Chapter4- III

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

臺大電機系

吳安宇教授

2025/04/7 v1.2

Outline

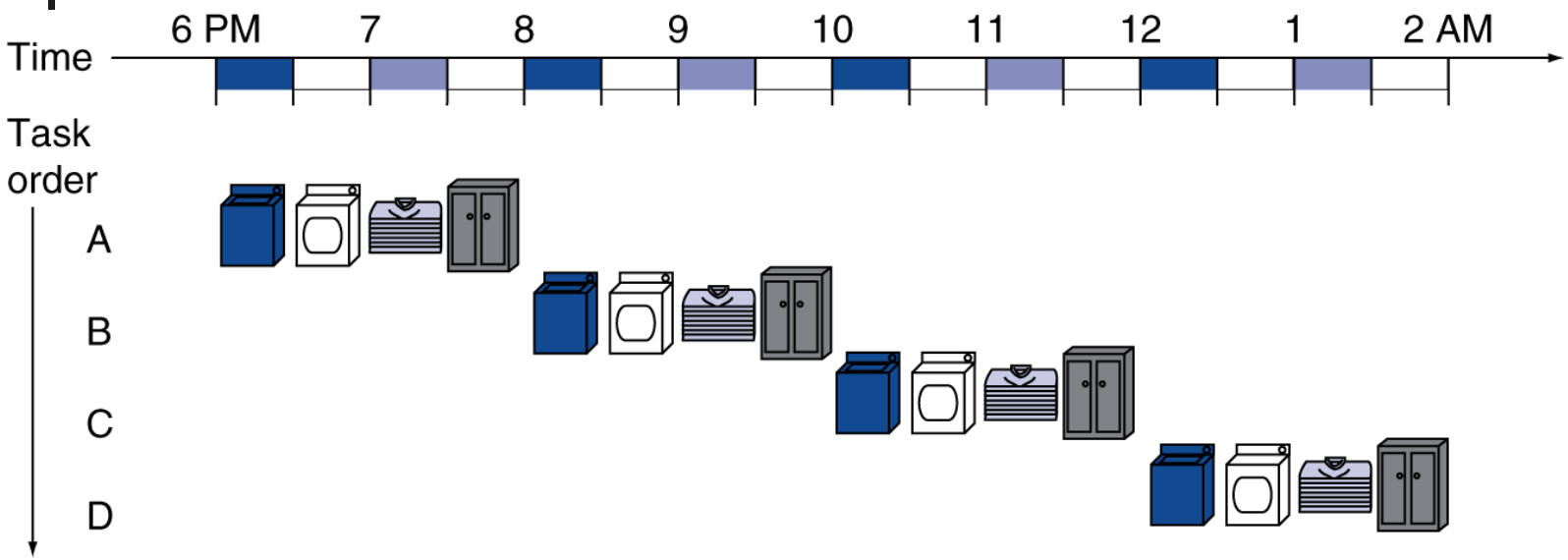
- 4.5 An Overview of Pipelining
- 4.6 A Pipelined Datapath & Control
- 4.7 Data Hazards and Forwarding v.s. Stalls
- 4.8 Control Hazards
- 4.9 Exceptions

Pipelining is Natural!

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

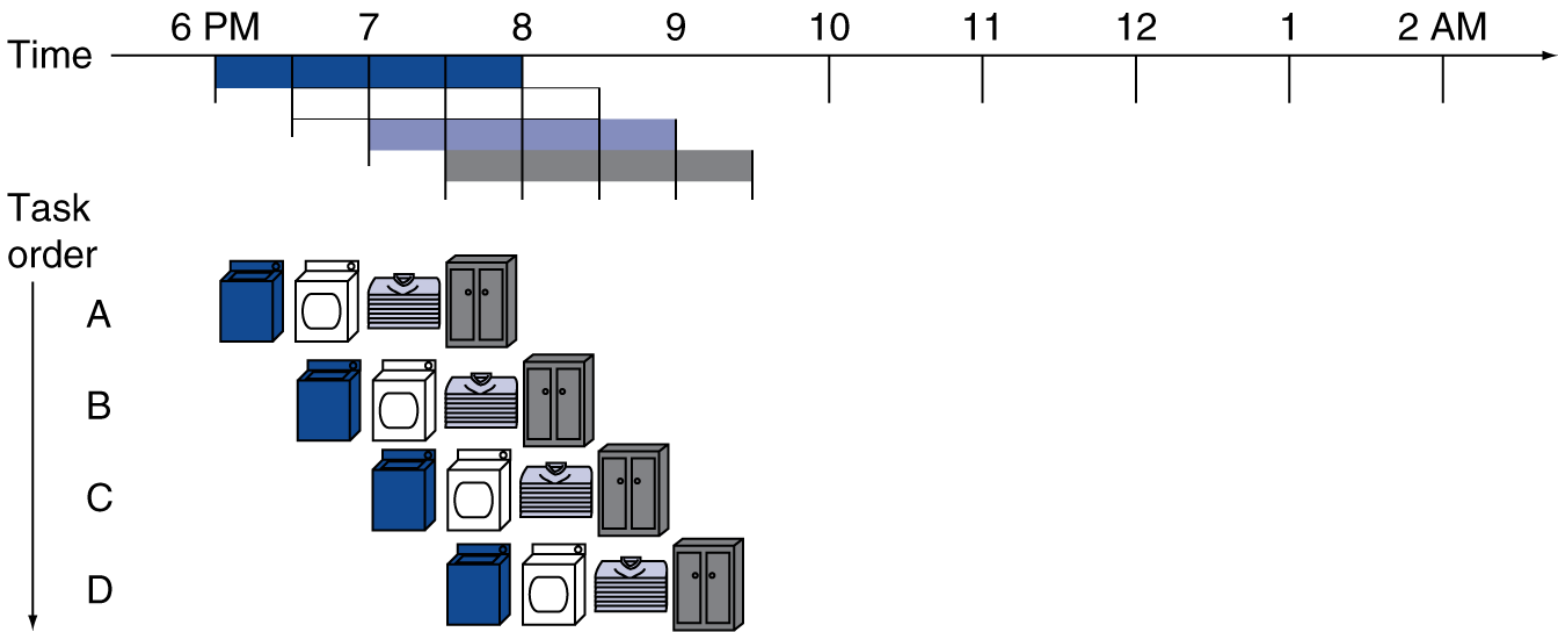
- The *washer, dryer, folder, storer* each take 30 minutes for their task

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

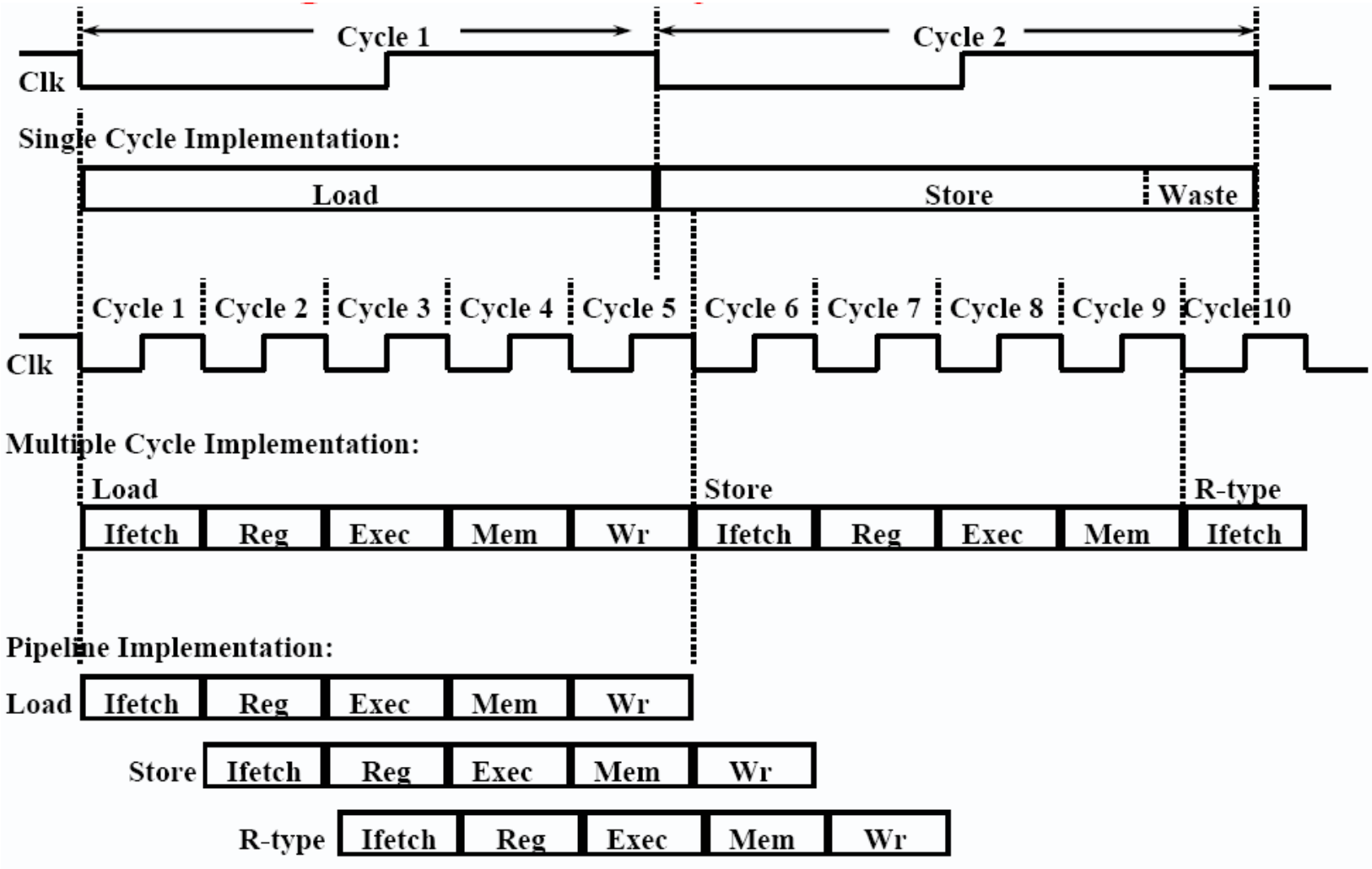


- Pipelined laundry takes 3.5 hours for 4 loads
- Throughput = 4/3.5 = 1.15 load/hour (latency = 2 hours)

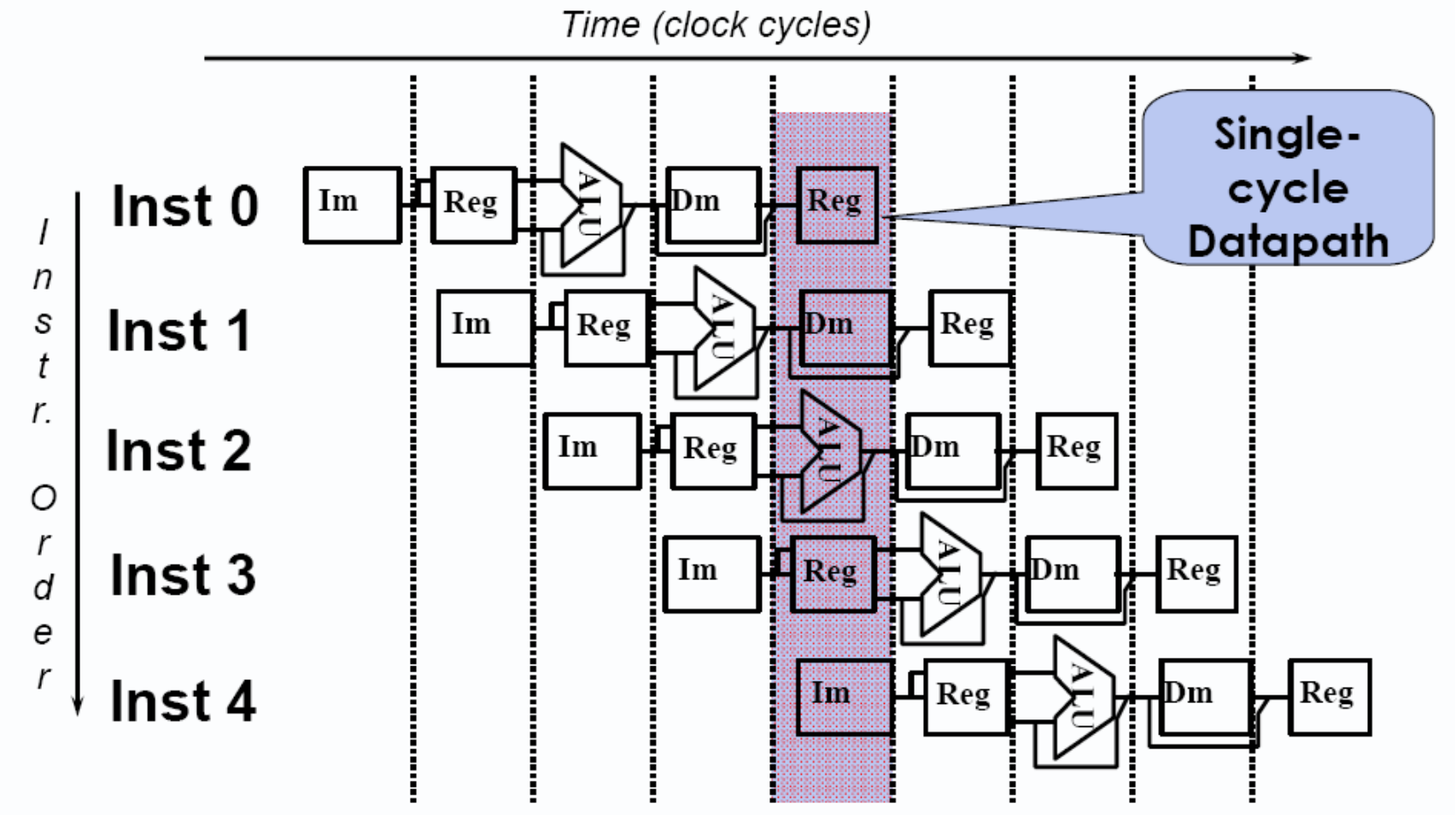
Review of Multi-cycle Execution Steps

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

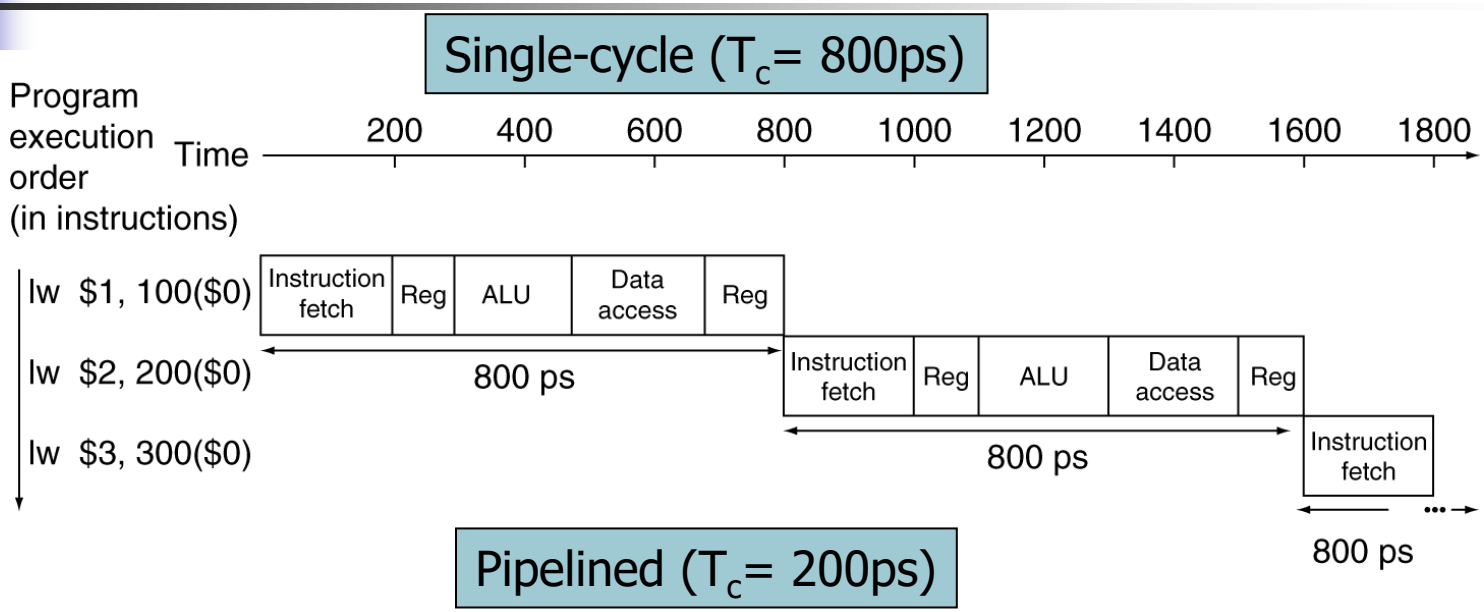
Single-, Multi-cycle, vs. Pipeline

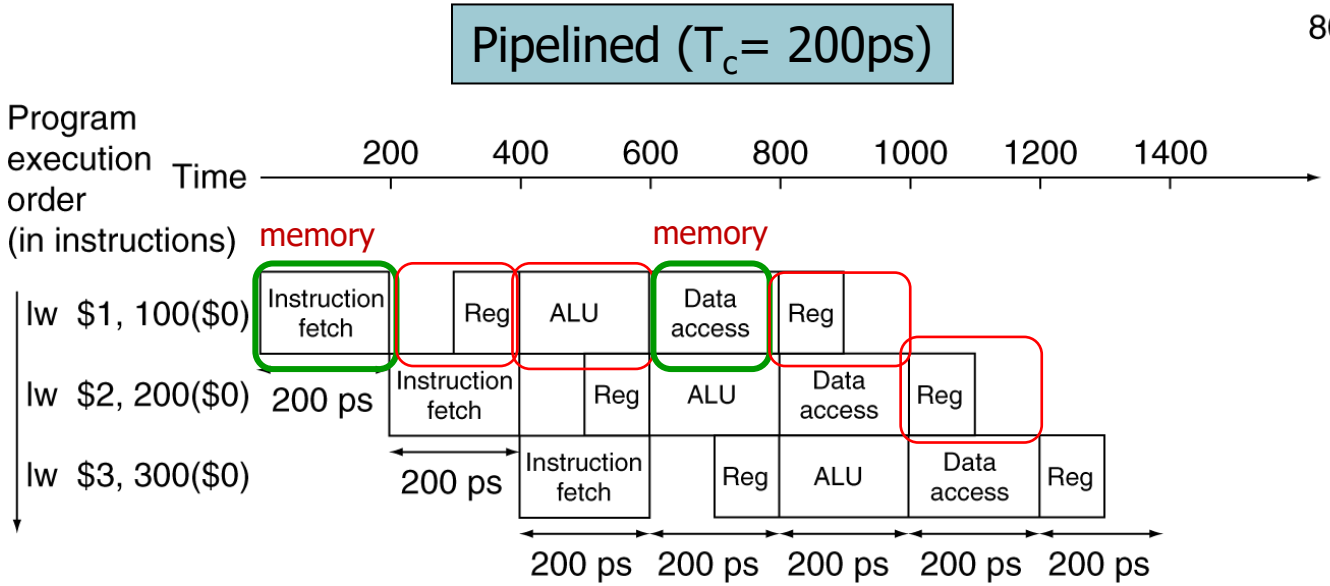


Why Pipeline? Because the Resources Are There!



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

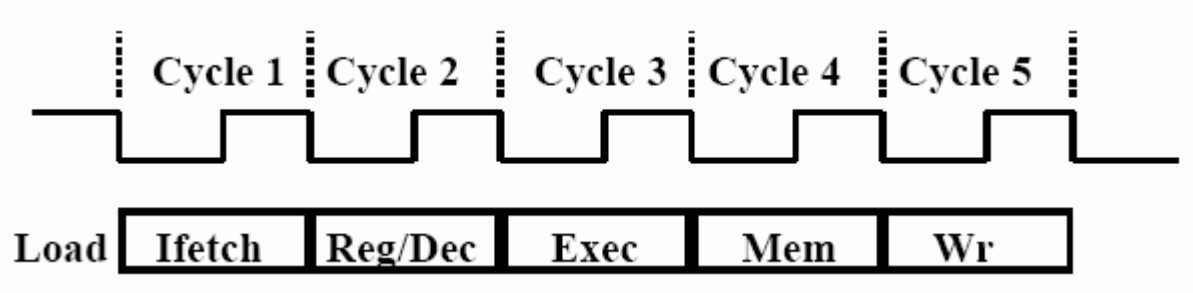
- 4.5 An Overview of Pipelining
- 4.6 A Pipelined Datapath & Control
- 4.7 Data Hazards and Forwarding v.s. Stalls
- 4.8 Control Hazards
- 4.9 Exceptions

Designing a Pipelined Processor

- Examine the datapath and control diagram
  - Starting with single or multi-cycle datapath??
  - Single or multi-cycle control?
- Partition datapath into five stages:
  - IF (instruction fetch), ID (instruction decode and register file read), EX (execution or address calculation), MEM (data memory access), WB (write back)

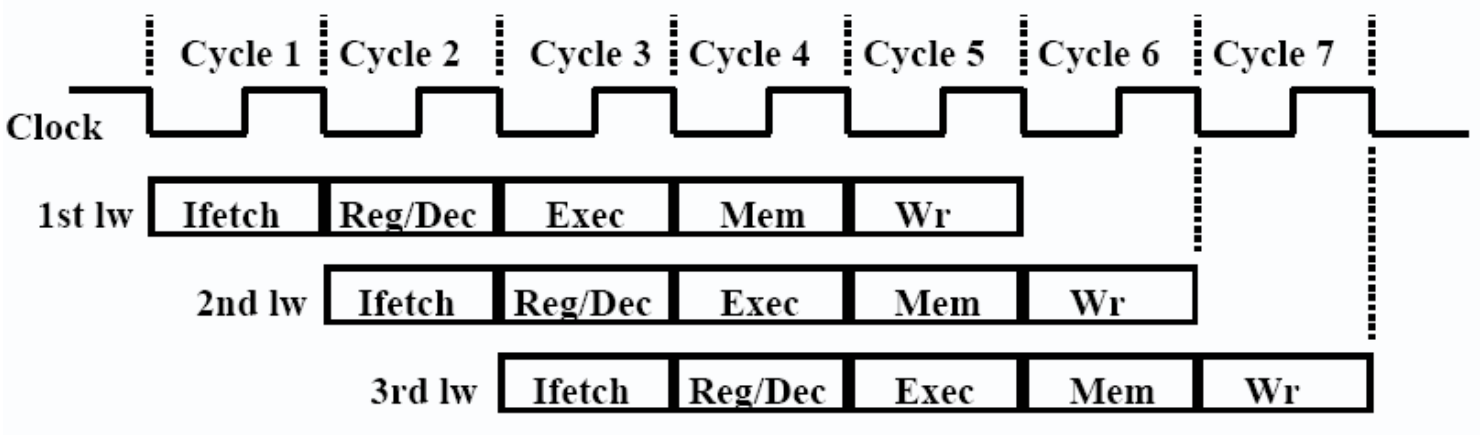
- Associate resources with states
- Ensure that flows do not conflict, or figure out how to resolve it
- Assert control in appropriate stages

Consider load word (lw)



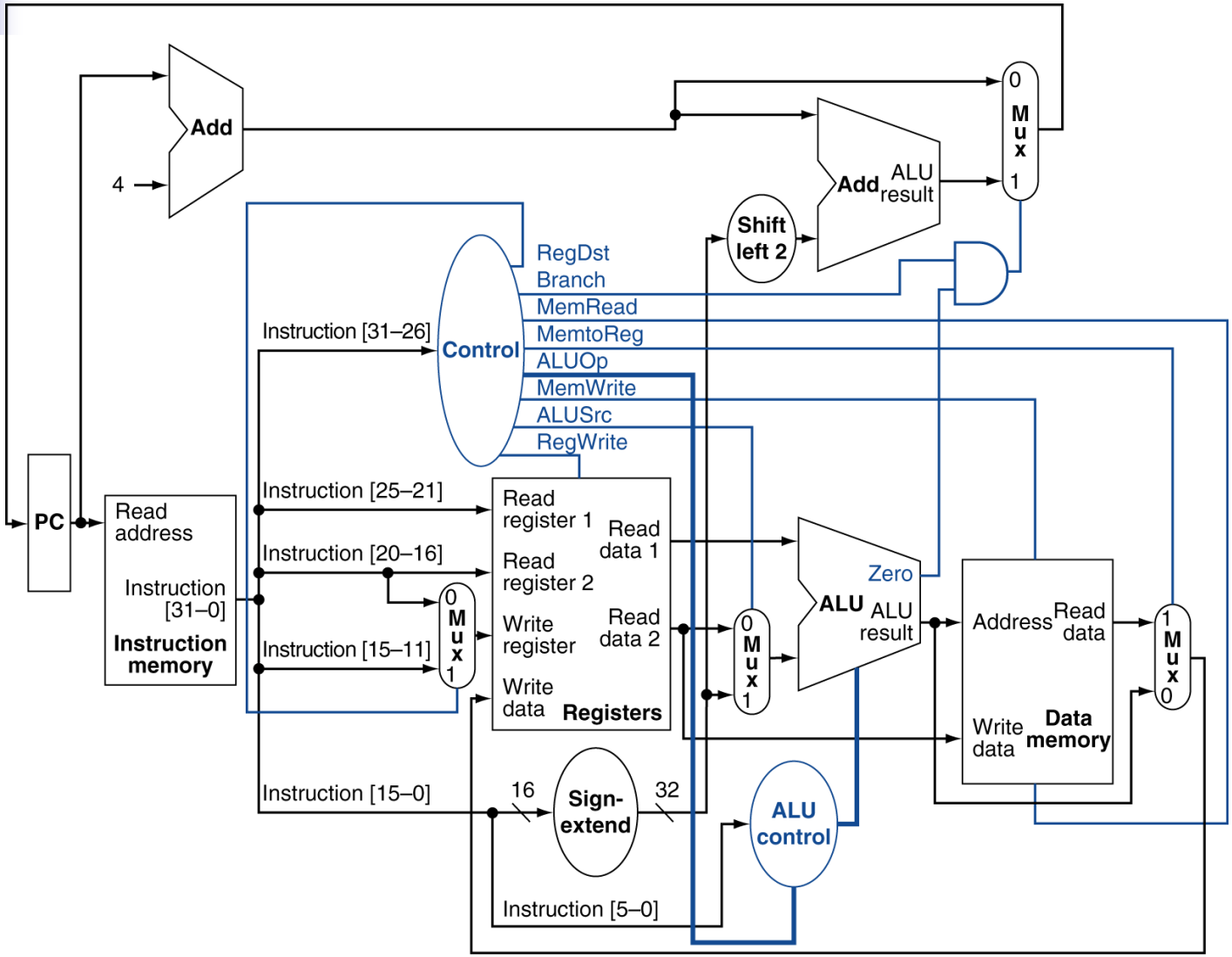
- IF: Instruction Fetch
  - Fetch the instruction from the Instruction Memory (IM)
- ID: Instruction Decode
  - Registers fetch and instruction decode (ID)
- EX: Calculate the memory address
- MEM: Read the data from the Data Memory
- WB: Write the data back to the register file

Pipelining load word

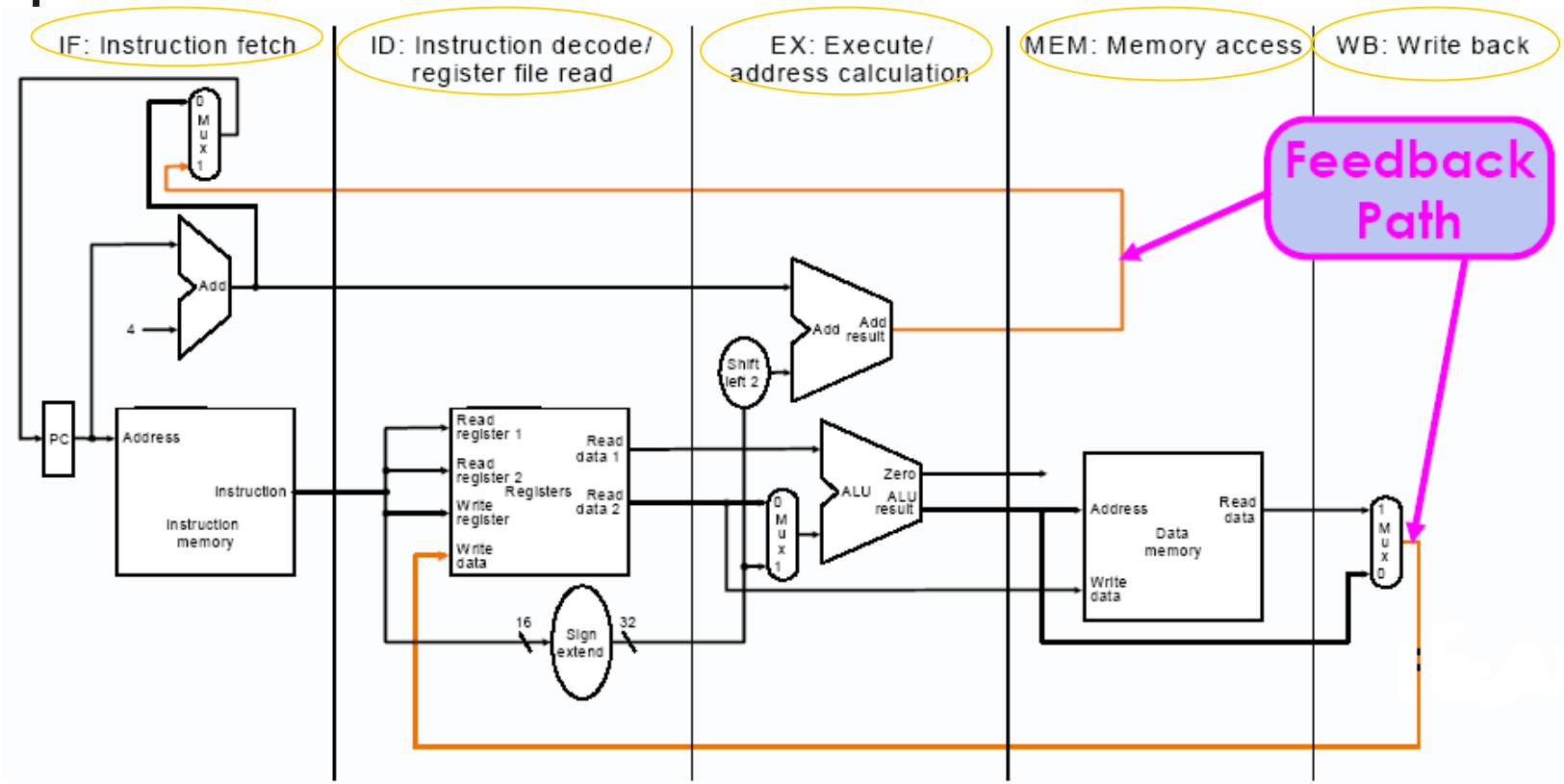


- 5 functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (busA and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the MEM stage
  - Register File's Write port (busW) for the WB stage

Start with Single-cycle MIPS Design

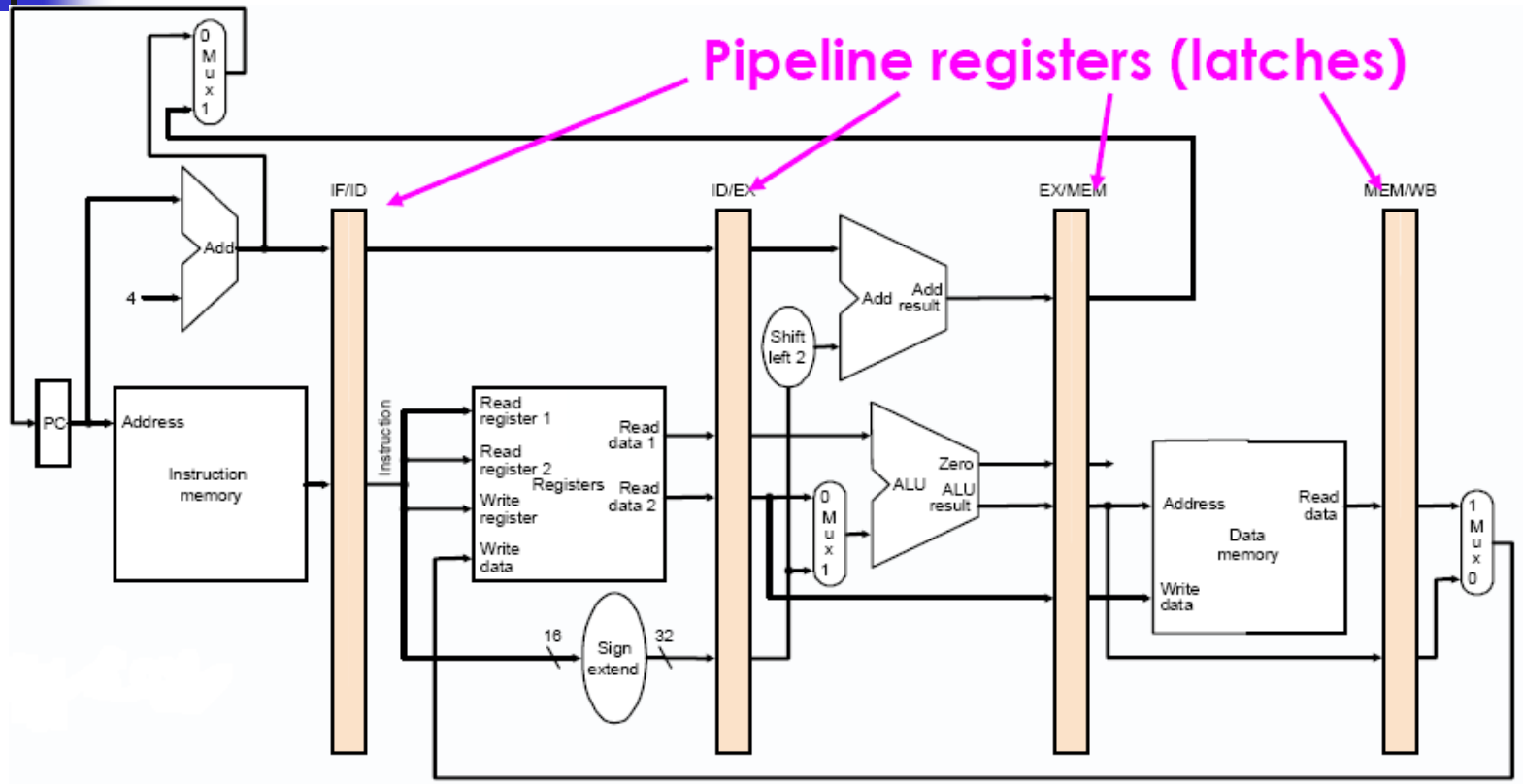


Split Single-cycle Datapath



What to add to split the datapath into stages?

