



Chapter 4

The Processor

Introduction

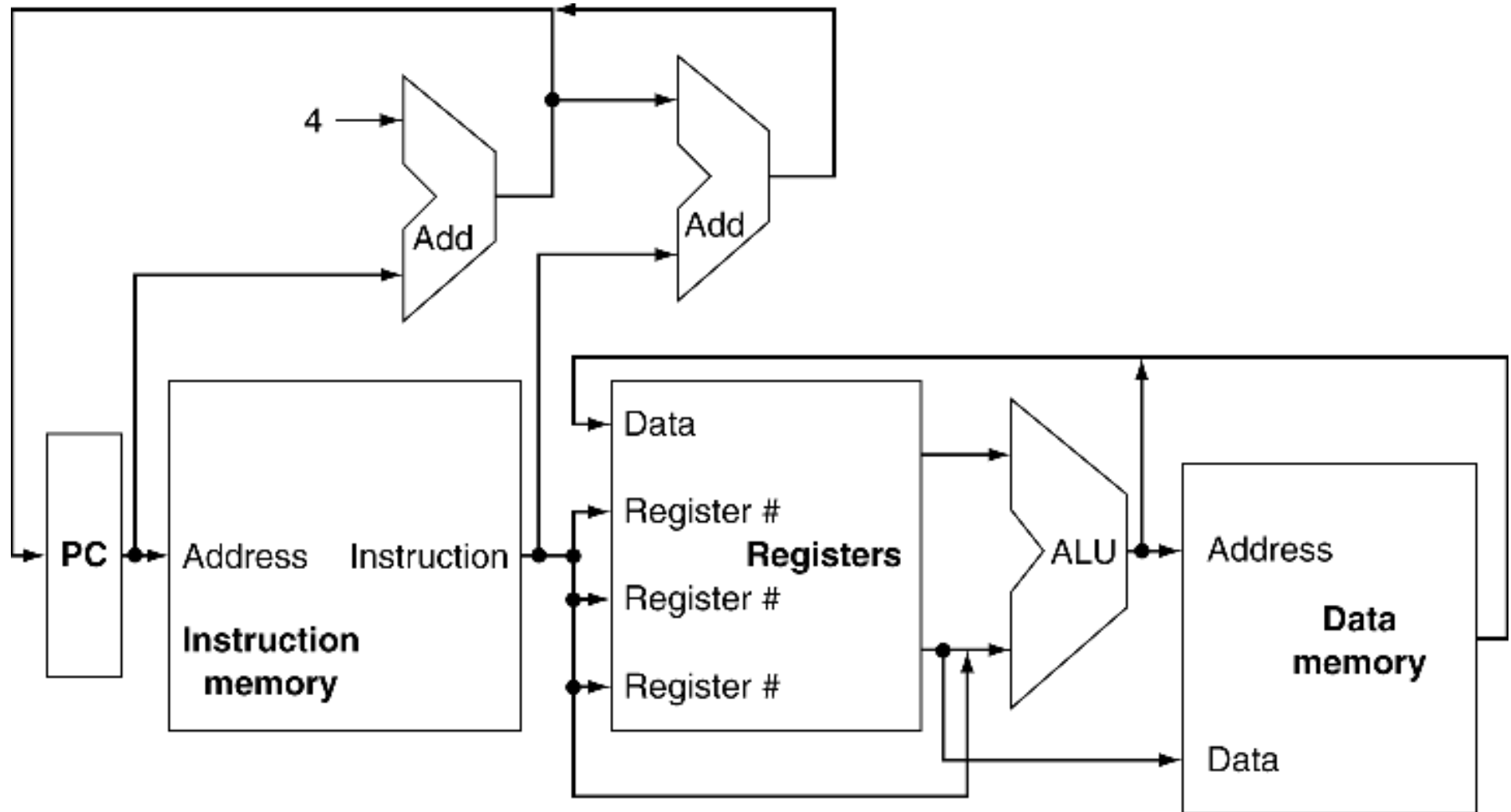
- CPU performance factors
 - Instruction count
 - Determined by ISA and compiler
 - CPI and Cycle time
 - Determined by CPU hardware
- We will examine three MIPS implementations
 - A single cycle version
 - Multi-Cycle version
 - A more realistic pipelined version
- Simple subset, shows most aspects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



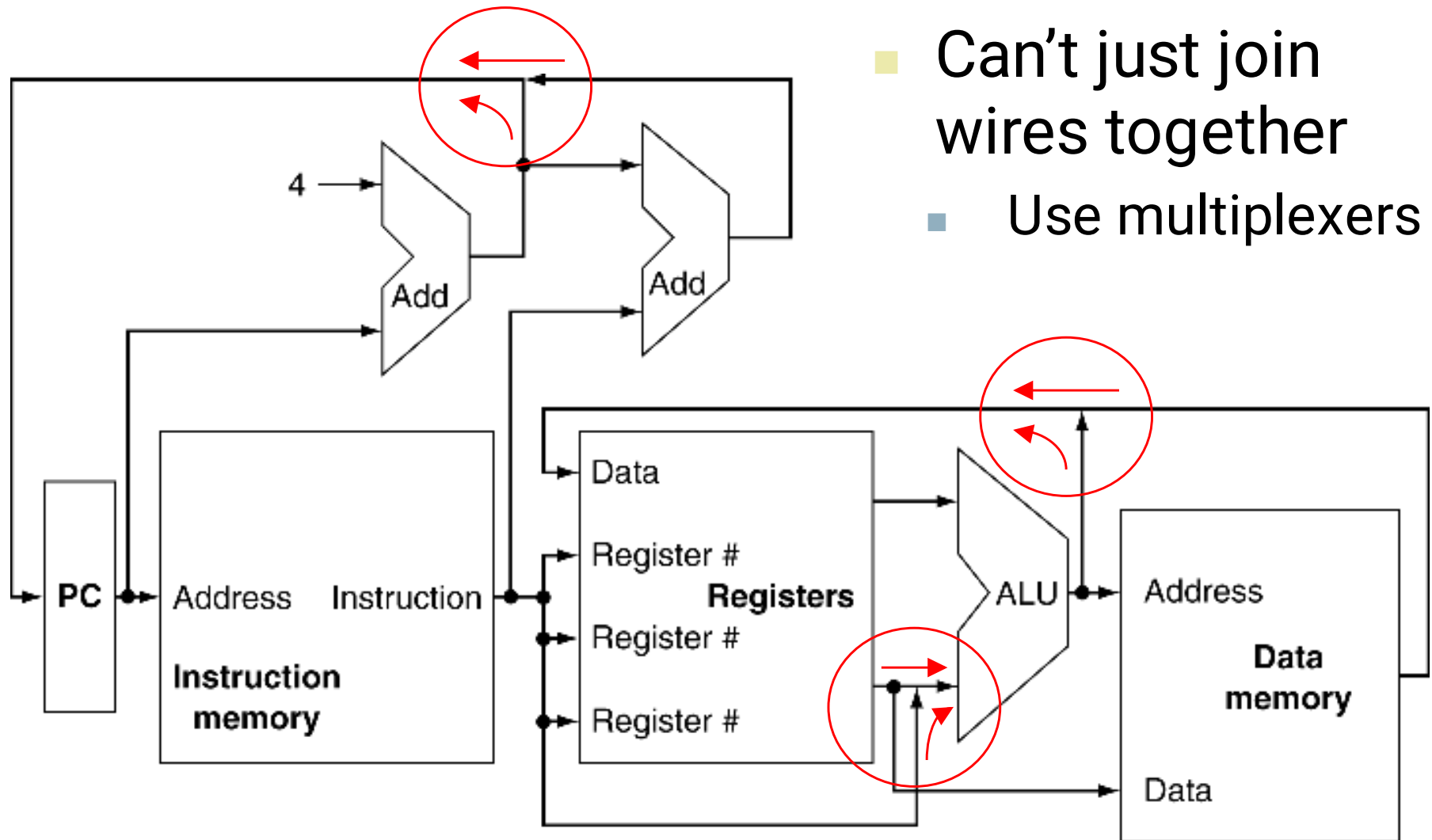
Instruction Execution

- 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
 - $PC \leftarrow \text{target address or } PC + 4$

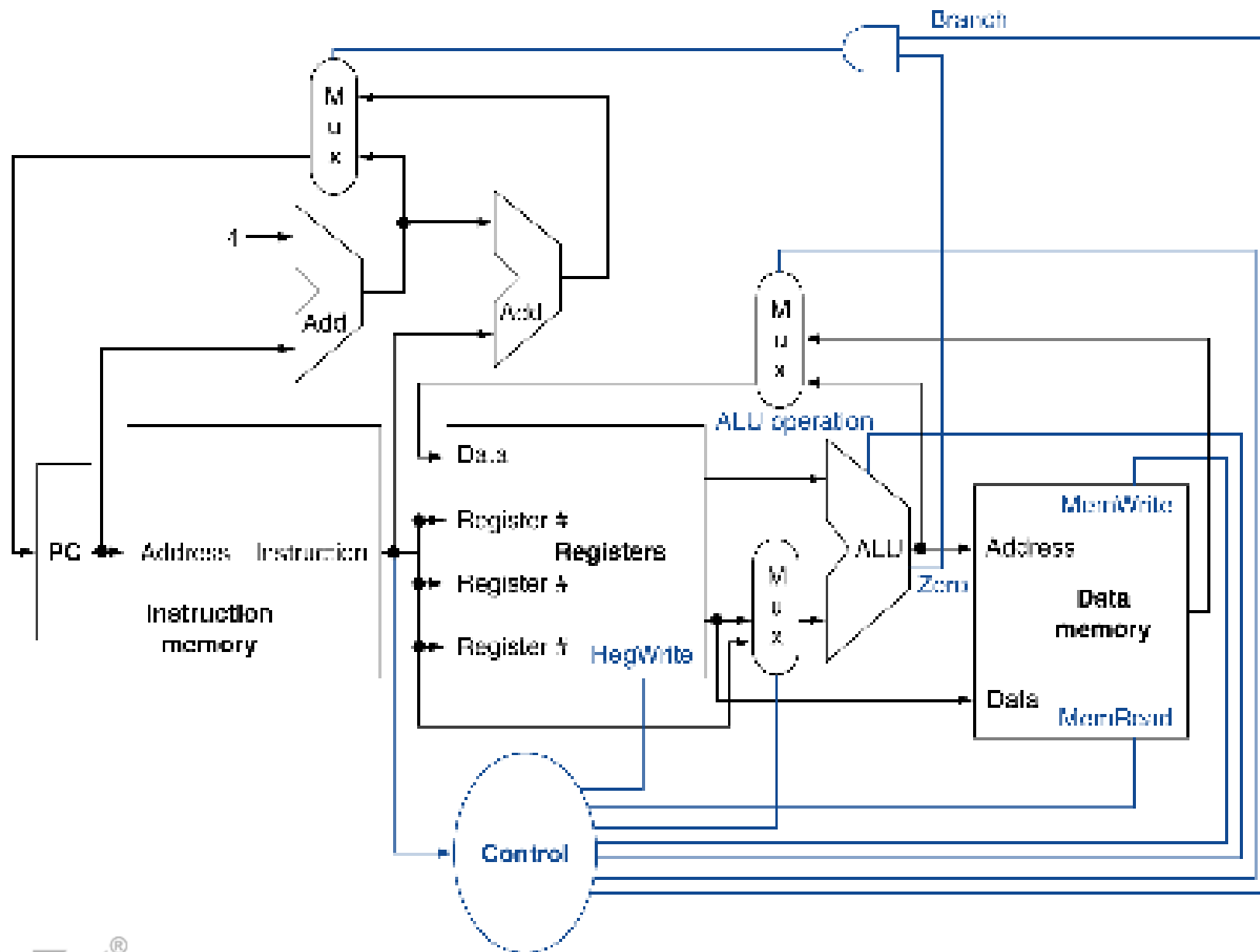
CPU Overview



Multiplexers



Control



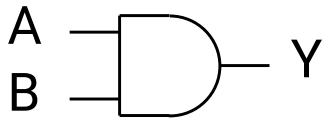
Logic Design Basics

- 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
 - Output is a function of input
- State (sequential) elements
 - Store information

Combinational Elements

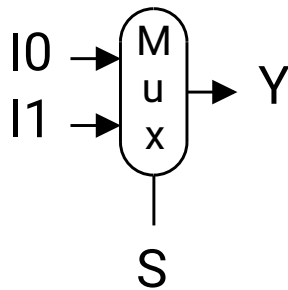
- AND-gate

- $Y = A \& B$



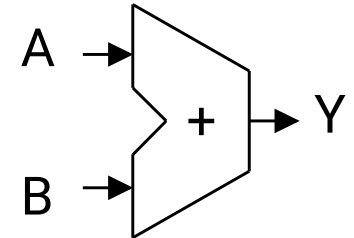
- Multiplexer

- $Y = S ? I1 : I0$



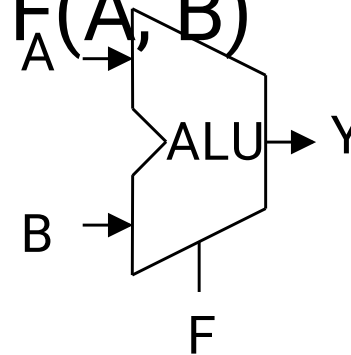
- Adder

- $Y = A + B$



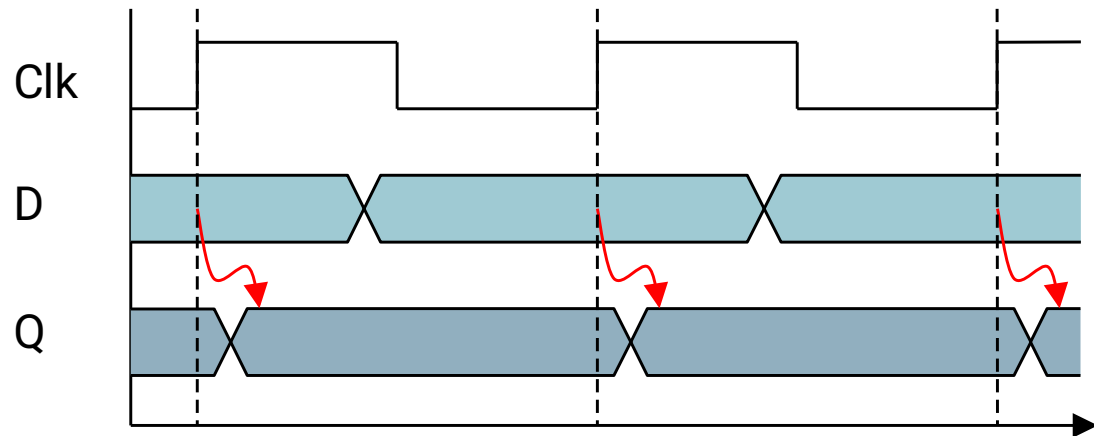
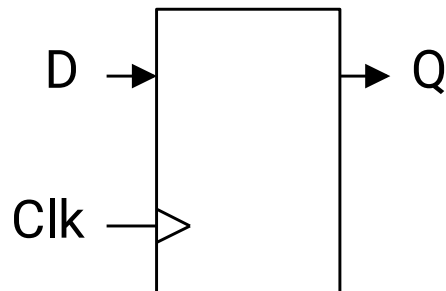
- Arithmetic/Logic Unit

- $Y = F(A, B)$



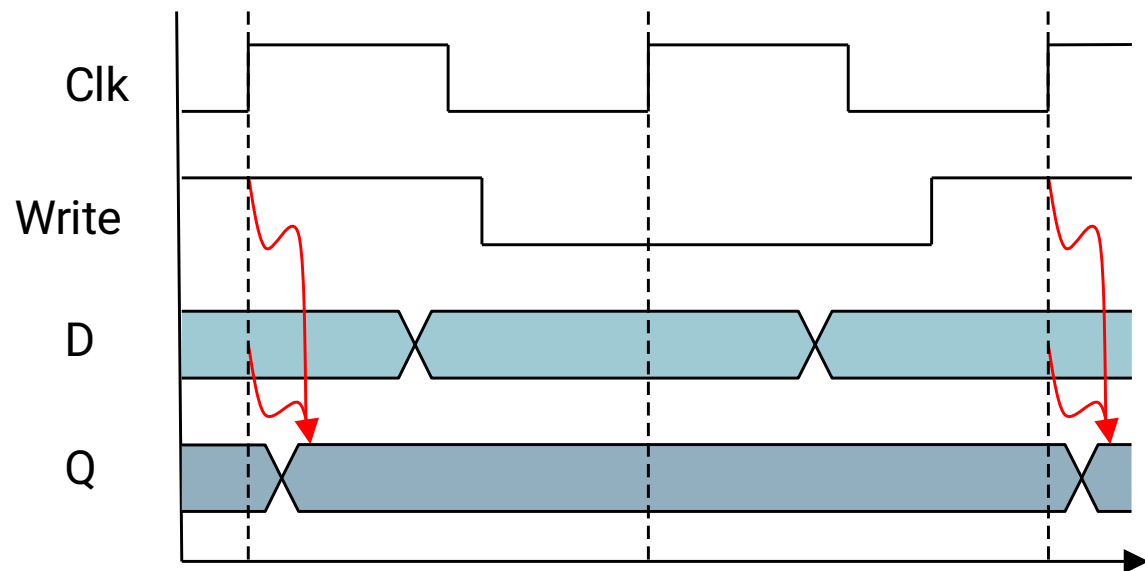
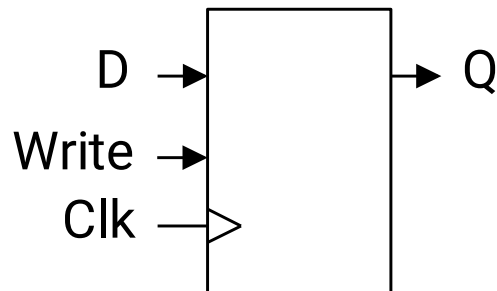
Sequential Elements

- 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



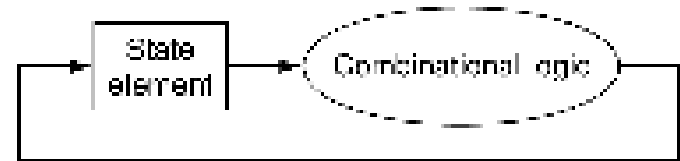
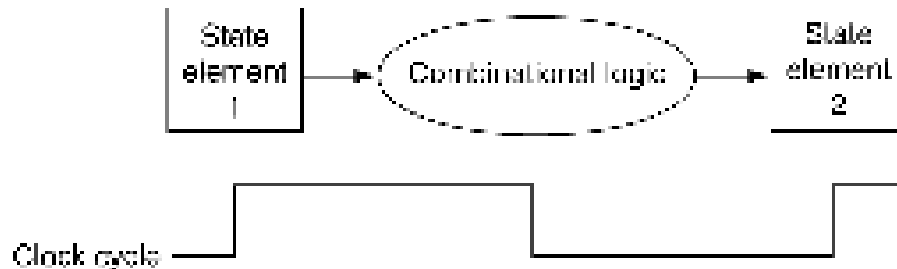
Sequential Elements

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

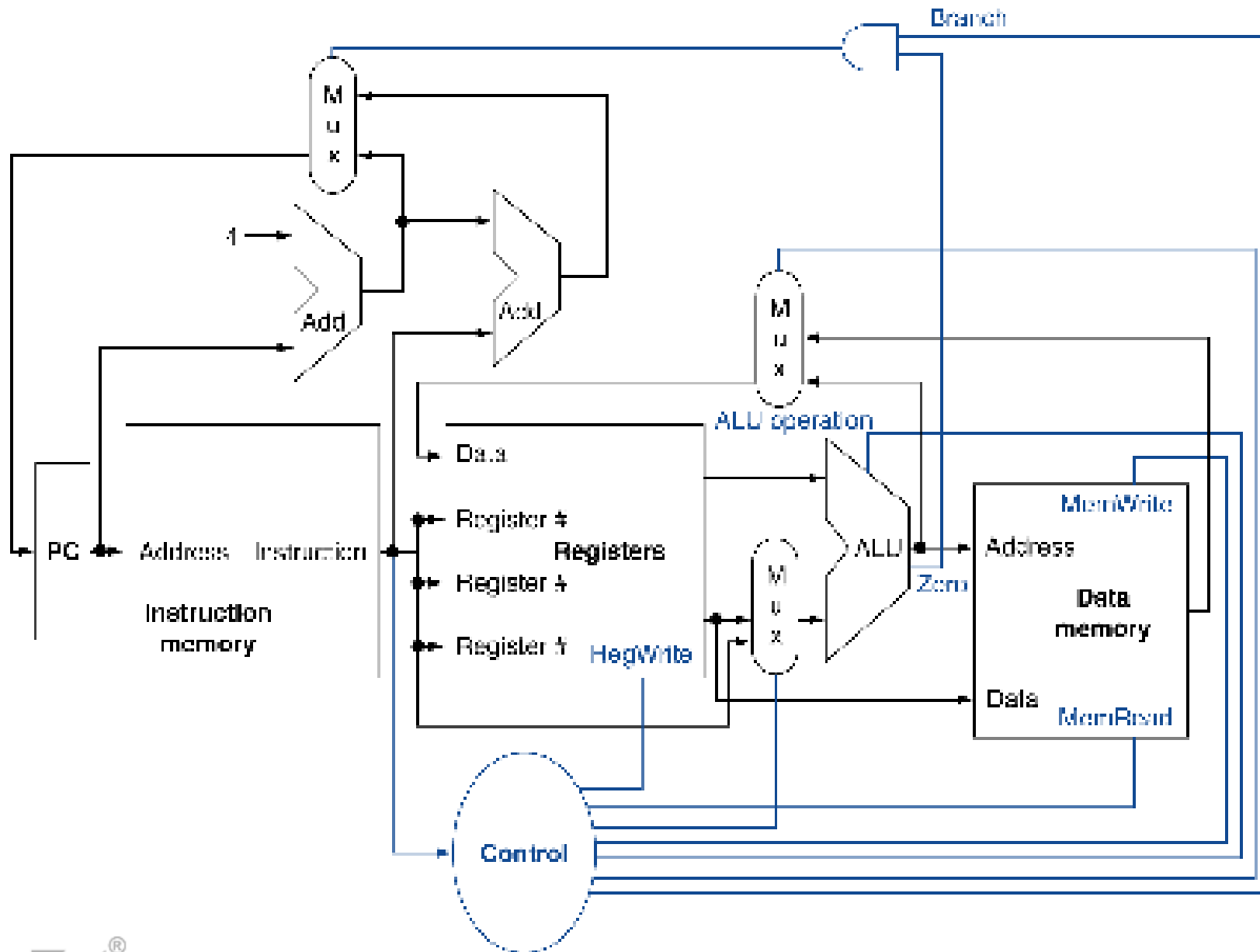


Clocking Methodology

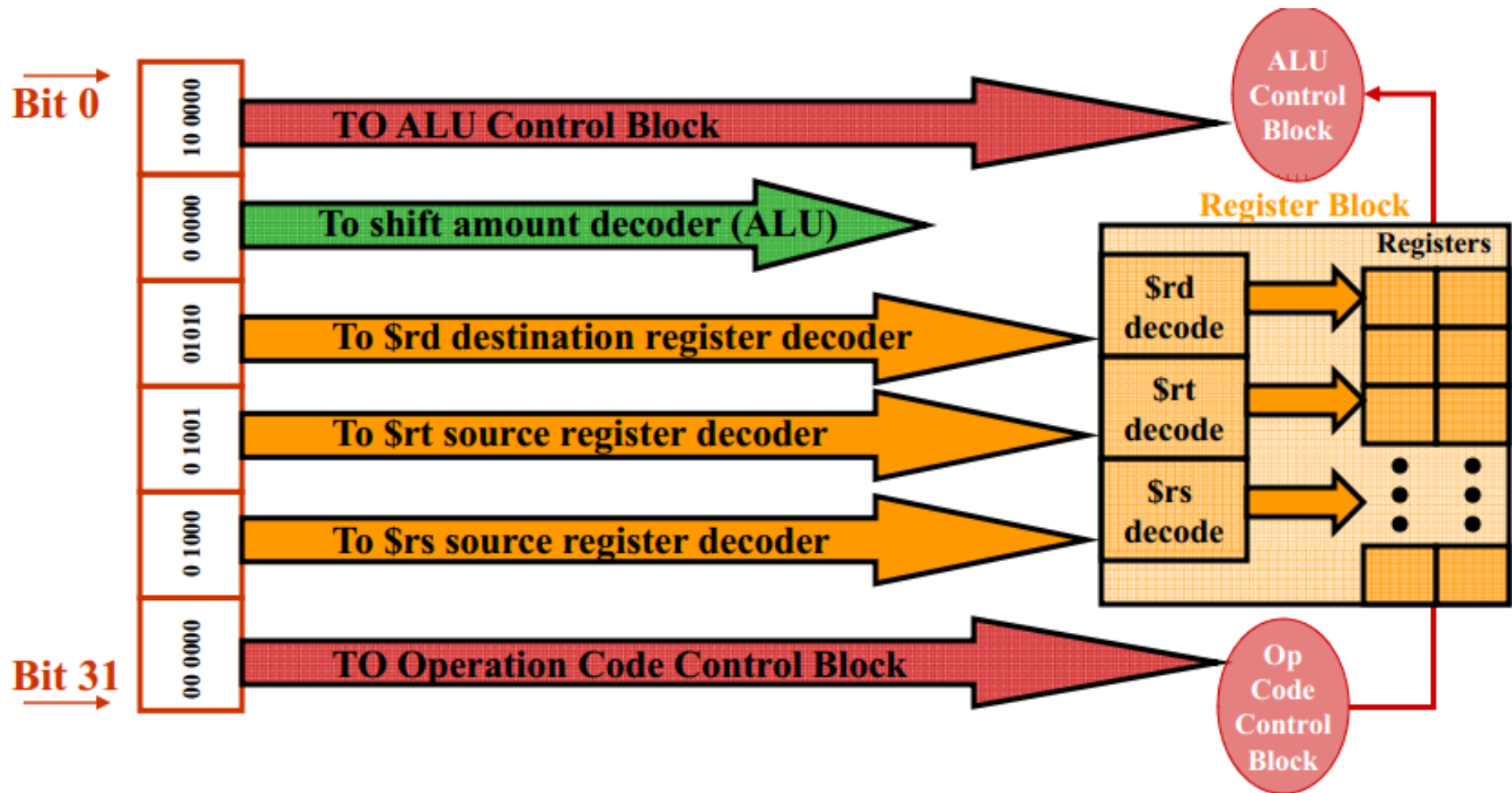
- Combinational logic transforms data during clock cycles
 - Between clock edges
 - Input from state elements, output to state element
 - Longest delay determines clock period



Data path with control



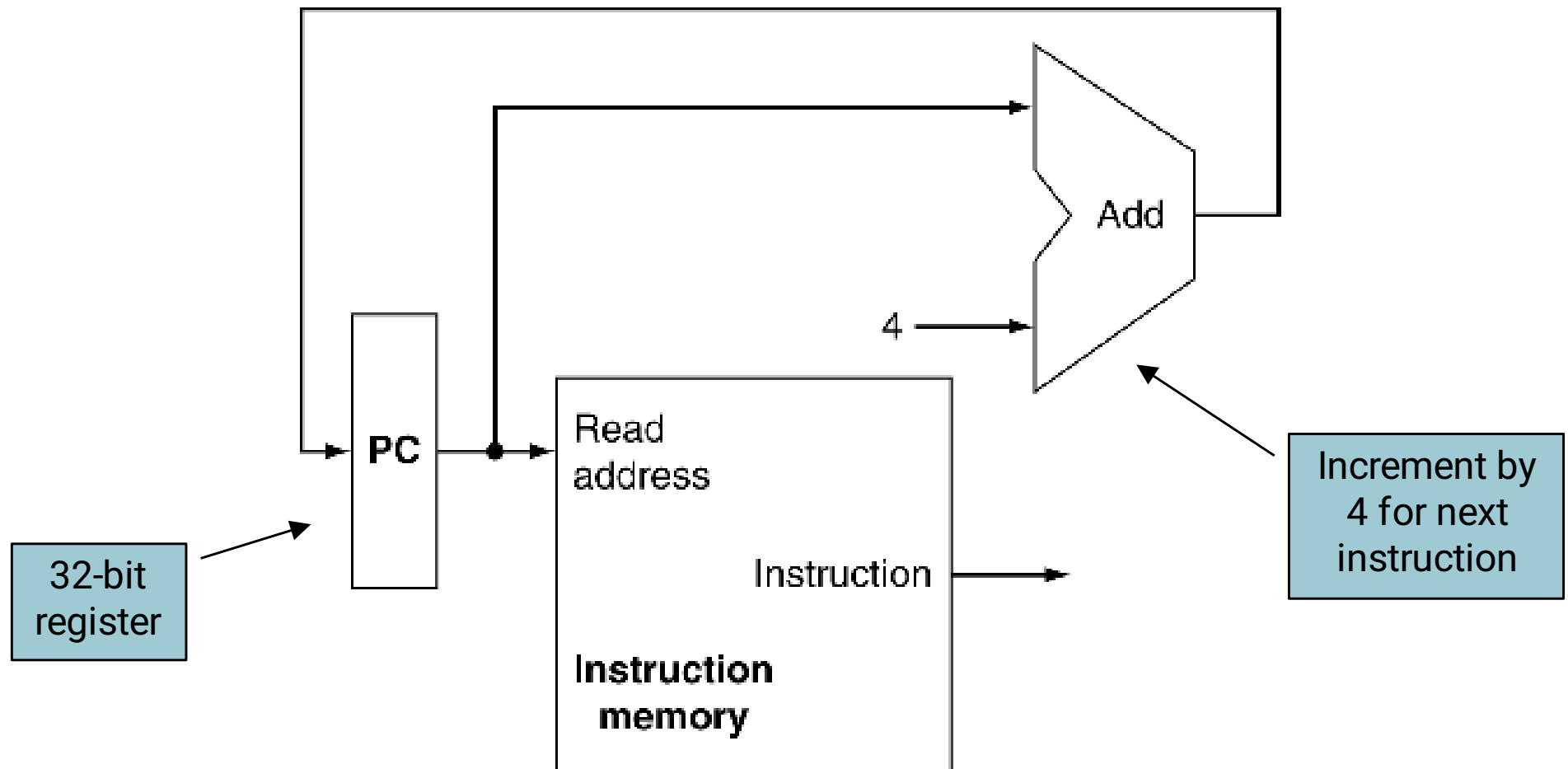
Instruction disposition



Building a Datapath

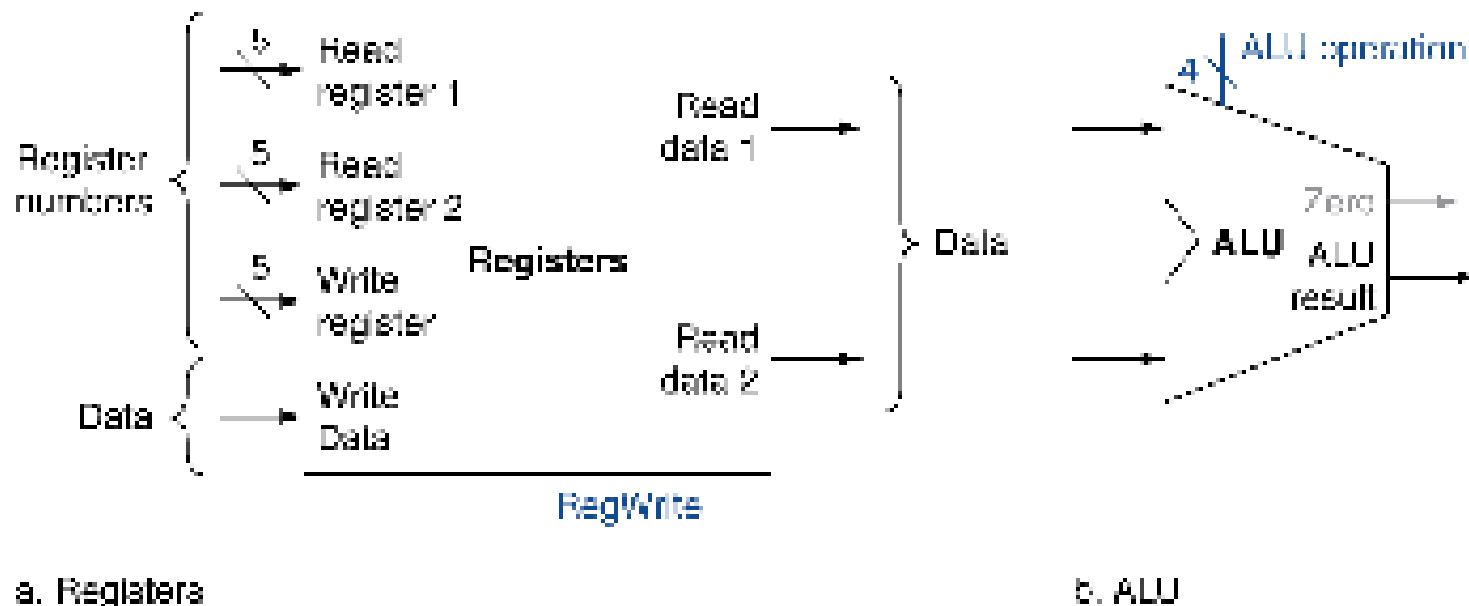
- Datapath
 - Elements that process data and addresses in the CPU
 - Registers, ALUs, mux's, memories, ...
- We will build a MIPS datapath incrementally
 - Refining the overview design

