COMP4611: Design and Analysis of Computer Architectures

Basic Pipelining

Lin Gu

CSE, HKUST

Pipelining – Boeing Everett

Courtesy photos by Boeing

Introduction to Pipelining

Pipelining: An implementation technique that overlaps the execution of multiple instructions. It is a **key** technique for processors to achieve high performance.

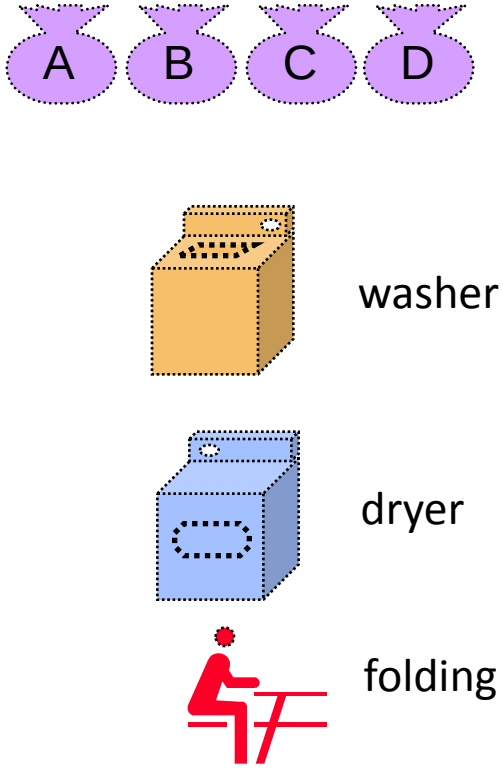
Laundry Example

Ann, Brian, Cathy, Dave
each has one load of clothes
to wash, dry, and fold

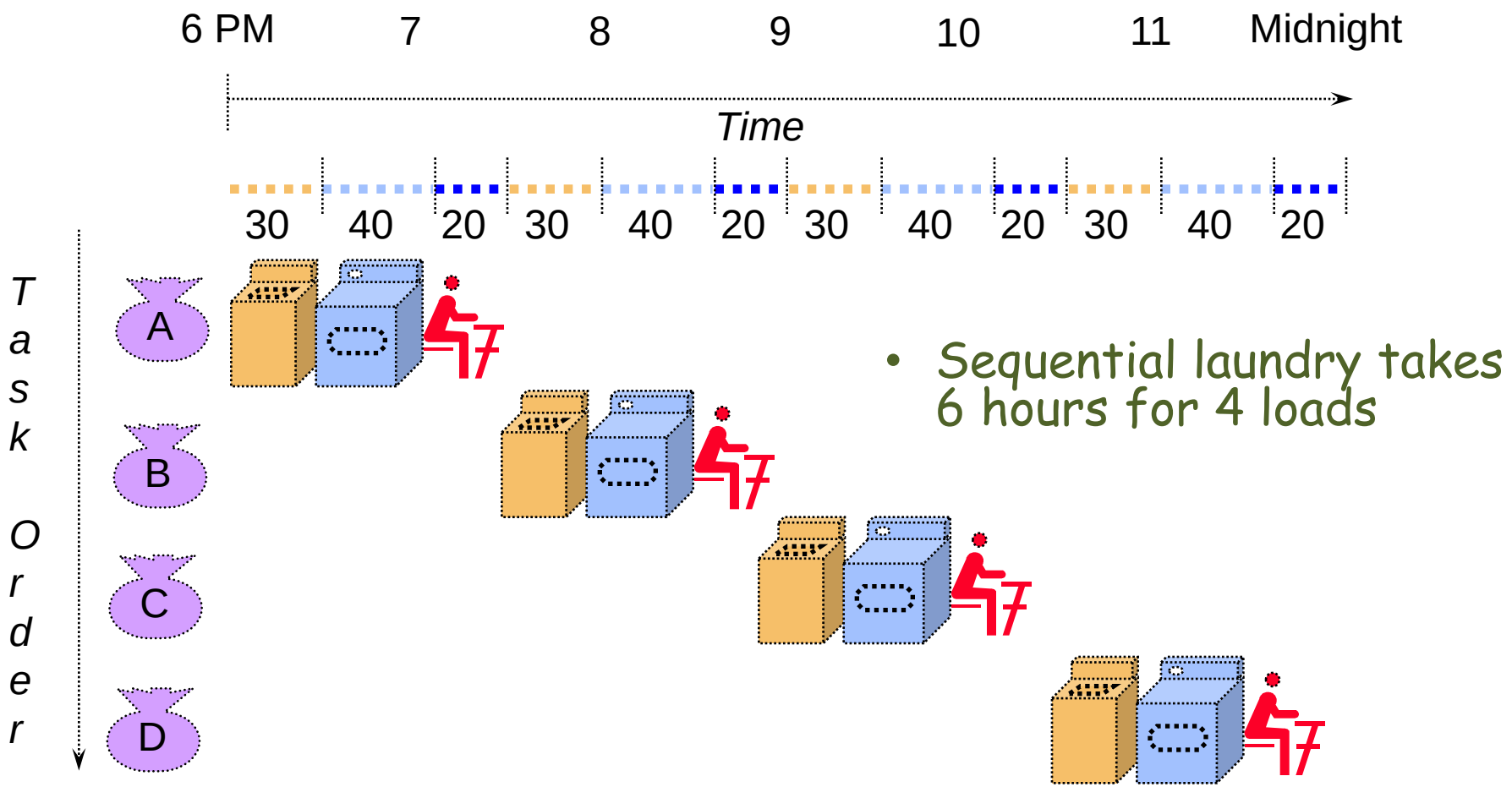
Washing takes 30 minutes

Drying takes 40 minutes

Folding takes 20 minutes

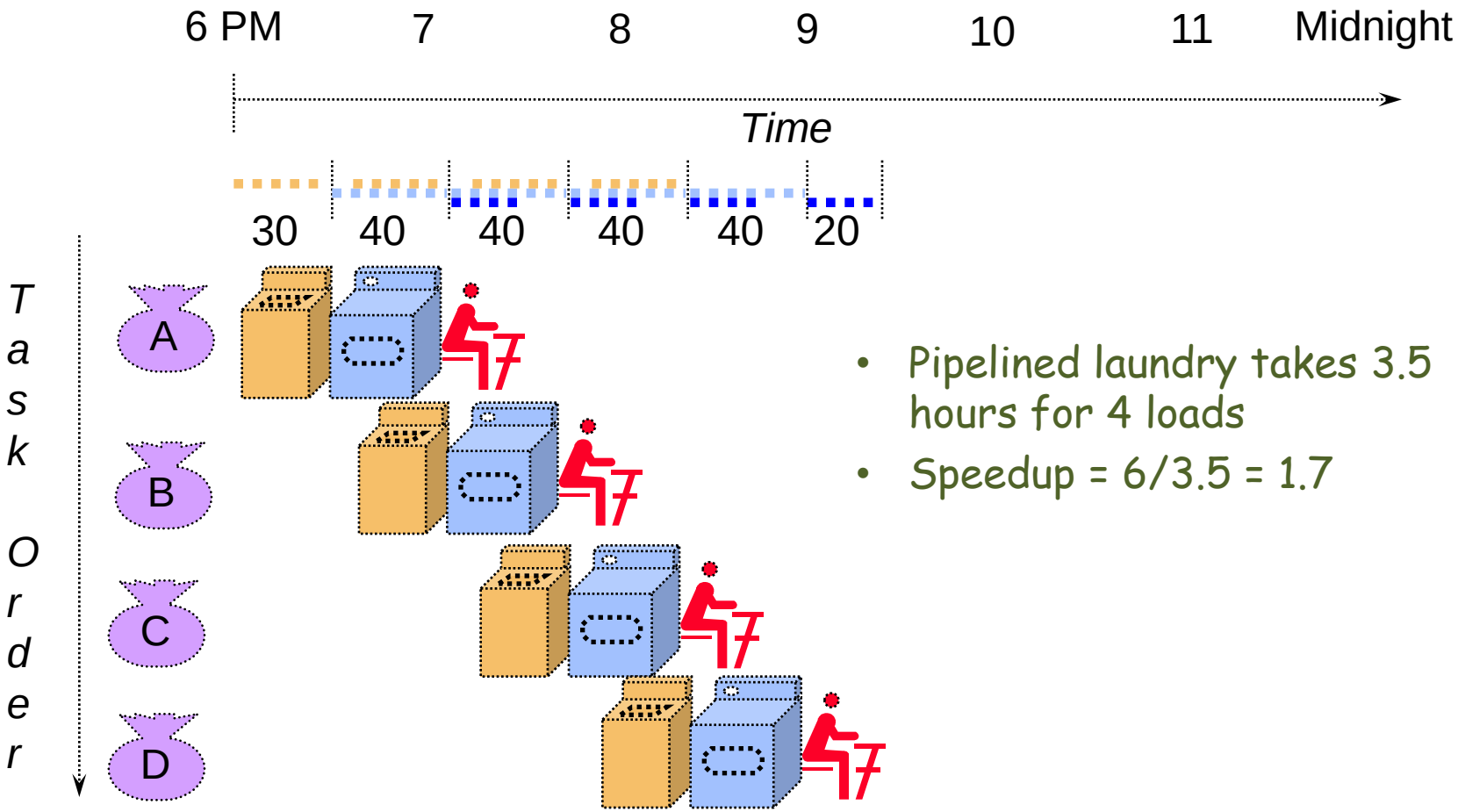


Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads

Pipelined Laundry: start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads
- Speedup = 6/3.5 = 1.7

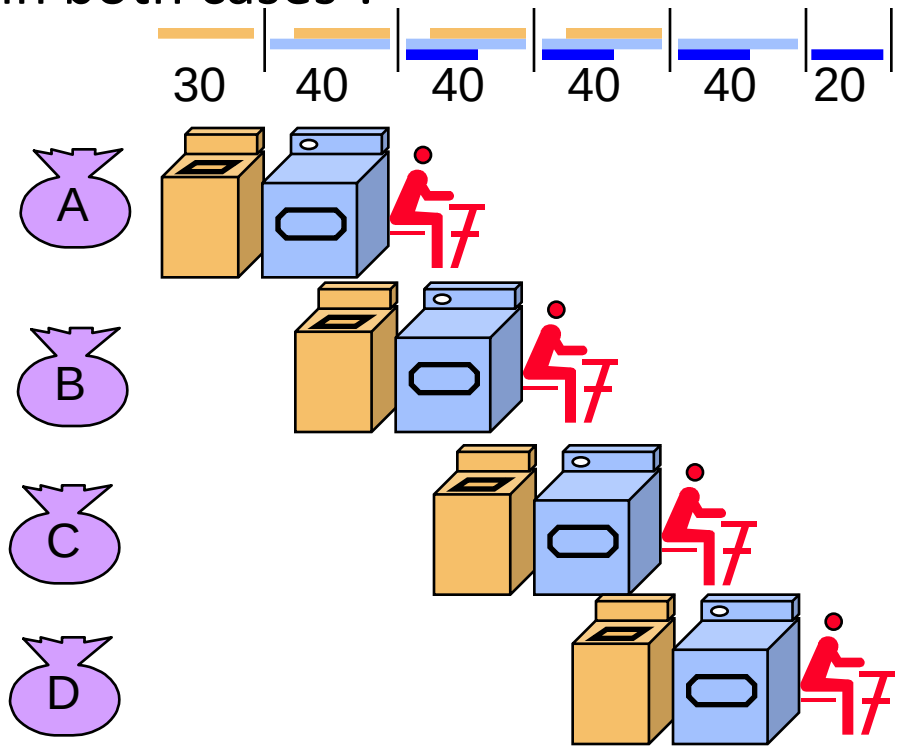
Pipelining Lessons

- Latency vs. Throughput

  Question

  – What is the latency in both cases ?

  – What is the throughput in both cases ?



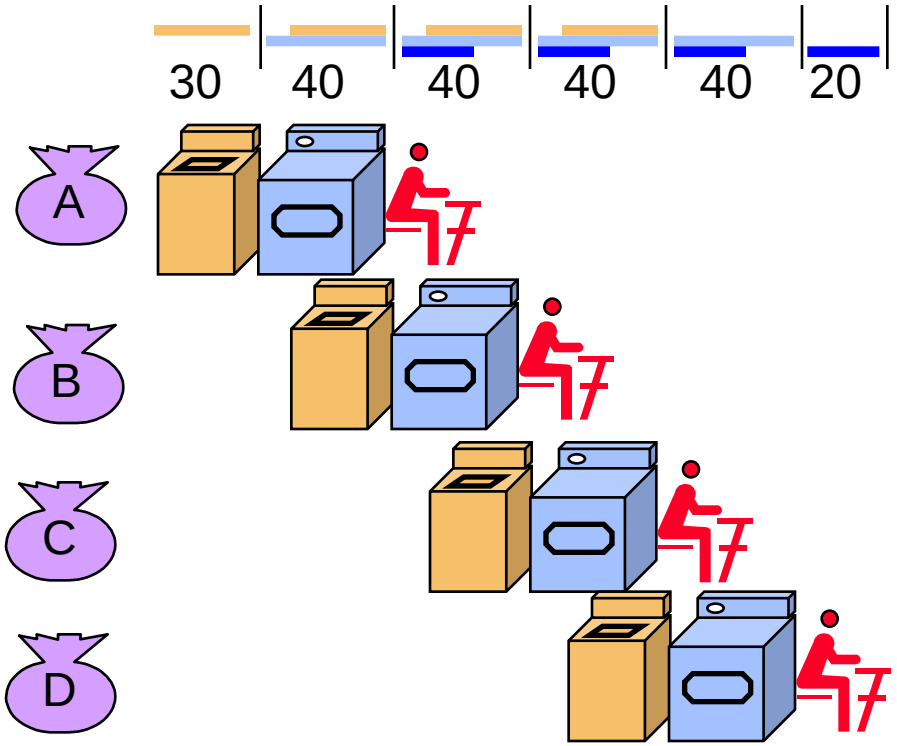
- Pipelining doesn't reduce latency of single task,
- It increases throughput of entire workload

Pipelining Lessons [cont'd...]

Question

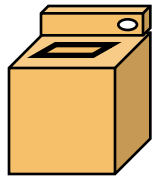
- What is the fastest operation in the example ?
- What is the slowest operation in the example

Pipeline rate limited by slowest pipeline stage

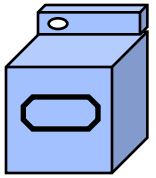


Pipelining Lessons [cont'd...]

- Washing takes 30 minutes



- Drying takes 40 minutes

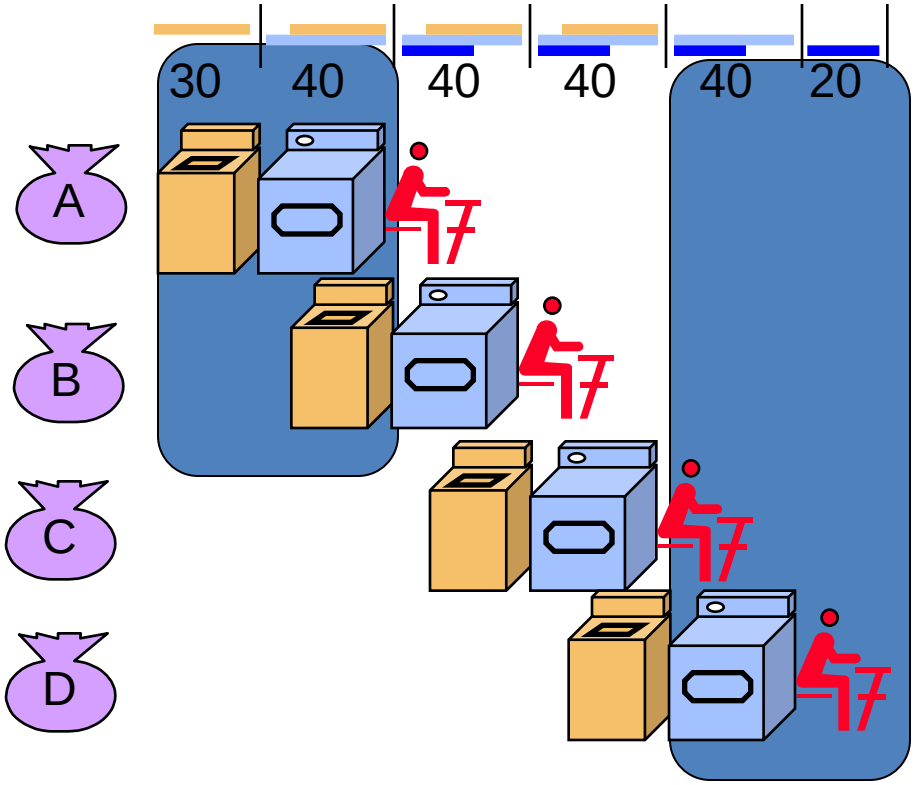


- Folding takes 20 minutes

- Question
  - Would it matter if "folding" also took 40 minutes

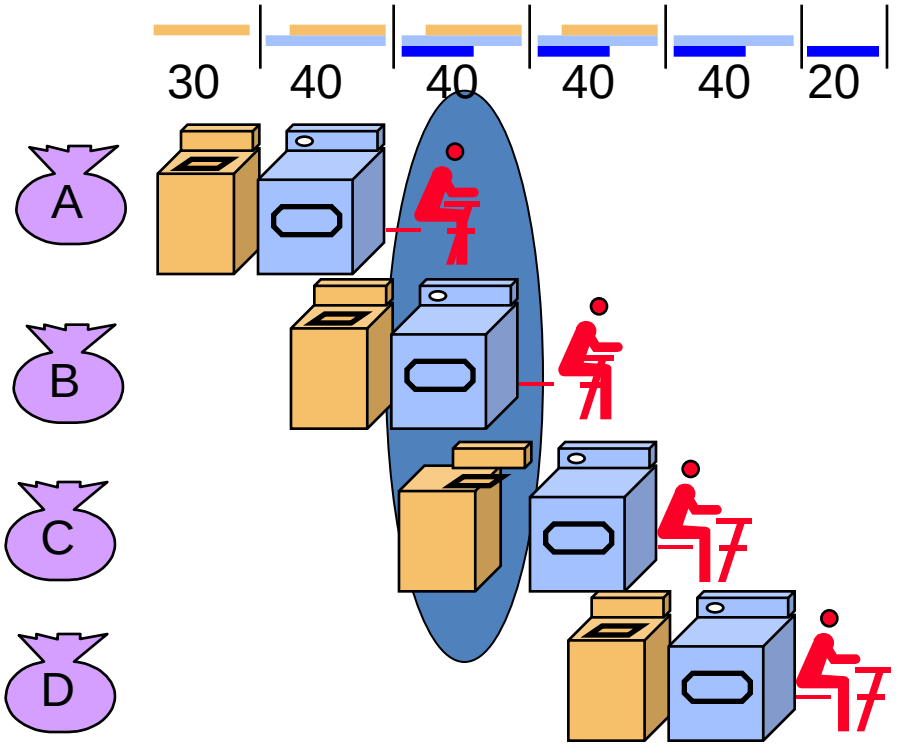
Unbalanced lengths of pipe stages reduce speedup

Pipelining Lessons [cont'd...]