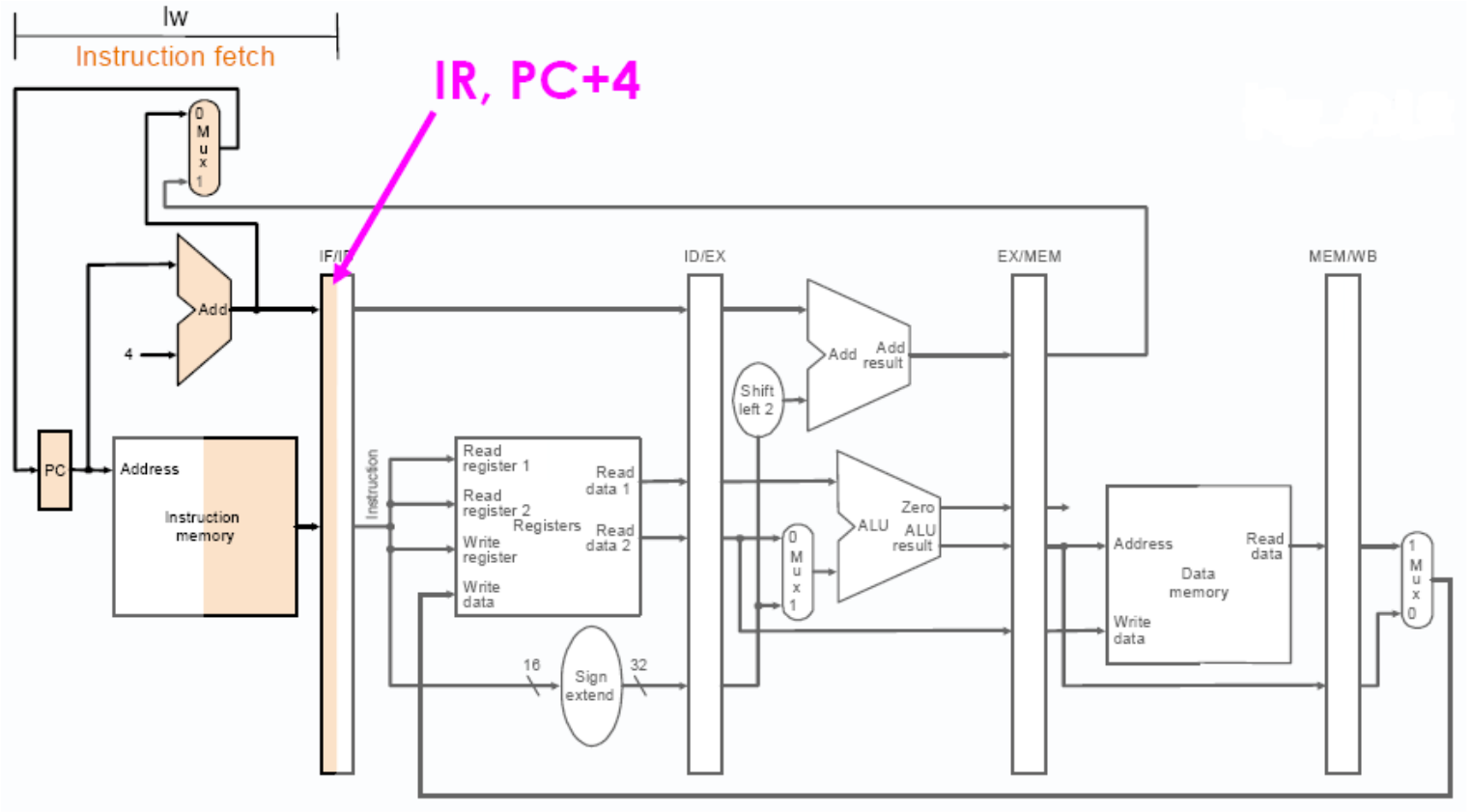
Add Pipeline Registers (stages)



- Use registers between stages to carry data and control

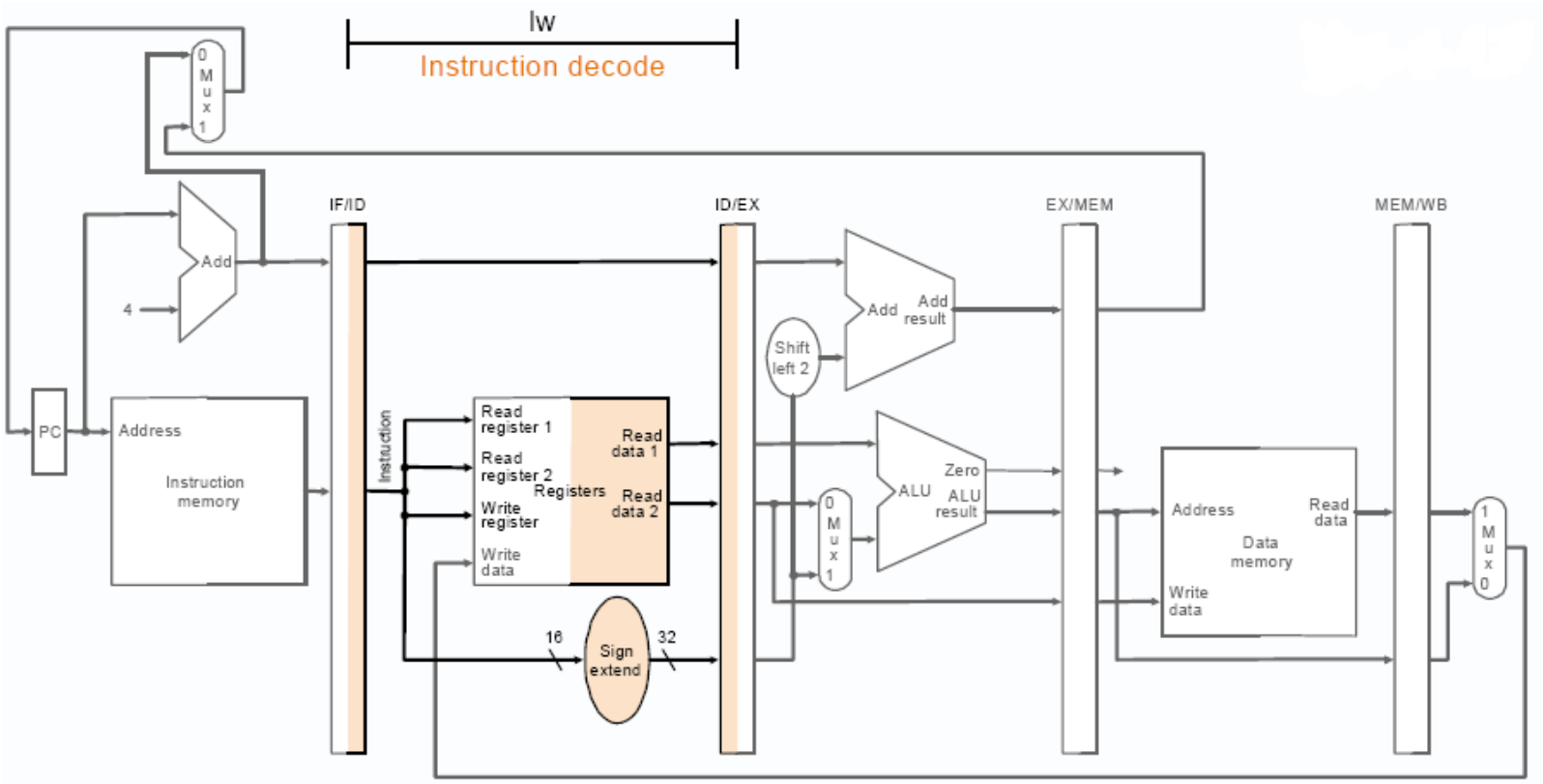
IF Stage of load word

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



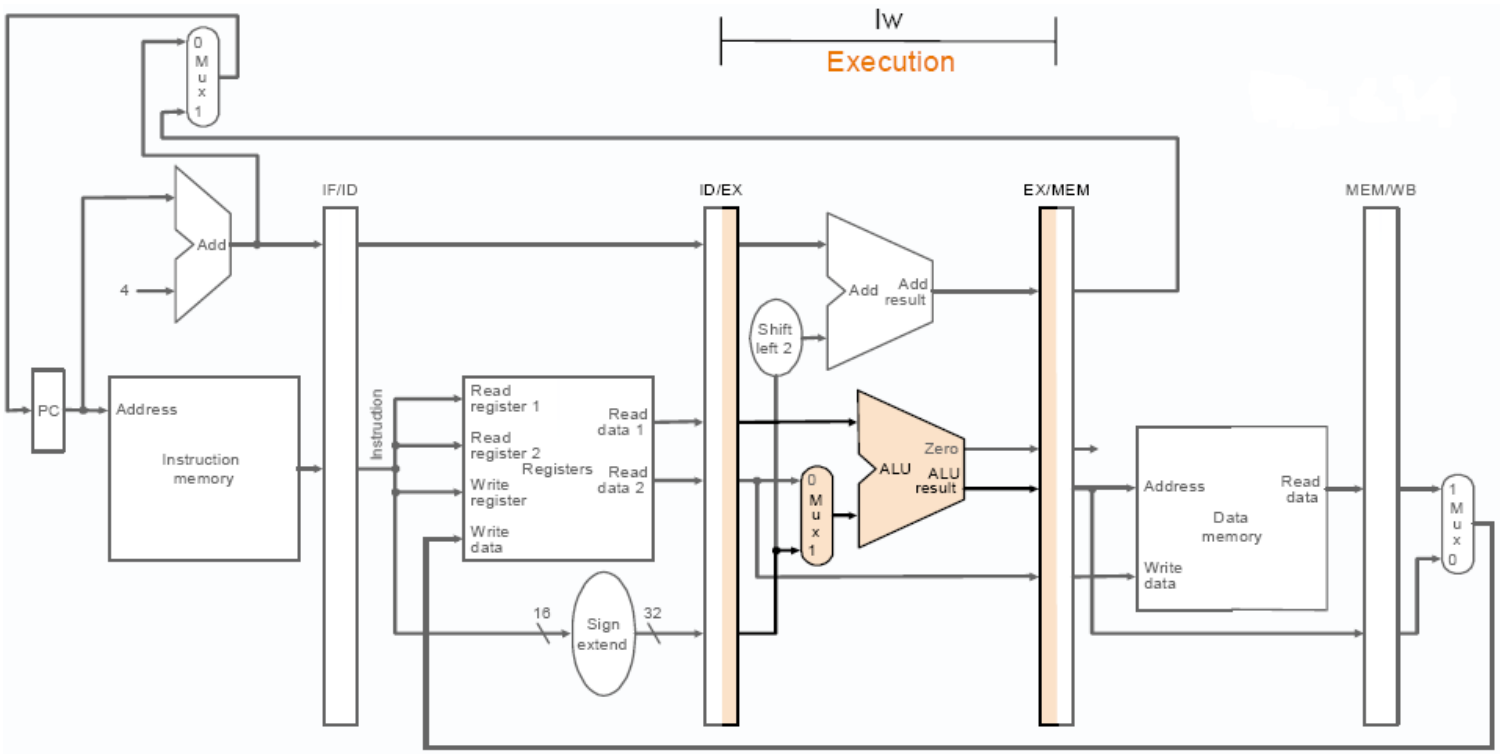
ID Stage of load word

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



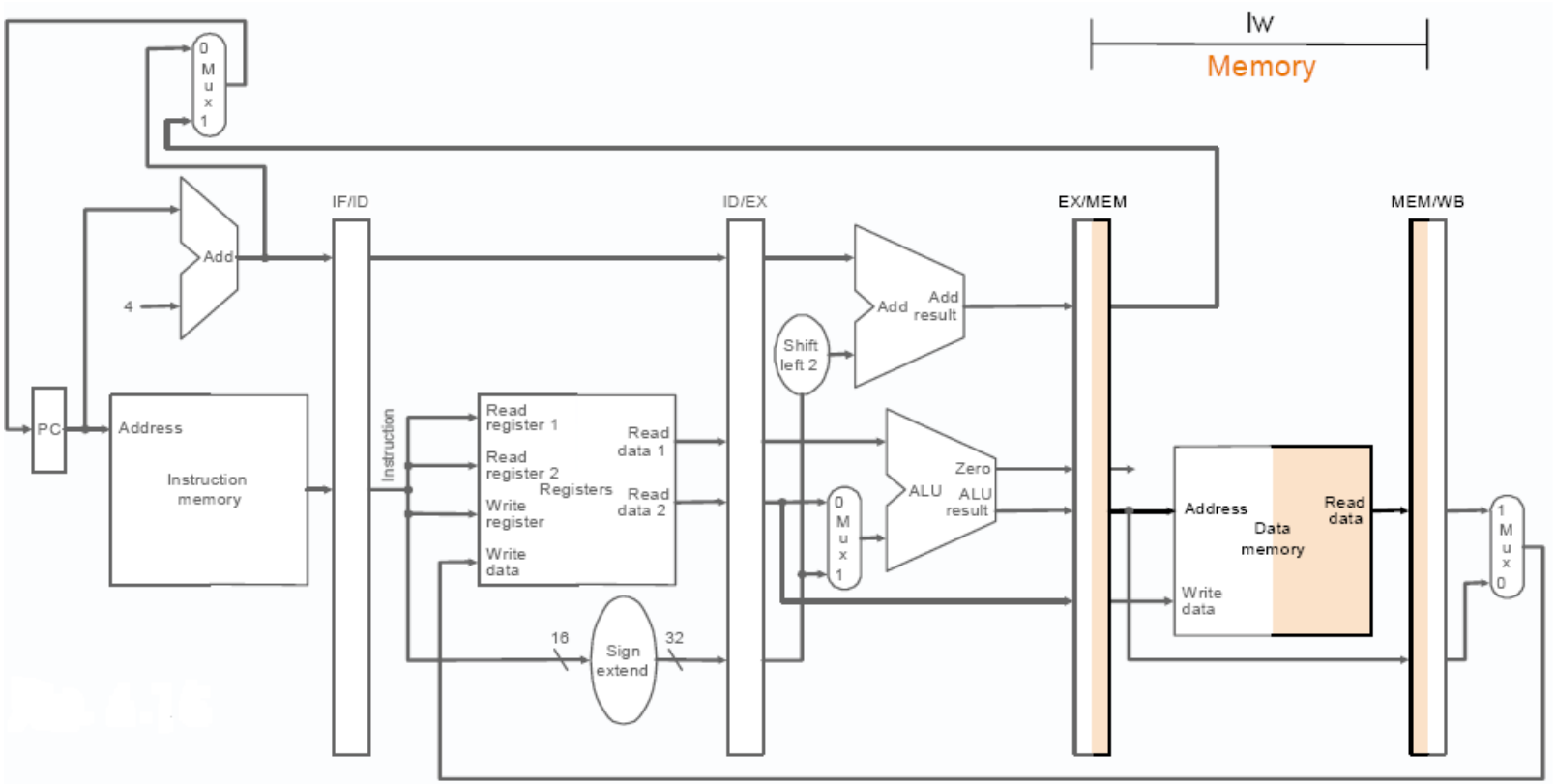
EX Stage of load word

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



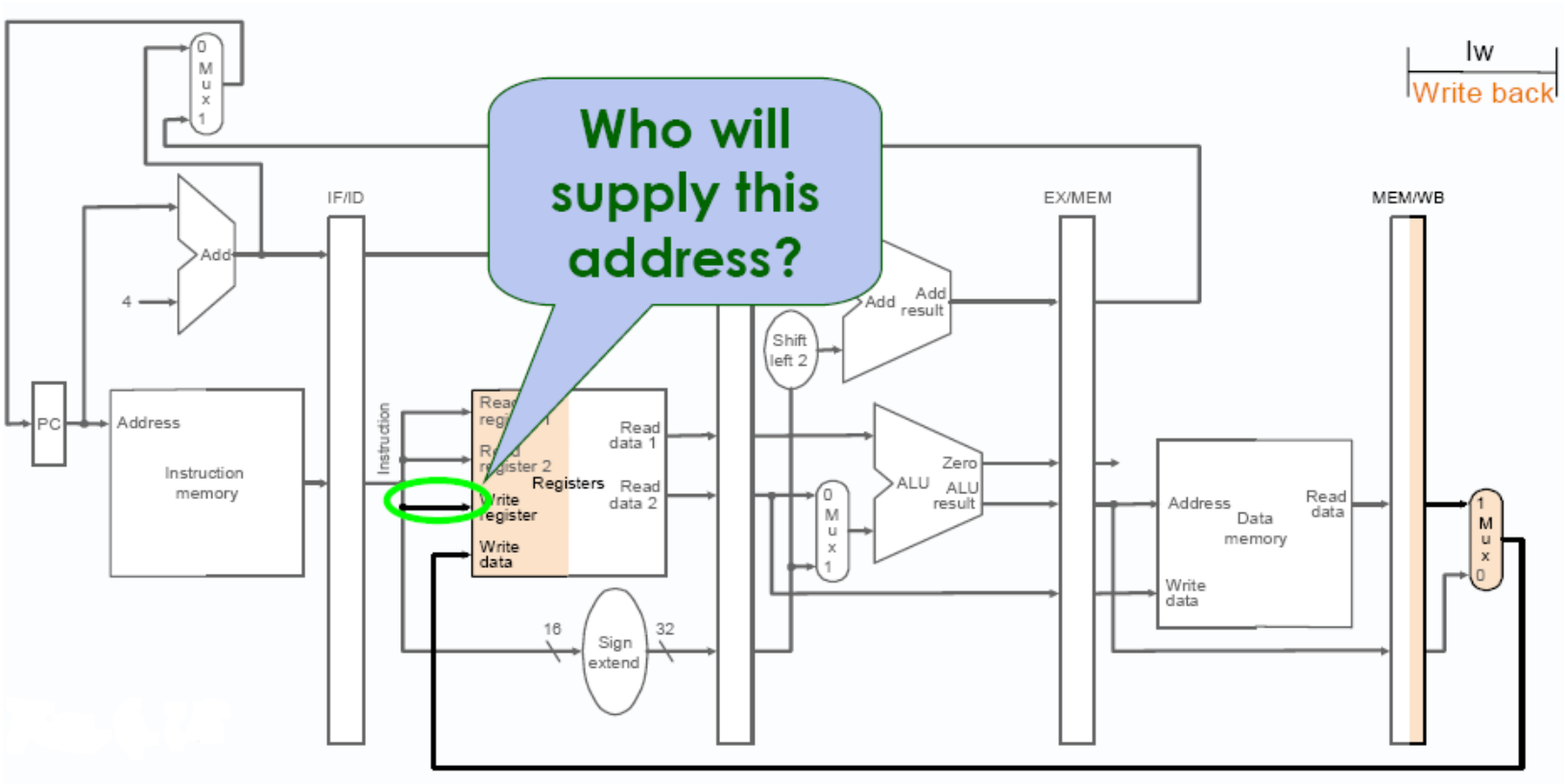
MEM Stage of load word

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

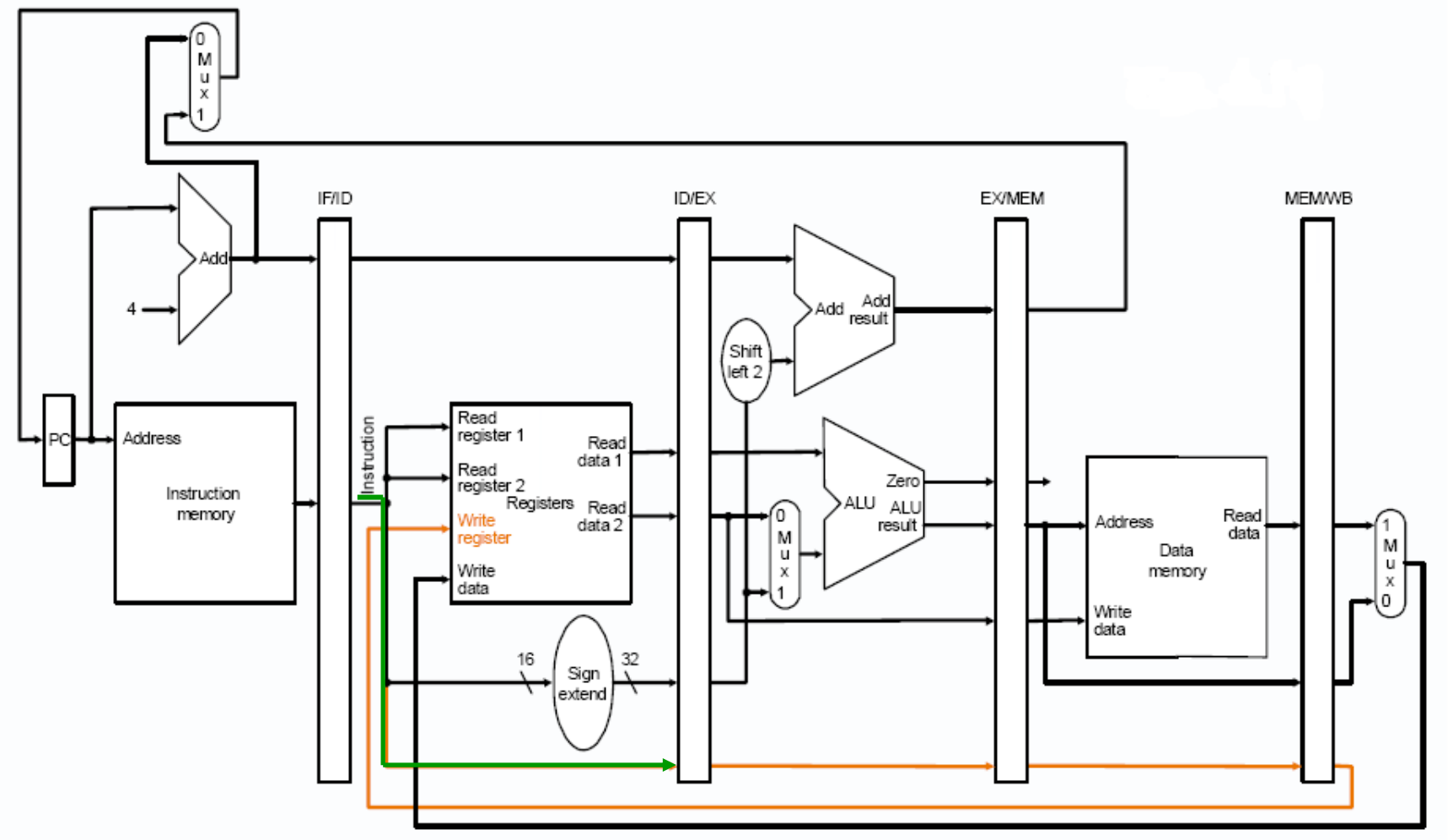


WB Stage of load

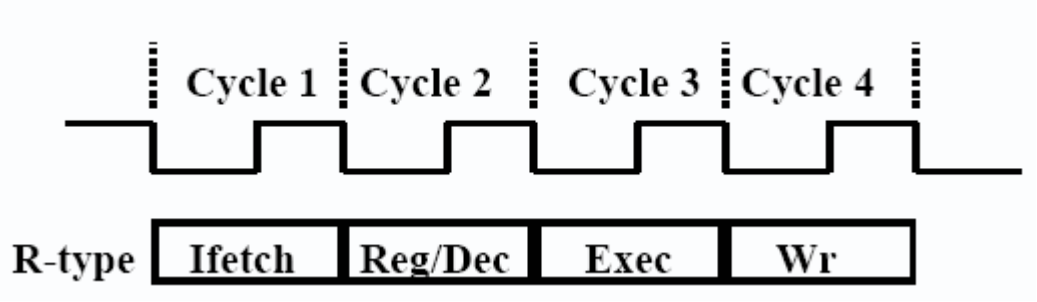
- Reg[IR[20-16]??] = MEM/WB



Solution for WB Stage

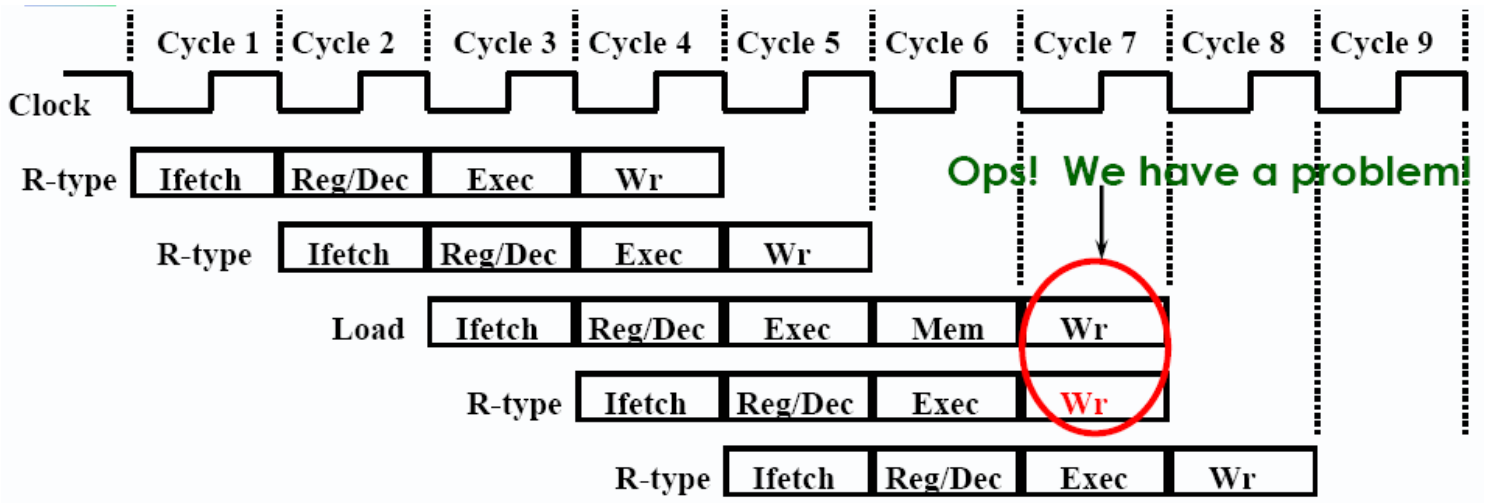


The Four Stages of R-type



- **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

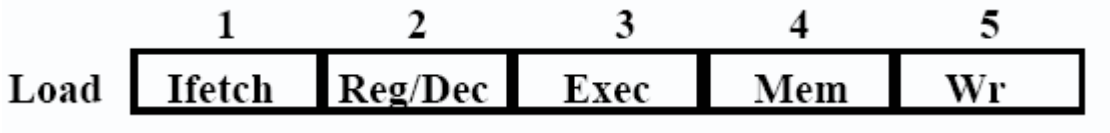
Pipelining R-type and load



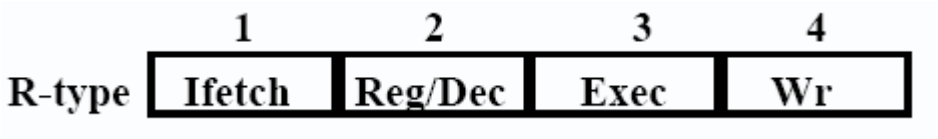
- We have a *structural hazard!*
  - Two instructions try to write to the same register file at the same time!
  - Only **one write port** is available

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

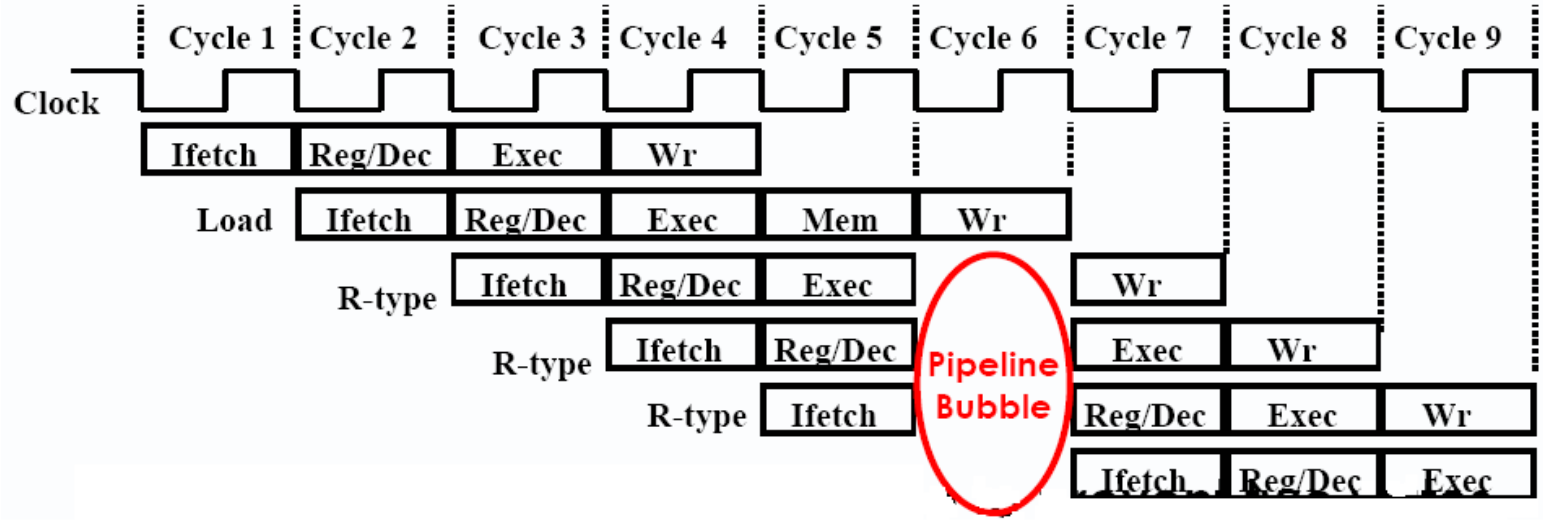


  - R-type uses Register File's write port during its 4th stage



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

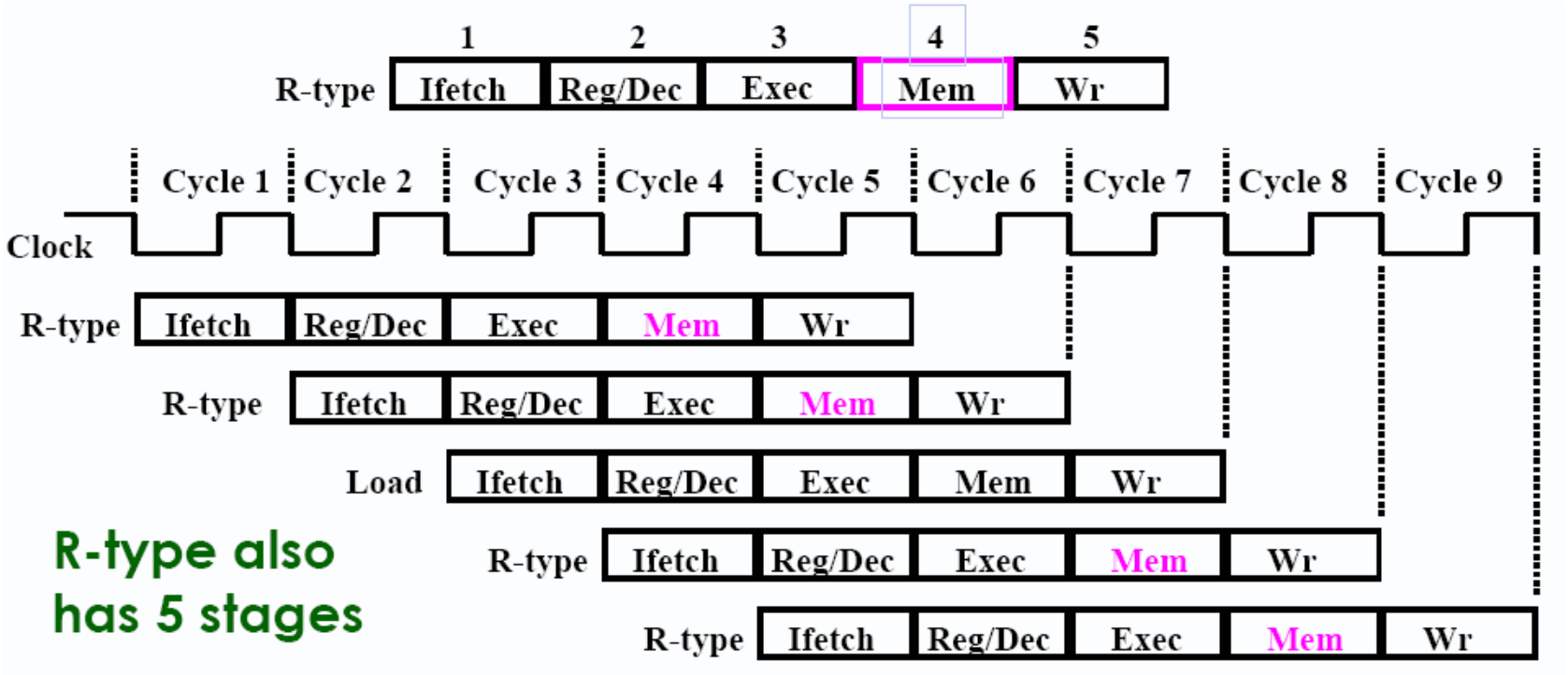
Solution 1: Insert Bubble (in hardware)



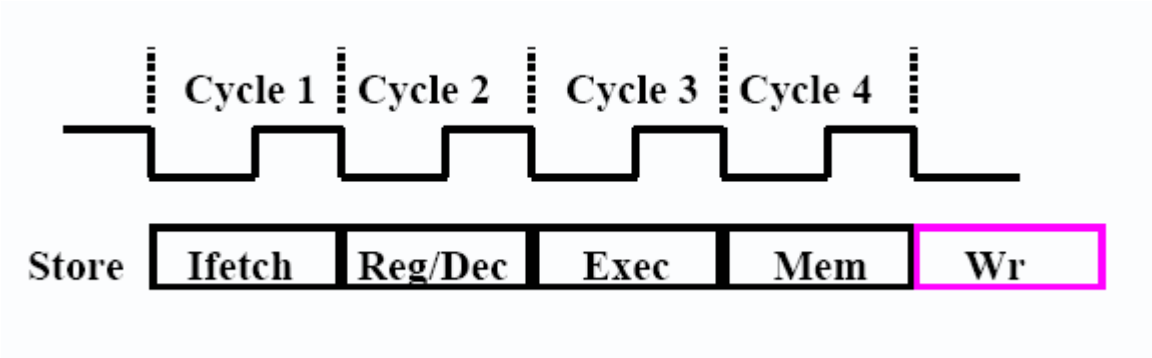
- Insert a bubble into the pipeline to prevent two writes at the same cycle
  - The control logic can be complex
  - Lose instruction fetch → No instruction is started in Cycle 6!

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

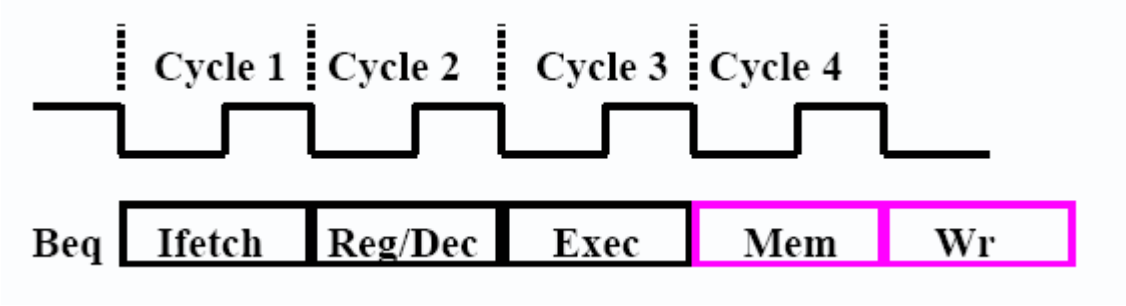


- **IF:** fetch the instruction from the Instruction Memory
- **ID:** registers fetch and instruction decode
- **EX:** calculate the memory address
- **MEM:** write the data into the Data Memory

Add an extra stage:

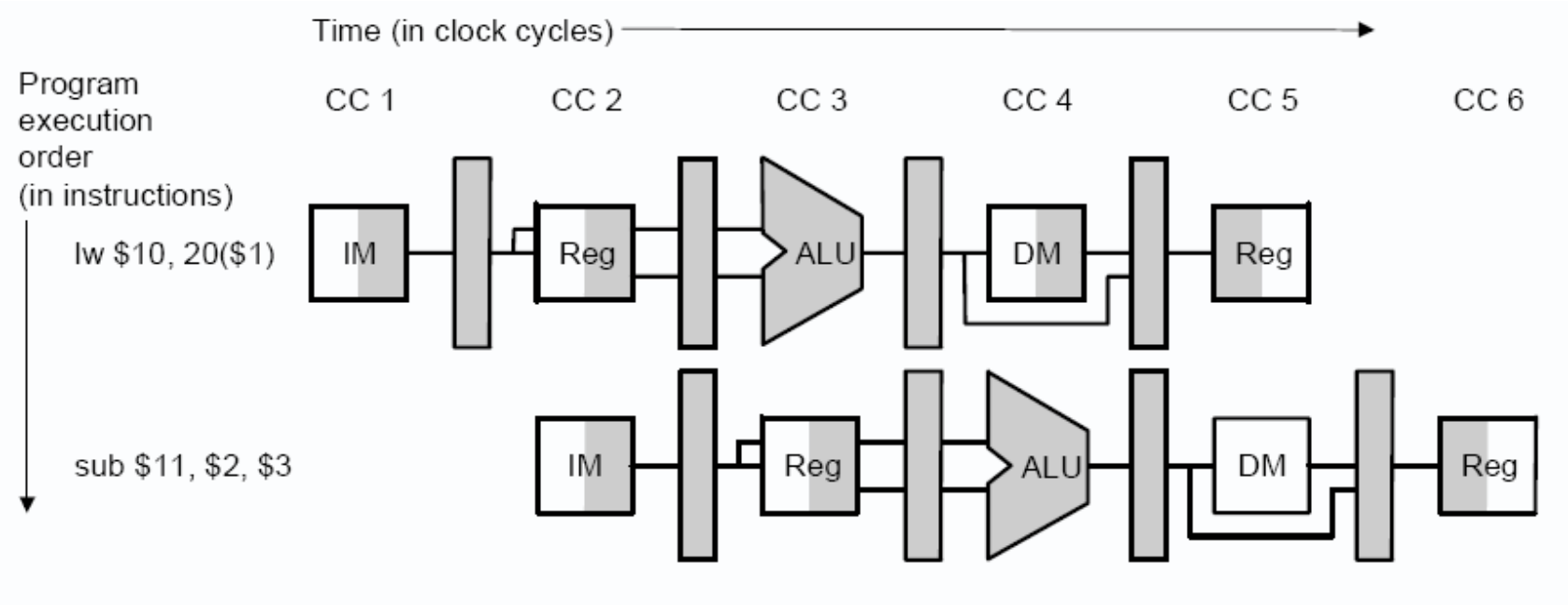
- **WB: NOP**

The Four Stages of beq



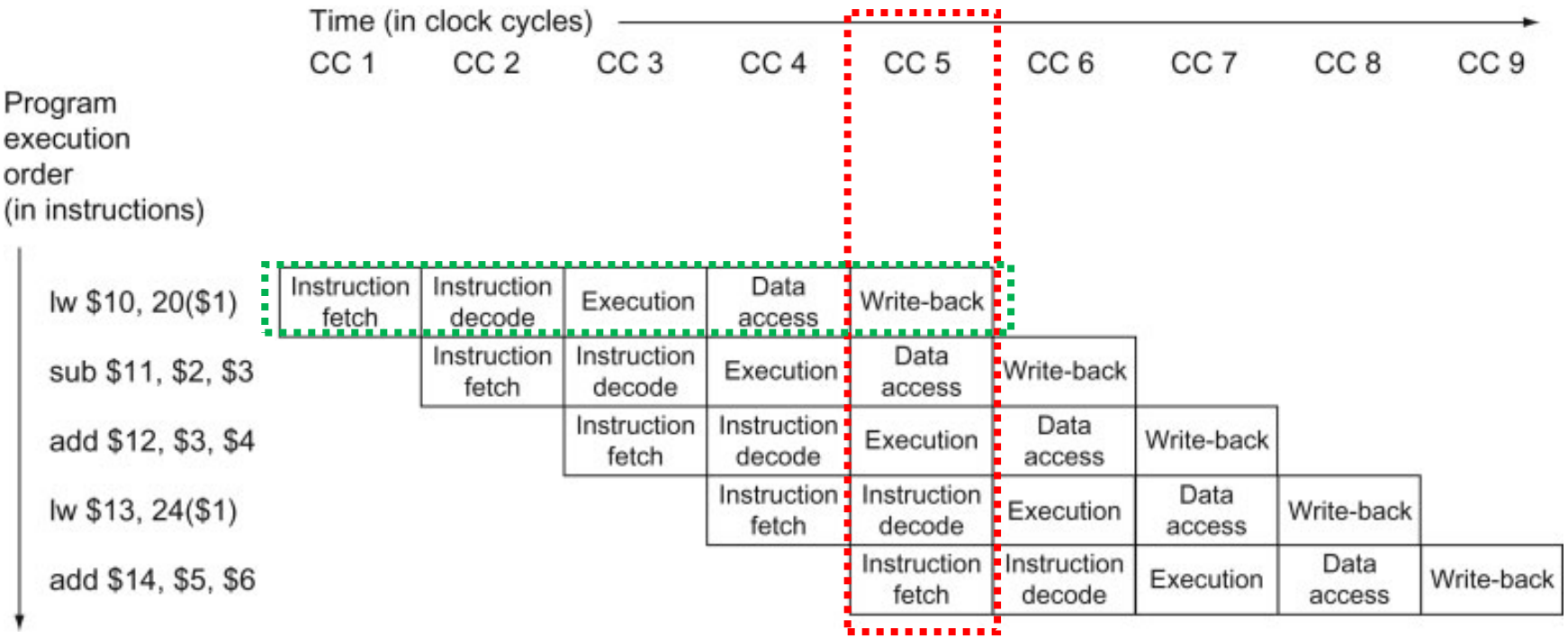
- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX:
  - compares the two register operand
  - select correct branch target address
  - latch into PC
- Add two extra stages:
- MEM: NOP
- WB: NOP

