

Chapter 6: Pipelining

Outline

- An overview of pipelining
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards
- Exceptions
- Superscalar and dynamic pipelining

Pipelining Is Natural!

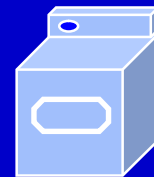
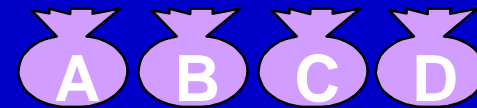
- Laundry example:

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

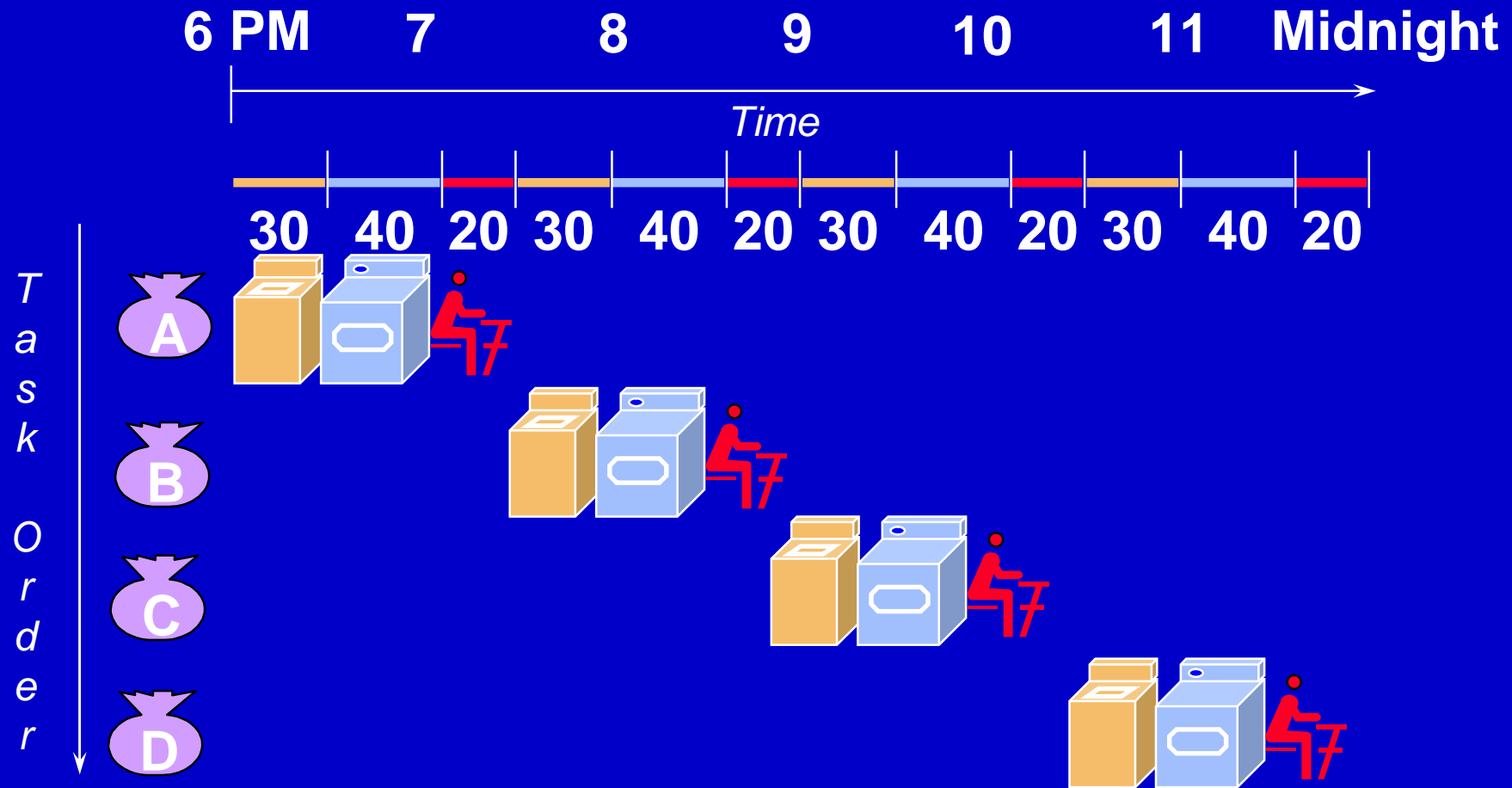
Washer takes 30 minutes

Dryer takes 40 minutes

“Folder” takes 20 minutes

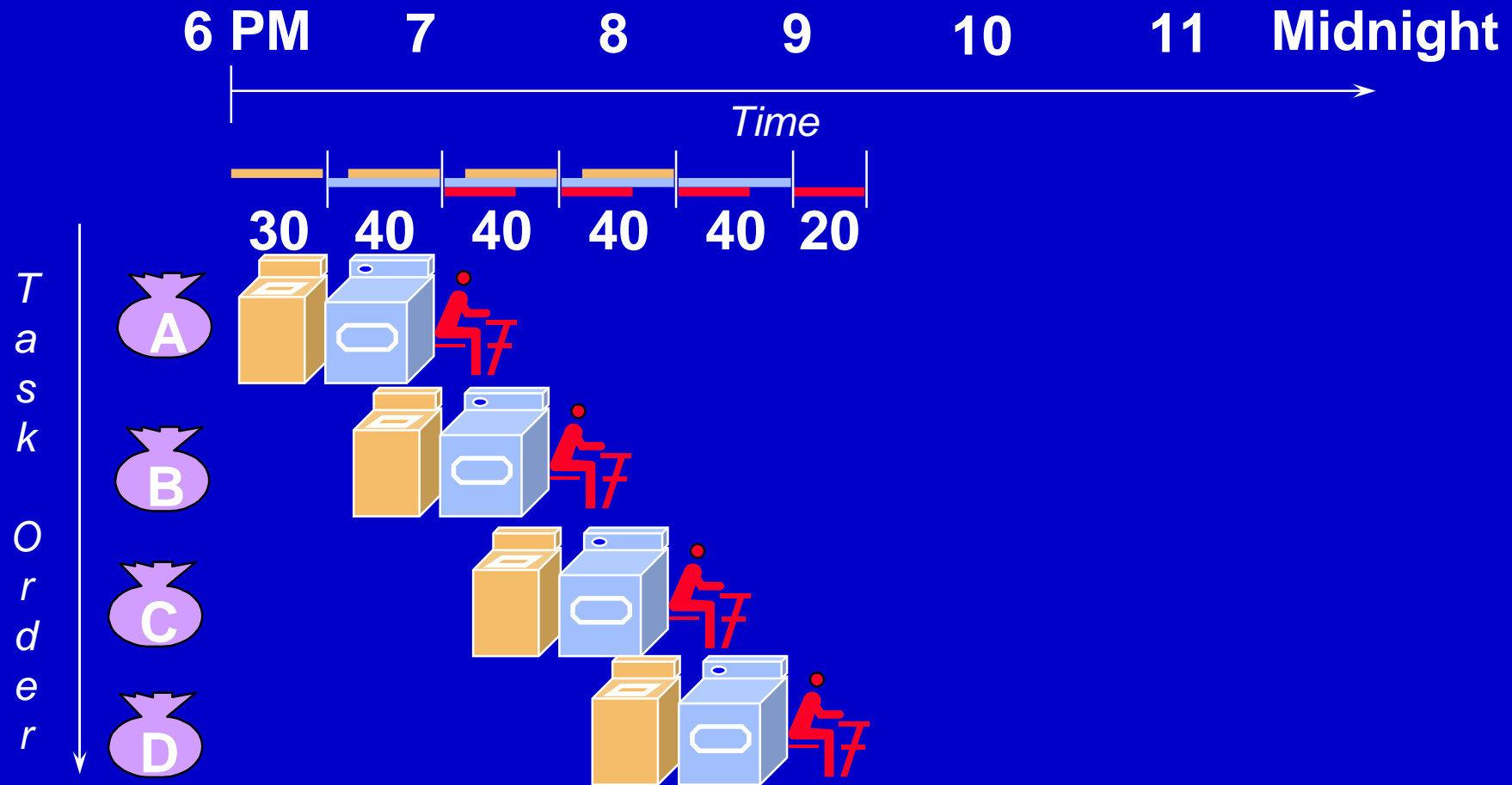


Sequential Laundry



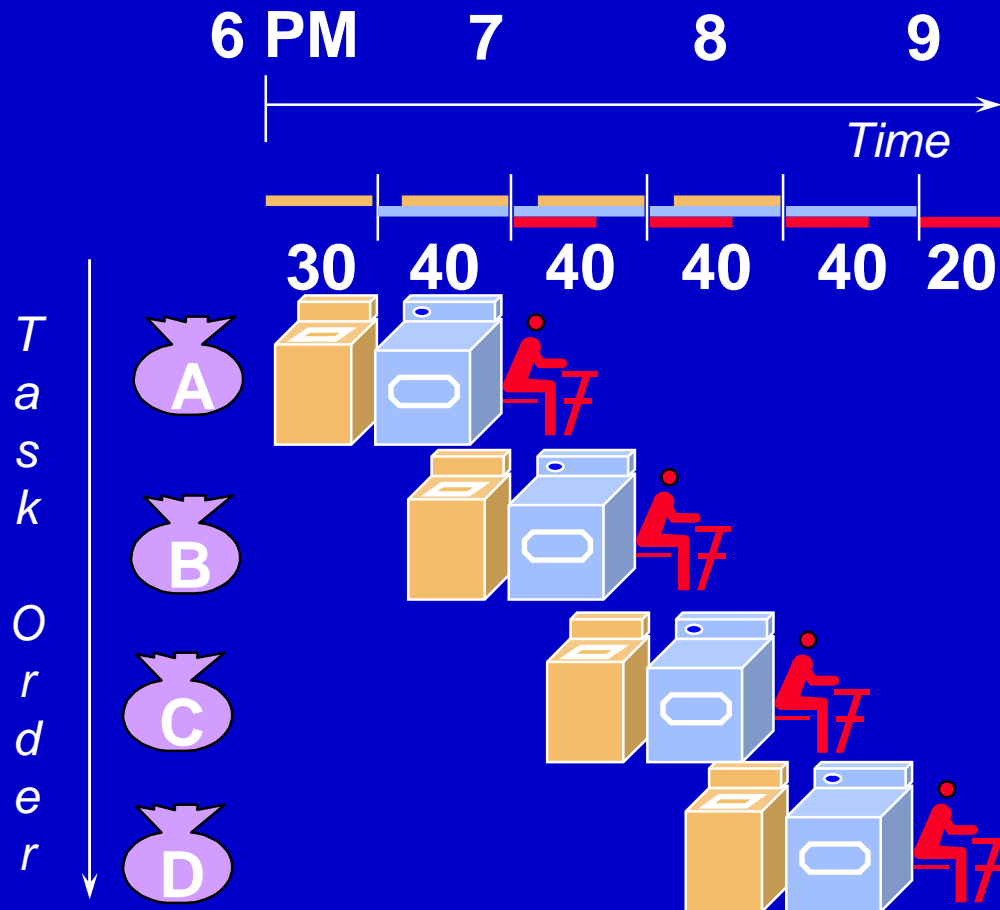
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would it take?

Pipelined Laundry: Start ASAP



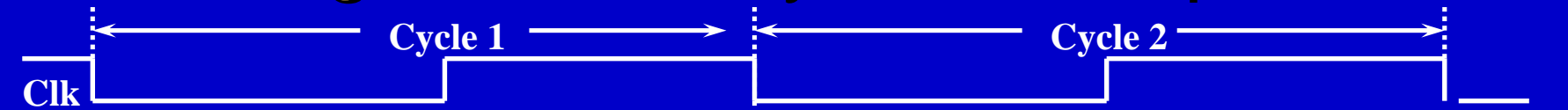
- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons

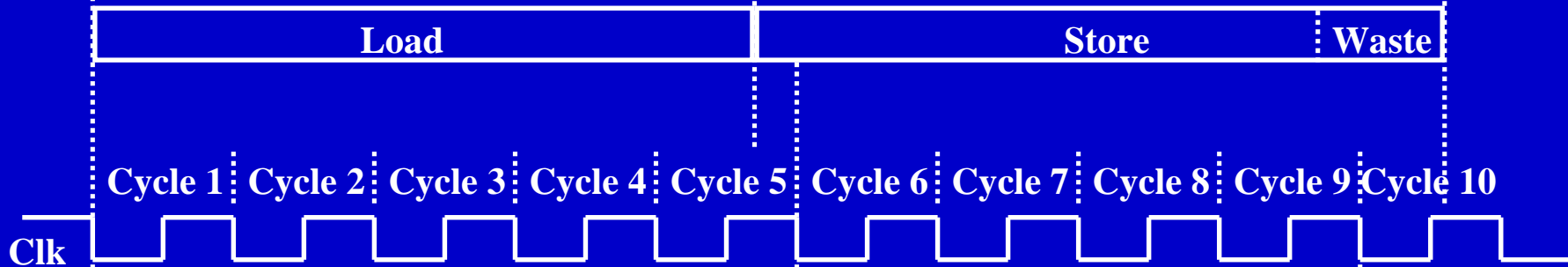


- Doesn't help **latency** of single task, but **throughput** of entire
- Pipeline rate limited by **slowest** stage
- **Multiple** tasks working at same time using different resources
- Potential speedup = **Number of pipe stages**
- Unbalanced stage length; time to **"fill"** & **"drain"** the pipeline reduce speedup
- Stall for dependences

Single-, Multi-Cycle, vs. Pipeline



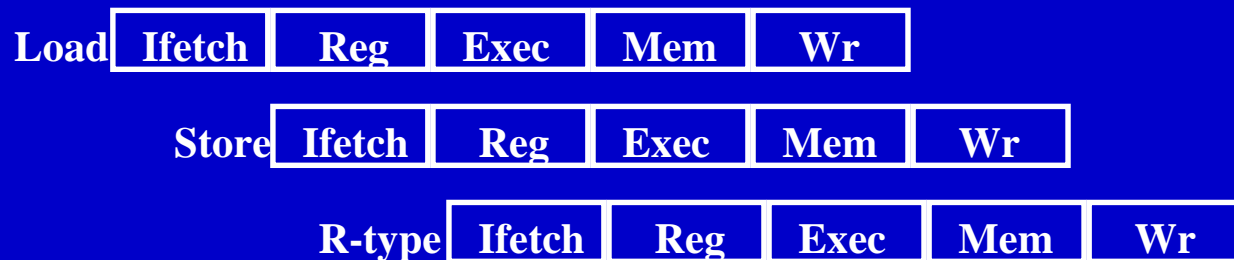
Single Cycle Implementation:



Multiple Cycle Implementation:



Pipeline Implementation:



Pipelining MIPS Execution

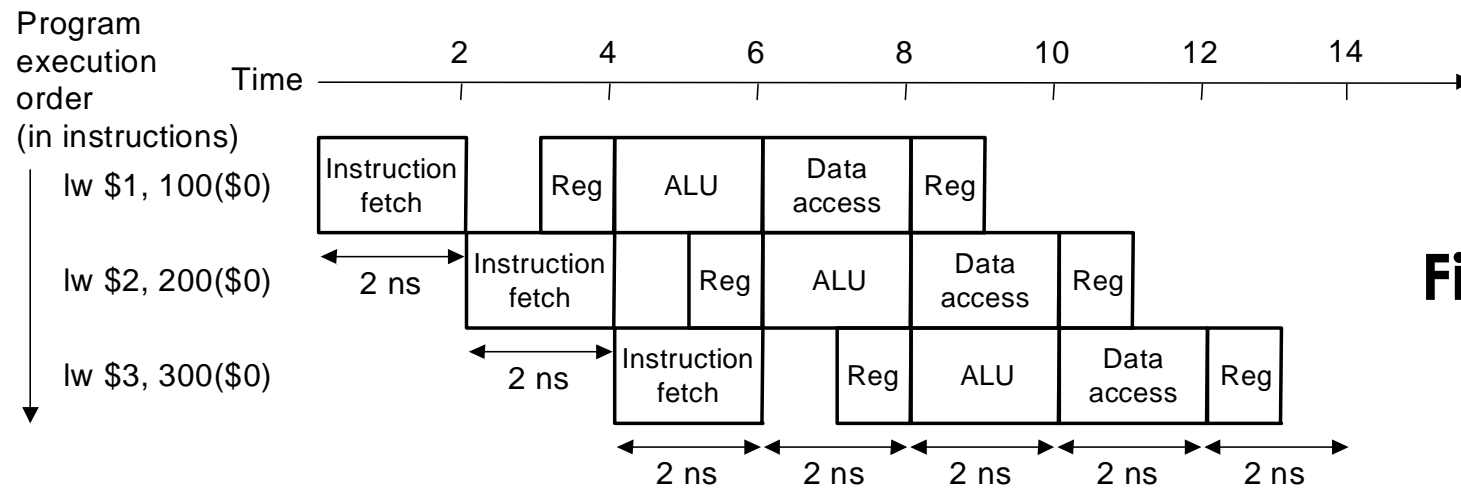
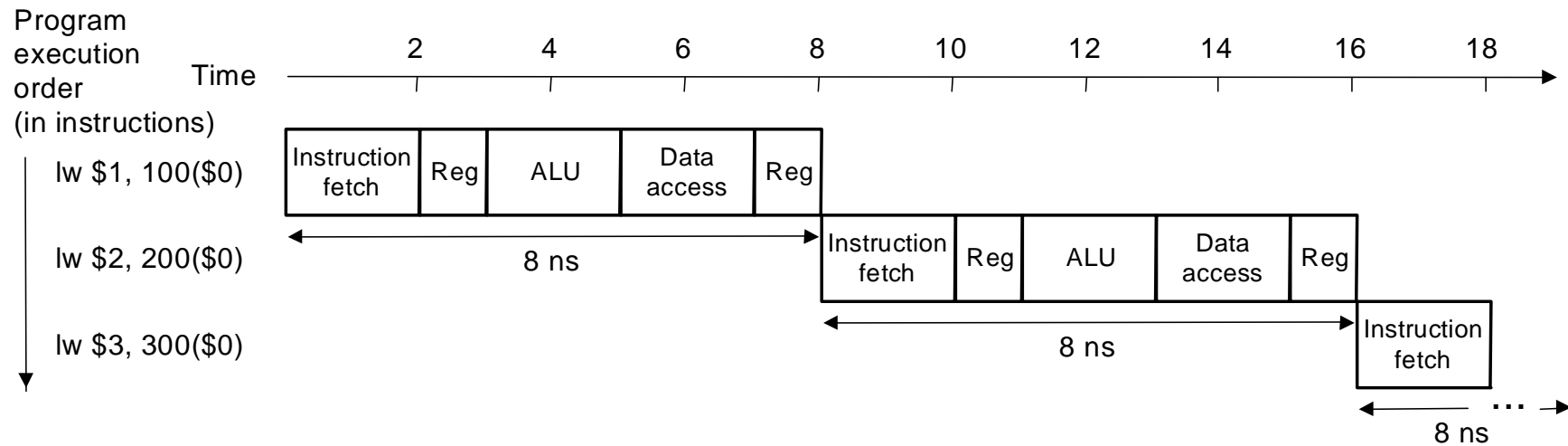
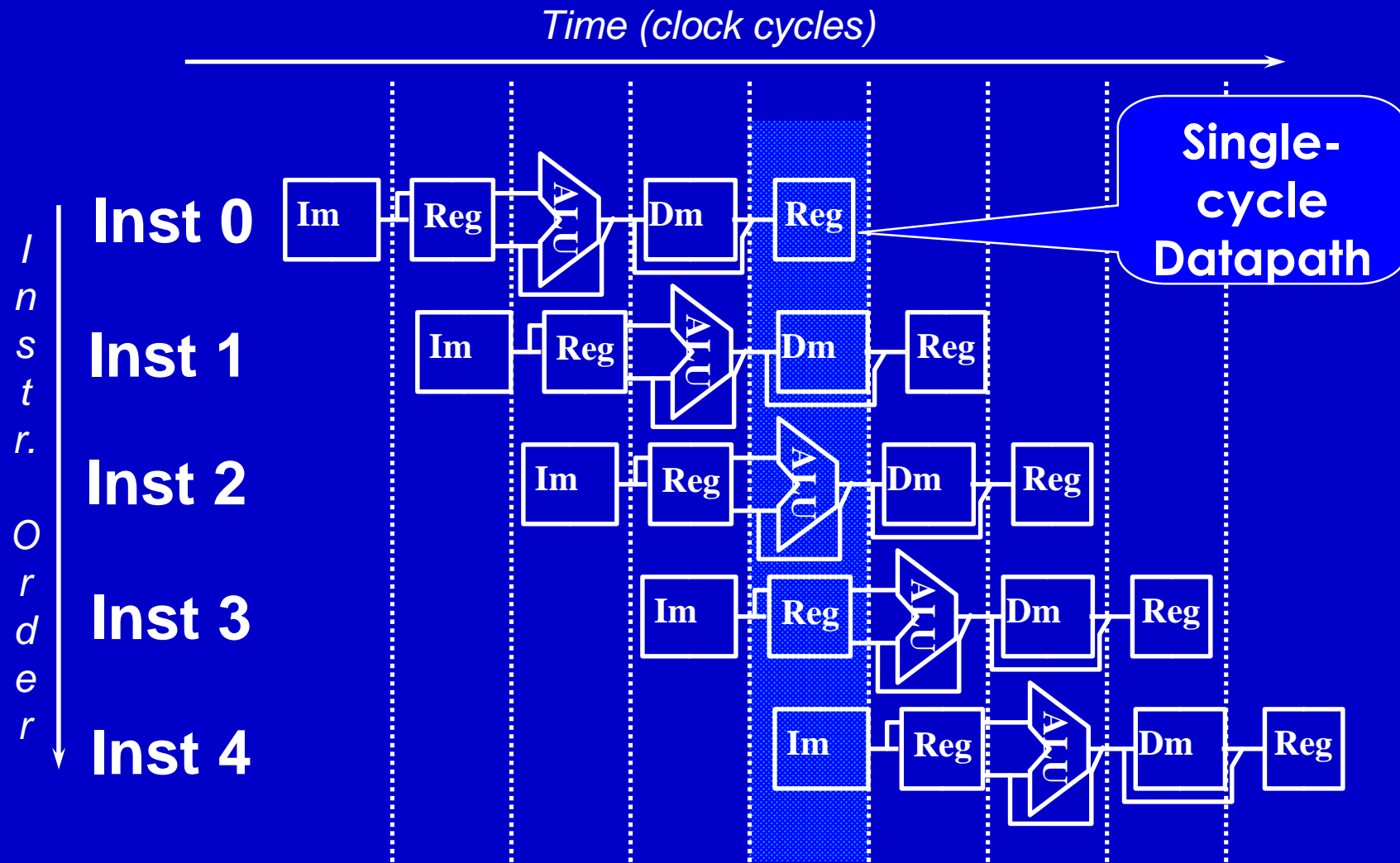


Fig. 6.3

Why Pipeline? Because the Resources Are There!



Hazard

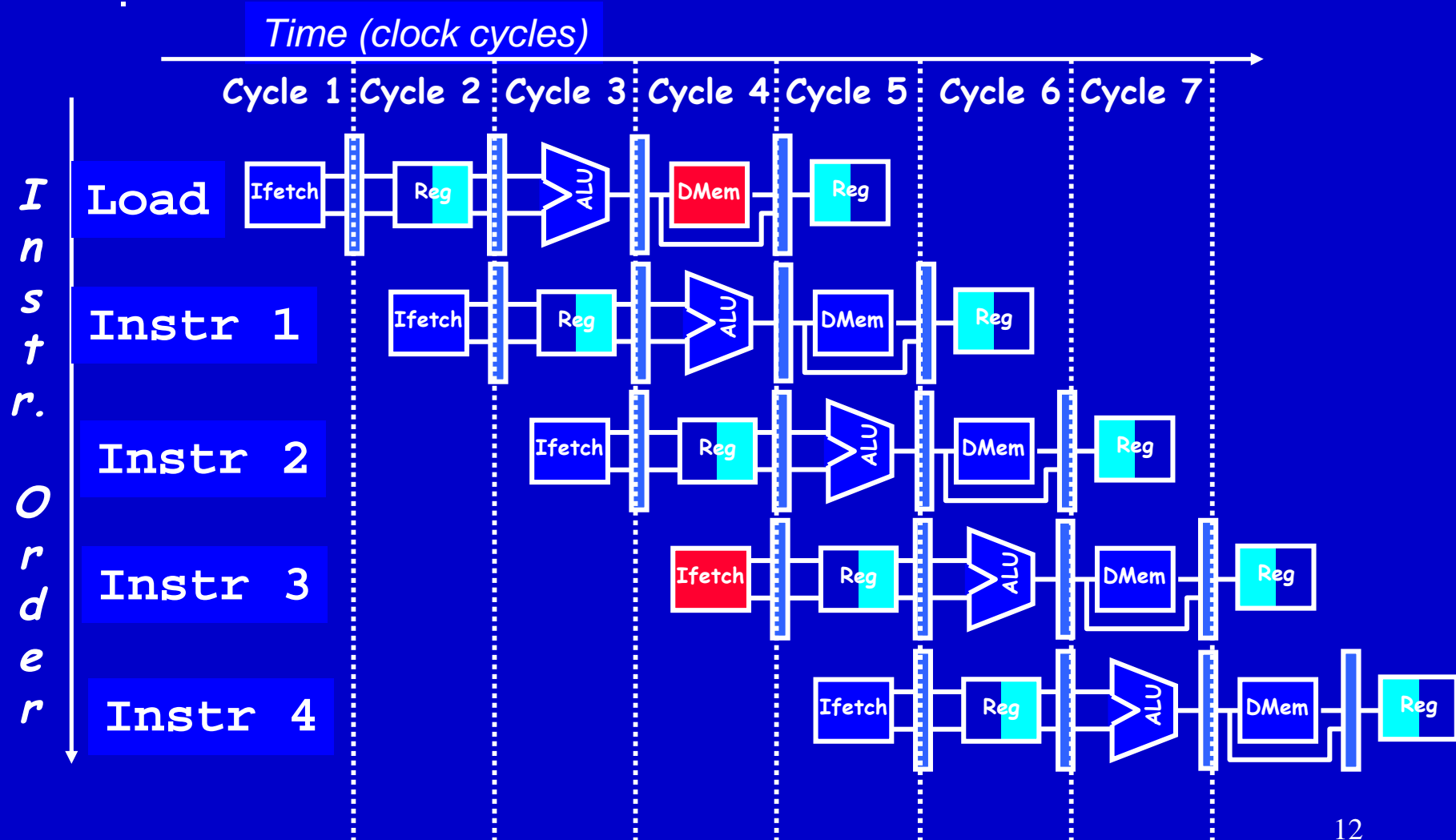
- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
 - **Structural hazards**: Hardware cannot support this combination of instructions - two instructions need the same resource.
 - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
 - **Control hazards**: Pipelining of branches & other instructions that change the PC
- Common solution is to **stall** the pipeline until the hazard is resolved, inserting one or more “**bubbles**” in the pipeline
- To do this, hardware or software must detect that a hazard has occurred.

Structural Hazards

- Structural hazards occur when two or more instructions need the same resource.
- Common methods for eliminating structural hazards are:
 - Duplicate resources
 - Pipeline the resource
 - Reorder the instructions
- It may be too expensive to eliminate a structural hazard, in which case the pipeline should stall.
- When the pipeline stalls, no instructions are issued until the hazard has been resolved.
- What are some examples of structural hazards?

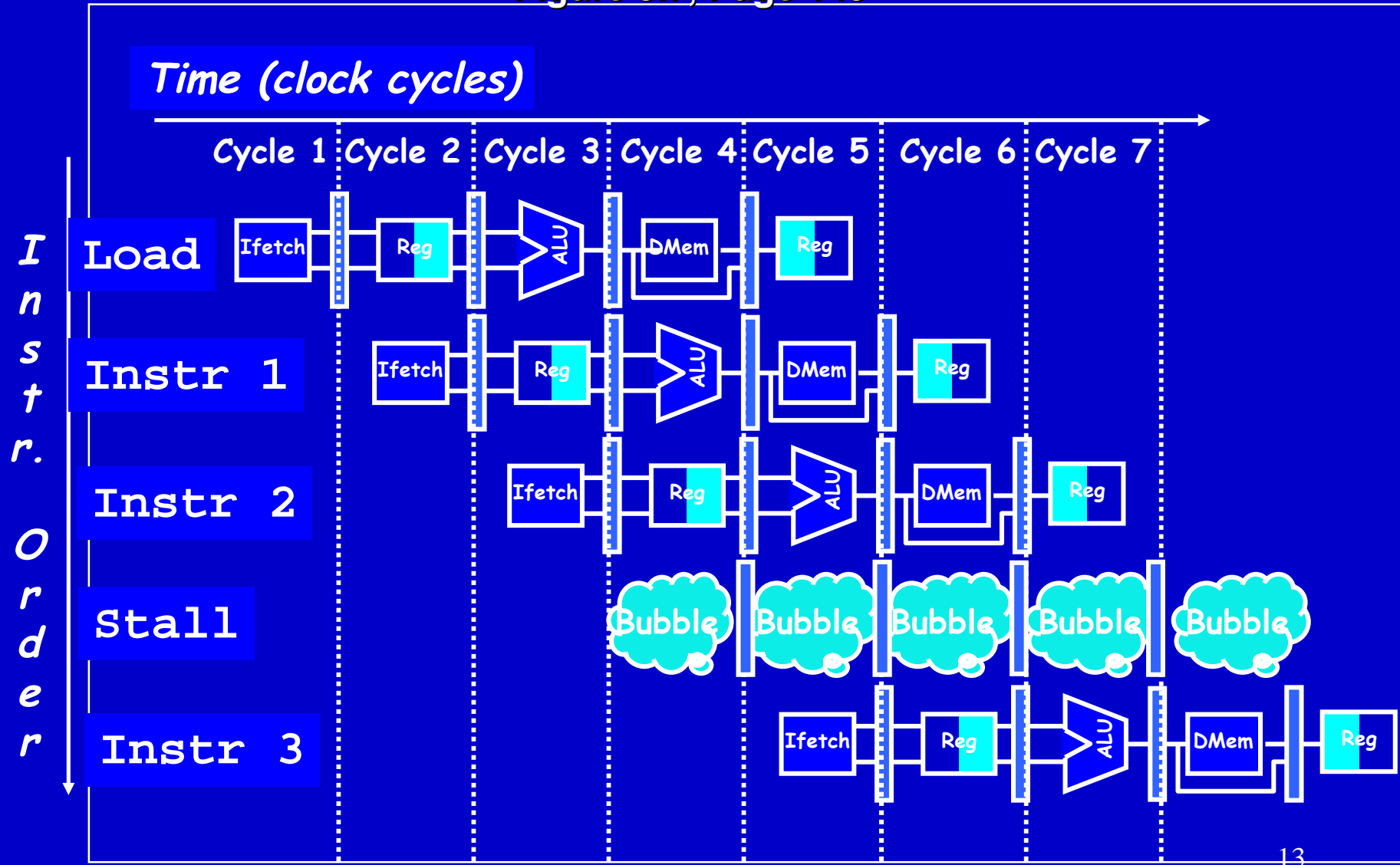
One Memory Port Structural Hazards

Figure 3.6, Page 142



One Memory Port Structural Hazards

Figure 3.7, Page 143



Outline

- An overview of pipelining
- A pipelined datapath (6.2)
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards
- Exceptions
- Superscalar and dynamic pipelining

Designing a Pipelined Processor

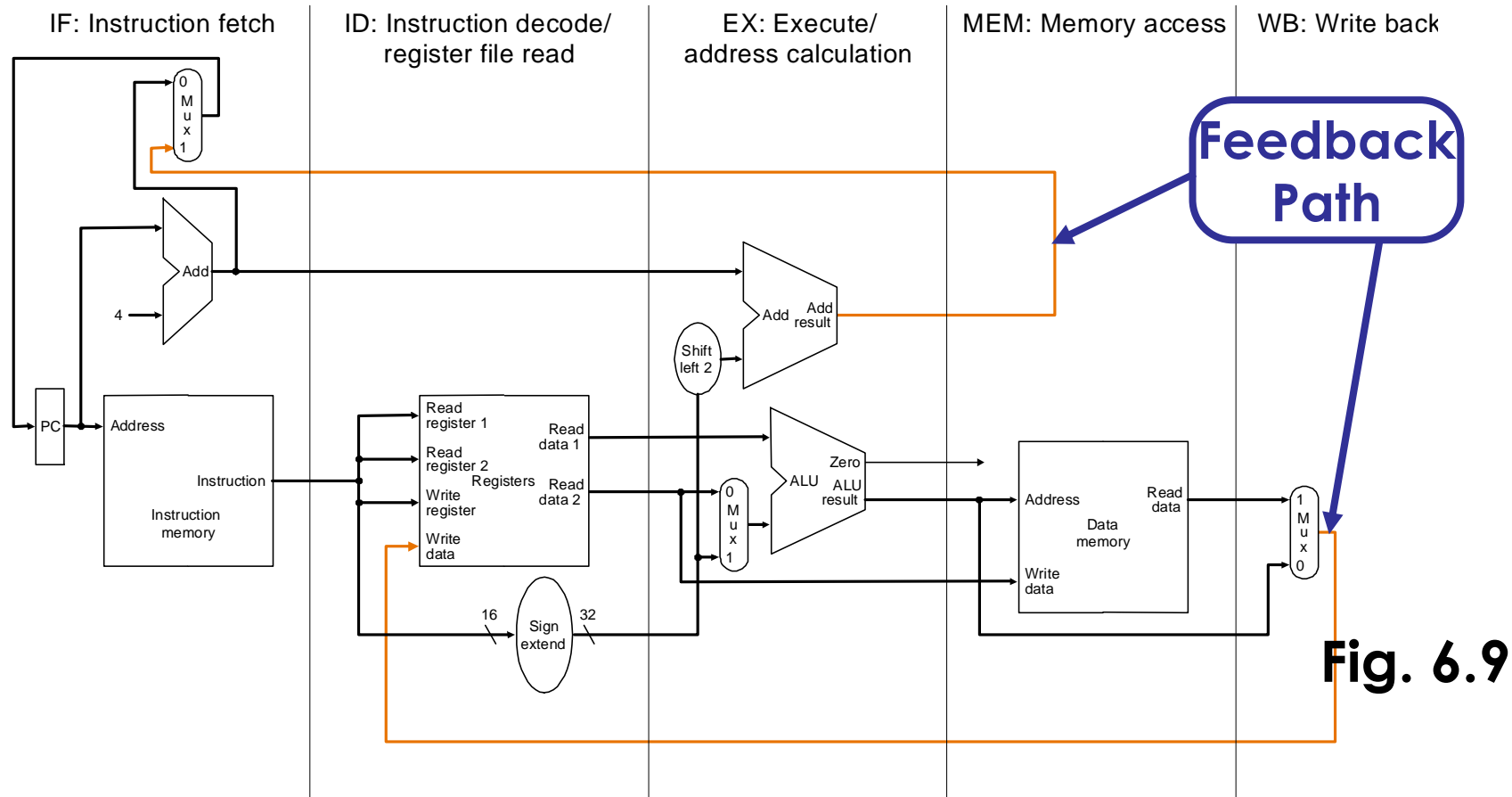
- Examine the datapath and control diagram
 - Starting with single- or multi-cycle datapath?
 - Single- or multi-cycle control?
- Partition datapath into stages:
 - IF (instruction fetch)
 - ID (instruction decode and register file read)
 - EX (execution or address calculation)
 - MEM (data memory access)
 - WB (write back)
- Associate resources with stages
- Ensure that flows do not conflict, or figure out how to resolve
- Assert control in appropriate stage