Time to "fill" pipeline and time to "drain" it reduce speedup

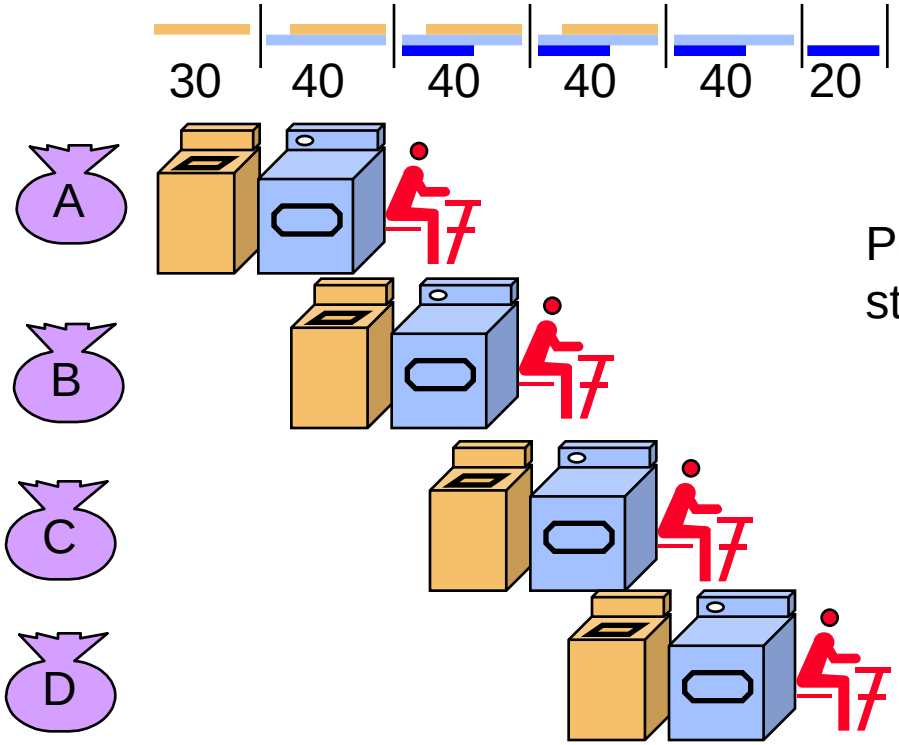
Pipelining Lessons [cont'd…]



Multiple tasks operating simultaneously using different resources

Pipelining Lessons [contd…]

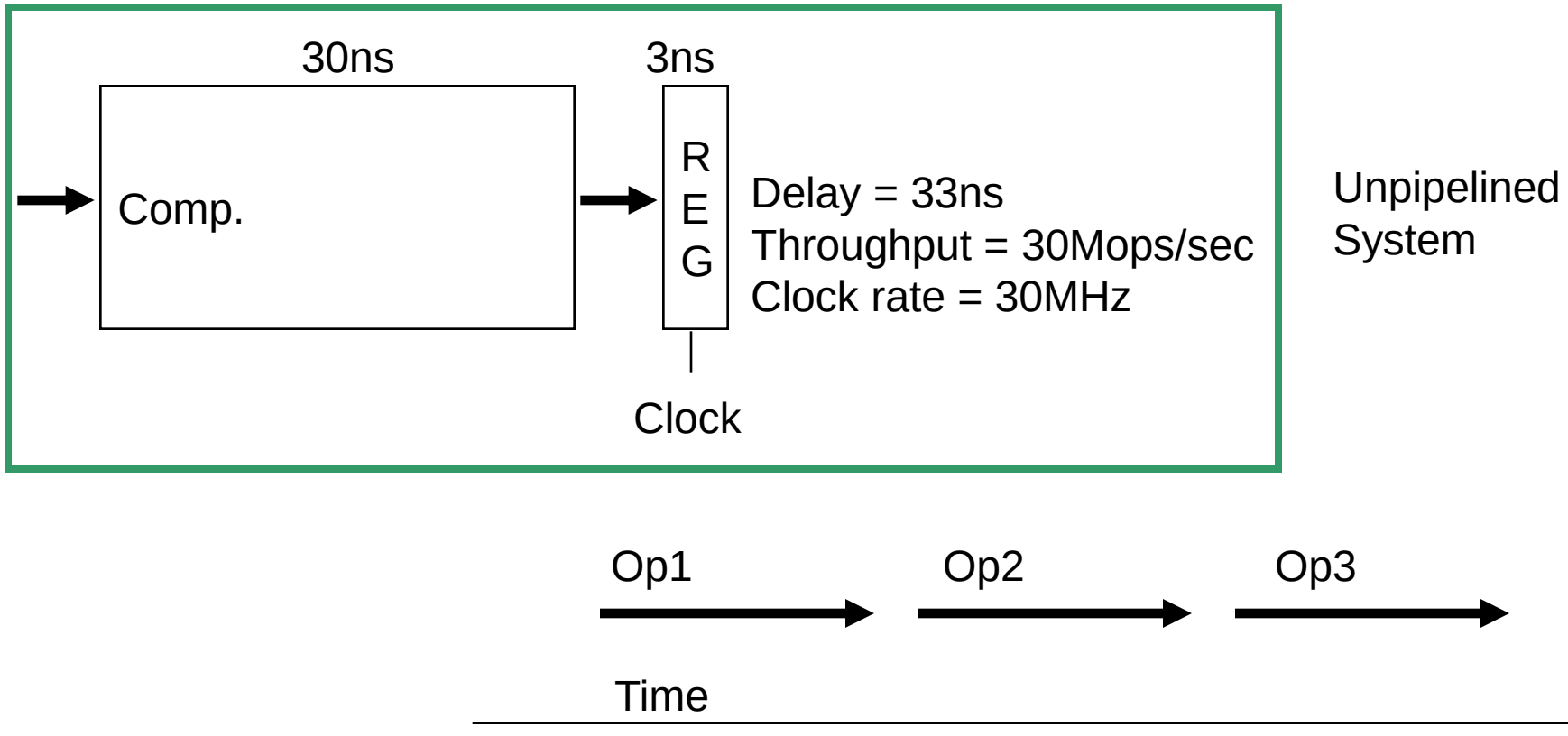
- Question
  - Would the speedup increase if we had more steps ?



Potential Speedup = Number of pipe stages

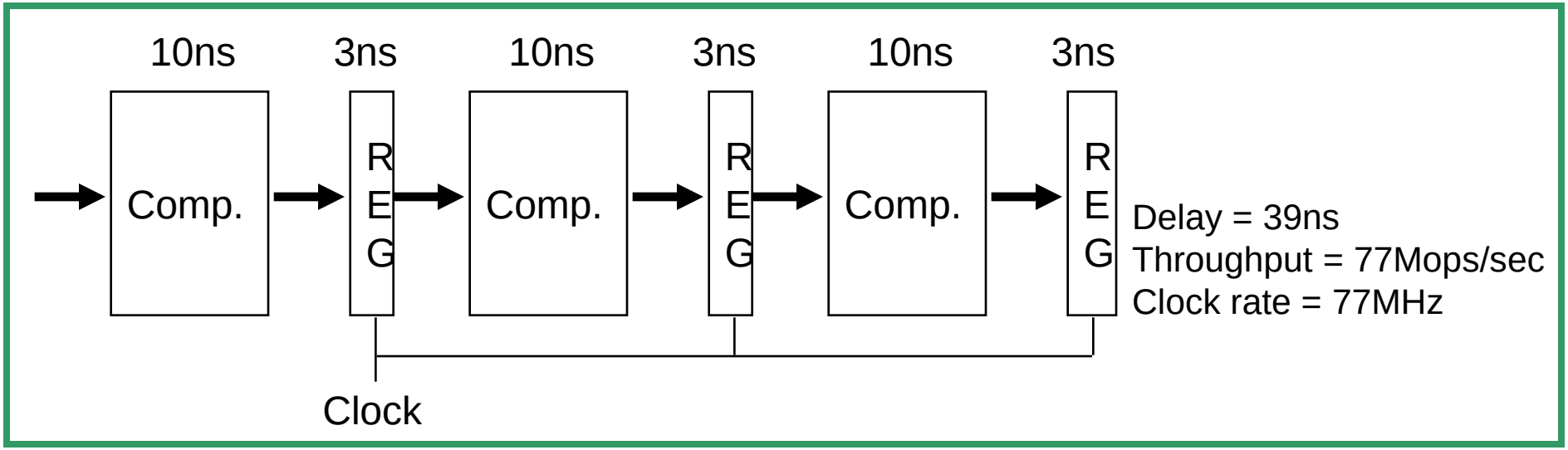
More steps = Deeper pipeline

Pipelining in Computation

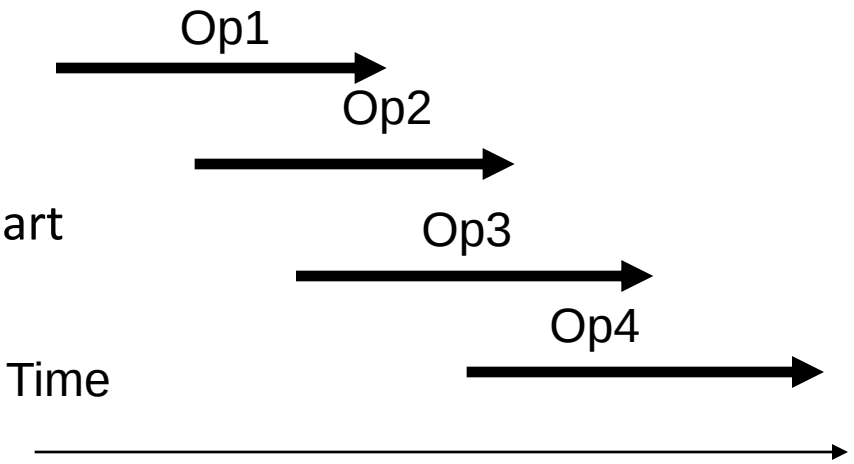


- One operation must complete before next can begin
- Operations spaced 33ns apart

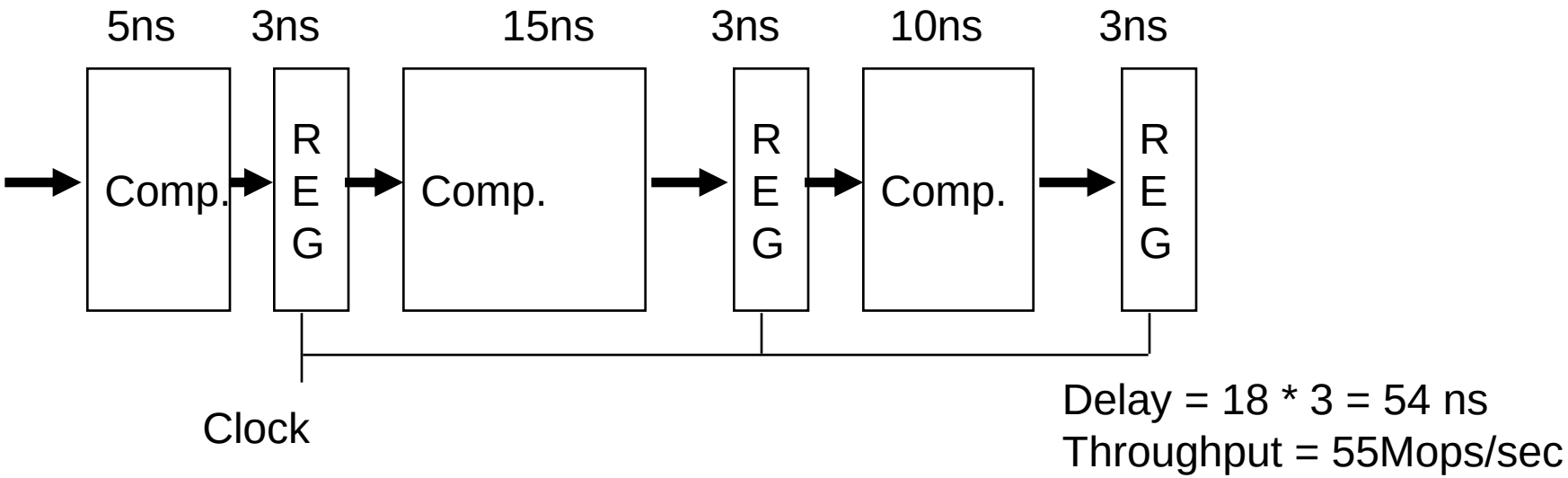
3 Stage Pipelining



- Space operations 13ns apart
- 3 operations occur simultaneously

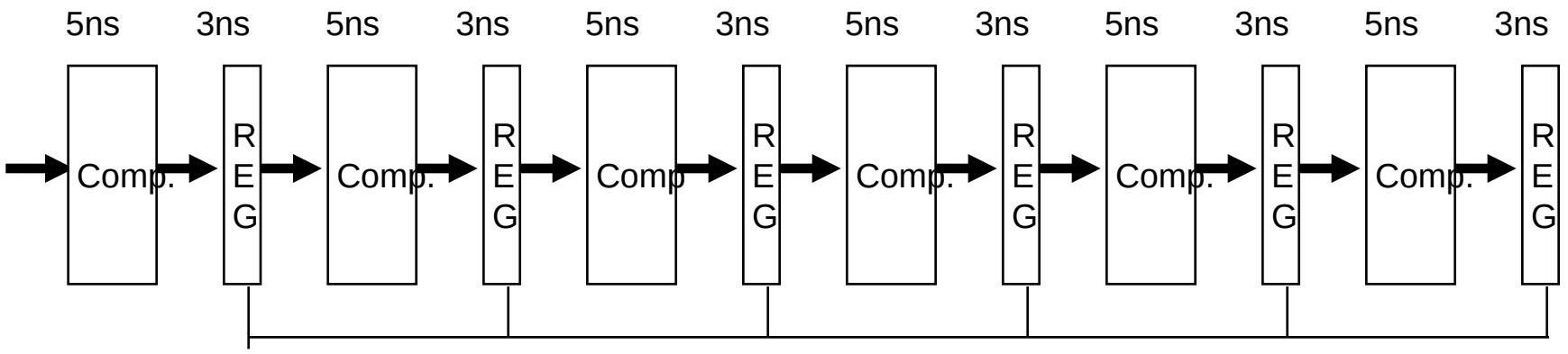


Limitation: Non-uniform Pipelining



- Throughput limited by slowest stage
  - Delay determined by clock period * number of stages
- Must attempt to balance stages

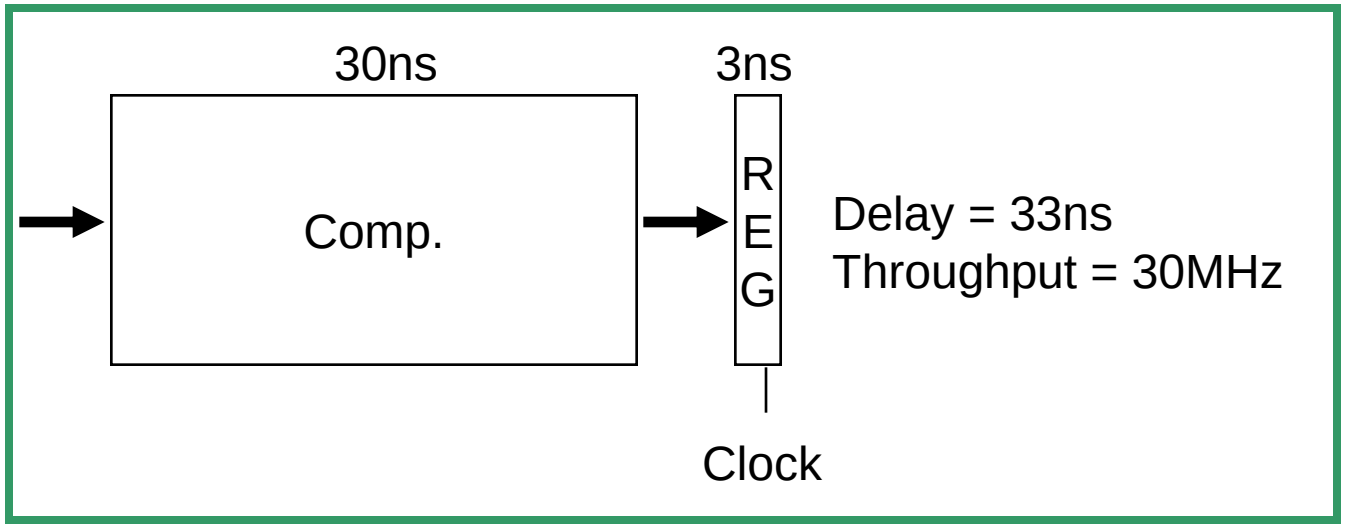
Limitation: Deep Pipelines



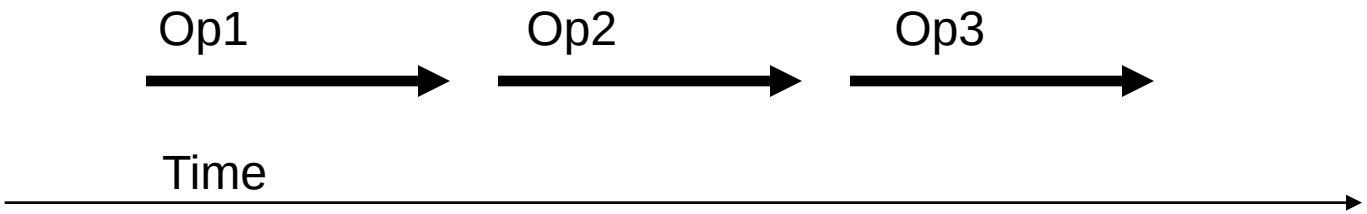
Latency = 48ns, Throughput = 128Mops/sec

- Diminishing returns as add more pipeline stages
- Register delays become limiting factor
  - Increased latency
  - Small throughput gains
  - More hazards

