

---

# IT4272E-COMPUTER SYSTEMS

TS. Phạm Ngọc Hưng

Viện CNTT&TT

[hungpn@soict.hust.edu.vn](mailto:hungpn@soict.hust.edu.vn)

---

# IT4272E-COMPUTER SYSTEMS

## Chapter 4: The Processor

[with materials from *Computer Organization and Design, 4<sup>th</sup> Edition*,  
Patterson & Hennessy, © 2008, MK]

# Introduction

---

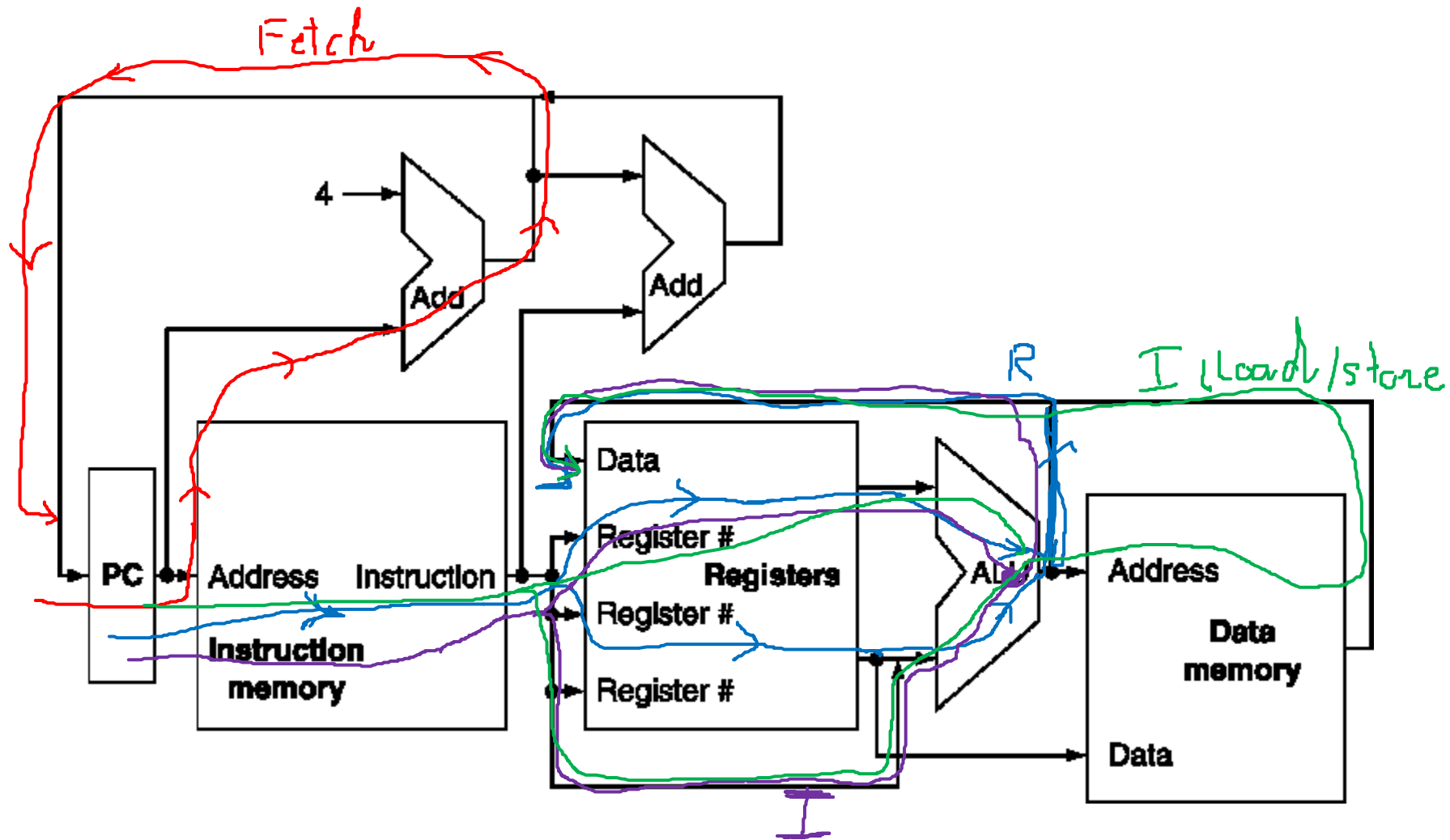
- ❑ CPU performance factors
  - | Instruction count
    - Determined by ISA and compiler
  - | CPI and Cycle time
    - Determined by CPU hardware
- ❑ We will examine two MIPS implementations
  - | A simplified version
  - | A more realistic pipelined version
- ❑ Simple subset, shows most aspects
  - | Memory reference: lw, sw
  - | Arithmetic/logical: add, sub, and, or, slt
  - | Control transfer: beq, j

## Instruction Execution

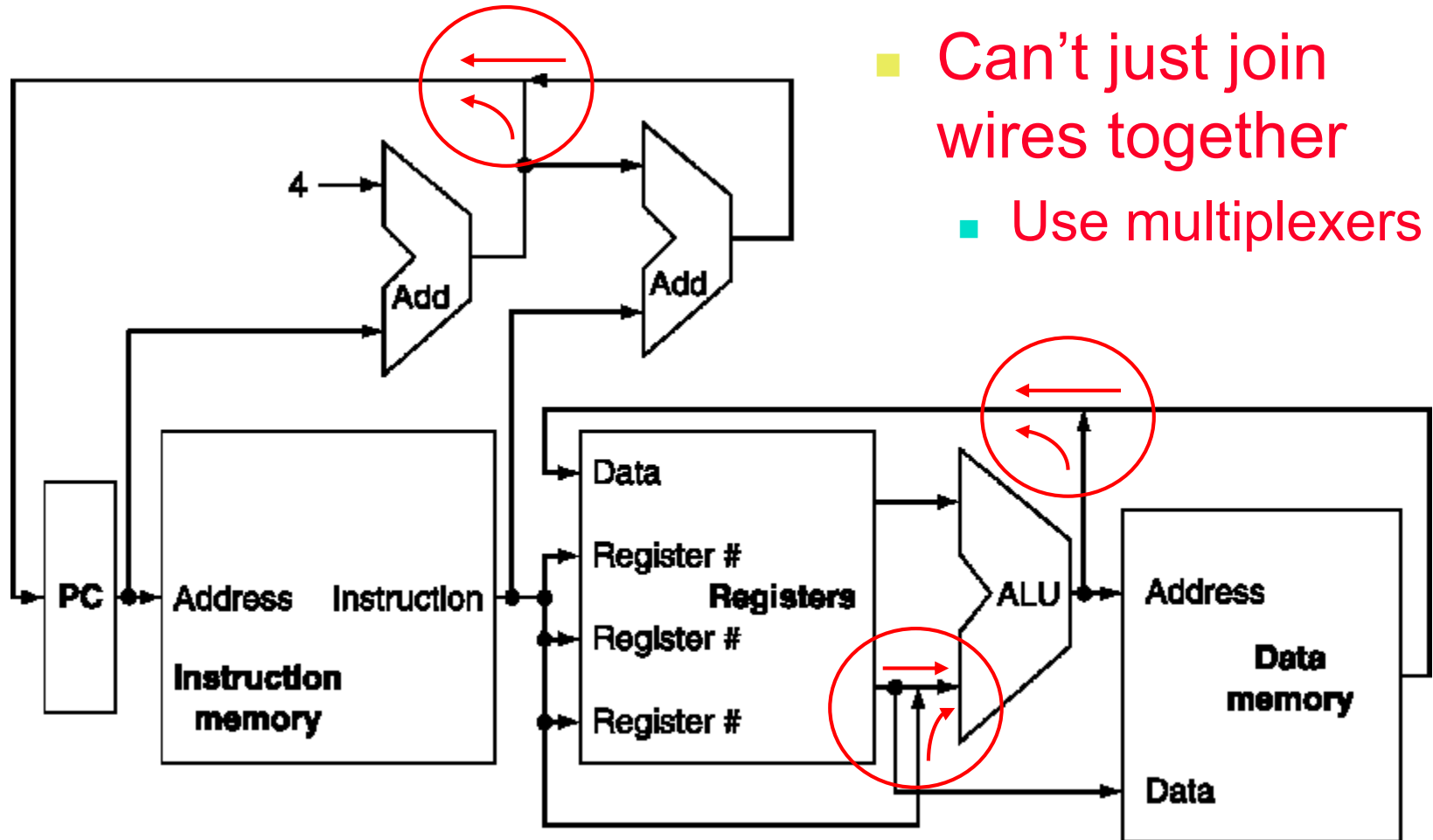
---

- ❑ PC → instruction memory, fetch instruction
- ❑ Register numbers → register file, read registers
- ❑ Depending on instruction class
  - | Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - | Access data memory for load/store
  - |  $PC \leftarrow \text{target address or } PC + 4$

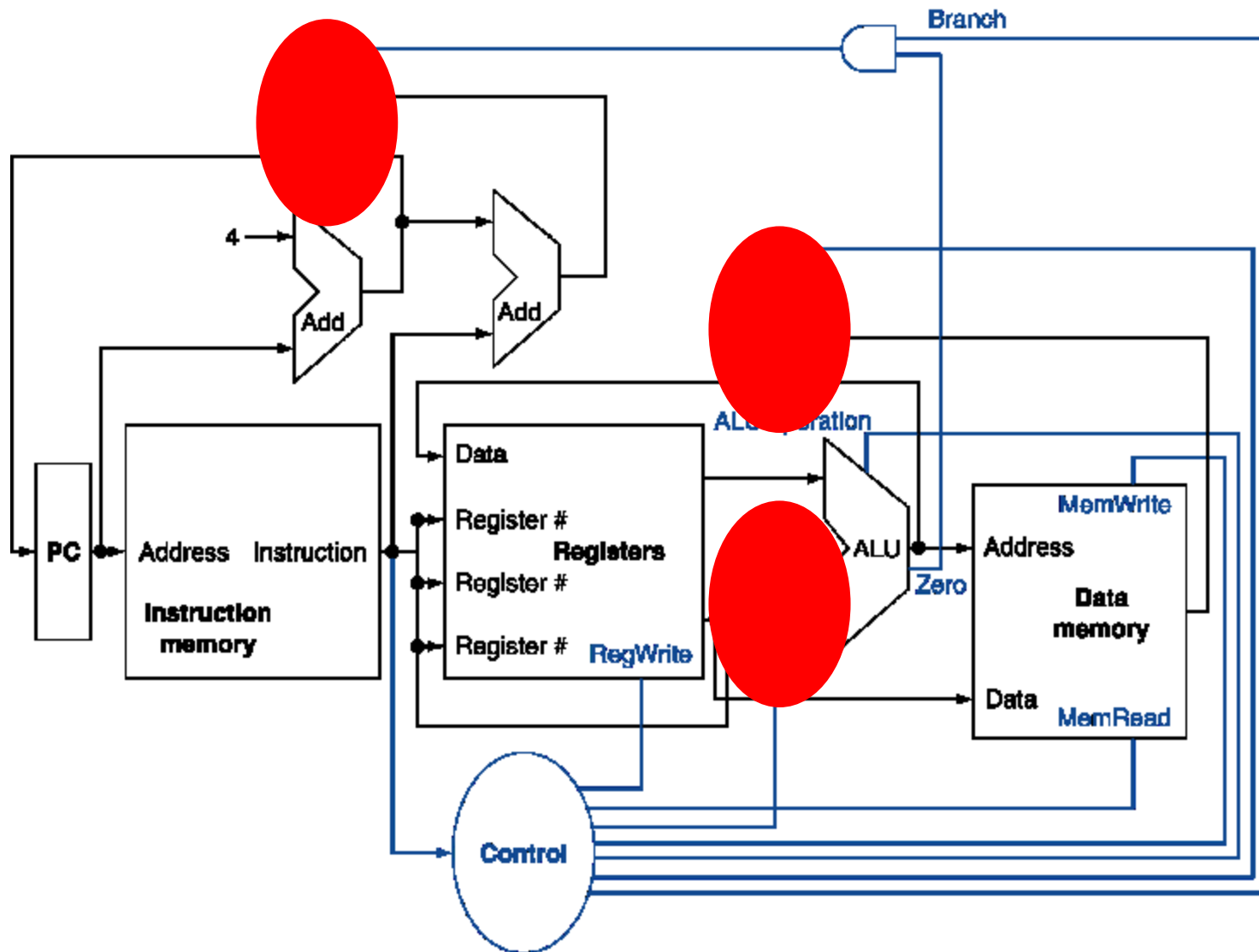
# CPU Overview



# Multiplexers

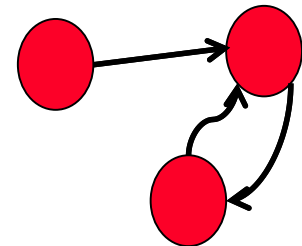


# Control



# Logic Design Basics

- ❑ Information encoded in binary
  - | Low voltage = 0, High voltage = 1
  - | One wire per bit
  - | Multi-bit data encoded on multi-wire buses
- ❑ Combinational element
  - | Operate on data, like and/or/not/add/sub
  - | Output is a function of input.  $\text{output} = f(\text{input})$
- ❑ State (sequential) elements
  - | Store information

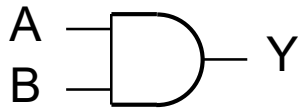




# Combinational Elements

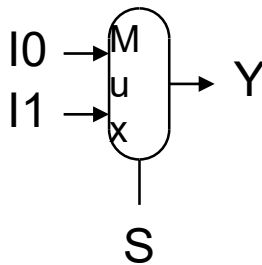
## ■ AND-gate

■  $Y = A \& B$



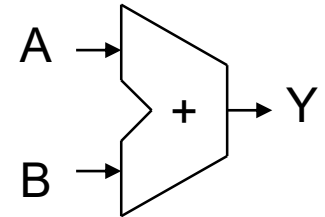
## ■ Multiplexer

■  $Y = S ? I1 : I0$



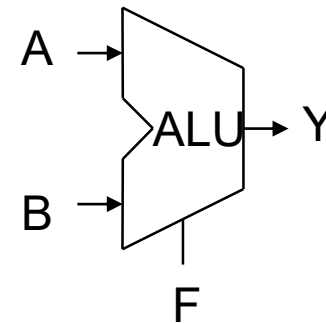
## ■ Adder

■  $Y = A + B$



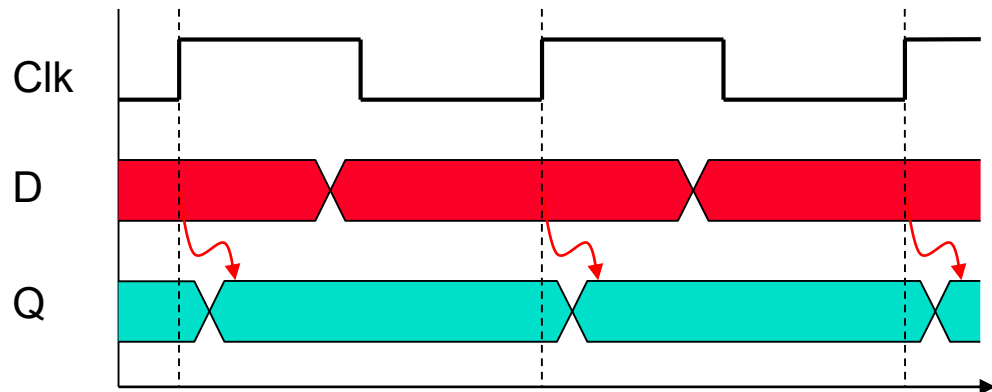
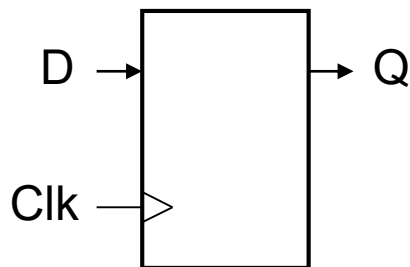
## ■ Arithmetic/Logic Unit

■  $Y = F(A, B)$



# Sequential Elements

- ❑ Register: stores data in a circuit
  - | Uses a clock signal to determine when to update the stored value
  - | Edge-triggered: update when Clk changes from 0 to 1

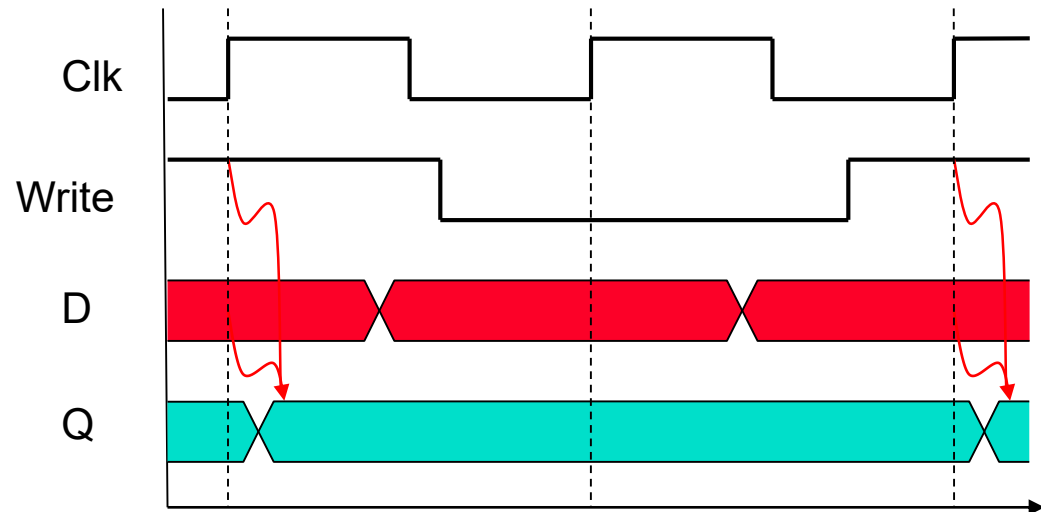
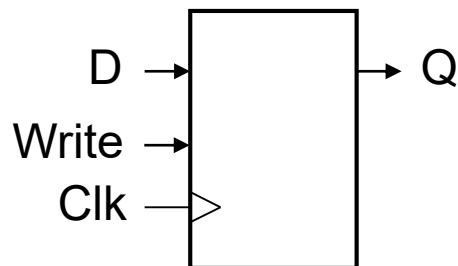


*edge /eddʒ/: sườn cạnh*

# Sequential Elements

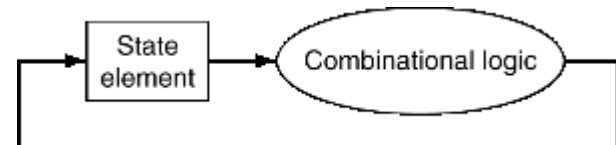
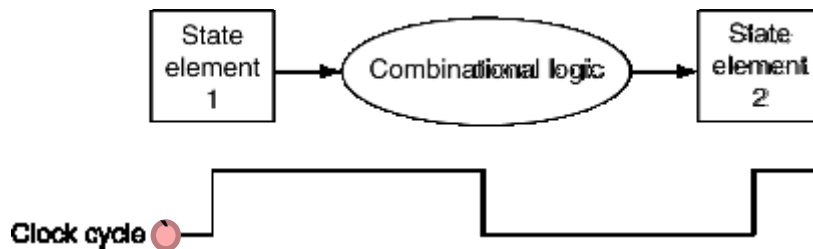
## ❑ Register with write control

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



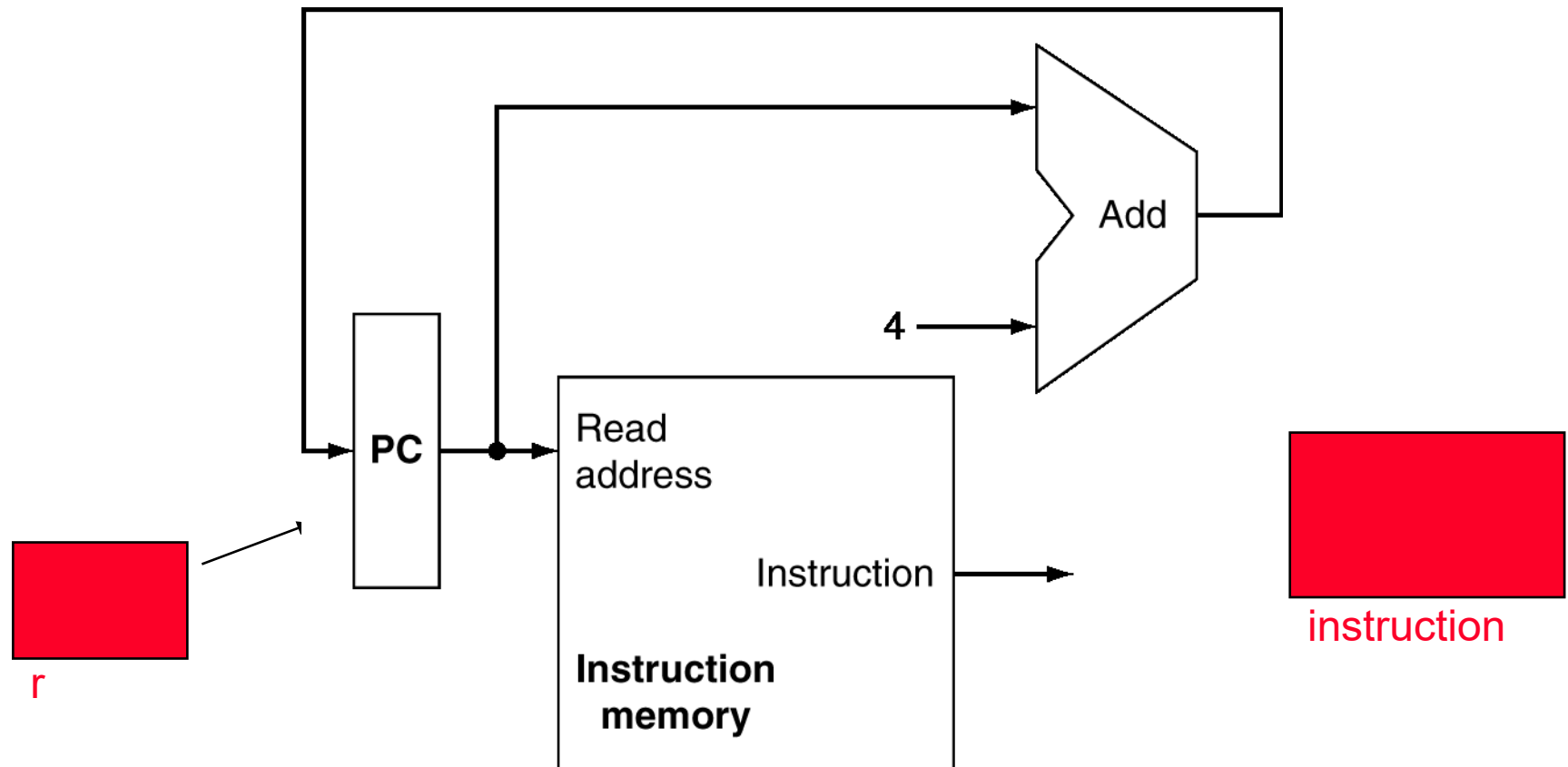
# Clocking Methodology

- ❑ Combinational logic transforms data during clock cycles
  - | Between clock edges
  - | Input from state elements, output to state element
  - | Longest delay determines clock period



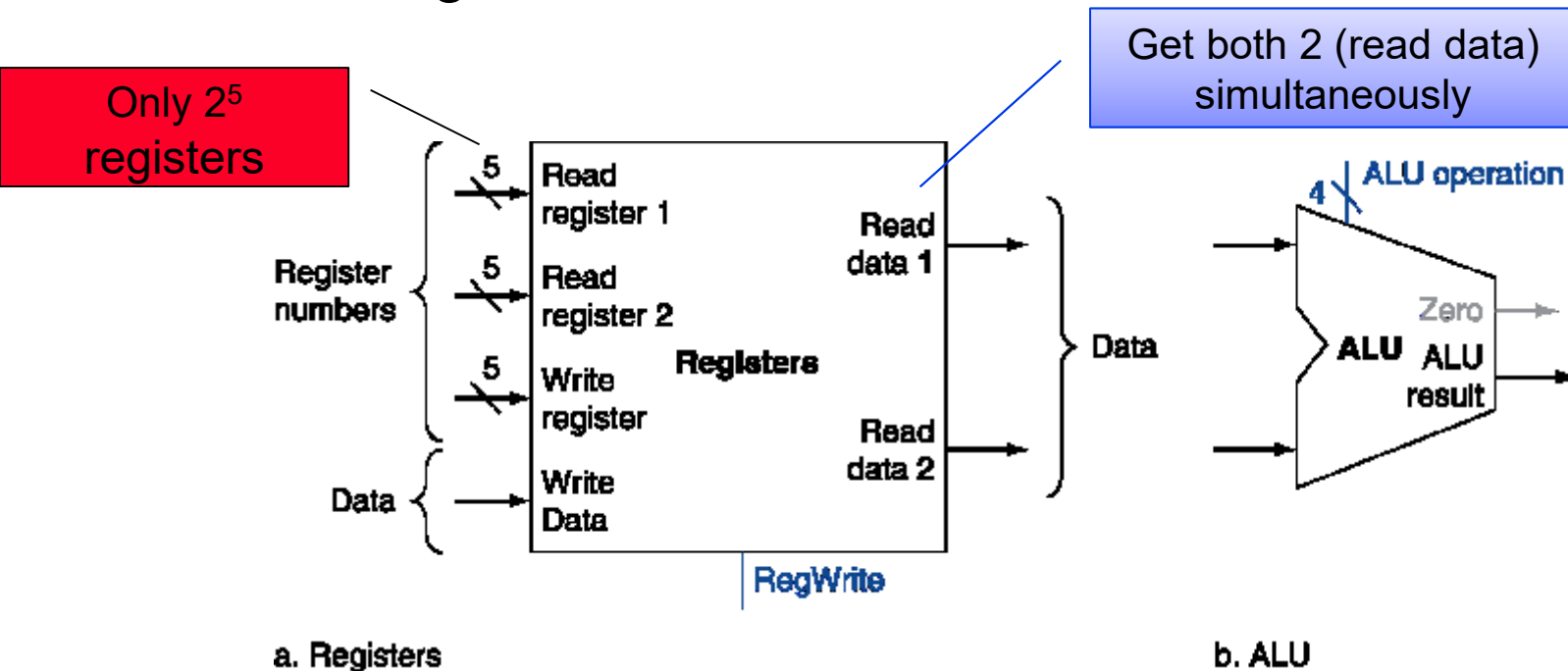
*methodology /,meθə'dɒlədʒi/: phương pháp học*  
*determine /di'tə:mi:n/: quyết định*

# Instruction Fetch



# R-Format Instructions

- ❑ Read two register operands
- ❑ Perform arithmetic/logical operation
- ❑ Write register result



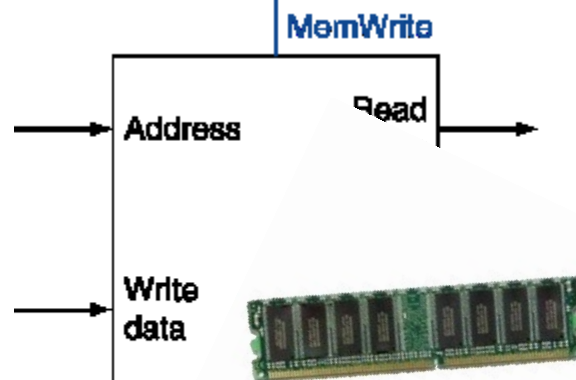
# Load/Store Instructions

LW

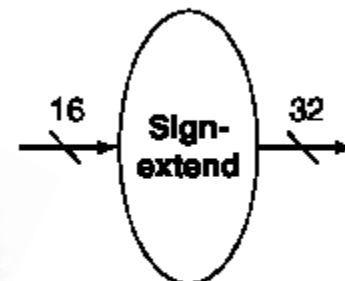
$R_D, \text{OFF16}(R_S)$

$R_D = \text{MEM32}(R_S + \text{OFF16}^+)$

- ❑ Read register operands
- ❑ Calculate address using 16-bit offset
  - | Use ALU, but sign-extend offset
- ❑ Load: Read memory and update register
- ❑ Store: Write register value to memory



a. Data memory unit



b. Sign extension unit

# Branch Instructions

- ❑ Read register operands

BEQ

RS, RT, OFF18

IF RS = RT, PC += OFF18<sup>±</sup>

- ❑ Compare operands

  - | Use ALU, subtract and check Zero output

- ❑ Calculate target address

  - | Sign-extend displacement

  - | Shift left 2 places (word displacement)

