CPE 431/531

Chapter 4 – The Processor

Dr. Rhonda Kay Gaede

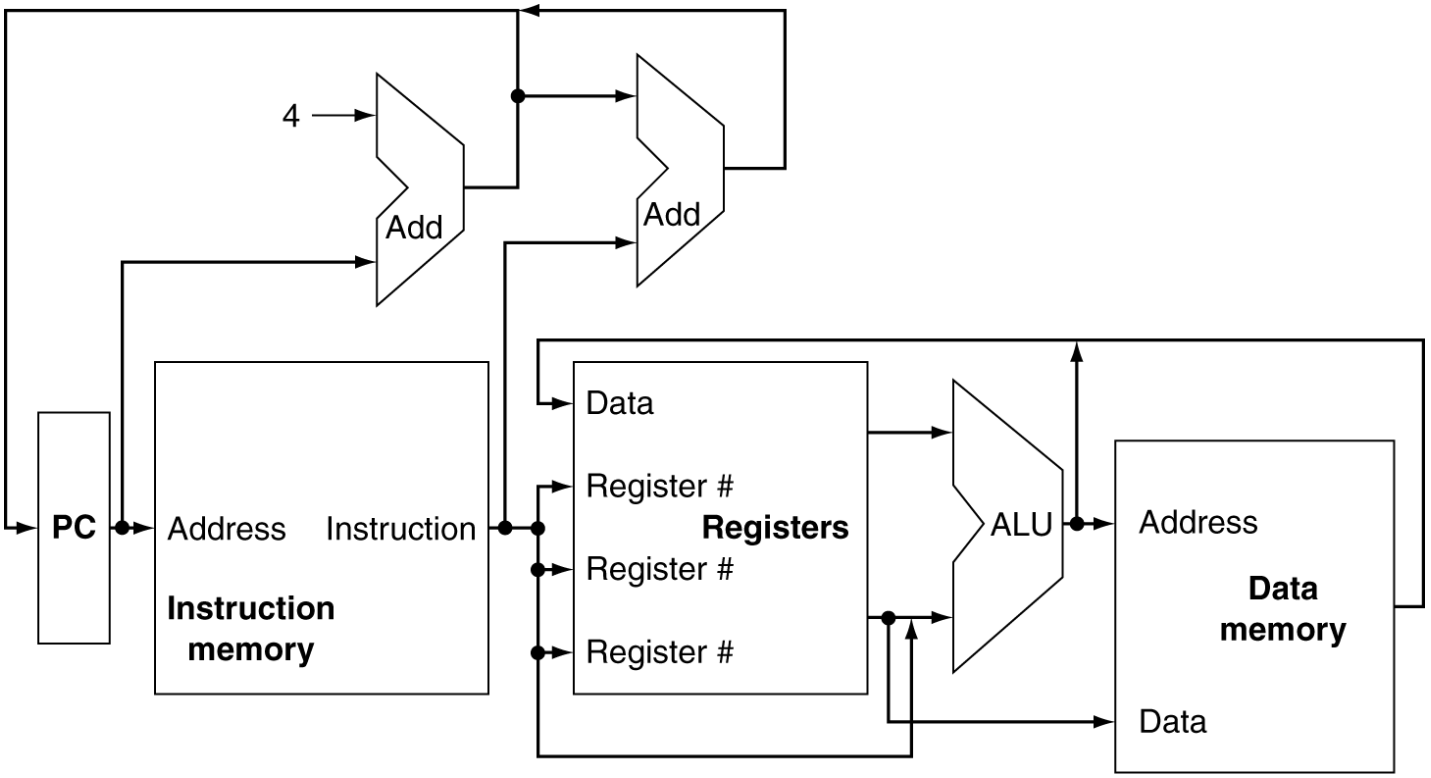
4.1 Implementation Basics

- Performance Factors
  - Instruction Count
  - Cycle Time
  - CPI
- A Basic MIPS Implementation
  - Simple subset: **`lw, sw, add, sub, and, or, slt, beq, j`**

4.1 Implementation Overview

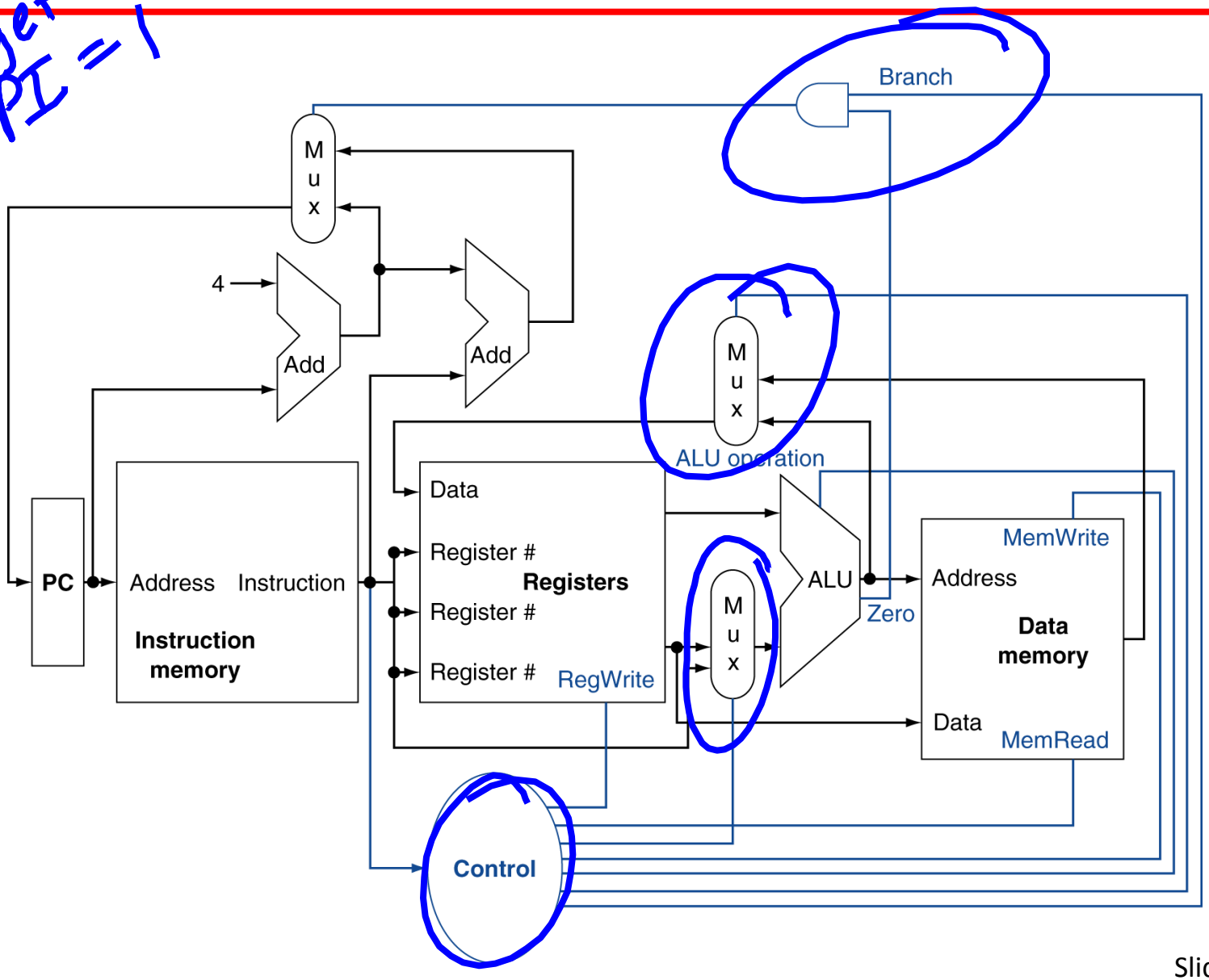
- All instructions begin the same way
  - instruction fetch
  - read two registers
  - instruction decode
- Then, it depends on the instruction
  - **`lw`**
  - **`sw`**
  - **`add`** et.al.
  - **`beq`**

4.1 Datapath


4
Add
Add
PC
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
ALU
Address
Data memory
Data

4.1 Datapath + Control

Target CPI = 1



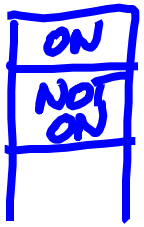
4.2 Classes and Values

- Two classes of logic
  - sequential
  - combinational

$O = f(I, S)$

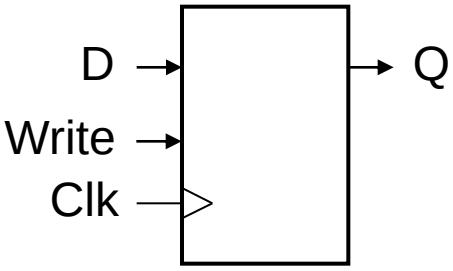
$O = f(I)$

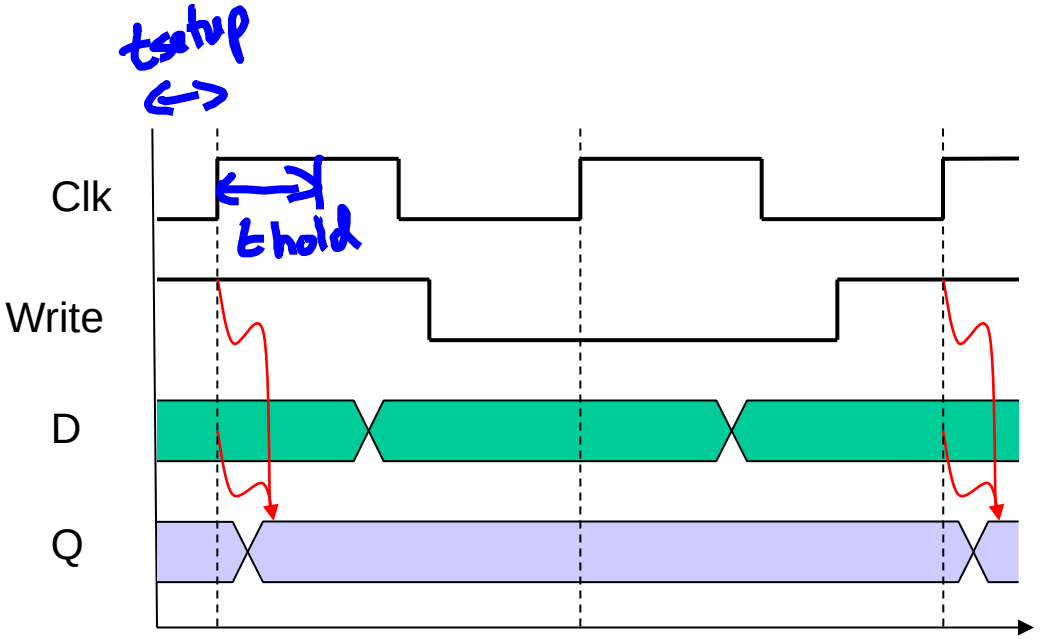
- Two logic values
  - asserted
  - deasserted



4.2 Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later.

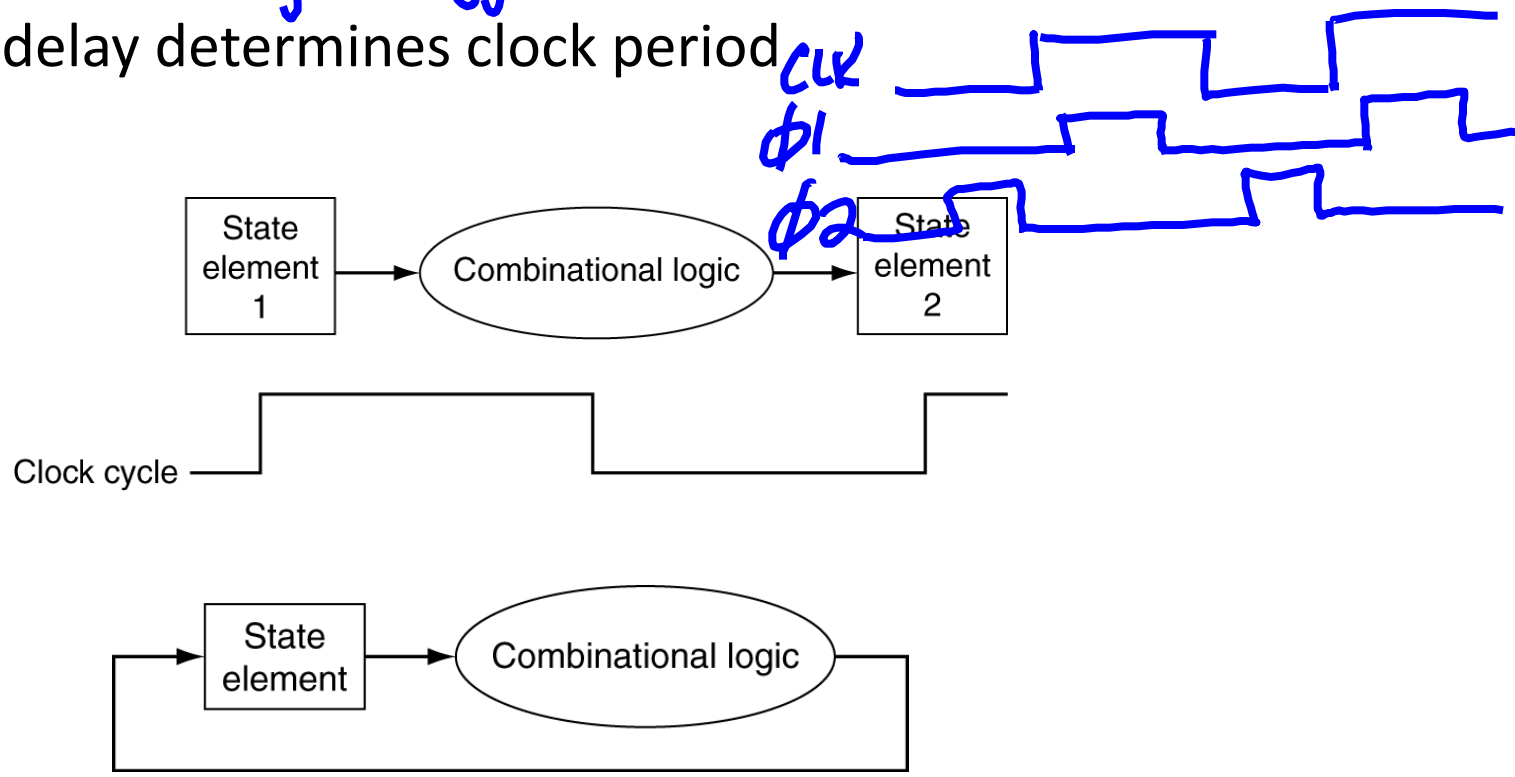




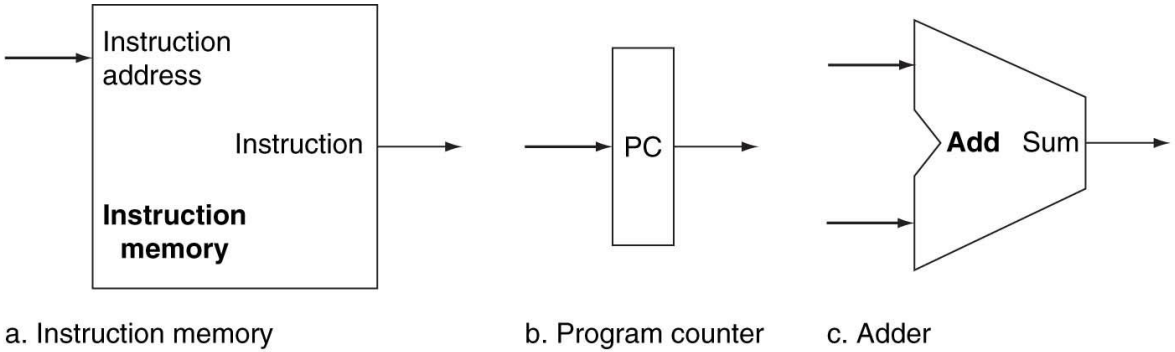
setup time
hold time

4.2 Clocking Methodology

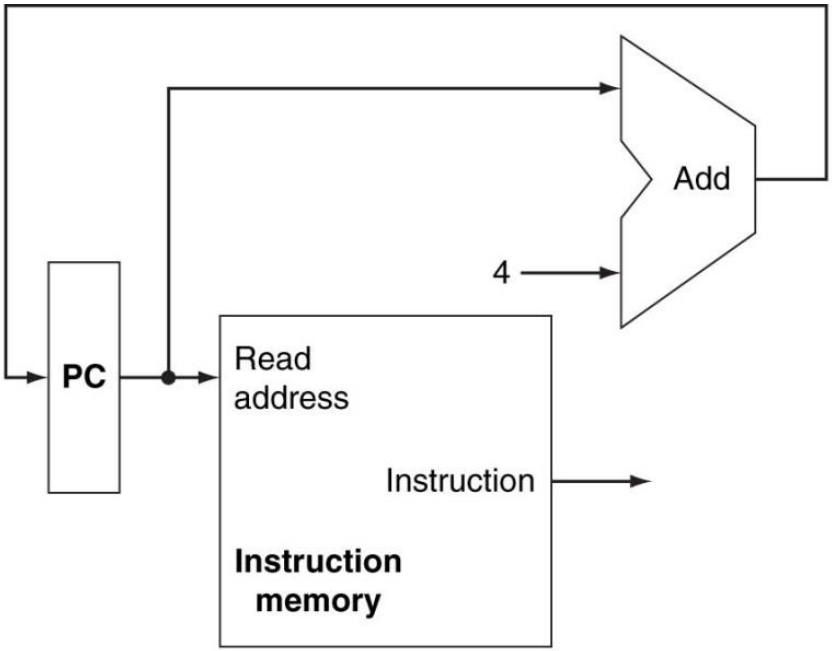
- A clocking methodology defines when signals can be _read_ and when they can be _written_.
- We assume an _edge-triggered_ clocking methodology.
- Longest delay determines clock period



4.3 Instruction Fetch and Default Sequencing

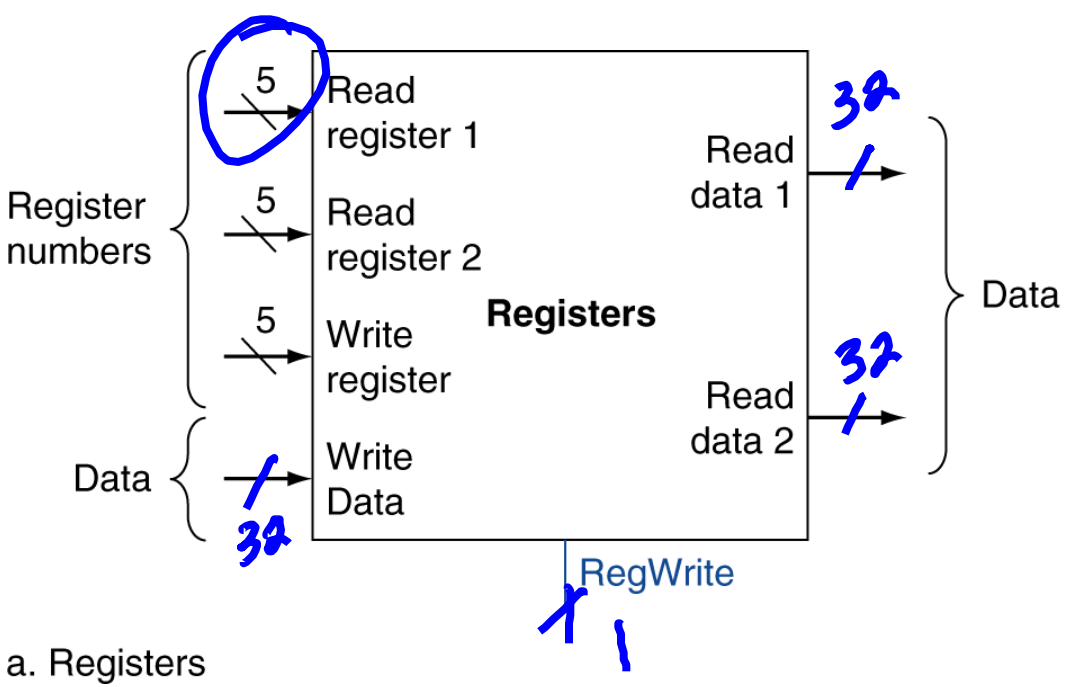


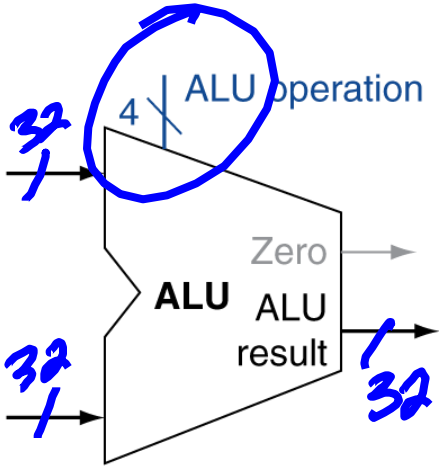
a. Instruction memory b. Program counter c. Adder



4.3 R-type Instruction Requirements

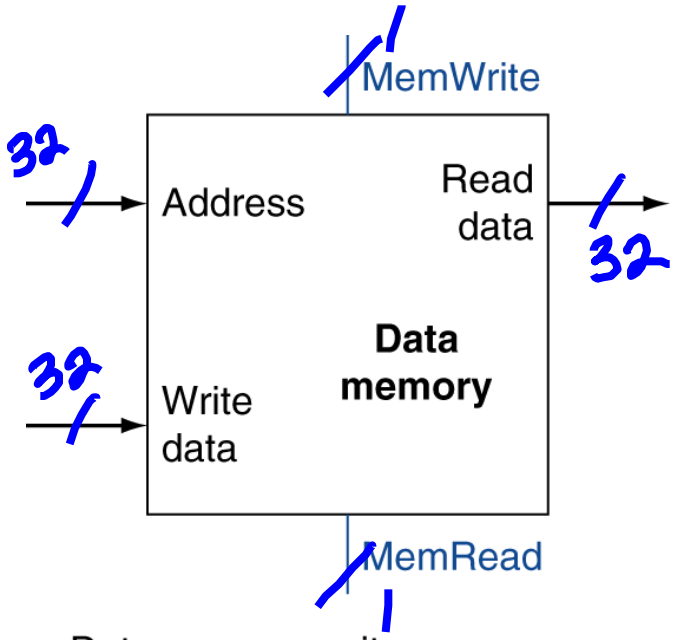
- Read two register operands
- Perform arithmetic/logical operation
- Write register results



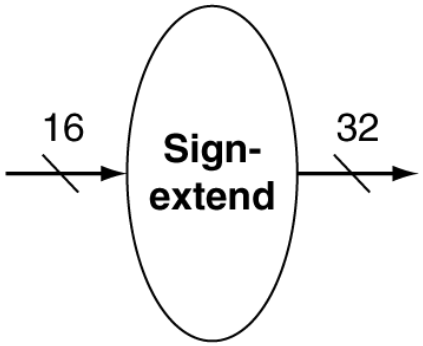


4.3 `lw/sw` Instruction Requirements

- Read register operands need 2 for sw, 1 for lw
- Calculate effective address using 16-bit offset
- Load: Read memory and update register
- Store: Write register value to memory



