CPU Performance
Pipelined CPU

Hakim Weatherspoon

CS 3410, Spring 2012

Computer Science

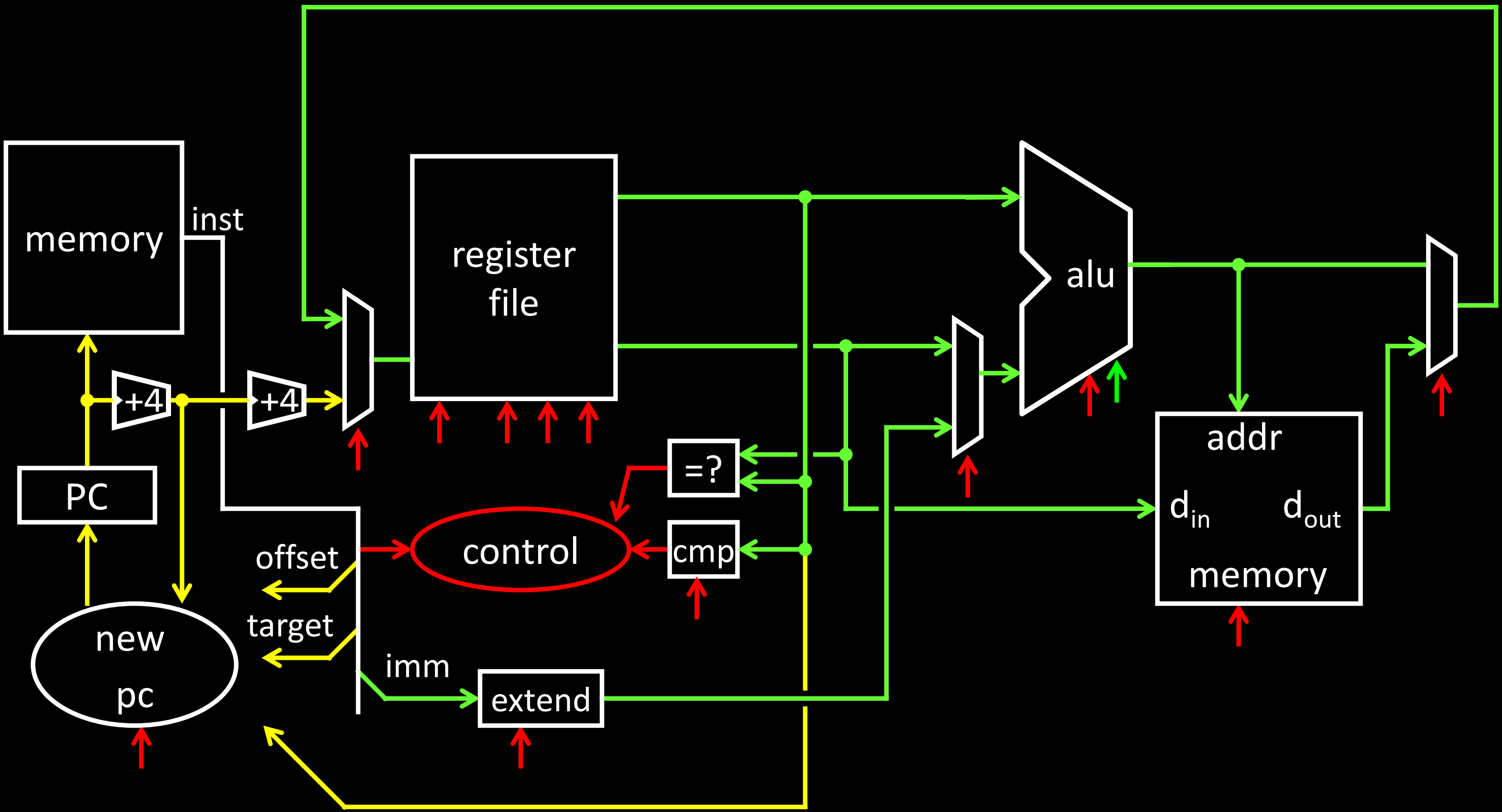
Cornell University

See P&H Chapters 1.4 and 4.5

“In a major matter, no details are small”

French Proverb

Big Picture: Building a Processor



A Single cycle processor

MIPS instruction formats

All MIPS instructions are 32 bits long, has 3 formats

R-type	op	rs	rt	rd	shamt	func
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-type	op	rs	rt	immediate
	6 bits	5 bits	5 bits	16 bits

J-type	op	immediate (target address)
	6 bits	26 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

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

Control flow

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

Goals for today

Review

- Remaining Branch Instructions

Performance

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

Pipelining

- Latency vs throuput

Memory Layout and

A Simple CPU: remaining branch instructions

Memory Layout

Examples (big/little endian):

r5 = 5

```
# r5 contains 5 (0x00000005)
```

r0 = 0

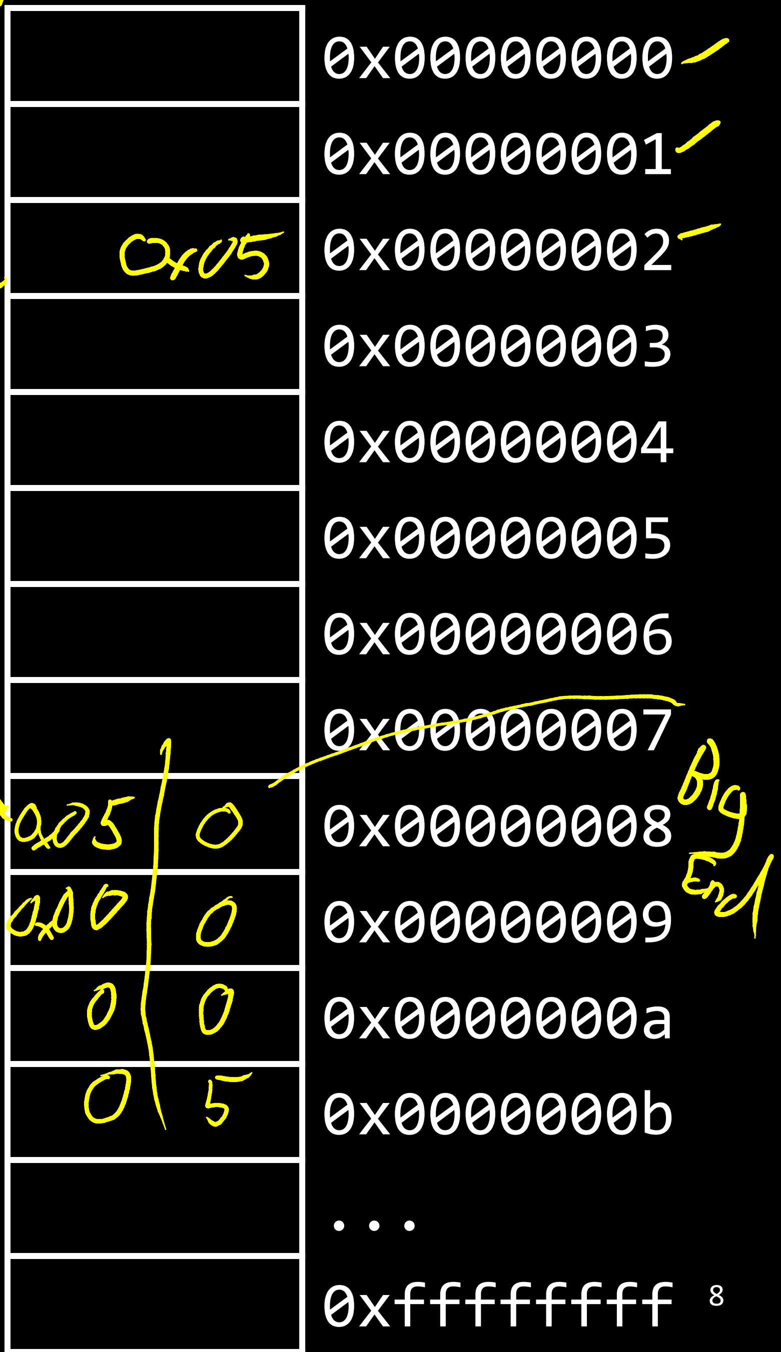
```
sb r5, 2(r0)
lb r6, 2(r0)
```

r6 = 0x05

big = MSB comes first (lowest addr)

