



Chapter 4

The Processor

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

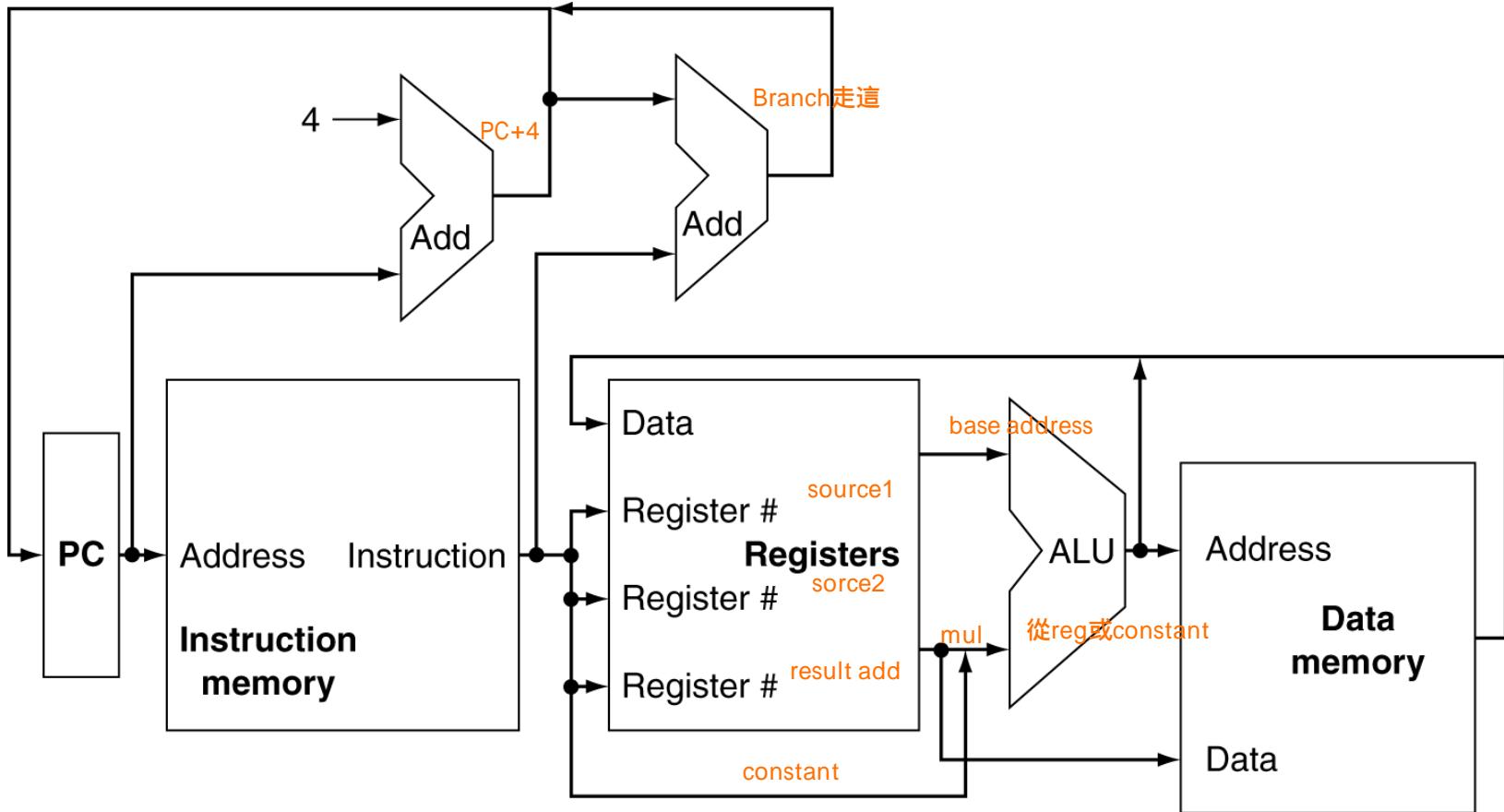


Instruction Execution

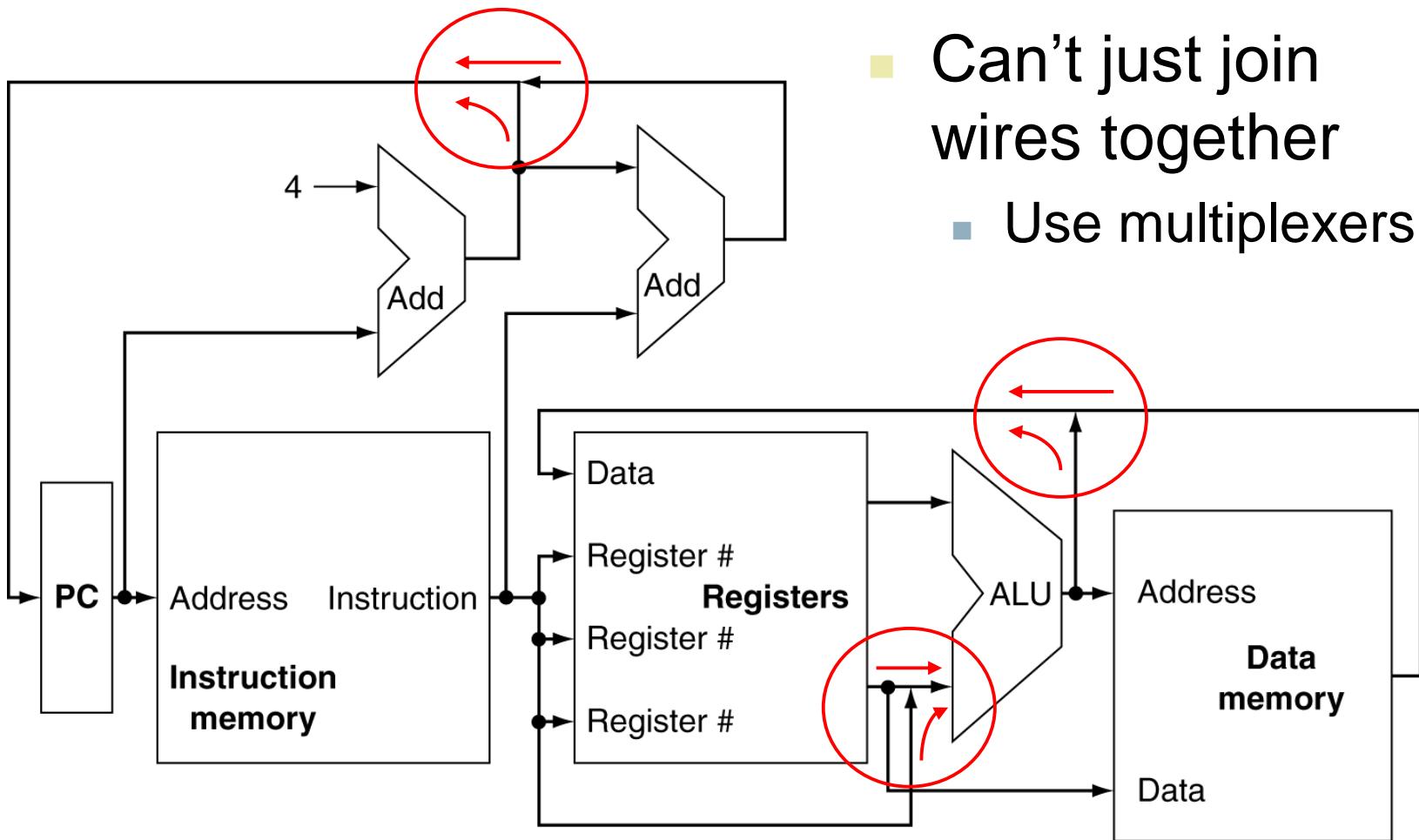
- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
 - Use ALU to calculate
 - Arithmetic result
 - Memory address for load/store
 - Branch target address
 - Access data memory for load/store
 - PC ← target address or PC + 4

one cycle one instruction

CPU Overview

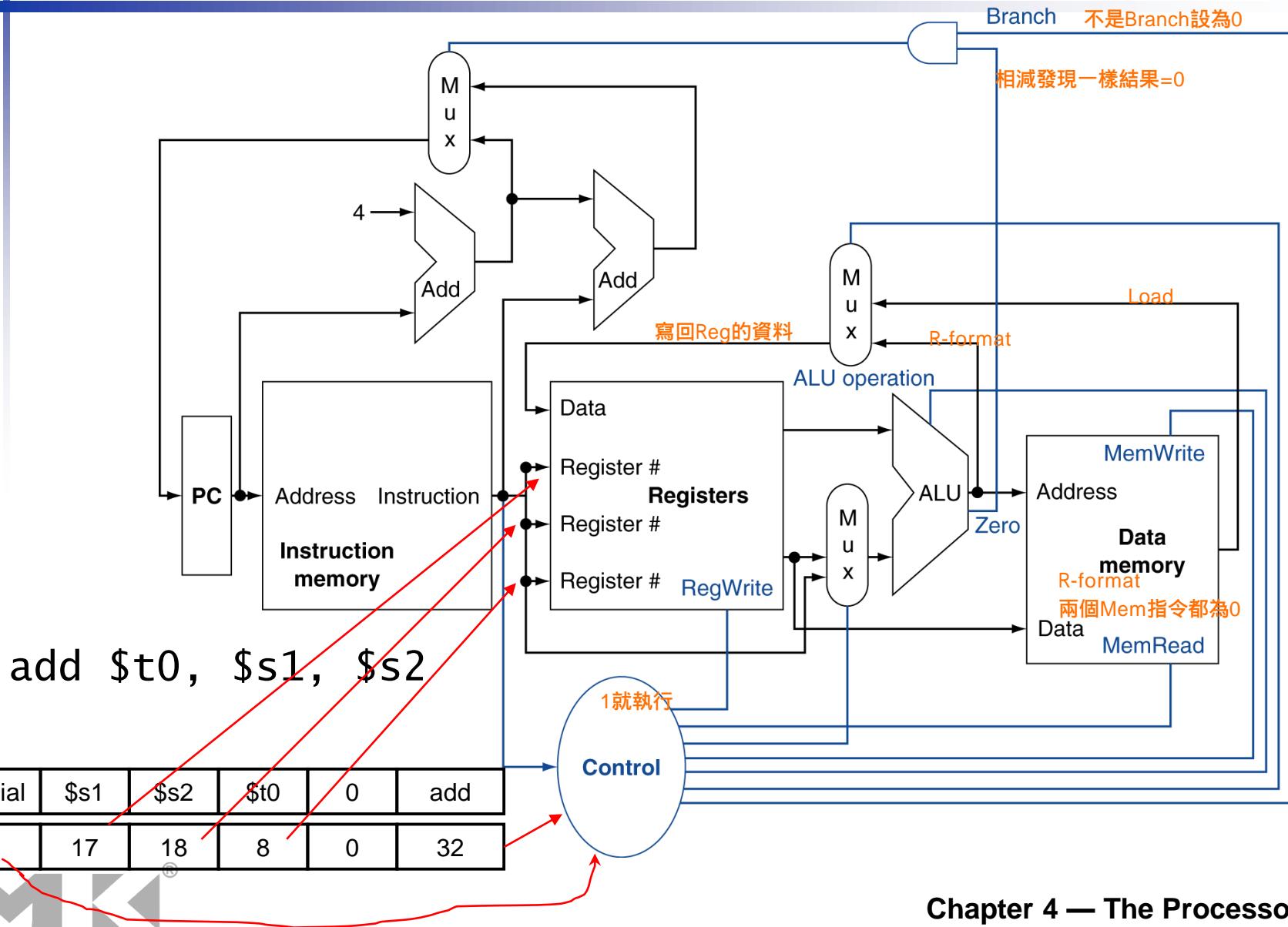


Multiplexers



- Can't just join wires together
 - Use multiplexers

Control

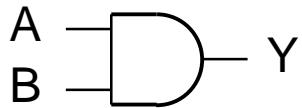


Logic Design Basics

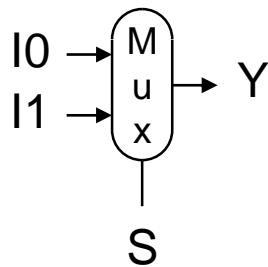
- Information encoded in binary
 - Low voltage = 0, High voltage = 1
 - One wire per bit
 - Multi-bit data encoded on multi-wire buses
- Combinational element
 - Operate on data
 - Output is a function of input
- State (sequential) elements
 - Store information

Combinational Elements

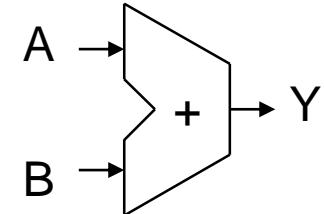
- AND-gate
 - $Y = A \& B$



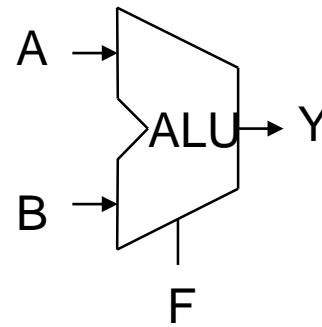
- Multiplexer
 - $Y = S ? I_1 : I_0$



- Adder
 - $Y = A + B$

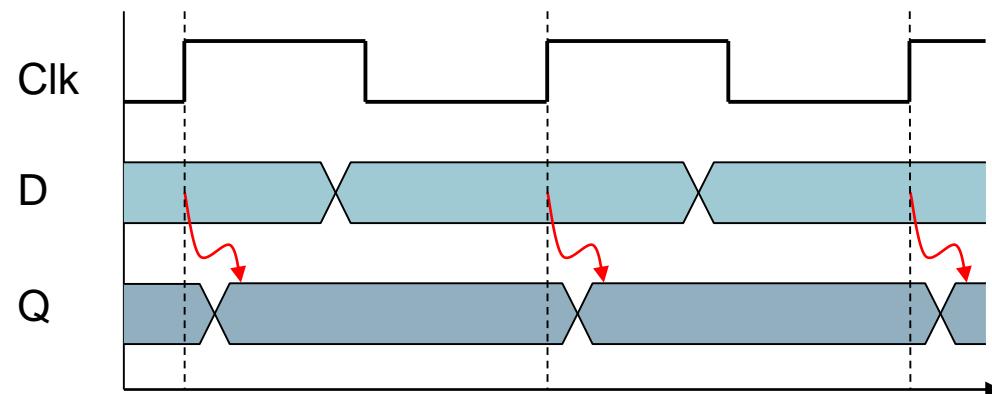
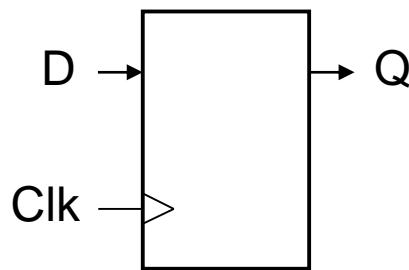


- Arithmetic/Logic Unit
 - $Y = F(A, B)$



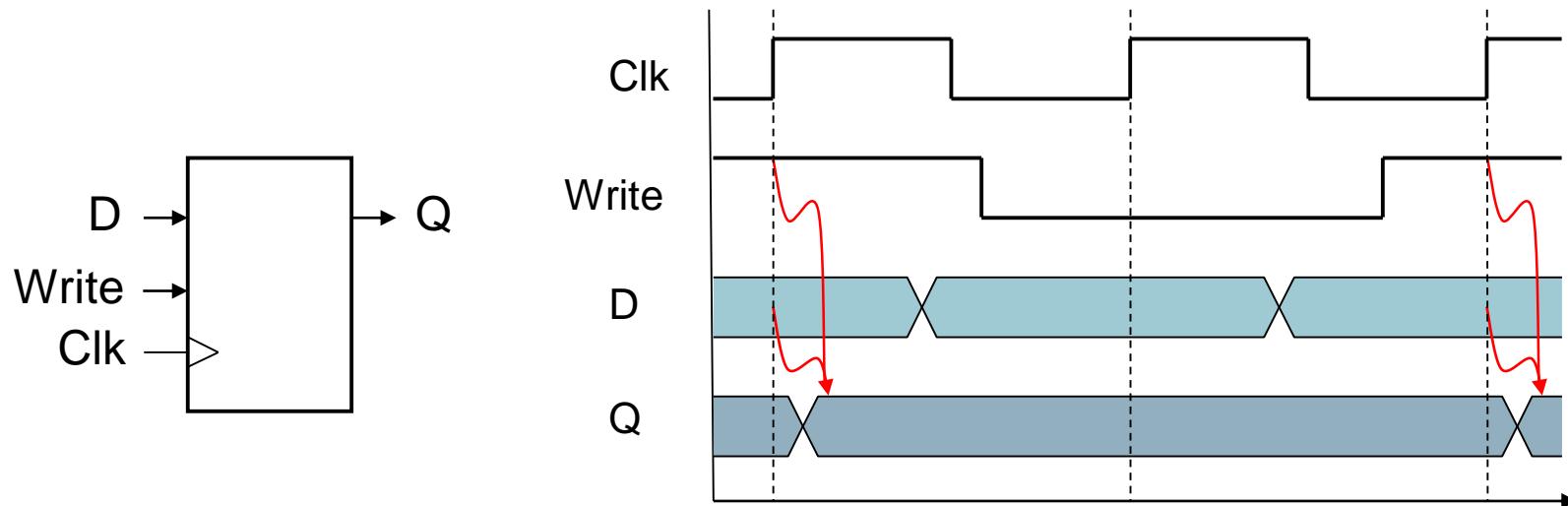
Sequential Elements

- 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



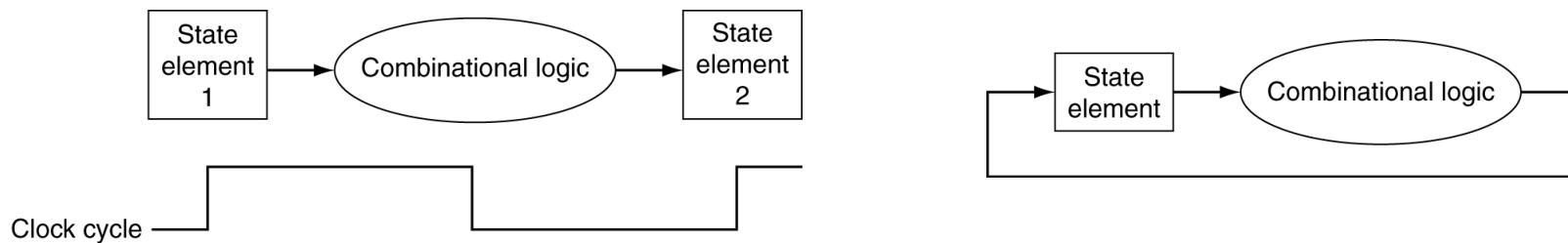
Sequential Elements

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



Clocking Methodology

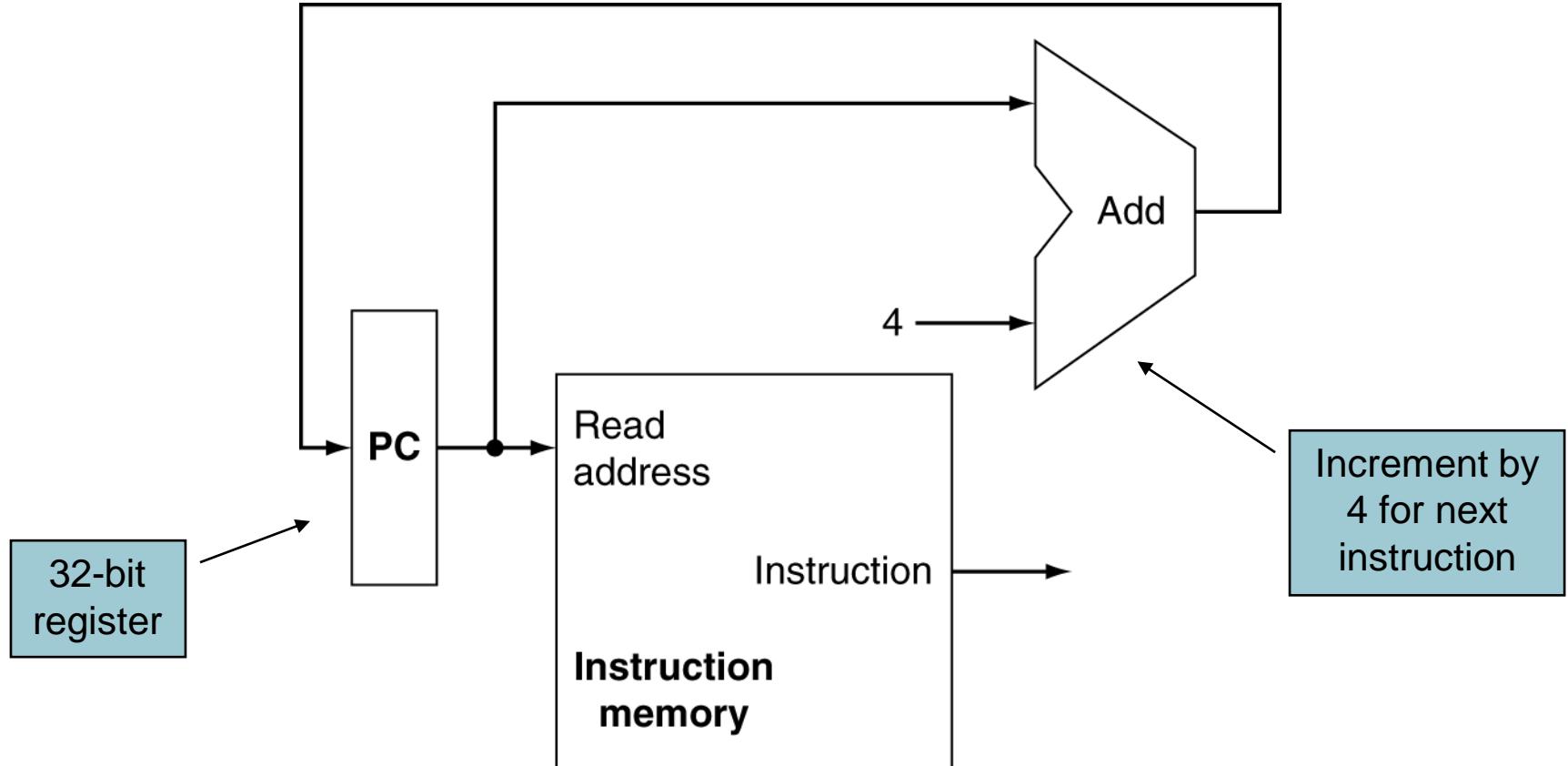
- Combinational logic transforms data during clock cycles
 - Between clock edges
 - Input from state elements, output to state element
 - Longest delay determines clock period



Building a Datapath

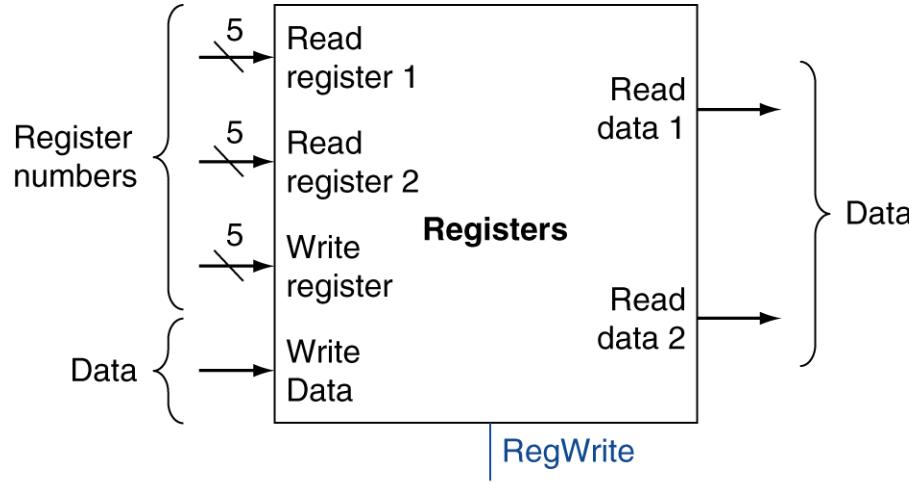
- Datapath
 - Elements that process data and addresses in the CPU
 - Registers, ALUs, mux's, memories, ...
- We will build a MIPS datapath incrementally
 - Refining the overview design

Instruction Fetch

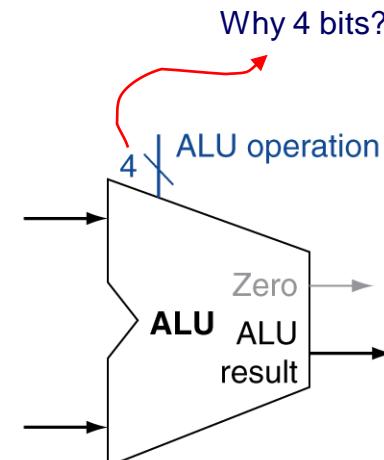


R-Format Instructions

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



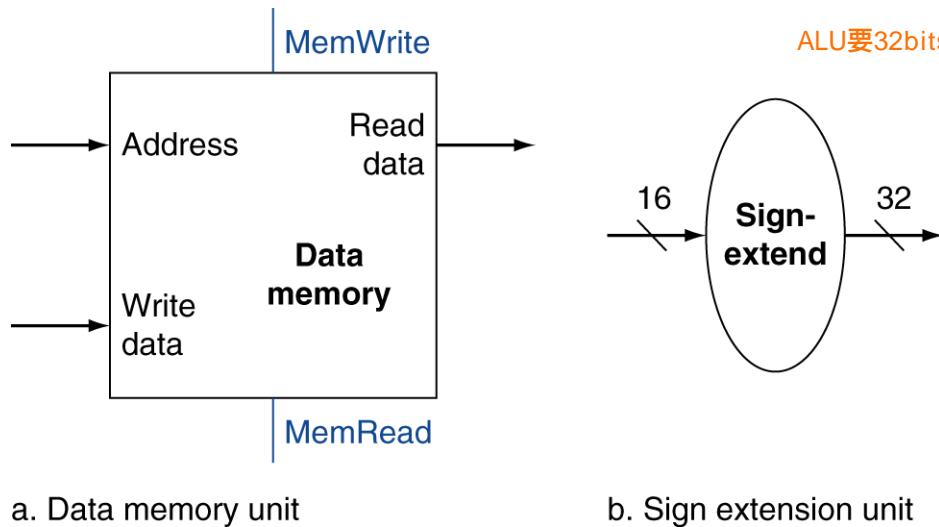
a. Registers



b. ALU

Load/Store Instructions

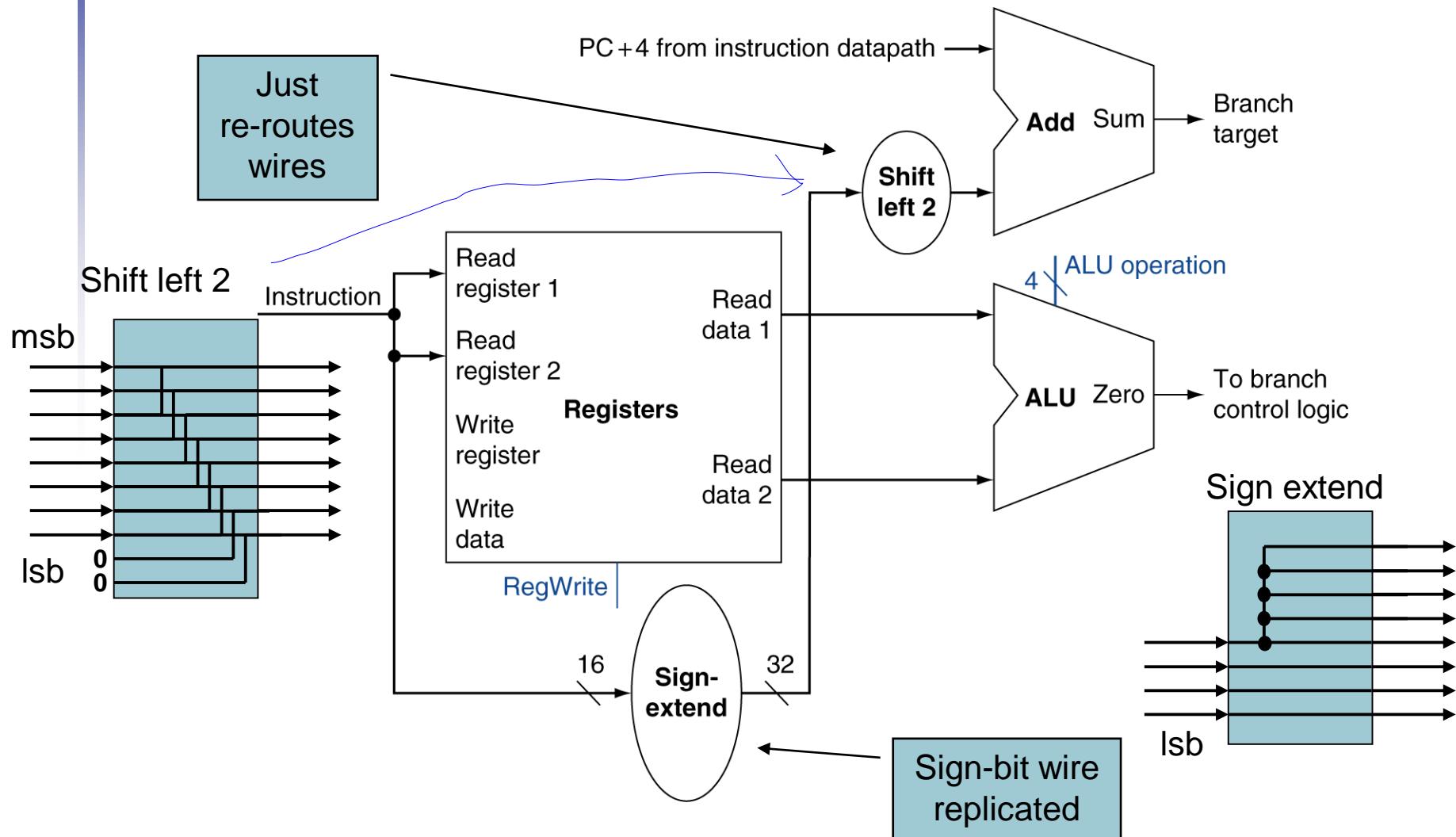
- Read register operands
- Calculate address using 16-bit offset
 - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



