Processor

Han Wang

CS3410, Spring 2012

Computer Science

Cornell University

See P&H Chapter 2.16-20, 4.1-4

Announcements

Project 1 Available

Design Document due in one week.

Final Design due in three weeks.

Work in group of 2.

Use your resources

- FAQ, class notes, book, Lab sections, office hours, Piazza

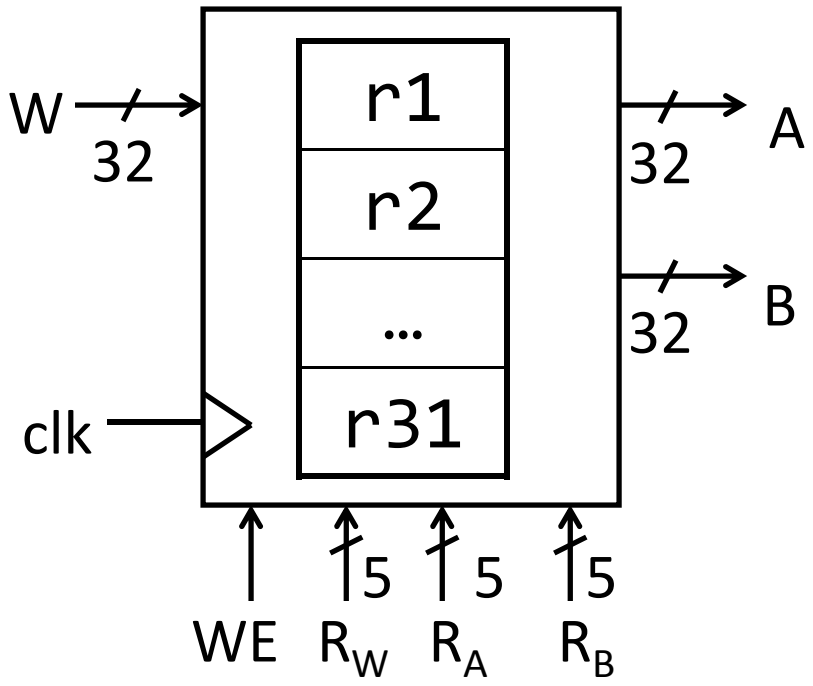
Make sure you

- Registered for class, can access CMS, have a Section, and have a project partner
- Check online syllabus/schedule, review slides and lecture notes, Office Hours.

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



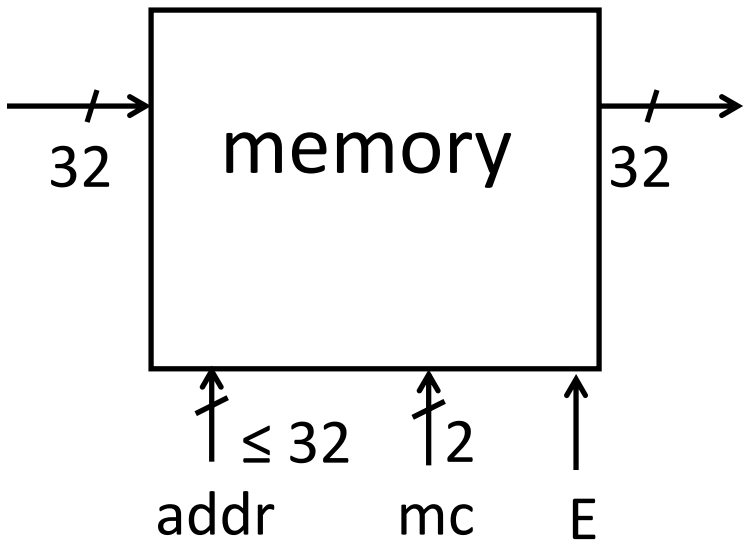
MIPS Register file

- 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 p121.

MIPS Memory

MIPS Memory

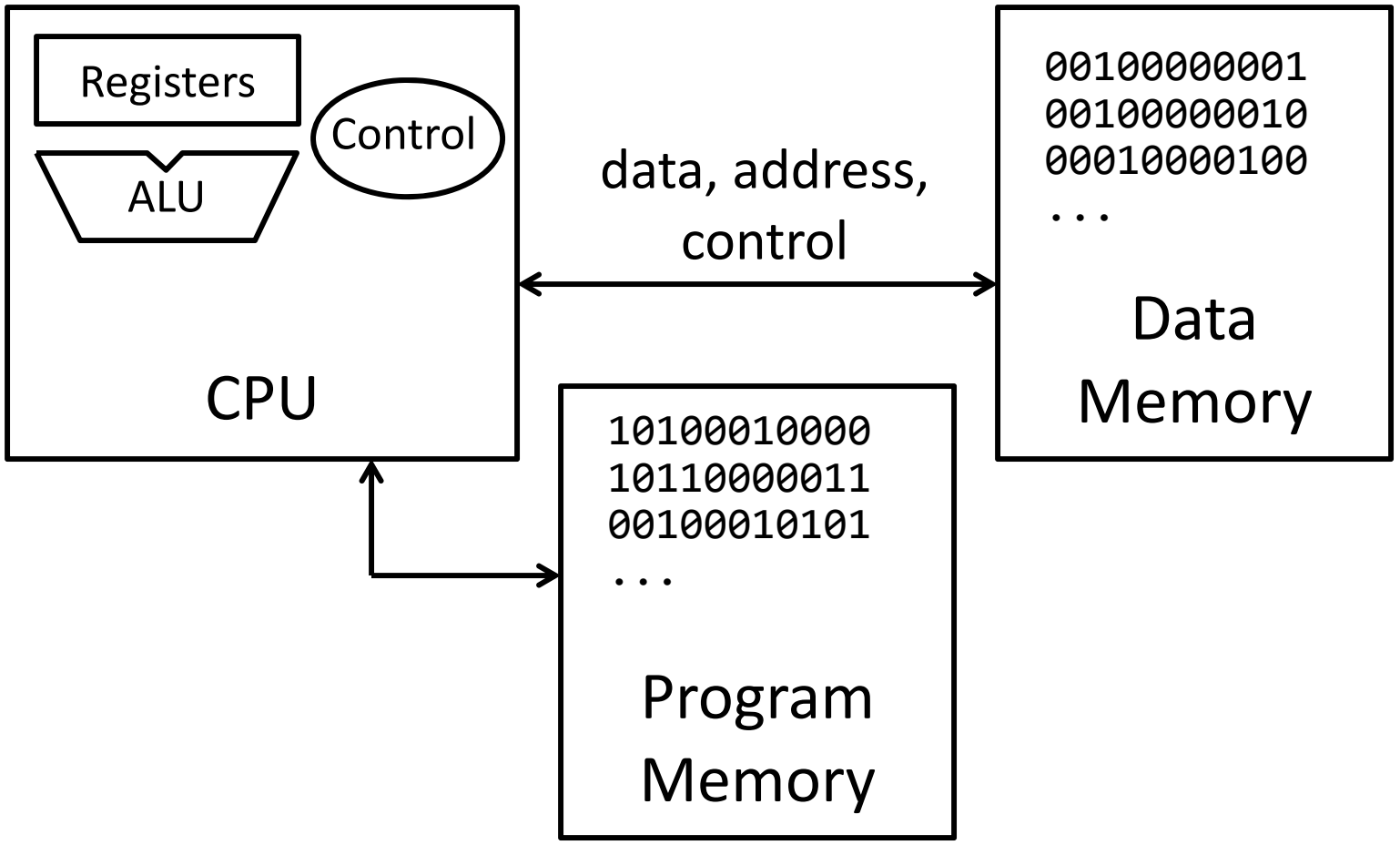
- Up to 32-bit address
- 32-bit data
  (but byte addressed)
- Enable + 2 bit memory control
  - 00: read word (4 byte aligned)
  - 01: write byte
  - 10: write halfword (2 byte aligned)
  - 11: write word (4 byte aligned)



Basic Computer System

Let's build a MIPS CPU

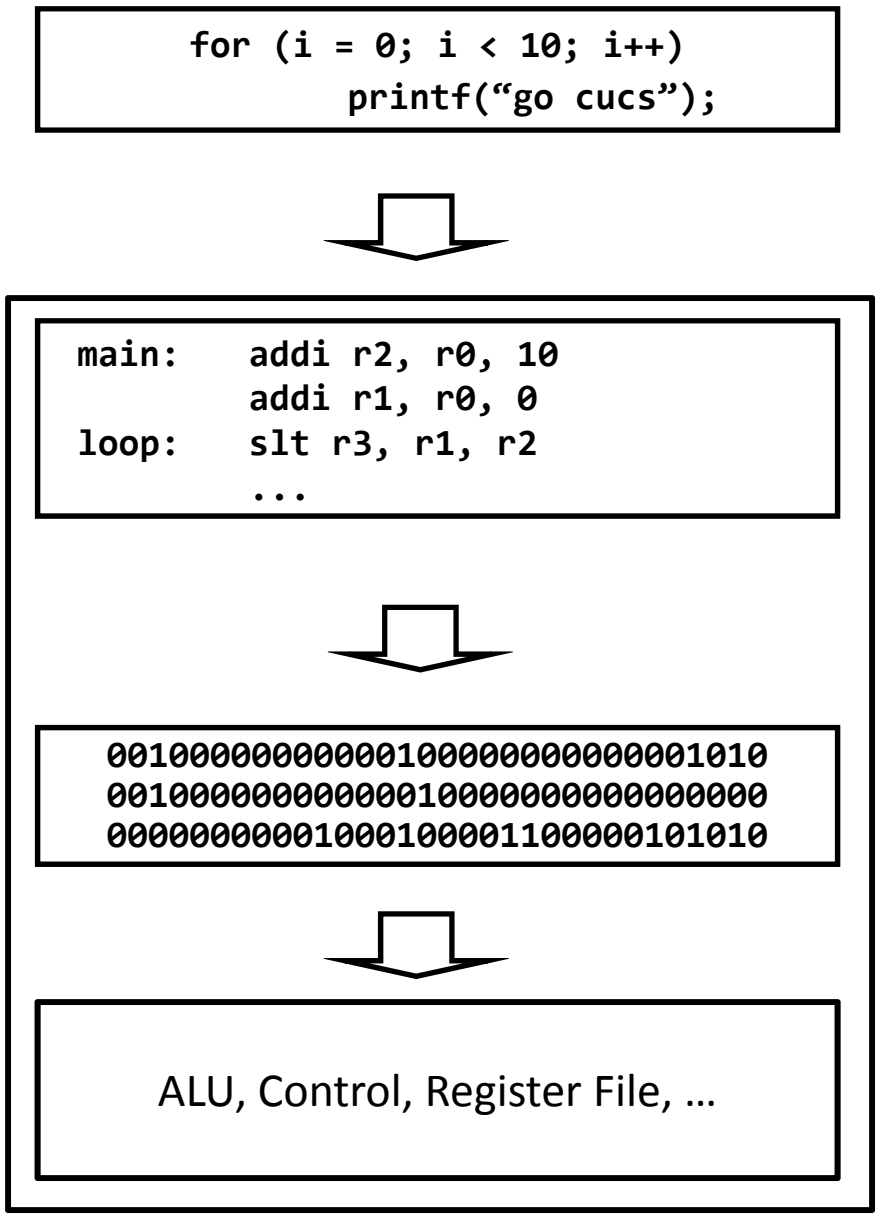
- …but using (modified) Harvard architecture



Agenda

- Stages of datapath
- Datapath Walkthroughs
- Detailed design / Instruction format

Levels of Interpretation



High Level Language

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

Assembly Language

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

Machine Langauge

- Binary-encoded assembly
- Labels become addresses

Machine Implementation

Instruction Set Architecture

- Instruction Set Architecture (ISA)
  - Different CPU architecture specifies different set of instructions. Intel x86, IBM PowerPC, Sun Sparc, MIPS, etc.
- MIPS
  - ≈ 200 instructions, 32 bits each, 3 formats
    - mostly orthogonal
  - all operands in registers
    - almost all are 32 bits each, can be used interchangeably
  - ≈ 1 addressing mode: Mem[reg + imm]
- x86 = Complex Instruction Set Computer (ClSC)
  - > 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]

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

Addition and Subtraction

- Addition
  - add $s0 $s2 $s3 (in MIPS)
  - Equivalent to a = b + c (in C)

    $s0, $s2, $s3 are associated with a, b and c.
- Subtraction
  - sub $s0 $s2 $s3 (in MIPS)
  - Equivalent to d = e – f (in C)

    $s0, $s2, $s3 are associated with d, e and f.

Five Stages of MIPS datapath

Basic CPU execution loop

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

- addu $s0, $s2, $s3
- slti $s0, $s2, 4
- lw $s0, 20($s3)
- 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 (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
  - for addu, read two registers.
  - for addi, read one registers.
  - for jal, read no registers.

