

CS4100: 計算機結構

Pipelining

國立清華大學資訊工程學系
一零零零學年度第二學期

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!

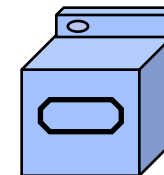
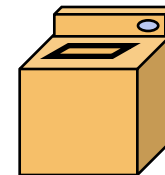
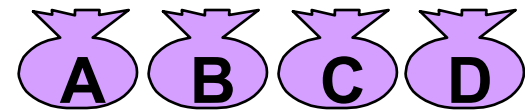
♦ Laundry example:

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

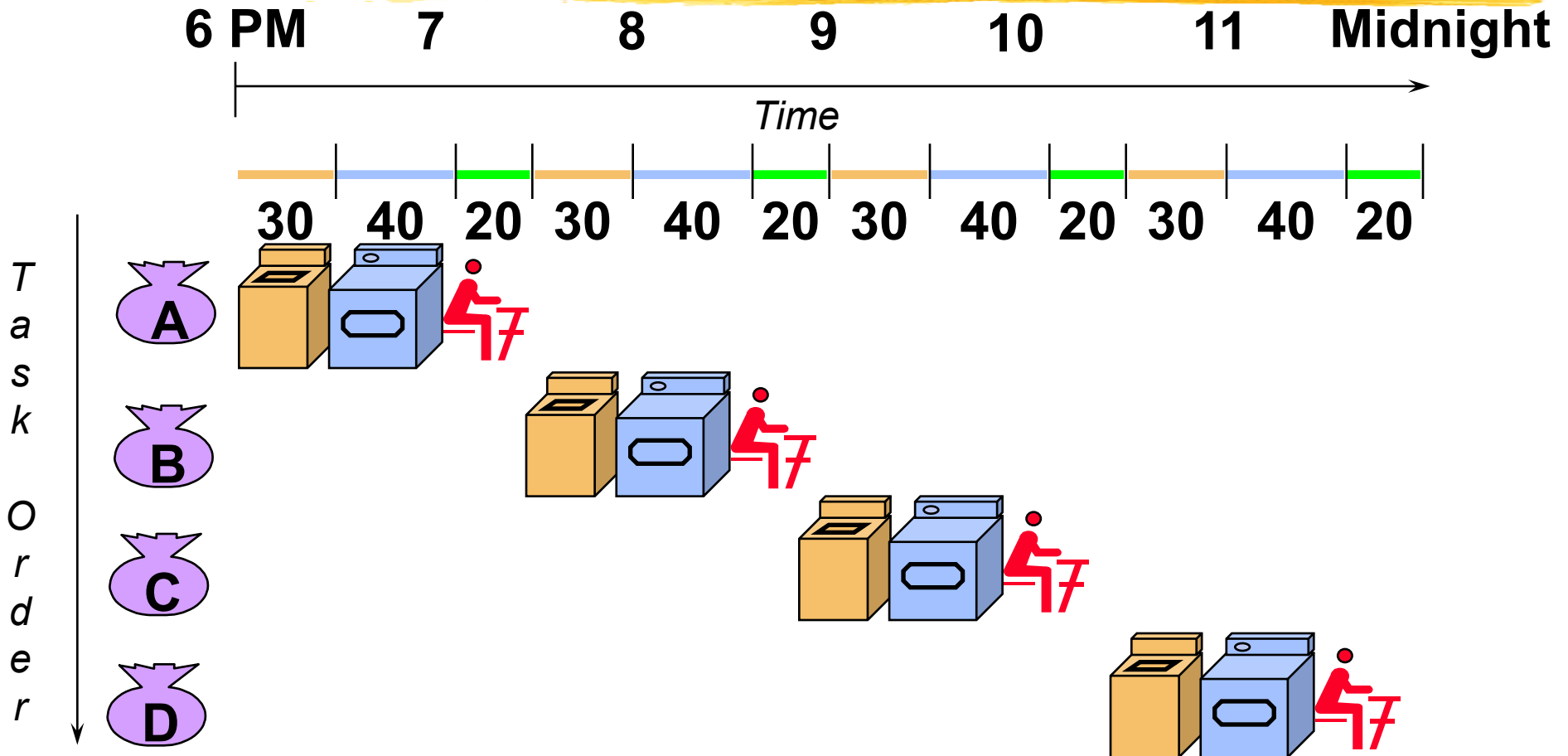
Washer takes 30 minutes

Dryer takes 40 minutes

“Folder” takes 20 minutes



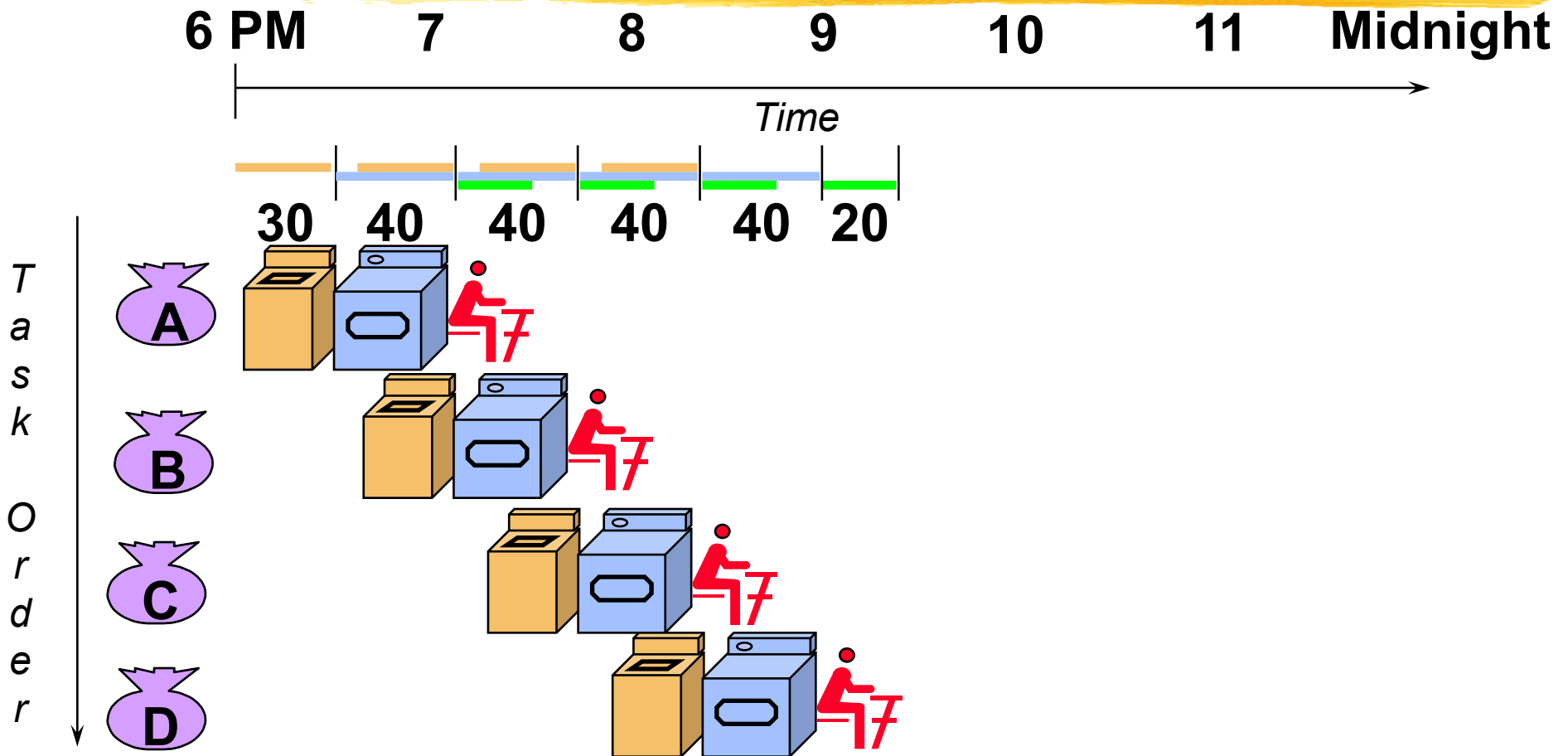
Sequential Laundry



◆ Sequential laundry takes 6 hours for 4 loads

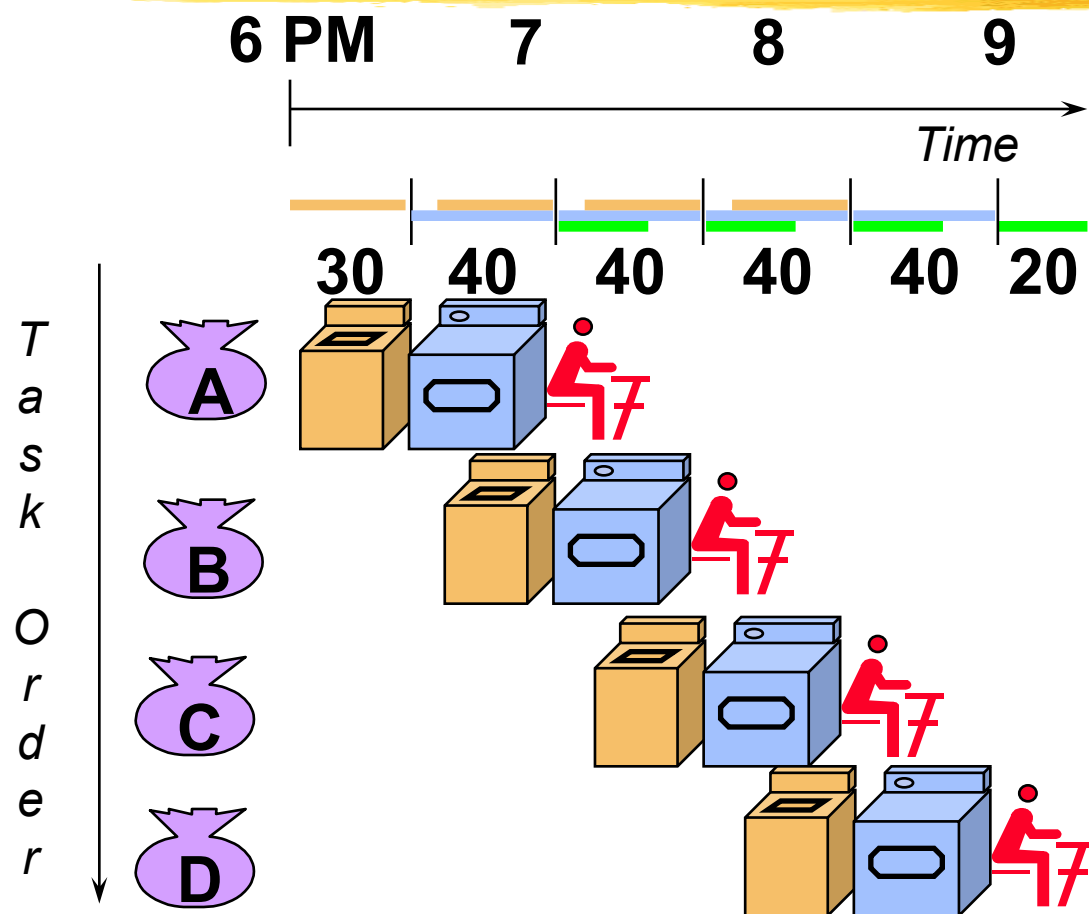
◆ If they learned pipelining, how long would it take?

Pipelined Laundry: Start ASAP



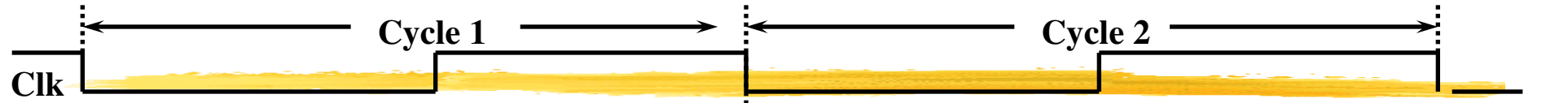
◆ Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons

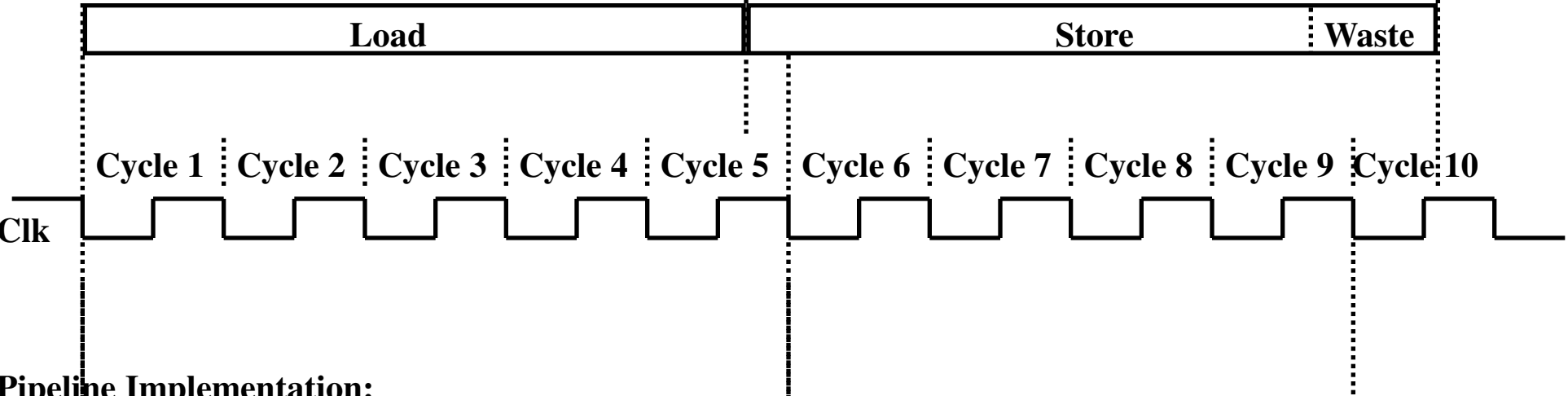


- ◆ Doesn't help **latency** of single task, but **throughput** of entire
- ◆ Pipeline rate limited by **slowest** stage
- ◆ **Multiple** tasks working at same time using different resources
- ◆ Potential speedup = **Number pipe stages**
- ◆ Unbalanced stage length; time to **"fill"** & **"drain"** the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

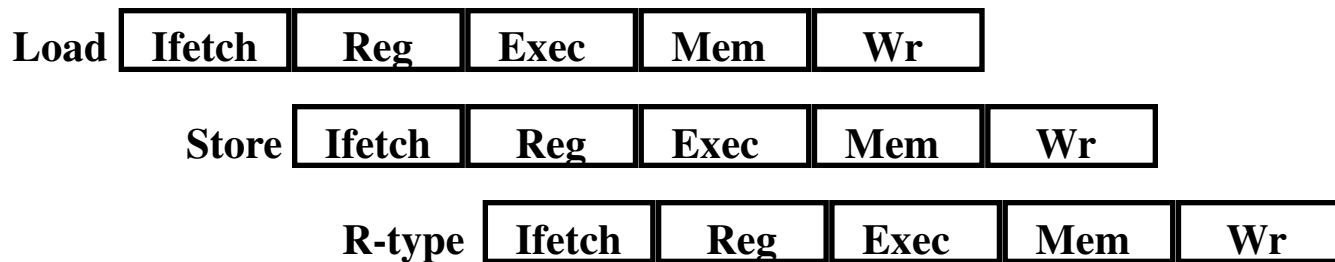
Single cycle vs. Pipeline



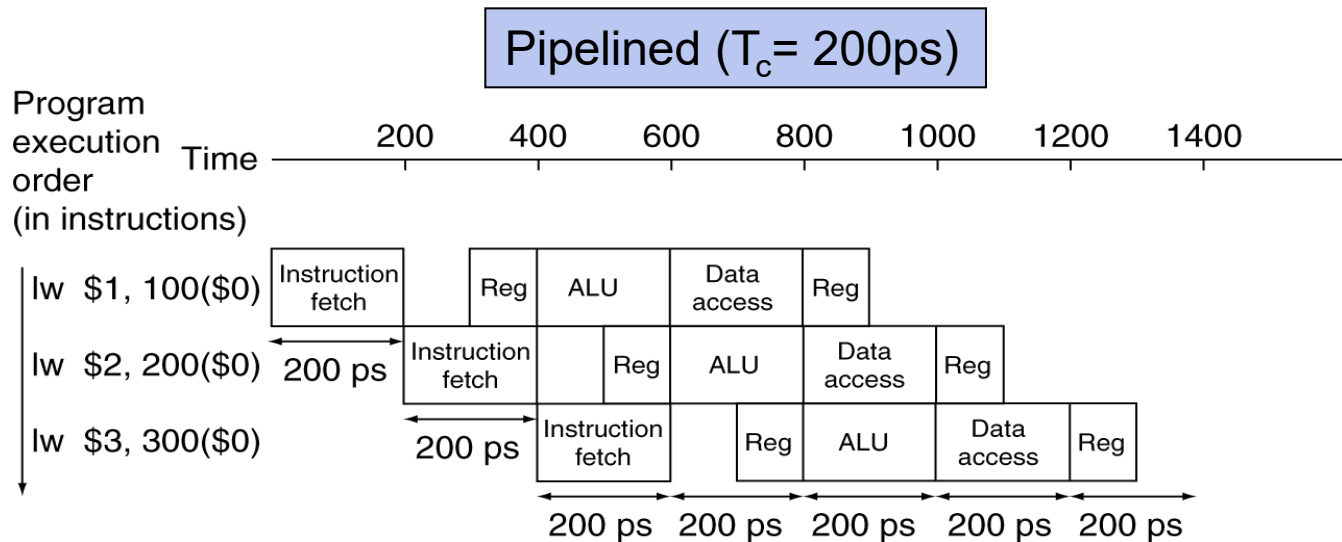
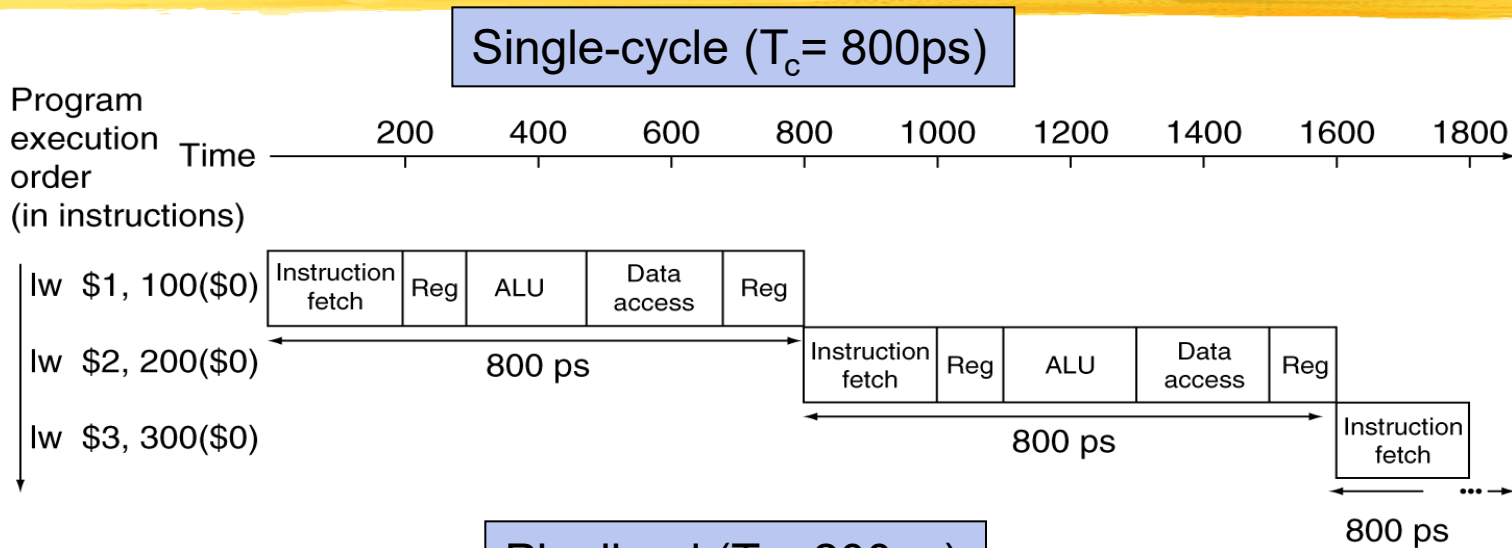
Single Cycle Implementation:



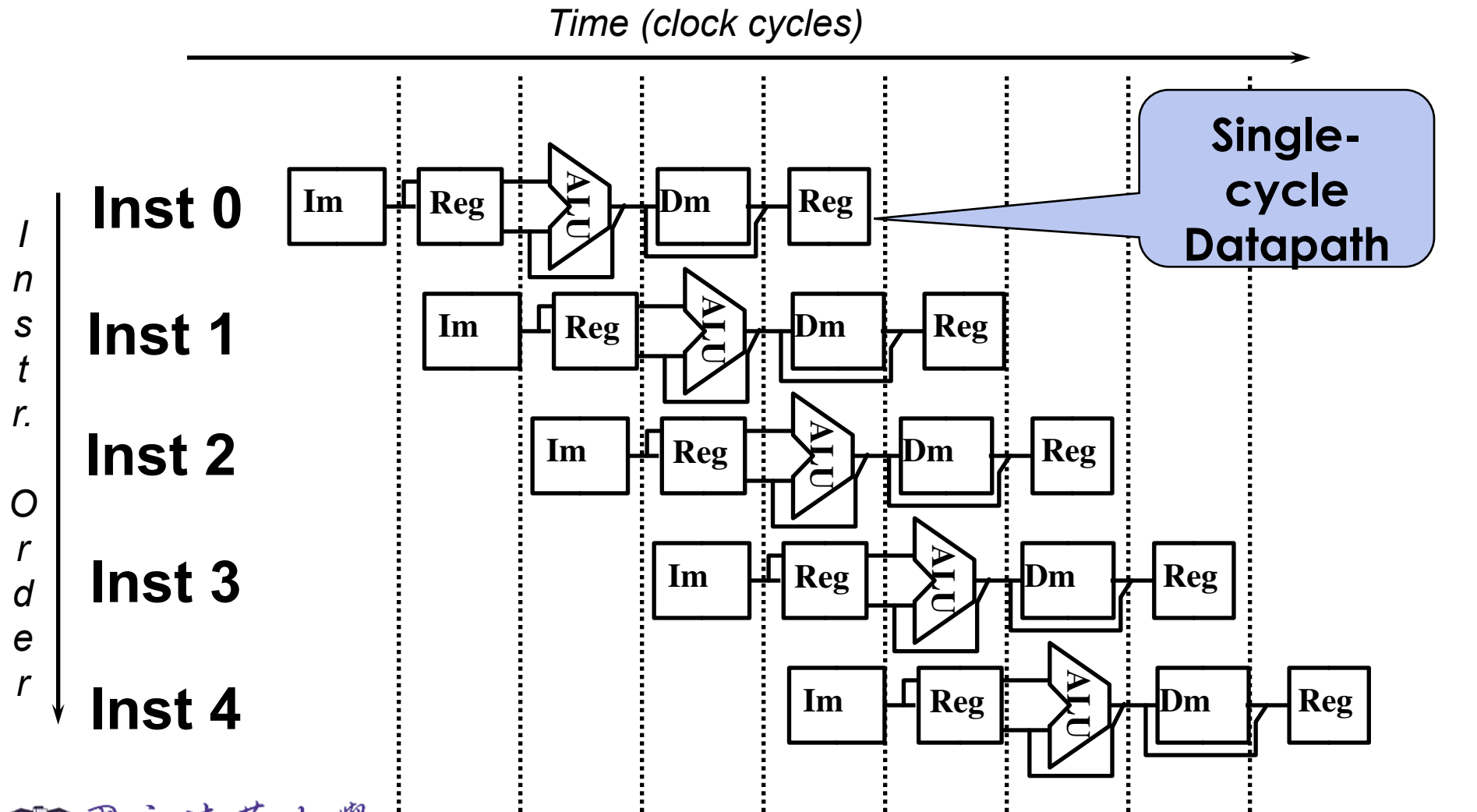
Pipeline Implementation:



Pipeline Performance



Why Pipeline? Because the Resources Are There!



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

Designing a Pipelined Processor

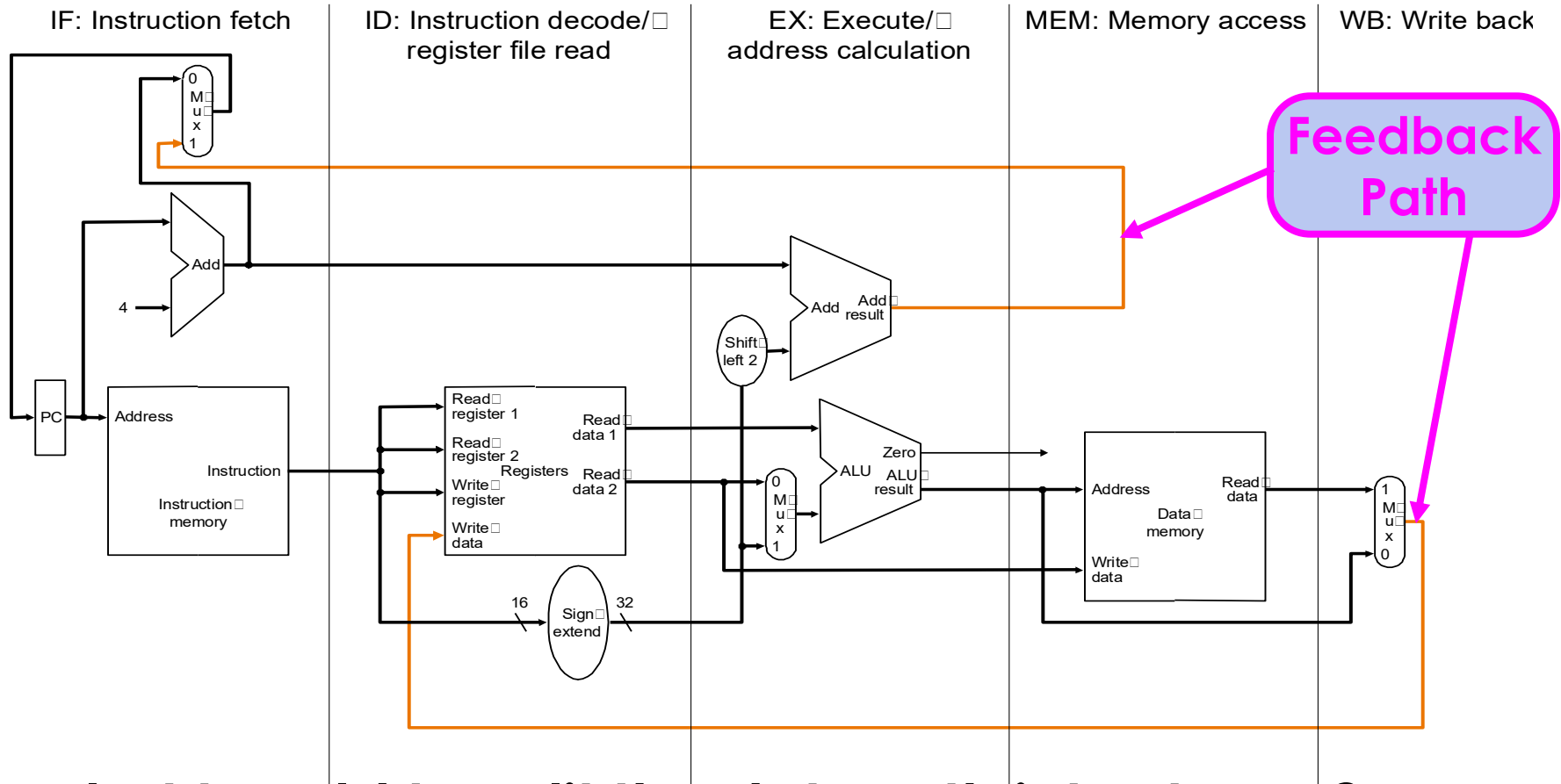
- ◆ Examine the datapath and control diagram
 - Starting with single cycle datapath
 - Single cycle control?
- ◆ Partition datapath into stages:
 - IF (instruction fetch), ID (instruction decode and register file read), EX (execution or address calculation), MEM (data memory access), WB (write back)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

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

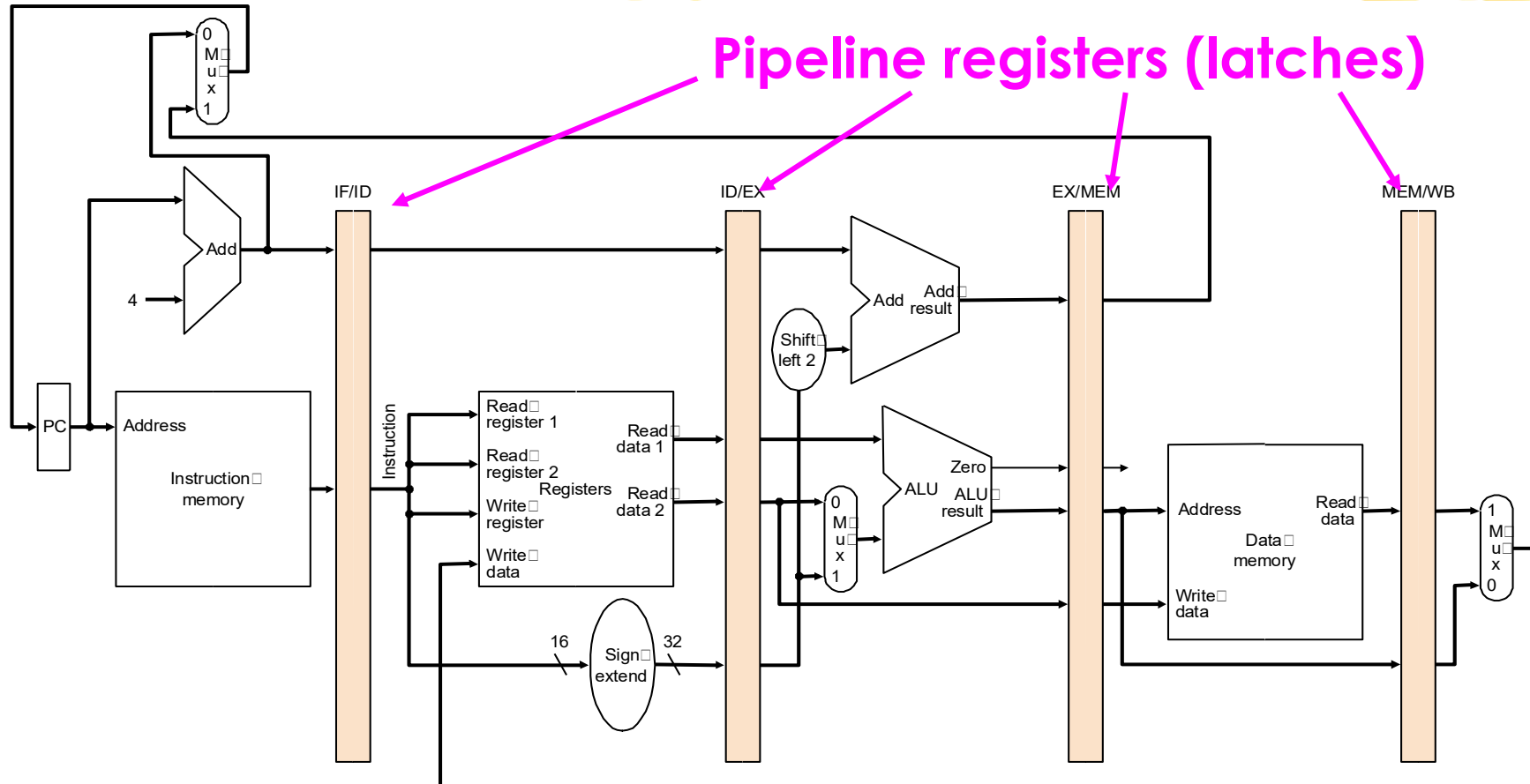
But, use single-cycle datapath ...

Split Single-cycle Datapath



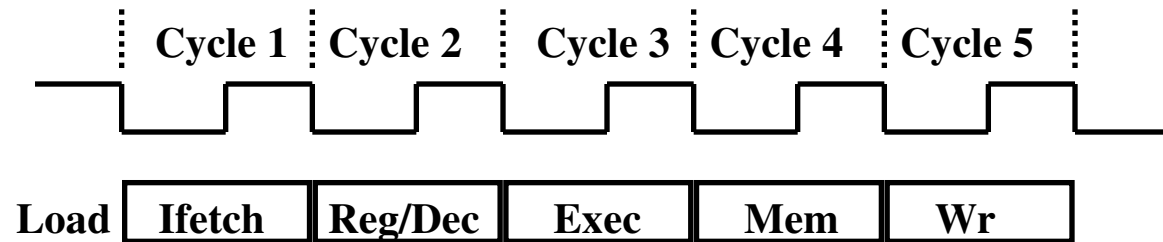
What to add to split the datapath into stages?

