

# Processor

**Anne Bracy**

**CS 3410**

Computer Science

Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, and Sireer.

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

# Goal for Today

Understanding the basics of a processor

We now have the technology to build a CPU!

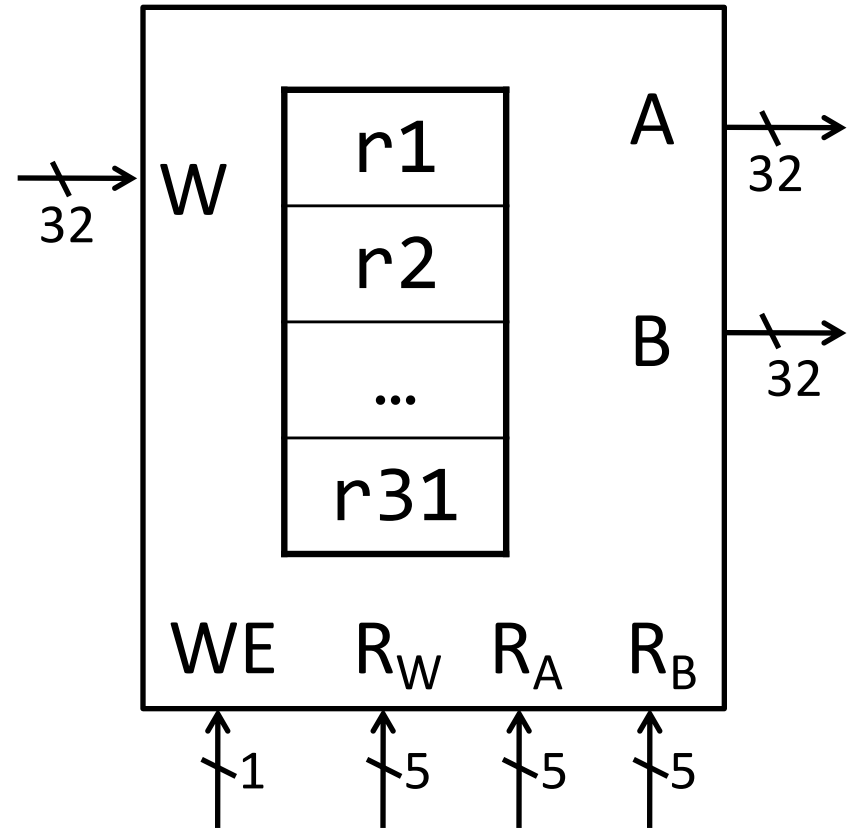
Putting it all together:

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

# MIPS Register file

## MIPS register file

- 32 x 32-bit registers
- r0 wired to zero
- Write port indexed via  $R_W$ 
  - on falling edge when  $WE=1$
- Read ports indexed via  $R_A, R_B$



## Registers

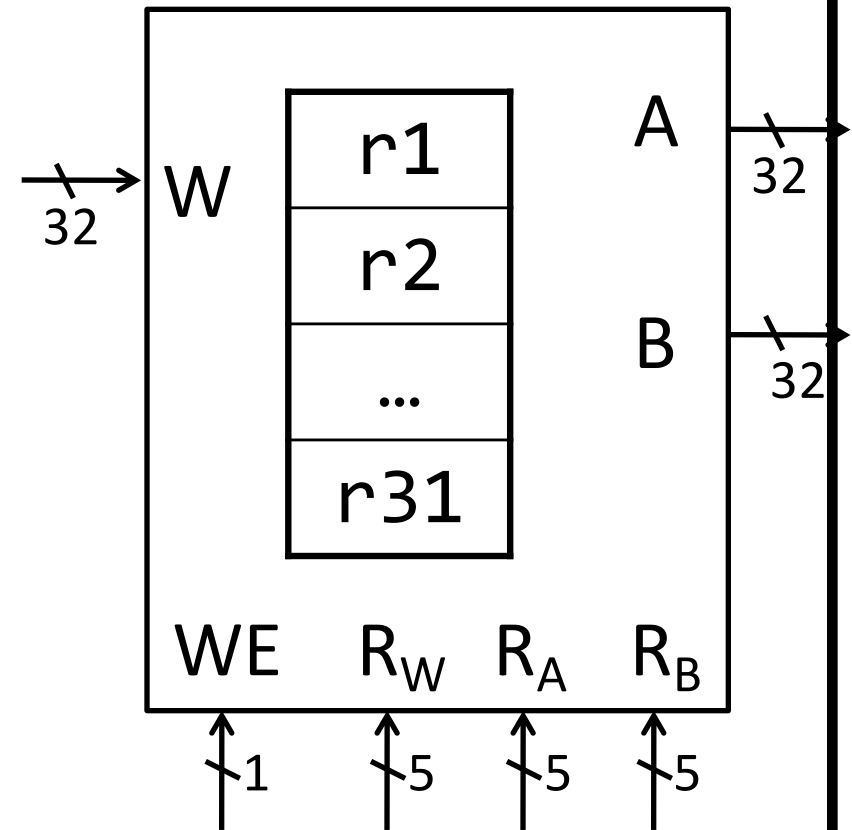
- Numbered from 0 to 31.
- Can be referred by number: \$0, \$1, \$2, ... \$31
- Convention, each register also has a name:
  - \$16 - \$23 → \$s0 - \$s7, \$8 - \$15 → \$t0 - \$t7

[P&H p105]

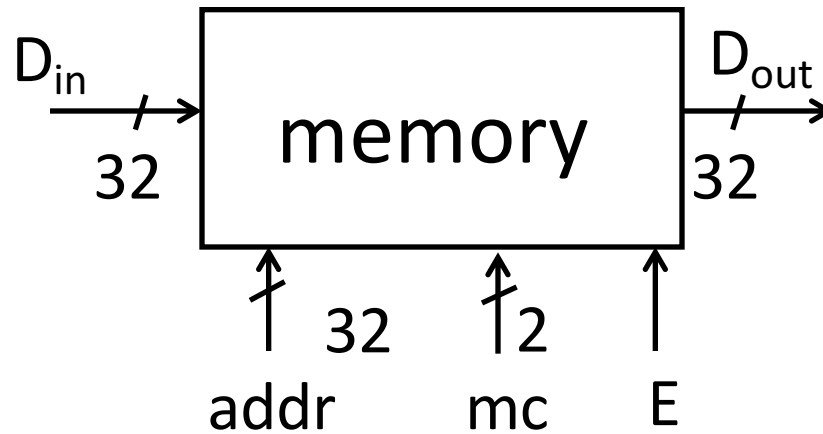
# iClicker Question

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

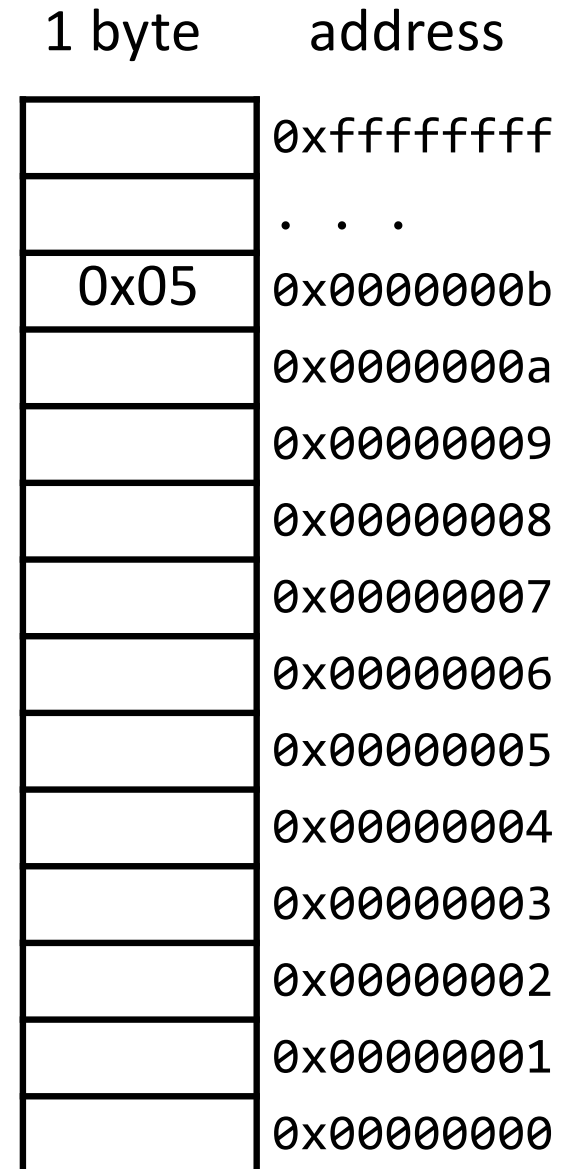
- (A)  $W, A, B \ 32 \rightarrow 64$
- (B)  $R_w, R_a, R_b \ 5 \rightarrow 6$
- (C)  $W \ 32 \rightarrow 64, R_w \ 5 \rightarrow 6$
- (D) A & B only



# MIPS Memory



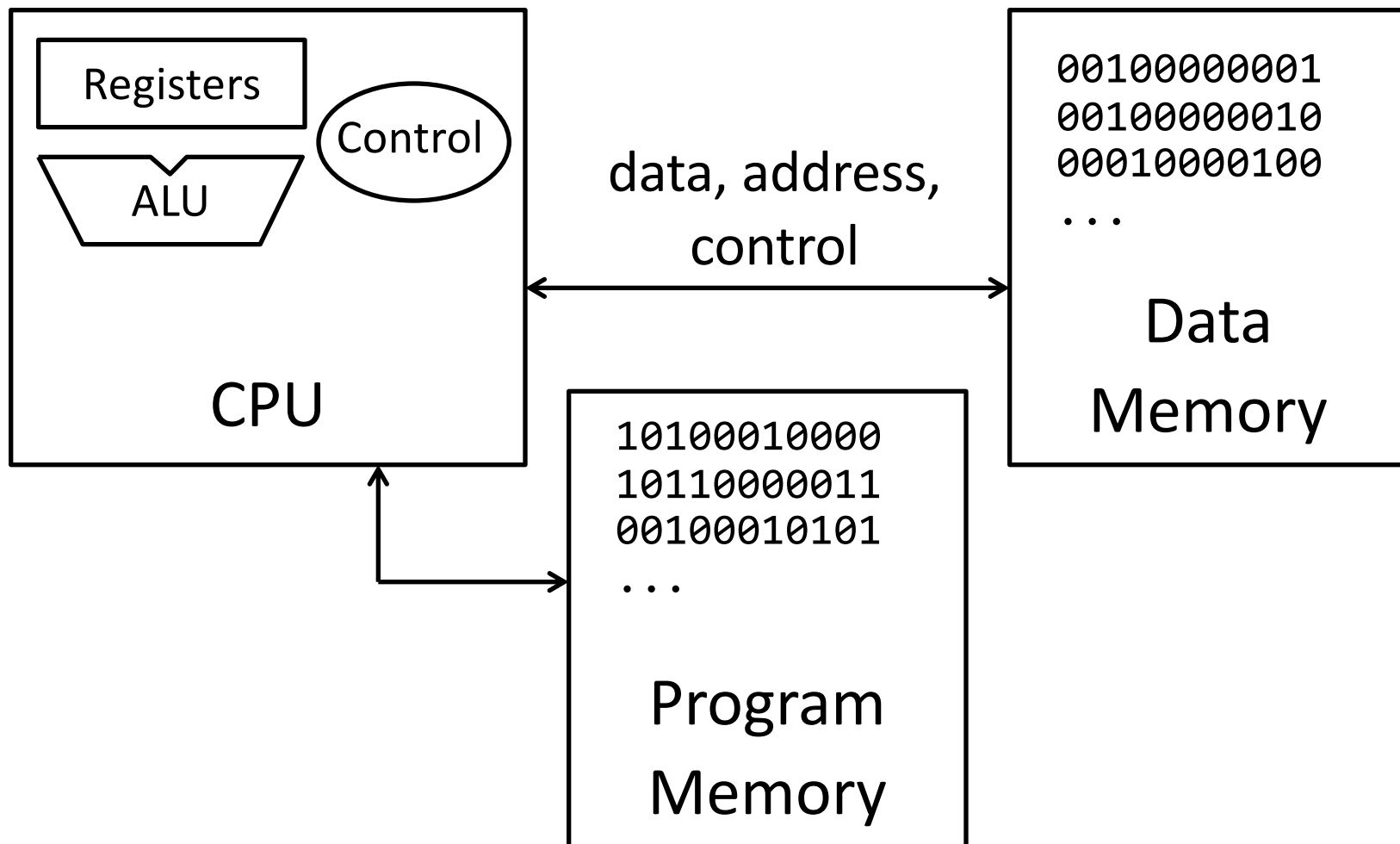
- 32-bit address
- 32-bit data (but byte addressed)
- Enable + 2 bit memory control ( $mc$ )
  - 00: read word (4 byte aligned)
  - 01: write byte
  - 10: write halfword (2 byte aligned)
  - 11: write word (4 byte aligned)