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 (4/5)

Stage 4: Memory access

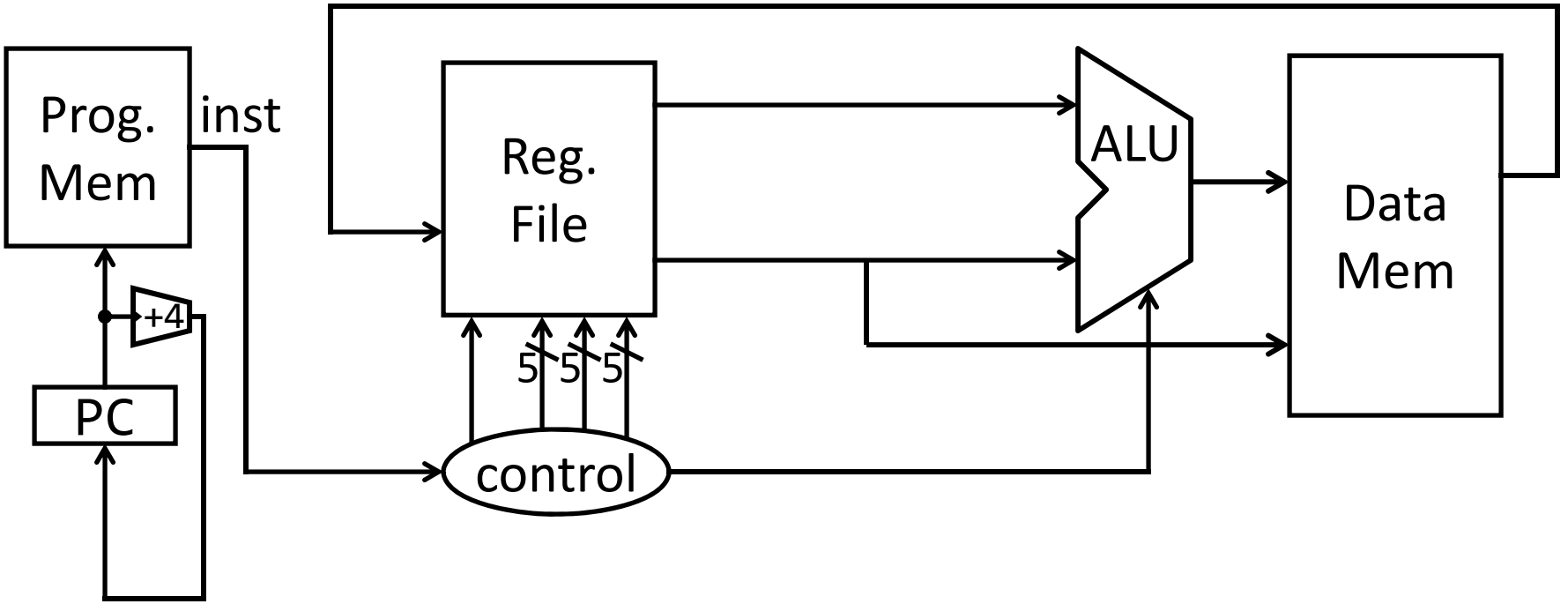
- Used by load and store instructions only.
- Other instructions will skip this stage.
- This stage is expected to be fast, why?

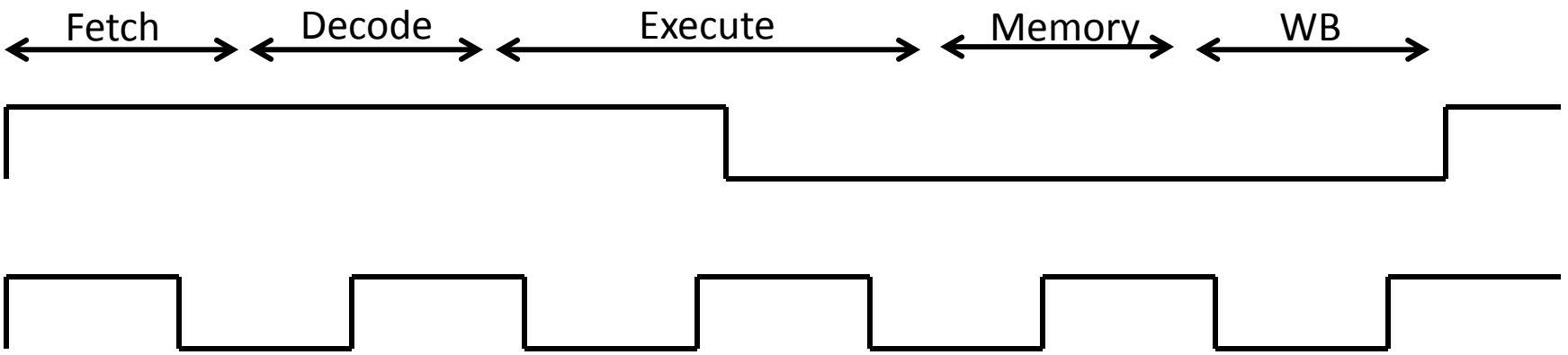
Stages of datapath (5/5)

Stage 5:

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

Datapath and Clocking

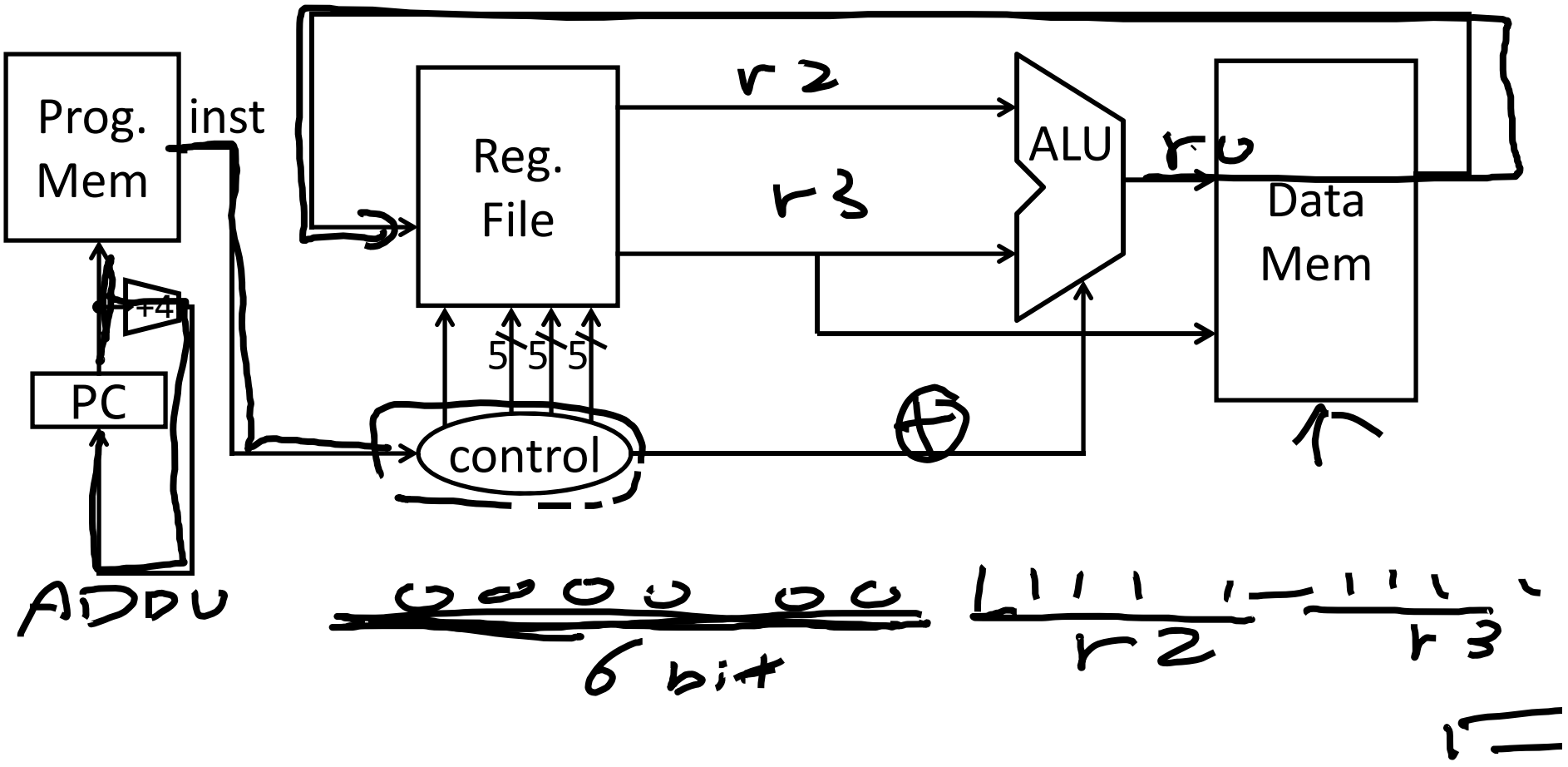




Datapath walkthrough

- addu $s0, $s2, $s3 # s0 = s2 + s3
  - Stage 1: fetch instruction, increment PC
  - Stage 2: decode to determine it is an addu, then read s2 and s3.
  - Stage 3: add s2 and s3
  - Stage 4: skip
  - Stage 5: write result to s0

Example: ADDU instruction



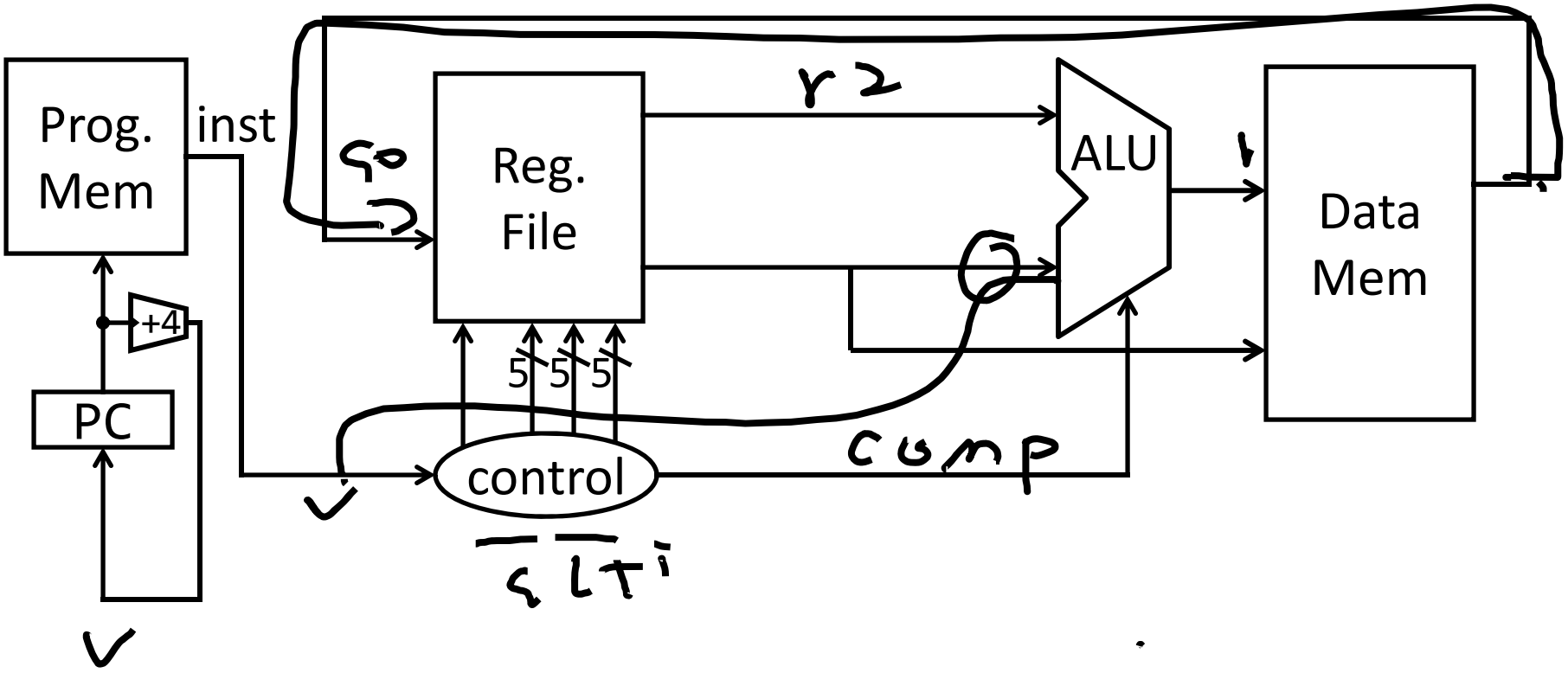
Datapath walkthrough

- slti $s0, $s2, 4

  #if (s2 < 4), s0 = 1, else s0 = 0

  - Stage 1: fetch instruction, increment PC.
  - Stage 2: decode to determine it is a slti, then read register s2.
  - Stage 3: compare s2 with 4.
  - Stage 4: do nothing
  - Stage 5: write result to s0.