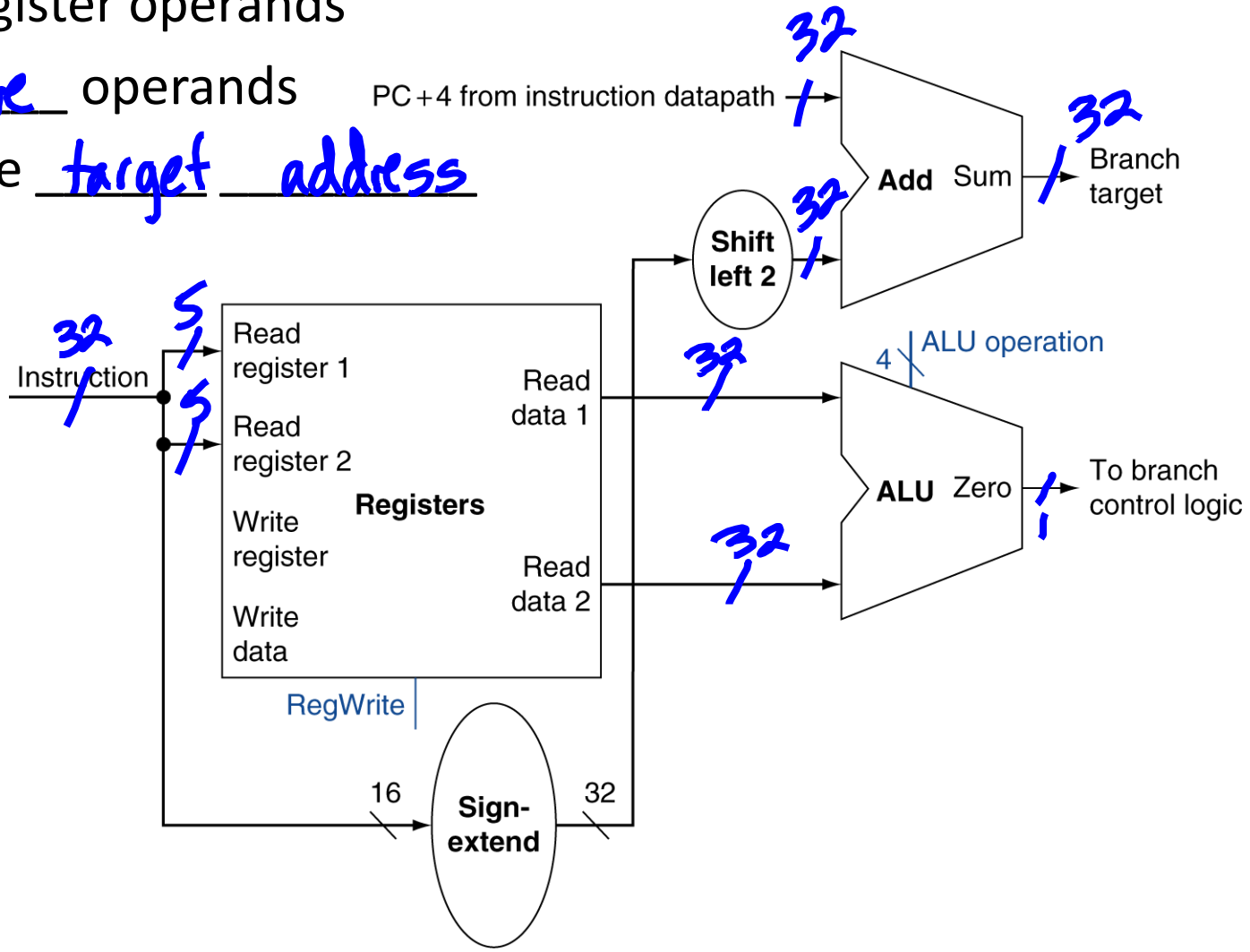
a. Data memory unit



b. Sign extension unit

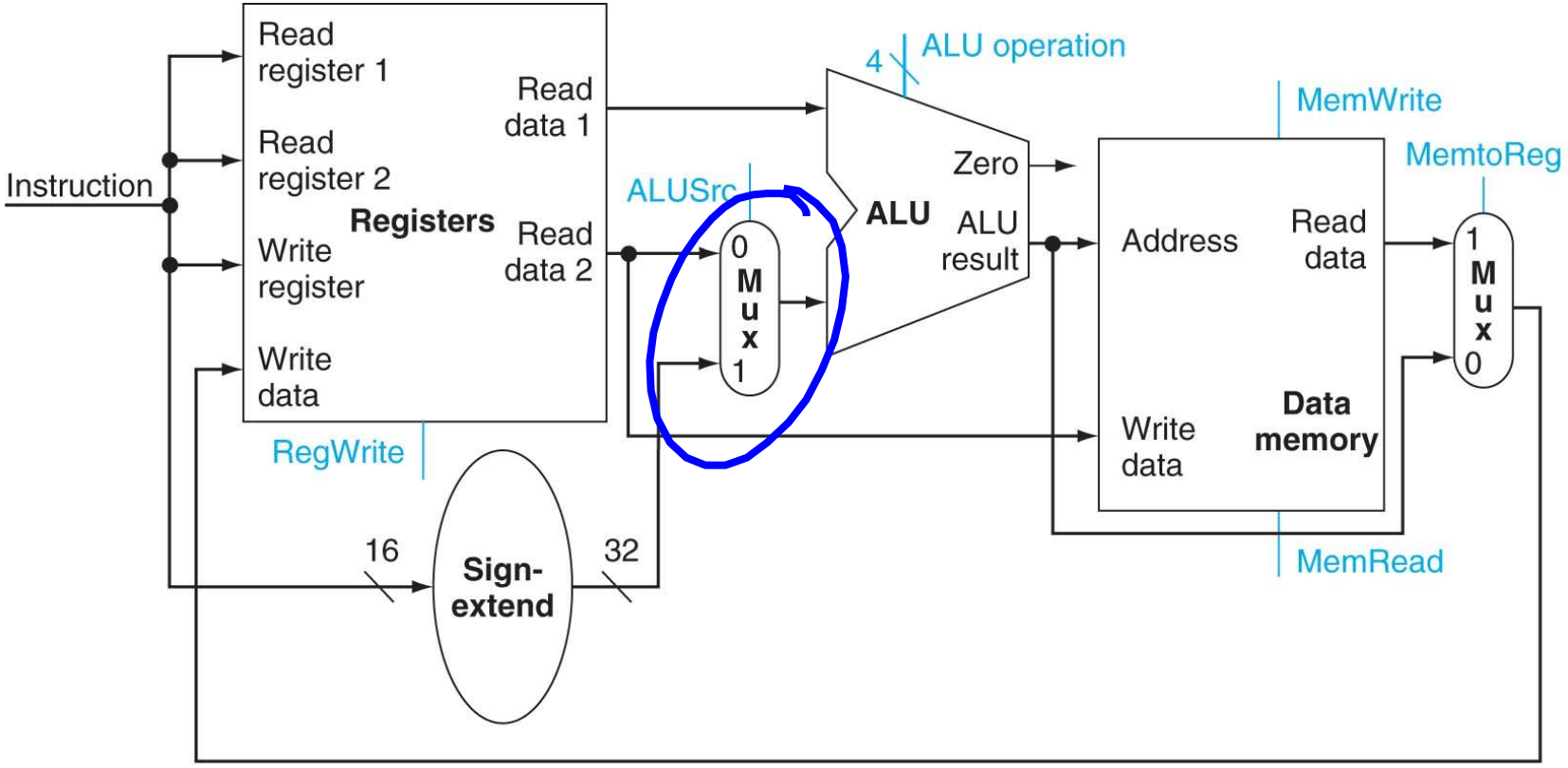
4.3 `beq` Instruction Requirements

- Read register operands
- Compare operands
- Calculate target address

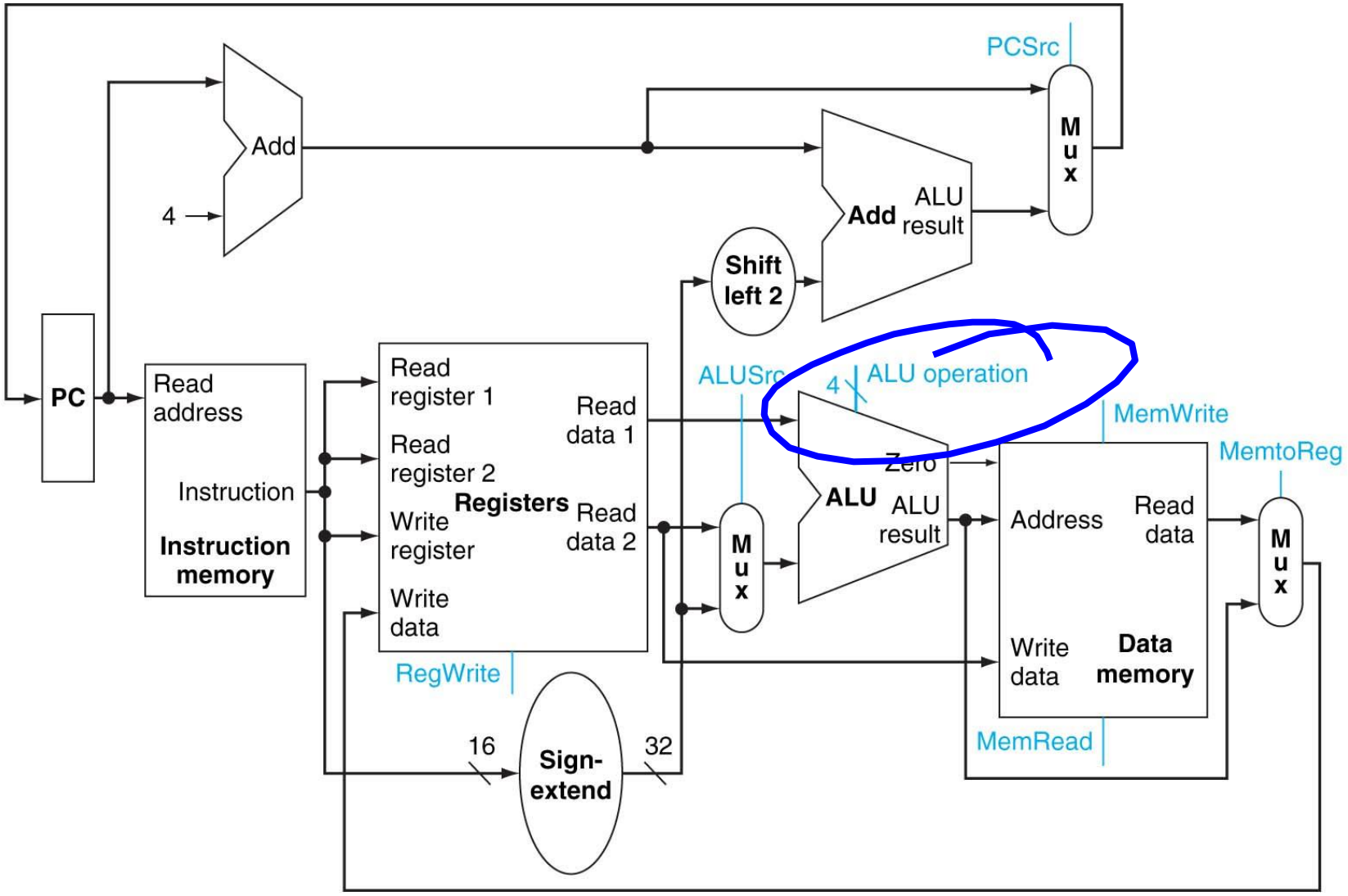


4.3 Creating a Single Datapath: R-type + `lw/sw`

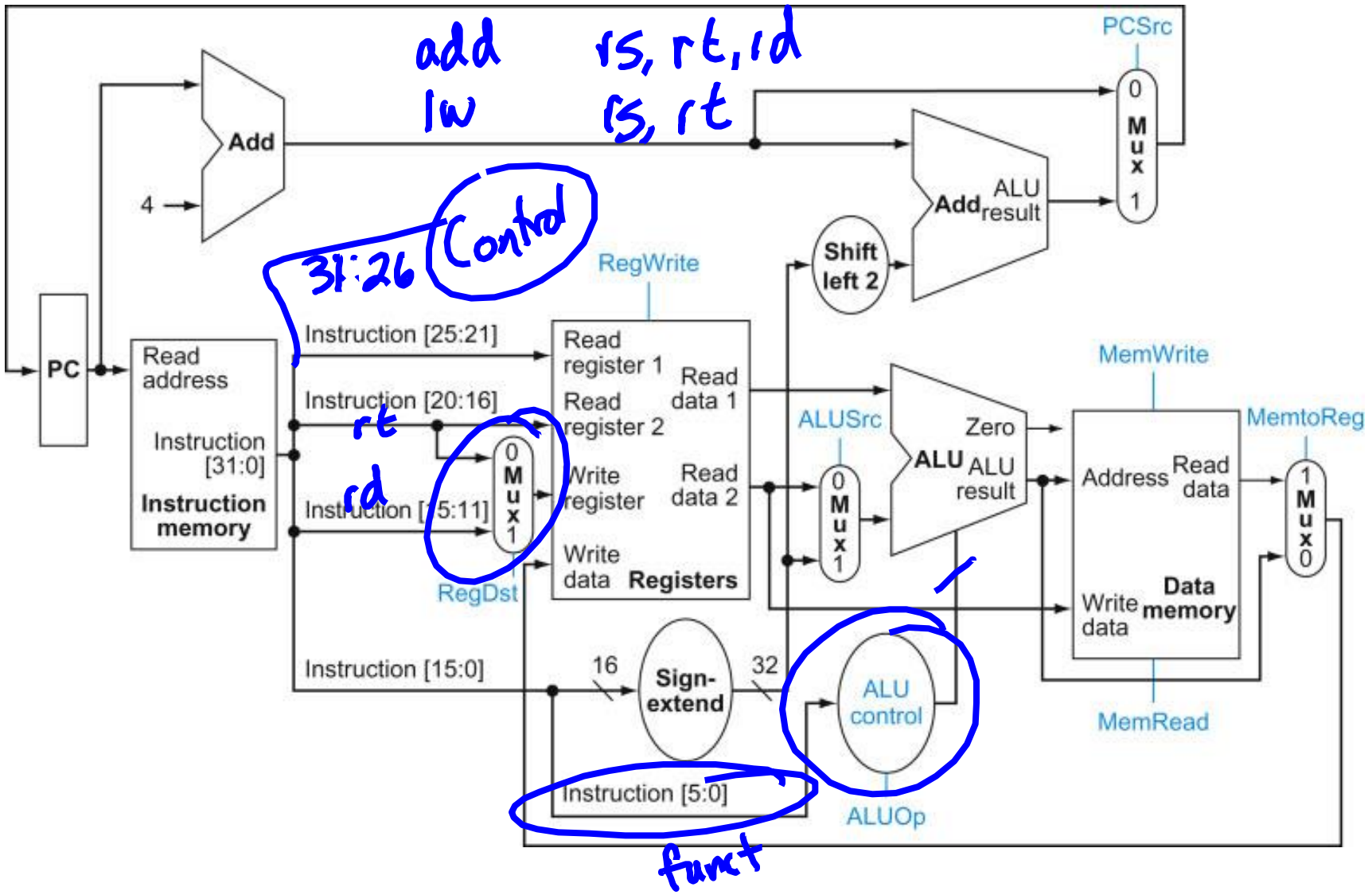
- We want a single-cycle implementation.
- Separate instruction and data memories are required.
- When I say sources, you say MUX.