Multiple-clock-cycle Pipeline Diagram

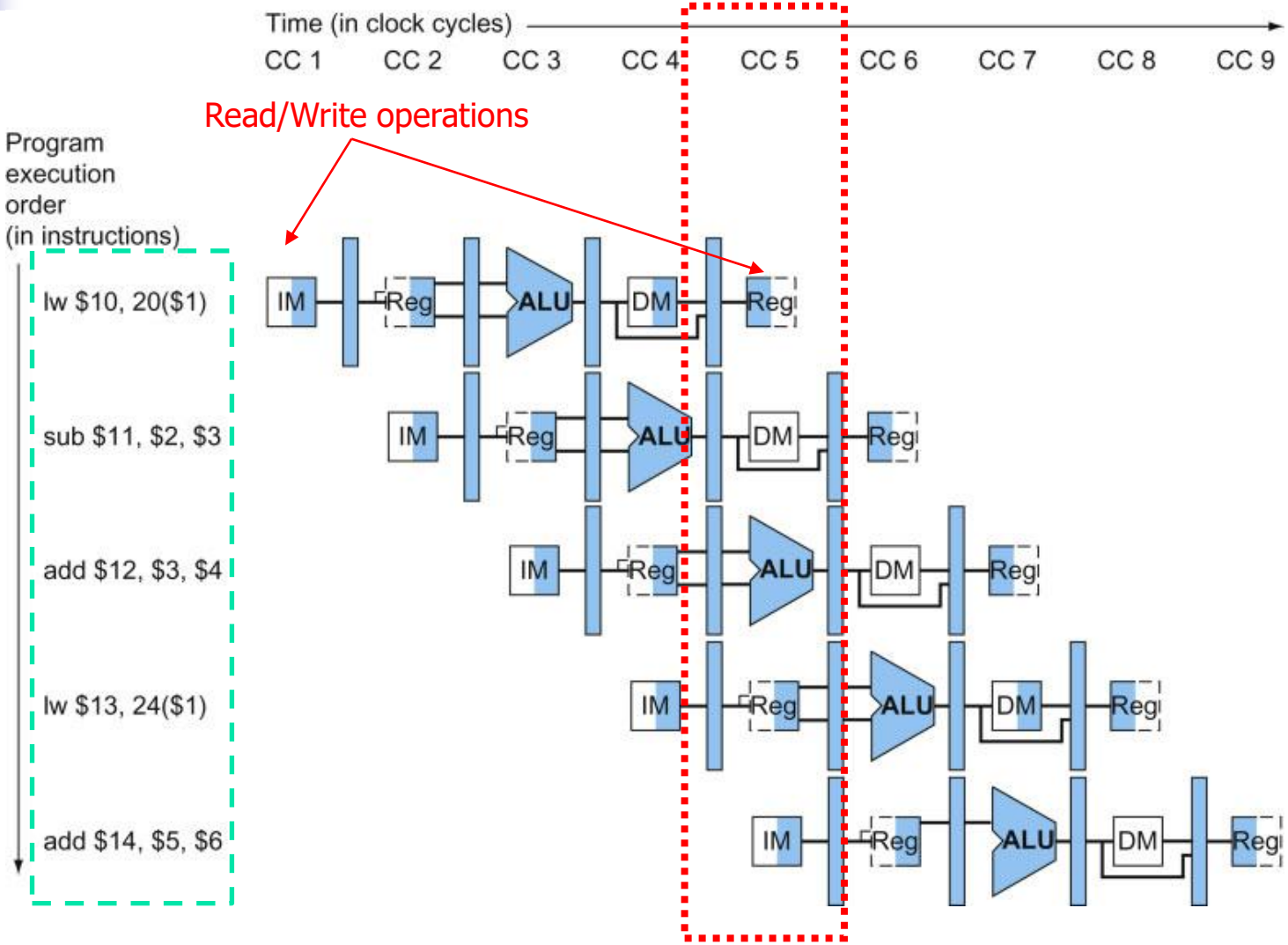


- (Fig. 4.34) Can help with answering questions like:
  - How many cycles to execute this code?
  - What is the ALU doing during cycle 4?
  - Help understand datapaths & Read/Write operations

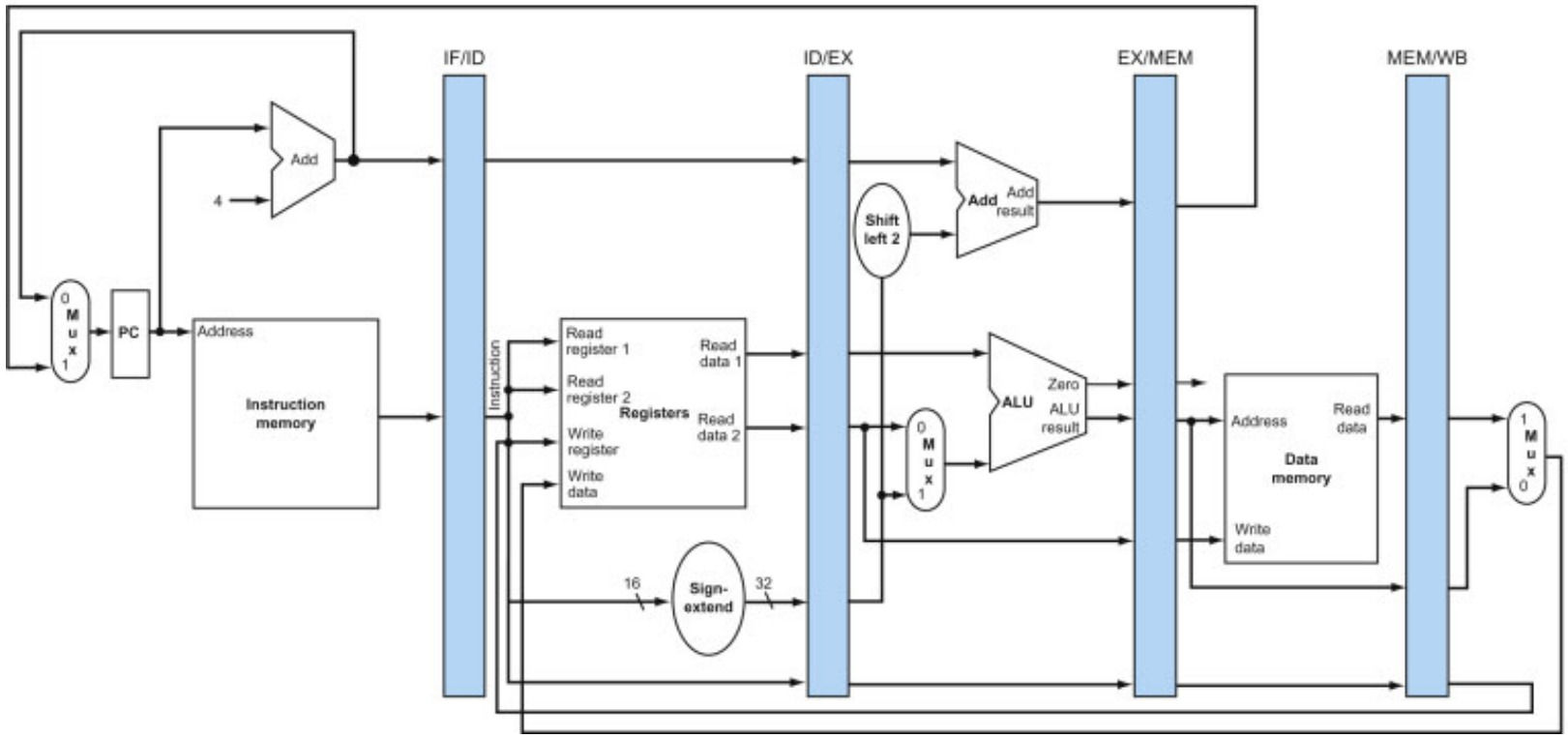
Traditional multi-clock-cycle Pipeline diagram of Fig. 4.44



Five instructions (Fig. 4.43) – Snapshot at CC5

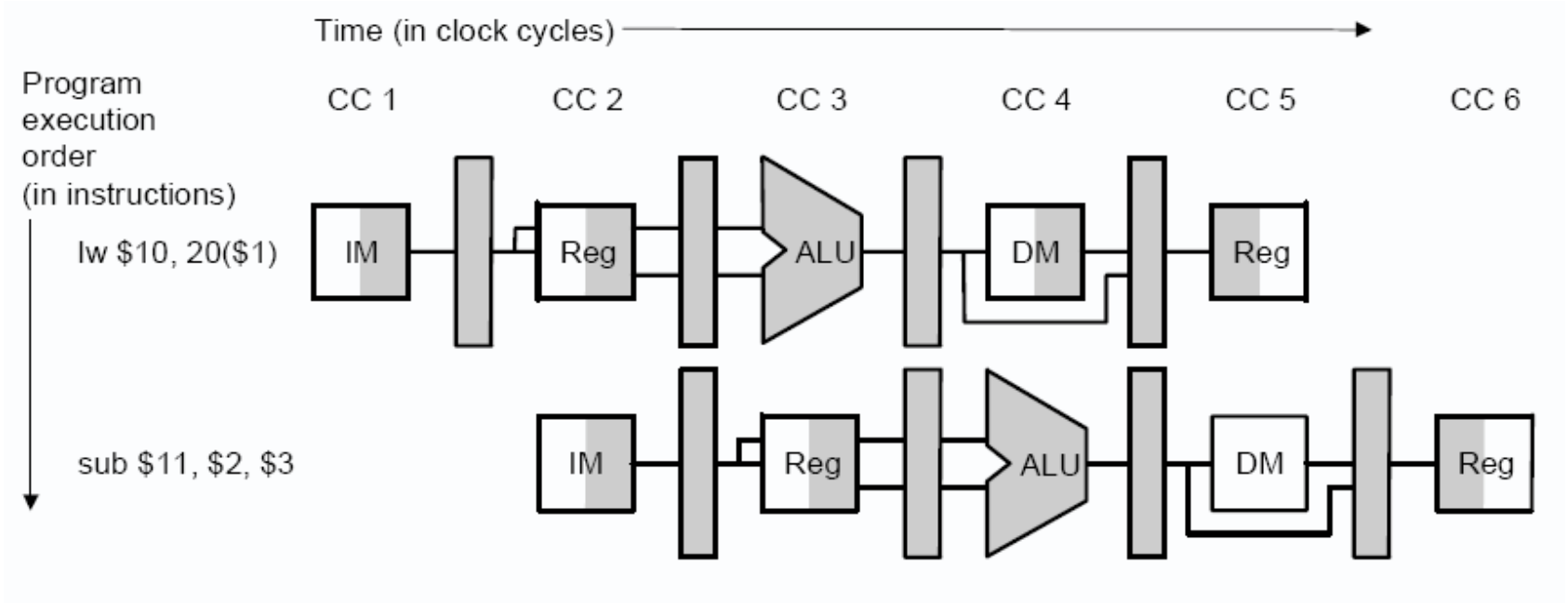


Single-clock-cycle diagram of Clock 5 of Fig. 4.45

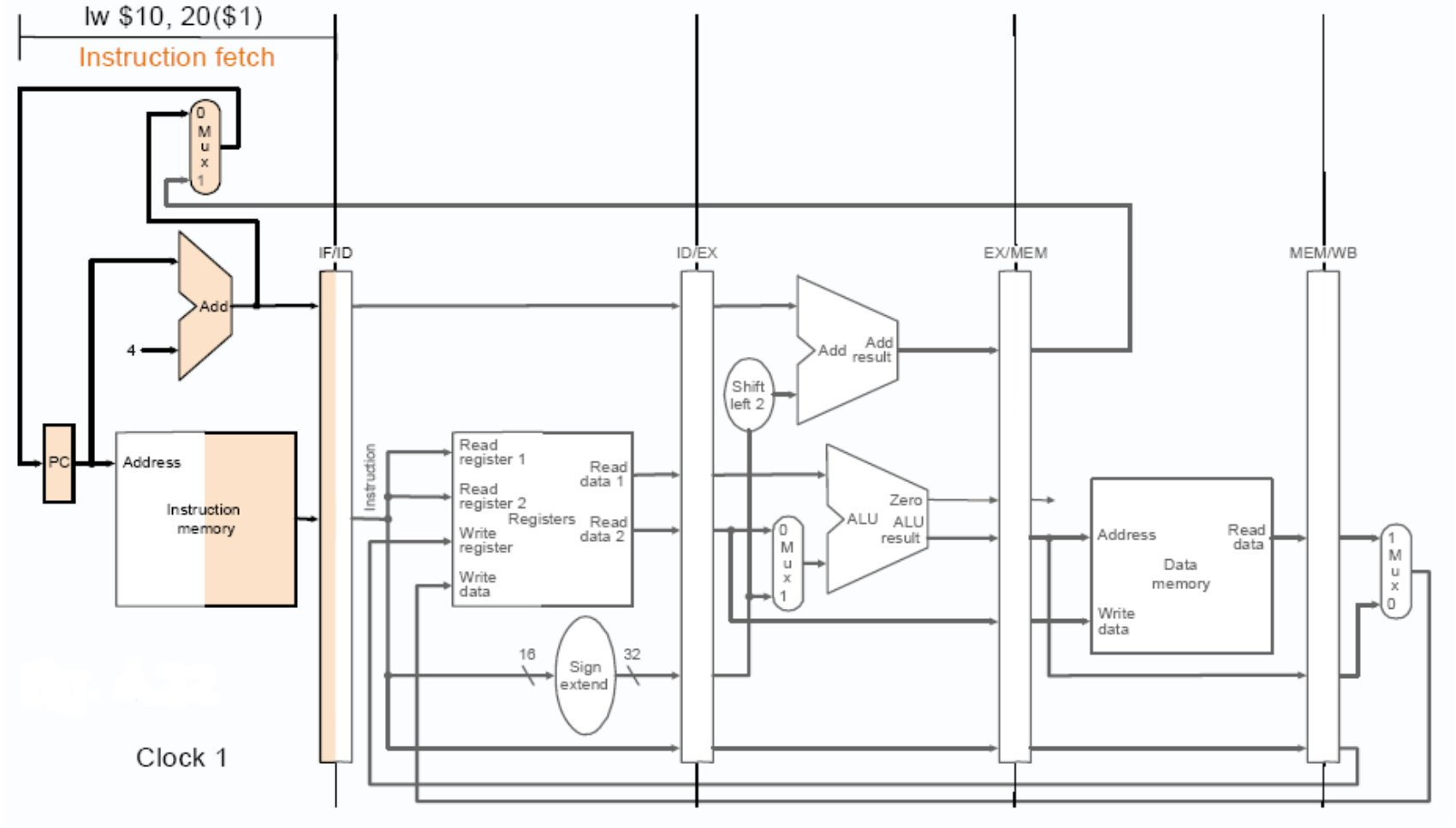


A vertical slice through a multiple-clock-cycle diagram!

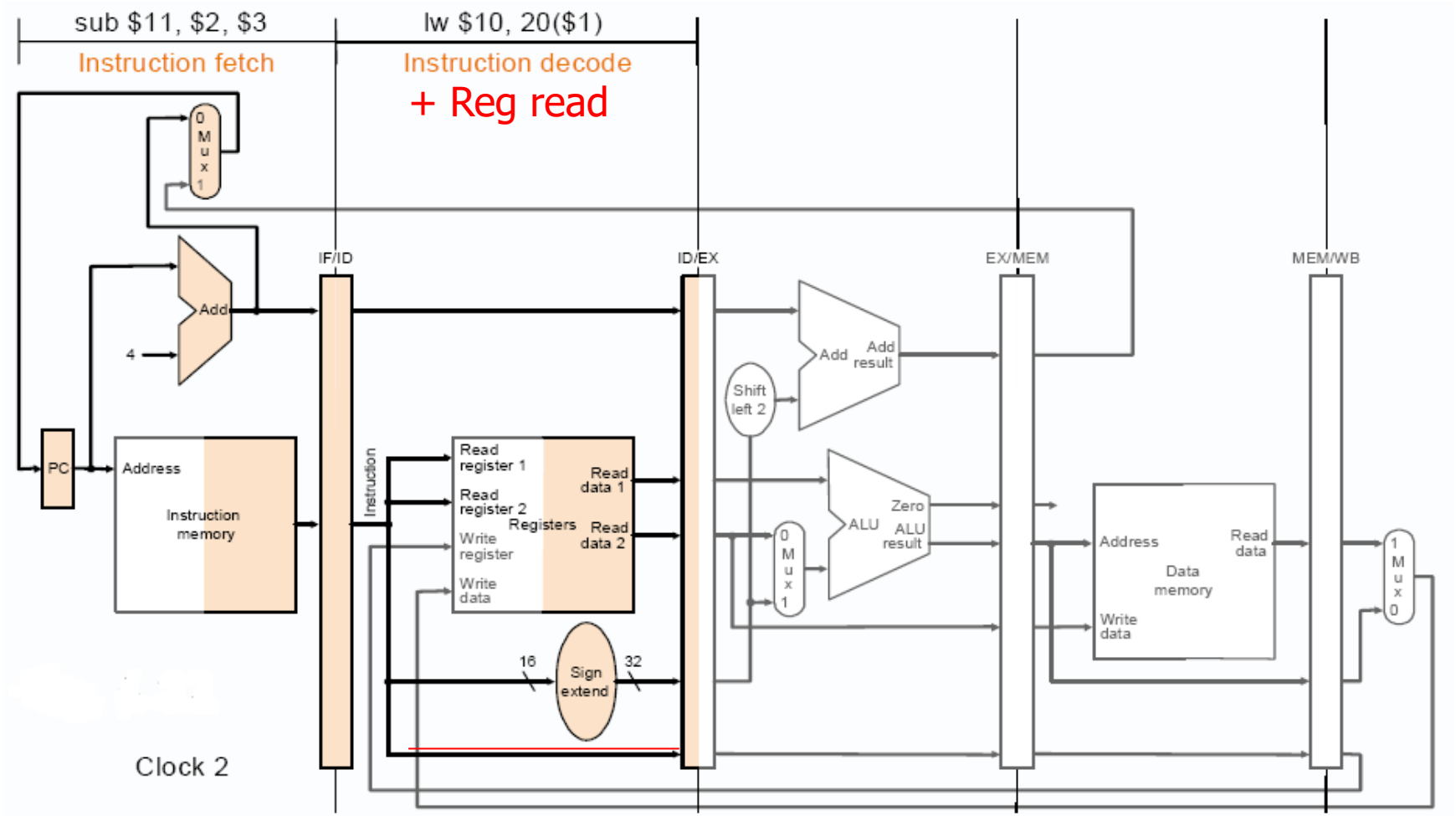
Example 1: *lw* → *sub*



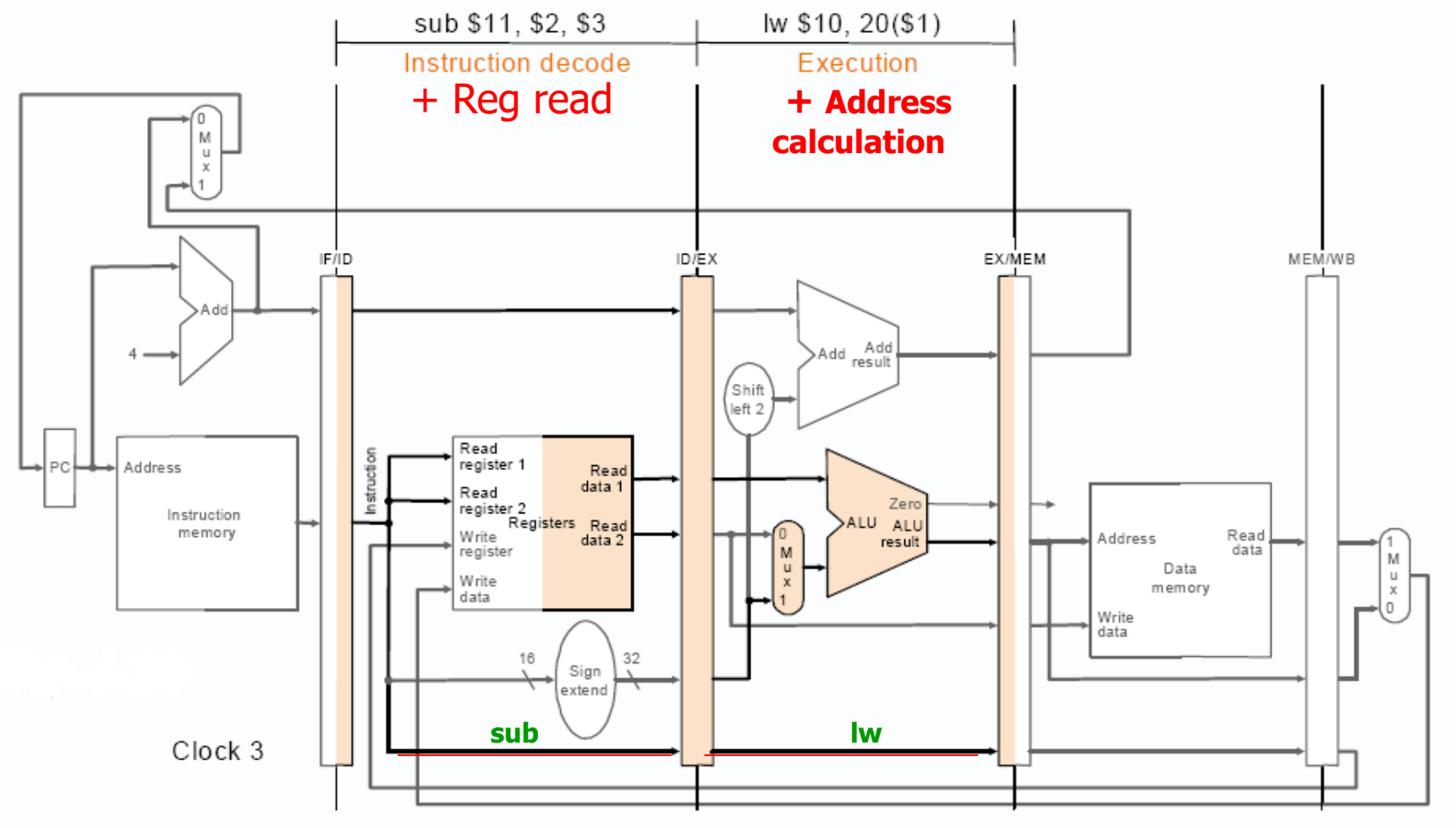
Example 1: Cycle 1



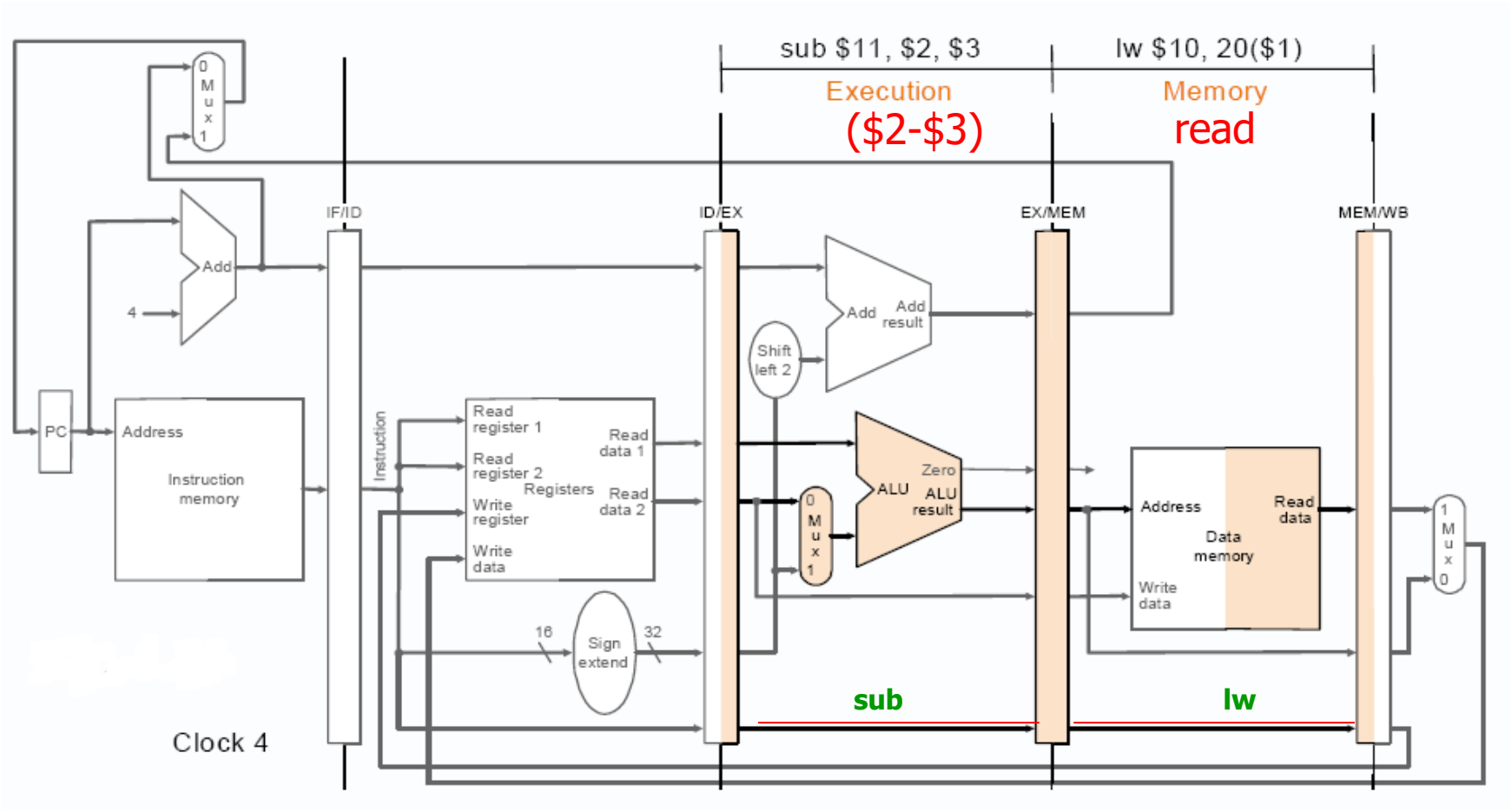
Example 1: Cycle 2



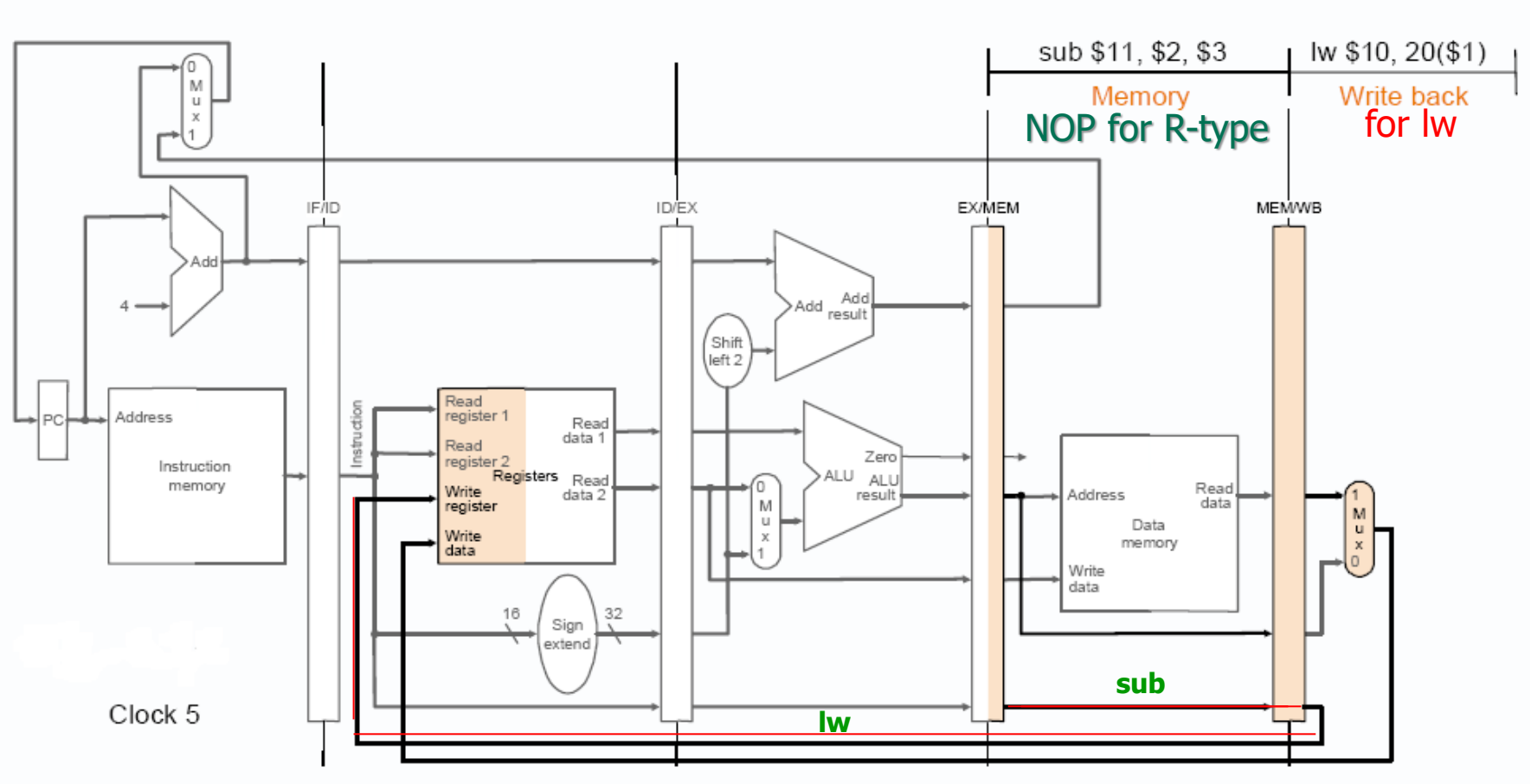
Example 1: Cycle 3



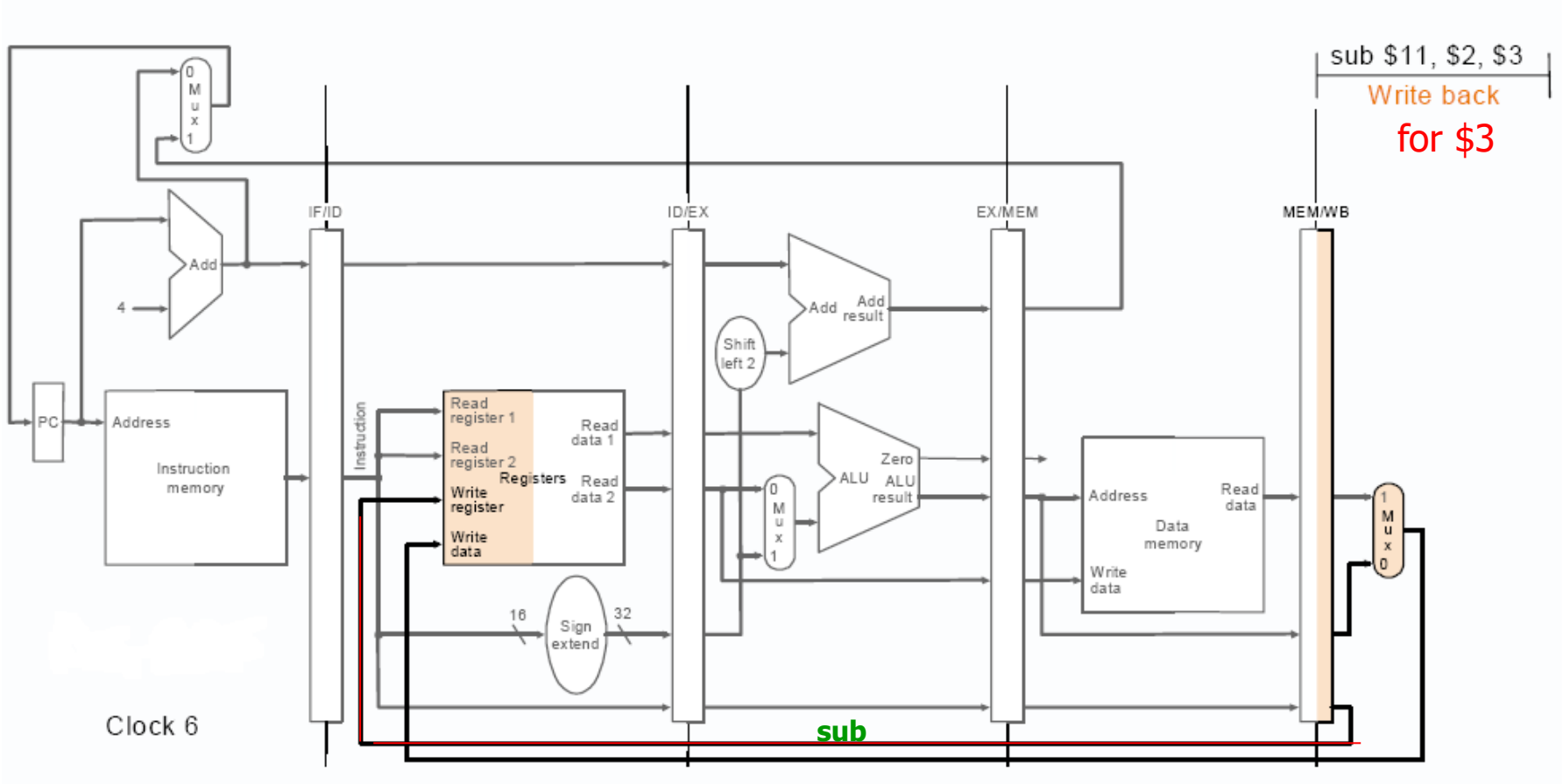
Example 1: Cycle 4



Example 1: Cycle 5

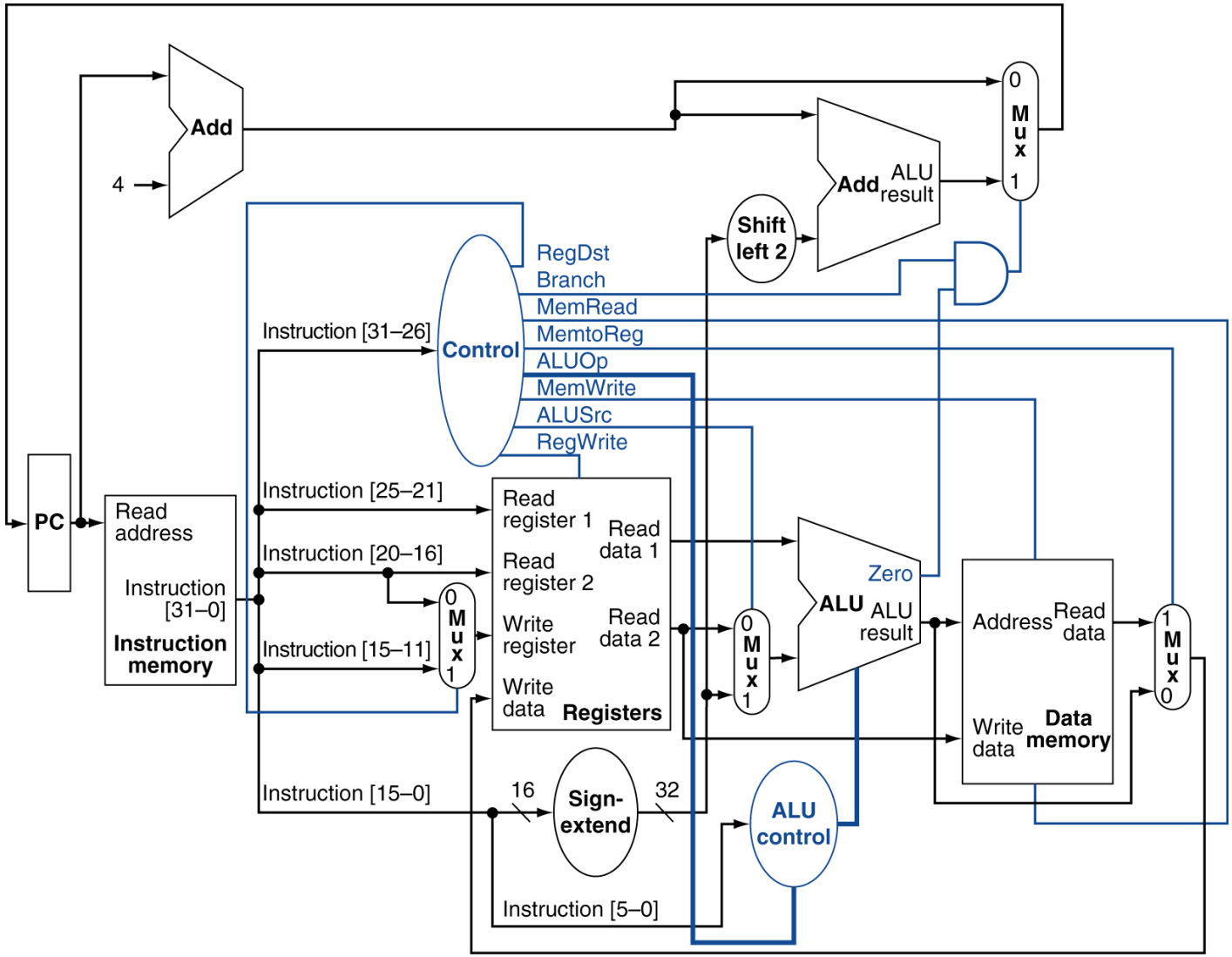


Example 1: Cycle 6



Control Signals in Pipeline

Review of MIPS Datapath With Control Signals



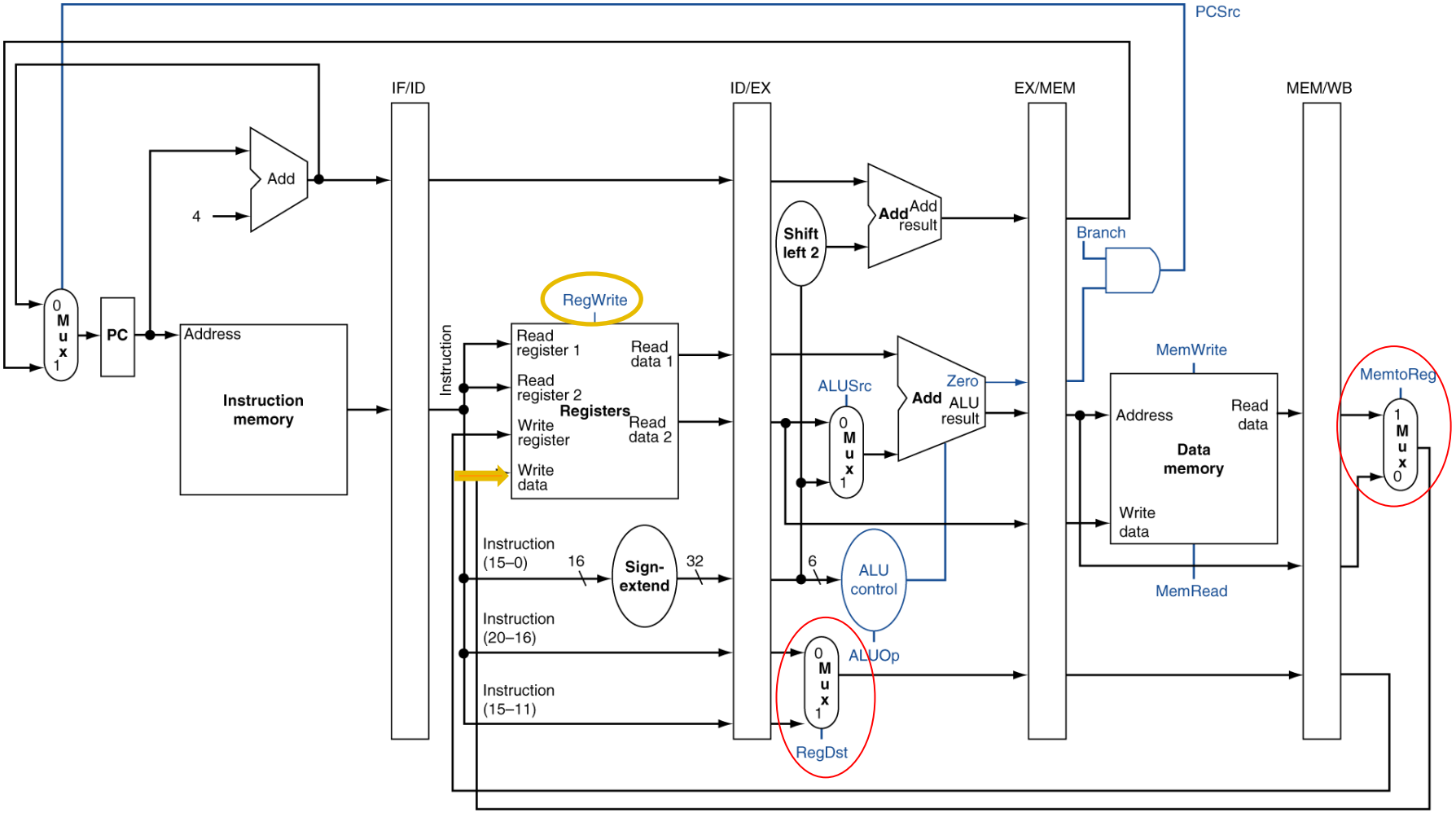
Control Signal Details (review)

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.