Example: SLTI instruction

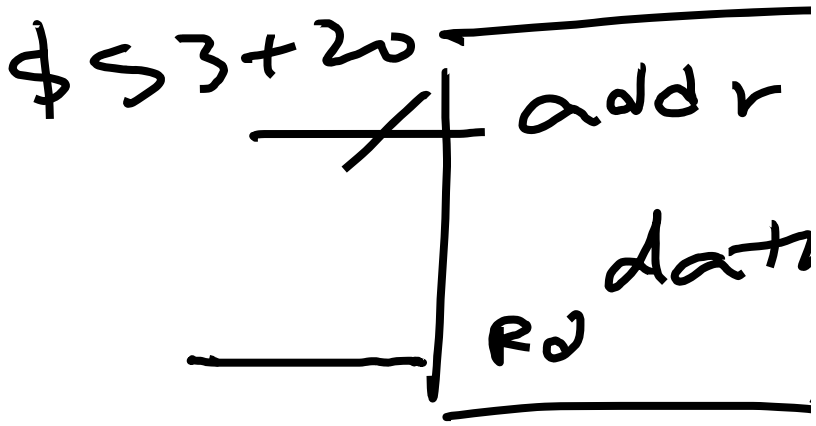


Datapath walkthrough

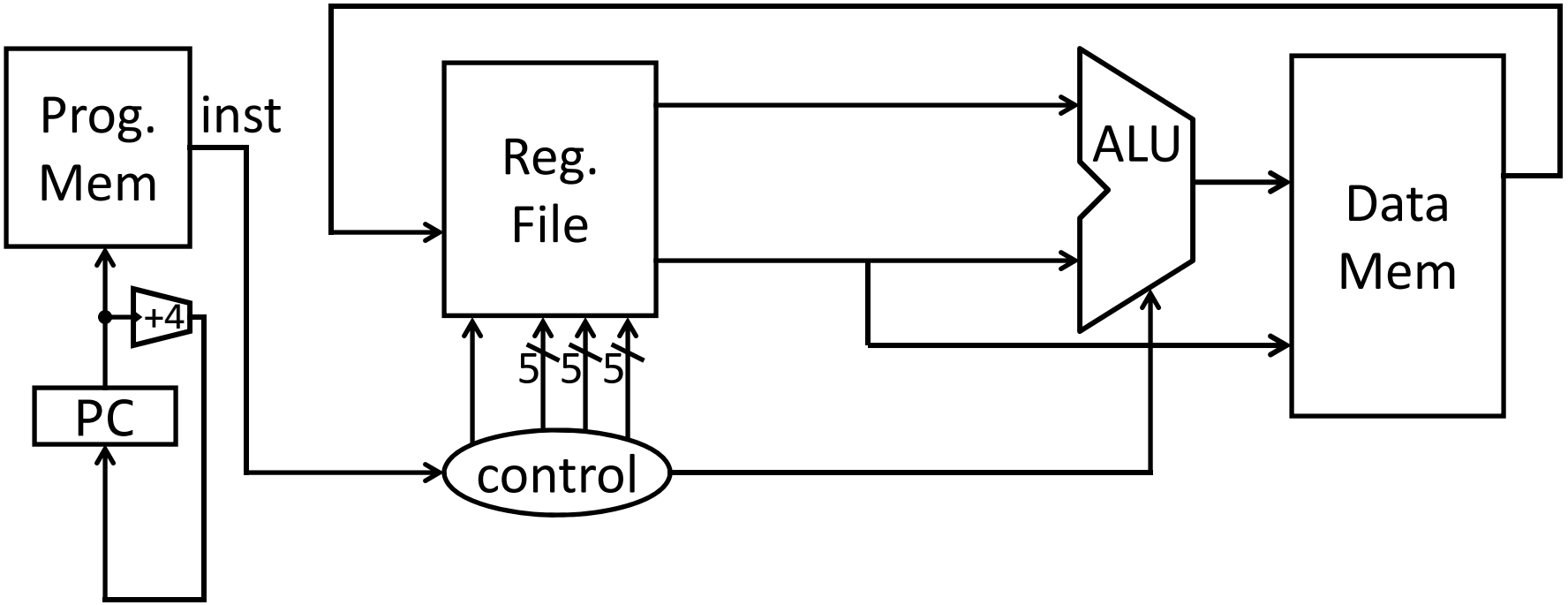
- lw $s0, 20($s3) # s0 = Mem[s3+20]

  R[rt] = M[R[rs] + SignExtImm]

  - Stage 1: fetch instruction, increment PC.
  - Stage 2: decode to determine it is lw, then read register s3
  - Stage 3: add 20 to s3 to compute address.
  - Stage 4: access memory, read value at memory address s3+20
  - Stage 5: write value to s0.



Example: LW instruction



Datapath walkthrough

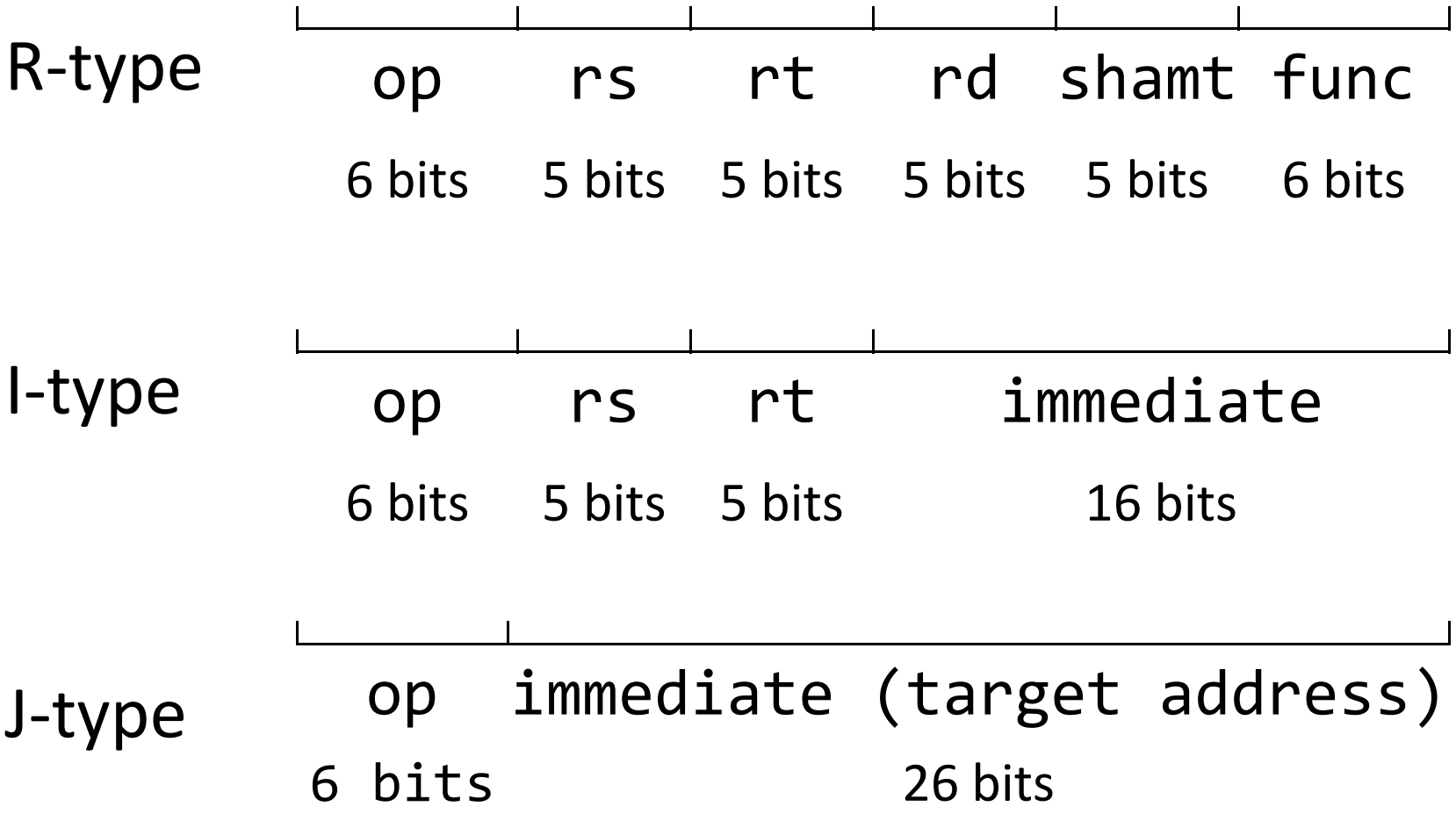
- j 0xdeadbeef

  PC = Jumpaddr

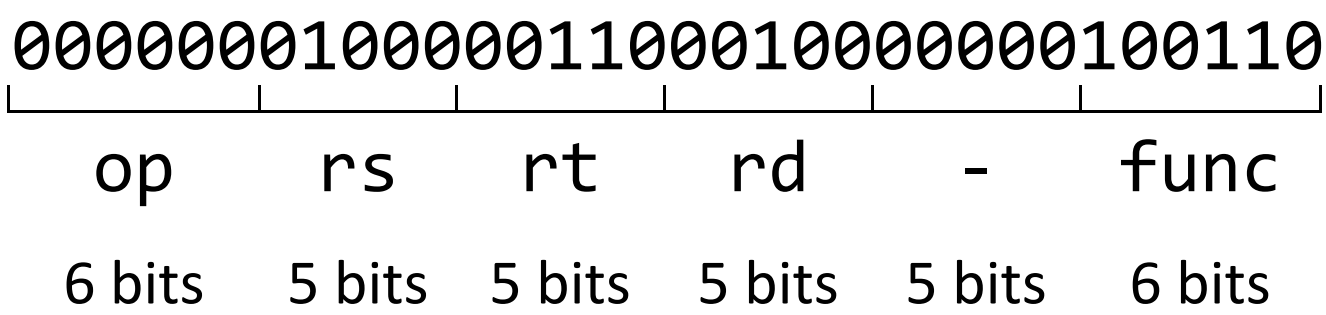
  - Stage 1: Fetch instruction, increment PC.
  - Stage 2: decode and determine it is a jal, update PC to 0xdeadbeef.
  - Stage 3: skip
  - Stage 4: skip
  - Stage 5: skip

MIPS instruction formats

All MIPS instructions are 32 bits long, has 3 formats



Arithmetic Instructions

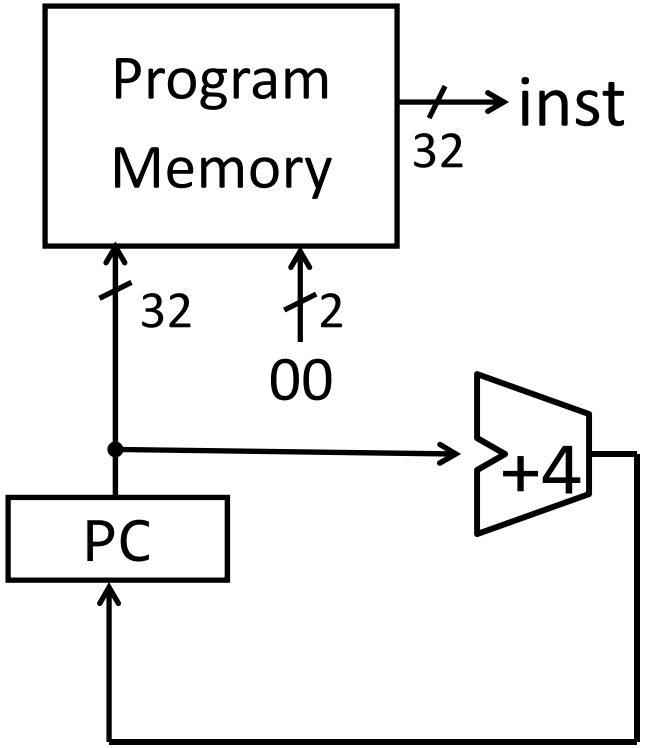


op	func	mnemonic	description
0x0	0x21	ADDU rd, rs, rt	R[rd] = R[rs] + R[rt]
0x0	0x23	SUBU rd, rs, rt	R[rd] = R[rs] – R[rt]
0x0	0x25	OR rd, rs, rt	R[rd] = R[rs] | R[rt]
0x0	0x26	XOR rd, rs, rt	R[rd] = R[rs] ⊕ R[rt]
0x0	0x27	NOR rd, rs rt	R[rd] = ~ (R[rs] | R[rt])