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

Big = 0
Little = 5



	Little End	Big End
0x00000000		
0x00000001		
0x00000002	0x05	
0x00000003		
0x00000004		
0x00000005		
0x00000006		
0x00000007		
0x00000008	0x05	0
0x00000009	0x00	0
0x0000000a	0	0
0x0000000b	0	5
...		
0xffffffff		

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			

Control Flow: Jump Register

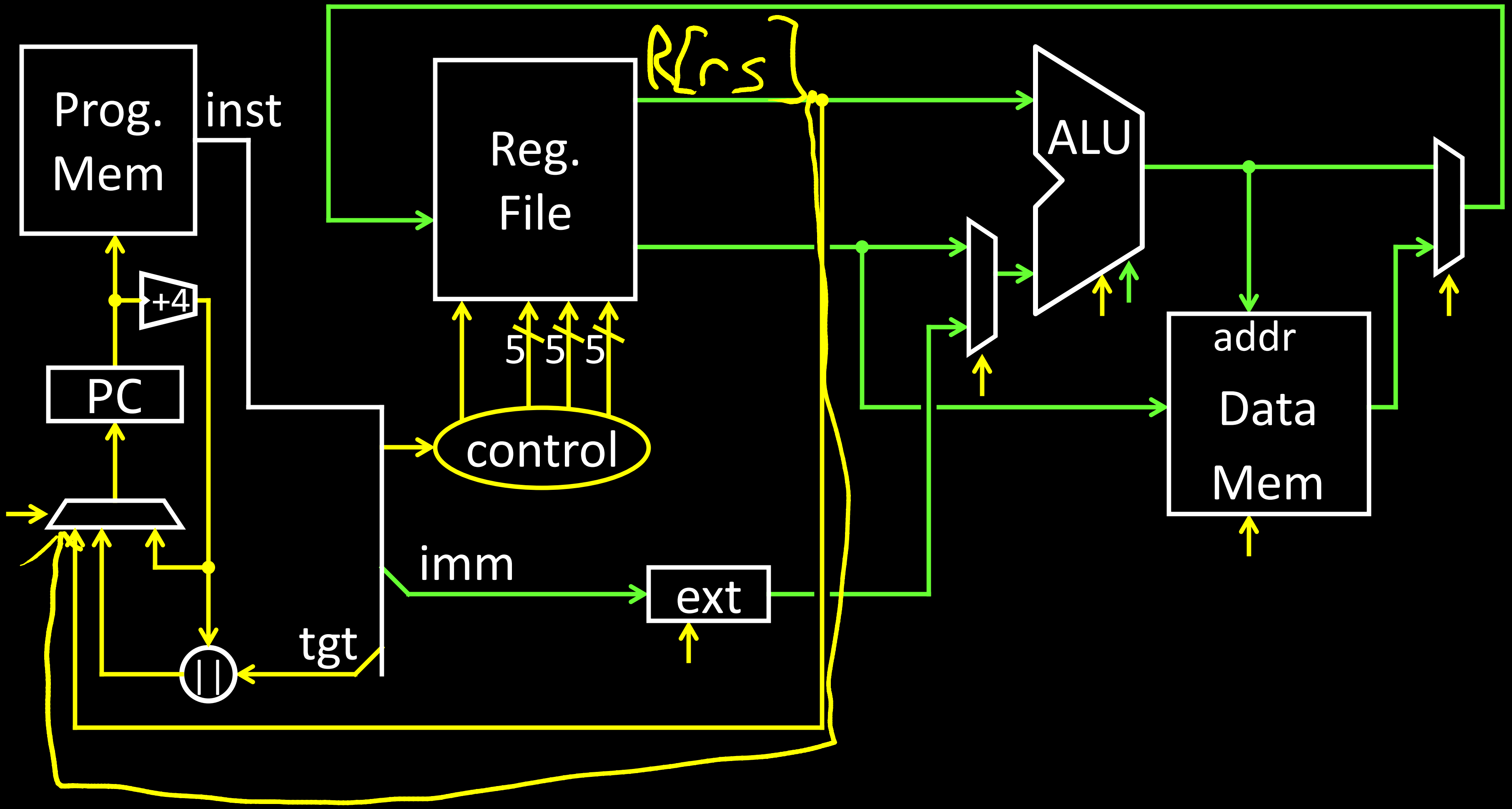
r3

000000	00011	00000	00000	00000	001000
op	rs	-	-	-	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R-Type

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 (2)

```
jump to 0xabcd1234
```

```
# assume 0 <= r3 <= 1
if (r3 == 0) jump to 0xdecafe00
else jump to 0xabcd1234
```

