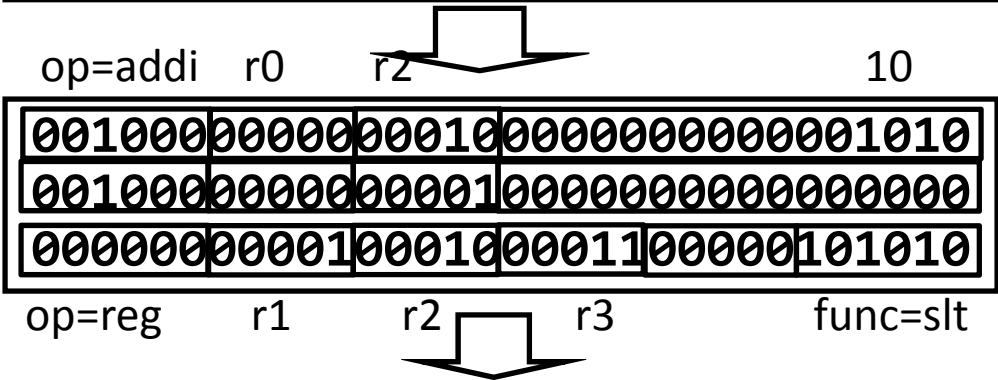
Programs written in a High Level Language

- C, Java, Python, Ruby, ...
- Loops, control flow, variables

Need translation to a lower-level computer understandable format

```
main: addi r2, r0, 10  
      addi r1, r0, 0  
loop: slt r3, r1, r2  
      ...
```

- **Assembly** is human readable machine language
- Processors operate on **Machine Language**



Machine Implementation

ALU, Control, Register File, ...

Levels of Interpretation: Instructions

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

High Level Language

- C, Java, Python, Ruby, ...
- Loops, control flow, variables

```
main: addi r2, r0, 10  
      addi r1, r0, 0  
loop: slt r3, r1, r2  
      ...
```

Assembly Language

- No symbols (except labels)
- One operation per statement

op=addi r0 r2 10

| | | | | |
|--------|--------|--------|------------------|--------|
| 001000 | 000000 | 000100 | 0000000000000000 | 1010 |
| 001000 | 000000 | 000010 | 0000000000000000 | 0000 |
| 000000 | 000010 | 000100 | 000110000000 | 101010 |

op=reg r1 r2 r3 func=slt

Machine Language

- Binary-encoded assembly
- Labels become addresses

ALU, Control, Register File, ...

Machine Implementation

Instruction Usage

Instructions are stored in memory, encoded in binary

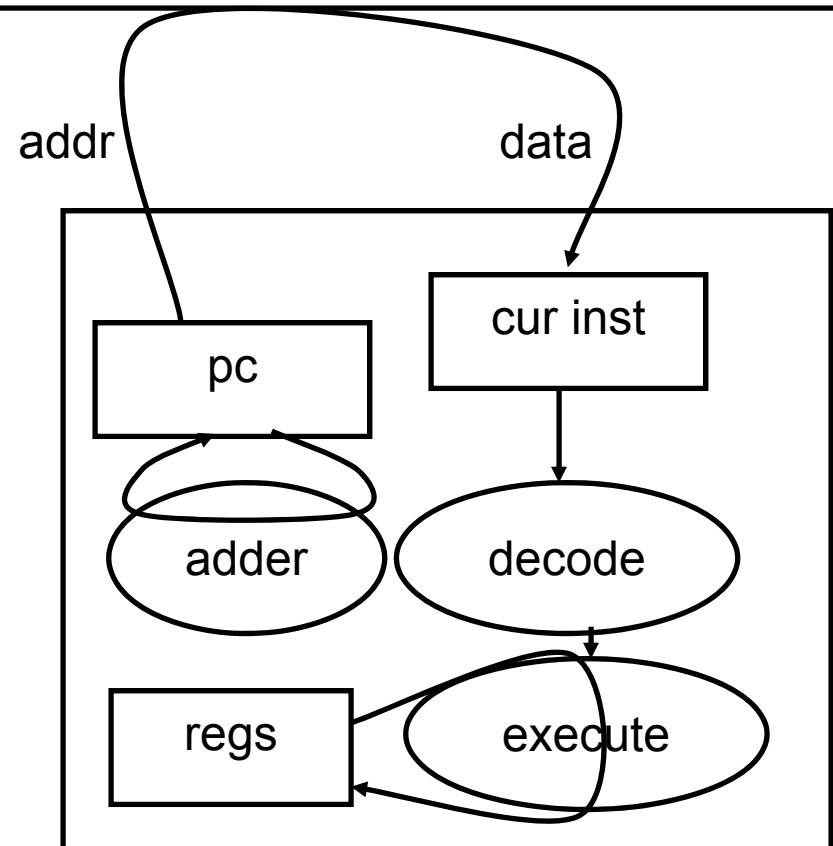
op=addi r0 r2 10

| | | | | |
|--------|--------|--------|------------------|--------|
| 001000 | 000000 | 000100 | 0000000000000000 | 1010 |
| 001000 | 000000 | 000010 | 0000000000000000 | 0000 |
| 000000 | 000010 | 000100 | 000110000000 | 101010 |

A basic processor

- fetches
- decodes
- executes

one instruction at a time



MIPS 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

- unconditional jumps
- conditional jumps (branches)
- jump and link (subroutine call)

Many other instructions are possible

- vector add/sub/mul/div, string operations
- manipulate coprocessor
- I/O

Instruction Set Architecture

The types of operations permissible in machine language define the ISA

- MIPS: load/store, arithmetic, control flow, ...
- VAX: load/store, arithmetic, control flow, strings, ...
- Cray: vector operations, ...

Two classes of ISAs

- Reduced Instruction Set Computers (RISC)
- Complex Instruction Set Computers (CISC)

We'll study the MIPS ISA in this course

Instruction Set Architecture

Instruction Set Architecture (ISA)

- Different CPU architecture specifies different set of instructions. Intel x86, IBM PowerPC, Sun Sparc, MIPS, etc.

MIPS

- \approx 200 instructions, 32 bits each, 3 formats
 - mostly orthogonal
- all operands in registers
 - almost all are 32 bits each, can be used interchangeably
- \approx 1 addressing mode: Mem[reg + imm]

x86 = Complex Instruction Set Computer (CISC)

- > 1000 instructions, 1 to 15 bytes each
- operands in special registers, general purpose registers, memory, on stack, ...
 - can be 1, 2, 4, 8 bytes, signed or unsigned
- 10s of addressing modes
 - e.g. Mem[segment + reg + reg*scale + offset]

Instructions

Load/store architecture

- Data must be in registers to be operated on
- Keeps hardware simple

Emphasis on efficient implementation

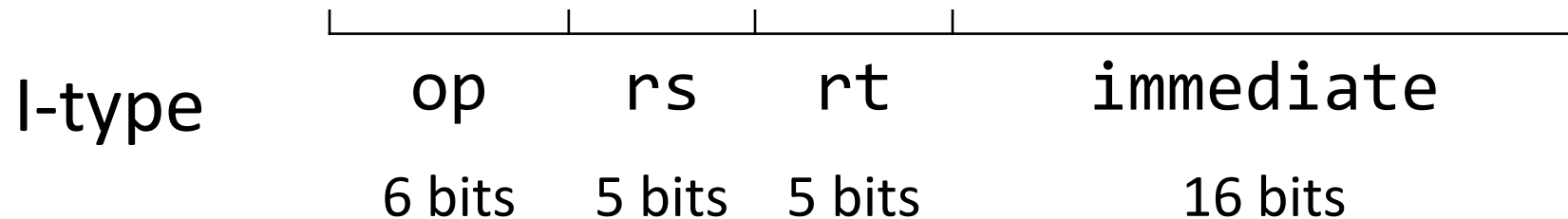
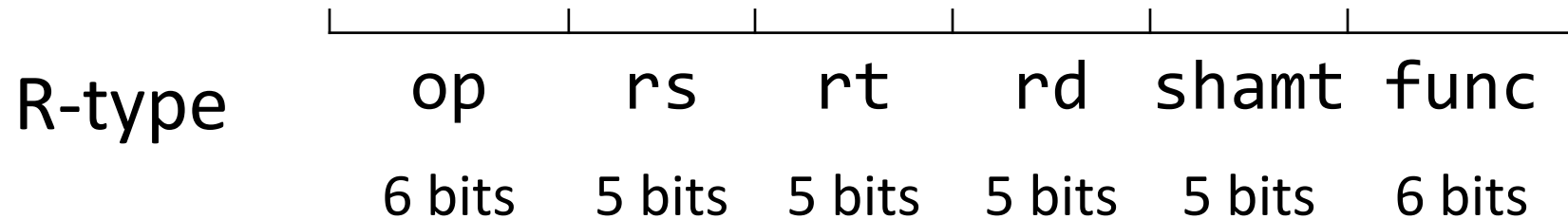
Integer data types:

- byte: 8 bits
- half-words: 16 bits
- words: 32 bits

MIPS supports signed and unsigned data types

MIPS instruction formats

All MIPS instructions are 32 bits long, has 3 formats



MIPS 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

Takeaway

A MIPS 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

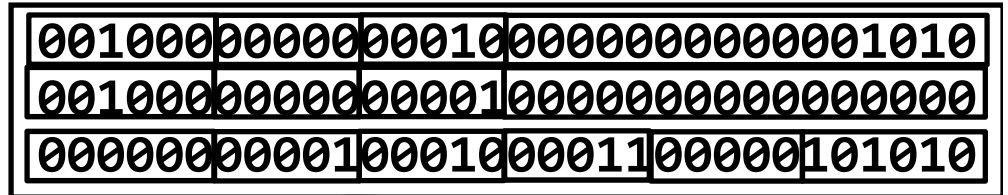
How are instructions executed?