Add Pipeline Registers



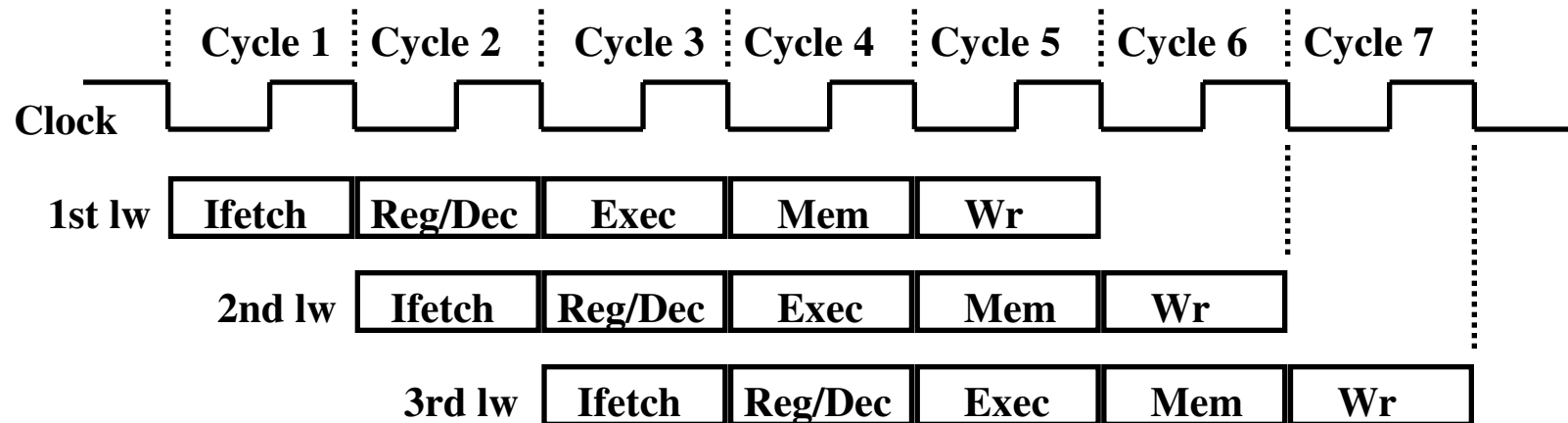
- ◆ **Use registers between stages to carry data and control**

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**

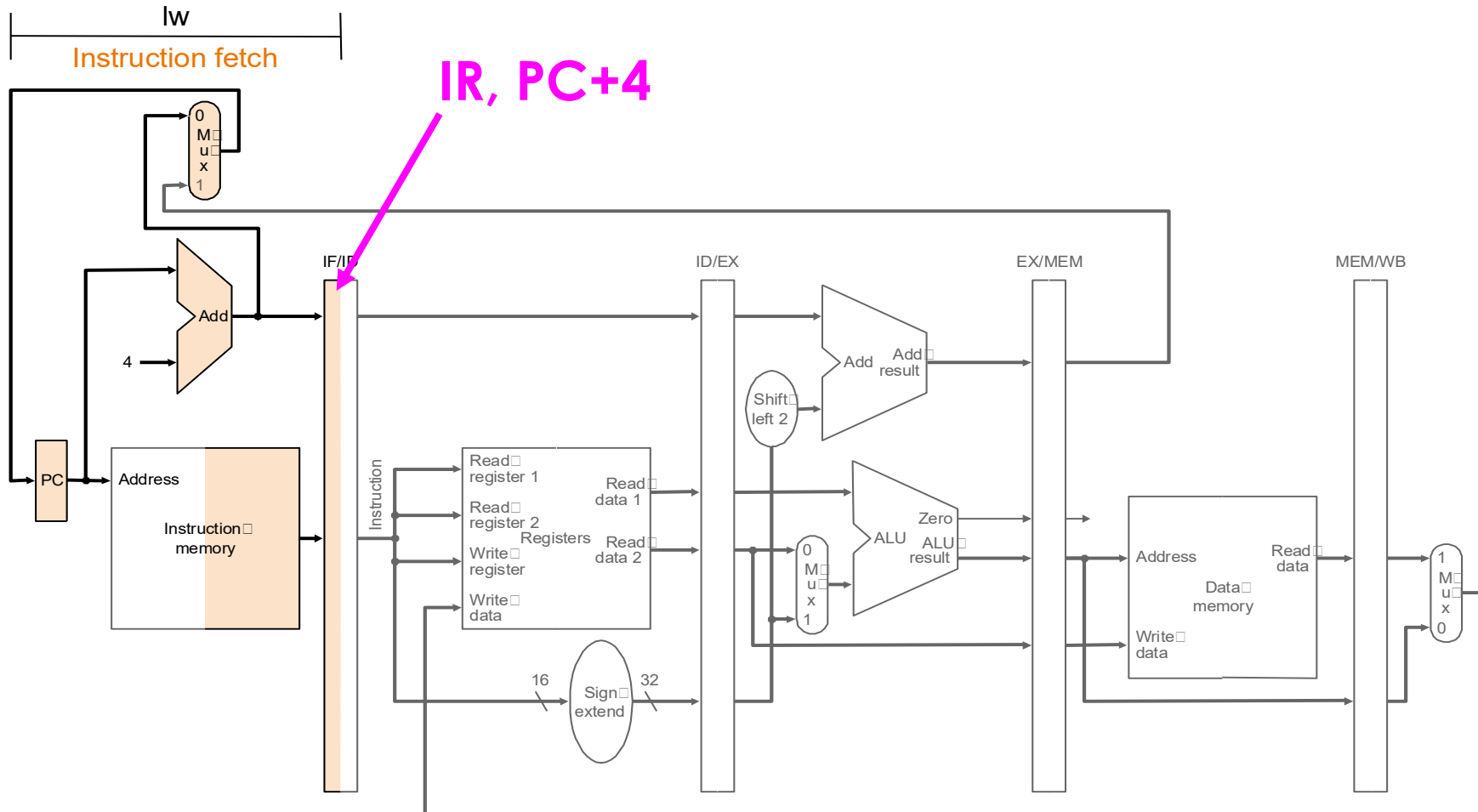
Pipelining load



- ◆ 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

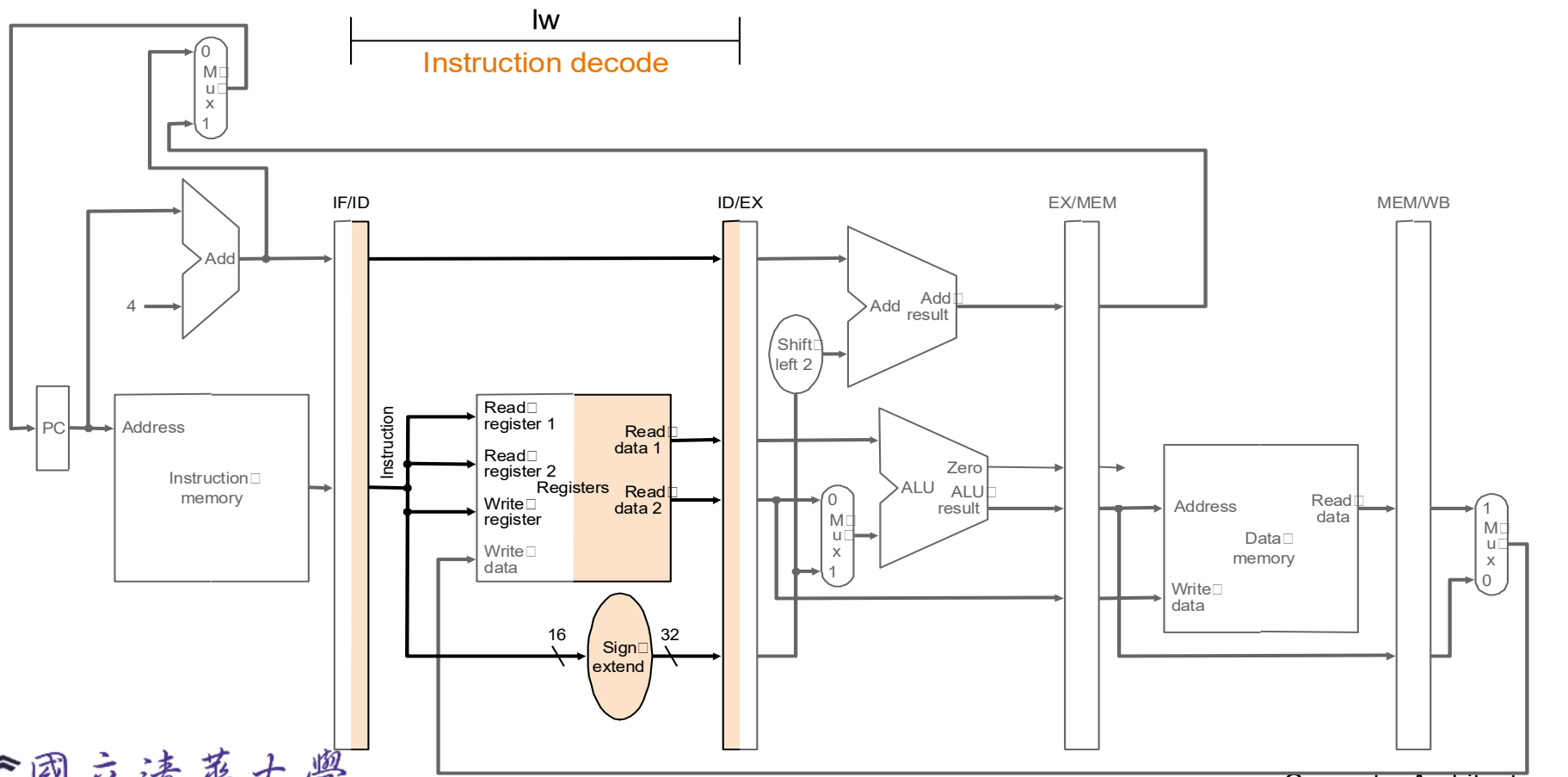
IF Stage of load

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



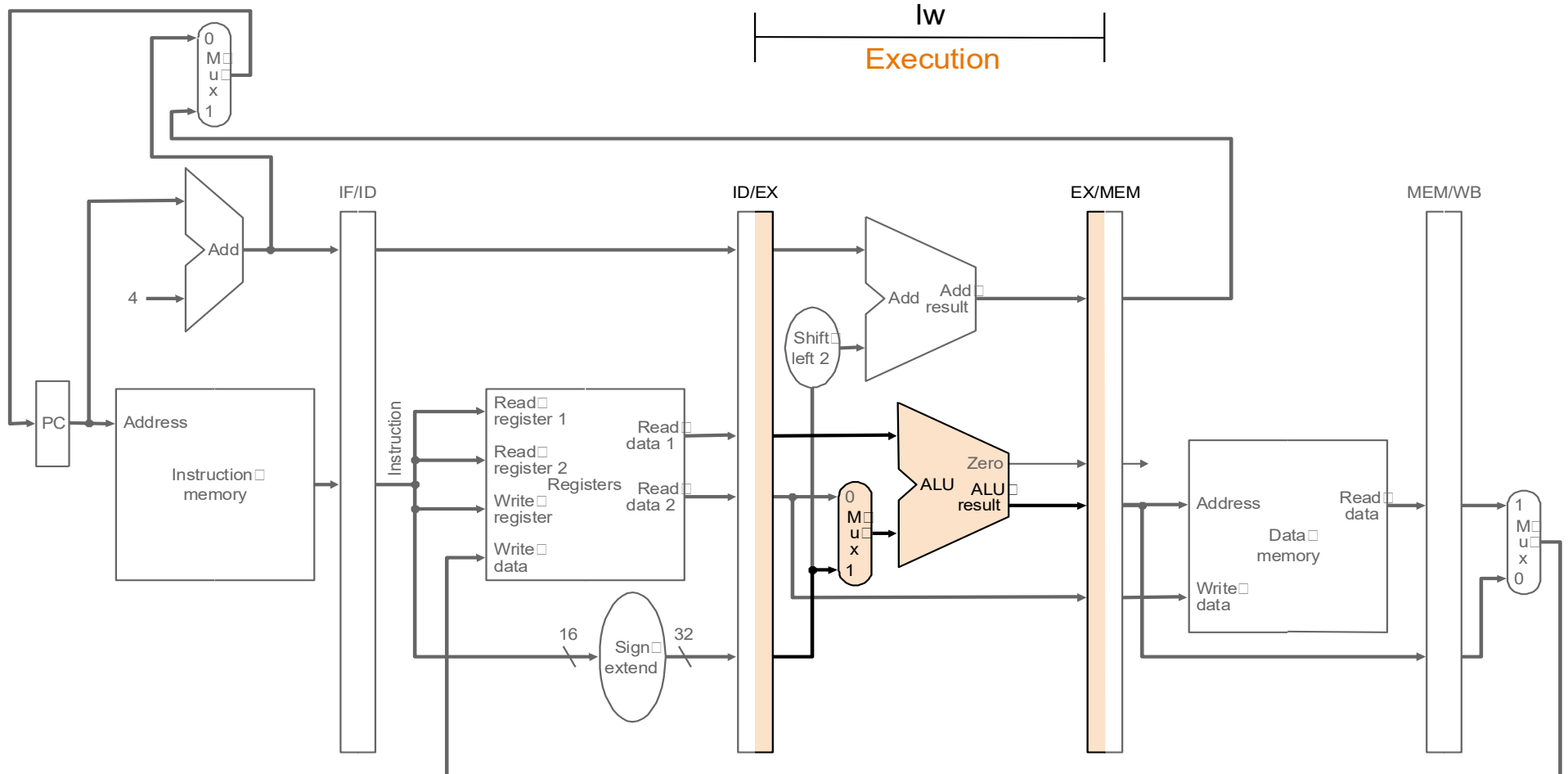
ID Stage of load

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



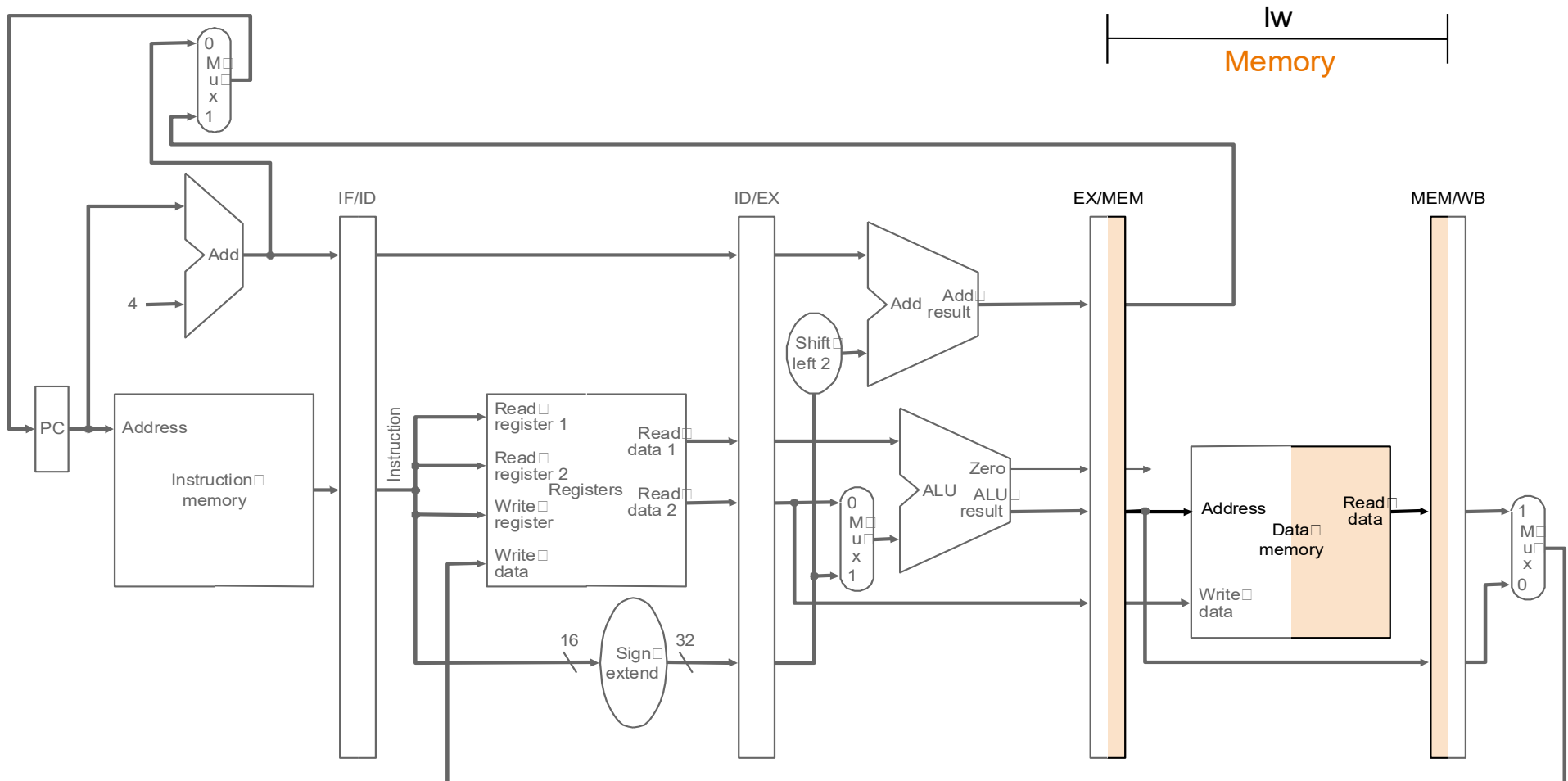
EX Stage of load

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



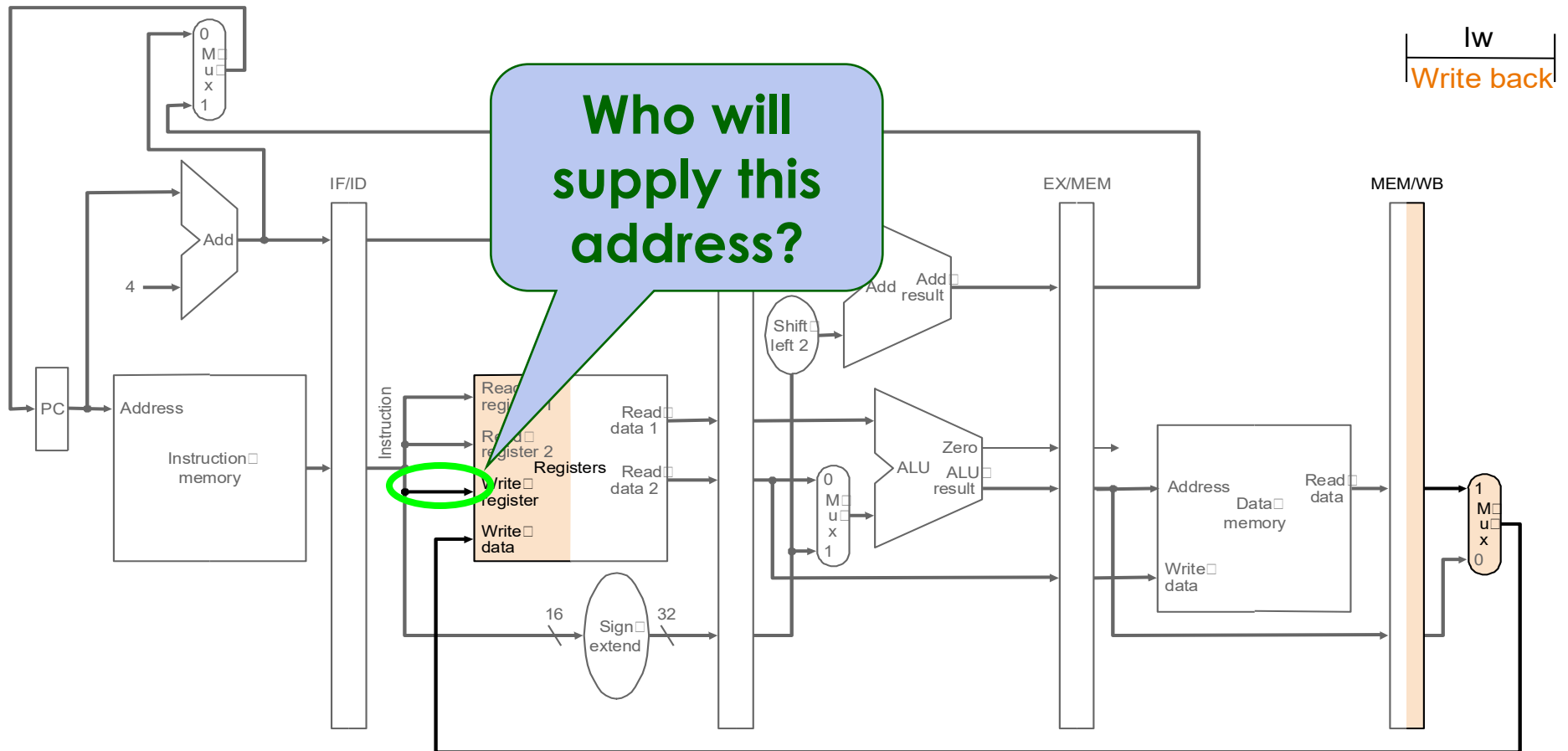
MEM State of load

◆ $MDR = mem[ALUout]$

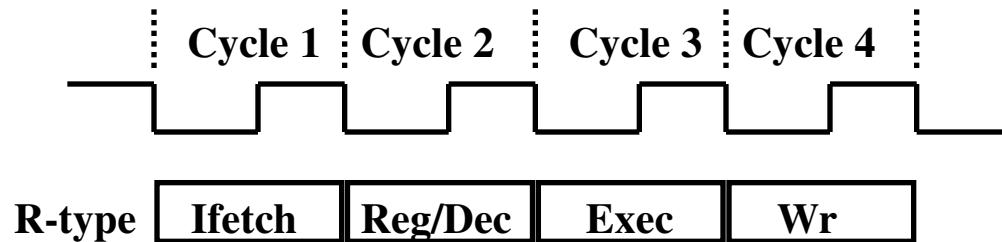


WB Stage of load

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

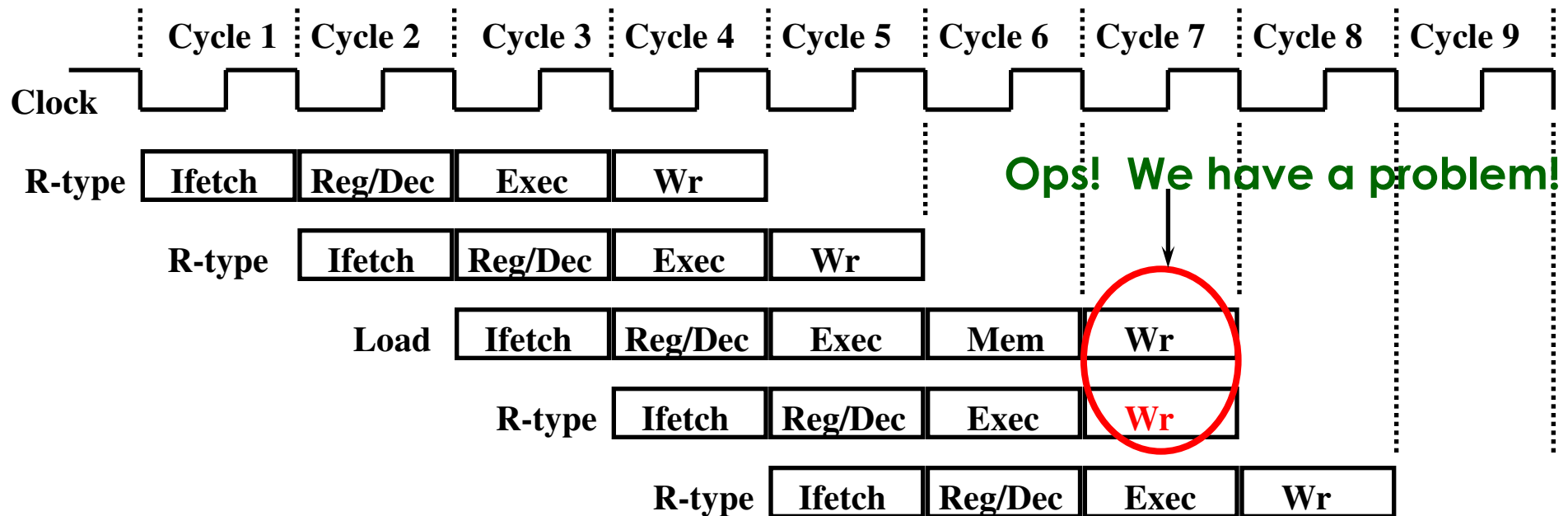


The Four Stages of R-type



- ◆ 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

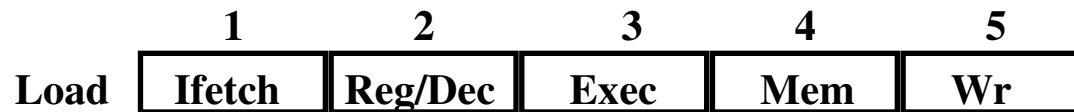


◆ We have a **structural hazard**:

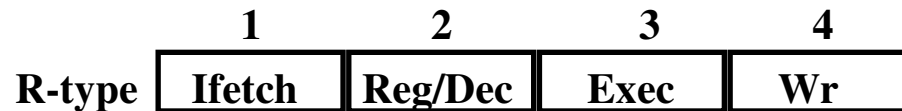
- Two instructions try to write to the register file at the same time!
- Only one write port

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



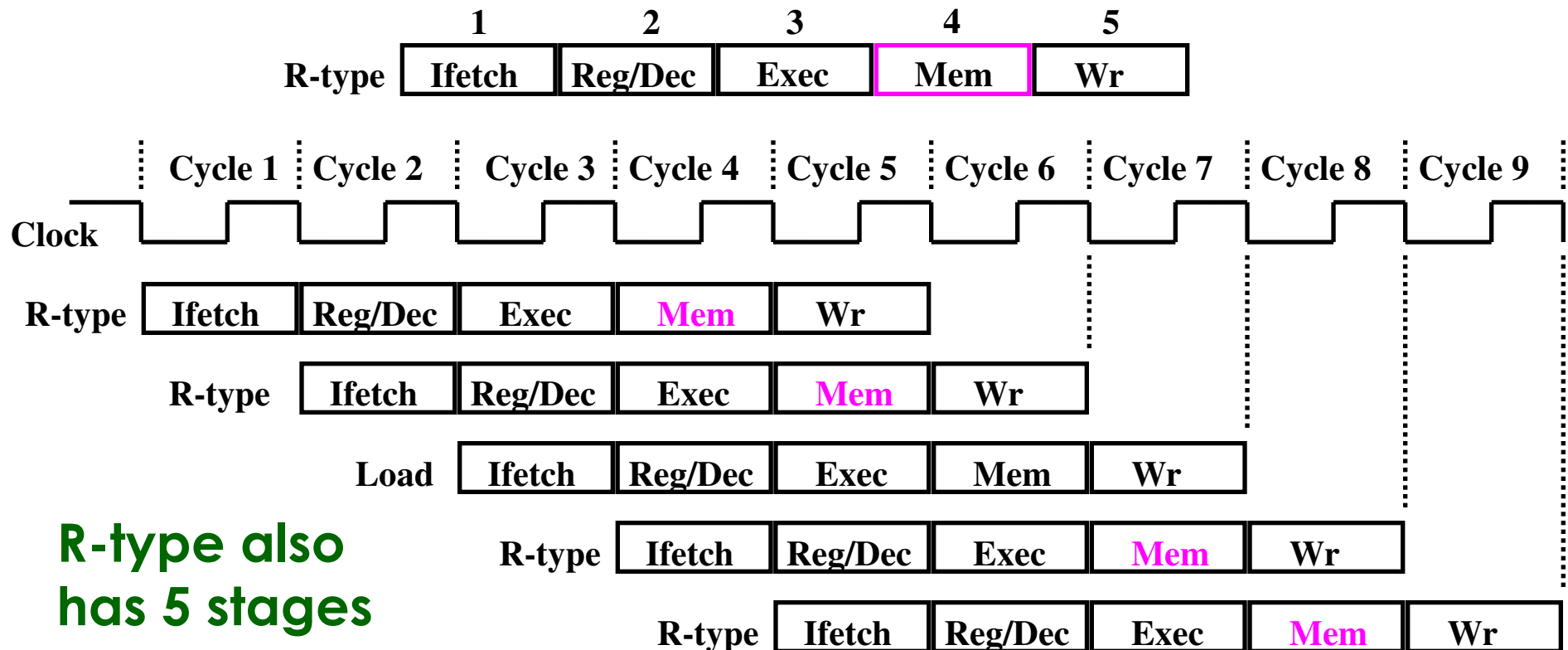
- R-type uses Register File's write port during its **4th** stage



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

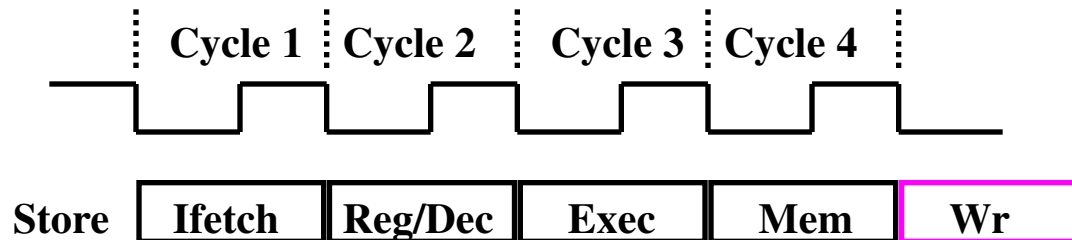
Solution: Delay R-type's Write

- ◆ 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

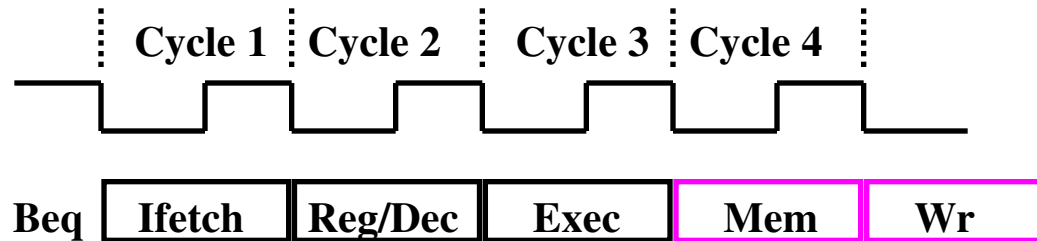


- ◆ IF: fetch the instruction from the Instruction Memory
- ◆ ID: registers fetch and instruction decode
- ◆ EX: calculate the memory address
- ◆ MEM: write the data into the Data Memory

Add an extra stage:

- ◆ WB: NOP

The Three Stages of beq

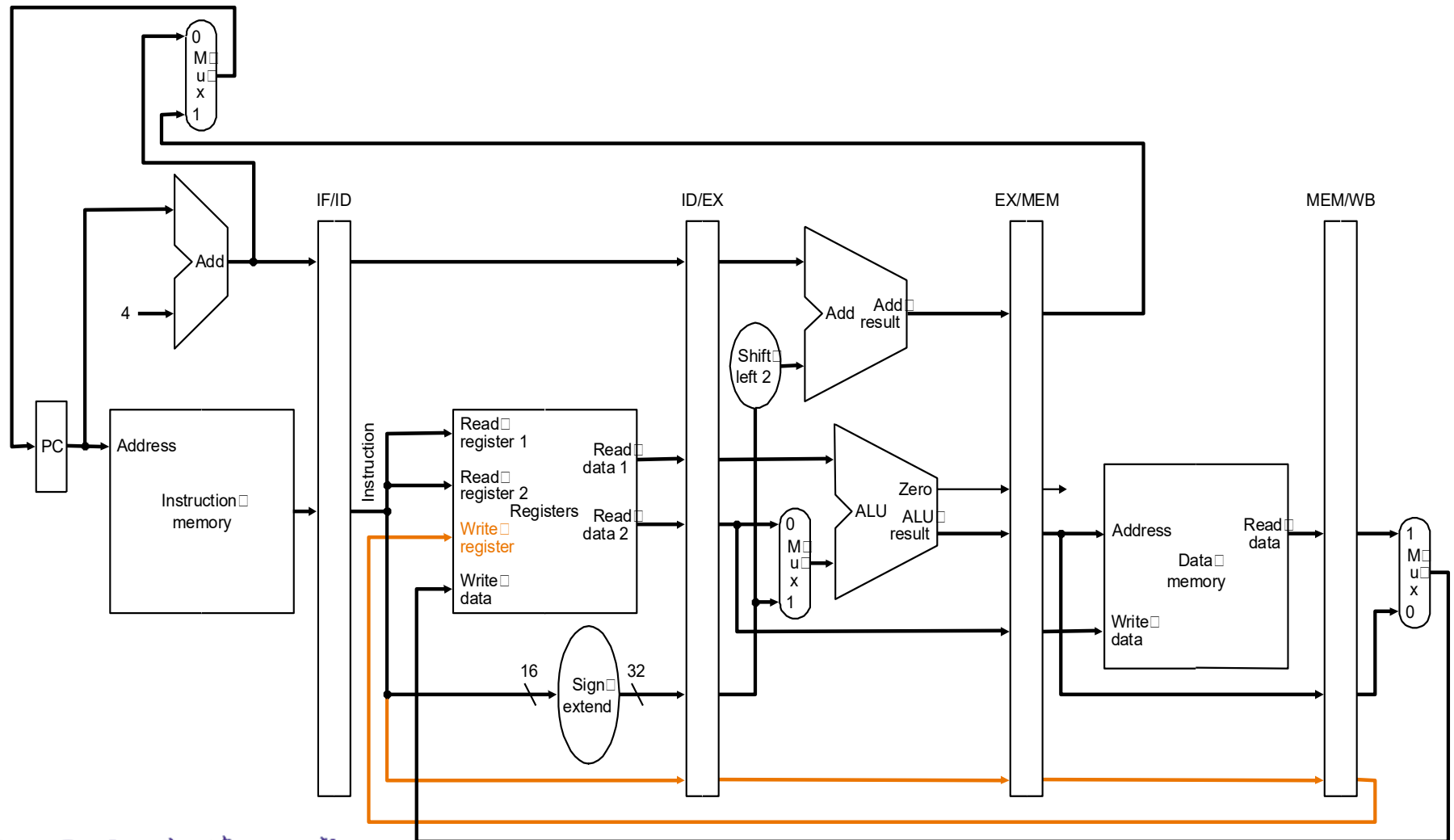


- ◆ IF: fetch the instruction from the Instruction Memory
- ◆ ID: registers fetch and instruction decode
- ◆ EX:
 - compares the two register operand
 - select correct branch target address
 - latch into PC