  - | Add to PC + 4

    - Already calculated by instruction fetch

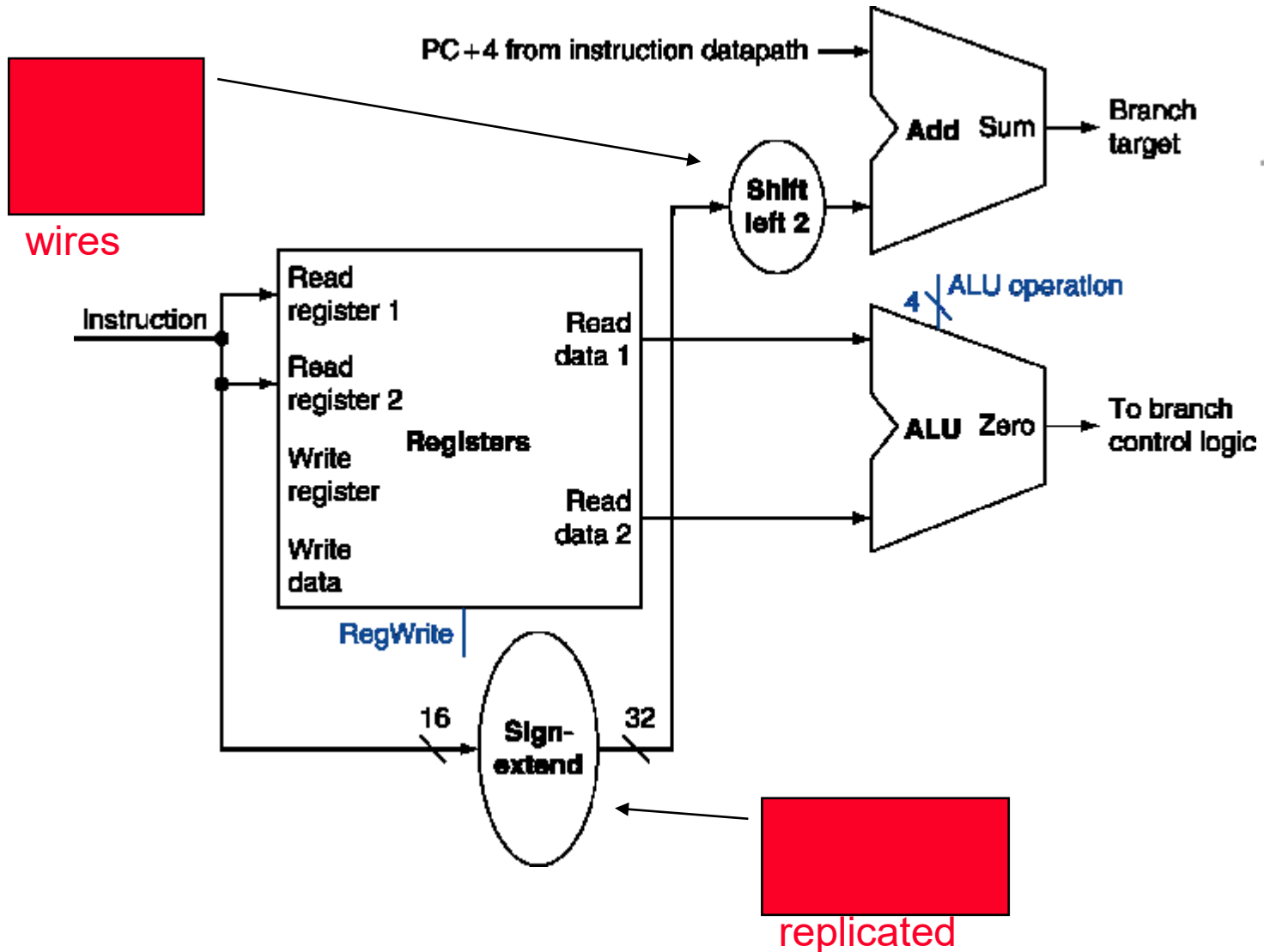


# Branch Instructions

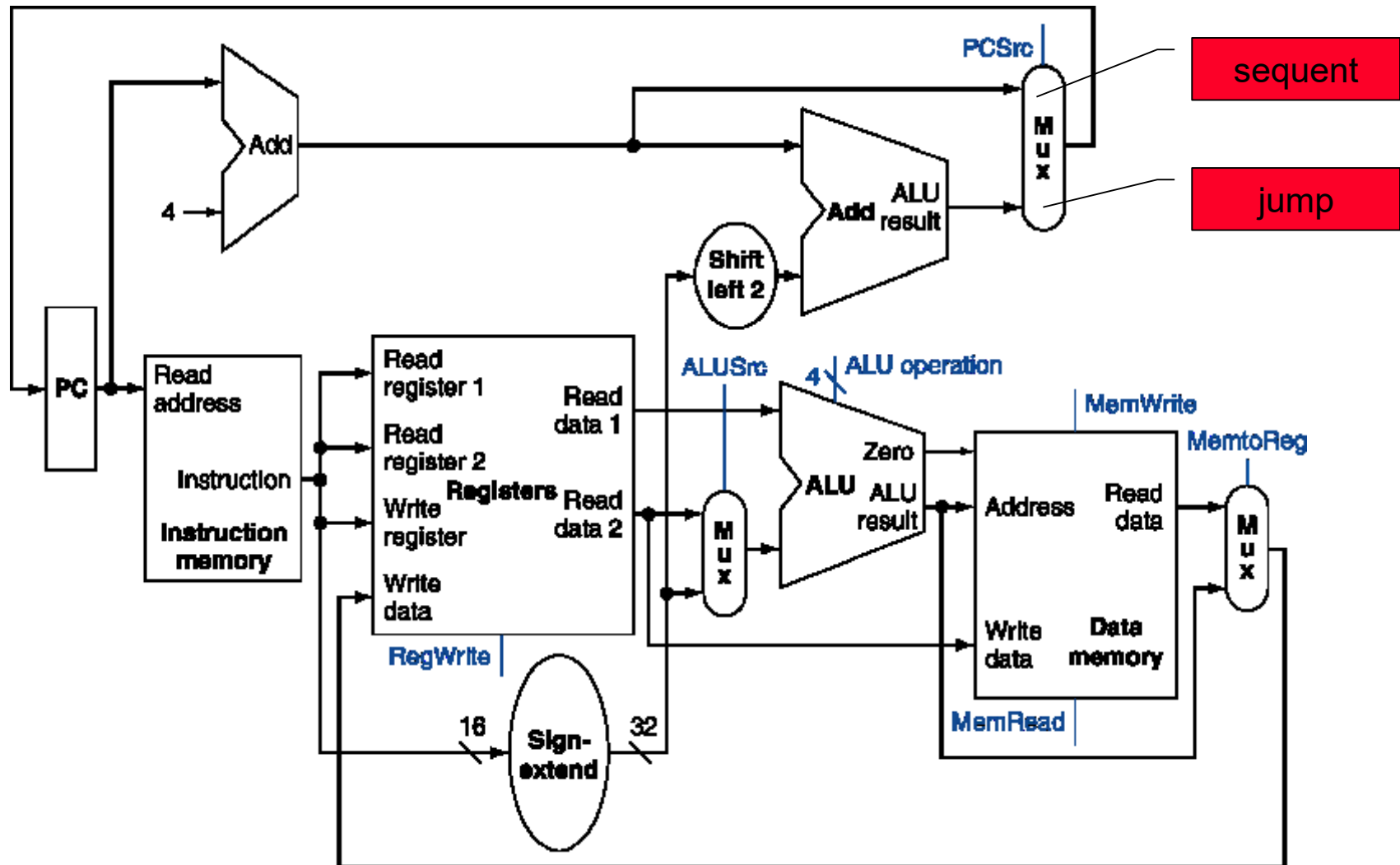
BEQ

$R_S, R_T, \text{OFF18}$

IF  $R_S = R_T, PC += \text{OFF18}^\pm$



# Full Datapath



## ALU Control


- ❑ ALU used for
  - | Load/Store: F = add
  - | Branch: F = subtract
  - | R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

*Scheme /ski:m/ Kế hoạch, Kịch bản*

# ALU Control

- ❑ Assume 2-bit ALUOp derived from opcode
  - | Combinational logic derives ALU control

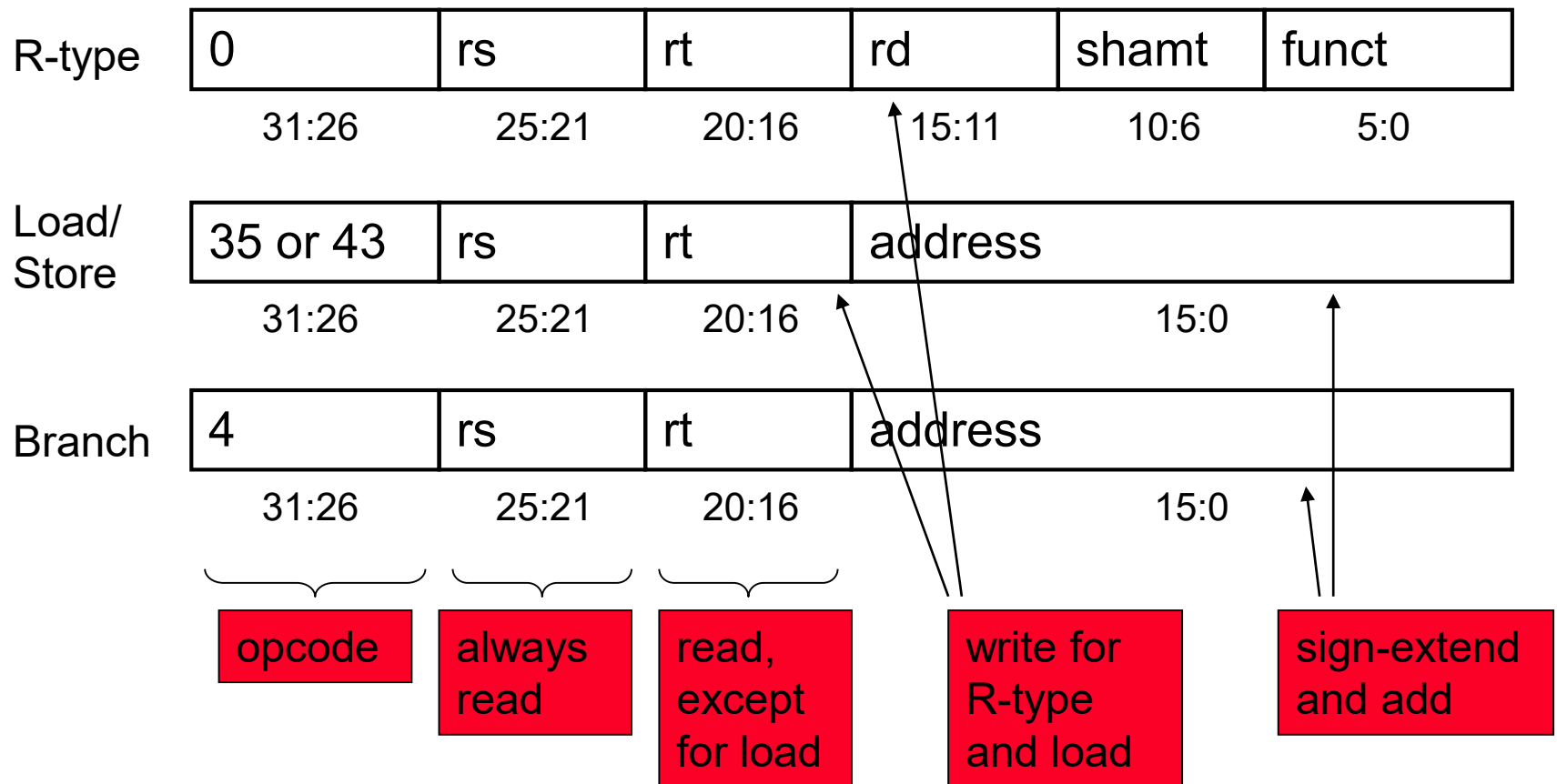


opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
Sw	00	store word	XXXXXX	add	0010
Beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

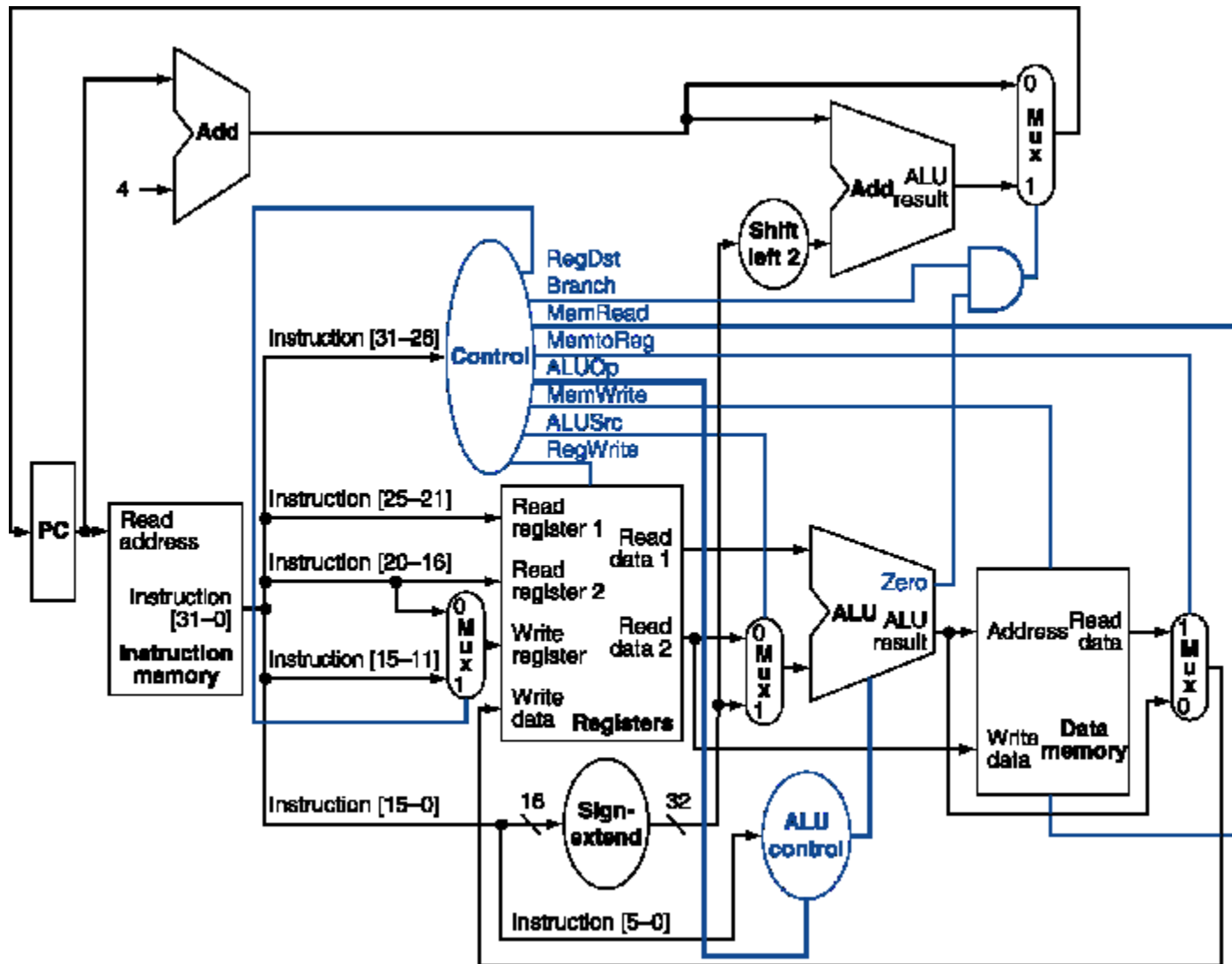
*derive /di'raiv/": nhận được từ...*

# The Main Control Unit

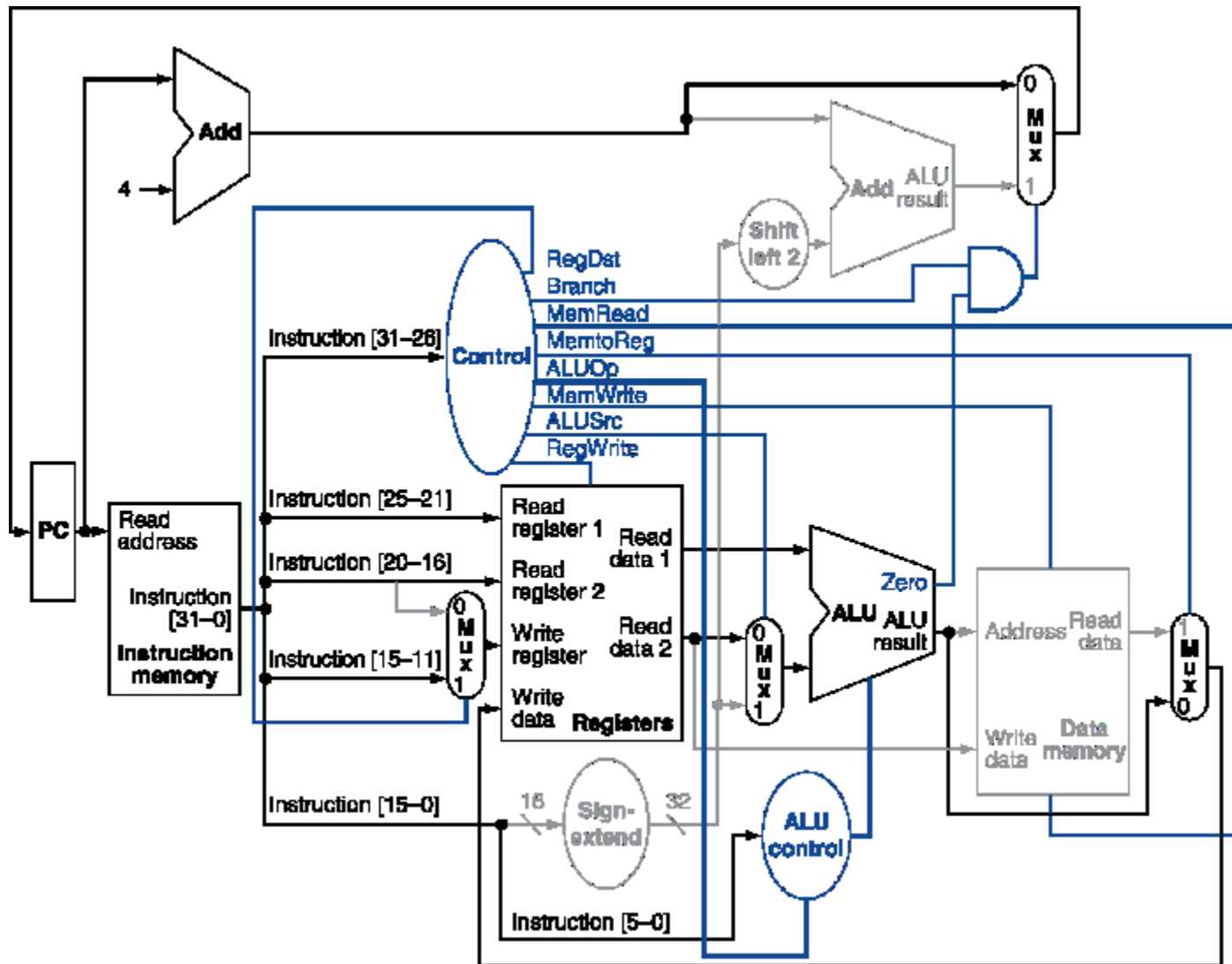
## ❑ Control signals derived from instruction



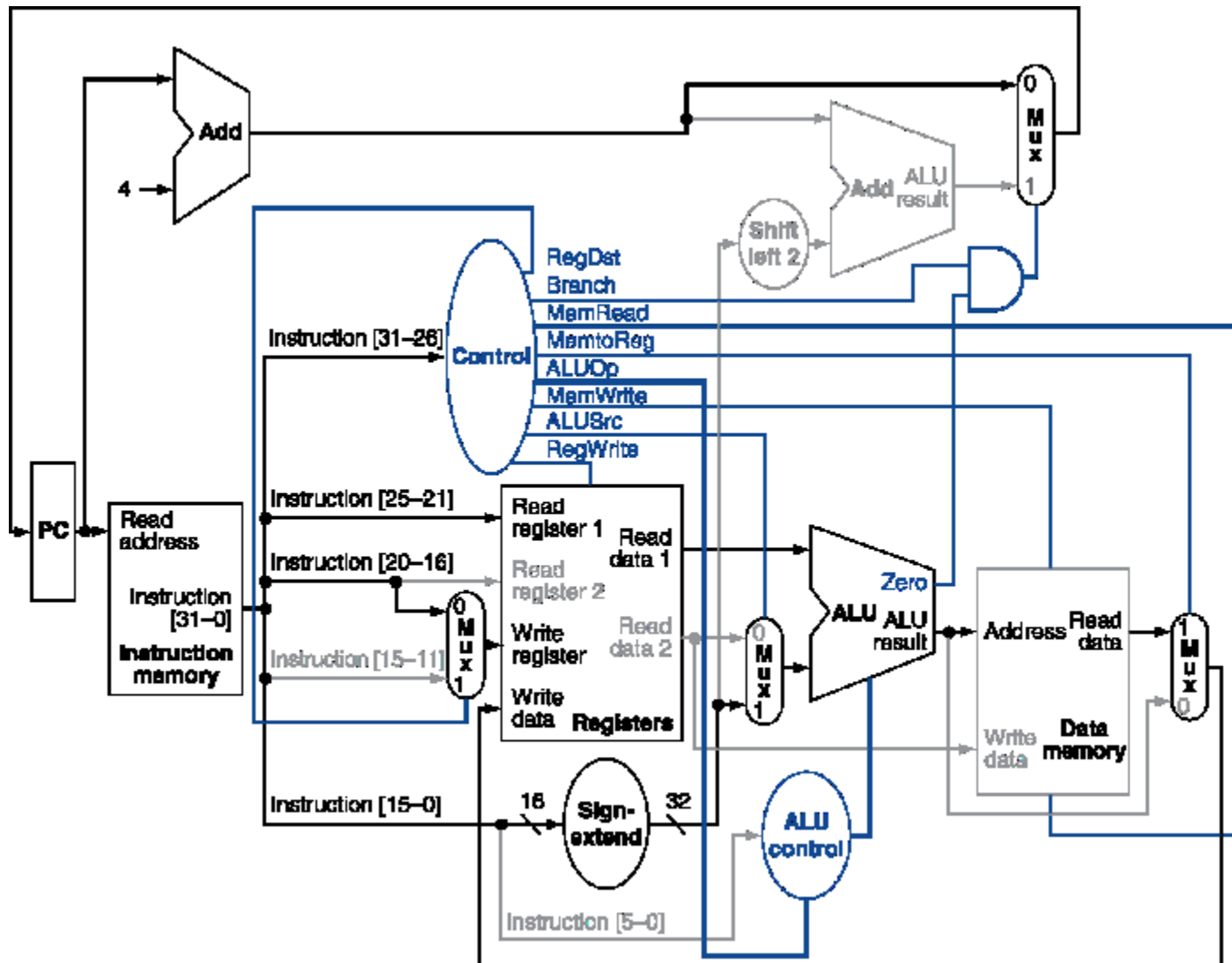
# Datapath With Control



# R-Type Instruction

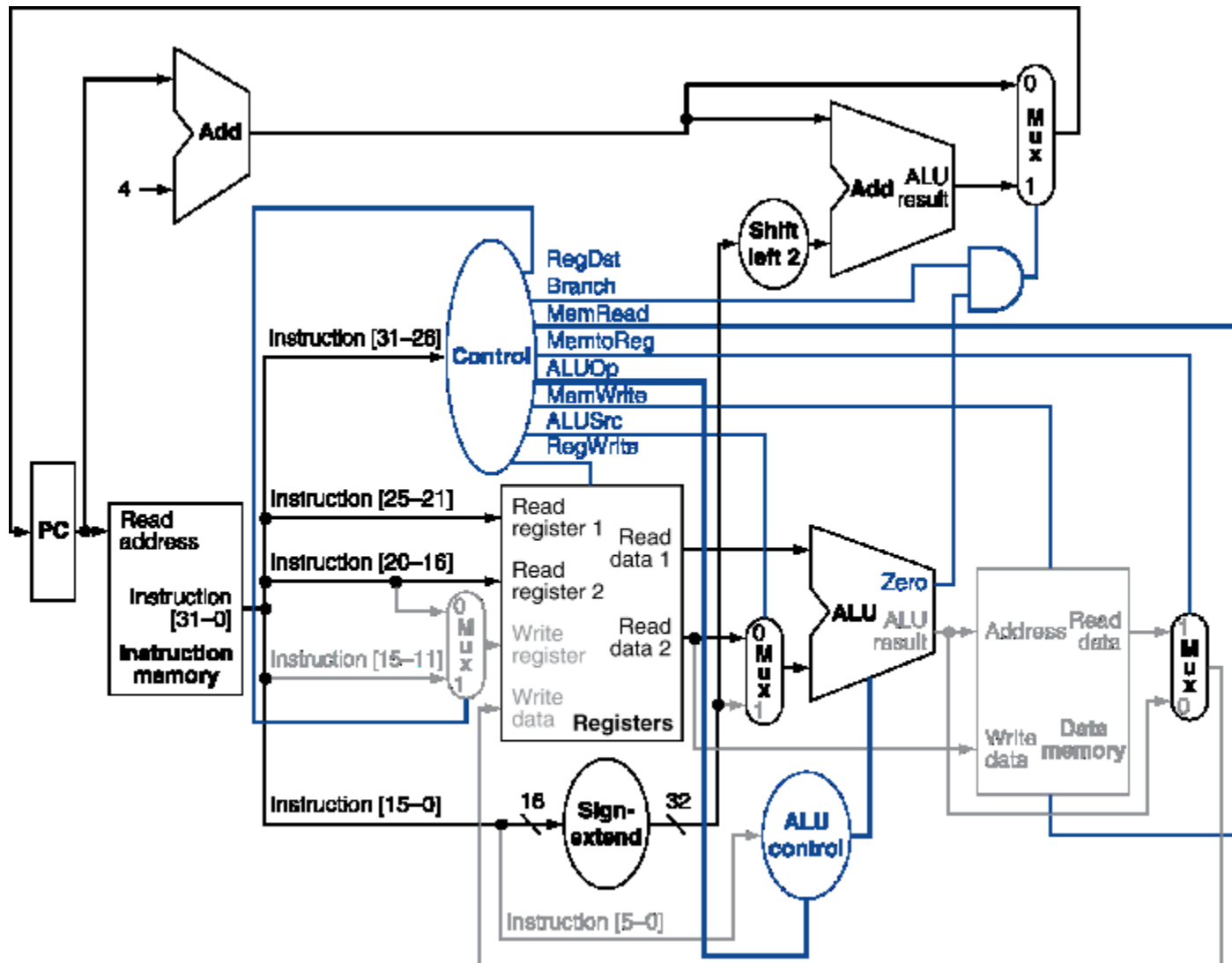


# Load Instruction

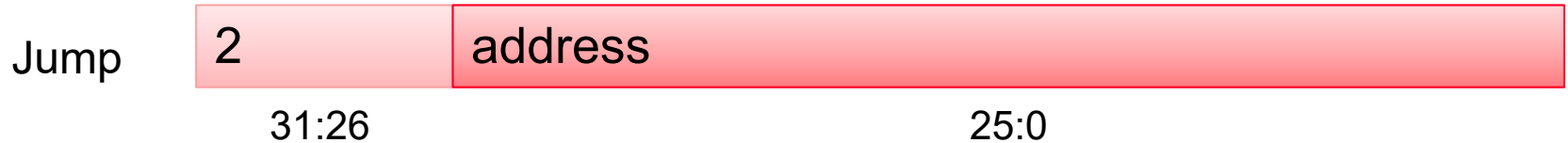




# Branch-on-Equal Instruction



## Implementing Jumps



- ❑ Jump uses word address

- ❑ (How) Update PC with concatenation of

- Top 4 bits of old PC
- 26-bit jump address
- 00

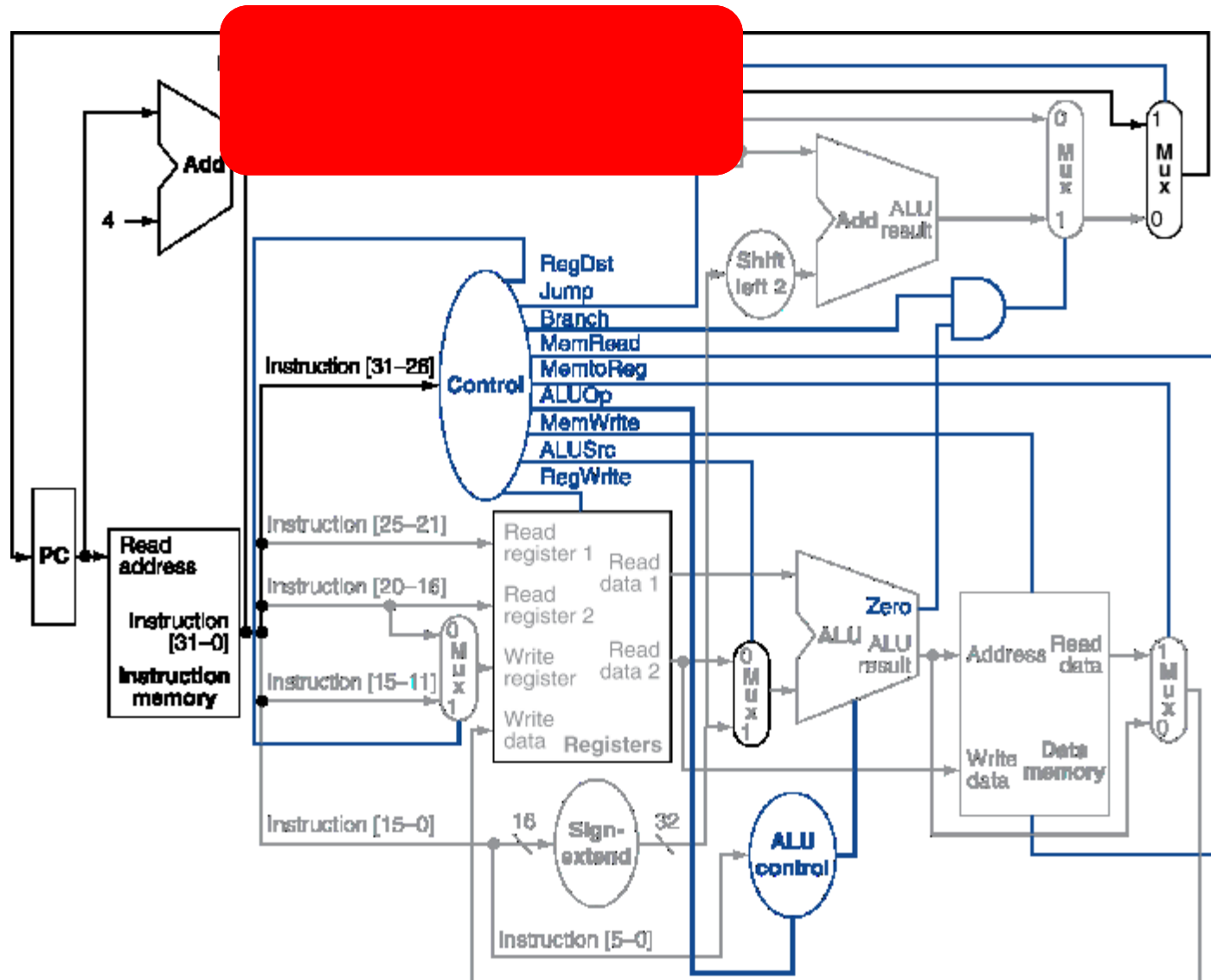


- ❑ (When) Need an extra control signal decoded from opcode



*concatenation* /kən,kæti'neɪʃn/: phép nối chuỗi

# Datapath With Jumps Added



# Performance Issues

- ❑ Longest delay determines clock period

- ❑ Critical path: load instruction

LW	$R_D, \text{OFF16}(R_S)$	$R_D = \text{MEM32}(R_S + \text{OFF16}^\pm)$
----	--------------------------	----------------------------------------------

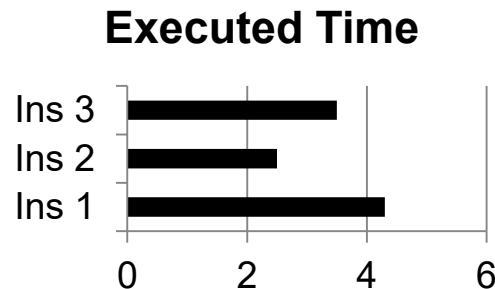
- ❑ Instruction memory → register file → ALU → data memory → register file

- ❑ Not feasible to vary period for different instructions

- ❑ Violates design principle

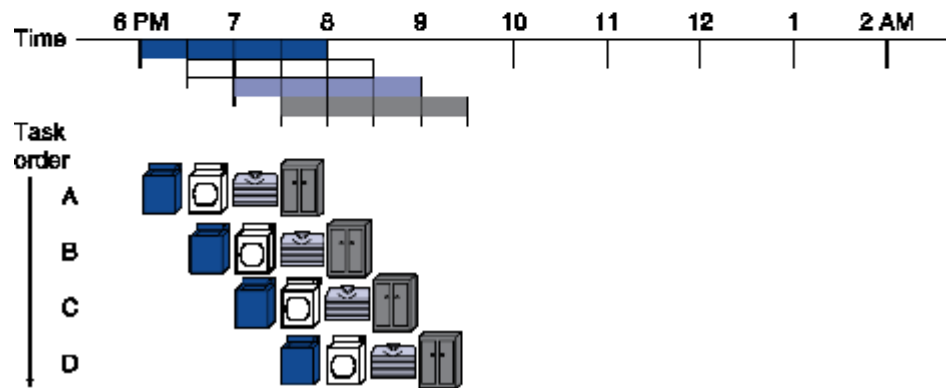
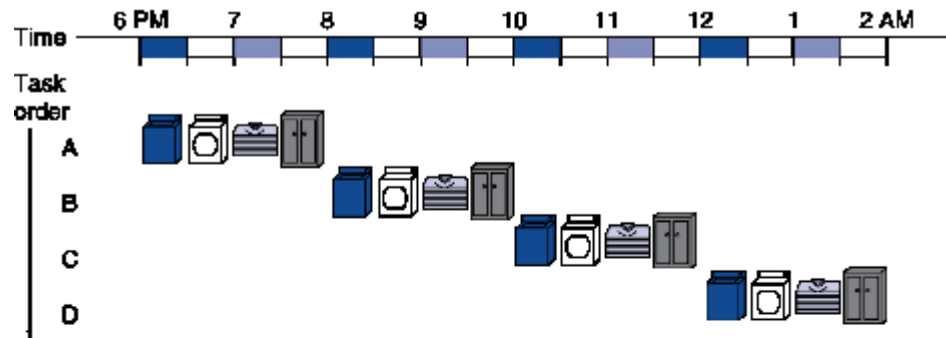
- ❑ Making the common case fast

