CS3410

Guest Lecture

A Simple CPU: remaining branch instructions
CPU Performance
Pipelined CPU

Tudor Marian

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)
```

	0x00000000
	0x00000001
0x5	0x00000002
	0x00000003
	0x00000004
	0x00000005
	0x00000006
	0x00000007
0 | 5 ✓	0x00000008
0 | 0	0x00000009
0 | 0	0x0000000a
5 | 0	0x0000000b
	...
	0xffffffff

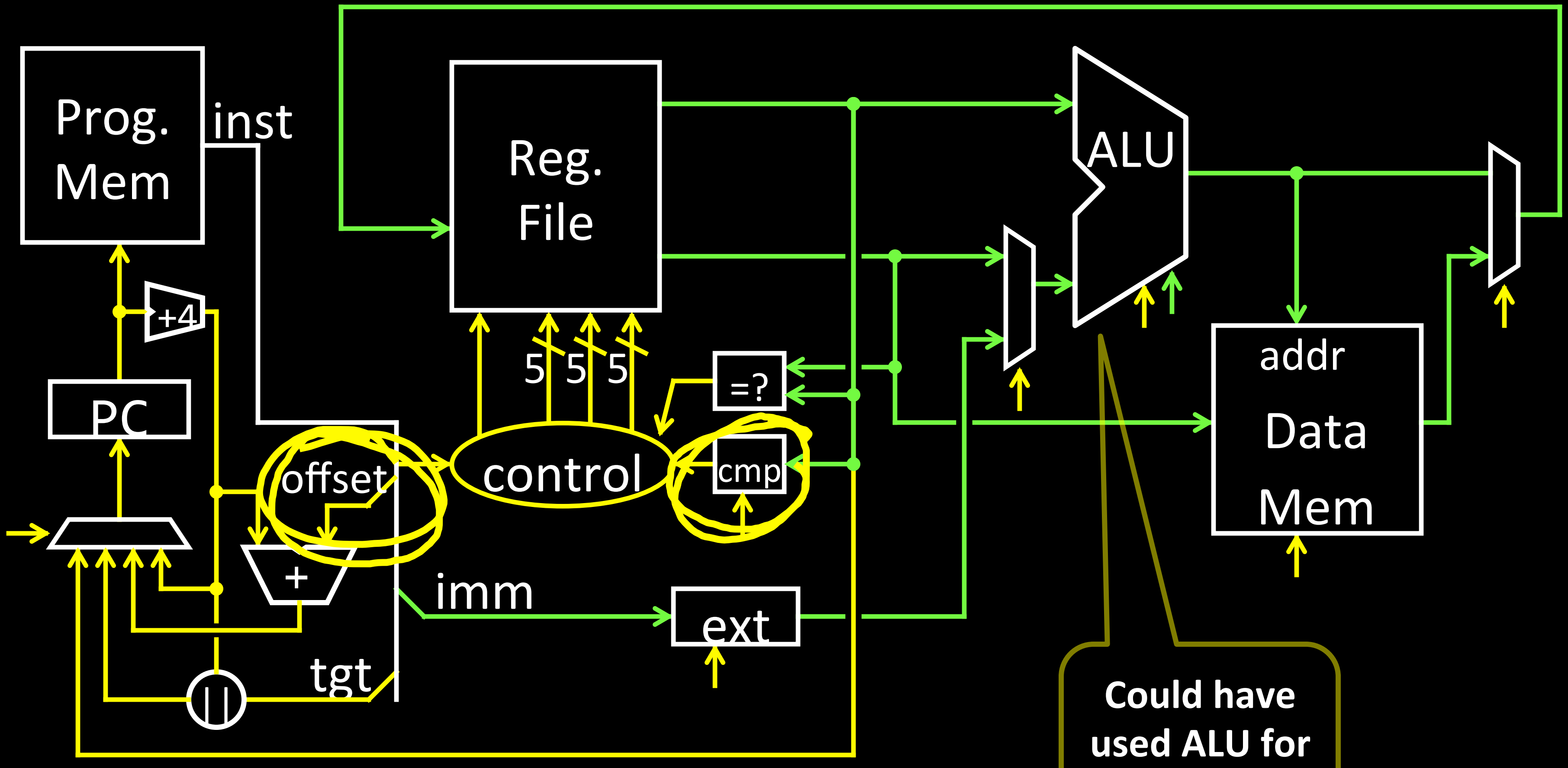
Conditional Jumps (cont.)

000001	00101	00001	0000000000000010
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)



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)

Function/procedure calls

```
00001100000001001000011000000010
```

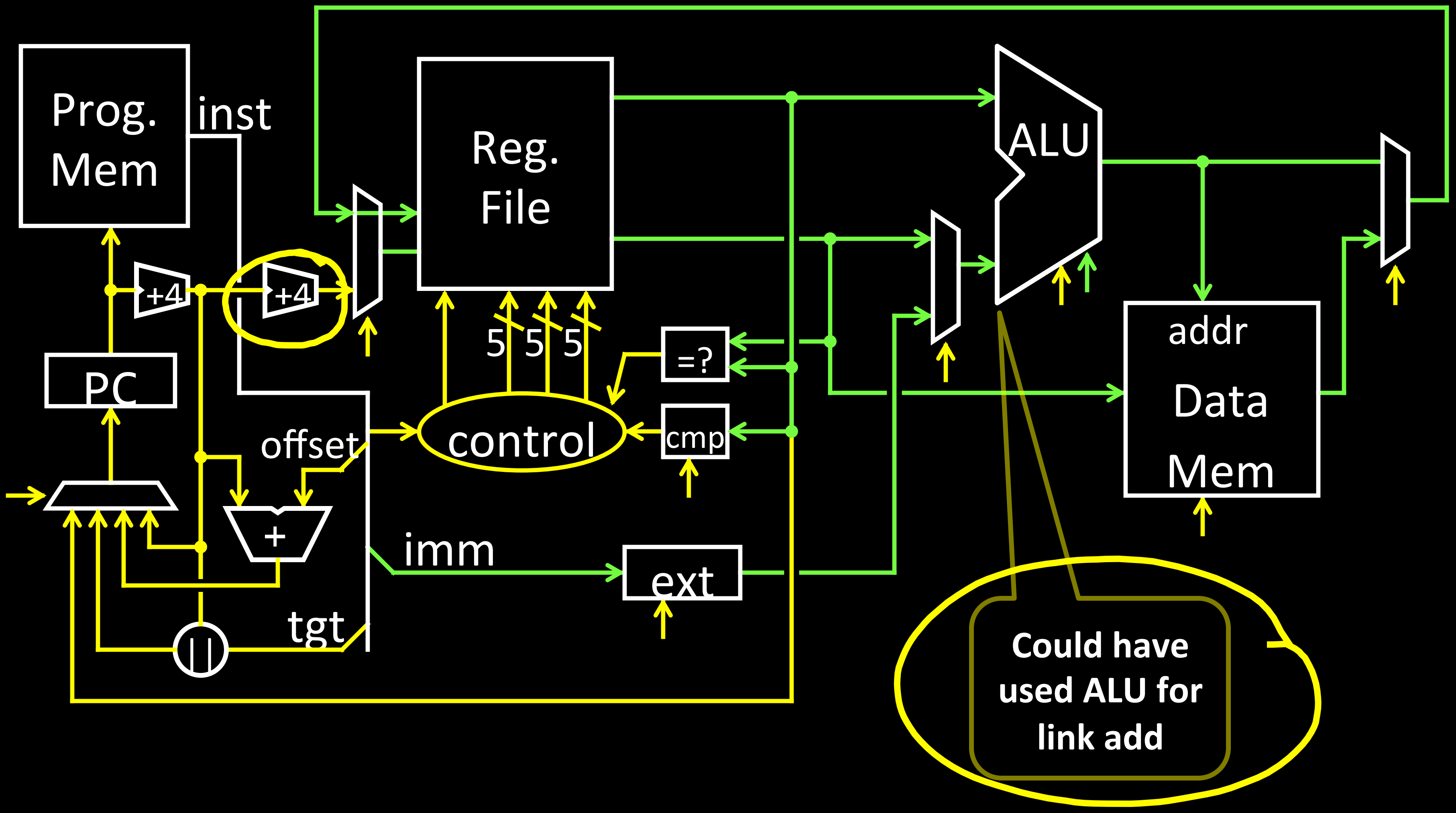
op — 6 bits

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)



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 to look for in a computer system?