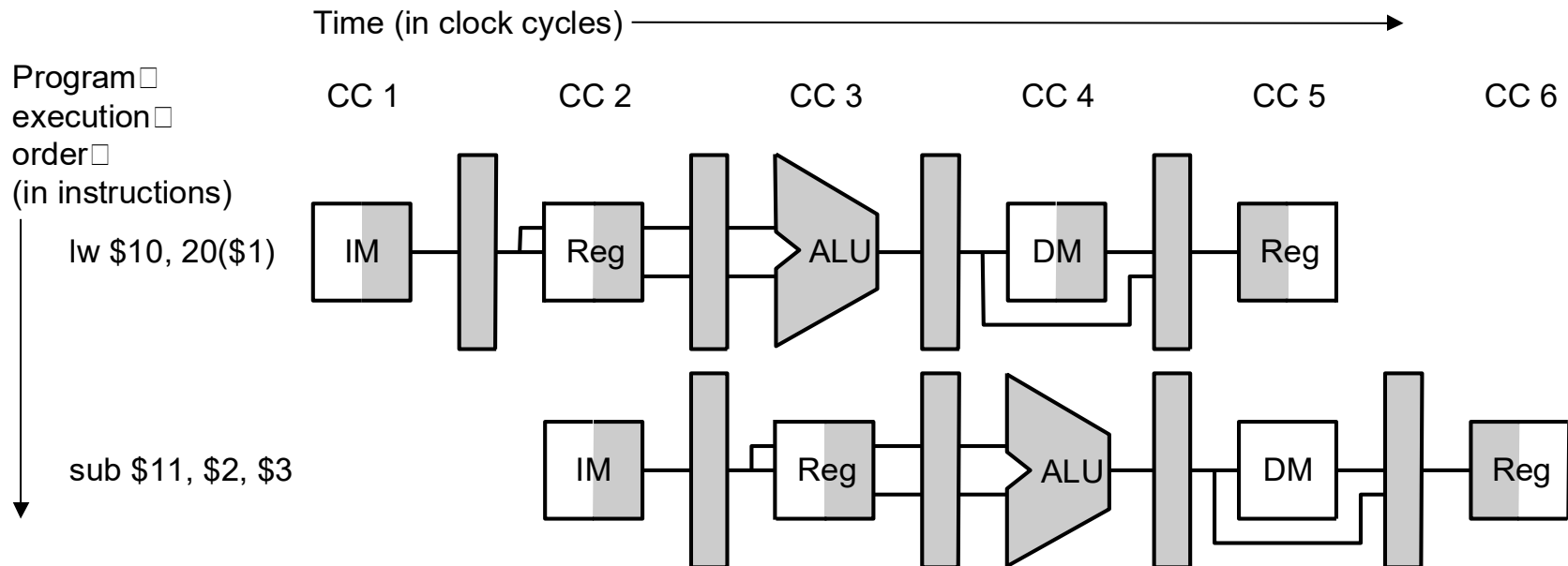
Add two extra stages:

- ◆ MEM: NOP
- ◆ WB: NOP

Pipelined Datapath



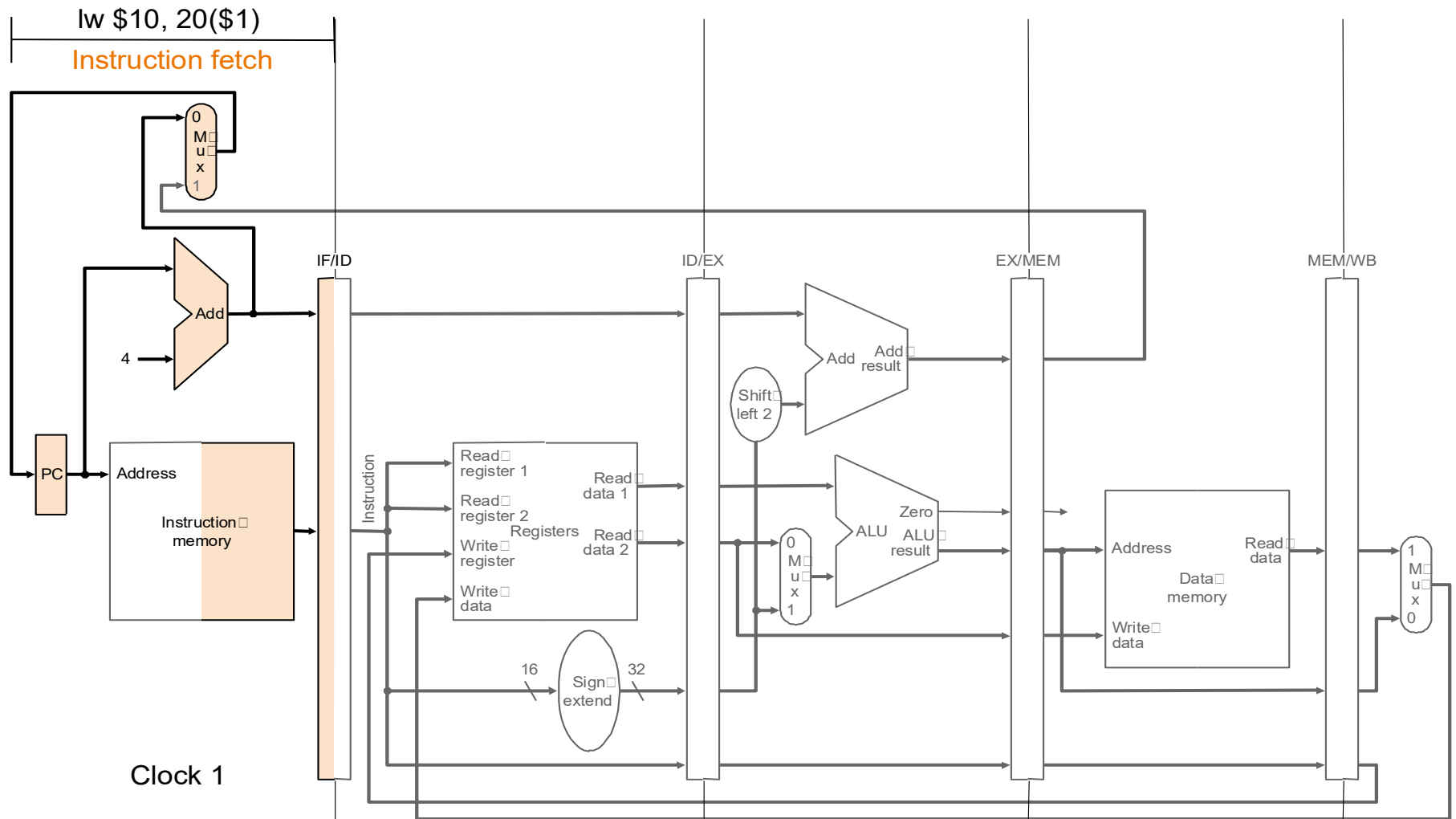
Graphically Representing Pipelines



◆ Can help with answering questions like:

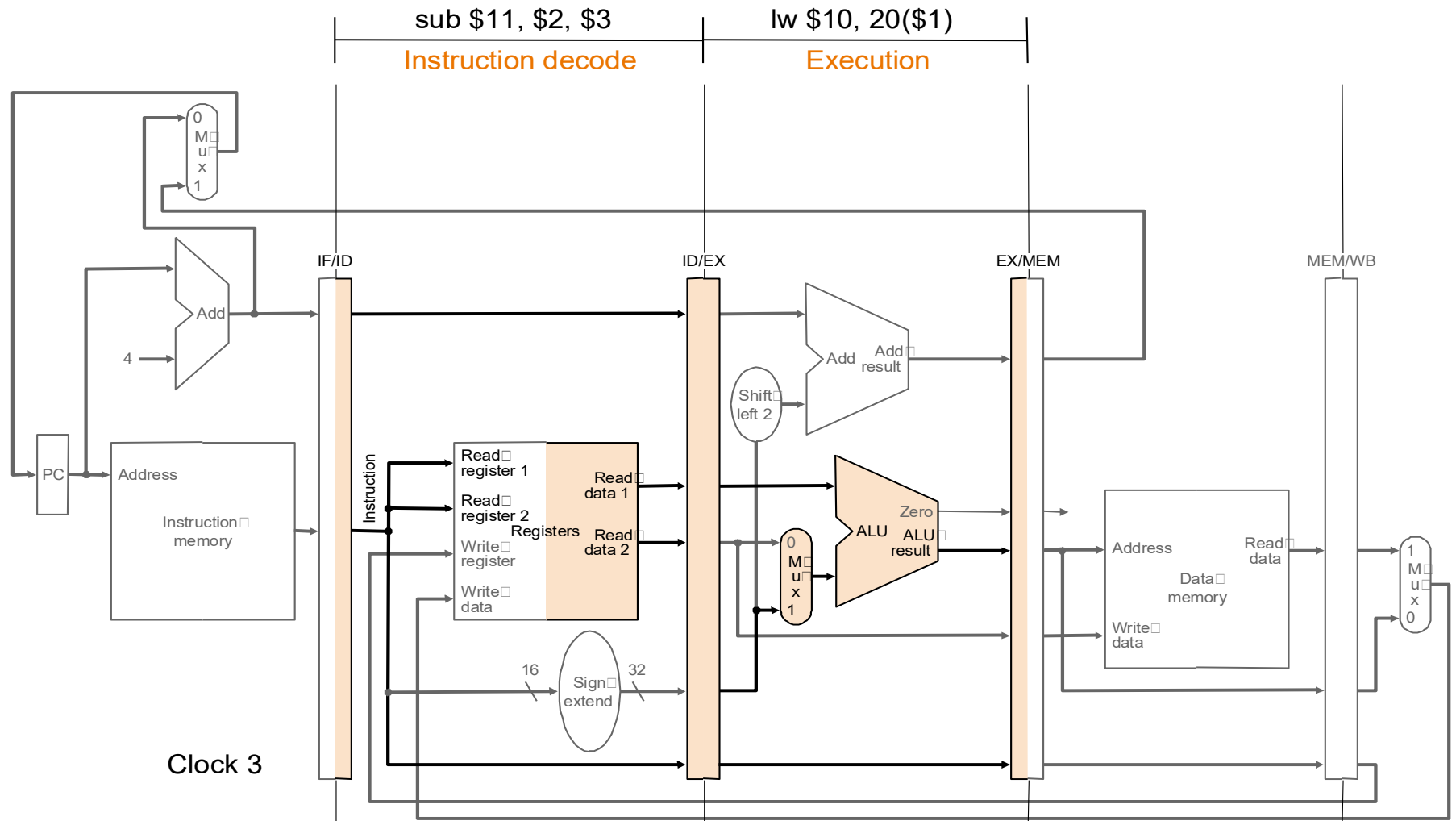
- How many cycles to execute this code?
- What is the ALU doing during cycle 4?
- Help understand datapaths

Example 1: Cycle 1

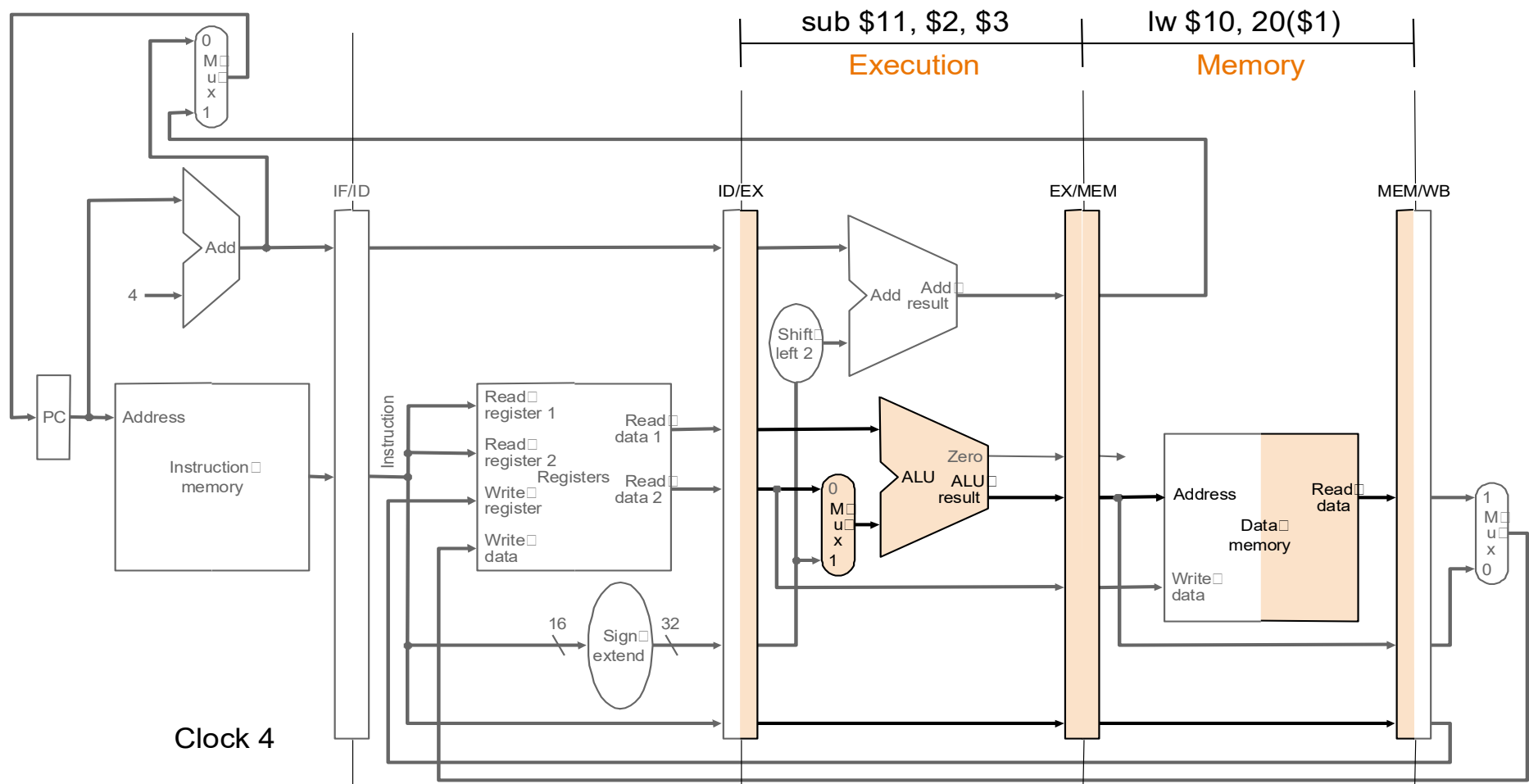




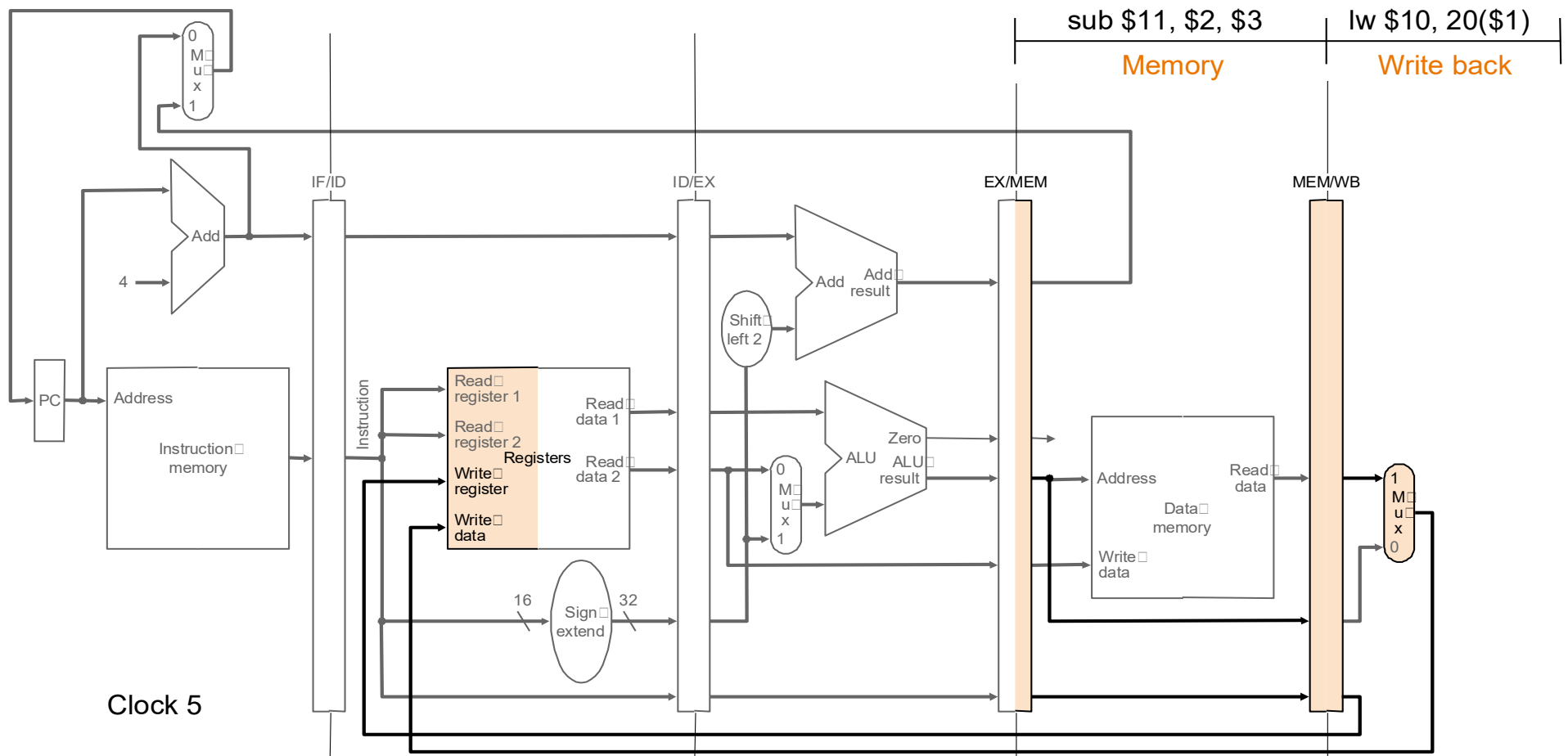
Example 1: Cycle 3



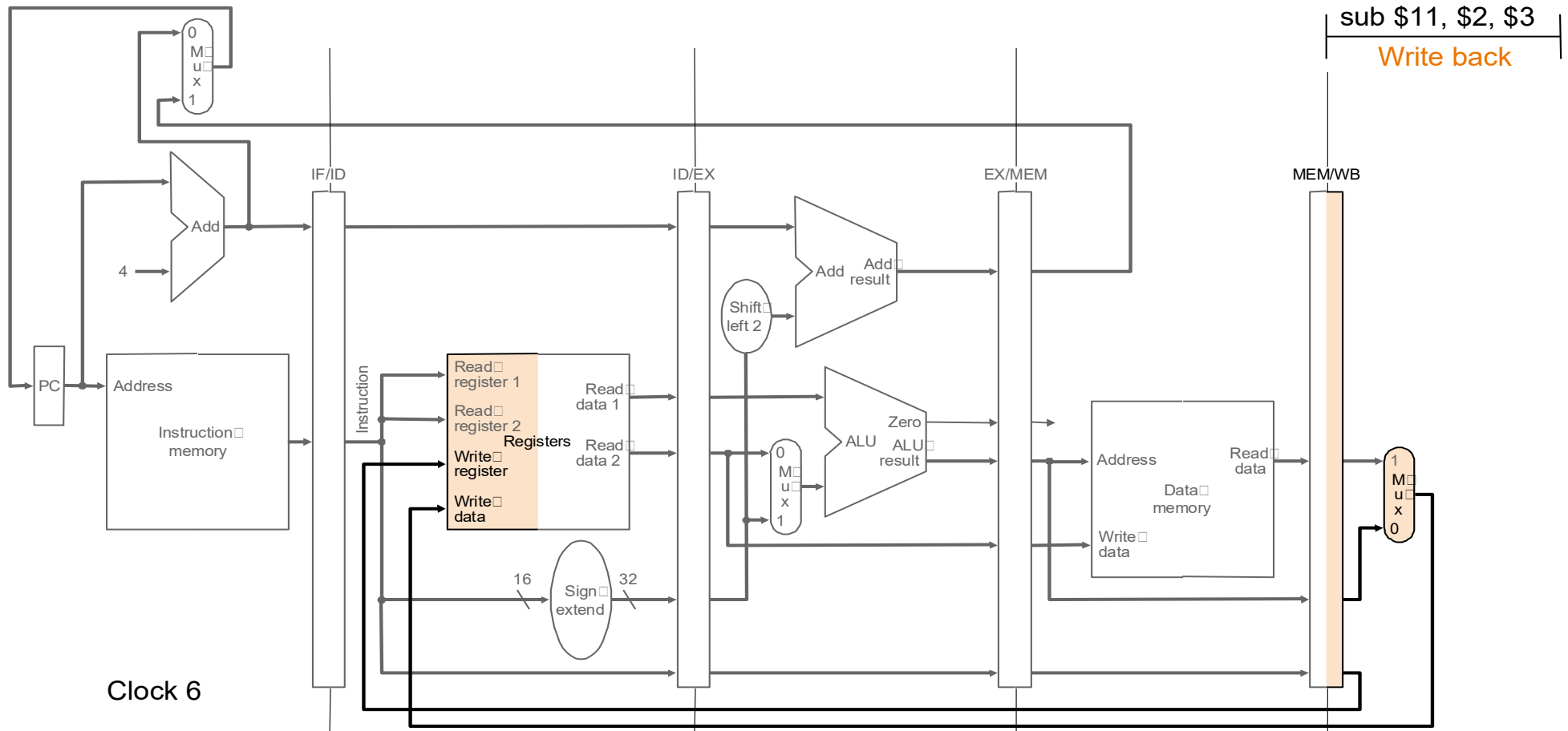
Example 1: Cycle 4



Example 1: Cycle 5



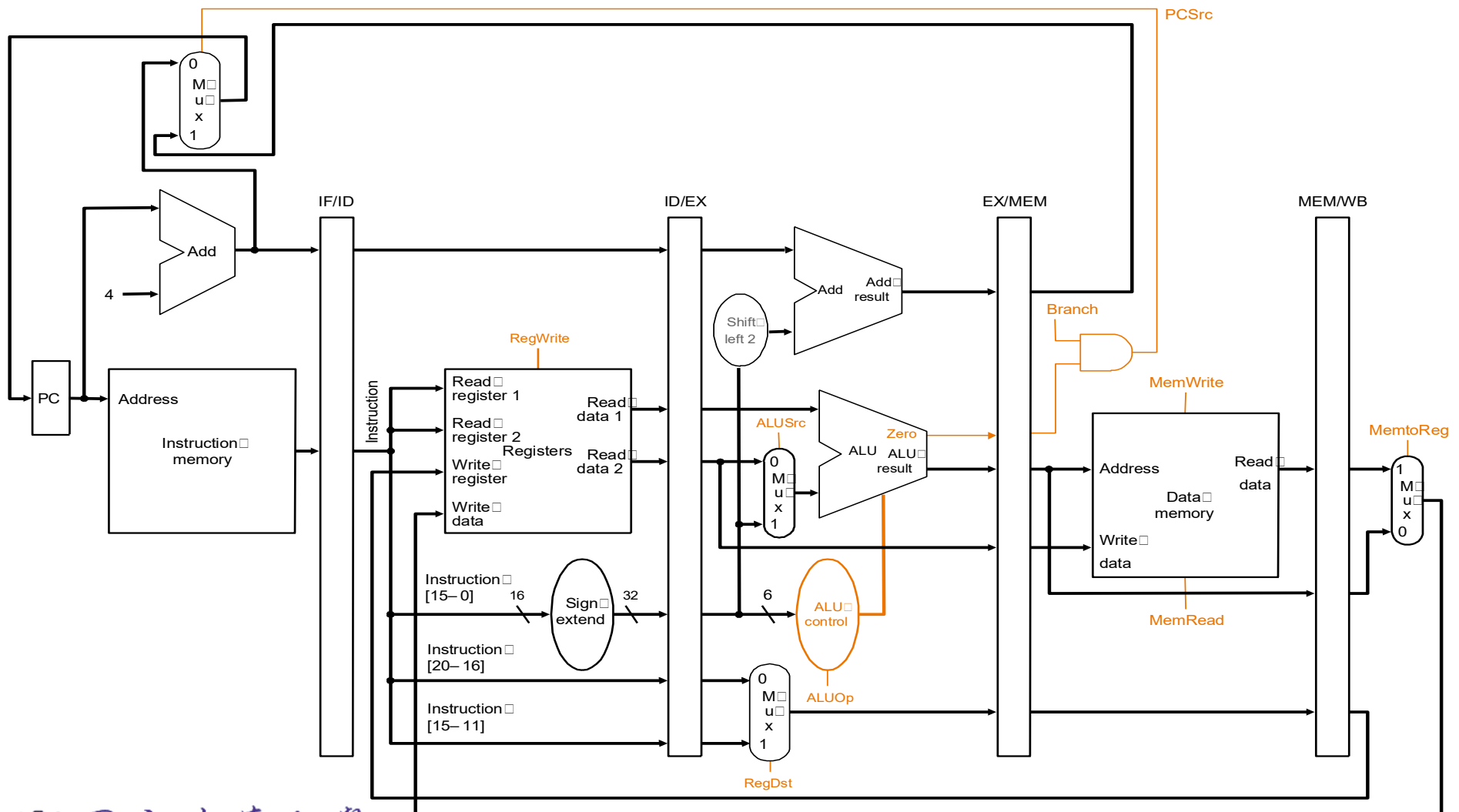
Example 1: Cycle 6



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

Pipeline Control: Control Signals



Group Signals According to Stages

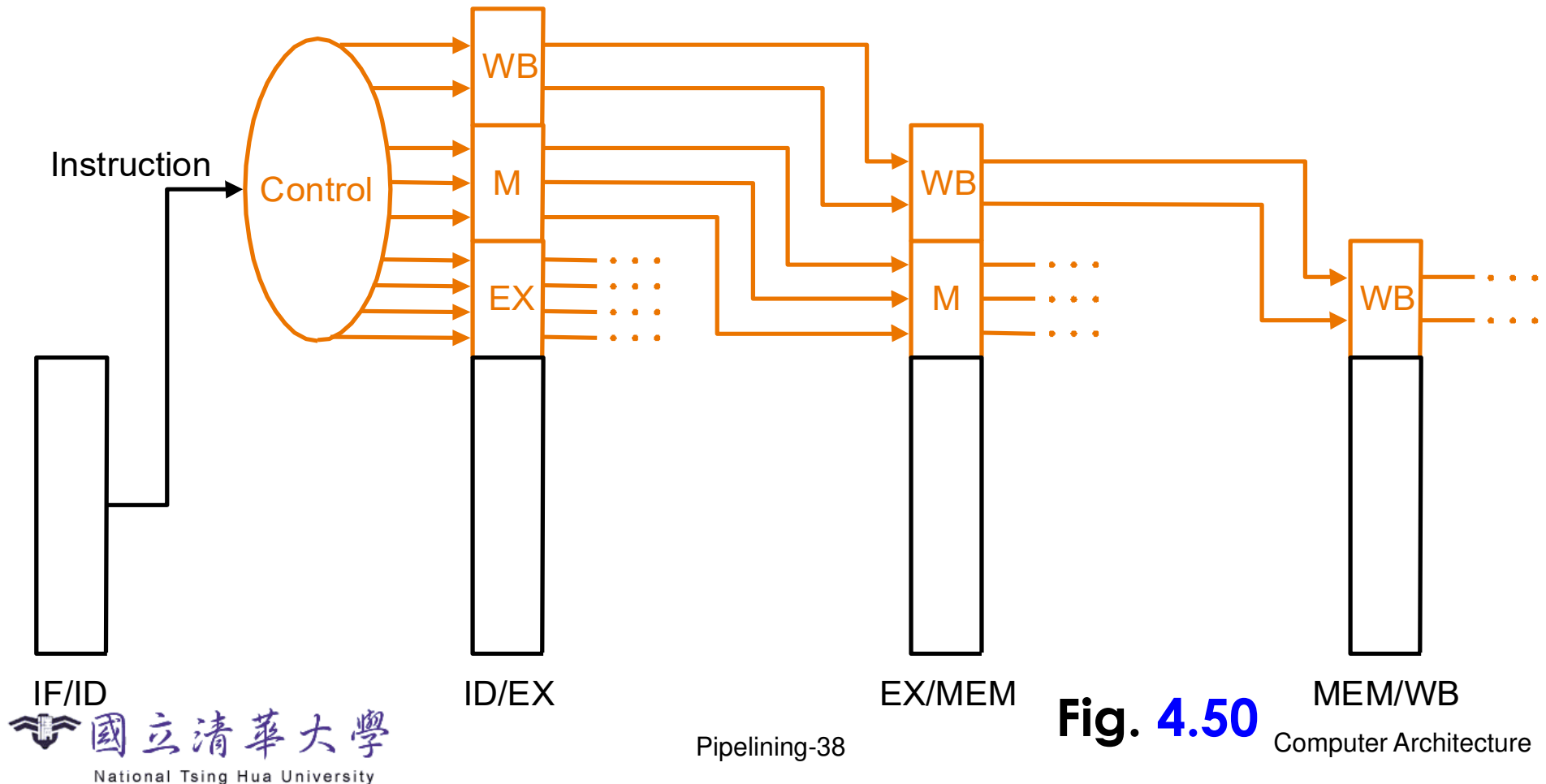
- ◆ Can use control signals of single-cycle CPU

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
1	1	0	0	0	0	0	1	0
0	0	0	1	0	1	0	1	1
X	0	0	1	0	0	1	0	X
X	0	1	0	1	0	0	0	X

Fig. 4.22

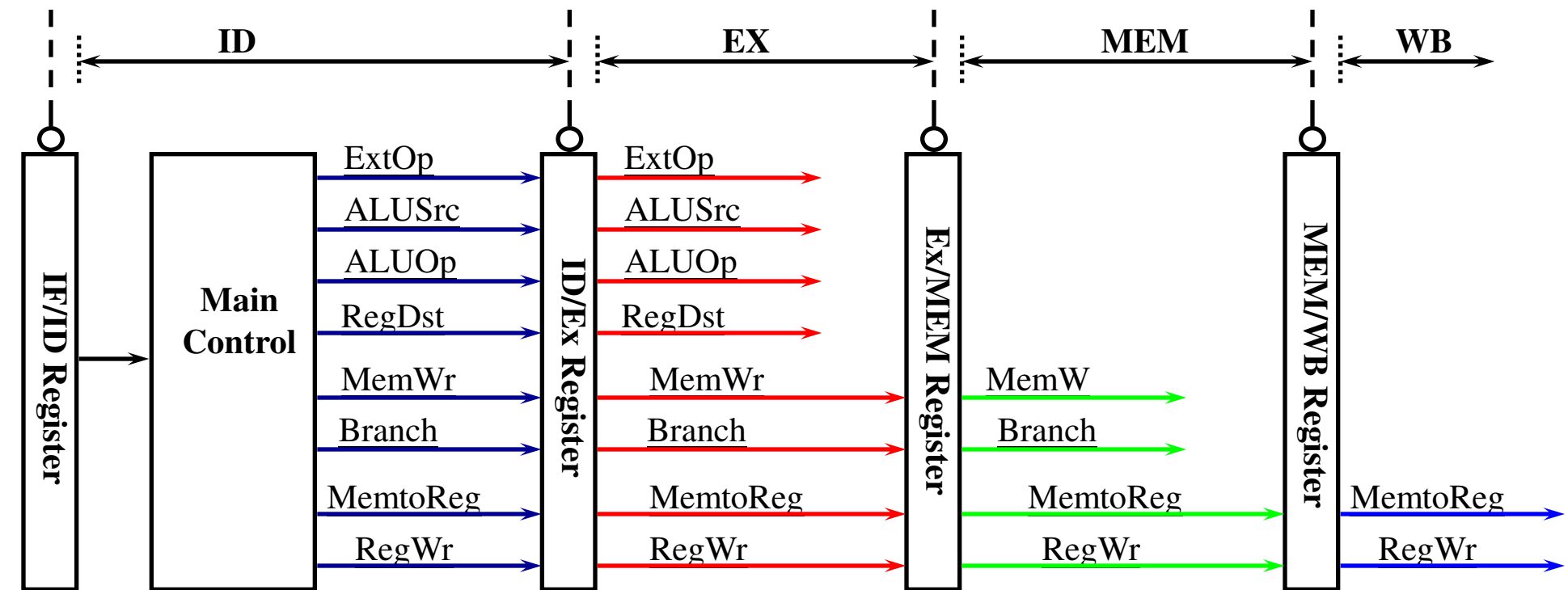
Data Stationary Control

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



Data Stationary Control (cont.)