- ❑ We will improve performance by pipelining



# Pipelining Analogy

- ❑ Pipelined laundry: overlapping execution
  - | Parallelism improves performance



Ford Model T –  
100 Years Later

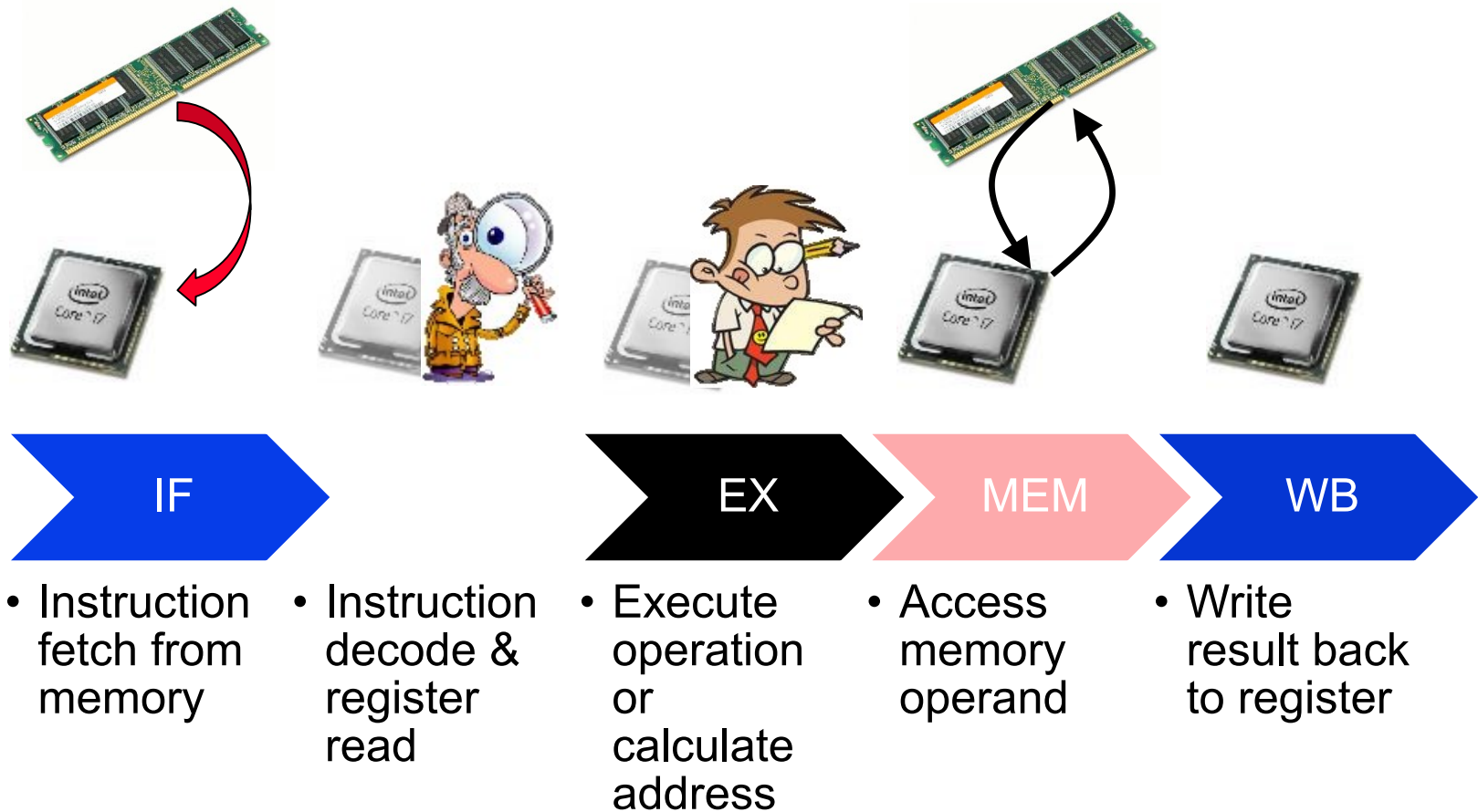


Ford Model T –  
100 Years Later



# MIPS Pipeline

- Five stages, one step per stage

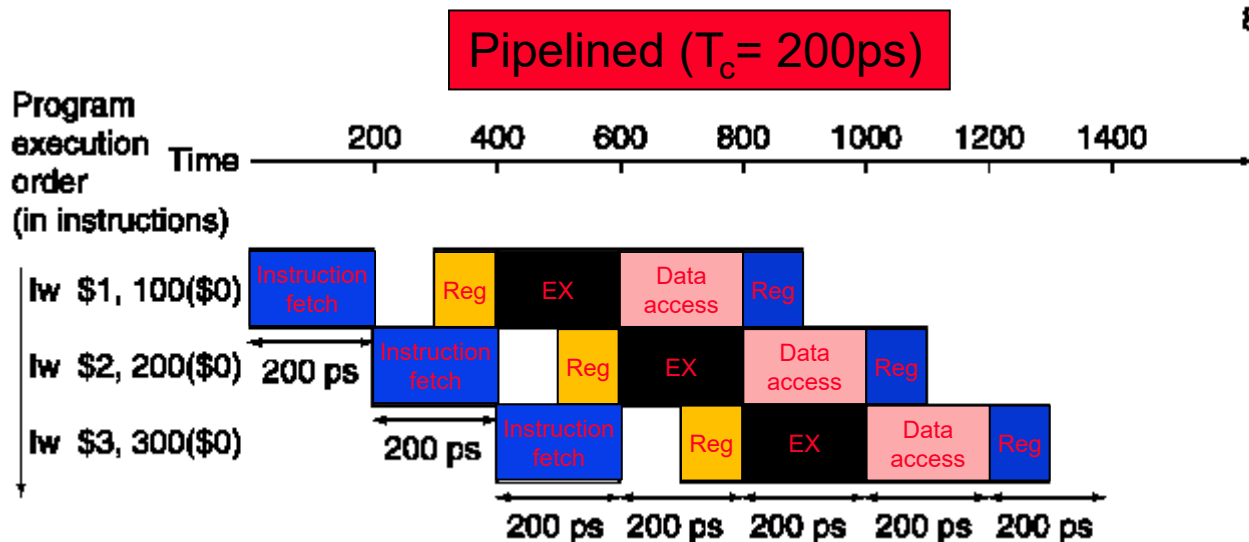
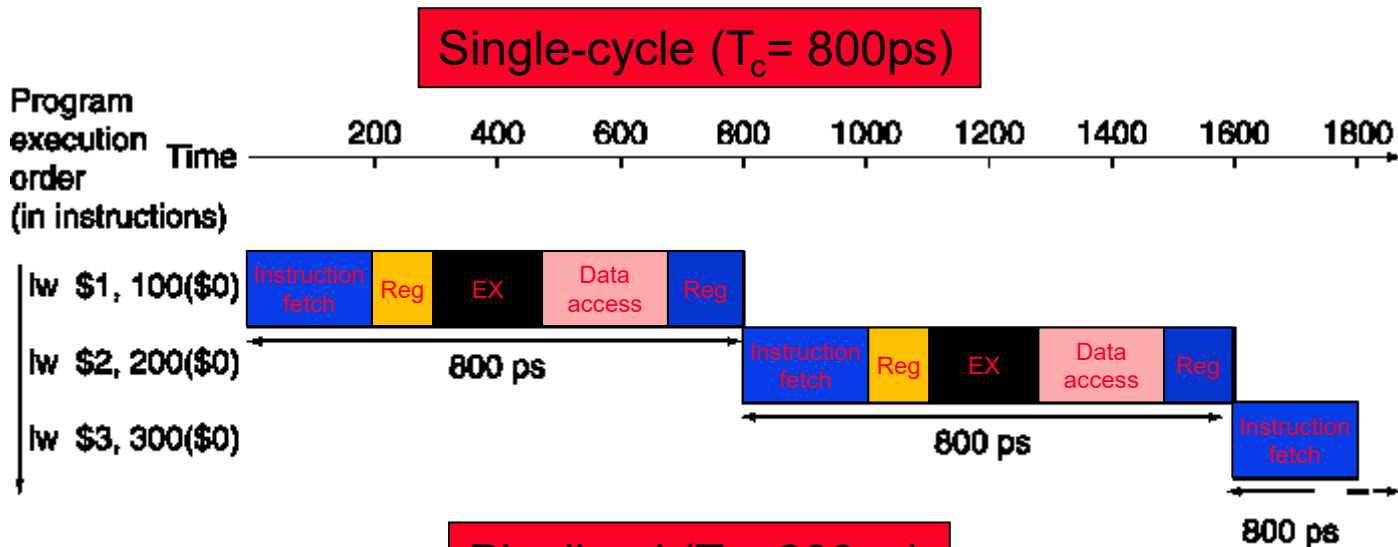


# Pipeline Performance

- ❑ Assume time for stages is
  - | 100ps for register read or write
  - | 200ps for other stages
- ❑ Compare pipelined datapath with single-cycle datapath

	IF		EX	MEM	WB	
Instr	Instr 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

# Pipeline Performance





# Pipeline Speedup

---

- ❑ If all stages are balanced

- | i.e., all take the same time

- |  $\text{Time between instructions}_{\text{pipelined}}$

- $\frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$

---

- ❑ If not ~~balanced~~, speedup is less

- |  $T_{\text{general}} = \text{Max}(T_{\text{stage}_1}, T_{\text{stage}_2}, \dots)$

- ❑ Speedup due to increased throughput

- | Latency (time for each instruction) does not decrease

# Pipelining and ISA Design

---

- ❑ MIPS ISA designed for pipelining

## All instructions are 32-bits

- Easier to fetch and decode in one cycle
- c.f. x86: 1- to 17-byte instructions



## Few and regular instruction formats

- Can decode and read registers in one step



## Load/store addressing

- Can calculate address in 3<sup>rd</sup> stage, access memory in 4<sup>th</sup> stage

## Alignment of memory operands

- Memory access takes only one cycle

# ISA: MIPS <=> Intel Core i

MIPS ↓

Intel →

Instruction 8 bits

Instruction 24bits

Instruction 64bits

Memory			
Address	Content		
8002037C	00 00 00 00	NOP	
80020380	00 00 00 00	NOP	
80020384	00 00 00 00	NOP	
80020388	0C 00 80 28	JAL	0x8028
8002038C	24 04 00 03	ADDIU	\$04, \$00, 0x3
80020390	0C 00 81 20	JAL	0x8120
80020394	00 00 00 00	NOP	
80020398	84 04 EA 2A	LH	\$04, 0xea2a(\$00)
8002039C	30 84 BF FF	ANDI	\$04, \$04, 0xbfff
800203A0	A4 04 EA 2A	SH	\$04, 0xea2a(\$00)
800203A4	3C 08 BF A0	LUI	\$08, 0xbfa0
800203A8	A1 00 00 00	SB	\$00, 0x0(\$08)
800203AC	3C 08 80 02	LUI	\$08, 0x8002
800203B0	25 08 06 5C	ADDIU	\$08, \$08, 0x65c
800203B4	3C 09 80 00	LUI	\$09, 0x8000
800203B8	35 29 00 80	ORI	\$09, \$09, 0x80
800203BC	8D 0A 00 00	LW	\$10, 0x0(\$08)
800203C0	8D 0B 00 04	LW	\$11, 0x4(\$08)
800203C4	AD 2A 00 00	SW	\$10, 0x0(\$09)
800203C8	AD 2B 00 04	SW	\$11, 0x4(\$09)
800203CC	40 02 60 00	MFC0	\$12, \$02
800203D0	3C 03 FF BF	LUI	\$03, 0xffbf
800203D4	34 63 FF FF	ORI	\$03, \$03, 0xffff
800203D8	00 43 10 24	AND	\$02, \$02, \$03
800203DC	34 42 00 01	ORI	\$02, \$02, 0x1
800203E0	34 42 20 00	ORI	\$02, \$02, 0x2000
800203E4	40 82 60 00	MTC0	\$12, \$02

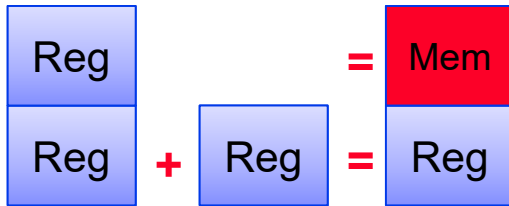
```
o.cpp Disassembly
File View Termin p
#include <stdio.h>
#include <stdlib.h>

int main(void, {
00483e4:      55                push    %ebp
00483e5:      89 e5             mov     %esp,%ebp
00483e7:      83 e4 f0          and     $0xffffffff0,%esp
00483ea:      83 ec 20          sub     $0x20,%esp
        int x = 0x0A0B0371;
00483ed:      c7 44 24 1c 71 03 0b movl    $0xa0b0371,0x1c(%esp)
00483f4:      0a               int     y = 0x03;
00483f5:      c7 44 24 18 03 00 00 movl    $0x3,0x18(%esp)
00483fc:      00               int     z = x + y;
00483fd:      8b 44 24 18       mov     0x18(%esp),%eax
0048401:      8b 54 24 1c       mov     0x1c(%esp),%edx
0048405:      8d 04 02          lea     (%edx,%eax,1),%eax
0048408:      89 44 24 14       mov     %eax,0x14(%esp)
        printf("%i + %i = %i\n",x,y,z);
004840c:      b8 00 85 04 08    mov     $0x8048500,%eax
0048411:      8b 54 24 14       mov     0x14(%esp),%edx
0048415:      89 54 24 0c       mov     %edx,0xc(%esp)
0048419:      8b 54 24 18       mov     0x18(%esp),%edx
004841d:      89 54 24 08       mov     %edx,0x8(%esp)
0048421:      8b 54 24 1c       mov     0x1c(%esp),%edx
0048425:      89 54 24 04       mov     %edx,0x4(%esp)
0048429:      89 04 24          mov     %eax, (%esp)
004842c:      e8 eb fe ff ff    call    804831c <printf@plt>
        return EXIT_SUCCESS;
0048431:      b8 00 00 00 00    mov     $0x0,%eax
}
0048436:      c9               leave
```

# ISA: MIPS <> Intel Core i

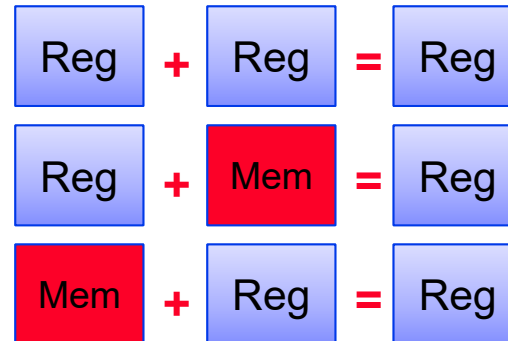
## MIPS

❑ LW S0,3(T0)  
ADD S1, S1,S0



## Intel Core i

❑ ADD AX,[SI+3]



Type	format (bits)					
-31-						-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

❑ ADD AX,BX

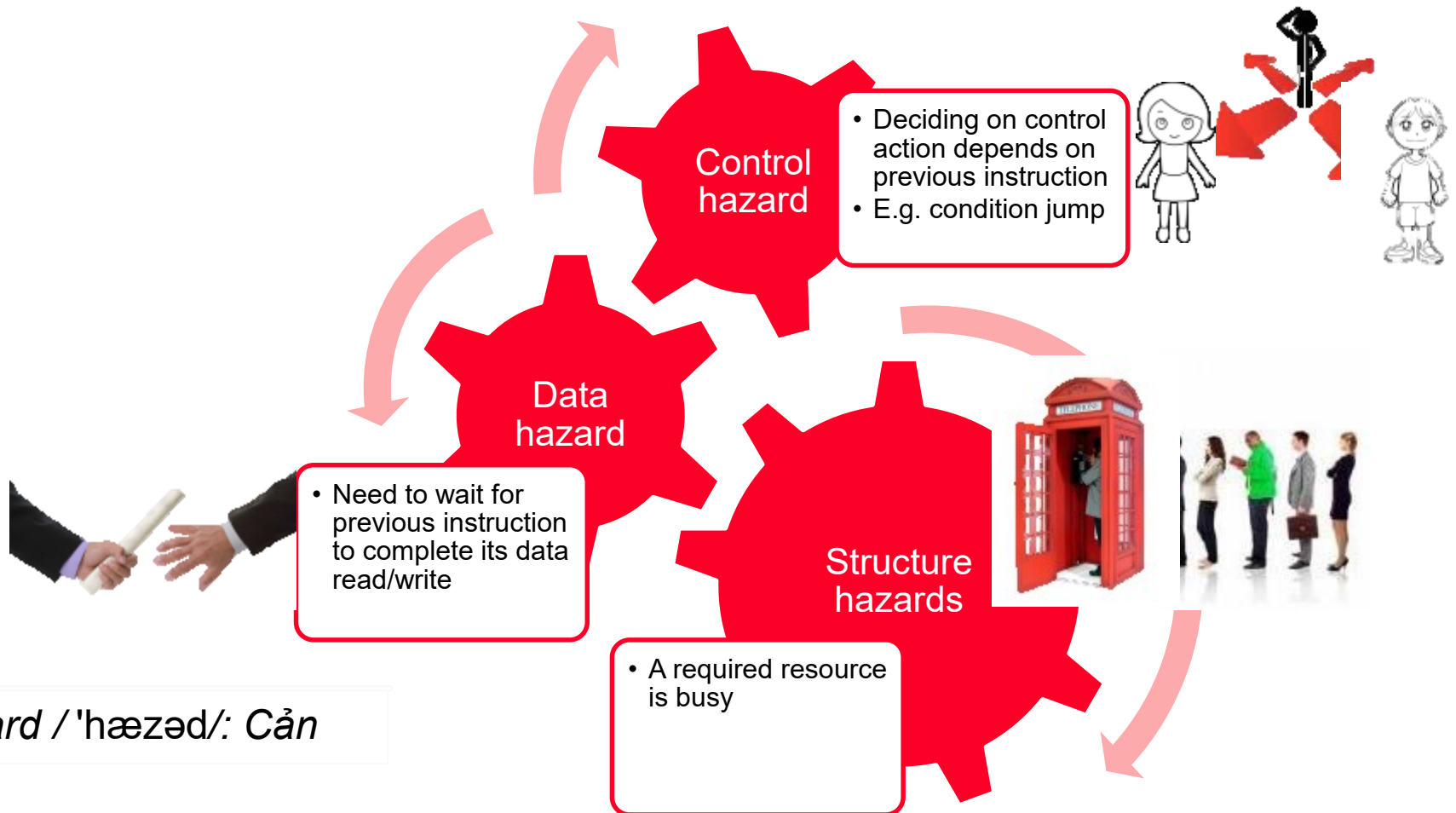
❑ ADD AX,[BX]

❑ ADD AX,245

# Hazards

- ❑ Situations that prevent the next cycle

start  the next instruction in



*Hazard* /'hæzəd/: Cẩn  
trở

# Structure Hazards



- ❑ Conflict for use of **a resource**
- ❑ In MIPS pipeline with **a single memory**
  - | Load/store requires data access
  - | Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- ❑ Hence, pipelined datapaths require **separate instruction/data memories**
  - | Or separate instruction/data caches

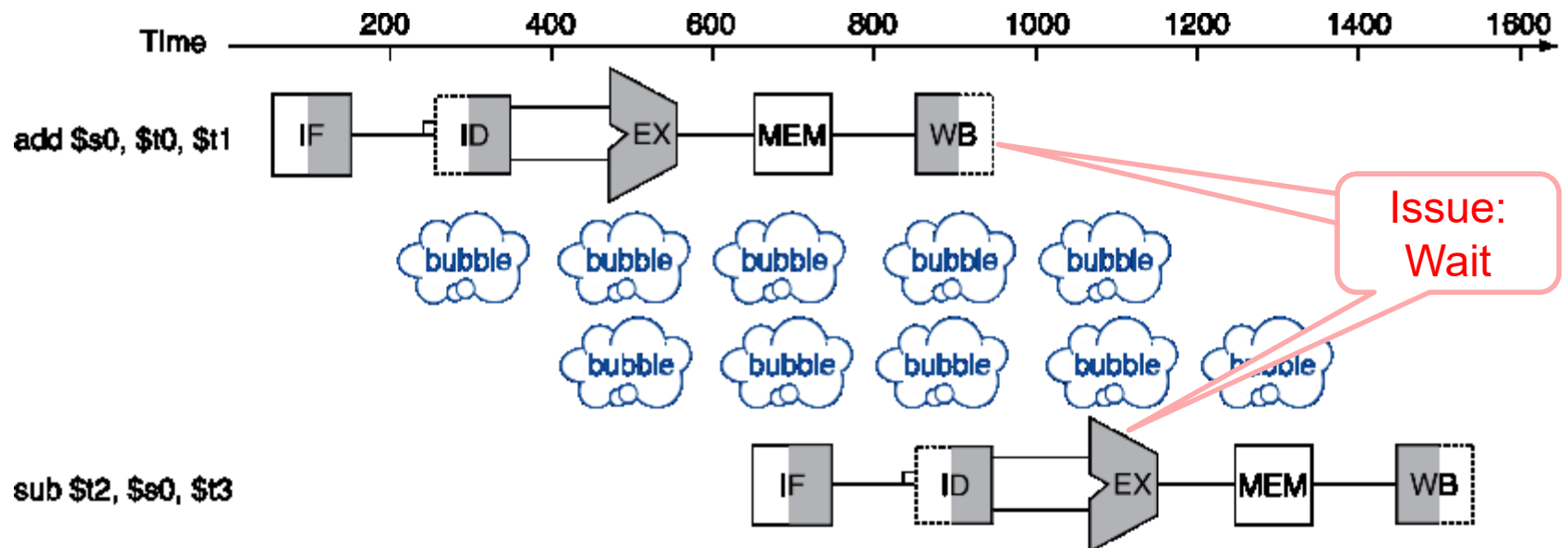
*stall* /stɔ:l/: trì hoãn

# Data Hazards



- ❑ An instruction depends on completion of data access by a previous instruction

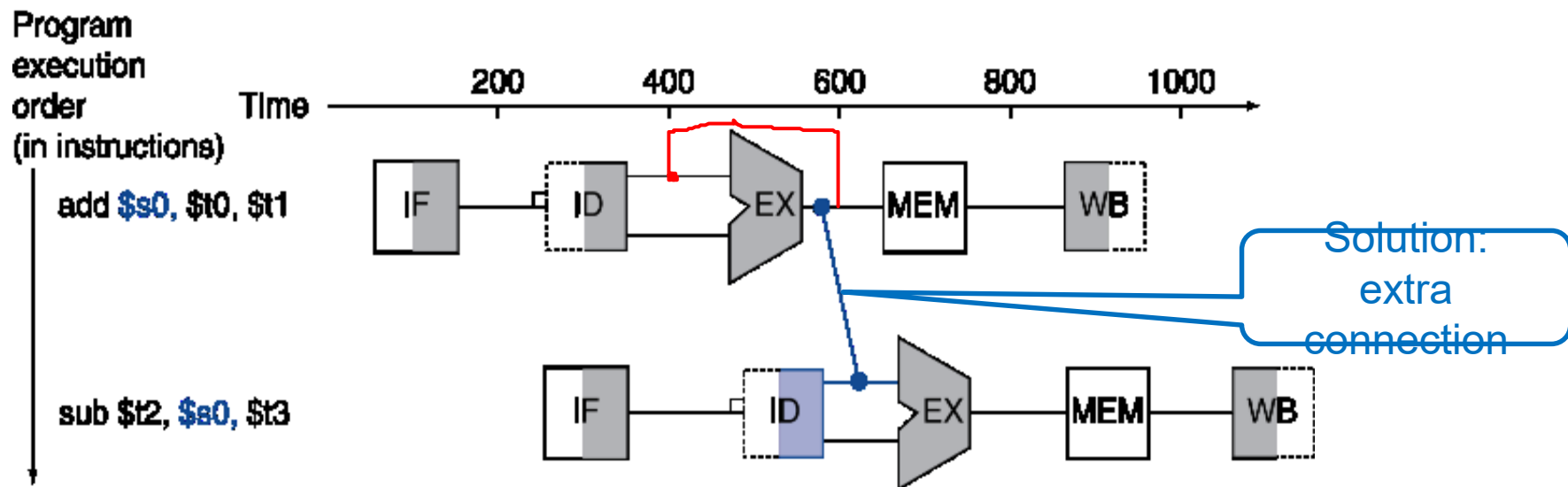
```
| add    $s0, $t0, $t1  
sub     $t2, $s0, $t3
```





## Forwarding (aka Bypassing)

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

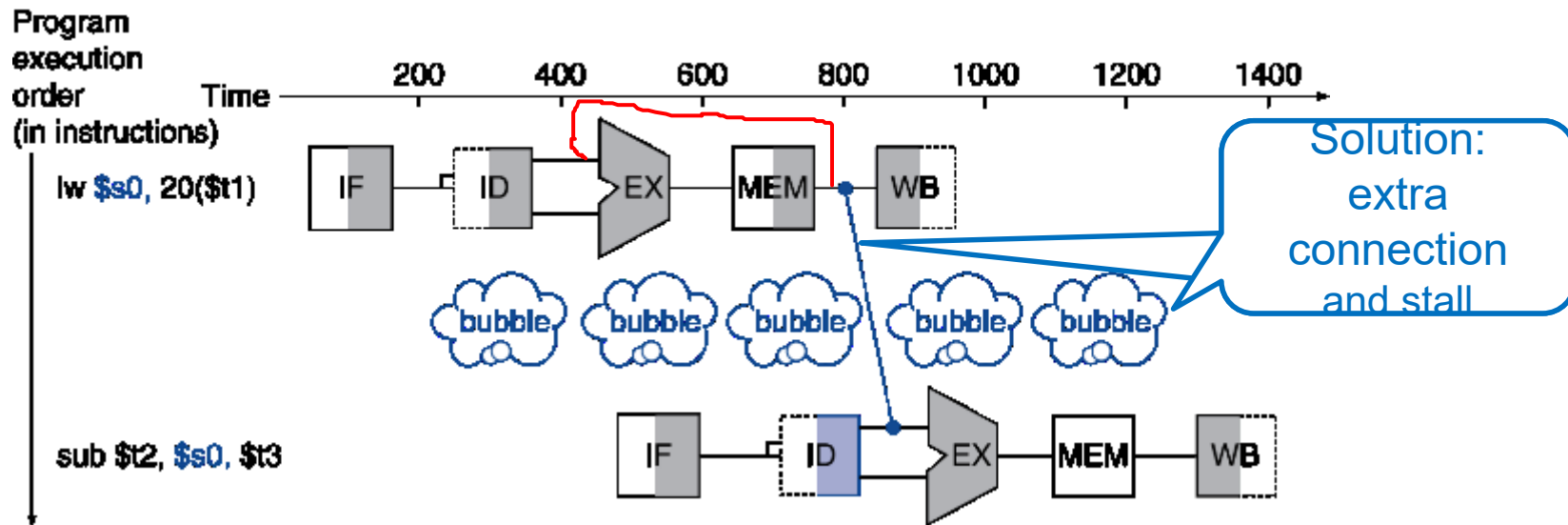






# Load-Use Data Hazard

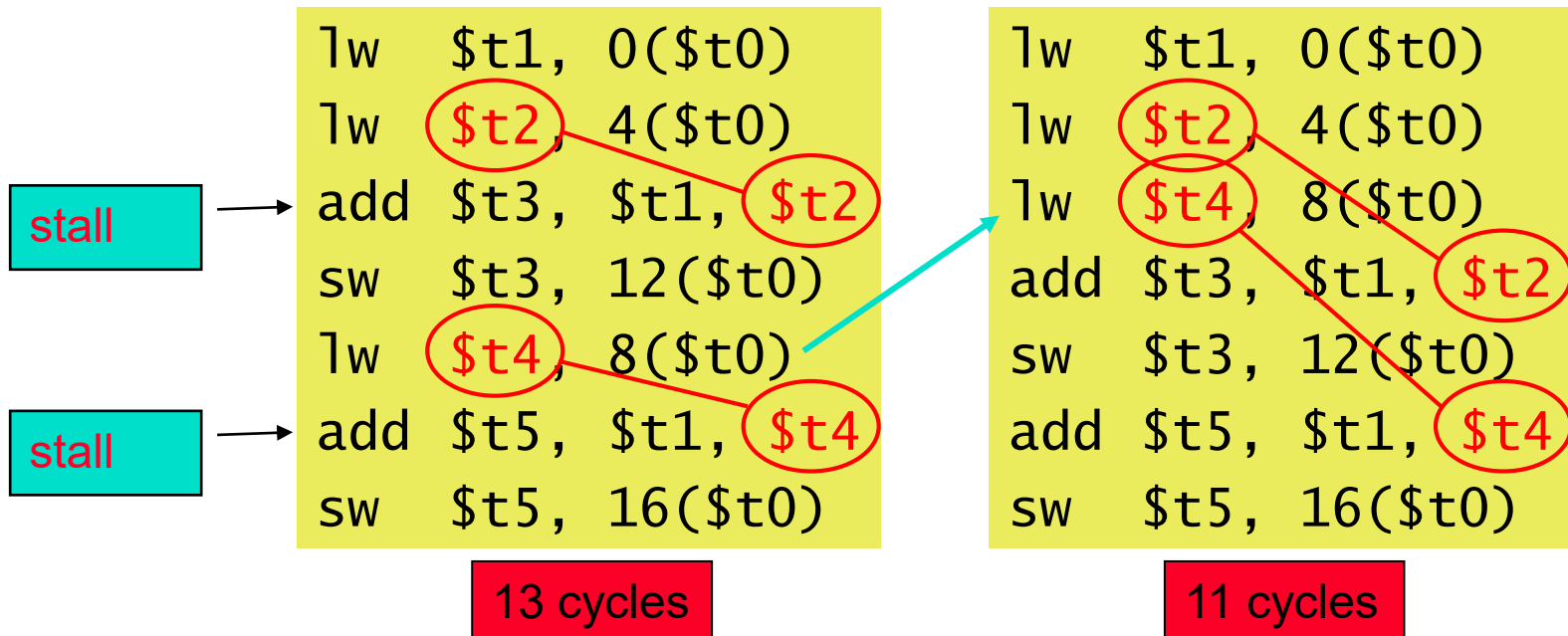
- ❑ Can't always avoid stalls by forwarding
  - | If value not computed when needed
  - | Can't forward backward in time!