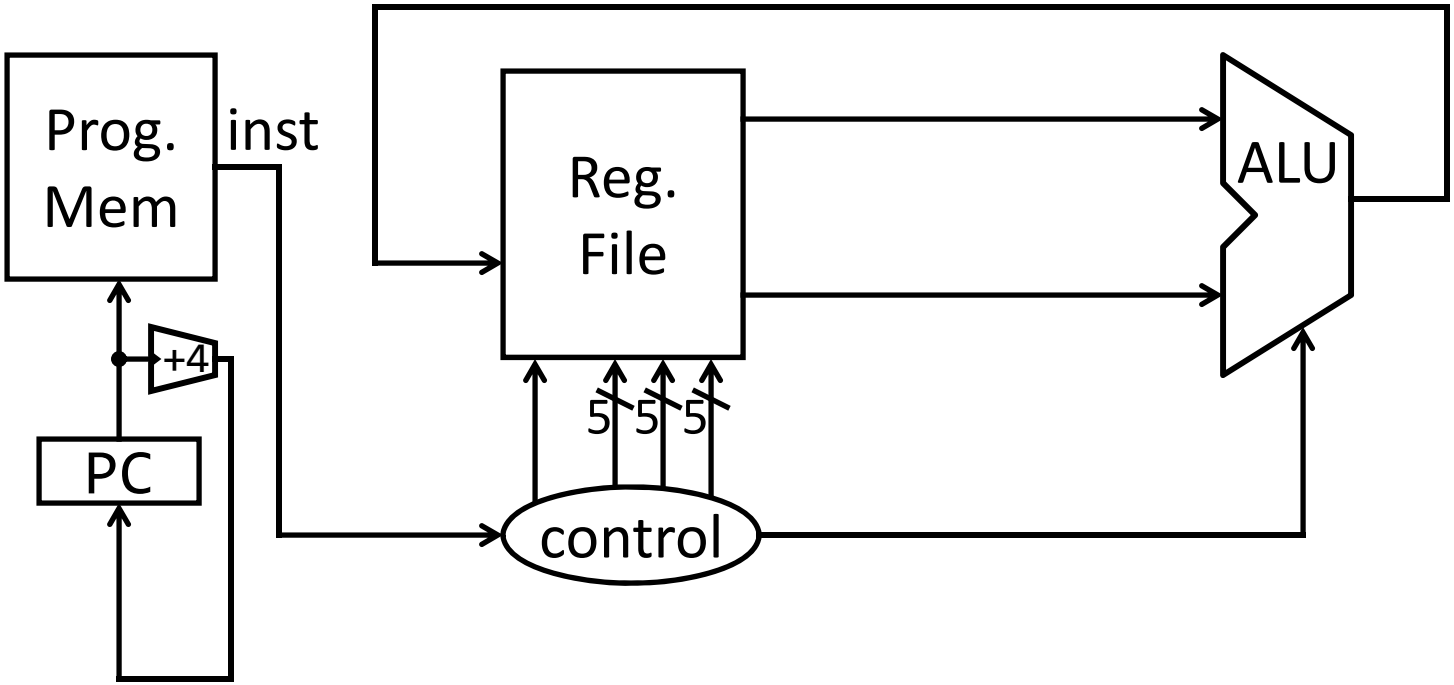
Instruction Fetch

Instruction Fetch Circuit

- Fetch instruction from memory
- Calculate address of next instruction
- Repeat



Arithmetic and Logic



Example Programs

r4 = (r1 + r2) | r3

r8 = 4*r3 + r4 – 1

r9 = 9

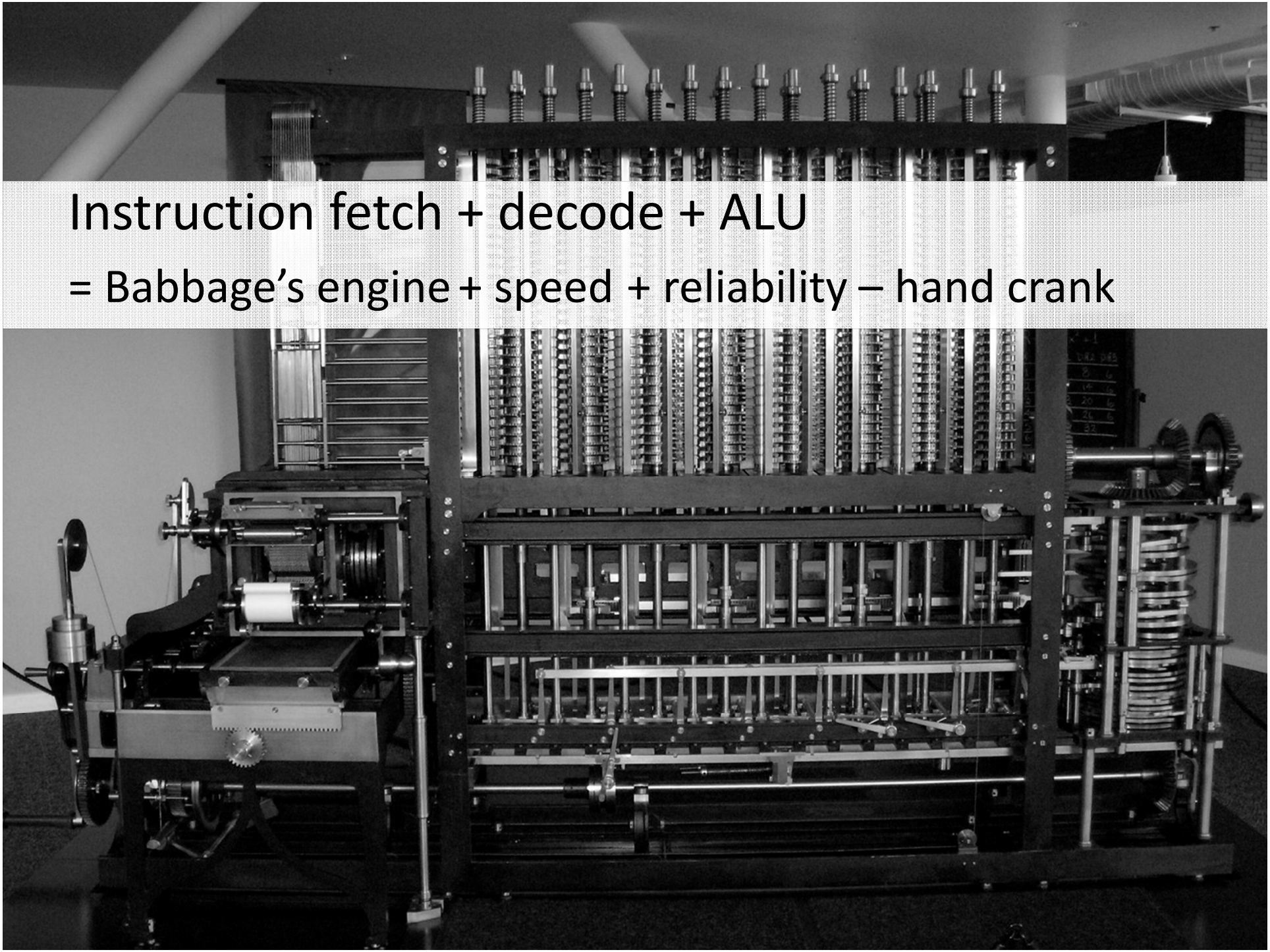
ADDU rd, rs, rt

SUBU rd, rs, rt

OR rd, rs, rt

XOR rd, rs, rt

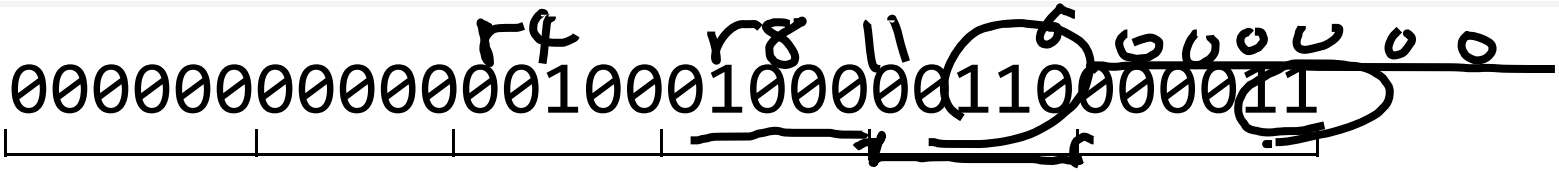
NOR rd, rs rt



Instruction fetch + decode + ALU

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

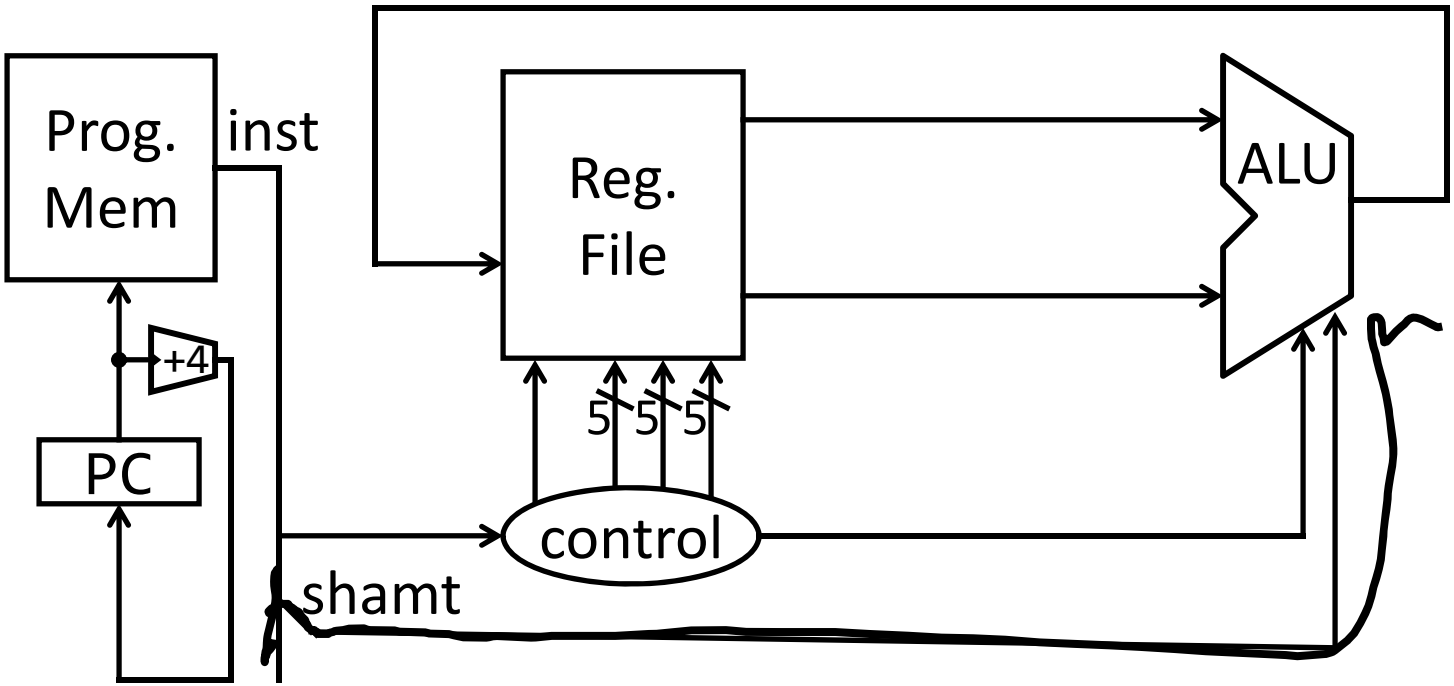
Arithmetic Instructions: Shift



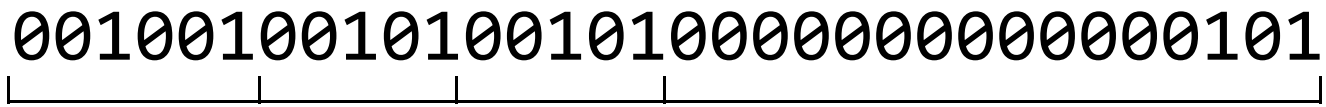
op	func	mnemonic	description
0x0	0x0	SLL rd, rs, shamt	R[rd] = R[rt] << shamt
0x0	0x2	SRL rd, rs, shamt	R[rd] = R[rt] >>> shamt (zero ext.)
0x0	0x3	SRA rd, rs, shamt	R[rd] = R[rs] >> shamt (sign ext.)