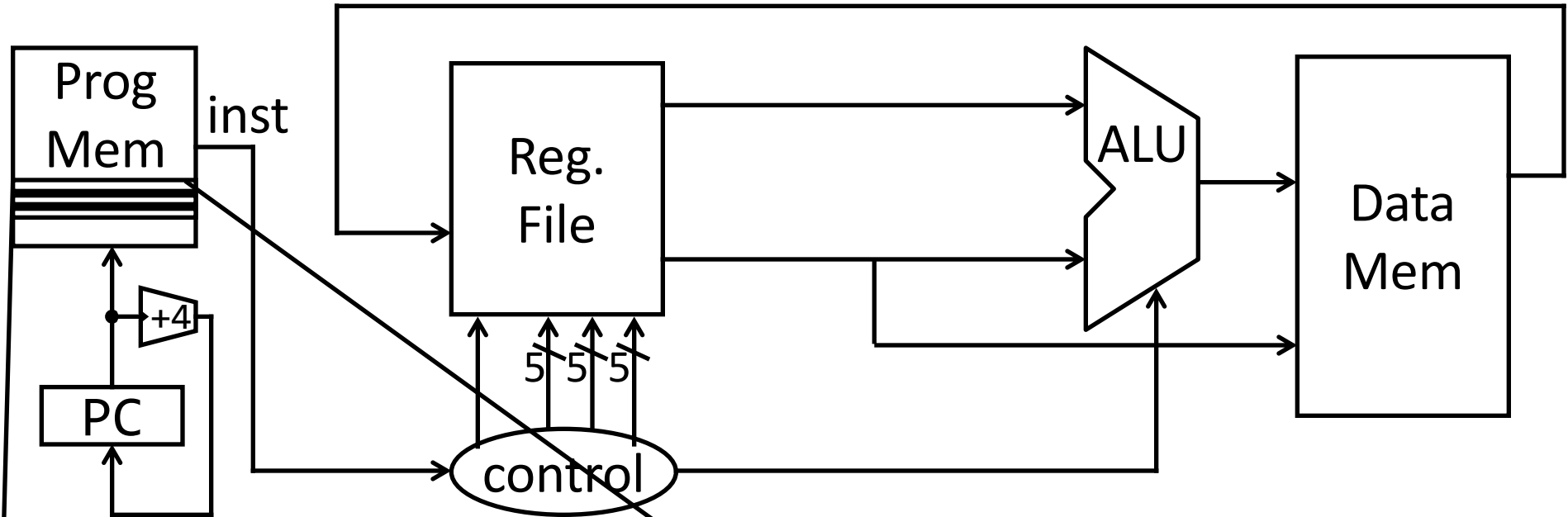
# Basic Processor

A MIPS CPU with a (modified) Harvard architecture

- Modified: insns & data in common addr space
- Not von Neumann: ours access insn & data in parallel



# Instruction Processing



Instructions:

stored in memory, encoded in binary

0010000000000001000000000000001010

0010000000000000100000000000000000

0000000001000100001100000101010

A basic processor

- fetches
- decodes
- executes

one instruction at a time

# Levels of Interpretation: Instructions

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



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

op=addi r0 r2 10



```
0010001000000001000000000000001010  
0010000000000000100000000000000000  
00000000001000100001100000101010
```



Instruction Set Architecture

ALU, Control, Register File, ...

## High Level Language

- C, Java, Python, Ruby, ...
- 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

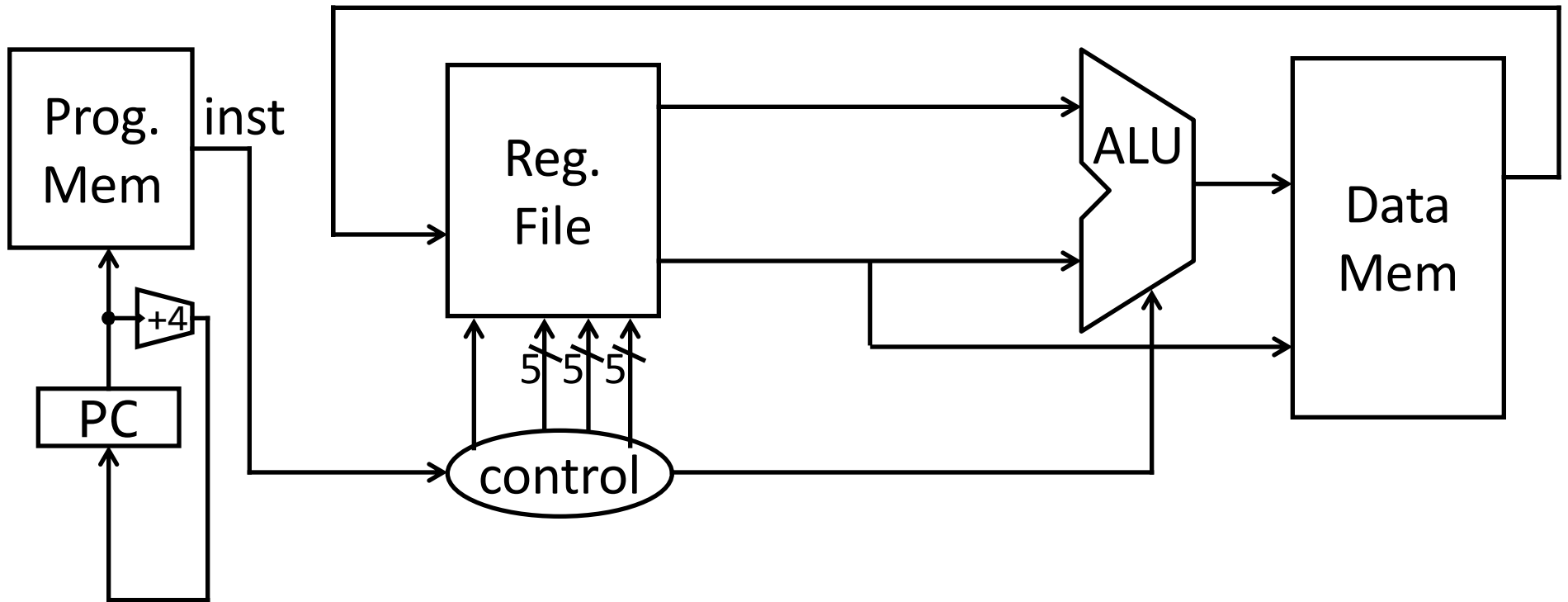
- Data must be in registers to be operated on  
For example:  $\text{array}[x] = \text{array}[y] + \text{array}[z]$   
1 add ?      OR      2 loads, an add, and a store ?
- Keeps HW simple → many RISC ISAs are load/store

# 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.

# Five Stages of MIPS Datapath



← Fetch → ← Decode → ← Execute → ← Memory → ← WB →



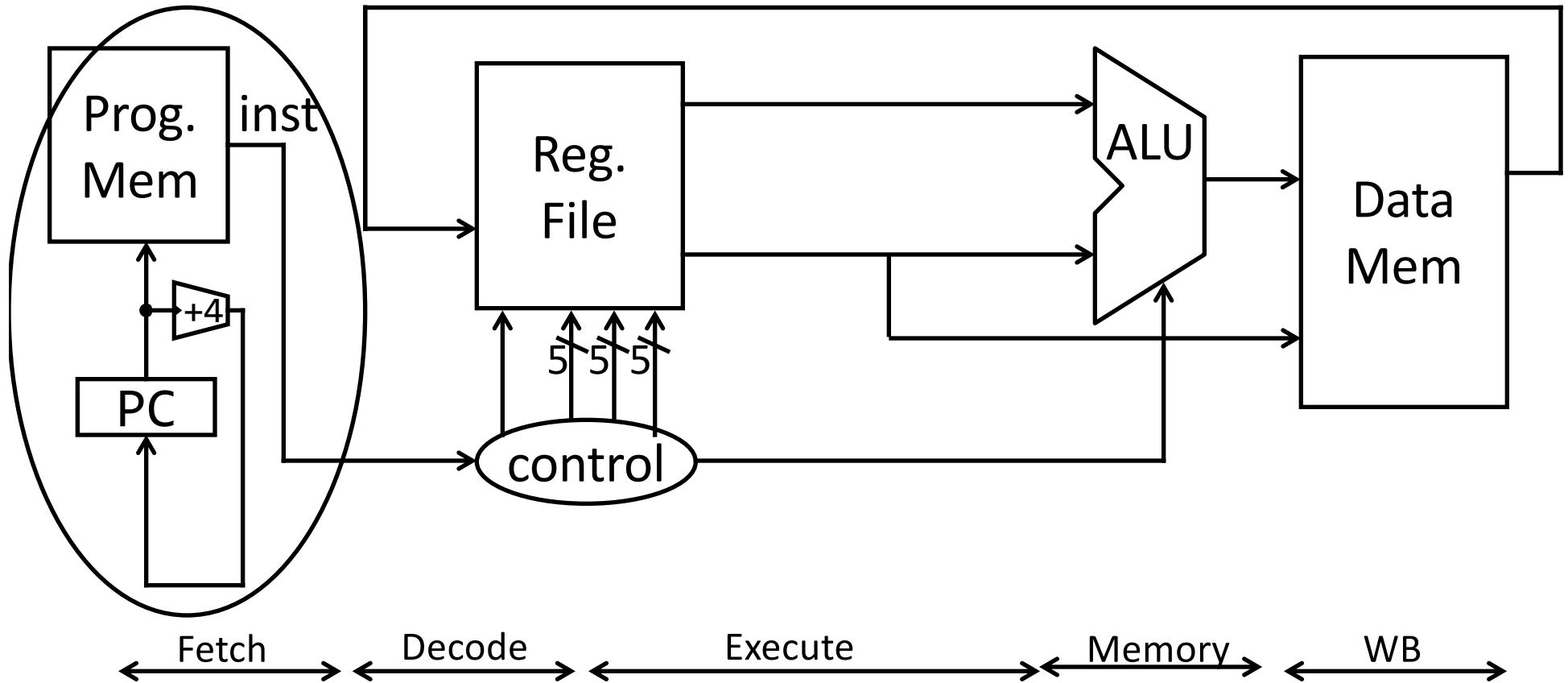
A Single cycle processor – this diagram is not 100% spatial