4.3 Adding `beq` to the R-type + `lw/sw` Datapath

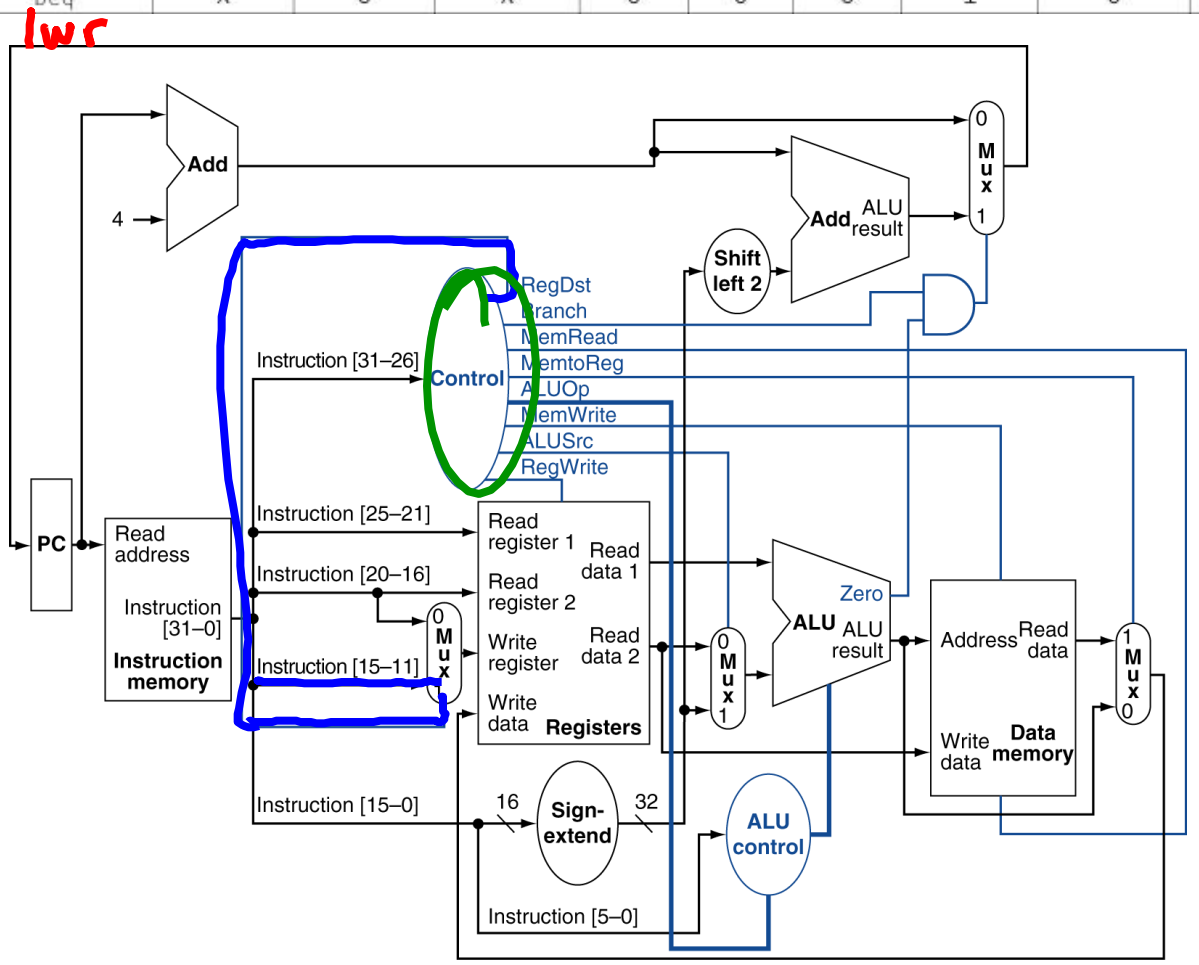


4.4 Defining Necessary Control



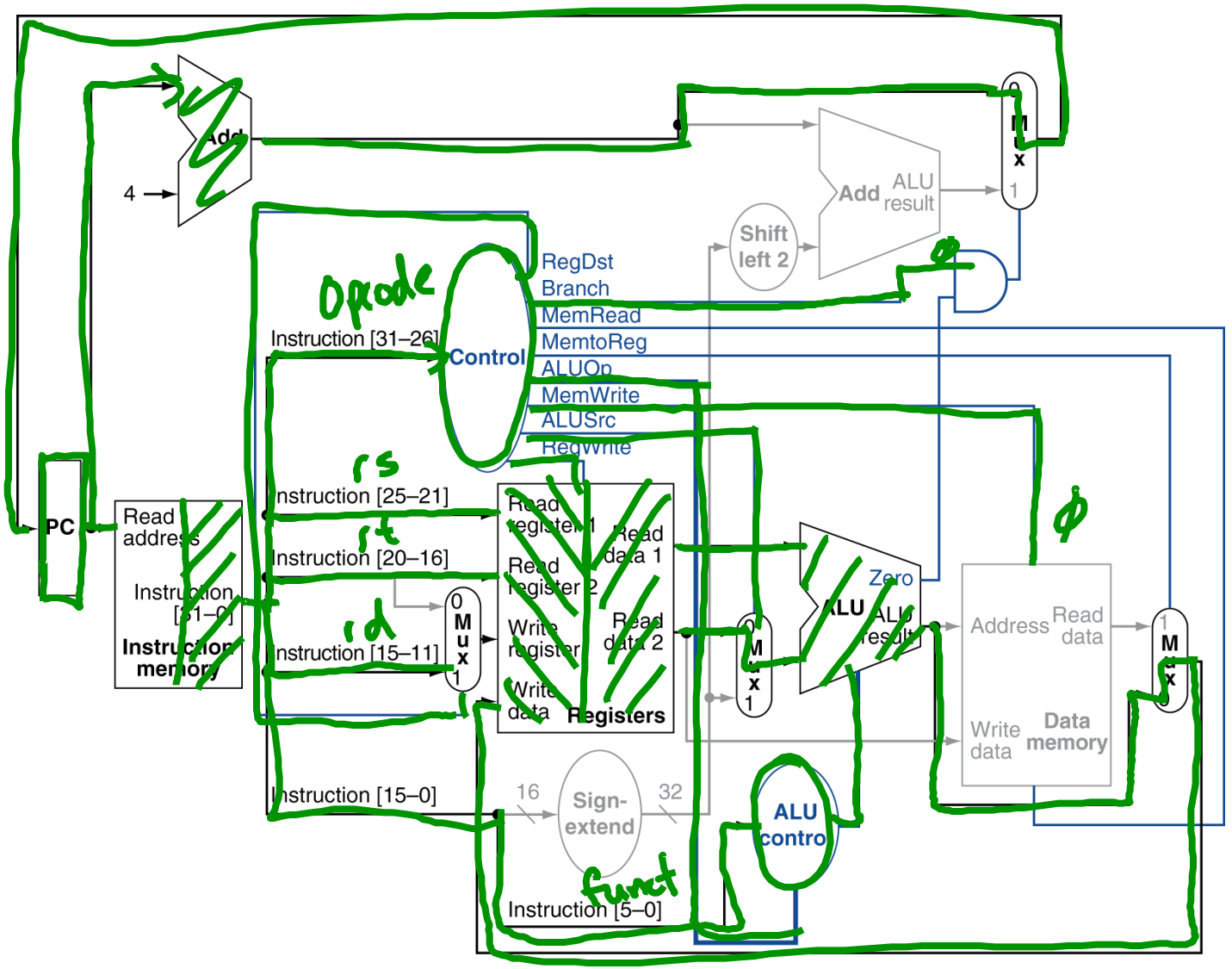
4.4 Adding a Control Unit

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

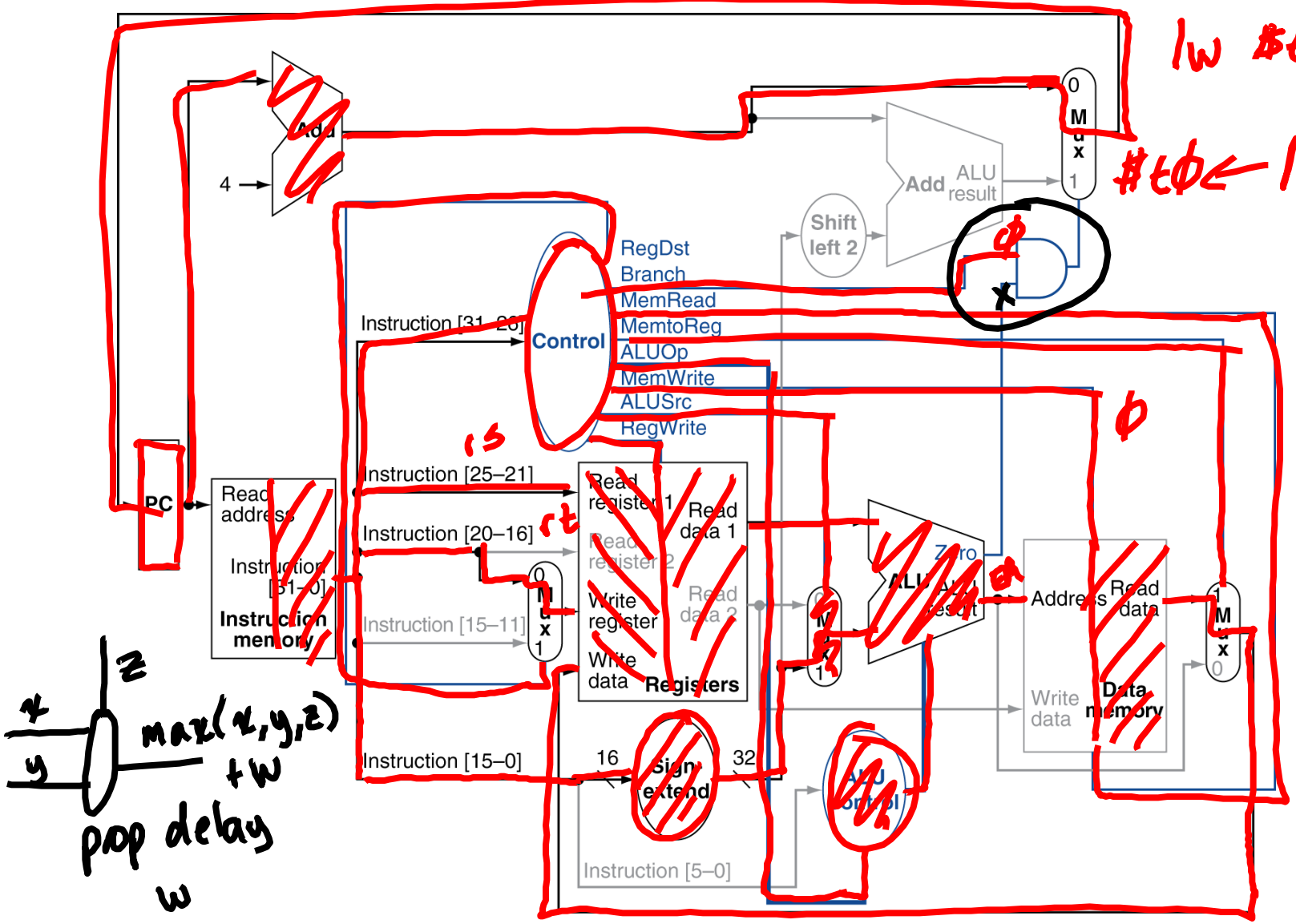


lw
MemRead
!
MemWrite
Ø

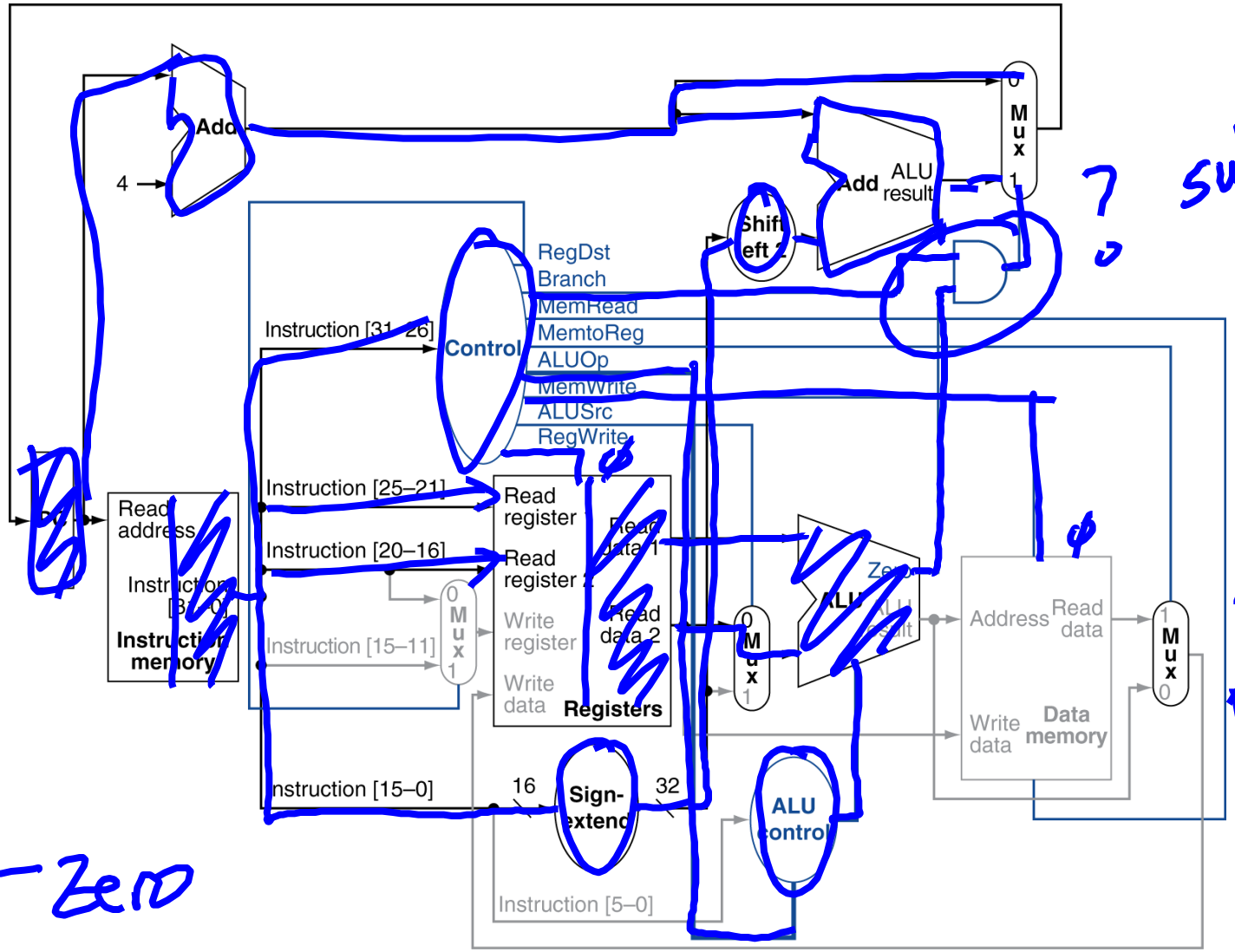
4.4 R-type Instruction Execution



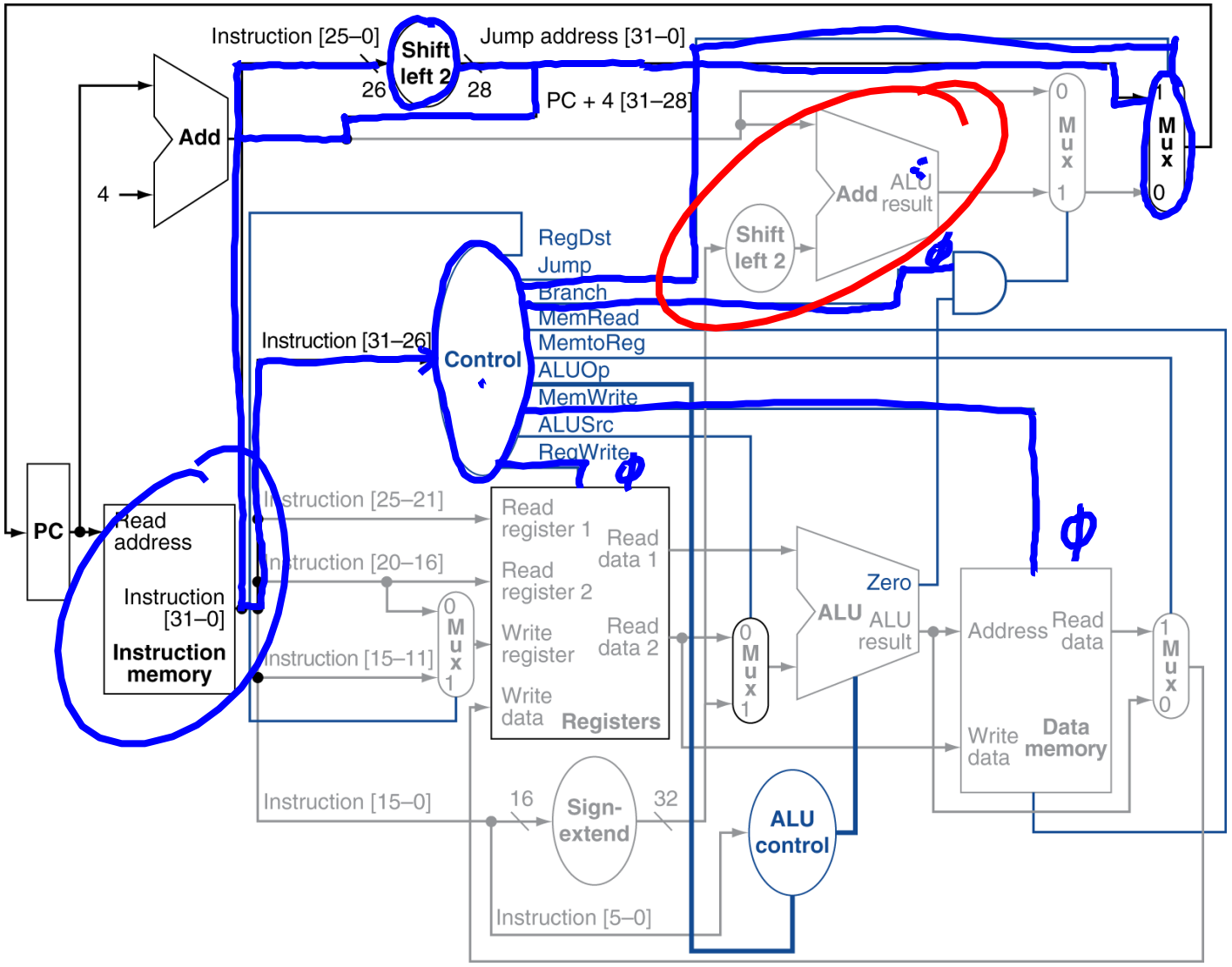
4.4 `lw` Instruction Execution



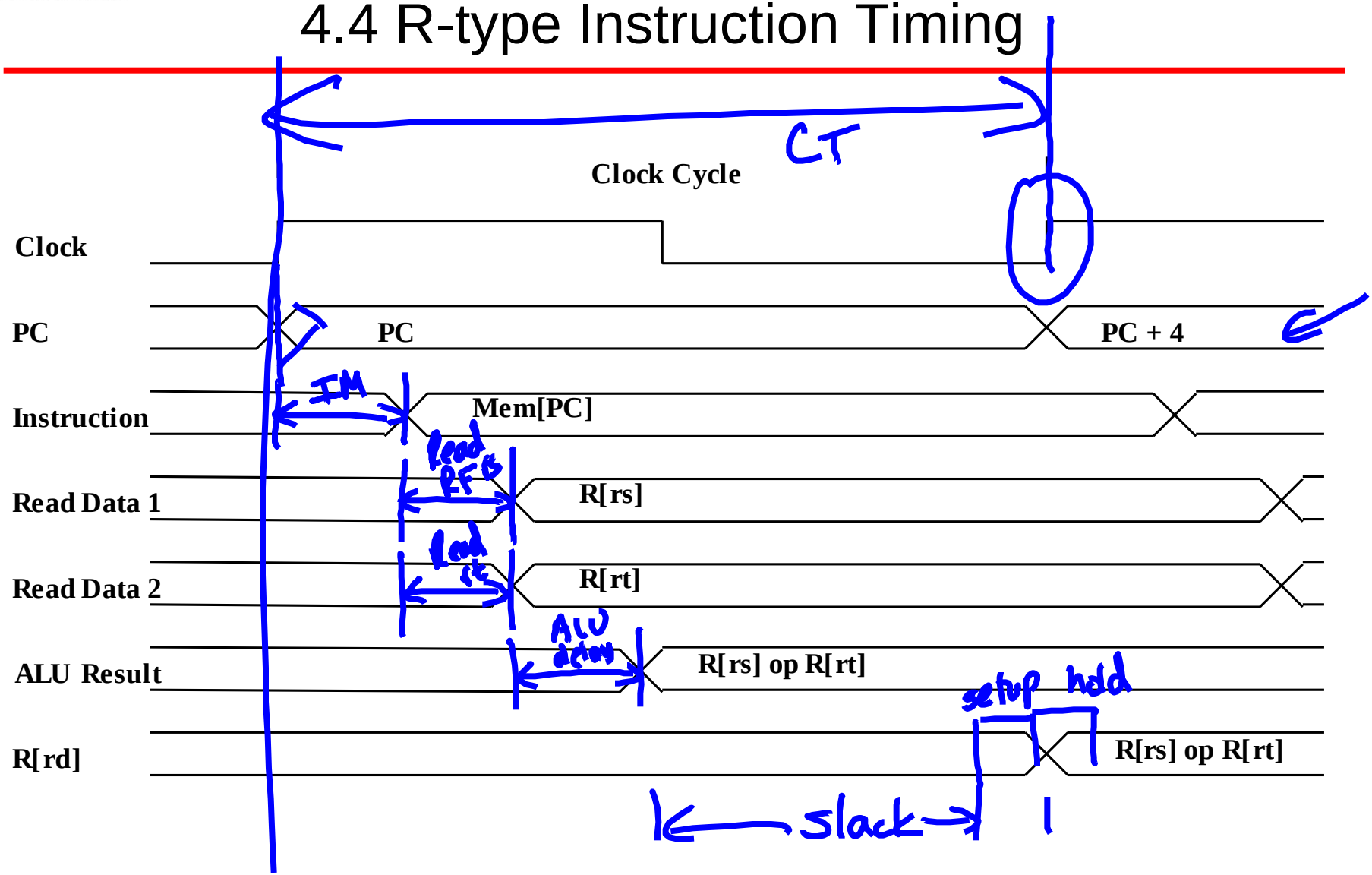
4.4 beq Instruction Execution



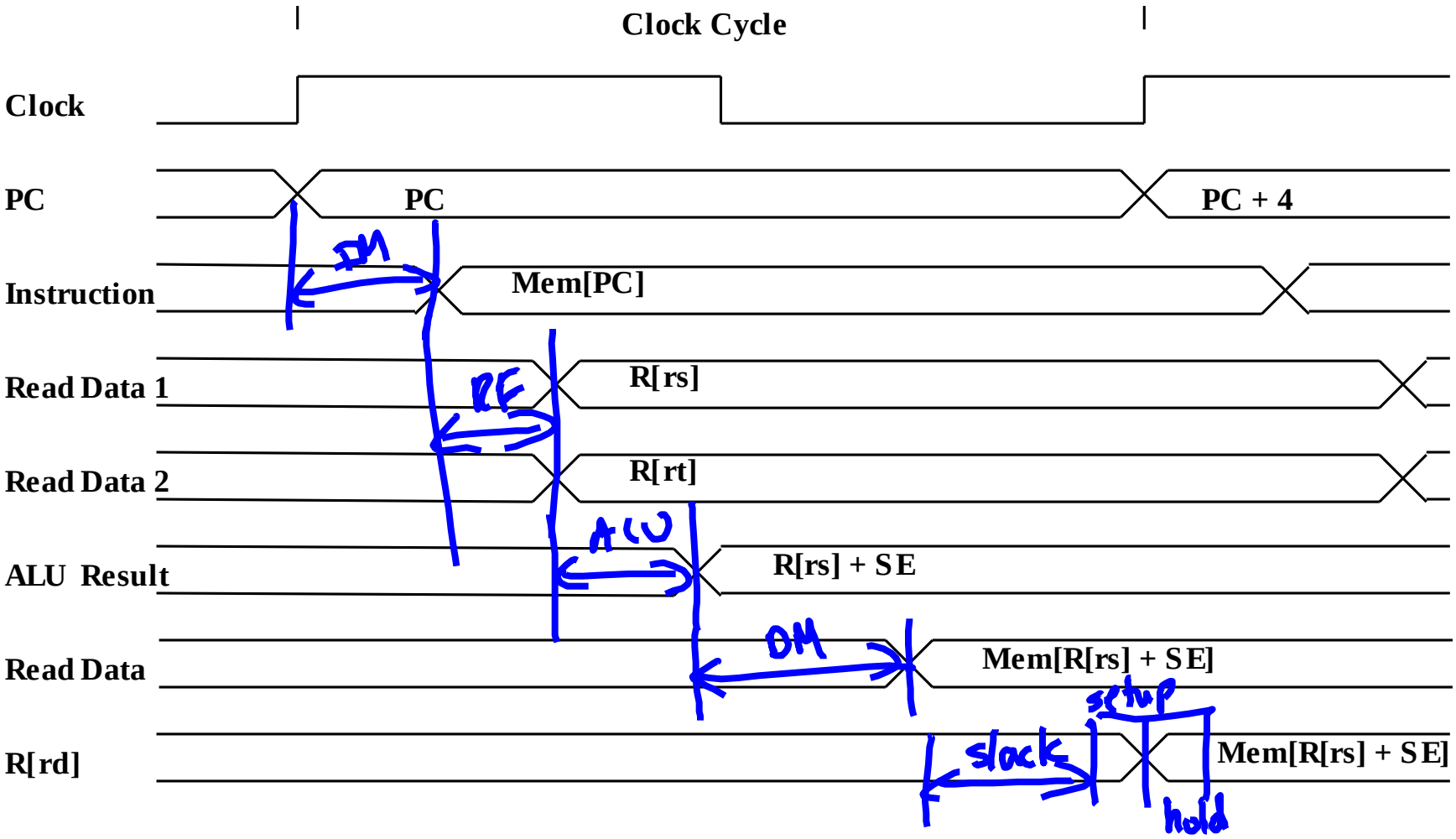
4.4 Implementing Jumps



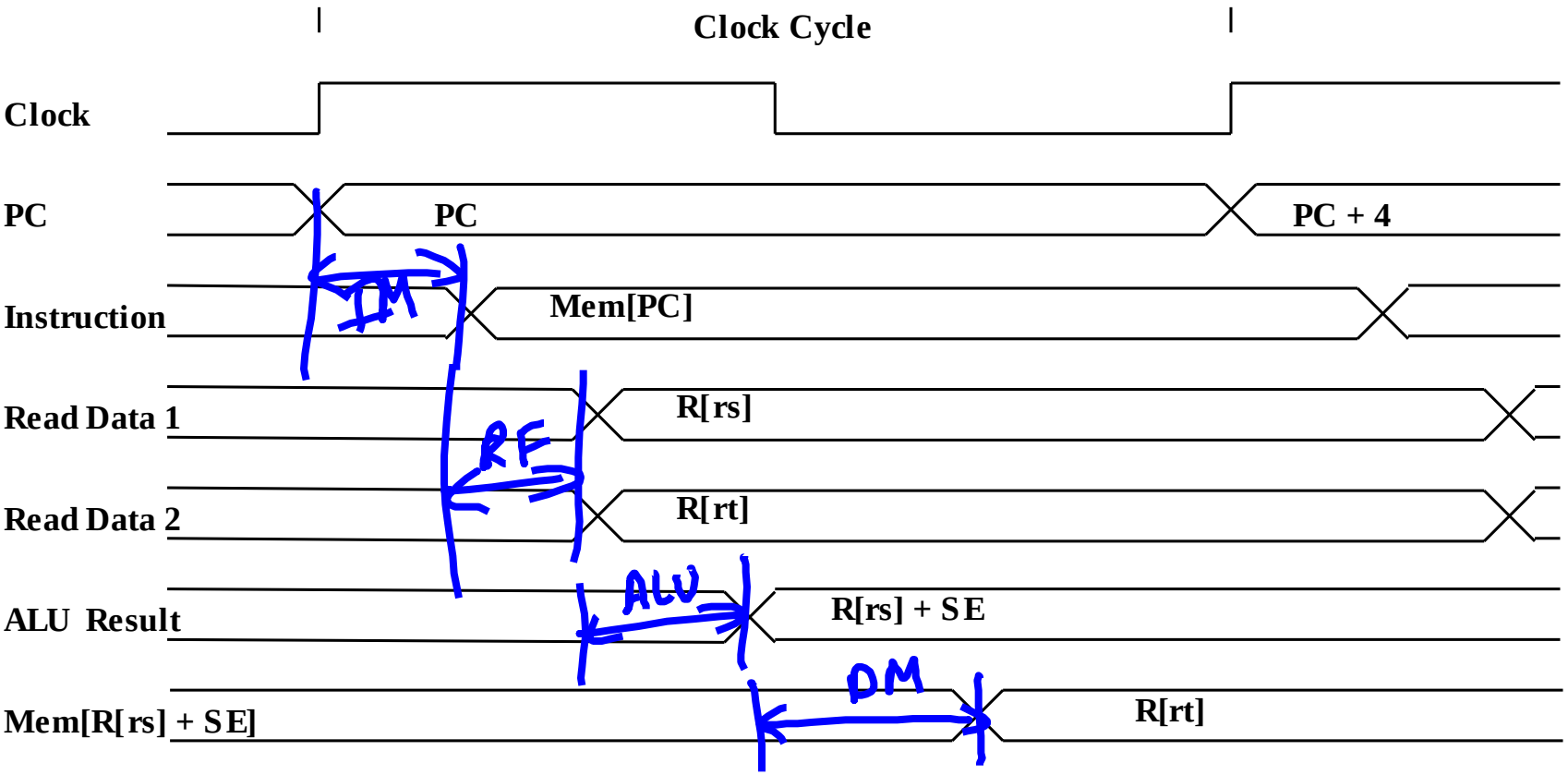
4.4 R-type Instruction Timing



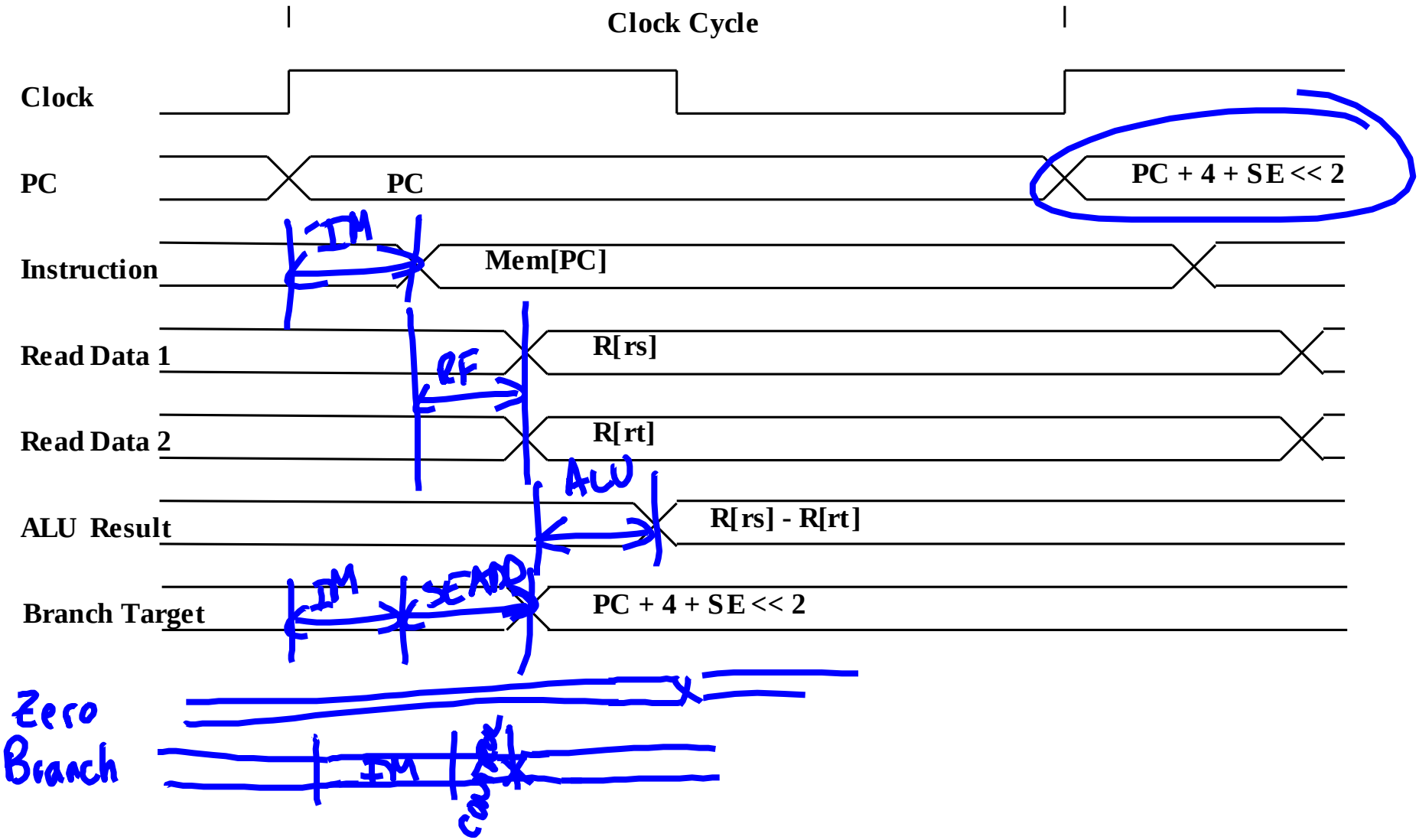
4.4 `lw` Instruction Timing



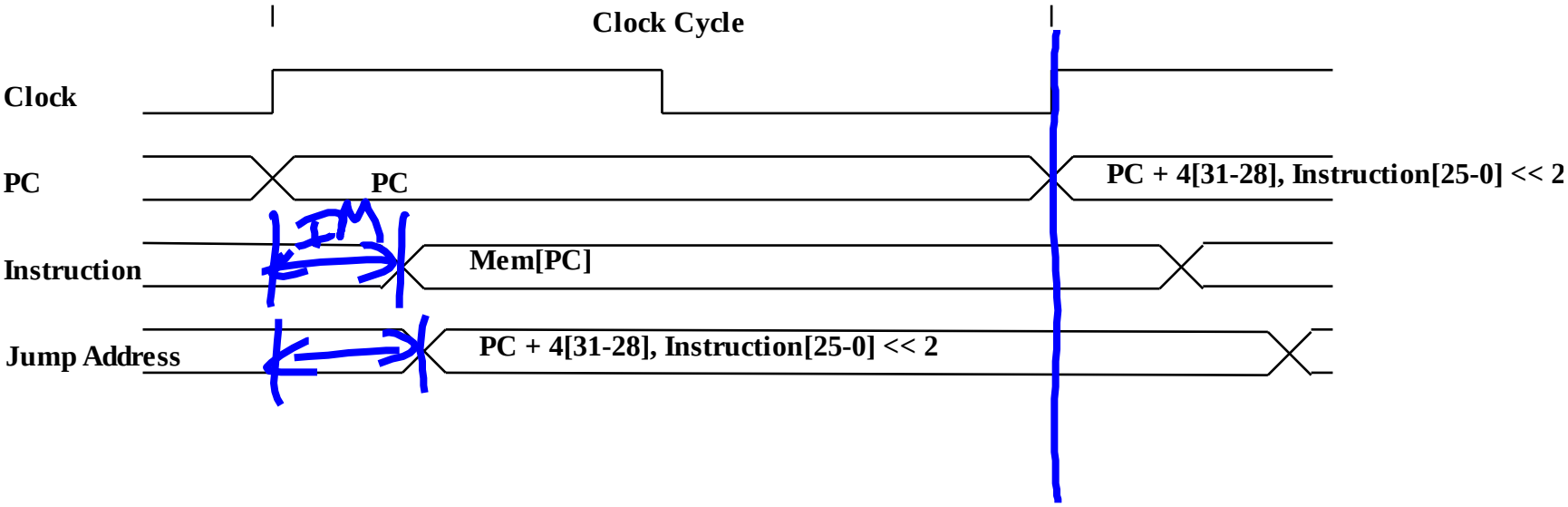
4.4 sw Instruction Timing



4.4 `beq` Instruction Timing



4.4 j Instruction Timing



Add an instruction to the datapath

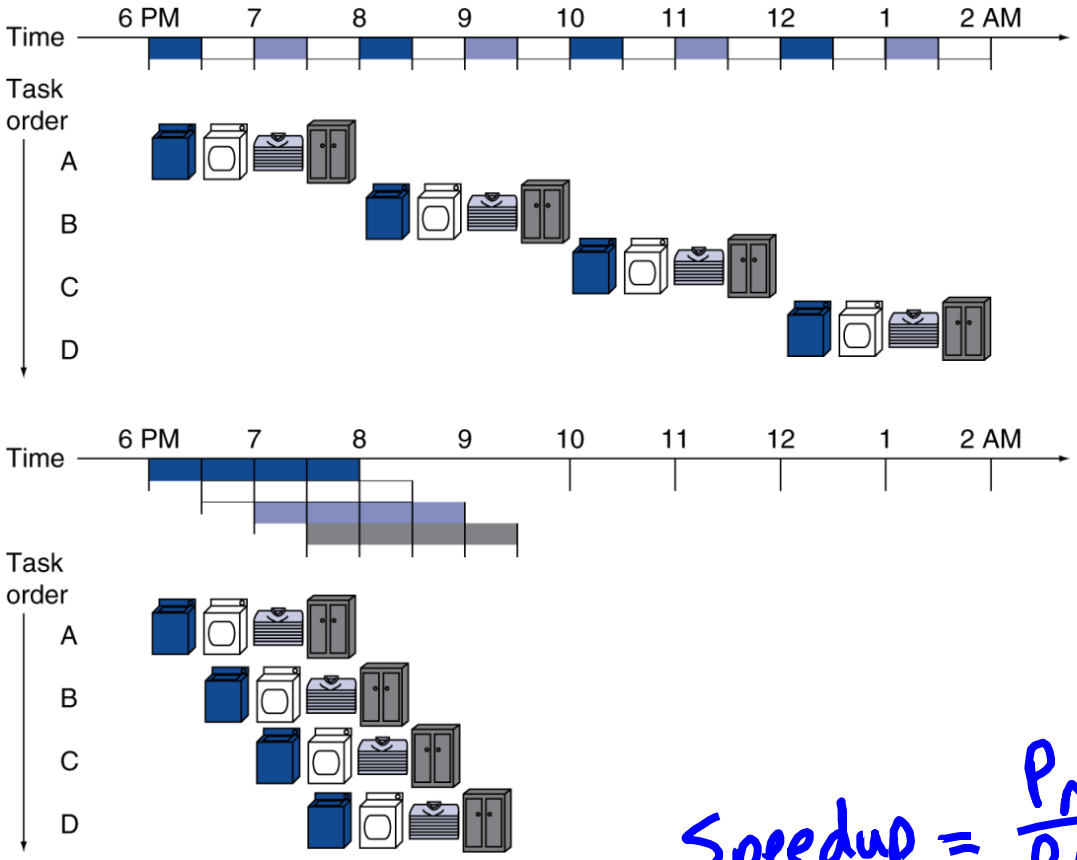
lwr $t0, $s0, $s1

$t0 ← MEM[$s0 + $s1]

4.5 An Overview of Pipelining

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
- Pipelining helps throughput, not individual execution time.
- You can't skip a stage.

4.5 The Laundry Analogy



Speedup

- One Load

  2 hours

- Four Loads

  Seral 8 hours

  Pipeline 3.5

  Speedup = 8/3.5 = 2.3

- N Loads as N $\rightarrow \infty$

$$\text{Speedup} = \frac{P_n}{P_{old}} = \frac{ET_{old}}{ET_{new}}$$

$$\frac{2n}{(0.5n + 1.5)} \approx 4$$

4.5 MIPS Processor Stages

- Instruction fetch (IF)
- Instruction decode and register read (ID)
- Execution (calculate address) (EX)
- Memory access (MEM)
- Register write (WB)

4.5 Single Cycle vs. Pipelined Performance

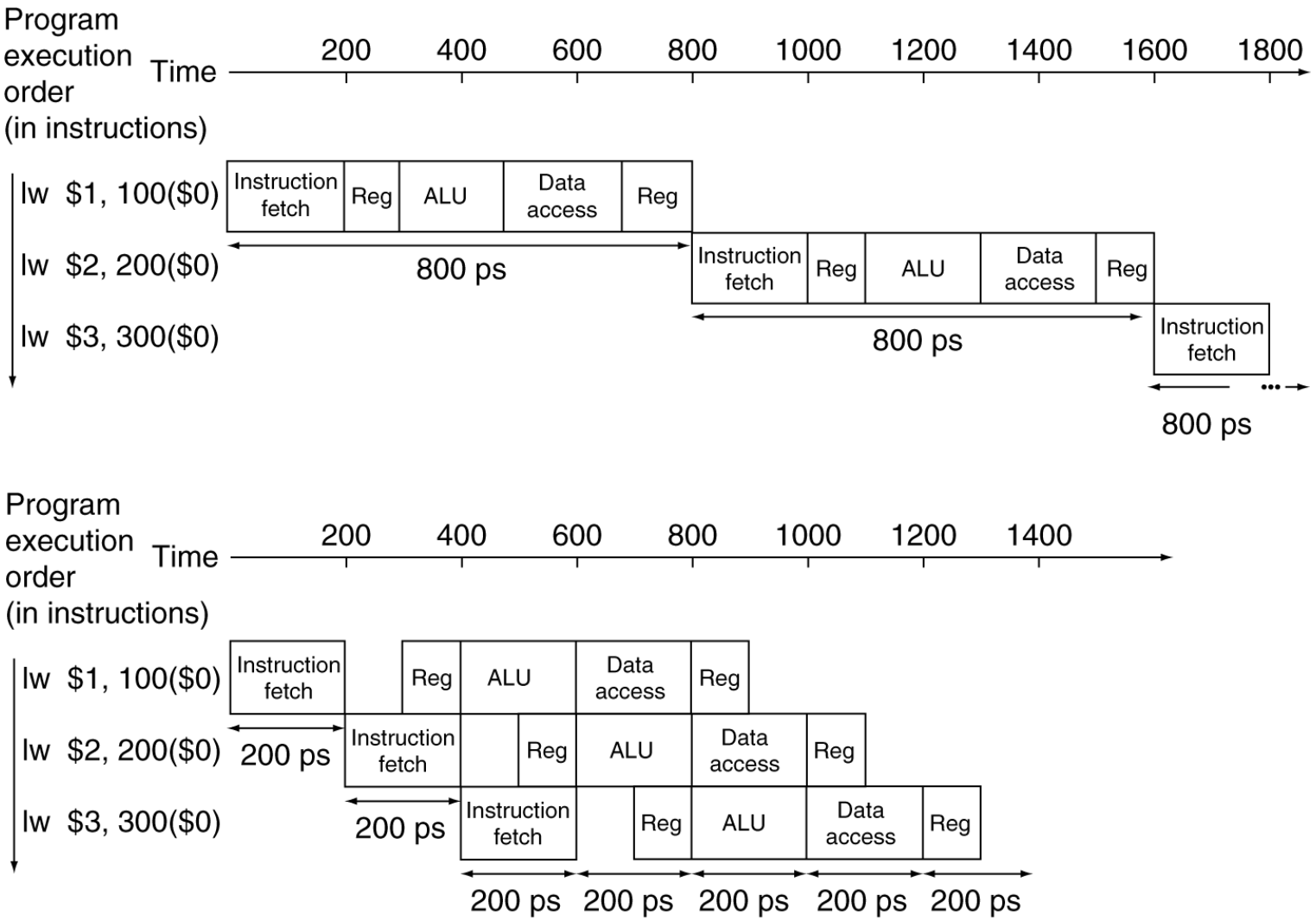
- Consider: **`lw, sw, add, sub, and, or, slt, beq`**
- Operation times: memory, ALU 200 ps, register 100 ps

Clock rate determined by slowest stage.

Pipelined

Instr	Instruction fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

4.5 Single Cycle vs. Pipelined Timeline



4.5 Designing Instruction Sets for Pipelining

- All MIPS instructions are the same length.
- MIPS has only a few instruction formats.
- Memory operands appear only in lw/sw
- Operands must be aligned in memory.

4.5 Pipeline Hazards

- Structural Hazard - not enough _hardware_.
- Data Hazards – one _instruction_ needs the _results_ of another
- Control Hazard – _decisions_ aren't made
  - Conservative Approach – _stall_
  - Alternative Approach – _predict_

4.5 Data Hazards

- A data hazard occurs when a result value has not yet been written to the register file.
- Consider

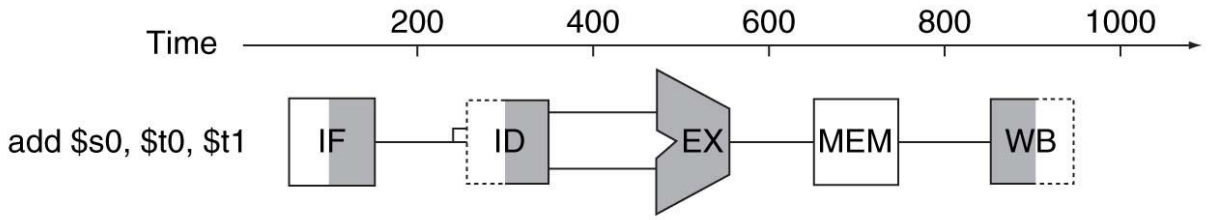
```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

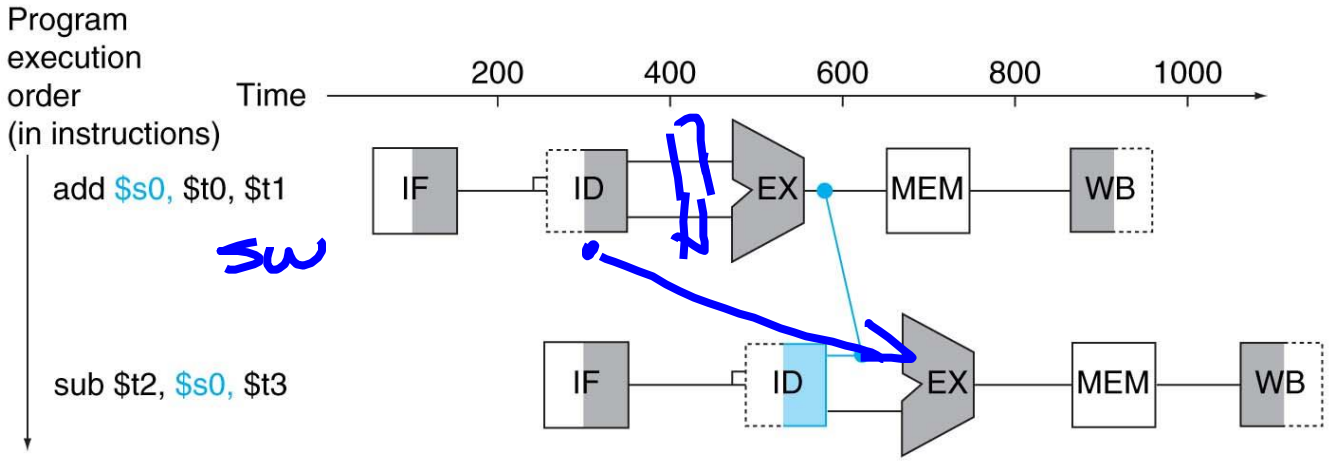
sw $s0, $t0, $t1
sub $t2, $s0, $t3

- Though the result is not stored until WB, it is available after the `add` has finished the EX stage, forward it to the right place.

4.5 Two Instruction Forwarding

- Forwarding paths are _valid_ only if the destination stage is _later_ than the source stage.

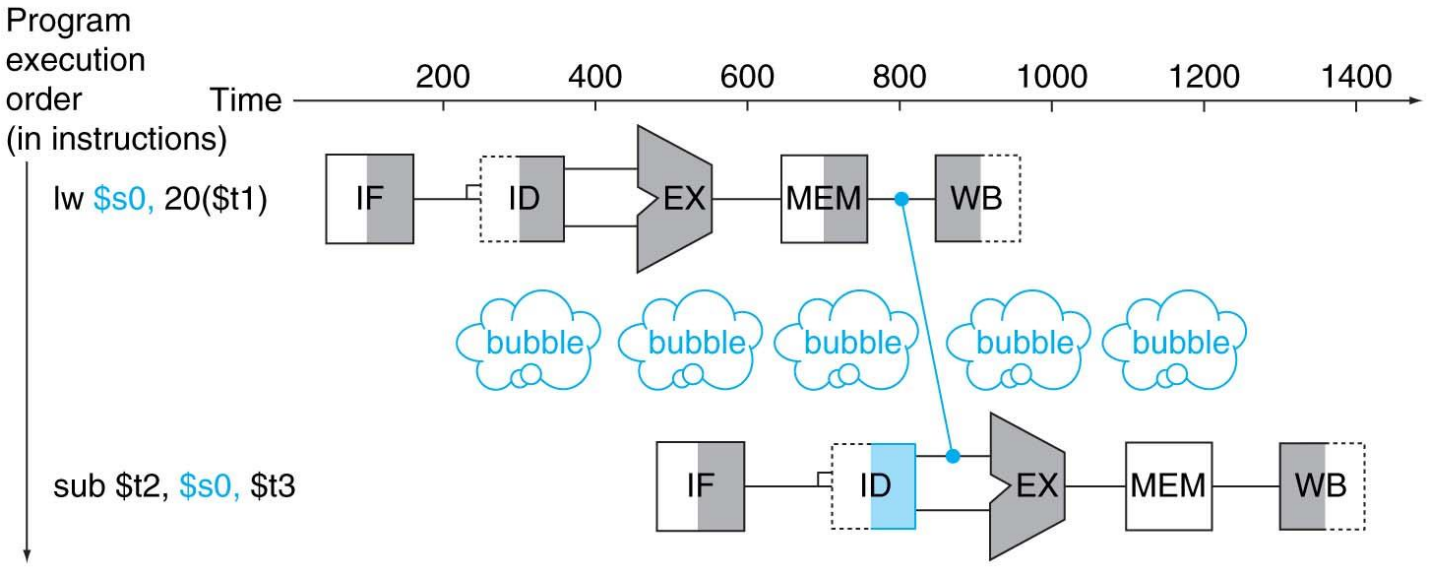




4.5 More on Forwarding

- Forwarding can't fix everything.
- Consider

```
lw      $s0, 20($t1)
sub     $t2, $s0, $t3
```

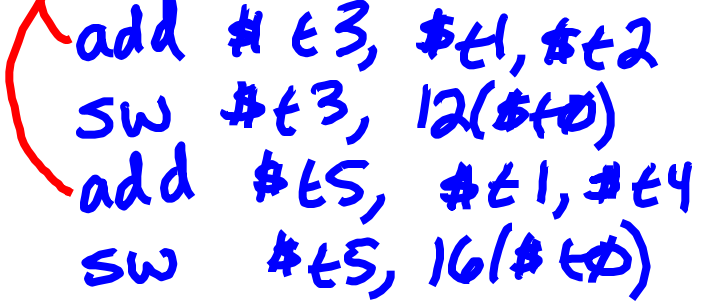


4.2 The Compiler Can Help

- Consider the following:

```
a = b + e;
c = b + f;
```

```
1 lw   $t1, 0($t0)
2 lw   $t2, 4($t0)
3 add  $t3, $t1, $t2
4 sw   $t3, 12($t0)
5 lw   $t4, 8($t0)
6 add  $t5, $t1, $t4
7 sw   $t5, 16($t0)
```

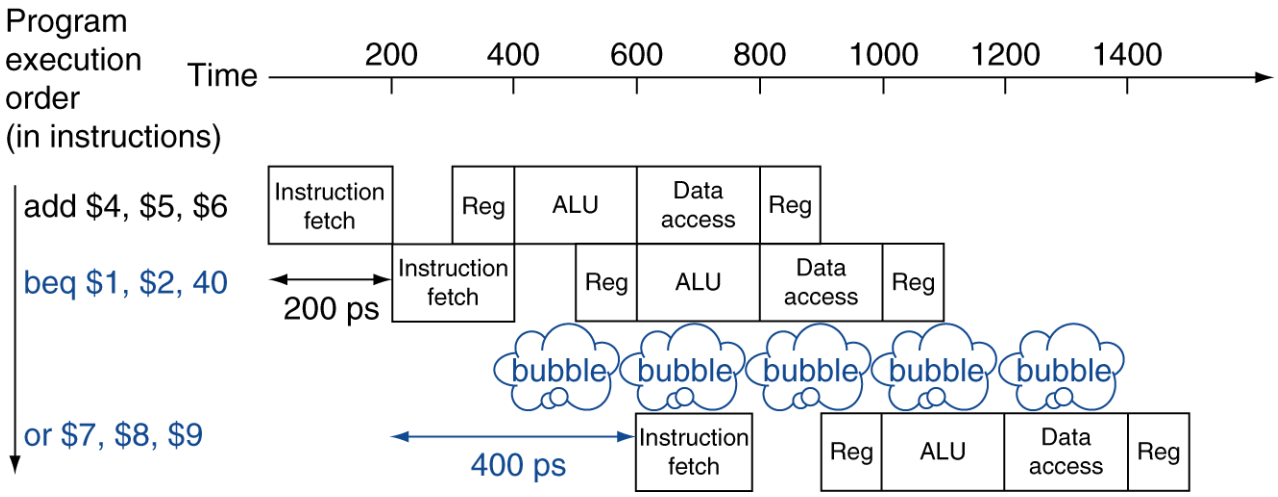


dependence
1-6
1-3
1-6
2-3

hazard?
handled by forwarding
no hazard problem

3-4 forwarding
5-6 problem
6-7 forwarding

4.2 Stalling on Control Hazards



Performance of "Stall on Branch"

SPEC

CPI = 1

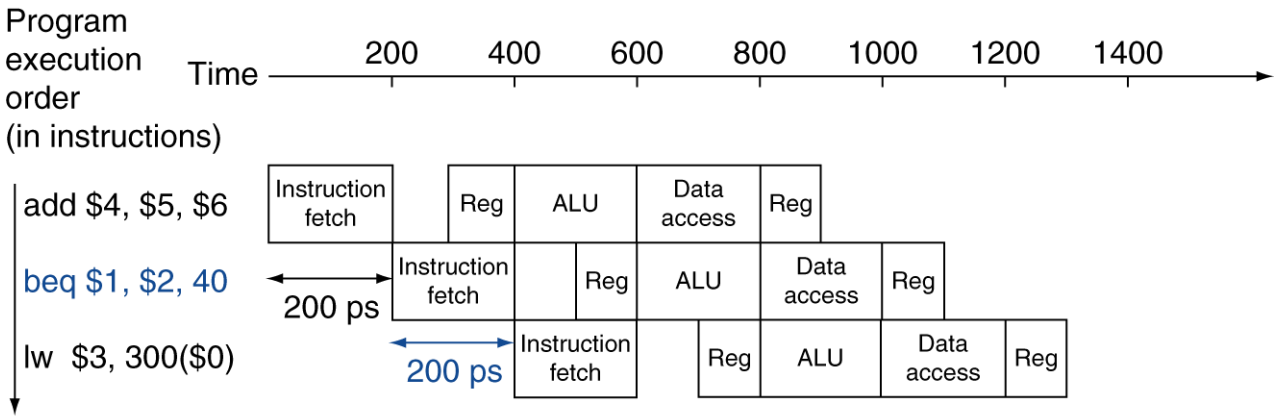
17% CPI branches = 2

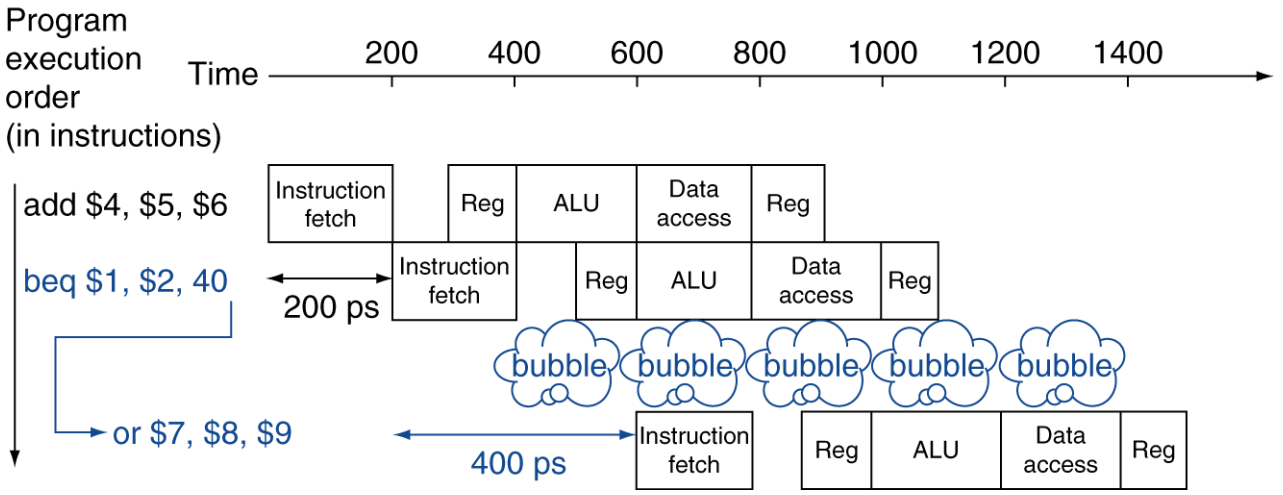
CPI = 0.83*1 + 0.17*2

= 1.17

slowdown of 1.17

4.2 Prediction for Control Hazards

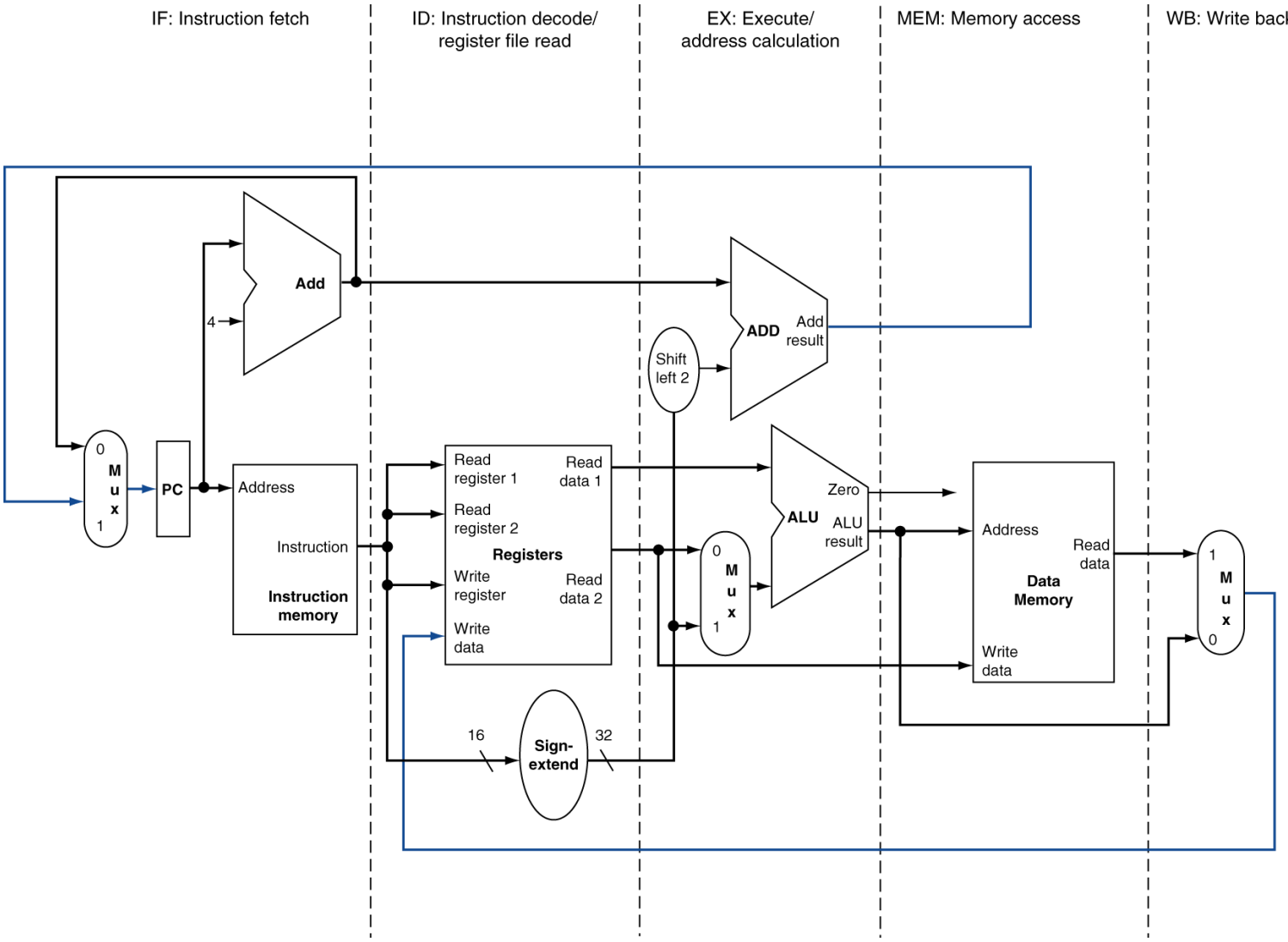




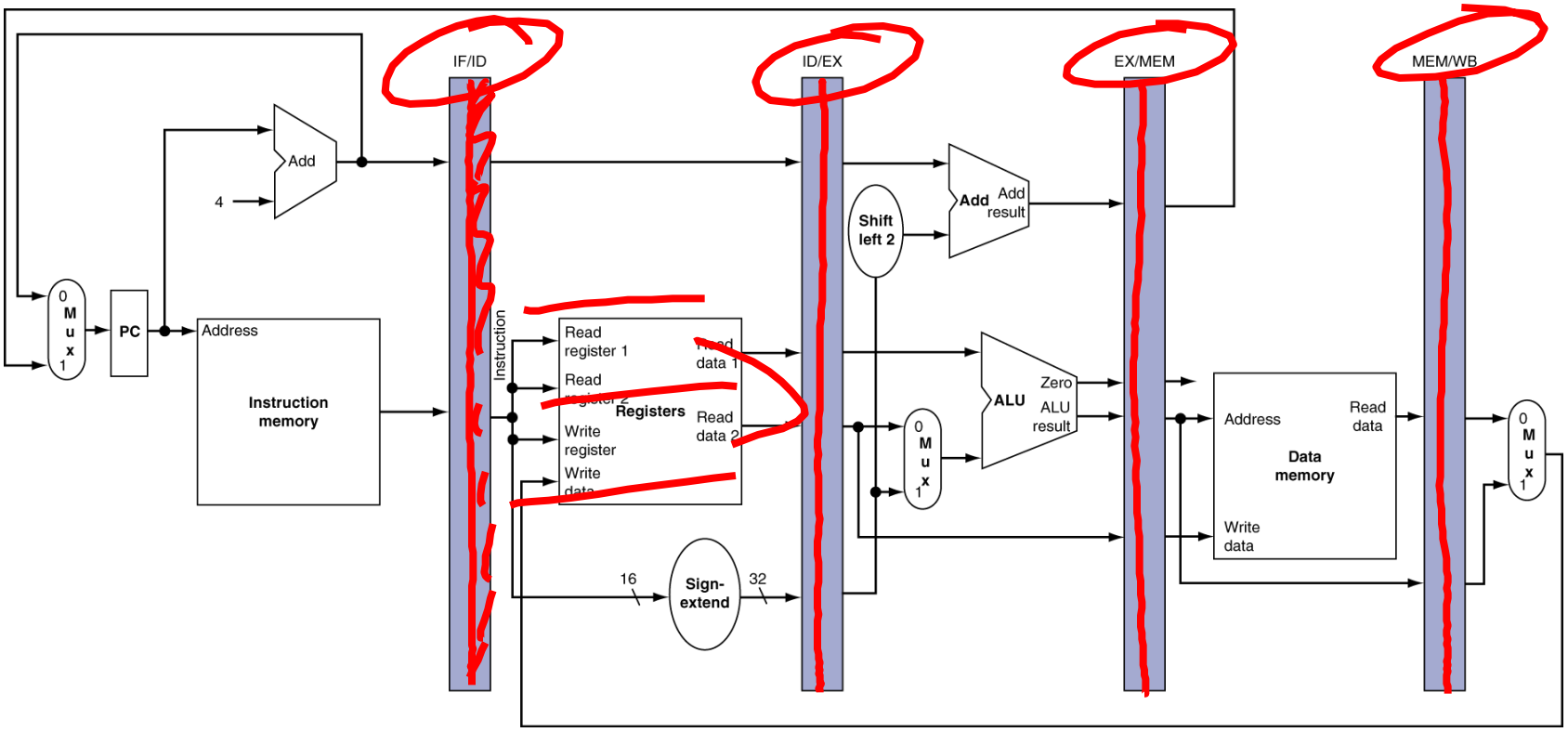
4.5 More Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - Record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

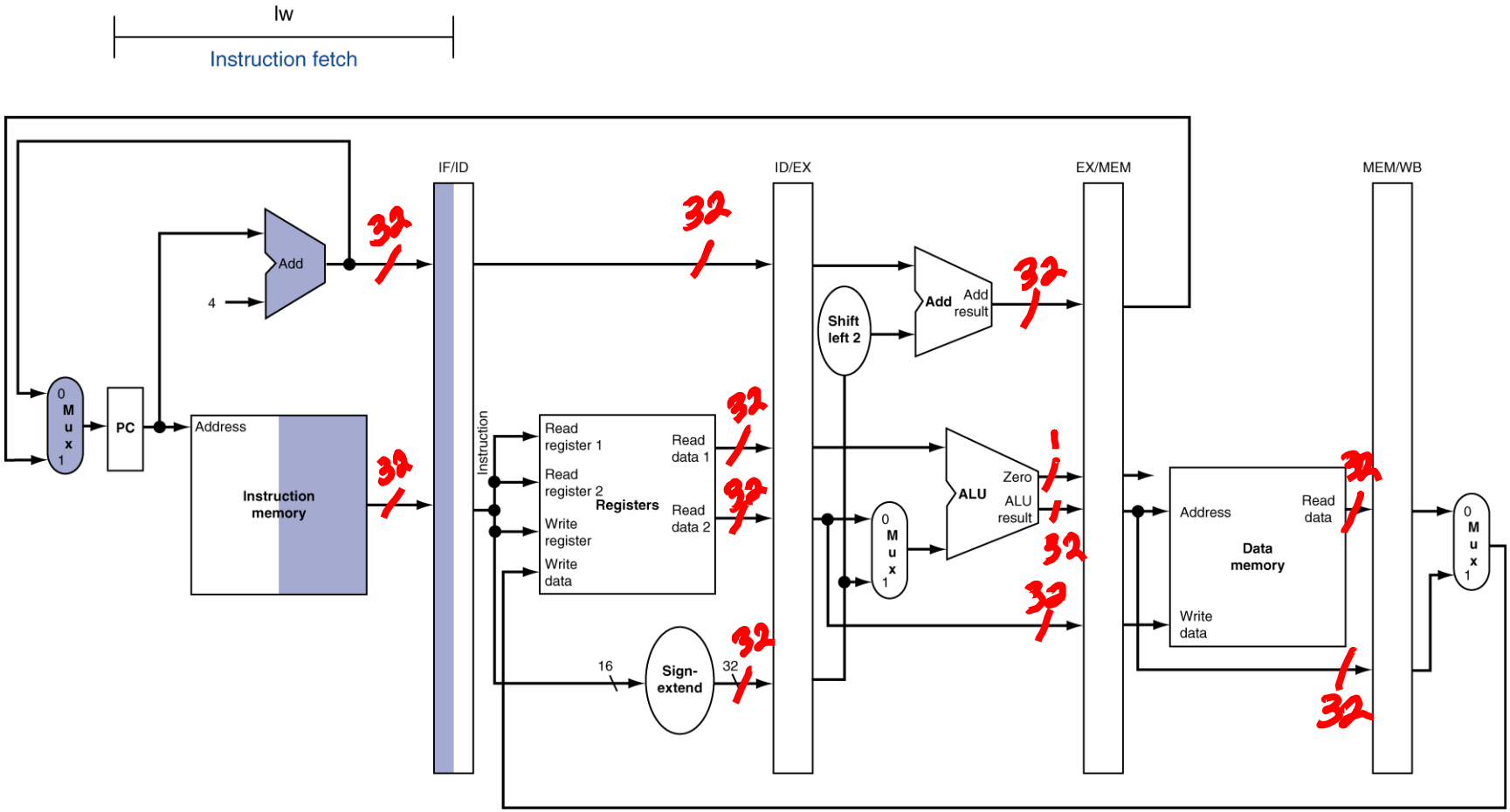
4.6 Identifying the Pipeline Stages



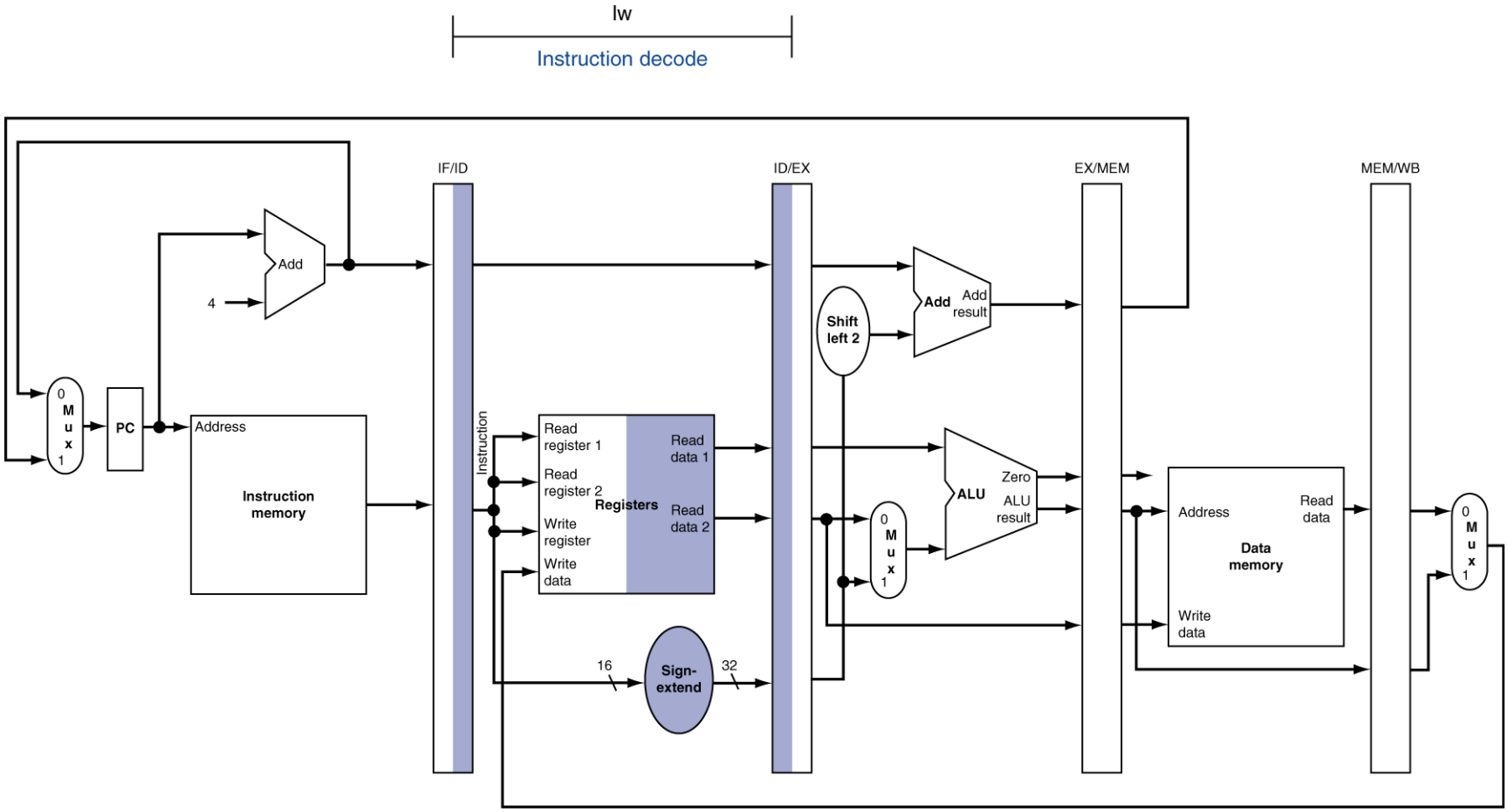
4.6 Adding Pipeline Registers



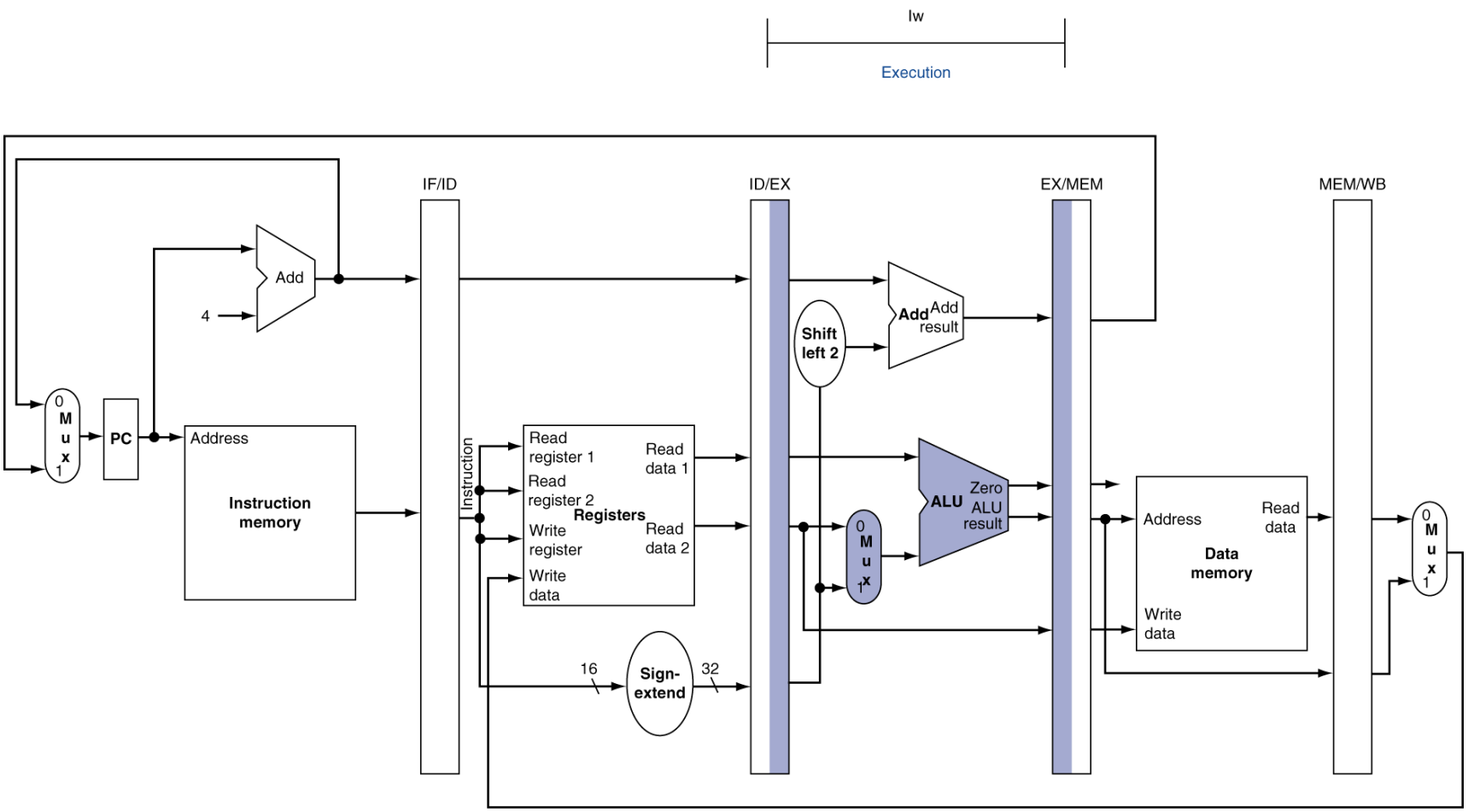
4.6 `lw` Instruction Execution: IF Stage



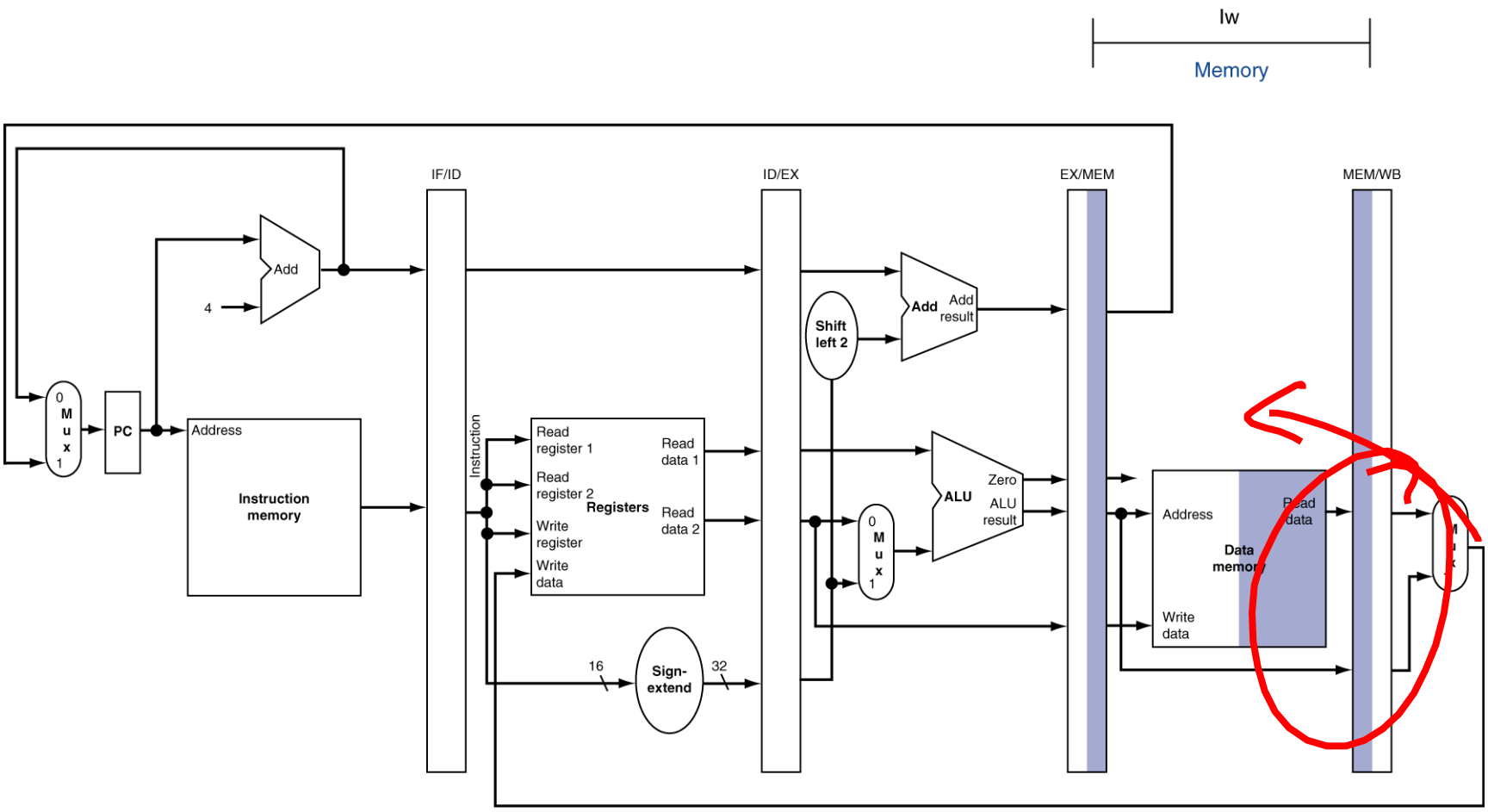
4.6 `lw` Instruction Execution: ID Stage



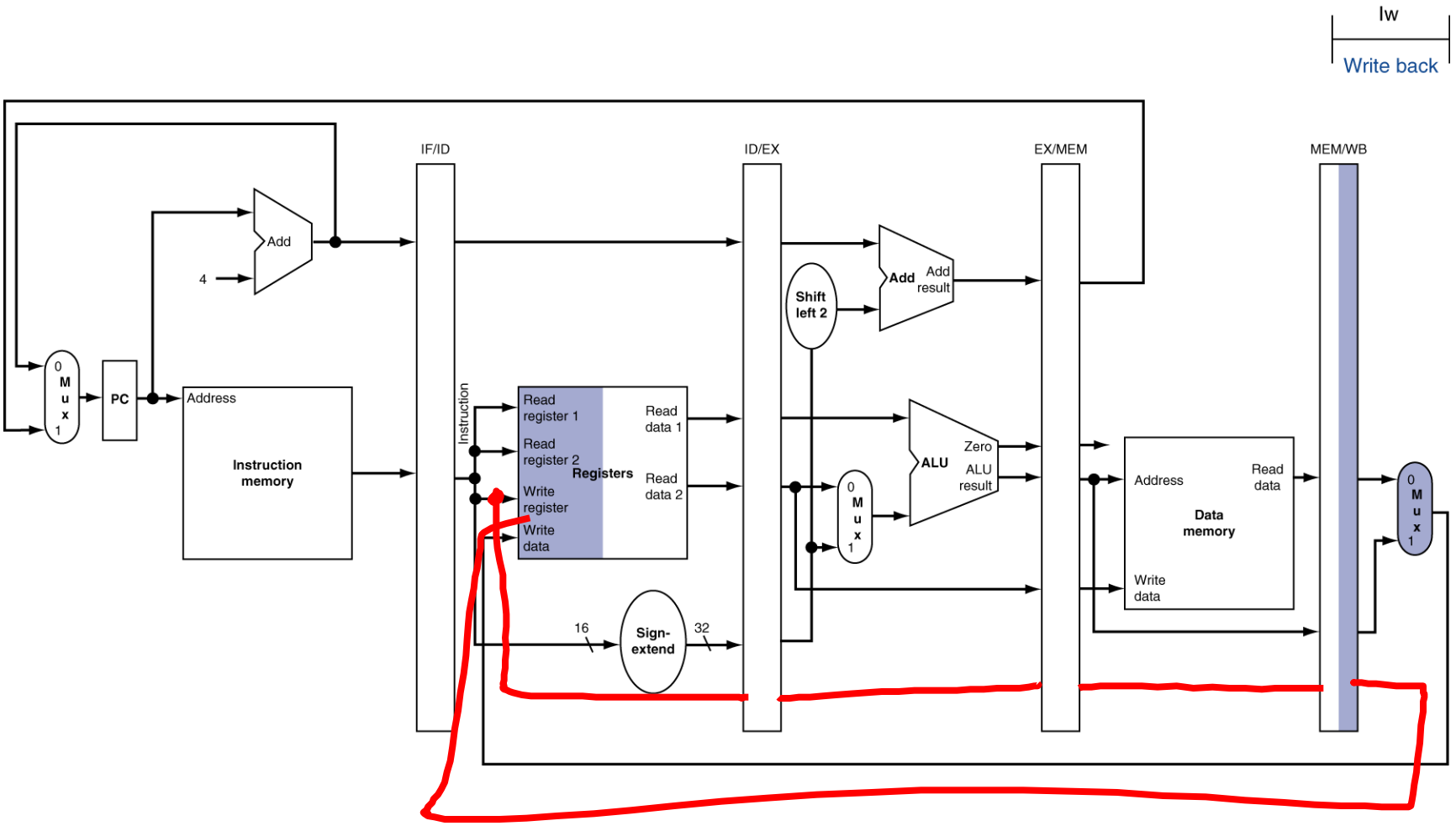
4.6 `lw` Instruction Execution: EX Stage



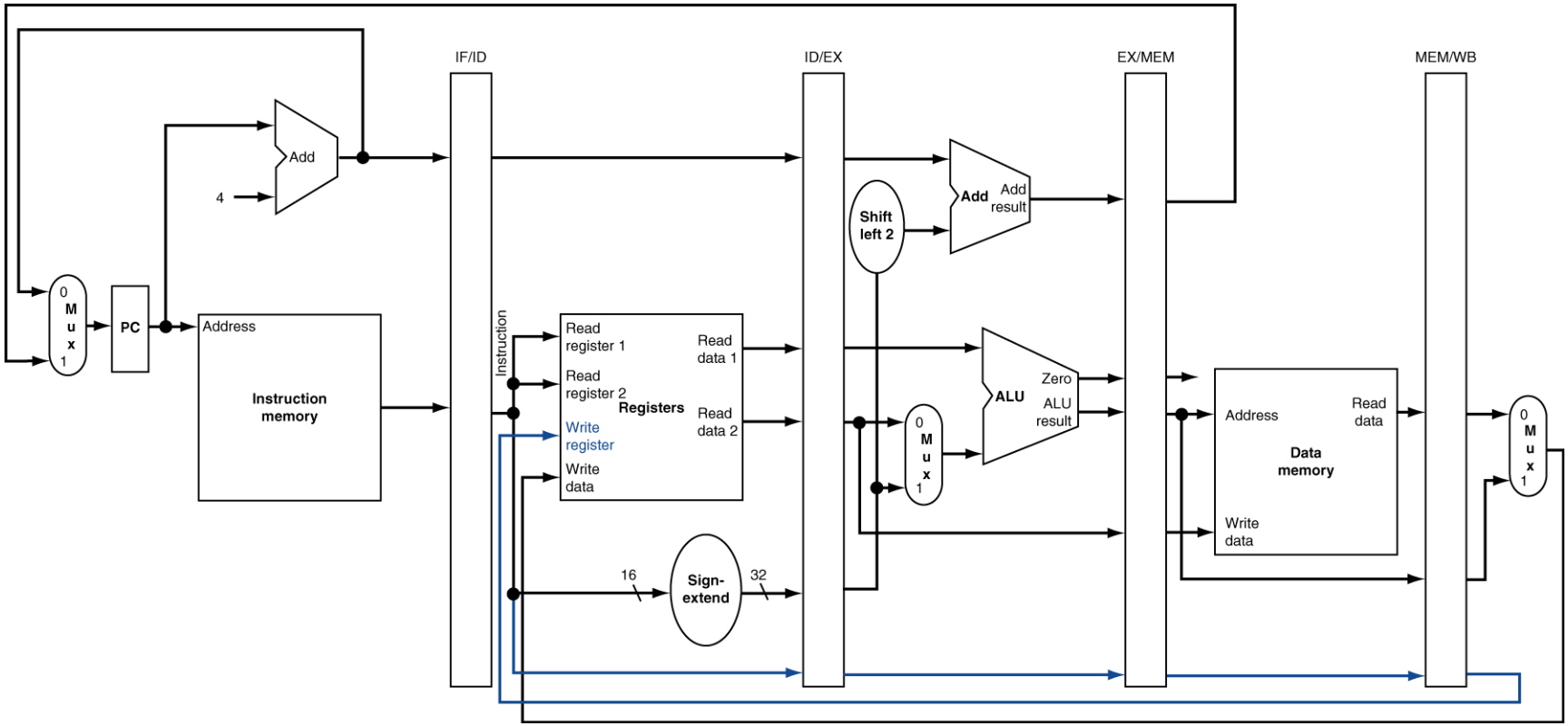
4.6 `lw` Instruction Execution: MEM Stage



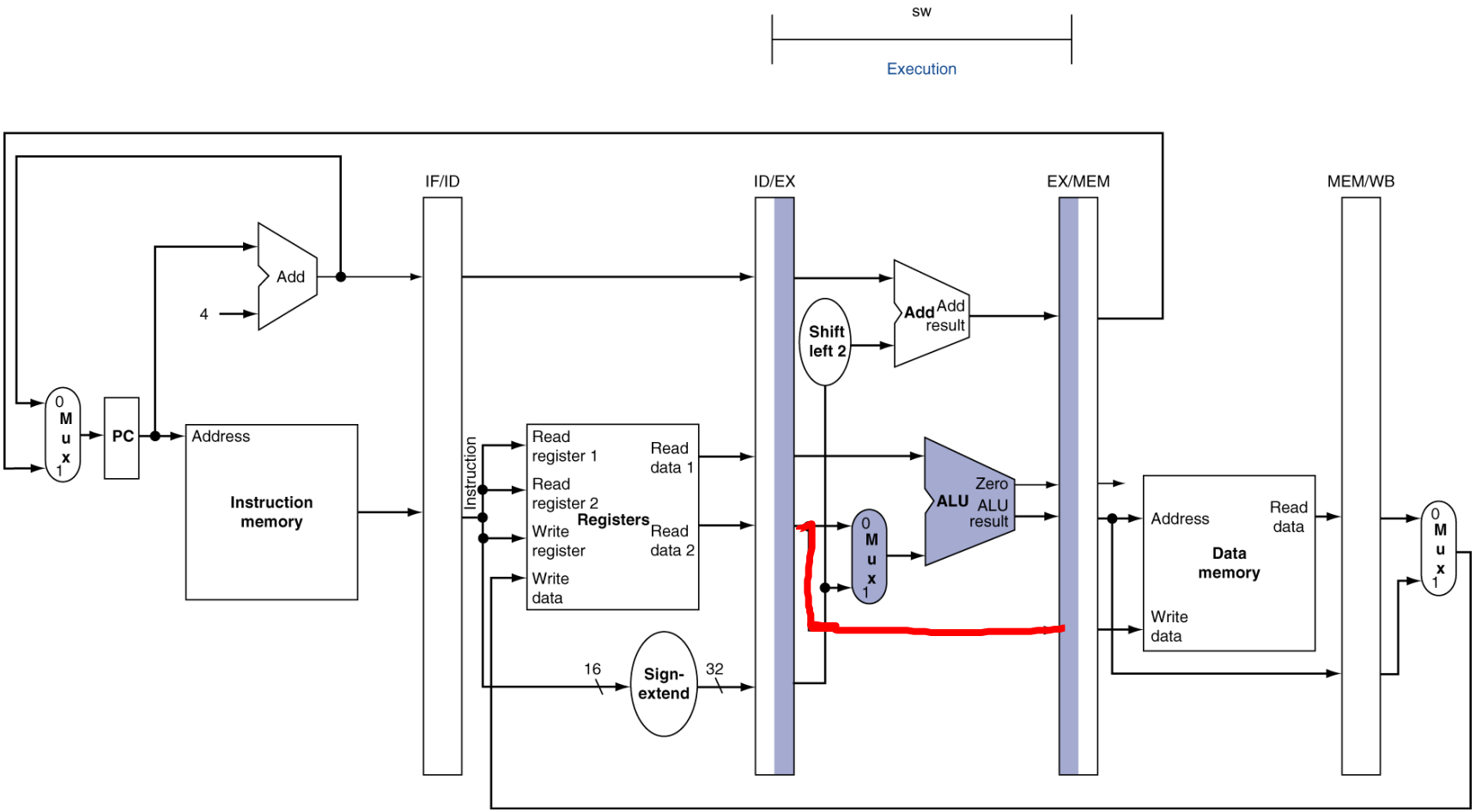
4.6 `lw` Instruction Execution: WB Stage



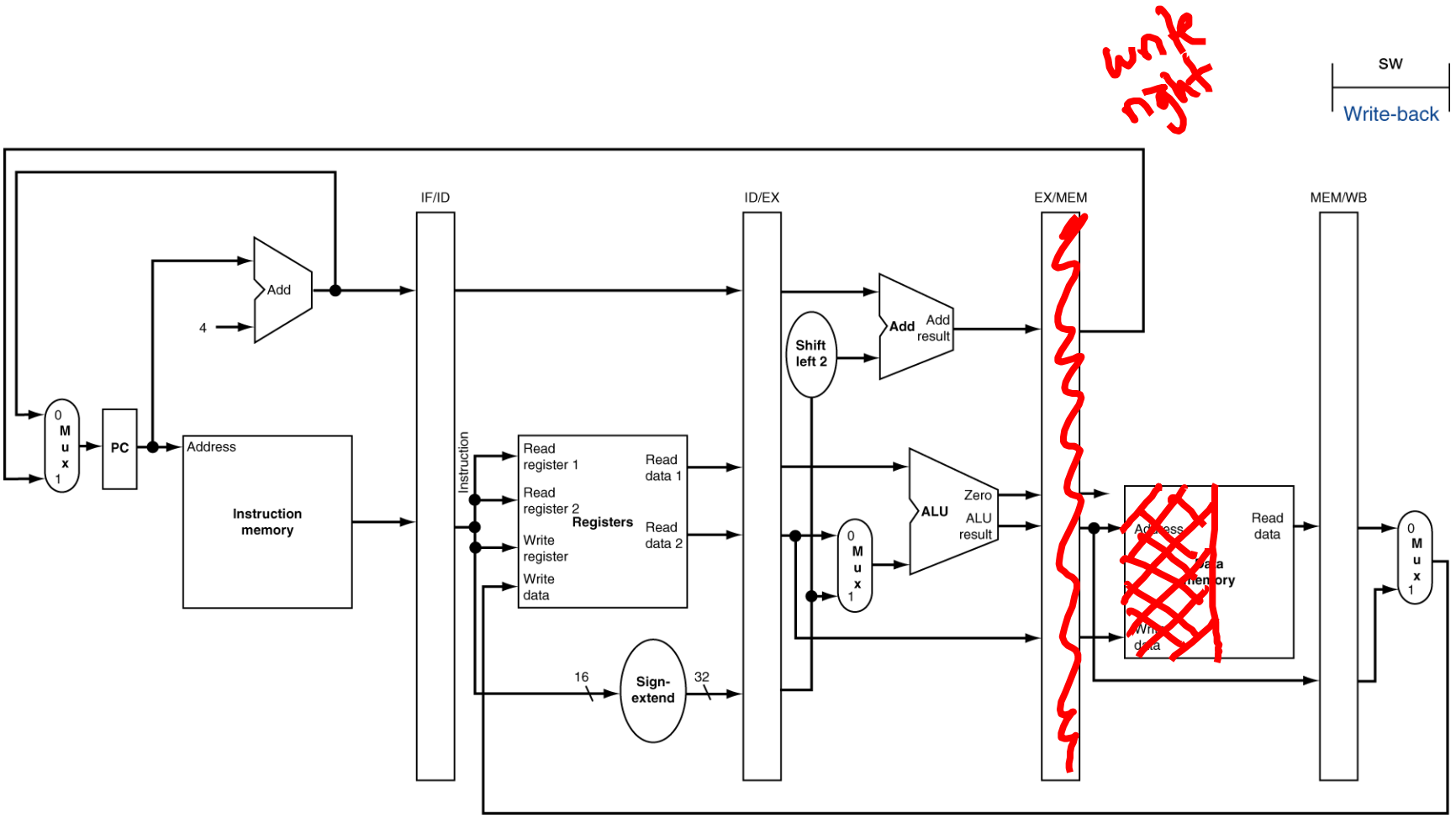
4.6 Corrected Datapath for `lw`



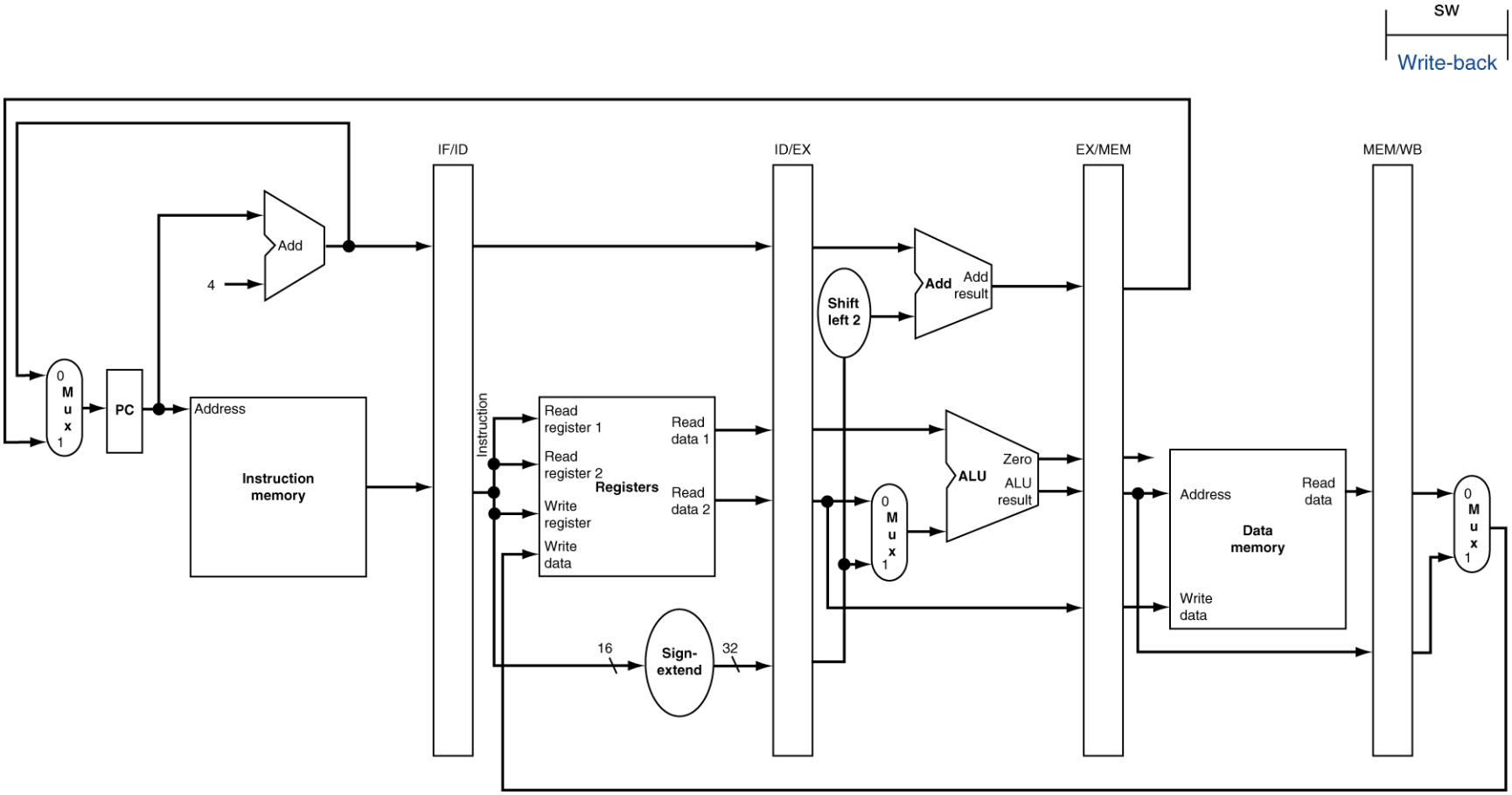
4.6 `sw` Instruction Execution: EX Stage



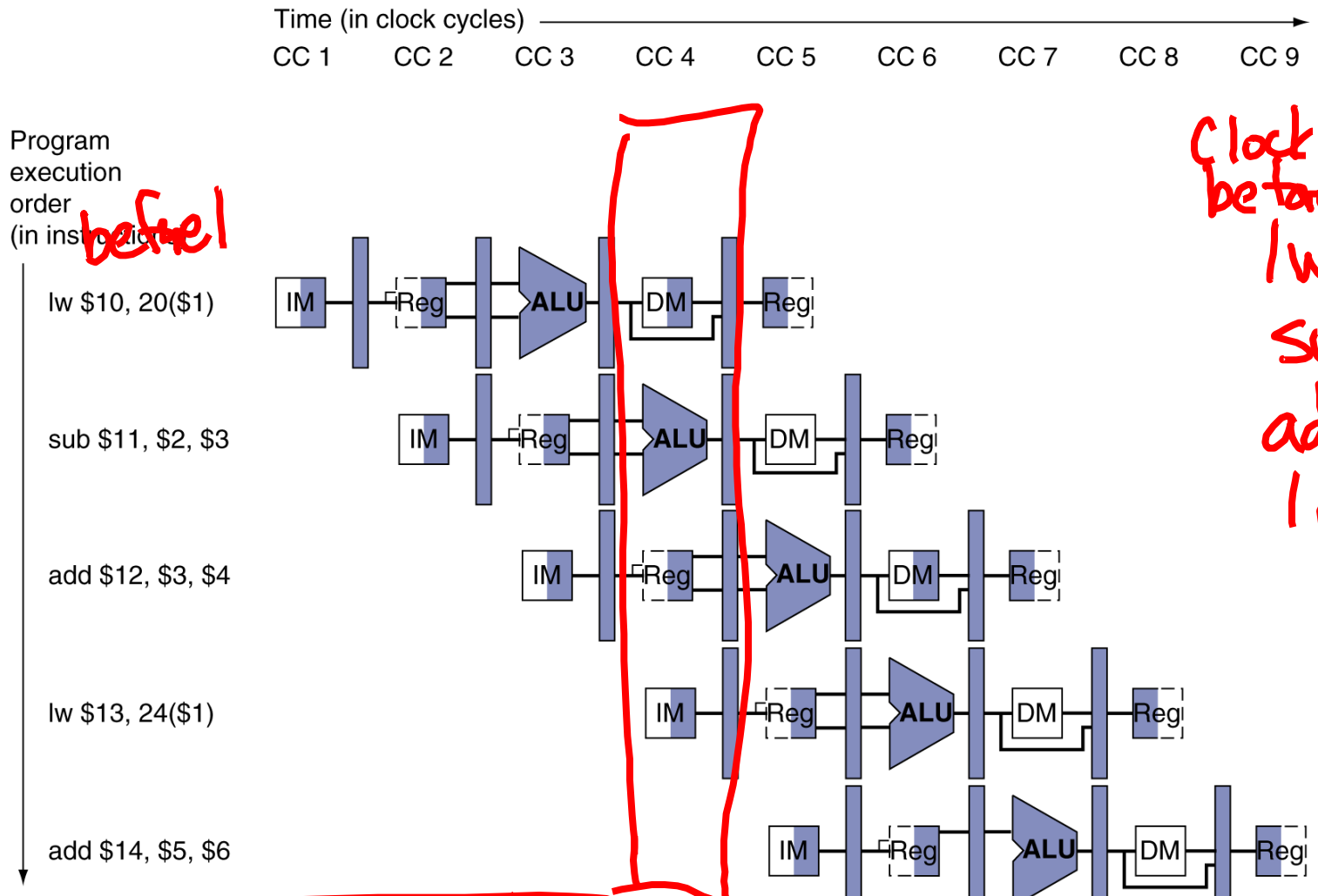
4.6 sw Instruction Execution: MEM Stage



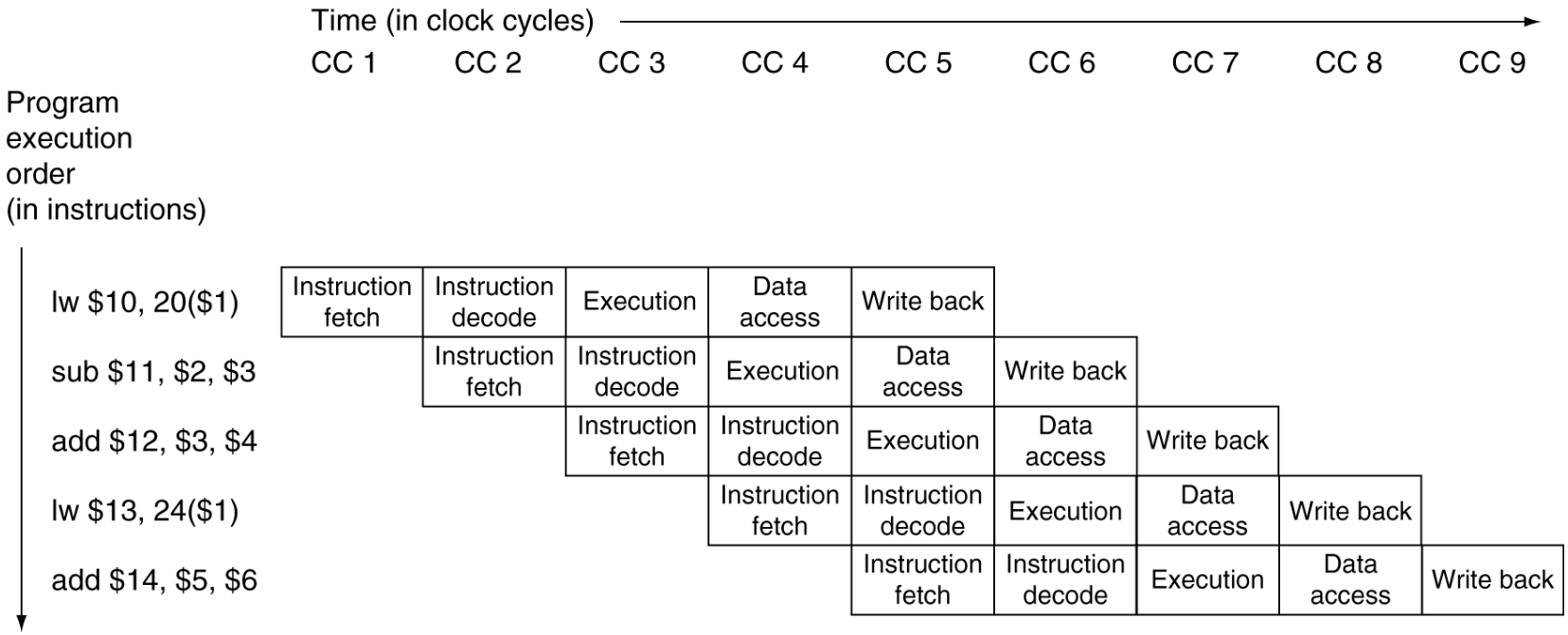
4.6 `sw` Instruction Execution: WB Stage



4.6 Stylized Multiple Clock Cycle Diagrams



4.6 Traditional Multiple Clock Cycle Diagrams



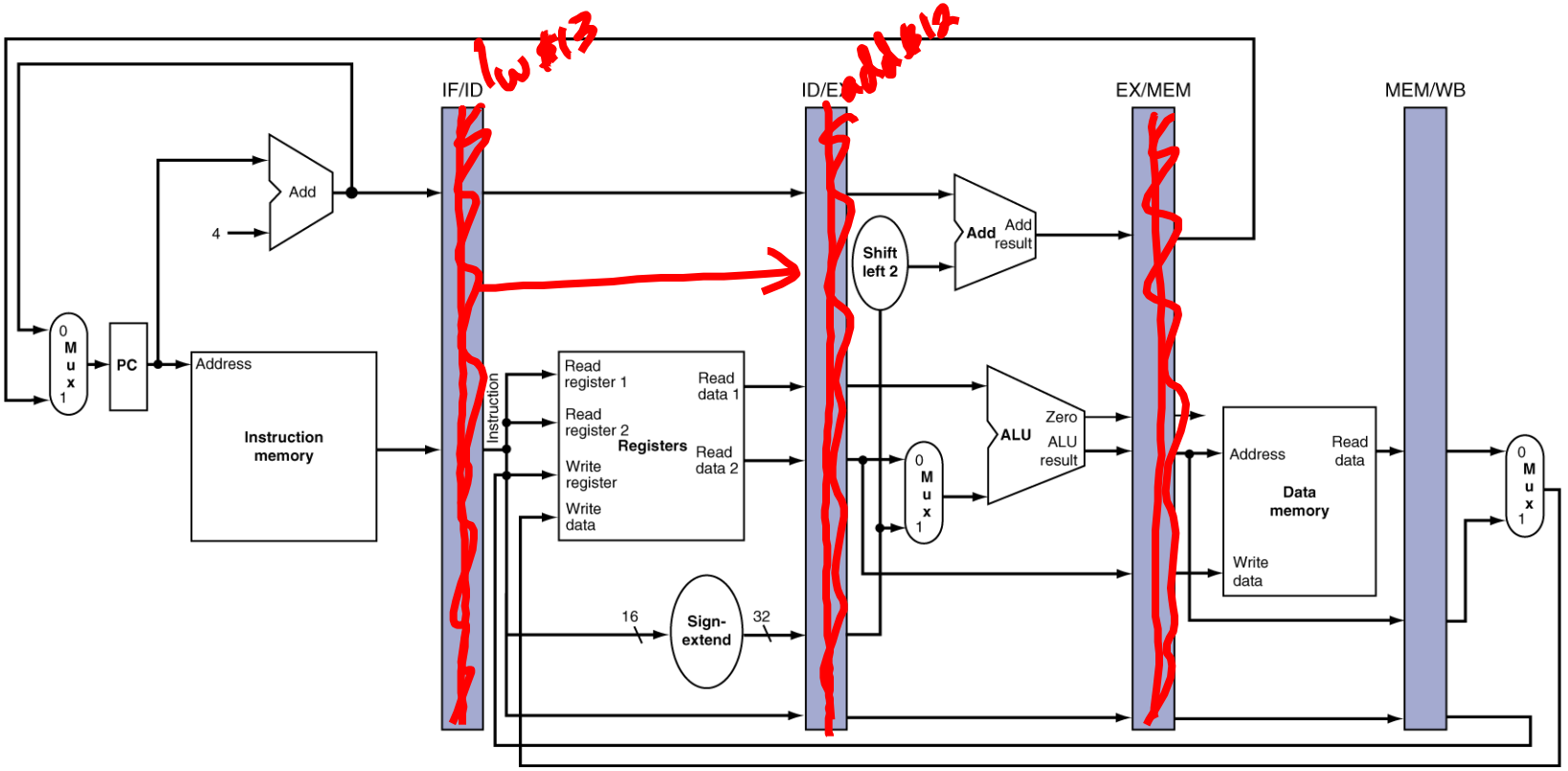
4.6 Alternate Multiple Clock Cycle Diagrams

Cycle	IF	ID	EX	MEM	WB
1	lw $10	ahead1	ahead2	ahead3	ahead4
2	sub $11	lw $10	ahead1	ahead2	ahead3
3	add $12	sub $11	lw $10	ahead1	ahead2
4	lw $13	add $12	sub $11	lw $10	ahead1
5	add $14	lw $13	add $12	sub $11	lw $10
6	after1	add $14	lw $13	add $12	sub $11
7	after2	after1	add $14	lw $13	add $12