Use Multicycle Execution Steps

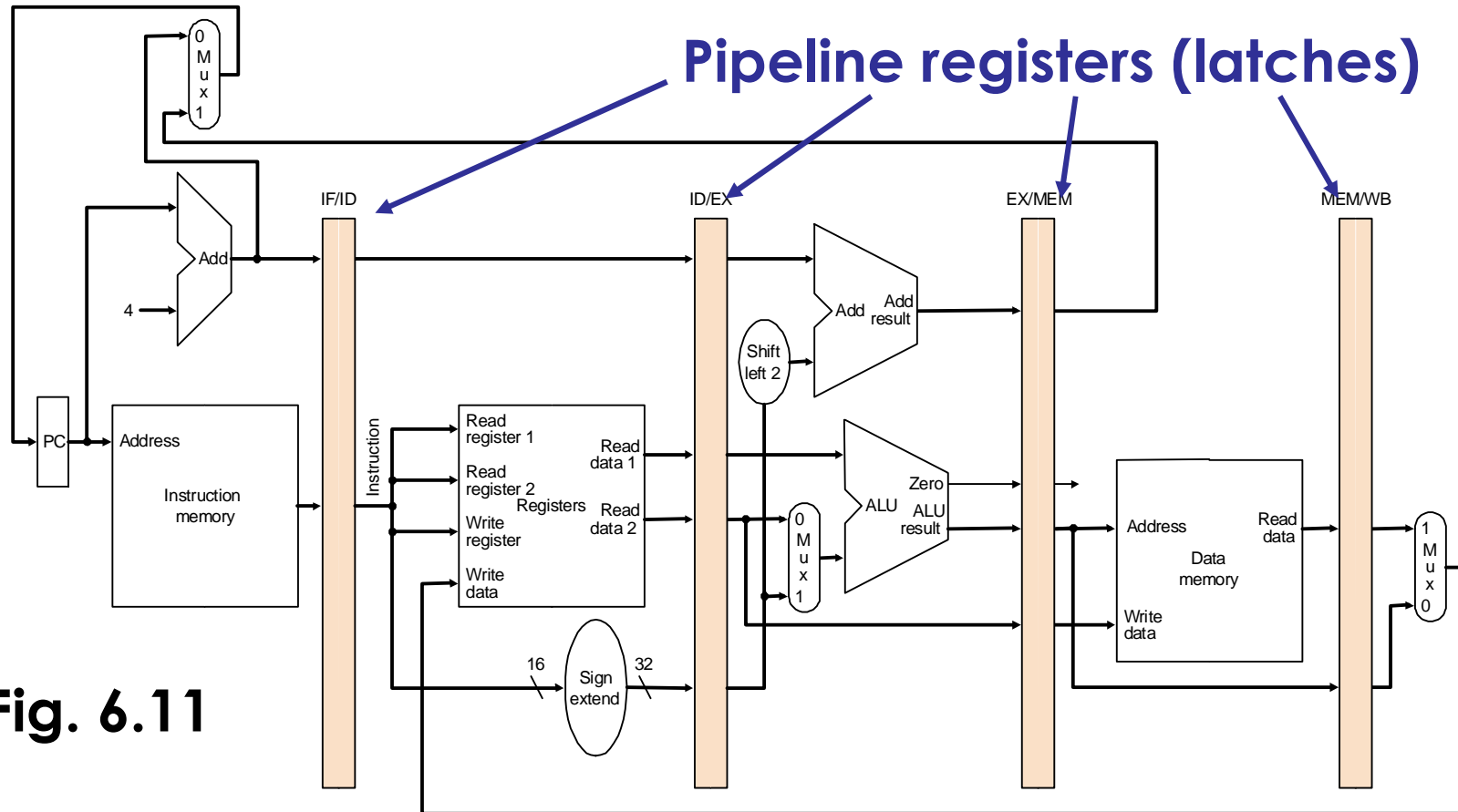
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	$IR = \text{Memory}[PC]$ $PC = PC + 4$			
Instruction decode/register fetch	$A = \text{Reg}[IR[25-21]]$ $B = \text{Reg}[IR[20-16]]$ $ALUOut = PC + (\text{sign-extend}(IR[15-0]) \ll 2)$			
Execution, address computation, branch/jump completion	$ALUOut = A \text{ op } B$	$ALUOut = A + \text{sign-extend}(IR[15-0])$	if $(A == B)$ then $PC = ALUOut$	$PC = PC[31-28] \parallel (IR[25-0] \ll 2)$
Memory access or R-type completion	$\text{Reg}[IR[15-11]] = ALUOut$	Load: $MDR = \text{Memory}[ALUOut]$ or Store: $\text{Memory}[ALUOut] = B$		
Memory read completion		Load: $\text{Reg}[IR[20-16]] = MDR$		

But, use single-cycle datapath ...

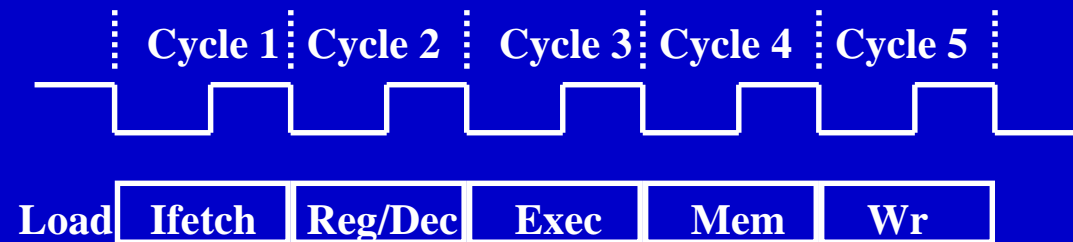
Split Single-cycle Datapath



Add Pipeline Registers

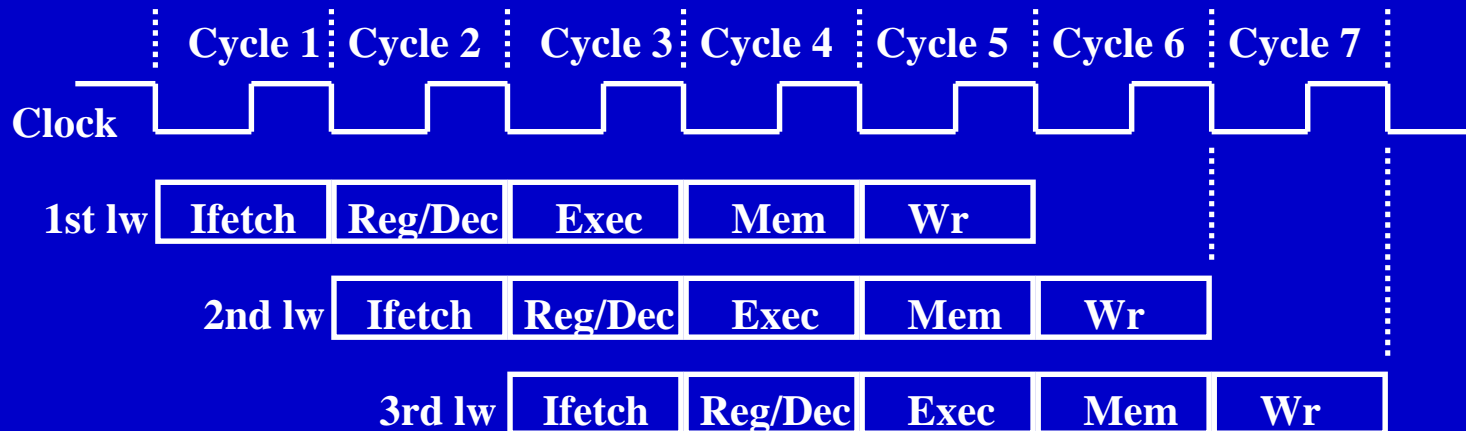


Consider Load



- **IF: Instruction Fetch**
 - Fetch the instruction from the Instruction Memory
- **ID: Instruction Decode**
 - Registers fetch and instruction decode
- **EX: Calculate the memory address**
- **MEM: Read the data from the Data Memory**
- **WB: Write the data back to the register file**

Pipelining load



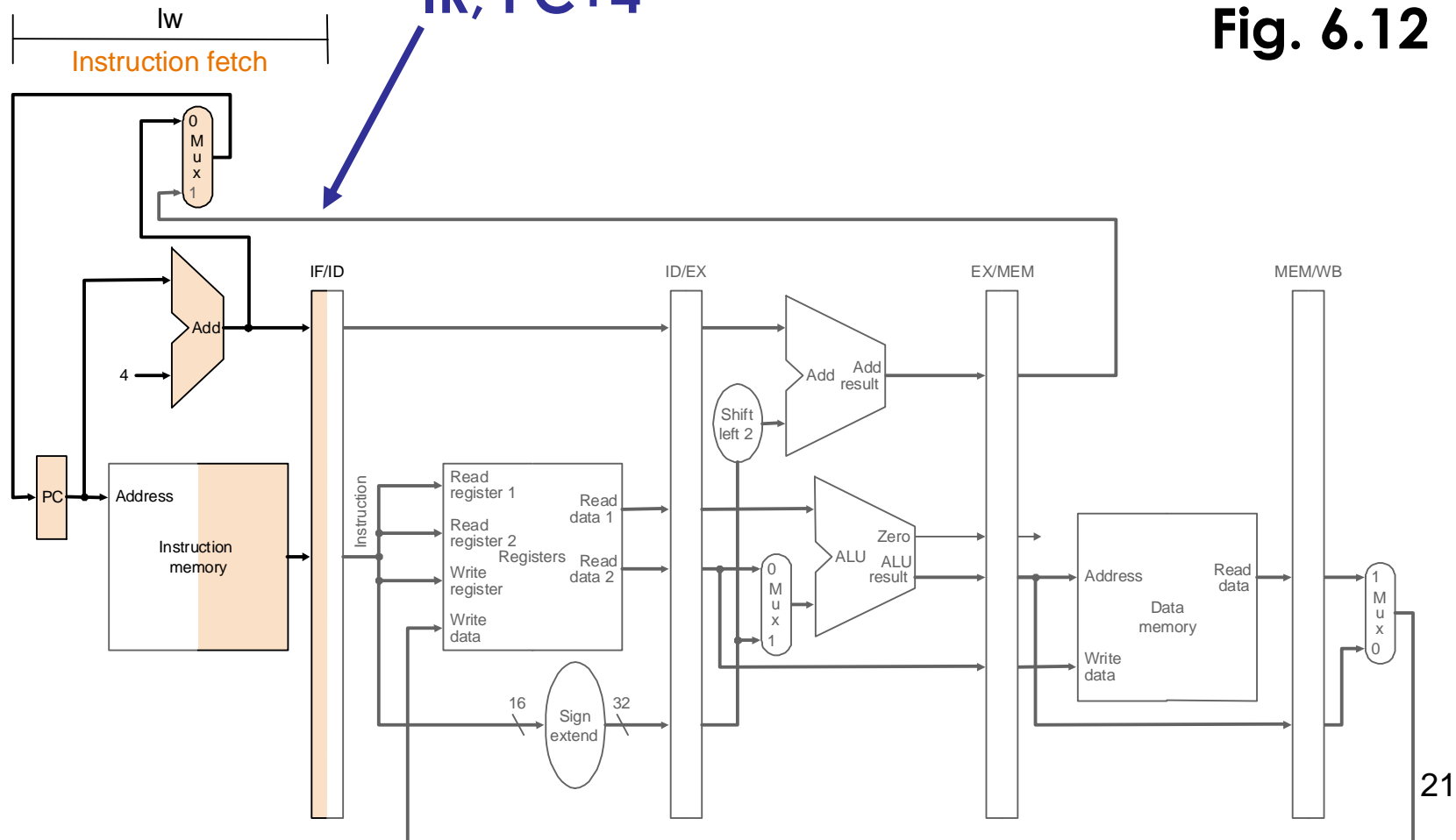
- 5 functional units in the pipeline datapath are:
 - **Instruction Memory** for the Ifetch stage
 - Register File's **Read ports** (busA and busB) for the Reg/Dec stage
 - **ALU** for the Exec stage
 - **Data Memory** for the MEM stage
 - Register File's **Write port** (busW) for the WB stage

IF Stage of Load

- $IR = \text{mem}[PC]; \quad PC = PC + 4$

IR, PC+4

Fig. 6.12



ID Stage of load

- $A = \text{Reg}[\text{IR}[25-21]]$; $B = \text{Reg}[\text{IR}[20-16]]$;

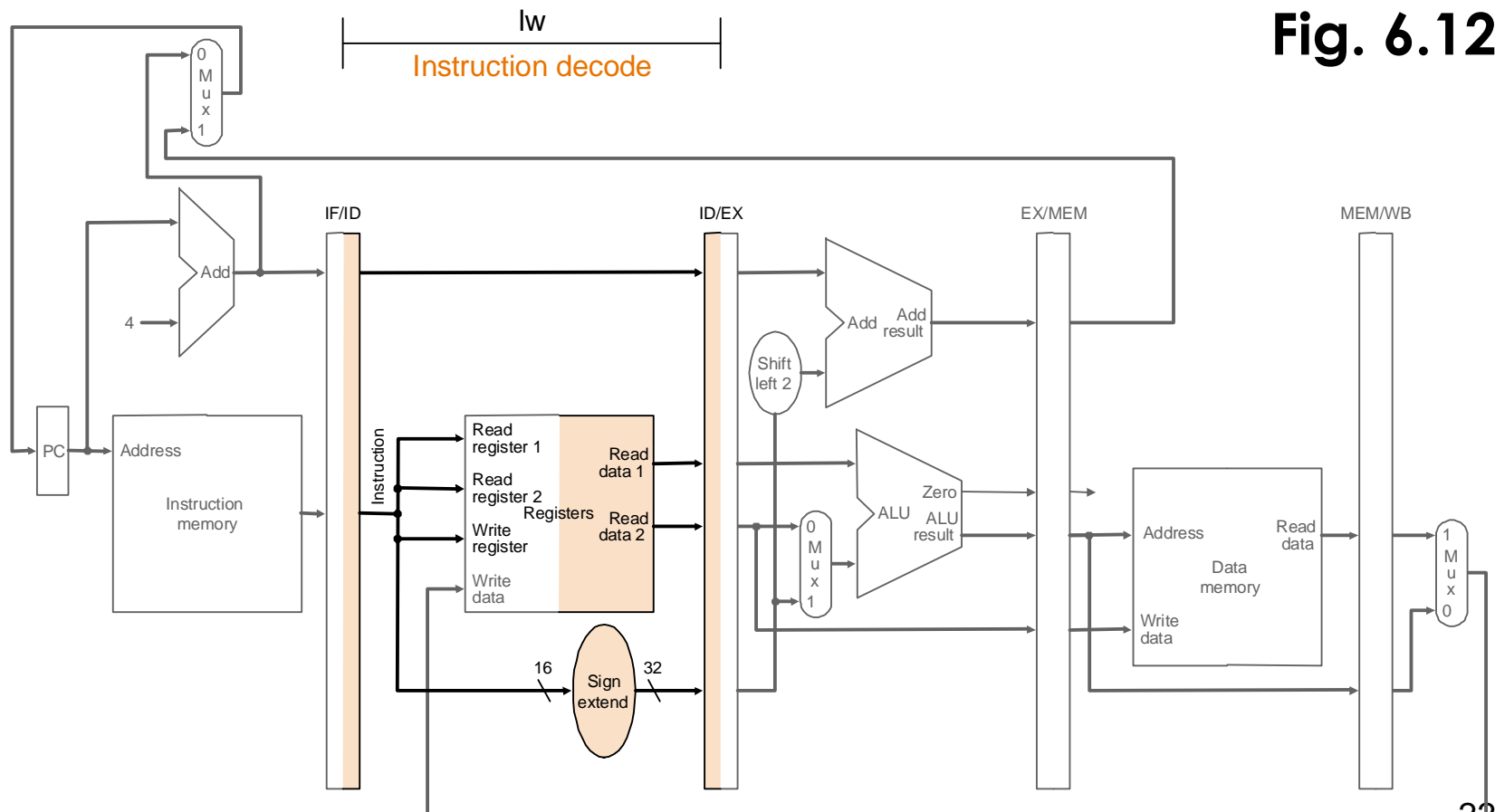


Fig. 6.12

EX Stage of load

- $ALU_{out} = A + \text{sign-ext}(IR[15-0])$

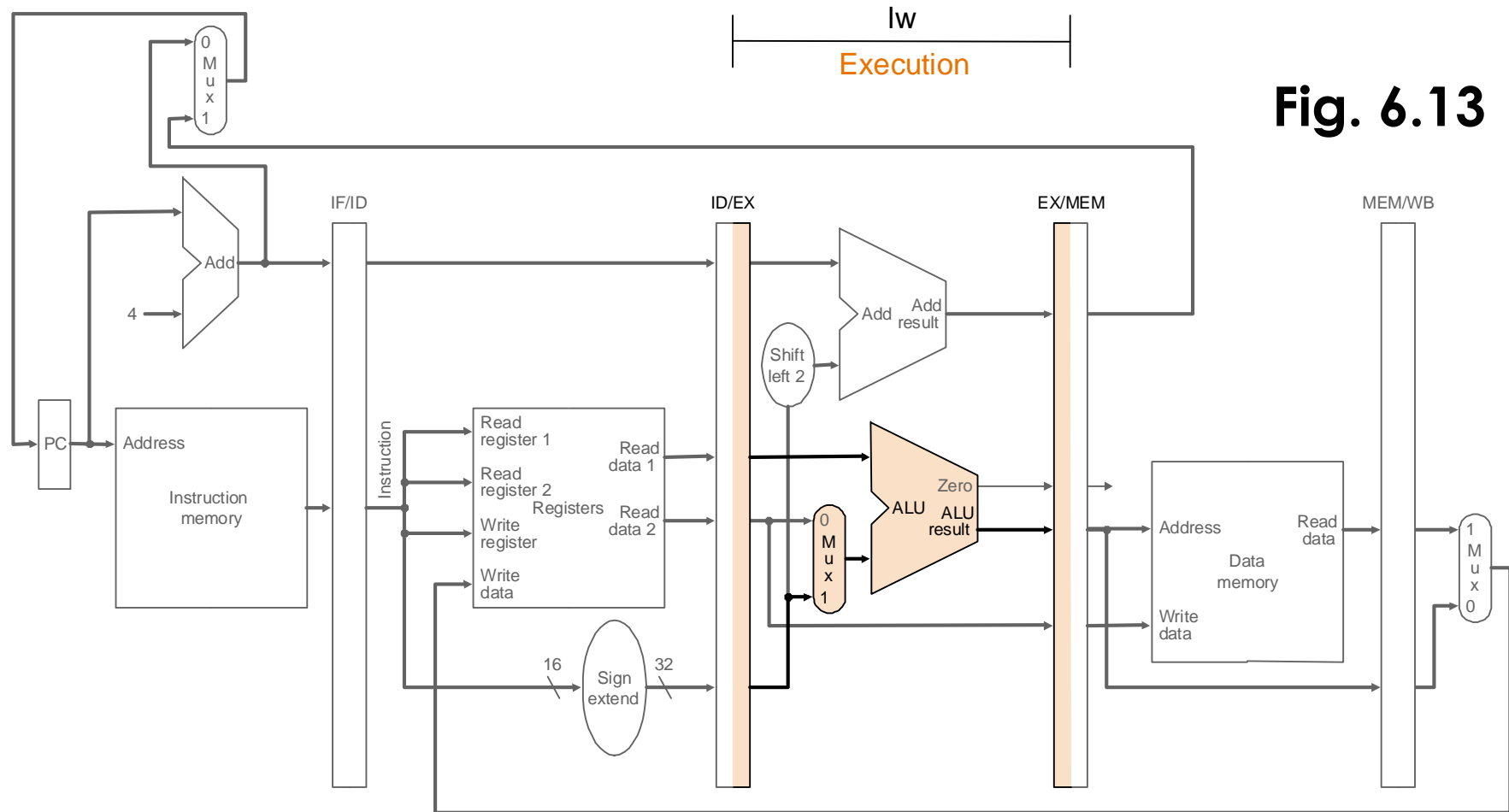


Fig. 6.13

MEM State of load

- MDR = mem[ALUout]

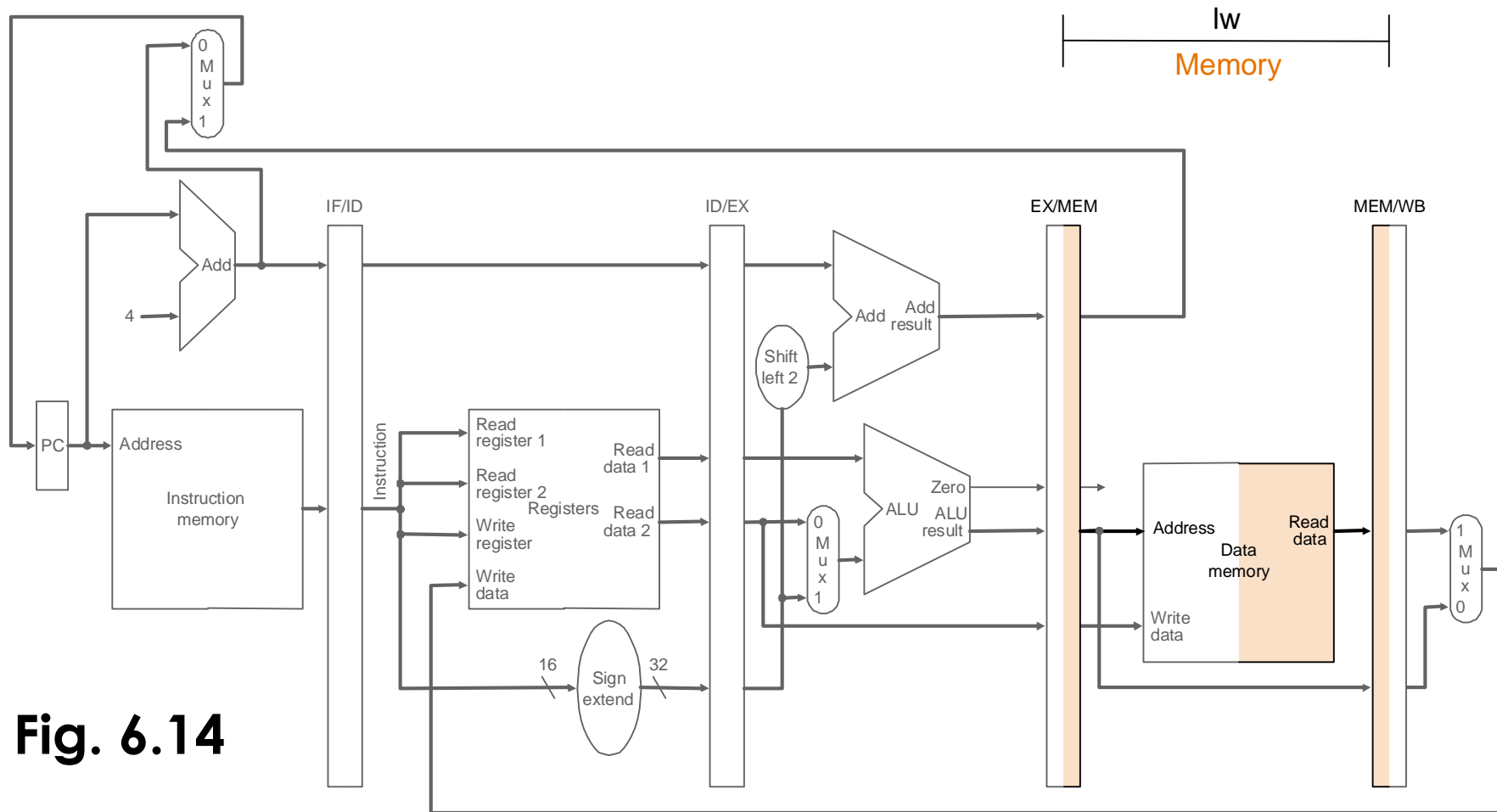


Fig. 6.14

WB Stage of load

- $\text{Reg}[\text{IR}[20-16]] = \text{MDR}$

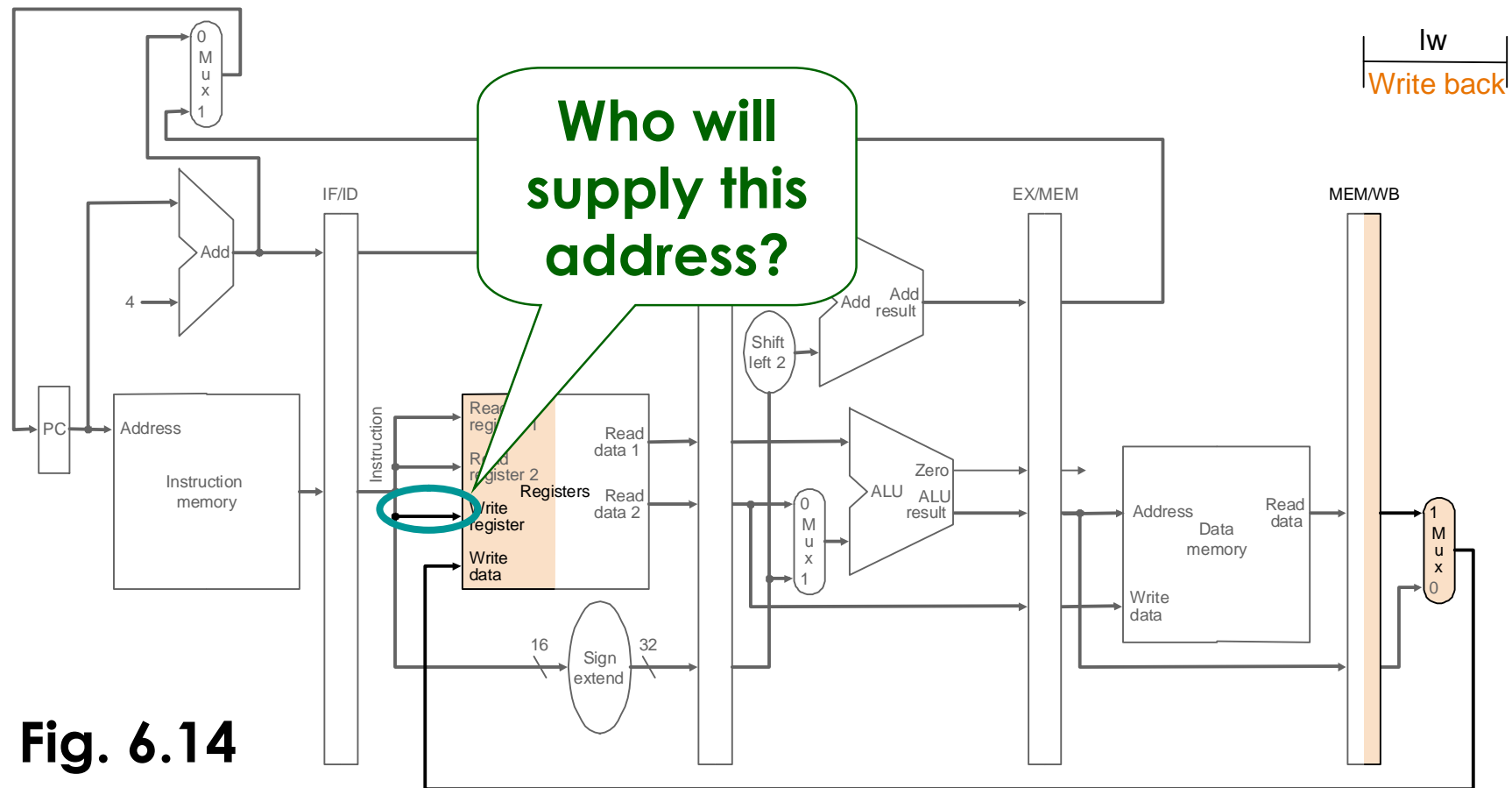
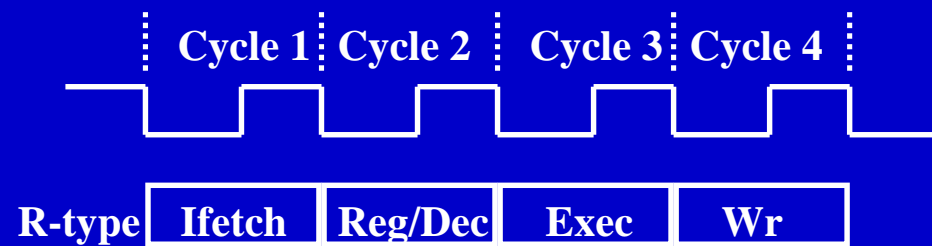


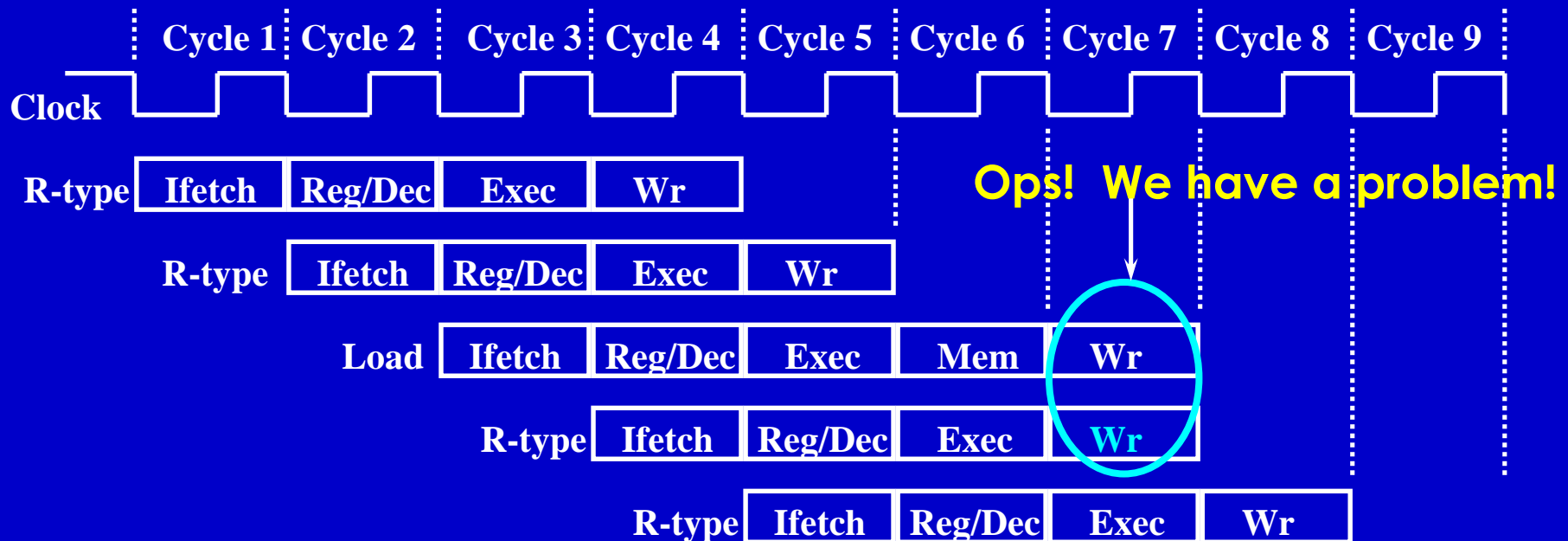
Fig. 6.14

The Four Stages of R-type



- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX: ALU operates on the two register operands
- WB: write ALU output back to the register file

Pipelining R-type and load



- We have a *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



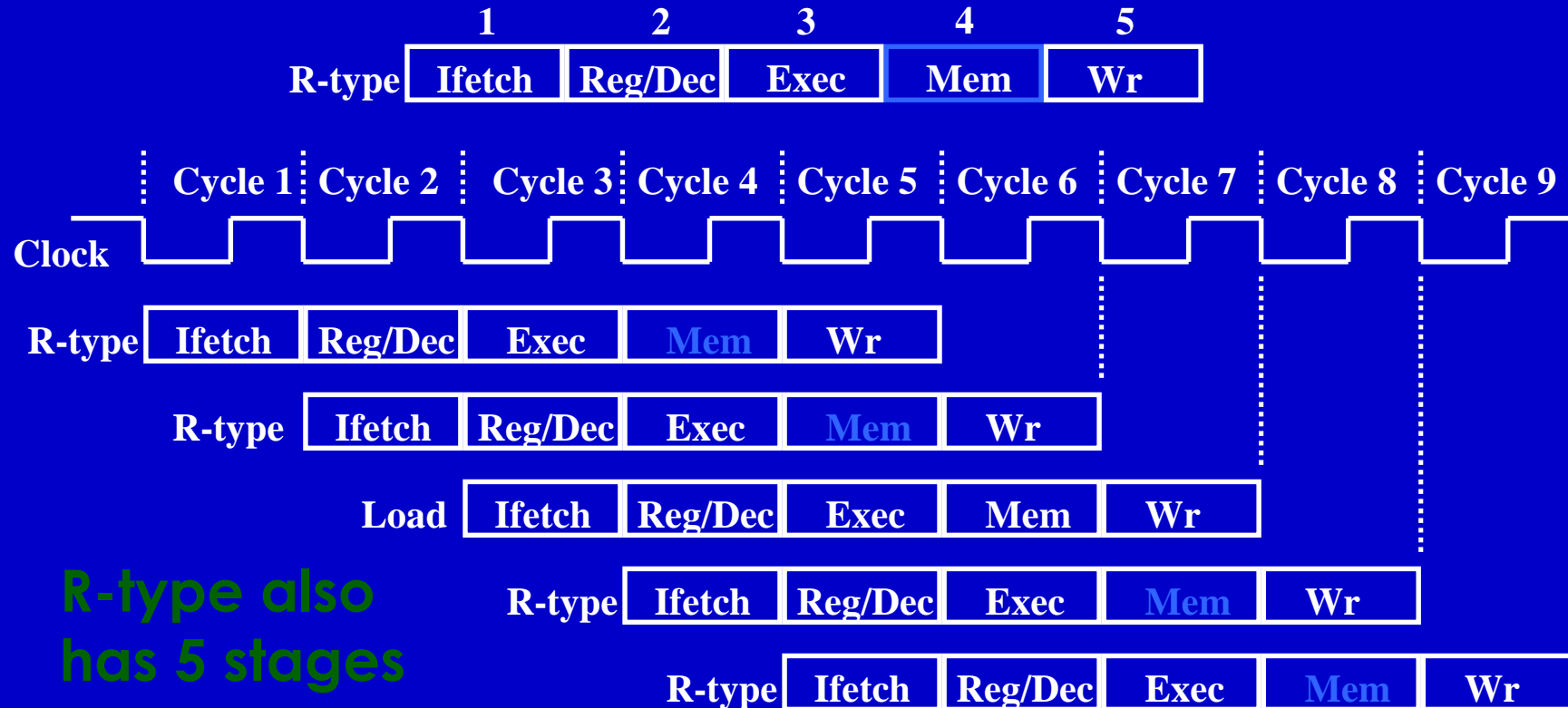
- R-type uses Register File's write port during its **4th** stage