Branch Instructions

- Read register operands
- 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

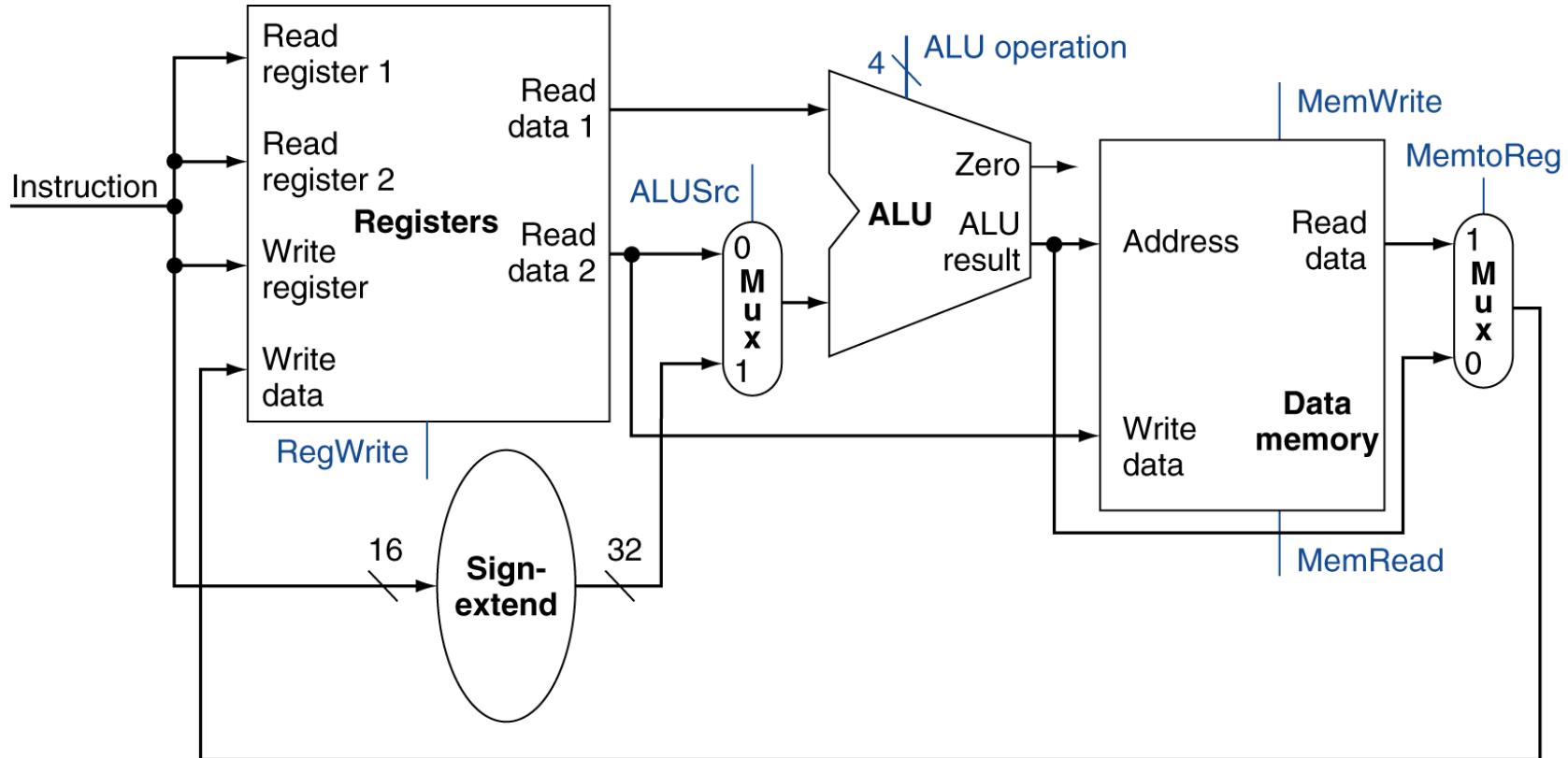


Composing the Elements

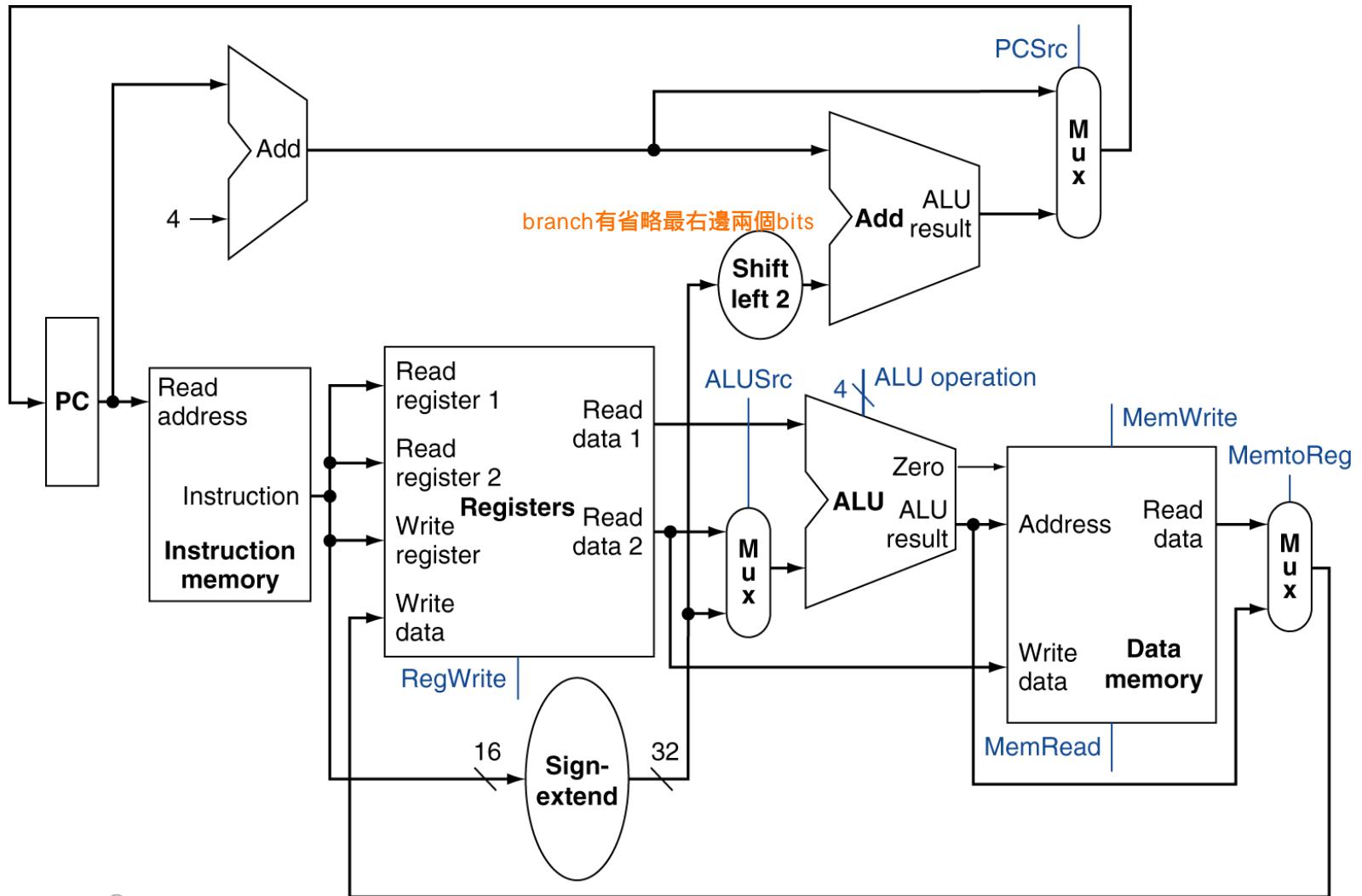
- First-cut data path does an instruction in one clock cycle
 - Each datapath element can only do one function at a time
 - Hence, we need to separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

—開始access instruction後面才抓data memories

R-Type/Load/Store Datapath

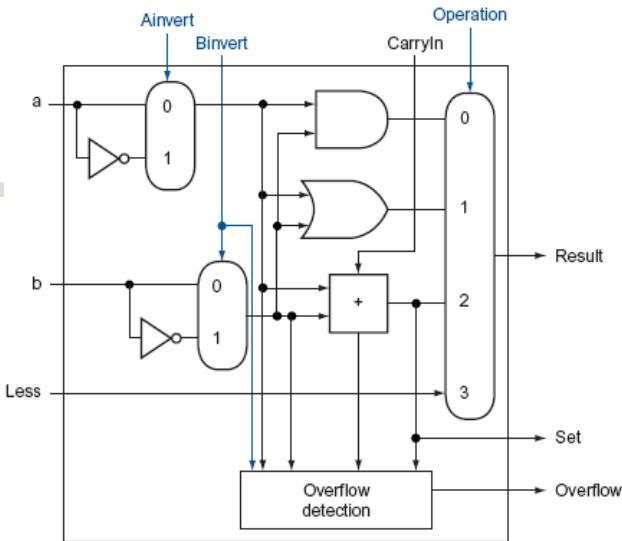


Full Datapath



ALU Control

- ALU used for
 - Load/Store: $F = \text{add}$
 - Branch: $F = \text{subtract}$
 - 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

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

Register Write Enable

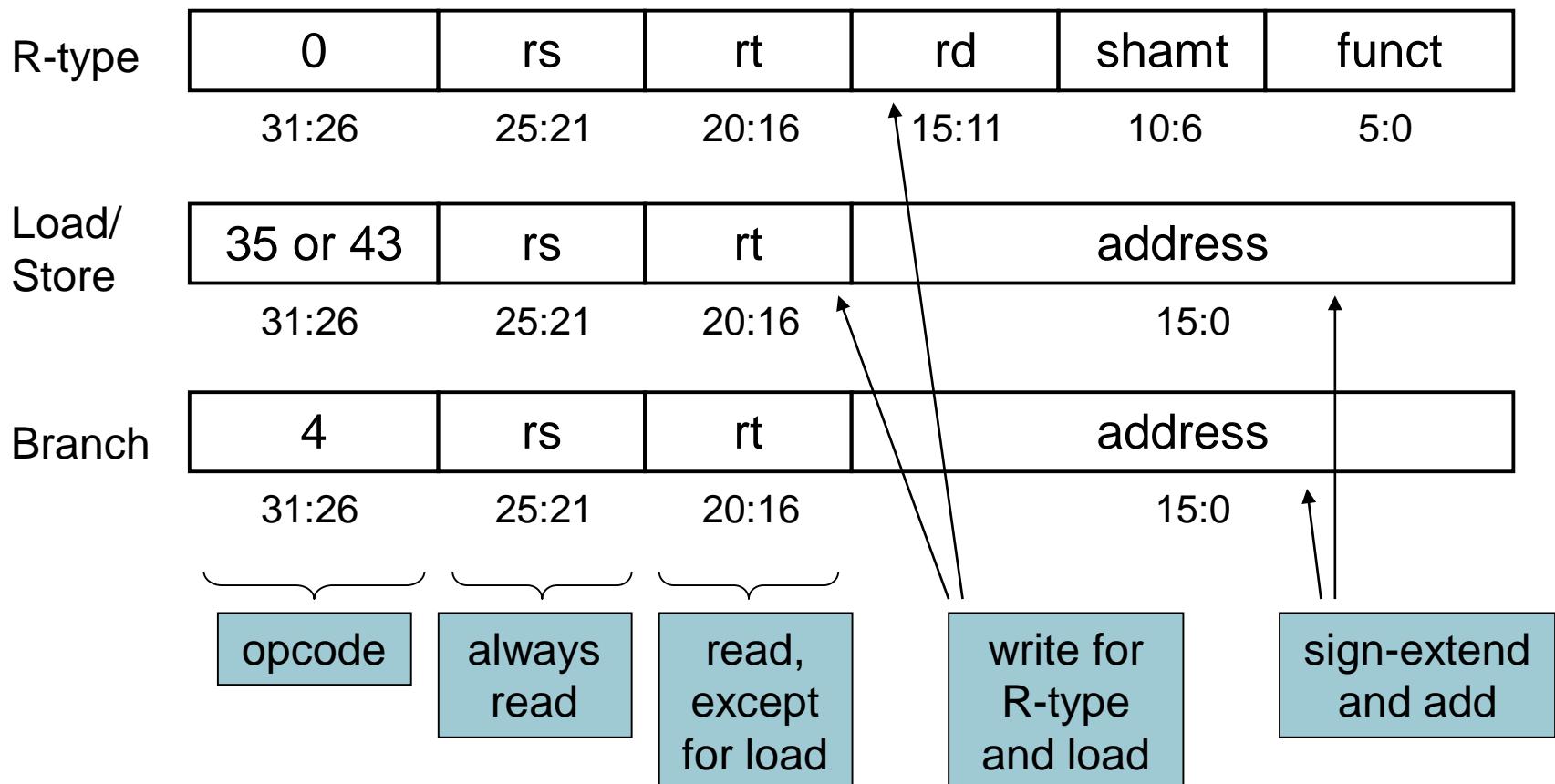
Memory Read/Write Enable

Multiplexer selection

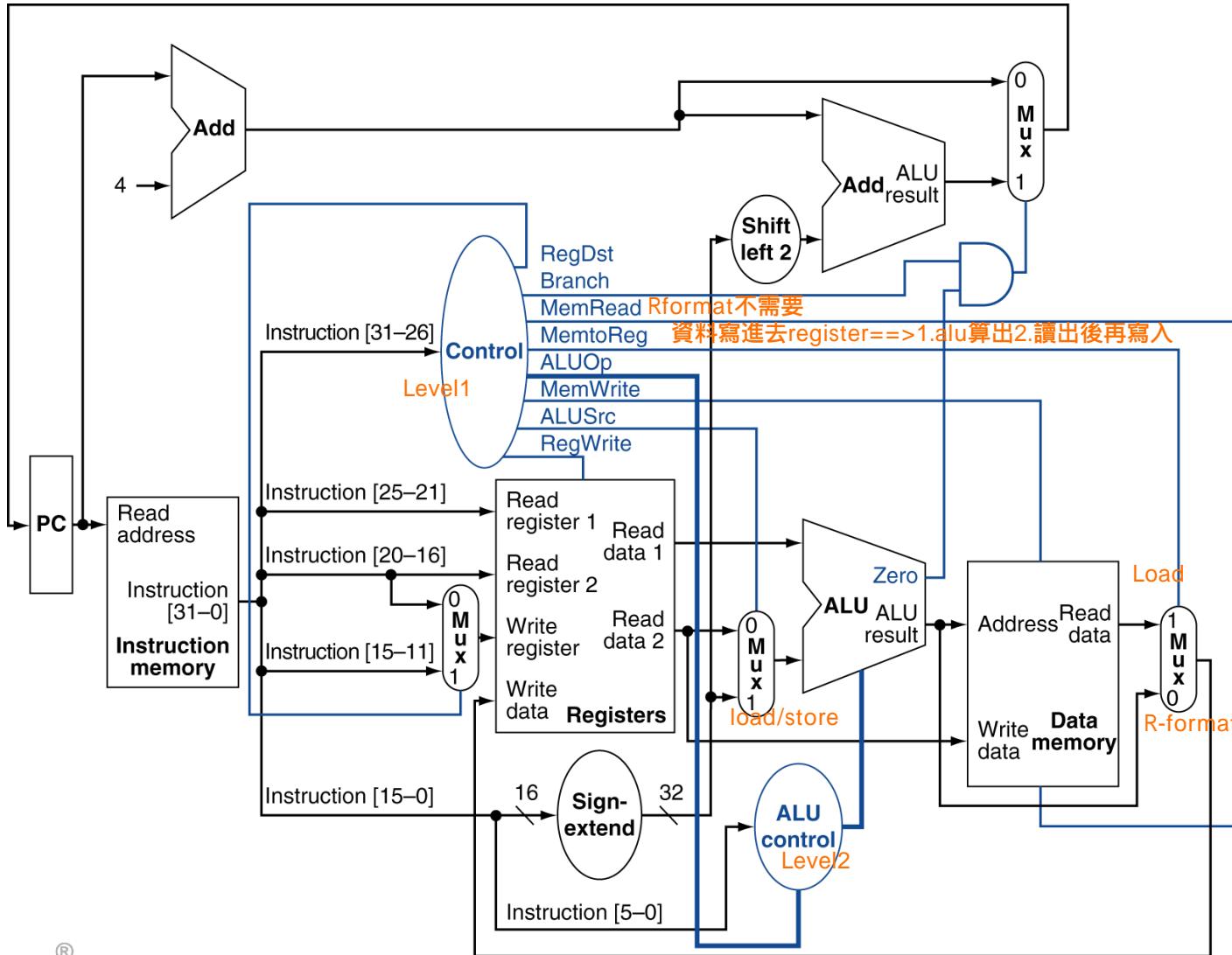
ALU Control

The Main Control Unit

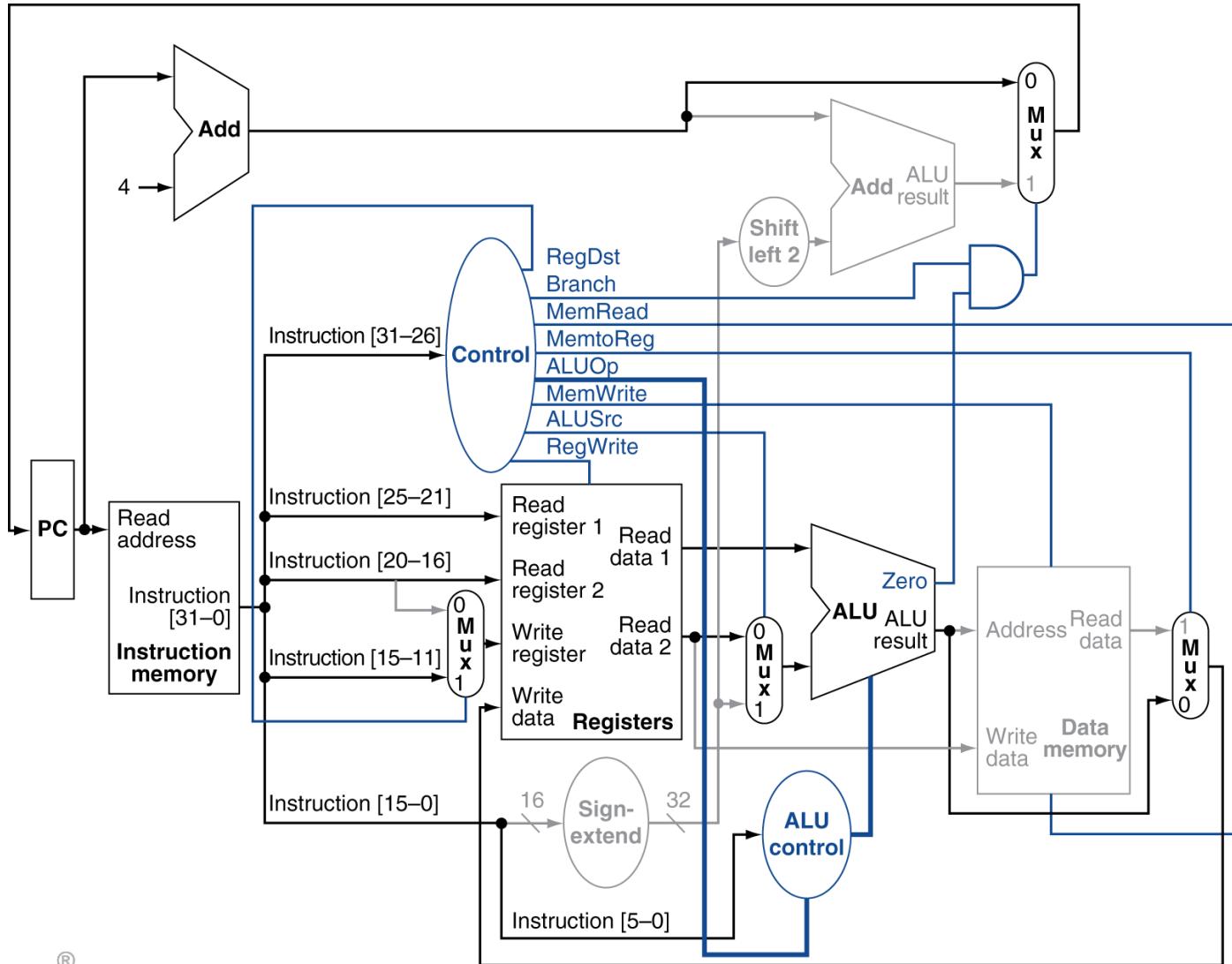
Control signals derived from instruction



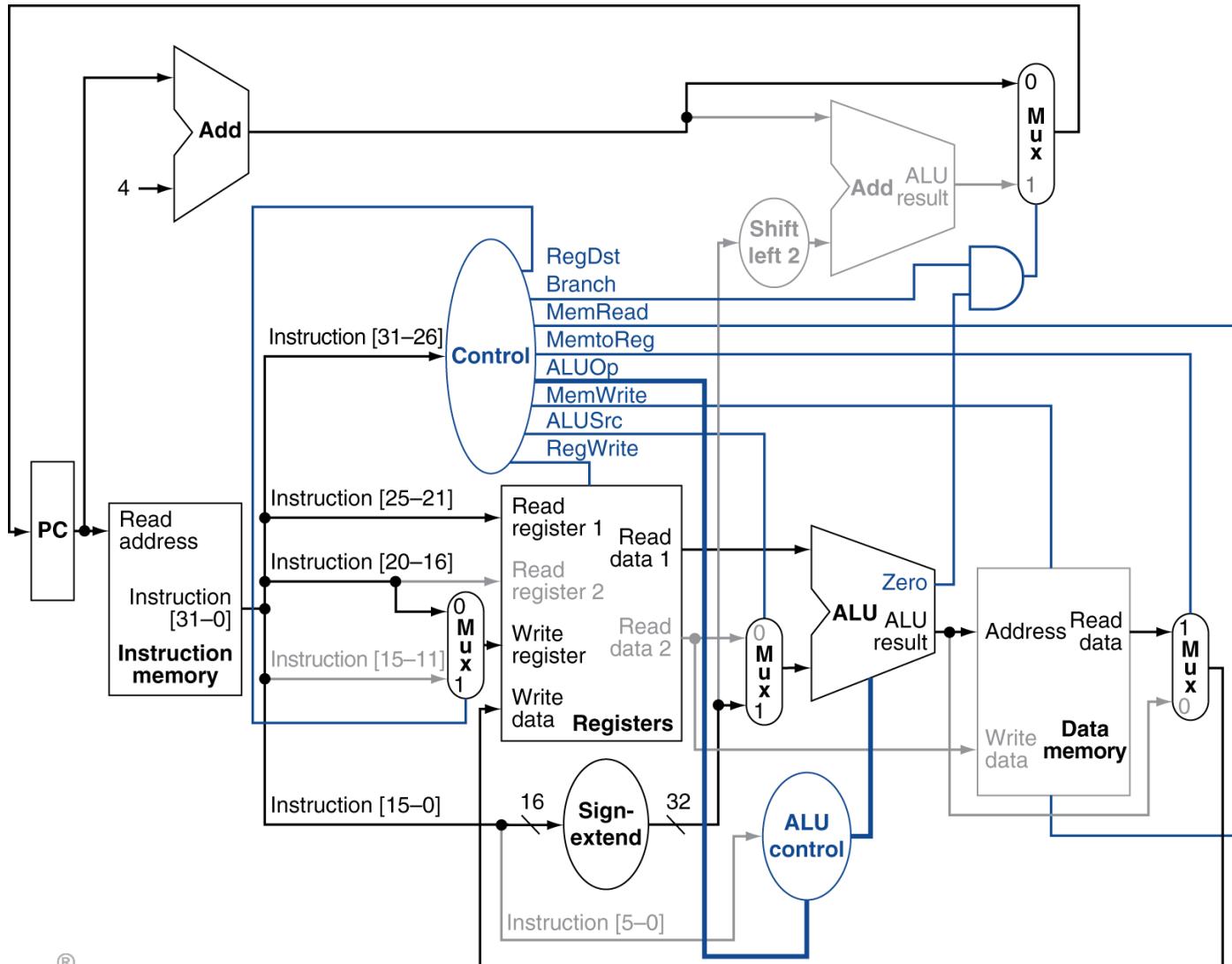
Datapath With Control



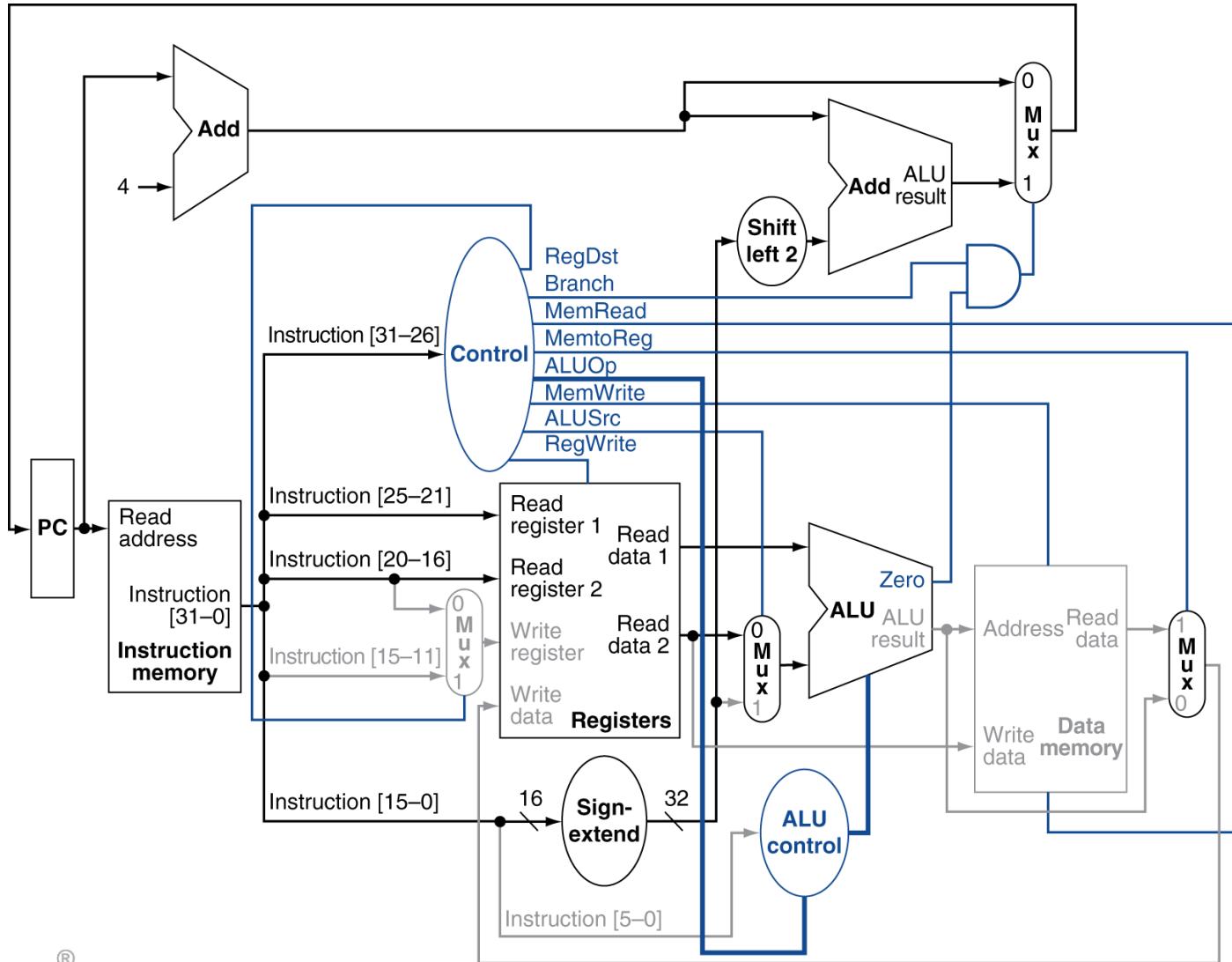
R-Type Instruction



Load Instruction

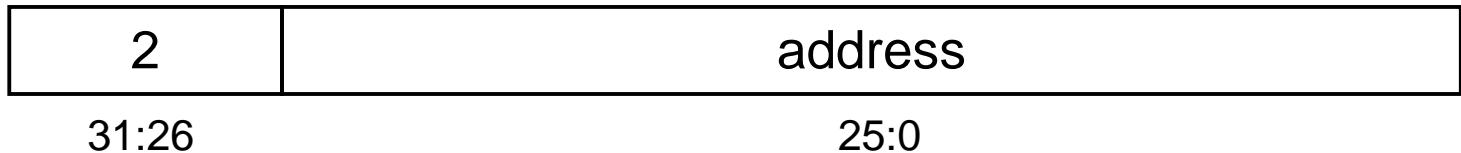


Branch-on-Equal Instruction



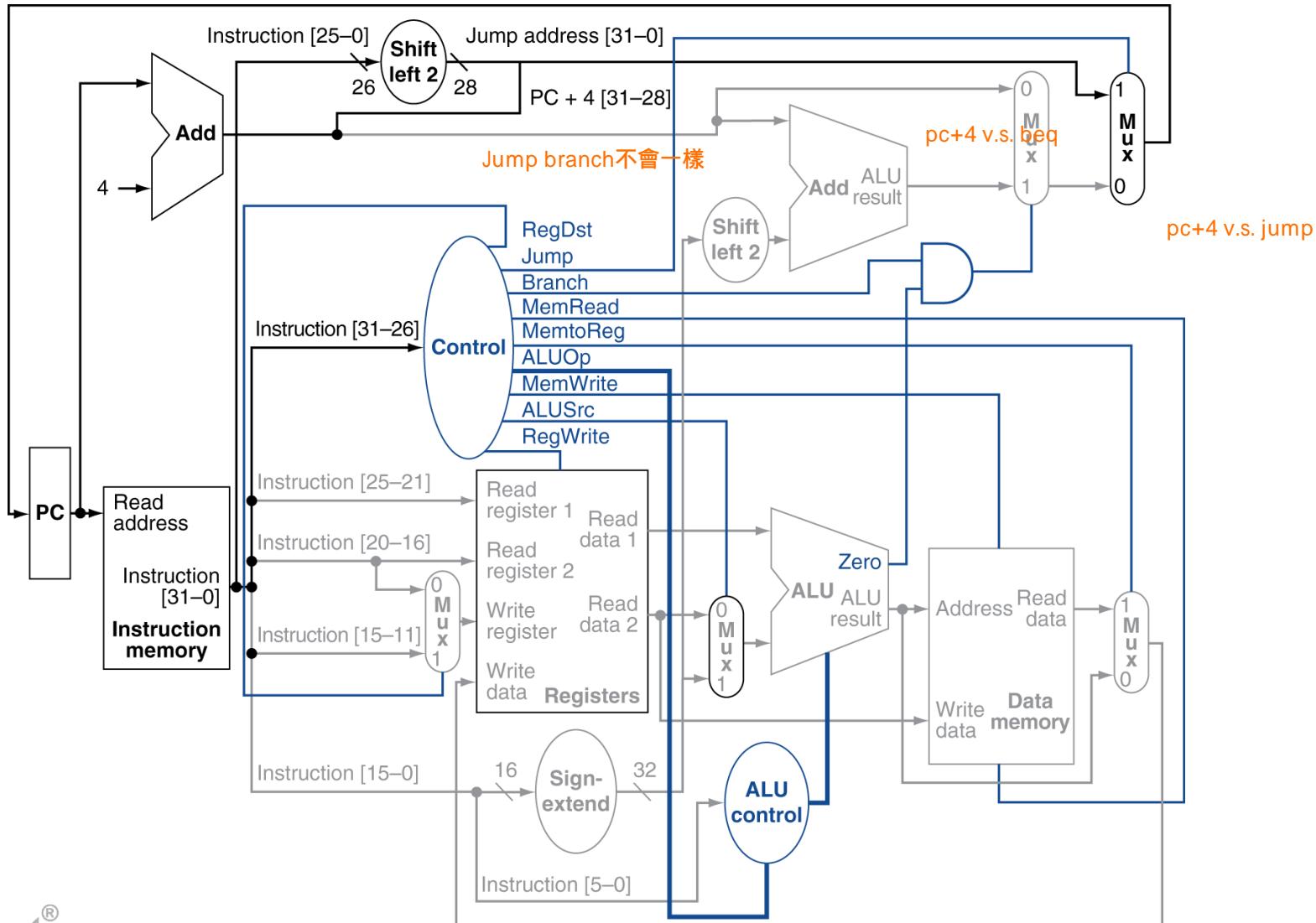
Implementing Jumps

Jump



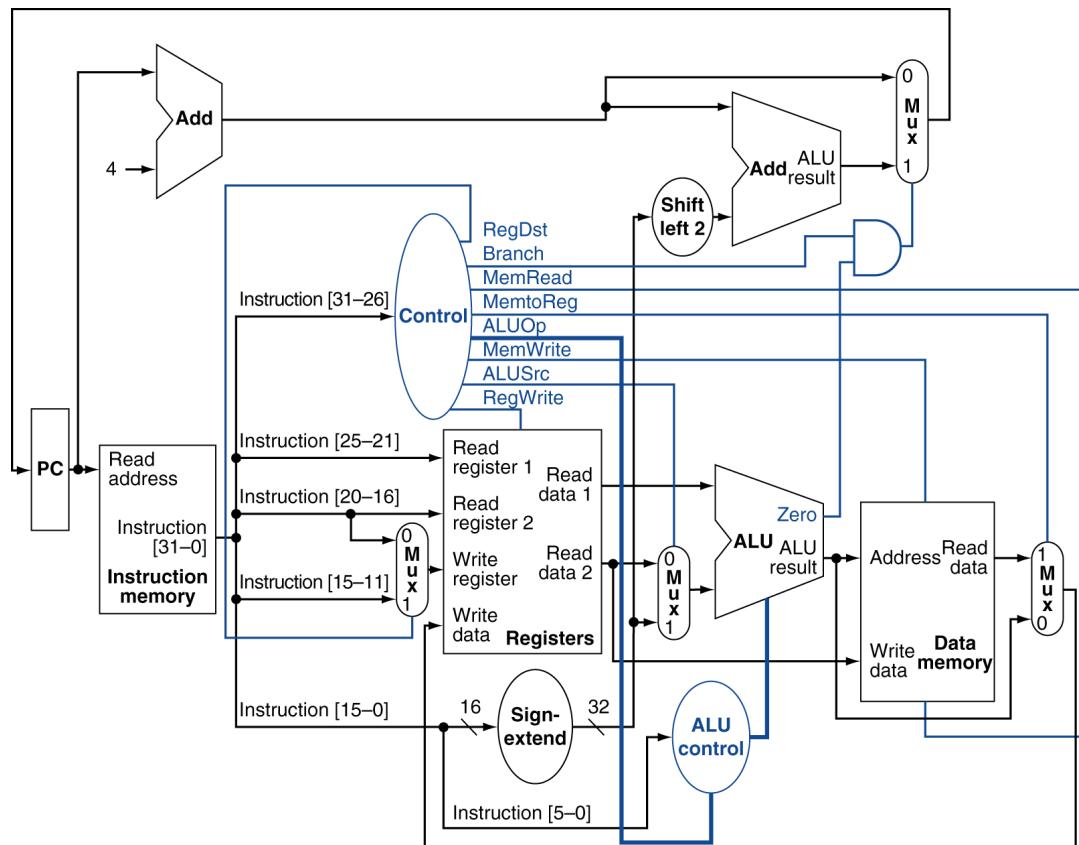
- Jump uses word address
- Update PC with concatenation of
 - Top 4 bits of old PC
 - 26-bit jump address
 - 00
- Need an extra control signal decoded from opcode