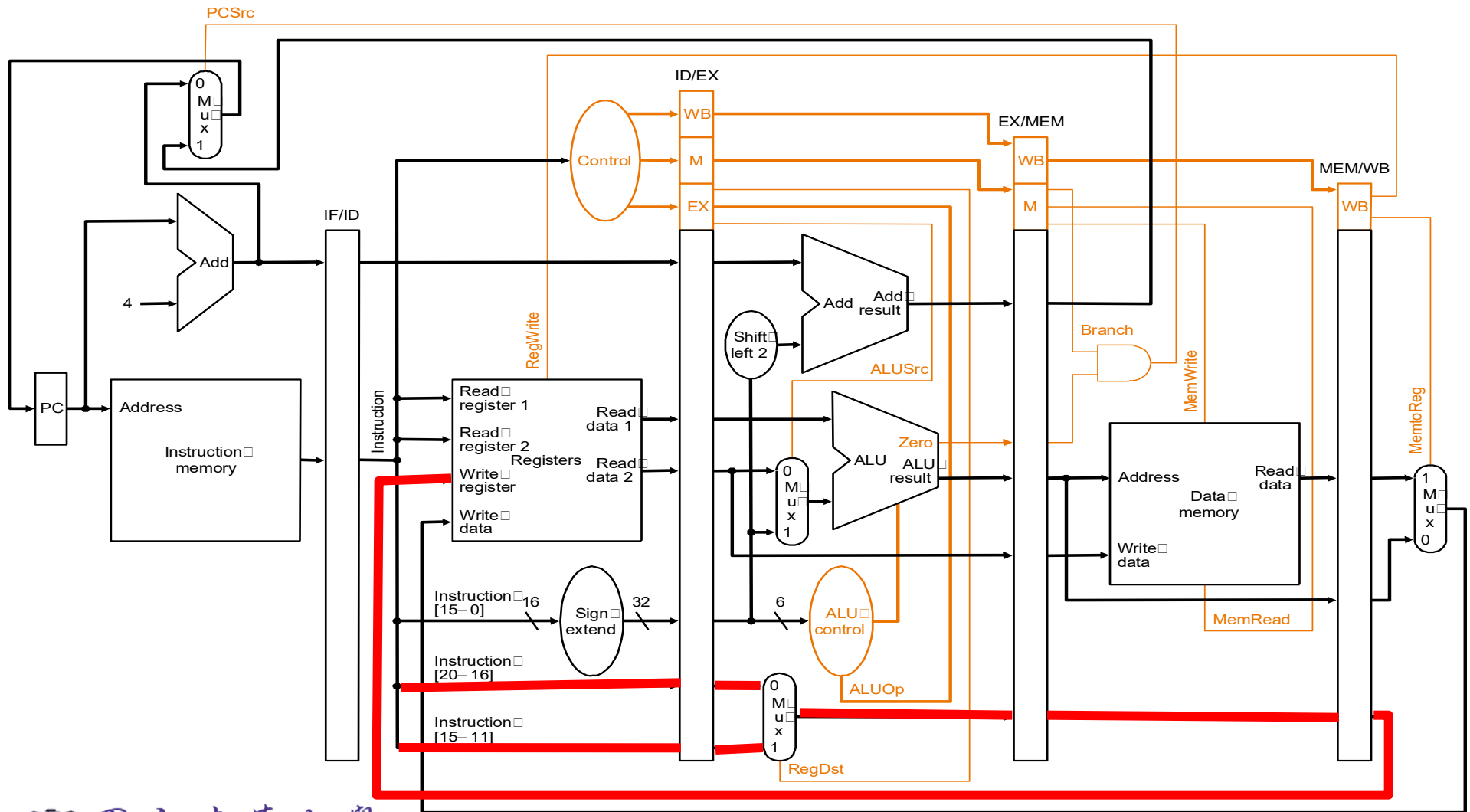
- ◆ Signals for EX (ExtOp, ALUSrc, ...) are used 1 cycle later
- ◆ Signals for MEM (MemWr, Branch) are used 2 cycles later
- ◆ Signals for WB (MemtoReg, MemWr) are used 3 cycles later



◆ **Reg[IR[20-16]] = MDR**



Datapath with Control



Let's Try it Out

lw \$10, 20(\$1)

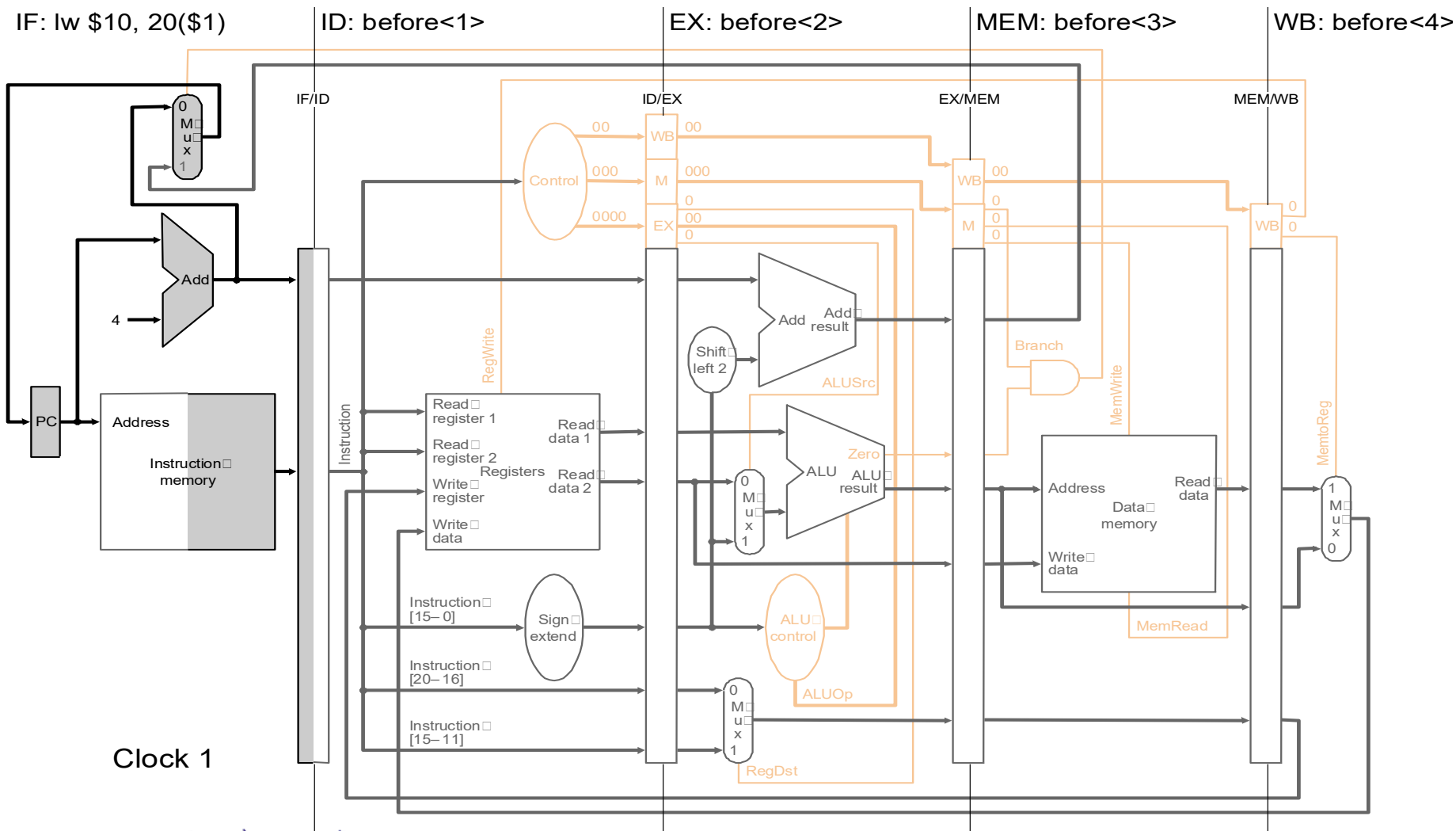
sub \$11, \$2, \$3

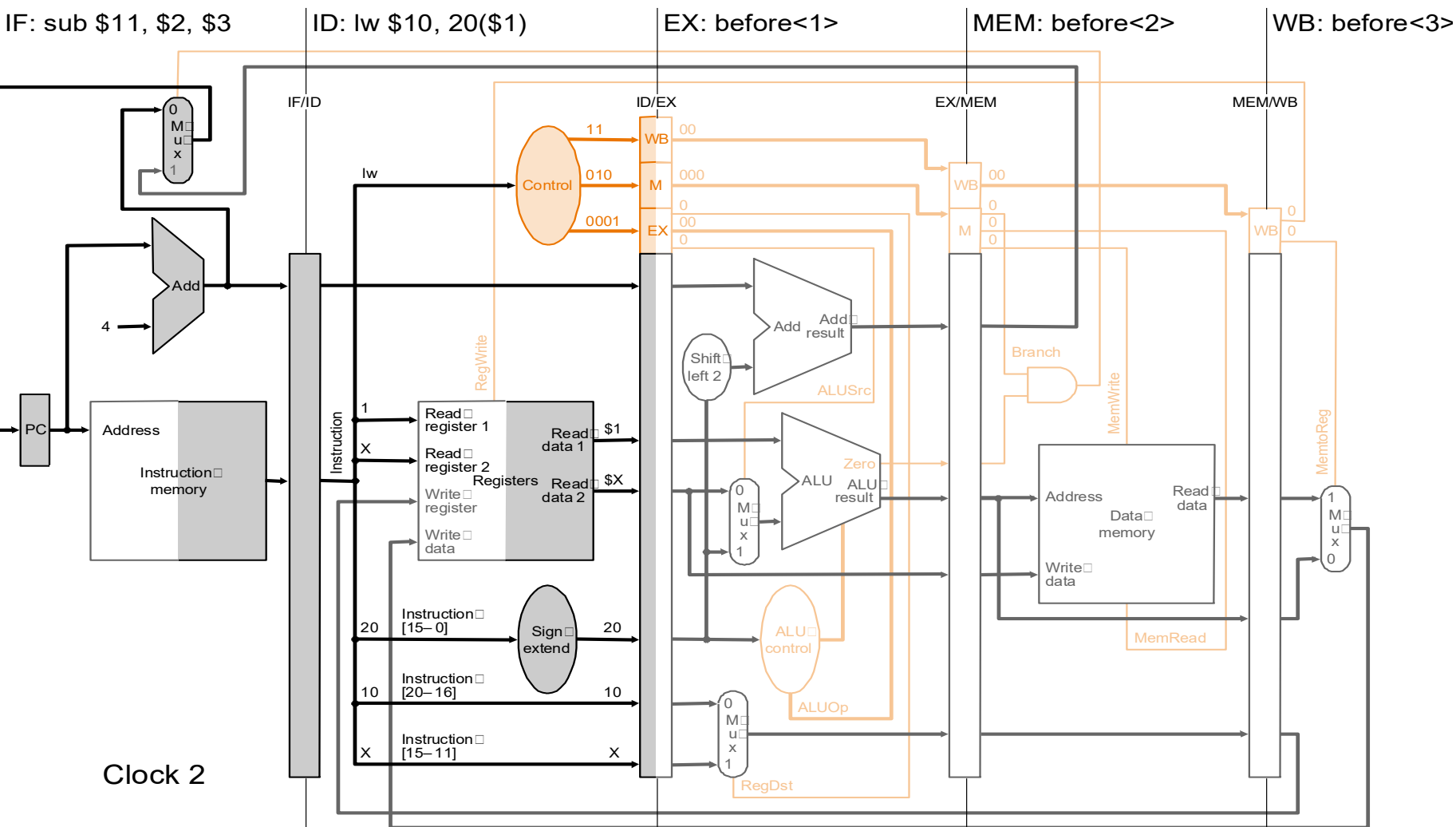
and \$12, \$4, \$5

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

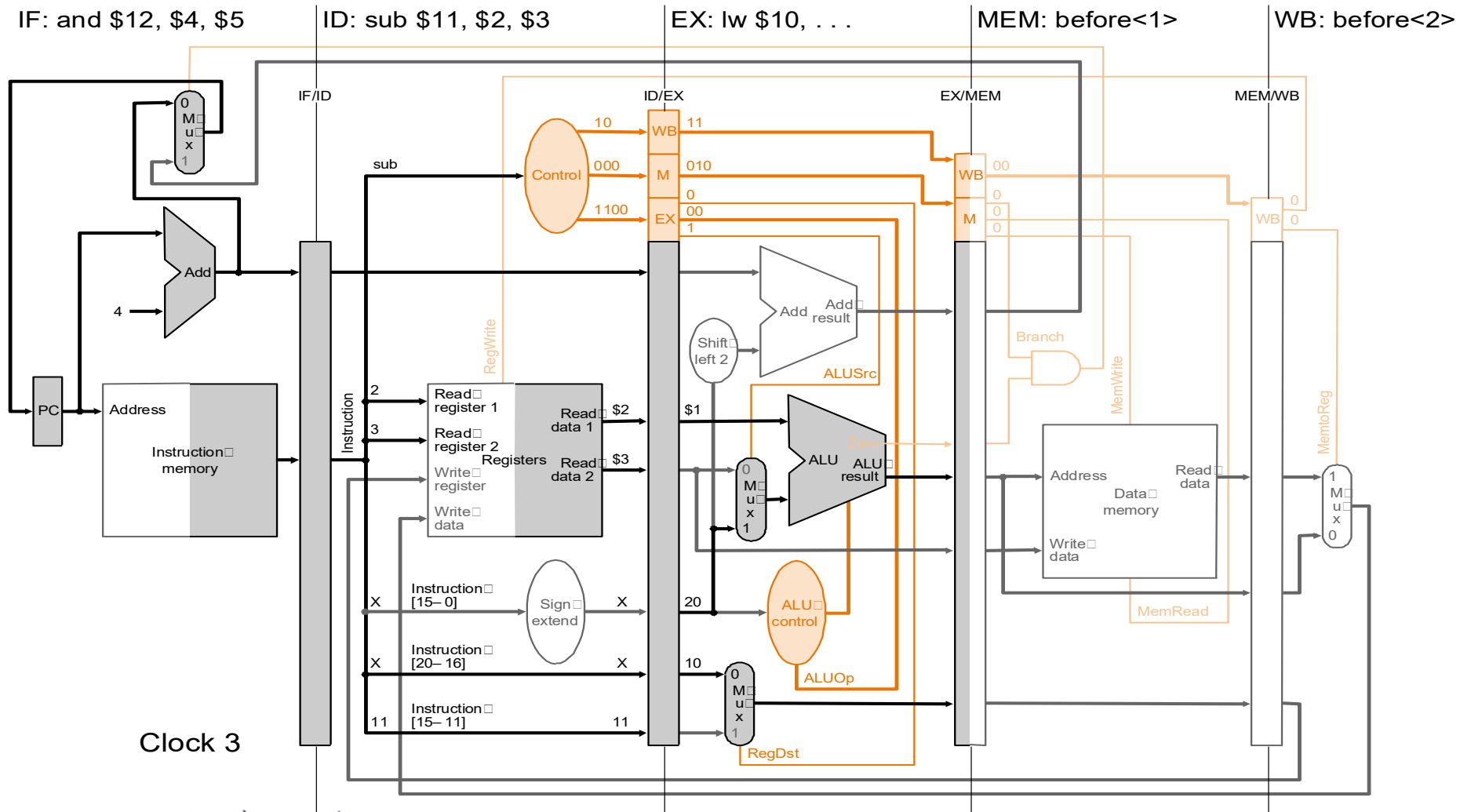
add \$14, \$8, \$9

Example 2: Cycle 1

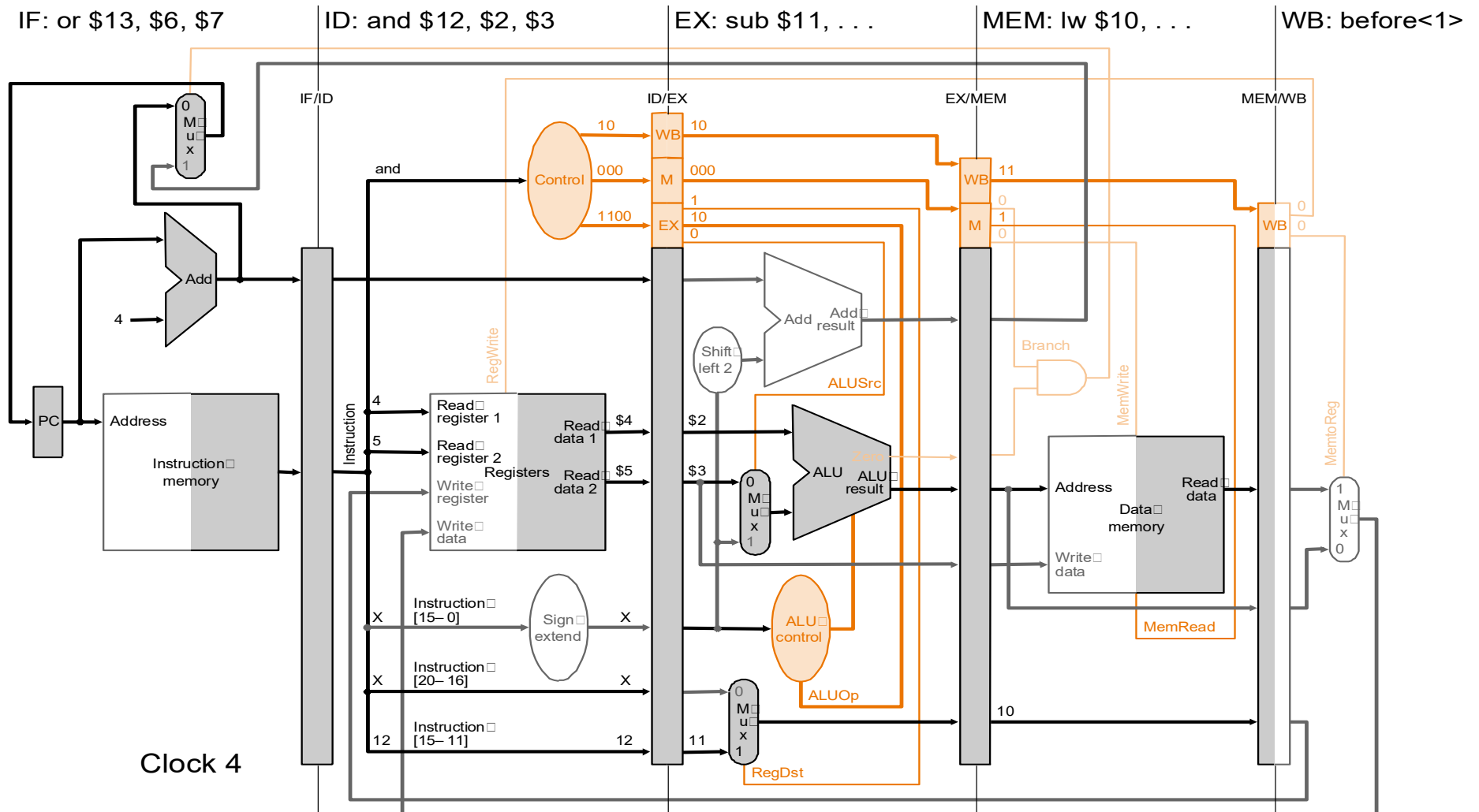




Example 2: Cycle 3

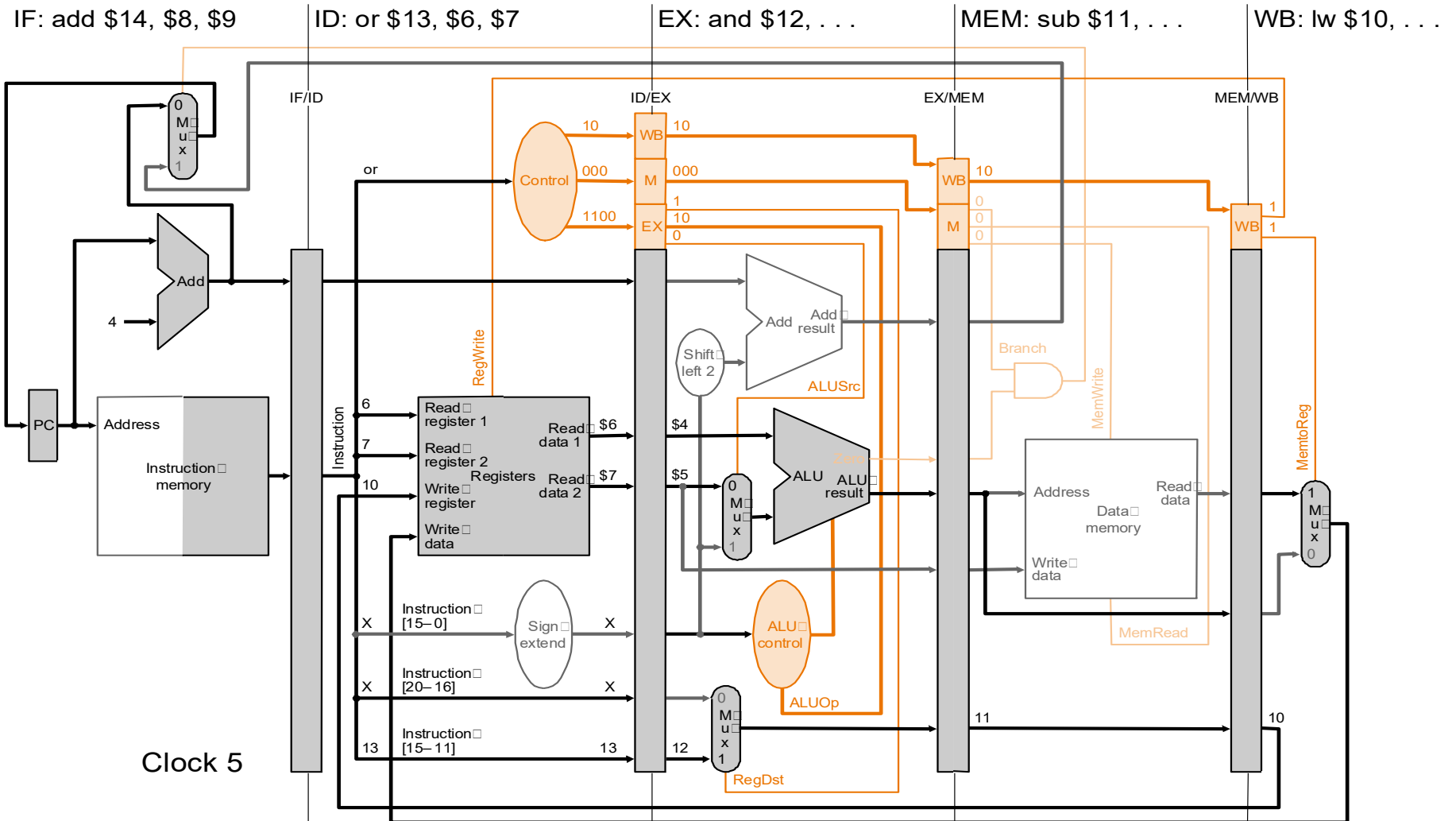


Example 2: Cycle 4

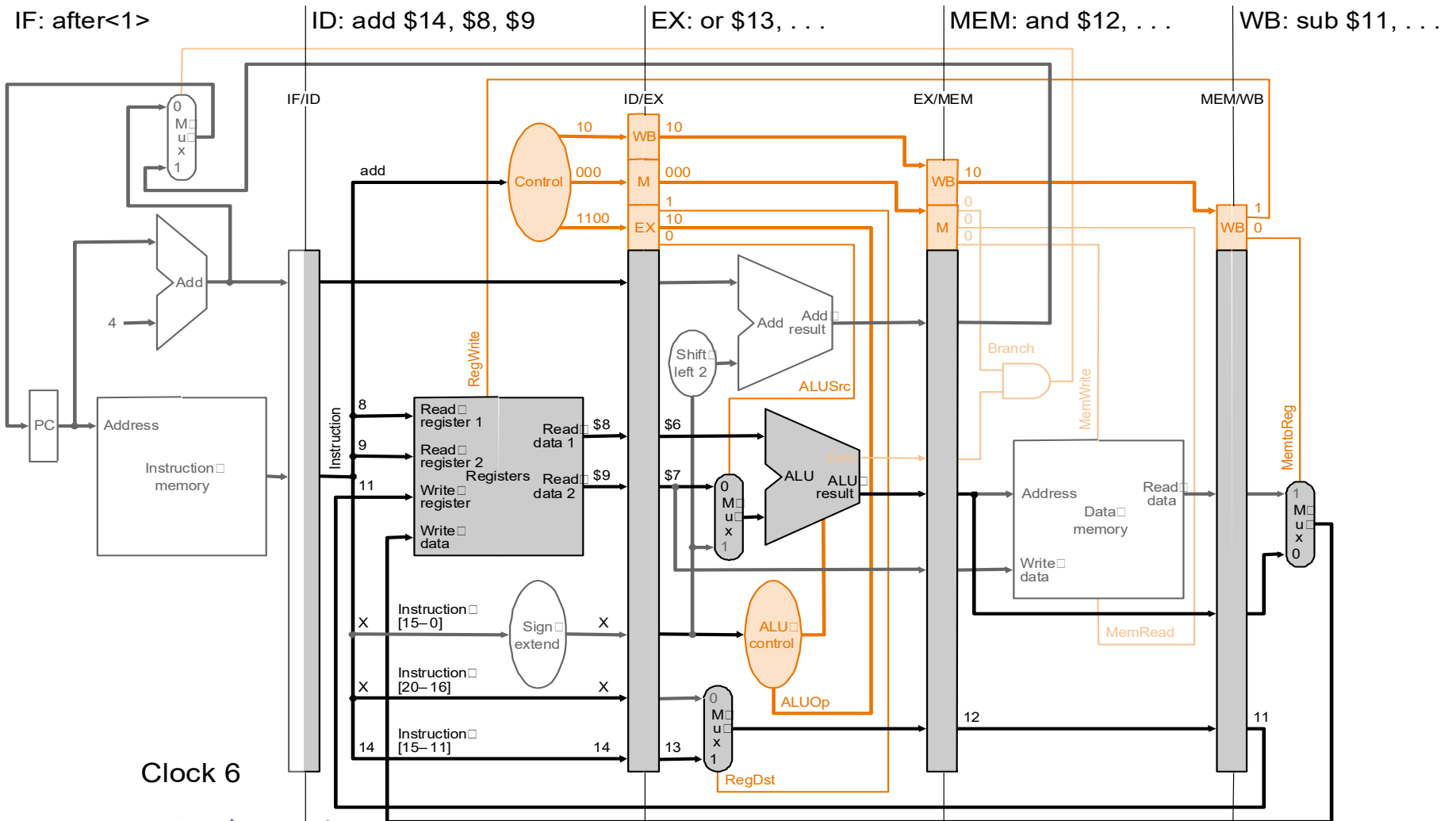


Clock 4

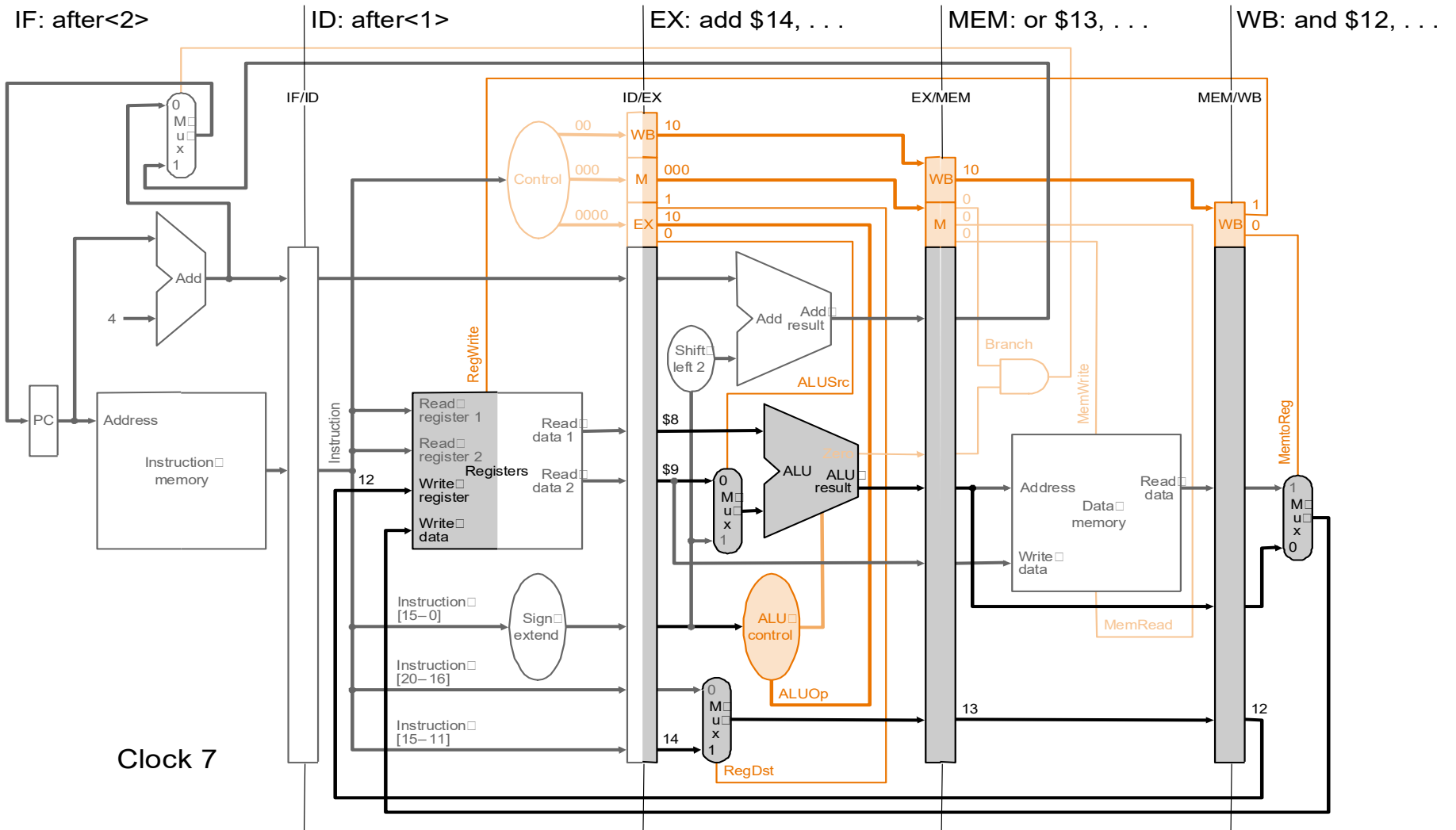
Example 2: Cycle 5



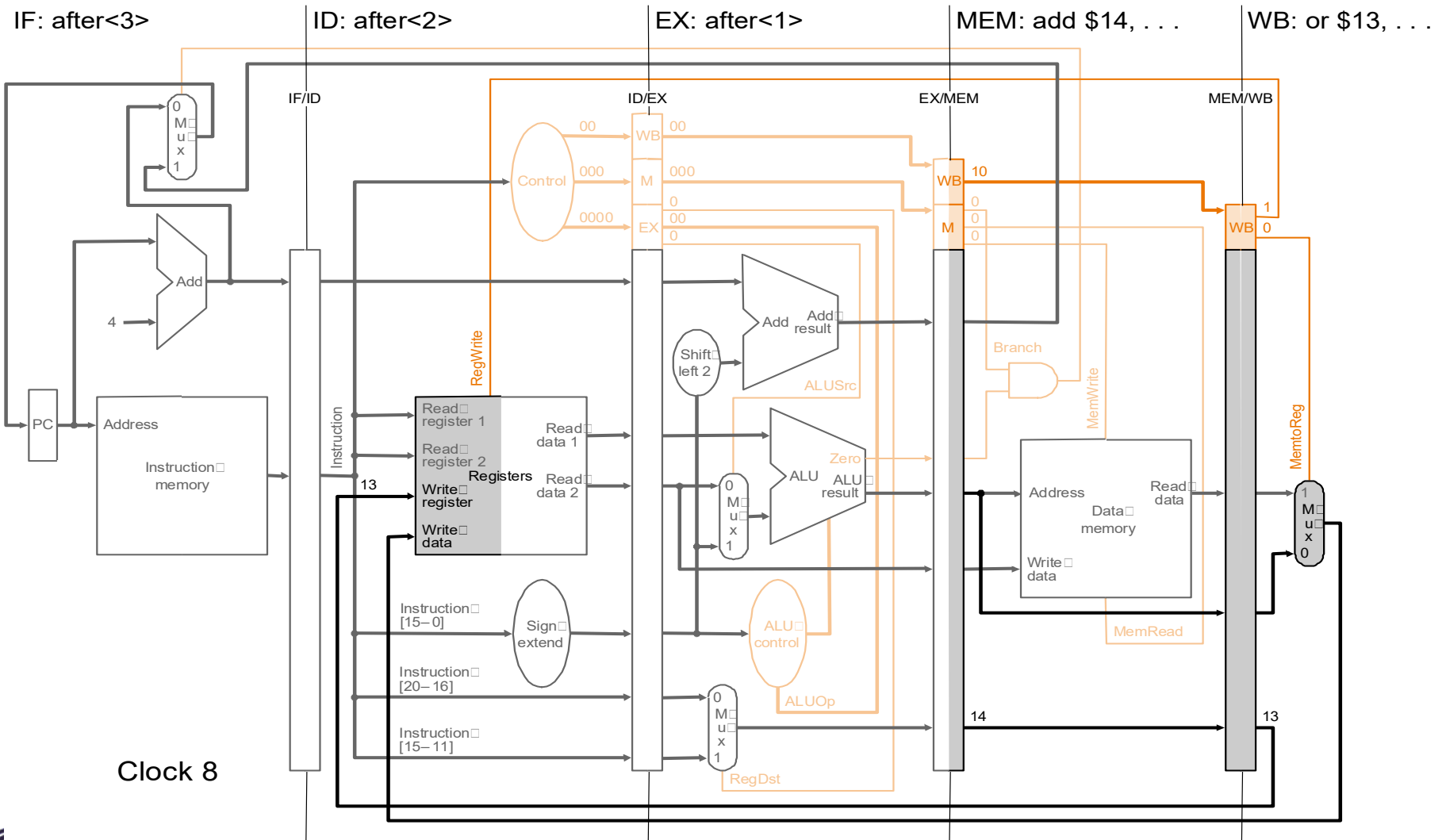
Example 2: Cycle 6



Example 2: Cycle 7

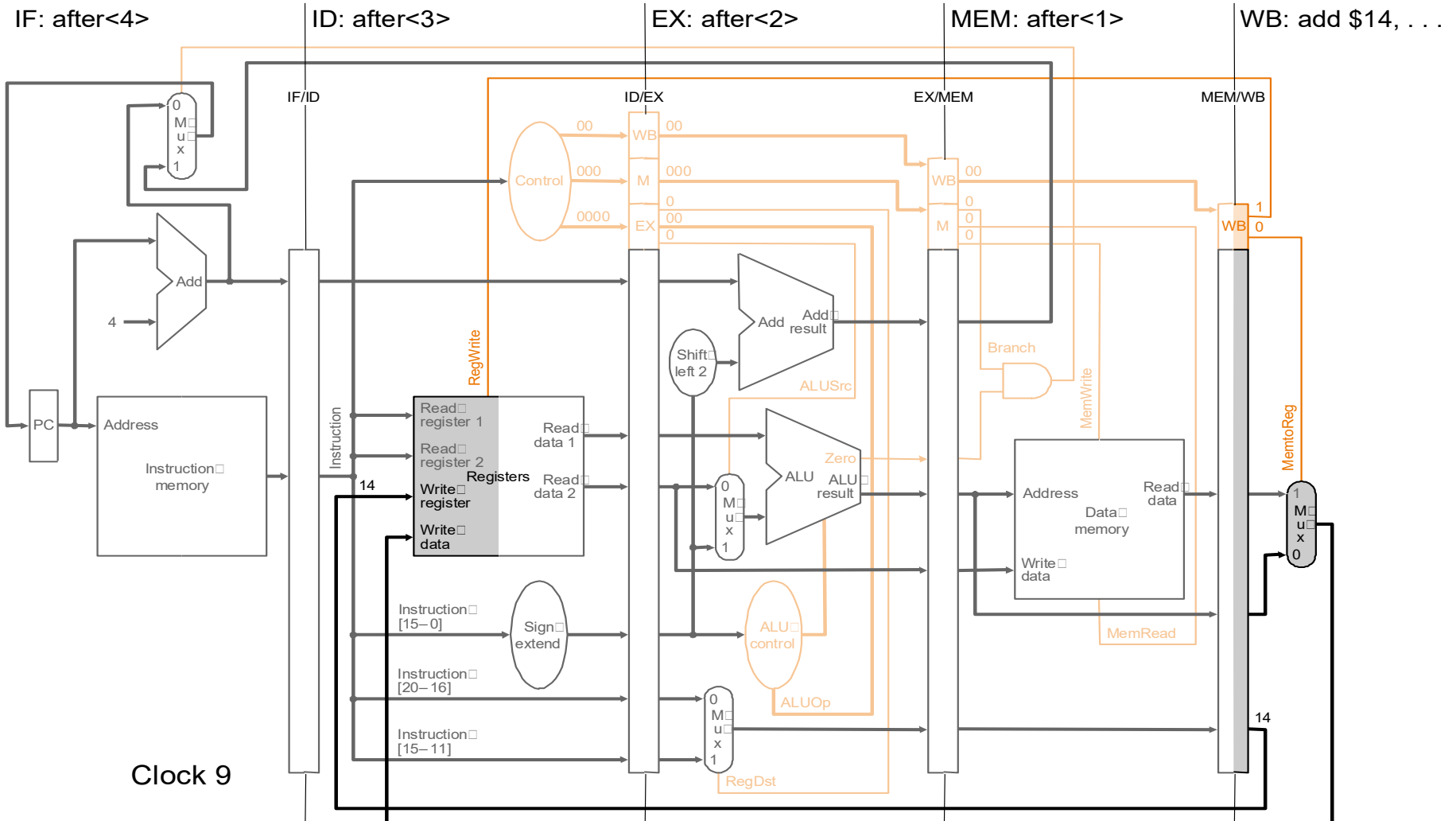


Example 2: Cycle 8



Clock 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 (R-Type and R-Type)
- ◆ Data hazards and stalls (Load and R-type)
- ◆ Branch hazards
- ◆ Exceptions
- ◆ Superscalar and dynamic pipelining

