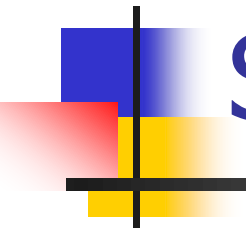
Chapter4- III

The Processor: Datapath and Control (Enhancing Performance with Pipelining)

臺大電機系

吳安宇教授

2018/05/19 v1



Brief Summary of Single-cycle MIPS CPU

Summary of MIPS Instructions

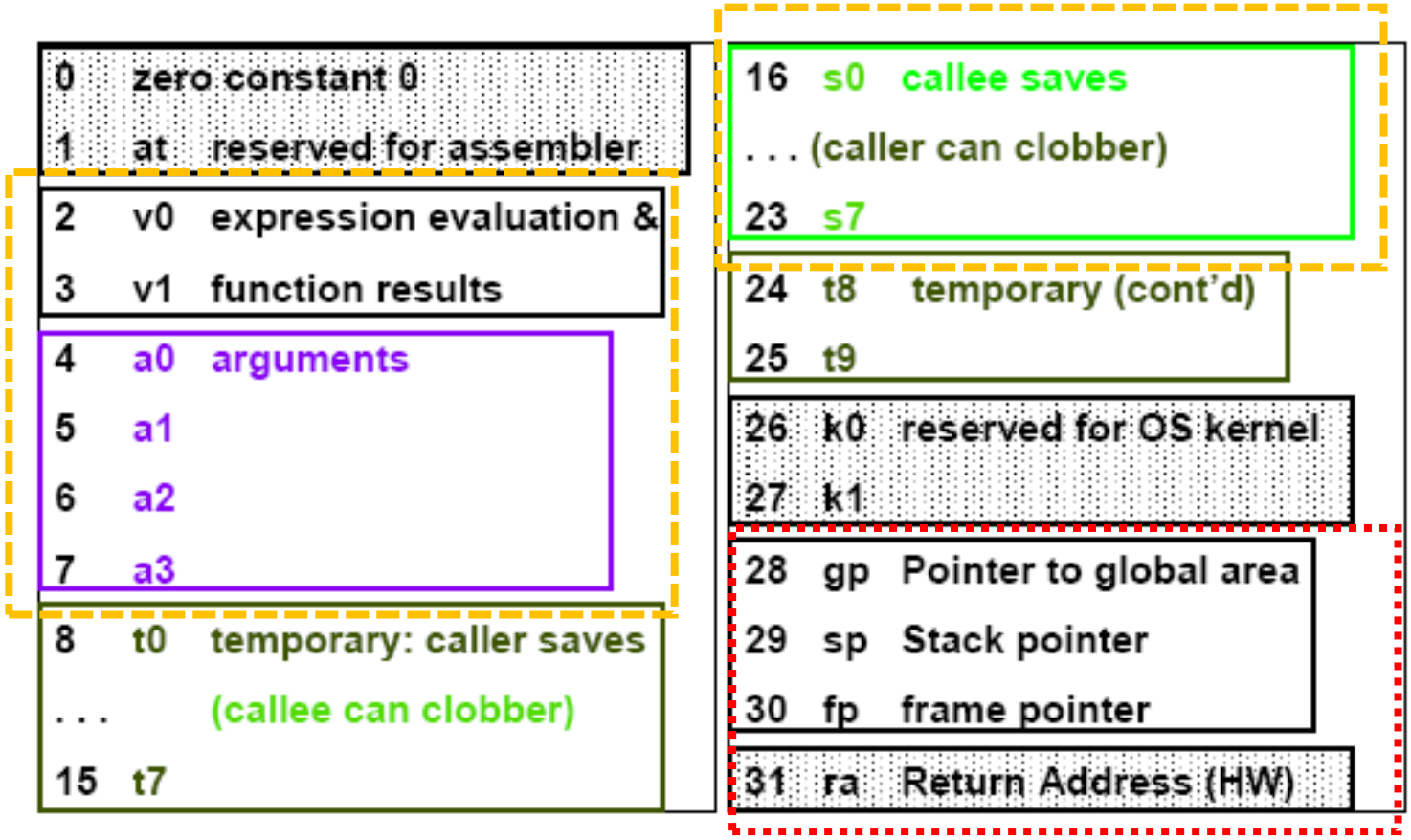
MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

MIPS Register Convention



MIPS machine language

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
and	R	0	18	19	17	0	36	and $s1,$s2,$s3
or	R	0	18	19	17	0	37	or $s1,$s2,$s3
nor	R	0	18	19	17	0	39	nor $s1,$s2,$s3
andi	I	12	18	17	100			andi $s1,$s2,100
ori	I	13	18	17	100			ori $s1,$s2,100
sll	R	0	0	18	17	10	0	sll $s1,$s2,10
srl	R	0	0	18	17	10	2	srl $s1,$s2,10
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

Showing Branch Offset in Machine Language

The *while* loop on page 107–108 was compiled into this MIPS assembler code:

```
80000  Loop: sll    $t1,$s3,2   # Temp reg $t1 = 4 * i
80004        add  $t1,$t1,$s6   # $t1 = address of save[i]
80008        lw   $t0,0($t1)    # Temp reg $t0 = save[i]
80012        bne  $t0,$s5,Exit  # go to Exit if save[i] ≠ k
80016        addi $s3,$s3,1     # i = i + 1
80020        j    Loop          # go to Loop
80024  Exit:
```

Location address vs Machine code!

The assembled instructions and their addresses are:

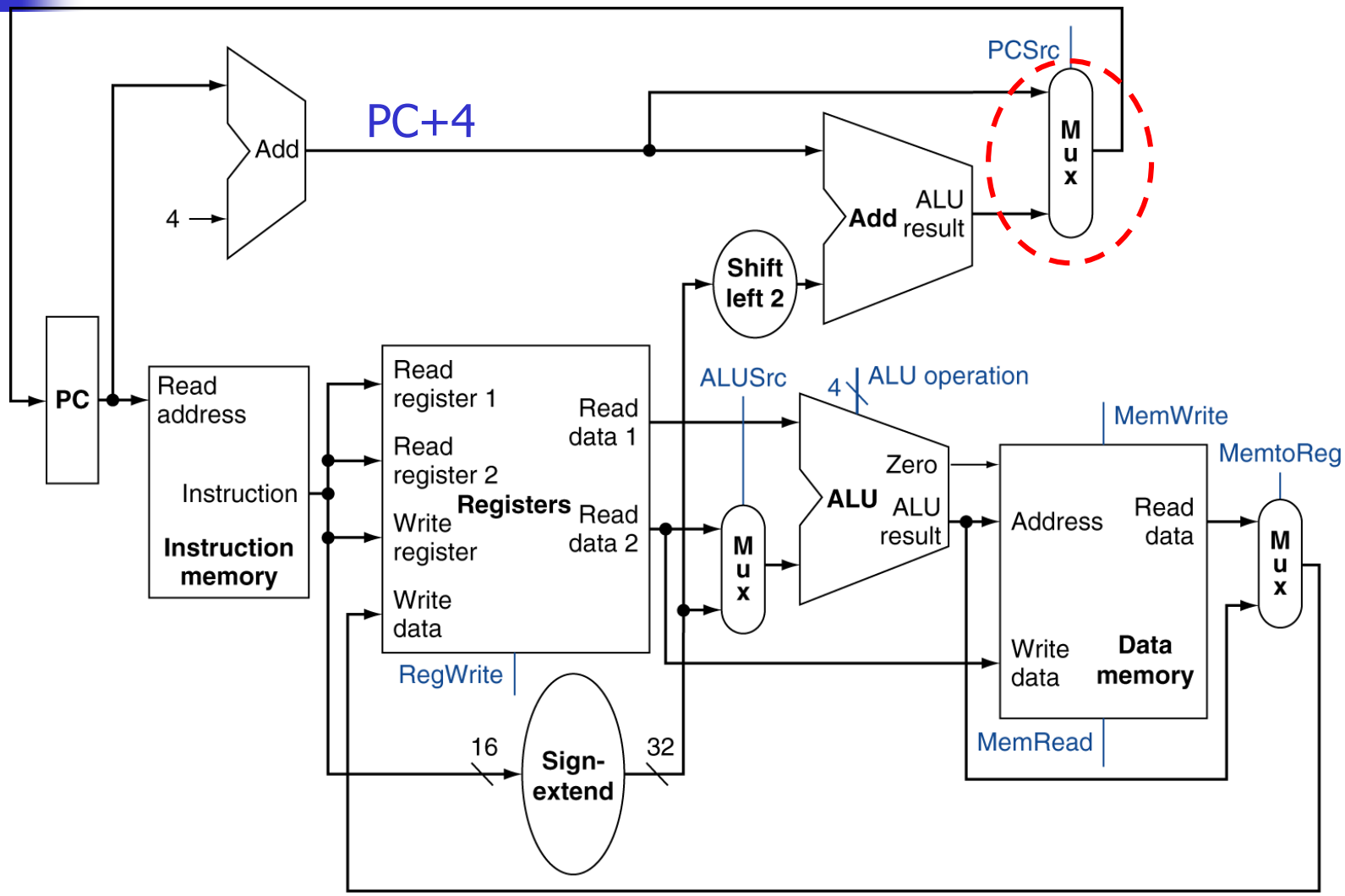
Starting address of the program

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

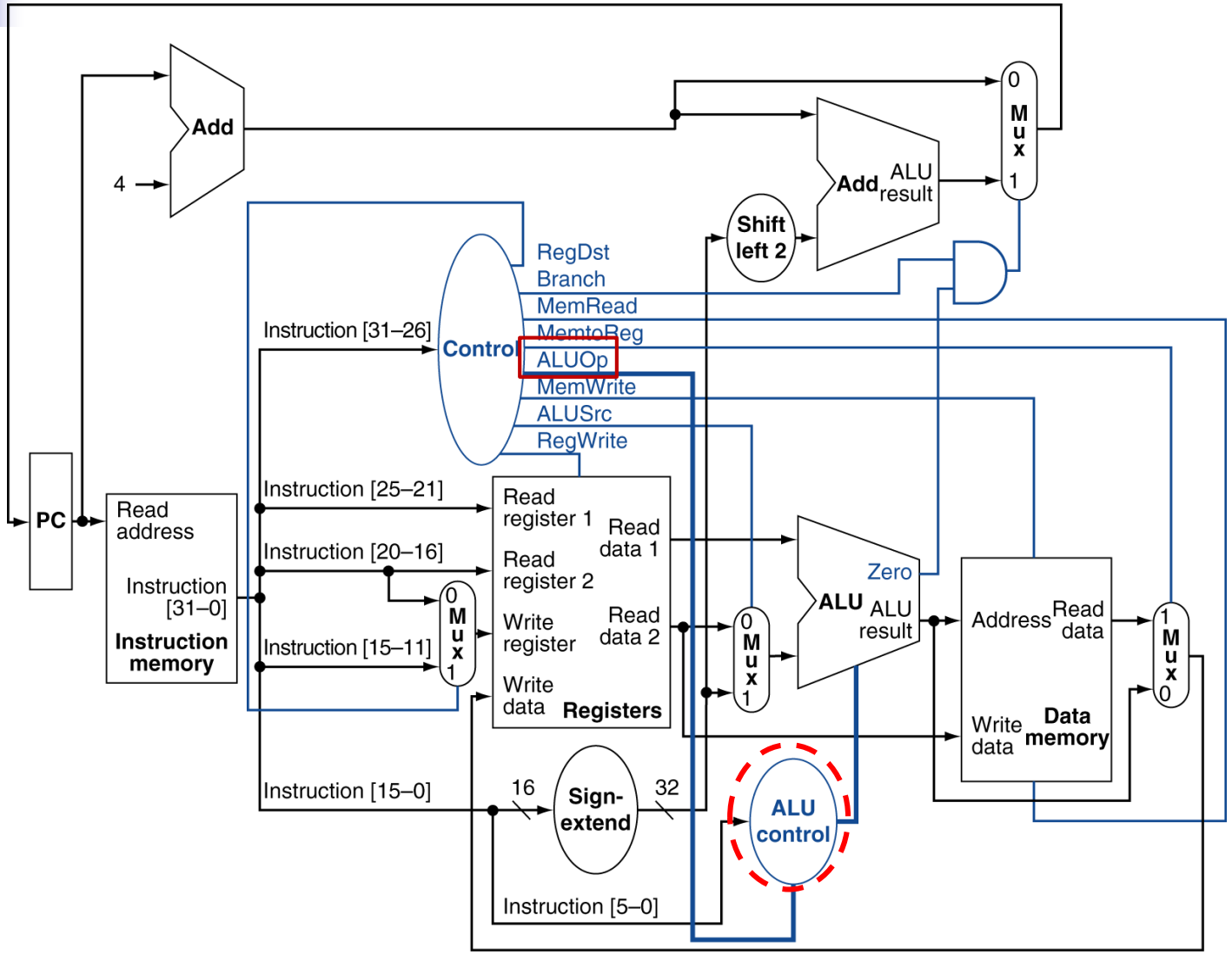
(24-16)/4

Loop=80000 (/4)

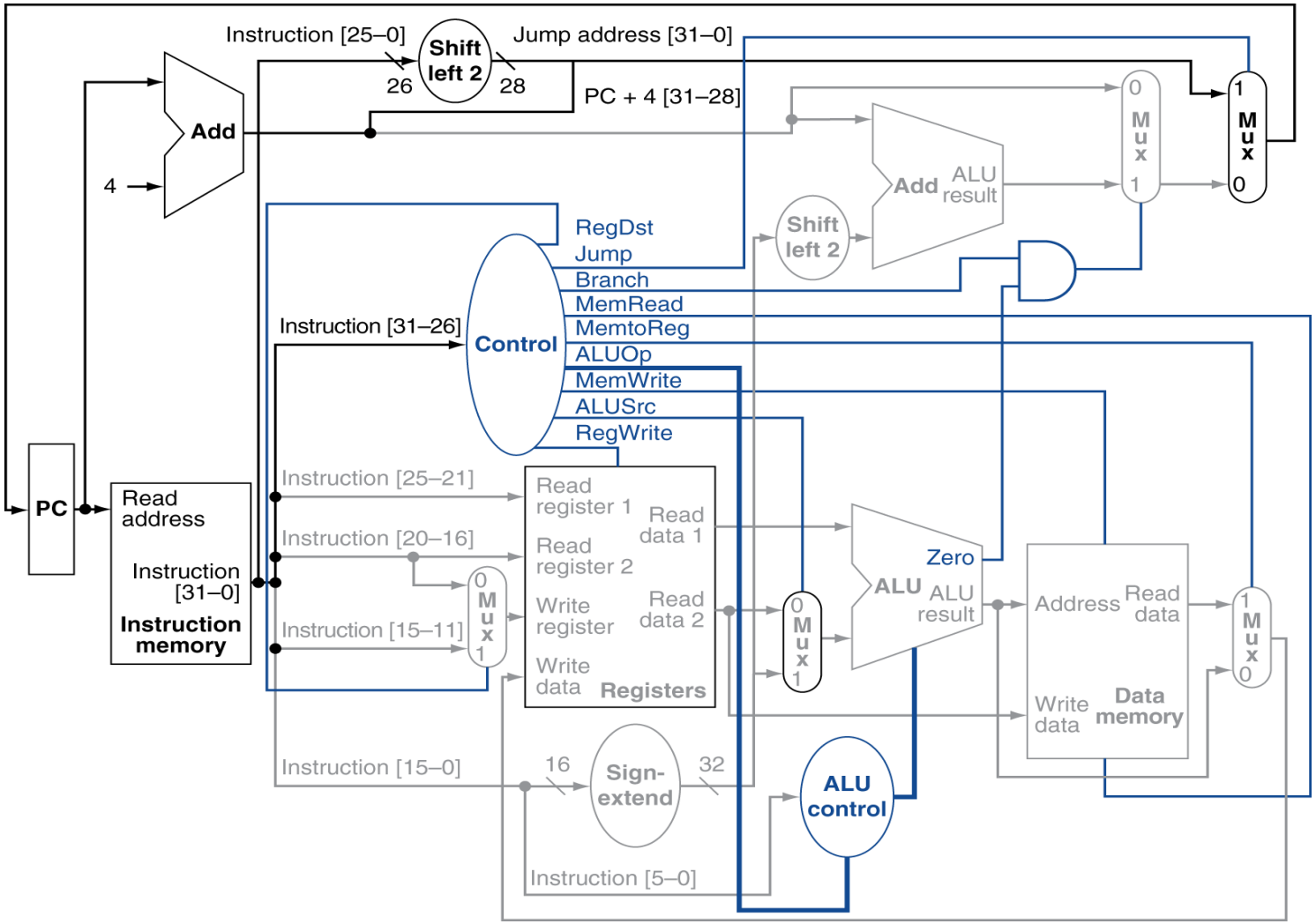
Simple Datapath for All three types of Instructions



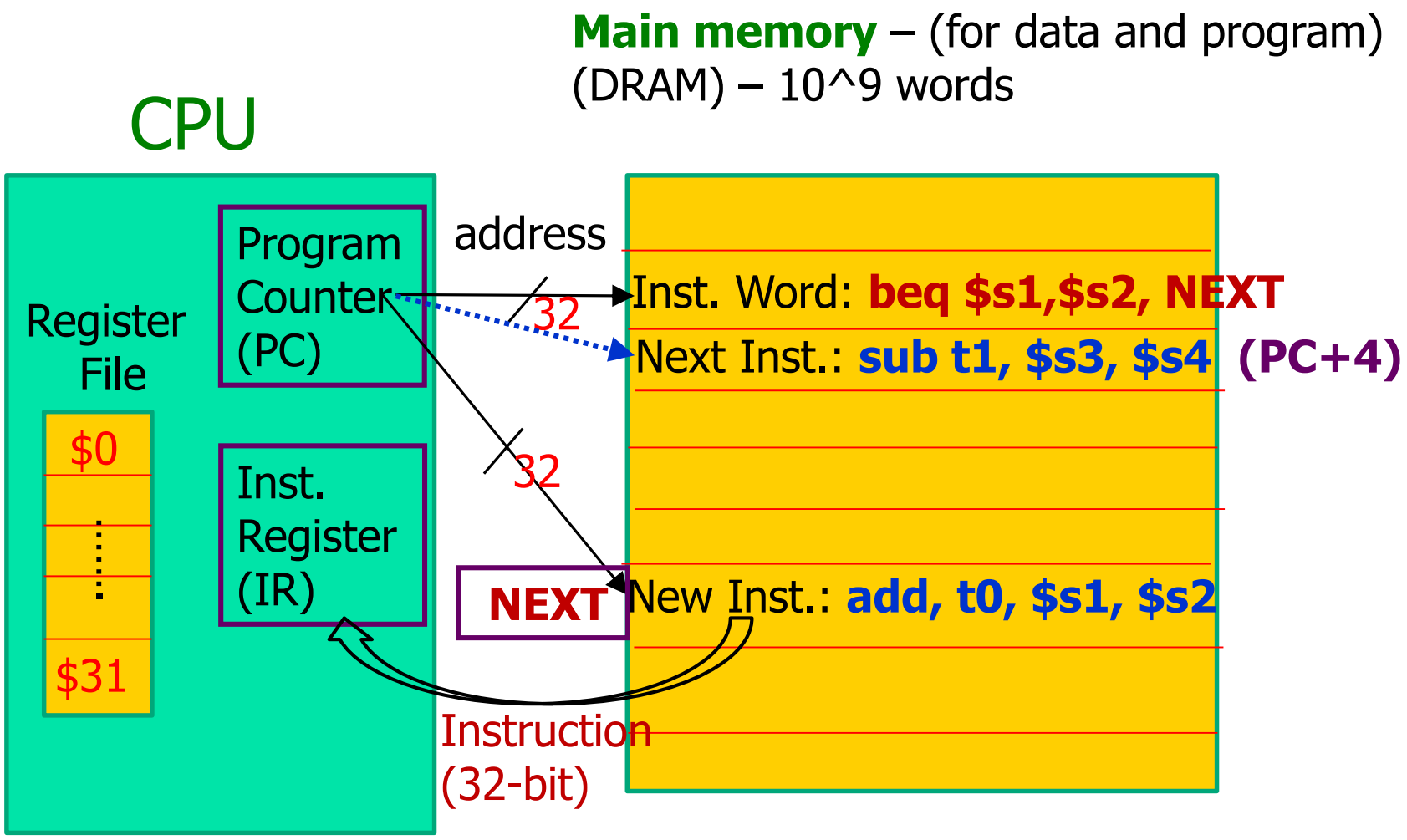
Basic Datapath with Control Signals



Single-cycle MIPS with 4 Instructions (R-type (add/sub), lw/sw, beq, J)



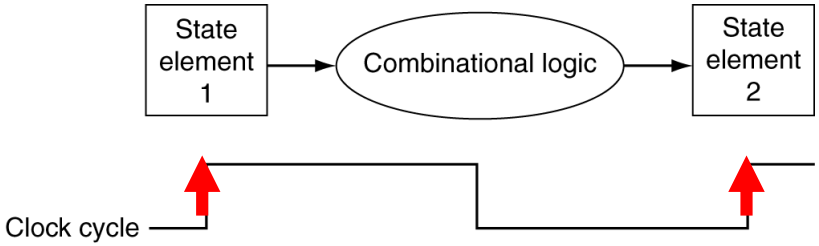
Addressing mode of Branch (Beq)

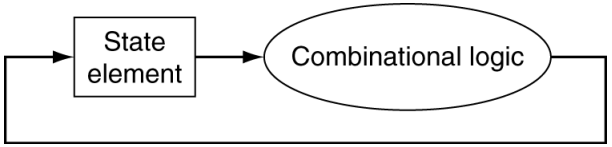


Clocking (recall)

- Combinational logic transforms data during clock cycles
  - An **Edge-triggered** methodology
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

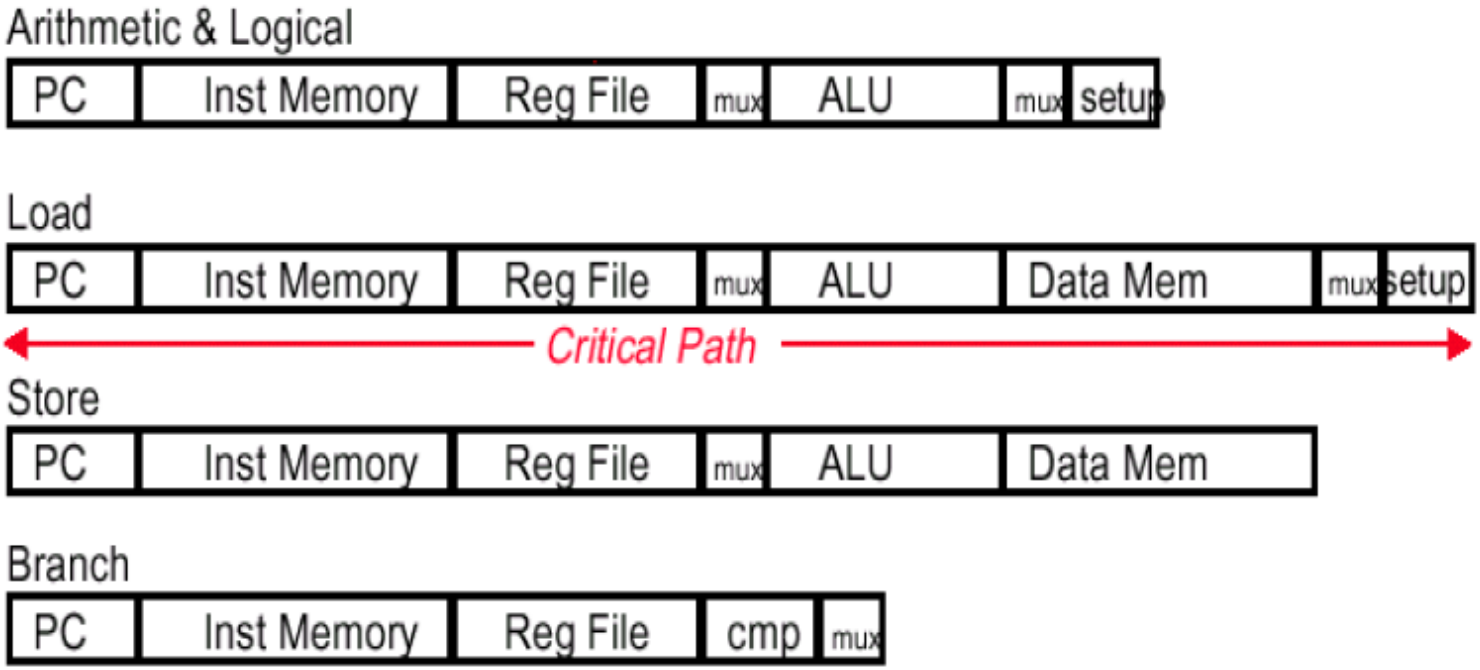
  - Typical execution:
    - Read contents of some state elements,
    - Send values through some combinational logic
    - Write results to one or more state elements





Single-cycle implementation

- Why a single-cycle implementation isn't used today?



  - Long cycle time for each instruction (load takes longest time)
  - All instructions take as much time as the slowest one

Performance of single-cycle implementation

- Example:
  - Assumption:
    - Memory units : 200 ps
    - ALU and adders : 100 ps
    - Register file (read / write) : 50 ps
    - Multiplexers, control unit, PC accesses, sign extension unit, and wires have no delay.
    - The **instruction mix**: 25% loads, 10% stores, 45% ALU instructions, 15% branches, 5% jumps.

Performance of single-cycle implementation

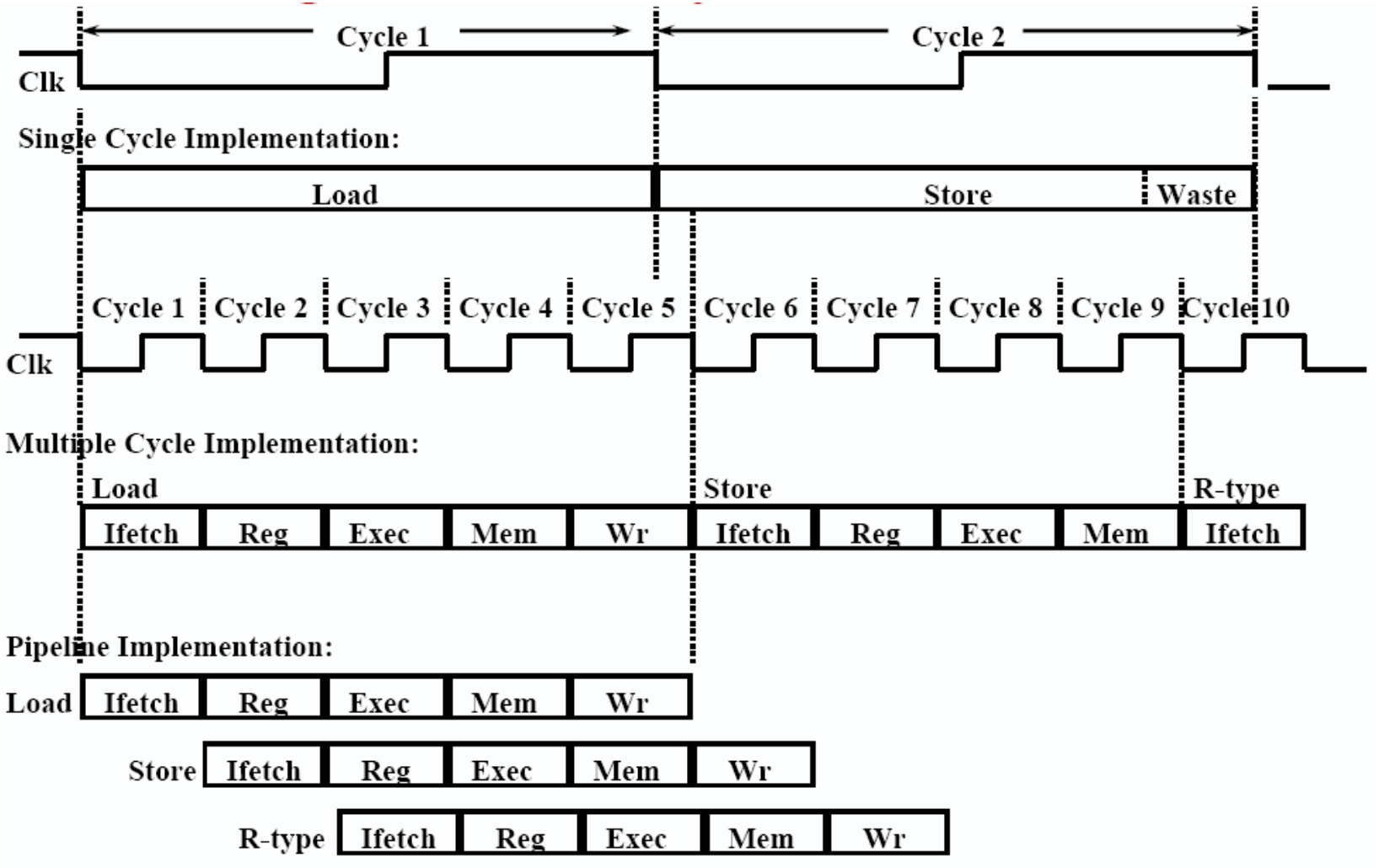
- Answer:
  - The critical path for the different instruction classes:

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

  - Compute the require length for each instruction class:

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

Single-, Multi-cycle, vs. Pipeline



Outline

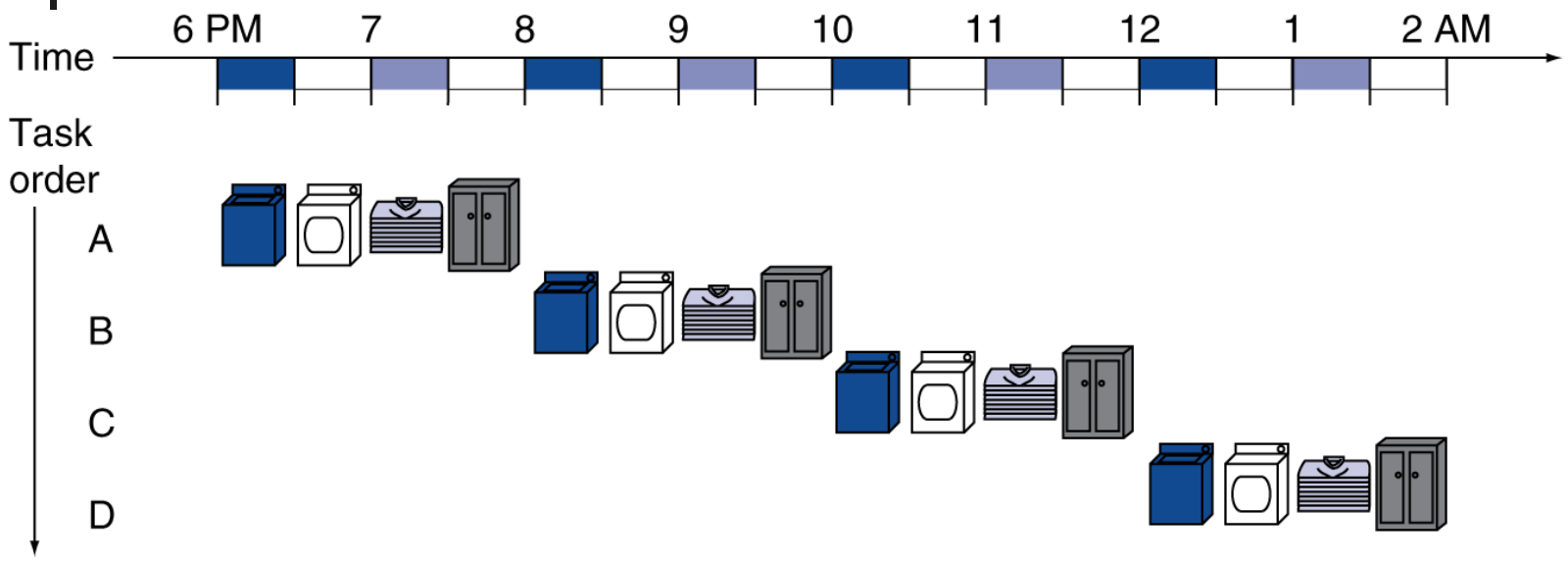
- 4.5 An Overview of Pipelining
- 4.6 A Pipelined Datapath & Control
- 4.7 Data Hazards and Forwarding v.s. Stalls
- 4.8 Control Hazards
- 4.9 Exceptions

Pipelining is Natural!