- Correctness: negotiable?
- 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, …

How to measure performance?

GHz ; $\frac{1}{\tau}$; 1GHz → 1cycle 1ns

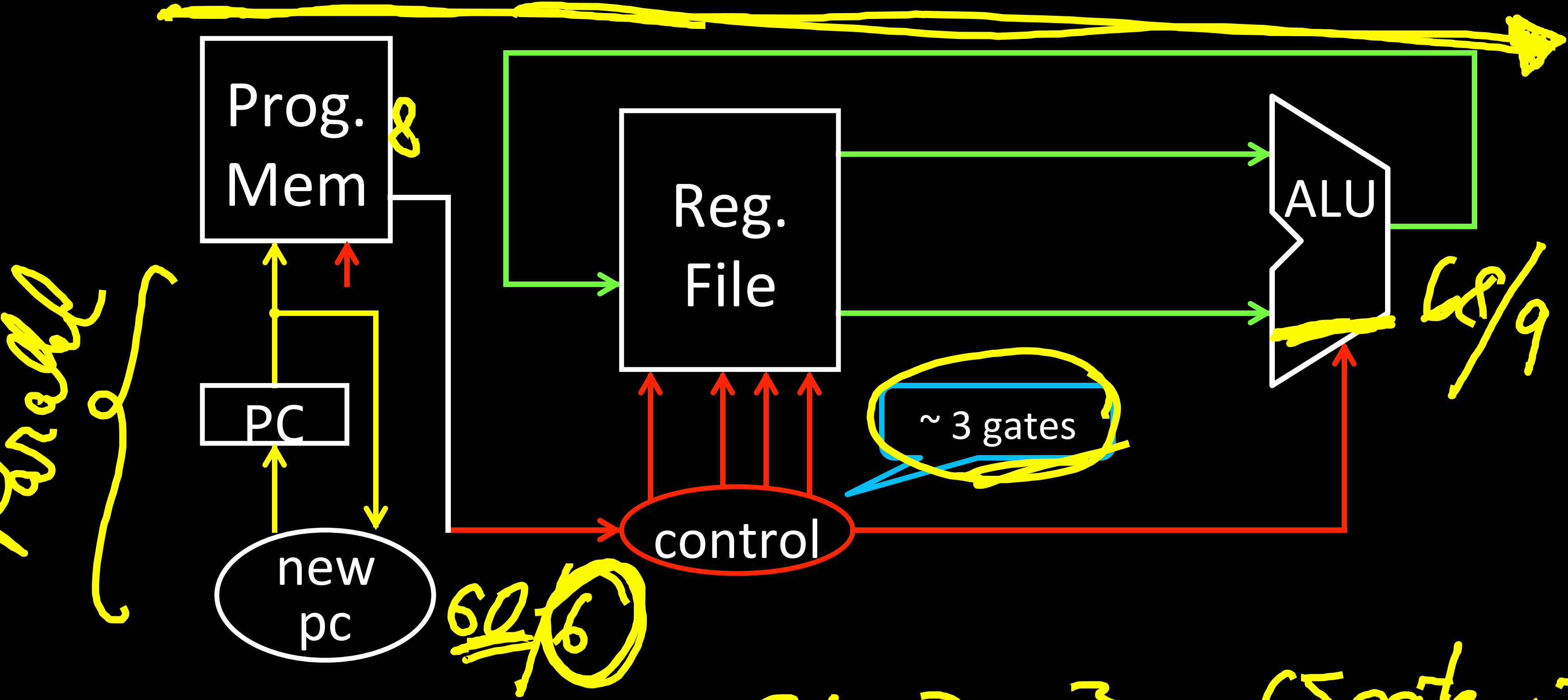
MIPS → 10^6 instr/sec

MFLOPS

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?



1 / 160 -

$31 \times 2 + 3 = 65$ gates $+ 3 \sim$

$(8 + 3 + 1 + 68) = 80$ gates

2 ns/

$8 + 3 + 9 + 1 = 21$ ga

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 — 1 gate
- 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

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

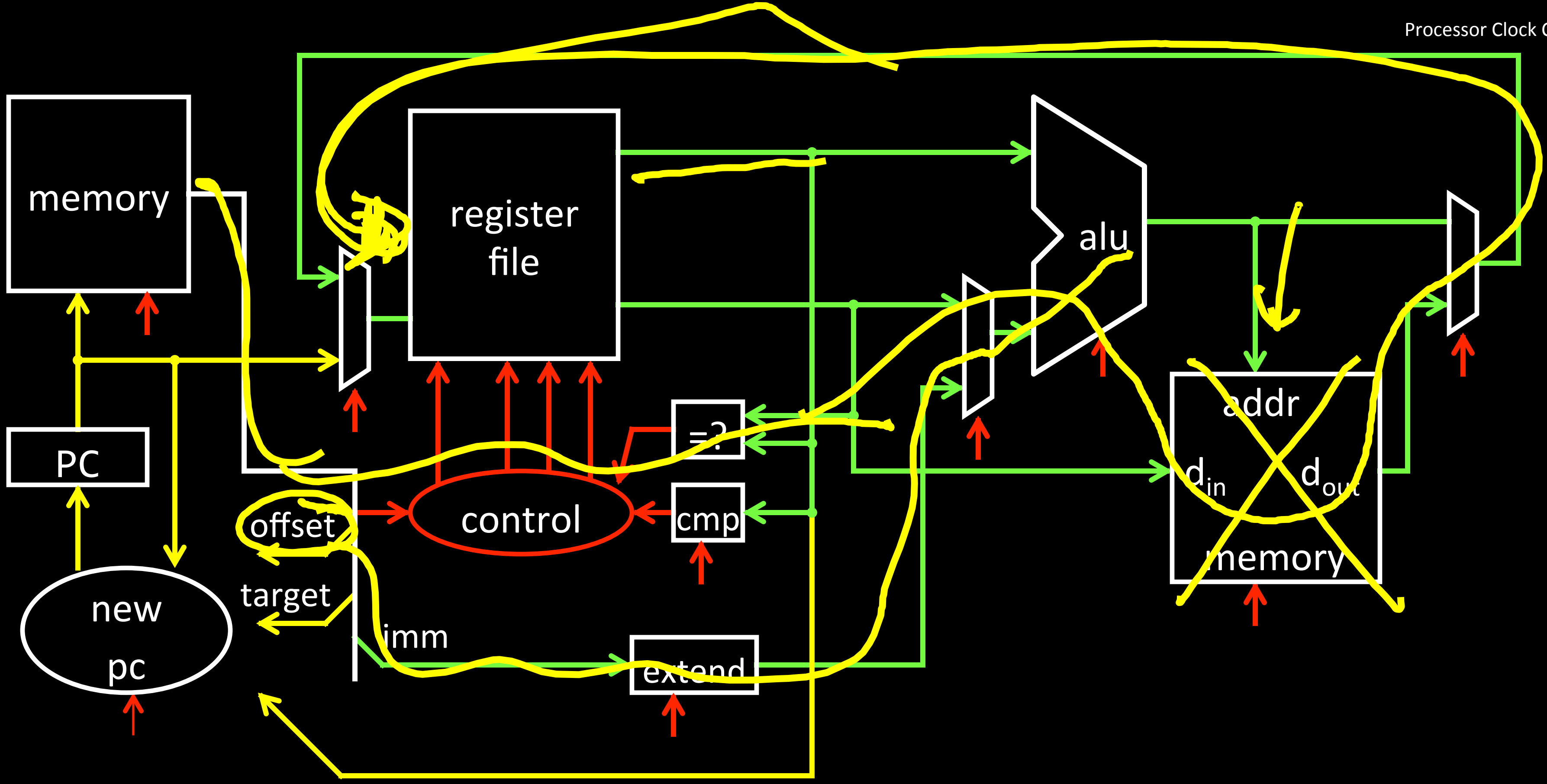
Critical Path

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

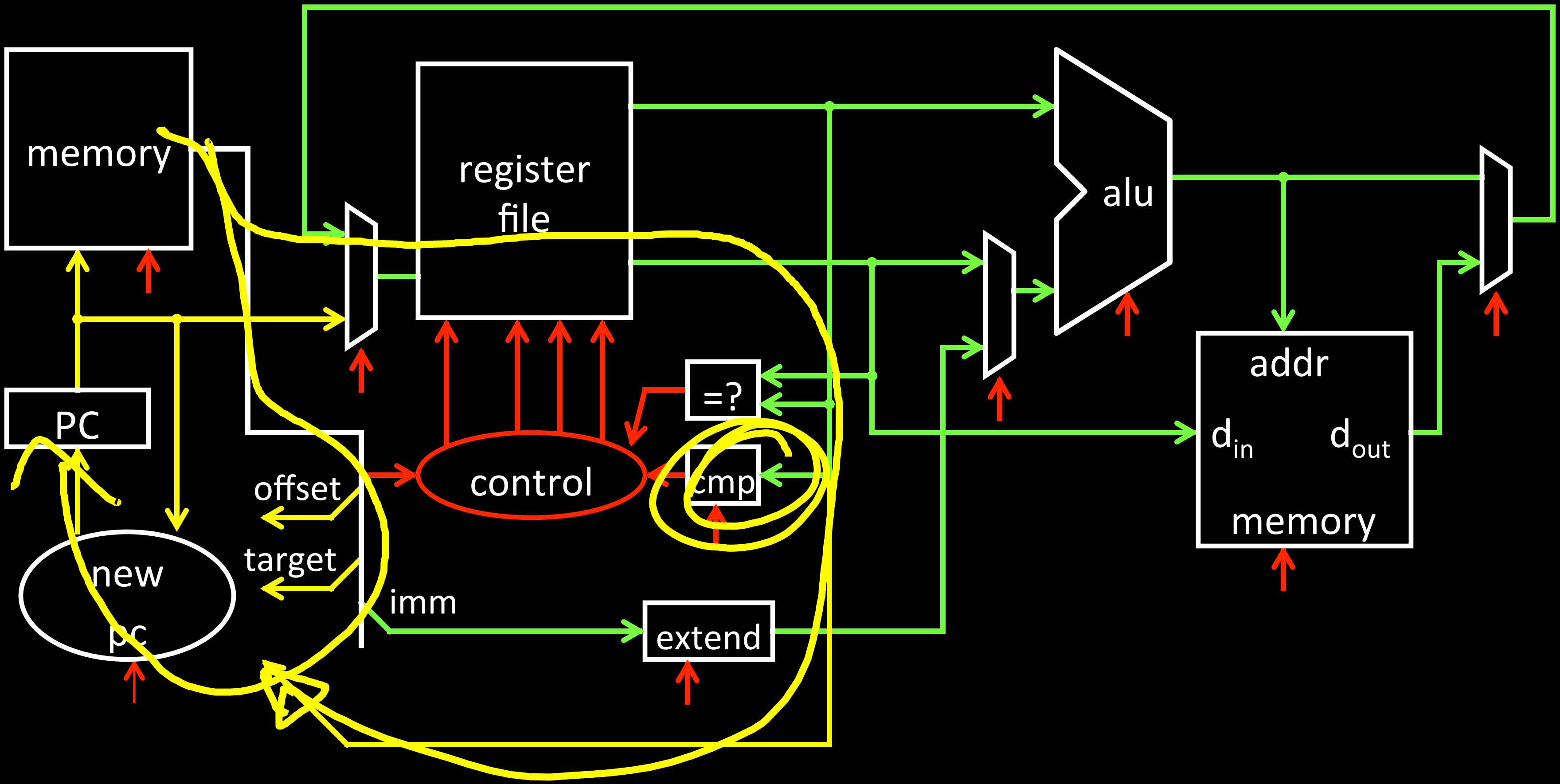
Strategy 1 (we just employed)

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

Single 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]