What is the general datapath to execute an instruction?

Instruction Usage

Instructions are stored in memory, encoded in binary

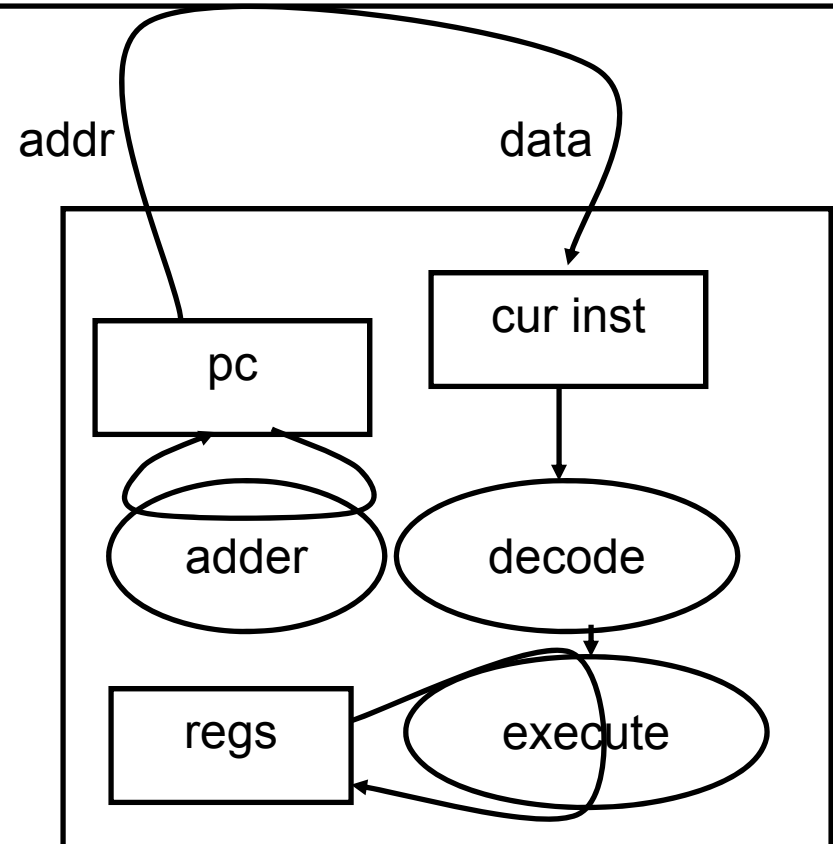
op=addi r0 r2 10



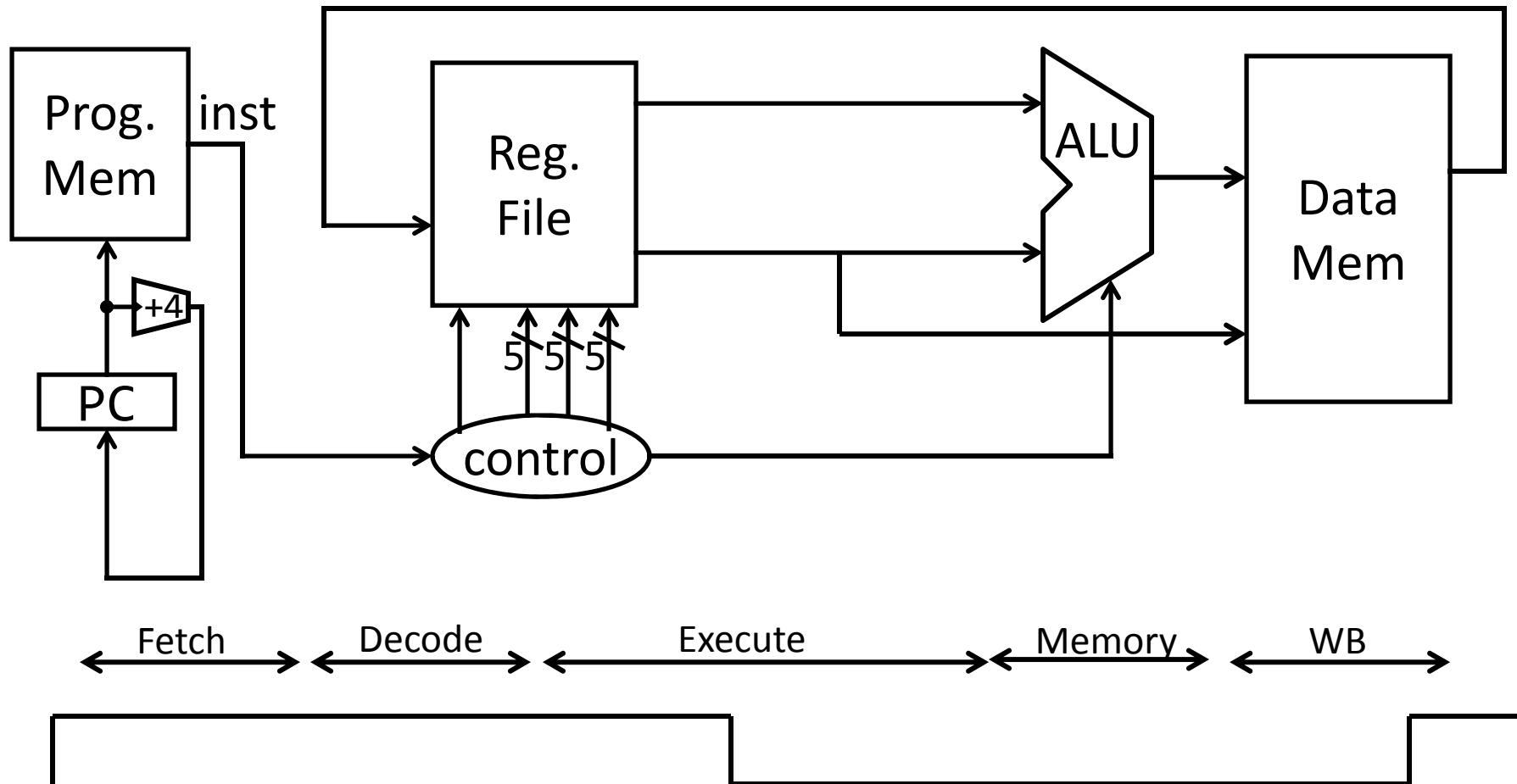
A basic processor

- fetches
- decodes
- executes

one instruction at a time



Five Stages of MIPS Datapath



A Single cycle processor

Five Stages of MIPS datapath

Basic CPU execution loop

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

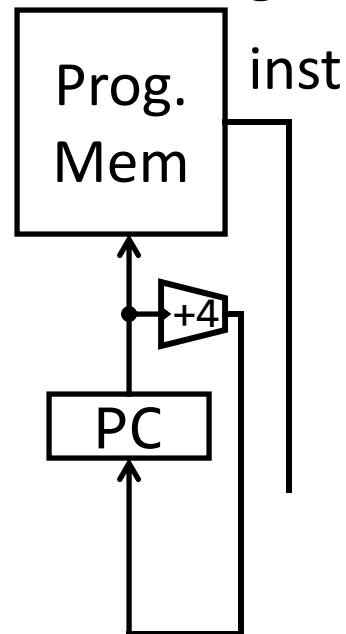
Instruction types/format

- Arithmetic/Register: `addu $s0, $s2, $s3`
- Arithmetic/Immediate: `slti $s0, $s2, 4`
- Memory: `lw $s0, 20($s3)`
- Control/Jump: `j 0xdeadbeef`

