EECS 322 Computer Architecture
Introduction to Pipelining

Based
on Dave
Patterson
slides

Instructor: Francis G. Wolff
wolff@eecs.cwru.edu
Case Western Reserve University

This presentation uses powerpoint animation: please viewshow

Comparison

CISC	RISC
Any instruction may reference memory	Only load/store references memory
Many instructions & addressing modes	Few instructions & addressing modes
Variable instruction formats	Fixed instruction formats
Single register set	Multiple register sets
Multi-clock cycle instructions	Single-clock cycle instructions
Micro-program interprets instructions	Hardware (FSM) executes instructions
Complexity is in the micro-program	Complexity is in the complier
Less to no pipelining	Highly pipelined
Program code size small	Program code size large

Pipelining (Designing…,M.J.Quinn, ‘87)

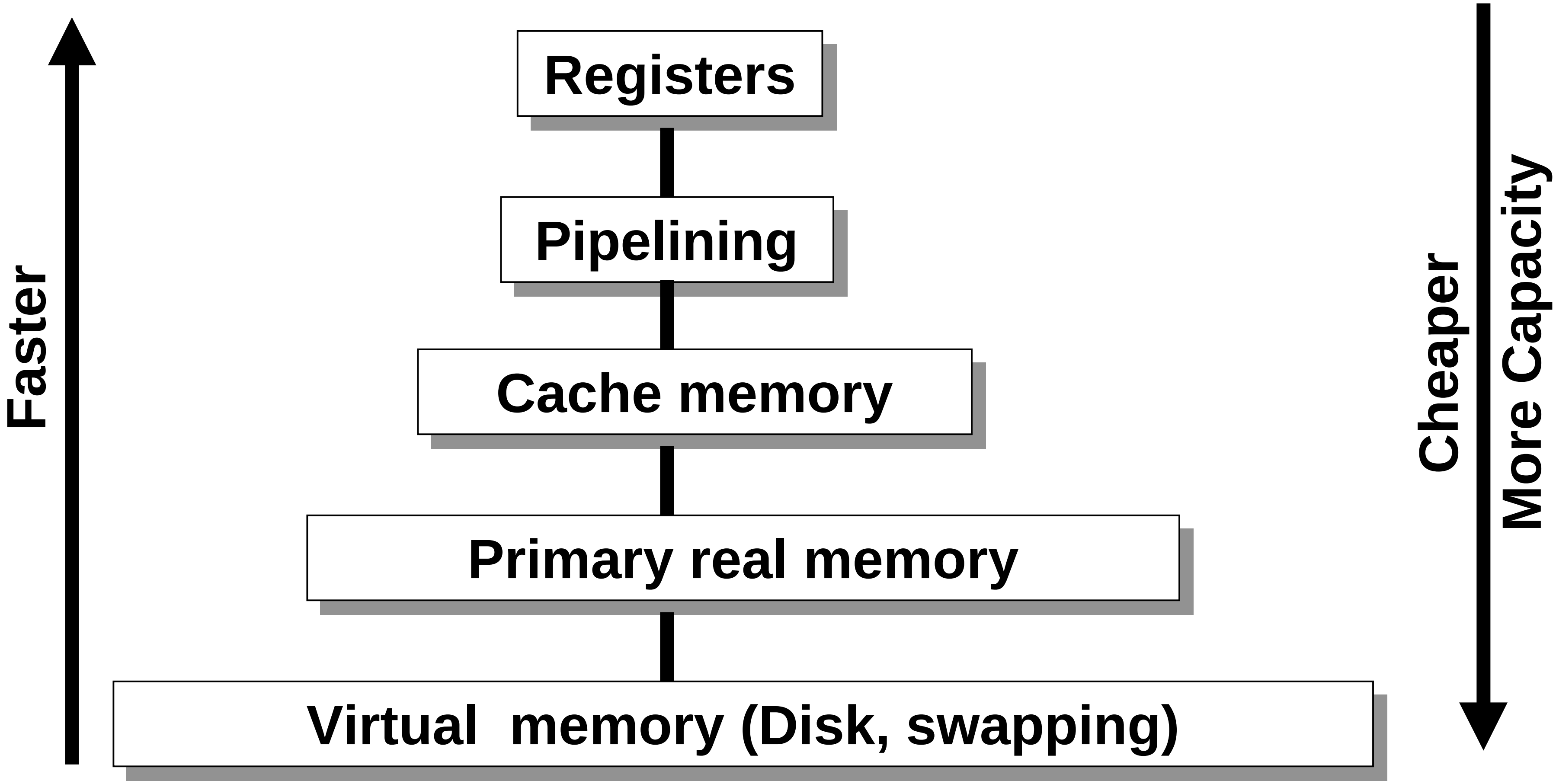
Instruction Pipelining is the use of pipelining to allow more than one instruction to be in some stage of execution at the same time.

Cache memory is a small, fast memory unit used as a buffer between a processor and primary memory

Ferranti ATLAS (1963):

- **Pipelining reduced the average time per instruction by 375%**
- **Memory could not keep up with the CPU, needed a cache.**

Memory Hierarchy



Pipelining versus Parallelism (Designing…,M.J.Quinn, ‘87)

Most high-performance computers exhibit a great deal of concurrency.

However, it is not desirable to call every modern computer a parallel computer.

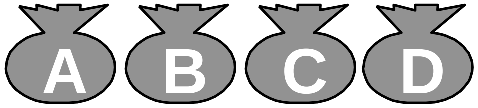
Pipelining and parallelism are 2 methods used to achieve concurrency.

Pipelining increases concurrency by dividing a computation into a number of steps.

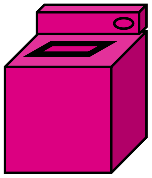
Parallelism is the use of multiple resources to increase concurrency.

Pipelining is Natural!

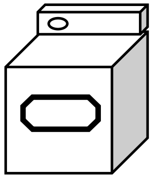
- Laundry Example



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

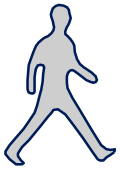


- Washer takes 30 minutes



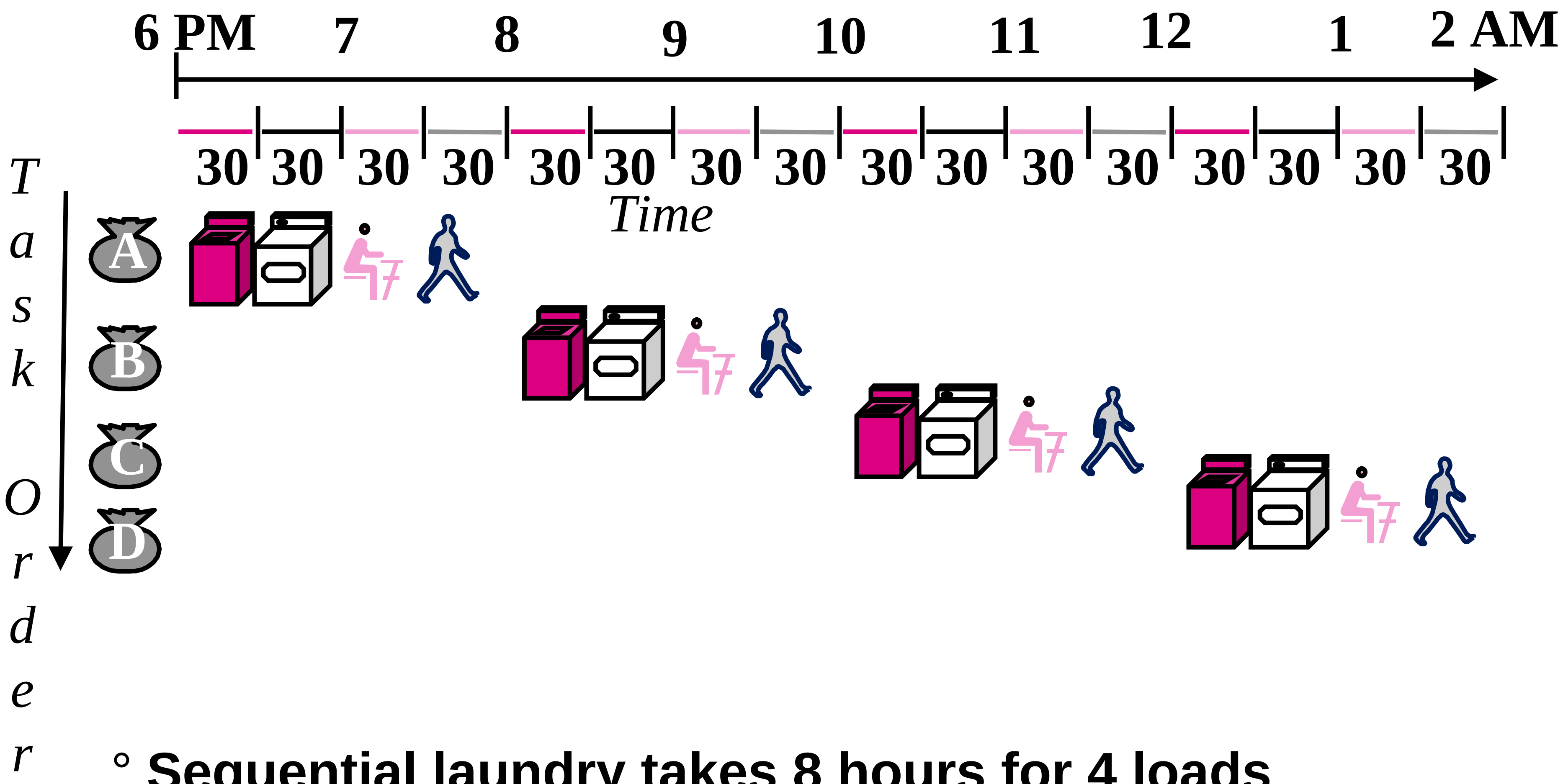
- Dryer takes 30 minutes

- “Folder” takes 30 minutes



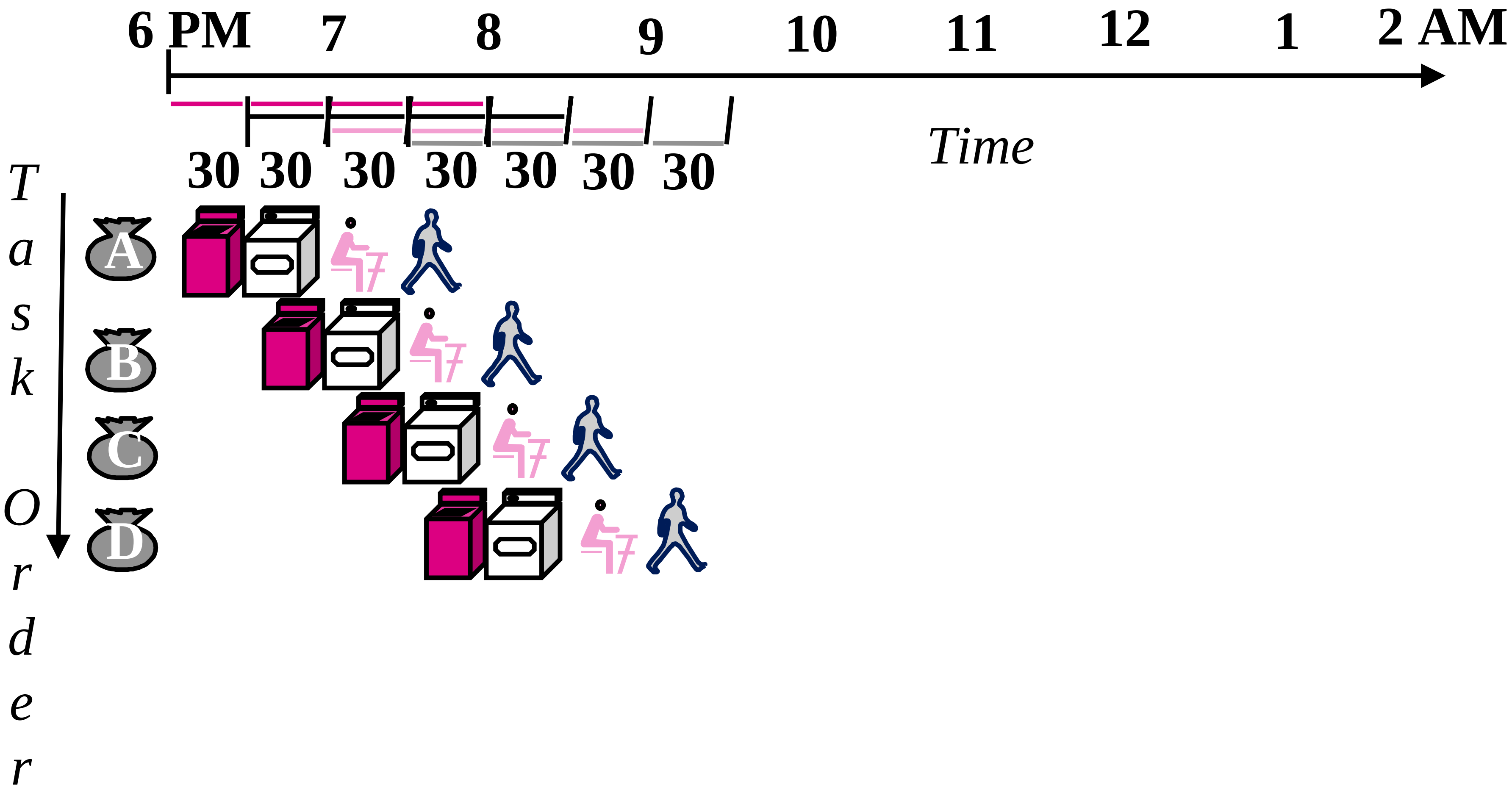
- “Stasher” takes 30 minutes to put clothes into drawers

Sequential Laundry



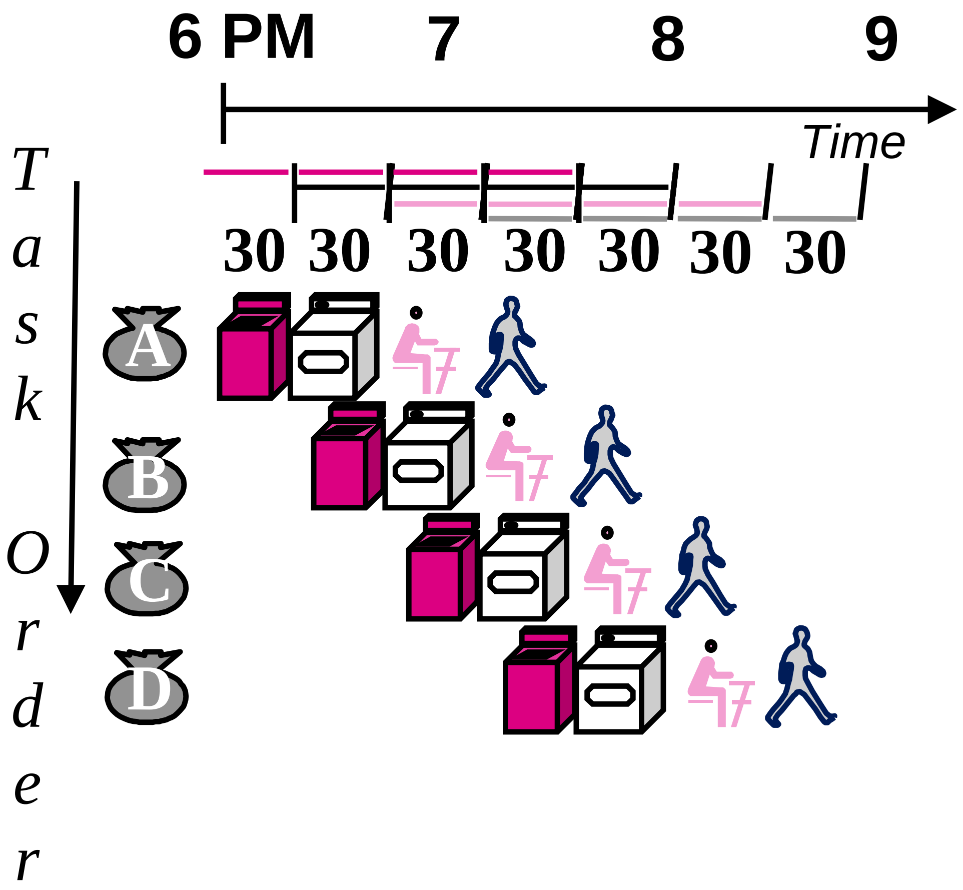
- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



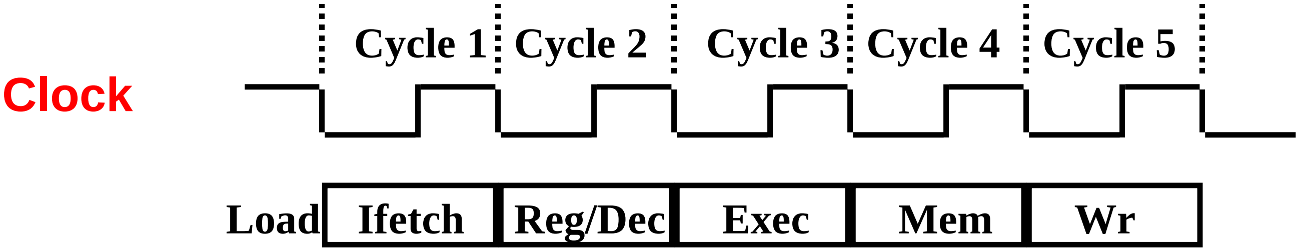
- Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Lessons



- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
- Stall for Dependences

The Five Stages of Load



Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5

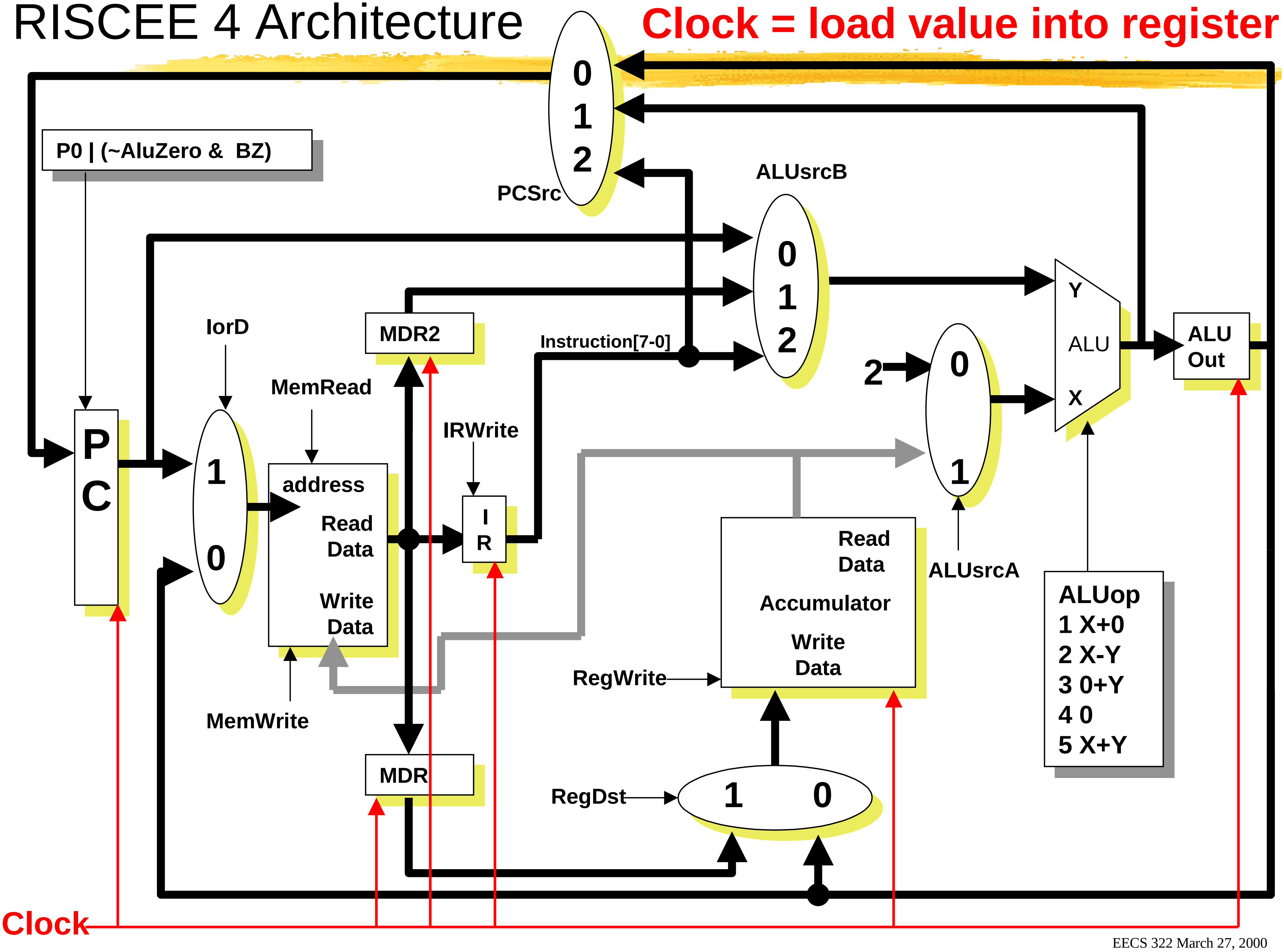
Clock

Load Ifetch Reg/Dec Exec Mem Wr

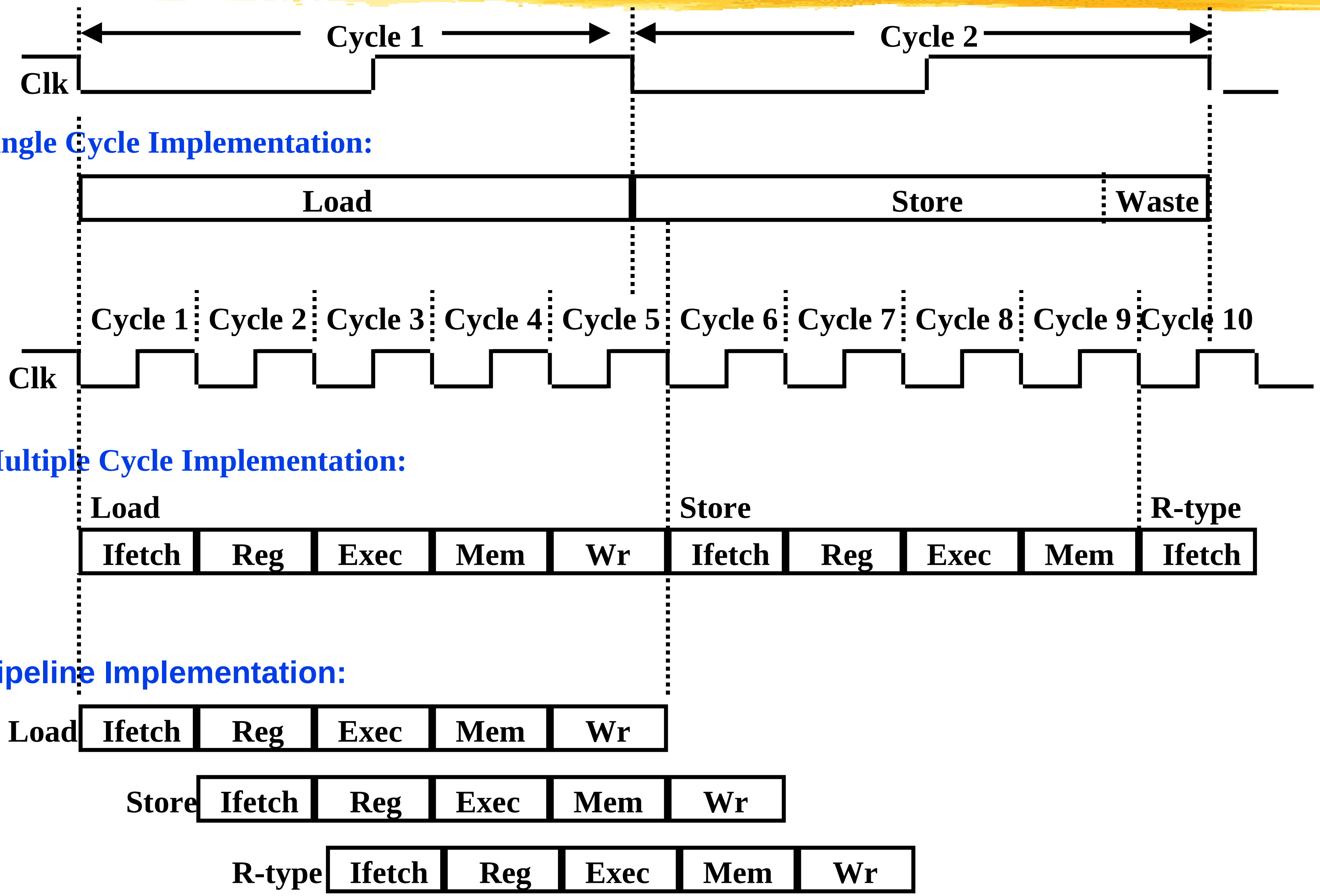
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

RISCEE 4 Architecture

Clock = load value into register



Single Cycle, Multiple Cycle, vs. Pipeline



Cycle 1 Cycle 2

Clk

Single Cycle Implementation:

Load Store Waste

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8 Cycle 9 Cycle 10

Clk

Multiple Cycle Implementation:

Load: Ifetch Reg Exec Mem Wr

Store: Ifetch Reg Exec Mem

R-type: Ifetch

Pipeline Implementation:

Load Ifetch Reg Exec Mem Wr

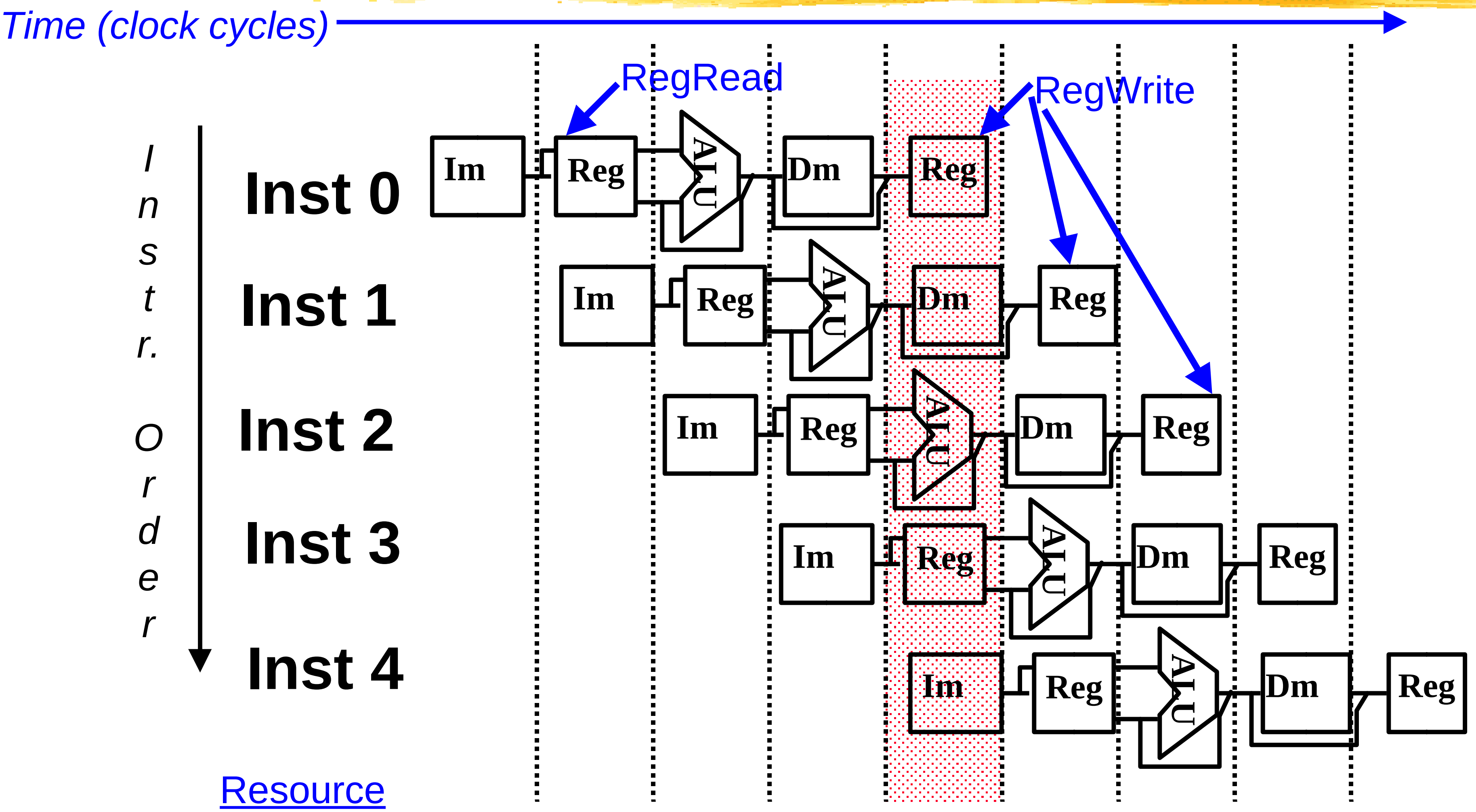
Store Ifetch Reg Exec Mem Wr

R-type Ifetch Reg Exec Mem Wr

Why Pipeline?

- Suppose we execute 100 instructions
- Single Cycle Machine
  - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
- Multicycle Machine
  - 10 ns/cycle x 4.6 CPI (due to inst mix) x 100 inst = 4600 ns
- Ideal pipelined machine
  - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns

Why Pipeline? Because the resources are there!



Resource									
MemInst	busy	busy	busy	busy	busy	idle	idle	idle	idle
MemData	idle	idle	idle	busy	busy	busy	busy	busy	idle
RegRead	idle	busy	busy	busy	busy	busy	idle	idle	idle
RegWrite	idle	idle	idle	idle	busy	busy	busy	busy	busy
ALU	idle	idle	busy	busy	busy	busy	busy	idle	idle

Can pipelining get us into trouble?

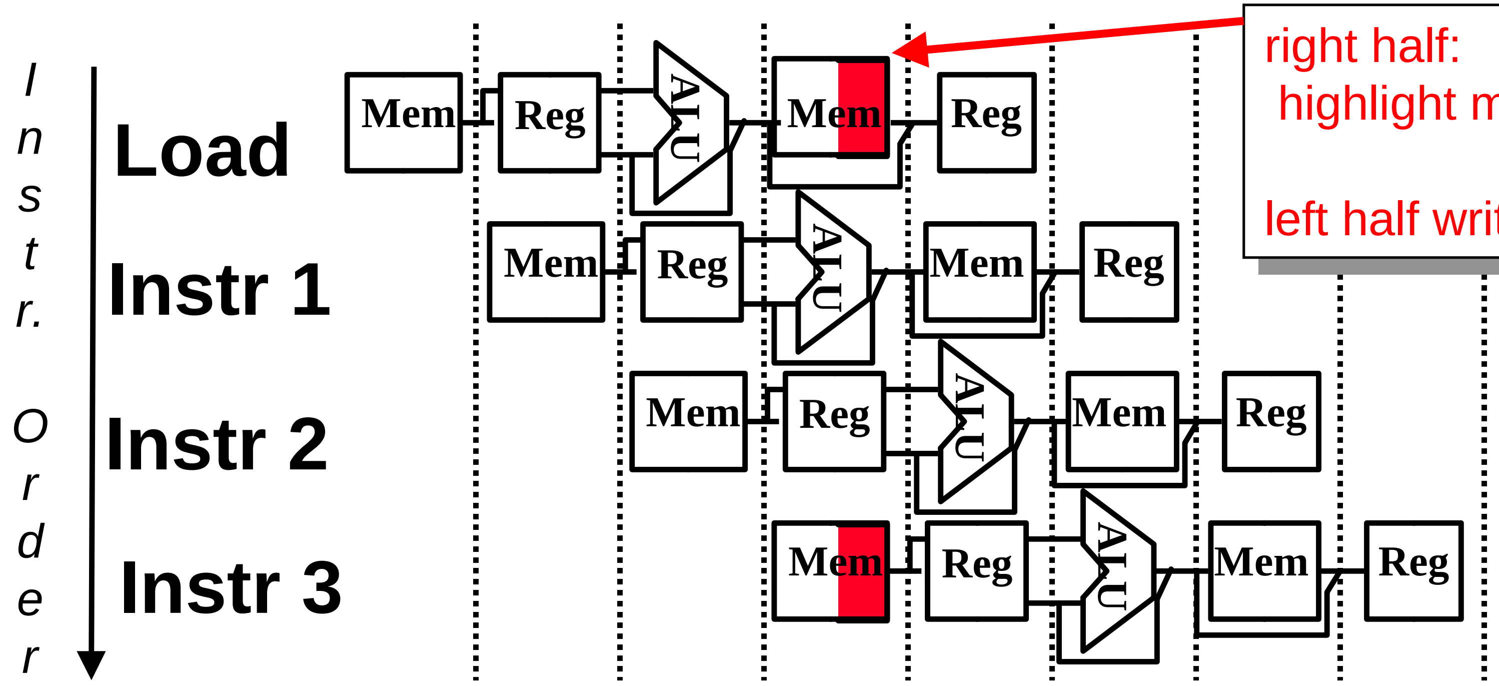
- **Yes: Pipeline Hazards**
  - structural hazards: attempt to use the same resource two different ways at the same time
    - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
  - data hazards: attempt to use item before it is ready
    - E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
    - instruction depends on result of prior instruction still in the pipeline
  - control hazards: attempt to make a decision before condition is evaulated
    - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
    - branch instructions
- **Can always resolve hazards by waiting**
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards

Single Memory (Inst & Data) is a Structural Hazard

structural hazards:

attempt to use the same resource two different ways at the same time

Previous example: Separate InstMem and DataMem



right half:
highlight means read.

left half write.

Detection is easy in this case!

Resource								
Mem(Inst & Data)	busy	busy	busy	busy	busy	idle	idle	idle
RegRead	idle	busy	busy	busy	busy	busy	idle	idle
RegWrite	idle	idle	idle	idle	busy	busy	busy	busy
ALU	idle	idle	busy	busy	busy	busy	busy	idle

Single Memory (Inst & Data) is a Structural Hazard

structural hazards:

attempt to use the same resource two different ways at the same time

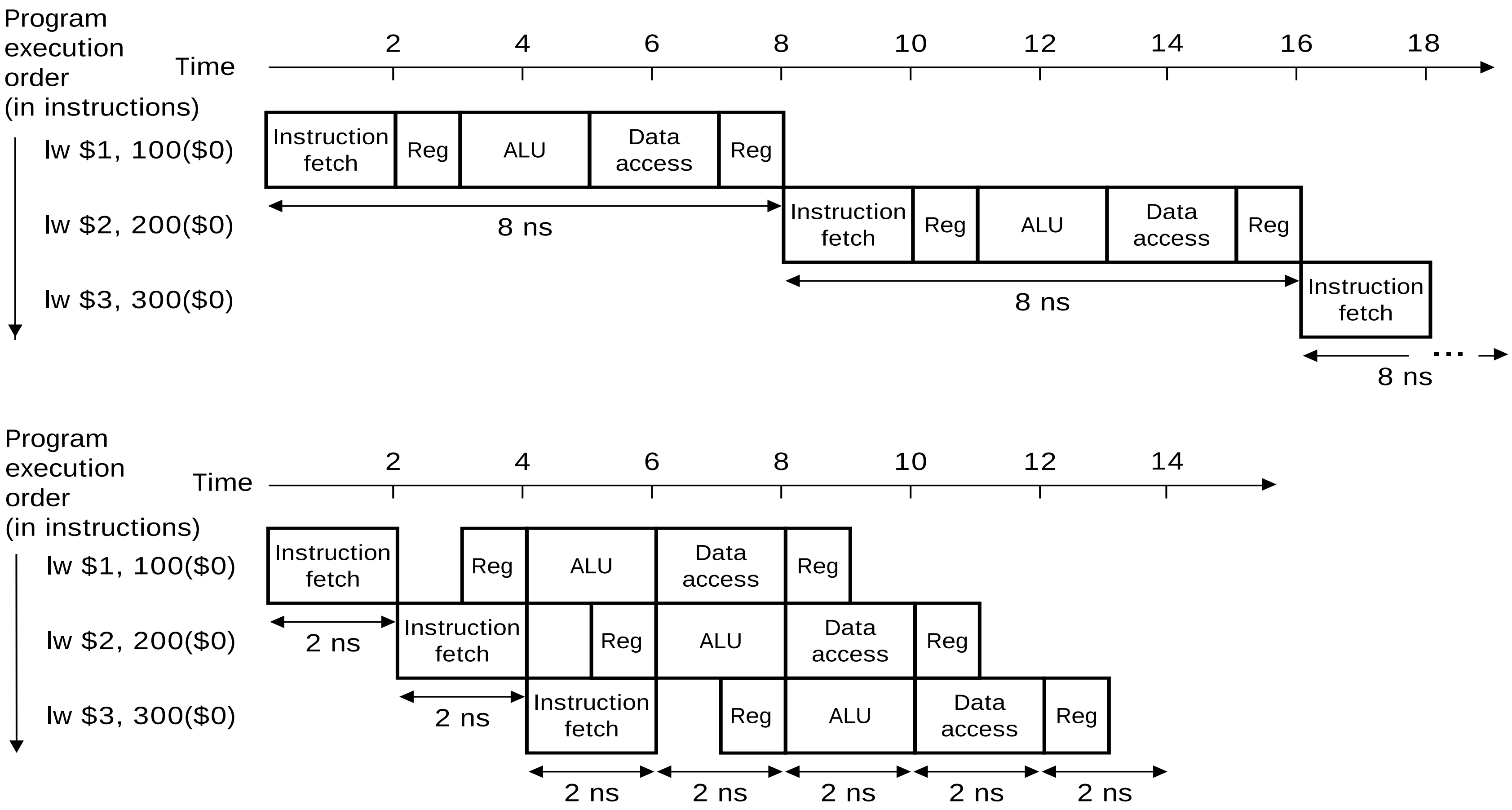
By change the architecture from a Harvard (separate instruction and data memory) to a von Neuman memory, we actually created a structural hazard!