Several ways to solve: forwarding, adding pipeline bubble, making instructions same length

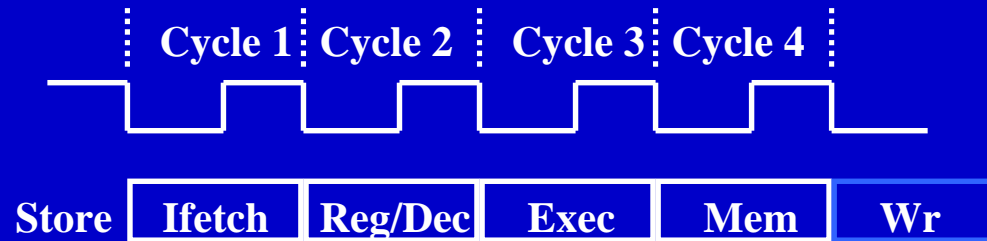
Solution: Delay R-type's Write

- Delay R-type's register write by one cycle:
 - R-type also use Reg File's write port at Stage 5
 - MEM is a NOP stage: **nothing is being done.**



R-type also has 5 stages

The Four Stages of store

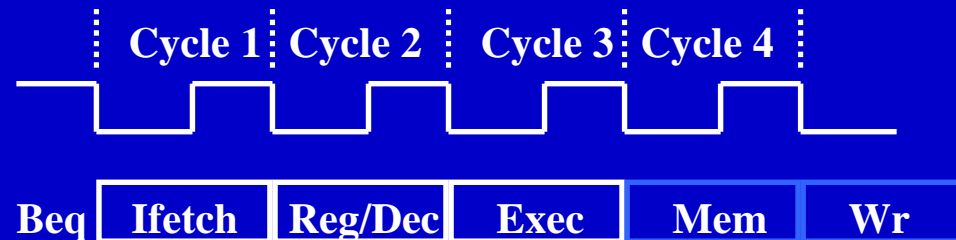


- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX: calculate the memory address
- MEM: write the data into the Data Memory

Add an extra stage:

- WB: NOP

The Three Stages of beq



- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX:
 - compares the two register operand
 - select correct branch target address
 - latch into PC

Add two extra stages:

- MEM: NOP
- WB: NOP

Pipelined Datapath

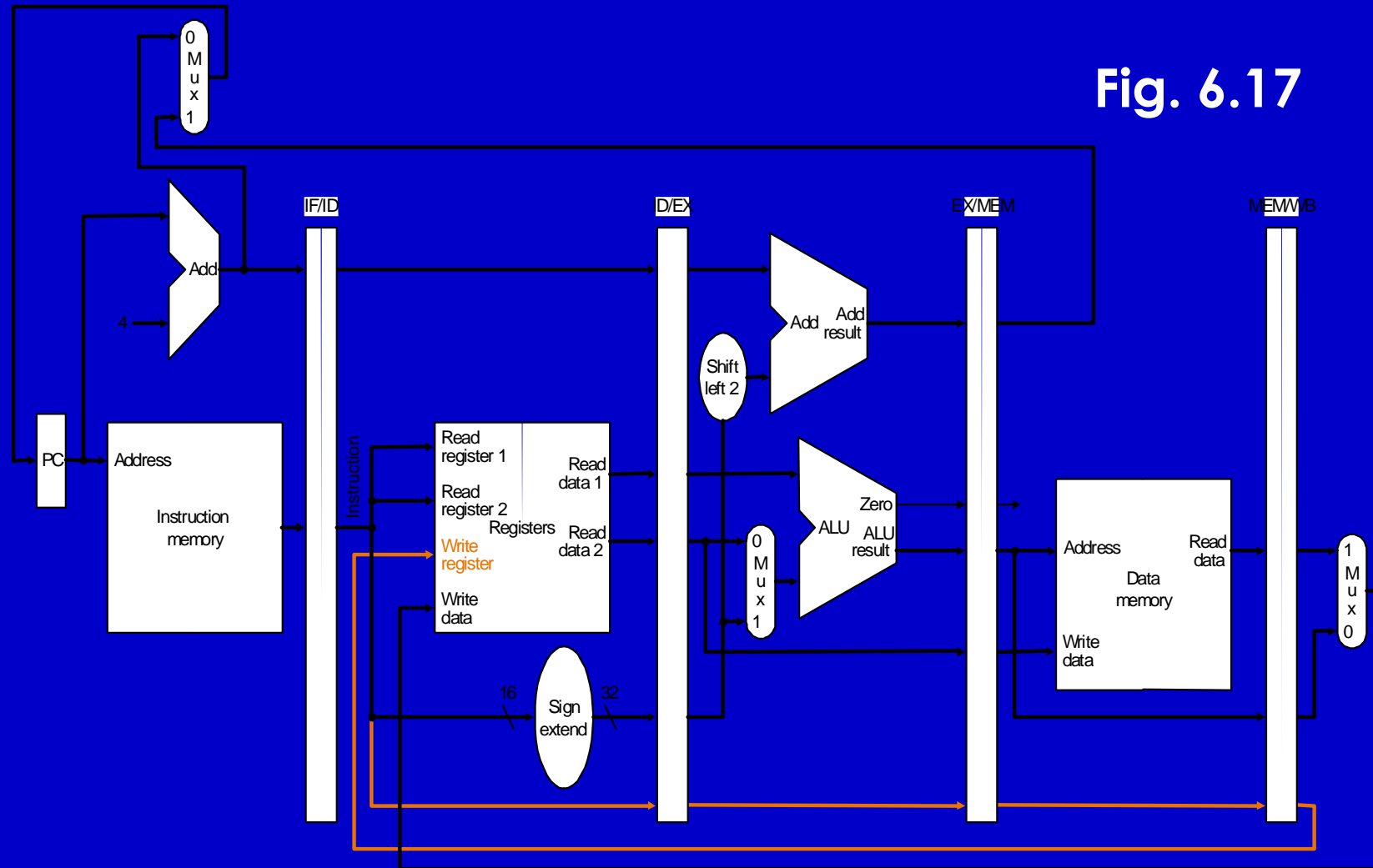
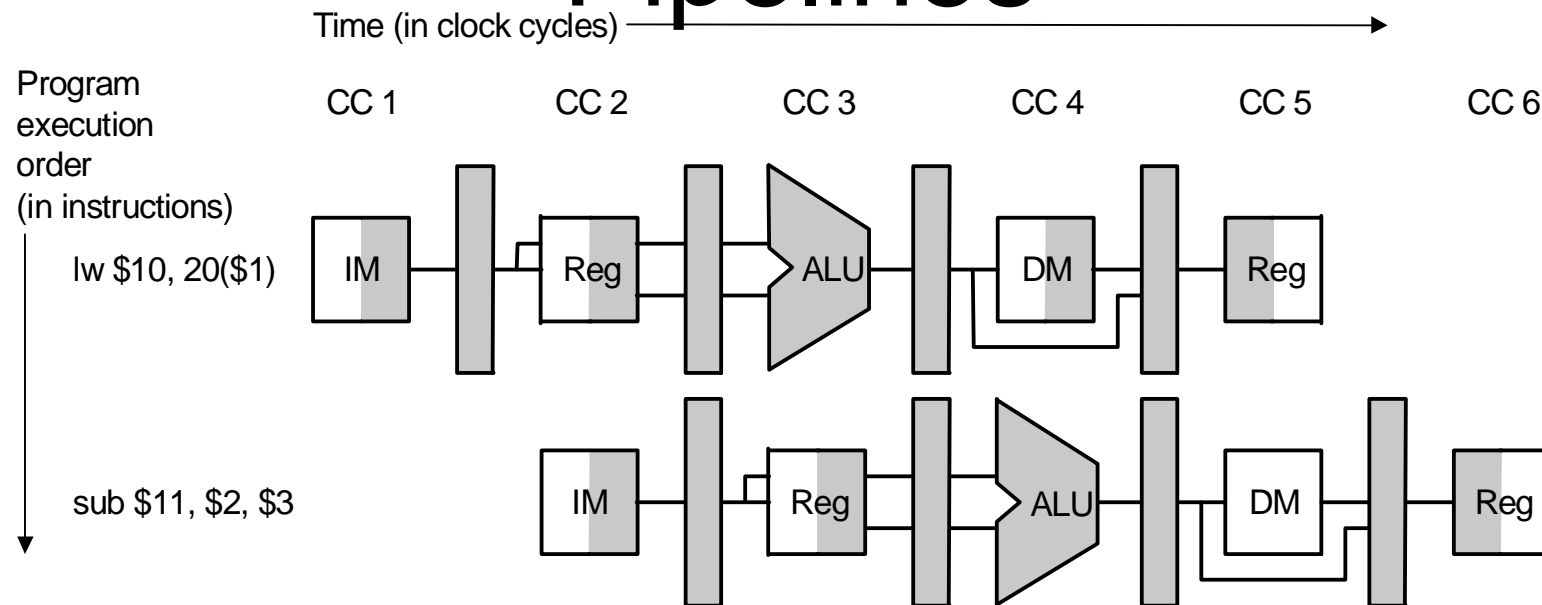


Fig. 6.17

Graphically Representing Pipelines



- Can help with answering questions like:
 - How many cycles to execute this code?
 - What is the ALU doing during cycle 4?
 - Help understand datapaths

Example 1: Cycle 1

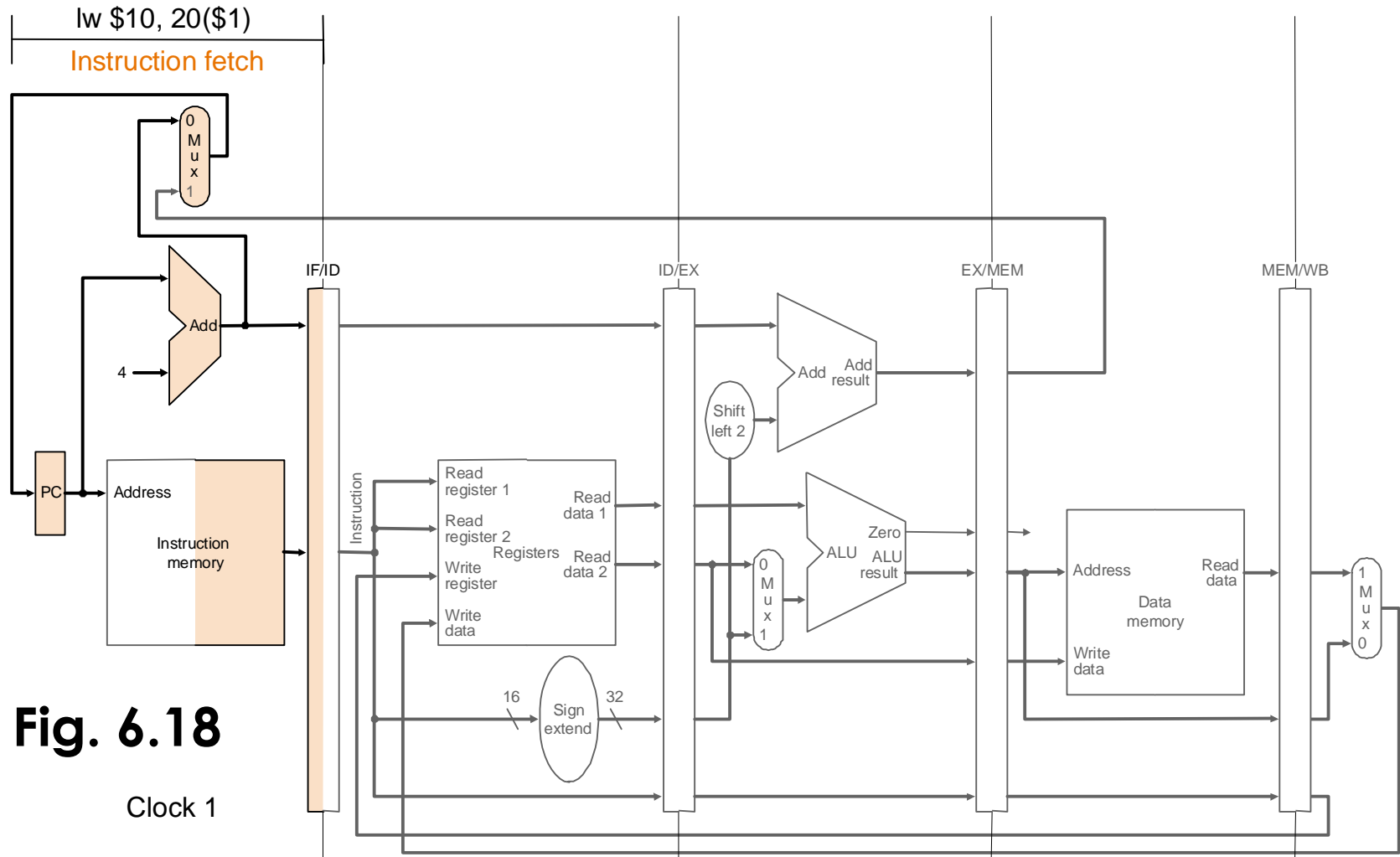


Fig. 6.18

Clock 1

Example 1: Cycle 2

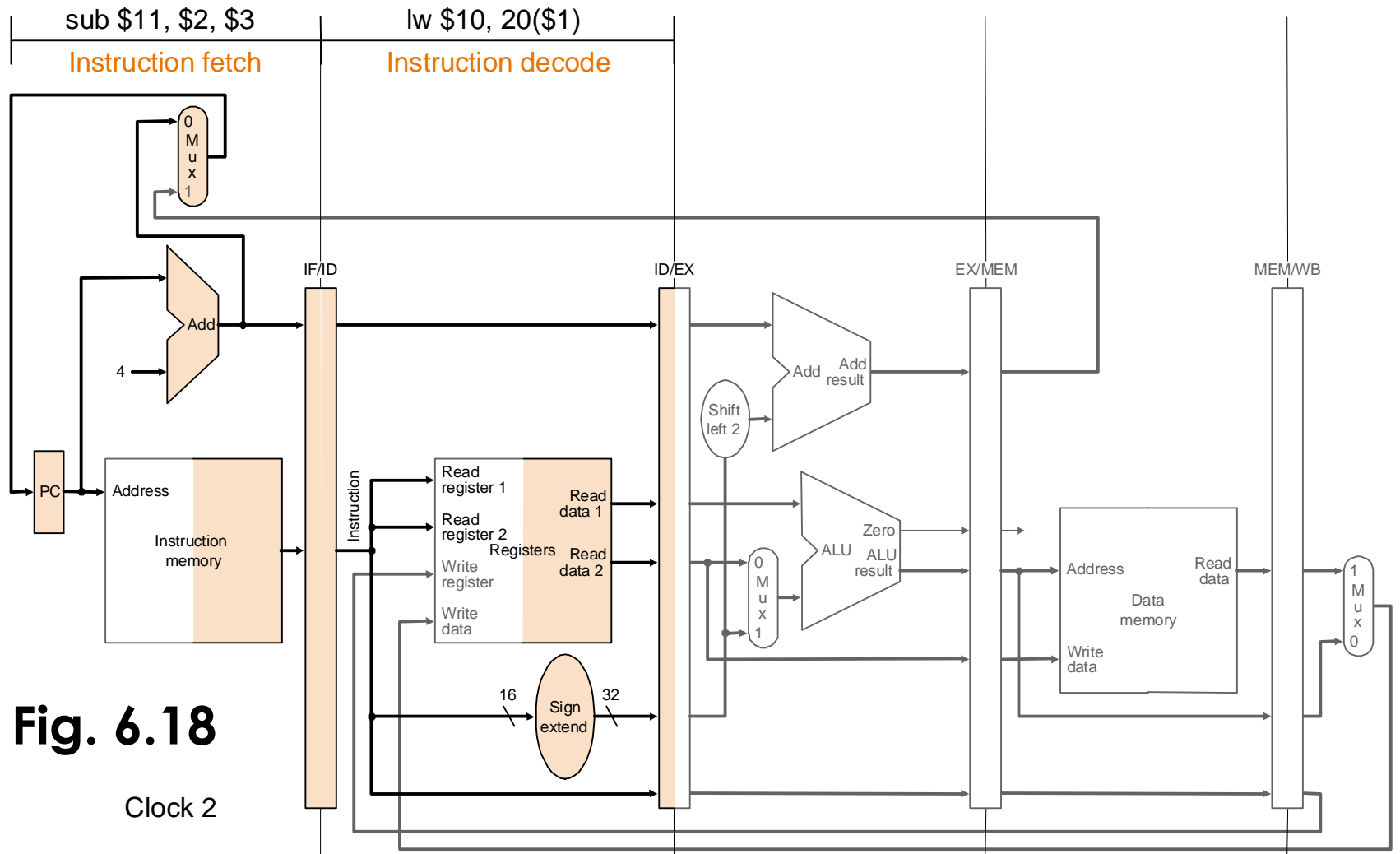


Fig. 6.18

Clock 2

Example 1: Cycle 3

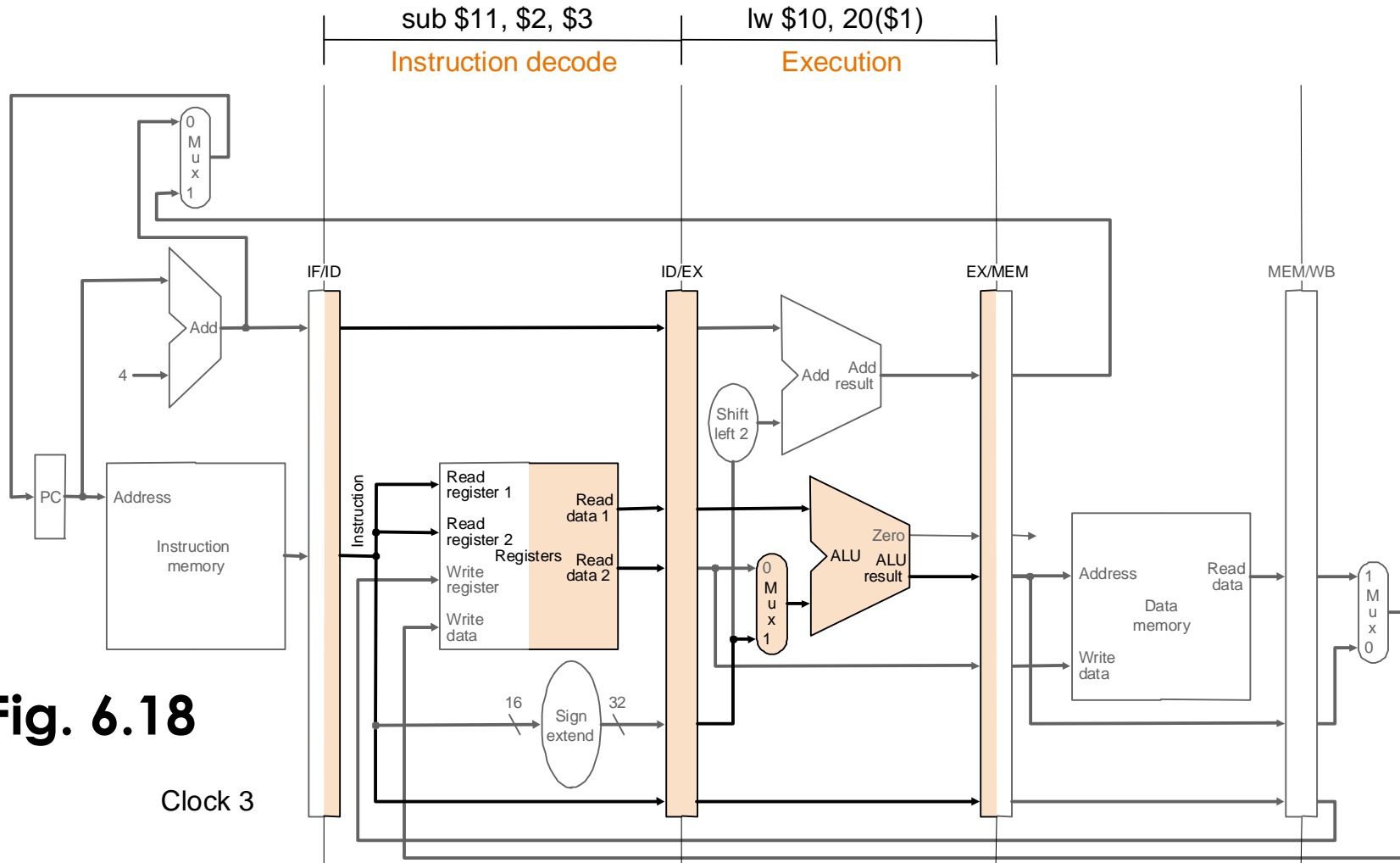
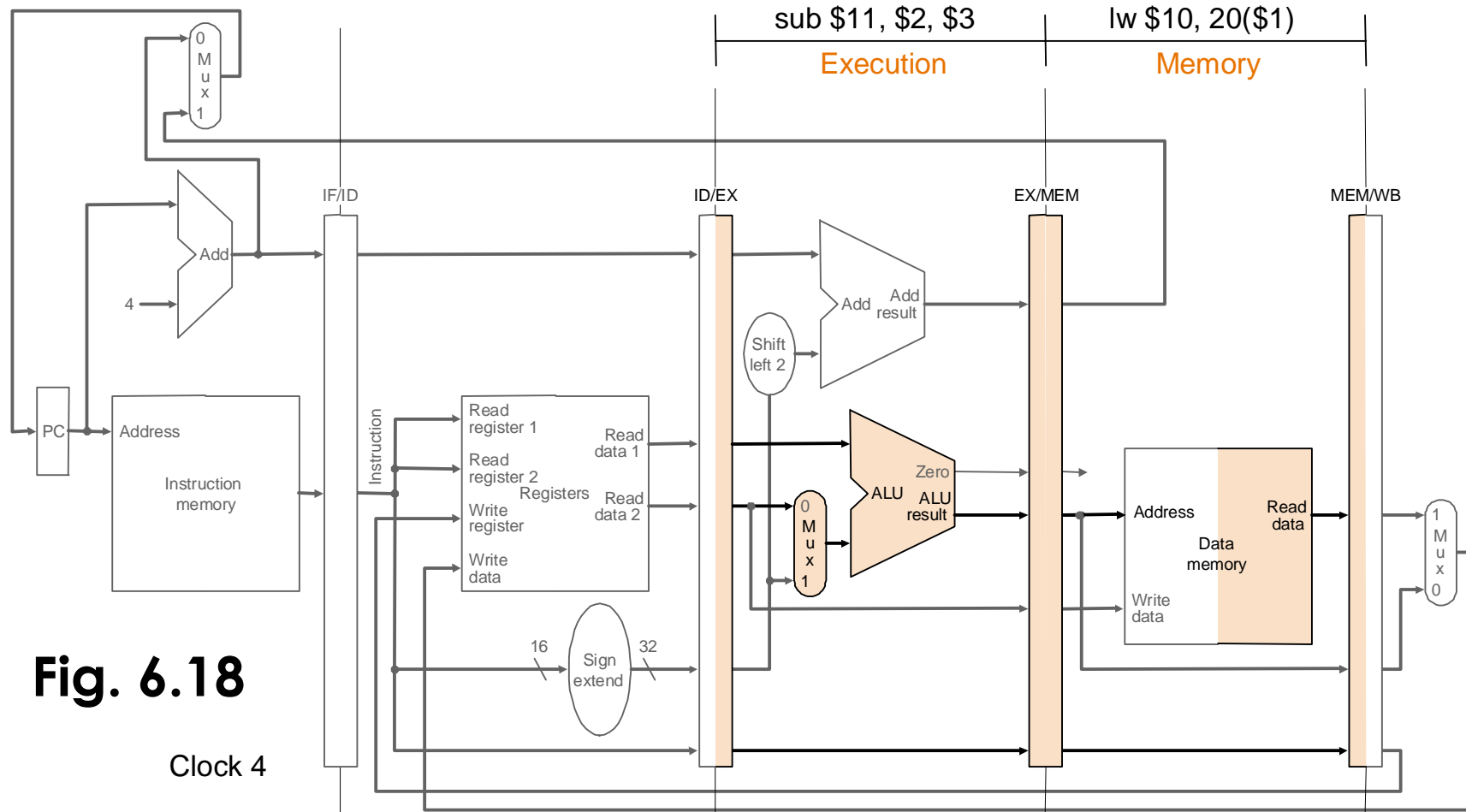


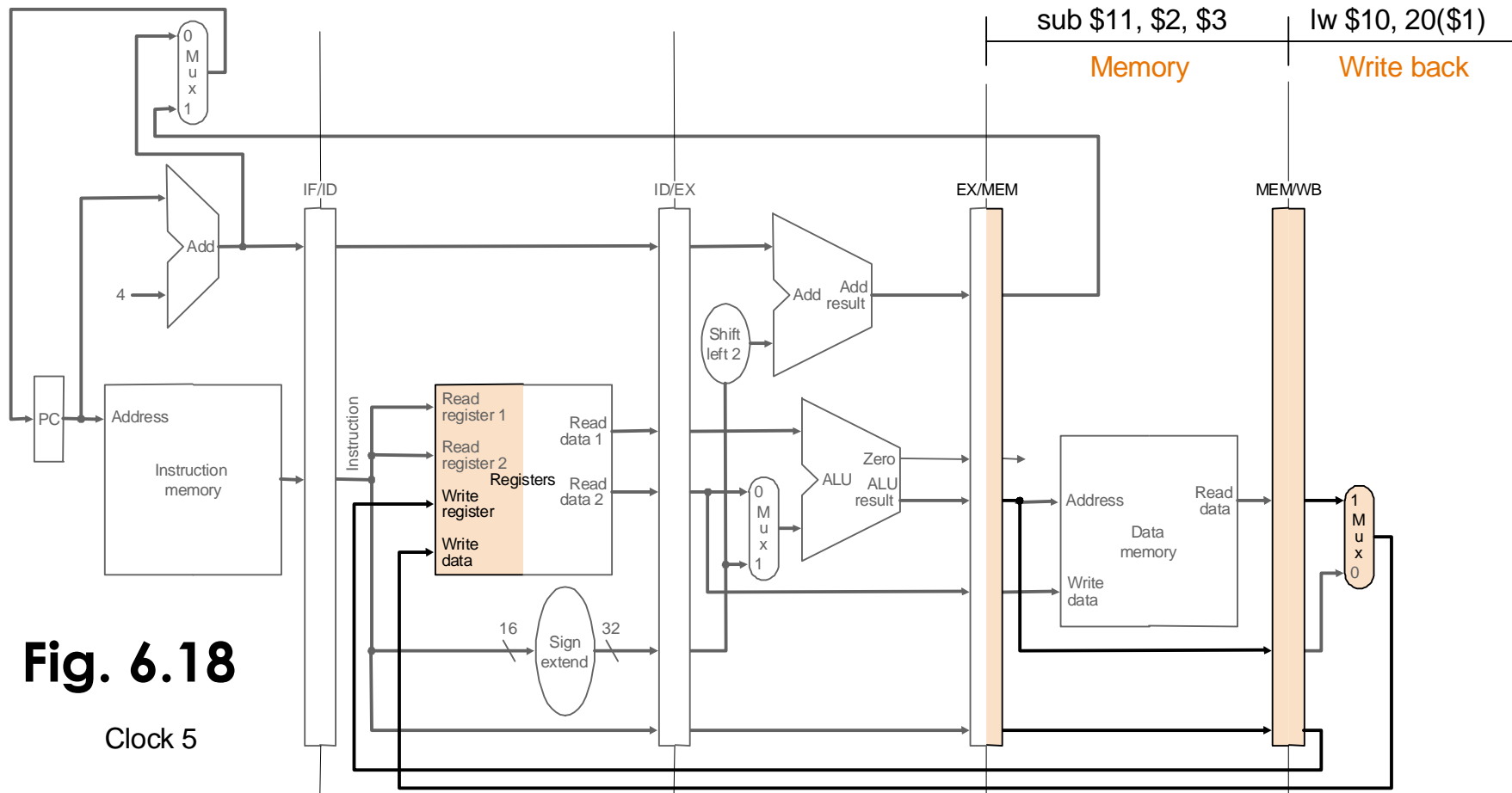
Fig. 6.18

Clock 3

Example 1: Cycle 4



Example 1: Cycle 5



Example 1: Cycle 6

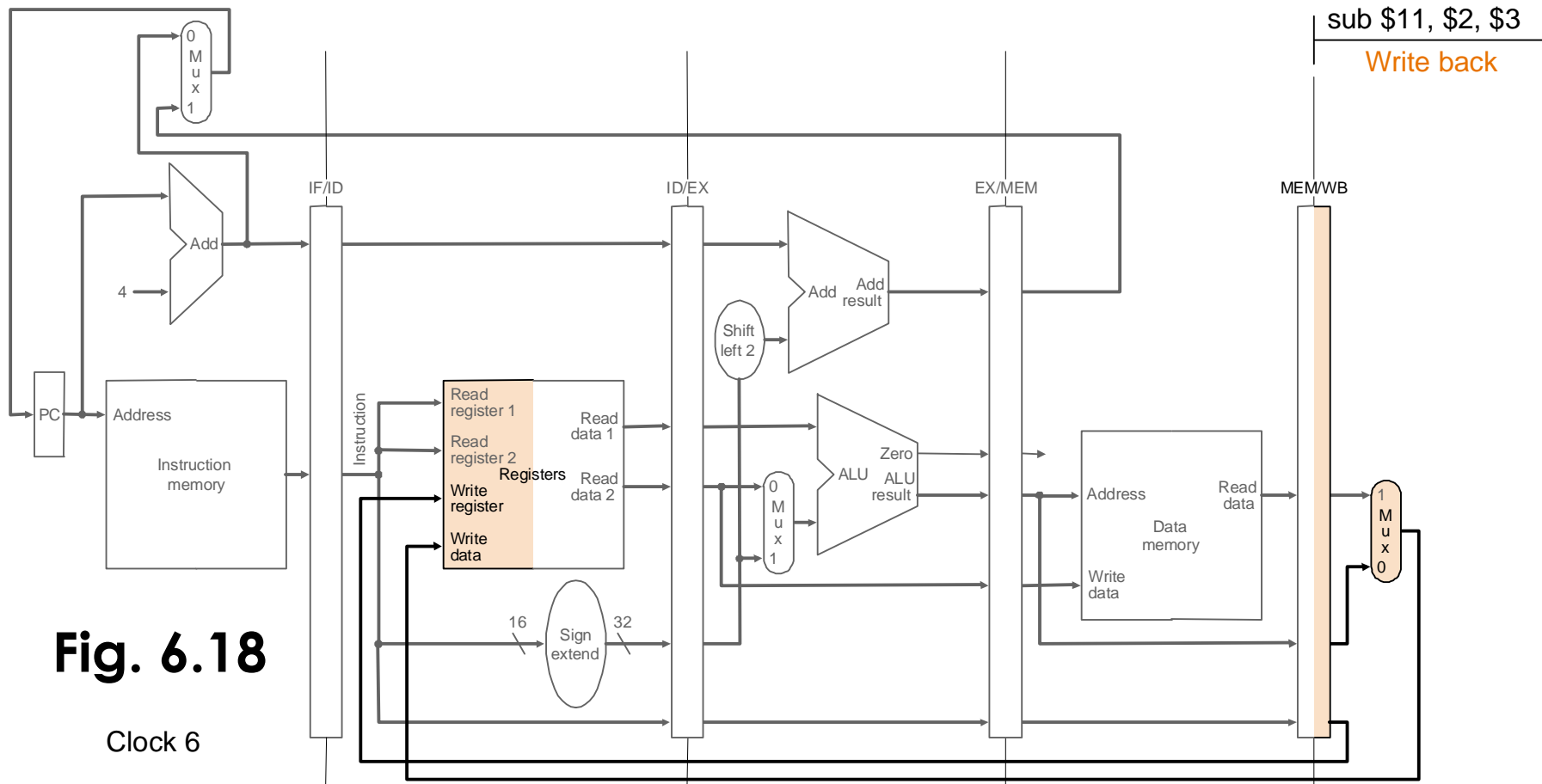


Fig. 6.18

Clock 6

Outline

- An overview of pipelining
- A pipelined datapath
- Pipelined control (6.3)
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards
- Exceptions
- Superscalar and dynamic pipelining

