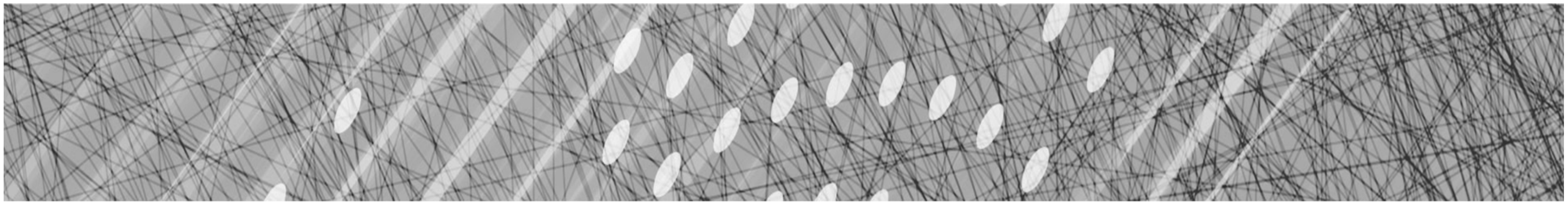
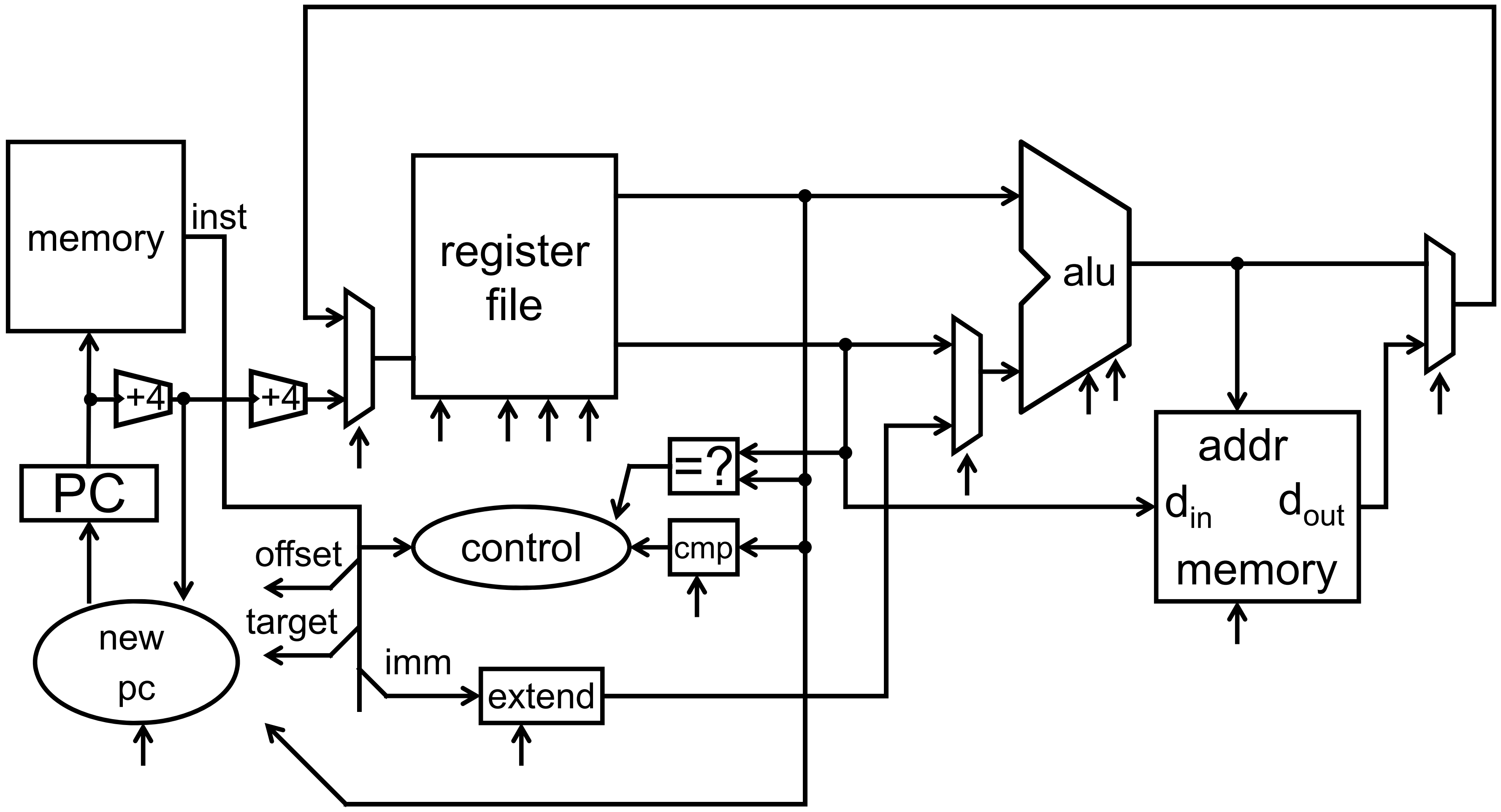


Pipelining

Hakim Weatherspoon
CS 3410
Computer Science
Cornell University

[Weatherspoon, Bala, Bracy, McKee, and Sirer]

Review: Single Cycle Processor



Review: Single Cycle Processor

- Advantages
  - Single cycle per instruction make logic and clock simple
- Disadvantages
  - Since instructions take different time to finish, memory and functional unit are not efficiently utilized
  - Cycle time is the longest delay
    - Load instruction
  - Best possible CPI is 1 (actually < 1 w parallelism)
    - However, lower MIPS and longer clock period (lower clock frequency); hence, lower performance

Review: Multi Cycle Processor

- Advantages
  - Better MIPS and smaller clock period (higher clock frequency)
  - Hence, better performance than Single Cycle processor
- Disadvantages
  - Higher CPI than single cycle processor
- Pipelining: Want better Performance
  - want small CPI (close to 1) with high MIPS and short clock period (high clock frequency)

Improving Performance

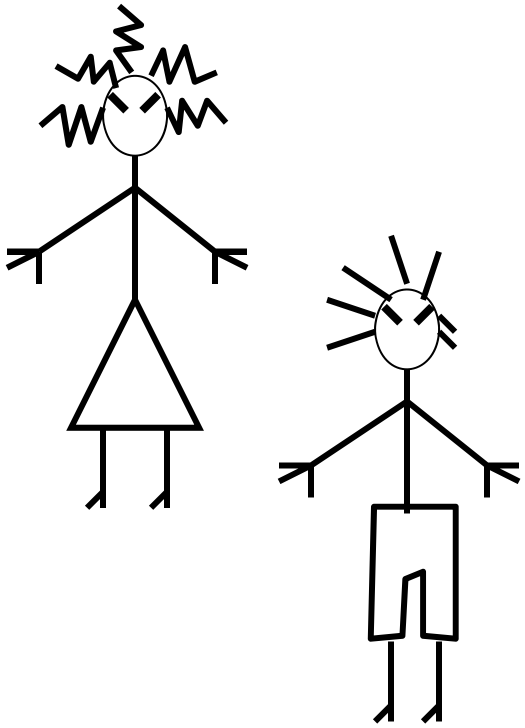
- Parallelism
- Pipelining
- Both!

The Kids

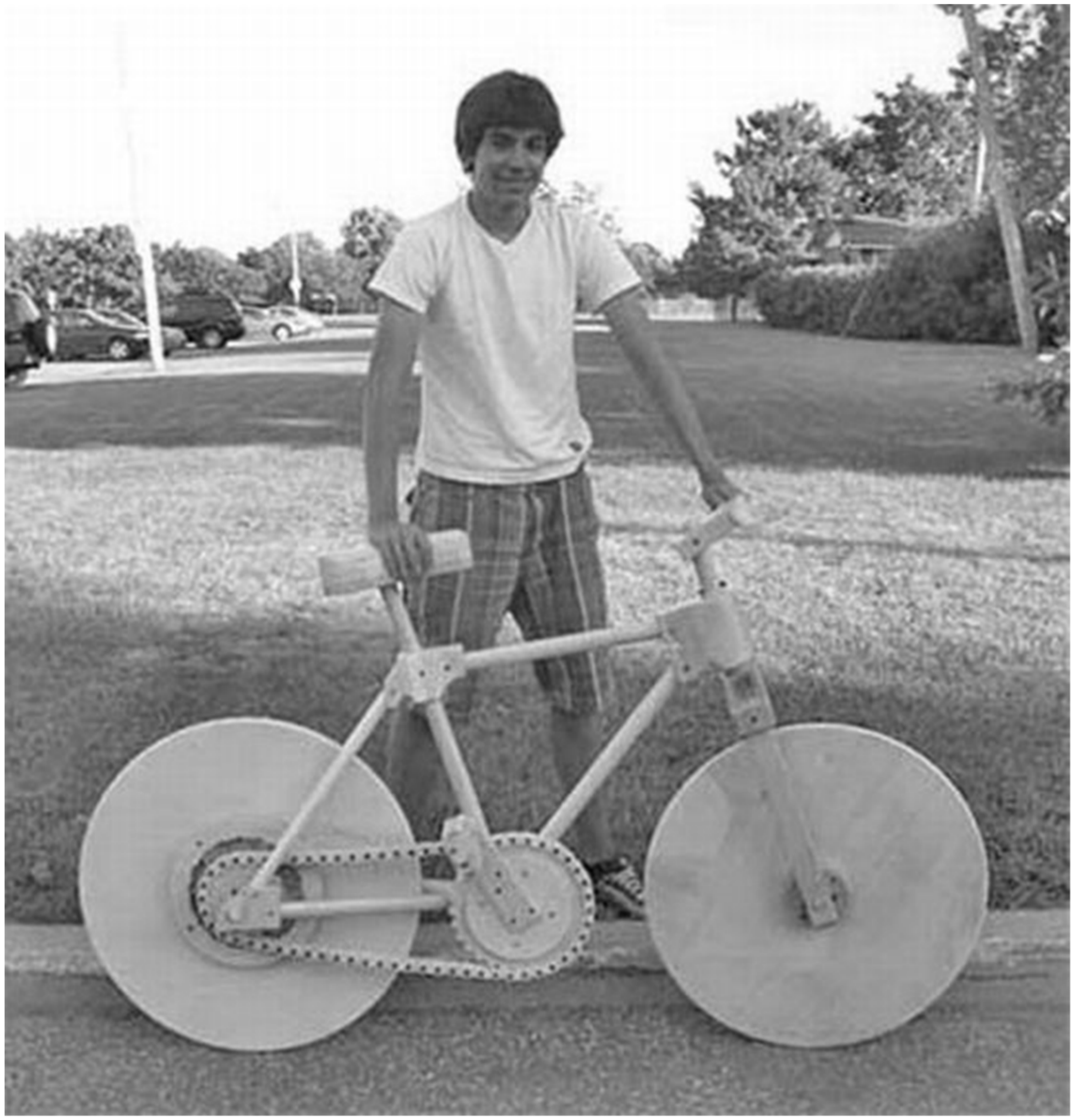
Alice

Bob

They don’t always get along…



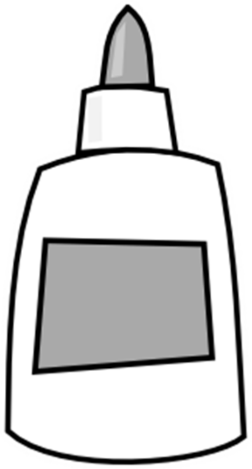
The Bicycle



The Materials

Saw

Drill

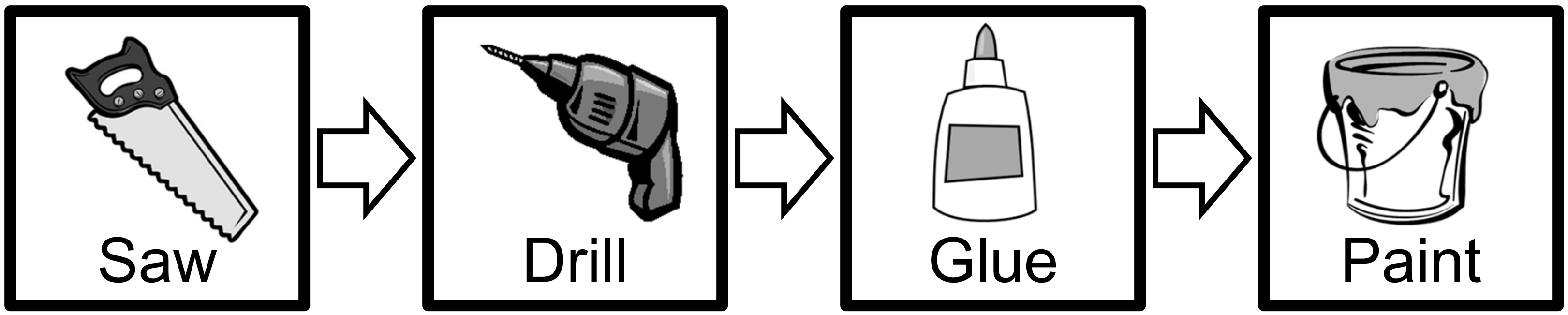


Glue

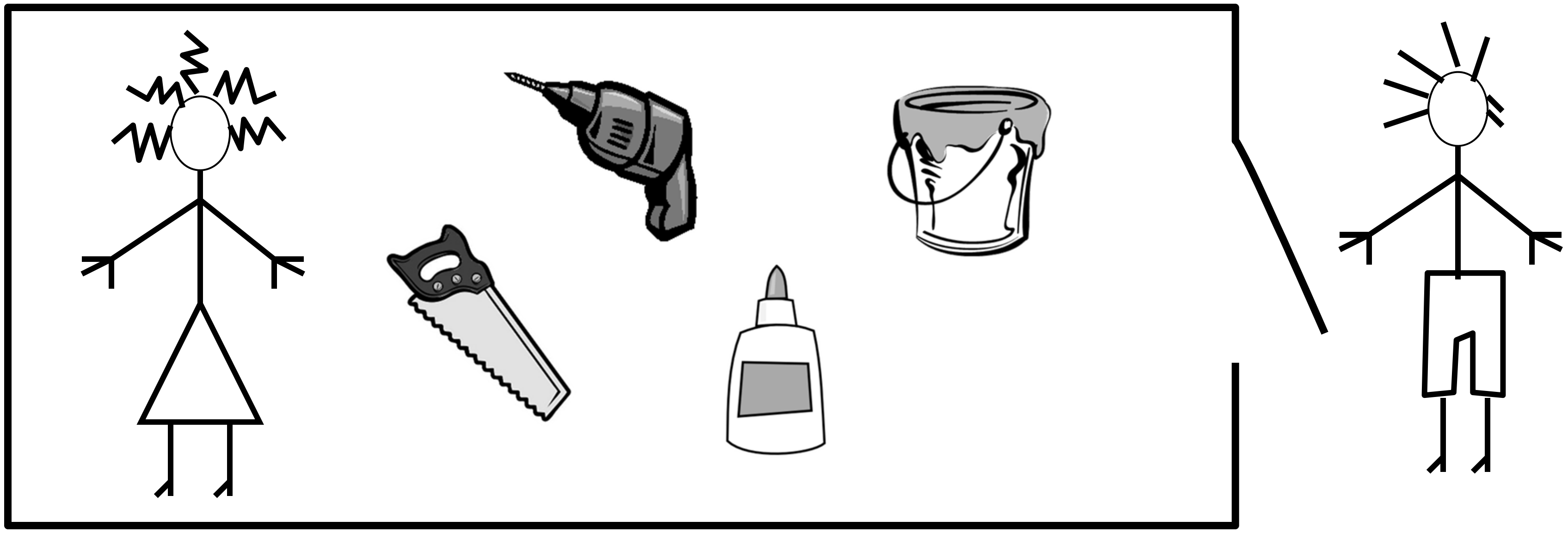
Paint

The Instructions

N pieces, each built following same sequence:



Design 1: Sequential Schedule



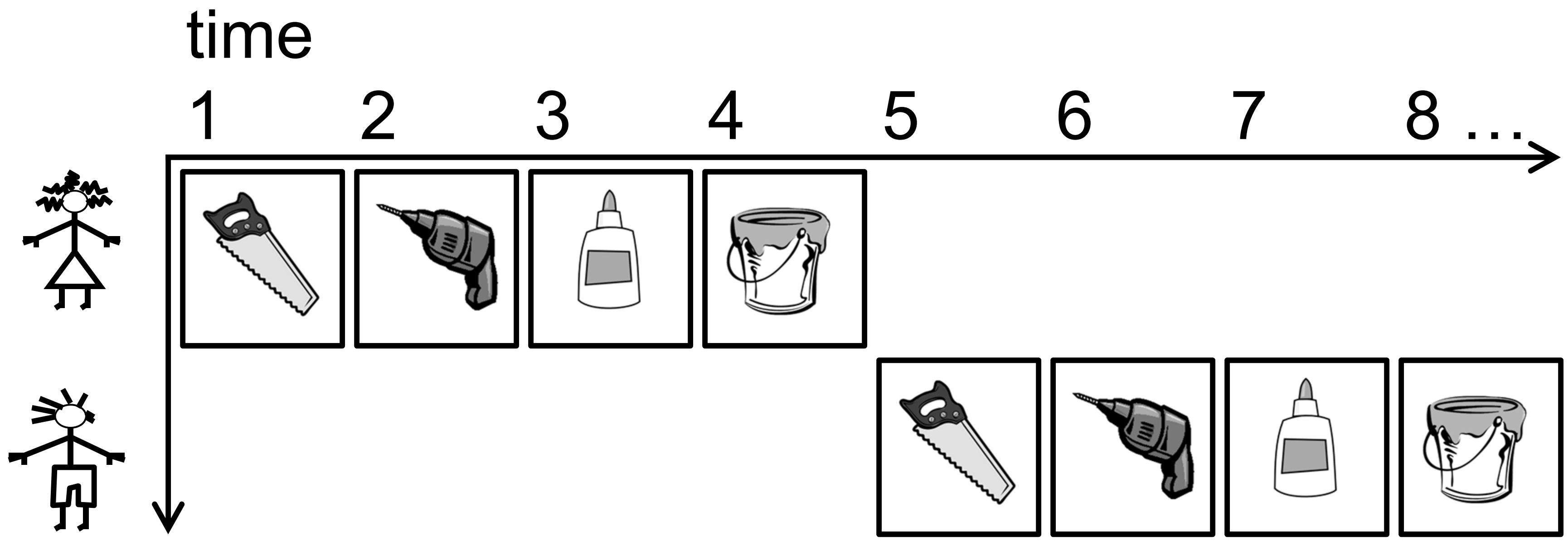
Alice owns the room

Bob can enter when Alice is finished

Repeat for remaining tasks

No possibility for conflicts

Sequential Performance



Latency:

Throughput:

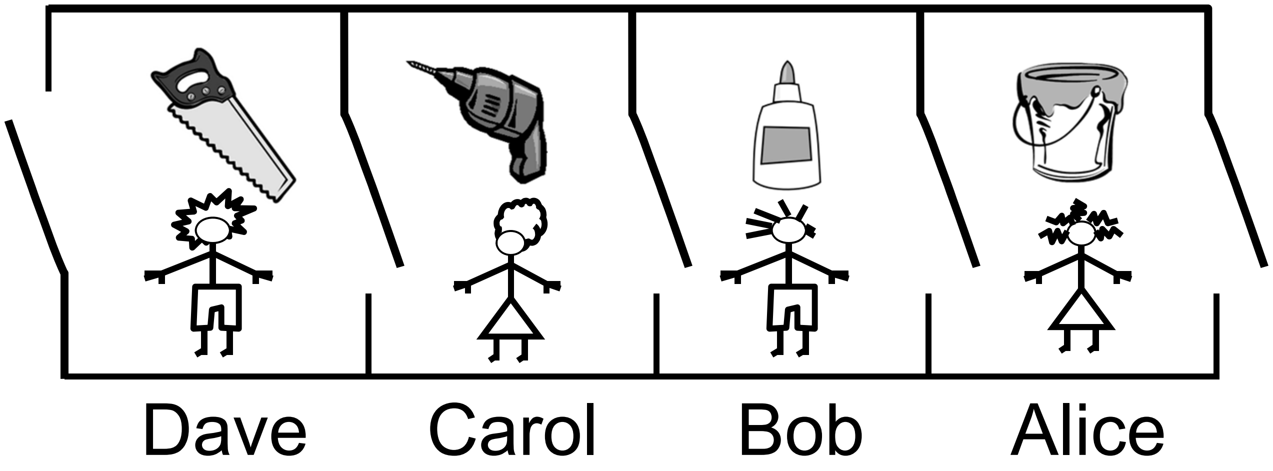
Concurrency:

Can we do better?