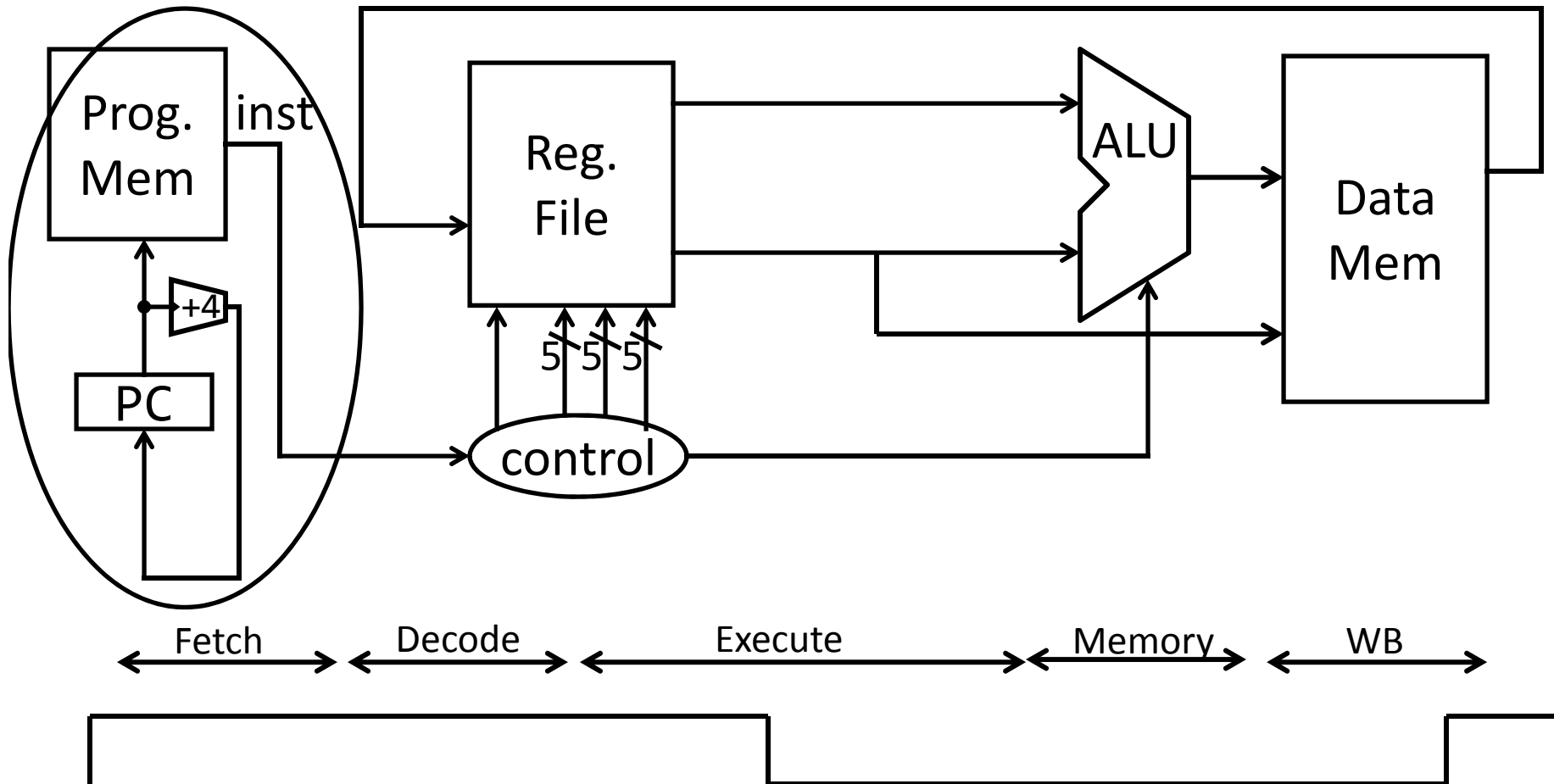
Stages of datapath (1/5)

Stage 1: Instruction Fetch

- Fetch 32-bit instruction from memory. (Instruction cache or memory)
- Increment PC accordingly.
 - +4, byte addressing
 - +N



Stages of datapath (1/5)

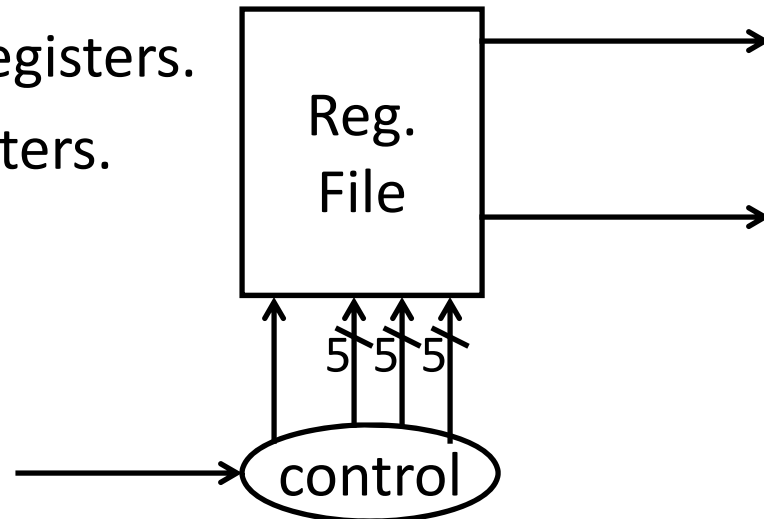


A Single cycle processor

Stages of datapath (2/5)

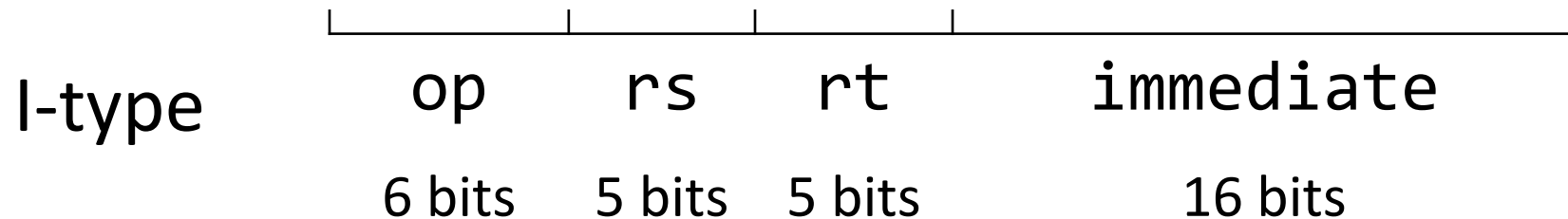
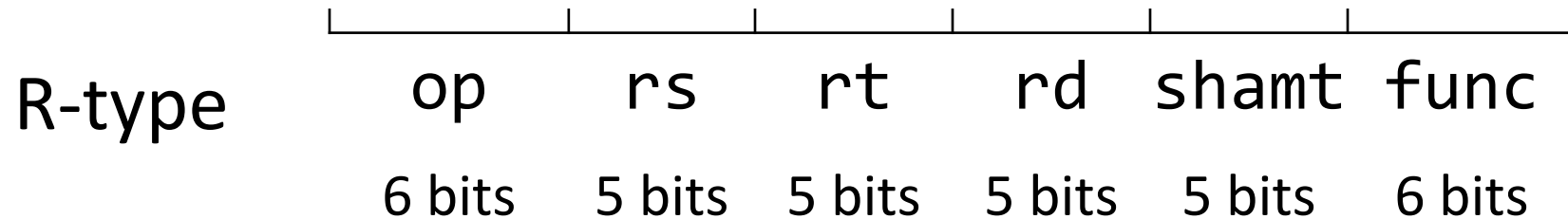
Stage 2: Instruction Decode

- Gather data from the instruction
- Read opcode to determine instruction type and field length
- Read in data from register file
 - E.g. for addu, read two registers.
 - E.g. for addi, read one registers.
 - E.g. for jal, read no registers.

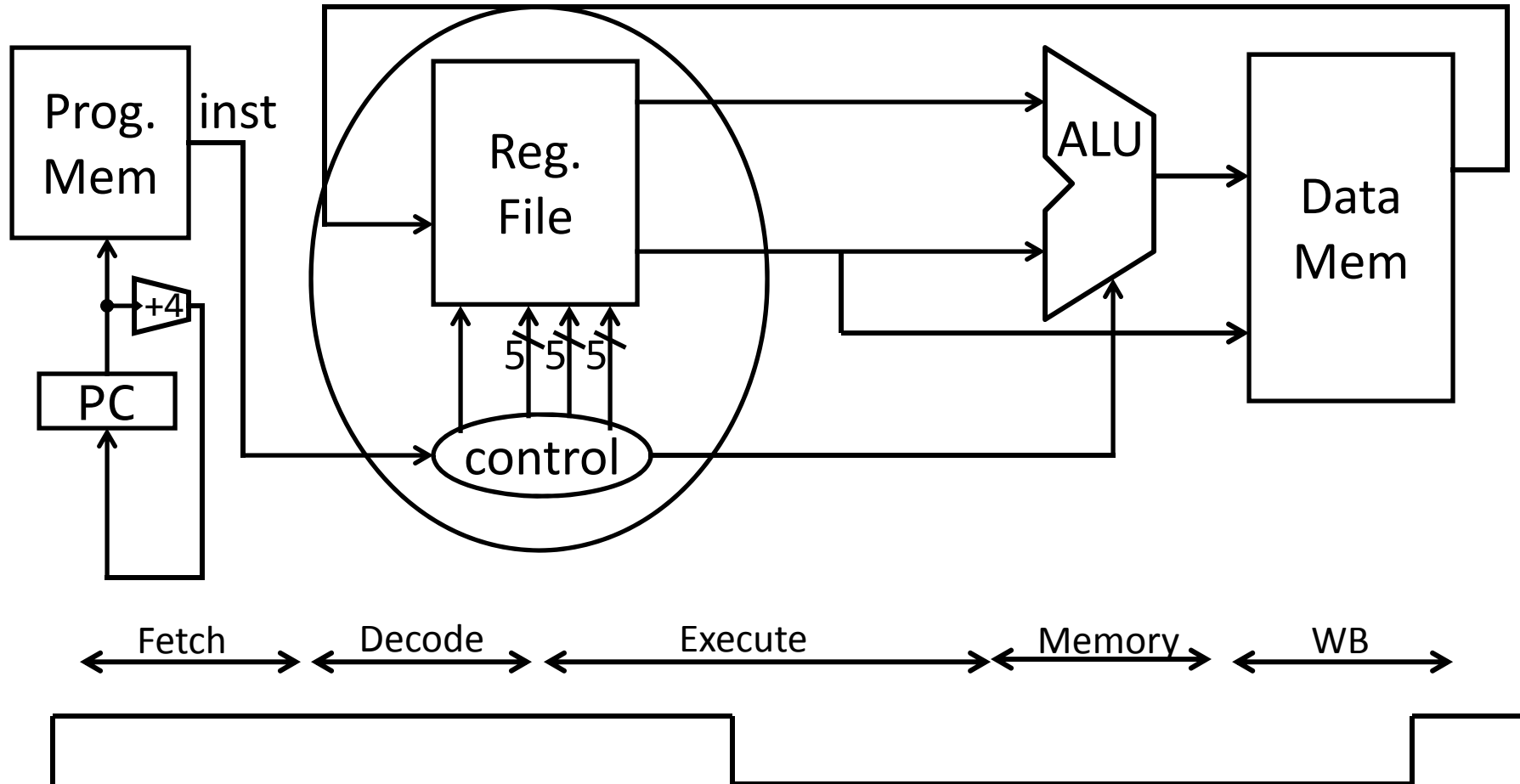


Stages of datapath (2/5)