Pipeline Control: Control Signals

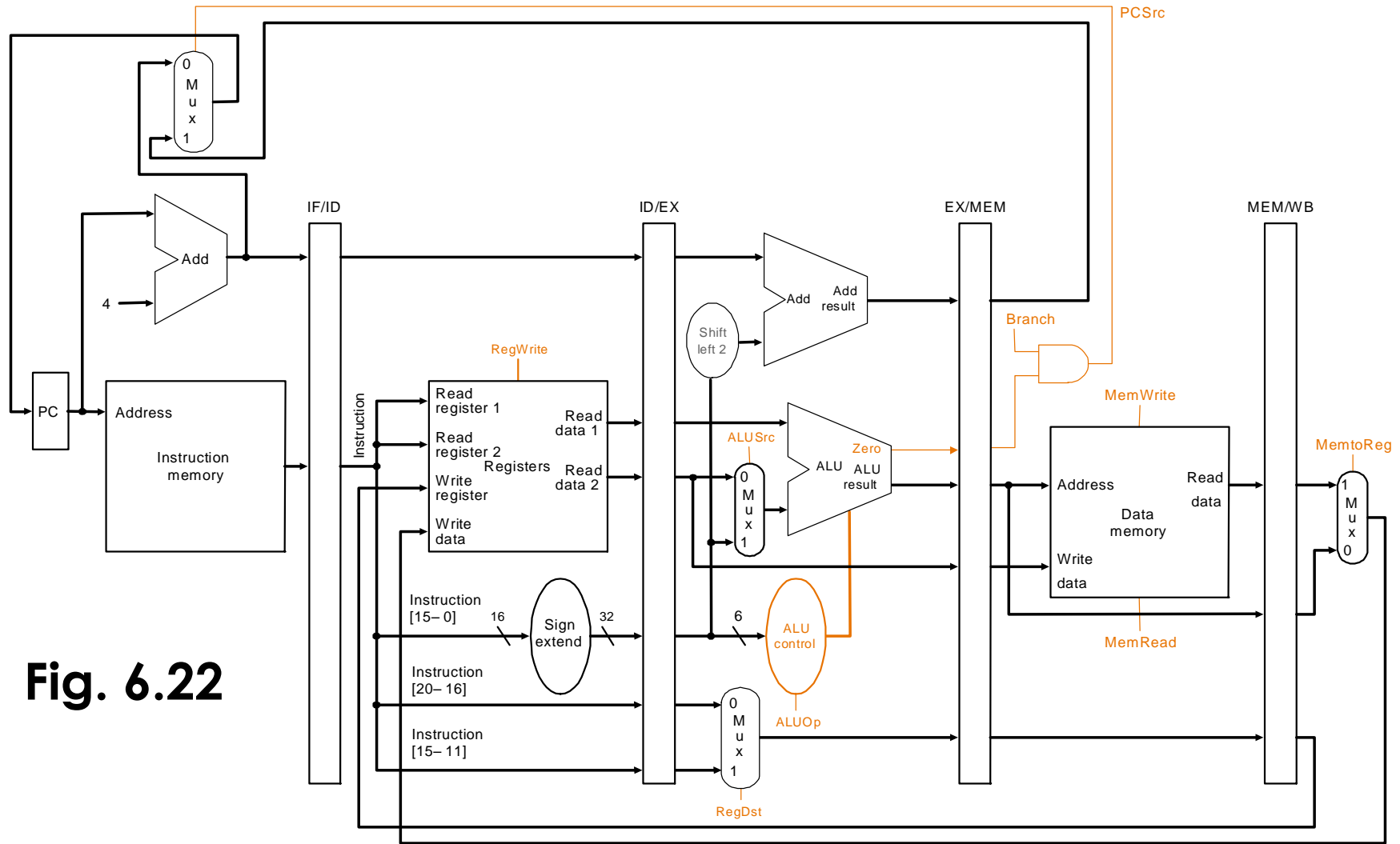


Fig. 6.22

Group Signals According to Stages

- Can use control signals of single-cycle CPU (Fig. 6.23, 6.24 \Leftrightarrow 5.12, 5.16)

	Execution/Address Calculation stage control lines			Memory access stage control lines			Write-back stage control lines		
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-type	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Fig. 6.25

Data Stationary Control

- Pass control signals along just like the data
 - Main control generates control signals during ID

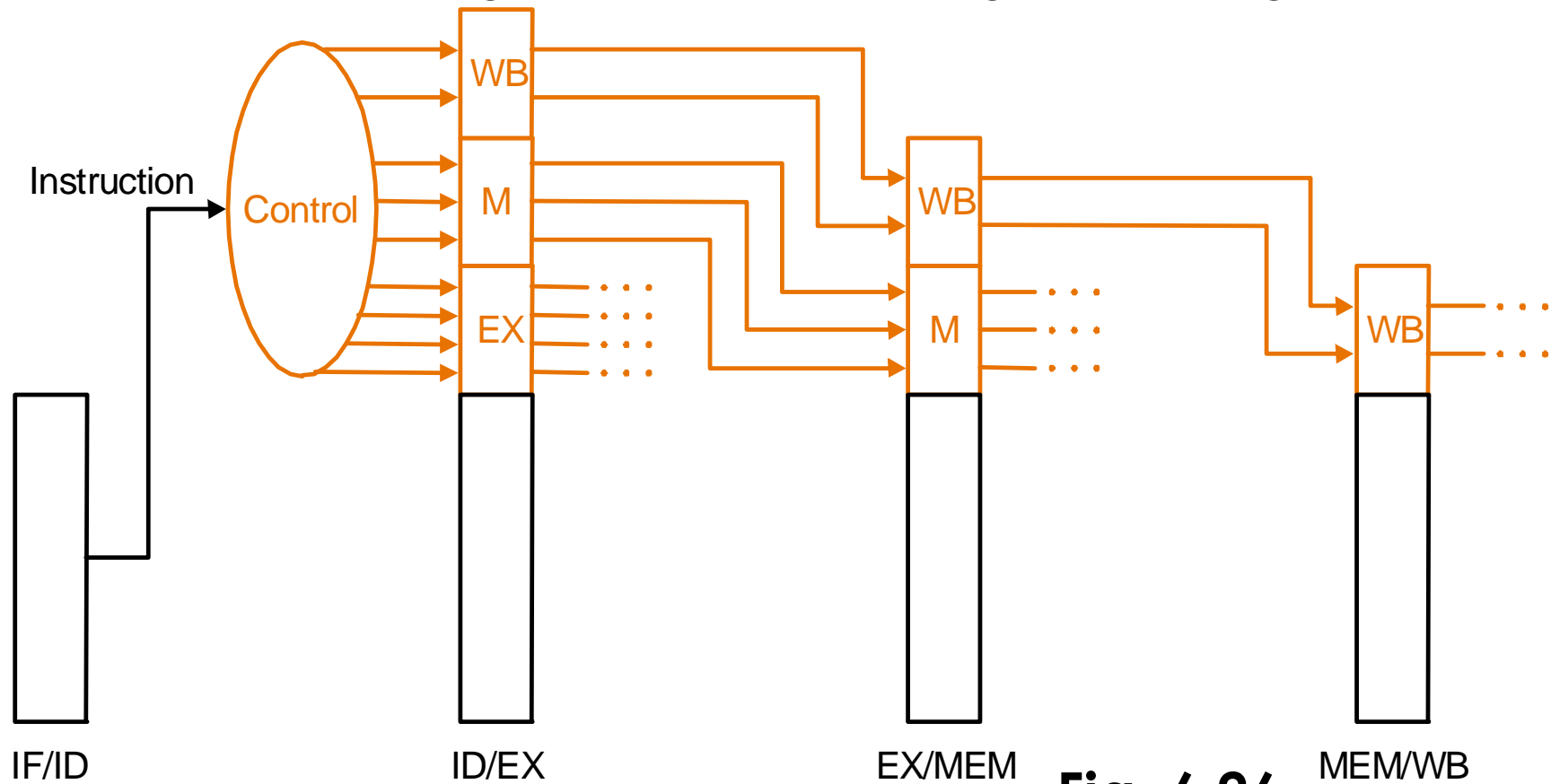
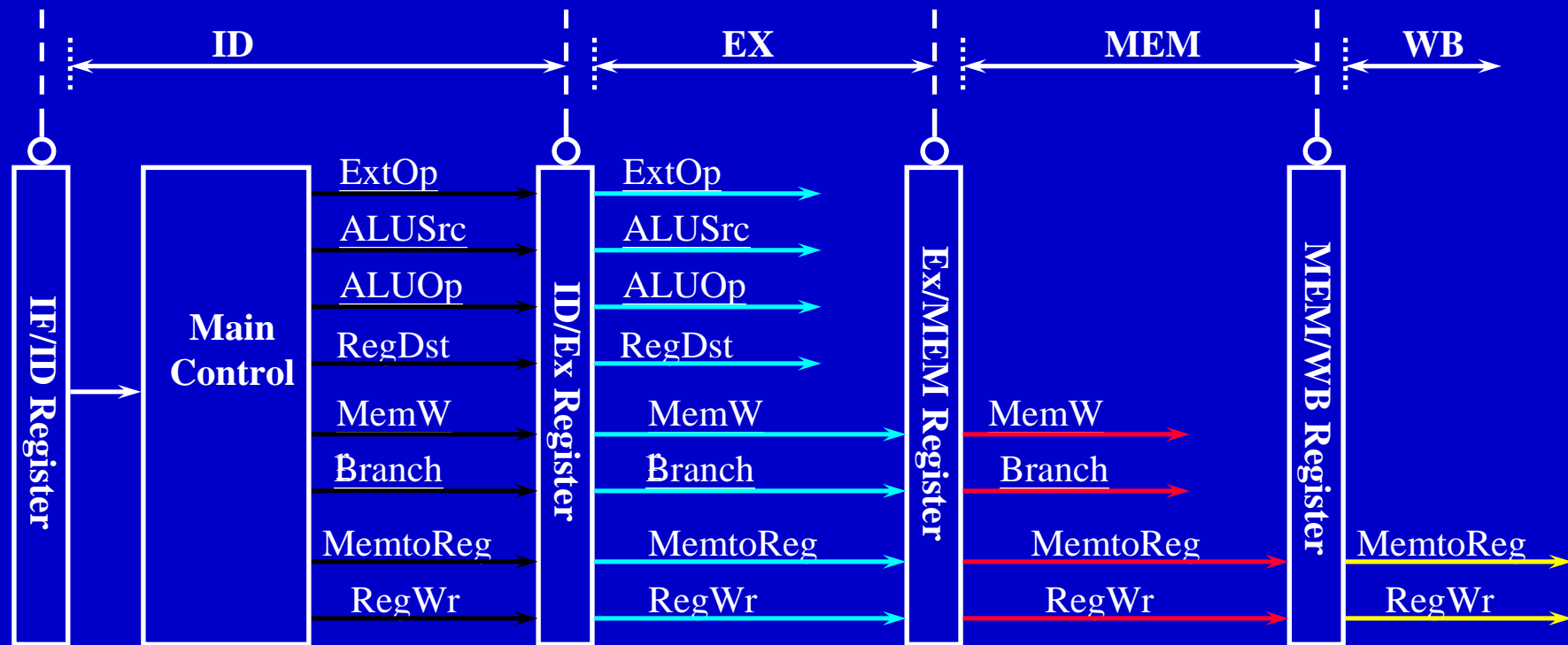


Fig. 6.26

Data Stationary Control (cont.)

- Signals for EX (ExtOp, ALUSrc, ...) are used 1 cycle later
- Signals for MEM (MemWr, Branch) are used 2 cycles later
- Signals for WB (MemtoReg, MemWr) are used 3 cycles later



WB Stage of load

- $\text{Reg}[\text{IR}[20-16]] = \text{MDR}$

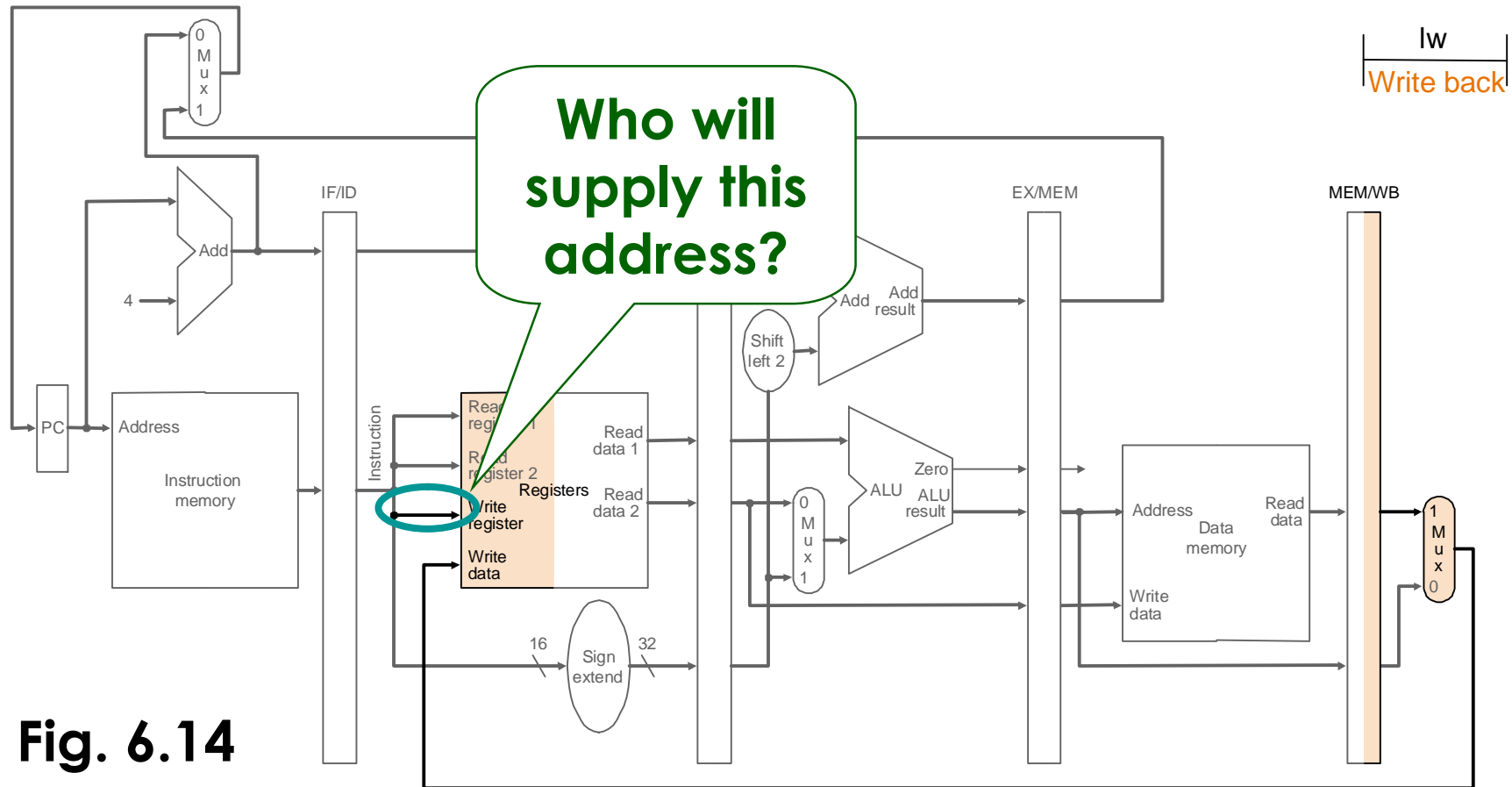


Fig. 6.14

Datapath with Control

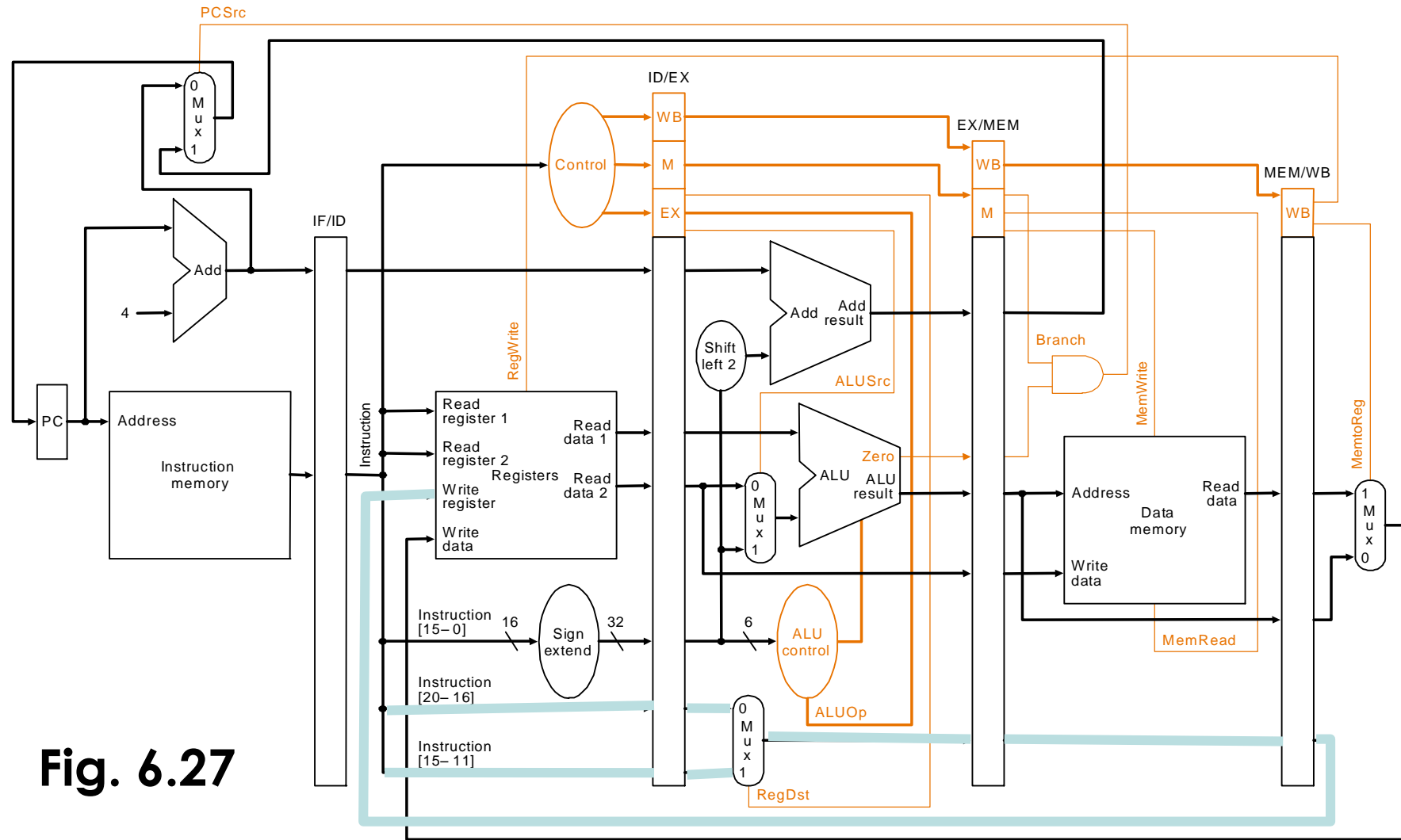


Fig. 6.27

Let's Try it Out

lw \$10, 20(\$1)