- Ann, Brian, Cathy, and Don each have dirty clothes to be *washed*, *dried*, *folded*, and *put away*

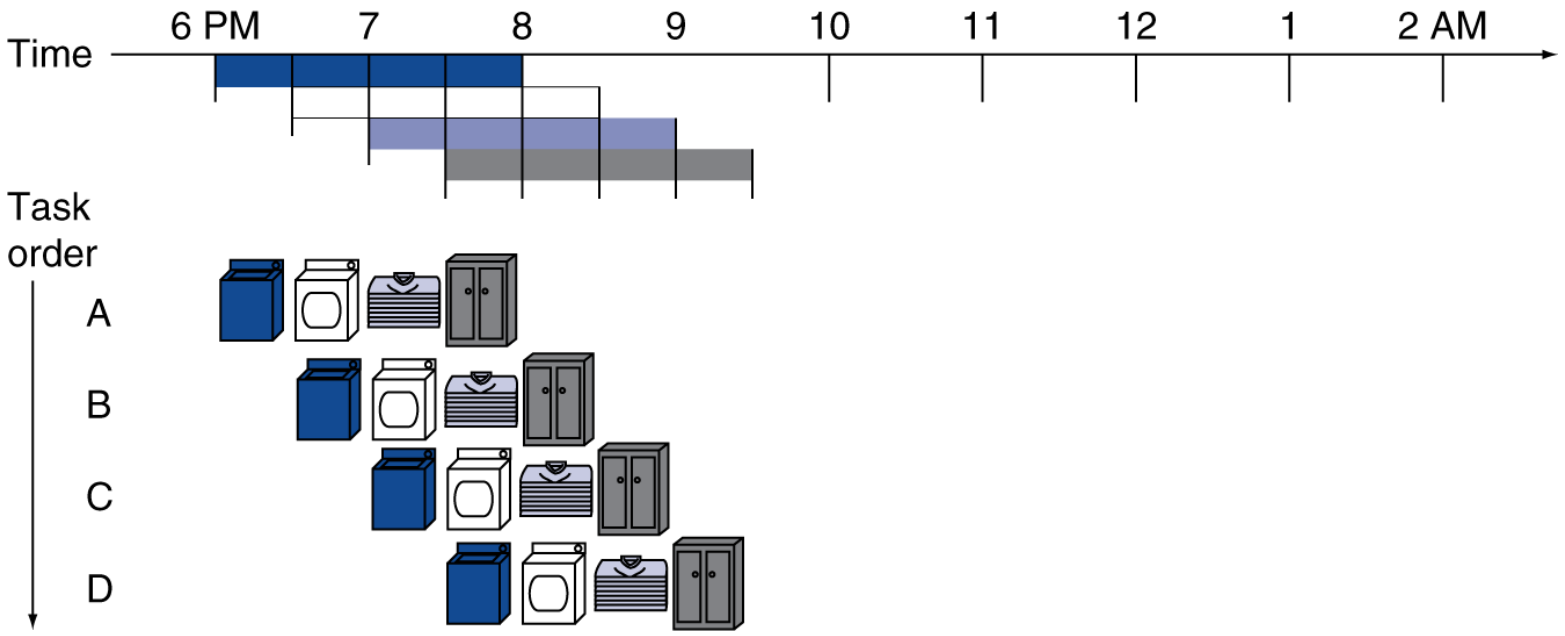
- The *washer, dryer, folder, storer* each take 30 minutes for their task

Sequential laundry



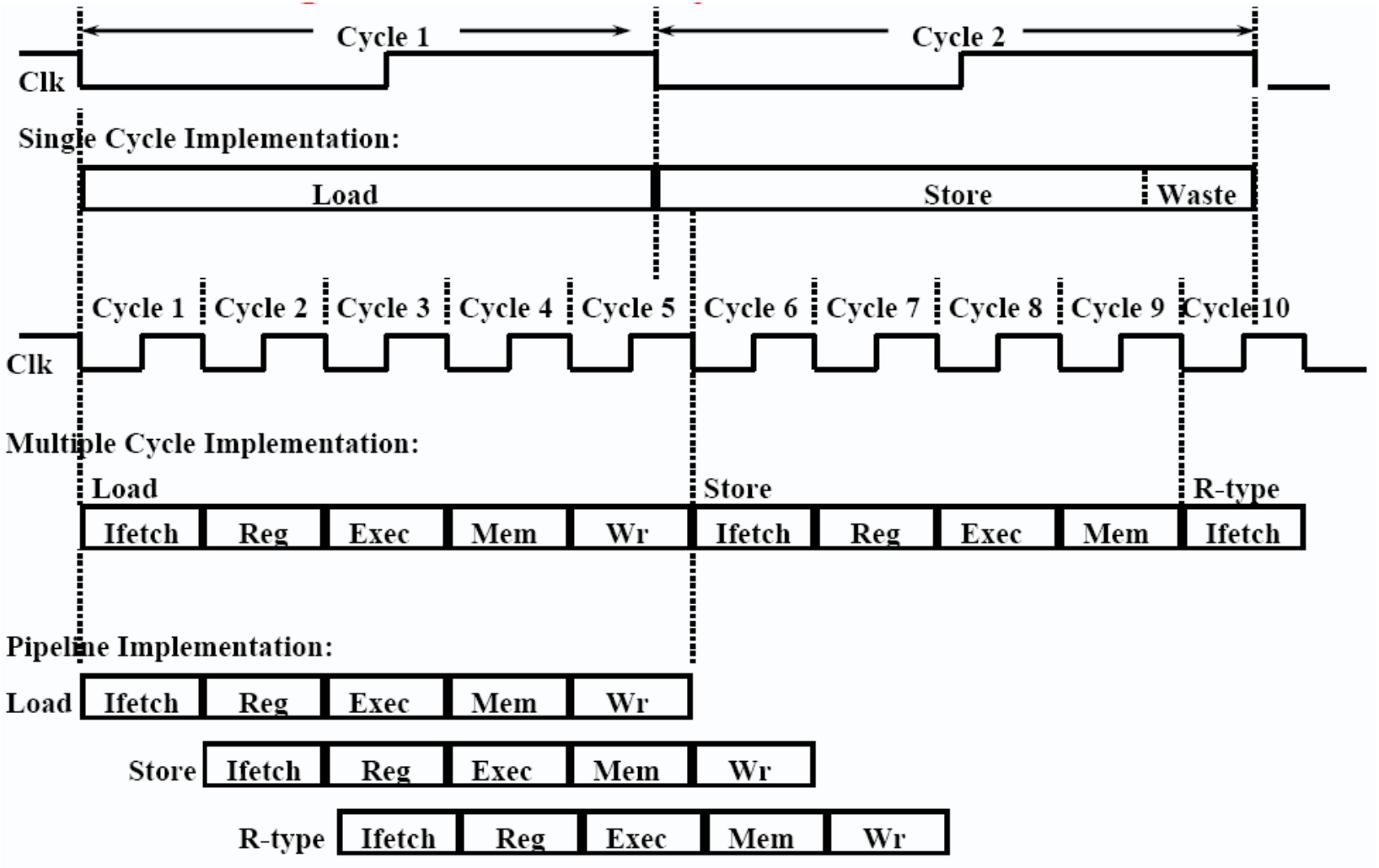
- If they learned pipelining, how long would it take?
- Sequential laundry takes 8 hours for 4 loads
- Throughput = 4/8 = 0.5 load/hour (latency = 2 hours)

Pipelined laundry

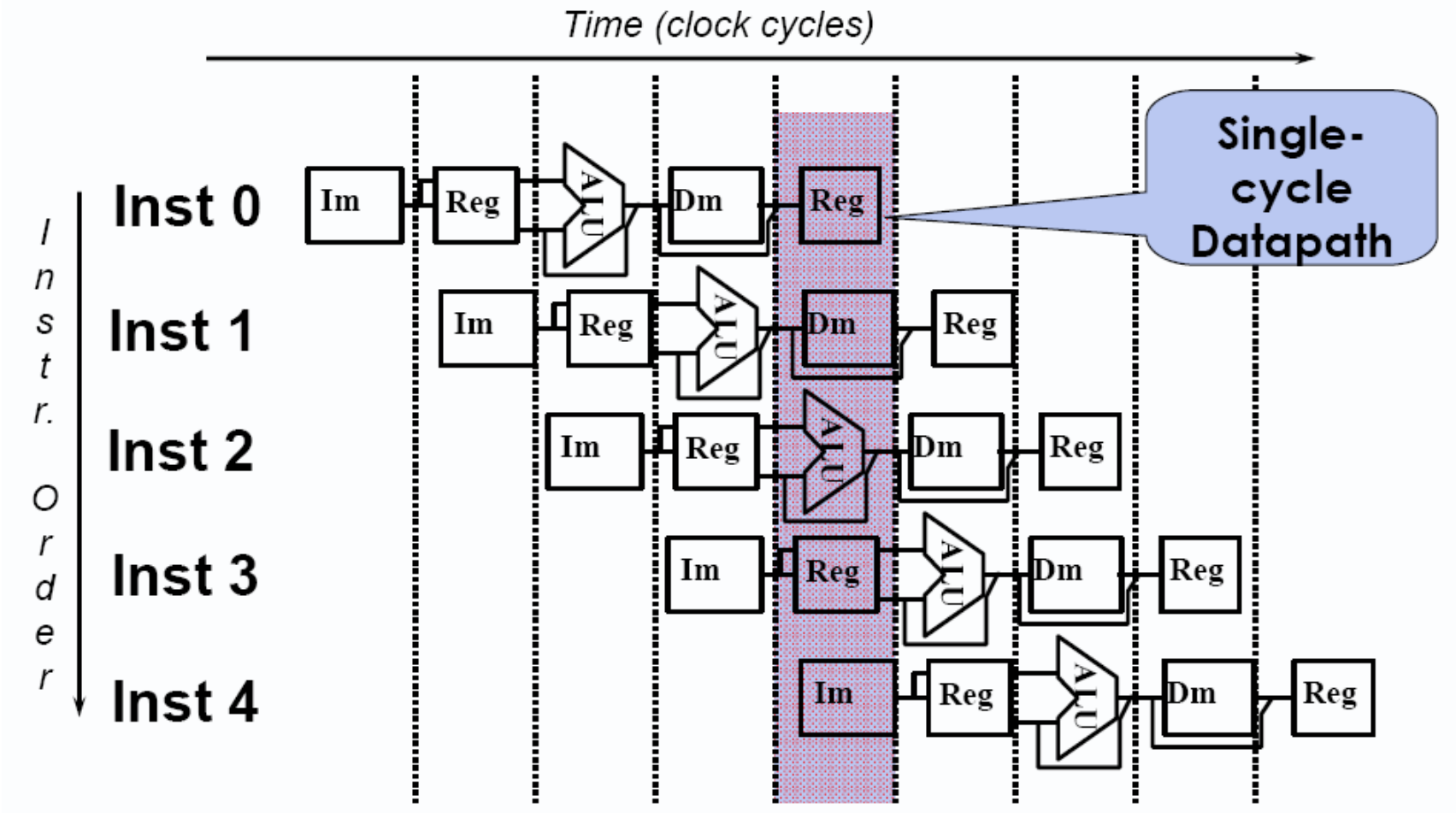


- Pipelined laundry takes 3.5 hours for 4 loads
- Throughput = 4/3.5 = 1.15 load/hour (latency = 2 hours)

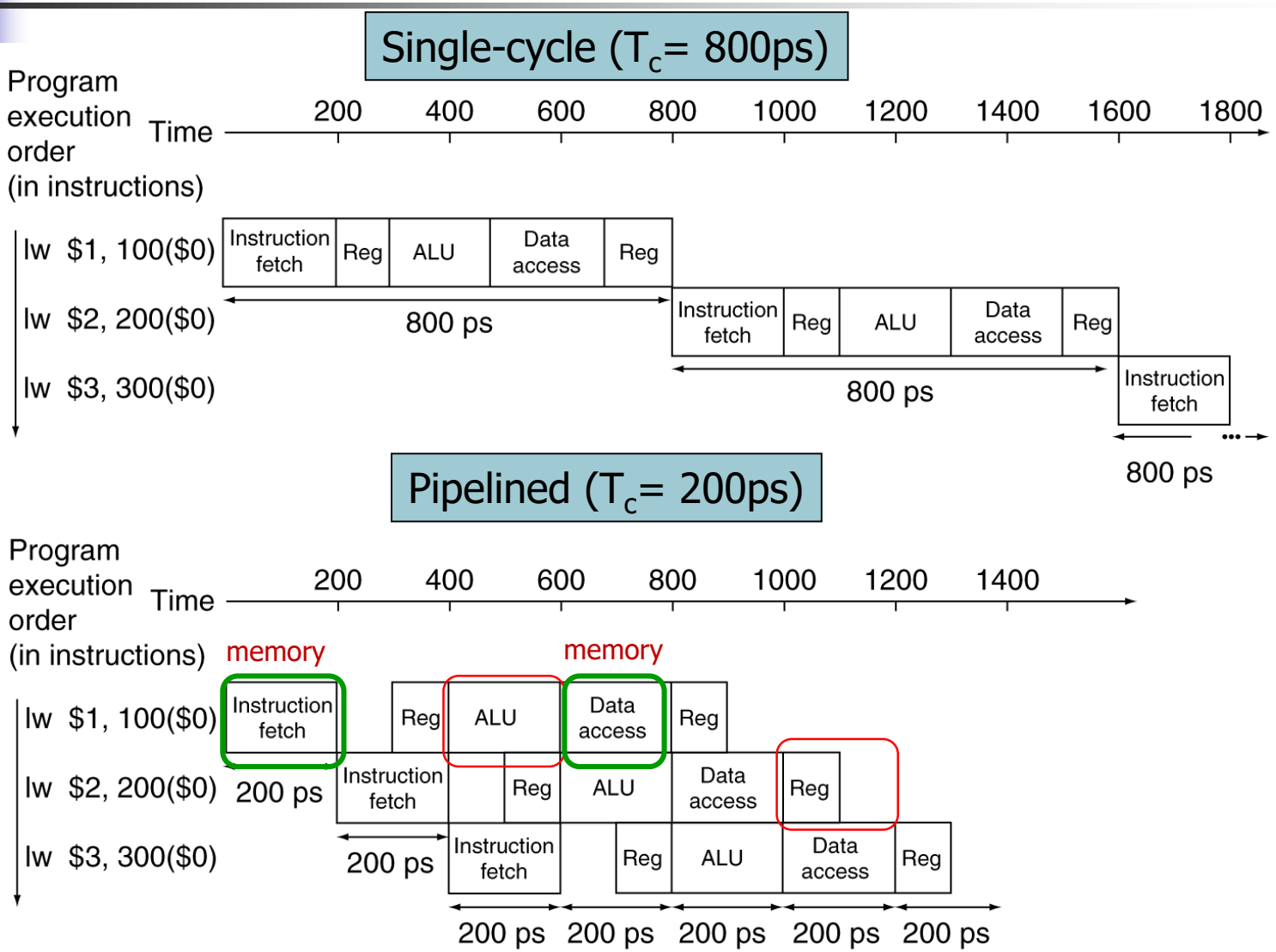
Single-, Multi-cycle, vs. Pipeline



Why Pipeline? Because the Resources Are There!



Pipelining MIPS Execution



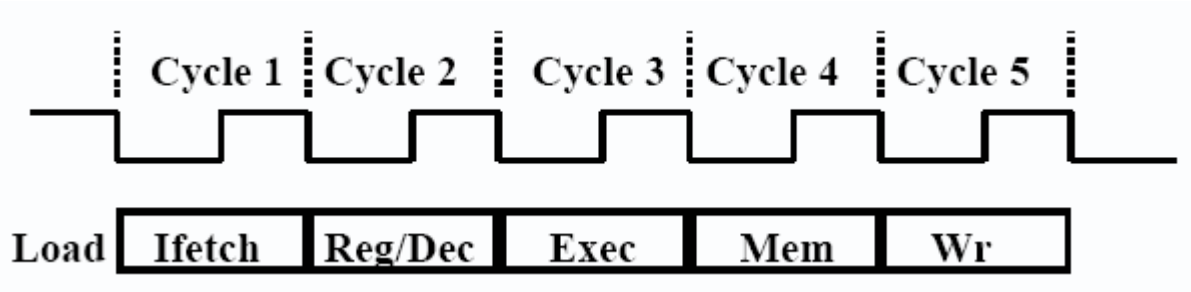
Performance Index of Pipelining

- Latency (pipeline)
  - The number of stages in a pipeline or the number of stages between two instructions during execution.
- Throughput (pipeline)
  - The number of instructions executed per unit time.

Outline

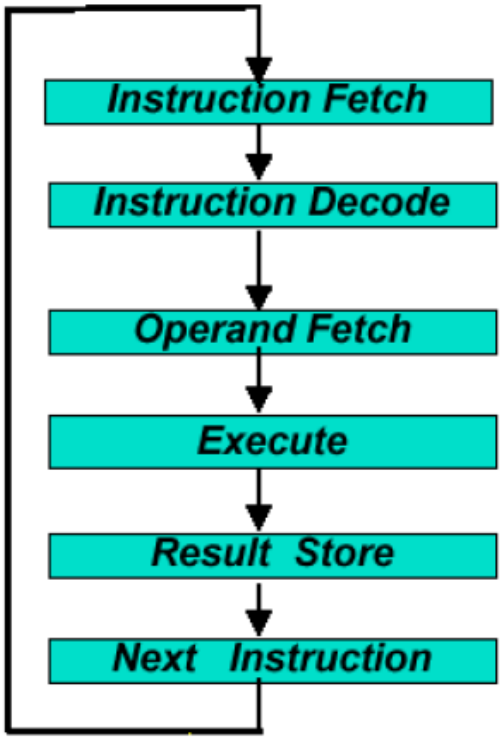
- 4.5 An Overview of Pipelining
- 4.6 A Pipelined Datapath & Control
- 4.7 Data Hazards and Forwarding v.s. Stalls
- 4.8 Control Hazards
- 4.9 Exceptions

Consider load word (lw)



- 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

Typical Instruction Execution



IF: access instruction memory

ID: identify opcode, generate control signals

OF: read register(s), route immediate field

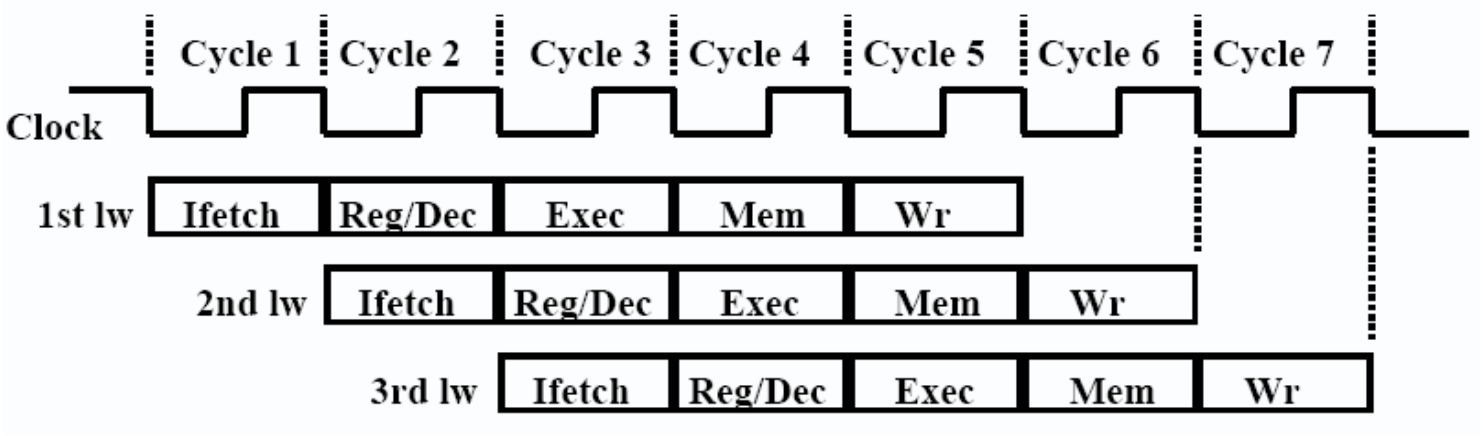
EX: perform operation; generate address, access data memory

RS: write register, write memory

NI: increment PC or update PC with new target

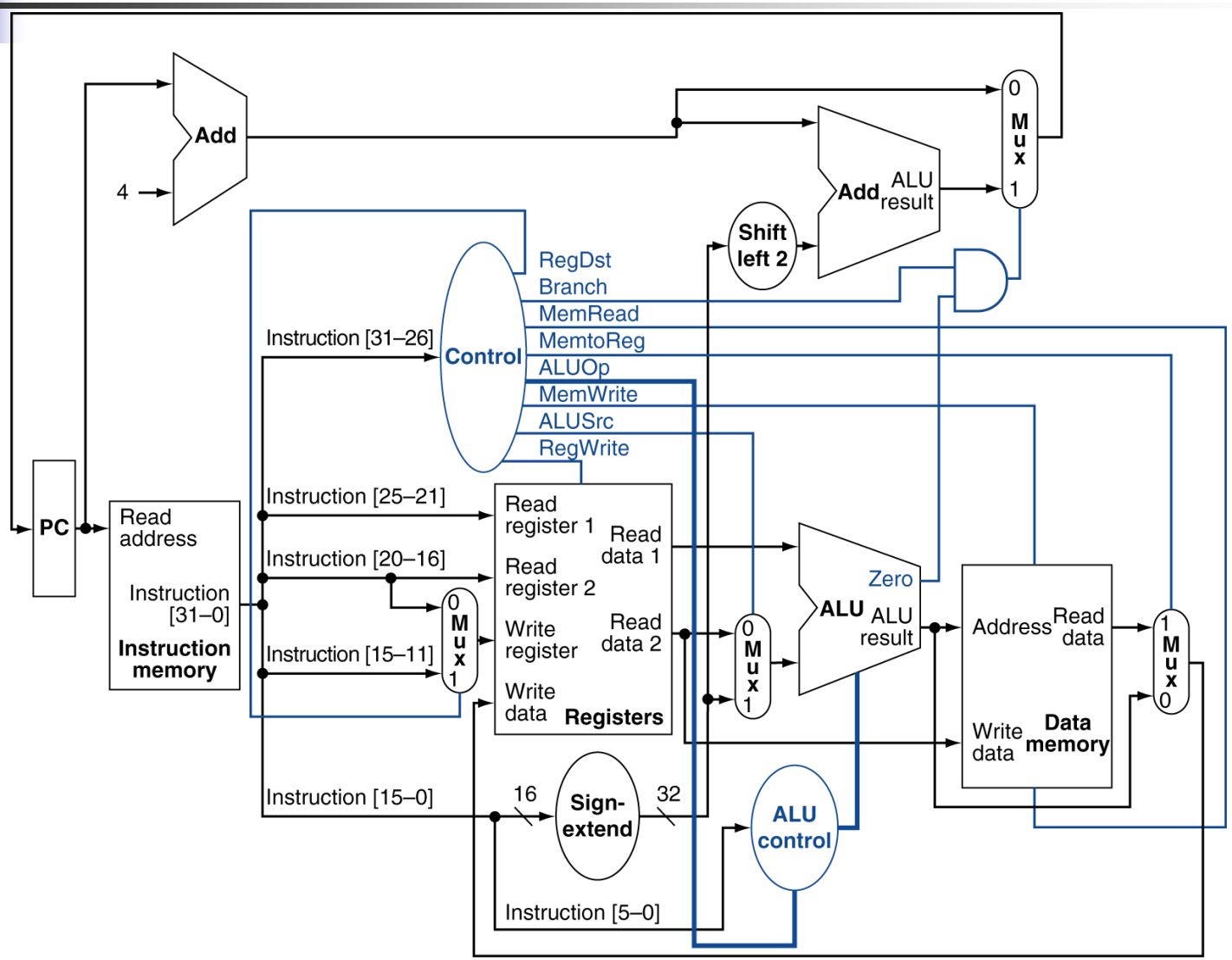
Note that each step does not necessarily correspond to a clock cycle. These only describe the basic flow of instruction execution. The details vary with instruction type.

Pipelining load word

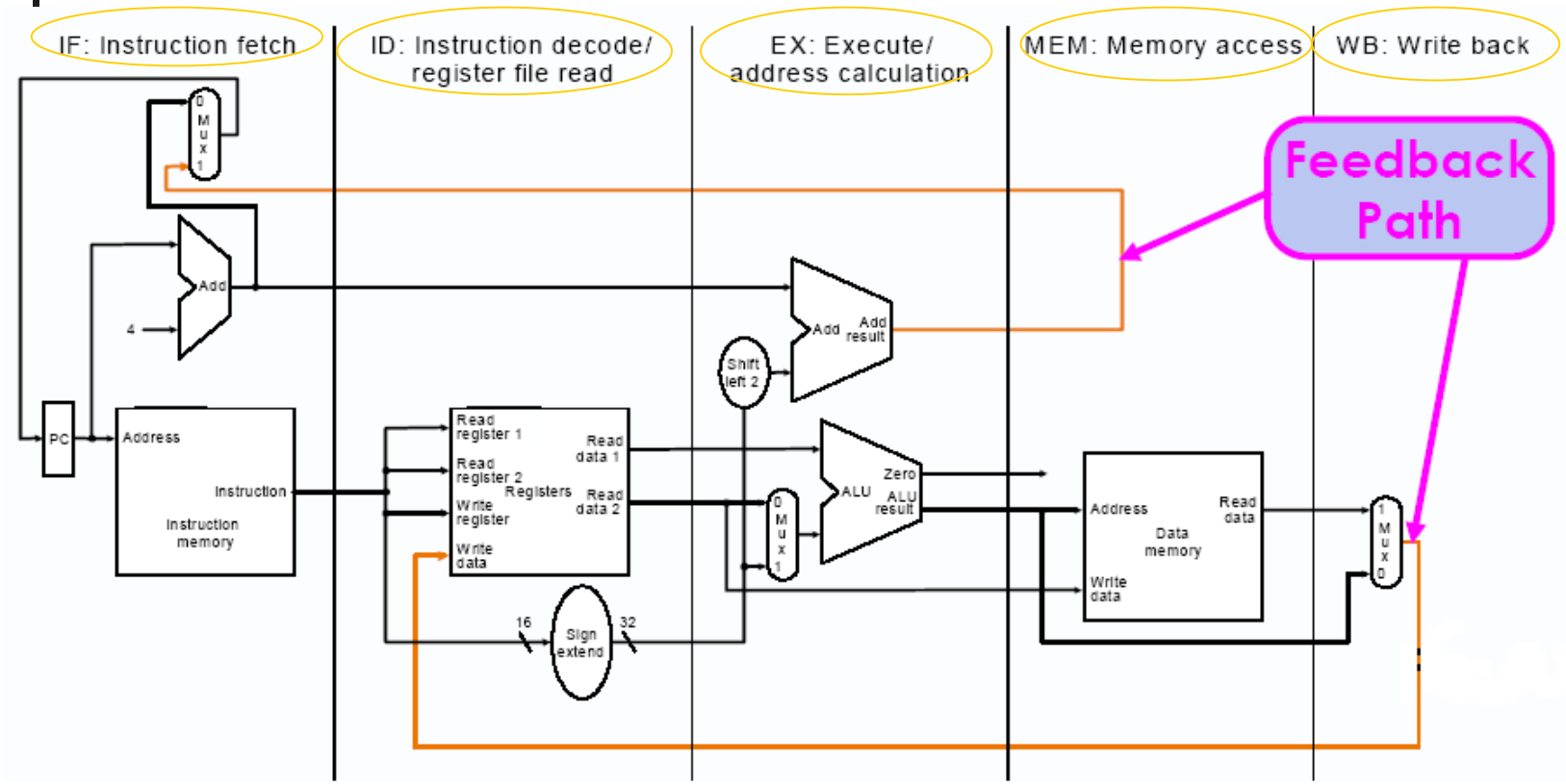


- 5 functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (busA and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the MEM stage
  - Register File's Write port (busW) for the WB stage

Start with Single-cycle MIPS Design

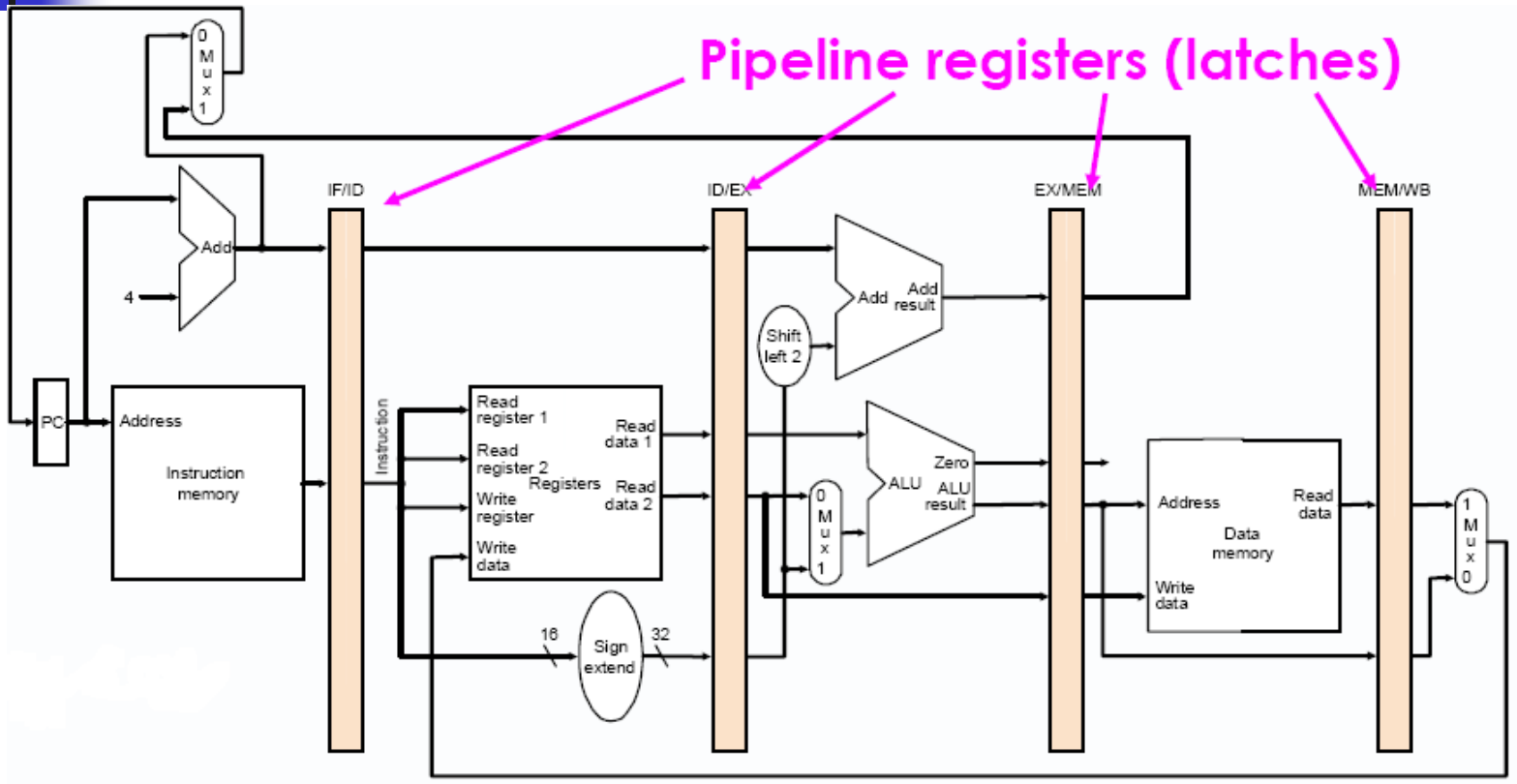


Split Single-cycle Datapath



What to add to split the datapath into stages?

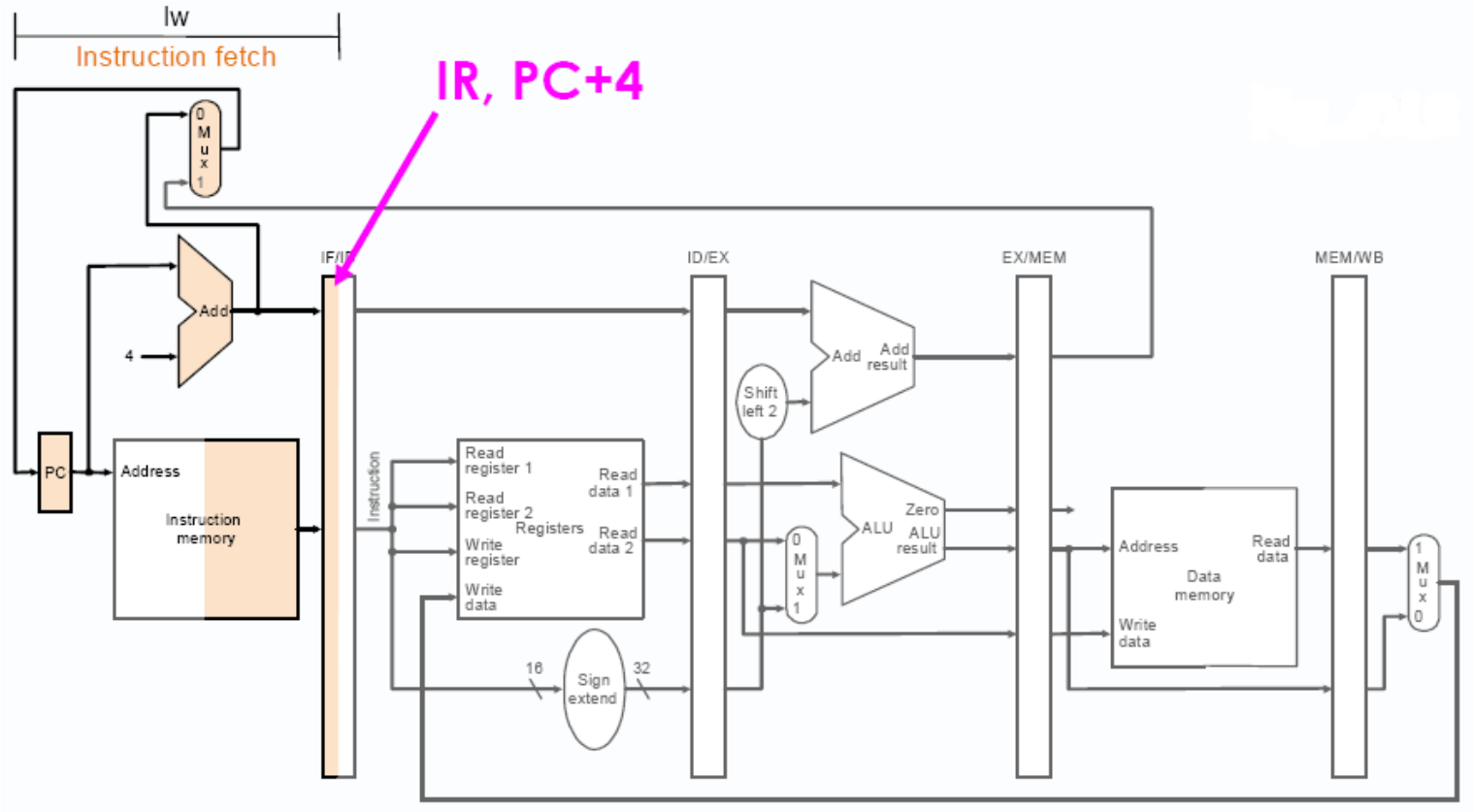
Add Pipeline Registers (stages)



- Use registers between stages to carry data and control

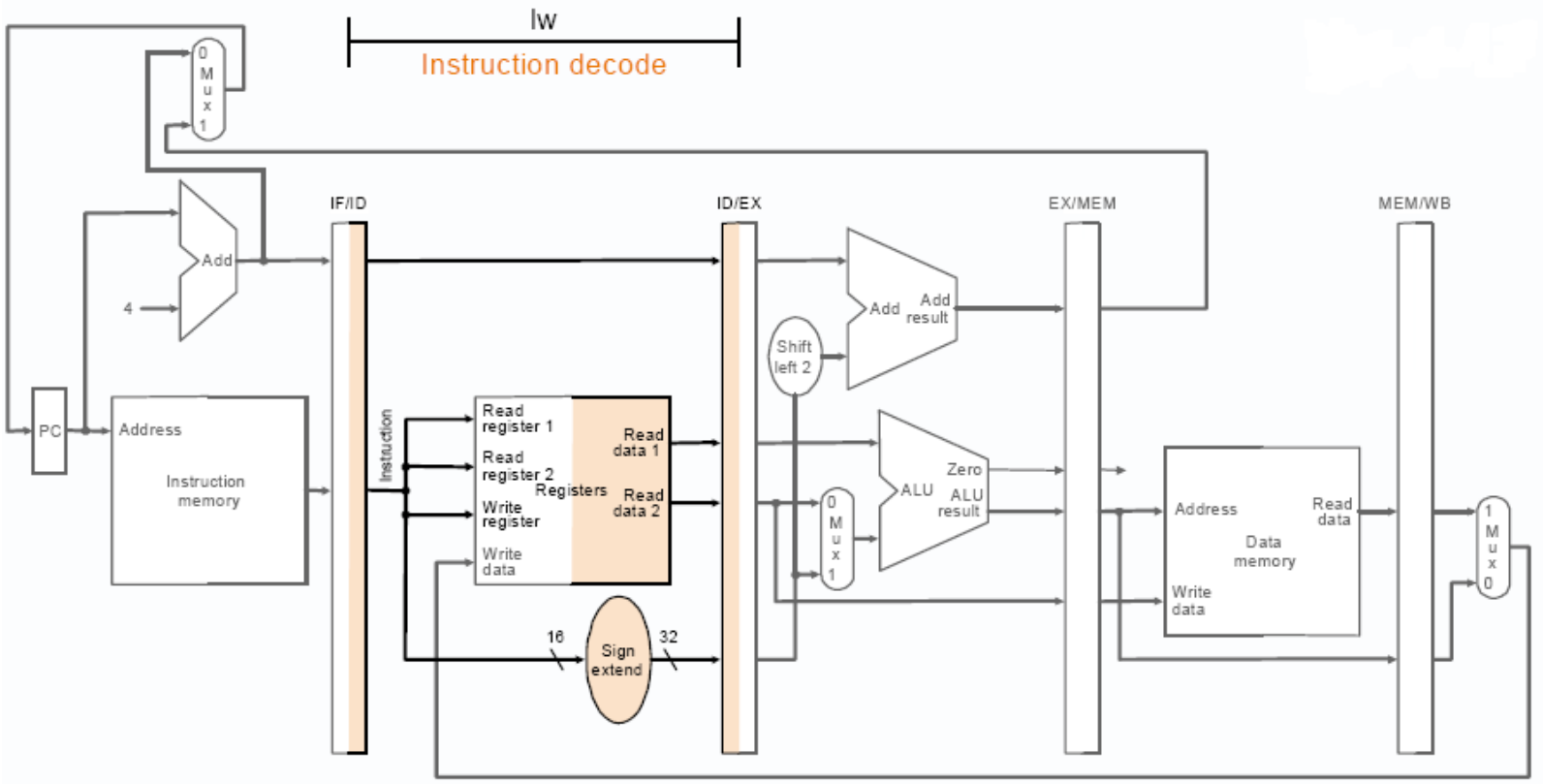
IF Stage of load word

- IF/ID= mem[PC] ; PC = PC + 4



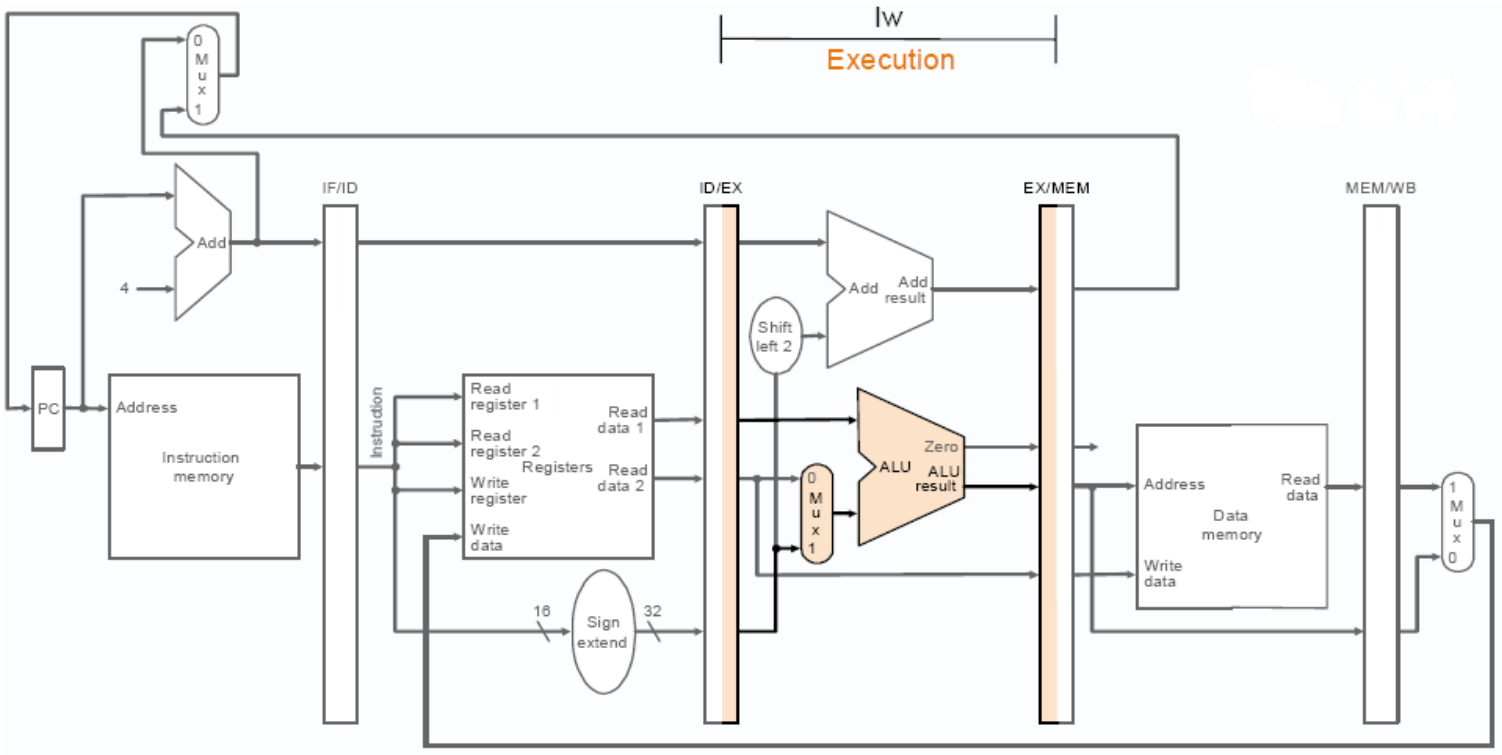
ID Stage of load word

- ID/EX(A)= Reg[IR[25-21]]; ID/EX(B)= Reg[IR[20-16]];
- ID/EX = Sign-extension of IF/ID[15:0]