# Code Scheduling to Avoid Stalls



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



# Control Hazards

---

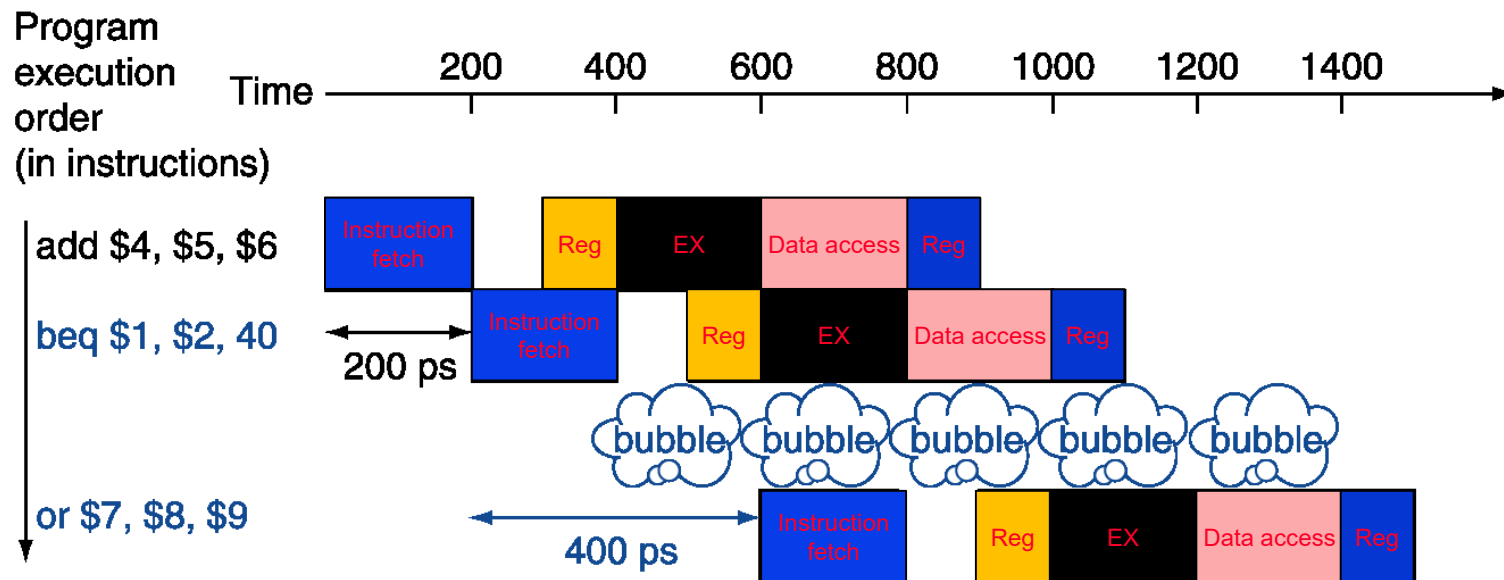


- ❑ Branch determines flow of control
  - | Fetching next instruction depends on branch outcome
  - | Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- ❑ In MIPS pipeline
  - | Need to compare registers and compute target early in the pipeline
  - | Add hardware to do it in ID stage

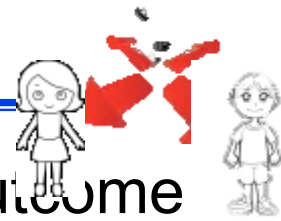
# Stall on Branch



- ❑ Wait until branch outcome determined before fetching next instruction

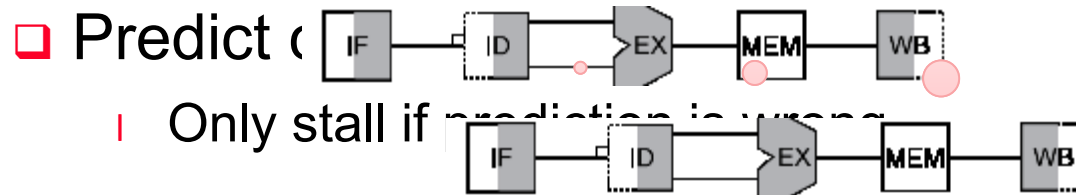


# Branch Prediction



- ❑ Longer pipelines can't readily determine branch outcome early

- Stall penalty becomes unacceptable



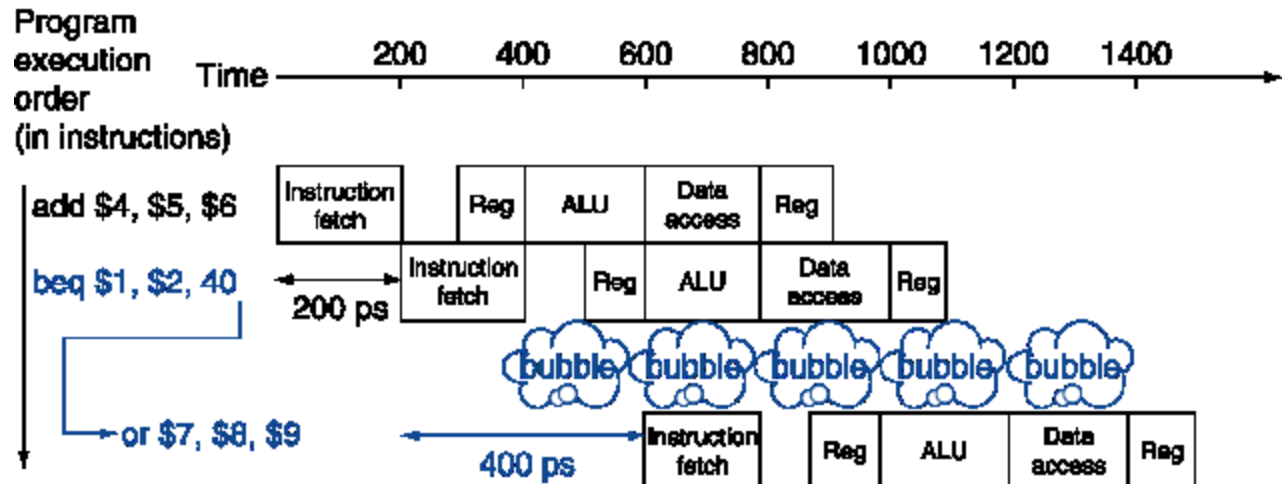
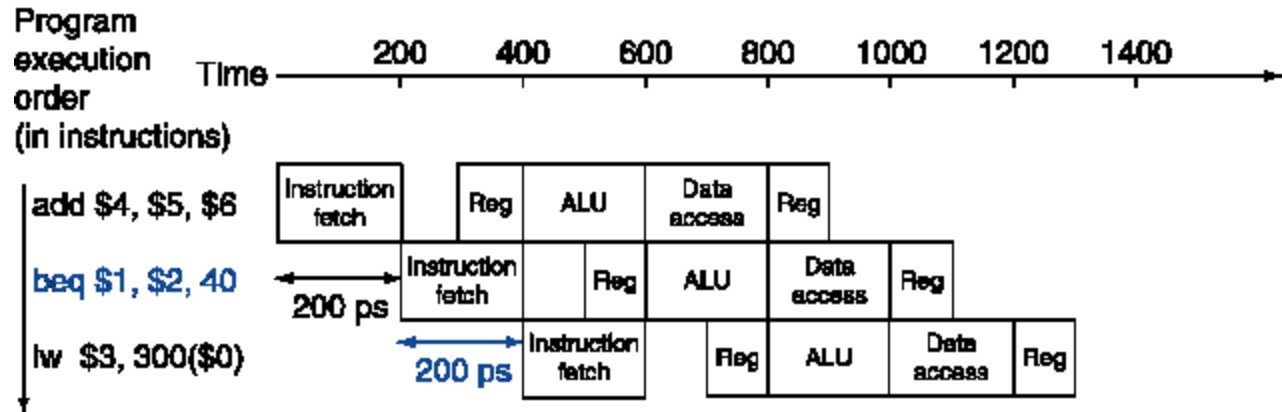
**Stop,**  
I want to  
change,  
but too late

- ❑ In MIPS pipeline

- Can predict branches not taken
  - Fetch instruction after branch, with no delay

*readily /'redili/ ~ easily*

# MIPS with Predict Not Taken





## Static branch prediction

- Based on typical branch behavior
- Example: loop and if-statement branches
  - Predict backward branches taken
  - Predict forward branches not taken

## Dynamic branch prediction

- 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

# Pipeline Summary

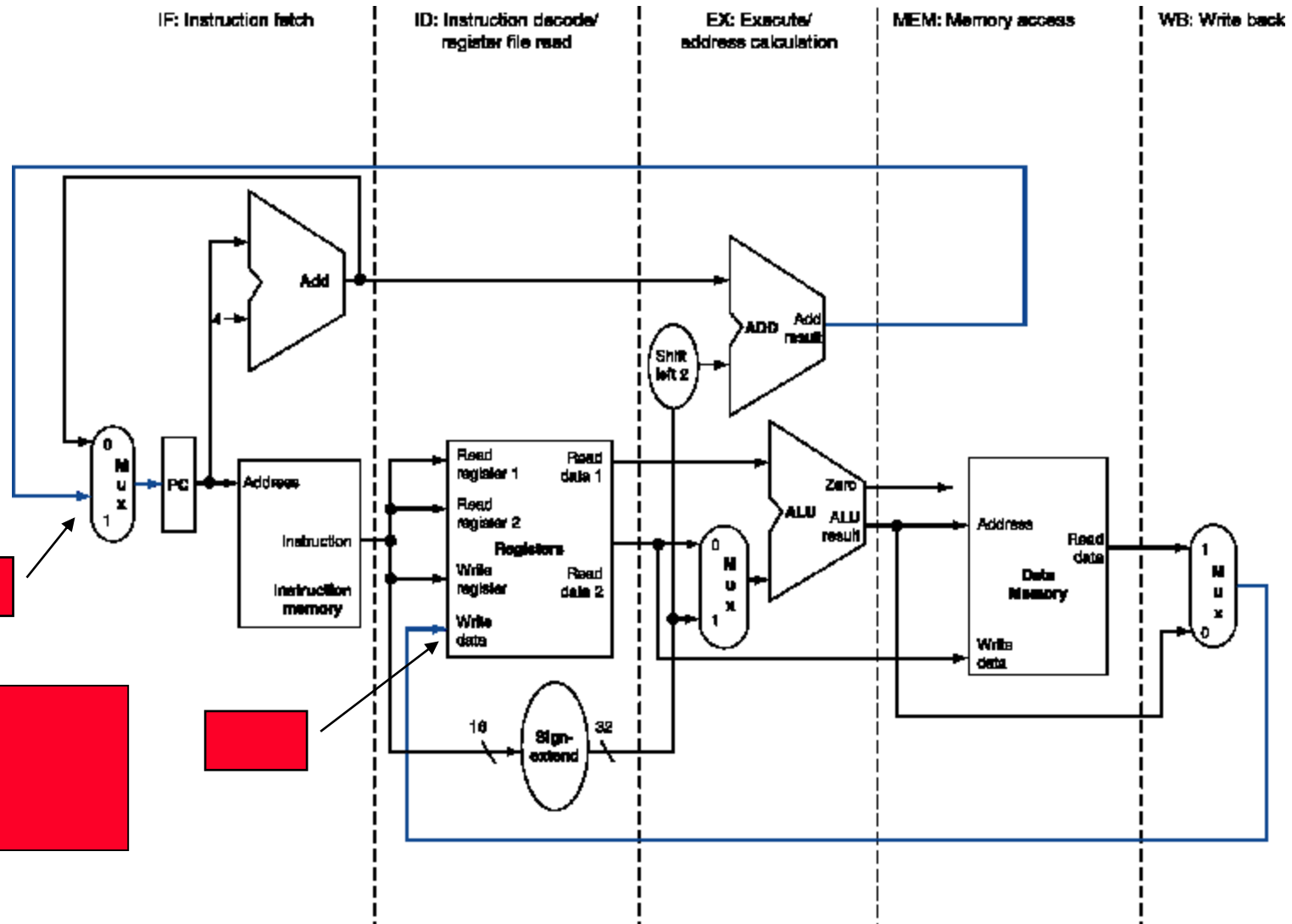
---

## The BIG Picture

- ❑ Pipelining improves performance by increasing instruction throughput
  - | Executes multiple instructions in parallel
  - | Each instruction has the same latency
- ❑ Subject to hazards
  - | Structure, data, control
- ❑ Instruction set design affects complexity of pipeline implementation

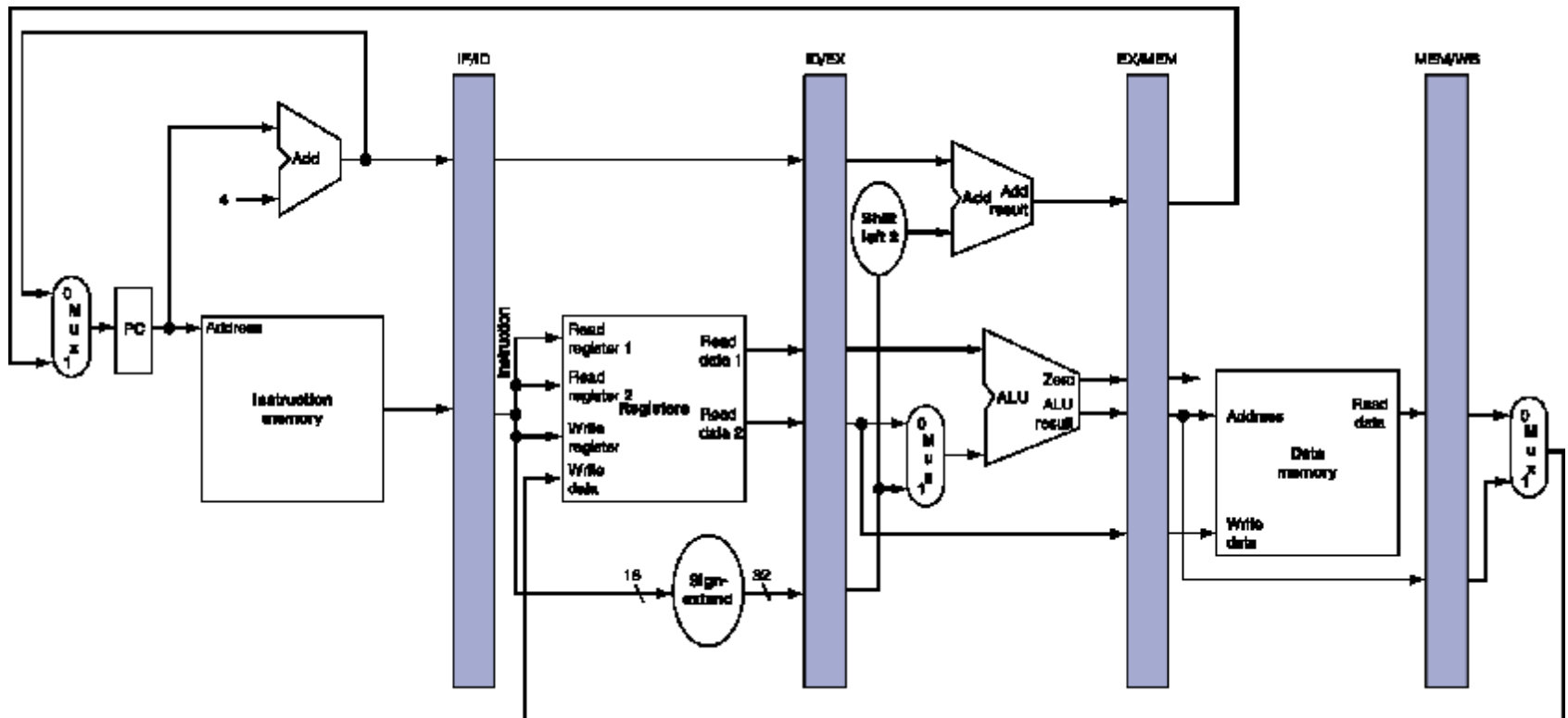


# MIPS Pipelined Datapath



# Pipeline registers

- ❑ Need registers between stages
  - | To hold information produced in previous cycle

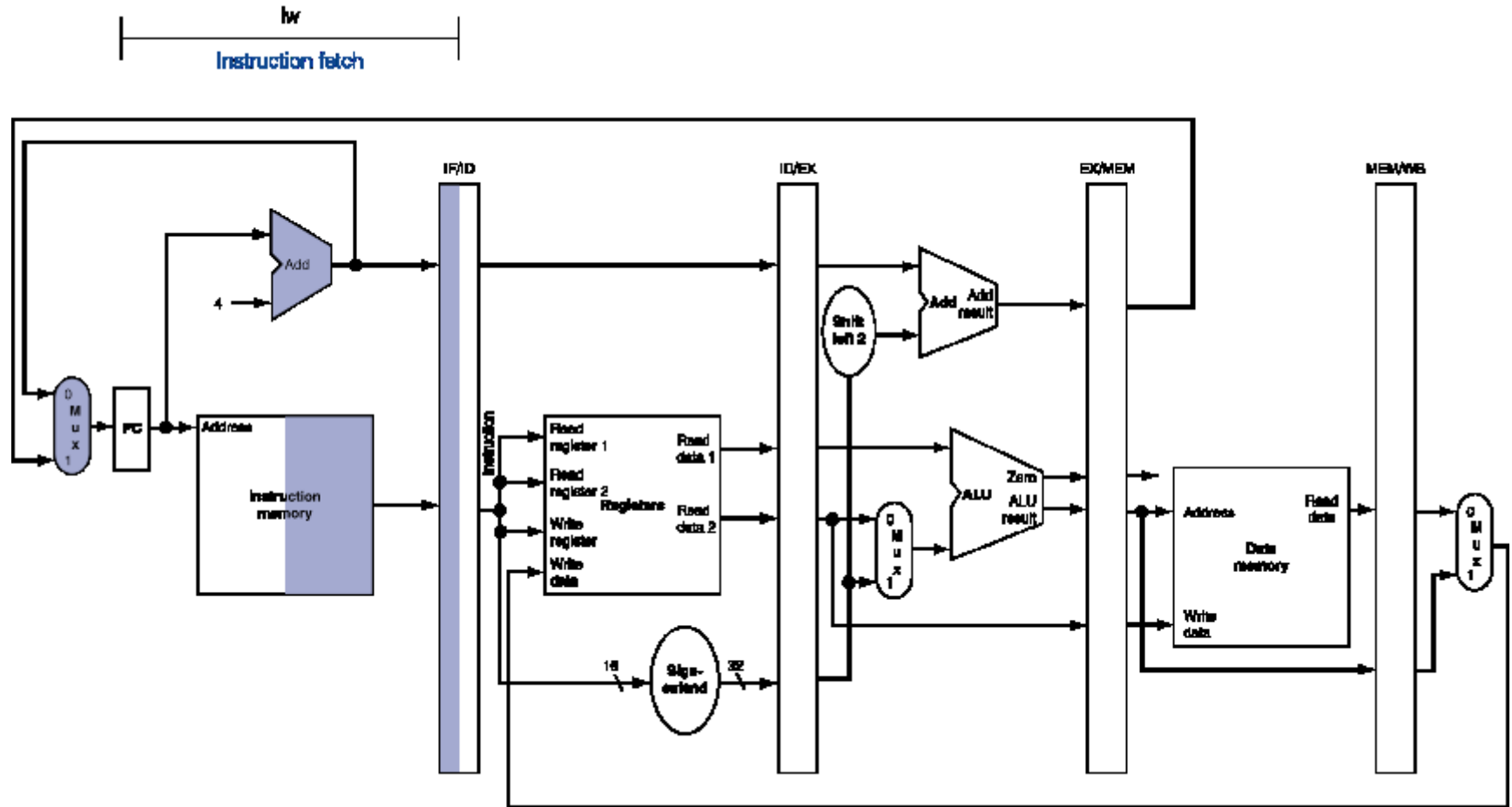


# Pipeline Operation

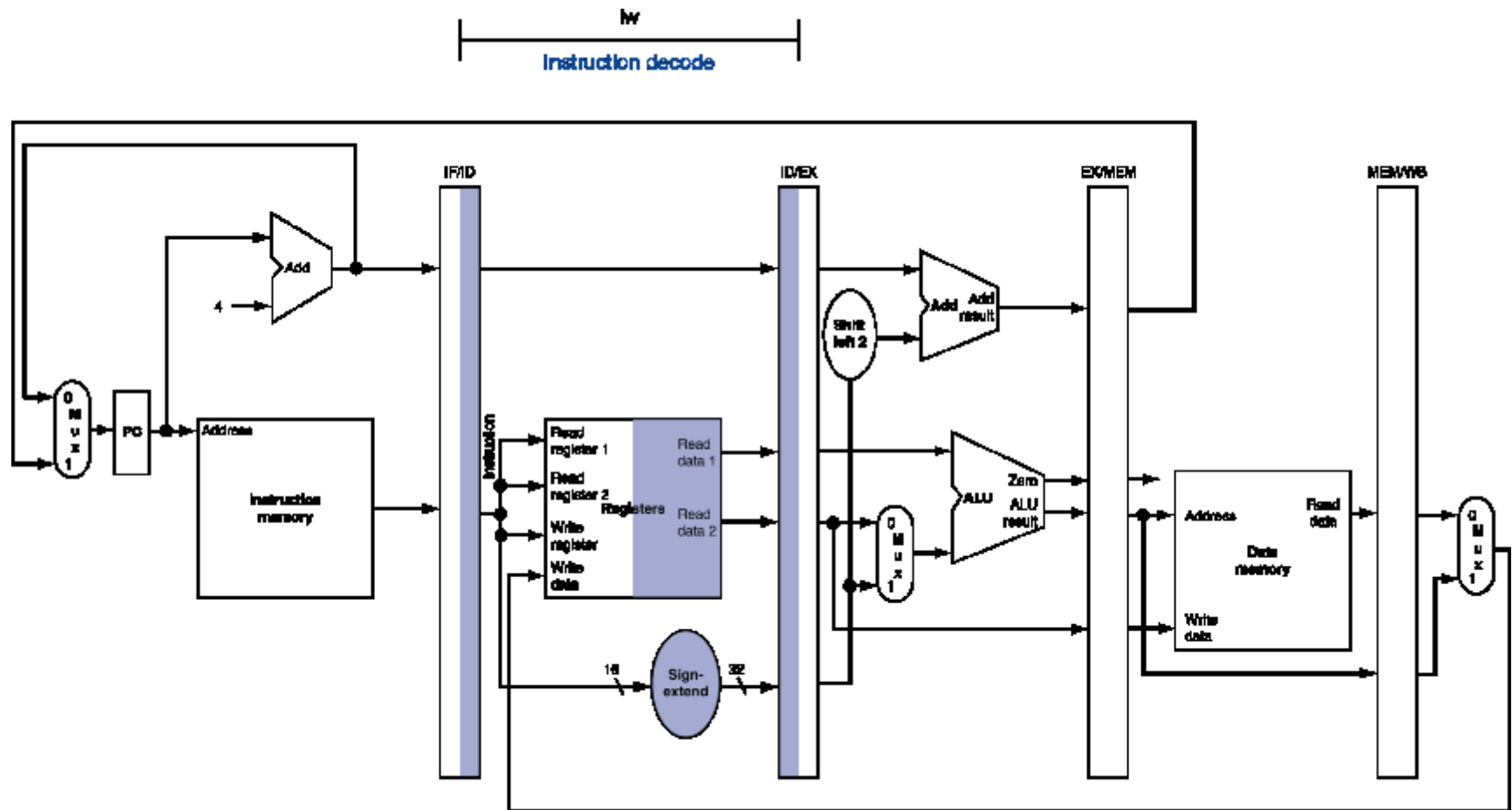
---

- ❑ Cycle-by-cycle flow of instructions through the pipelined datapath
  - | “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - | c.f. “multi-clock-cycle” diagram
    - Graph of operation over time
- ❑ We’ll look at “single-clock-cycle” diagrams for load & store

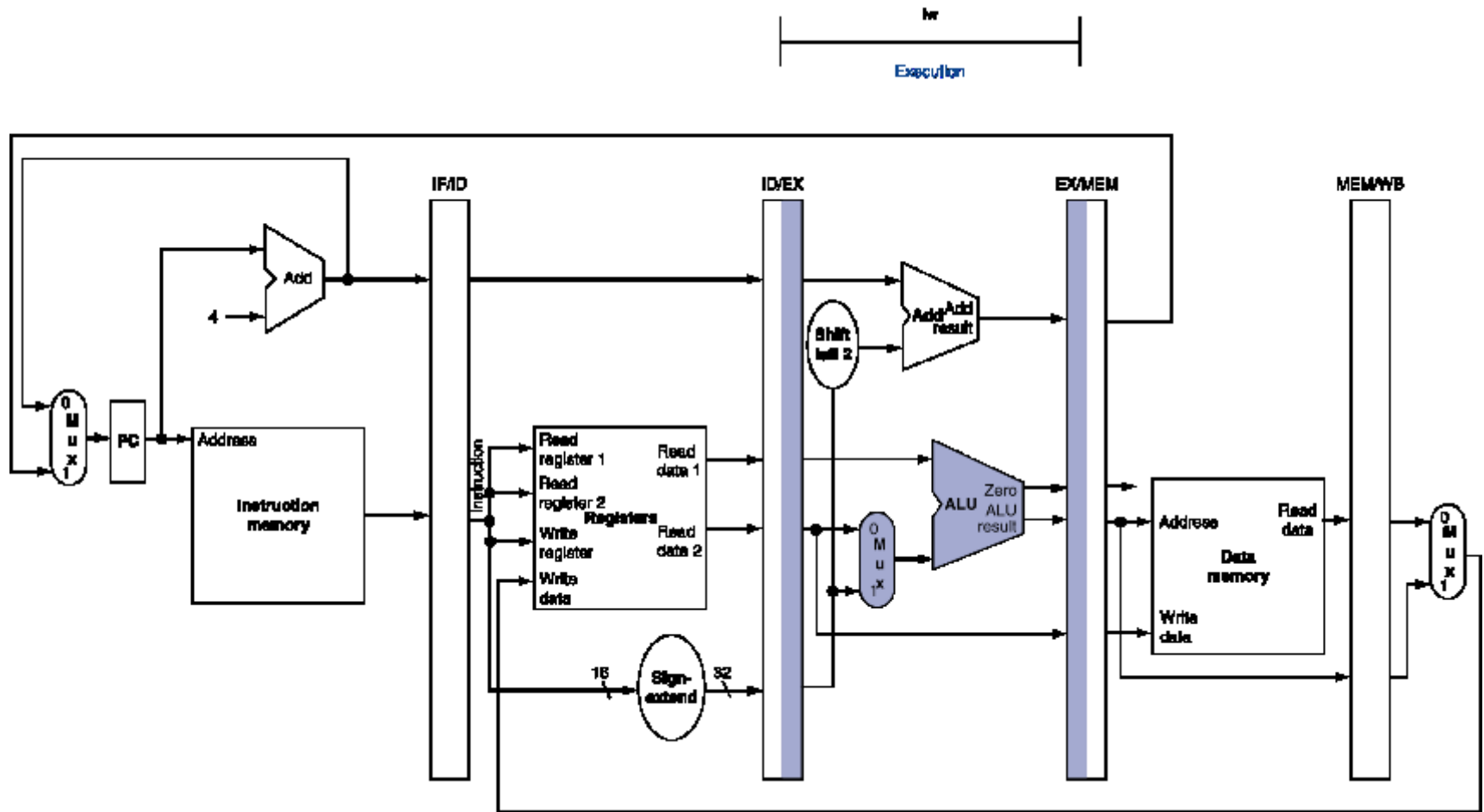
# IF for Load, Store, ...



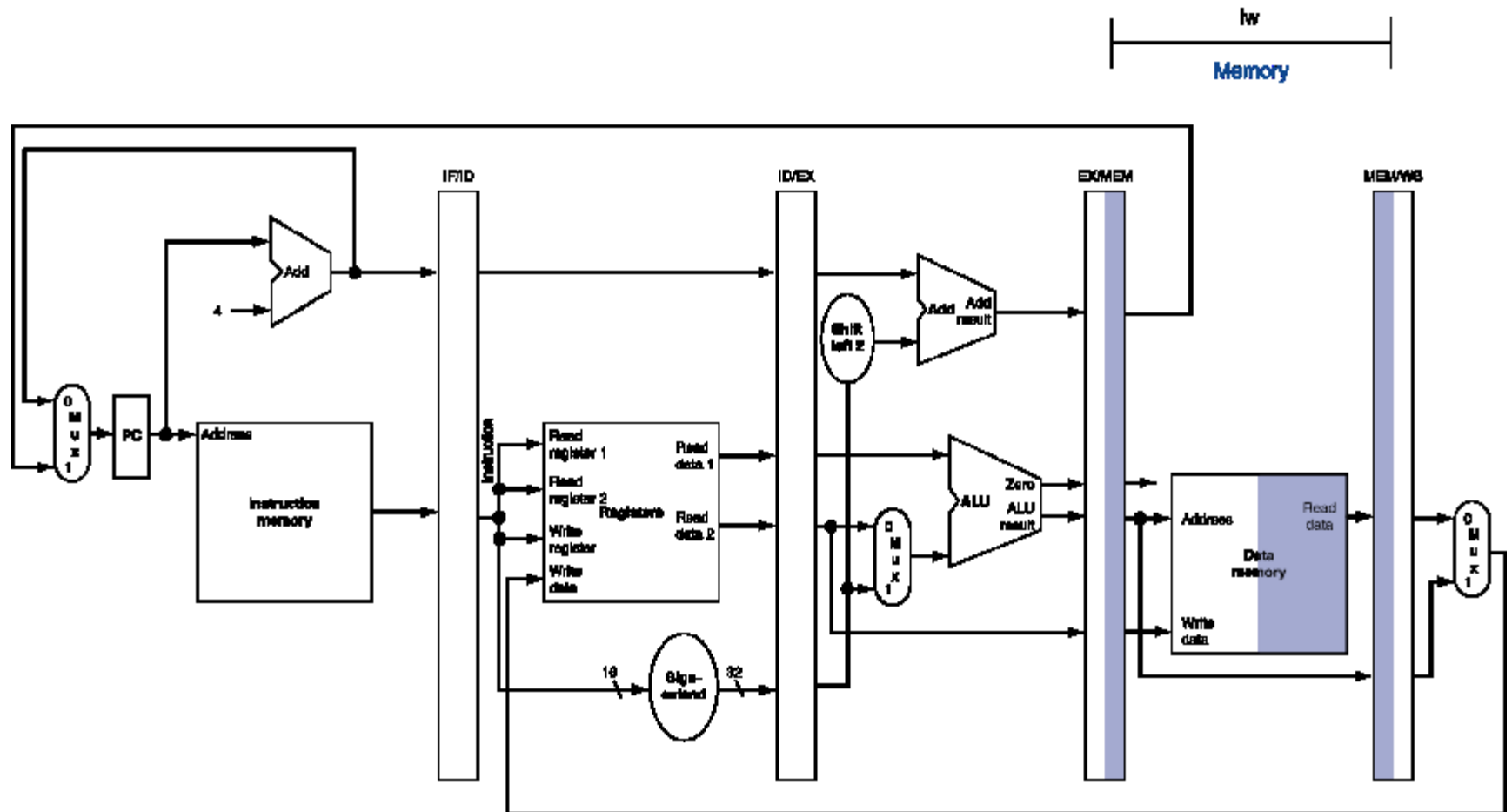
# ID for Load, Store, ...