CPI =

Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*



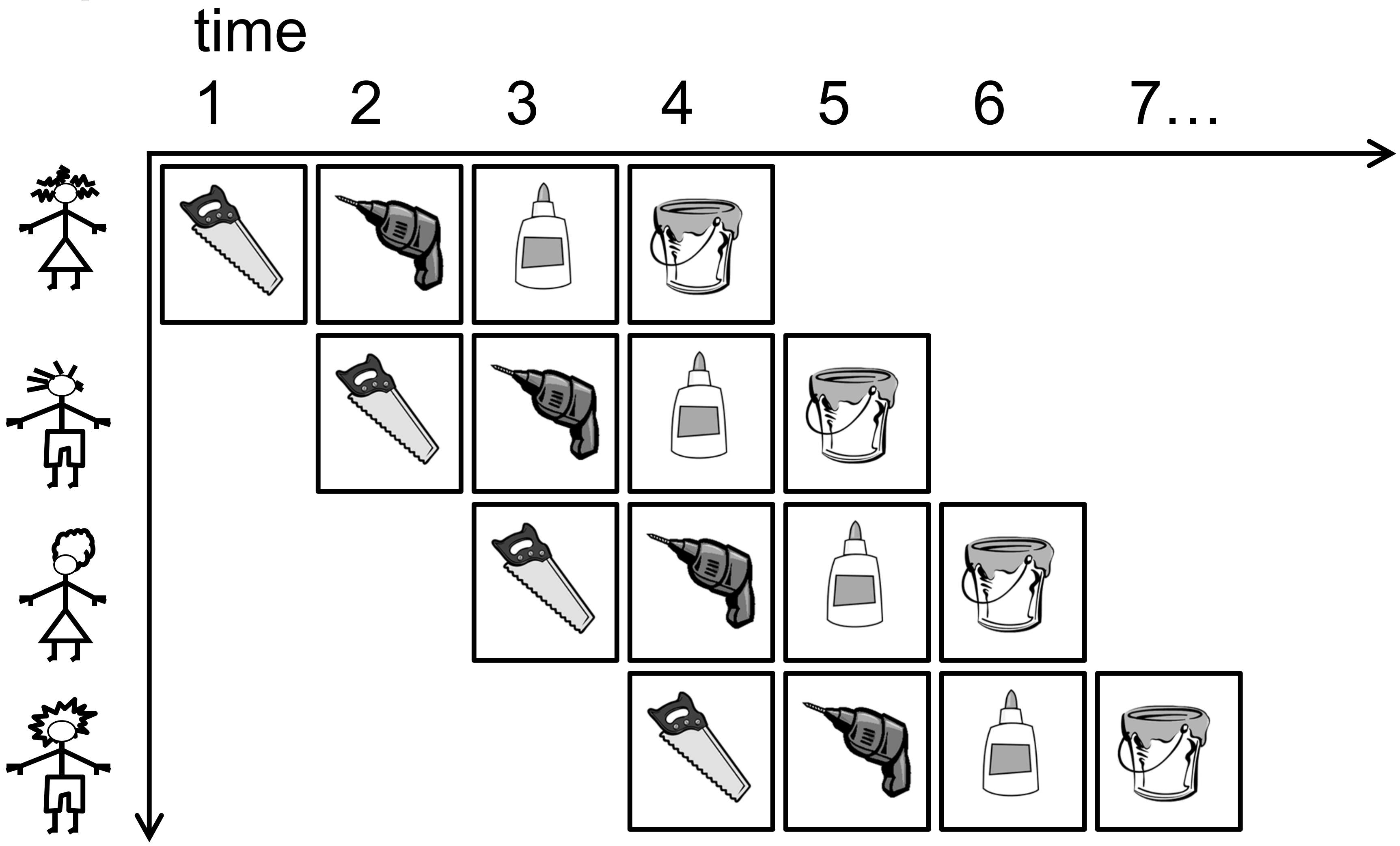
One person owns a stage at a time

4 stages

4 people working simultaneously

Everyone moves right in lockstep

Pipelined Performance



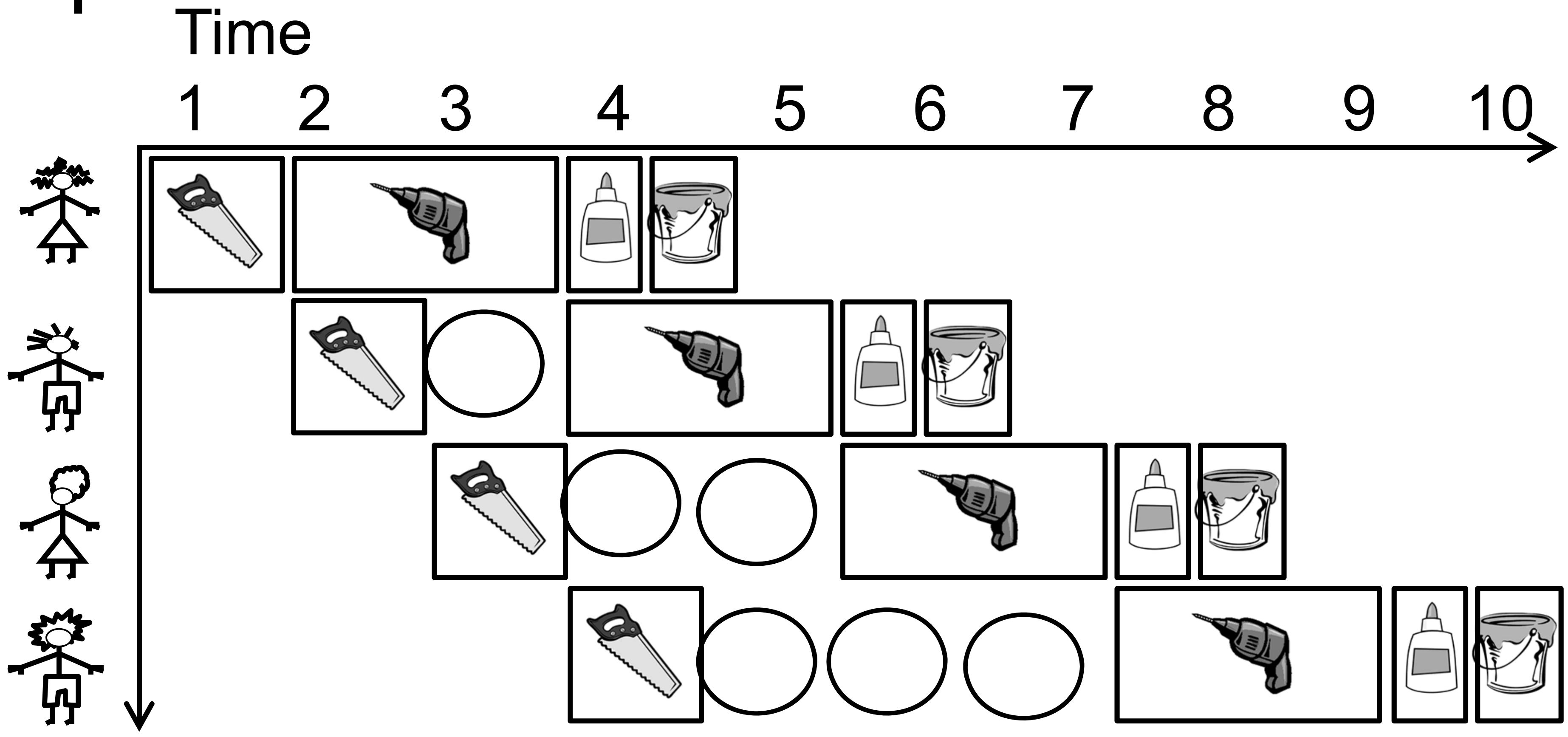
Latency:

Throughput:

Concurrency:

CPI =

Pipelined Performance



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

Latency:

Throughput: CPI =

Lessons

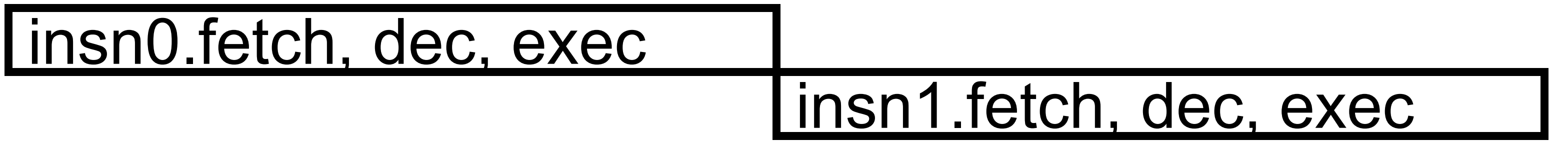
- Principle:
- Throughput increased by parallel execution
- Balanced pipeline very important
  - Else slowest stage dominates performance

- Pipelining:
  - Identify *pipeline stages*
  - Isolate stages from each other
  - Resolve pipeline *hazards* (next lecture)

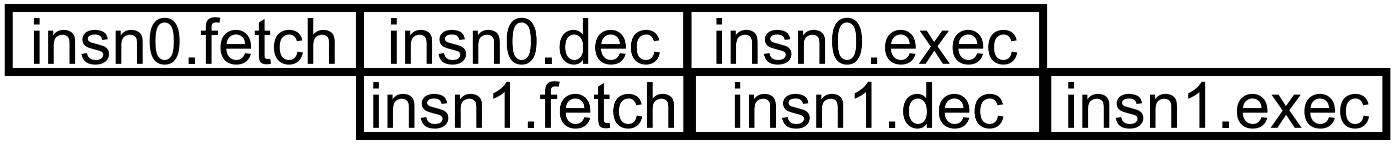
Single Cycle vs Pipelined Processor

Single Cycle → Pipelining

Single-cycle

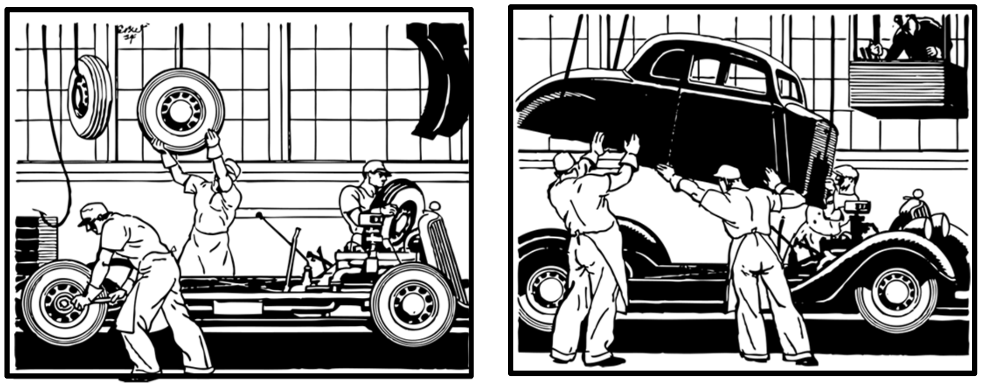


Pipelined



Agenda

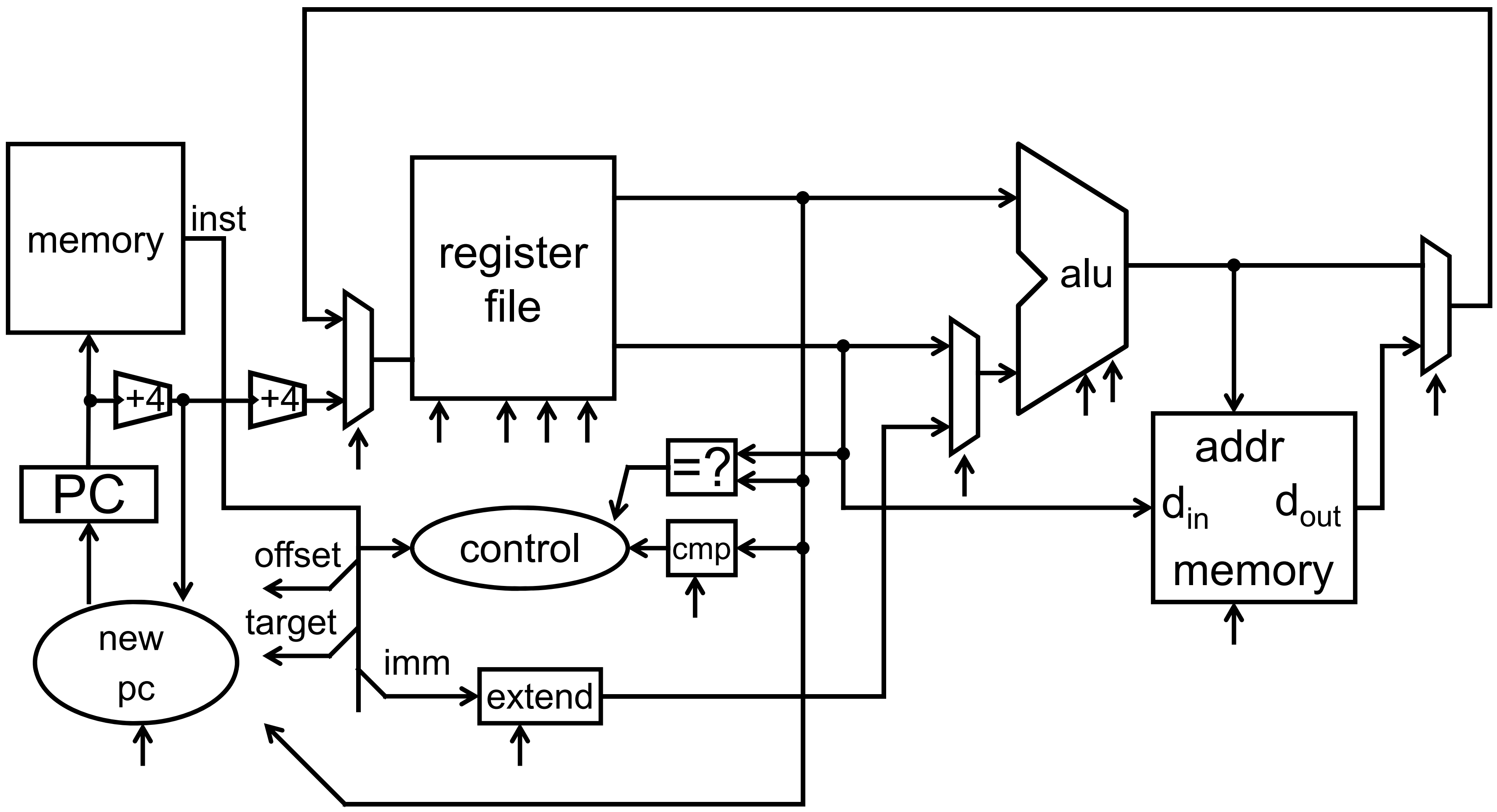
- 5-stage Pipeline
- Implementation
- Working Example



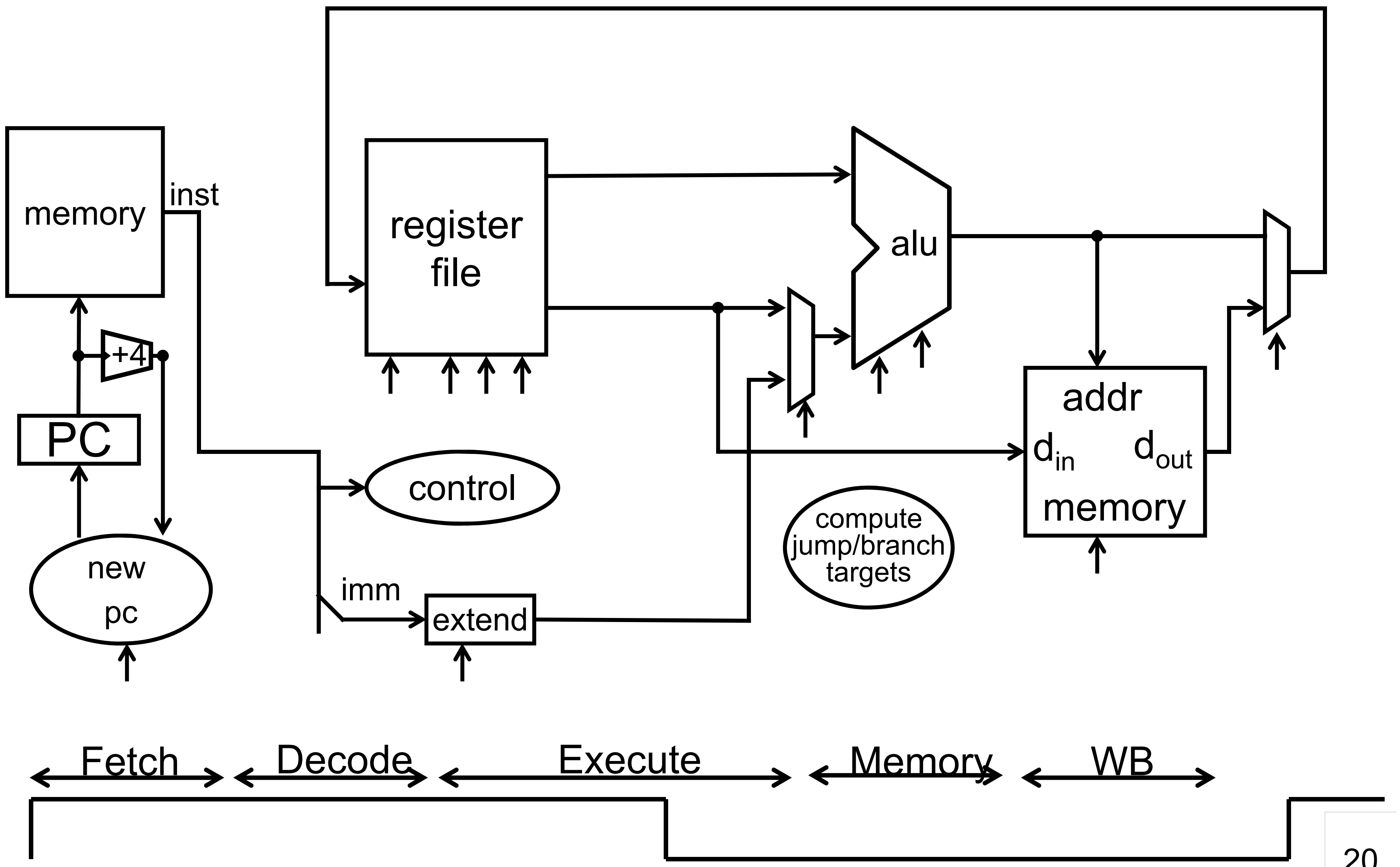
Hazards

- Structural
- Data Hazards
- Control Hazards

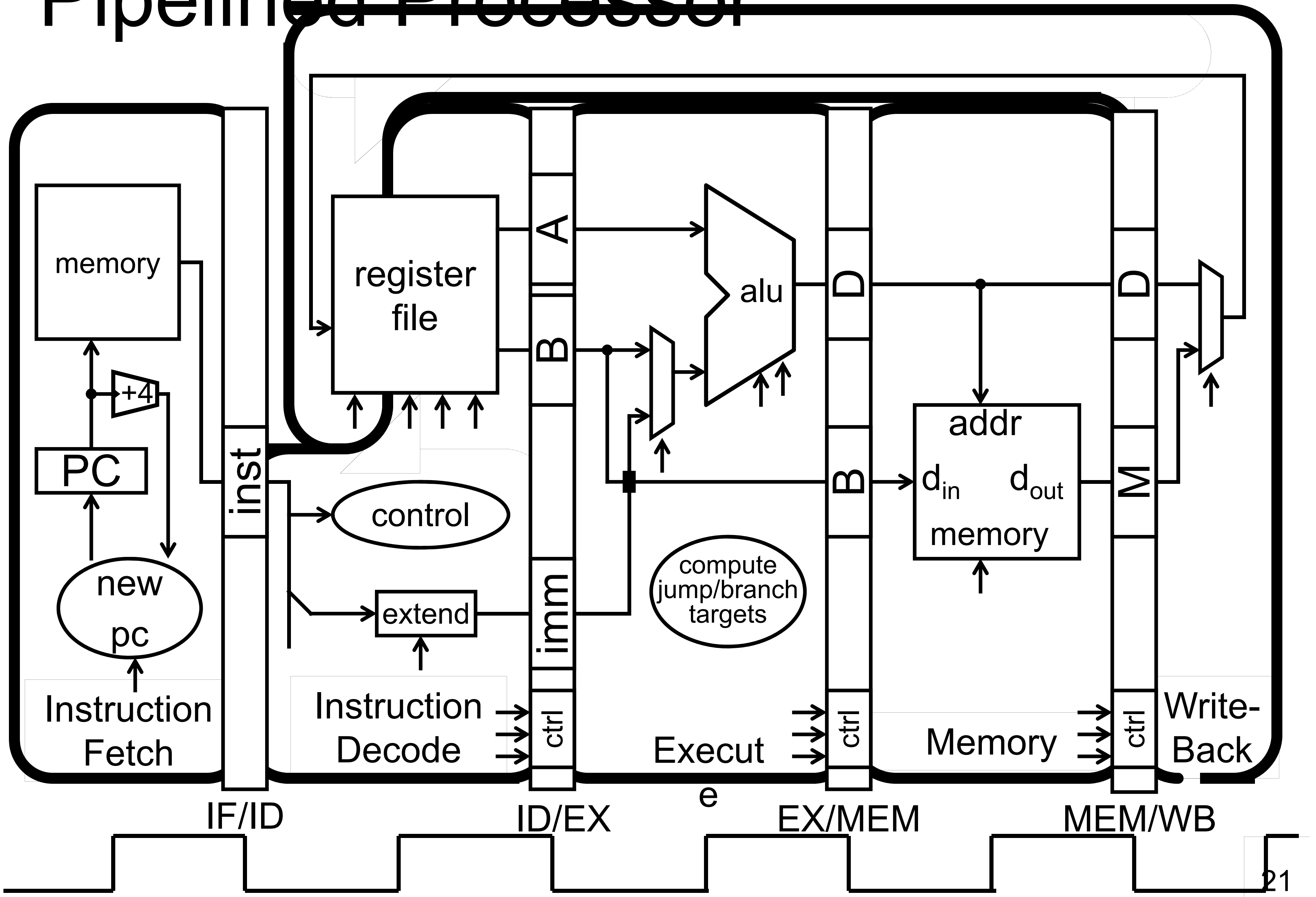
Review: Single Cycle Processor



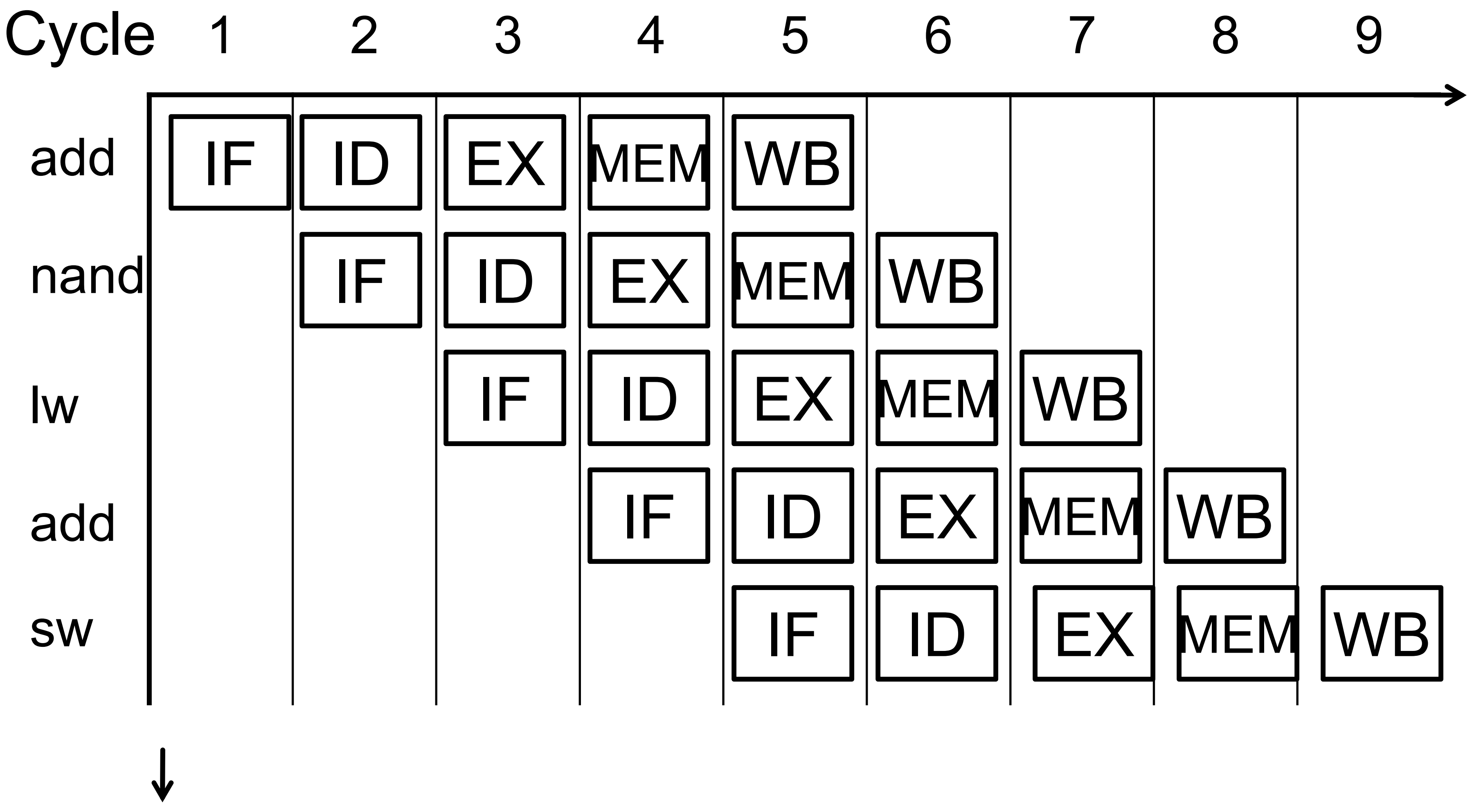
Pipelined Processor



Pipelined Processor



Time Graphs



Latency:

Throughput:

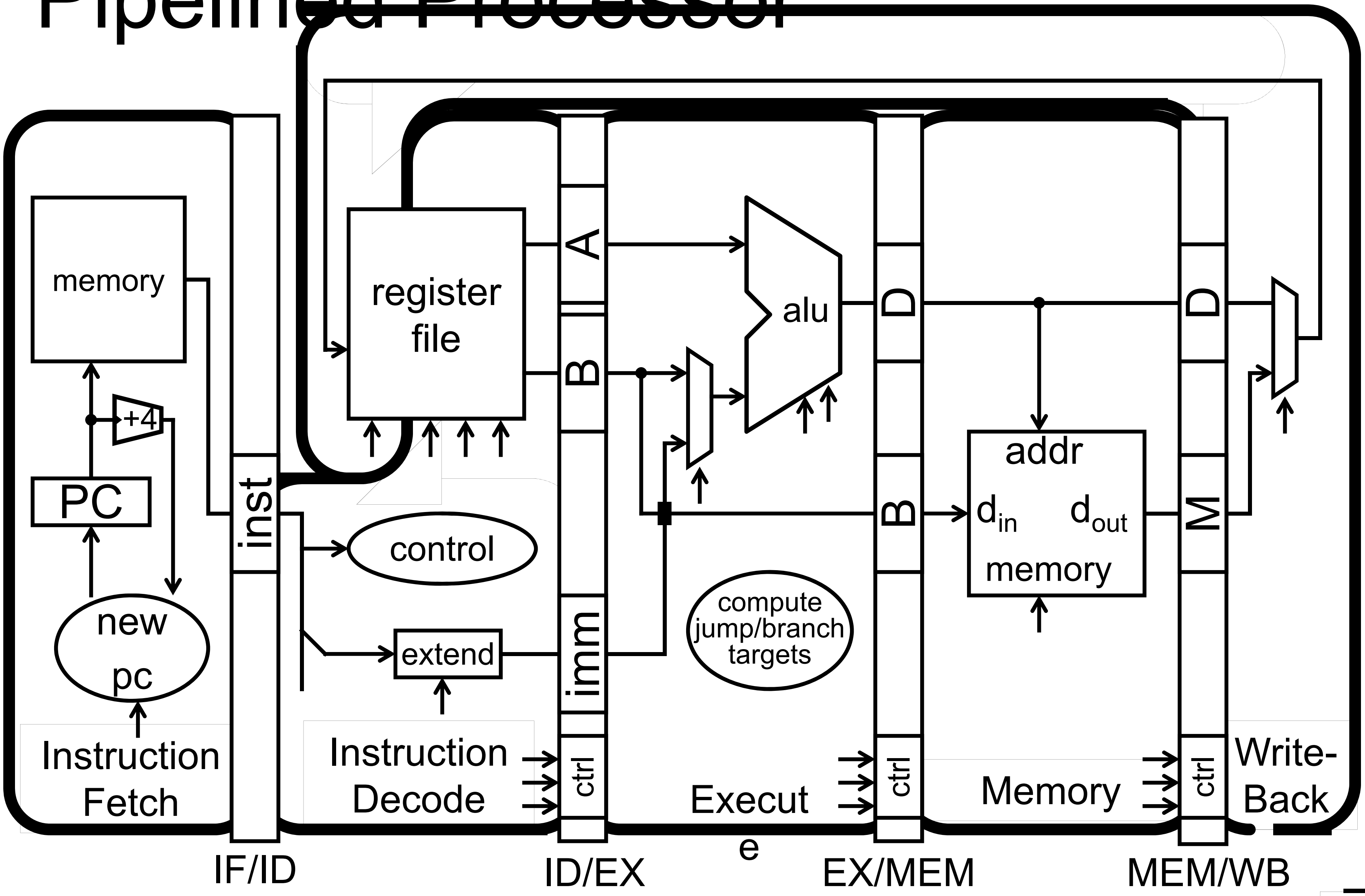
Concurrency:

CPI =

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
    - Each stage begins by reading values *from* latch
    - Each stage ends by writing values *to* latch
- Resolve hazards

Pipelined Processor



Pipeline Stages

Stage	Perform Functionality	Latch 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

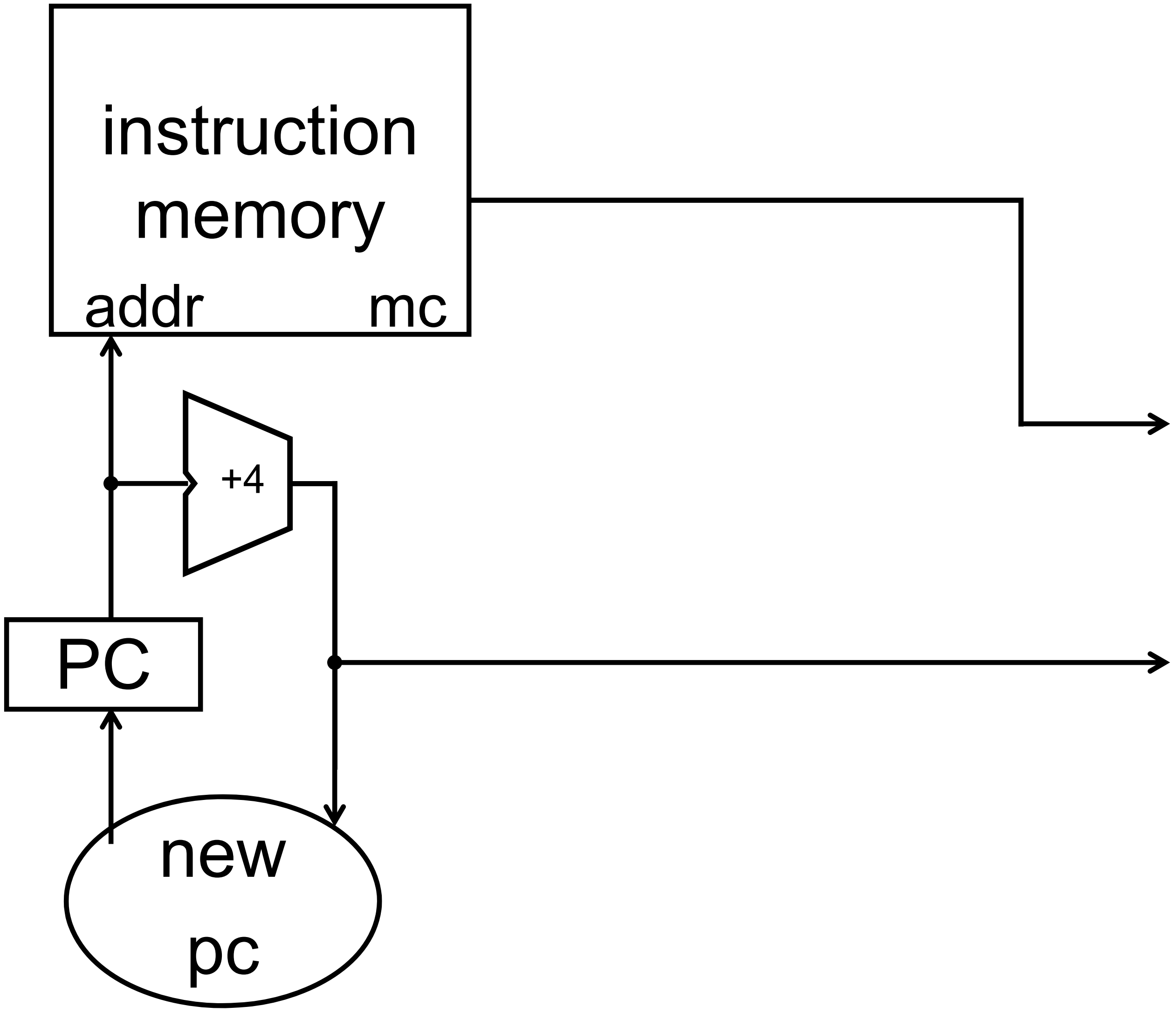
Fetch a new instruction every cycle

- 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)



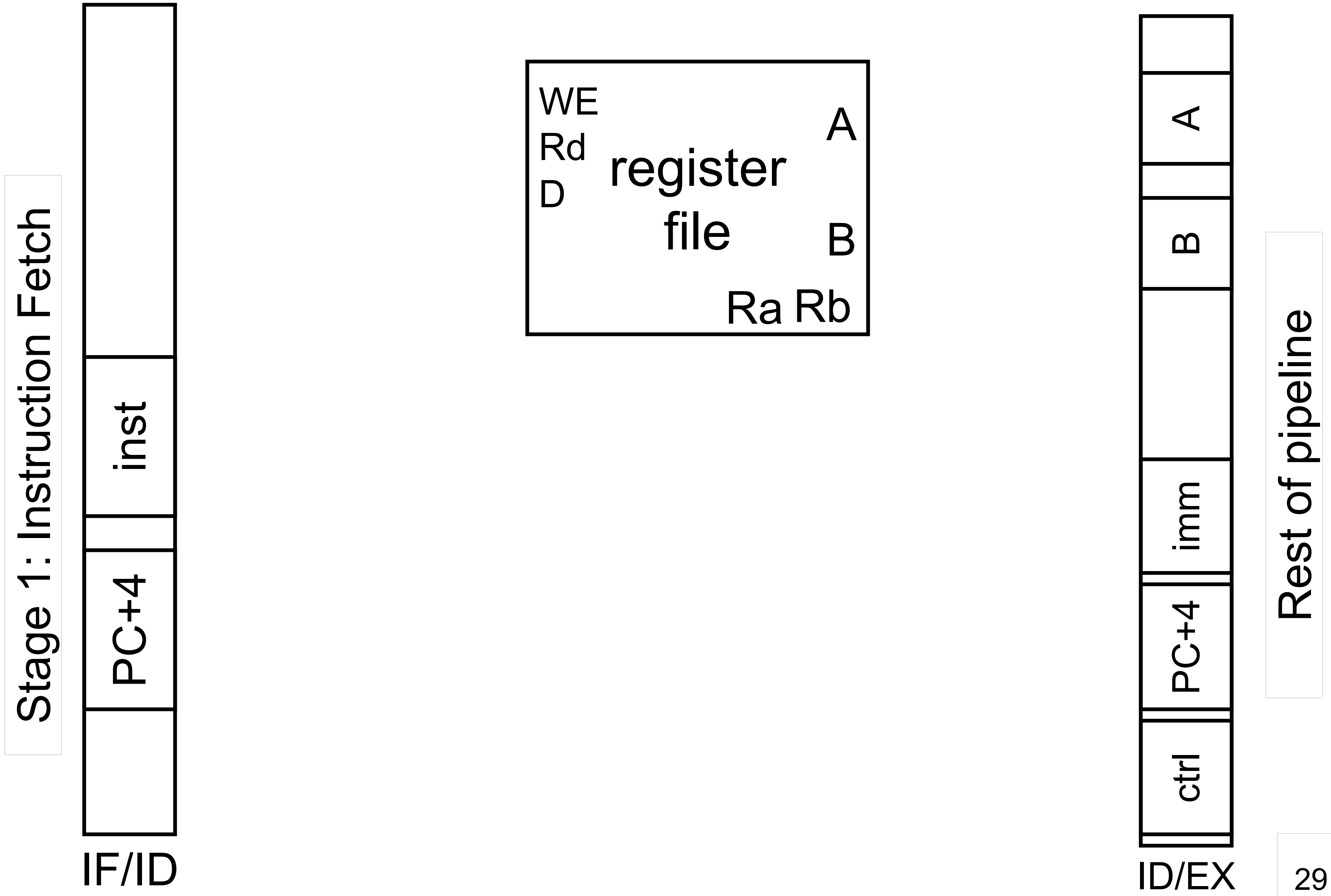
Decode

- 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


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

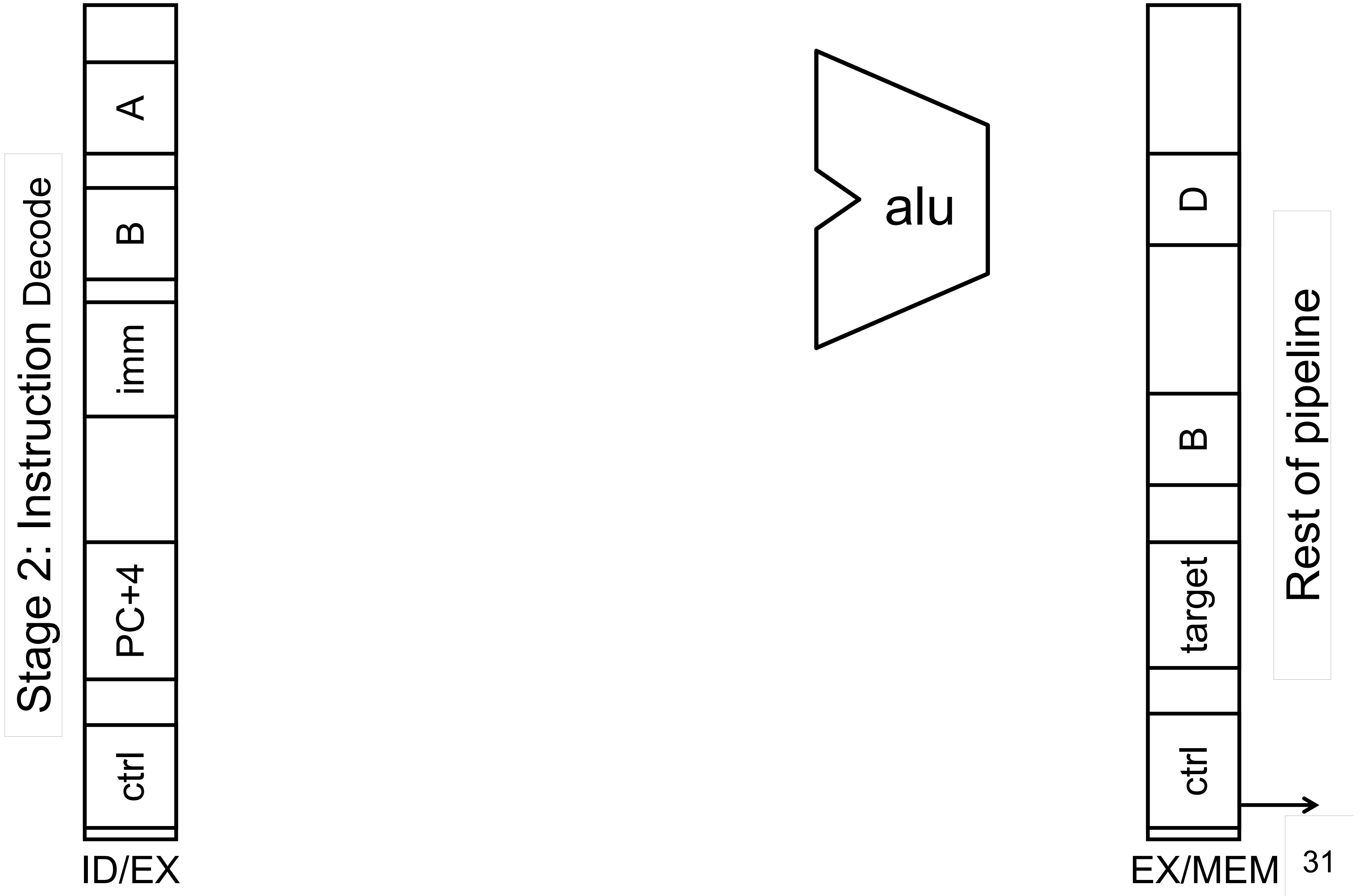
Execute (EX)

- 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)


Stage 2: Instruction Decode
A
B
imm
PC+4
ctrl
ID/EX
alu
D
B
target
ctrl
EX/MEM
Rest of pipeline

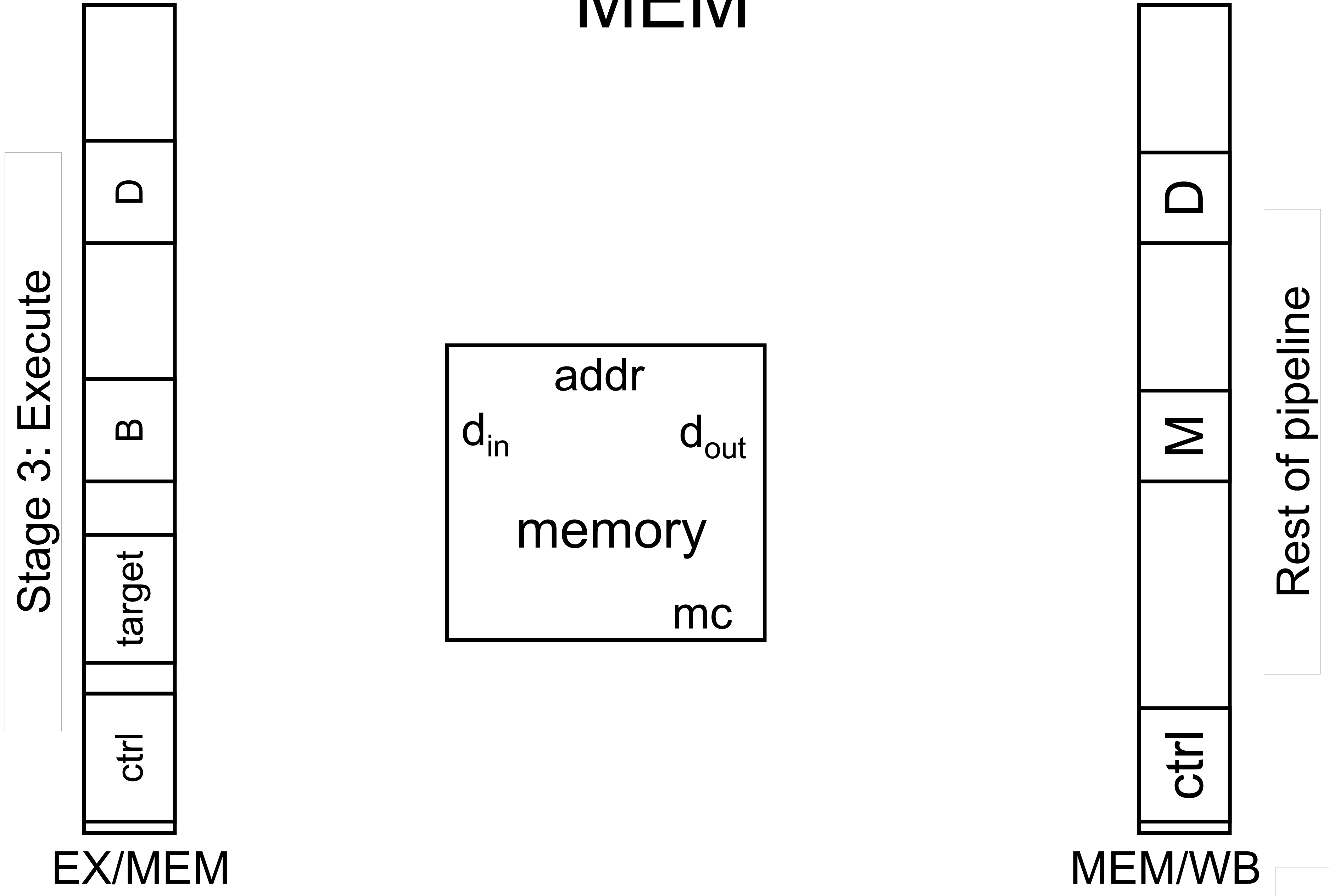
MEM

- 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



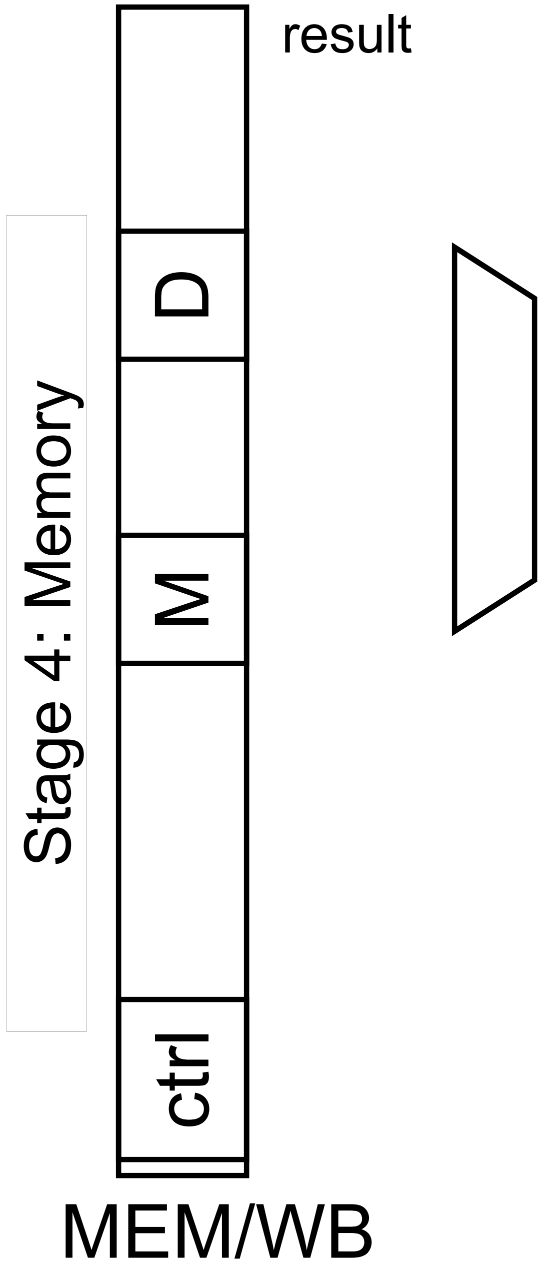
Stage 3: Execute
D
B
target
ctrl
EX/MEM
addr
d_{in}
d_{out}
memory
mc
D
M
ctrl
Rest of pipeline
MEM/WB