Structural hazards can be avoid by changing

- hardware: design of the architecture (splitting resources)
- software: re-order the instruction sequence
- software: delay

Pipelining

- Improve perfomance by increasing instruction throughput



Ideal speedup is number of stages in the pipeline. Do we achieve this?

Stall on Branch

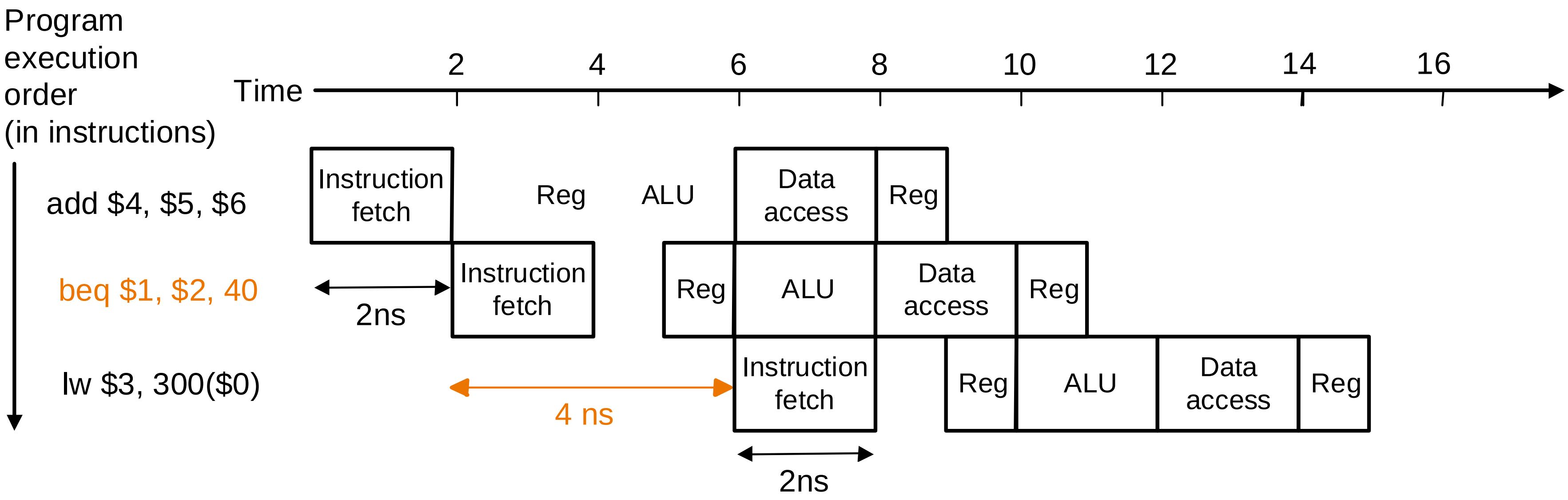


Figure 6.4

Predicting branches

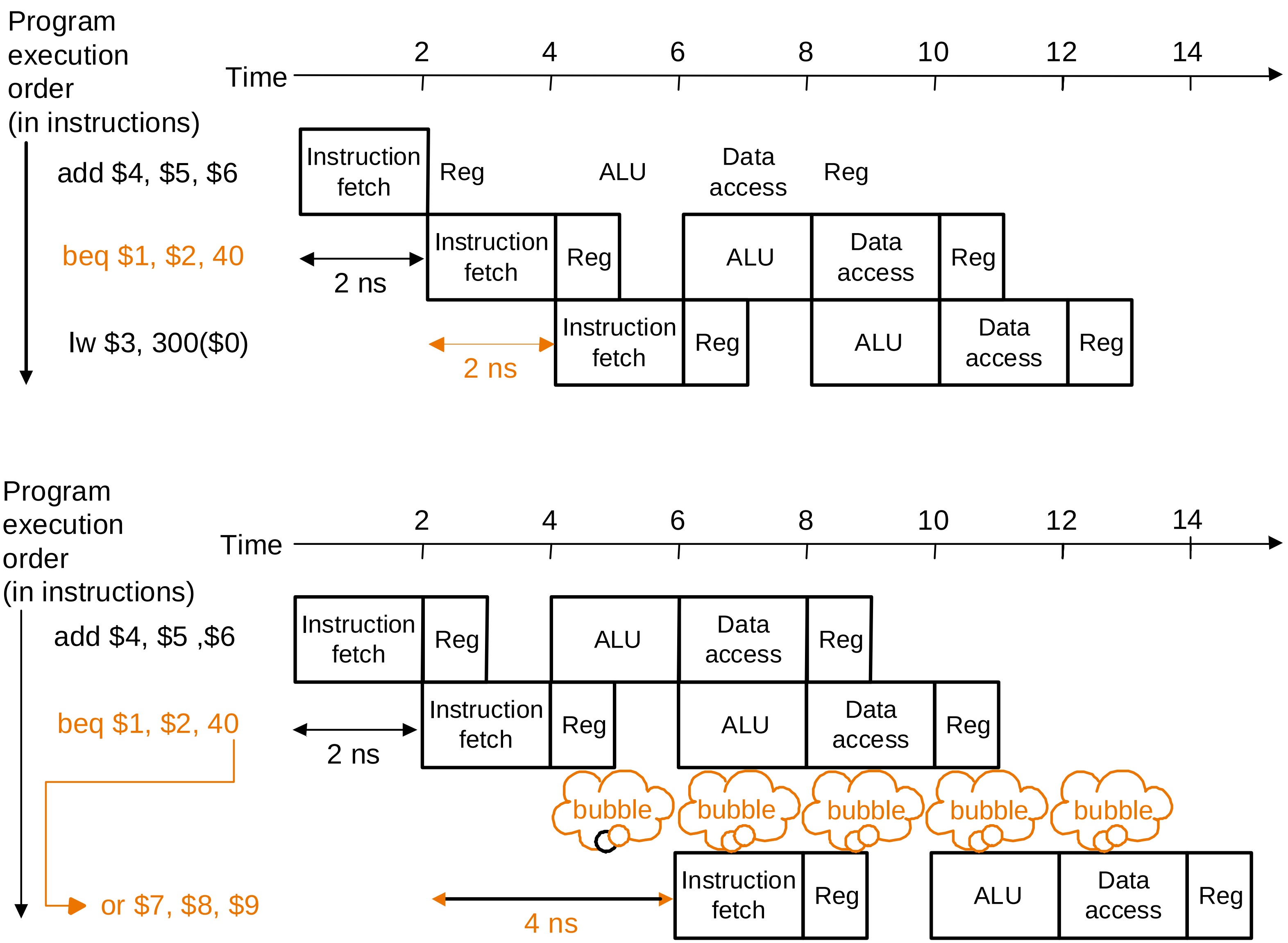


Figure 6.5

Delayed branch

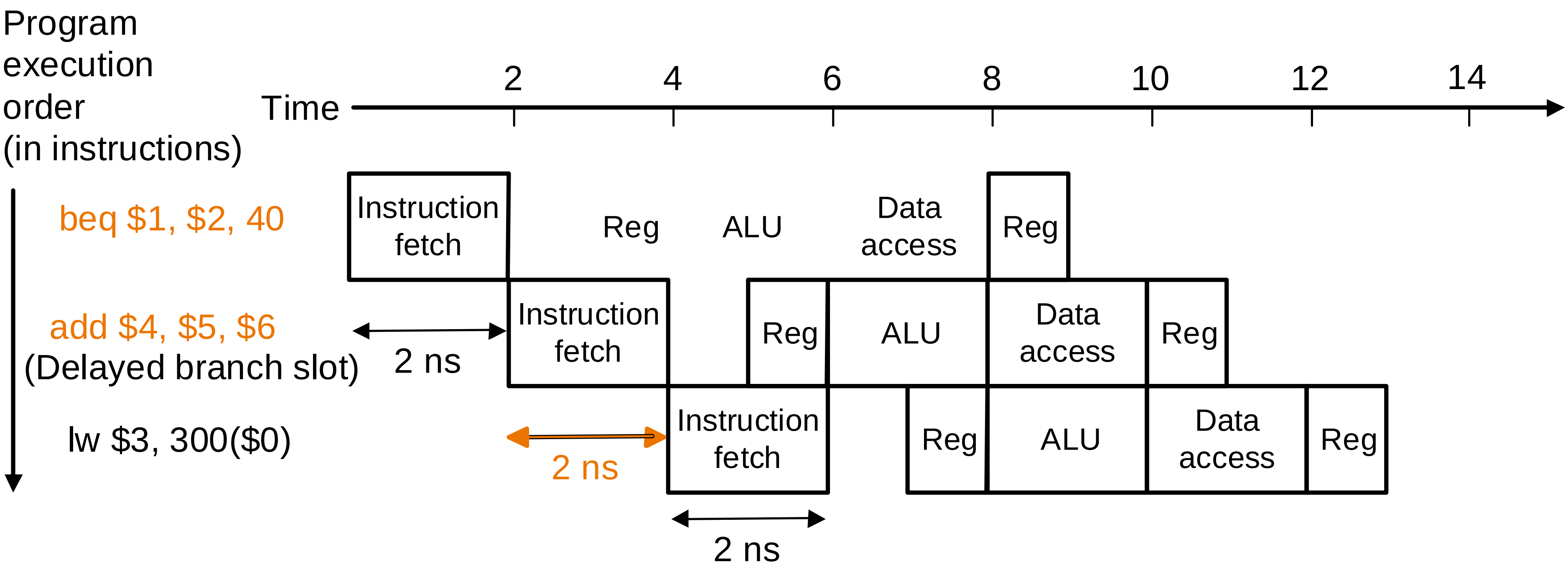
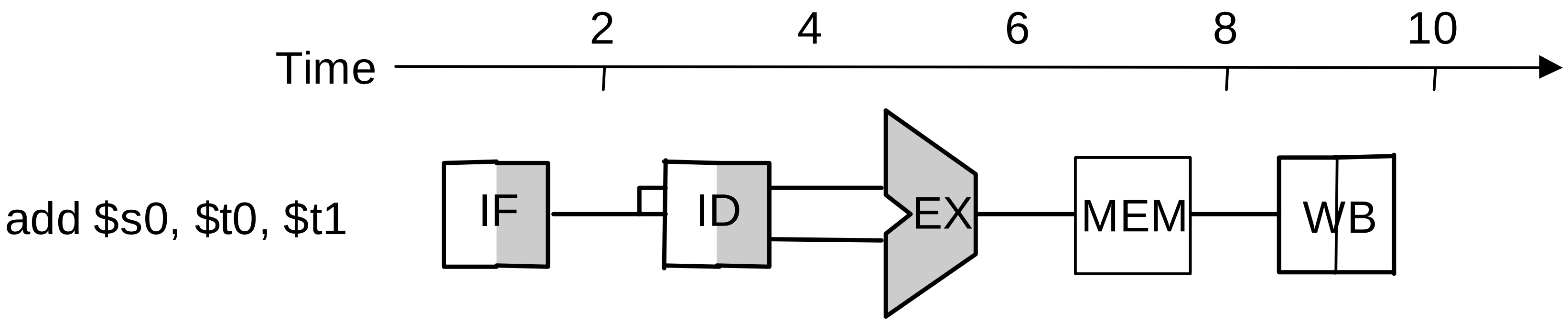


Figure 6.6

Instruction pipeline

Figure 6.7



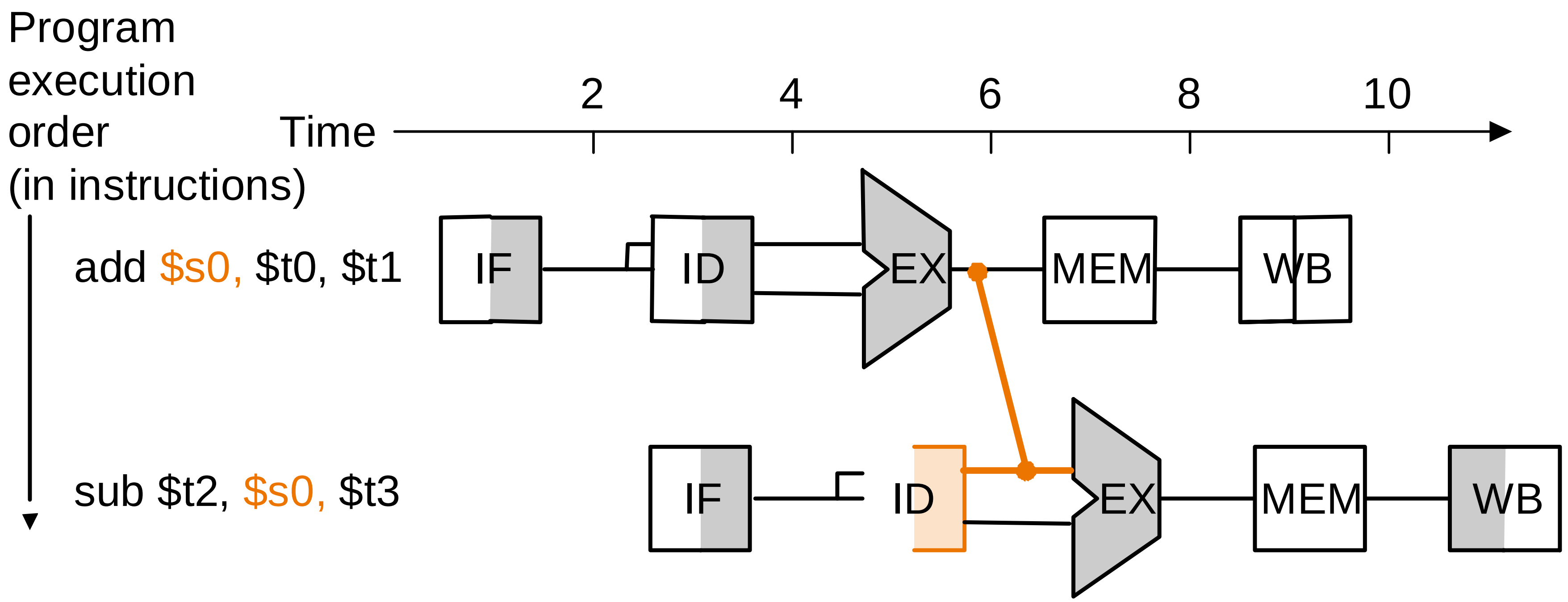
Pipeline stages

• IF	instruction fetch (read)
• ID	instruction decode and register read (read)
• EX	execute alu operation
• MEM	data memory (read or write)
• WB	Write back to register

Resources

• Mem	instr. & data memory
• RegRead1	register read port #1
• RegRead2	register read port #2
• RegWrite	register write
• ALU	alu operation

Forwarding



Program execution order (in instructions)

