Computer Architecture

Week 7: Pipelining

Fenerbahçe University

Professor & TAs

Prof: Dr. Vecdi Emre Levent

Office: 311

Email: emre.levent@fbu.edu.tr

TA: Arş. Gör. Uğur Özbalkan

Office: 311

Email: ugur.ozbalkan@fbu.edu.tr

Course Plan

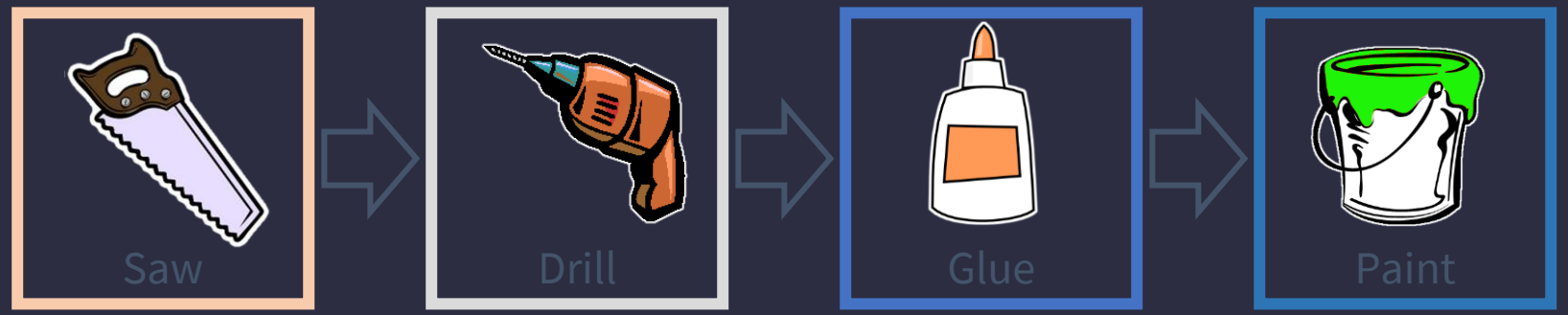
- Pipelining and Performance

Improving Performance

- Parallelism
- Pipelining
- Both!

Pipelining Example: The Instructions

N pieces, each built following same sequence:



Design 1: Sequential Schedule

Alice owns the room

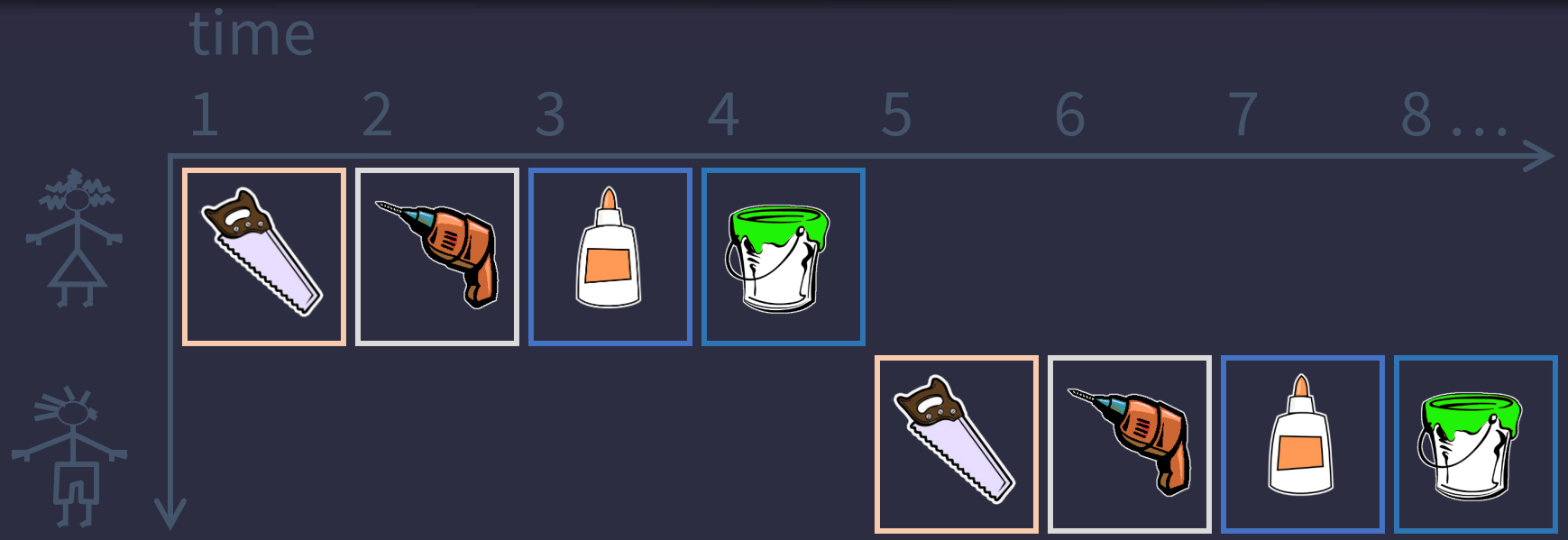
Bob can enter when Alice is finished

Repeat for remaining tasks

No possibility for conflicts

Sequential Performance

- Elapsed Time for Alice: 4
- Elapsed Time for Bob: 4
- Total elapsed time: 4*N
- Can we do better?



Latency:	4 hours/task
Throughput:	1 task/4 hrs
Concurrency:	1

CPI = 4

Design 2: Pipelined Design

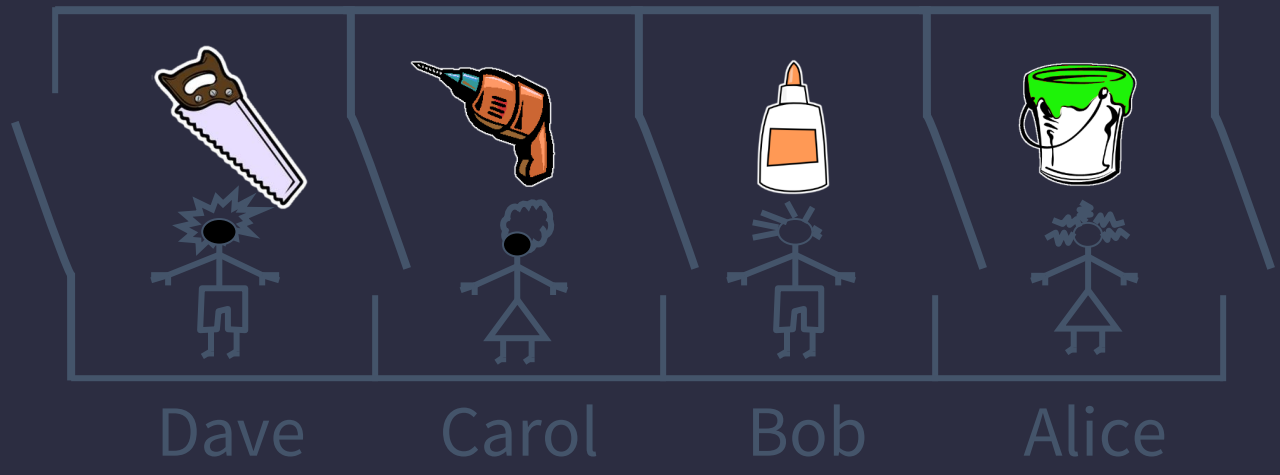
Partition room into *stages* of a *pipeline*

One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep



Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*

One person owns a stage at a time

4 stages

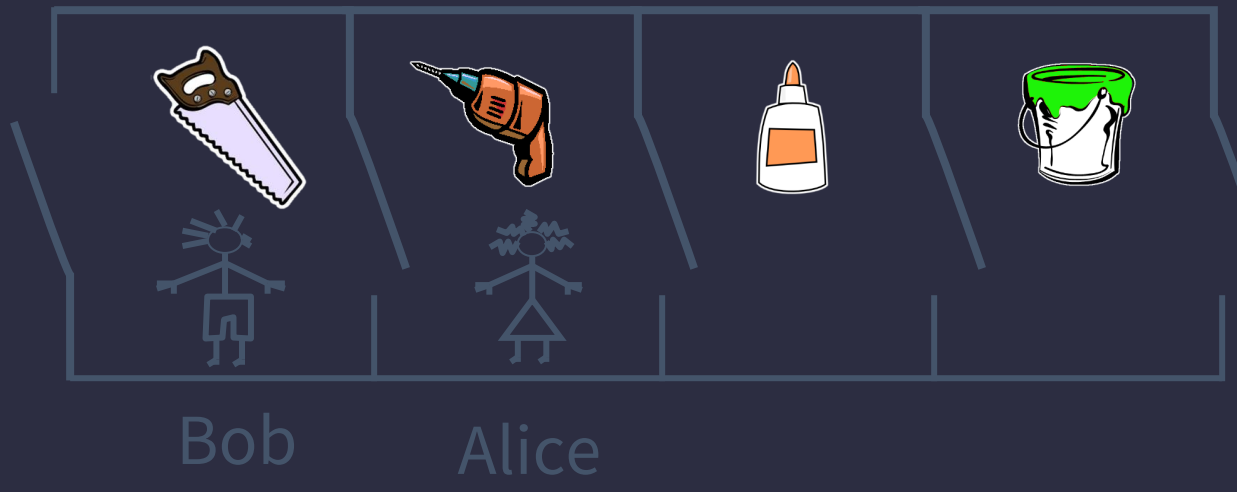
4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete

Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*



One person owns a stage at a time

4 stages

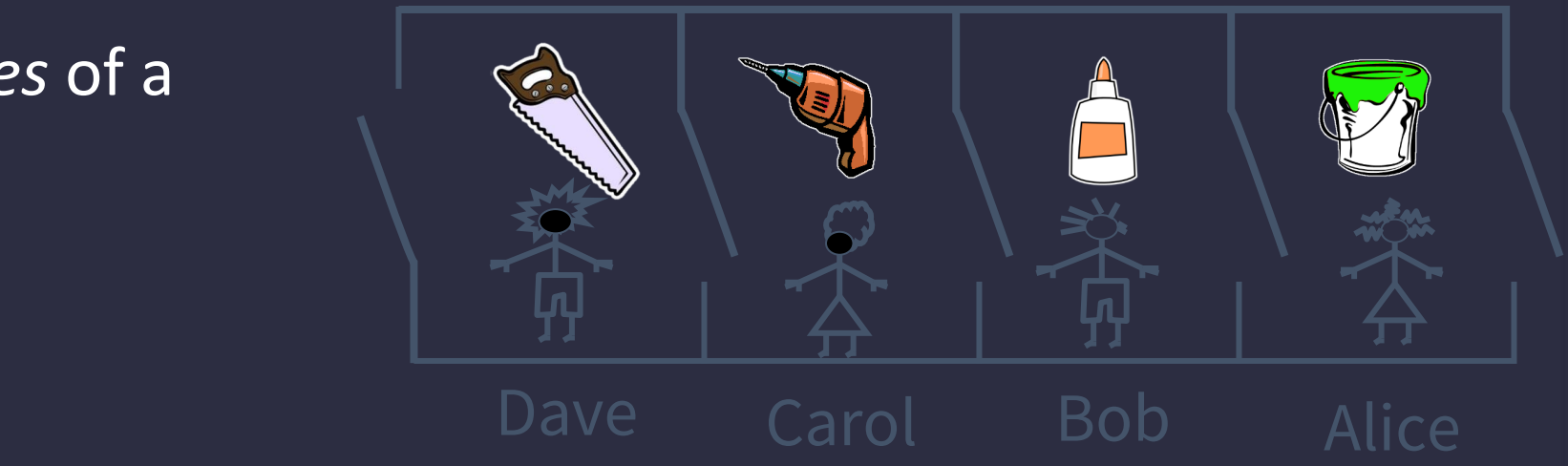
4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete

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

It still takes all four stages for one job to complete

Pipelined Performance

Latency: 4 hrs/task

Throughput: 1 task/hr

Concurrency: 4

CPI = 1

Pipelined Performance

What if drilling takes twice as long, but gluing and paint take ½ as long?

Latency: 4 cycles/task

Throughput: 1 task/2 cycles

CPI = 2

Lessons

- Principle:
  - Throughput increased by parallel execution
  - Balanced pipeline very important
  - Else slowest stage dominates performance

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

Goals for today

Performance

- What is performance?
- How to get it?

Complex question

- How fast is the processor?
- How fast your application runs?
- How quickly does it respond to you?
- How fast can you process a big batch of jobs?
- How much power does your machine use?

Measures of Performance

Clock speed

- 1 KHz, 10^3 Hz: cycle is 1 millisecond, ms, (10^{-6})
- 1 MHz, 10^6 Hz: cycle is 1 microsecond, us, (10^{-6})
- 1 Ghz, 10^9 Hz: cycle is 1 nanosecond, ns, (10^{-9})
- 1 Thz, 10^{12} Hz: cycle is 1 picosecond, ps, (10^{-12})

Instruction/application performance

- MIPs (Millions of instructions per second)
- FLOPs (Floating point instructions per second)
    - GPUs: GeForce GTX Titan (2,688 cores, 4.5 Tera flops, 7.1 billion transistors, 42 Gigapixel/sec fill rate, 288 GB/sec)

Measures of Performance

CPI: "Cycles per instruction"→ Cycle/instruction for **on average**

- **IPC** = 1/CPI
  - Used more frequently than CPI
  - Favored because "bigger is better"
- Different instructions have different cycle costs
  - E.g., "add" typically takes 1 cycle, "divide" takes >10 cycles
- Depends on relative instruction frequencies

Measures of Performance

CPI example

- Program has equal ratio: integer, memory, floating point
- Cycles per inst type: integer = 1, memory = 2, FP = 3
- What is the CPI? (33% * 1) + (33% * 2) + (33% * 3) = 2

Measures of Performance

General public (mostly) ignores CPI

- Equates clock frequency with performance!

Which processor would you buy?

- Processor A: CPI = 2, clock = 5 GHz
- Processor B: CPI = 1, clock = 3 GHz
  - B is faster (assuming same ISA/compiler)

Measures of Performance

Latency

- How long to finish my program
  - Response time, elapsed time, wall clock time
  - CPU time: user and system time

Throughput

- How much work finished per unit time

Ideal: Want high throughput, low latency

… also, low power, cheap ($$) etc.

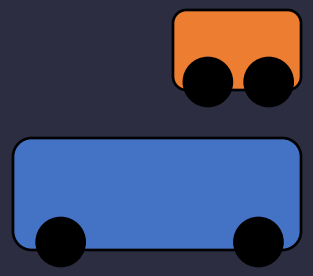
Example: Car vs. Bus

Car: speed = 60 km/hour, capacity = 5

Bus: speed = 20 km/hour, capacity = 60

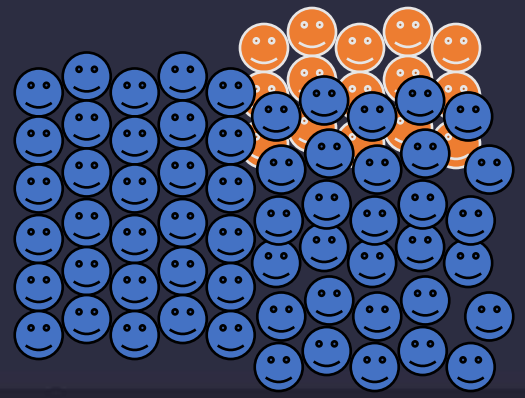
Task: transport passengers 10 km

	Latency (min)	Throughput (30 Min)
Car	10 min	15 PPH
Bus	30 min	60 PPH



#1 Car Throughput

#2 Bus Throughput



How to make the computer faster?

- Decrease latency
- Critical Path
  - Longest path determining the minimum time needed for an operation
  - Determines minimum length of clock cycle i. e. determines maximum clock frequency

- Optimize for latency on the critical path
  - Parallelism
  - Pipelining
  - Both