WB

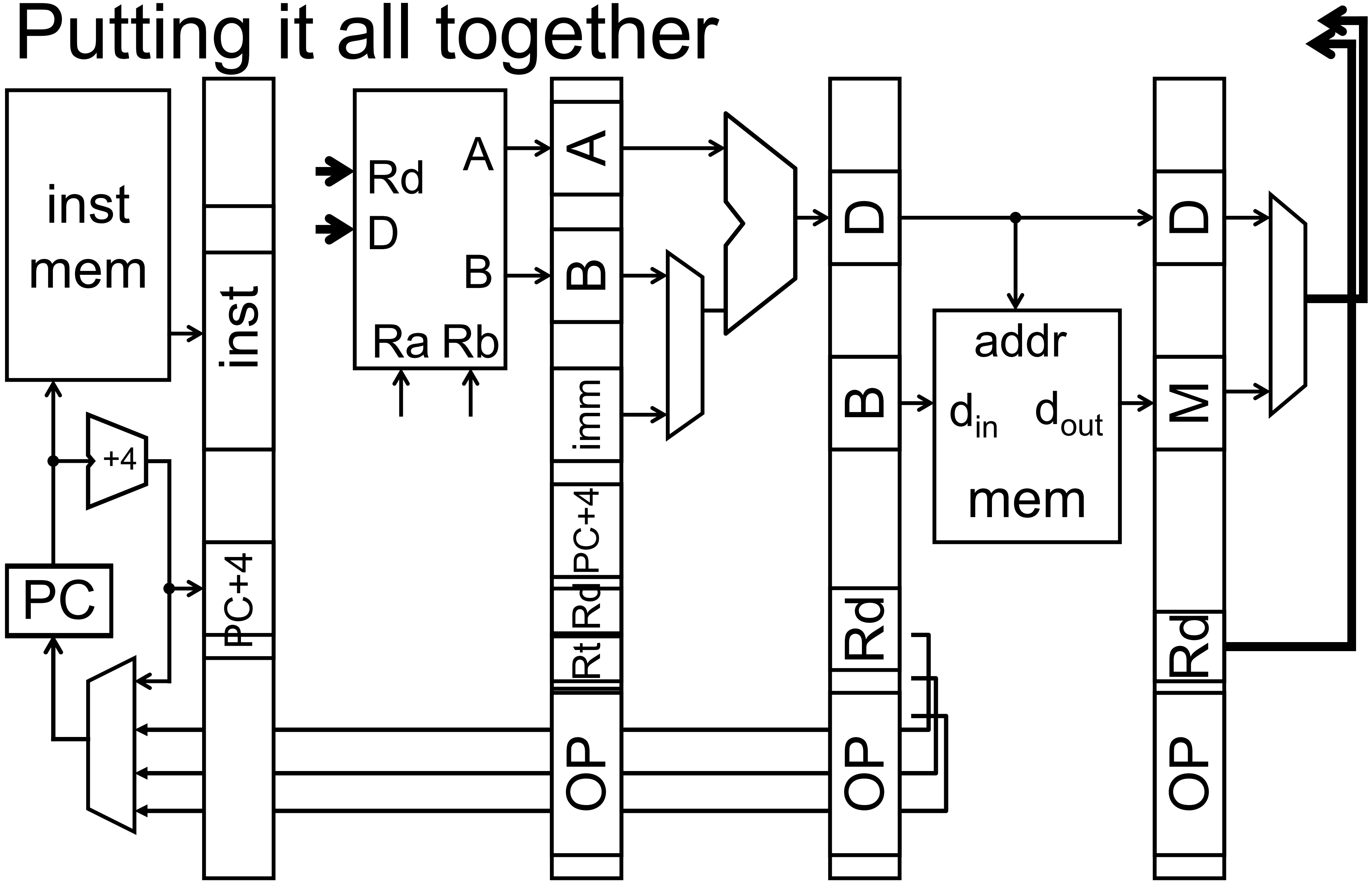
- Stage 5: Write-back

- 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



IF/ID ID/EX EX/MEM MEM/WB

Takeaway

- Pipelining is a powerful technique to mask latencies and increase throughput
  - Logically, instructions execute one at a time
  - Physically, instructions execute in parallel
    - Instruction level parallelism

- Abstraction promotes decoupling
  - Interface (ISA) vs. implementation (Pipeline)

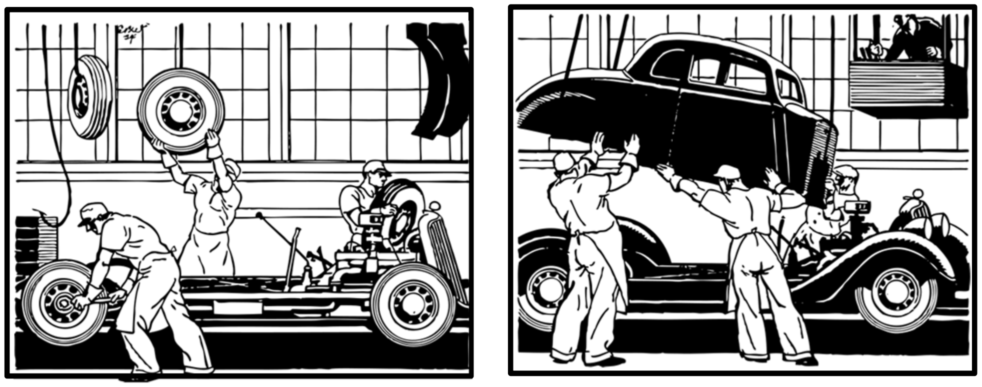
RISC-V is *designed* for pipelining

- 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

Agenda

5-stage Pipeline

- Implementation
- Working Example



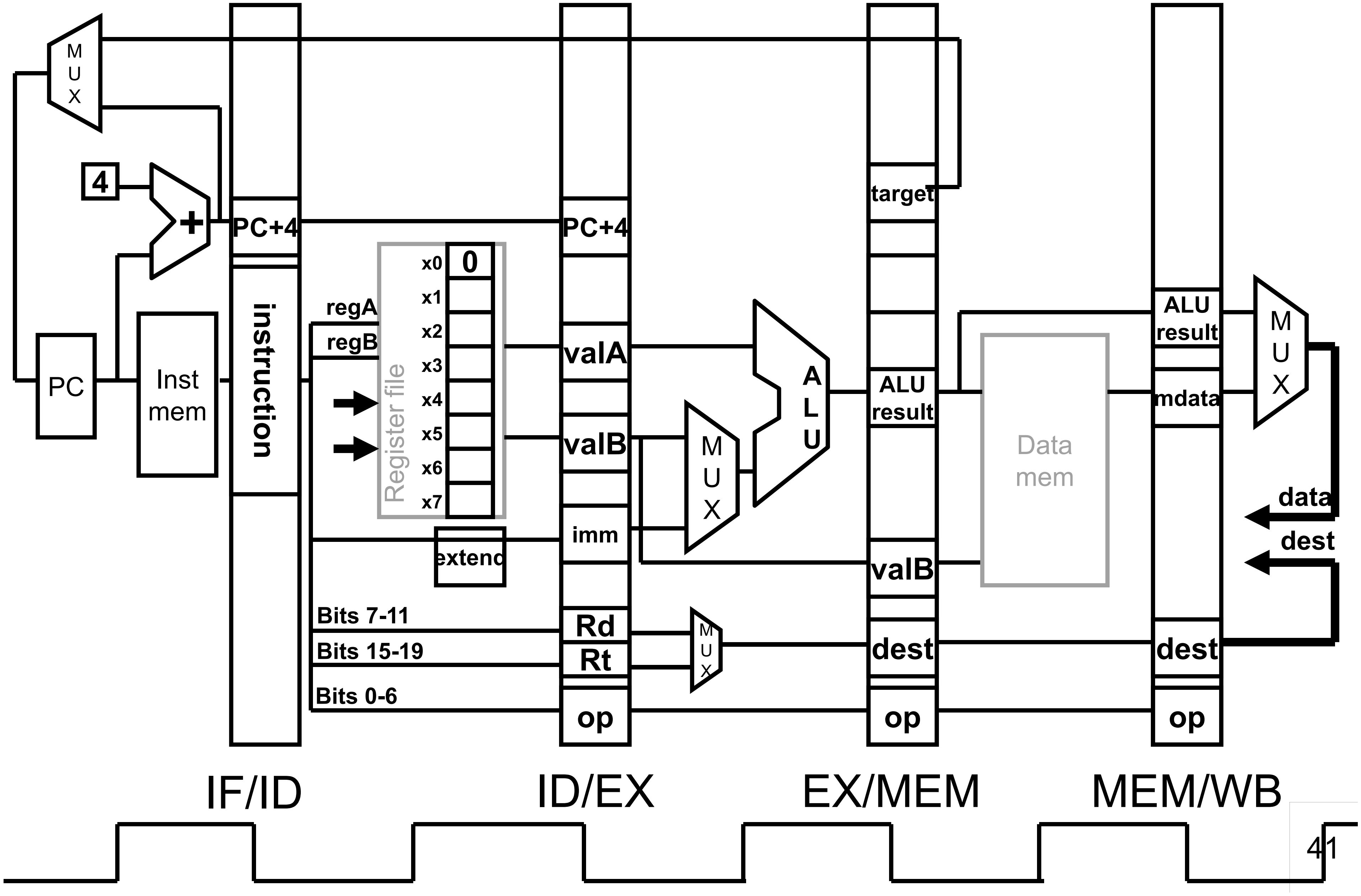
Hazards

- Structural
- Data Hazards
- Control Hazards

Example: Sample Code (Simple)

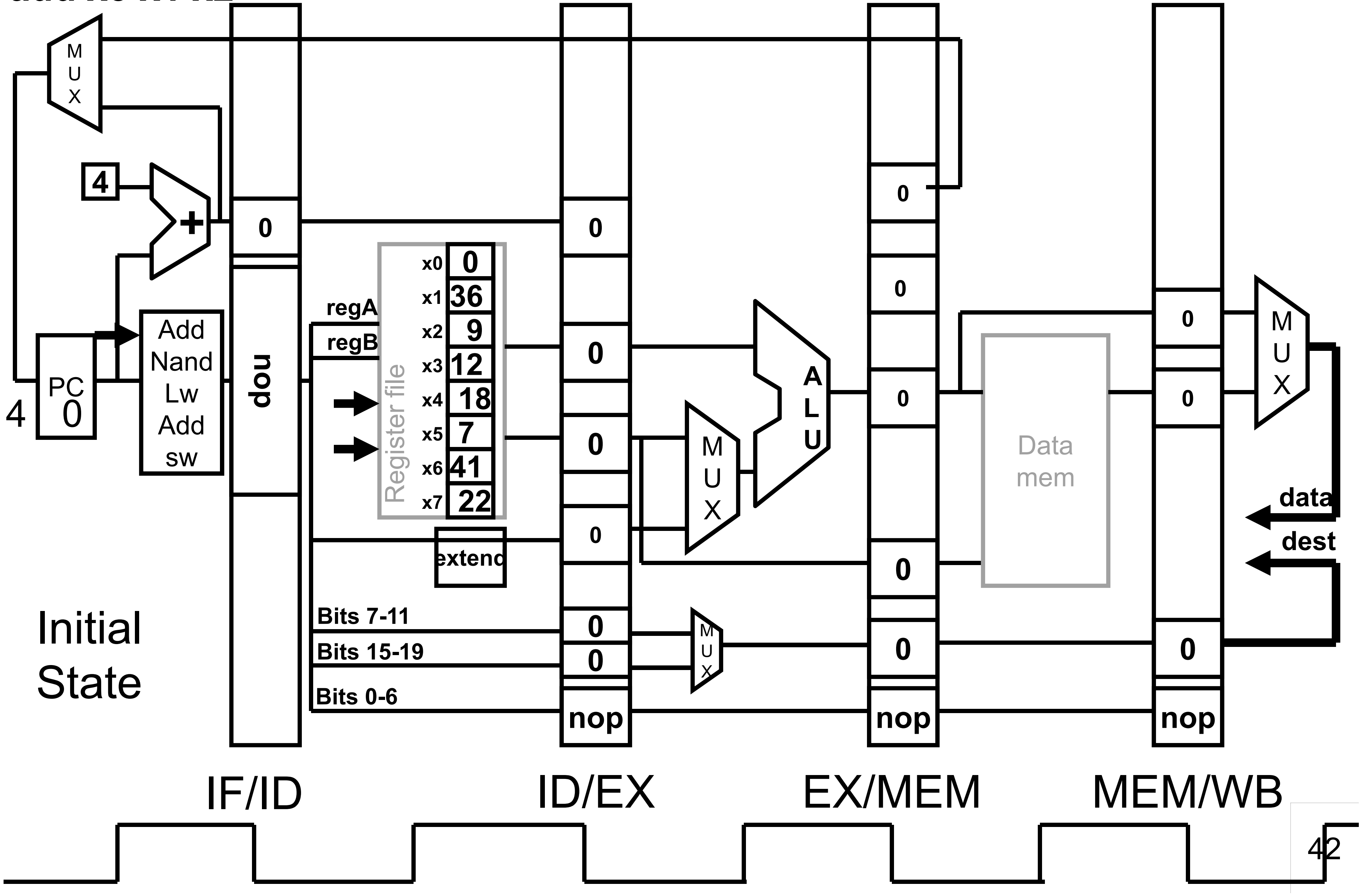
```
add    x3 ← x1, x2
nand   x6 ← x4, x5
lw     x4 ← x2, 20
add    x5 ← x2, x5
sw     x7 → x3, 12
```

Assume 8-register machine


MUX
4
+
PC+4
PC
Inst mem
instruction
regA
regB
Register file
x0
0
x1
x2
x3
x4
x5
x6
x7
extend
Bits 7-11
Bits 15-19
Bits 0-6
PC+4
valA
valB
imm
Rd
Rt
op
MUX
ALU
MUX
target
ALU result
valB
dest
op
Data mem
ALU result
mdata
dest
op
MUX
data
dest
IF/ID
ID/EX
EX/MEM
MEM/WB

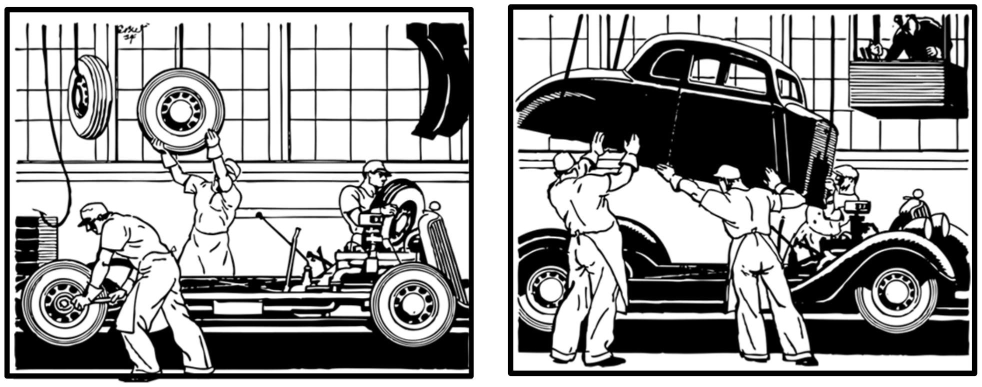
At time 1, Fetch add x3 x1 x2

Example: Start State @ Cycle 0



Agenda

5-stage Pipeline
- Implementation
- Working Example



Hazards
- Structural
- Data Hazards
- Control Hazards

Hazards

Correctness problems associated w/ processor design

1. Structural hazards

   Same resource needed for different purposes at the same time (Possible: ALU, Register File, Memory)

2. Data hazards

   Instruction output needed before it’s available

3. Control hazards

   Next instruction PC unknown at time of Fetch

Dependences and Hazards

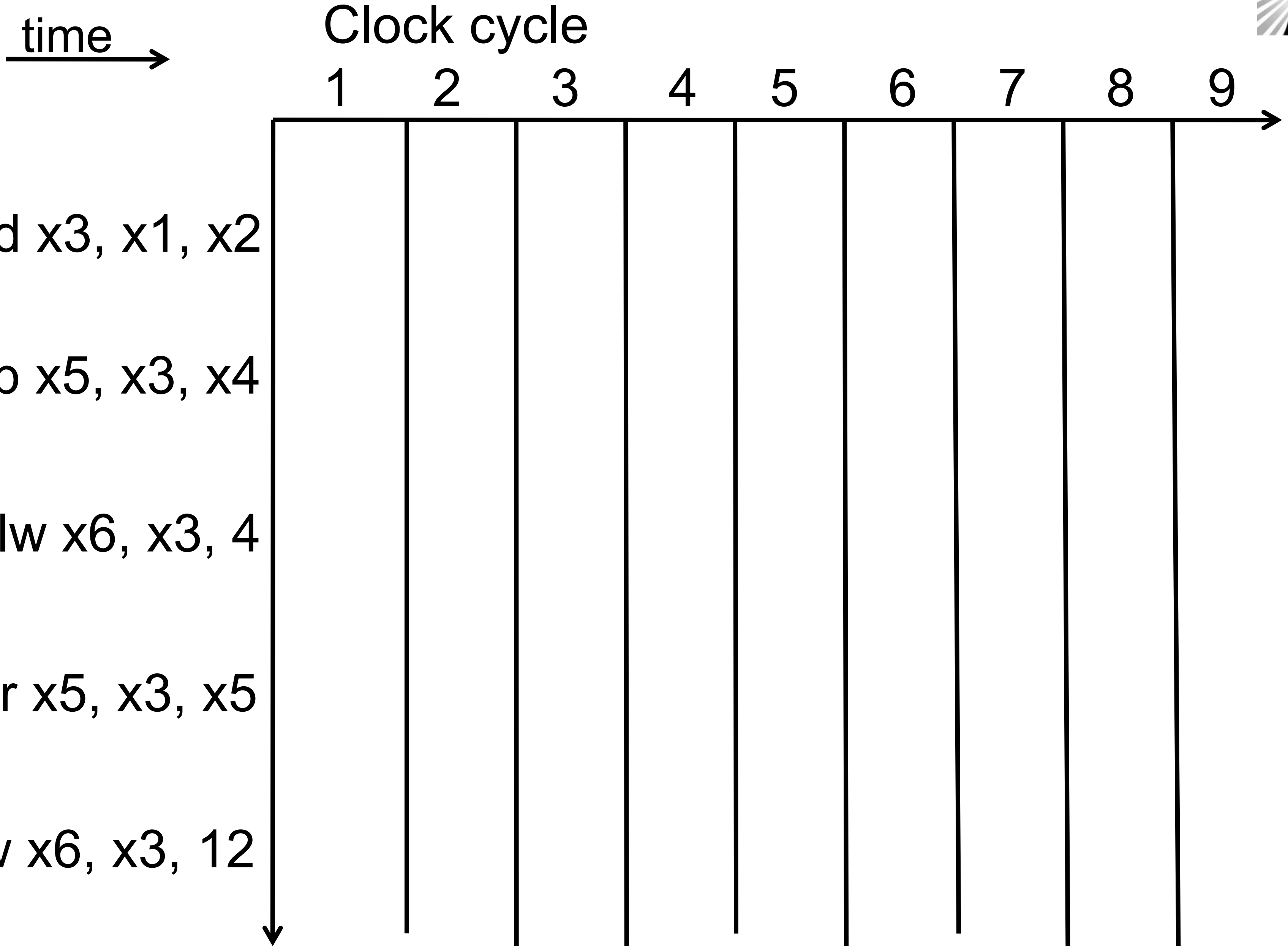
Dependence: relationship between two insns

- **Data**: two insns use same storage location
- **Control**: 1 insn affects whether another executes at all
- *Not a bad thing*, programs would be boring otherwise
- Enforced by making older insn go before younger one
  - Happens naturally in single-/multi-cycle designs
  - But not in a pipeline

Hazard: dependence & possibility of wrong insn order

- Effects of wrong insn order cannot be externally visible
- *Hazards are a bad thing*: most solutions either complicate the hardware or reduce performance

Where are the Data Hazards?



```
add x3, x1, x2

sub x5, x3, x4

lw x6, x3, 4

or x5, x3, x5

sw x6, x3, 12
```

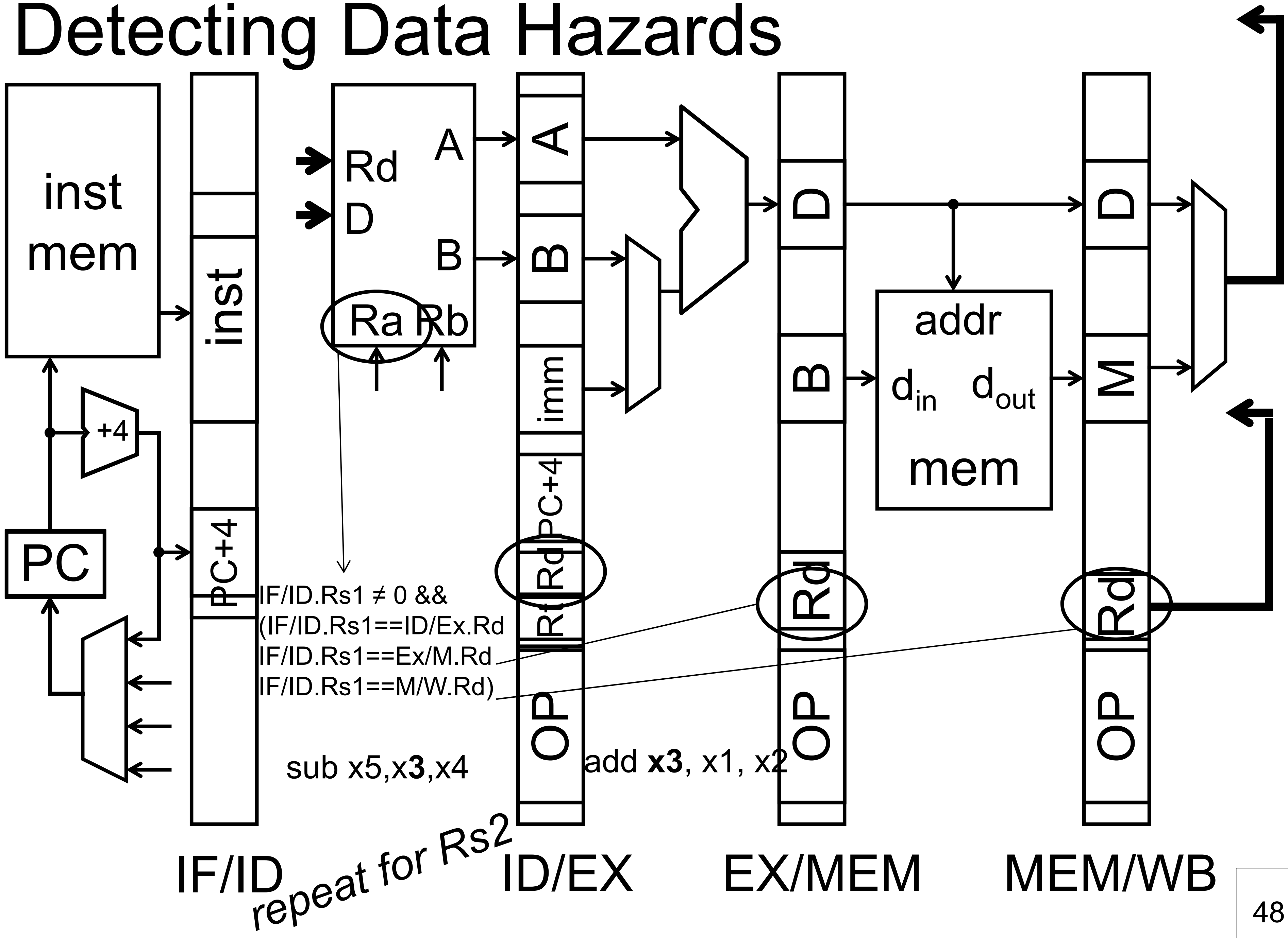
Data Hazards

- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- next instructions may read values about to be written

i.e. add x3, x1, x2
    sub x5, x3, x4

How to detect?