Instruction Fetch



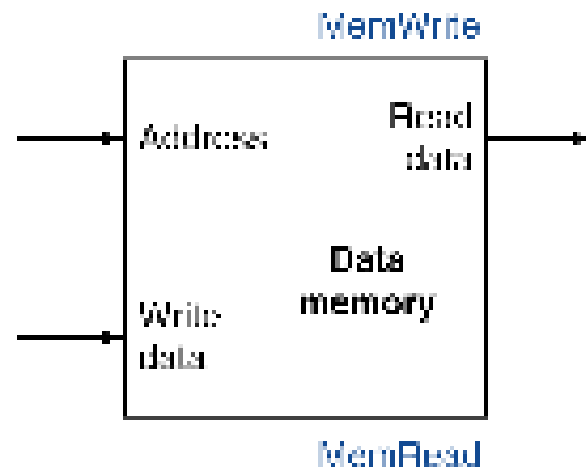
R-Format Instructions

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

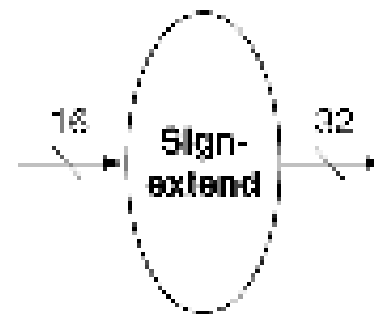


Load/Store Instructions

- 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



a. Data memory unit

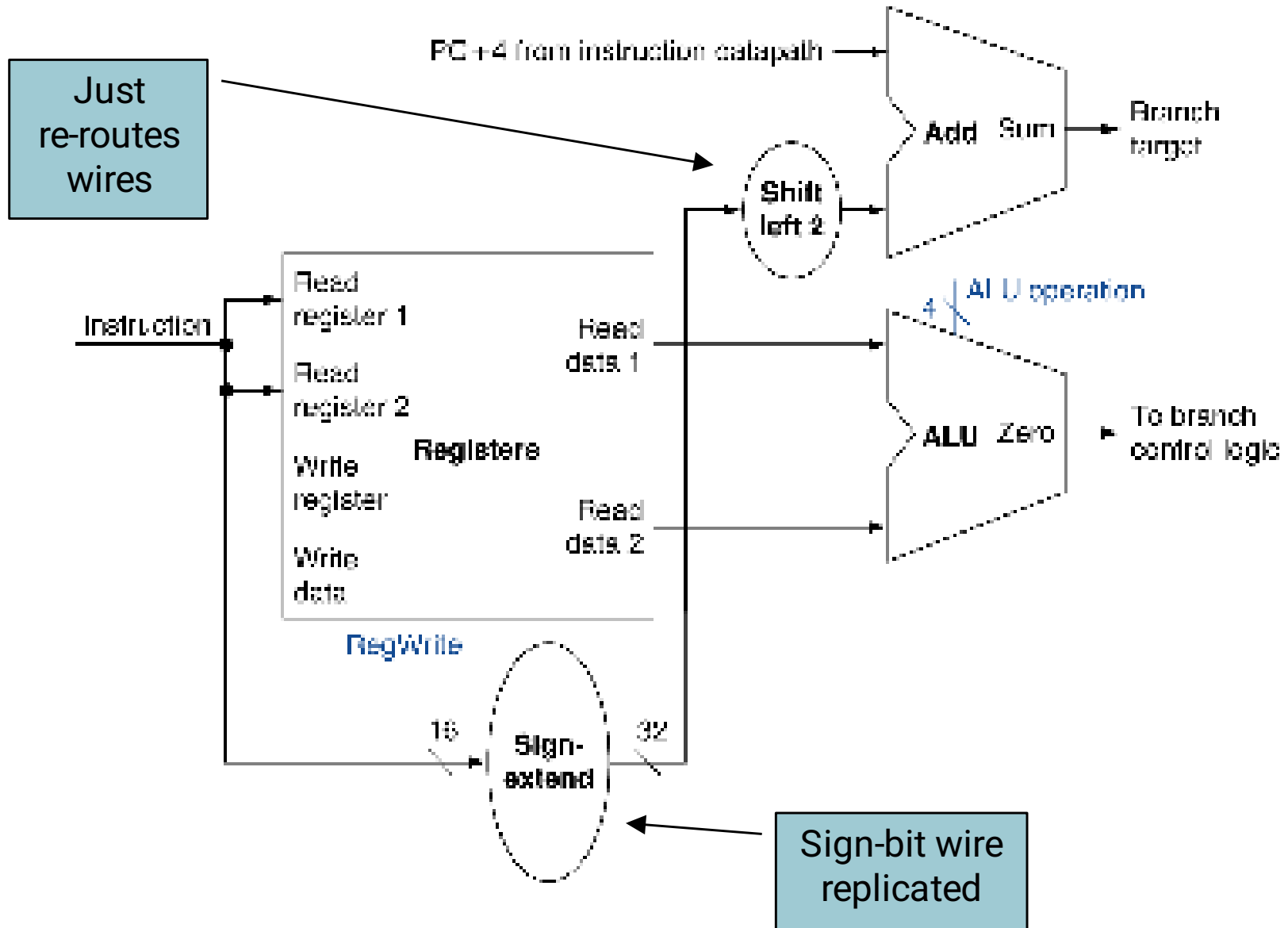


b. Sign extension unit

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

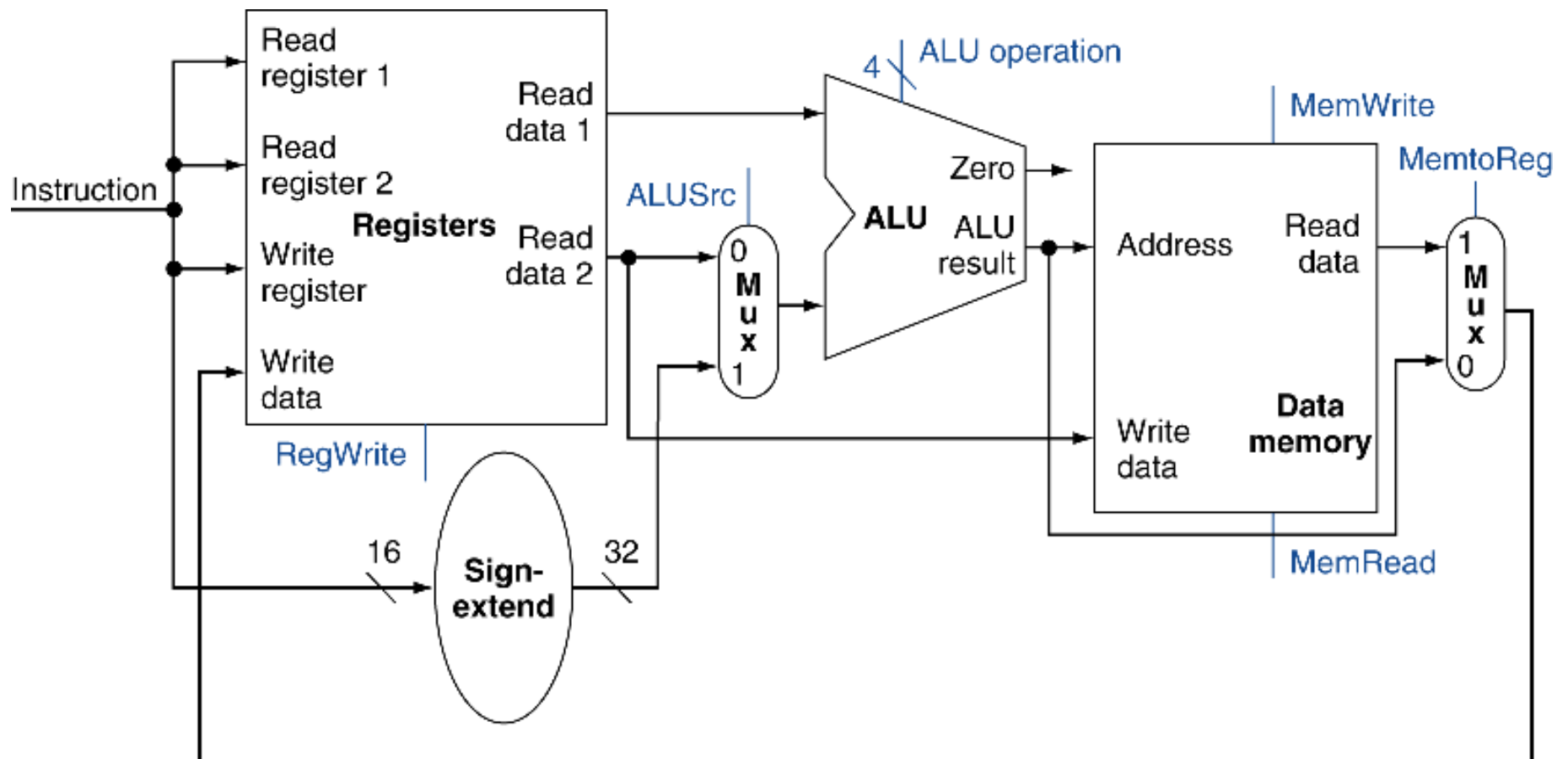
Branch Instructions



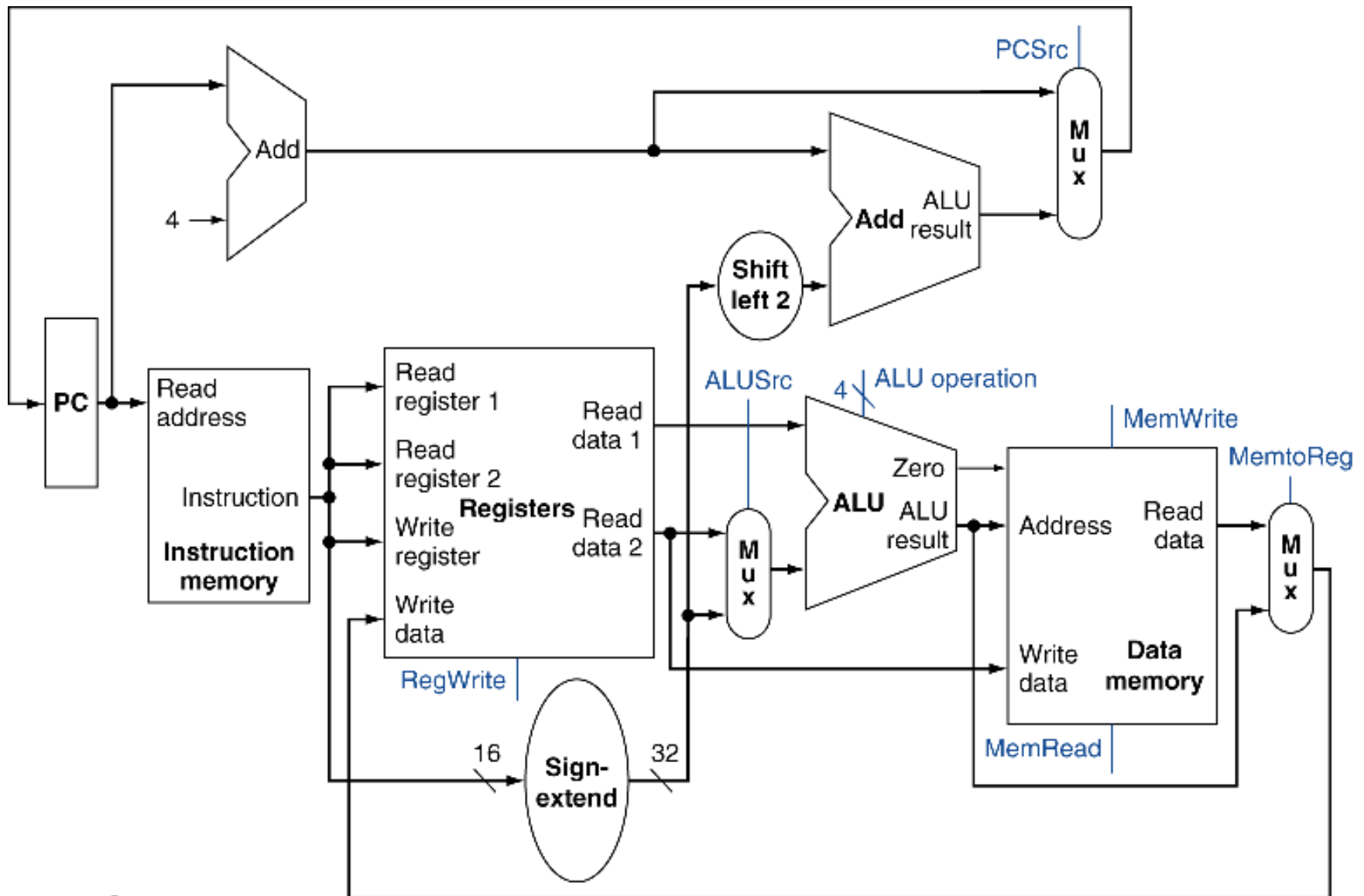
Composing the Elements

- 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 separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

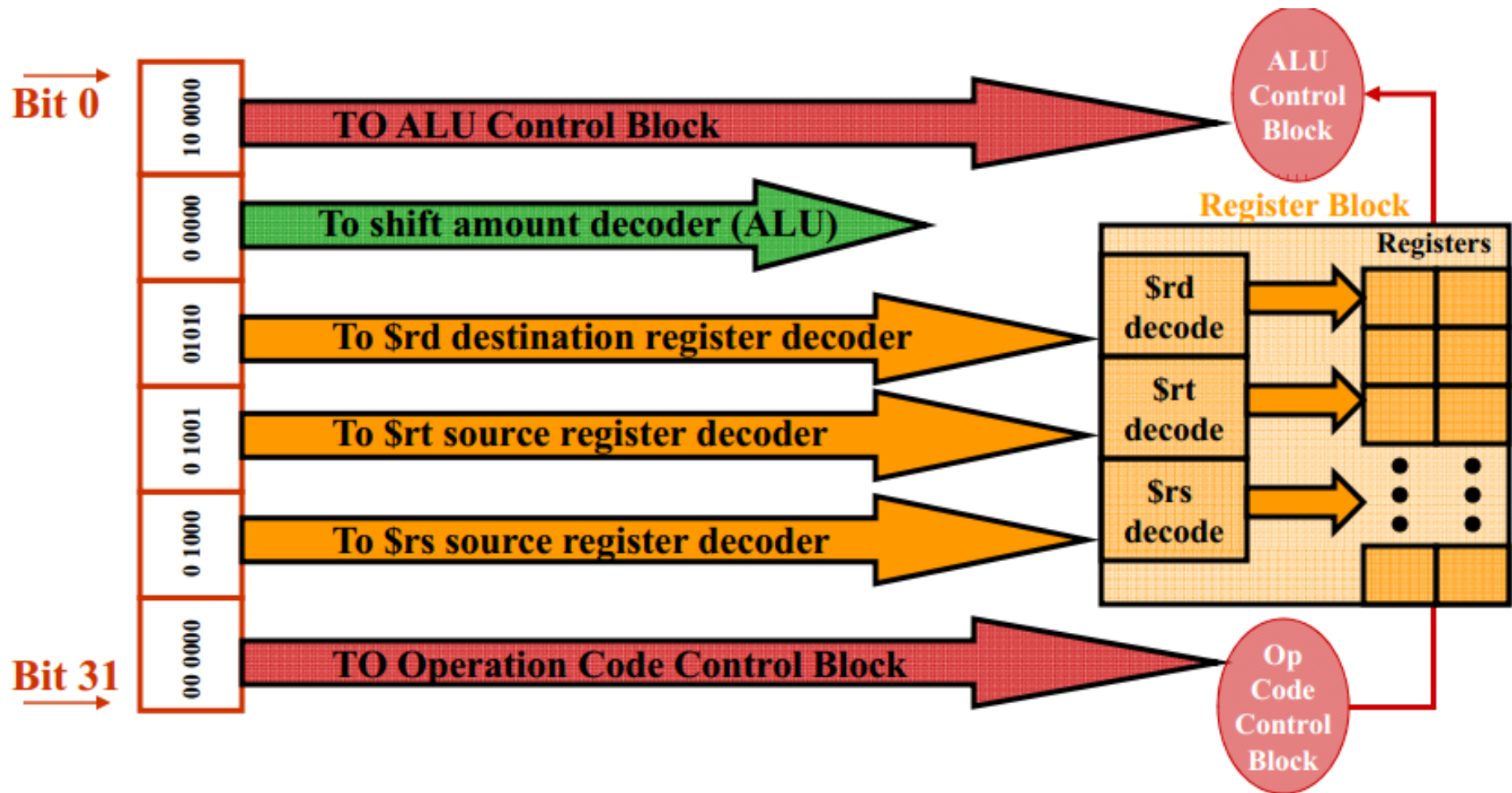
R-Type/Load/Store Datapath



Full Datapath



Instruction disposition



Control unit design

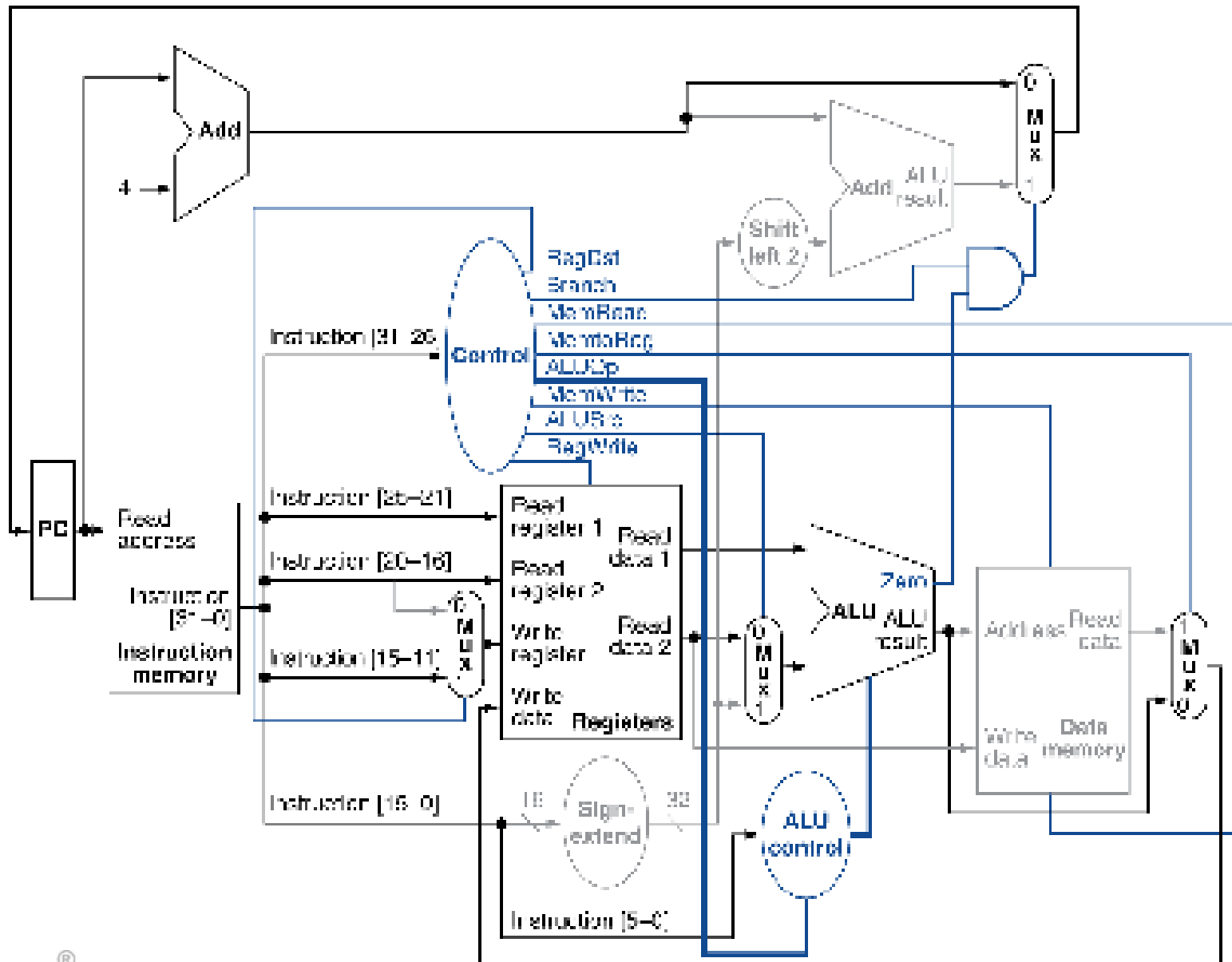
- The control unit must be able to take inputs and generate
 - A write signal for each state element,
 - Selector control for each multiplexor and
 - The ALU control.

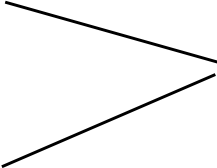
ALU Control

- 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

R-Type Instruction



- Must describe hardware to compute 4-bit ALU control input
 - given instruction type
 - 00 = lw, sw
 - 01 = beq,
 - 10 = arithmetic
 - function code for arithmetic
- ALUOp
computed from instruction type
- 

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

Truth table for ALU control bits

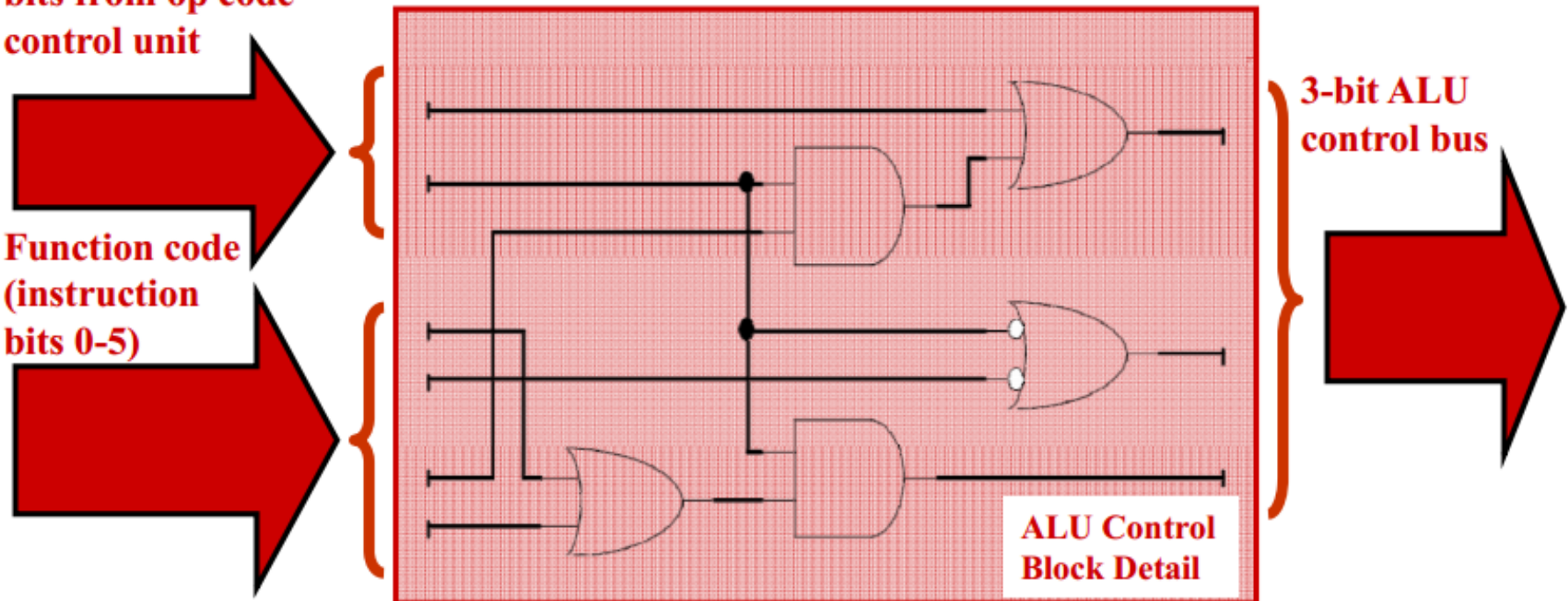
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

Sample ckt for a 3 bit ALU cntrl

ALU Control Block

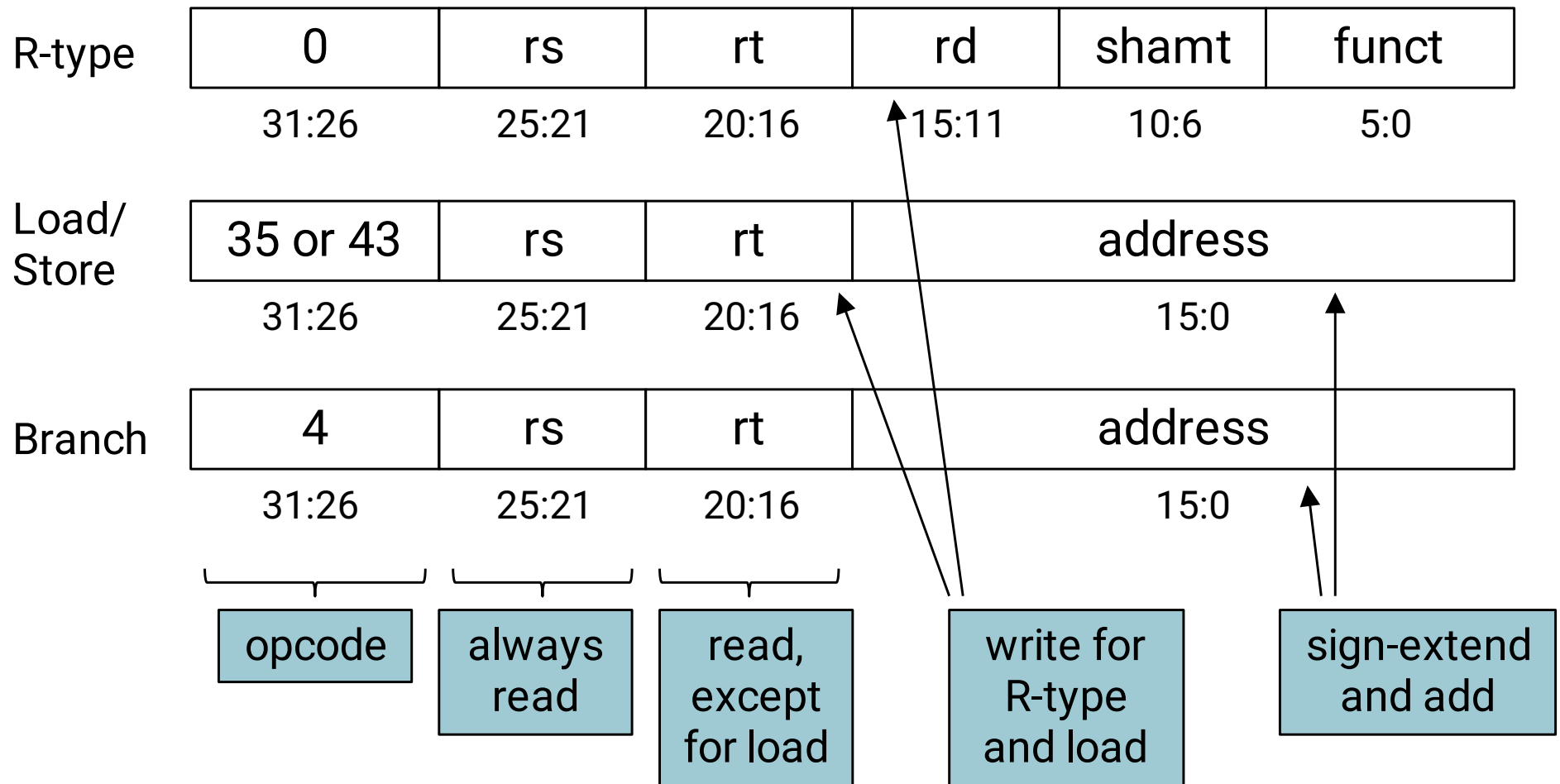
ALU operation
bits from op code
control unit

Function code
(instruction
bits 0-5)

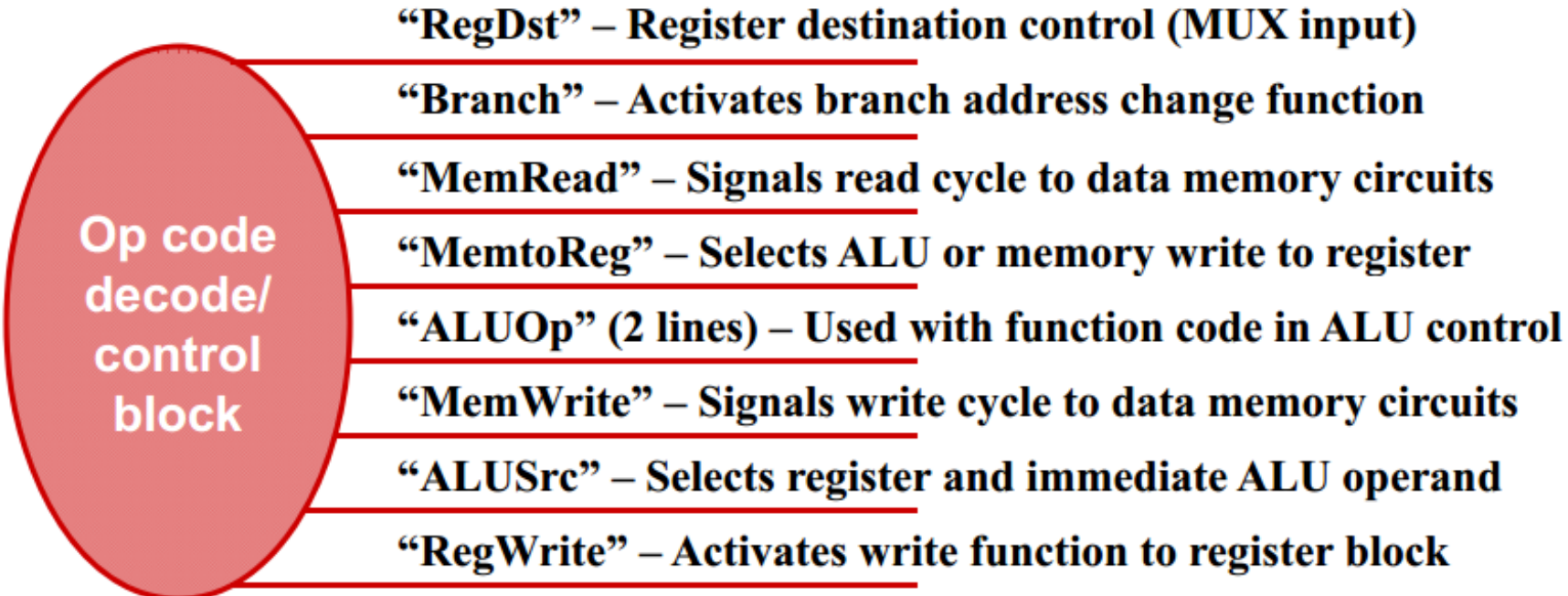


The Main Control Unit

■ Control signals derived from instruction



Main control unit



Op code
decode/
control
block

“RegDst” – Register destination control (MUX input)

“Branch” – Activates branch address change function

“MemRead” – Signals read cycle to data memory circuits

“MemtoReg” – Selects ALU or memory write to register

“ALUOp” (2 lines) – Used with function code in ALU control

“MemWrite” – Signals write cycle to data memory circuits

“ALUSrc” – Selects register and immediate ALU operand

“RegWrite” – Activates write function to register block

Effect of control signals

Signal Name	When Signal = 1	When Signal = 0
RegDst	Write reg. = \$rd (bits 11–15)	Write reg. = \$rt (bits 16–20)
Branch	ALU branch compare activated	No branch activated
MemRead	Memory data → write register	No data read from memory
MemtoReg	Memory data → write register	ALU results → write register
ALUOp	NA; lines go to ALU control block	NA; lines go to ALU control block
MemWrite	ALU or register data → memory	No data written to memory
ALUSrc	2nd ALU operand is immediate (sign-extended instr. bits 0–15)	2nd ALU operand is from \$rt (instruction bits 16–20)
RegWrite	Memory/ALU data → write reg.	No input to register block

Setting of control signals through opcode fields

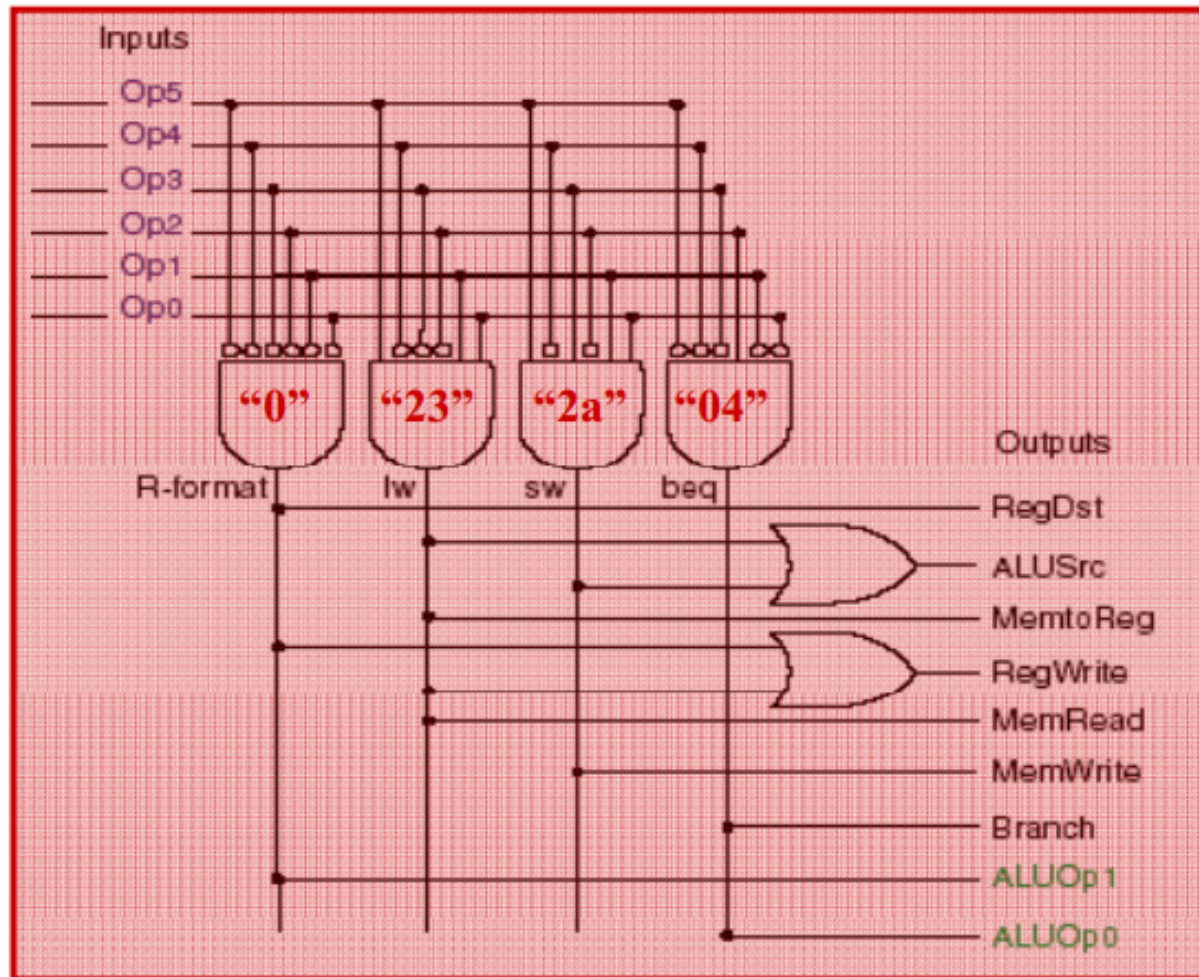
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Truth table for control functions