Time

2 4 6 8 10

add $s0, $t0, $t1

IF ID EX MEM WB

sub $t2, $s0, $t3

IF ID EX MEM WB

Figure 6.8

Load Forwarding

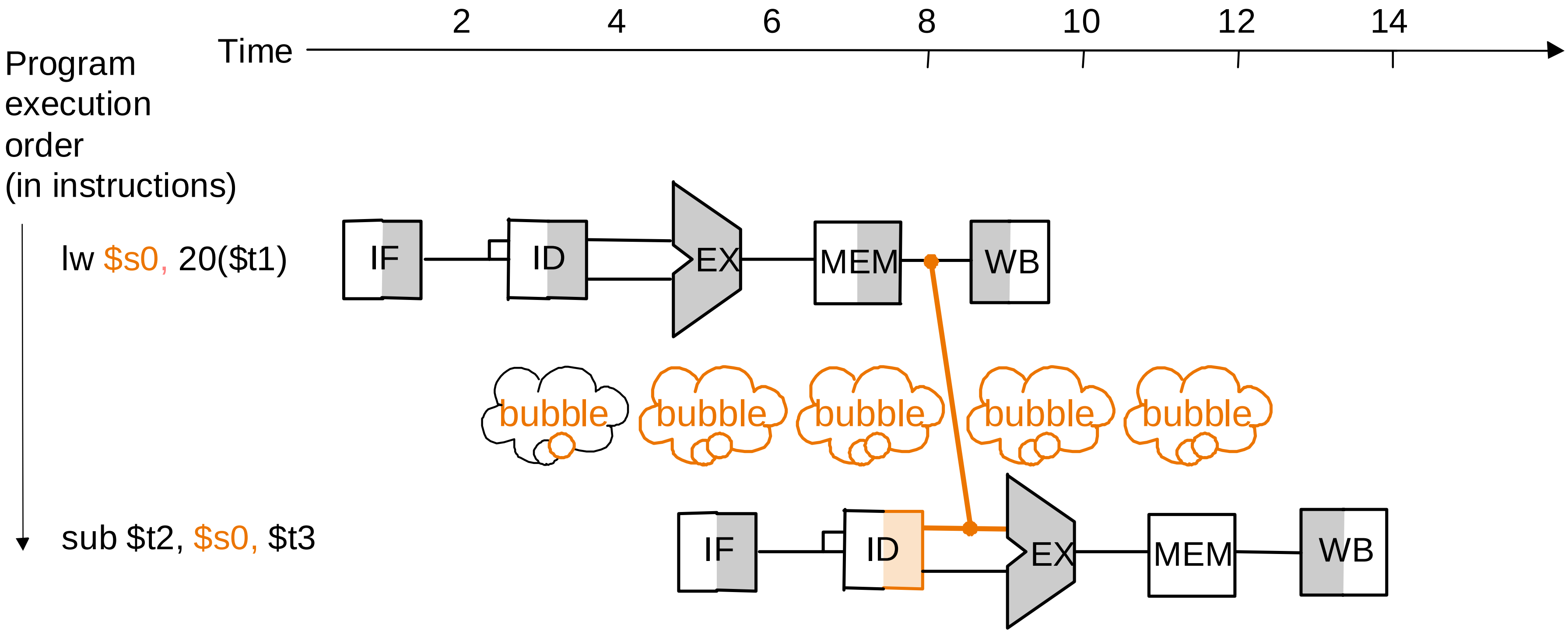


Figure 6.9

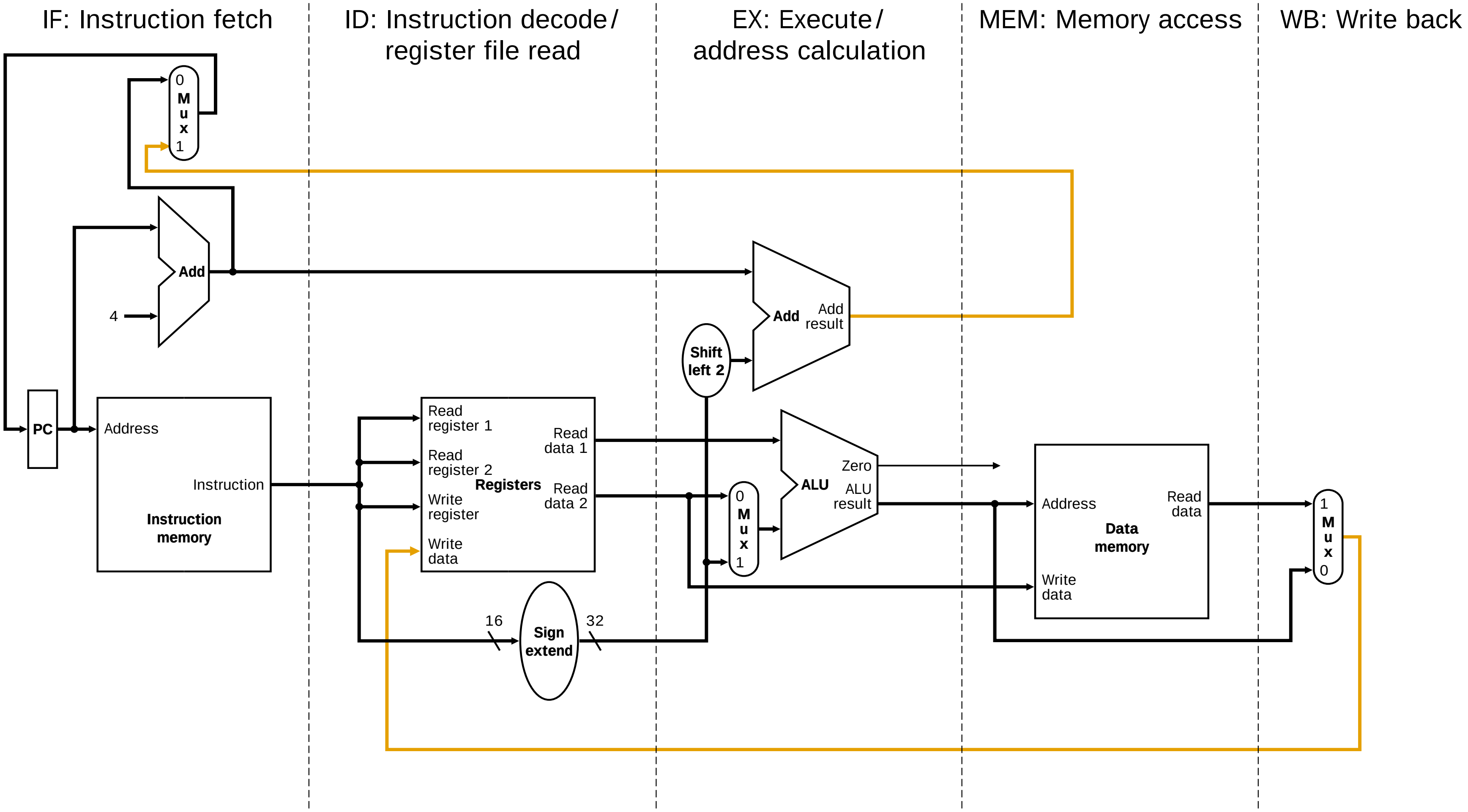
Reordering

```
lw    $t0, 0($t1)        $t0=Memory[0+$t1]
lw    $t2, 4($t1)        $t2=Memory[4+$t1]
sw    $t2, 0($t1)        Memory[0+$t1]=$t2
sw    $t0, 4($t1)        Memory[4+$t1]=$t0
```

```
lw    $t2, 4($t1)
lw    $t0, 0($y1)
sw    $t2, 0($t1)
sw    $t0, 4($t1)
```

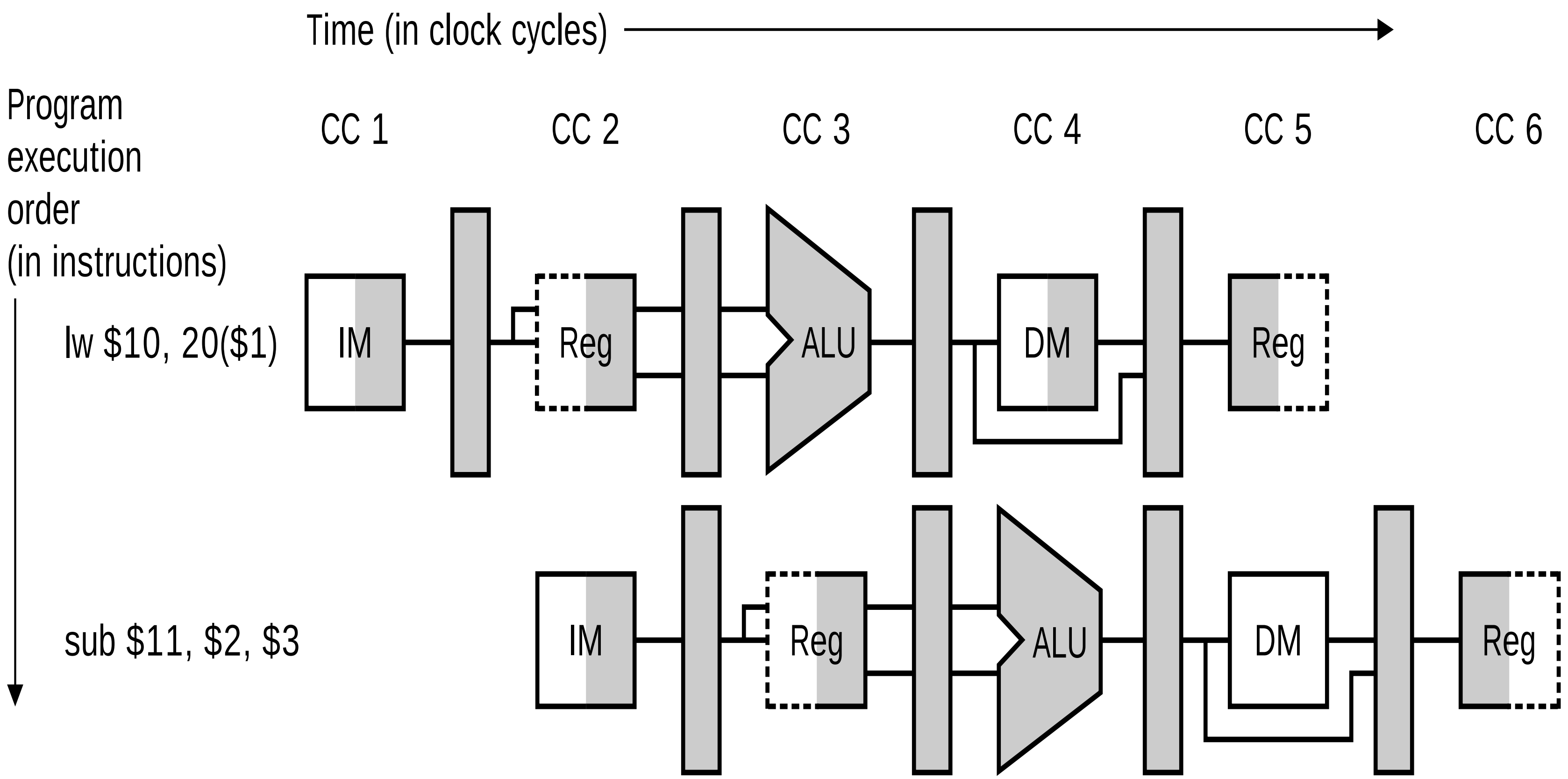
Figure 6.9

Basic Idea: split the datapath



- *What do we need to add to actually split the datapath into stages?*

Graphically Representing Pipelines



° **Can help with answering questions like:**

- **how many cycles does it take to execute this code?**
- **what is the ALU doing during cycle 4?**
- **use this representation to help understand datapaths**

Pipeline datapath with registers

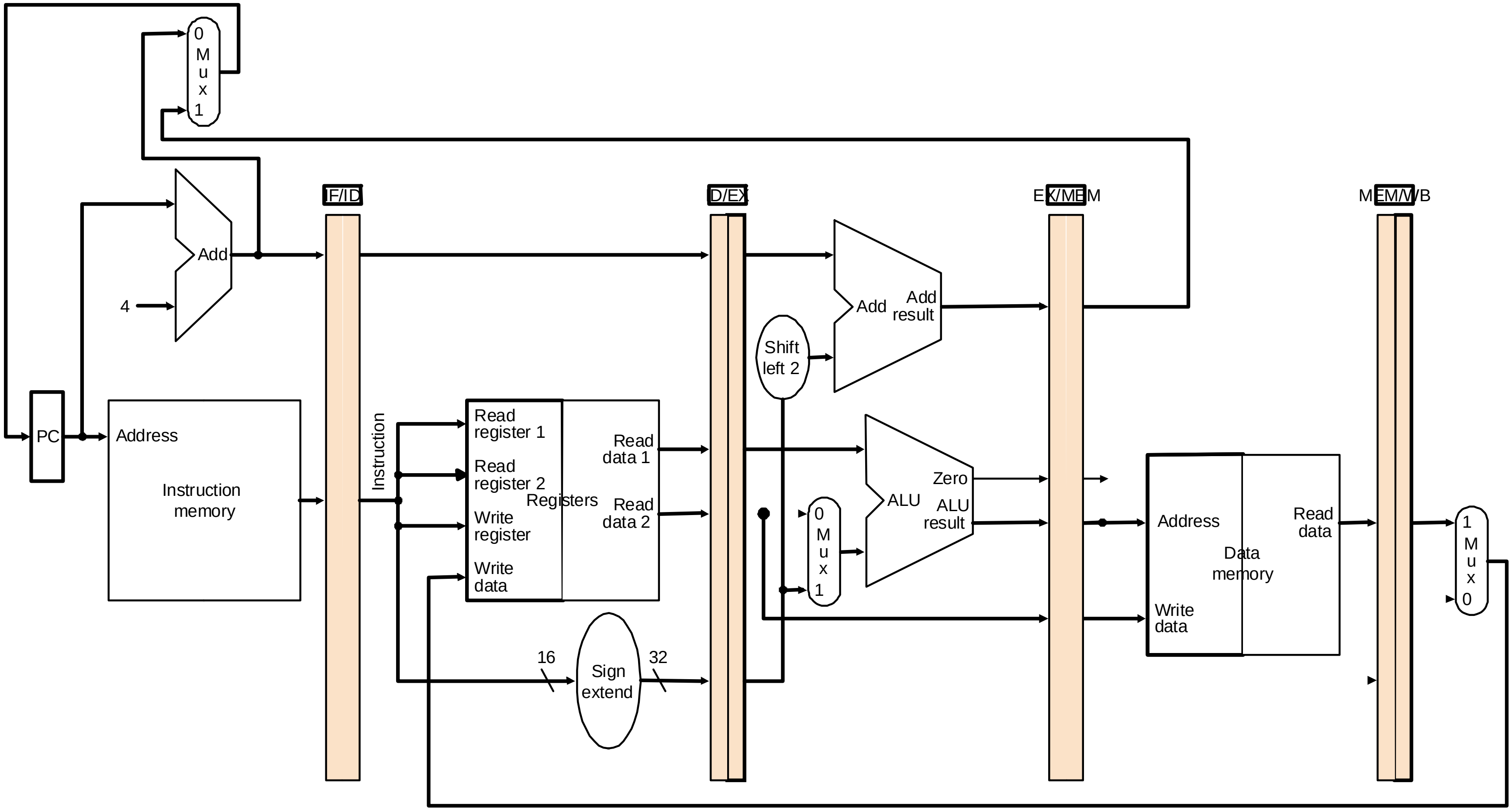
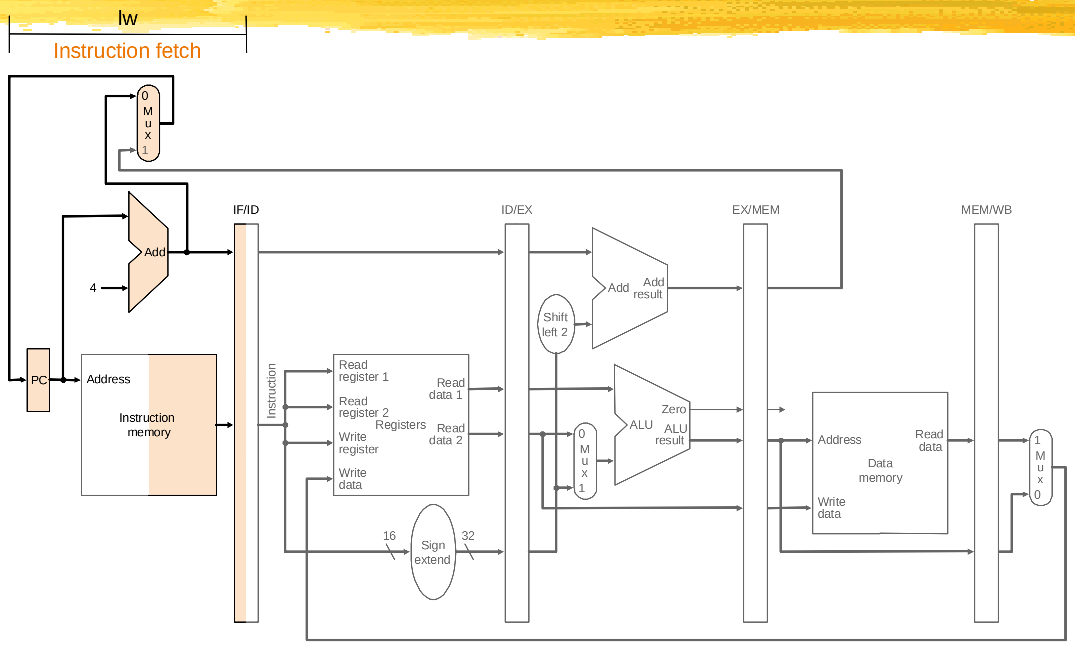


Figure 6.12

Load instruction fetch and decode



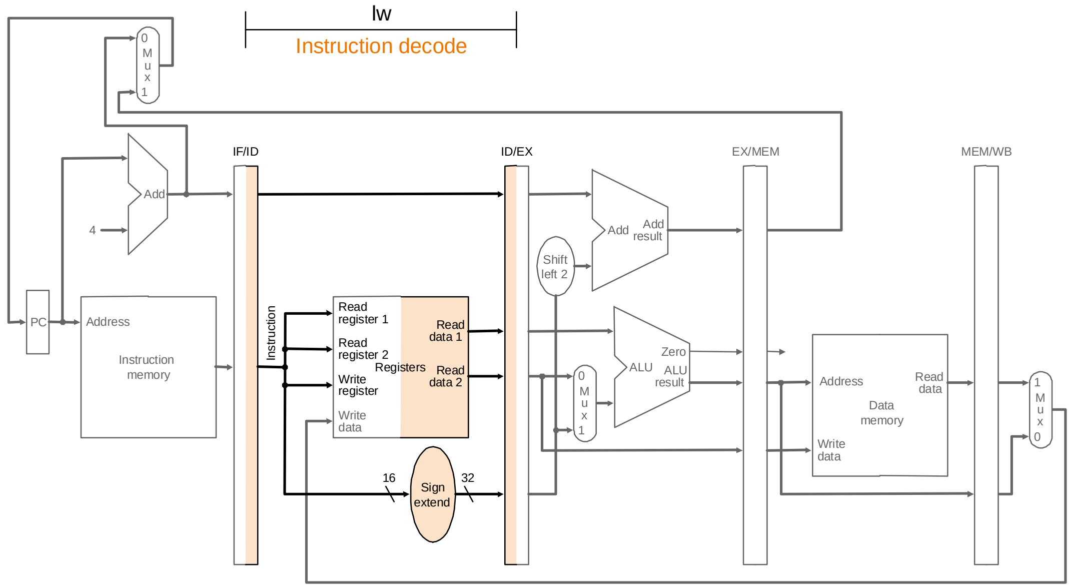


Figure 6.13

Load instruction execution

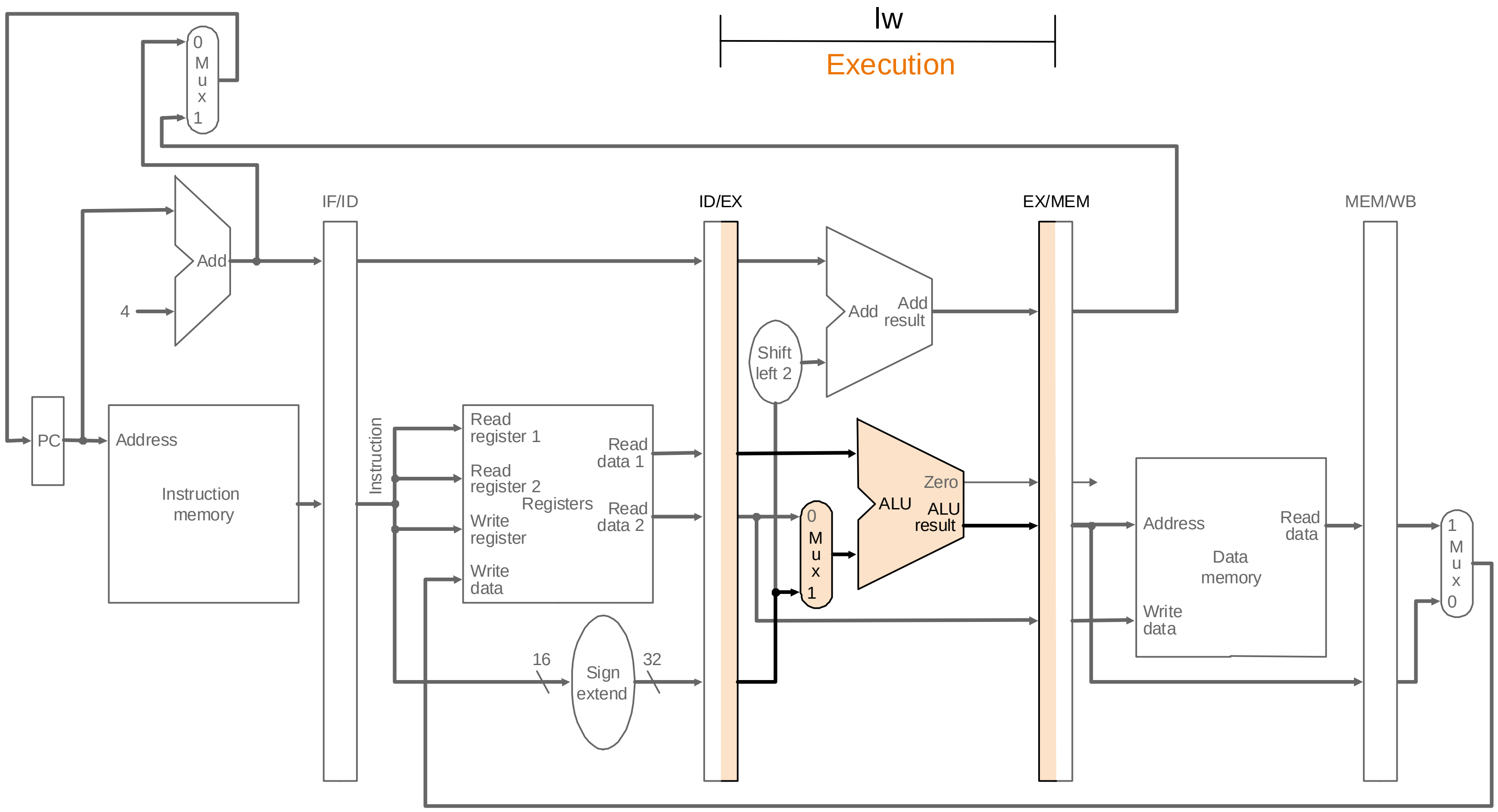
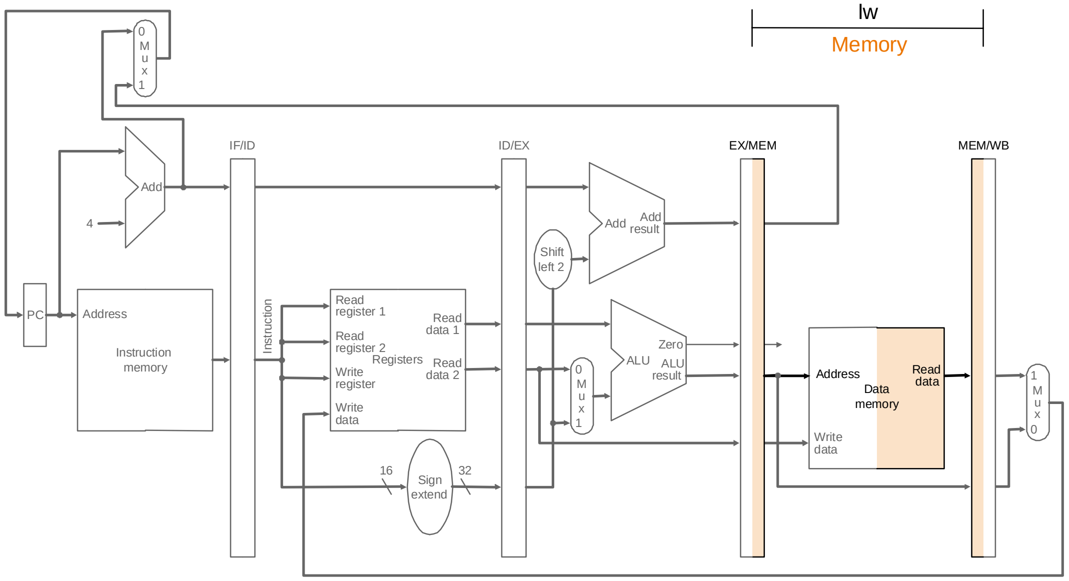