# Five Stages of MIPS datapath

Basic CPU execution loop

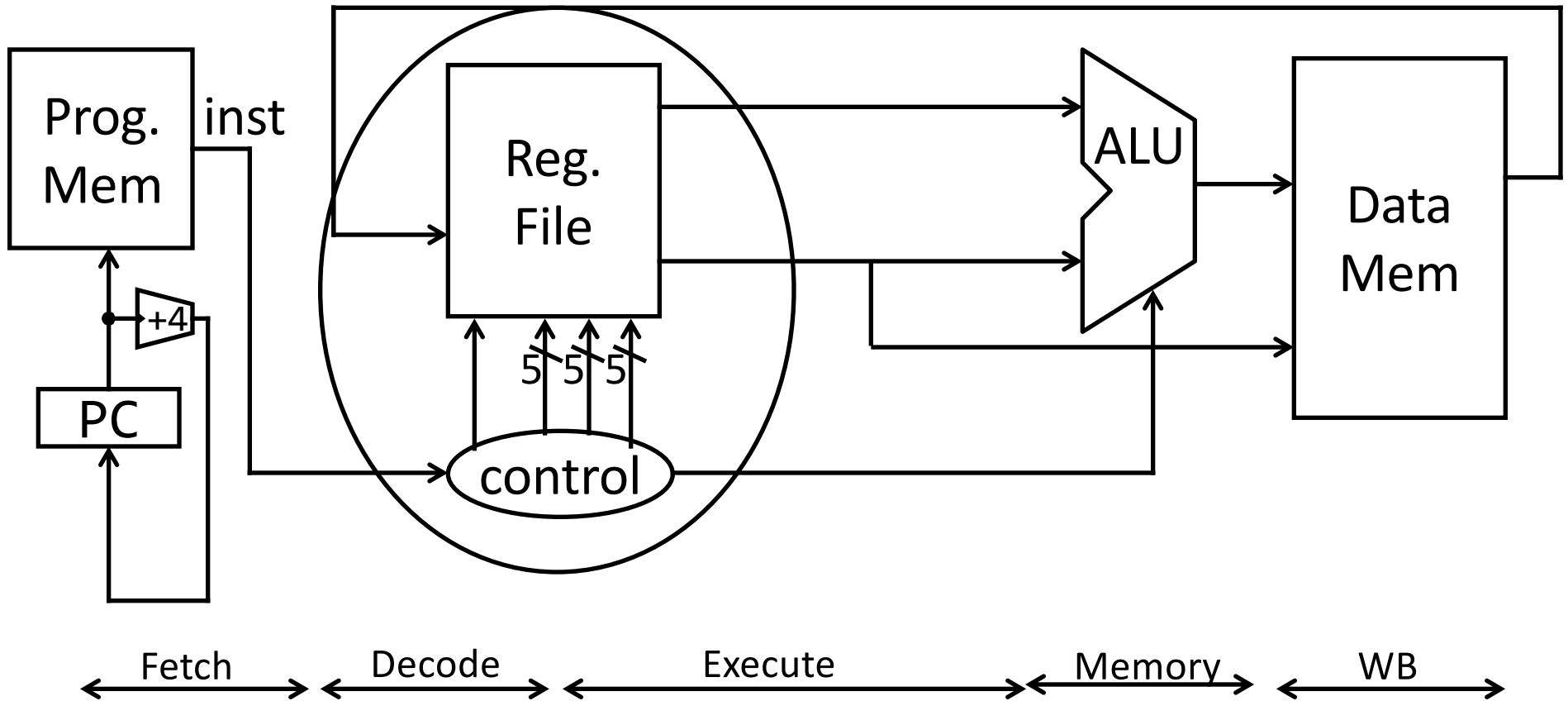
1. Instruction Fetch
2. Instruction Decode
3. Execution (ALU)
4. Memory Access
5. Register Writeback

# Stage 1: Instruction Fetch



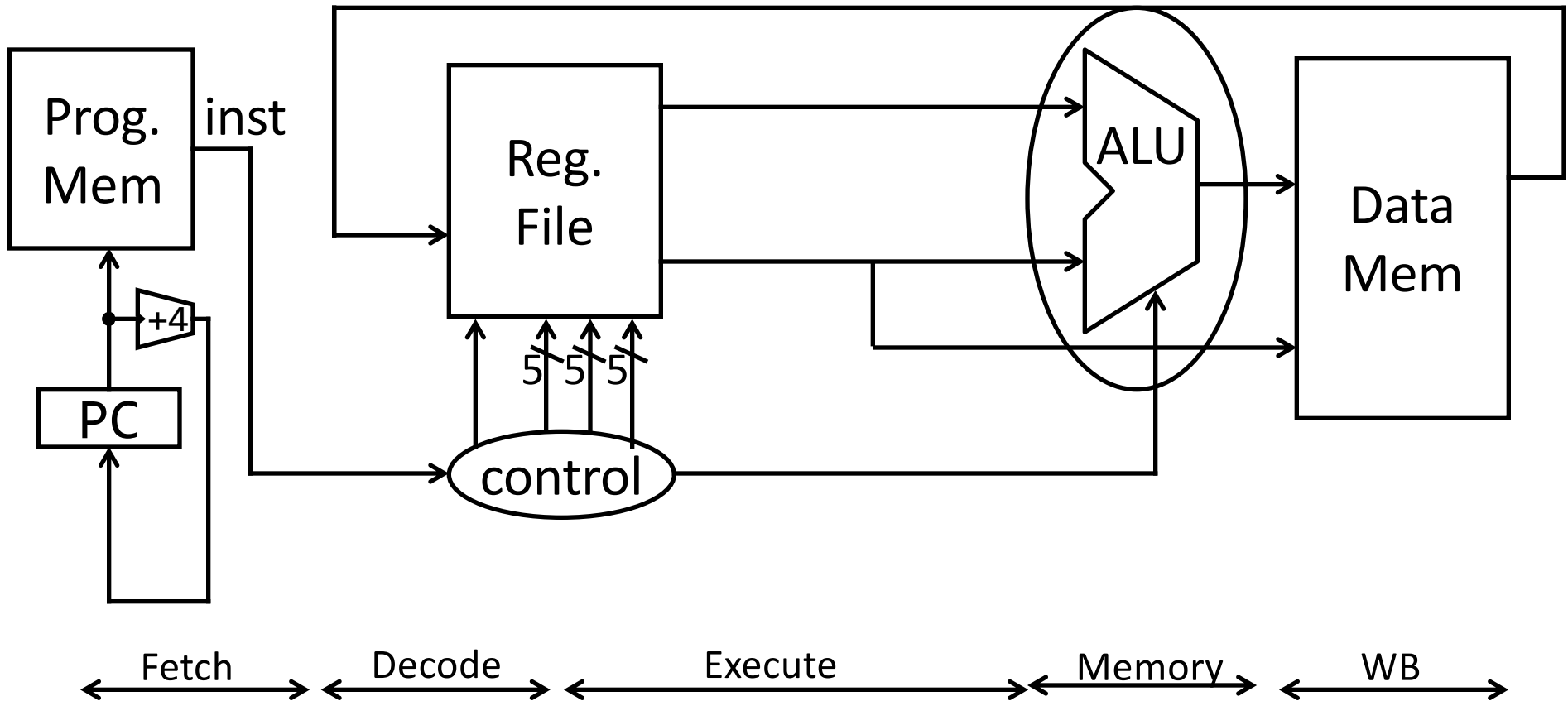
- Fetch 32-bit instruction from memory
- Increment  $PC = PC + 4$

# Stage 2: Instruction Decode



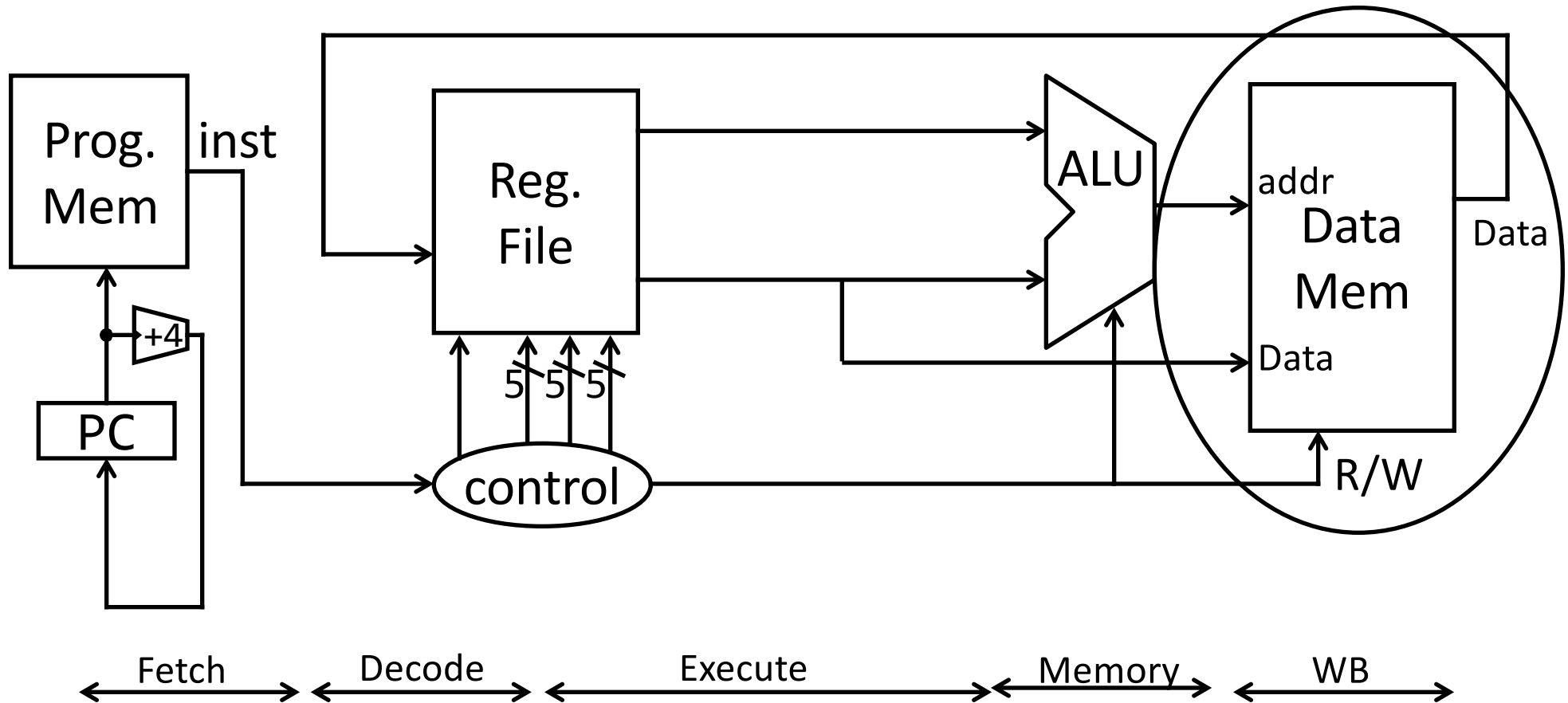
- Gather data from the instruction
- Read opcode; determine instruction type, field lengths
- Read in data from register file  
(0, 1, or 2 reads for `jump`, `addi`, or `add`, respectively)

# Stage 3: Execution (ALU)



- Useful work done here (+, -, \*, /), shift, logic operation, comparison (slt)
- Load/Store? `lw $t2, 32($t3)` → Compute address

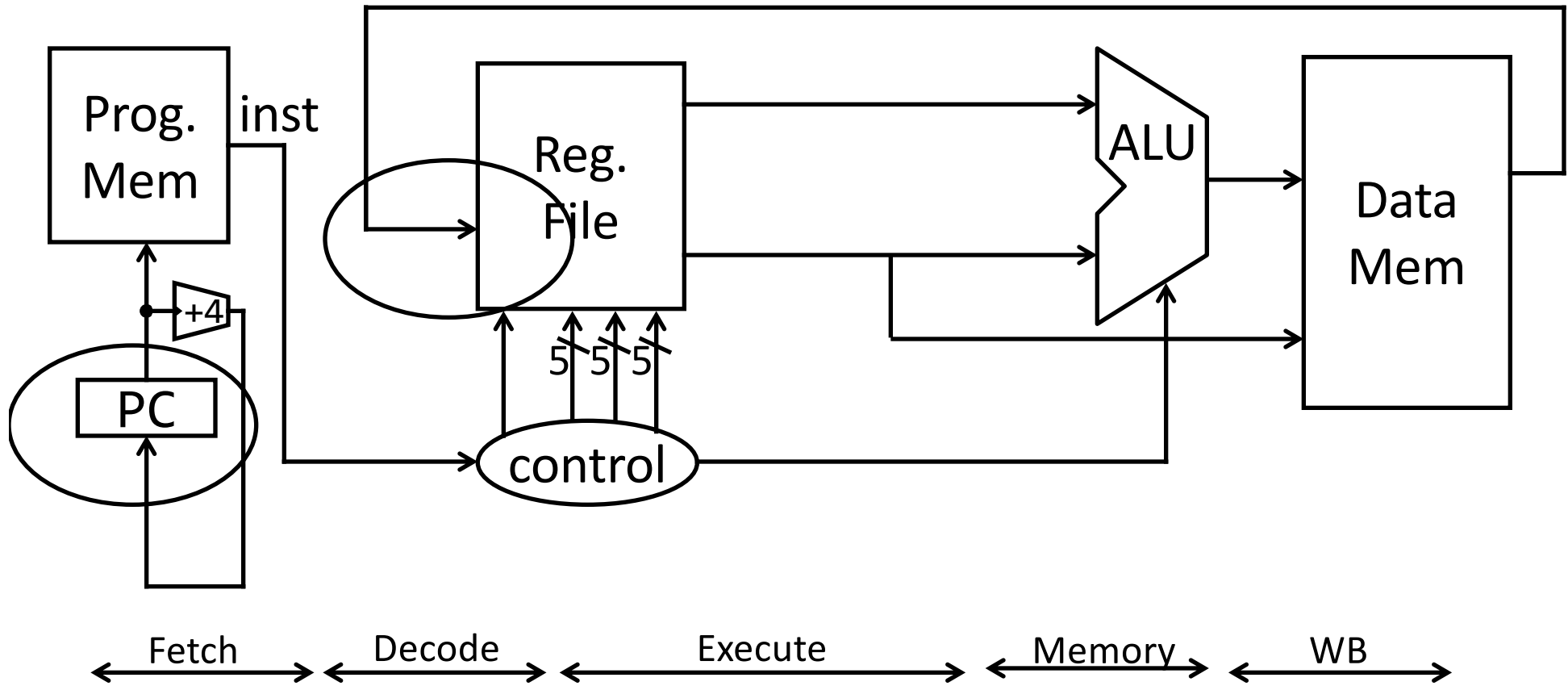
# Stage 4: Memory access



- 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 MIPS instructions are shorter than 32 bits.