Datapath With Jumps Added



Control Signal Settings

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0 ^{Alu}	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	沒有要寫到register	X	1	X	0	0	1	0	0
beq	X	0	X	0	0	0	1	0	1



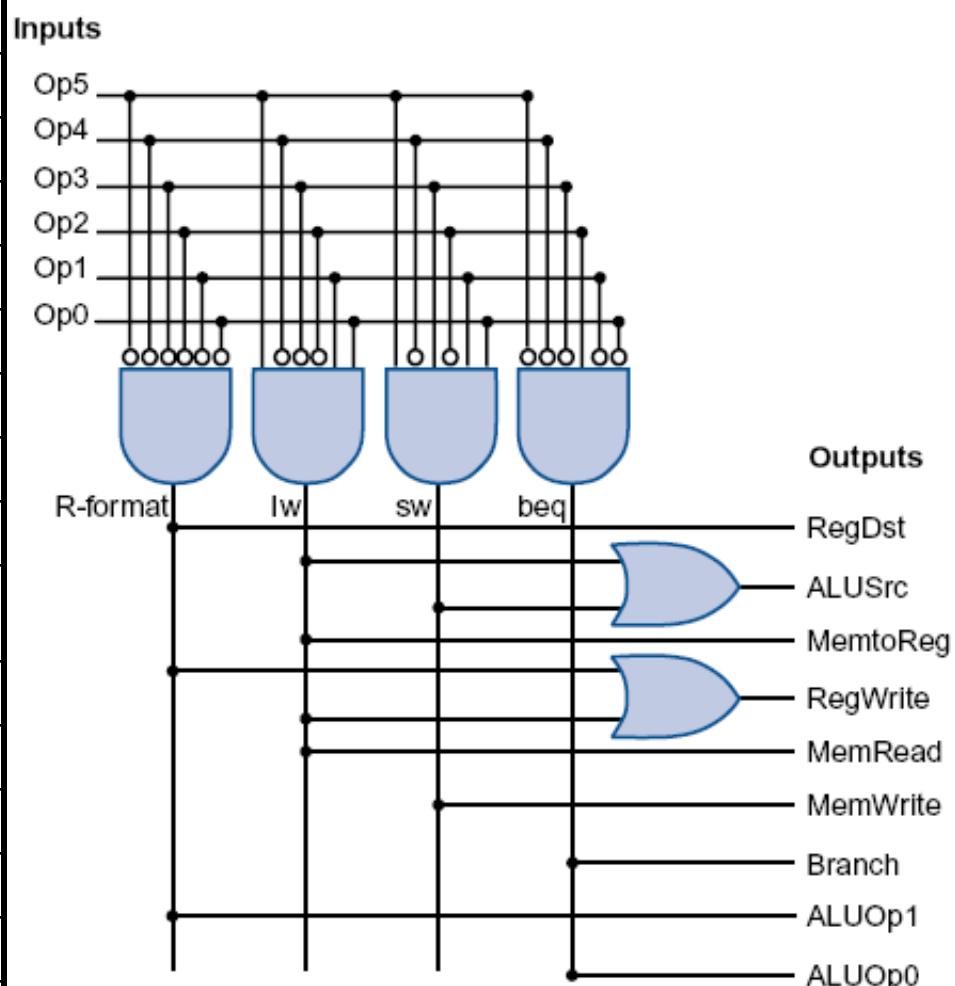
Control Signal Settings

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
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
Input or output	Signal name	R-format		lw		sw		beq	
Inputs	Op5	0		1		1		0	
	Op4	0		0		0		0	
	Op3	0		0		1		0	
	Op2	0		0		0		1	
	Op1	0		1		1		0	
	Op0	0		1		1		0	
Outputs	RegDst	1		0		X		X	
	ALUSrc	0		1		1		0	
	MemtoReg	0		1		X		X	
	RegWrite	1		1		0		0	
	MemRead	0		1		0		0	
	MemWrite	0		0		1		0	
	Branch	0		0		0		1	
	ALUOp1	1		0		0		0	
	ALUOp2	0		0		0		1	



The First-Level Control Unit (Main)

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp2	0	0	0	1



Two-Level Control Unit – the Second Level

- Describe it using a truth table (can turn into gates):

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	Load word	XXXXXX	Add	0010
SW	00	Store word	XXXXXX	Add	0010
Branch equal	01	Branch equal	XXXXXX	Subtract	0110
R-type	10	Add	100000	Add	0010
R-type	10	Subtract	100010	Subtract	0110
R-type	10	AND	100100	And	0000
R-type	10	OR	100101	Or	0001
R-type	10	Slt	101010	Set less than	0111

ALUOp		Funct field							Operation output		
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	C ₃	C ₂	C ₁	C ₀
0	0	X	X	X	X	X	X	0	0	1	0 (lw/sw)
X	1	X	X	X	X	X	X	0	1	1	0 (beq)
1	X	X	X	0	0	0	0	0	0	1	0 (add)
1	X	X	X	0	0	1	0	0	1	1	0 (sub)
1	X	X	X	0	1	0	0	0	0	0	0 (AND)
1	X	X	X	0	1	0	1	0	0	0	1 (OR)
1	X	X	X	1	0	1	0	0	1	1	1 (slt)

特殊簡化:==>只看ALUOp12, 如果要確定是beq會用哪一個bit去確認?選ALUOp0

The Second-Level Control Unit (ALU)

如果是AluOp1==1則就是RTYPE

盡量分兩組

C1的AUTOp1設X是因為前面兩個case保證輸出是1了
所以只要考慮下面五個即可!

- Classify rows into 2 sets (1 and 0). Find the simplest rules to distinguish the 1-set from the 0-set.

ALUOp		Funct field						Operation			
		F5	F4	F3	F2	F1	F0	C ₃	C ₂	C ₁	C ₀
1	0	0	X	X	X	X	X	0	0	1	0 (lw/sw)
2	X (0)	1	X	X	X	X	X	0	1	1	0 (beq)
3	1	X (0)	X	X	0	0	0	0	0	1	0 (add)
4	1	X (0)	X	X	0	0	1	0	1	1	0 (sub)
5	1	X (0)	X	X	0	1	0	0	0	0	0 (AND)
6	1	X (0)	X	X	0	1	0	1	0	0	1 (OR)
7	1	X (0)	X	X	1	0	1	0	1	1	1 (slt)

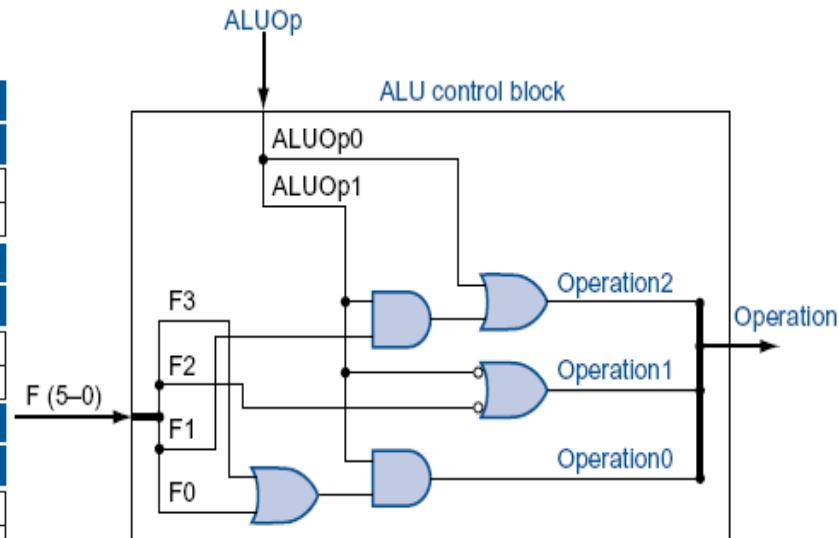
沒辦法區分

就是don't care

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
X	1	X	X	X	X	X	X
1	X	X	X	X	X	1	X

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
0 (也可設1)	X	X	X	X	X	X	X
X	X	X	X	X	0	X	X

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
1	X	X	X	X	X	X	1
1	X	X	X	1	X	X	X



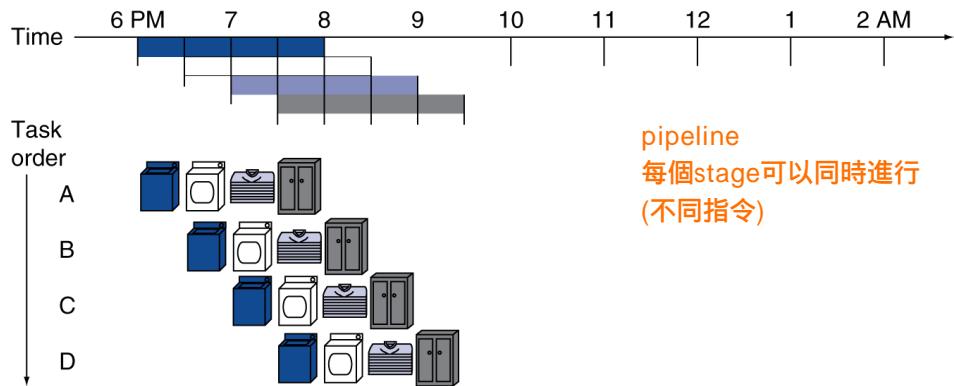
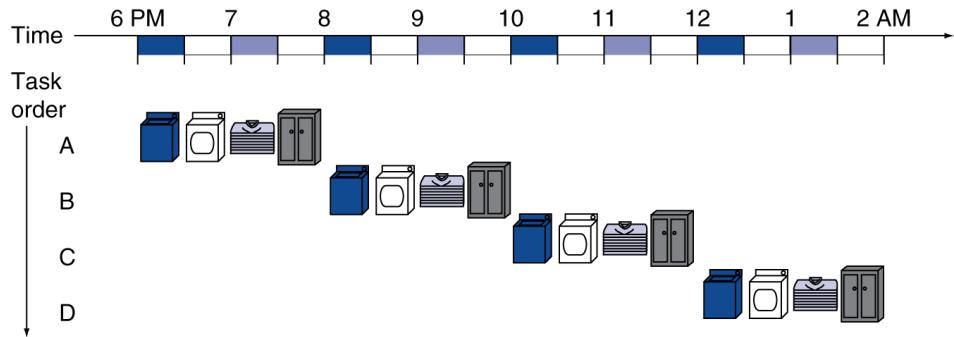
Performance Issues of Single Cycle Design

為什麼需要Pipe line?
每個指令不同但是都用同樣的時間!

- Longest delay determines clock period
 - Critical path: load instruction
 - 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



- Four loads:
 - Speedup
 $= 8/3.5 = 2.3$
 - Non-stop:
 - Speedup
 $= 2n/(0.5n + 1.5) \approx 4$
= number of stages
- 第一個結束的前三個
stage結束需要時間

MIPS Pipeline

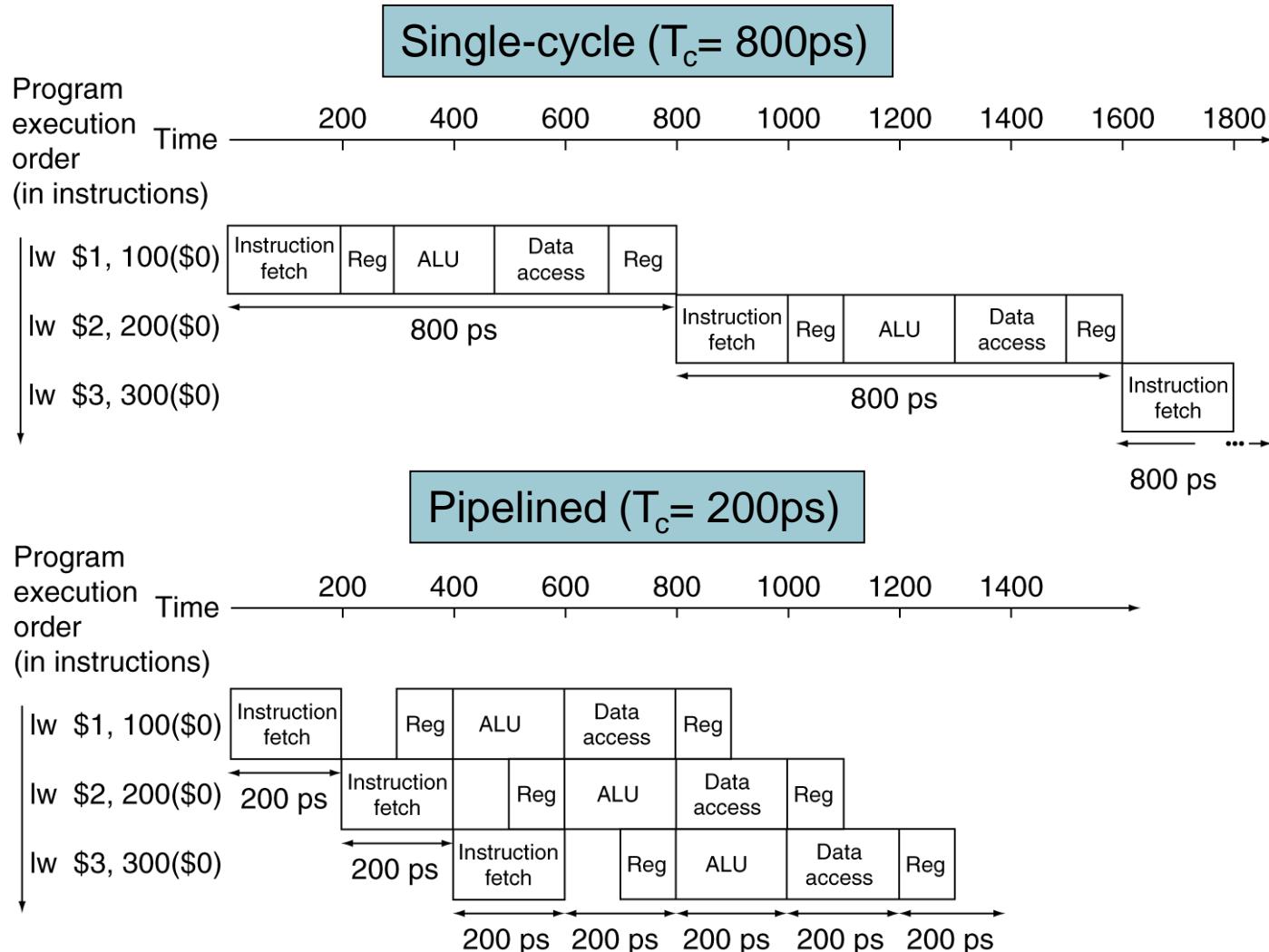
- Five stages, one step per stage
 1. IF: Instruction fetch from memory
 2. ID: Instruction decode & register read
 3. EX: Execute operation or calculate address
 4. MEM: Access memory operand
 5. WB: Write result back to register

Pipeline Performance

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

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
 - Time between instructions_{pipelined}
= Time between instructions_{nonpipelined}
—————
Number of stages
- If not balanced, speedup is less
- 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 3rd stage, access memory in 4th stage
 - Alignment of memory operands
 - Memory access takes only one cycle

難控制!

分兩層指令進行優化!

Hazards

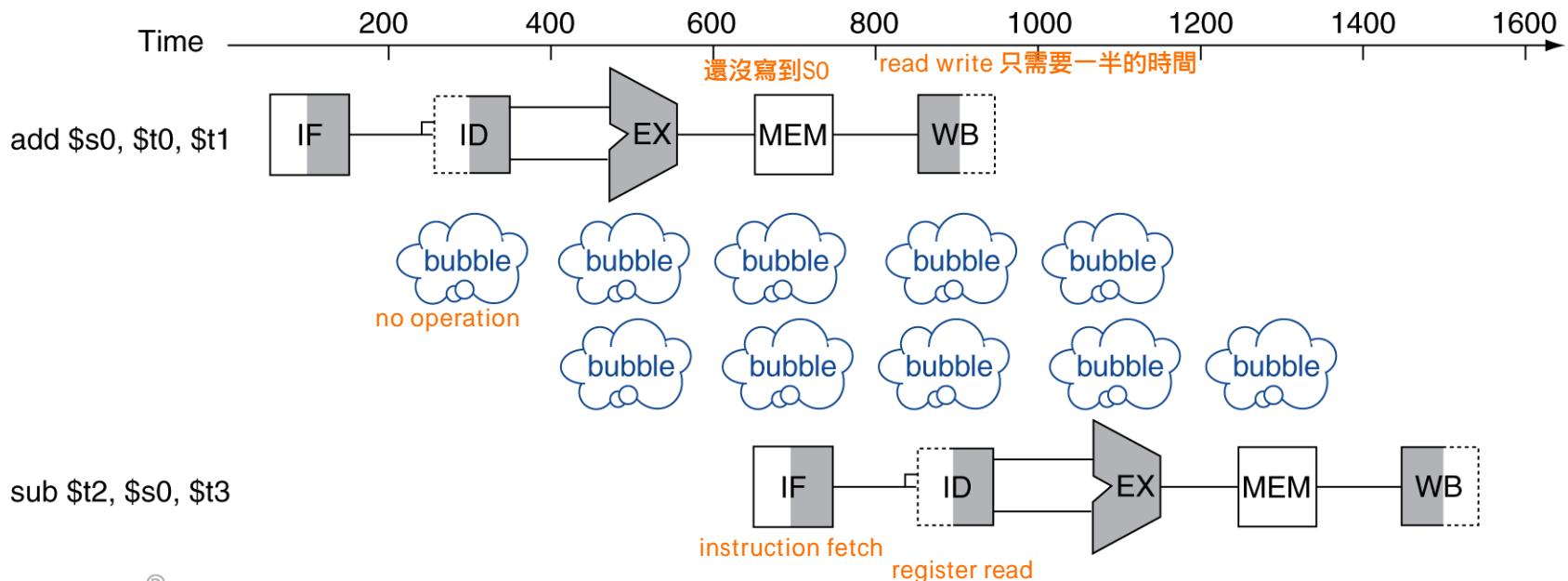
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
 - A required resource is busy
- Data hazard還沒寫進去
 - Need to wait for previous instruction to complete its data read/write
- Control hazardbranch==> 要等
 - Deciding on control action depends on previous instruction

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

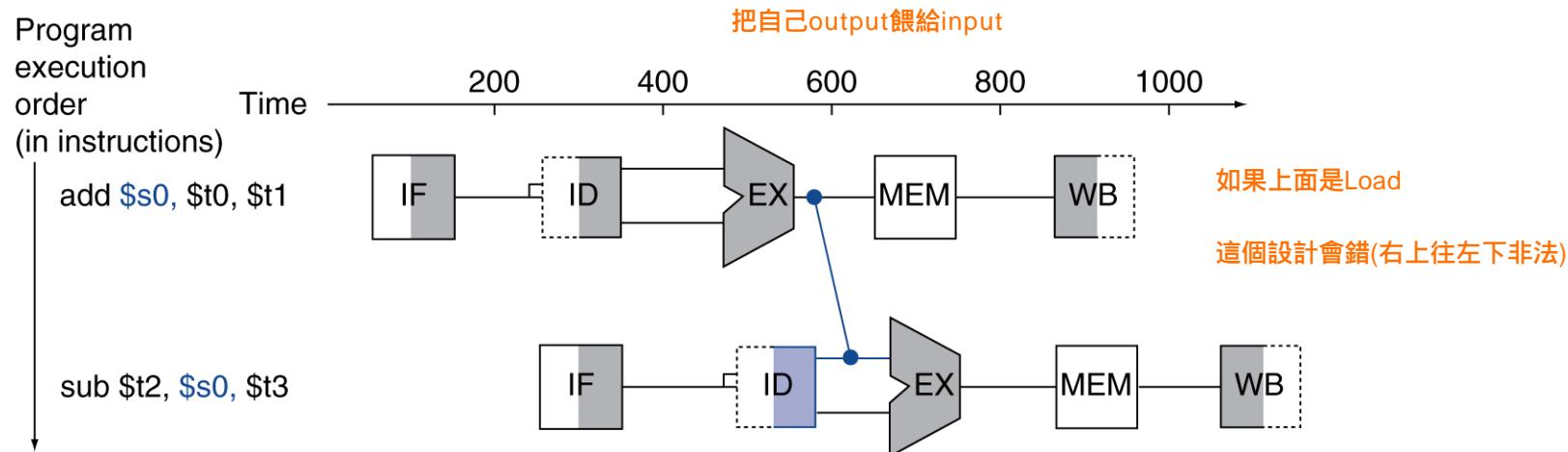
Data Hazards

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



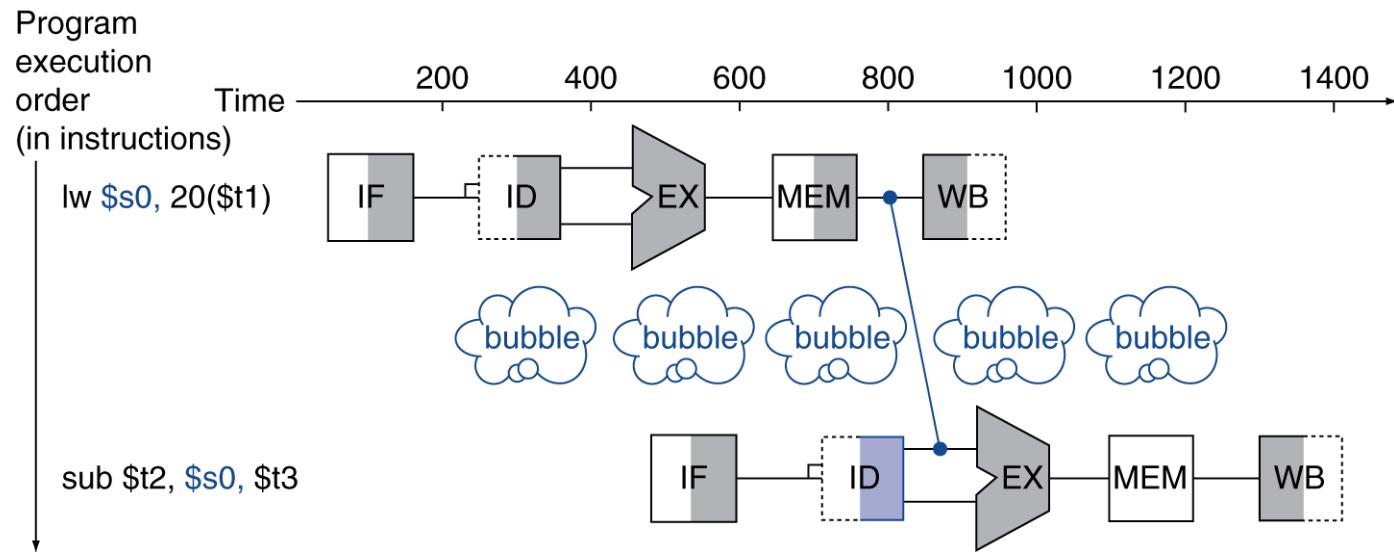
Forwarding (aka Bypassing)

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



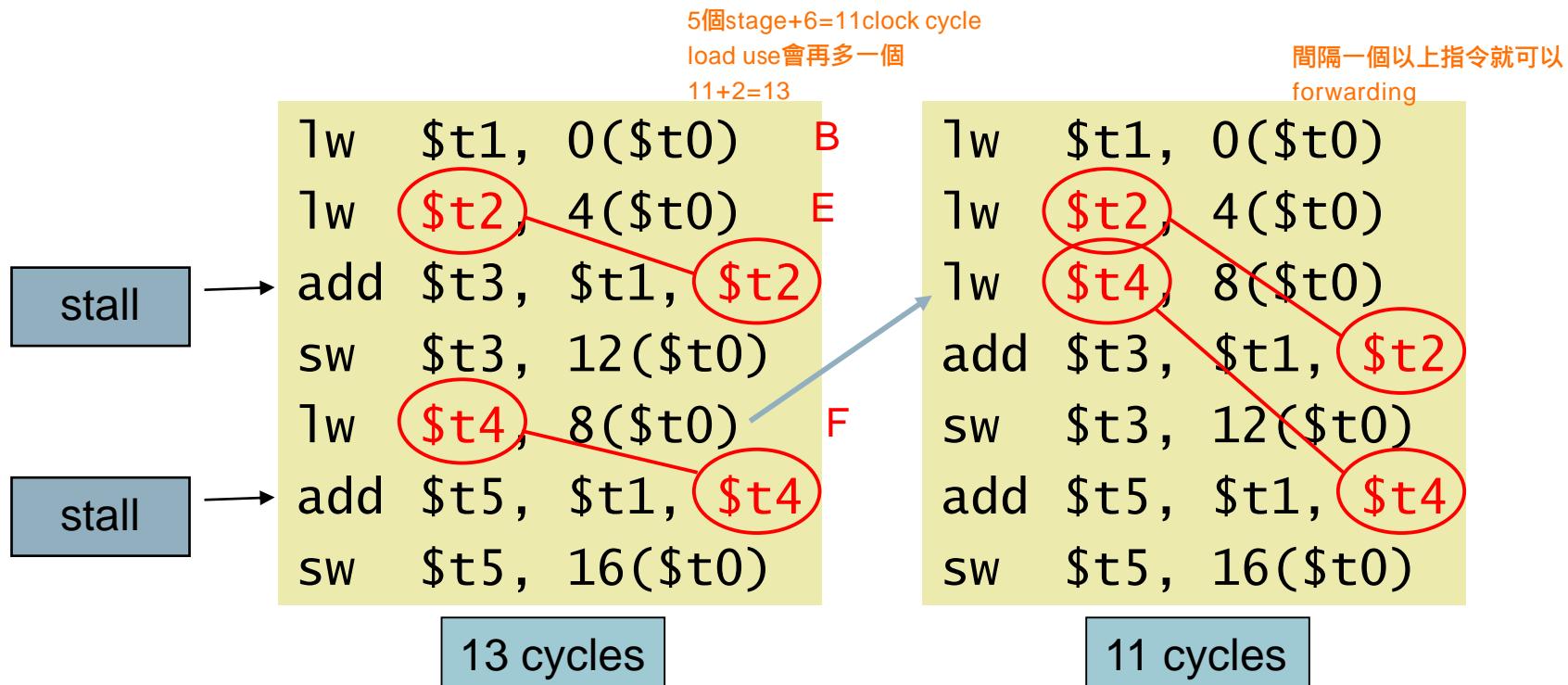
Load-Use Data Hazard

- 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 $A = B + E; C = B + F;$

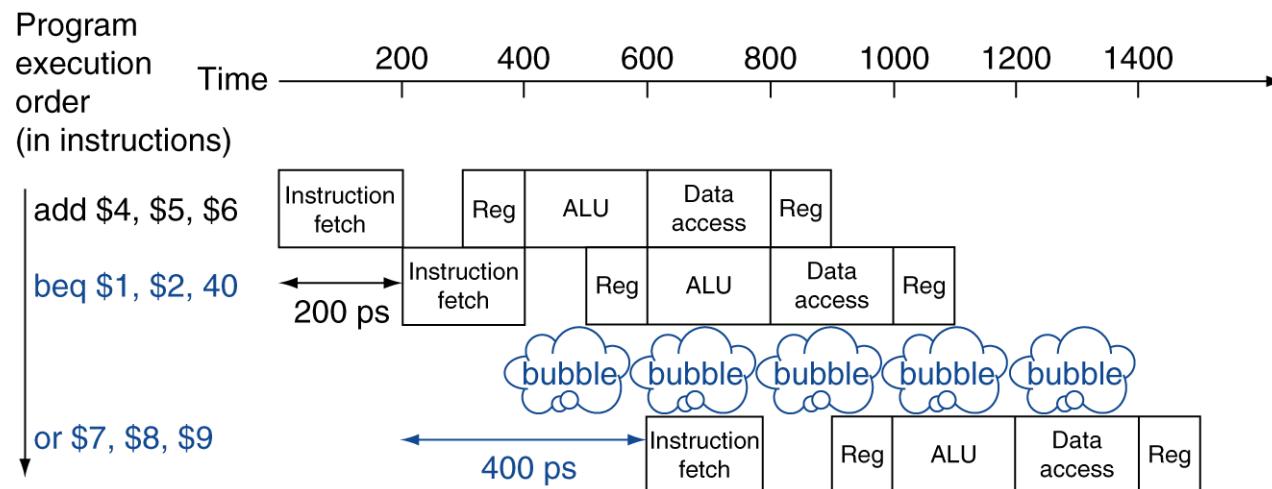


Control Hazards

- 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 comparison in ID stage

Stall on Branch

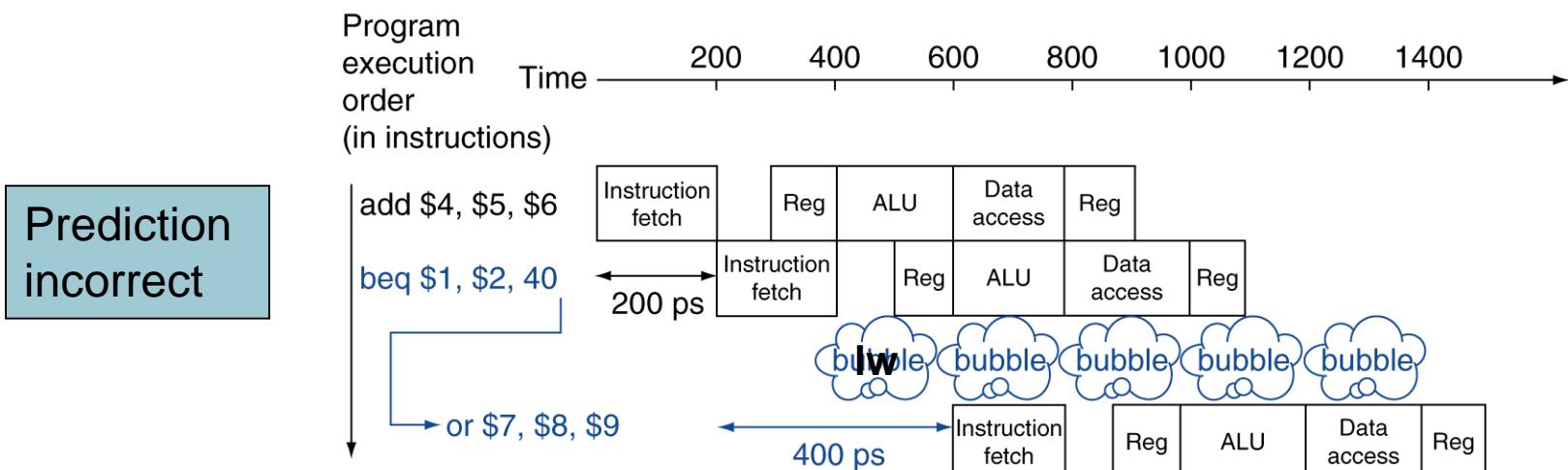
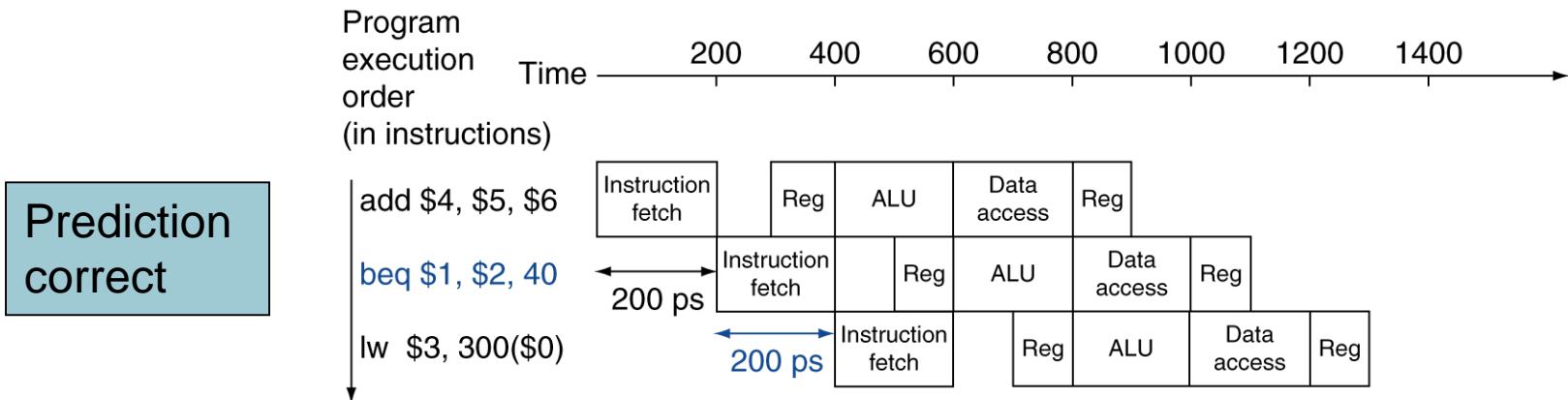
- Wait until branch outcome determined before fetching next instruction