Control Flow: Branches

00010000101000010000000000000011

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

I-Type

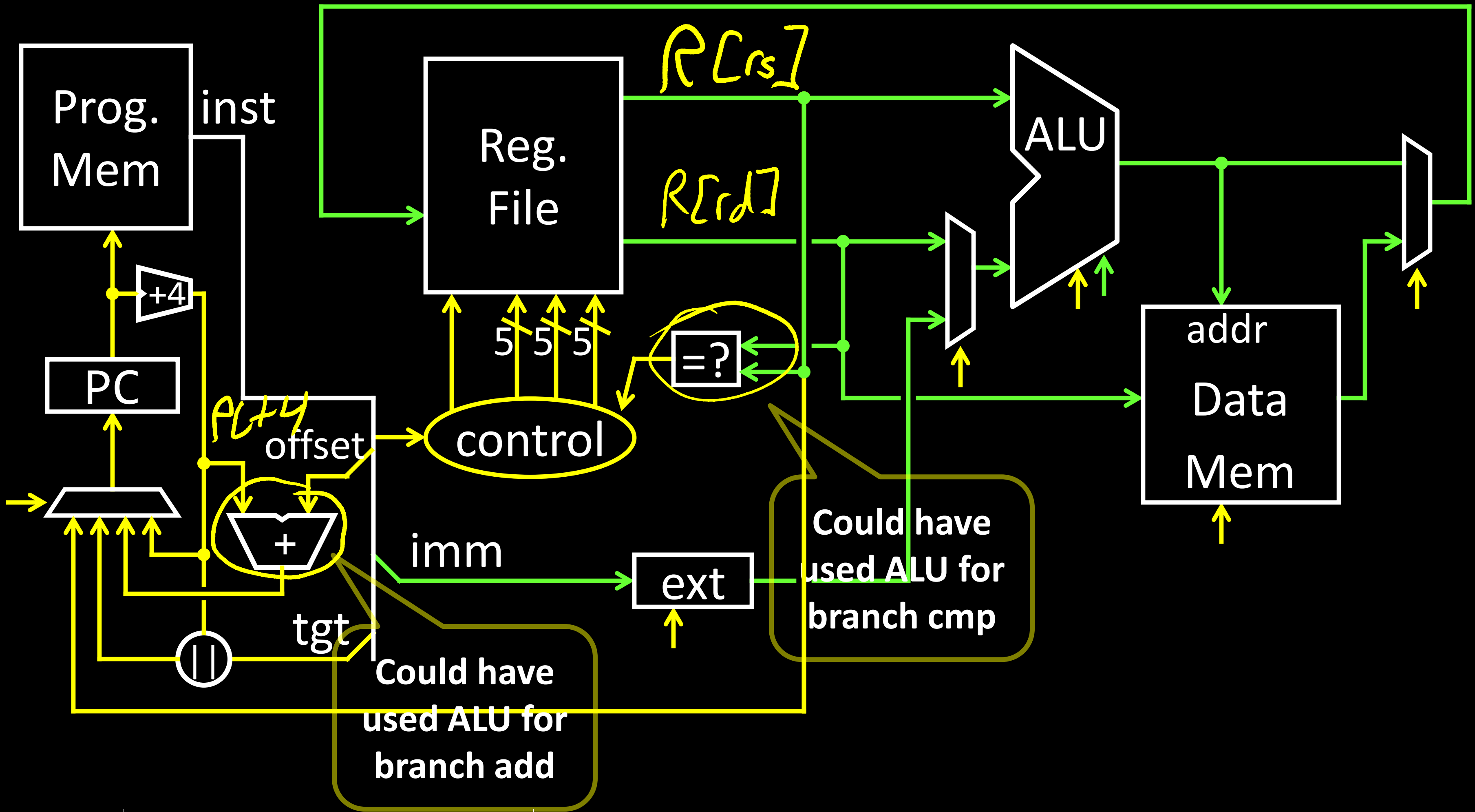
signed offsets

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 (3)

```
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.)

`00000100101000010000000000000010`

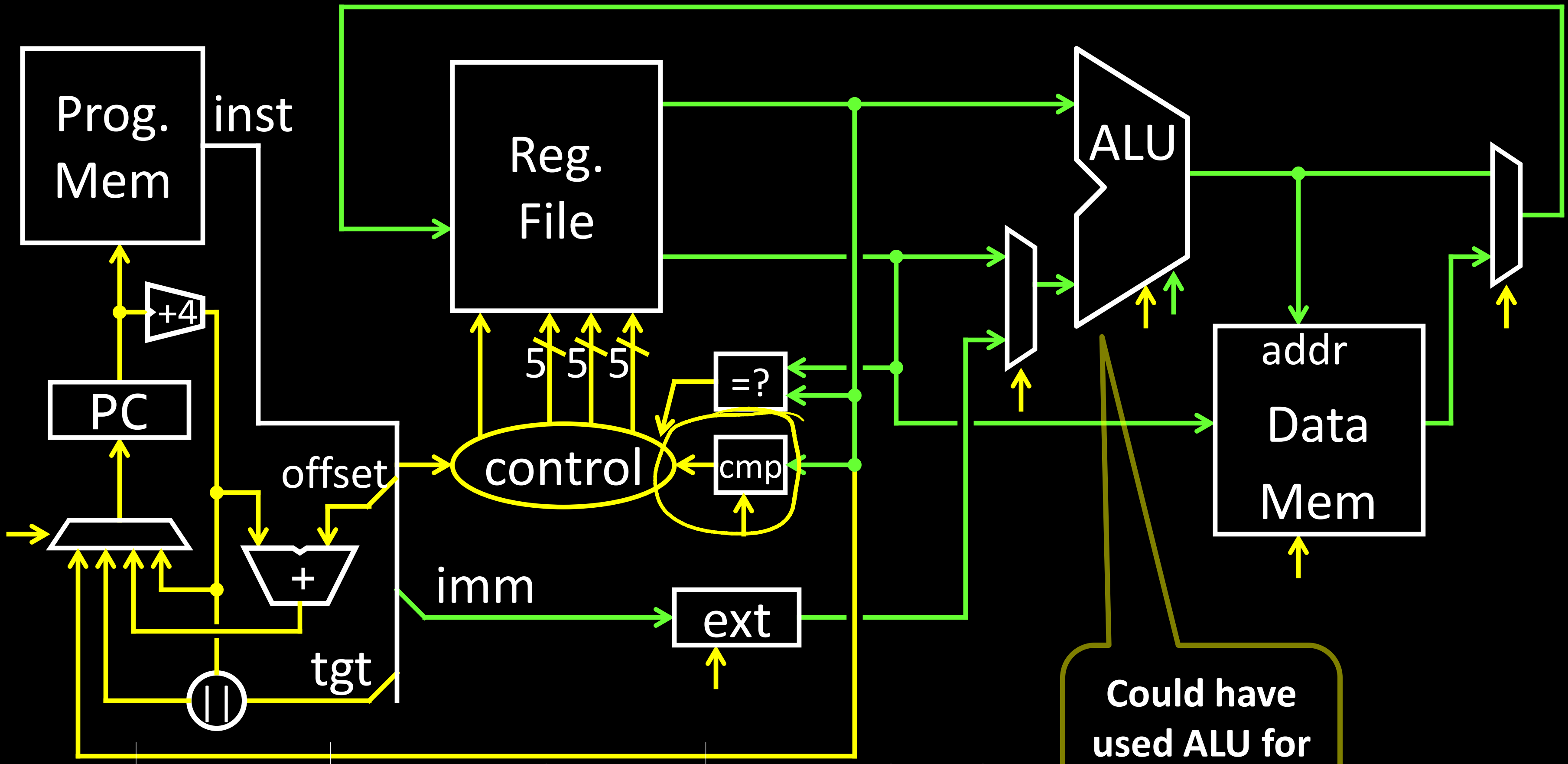
op	rs	subop	offset
6 bits	5 bits	5 bits	16 bits

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

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

000011 00000001001000011000000010

op — 6 bits

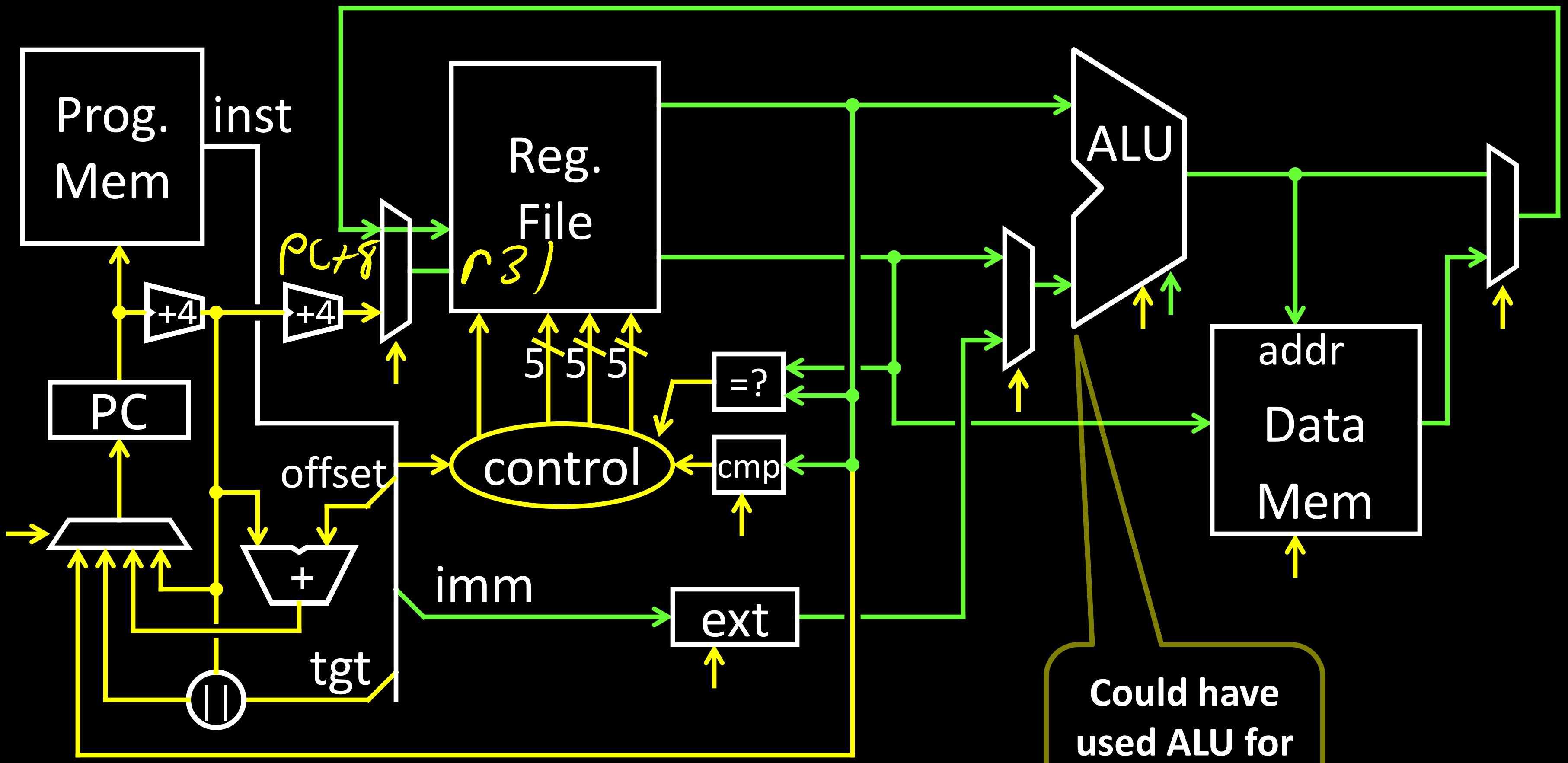
immediate — 26 bits

J-Type

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



Could have used ALU for link add

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

Performance

See: P&H 1.4

What is instruction is the longest

A) LW

B) SW

C) ADD/SUB/AND/OR/etc

D) BEQ

E) J

Design Goals

What to look for in a computer system?

- Correctness?
- Cost
  - purchase cost = f(silicon size = gate count, economics)