Control Signal Details (review)

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

Pipeline Control: Control Signals



Control Unit Design (from Single-cycle design)

The setting of the control lines is completed by the “opcode” field (op[5:0]) of the instruction.

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	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

Group Signals According to Stages

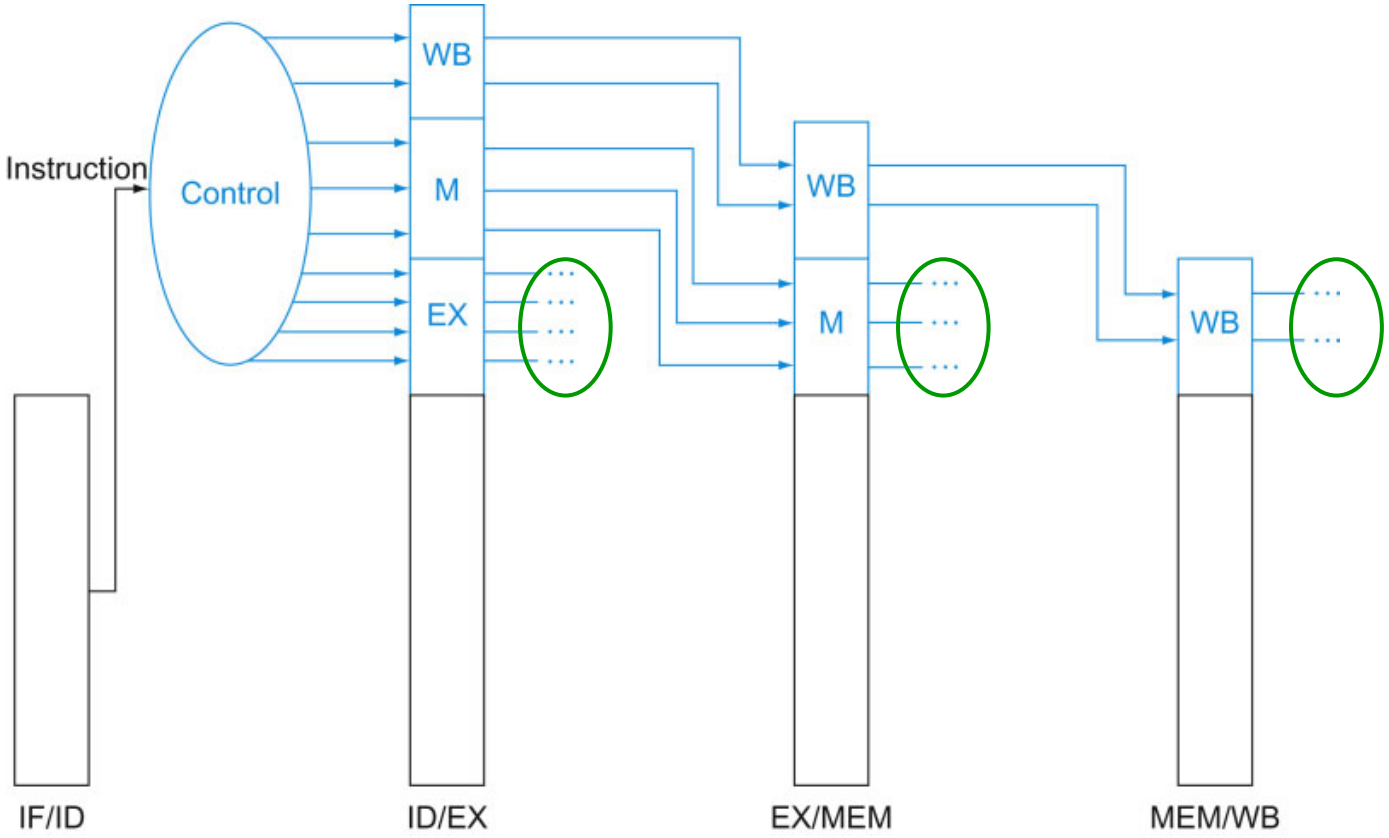
- Can use control signals of single-cycle CPU

	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)

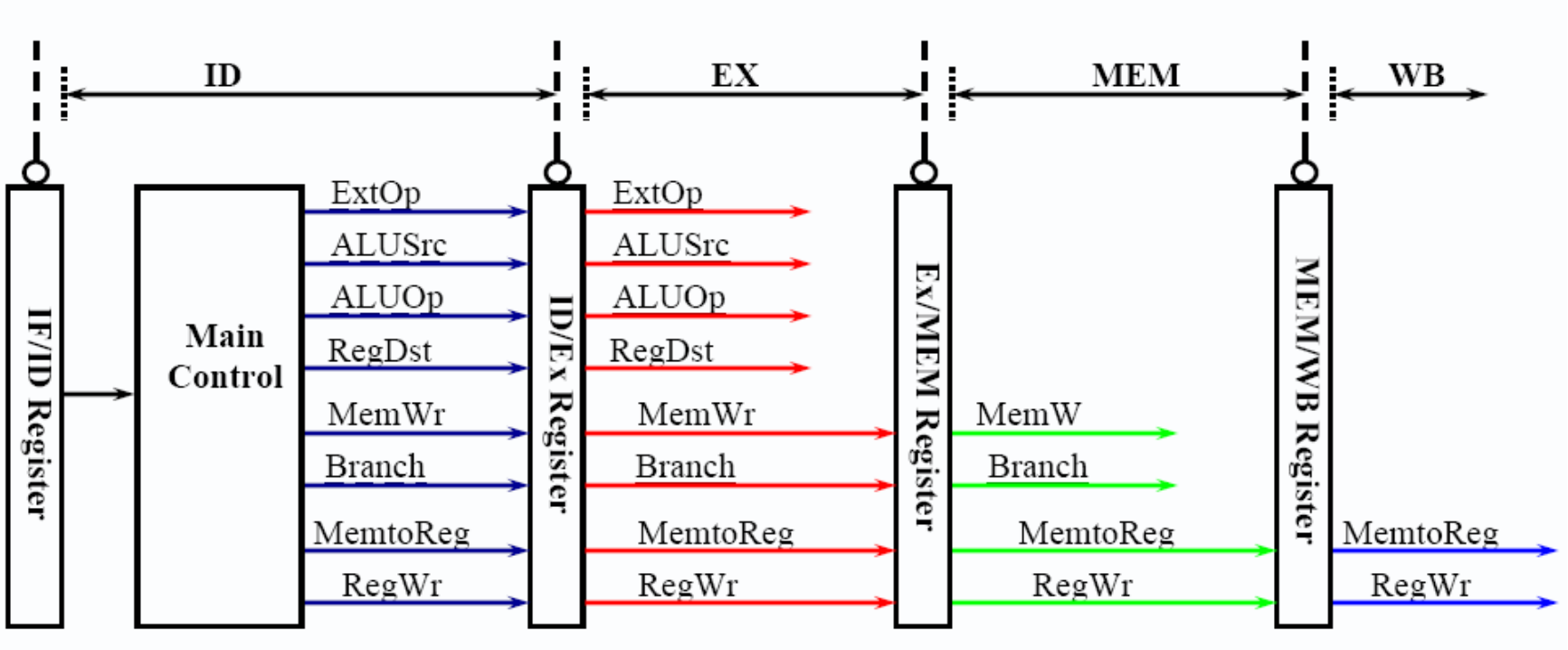
Data Stationary Control

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

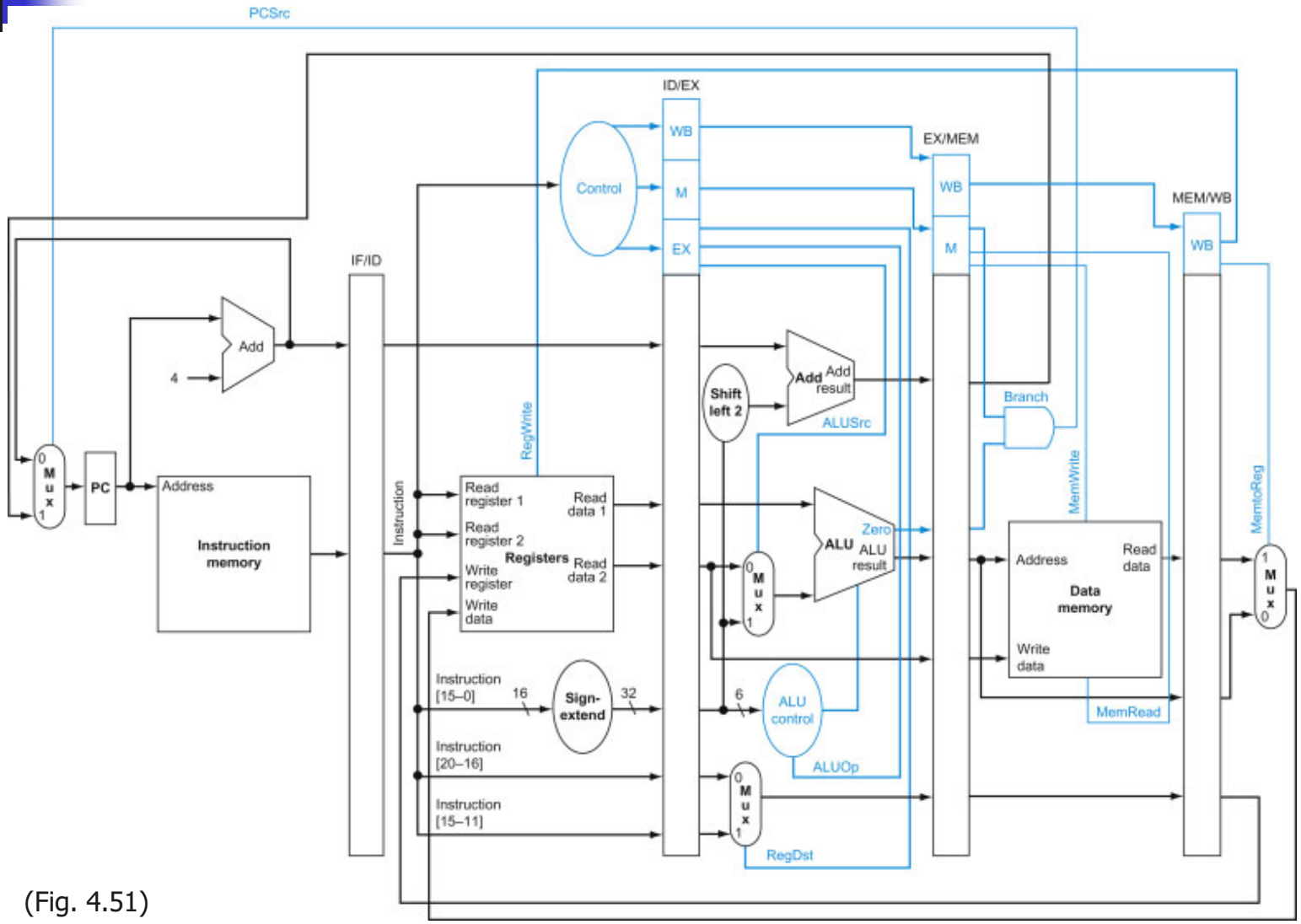


Data Stationary Control (cont.)

- 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



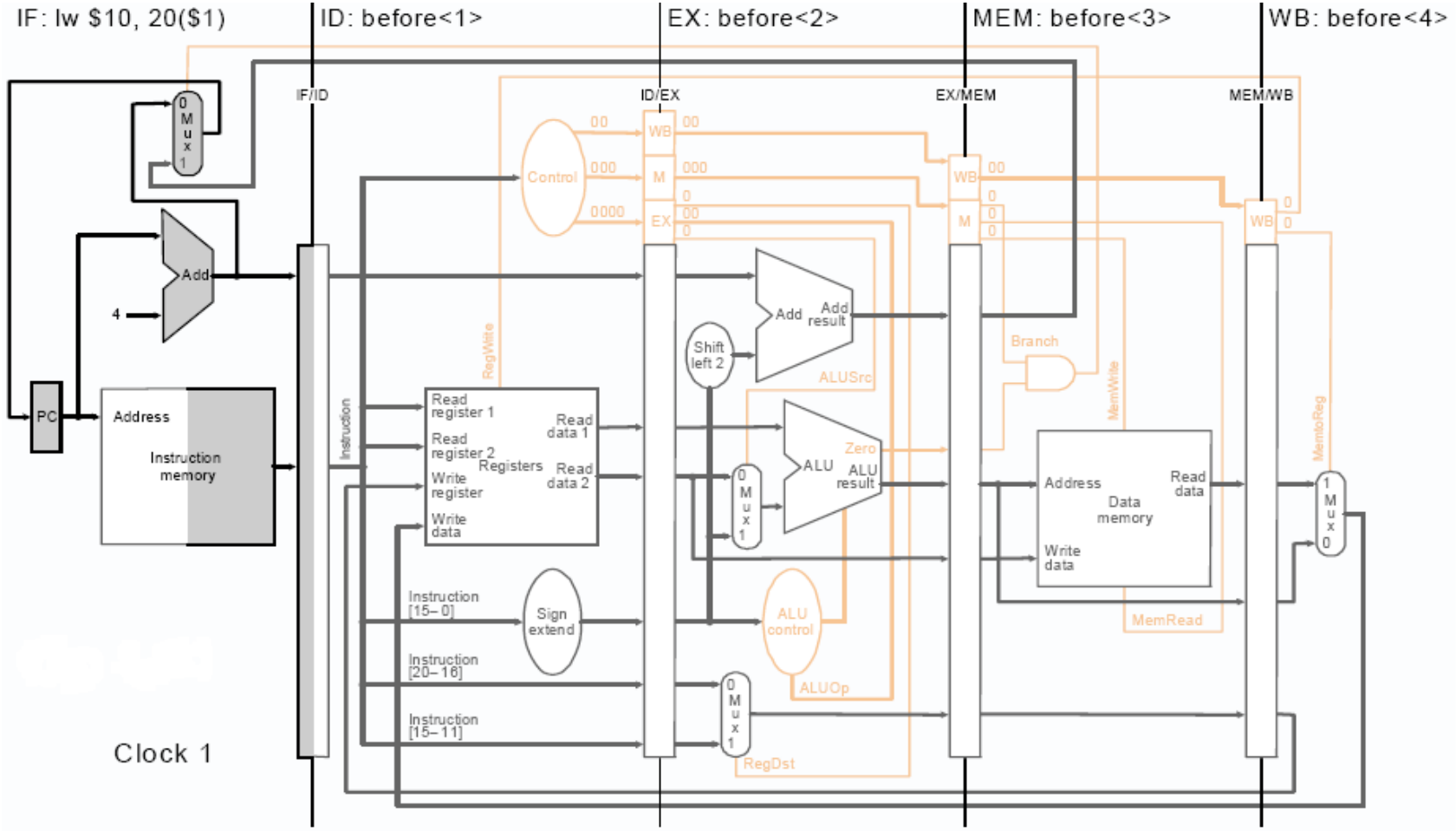
(Fig. 4.51)

Let's Try it Out

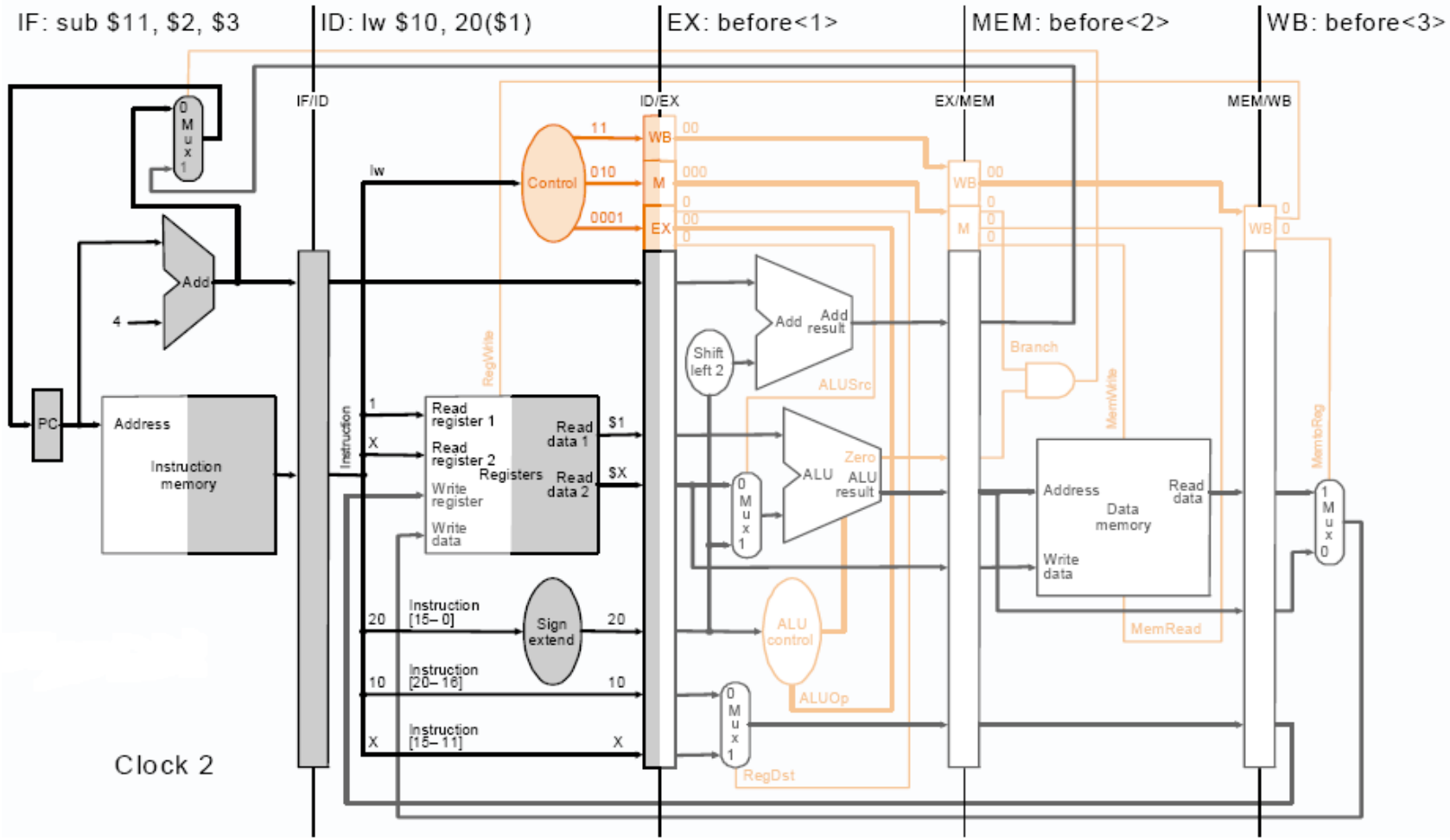
Sample Assembly Program (watch fill/flush)

