CPU Performance
Pipelined CPU

Hakim Weatherspoon

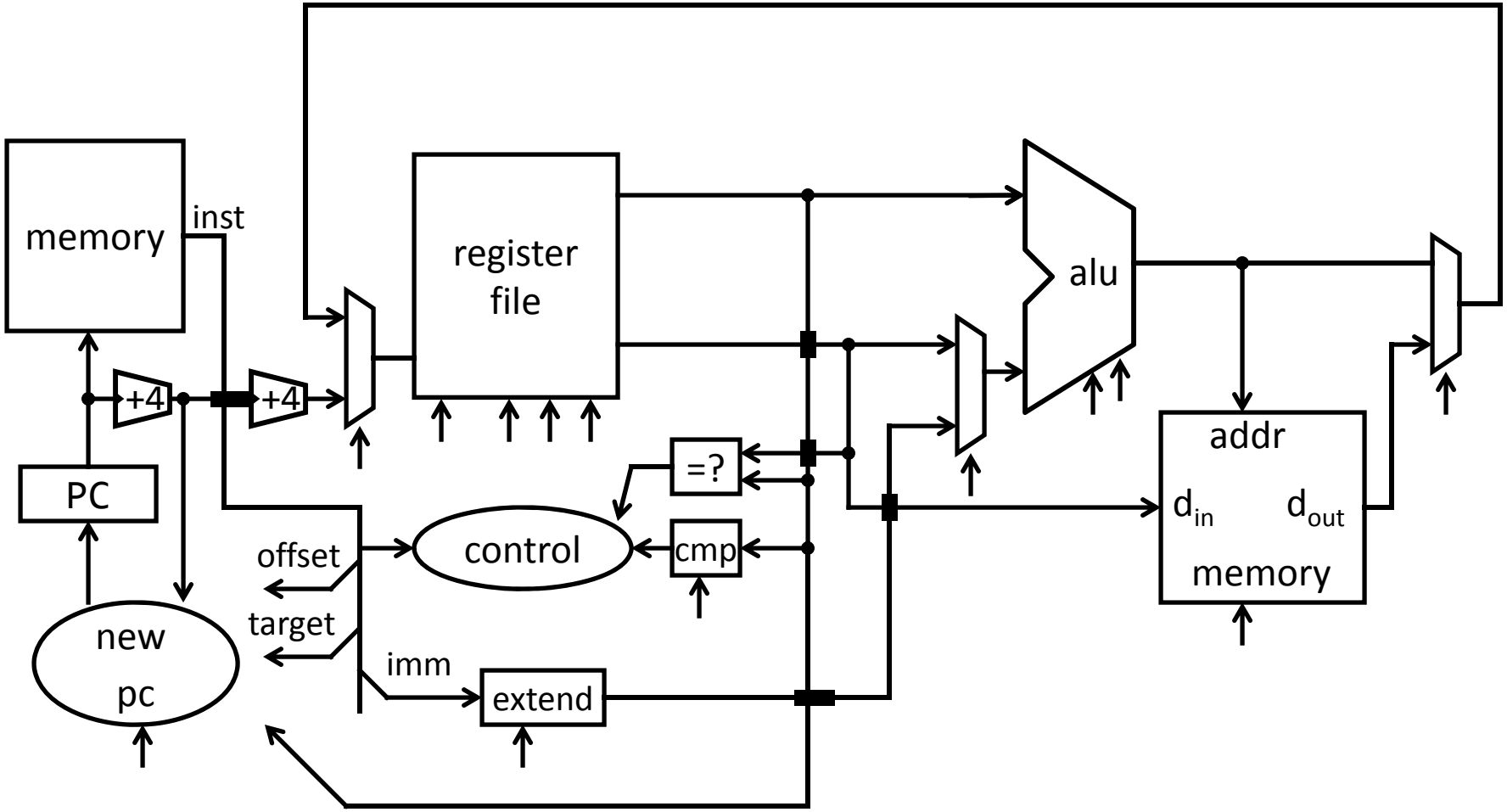
CS 3410, Spring 2013

Computer Science

Cornell University

See P&H Chapters 1.4 and 4.5

Big Picture: Building a Processor



A Single cycle processor

Goals for today

MIPS Datapath

- Memory layout
- Control Instructions

Performance

- CPI (Cycles Per Instruction)
- MIPS (Instructions Per Cycle)
- Clock Frequency

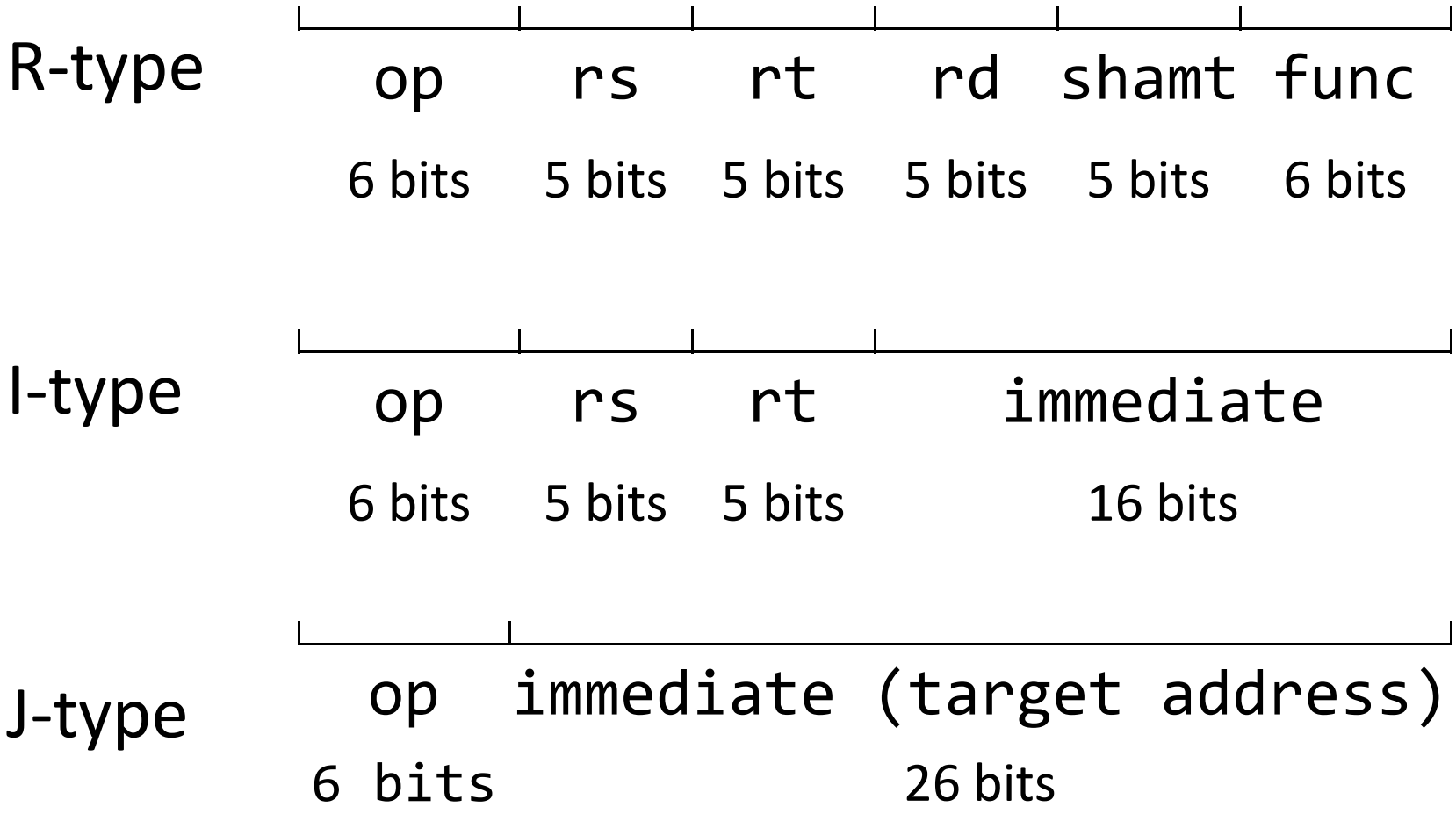
Pipelining

- Intuition on Latency vs throuput

Memory Layout and Control instructions

MIPS instruction formats

All MIPS instructions are 32 bits long, has 3 formats



MIPS Instruction Types

Arithmetic/Logical

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

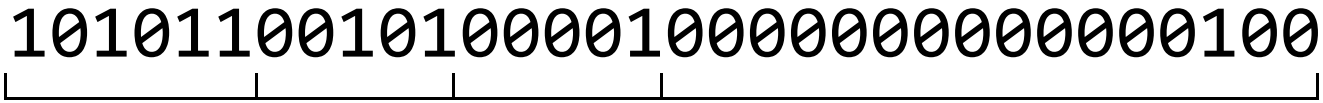
Memory Access

- 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



op rs rd offset

6 bits 5 bits 5 bits 16 bits

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]