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

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

Strategy 2

- Multiple cycles to complete a single instruction

E.g: Assume:

- load/store: 100 ns → @ $\frac{1}{100ns}$ = 10 MHz
- arithmetic: 50 ns → @ $\frac{1}{50ns}$ = 20MHz
- branches: 33 ns → @ 30 MHz

66ns
99ns

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

10 MHz = 10 Mi

1 CPI

Instruction mix for some program P, assume:

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

$CPI = 0.25 \times 3 + \ldots$

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

Single-Cycle @ 15 MHz

800 MHz PIII "faster" than 1 GHz P4

$2.1 \text{ CPI} \Rightarrow \frac{30 \cdot 10^6 \text{ cy}}{2.1 \text{ cycles}} \approx 15 \text{ MIPS}$

→ 10 MIPS

→ 15 MIPS

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

0.75
1.2
0.15

CPi . 2.1

15MIPS

5, ns → 1/5 →

0.75
0.6
0.5

1.50

0.75
0.15
0.15

1.0

$\frac{30\text{ MHz}}{1.5\text{CPi}} = 20\text{MIPS}$

.) 1 cycle ...

30x4 MHz

$\frac{30\text{ MHz}}{1.05} = 30$

Amdahl’s Law

lw.

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

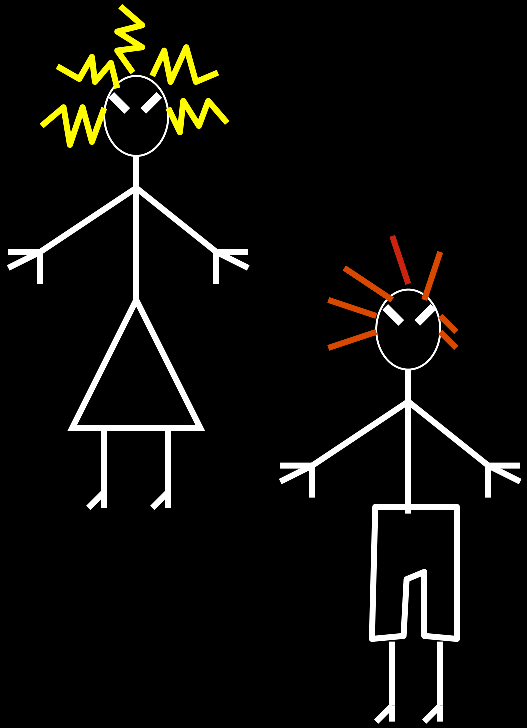
Pipelining

See: P&H Chapter 4.5

Alice

Bob

They don’t always get along…



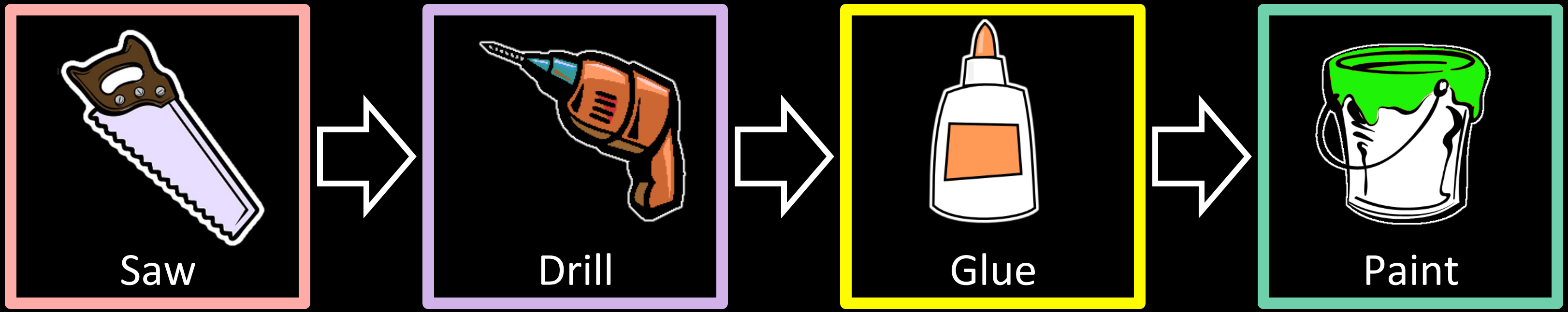
Saw

Drill

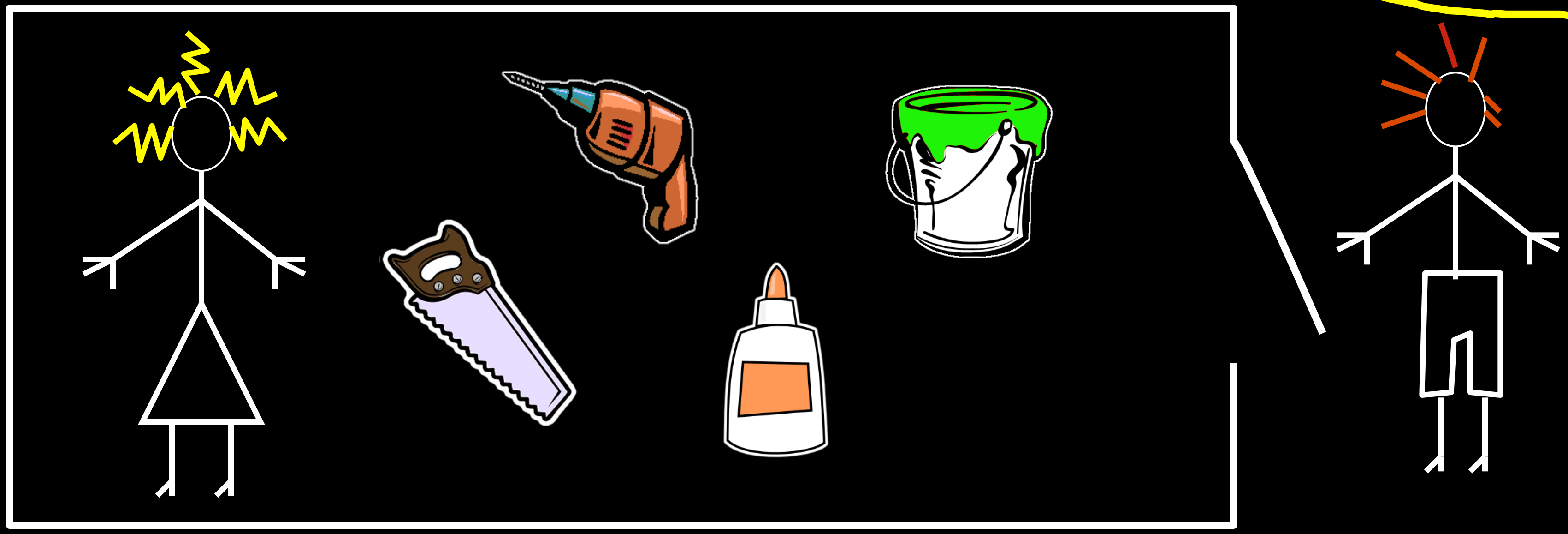
Glue

Paint

N pieces, each built following same sequence:



Design 1: Sequential Schedule



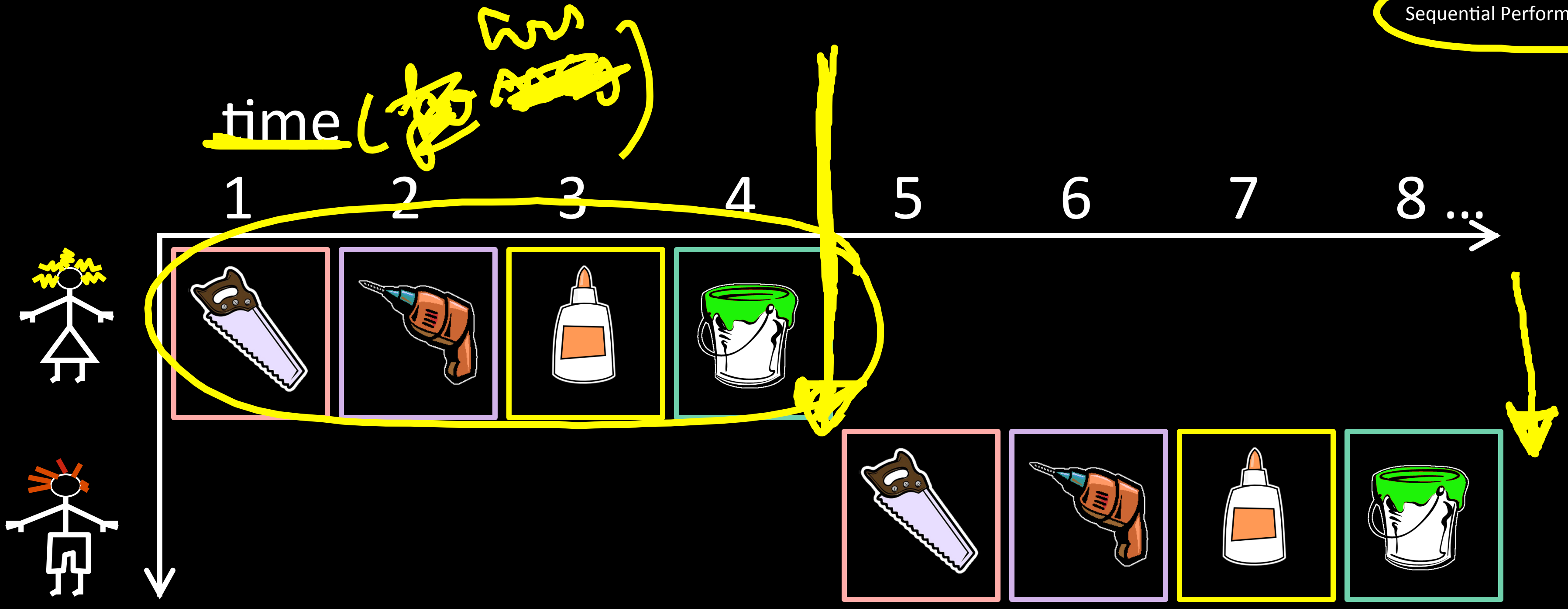
Alice owns the room

Bob can enter when Alice is finished

Repeat for remaining tasks

No possibility for conflicts

Sequential Performance



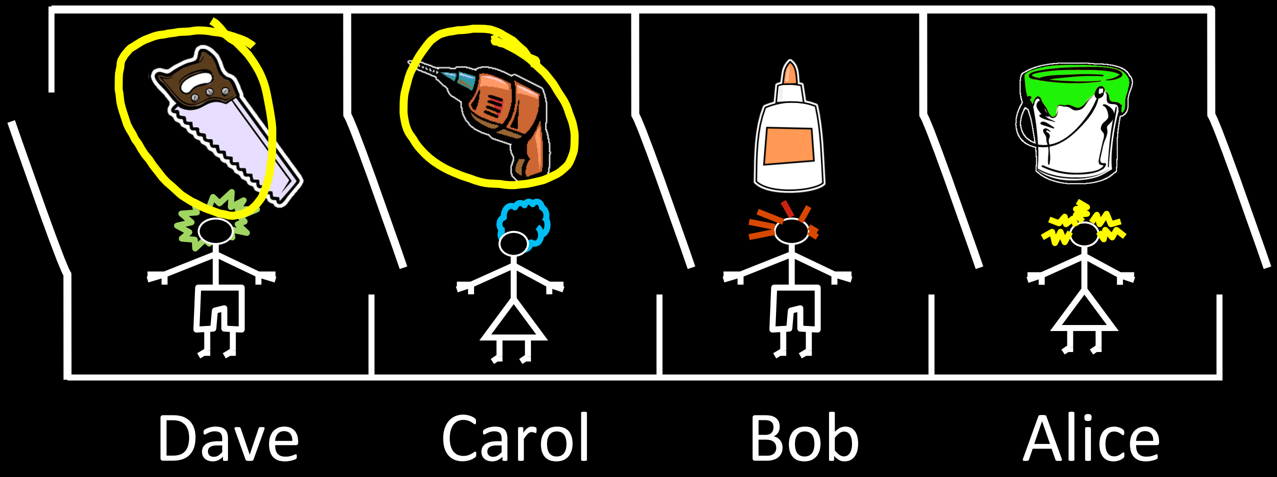
Latency: 4 hrs / task

Throughput: 1 task / 4 hrs

Concurrency: 1

Can we do better?