  - operating cost = f(energy, cooling)
  - operating cost >= purchase cost
- Efficiency
  - power = f(transistor usage, voltage, wire size, clock rate, …)
  - heat = f(power)
    - Intel Core i7 Bloomfield: 130 Watts
    - AMD Turion: 35 Watts
    - Intel Core 2 Solo: 5.5 Watts
    - Cortex-A9 Dual Core @800MHz: 0.4 Watts
- Performance
- Other: availability, size, greenness, features, …

Performance

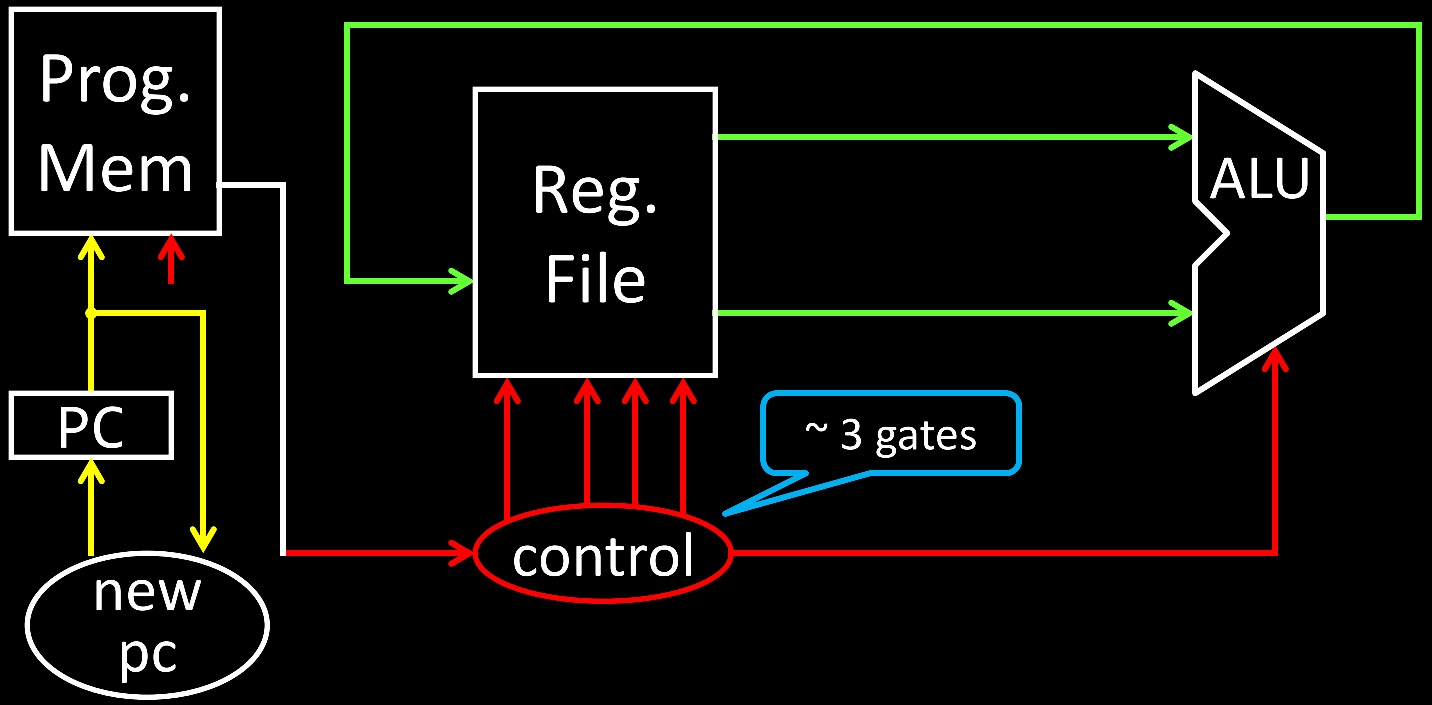
How to measure performance?

- 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

How Fast?



Assumptions:

- alu: 32 bit ripple carry + some muxes
- next PC: 30 bit ripple carry
- control: minimized for delay (~3 gates)
- transistors: 2 ns per gate
- prog,. memory: 16 ns (as much as 8 gates)
- register file: 2 ns access
- ignore wires, register setup time

Better:

- alu: 32 bit carry lookahead + some muxes (~ 9 gates)
- next PC: 30 bit carry lookahead (~ 6 gates)

Better Still:

- next PC: cheapest adder faster than 21 gate delays

All signals are stable