ex: r5 = r3 * 8

Shift



Arithmetic Instructions: Immediates

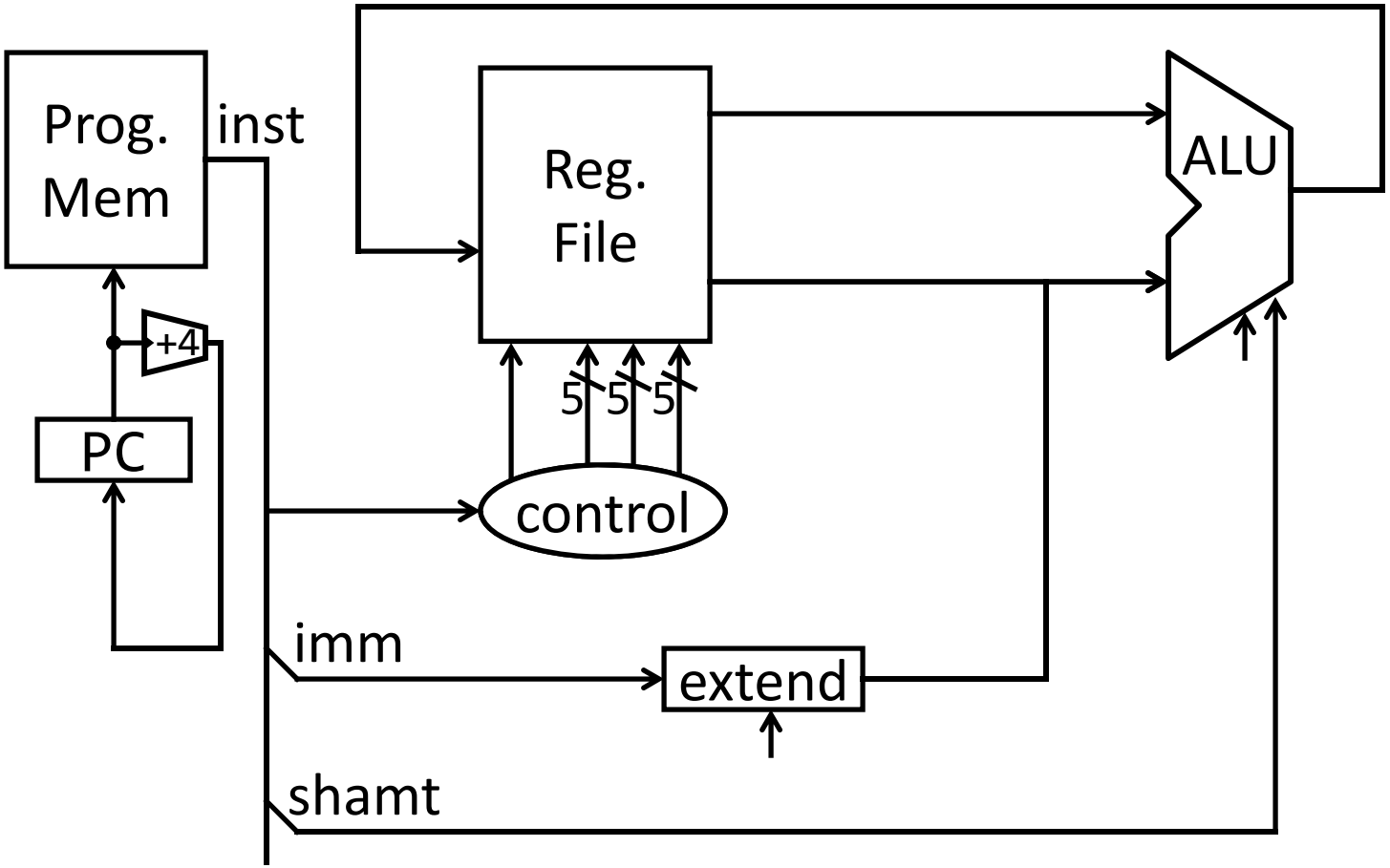


I-Type

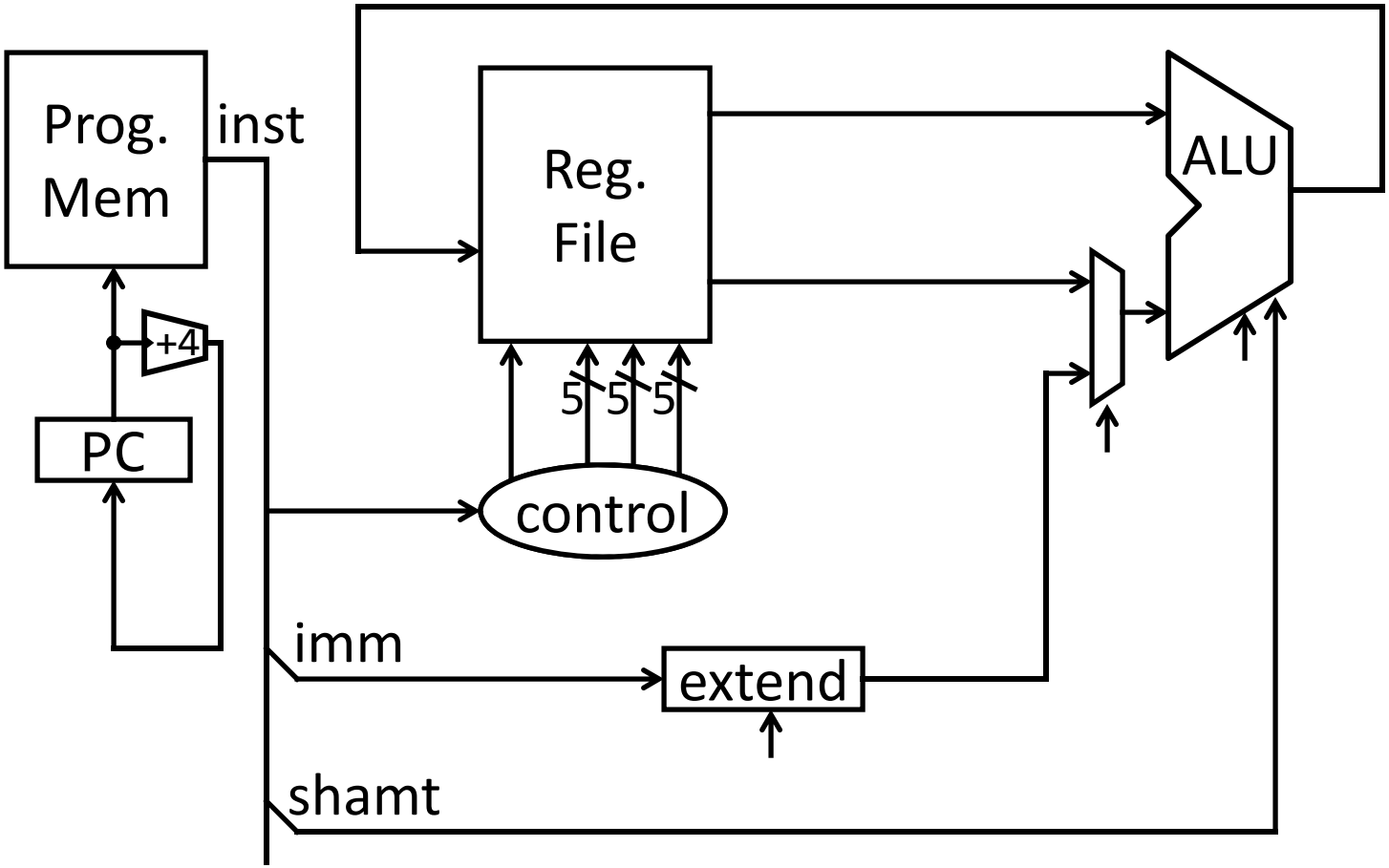
op	mnemonic	description
0x9	ADDIU rd, rs, imm	R[rd] = R[rs] + sign_extend(imm)
0xc	ANDI rd, rs, imm	R[rd] = R[rs] & zero_extend(imm)
0xd	ORI rd, rs, imm	R[rd] = R[rs] | zero_extend(imm)