Branch Prediction

- Longer pipelines can't readily determine branch outcome early
 - Stall penalty becomes unacceptable
- Predict outcome of branch 猜錯不會變差
 - Only stall if prediction is wrong
- In MIPS pipeline
 - Can predict branches not taken
 - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken



More-Realistic Branch Prediction

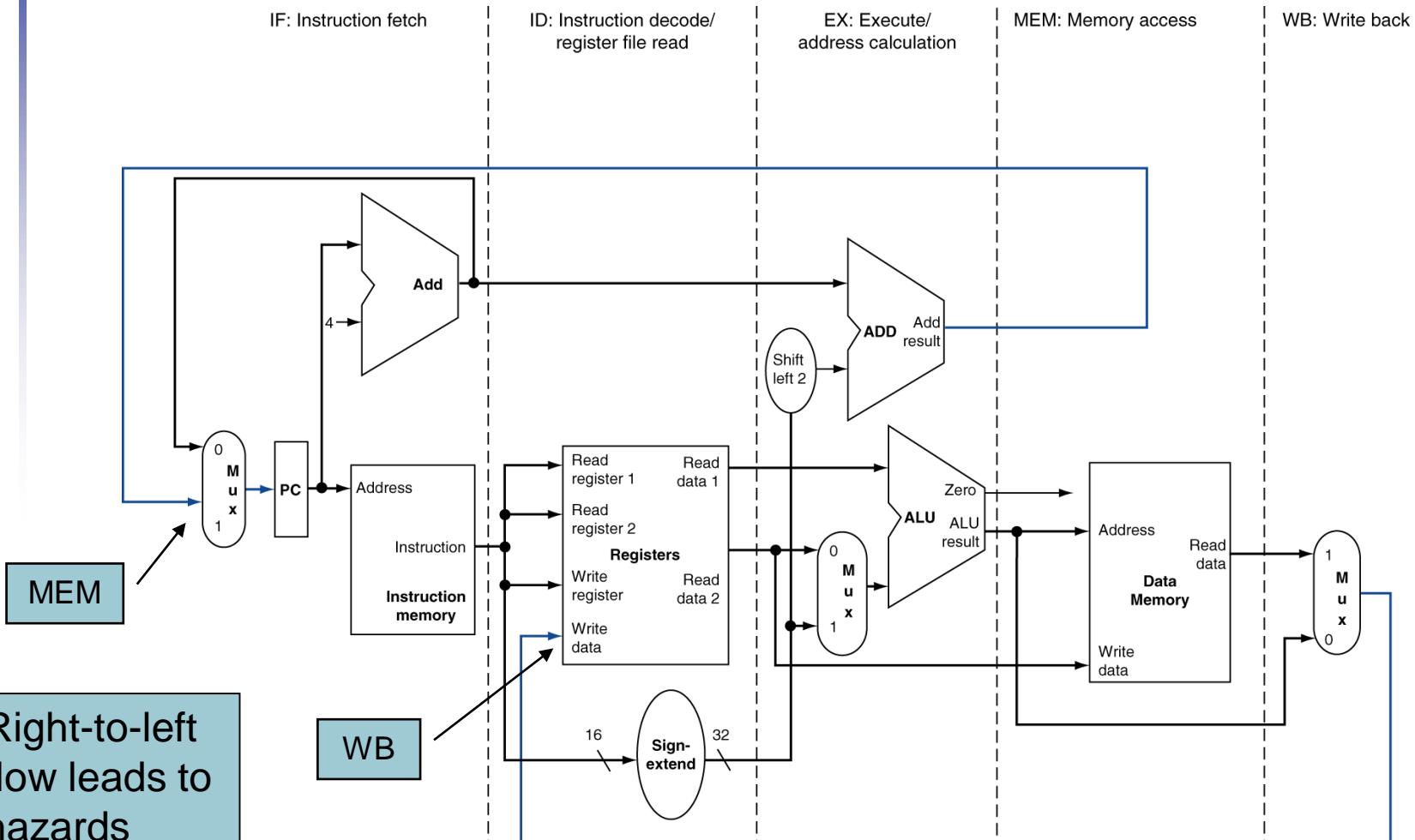
- Static branch prediction
 - Based on typical branch behavior
 - Example: loop and if-statement branches
 - Predict backward branches taken
 - Predict forward branches not taken
- Dynamic branch prediction
 - Hardware measures actual branch behavior
 - e.g., record recent history of each branch
 - Assume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

Pipeline Summary

The BIG Picture

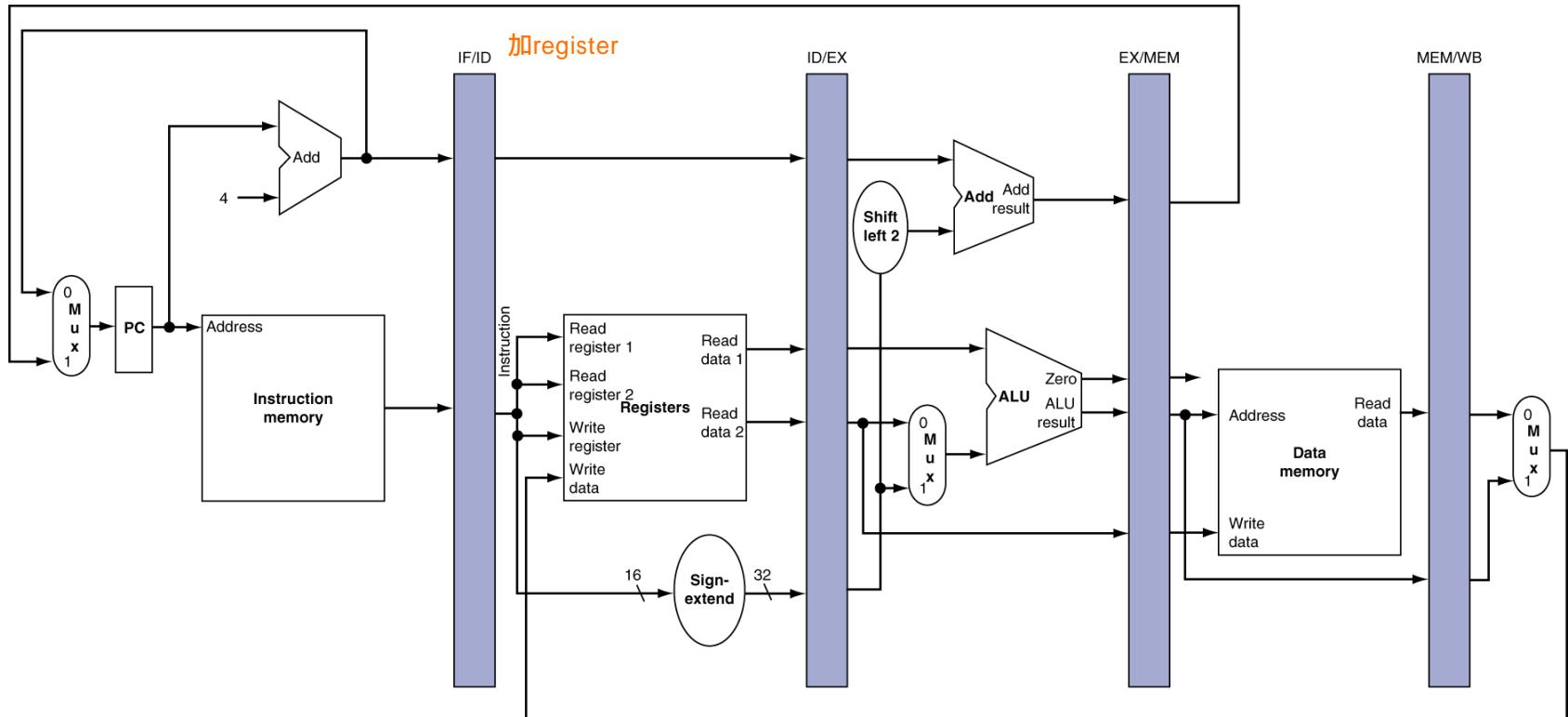
- Pipelining improves performance by increasing instruction throughput
 - Executes multiple instructions in parallel
 - Each instruction has the same latency
- Subject to hazards
 - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

MIPS Pipelined Datapath



Pipeline registers

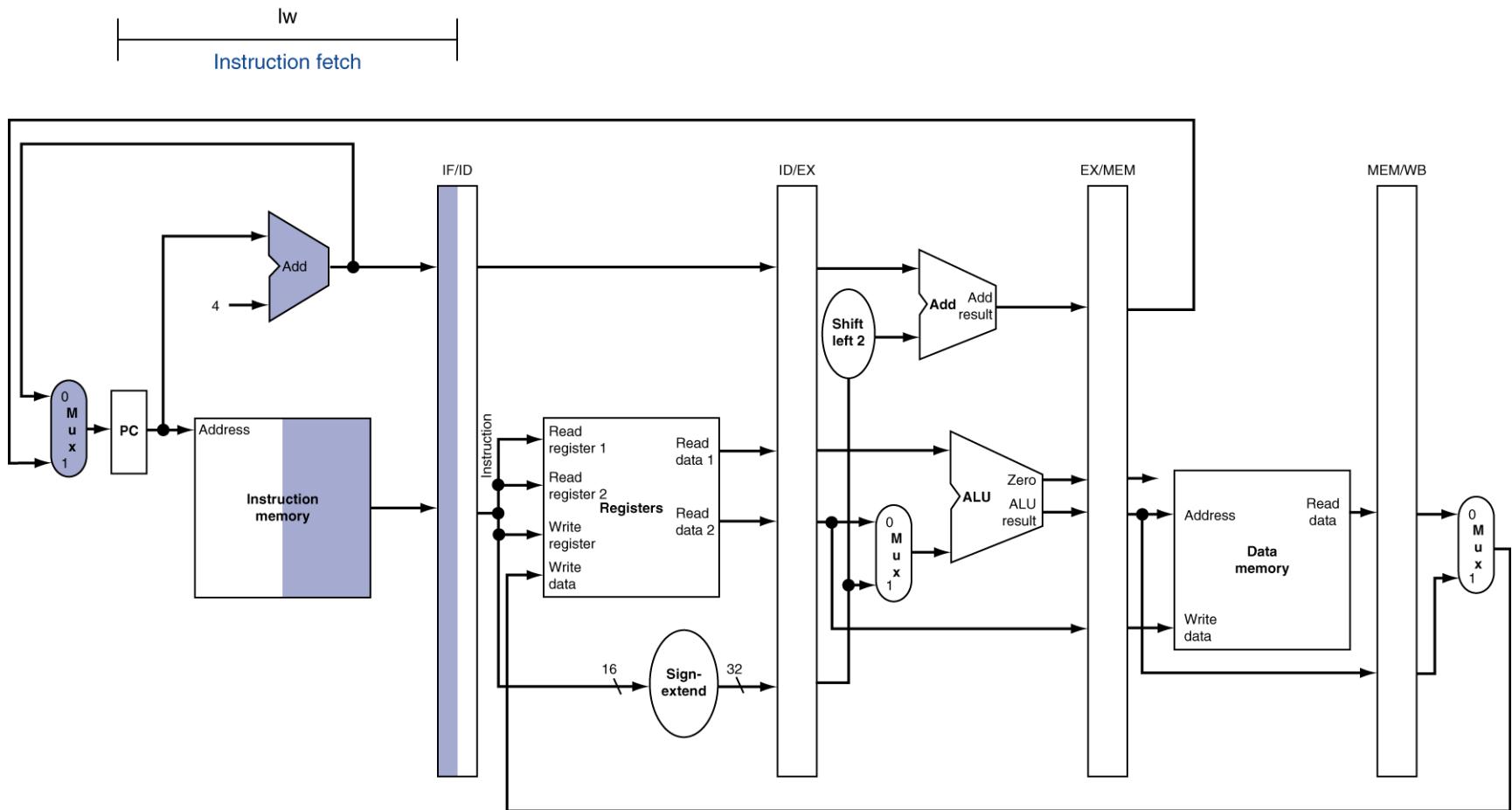
- Need registers between stages
 - To hold information produced in previous cycle



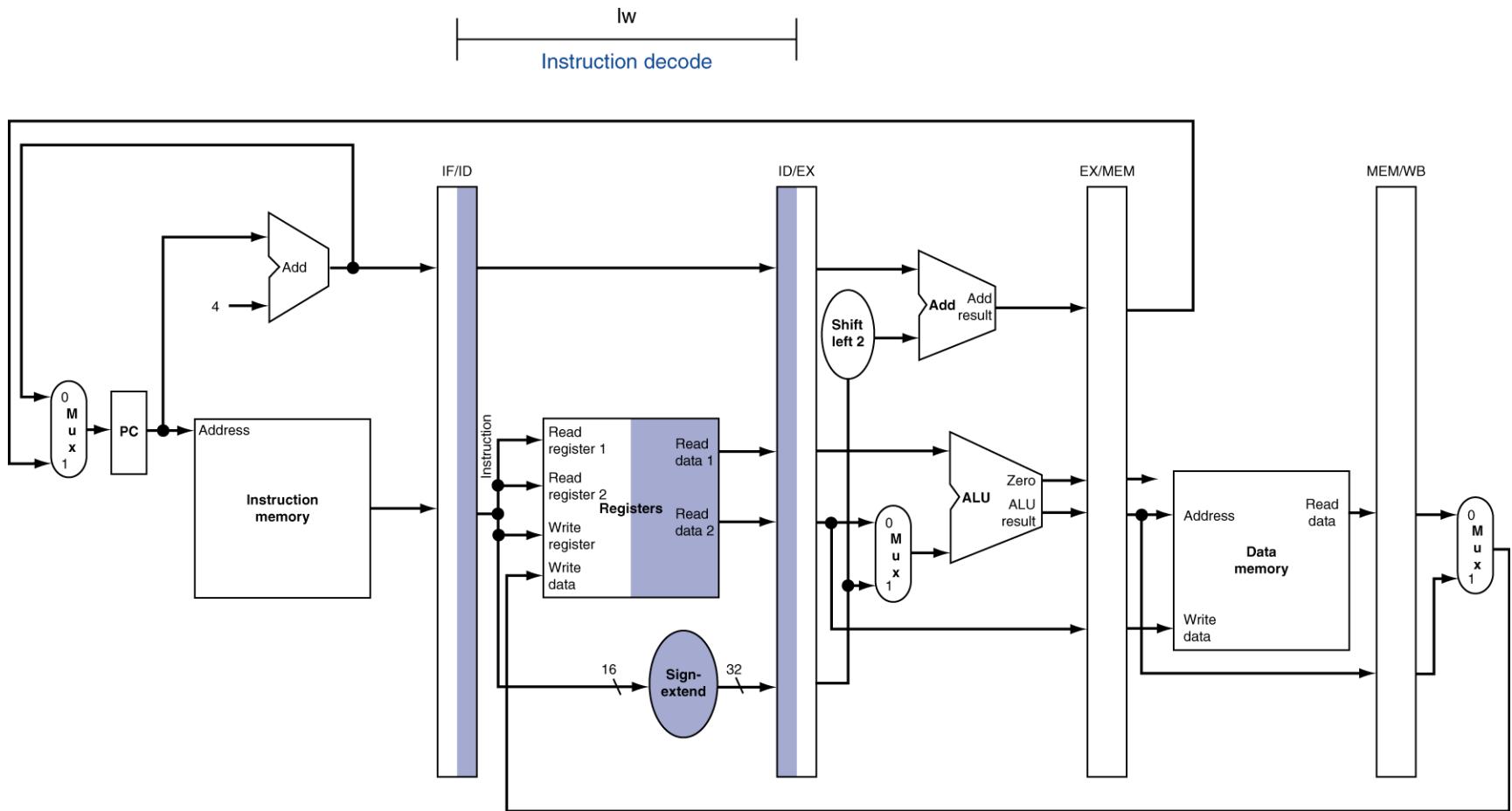
Pipeline Operation

- 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

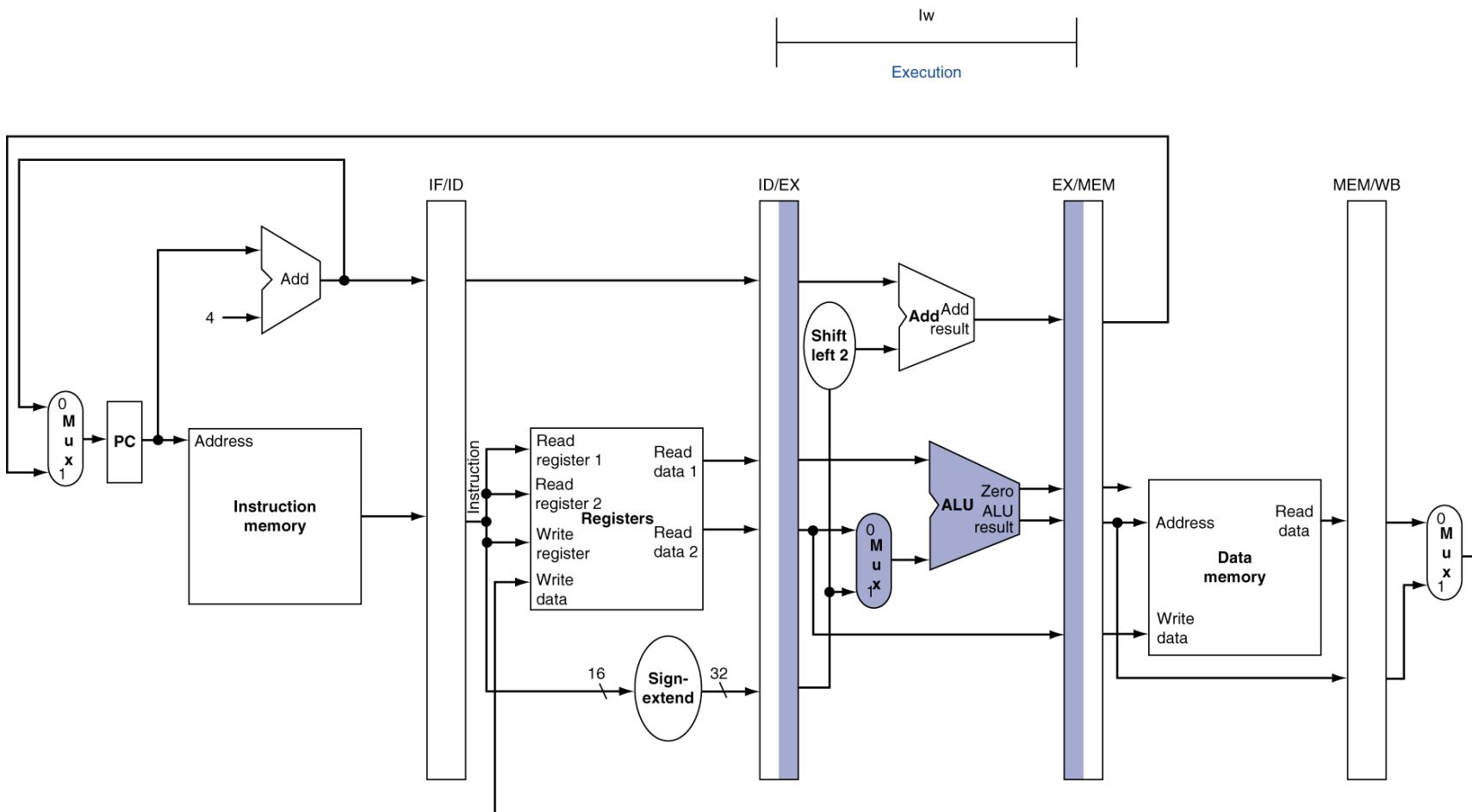
IF for Load, Store, ...



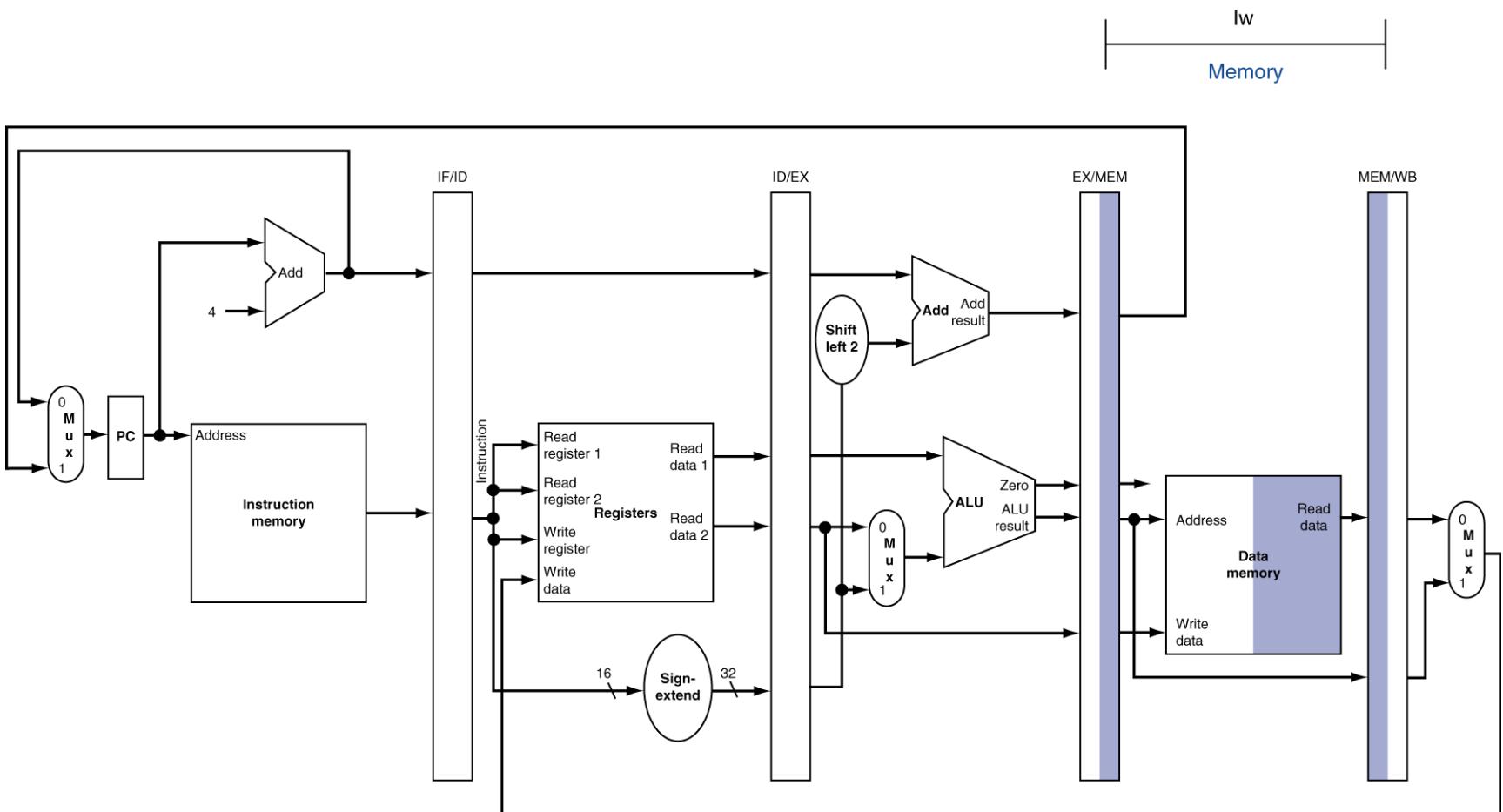
ID for Load, Store, ...