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

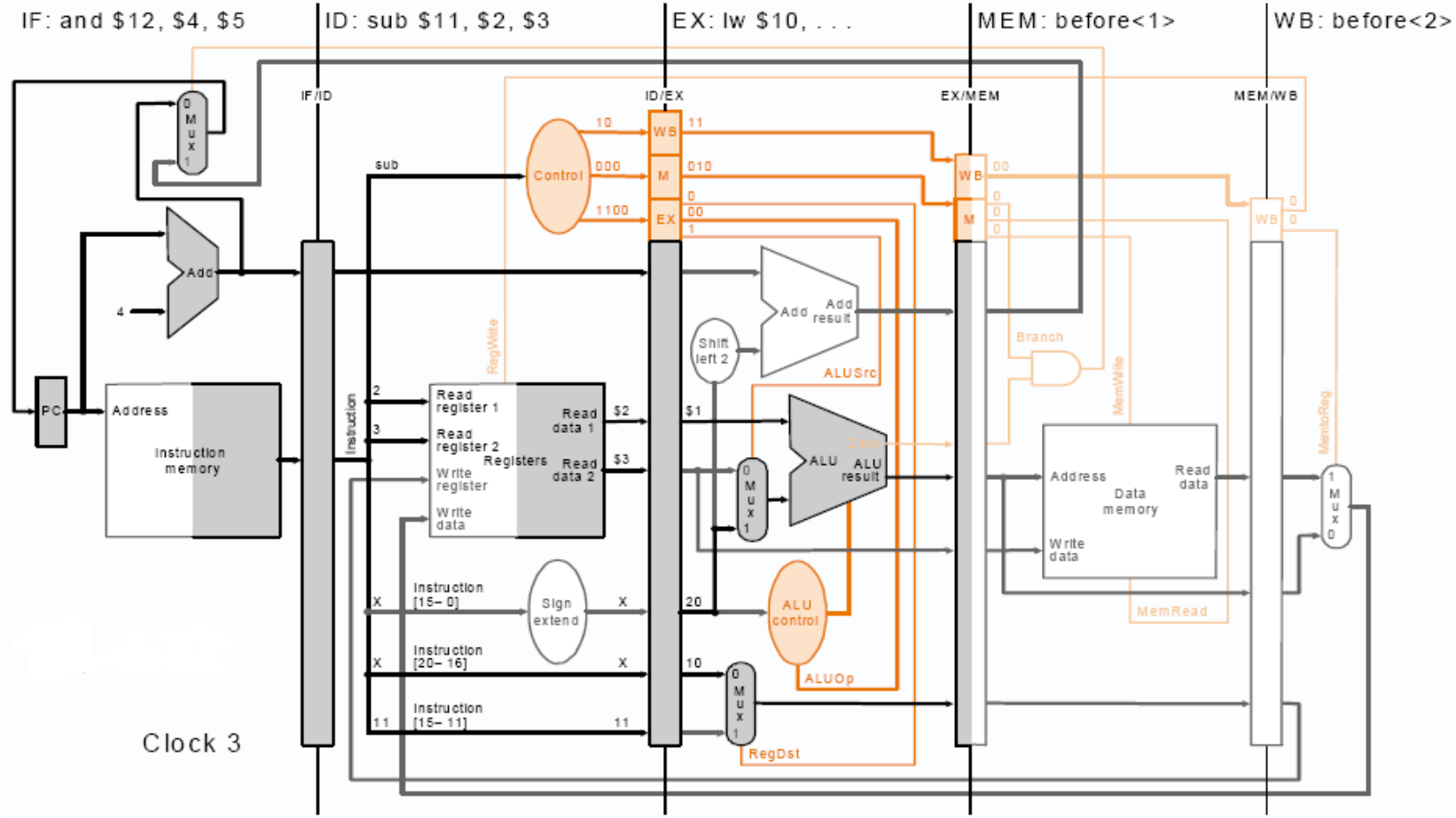
Example 2: Cycle 1



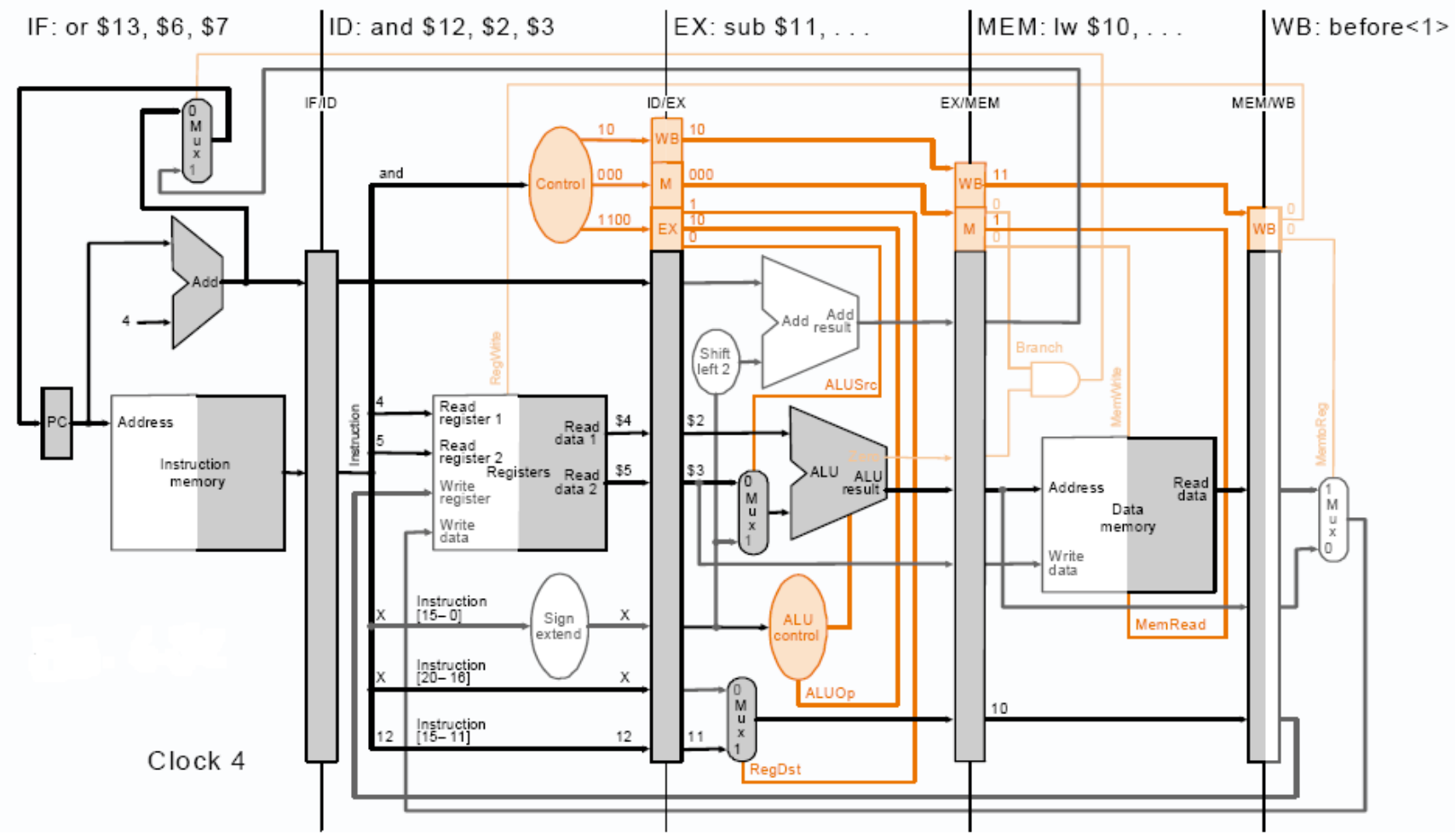
Example 2: Cycle 2



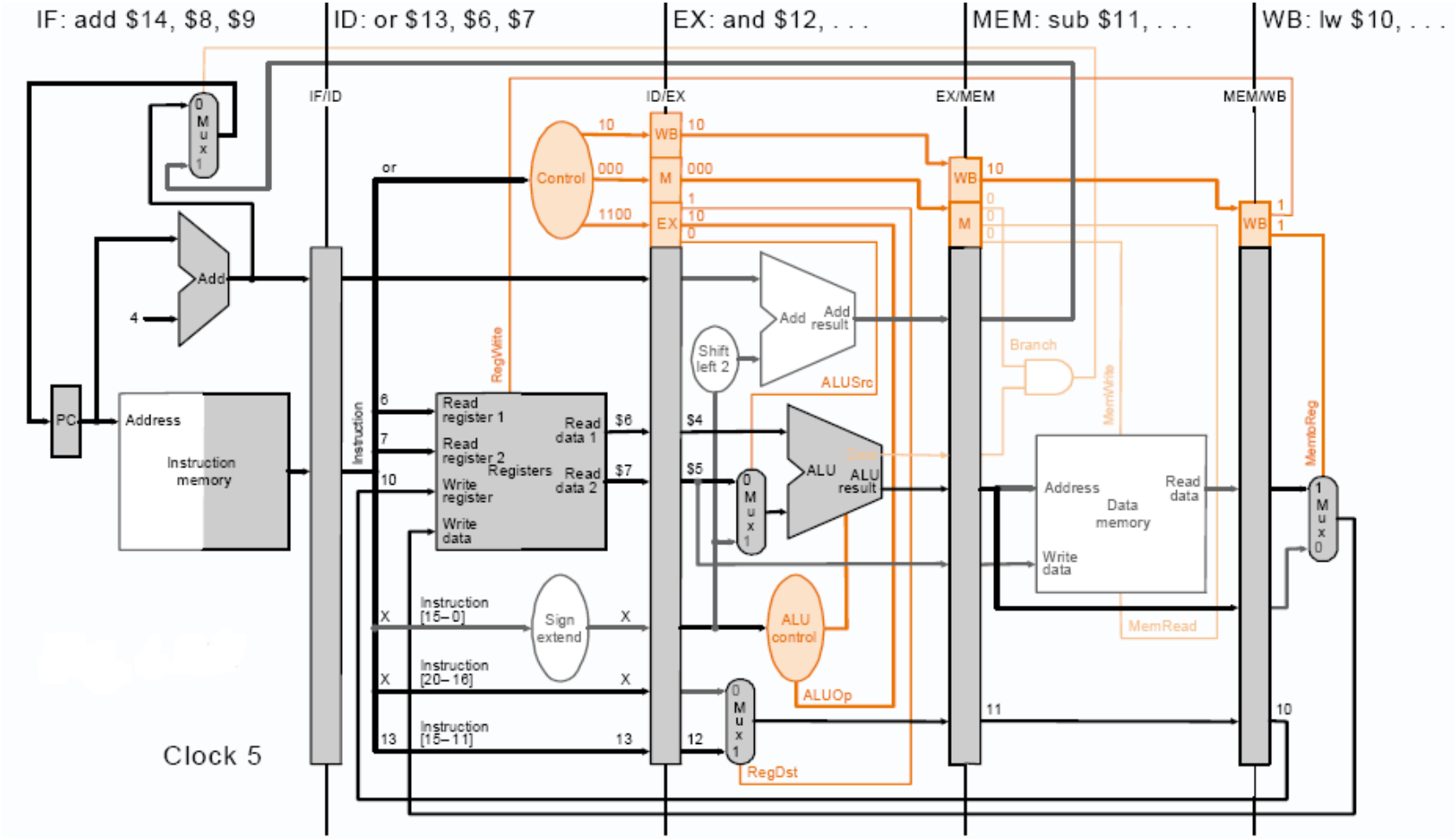
Example 2: Cycle 3



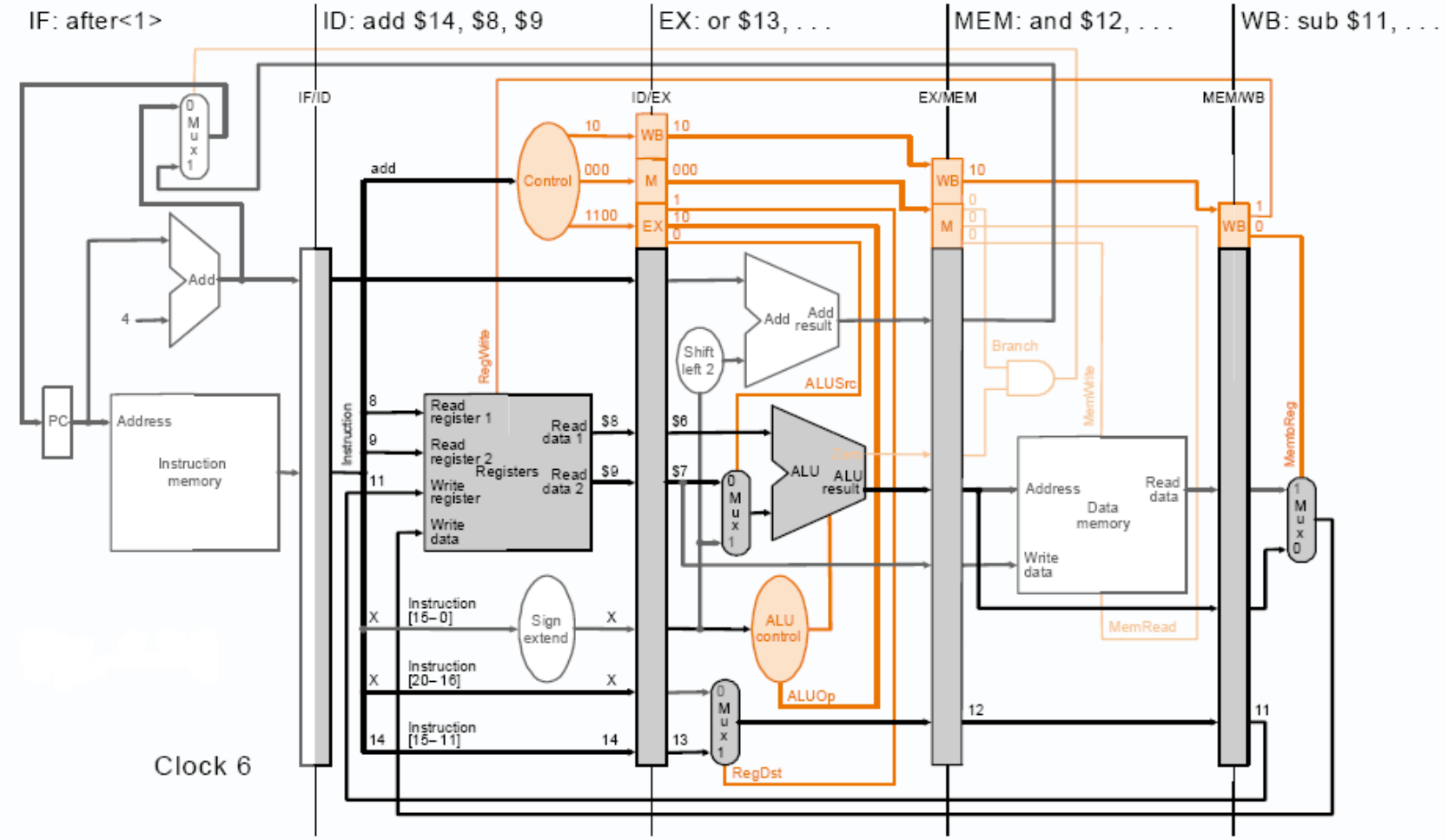
Example 2: Cycle 4



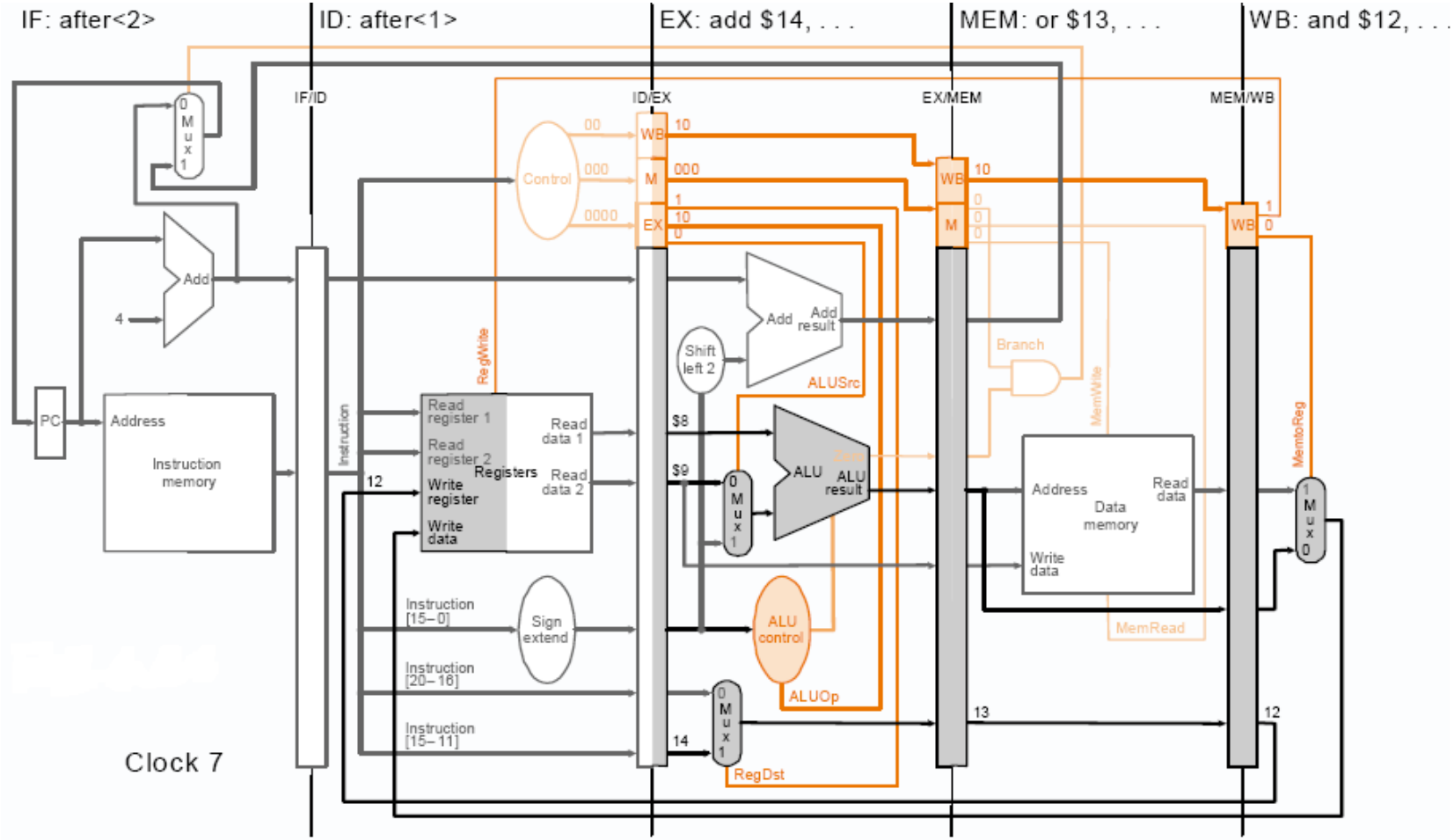
Example 2: Cycle 5



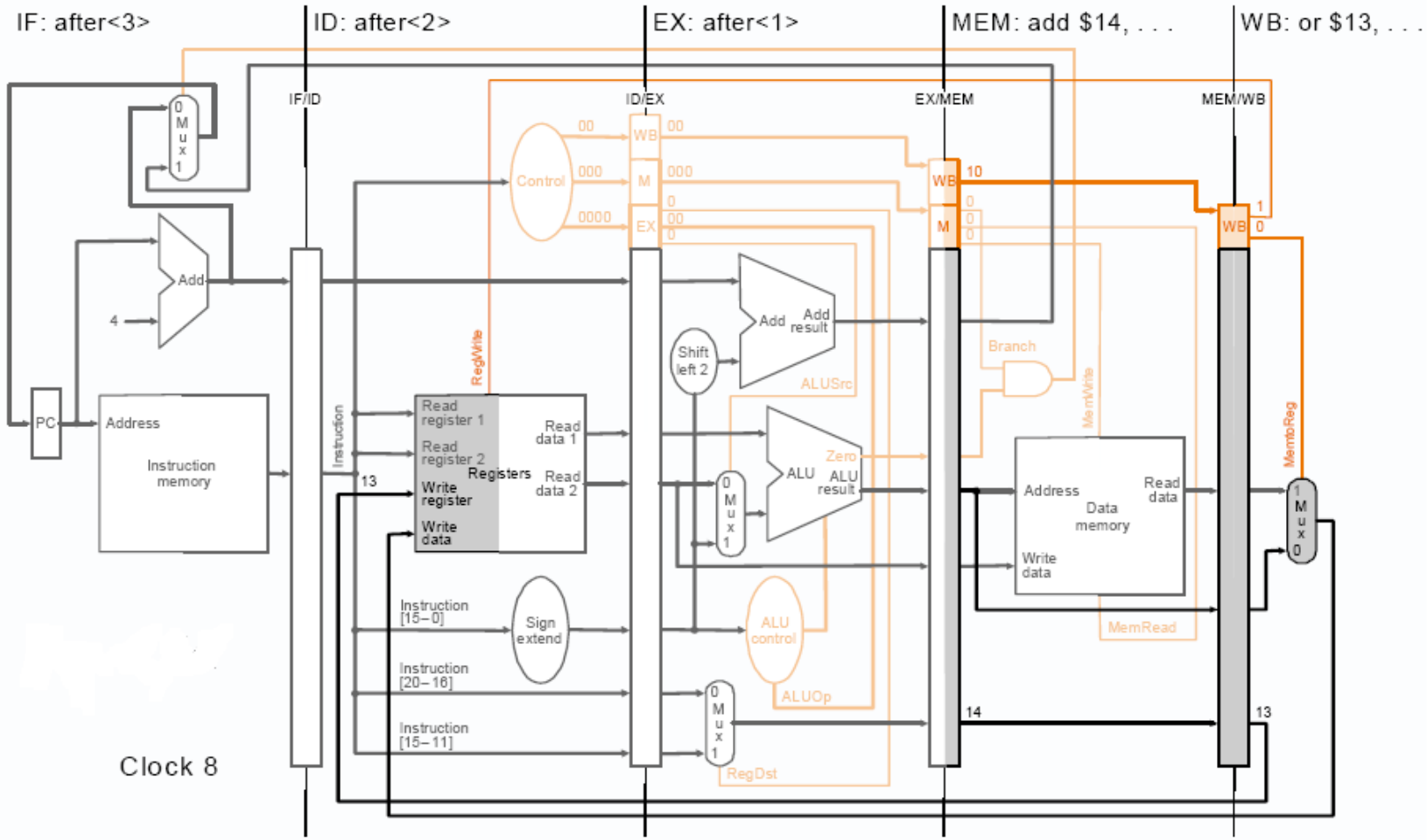
Example 2: Cycle 6



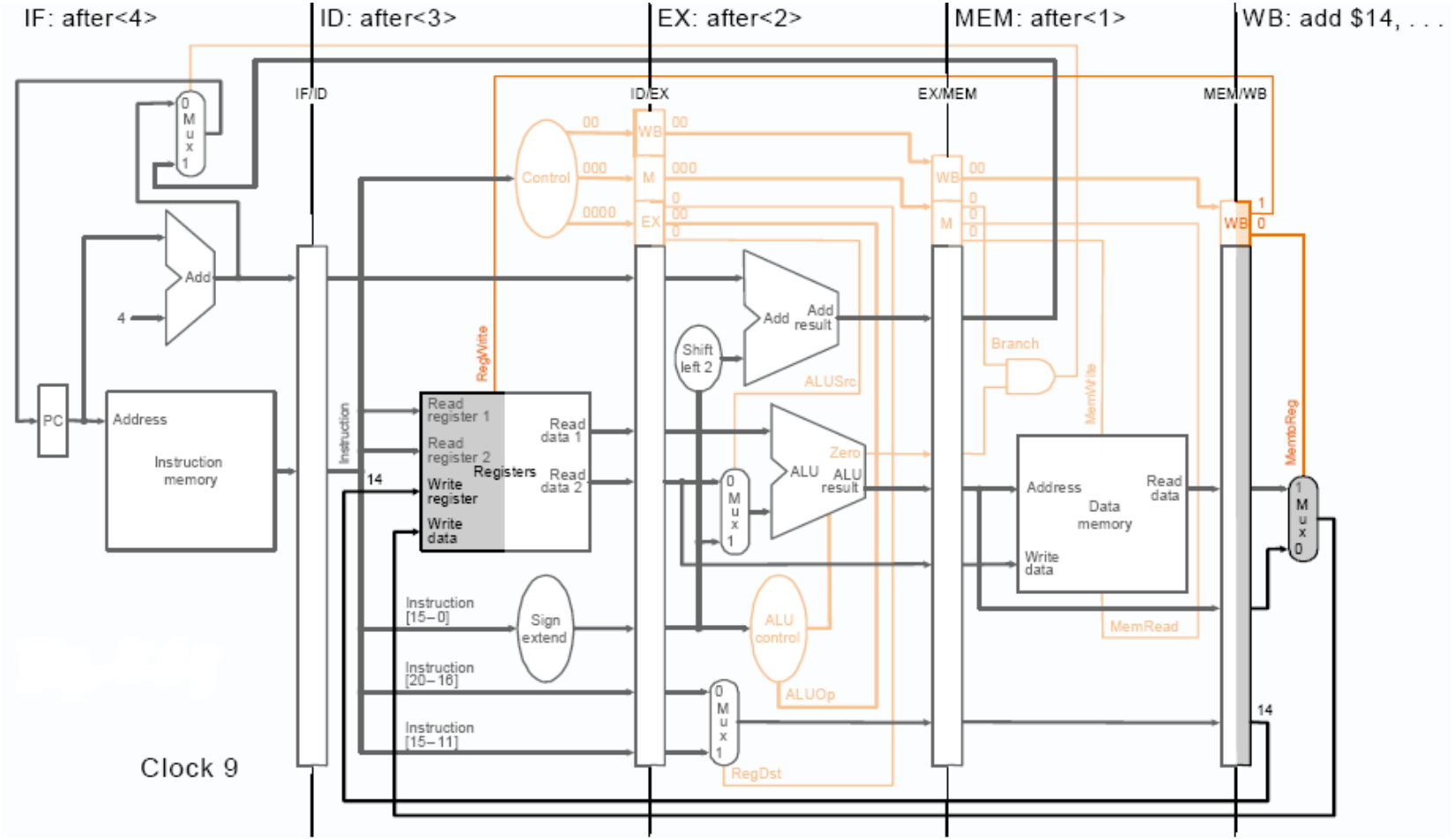
Example 2: Cycle 7



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

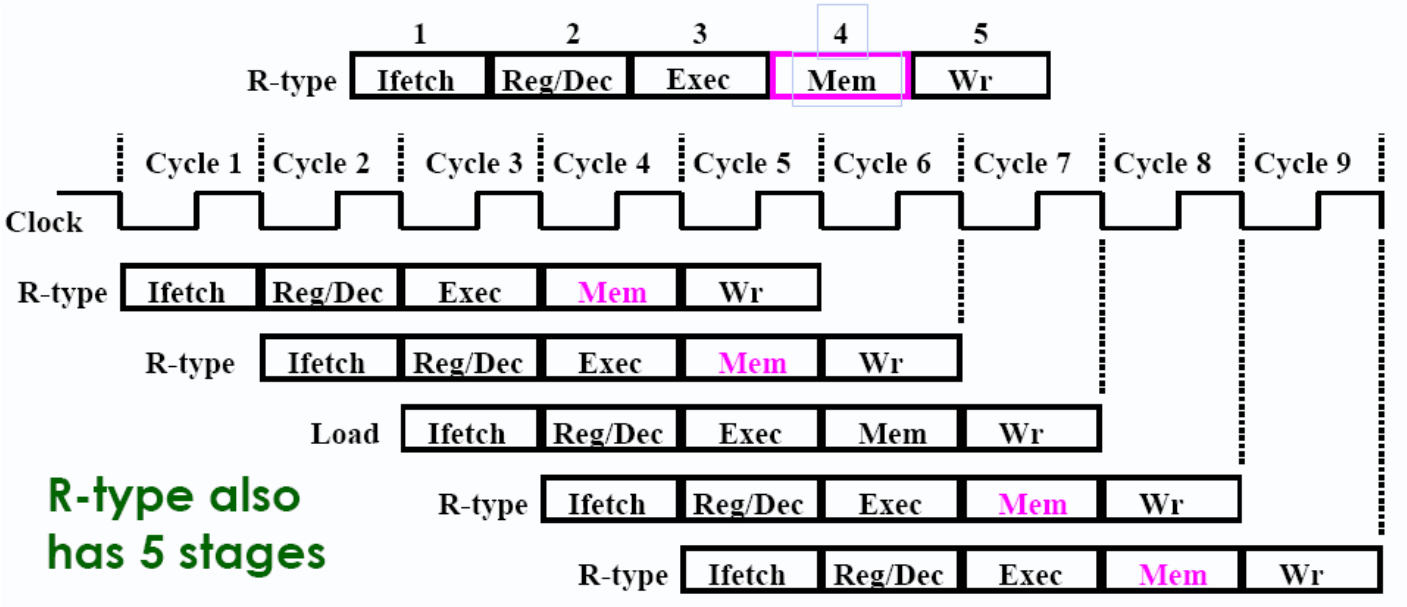
Outline

- 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.

Control hazard

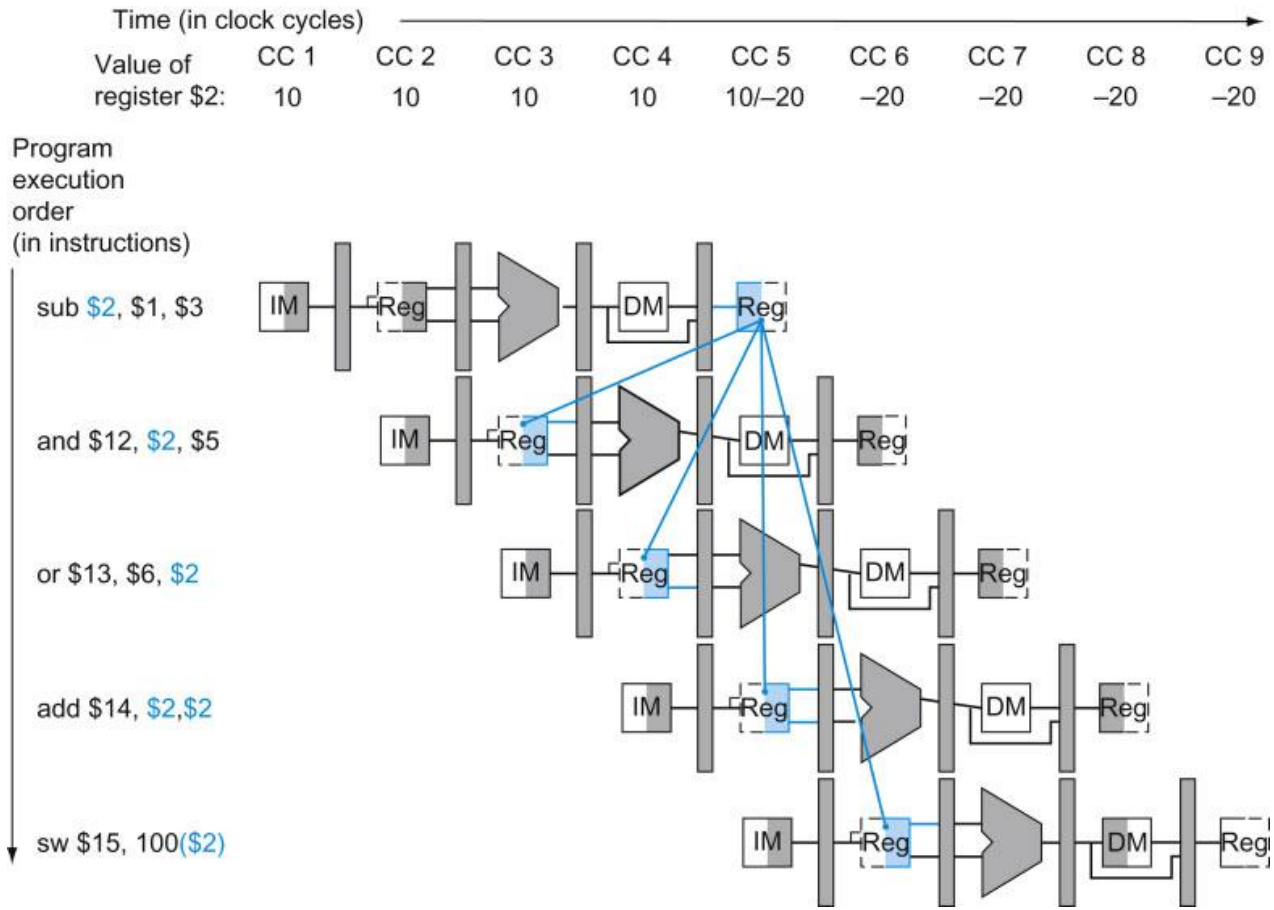
- **Control hazard**
  - Also called **branch hazard**. An occurrence in which the proper instruction cannot execute in the proper clock cycle because the **instruction that was fetched is NOT the one that is needed**; that is, the flow of instruction addresses is not what the pipeline expected **(dynamic).**

- *Untaken branch*
  - One that falls through to the successive instruction. A **taken branch** is one that causes transfer to the branch target

- *Solutions: Branch prediction*
  - A method of resolving a branch hazard that **assumes a given outcome for the branch**, and proceeds from that assumption rather than waiting to ascertain the actual outcome

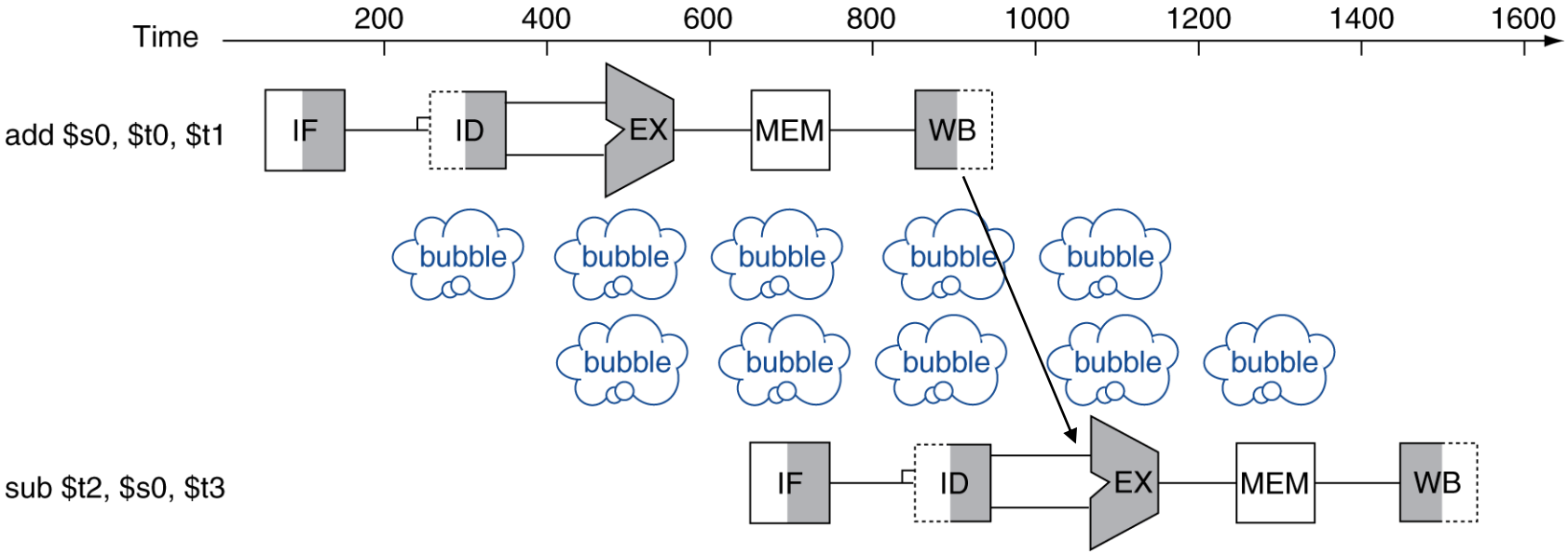
Data Hazards

- Order of operand accesses changed by pipeline (Fig. 5.52)
  - Starting next instruction before first is finished
  - Dependencies "go backward in time"



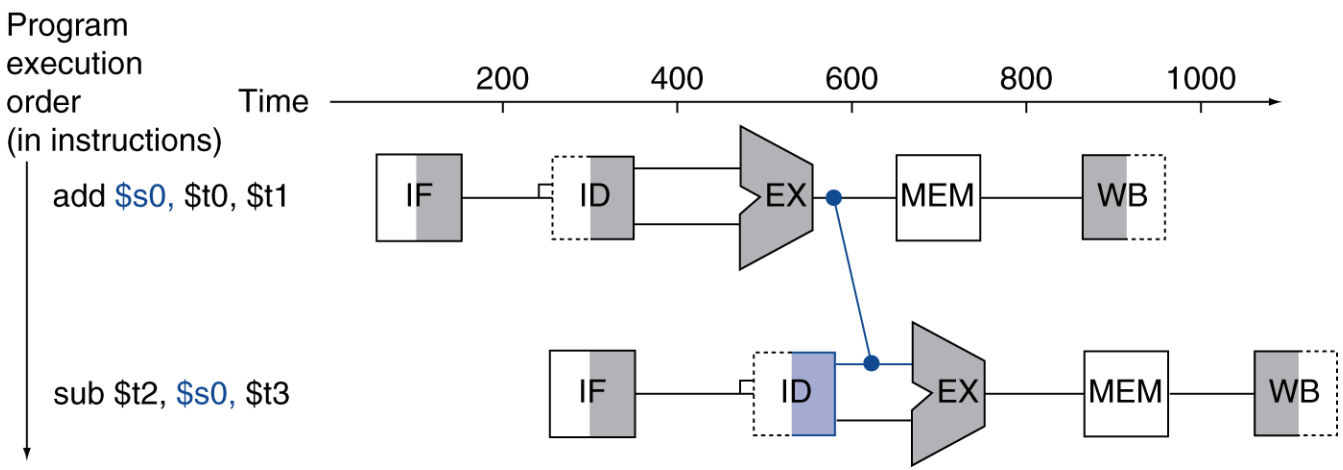
Data Hazards

- 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



Handling Data Hazards

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

Software Solution

- Speculation – remove dependency by **reordering**
- Before

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add     $t3, $t1, $t2      % load-use data hazard
sw      $t3, 12($t0)
lw      $t4, 8($t0)
add     $t5, $t1, $t4
sw      $t5, 16($t0)
```

- After

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
lw      $t4, 8($t0)
add     $t3, $t1, $t2
sw      $t3, 12($t0)
add     $t5, $t1, $t2
sw      $t5, 16($t0)
```

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`

stall

stall

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1,$t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1,$t4
sw   $t5, 16($t0)
```

13 cycles

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1,$t2
sw   $t3, 12($t0)
add  $t5, $t1,$t4
sw   $t5, 16($t0)
```