- 80 gates => clock period of at least 160 ns, max frequency ~6MHz

Better:

- 21 gates => clock period of at least 42 ns, max frequency ~24MHz

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

Optimization: Summary

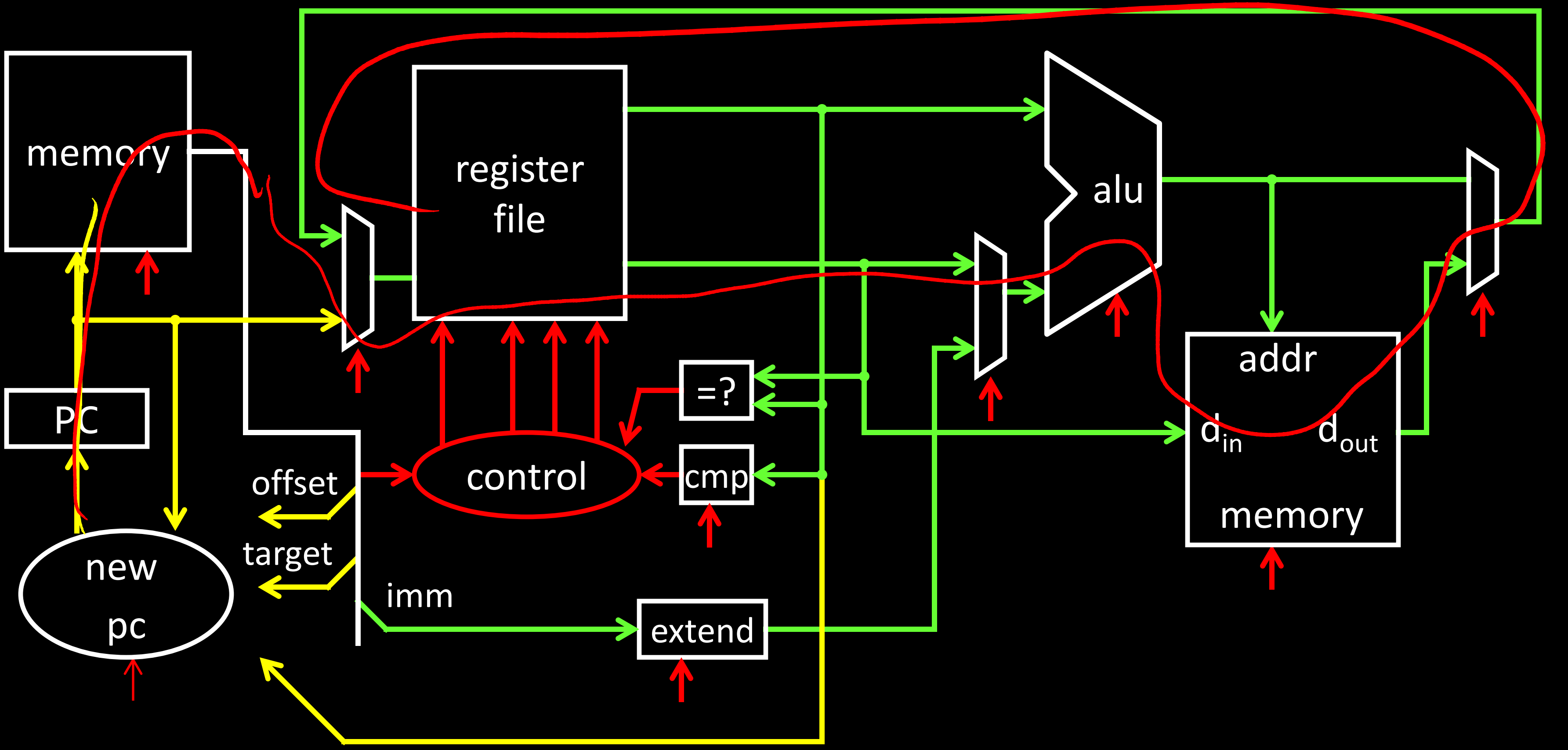
Critical Path

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

Strategy 1 (we just employed)

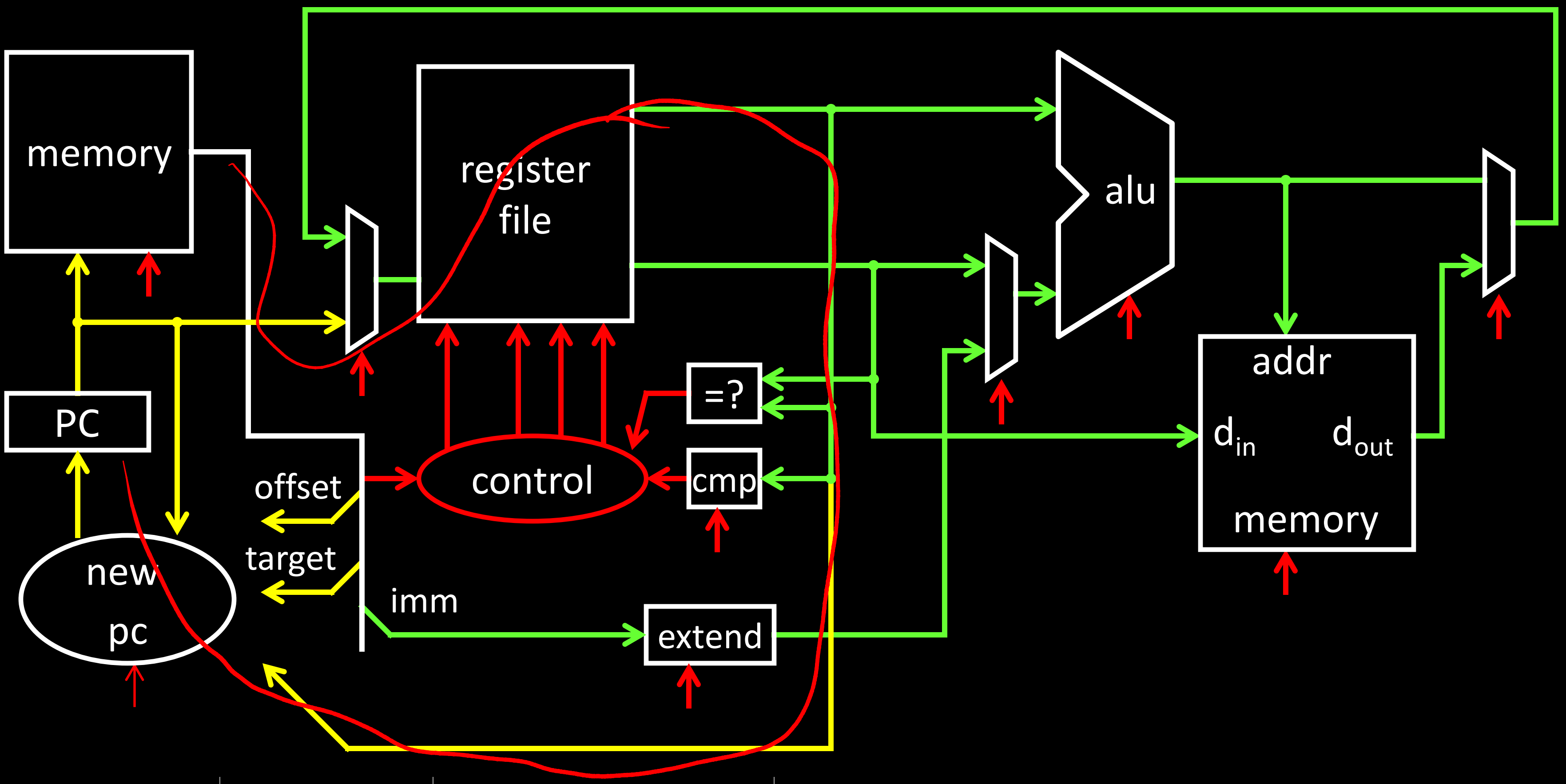
- Optimize for delay on the critical path
- Optimize for size / power / simplicity elsewhere
  - next PC

Processor Clock Cycle



op	mnemonic	description
0x20	LB rd, offset(rs)	R[rd] = sign_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]
0x2b	SW rd, offset(rs)	Mem[offset+R[rs]] = R[rd]

Processor Clock Cycle



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

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

Multi-Cycle Instructions

Strategy 2

$ms = 10^{-3}$

$\mu s = 10^{-6}$

$ns = 10^{-9}$

- Multiple cycles to complete a single instruction

E.g: Assume:

- load/store: 100 ns
- arithmetic: 50 ns