Pipelining in Computation

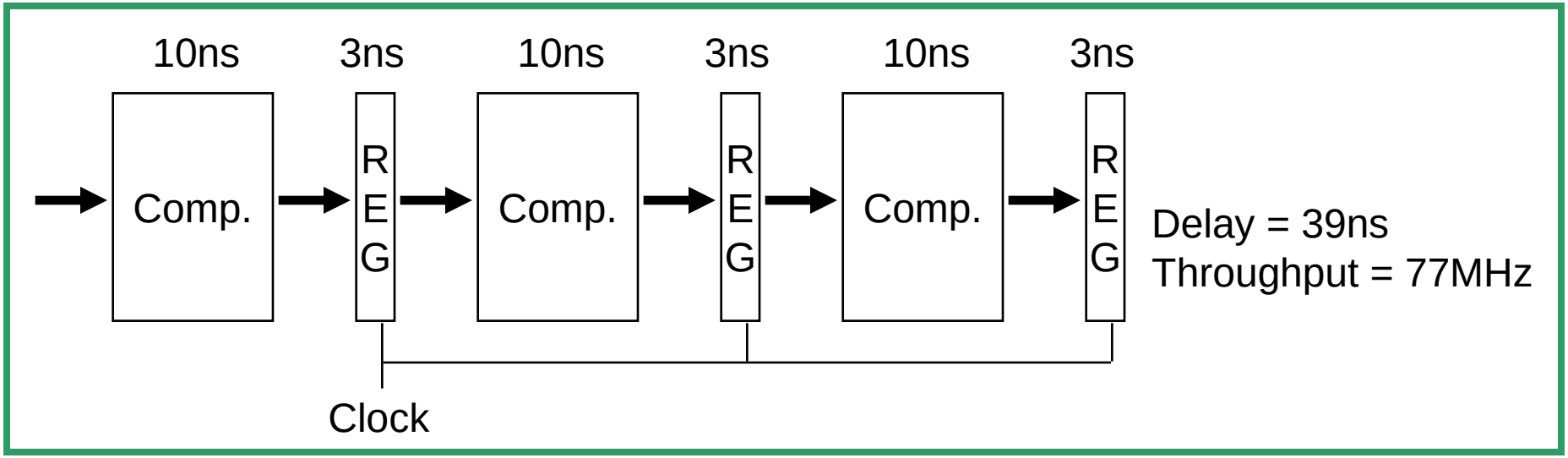


Unpipelined System

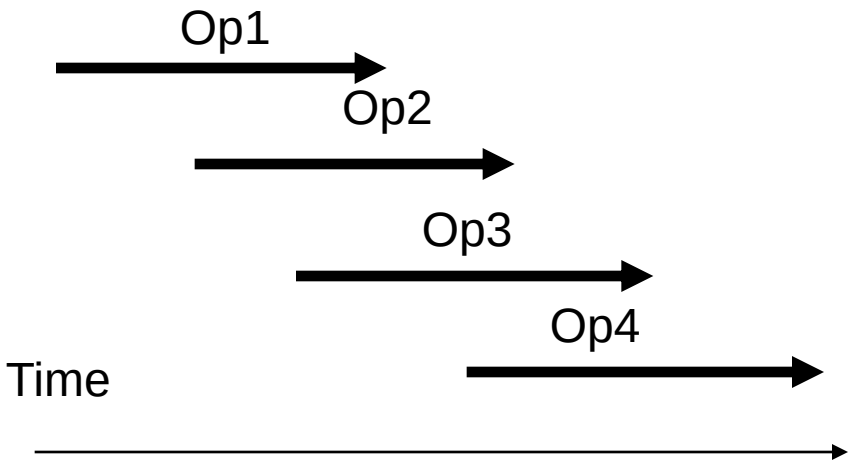


- One operation must complete before next can begin
- Operations spaced 33ns apart

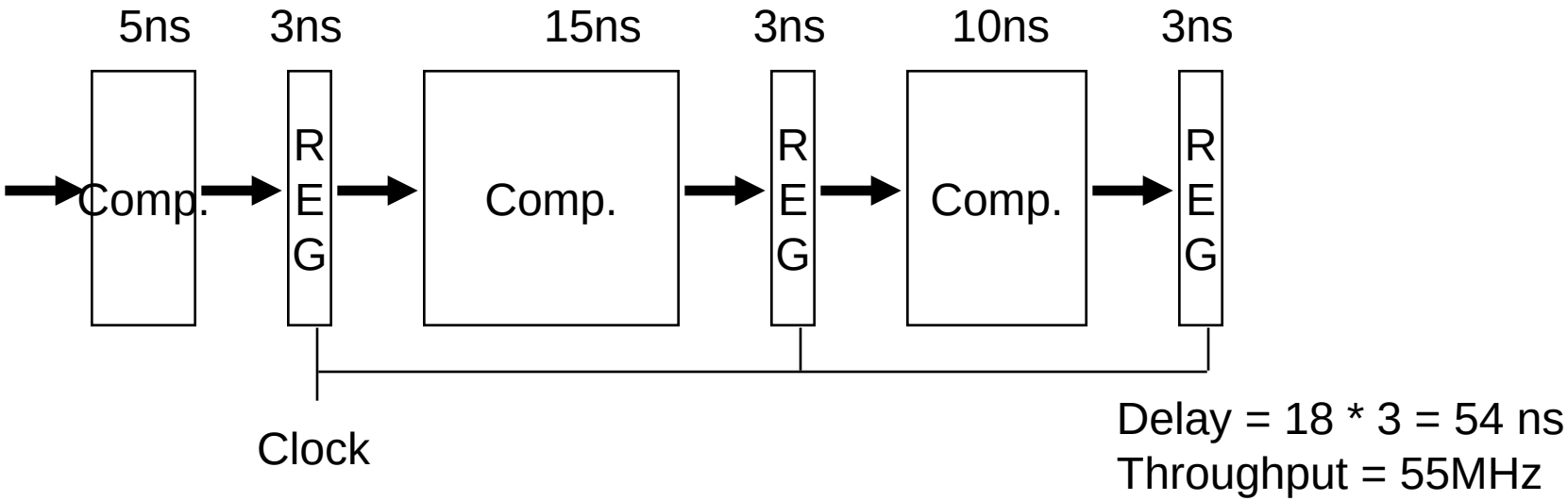
3 Stage Pipelining



- Space operations 13ns apart
- 3 operations occur simultaneously

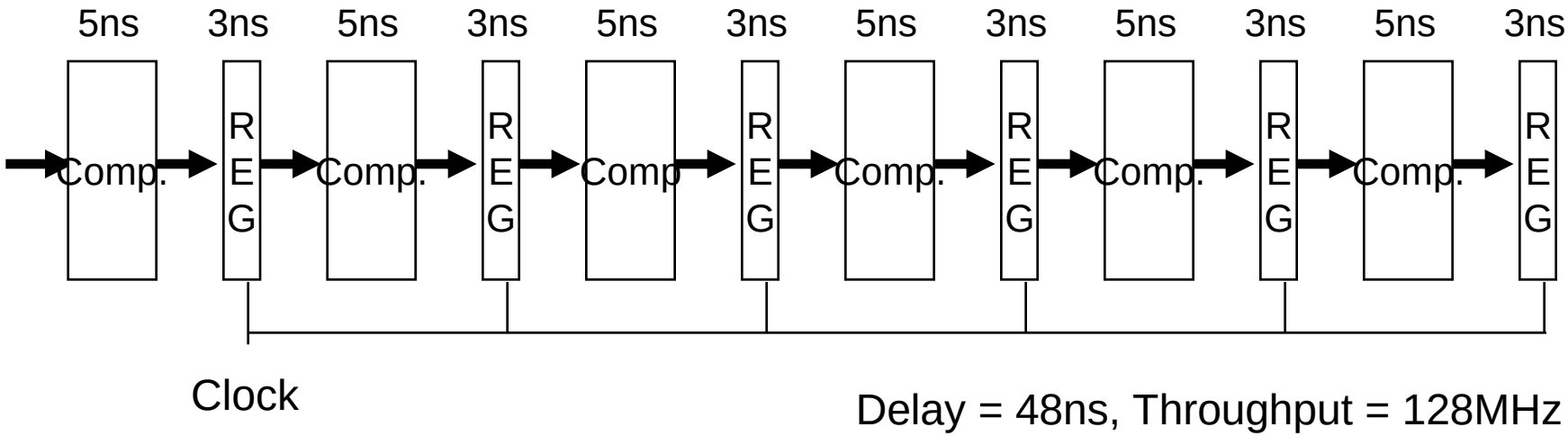


Limitation: Nonuniform Pipelining



- Throughput limited by slowest stage
  - Delay determined by clock period * number of stages
- Must attempt to balance stages

Limitation: Deep Pipelines



- Diminishing returns as add more pipeline stages
- Register delays become limiting factor
  - Increased latency
  - Small throughput gains
  - More hazards

Computer (Processor) Pipelining

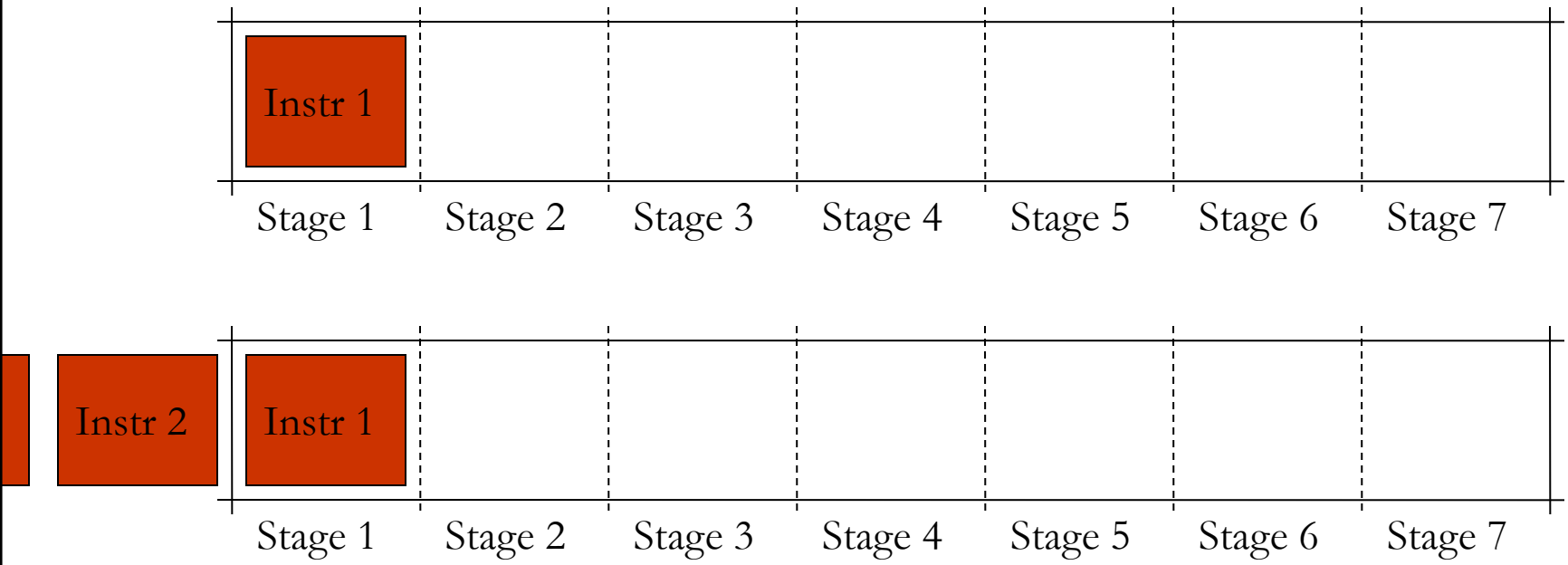
- It is one *KEY* method of achieving High-Performance in modern microprocessors

- A major advantage of pipelining over "parallel processing" is that it is not visible to the programmer

- An instruction execution pipeline involves a number of steps, where each step completes a part of the operations required for executing an instruction.
  - Each step is called *a pipe stage* or *a pipe segment.*
  - The stages or steps are connected one to the next to form a pipe -- instructions enter at one end and progress through the stages and exit at the other end.

Pipelining

- Multiple instructions overlapped in execution
- Throughput increased
- Doesn't reduce time for individual instructions



Computer Pipelining

- ***Throughput*** of an instruction pipeline is determined by how frequently instructions exit the pipeline.
- An instruction moves one step down the pipeline after every time interval C, where C equals to ***the cycle time or machine cycle*** (1/Clock Rate) and is determined by the stage with the longest processing delay (slowest pipeline stage).

Pipelining: Design Goals

- An important pipeline design consideration is to balance the length of each pipeline stage.

- If all stages are perfectly balanced, then the time per instruction on a pipelined machine (assuming ideal conditions with no stalls) is

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

Pipelining: Design Goals

- Under these ideal conditions:
  - Speedup from pipelining equals the number of pipeline stages: n
  - One instruction is completed every cycle, CPI = 1 .
  - This is an asymptote of course, but +10% is commonly achieved
    - Difference is due to difficulty in achieving balanced stage design
- Two ways to view the performance mechanism
  - Reduced CPI (i.e. non-piped to piped change)
    - Close to 1 instruction/cycle if you're lucky
  - Reduced cycle-time (i.e. increasing pipeline depth)
    - Work split into more stages
    - Simpler stages result in faster clock cycles

Implementation of MIPS

- We use the MIPS processor as an example to demonstrate the concepts of computer pipelining.
- MIPS ISA reflects careful measurements and informed architectural decisions.
- The design of its pipeline needs to answer the following questions

  How are instructions executed?

  How to pipeline them?

How to Execute an Instruction?

"Vēnī, vīdī, vīcī"

– "I came, I saw, I conquered"

Bring the instruction to the processor – "Fetch"

Examine the instruction – "Decode"

Do the work – "Execute, Memory-Access, Writeback"

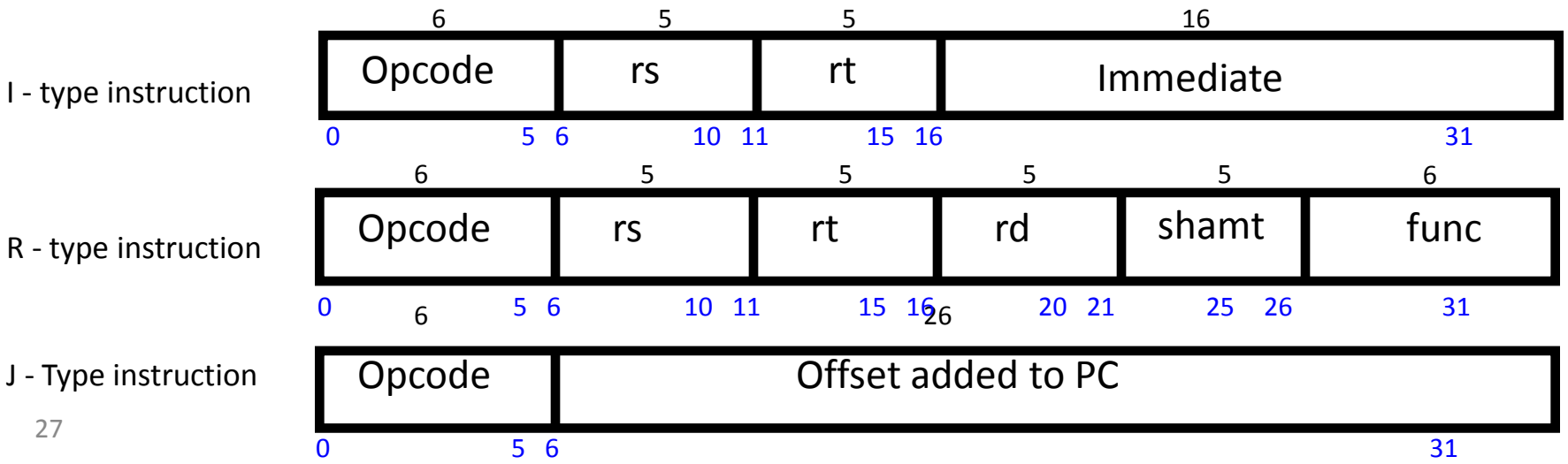
IF, ID, EX, MEM, WB

A Basic Multi-Cycle Implementation of MIPS

1 Instruction fetch cycle (IF):

IR ← Mem[PC]
NPC ← PC + 4

Note: IR (instruction register), NPC (next sequential program counter register)



A Basic Multi-Cycle Implementation of MIPS

2 Instruction decode/register fetch cycle (ID):

Parse instruction

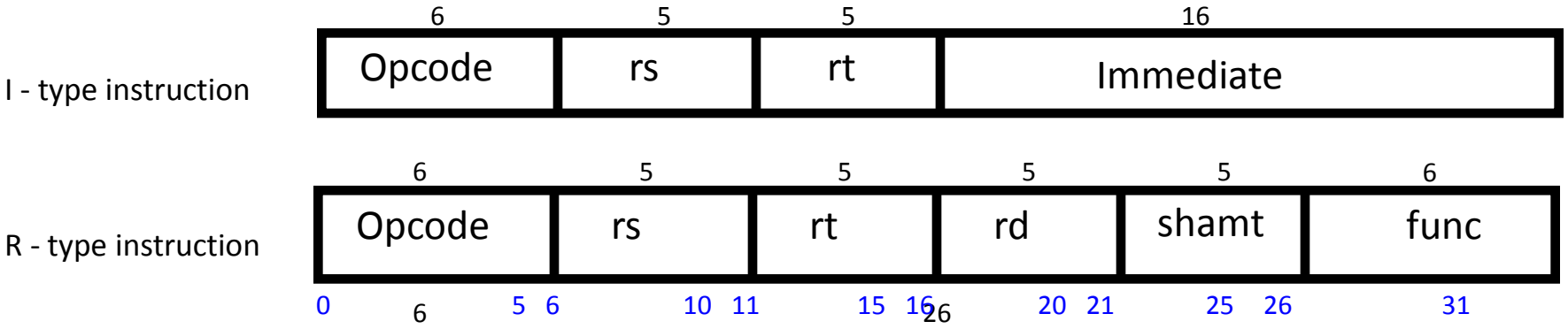
Read:

A ← Regs[rs];

B ← Regs[rt];

Imm ← $((IR_{16})^{16}\#\#IR_{16..31})$ sign-extended immediate field of IR

Note: A, B, Imm are temporary registers



A Basic Implementation of MIPS (continued)

3 Execution/Effective address cycle (EX):

- **Memory reference:**

```
ALUOutput ← A + Imm;
```

- **Register-Register ALU instruction:**

```
ALUOutput ← A op B;
```

- **Register-Immediate ALU instruction:**

```
ALUOutput ← A op Imm;
```

- **Branch (simplification – only consider BEQZ):**

```
ALUOutput ← NPC + Imm;
cond ← (A == 0)
```

A Basic Implementation of MIPS (continued)

4 Memory access/branch completion cycle (MEM):

- **Memory reference:**

  LMD ← Mem[ALUOutput] or
  Mem[ALUOutput] ← B;

- **Branch:**

  if (cond) PC ← ALUOutput else PC ← NPC

Note: LMD (load memory data) register

A Basic Implementation of MIPS (continued)

5 Write-back cycle (WB):

- **Register-Register ALU instruction:**

  Regs[rd] ← ALUOutput;