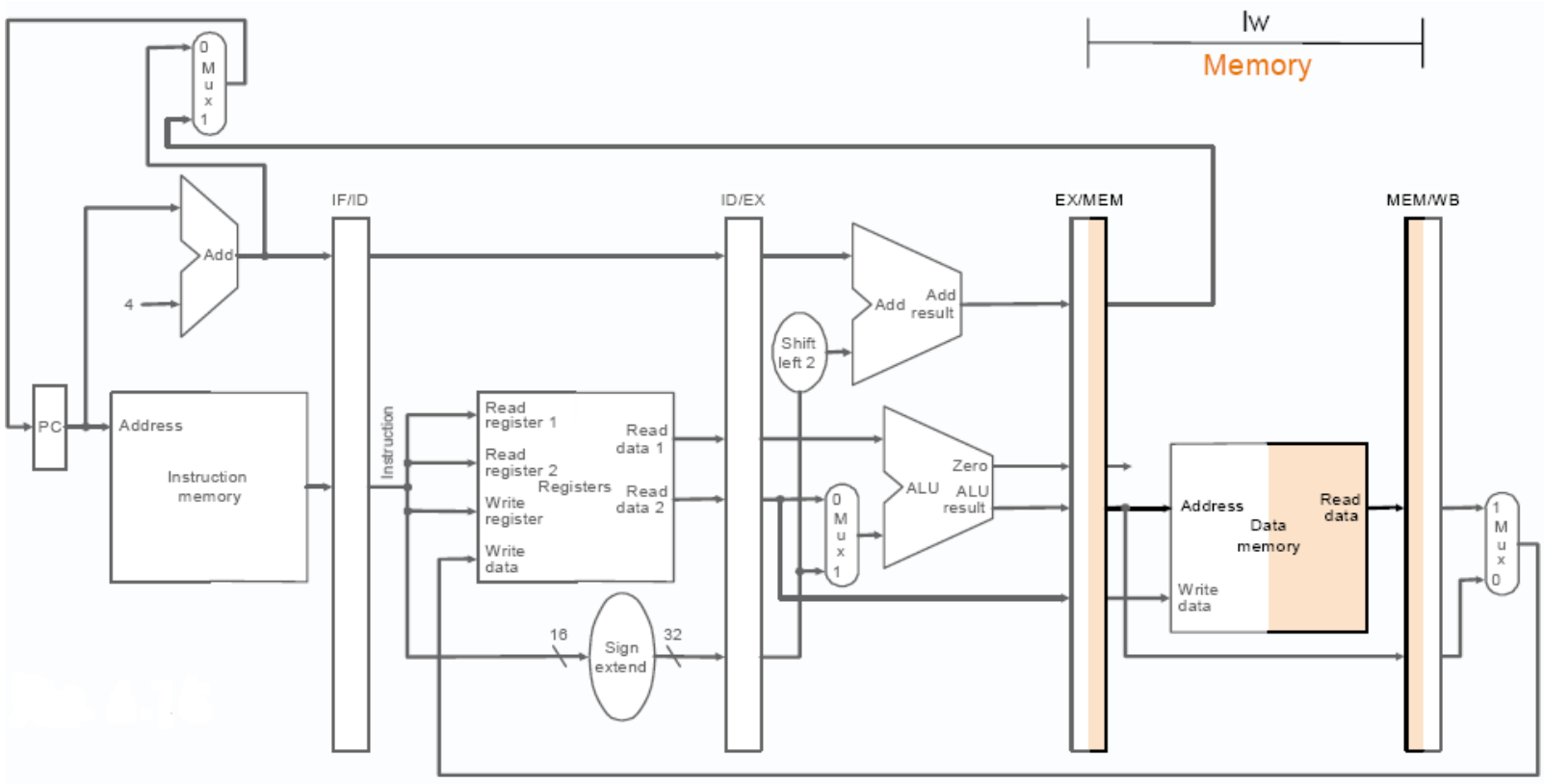
EX Stage of load word

- EX/MEM = ID/EX(A) + ID/EX[15:0] % address computation



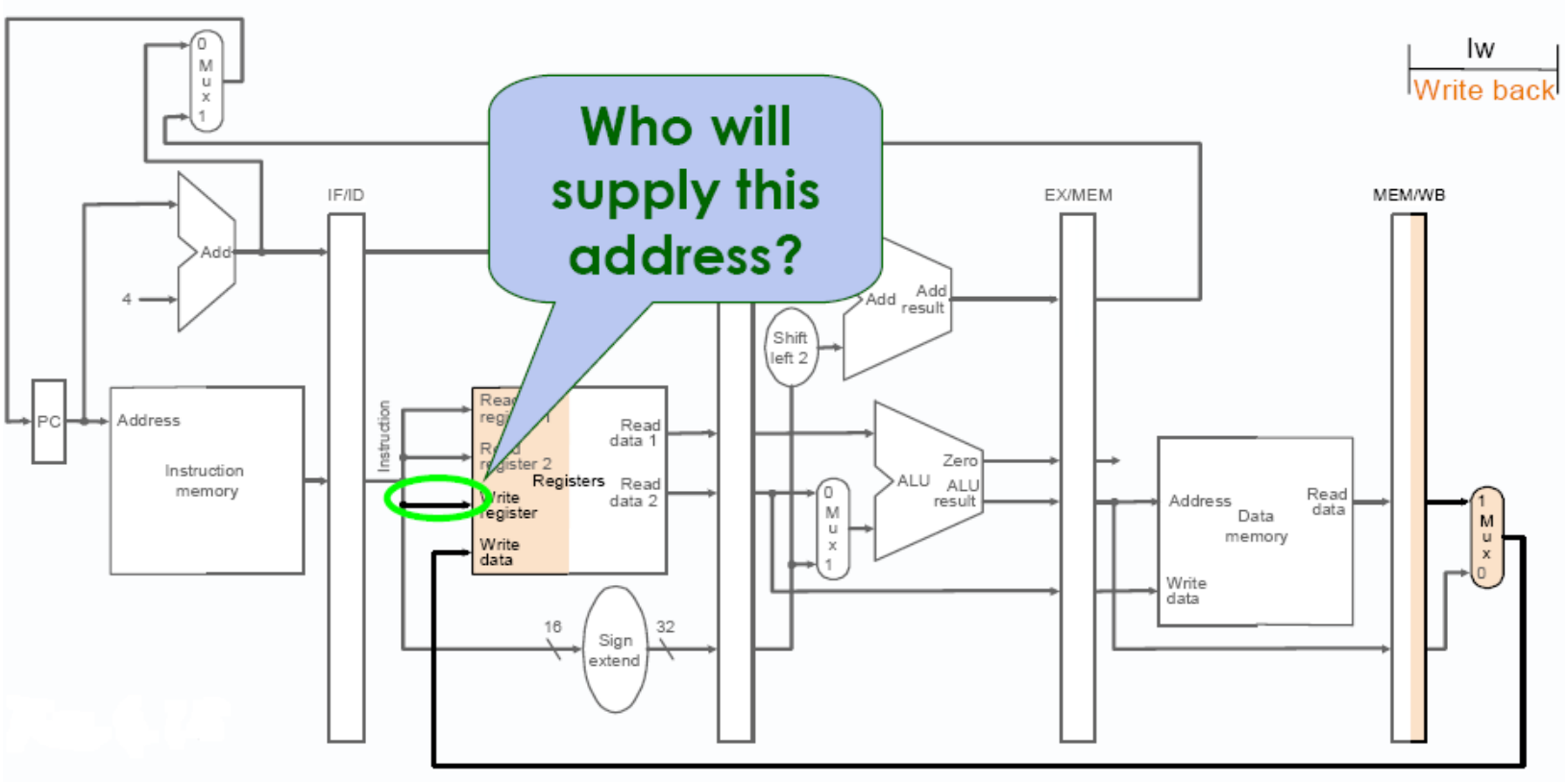
MEM Stage of load word

- MEM/WB = mem[ALUout] = mem[EX/MEM]

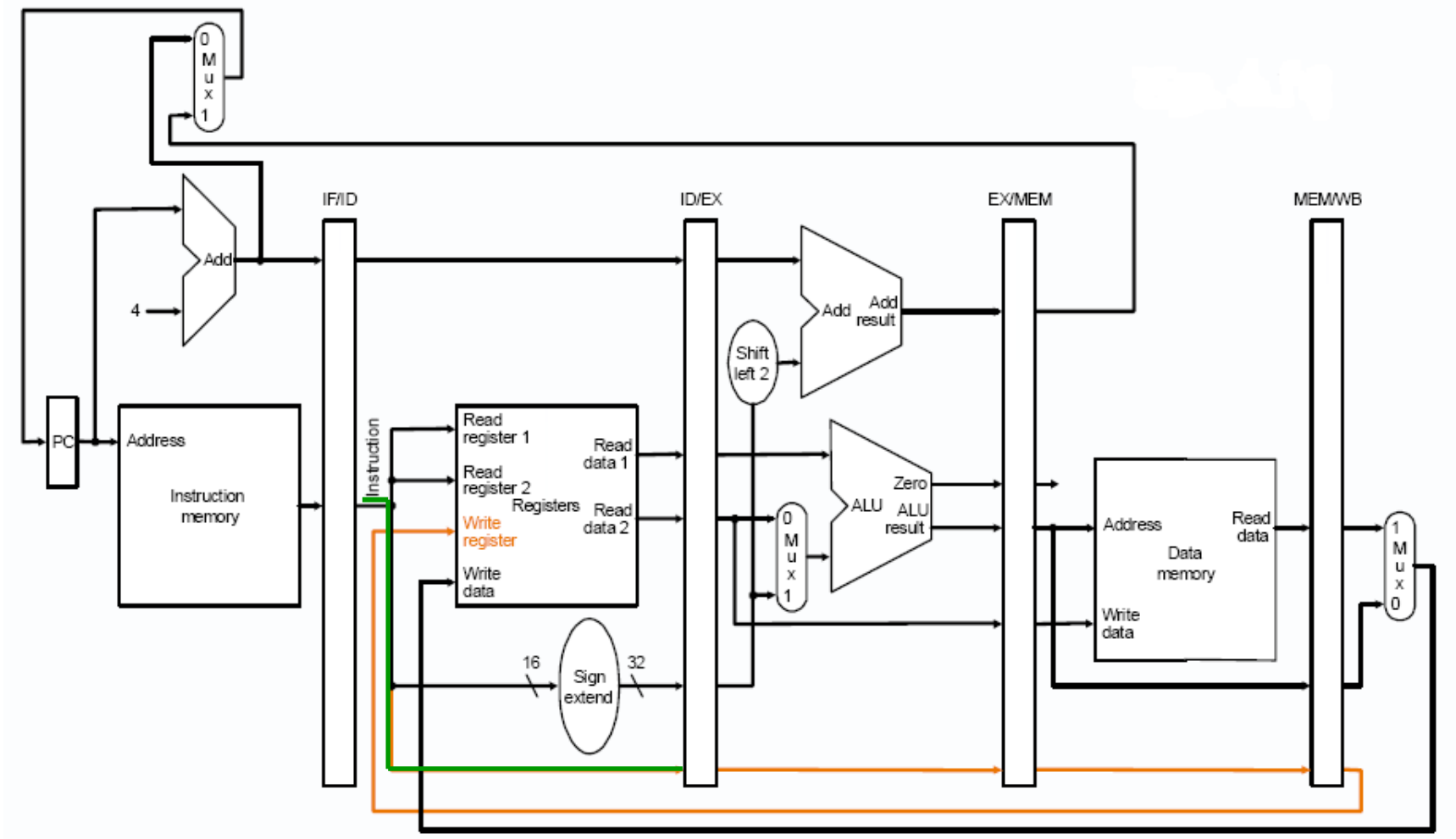


WB Stage of load

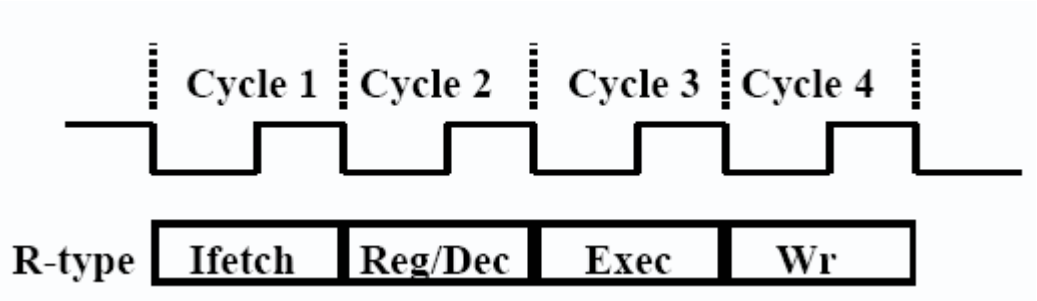
- Reg[IR[20-16]??] = MEM/WB



Solution for WB Stage

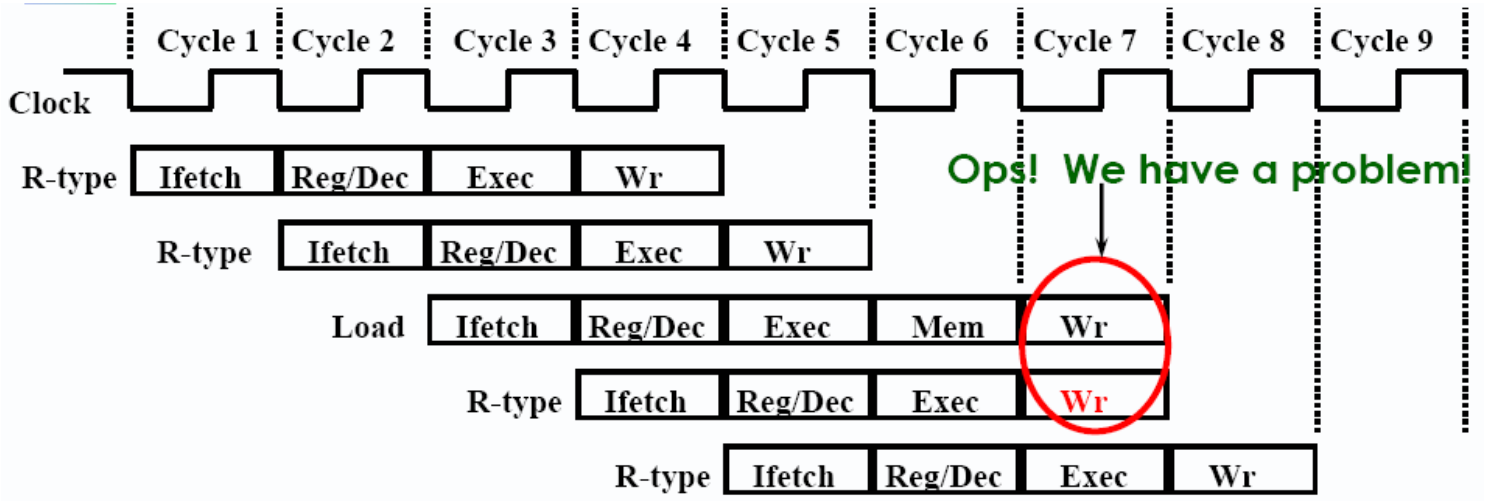


The Four Stages of R-type



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

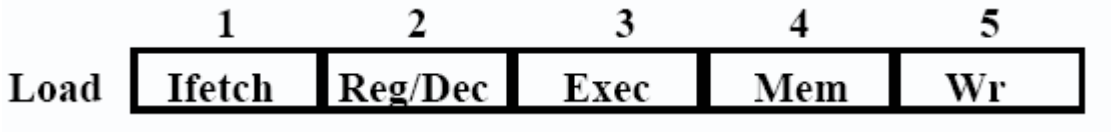
Pipelining R-type and load



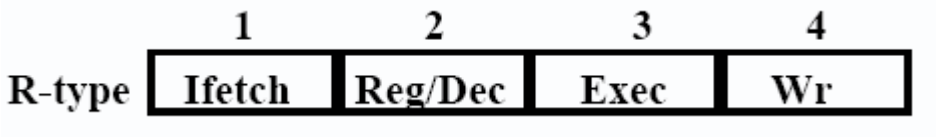
- We have a *structural hazard!*
  - Two instructions try to write to the same register file at the same time!
  - Only **one write port** is available

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

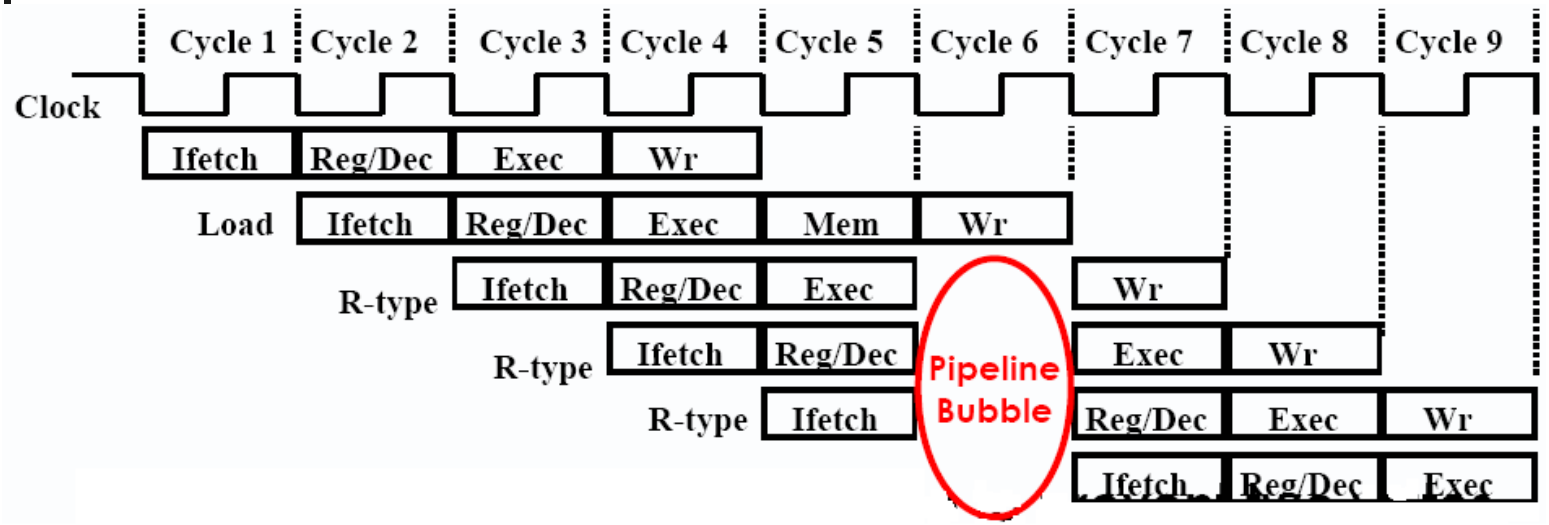


  - R-type uses Register File's write port during its 4th stage



Several ways to solve: 1). adding pipeline bubble, 2). making instructions same length, 3). forwarding (later)

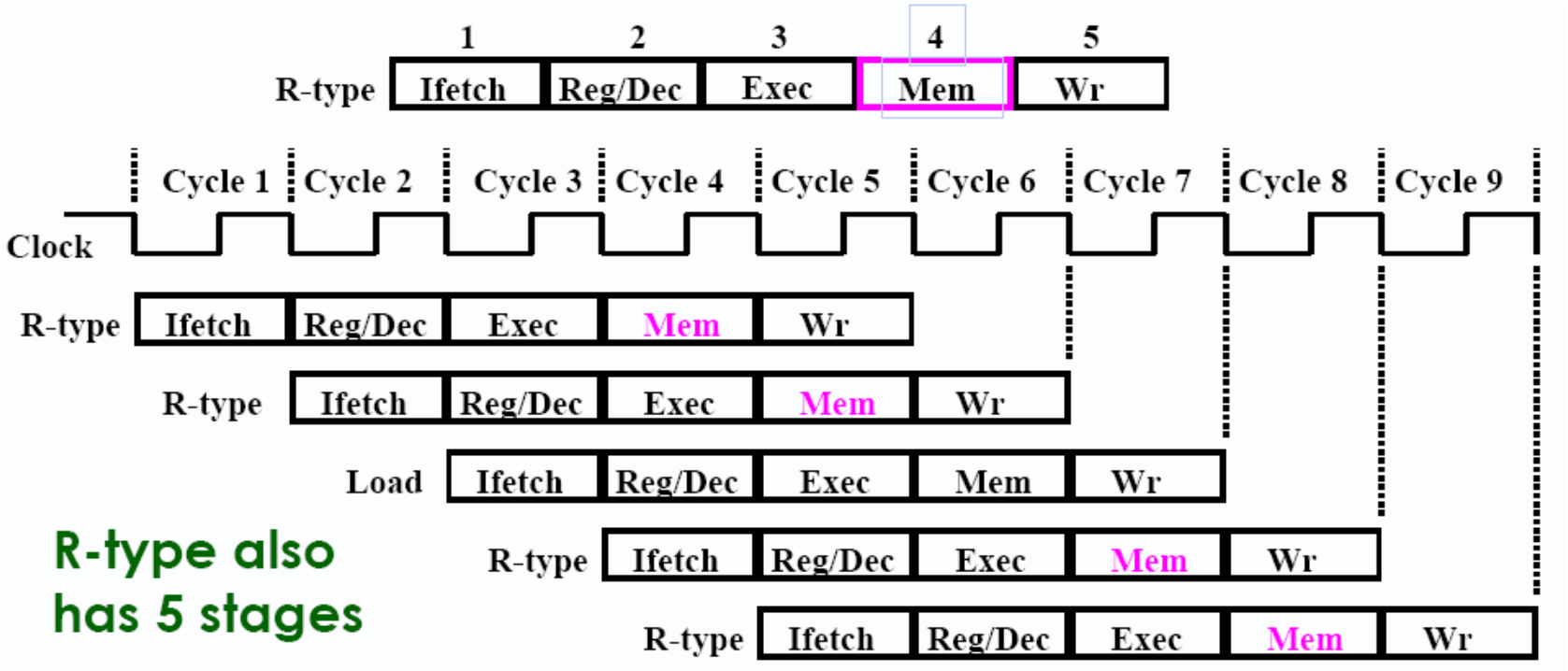
Solution 1: Insert Bubble (in hardware)



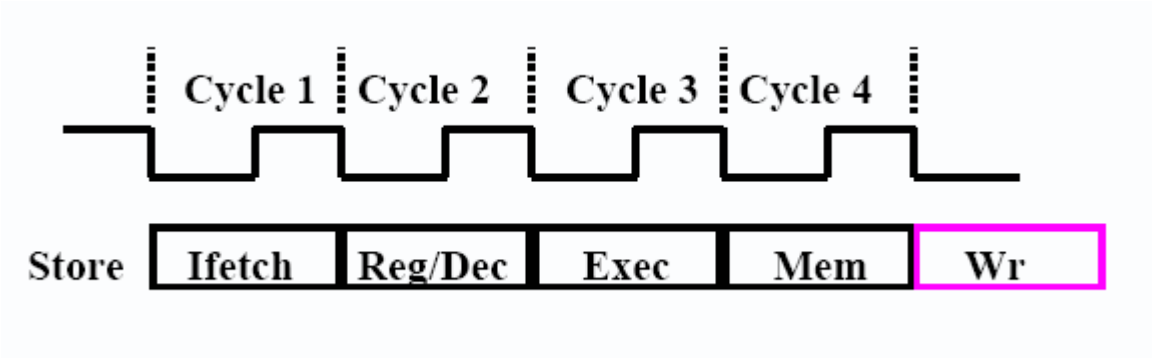
- Insert a bubble into the pipeline to prevent two writes at the same cycle
  - The control logic can be complex
  - Lose instruction fetch → No instruction is started in Cycle 6!

Solution 2: Delay R-type's Write with NOP

- Delay R-type's register write by one cycle:
  - R-type also use Reg File's write port at Stage 5 (like ***lw***)
  - MEM is a NOP (No Operation) stage: nothing is being done.



The Four Stages of store

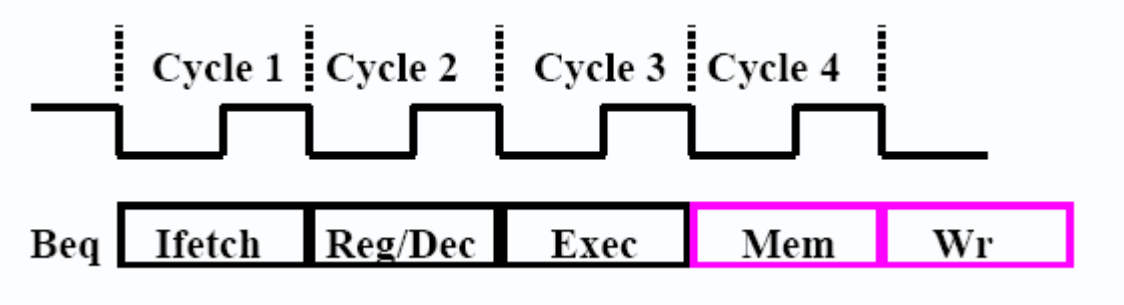


- **IF:** fetch the instruction from the Instruction Memory
- **ID:** registers fetch and instruction decode
- **EX:** calculate the memory address
- **MEM:** write the data into the Data Memory

Add an extra stage:

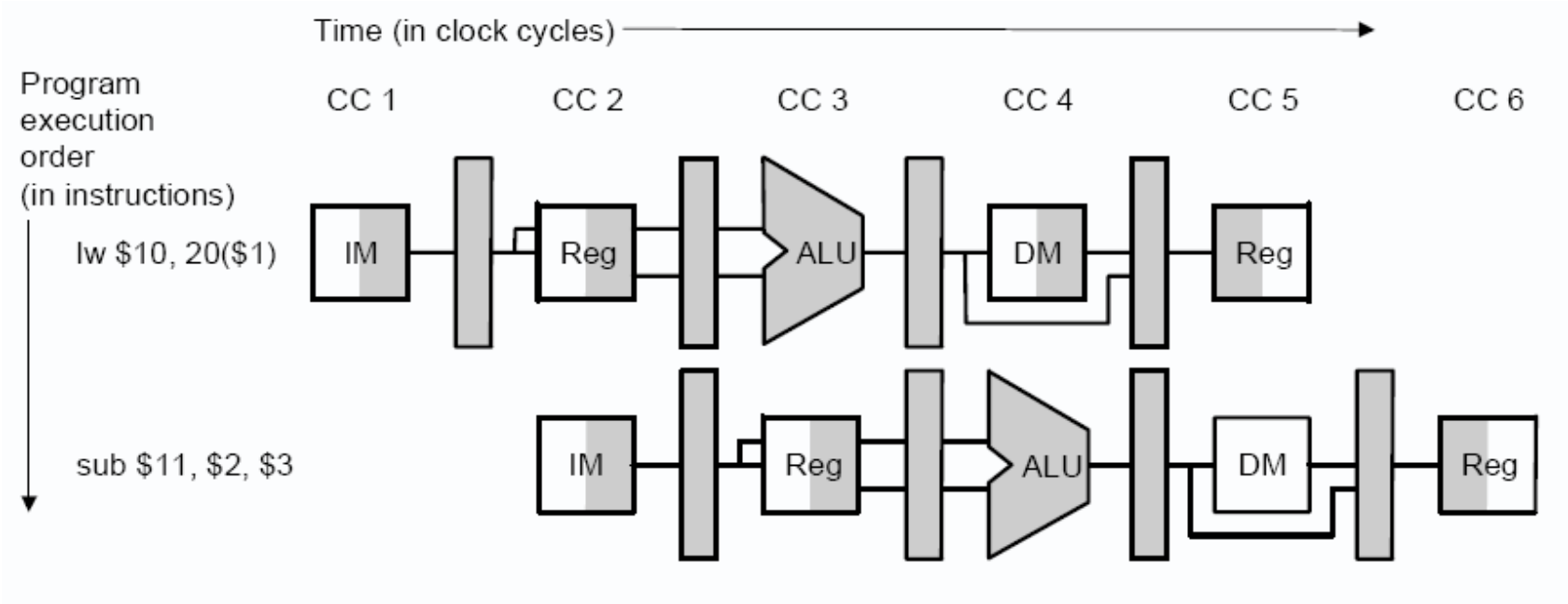
- **WB: NOP**

The Four Stages of beq



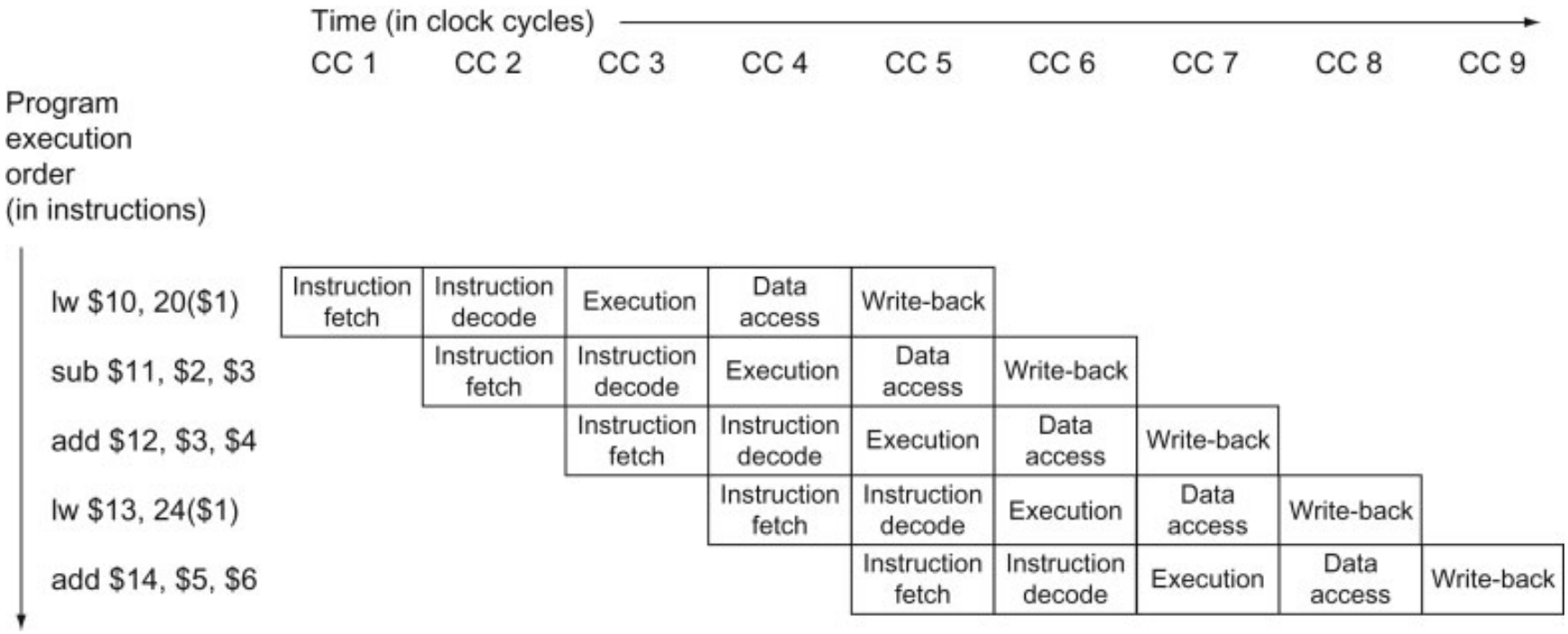
- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX:
  - compares the two register operand
  - select correct branch target address
  - latch into PC
- Add two extra stages:
- MEM: NOP
- WB: NOP

Multiple-clock-cycle Pipeline Diagram

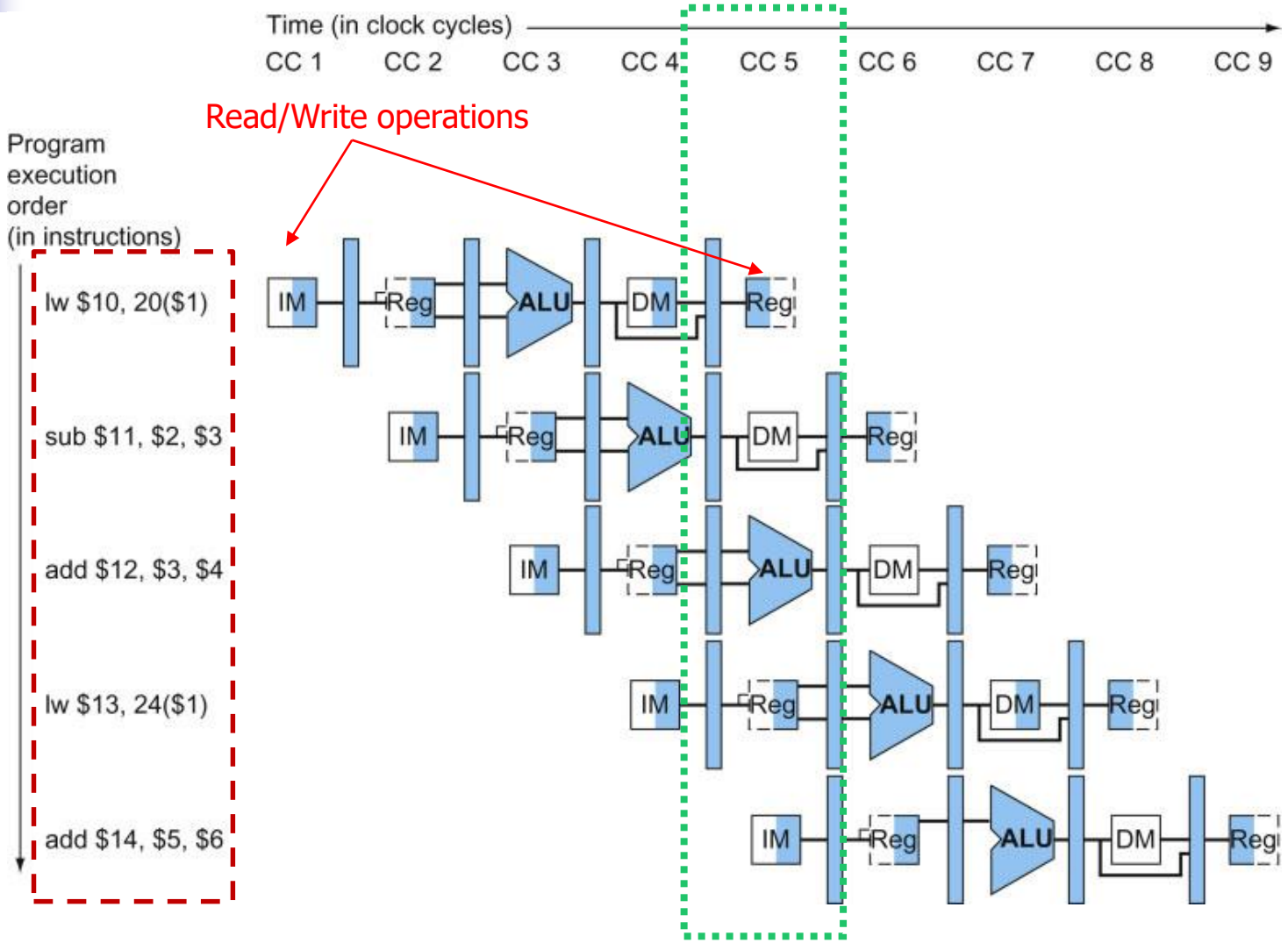


- Can help with answering questions like:
  - How many cycles to execute this code?
  - What is the ALU doing during cycle 4?
  - Help understand datapaths & Read/Write operations

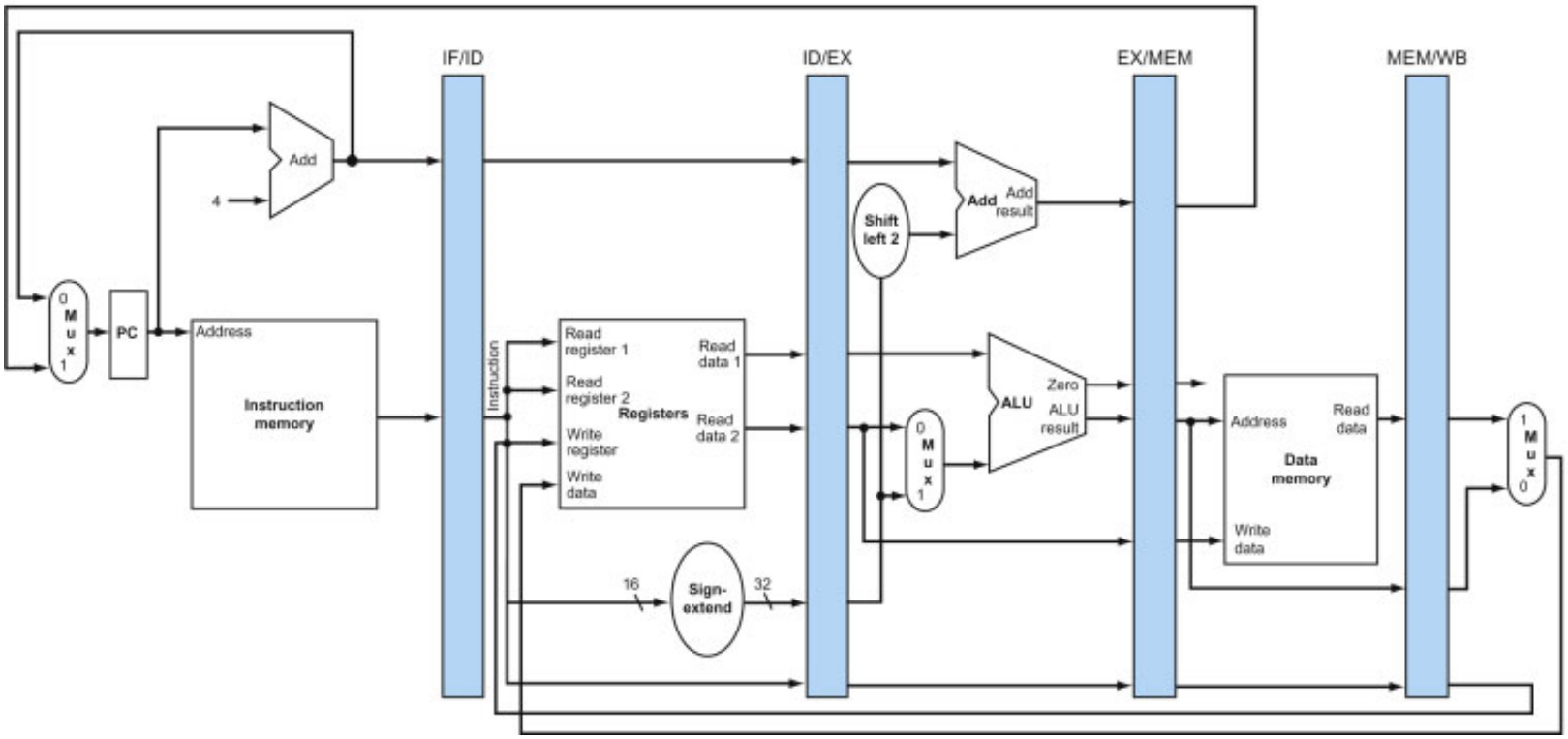
Traditional multi-clock-cycle Pipeline diagram of Fig. 4.44