# 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

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

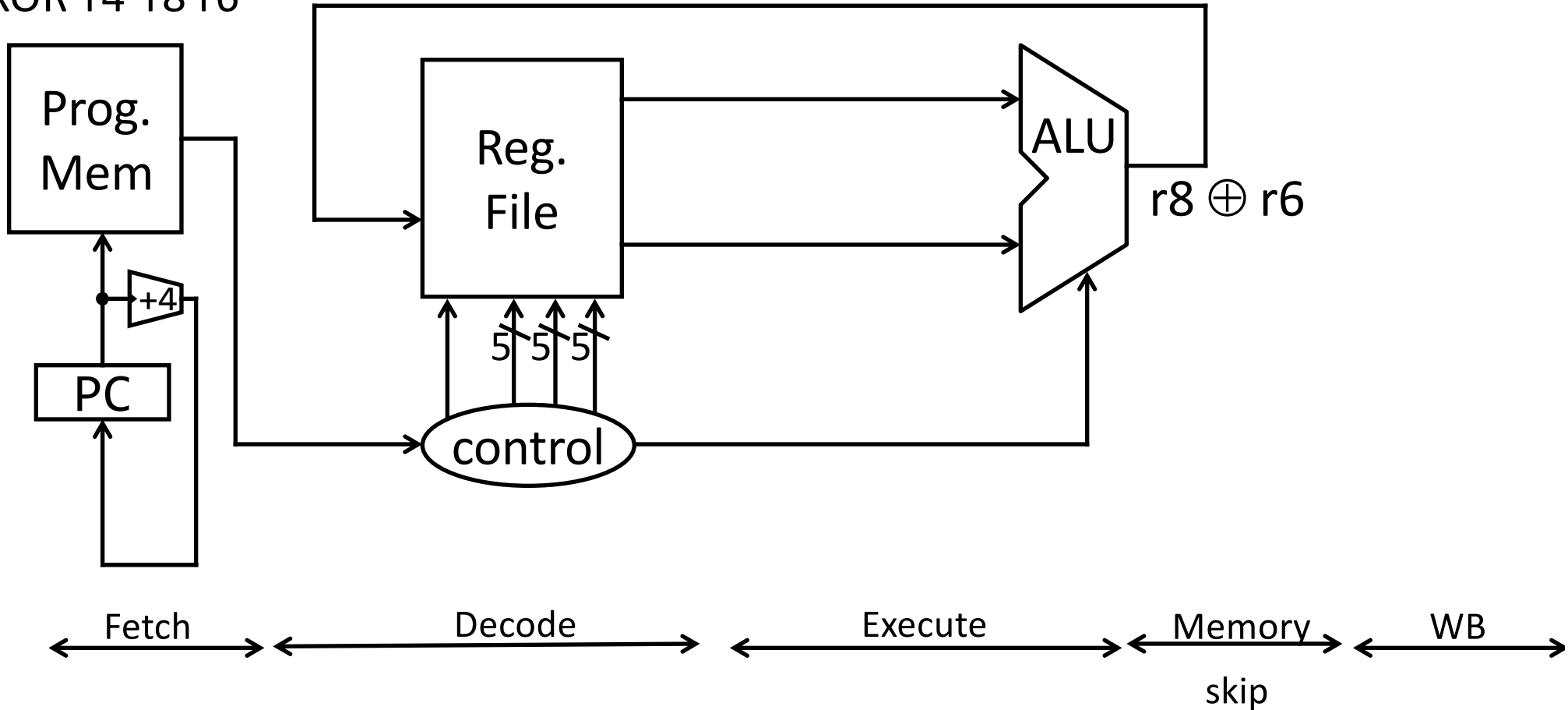
## Control flow

- J-type: fixed offset jumps, jump-and-link
- R-type: register absolute jumps
- I-type: conditional branches: pc-relative addresses



# Arithmetic and Logic

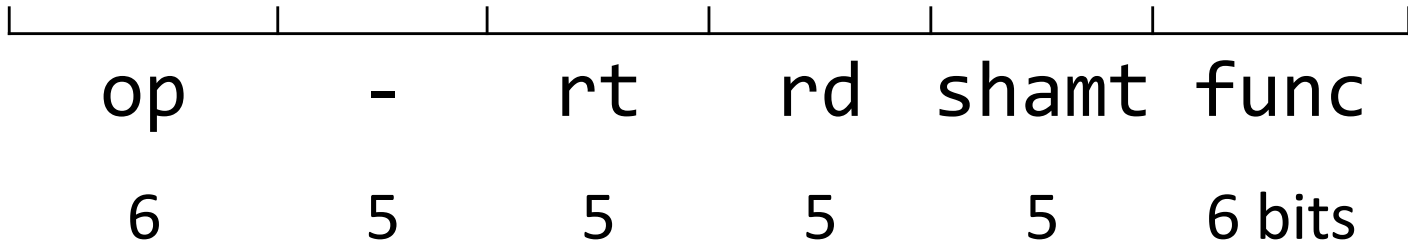
XOR r4 r8 r6



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

# R-Type (2): Shift Instructions

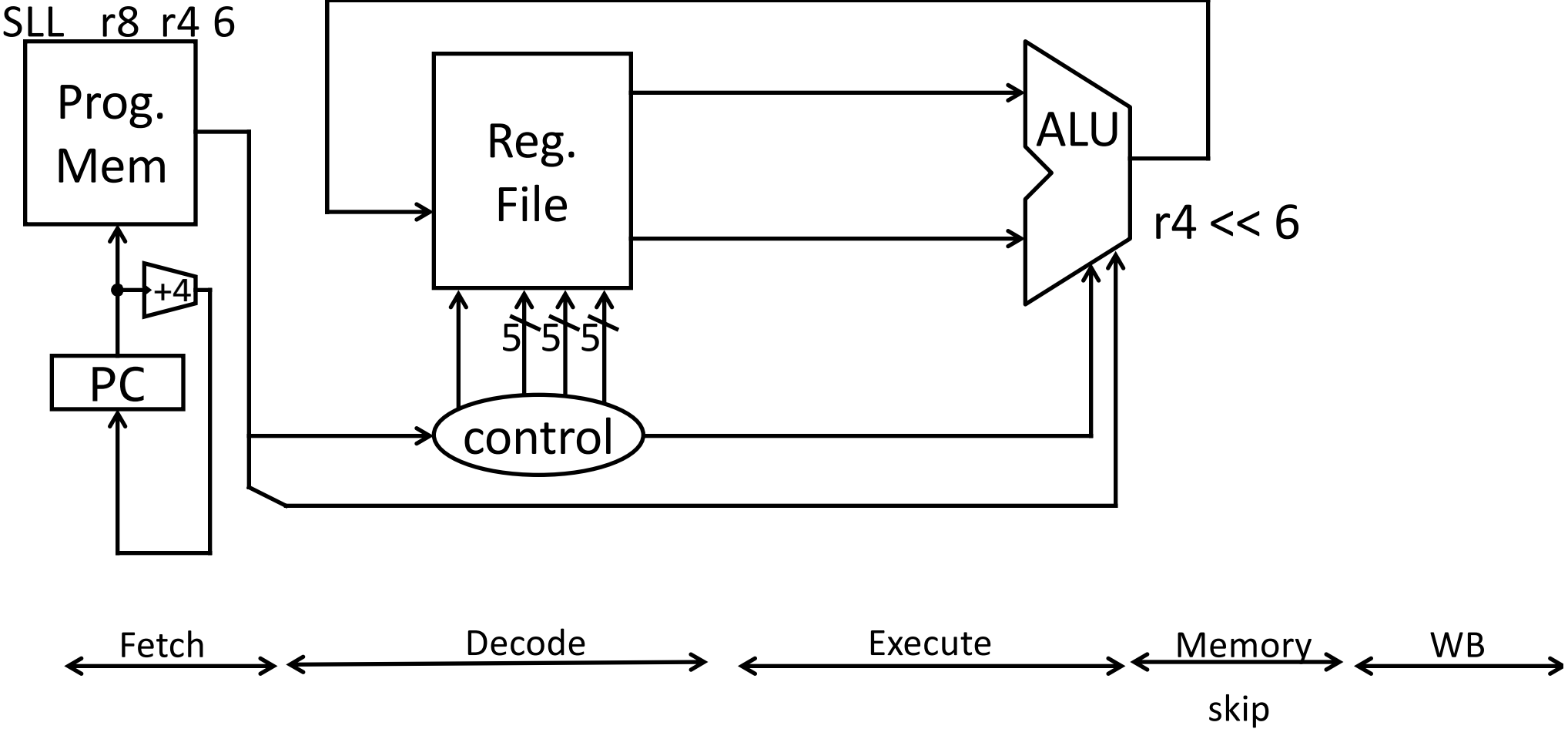
0000000000000001000100000110000000



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.)

example: r8 = r4 \* 64      # SLL r8, r4, 6  
           r8 = r4 << 6

# Shift



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

# SLL r8, r4, 6

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

001001001010010100000000000000101



op          rs          rd          immediate  
6          5          5          16 bits

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	$R[rd] = R[rs]   \text{zero\_extend}(imm)$

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

$r5 += 5$

What if immediate is negative?

$r5 += -1$        $r5 += 65535$

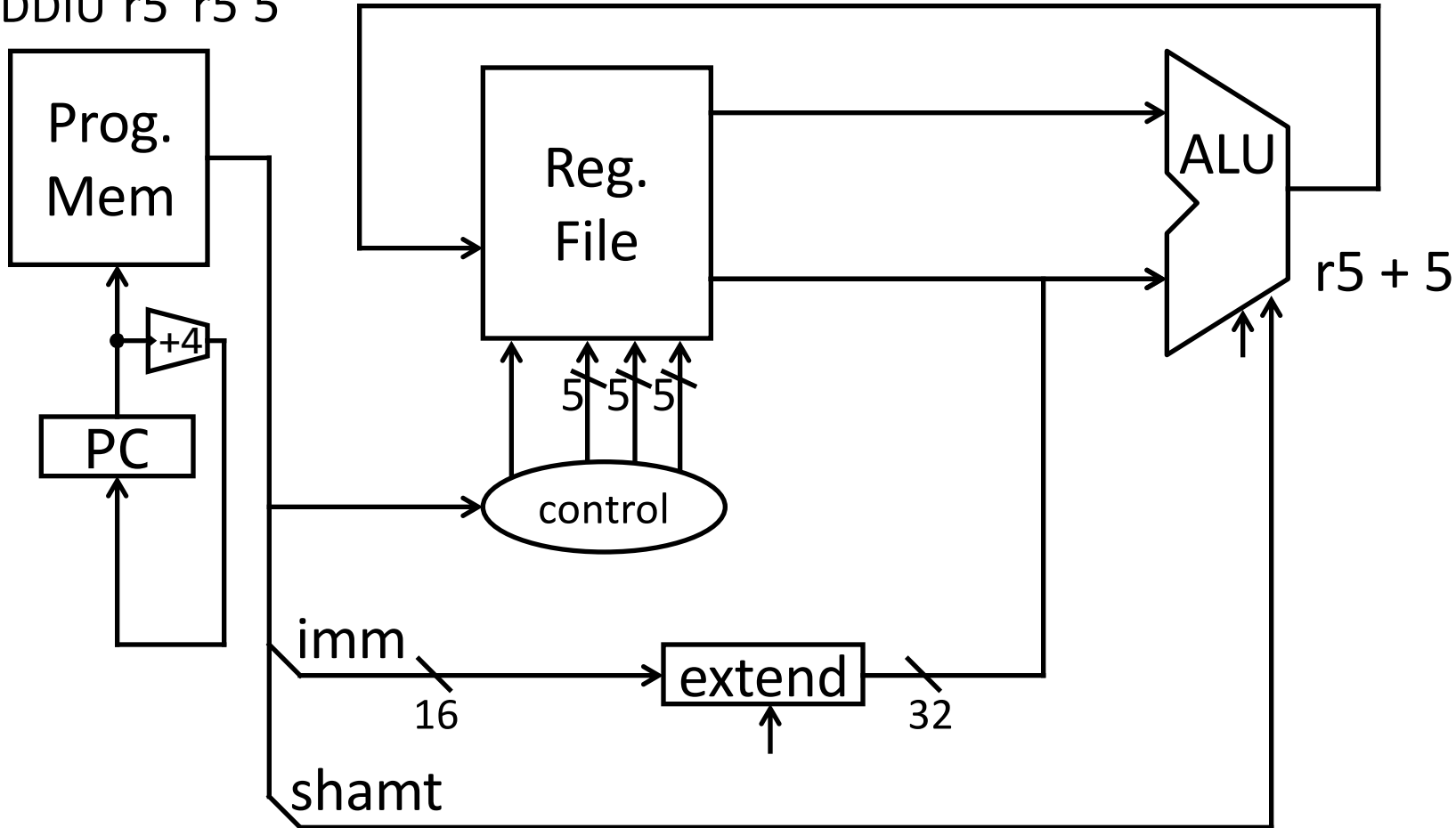
Unsigned means no overflow detection.