Detecting Data Hazards



IF/ID.Rs1 ≠ 0 &&
(IF/ID.Rs1==ID/Ex.Rd
IF/ID.Rs1==Ex/M.Rd
IF/ID.Rs1==M/W.Rd)

sub x5,x**3**,x4

add **x3**, x1, x2

repeat for Rs2

Data Hazards

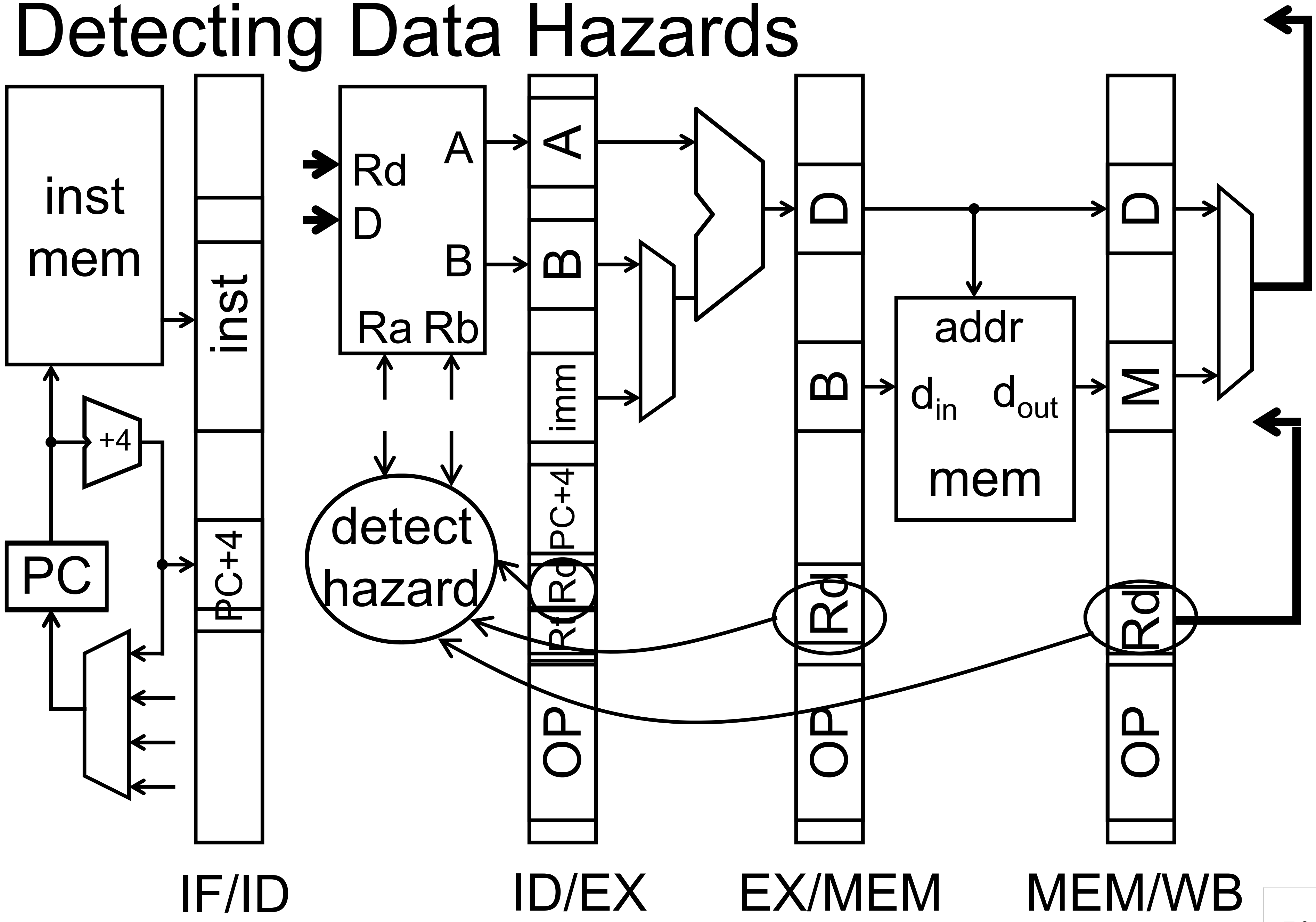
Data Hazards

- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- next instructions may read values about to be written

How to detect? Logic in ID stage:

```
    stall = (IF/ID.Rs1 != 0 &&
              (IF/ID.Rs1 == ID/EX.Rd ||
               IF/ID.Rs1 == EX/M.Rd ||
               IF/ID.Rs1 == M/WB.Rd))
              || (same for Rs2)
```

Detecting Data Hazards



Takeaway

Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

Next Goal

What to do if data hazard detected?

Possible Responses to Data Hazards

1. Do Nothing
   - Change the ISA to match implementation
   - "Hey compiler: don't create code w/data hazards!"

     (*We can do better than this)*
2. Stall
   - Pause current and subsequent instructions till safe
3. Forward/bypass
   - Forward data value to where it is needed

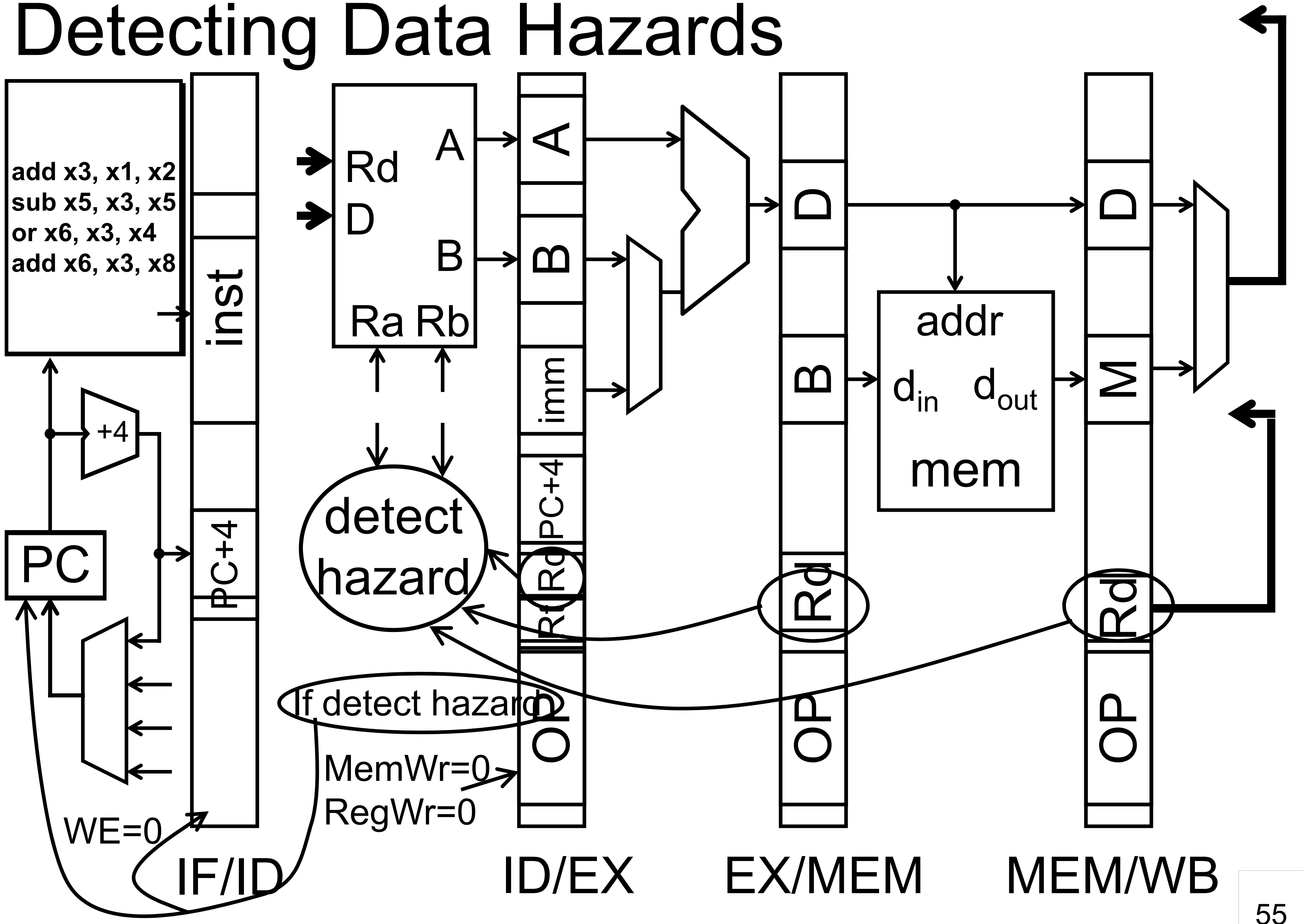
     (Only works if value actually exists already)

Stalling

How to stall an instruction in ID stage

- prevent IF/ID pipeline register update
  - stalls the ID stage instruction
- convert ID stage instr into nop for later stages
  - innocuous “bubble” passes through pipeline
- prevent PC update
  - stalls the next (IF stage) instruction

Detecting Data Hazards



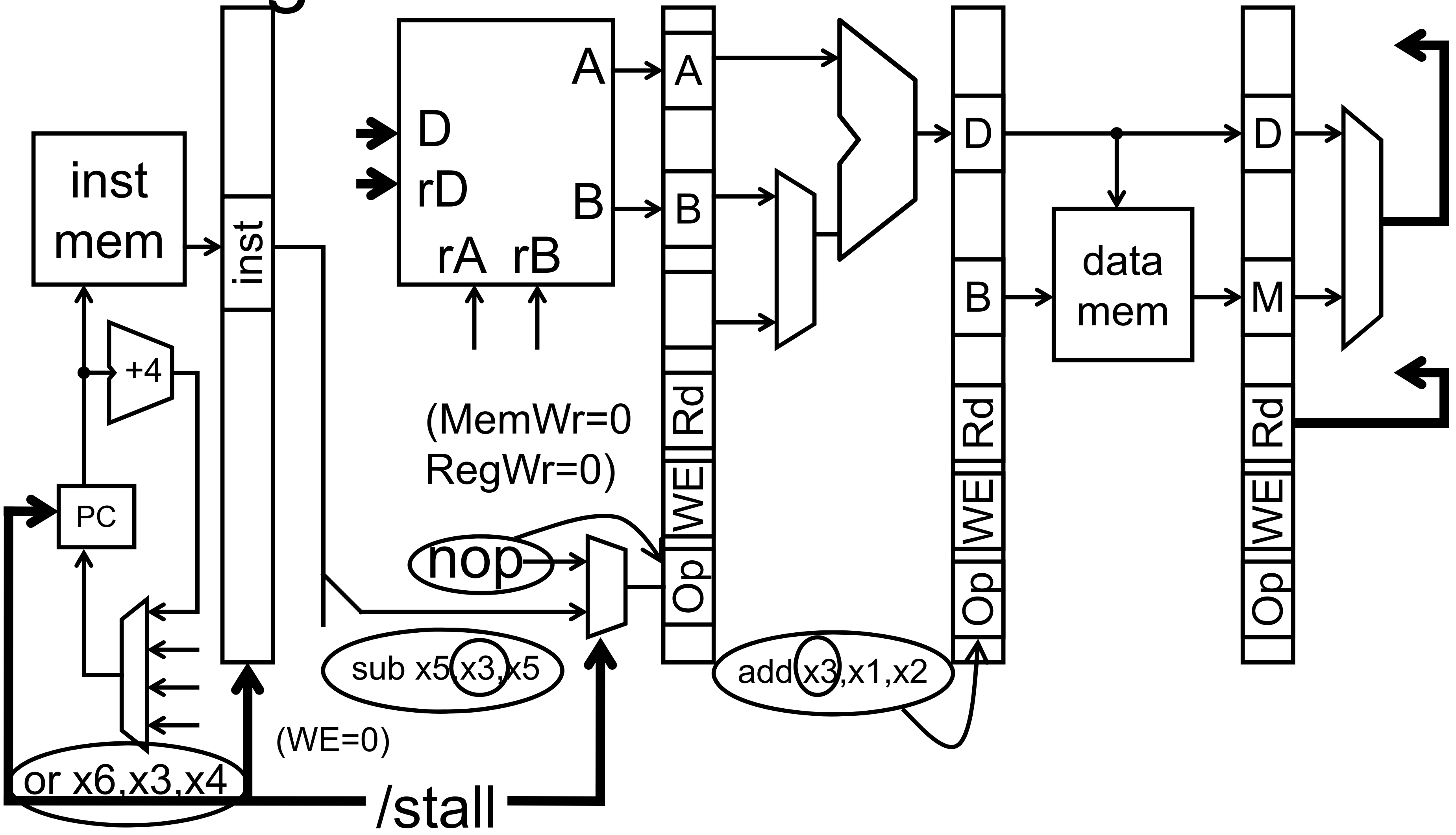
Stalling

time →

Clock cycle

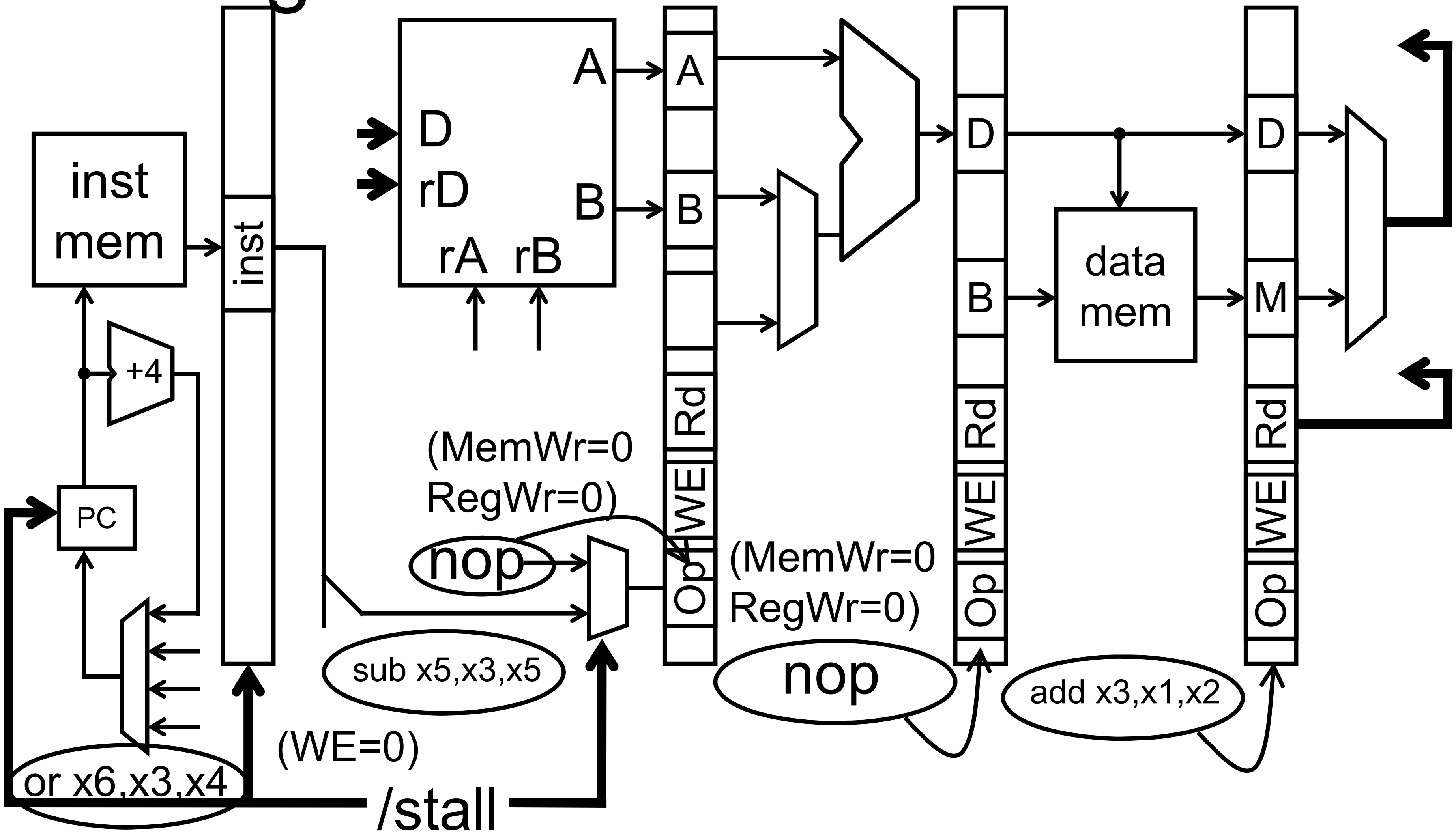
	1	2	3	4	5	6	7	8
add x3, x1, x2								
sub x5, x3, x5								
or x6, x3, x4								
add x6, x3, x8								

Stalling



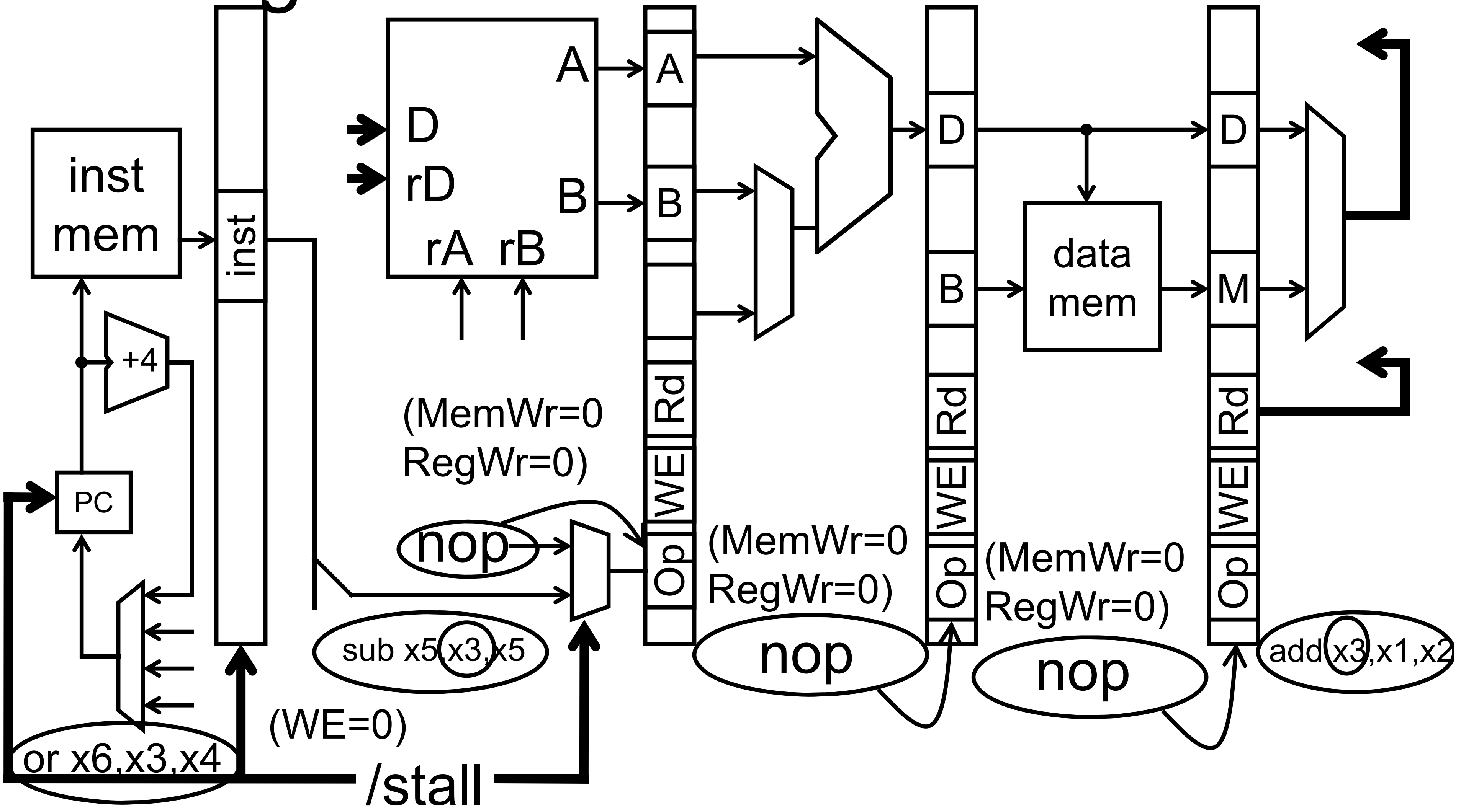
NOP = If(IF/ID.rA ≠ 0 &&
(IF/ID.rA==ID/Ex.Rd ← STALL CONDITION MET
IF/ID.rA==Ex/M.Rd
IF/ID.rA==M/W.Rd))

Stalling



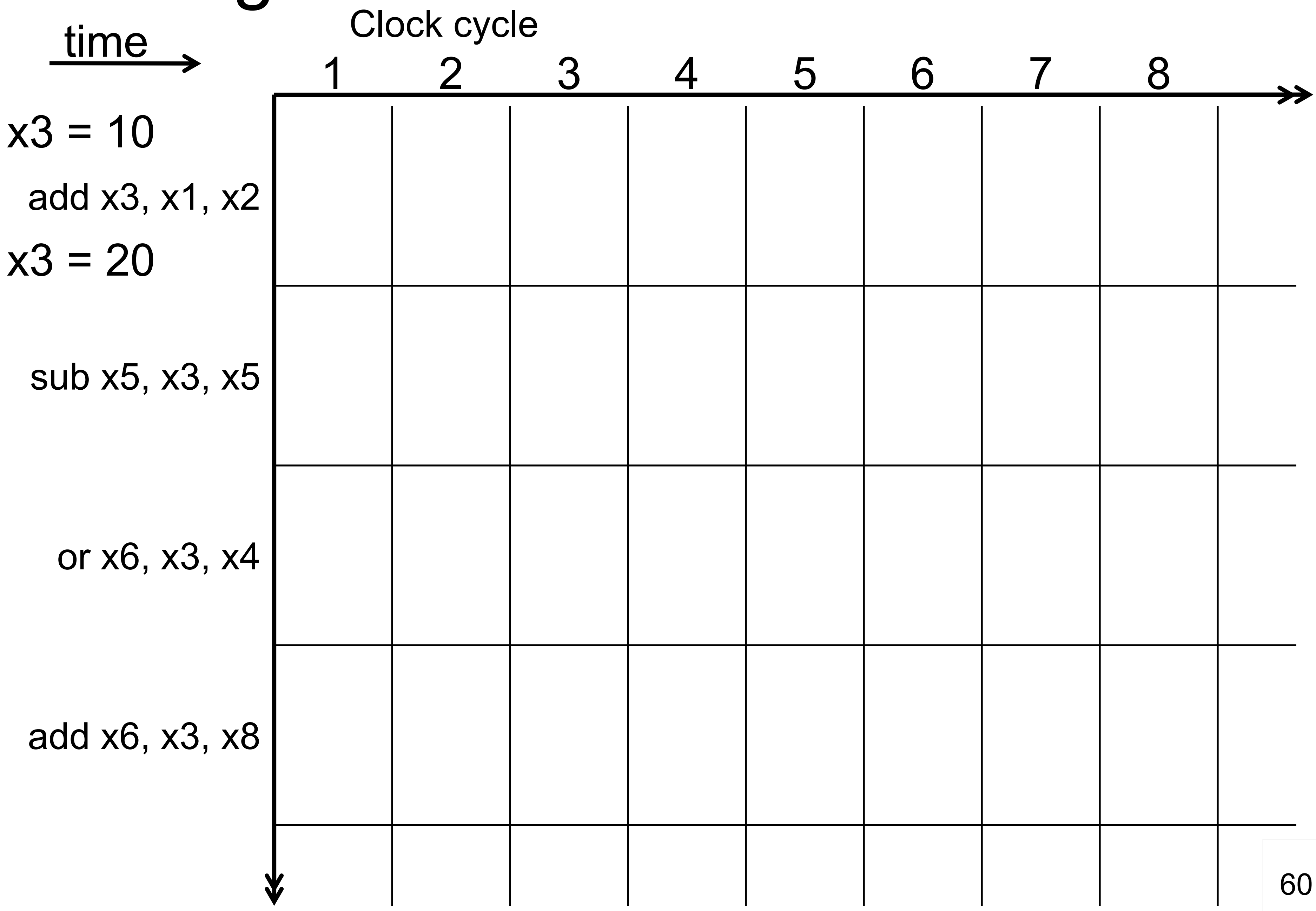
NOP = If(IF/ID.rA ≠ 0 &&
(IF/ID.rA==ID/Ex.Rd
IF/ID.rA==Ex/M.Rd ← STALL CONDITION MET
IF/ID.rA==M/W.Rd))

Stalling



NOP = If(IF/ID.rA ≠ 0 &&
(IF/ID.rA==ID/Ex.Rd
IF/ID.rA==Ex/M.Rd
IF/ID.rA==M/W.Rd)) ← STALL CONDITION MET

Stalling



time
Clock cycle
1
2
3
4
5
6
7
8
x3 = 10
add x3, x1, x2
x3 = 20
sub x5, x3, x5
or x6, x3, x4
add x6, x3, x8

Stalling

How to stall an instruction in ID stage

- prevent IF/ID pipeline register update
  - stalls the ID stage instruction
- convert ID stage instr into nop for later stages
  - innocuous “bubble” passes through pipeline
- prevent PC update
  - stalls the next (IF stage) instruction

Takeaway

Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

Stalling, preventing a dependent instruction from advancing, is one way to resolve data hazards.