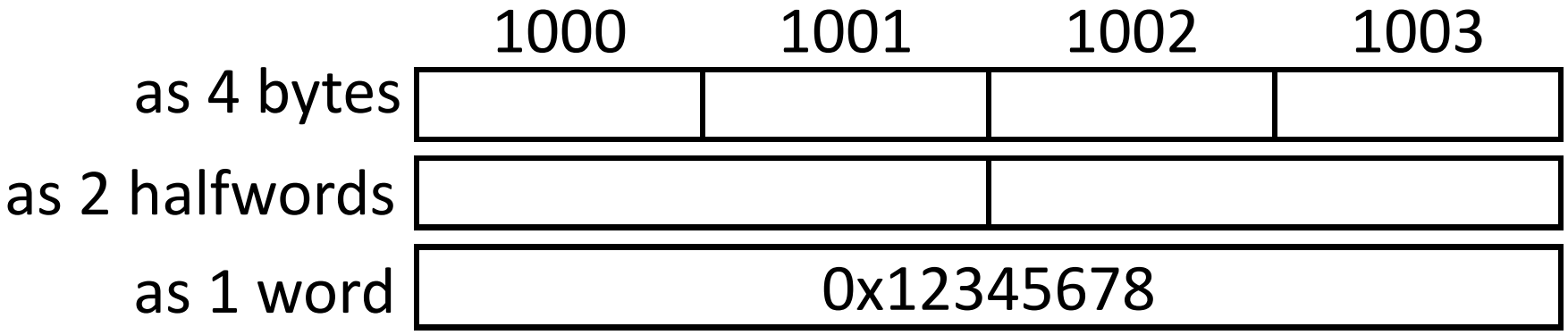
signed offsets

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

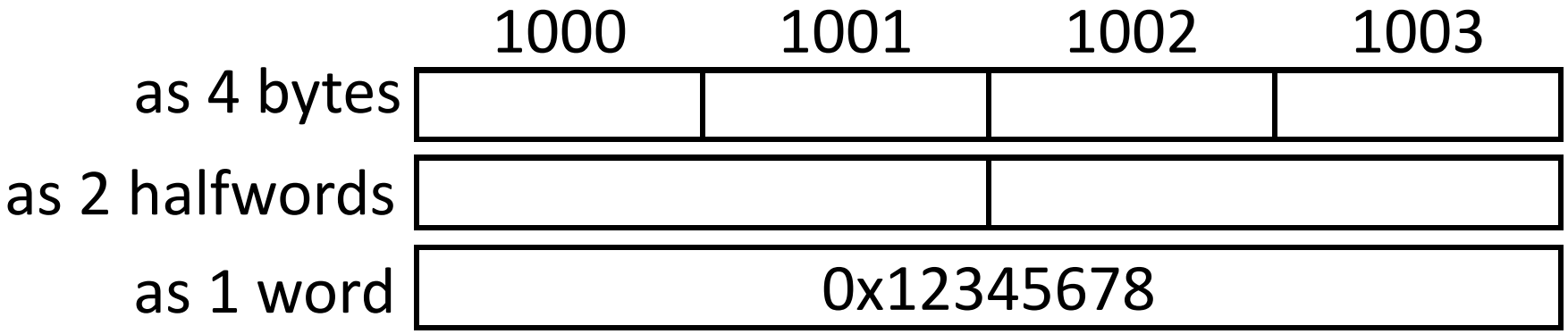
Endianness

Endianness: Ordering of bytes within a memory word

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



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



Memory Layout

Examples (big/little endian):

r5 contains 5 (0x00000005)

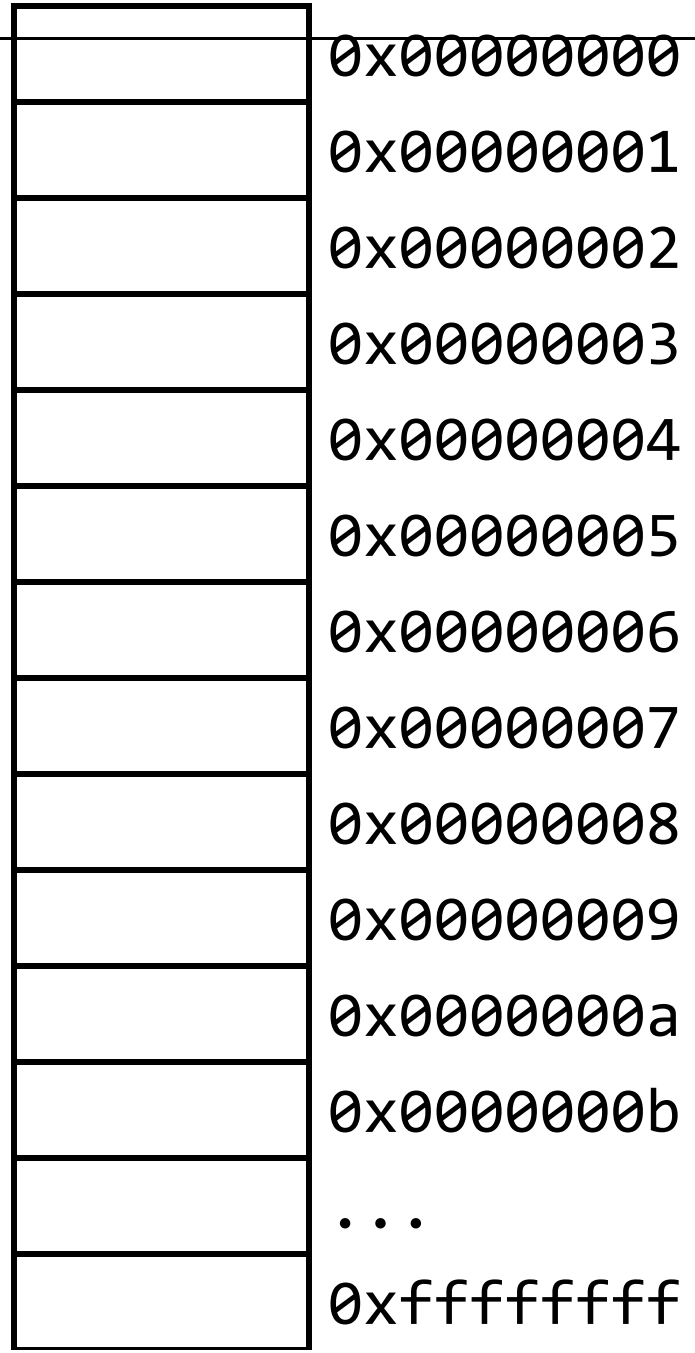
SB r5, 2(r0)

LB r6, 2(r0)

SW r5, 8(r0)

LB r7, 8(r0)

LB r8, 11(r0)