8	after3	after2	after1	add $14	lw $13
9	after4	after3	after2	after1	add $14

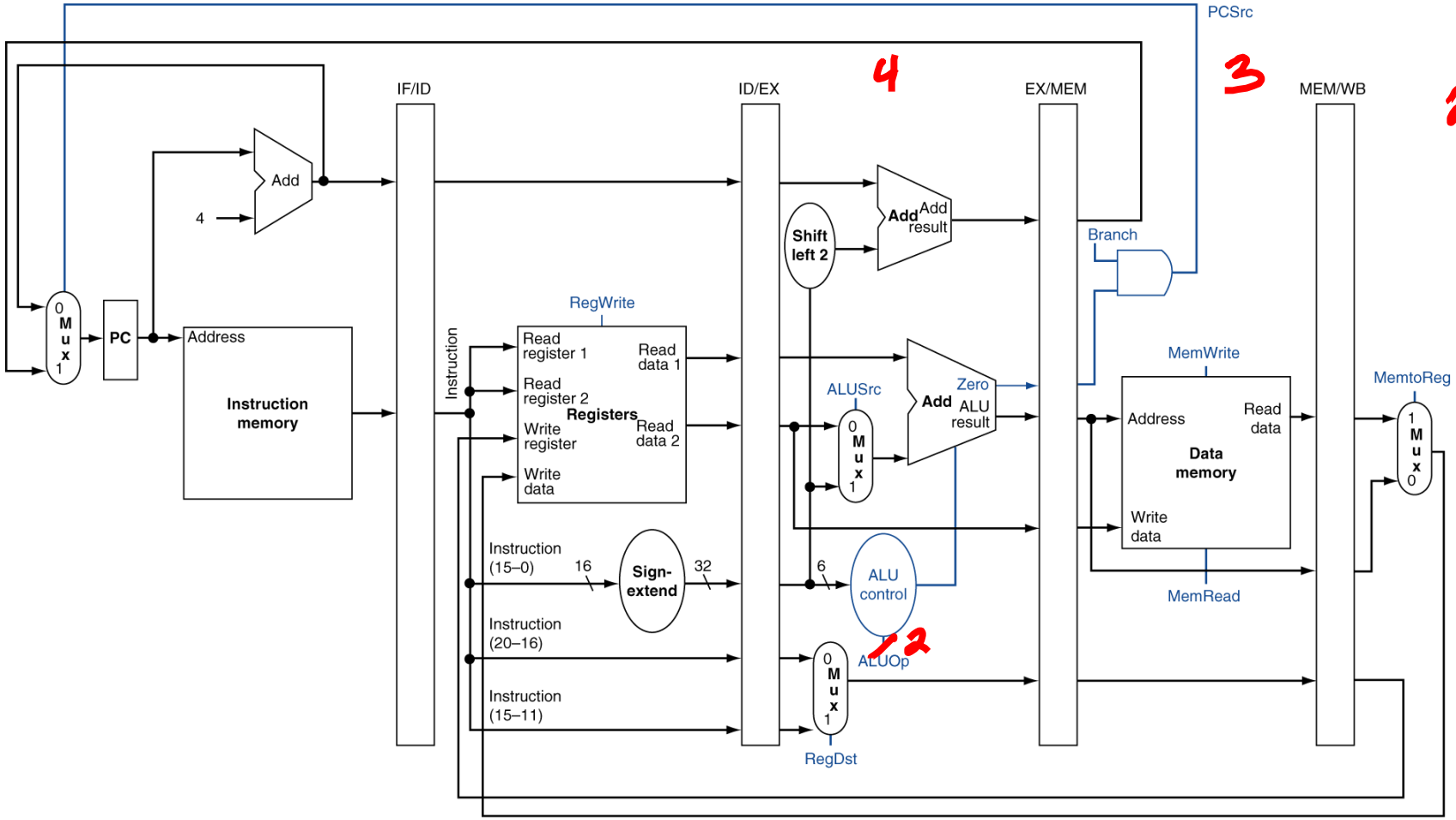
Cycle 5
IF/ID lw $13
ID/EX add $12
EX/MEM sub $11
MEM/WB lw $10

4.6 Cycle 5 Slice

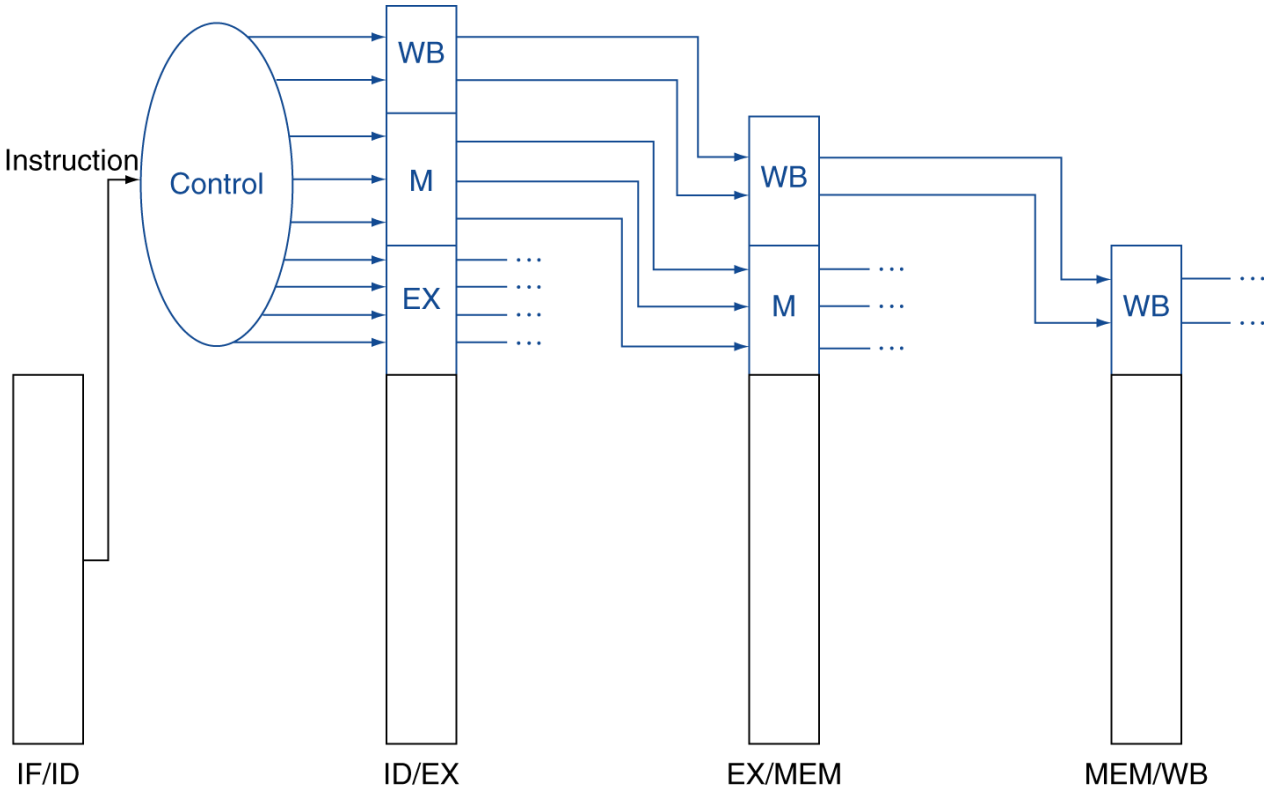
add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



4.6 Identifying Pipelined Control Lines

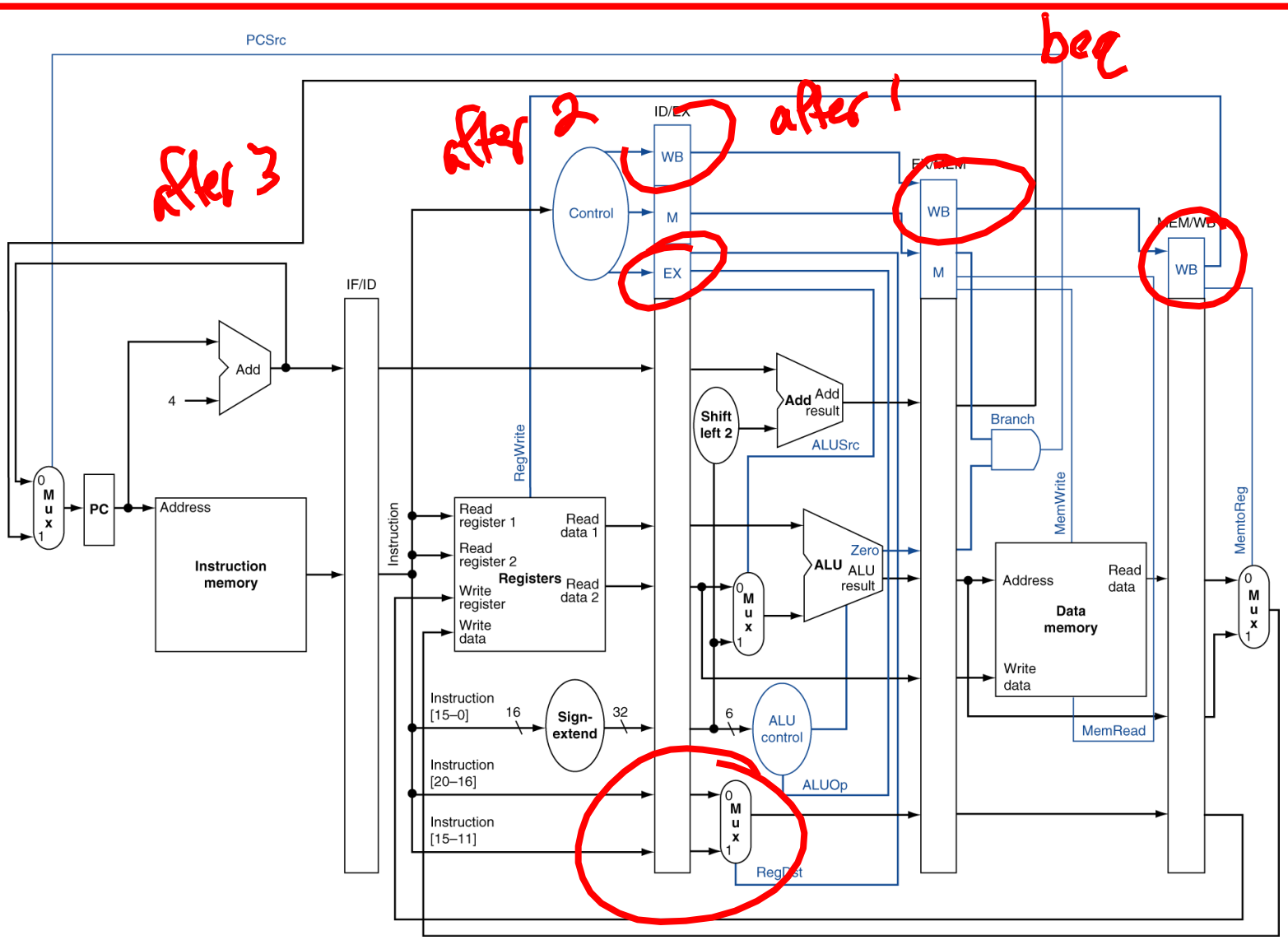


4.6 Generating and Saving Control Lines



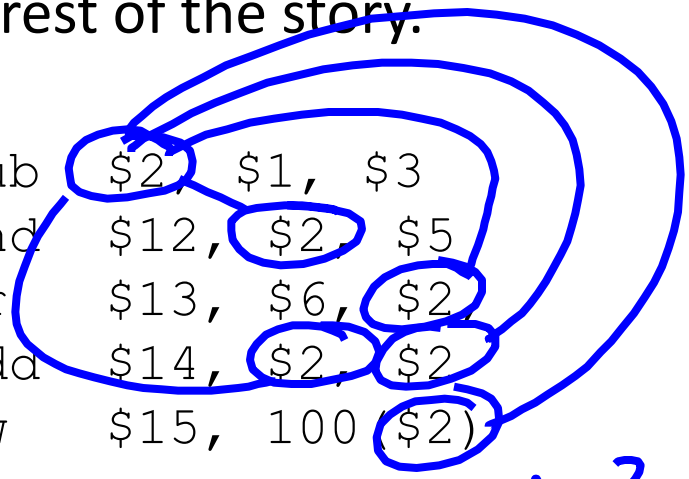
- EX ALUSrc ALUOp RegDst
- MEM Branch MemWrite MemRead
- WB MemtoReg RegWrite

4.6 Putting it all Together



4.7 Data Dependencies

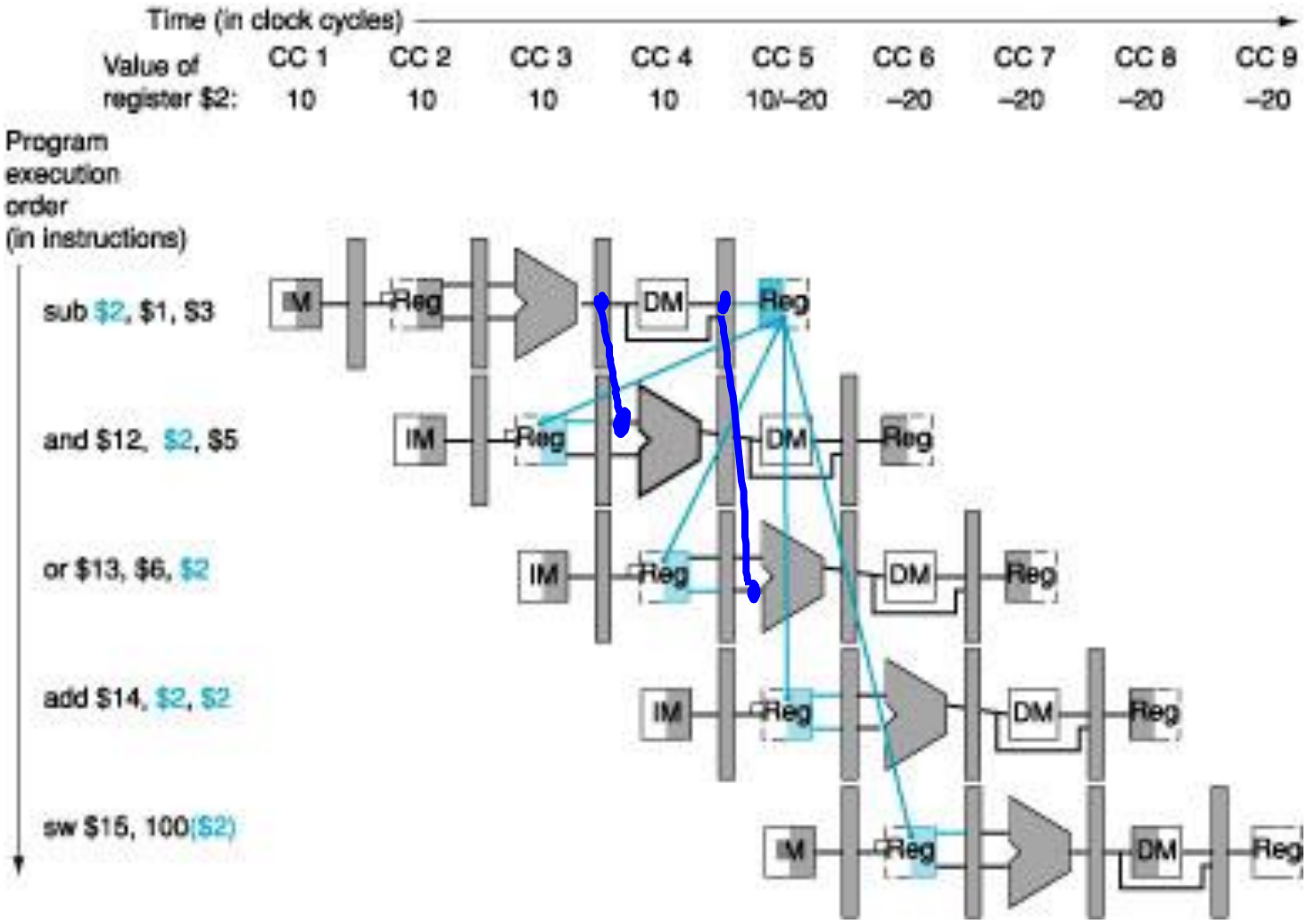
In the previous example, there were no data dependencies.
Now, the rest of the story.



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

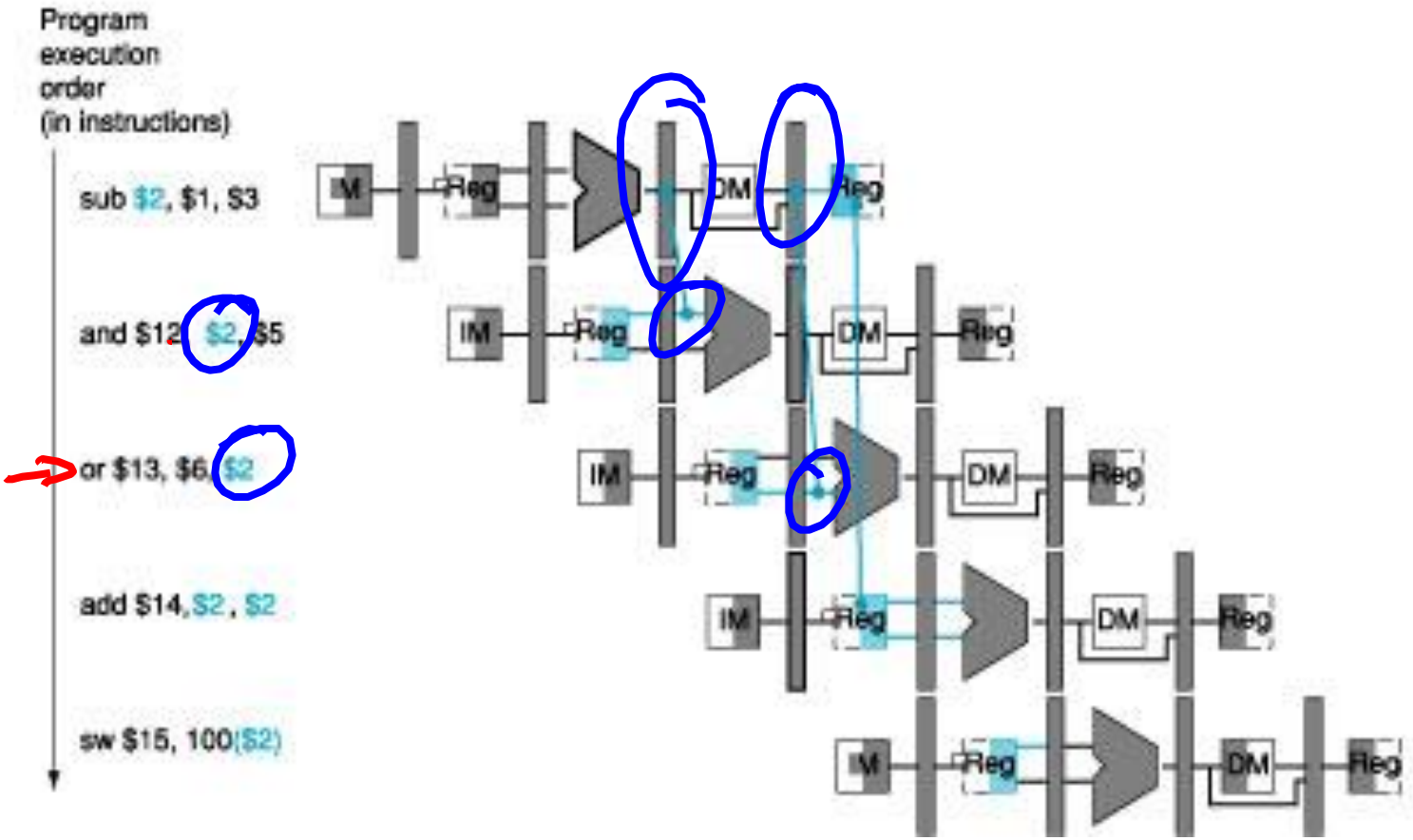
Dependences	Hazards?	Forwarding?
1 + 2	yes	yes
1 + 3	yes	yes
1 + 4	yes	yes
1 + 4	yes	yes
1 + 5	no	

4.7 Which Data Dependencies are Hazards

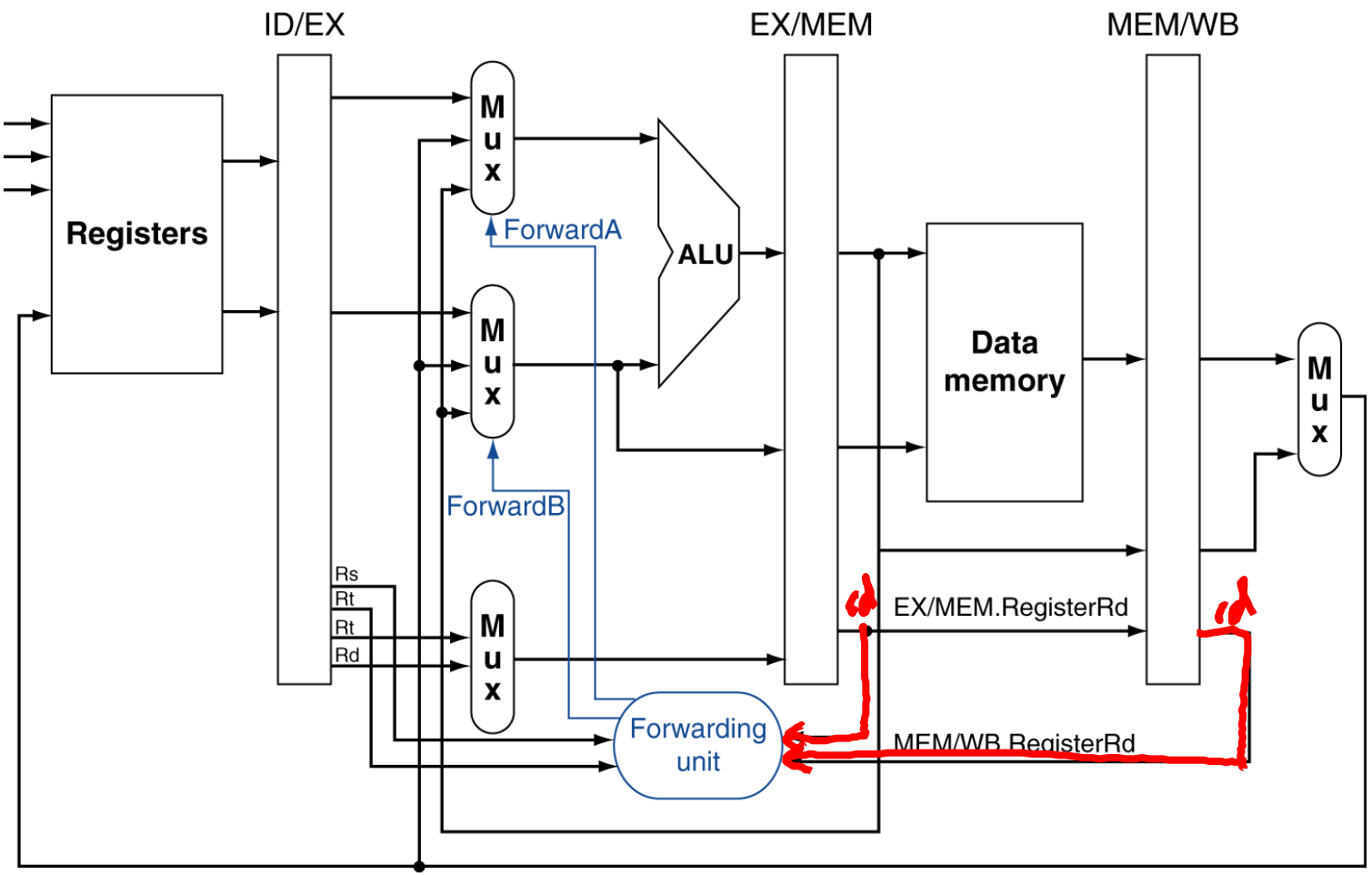


4.7 Forwarding in Action

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



4.7 Forwarding Paths



b. With forwarding

4.7 Forwarding Unit: Classifying Hazards

- Type 1 – The information needed in the EX stage by an instruction is the result of the instruction one stage ahead (found in the EX/MEM pipeline register)
  - a. The information is needed in R[rs]
  - b. The information is needed as R[rt]

A bus for ALU, first operand
B bus for ALU, second operand

- Type 2 – The information needed in the EX stage by an instruction is the result of the instruction two stages ahead (found in the MEM/WB pipeline register)
  - a. The information is needed in R[rs]
  - b. The information is needed as R[rt]

4.7 Forwarding Unit: Type 1 Hazards

Type 1 Hazard

If (ahead1 writes to register and (ahead1 doesn't have $zero as the destination register) and (ahead1 is writing to a register read by Current Instruction)) Forward from EX/MEM a) Forward to RegisterRs, b) Forward to RegisterRd

ahead1 writes to register

EX/MEM.RegWrite = '1'

ahead1 doesn't have $zero as the destination register

EX/MEM.RegisterRd ≠ 0

ahead1 is writing to a register read by Current Instruction

a) EX/MEM.RegisterRd = ID/EX.RegisterRs