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
	ALUOp0	0	0	0	1

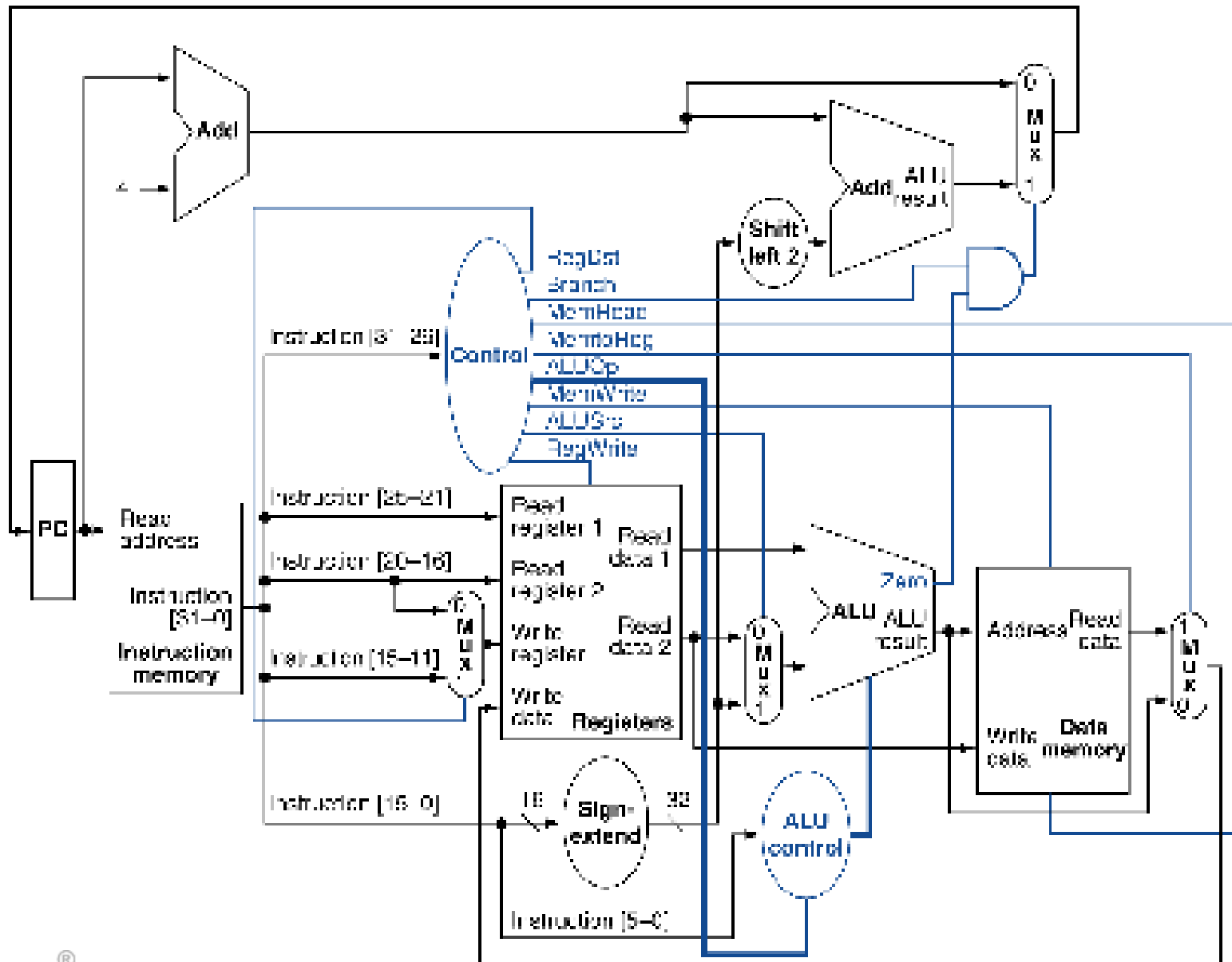
Main control unit

Instruction
bits 26-31

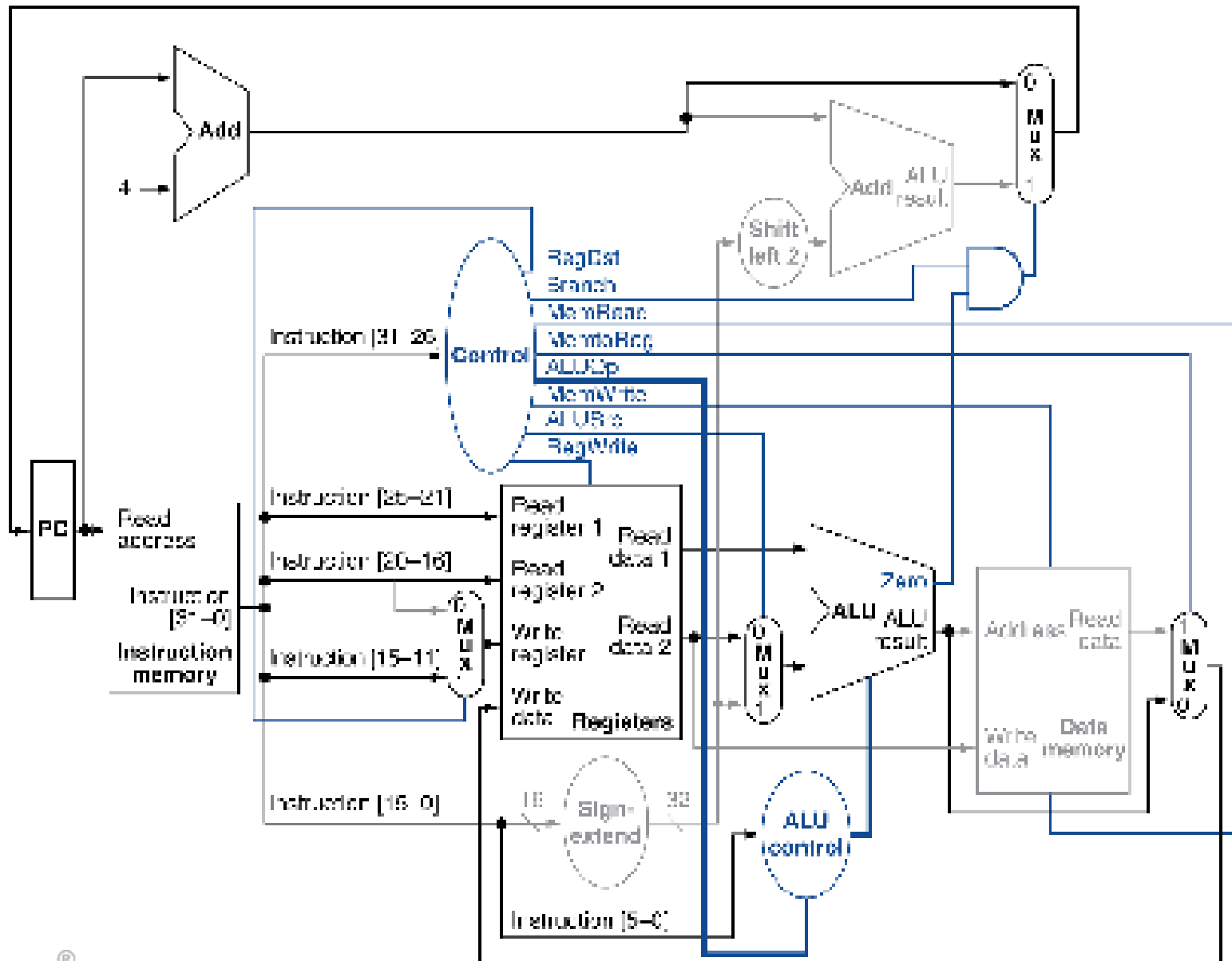


TO ALU
control block

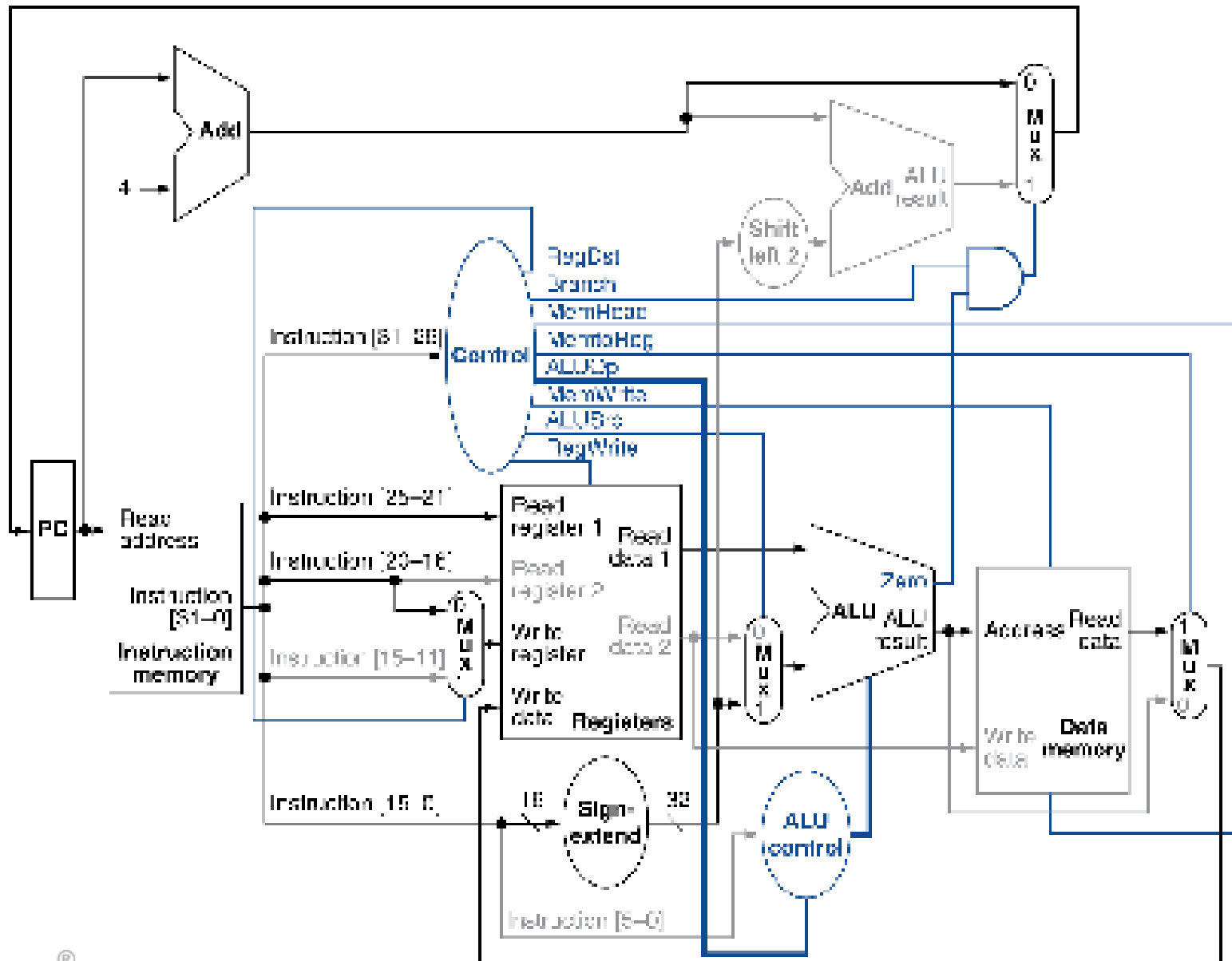
Datapath With Control



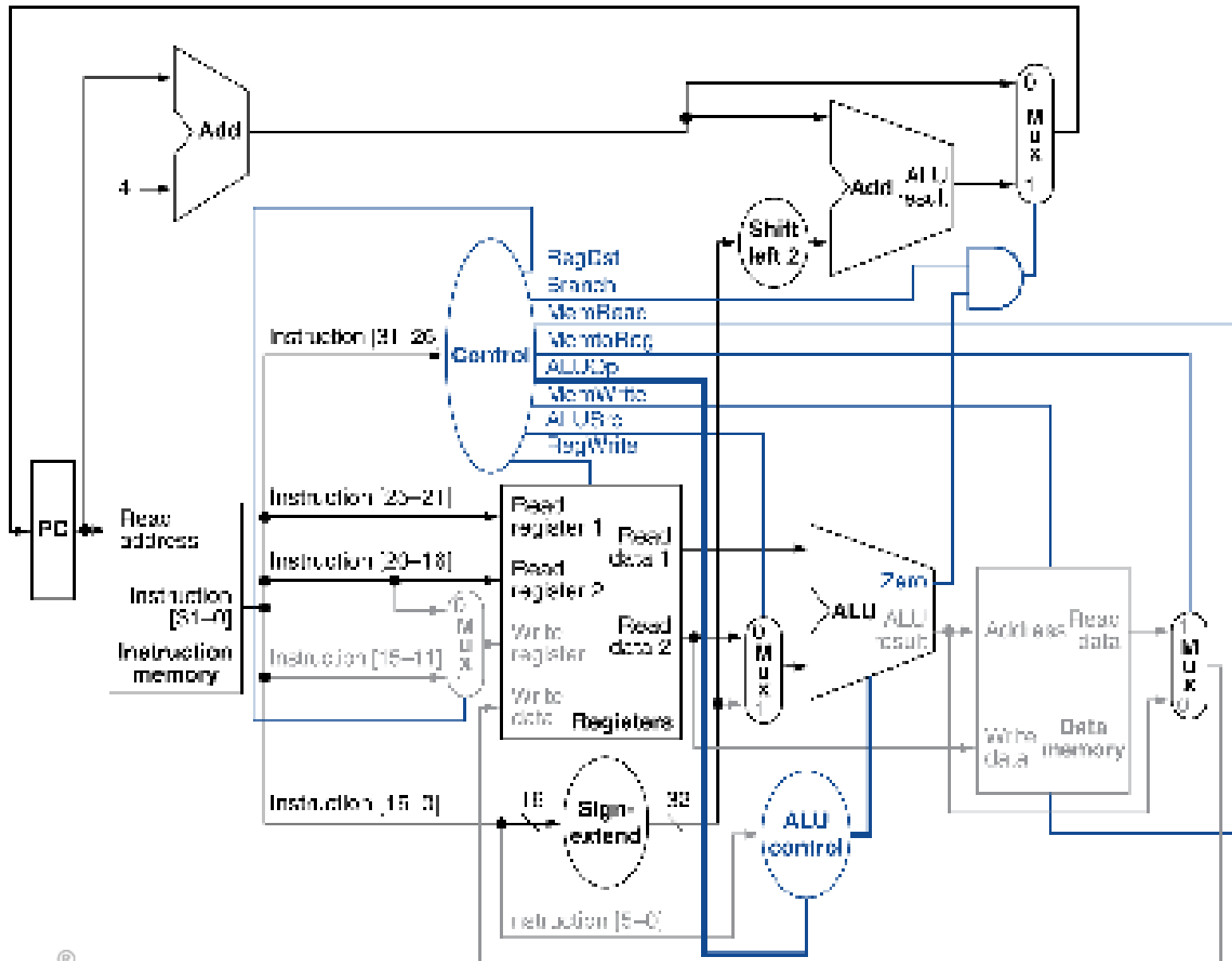
R-Type Instruction



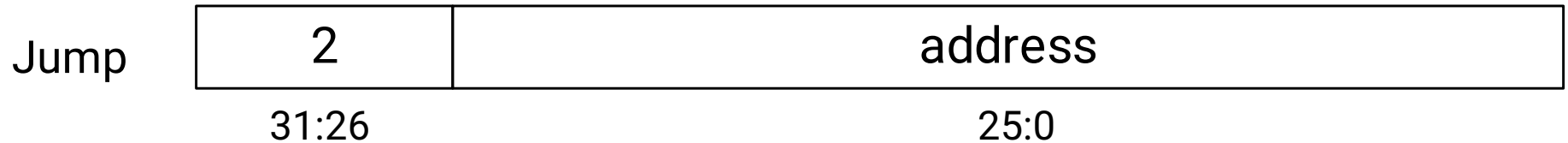
Load Instruction



Branch-on-Equal Instruction

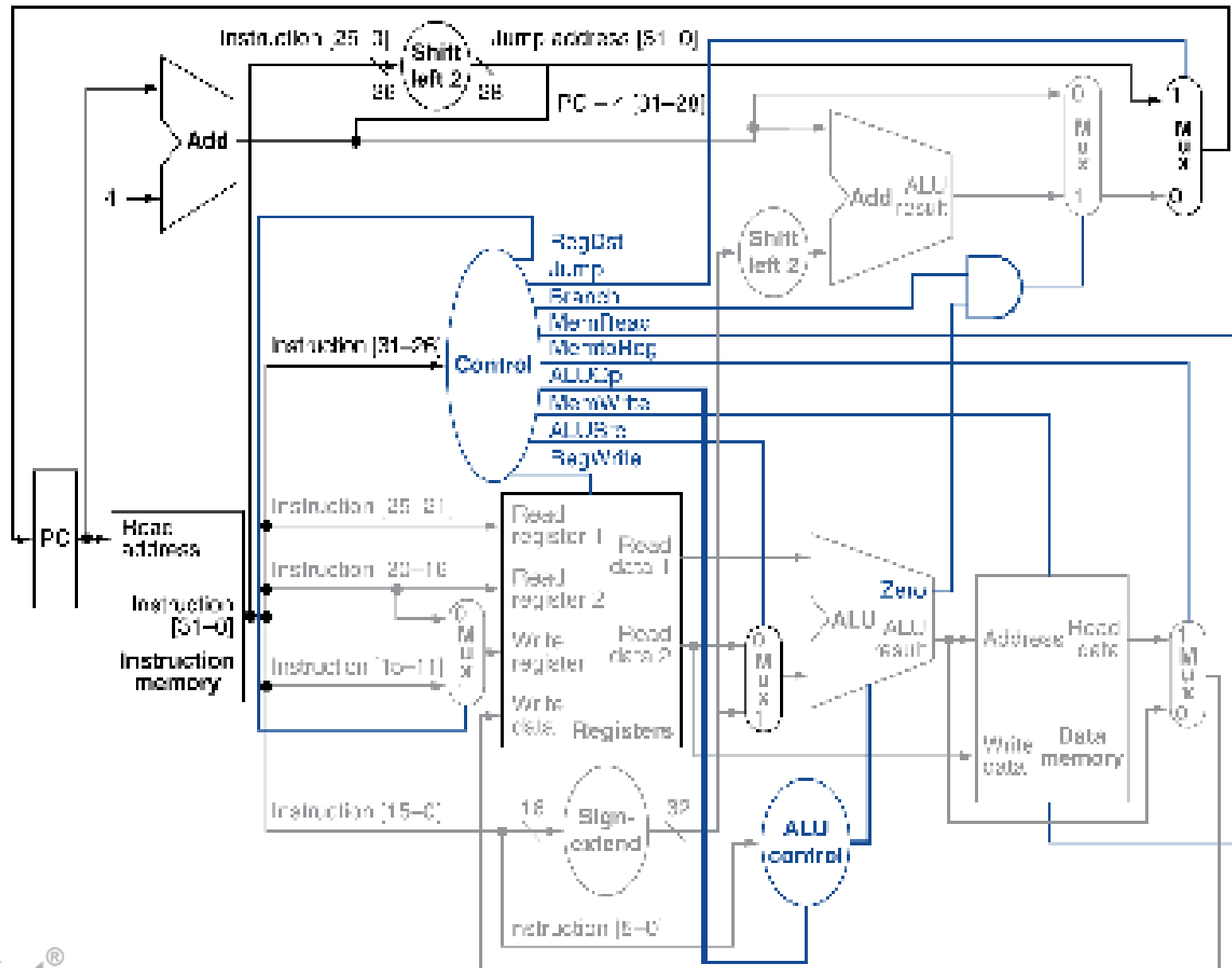


Implementing Jumps



- 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



Performance Issues

- The clock is determined by the longest possible path in the processor
 - 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
 - Clock cycle is the worst case delay for all instructions

Clock cycle time

- Assume time for stages is
 - 100ps for register read or write
 - 200ps for other stages

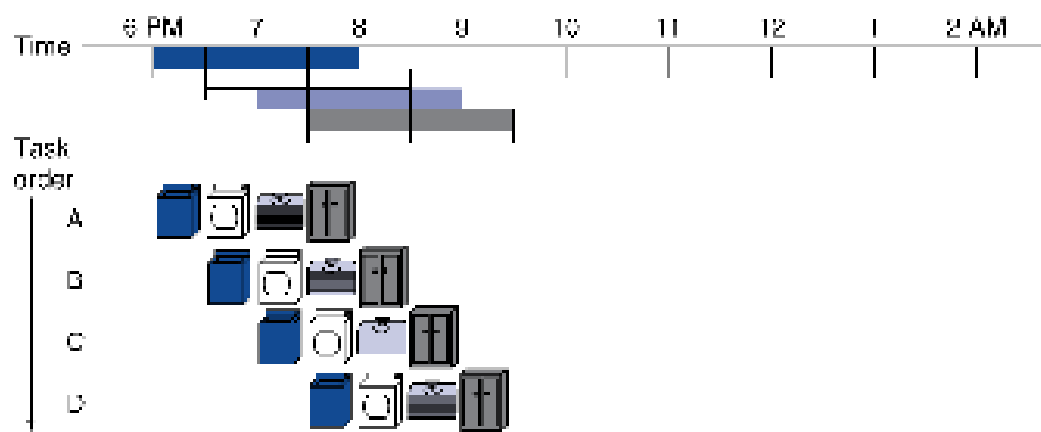
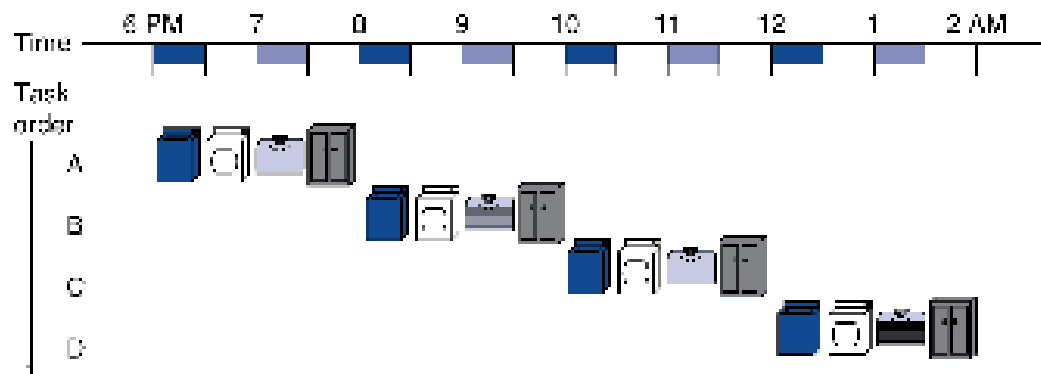
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

pipelining

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution
- Today pipelining is key to making processors fast
- People who has done a lot of laundry has intuitively used pipelining

Pipelining Analogy

- Pipelined laundry: overlapping execution
 - Parallelism improves performance



- Four loads:

- Speedup
 $= 8 / 3.5 = 2.3$

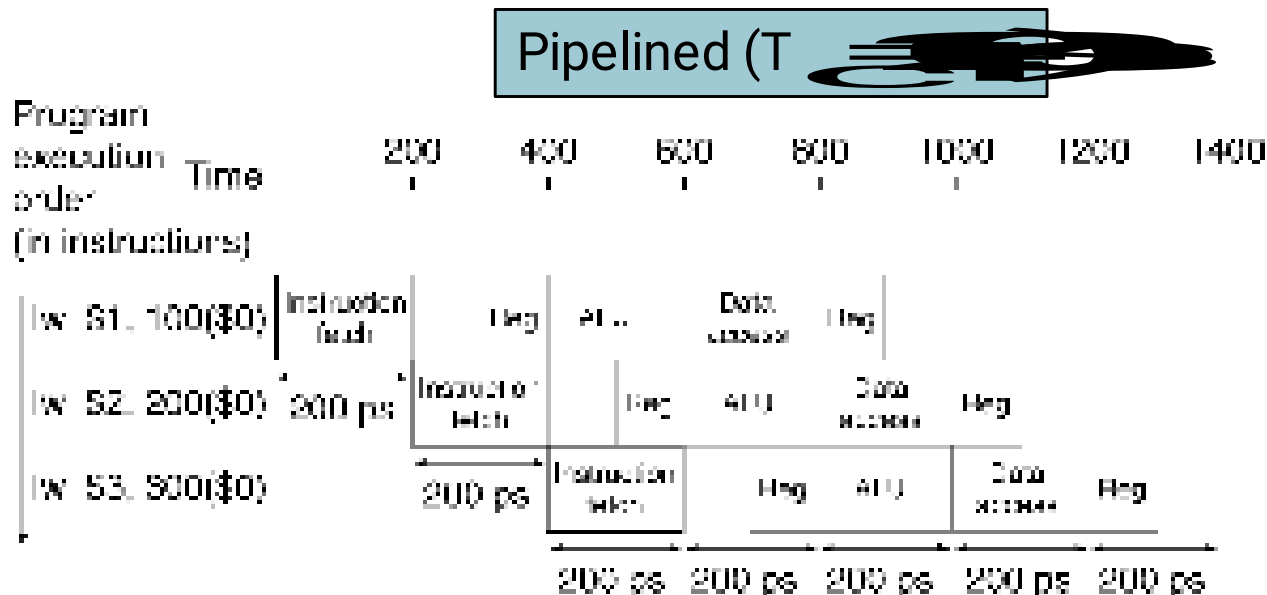
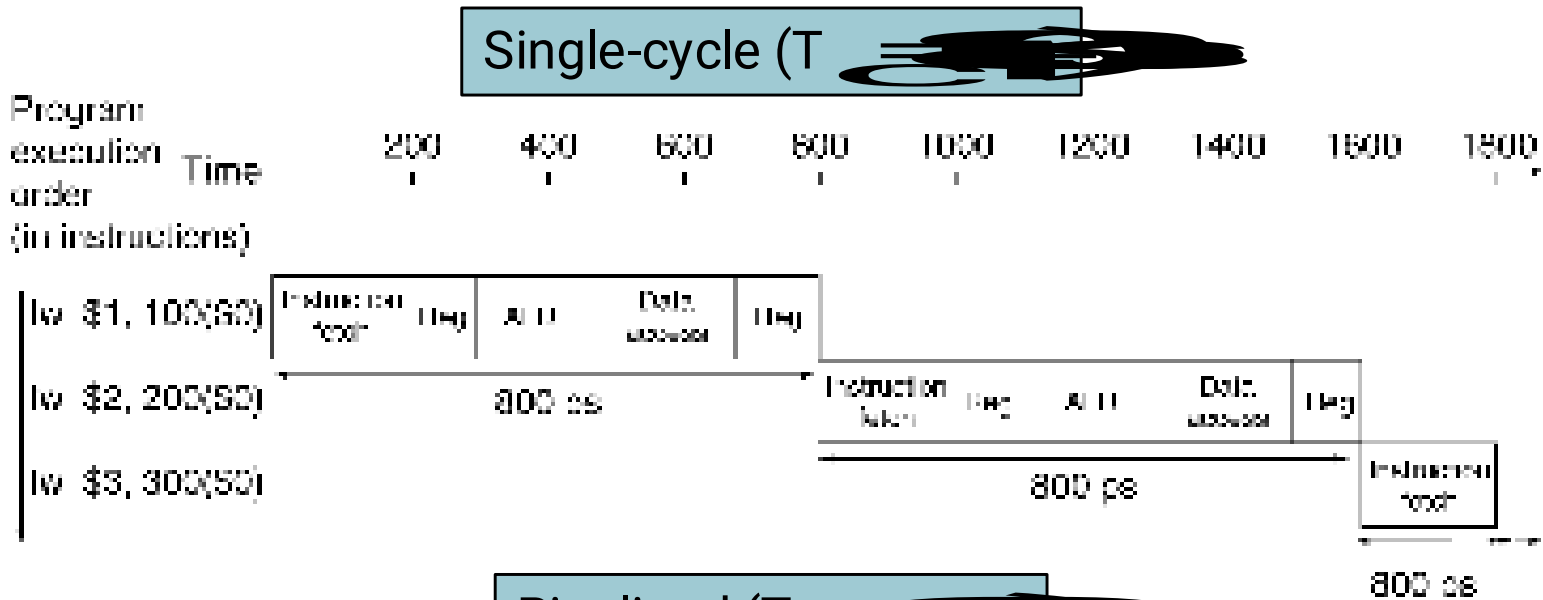
- Non-stop:

- Speedup
 $= 2n / 0.5n + 1.5 \approx 4$
 $= \text{number of stages}$

MIPS Pipeline

- Same principles apply to processors where we pipeline instruction execution
- MIPS instruction classically take 5 steps.
- 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





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 Speedup

- If all stages are balanced
 - i.e., all take the same time
 - Time between instructions 
= $\frac{\text{Time between instructions}}{\text{Number of stages}}$ 
 - *potential* speedup = number of pipe stages
- If not balanced, speedup is less
- Speedup due to increased throughput
 - Latency (time for each instruction) does not decrease

What happens when we increase no of instructions(1,000,000)

Pipelining MIPS

- What makes it easy with MIPS?
 - *all instructions are same length*
 - so fetch and decode stages are similar for all instructions
 - *just a few instruction formats*
 - simplifies instruction decode and makes it possible in one stage
 - *memory operands appear only in load/stores*
 - so memory access can be deferred to exactly one later stage
 - *operands are aligned in memory*
 - one data transfer instruction requires one memory access stage

Pipelining: Keep in Mind

- Pipelining *does not reduce latency* of a single task, it *increases throughput* of entire workload
- Pipeline rate *limited by longest stage*
 - *potential* speedup = number pipe stages
 - *unbalanced lengths* of pipe stages reduces speedup
- Time to *fill* pipeline and time to *drain* it – when there is *slack* in the pipeline – reduces speedup

Pipelining MIPS

- What makes it hard?
 - **structural hazards**: different instructions, at different stages, in the pipeline want to use the same hardware resource
 - **control hazards**: succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction, already in pipeline
 - **data hazards**: an instruction in the pipeline requires data to be computed by a previous instruction still in the pipeline
- Before actually building the pipelined datapath and control we first briefly examine these potential hazards individually...

Hazards

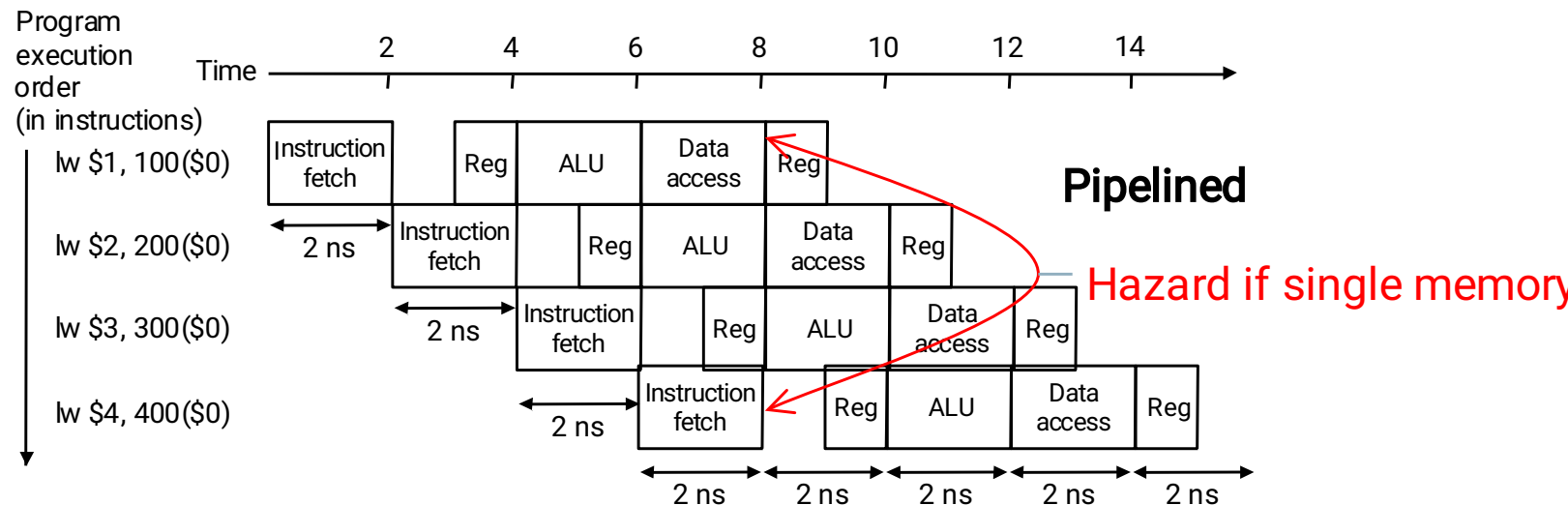
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
 - A required resource is busy
- Data hazard
 - Need to wait for previous instruction to complete its data read/write
- Control hazard
 - Deciding on control action depends on previous instruction

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

Structural Hazards

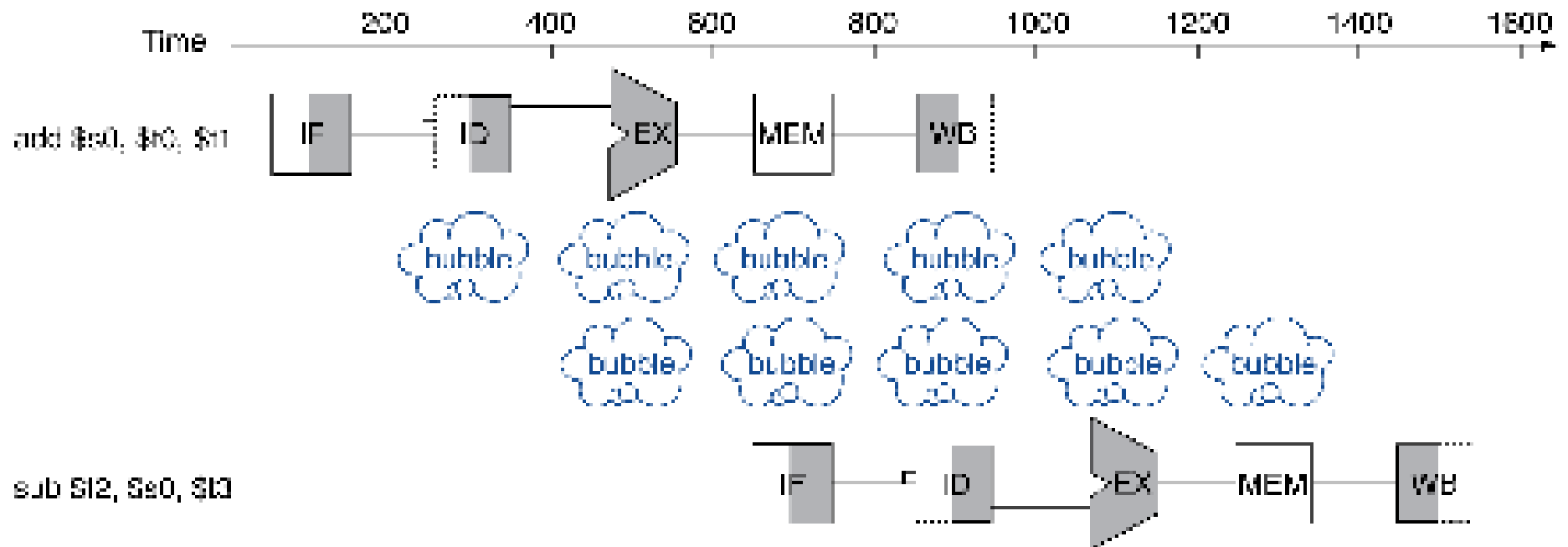
- *Structural hazard*: inadequate hardware to simultaneously support all instructions in the pipeline in the same clock cycle
- E.g., suppose *single – not separate* – instruction and data memory in pipeline below with *one read port*
 - then a structural hazard between first and fourth lw instructions



- *MIPS was designed to be pipelined*: structural hazards are easy to avoid!

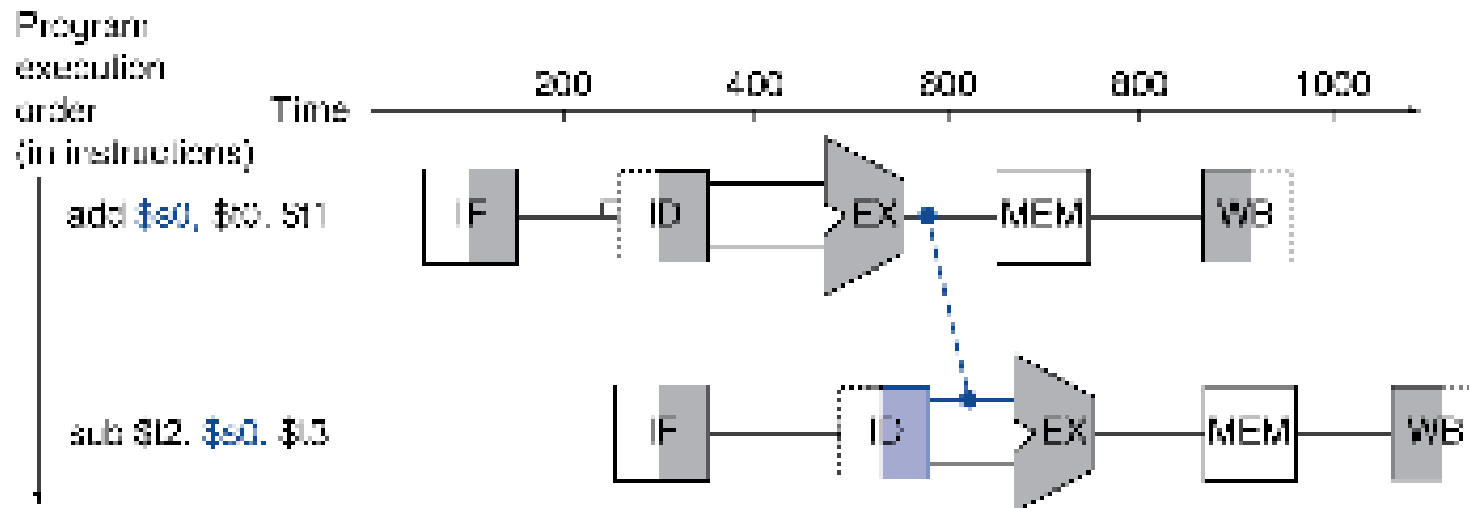
Data Hazards

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



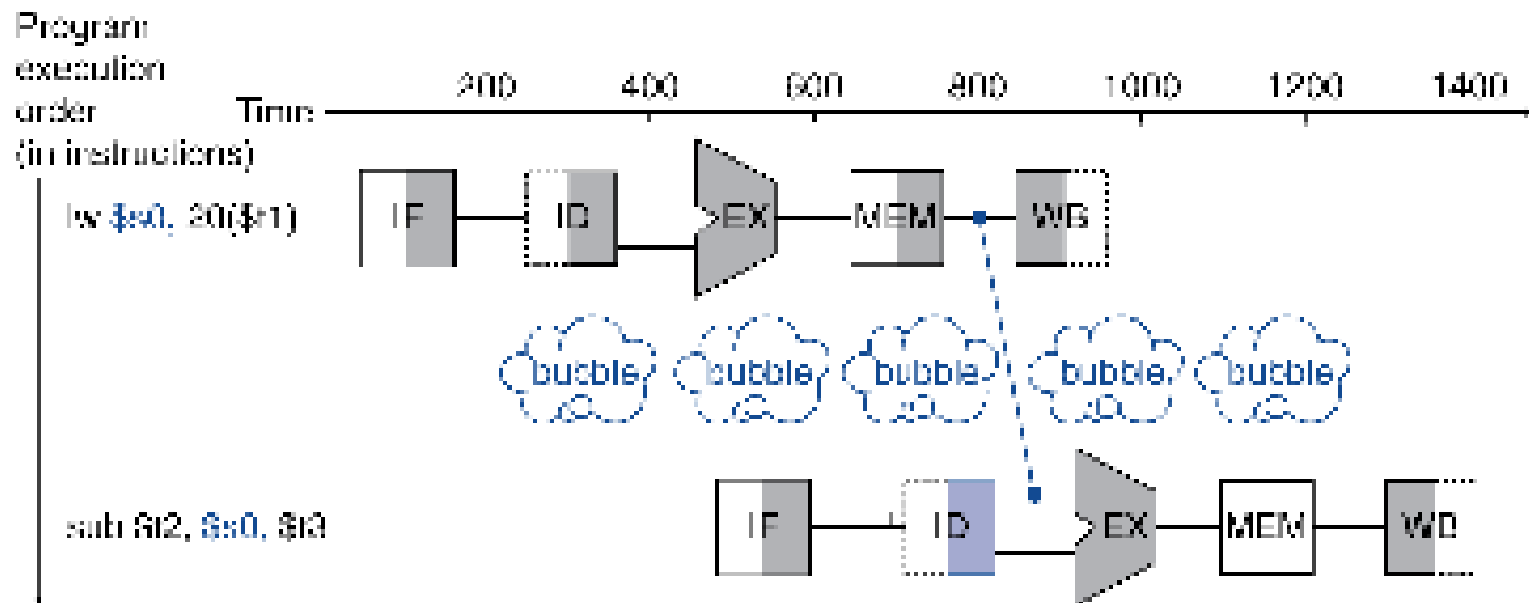
Forwarding (aka Bypassing)