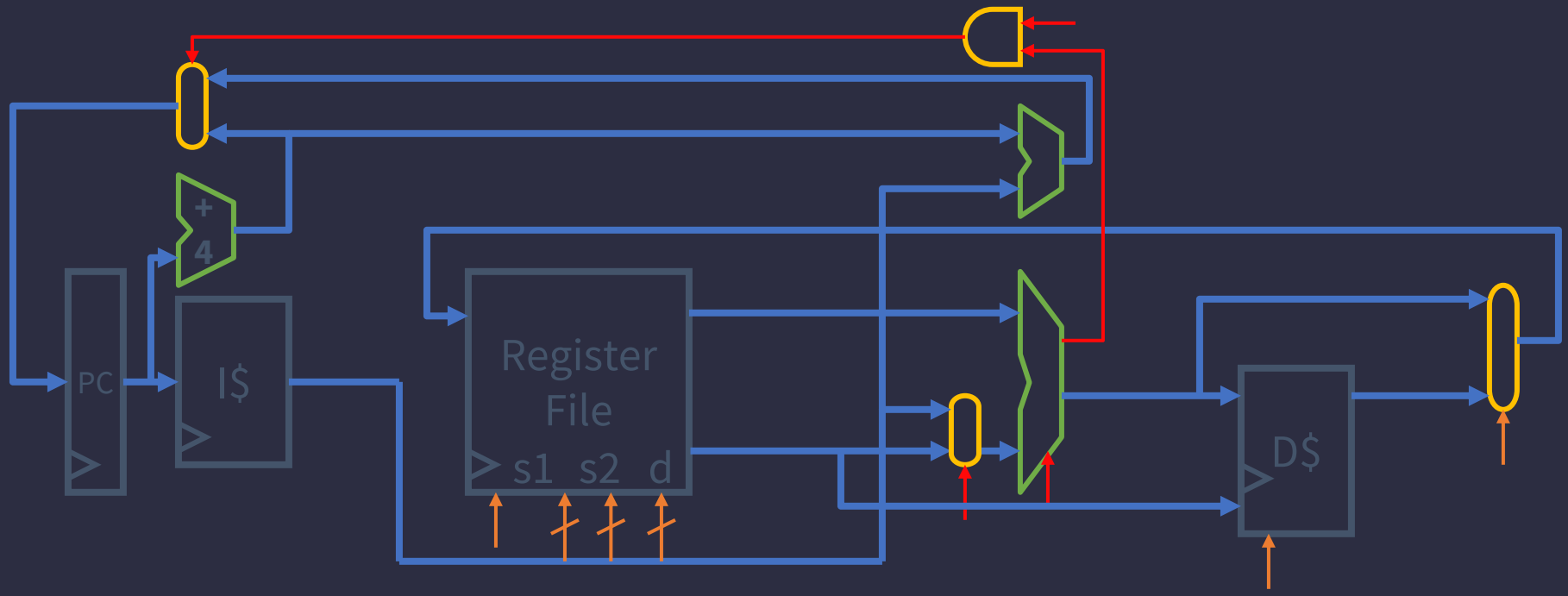
Review: Single-Cycle Datapath

Single-cycle datapath

- Fetch, decode, execute one instruction/cycle

+ Low CPI, 1 by definition

– Long clock period: accommodate slowest insn

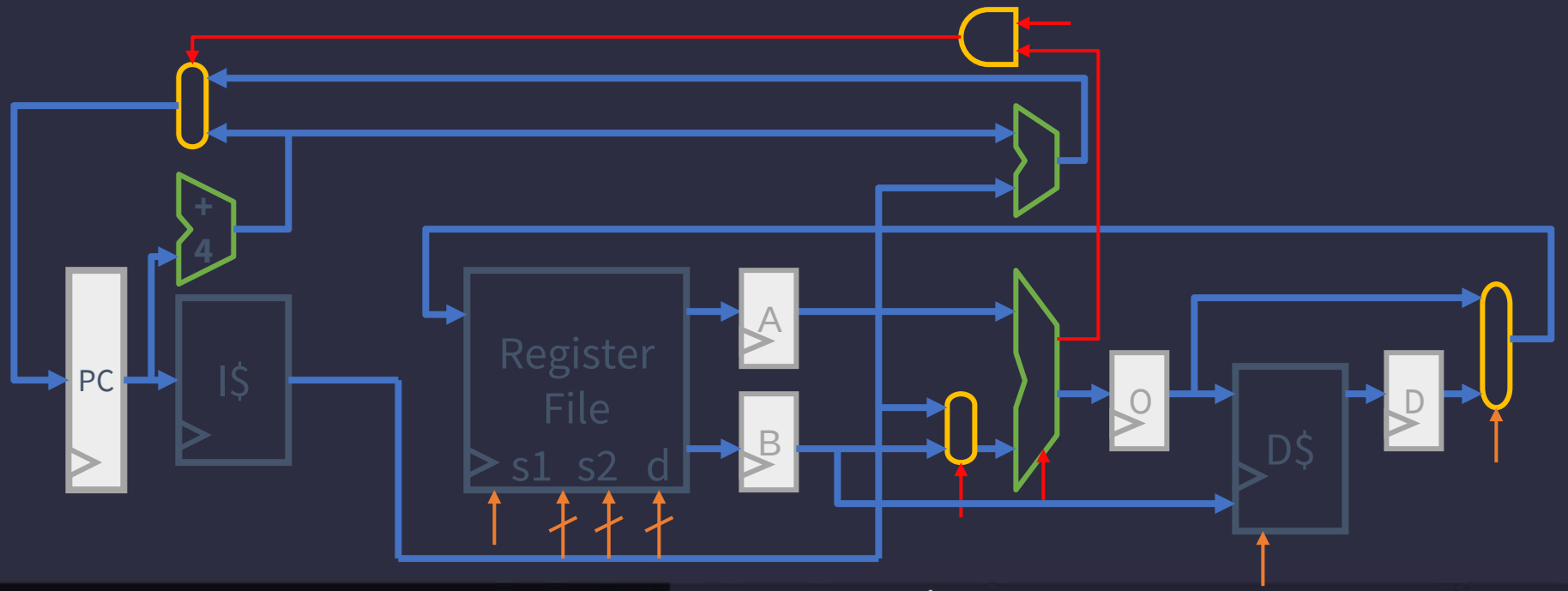


New: Multi-Cycle Datapath

Multi-cycle datapath: attacks slow clock

- Fetch, decode, execute one insn over multiple cycles
- **Allows insns to take different number of cycles**

± Opposite of single-cycle: short clock period, high CPI



Single vs. Multi-cycle Performance

Single-cycle

- Clock period = 50ns, CPI = 1
- Performance = **50ns/insn**

Multi-cycle: opposite performance split

+ Shorter clock period
– Higher CPI

Example

- branch: 20% (**3** cycles), ld: 20% (**5** cycles), ALU: 60% (**4** cycle)
- Clock period = **10ns**, CPI = (20%*3)+(20%*5)+(60%*4) = 4
- Performance = **40ns/insn**

Multi-Cycle Instructions

But what to do when operations take diff. times?

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
ps = 10^{-12} seconds

Single-Cycle CPU

10 MHz (100 ns cycle) with

– 1 cycle per instruction

Multi-Cycle Instructions

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
ps = 10^{-12} seconds

Single-Cycle CPU

10 MHz (100 ns cycle) with

– 1 cycle per instruction

Multi-Cycle CPU

30 MHz (33 ns cycle) with

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

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 ← 30M cycles/sec ÷2.1 cycles/instr ≈15 MIPS

vs

Single-Cycle @ 10 MHz ← 10 MIPS = 10M cycles/sec ÷ 1 cycle/instr

MIPS = millions of instructions per second

Total Time

CPU Time = # Instructions x CPI x Clock Cycle Time

Instructions per program: “dynamic instruction count”

- Runtime count of instructions executed by the program
- Determined by program, compiler, ISA

Cycles per instruction: “CPI”

- How many *cycles* does an instruction take to execute?
- Determined by program, compiler, ISA, micro-architecture

Total Time

Seconds per cycle: clock period, length of each cycle

- Inverse metric: cycles/second (Hertz) or cycles/ns (Ghz)
- Determined by micro-architecture, technology parameters

Total Time

CPU Time = # Instructions x CPI x Clock Cycle Time

sec/prgrm = Instr/prgm x cycles/instr x seconds/cycle

E.g. Say for a program with 400k instructions, 30 MHz, CPI 2.1:

CPU [Execution] Time = ?

Total Time

CPU Time = # Instructions x CPI x Clock Cycle Time

sec/prgrm = Instr/prgm x cycles/instr x seconds/cycle

E.g. Say for a program with 400k instructions, 30 MHz, CPI 2.1 :

CPU [Execution] Time = 400k x 2.1 x 33 ns = 27 ms

Total Time

CPU Time = # Instructions x CPI x Clock Cycle Time

sec/prgrm = Instr/prgm x cycles/instr x seconds/cycle

E.g. Say for a program with 400k instructions, 30 MHz, CPI 2.1:
CPU [Execution] Time = 400k x 2.1 x 33 ns = 27 ms

How do we increase performance?

- Need to reduce CPU time
  - Reduce #instructions
  - Reduce CPI
  - Reduce Clock Cycle Time

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

$$CPI = 0.25 \times 3 + 0.6 \times 2 + 0.15 \times 1$$
$$= 2.1$$

Goal: Make processor run 2x faster,
i.e. 30 MIPS instead of 15 MIPS

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~~ 1
- 15% branches, CPI = 1

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

$$= 1.5$$

First lets try CPI of 1 for arithmetic.

- Is that 2x faster overall?
- How much does it improve performance?

No

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~~ X
- 15% branches, CPI = 1

$$CPI = 1.05 = 0.25 \times 3 + 0.6 \times \underline{X} + 0.15 \times 1$$
$$1.05 = .75 + 0.6X + 0.15$$
$$X = 0.25$$

But, want to half our CPI from 2.1 to 1.05.

Let new arithmetic operation have a CPI of X. X =?

Then, X = 0.25, which is a significant improvement

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~~ 0.25
- 15% branches, CPI = 1

To double performance CPI for arithmetic operations have to go from 2 to 0.25 but 0.25 CPI is not practically not possible