MIPS Instruction Types

Arithmetic/Logical

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

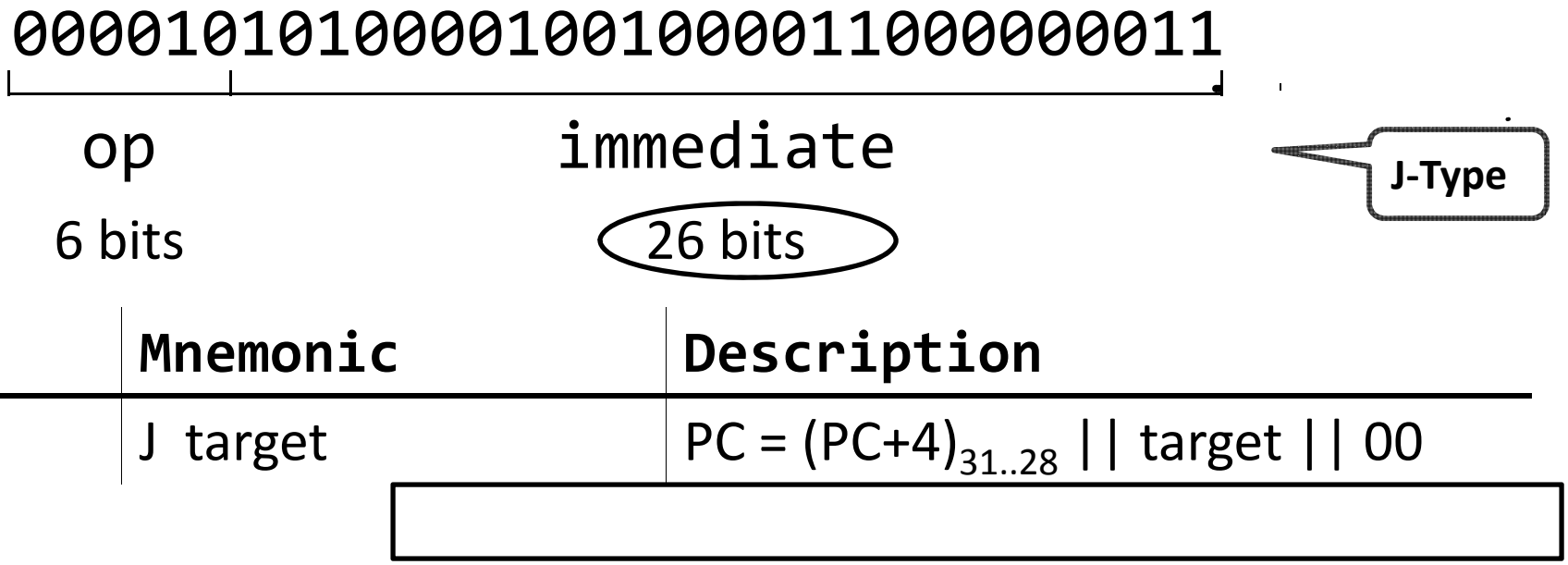
Memory Access

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

Control flow

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

Control Flow: Absolute Jump



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

Absolute addressing for jumps

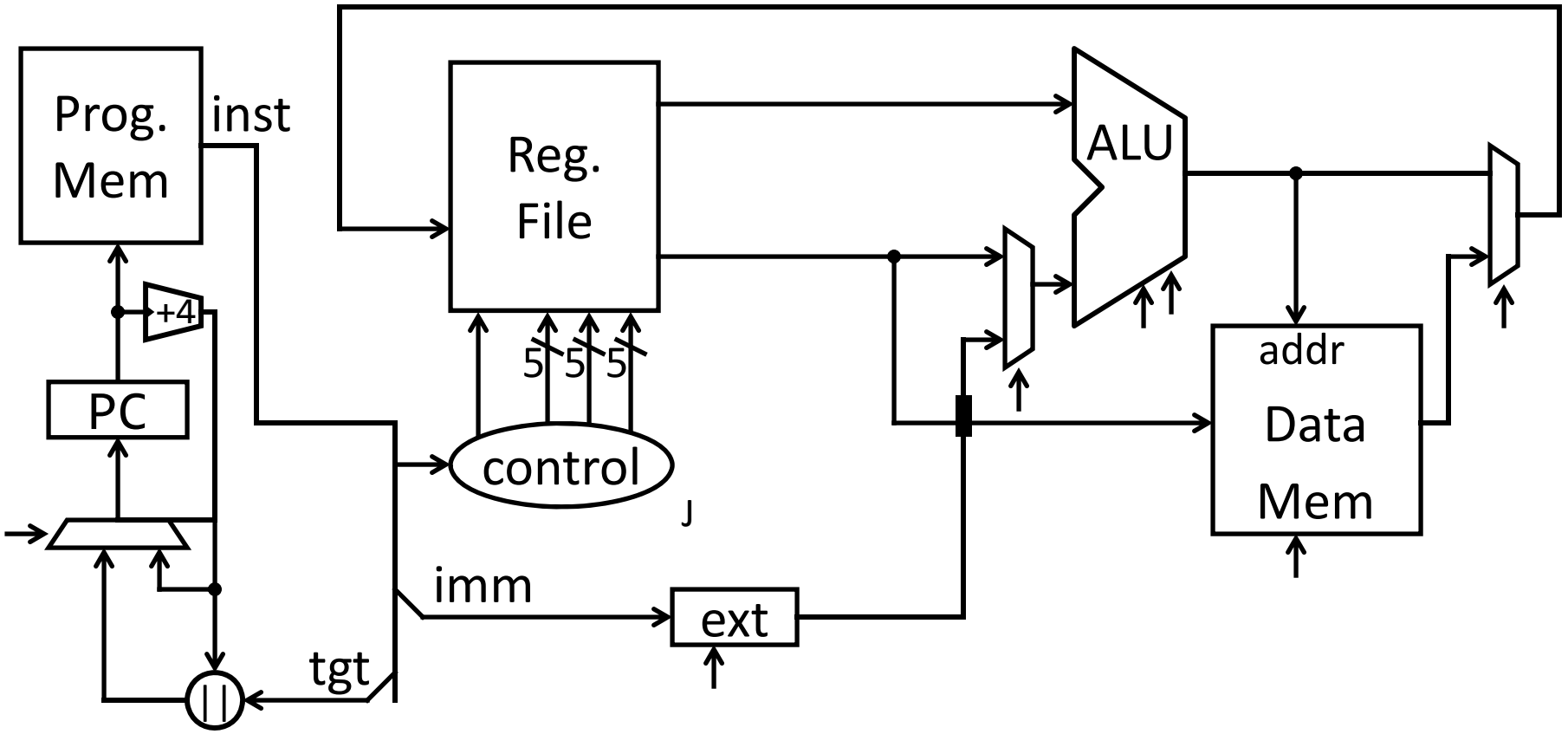
$(PC+4)_{31..28}$ will be the same

- 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:

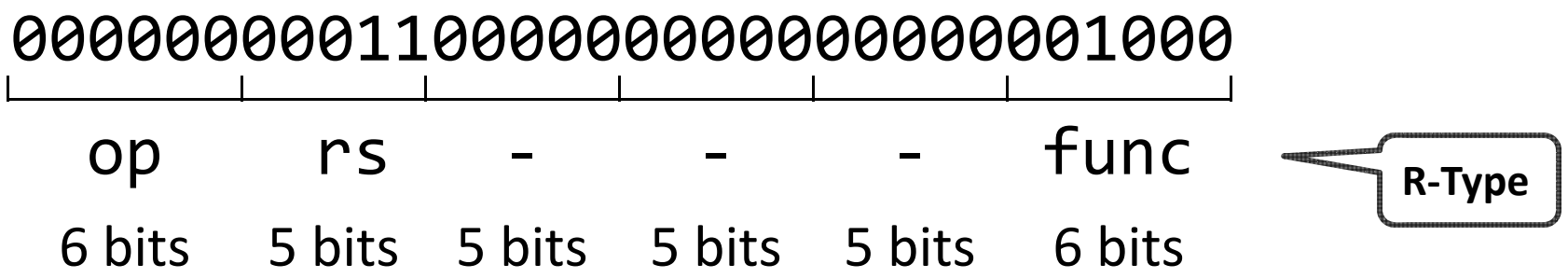
- jump targets computed using *already incremented* PC

Absolute Jump



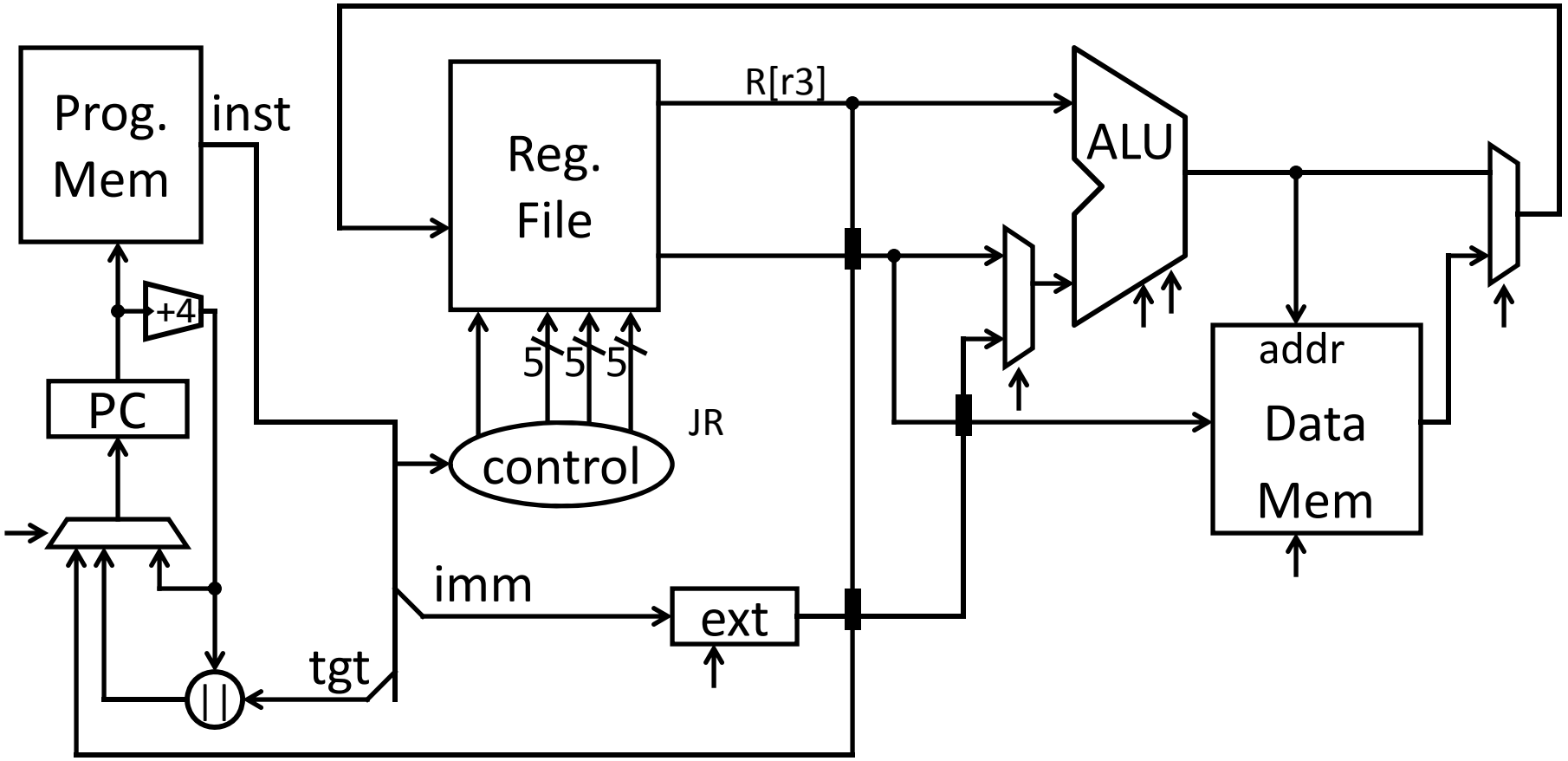
op	Mnemonic	Description
0x2	J target	PC = $(PC+4)_{31..28}$ || target || 00

Control Flow: Jump Register



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

Jump Register



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

Examples

E.g. Use Jump or Jump Register instruction to

jump to 0xabcd1234

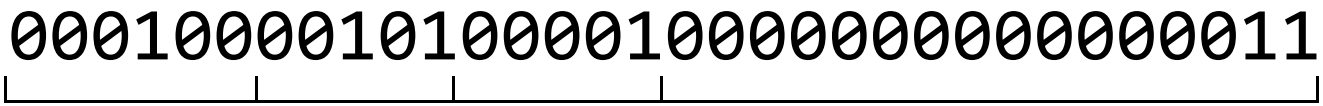
But, what about a jump based on a condition?