- **Register-Immediate ALU instruction:**

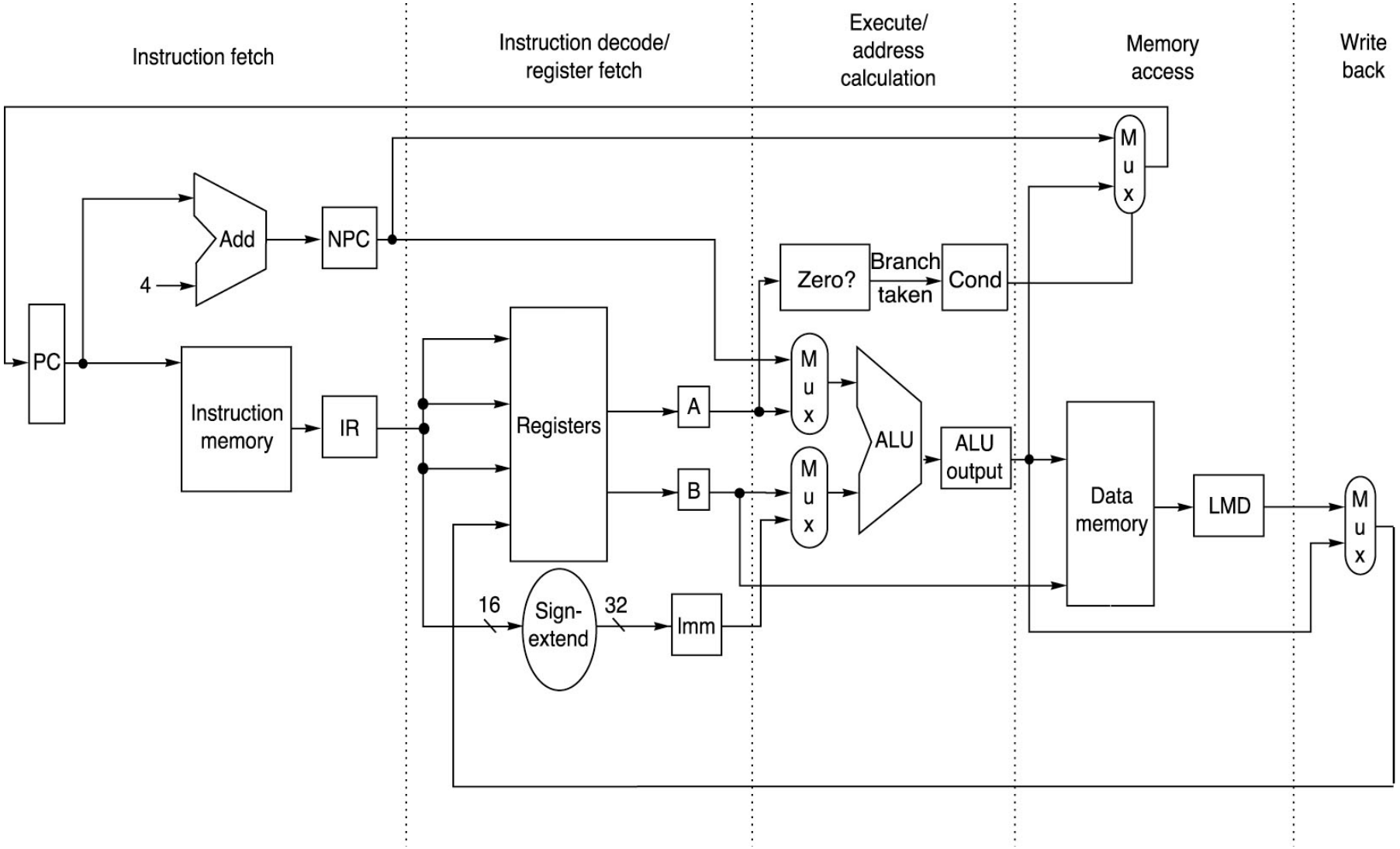
  Regs[rt] ← ALUOutput;

- **Load instruction:**

  Regs[rt] ← LMD;

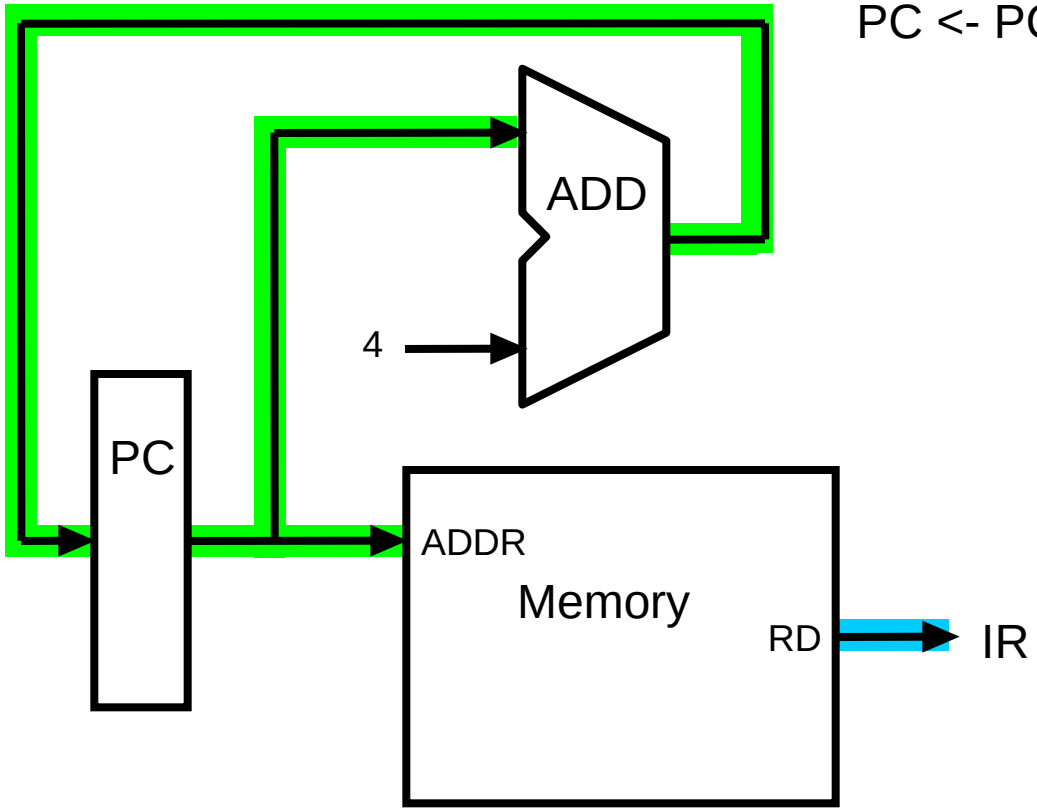
Note: LMD (load memory data) register

Basic MIPS Integer Datapath Implementation



Datapath for Instruction Fetch

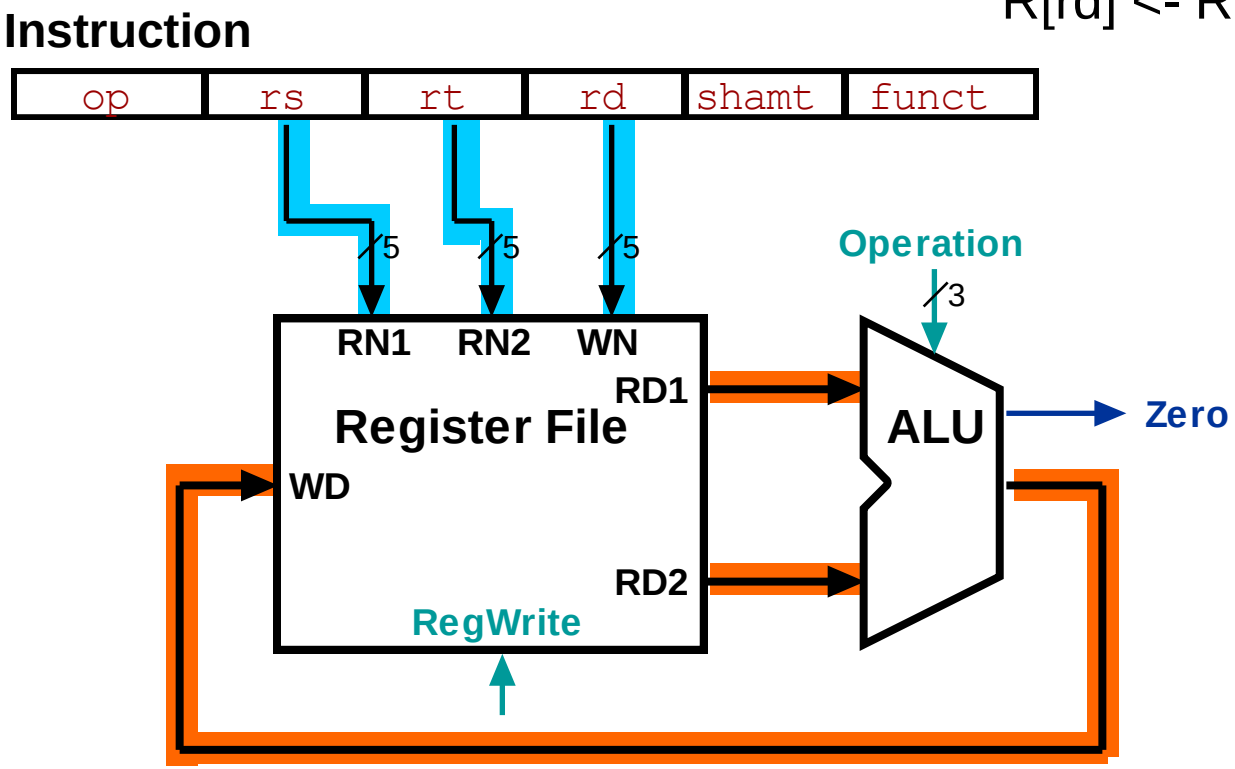
IR <- MEM[PC]
PC <- PC + 4



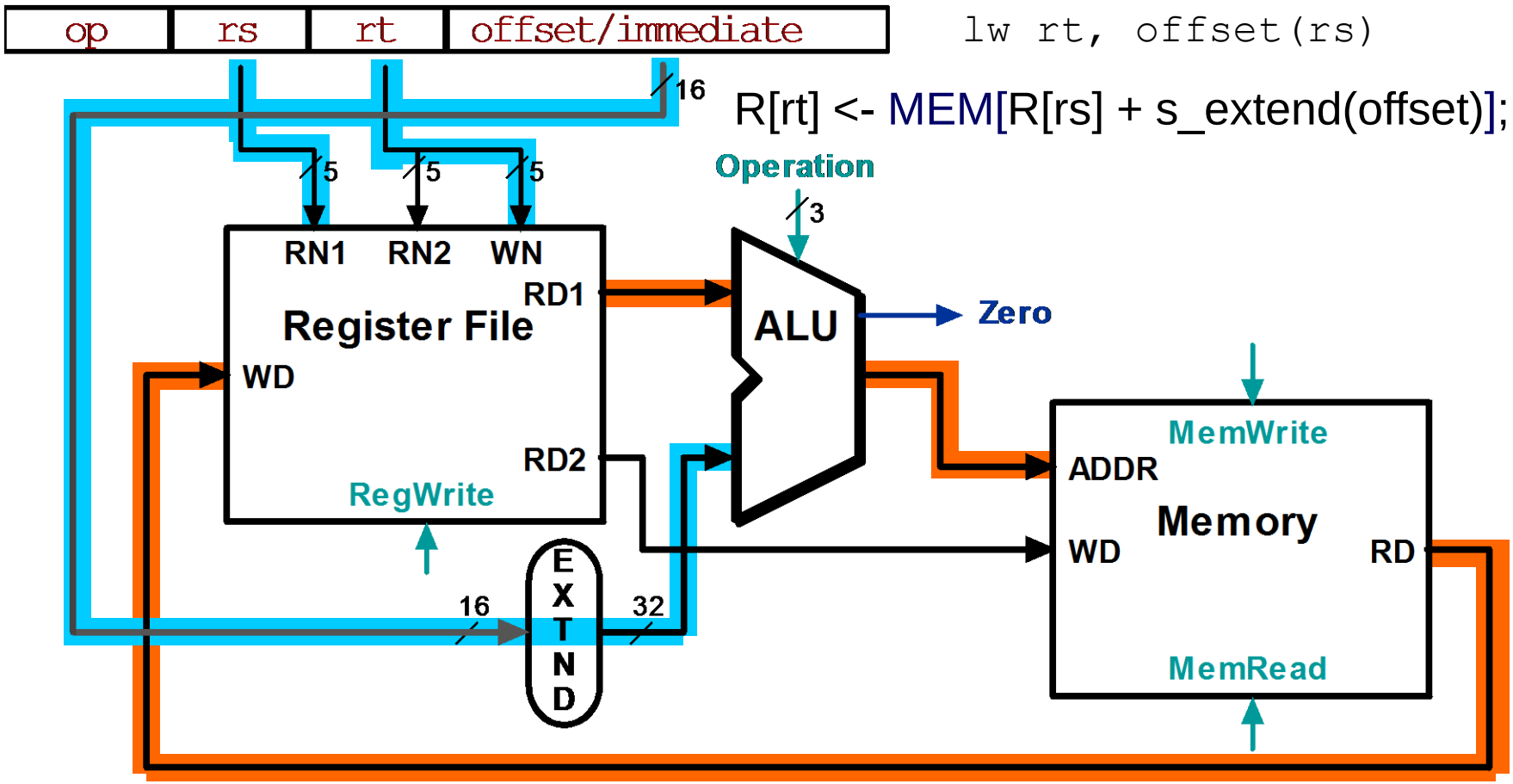
Datapath for R-Type Instructions

```
add rd, rs, rt
```

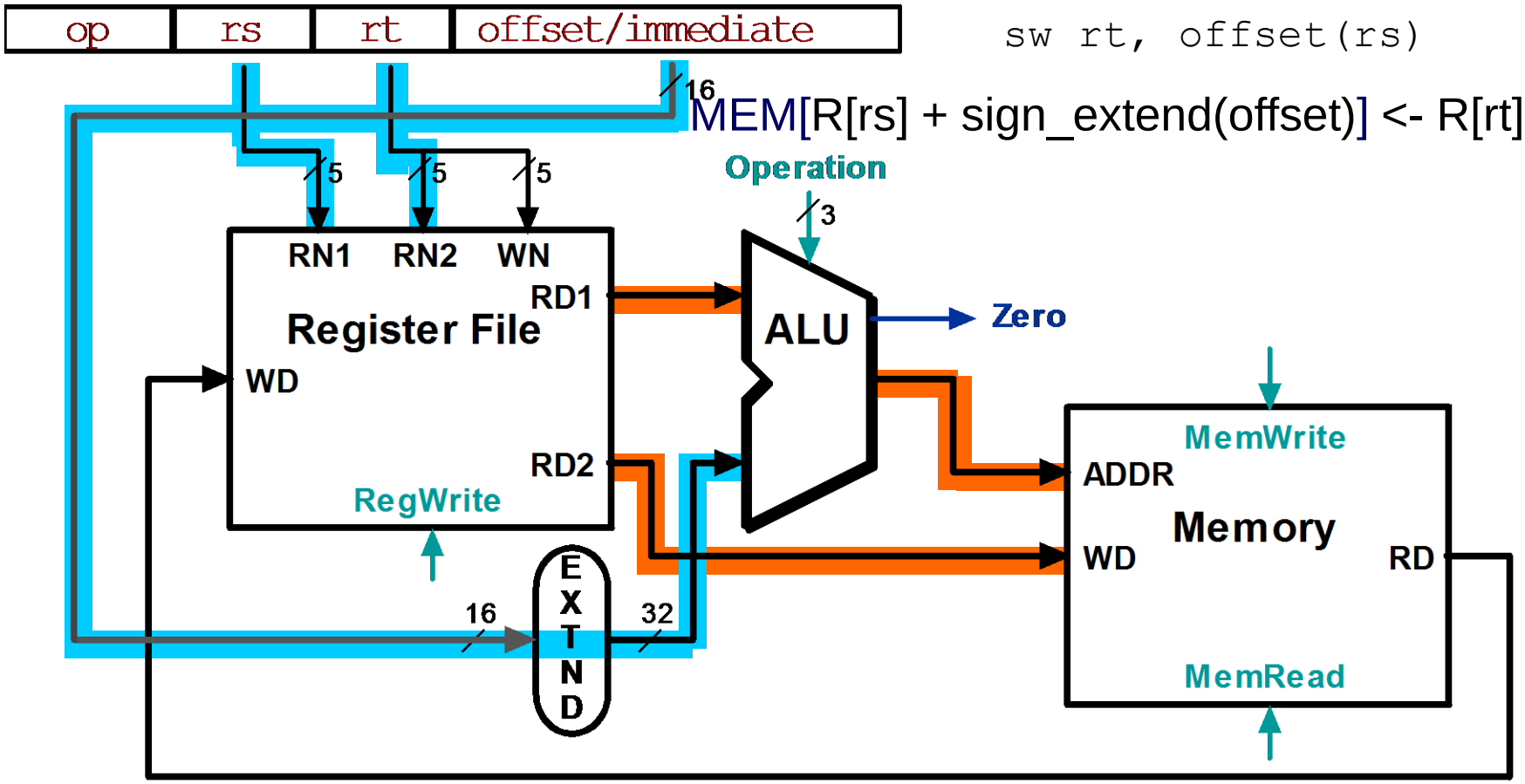
R[rd] <- R[rs] + R[rt];