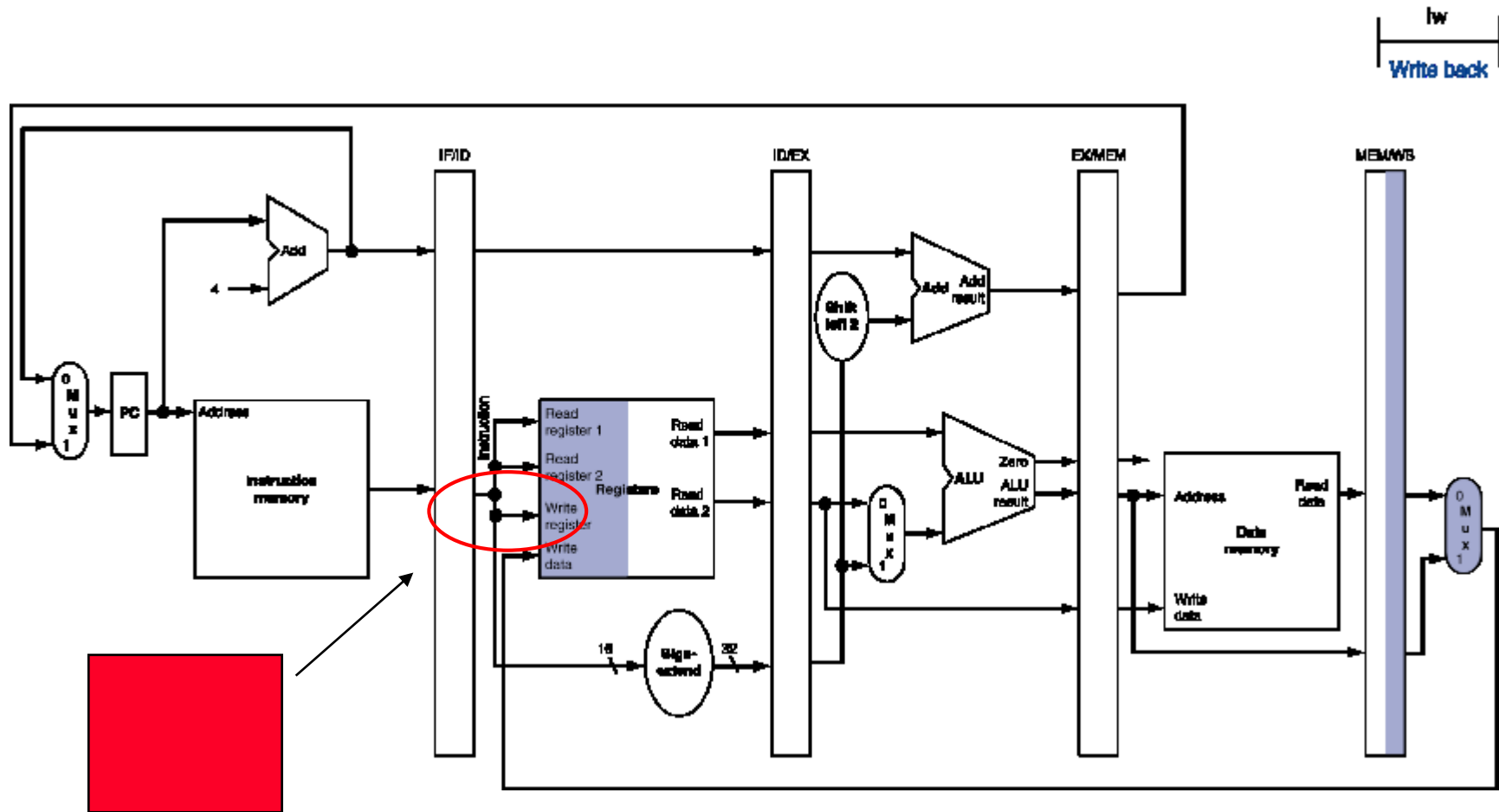
# EX for Load



# MEM for Load

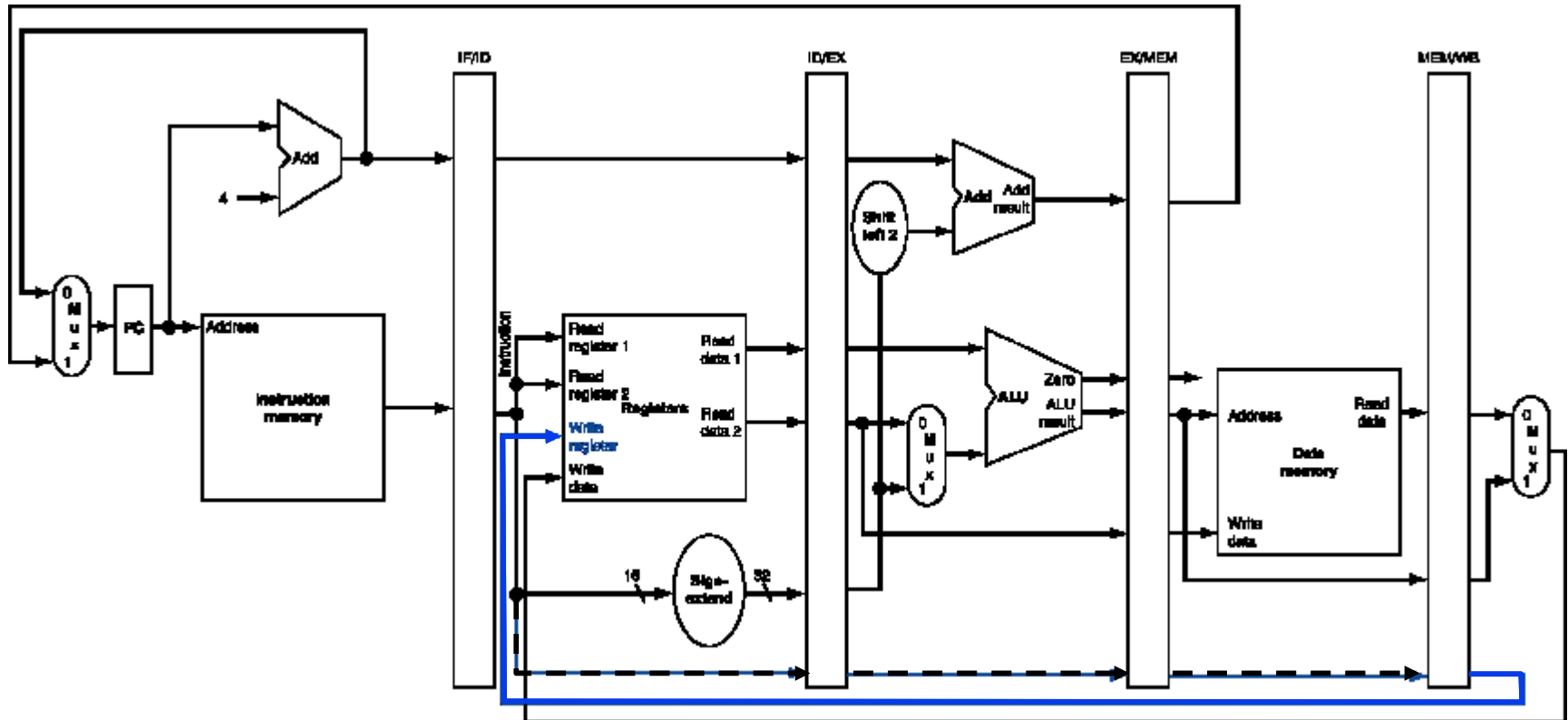


# WB for Load

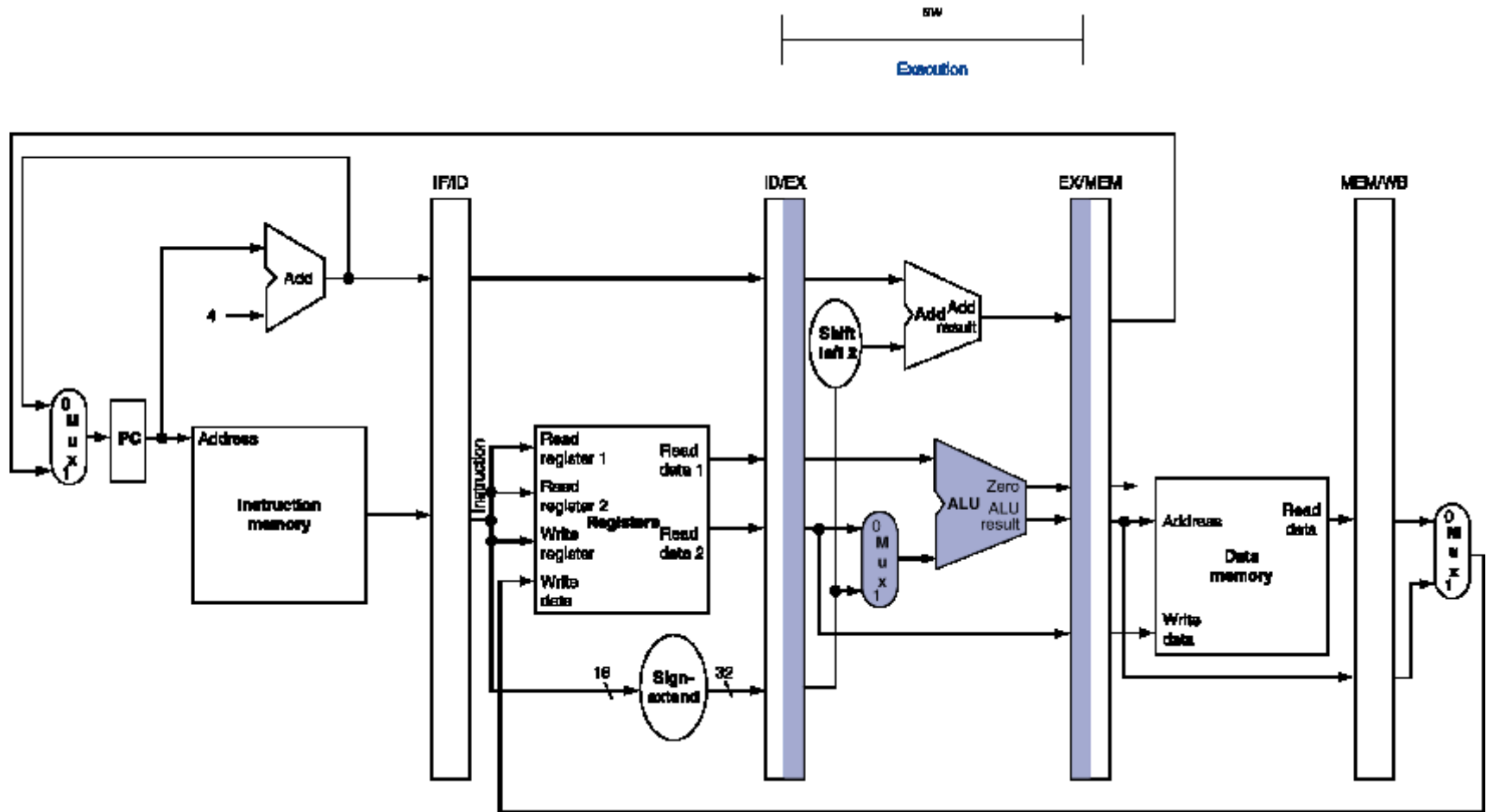




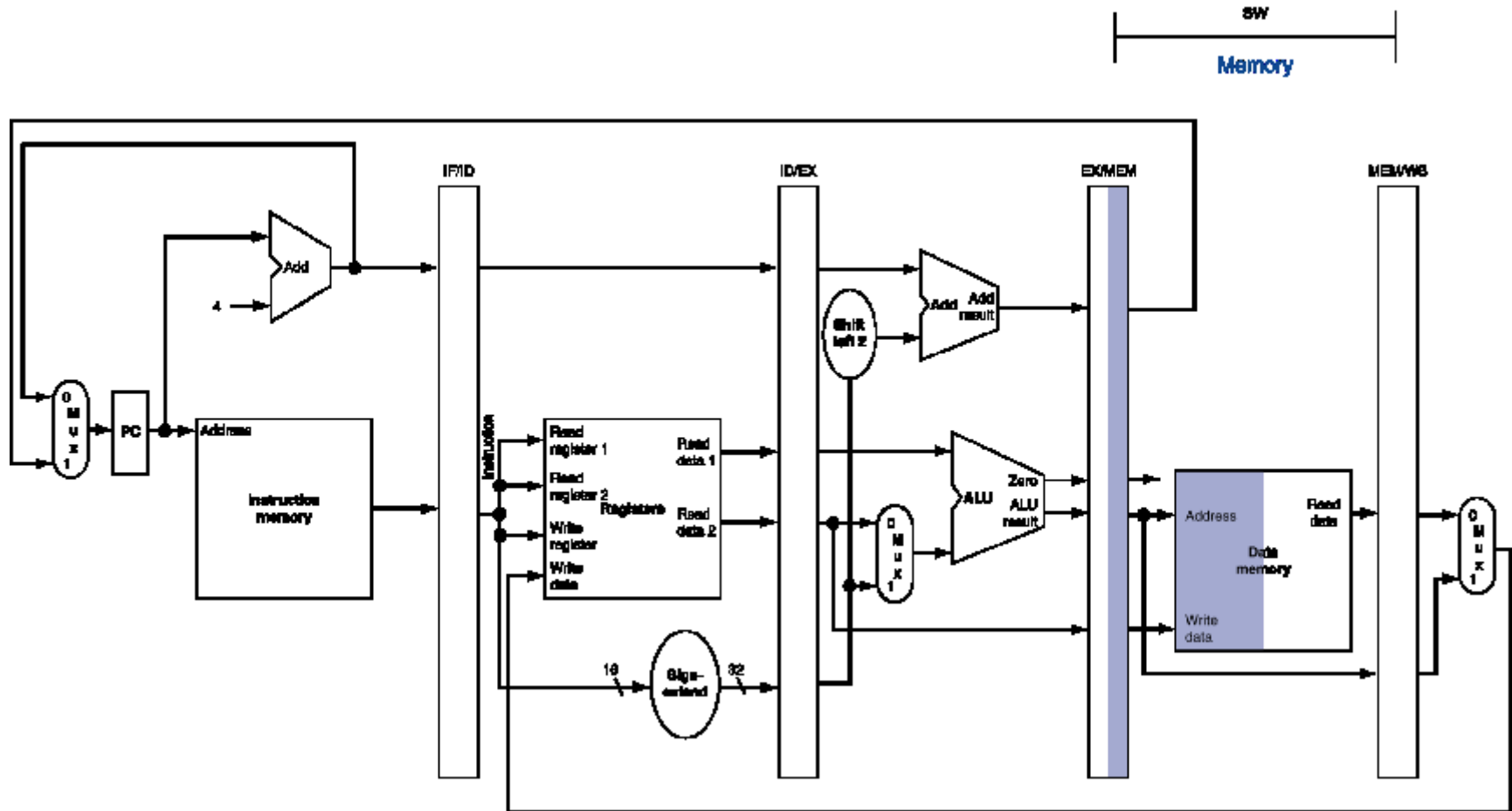
# Corrected Datapath for Load



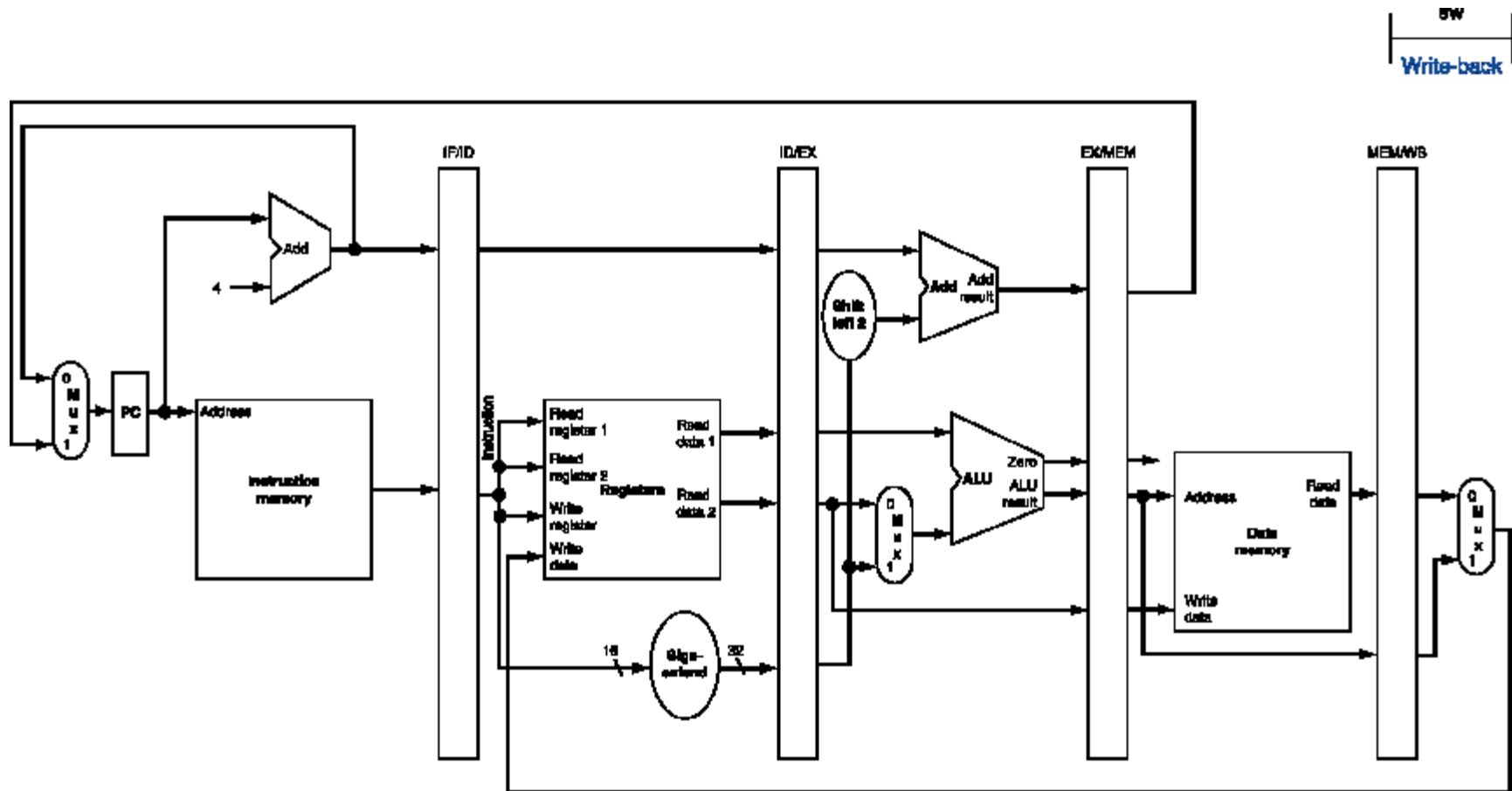
# EX for Store



# MEM for Store

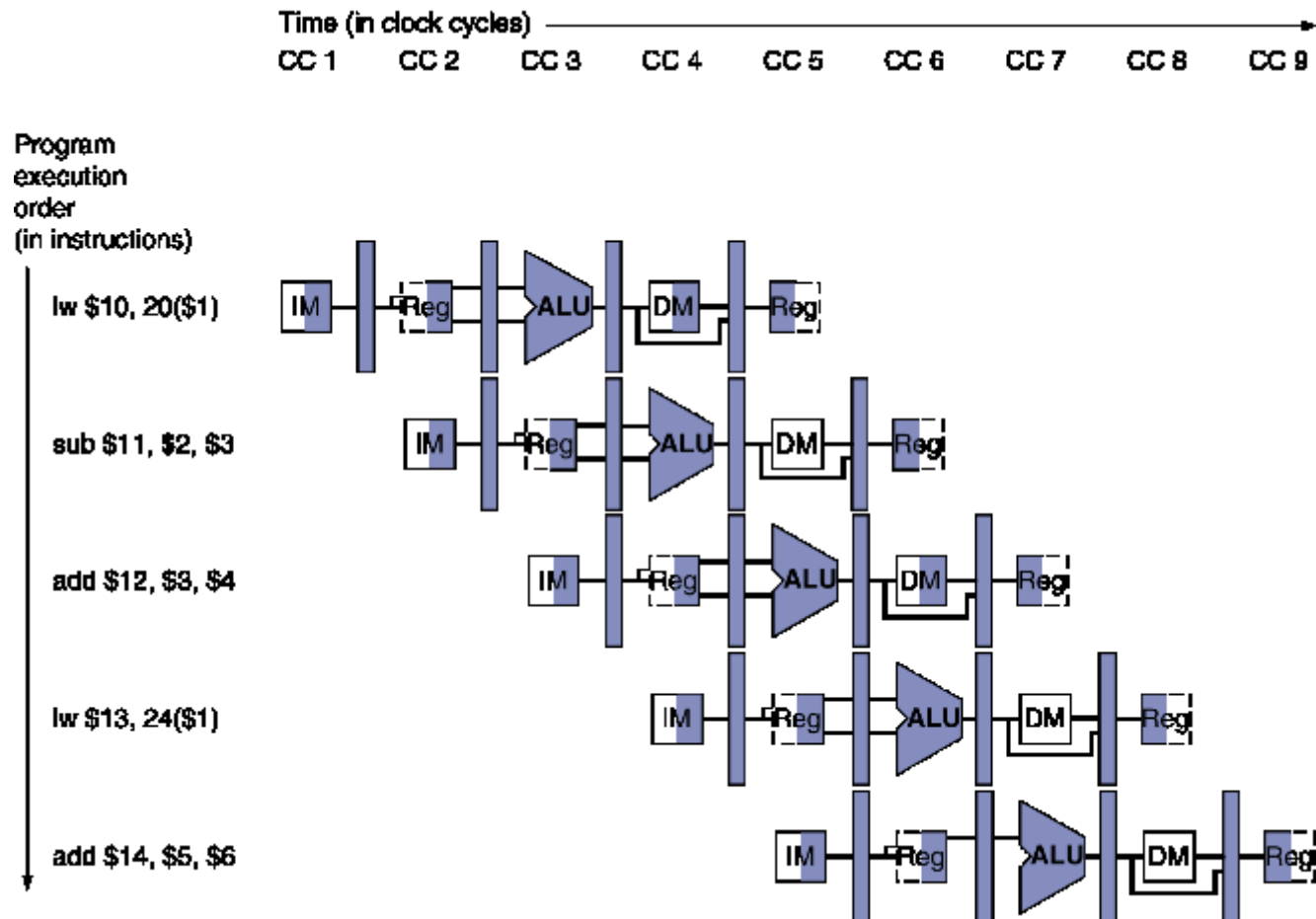


# WB for Store



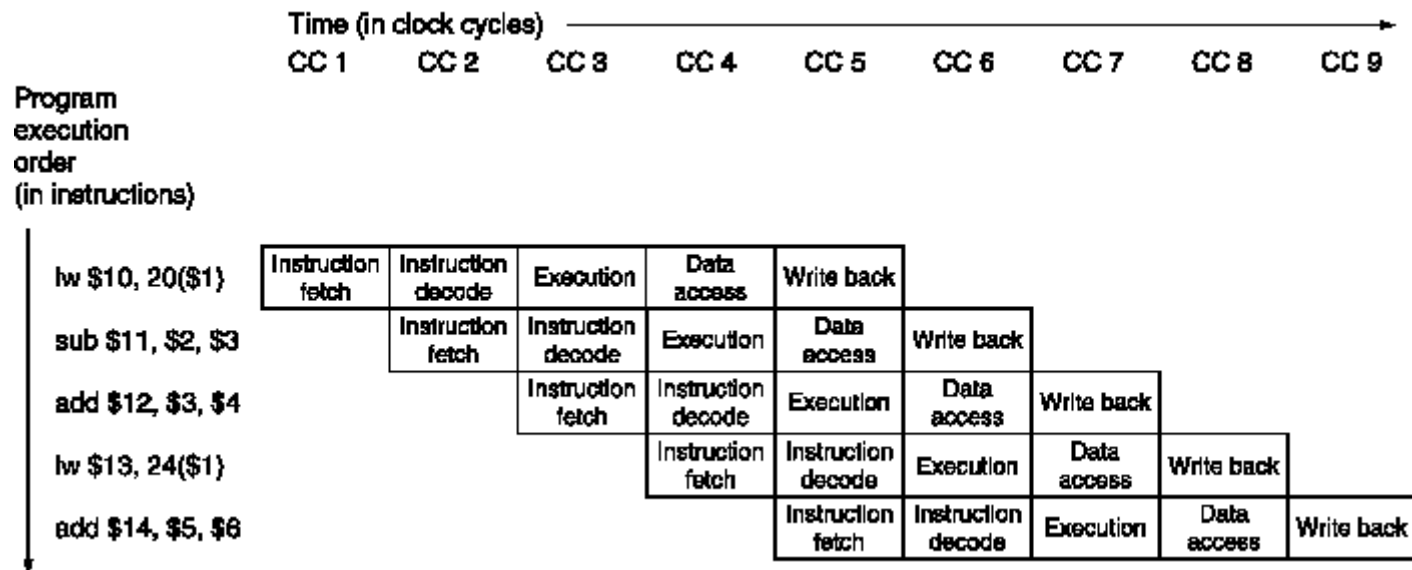
## Multi-Cycle Pipeline Diagram

- Form showing resource usage



# Multi-Cycle Pipeline Diagram

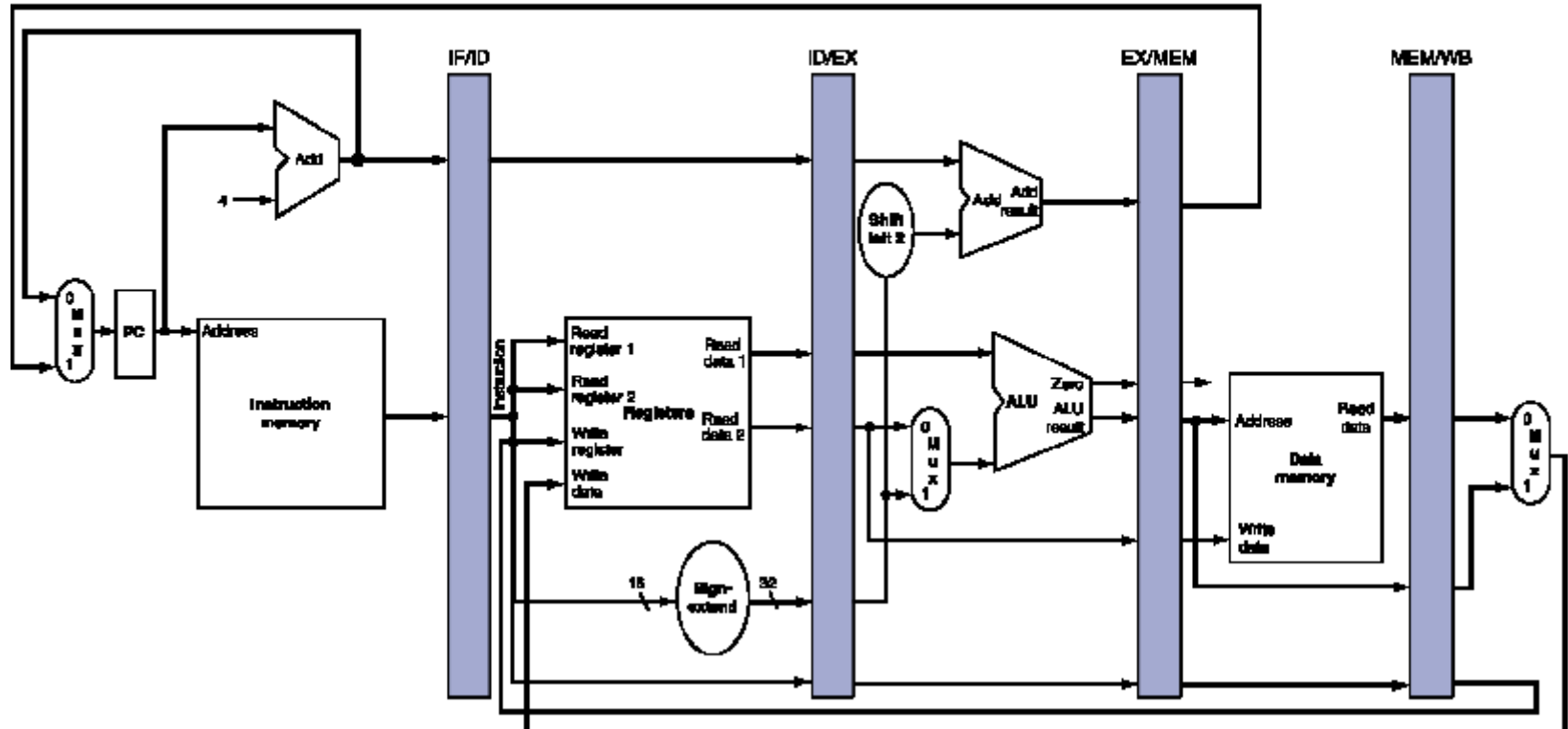
## ❑ Traditional form



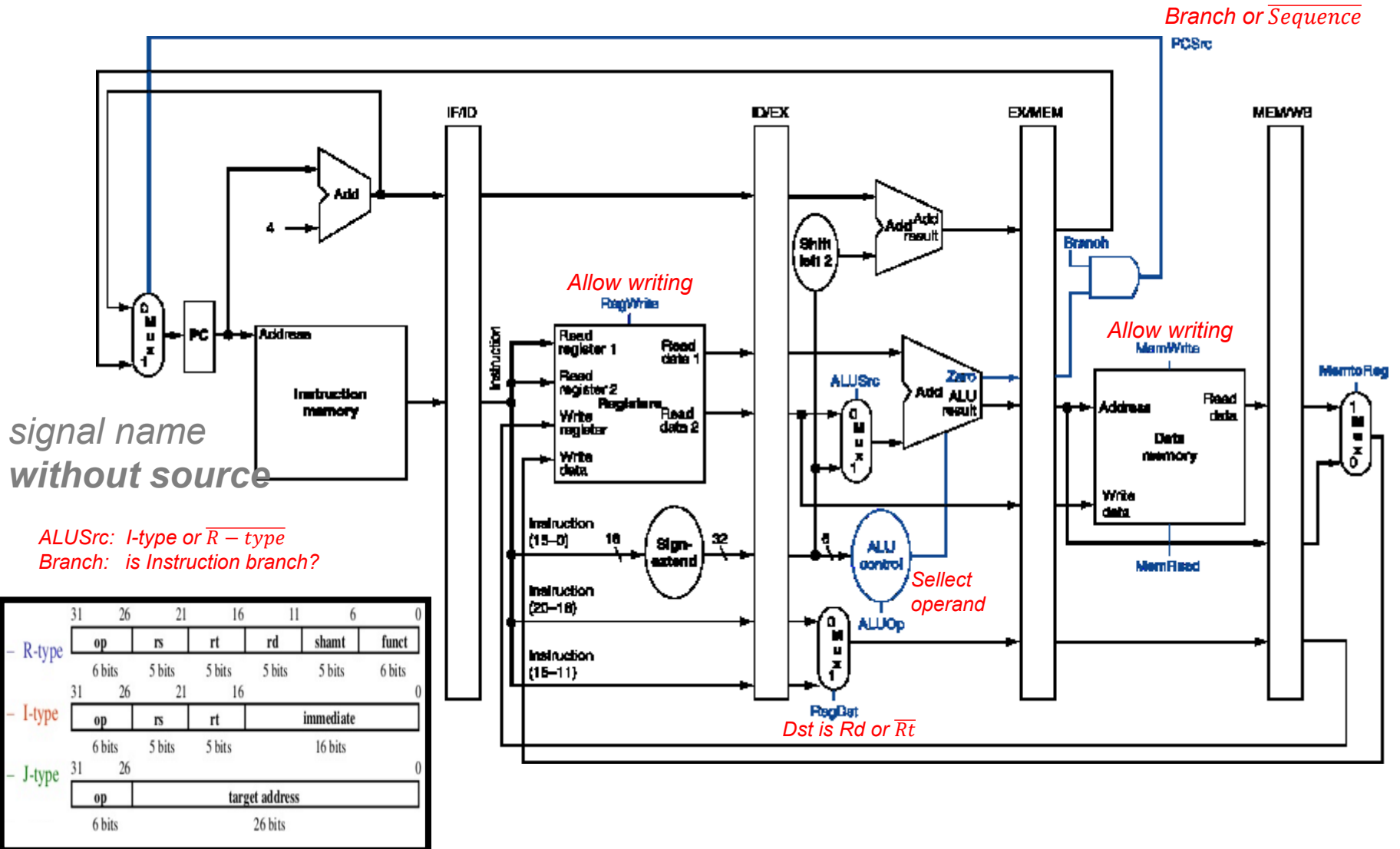
# Single-Cycle Pipeline Diagram

## ❑ State of pipeline in a given cycle

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



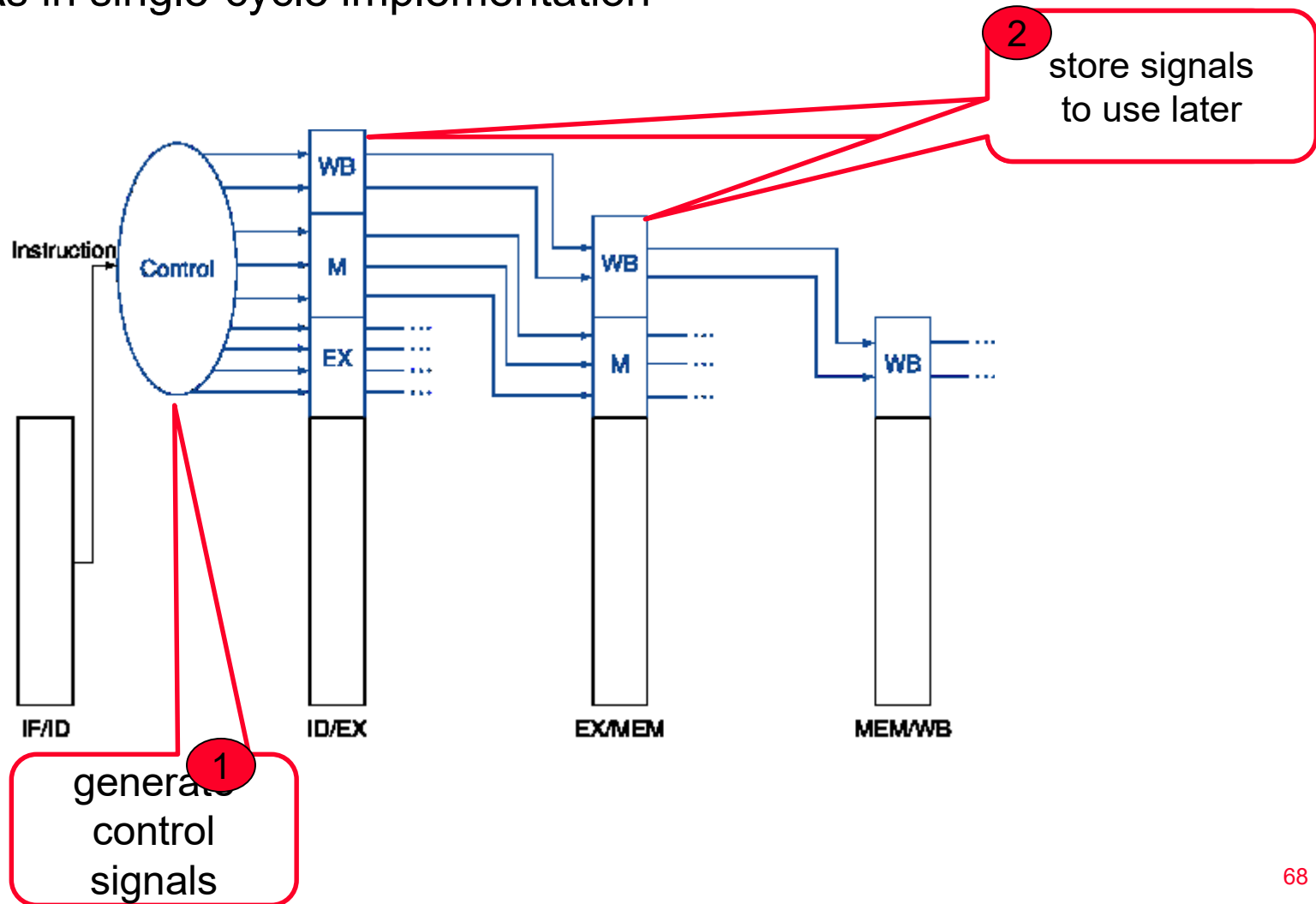
## Pipelined Control (Simplified)



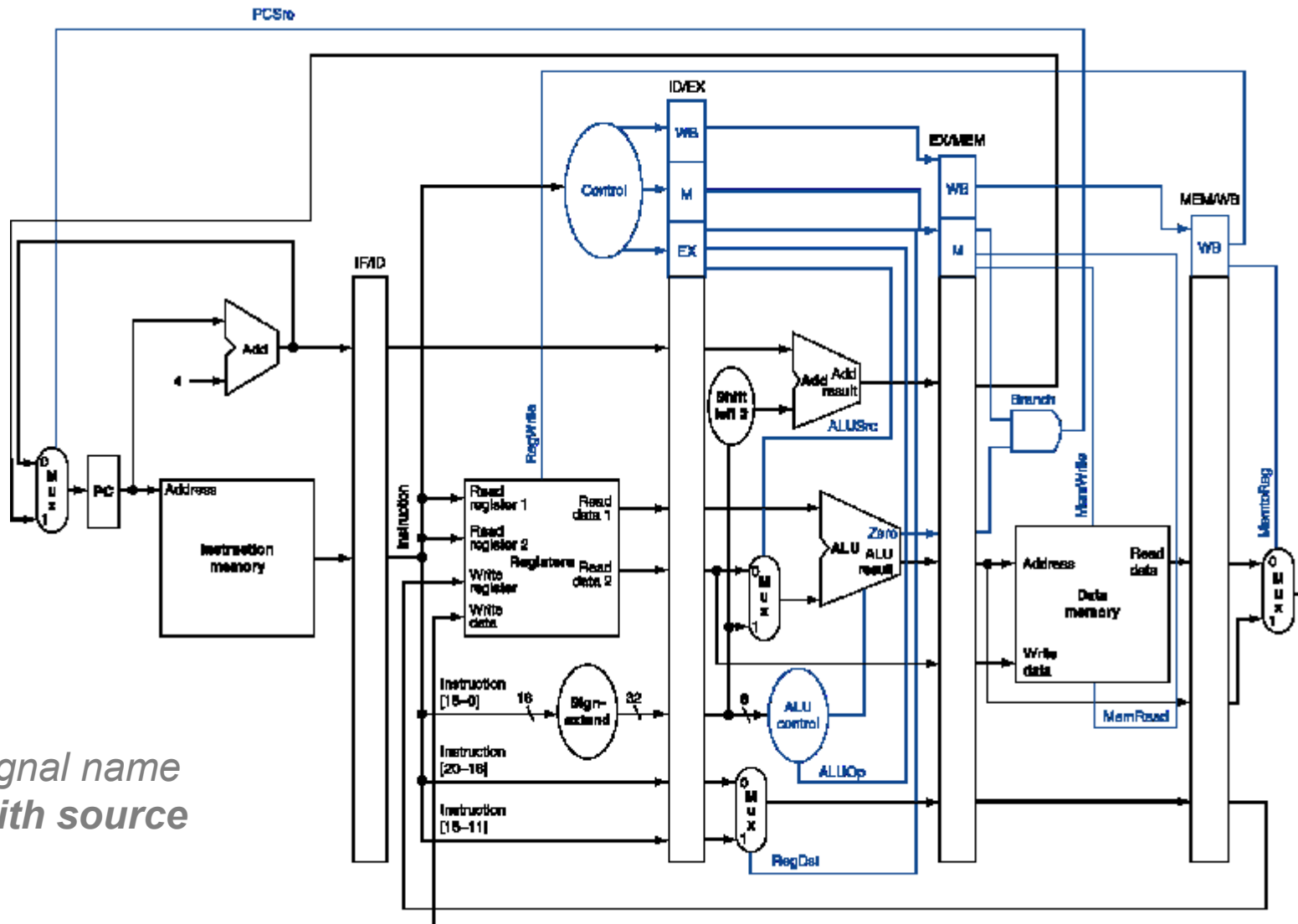


# Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



# Pipelined Control



*signal name  
with source*

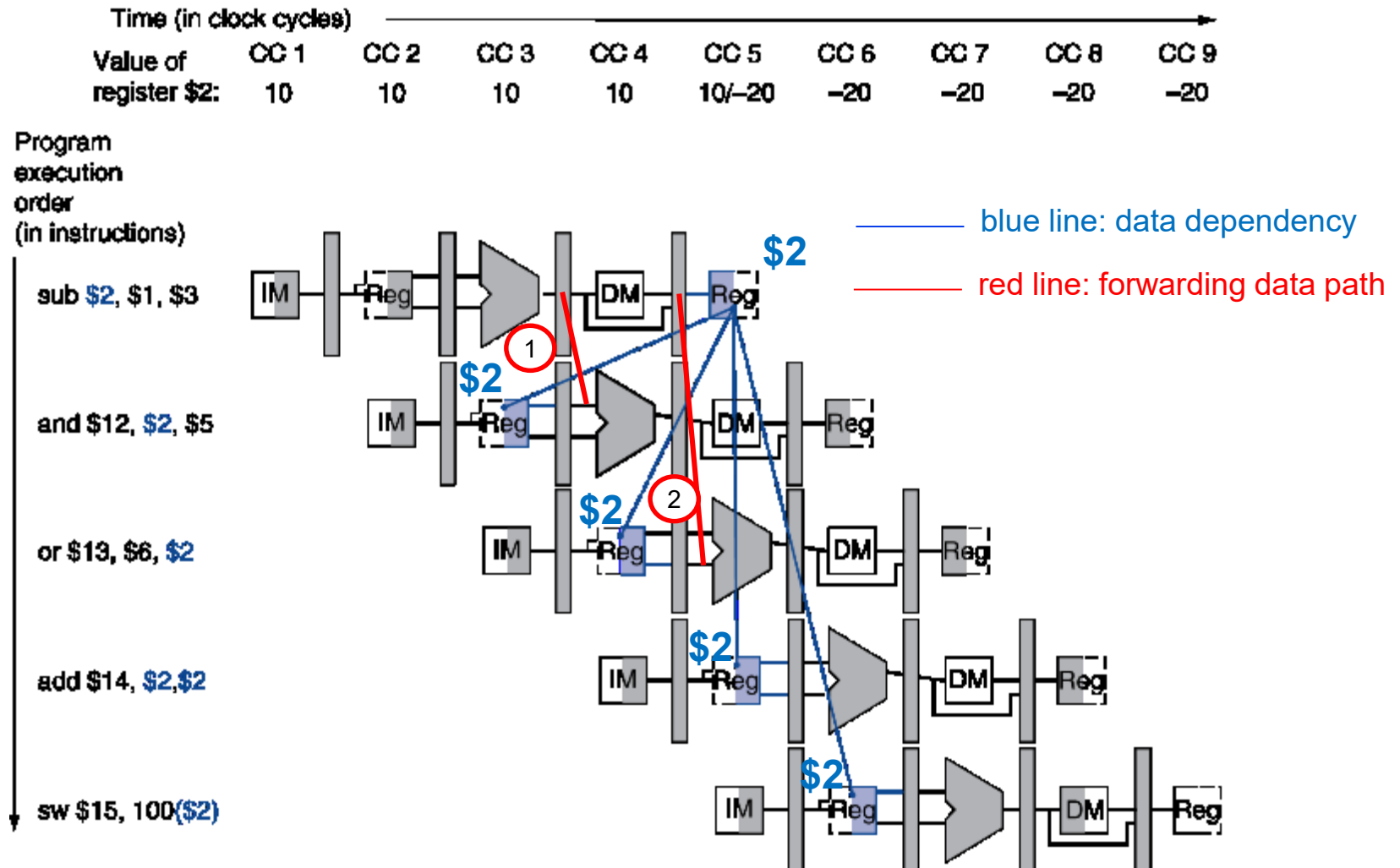
# Data Hazards in ALU Instructions

- ❑ Consider this sequence:

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

- ❑ We can resolve hazards with forwarding
  - | How do we detect when to forward?

# Dependencies & Forwarding



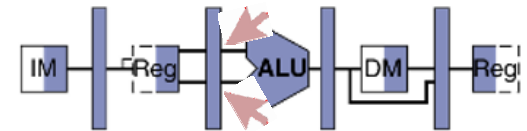
# Detecting the Need to Forward

## Pass register numbers along pipeline

- e.g., ID/EX.RegisterRs: means that, register number for Rs sitting in ID/EX pipeline register

## ALU operand register numbers in EX stage are given by

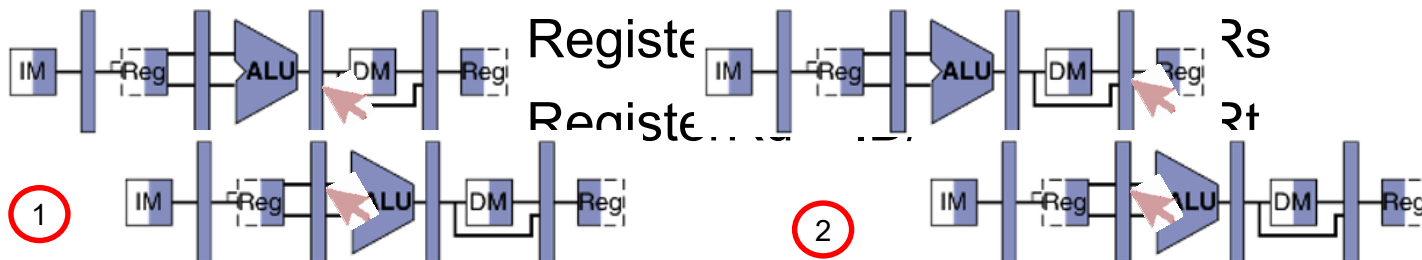
- ID/EX.RegisterRs, ID/EX.RegisterRt



## Data hazards when

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt



Fwd from  
EX/MEM  
pipeline reg

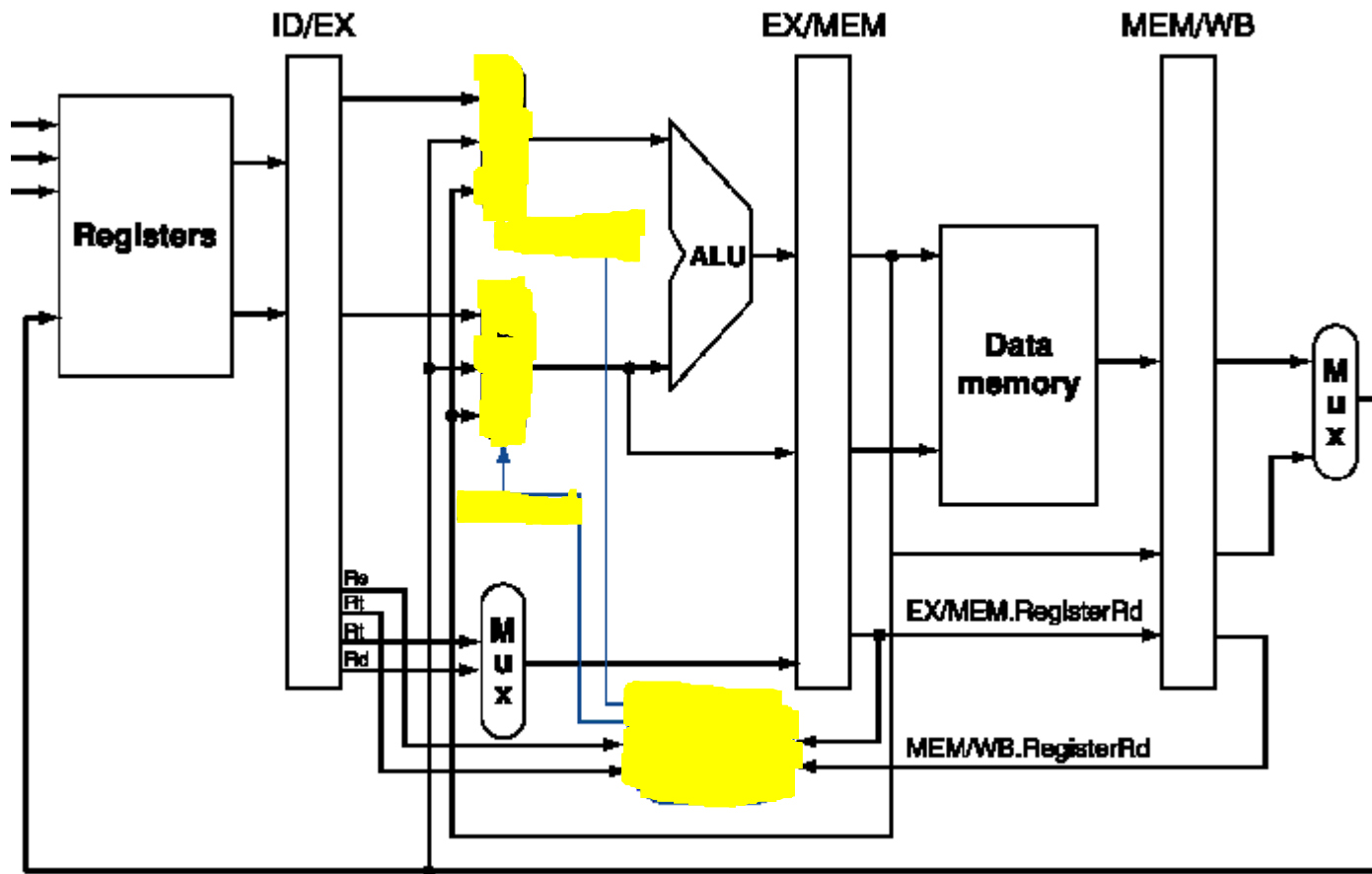
Fwd from  
MEM/WB  
pipeline reg

# Detecting the Need to Forward

---

- ❑ But only if forwarding instruction will write **to a register!** (if register has no change, no need to forward)