assume 0 <= r3 <= 1

if (r3 == 0) jump to 0xdecafe00

else jump to 0xabcd1234

Control Flow: Branches



op	rs	rd	offset
6 bits	5 bits	5 bits	16 bits

I-Type

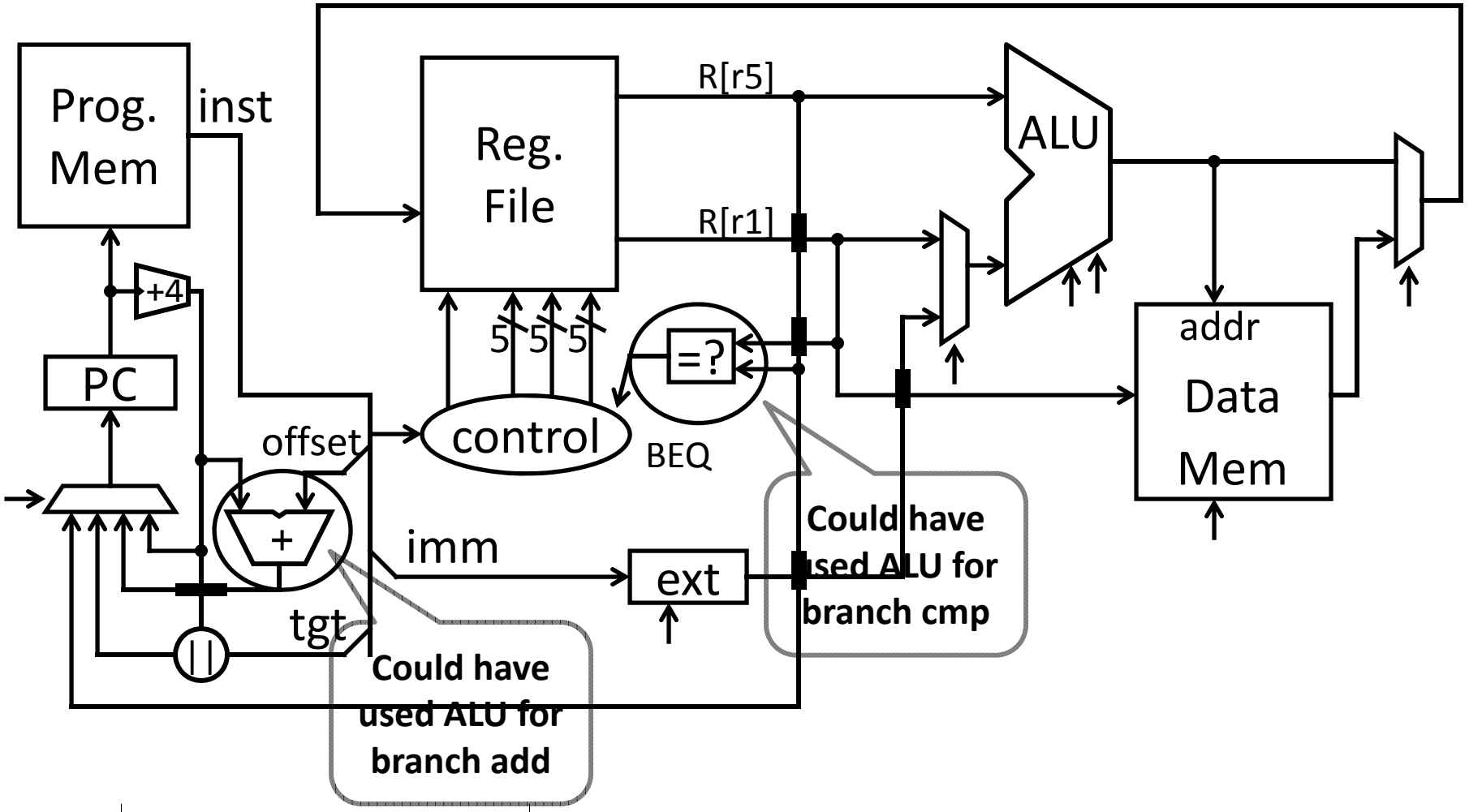
signed offsets

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 (2)

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

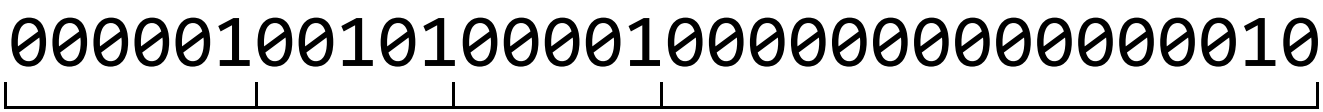
Absolute Jump



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)

Control Flow: More Branches

Conditional Jumps (cont.)

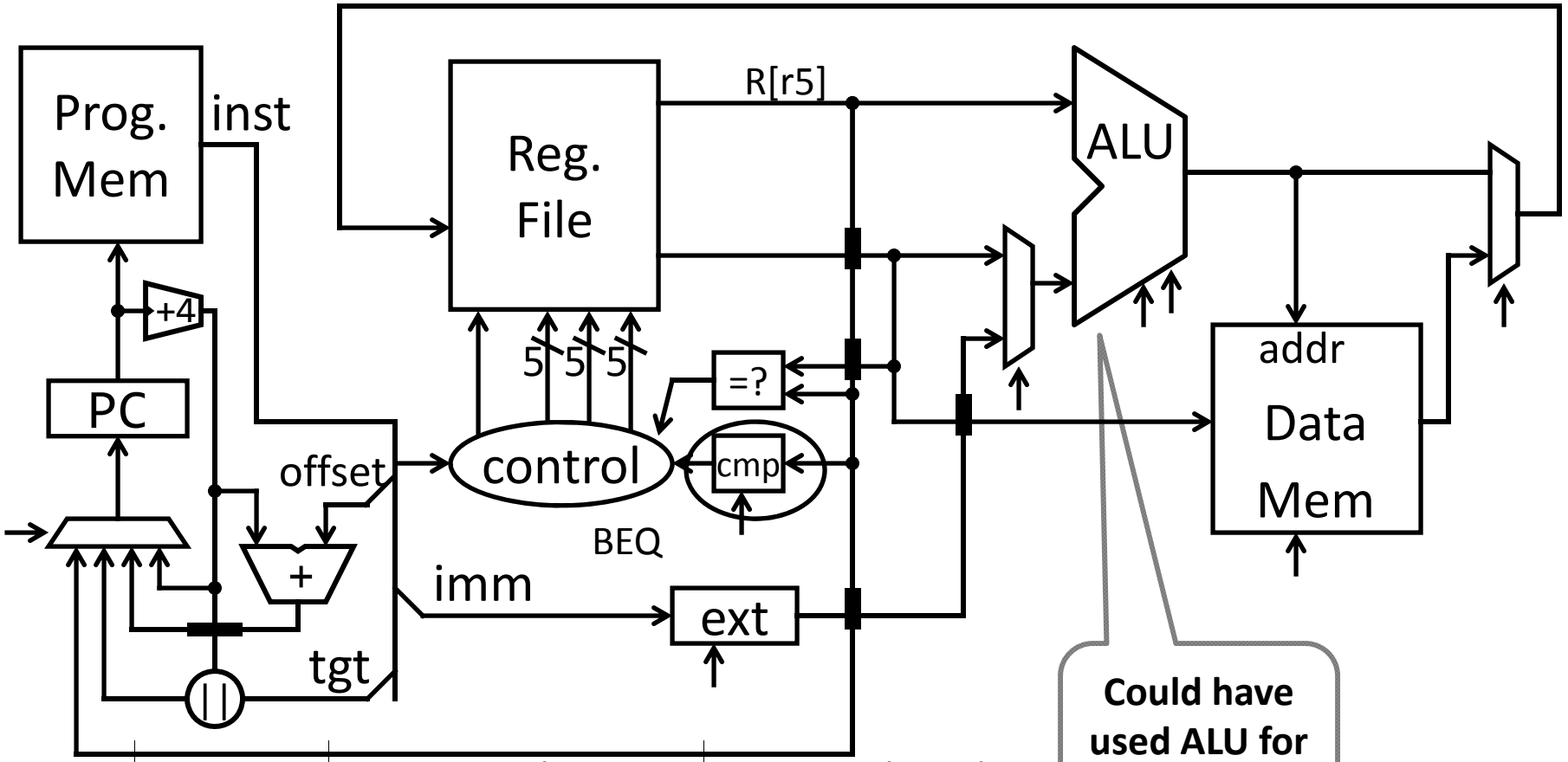


almost I-Type

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)

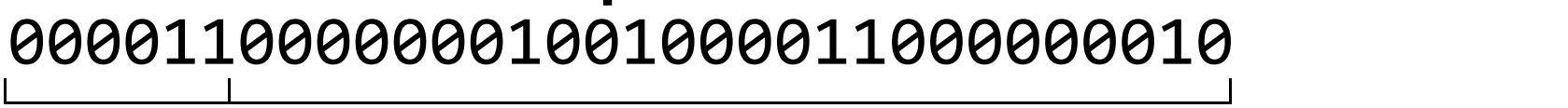
Absolute Jump



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)