EX for Load

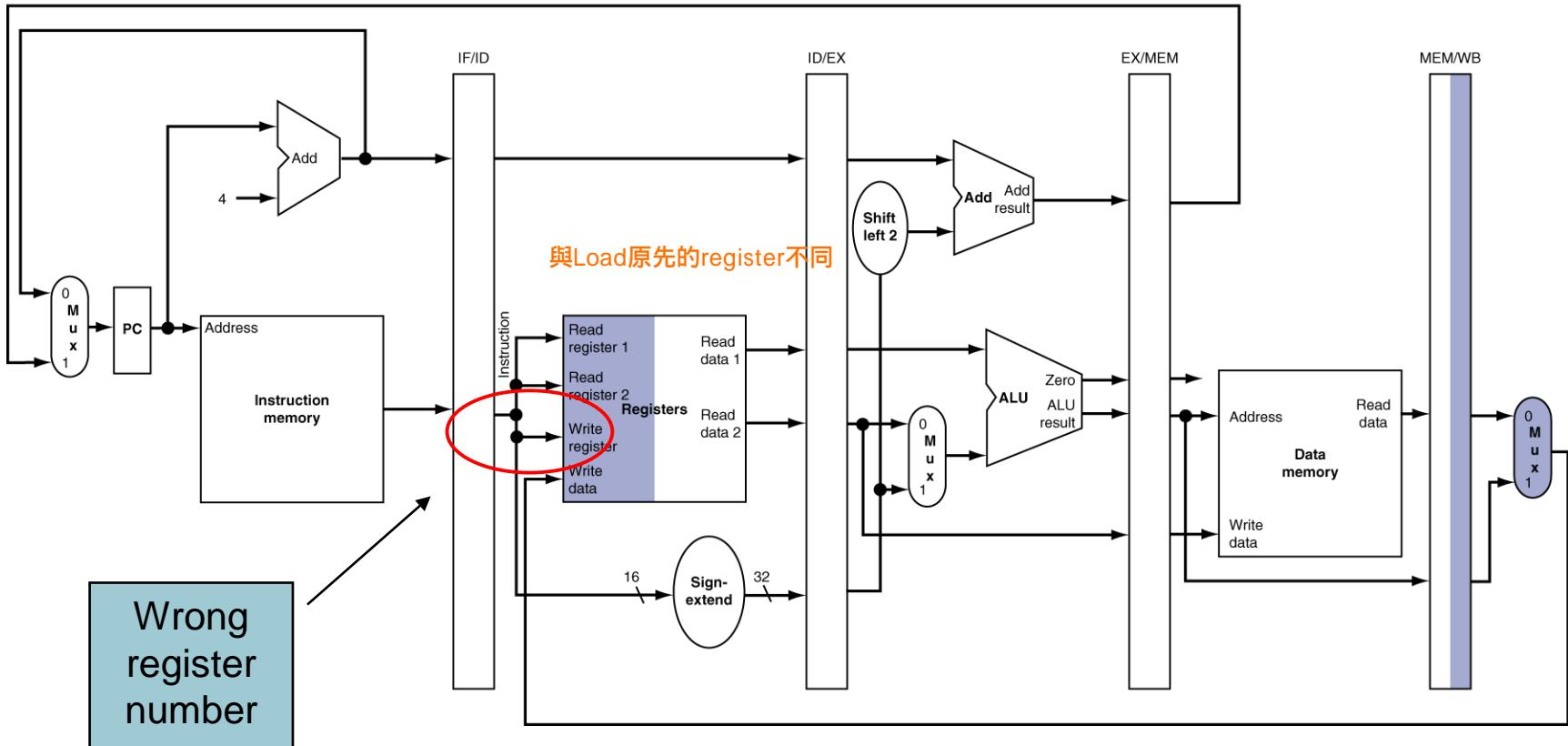


MEM for Load

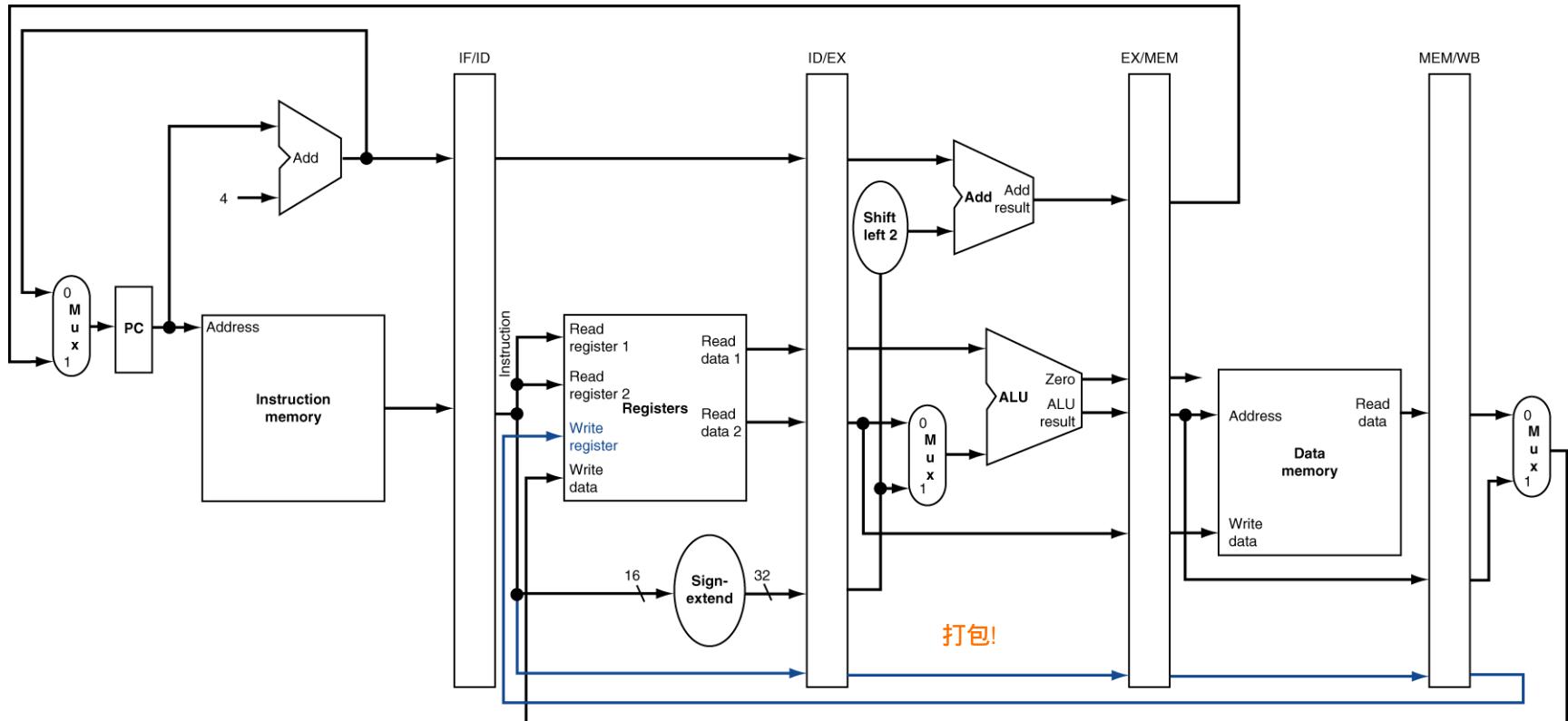


WB for Load

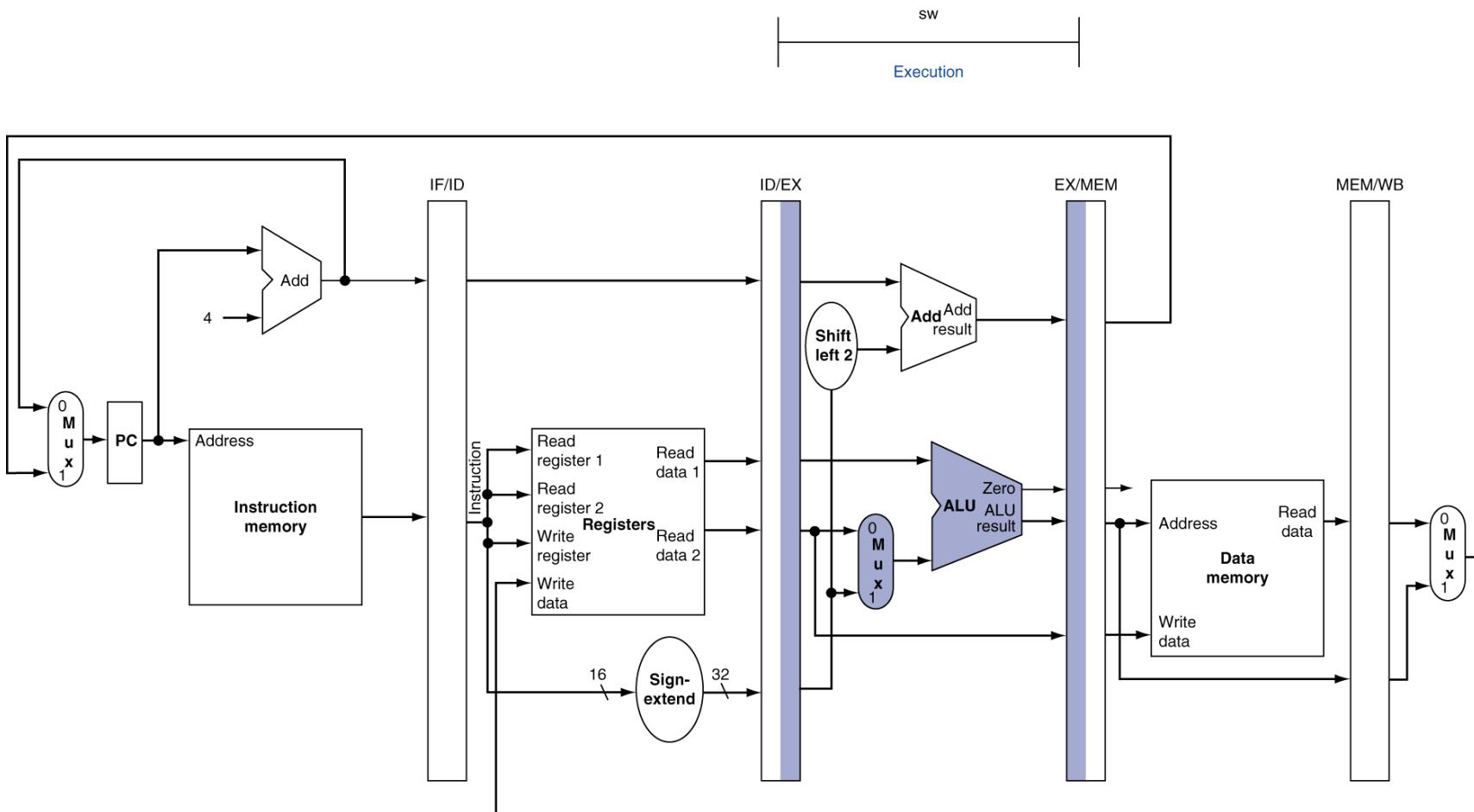
Iw
Write back



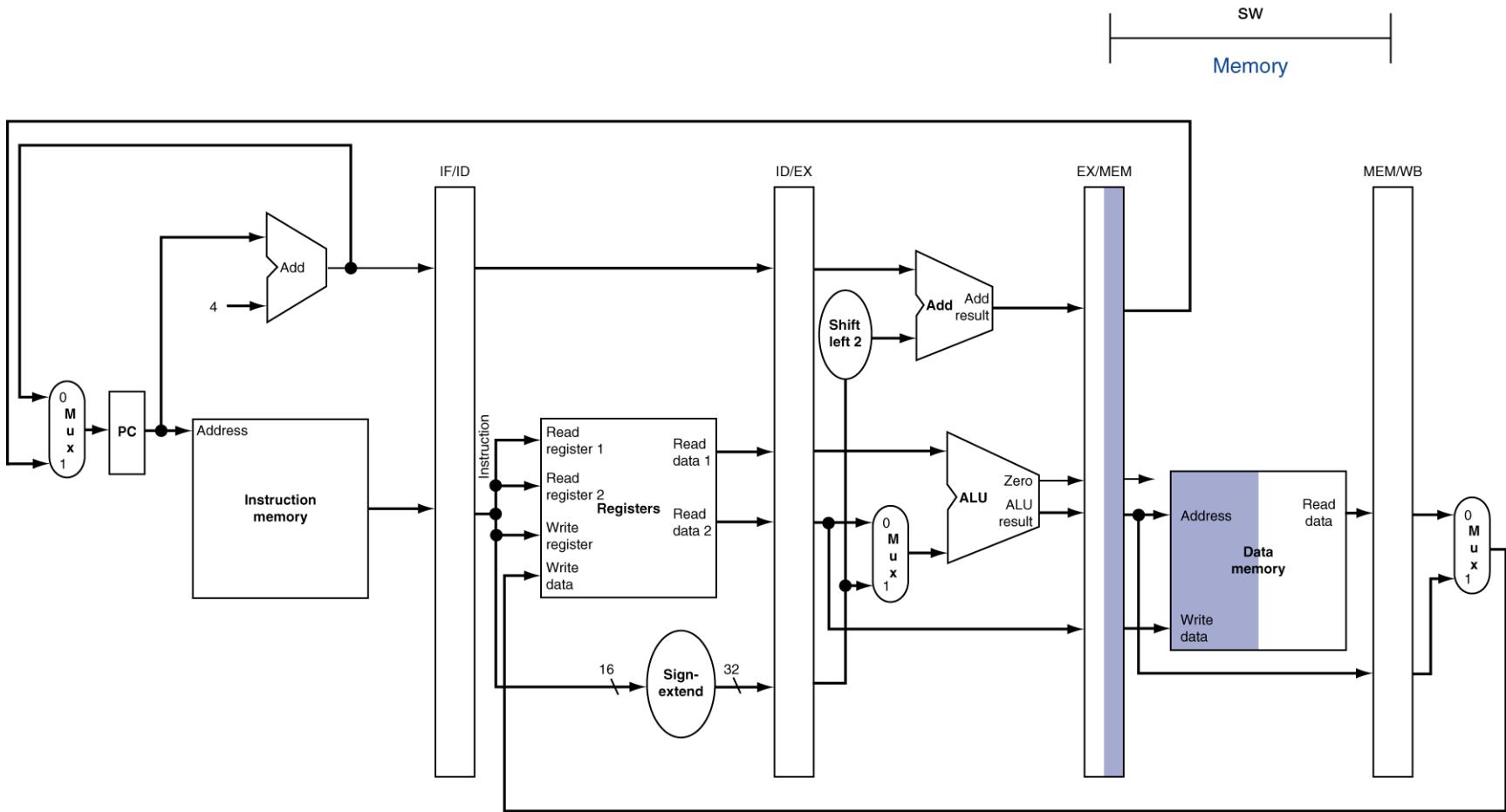
Corrected Datapath for Load



EX for Store

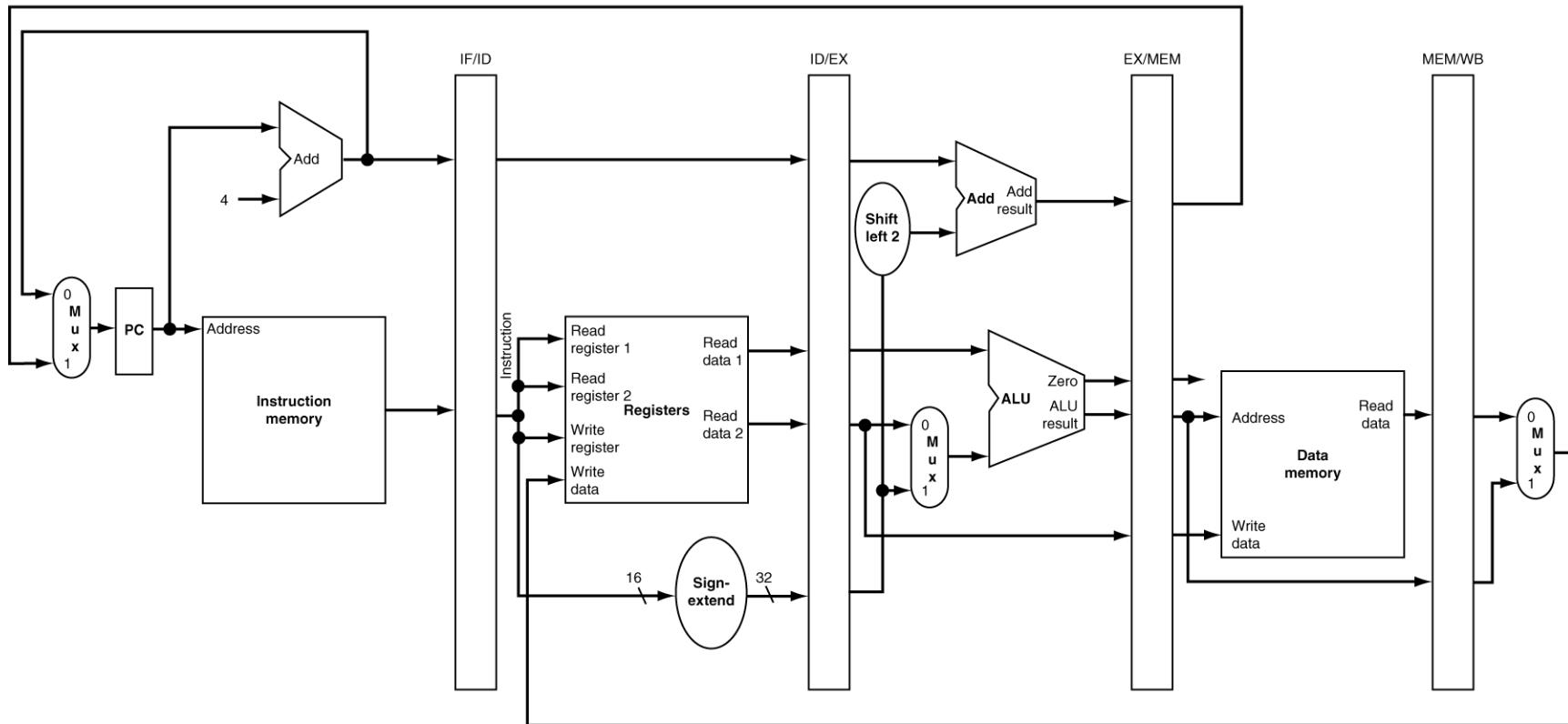


MEM for Store



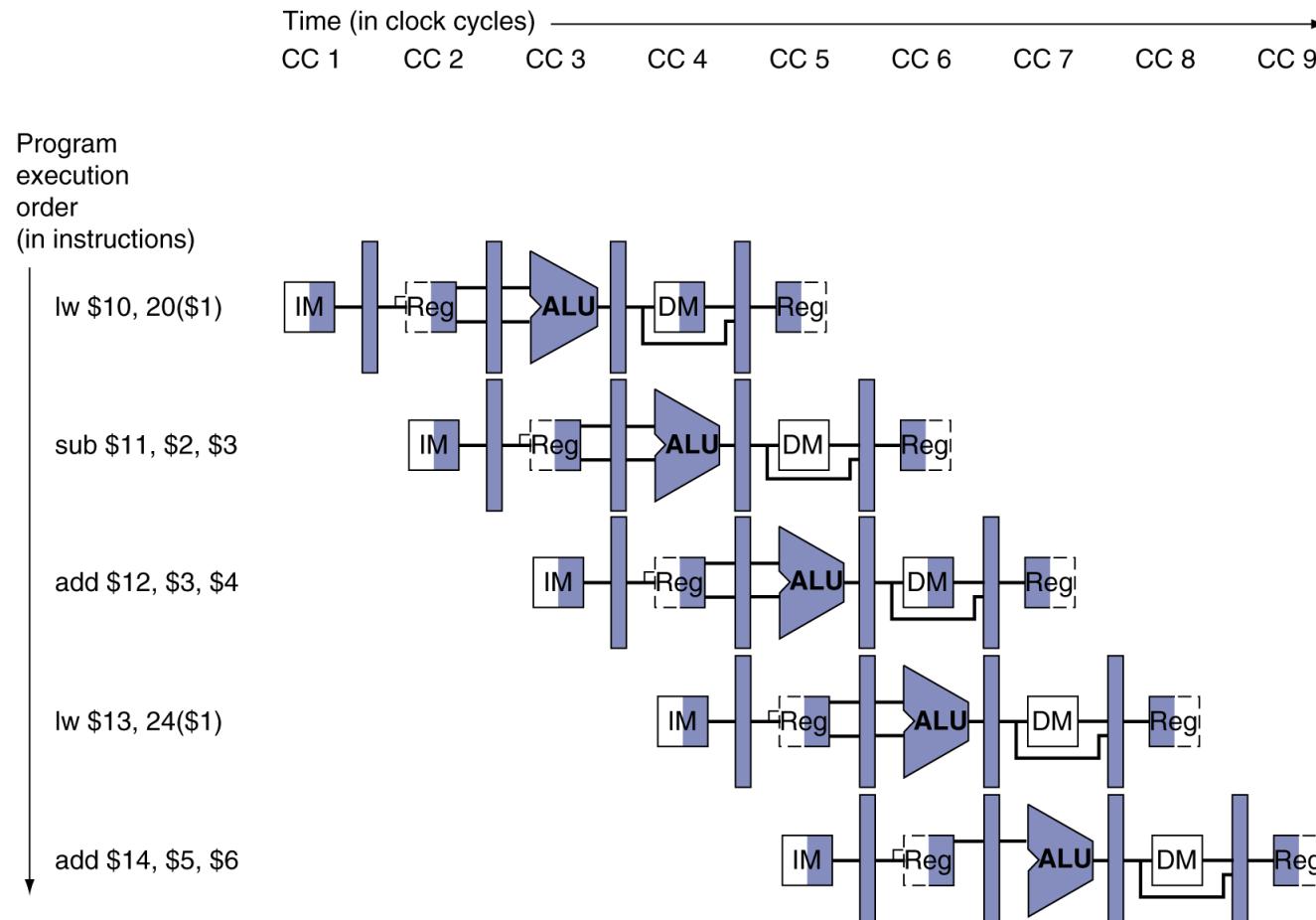
WB for Store

SW
Write-back



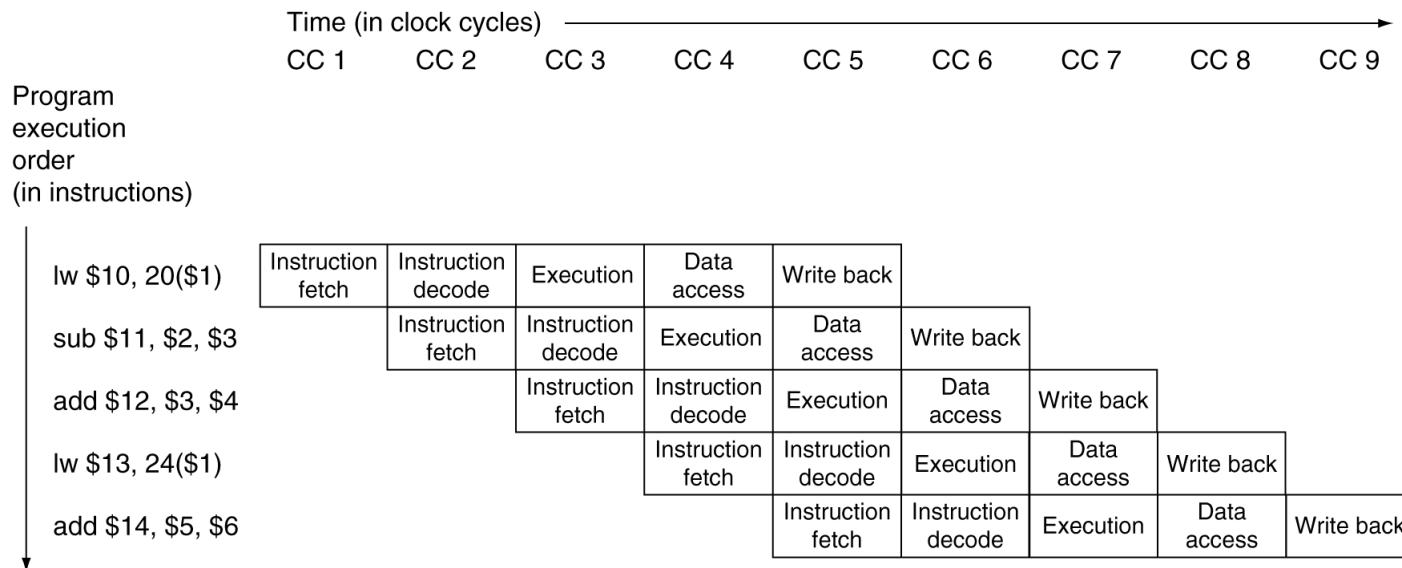
Multi-Cycle Pipeline Diagram

- Form showing resource usage



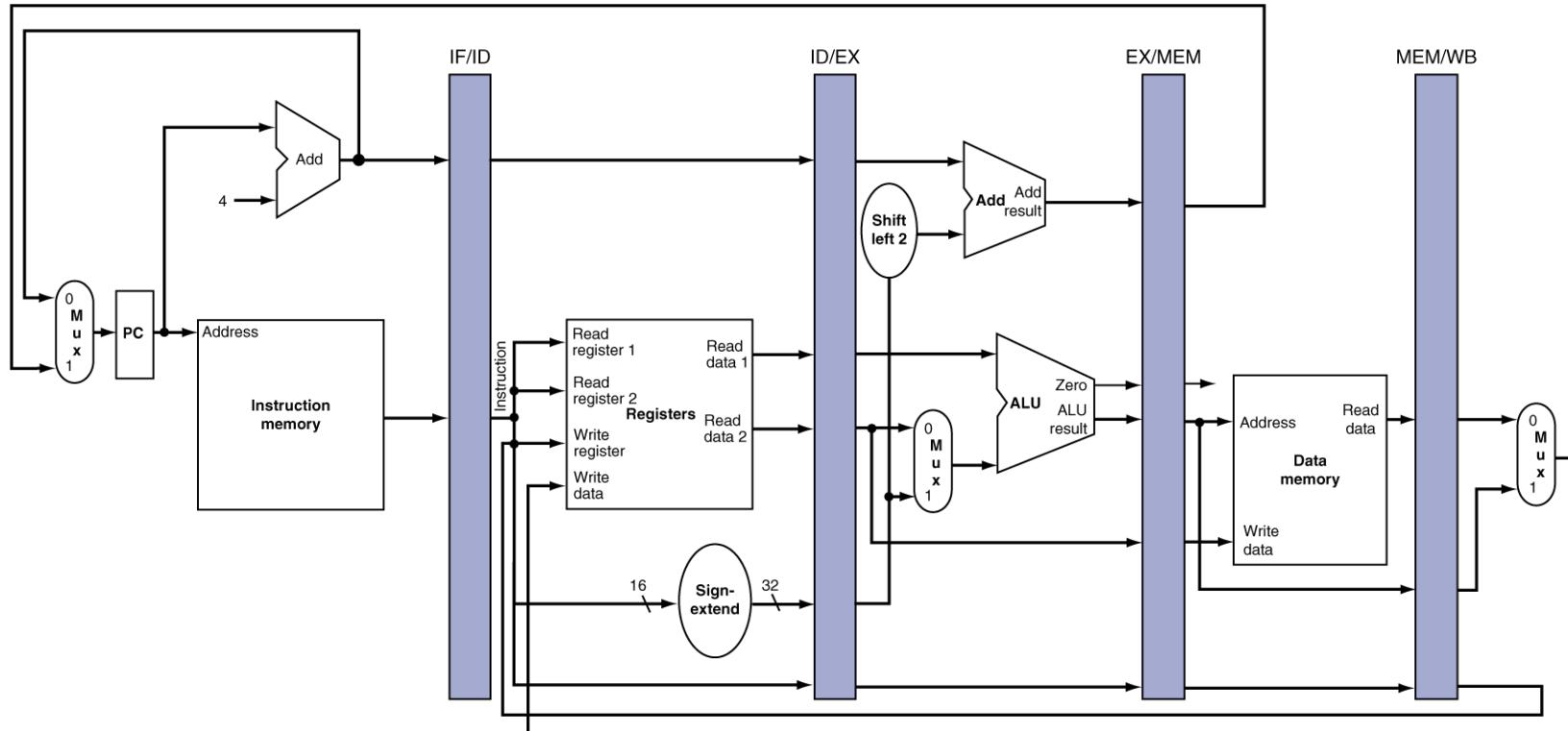
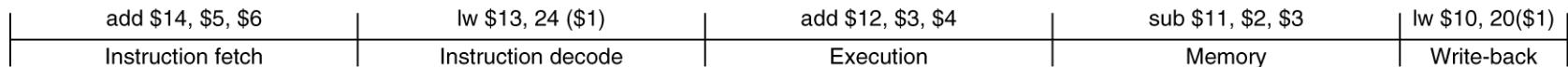
Multi-Cycle Pipeline Diagram

Traditional form

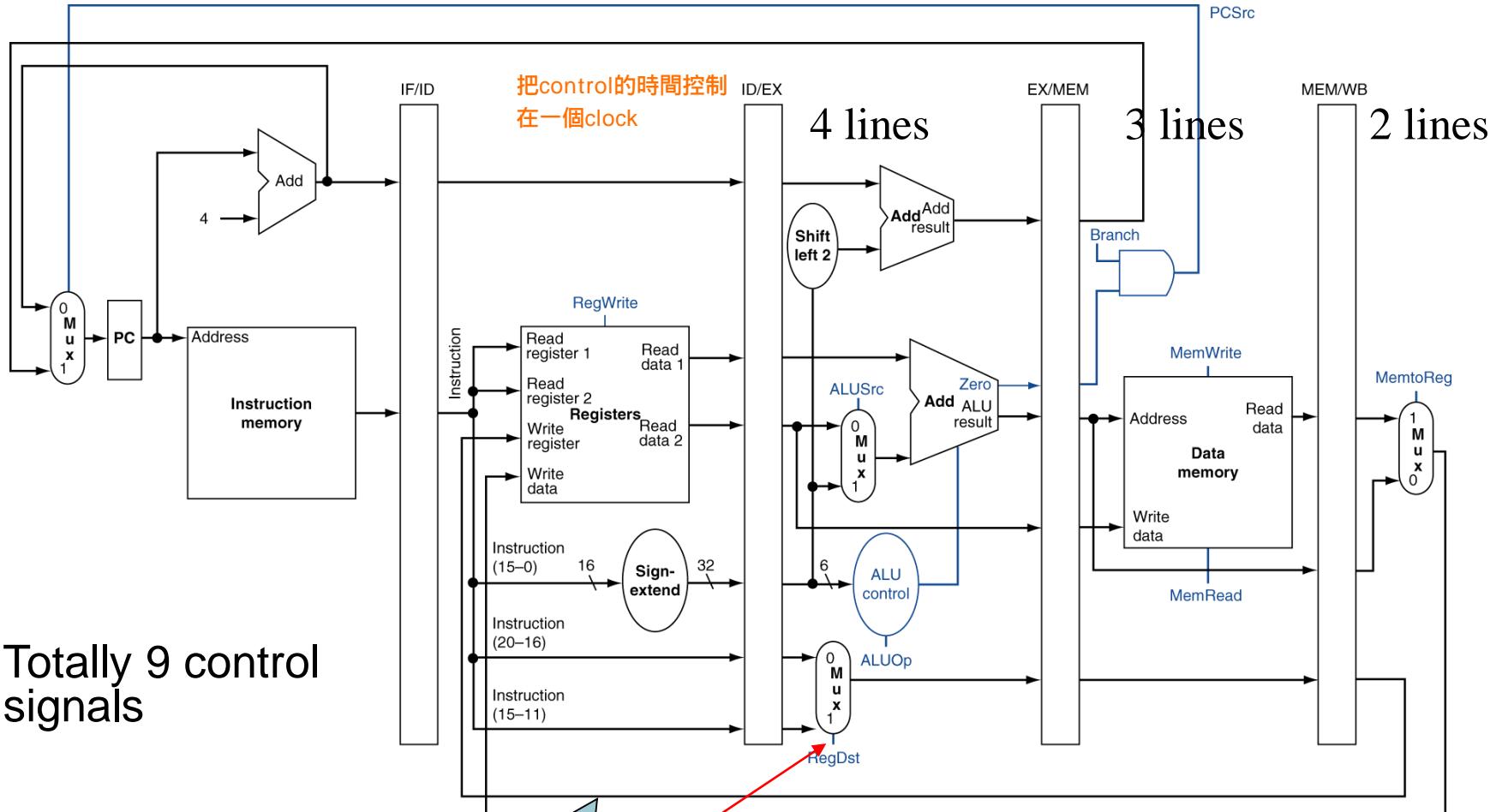


Single-Cycle Pipeline Diagram

State of pipeline in a given cycle



Pipelined Control (Simplified)



Pipelining Control Values

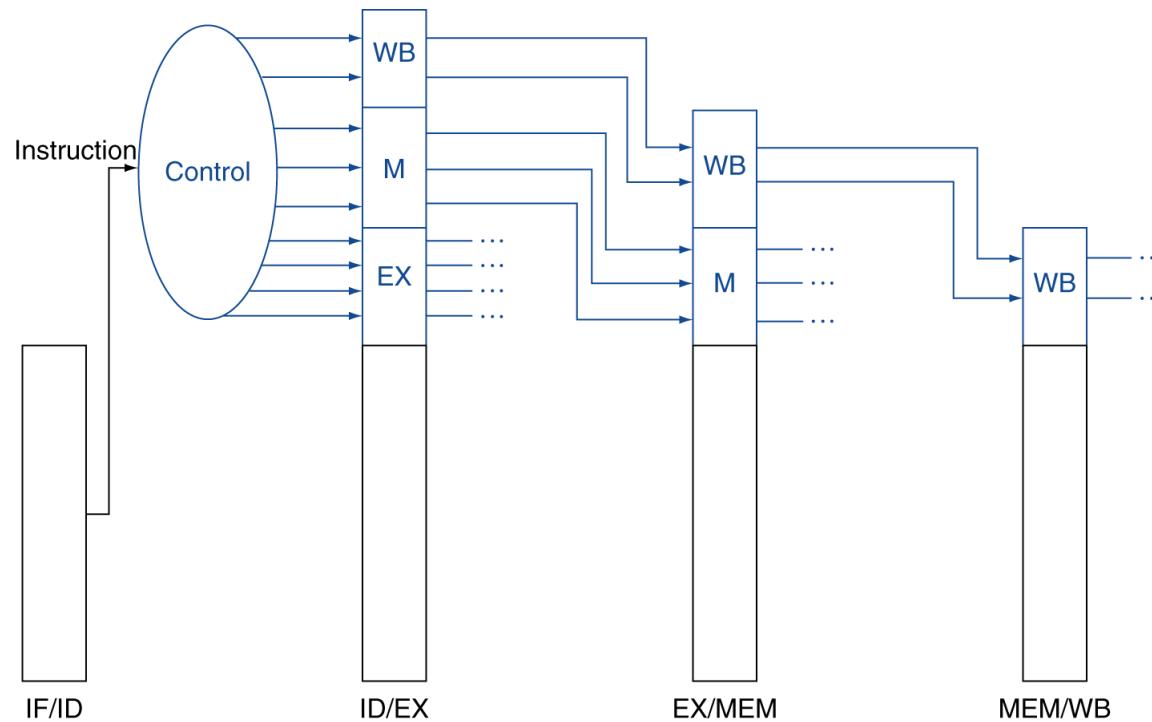
Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	Load word	XXXXXX	Add	0010
SW	00	Store word	XXXXXX	Add	0010
Branch equal	01	Branch equal	XXXXXX	Subtract	0110
R-type	10	Add	100000	Add	0010
R-type	10	Subtract	100010	Subtract	0110
R-type	10	AND	100100	And	0000
R-type	10	OR	100101	Or	0001
R-type	10	Slt	101010	Set less than	0111

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	x

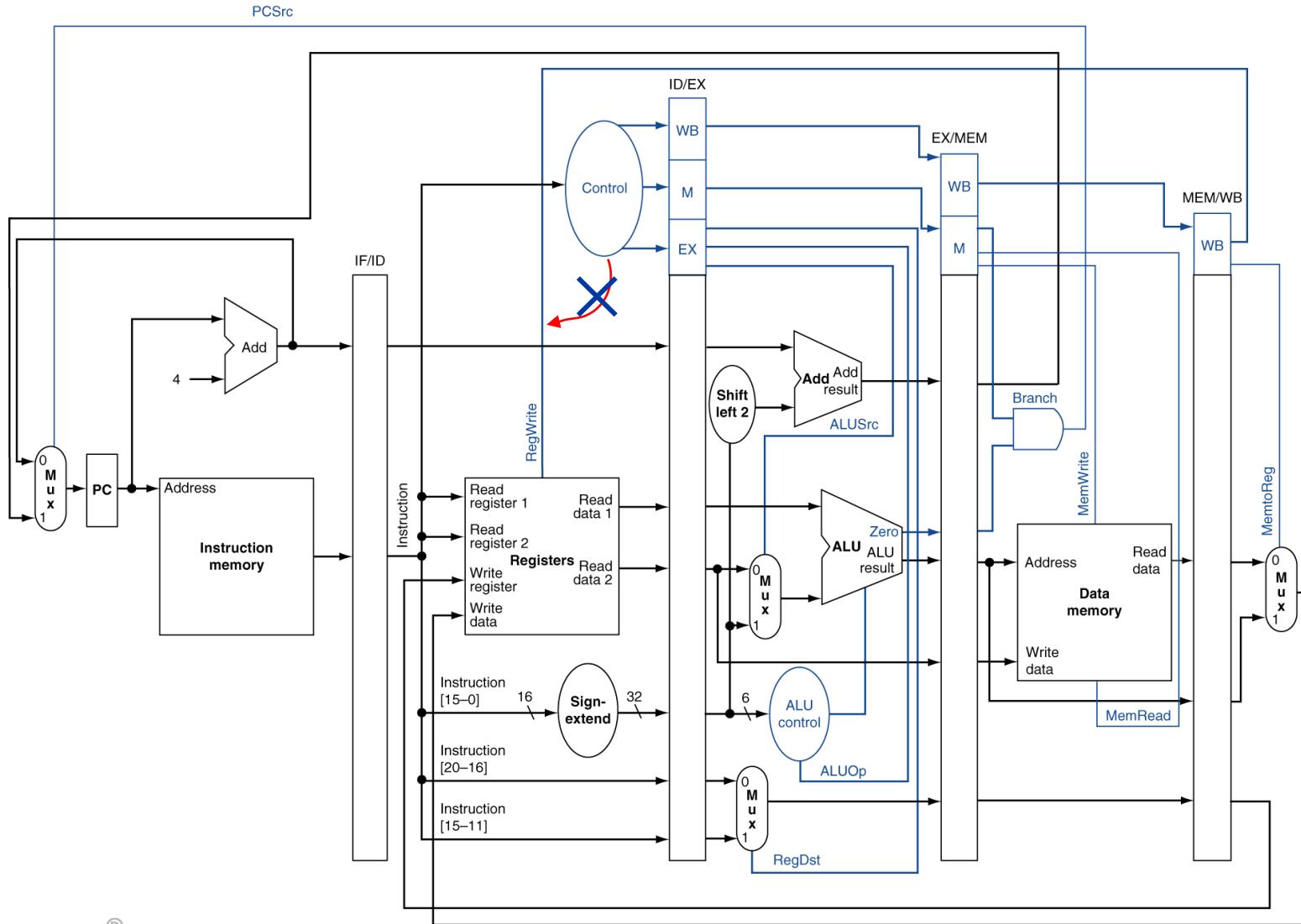


Pipelined Control

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



Pipelined Control



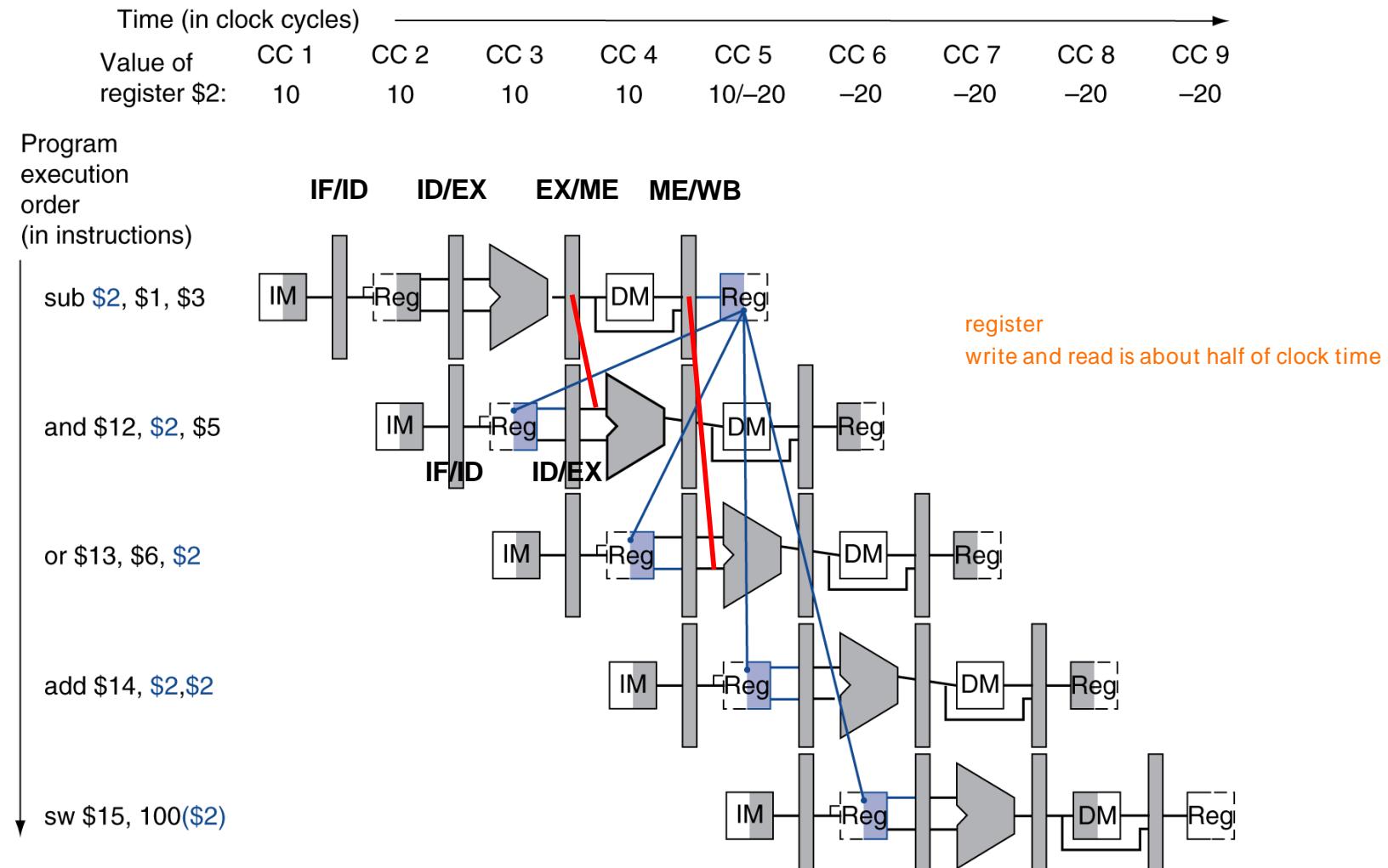
Data Hazards in ALU Instructions

- Consider this sequence:

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

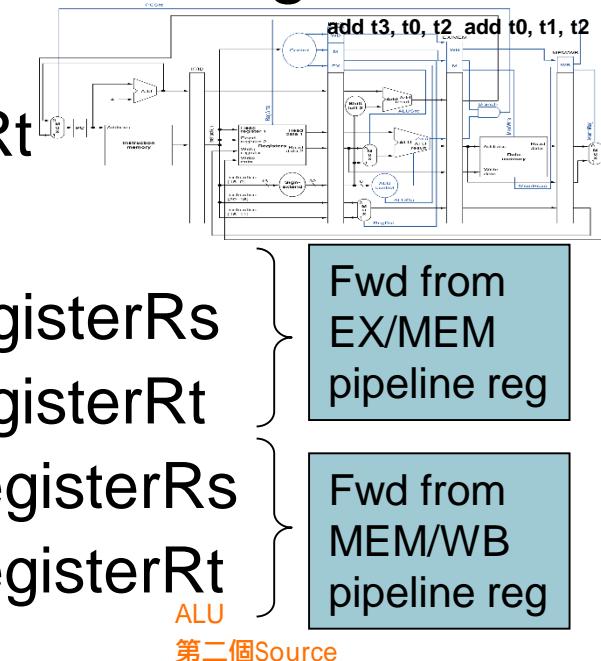
- We can resolve hazards with forwarding
 - How do we detect when to forward?