Figure 6.14

Load instruction memory and write back



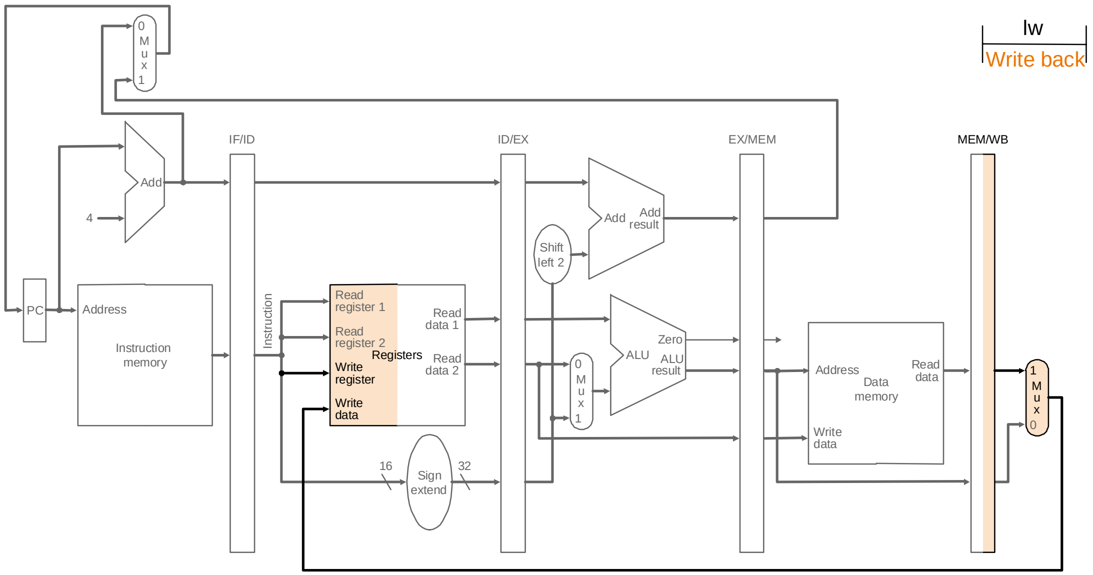


Figure 6.15

Store instruction execution

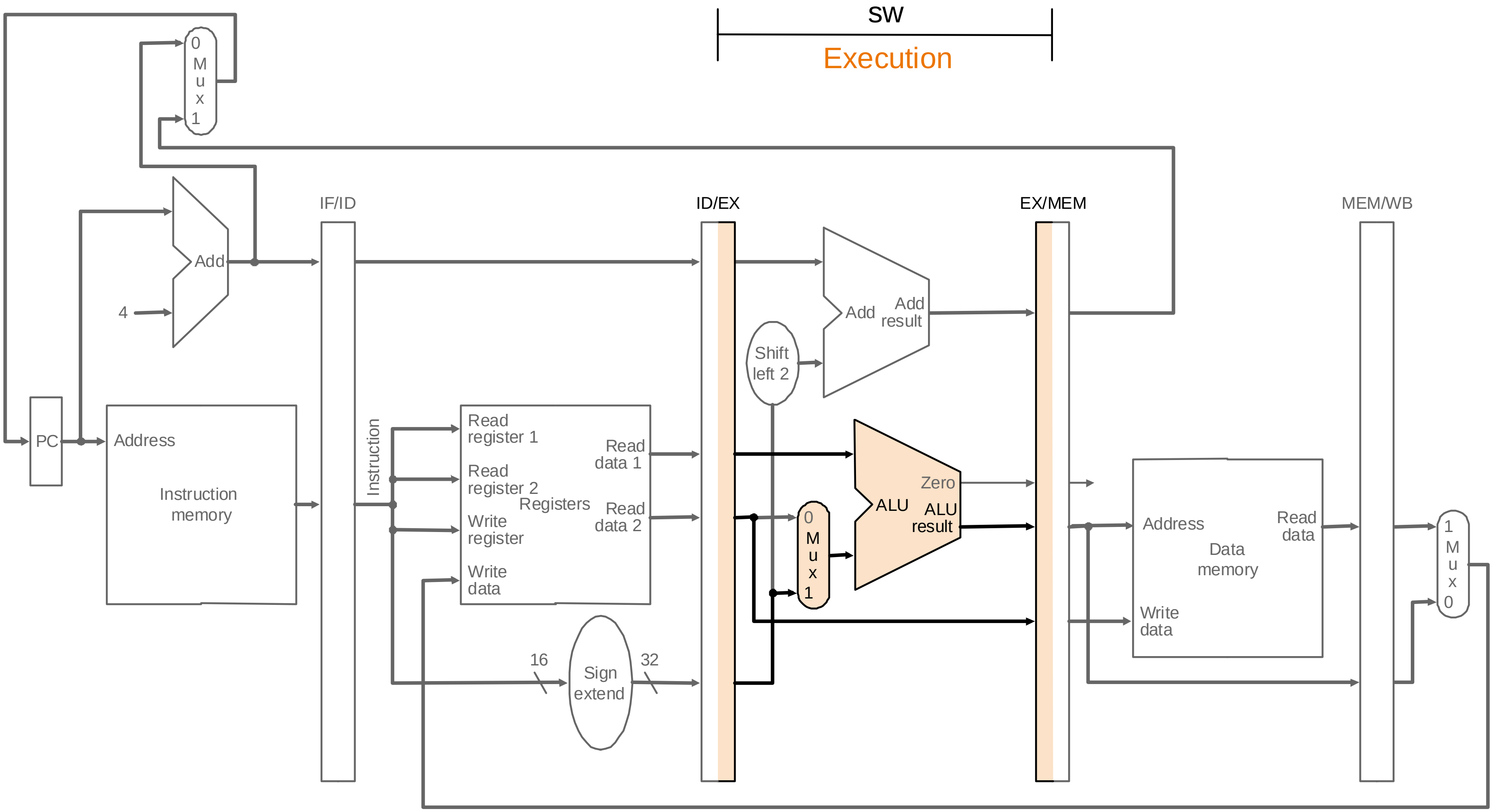


Figure 6.16

Store instruction memory and write back

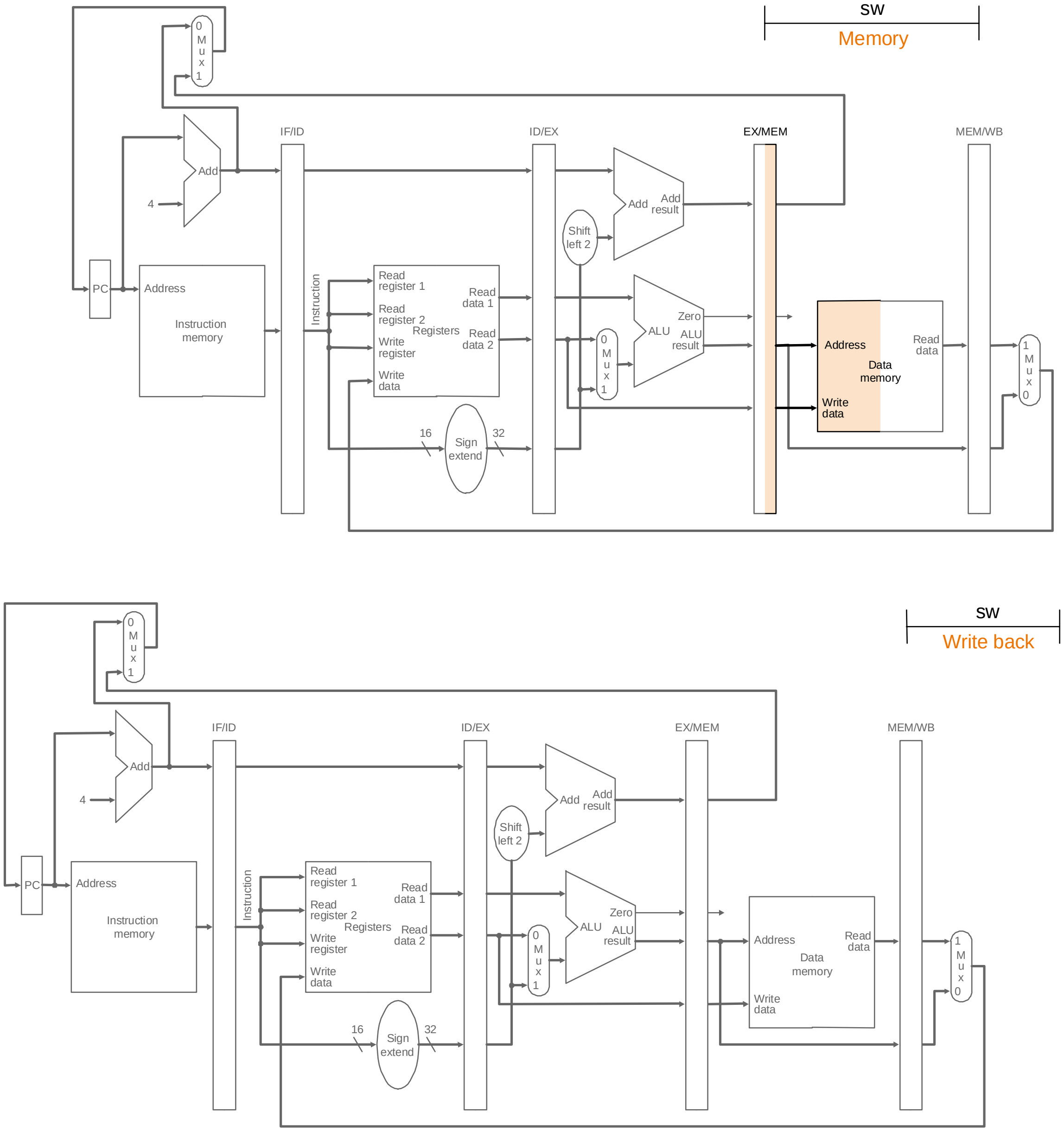


Figure 6.17

Load instruction: corrected datapath

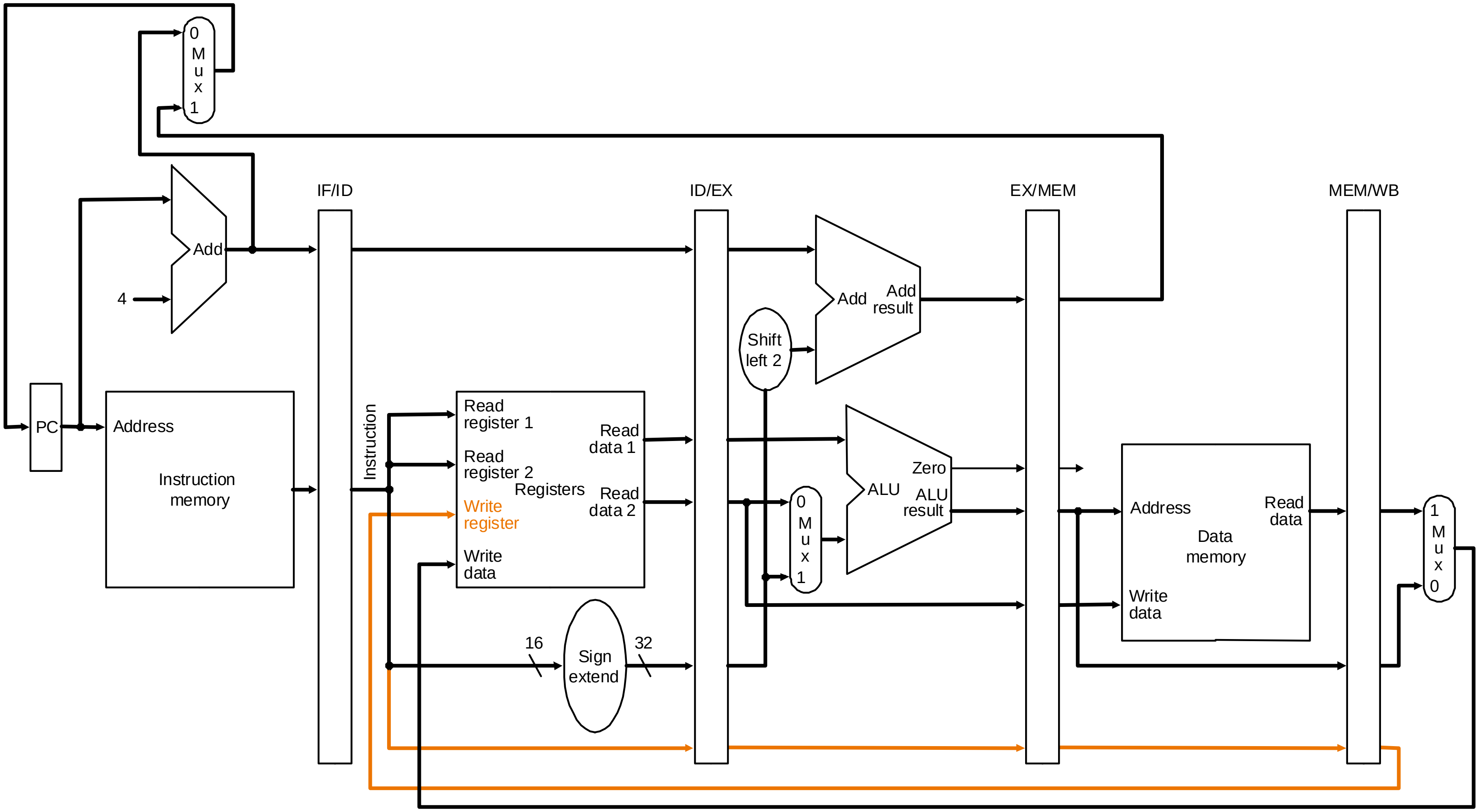


Figure 6.18

Load instruction: overall usage

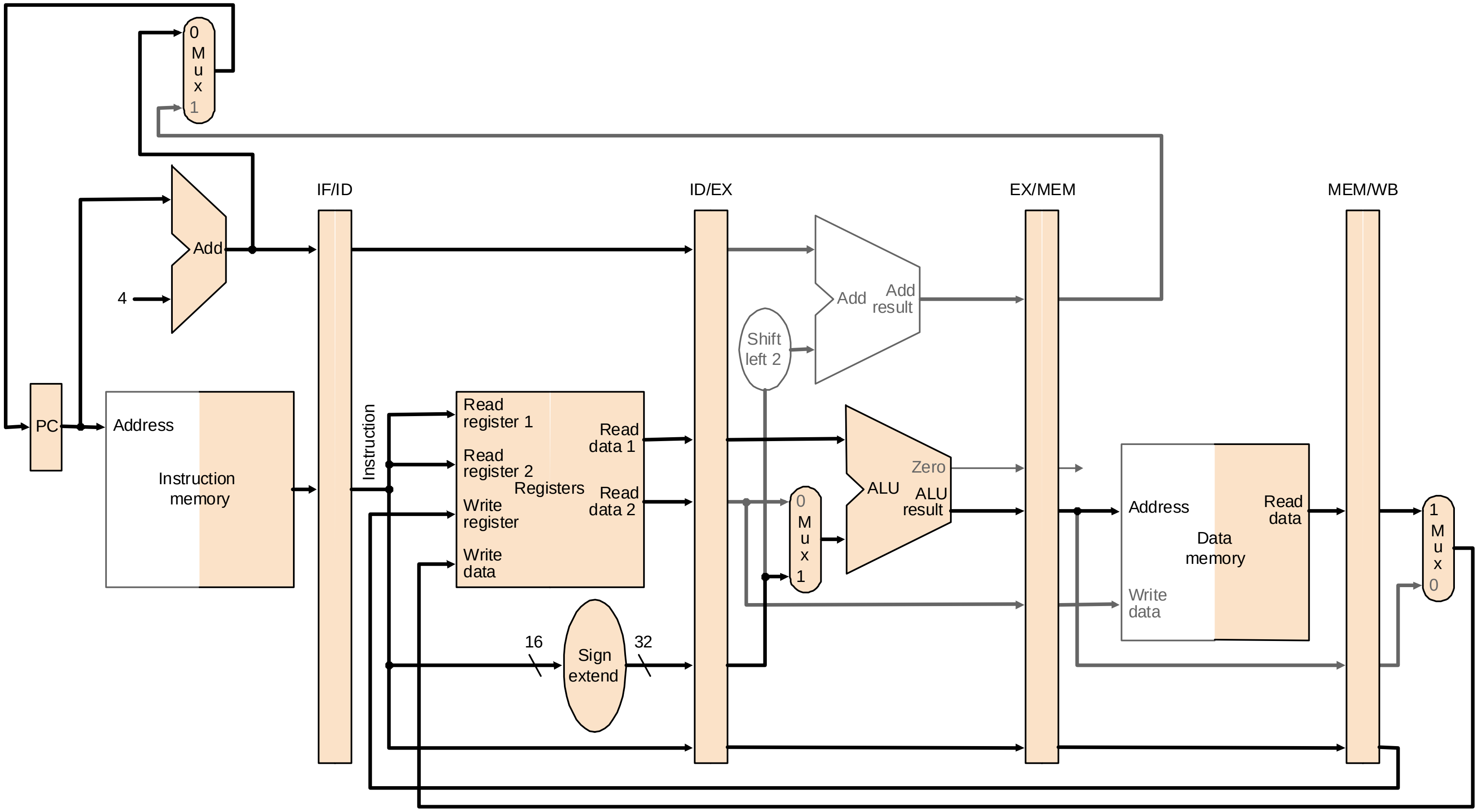
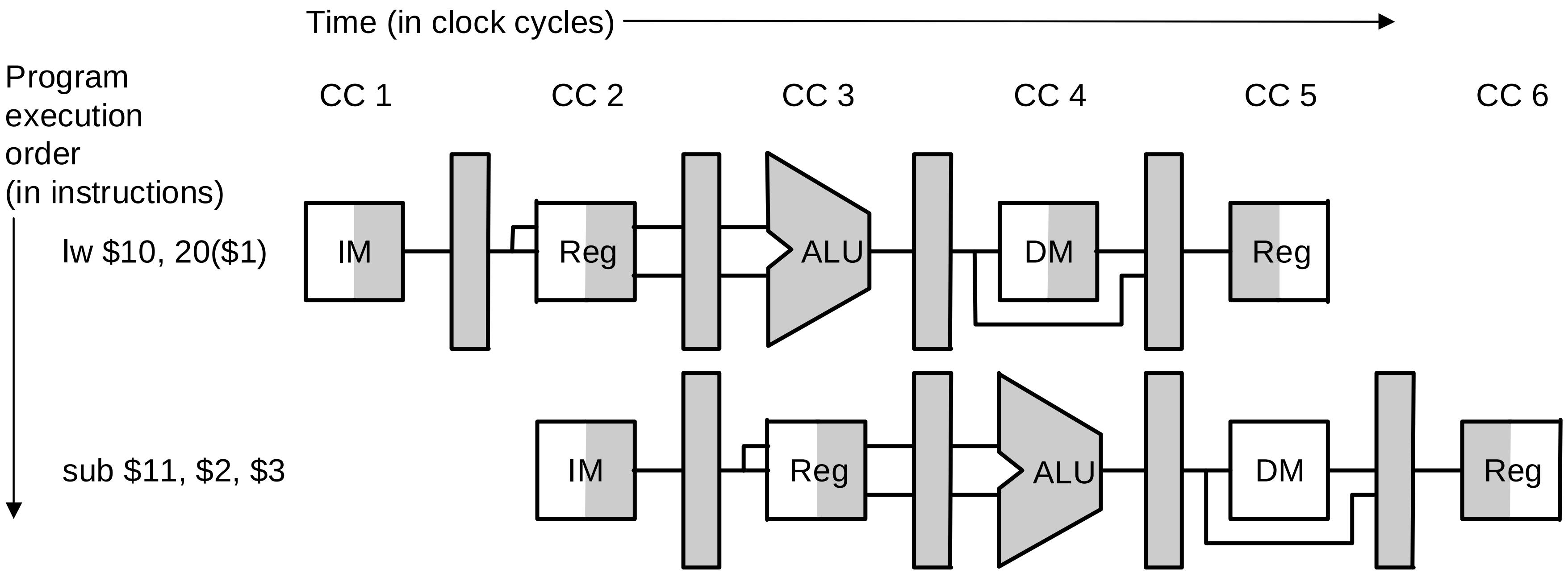


