

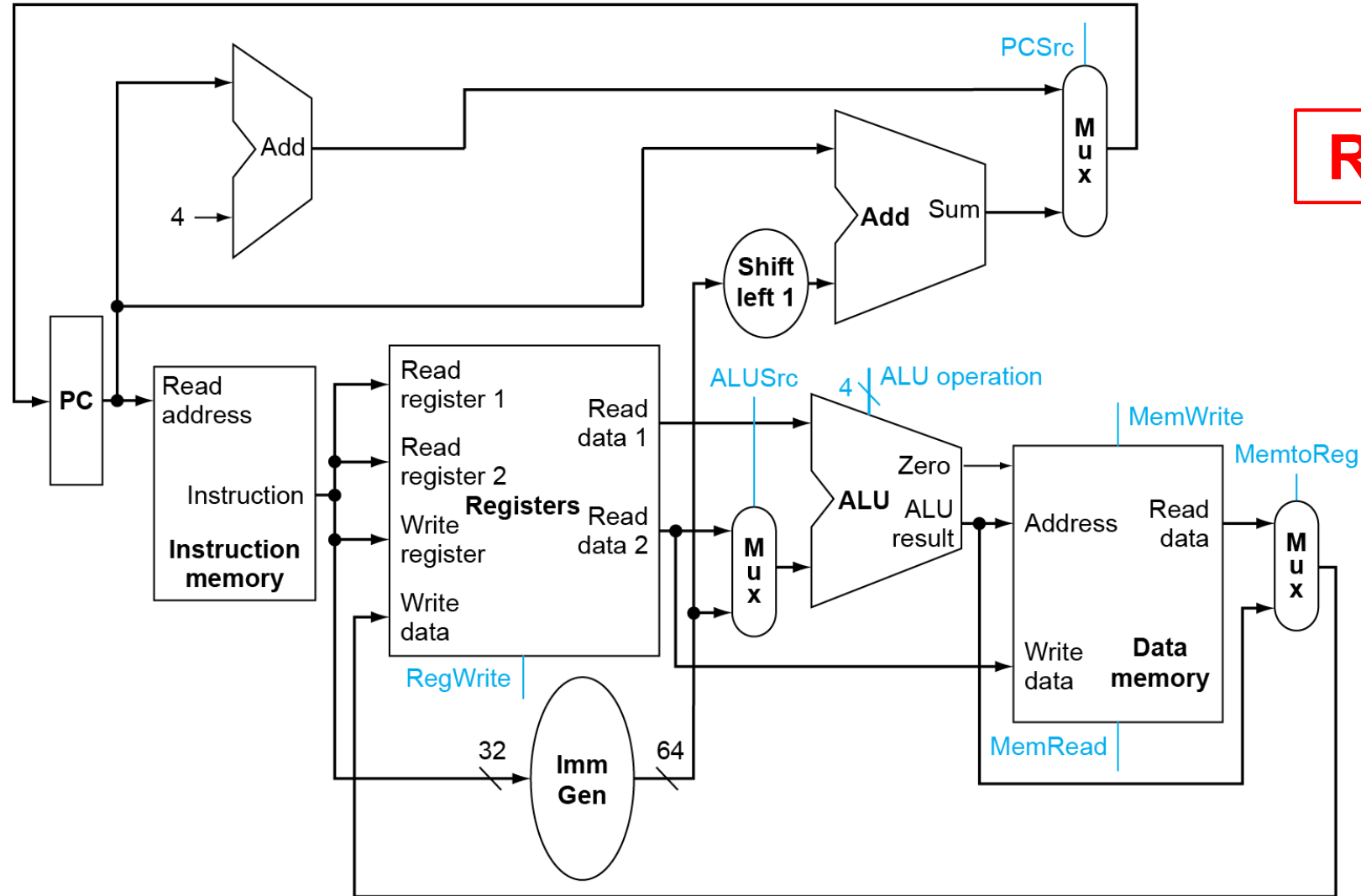
# Chapter 4

## The Processor

*Adapted by Prof. Gheith Abandah*

*Last updated 15/10/2023*

# Single-Cycle RISC-V Processor Implementation



**Review**

# Contents

- 4.6 An Overview of Pipelining
- 4.7 Pipelined Datapath and Control
- 4.8 Data Hazards: Forwarding versus Stalling
- 4.9 Control Hazards
- 4.10 Exceptions
- 4.11 Parallelism via Instructions
- 4.12 Putting it All Together: The Intel Core i7 6700 and ARM Cortex-A53
- 4.15 Fallacies and Pitfalls
- 4.16 Concluding Remarks