Dependencies & Forwarding



Detecting the Need to Forward

- Pass register numbers along pipeline
 - e.g., ID/EX.RegisterRs = 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
 - EX/MEM.RegisterRd = ID/EX.RegisterRs
 - EX/MEM.RegisterRd = ID/EX.RegisterRt
 - MEM/WB.RegisterRd = ID/EX.RegisterRs
 - MEM/WB.RegisterRd = ID/EX.RegisterRt

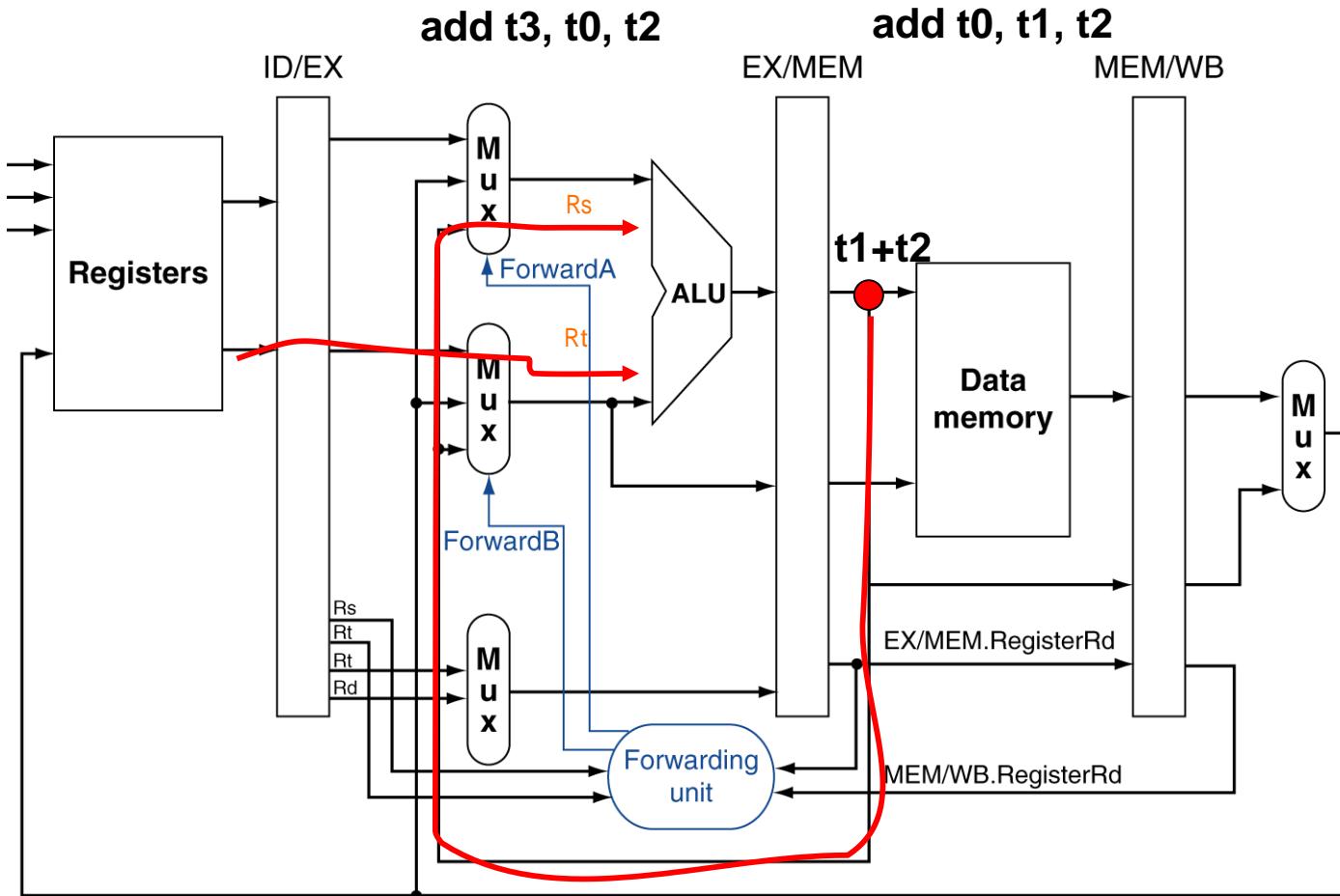


Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
 - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not \$zero
 - EX/MEM.RegisterRd $\neq 0$,
MEM/WB.RegisterRd $\neq 0$

必須確定第一個指令是write

Forwarding Paths



b. With forwarding

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



Forwarding Conditions

- EX hazard
 - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
ForwardA = 10
 - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
ForwardB = 10
- MEM hazard
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01

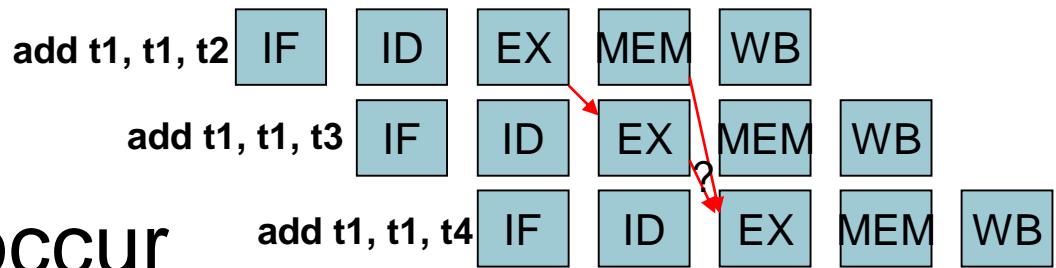
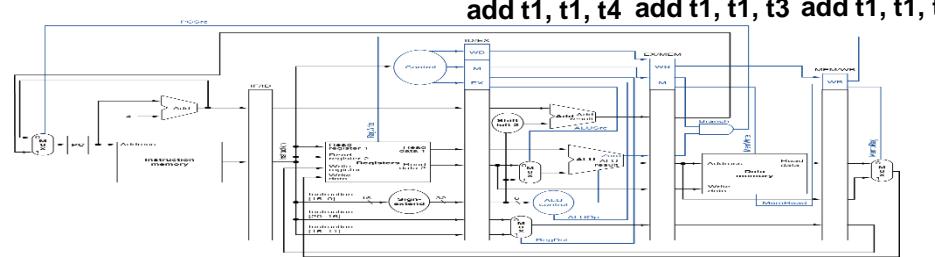
Double Data Hazard

- 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 ≠ 0)
第一個source and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

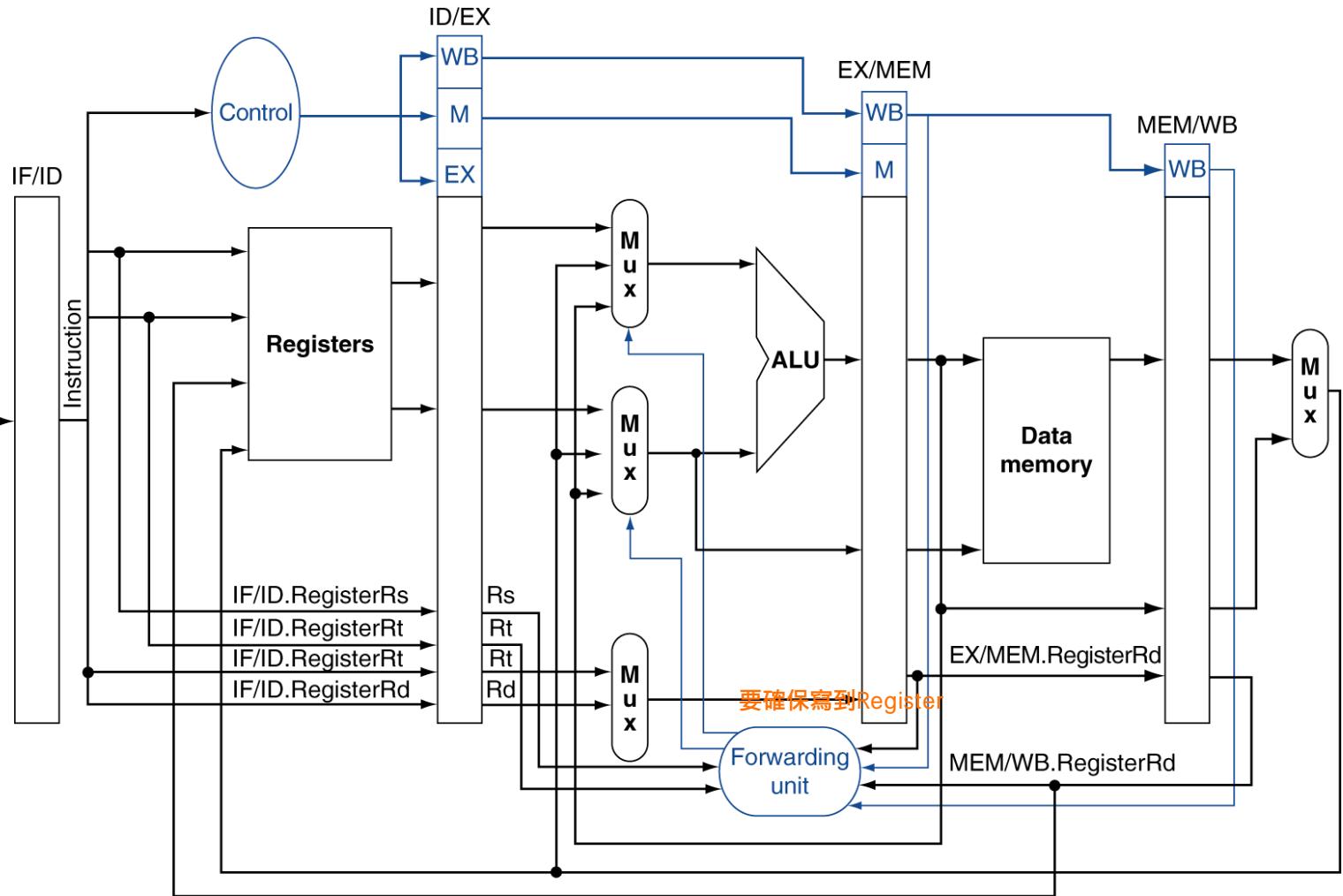
把EX hazard的可能拿掉!這樣就可以變免兩個來源的forwarding!

ForwardA = 01

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
第二個source and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

ForwardB = 01

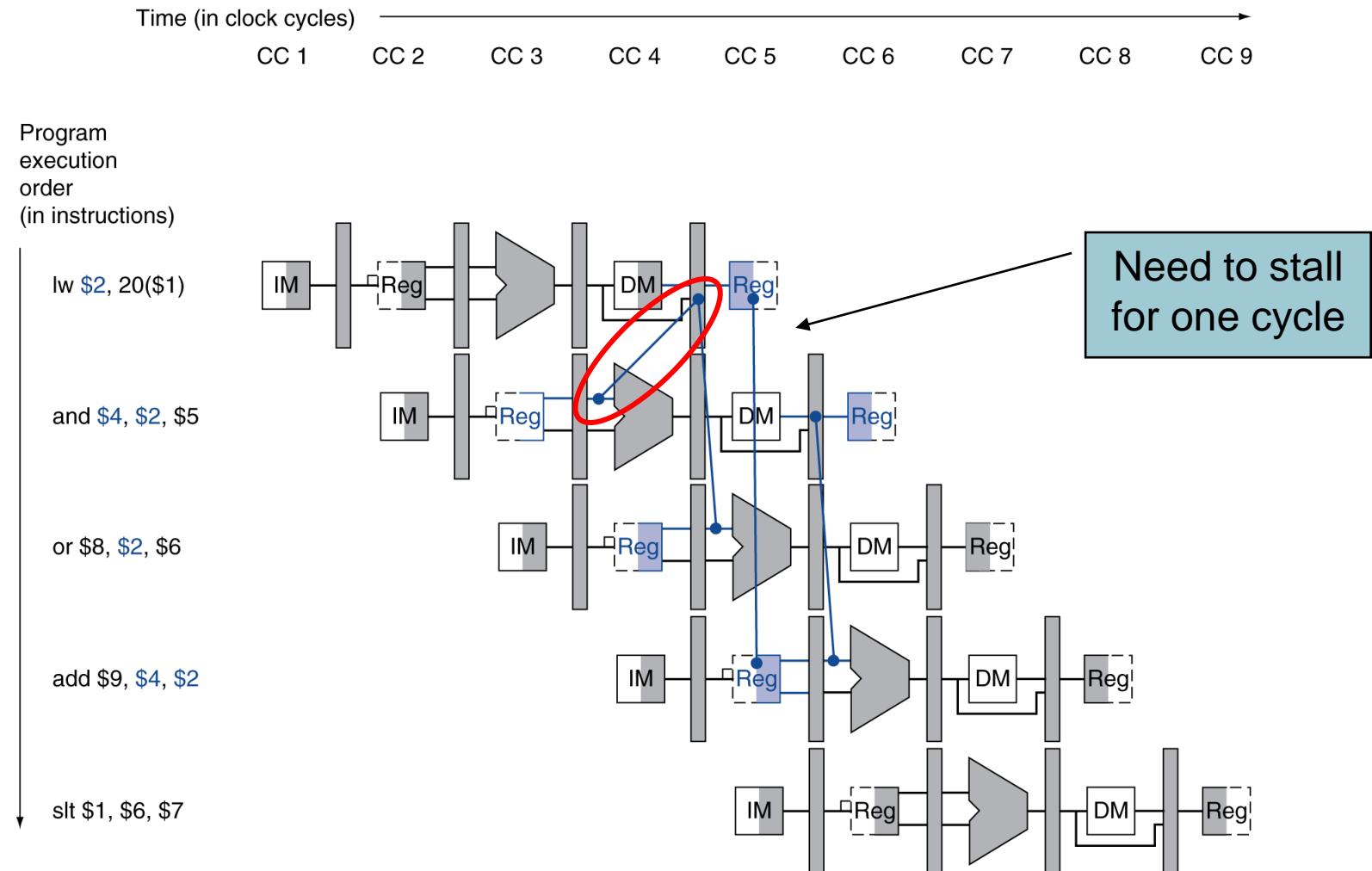
Datapath with Forwarding



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



Load-Use Data Hazard



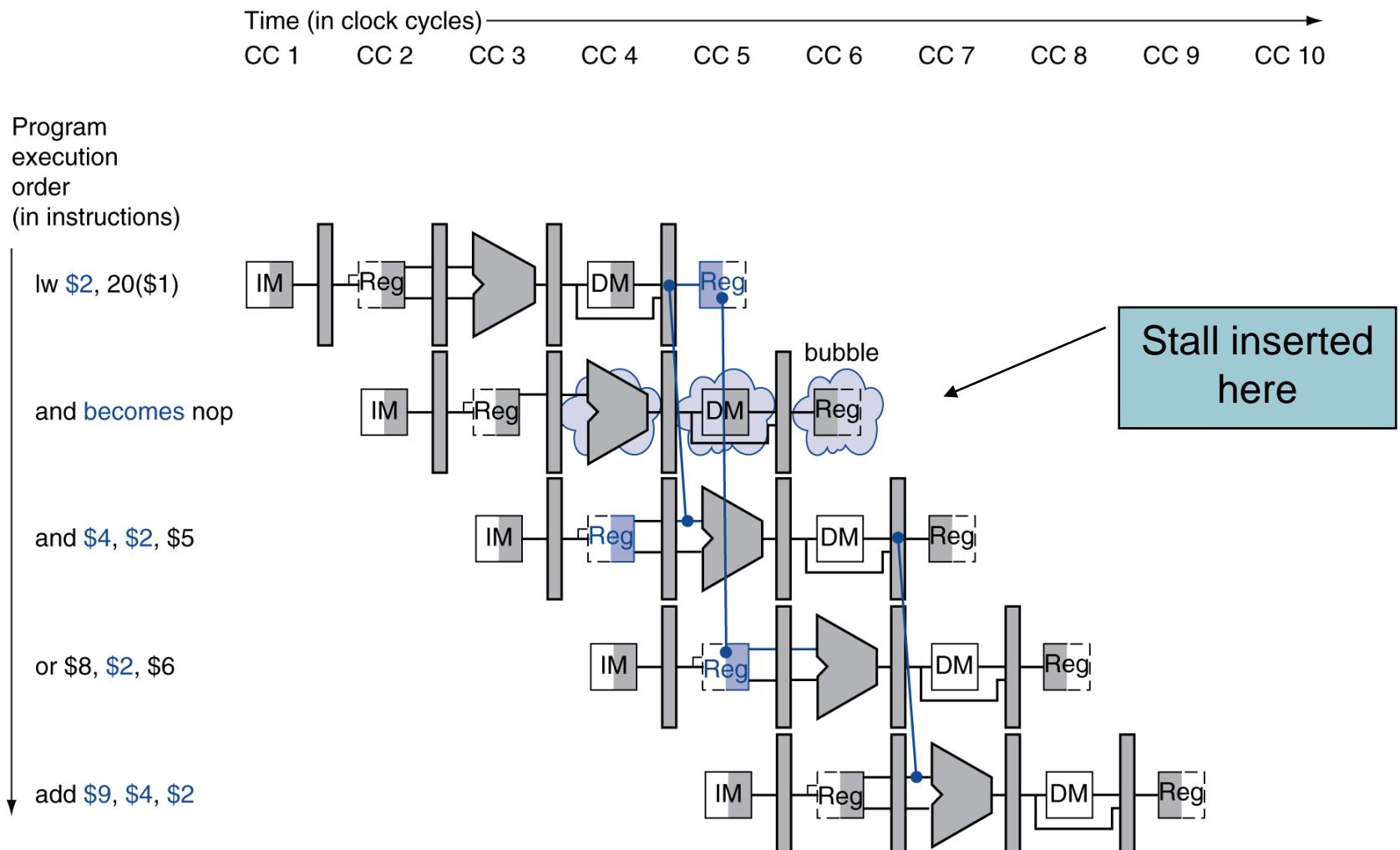
Load-Use Hazard Detection

- 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
$$((ID/EX.RegisterRt = IF/ID.RegisterRs) \text{ or } (ID/EX.RegisterRt = IF/ID.RegisterRt))$$
- If detected, stall and insert bubble