Pipeline Hazards

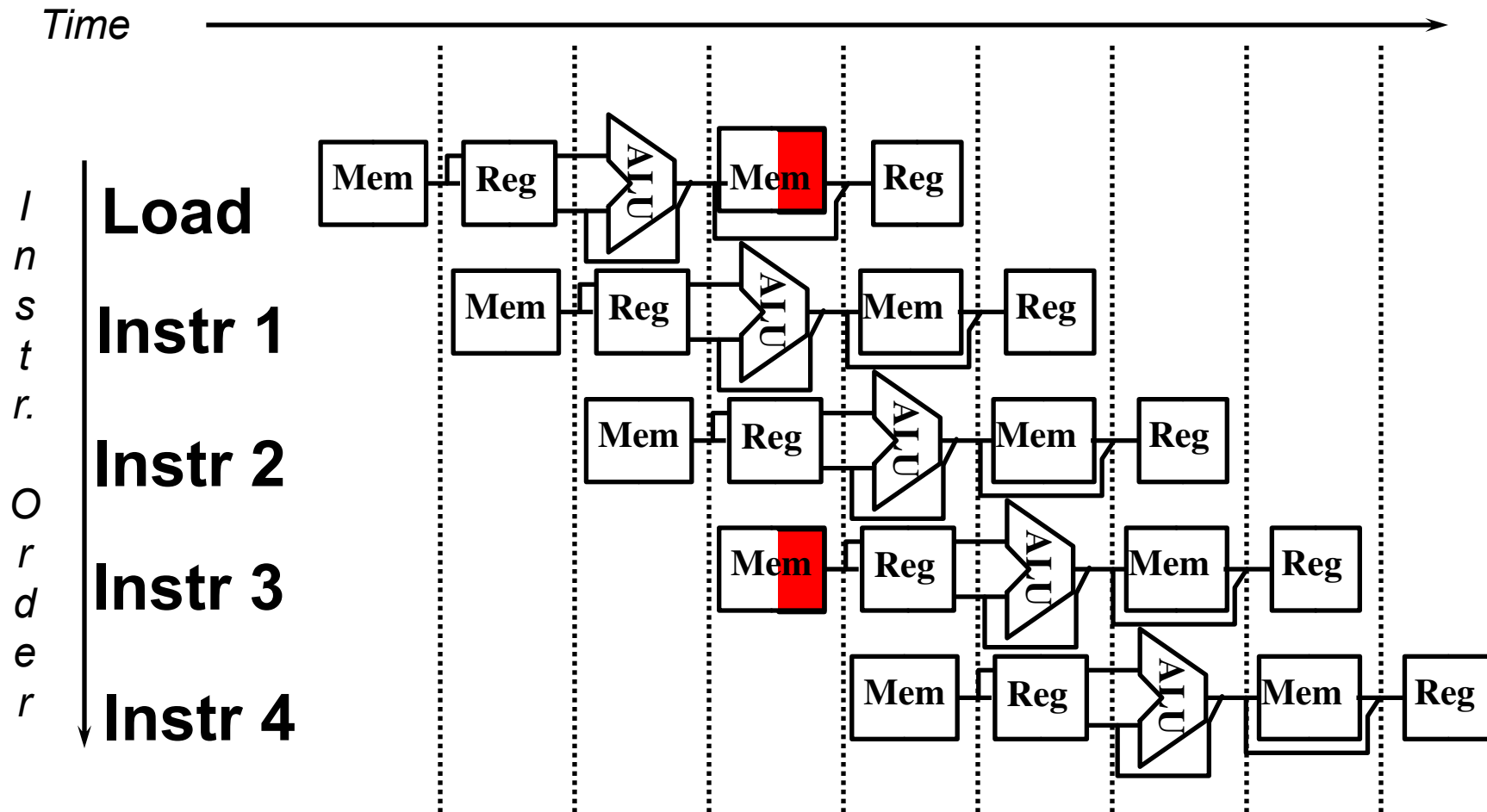
◆ Pipeline Hazards:

- **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**

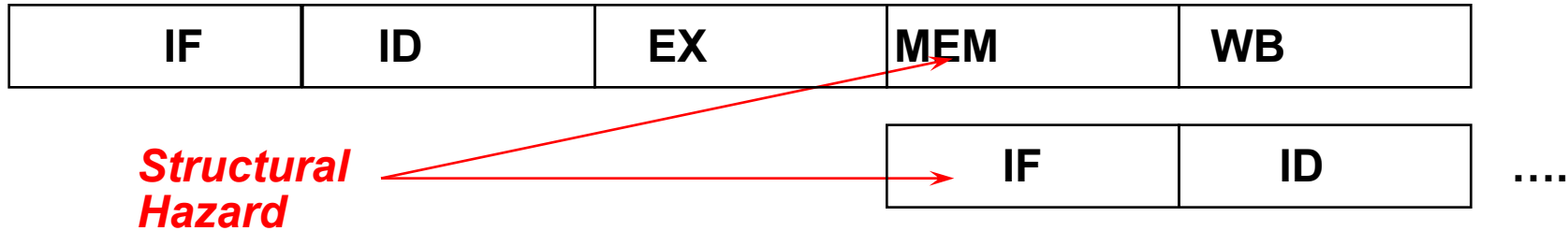
- pipeline control must detect the hazard
- take action (or delay action) to resolve hazards

Structural Hazard: Single Memory

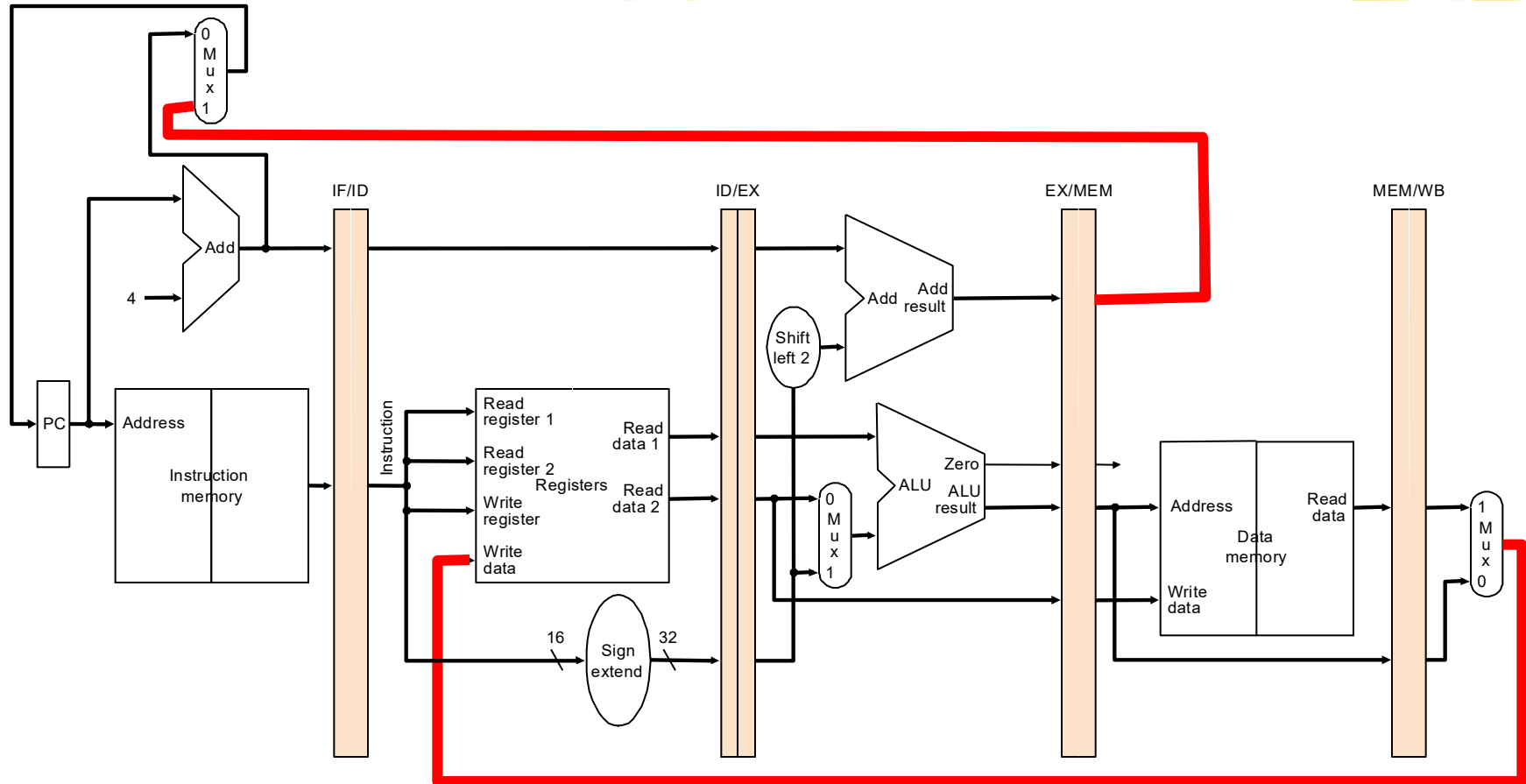


Use 2 memory: data memory and instruction memory

Pipeline Hazards Illustrated



Feedback Path

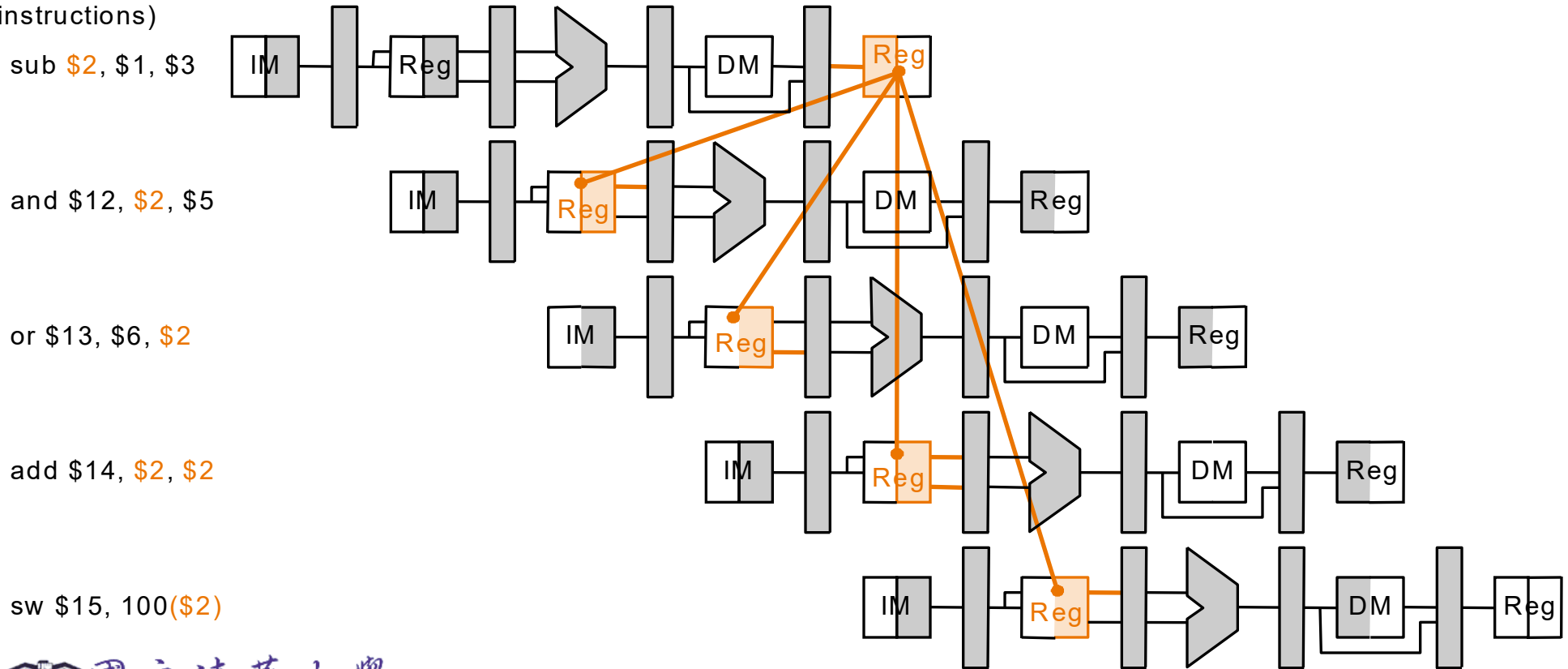


Data Hazards

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

Program
execution
order
(in instructions)

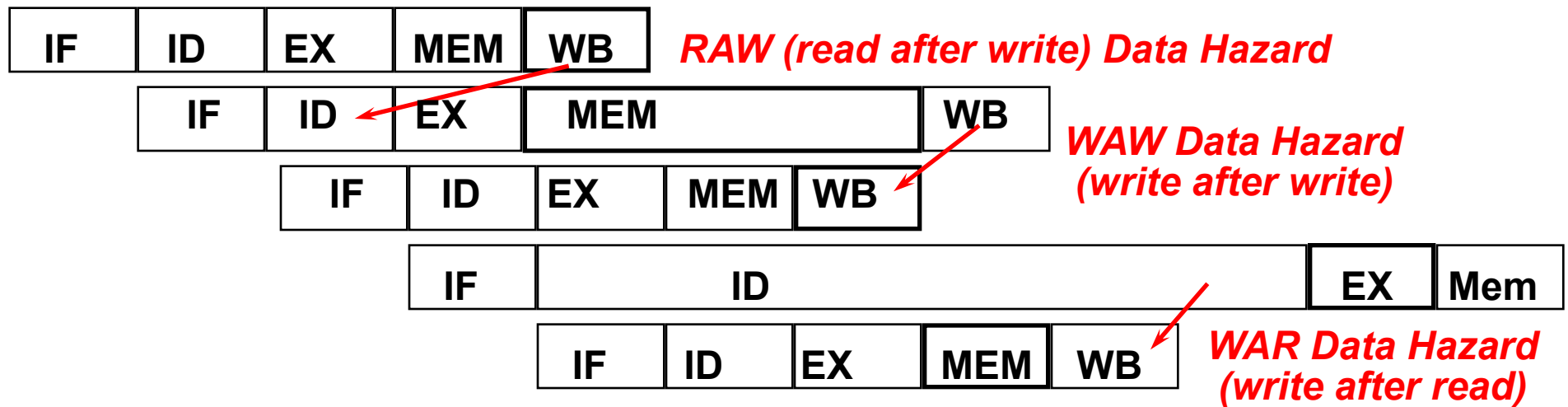


Types of Data Hazards

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 (software solution)
 - Forward (hardware solution)
 - Stall (hardware solution)

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

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

Program
execution
order
(in instructions)

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

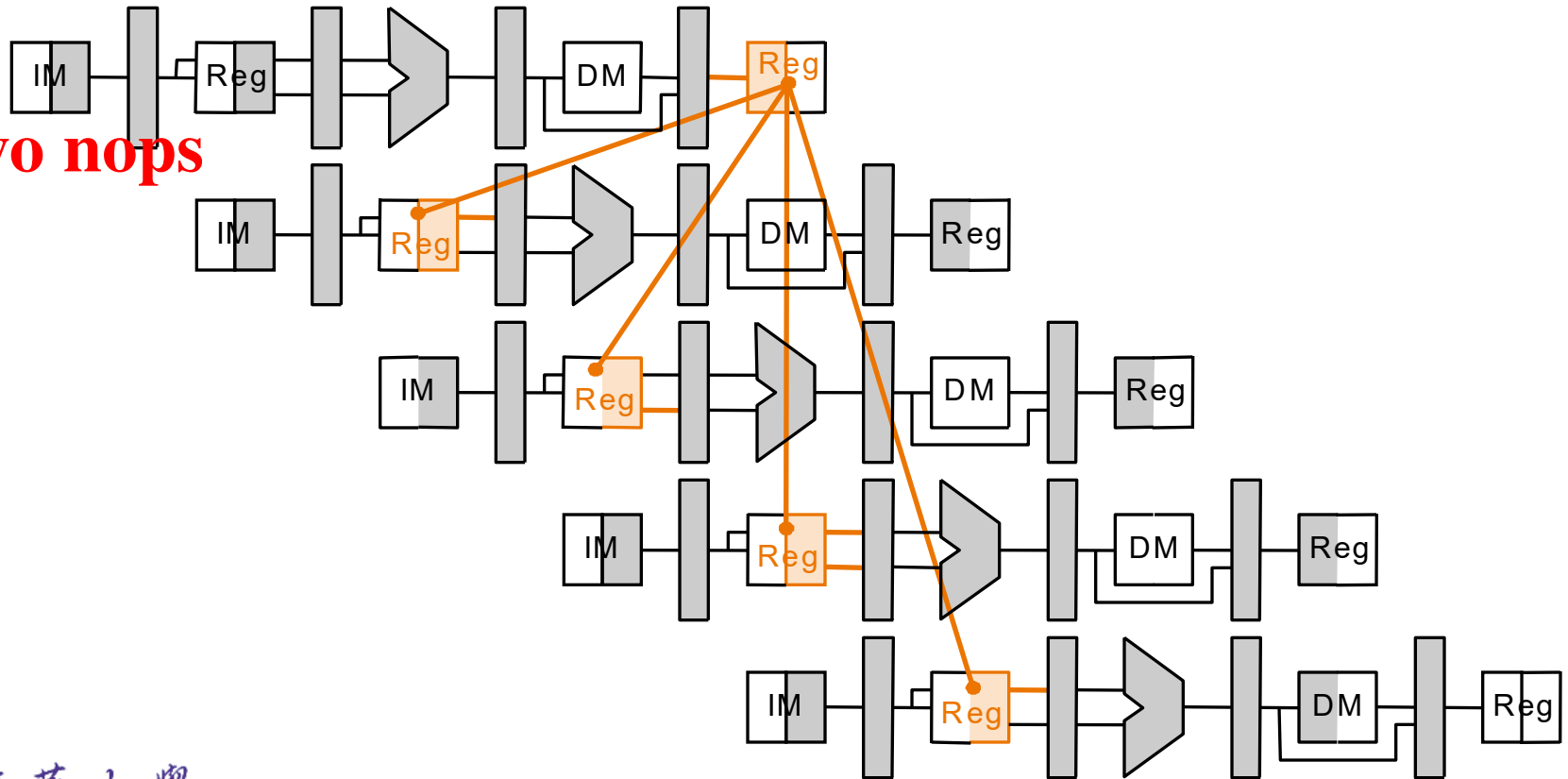
and \$12, \$2, \$5

or \$13, \$6, \$2

add \$14, \$2, \$2

sw \$15, 100(\$2)

Insert two nops

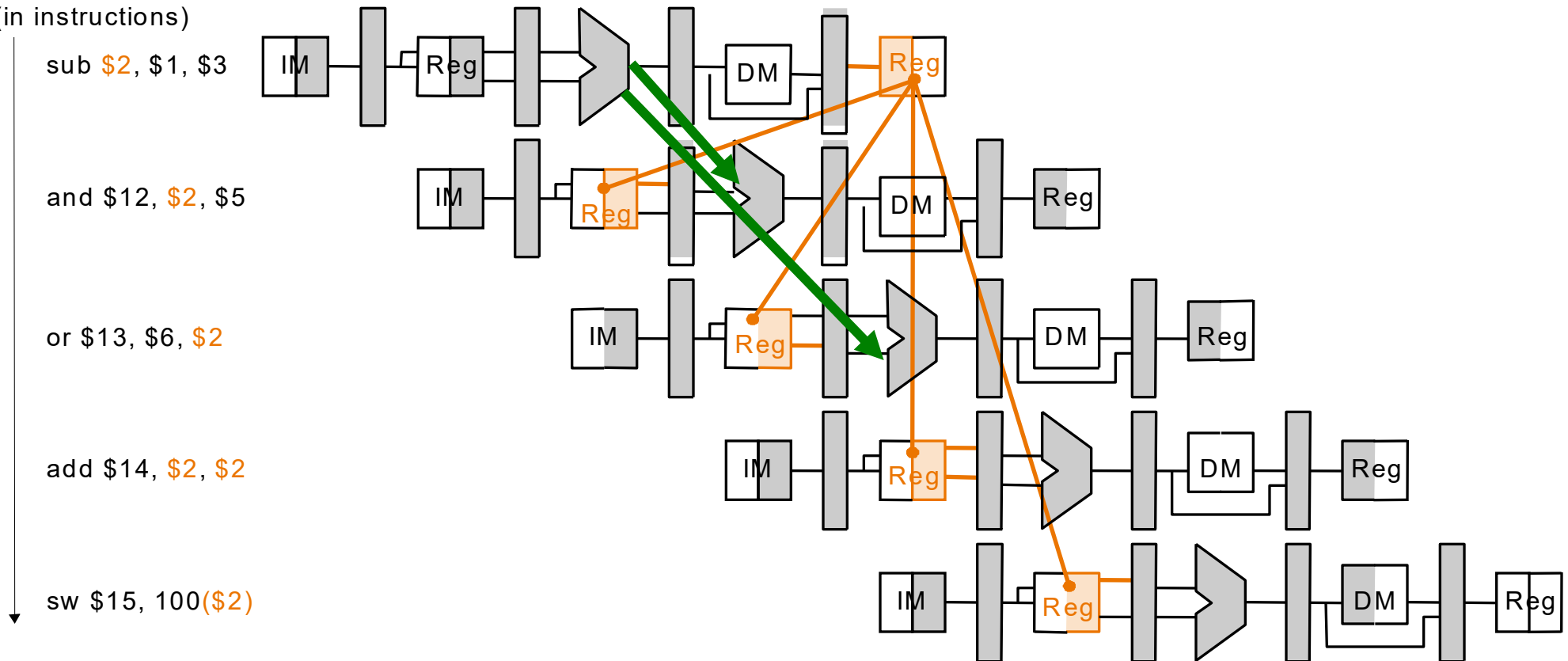


Data Hazards : Forwarding

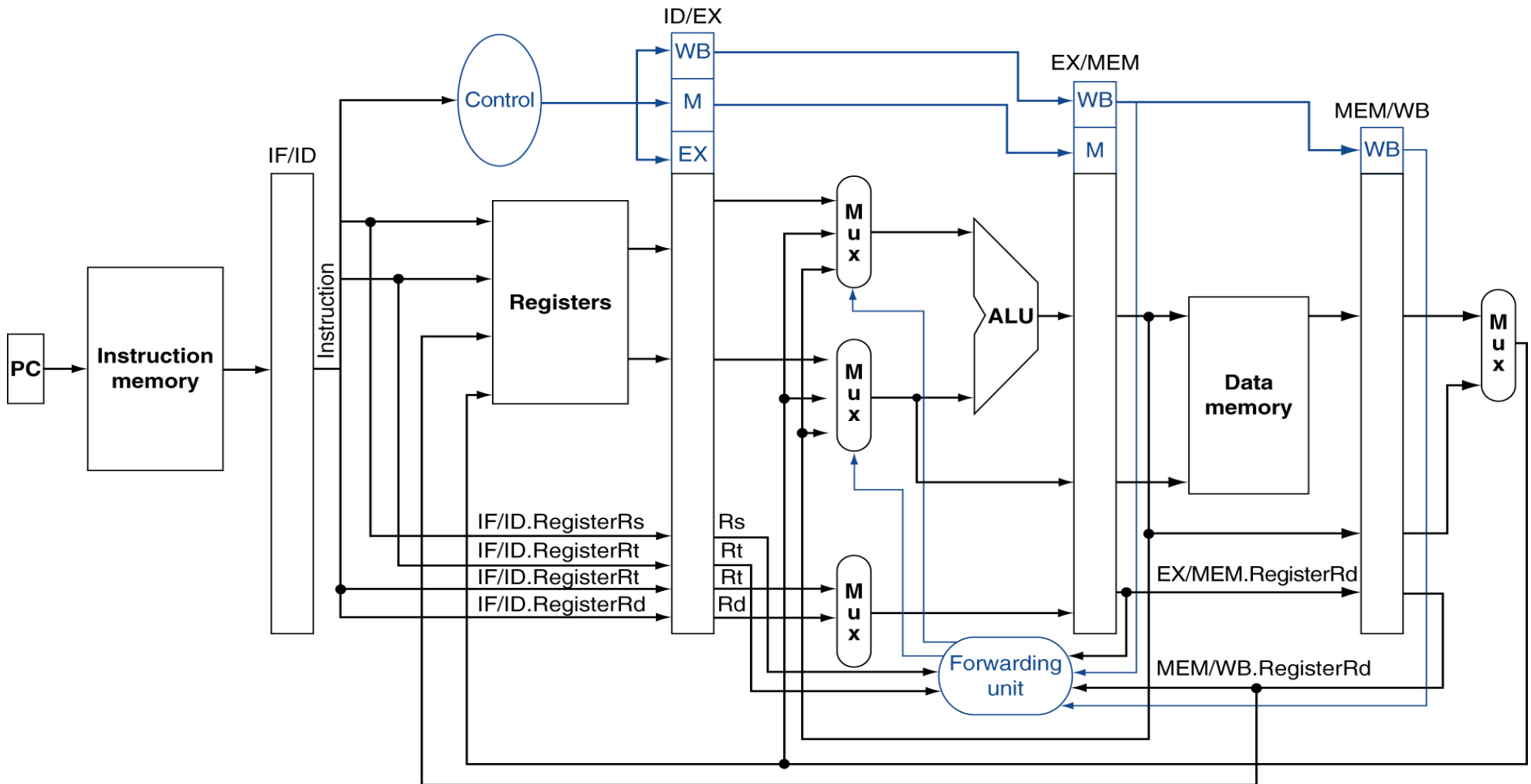
Time (in clock cycles)

	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

Program execution order (in instructions)



Datapath with Forwarding



Control: Detecting Data Hazards

◆ Hazard conditions:

- 1a. $\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs}$
- 1b. $\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRt}$
- 2a. $\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs}$
- 2b. $\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt}$

◆ 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.)

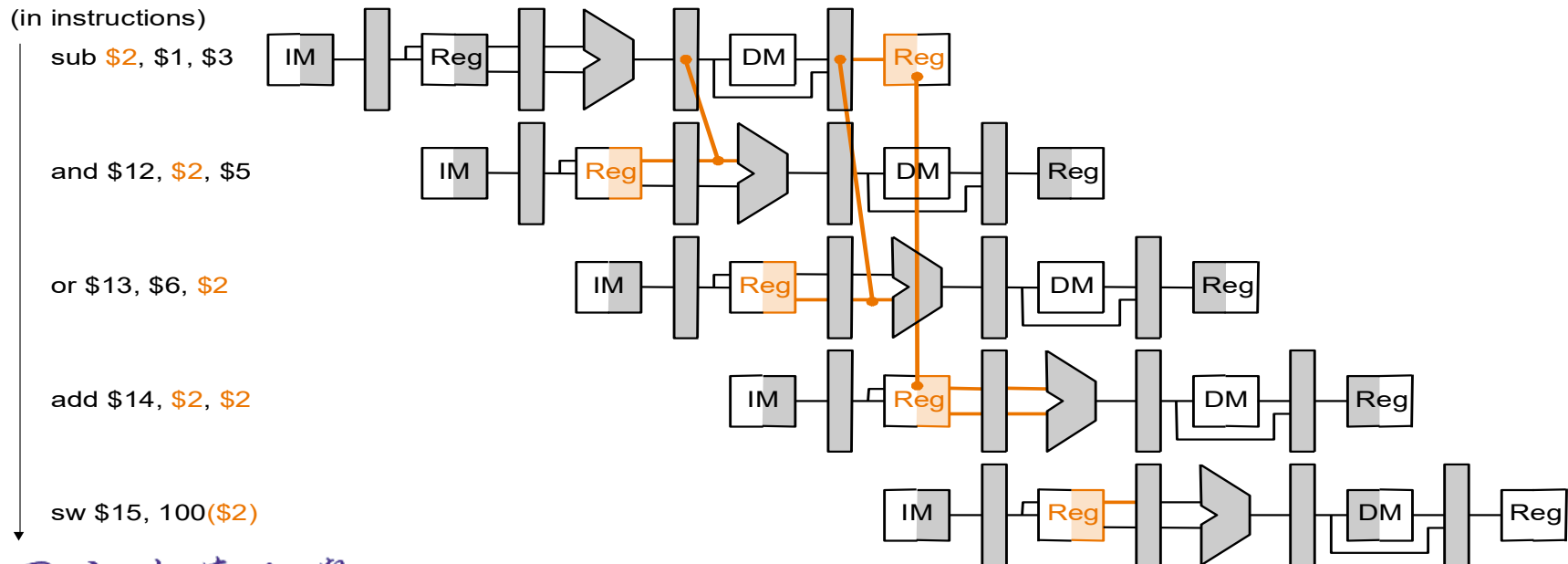
- ◆ Hazard conditions using control signals:
 - At EX stage:
EX/MEM.RegWrite and (EX/MEM.RegRd \neq 0)
and (EX/MEM.RegRd=ID/EX.RegRs)
 - At MEM stage:
MEM/WB.RegWrite and (MEM/WB.RegRd \neq 0)
and (MEM/WB.RegRd=ID/EX.RegRs)
 - (replace ID/EX.RegRt for ID/EX.RegRs for the other two conditions)

Resolving Hazards: Forwarding

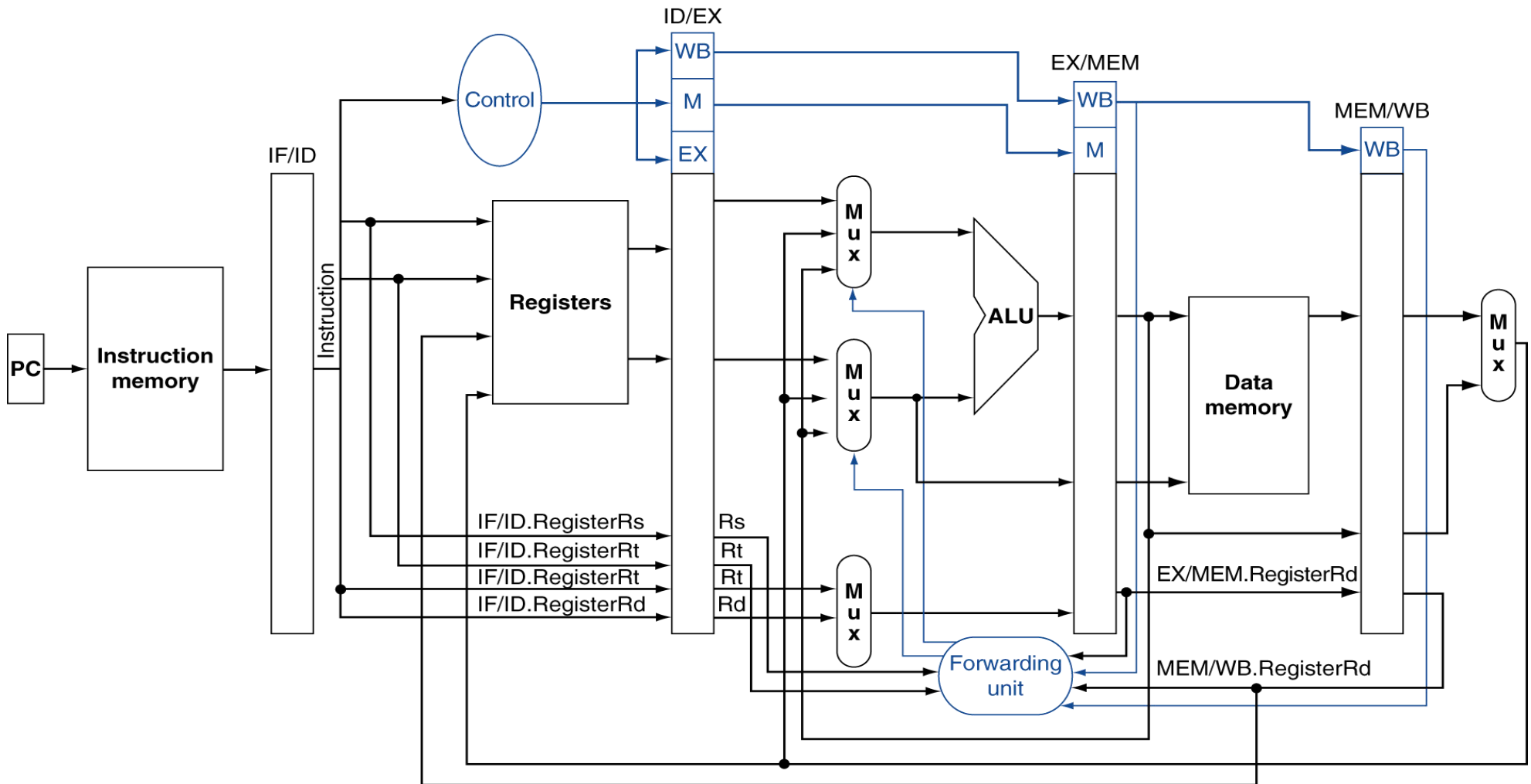
- ◆ Use temporary results, e.g., those in pipeline registers, don't wait for them to be written