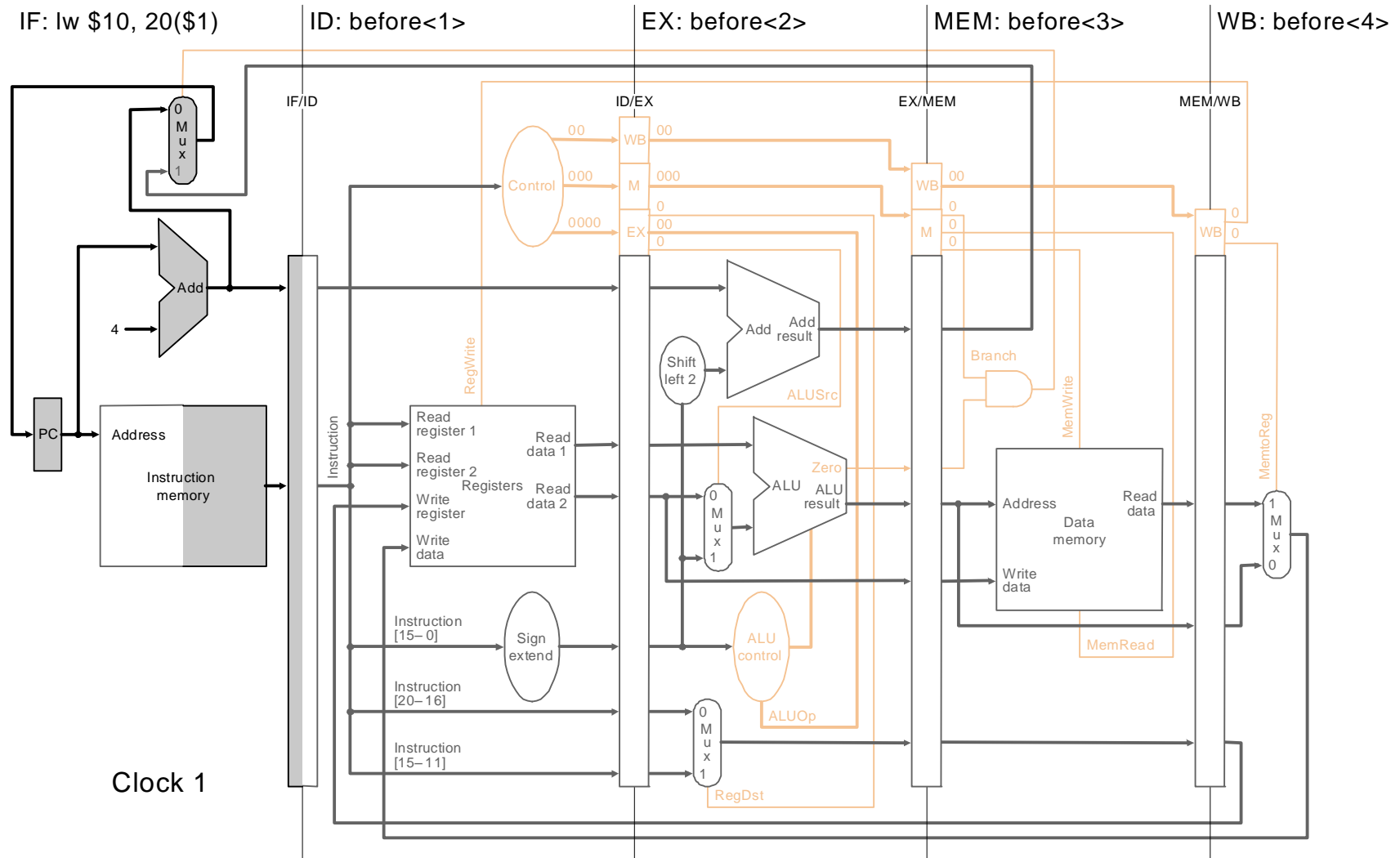
sub \$11, \$2, \$3

and \$12, \$4, \$5

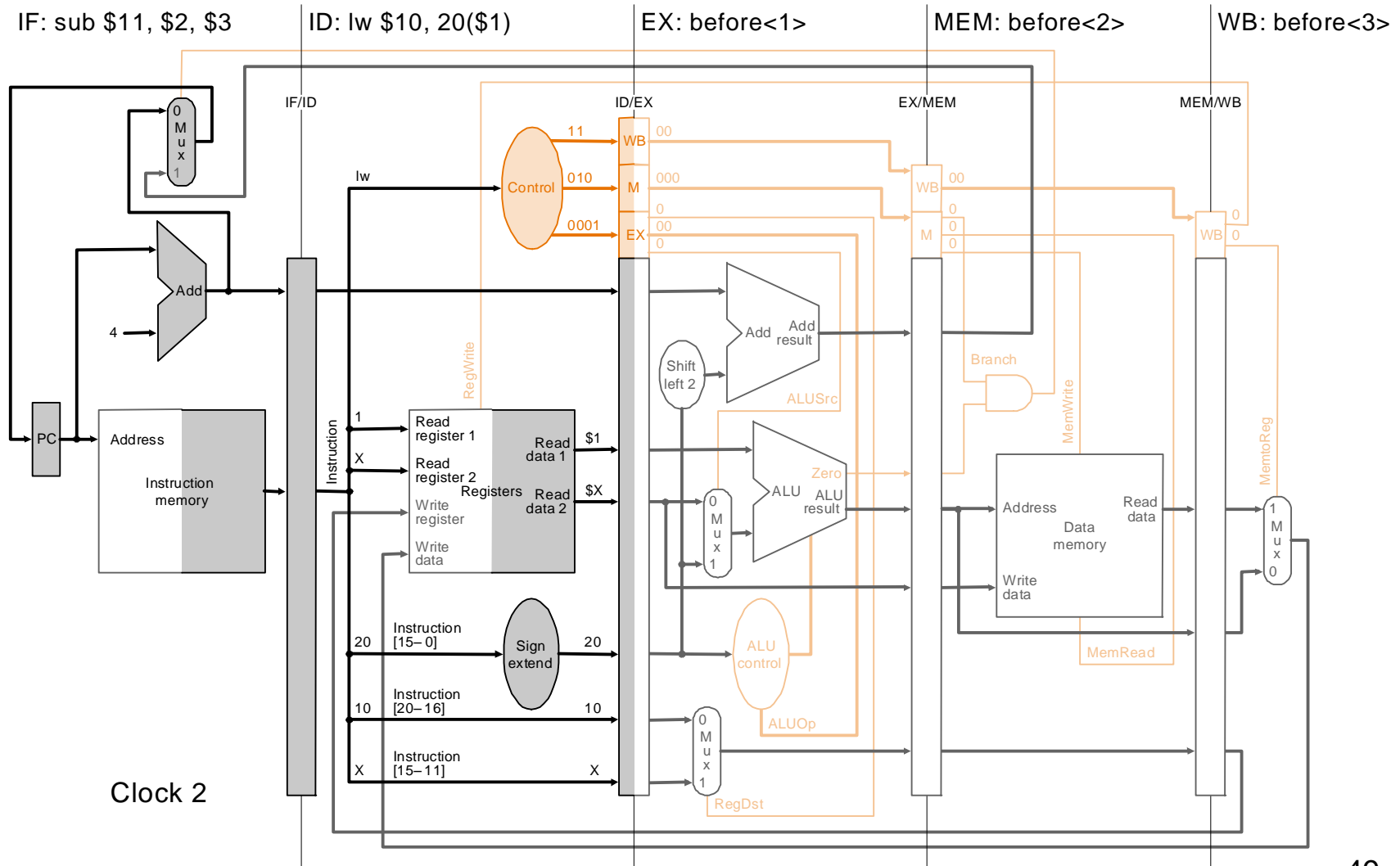
or \$13, \$6, \$7

add \$14, \$8, \$9

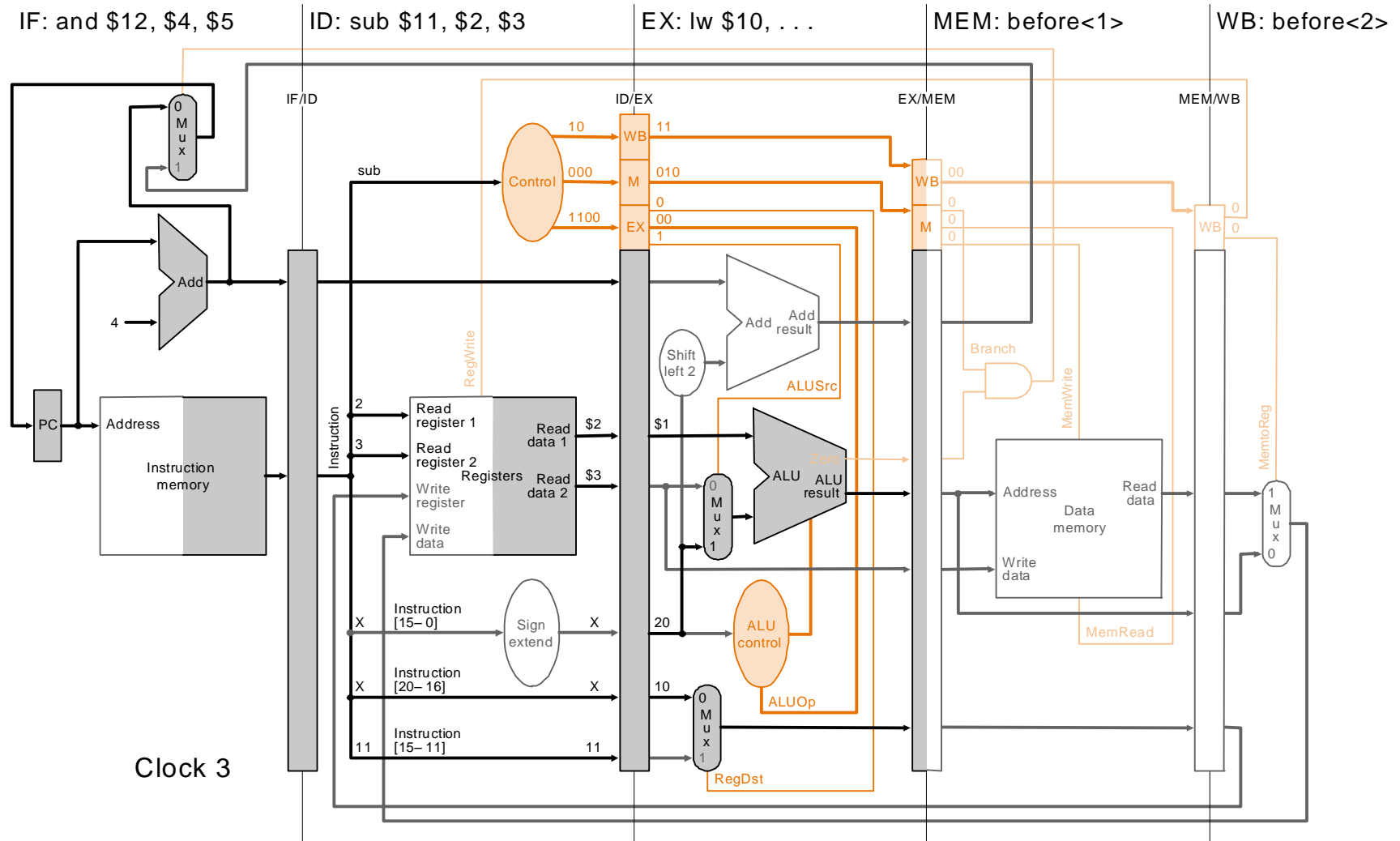
Example 2: Cycle 1



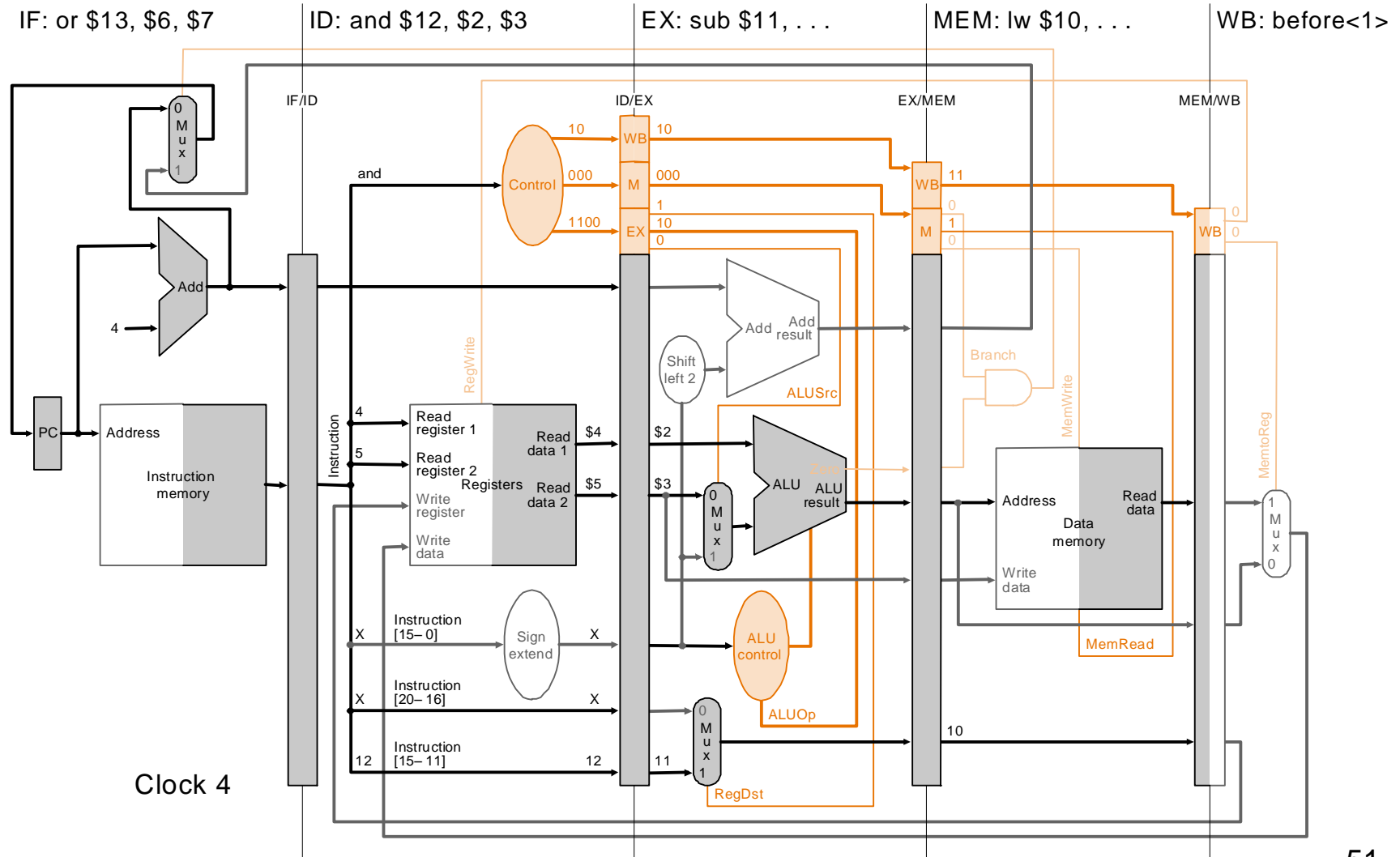
Example 2: Cycle 2



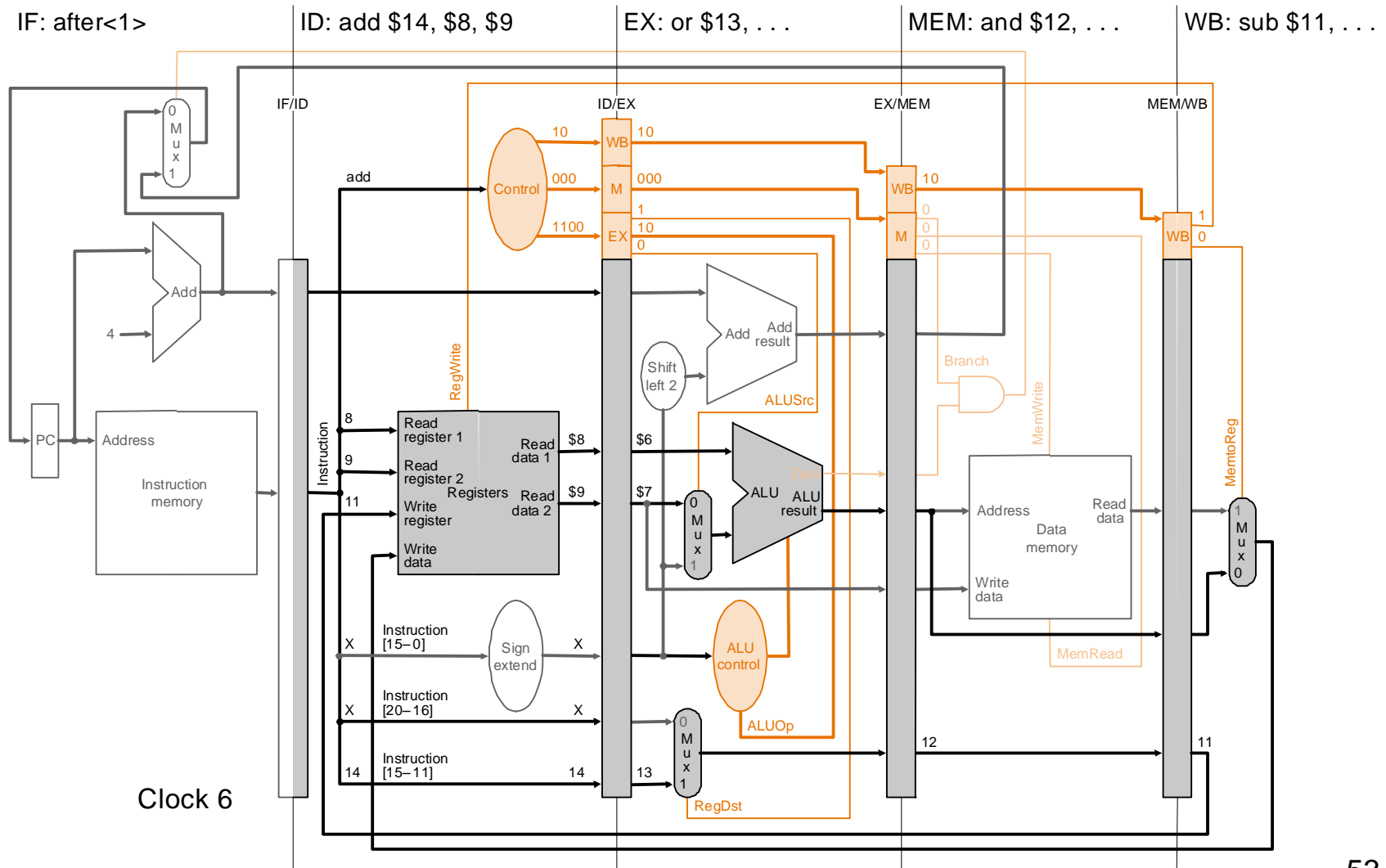
Example 2: Cycle 3



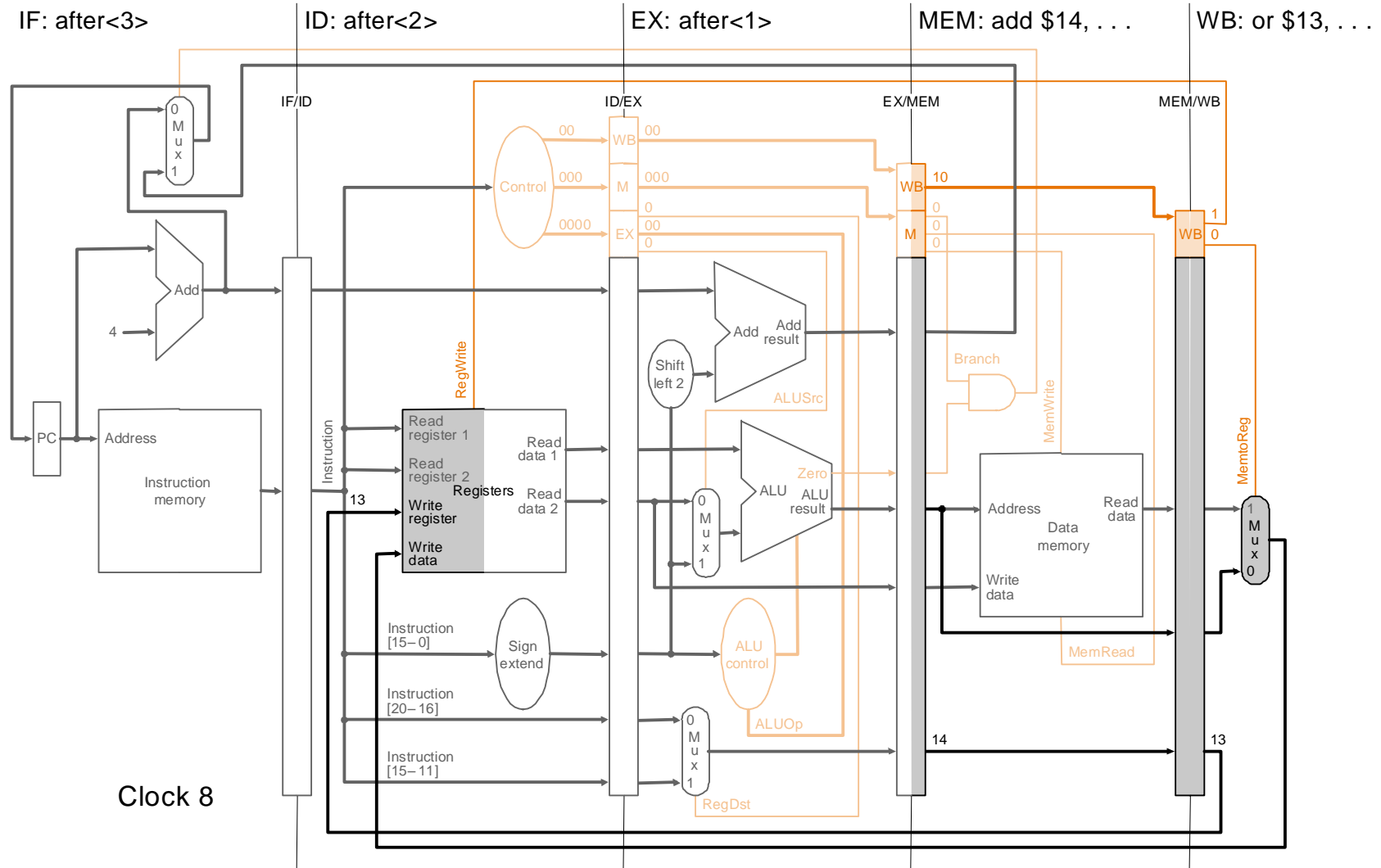
Example 2: Cycle 4



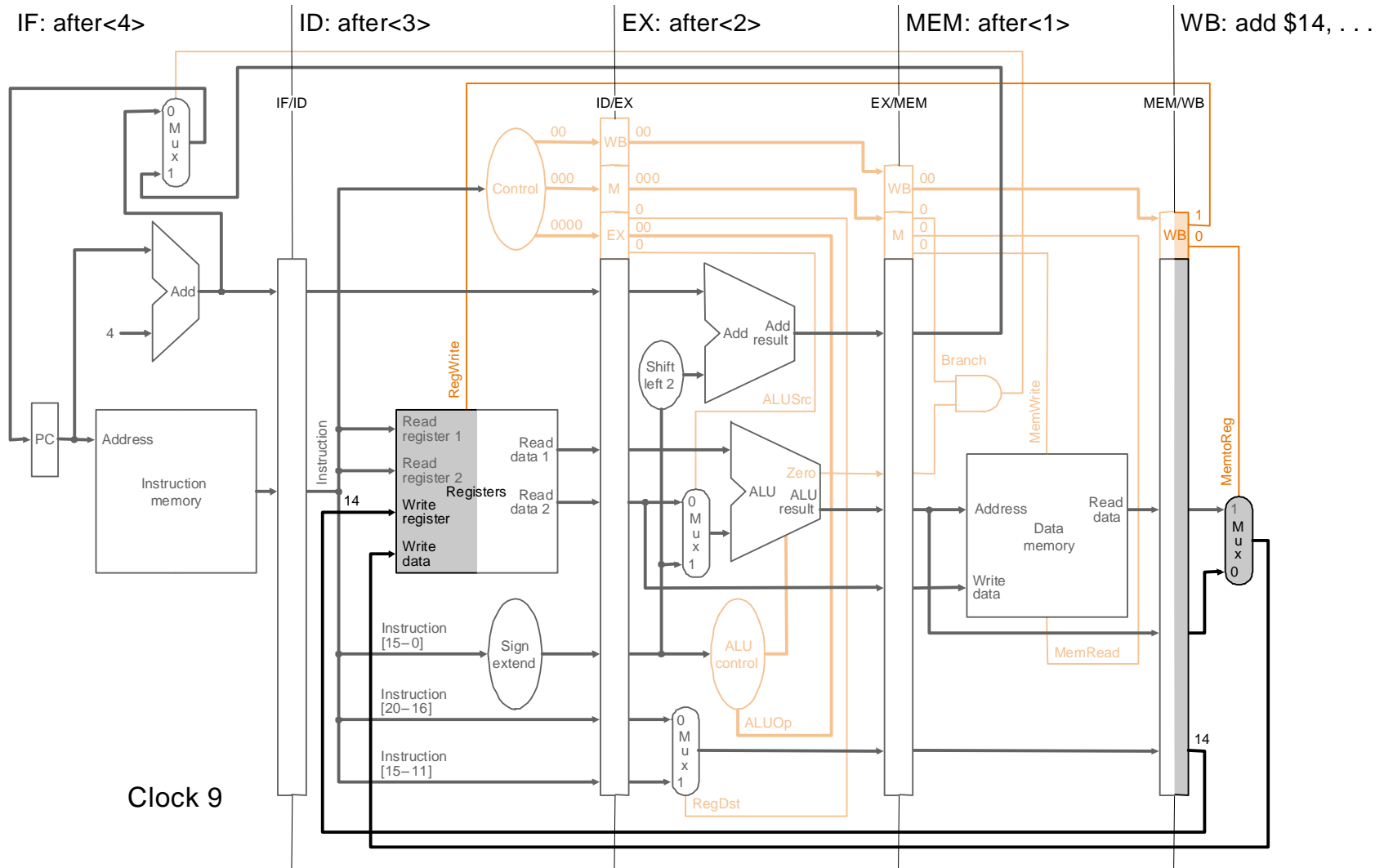
Example 2: Cycle 6



Example 2: Cycle 8



Example 2: Cycle 9



Summary of Pipeline Basics

- Pipelining is a fundamental concept
 - Multiple steps using distinct resources
 - Utilize capabilities of datapath by pipelined instruction processing
 - Start next instruction while working on the current one
 - Limited by length of longest stage (plus fill/flush)
 - Need to detect and resolve hazards
- What makes it easy in MIPS?
 - All instructions are of the same length
 - Just a few instruction formats
 - Memory operands only in loads and stores
- What makes pipelining hard? hazards

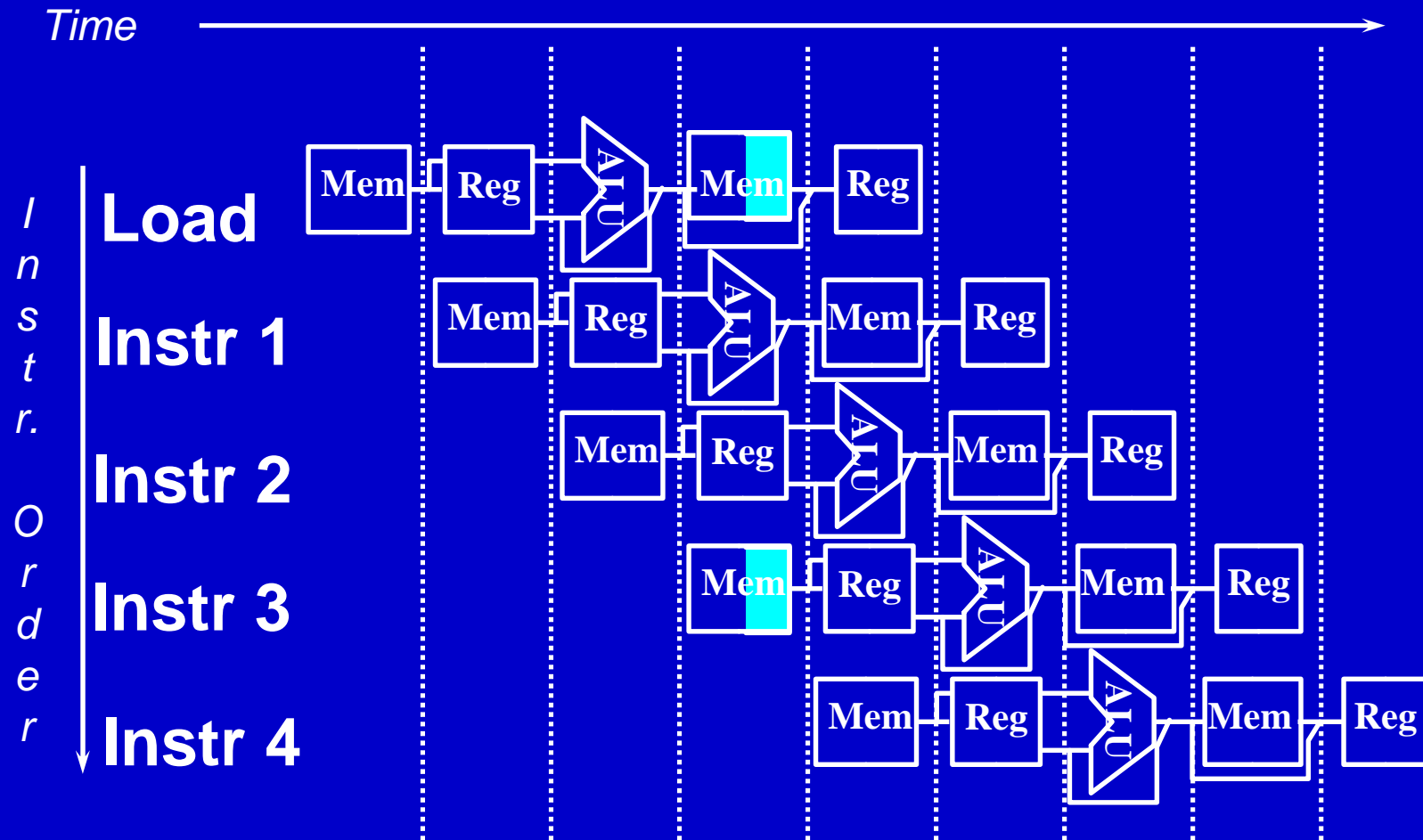
Outline

- An overview of pipelining
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding (6.4)
- Data hazards and stalls (6.5)
- Branch hazards
- Exceptions
- Superscalar and dynamic pipelining

Pipeline Hazards

- Pipeline Hazards:
 - **Structural hazards**: attempt to use the same resource in two different ways at the same time
 - Ex.: combined washer/dryer or folder busy doing something else (watching TV)
 - **Data hazards**: attempt to use item before ready
 - Instruction depends on result of prior instruction still in the pipeline
 - **Control hazards**: attempt to make decision before condition is evaluated
 - Ex.: wash football uniforms and need to see result of previous load to get proper detergent level
 - Branch instructions
- Can always resolve hazards by **waiting**
 - pipeline control must detect the hazard
 - take action (or delay action) to resolve hazards

Structural Hazard: Single Memory



Use 2 memory: data memory and instruction memory

Data Hazards

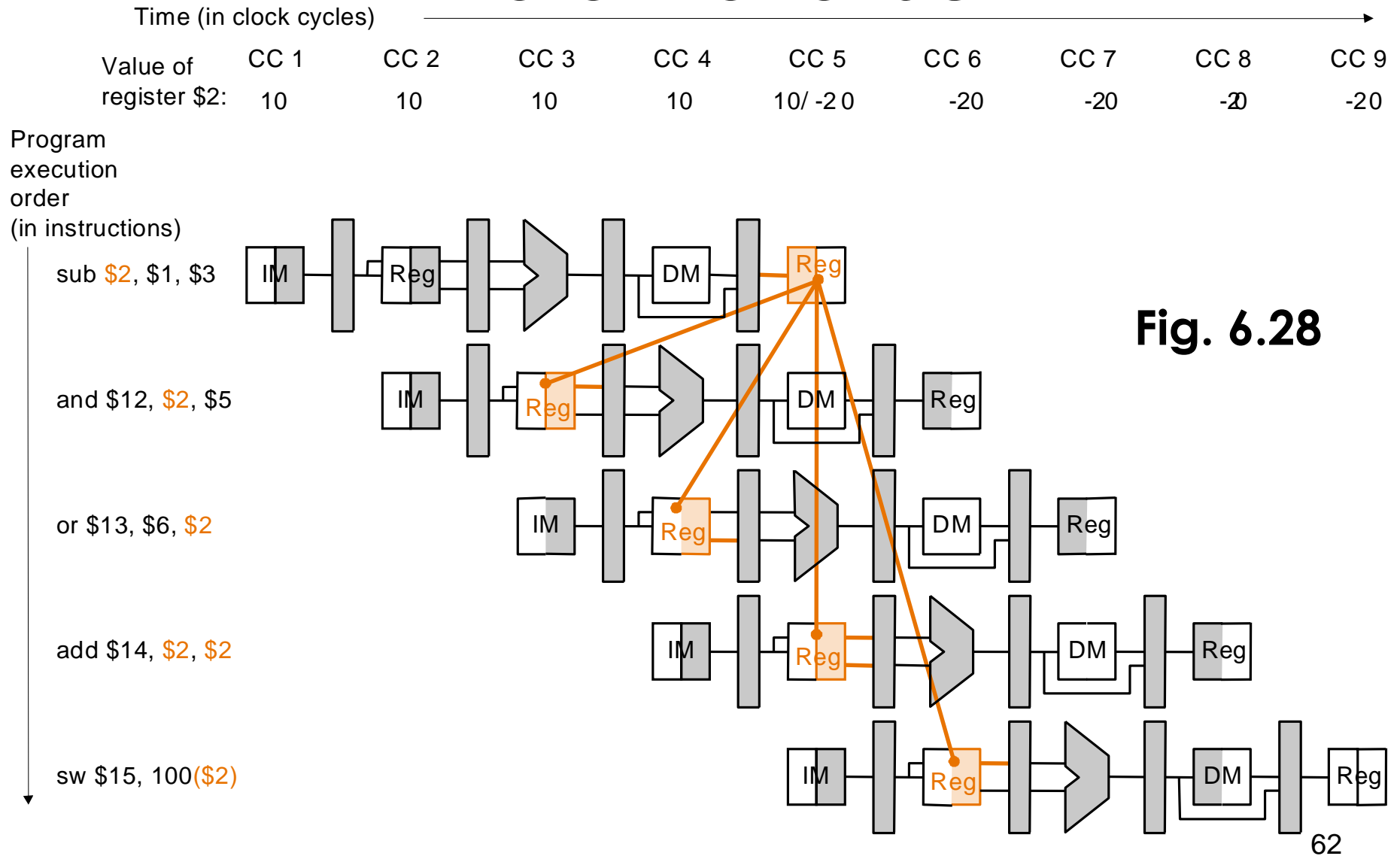


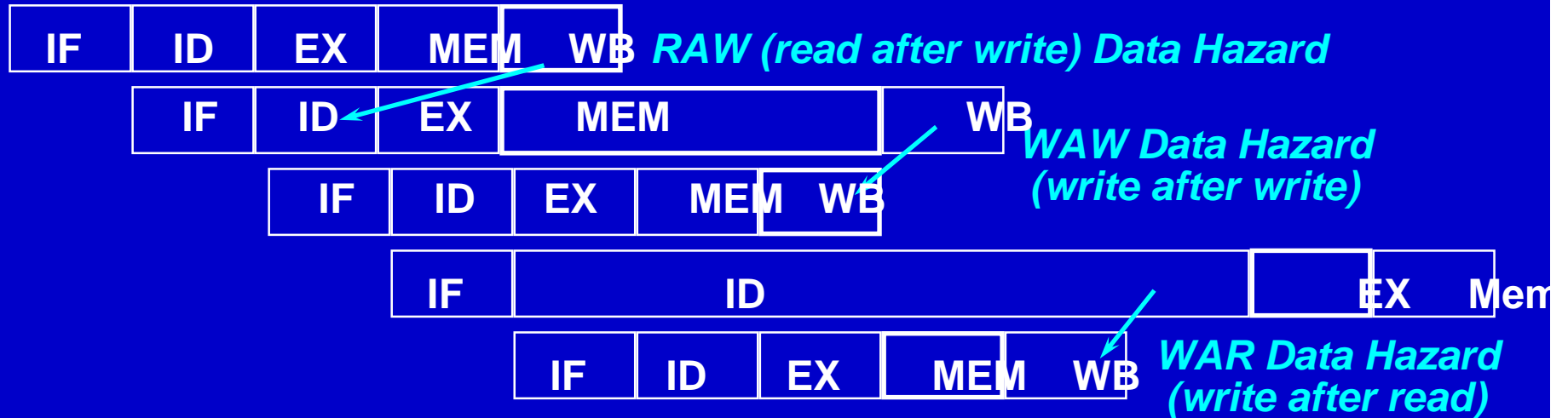
Fig. 6.28

Types of Data Hazards

Three types: (inst. i1 followed by inst. i2)

- **RAW (read after write):**
i2 tries to read operand before i1 writes it
- **WAR (write after read):**
i2 tries to write operand before i1 reads it
 - Gets wrong operand, e.g., autoincrement addr.
 - Can't happen in MIPS 5-stage pipeline because:
 - All instructions take 5 stages, and reads are always in stage 2, and writes are always in stage 5
- **WAW (write after write):**
i2 tries to write operand before i1 writes it
 - Leaves wrong result (i1's not i2's); occur only in pipelines that write in more than one stage
 - Can't happen in MIPS 5-stage pipeline because:
 - All instructions take 5 stages, and writes are always in stage 5

Pipeline Hazards Illustrated



Handling Data Hazards

- Use simple, fixed designs
 - Eliminate WAR by always fetching operands early (ID) in pipeline
 - Eliminate WAW by doing all write backs in order (last stage, static)
 - These features have a lot to do with ISA design
- Internal forwarding in register file:
 - Write in first half of clock and read in second half
 - Read delivers what is written, resolve hazard between sub and add
- **Detect** and **resolve** remaining ones
 - Compiler inserts NOP
 - Forward
 - Stall

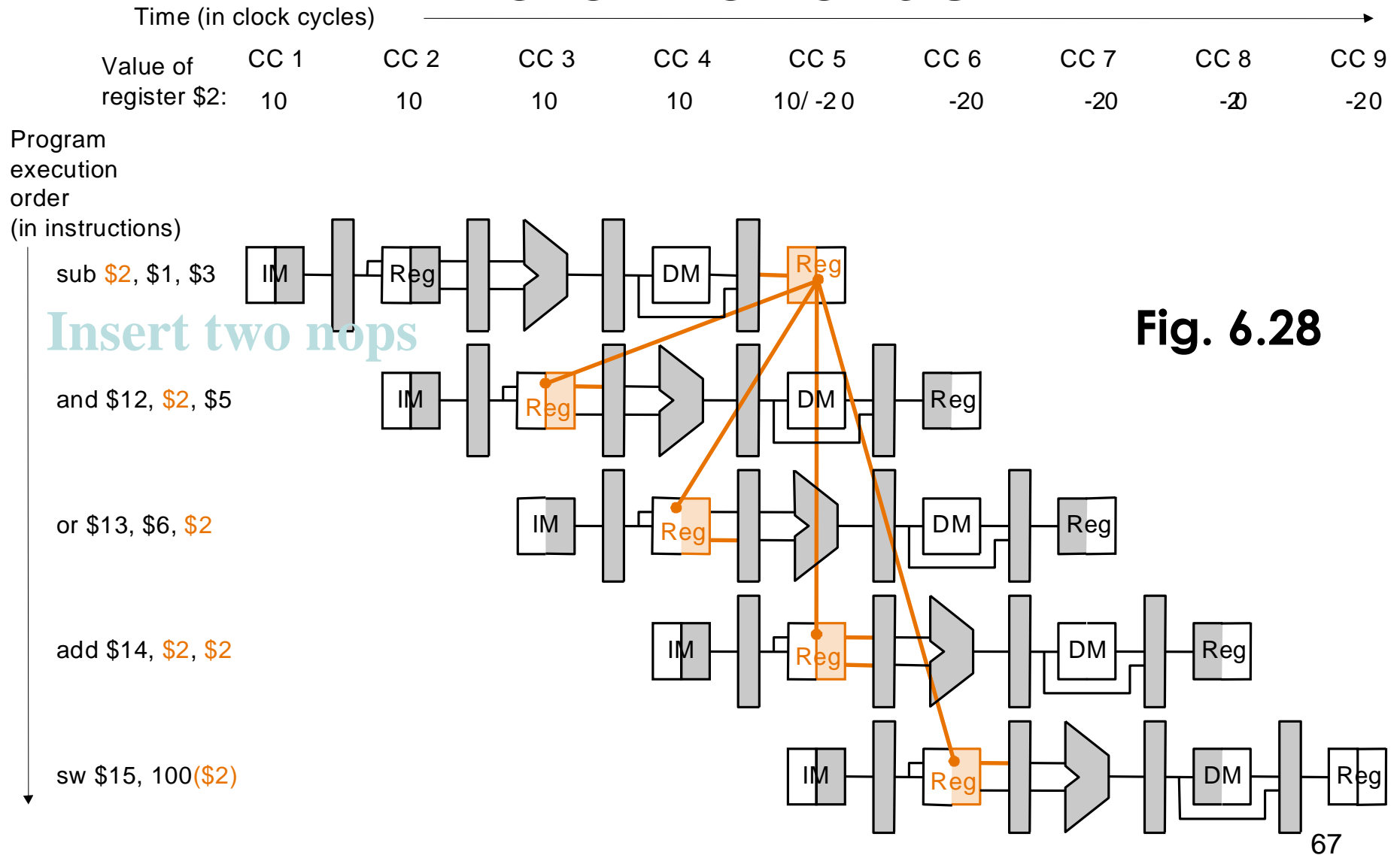
Software Solution

- Have compiler guarantee no hazards
- Where do we insert the NOPs?

```
sub  $2,  $1,  $3
and  $12, $2,  $5
or   $13, $6,  $2
add  $14, $2,  $2
sw   $15, 100($2)
```

- Problem: this really slows us down!

Data Hazards



Data Hazards : Forwarding

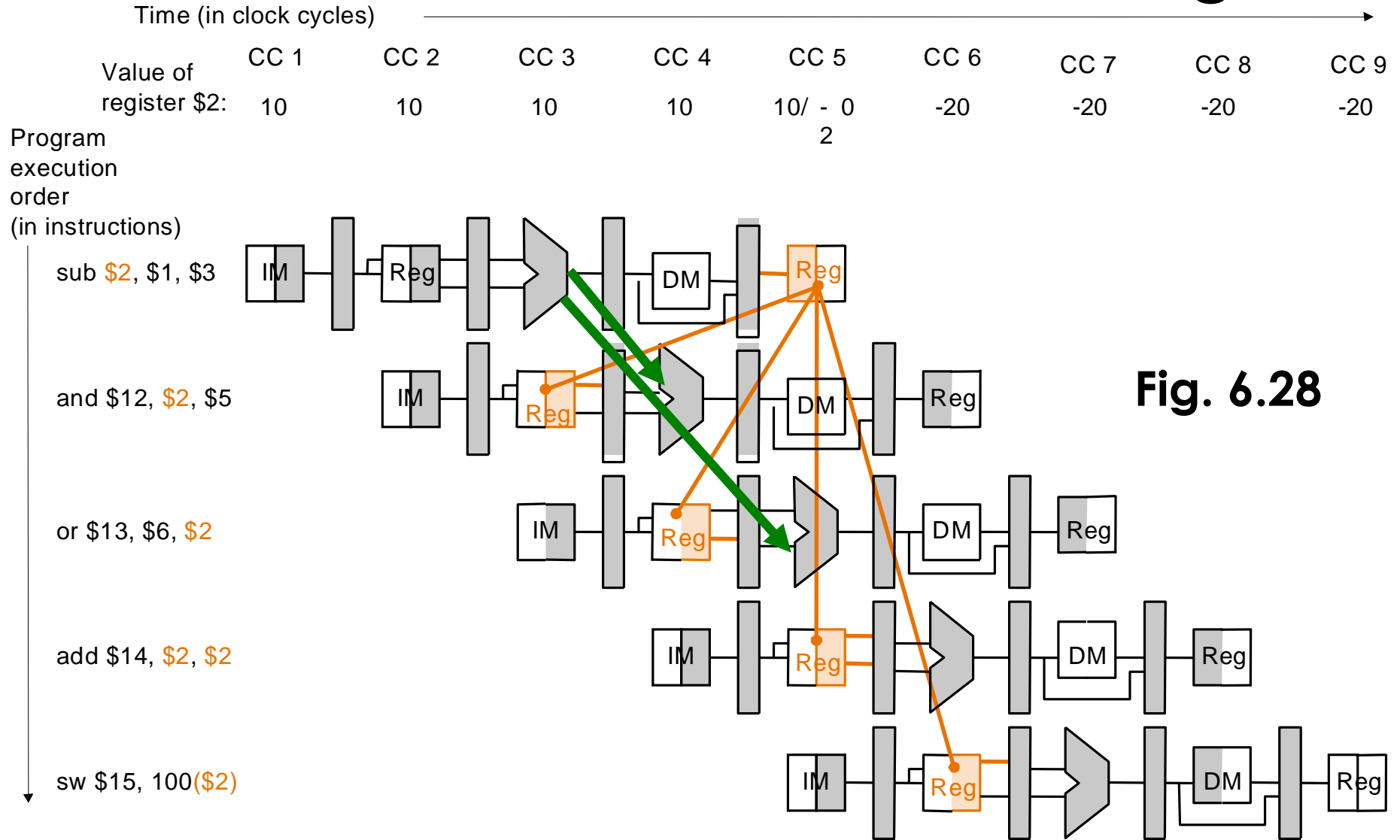


Fig. 6.28

Pipeline with Forwarding

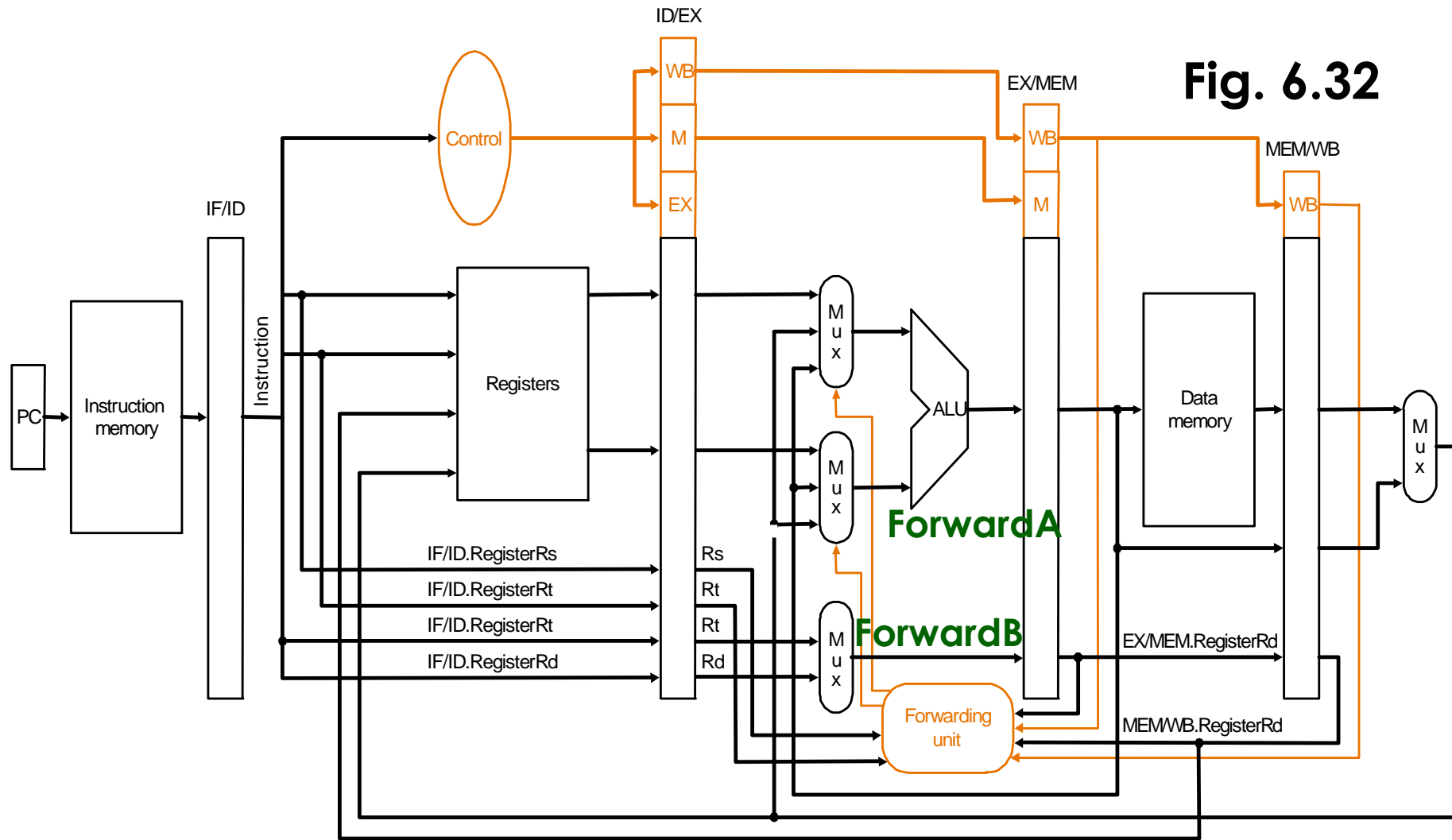


Fig. 6.32

Detecting Data Hazards

- Hazard conditions:
 - 1a. $EX/MEM.RegisterRd = ID/EX.RegisterRs$
 - 1b. $EX/MEM.RegisterRd = ID/EX.RegisterRt$
 - 2a. $MEM/WB.RegisterRd = ID/EX.RegisterRs$
 - 2b. $MEM/WB.RegisterRd = ID/EX.RegisterRt$
- Two optimizations:
 - Don't forward if instruction does not write register
=> check if RegWrite is asserted
 - Don't forward if destination register is \$0
=> check if RegisterRd = 0

Detecting Data Hazards (cont.)