Figure 6.19

Multi-clock-cycle pipeline diagram



Time (in clock cycles)

Program execution order (in instructions)

CC 1 CC 2 CC 3 CC 4 CC 5 CC 6

lw $10, 20($1)

sub $11, $2, $3

Time (in clock cycles)

Program execution order (in instructions)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6
lw $10, $20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back	
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back

Figure 6.20-21

Single-cycle #1-2

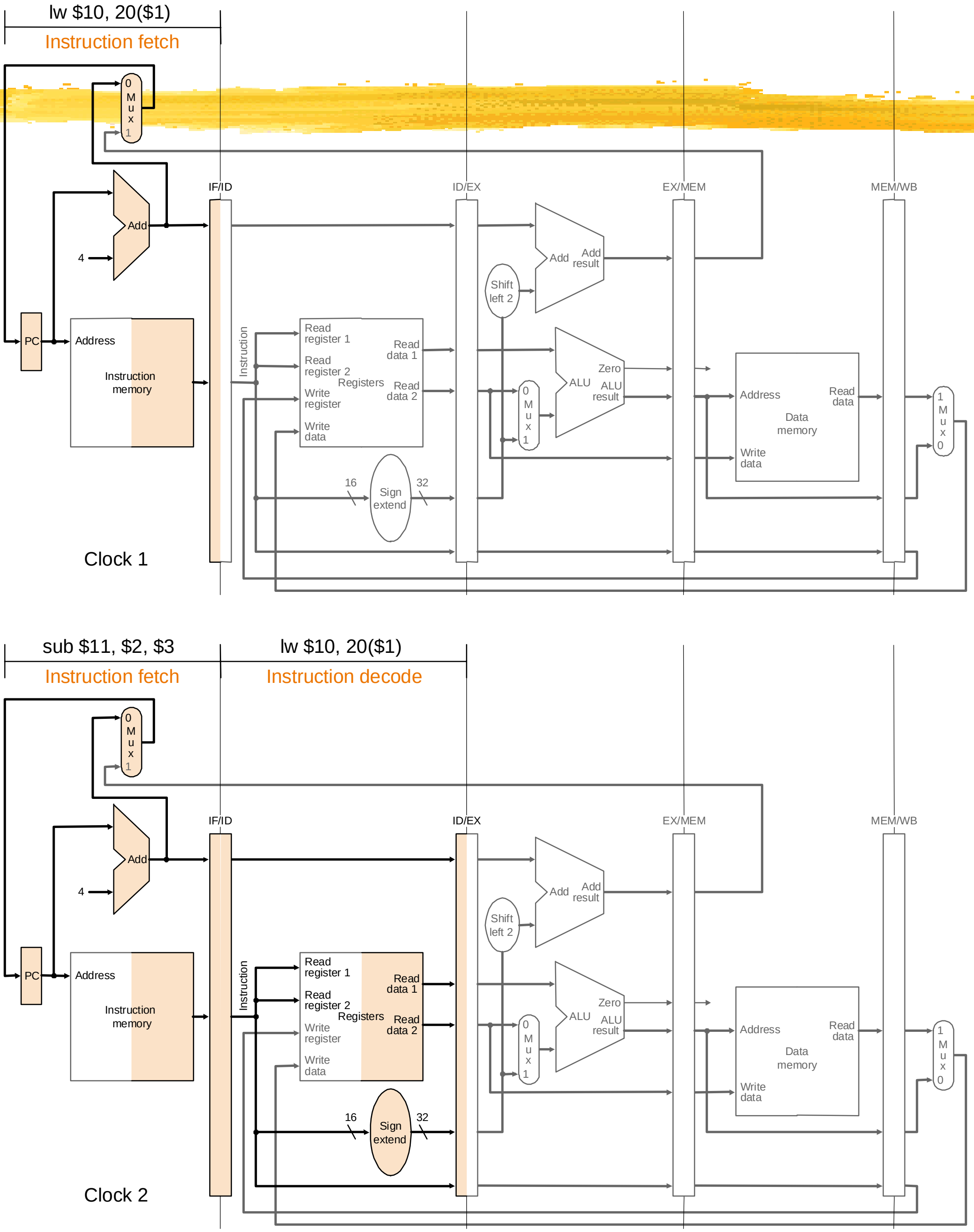


Figure 6.22

Single-cycle #3-4

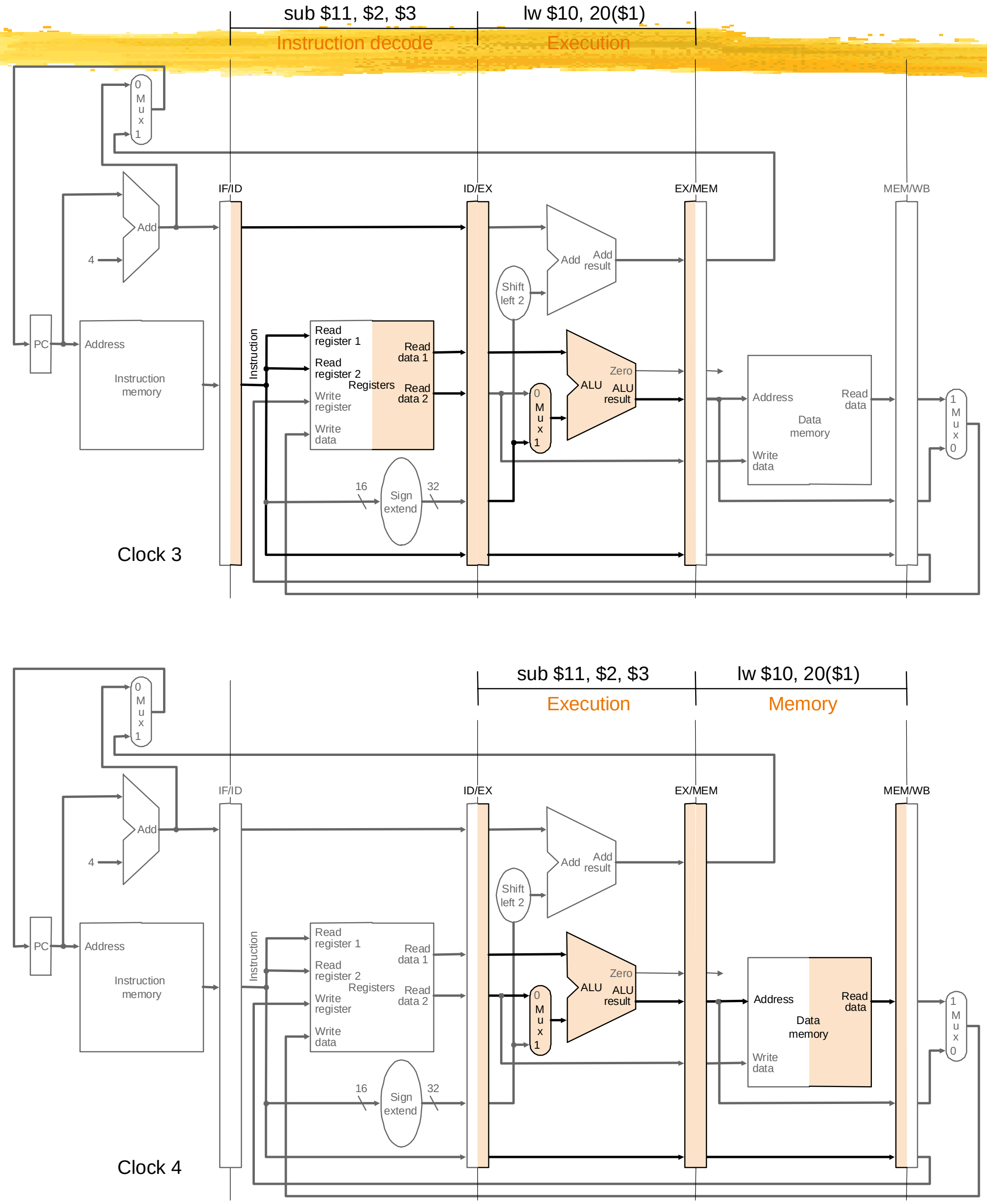


Figure 6.23

Single-cycle #5-6

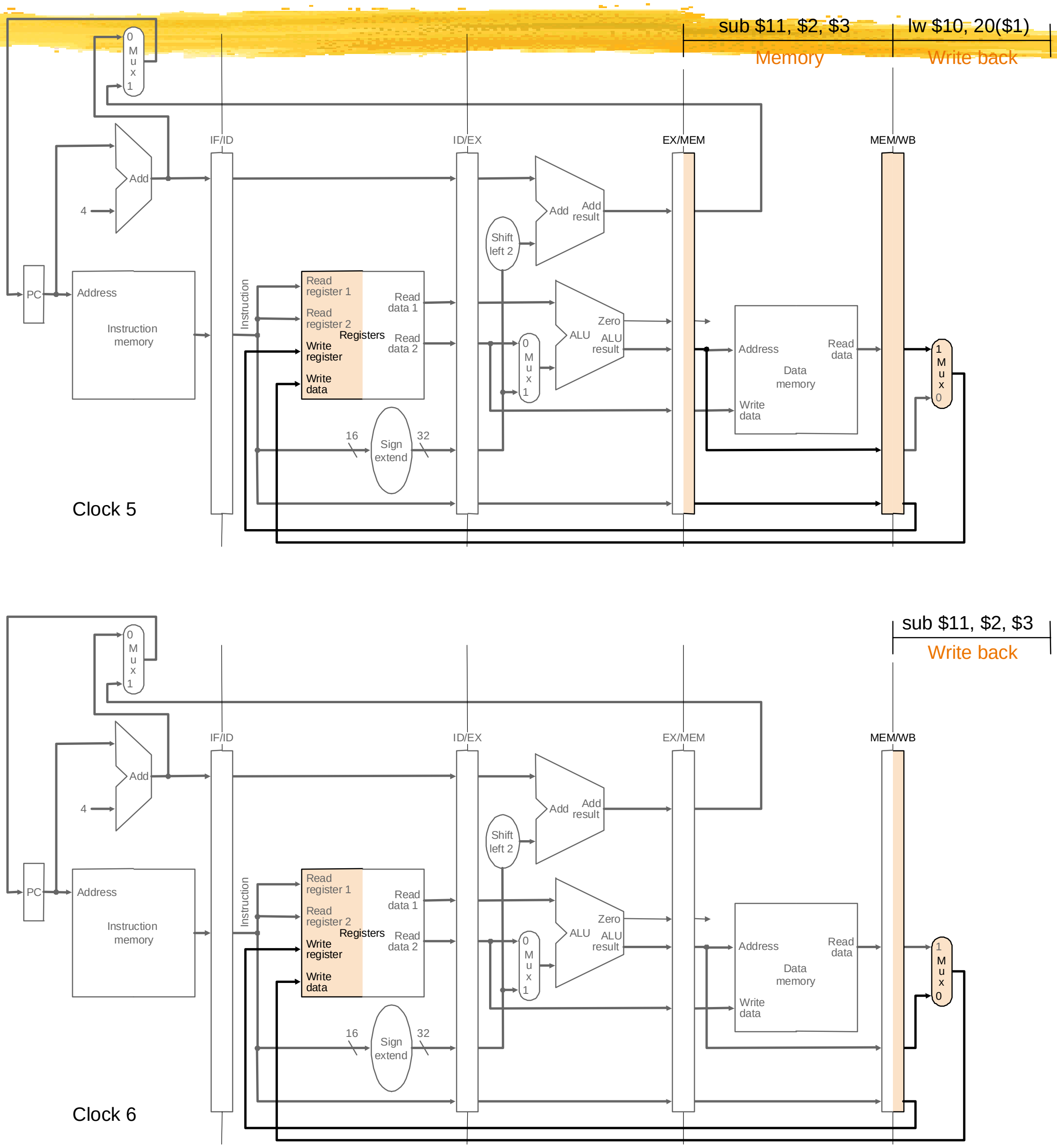
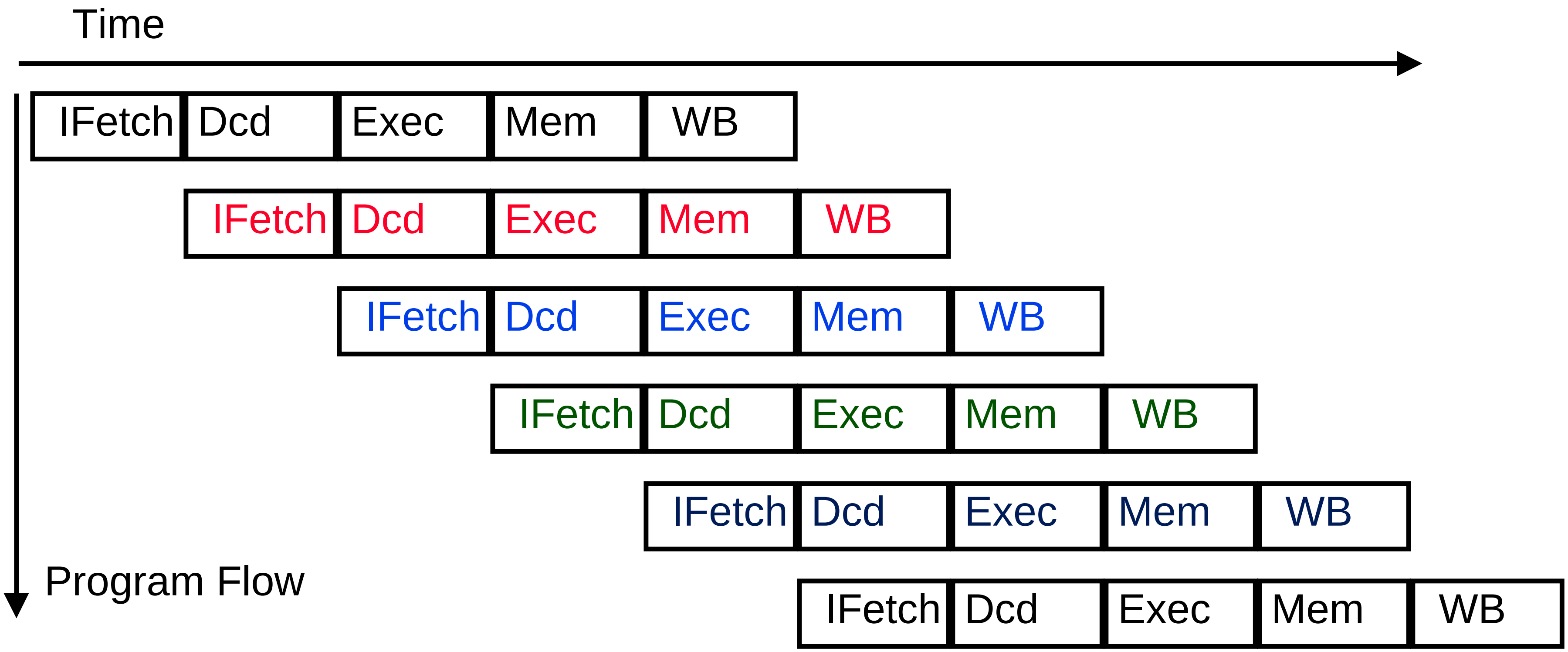