	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)



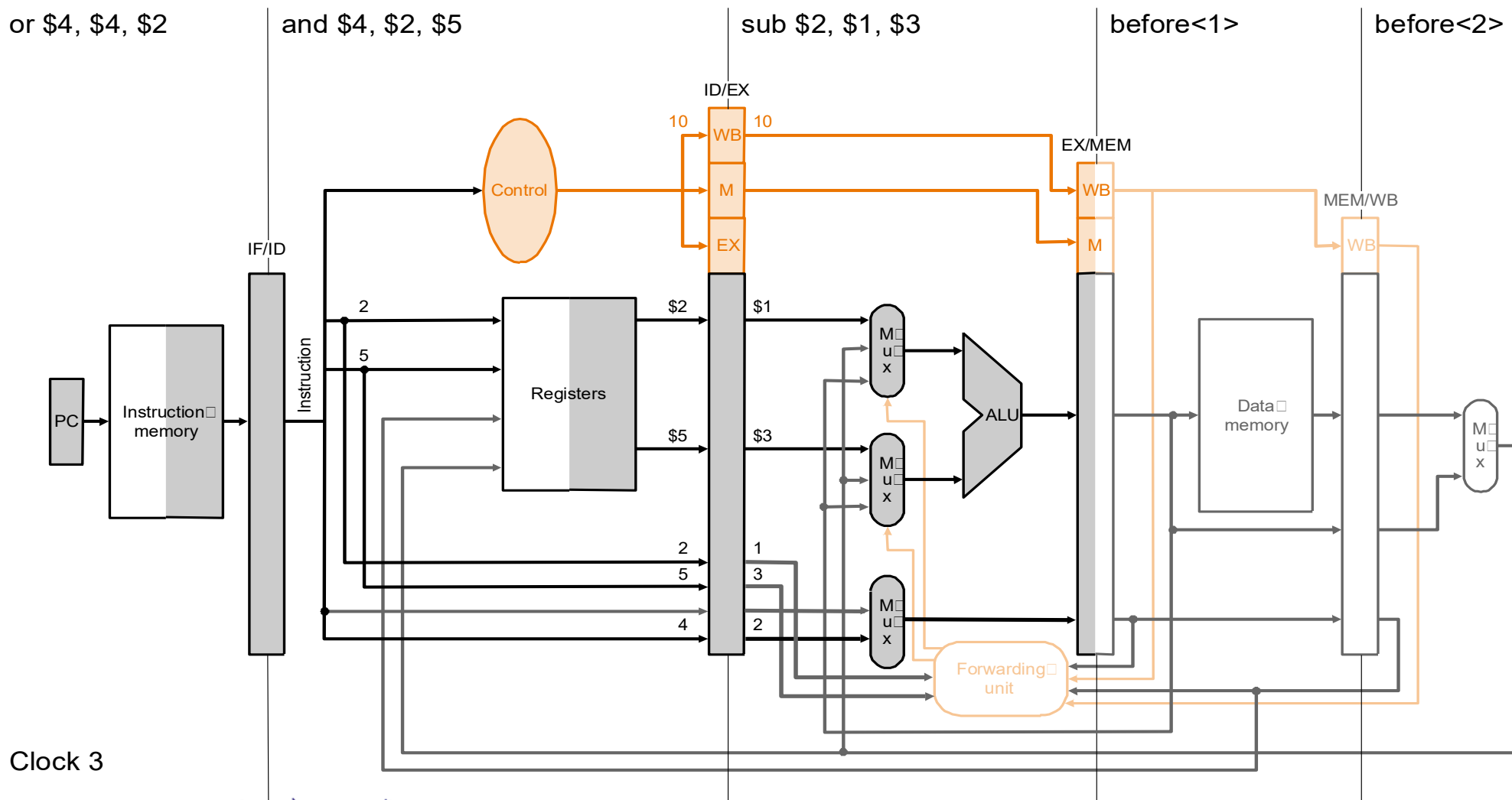
Datapath with Forwarding



Forwarding Logic

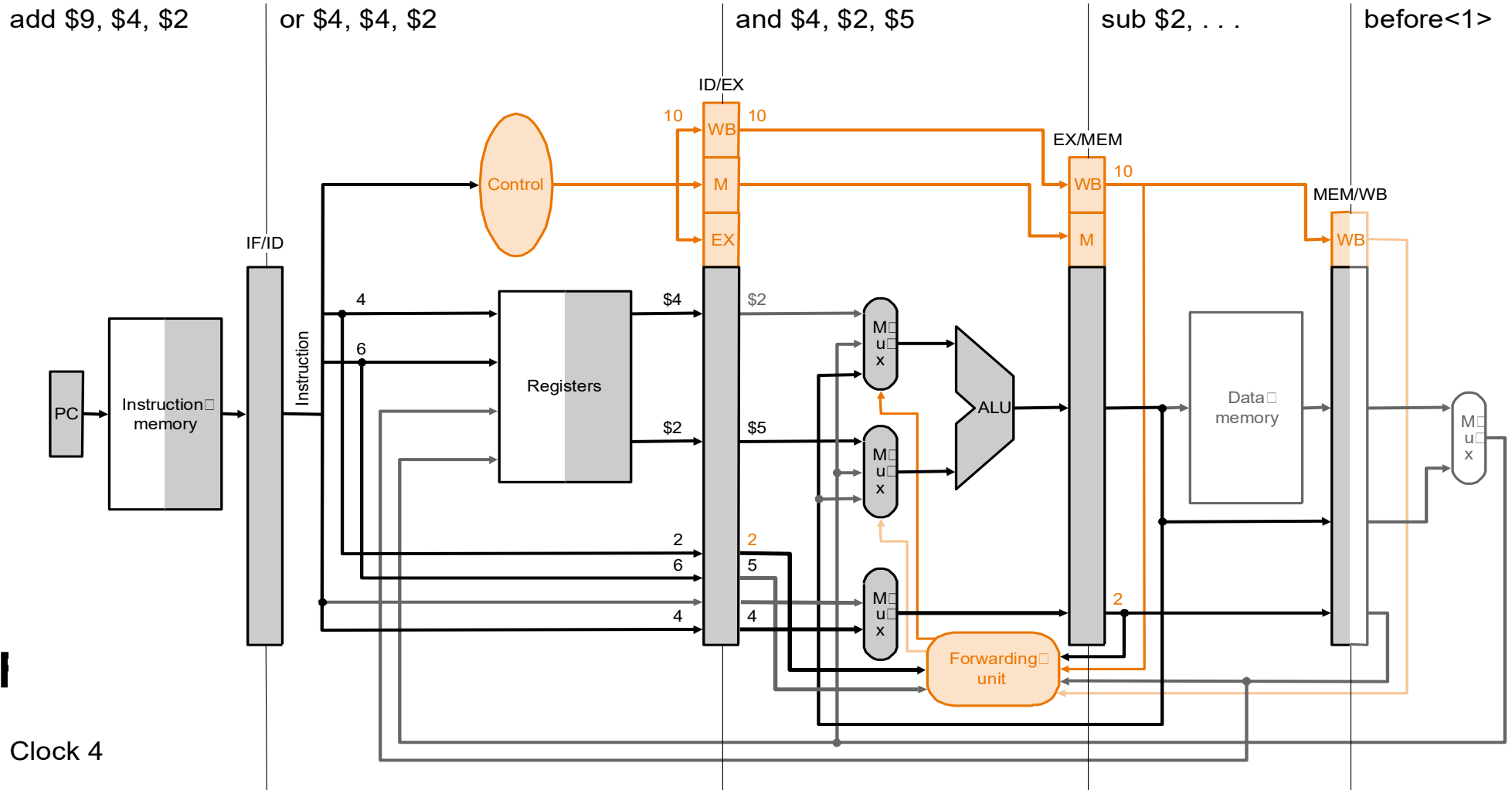
- ◆ Forwarding: input to ALU from any pipe reg.
 - Add multiplexors to ALU input 
 - Control forwarding in EX => carry Rs in ID/EX
 - ◆ Control signals for forwarding:
 - If both WB and MEM forward, e.g., add \$1,\$1,\$2; add \$1,\$1,\$3; add \$1,\$1,\$4; => let MEM forward
 - EX hazard:
 - if (EX/MEM.RegWrite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

Example 3: Cycle 3



Clock 3

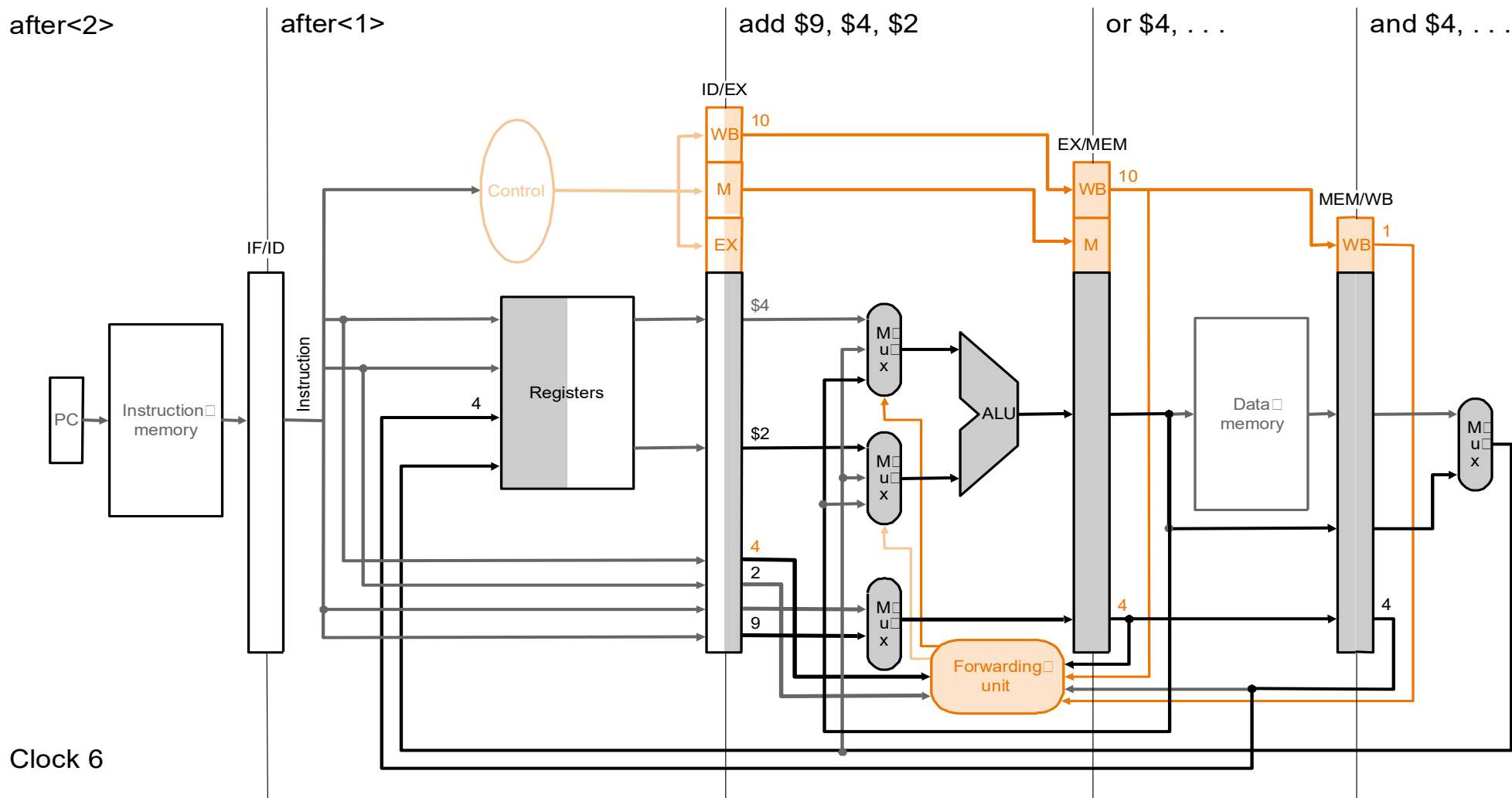
Example 3: Cycle 4



Clock 4



Example 3: Cycle 6



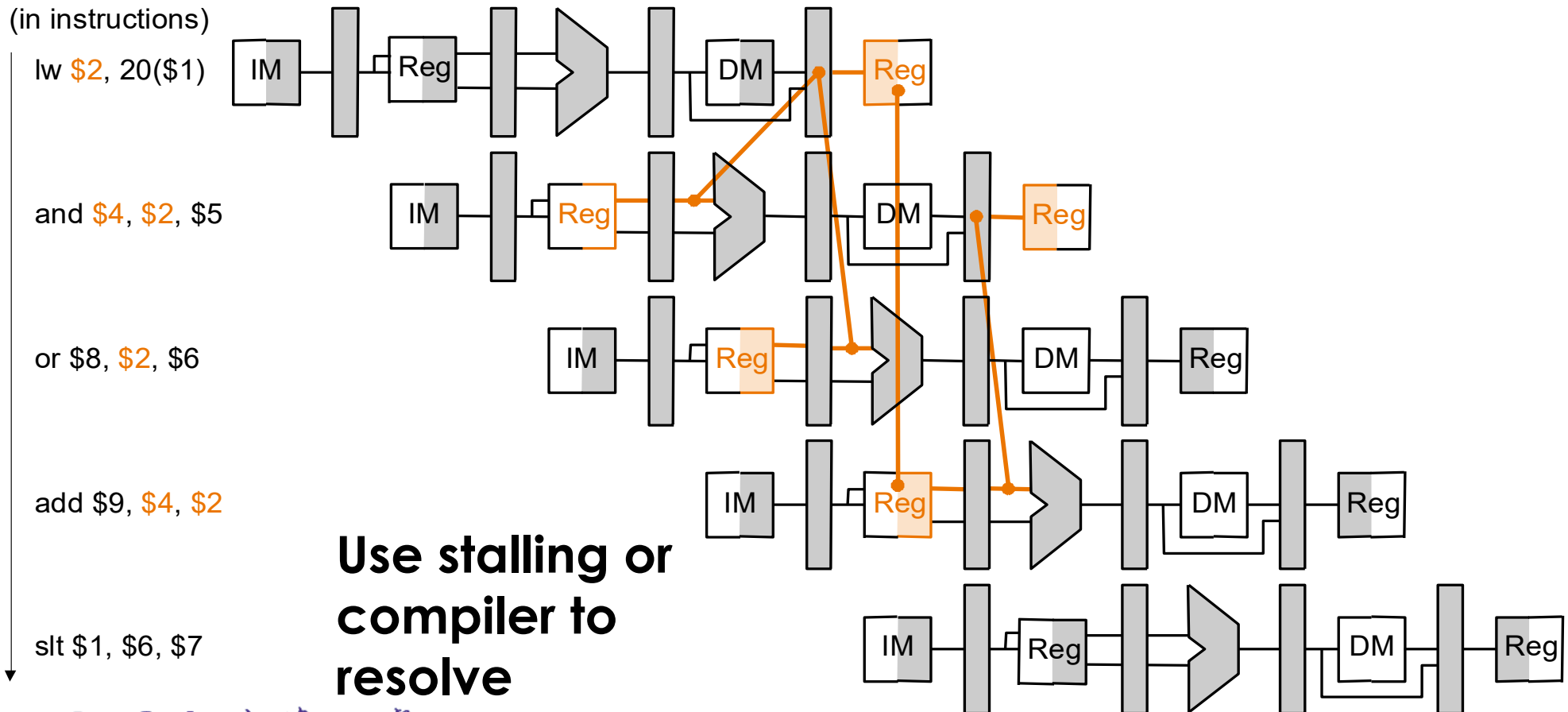
Clock 6

Outline

- ◆ An overview of pipelining
- ◆ A pipelined datapath
- ◆ Pipelined control
- ◆ Data hazards and forwarding (R-Type and R-Type)
- ◆ Data hazards and stalls (Load and R-type)
- ◆ Branch hazards
- ◆ Exceptions
- ◆ Superscalar and dynamic pipelining

Can't Always Forward

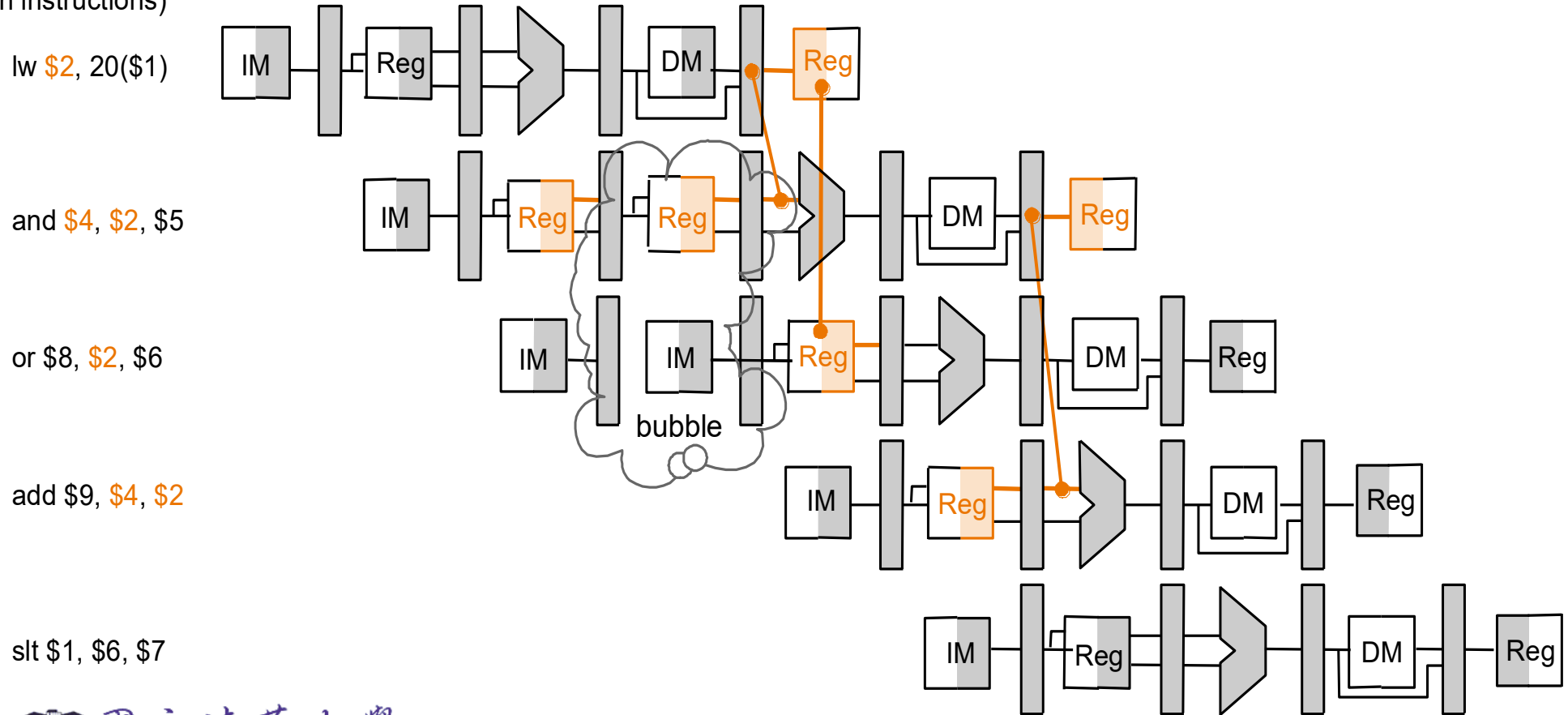
- ◆ **lw can still cause a hazard:**
 - if is followed by an instruction to read the loaded reg.



Stalling

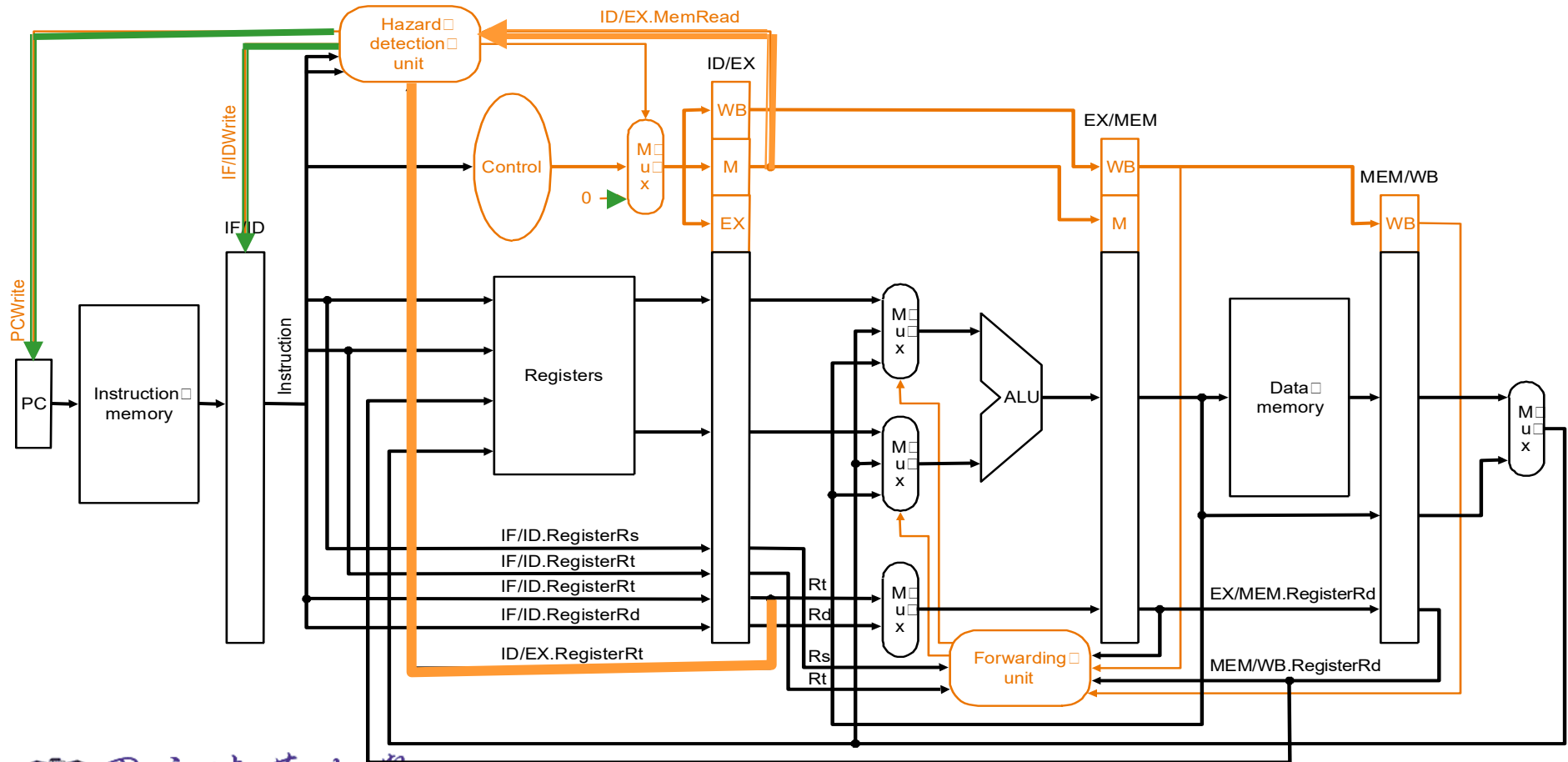
- ◆ Stall pipeline by keeping instructions in same stage and inserting an NOP instead

(in instructions)



Datapath with Stalling Unit

- ◆ Forwarding controls ALU inputs, hazard detection controls PC, IF/ID, control signals