Single Cycle vs Pipelined Processor

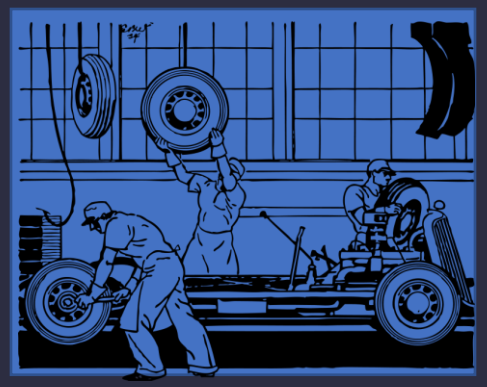
Single Cycle → Pipelining

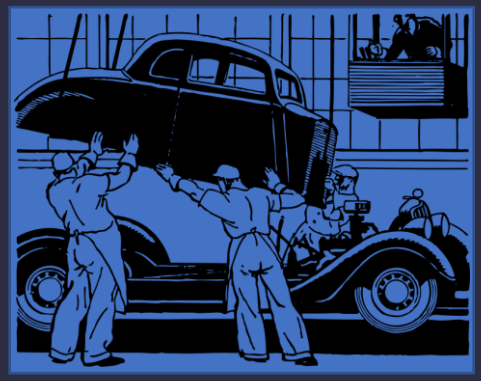
Single-cycle

Pipelined

Agenda

- 5-stage Pipeline
- Implementation
- Working Example

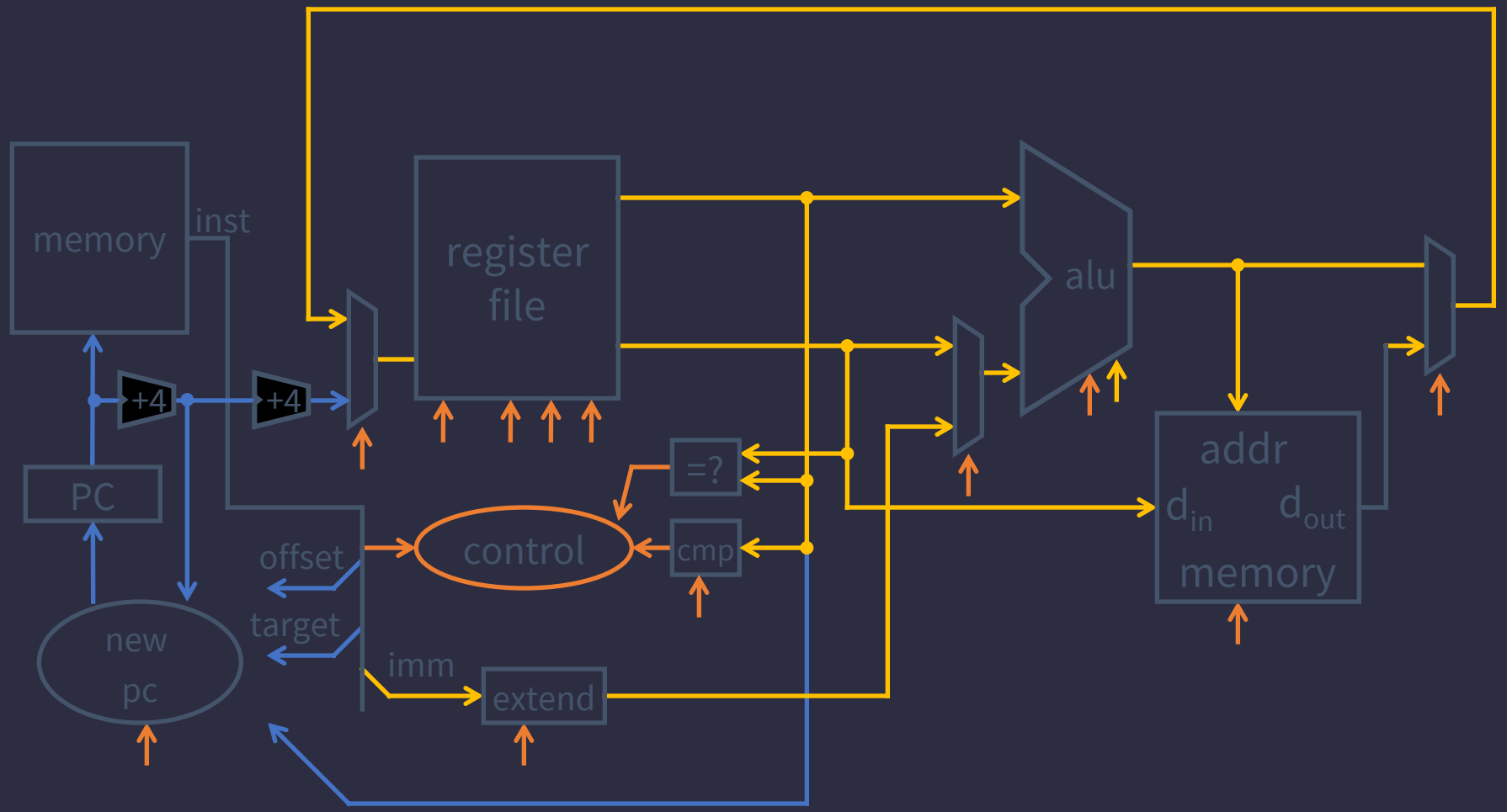




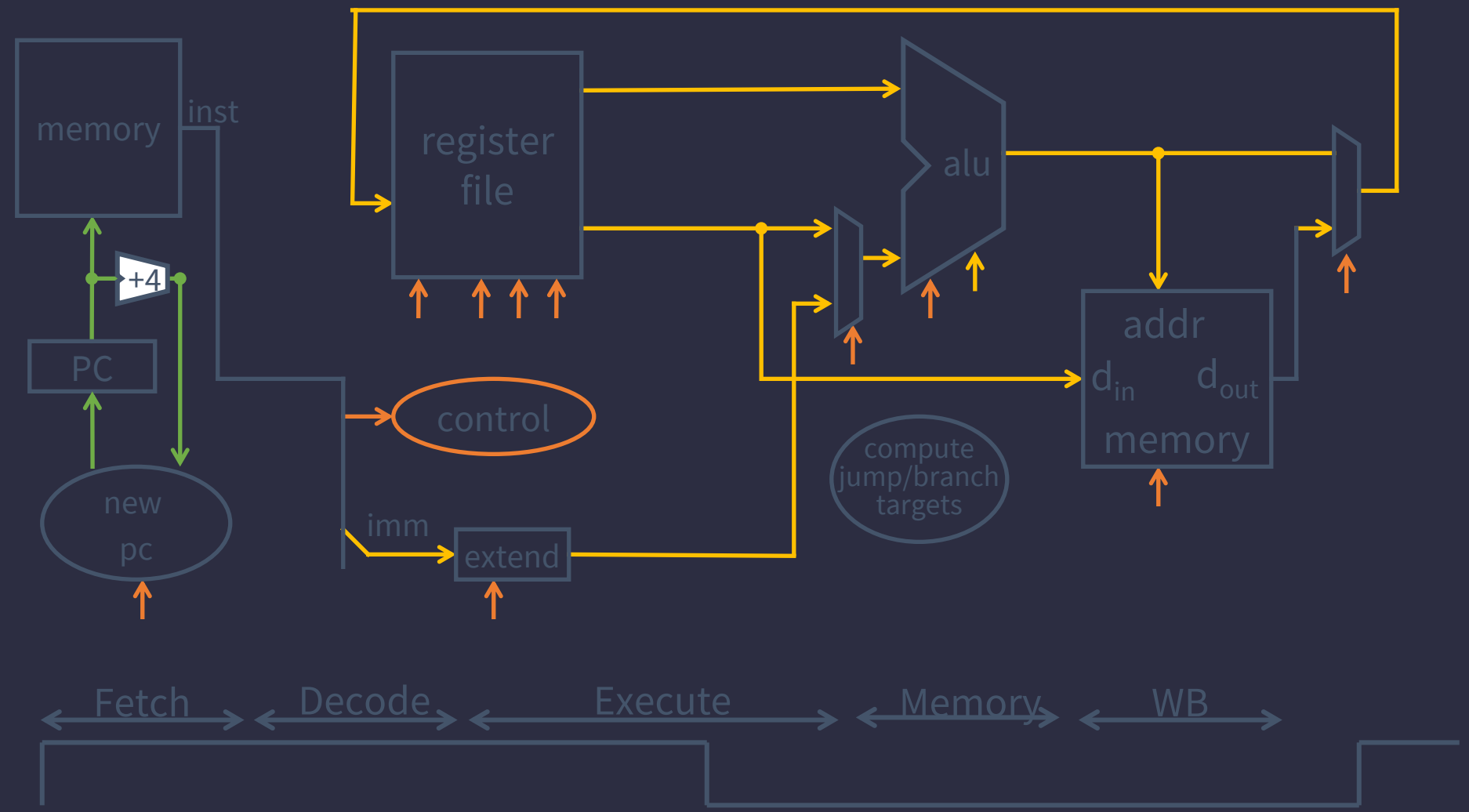
Hazards

- Structural
- Data Hazards
- Control Hazards

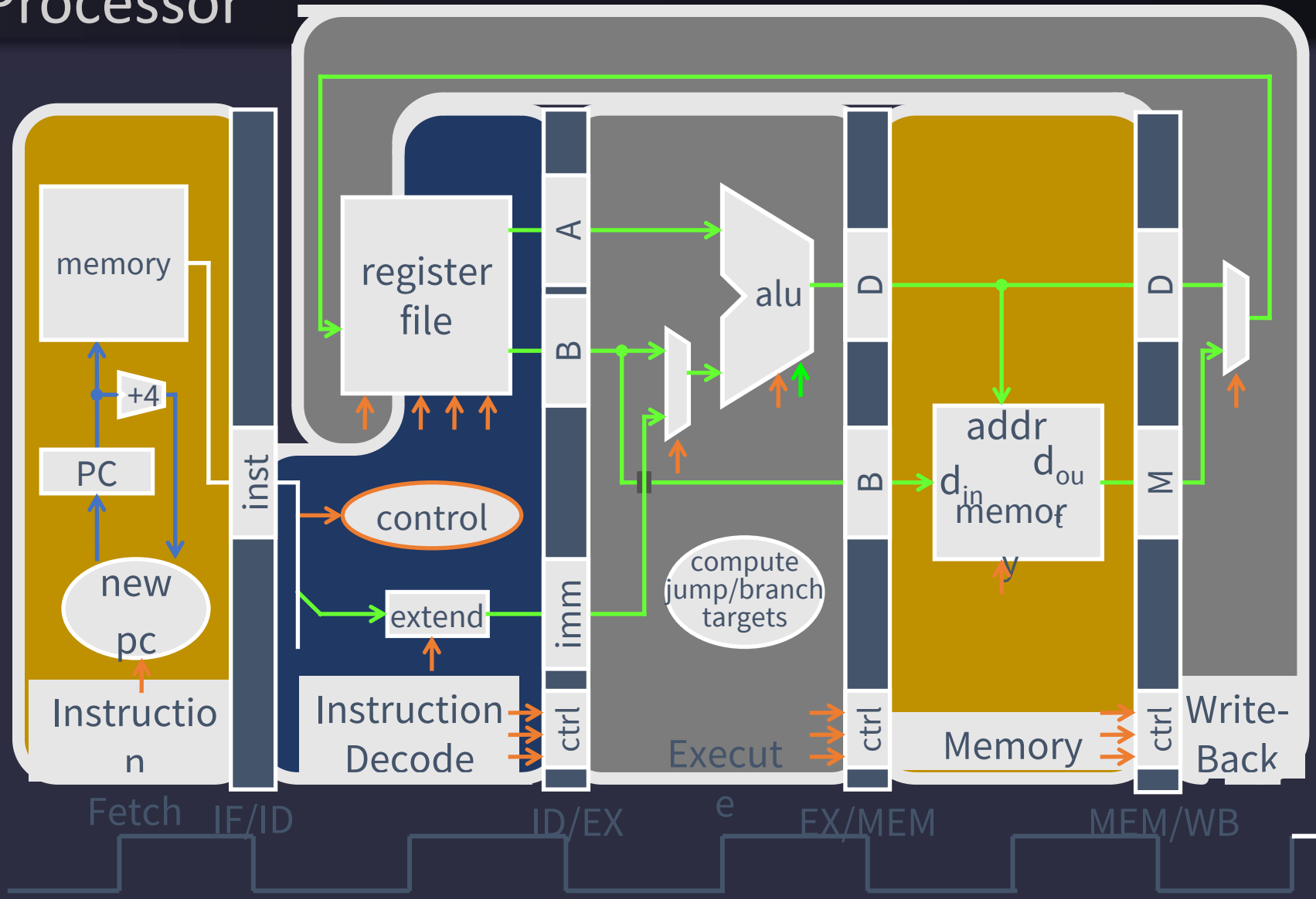
Review: Single Cycle Processor



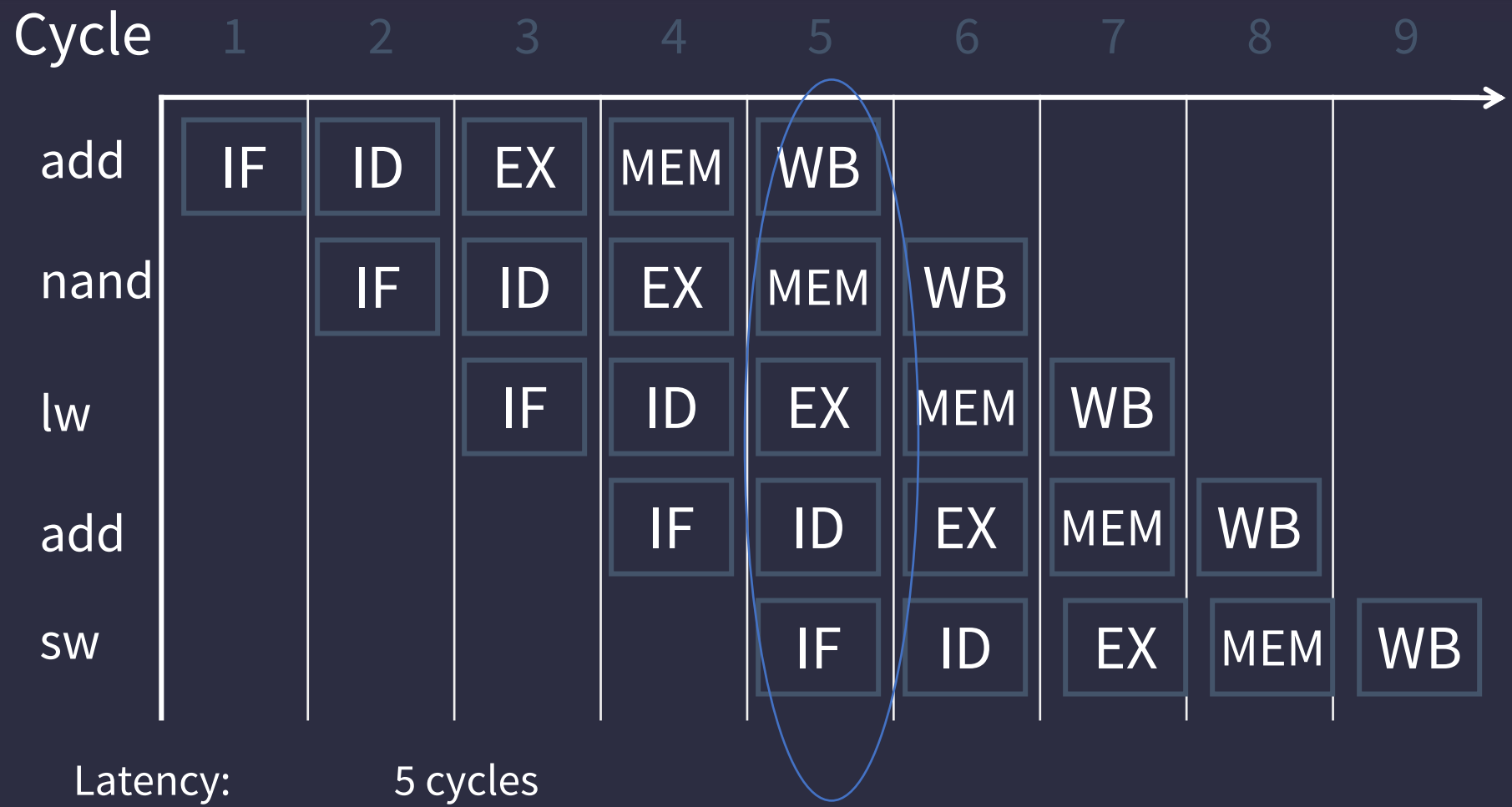
Pipelined Processor



Pipelined Processor



Time Graphs



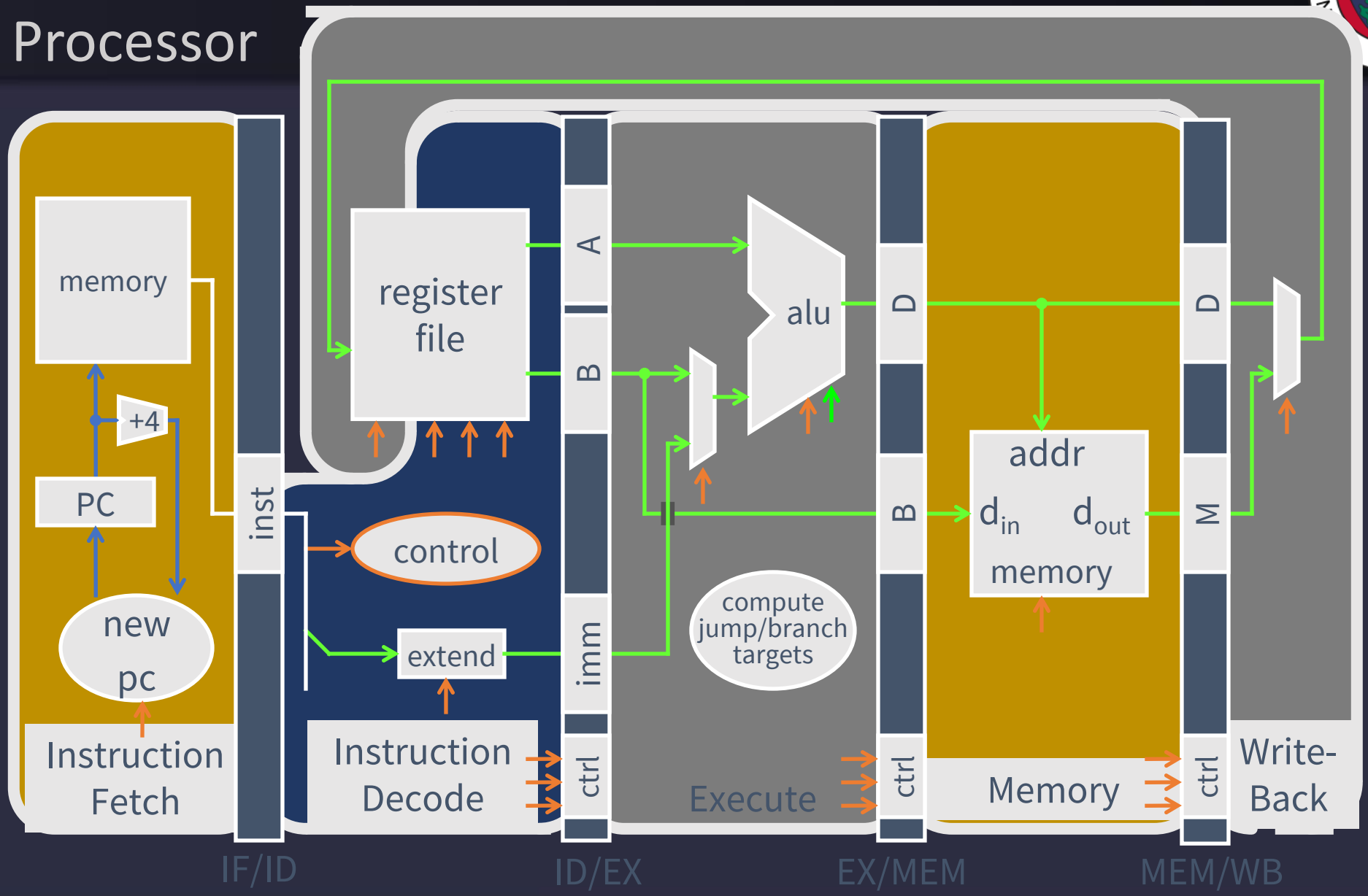
Latency: 5 cycles
Throughput: 1 insn/cycle
Concurrency: 5

CPI = 1

Principles of Pipelined Implementation

- Break datapath into multiple cycles (here 5)
  - Parallel execution increases throughput
  - Balanced pipeline very important
    - Slowest stage determines clock rate
    - Imbalance kills performance
- Add pipeline registers (flip-flops) for isolation
- Resolve hazards

Pipelined Processor



Pipeline Stages

Stage	Perform Functionality	Register values of interest
Fetch	Use PC to index Program Memory, increment PC	Instruction bits (to be decoded) PC + 4 (to compute branch targets)
Decode	Decode instruction, generate control signals, read register file	Control information, Rd index, immediates, offsets, register values (Ra, Rb), PC+4 (to compute branch targets)
Execute	Perform ALU operation Compute targets (PC+4+offset, etc.) in case this is a branch, decide if branch taken	Control information, Rd index, *etc.* Result of ALU operation, value in case this is a store instruction
Memory	Perform load/store if needed, address is ALU result	Control information, Rd index, *etc.* Result of load, pass result from execute
Writeback	Select value, write to register file	

Instruction Fetch (IF)