Figure 6.24

Conventional Pipelined Execution Representation



Time

IFetch Dcd Exec Mem WB

IFetch Dcd Exec Mem WB

IFetch Dcd Exec Mem WB

IFetch Dcd Exec Mem WB

IFetch Dcd Exec Mem WB

IFetch Dcd Exec Mem WB

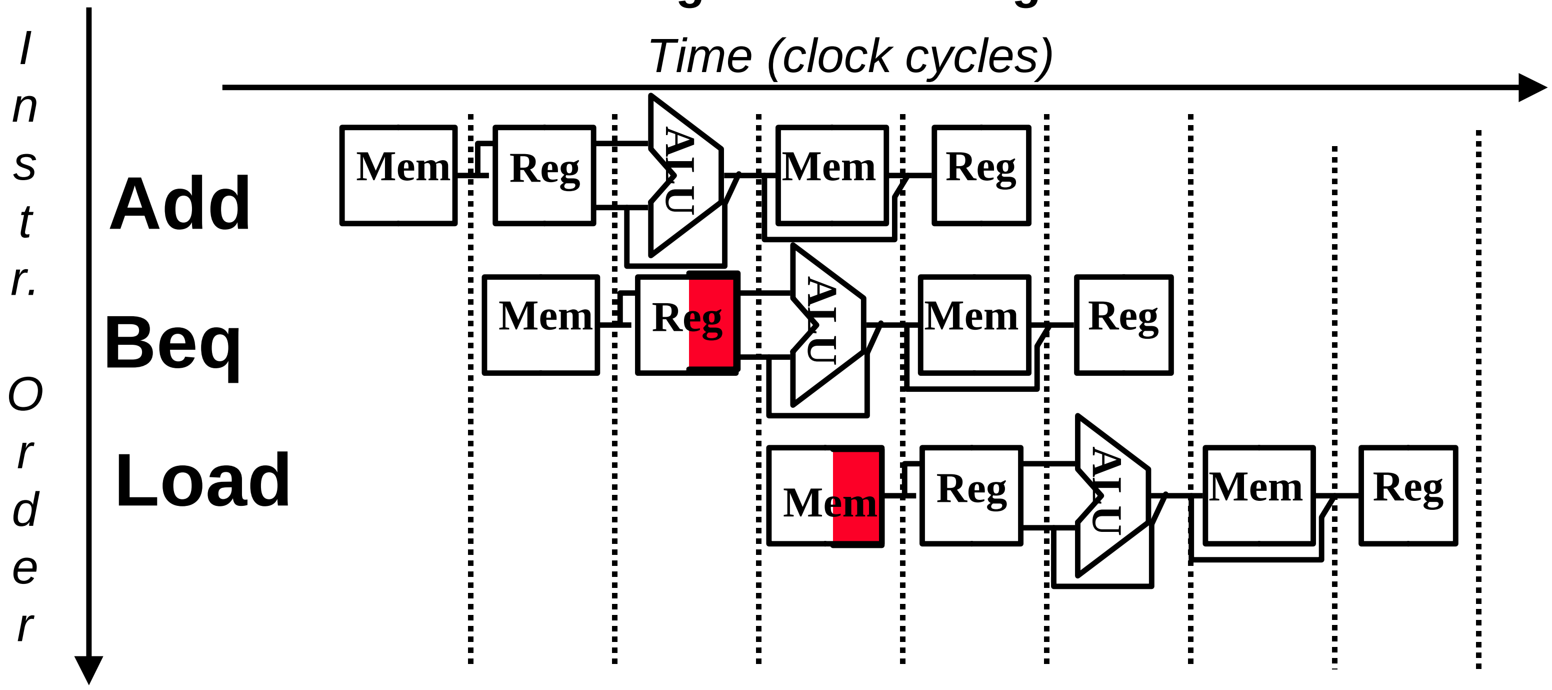
Program Flow

Structural Hazards limit performance

- Example: if 1.3 memory accesses per instruction and only one memory access per cycle then
  - average CPI   1.3
  - otherwise resource is more than 100% utilized

Control Hazard Solutions

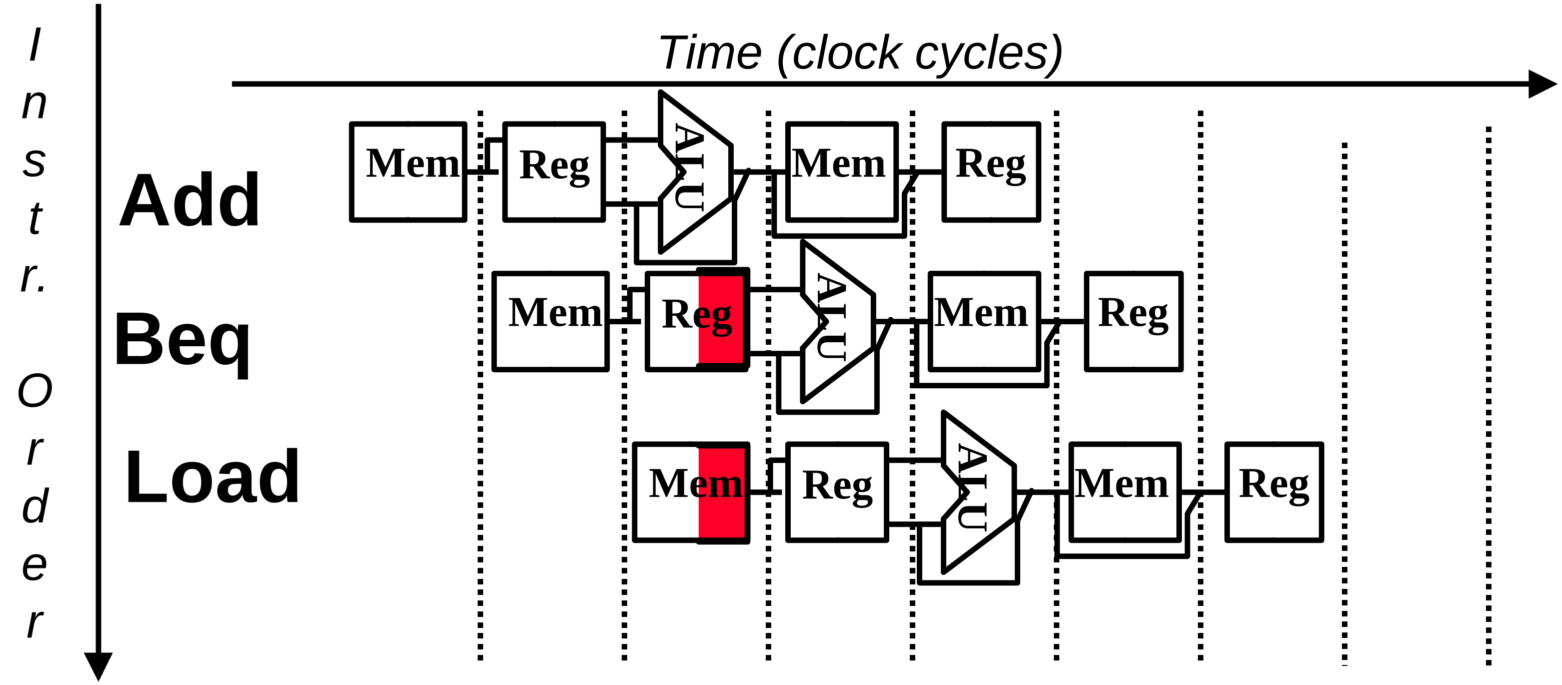
- Stall: wait until decision is clear
  - Its possible to move up decision to 2nd stage by adding hardware to check registers as being read



- Impact: 2 clock cycles per branch instruction => slow

Control Hazard Solutions

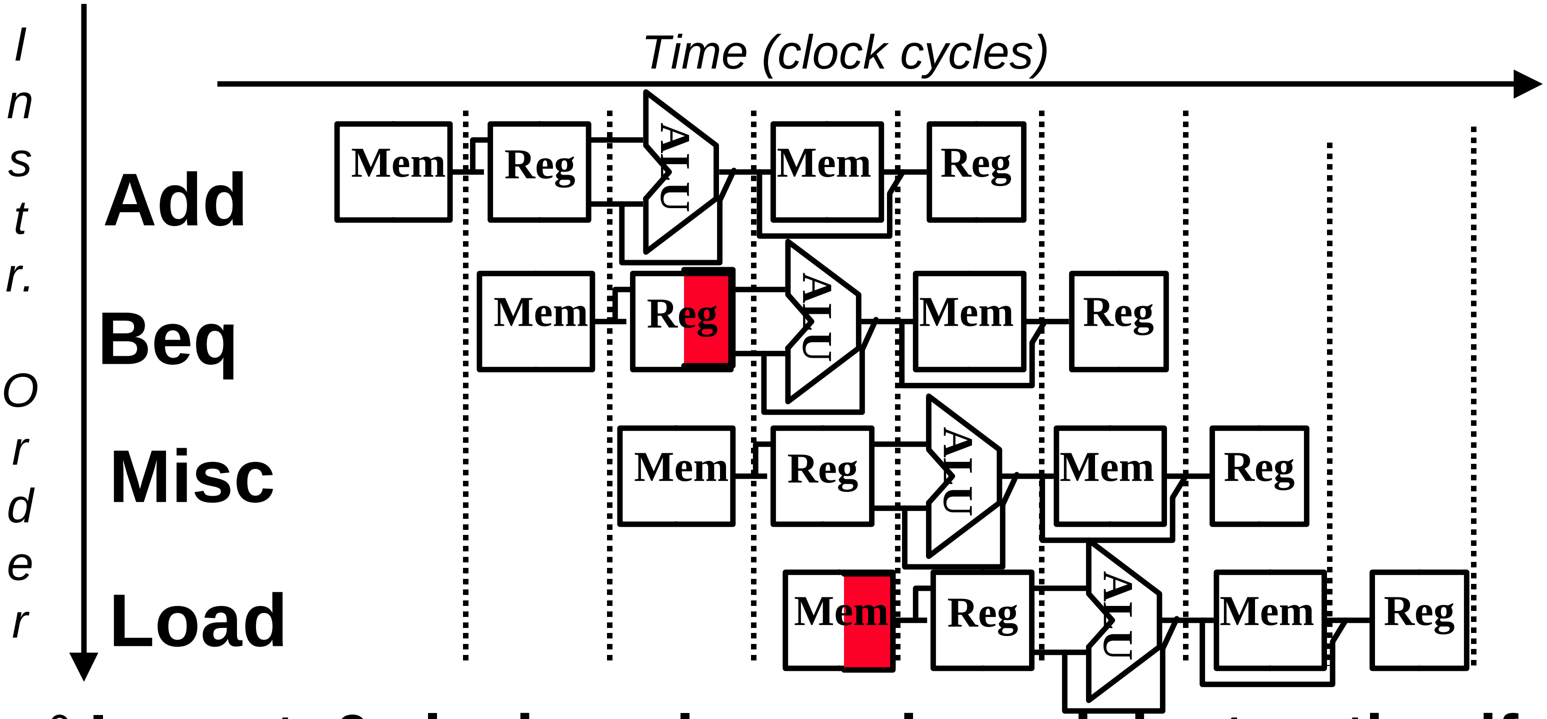
- Predict: guess one direction then back up if wrong
  - Predict not taken



- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right - 50% of time)
- More dynamic scheme: history of 1 branch (- 90%)

Control Hazard Solutions

- Redefine branch behavior (takes place after next instruction) “delayed branch”



- Impact: 0 clock cycles per branch instruction if can find instruction to put in “slot” (- 50% of time)

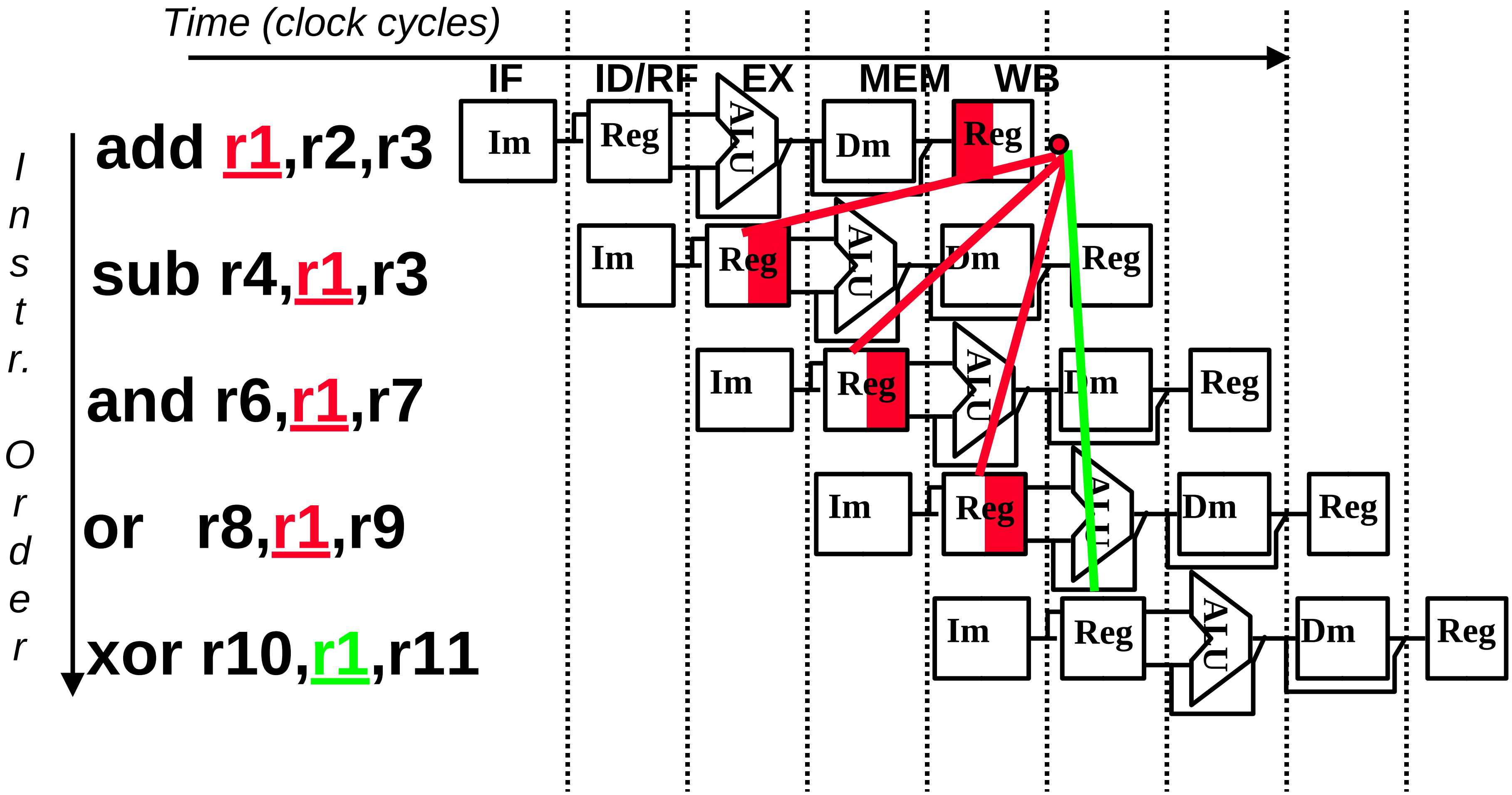
- As launch more instruction per clock cycle, less useful

Data Hazard on r1

```
add r1 ,r2,r3
sub r4, r1 ,r3
and r6, r1 ,r7
or   r8, r1 ,r9
xor r10, r1 ,r11
```

Data Hazard on r1:

- **Dependencies backwards in time are hazards**



Time (clock cycles)

Instr. Order

add r1,r2,r3

sub r4,r1,r3

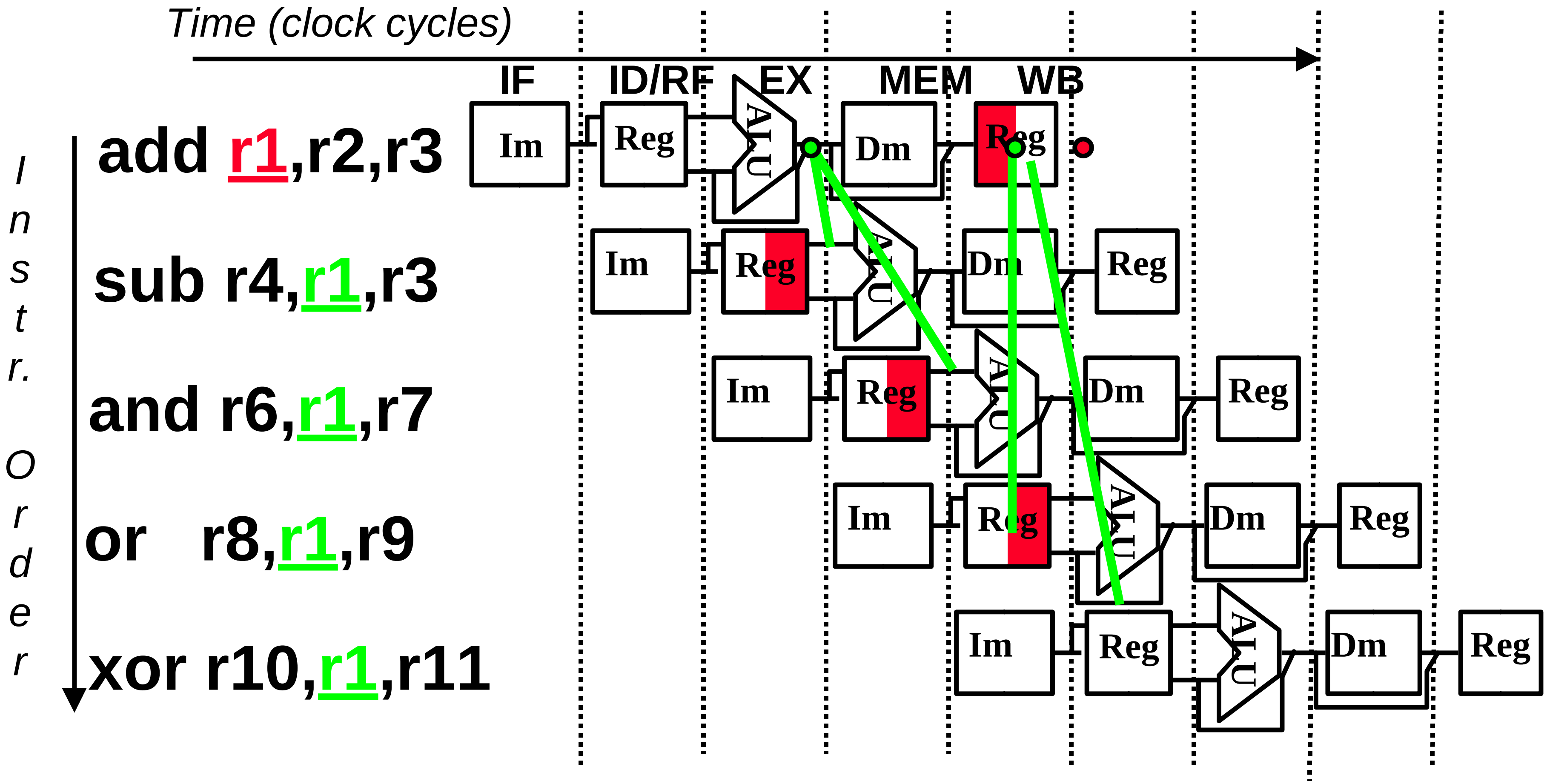
and r6,r1,r7

or r8,r1,r9

xor r10,r1,r11

Data Hazard Solution:

- “Forward” result from one stage to another



Time (clock cycles)