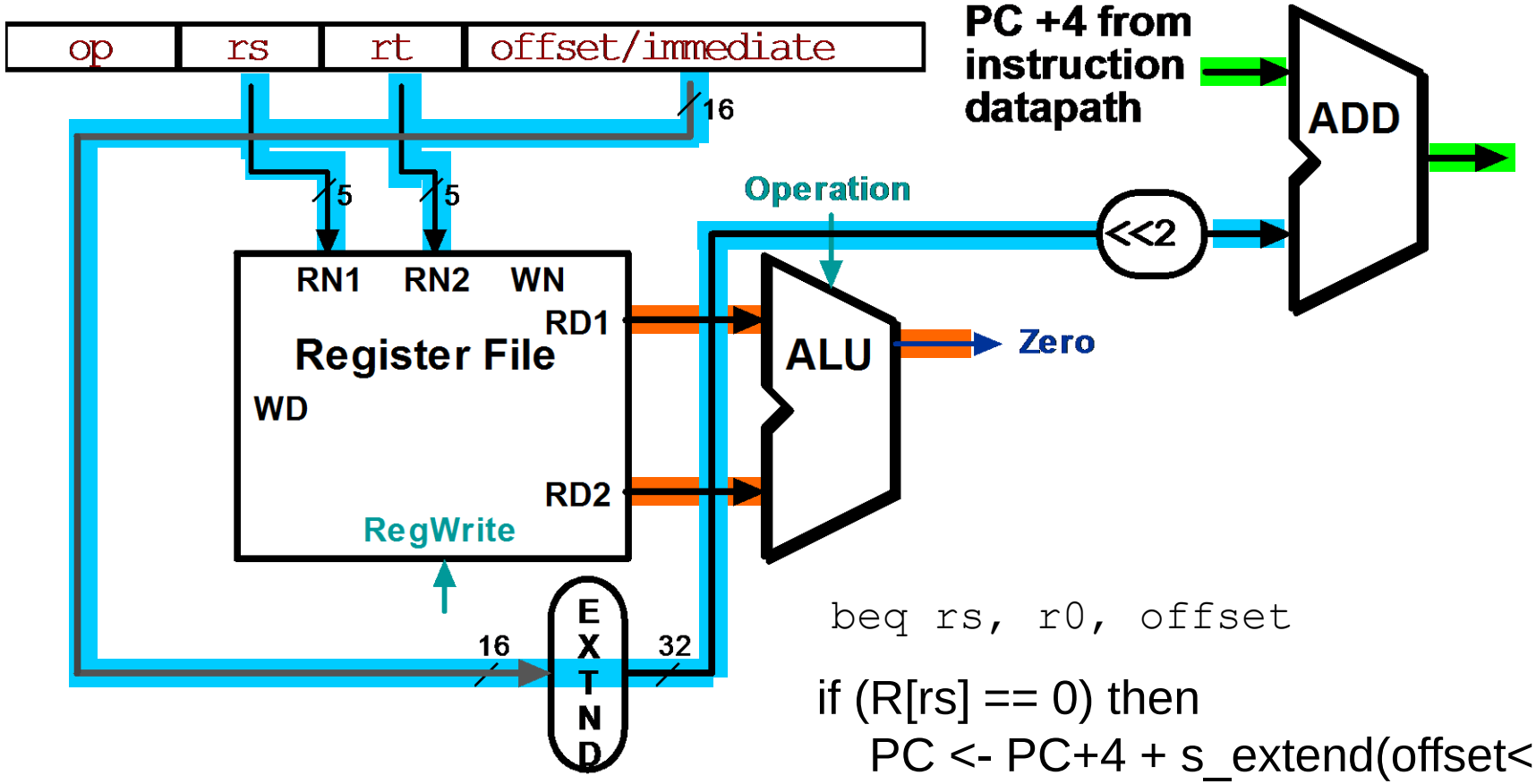
Datapath for Load/Store Instructions



Datapath for Load/Store Instructions



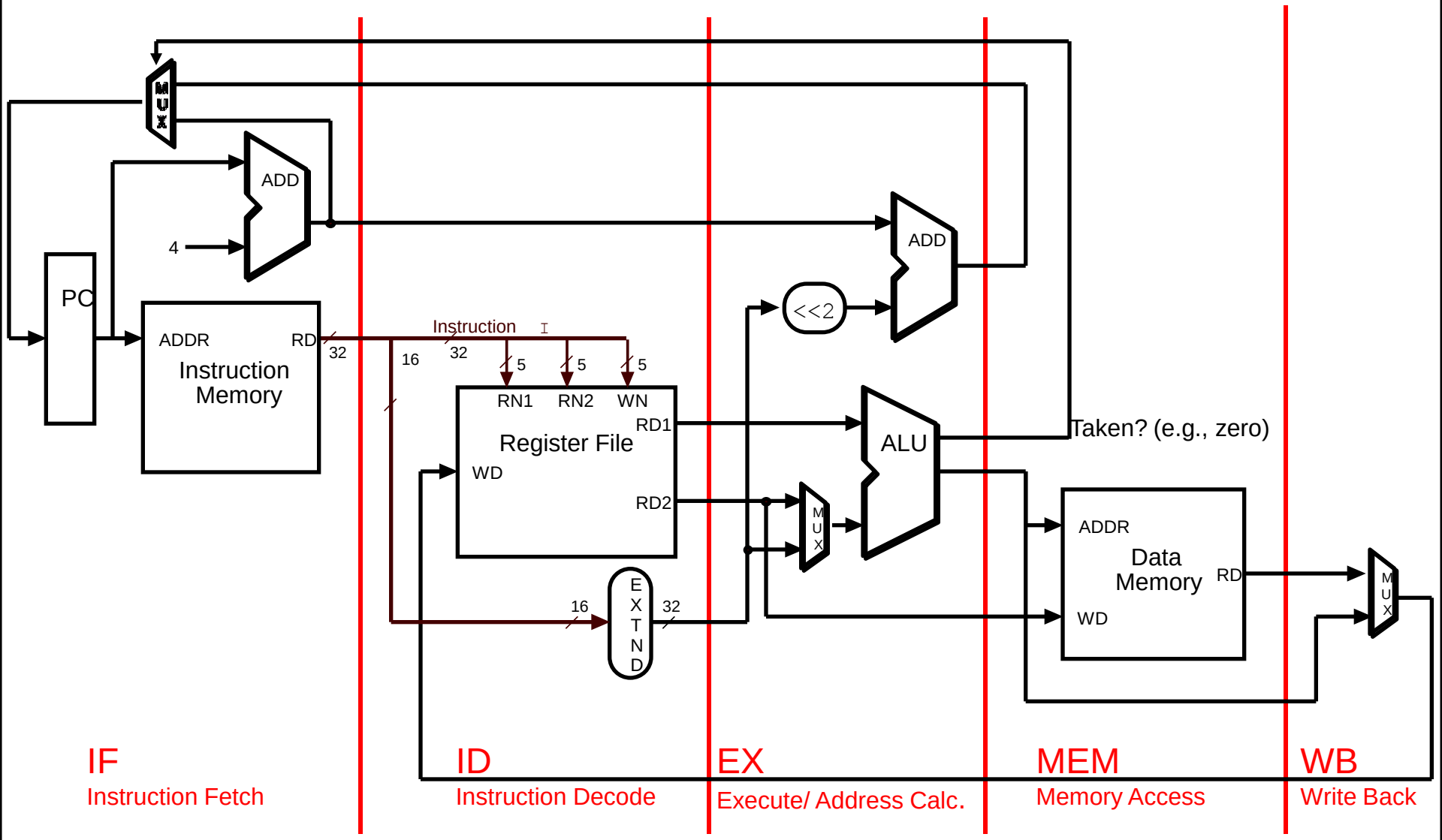
Datapath for Branch Instructions




```
beq rs, r0, offset
```

if (R[rs] == 0) then
    PC <- PC+4 + s_extend(offset<<2)

Basic MIPS Processor (More Details)



Pipelining - Key Idea

- Question: What happens if we break execution into multiple cycles?
- Answer: in the best case, we can start executing a new instruction on each clock cycle - this is pipelining
- Pipelining stages:
  - IF - Instruction Fetch
  - ID - Instruction Decode
  - EX - Execute / Address Calculation
  - MEM - Memory Access (read / write)
  - WB - Write Back (results into register file)

Simple MIPS Pipelined Integer Instruction Processing

	Clock Number → Time in clock cycles →								
Instruction Number	1	2	3	4	5	6	7	8	9
Instruction I	IF	ID	EX	MEM	WB				
Instruction I+1		IF	ID	EX	MEM	WB			
Instruction I+2			IF	ID	EX	MEM	WB		
Instruction I+3				IF	ID	EX	MEM	WB	
Instruction I +4					IF	ID	EX	MEM	WB

← Time to fill the pipeline →

MIPS Pipeline Stages:

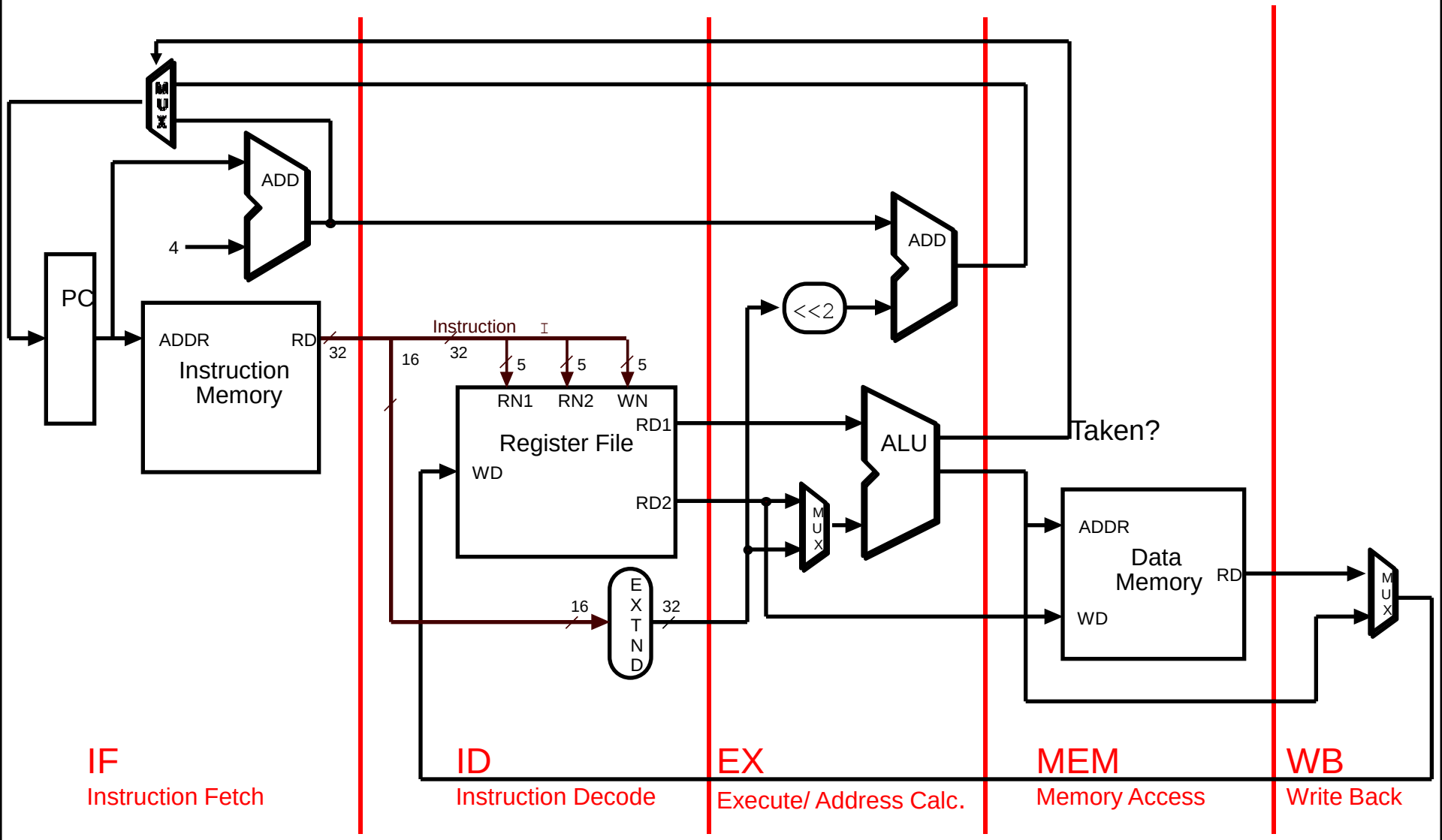
IF = Instruction Fetch
ID = Instruction Decode
EX = Execution
MEM = Memory Access
WB = Write Back

First instruction, I Completed

Last instruction, I+4 completed

How to design the datapaths to support pipelining?

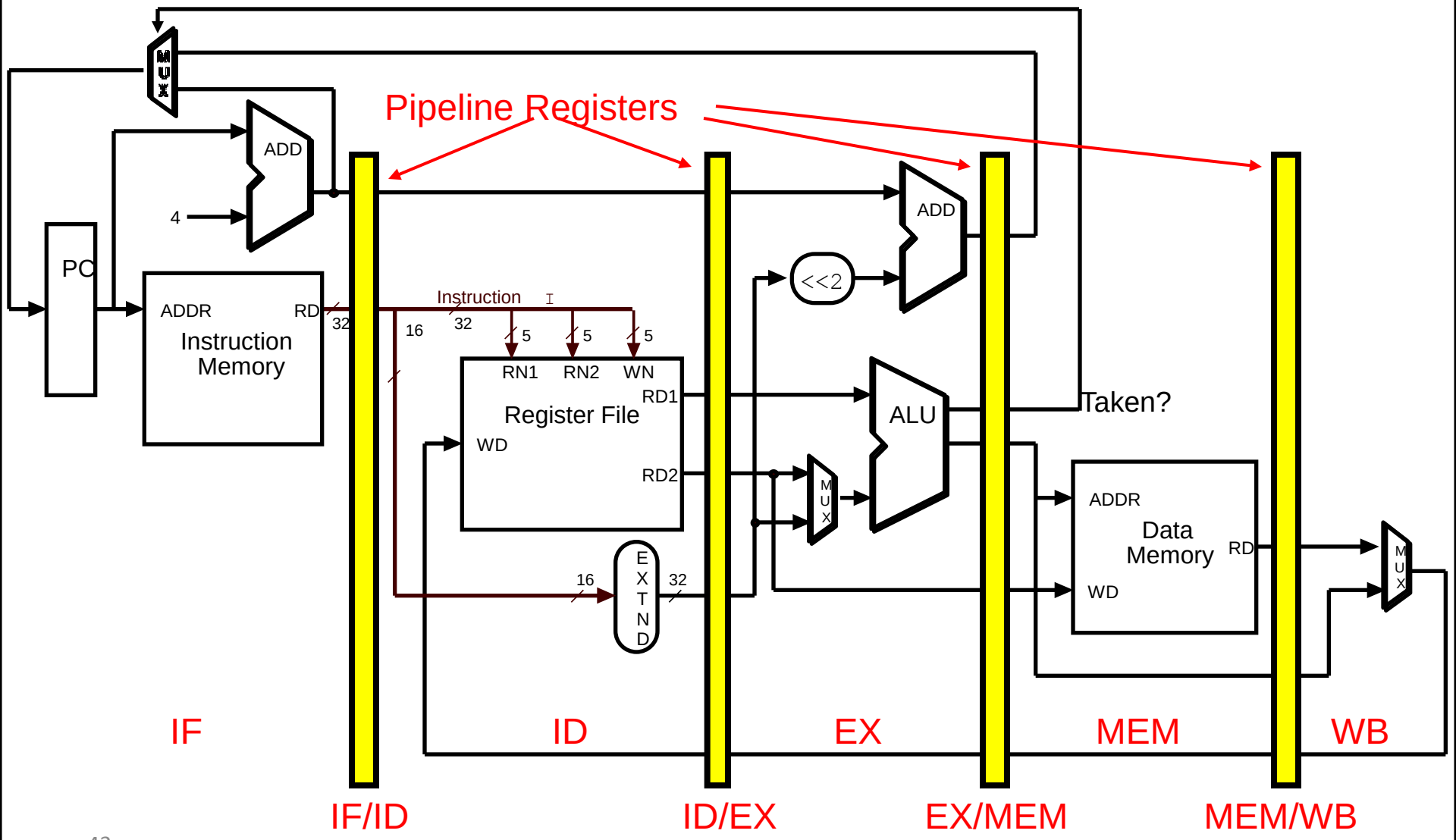
Basic MIPS Processor (More Details)



Pipeline Registers

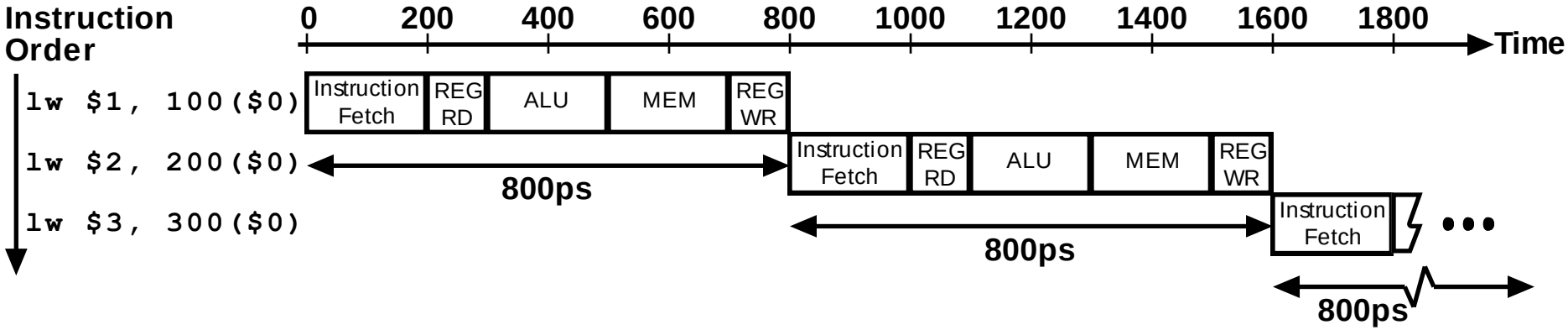
- Need pipeline registers (latches) between stages to hold data for instructions
- Pipeline registers are named with 2 stages (the stages that the register is "between.")
- ANY information needed in a later pipeline stage MUST be passed via a pipeline register
  - Example:IF/ID register gets
    - instruction
    - PC+4
- No register is needed after WB. Results from the WB stage are already stored in the register file.