# Contents

---

## 4.6 An Overview of Pipelining

Pipelining Analogy

RISC-V Pipeline

Pipeline Performance

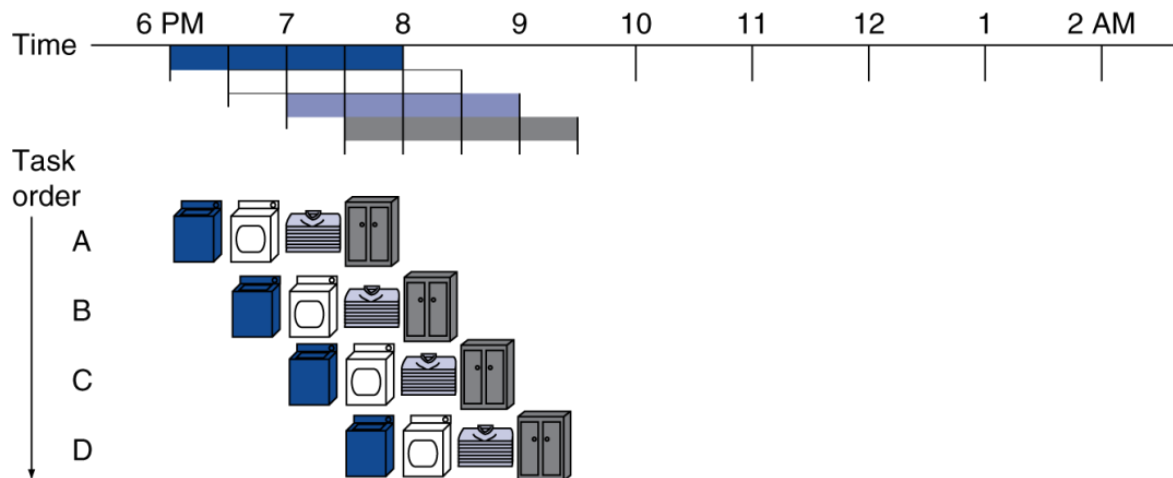
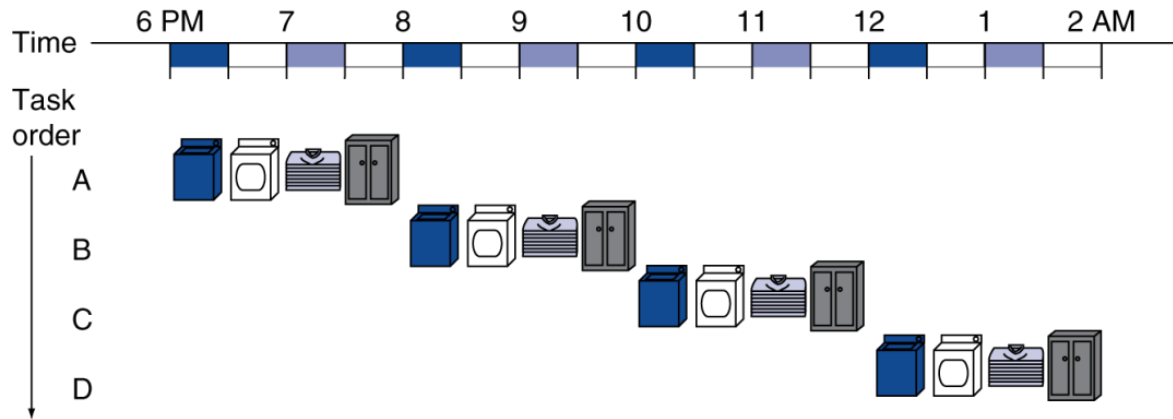
Pipelining and ISA Design

Pipeline Hazards

Pipeline Summary

# Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- For 4 loads:

- Speedup  
 $= 8/3.5 = 2.3$

- Non-stop ( $n \rightarrow \infty$ ):