11 cycles

Software Solution

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

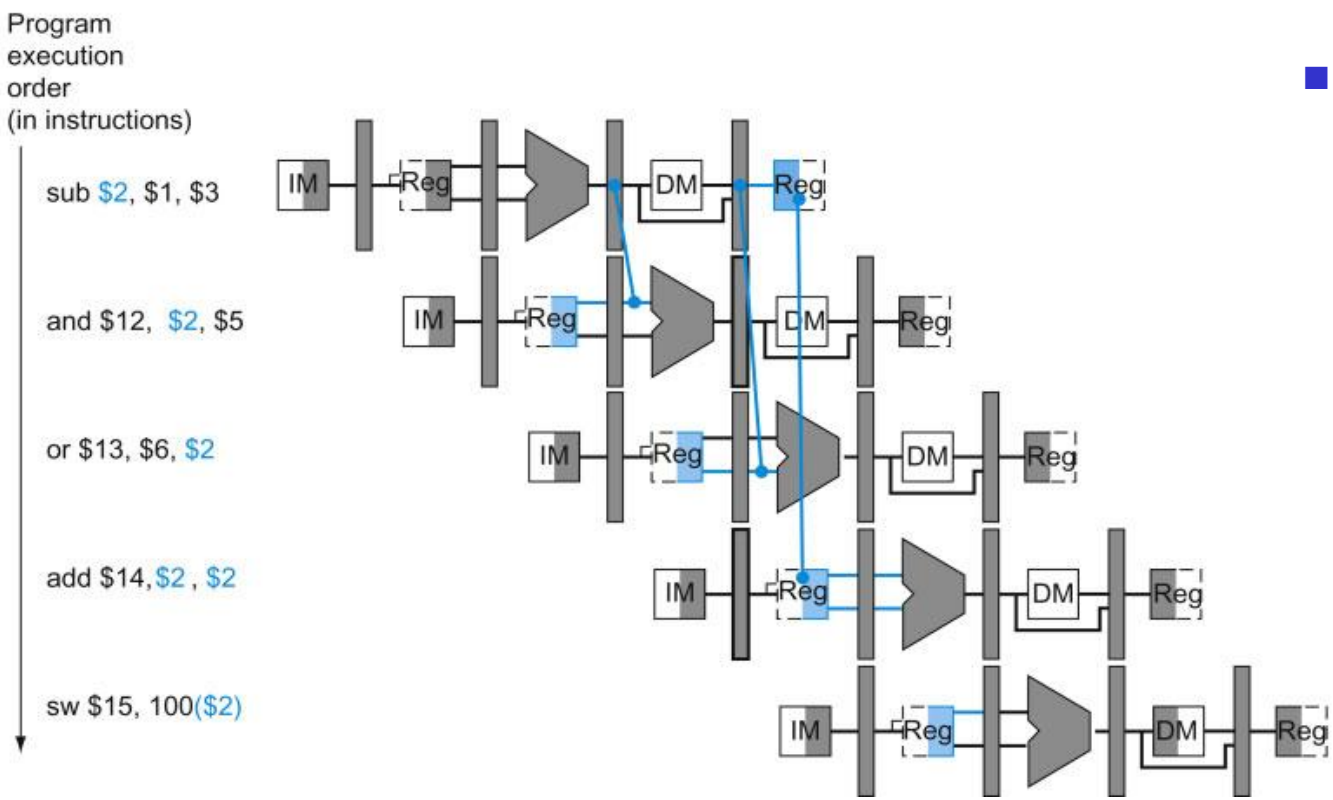
```
sub   $2, $1, $3          % two NOPs
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

- Problem: not efficient enough!

Resolving Hazards in Hardware: Forwarding (bypassing)

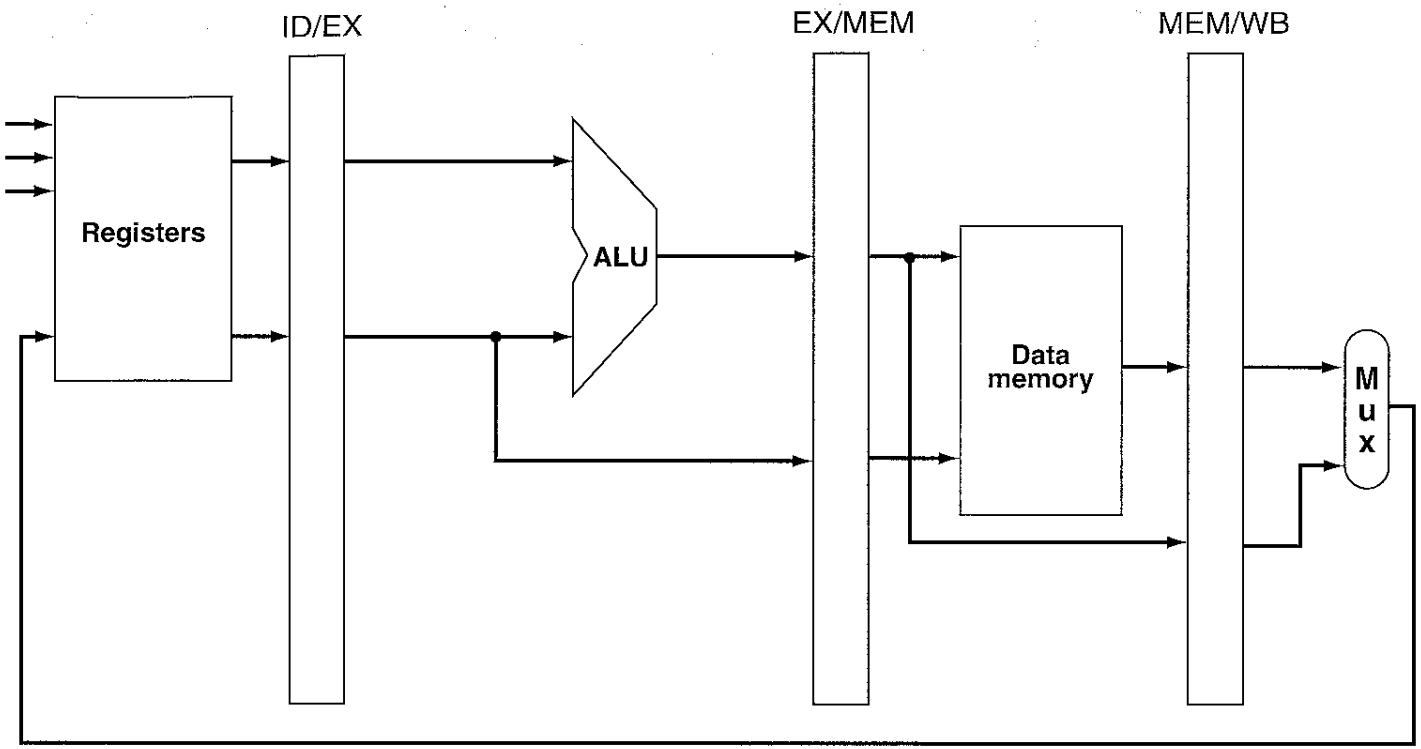
Time (in clock cycles) →

	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



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

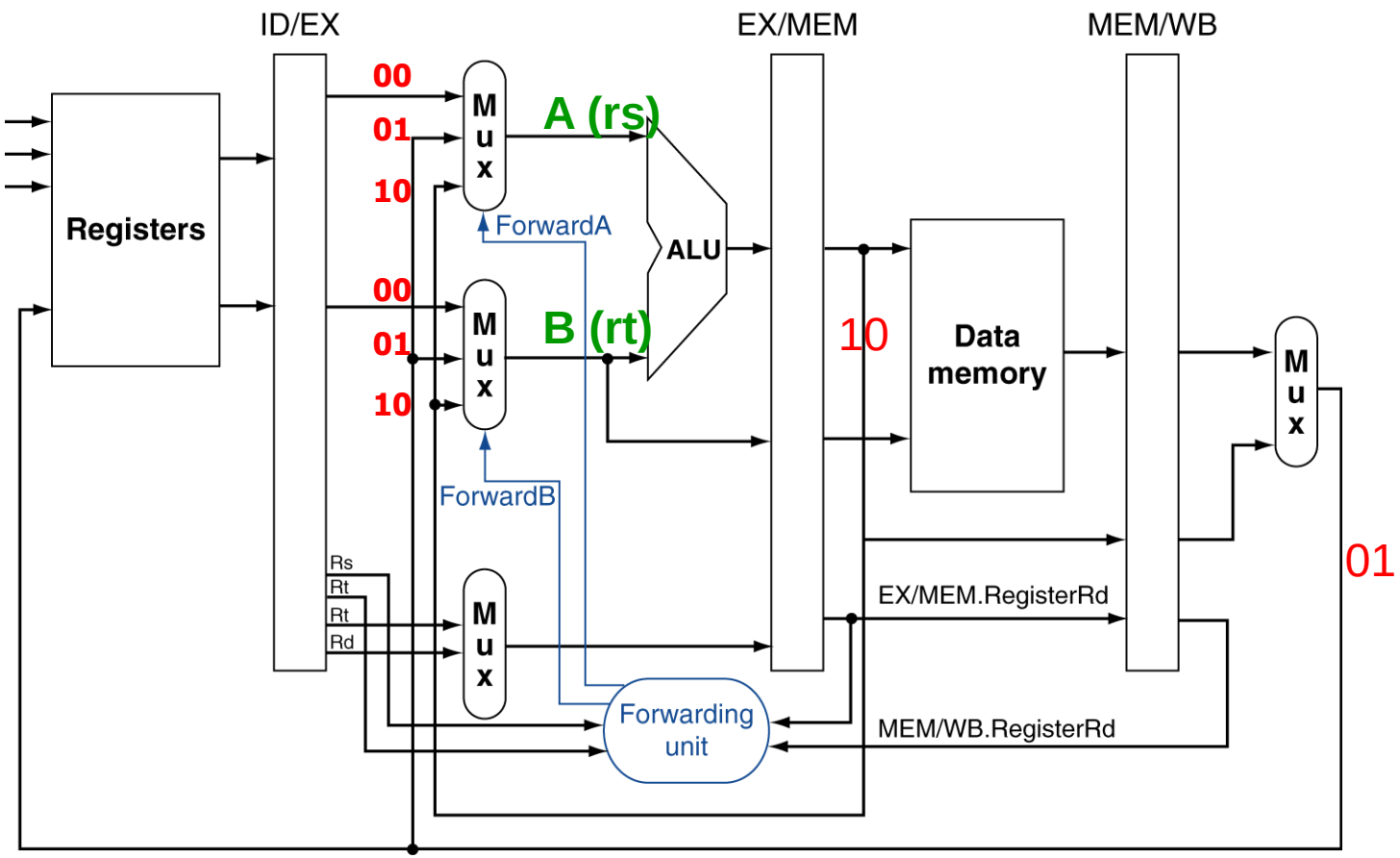
No Forwarding



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

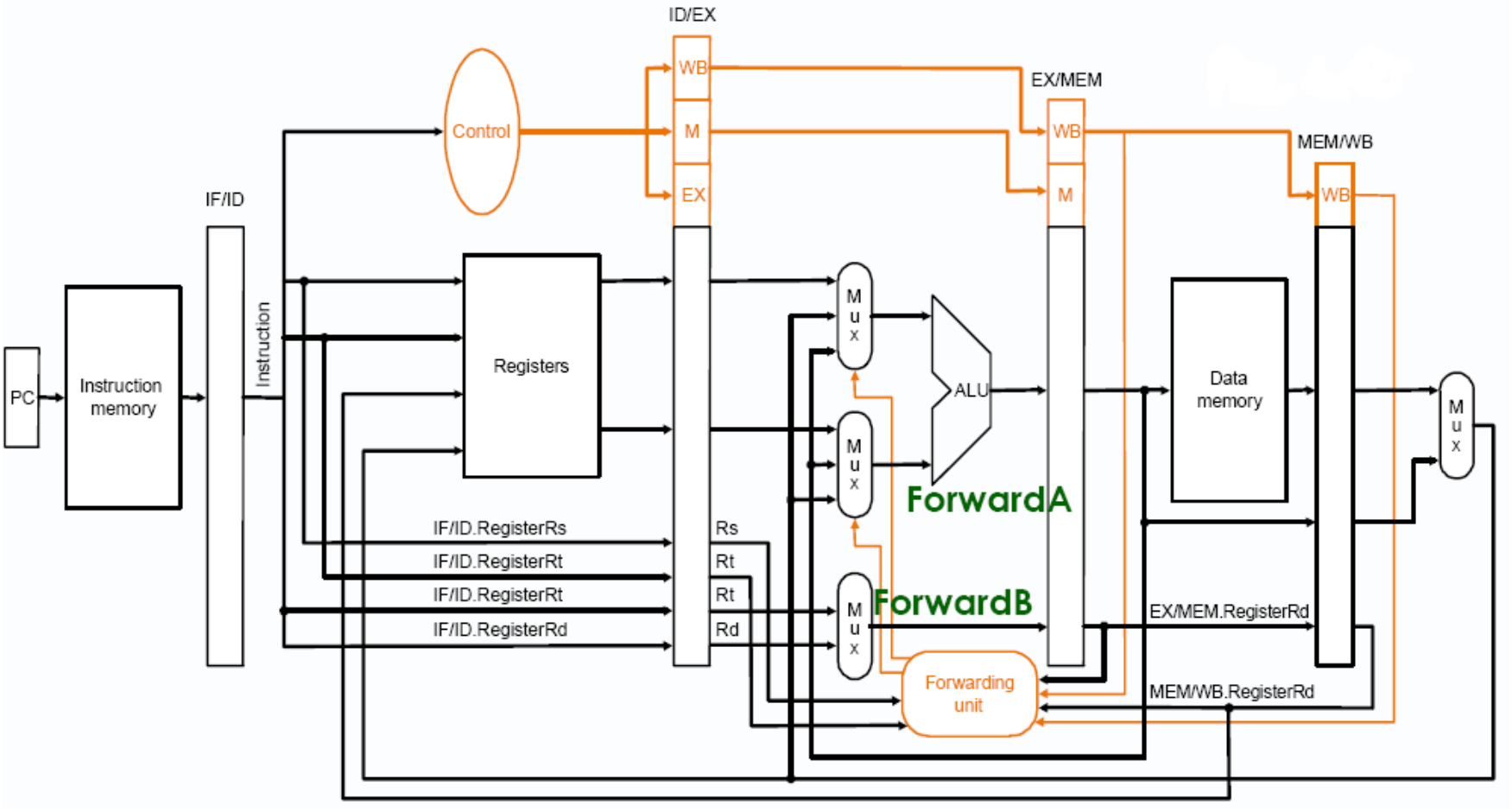
Detecting Data Hazards (cont.)

- 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)

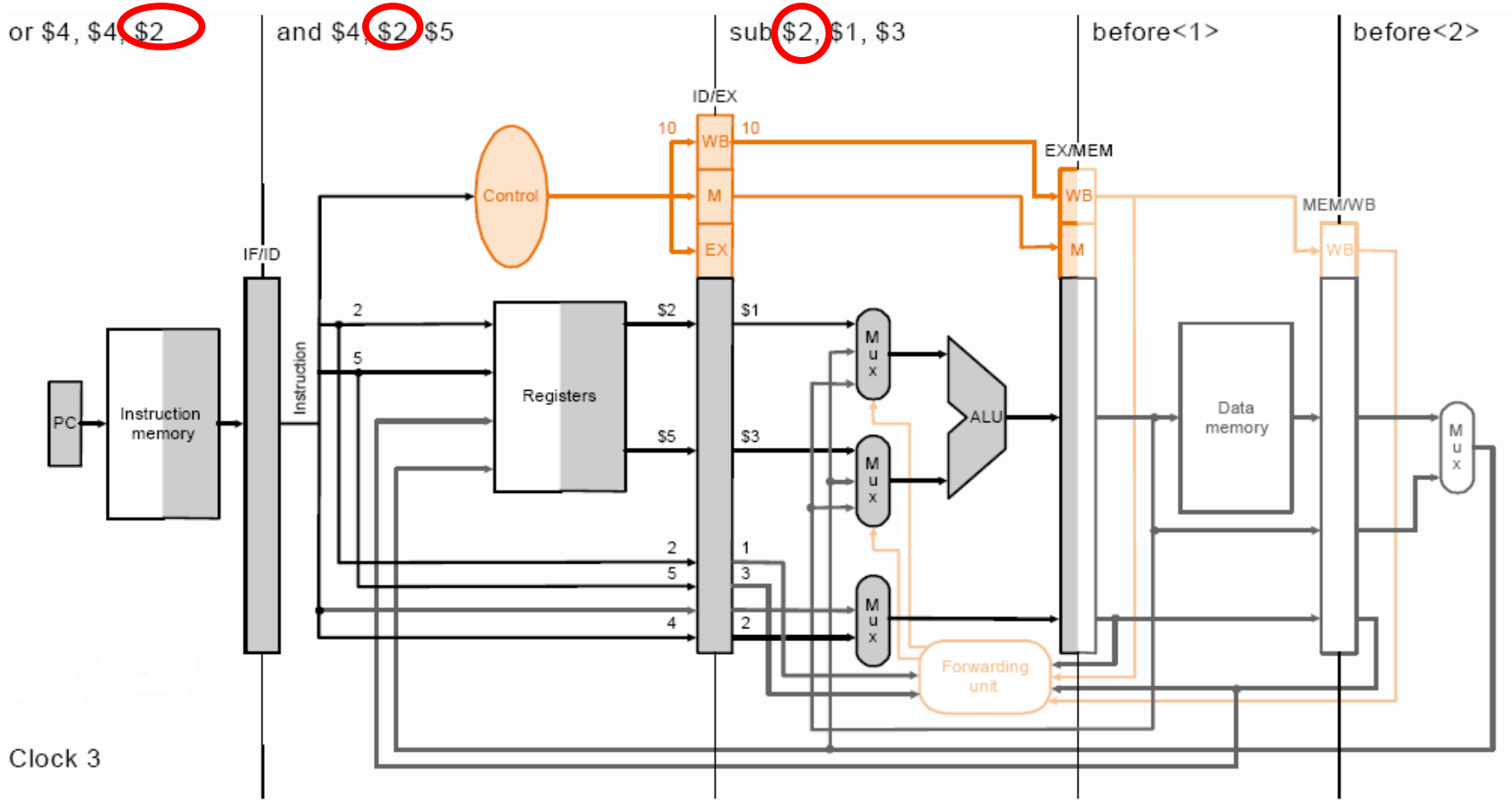
    ForwardA = 01
  - If (MEM/WB.RegWrite
    and (MEM/WB.RegRd ≠ 0)
    and (MEM/WB.RegRd = ID/EX.RegRt)

    ForwardB = 01

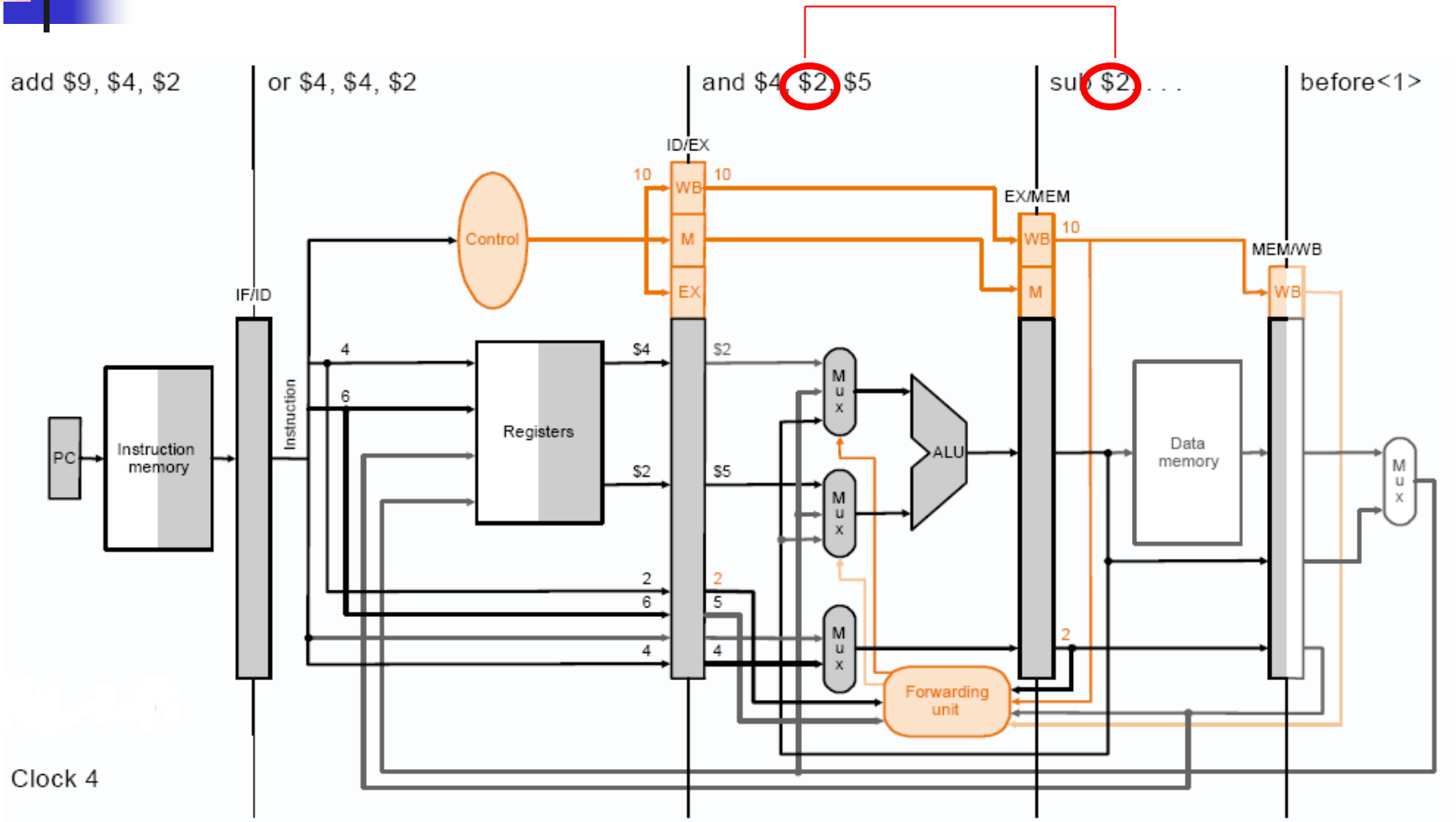
Pipeline with Forwarding



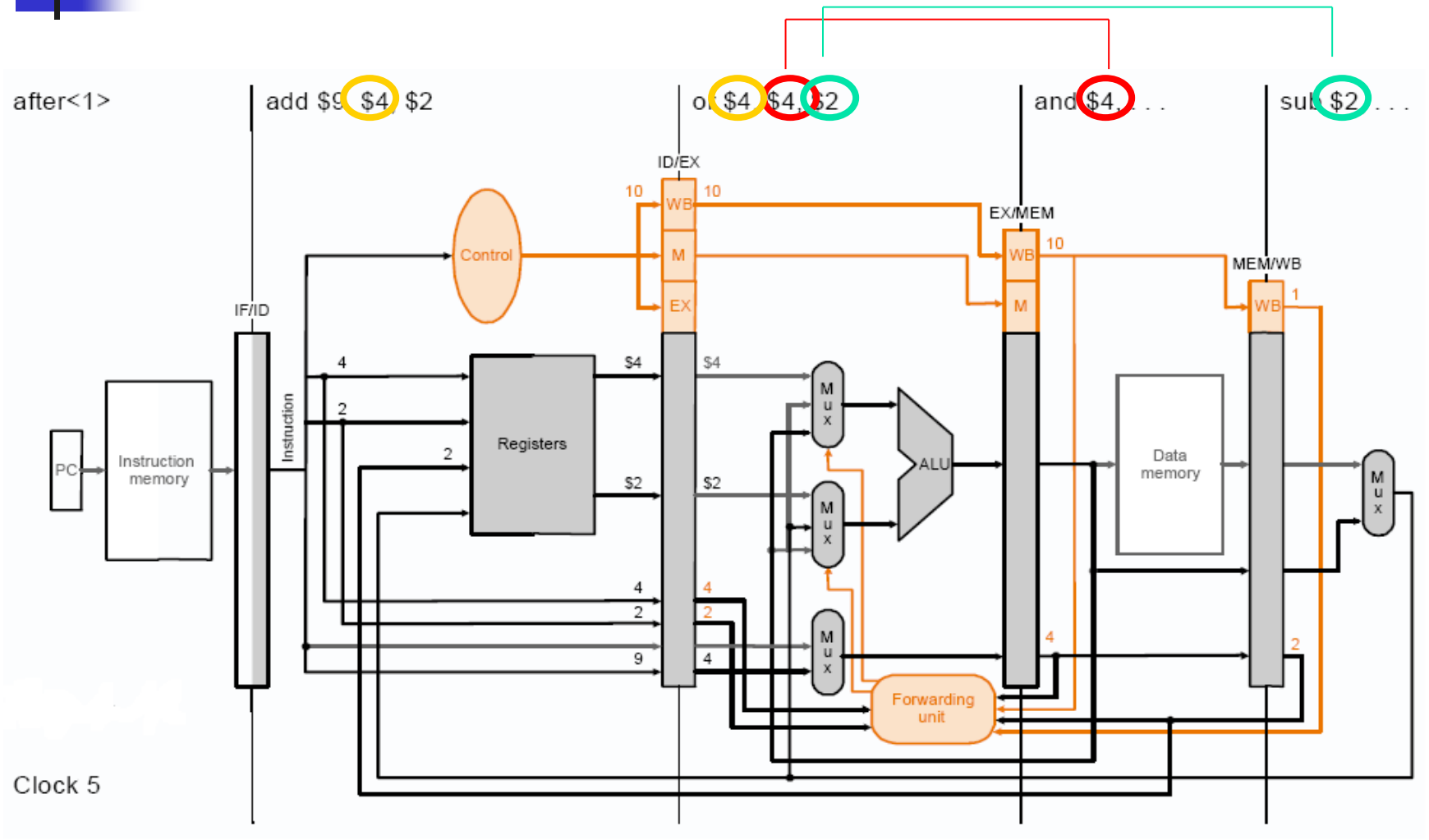
Example 3: Cycle 3



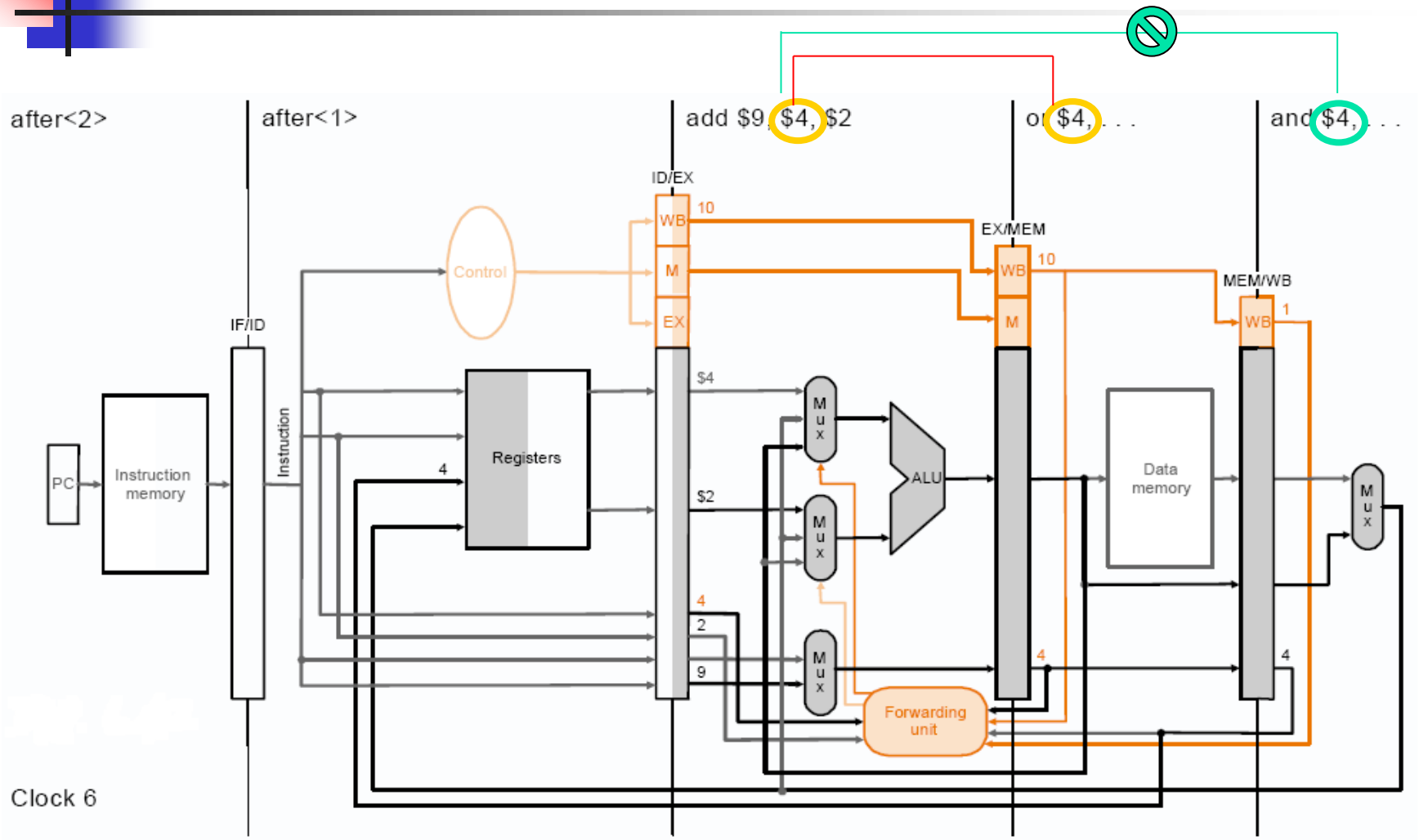
Example 3: Cycle 4



Example 3: Cycle 5

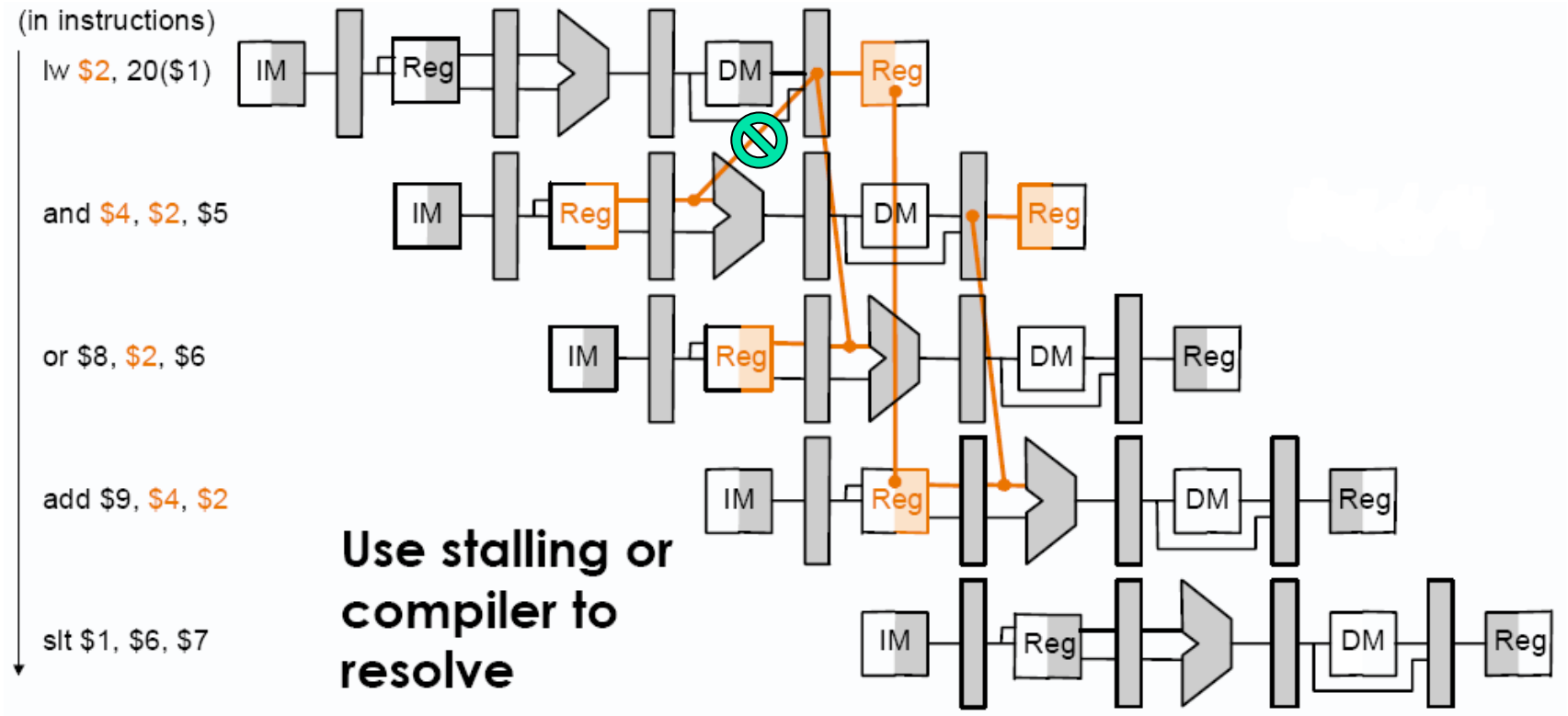


Example 3: Cycle 6

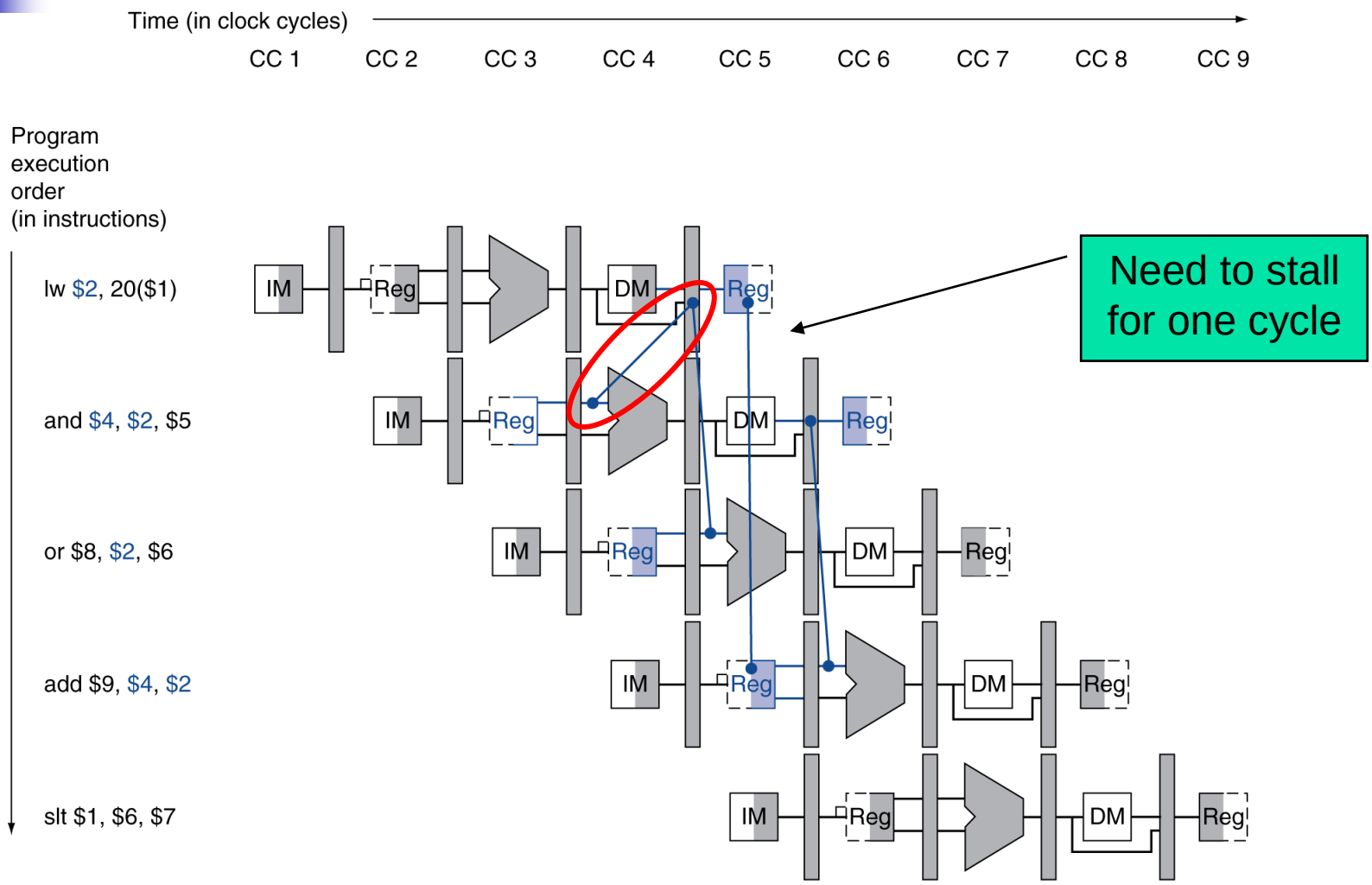


II. Load-Use Data Hazard

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

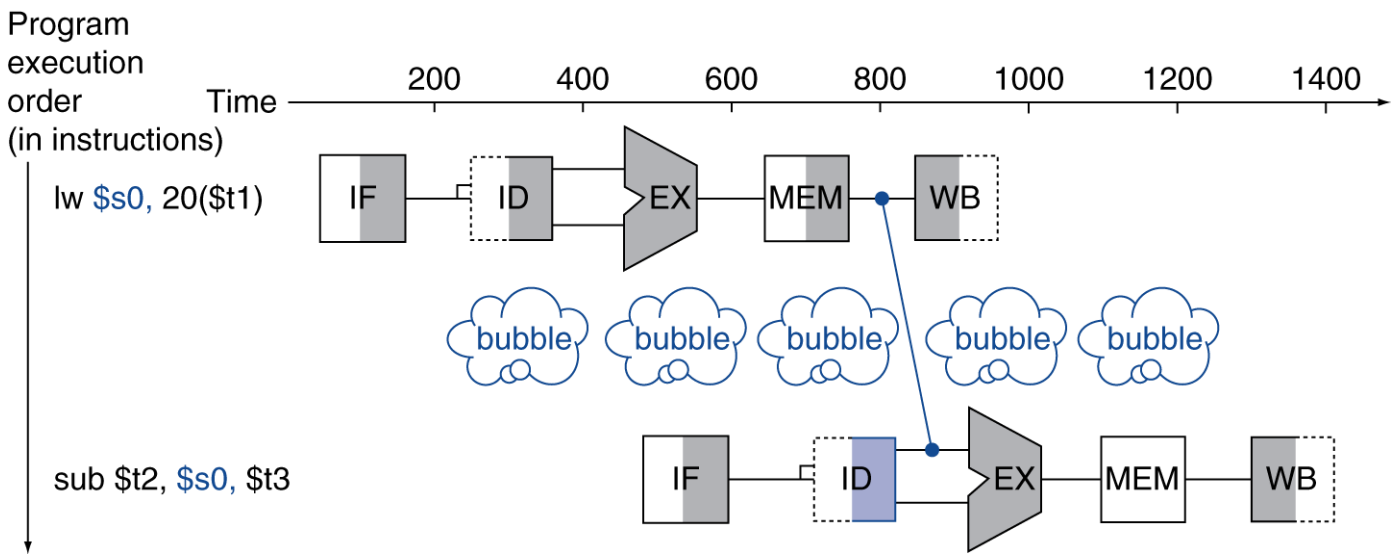


Load-Use Data Hazard



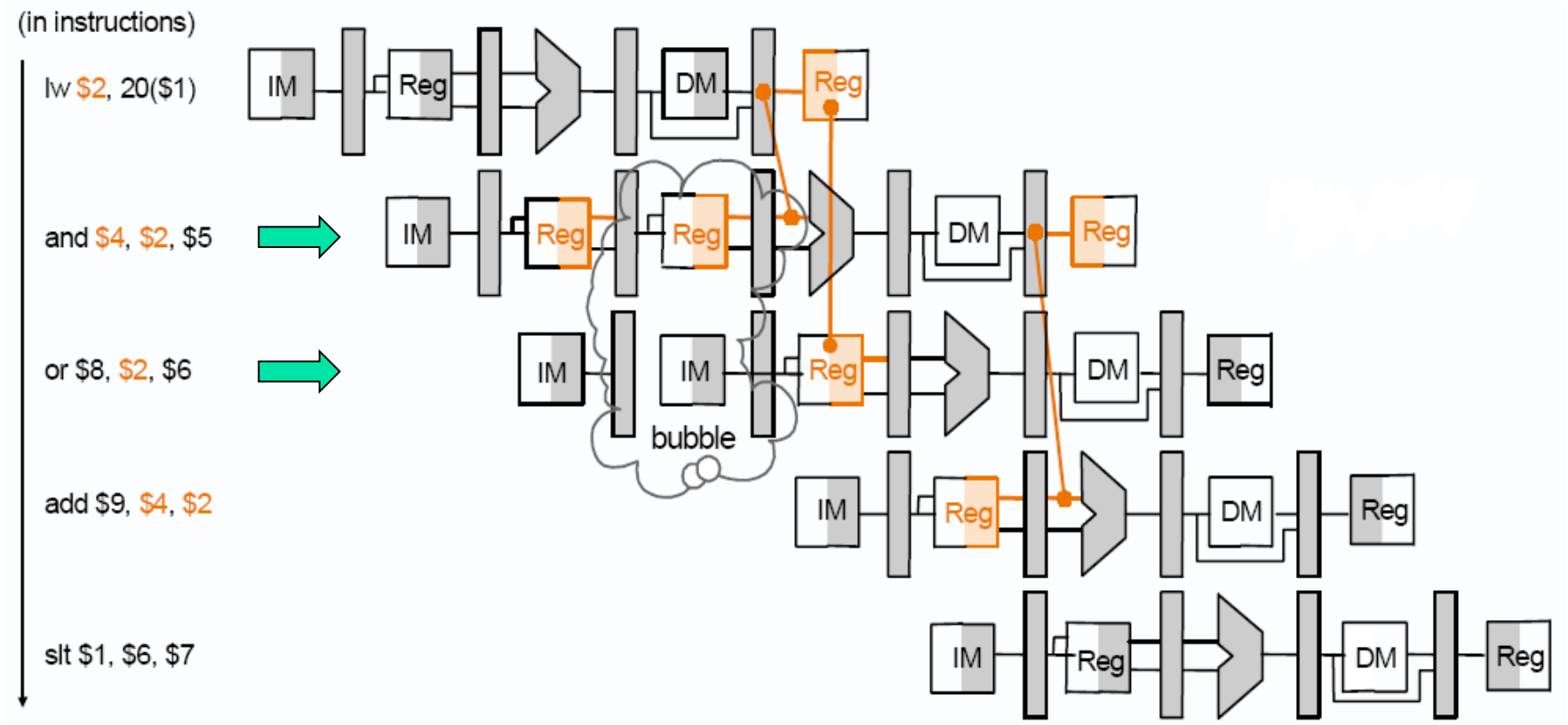
Can't Always Forward

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



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 instruction)*

Handling Stalls

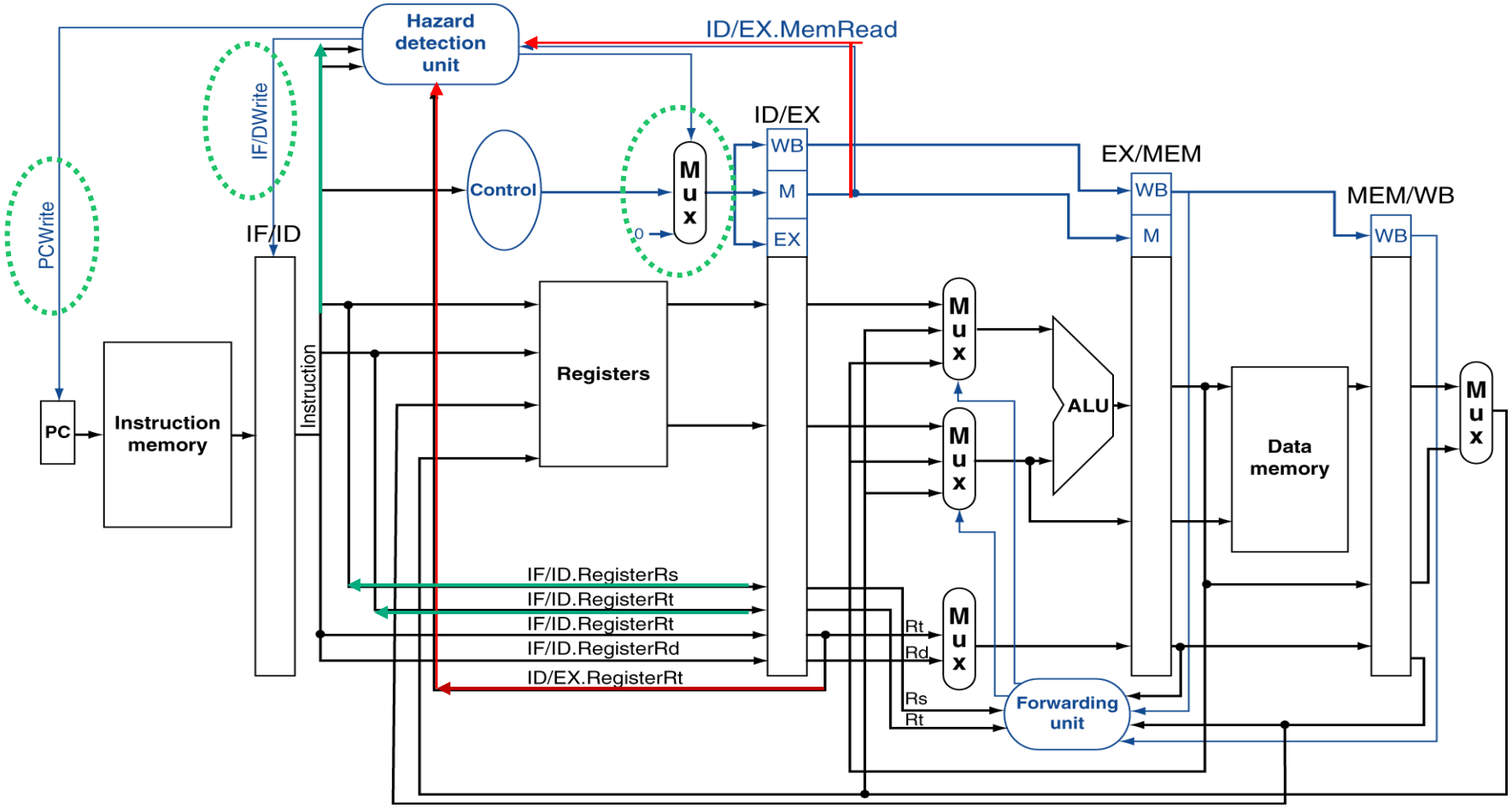
- How to stall?
  - Stall instruction in IF and ID: Don't change **PC** and **IF/ID**

    → 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**