I n s t r. O r d e r

add r1,r2,r3

sub r4,r1,r3

and r6,r1,r7

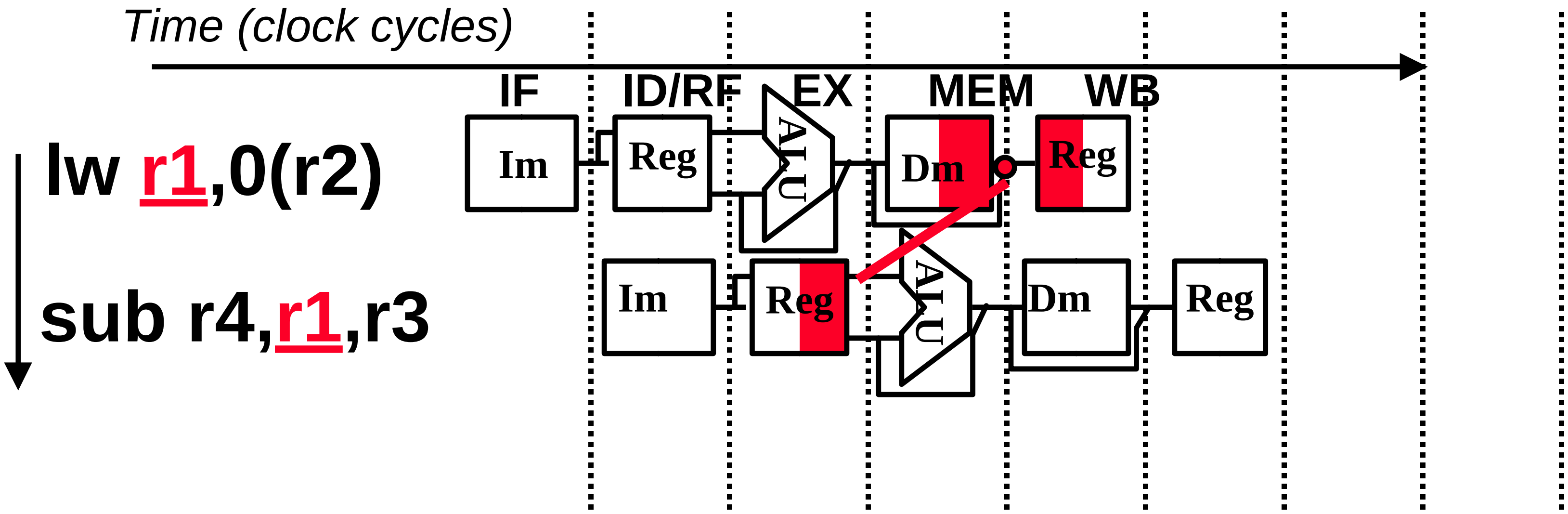
or r8,r1,r9

xor r10,r1,r11

- “or” OK if define read/write properly

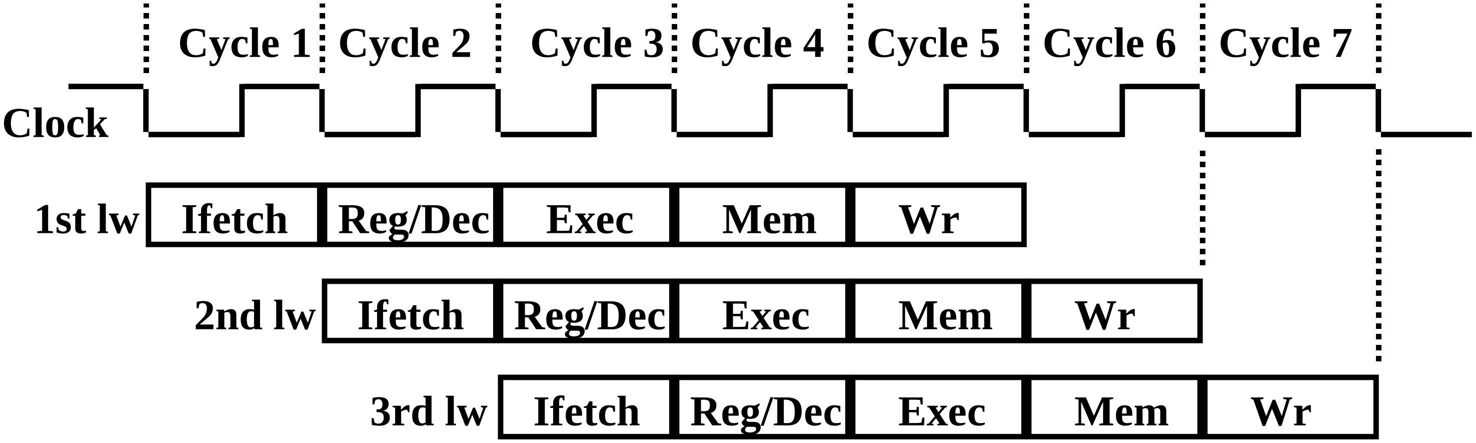
Forwarding (or Bypassing): What about Loads

- Dependencies backwards in time are hazards



- Can’t solve with forwarding:
- Must delay/stall instruction dependent on loads

Pipelining the Load Instruction

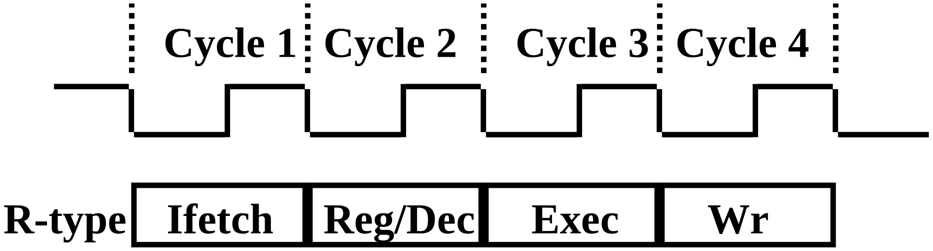


	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7
1st lw	Ifetch	Reg/Dec	Exec	Mem	Wr		
2nd lw		Ifetch	Reg/Dec	Exec	Mem	Wr	
3rd lw			Ifetch	Reg/Dec	Exec	Mem	Wr

° **The five independent functional units in the pipeline datapath are:**

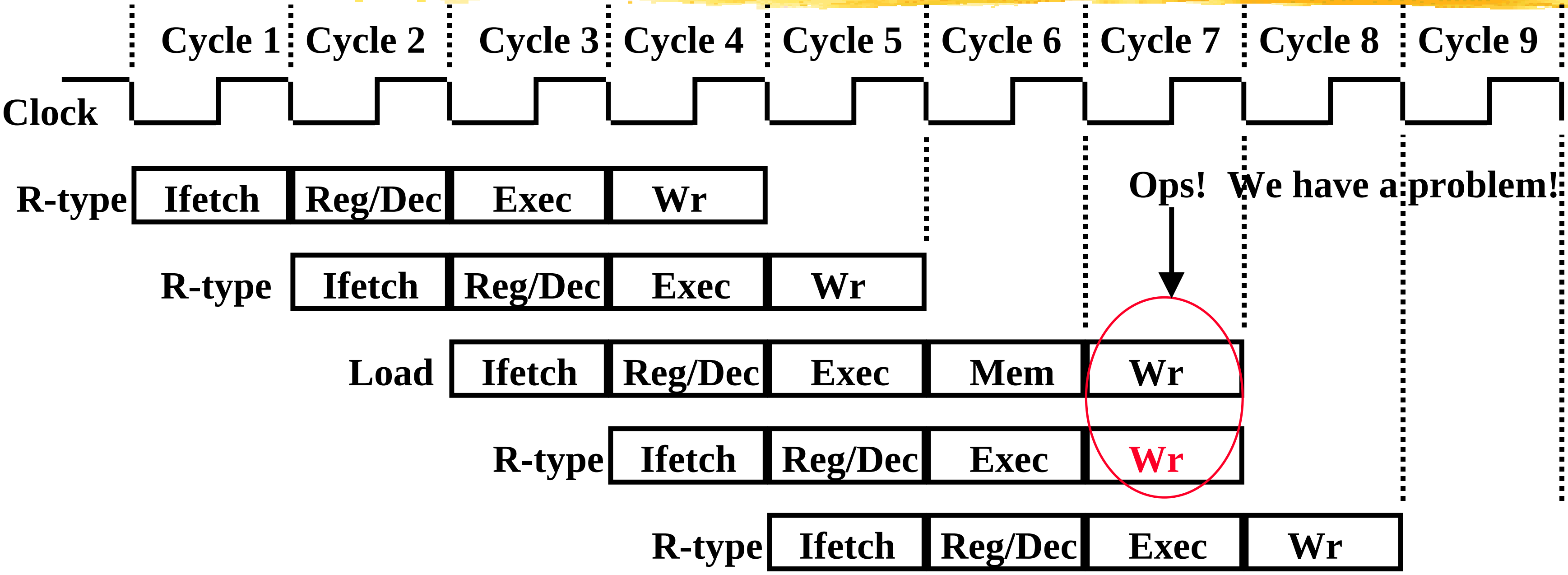
- **Instruction Memory for the Ifetch stage**
- **Register File’s Read ports (bus A and busB) for the Reg/Dec stage**
- **ALU for the Exec stage**
- **Data Memory for the Mem stage**
- **Register File’s Write port (bus W) for the Wr stage**

The Four Stages of R-type



- **Ifetch: Instruction Fetch**
  - **Fetch the instruction from the Instruction Memory**
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec:**
  - **ALU operates on the two register operands**
  - **Update PC**
- **Wr: Write the ALU output back to the register file**

Pipelining the R-type and Load Instruction



- We have pipeline conflict or structural hazard:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

Important Observation

- Each functional unit can only be used **once** per instruction

- Each functional unit must be used at the **same** stage for all instructions:
  - Load uses Register File’s Write Port during its 5th stage

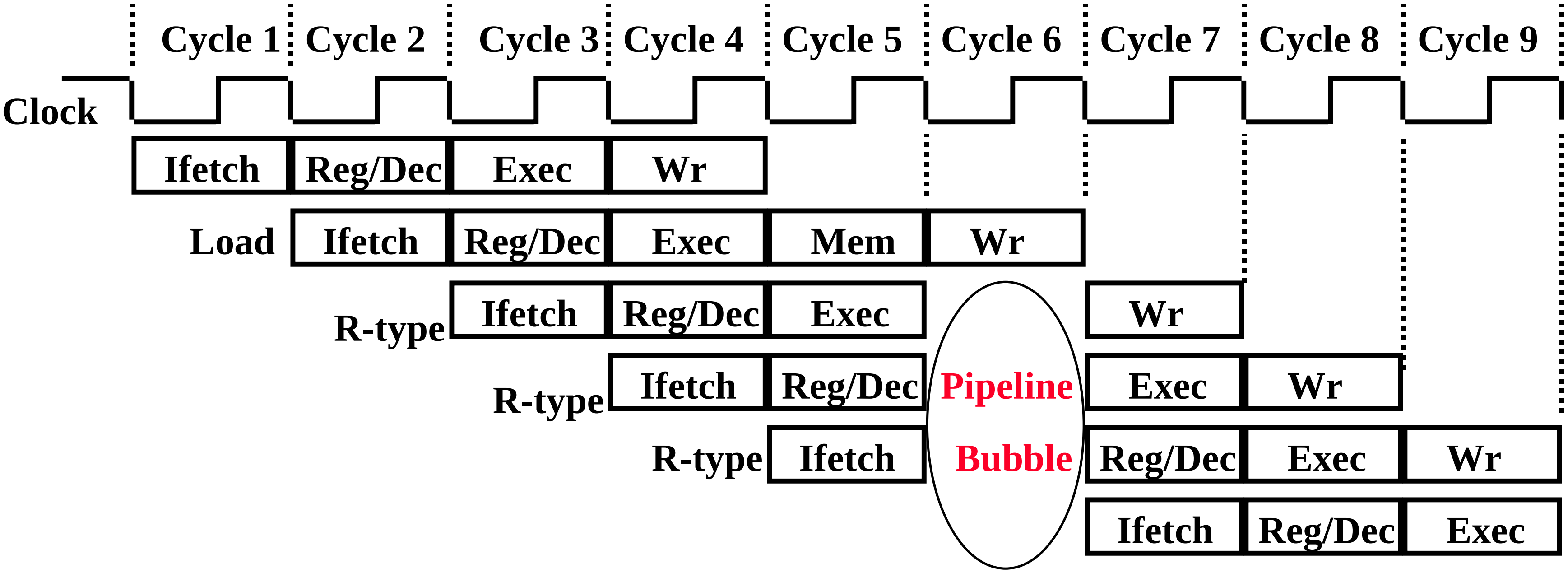
	1	2	3	4	5
Load	Ifetch	Reg/Dec	Exec	Mem	Wr

  - R-type uses Register File’s Write Port during its 4th stage

	1	2	3	4
R-type	Ifetch	Reg/Dec	Exec	Wr

- 2 ways to solve this pipeline hazard.

Solution 1: Insert "Bubble" into the Pipeline

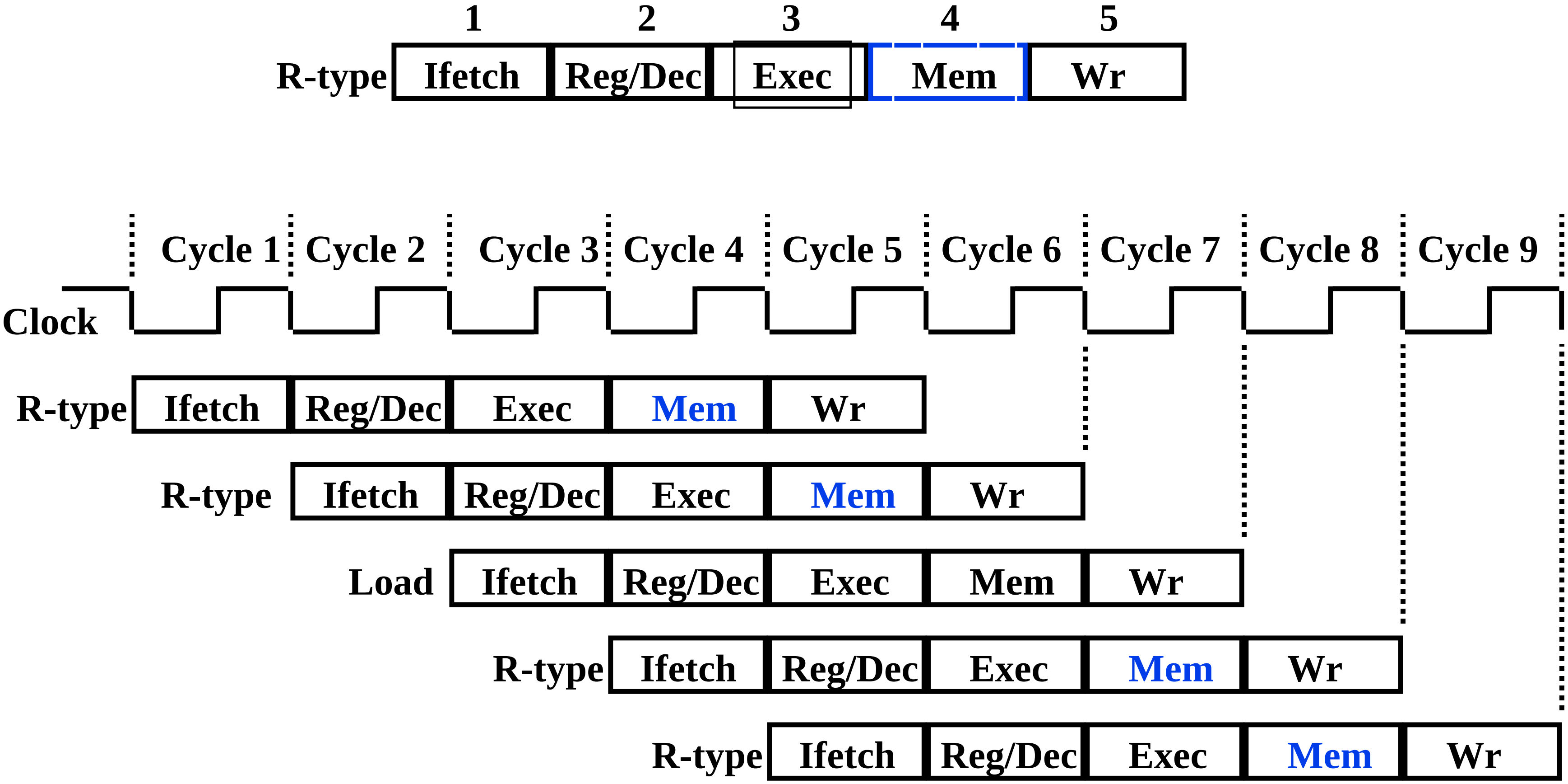


- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!

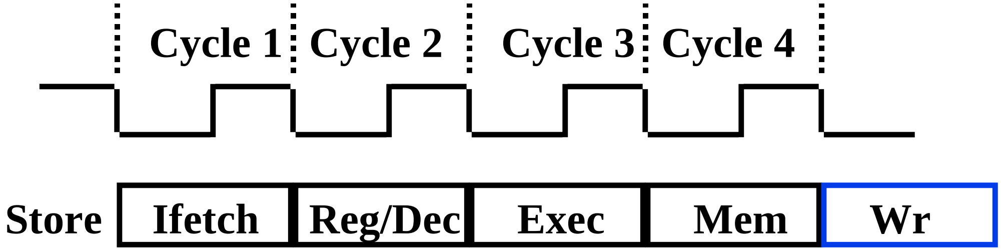
Solution 2: Delay R-type’s Write by One Cycle

° Delay R-type’s register write by one cycle:

- Now R-type instructions also use Reg File’s write port at Stage 5
- Mem stage is a NOOP stage: nothing is being done.

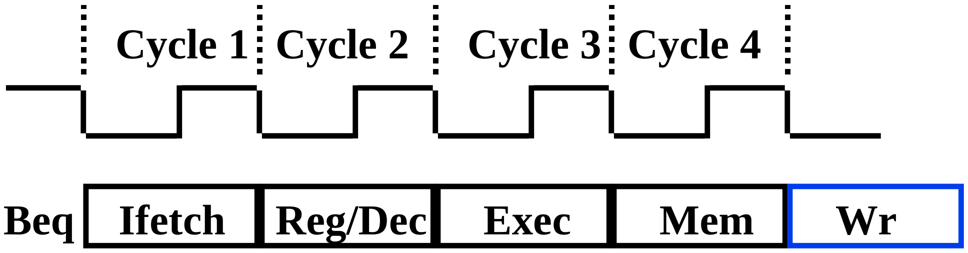


The Four Stages of Store



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

The Three Stages of Beq



Cycle 1 Cycle 2 Cycle 3 Cycle 4

Beq | Ifetch | Reg/Dec | Exec | Mem | Wr

- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec:
  - Registers Fetch and Instruction Decode
- Exec:
  - compares the two register operand,
  - select correct branch target address
  - latch into PC

Summary: Pipelining

- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction
- We’ll build a simple pipeline and look at these issues
- We’ll talk about modern processors and what really makes it hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

Summary

- **Pipelining is a fundamental concept**
  - **multiple steps using distinct resources**
- **Utilize capabilities of the Datapath by pipelined instruction processing**
  - **start next instruction while working on the current one**
  - **limited by length of longest stage (plus fill/flush)**
  - **detect and resolve hazards**