  - | EX/MEM.RegWrite, MEM/WB.RegWrite
  - | e.g: `sw $3, 0($2)`      # \$3 not changed
  - `add $1, $3, $3`      # no need
- ❑ And only if Rd for that instruction is **not \$zero** (\$zero is readonly)
  - | EX/MEM.RegisterRd  $\neq$  0,  
MEM/WB.RegisterRd  $\neq$  0

# Forwarding Paths



b. With forwarding

# Forwarding Conditions

---

## ❑ EX hazard

- | if (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))

ForwardA = 10

- | if (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))

ForwardB = 10

## ❑ MEM hazard

- | if (MEM/WB.RegWrite and (MEM/WB.RegisterRd  $\neq$  0)  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

ForwardA = 01

- | if (MEM/WB.RegWrite and (MEM/WB.RegisterRd  $\neq$  0)  
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

ForwardB = 01



# Double Data Hazard

---

- ❑ Consider the sequence:

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

- ❑ Both hazards occur
  - | Want to use the most recent
- ❑ Revise MEM hazard condition
  - | Only fwd if EX hazard condition isn't true

# Revised Forwarding Condition

## ❑ MEM hazard

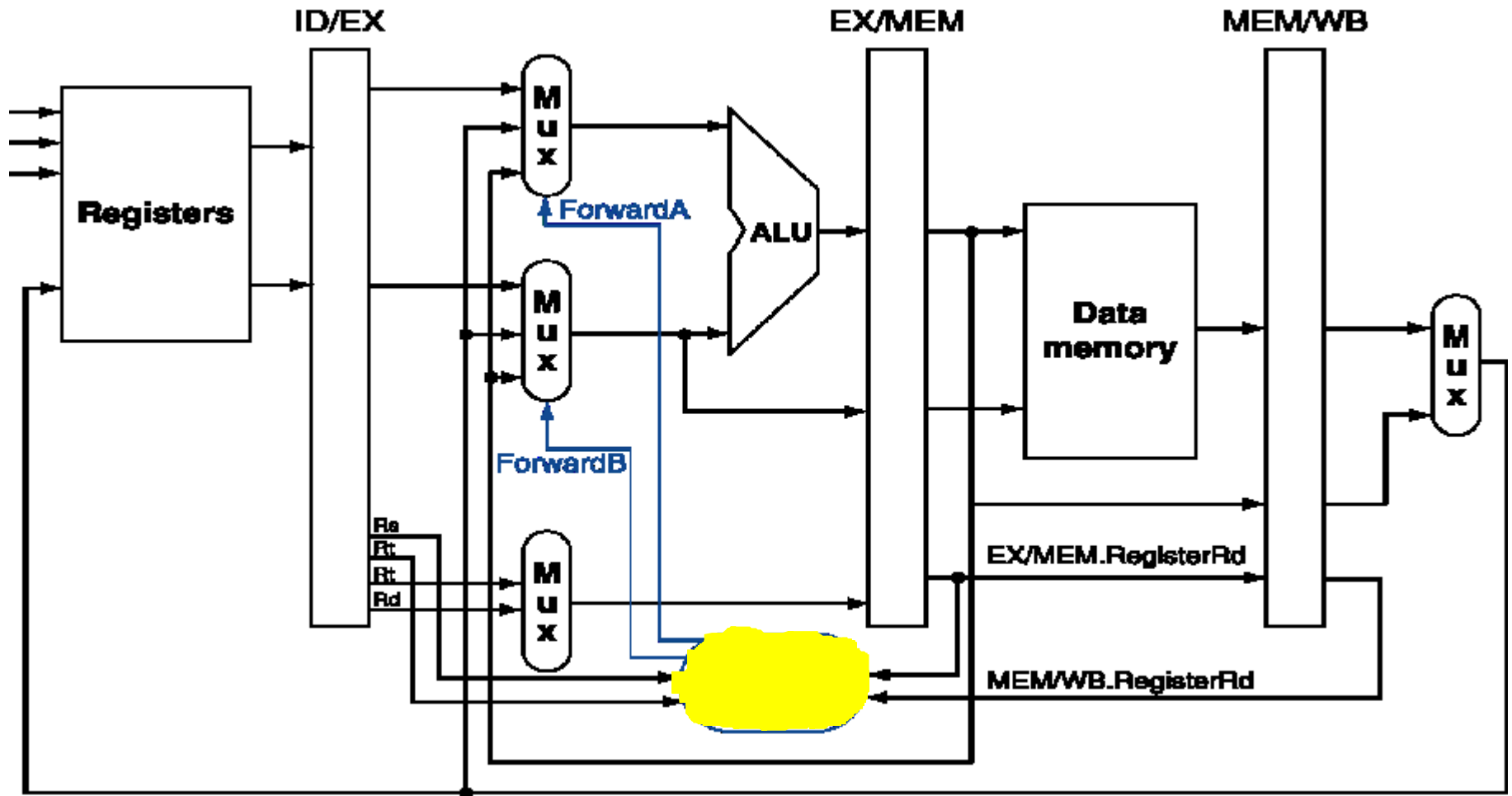
- | if (MEM/WB.RegWrite and (MEM/WB.RegisterRd  $\neq$  0)  
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

ForwardA = 01

- | if (MEM/WB.RegWrite and (MEM/WB.RegisterRd  $\neq$  0)  
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

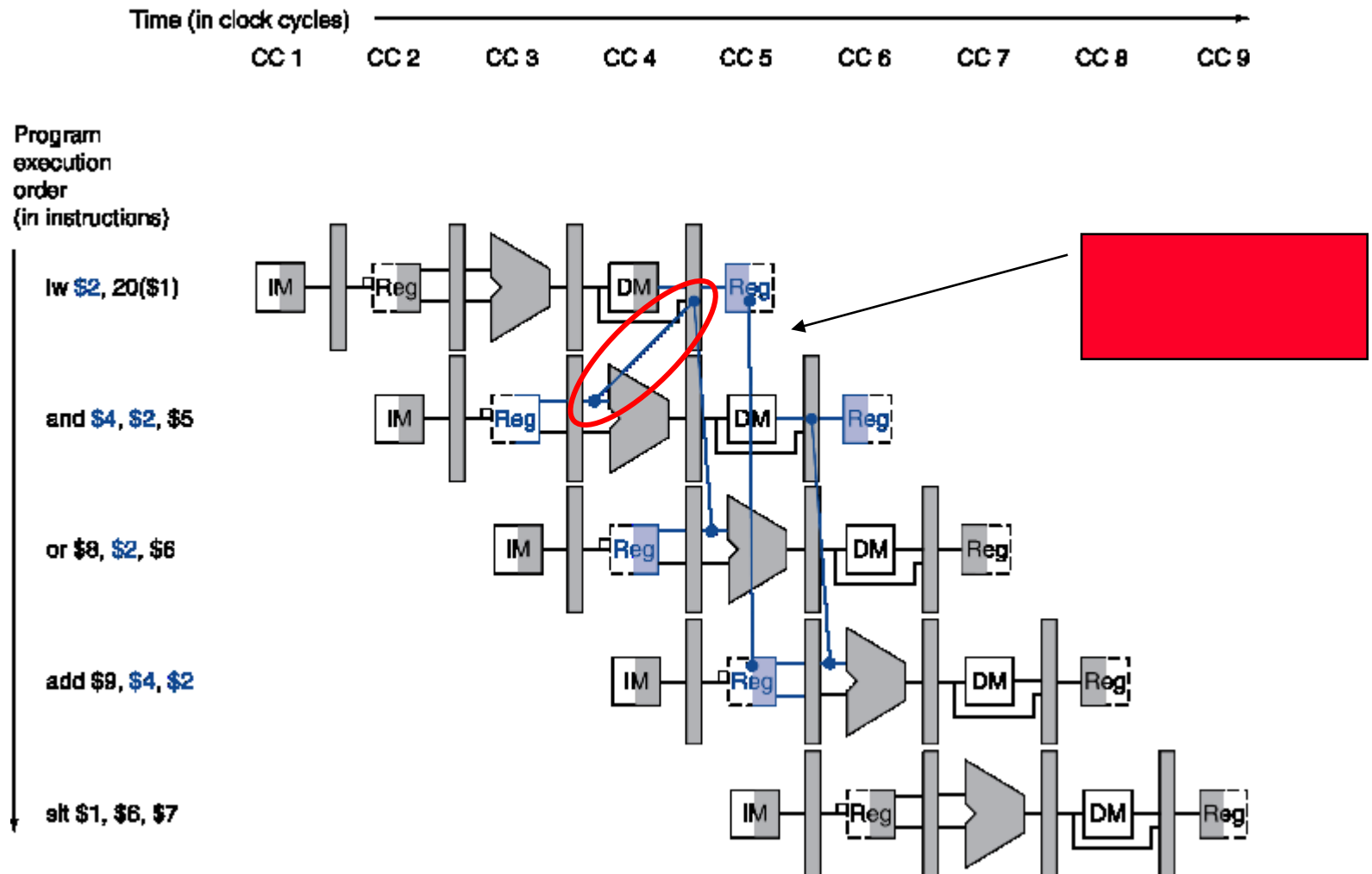
ForwardB = 01

# Datapath with Forwarding



b. With forwarding


# Load-Use Data Hazard



## Load-Use Hazard Detection

---

- ❑ Check when using instruction is decoded in ID stage
- ❑ ALU operand register numbers in ID stage are given by
  - | IF/ID.RegisterRs, IF/ID.RegisterRt
- ❑ Load-use hazard when
  - | ID/EX.MemRead and  
((ID/EX.RegisterRt = IF/ID.RegisterRs) or  
(ID/EX.RegisterRt = IF/ID.RegisterRt))
- ❑ If detected, stall and insert bubble



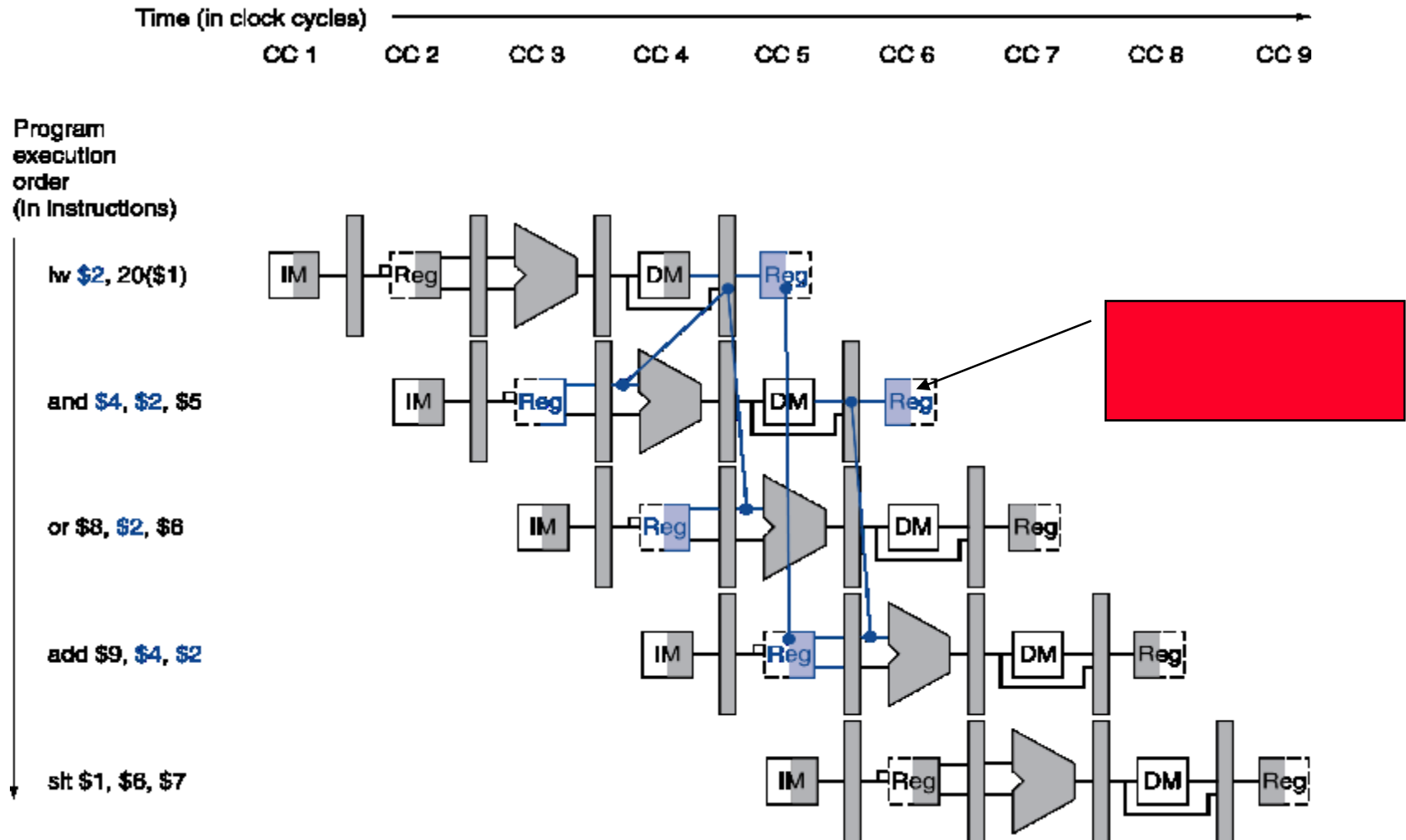
Register Rt contains  
the value from Mem