Stage 1: Instruction Fetch

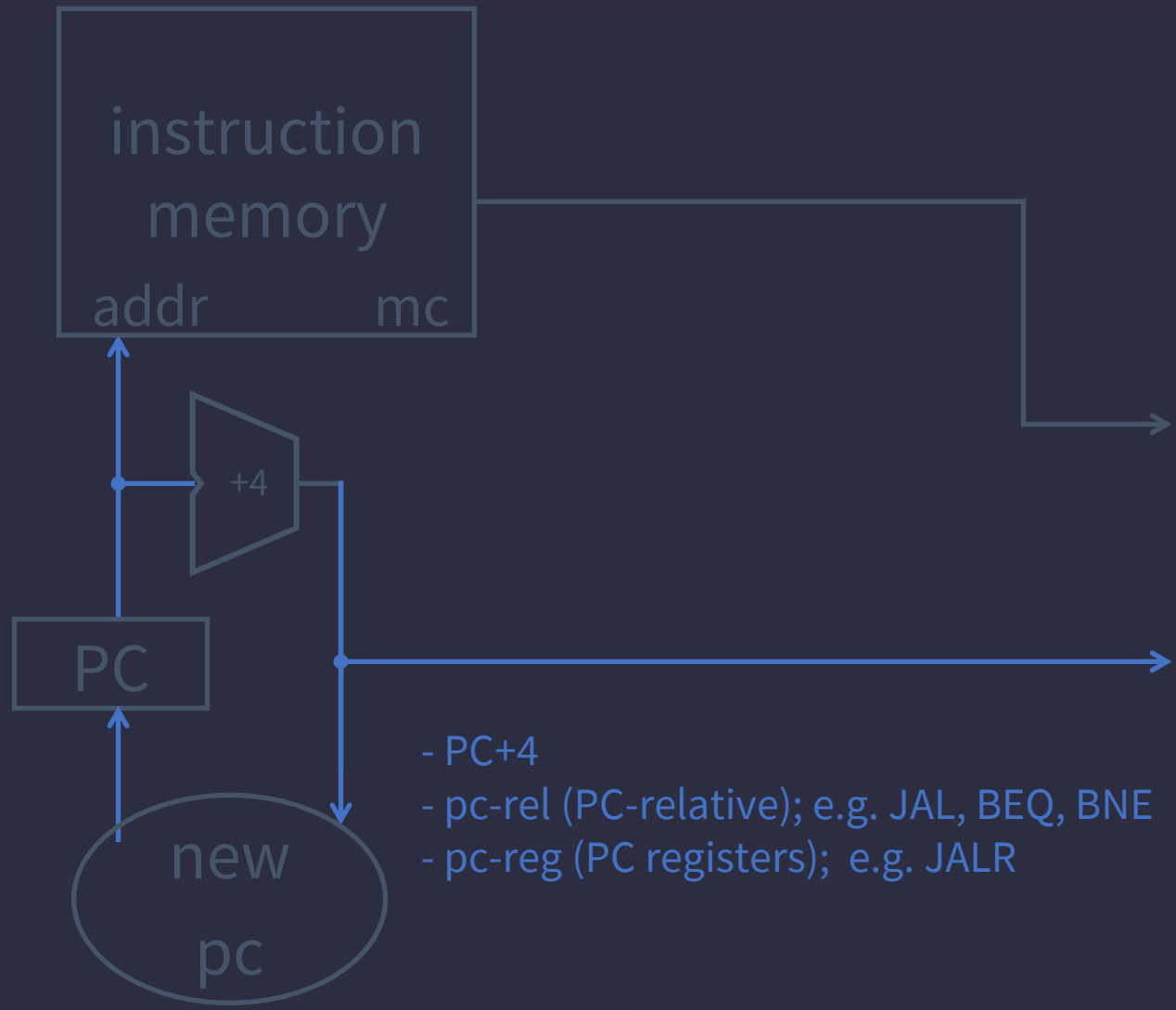
Fetch a new instruction every cycle

- Current PC is index to instruction memory
- Increment the PC at end of cycle (assume no branches for now)

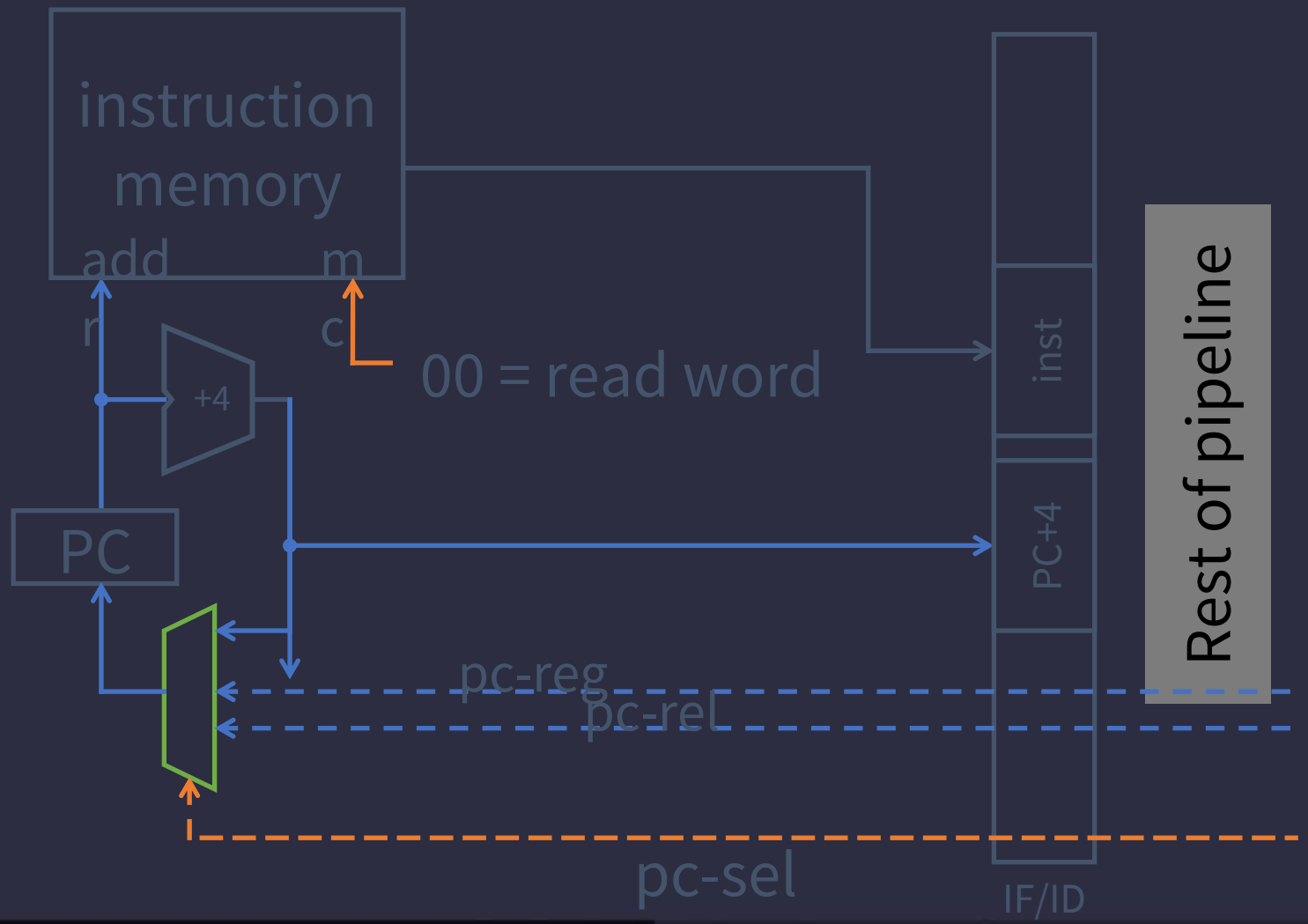
Write values of interest to pipeline register (IF/ID)

- Instruction bits (for later decoding)
- PC+4 (for later computing branch targets)

Instruction Fetch (IF)



Instruction Fetch (IF)