b) EX/MEM.RegisterRd = ID/EX.RegisterRt

4.7 Forwarding Unit: Type 2 Hazards

Type 2 Hazard

If (ahead2 writes to register and (ahead2 doesn't have $zero as the destination register) and (ahead2 is writing to a register read by Current Instruction)) Forward from MEM/WB a) Forward to RegisterRs, b) Forward to RegisterRd

ahead2 writes to register

MEM/WB.RegWrite = '1'

ahead2 doesn't have $zero as the destination register

MEM/WB.RegisterRd ≠ 0

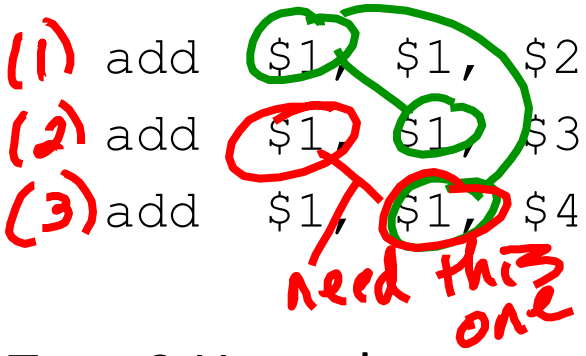
ahead2 is writing to a register read by Current Instruction

a) MEM/WB.RegisterRd = ID/EX.RegisterRs

b) MEM/WB.RegisterRd = ID/EX.RegisterRt

4.7 Double Data Hazard Jeopardy

Consider



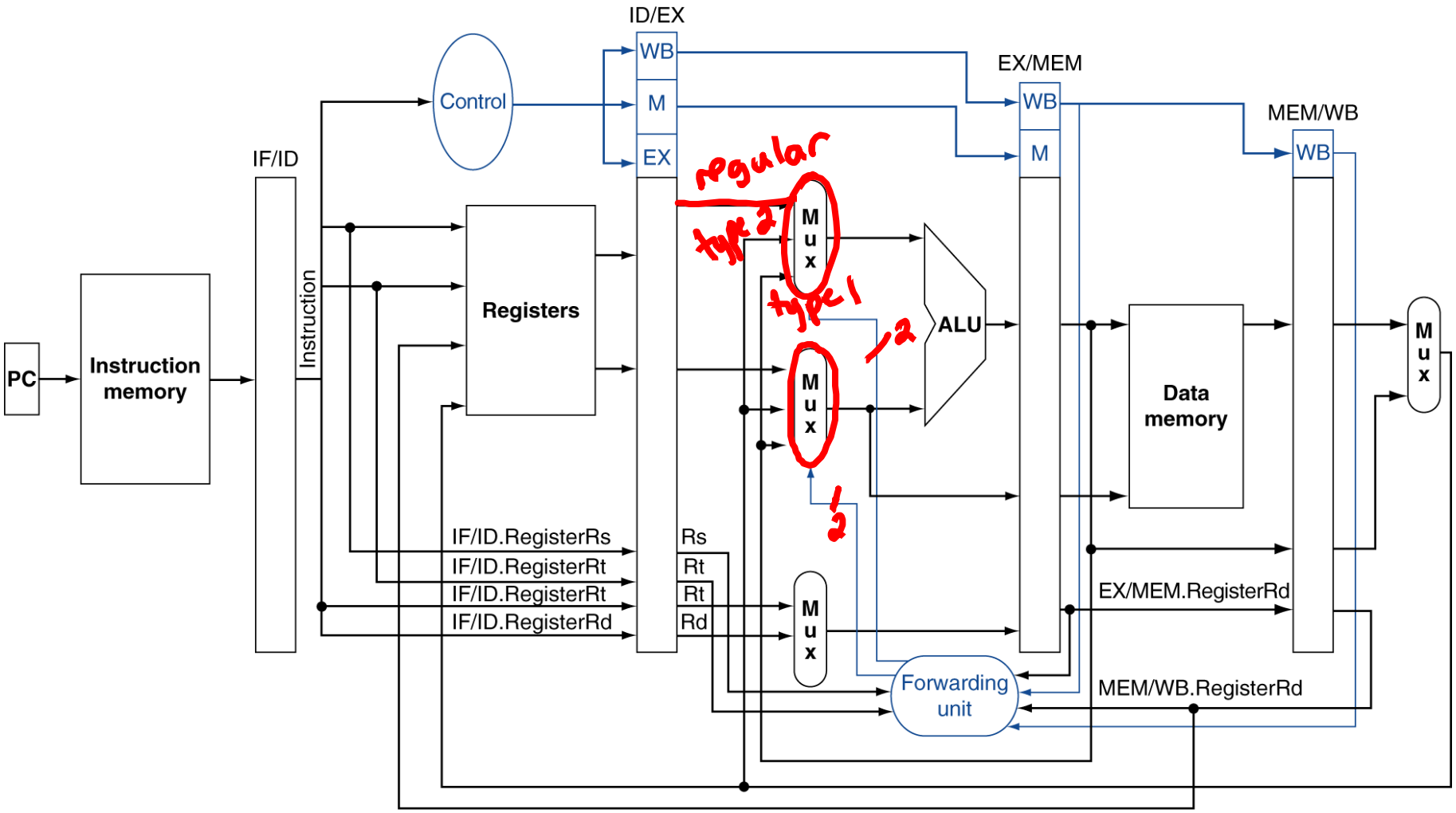
(2) sees a result from 1, forwarding

(3) sees Type 1 hazard and Type 2 hazard

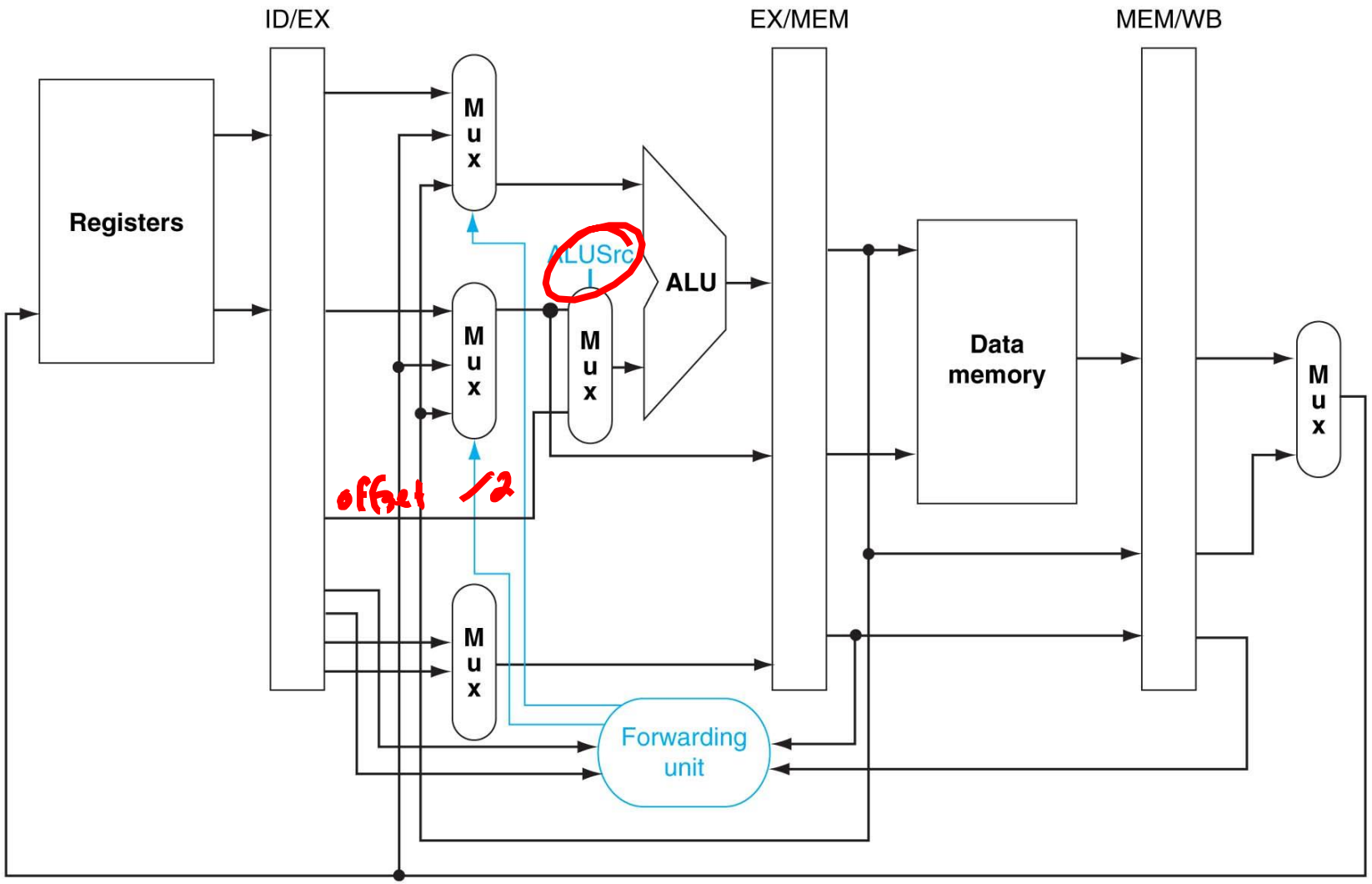
Type 2 Hazard

If (ahead2 writes to register and (ahead2 doesn't have $zero as the destination register) and (ahead2 is writing to a register read by Current Instruction) and (no Type 1 Hazard)) Forward from MEM/WB a) Forward to RegisterRs, b) Forward to RegisterRd

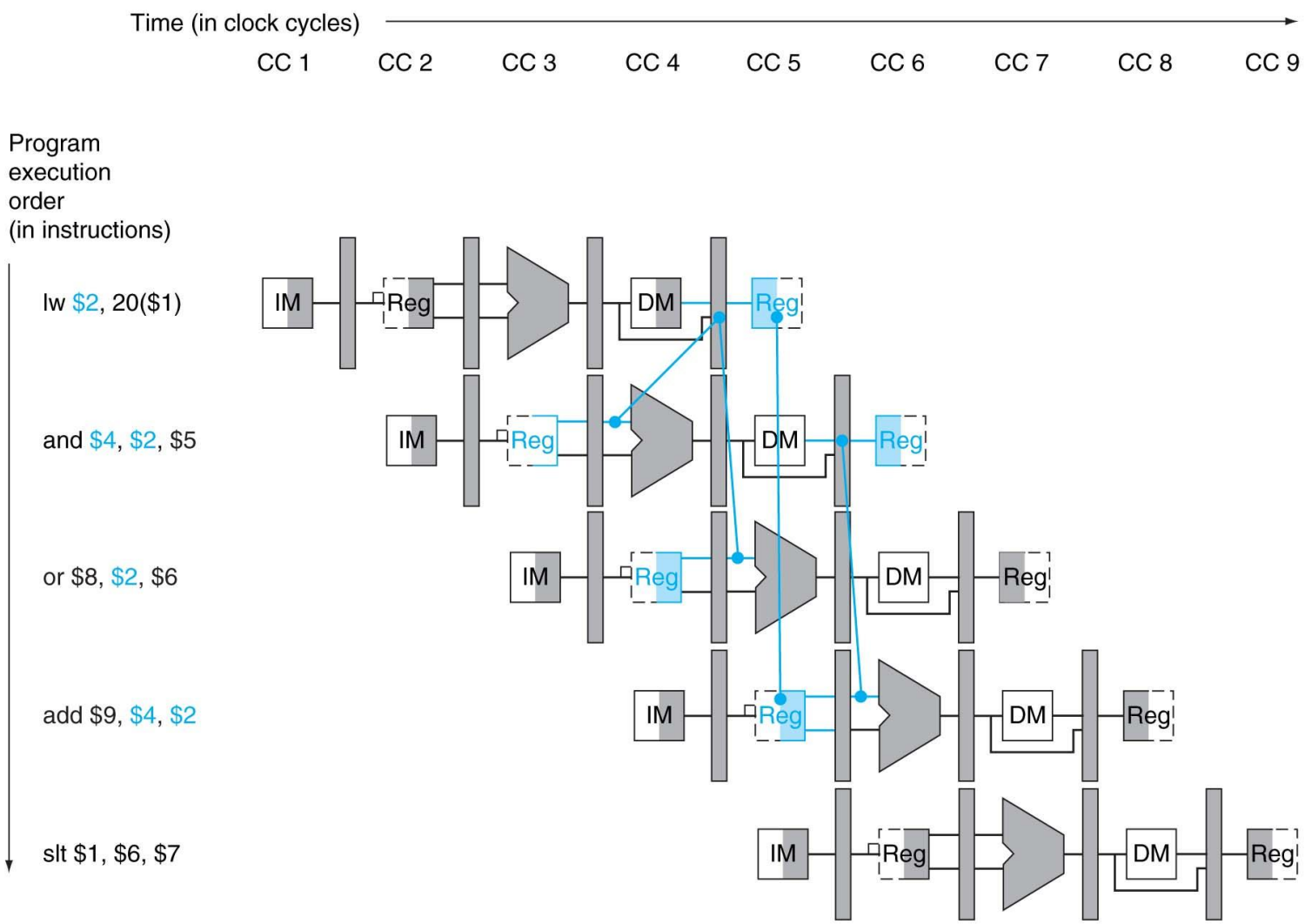
4.7 Forwarding Datapath with Control



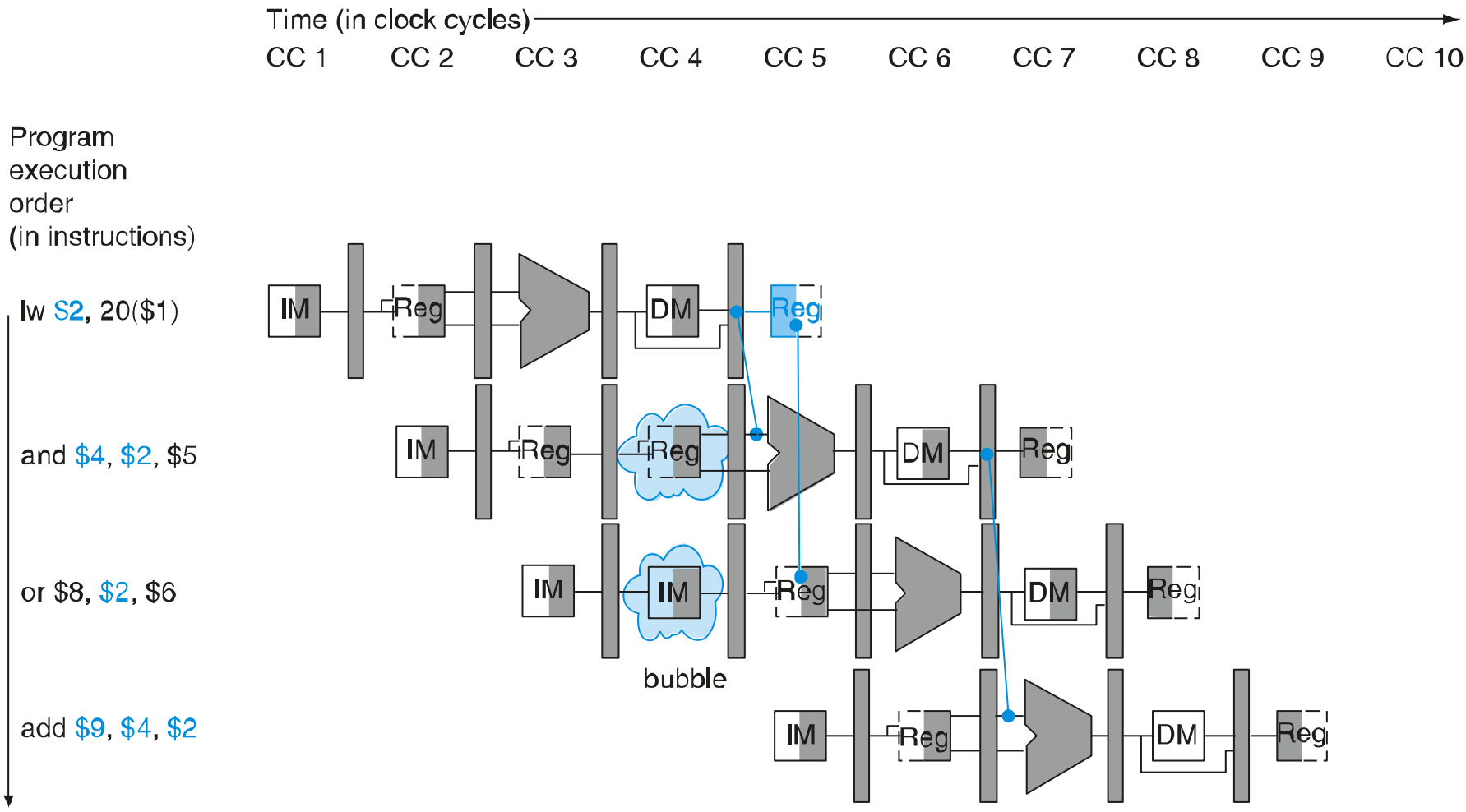
4.7 EX Forwarding Completed



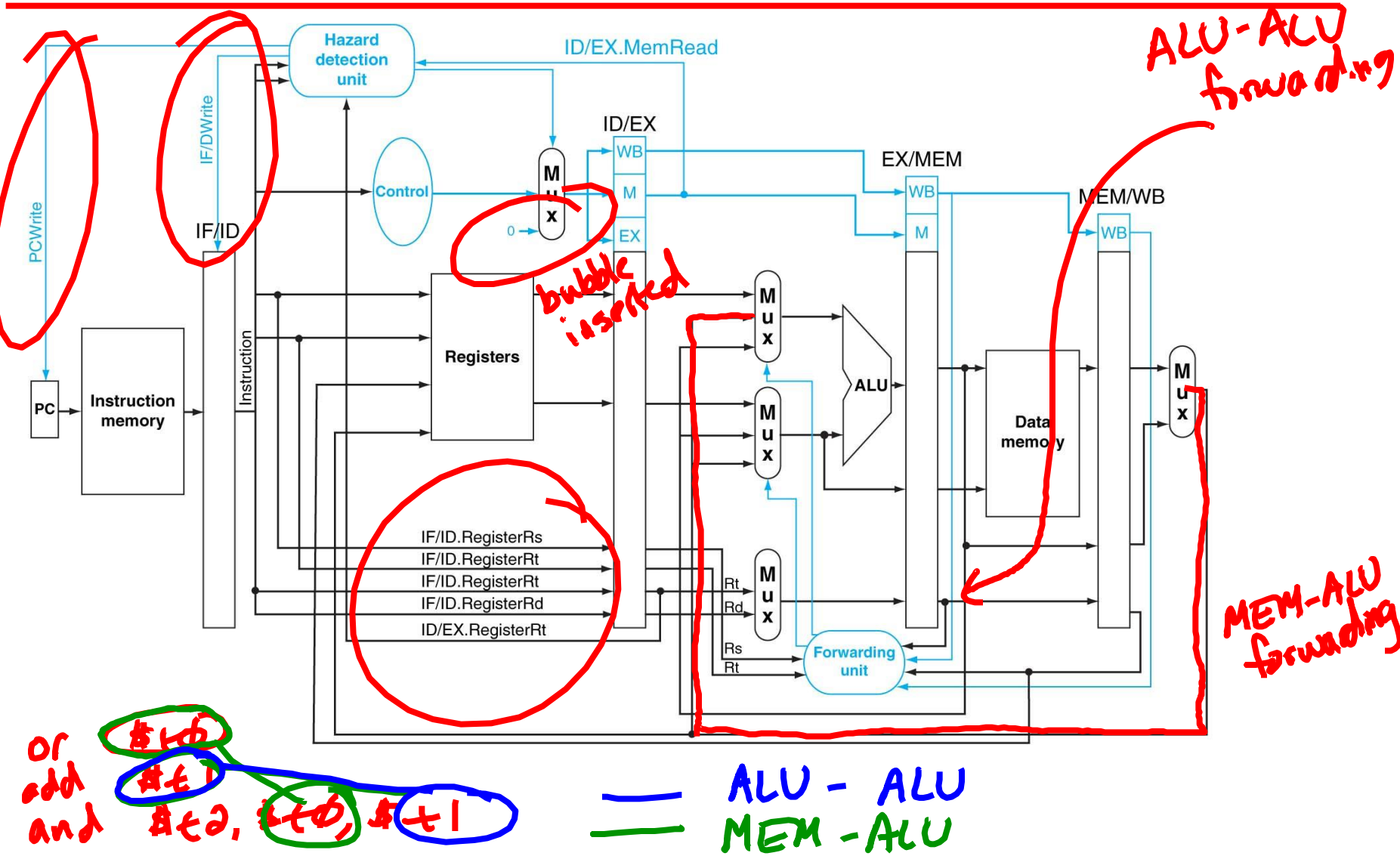
4.7 Forwarding Can't Always Save the Day



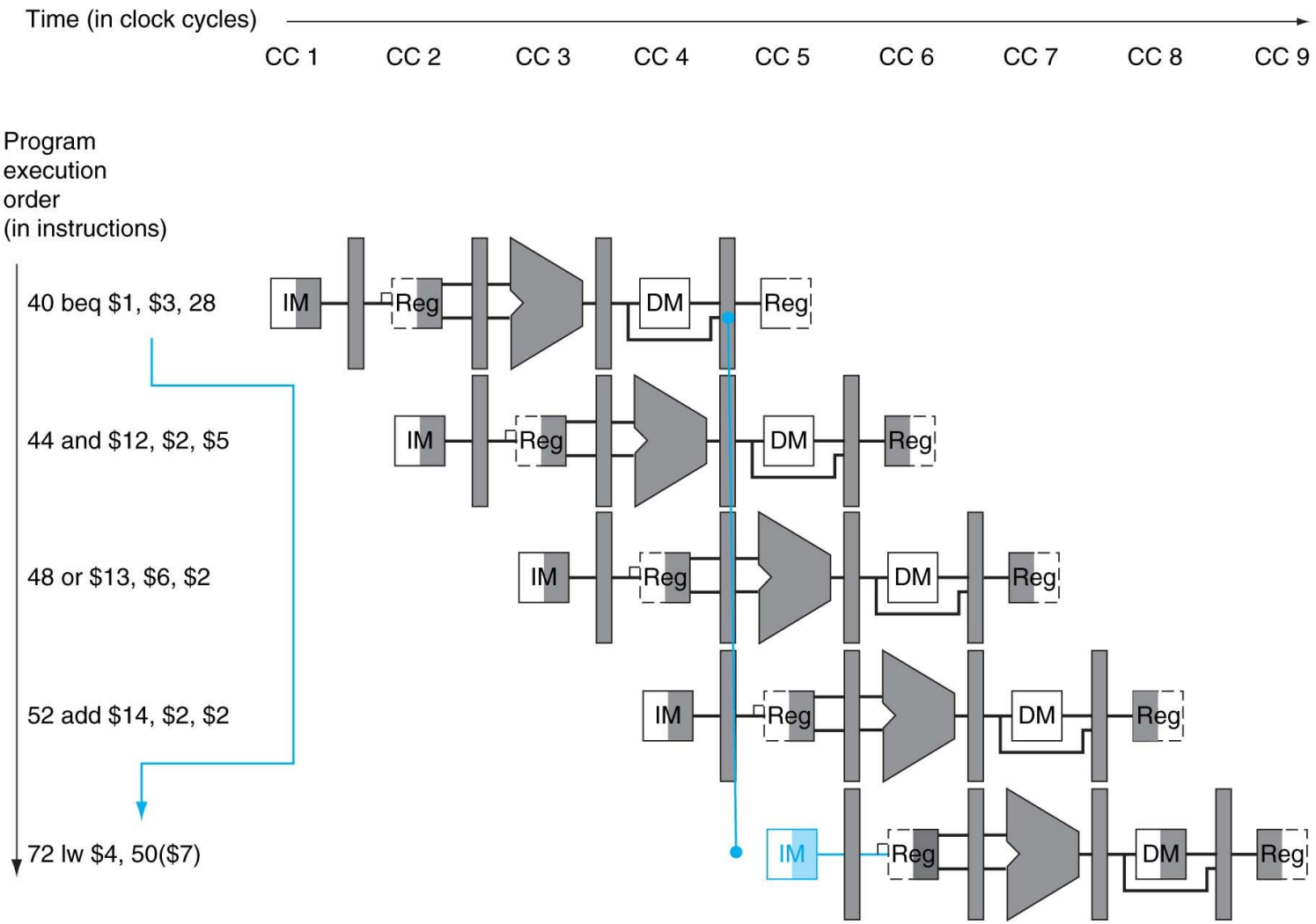
4.7 Stalling in Action



4.7 Hazard Detection Unit



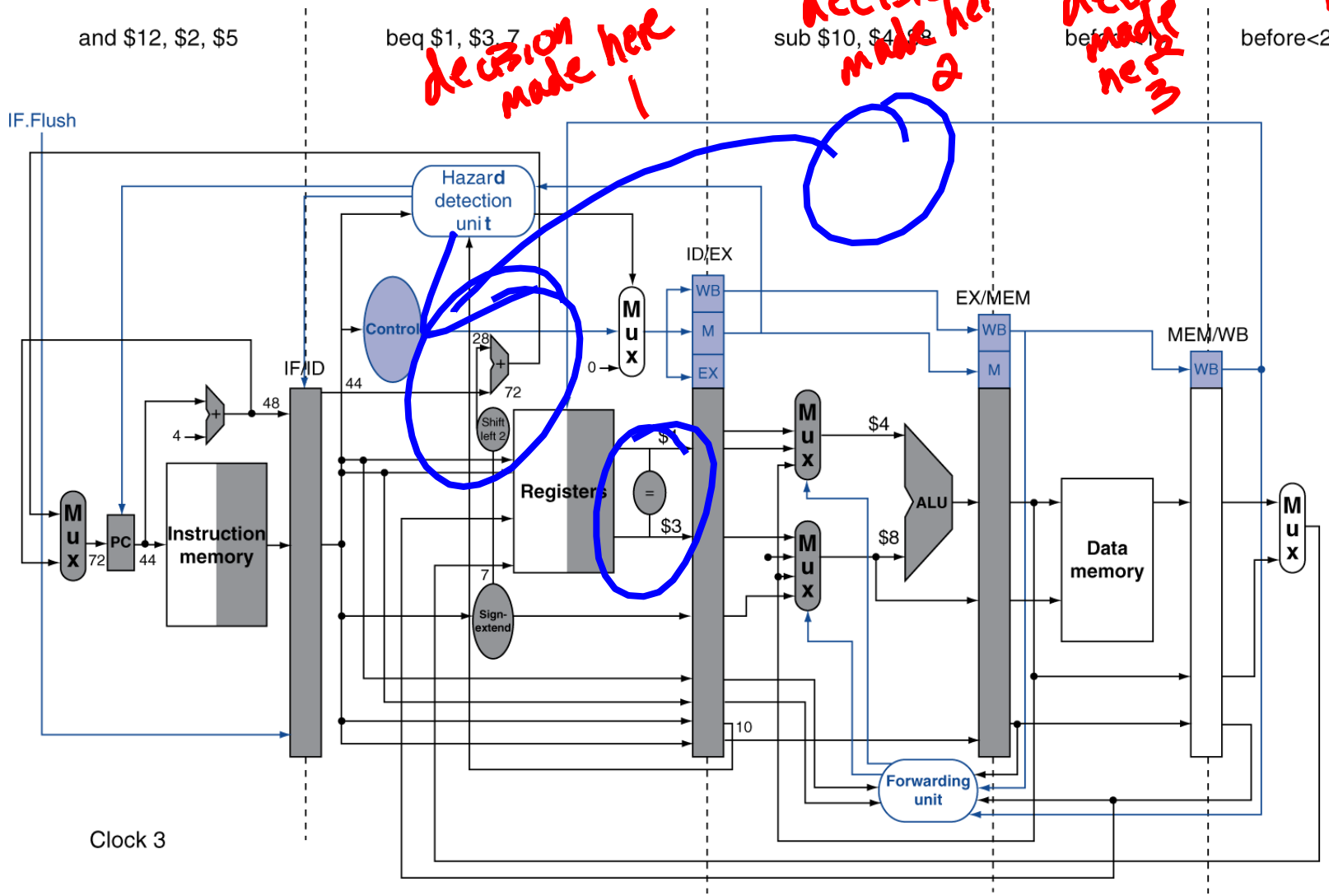
4.8 Control Hazard Example



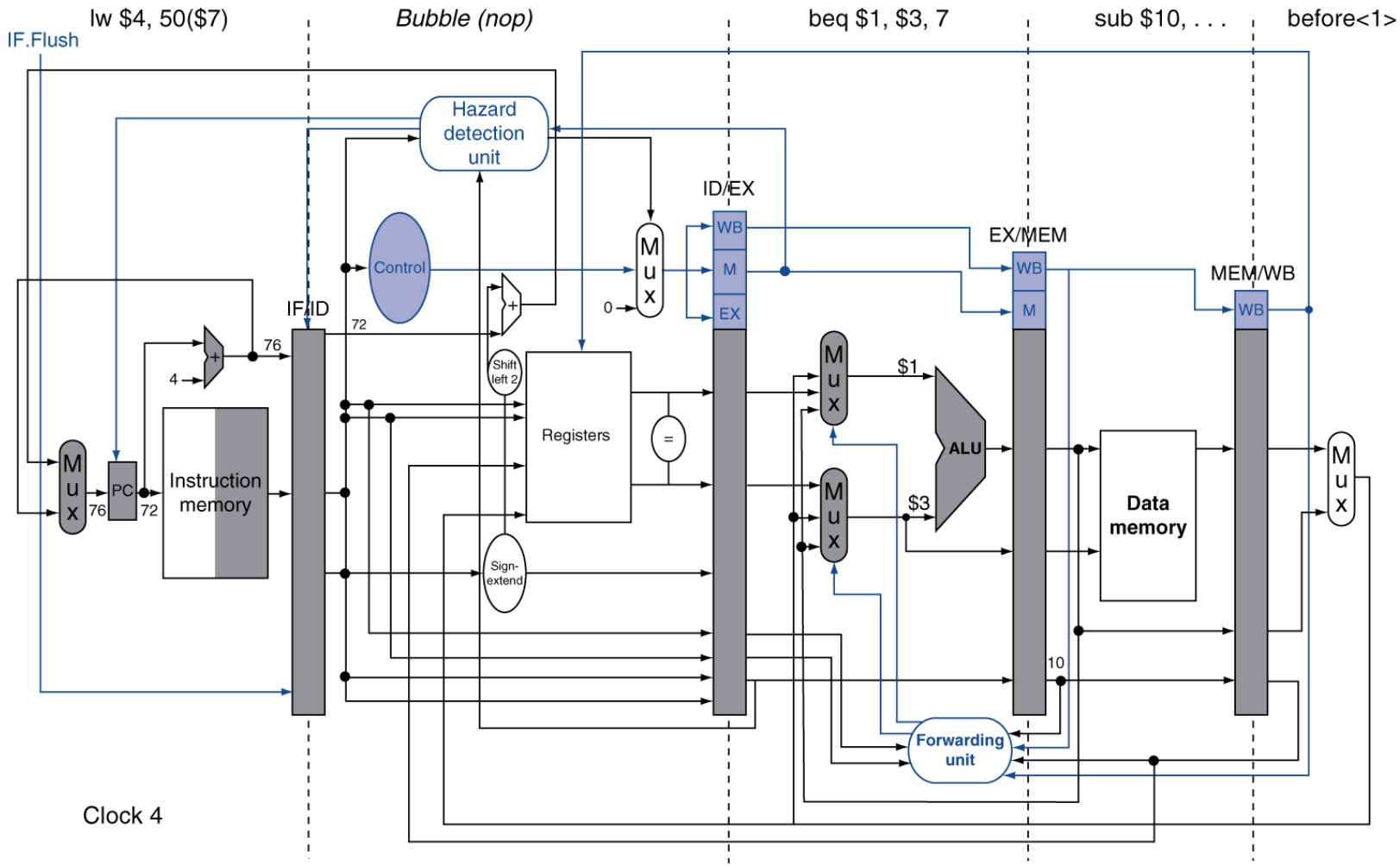
4.8 Approaches to Control Hazards

- Assume Branch Not Taken
  - If taken, instructions must be discarded
  - Change control values to 0 in IF, ID, EX
- Reducing the Delay of Branches – Two Items Needed
  - computing the branch target
  - evaluating the branch decision
  - move both to ID stage
  - only one instruction needs to be flushed

4.8 Changes to Reduce Branch Delay



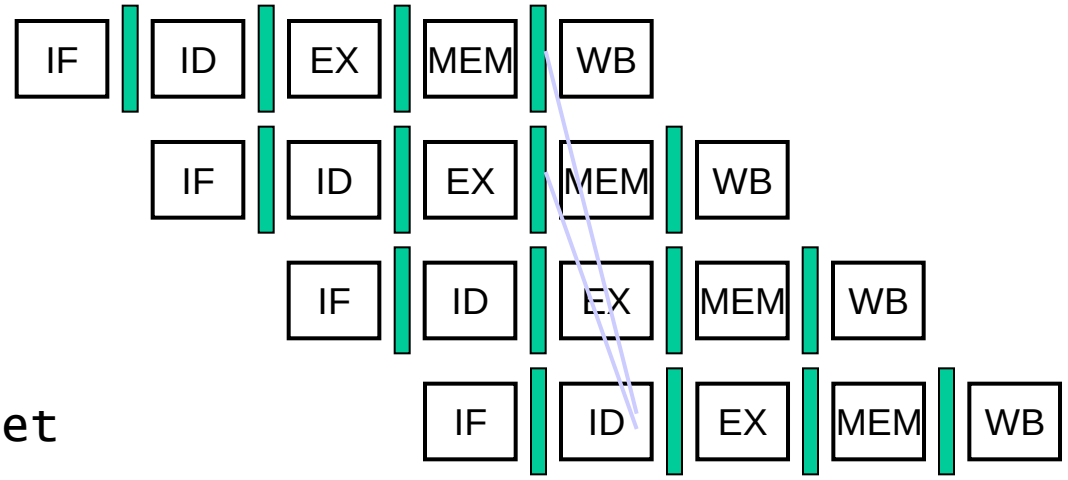
4.8 Branch Taken Example



4.8 Data Hazards for Branch (1)

If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

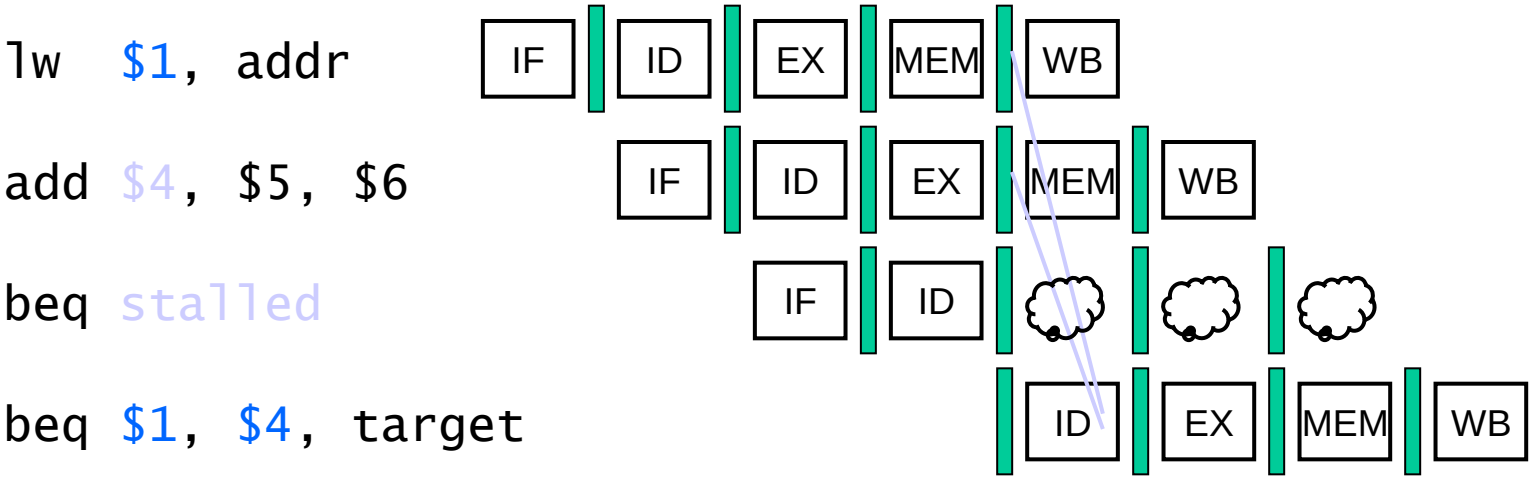
```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```