- branches: 33 ns

10 MHz

$\frac{1}{100ns}$ $\frac{1}{100 \times 10^{-9}}$ 10 MHz

20 MHz

$\frac{1}{10^{-7}} = 10^{7} = 10 \times 10^{6}$

$= 10 MHz$

30 MHz

Multi-Cycle CPU

Multi Cycle

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

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:

$CPI = 0.25 \times 3 + 0.6 \times 2 + 0.15 \times 1$

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

$= 0.75 + 1.2 + 0.15 = 2.1$

average *cycles per instruction* (CPI) = 2.1

$= 2.1$

Multi-Cycle @ 30 MHz

Single-Cycle @ 10 MHz

Single-Cycle @ 15 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

AR CPI=2	AR CPI=1	AR CPI=0.25
.75	0.75	0.75
1.2	0.6	0.15
.15	0.15	0.15
2.1	1.50	1.05

$$\frac{30\ MHz}{2.1\ CPI} = \frac{30 \cdot \frac{Mcycle}{sec}}{2.1\ \frac{cycle}{inst}} \approx 15\ \frac{Minst}{sec} \approx 15\ MIPS$$

$$\frac{30\ MHz}{1.5\ CPI} = 20\ MIPS$$

$$\frac{30}{1.05} \approx 30\ MIPS$$

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/CS3410/2012sp/schedule.html

Slides and Reading for lectures