All MIPS instructions are 32 bits long, has 3 formats



Stages of datapath (2/5)

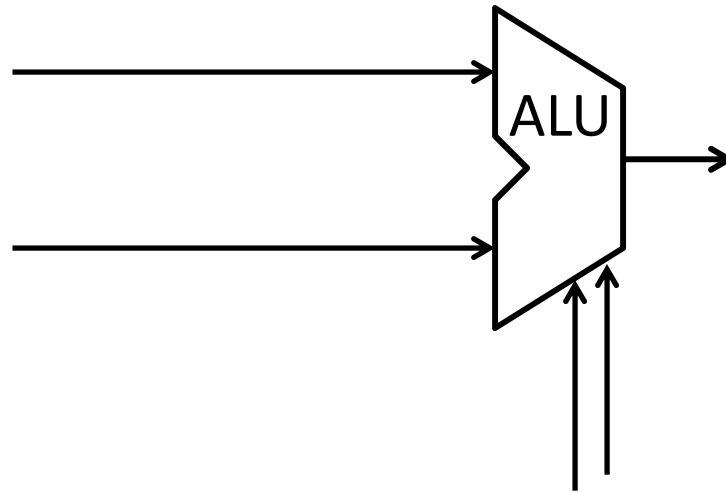


A Single cycle processor

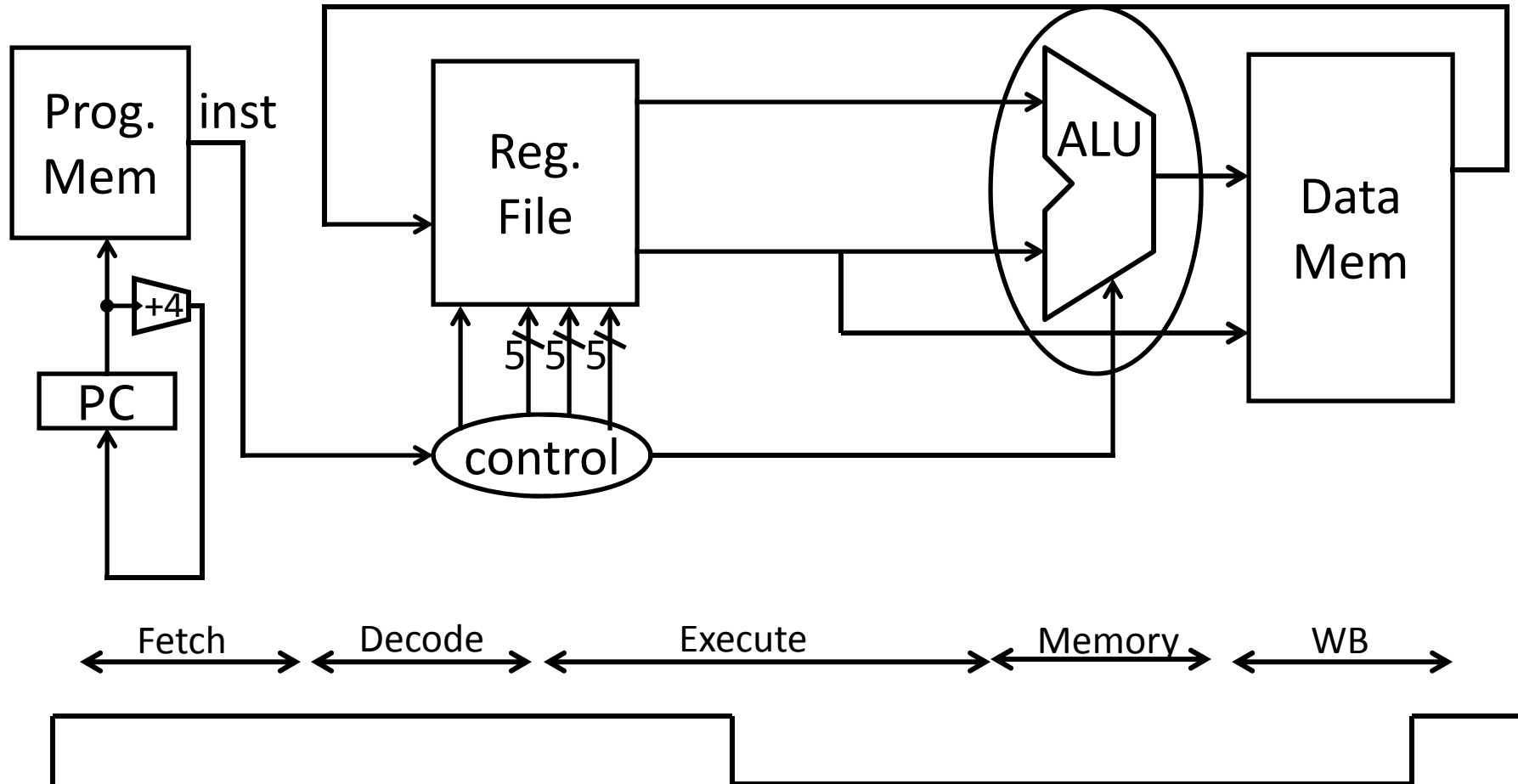
Stages of datapath (3/5)

Stage 3: Execution (ALU)

- Useful work is done here (+, -, *, /), shift, logic operation, comparison (slt).
- Load/Store?
 - lw \$t2, 32(\$t3)
 - Compute the address of the memory.



Stages of datapath (3/5)

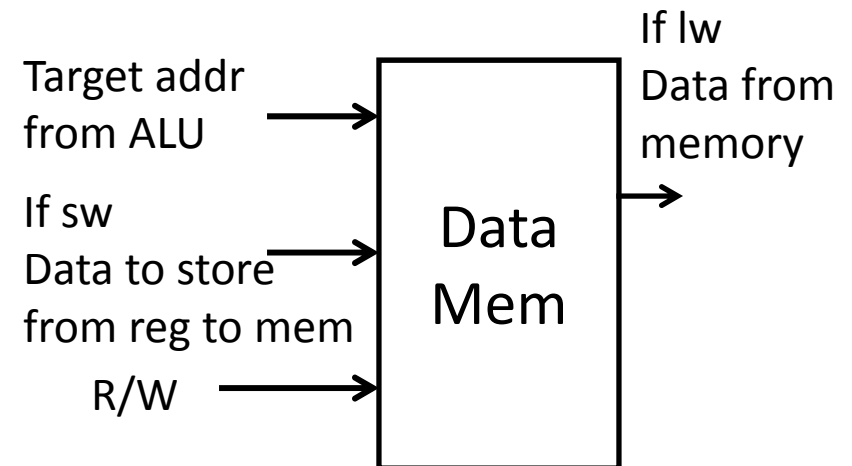


A Single cycle processor

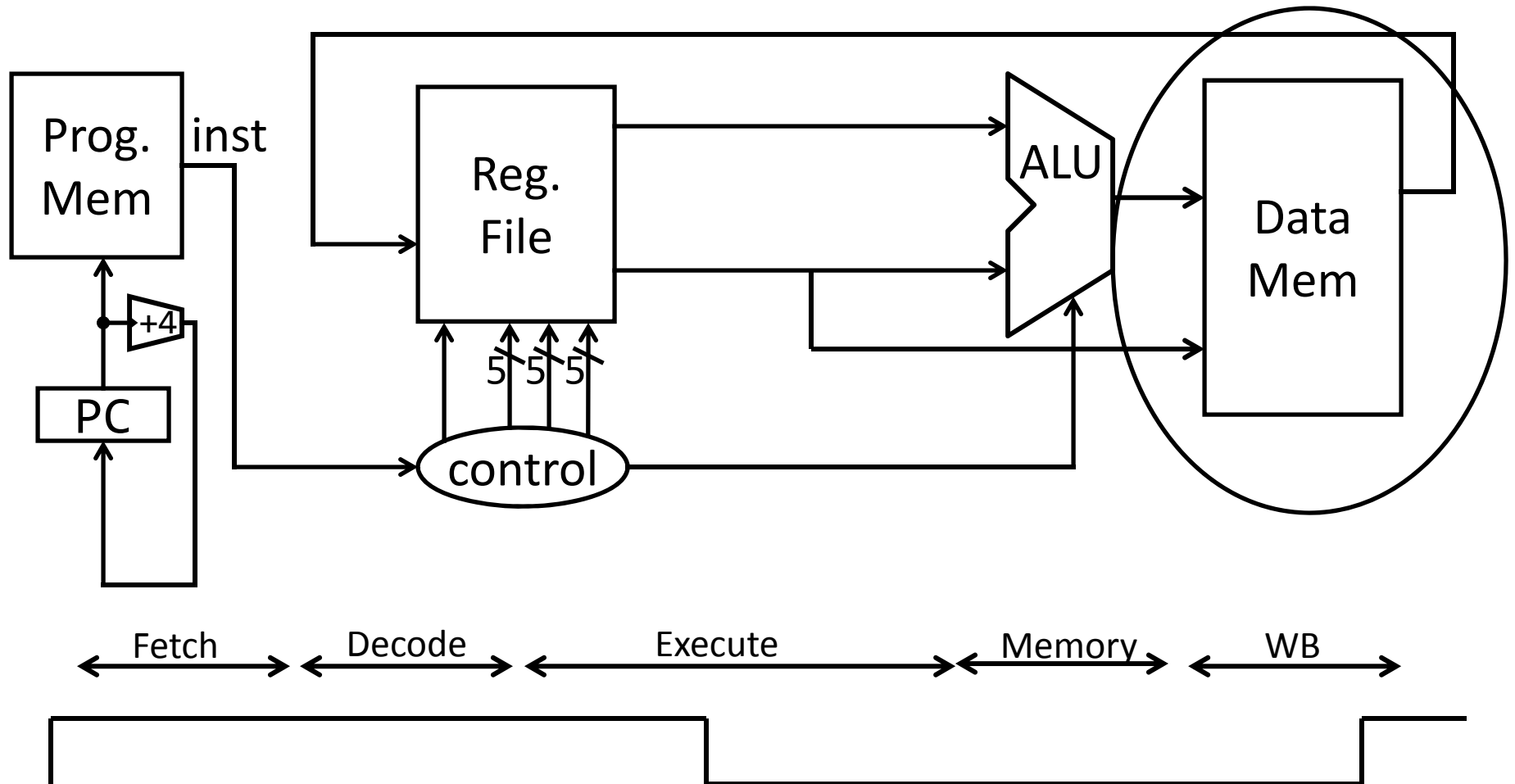
Stages of datapath (4/5)