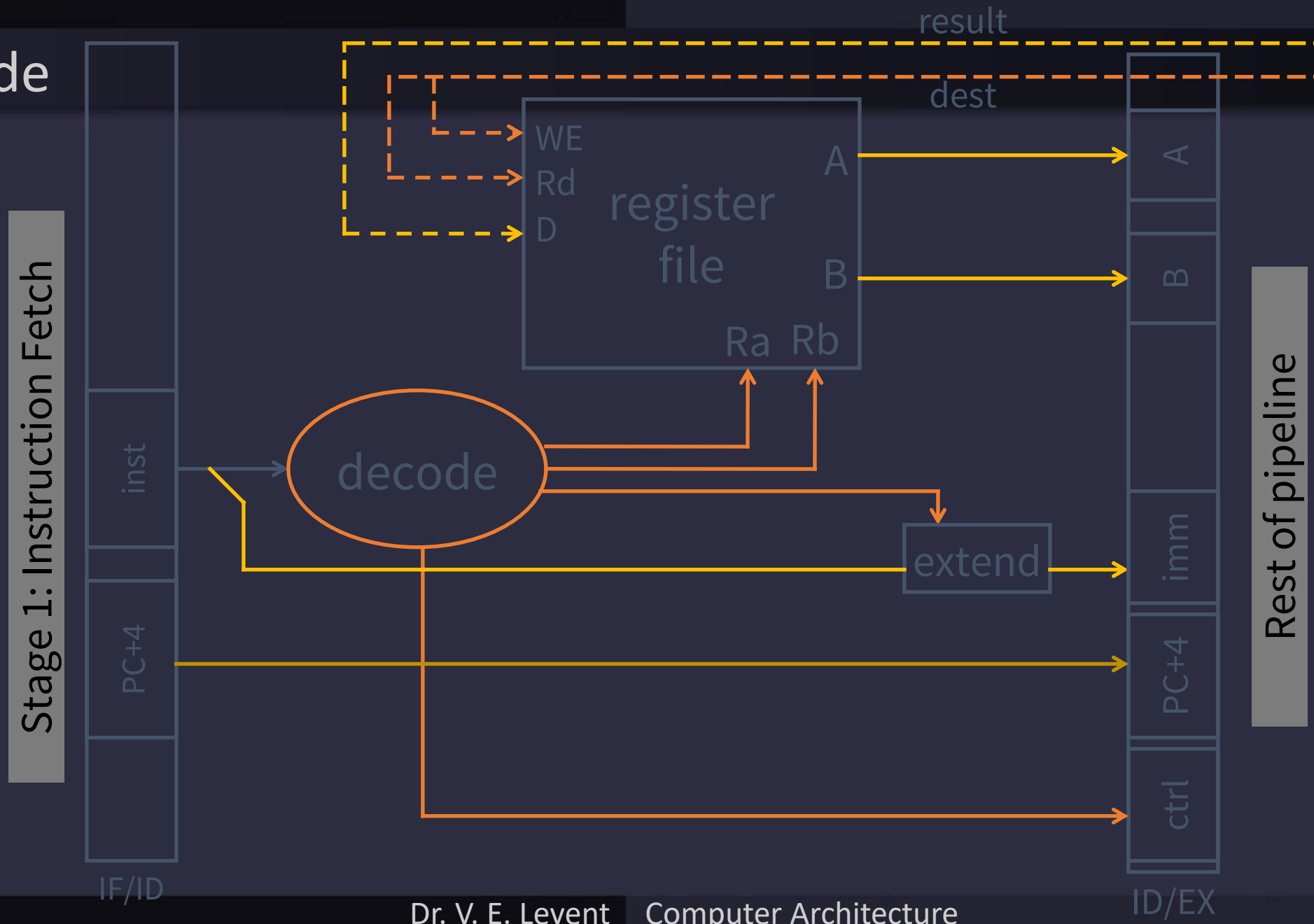
Decode

- Stage 2: Instruction Decode

- On every cycle:
  - Read IF/ID pipeline register to get instruction bits
  - Decode instruction, generate control signals
  - Read from register file

- Write values of interest to pipeline register (ID/EX)
  - Control information, Rd index, immediates, offsets, …
  - Contents of Ra, Rb
  - PC+4 (for computing branch targets later)

Decode


result
dest
WE
Rd
D
register file
A
B
Ra
Rb
Stage 1: Instruction Fetch
inst
PC+4
decode
extend
A
B
imm
PC+4
ctrl
Rest of pipeline
IF/ID
ID/EX

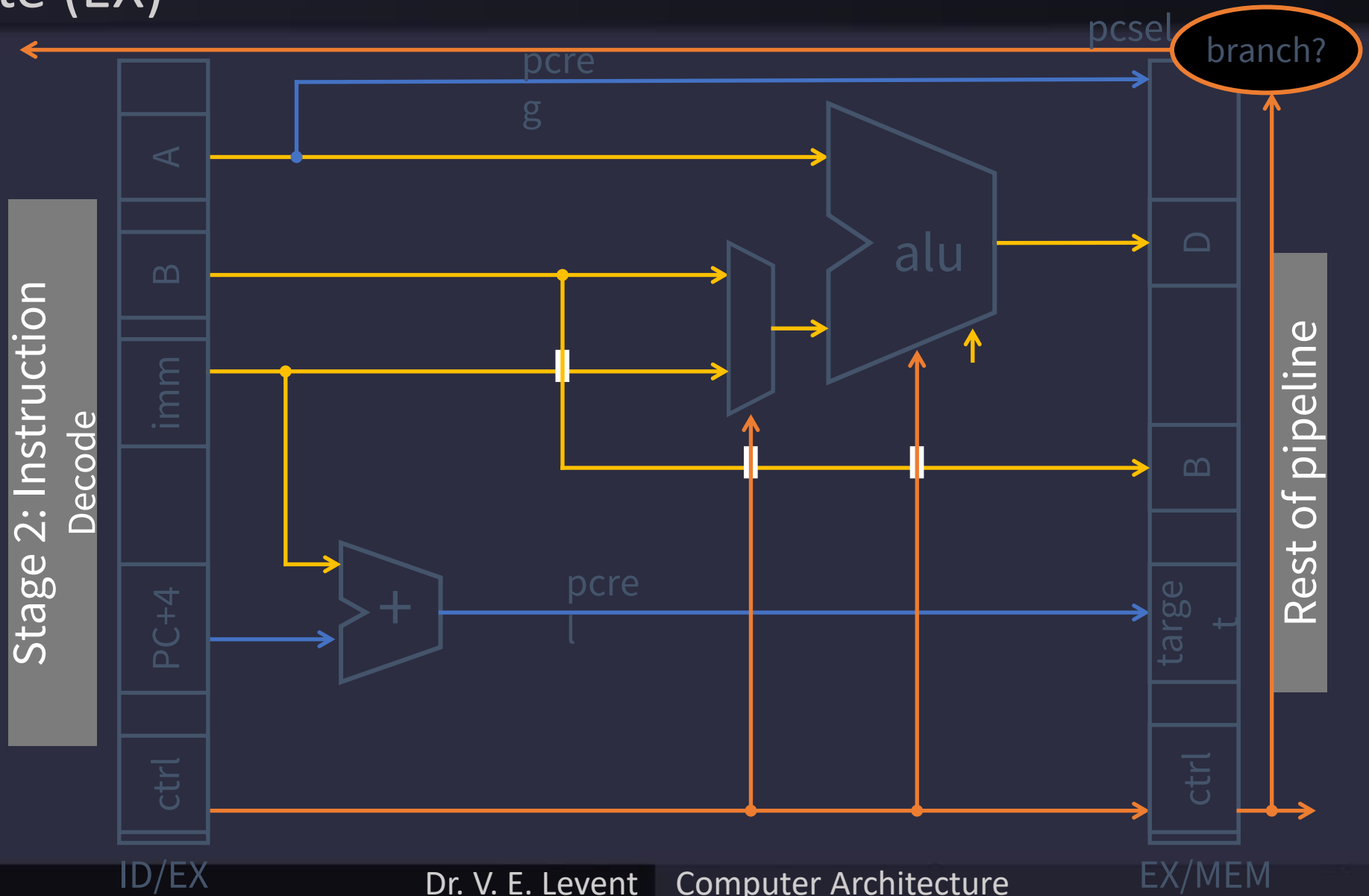
Execute (EX)

- Stage 3: Execute

- On every cycle:
  - Read ID/EX pipeline register to get values and control bits
  - Perform ALU operation
  - Compute targets (PC+4+offset, etc.) *in case* this is a branch
  - Decide if jump/branch should be taken

- Write values of interest to pipeline register (EX/MEM)
  - Control information, Rd index, …
  - Result of ALU operation
  - Value *in case* this is a memory store instruction

Execute (EX)


pcsel
branch?
pcre
g
A
B
imm
PC+4
ctrl
Stage 2: Instruction
Decode
alu
+
pcre
l
D
B
targe
t
ctrl
Rest of pipeline
ID/EX
EX/MEM