Office Hours

Homework and Programming Assignments

Prelims (in evenings):

- Tuesday, February 28th
- Thursday, March 29th
- Thursday, April 26th

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
- 20% deducted per day late after slip days are exhausted

Regrade policy

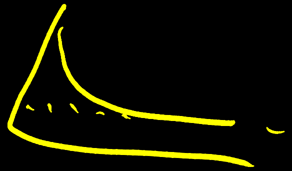
- Submit written request to lead TA,
  and lead TA will pick a different grader
- Submit another written request,
  lead TA will regrade directly
- Submit yet another written request for professor to regrade.

Amdahl's Law

Amdahl's Law

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

Pipelining

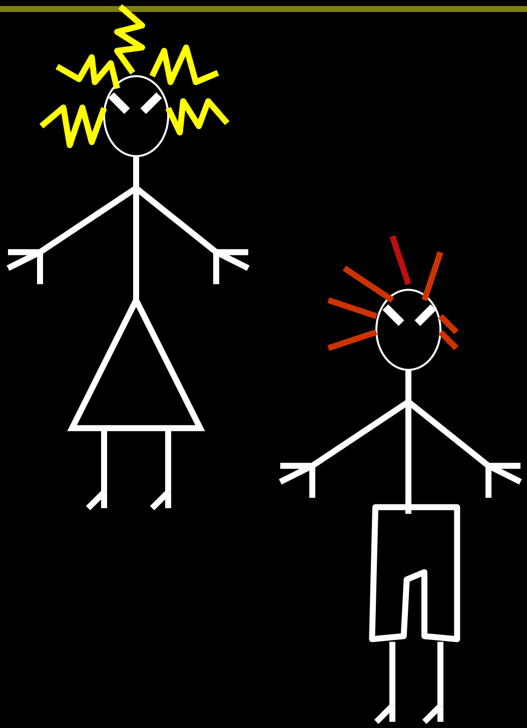
See: P&H Chapter 4.5

The Kids

Alice

Bob

They don’t always get along…



The Bicycle

The Materials

Saw

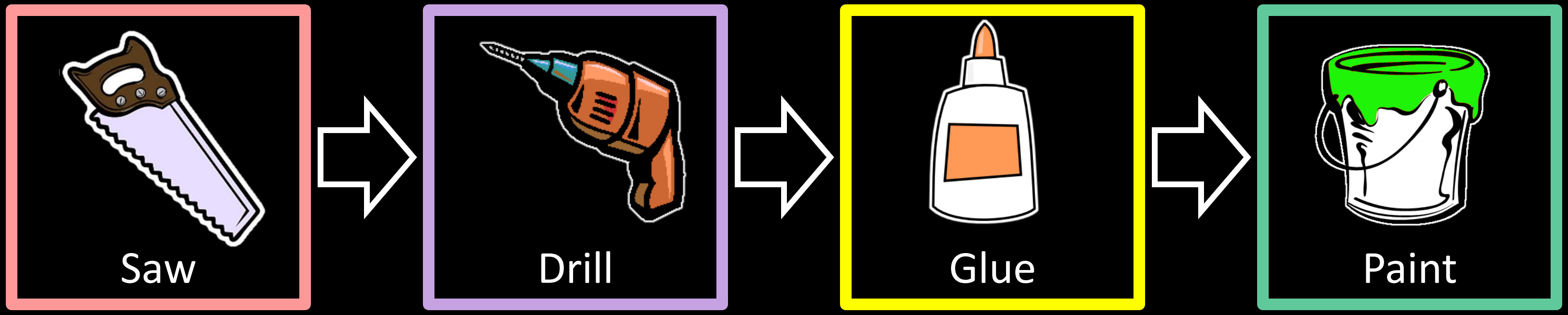
Drill

Glue

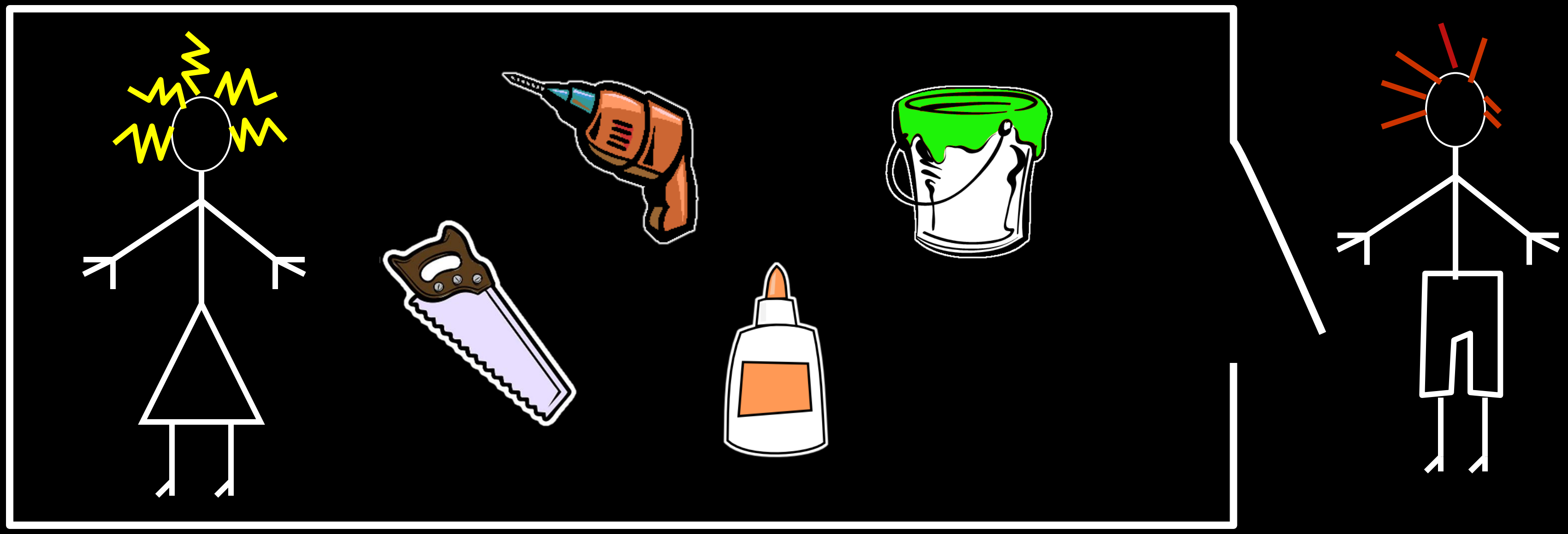
Paint

The Instructions

N pieces, each built following same sequence:



Design 1: Sequential Schedule



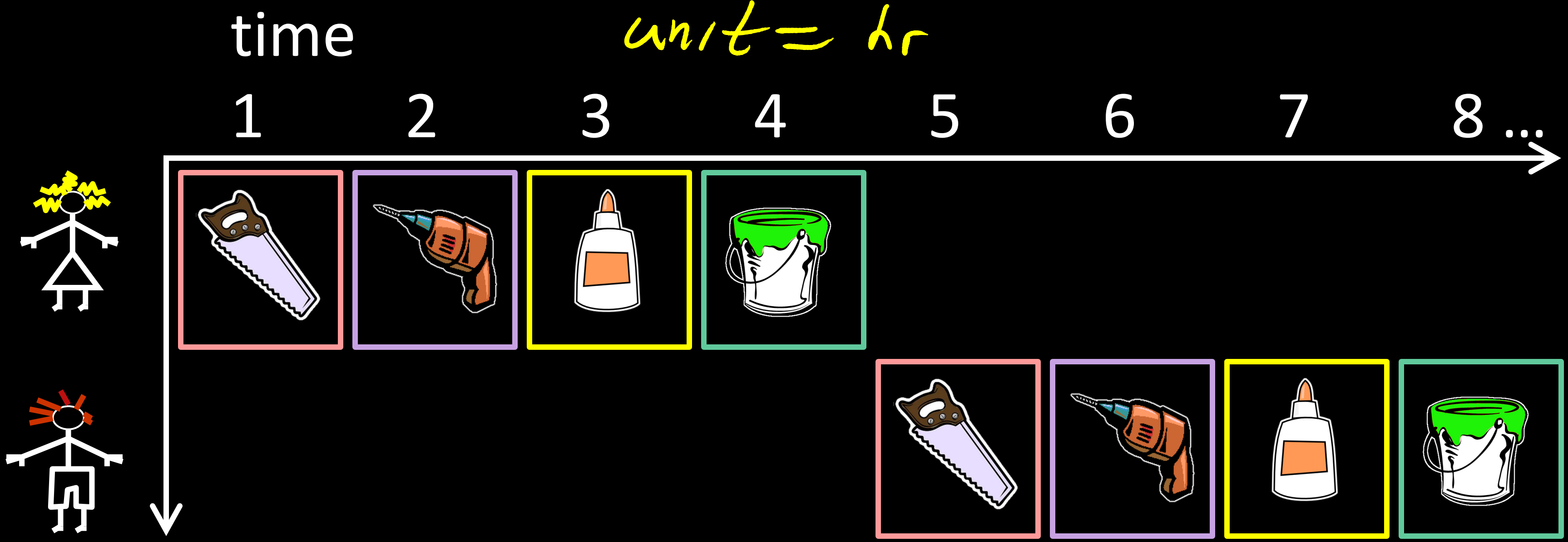