ex: r5 += 5 ex: r9 = -1 ex: r9 = 65535

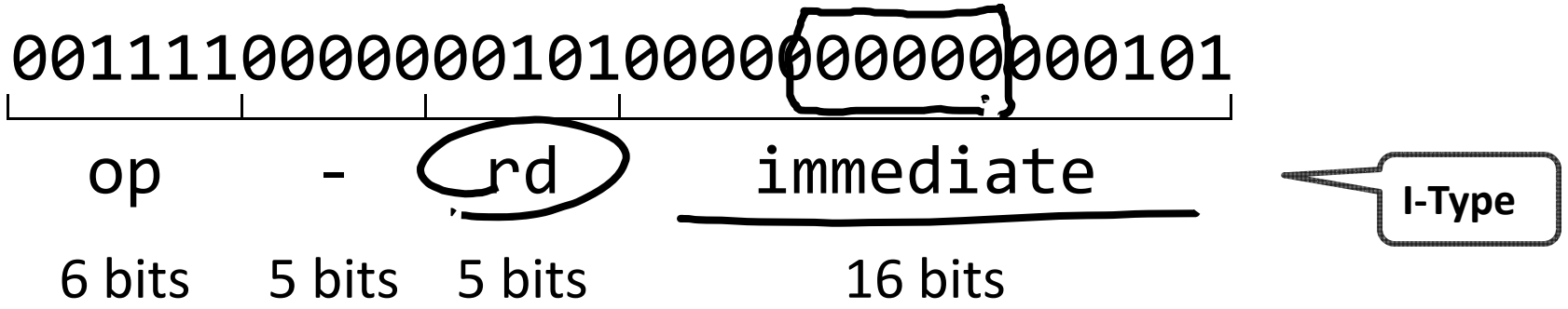
Immediates



Immediates



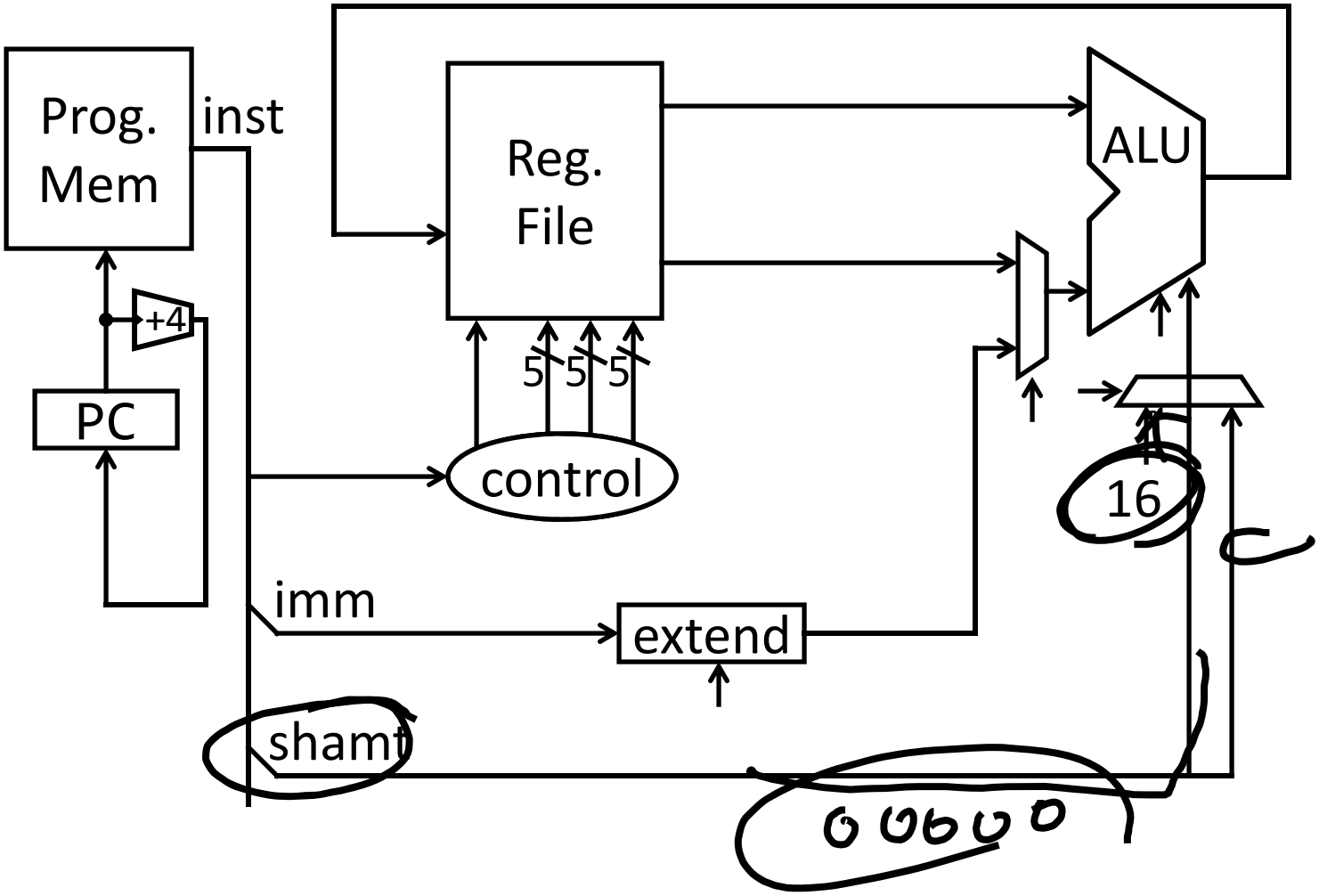
Arithmetic Instructions: Immediates



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

ex: r5 = 0xdeadbeef

Immediates


Prog.
Mem
inst
Reg.
File
ALU
+4
PC
5
5
5
control
16
imm
extend
shamt

MIPS Instruction Types

Arithmetic/Logical

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

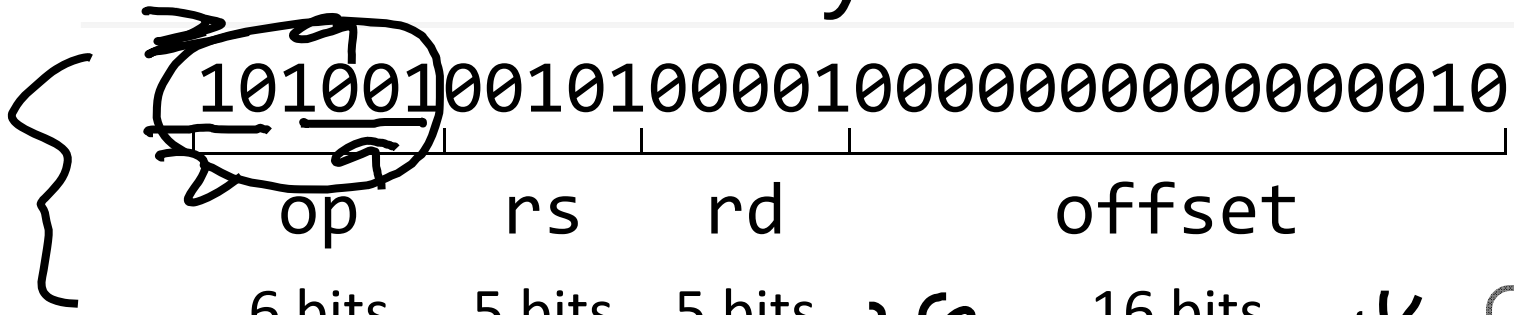
Memory Access

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

Control flow

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

Memory Instructions



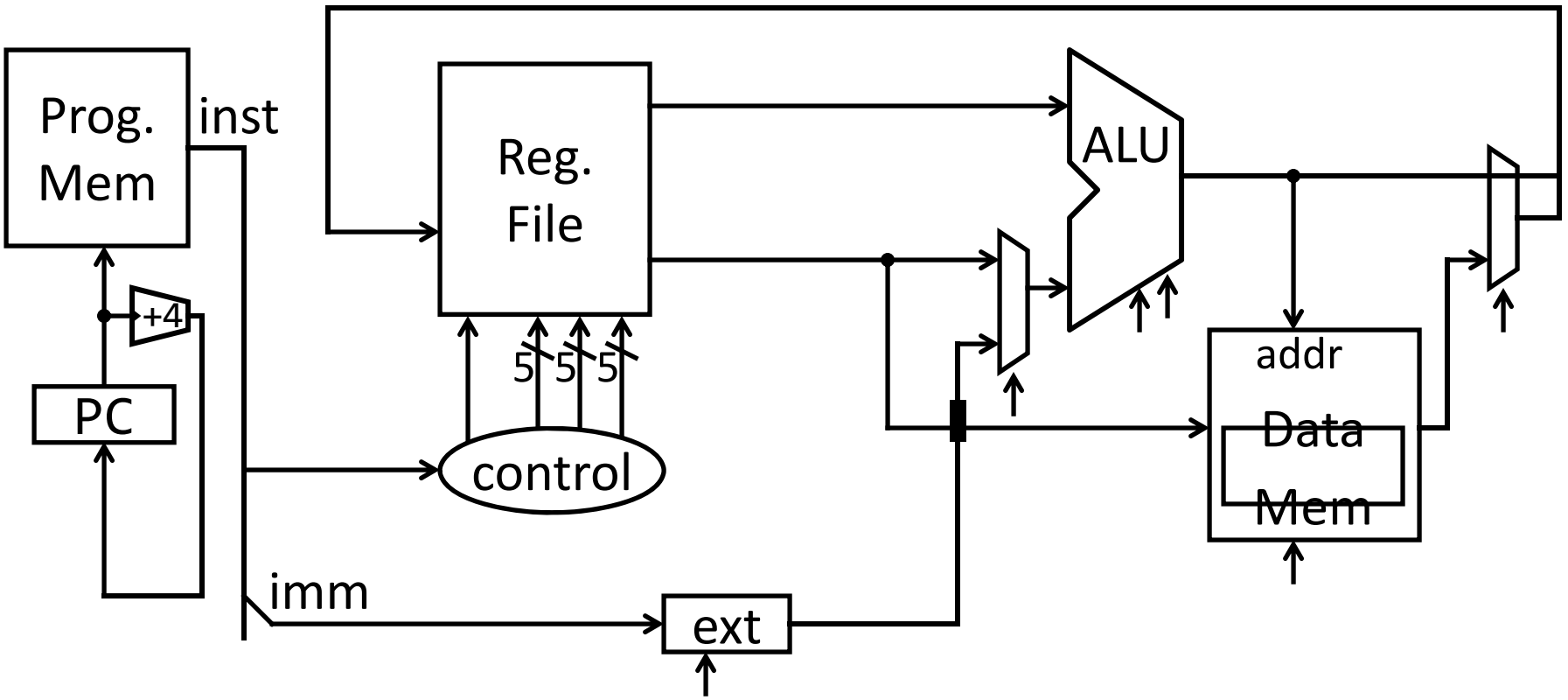
I-Type

base + offset addressing

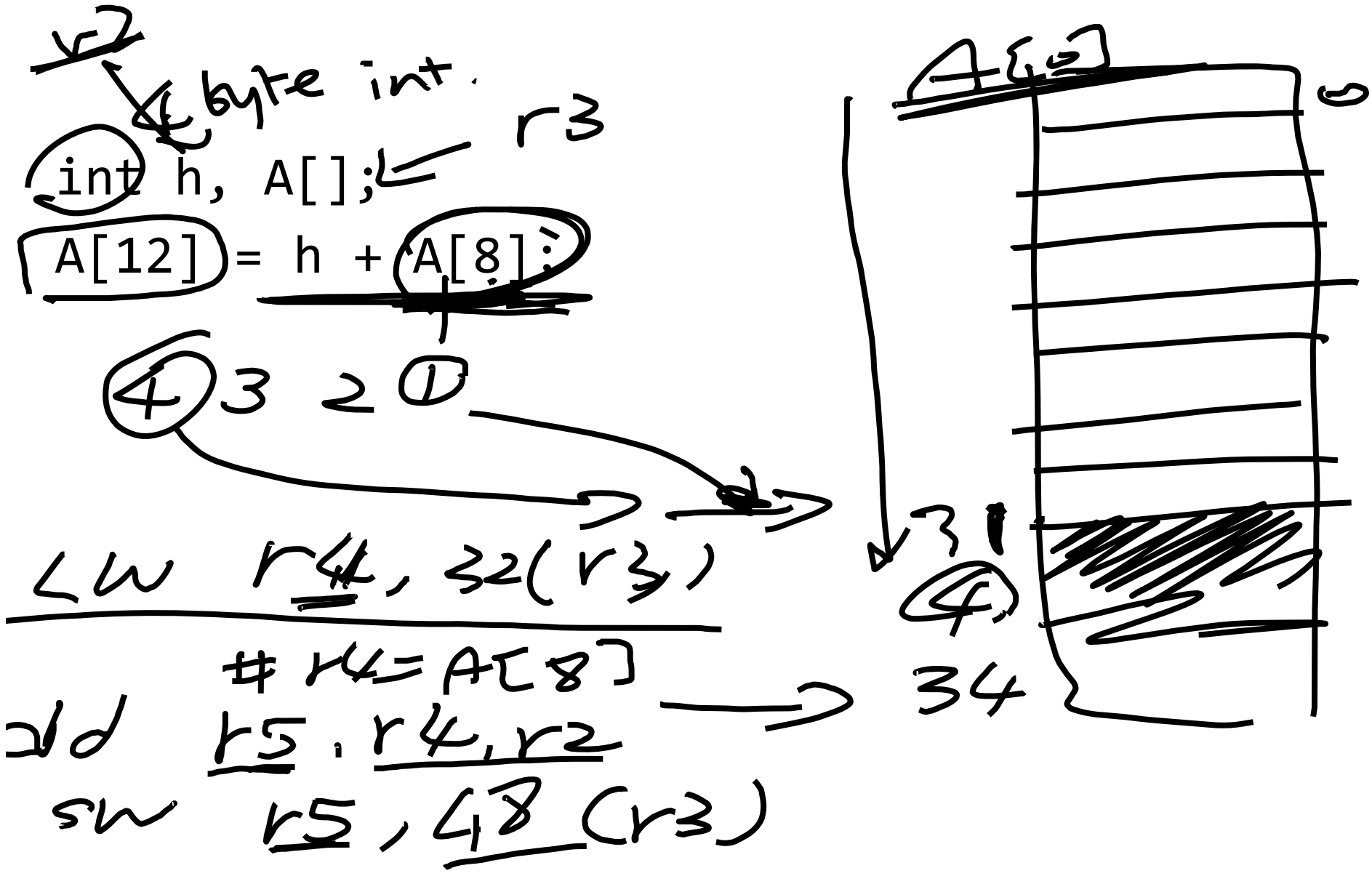
op	mnemonic	description
0x20	LB rd, offset(rs)	R[rd] = sign_ext(Mem[offset+R[rs]])
0x24	LBU rd, offset(rs)	R[rd] = zero_ext(Mem[offset+R[rs]])
0x21	LH rd, offset(rs)	R[rd] = sign_ext(Mem[offset+R[rs]])
0x25	LHU rd, offset(rs)	R[rd] = zero_ext(Mem[offset+R[rs]])
0x23	LW rd, offset(rs)	R[rd] = Mem[offset+R[rs]]
0x28	SB rd, offset(rs)	Mem[offset+R[rs]] = R[rd]
0x29	SH rd, offset(rs)	Mem[offset+R[rs]] = R[rd]
0x2b	SW rd, offset(rs)	Mem[offset+R[rs]] = R[rd]

signed offsets

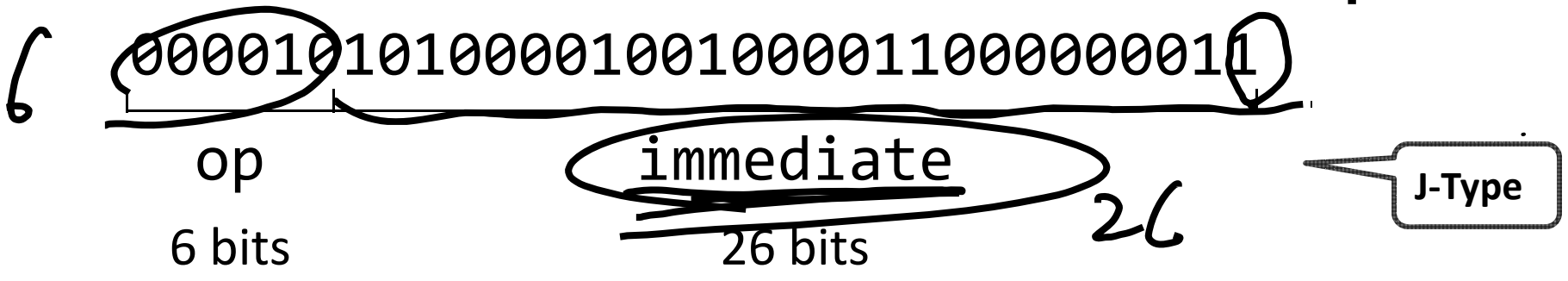
Memory Operations



Example


4 byte int.
r3
int h, A[];
A[12] = h + A[8];
4 3 2 1
lw r4, 32(r3)
# r4 = A[8]
add r5, r4, r2
sw r5, 48(r3)
34

Control Flow: Absolute Jump



op	mnemonic	description
0x2	J target	PC = $(PC+4)_{31..28}$ || target || 00

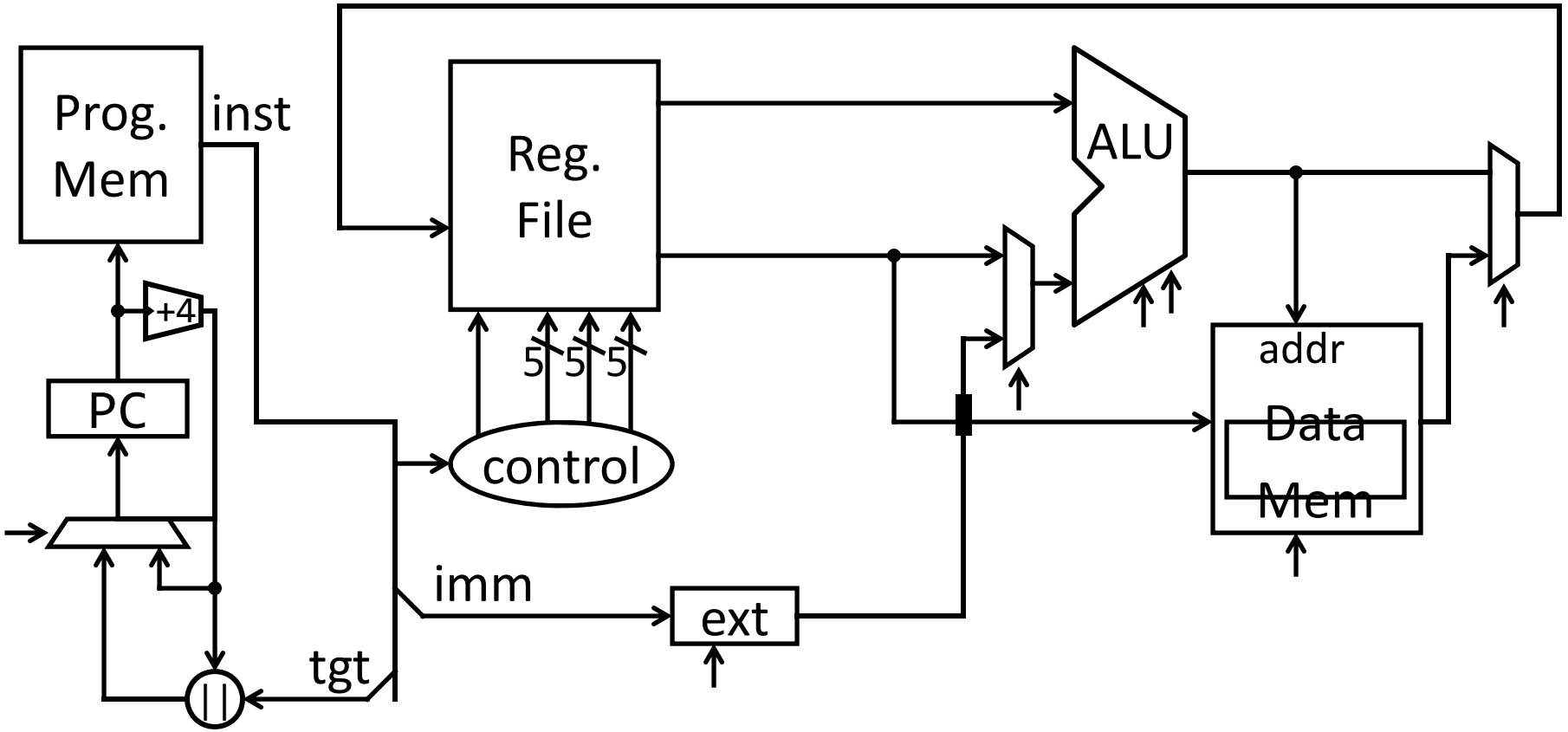
Absolute addressing for jumps

- Jump from 0x30000000 to 0x20000000?
  - But: Jumps from 0x2FFFFFFF to 0x3xxxxxxx are possible, but not reverse
- Trade-off: out-of-region jumps vs. 32-bit instruction encoding

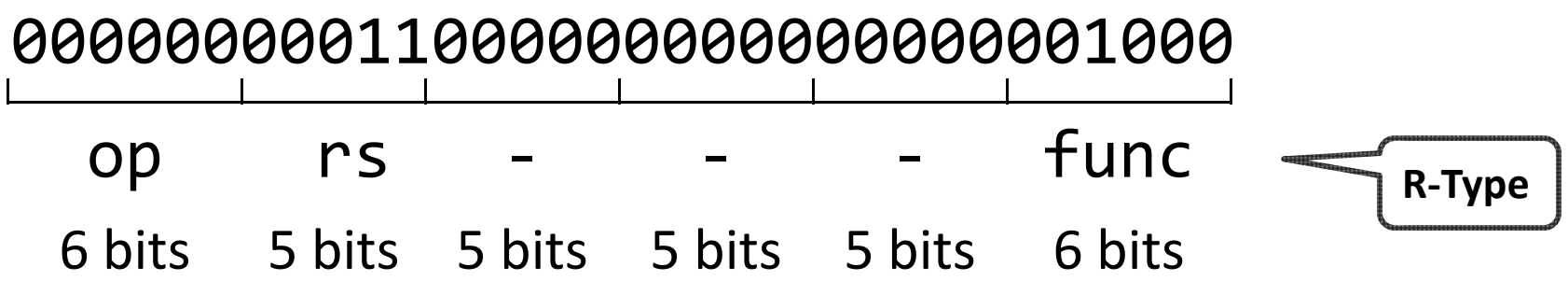
MIPS Quirk:

- jump targets computed using *already incremented* PC

Absolute Jump

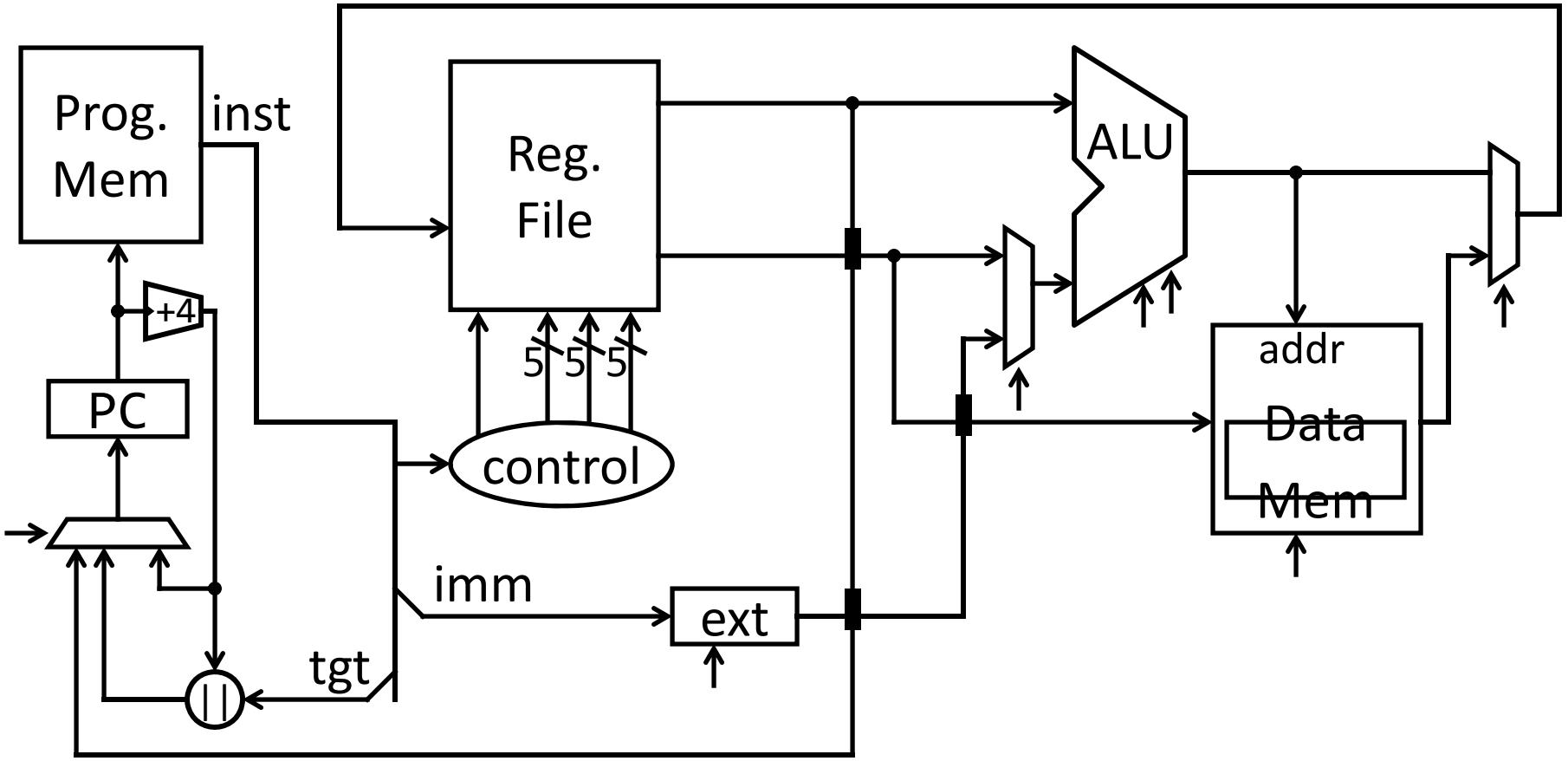


Control Flow: Jump Register



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

Jump Register



Examples (2)

jump to 0xabcd1234

assume 0 <= r3 <= 1

if (r3 == 0) jump to 0xdecafe00

else jump to 0xabcd1234

Examples (2)

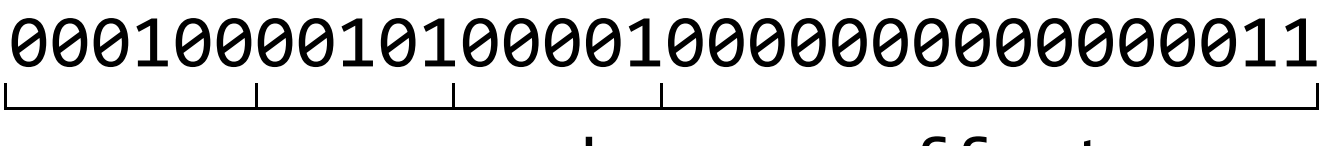
jump to 0xabcd1234

assume 0 <= r3 <= 1

if (r3 == 0) jump to 0xdecafe0

else jump to 0xabcd1234

Control Flow: Branches



I-Type

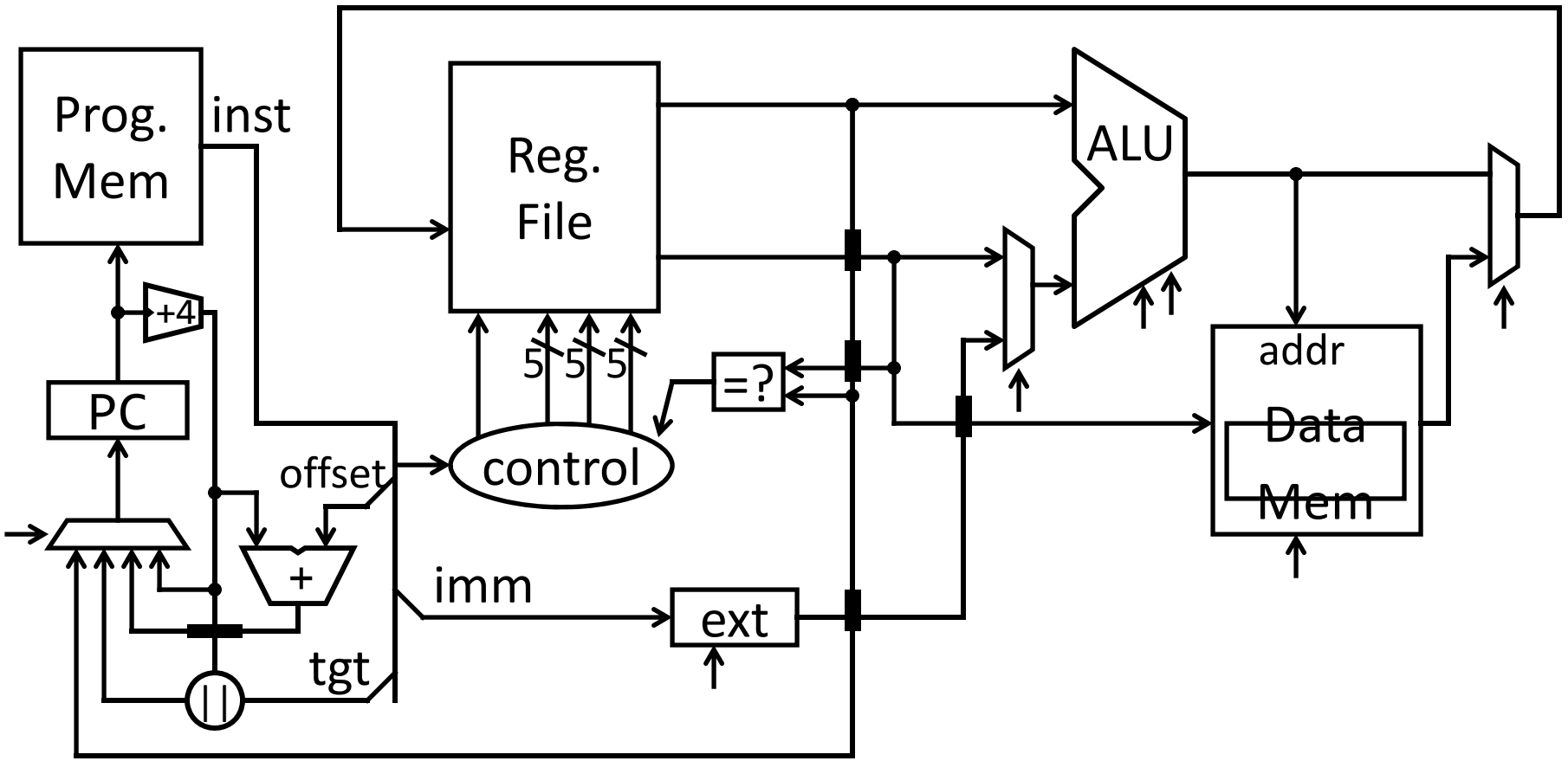
signed offsets

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)