- Hazard conditions using control signals:
 - At EX stage:
EX/MEM.RegWrite and (EX/MEM.RegRd≠0)
and (EX/MEM.RegRd=ID/EX.RegRs)
 - At MEM stage:
MEM/WB.RegWrite and (MEM/WB.RegRd≠0)
and (MEM/WB.RegRd=ID/EX.RegRs)
 - (replace ID/EX.RegRt for ID/EX.RegRs for the other two conditions)

Resolving Hazards: Forwarding

- Use temporary results, e.g., those in pipeline registers, don't wait for them to be written

Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register \$2 :	10	10	10	10	10/-20	-20	-20	-20	-20
Value of EX/MEM :	X	X	X	-20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	-20	X	X	X	X

Program execution order (in instructions)

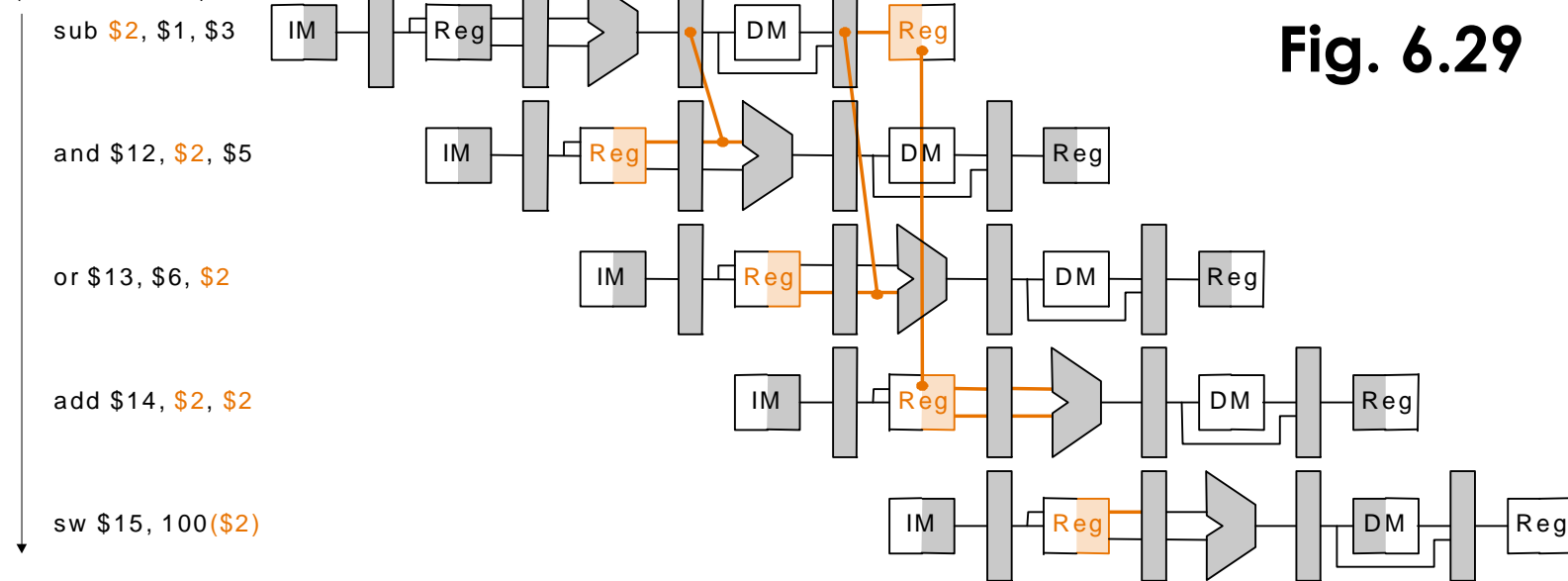


Fig. 6.29

Pipeline with Forwarding

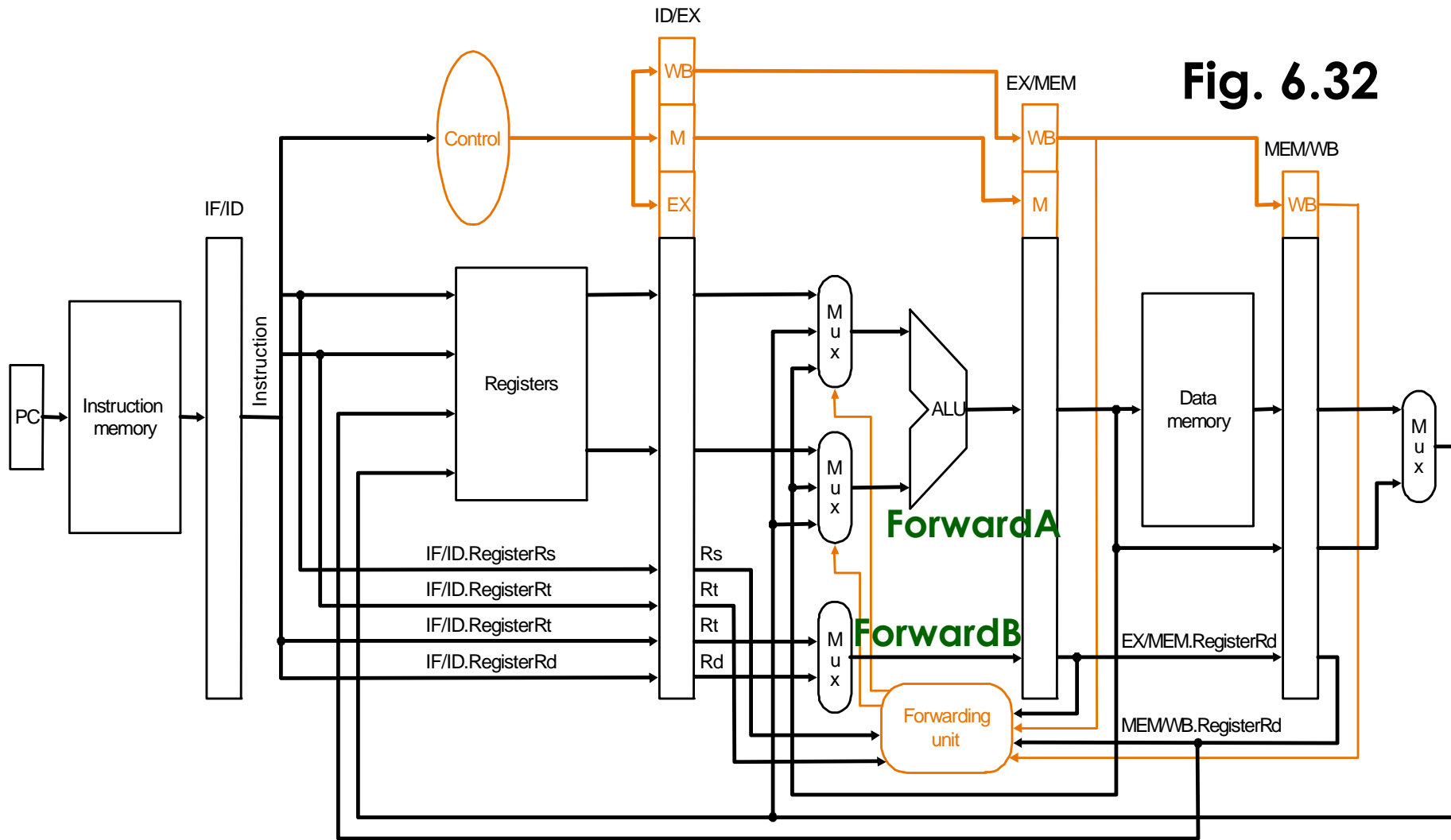



Fig. 6.32

Forwarding Logic

- Forwarding: input to ALU from any pipe reg.
 - Add multiplexors to ALU input 
 - Control forwarding in EX => carry Rs in ID/EX
- Control signals for forwarding:
 - If both WB and MEM forward, e.g., `add $1,$1,$2; add $1,$1,$3; add $1,$1,$4;` => let MEM forward
 - EX hazard:
 - `if (EX/MEM.RegWrite and (EX/MEM.RegRd≠0) and (EX/MEM.RegRd=ID/EX.RegRs)) ForwardA=10`
 - MEM hazard:
 - `if (MEM/WB.RegWrite and (MEM/WB.RegRd≠0) and (EX/MEM.RegRd ≠ ID/EX.Reg.Rs) and (MEM/WB.RegRd=ID/EX.RegRs)) ForwardA=01`

Example 3: Cycle 3

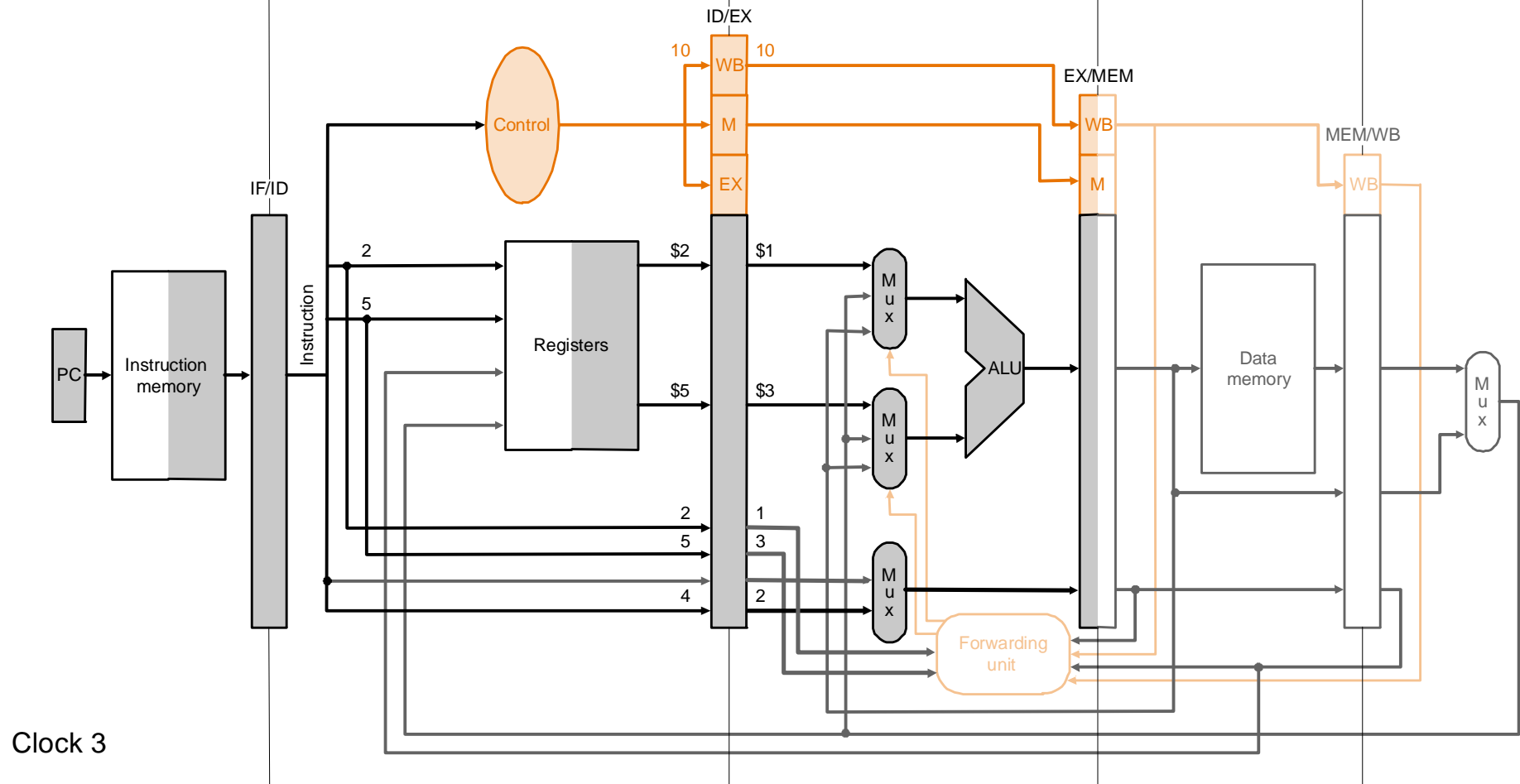
or \$4, \$4, \$2

and \$4, \$2, \$5

sub \$2, \$1, \$3

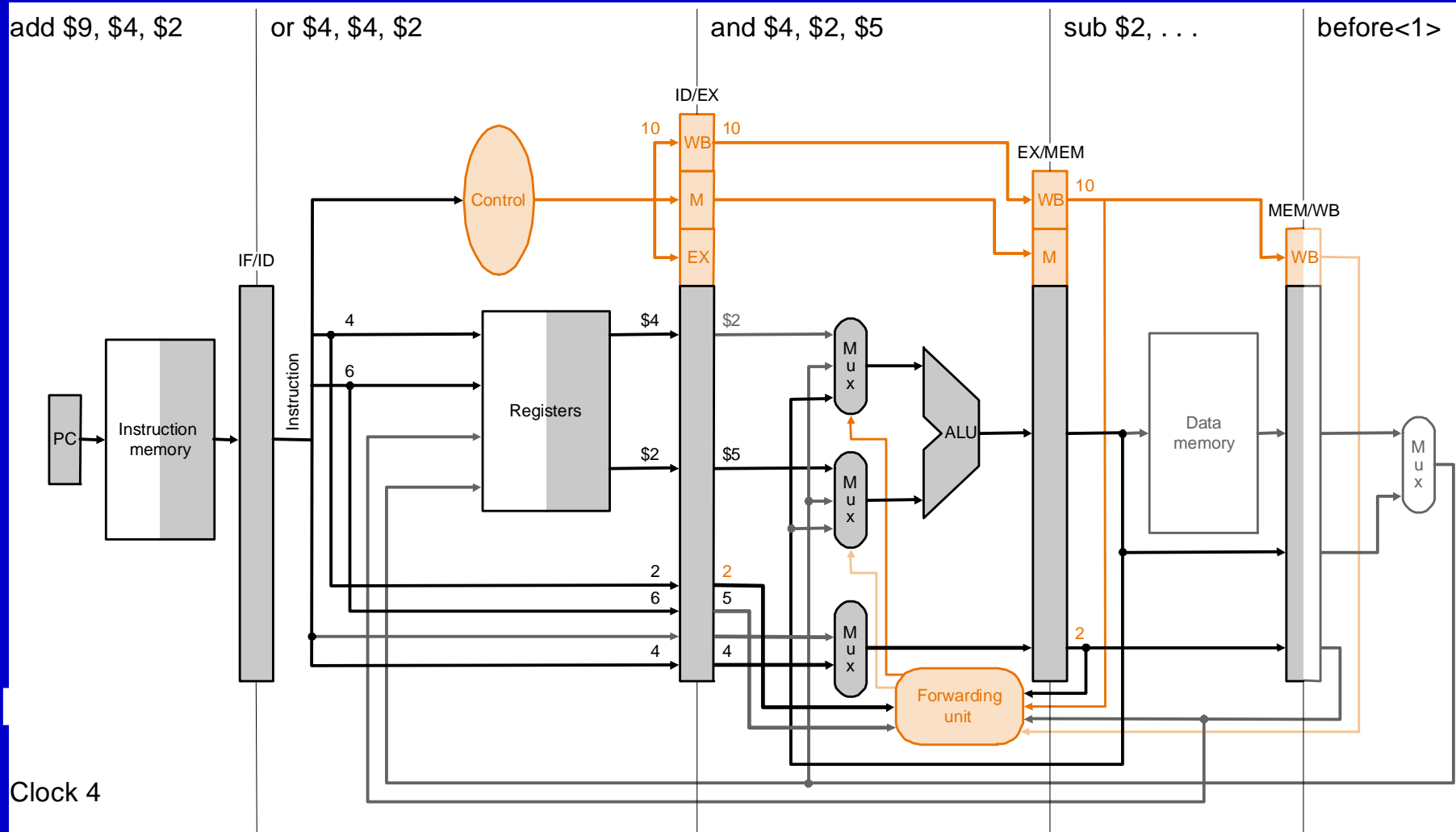
before<1>

before<2>

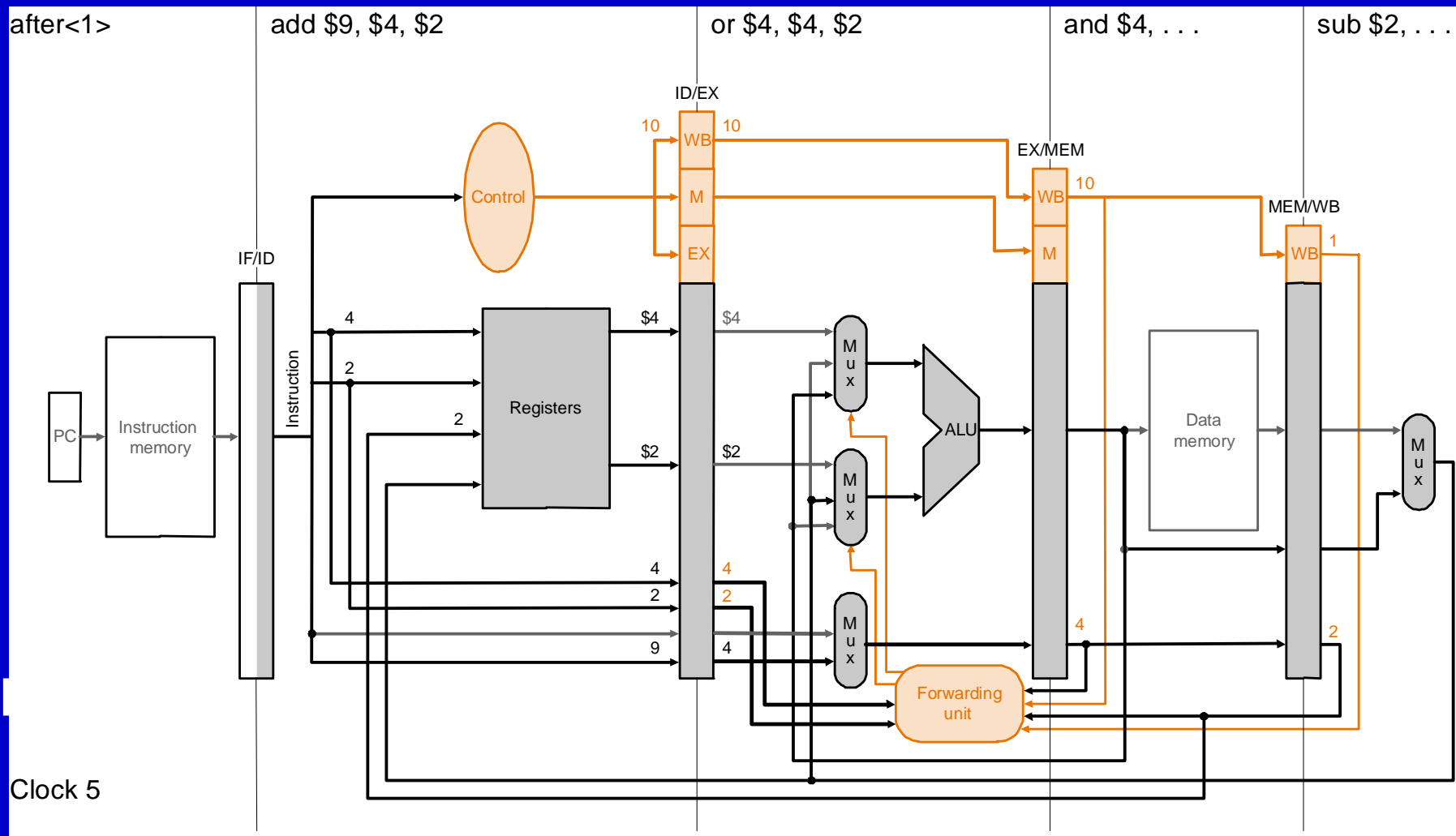


Clock 3

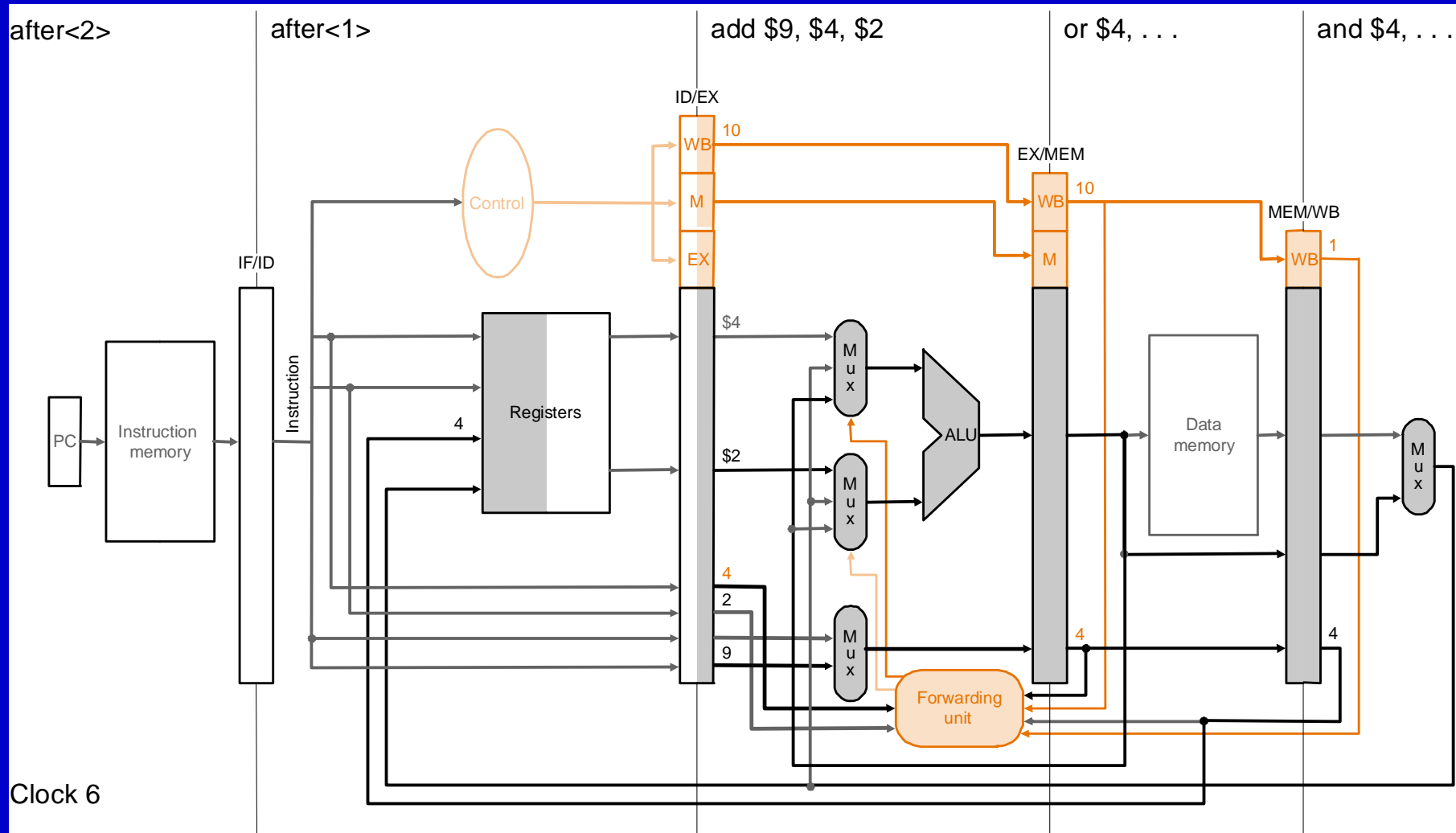
Example 3: Cycle 4



Example 3: Cycle 5



Example 3: Cycle 6



Can't Always Forward

- `lw` can still cause a hazard:
 - if it is followed by an instruction to read the loaded reg.

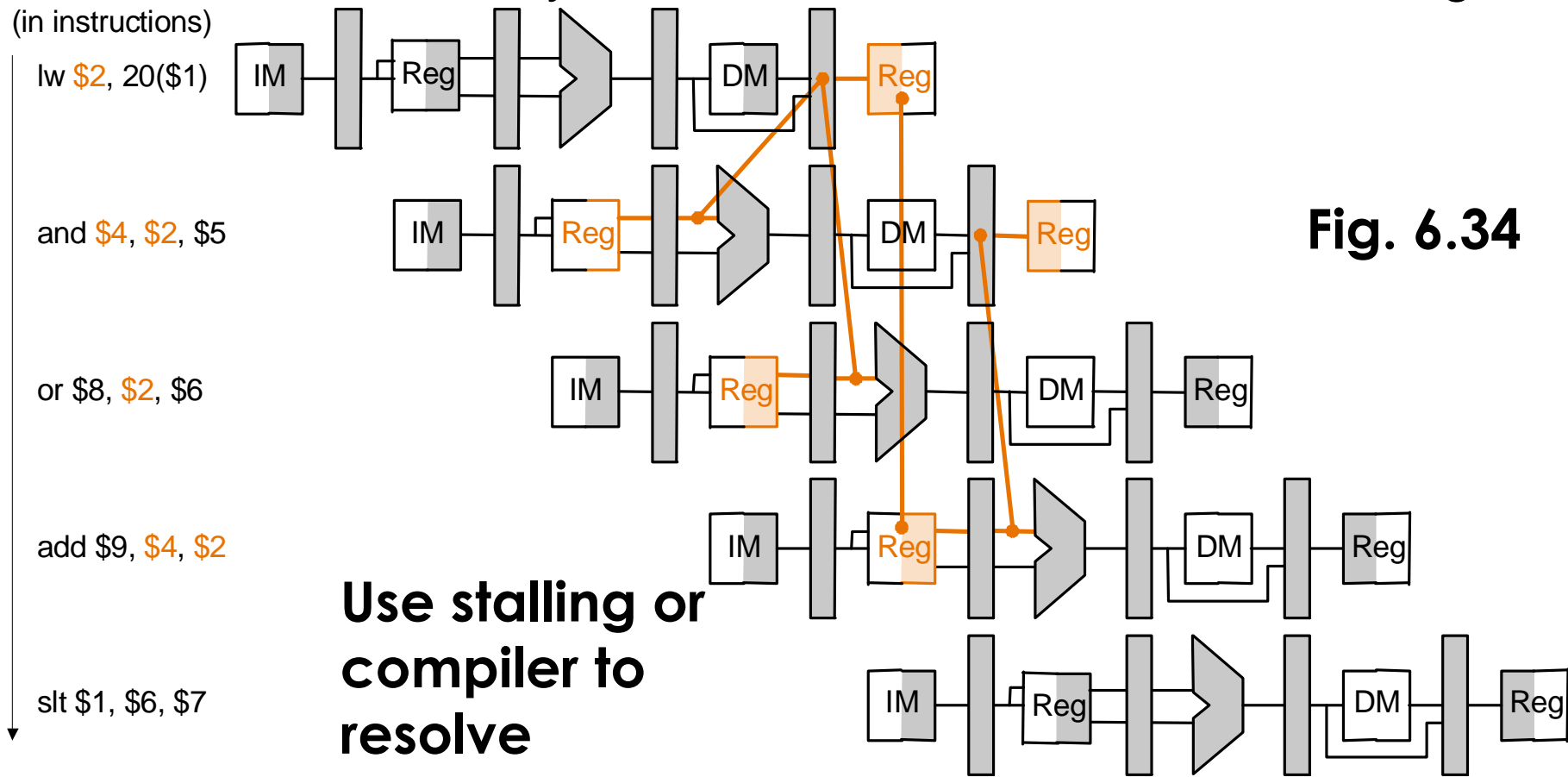
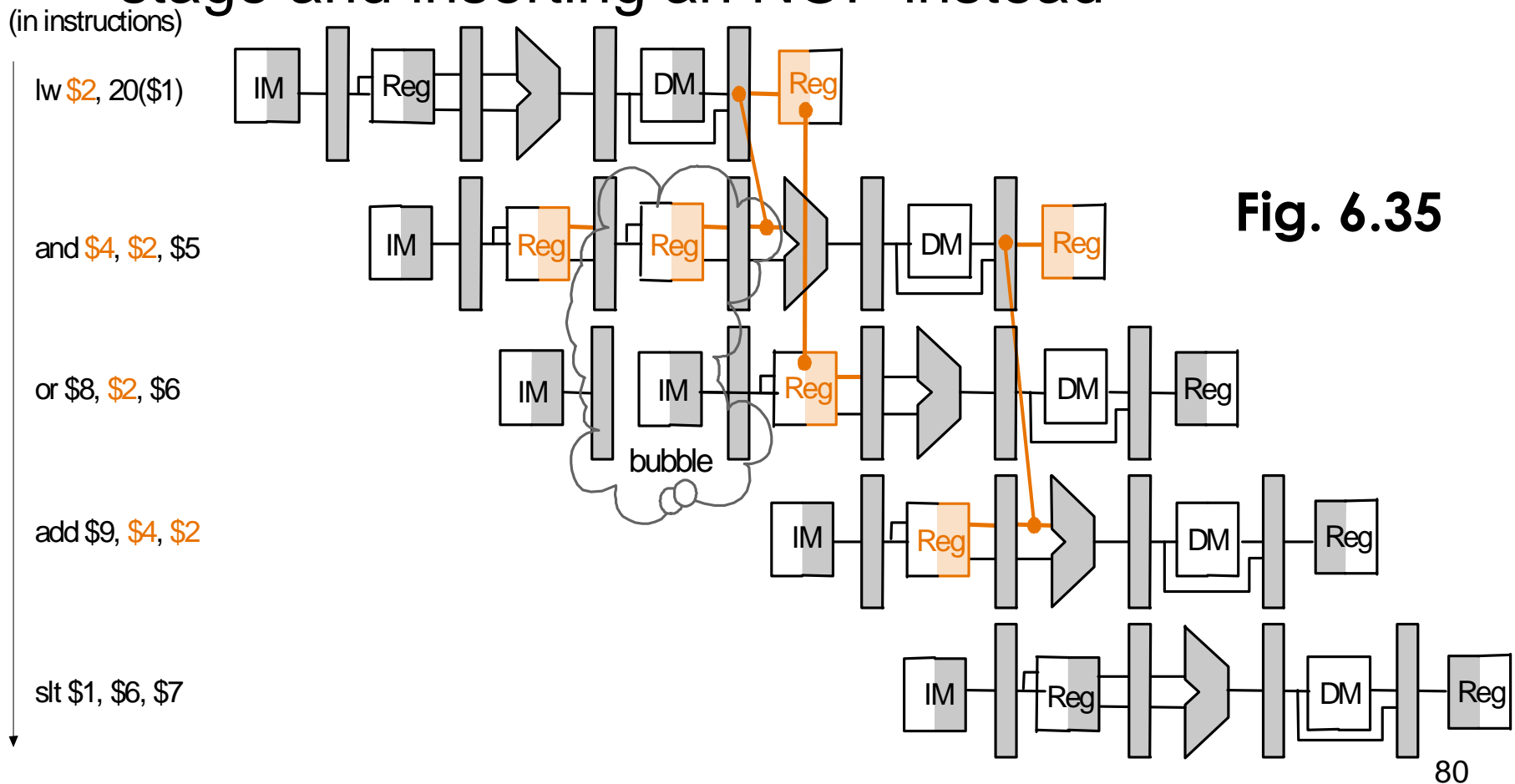


Fig. 6.34

Stalling

- Stall pipeline by keeping instructions in same stage and inserting an NOP instead



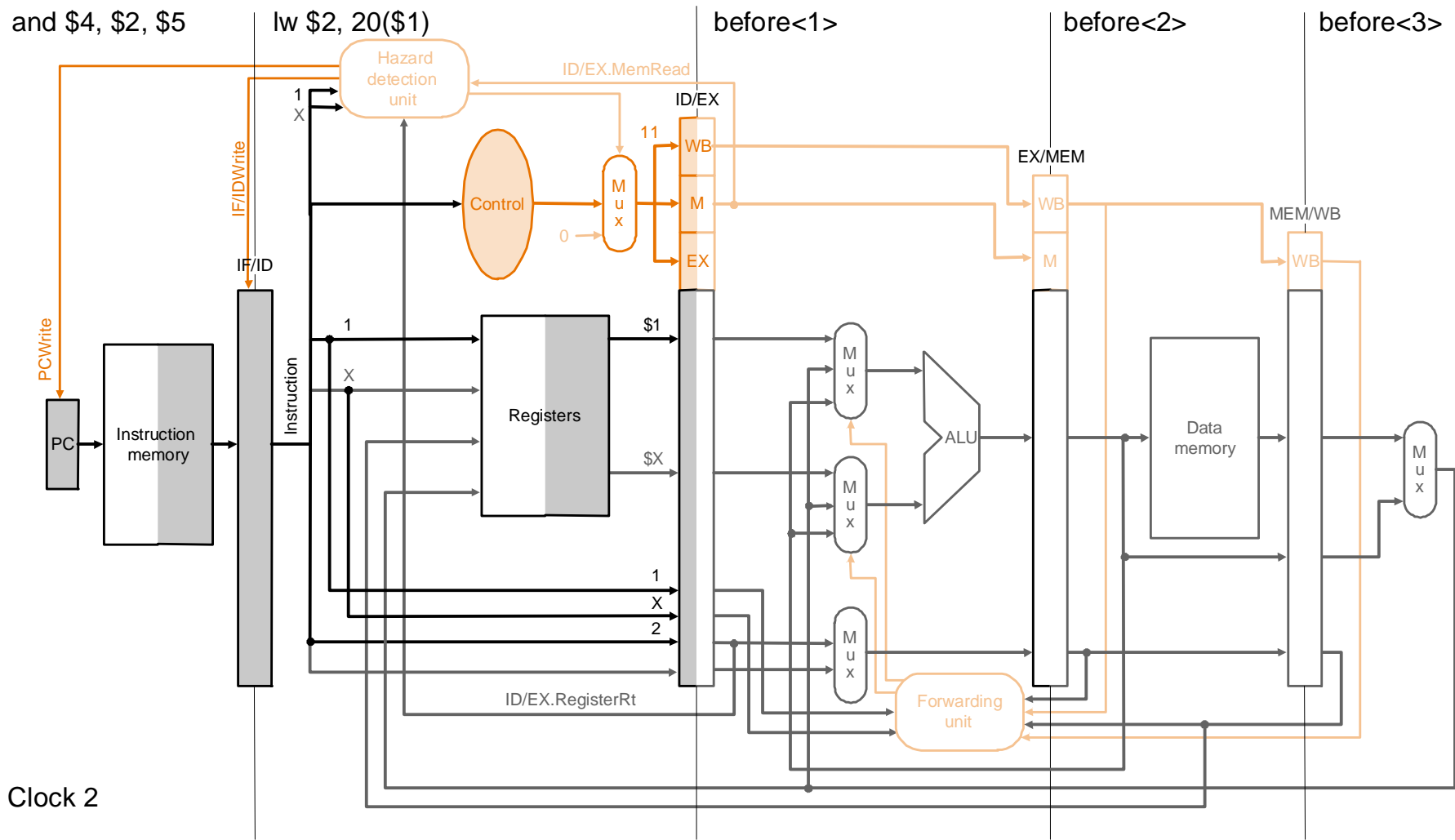
Handling Stalls

- Hazard detection unit in ID to insert stall between a load instruction and its use:

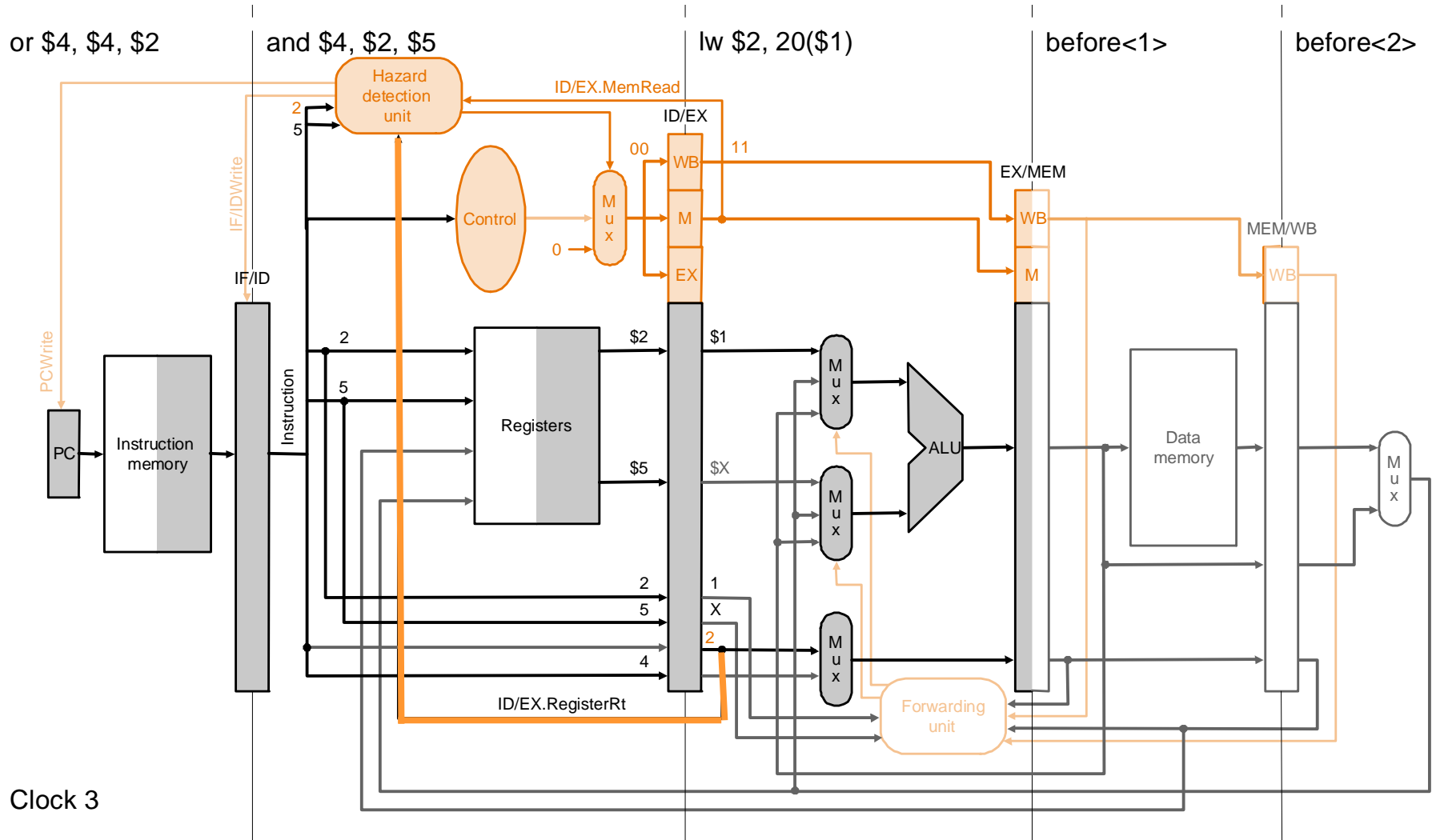
```
if (ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.registerRt))
    stall the pipeline for one cycle
(ID/EX.MemRead=1 indicates a load instruction)
```

- How to stall?
 - Stall instruction in IF and ID: not change PC and IF/ID
=> the stages re-execute the instructions
 - What to move into EX: insert an NOP by changing EX, MEM, WB control fields of ID/EX pipeline register to 0
 - as control signals propagate, all control signals to EX, MEM, WB are deasserted and no registers or memories are written

Example 4: Cycle 2

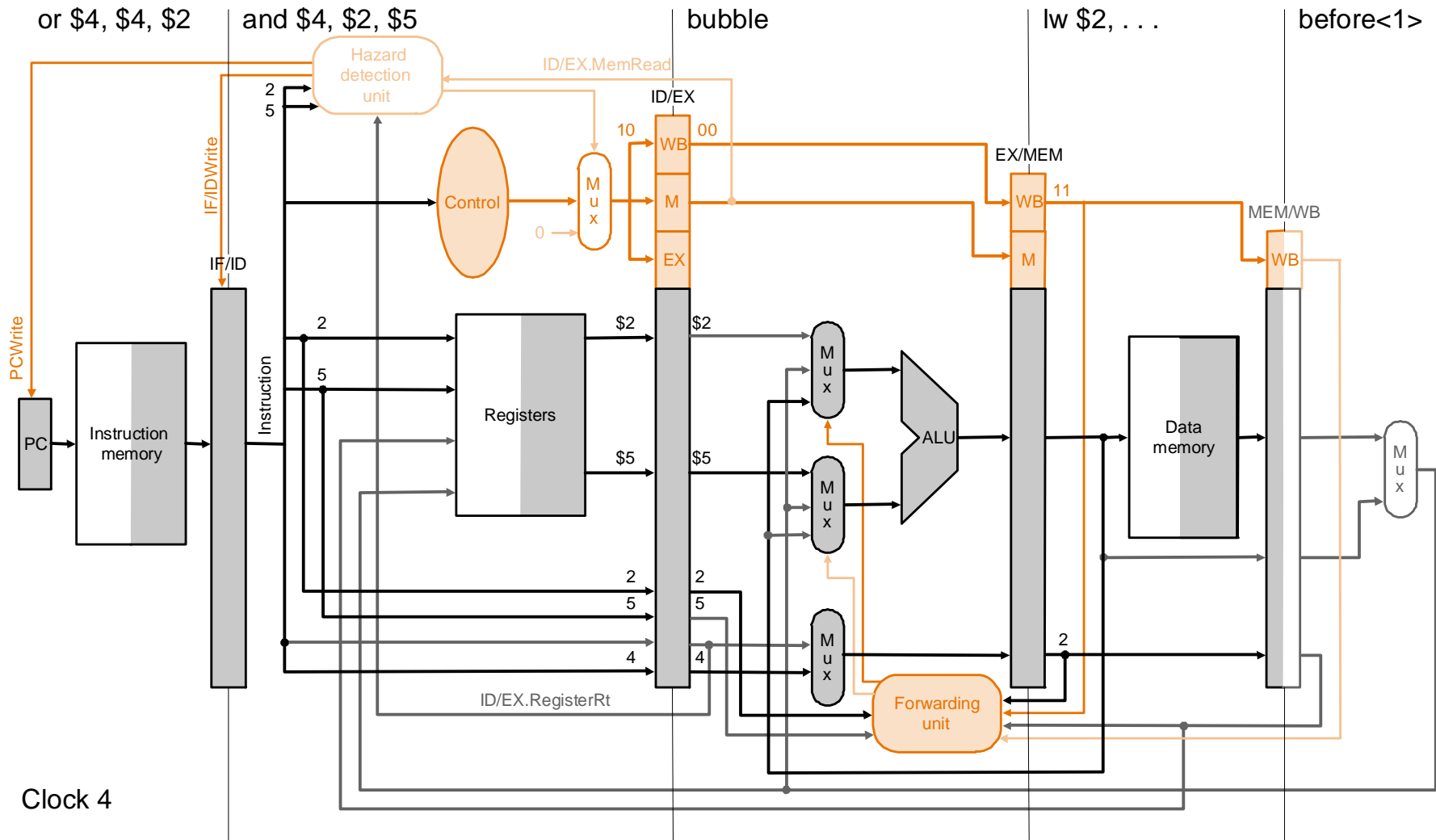


Example 4: Cycle 3

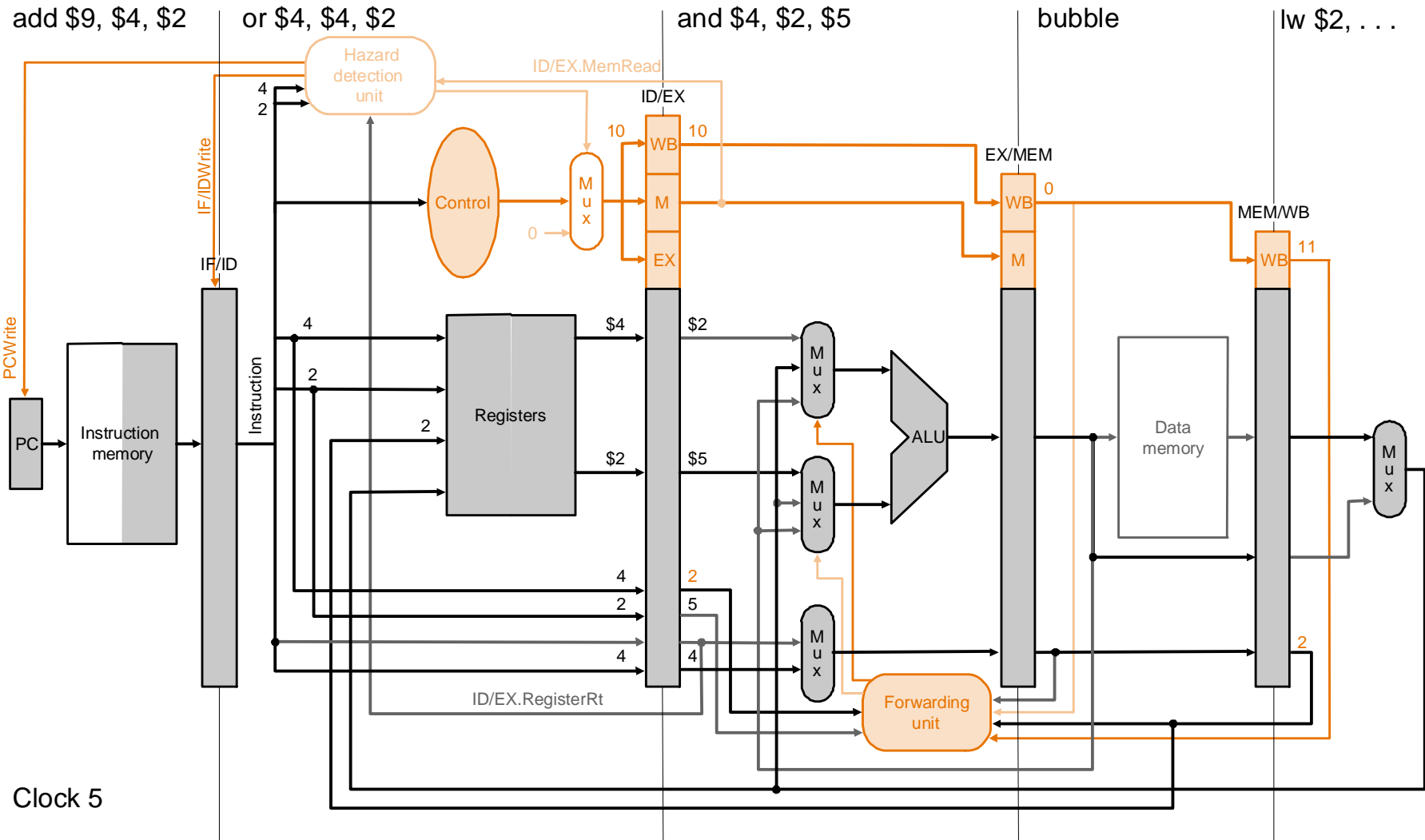


Clock 3

Example 4: Cycle 4

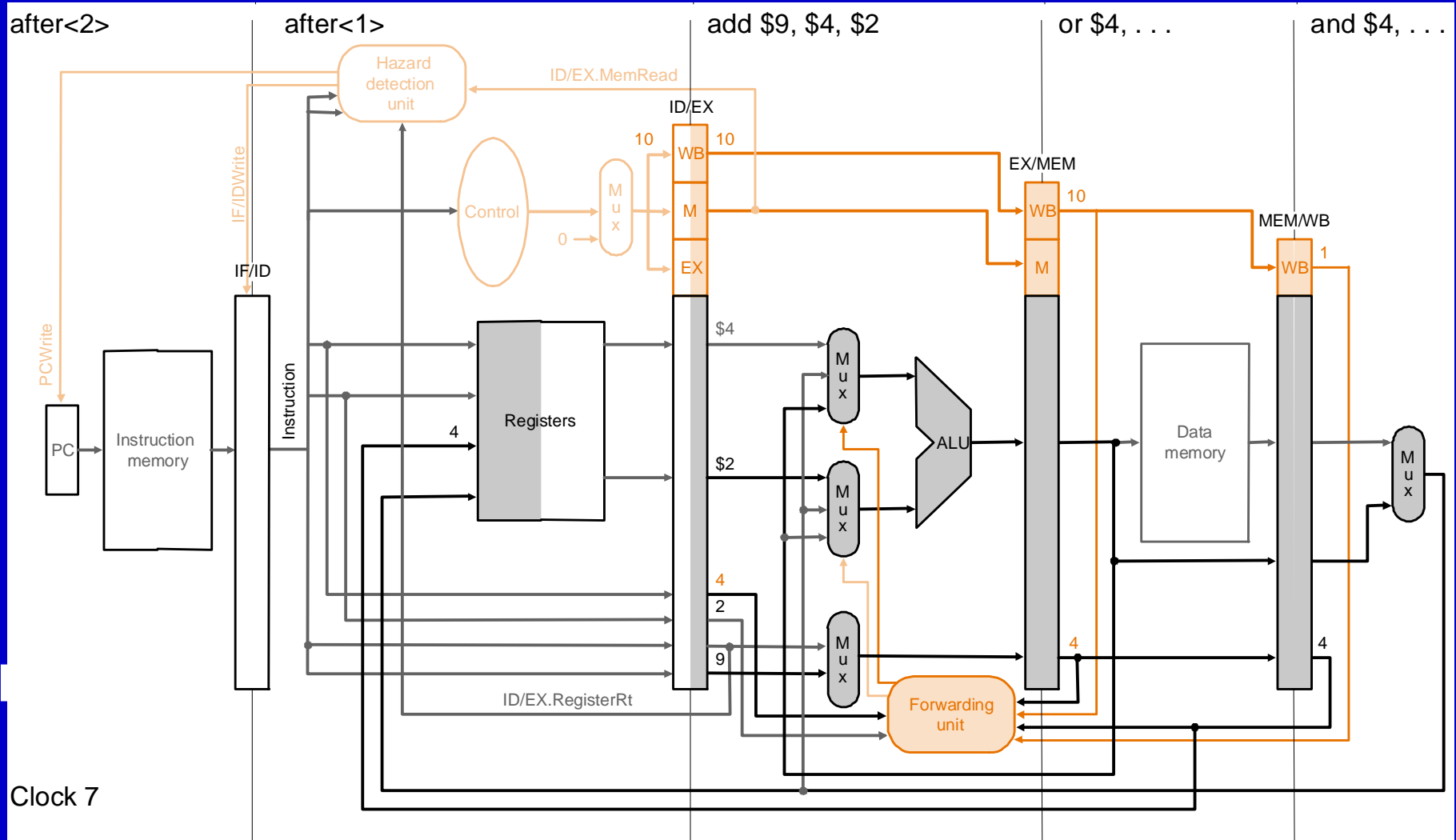


Example 4: Cycle 5



Clock 5

Example 4: Cycle 7



Clock 7

Outline

- An overview of pipelining
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards (6.6)
- Exceptions
- Superscalar and dynamic pipelining

Feedback Path

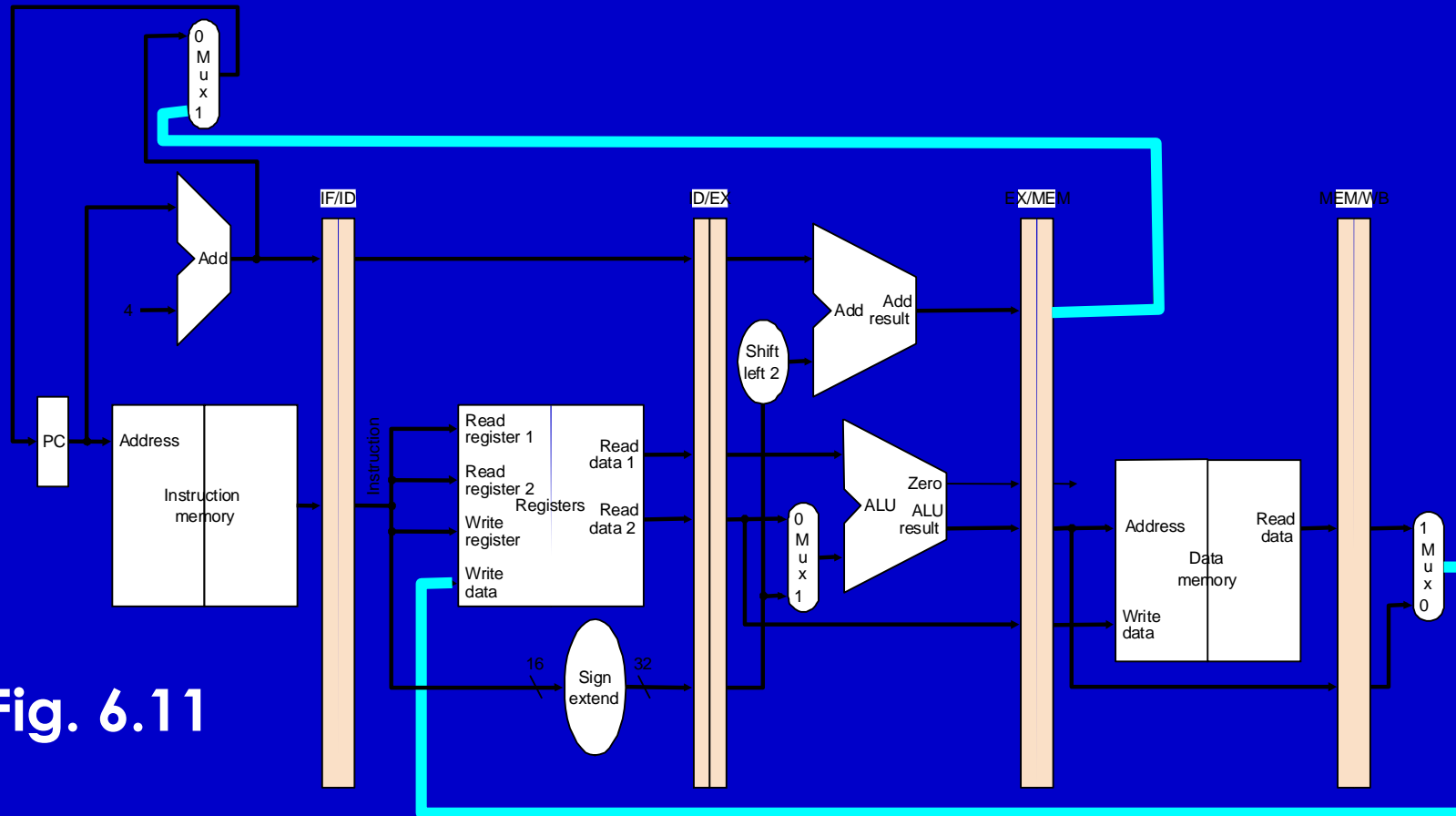
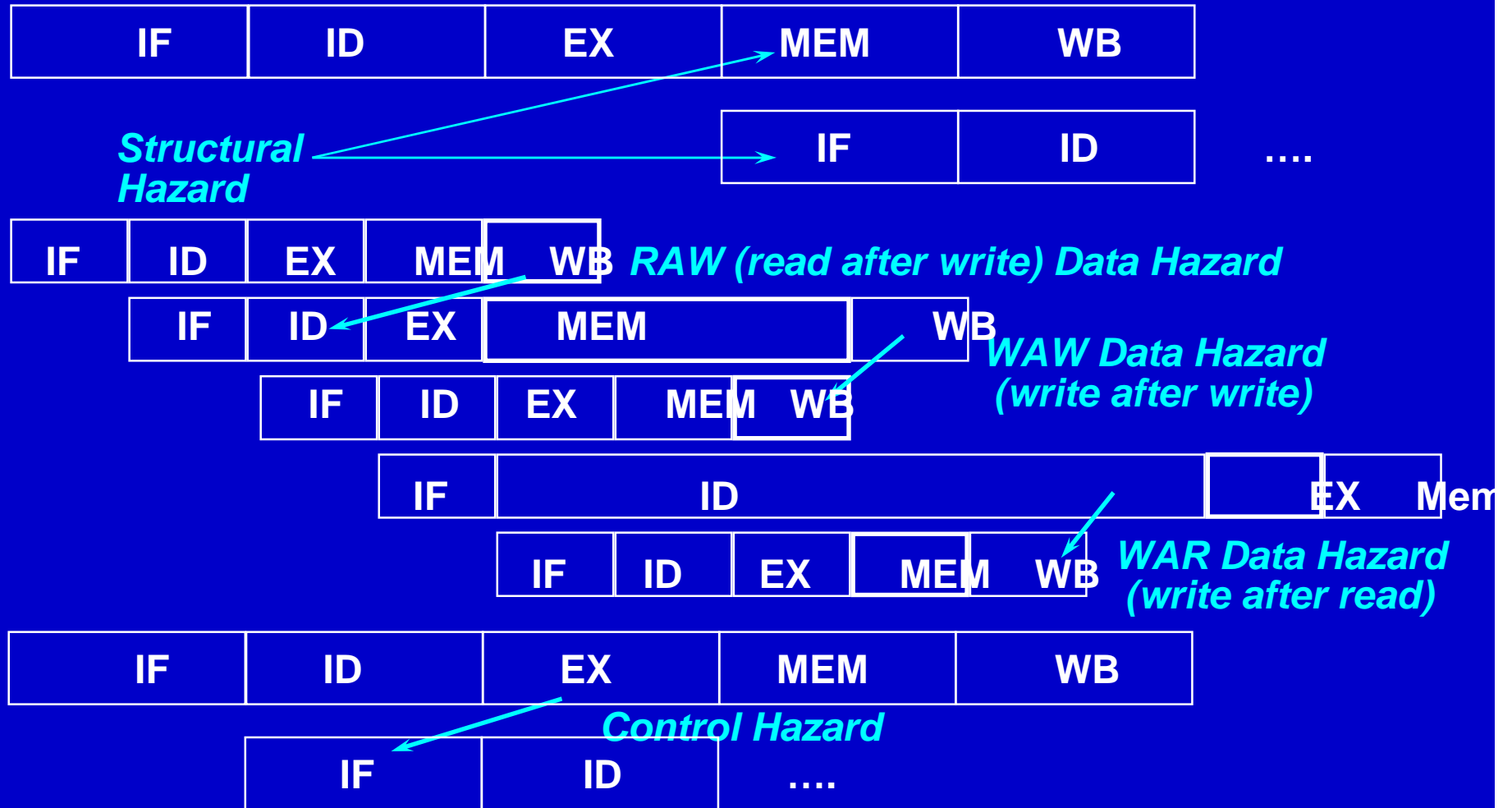


Fig. 6.11

Pipeline Hazards Illustrated



Branch Hazards

- When decide to branch, other inst. are in pipeline!

(in instructions)

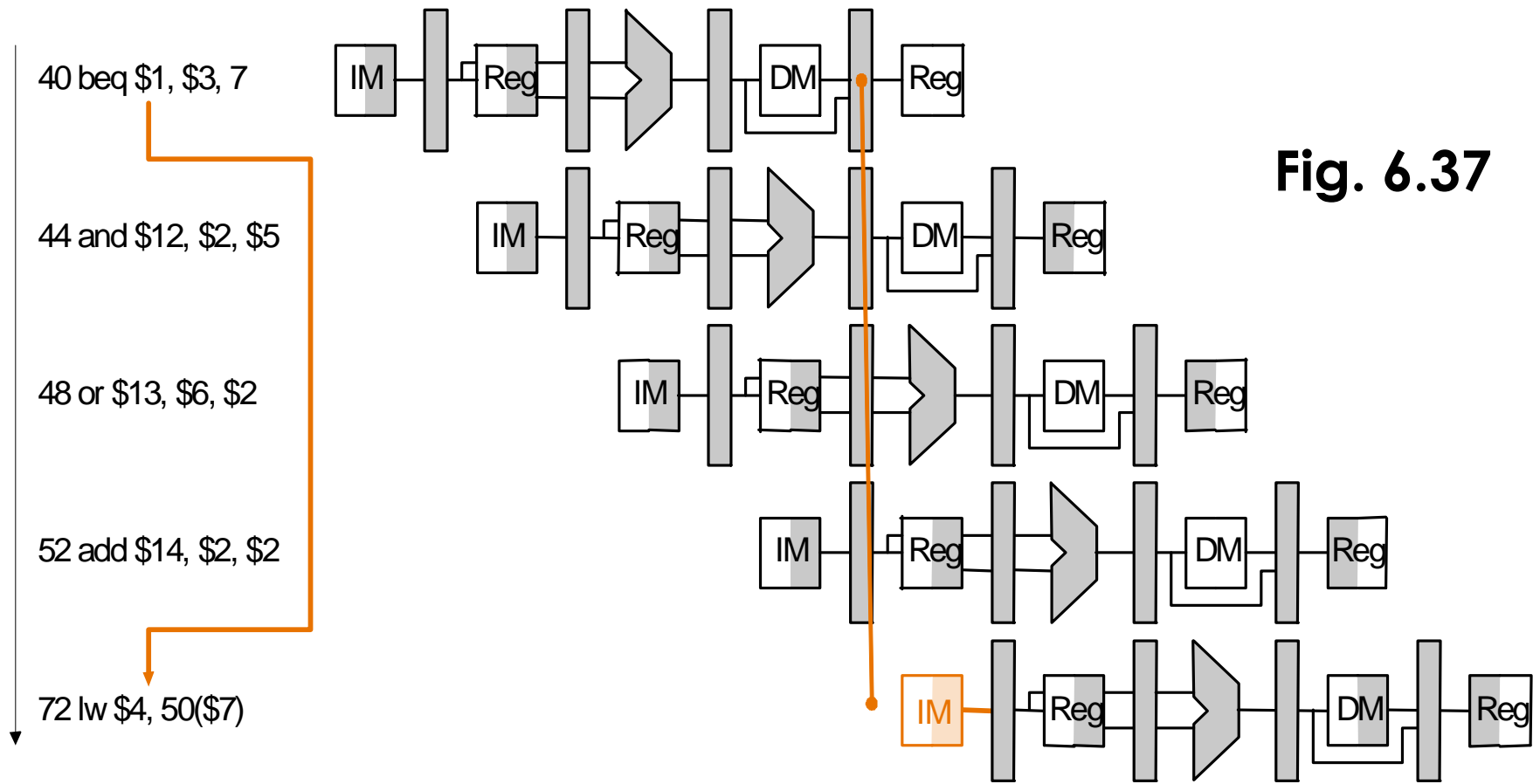



Fig. 6.37

Handling Branch Hazard

- Predict branch always not taken
 - Need to add hardware for flushing inst. if wrong
 - Branch decision made at MEM => need to flush instruction in IF/ID, ID/MEM by changing control values to 0
- Reduce delay of taken branch by moving branch execution earlier in the pipeline
 - Move up branch address calculation to ID
 - Check branch equality at ID (using XOR) by comparing the two registers read during ID
 - Branch decision made at ID => one instruction to flush
 - Add a control signal, IF.Flush, to zero instruction field of IF/ID => making the instruction an NOP 
- Dynamic branch prediction
- Compiler rescheduling, delay branch