Five instructions (Fig. 4.43) – Snapshot at CC5

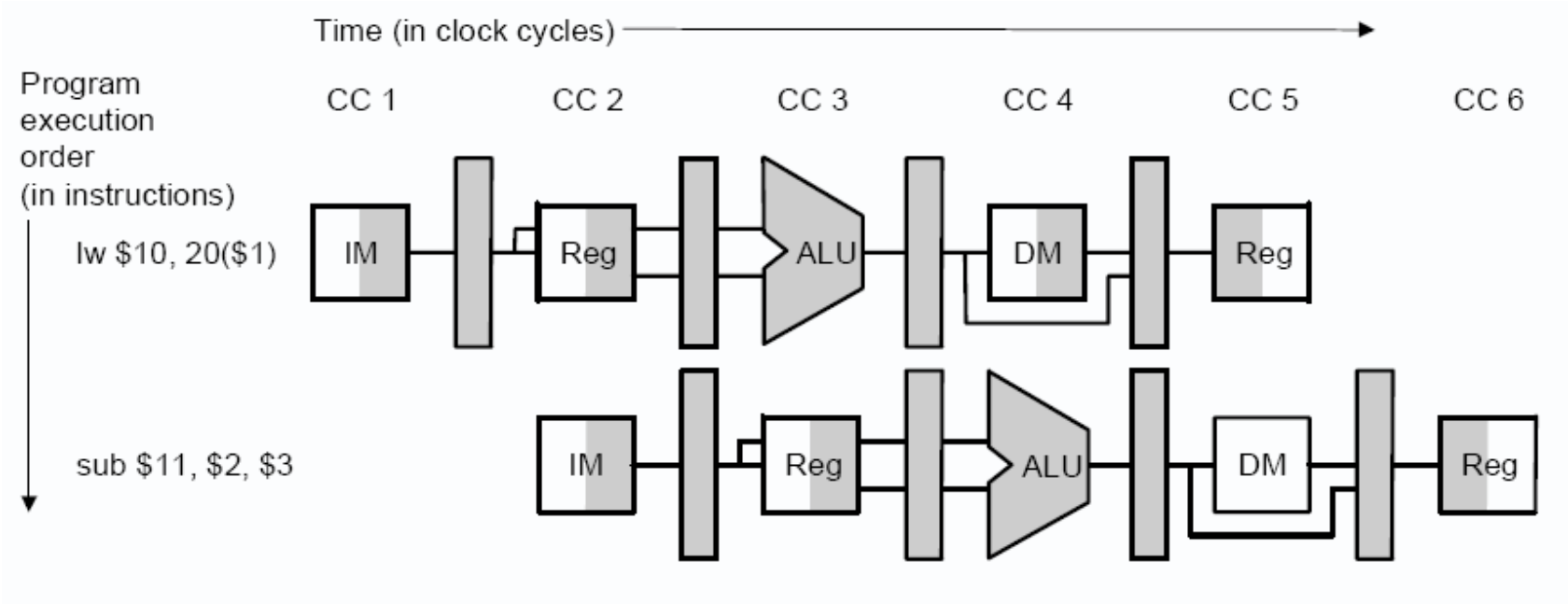


Single-clock-cycle diagram of Clock 5 of Fig. 4.45

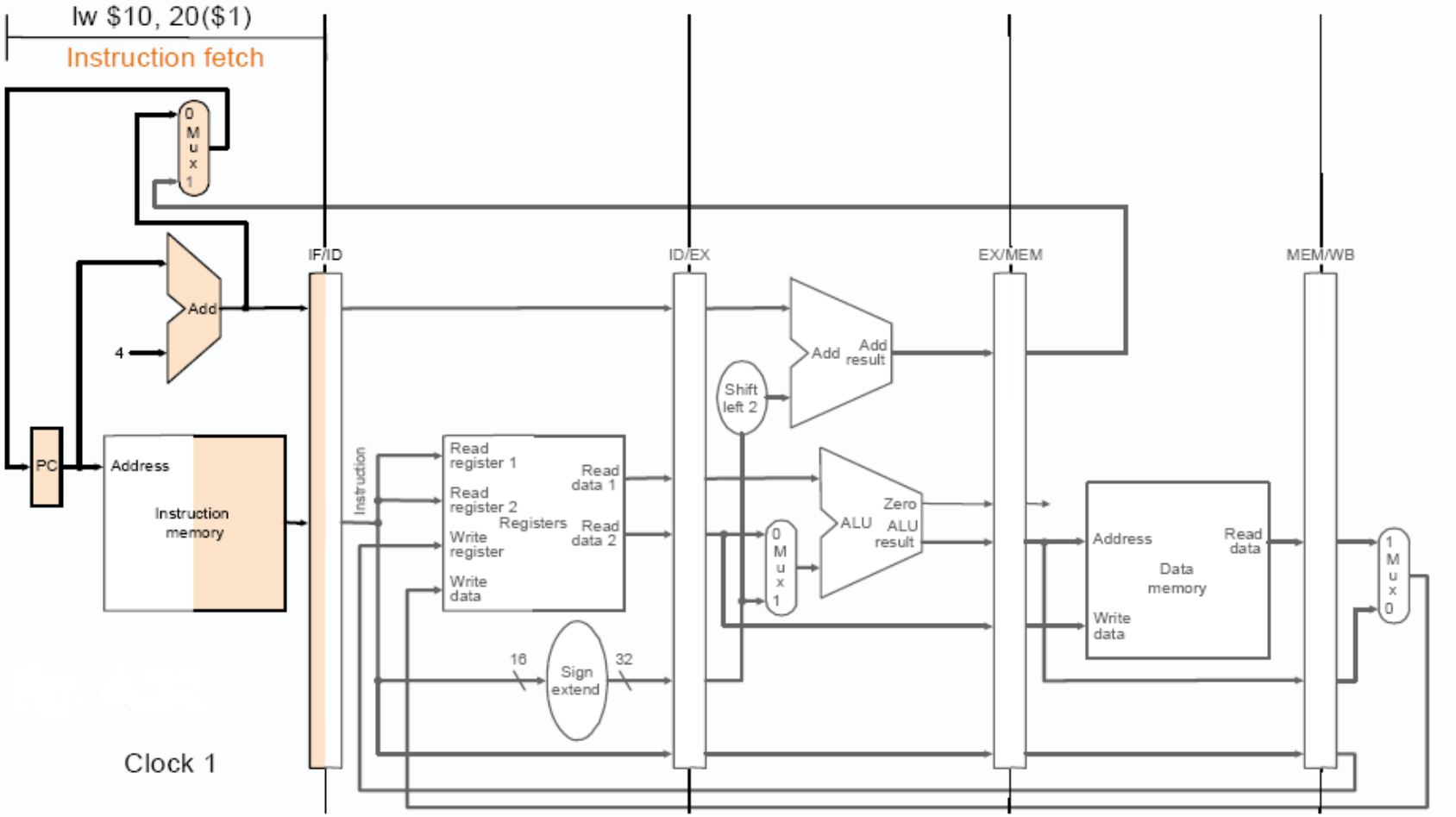


A vertical slice through a multiple-clock-cycle diagram!

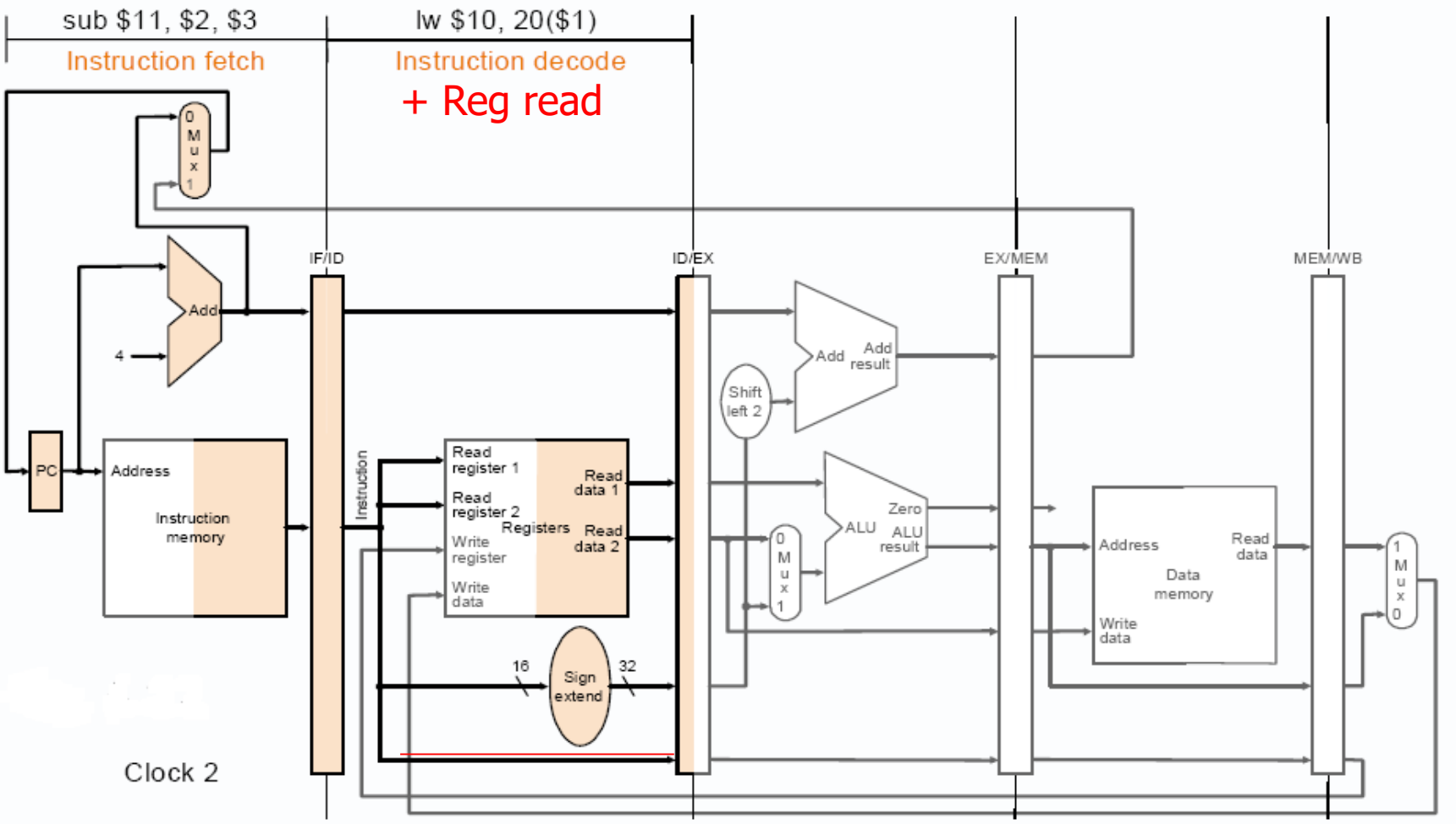
Example 1: *lw* ➔ *sub*



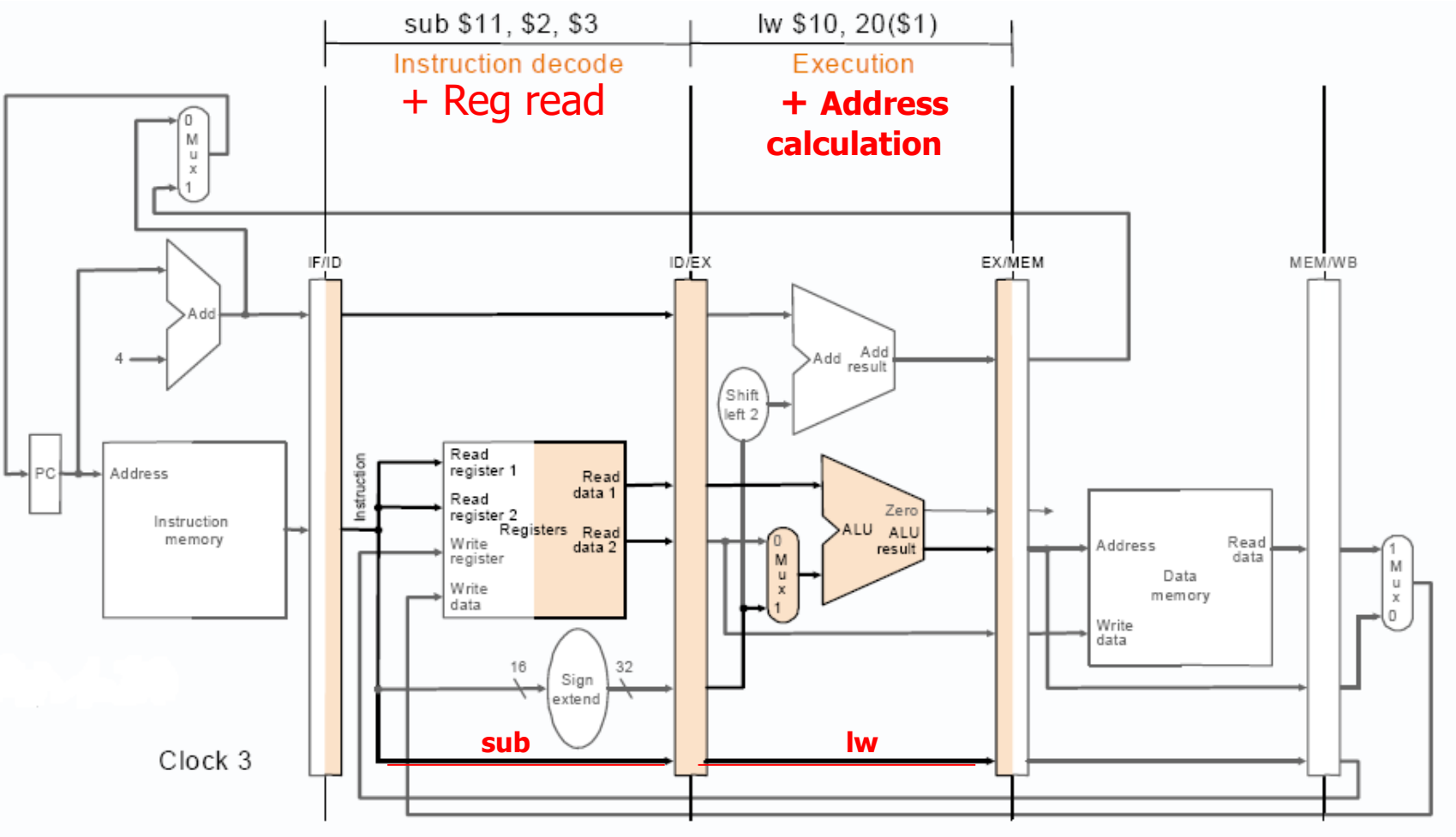
Example 1: Cycle 1



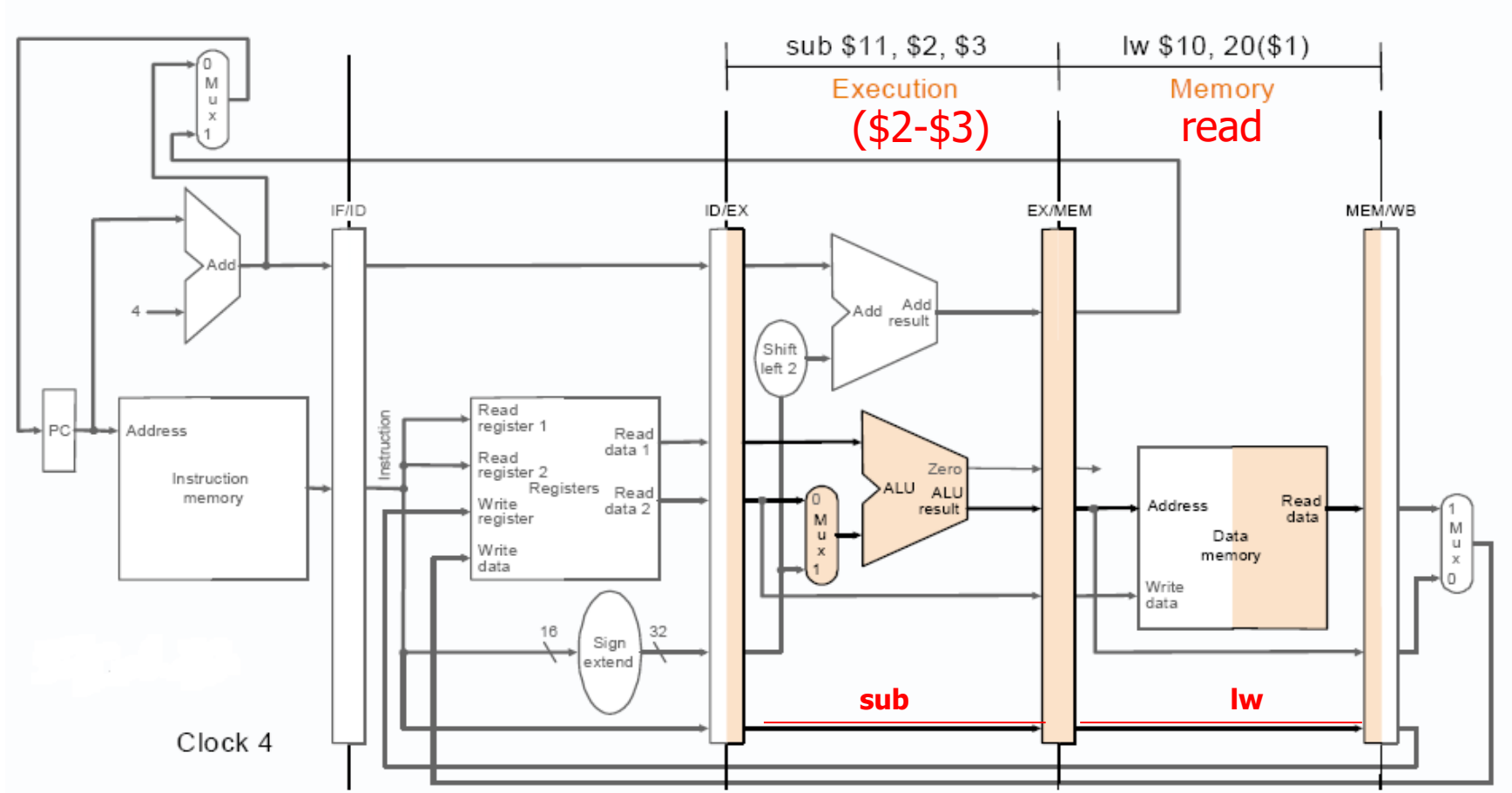
Example 1: Cycle 2



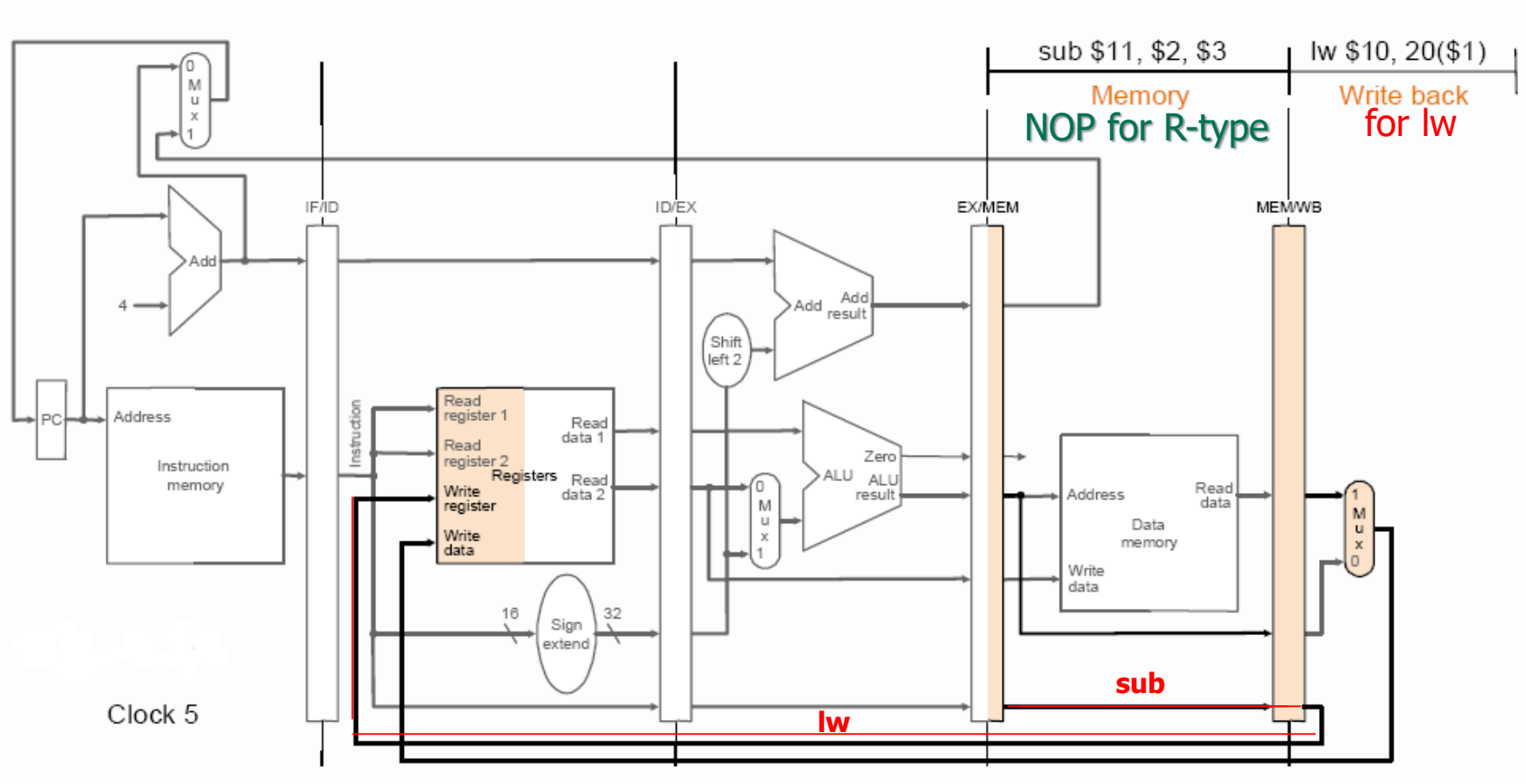
Example 1: Cycle 3



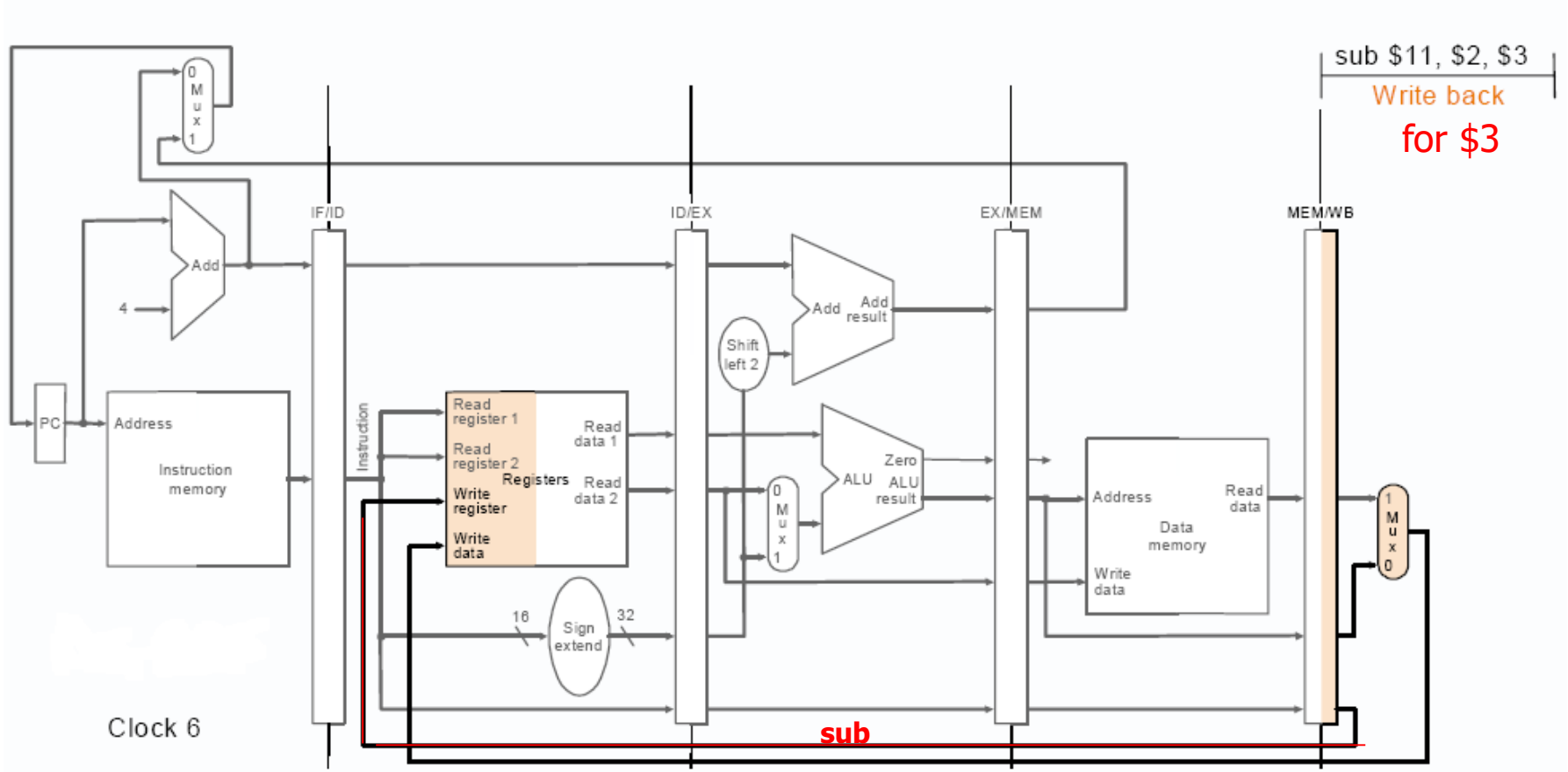
Example 1: Cycle 4

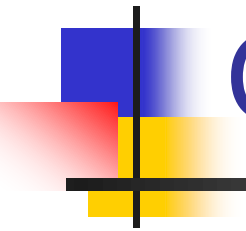


Example 1: Cycle 5



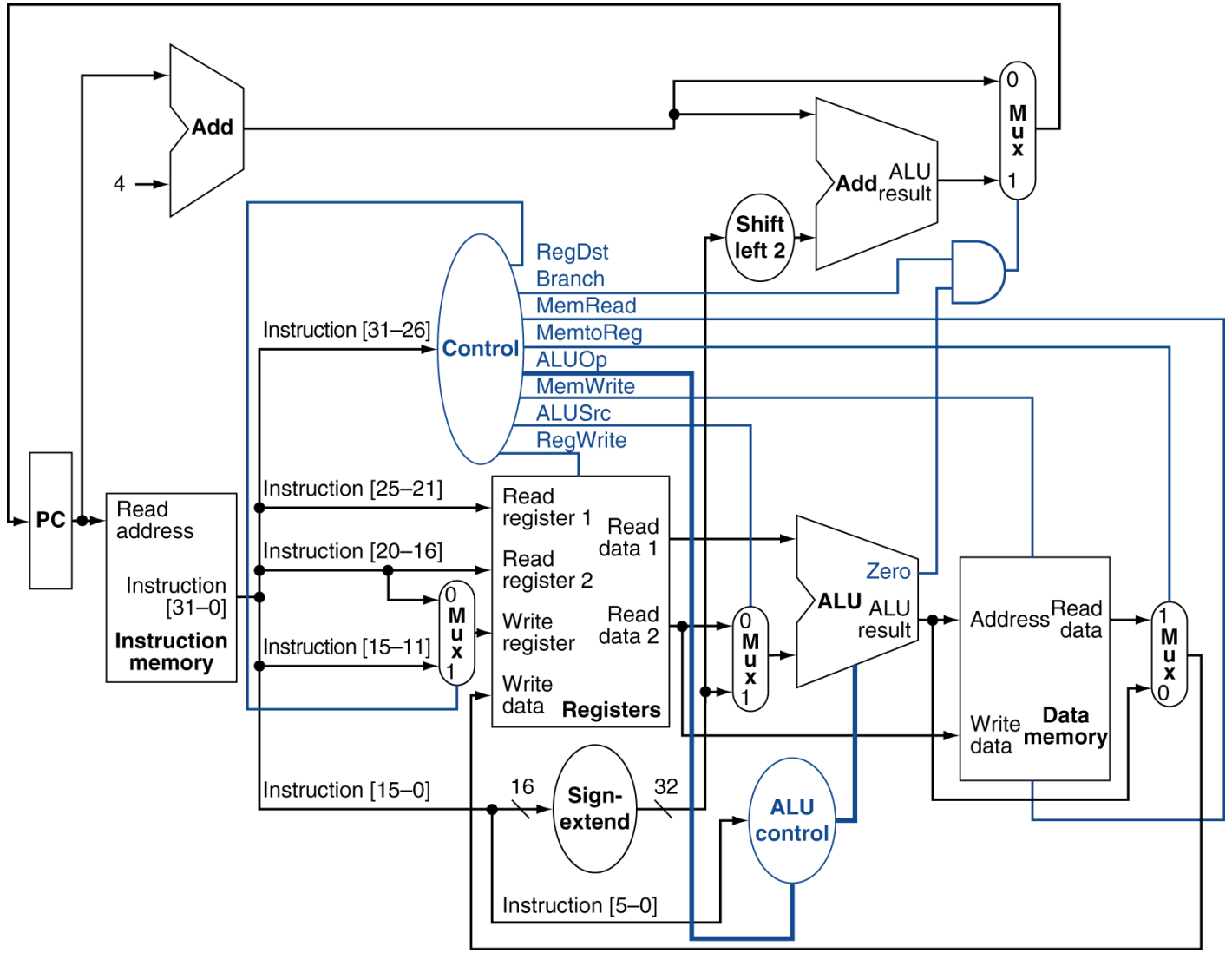
Example 1: Cycle 6





Control Signals in Pipleline

Review of MIPS Datapath With Control Signals



Control Signal Details (review)

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Group Signals According to Stages

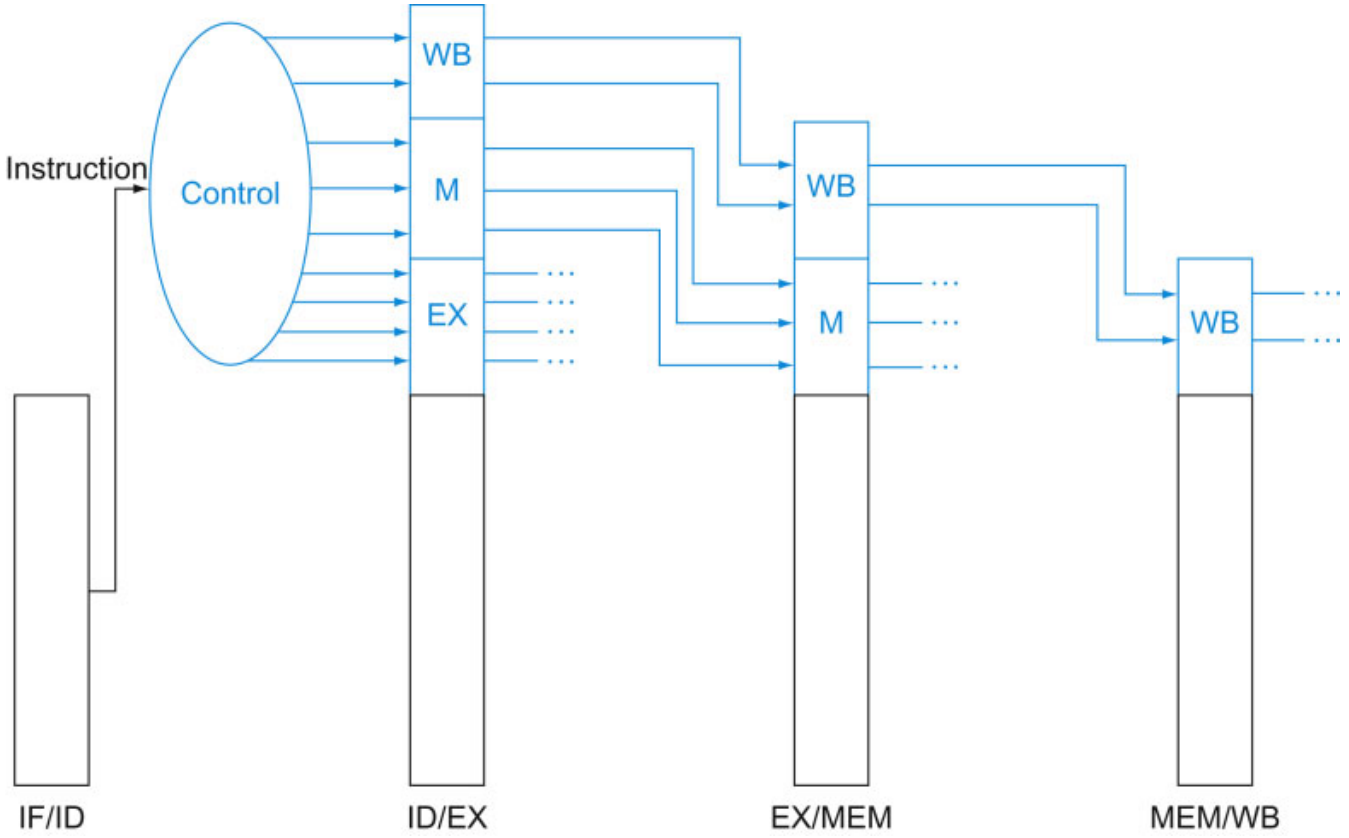
- Can use control signals of single-cycle CPU

	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
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

EX (type) MEM (type) WB (type)

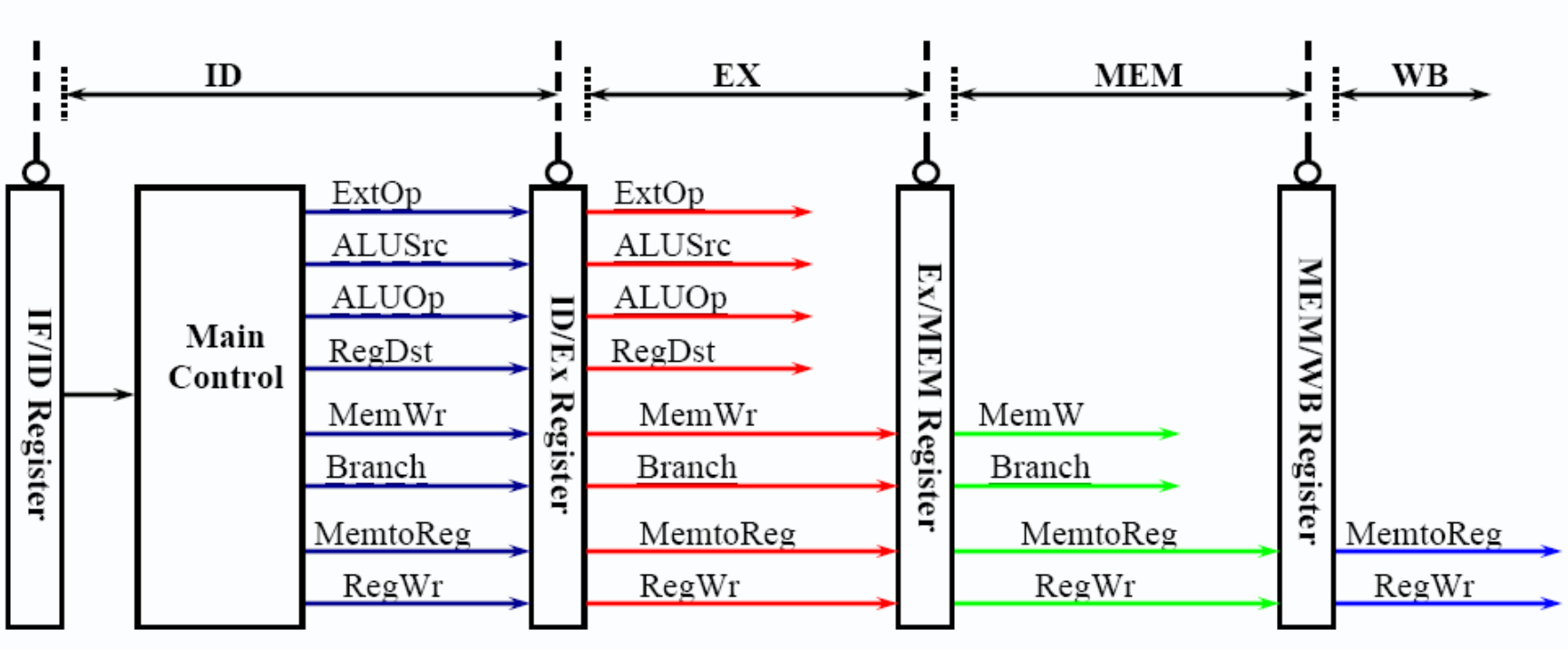
Data Stationary Control

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

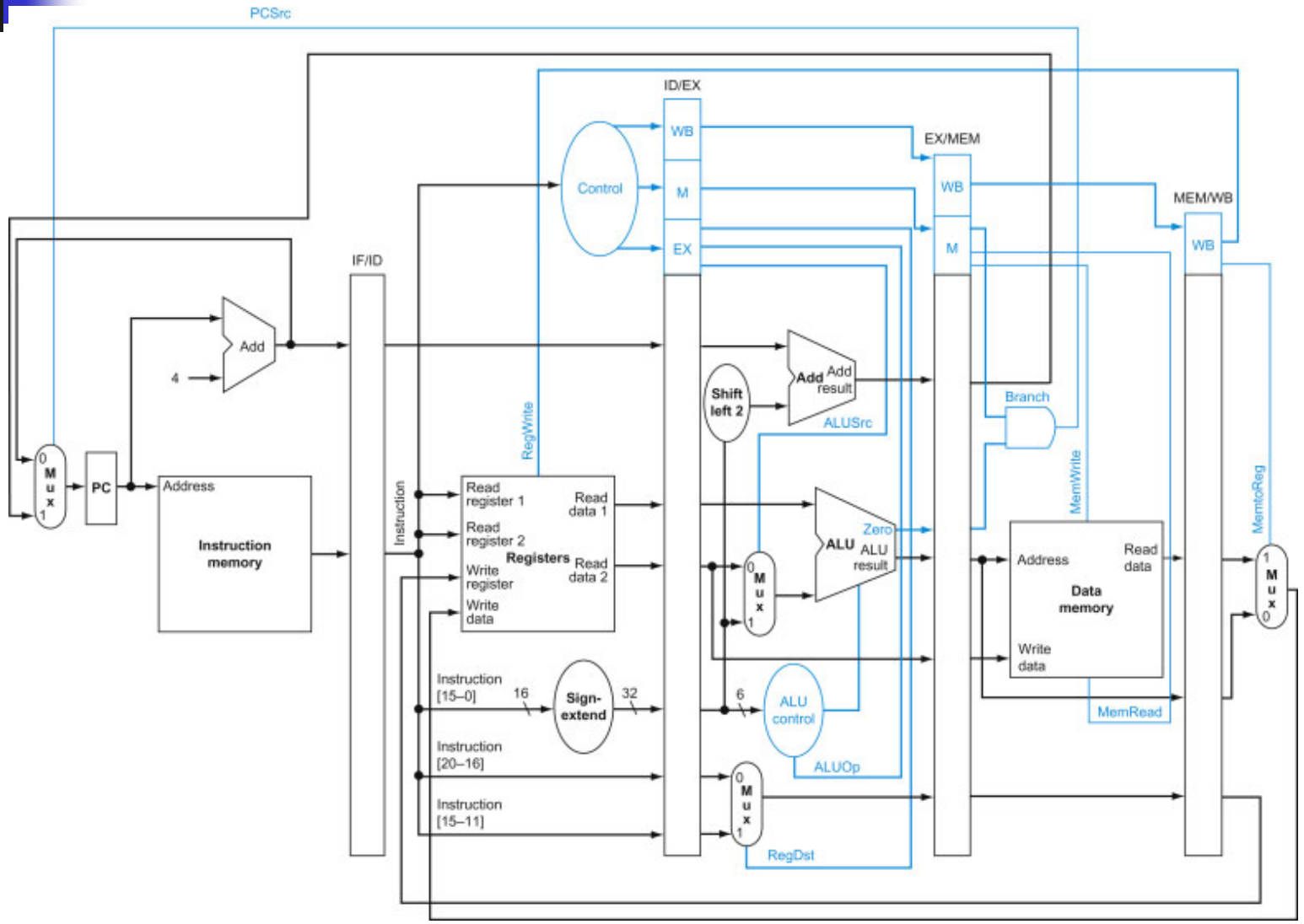


Data Stationary Control (cont.)