Stage 4: Memory access

- Used by load and store instructions only.
- Other instructions will skip this stage.



Stages of datapath (4/5)

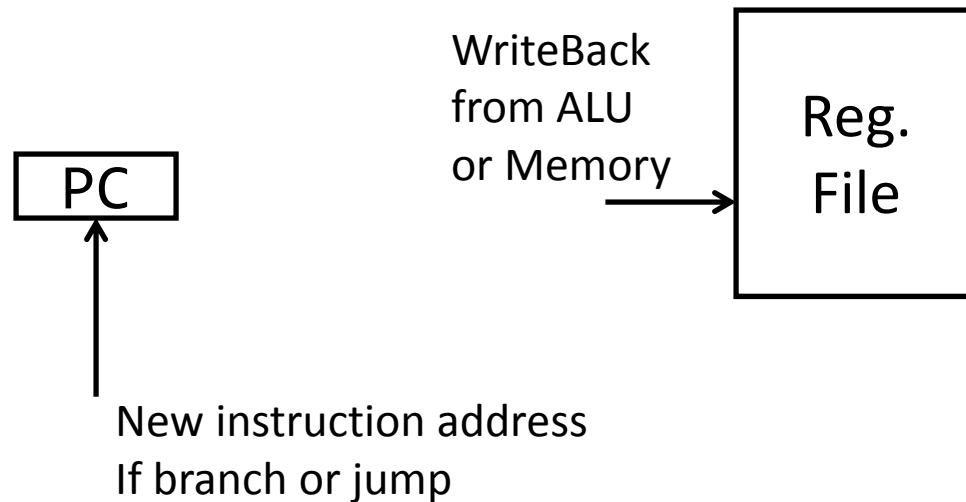


A Single cycle processor

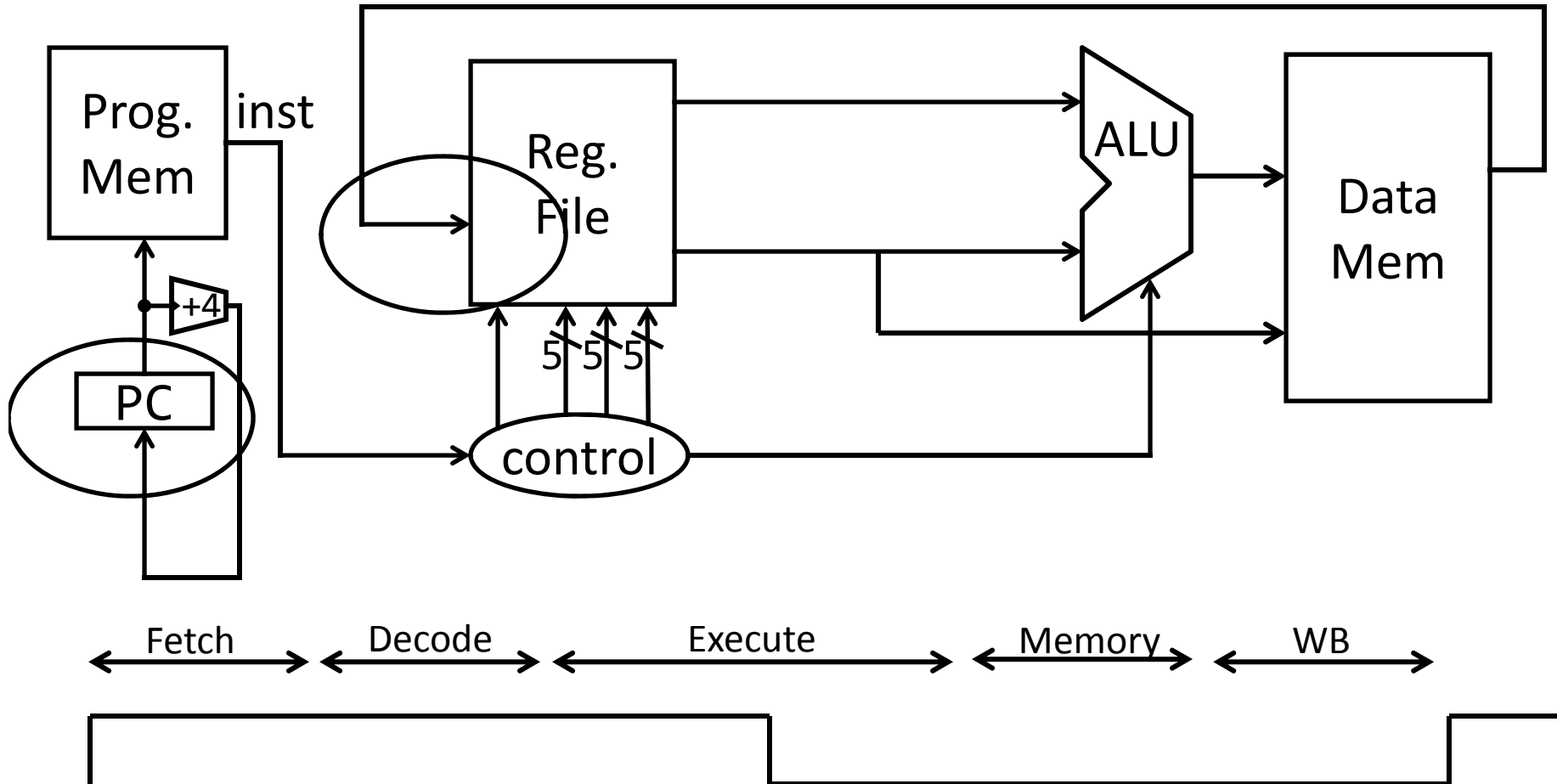
Stages of datapath (5/5)

Stage 5:

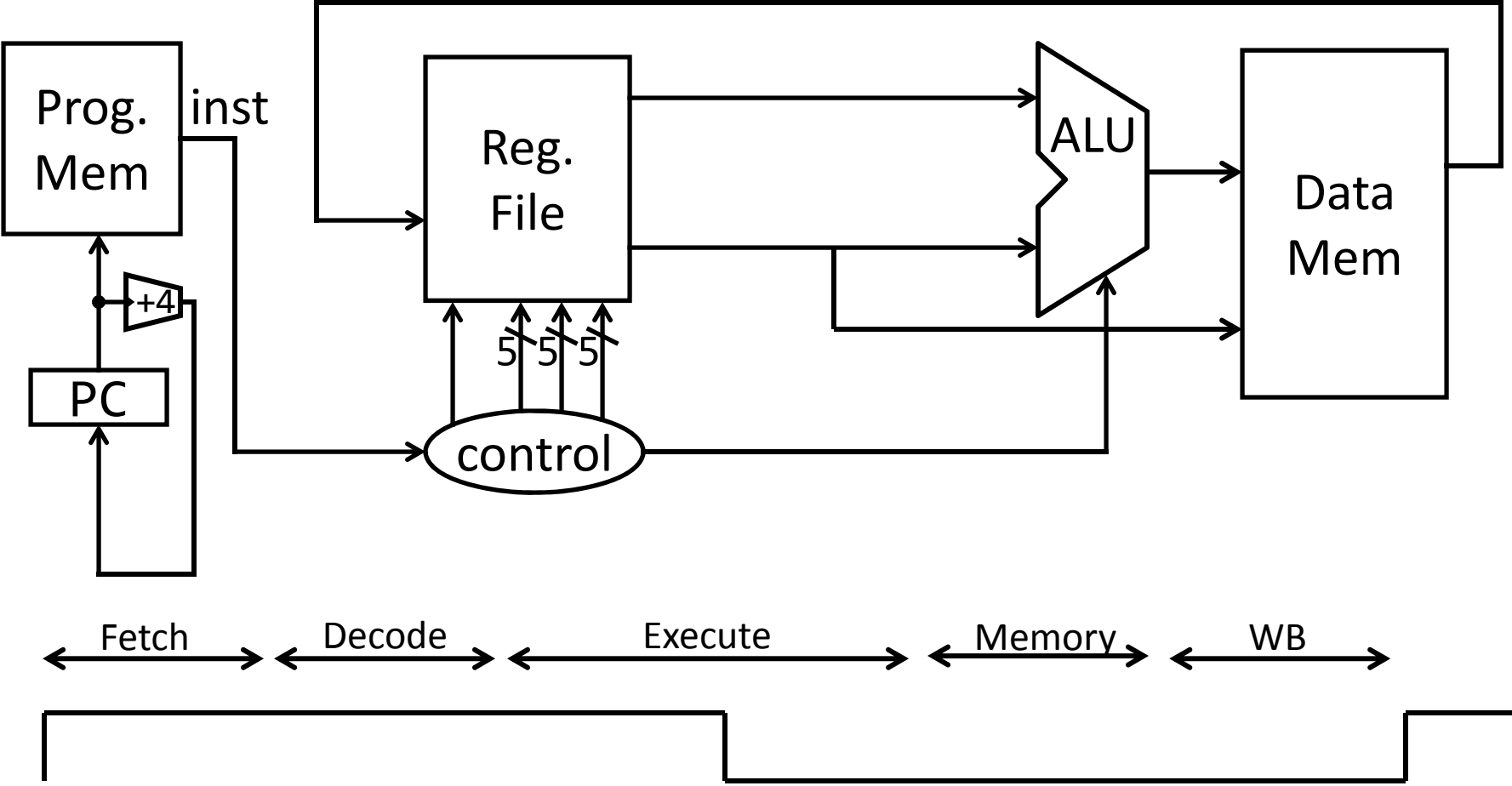
- For instructions that need to write value to register.
- Examples: arithmetic, logic, shift, etc, load.
- Store, branches, jump??



Stages of datapath (5/5)



Full Datapath



Takeaway

The datapath for a MIPS 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 MIPS instructions

Next Goal

Specific datapaths MIPS Instructions

MIPS Instruction Types

Arithmetic/Logical

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

Memory Access

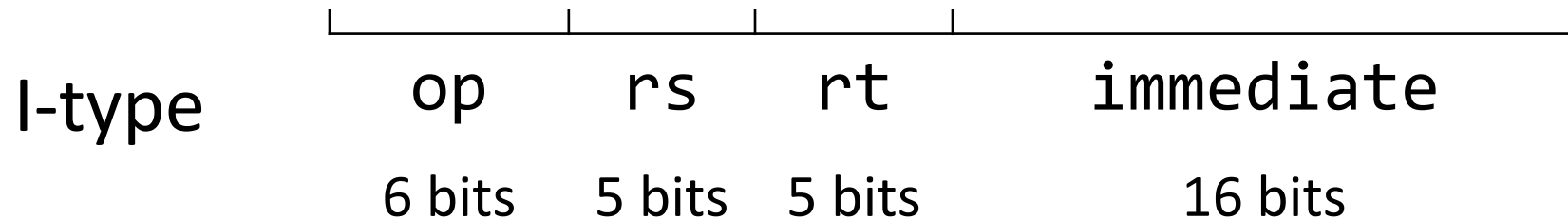
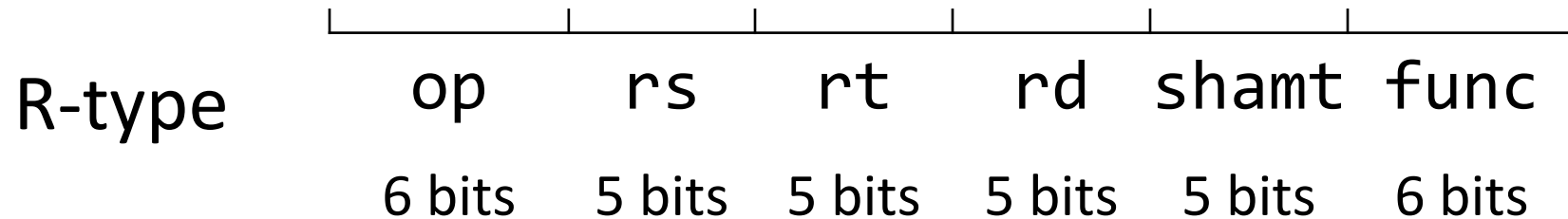
- load/store between registers and memory
- word, half-word and byte operations

Control flow

- conditional branches: pc-relative addresses
- jumps: fixed offsets, register absolute

MIPS instruction formats

All MIPS instructions are 32 bits long, has 3 formats



Arithmetic Instructions

00000001000001100010000000100110

op rs rt rd - func

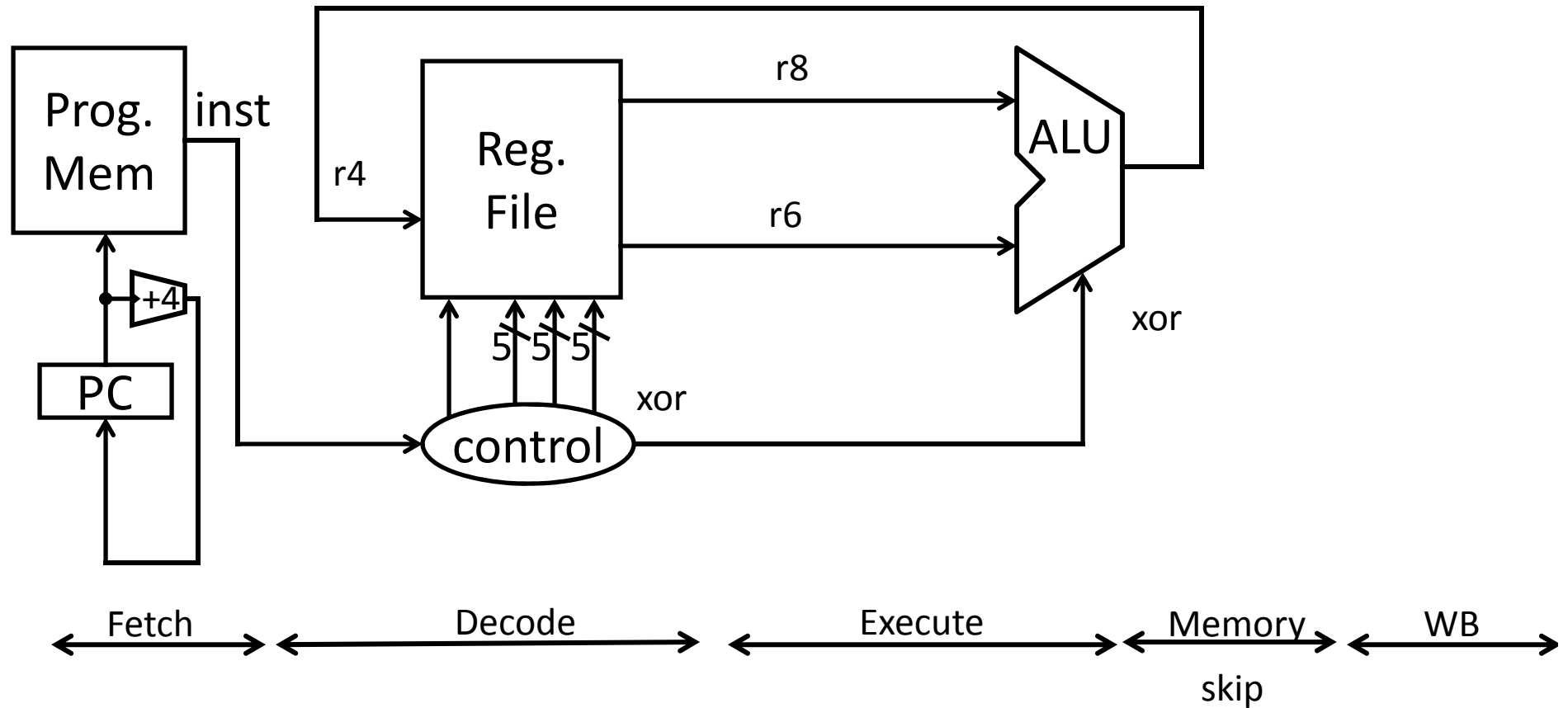
6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