Control Flow: Jump and Link

Function/procedure calls



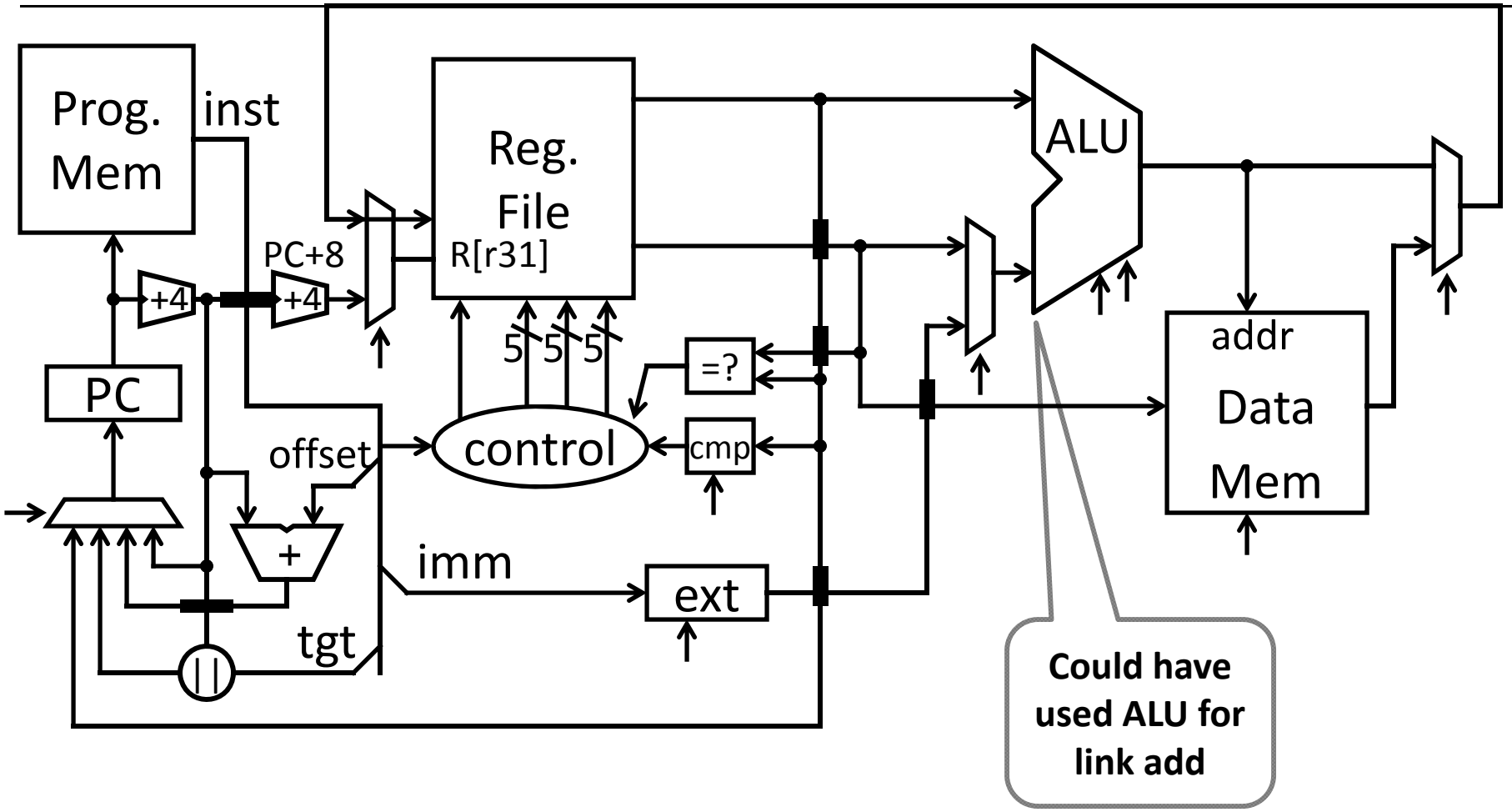
J-Type

Discuss later

op	mnemonic	description
0x3	JAL target	r31 = PC+8 (+8 due to branch delay slot) $PC = (PC+4)_{31..28} \,||\, (target << 2)$

op	mnemonic	description
0x2	J target	$PC = (PC+4)_{31..28} \,||\, (target << 2)$

Absolute Jump



op	mnemonic	description
0x3	JAL target	r31 = PC+8 (+8 due to branch delay slot) PC = $(PC+4)_{31..28}$ || (target << 2)

Goals for today

MIPS Datapath

- Memory layout
- Control Instructions

Performance

- CPI (Cycles Per Instruction)
- MIPS (Instructions Per Cycle)
- Clock Frequency

Pipelining

- Intuition on Latency vs throughput

Next Goal

How do we measure performance?

What is the performance of a single cycle CPU?

See: P&H 1.4

Performance

How to measure performance?

- GHz (billions of cycles per second)
- MIPS (millions of instructions per second)
- MFLOPS (millions of floating point operations per second)
- Benchmarks (SPEC, TPC, …)

Metrics

- latency: how long to finish my program
- throughput: how much work finished per unit time

Latency: Processor Clock Cycle

Critical Path

- Longest path from a register output to a register input
- Determines minimum cycle, maximum clock frequency

How do we make the CPU perform better

(e.g. cheaper, cooler, go “faster”, …)?

- Optimize for delay on the critical path
- Optimize for size / power / simplicity elsewhere

Latency: Optimize Delay on Critical Path

E.g. Adder performance

32 Bit Adder Design	**Space**	**Time**
Ripple Carry	≈ 300 gates	≈ 64 gate delays
2-Way Carry-Skip	≈ 360 gates	≈ 35 gate delays
3-Way Carry-Skip	≈ 500 gates	≈ 22 gate delays
4-Way Carry-Skip	≈ 600 gates	≈ 18 gate delays
2-Way Look-Ahead	≈ 550 gates	≈ 16 gate delays
Split Look-Ahead	≈ 800 gates	≈ 10 gate delays
Full Look-Ahead	≈ 1200 gates	≈ 5 gate delays

Throughput: Multi-Cycle Instructions

Strategy 2

- Multiple cycles to complete a single instruction

E.g: Assume:

- load/store: 100 ns ⟵ 10 MHz
- arithmetic: 50 ns ⟵ 20 MHz
- branches: 33 ns ⟵ 30 MHz

ms = 10^{-3} second
us = 10^{-6} seconds
ns = 10^{-9} seconds

Multi-Cycle CPU

30 MHz (33 ns cycle) with

- 3 cycles per load/store
- 2 cycles per arithmetic
- 1 cycle per branch

Faster than Single-Cycle CPU?

10 MHz (100 ns cycle) with

- 1 cycle per instruction

Cycles Per Instruction (CPI)

Instruction mix for some program P, assume:

- 25% load/store (3 cycles / instruction)
- 60% arithmetic (2 cycles / instruction)
- 15% branches (1 cycle / instruction)

Multi-Cycle performance for program P:

3 * .25 + 2 * .60 + 1 * .15 = 2.1

average *cycles per instruction* (CPI) = 2.1

Multi-Cycle @ 30 MHz

Single-Cycle @ 10 MHz

800 MHz PIII “faster” than 1 GHz P4

Example

Goal: Make Multi-Cycle @ 30 MHz CPU (15MIPS) run 2x faster by making arithmetic instructions faster

Instruction mix (for P):

- 25% load/store, CPI = 3
- 60% arithmetic, CPI = 2
- 15% branches, CPI = 1

Amdahl's Law

Amdahl's Law

$$\text{Execution time after improvement} = \frac{\text{execution time affected by improvement}}{\text{amount of improvement}} + \text{execution time unaffected}$$

Or:

Speedup is limited by popularity of improved feature

Corollary:

Make the common case fast

Caveat:

Law of diminishing returns

Administrivia

Required: partner for group project

Project1 (PA1) and Homework2 (HW2) are both out

PA1 Design Doc and HW2 due in one week, start early

Work alone on HW2, but in group for PA1

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)

Use your resources

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

Administrivia

Check online syllabus/schedule

- http://www.cs.cornell.edu/Courses/CS3410/2013sp/schedule.html

Slides and Reading for lectures

Office Hours

Homework and Programming Assignments

Prelims (in evenings):

- Tuesday, February 26th
- Thursday, March 28th
- Thursday, April 25th

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.