Stalling introduces NOPs (“bubbles”) into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file. *Bubbles in pipeline significantly decrease performance.

Possible Responses to Data Hazards

1. Do Nothing
    - Change the ISA to match implementation
    - "Compiler: don't create code with data hazards!"

      (*Nice try, we can do better than this)*
2. Stall
    - Pause current and subsequent instructions till safe
3. Forward/bypass
    - Forward data value to where it is needed

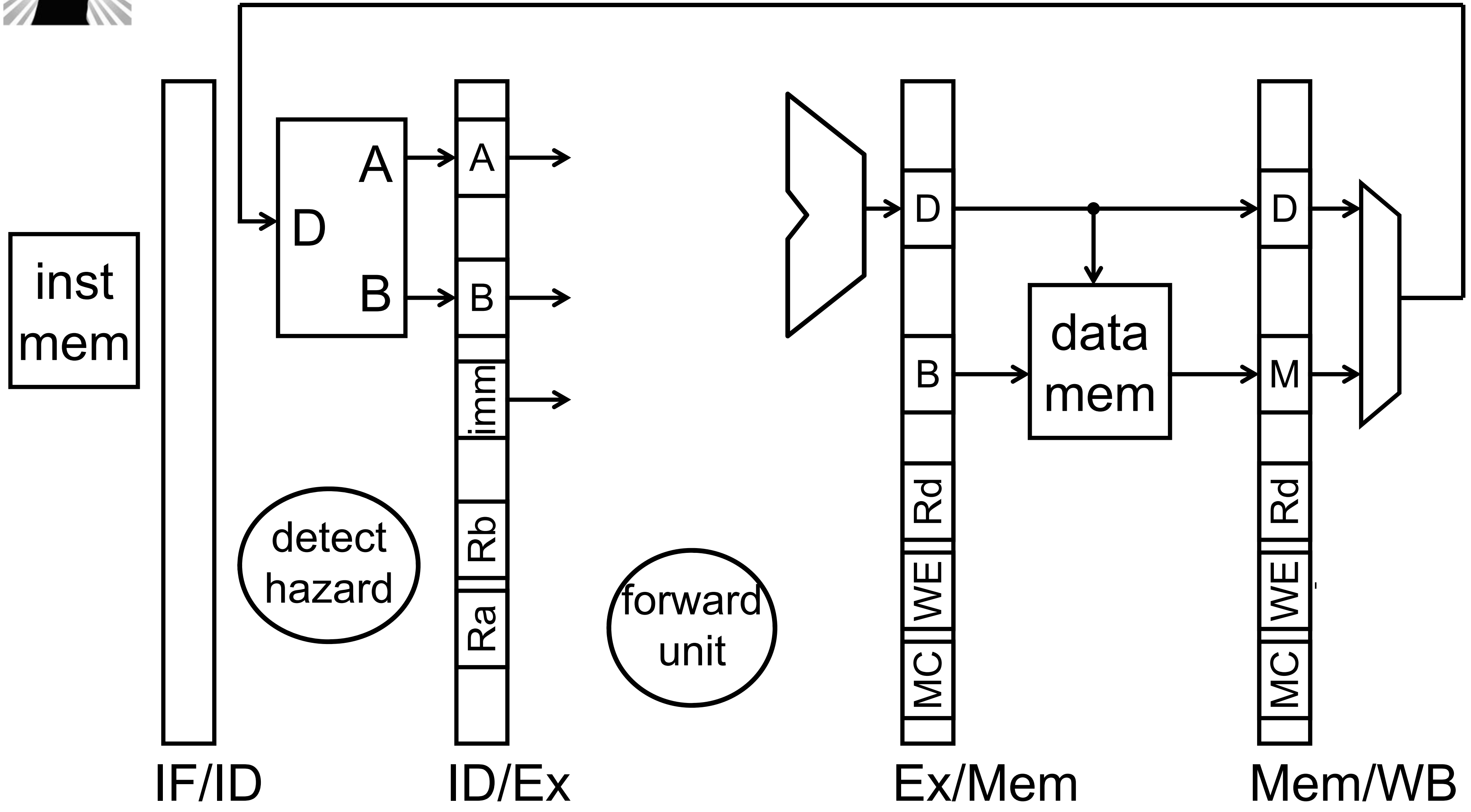
      (Only works if value actually exists already)

Forwarding

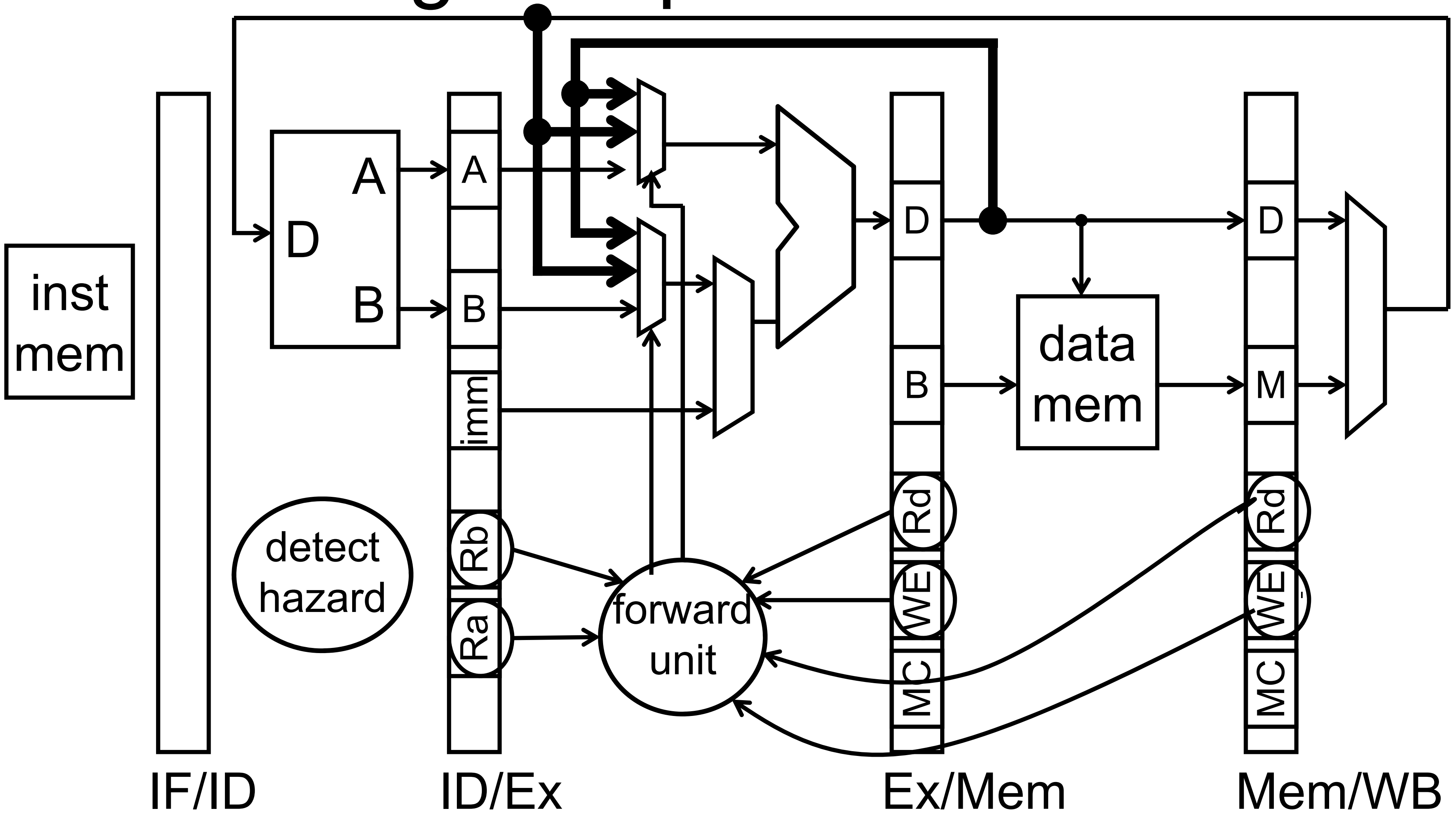
- Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register).

- Three types of forwarding/bypass
  - Forwarding from Ex/Mem registers to Ex stage (M$\rightarrow$Ex)
  - Forwarding from Mem/WB register to Ex stage (W$\rightarrow$Ex)
  - RegisterFile Bypass

Add the Forwarding Datapath



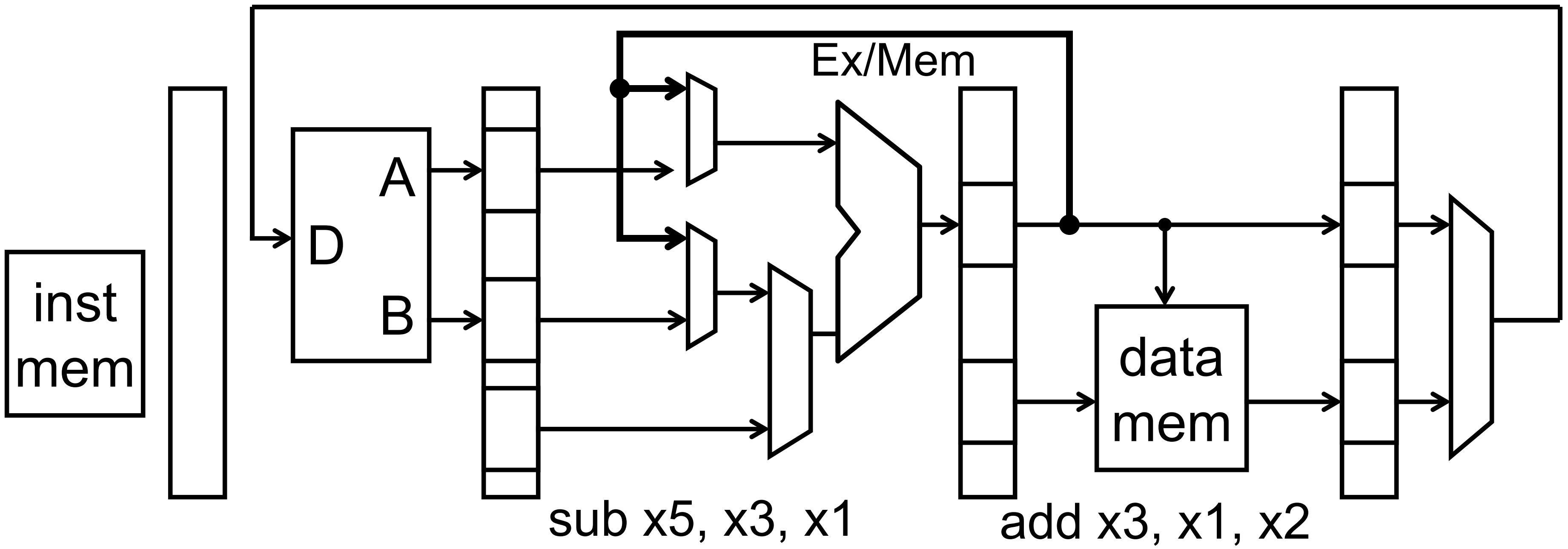
Forwarding Datapath

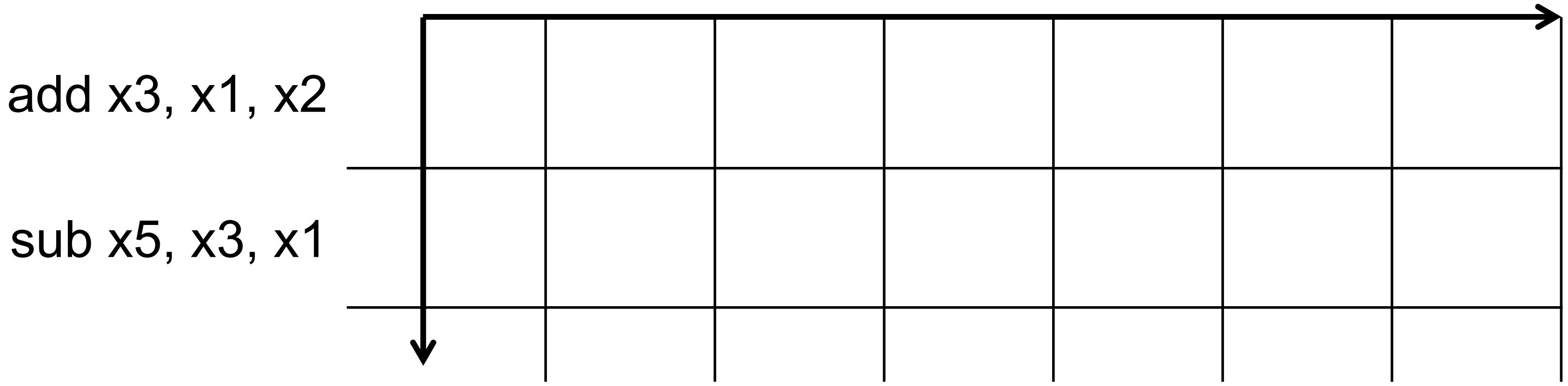


Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass

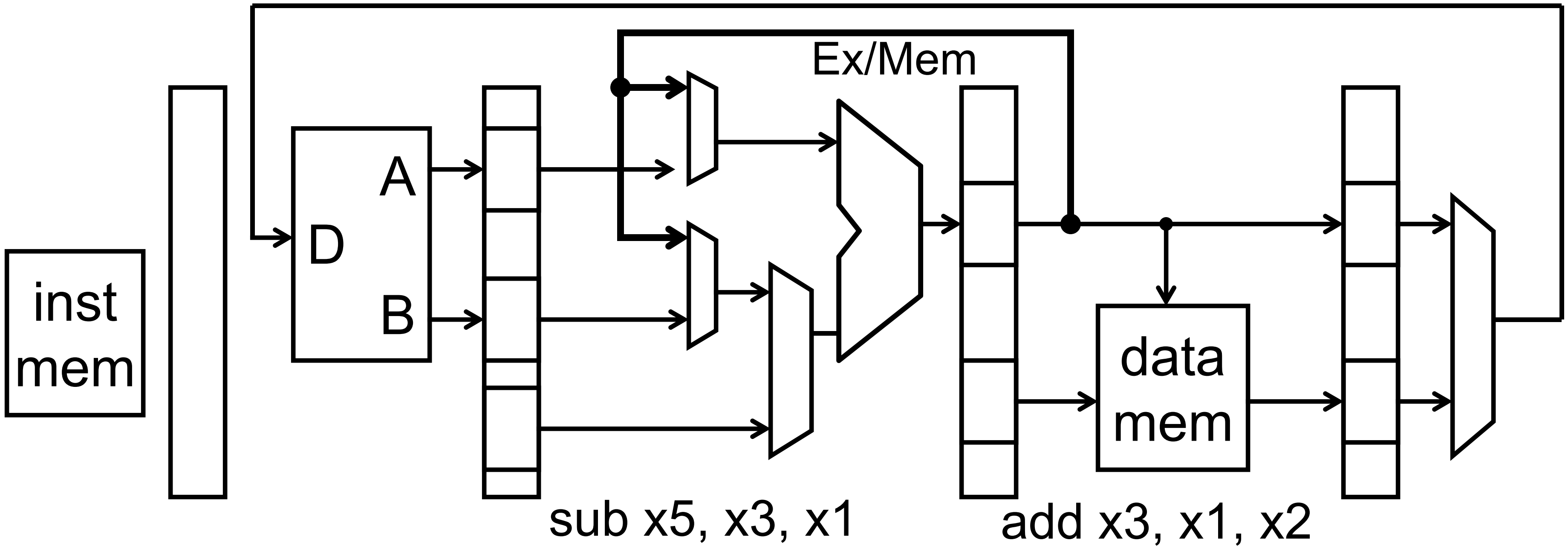
Forwarding Datapath 1: Ex/MEM → EX





Problem: EX needs ALU result that is in MEM stage
Solution: add a bypass from EX/MEM.D to start of EX

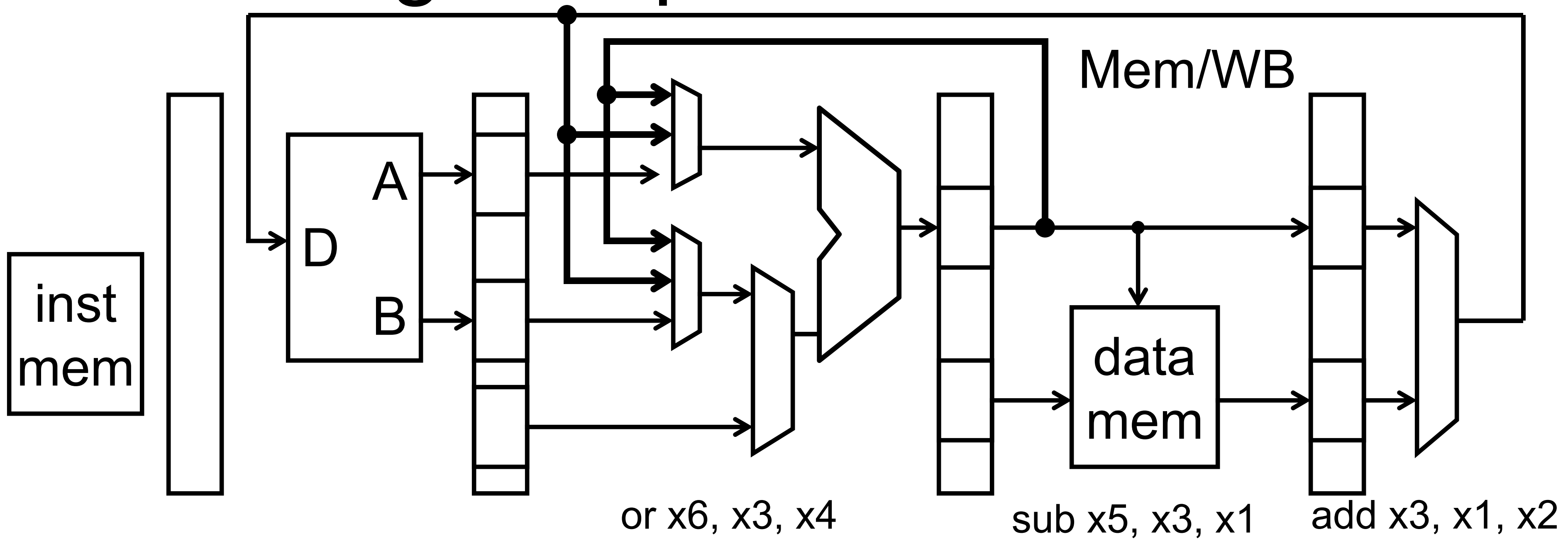
Forwarding Datapath 1: Ex/MEM → EX



Detection Logic in Ex Stage:

forward = (Ex/M.WE && EX/M.Rd != 0 &&
ID/Ex.Rs1 == Ex/M.Rd)
|| (same for Rs1)

Forwarding Datapath 2: Mem/WB→ EX



add x3, x1, x2
sub x5, x3, x1
or x6, x3, x4

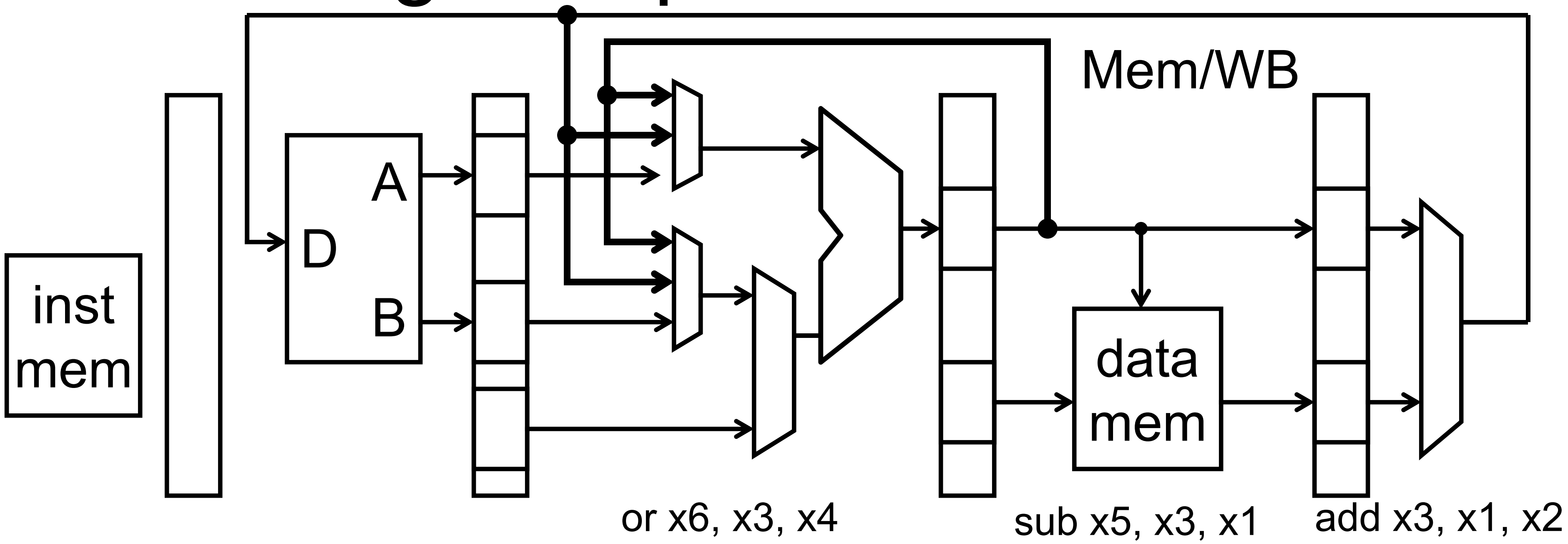
Problem: EX needs value being written by WB