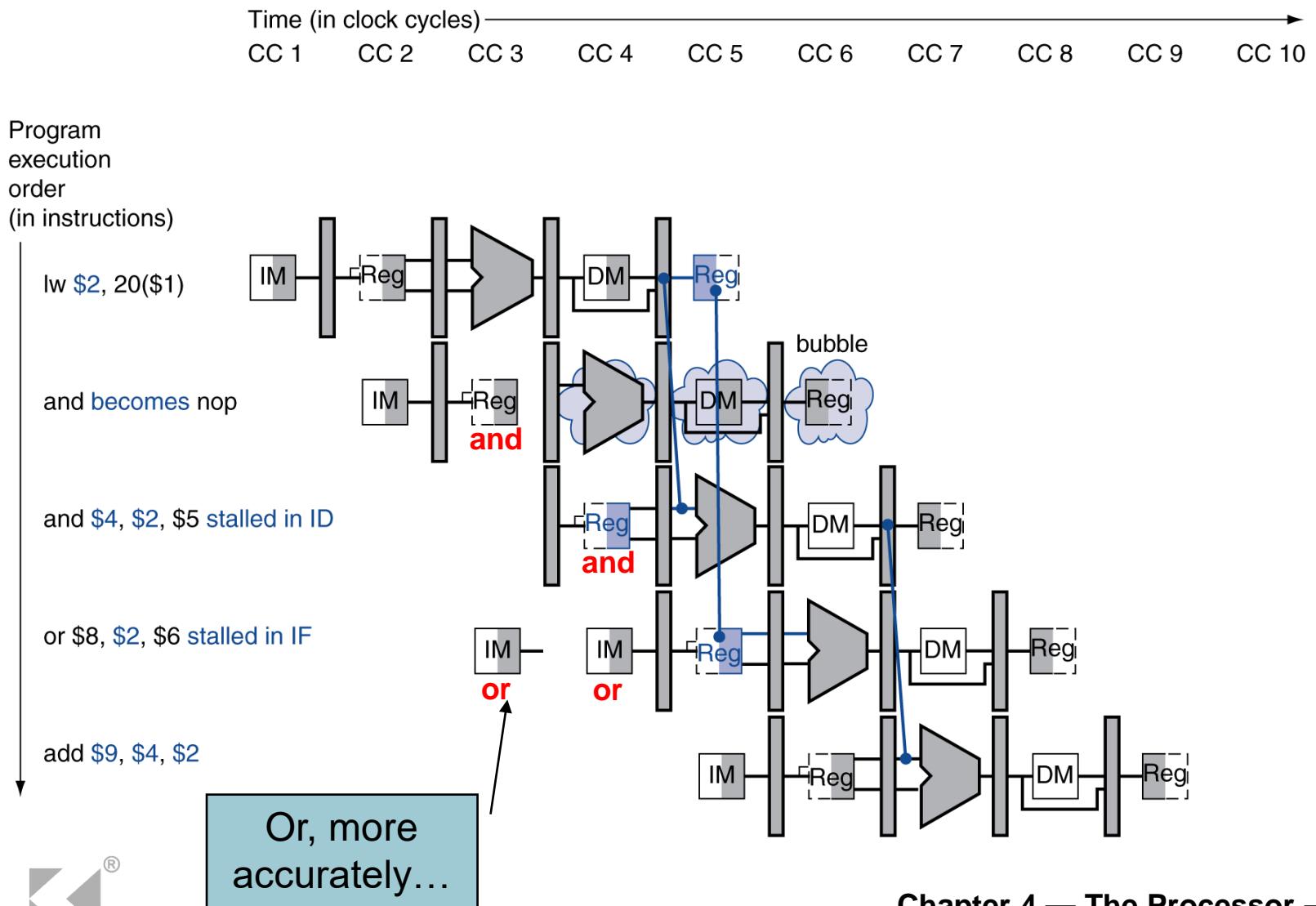
How to Stall the Pipeline

- 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

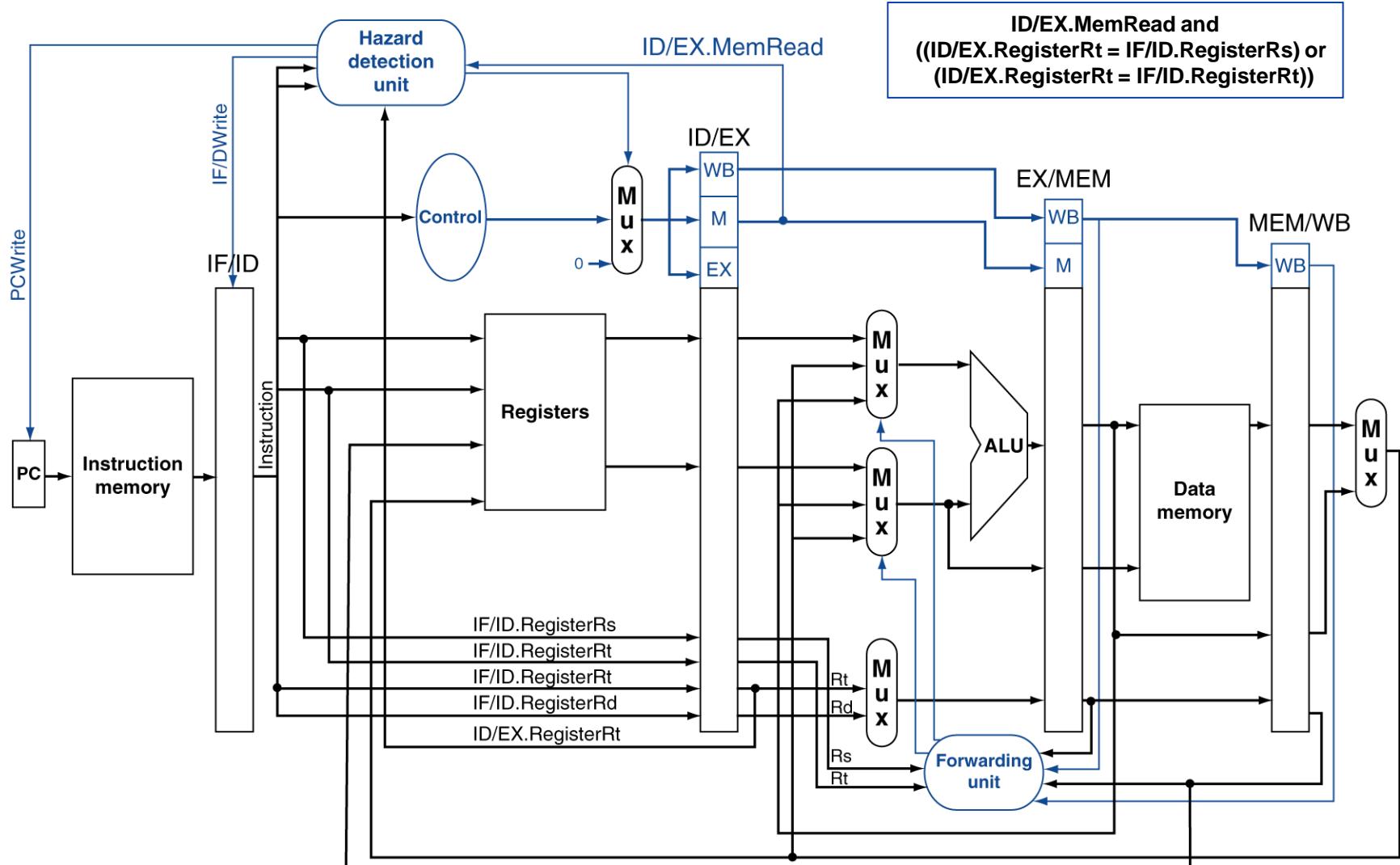
Stall/Bubble in the Pipeline



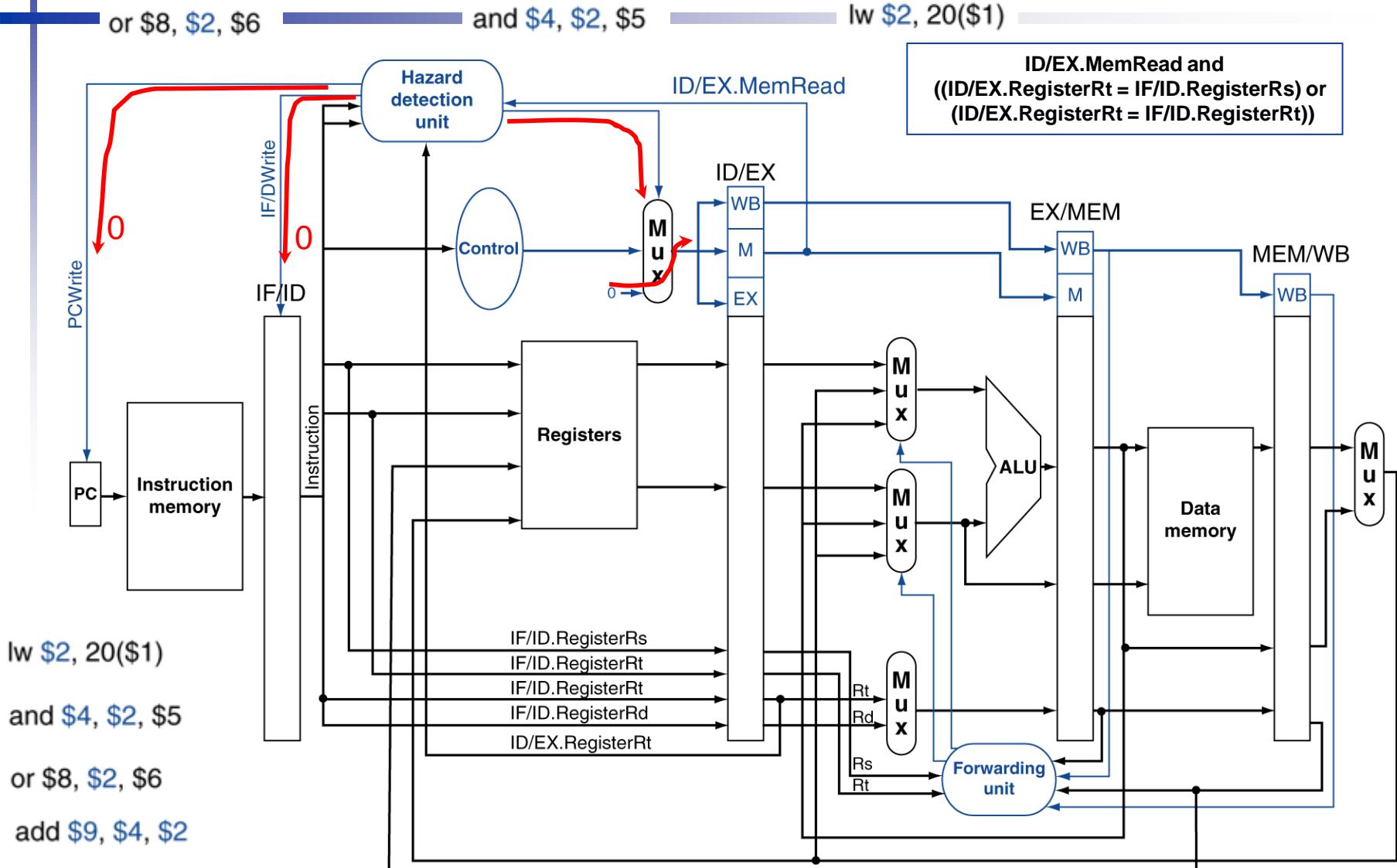
Stall/Bubble in the Pipeline



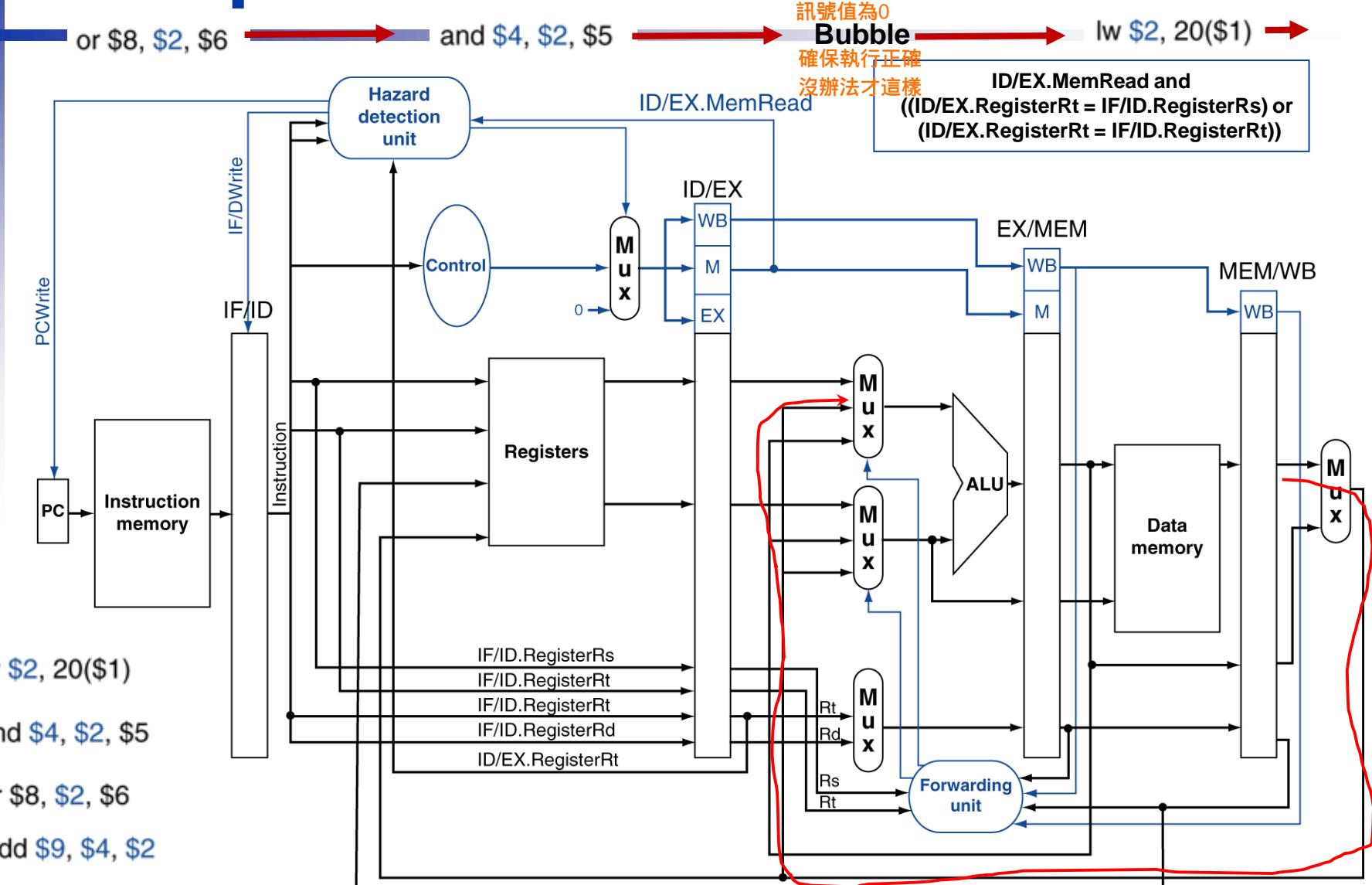
Datapath with Hazard Detection



Datapath with Hazard Detection



Datapath with Hazard Detection



Stalls and Performance

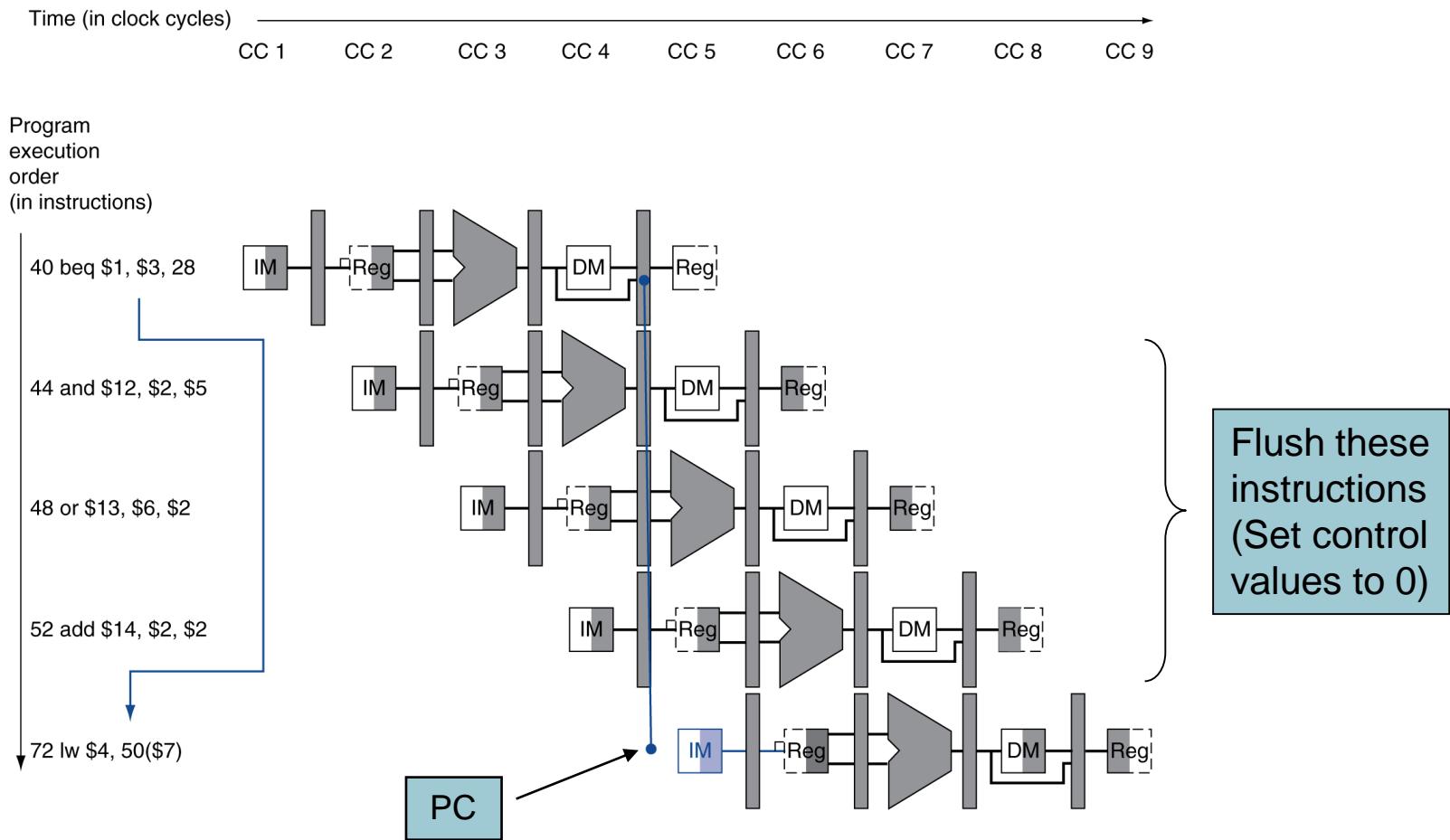
The BIG Picture

- 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

general purpose

Branch Hazards

- If branch outcome determined in MEM

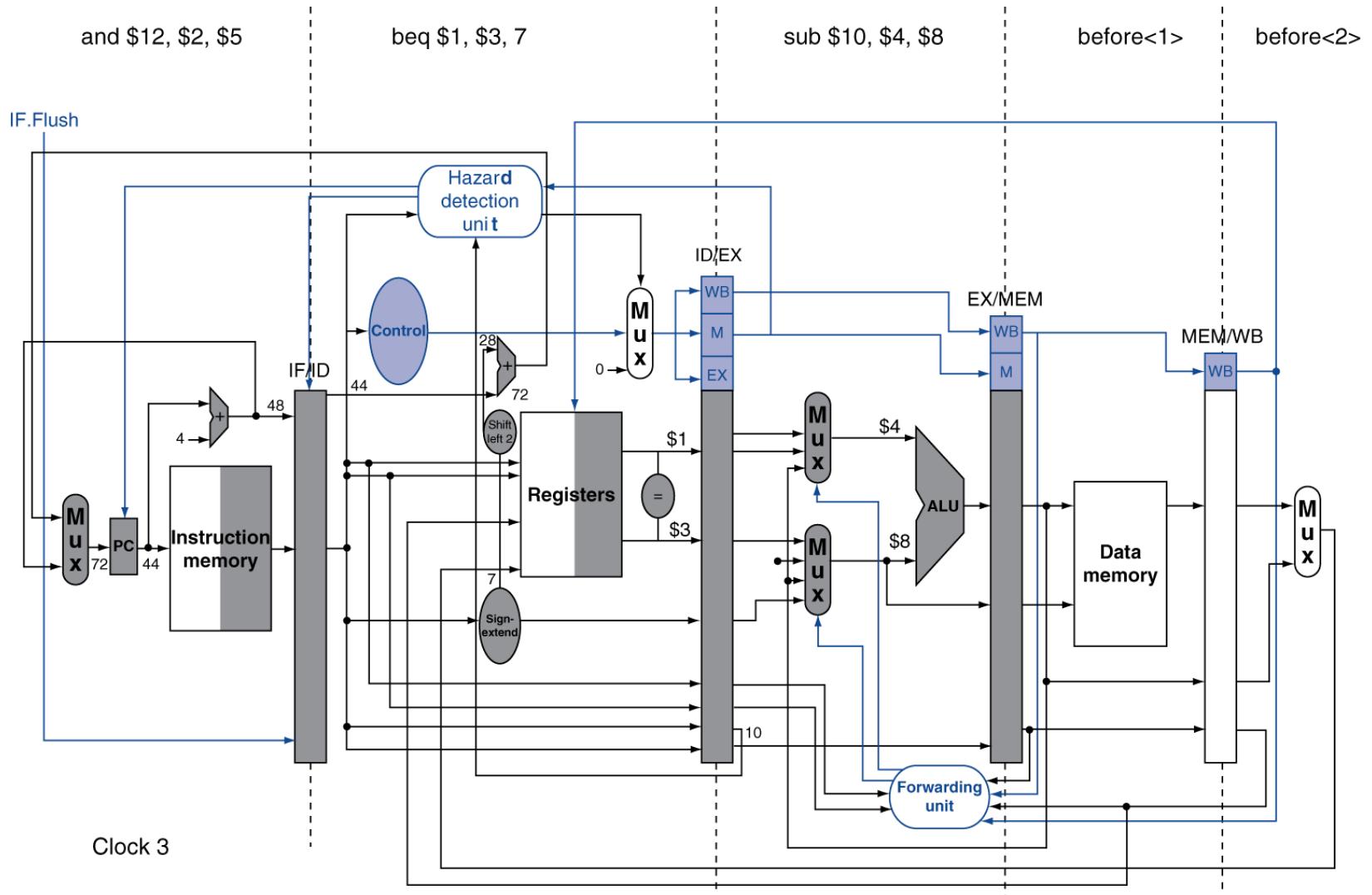


Reducing Branch Delay

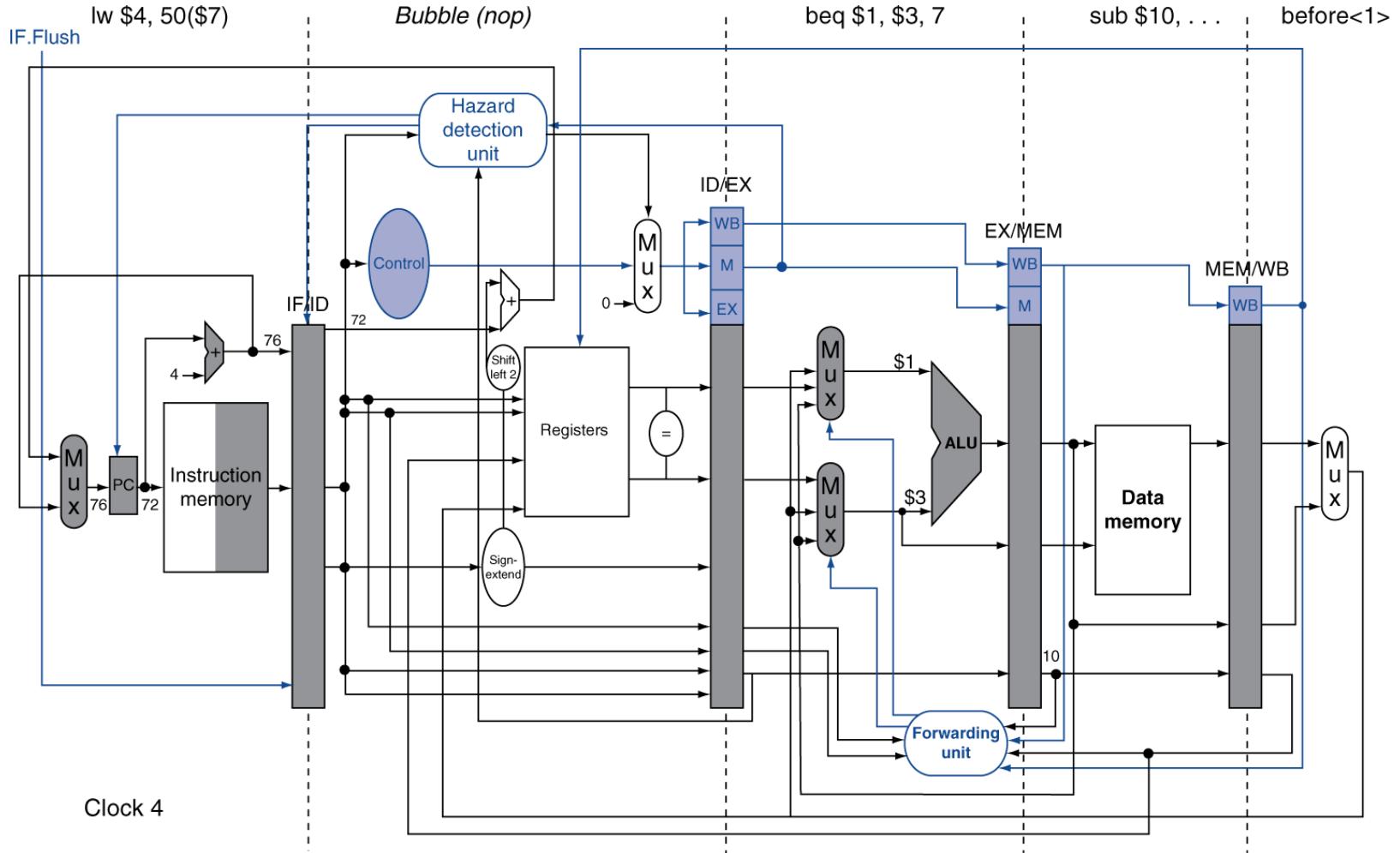
- Move hardware to determine outcome to ID stage
 - Target address adder
 - Register comparator
- Example: branch taken

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

Example: Branch Taken

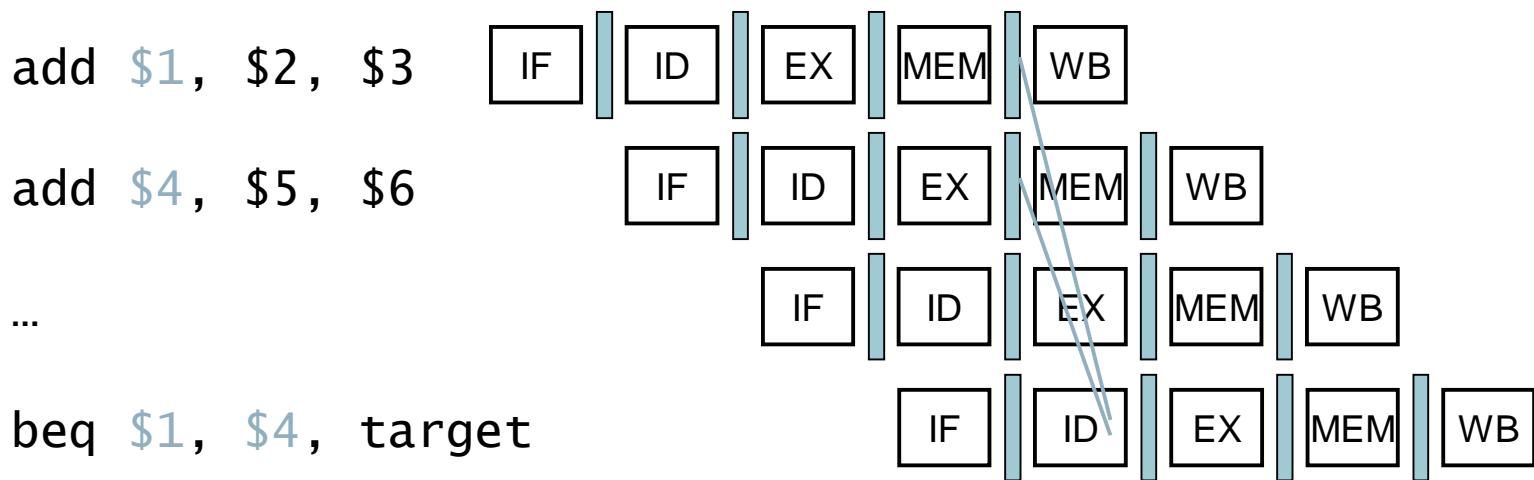


Example: Branch Taken



Data Hazards for Branches

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

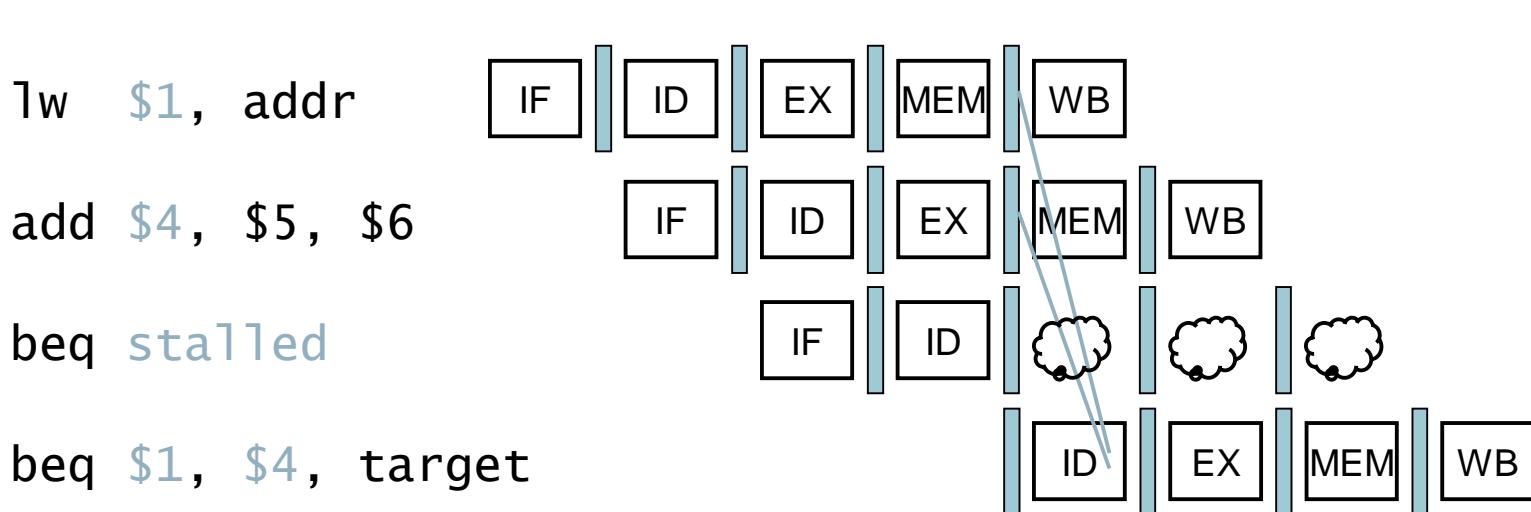


- Can resolve using forwarding

Data Hazards for Branches

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

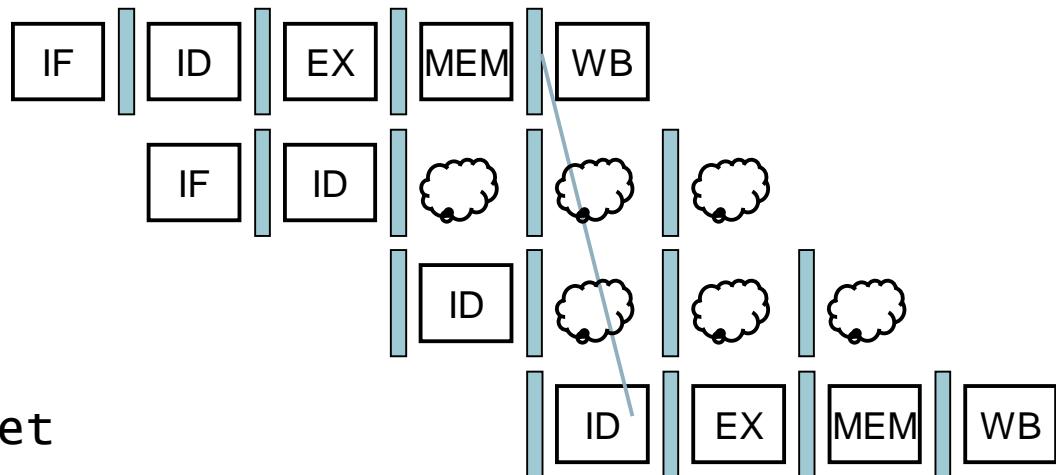
沒考慮forwarding的狀況



Data Hazards for Branches

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

lw \$1, addr
beq stalled
beq stalled
beq \$1, \$0, target



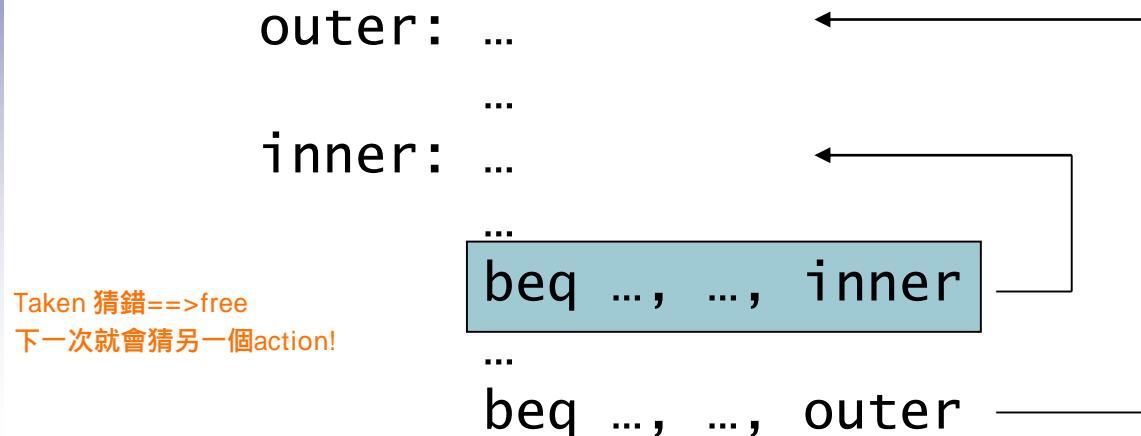
Dynamic Branch Prediction

- 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!

各個loop有自己的branch prediction(不共用)



Taken 猜錯==>free
下一次就會猜另一個action!

Assume inner loop
Iterates three times

One-bit	Two-bit
1 Taken V	1 Taken V
2 Taken V	2 Taken V
3 Taken X	3 Taken X
1 Ntaken X	1 Taken V
2 Taken V	2 Taken V
3 Taken X	3 Taken X

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

A loop branch that branches nine times in a row, then is not taken once. What's the prediction accuracy if 1-bit predictor is adopted.

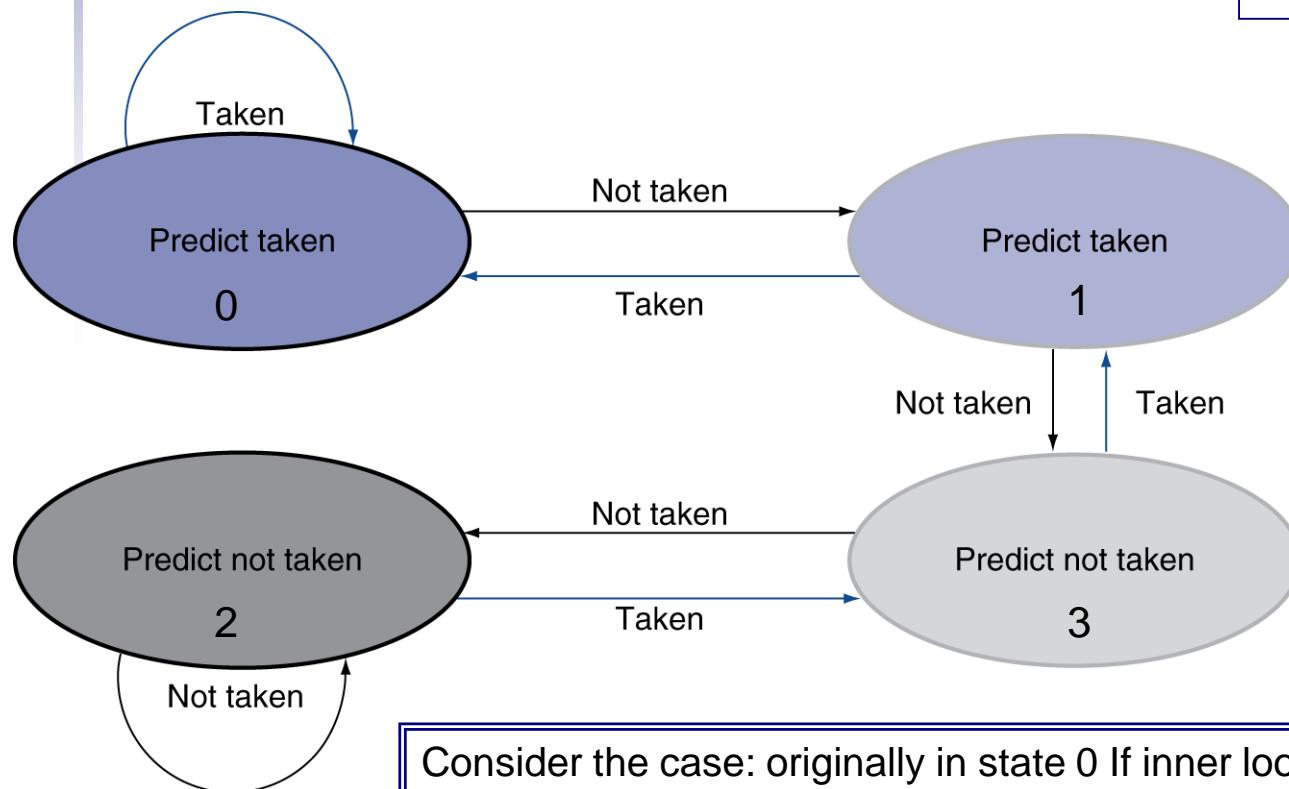
Ans. Consider the case the branch has been executed one time. 1st and last prediction are wrong : 80%



2-Bit Predictor

- Only change prediction on two successive mispredictions

Assume inner and outer loops both iterate three times

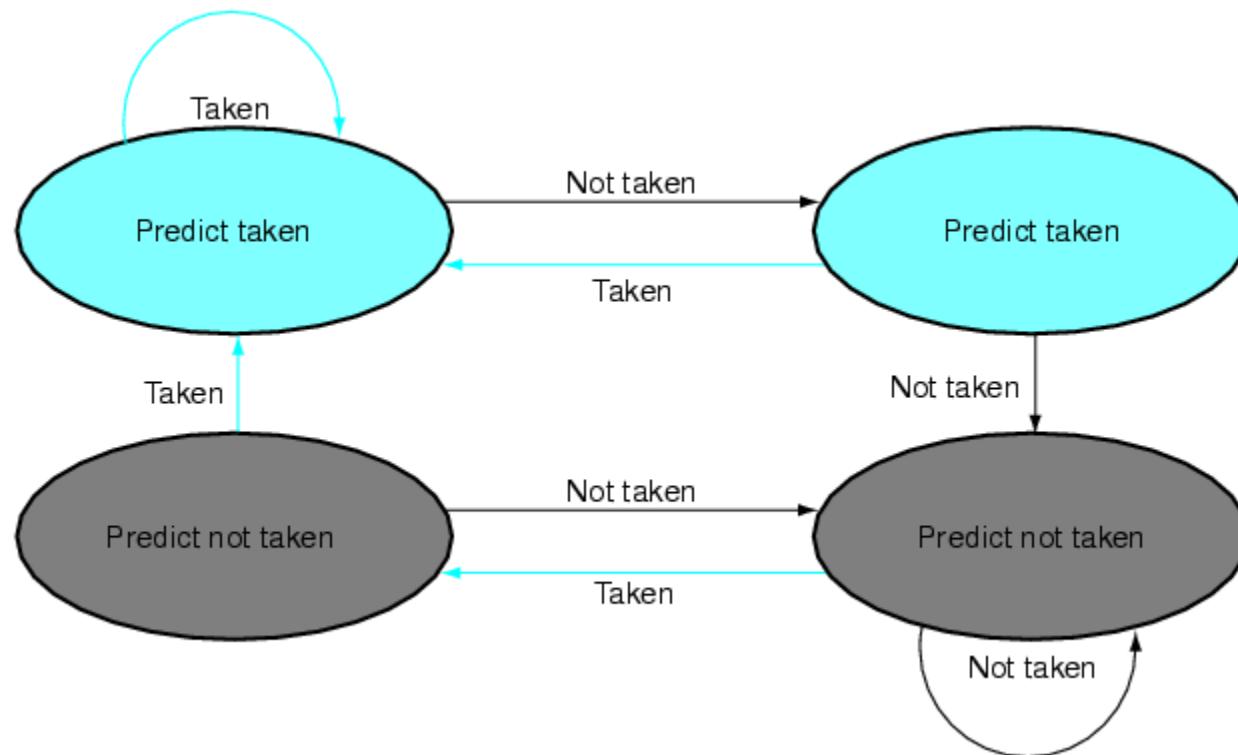


Inner loop	outer loop
1 Taken V	
2 Taken V	
3 Taken X	
	1 Taken V
1 Taken V	
2 Taken V	
3 Taken X	
	2 Taken V
1 Taken V	
2 Taken V	
3 Taken X	
	3 Taken X

Consider the case: originally in state 0 If inner loop bypasses two times and then repeats several times.

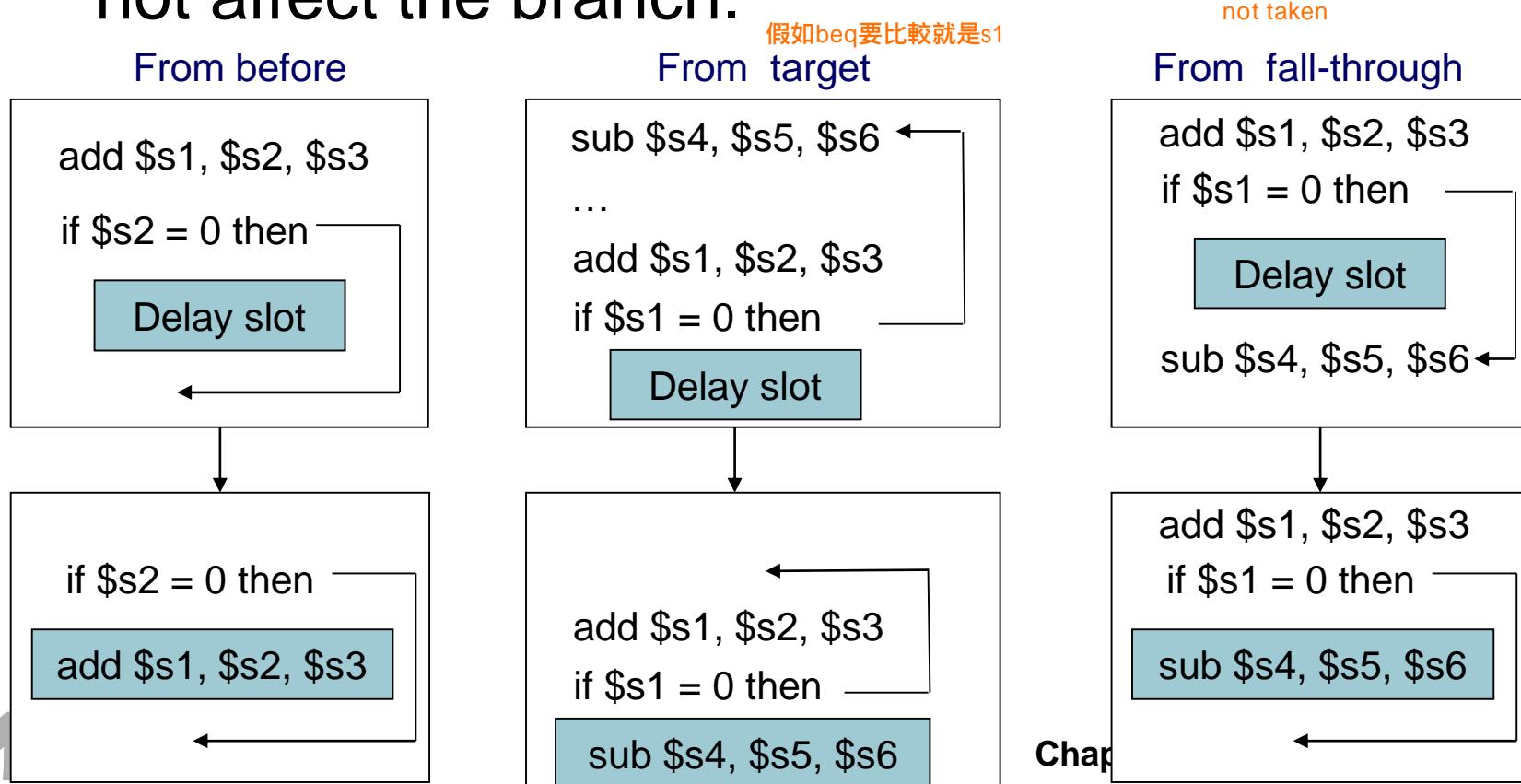
Two-bit Prediction Scheme

- In a 2-bit scheme, a state change between taken and untaken is made only after twice mis-prediction



Branch Delay Slot

- Branch delay slot – the slot directly after the delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.