The immediate *can* be negative!



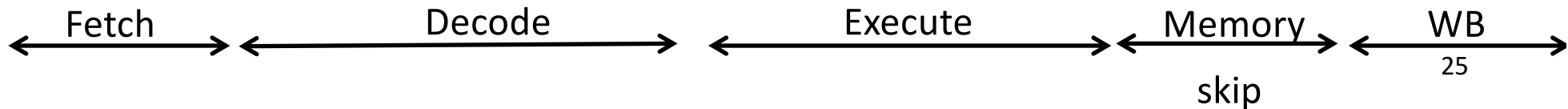
# Arithmetic w/immediates

ADDIU r5 r5 5



Example:  $r5 = r5 + 5$

# ADDIU r5, r5, 5



# iClicker Question

Are you coming to the Homework 1 Review Session?

- (A) Yes, I'm coming tonight (Tuesday).
- (B) Yes, I'm coming tomorrow (Wednesday).
- (C) Yes, but I don't know which night.
- (D) Not sure yet.
- (E) I won't be attending either.

# I-Type (2): "Load" Upper Immediate

0011110000000101000000000000101



op            -            rd            immediate  
6            5            5            16 bits

WORST  
NAME  
EVER!

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

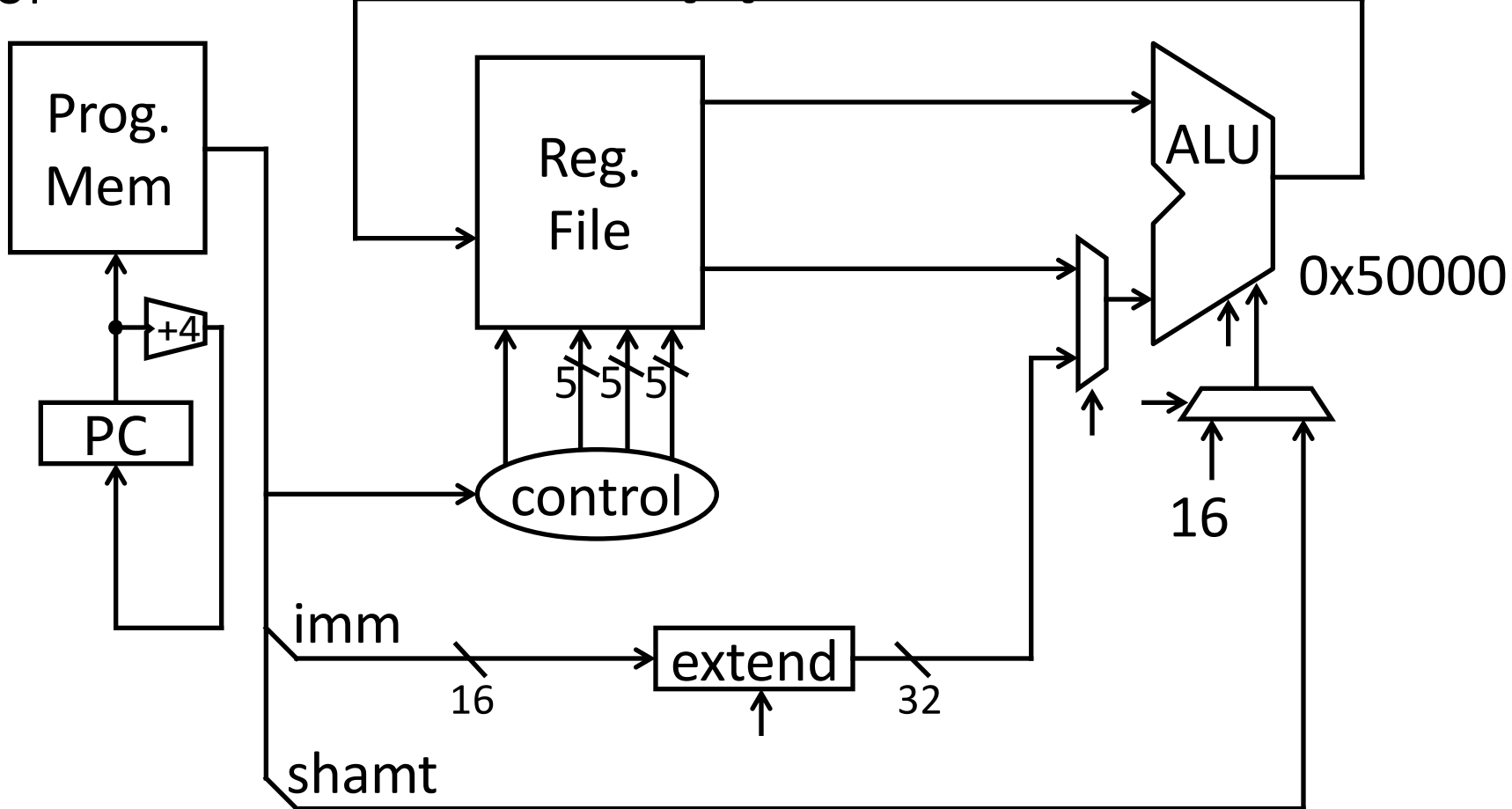
example: r5 = 0x50000    # LUI r5, 5

Example: LUI r5, 0xdead  
          ORI r5, r5 0xbeef

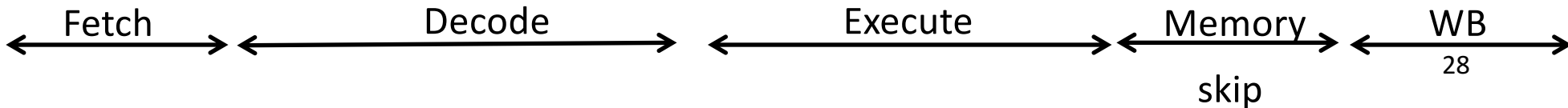
What does r5 = ?

# Load Upper Immediate

LUI r5 5



Example: 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

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

## Control flow

- J-type: fixed offset jumps, jump-and-link
- R-type: register absolute jumps
- I-type: conditional branches: pc-relative addresses

# Memory Layout Options



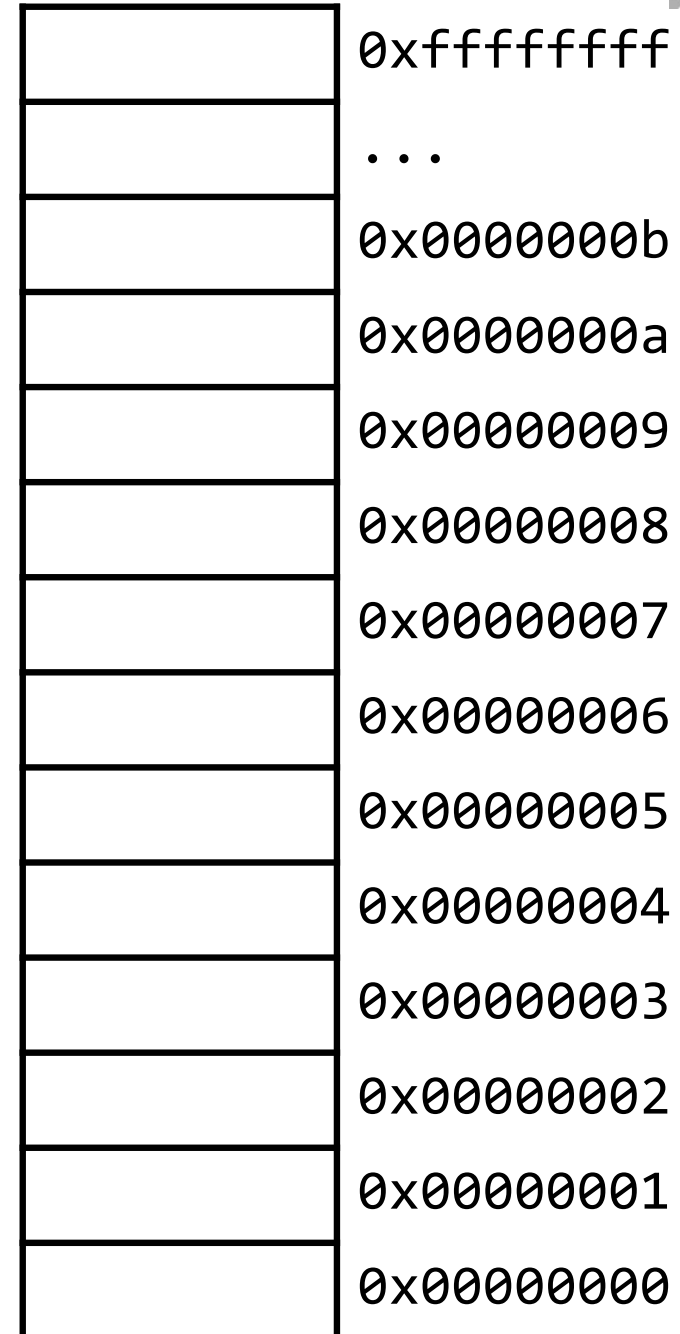
# r5 contains 5 (0x00000005)

SB r5, 0(r0)

SB r5, 2(r0)

SW r5, 8(r0)

Two ways to store a word in memory.