- 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



Pipelined Datapath with Control Signals

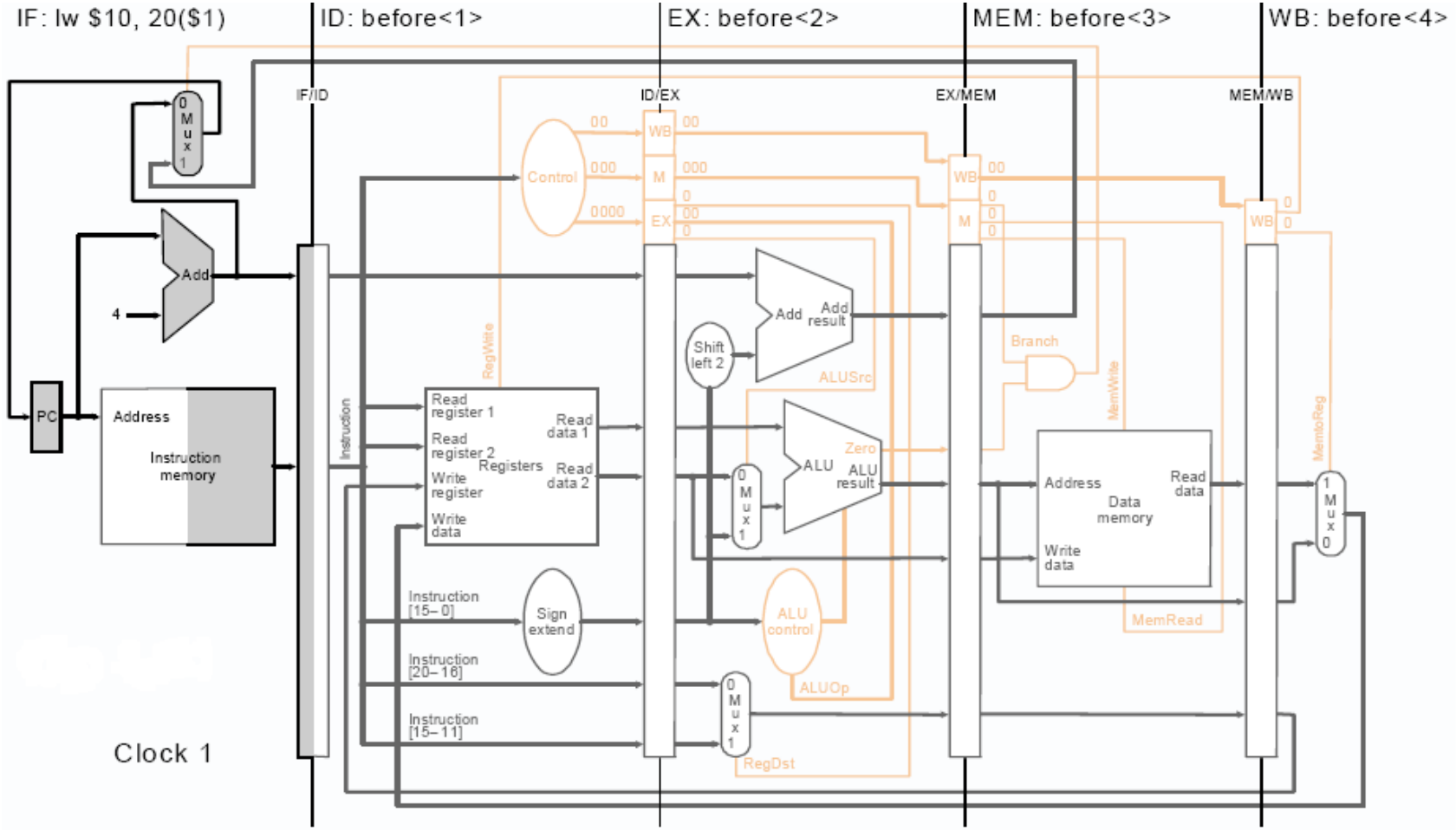


Let's Try it Out

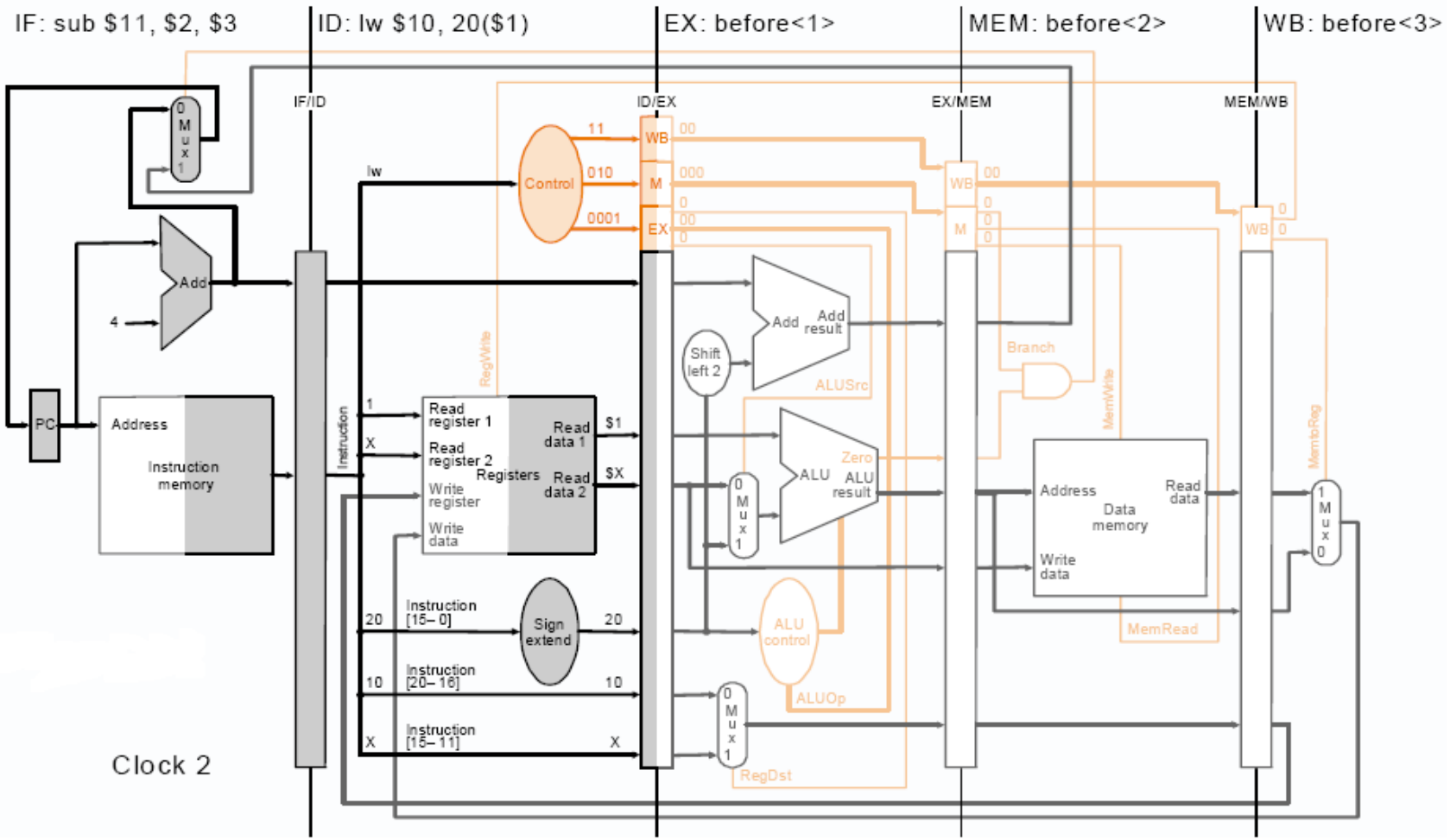
Sample Assembly Program (watch fill/flush)

```
lw    $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or    $13, $6, $7
add $14, $8, $9
```

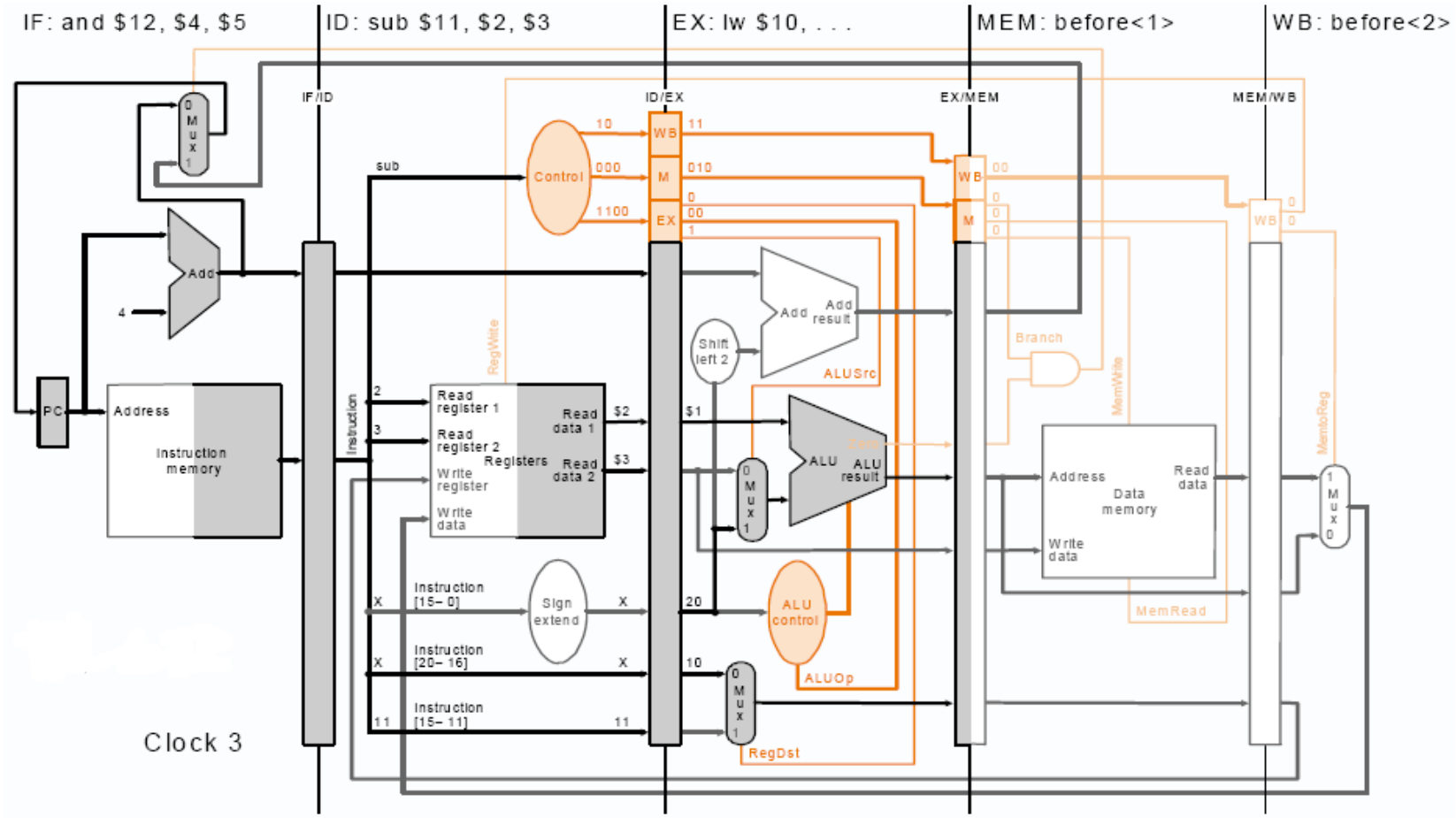
Example 2: Cycle 1



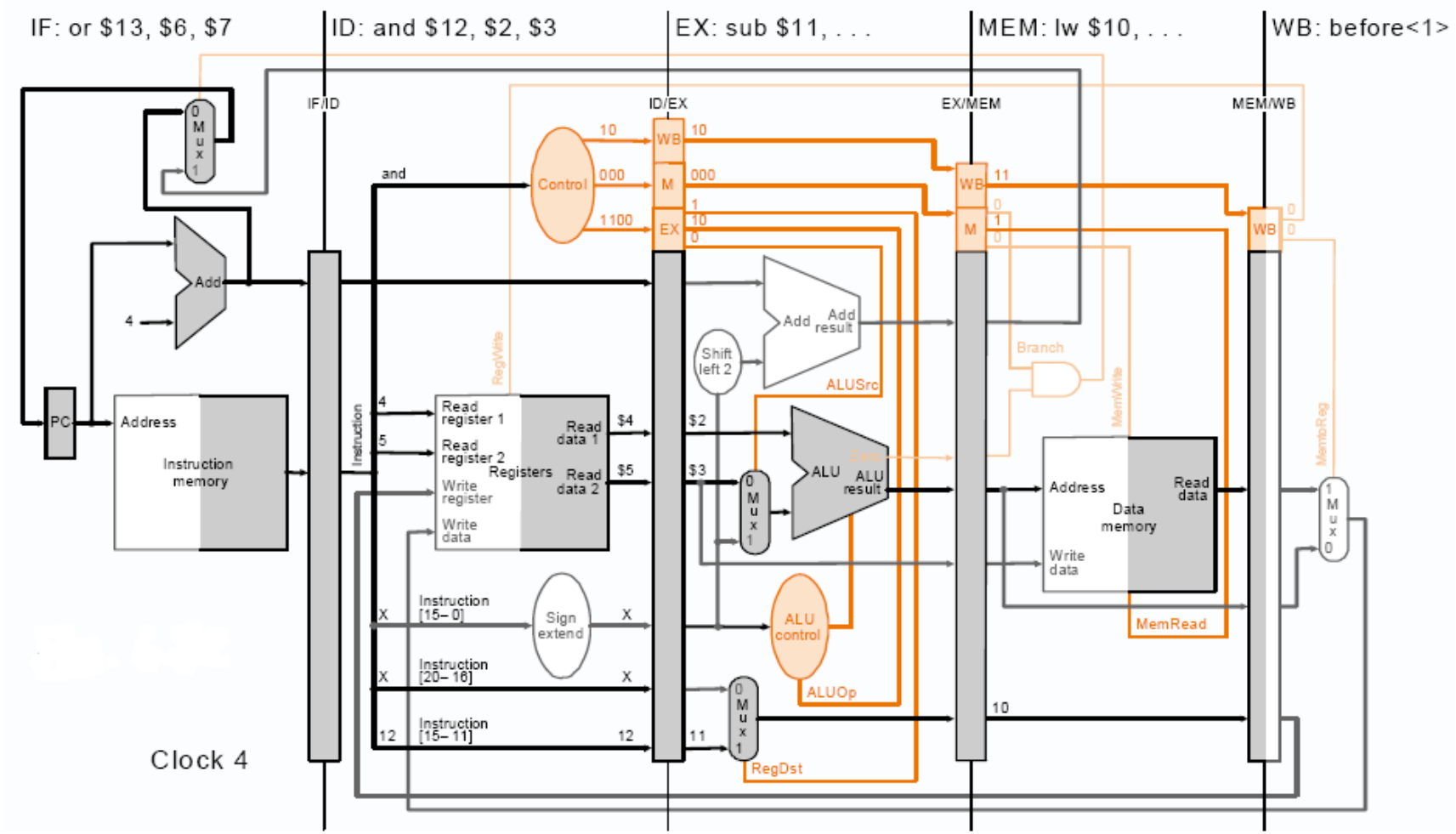
Example 2: Cycle 2



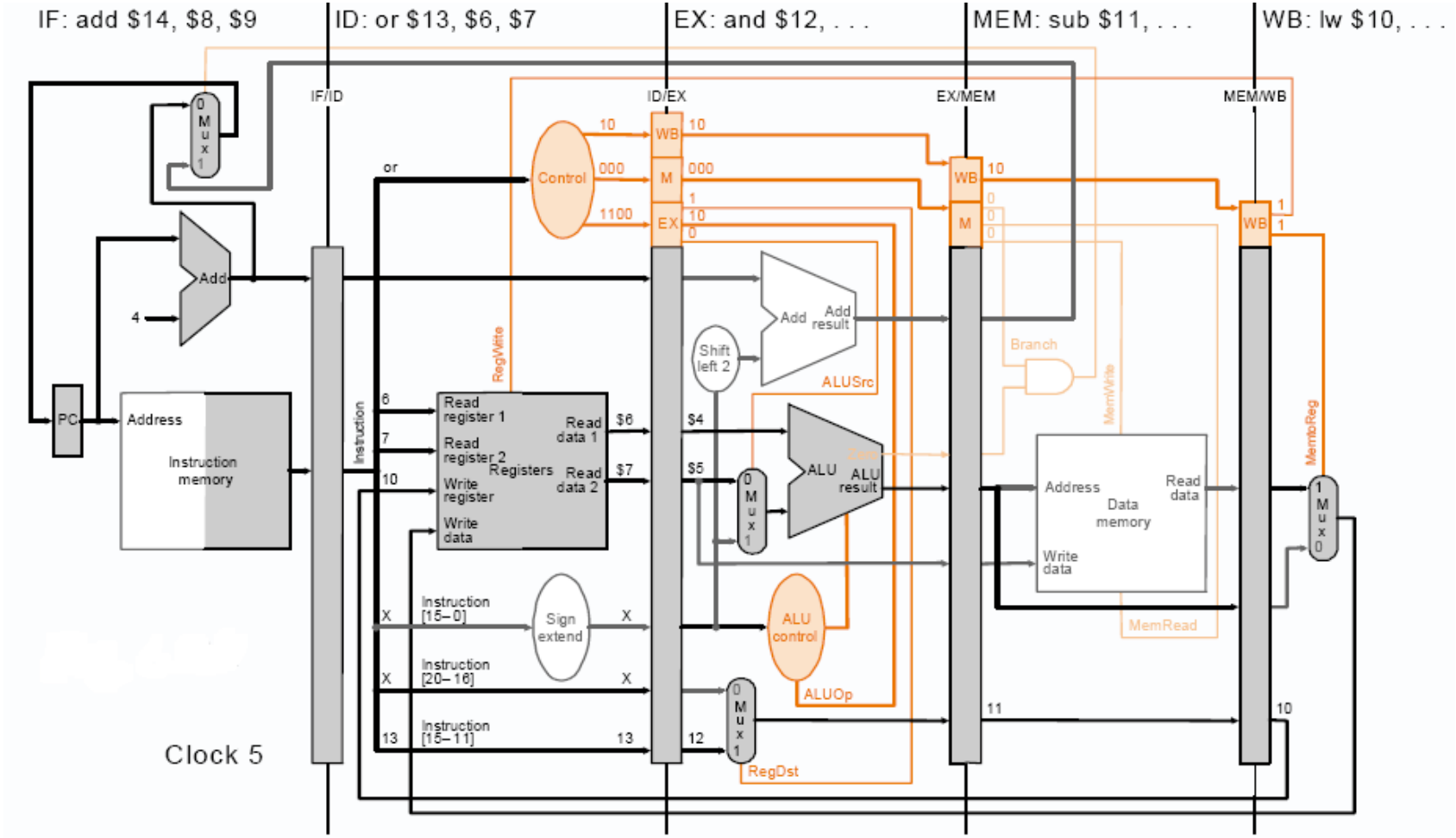
Example 2: Cycle 3



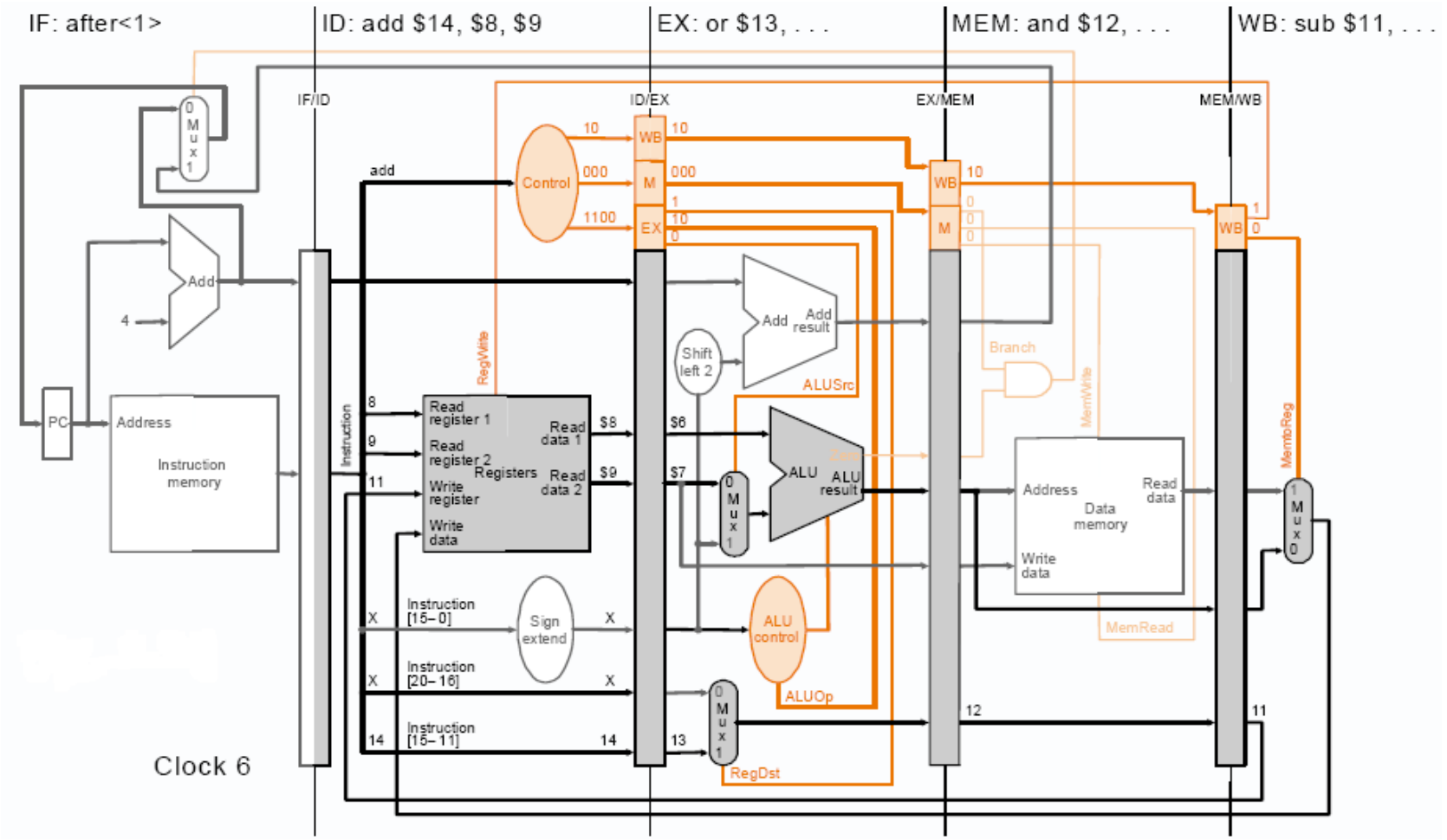
Example 2: Cycle 4



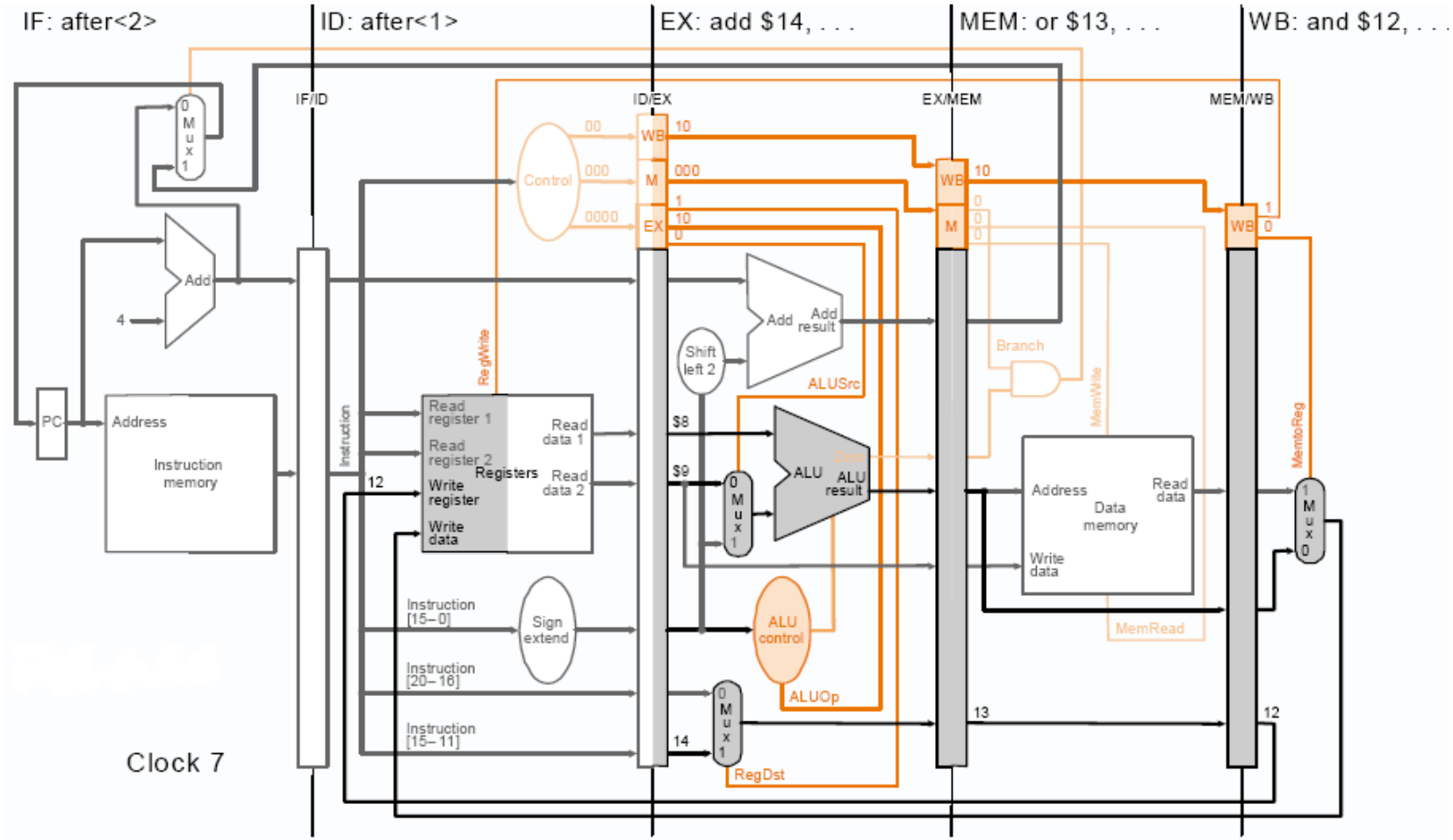
Example 2: Cycle 5



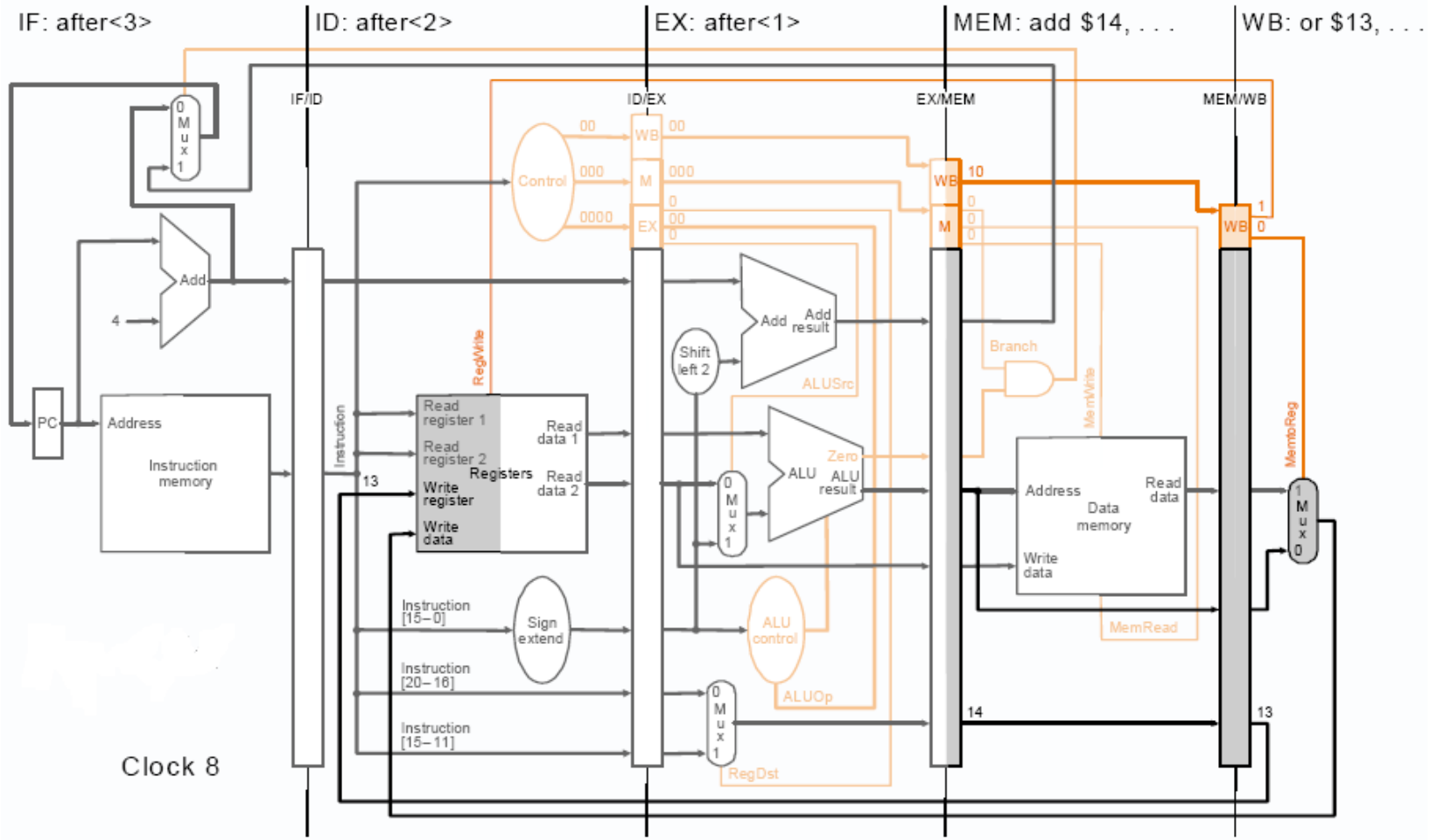
Example 2: Cycle 6



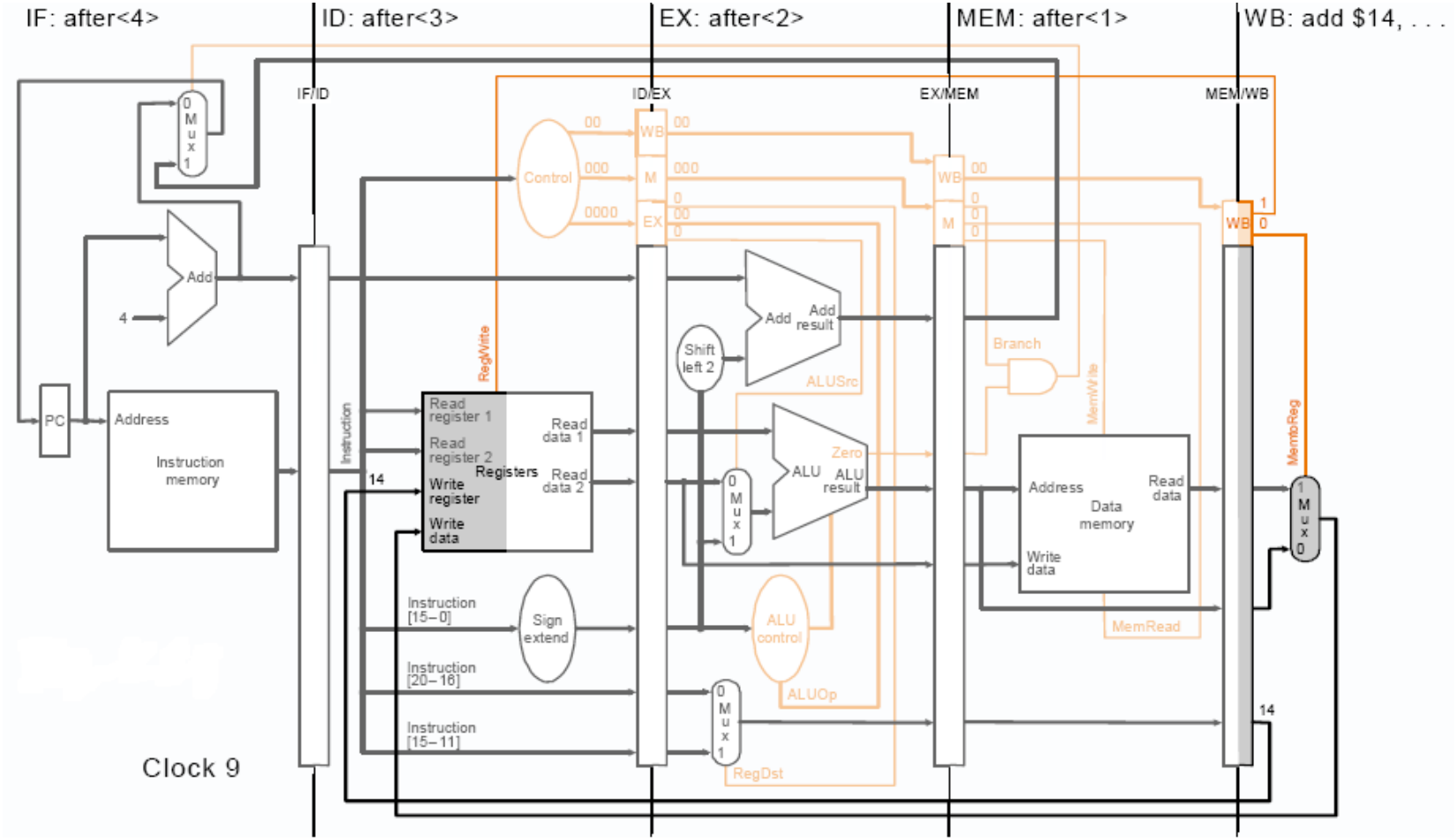
Example 2: Cycle 7



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

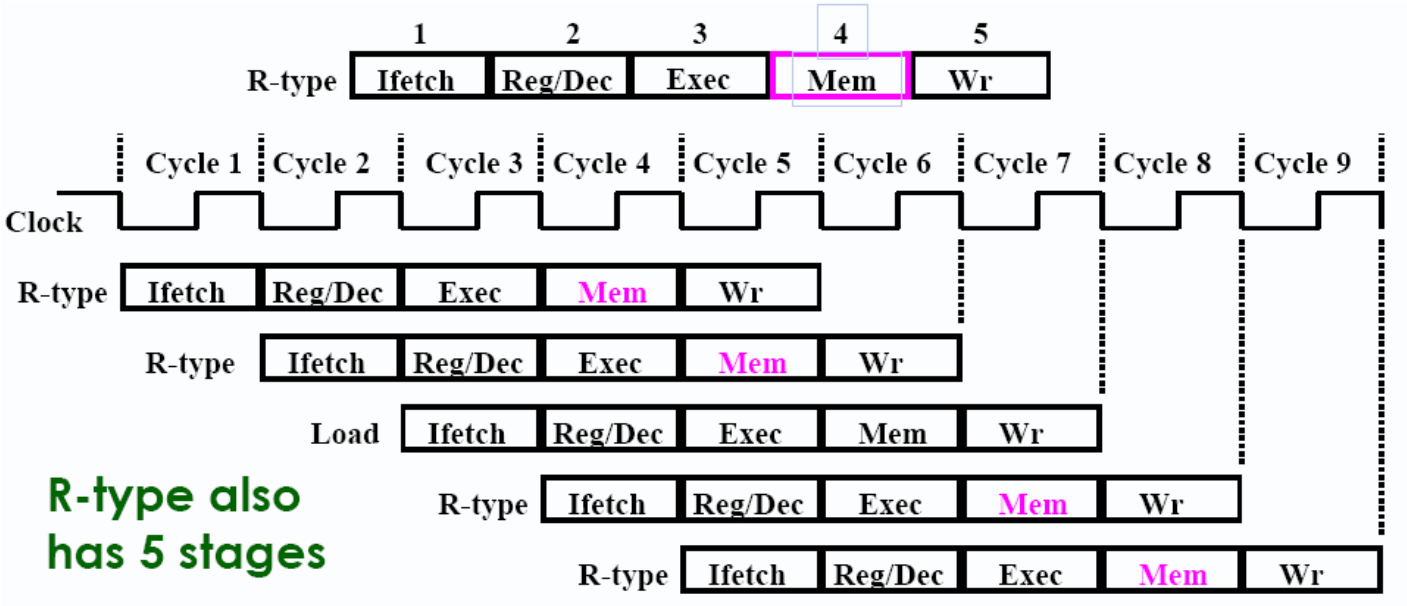
Outline

- 4.5 An Overview of Pipelining
- 4.6 A Pipelined Datapath & Control
- 4.7 Data Hazards and Forwarding v.s. Stalls
- 4.8 Control Hazards
- 4.9 Exceptions

Pipeline Hazards

- **Structural hazard**
  - An occurrence in which a planned instruction cannot execute in the proper clock cycle because the hardware **cannot** support the combination of instructions that are set to execute in the given clock cycle (can be avoided in advance).