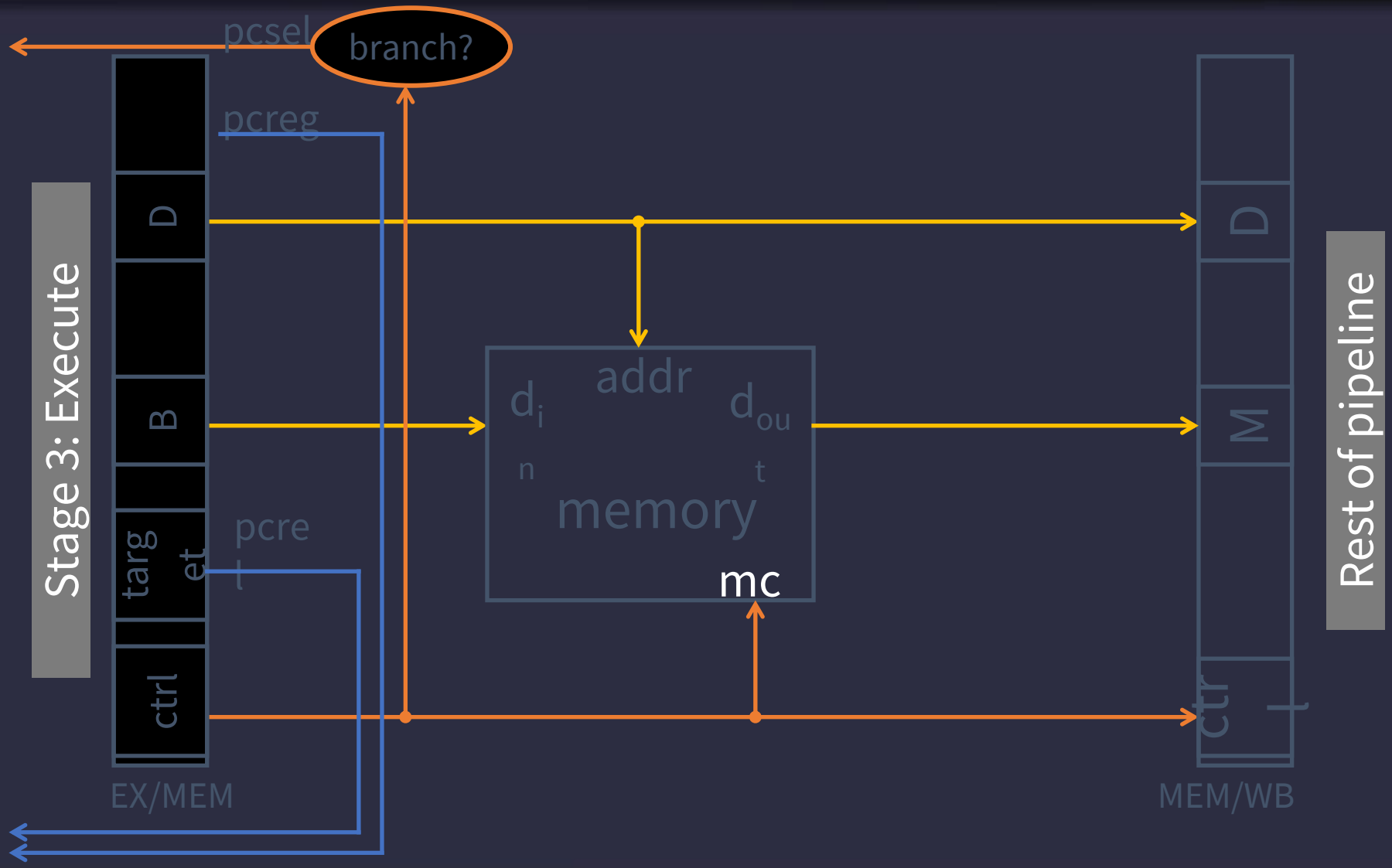
MEM

- Stage 4: Memory

- On every cycle:
  - Read EX/MEM pipeline register to get values and control bits
  - Perform memory load/store if needed
    - address is ALU result

- Write values of interest to pipeline register (MEM/WB)
  - Control information, Rd index, …
  - Result of memory operation
  - Pass result of ALU operation

MEM

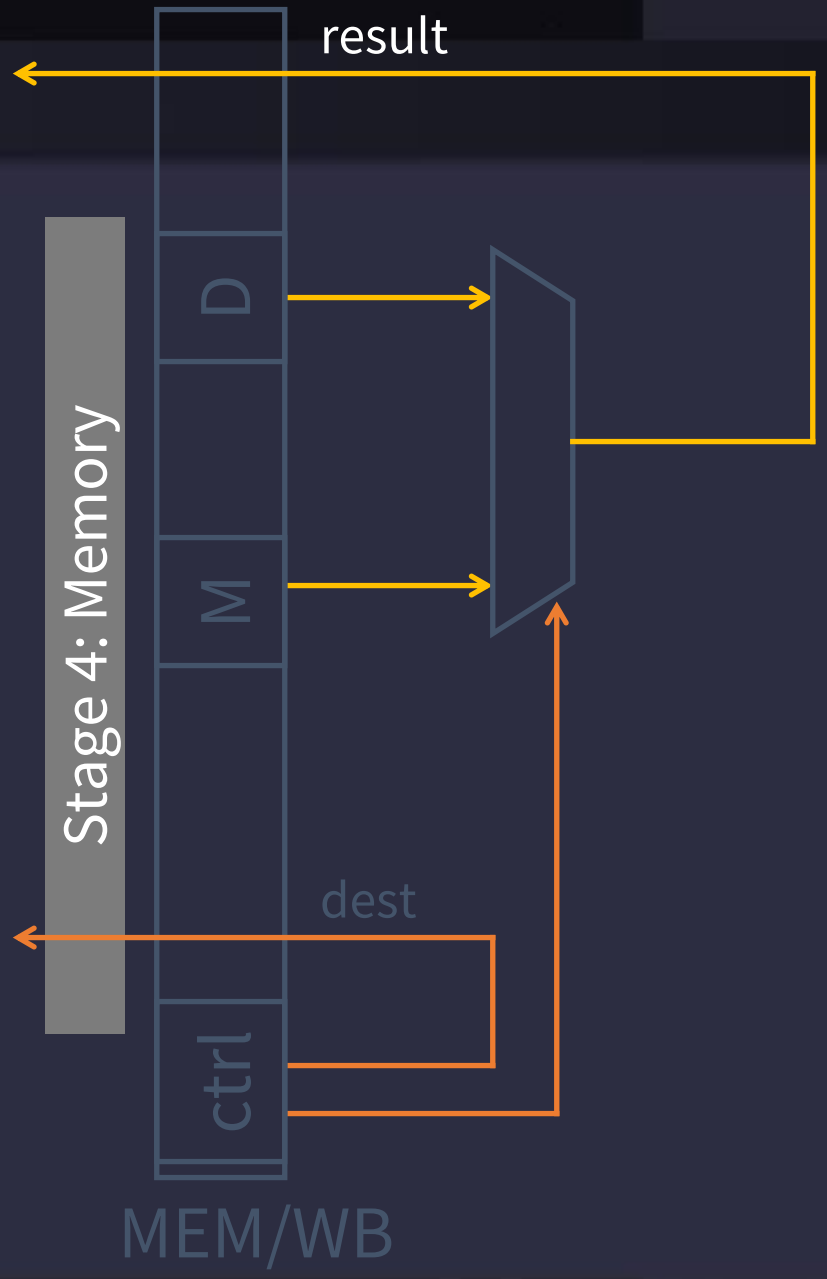

pcsel
branch?
pcreg
Stage 3: Execute
D
B
targ
et
ctrl
pcre
t
addr
d
i
n
d
ou
t
memory
mc
EX/MEM
D
M
ctr
l
MEM/WB
Rest of pipeline

WB

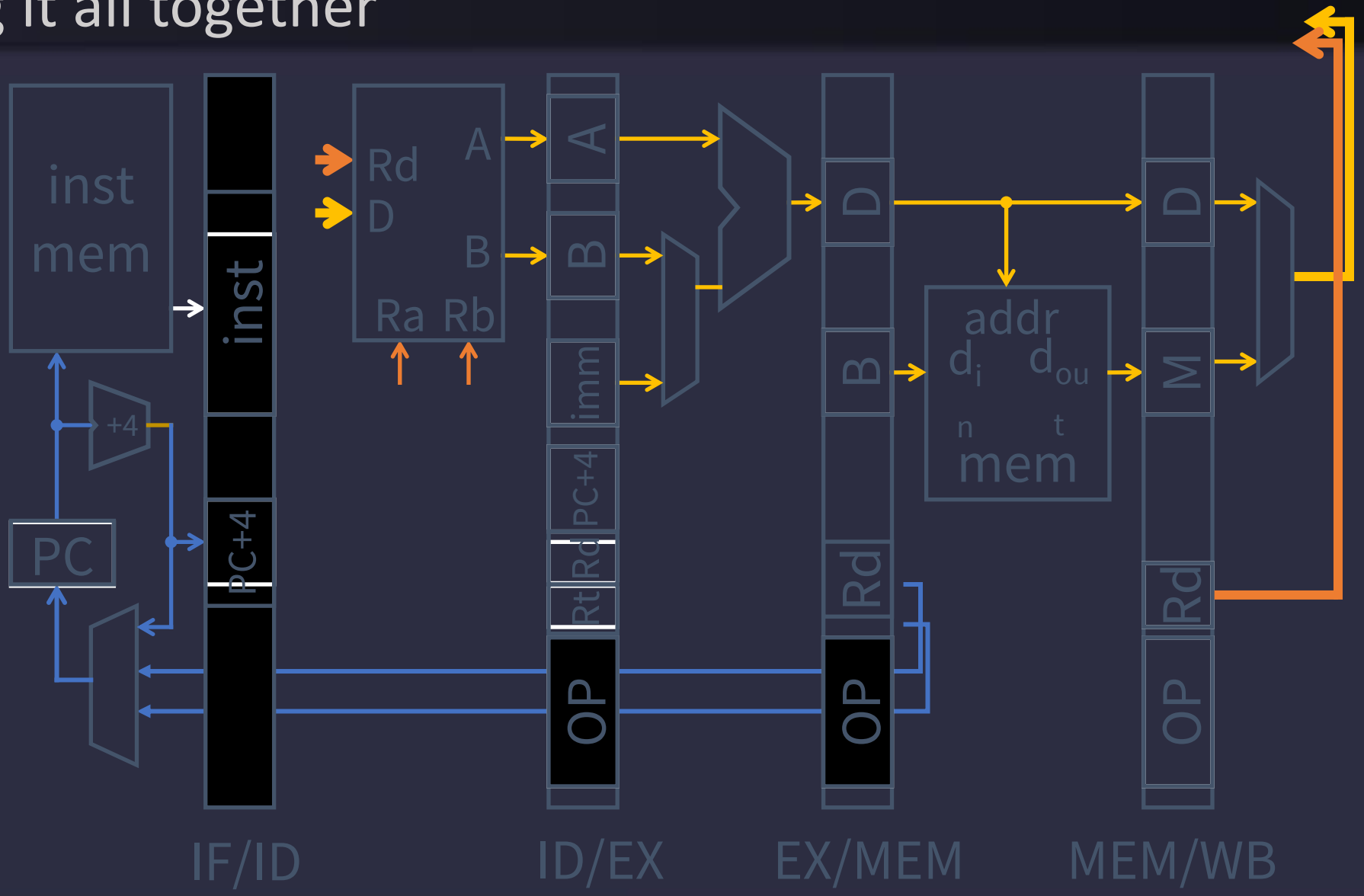
- Stage 5: Write-back

- On every cycle:
  - Read MEM/WB pipeline register to get values and control bits
  - Select value and write to register file

WB



Putting it all together



RISC-V is *designed* for pipelining

- Instructions same length
  - 32 bits, easy to fetch and then decode
- 4 types of instruction formats
  - Easy to route bits between stages
  - Can read a register source before even knowing what the instruction is
- Memory access through lw and sw only
  - Access memory after ALU