R-Type

| op | func | mnemonic | description |
|-----|------|-----------------|----------------------------------|
| 0x0 | 0x21 | ADDU rd, rs, rt | $R[rd] = R[rs] + R[rt]$ |
| 0x0 | 0x23 | SUBU rd, rs, rt | $R[rd] = R[rs] - R[rt]$ |
| 0x0 | 0x25 | OR rd, rs, rt | $R[rd] = R[rs] R[rt]$ |
| 0x0 | 0x26 | XOR rd, rs, rt | $R[rd] = R[rs] \oplus R[rt]$ |
| 0x0 | 0x27 | NOR rd, rs, rt | $R[rd] = \sim (R[rs] R[rt])$ |

ex: $r4 = r8 \oplus r6$ # XOR r4, r8, r6

Arithmetic and Logic



ex: $r4 = r8 \oplus r6$ # XOR r4, r8, r6

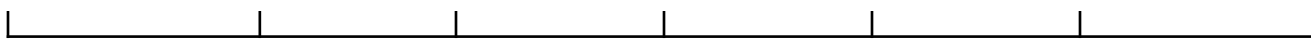


Instruction fetch + decode + ALU

= Babbage's engine + speed + reliability – hand crank

Arithmetic Instructions: Shift

00000000000000100010000011000000



op - rt rd shamt func

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

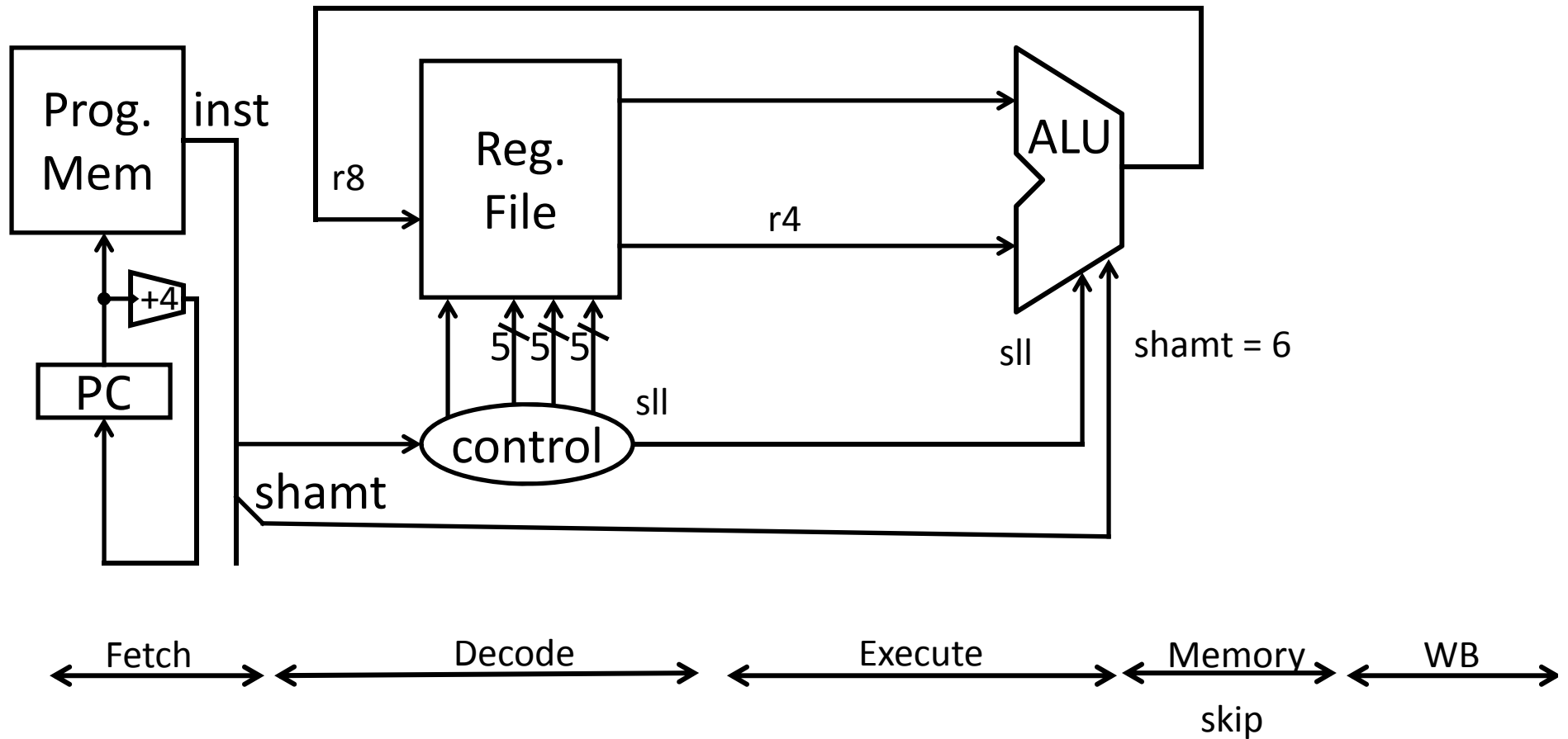


| op | func | mnemonic | description |
|-----|------|-------------------|-------------------------------------|
| 0x0 | 0x0 | SLL rd, rt, shamt | R[rd] = R[rt] << shamt |
| 0x0 | 0x2 | SRL rd, rt, shamt | R[rd] = R[rt] >>> shamt (zero ext.) |
| 0x0 | 0x3 | SRA rd, rt, shamt | R[rd] = R[rt] >> shamt (sign ext.) |

ex: r8 = r4 * 64 # SLL r8, r4, 6

 r8 = r4 << 6

Shift

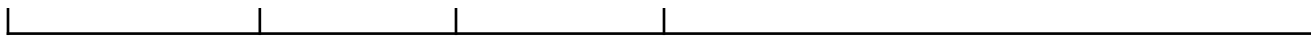


ex: $r8 = r4 * 64$
 $r8 = r4 \ll 6$

`# SLL r8, r4, 6`

Arithmetic Instructions: Immediates

001001001010010100000000000000101



op

rs

rd

immediate

6 bits

5 bits

5 bits

16 bits

I-Type

| op | mnemonic | description |
|-----|-------------------|---|
| 0x9 | ADDIU rd, rs, imm | $R[rd] = R[rs] + \text{sign_extend}(imm)$ |
| 0xc | ANDI rd, rs, imm | $R[rd] = R[rs] \& \text{zero_extend}(imm)$ |
| 0xd | ORI rd, rs, imm | $R[rd] = R[rs] \text{zero_extend}(imm)$ |

ex: $r5 = r5 + 5$ # ADDIU r5, r5, 5

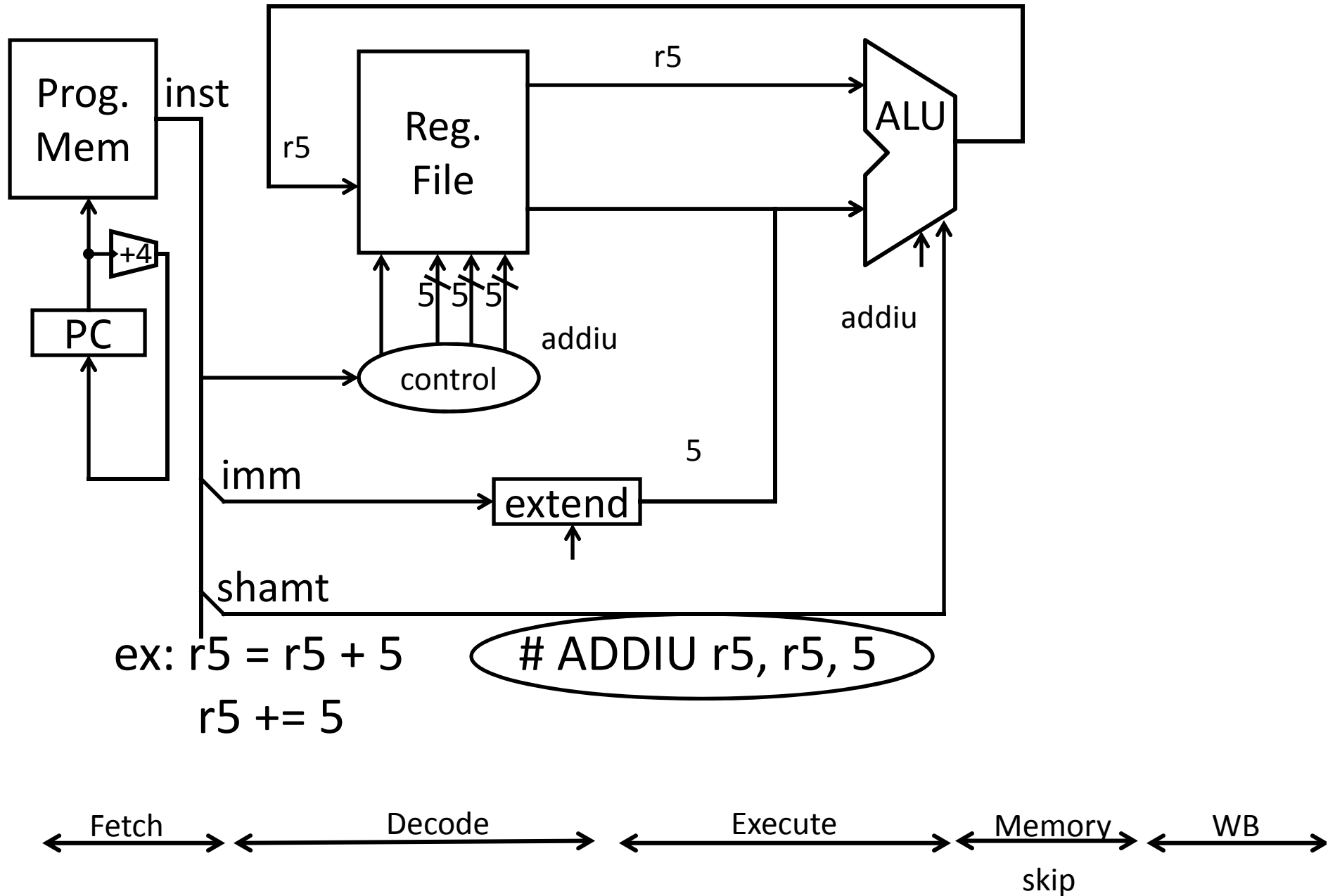
$r5 += 5$

What if immediate is negative?