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

# RISC-V 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

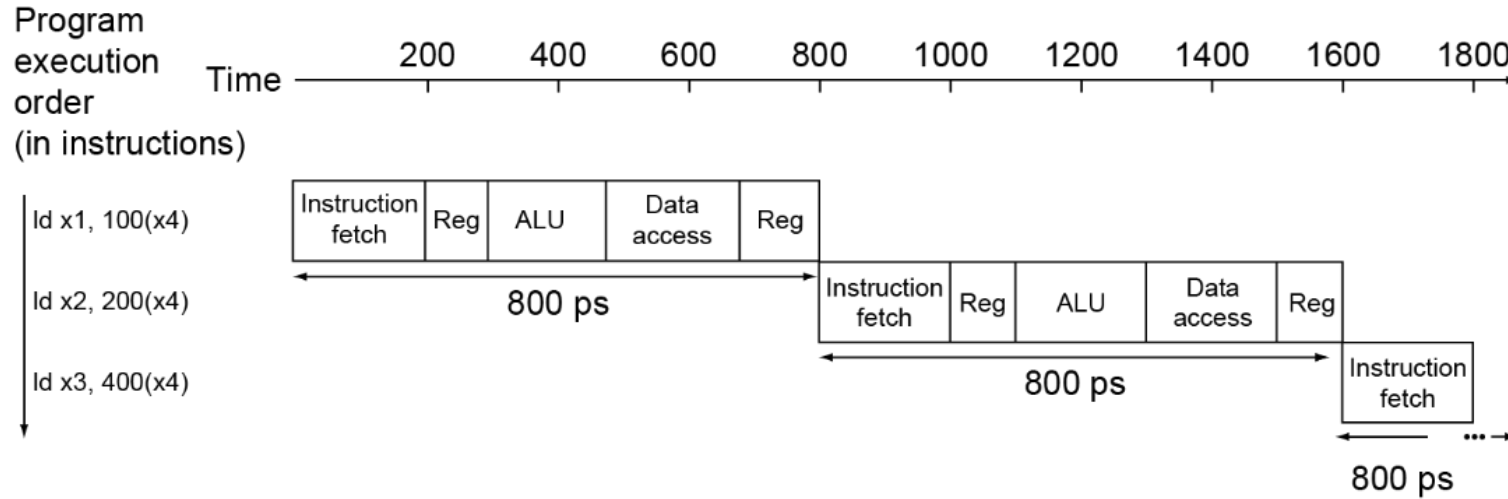
# Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Single-cycle datapath:

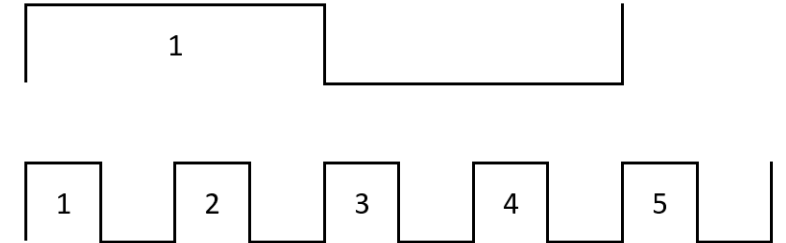
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- Pipelined datapath time = stages × 200 ps = 1,000 ps

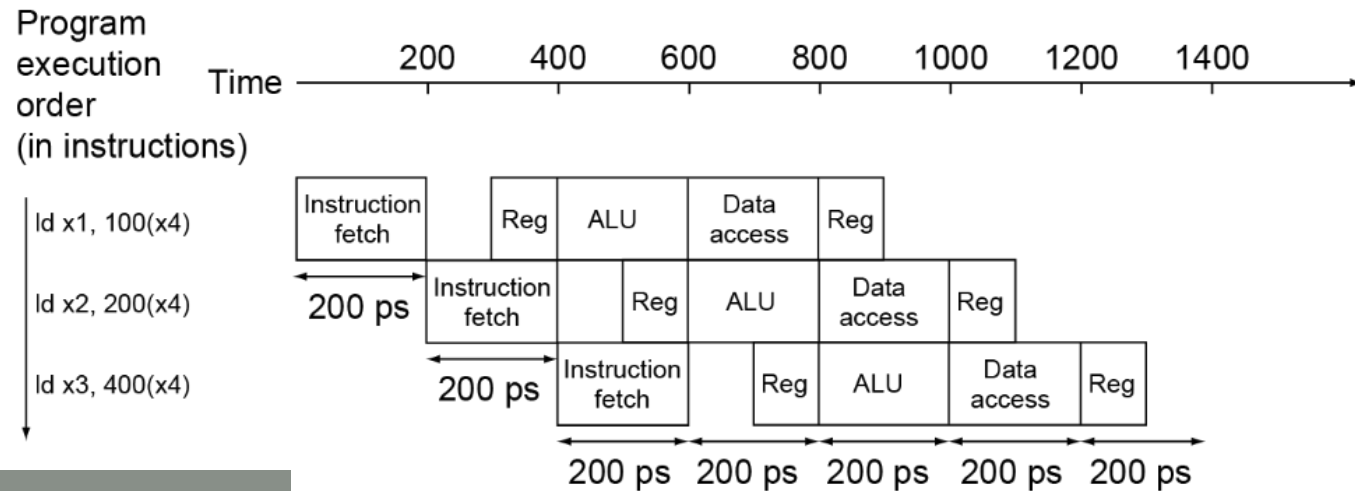
# Pipeline Performance



Single-cycle ( $T_c = 800\text{ps}$ )