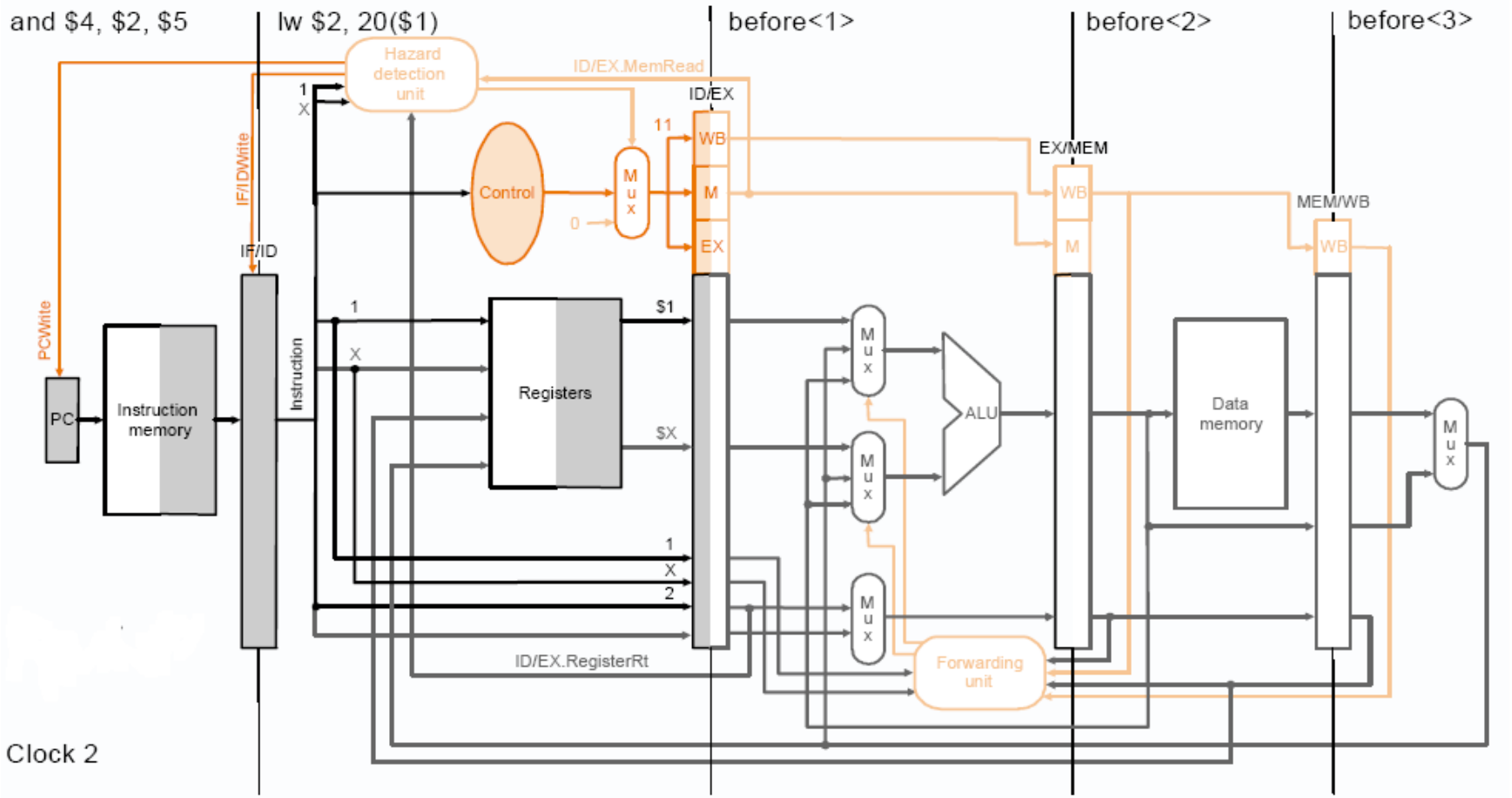
  → As control signals propagate, all control signals to EX, MEM, WB are deasserted and no registers or memories are written

Pipeline with Stalling Unit

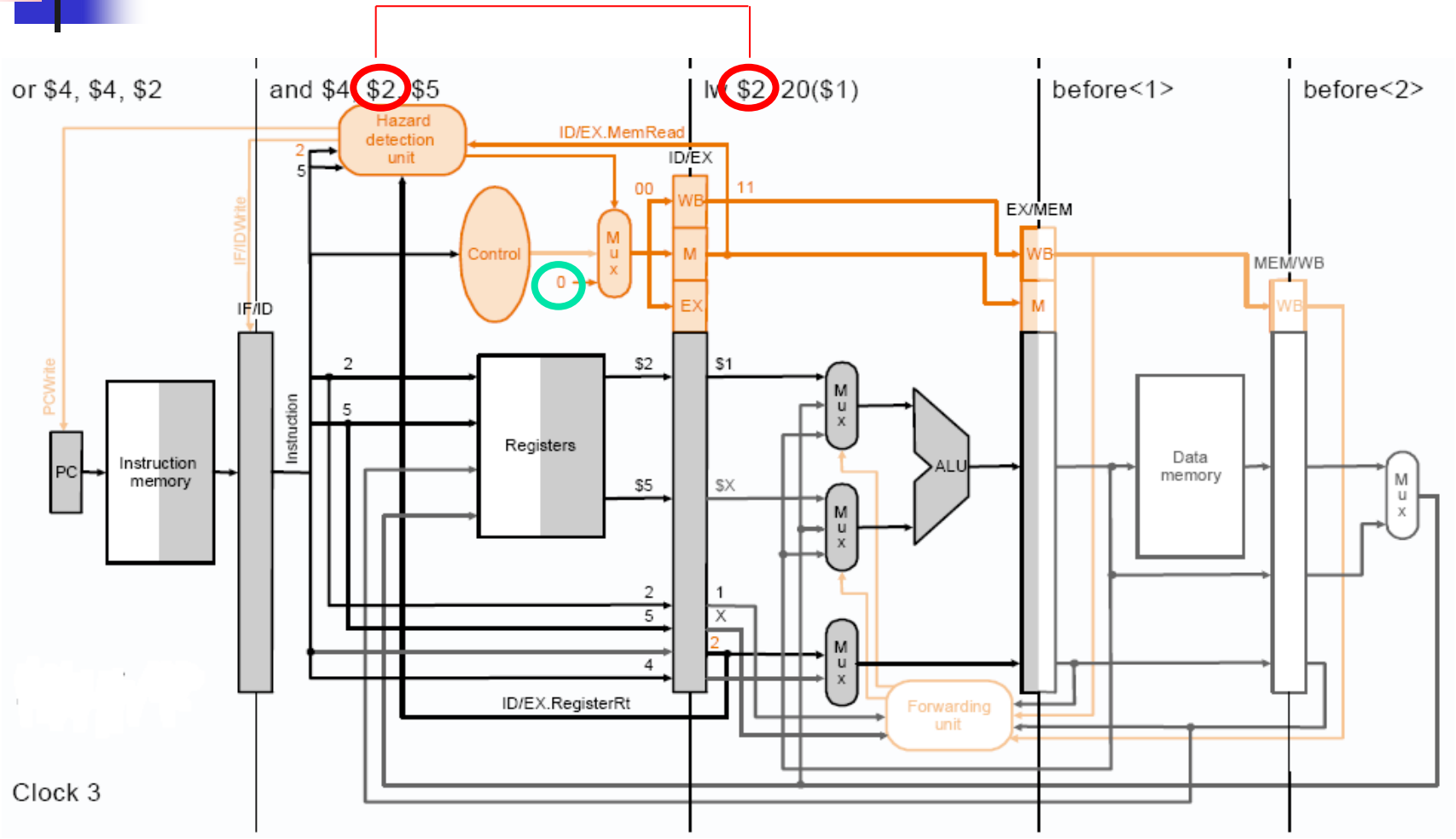
- Forwarding controls ALU inputs; hazard detection controls PC, IF/ID, and control signals



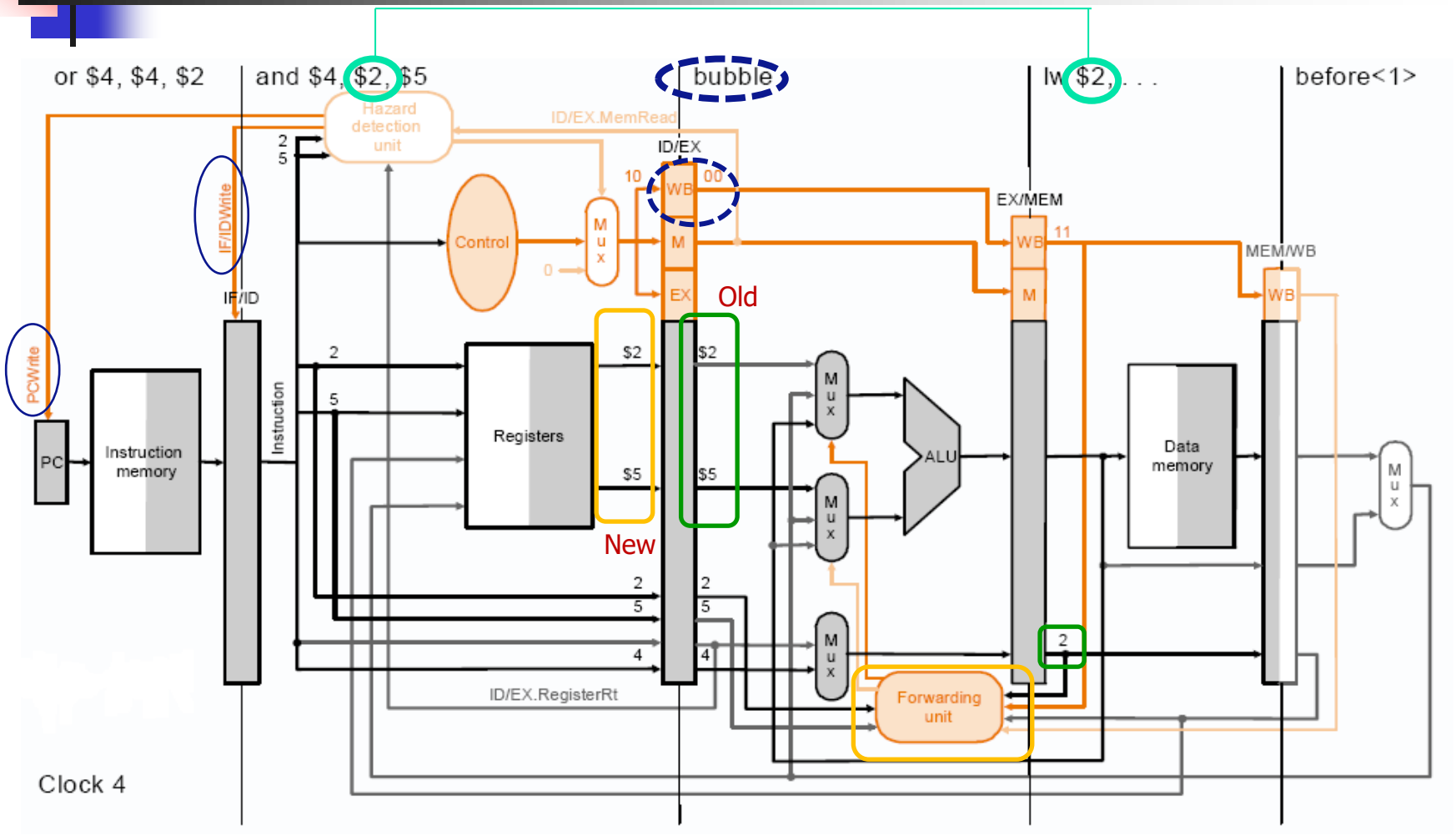
Example 4: Cycle 2



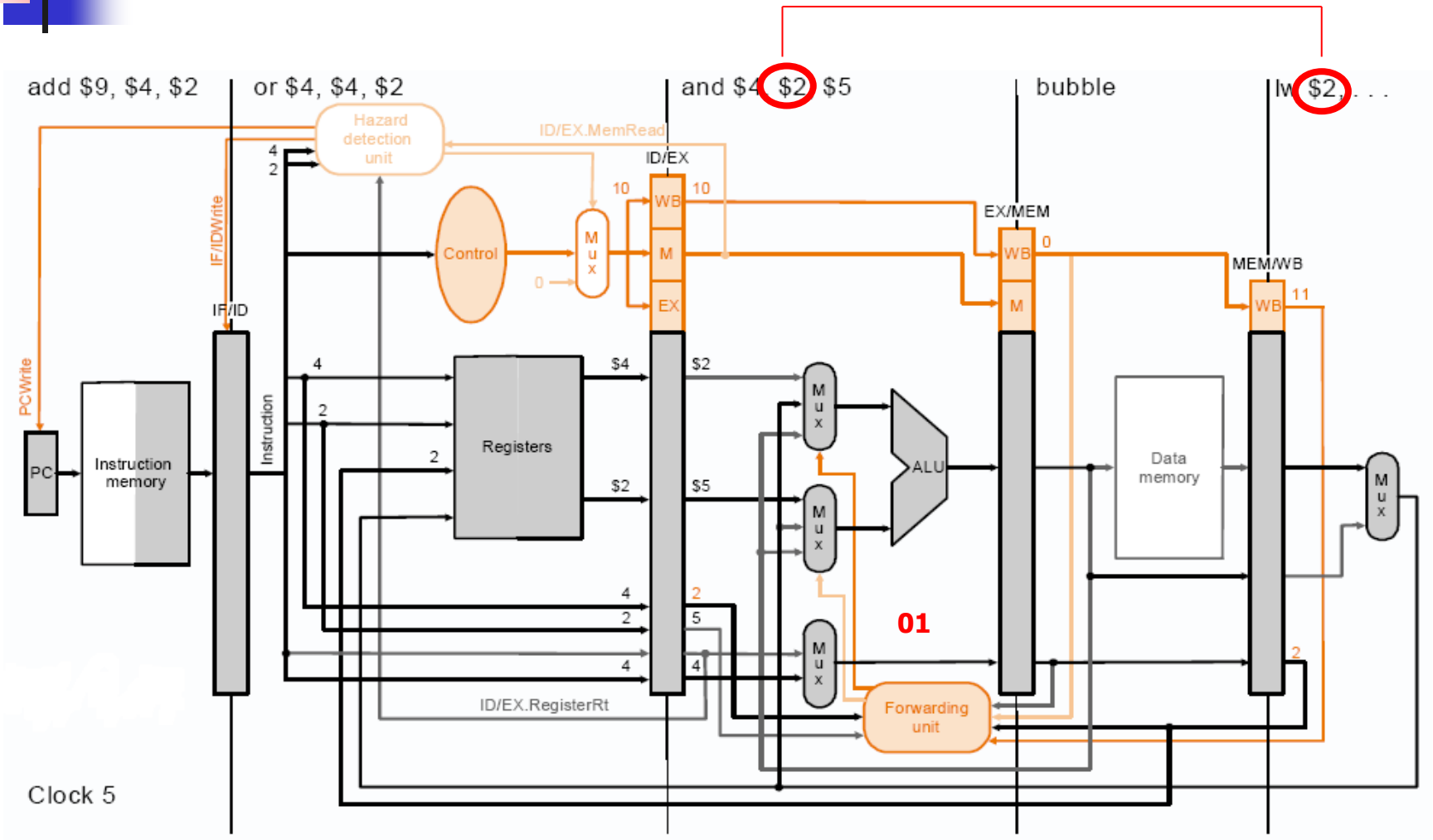
Example 4: Cycle 3



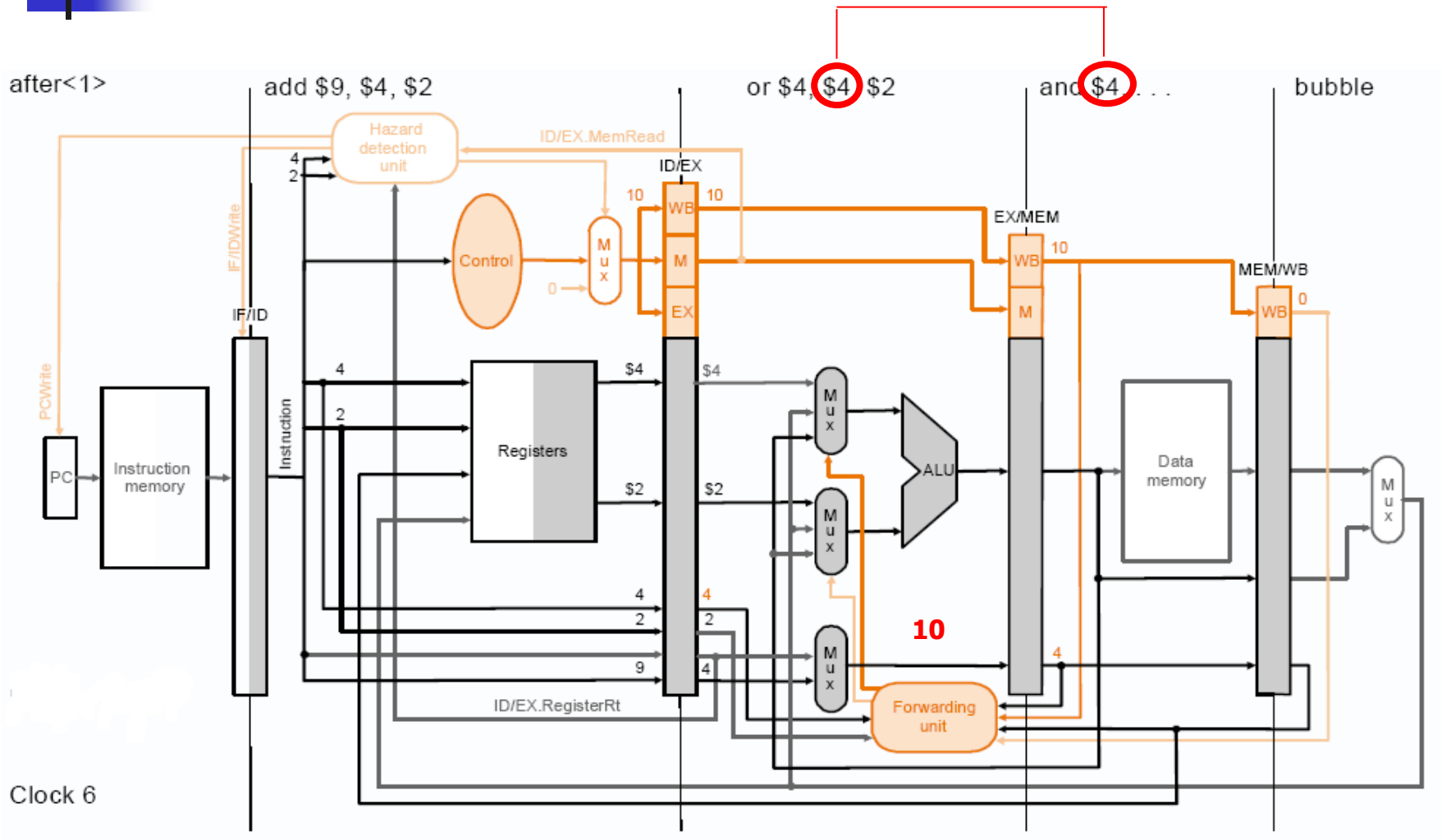
Example 4: Cycle 4



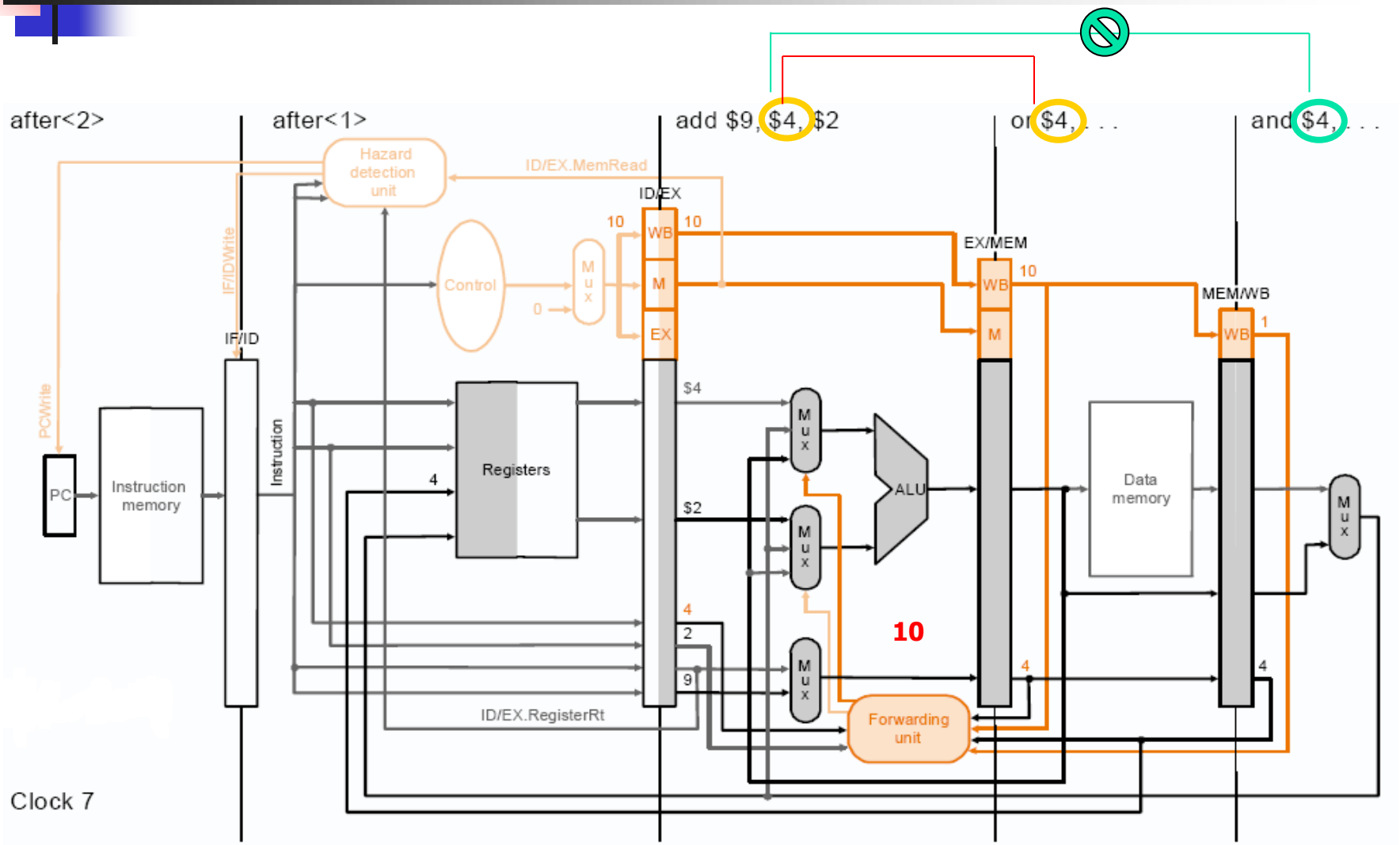
Example 4: Cycle 5



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

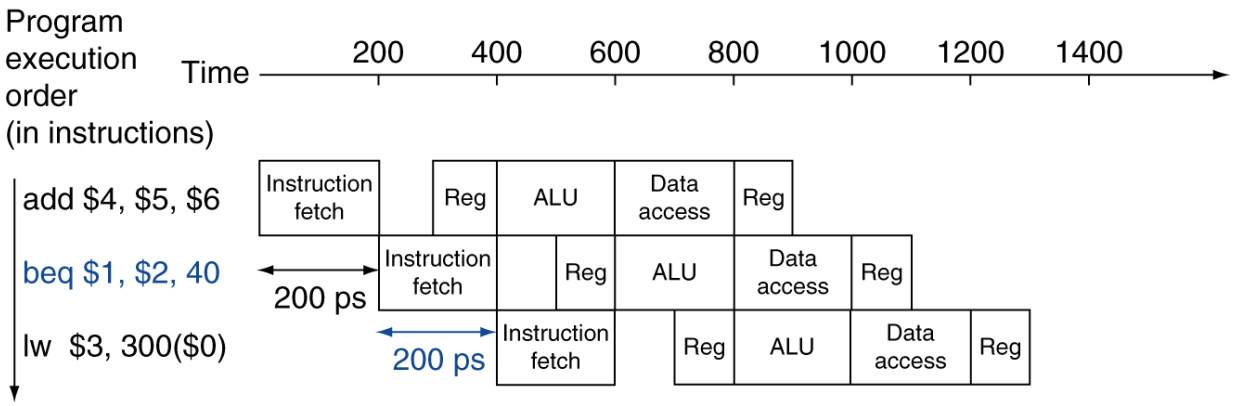
Control Hazards

- 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

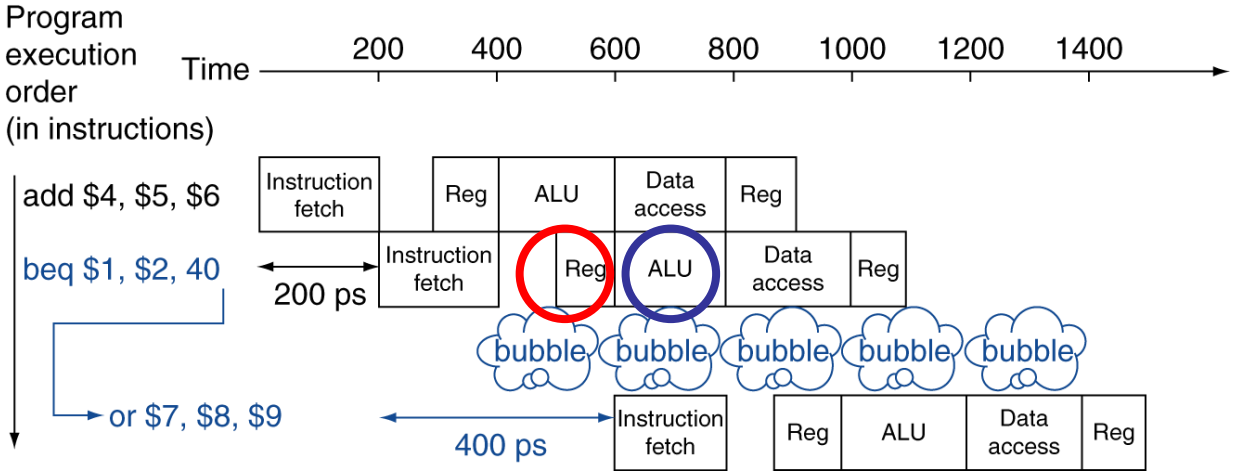
Stall on Branch

Wait until branch outcome determined before fetching next instruction

Branch not taken

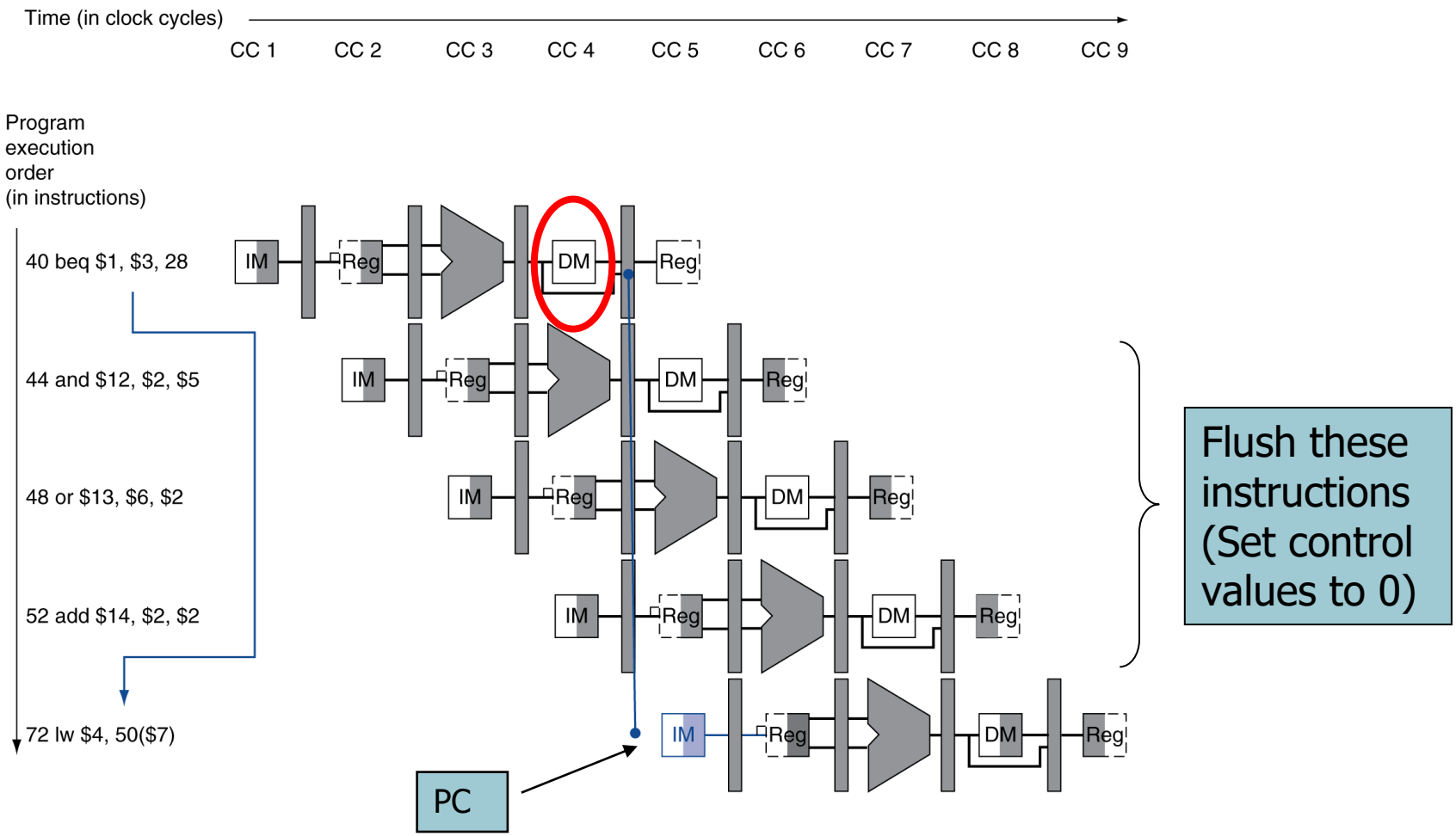


Branch taken



Branch Hazards

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



Handling Branch Hazard

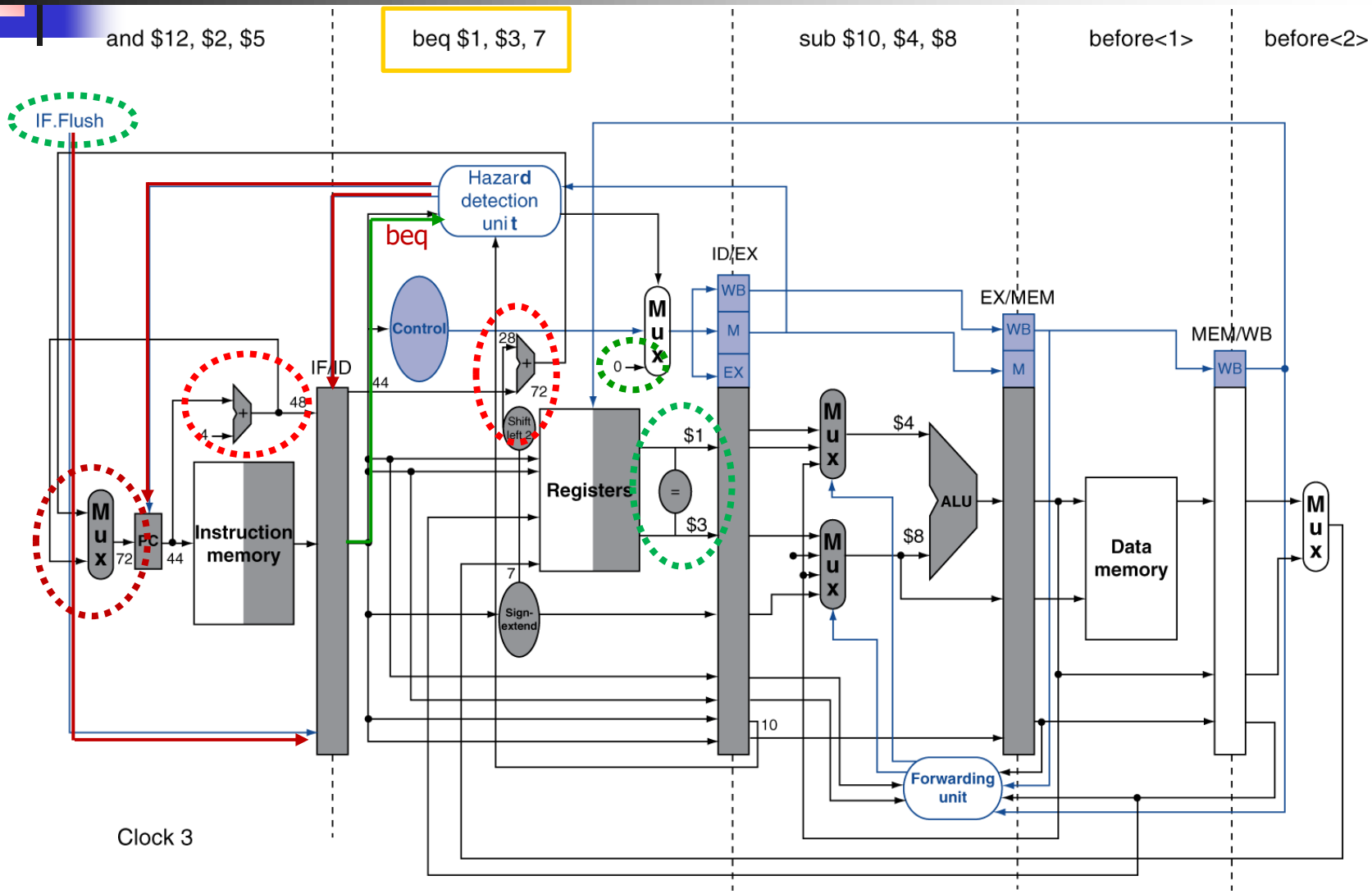
- Predict branch always not taken
  - Need to add hardware for flushing inst. if wrong
  - Branch decision made at MEM stage → need to flush inst. in IF, ID, EX by changing control values to 0
- Reduce delay of taken branch by moving branch execution **earlier** in the pipeline
  - Move up branch address 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)

Reducing Branch Delay

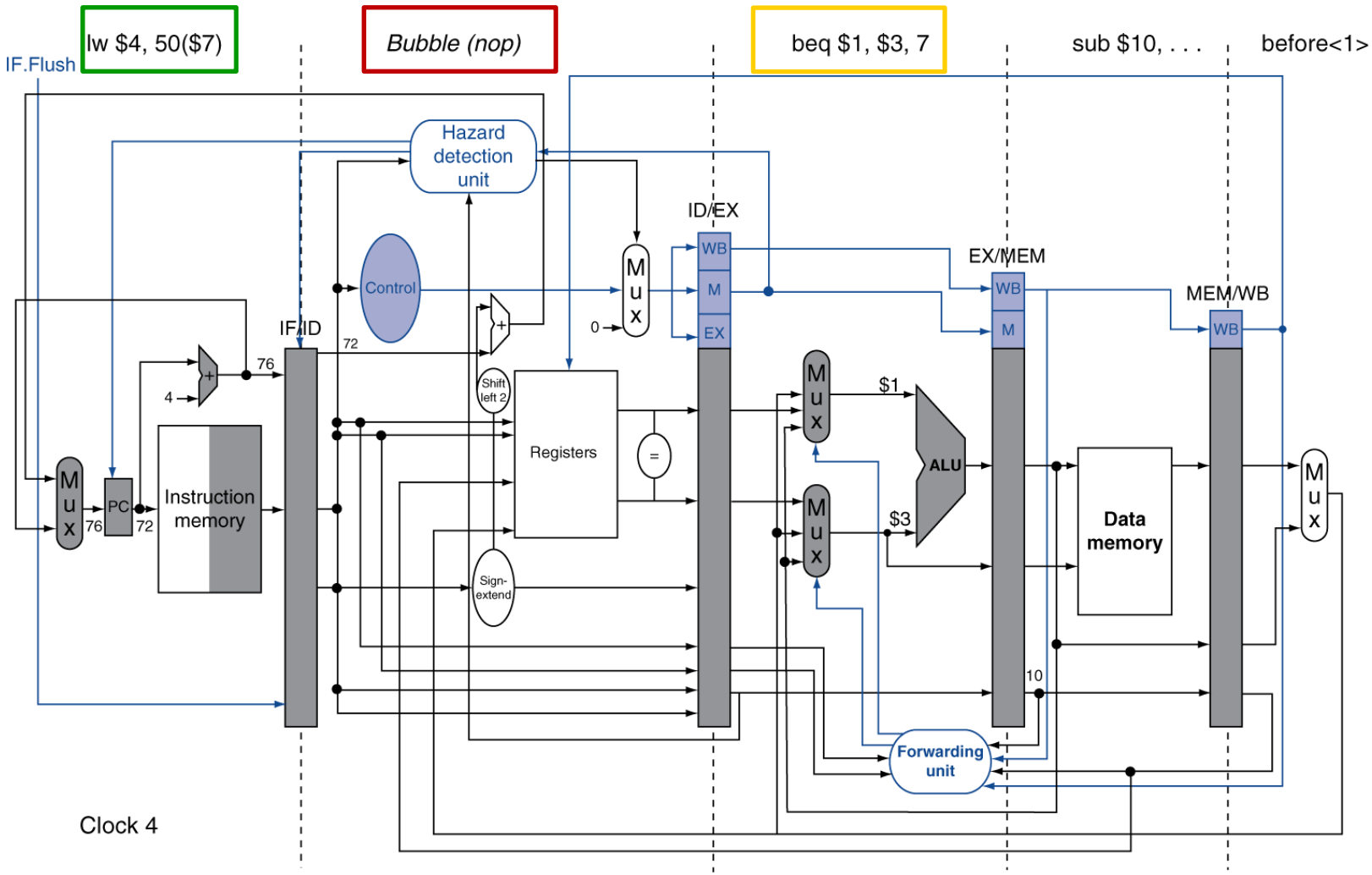
- Move hardware to determine outcome to **ID stage**
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 7
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
     ...
72:  lw   $4, 50($7)
```