Problem: EX needs value being written by WB

Solution: Add bypass from WB final value to start of EX

Solution: Add bypass from WB final value to start of EX

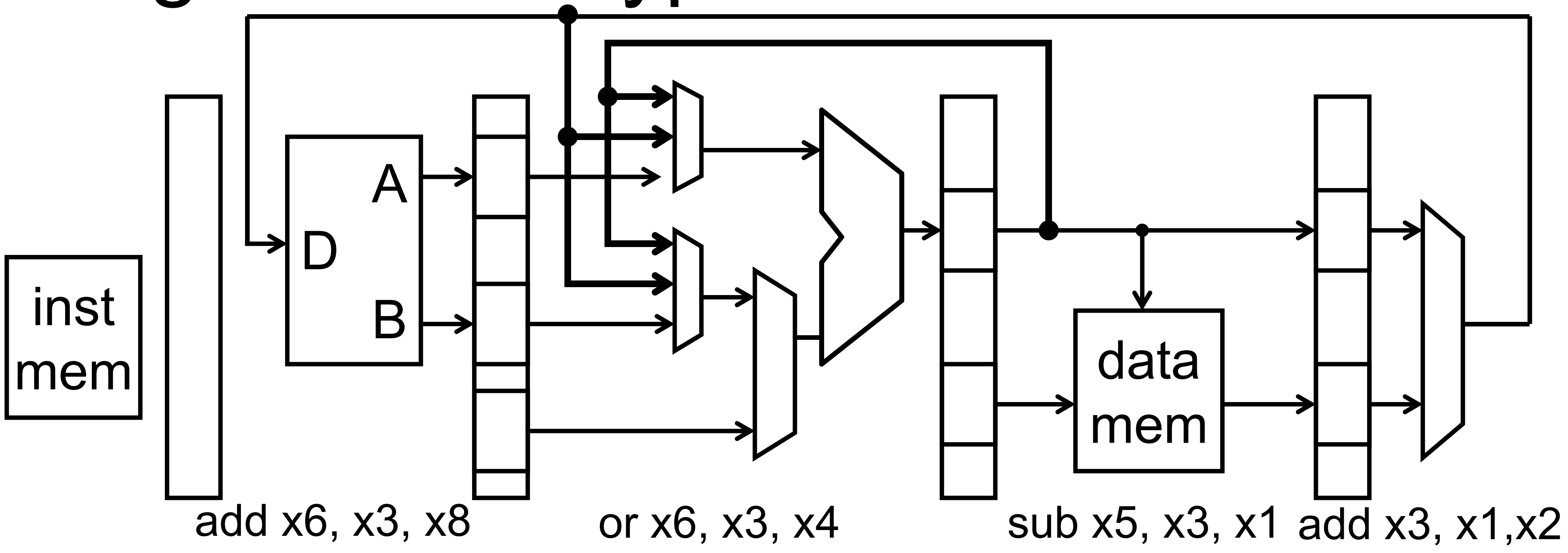
Forwarding Datapath 2: Mem/WB→ EX



Detection Logic:

forward = (M/WB.WE && M/WB.Rd != 0 &&
ID/Ex.Rs1 == M/WB.Rd &&
not (ID/Ex.WE && Ex/M.Rd != 0 &&
ID/Ex.Rs1 == Ex/M.Rd)
|| (same for Rs2)

Register File Bypass

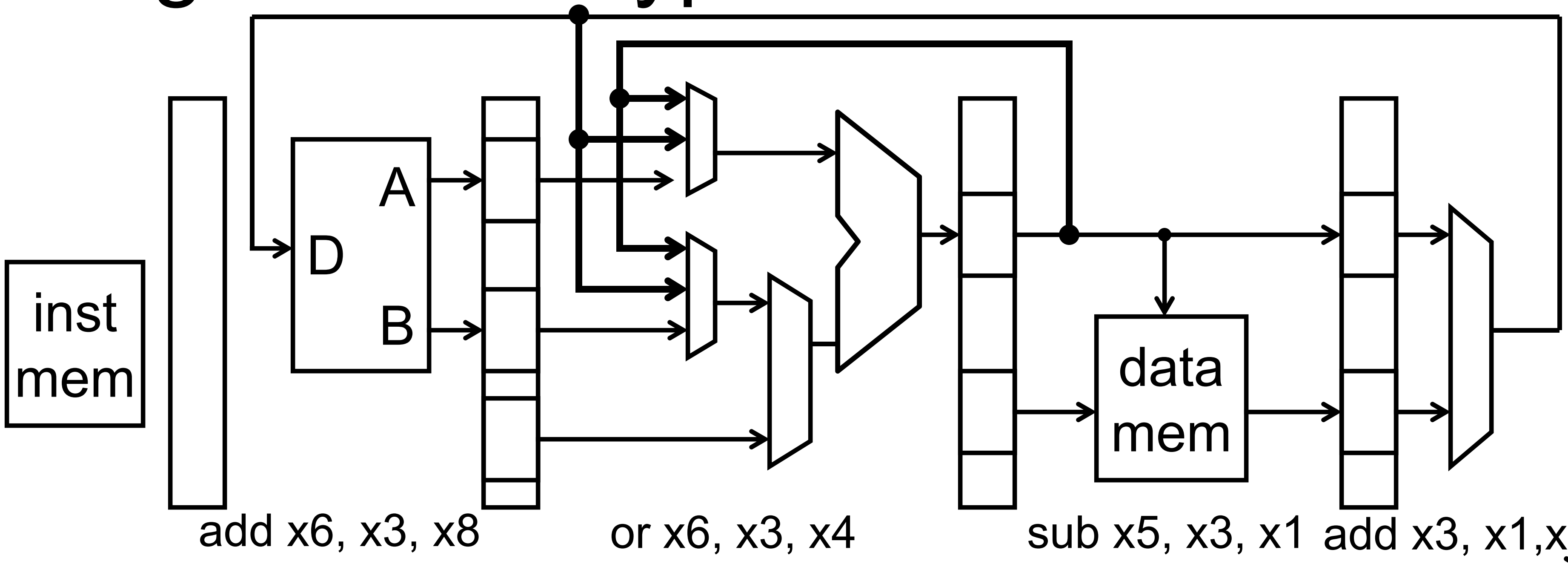


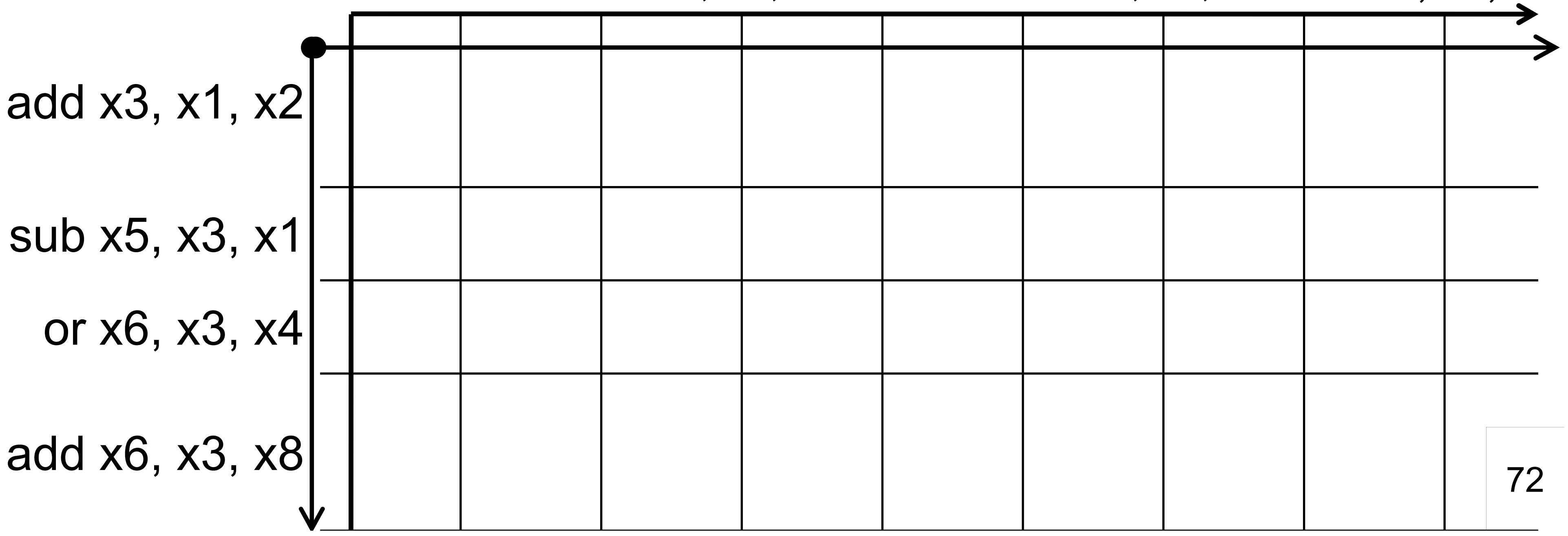
Problem: Reading a value that is currently being written

Solution: just negate register file clock

- writes happen at end of first half of each clock cycle
- reads happen during second half of each clock cycle

Register File Bypass

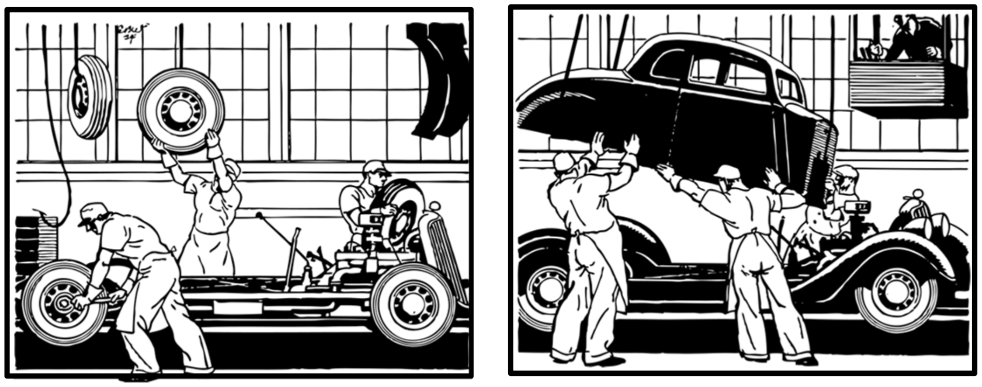




Agenda

5-stage Pipeline

- Implementation
- Working Example



Hazards

- Structural
- Data Hazards
- Control Hazards

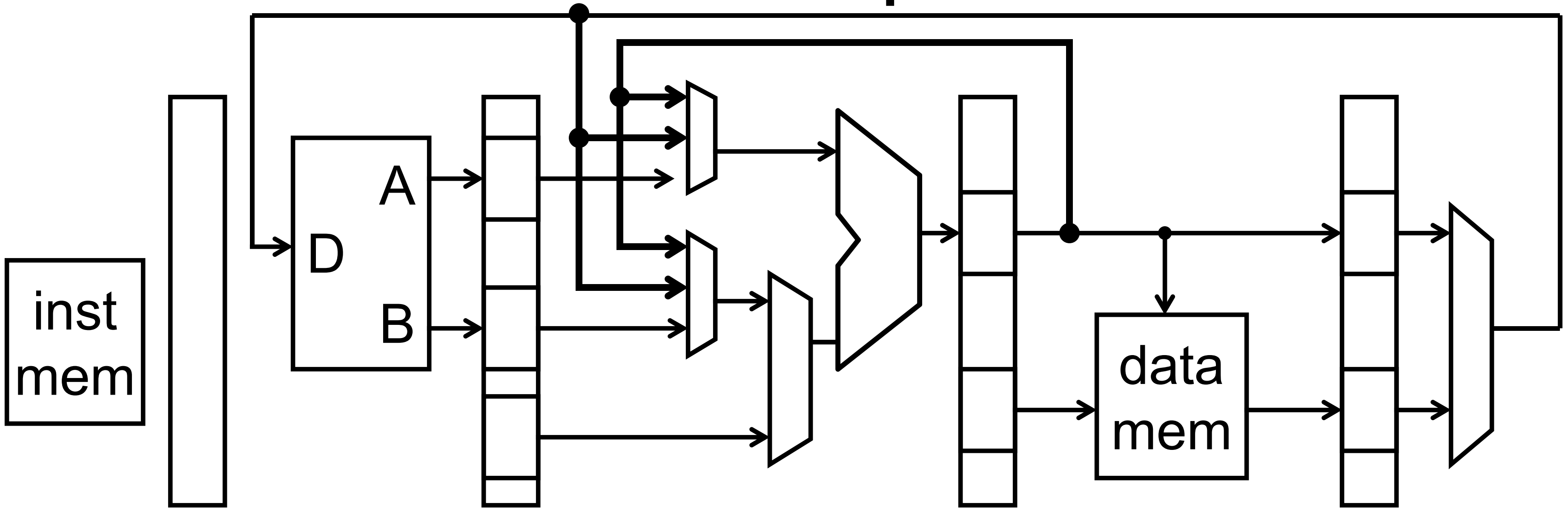
Forwarding Example 2

time →

Clock cycle

	1	2	3	4	5	6	7	8	
add x3, x1, x2									
sub x5, x3, x5									
lw x6, x3, 4									
or x6, x3, x4									
sw x6, x3, 12									

Load-Use Hazard Explained



or x6, x3, **x4** lw **x4**, x8, 20

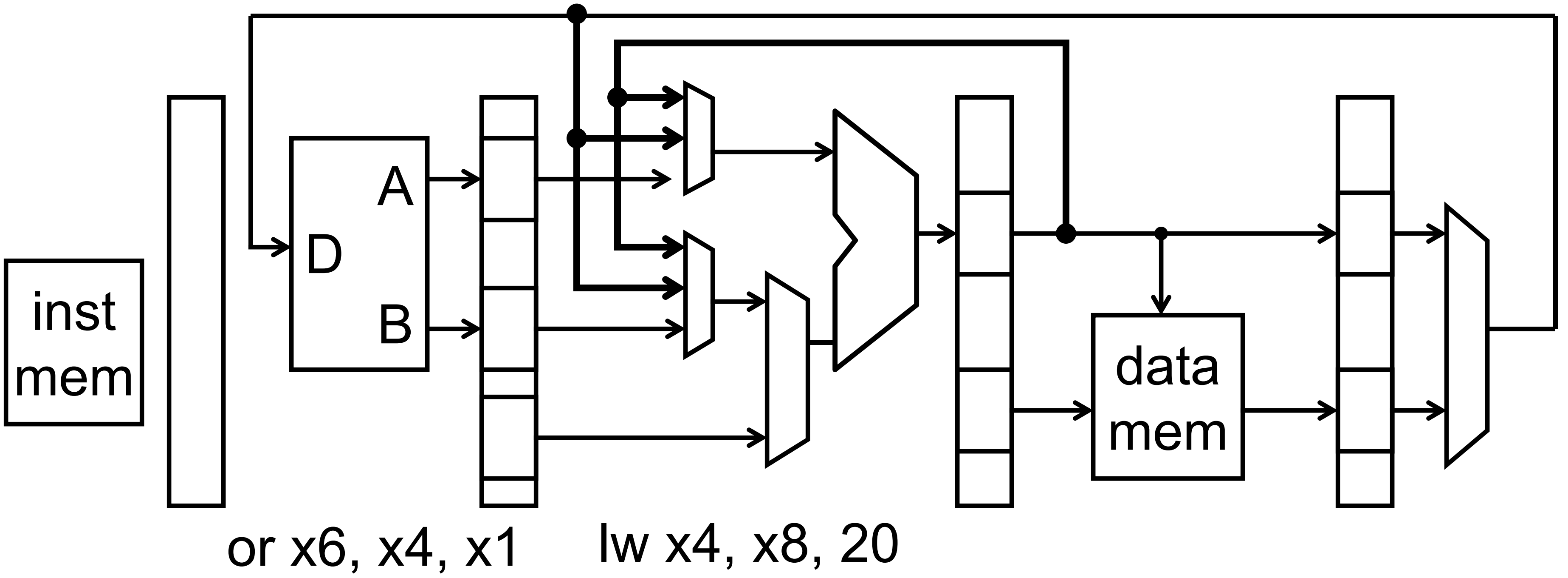
Data dependency after a load instruction:

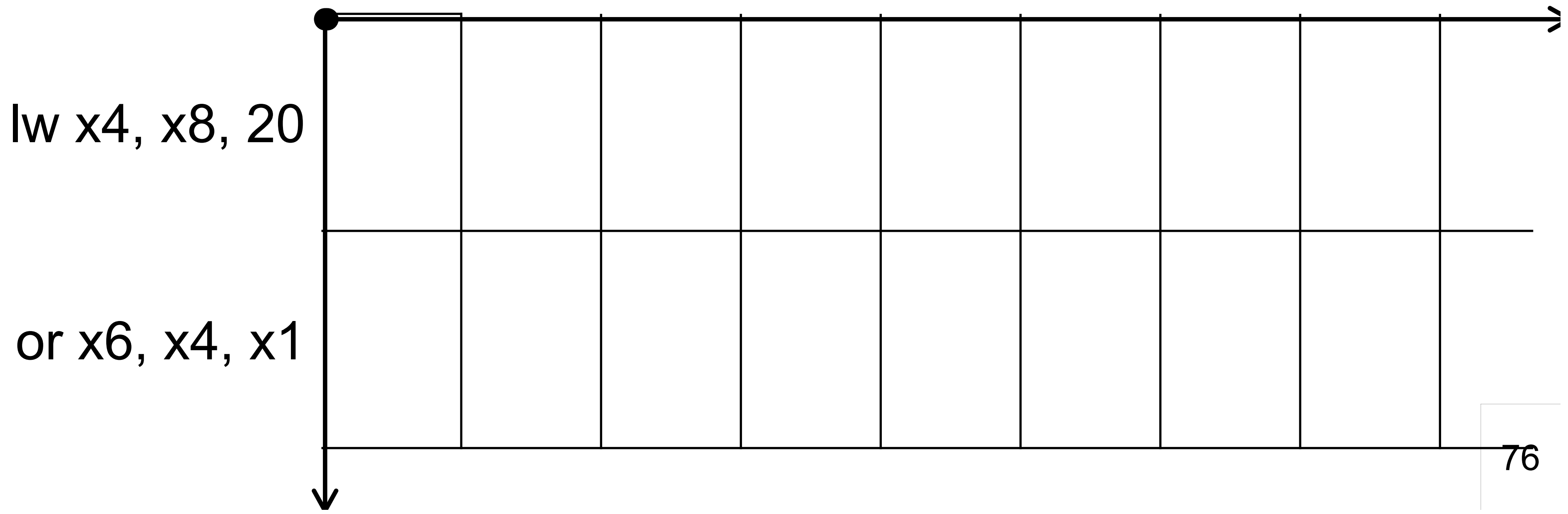
- Value not available until ***after*** the M stage