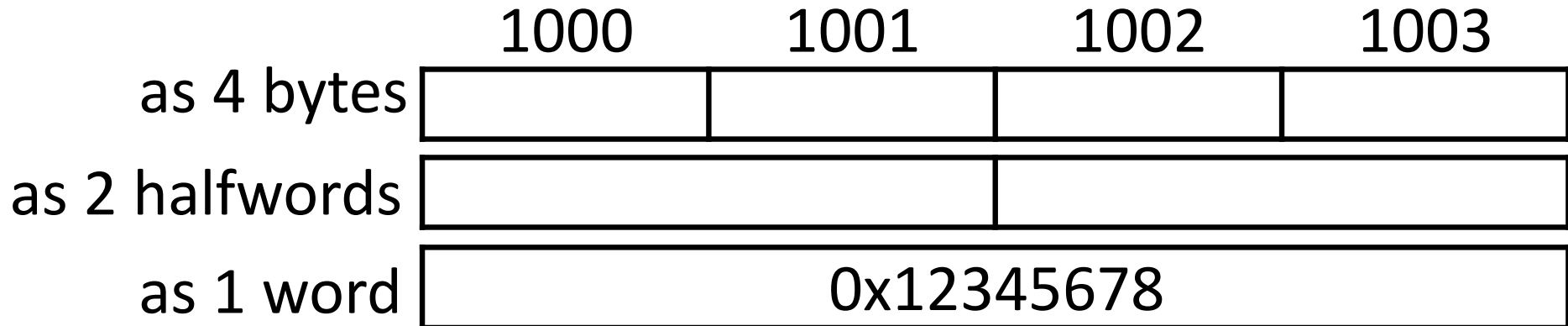


# Endianness

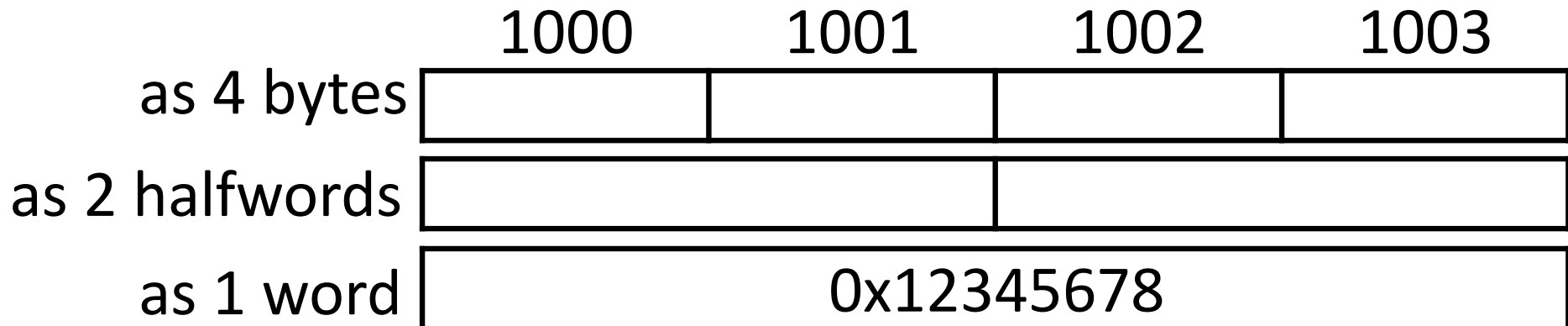


Endianness: Ordering of bytes within a memory word

Little Endian = least significant part first (MIPS, x86)



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



# Endianness

Endianness: Ordering of bytes within a memory word

Little Endian = least significant part first (MIPS, x86)

	1000	1001	1002	1003
as 4 bytes	0x78	0x56	0x34	0x12
as 2 halfwords	0x5678		0x1234	
as 1 word	0x12345678			

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

	1000	1001	1002	1003
as 4 bytes	0x12	0x34	0x56	0x78
as 2 halfwords	0x1234		0x5678	
as 1 word	0x12345678			



# Big Endian Memory Layout



# r5 contains 5 (0x00000005)

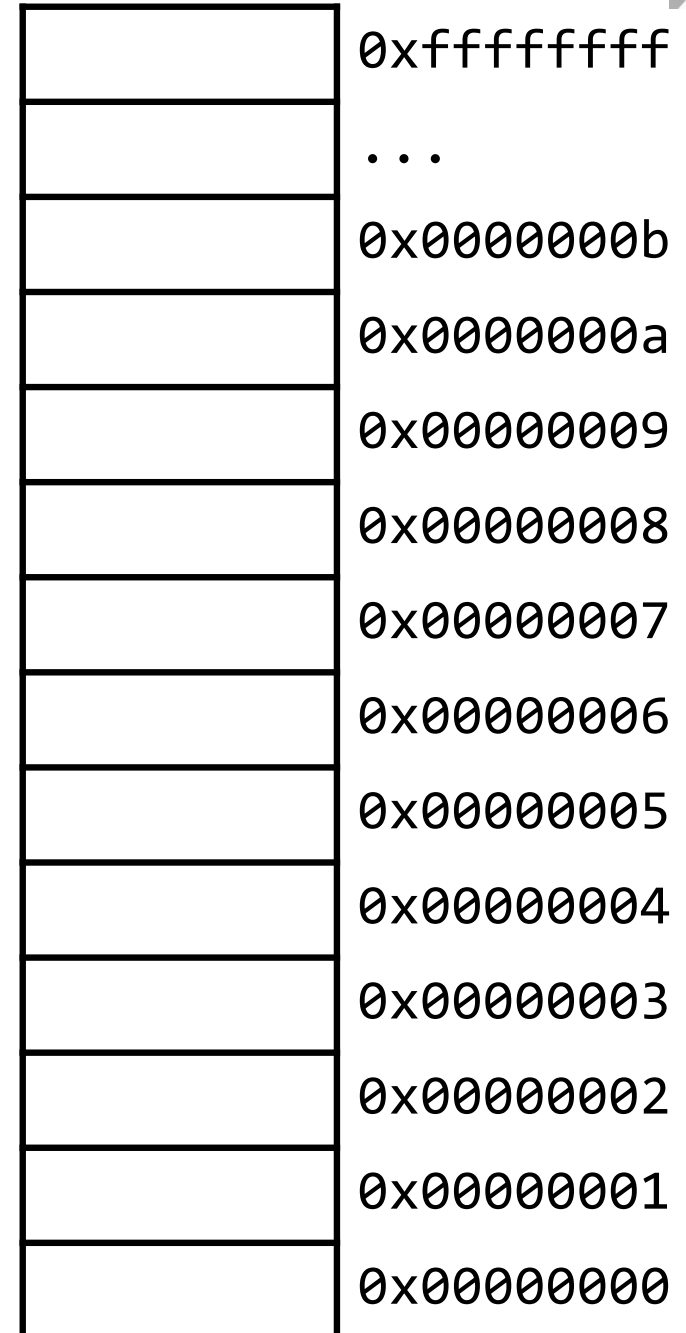
SB r5, 2(r0)

LB r6, 2(r0)

SW r5, 8(r0)

LB r7, 8(r0)

LB r8, 11(r0)



# Big Endian Memory Layout

# r5 contains 5 (0x00000005)

SB r5, 2(r0)

LB r6, 2(r0)

# R[r6] = 0x05

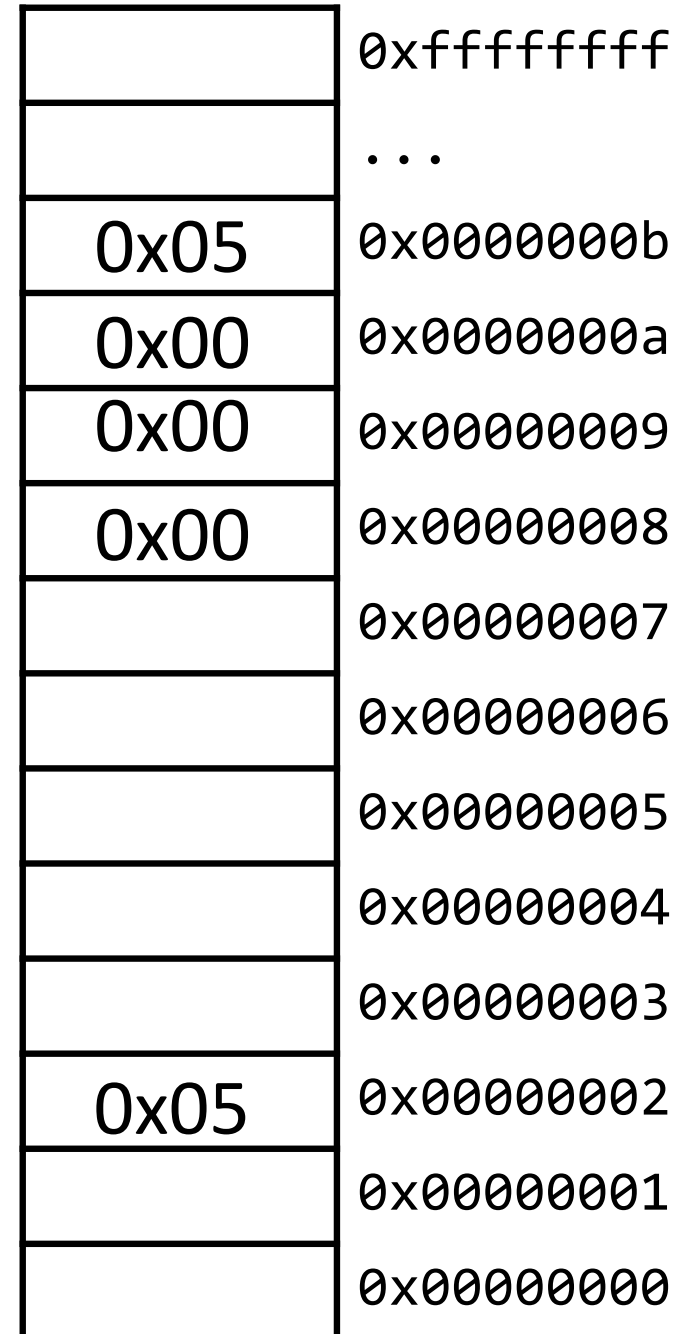
SW r5, 8(r0)

LB r7, 8(r0)

LB r8, 11(r0)

# R[r7] = 0x00

# R[r8] = 0x05



# I-Type (3): Memory Instructions

101011001010000100000000000000100

op      rs      rd      offset  
 6      5      5      16 bits

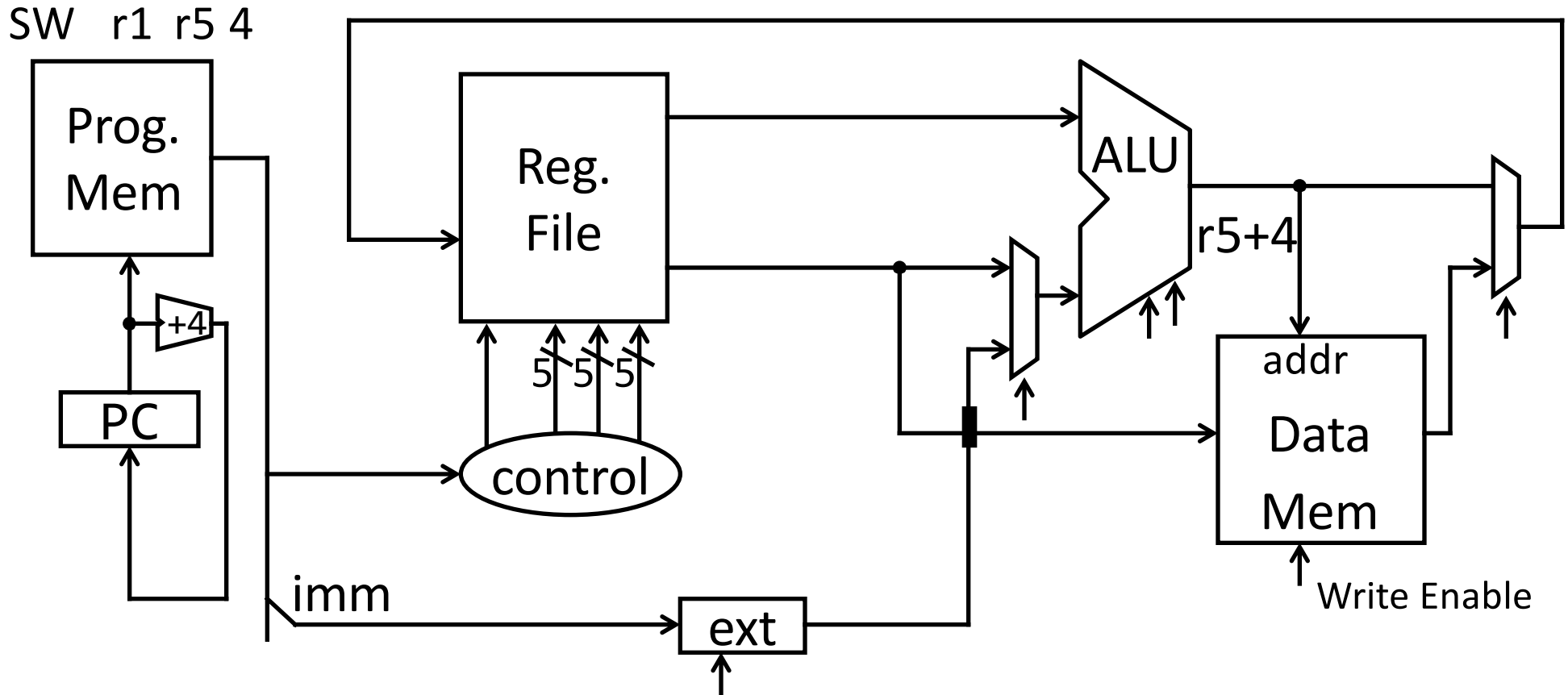
op	mnemonic	description
0x23	LW rd, offset(rs)	R[rd] = Mem[offset+R[rs]]
⇒ 0x2b	SW rd, offset(rs)	Mem[offset+R[rs]] = R[rd]

base + offset  
addressing

signed  
offsets

Example: = Mem[4+r5] = r1      # SW r1, 4(r5)