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



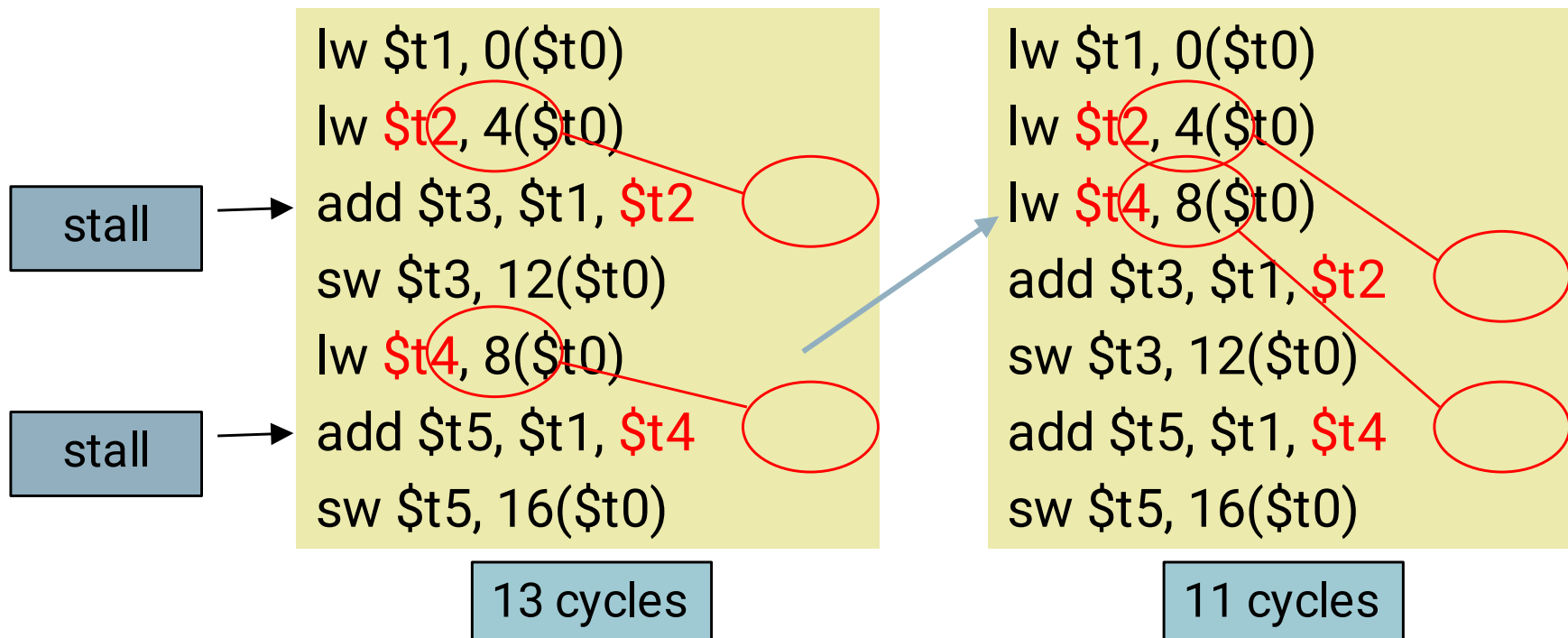
Load-Use Data Hazard

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

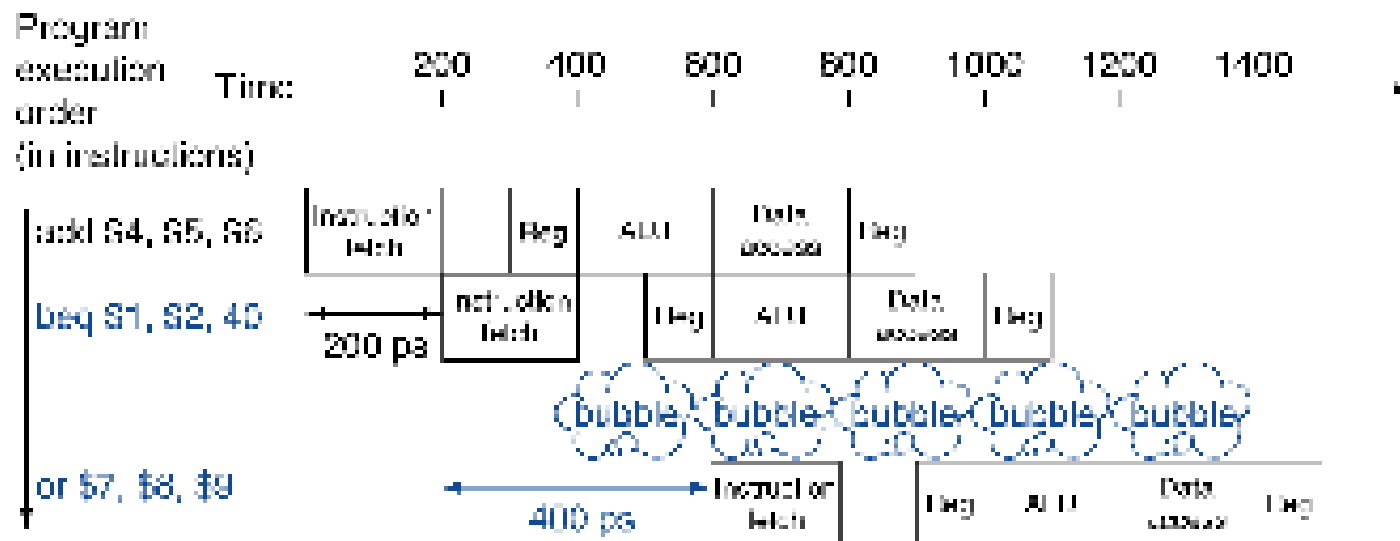


Control Hazards

- 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

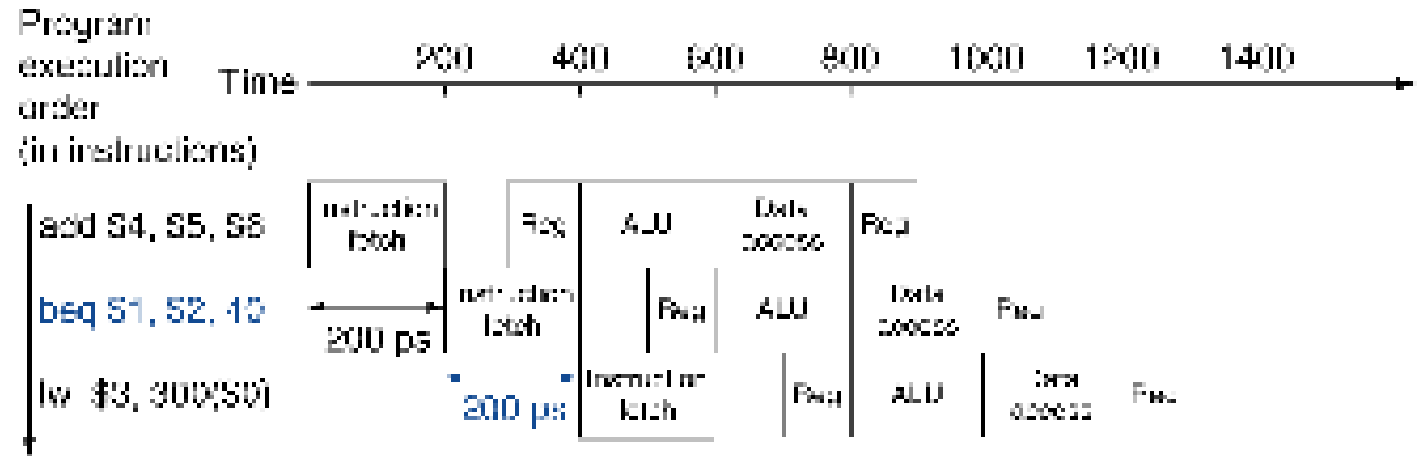


Branch Prediction

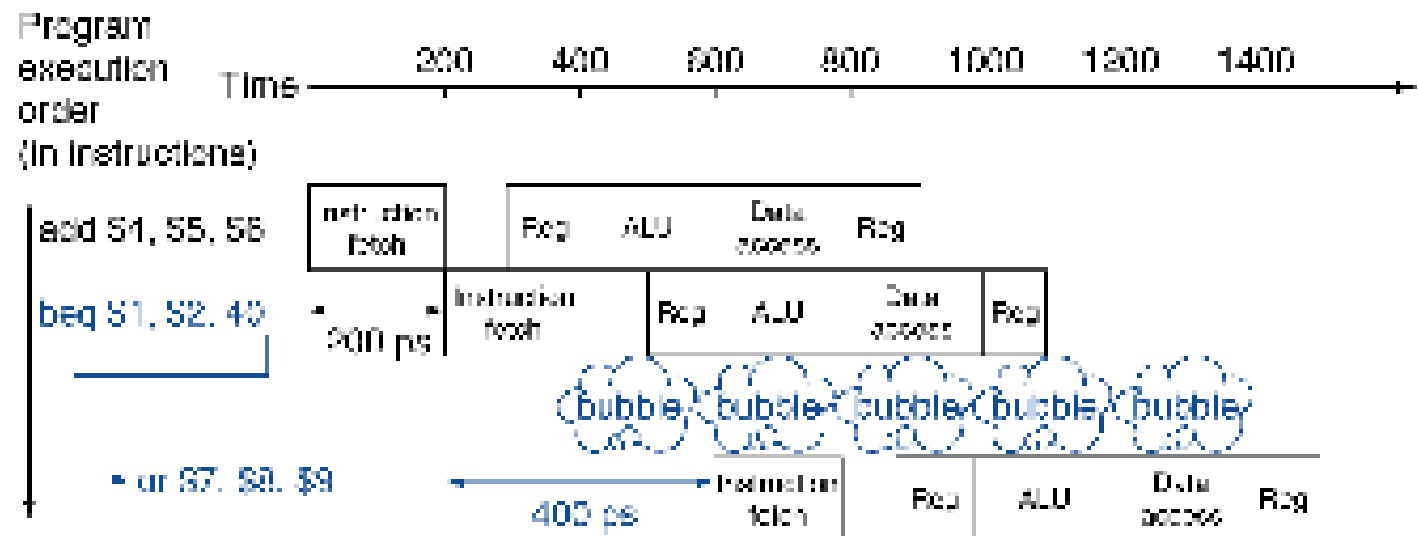
- Longer pipelines can't readily determine branch outcome early
 - Stall penalty becomes unacceptable
- **Predict outcome of branch**
 - Only stall if prediction is wrong
- In MIPS pipeline
 - Can predict branch outcome as **not taken**
 - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



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
- Pipelining is a technique that exploits parallelism among the instructions in a sequential instruction stream.

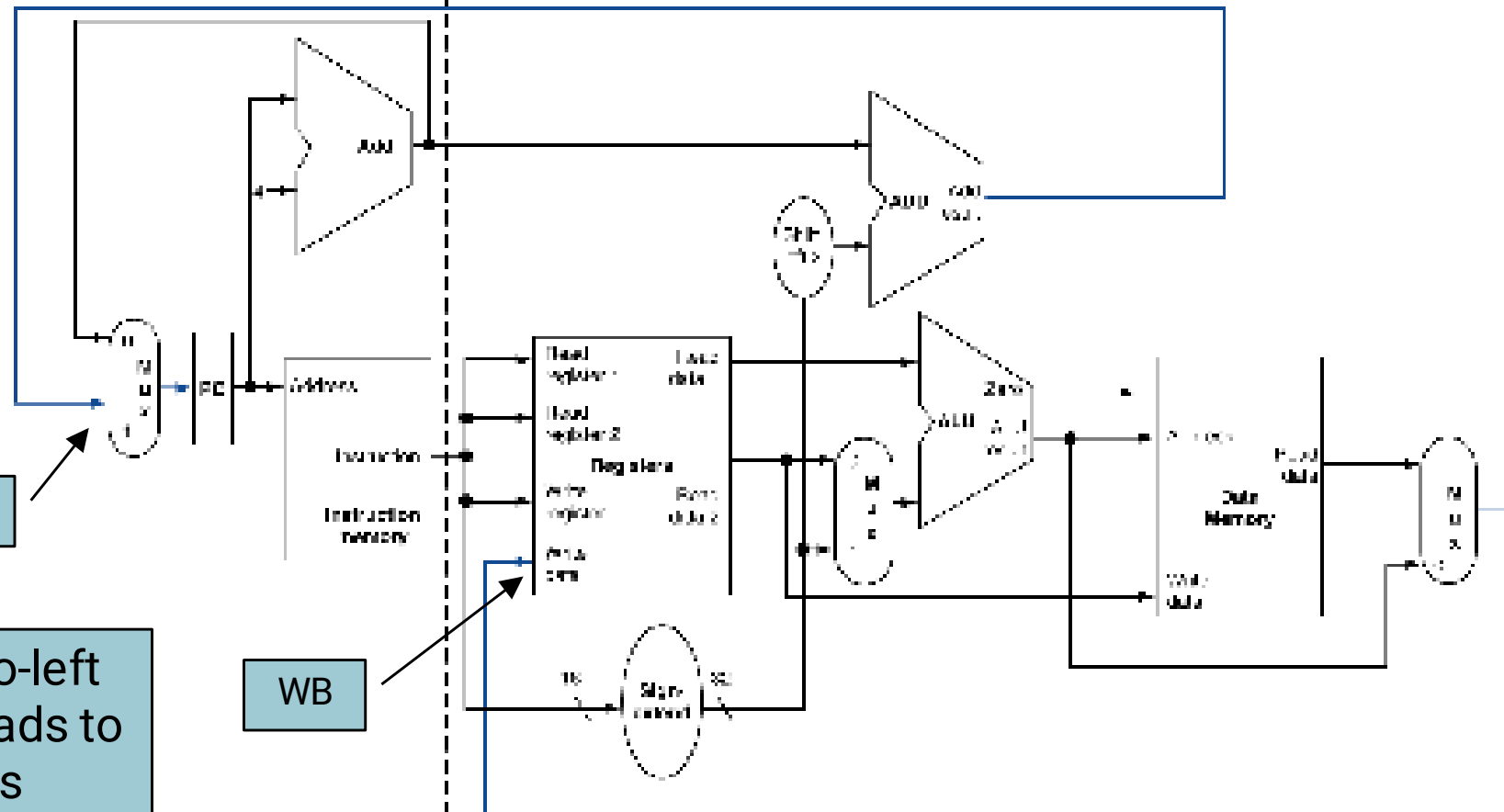
MIPS Pipelined Datapath

IF: Instruction fetch

ID: Instruction decoded/
register file readEX: Execute/
arithmetic/logic operation

MEM: Memory access

WB: Write back



MEM

WB

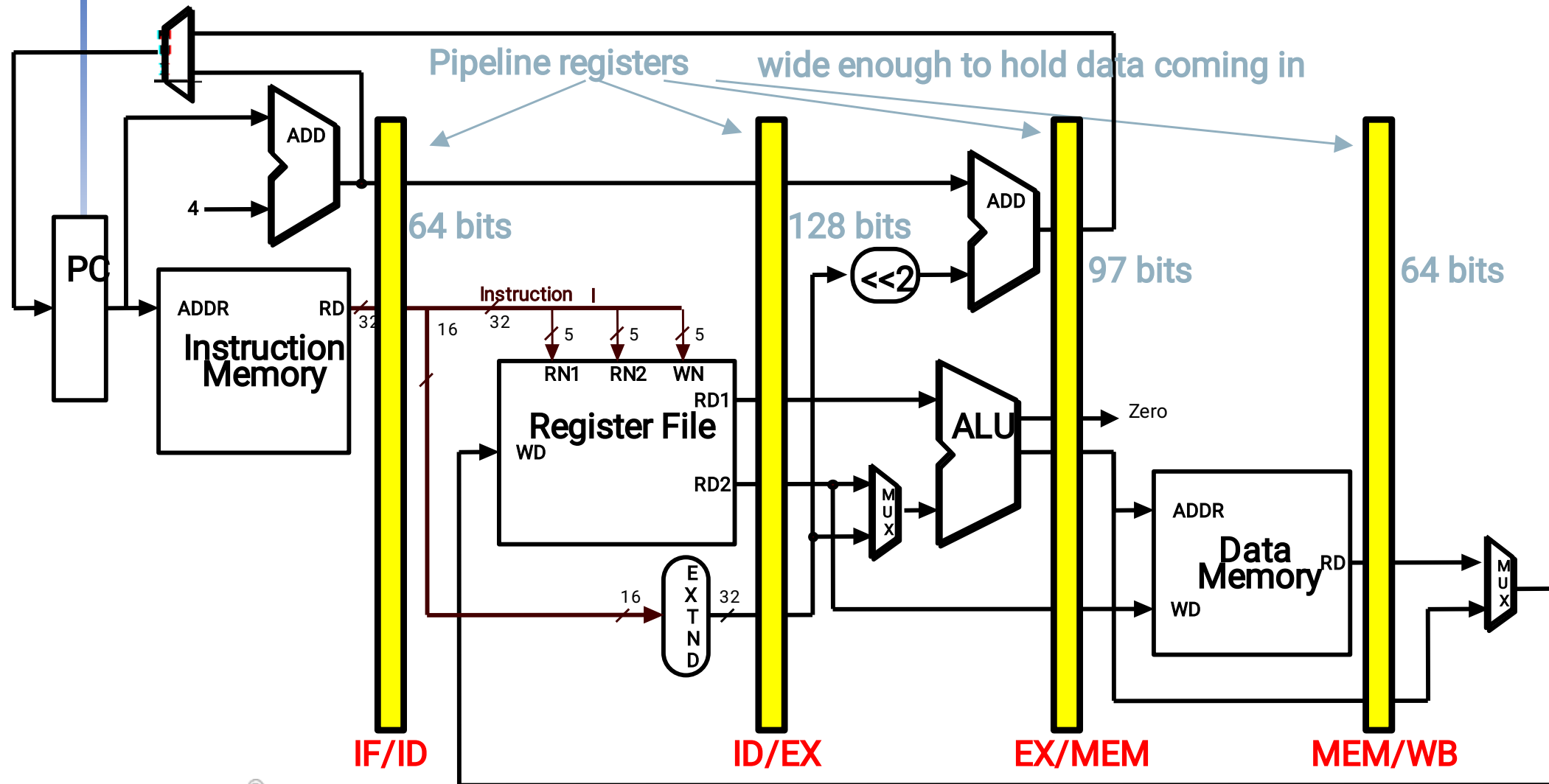
Right-to-left
flow leads to
hazards



MIPS pipeline data path

- Division of an instruction into 5 stages means a **five stage pipeline** which in turn means that up to 5 instructions will be in execution during any single clock cycle
- Instructions and data move generally from **left to right** through the 5 stages as they complete execution

Pipelined Datapath

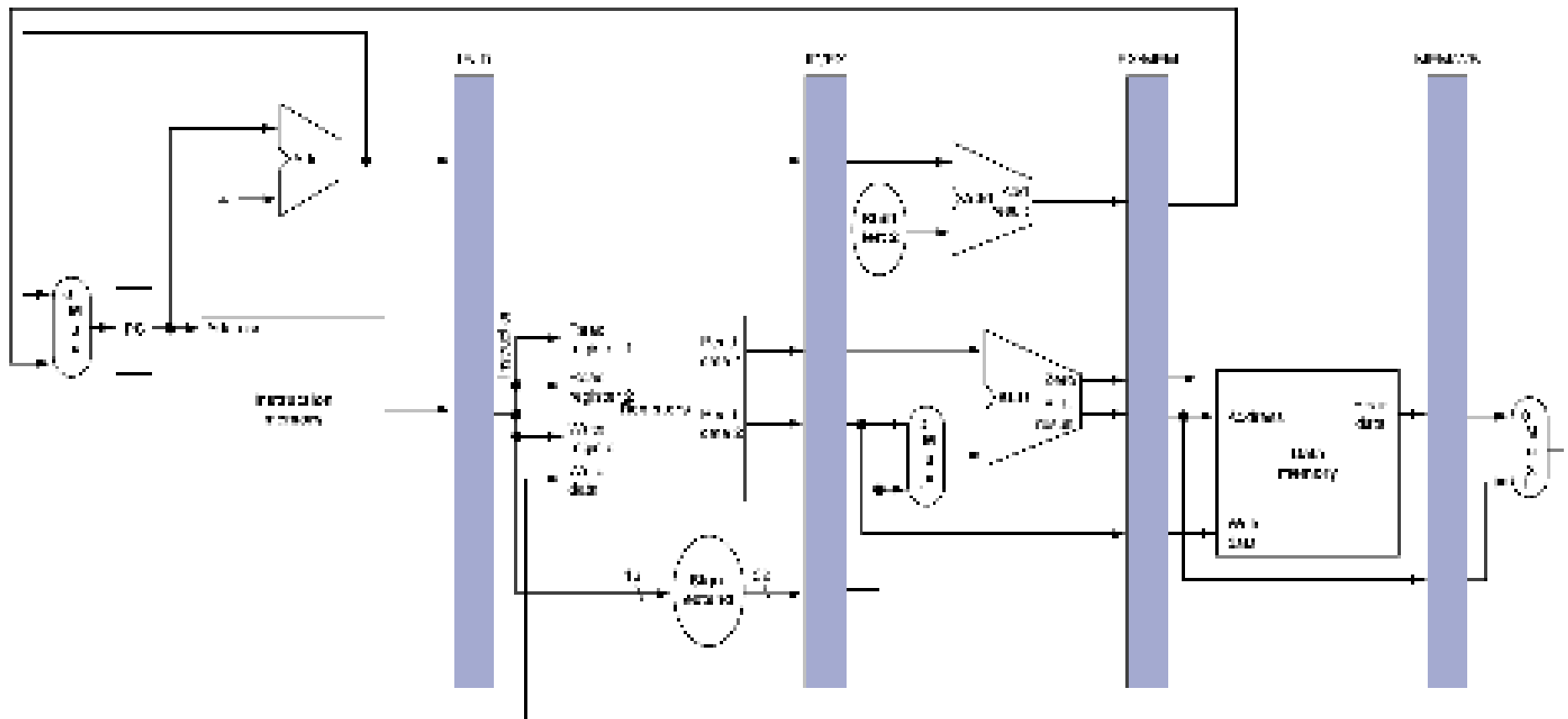


MIPS pipeline data path

- There are 2 exceptions to this left to right flow
 - The WB stage which places the result into reg
 - Selection of next value of PC
- Data flow from right to left doesn't affect the current instruction.
- Later instruction in the pipe are influenced
 - WB right to left move causes data hazards
 - Second leads to control hazards

Pipeline registers

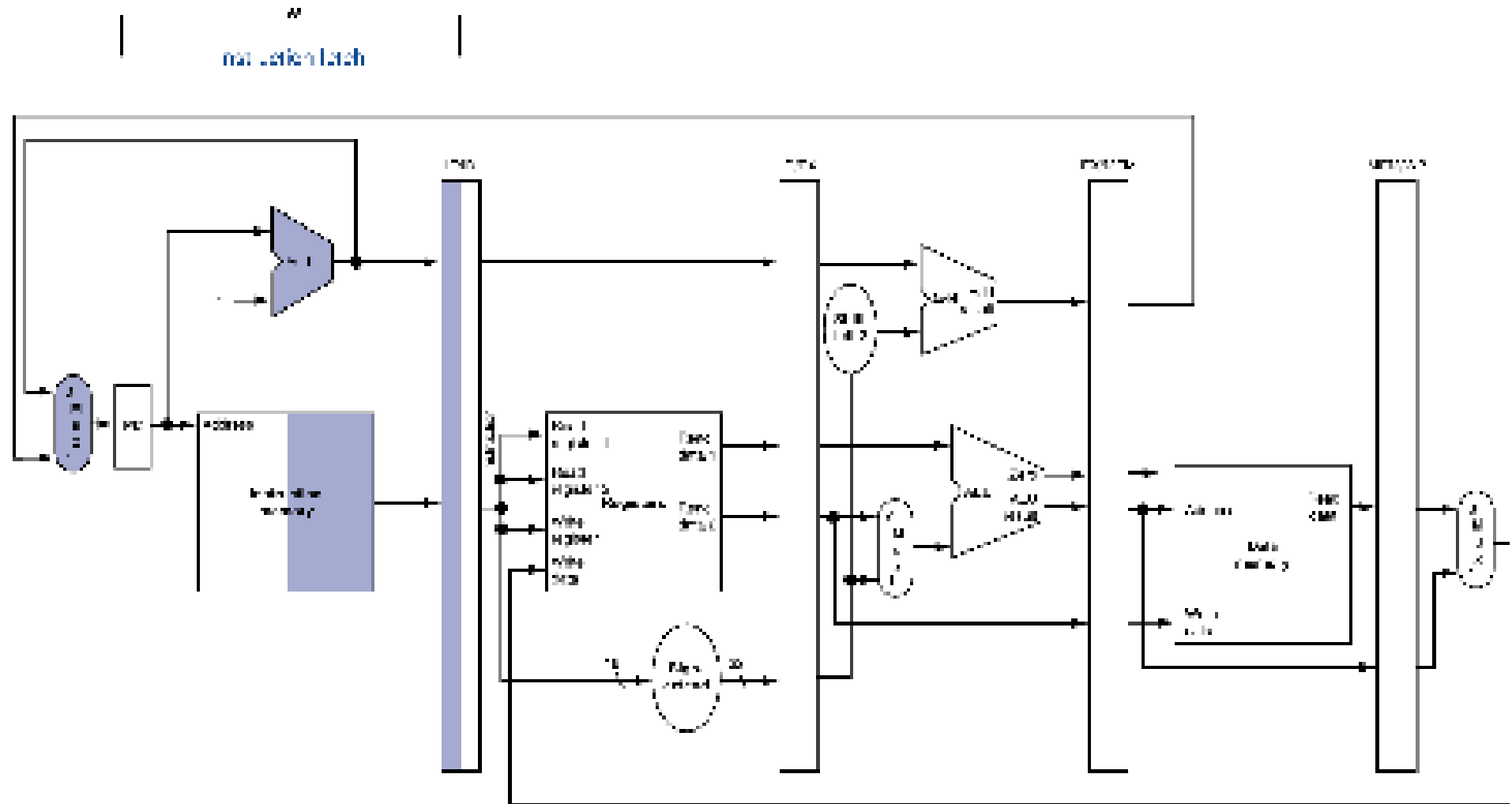
- Need registers between stages
 - To hold information produced in previous cycle



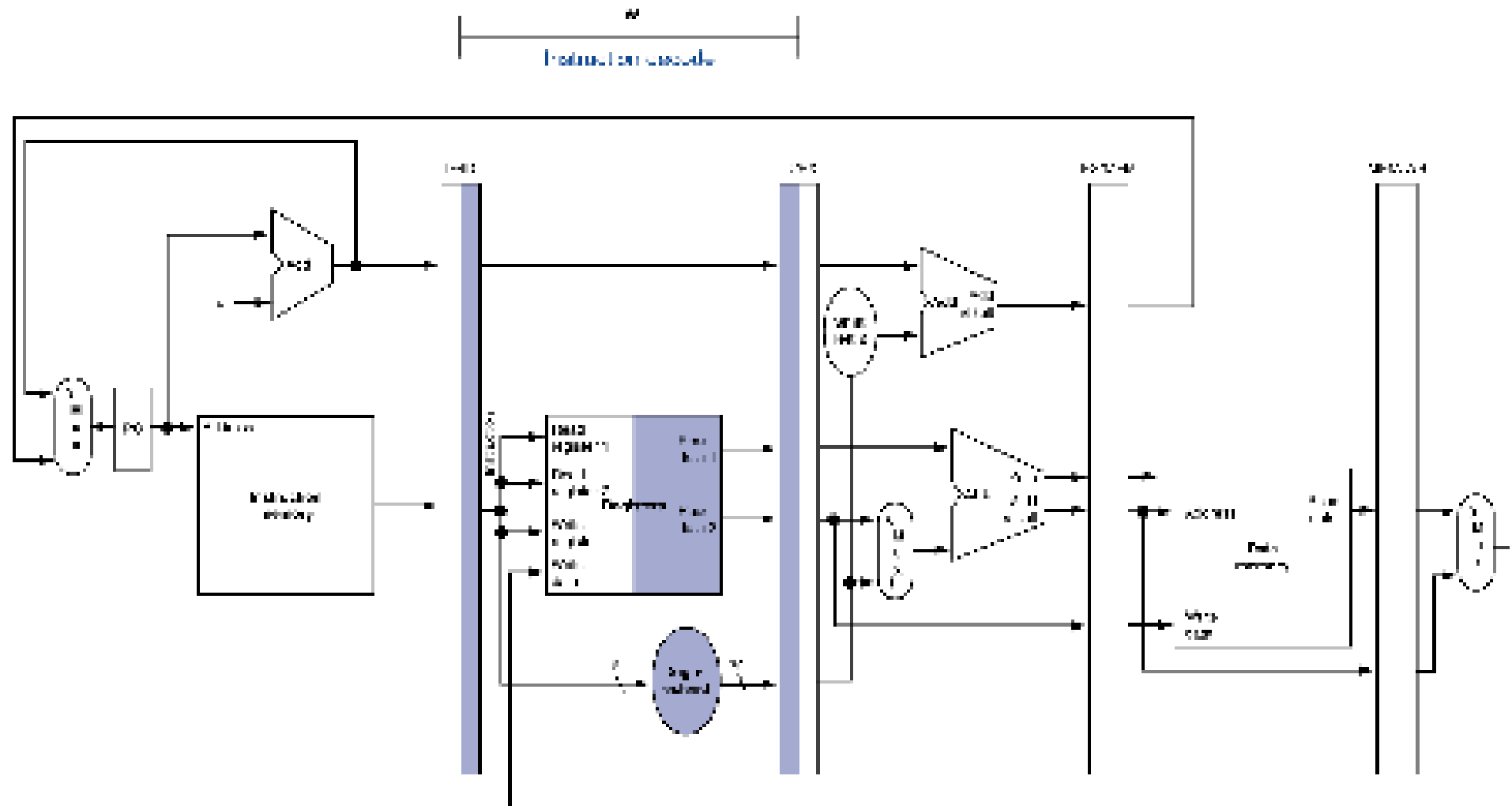
Pipeline Operation

- 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

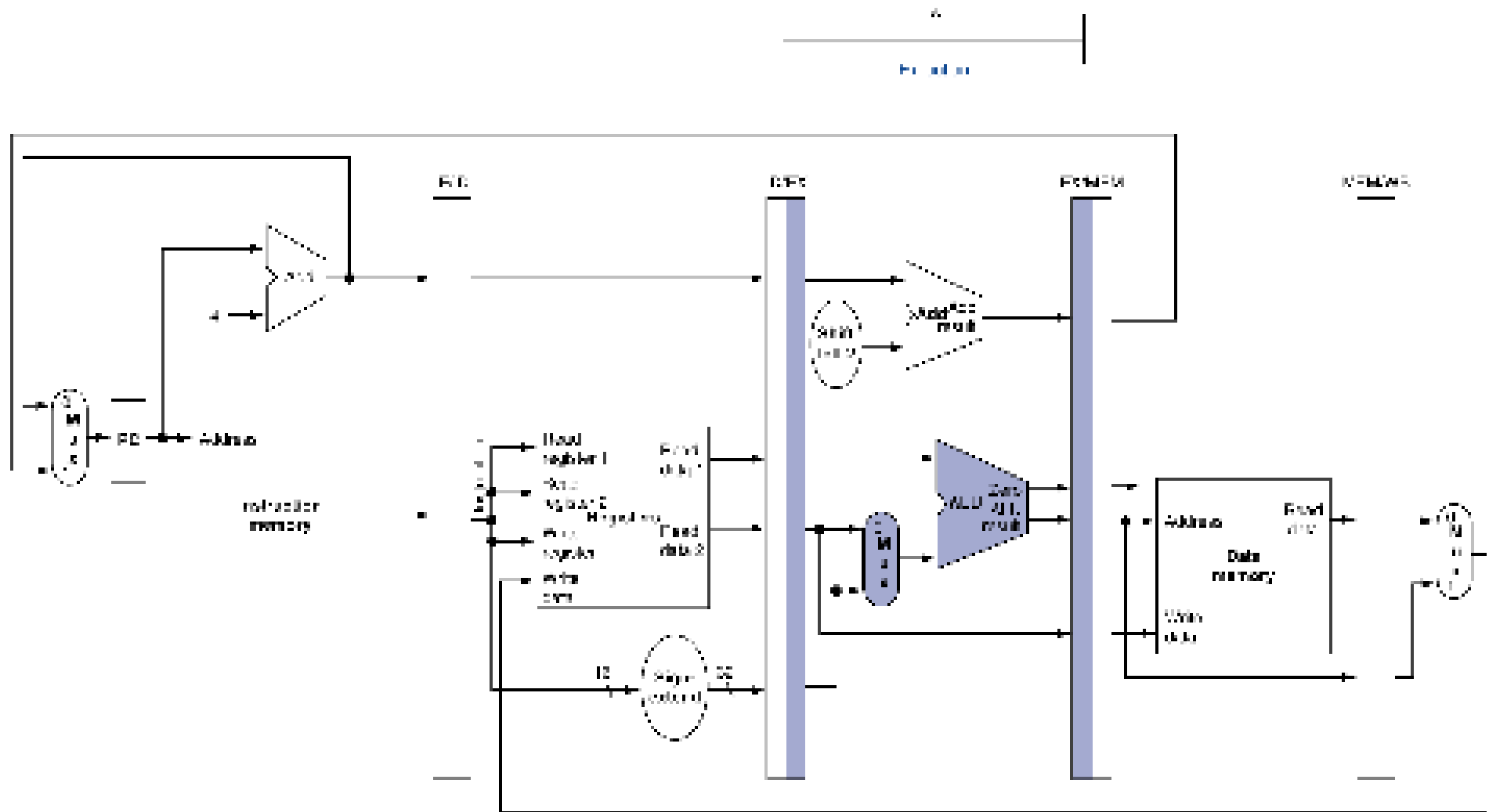
IF for Load, Store, ...



ID for Load, Store, ...