Examples (3)

```
if (i == j) { i = i * 4; }
else { j = i - j; }
```

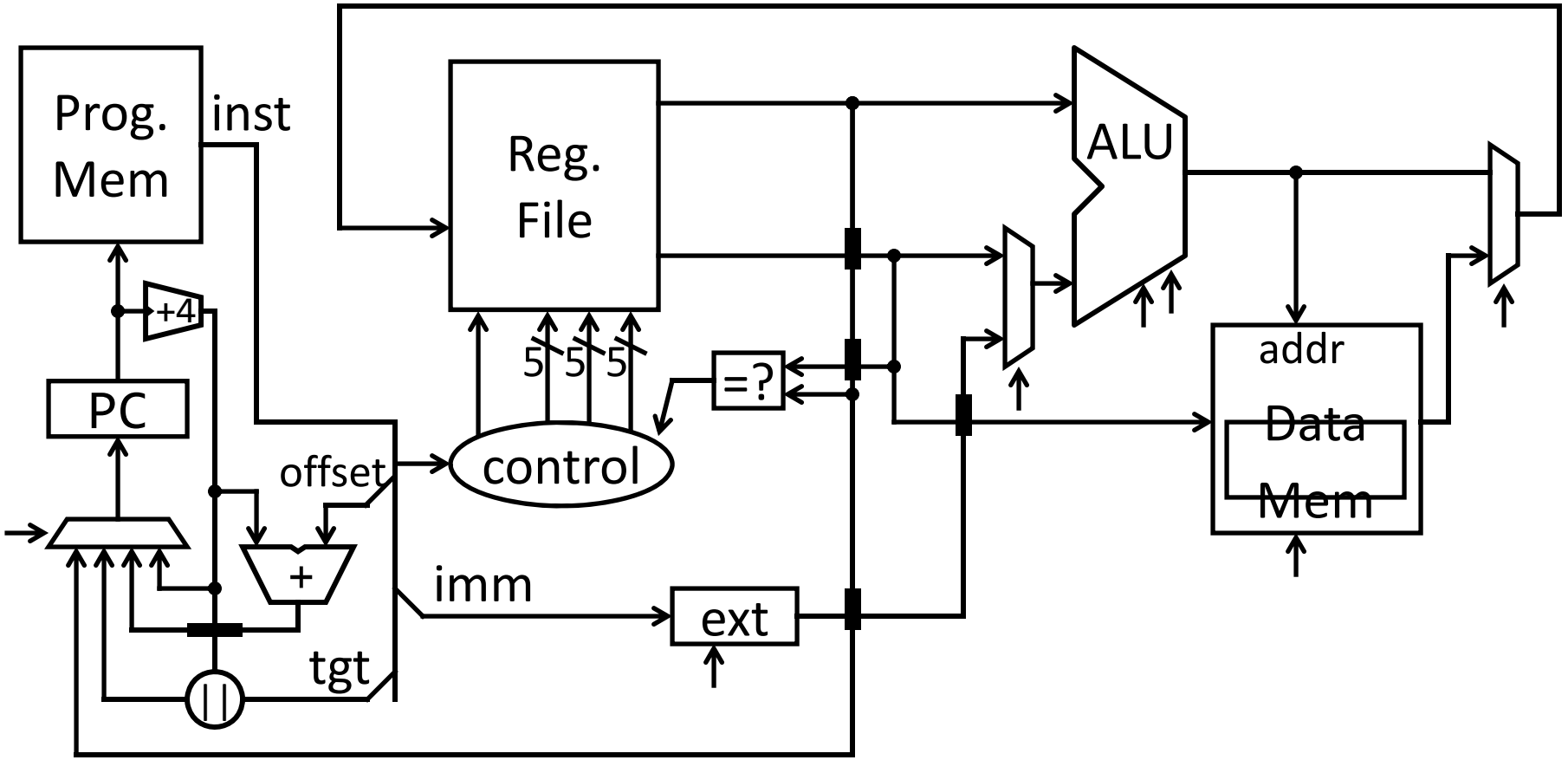
Absolute Jump



Could have used ALU for branch add

Could have used ALU for branch cmp

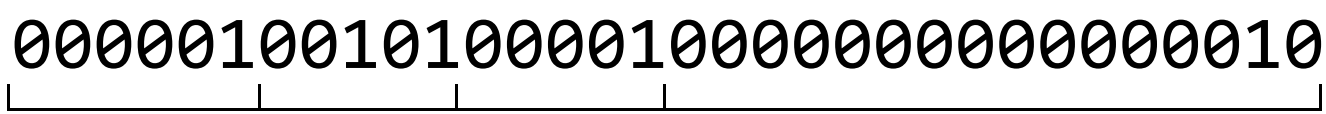
Absolute Jump



Could have used ALU for branch add

Could have used ALU for branch cmp

Control Flow: More Branches

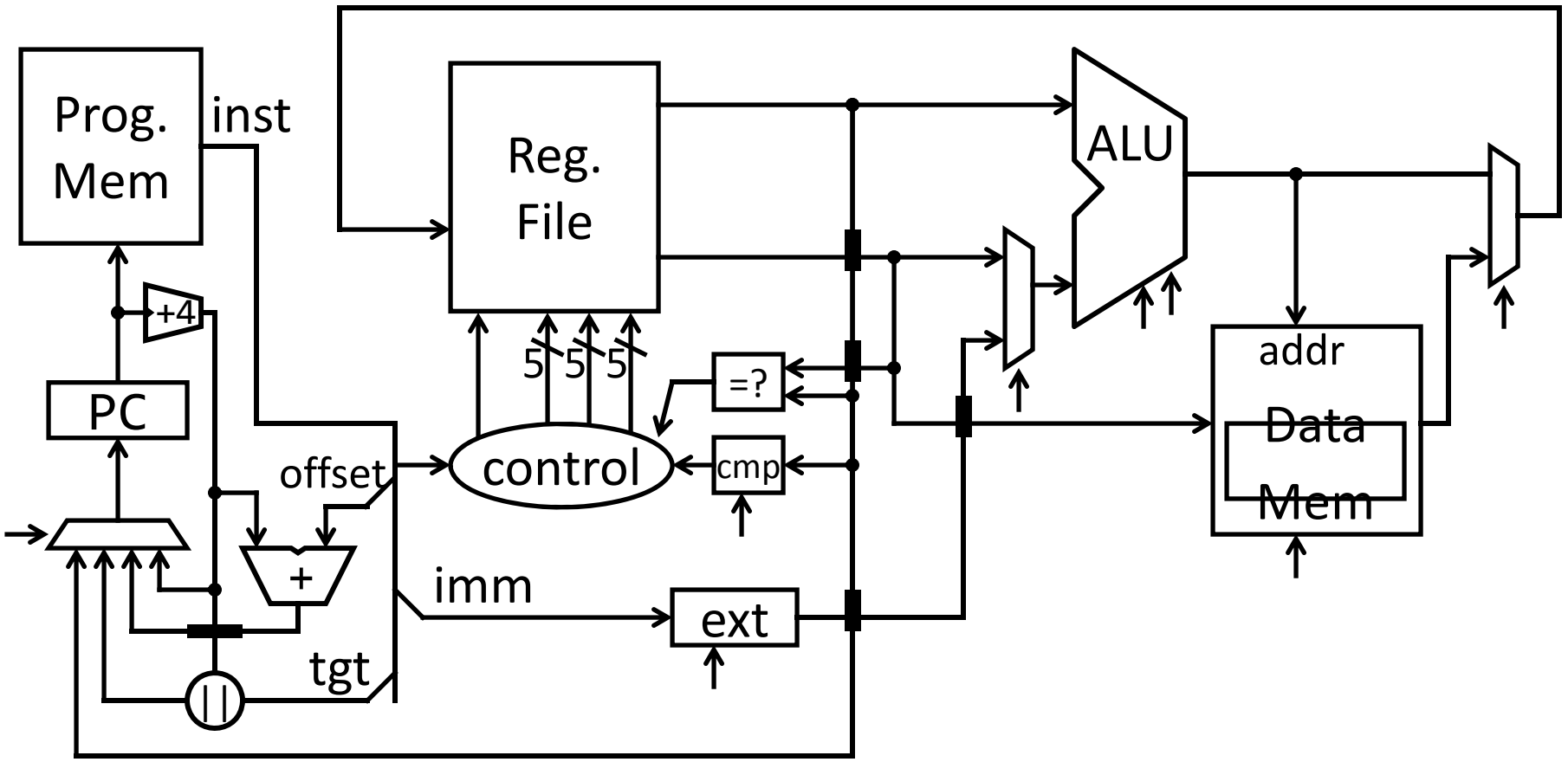


almost I-Type

signed offsets

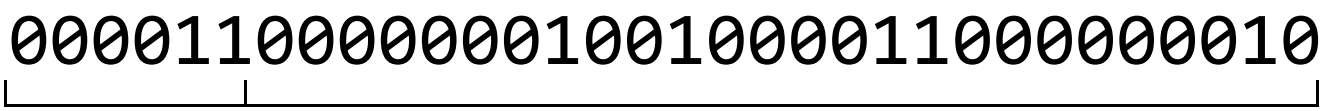
op	subop	mnemonic	description
0x1	0x0	BLTZ rs, offset	if R[rs] < 0 then PC = PC+4+ (offset<<2)
0x1	0x1	BGEZ rs, offset	if R[rs] ≥ 0 then PC = PC+4+ (offset<<2)
0x6	0x0	BLEZ rs, offset	if R[rs] ≤ 0 then PC = PC+4+ (offset<<2)
0x7	0x0	BGTZ rs, offset	if R[rs] > 0 then PC = PC+4+ (offset<<2)

Absolute Jump



Could have used ALU for branch cmp

Control Flow: Jump and Link



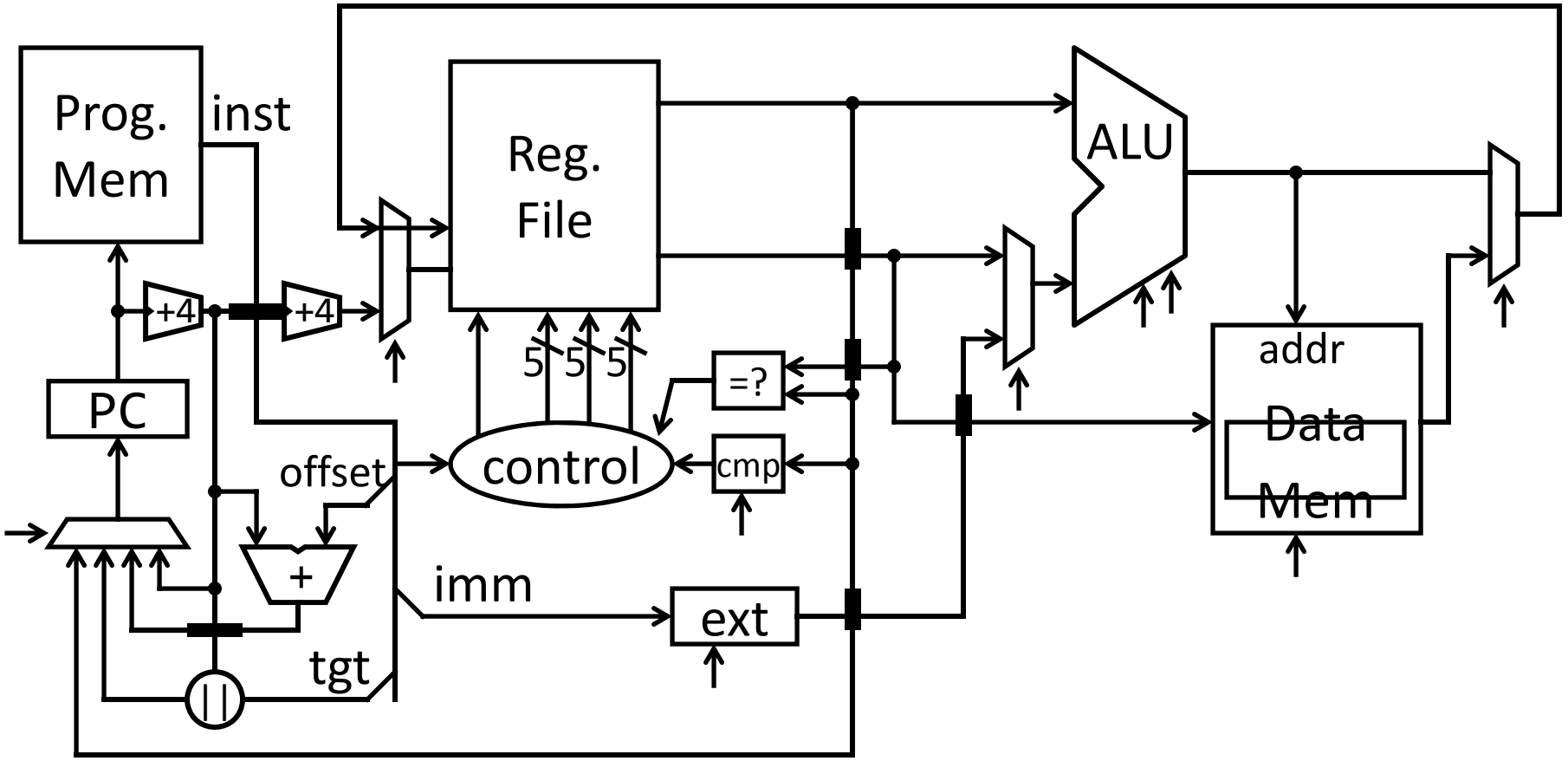
op immediate

6 bits 26 bits

J-Type

op	mnemonic	description
0x3	JAL target	r31 = PC+8 PC = $(PC+4)_{32..29}$ || target || 00

Absolute Jump



Could have used ALU for link add

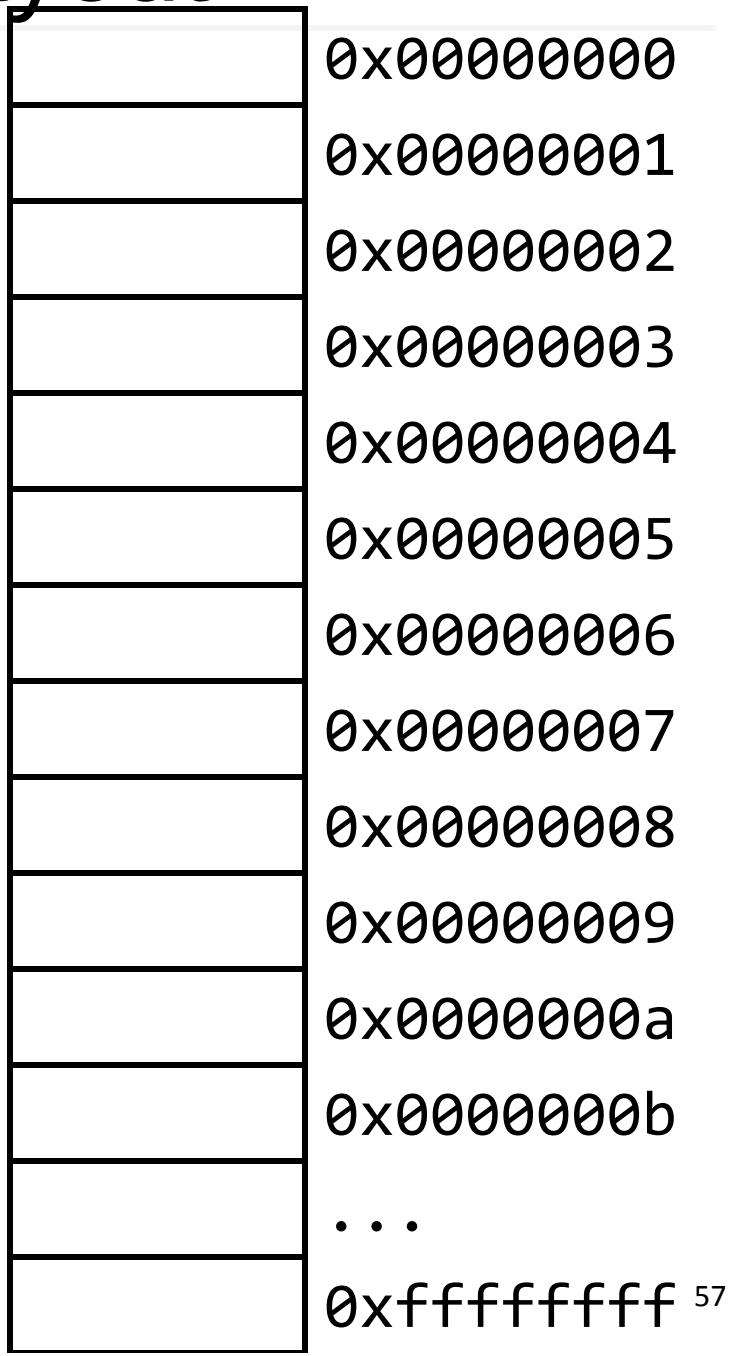
Memory Layout

Examples:

```
# r5 contains 0x5
sb r5, 2(r0)
```

M[R[rs] + SigExtImm](7:0) = R[rd](7:0)

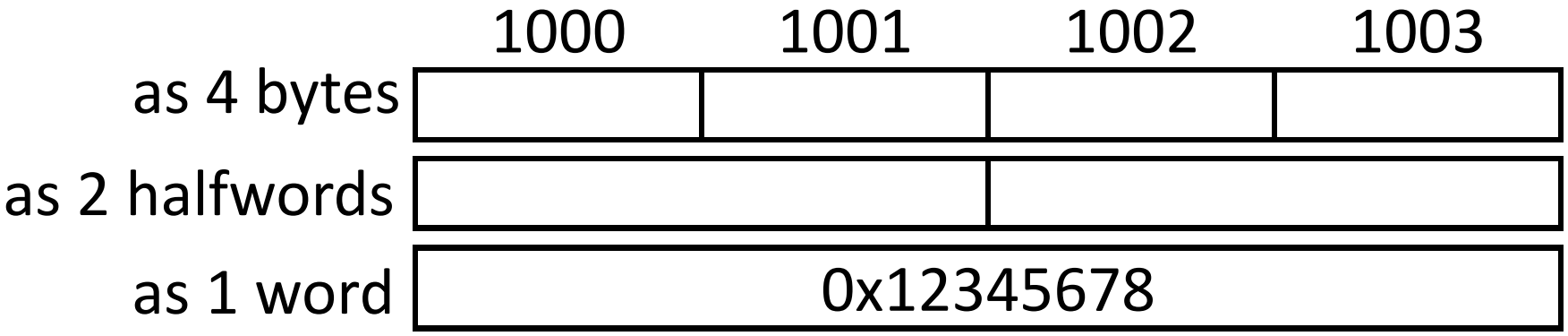
```
lb r6, 2(r0)

sw r5, 8(r0)
lb r7, 8(r0)
lb r8, 11(r0)
```



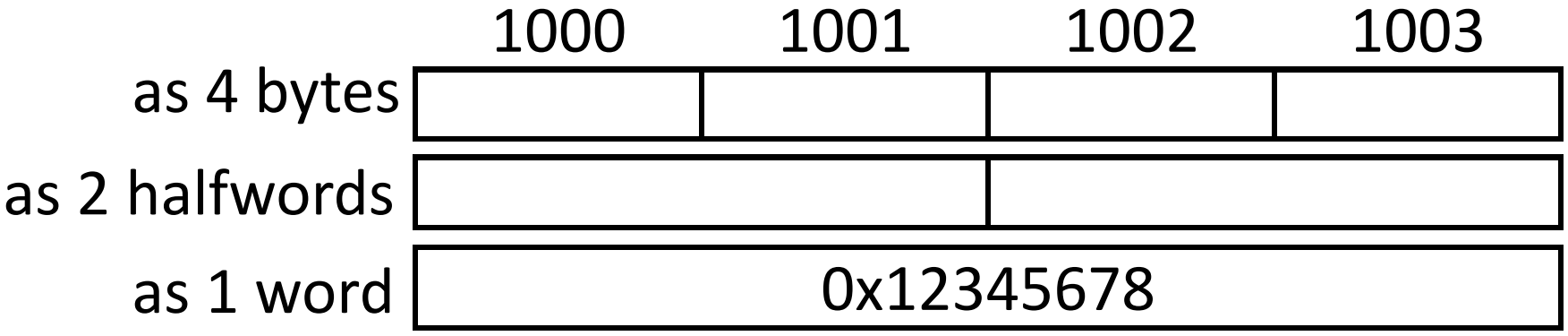
Endianness

Endianness: Ordering of bytes within a memory word

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



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



Next Time

CPU Performance

Pipelined CPU