Partition room into *stages* of a *pipeline*



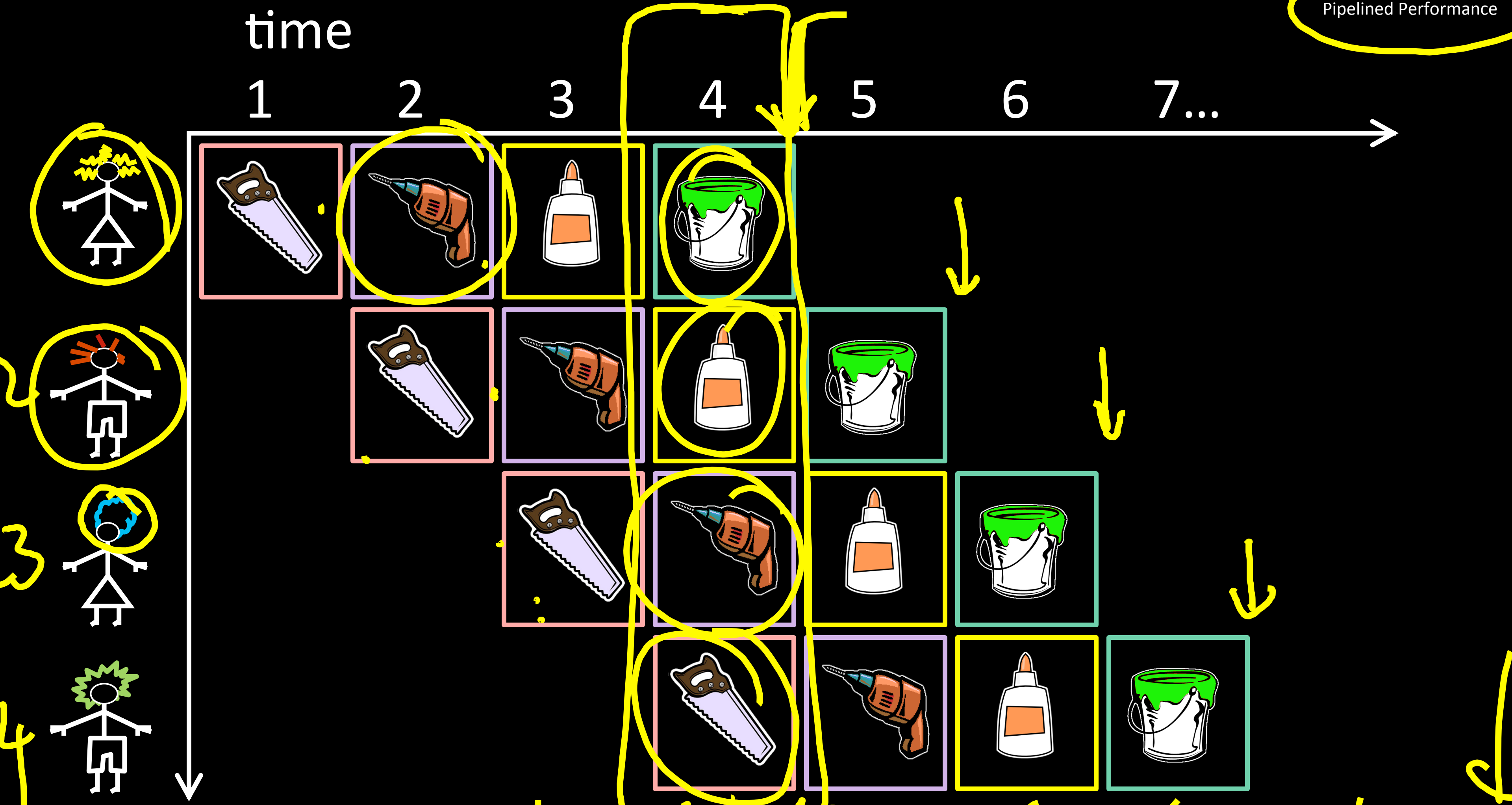
One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep

Pipelined Performance

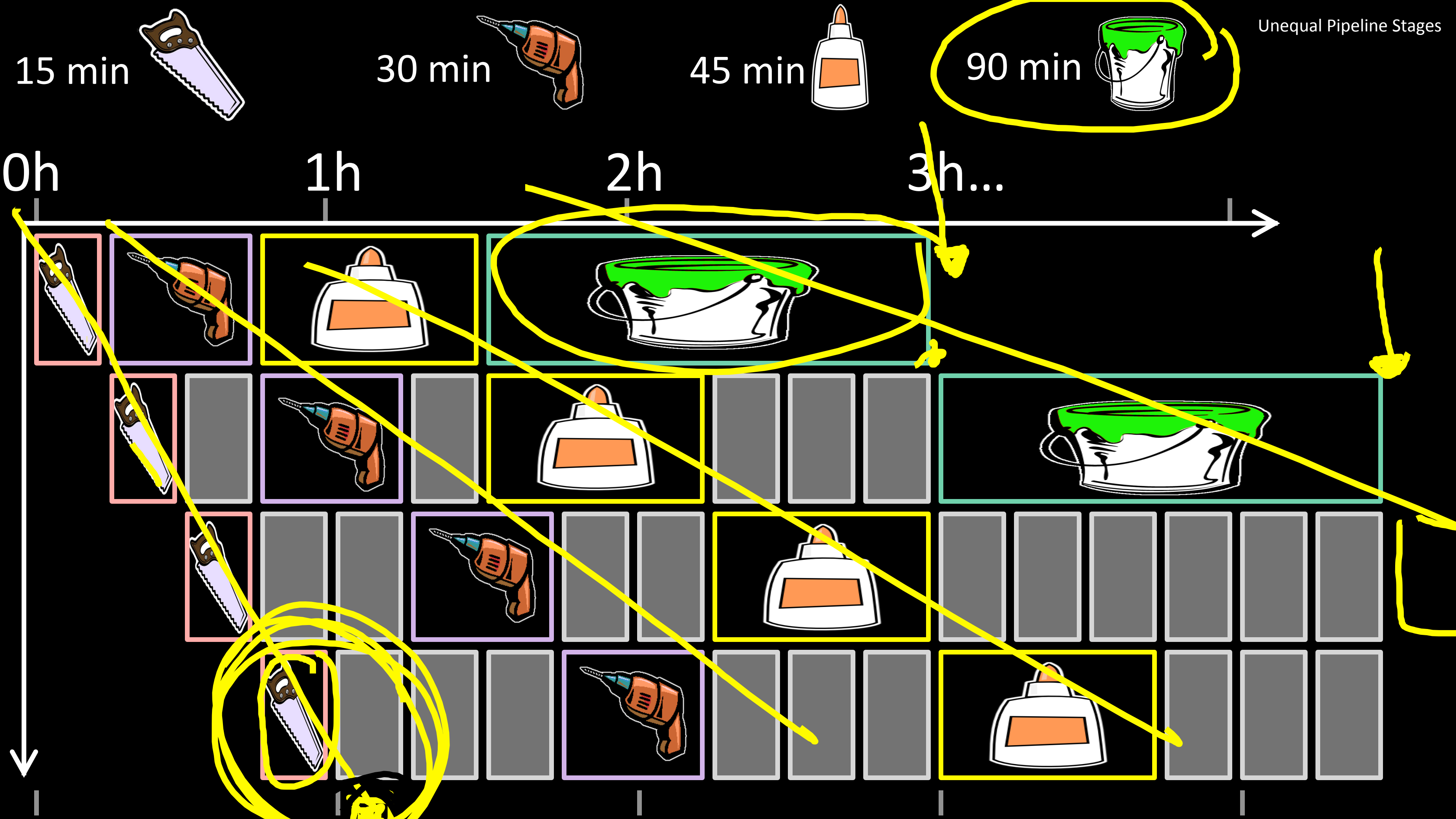


Latency: 4 hrs/task

Throughput: 1 task/hrs

Concurrency: 4

Unequal Pipeline Stages

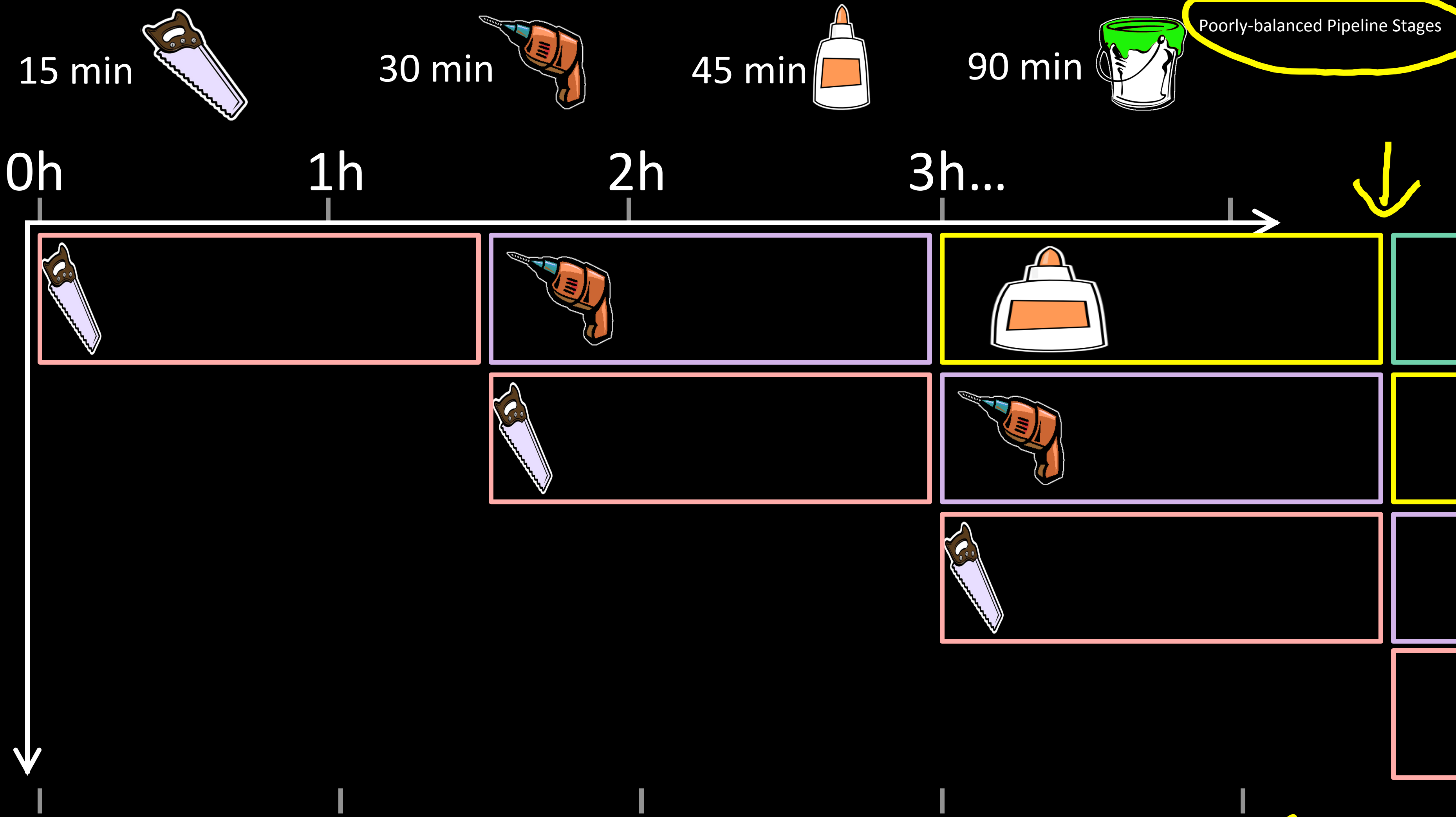


Latency: ∞

Throughput: 1 task / 90 min

Concurrency: $\boxed{\infty}$