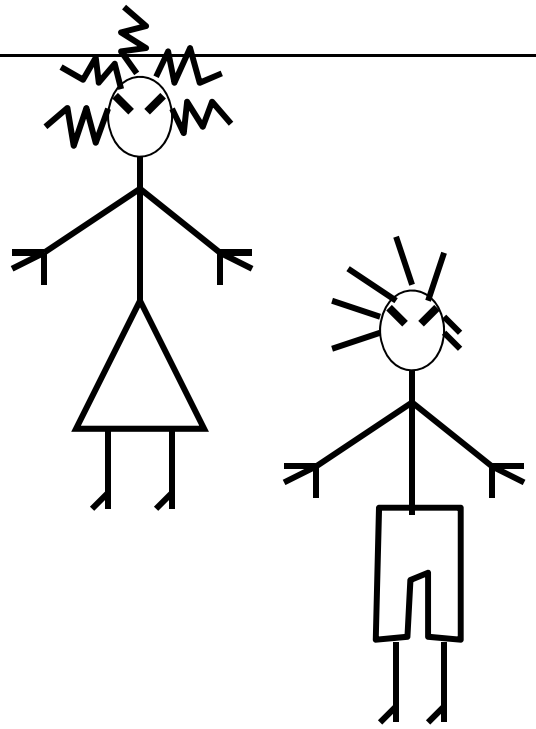
Pipelining

See: P&H Chapter 4.5

The Kids

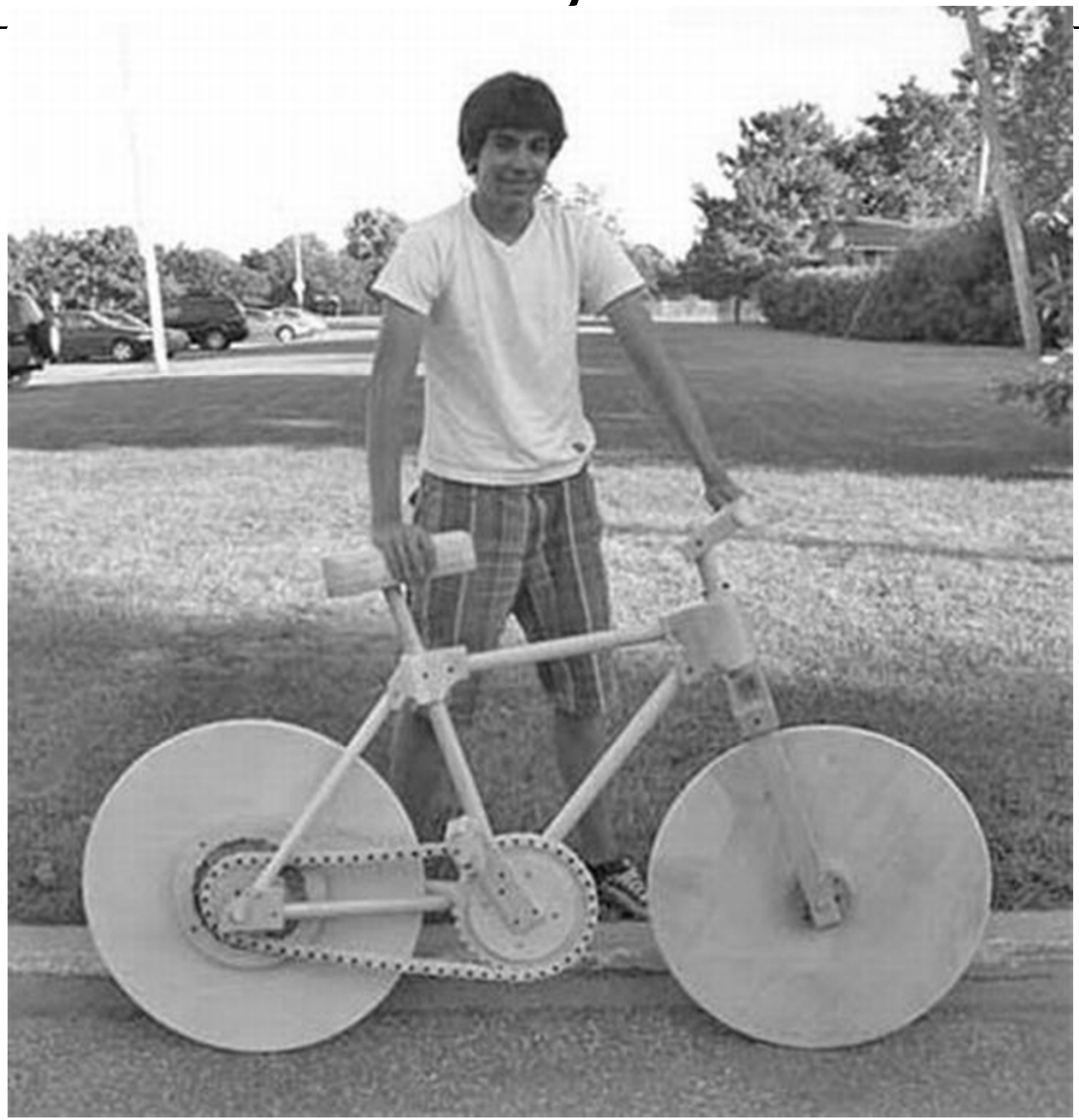
Alice

Bob



They don’t always get along…

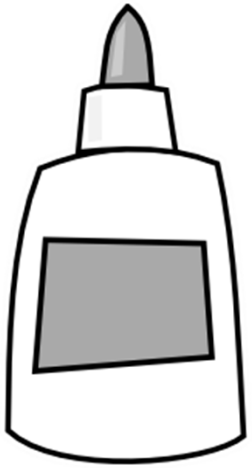
The Bicycle



The Materials

Saw

Drill

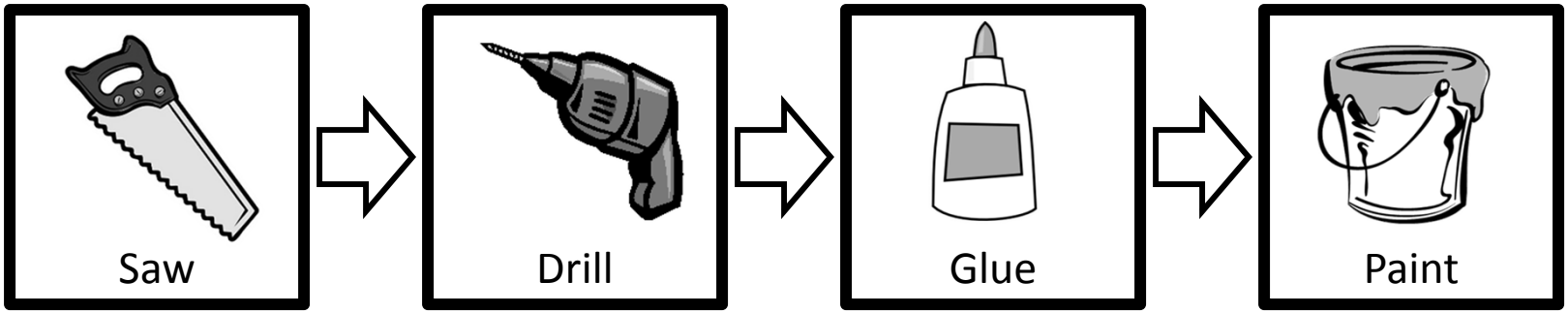


Glue

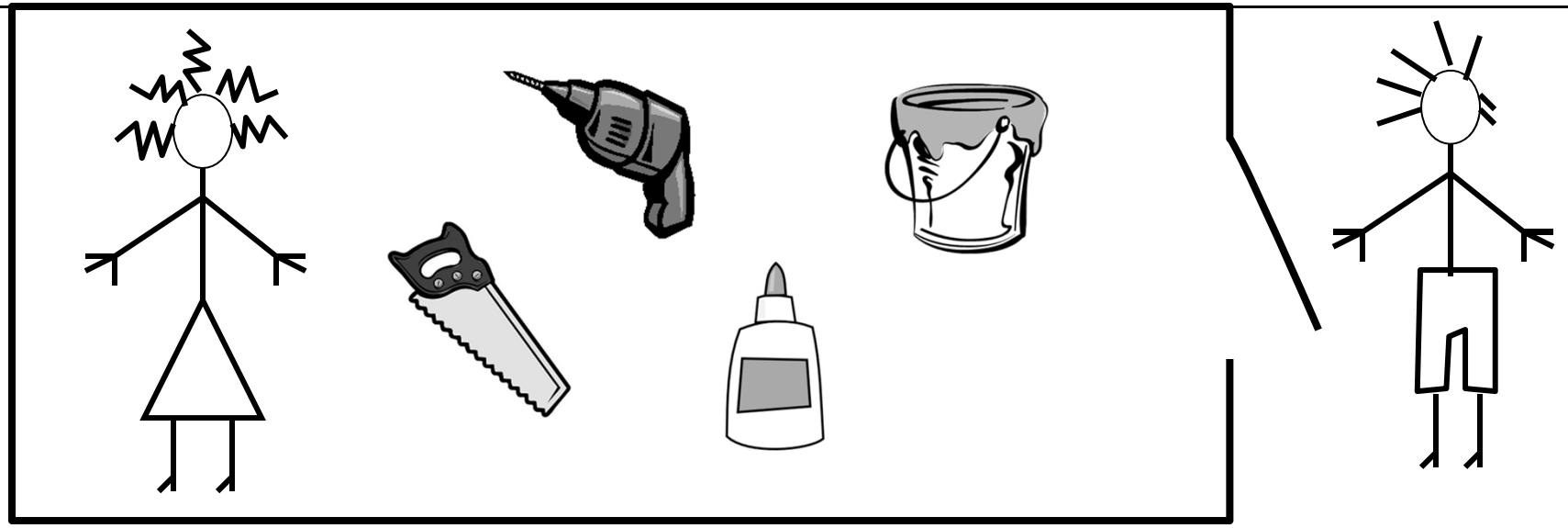
Paint

The Instructions

N pieces, each built following same sequence:



Design 1: Sequential Schedule



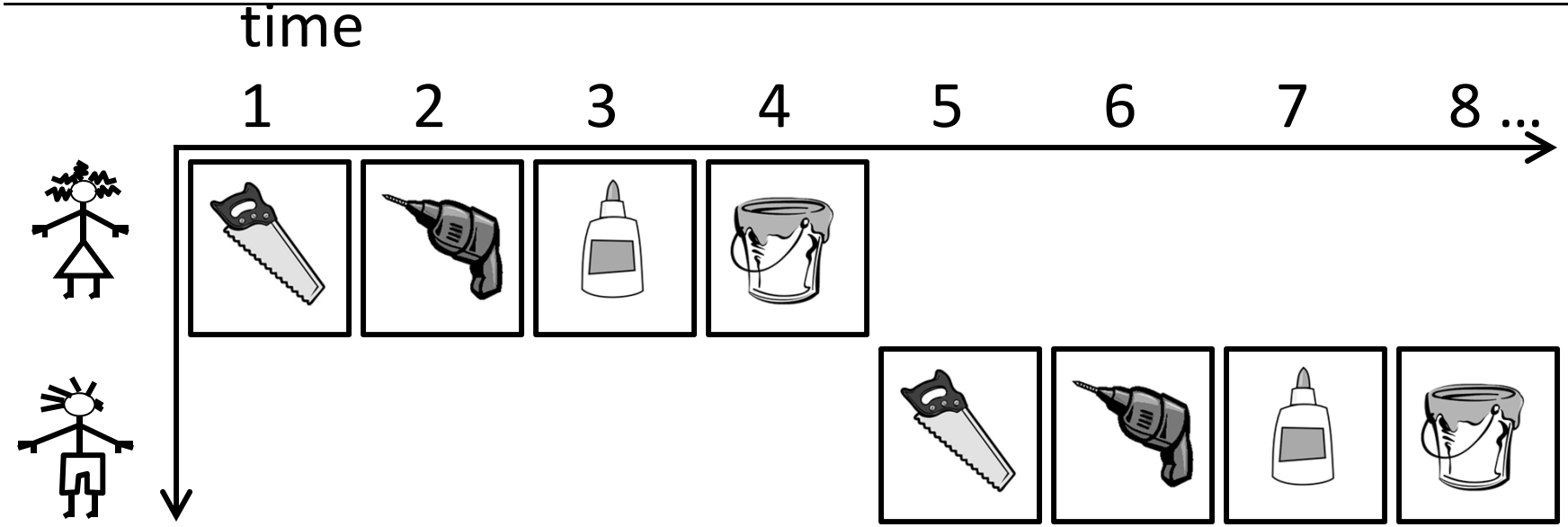
Alice owns the room

Bob can enter when Alice is finished

Repeat for remaining tasks

No possibility for conflicts

Sequential Performance



Latency:

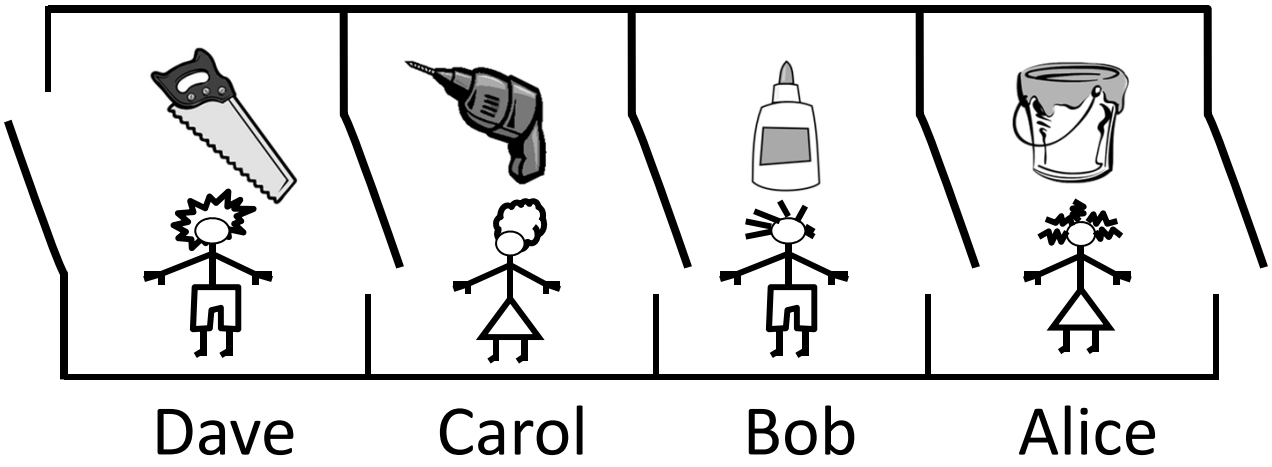
Throughput:

Concurrency:

Can we do better?

Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*



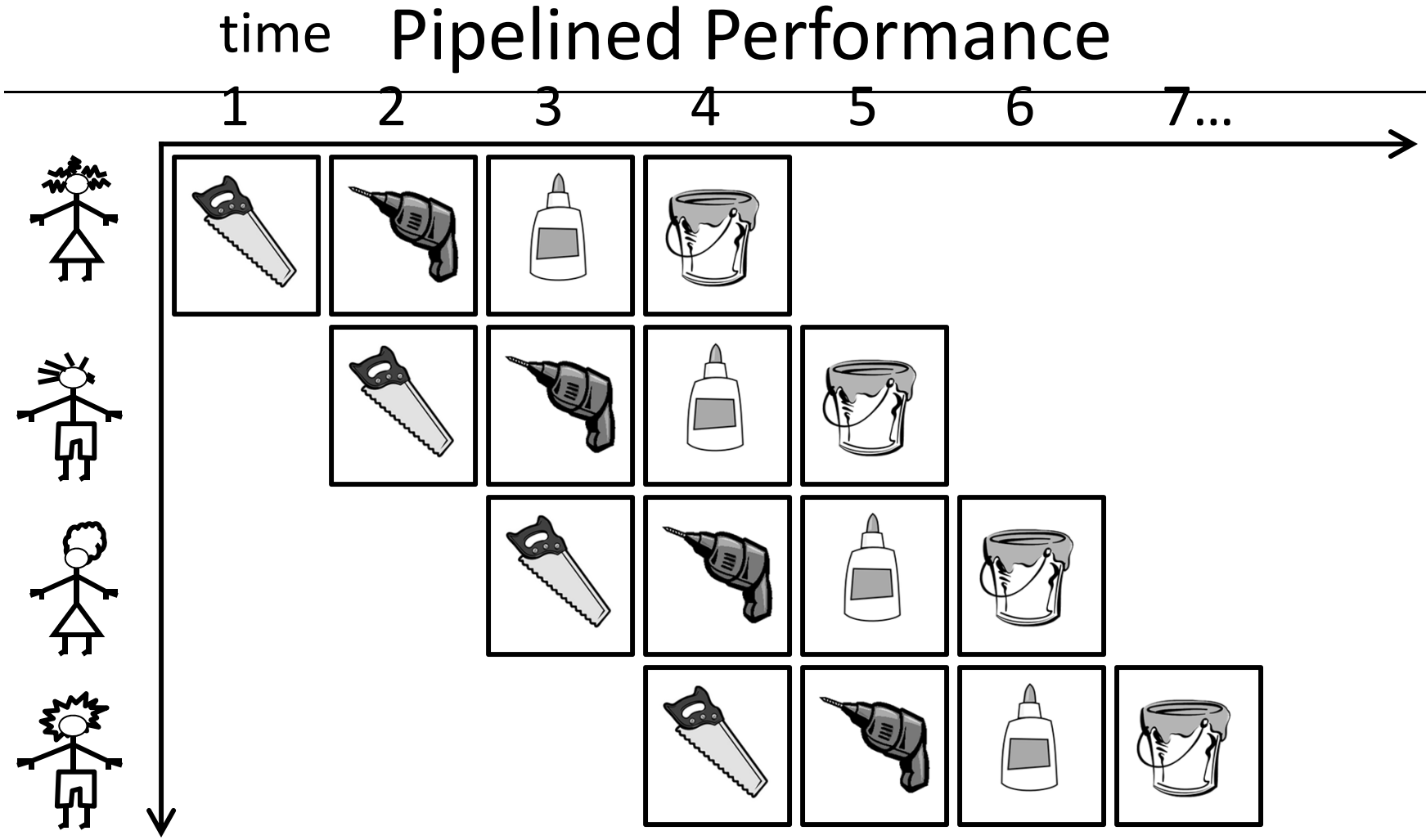
One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep

Pipelined Performance



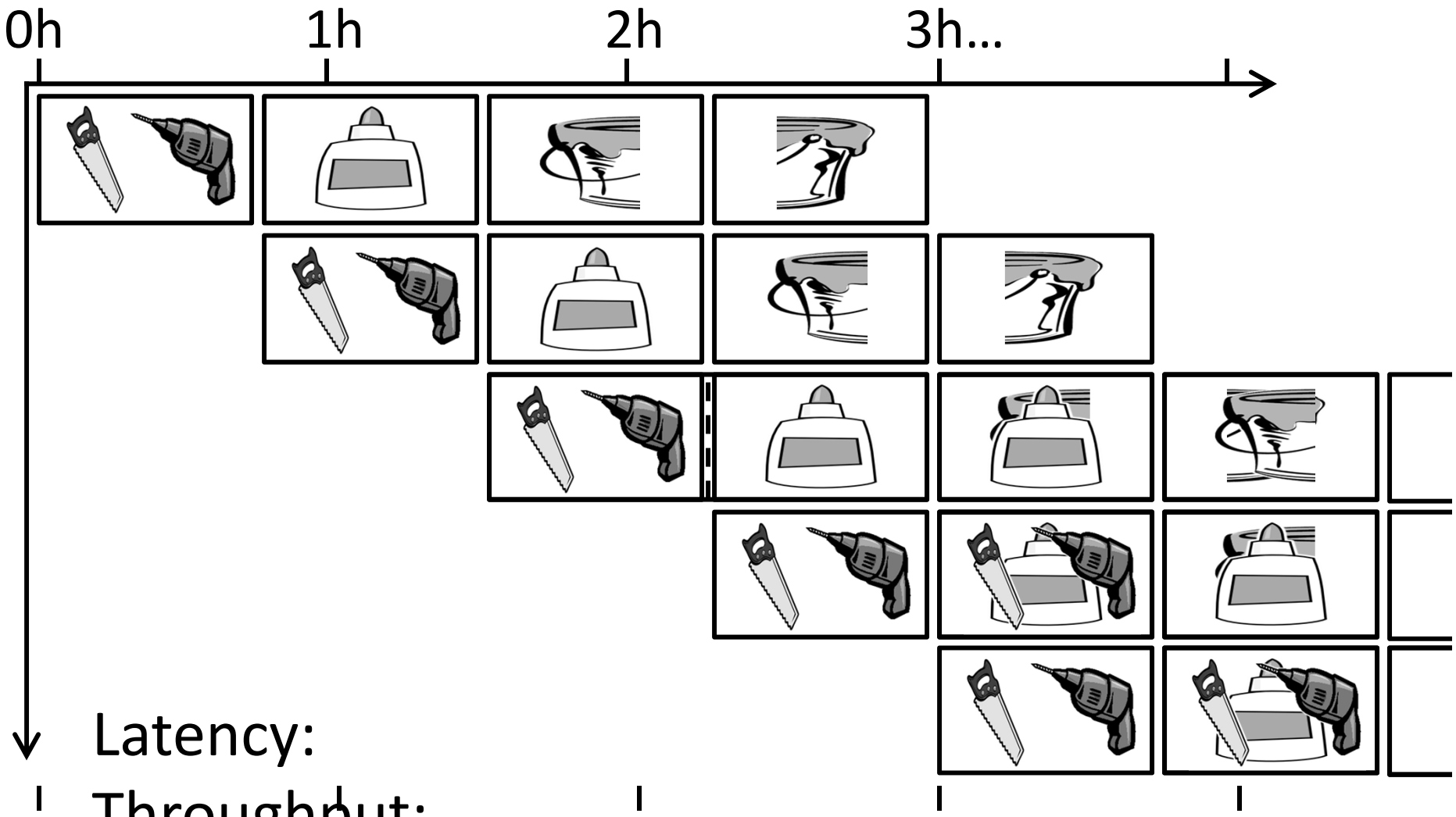
Latency:

Throughput:

Concurrency:

Pipeline Hazards

Q: What if glue step of task 3 depends on output of task 1?