Alice owns the room

Bob can enter when Alice is finished

Repeat for remaining tasks

No possibility for conflicts

Sequential Performance



unit= hr

Latency: 4 hr / task

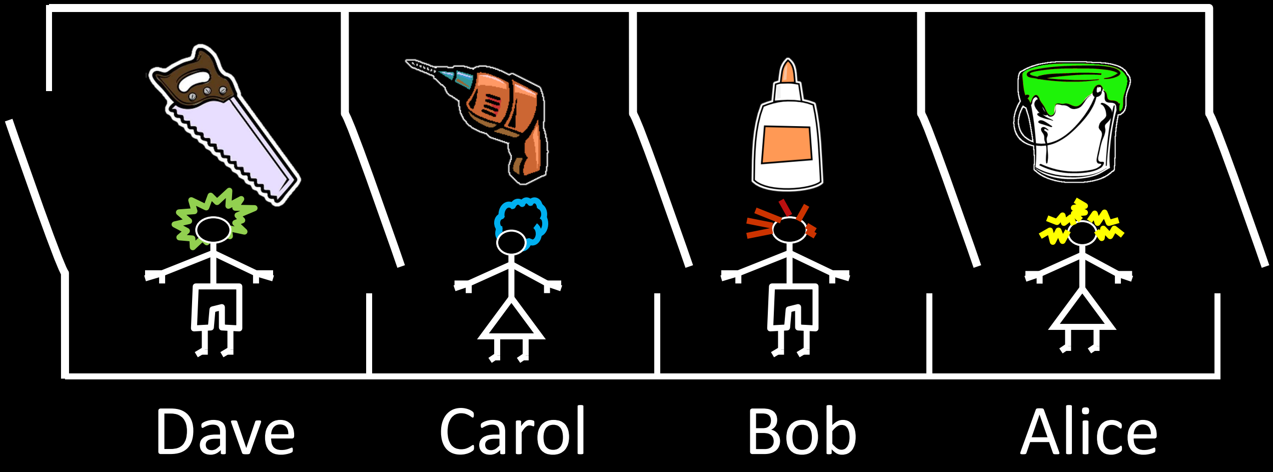
Throughput: 1 task / 4hrs

Concurrency: 1

Can we do better?

Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*



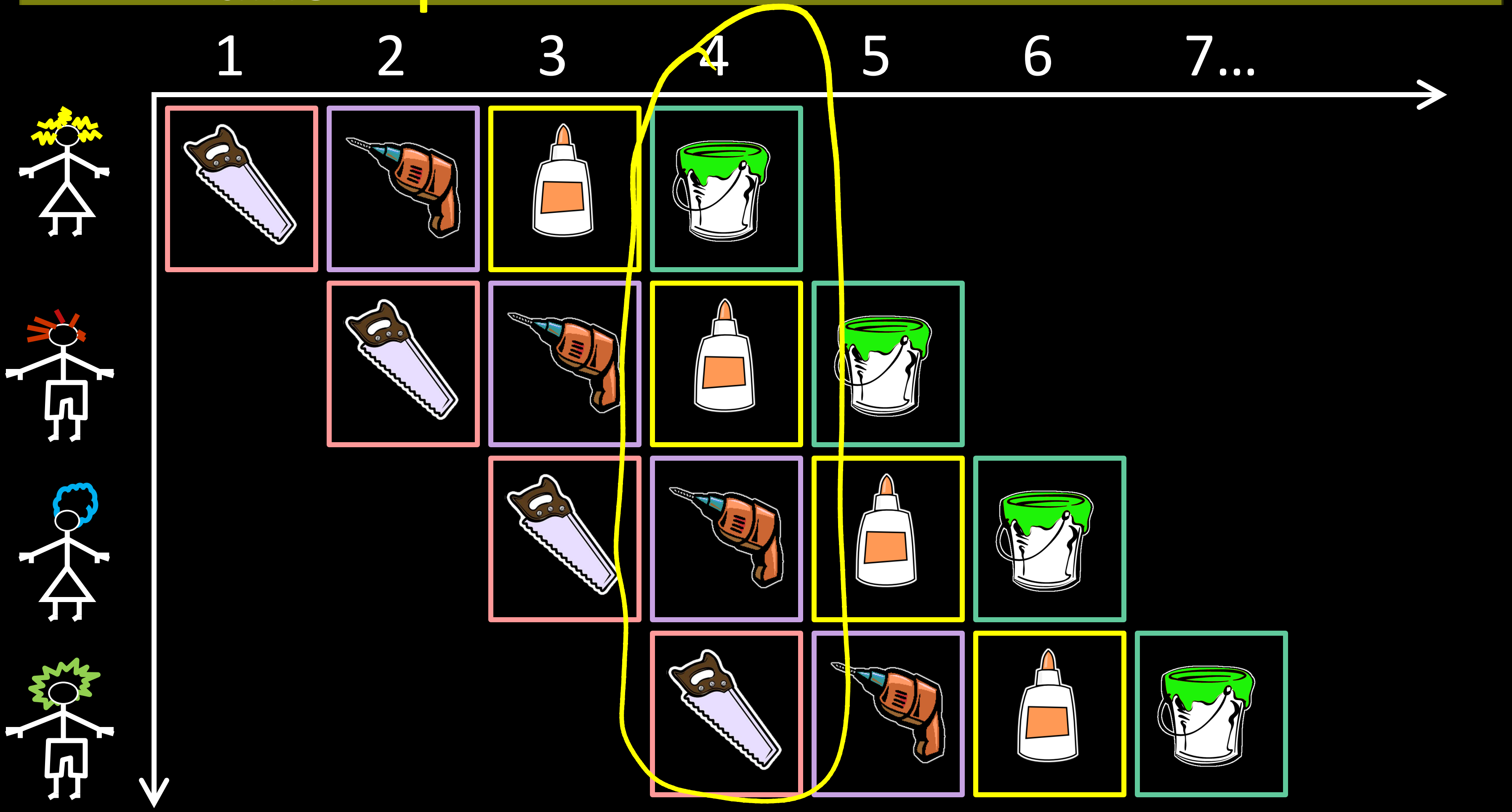
One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep

Pipelined Performance



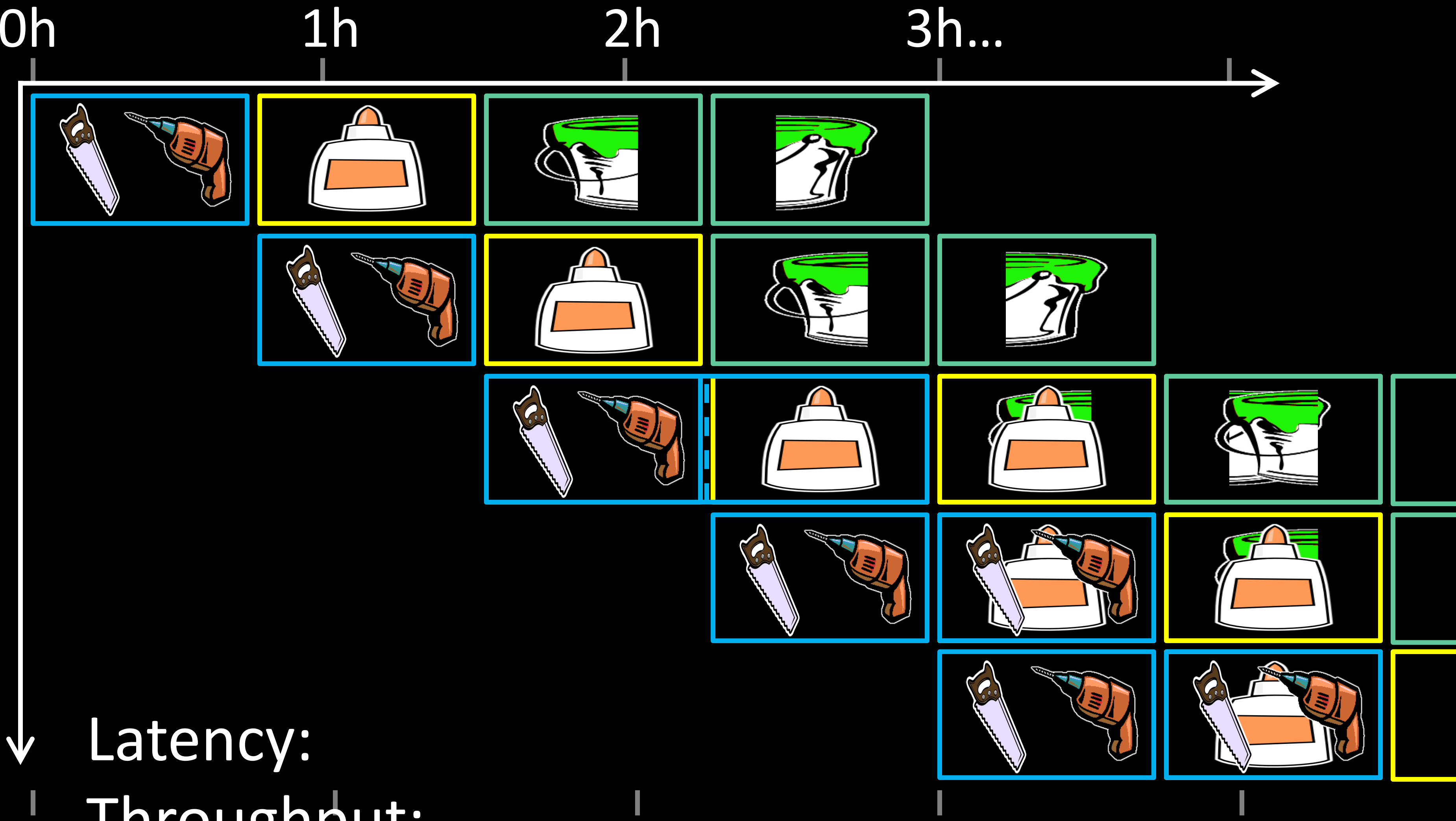
Latency: 4hrs /task

Throughput: 1 task/hr

Concurrency: 4)

Pipeline Hazards

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



Latency:

Throughput:

Concurrency:

Lessons

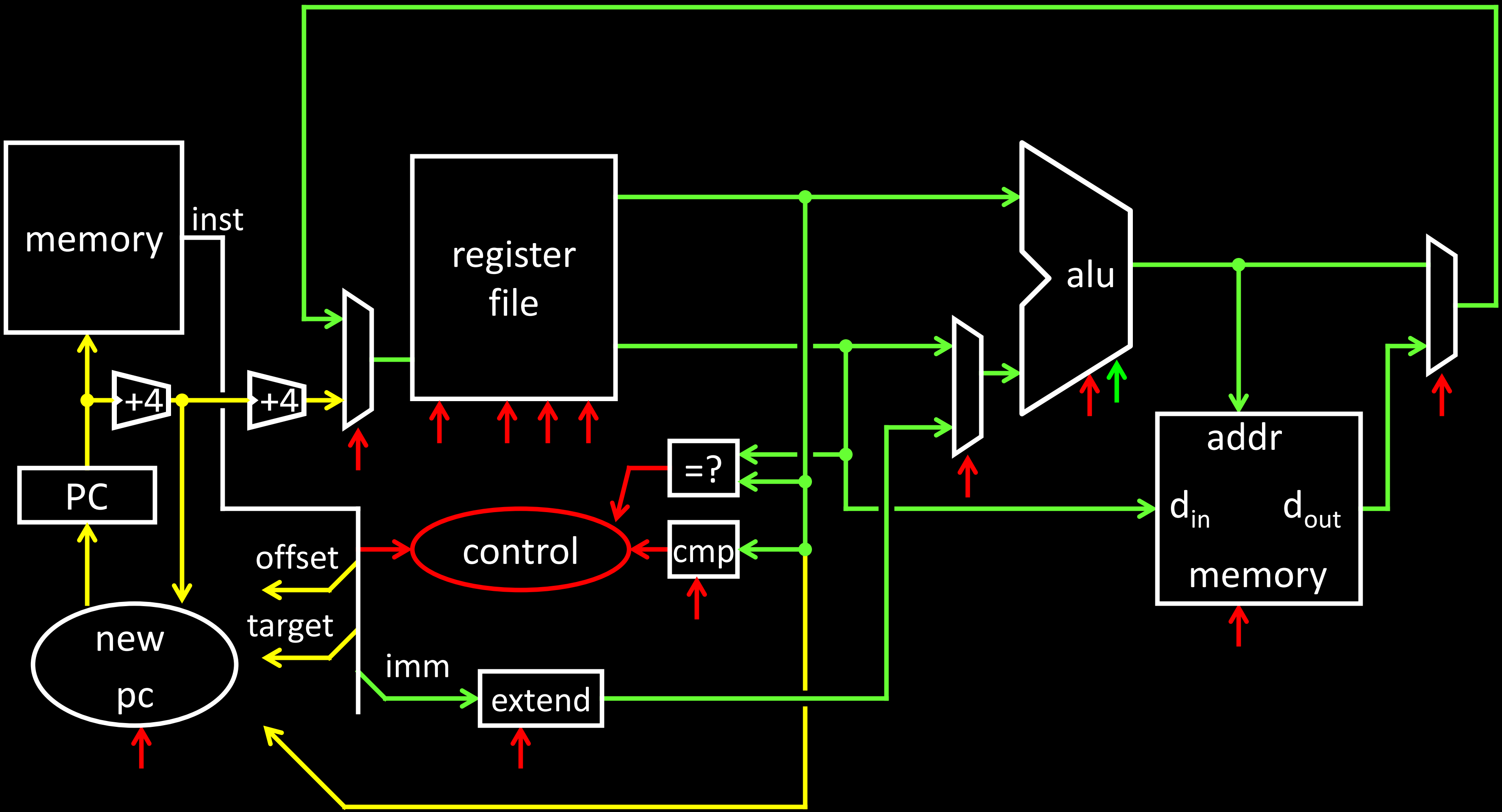
Principle:

Throughput increased by parallel execution

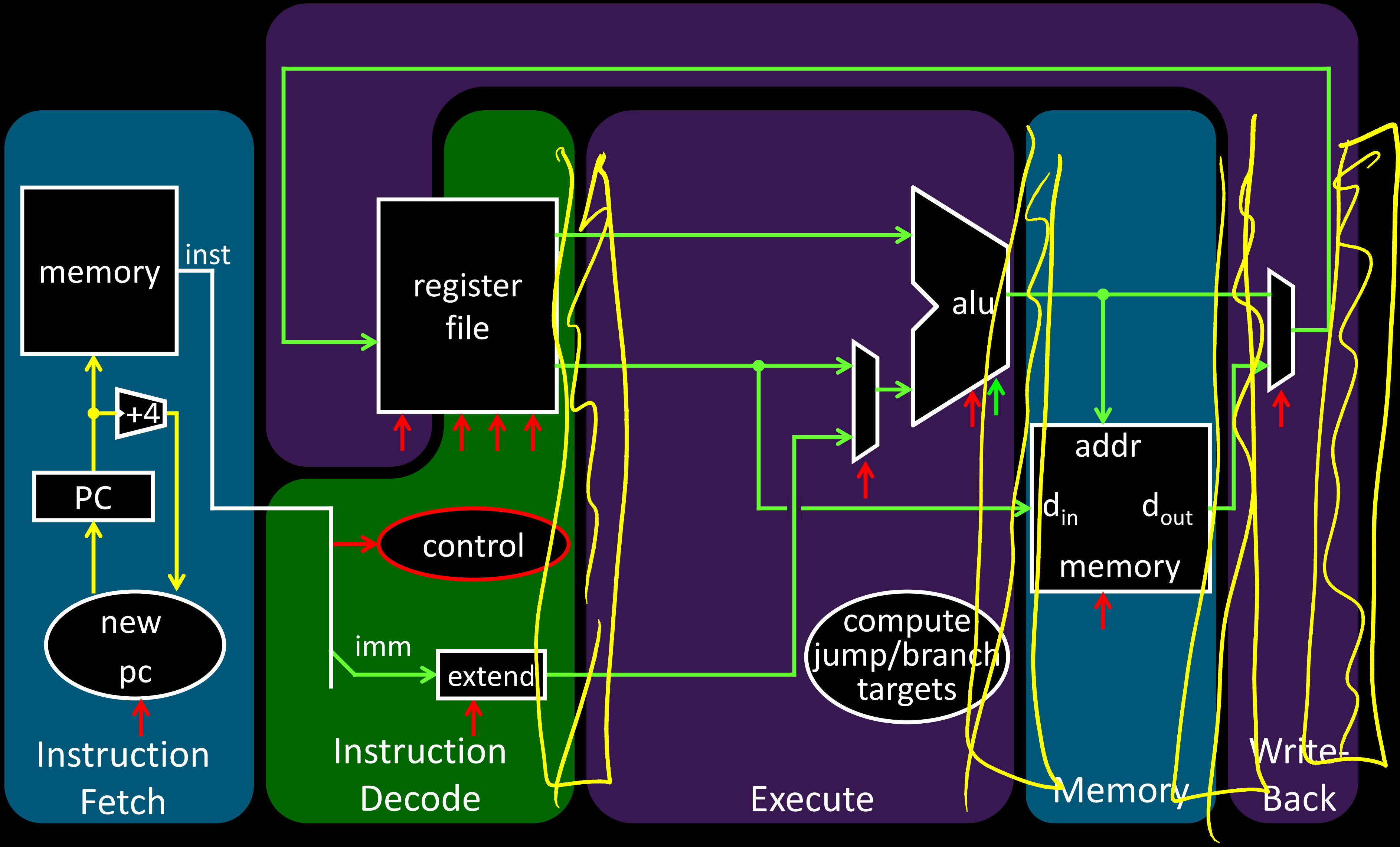
Pipelining:

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

A Processor


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

A Processor



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