- Ex:



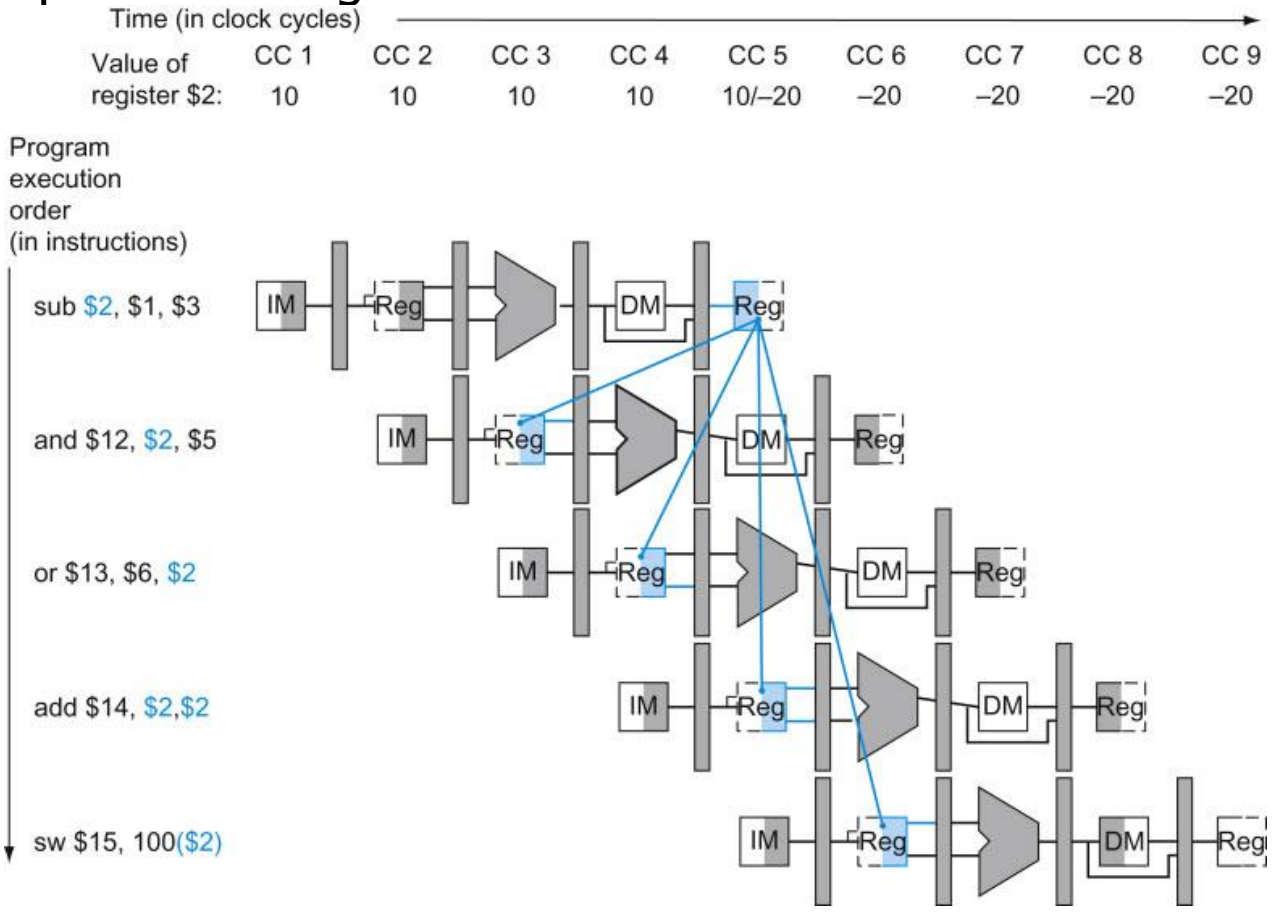
Data hazard

- **Data hazard**
  - Also called pipeline data hazard. An occurrence in which a planned instruction cannot execute in the proper clock cycle because **data** that is needed to execute the instruction **is not yet available (dynamic).**
- **Load-use** data hazard
  - A specific form of data hazard in which the data requested by a load instruction (***lw***) (read data from main memory) has not yet become available when it is requested.

- *Solution 1: Pipeline stall*
  - Also called *bubble*. A stall initiated in order to resolve a hazard
- *Solution 2: Forwarding*
  - Also called bypassing. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible register or memory.

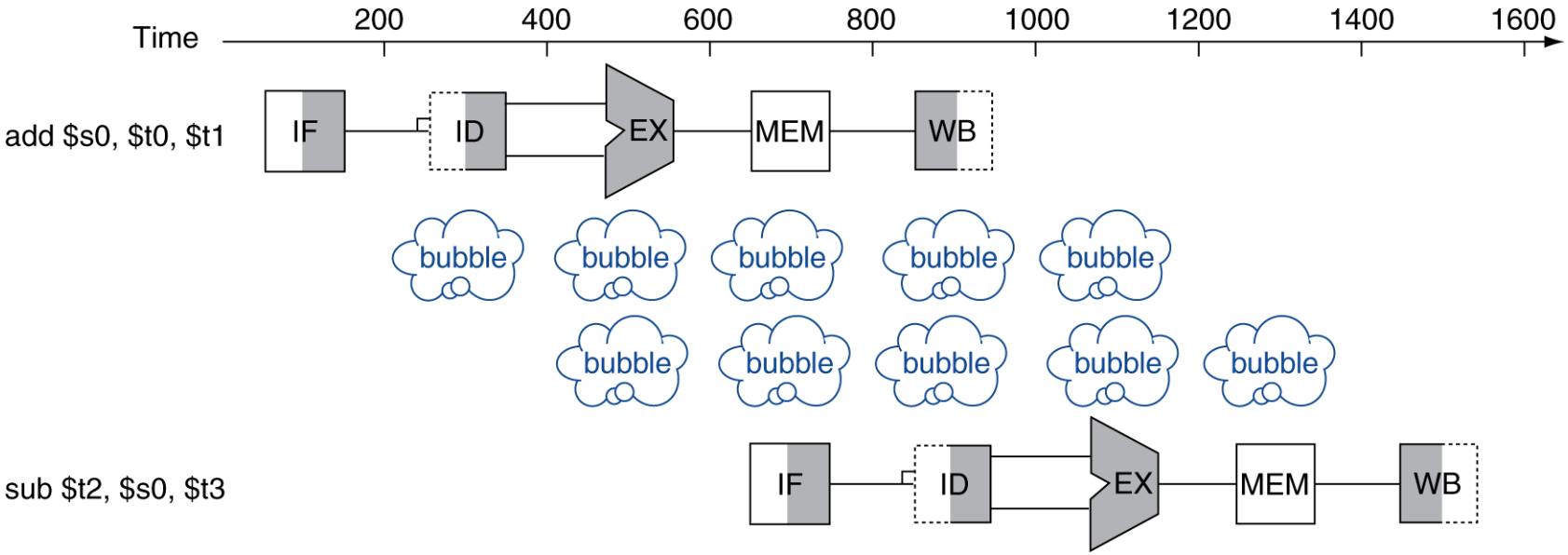
Data Hazards

- Order of operand accesses changed by pipeline
  - Starting next instruction before first is finished
  - Dependencies “go backward in time”



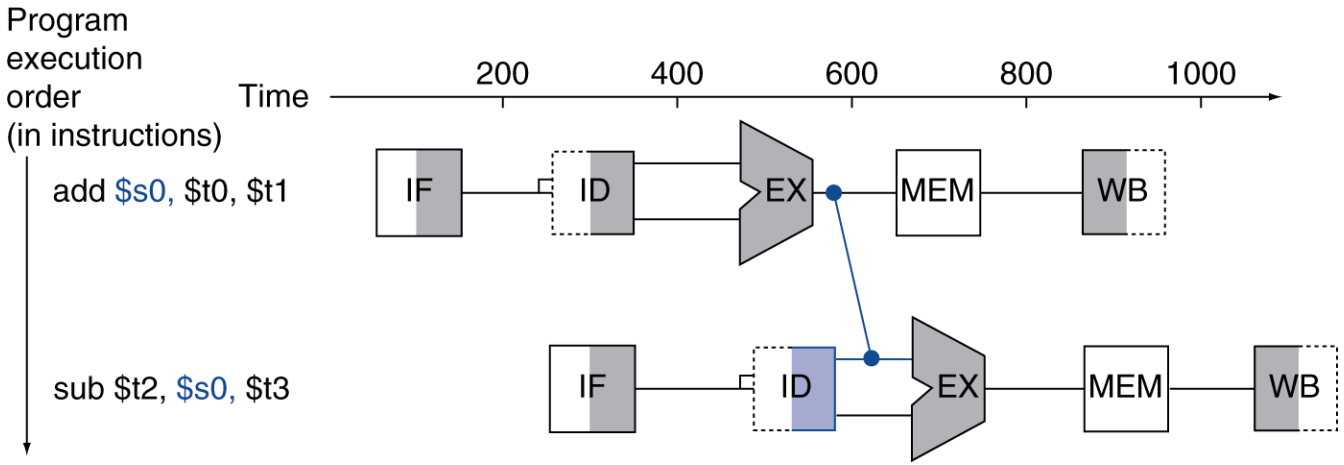
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - `add $s0, $t0, $t1`
    `sub $t2, $s0, $t3`



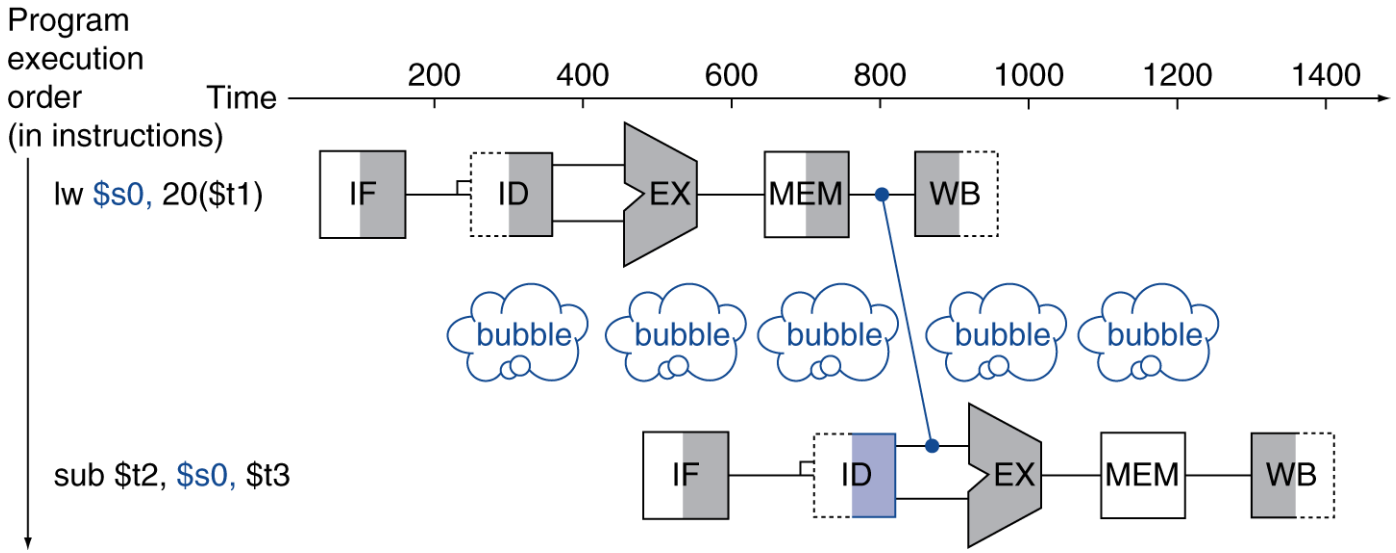
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!

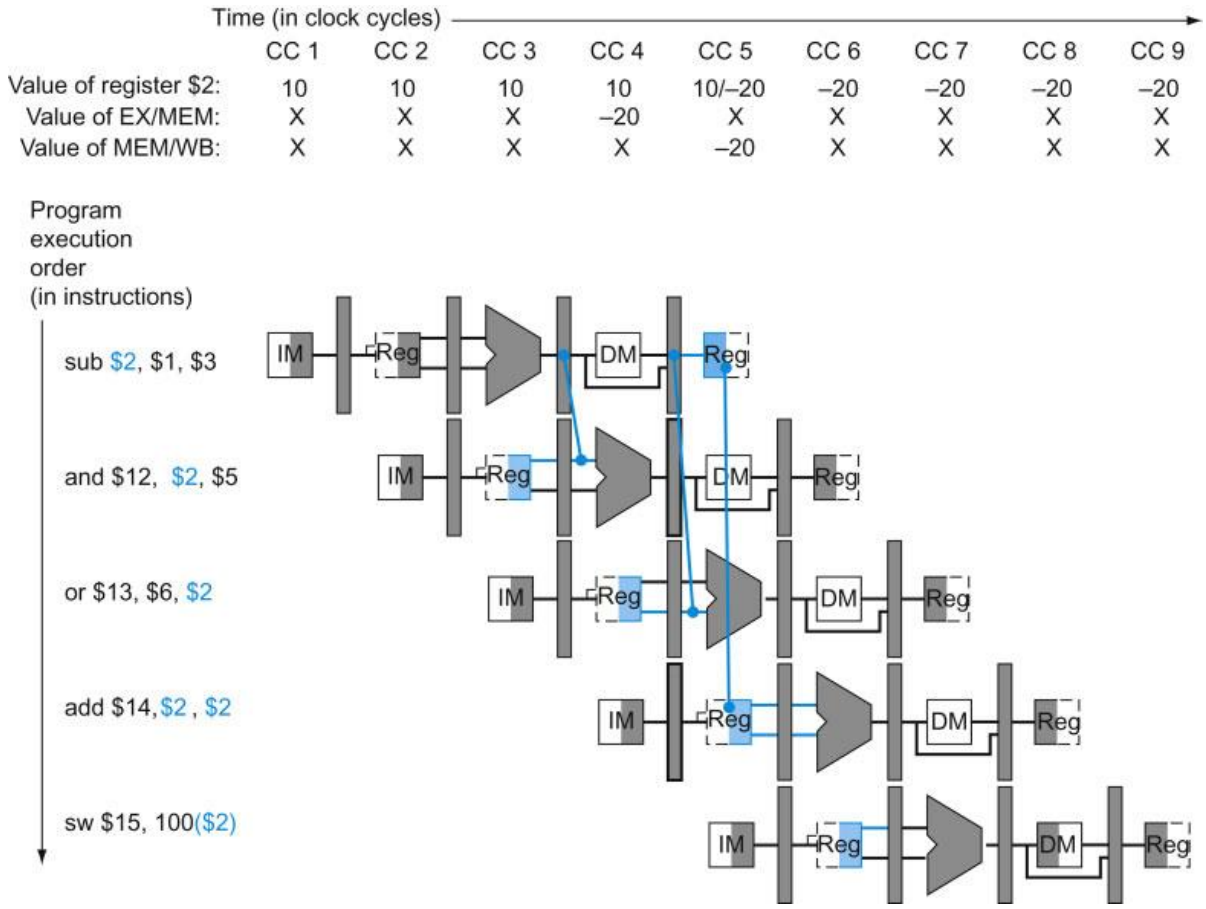


Handling Data Hazards

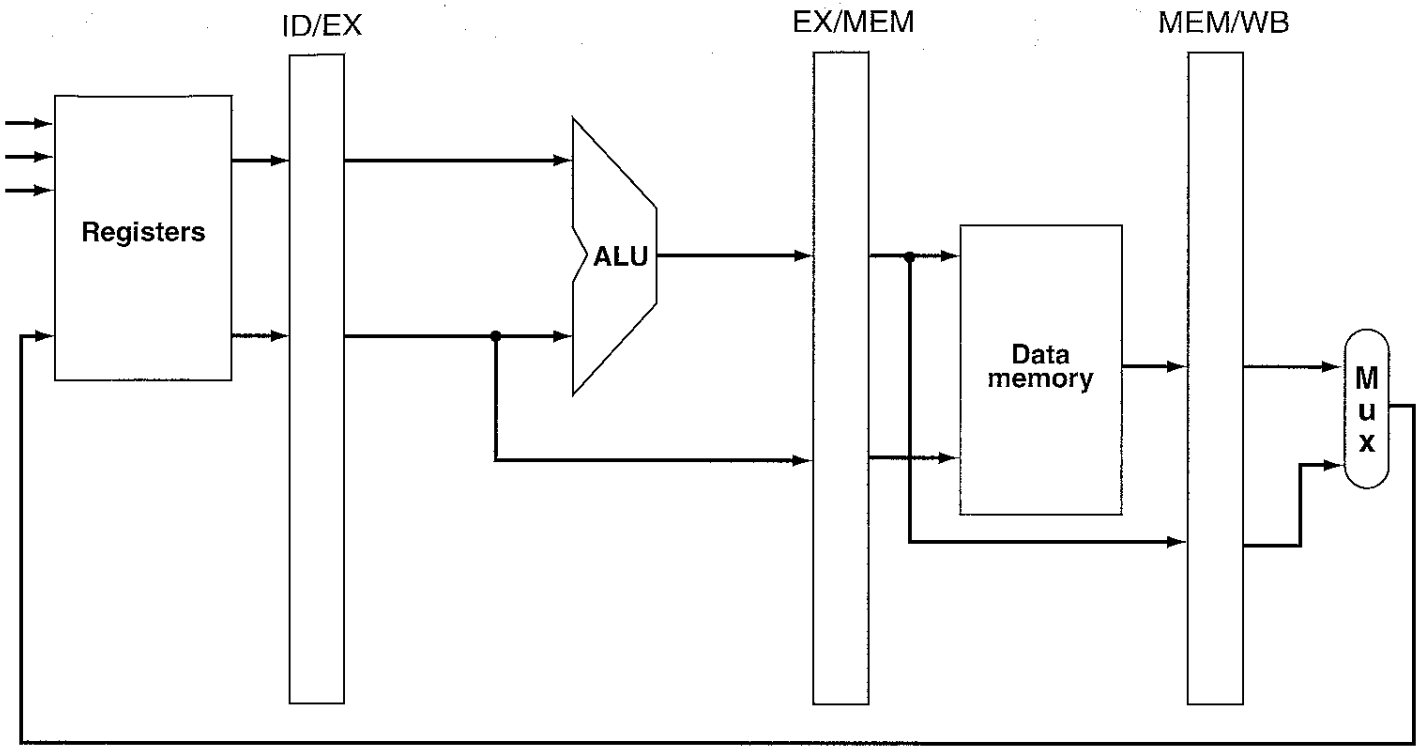
- Solutions
  - Compiler inserts NOP (or reorder)
  - Forwarding (bypass)
  - Stall (insert Bubble)

Resolving Hazards in Hardware: Forwarding (bypassing)

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



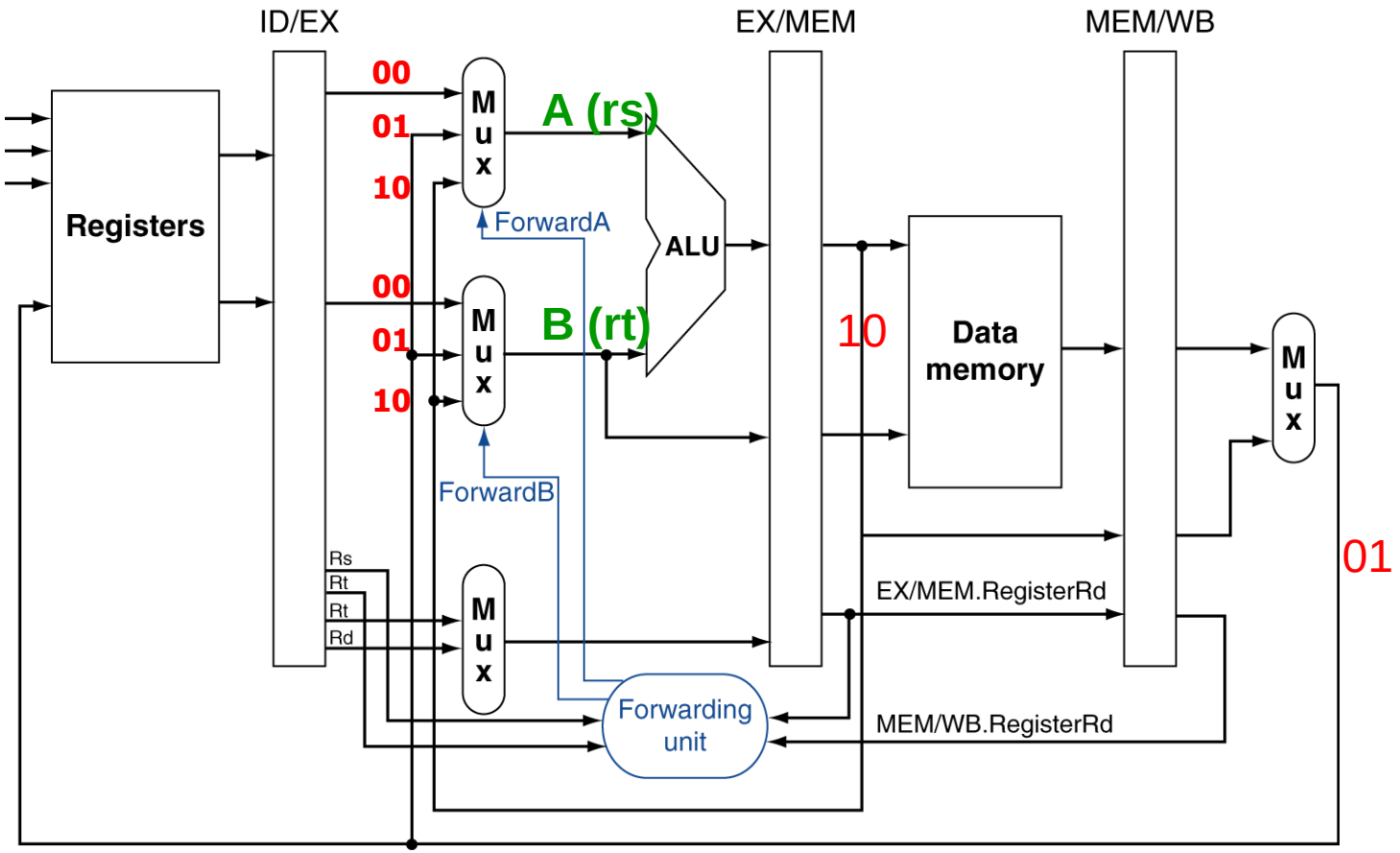
No Forwarding



a. No forwarding

With Forwarding

- Forwarding: input to ALU from any pipe registers
  - → Add multiplexors to ALU inputs



b. With forwarding

Control Signals for FWD unit

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

FIGURE 4.55 The control values for the forwarding multiplexors in Figure 4.54. The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

Detecting Data Hazards

- Hazard conditions:
  - EX/MEM.RegisterRd = ID/EX.RegisterRs (1a)
  - EX/MEM.RegisterRd = ID/EX.RegisterRt (1b)
  - MEM/WB.RegisterRd = ID/EX.RegisterRs (2a)
  - MEM/WB.RegisterRd = ID/EX.RegisterRt (2b)

- 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 hazard):
  - If (EX/MEM.RegWrite
    and (EX/MEM.RegRd≠0)
    and (EX/MEM.RegRd=ID/EX.RegRs)

    ForwardA = 10
  - If (EX/MEM.RegWrite
    and (EX/MEM.RegRd≠0)
    and (EX/MEM.RegRd=ID/EX.RegRt)

    ForwardB = 10

Detecting Data Hazards (cont.)

- Hazard conditions using control signals:
- At MEM stage (MEM hazard):
  - If (MEM/WB.RegWrite
    and (MEM/WB.RegRd ≠ 0)
    and (MEM/WB.RegRd = ID/EX.RegRs)

    ForwardA = 01
  - If (MEM/WB.RegWrite
    and (MEM/WB.RegRd ≠ 0)
    and (MEM/WB.RegRd = ID/EX.RegRt)

    ForwardB = 01

Forwarding Logic

- More complicated Control signals for forwarding (ref):
  - Forward from MEM stage because it is the more recent result

```
add $1,$1,$2;
add $1,$1,$3;
add $1,$1,$4;
```

⇒ The result is forwarded from the MEM stage (see p.298)

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Handling Data Hazards

- Detect (see example on p. 294)

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

The first hazard in the sequence on page 293 is on register $2, between the result of `sub $2,$1,$3` and the first read operand of `and $12,$2,$5`. This hazard can be detected when the `and` instruction is in the EX stage and the prior instruction is in the MEM stage, so this is hazard 1a:

EX/MEM.RegisterRd = ID/EX.RegisterRs = $2

Detect (see example on p. 294)

Dependence Detection

Classify the dependences in this sequence from page 363:

EXAMPLE

```
sub   $2,    $1, $3   # Register $2 set by sub
and   $12,   $2, $5   # 1st operand($2) set by sub
or    $13,   $6, $2   # 2nd operand($2) set by sub
add   $14,   $2, $2   # 1st($2) & 2nd($2) set by sub
sw    $15,   100($2) # Index($2) set by sub
```

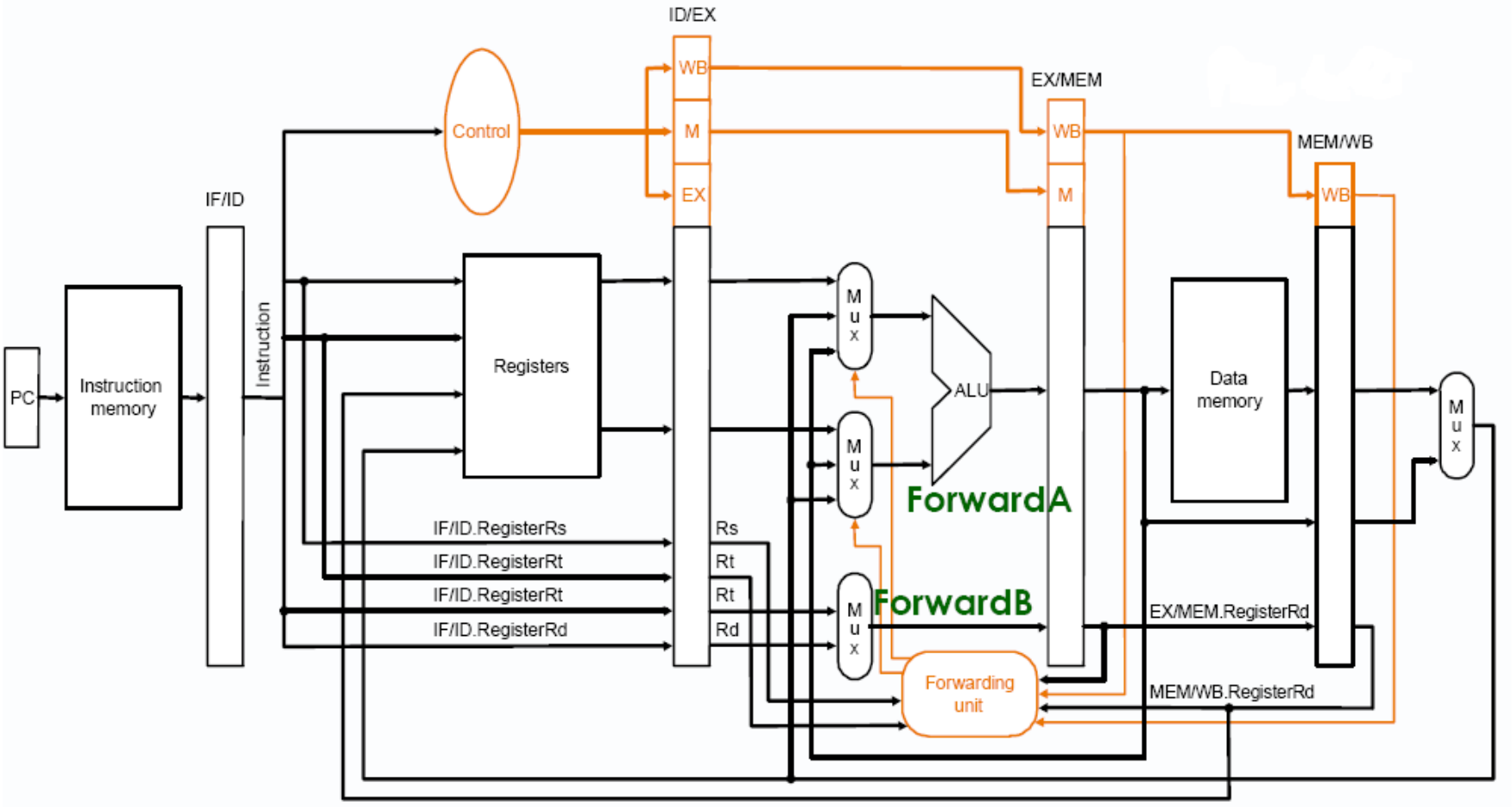
ANSWER

As mentioned above, the sub-and is a type 1a hazard. The remaining hazards are as follows:

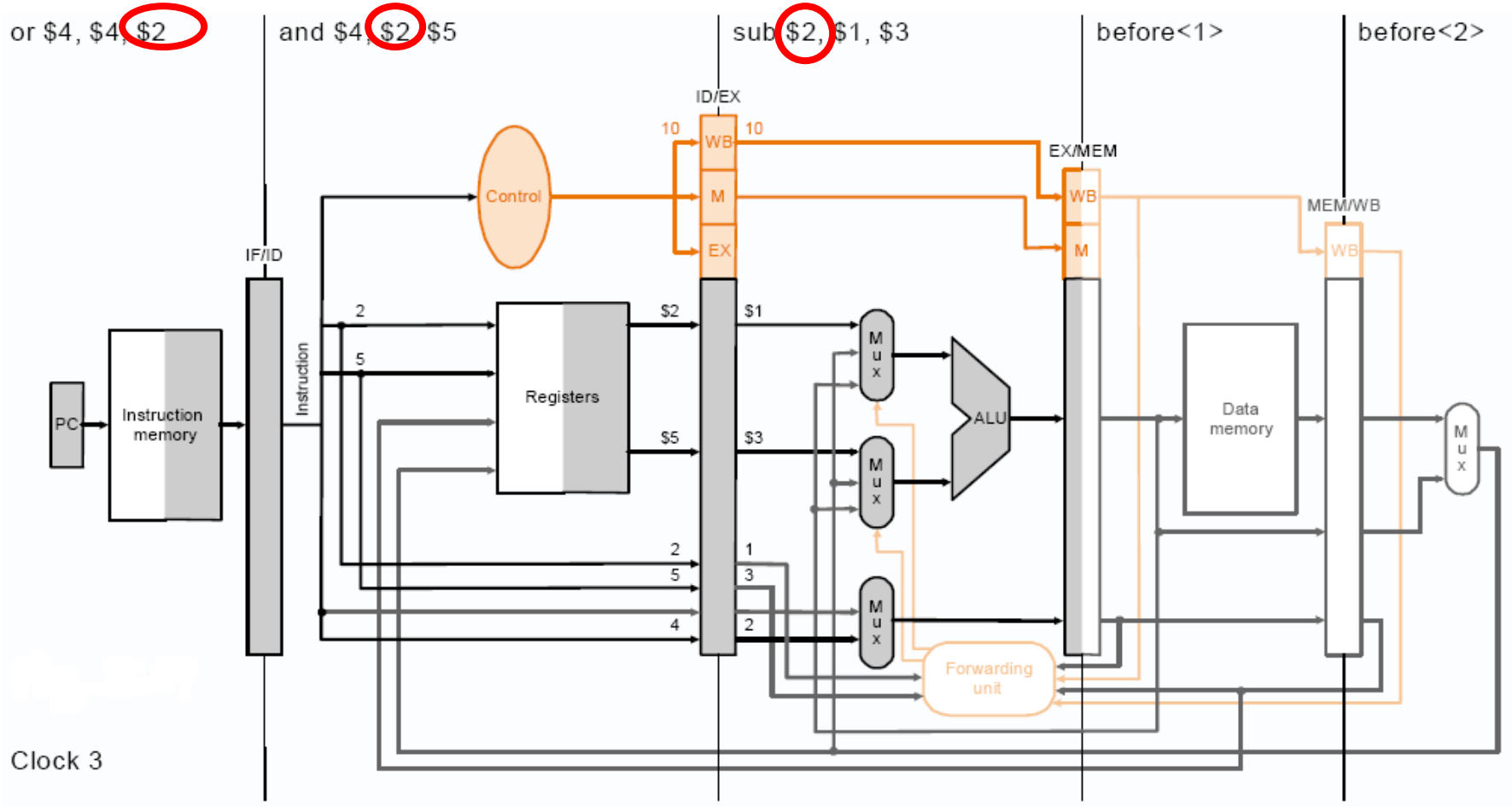
- The sub-or is a type 2b hazard:

  MEM/WB.RegisterRd = ID/EX.RegisterRt = $2
- The two dependences on sub-add are not hazards because the register file supplies the proper data during the ID stage of add.
- There is no data hazard between sub and sw because sw reads $2 the clock cycle *after* sub writes $2.