Basic Pipelined Processor

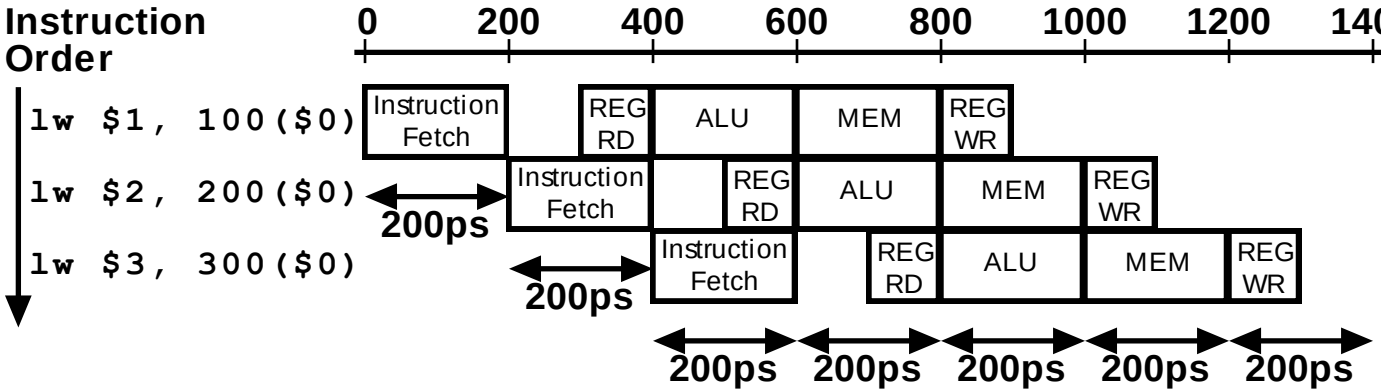


Non-Pipelined vs. Pipelined Execution

Non-Pipelined



Pipelined

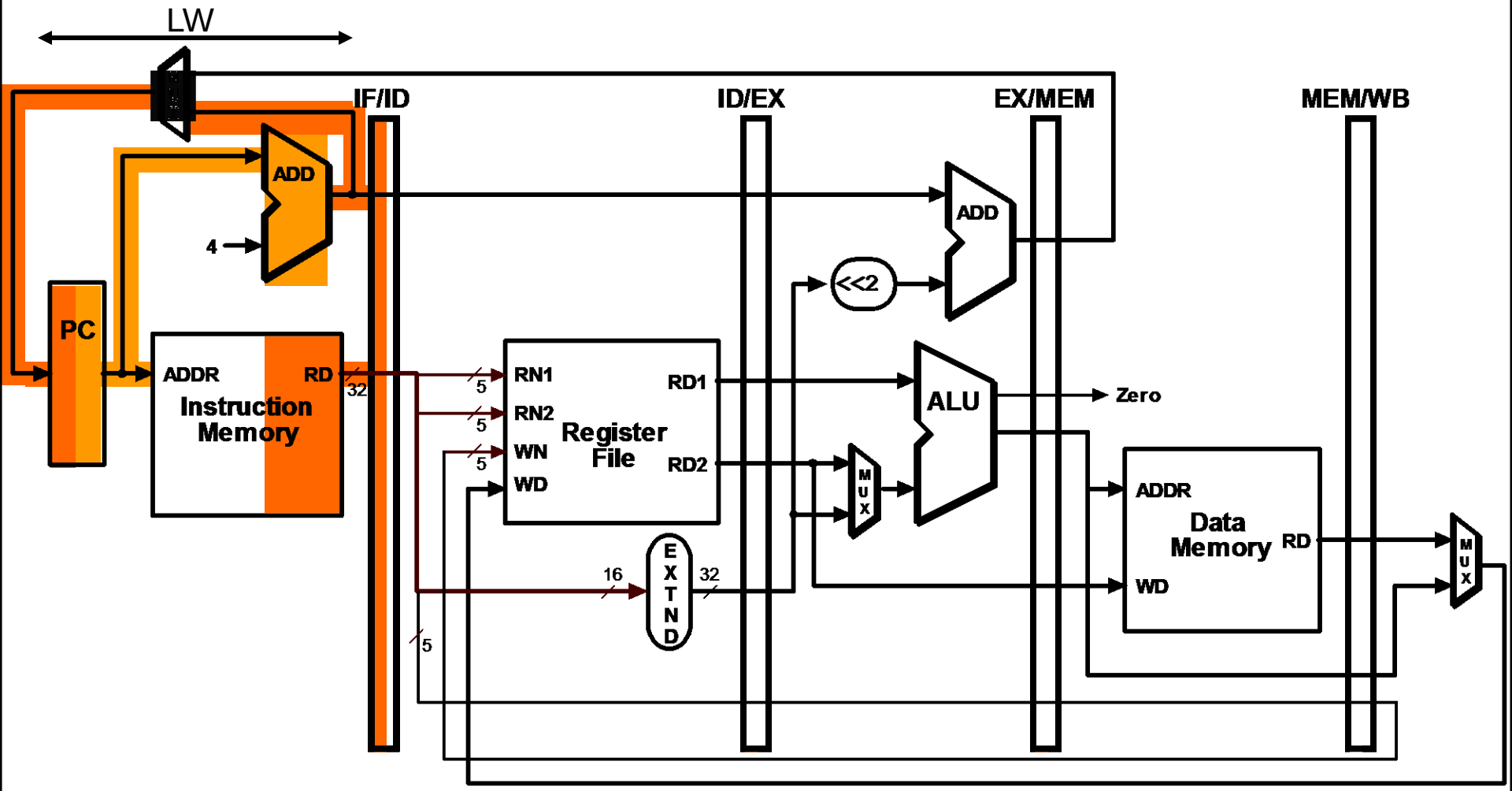


Pipelined Example - Executing Multiple Instructions

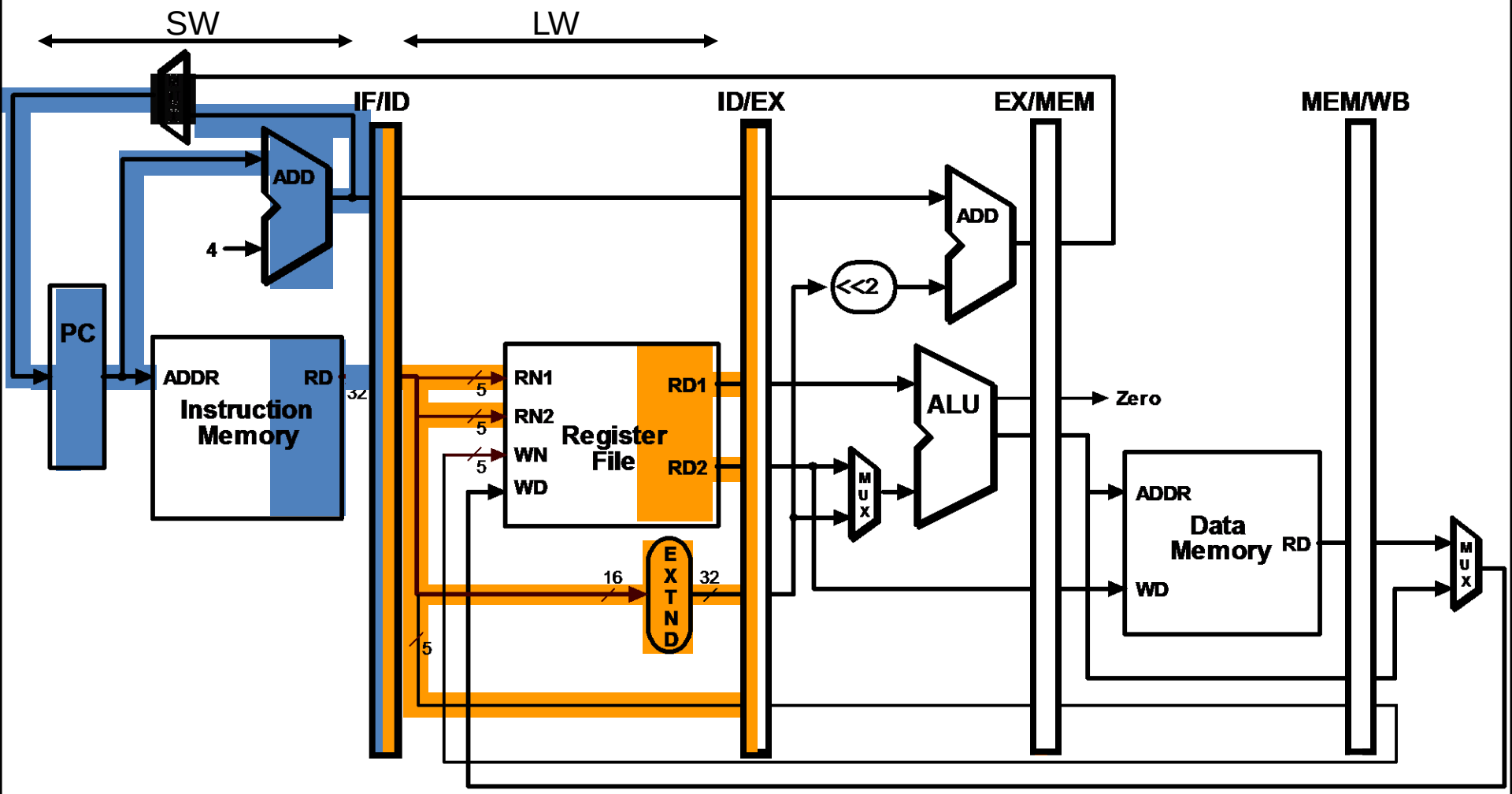
- Consider the following instruction sequence:

```
lw $r0, 10($r1)
sw $sr3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```

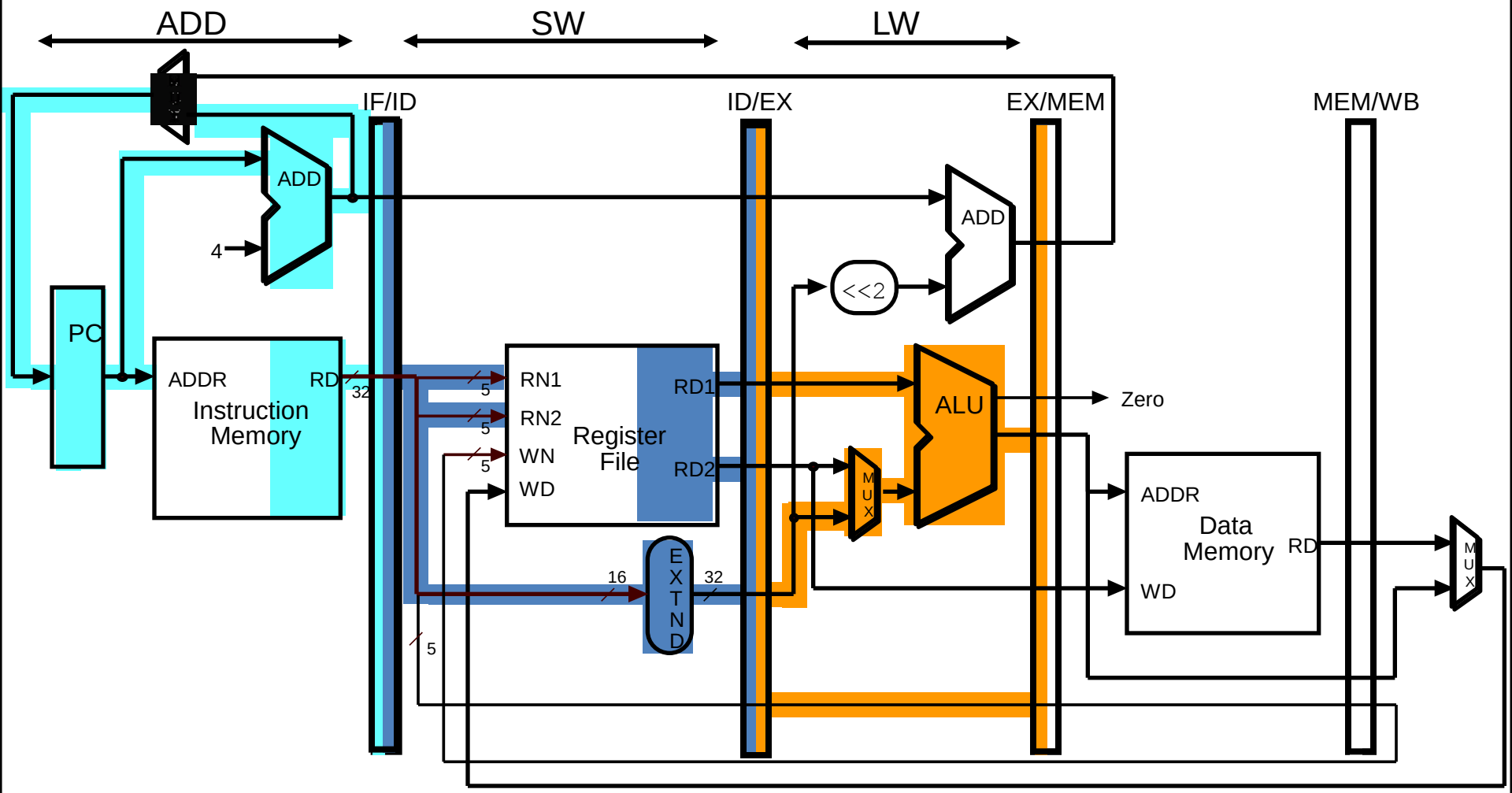
Executing Multiple Instructions
Clock Cycle 1



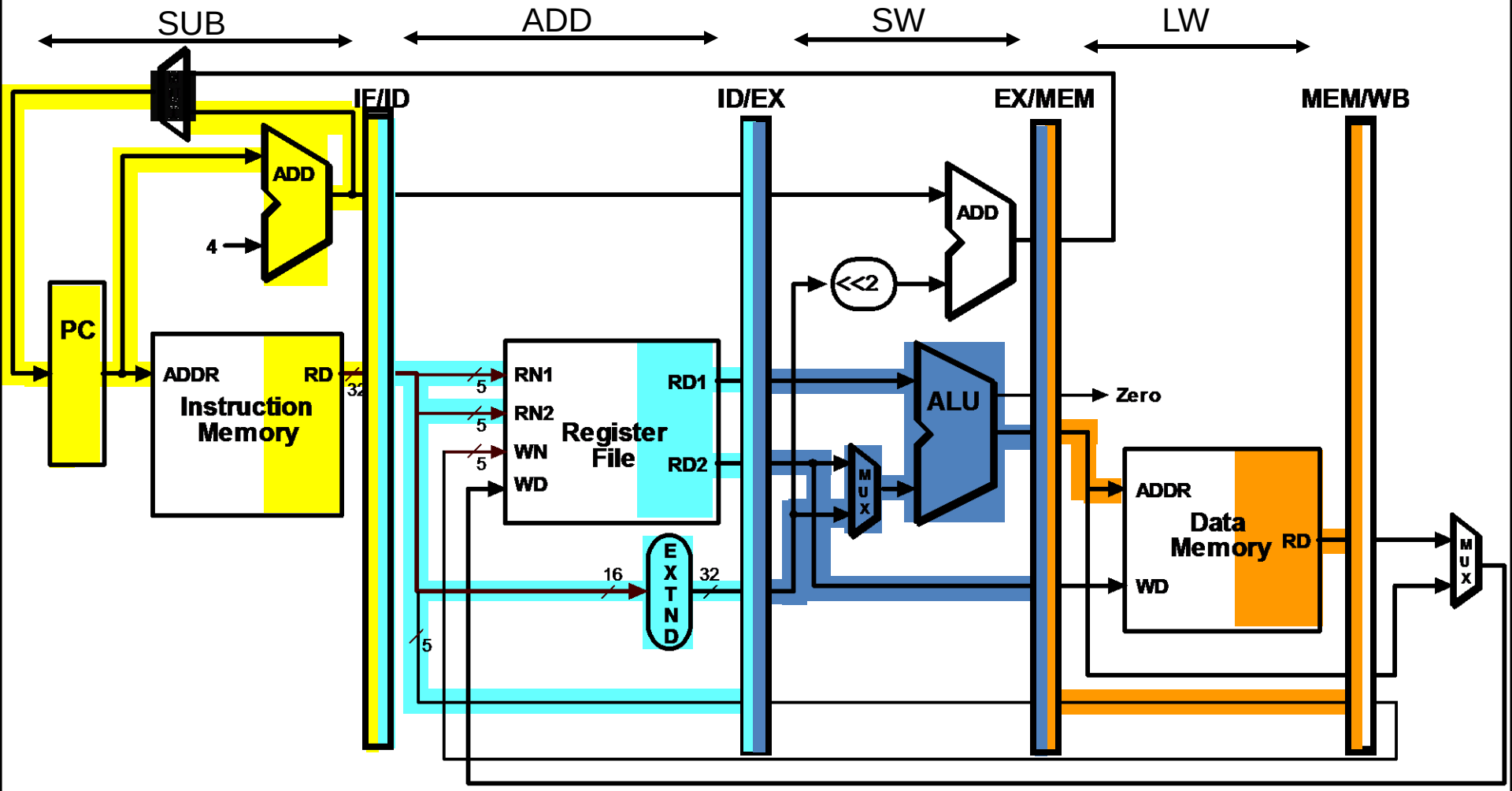
Executing Multiple Instructions
Clock Cycle 2



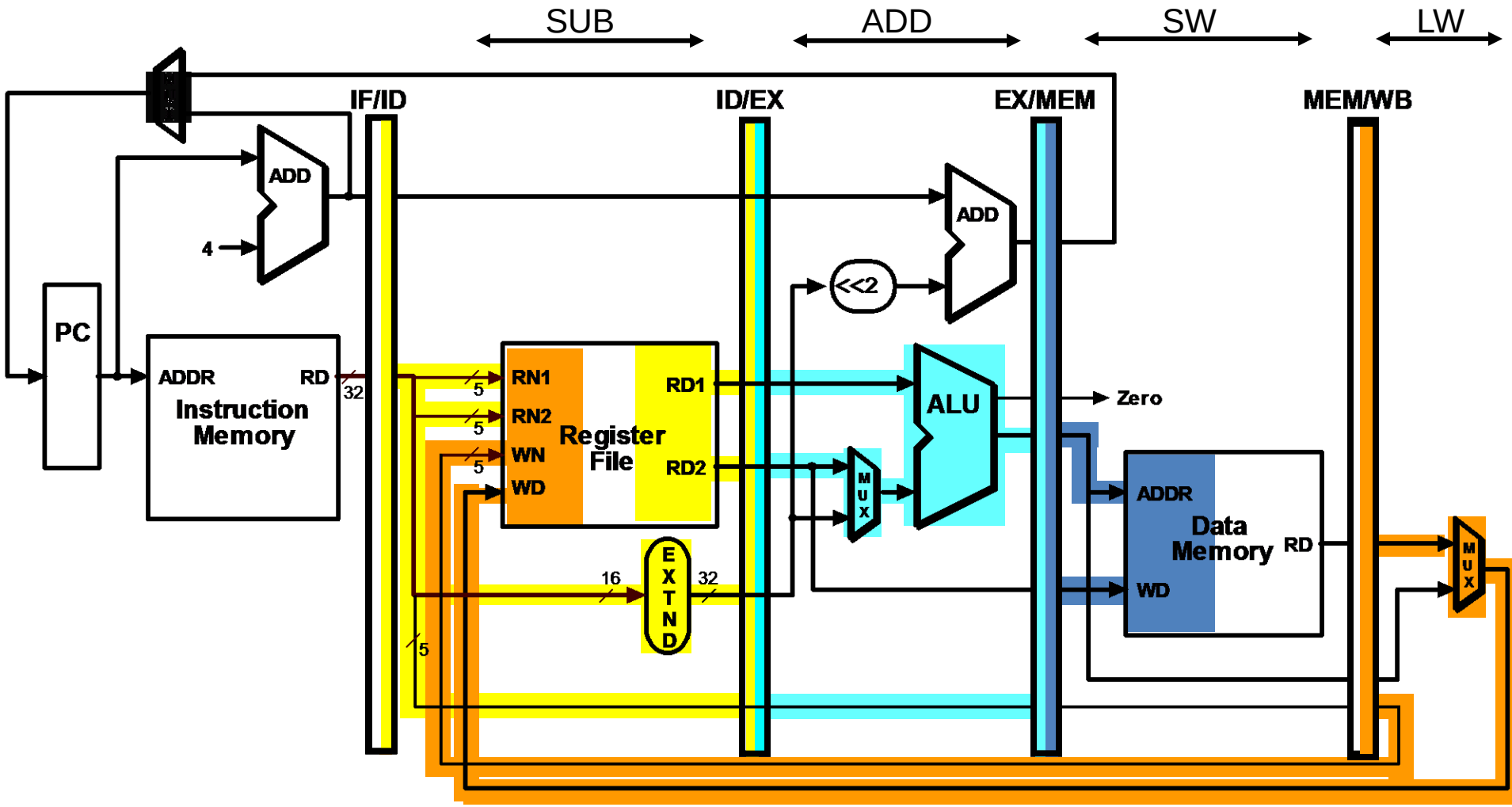
Executing Multiple Instructions
Clock Cycle 3



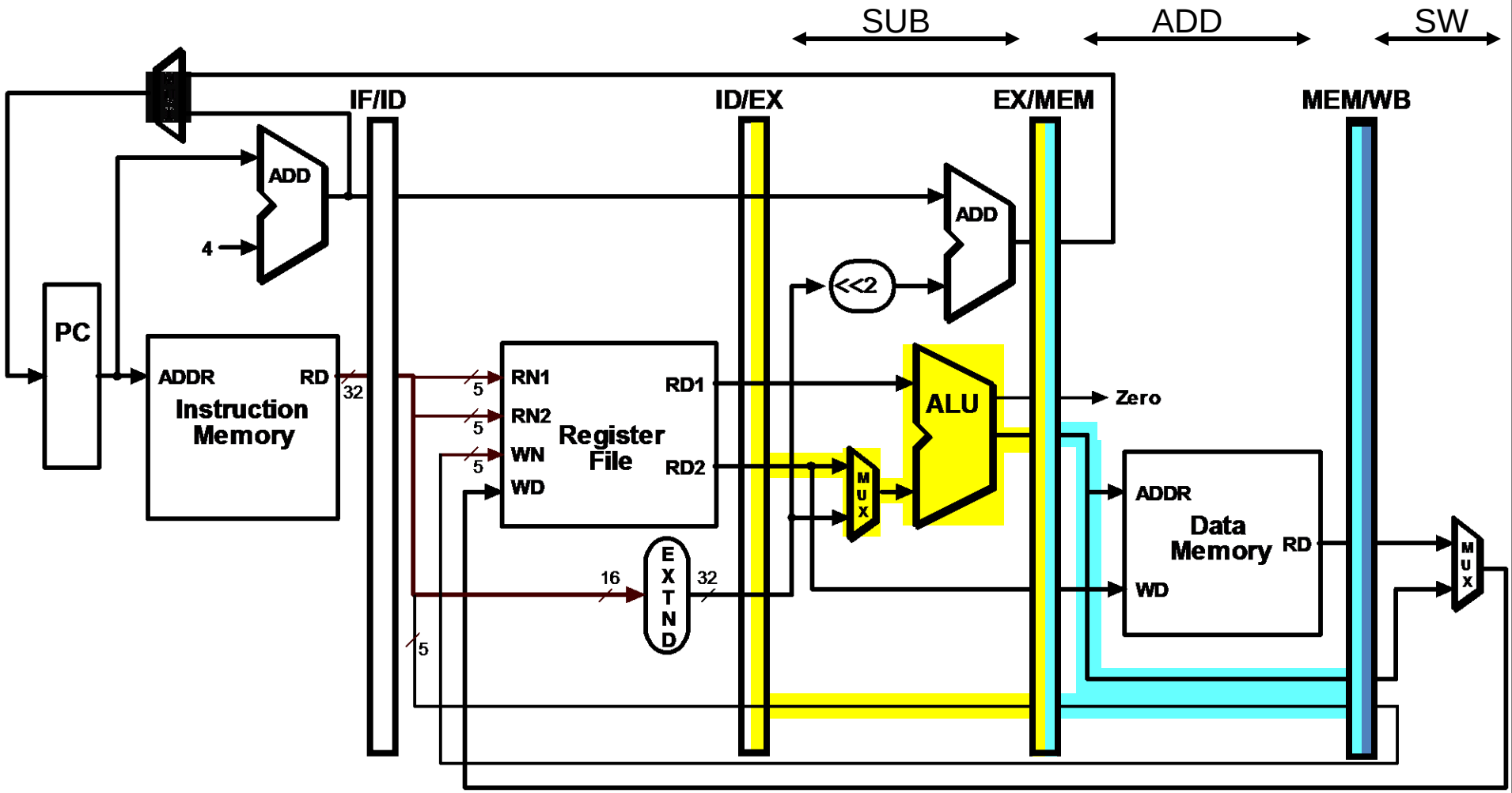
Executing Multiple Instructions
Clock Cycle 4



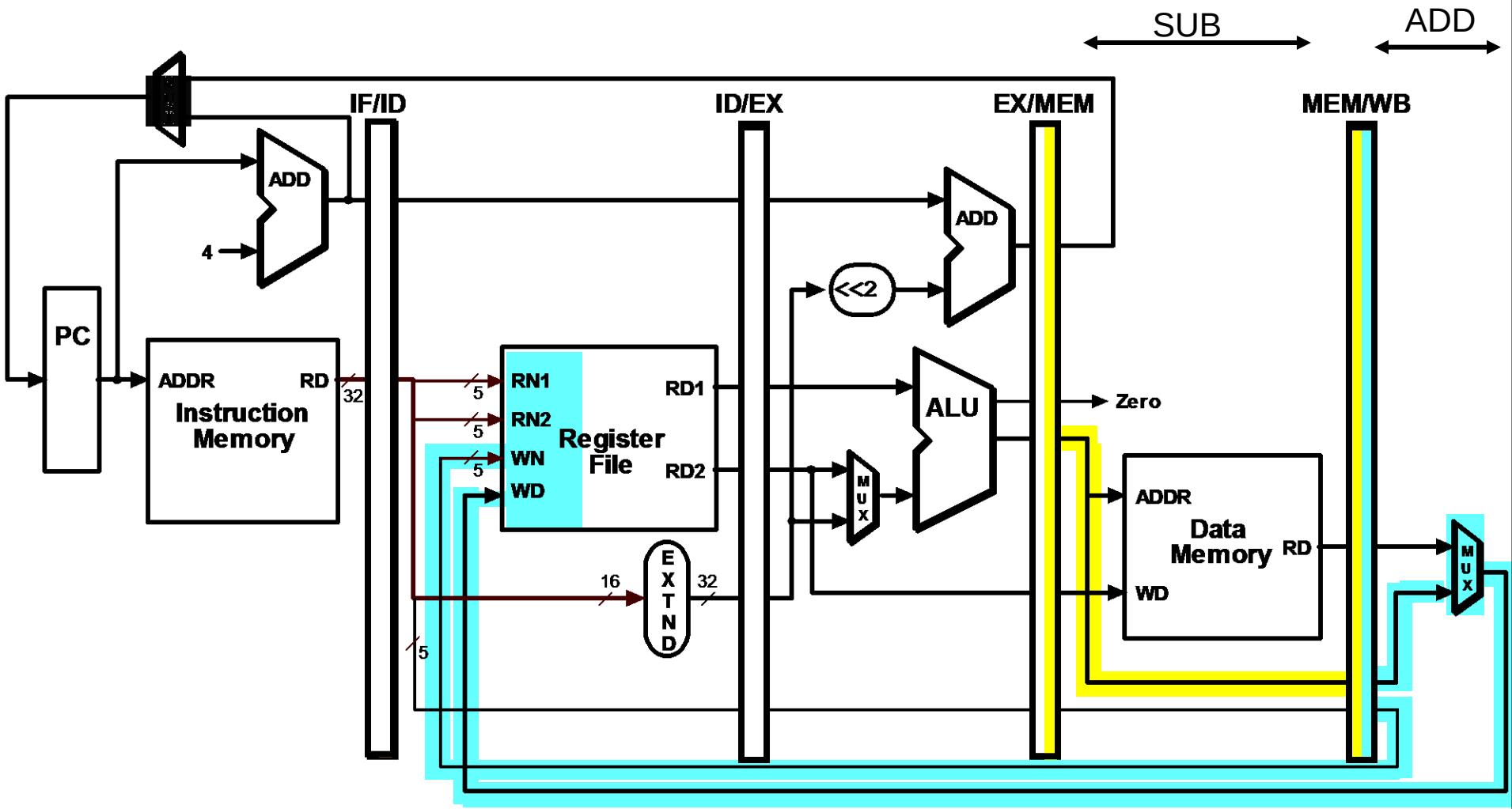
Executing Multiple Instructions
Clock Cycle 5



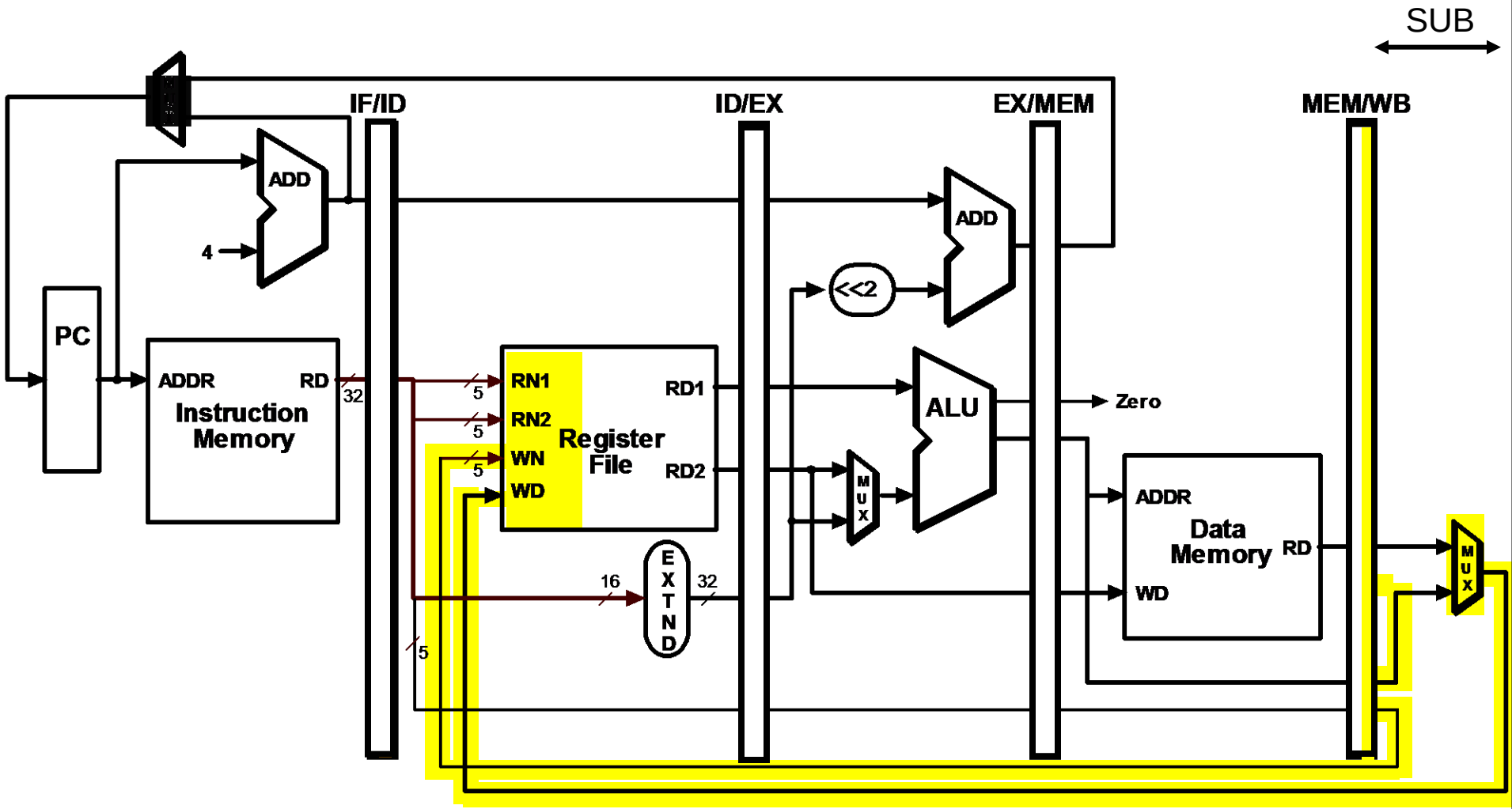
Executing Multiple Instructions
Clock Cycle 6



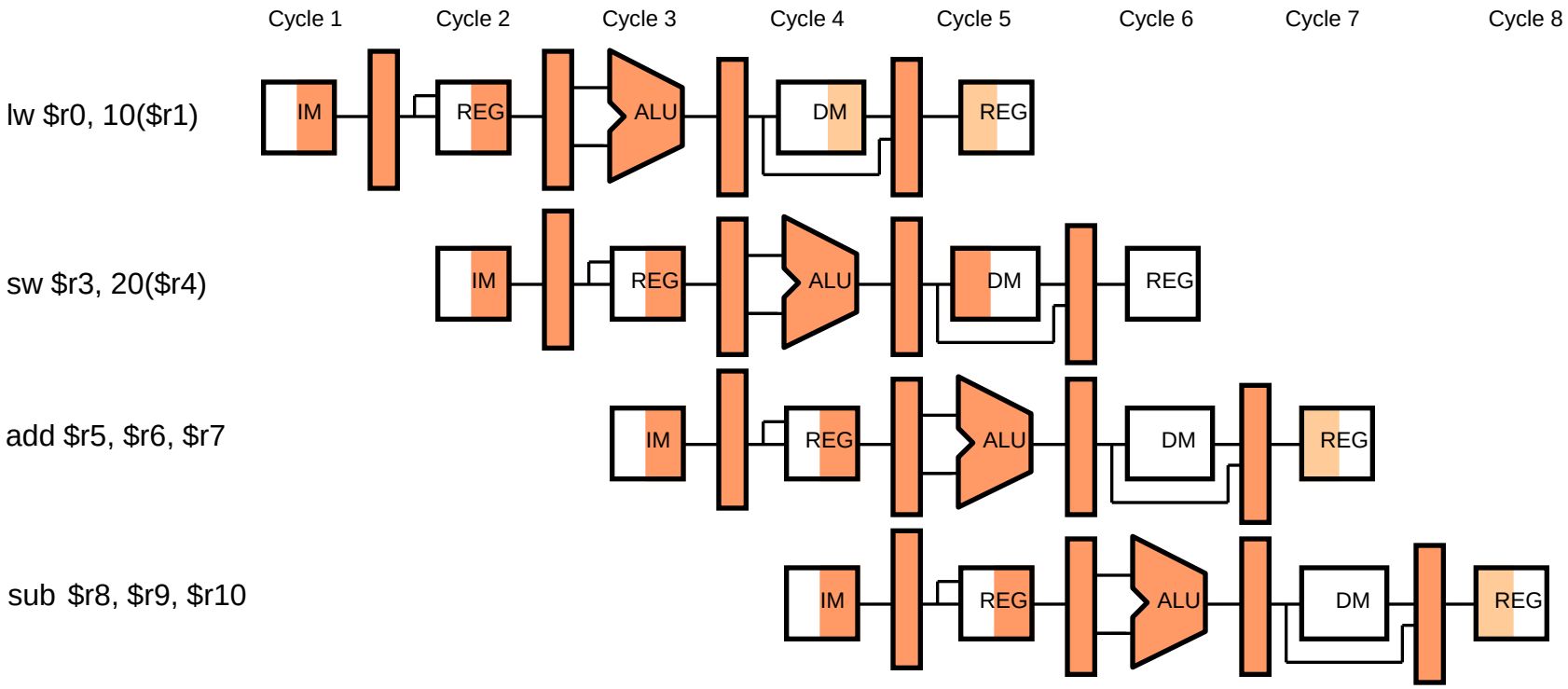
Executing Multiple Instructions
Clock Cycle 7



Executing Multiple Instructions
Clock Cycle 8



Alternative View - Multicycle Diagram



Pipeline Performance

- How much does pipelining improve the performance? How to calculate throughput and latency of a pipelined processor?
- How to design the pipeline to increase speedup?
- …

Throughput, latency, hazards

Pipelining Performance Example

- Example: For an unpipelined CPU:
  - **Clock cycle = 1ns, 4 cycles for ALU operations and branches and 5 cycles for memory operations with instruction frequencies of 40%, 20% and 40%, respectively.**
  - **If pipelining adds 0.2 ns to the machine clock cycle then the speedup in instruction execution from pipelining is:**

Non-pipelined Average instruction execution time = Clock cycle x Average CPI

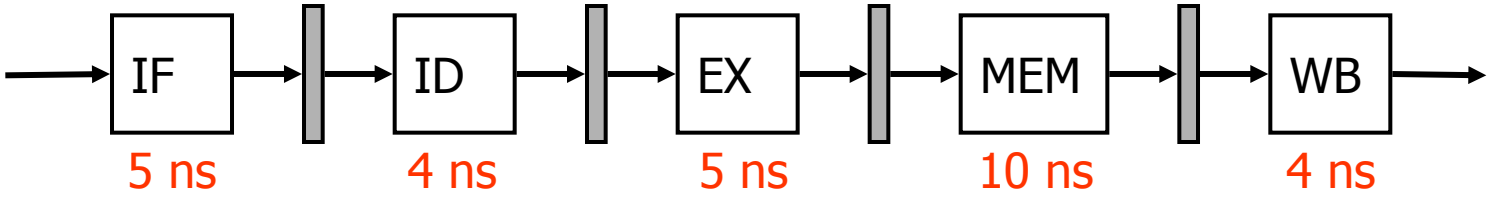
= 1 ns x ((40% + 20%) x 4 + 40%x 5) = 1 ns x 4.4 = 4.4 ns