## How to Stall the Pipeline

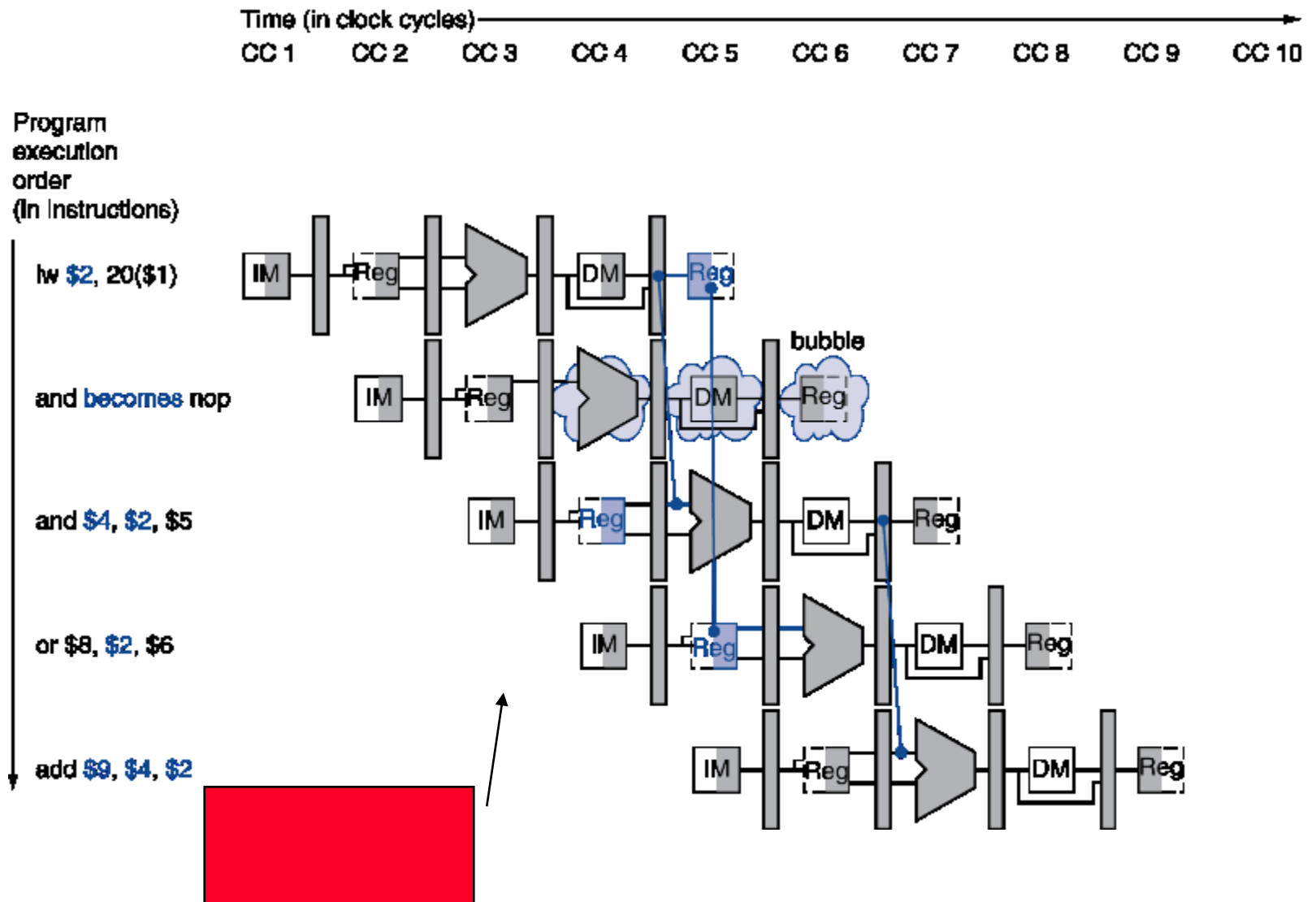
---

- ❑ Force control values in ID/EX register to 0
  - | EX, MEM and WB do nop (no-operation)
- ❑ Prevent update of PC and IF/ID register
  - | Using instruction is decoded again
  - | Following instruction is fetched again
  - | 1-cycle stall allows MEM to read data for 1w
    - Can subsequently forward to EX stage

# Stall/Bubble in the Pipeline

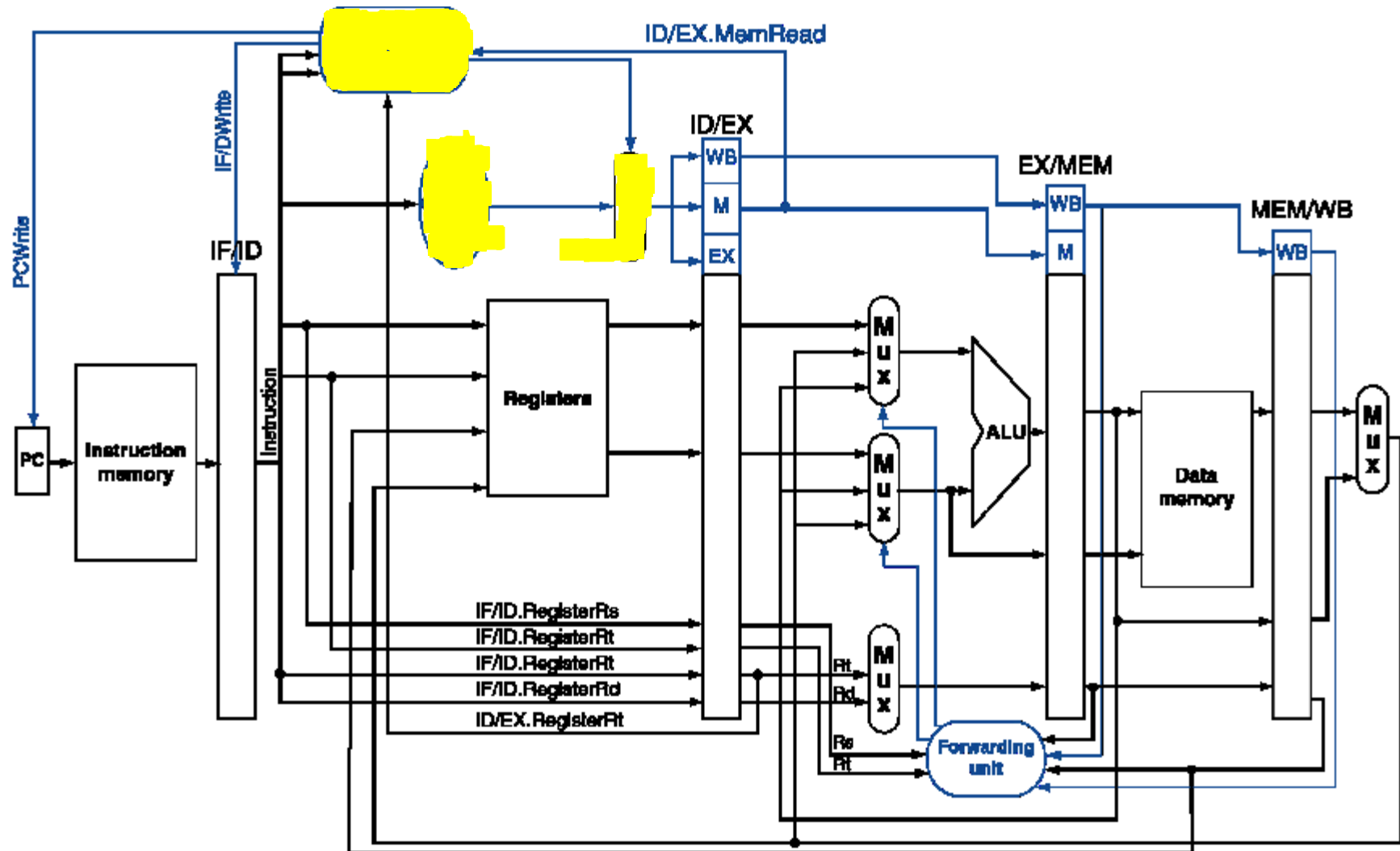


# Stall/Bubble in the Pipeline





# Datapath with Hazard Detection



# Stalls and Performance

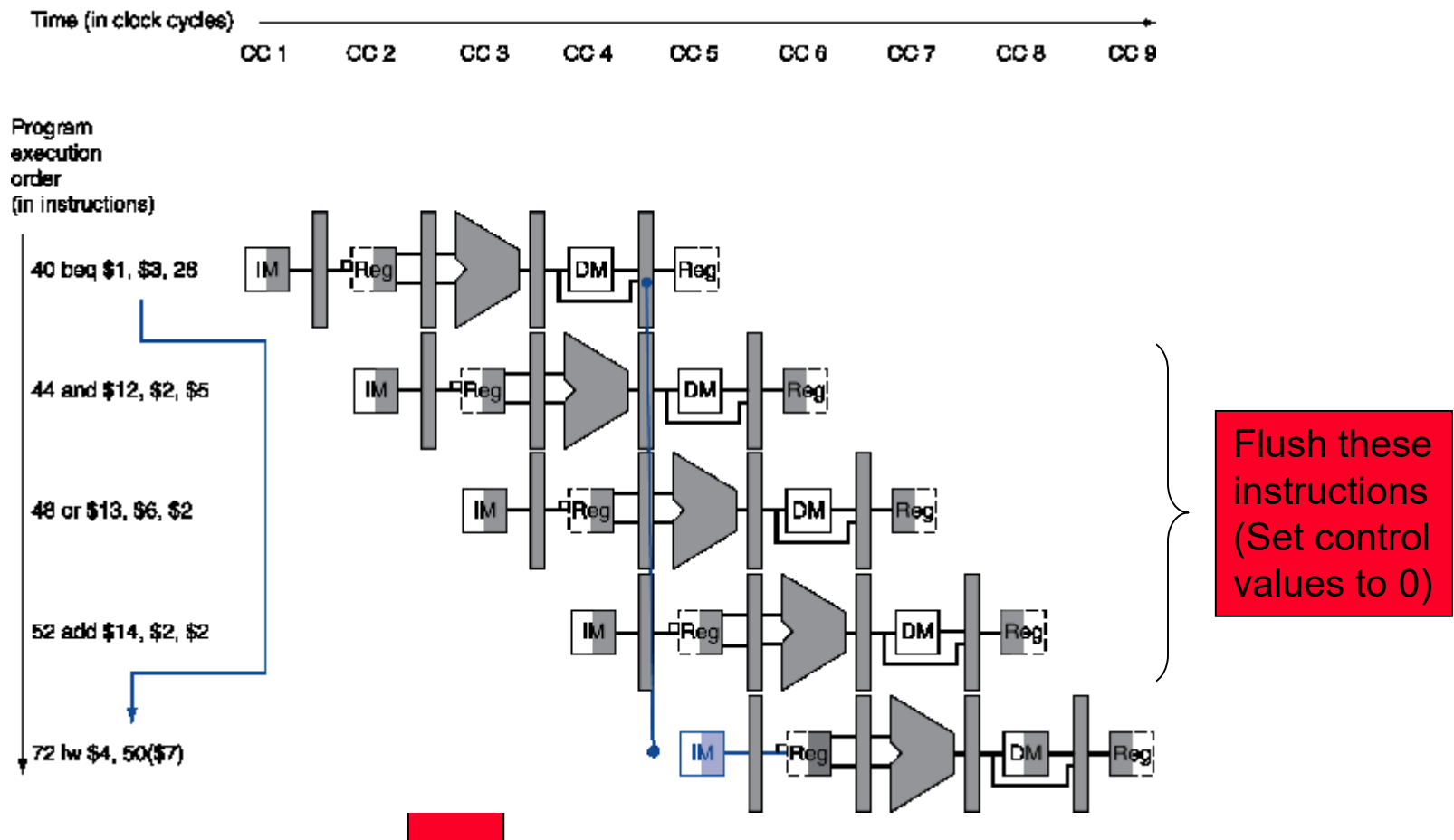
---

## The BIG Picture

- ❑ Stalls reduce performance
  - | But are required to get correct results
- ❑ Compiler can arrange code to avoid hazards and stalls
  - | Requires knowledge of the pipeline structure

# Branch Hazards

- If branch outcome determined in MEM



## Reducing Branch Delay

---

### ❑ Move hardware to determine outcome to ID stage

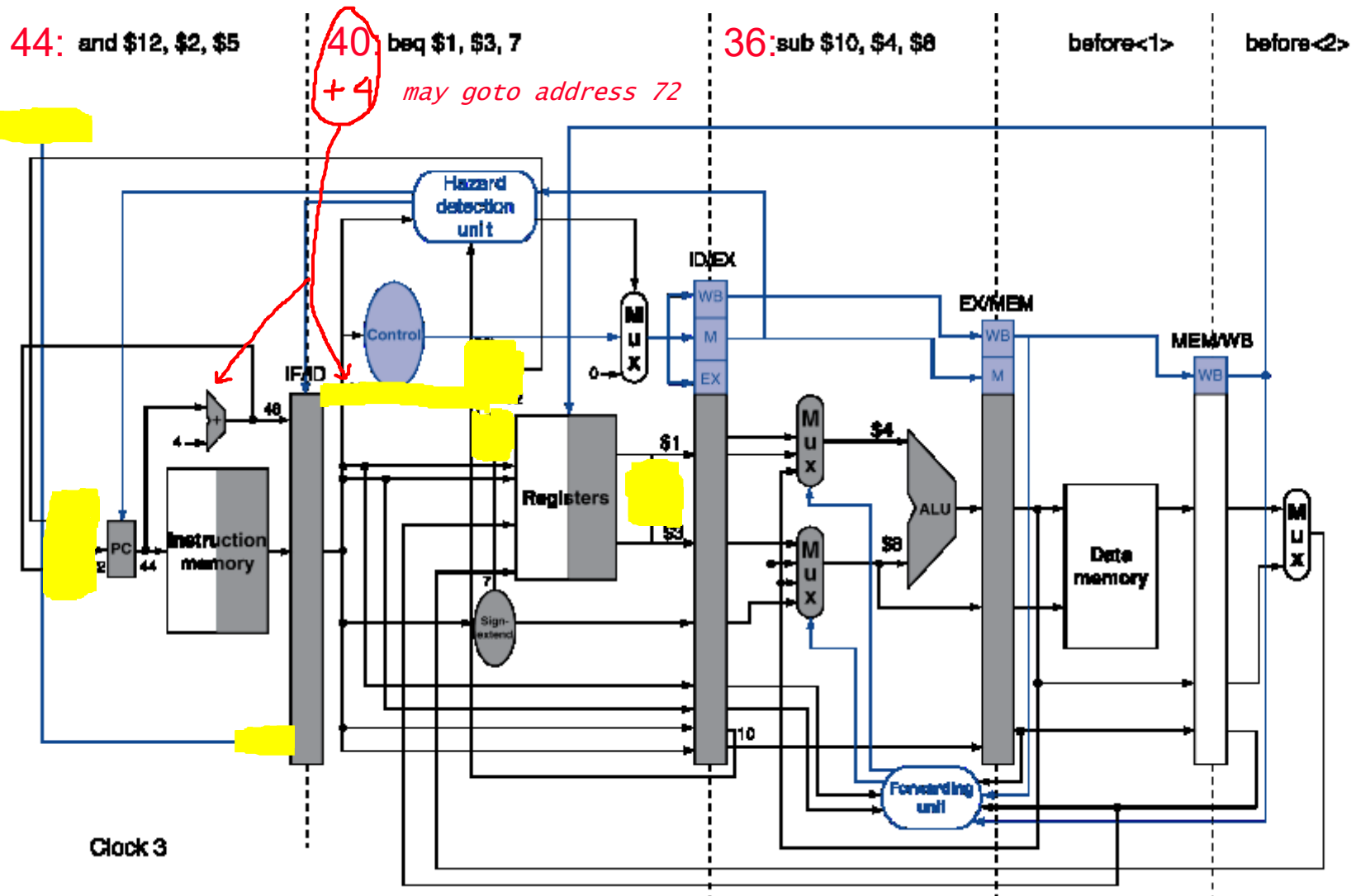
- | Target address adder (branch addr = current addr + offset)
- | Register comparator

### ❑ Example: branch taken

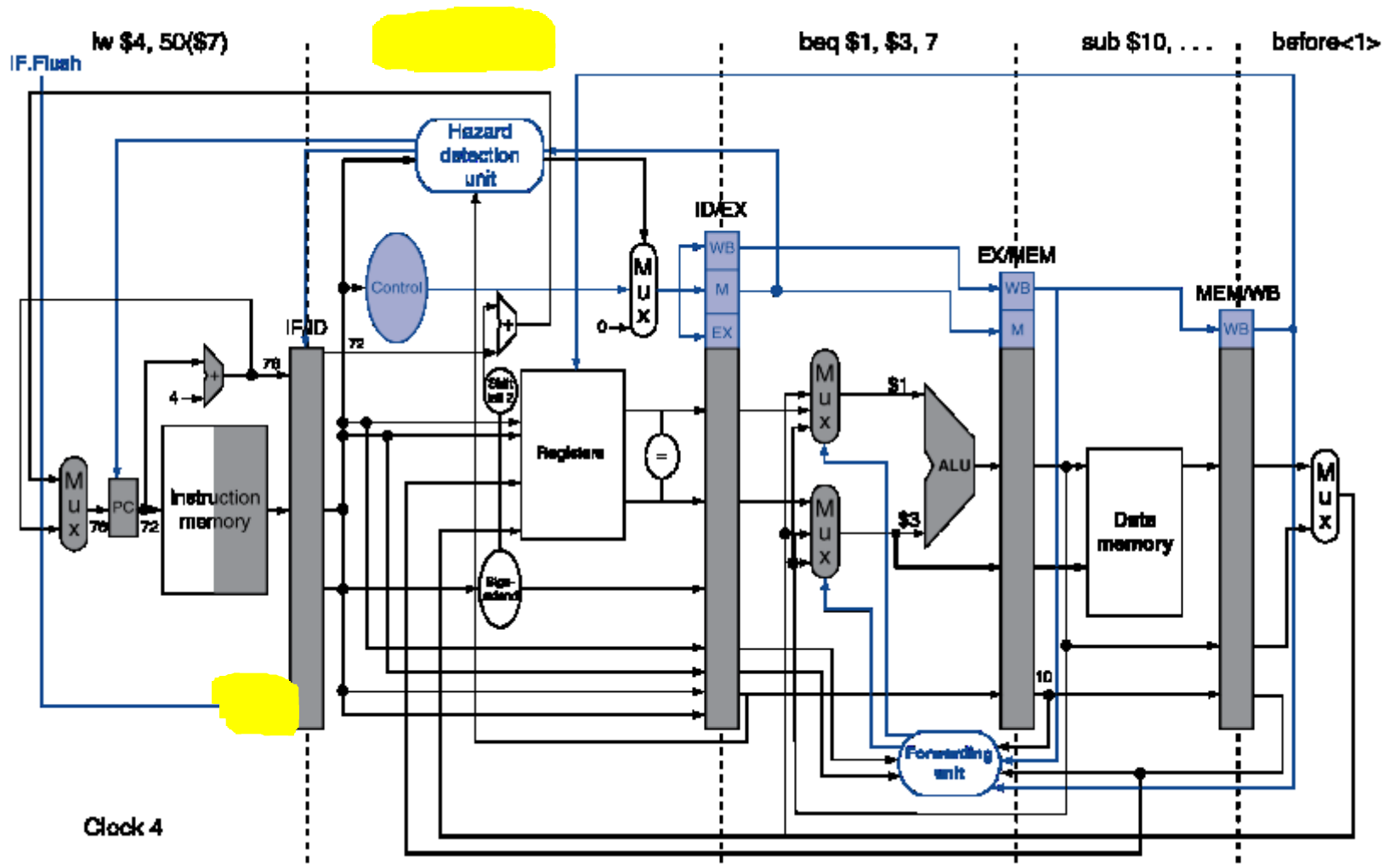
```
36:  sub    $10, $4, $8
40:  beq    $1,  $3, 7
44:  and    $12, $2, $5
48:  or     $13, $2, $6
52:  add    $14, $4, $2
56:  slt    $15, $6, $7

72:  .w     $4, 50($7)
```

# Example: Branch Taken

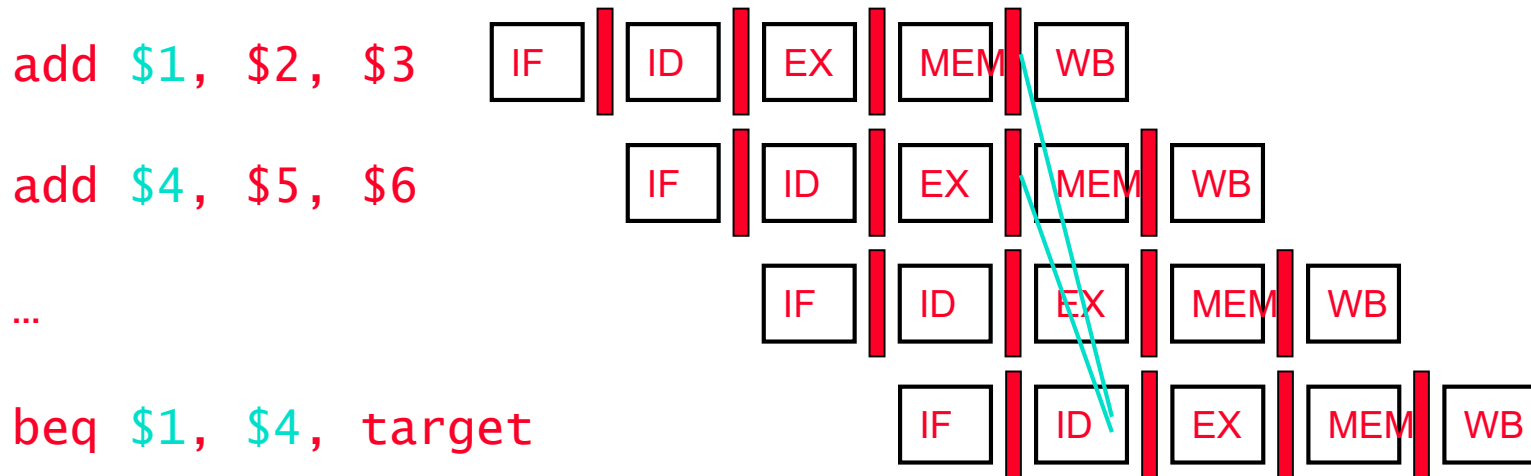


# Example: Branch Taken



# Data Hazards for Branches

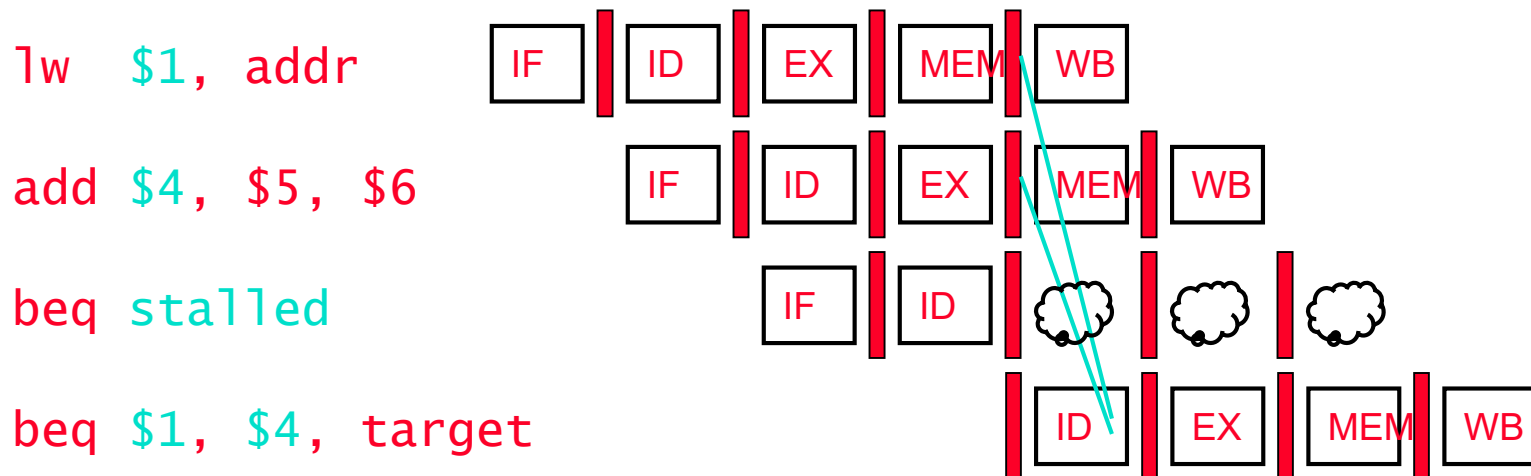
- ❑ If a comparison register is a destination of 2<sup>nd</sup> or 3<sup>rd</sup> preceding ALU instruction



- Can resolve using forwarding

# Data Hazards for Branches

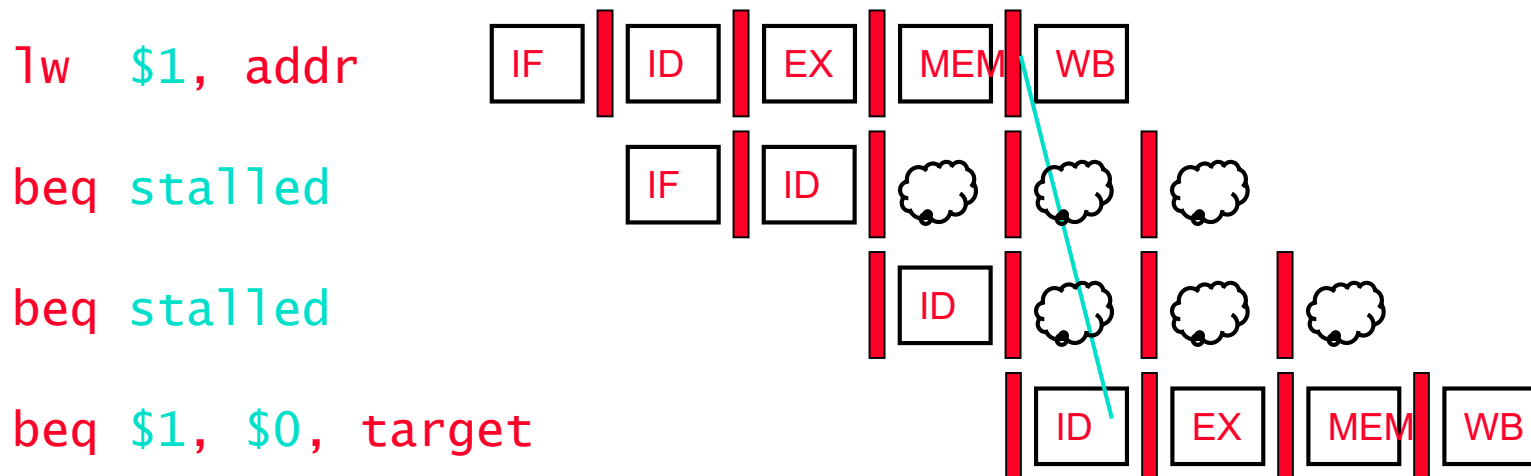
- ❑ If a comparison register is a destination of preceding ALU instruction or 2<sup>nd</sup> preceding load instruction
  - | Need 1 stall cycle





# Data Hazards for Branches

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



## Dynamic Branch Prediction

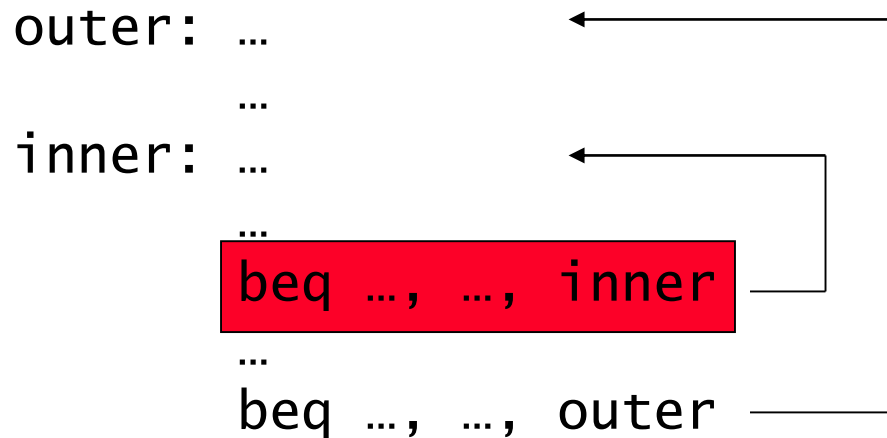
---

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

# 1-Bit Predictor: Shortcoming

---

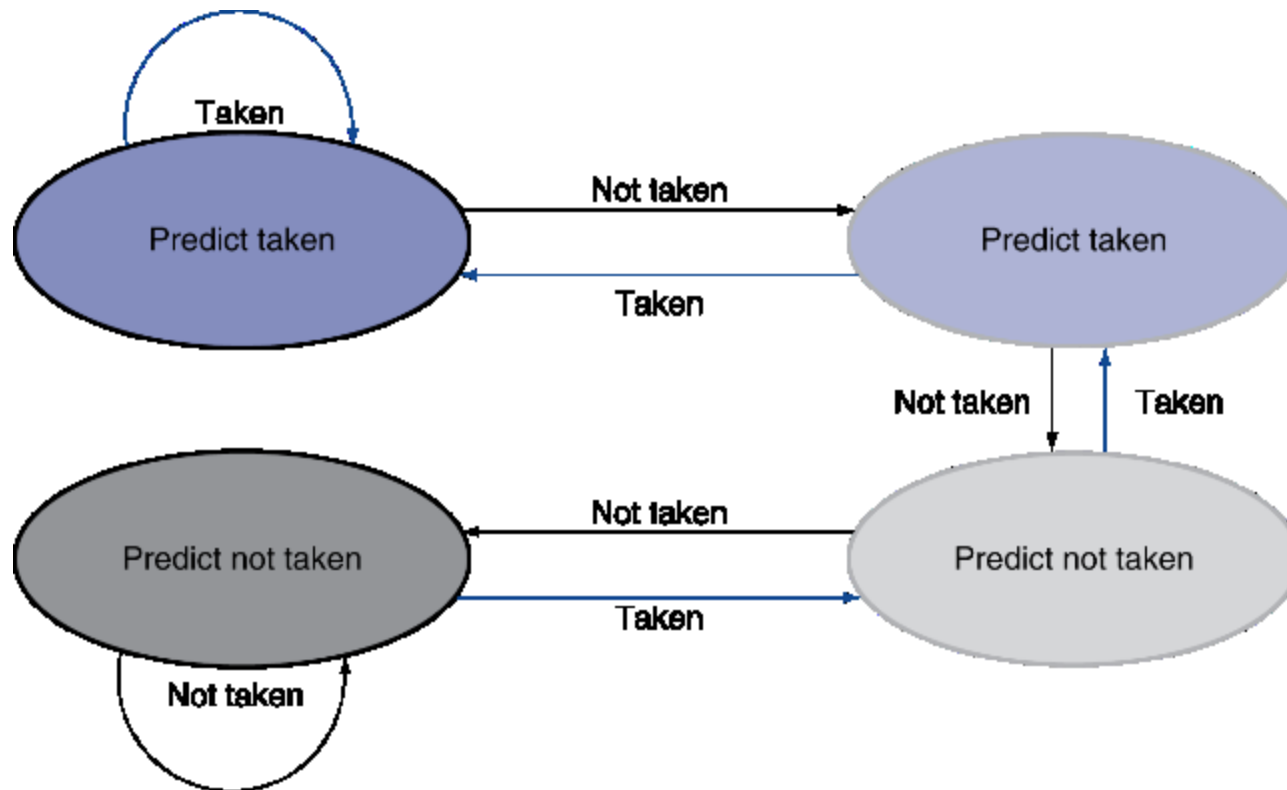
- ❑ Inner loop branches mispredicted twice!