Pipeline with Flushing

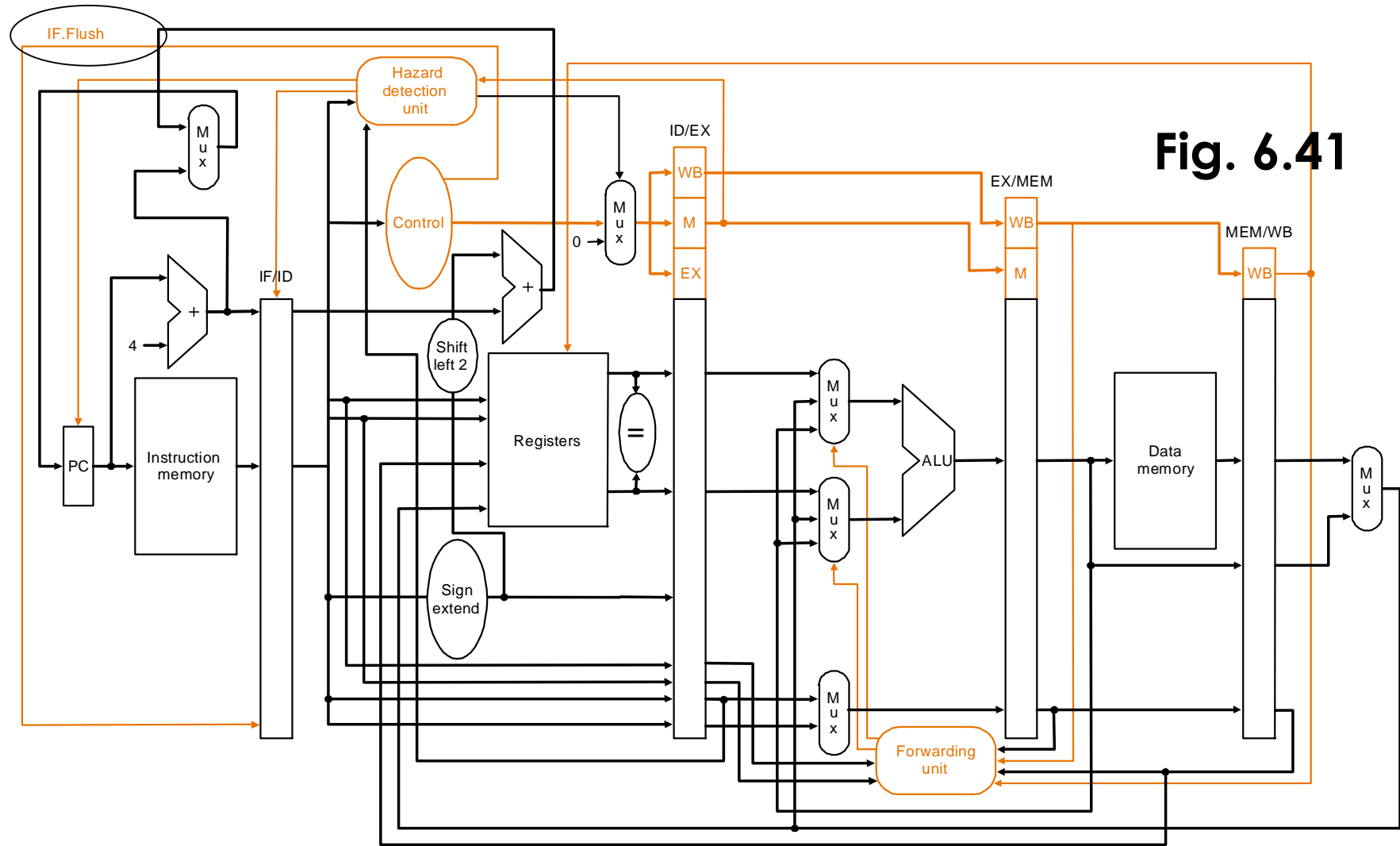


Fig. 6.41

Example 5: Cycle 3

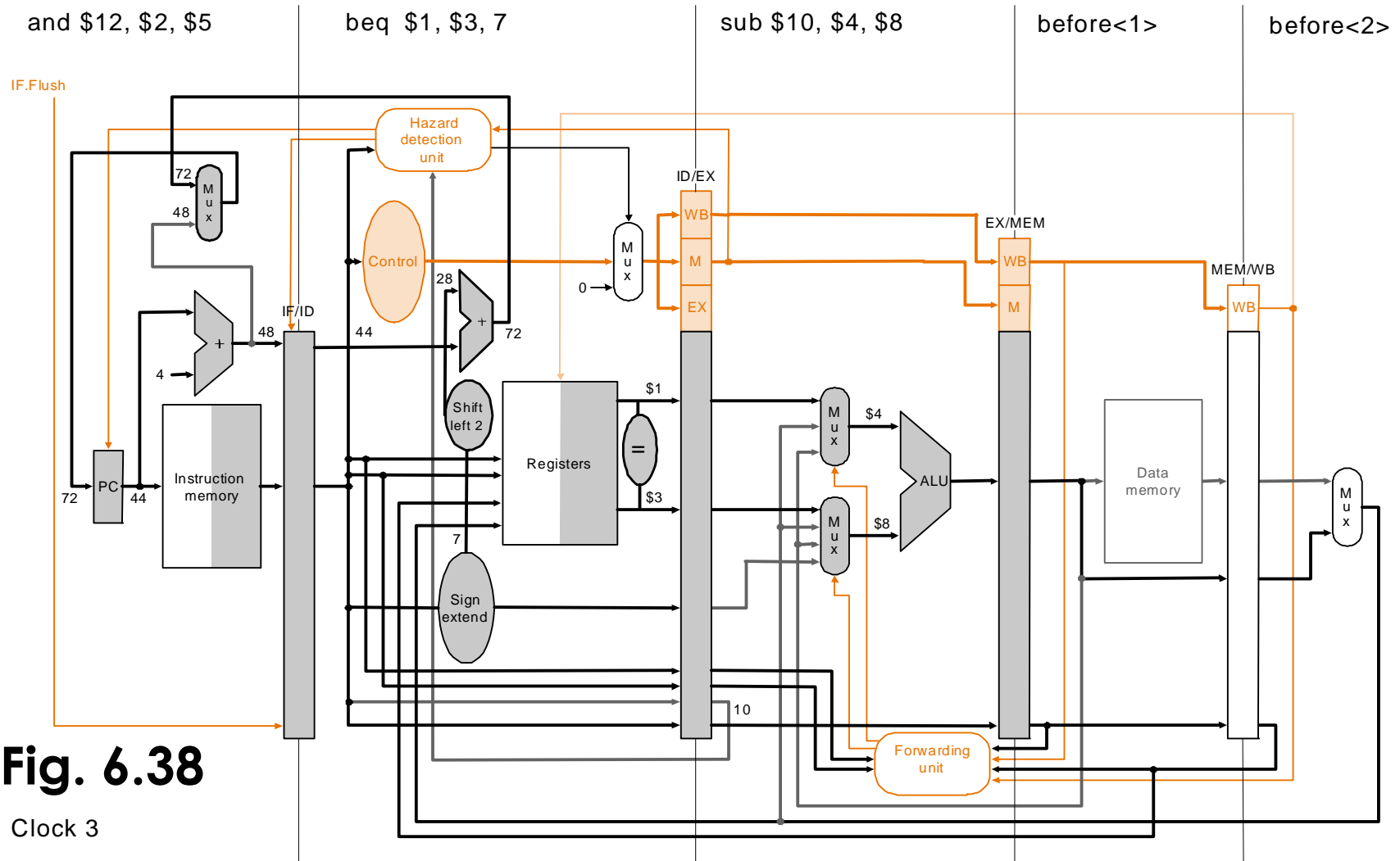


Fig. 6.38

Clock 3

Example 5: Cycle 4

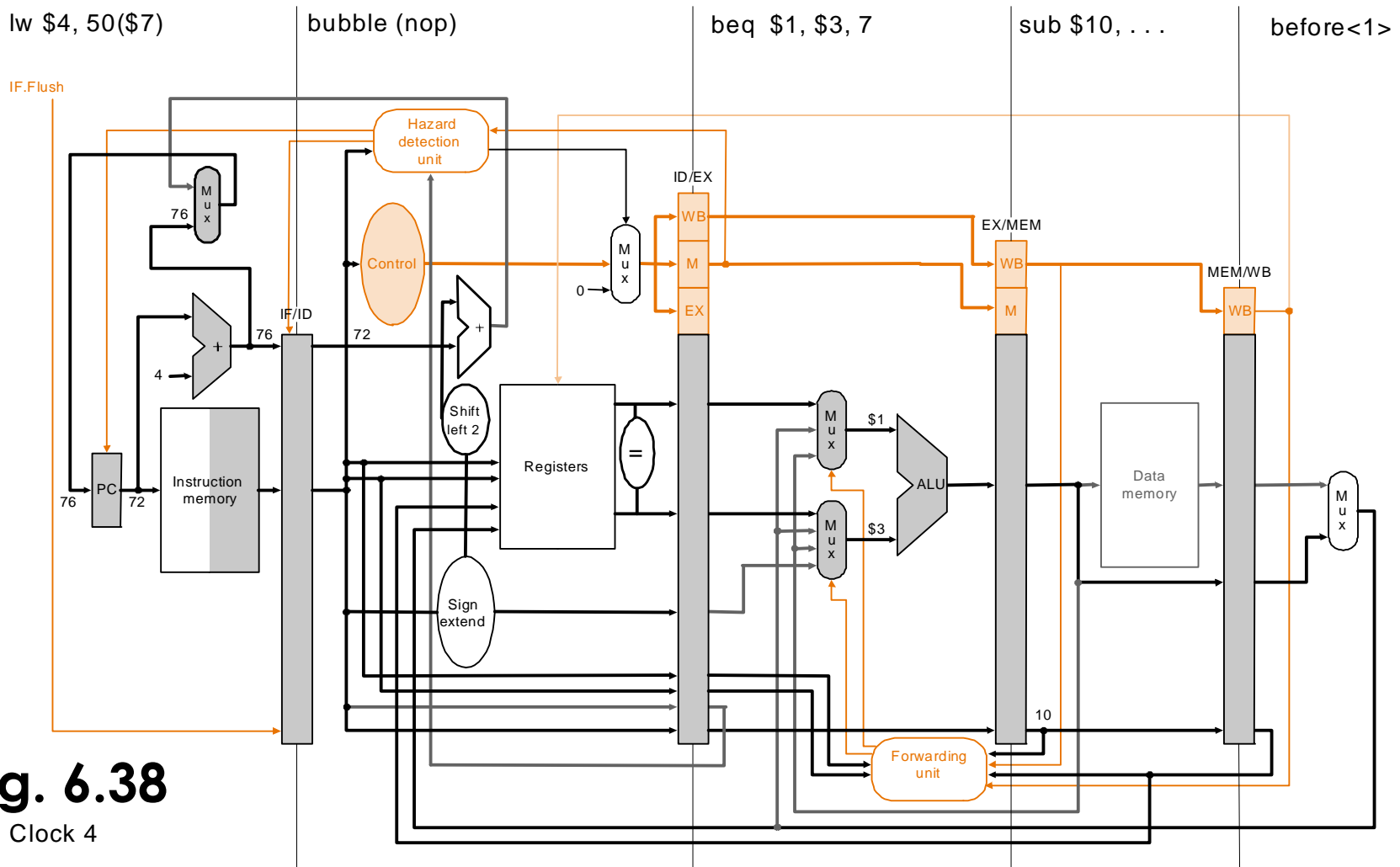
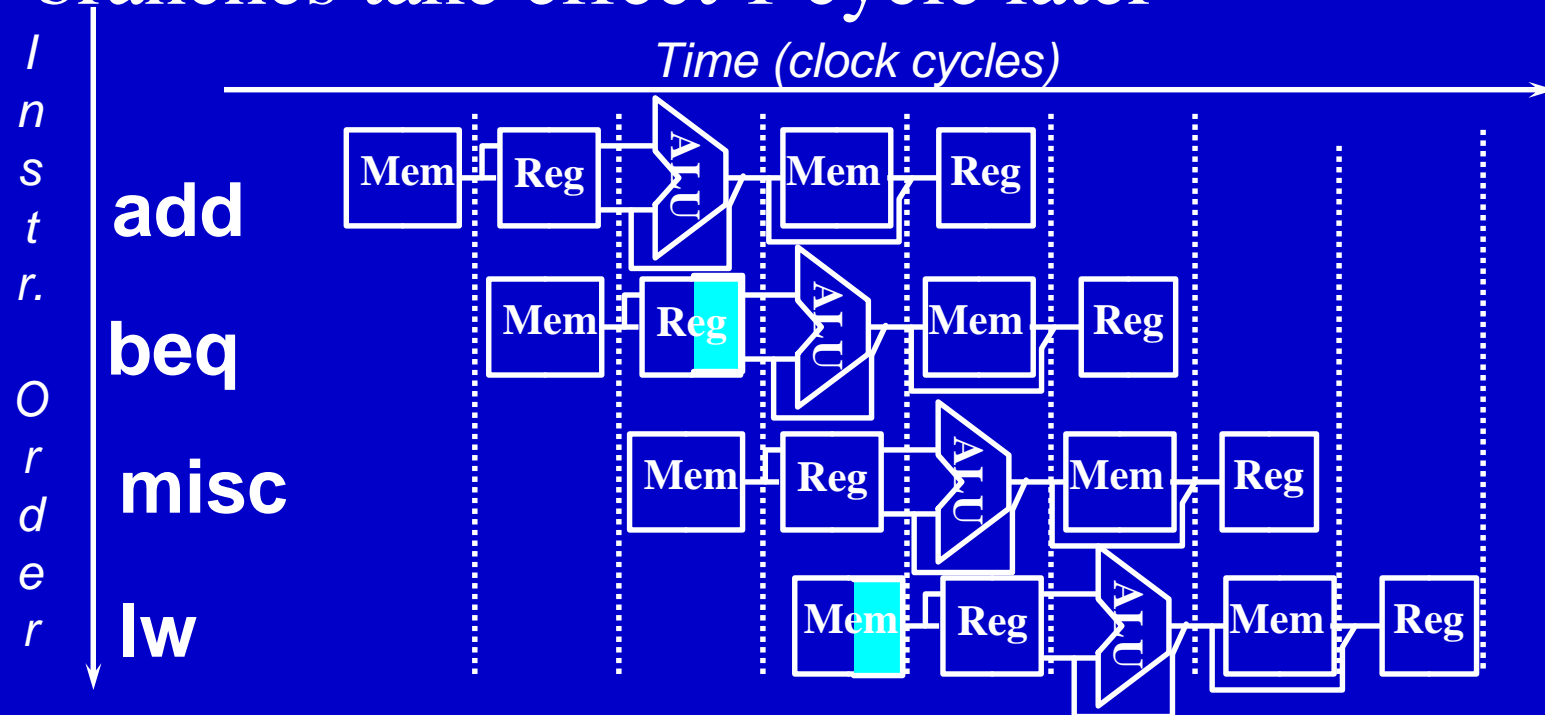


Fig. 6.38

Clock 4

Delayed Branch

- Predict-not-taken + branch decision at ID
 - => the following instruction is always executed
 - => branches take effect 1 cycle later



Dynamic Branch Prediction

- Performance = $f(\text{accuracy, cost of misprediction})$
- Branch History Table: Lower bits of PC address index table of 1-bit values
 - Says whether or not branch taken last time
 - No address check
- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iterations before exit):
 - End of loop case, when it exits instead of looping as before
 - First time through loop on *next* time through code, when it predicts exit instead of looping

1-Bit Prediction

- For each branch, keep track of what happened last time and use that outcome as the prediction
- What are prediction accuracies for branches 1 and 2 below:

```
while (1) {  
    for (i=0;i<10;i++) { branch-1  
        ...  
    }  
    for (j=0;j<20;j++) { branch-2  
        ...  
    }  
}
```

2-Bit Prediction

- For each branch, maintain a 2-bit saturating counter:
if the branch is taken: $\text{counter} = \min(3, \text{counter} + 1)$
if the branch is not taken: $\text{counter} = \max(0, \text{counter} - 1)$
- If ($\text{counter} \geq 2$), predict taken, else predict not taken
- Advantage: a few atypical branches will not influence the prediction (a better measure of “the common case”)
- Especially useful when multiple branches share the same counter (some bits of the branch PC are used to index into the branch predictor)
- Can be easily extended to N-bits (in most processors, $N=2$)

N-bit Branch Prediction Buffers

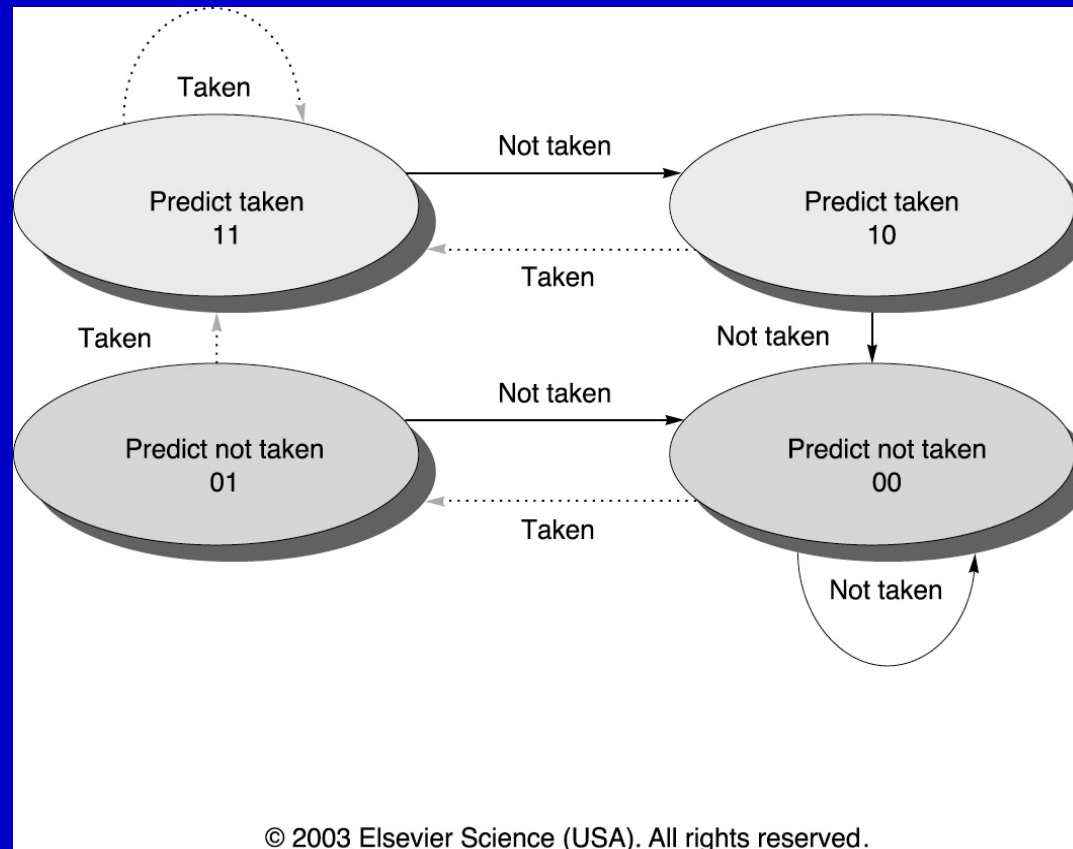
- When the counter is greater than or equal to one-half of its maximum value (2^{n-1}), the branch is predicted as taken.
- The counter is increased on a taken branch and decremented on an untaken branch.
- A branch buffer can be implemented as a small cache accessed during the **IF stage**.

N-bit Branch Prediction Buffers

Use an n-bit saturating counter

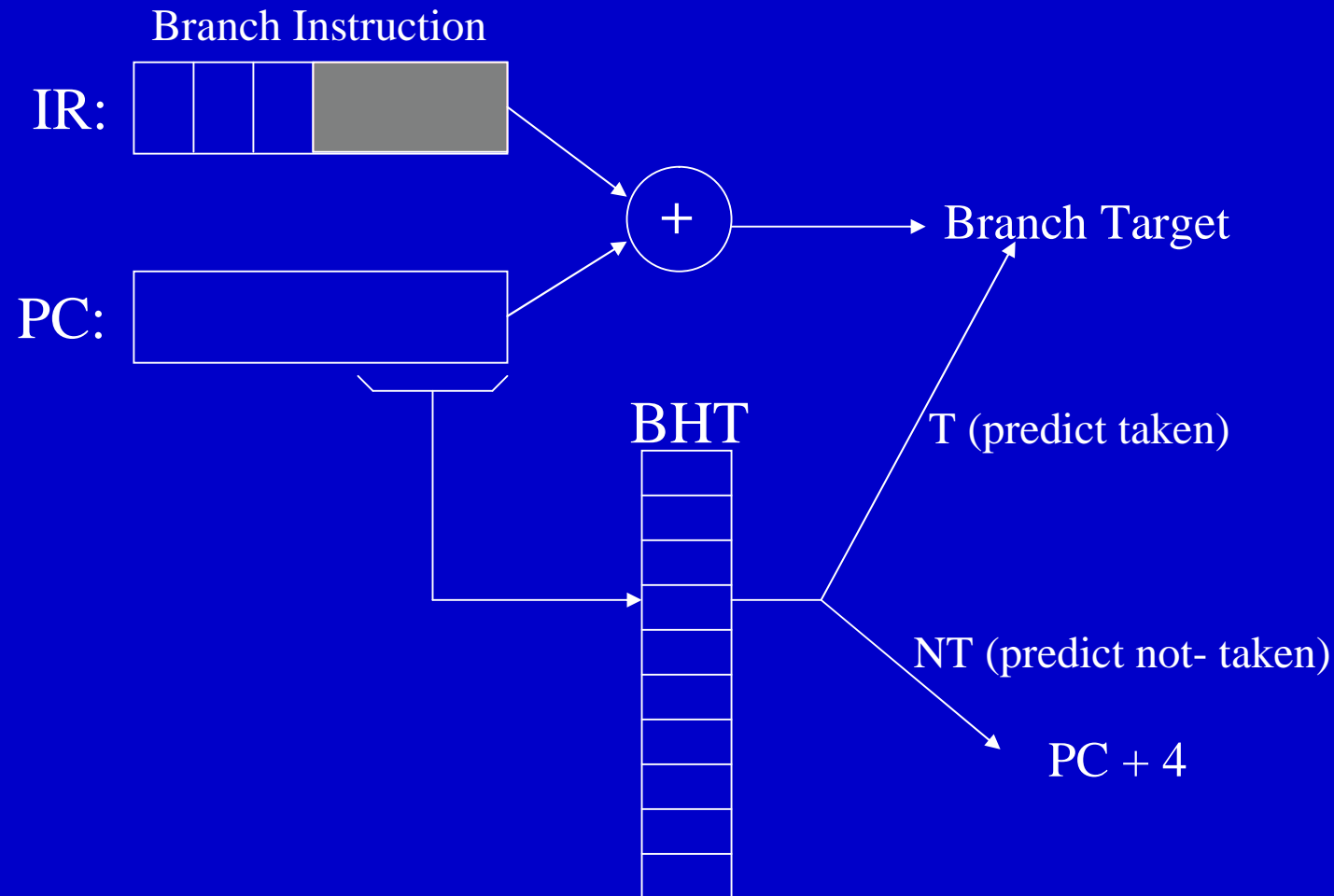
Only the loop exit causes a misprediction

2-bit predictor almost as good as any general n-bit predictor



Basic Branch Prediction Buffers

a.k.a. **Branch History Table (BHT)** - Small direct-mapped cache of T/NT bits



Outline

- An overview of pipelining
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards
- Exceptions
- Superscalar and dynamic pipelining

What about Exceptions?

- Another form of branch hazard
 - How to stop the pipeline? restart?
 - Who caused the interrupt?
 - Who to serve first, if multiple interrupts at the same time?

Handling Exceptions

How to stop the pipeline? restart?

- Suppose overflow occur at add \$1, \$2, \$1
 - Disable writes of instructions till trap hits WB, e.g., flush following instructions using IF.Flush, ID.Flush, EX.Flush to cause multiplexers to zero control signals (overflow exception detected at EX => flush offending instruction)
 - Force trap instruction into IF, e.g., fetch from 4000 0040hex by adding 4000 0040hex to PC input MUX
 - Save address of offending instruction in EPC

Pipeline with Exception

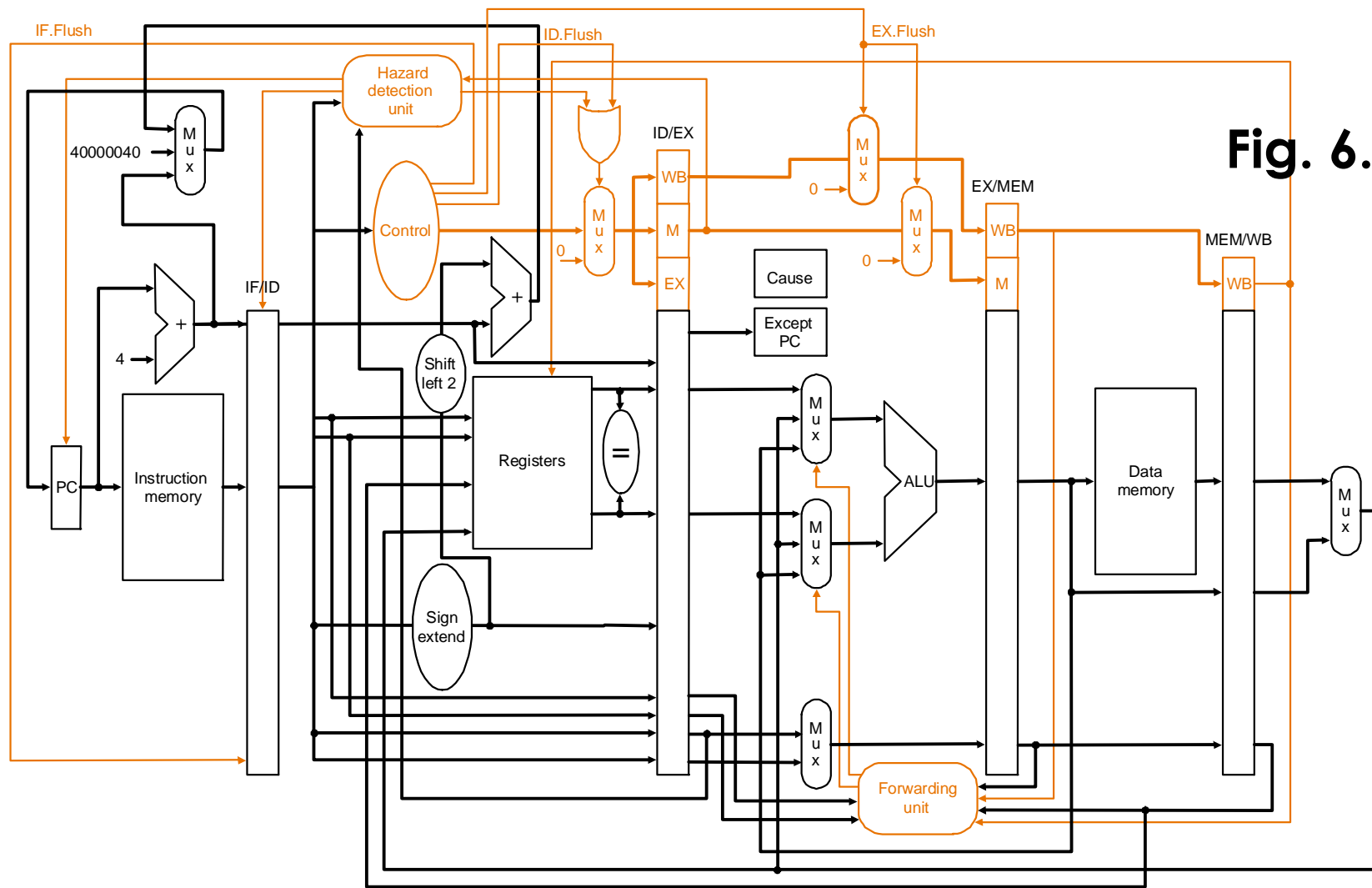


Fig. 6.55

Who caused the exception ?

- 5 instructions executing in 5 stage pipeline
- Who caused the exception? Need to know in which stage an exception can occur => help determine cause

<i>Stage</i>	<i>Problem interrupts occurring</i>
IF	Page fault; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault; misaligned memory access; memory error; mem-protection violation;

When to Serve?

- Who to serve first, if multiple interrupts at the same time?
 - Multiple interrupts: use priority hardware to choose the earliest instruction to interrupt
 - External interrupts: flexible in when to interrupt

Outline

- An overview of pipelining
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards
- Exceptions
- Superscalar and dynamic pipelining

Instruction Level Parallelism

How to increase the potential amount of ILP:

- Increase the depth of the pipeline to overlap more instructions
 - super-pipeline
- Launch multiple instructions
 - Static multiple issue (decision made by compiler before execution)
 - Dynamic multiple issue (decision made during execution by the processor)

Different Pipelined Designs

□ Pipelining

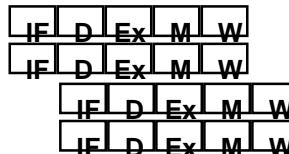


Limitation

Issue rate, FU stalls, FU depth

□ Super-scalar

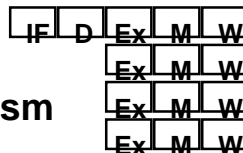
- Issue multiple scalar instructions per cycle



Hazard resolution

□ VLIW (EPIC)

- Each instruction specifies multiple scalar operations
- Compiler determines parallelism



Packing

Static Multiple Issue

- Use compiler to assist with packing instructions and handling hazard
- Very Long Instruction Word (VLIW)
- Explicitly Parallel Instruction Computer (EPIC) (Intel IA-64)

A Static Two-issue Datapath

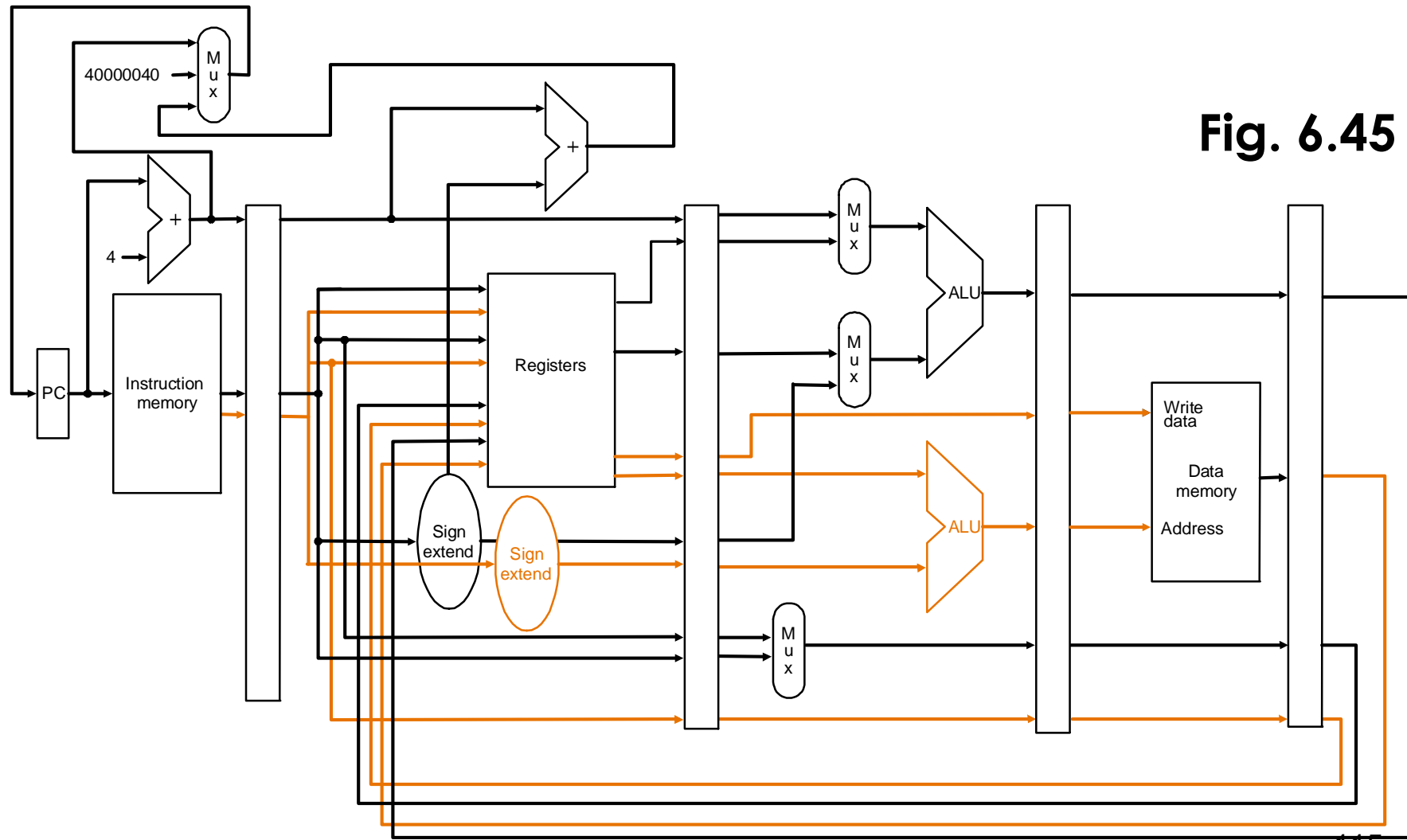


Fig. 6.45

Dynamic Multiple Issue

- The hardware performs the scheduling?
 - hardware tries to find instructions to execute
 - out of order execution is possible
 - speculative execution and dynamic branch prediction
- Superscalar

Superscalar: Three Primary Units

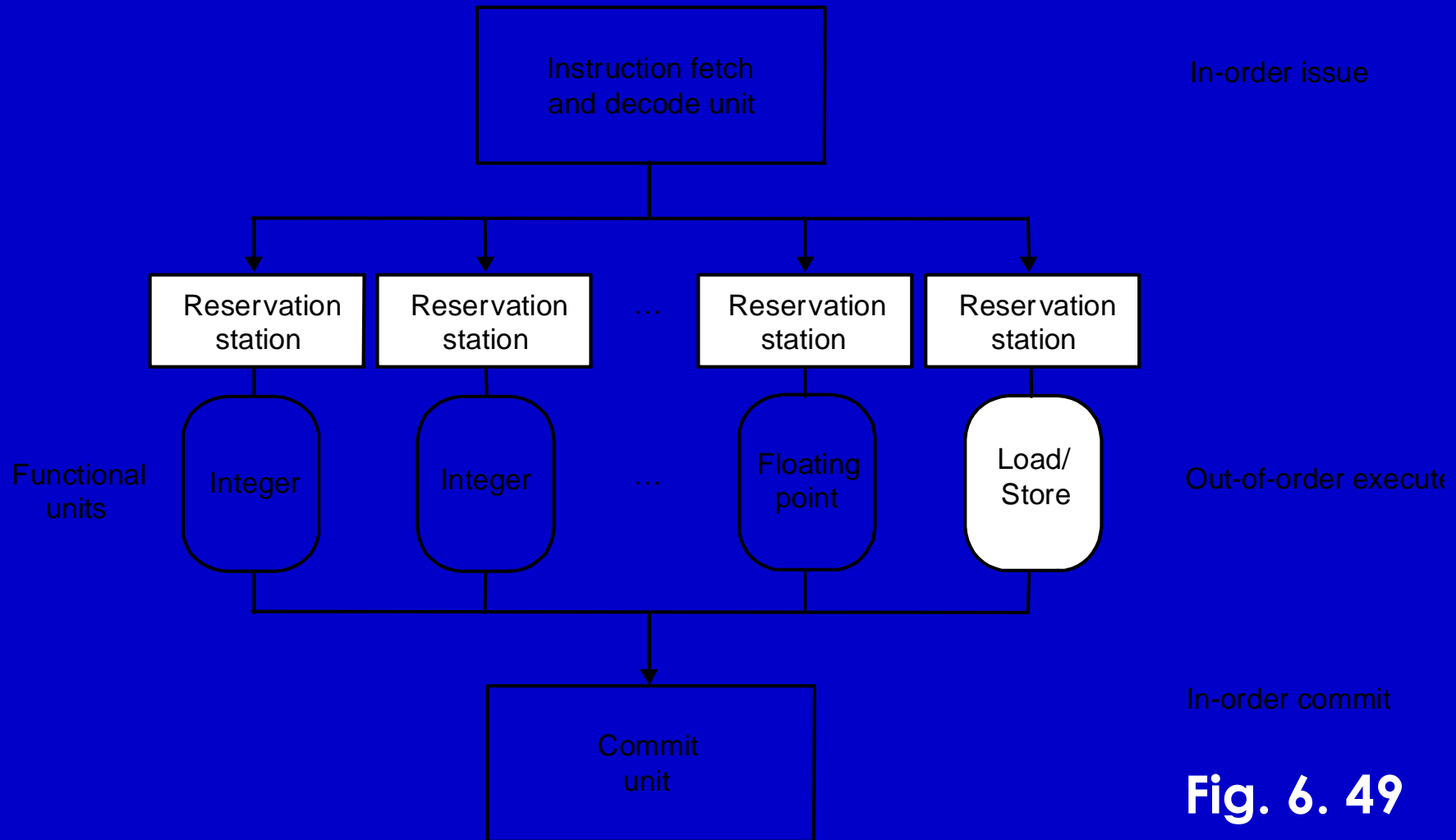
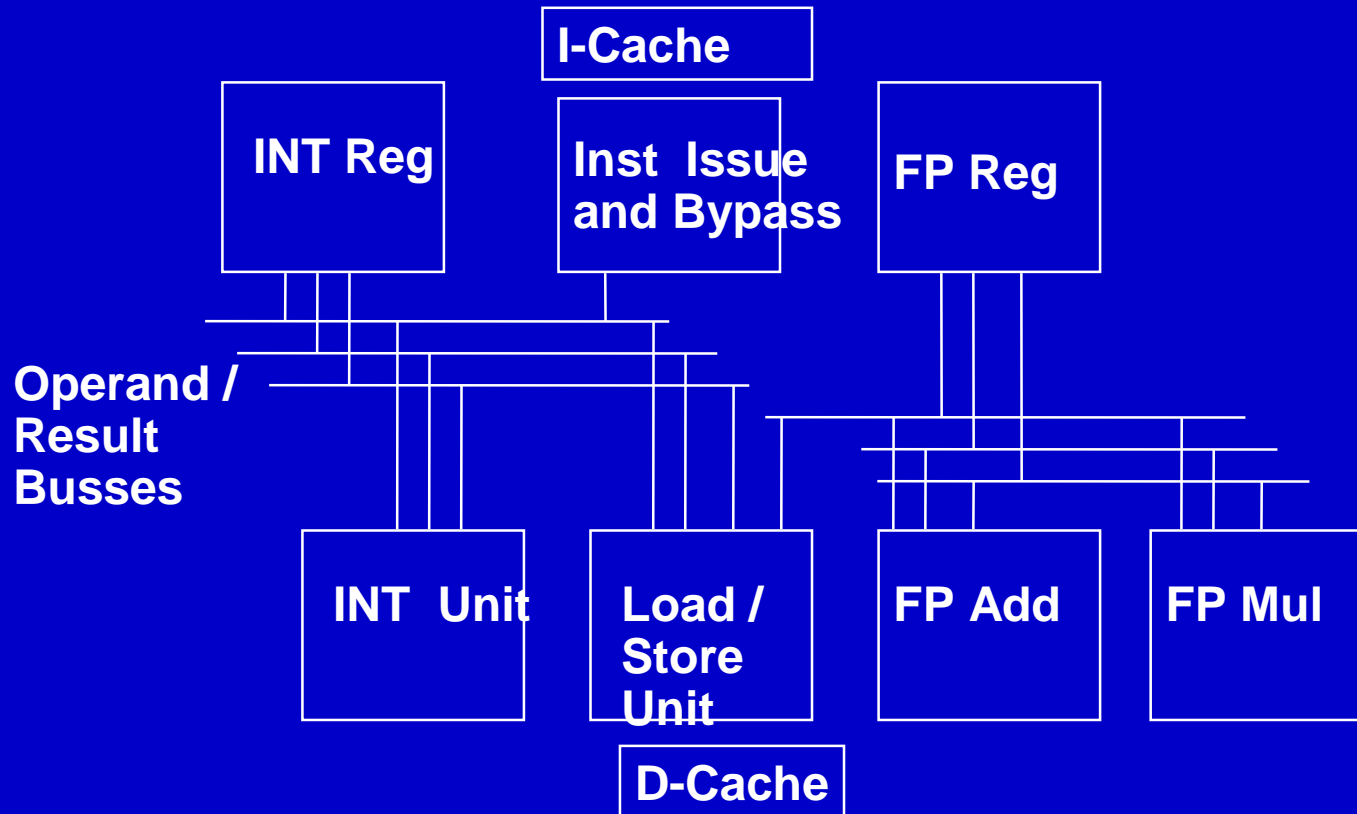


Fig. 6. 49

Simple Superscalar

- Independent INT and FP issue to separate pipelines



Dynamic Scheduling

- All modern processors are very complicated
 - DEC Alpha 21264: 9 stage pipeline, 6 instruction issue
 - PowerPC and Pentium: branch history table
 - Compiler technology important

Summary

- Pipelines pass control information down the pipe just as data moves down pipe
- Forwarding/stalls handled by local control
- Exceptions stop the pipeline
- MIPS instruction set architecture made pipeline visible (delayed branch, delayed load)
- More performance from deeper pipelines, parallelism