Poorly-balanced Pipeline Stages

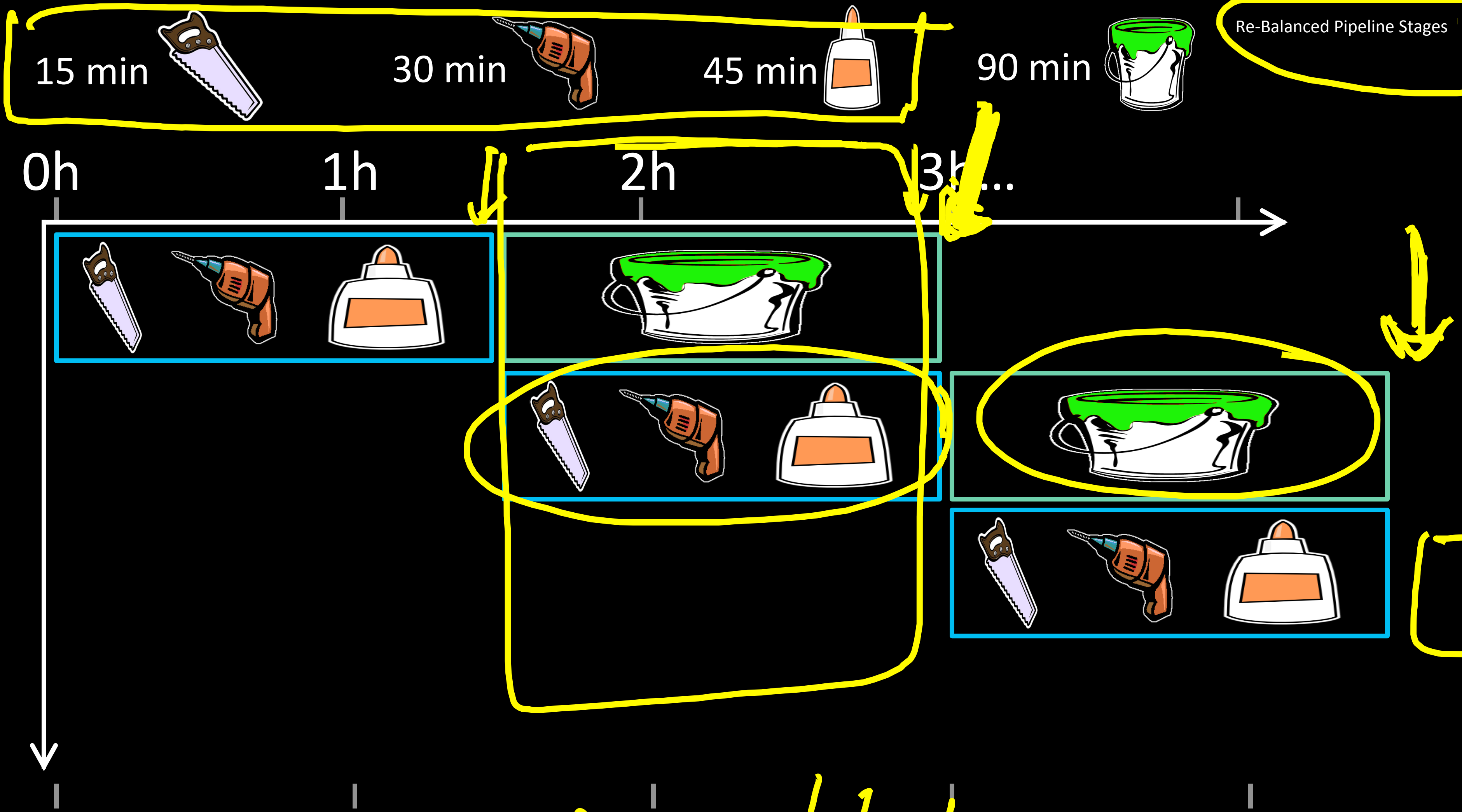


Latency: 4×90 min = 6 hrs / task

Throughput: 1 task / 90

Concurrency: 4

Re-Balanced Pipeline Stages

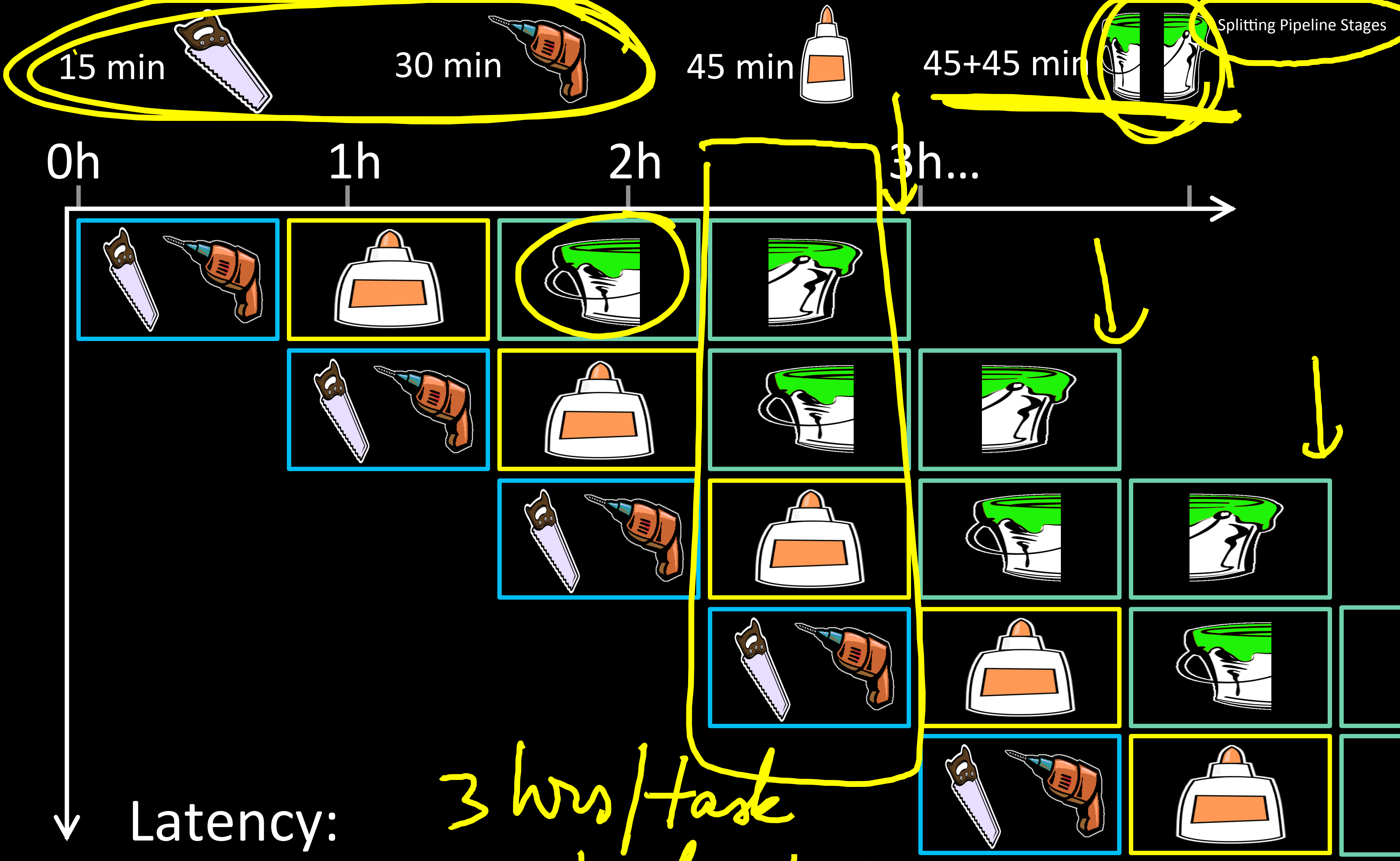


Latency: 3 hrs / task

Throughput: 1 task / 90 min

Concurrency: 2

Splitting Pipeline Stages

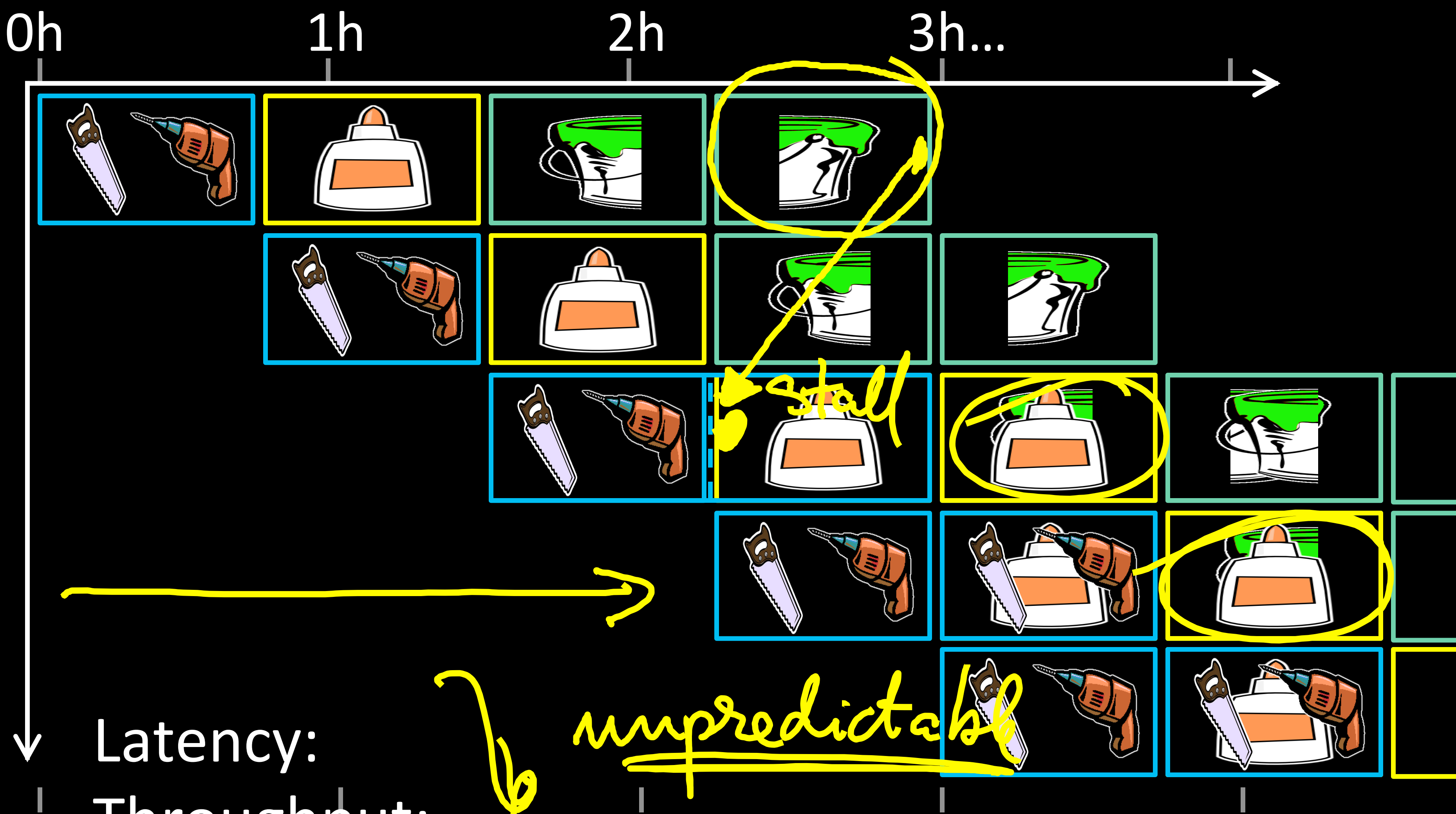


Latency: 3 hrs/task

Throughput: 1 task 45 min