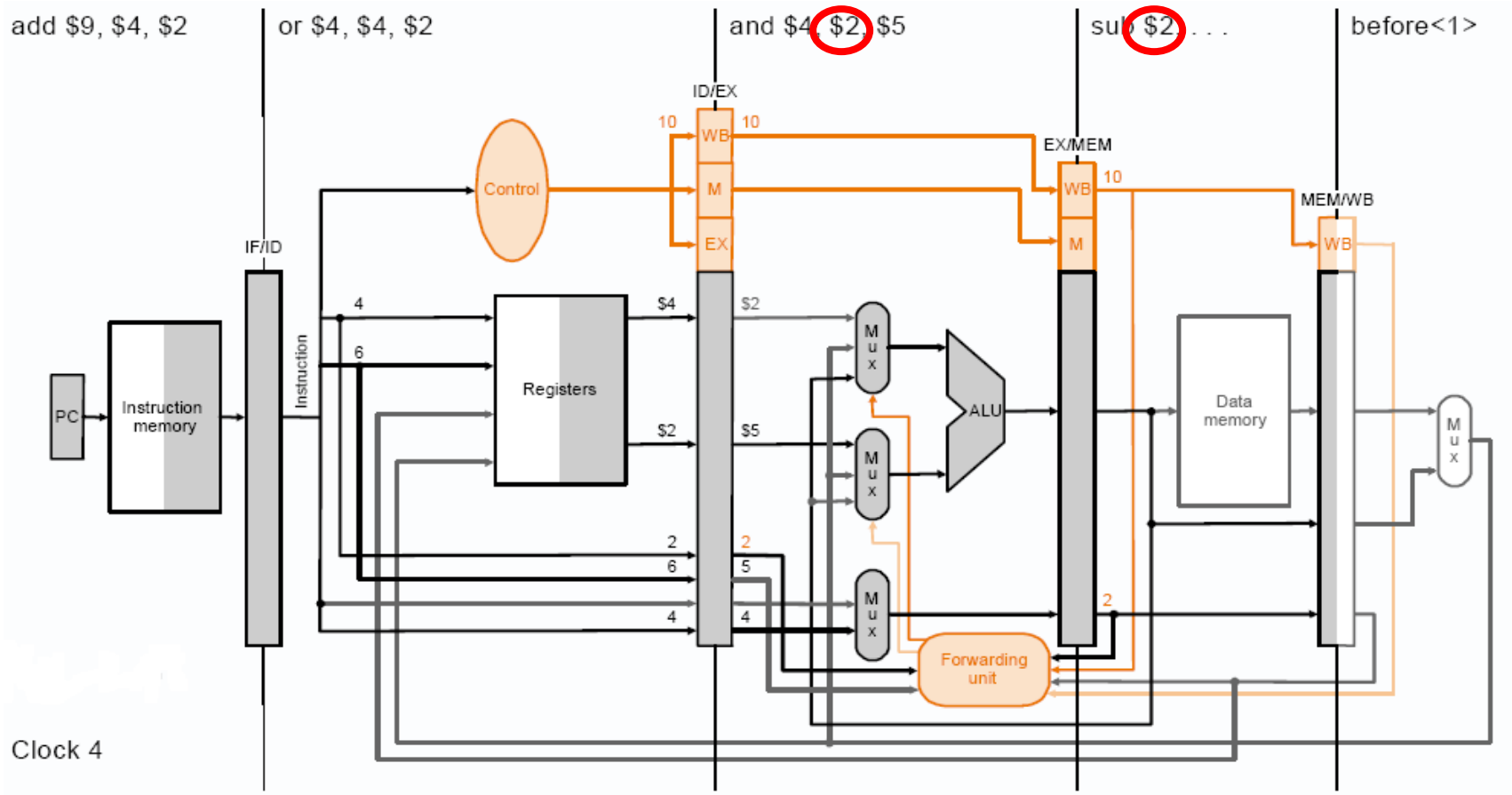
Pipeline with Forwarding



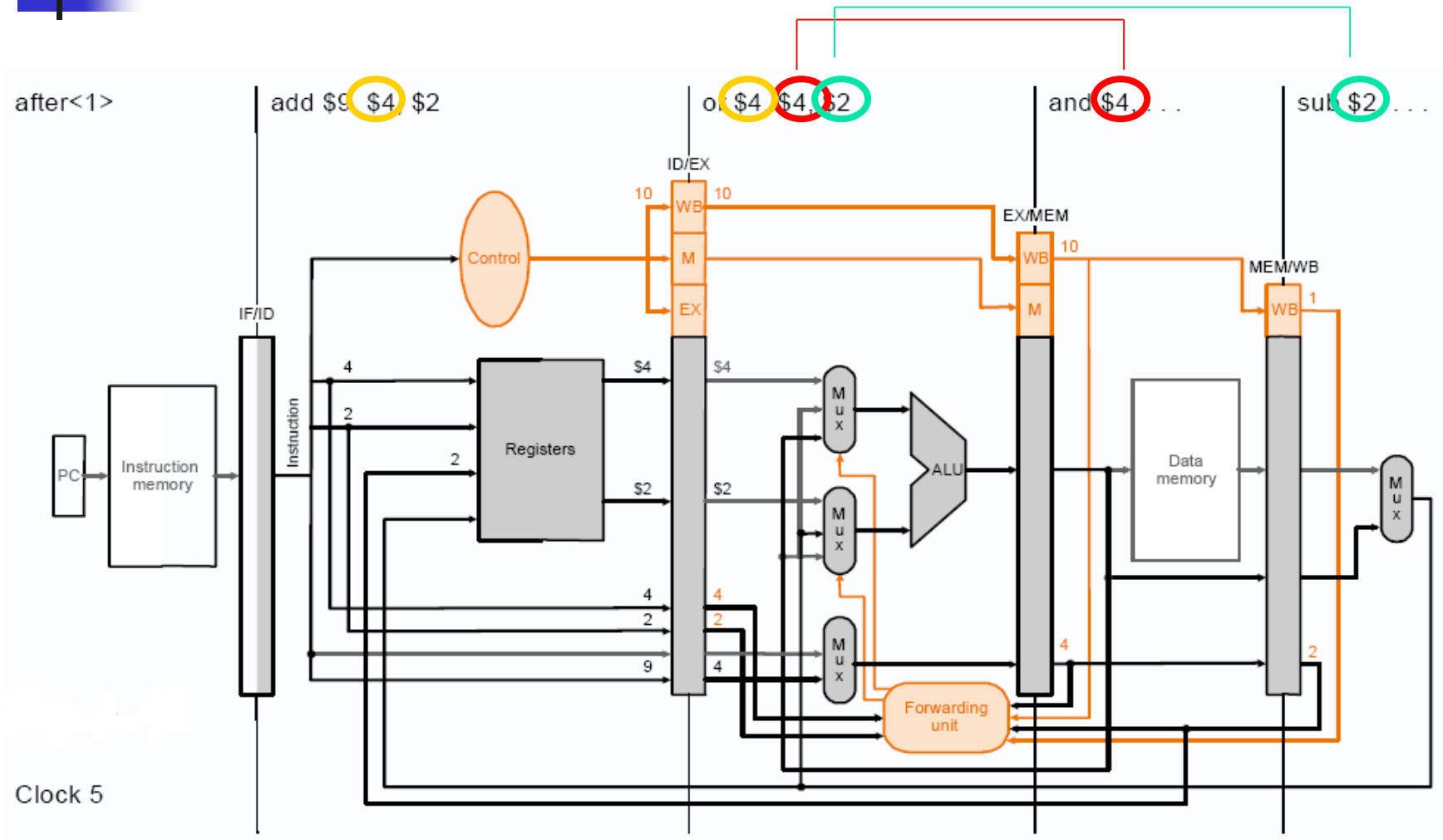
Example 3: Cycle 3



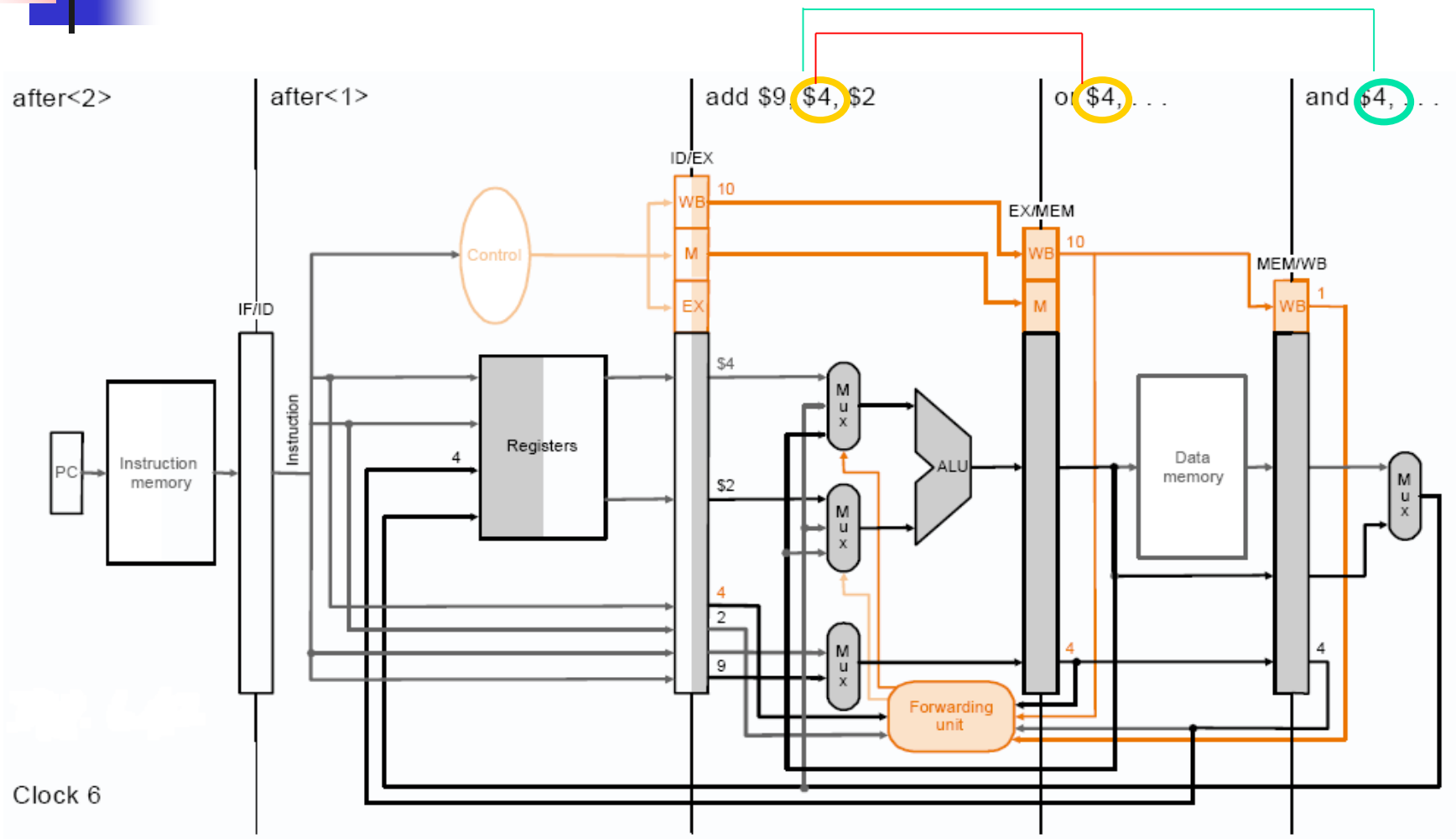
Example 3: Cycle 4



Example 3: Cycle 5

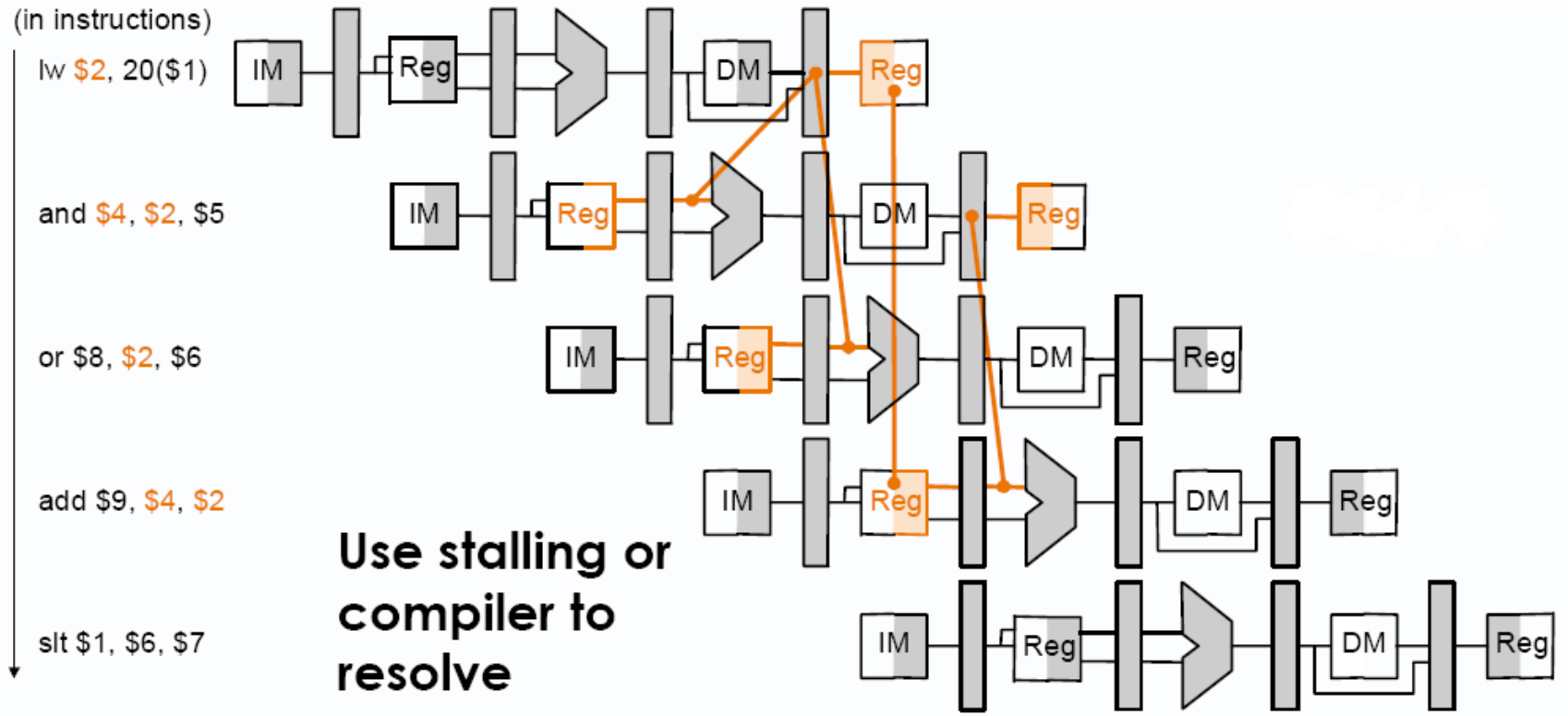


Example 3: Cycle 6

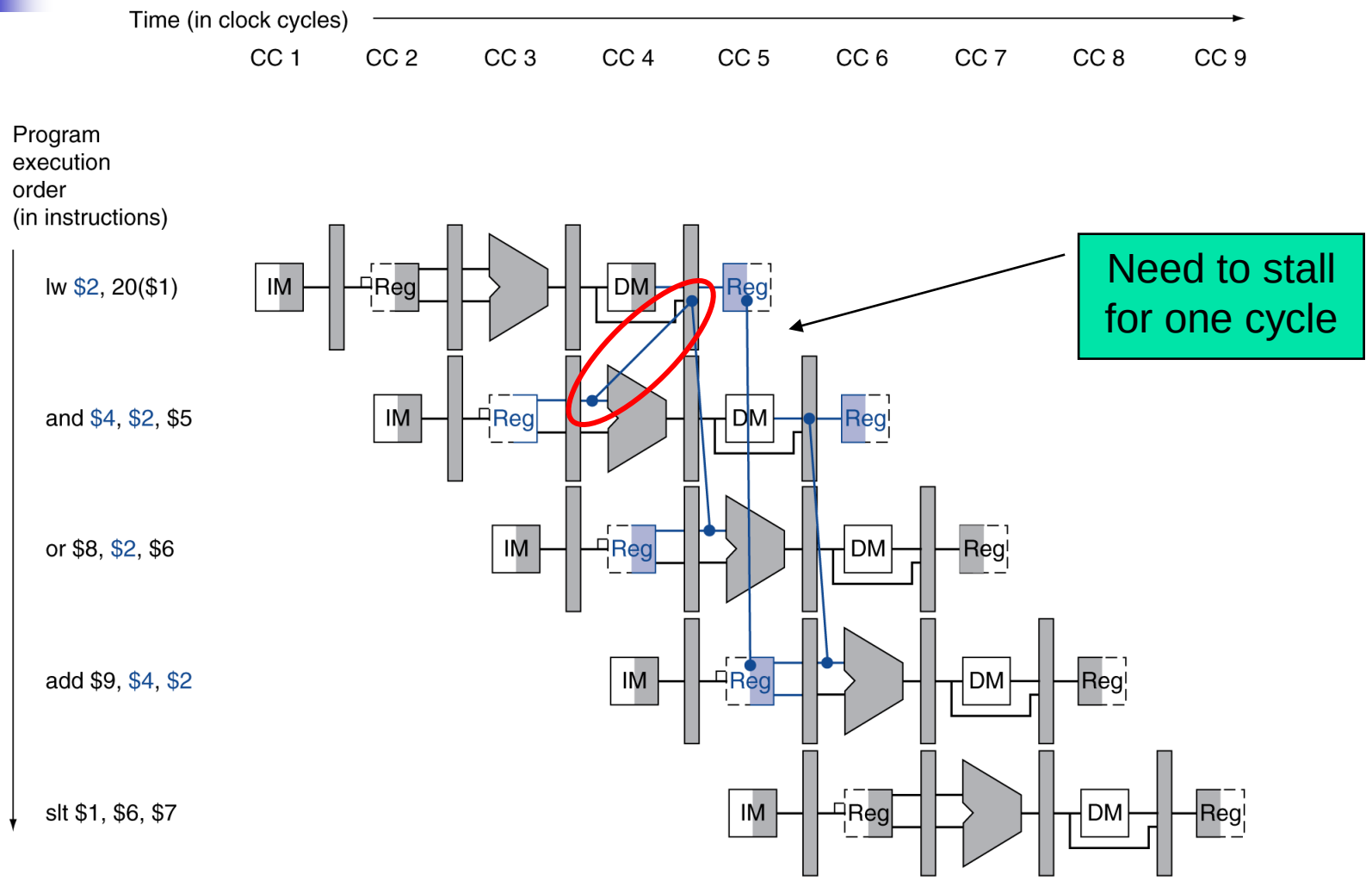


Can't Always Forward

- lw can still cause a hazard (*load-use data hazard*):
  - if lw is followed by an instruction to read the loaded reg.