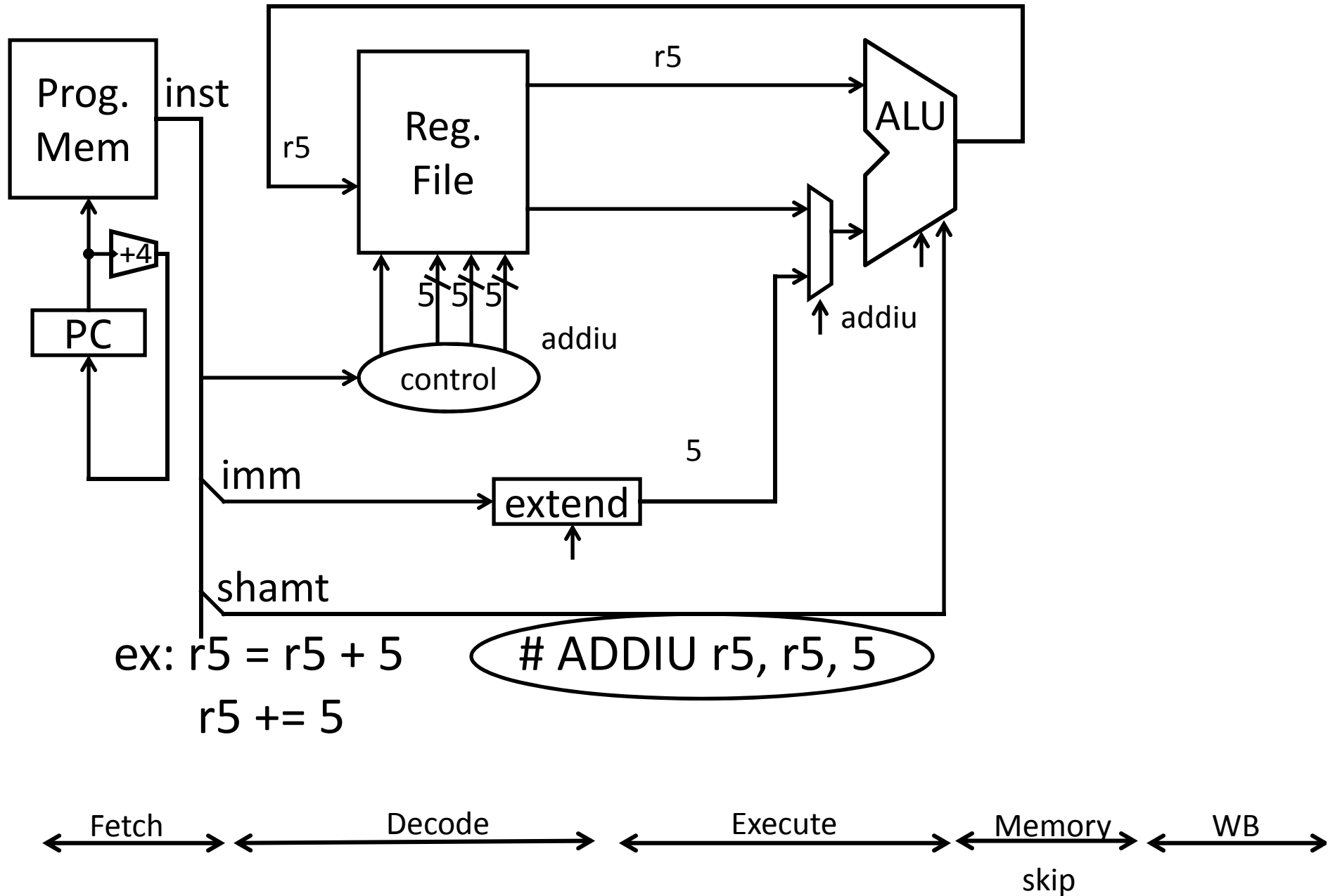
ex: $r5 += -1$

ex: $r5 += 65535$

Immediates



Immediates



Arithmetic Instructions: Immediates

00111100000001010000000000000101



op

-

rd

immediate

6 bits

5 bits

5 bits

16 bits



I-Type

| op | mnemonic | description |
|-----|-------------|-------------------|
| 0xF | LUI rd, imm | R[rd] = imm << 16 |

ex: r5 = 0x50000 # LUI r5, 5

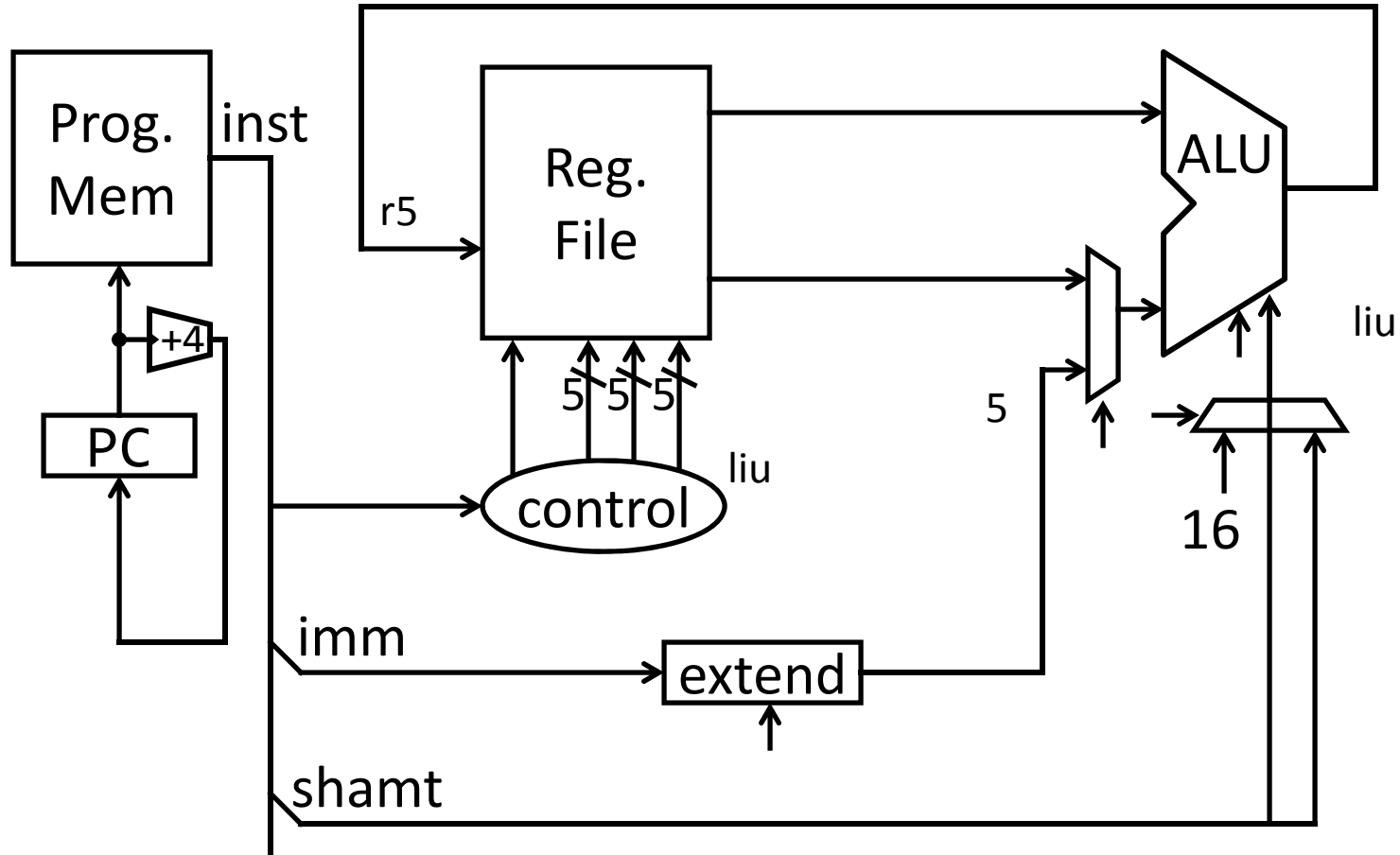
ex: LUI r5, 0xdead

ORI r5, r5 0xbeef

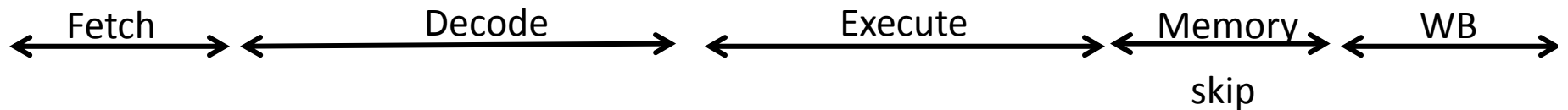
What does r5 = ?

r5 = 0xdeadbeef

Immediates



ex: $r5 = 0x50000$ # LUI *r5*, 5



MIPS Instruction Types

Arithmetic/Logical

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

Memory Access

- load/store between registers and memory
- word, half-word and byte operations

Control flow

- conditional branches: pc-relative addresses
- jumps: fixed offsets, register absolute

Next
Time

Summary

We have all that it takes to build a processor!

- Arithmetic Logic Unit (ALU)—Lab0 & 1, Lecture 2 & 3
- Register File—Lecture 4 and 5
- Memory—Lecture 5
 - SRAM: cache
 - DRAM: main memory

A MIPS 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!!

We know the data path for the MIPS ISA register and memory instructions