→Next instruction cannot proceed if dependent

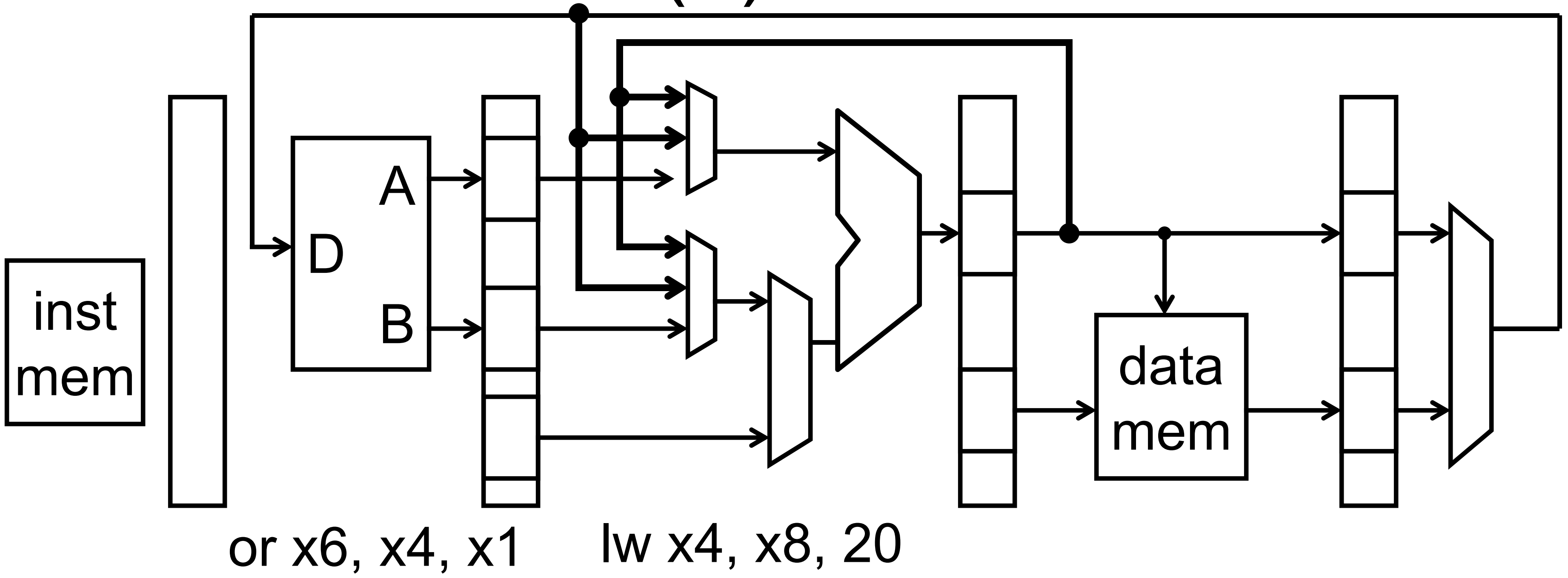
THE KILLER HAZARD

Load-Use Stall





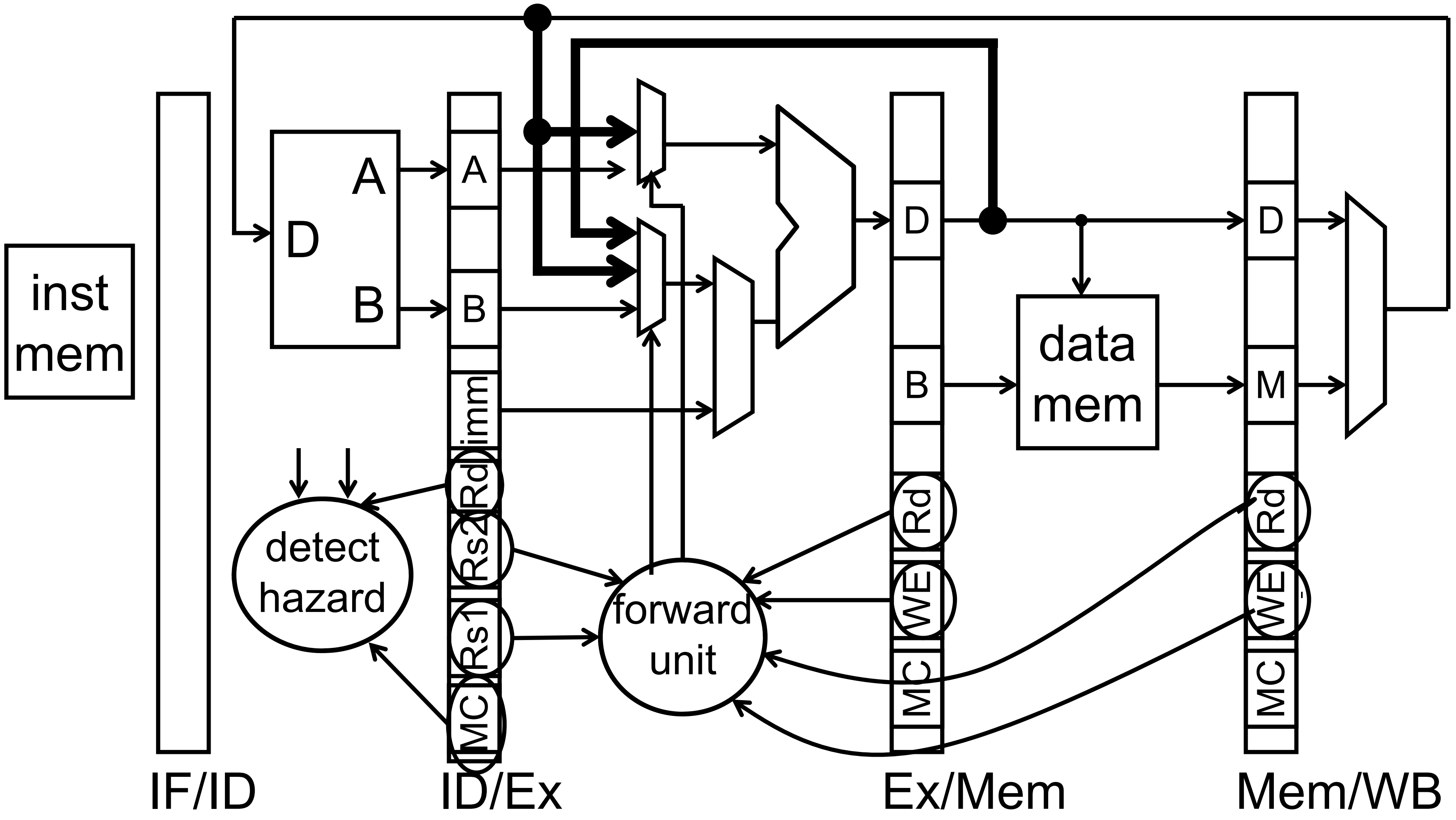
Load-Use Stall (1)



lw x4, x8, 20	IF	ID	Ex					
or x6, x4, x1		IF	ID					

load-use stall

Load-Use Detection



Stall = If(ID/Ex.MemRead &&
IF/ID.Rs1 == ID/Ex.Rd

Resolving Load-Use Hazards

RISC-V Solution : Load-Use Stall

- Stall must be inserted so that load instruction can go through and update the register file.
- Forwarding from RAM (Memory) is not an option.
- In some cases, real world compilers can optimize to avoid these situations.

Takeaway

Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

Stalling, preventing a dependent instruction from advancing, is one way to resolve data hazards. Stalling introduces NOPs ("bubbles") into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file. Bubbles (nops) in pipeline significantly decrease performance.

Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register). Better performance than stalling.

Quiz

Find all hazards, and say how they are resolved:

```
add  x3, x1, x2
nand x5, x3, x4
add  x2, x6, x3
lw    x6, x3, 24
sw    x6, x2, 12
```

Quiz

Find all hazards, and say how they are resolved:

```
add   x3, x1, x2
sub   x3, x2, x1
nand x4, x3, x1
or    x0, x3, x4
xor   x1, x4, x3
sb    x4, x0, 1
```

Data Hazard Recap

Delay Slot(s)

- Modify ISA to match implementation

Stall

- Pause current and all subsequent instructions

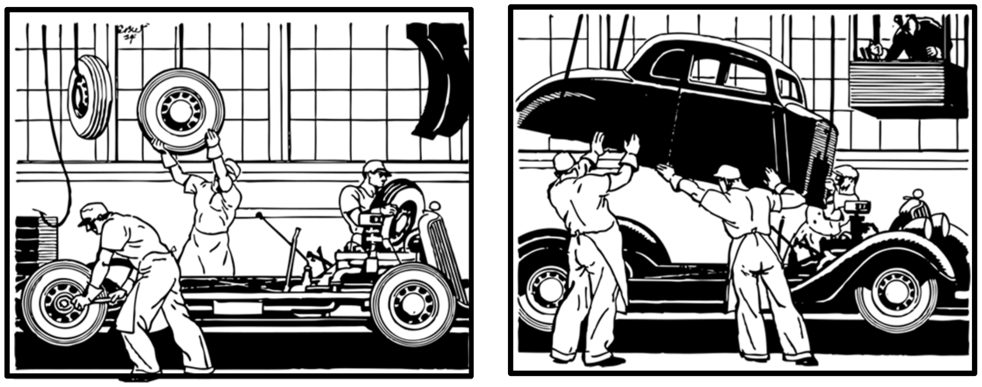
Forward/Bypass

- Try to steal correct value from elsewhere in pipeline
- Otherwise, fall back to stalling or require a delay slot

Tradeoffs?

Agenda

5-stage Pipeline
- Implementation
- Working Example



Hazards
- Structural
- Data Hazards
- Control Hazards

A bit of Context

```
i = 0;
do {
  n += 2;
  i++;
} while(i < max)
i = 7;
n--;
```

i → x1
Assume:
n → x2
max → x3

```
x10          addi x1, x0, 0      # i=0
x14    Loop: addi x2, x2, 2      # n += 2
x18          addi x1, x1, 1      # i++
x1C          blt  x1, x3, Loop   # i<max?
x20          addi x1, x0, 7      # i = 7
x24          subi x2, x2, 1      # n--
```

Control Hazards

Control Hazards

- instructions are fetched in stage 1 (IF)
- branch and jump decisions occur in stage 3 (EX)

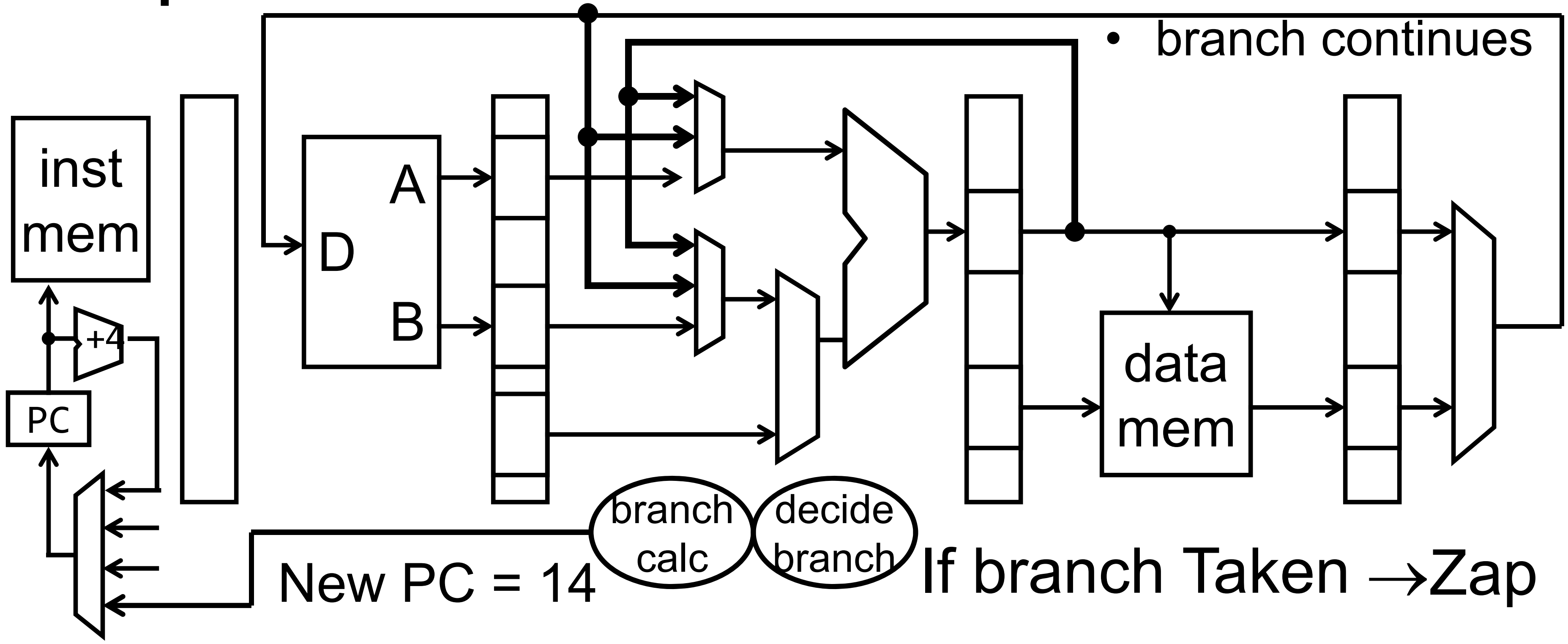
  → next PC not known until ***2 cycles*** *after* branch/jump

```
x1C     blt  x1, x3, Loop
x20     addi x1, x0, 7
x24     subi x2, x2, 1
```

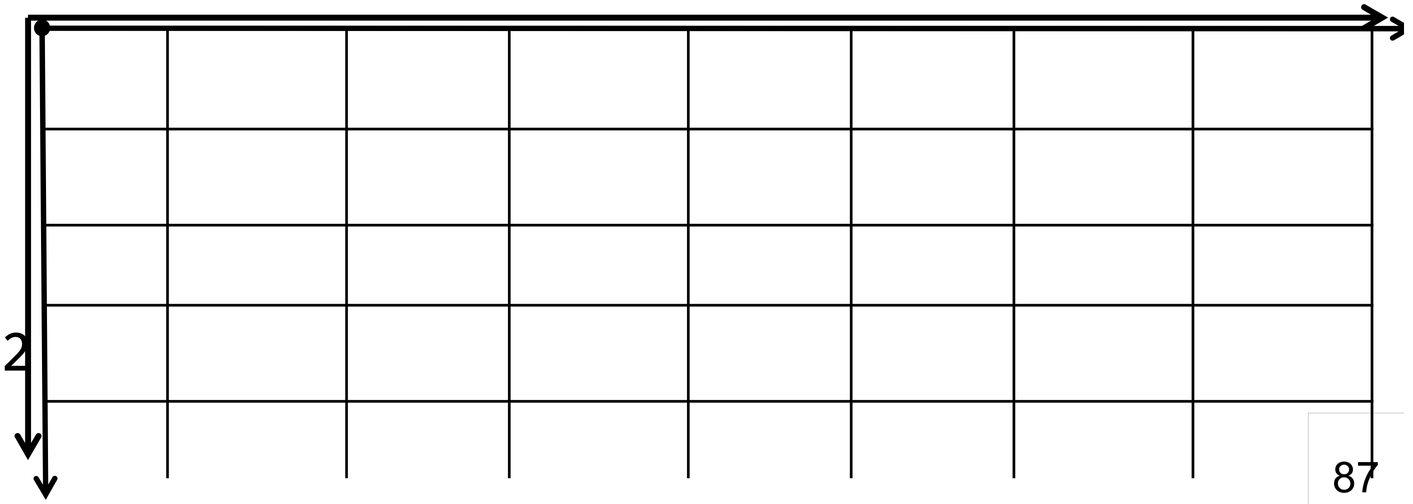
Branch ***<u>not</u>*** taken?
  No Problem!
Branch taken?
  Just fetched 2 insns
  → **Zap & Flush**

Zap & Flash

- prevent PC update
- clear IF/ID latch
- branch continues



If branch Taken →Zap

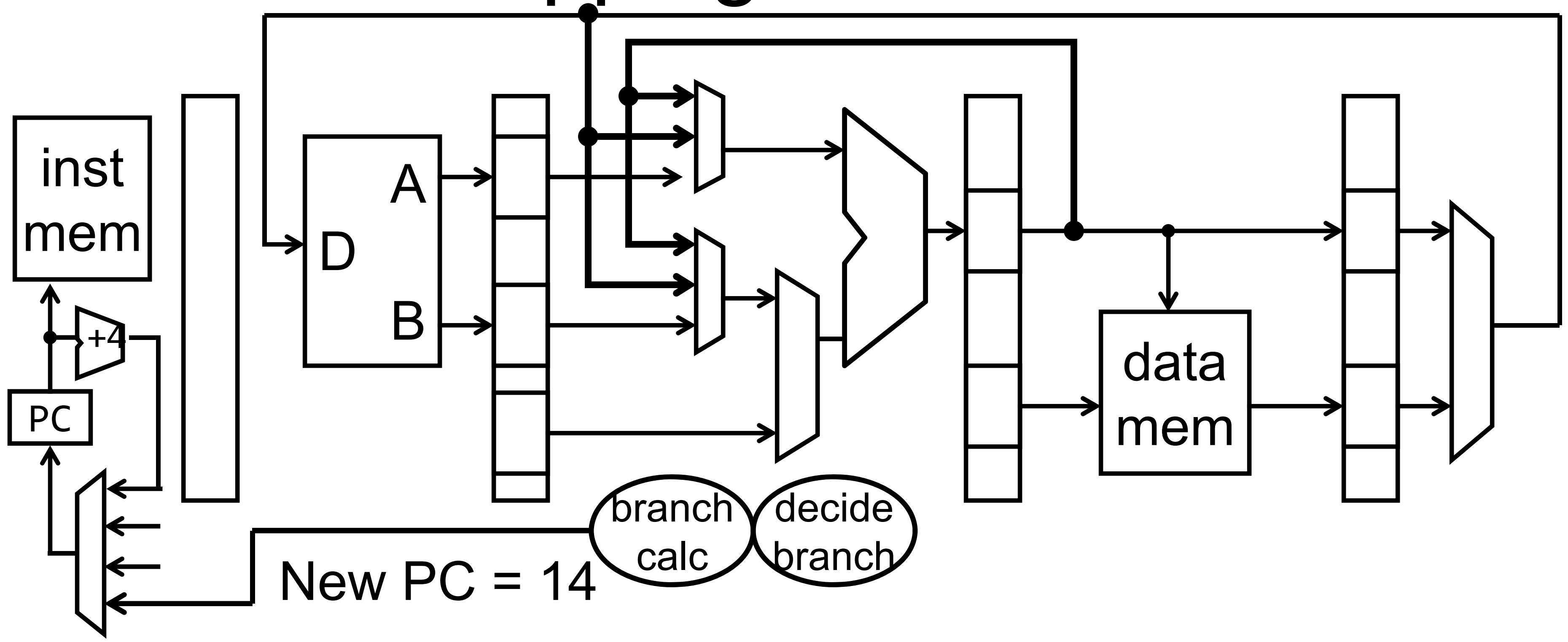


```
1C blt x1,x3,L
20 addi x1,x0,7
24 subi x2,x2,1
14 L:addi x2,x2,2
```

Reducing the cost of control hazard

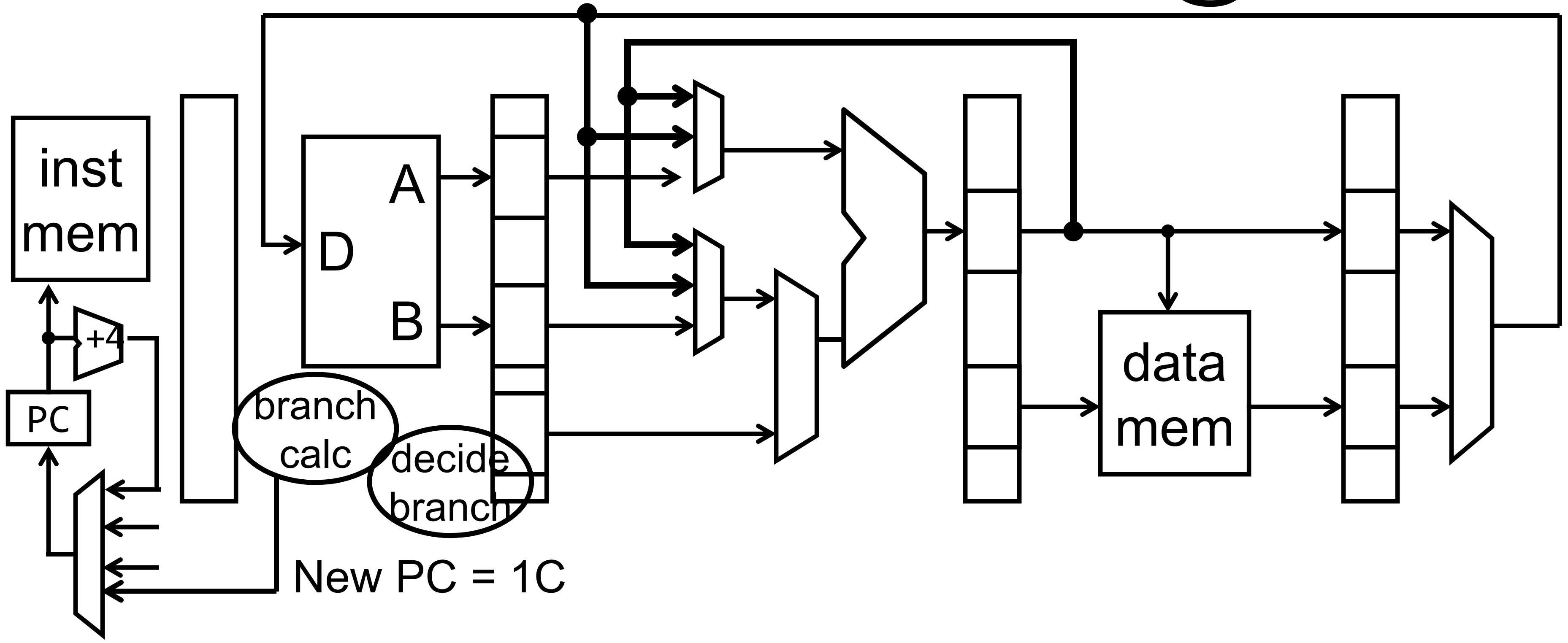
1. Resolve Branch at Decode
   - Some groups do this for Project 3, your choice
   - Move branch calc from EX to ID
   - Alternative: just zap 2nd instruction when branch taken
2. Branch Prediction
   - Not in 3410, but every processor worth *anything* does this (*no offense!*)

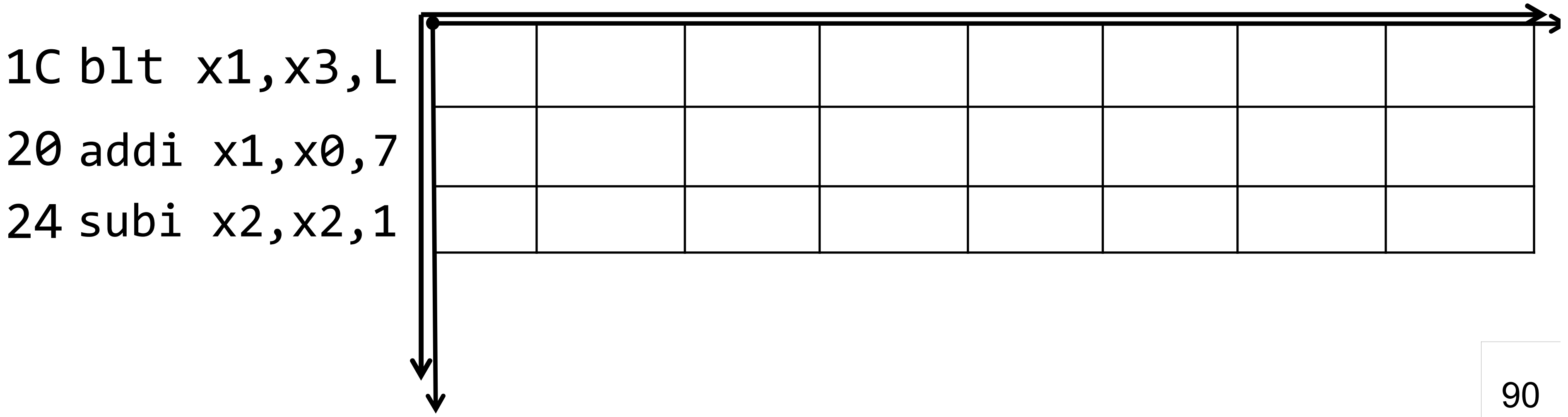
Problem: Zapping 2 insns/branch



1C blt x1,x3,L							
20 addi x1,x0,7							
24 subi x2,x2,1							

Soln #1: Resolve Branches @ Decode




1C blt x1,x3,L
20 addi x1,x0,7
24 subi x2,x2,1

Branch Prediction

Most processor support **Speculative Execution**

- *Guess* direction of the branch
  - Allow instructions to move through pipeline
  - Zap them later if guess turns out to be wrong
- A *must* for long pipelines

Summary

Control hazards

- Is branch taken or not?
- Performance penalty: stall and flush

Reduce cost of control hazards

- Move branch decision from Ex to ID
    - 2 nops to 1 nop
- Branch prediction
    - Correct. Great!
    - Wrong. Flush pipeline. Performance penalty

Hazards Summary

Data hazards

Control hazards

Structural hazards

- resource contention
- so far: impossible because of ISA and pipeline design

Hazards Summary

Data hazards

- register file reads occur in stage 2 (IF)
- register file writes occur in stage 5 (WB)
- next instructions may read values soon to be written

Control hazards

- branch instruction may change the PC in stage 3 (EX)
- next instructions have already started executing

Structural hazards

- resource contention
- so far: impossible because of ISA and pipeline design

Data Hazard Takeaways

Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. Pipelined processors need to detect data hazards.

Stalling, preventing a dependent instruction from advancing, is one way to resolve data hazards. Stalling introduces NOPs ("bubbles") into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file. Nops significantly decrease performance.

Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register). Better performance than stalling.

Control Hazard Takeaways

Control hazards occur because the PC following a control instruction is not known until control instruction is executed. If branch is taken → need to zap instructions. 1 cycle performance penalty.

We can reduce cost of a control hazard by moving branch decision and calculation from Ex stage to ID stage.