Example: Branch Taken



Example: Branch Taken

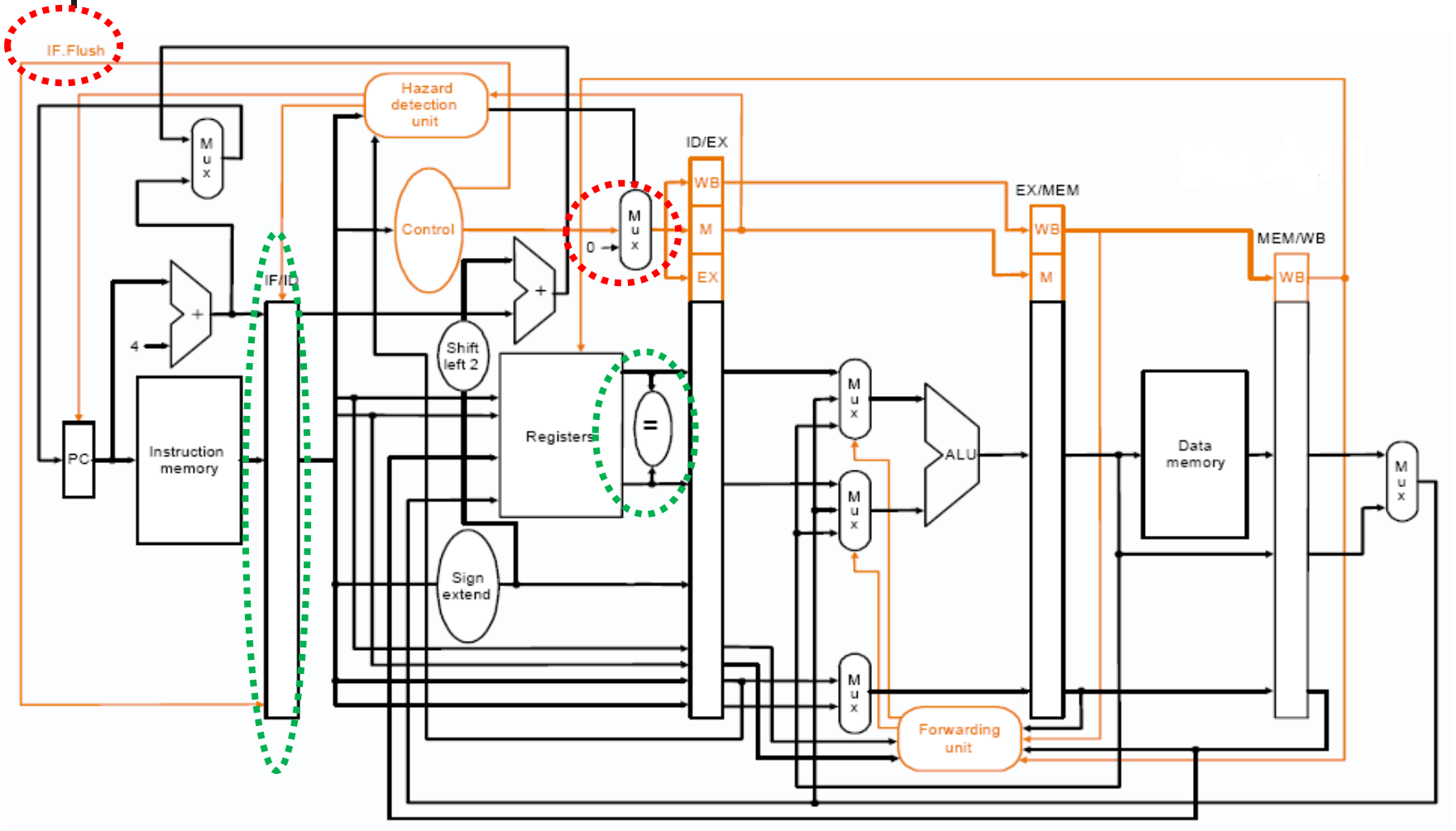


Reducing Branch Delay

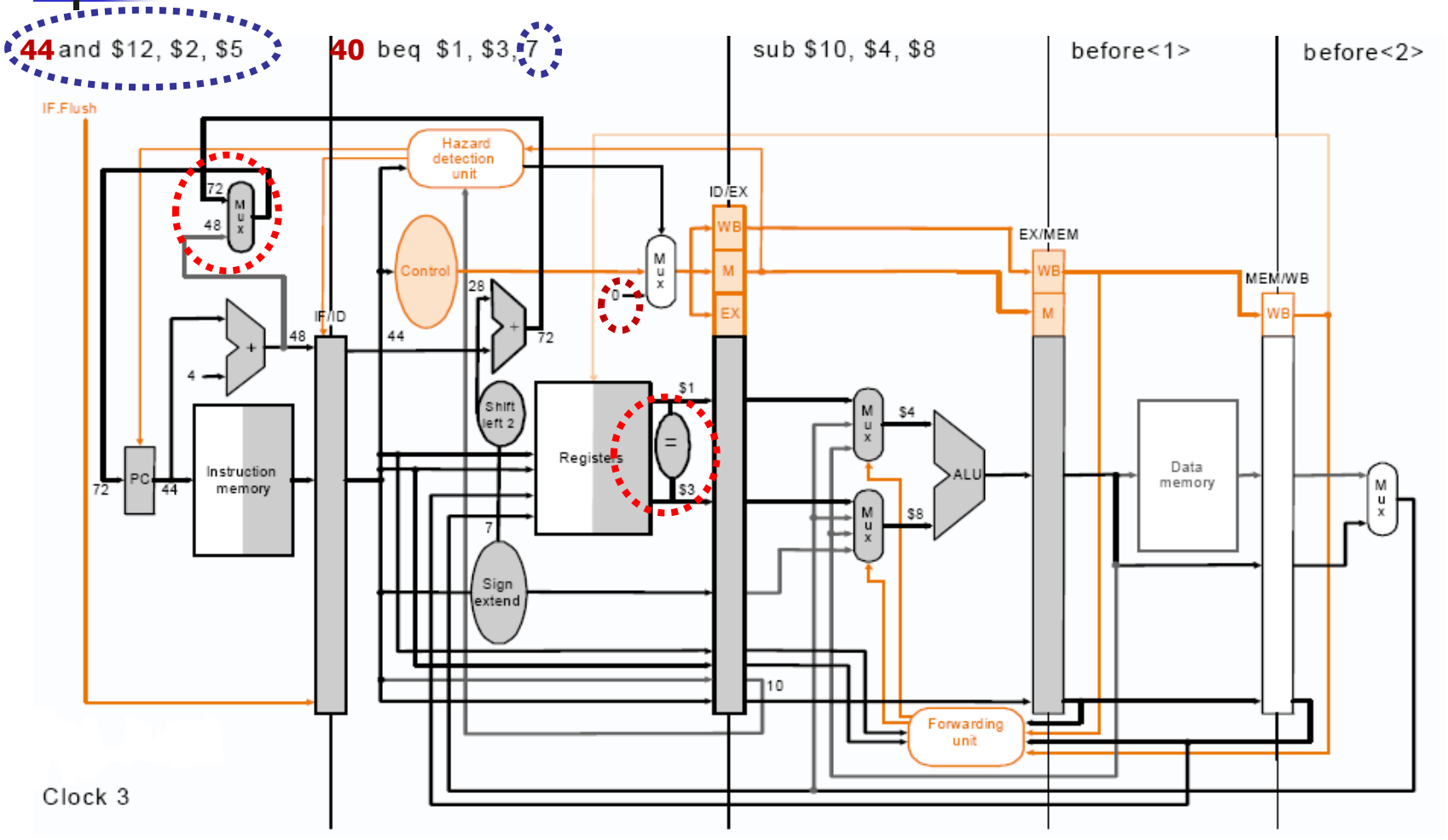
- Move hardware to determine outcome to **ID stage**
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 7
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
     ...
72:  lw   $4, 50($7)
```

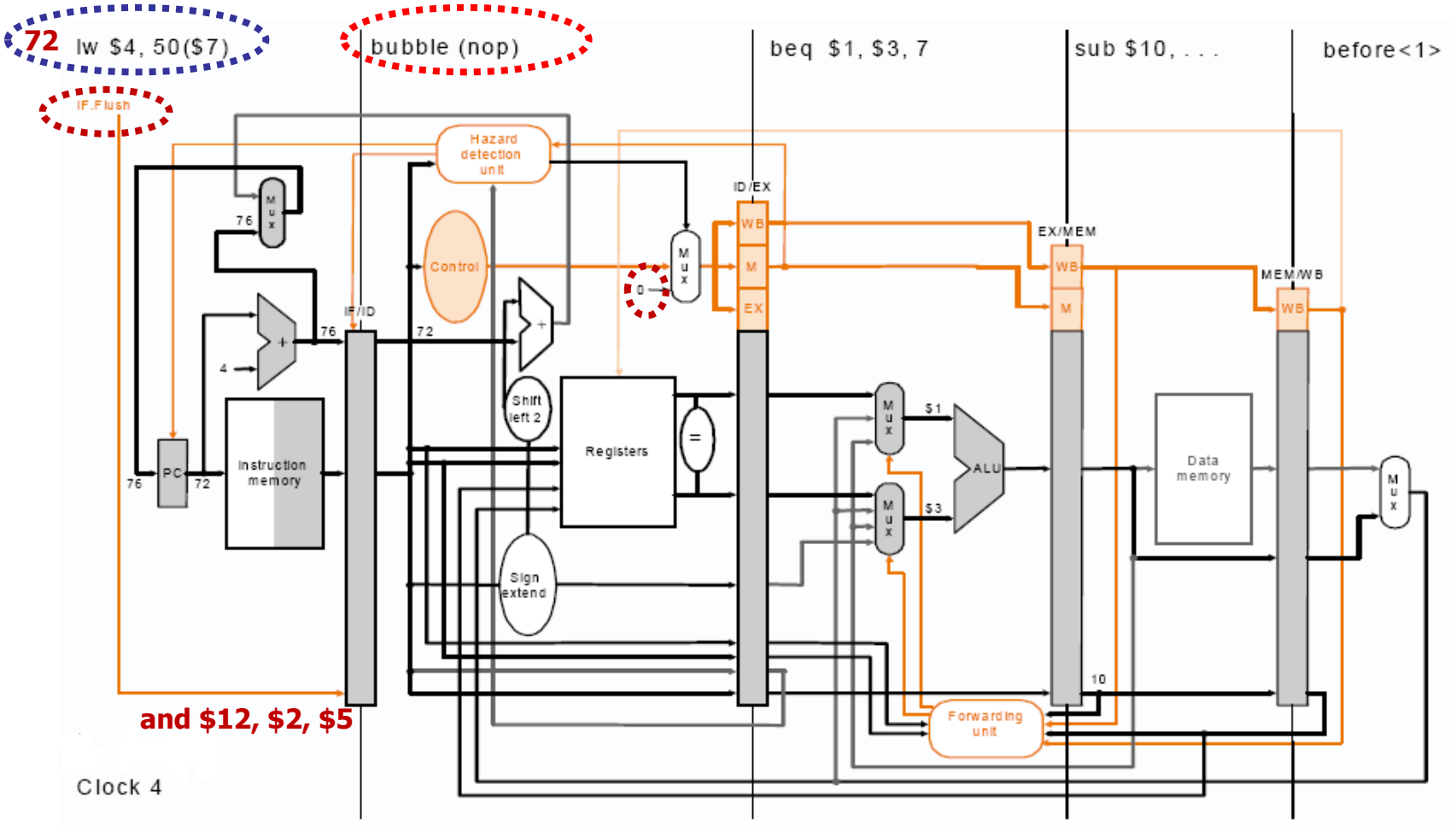
Another example: Pipeline with Flushing



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

1-Bit Predictor: Shortcoming

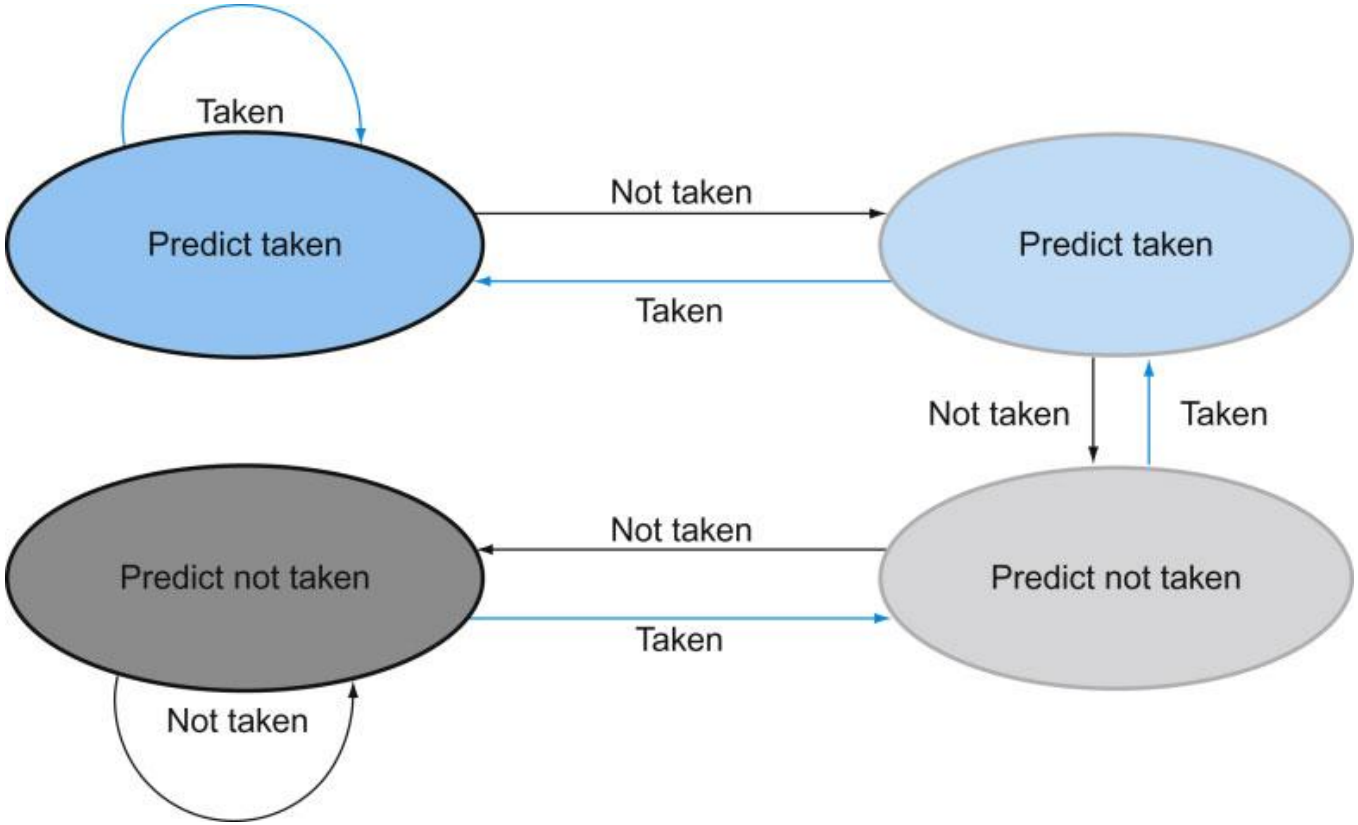
- Inner loop branches mispredicted twice!

```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

  - Mispredict as *not taken* on first iteration of inner loop (initial branch flag = False, not taken)
  - Mispredict as *taken* on last iteration of inner loop

2-Bit Predictor

- Only change prediction on two successive mispredictions



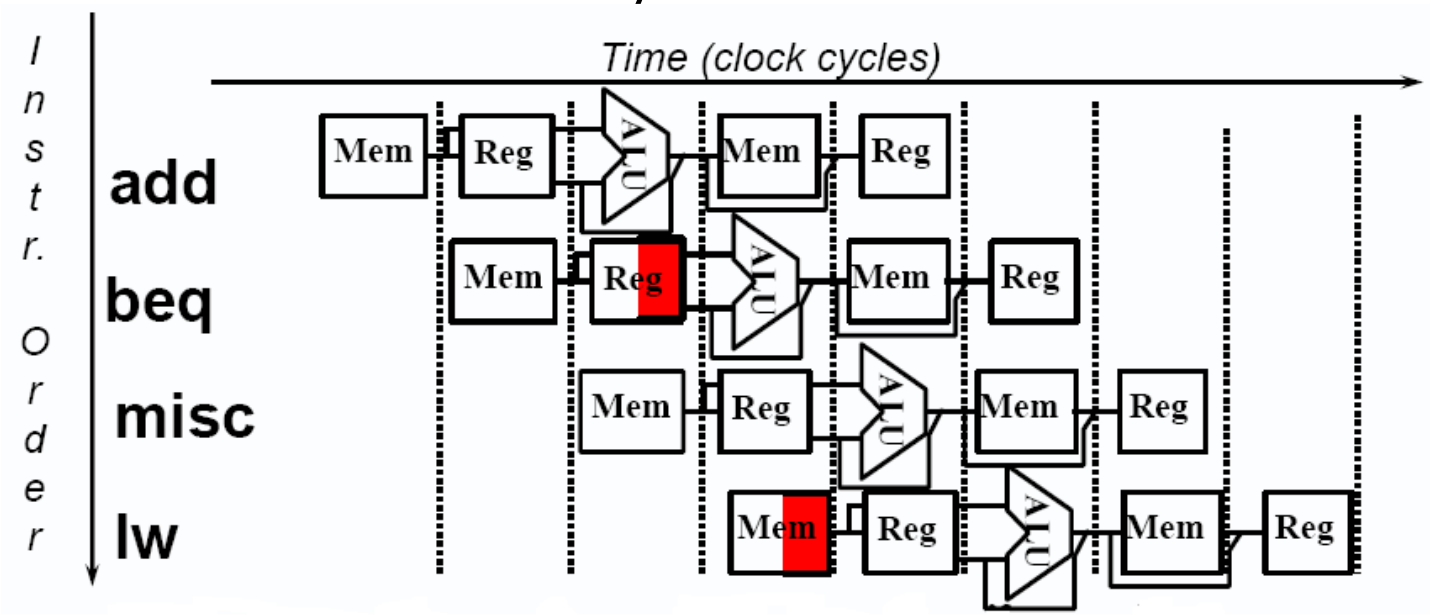
Dynamic Branch Prediction

- 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) – for statistics
  - 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 (skip)

- Predict-not-taken + branch decision at ID
  → the following inst. is always executed
  → branches take effect 1 cycle later



- 0 clock cycle per branch instruction if can find instruction to put in slot (≅50% of 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 (refer to Chap.4 - part I slide), not this pipelined version

Interrupt and Exception

Old textbook in multi-cycle implementation – more easy to understand

Interrupt and Exception (Chap.4.10)

- Interrupts were initially created to handle unexpected events like **arithmetic overflow** and to **signal requests for service from I/O devices.**
- Some events generated internally or externally:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Interrupt and Exception (Chap.4.10)

- ***Exception:*** any unexpected change in control flow without distinguishing whether the cause is internal or external
- ***Interrupt:*** only when the event is externally caused
- We will only discuss how to handle ***an undefined instruction*** or an ***arithmetic overflow*** in this chapter.

- How exceptions are handled:
  - Save the address of the offending instruction in the ***Exception Program Counter (EPC)*** and transfer control to the operating system at some specified address.
  - Take some predefined action in response to an overflow, or stop the execution of the program and report an error (Execute ***Interrupt Service Routine, ISR***)
  - Terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program.

Handling Exceptions in MIPS

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)

1. Save PC of offending (or interrupted) instruction
   - In MIPS: Exception Program Counter (EPC)
2. Save indication of the problem
   - In MIPS: **Cause register**
   - We'll assume 1-bit
     - 0 for undefined opcode, 1 for overflow
3. Jump to handler at 8000 00180

Interrupt Registers

Two main methods used to communicate the reason for an exception:

1. **Cause register**: A **status register** which holds a field that indicates the reason for the exception (used in MIPS architecture)
2. **Vectored interrupt**: An interrupt for which the address to which control is transferred is determined by the cause of the exception.

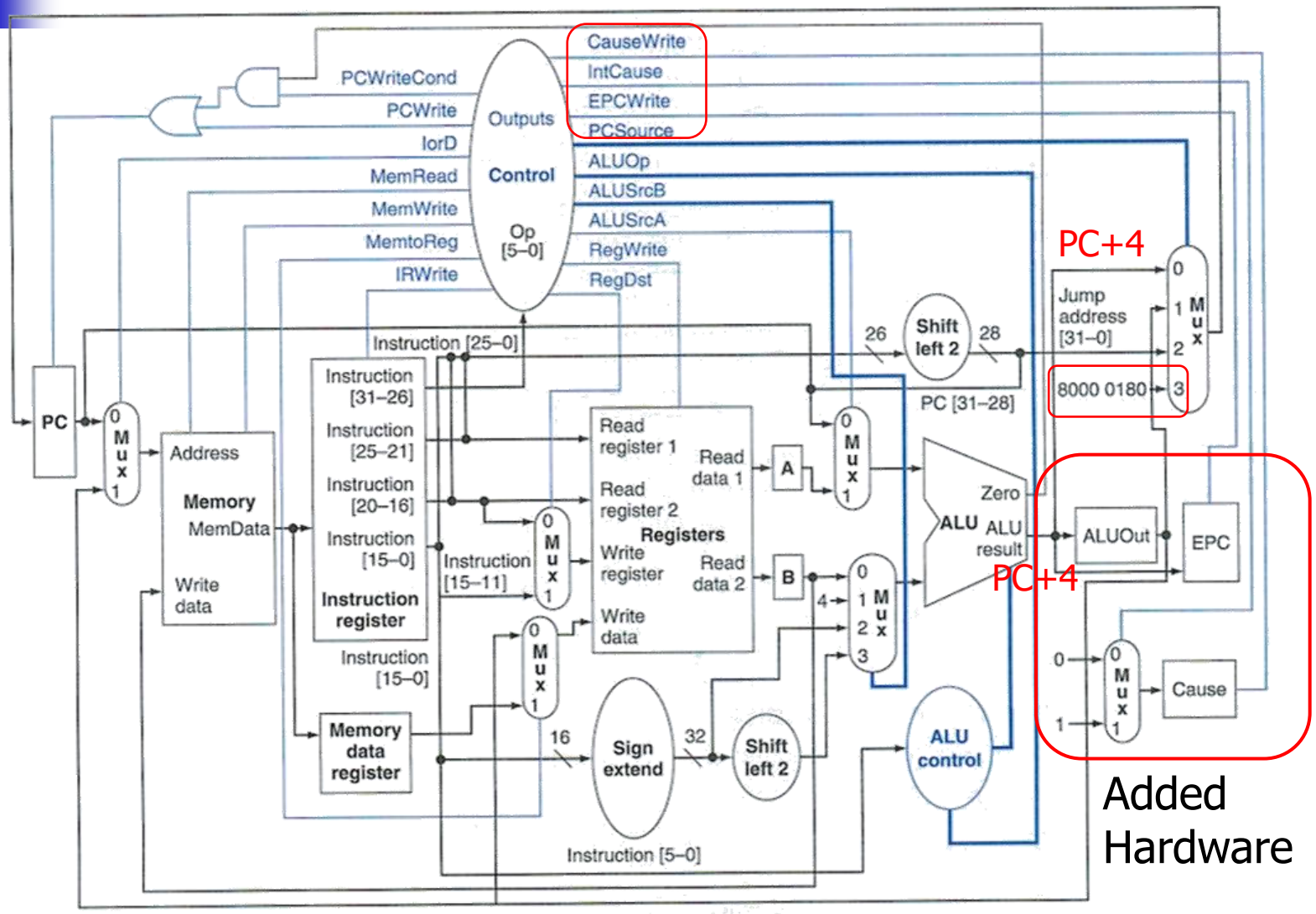
Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

- The operating system knows the reason for the exception by the address at which it is initiated.
- The address are separated by 32 bytes or 8 instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence.

Support Exception in MIPS CPU

- For MIPS exception system
  - Two additional registers to the datapath:
    1. **EPC (exception program counter):** A 32-bit register used to hold the address of the affected instruction.
    2. **Cause:** A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits.
  - 3 Additional control signals:
    - *EPCWrite* (update the problem instruction address)
    - *CauseWrite* (update the Cause number)
    - *IntCause*
  - Change the 3-way multiplexor (controlled by PCSouse) to a 4-way multiplexor, with additional input wired to the constant value **8000 0180$_{hex}$** → The Operating system entry point for exception handling subroutines (for arithmetic overflow)

Support Exception in Multi-Cycle MIPS CPU



Exceptions in a Pipelined Design

Optional

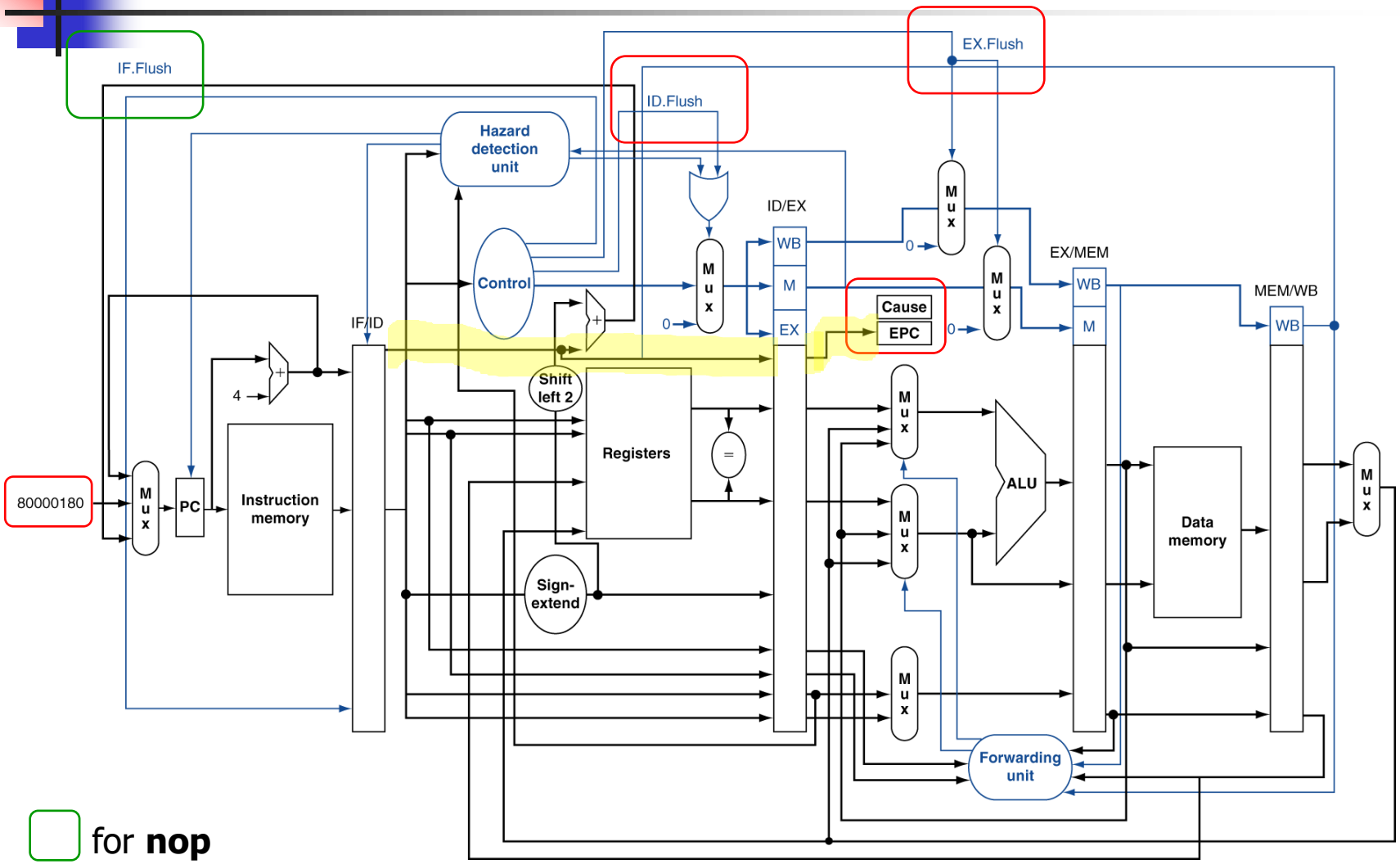
Exceptions in a Pipelined Design

- 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** instruction and subsequent instructions
  - Set **Cause** and **EPC register** values
  - Transfer control to ISR handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions



for **nop**

Exception Properties

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

Exception Example

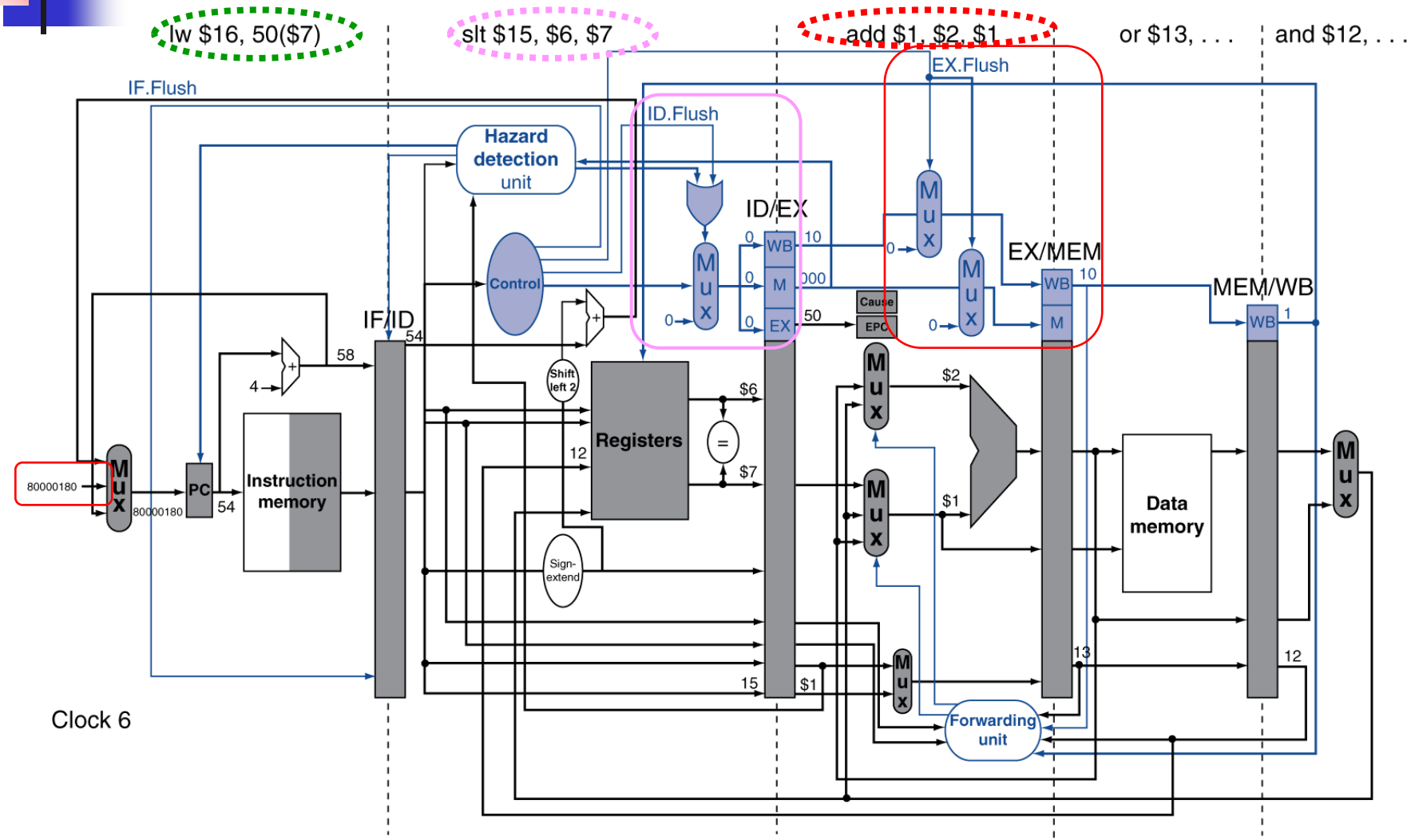
- Exception on add in

```
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)
…
```

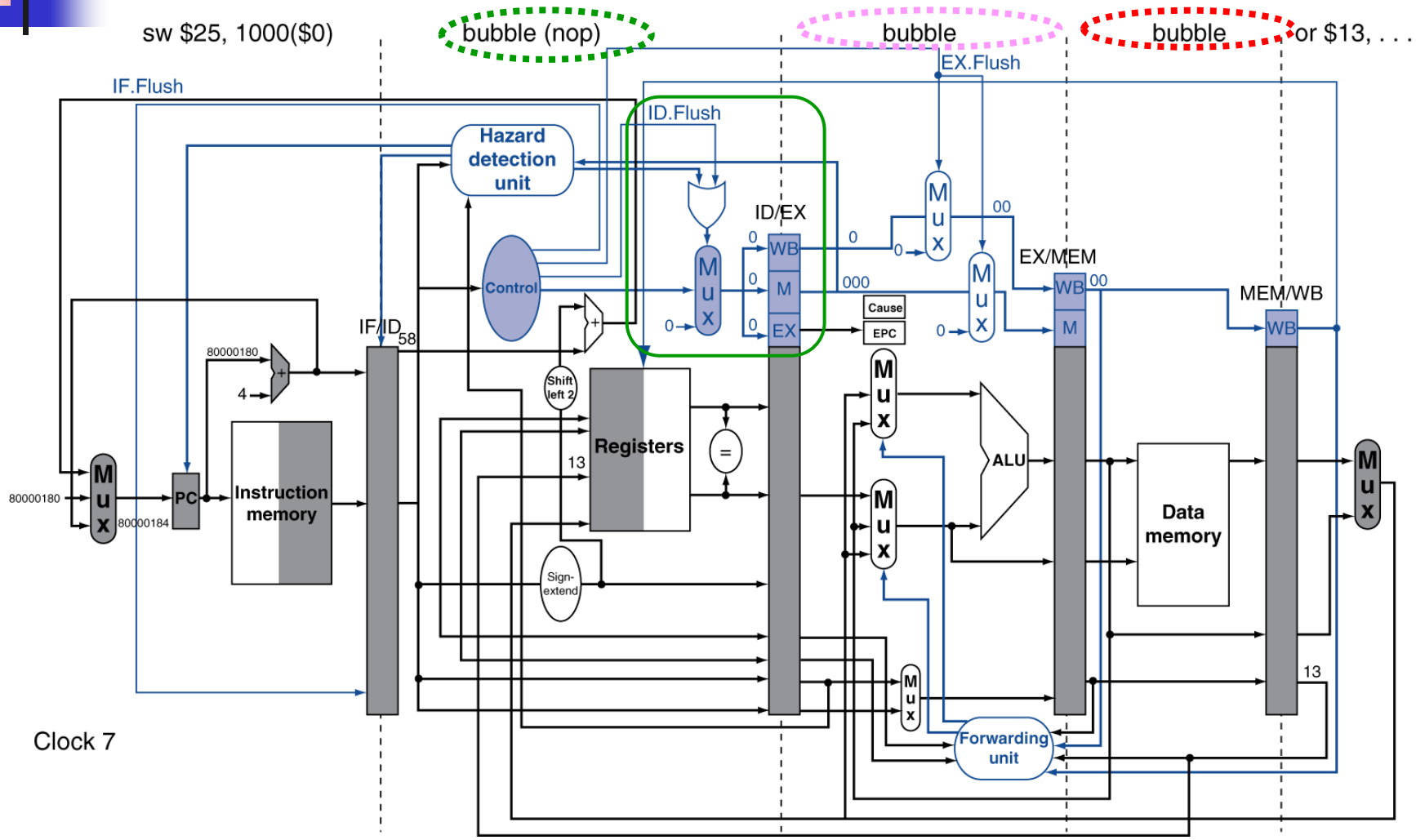
- Handler (ISR at 80000180)

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

Exception Example



Exception Example



Chap. 4.10
Parallelism via Instructions - Performance Enhancement

(for reference only)

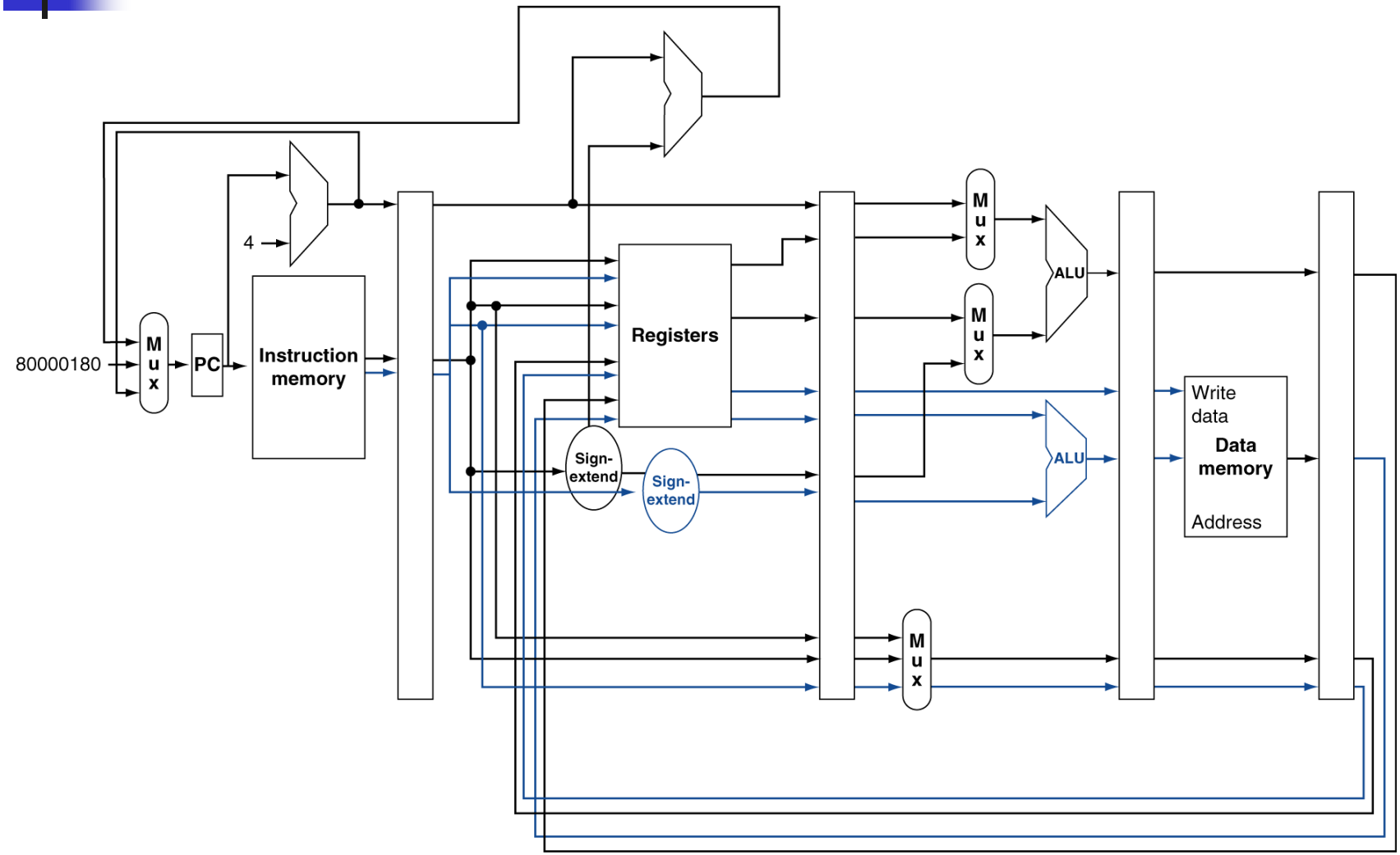
Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP (instruction-level parallelism)
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ 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

MIPS with Static Dual Issues



MIPS with Static Dual Issue

- Two-issue packets
  - One Integer ALU/branch instruction + One load/store instruction (combined)
  - 64-bit aligned
    - ALU/branch, then load/store
    - If we cannot make it, 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

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 (5~8 instructions per packet)
  - 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

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 (sw-lw change to lw-sw)
    - 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

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/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	1	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

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.**