Can resolve using forwarding

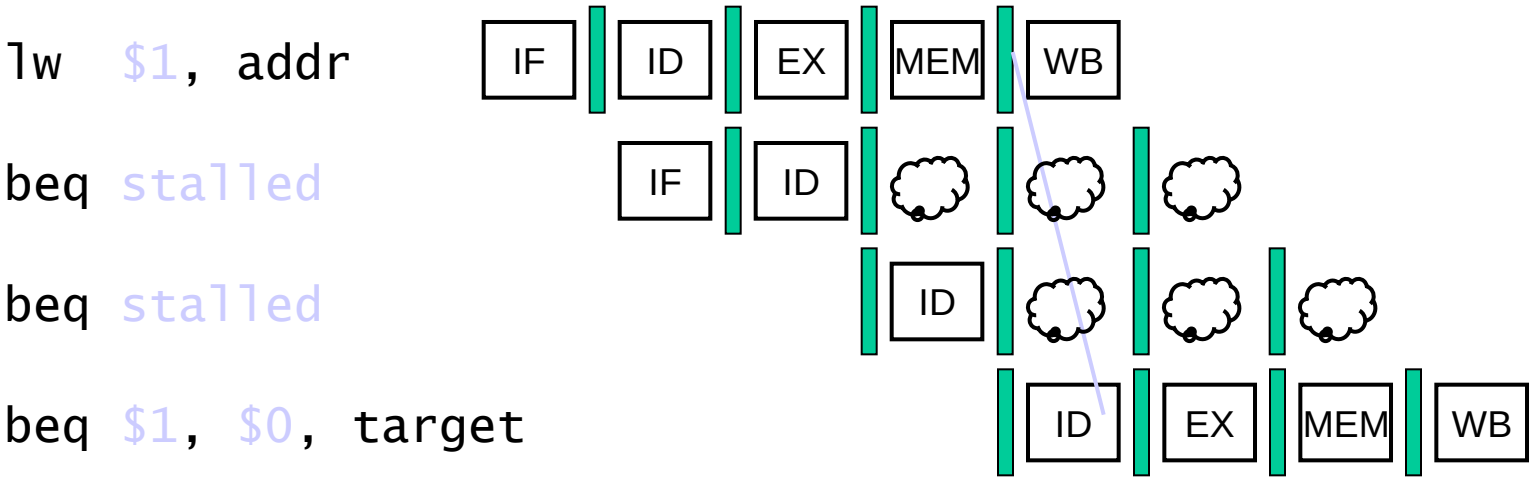
4.8 Data Hazards for Branches (2)

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle



4.8 Data Hazards for Branches (3)

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles -



4.8 Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch history table
  - indexed by recent branch instruction addresses
  - Stores outcome
  - To execute a branch
    - Check table, expect the same outcome
    - Fetch from prediction
    - Correct if necessary and update table

4.8 Shortcoming of 1-Bit History

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of Inner loop next time

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

Outer	Inner	Predict	Actual
1	1	?	T
1	2	T	T
1	3	T	T
1	4	T	T
1	5	T	NT
2	1	NT	T
2	2	T	T
2	3	T	T
2	4	T	T
2	5	T	NT

- Inner loop branches mispredicted twice!