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

## 2-Bit Predictor

- Only change prediction on two successive mispredictions



# Calculating the Branch Target

---

- ❑ Even with predictor, still need to calculate the target address
  - | 1-cycle penalty for a taken branch
- ❑ Branch target buffer
  - | Cache of target addresses
  - | Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

## Exceptions and Interrupts

---

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

## Handling Exceptions

---

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

# An Alternate Mechanism

---

## ❑ Vectored Interrupts

- | Handler address determined by the cause

## ❑ Example:

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

## ❑ Instructions either

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



# Handler Actions

---

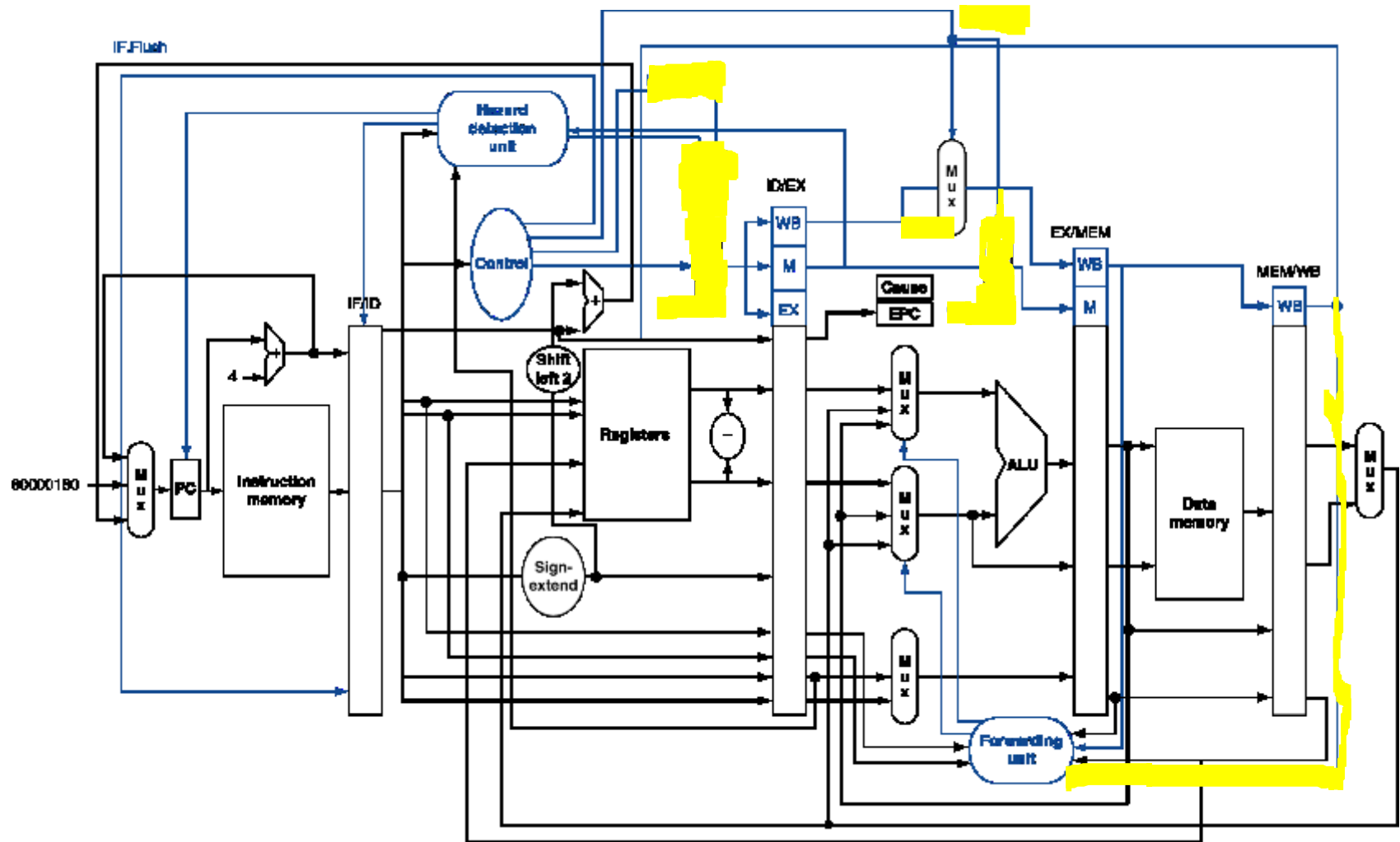
- ❑ Read cause, and transfer to relevant handler
- ❑ Determine action required
- ❑ If restartable
  - | Take corrective action
  - | use EPC to return to program
- ❑ Otherwise
  - | Terminate program
  - | Report error using EPC, cause, ...

# Exceptions in a Pipeline

---

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

# Pipeline with Exceptions



# Exception Properties

---

## ❑ Restartable exceptions

- | Pipeline can flush the instruction
- | Handler executes, then returns to the instruction
  - Refetched and executed from scratch

## ❑ PC saved in EPC register

- | Identifies causing instruction
- | Actually PC + 4 is saved
  - Handler must adjust

## Exception Example

---

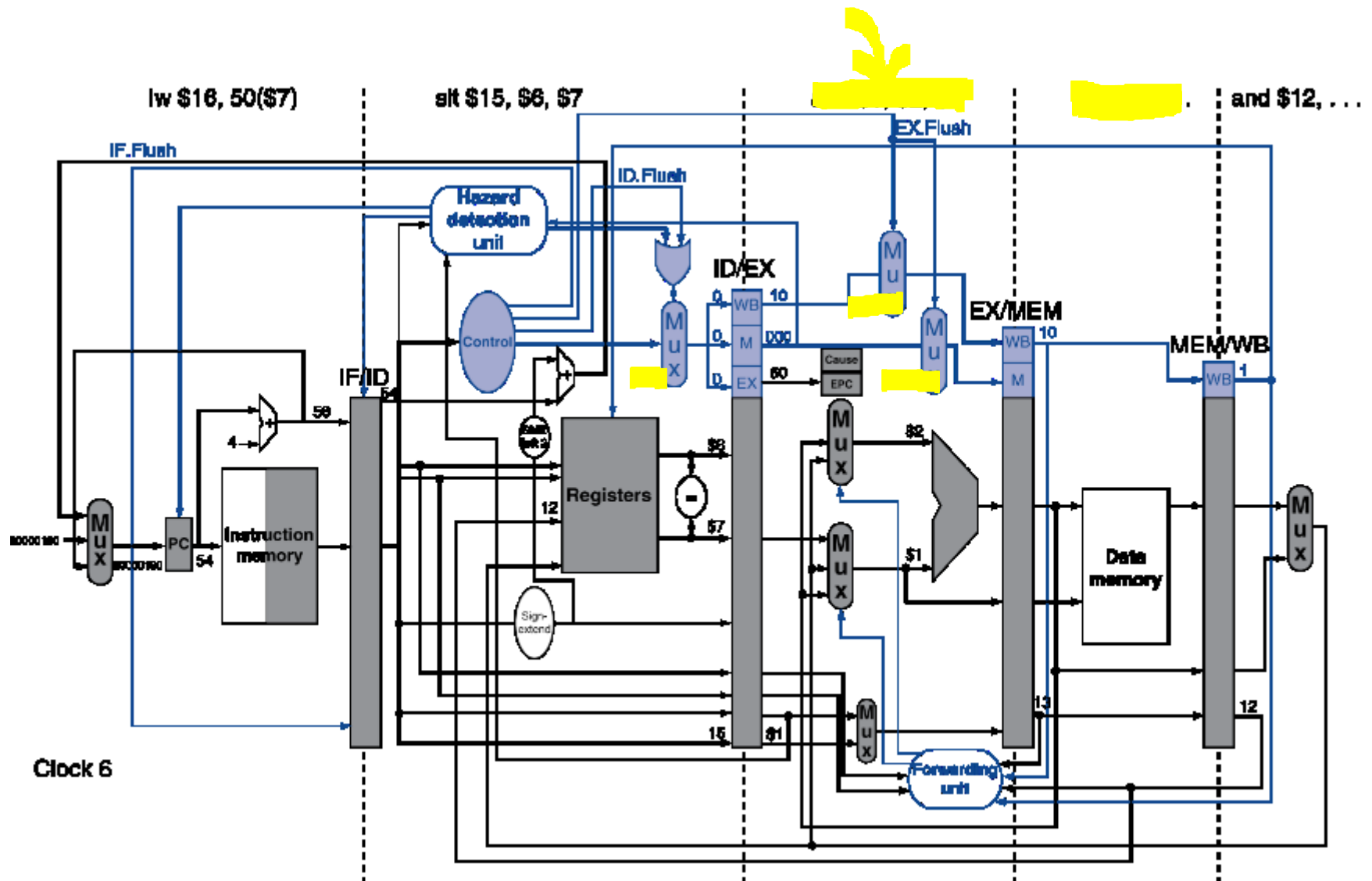
### ❑ Exception on **add** in

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

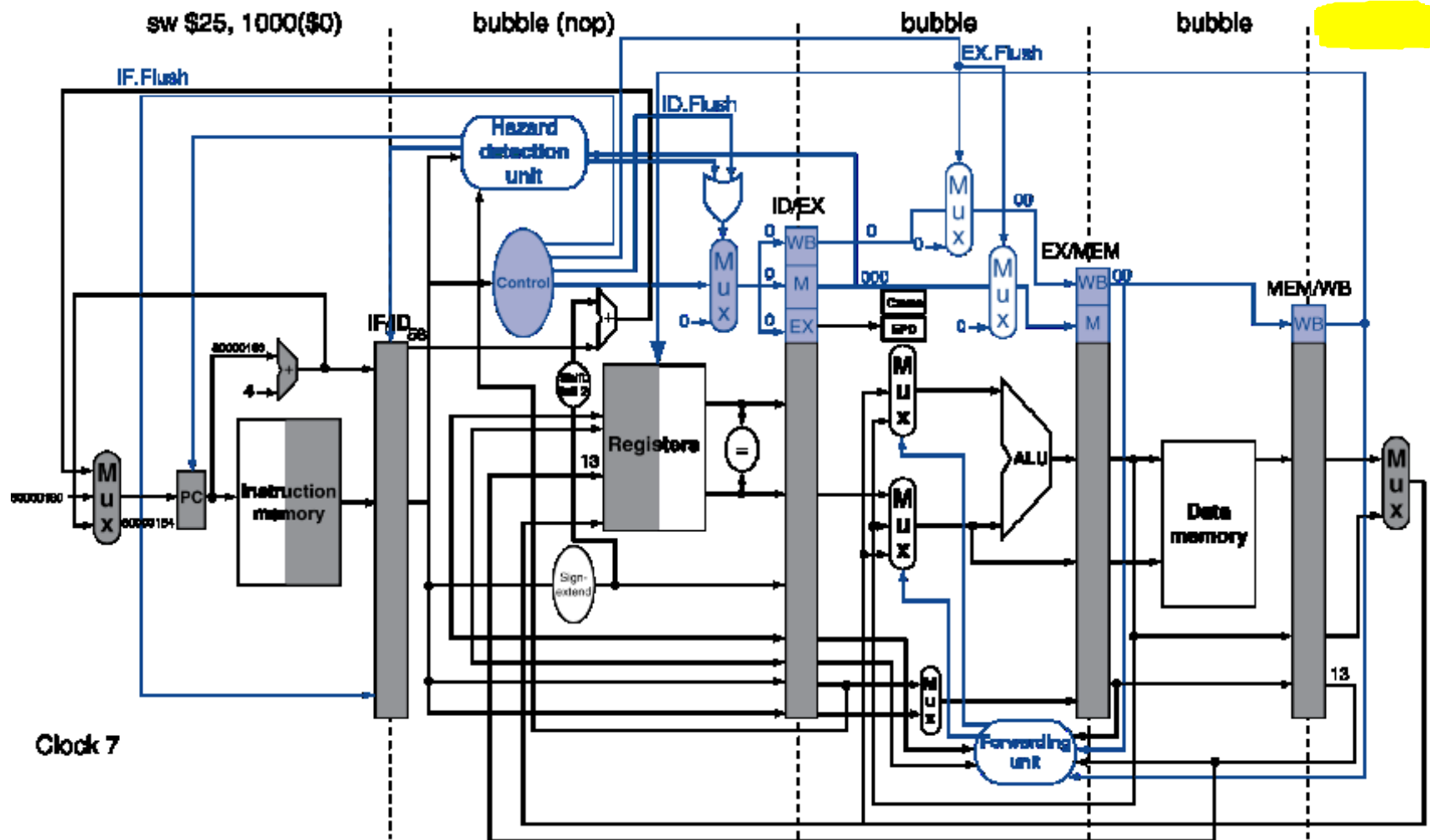
### ❑ Handler

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

# Exception Example



# Exception Example



## Multiple Exceptions

---

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



## Imprecise Exceptions

---

- ❑ Just stop pipeline and save state
  - | Including exception cause(s)
- ❑ Let the handler work out
  - | Which instruction(s) had exceptions
  - | Which to complete or flush
    - May require “manual” completion
- ❑ Simplifies hardware, but more complex handler software
- ❑ Not feasible for complex multiple-issue out-of-order pipelines

# Instruction-Level Parallelism (ILP)

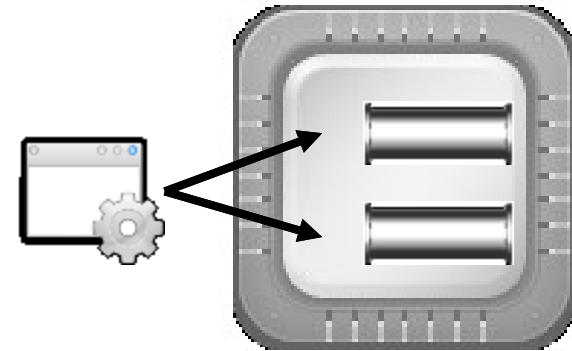
- ❑ Pipelining: executing multiple instructions in parallel
- ❑ To increase ILP
  - | Deeper pipeline
    - Less work per stage  $\Rightarrow$  shorter clock cycle
  - | Multiple issue
    - Replicate pipeline stages  $\Rightarrow$  multiple pipelines
    - Start multiple instructions per clock cycle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

# Multiple Issue

---

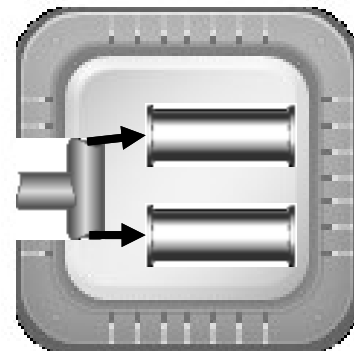
## ❑ Static multiple issue

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



## ❑ Dynamic multiple issue

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



# Speculation

---

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

Speculation /,spekju'leɪʃn/: Suy  
đoán

# ~~Compiler/Hardware Speculation~~

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

# Speculation and Exceptions

---

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

# Static Multiple Issue

---

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

issue packet 1

- instruction X1
- instruction Y1

issue packet 2

- instruction X2
- instruction Y2

# Scheduling Static Multiple Issue

---

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



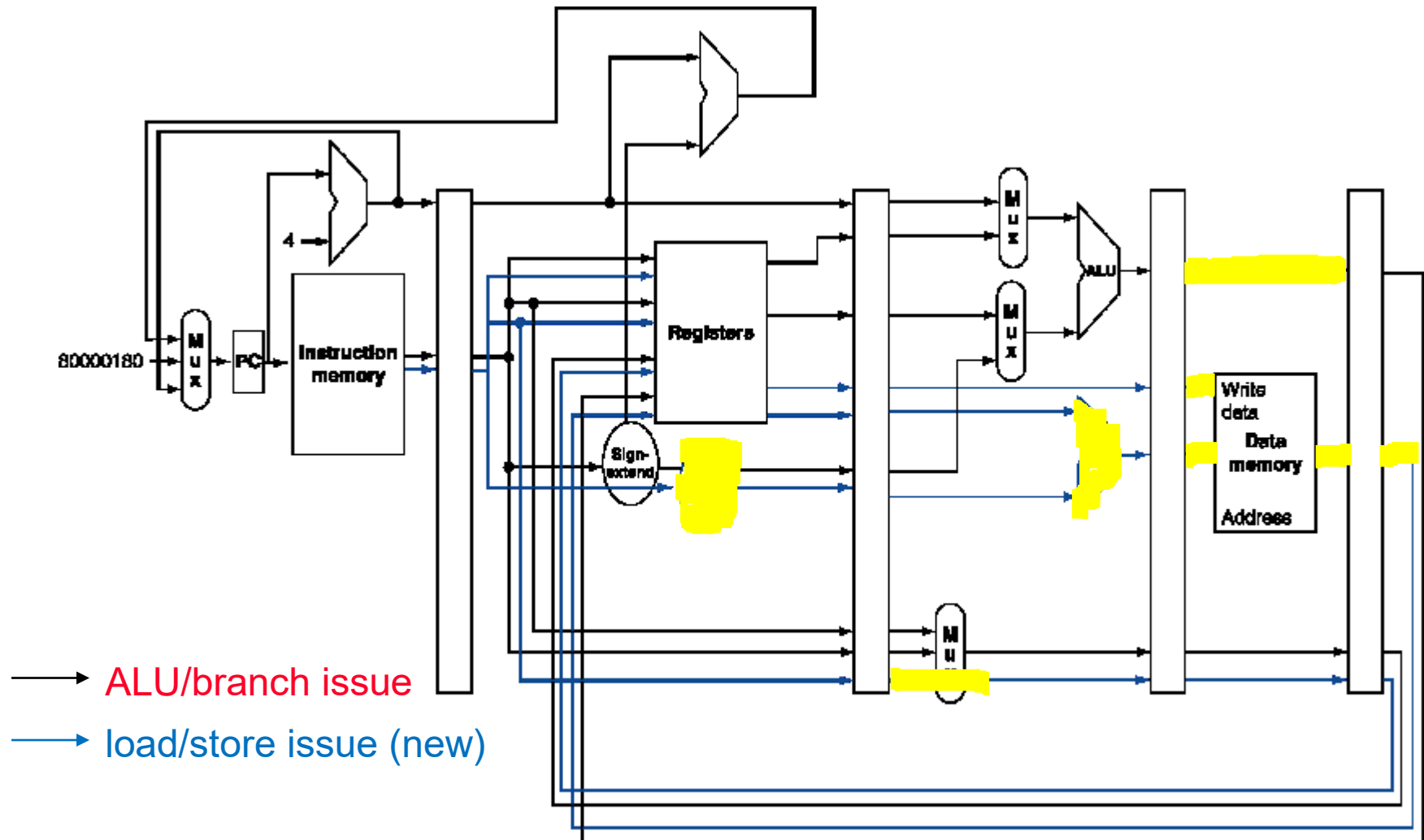
# MIPS with Static Dual Issue

## ❑ Two-issue packets

- | One ALU/branch instruction
- | One load/store instruction
- | 64-bit aligned
  - ALU/branch, then load/store
  - Pad an unused instruction with nop

		Address	Instruction type	Pipeline Stages					
packet 1	{	n	ALU/branch	IF	ID	EX	MEM	WB	
		n + 4	Load/store	IF	ID	EX	MEM	WB	
packet 2	{	n + 8	ALU/branch		IF	ID	EX	MEM	WB
		n + 12	Load/store		IF	ID	EX	MEM	WB
packet 3	{	n + 16	ALU/branch			IF	ID	EX	MEM
		n + 20	Load/store			IF	ID	EX	MEM

# MIPS with Static Dual Issue



# Hazards in the Dual-Issue MIPS

---

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

# Scheduling Example

## □ Schedule this for dual-issue MIPS

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

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

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

# Loop Unrolling

---

- ❑ Replicate loop body to expose more parallelism
  - | Reduces loop-control overhead E.g. ~~do while(...)~~
- ❑ Use different registers per replication
  - | Called “register renaming”
  - | Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

*Unrolling: tháo tung, mở rộng*

# Loop Unrolling Example

For (...)

$A[i] = A[i] + \$s2$



For (...)

$A[i] = A[i] + \$s2$ ;  $A[i+1] = A[i+1] + \$s2$ ;  
 $A[i+2] = A[i+2] + \$s2$ ;  $A[i+3] = A[i+3] + \$s2$ ;

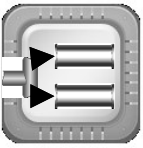
	ALU/branch	Load/store	cycle
Loop:	addi \$s1, \$s1, -16	lw \$t0, 0(\$s1) //A[i]	1
	nop	lw \$t1, 12(\$s1) //A[i+1]	2
	addu \$t0, \$t0, \$s2	lw \$t2, 8(\$s1) //A[i+2]	3
	addu \$t1, \$t1, \$s2	lw \$t3, 4(\$s1) //A[i+3]	4
	addu \$t2, \$t2, \$s2	sw \$t0, 16(\$s1) //A[i]	5
	addu \$t3, \$t3, \$s2	sw \$t1, 12(\$s1)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

□  $IPC = 14/8 = 1.75$

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

# Dynamic Multiple Issue

---



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

# Dynamic Pipeline Scheduling

---

- ❑ Allow the CPU to execute instructions out of order to avoid stalls

- | But commit result to registers in order

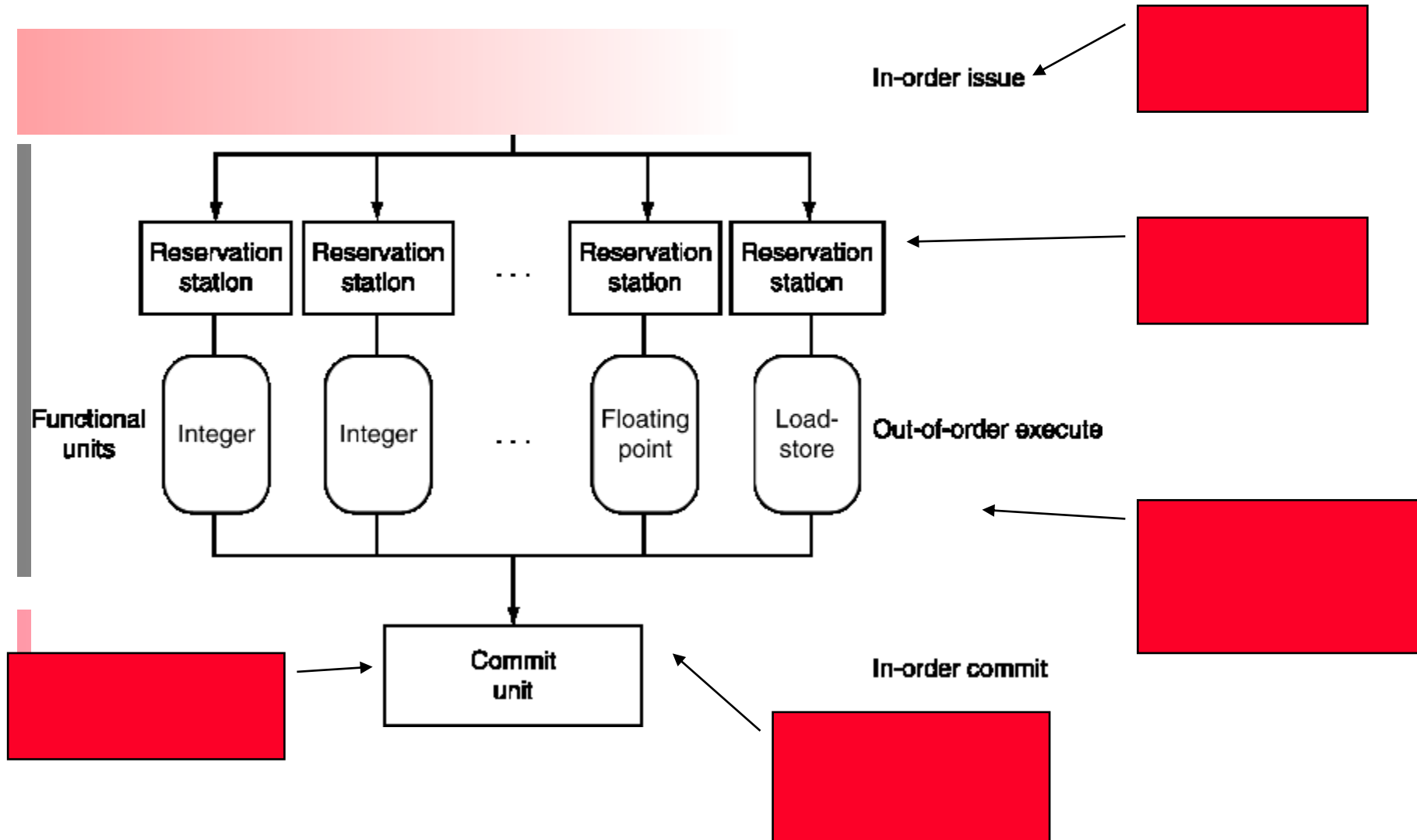
- ❑ Example

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

- | Can start sub while addu is waiting for lw



# Dynamically Scheduled CPU



# Register Renaming

---

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

# Speculation

---

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

# Why Do Dynamic Scheduling?

---

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

# Does Multiple Issue Work?

---

## The BIG Picture

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

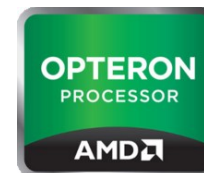
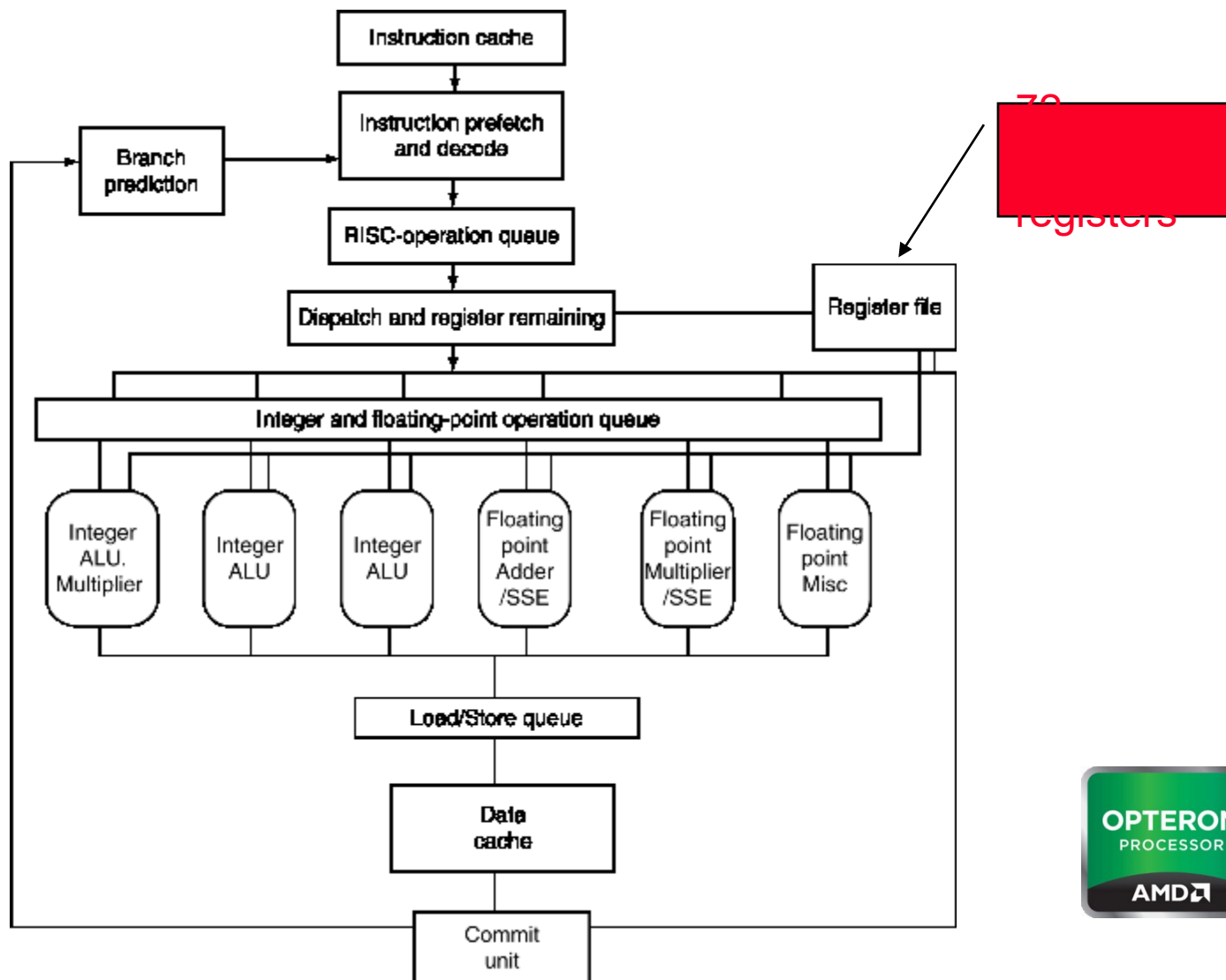
# Power Efficiency

---

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

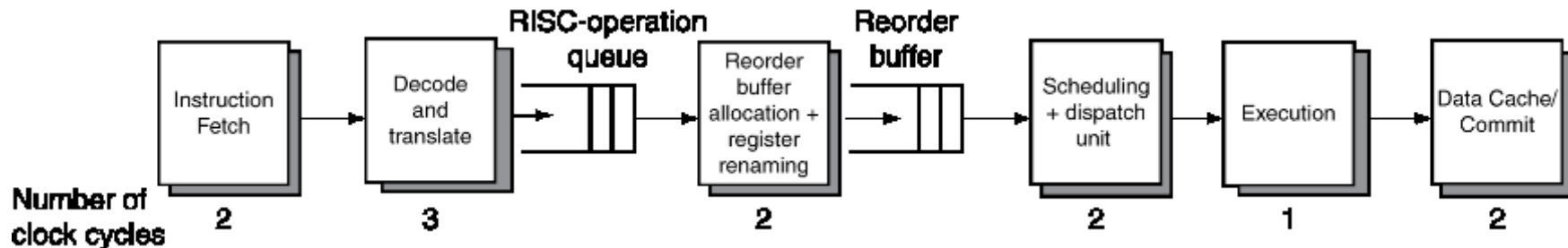
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4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

# The Opteron X4 Microarchitecture



# The Opteron X4 Pipeline Flow

## ❑ For integer operations



- FP is 5 stages longer
- Up to 106 RISC-ops in progress

## ■ Bottlenecks

- Complex instructions with long dependencies
- Branch mispredictions
- Memory access delays



# Fallacies

---

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

# Pitfalls

---

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

## Concluding Remarks

---

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