In the 5-stage pipelined implementation, the latency is 6ns, but the average instruction execution time is: 1 ns + 0.2 ns = 1.2 ns

Speedup from pipelining = $\frac{\text{Instruction time unpipelined}}{\text{Instruction time pipelined}}$

= 4.4 ns / 1.2 ns = 3.7 times faster

Pipeline Throughput and Latency: A More realistic Examples

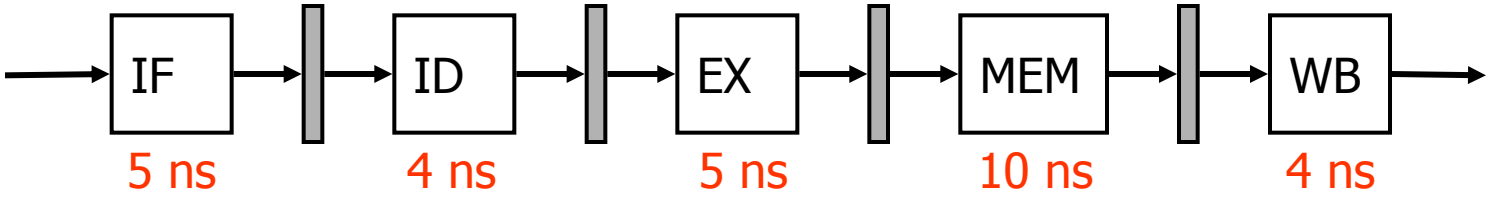


Consider the pipeline above with the indicated delays. We want to know what the *pipeline throughput* and the *pipeline latency are*.

Pipeline throughput: instructions completed per second.

Pipeline latency: how long does it take to execute a single instruction in the pipeline.

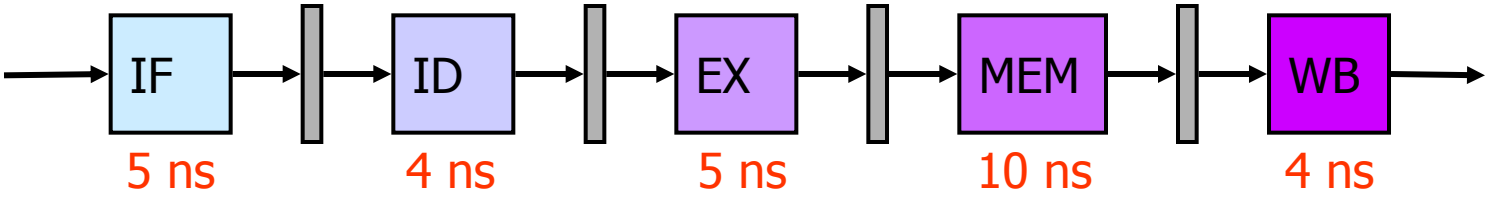
Pipeline Throughput and Latency



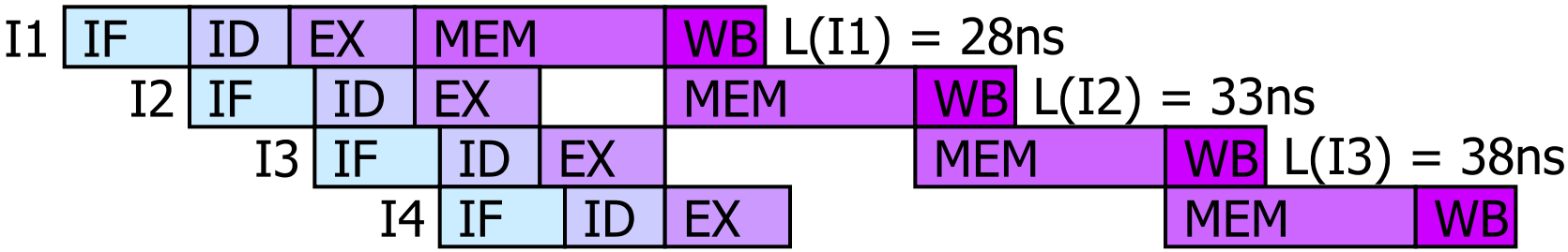
Pipeline latency: how long does it take to execute an instruction in the pipeline.

$$L = lat(IF) + lat(ID) + lat(EX) + lat(MEM) + lat(WB)$$
$$= 5ns + 4ns + 5ns + 10ns + 4ns = 28ns$$

Pipeline Throughput and Latency

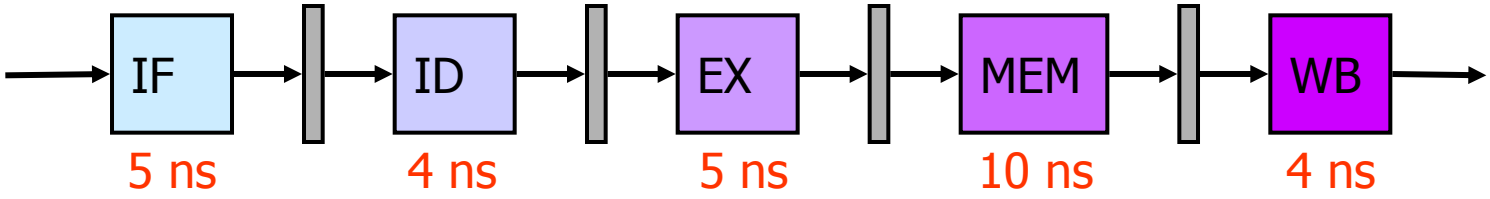


Simply adding the latencies to compute the pipeline latency only would work for an isolated instruction

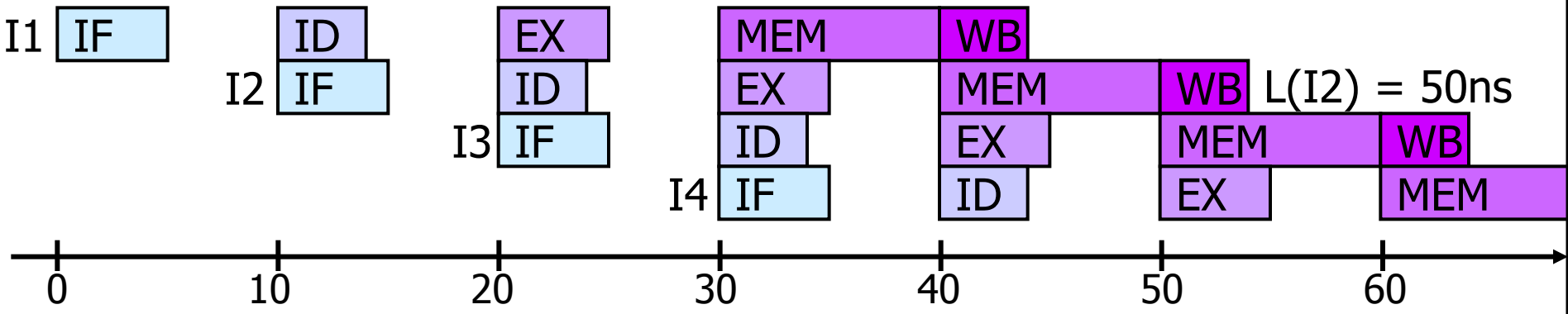


We are in trouble! The latency is not constant. This happens because this is an unbalanced pipeline. The solution is to make every state the same length as the longest one.

Synchronous Pipeline Throughput and Latency

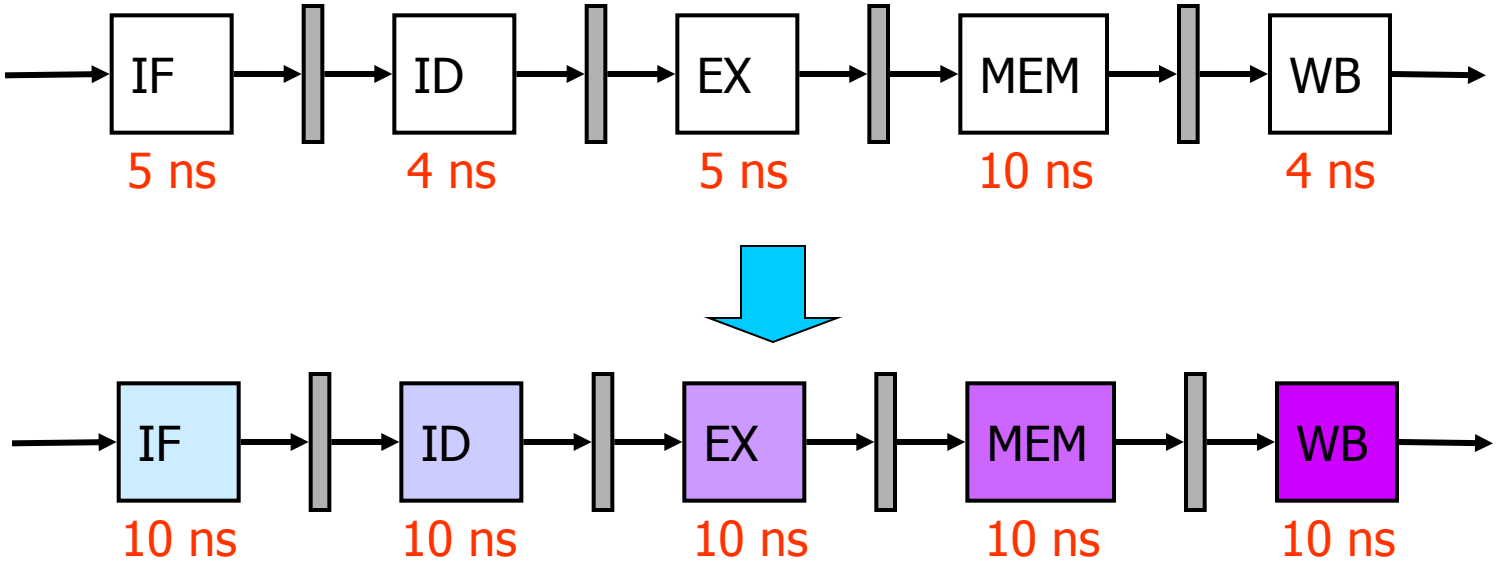


The slowest pipeline stage also limits the latency!!



L(I1) = L(I2) = L(I3) = L(I4) = 50ns

Pipeline Throughput and Latency



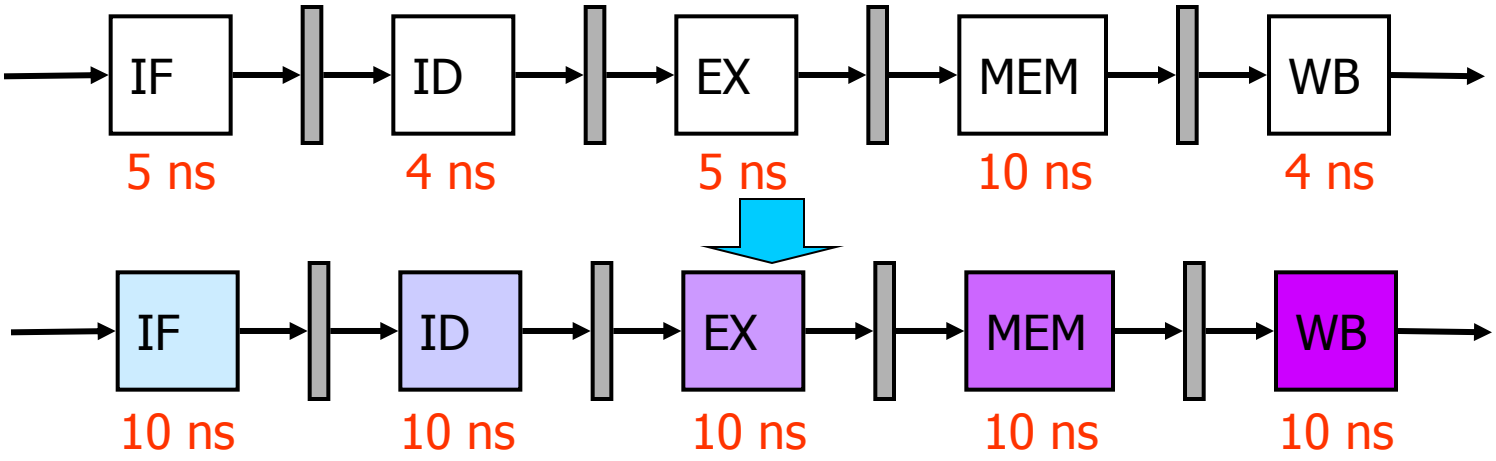
Pipeline throughput: how frequently are instructions completed?

$$= 1 instr / \max\left[lat(IF), lat(ID), lat(EX), lat(MEM), lat(WB)\right]$$

$$= 1 instr / \max\left[5ns, 4ns, 5ns, 10ns, 4ns\right]$$

$$= 1 instr / 10ns \quad (ignoring\ pipeline\ register\ overhead)$$

Pipeline Throughput and Latency



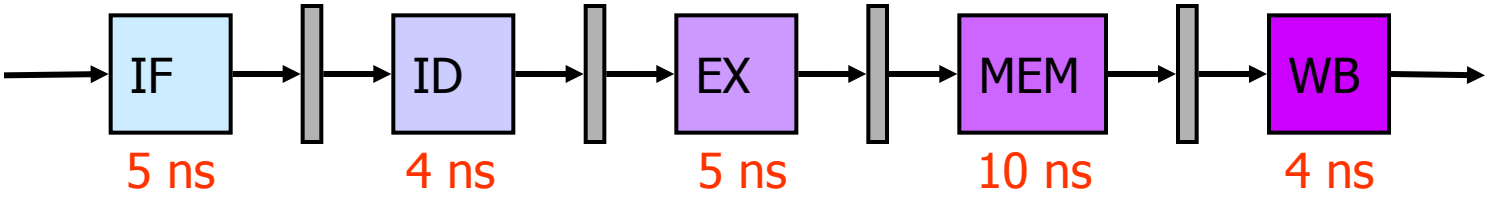