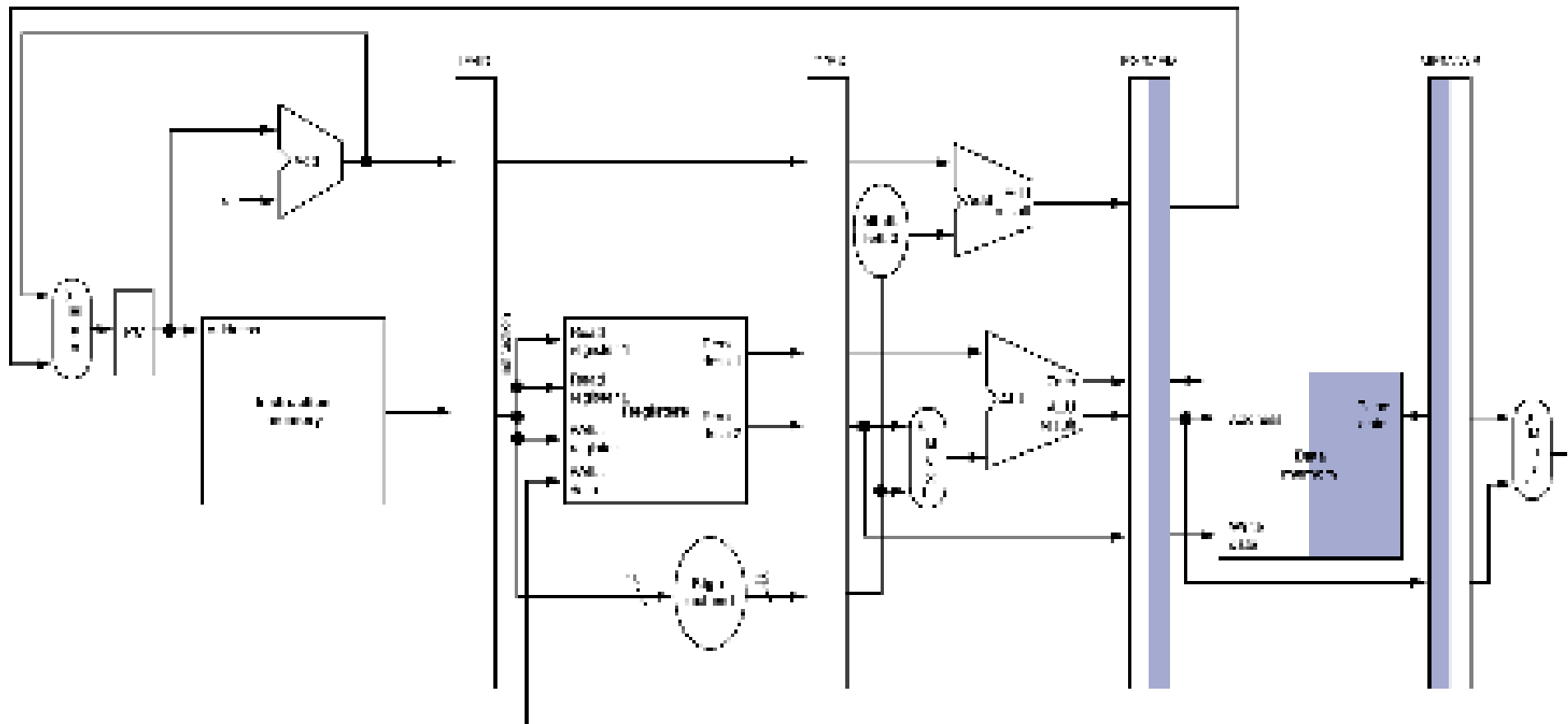


EX for Load



MEM for Load

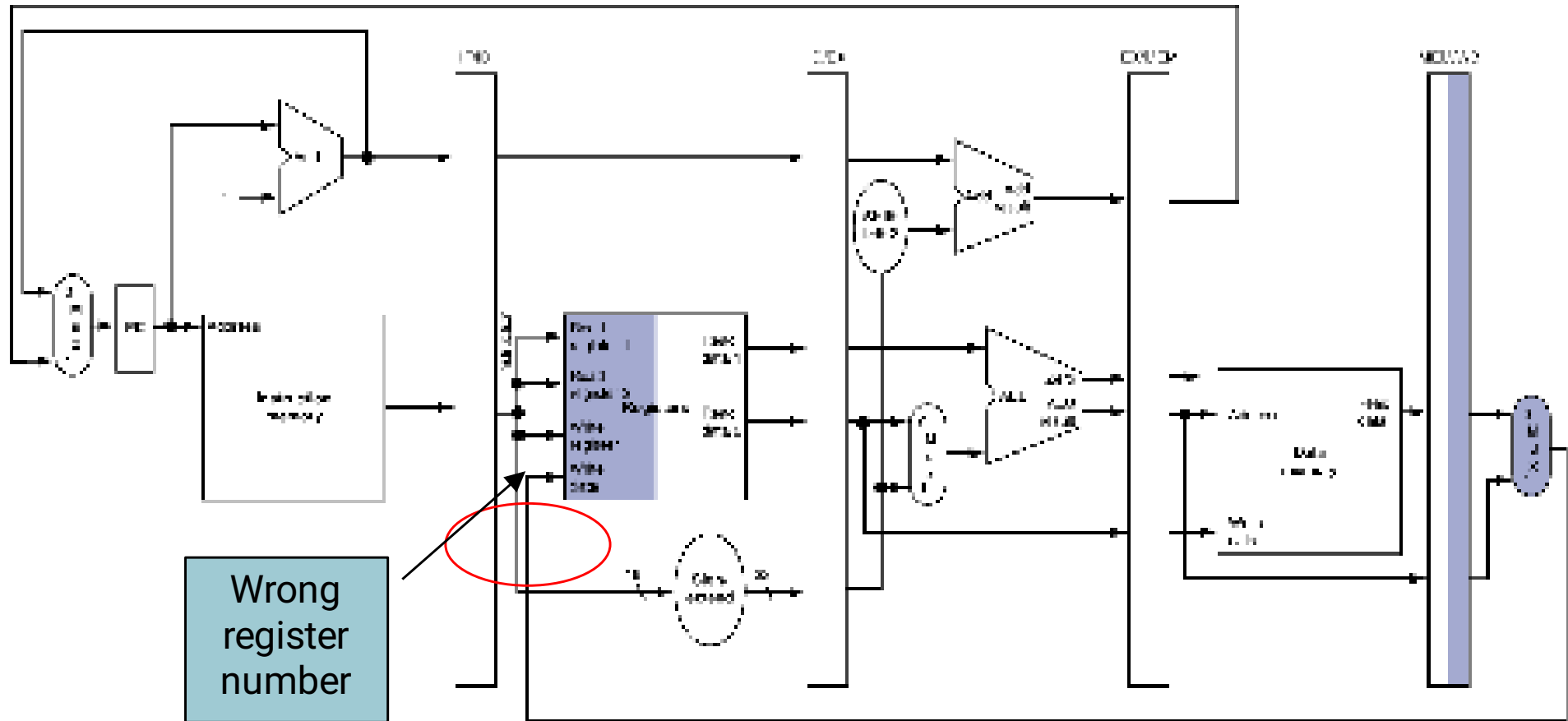
MEM
Memory



WB for Load

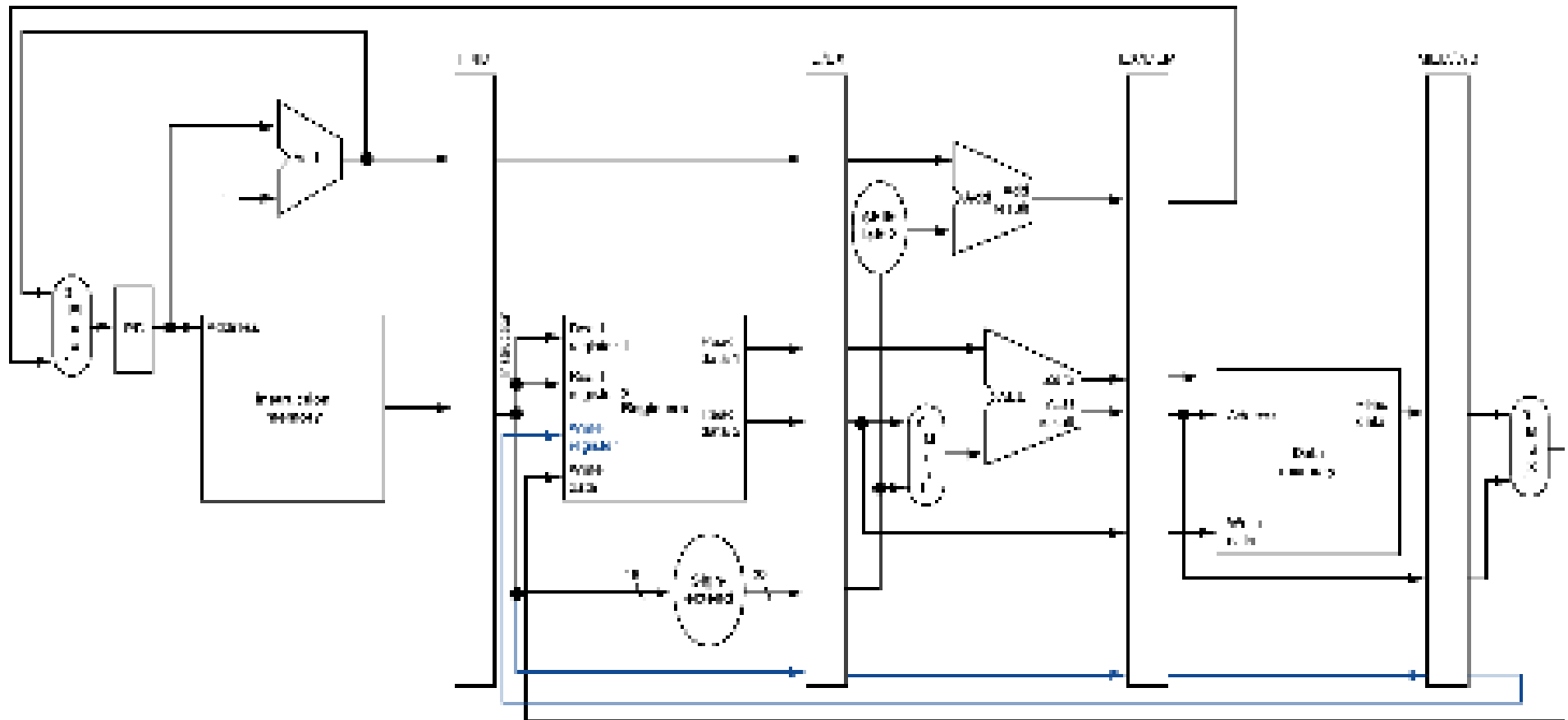
12

Write Back

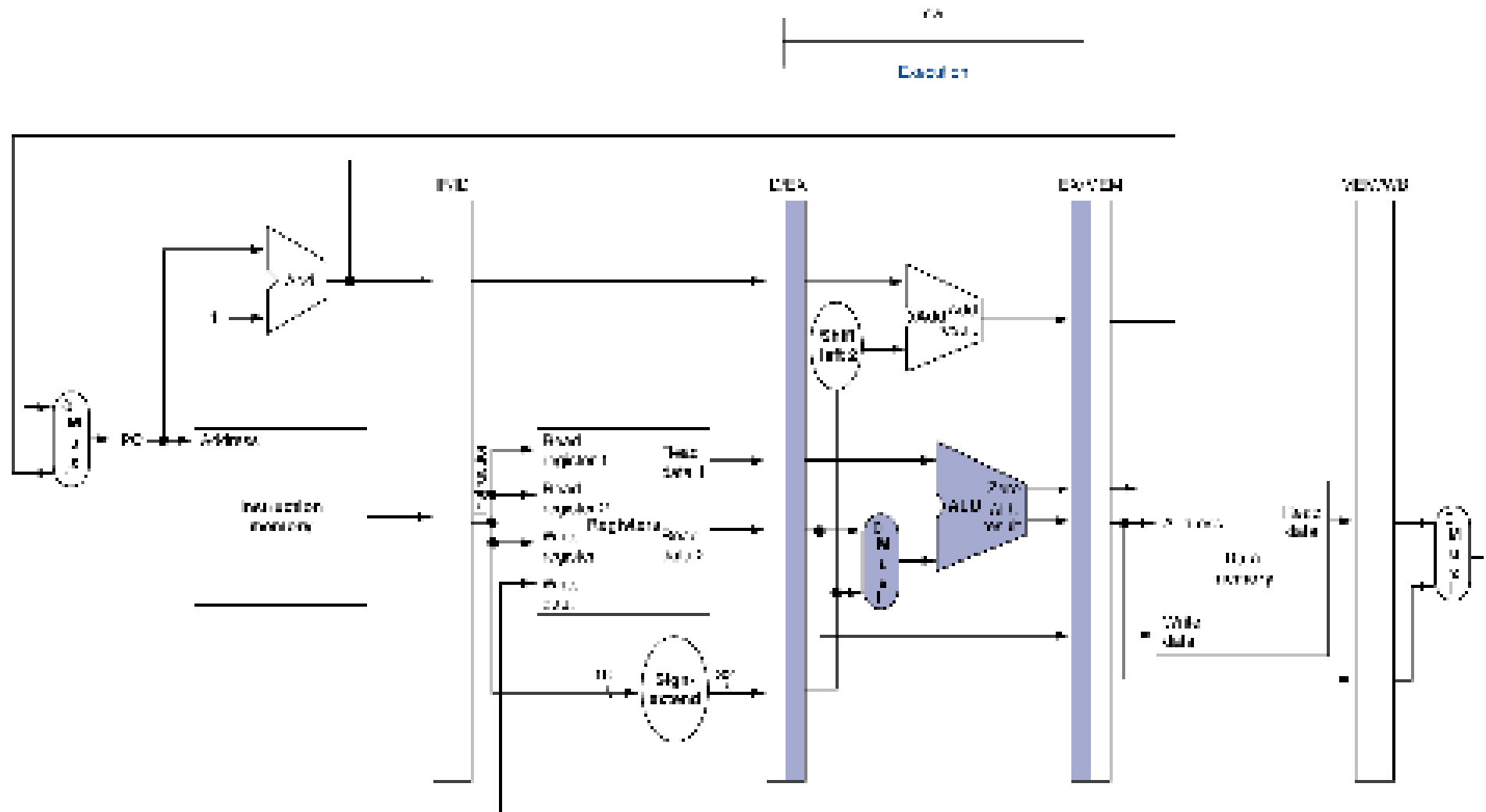


Any information needed in the later stage pipe must be passed to that stage via pipeline registers. Otherwise information will be lost when the next instruction enters the pipeline

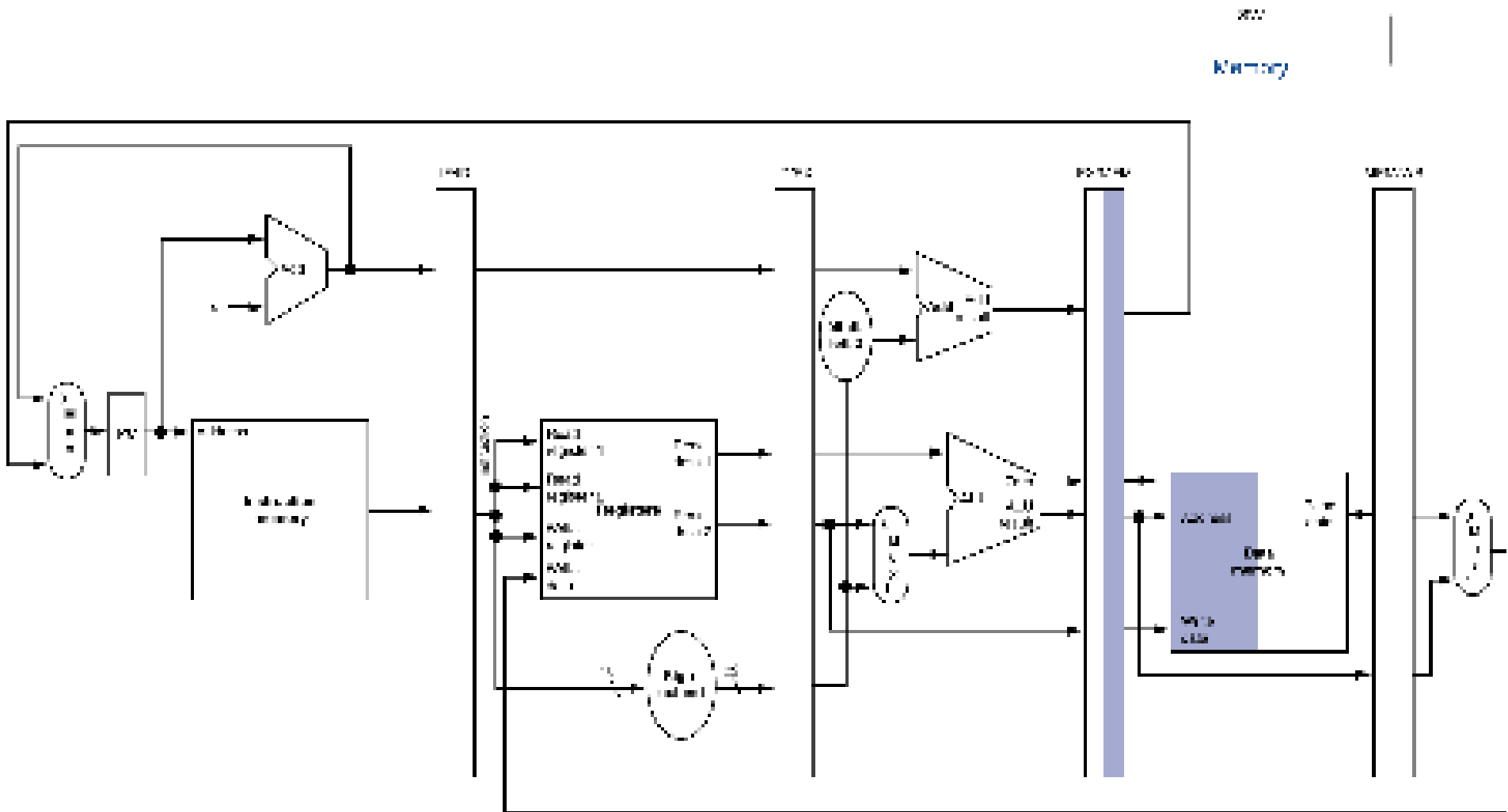
Corrected Datapath for Load



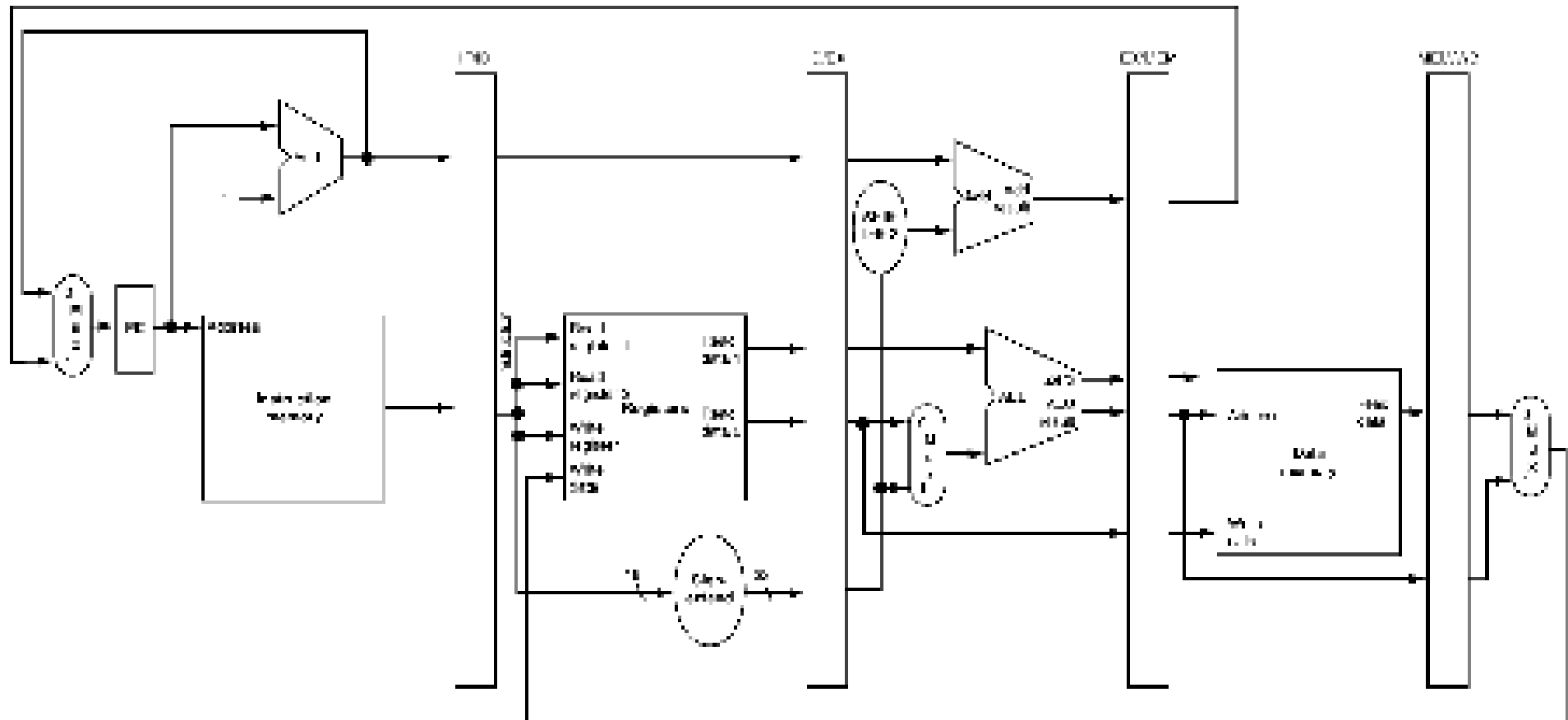
EX for Store



MEM for Store



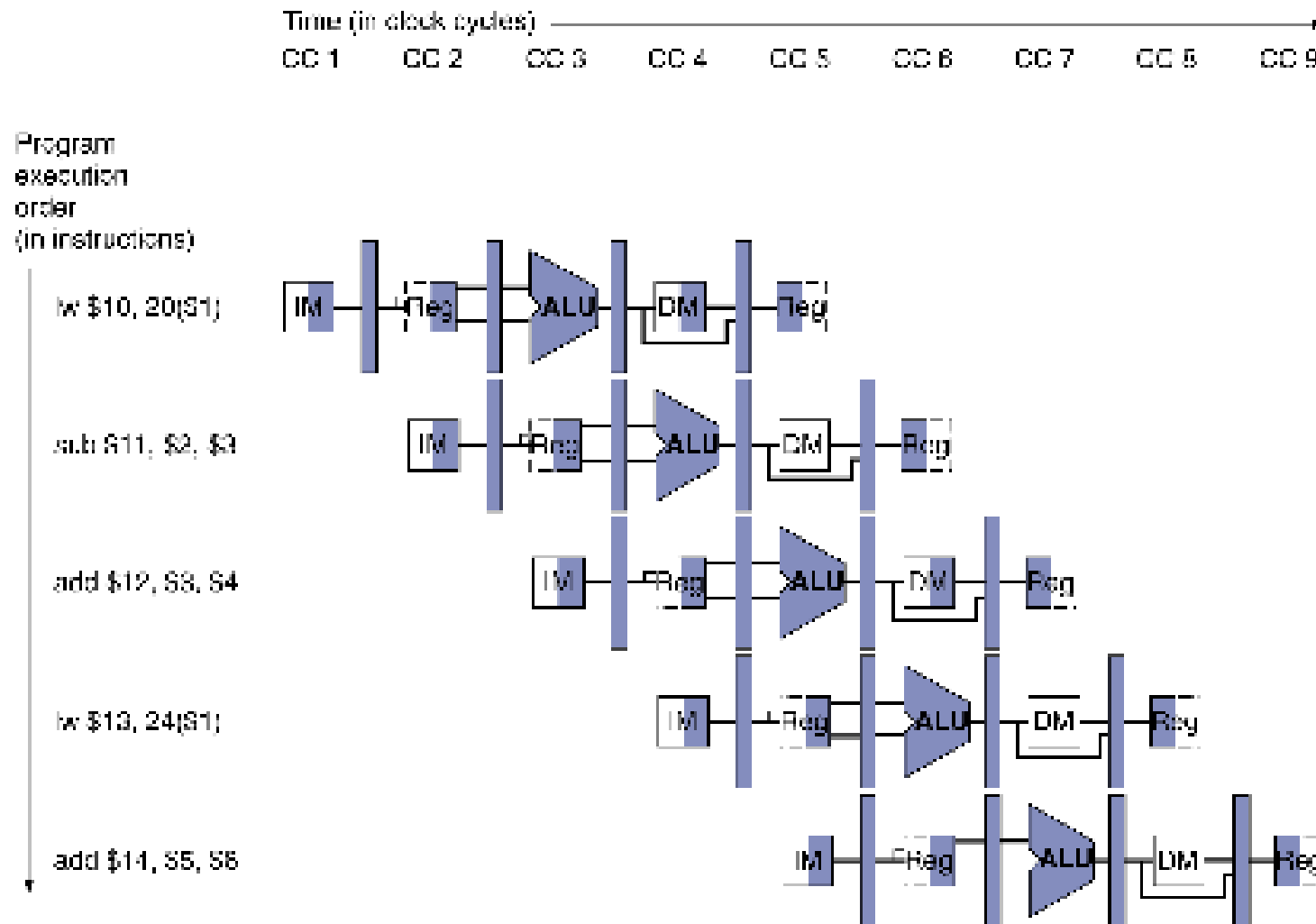
WB for Store



Each logical component of the datapath can be used only within a single pipeline stage.
Otherwise we will have a structural hazard

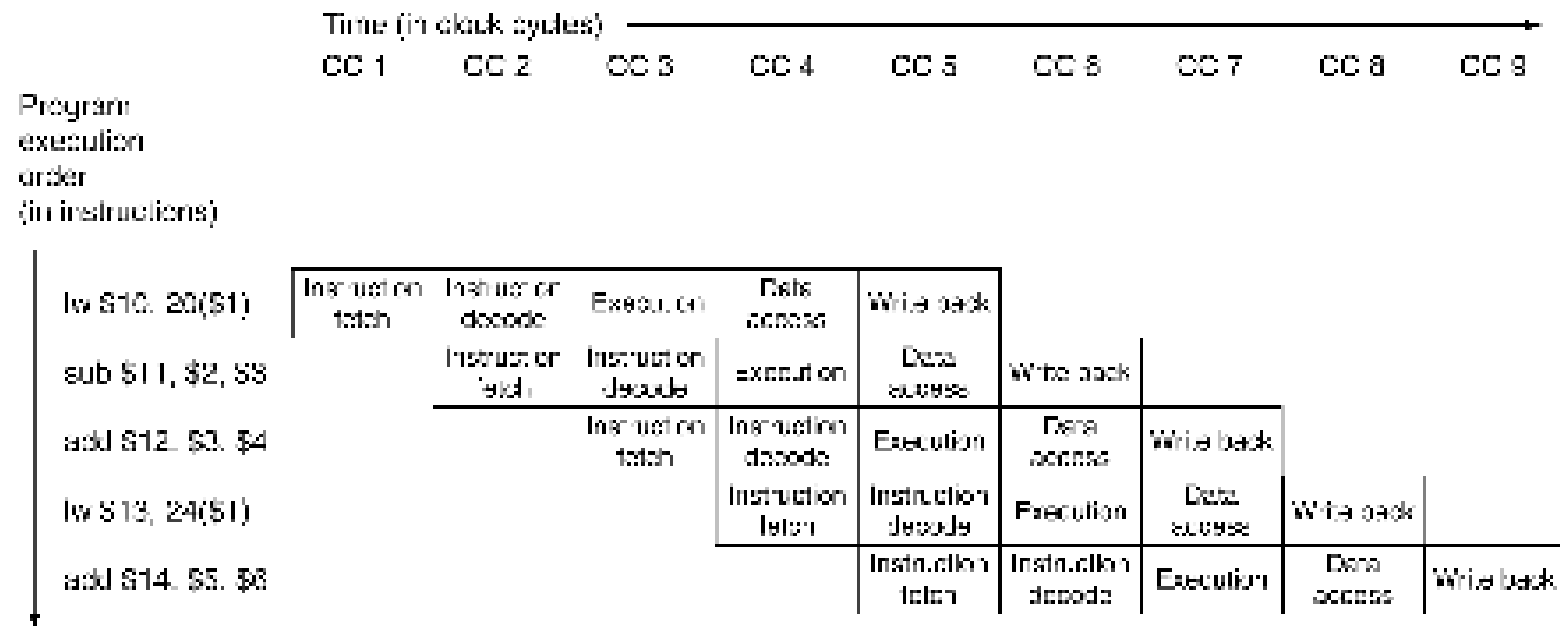
Multi-Cycle Pipeline Diagram

- Form showing resource usage



Multi-Cycle Pipeline Diagram

■ Traditional form



Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle (5th clock cycle)
- Single cycle diagram represents a vertical slice through a set of multi cycle diagram

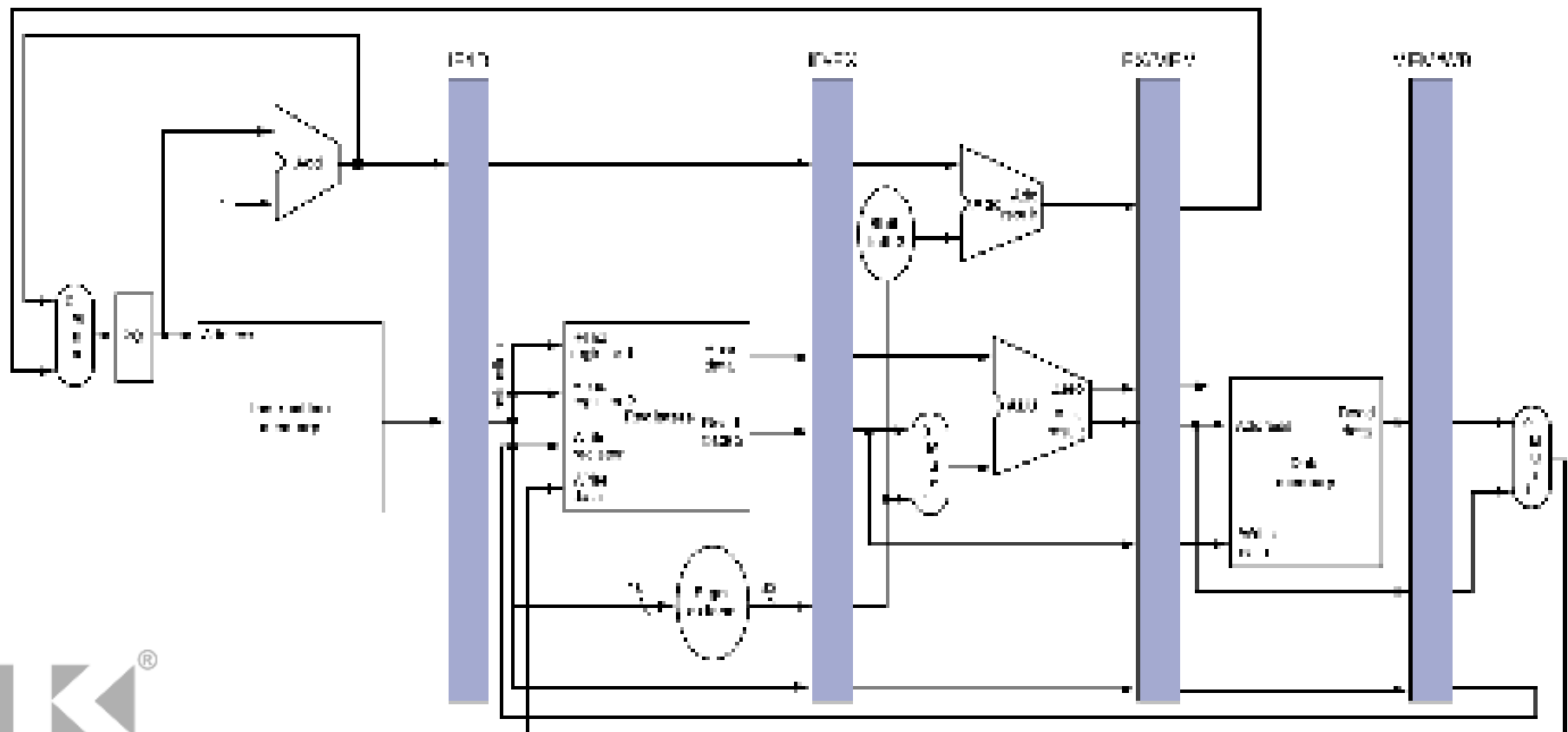
read \$t0, \$t1, \$t6
Instruction fetch

add \$t0, \$t1, \$t1
Instruction decode

add \$t0, \$t1, \$t1
Execution

add \$t0, \$t1, \$t1
Memory

lw \$t0, 20(\$t1)
Write-back



Pipelined Example

- Consider the following instruction sequence:

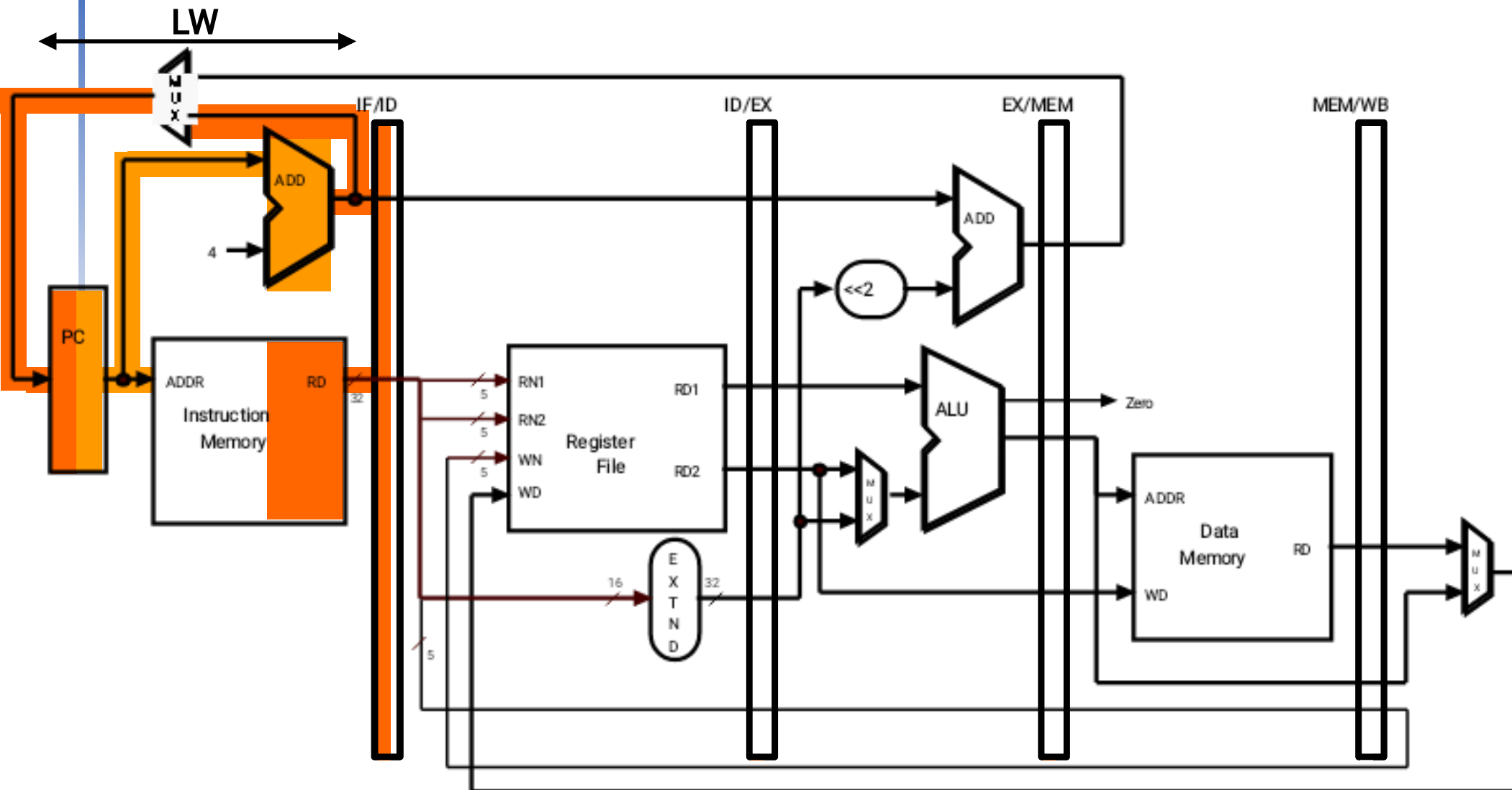
lw \$t0, 10(\$t1)

sw \$t3, 20(\$t4)

add \$t5, \$t6, \$t7

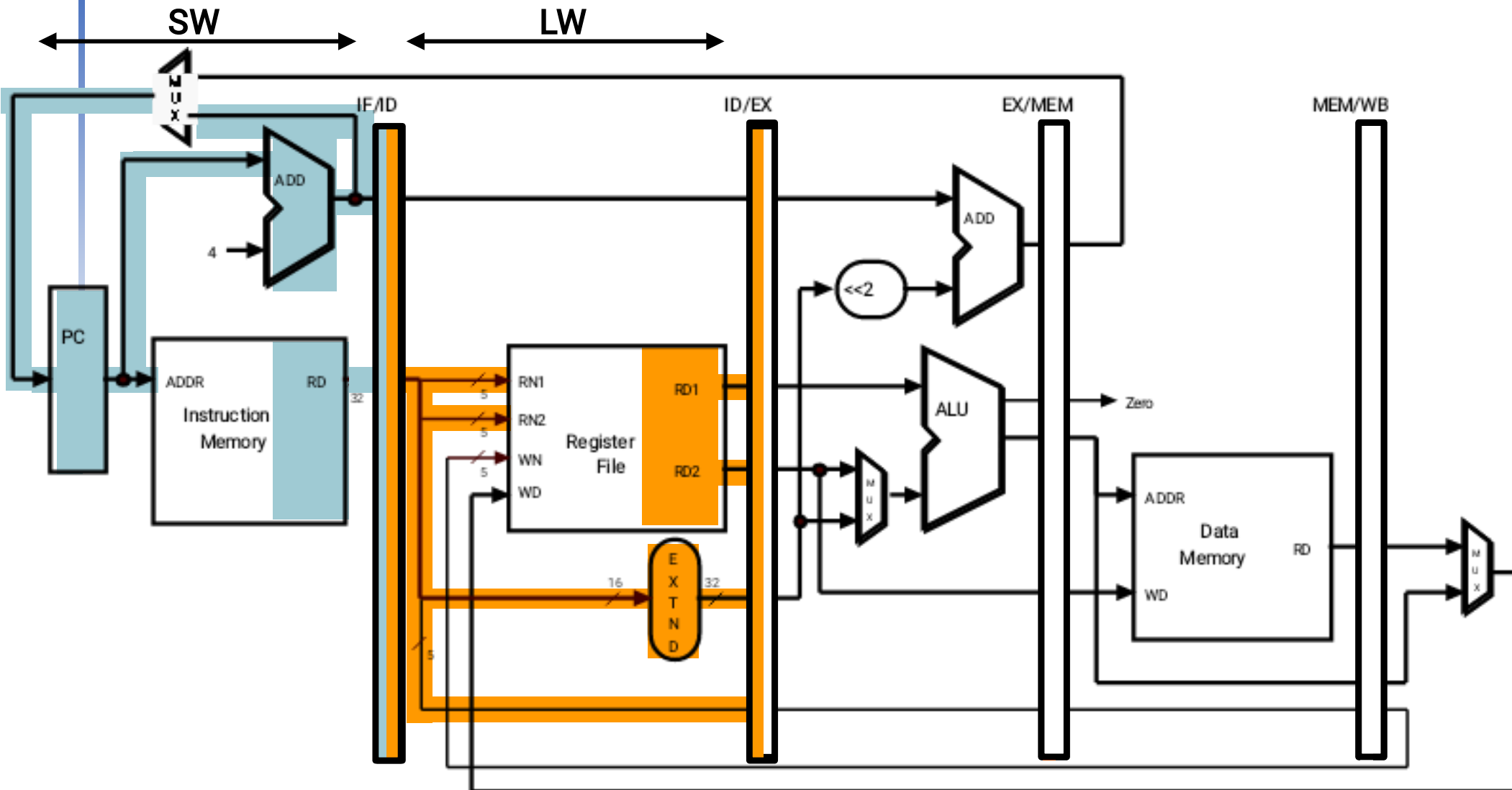
sub \$t8, \$t9, \$t10

Single Clock Cycle Diagram: Clock Cycle 1



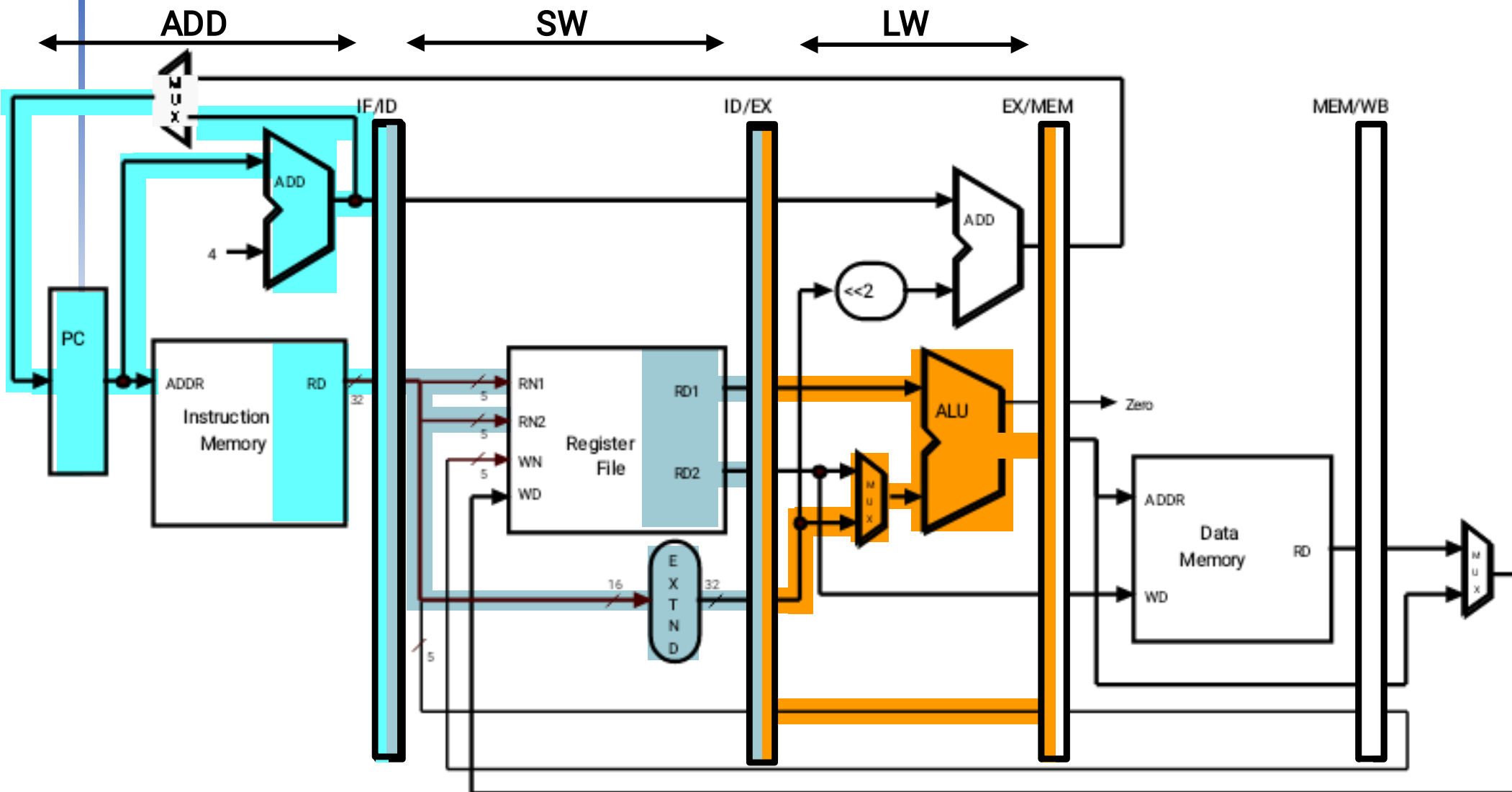
Single Clock Cycle Diagram:

Clock Cycle 2



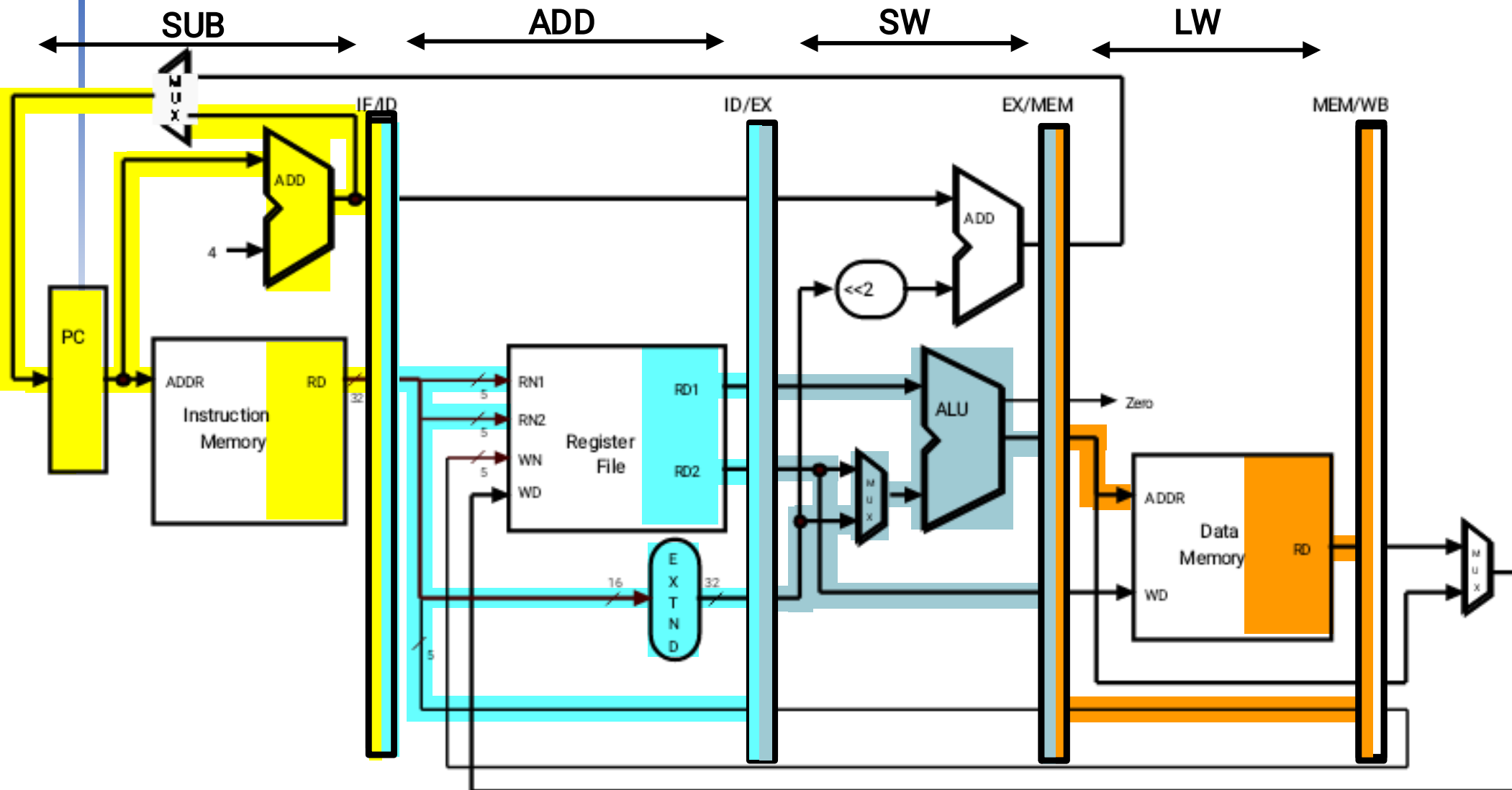
Single Clock Cycle Diagram:

Clock Cycle 3



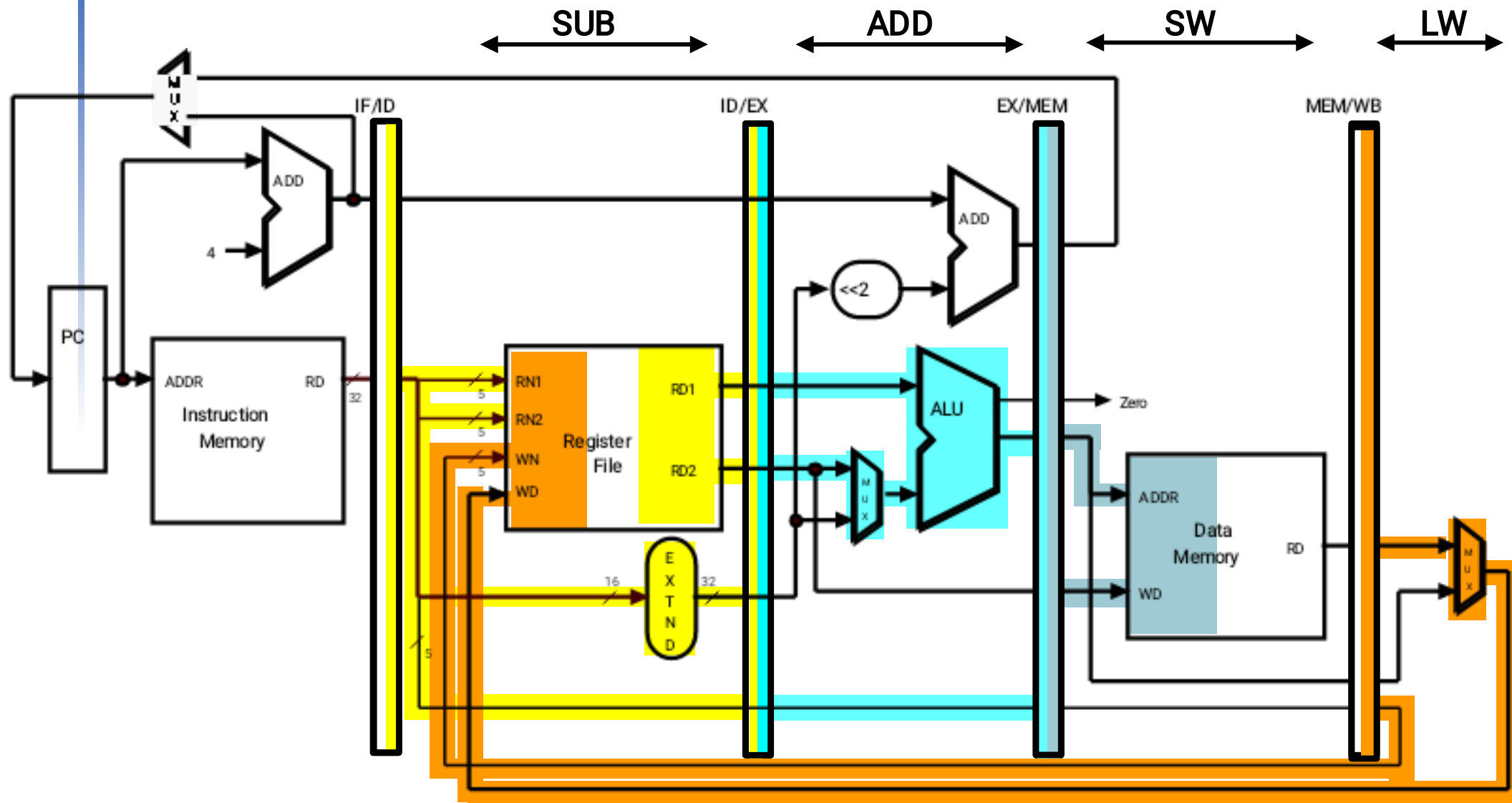
Single Clock Cycle Diagram:

Clock Cycle 4



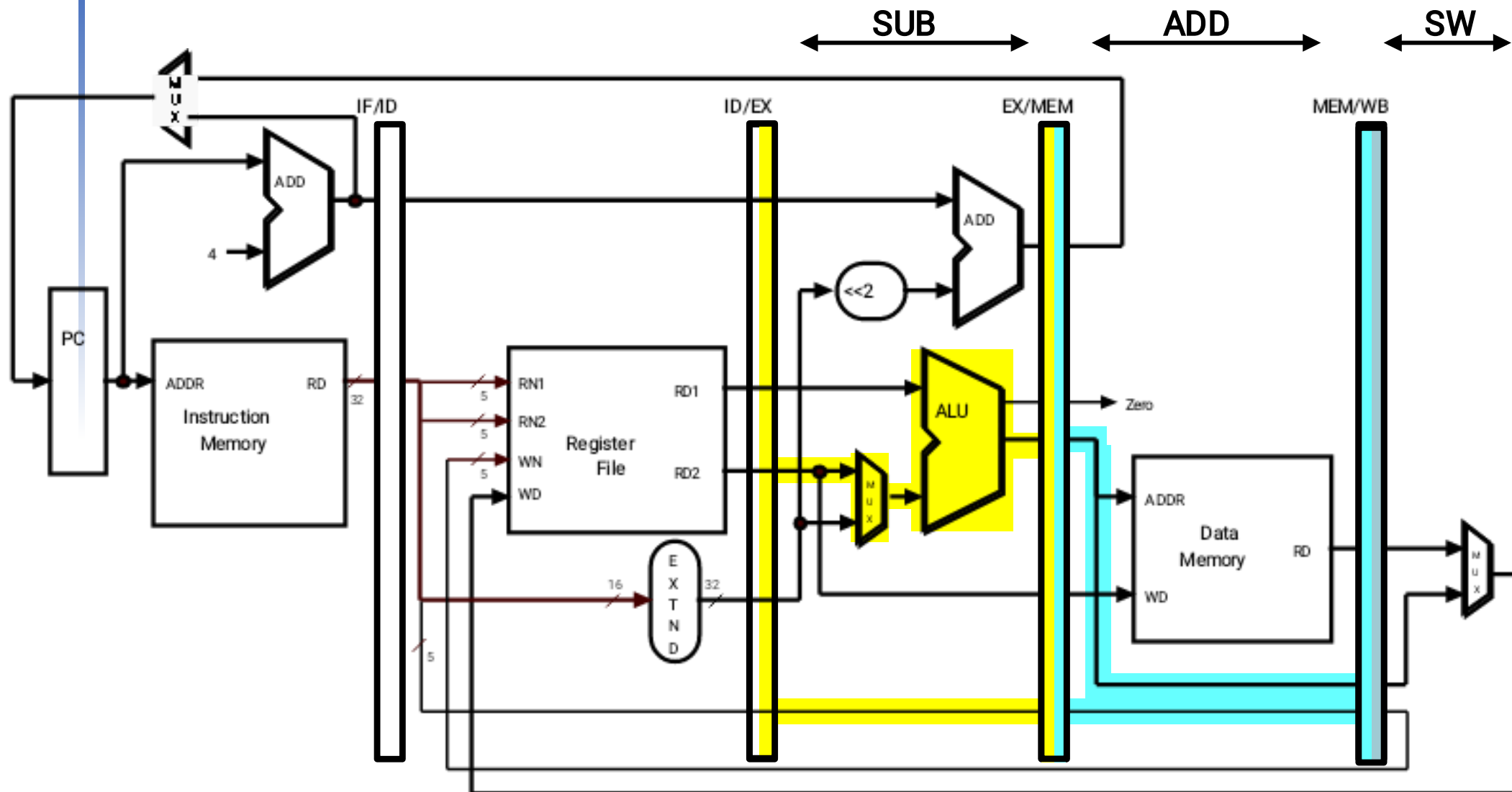
Single Clock Cycle Diagram:

Clock Cycle 5



Single Clock Cycle Diagram:

Clock Cycle 6

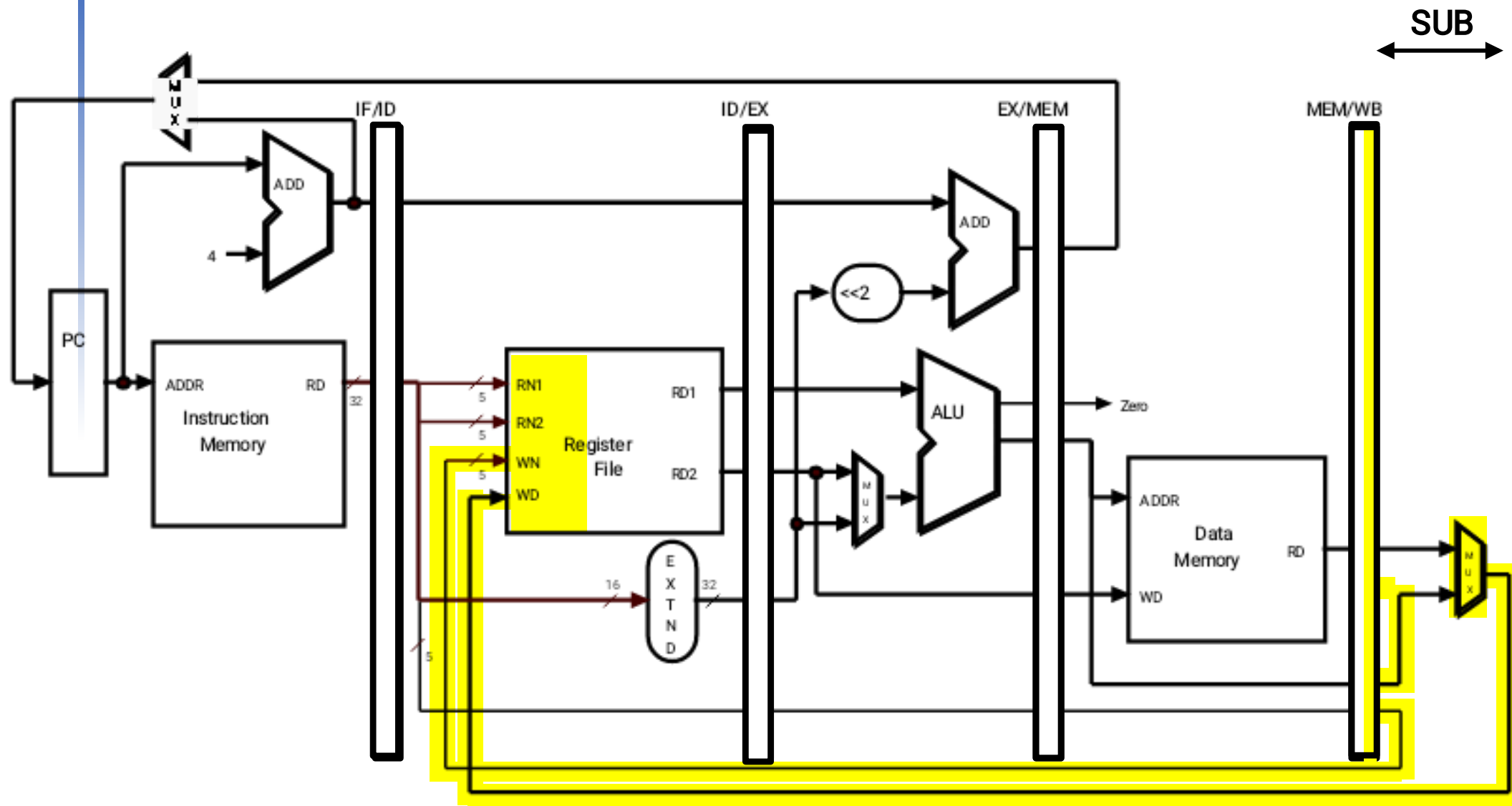


Clock Cycle 7

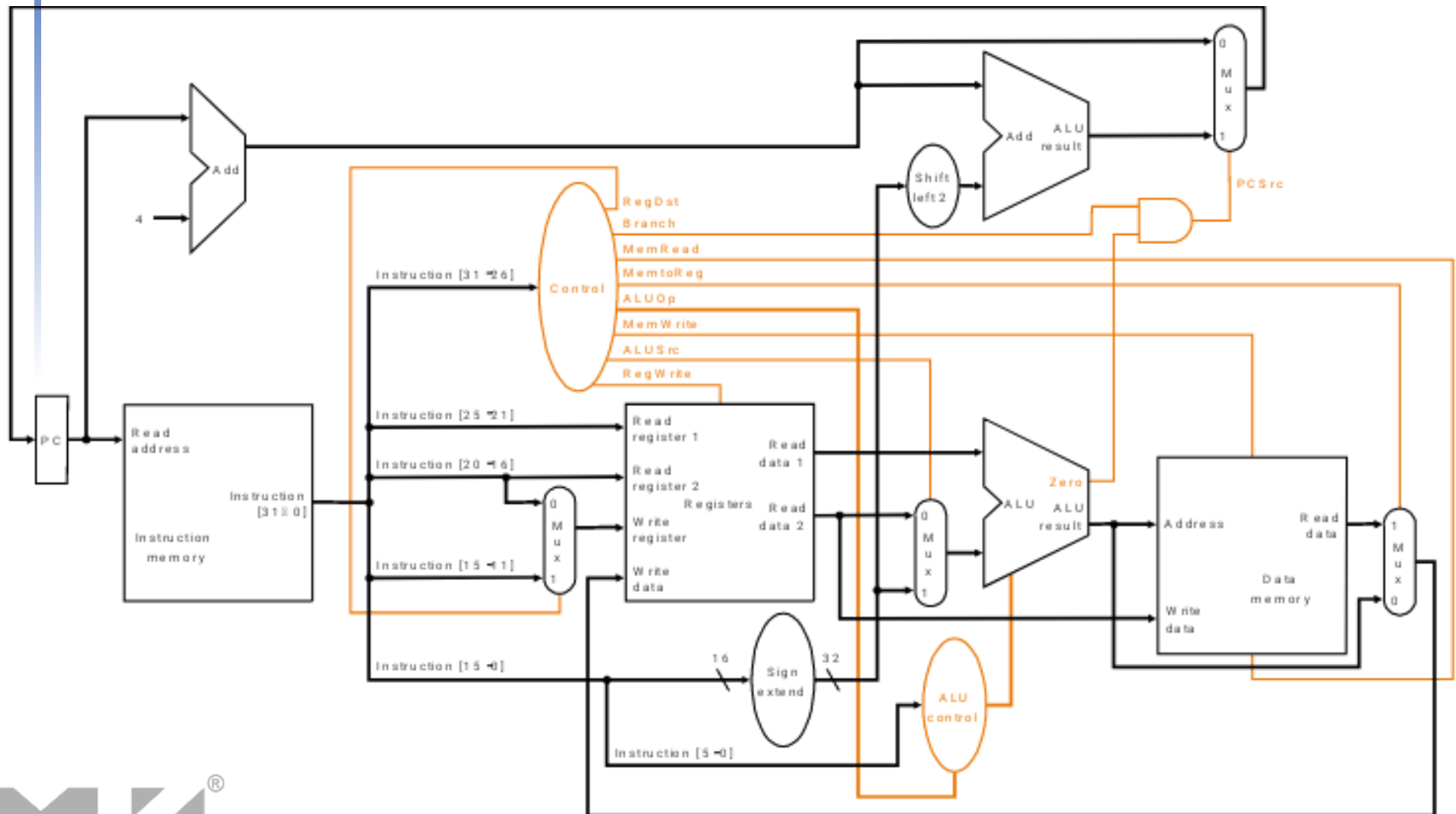


Single Clock Cycle Diagram:

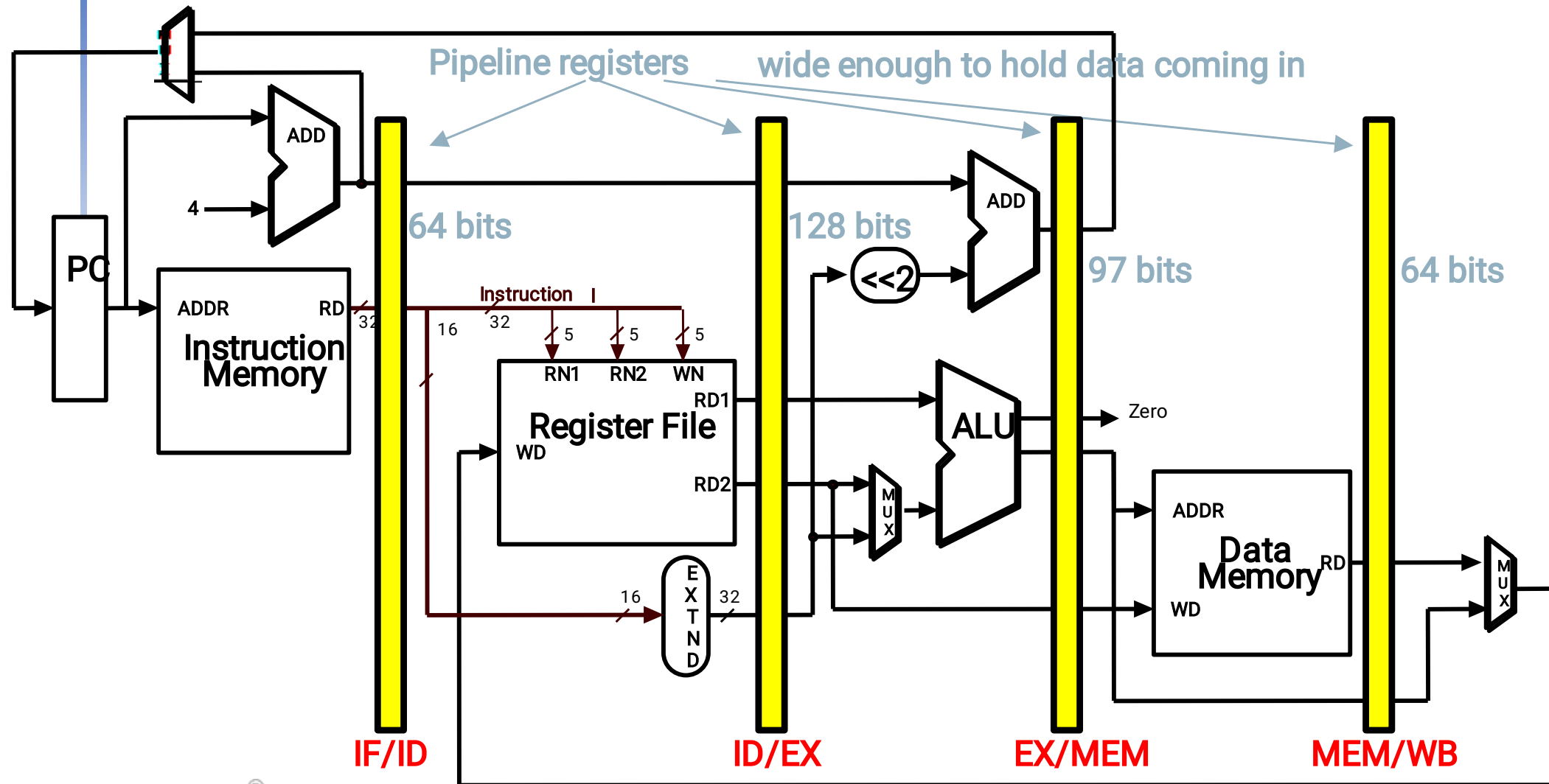
Clock Cycle 8



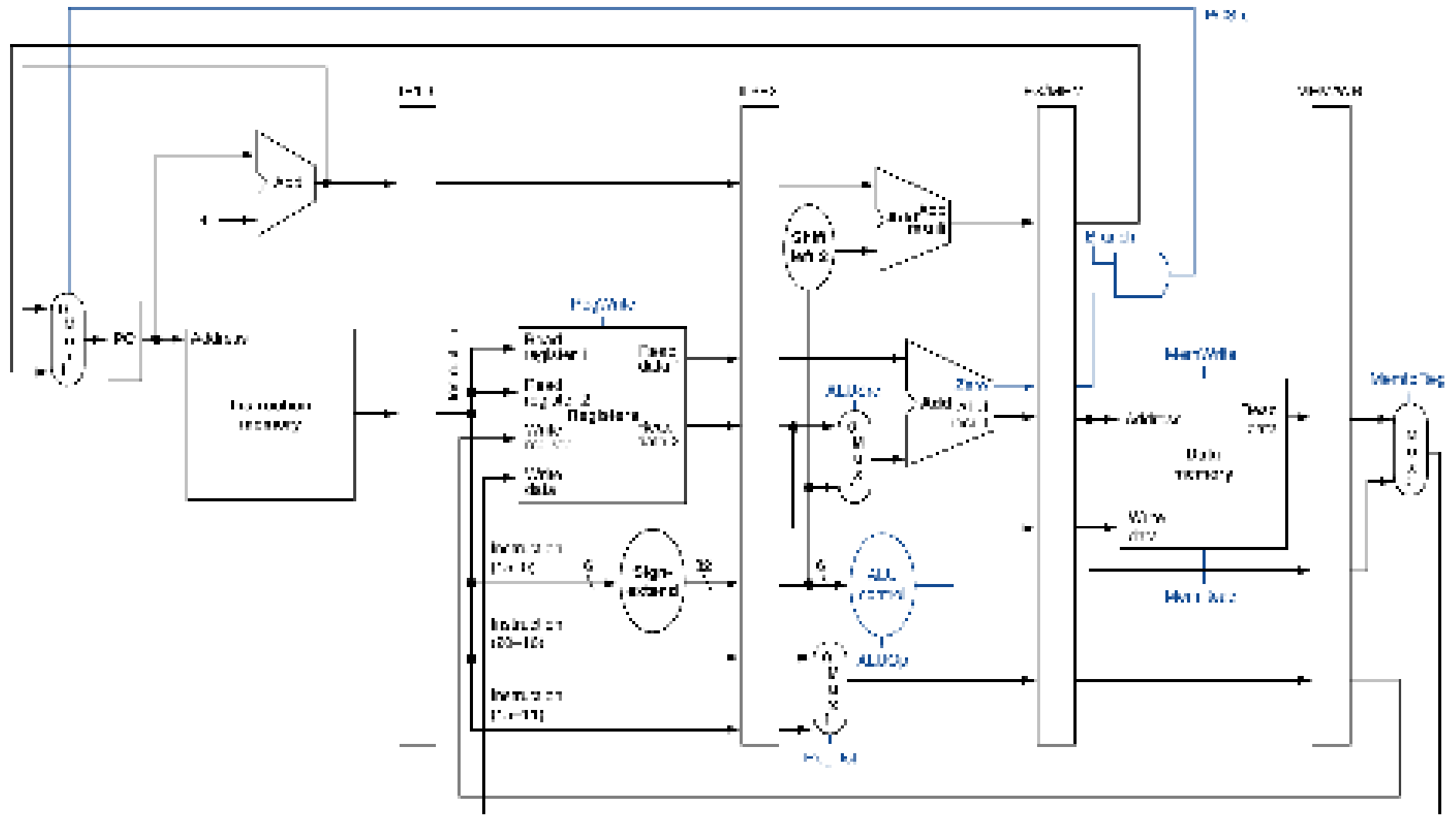
Recall Single-Cycle Control – the Datapath



Pipelined Datapath

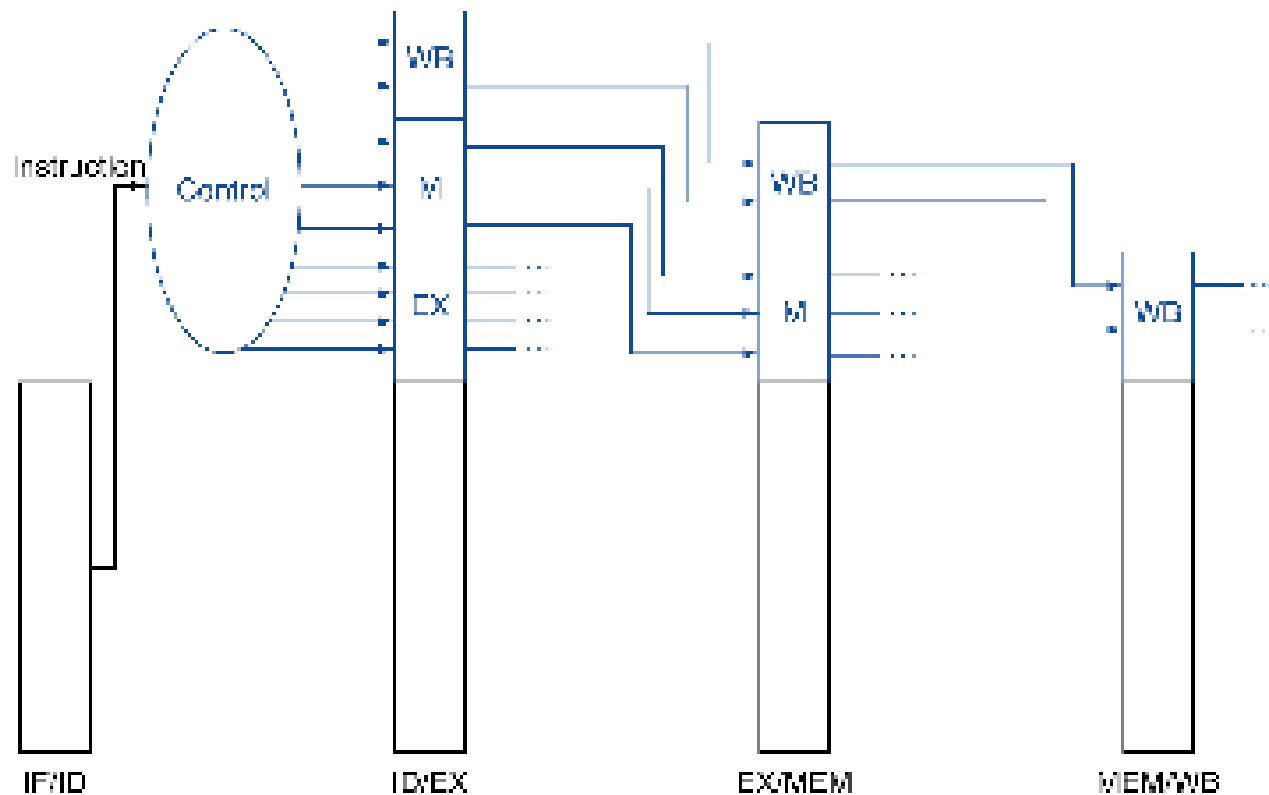


Pipelined Control (Simplified)

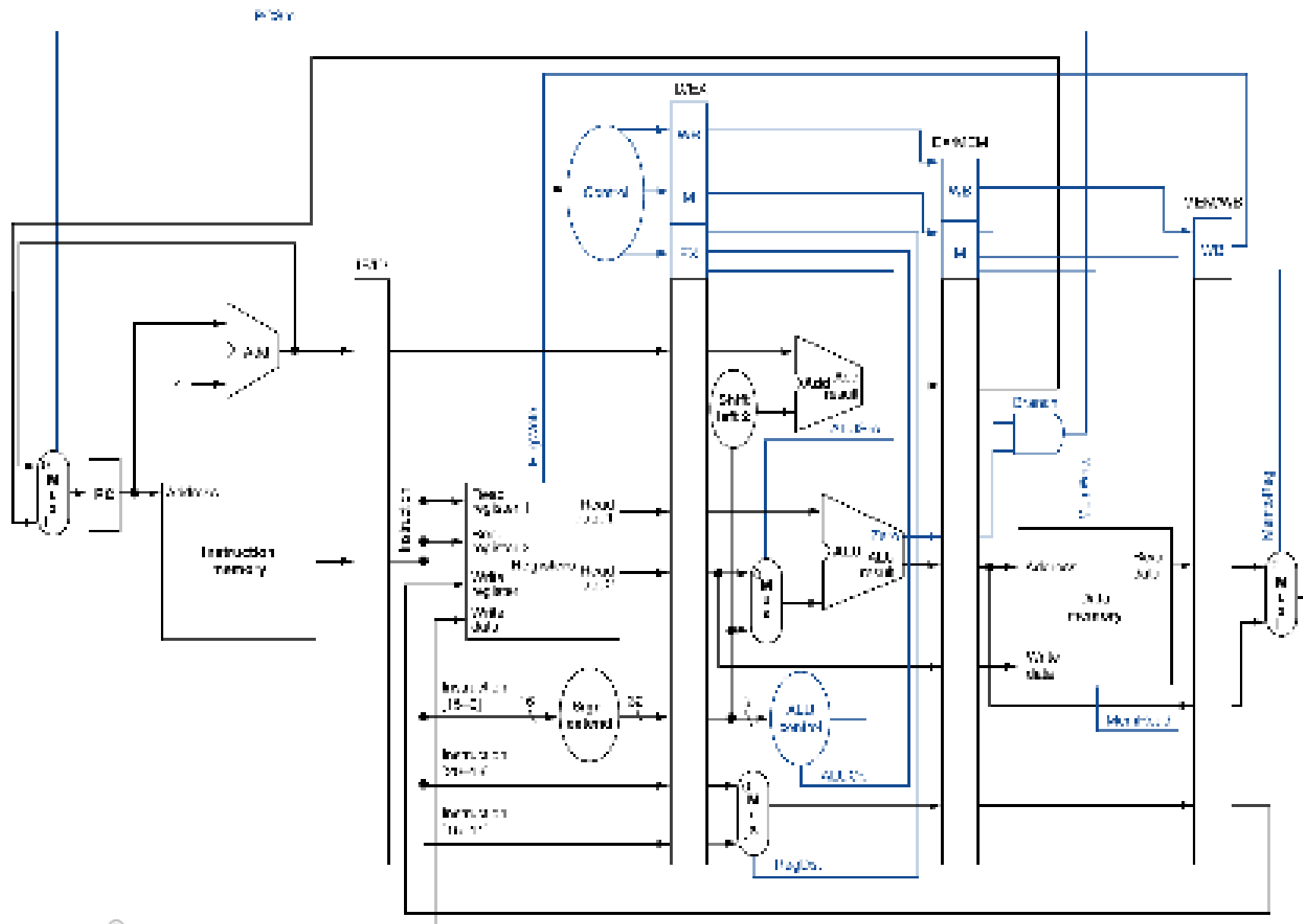


Pipelined Control

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



Pipelined Control



Revisiting Hazards

- So far our datapath and control have ignored hazards
- We shall revisit *data hazards* and *control hazards* and enhance our datapath and control to handle them in *hardware* ...

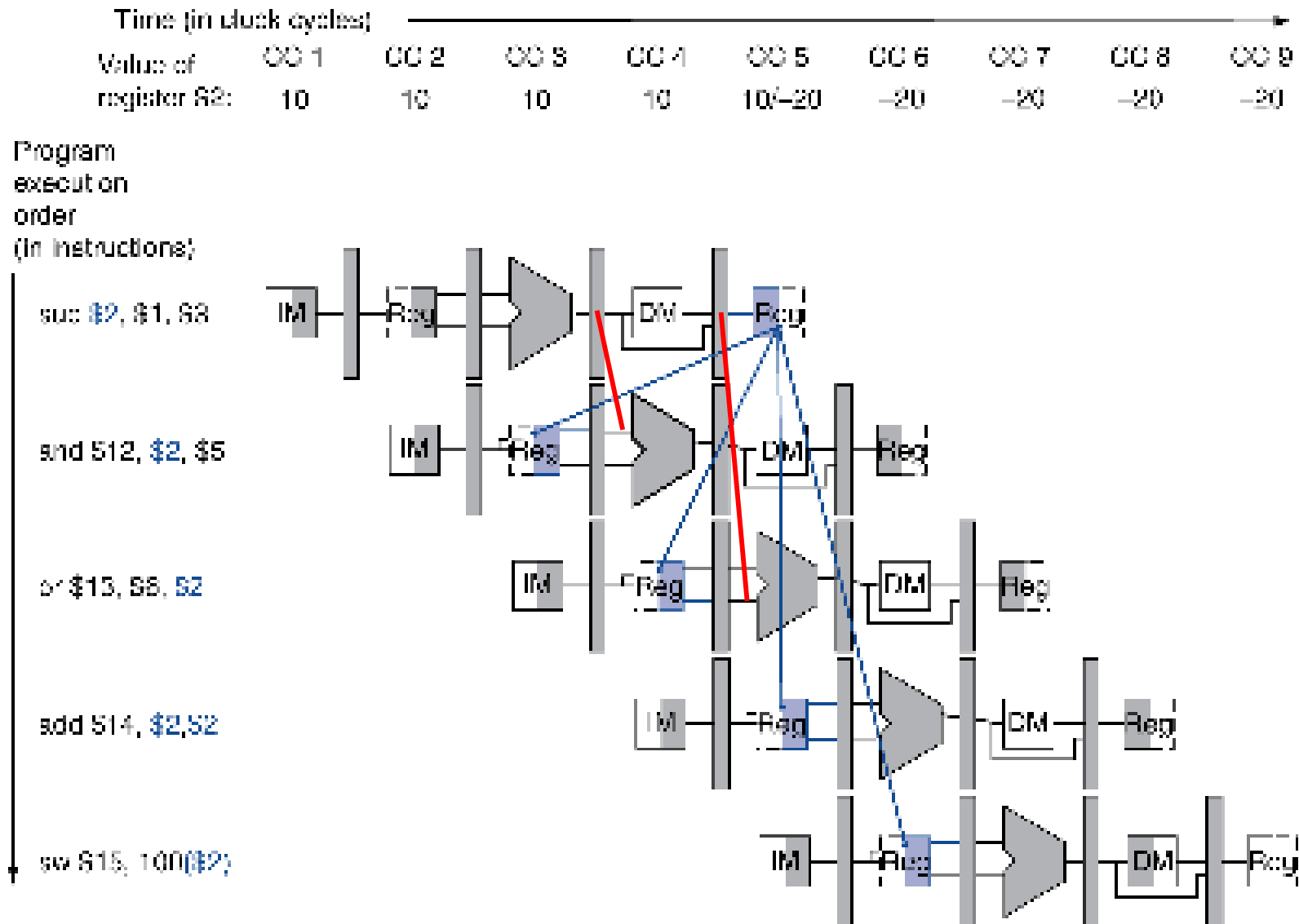
Data Hazards in ALU Instructions

- 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?