# Memory Operations



Example:  $= \text{Mem}[4+r5] = r1$  # SW r1, 4(r5)

# More Memory Instructions

101011001010000100000000000000100

op          rs          rd          offset  
 6            5            5            16 bits

op	mnemonic	description
0x20	LB rd, offset(rs)	$R[rd] = \text{sign\_ext}(\text{Mem}[\text{offset}+R[rs]])$
0x24	LBU rd, offset(rs)	$R[rd] = \text{zero\_ext}(\text{Mem}[\text{offset}+R[rs]])$
0x21	LH rd, offset(rs)	$R[rd] = \text{sign\_ext}(\text{Mem}[\text{offset}+R[rs]])$
0x25	LHU rd, offset(rs)	$R[rd] = \text{zero\_ext}(\text{Mem}[\text{offset}+R[rs]])$
0x23	LW rd, offset(rs)	$R[rd] = \text{Mem}[\text{offset}+R[rs]]$
0x28	SB rd, offset(rs)	$\text{Mem}[\text{offset}+R[rs]] = R[rd]$
0x29	SH rd, offset(rs)	$\text{Mem}[\text{offset}+R[rs]] = R[rd]$
0x2b	SW rd, offset(rs)	$\text{Mem}[\text{offset}+R[rs]] = R[rd]$

# 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

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

## Control flow

- J-type: fixed offset jumps, jump-and-link
- R-type: register absolute jumps
- I-type: conditional branches: pc-relative addresses

# J-Type (1): Absolute Jump

00001001000000000000000000000000000000000001

op

immediate

6

26 bits

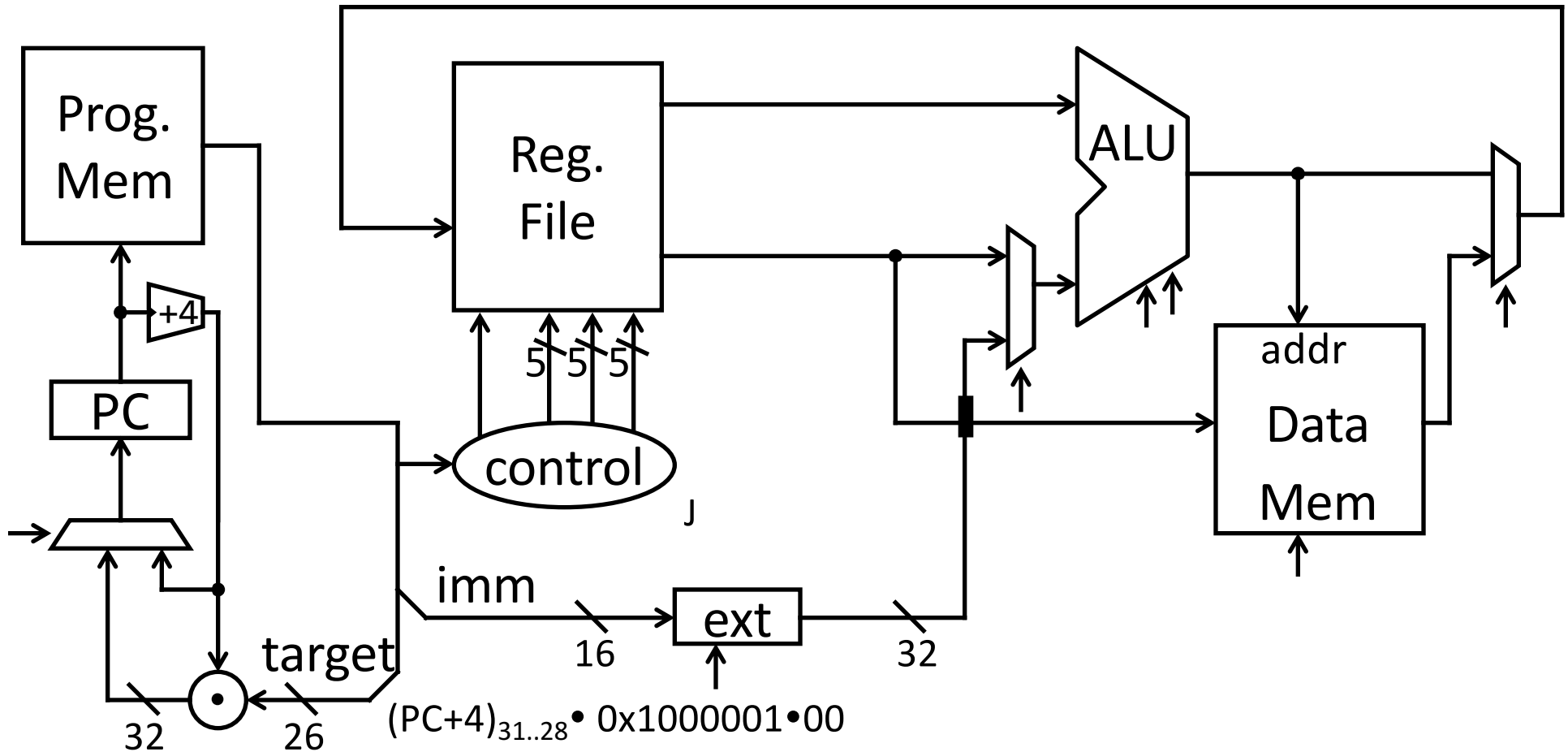
op	Mnemonic	Description	"•" = concatenate
0x2	J target	PC = (PC+4) <sub>31..28</sub> • target • 00	
		(PC+4) <sub>31..28</sub>	target
		4 bits	26 bits
			00
			2 bits

(PC+4)<sub>31..28</sub> 0100000000000000000000000000000000000001 00

## MIPS Quirk:

jump targets computed using *already incremented* PC

# Absolute Jump



$(PC+4)_{31..28} \cdot 0x4000004$

Example:  $PC = (PC+4)_{31..28} \cdot \text{target} \cdot 00 \quad \# J 0x1000001$



# R-Type (3): Jump Register

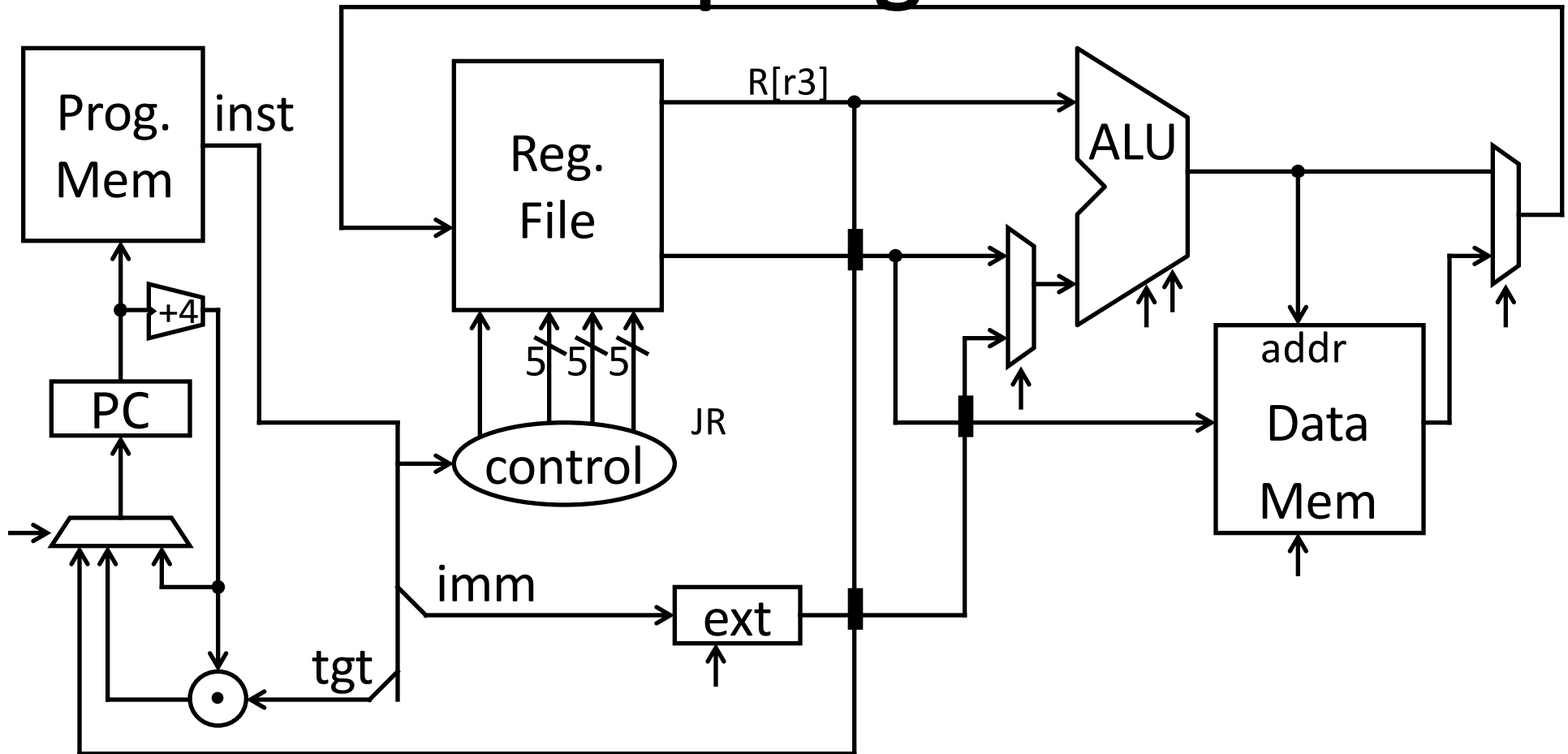
0000000001100000000000000000000001000

op	rs	-	-	-	func
6	5	5	5	5	6 bits

op	func	mnemonic	description
0x0	0x08	JR rs	PC = R[rs]

Example: JR r3

# Jump Register



ex: JR r3

op	func	mnemonic	description
0x0	0x08	JR rs	PC = R[rs]

# iClicker Question

What is a good trait about the Jump Register instruction?

- (A) Since registers are 32 bits, you can specify any address.
- (B) The address you're jumping to is programmable. It doesn't have to be hard-coded in the instruction because it lives in a register.
- (C) It allows you to jump to an instruction with an address ending in something other than 00, which is very useful.
- (D) Both A and B.
- (E) Both A and C.

# Moving Beyond Jumps

Can use Jump or Jump Register instruction to jump to 0xabcd1234

What about a jump based on a condition?

# assume  $0 \leq r3 \leq 1$

if ( $r3 == 0$ ) jump to 0xdecafe00

else jump to 0xabcd1234

# I-Type (4): Branches

0001000010100001000000000000000011

op

rs

rd

offset

6

5

5

16 bits

← signed

op	mnemonic	description
0x4	BEQ rs, rd, offset	if R[rs] == R[rd] then PC = PC+4 + (offset<<2)
0x5	BNE rs, rd, offset	if R[rs] != R[rd] then PC = PC+4 + (offset<<2)