4.9 Exceptions and Interrupts

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

4.9 Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: cause register
  - We'll assume 1-bit, 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 0180

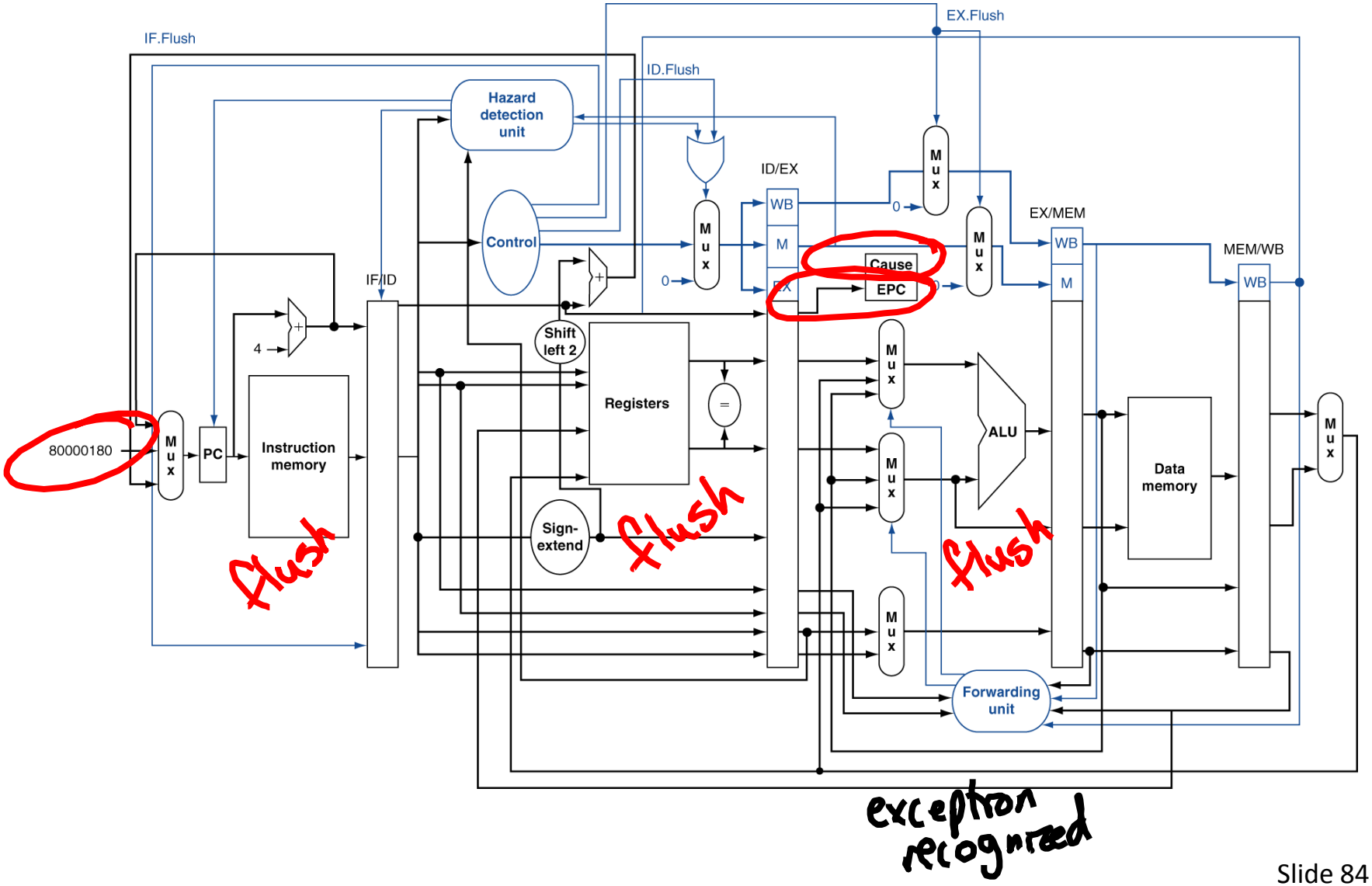
4.9 An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040
- Instructions at destination address either
  - Deal with the interrupt, or
  - Jump to real handler

4.9 Handler Actions

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

4.9 Pipeline with Exceptions



4.9 Exception Properties

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

4.9 Exception Example (1)

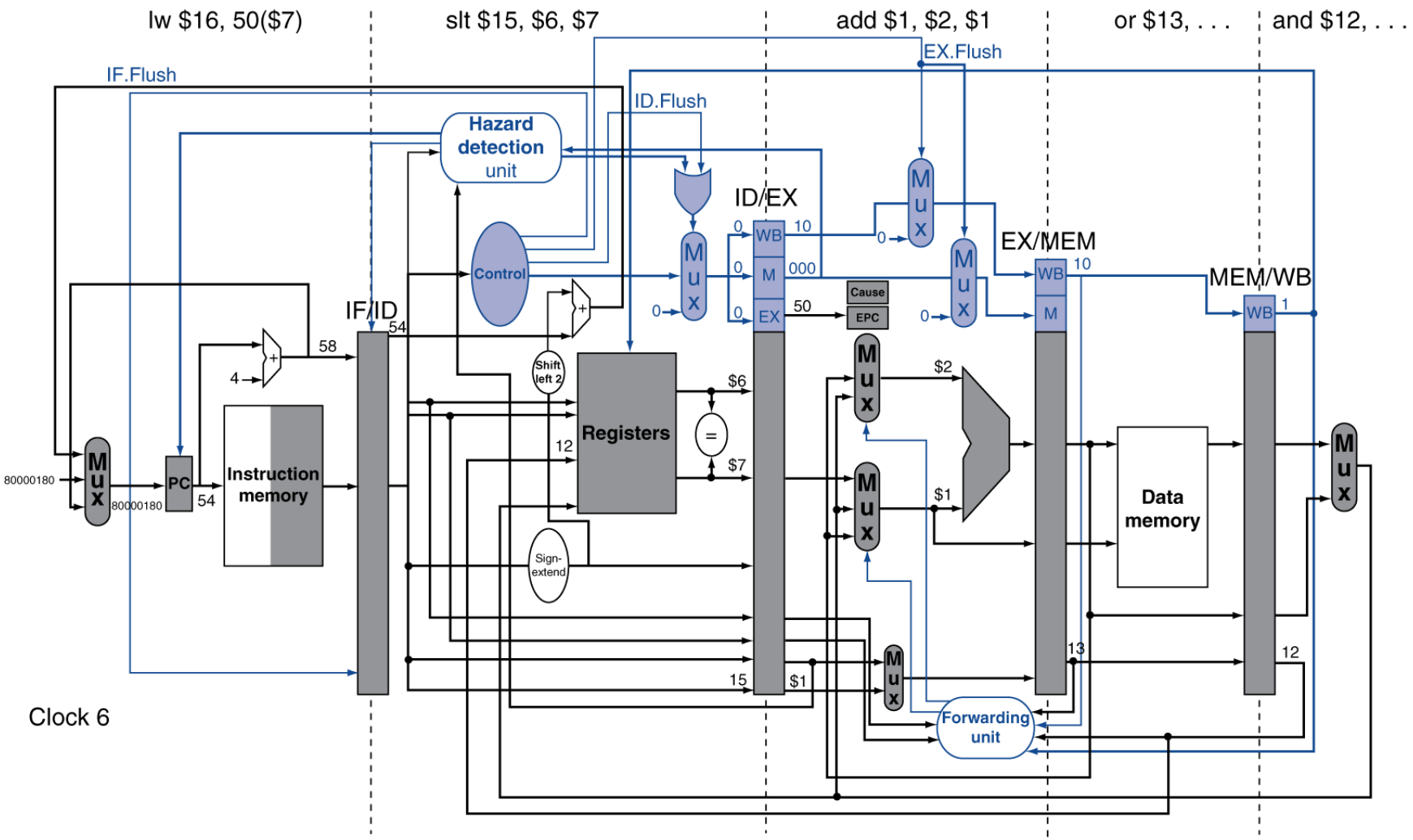
- Exception on `add` in

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

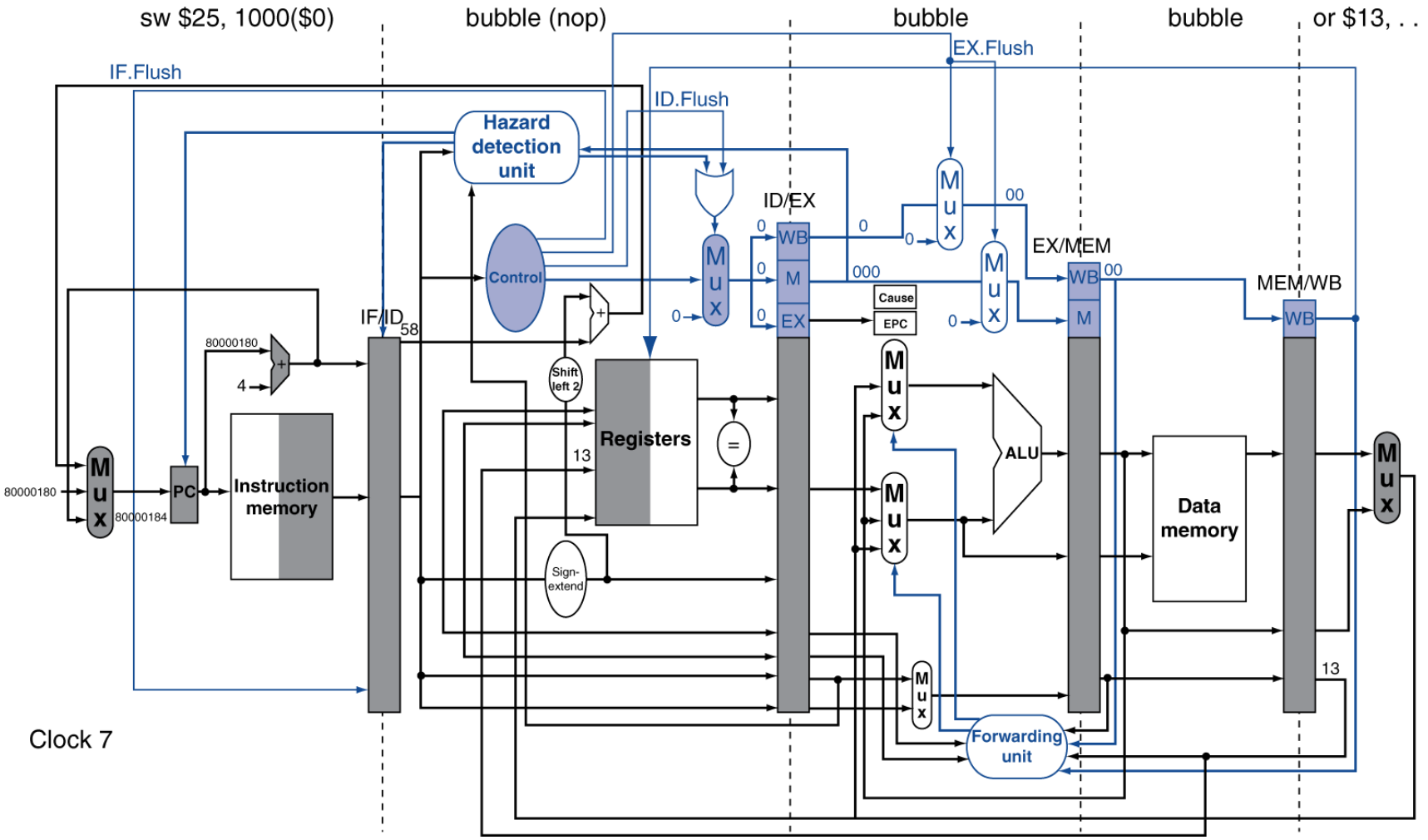
- Handler

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

4.9 Exception Example (2)



4.9 Exception Example (3)



4.9 Multiple Exceptions

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

4.9 Imprecise Exceptions

- Just stop pipeline and save state, including exception cause(s)
- Let the handler work out
  - which instruction(s) had exceptions
  - which to complete or flush
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

4.10 Parallelism via Instructions

- 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

4.10 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

4.10 Speculation

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

4.10 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 correct
  - Flush buffers on incorrect speculation

4.10 Static Multiple Issue

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

4.10 Scheduling Static Multiple Issue

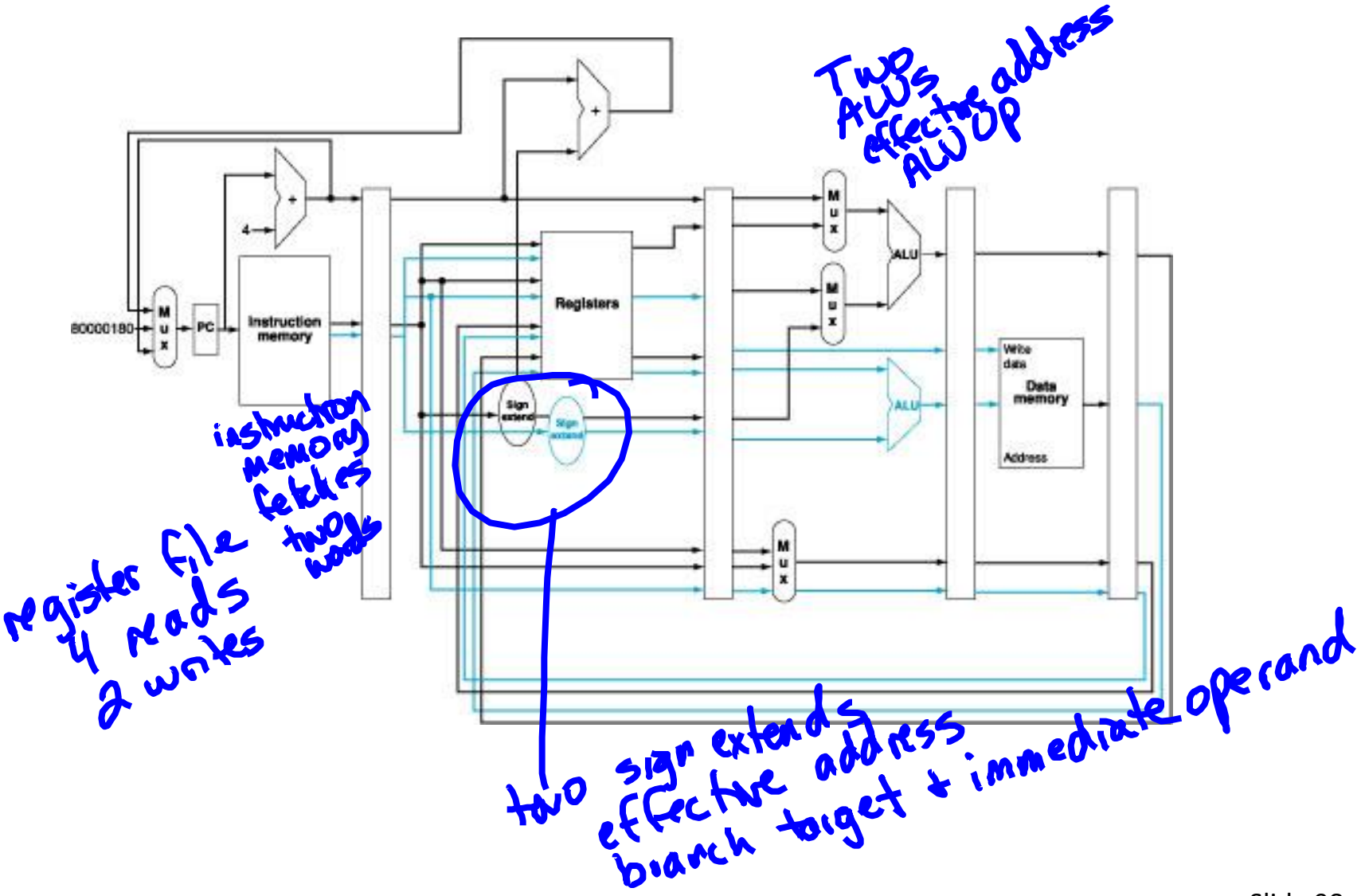
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies within a packet
  - Possibly some dependencies between packets
    - Varies between ISA; compiler must know!
  - Pad with nop if necessary

4.10 MIPS Static Dual Issue (1)

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction

Instruction type	Pipeline Stages						
ALU/branch	IF	ID	EX	MEM	WB		
Load/store	IF	ID	EX	MEM	WB		
ALU/branch		IF	ID	EX	MEM	WB	
Load/store		IF	ID	EX	MEM	WB	
ALU/branch			IF	ID	EX	MEM	WB
Load/store			IF	ID	EX	MEM	WB

4.10 MIPS Static Dual Issue (2)



4.10 Scheduling Example

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

5 instruction 4 at a time 2 or 3 depends the dependency we can do with or minimum is 4 cycles

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

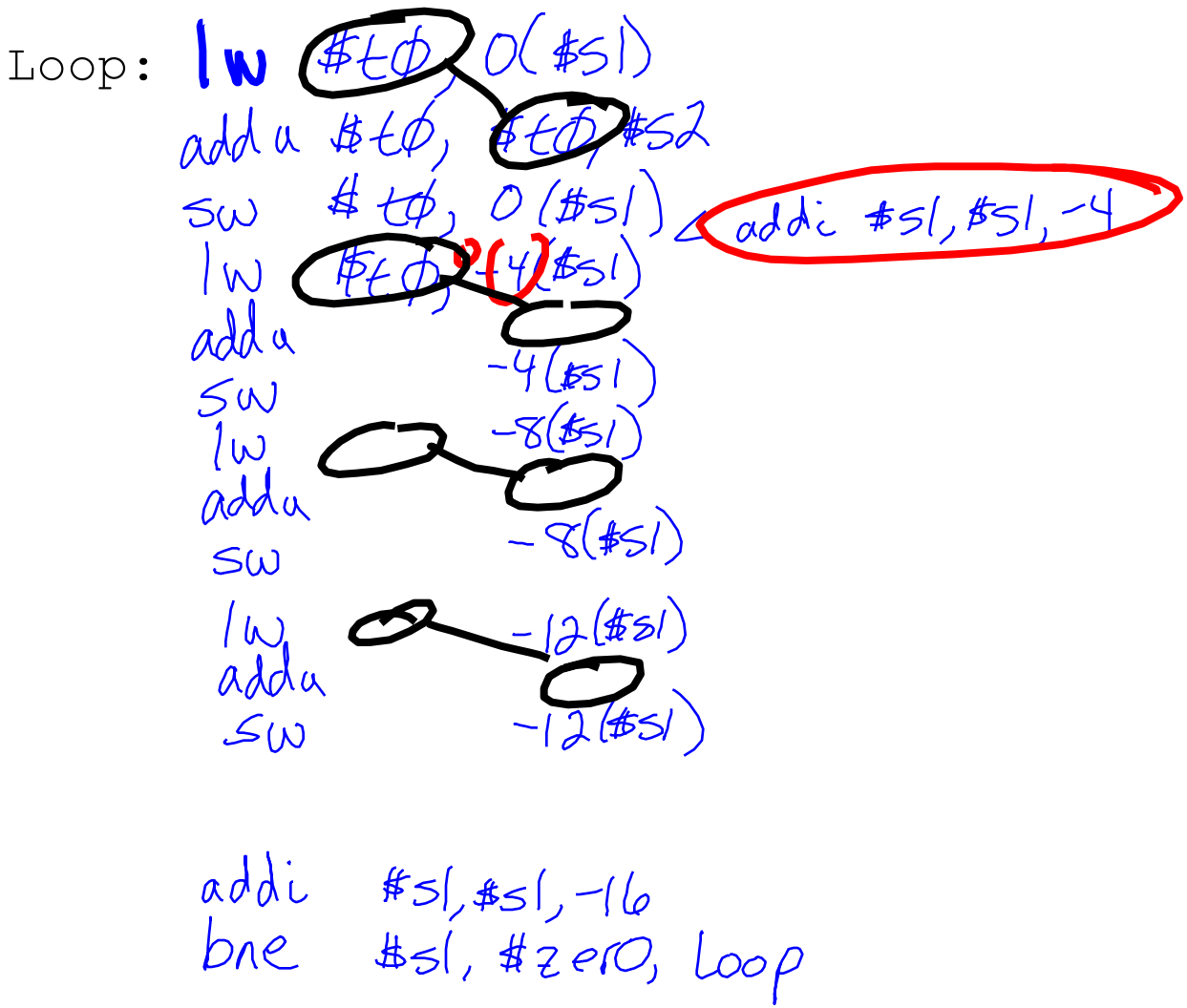
- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Utilization = 5/8 = 62.5%

4.10 Loop Unrolling (1)

```
Loop: lw
      addu
      sw
      addi
      lw
      addu
      sw
      addi
      lw
      addu
      sw
      addi
      lw
      addu
      sw
      addi
      bne
```

4.10 Loop Unrolling (2)



4.10 Loop Unrolling (3)

```
Loop: lw   $t0
      lw   $t1
      lw   $t2
      lw   $t3
      addu
      addu
      addu
      addu
      sw   $t0
      sw   $t1
      sw   $t2
      sw   $t3
      addi
      bne
```

4.10 Scheduling Unrolled Loop

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called "register renaming"

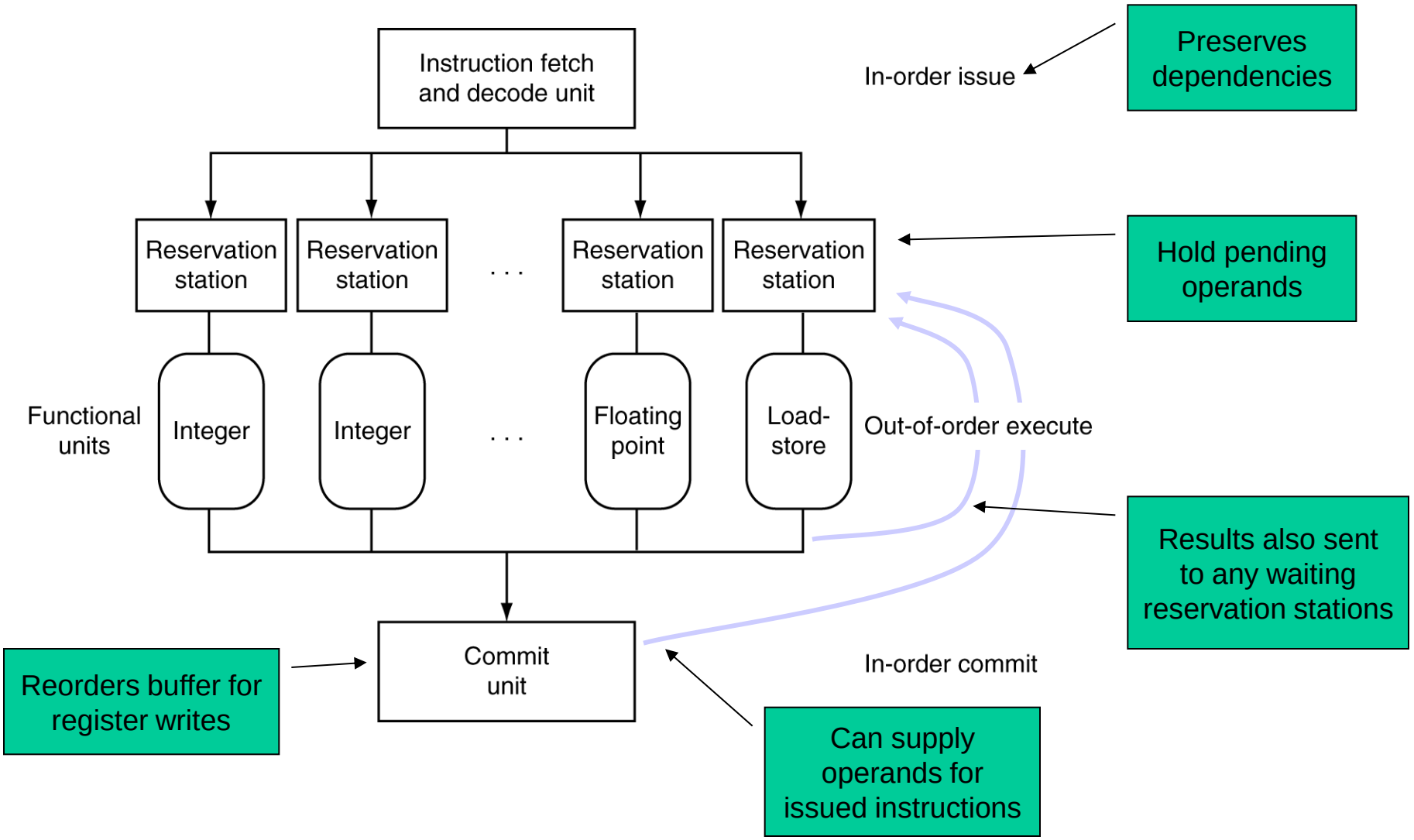
IPC = 14/8 = 1.75

Utilization = 14/16 = 87.5%

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1, -16	lw $t0, 0($s1)	1
	nop	lw $t1, 12-4($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8-8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4-12($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16-0($s1)	5
	addu $t3, $t3, $s3	sw $t1, 12-4($s1)	6
nop	addi $s1, $s1, -16	sw $t2, 8-8($s1)	7
	bne $s1, $zero, Loop	sw $t3, -12($s1)	8

4 4

4.10 Dynamically Scheduled CPU



4.10 Speculation

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

4.10 Does Multiple Issue Work?

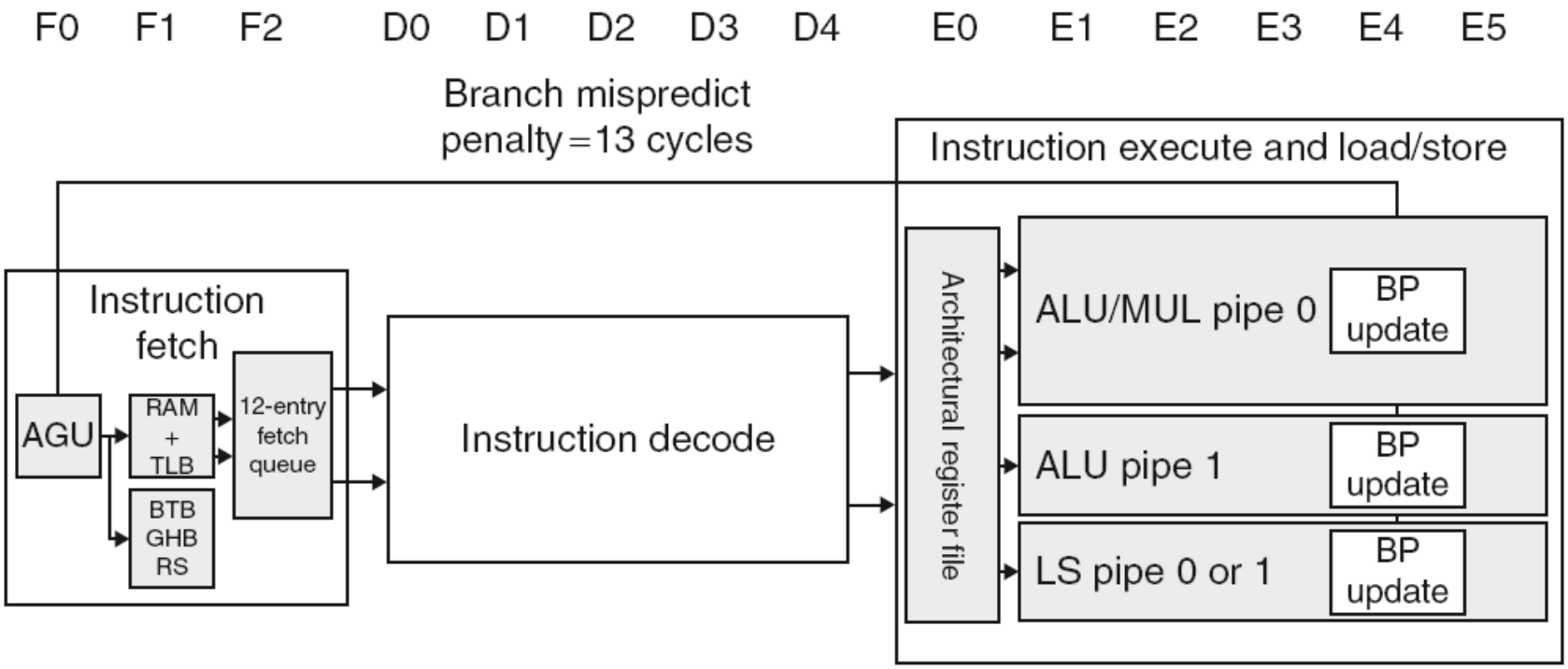
- Yes, but not as much as we'd like
- Programs have real dependences that limit ILP
- Some dependences are hard to eliminate
  - pointer aliasing
- Some parllelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full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

4.10 Power Efficiency

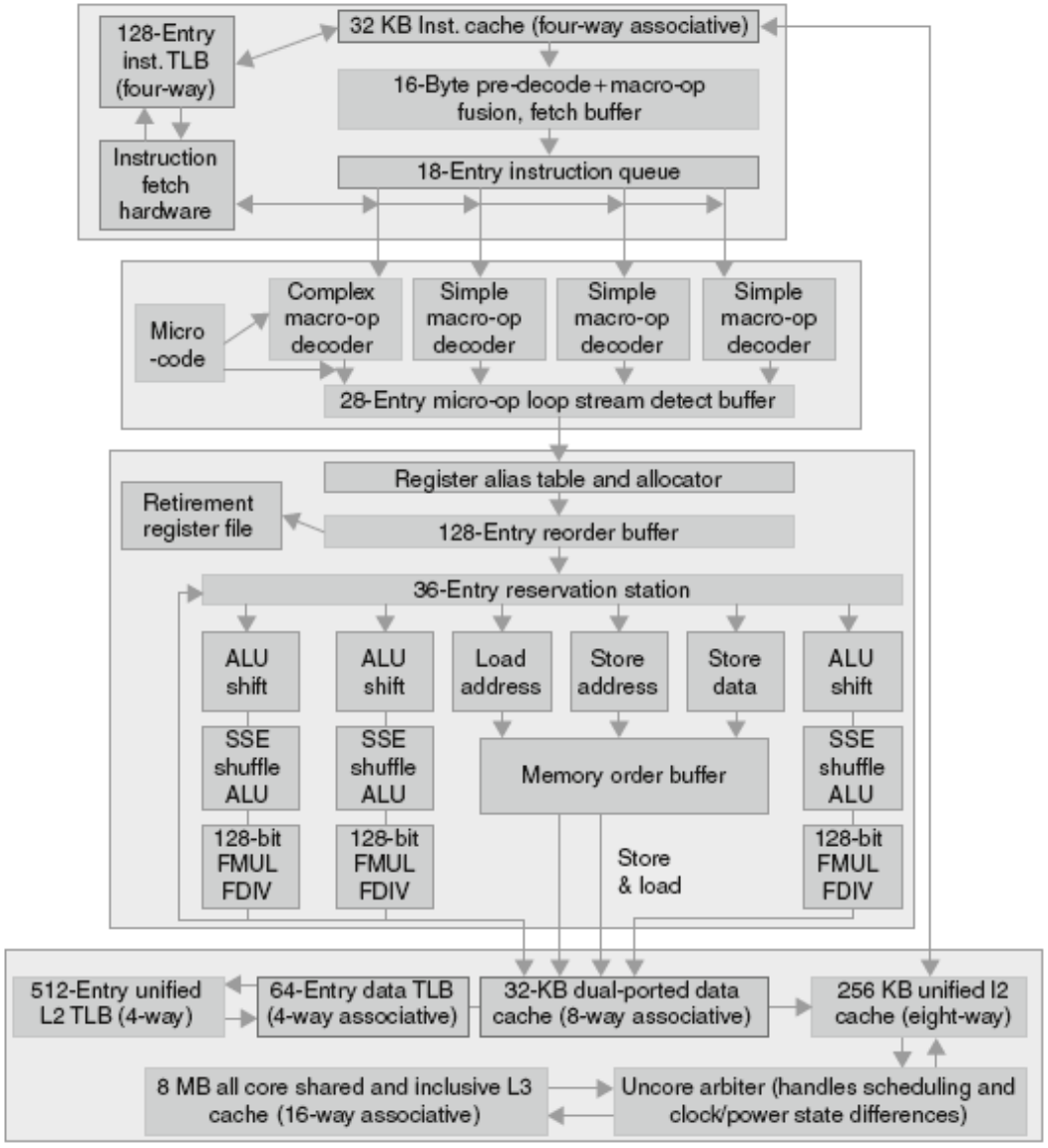
- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

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

4.11 ARM Cortex A8 Pipeline



4.11 Core i7 Pipeline



4.15 Concluding Remarks

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