Concurrency: 4

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



Latency:

Throughput:

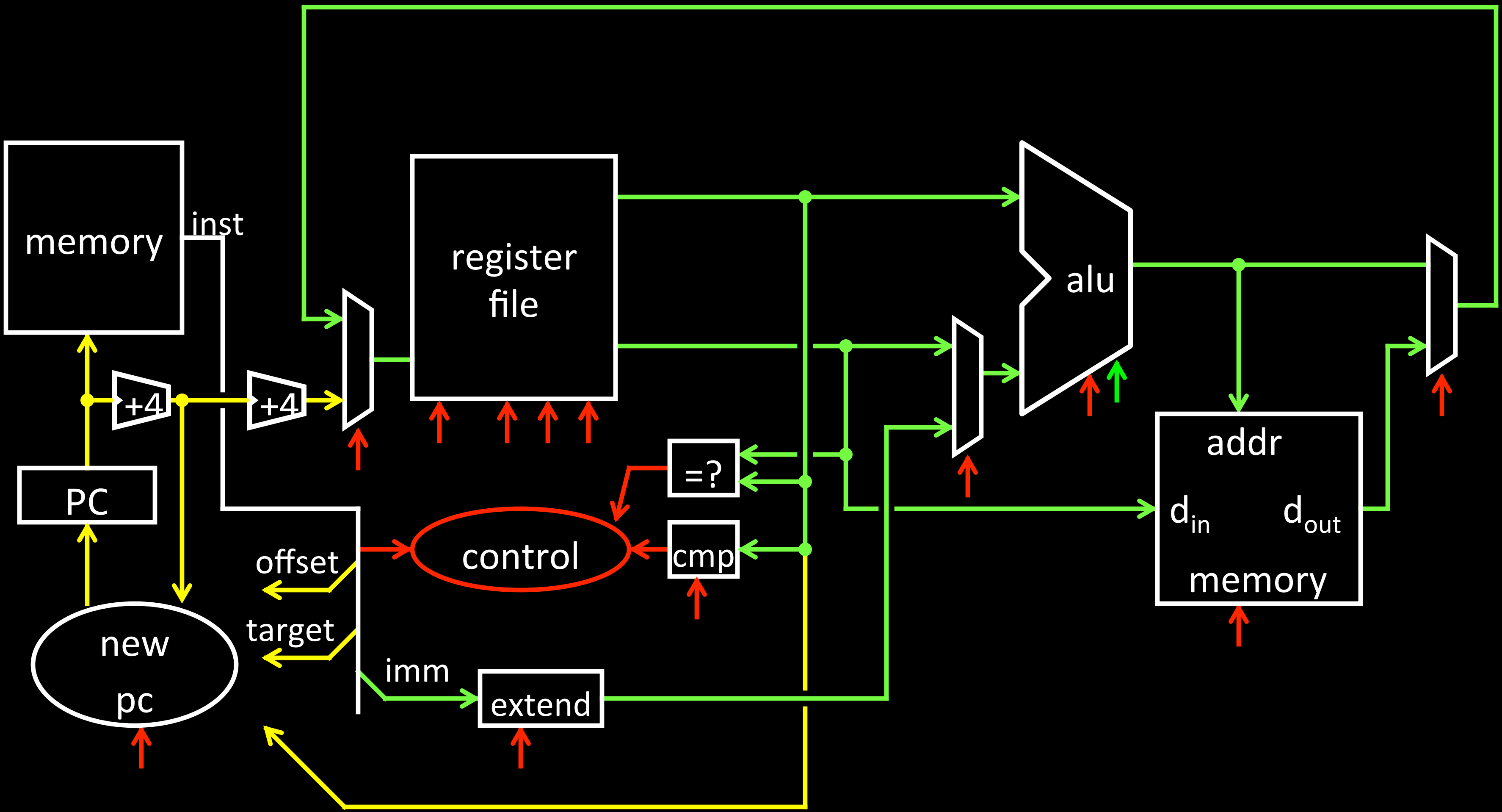
Concurrency:

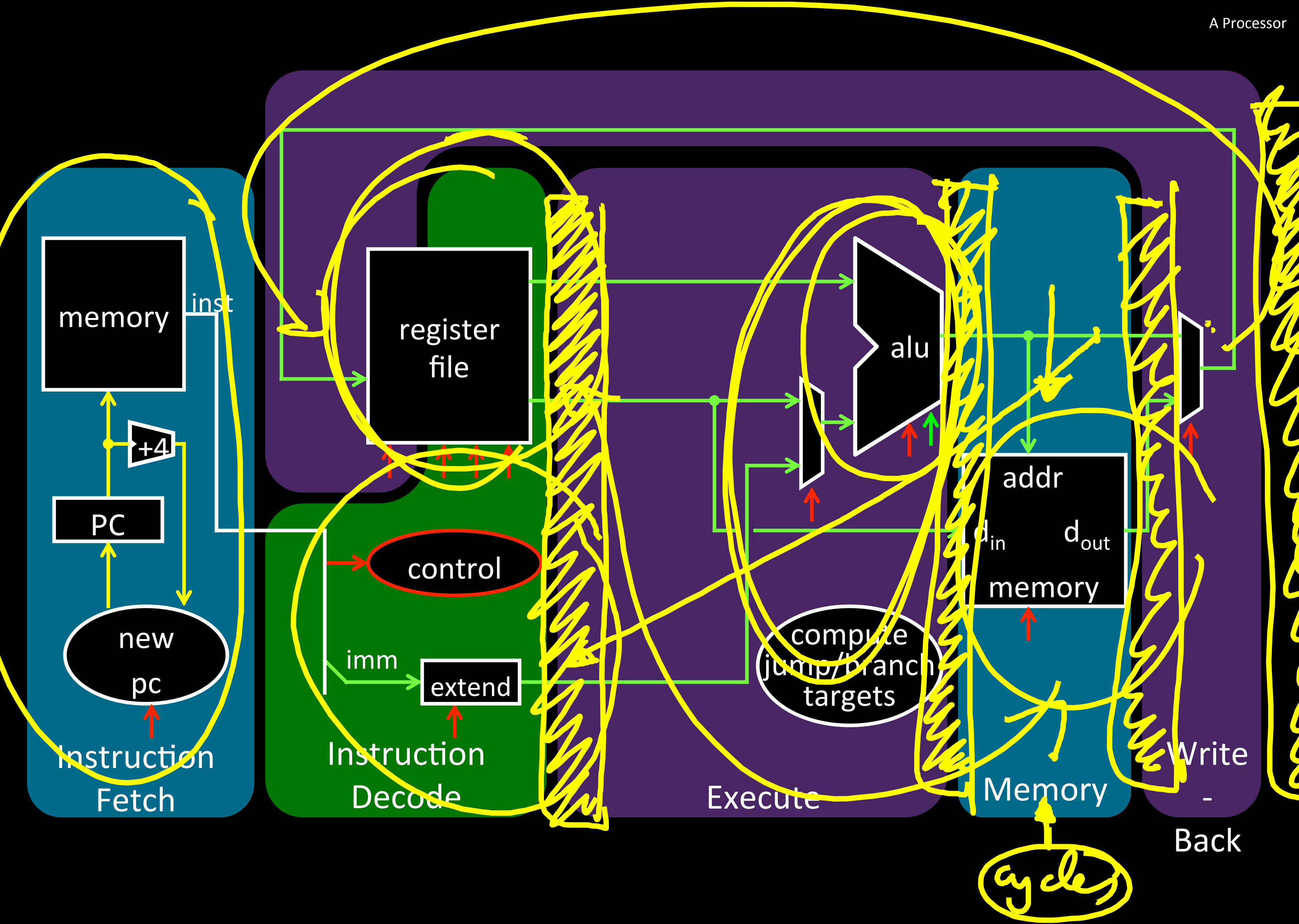
Principle:

Throughput increased by parallel execution

Pipelining:

- Identify *pipeline stages*
- Isolate stages from each other
- Resolve pipeline *hazards*


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


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

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 (1 gat
3. Execute (EX)
   - perform ALU operation, compute jump/branch targets
4. Memory (MEM)
   - access memory if needed
5. Writeback (WB)
   - update register file

5 instr / cycle

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

Slides thanks to Kevin Walsh, Sally McKee, and Kavita Bala