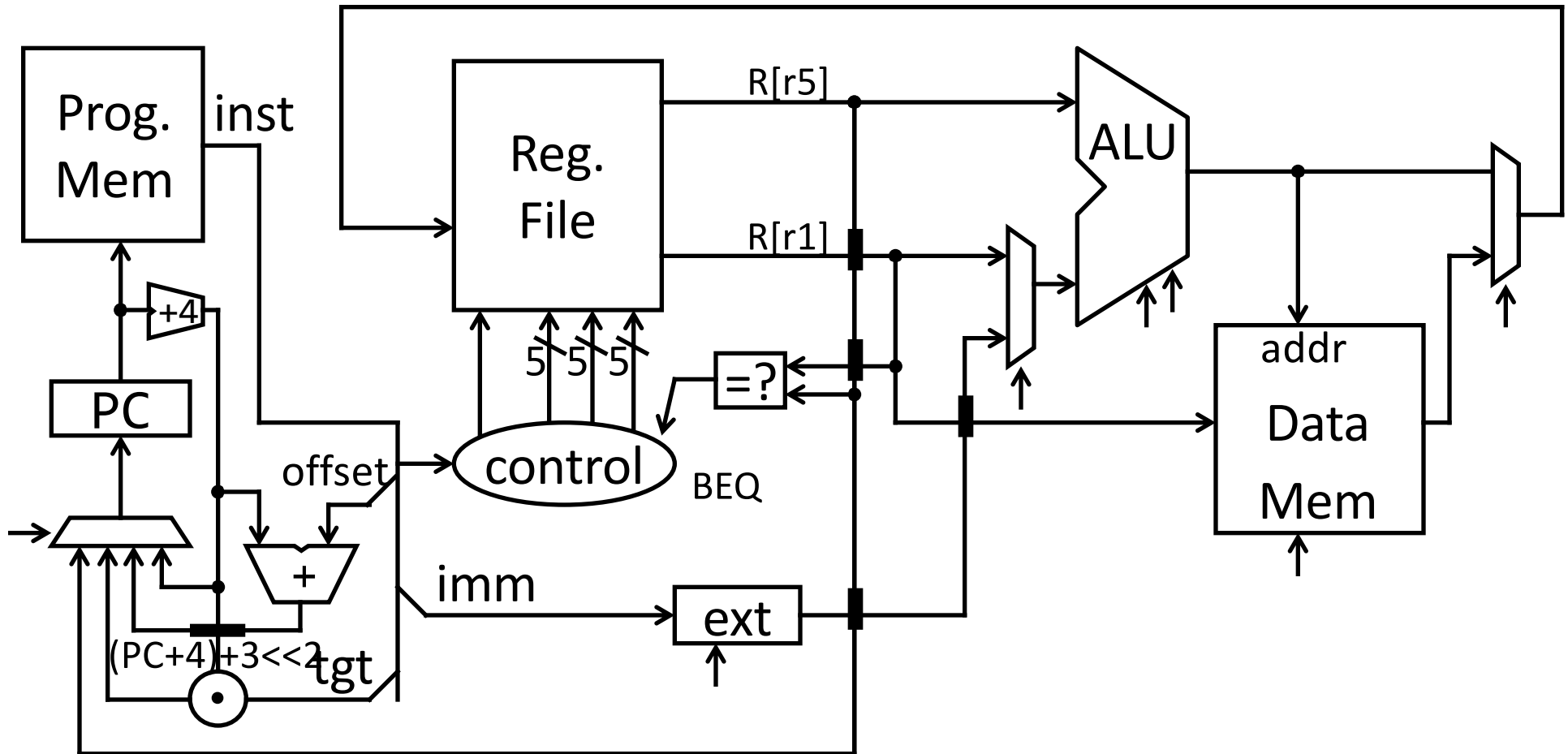
Example: BEQ r5, r1, 3

**if (R[r5]==R[r1])**

**PC = PC+4 + 12 (i.e. 12 == 3<<2)**

A word about all these +'s...

# Control Flow: Branches



ex: BEQ r5, r1, 3

op	mnemonic	description
0x4	BEQ rs, rd, offset	if $R[rs] == R[rd]$ then $PC = PC+4 + (offset \ll 2)$

# I-Type (5): Conditional Jumps

00000100101000010000000000000010

op      rs      subop      offset  
 6 bits   5 bits   5 bits      16 bits

op	subop	mnemonic	description
0x1	0x0	BLTZ rs, offset	if R[rs] < 0 then PC = PC+4+ (offset<<2)
0x1	0x1	BGEZ rs, offset	if R[rs] ≥ 0 then PC = PC+4+ (offset<<2)
0x6	0x0	BLEZ rs, offset	if R[rs] ≤ 0 then PC = PC+4+ (offset<<2)
0x7	0x0	BGTZ rs, offset	if R[rs] > 0 then PC = PC+4+ (offset<<2)

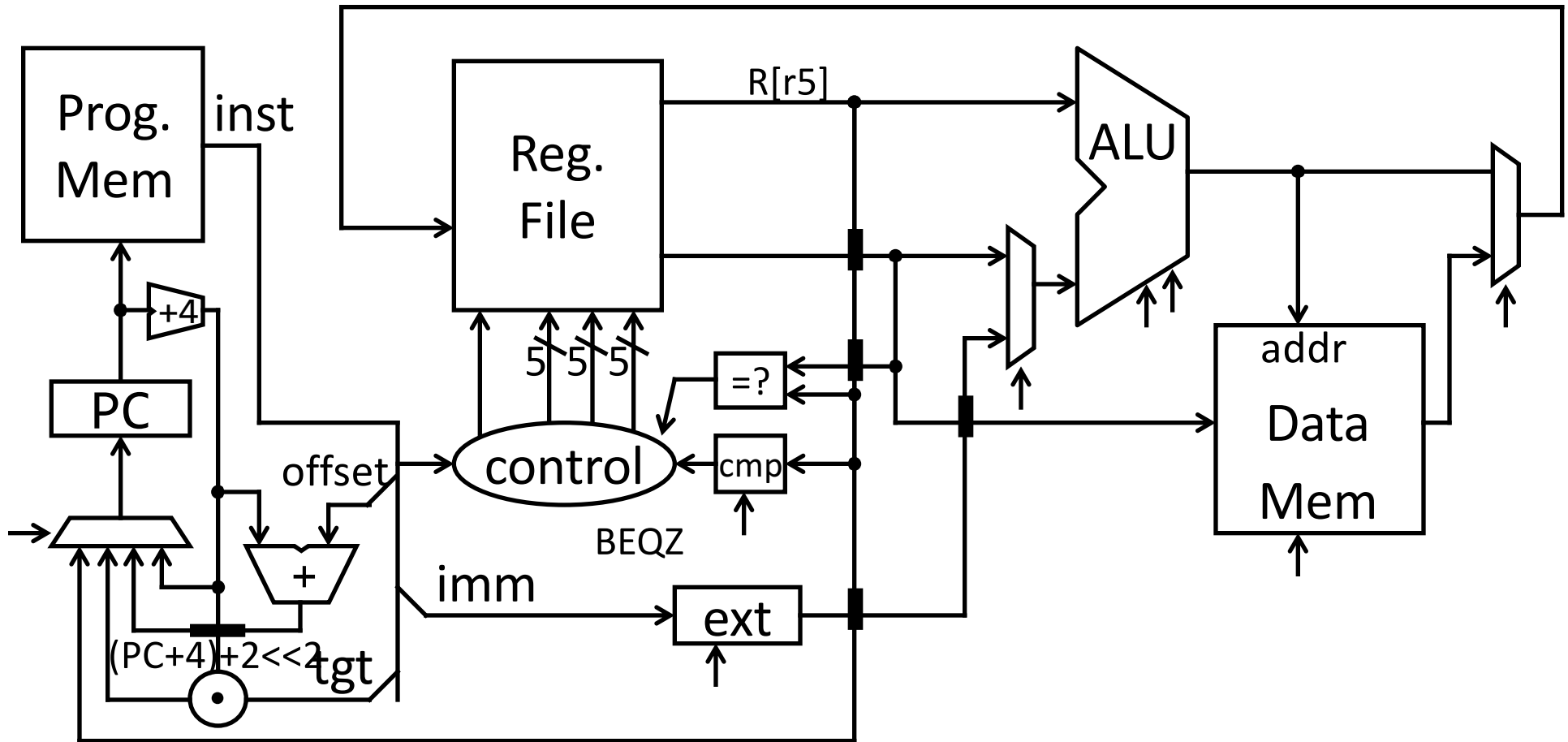
signed

Example: BGEZ r5, 2

**if (R[r5] ≥ 0)**

**PC = PC+4 + 8 (i.e. 8 == 2<<2)**

# Control Flow: More Branches



ex: BGEZ r5, 2

op	subop	mnemonic	description
0x1	0x1	BGEZ rs, offset	if $R[rs] \geq 0$ then $PC = PC+4+ (\text{offset} \ll 2)$



# J-Type (2): Jump and Link

00001101000000000000000000000000000000000001



op

immediate

6 bits

26 bits

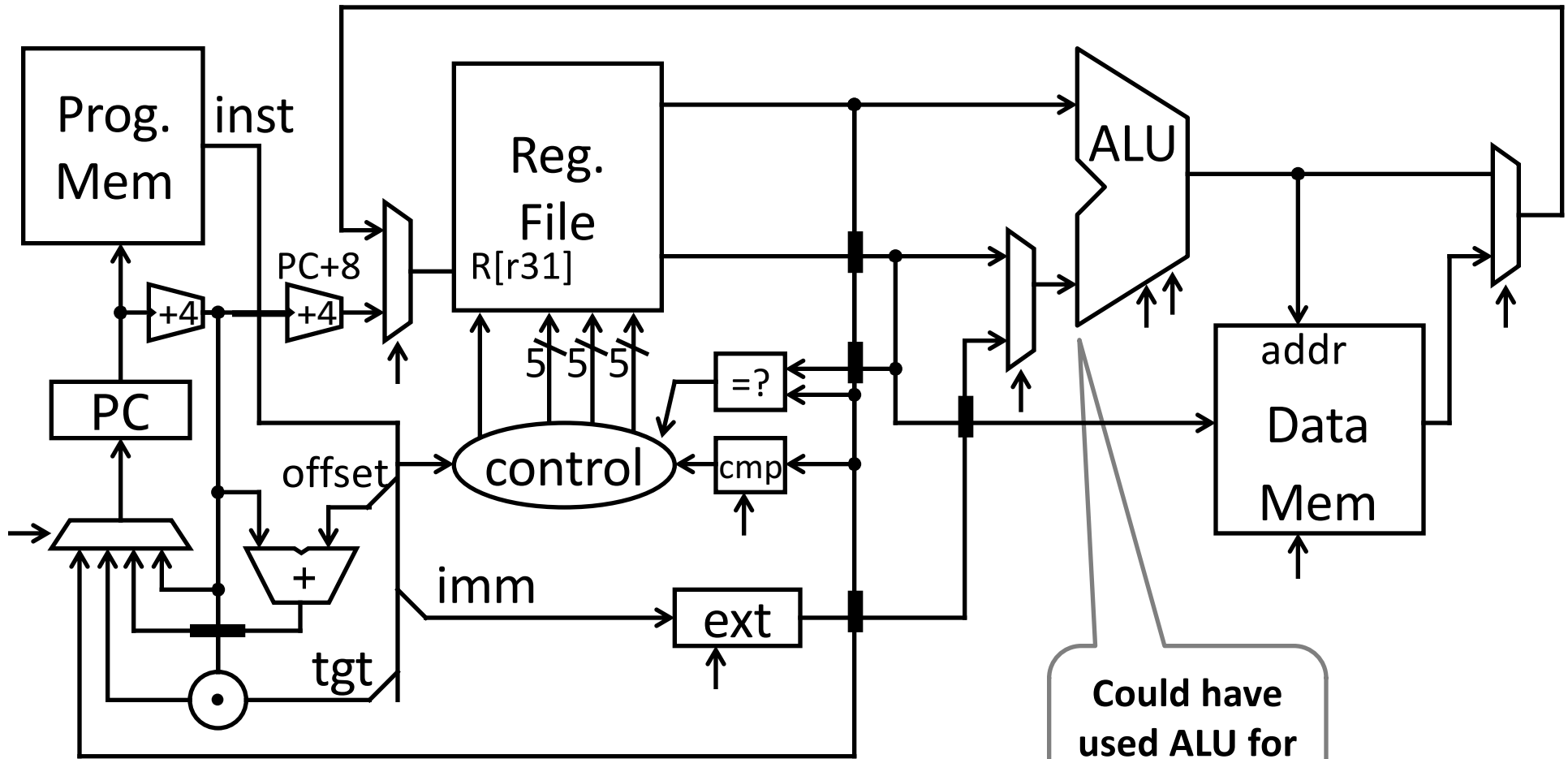
op	mnemonic	description
0x3	JAL target	r31 = PC+8 (+8 due to branch delay slot) PC = (PC+4) <sub>31..28</sub> • target • 00

Discuss later

Why?

Function/procedure calls

# Jump and Link



ex: JAL 0x1000001      r31 = PC+8

PC = (PC+4)<sub>31..28</sub> • 0x4000004  
**description**

op	mnemonic	description
0x3	JAL target	r31 = PC+8 (+8 due to branch delay slot) PC = (PC+4) <sub>31..28</sub> • (target << 2)

Could have used ALU for link add

# 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

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



## Control flow

- J-type: fixed offset jumps, jump-and-link
- R-type: register absolute jumps
- I-type: conditional branches: pc-relative addresses

## Many other instructions possible:

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

# iClicker Question

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

- (A) Gates & Logic
- (B) Finite State Machines
- (C) The MIPS Processor Design
- (D) MIPS Assembly
- (E) Something Else

# Summary

We have all that it takes to build a processor!

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

MIPS 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