Hardware Solution: Forwarding

- Idea: *use intermediate data*, do not wait for result to be finally written to the destination register. Two steps:
 1. *Detect* data hazard
 2. *Forward* intermediate data to resolve hazard

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
 - 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

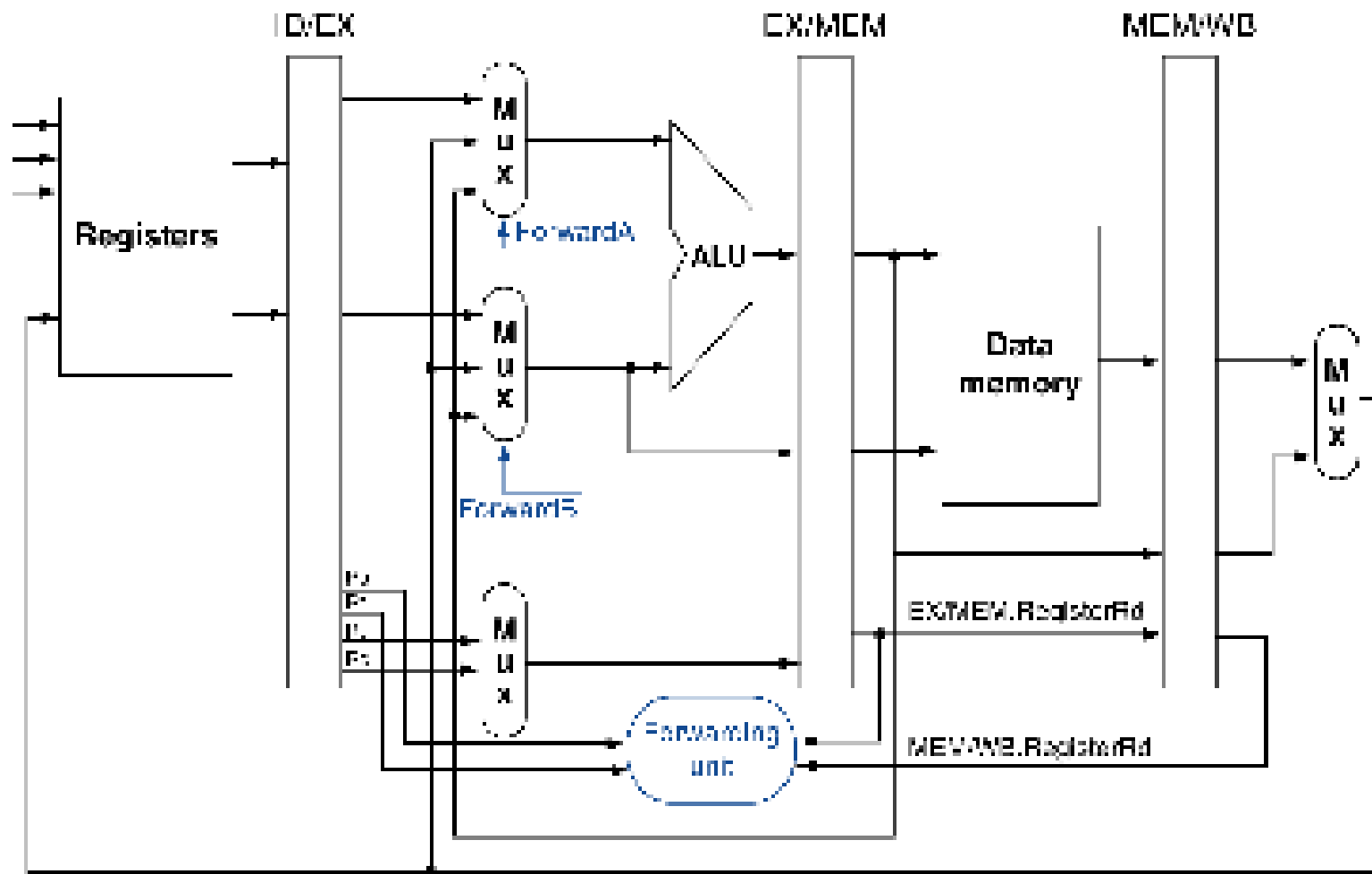
Fwd from
EX/MEM
pipeline reg

Fwd from
MEM/WB
pipeline reg

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
 - 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

Forwarding Paths



b. With forwarding

Forwarding Hardware: Multiplexor Control

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

Depending on the selection in the rightmost multiplexor
(see datapath with control diagram)

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

For a better understanding

- If (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01
- If (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01
- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01

Revised Forwarding Condition

- MEM hazard

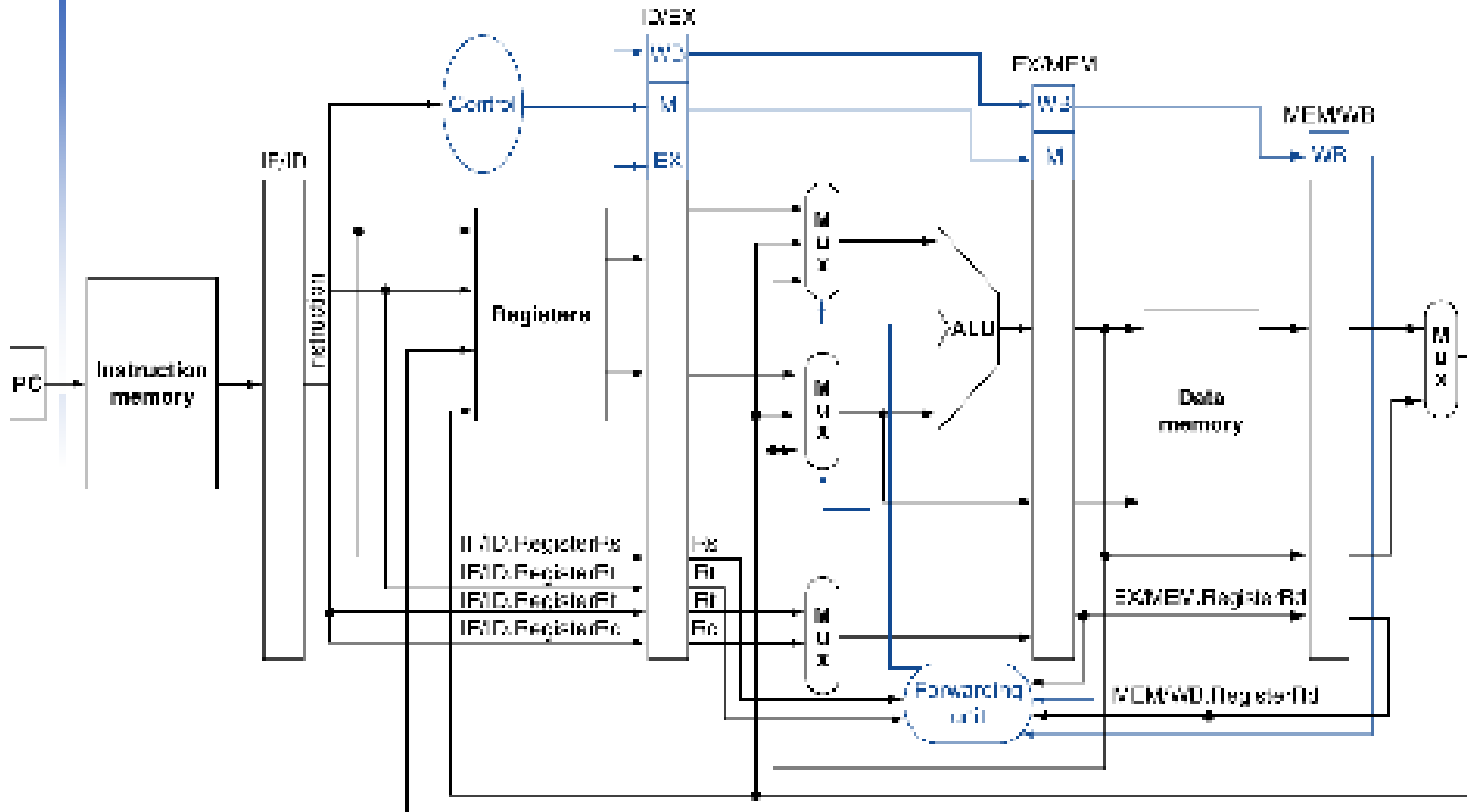
- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd \neq 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

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))

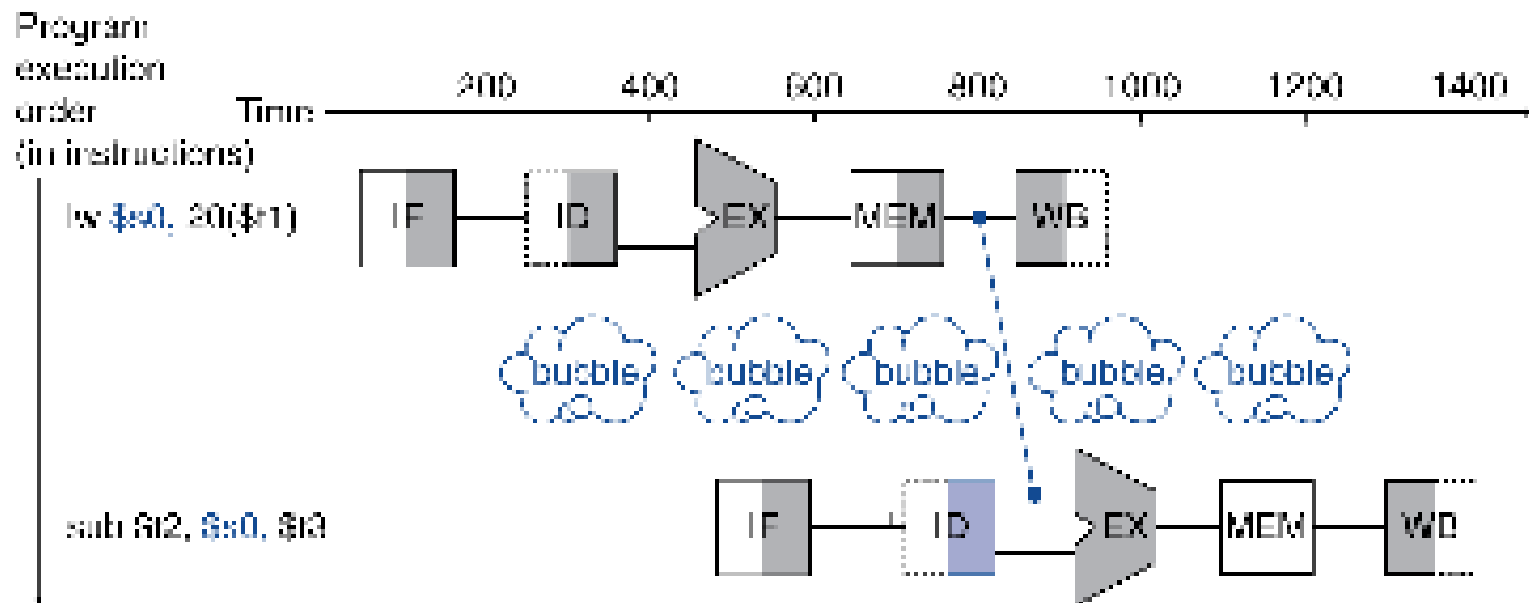
ForwardB = 01

| Datapath with Forwarding

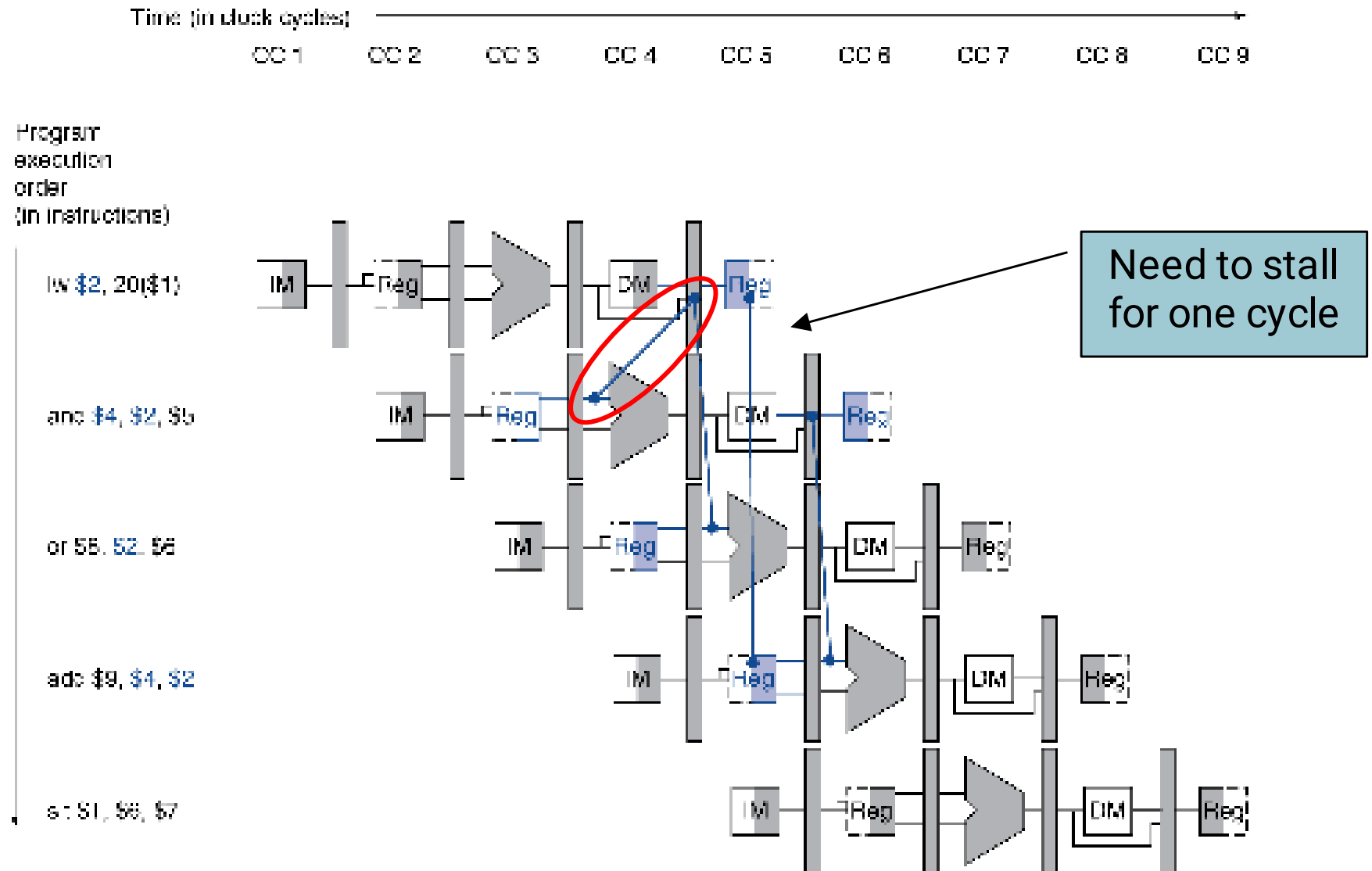


Load-Use Data Hazard

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



Load-Use Data Hazard



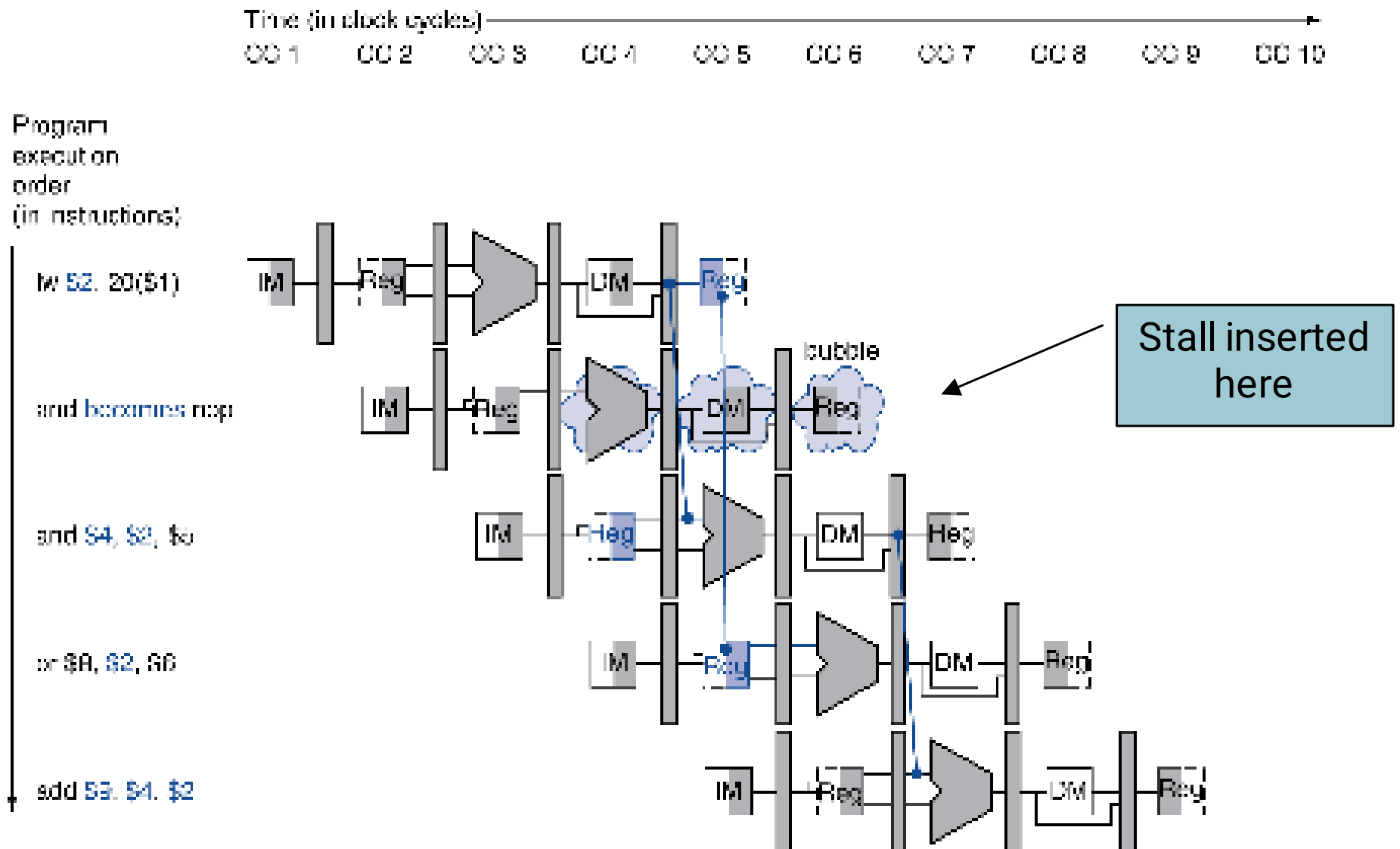
Load-Use Hazard Detection

- 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))
stall the pipeline

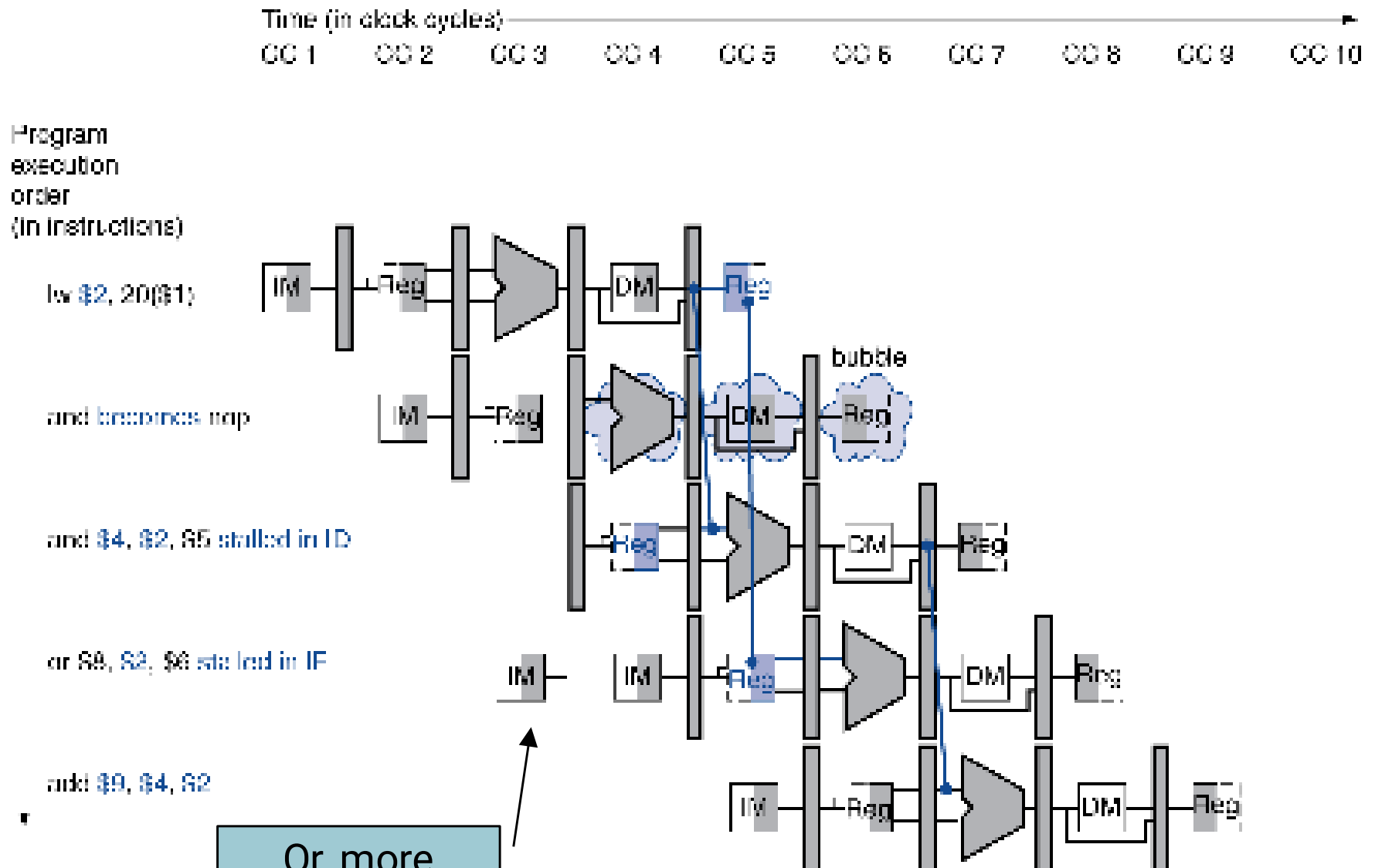
How to Stall the Pipeline

- 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 lw
 - Can subsequently **forward** to EX stage

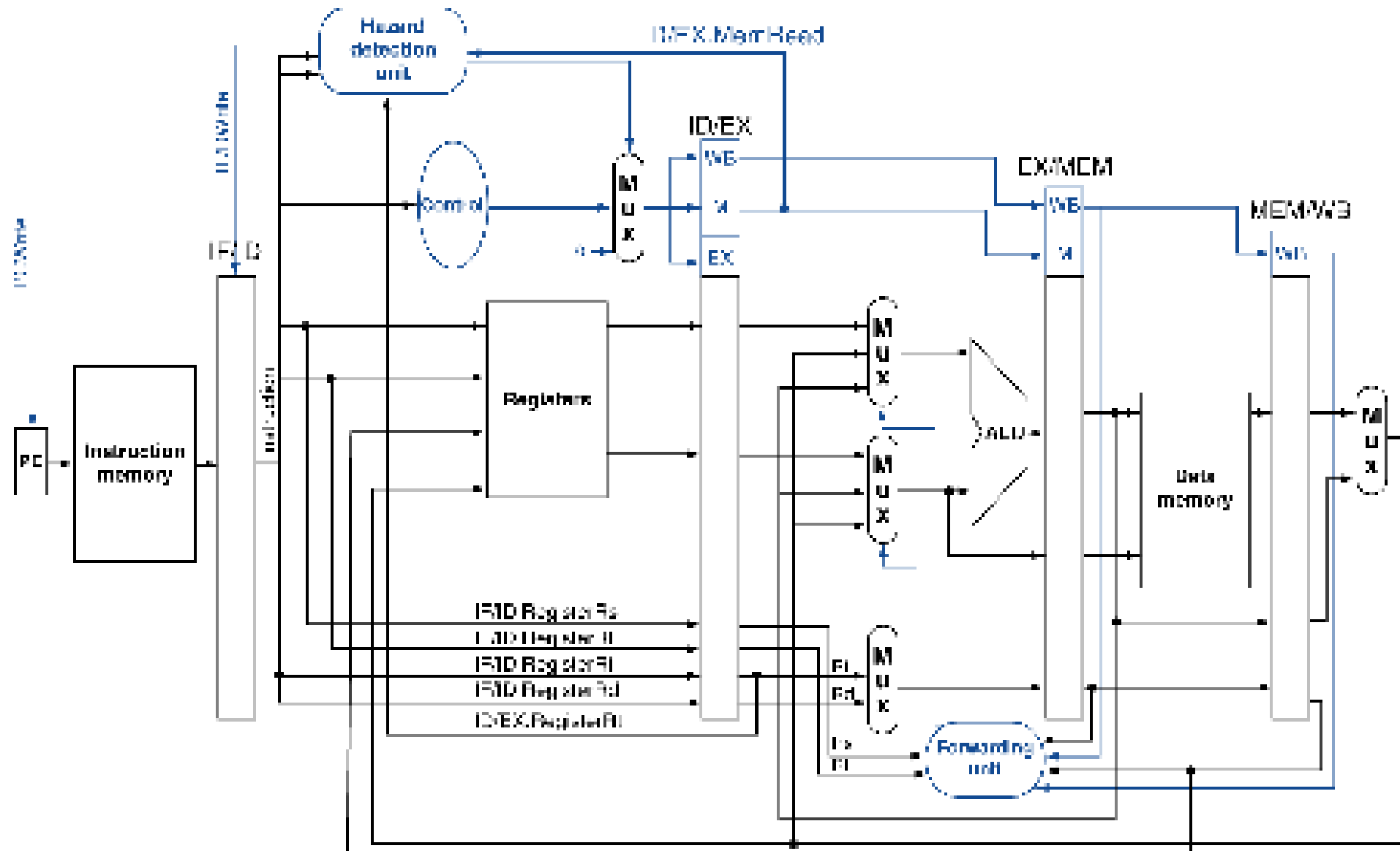
Stall/Bubble in the Pipeline



Stall/Bubble in the Pipeline



Datapath with Hazard Detection



Stalls and Performance

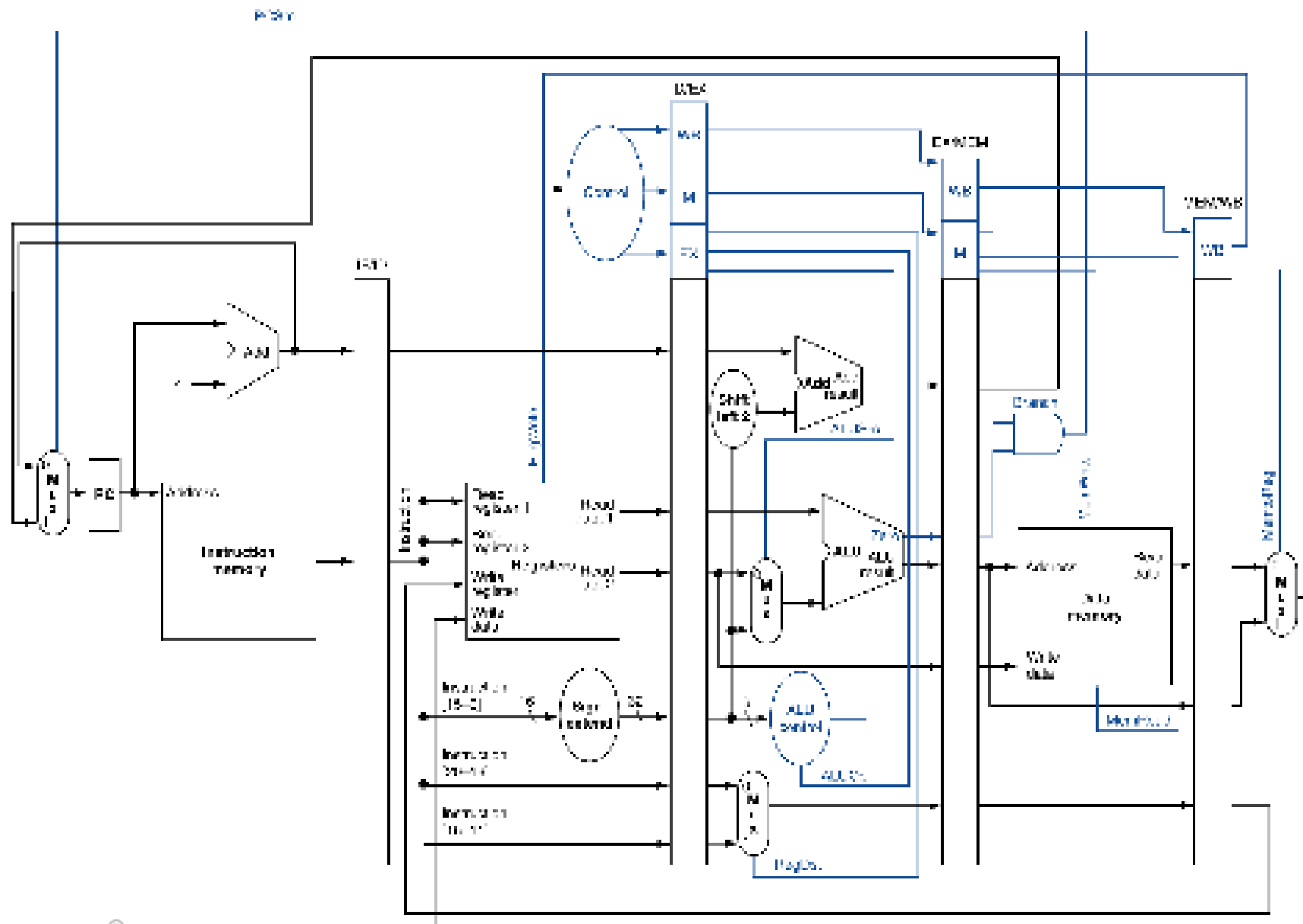
The BIG Picture

- 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

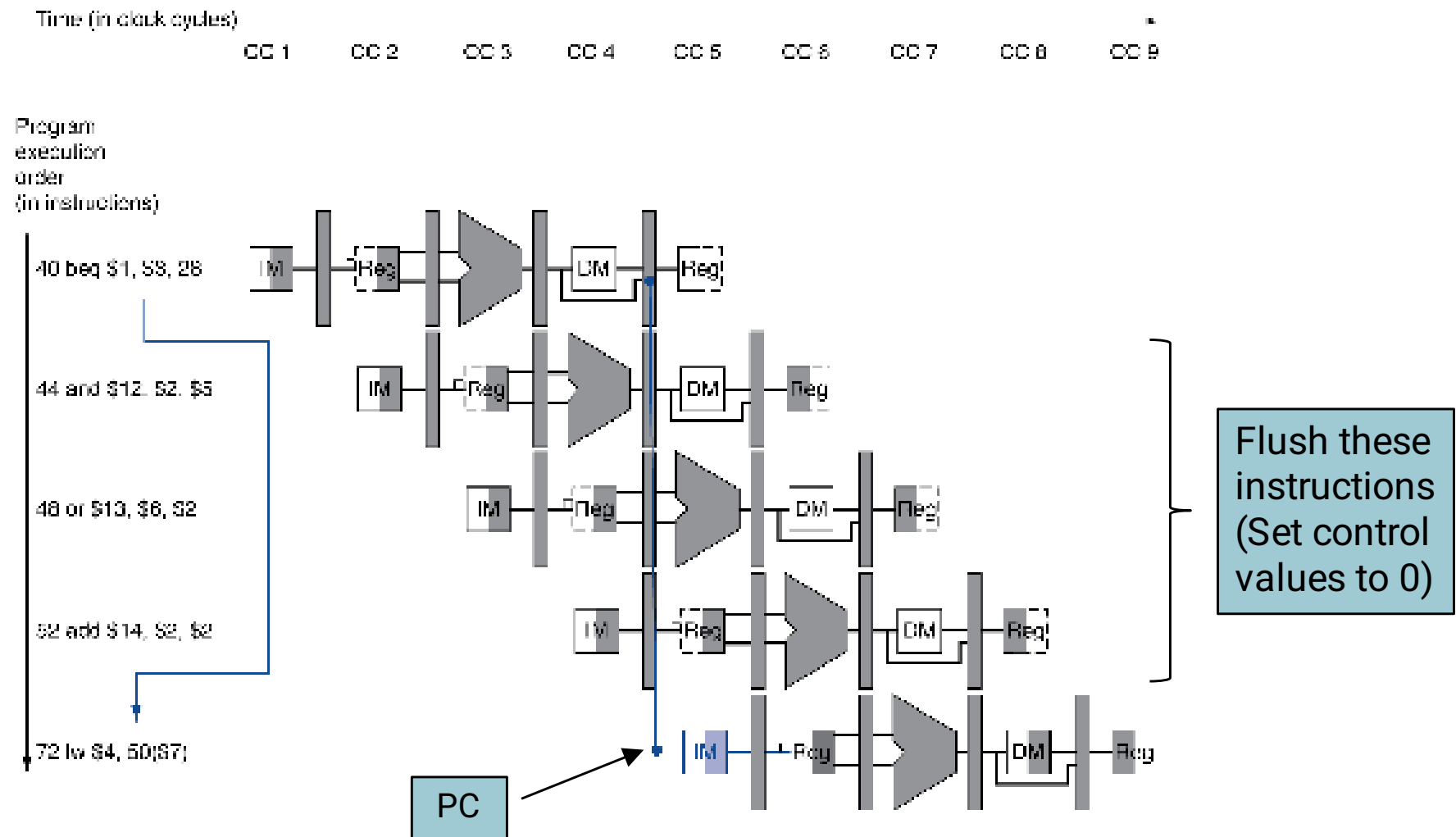
- Hazards involving branches
- **A new instruction must be fetched at every clock cycle to sustain the pipeline.**
- In our design the decision about whether to branch doesn't occur until the MEM pipeline stage comes.
- **This delay in determining the proper instruction to fetch is called a control hazard**

Pipelined Control



Branch Hazards

- If branch outcome determined in MEM



Branch hazards

- They are simple to understand
- They occur less frequently than data hazards
- There is nothing as effective solution against control hazards as **forwarding** is against data hazards
- Two simpler techniques
 - Assume branch not taken (static prediction)
 - Dynamic branch prediction

- A common improvement over branch stalling is to assume that the branch will **not be taken** and thus continue execution down the sequential instruction stream.
- If the branch is **taken** the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target.
- Discarding the instructions means, we must be able to flush the instructions in the IF, ID and EX stages of pipe.