Control: Handling Stalls

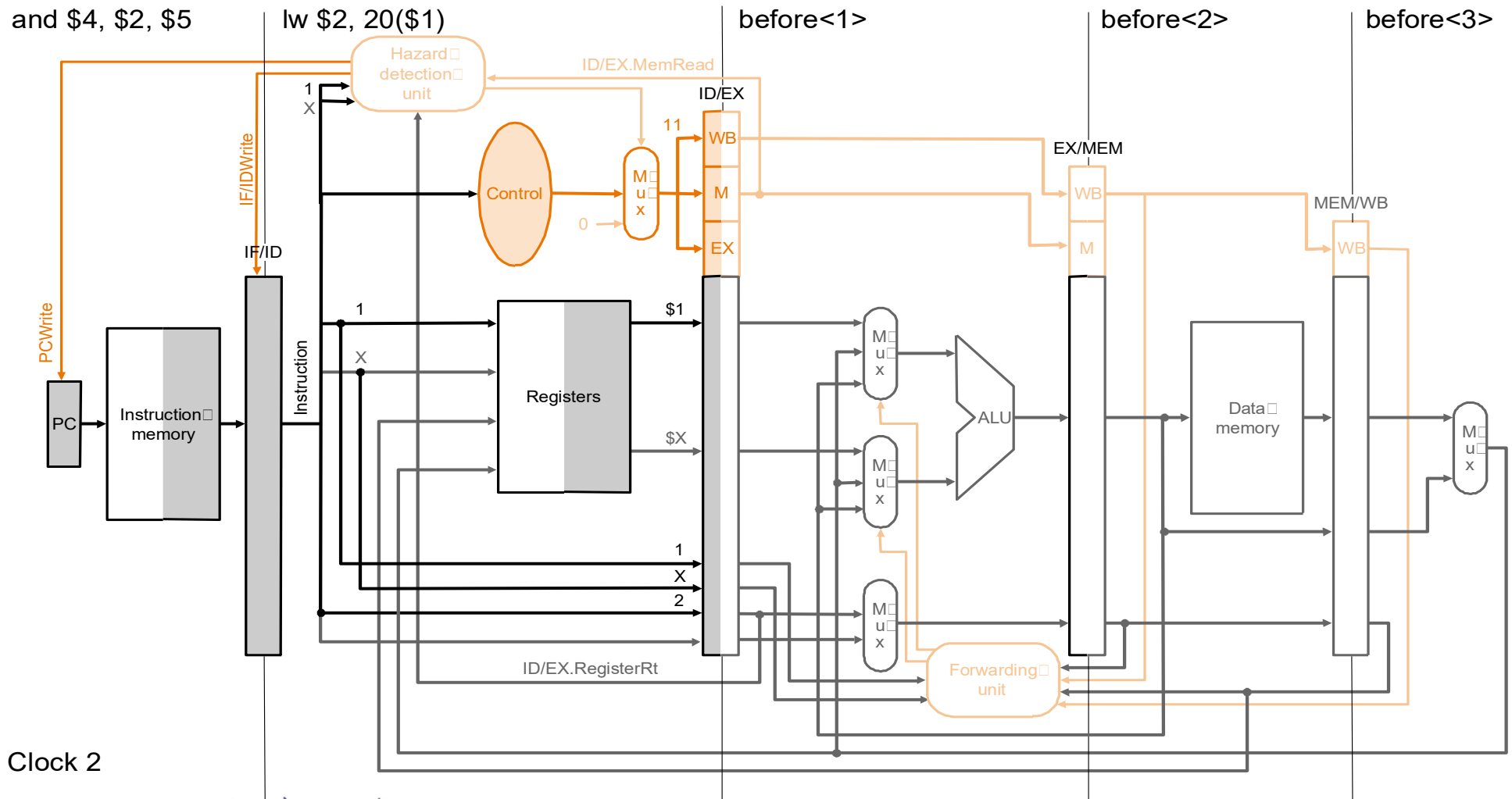
- ◆ Hazard detection unit in ID to insert stall between a load instruction and its use:

```
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?

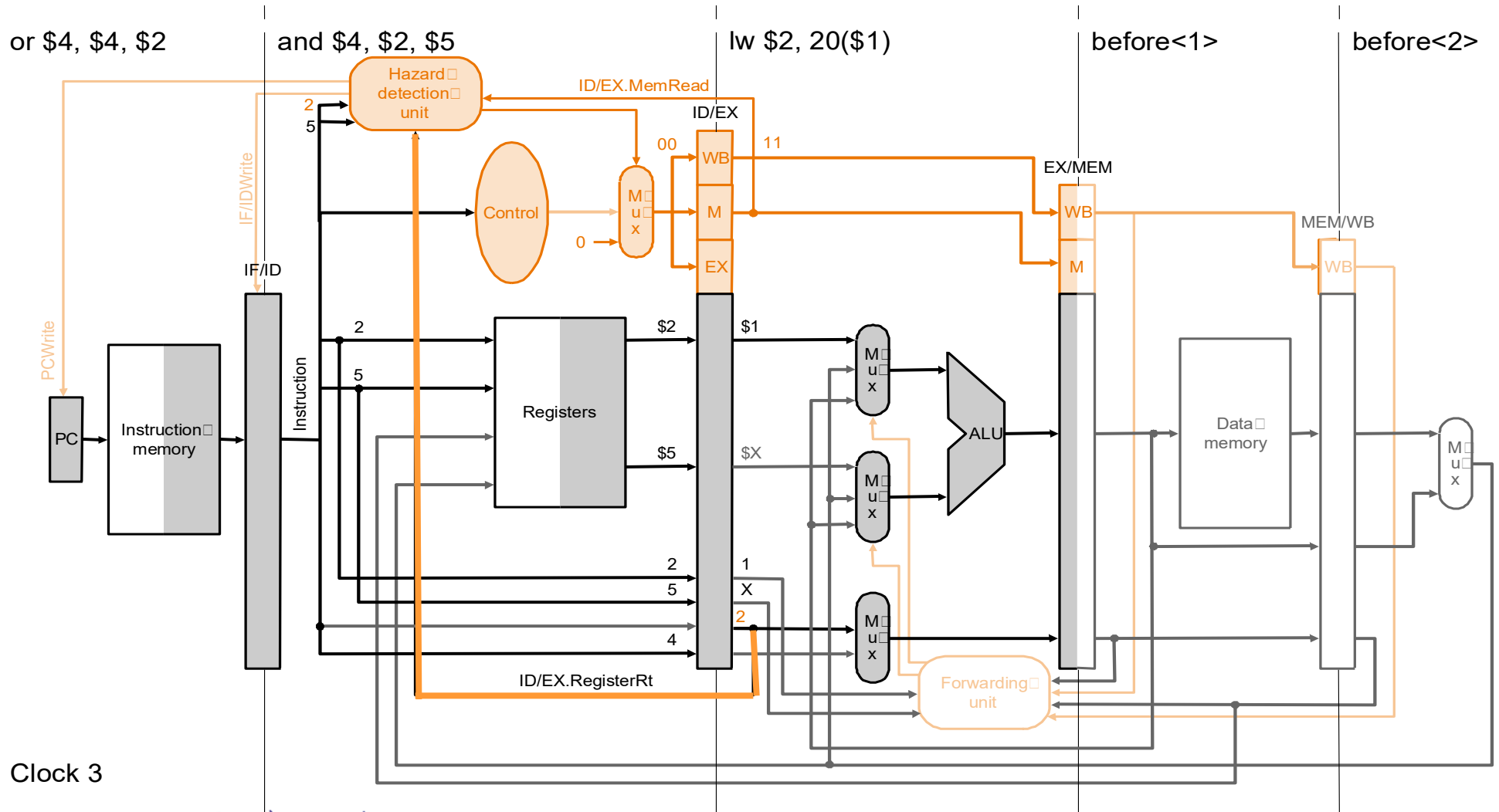
- Stall instruction in IF and ID: not change PC and IF/ID
=> the stages re-execute the instructions
- What to move into EX: insert an NOP by changing EX, MEM, WB control fields of ID/EX pipeline register to 0
 - as control signals propagate, all control signals to EX, MEM, WB are deasserted and no registers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

Example 4: Cycle 2



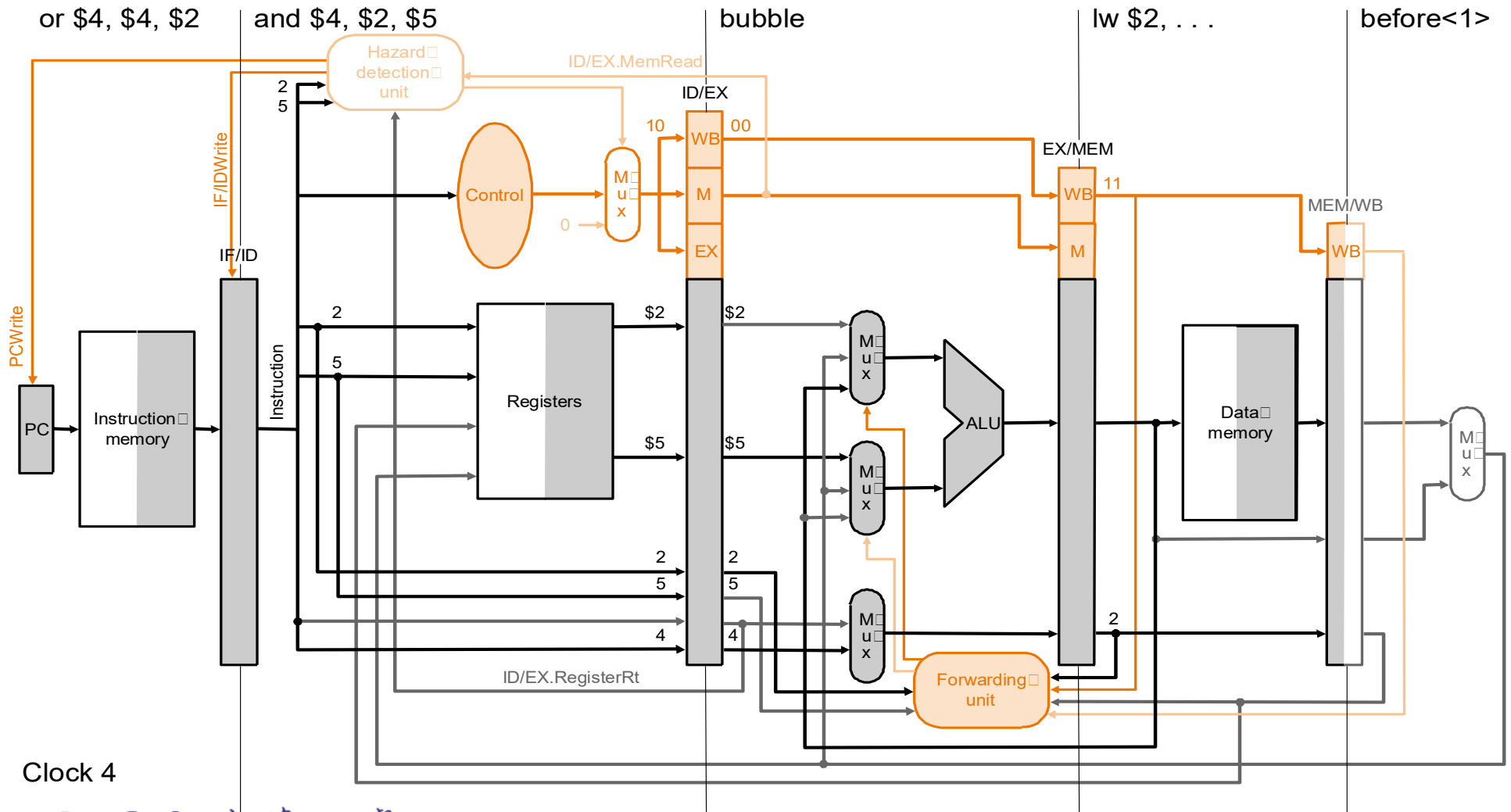
Clock 2

Example 4: Cycle 3



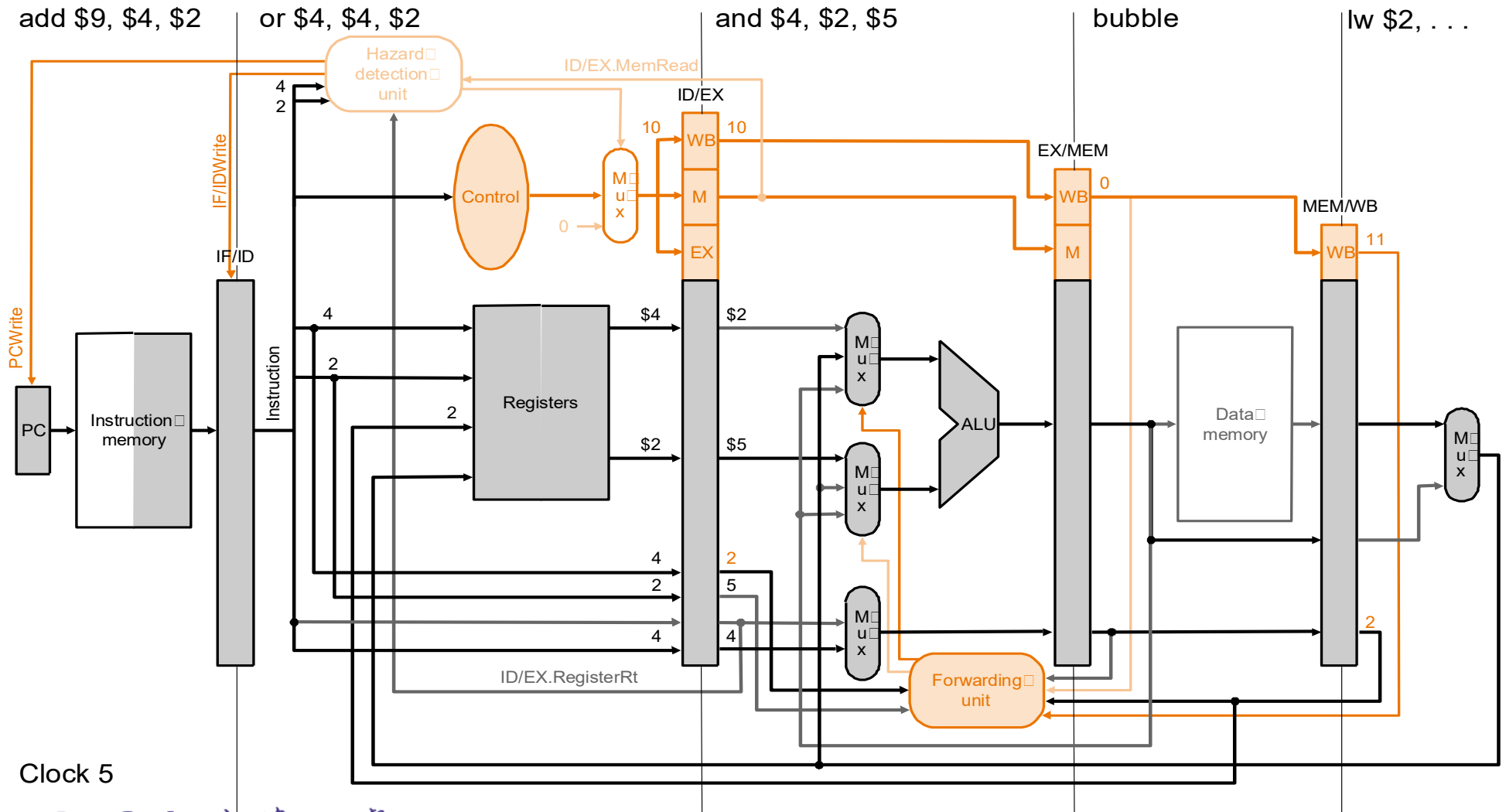
Clock 3

Example 4: Cycle 4



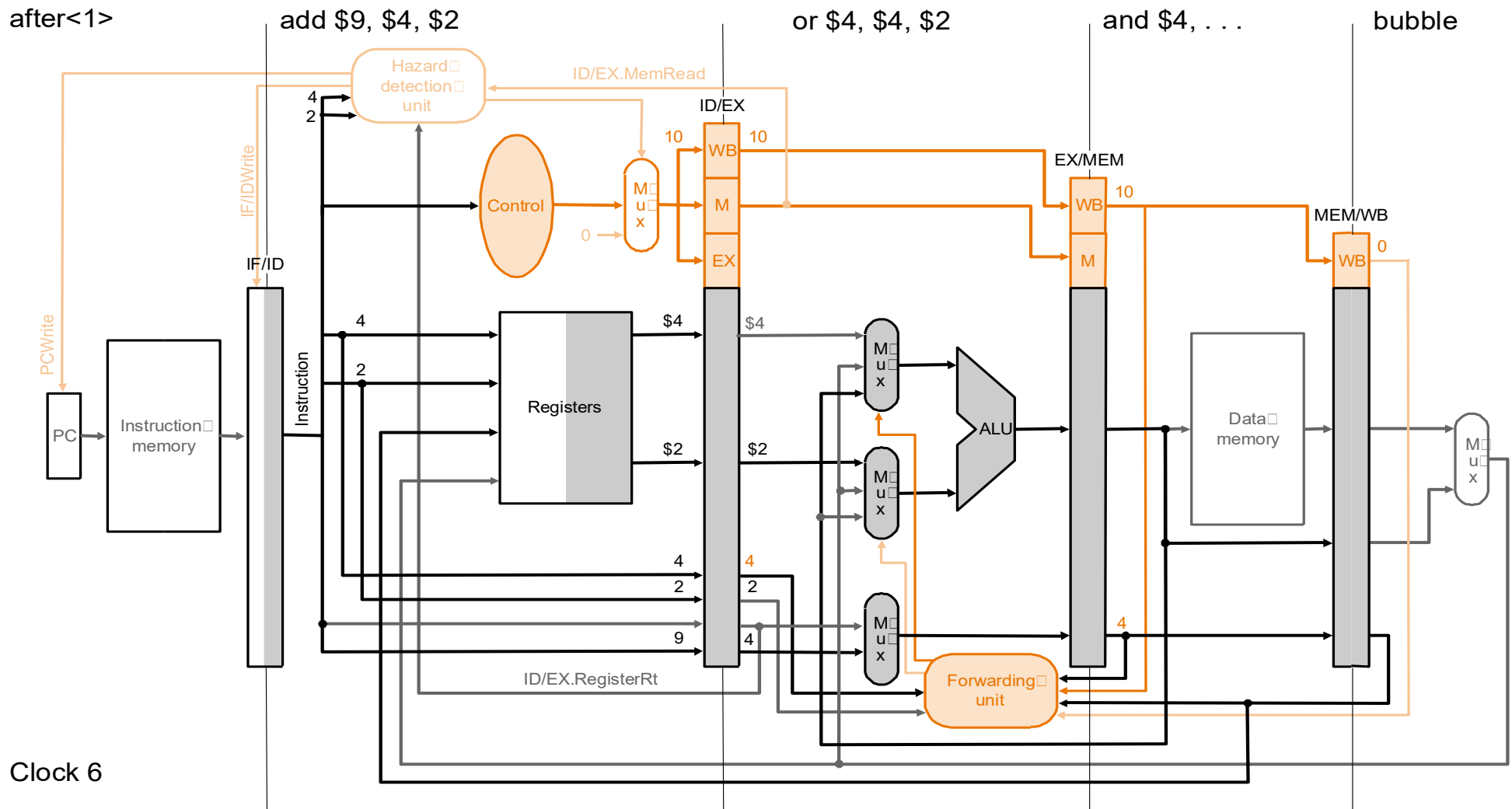
Clock 4

Example 4: Cycle 5



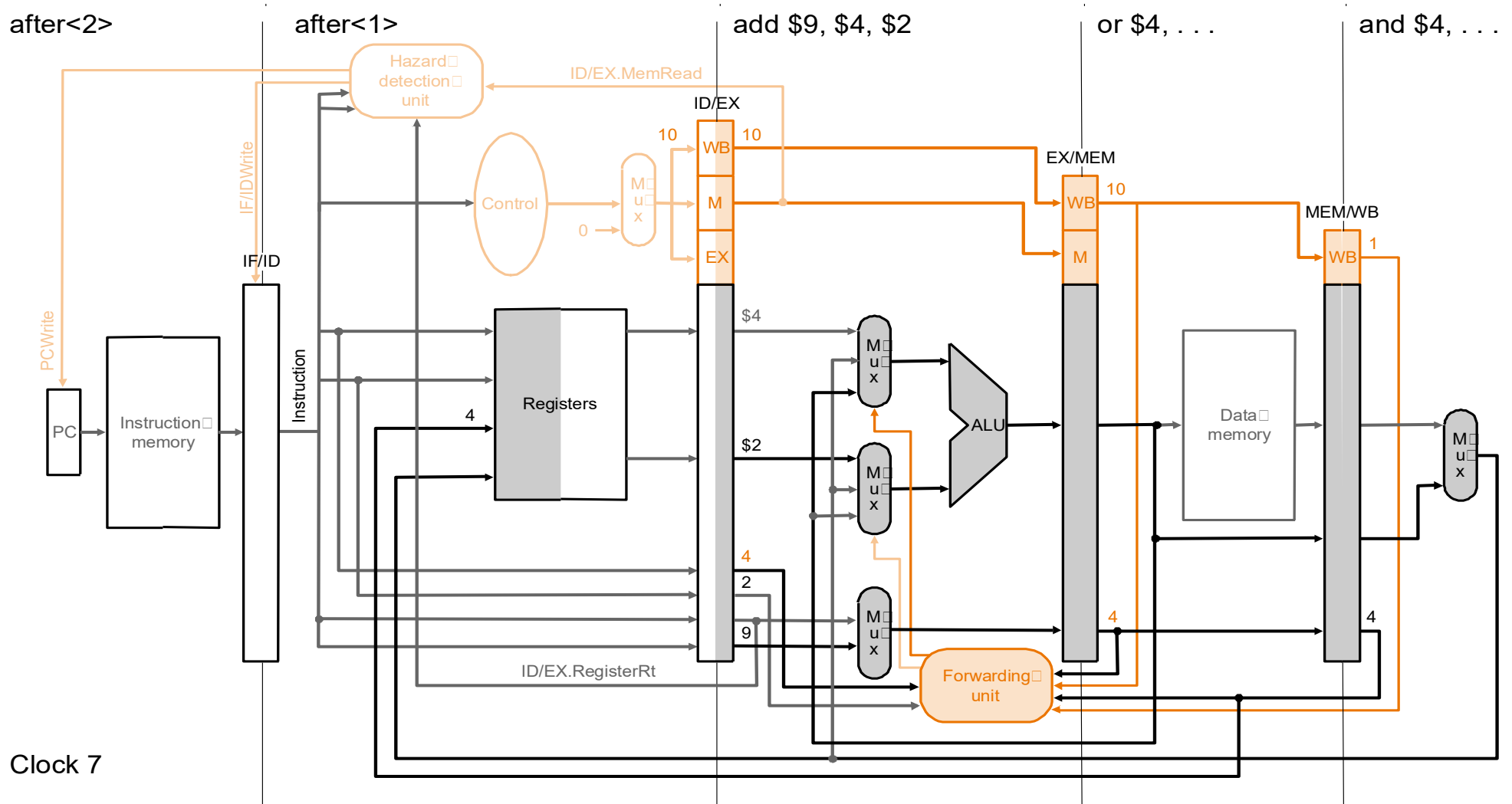
Clock 5

Example 4: Cycle 6



Clock 6

Example 4: Cycle 7

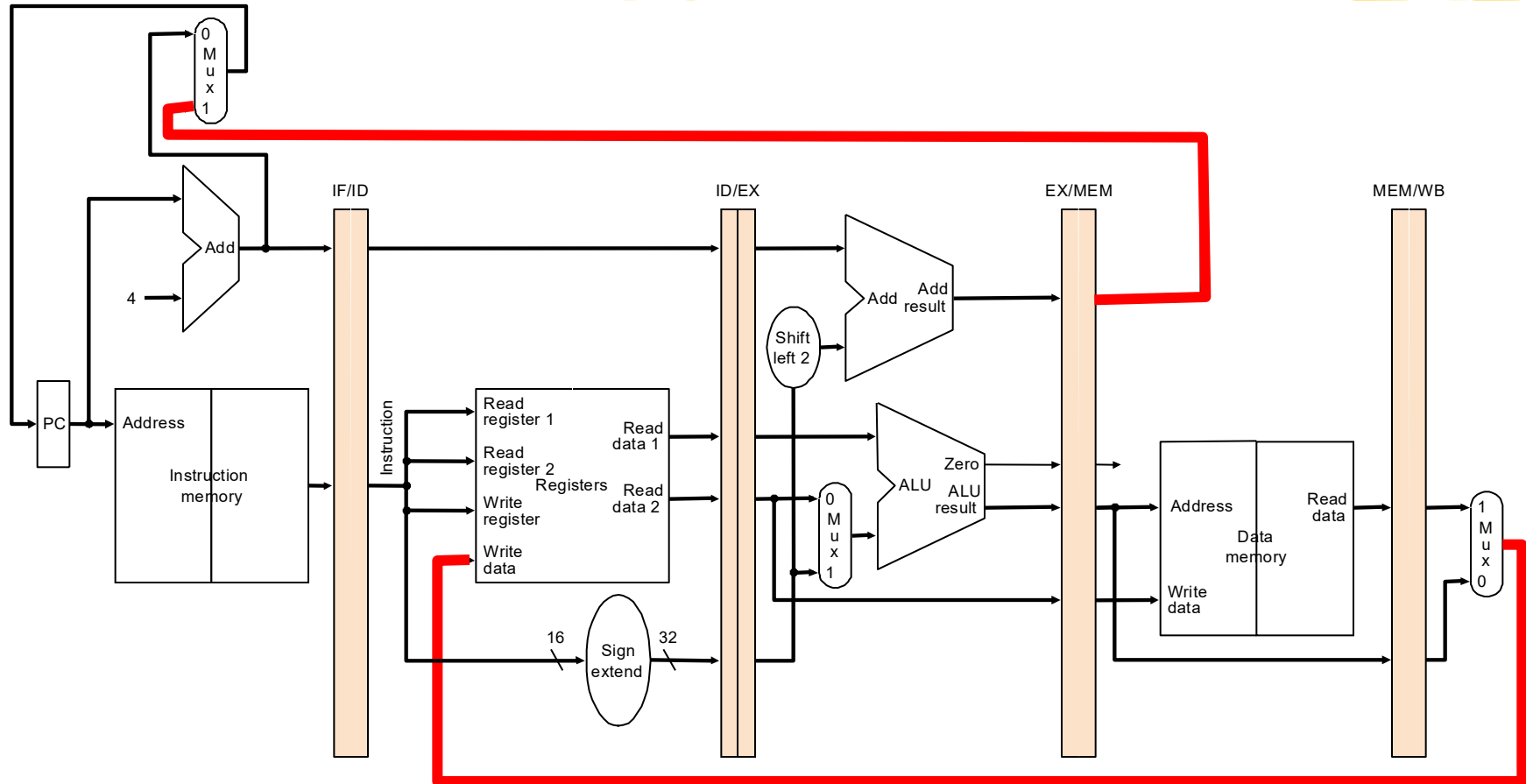


Clock 7

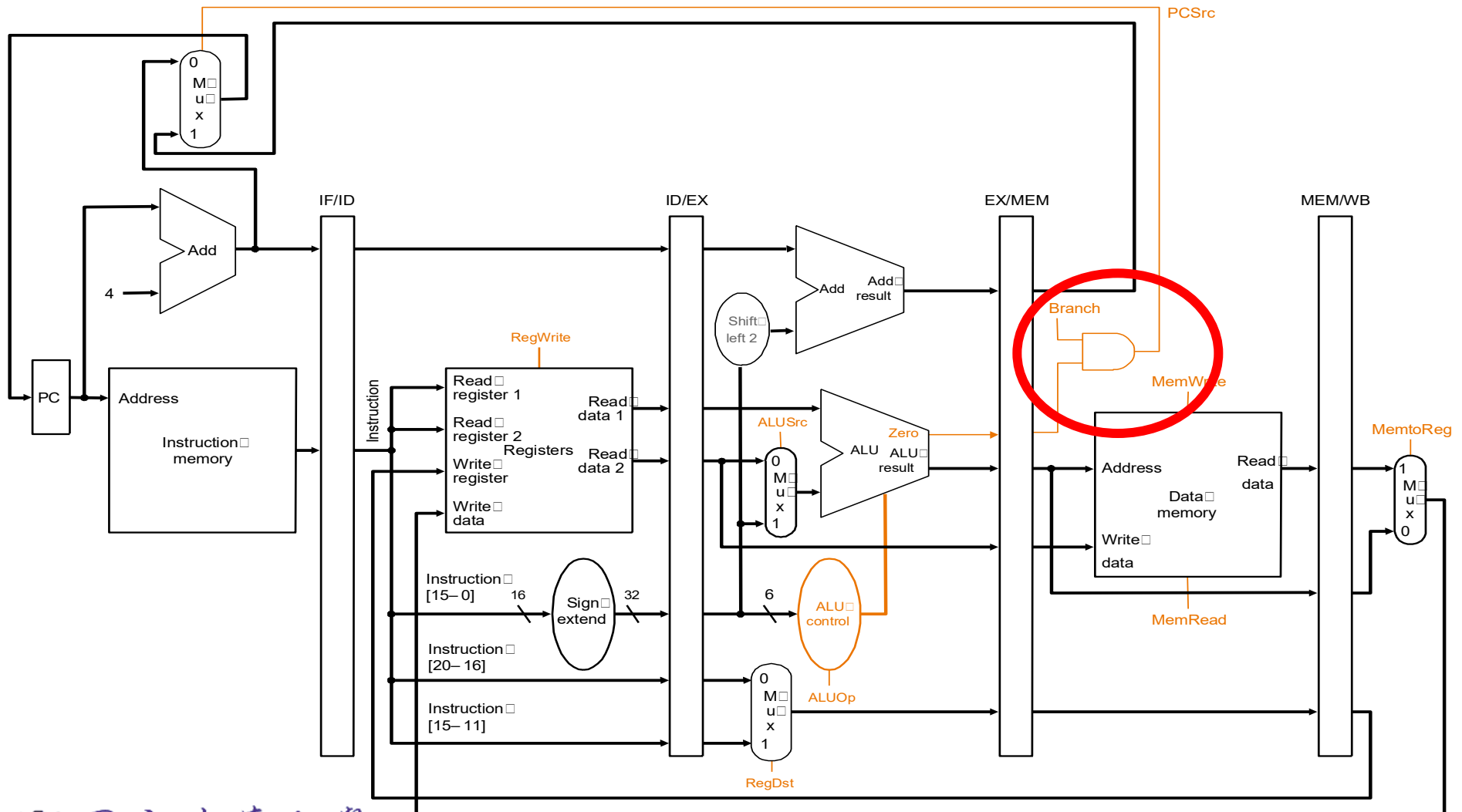
Outline

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

Feedback Path



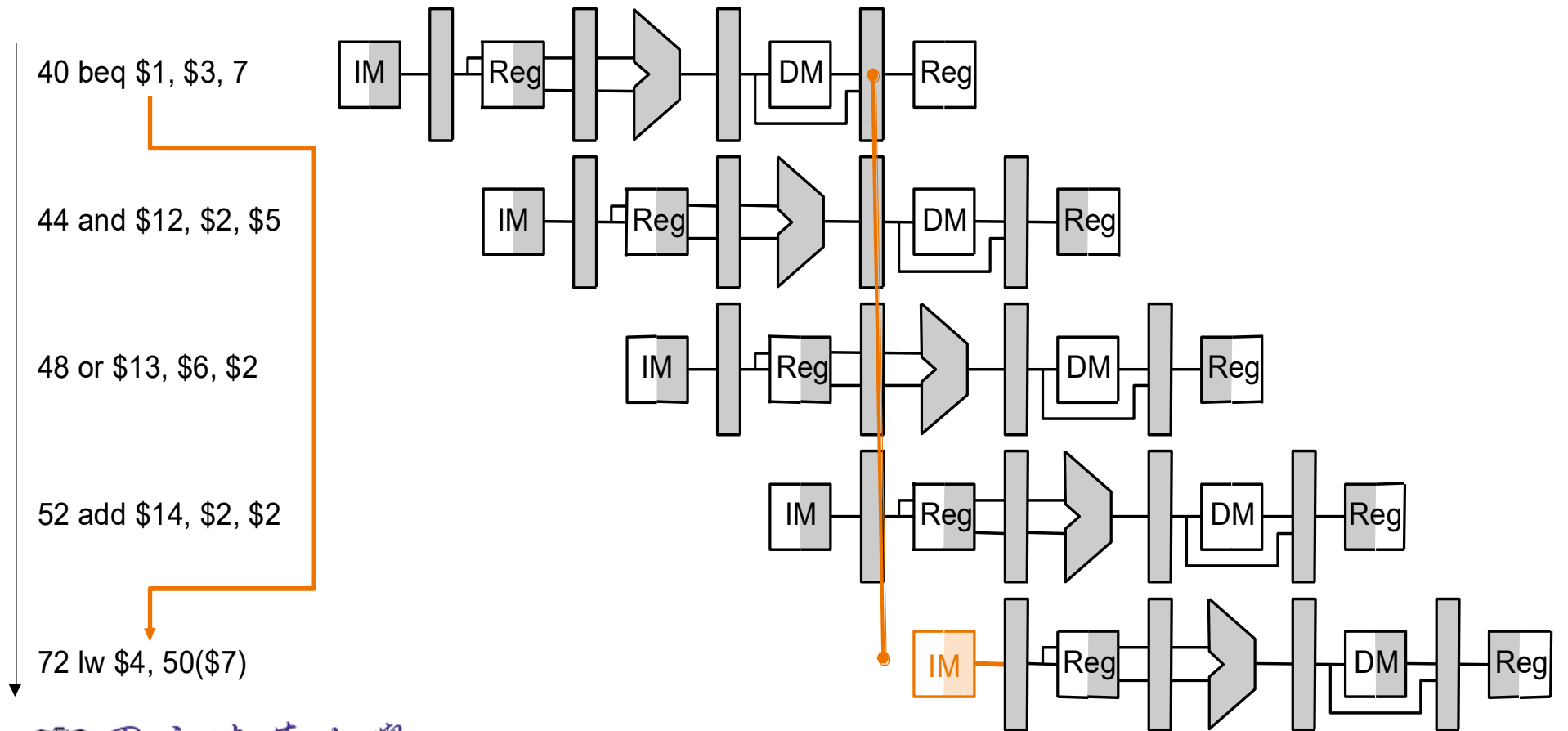
Pipeline Datapath with Control Signals




Branch Hazards

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

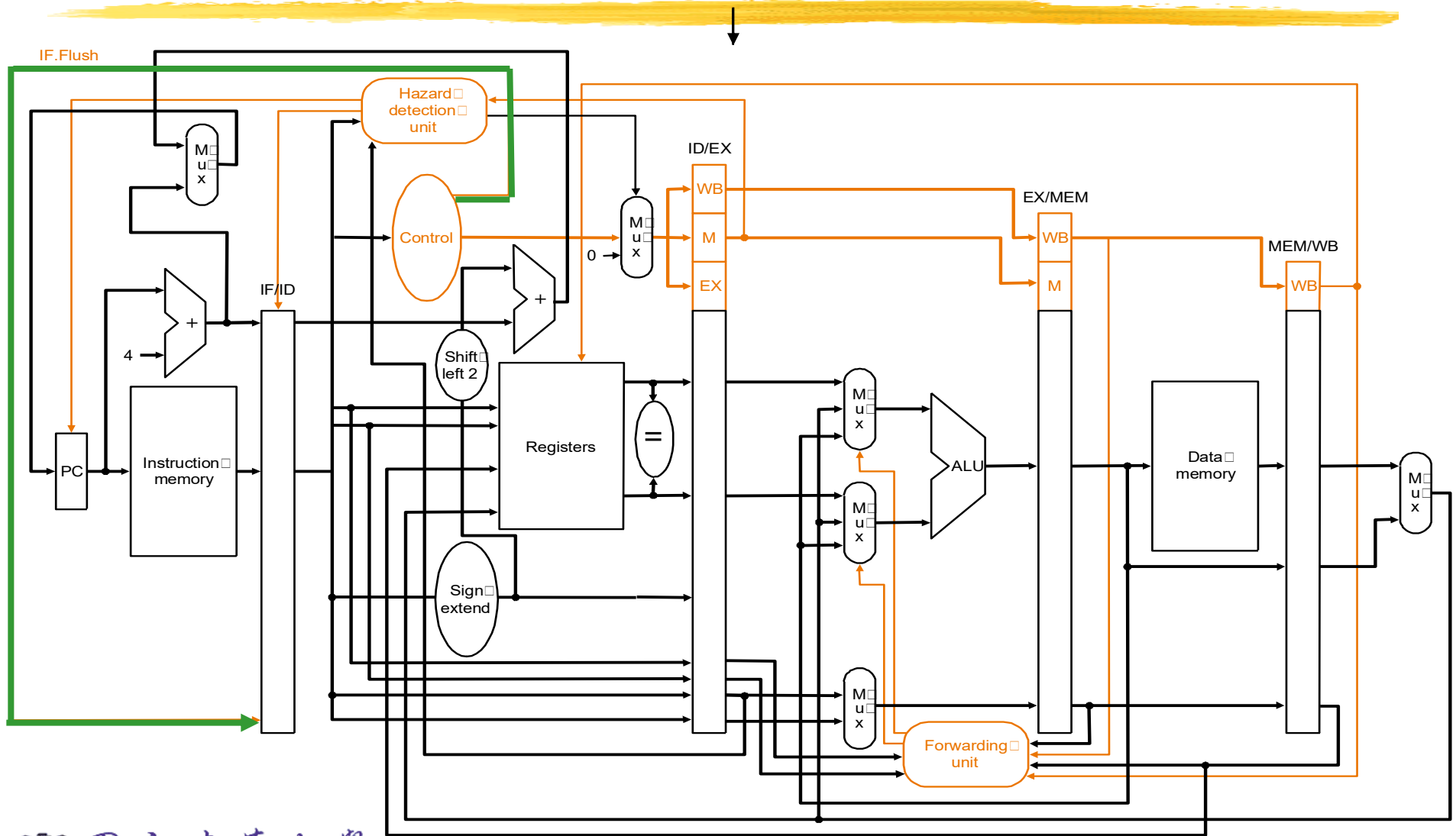
(in instructions)



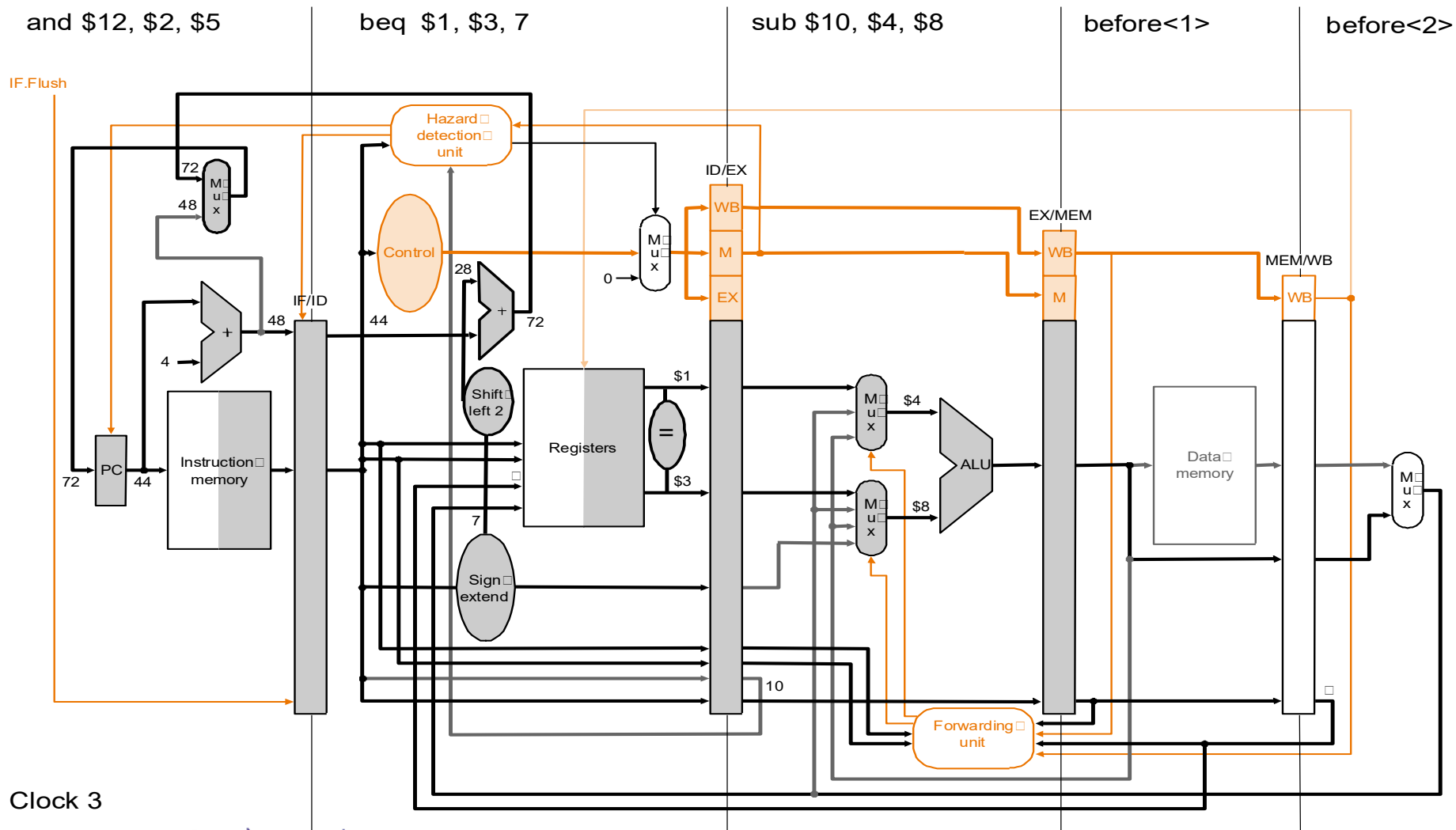
Handling Branch Hazard

- ◆ 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/EX 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

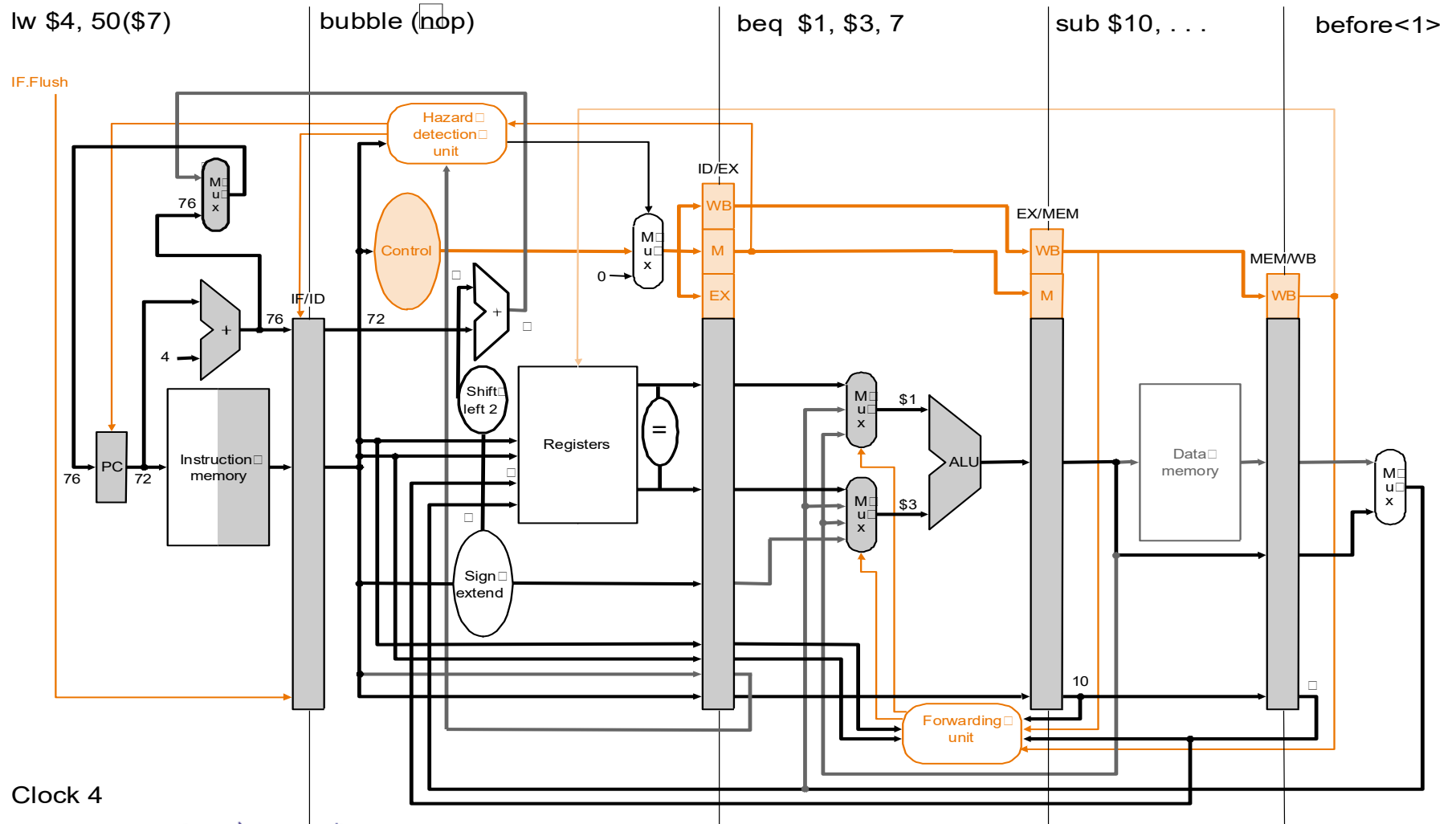


Example 5: Cycle 3



Clock 3

Example 5: Cycle 4



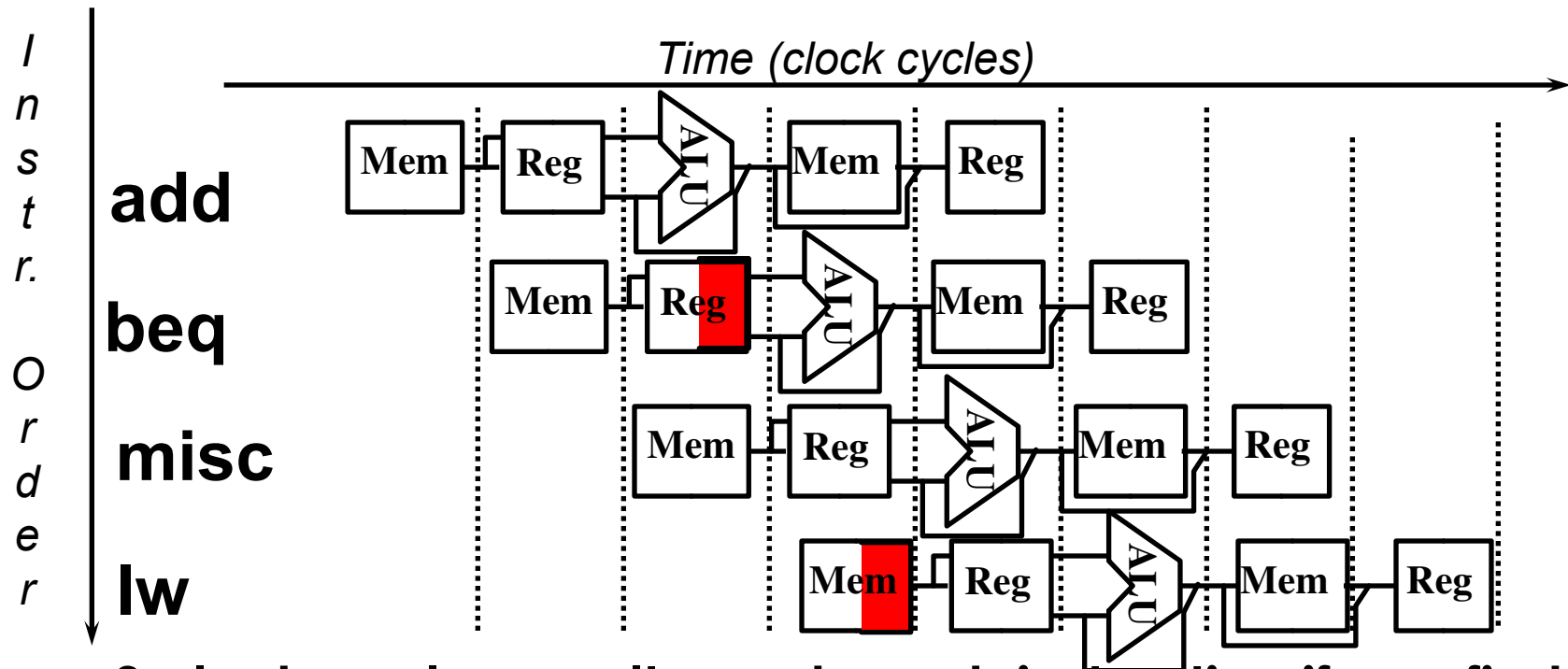
Clock 4

Dynamic Branch Prediction

- ◆ In deeper and superscalar pipelines, branch penalty is more significant
- ◆ Use dynamic prediction (e.g. loop)
 - Branch prediction buffer (aka branch history table)
 - Indexed by recent branch instruction addresses
 - Stores outcome (taken/not taken)
 - To execute a branch
 - Check table, expect the same outcome
 - Start fetching from fall-through or target
 - If wrong, flush pipeline and flip prediction

Delayed Branch

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



- 0 clock cycle penalty per branch instruction if can find instruction to put in slot ($\approx 50\%$ of time)

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

Exceptions and Interrupts

- ◆ “Unexpected” events requiring change in flow of control
 - Different ISAs use the terms differently
- ◆ Exception
 - Arises within the CPU
 - e.g., undefined opcode, overflow, syscall, ...
- ◆ Interrupt
 - From an external I/O controller
- ◆ Dealing with them without sacrificing performance is hard

Handling Exceptions

- ◆ In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- ◆ Save PC of offending (or interrupted) instruction
 - In MIPS: Exception Program Counter (EPC)
- ◆ Save indication of the problem
 - In MIPS: Cause register
 - We'll assume 1-bit
 - 0 for undefined opcode, 1 for overflow
- ◆ Jump to handler at 8000 00180

An Alternate Mechanism

- ◆ **Vectored Interrupts**

- Handler address determined by the cause

- ◆ **Example:**

- Undefined opcode: C000 0000
- Overflow: C000 0020
- ...: C000 0040

- ◆ **Instructions either**

- Deal with the interrupt, or
- Jump to real handler

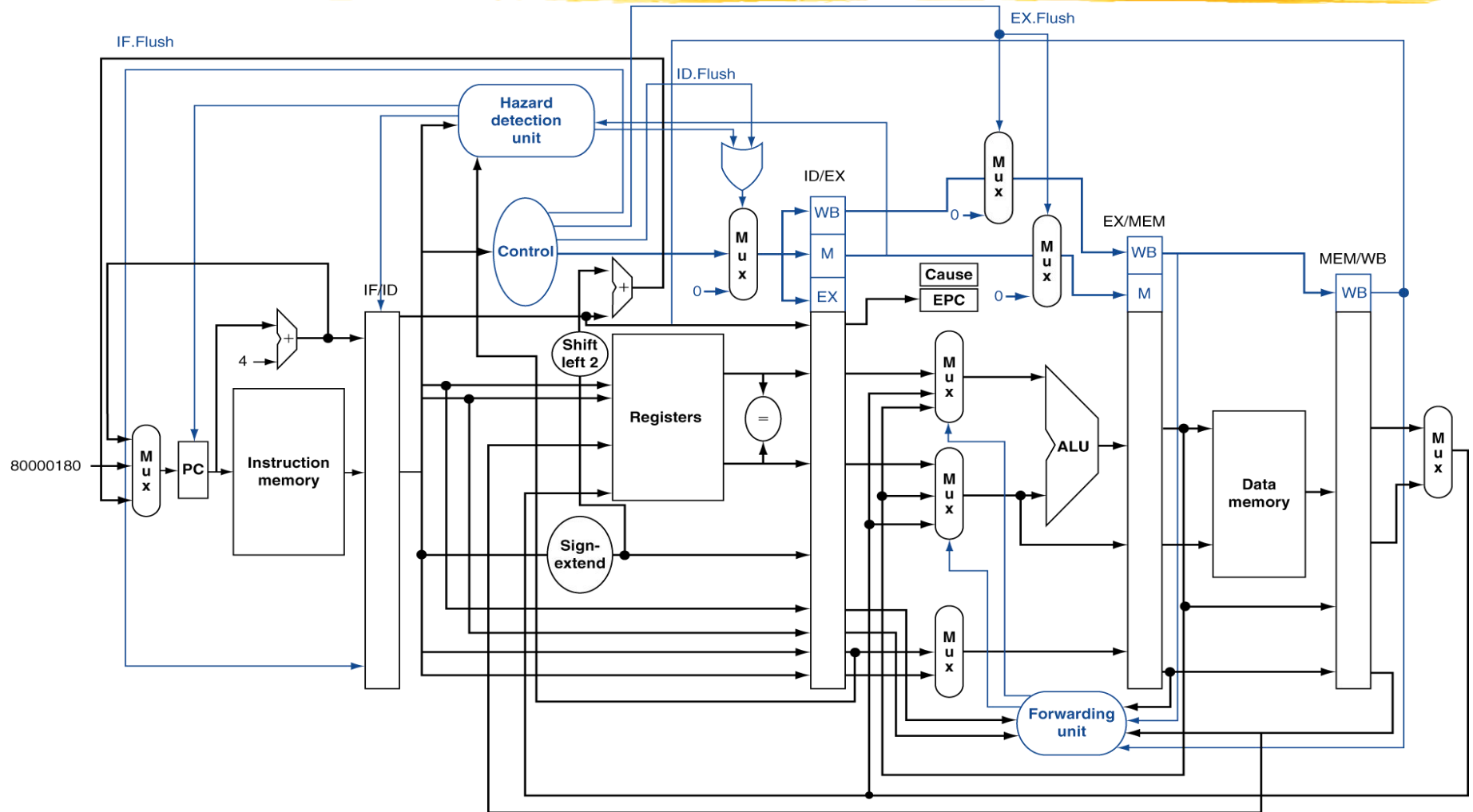
Handler Actions

- ◆ Read cause, and transfer to relevant handler
- ◆ Determine action required
- ◆ If restartable
 - Take corrective action
 - use EPC to return to program
- ◆ Otherwise
 - Terminate program
 - Report error using EPC, cause, ...

Exceptions in a Pipeline

- ◆ Another form of control hazard
- ◆ Consider overflow on add in EX stage
add \$1, \$2, \$1
 - Prevent \$1 from being clobbered
 - Complete previous instructions
 - Flush add and subsequent instructions
 - Set Cause and EPC register values
 - Transfer control to handler
- ◆ Similar to mispredicted branch
 - Use much of the same hardware

Pipeline with Exceptions



Exception Properties

- ◆ **Restartable exceptions**
 - Pipeline can flush the instruction
 - Handler executes, then returns to the instruction
 - Refetched and executed from scratch
- ◆ **PC saved in EPC register**
 - Identifies causing instruction
 - Actually PC + 4 is saved
 - Handler must adjust

Exception Example

◆ Exception on **add** in

40	sub	\$11,	\$2,	\$4
44	and	\$12,	\$2,	\$5
48	or	\$13,	\$2,	\$6
4C	add	\$1,	\$2,	\$1
50	slt	\$15,	\$6,	\$7
54	lw	\$16,	50(\$7)	

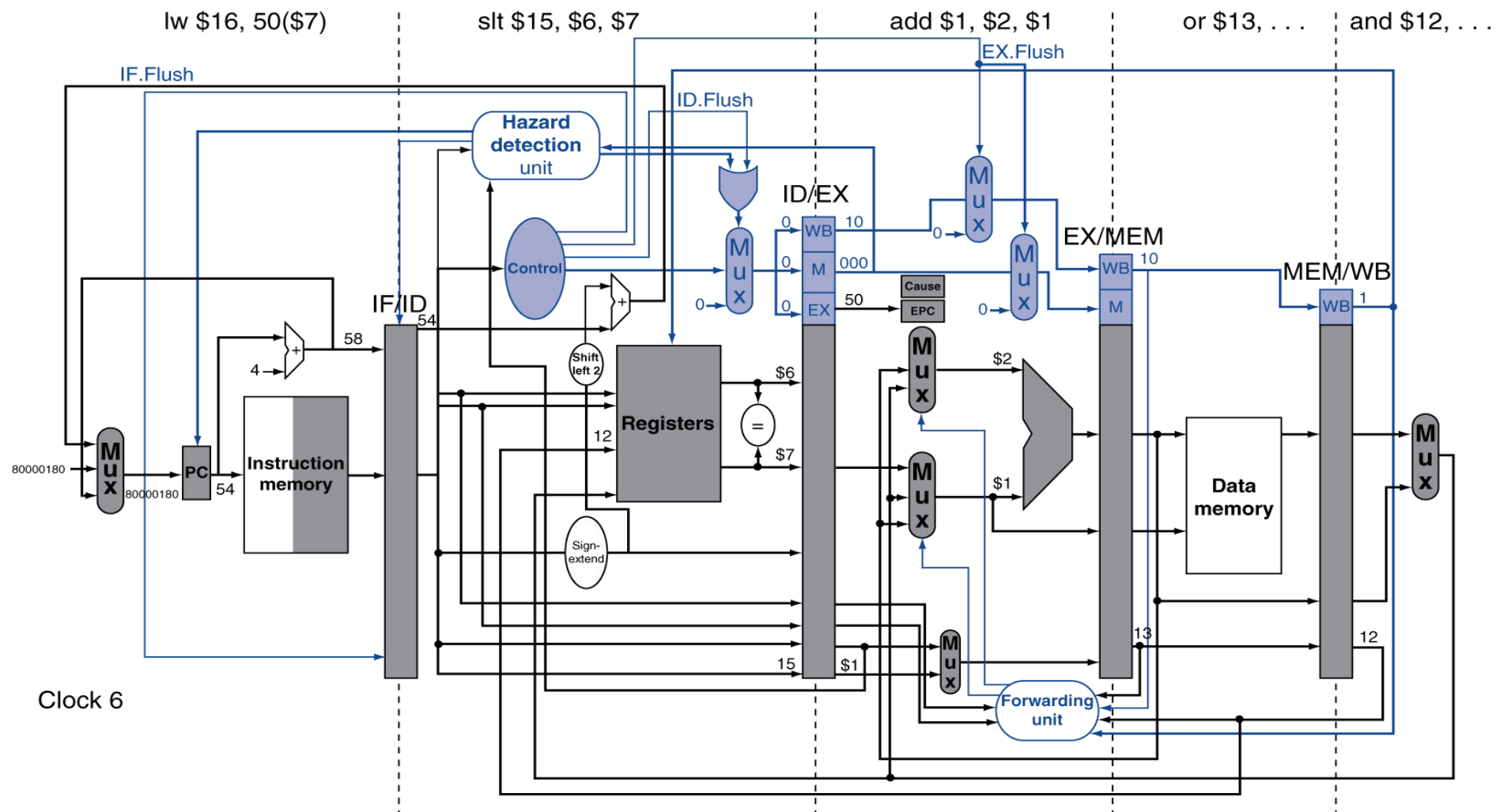
...

◆ Handler

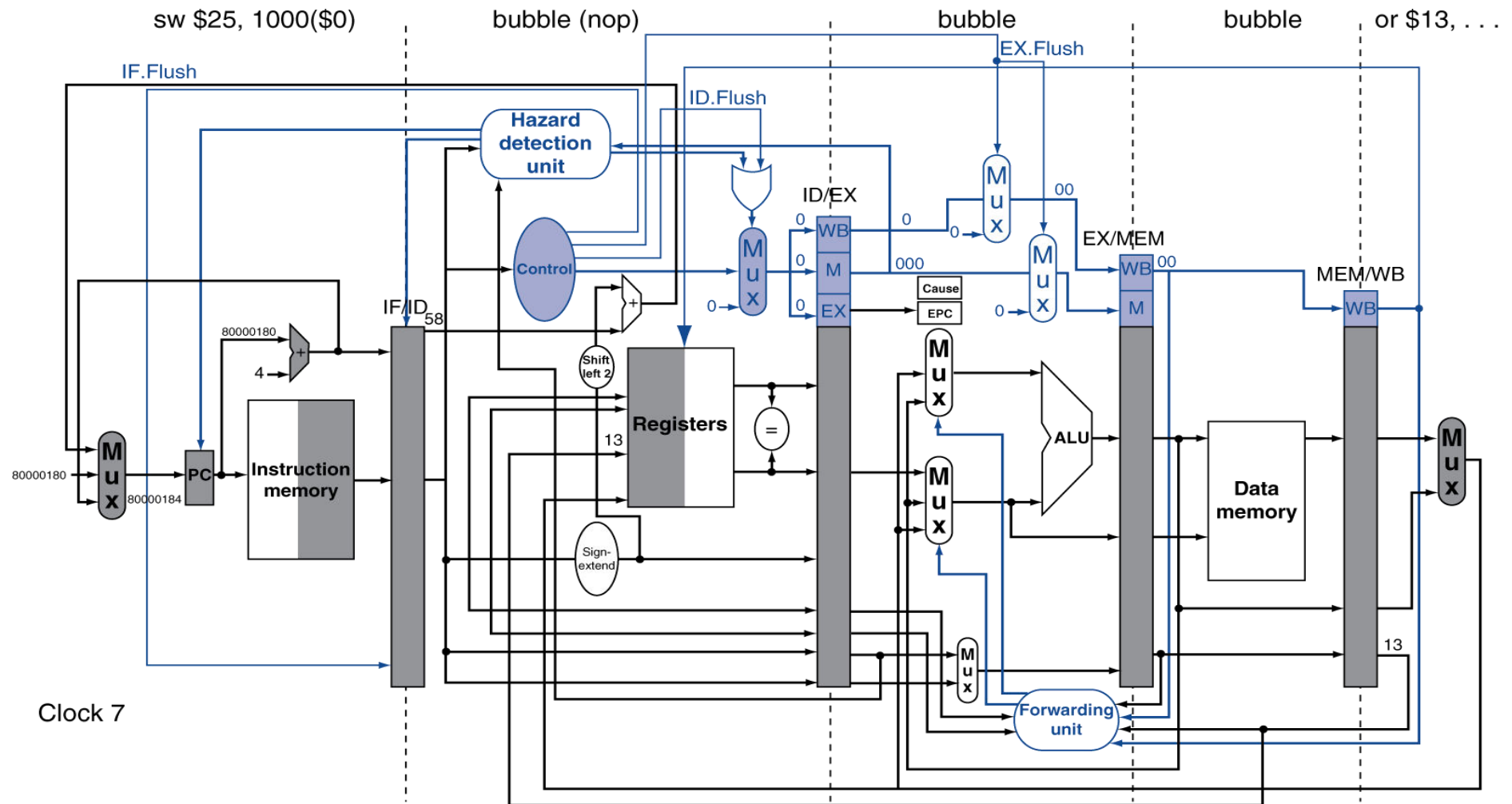
80000180	sw	\$25,	1000(\$0)
80000184	sw	\$26,	1004(\$0)

...

Exception Example



Exception Example



Multiple Exceptions

- ◆ **Pipelining overlaps multiple instructions**
 - Could have multiple exceptions at once
- ◆ **Simple approach: deal with exception from earliest instruction**
 - Flush subsequent instructions
 - “Precise” exceptions
- ◆ **In complex pipelines**
 - Multiple instructions issued per cycle
 - Out-of-order completion
 - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- ◆ Just stop pipeline and save state
 - Including exception cause(s)
- ◆ Let the handler work out
 - Which instruction(s) had exceptions
 - Which to complete or flush
 - May require “manual” completion
- ◆ Simplifies hardware, but more complex handler software
- ◆ Feasible for complex multiple-issue out-of-order pipelines

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 (ILP)

- ◆ **Pipelining: executing multiple instructions in parallel**
- ◆ **To increase ILP**
 - **Deeper pipeline**
 - Less work per stage \Rightarrow shorter clock cycle
 - **Multiple issue**
 - Replicate pipeline stages \Rightarrow multiple pipelines
 - Start multiple instructions per clock cycle
 - $CPI < 1$, so use Instructions Per Cycle (IPC)
 - E.g., 4GHz 4-way multiple-issue
 - 16 BIPS, peak $CPI = 0.25$, peak $IPC = 4$
 - But dependencies reduce this in practice

Multiple Issue

◆ Static multiple issue

- Compiler groups instructions to be issued together
- Packages them into “issue slots”
- Compiler detects and avoids hazards

◆ Dynamic multiple issue

- CPU examines instruction stream and chooses instructions to issue each cycle
- Compiler can help by reordering instructions
- CPU resolves hazards using advanced techniques at runtime

Speculation

- ◆ “Guess” what to do with an instruction
 - Start operation as soon as possible
 - Check whether guess was right
 - If so, complete the operation
 - If not, roll-back and do the right thing
- ◆ Common to static and dynamic multiple issue
- ◆ Examples
 - Speculate on branch outcome
 - Roll back if path taken is different
 - Speculate on load
 - Roll back if location is updated

Compiler/Hardware Speculation

- ◆ **Compiler can reorder instructions**
 - e.g., move load before branch
 - Can include “fix-up” instructions to recover from incorrect guess
- ◆ **Hardware can look ahead for instructions to execute**
 - Buffer results until it determines they are actually needed
 - Flush buffers on incorrect speculation

Speculation and Exceptions

- ◆ What if exception occurs on a speculatively executed instruction?
 - e.g., speculative load before null-pointer check
- ◆ Static speculation
 - Can add ISA support for deferring exceptions
- ◆ Dynamic speculation
 - Can buffer exceptions until instruction completion (which may not occur)

Static Multiple Issue

- ◆ **Compiler groups instructions into “issue packets”**
 - Group of instructions that can be issued on a single cycle
 - Determined by pipeline resources required
- ◆ **Think of an issue packet as a very long instruction**
 - Specifies multiple concurrent operations
 - \Rightarrow Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

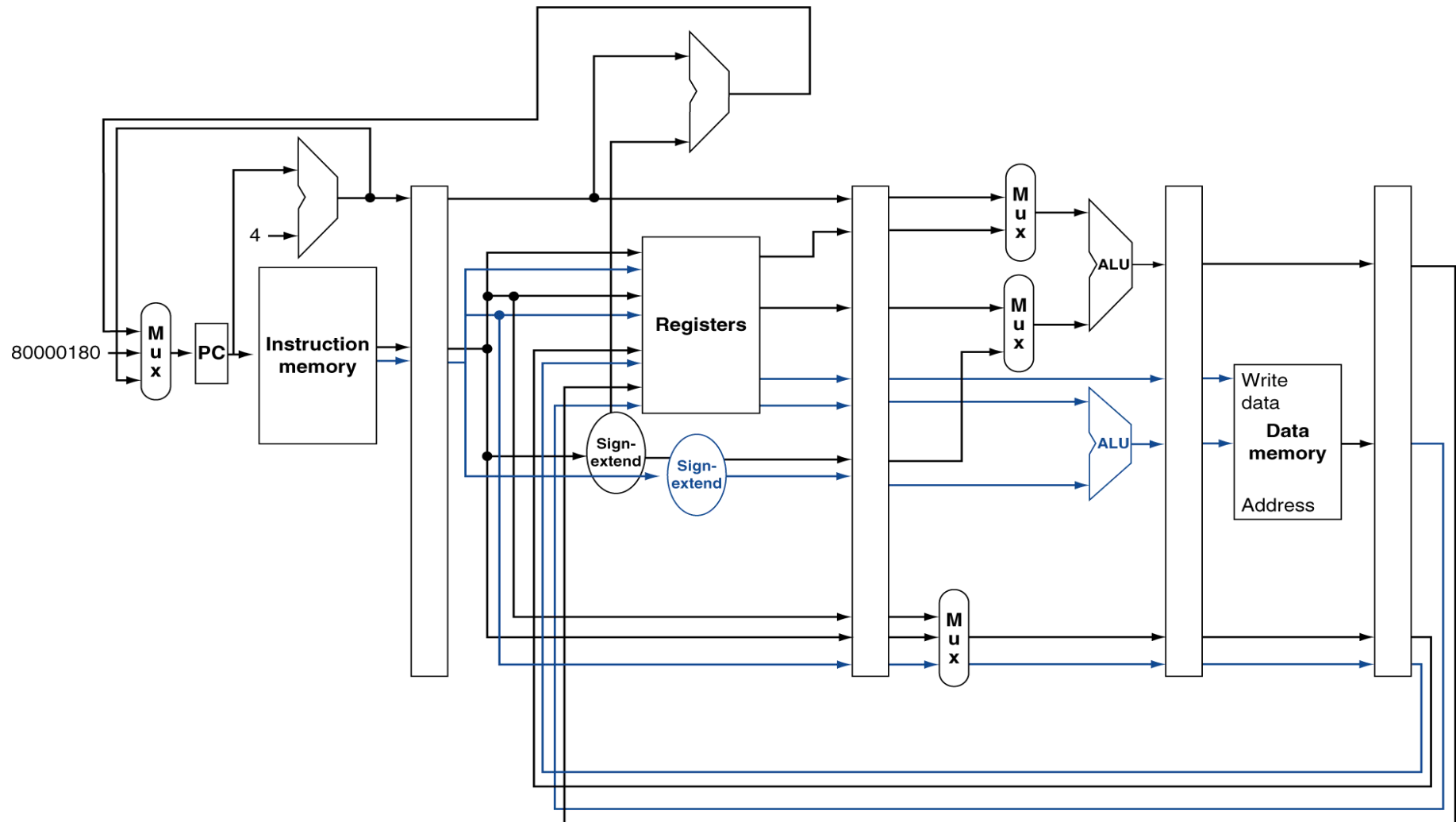
- ◆ **Compiler must remove some/all hazards**
 - Reorder instructions into issue packets
 - No dependencies within a packet
 - Possibly some dependencies between packets
 - Varies between ISAs; compiler must know!
 - Pad with nop if necessary

MIPS with Static Dual Issue

- ◆ Two-issue packets
 - One ALU/branch instruction
 - One load/store instruction
 - 64-bit aligned
 - ALU/branch, then load/store
 - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS with Static Dual Issue



Hazards in the Dual-Issue MIPS

- ◆ More instructions executing in parallel
- ◆ EX data hazard
 - Forwarding avoided stalls with single-issue
 - Now can't use ALU result in load/store in same packet
 - add **\$t0**, \$s0, \$s1
 - load \$s2, 0(**\$t0**)
 - Split into two packets, effectively a stall
- ◆ Load-use hazard
 - Still one cycle use latency, but now two instructions
- ◆ More aggressive scheduling required

Scheduling Example

◆ Schedule this for dual-issue MIPS