How long does it take to execute (issue) 20000 instructions in this pipeline? (disregard latency, bubbles caused by branches, cache misses, hazards, fill, drain)

$$ExecTime_{pipe} = 20000 \times 10ns = 200000ns = 200\mu s$$

How long would it take using the same modules without pipelining?

$$ExecTime_{non-pipe} = 20000 \times 28ns = 560000ns = 560\mu s$$

Pipeline Throughput and Latency



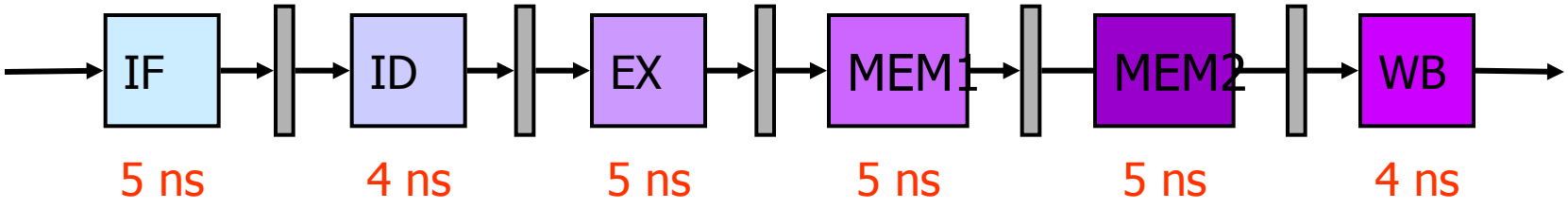
Thus the speedup of the pipeline is:

$$Speedup_{pipe} = \frac{ExecTime_{non-pipe}}{ExecTime_{pipe}} = \frac{560\mu s}{200\mu s} = 2.8$$

How can we improve this pipeline design?

We need to reduce the unbalance to increase the clock speed.

Pipeline Throughput and Latency



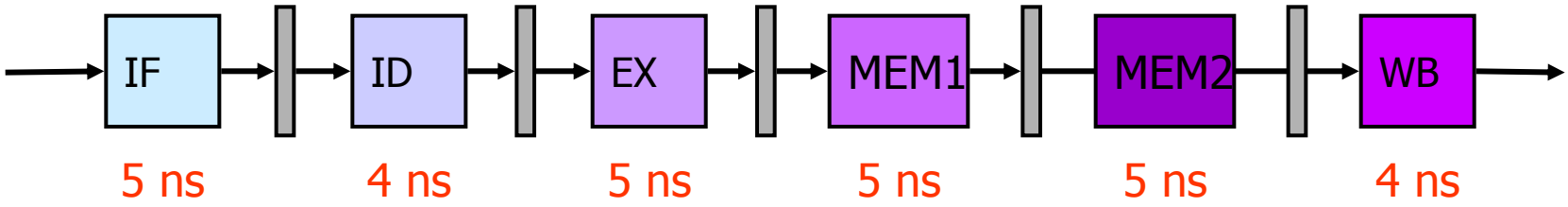
Now we have one more pipeline stage, but the maximum latency of a single stage is reduced in half.

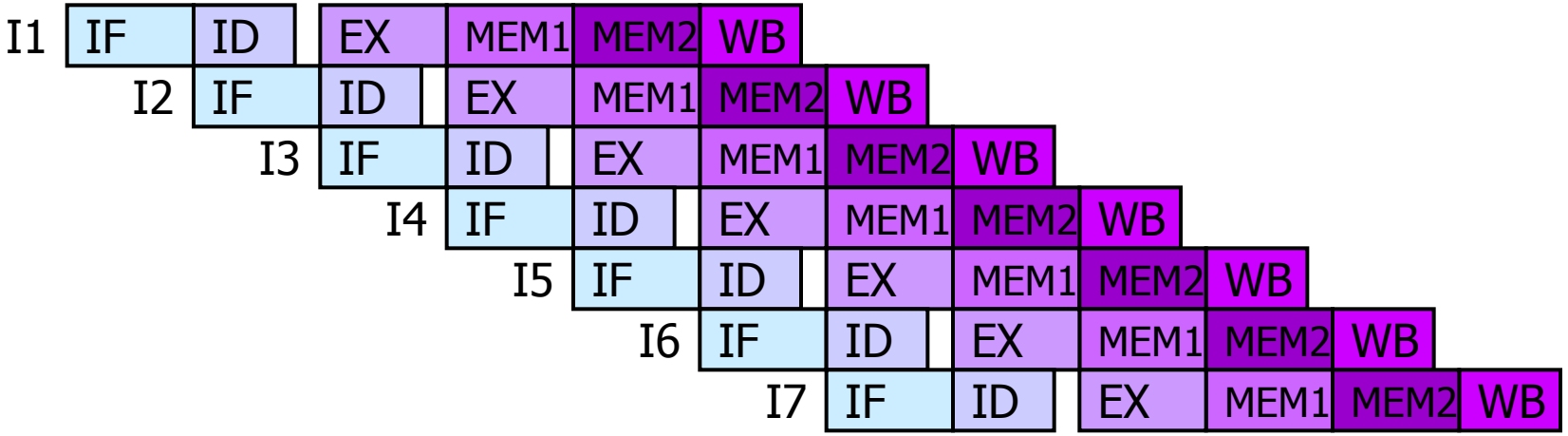
$$T = 1instr / \max(lat(IF), lat(ID), lat(EX), lat(MEM1), lat(MEM2), lat(WB))$$
$$= 1instr / \max(5ns, 4ns, 5ns, 5ns, 5ns, 4ns)$$
$$= 1instr / 5ns$$

The new latency for a single instruction is:

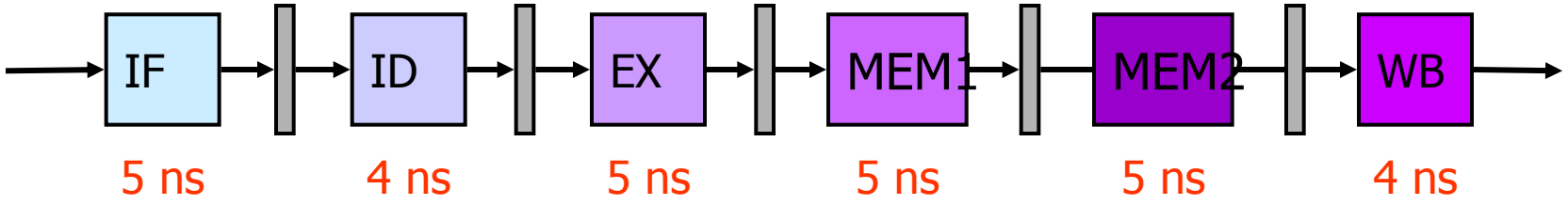
$$L = 6 \times 5ns = 30ns$$

Pipeline Throughput and Latency





Pipeline Throughput and Latency



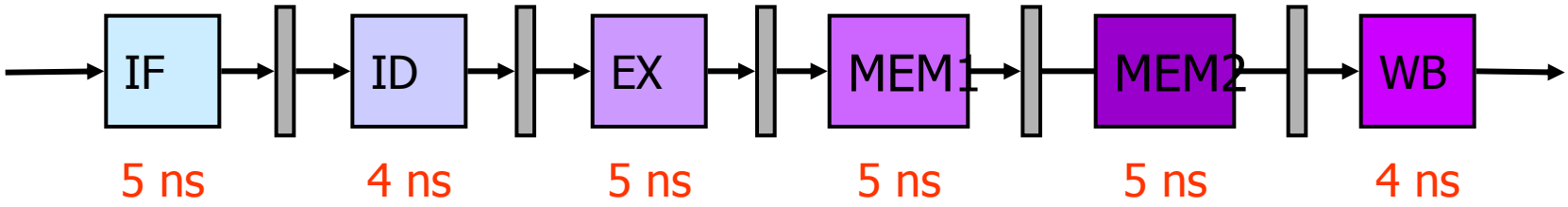
How long does it take to execute 20000 instructions in this pipeline? (disregard stalls/bubbles caused by branches, cache misses, etc, for now)

$$ExecTime_{pipe} = 20000 \times 5ns = 100000ns = 100\mu s$$

Thus the speedup that we get from the pipeline is:

$$Speedup_{pipe} = \frac{ExecTime_{non-pipe}}{ExecTime_{pipe}} = \frac{560\mu s}{100\mu s} = 5.6$$

Pipeline Throughput and Latency



What have we learned from this example?

1. It is important to balance the delays in the stages of the pipeline

2. The throughput of a pipeline is 1/max(stage_latency).

3. The latency is N×max(stage_latency), where N is the number of stages in the pipeline.