Lessons

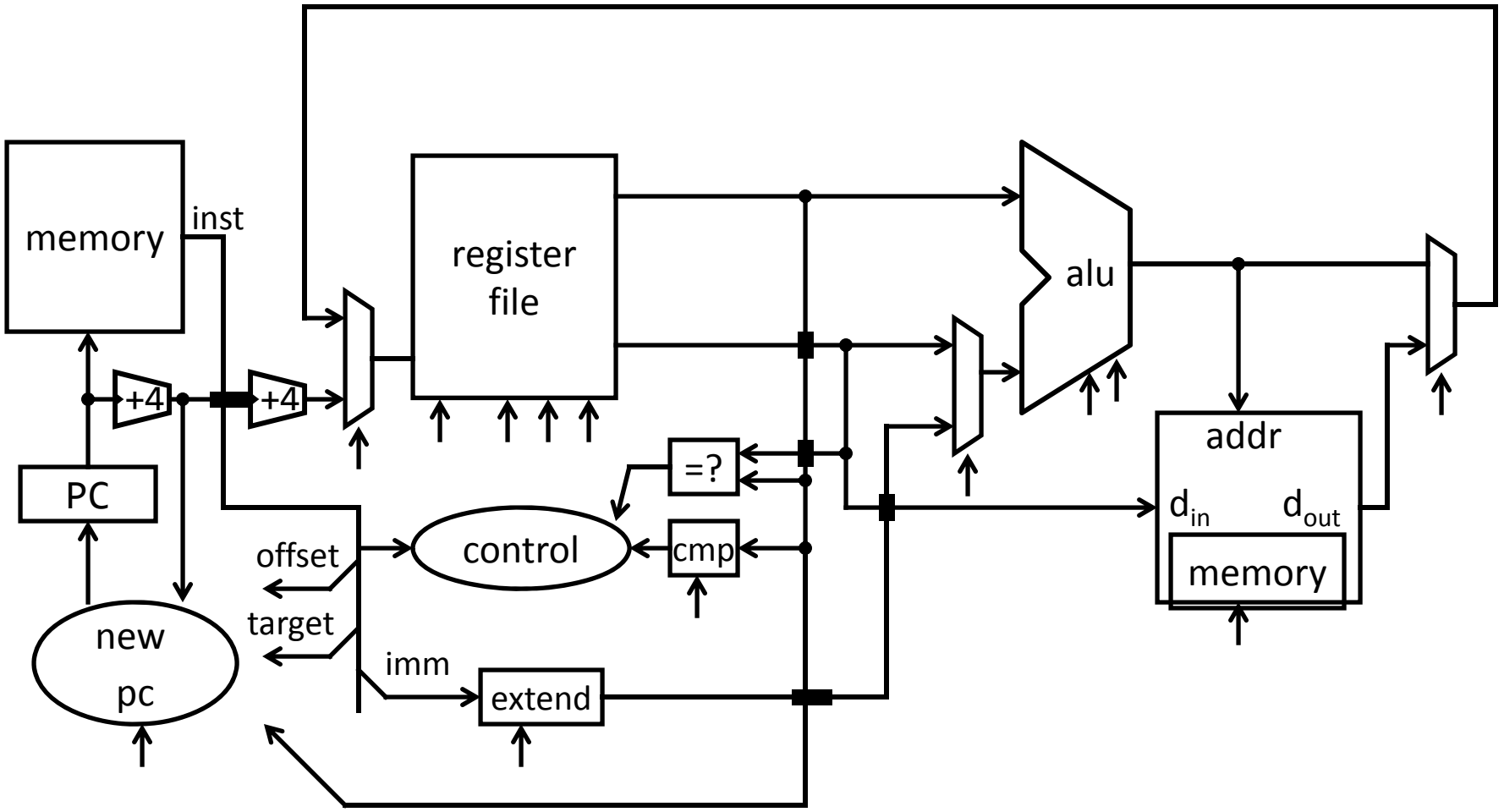
Principle:

Throughput increased by parallel execution

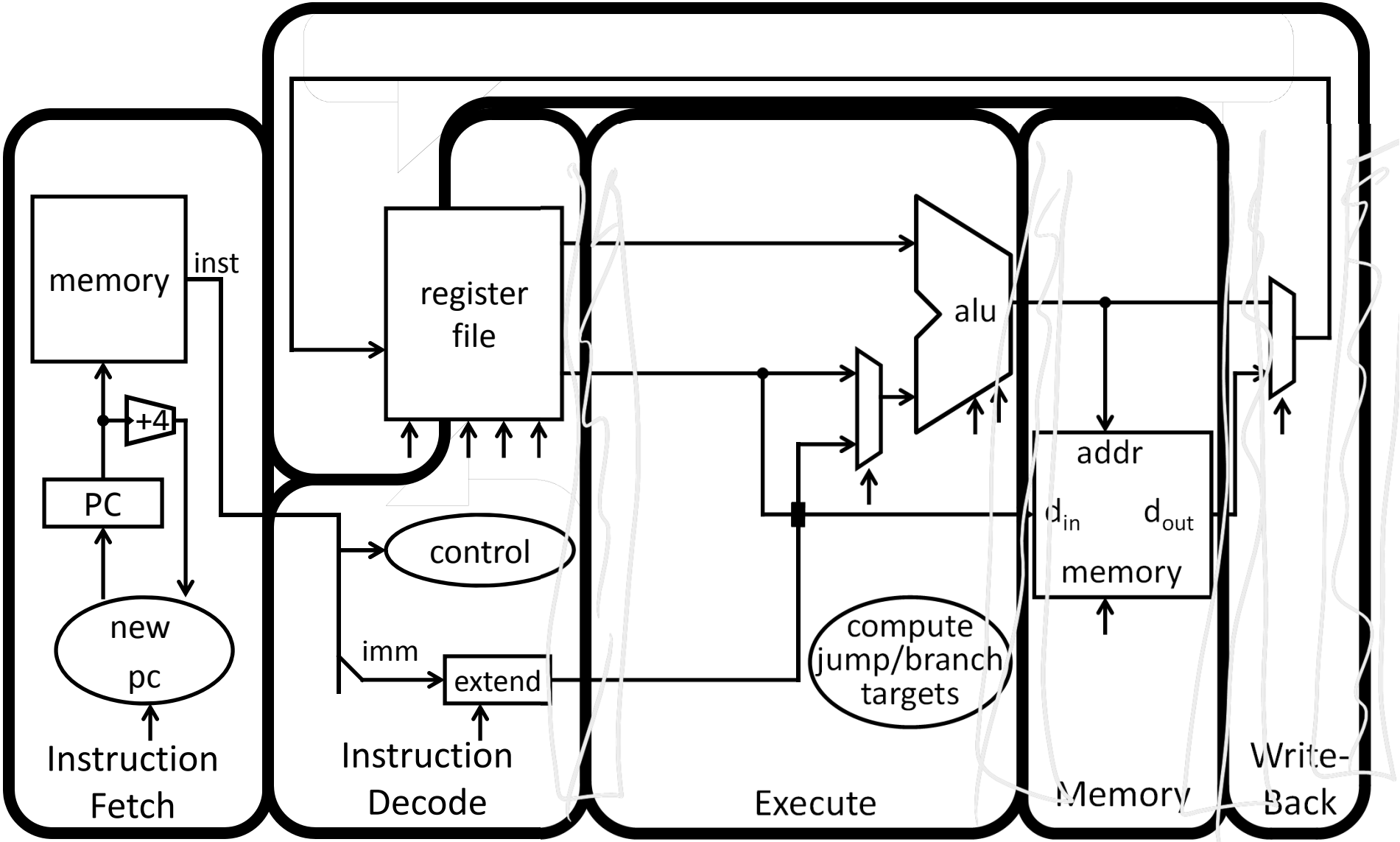
Pipelining:

- Identify *pipeline stages*
- Isolate stages from each other
- Resolve pipeline *hazards* (next week)

A Processor


memory
inst
register file
alu
+4
+4
PC
=?
control
cmp
offset
target
imm
extend
new pc
addr
d_{in}
d_{out}
memory

A Processor


memory
inst
register file
alu
+4
PC
addr
d_{in}
d_{out}
memory
control
new pc
imm
extend
compute jump/branch targets
Instruction Fetch
Instruction Decode
Execute
Memory
Write-Back

Basic Pipeline

Five stage “RISC” load-store architecture

1. Instruction fetch (IF)
   - get instruction from memory, increment PC
2. Instruction Decode (ID)
   - translate opcode into control signals and read registers
3. Execute (EX)
   - perform ALU operation, compute jump/branch targets
4. Memory (MEM)
   - access memory if needed
5. Writeback (WB)
   - update register file

Principles of Pipelined Implementation

Break instructions across multiple clock cycles (five, in this case)

Design a separate stage for the execution performed during each clock cycle

Add pipeline registers (flip-flops) to isolate signals between different stages