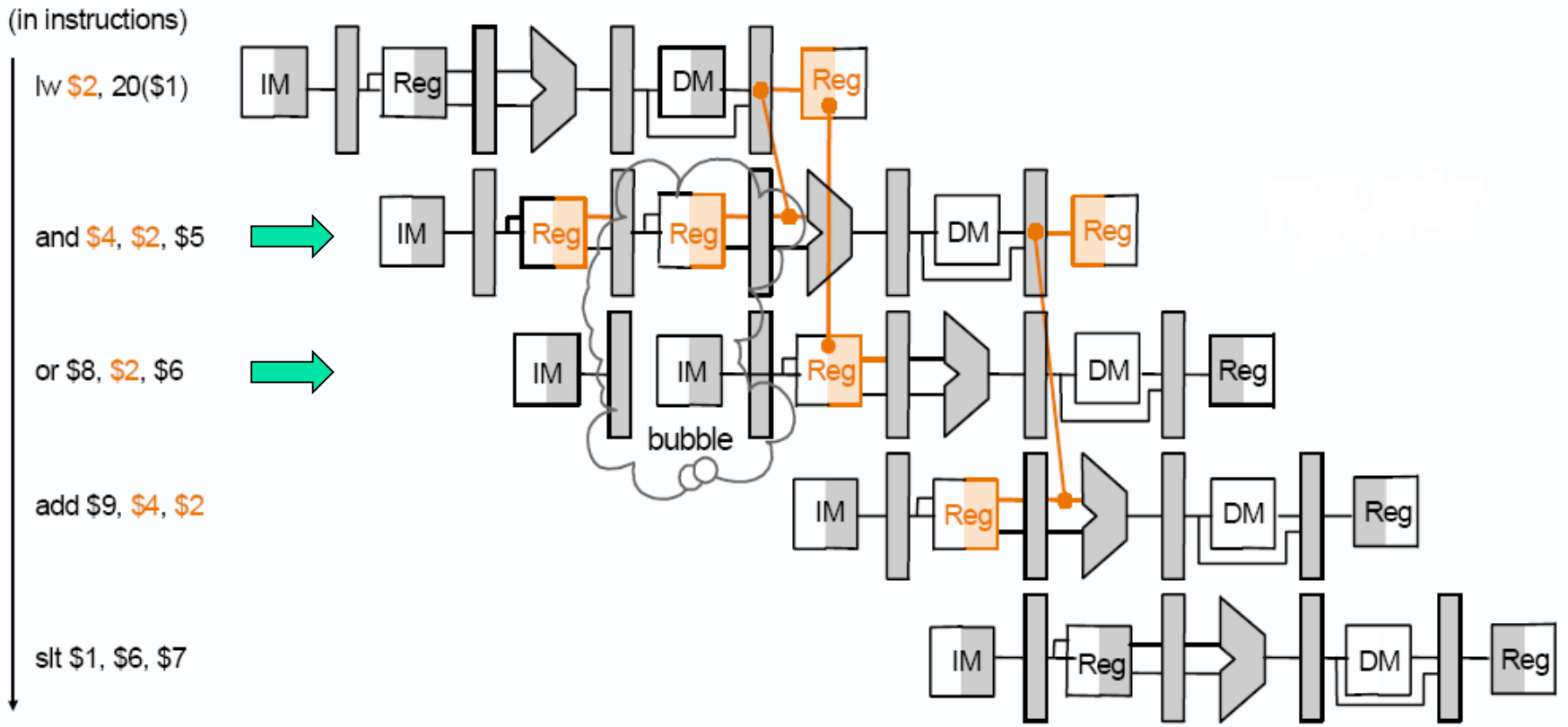


Load-Use Data Hazard



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                  % lw
    ((ID/EX.RegisterRt= IF/ID.RegisterRs) or  % lw.Rt = Reg.Rs
    (ID/EX.RegisterRt= IF/ID.registerRt))     % lw.Rt = Reg.Rt
```

→ Stall the pipeline for one cycle

(*ID/EX.MemRead=1 indicates a load (lw) instruction)*

Handling Stalls

- How to stall?
  - Stall instruction in IF and ID: Don't change PC and IF/ID

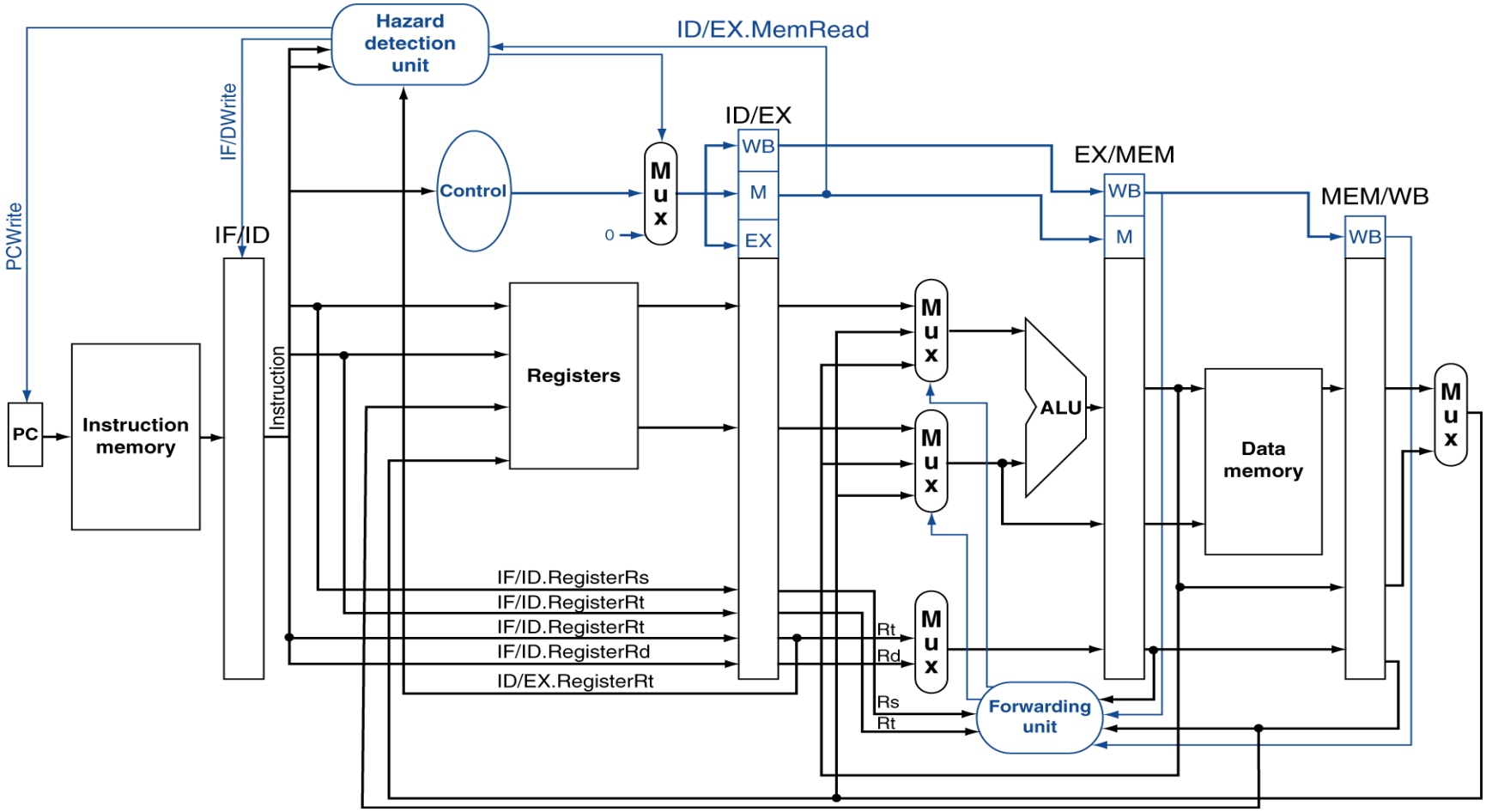
    → the stages re-fetch the next instruction(s)

- What to move into EX: insert an NOP by changing W, MEM, EX control fields of ID/EX pipeline register **to 0**

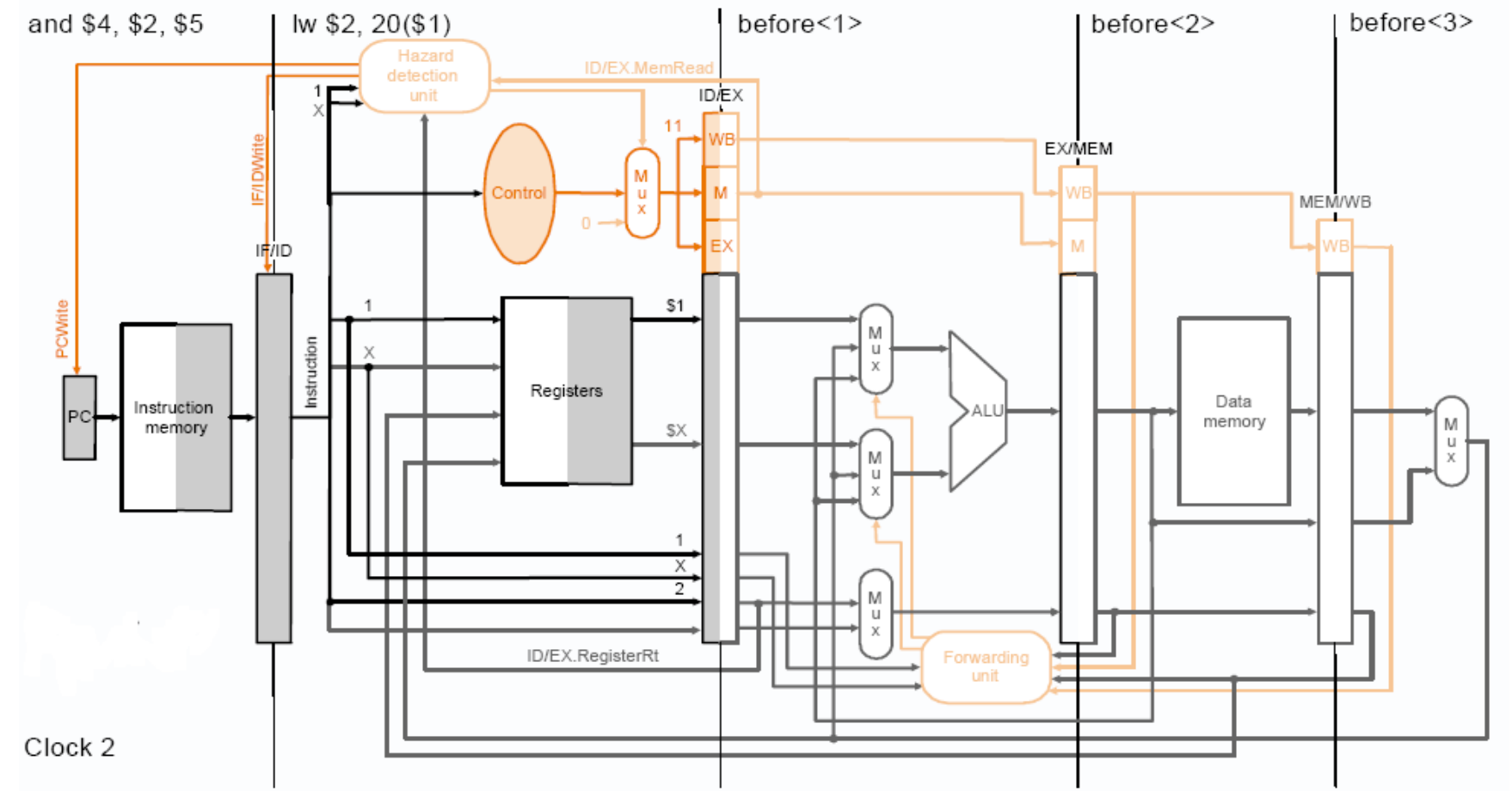
→ As control signals propagate, all control signals to EX, MEM, WB are deasserted and no registers or memories are written

Pipeline with Stalling Unit

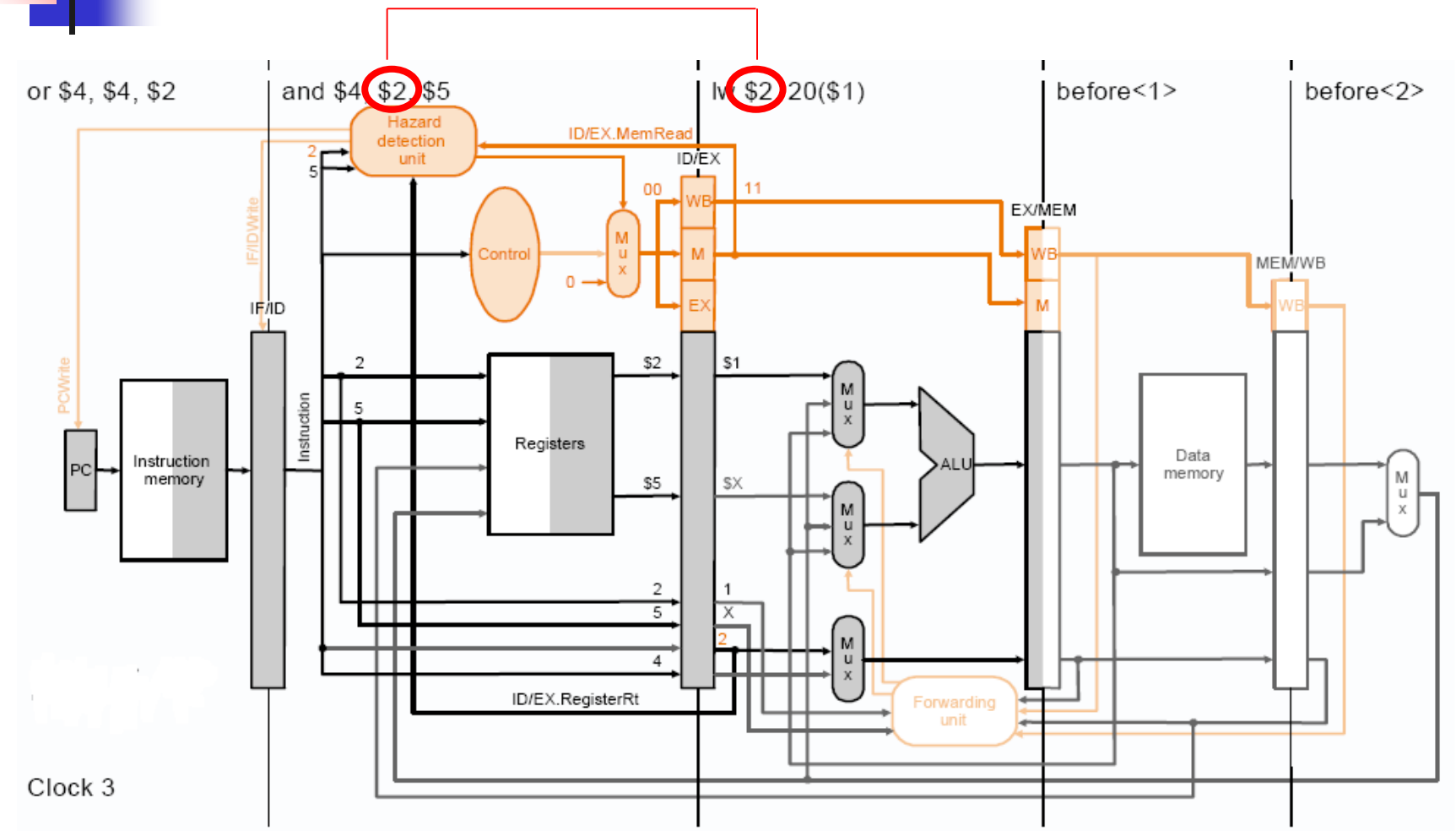
- Forwarding controls ALU inputs; hazard detection controls PC, IF/ID, and control signals



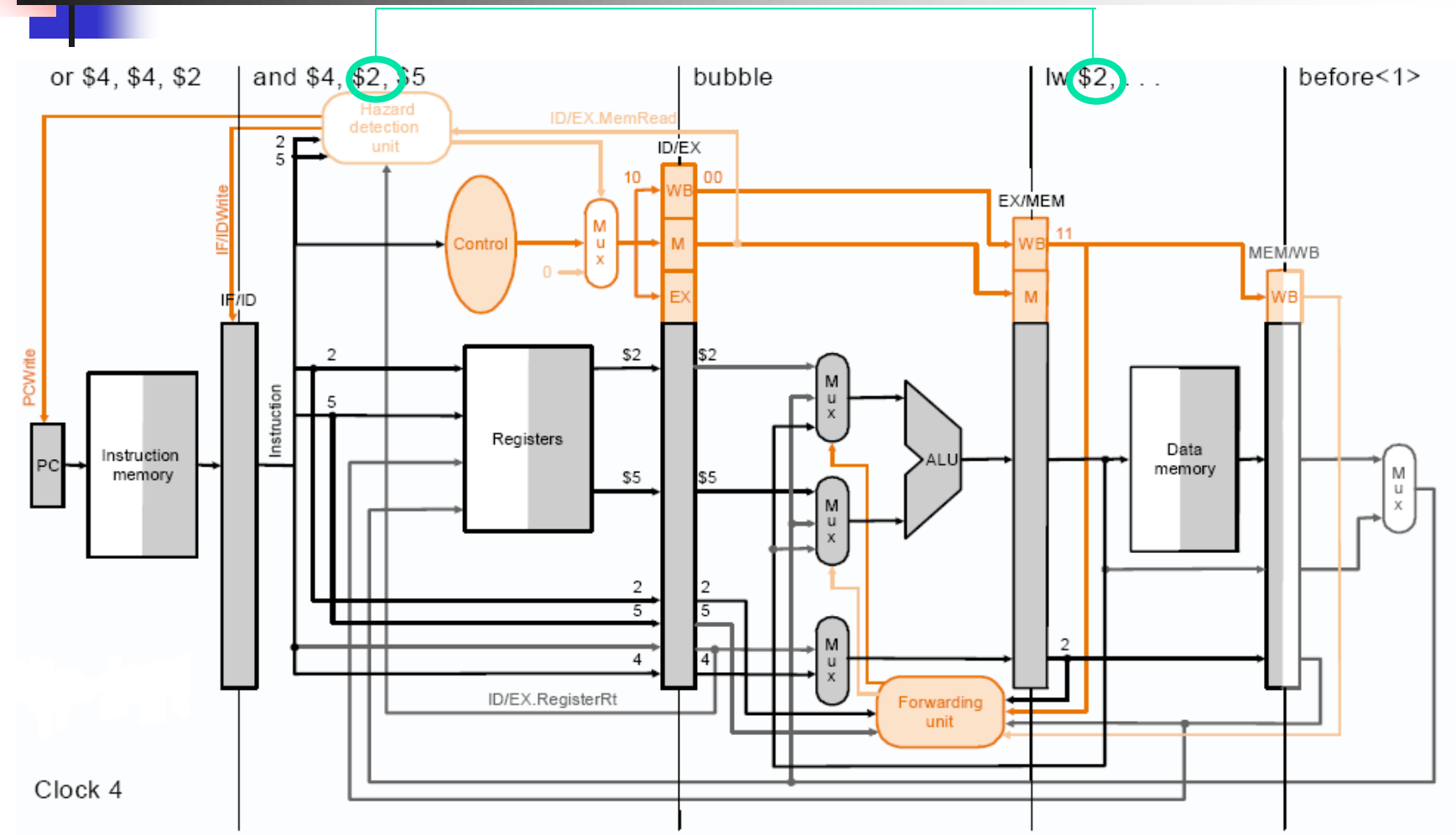
Example 4: Cycle 2



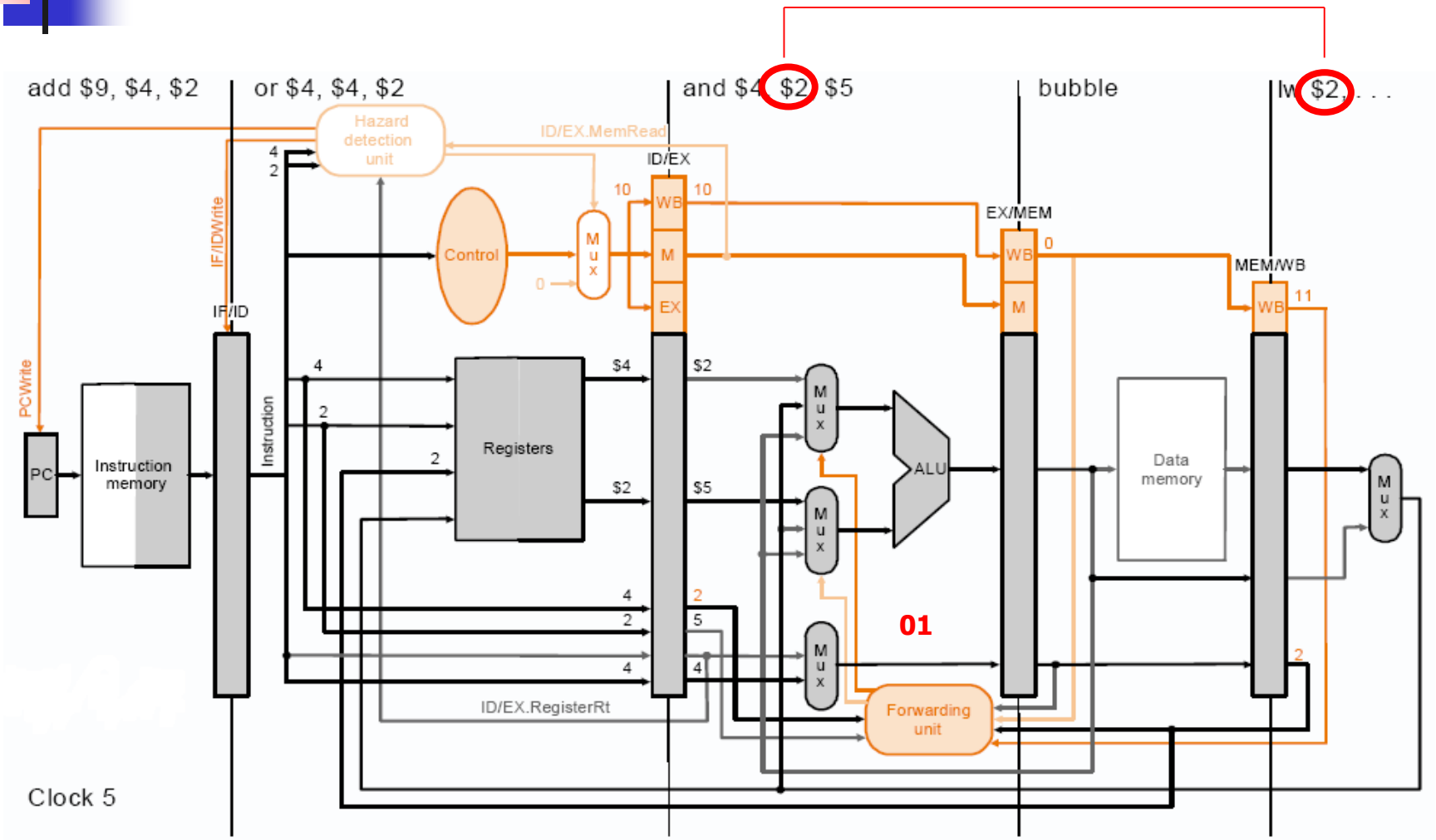
Example 4: Cycle 3



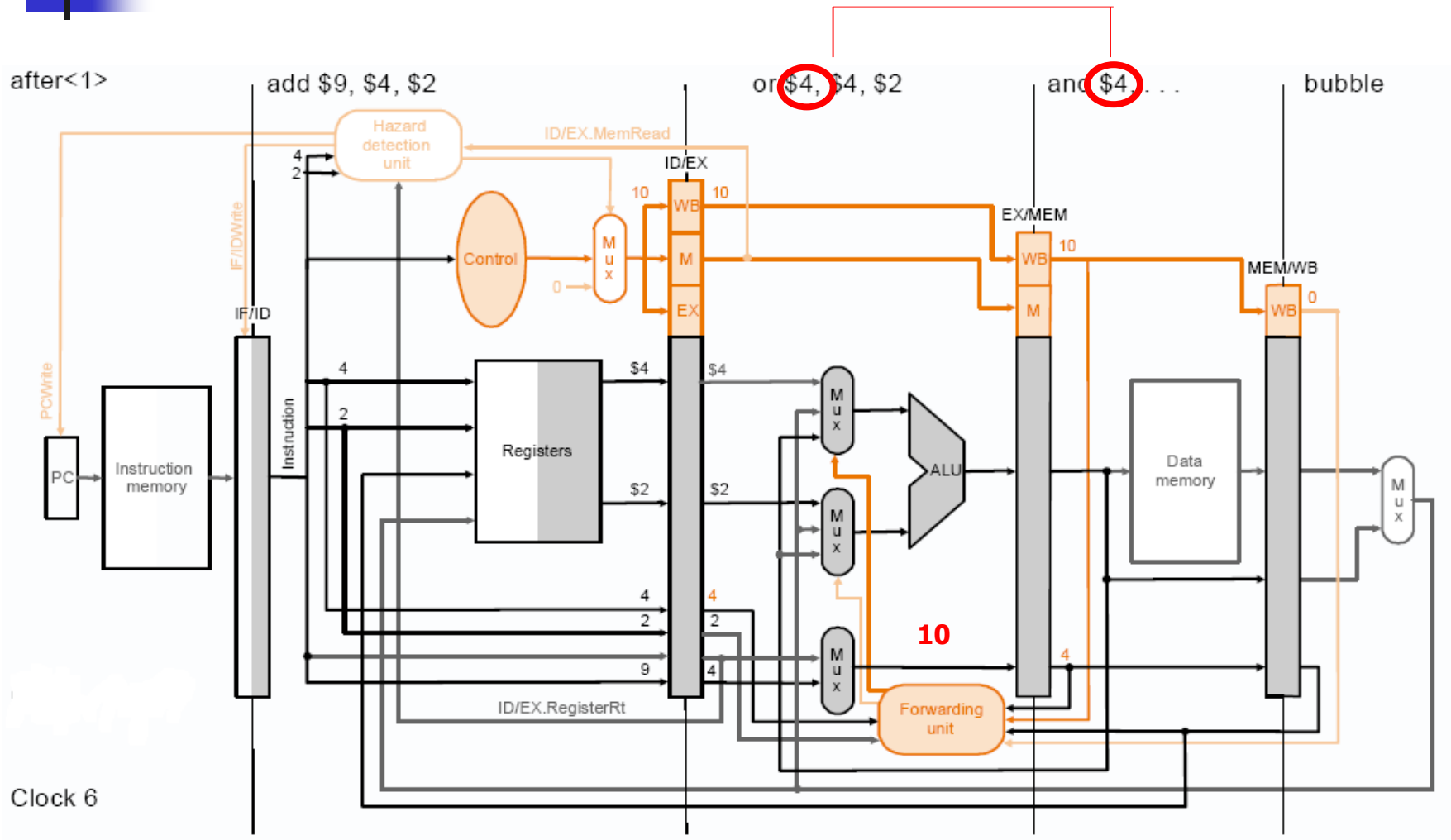
Example 4: Cycle 4



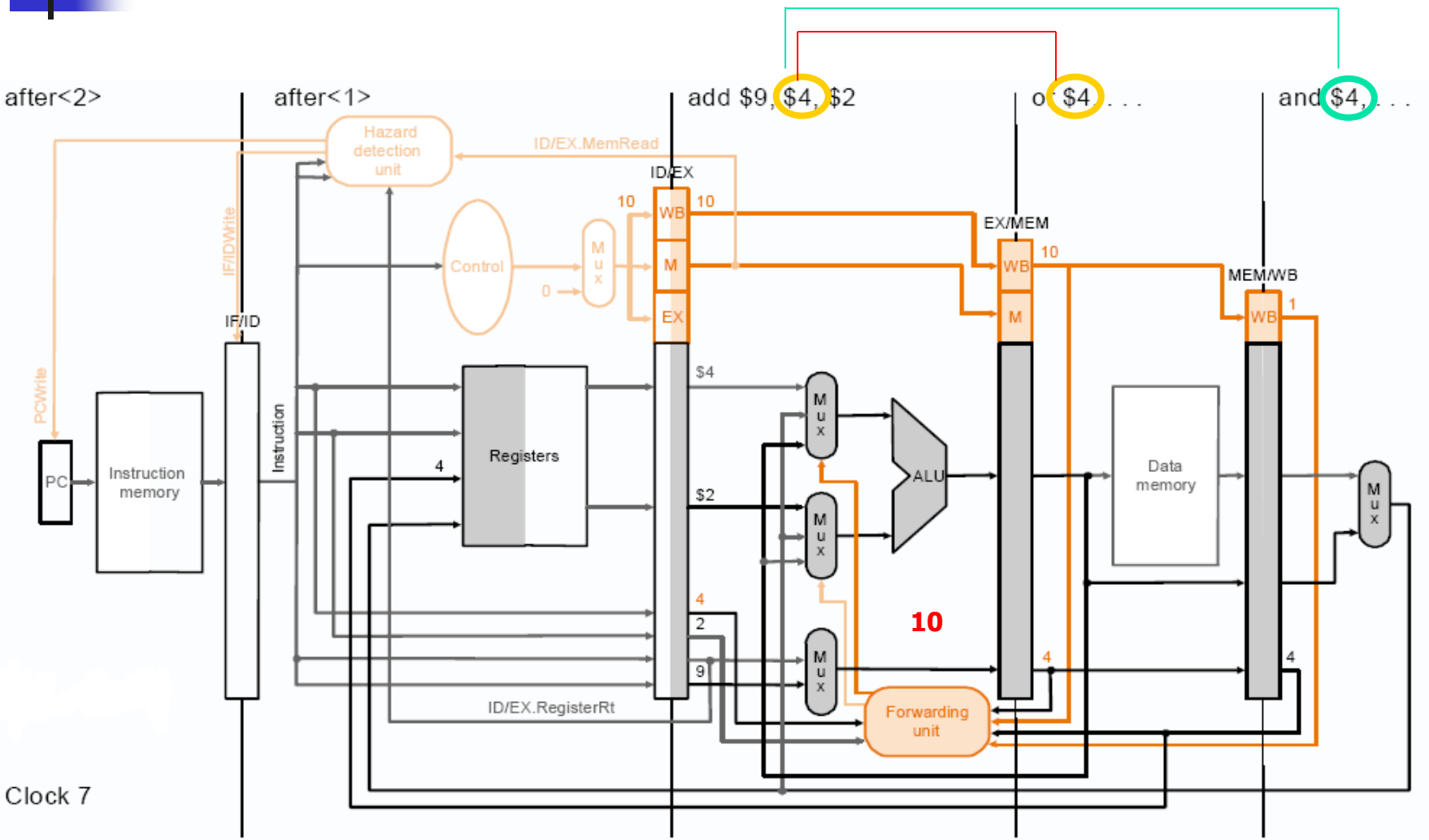
Example 4: Cycle 5



Example 4: Cycle 6

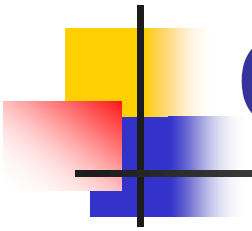


Example 4: Cycle 7



Outline

- 4.5 An Overview of Pipelining
- 4.6 A Pipelined Datapath & Control
- 4.7 Data Hazards and Forwarding v.s. Stalls
- 4.8 Control Hazards
- 4.9 Exceptions



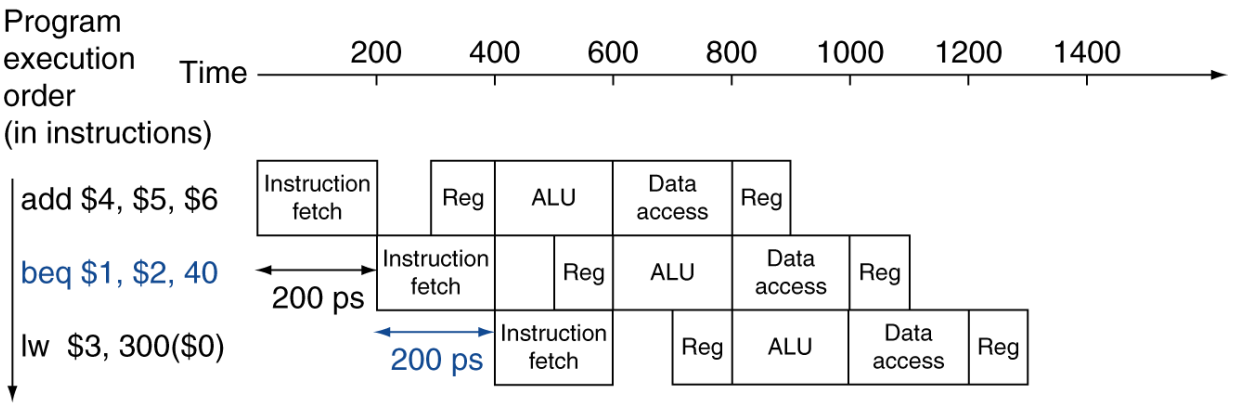
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
- In MIPS pipeline
  - Old way: Decide Branch result in EX stage
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

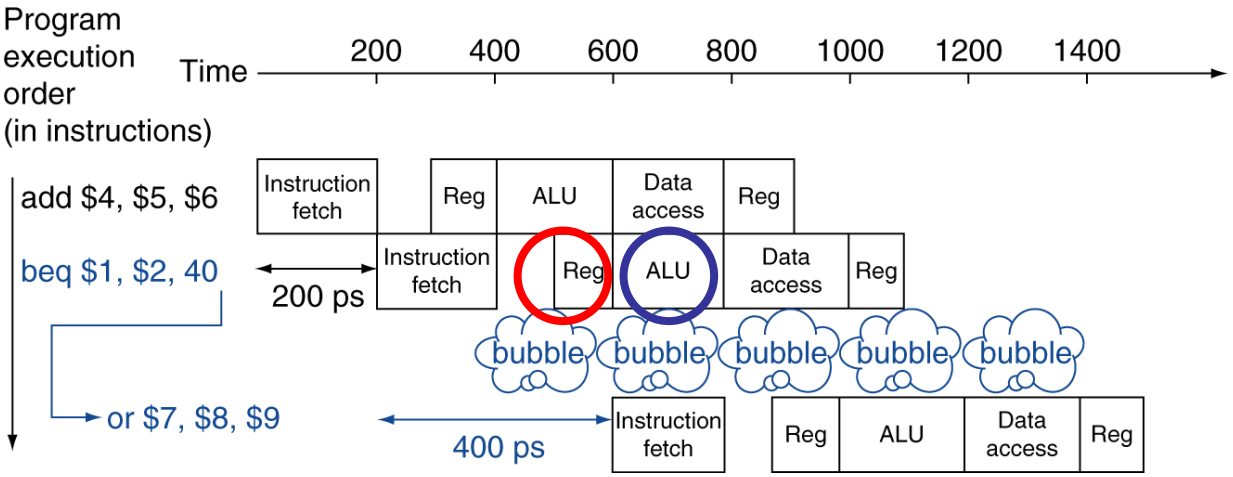
Stall on Branch

Wait until branch outcome determined before fetching next instruction

Branch not taken

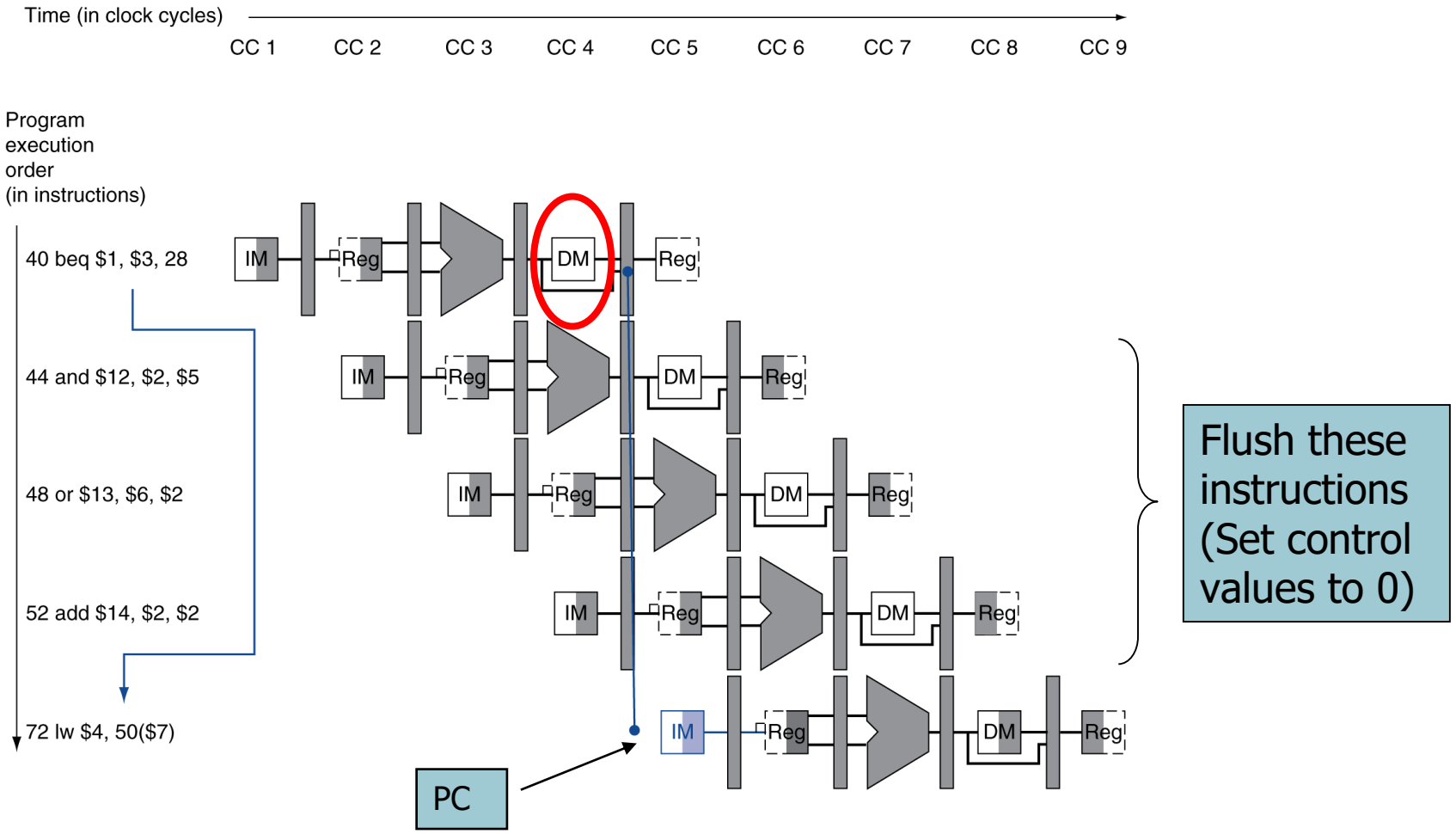


Branch taken



Branch Hazards

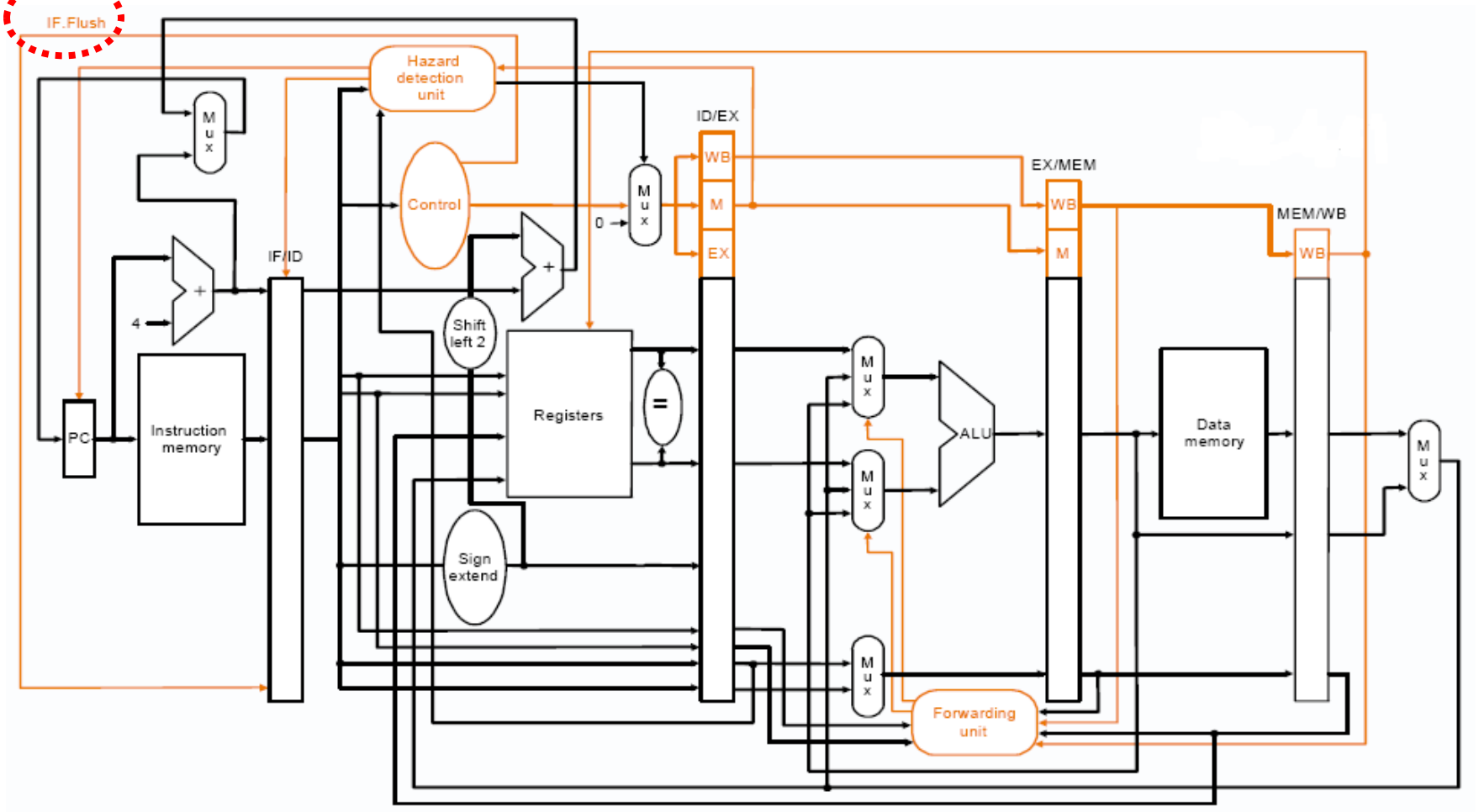
- When decide to branch, other instructions are in pipeline!



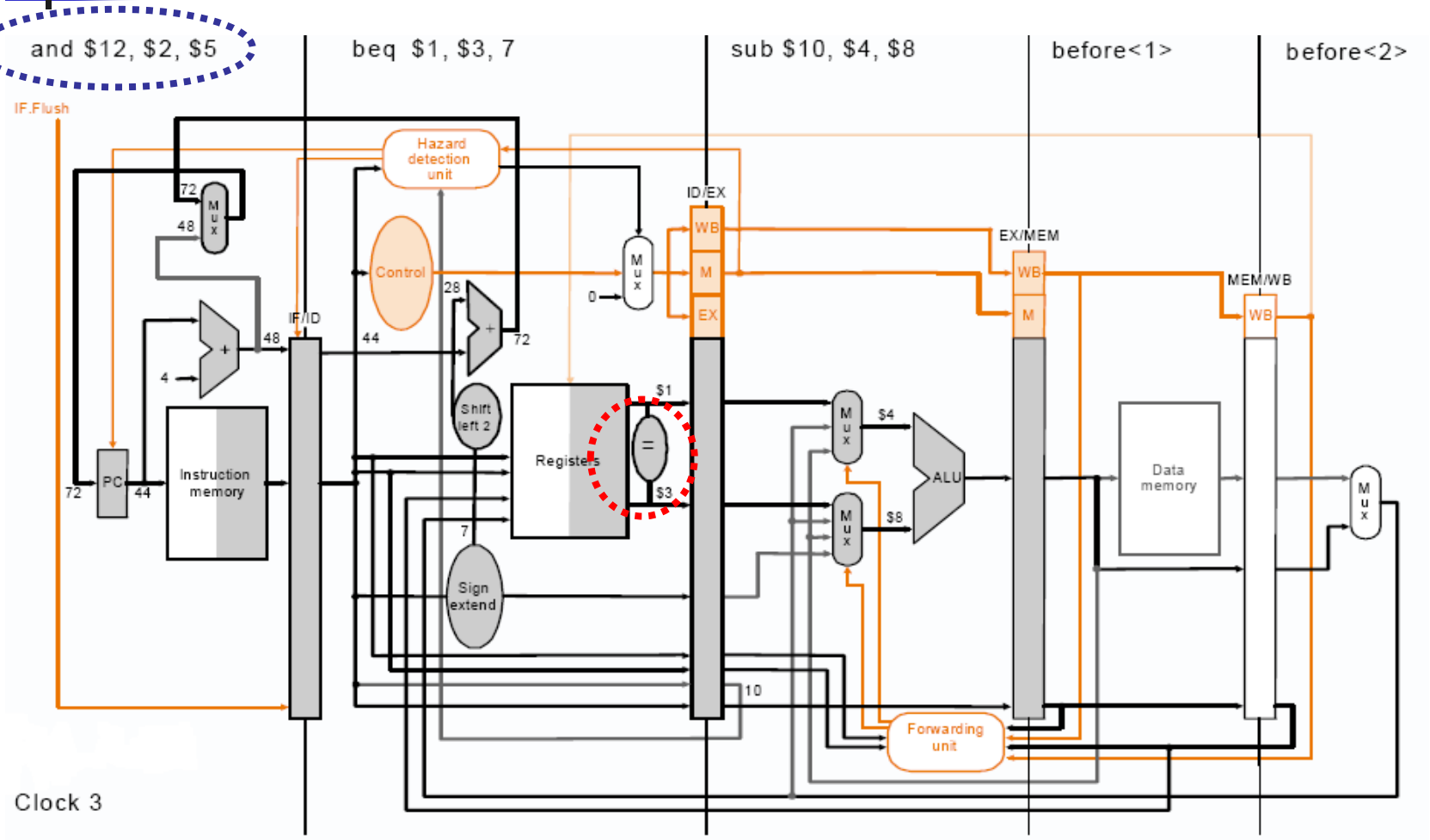
Handling Branch Hazard

- Predict branch always not taken
  - Need to add hardware for flushing inst. if wrong
  - Branch decision made at MEM stage → need to flush inst. in IF, 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 gates) by comparing the two registers read **during ID**
  - Branch decision made at EX stage → ONLY one inst. to flush
  - Add a control signal, IF.Flush, to zero instruction field of IF/ID → making the instruction an NOP
- Dynamic branch prediction (later)
- Compiler rescheduling, delayed branch (read by yourself)

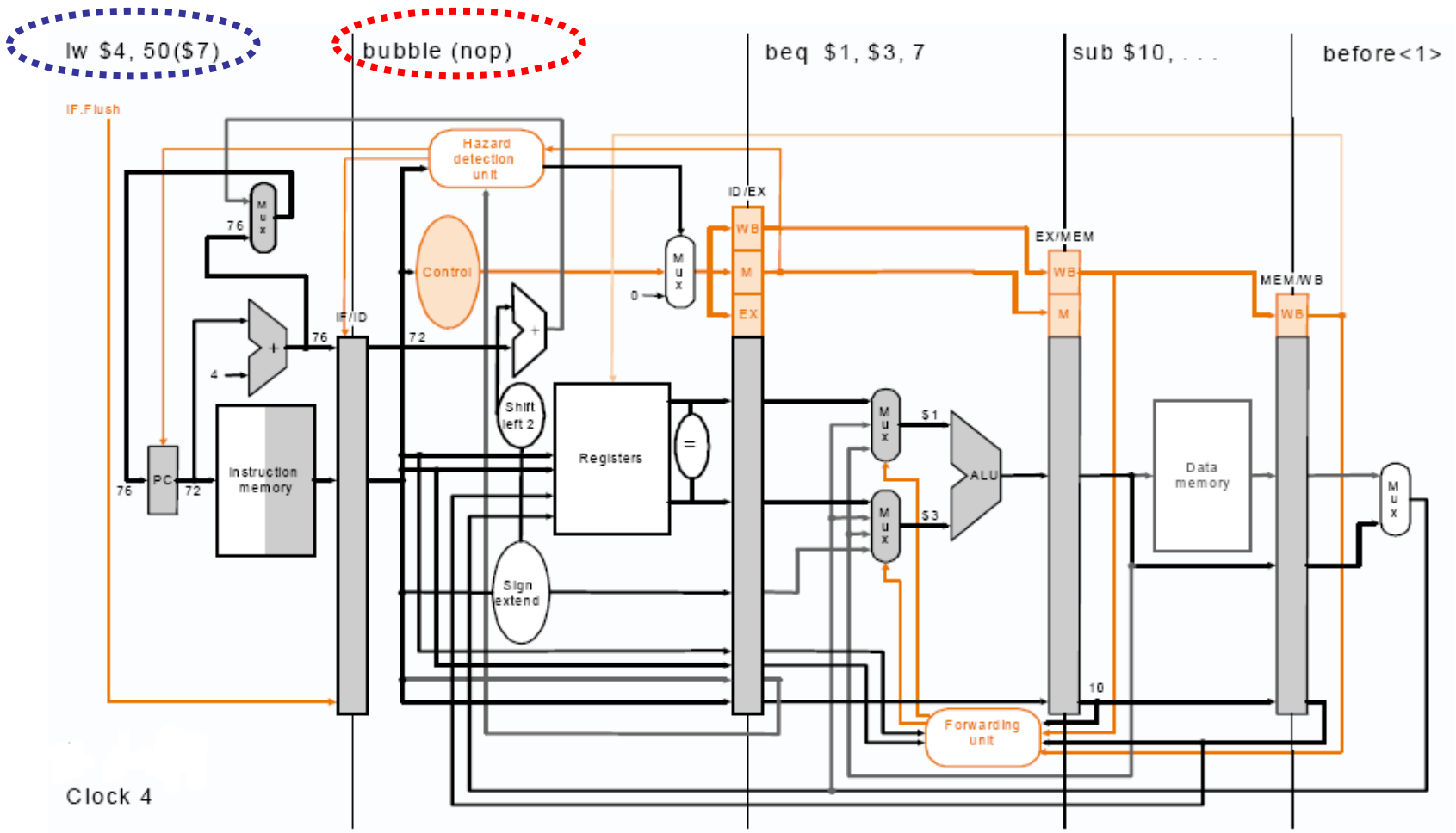
Pipeline with Flushing



Example 5: Cycle 3



Example 5: Cycle 4

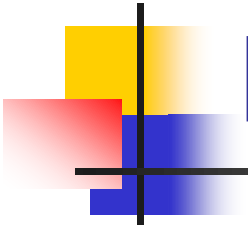


More-Realistic Branch Prediction

- Static branch prediction (compiler-based, off-line)
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
  - Some DSP processors uses RC (Repeat Counter register) to record loop times.

- Dynamic branch prediction (on-line)
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Dynamic Branch Prediction

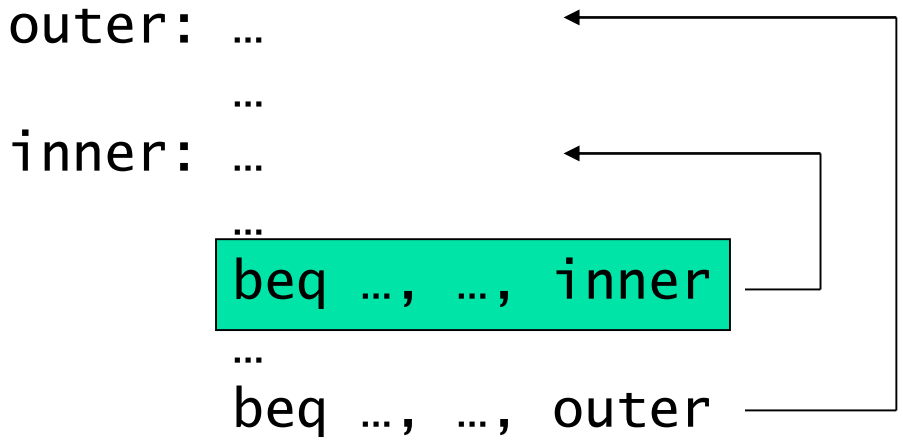


- In deeper and superscalar pipelines, branch penalty is more significant

- Use dynamic prediction
  - 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

1-Bit Predictor: Shortcoming

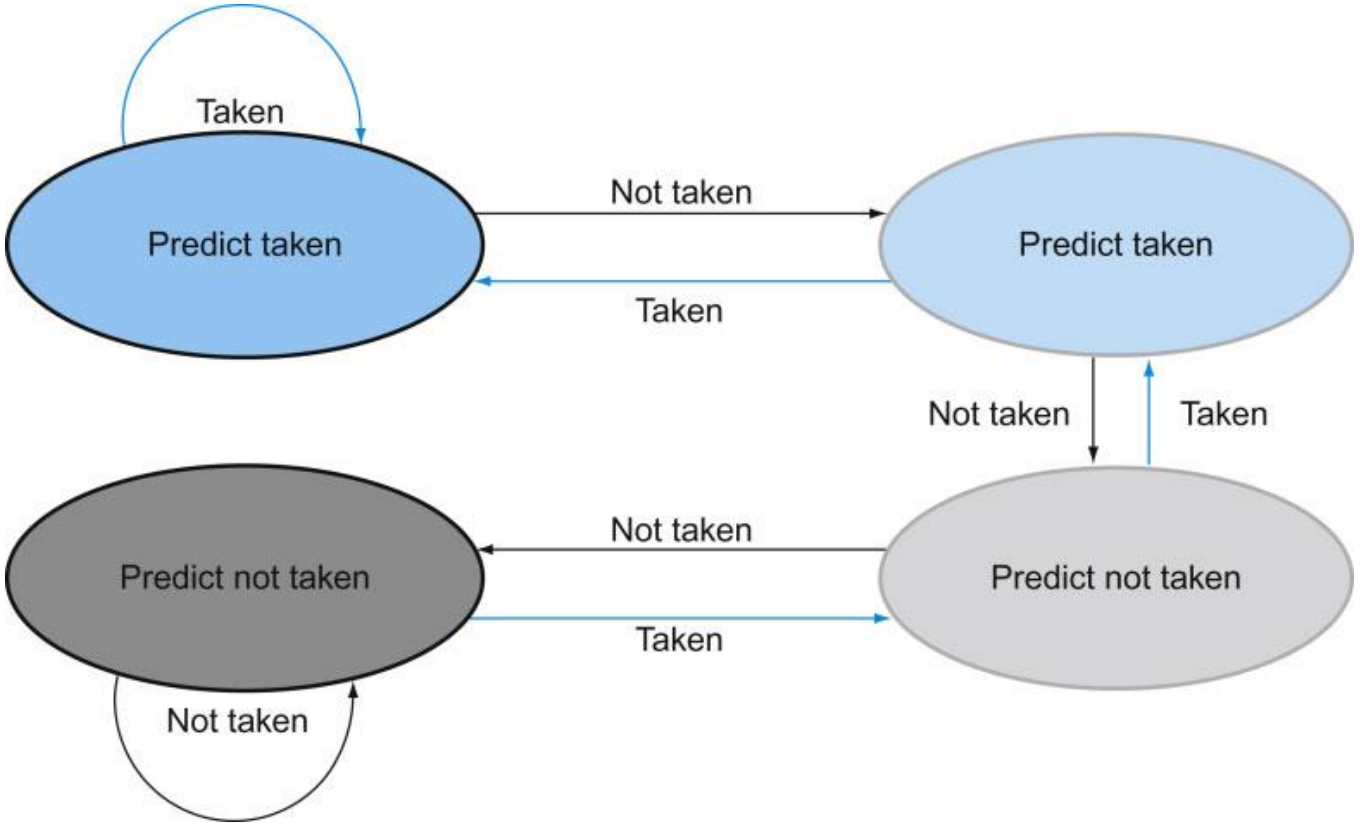
- Inner loop branches mispredicted twice!



  - Mispredict as *not taken* on first iteration of inner loop
  - Mispredict as *taken* on last iteration of inner loop

2-Bit Predictor

- Only change prediction on two successive mispredictions



Summary

- Pipelines pass control information down the pipe just as data moves down pipe
- Forwarding/stalls can be easily handled by local control
- Exceptions stop the pipeline
- MIPS's simple instruction set architecture made pipeline visible and manageable.

Summary

- More performance from deeper pipelines, parallelism (Chap. 4.10, page 332) → **may explore in your final project reports!**
  - Branch prediction
  - Multiple issues
  - VLIW (Very long length Instruction Word)
  - Superscalar
  - Dynamic scheduling
  - Out-of-order execution
  - Speculation
  - Reorder buffer
  - Register renaming
- **For more details, please check the classical textbook "Computer Architecture – A Quantitative Approach," by Hennessy and Patterson, 4th ed., 2007.**