Reducing the Delay of Branches

- One way to improve branch performance is to reduce the **cost of the taken branch**
- If we move the branch execution earlier in the pipeline, then fewer instructions need be flushed
- Branches rely only on simple tests
- So move branch decision up into ID stage

Reducing Branch Delay

- Moving the branch decision up requires two actions to occur earlier
 - Computing the branch target address
 - Evaluating the branch decision
- Move hardware to determine outcome to ID stage
 - Target address adder
 - Register comparator

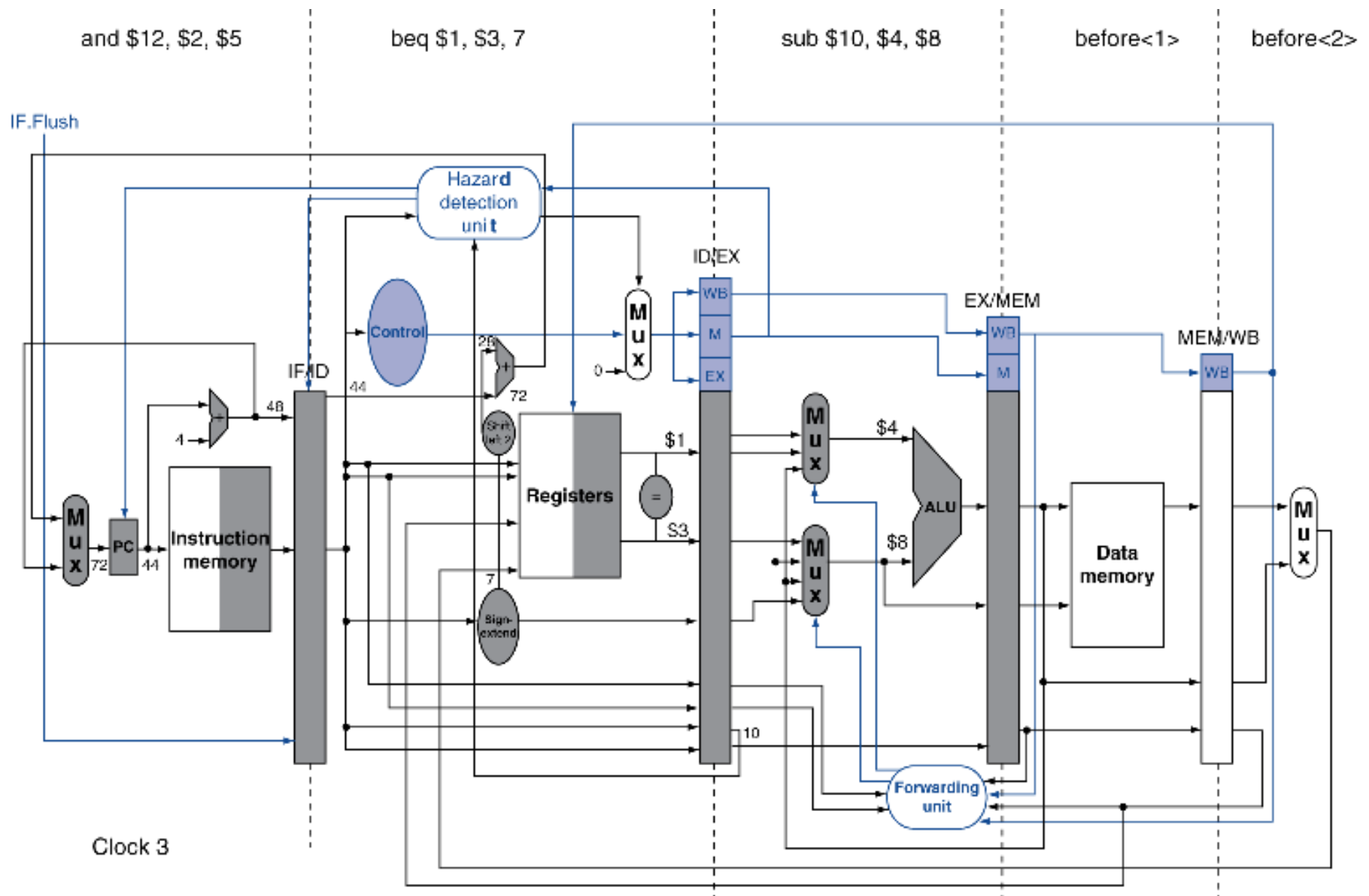
- Example: branch taken

```
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
```

...

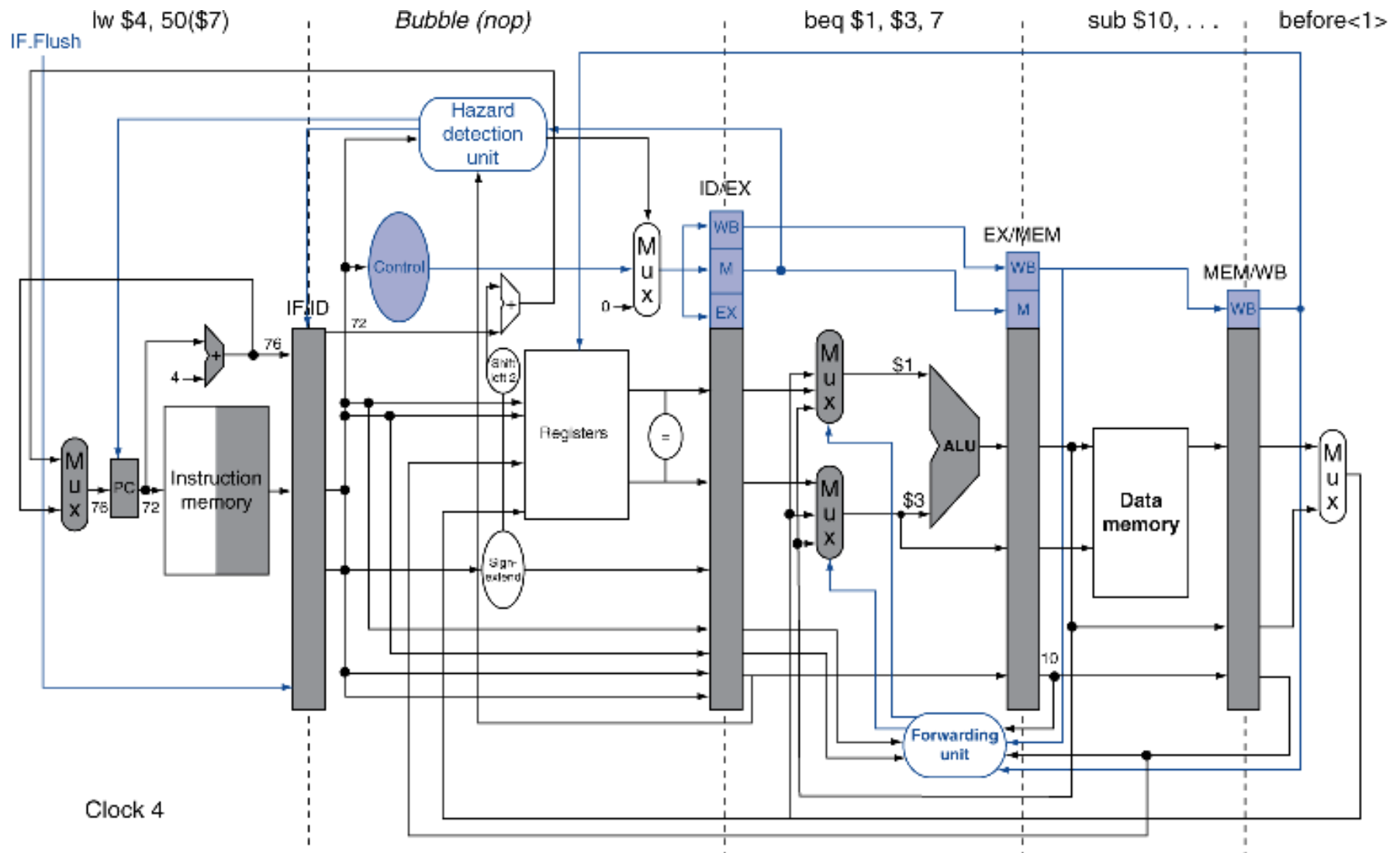
```
72: lw  $4, 50($7)
```

Example: Branch Taken



Clock 3

Example: Branch Taken



Clock 4

- Harder part is the branch decision itself
- Moving the branch test to the ID stage implies **additional forwarding and hazard detection hardware**, since a branch dependent on a result still in the pipeline
- So we will need to forward results to the equality test logic that operates during ID.

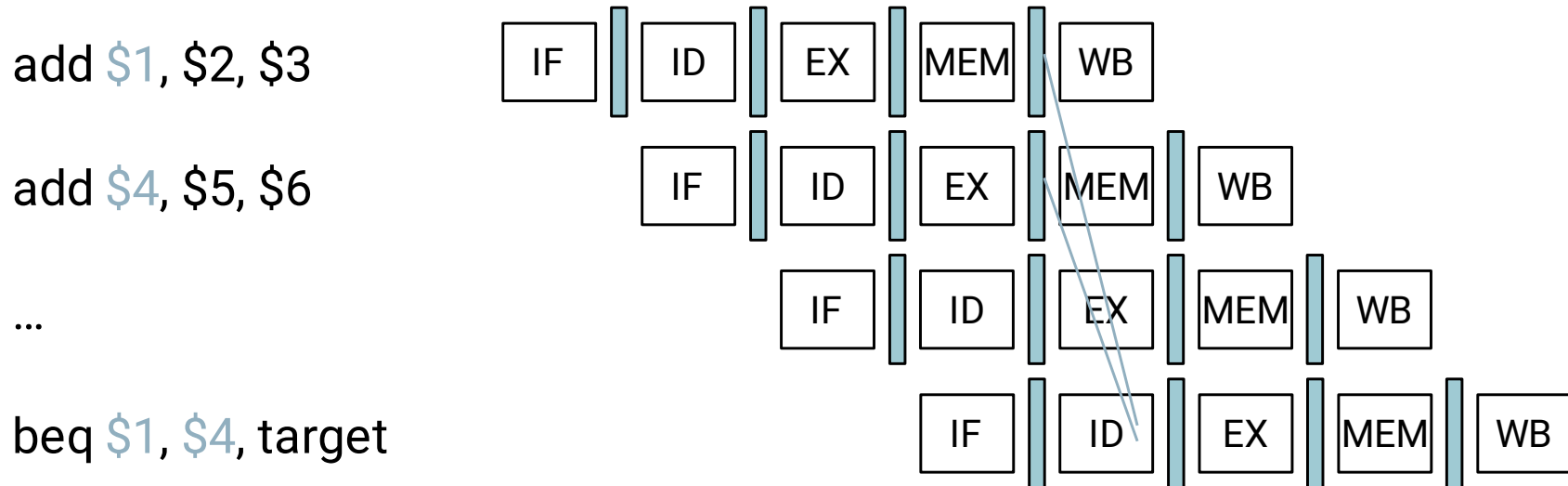
2 complicating factors

- The introduction of equality test unit in ID will require new **forwarding logic**. The bypassed source operands can come from the ALU/MEM or MEM/WB stages.
- The values in a branch comparison are needed during ID but may be produced later in time, a **stall** will be needed.

- **Despite these difficulties**, moving the branch execution to the ID stage is an **improvement**
- Because it reduces the penalty of a branch to only one instruction if the branch is taken(The one currently being fetched)

Data Hazards for Branches

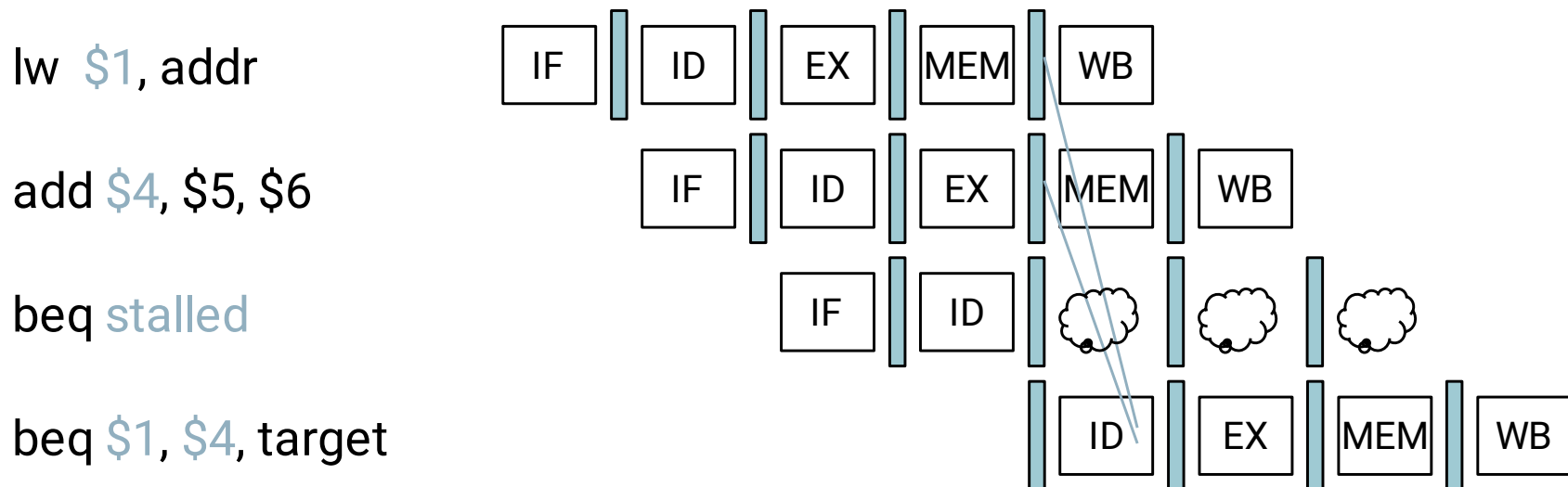
- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction



- 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



Data Hazards for Branches

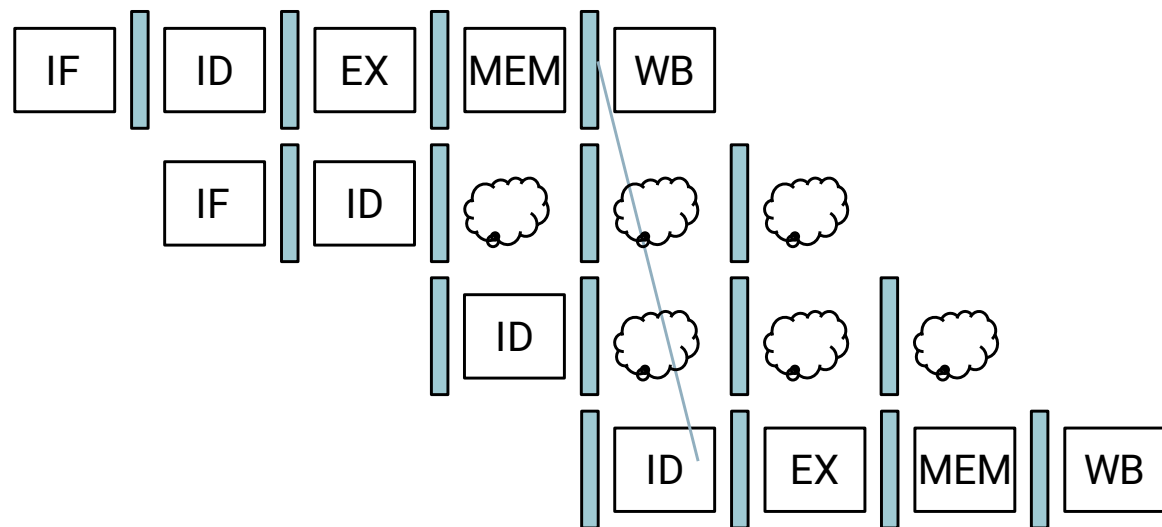
- If a comparison register is a destination of immediately preceding load instruction
 - Need 2 stall cycles

lw \$1, addr

beq stalled

beq stalled

beq \$1, \$0, target



- To flush instructions in IF stage, we add a control line called IF.Flush that zeros the instruction field of IF/ID pipeline register
- Clearing the register transforms the fetched instruction into *nop*

Dynamic branch prediction

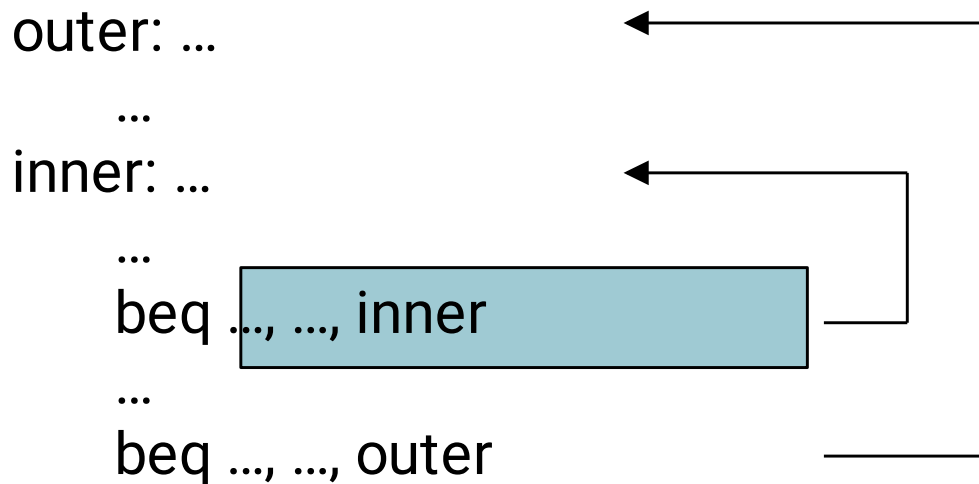
- In deeper and superscalar pipelines, branch penalty is more significant
- **With more hardware** it is possible to try to predict branch behavior **during program execution time**
- One approach is to lookup the address of the instruction to **see if a branch was taken the last time this instruction was executed**

- One implementation of that approach is **branch history table(BHT)**
- BHT is a small memory indexed by the lowest portion of the address of the branch instruction
- The memory contains a bit that says whether the branch was recently taken or not

- Starts off as T, flips when ever the branch behaves opposite to prediction
- Limitation: even if a branch is almost always taken, we will predict incorrectly twice, rather than once, when it is not taken

1-Bit Predictor: Shortcoming

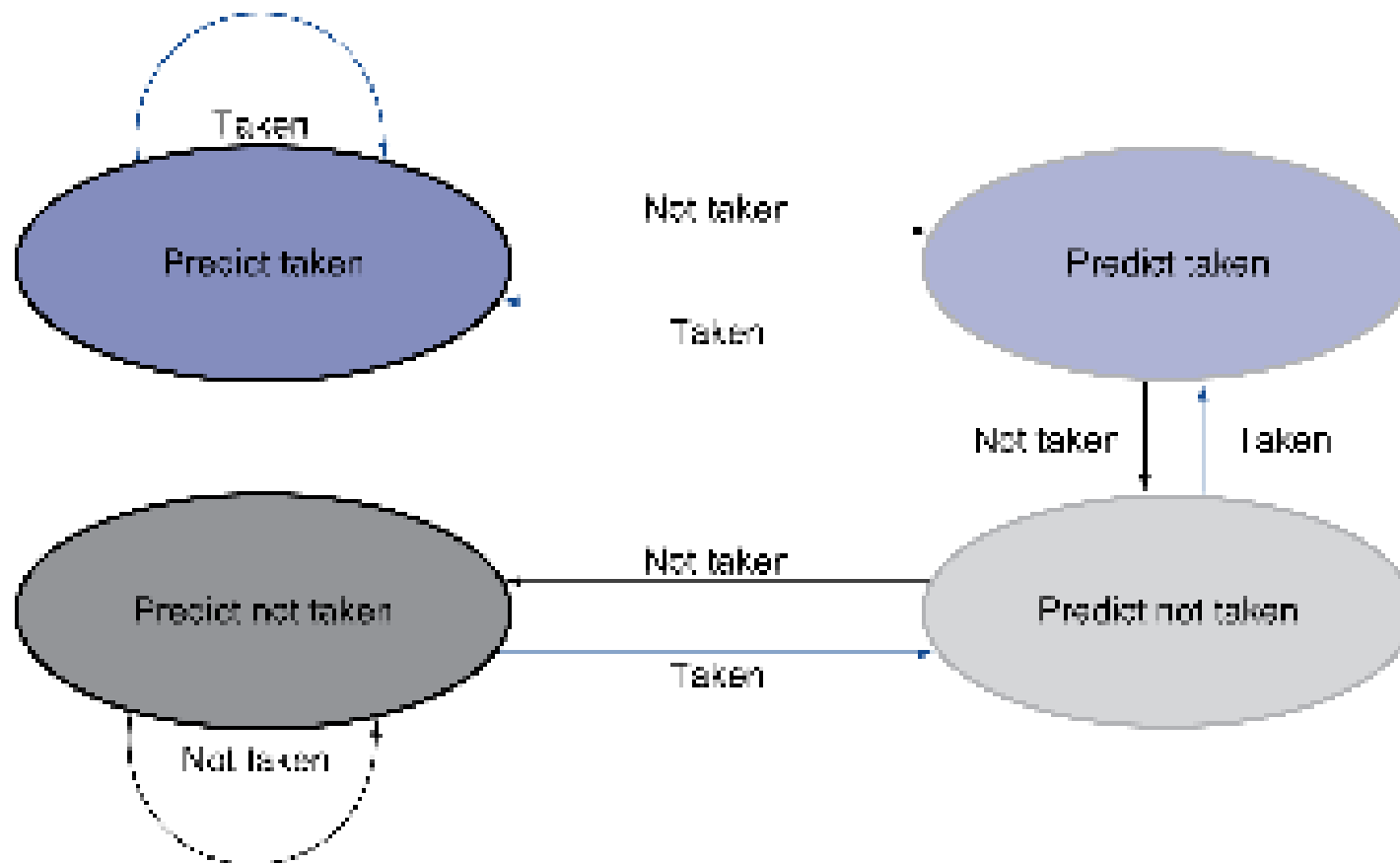
- Inner loop branches mispredicted **twice!**



- 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

- Control is the most challenging aspect of processor design
 - One of the hardest part of control is implementing exceptions
- Exceptions: events other than branches or jumps that change the normal flow of instruction execution
- Were Initially created to handle unexpected events from within the processor

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

Exception / Interrupt

- Exception : An unscheduled event that disrupts program execution
- Interrupt: An execution that comes from outside of the processor

Type of event	From where	MIPS technology
I/O device request	External	Interrupt
Invoke the OS from user pgm	Internal	Exception
Arithmetic overflow	Internal	Exception
Undefined instruction	Internal	Exception
h/w malfunctions	Either	Interrupt or exception

- Detecting exceptional conditions and taking the appropriate actions is often on the critical timing path of the processor, which determines the clock cycle time and thus performance

Handling Exceptions

- eg: undefined instructions and overflow
 - Add \$s1, \$s2, \$s1
- The basic action that the processor must perform when an exception occur is to
 - Save PC address of offending (or interrupted) instruction
 - In MIPS: Exception Program Counter (EPC)
 - Save indication of the problem
 - In MIPS: Cause register
 - We'll assume 1-bit
 - 0 for undefined opcode, 1 for overflow
 - Transfer control to OS(Jump to handler at 8000 00180)

- The OS then take appropriate action, which may involve
 - Providing service to user pgm
 - Taking some predefined action in response to overflow
 - Stopping execution of the pgm and reporting the error
- After performing what ever action is required because of exception, the OS can terminate/continue the pgm execution(using EPC to determine where to start)

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

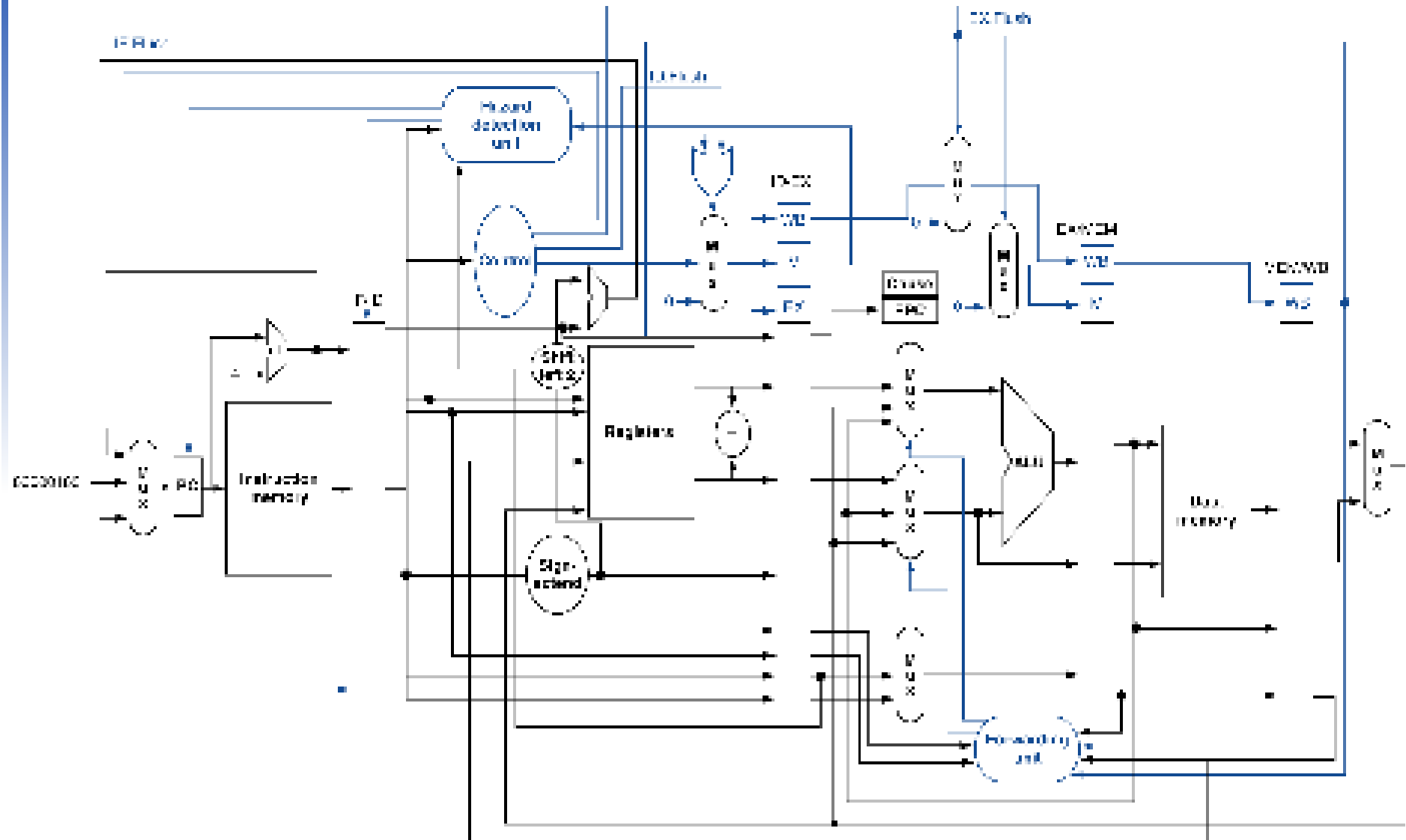
Handler Actions

- 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

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

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)
```

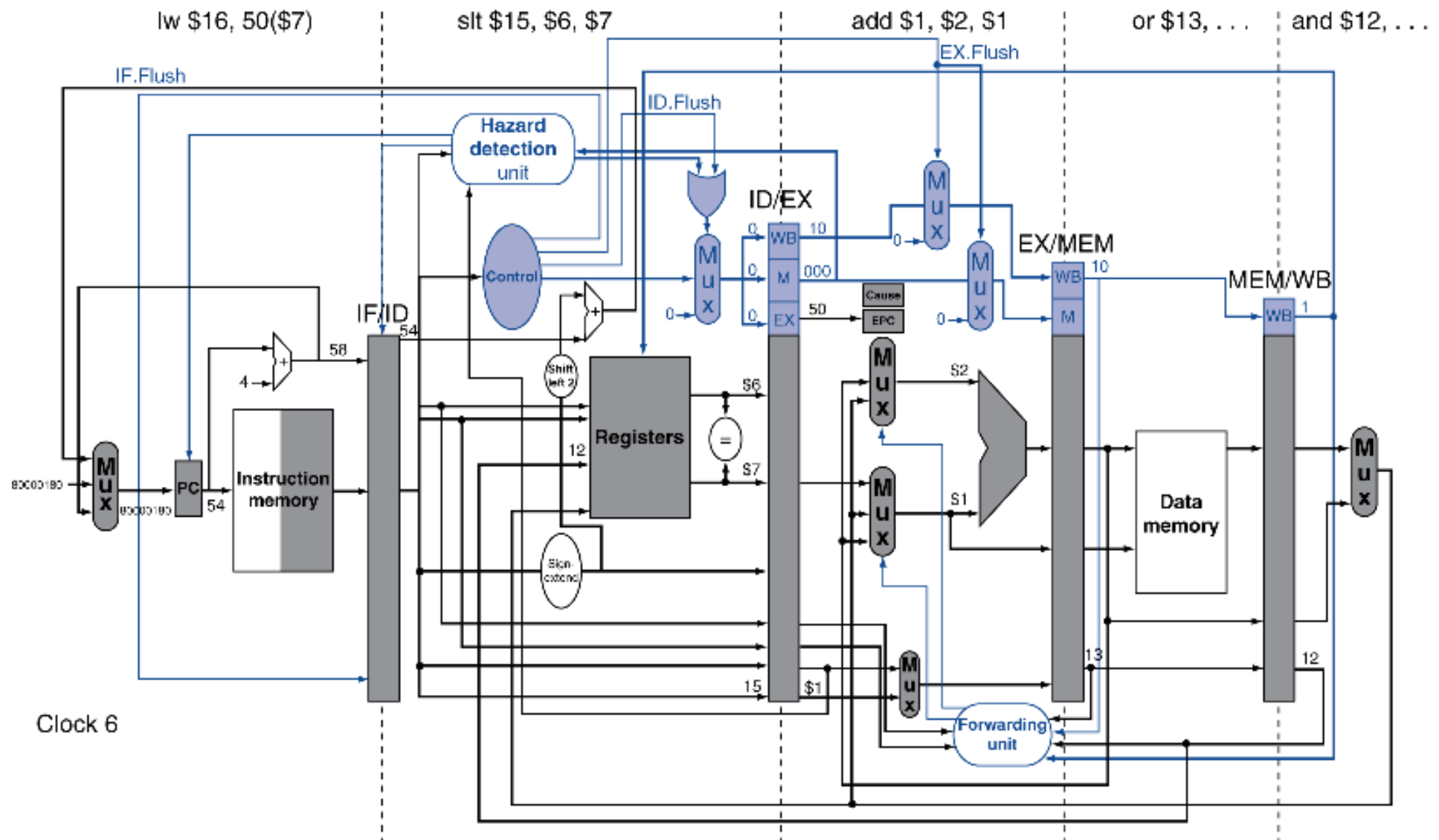
...

- Handler

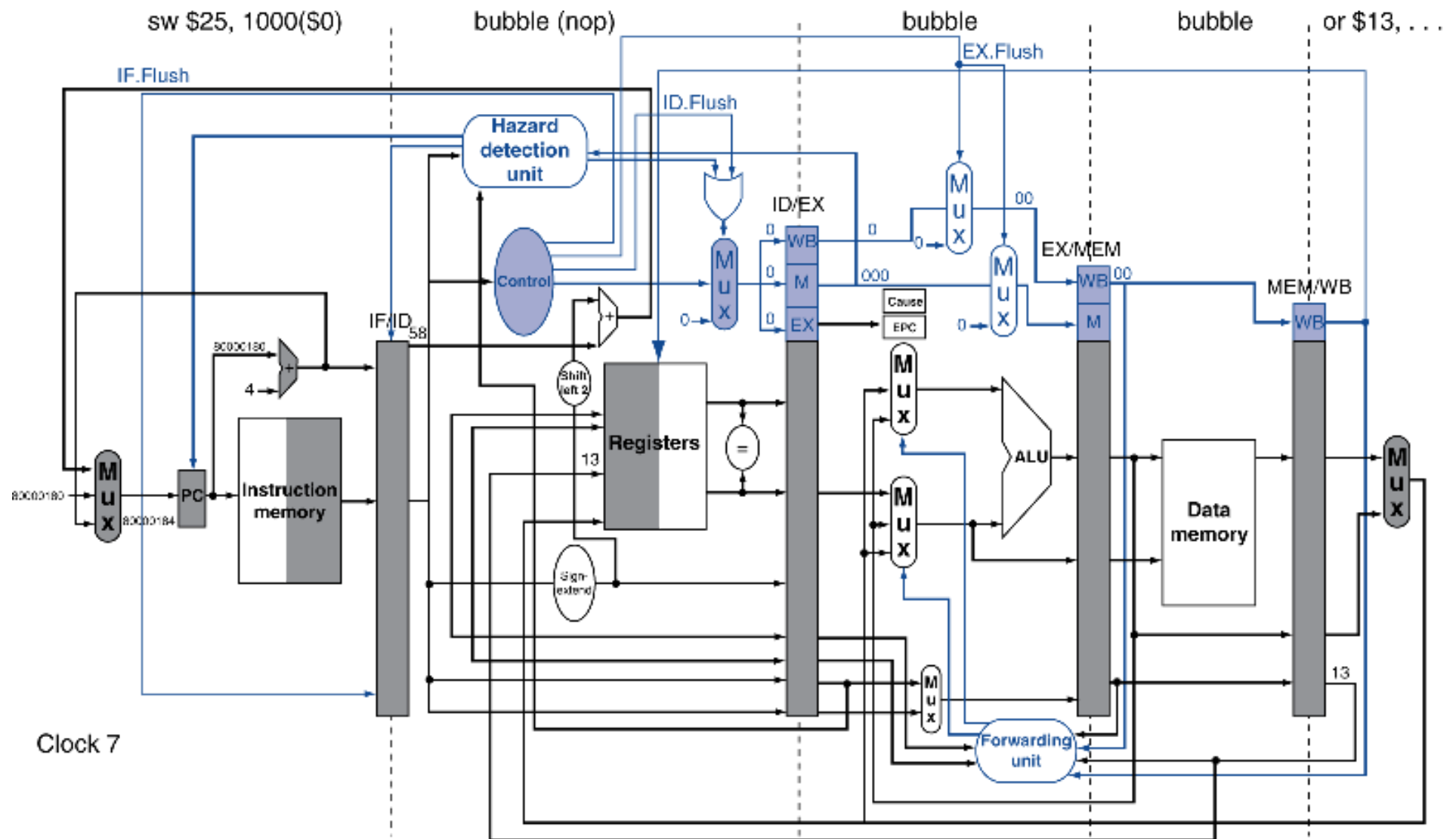
```
80000180 sw $25, 1000($0)
80000184 sw $26, 1004($0)
```

...

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
- In complex pipelines
 - Multiple instructions issued per cycle
 - Out-of-order completion
 - Maintaining exceptions is difficult!

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

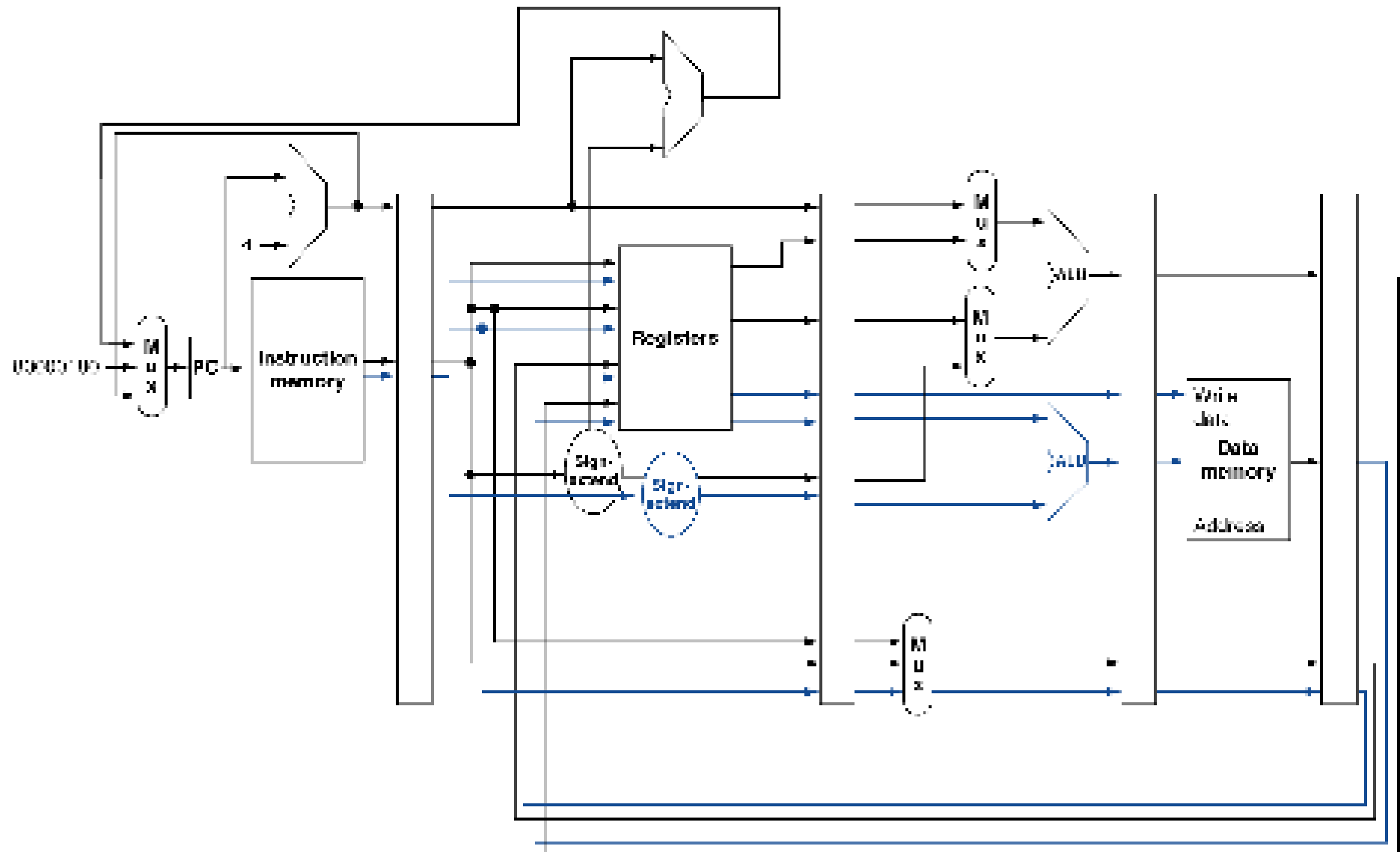
- Static multiple issue
 - Compiler groups instructions to be issued together
 - Packages them into “issue slots”
 - Compiler detects and avoids hazards
- Dynamic multiple issue
 - CPU examines instruction stream and chooses instructions to issue each cycle
 - Compiler can help by reordering instructions
 - CPU resolves hazards using advanced techniques at runtime

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						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS with Static Dual Issue



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

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