```
Loop: lw    $t0, 0($s1)      # $t0=array element
      addu  $t0, $t0, $s2    # add scalar in $s2
      sw    $t0, 4($s1)     # store result
      addi  $s1, $s1, -4    # decrement pointer
      bne   $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw \$t0, 0(\$s1)	1
	addi \$s1, \$s1, -4	nop	2
	addu \$t0, \$t0, \$s2	nop	3
	bne \$s1, \$zero, Loop	sw \$t0, 4(\$s1)	4

- $IPC = 5/4 = 1.25$ (c.f. peak $IPC = 2$)

Loop Unrolling

- ◆ Replicate loop body to expose more parallelism
 - Reduces loop-control overhead
- ◆ Use different registers per replication
 - Called “register renaming”
 - Avoid loop-carried “anti-dependencies”
 - Store followed by a load of the same register
 - Aka “name dependence”
 - ✧ Reuse of a register name

Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi \$s1 , \$s1, -16	lw \$t0 , 0(\$s1)	1
	nop	lw \$t1 , 4(\$s1)	2
	addu \$t0 , \$t0 , \$s2	lw \$t2 , 8(\$s1)	3
	addu \$t1 , \$t1 , \$s2	lw \$t3 , 12(\$s1)	4
	addu \$t2 , \$t2 , \$s2	sw \$t0 , 4(\$s1)	5
	addu \$t3 , \$t4 , \$s2	sw \$t1 , 8(\$s1)	6
	nop	sw \$t2 , 12(\$s1)	7
	bne \$s1 , \$zero, Loop	sw \$t3 , 16(\$s1)	8

◆ $IPC = 14/8 = 1.75$

- Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- ◆ “Superscalar” processors
- ◆ CPU decides whether to issue 0, 1, 2, ... each cycle
 - Avoiding structural and data hazards
- ◆ Avoids the need for compiler scheduling
 - Though it may still help
 - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling

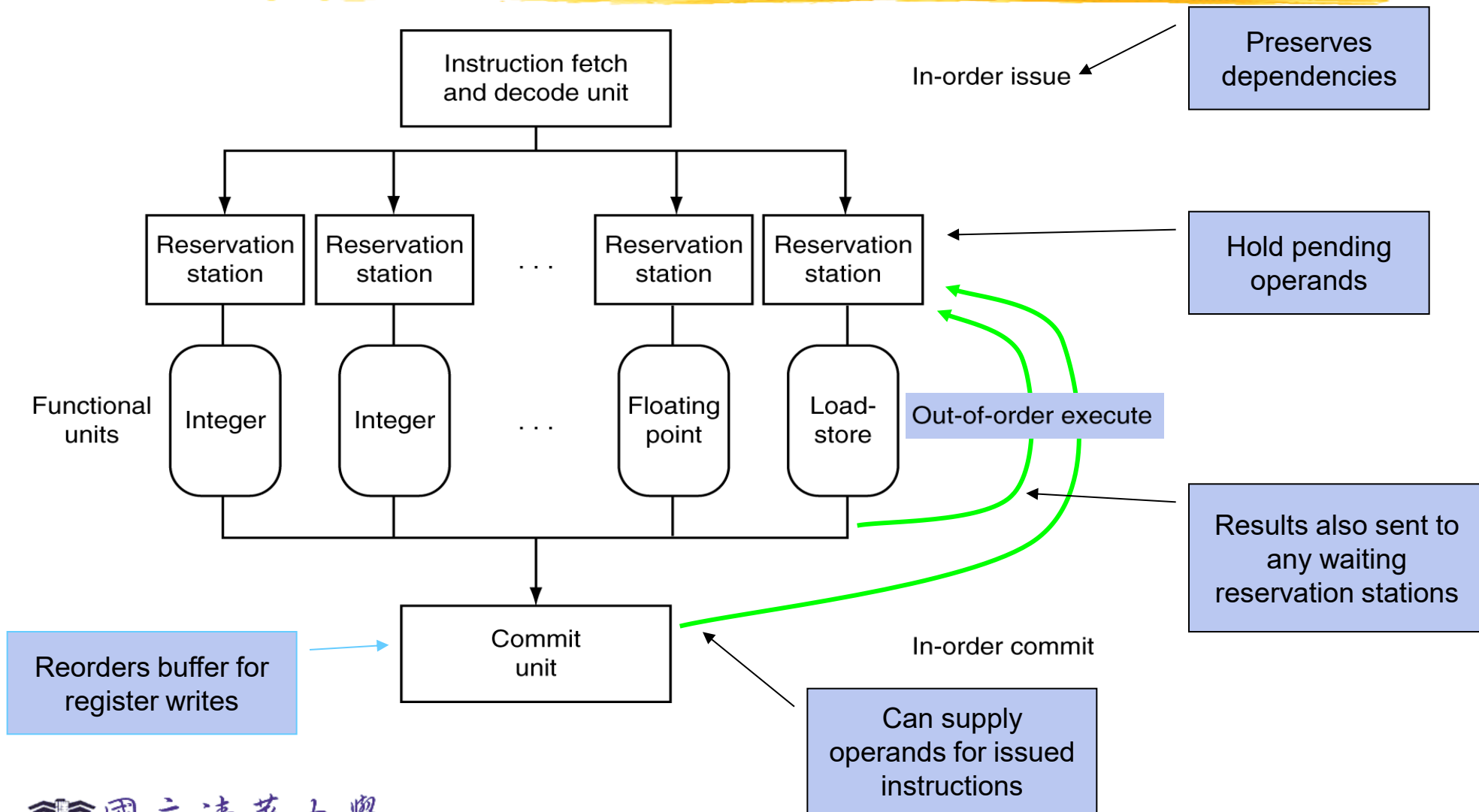
- ◆ Allow the CPU to execute instructions out of order to avoid stalls
 - But commit result to registers in order

- ◆ Example

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub      $s4, $s4, $t3
slli    $t5, $s4, 20
```

- Can start sub while addu is waiting for lw

Dynamically Scheduled CPU



Register Renaming

- ◆ Reservation stations and reorder buffer effectively provide register renaming
- ◆ On instruction issue to reservation station
 - If operand is available in register file or reorder buffer
 - Copied to reservation station
 - No longer required in the register; can be overwritten
 - If operand is not yet available
 - It will be provided to the reservation station by a function unit
 - Register update may not be required

Speculation

- ◆ Predict branch and continue issuing
 - Don't commit until branch outcome determined
- ◆ Load speculation
 - Avoid load and cache miss delay
 - Predict the effective address
 - Predict loaded value
 - Load before completing outstanding stores
 - Bypass stored values to load unit
 - Don't commit load until speculation cleared

Why Do Dynamic Scheduling?

- ◆ Why not just let the compiler schedule code?
- ◆ Not all stalls are predictable
 - e.g., cache misses
- ◆ Can't always schedule around branches
 - Branch outcome is dynamically determined
- ◆ Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

The BIG Picture

- ◆ Yes, but not as much as we'd like
- ◆ Programs have real dependencies that limit ILP
- ◆ Some dependencies are hard to eliminate
 - e.g., pointer aliasing
- ◆ Some parallelism is hard to expose
 - Limited window size during instruction issue
- ◆ Memory delays and limited bandwidth
 - Hard to keep pipelines full
- ◆ Speculation can help if done well

Power Efficiency

- ◆ Complexity of dynamic scheduling and speculations requires power
- ◆ Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

Fallacies

- ◆ **Pipelining is easy (!)**
 - The basic idea is easy
 - The devil is in the details
 - e.g., detecting data hazards
- ◆ **Pipelining is independent of technology**
 - So why haven't we always done pipelining?
 - More transistors make more advanced techniques feasible
 - Pipeline-related ISA design needs to take account of technology trends
 - e.g., predicted instructions

Pitfalls

- ◆ **Poor ISA design can make pipelining harder**
 - e.g., complex instruction sets (VAX, IA-32)
 - Significant overhead to make pipelining work
 - IA-32 micro-op approach
 - e.g., complex addressing modes
 - Register update side effects, memory indirection
 - e.g., delayed branches
 - Advanced pipelines have long delay slots

Concluding Remarks

- ◆ ISA influences design of datapath and control
- ◆ Datapath and control influence design of ISA
- ◆ Pipelining improves instruction throughput using parallelism
 - More instructions completed per second
 - Latency for each instruction not reduced
- ◆ Hazards: structural, data, control
- ◆ Multiple issue and dynamic scheduling (ILP)
 - Dependencies limit achievable parallelism
 - Complexity leads to the power wall