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

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



Handling Exceptions

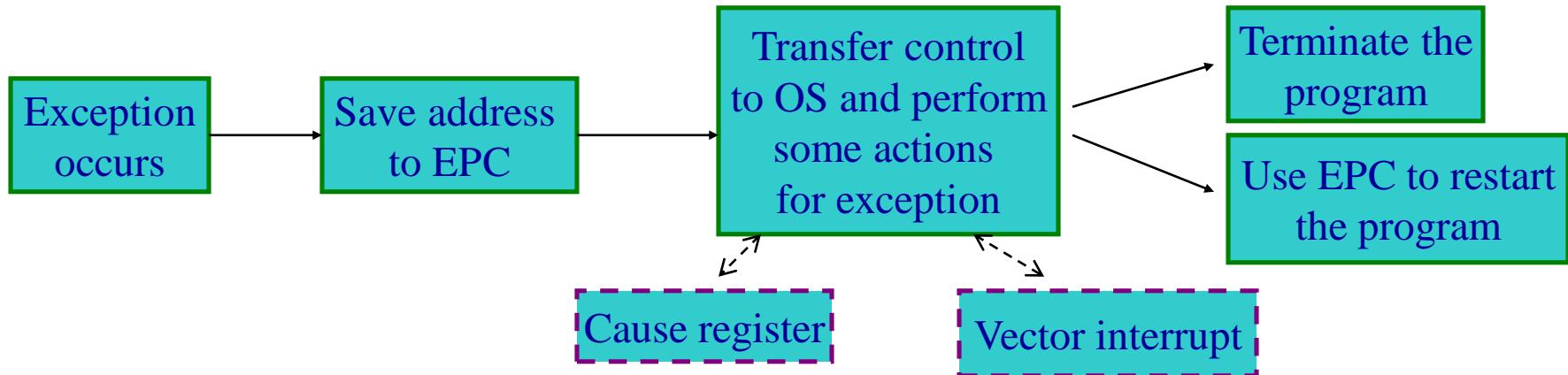
- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
 - In MIPS: Exception Program Counter (EPC)
- For OS, the ways to communicate with exceptions
 - In MIPS: Cause register – save indication of the problem
 - We'll assume 1-bit
 - 0 for undefined opcode, 1 for overflow
 - Vectored interrupts
- In MIPS, jump to handler at 8000 00180 統一入口點

An Alternate Mechanism

- Vectored Interrupts
 - Handler address determined by the cause (exception type)
- Example:
 - Undefined opcode: C000 0000
 - Overflow: C000 0020
 - ...: C000 0040
- Instructions either
 - Deal with the interrupt, or
 - Jump to real handler

Handler Actions

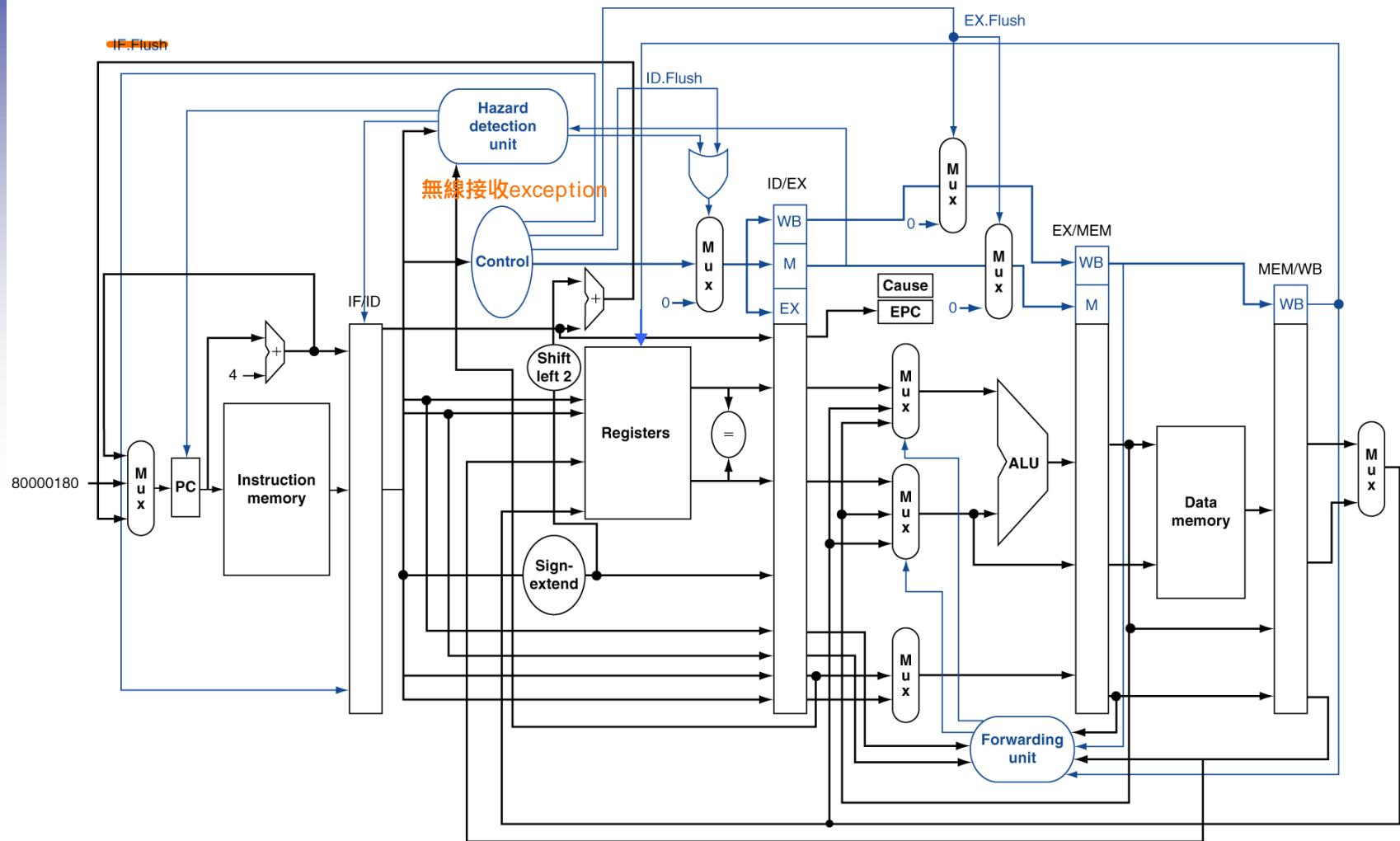
- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
 - Take corrective action
 - use EPC to return to program
- Otherwise
 - Terminate program
 - Report error using EPC, cause, ...



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 跟branch一樣
- 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 所以要減4再存
 - 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)
```

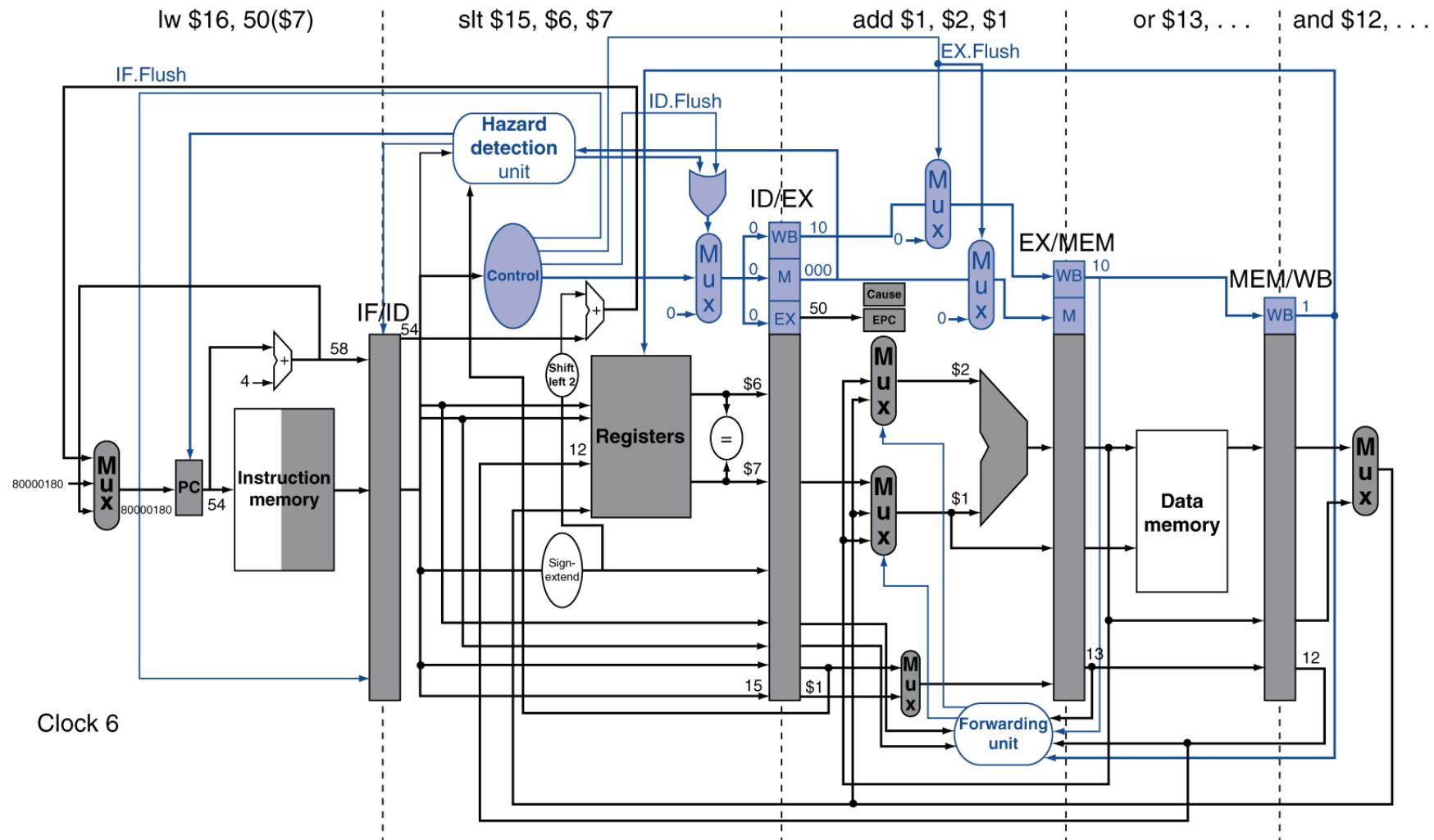
...

- Handler

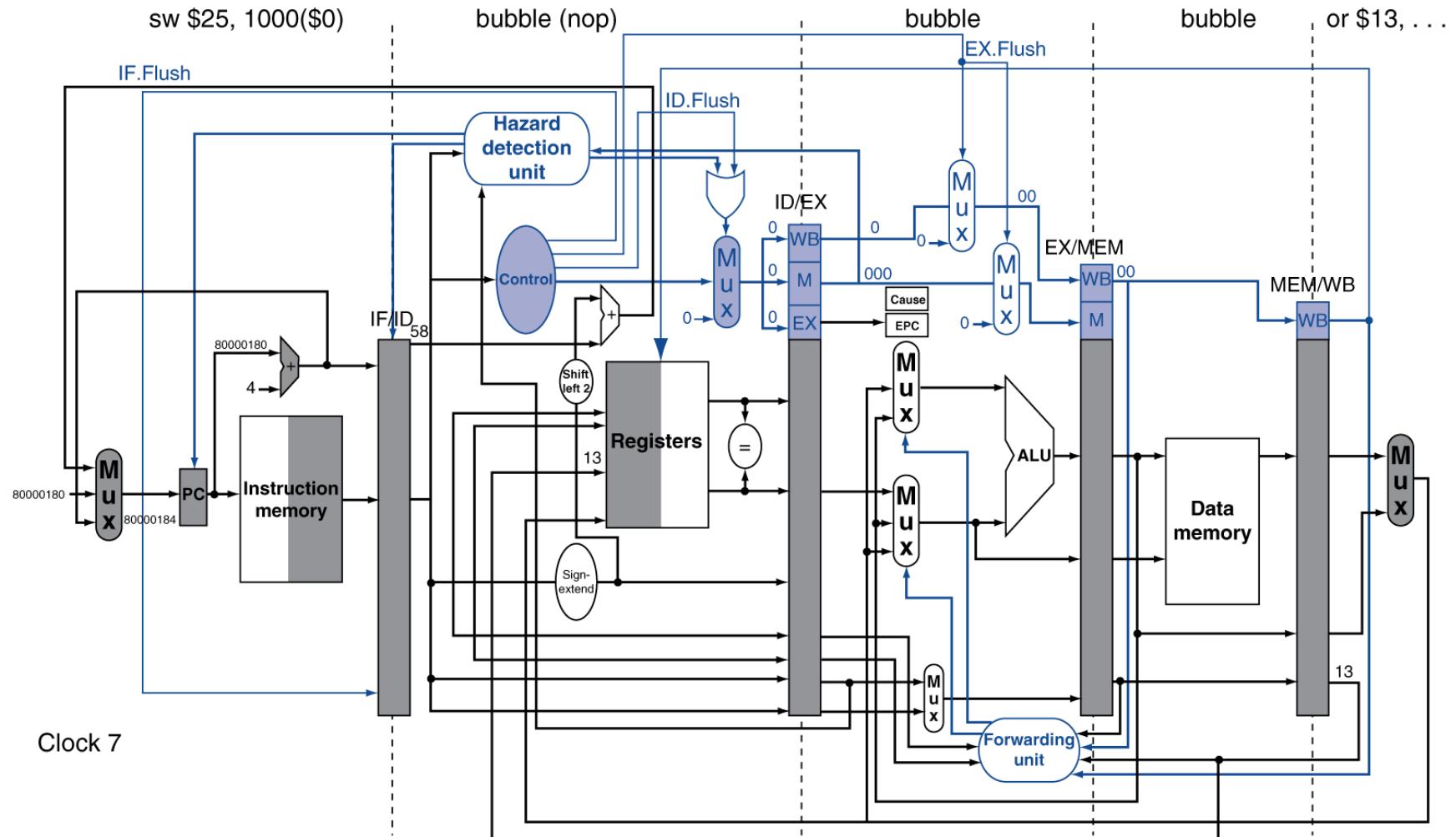
```
80000180      sw    $25, 1000($0)
80000184      sw    $26, 1004($0)
```

...

Exception Example



Exception Example



Multiple Exceptions

大概提一下

- 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 不一定與最早發生的順序一樣!
 - Multiple instructions issued per cycle
 - Out-of-order completion
 - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Imprecise exception –Interruptions or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.
- 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 (e.g. sw precedes lw)
 - Roll back if location is updated

可能lw sw的地址一樣!!

sw \$2, 100(\$3)
lw \$4, 200(\$5)

Compiler/Hardware Speculation

- Compiler can reorder instructions
 - e.g., move an instruction across a branch or a load across a store
 - Can include accuracy check and “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
 - Buffer results and write to register or memory until it determines they are actually needed
 - Flush buffers on incorrect speculation

beq ...
add...

sw ...
lw ...



Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
 - speculative load before null-pointer check
 - Speculative `lw` changes the value of a register such that an exception occurs on an instruction preceding `lw`
- Static speculation
 - Can add ISA support for deferring exceptions
- Dynamic speculation
 - Can buffer exceptions until instruction completion (which may not occur)

`*ptr = 10; // ptr = 0;`

`lw $2, 100($4); // $4 + 100 is the location of ptr` ← Speculative `lw`



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)

Scheduling Static Multiple Issue

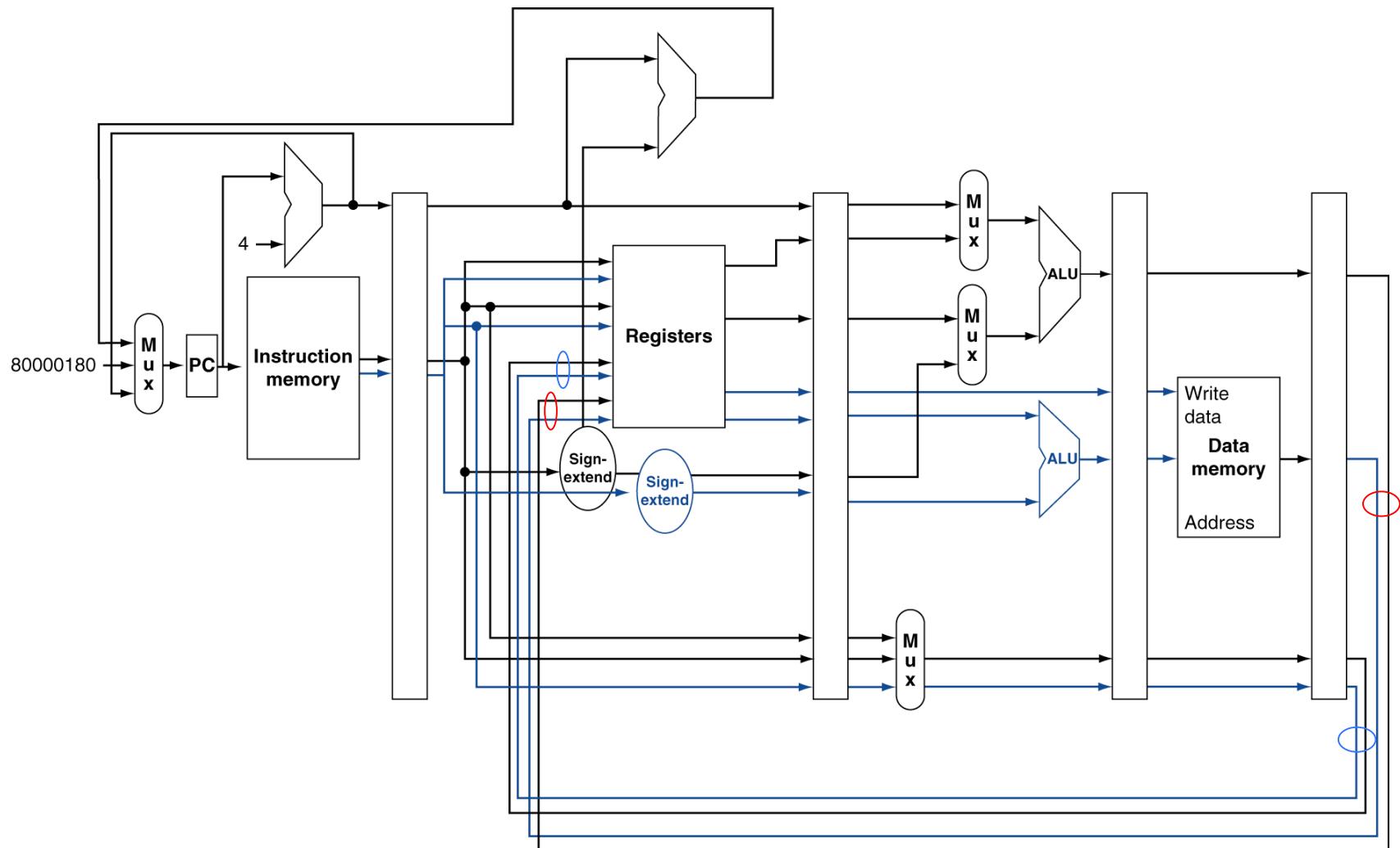
- Compiler must remove some/all hazards
 - Reorder instructions into issue packets
 - No dependencies within a packet
 - Possibly some dependencies between packets
 - Varies between ISAs; compiler must know!
 - Pad with nop if necessary

MIPS with Static Dual Issue

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

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

MIPS with Static Dual Issue



Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- Removing hazards in static multiple issues
 - Compiler-based: code scheduling and no-op insertion.
 - Hardware-based: detect data hazard and generate all stalls between two issue packets, compiler avoids all dependencies within an instruction pair.
- EX data hazard
 - Forwarding avoided stalls with single-issue
 - Now can't use ALU result in load/store in same packet
 - add \$t0, \$s0, \$s1 load \$s2, 0(\$t0)
 - Split into two packets, effectively a stall
- Load-use hazard # of clock cycles between a lw and an instruction that can use the result of lw without stalling the pipeline
 - Still one cycle use latency, but now two instructions
 - E.g. lw, add, sw (add and sw use data from lw), (c1, c3, c4) becomes (c1, c3)
- 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	Load/store	cycle
Loop:	nop	lw \$t0, 0(\$s1)	1
	addi \$s1, \$s1,-4	nop	2
	addu \$t0, \$t0, \$s2	nop	3
	bne \$s1, \$zero, Loop	sw \$t0, 4(\$s1)	4

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

Loop Unrolling

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

```
for (i=5; i >= 0;i--)  
    a[i]=a[i]+5;
```

```
a[4]=a[4]+5;  
a[3]=a[3]+5;  
a[2]=a[2]+5;  
a[1]=a[1]+5;  
a[0]=a[0]+5;
```

Many anti-dependencies

```
lw $t0, 0($s1)  
addu $t0, $t0, $s2  
sw $t0, 0($s1)  
addi $s1, $s1,-4  
  
lw $t0, 0($s1)  
addu $t0, $t0, $s2  
sw $t0, 0($s1)  
addi $s1, $s1,-4  
...
```

a[4]=a[4]+5;

a[3]=a[3]+5;

Loop Unrolling Example

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

- IPC = 14/8 = 1.75
 - Closer to 2, but at cost of registers and code size

Loop: lw \$t0, 0(\$s1)
addu \$t0, \$t0, \$s2
sw \$t0, 0(\$s1)
addi \$s1, \$s1,-4
bne \$s1, \$zero, Loop

lw \$t0, 0(\$s1)
addu \$t0, \$t0, \$s2
sw \$t0, 0(\$s1)
addi \$s1, \$s1,-4

lw \$t0, 0(\$s1)
addu \$t0, \$t0, \$s2
sw \$t0, 0(\$s1)
addi \$s1, \$s1,-4

Dynamic Multiple Issue

- “Superscalar” processors
 - An advanced pipelining techniques that enables the processor to execute more than one instruction per clock cycle **by selecting them during execution.**
- 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

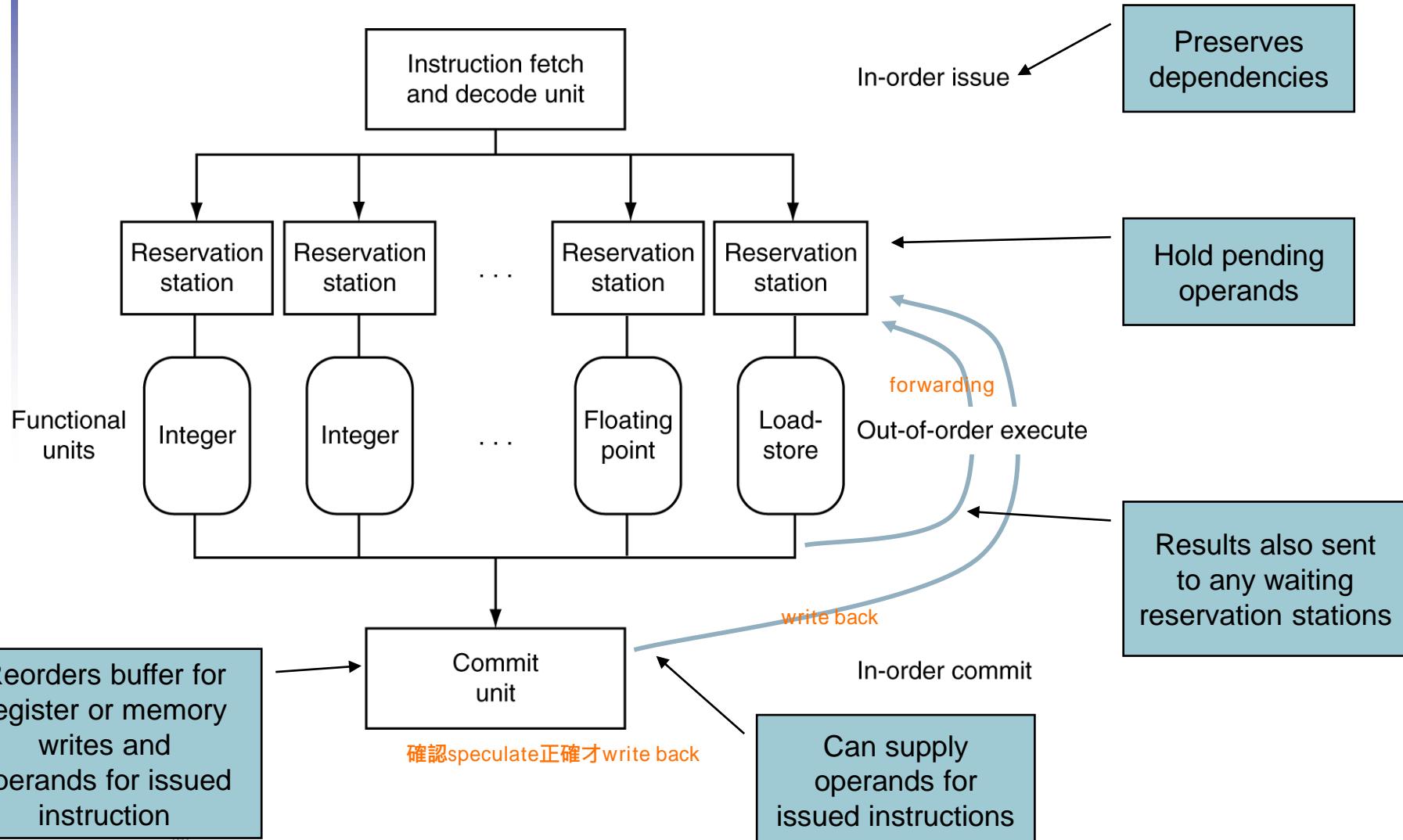
Dynamic Pipeline Scheduling

- 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
slt   $t5, $s4, 20
```

- Can start sub while addu is waiting for lw

Dynamic Pipeline Scheduling



Register Renaming

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

Speculation

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

Why Do Dynamic Scheduling?

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

Does Multiple Issue Work?

The BIG Picture

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

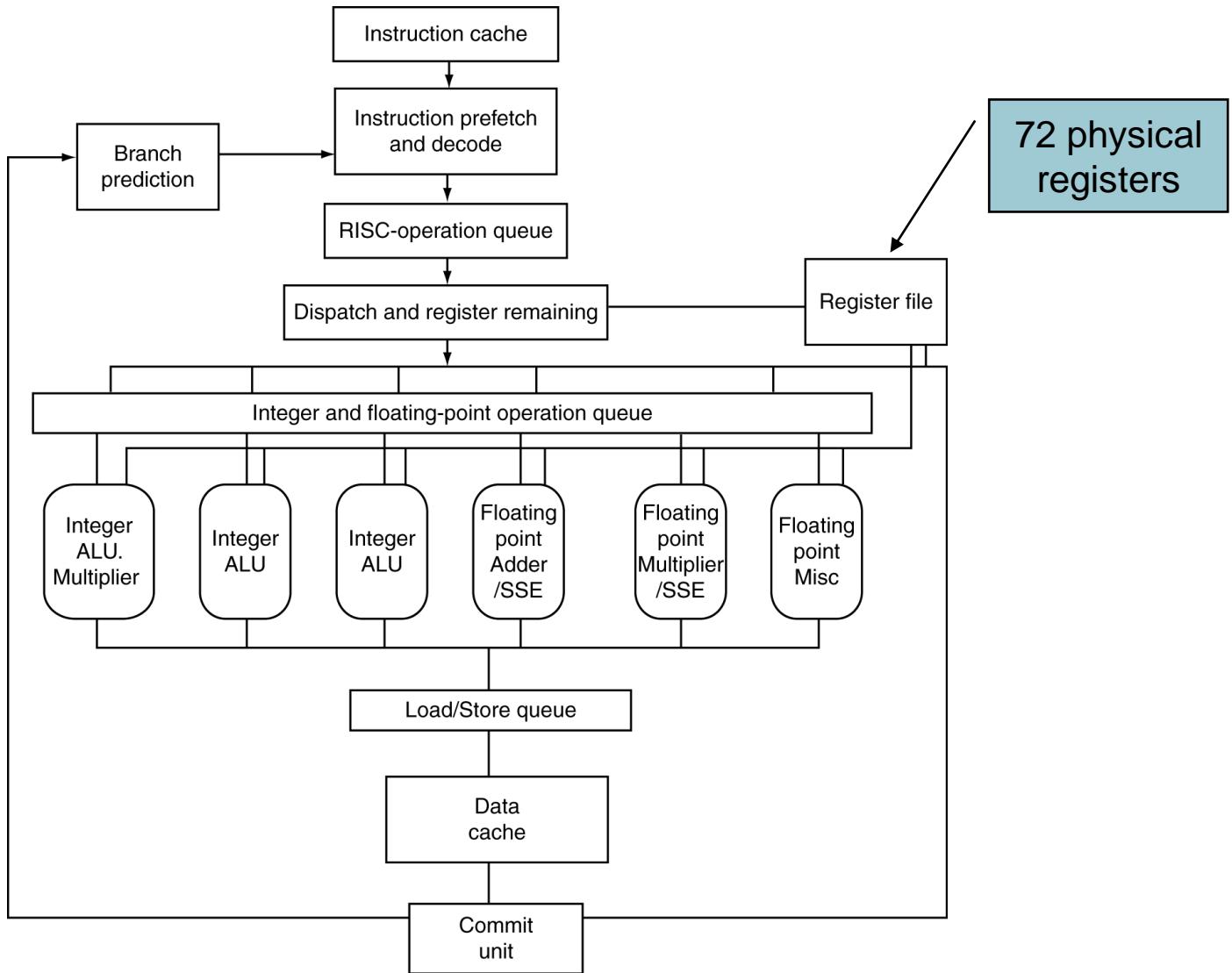


Power Efficiency

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

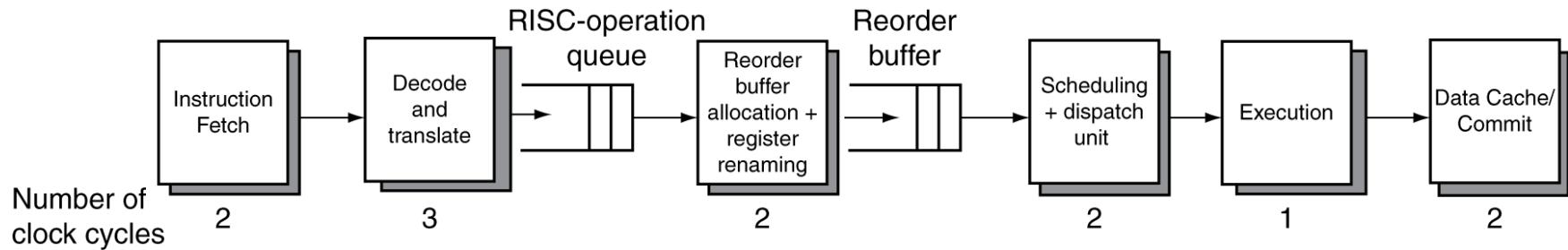
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4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

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