Pipelined ( $T_c = 200\text{ps}$ )





# Pipeline Speedup

- If all stages are balanced

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

- Time between instructions<sub>pipelined</sub> =

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

- If not balanced, speedup is less.
- Speedup is due to the increased throughput.
  - Latency (time for each instruction) does not decrease

# Pipelining and ISA Design

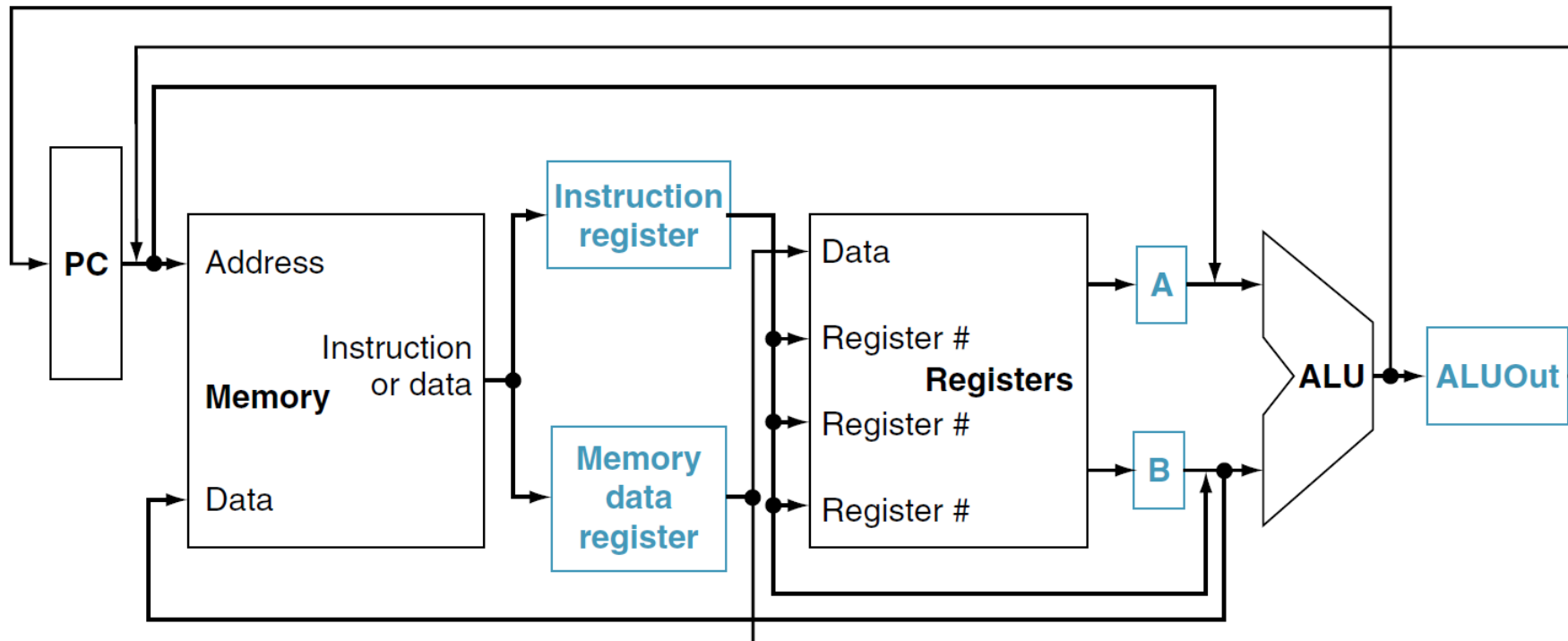
- RISC-V ISA is designed for pipelining.
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3<sup>rd</sup> stage, access memory in 4<sup>th</sup> stage

# Pipeline Hazards

- Situations that prevent starting the next instruction in the next cycle
  1. Structure hazards
    - When a required resource is busy for another instruction

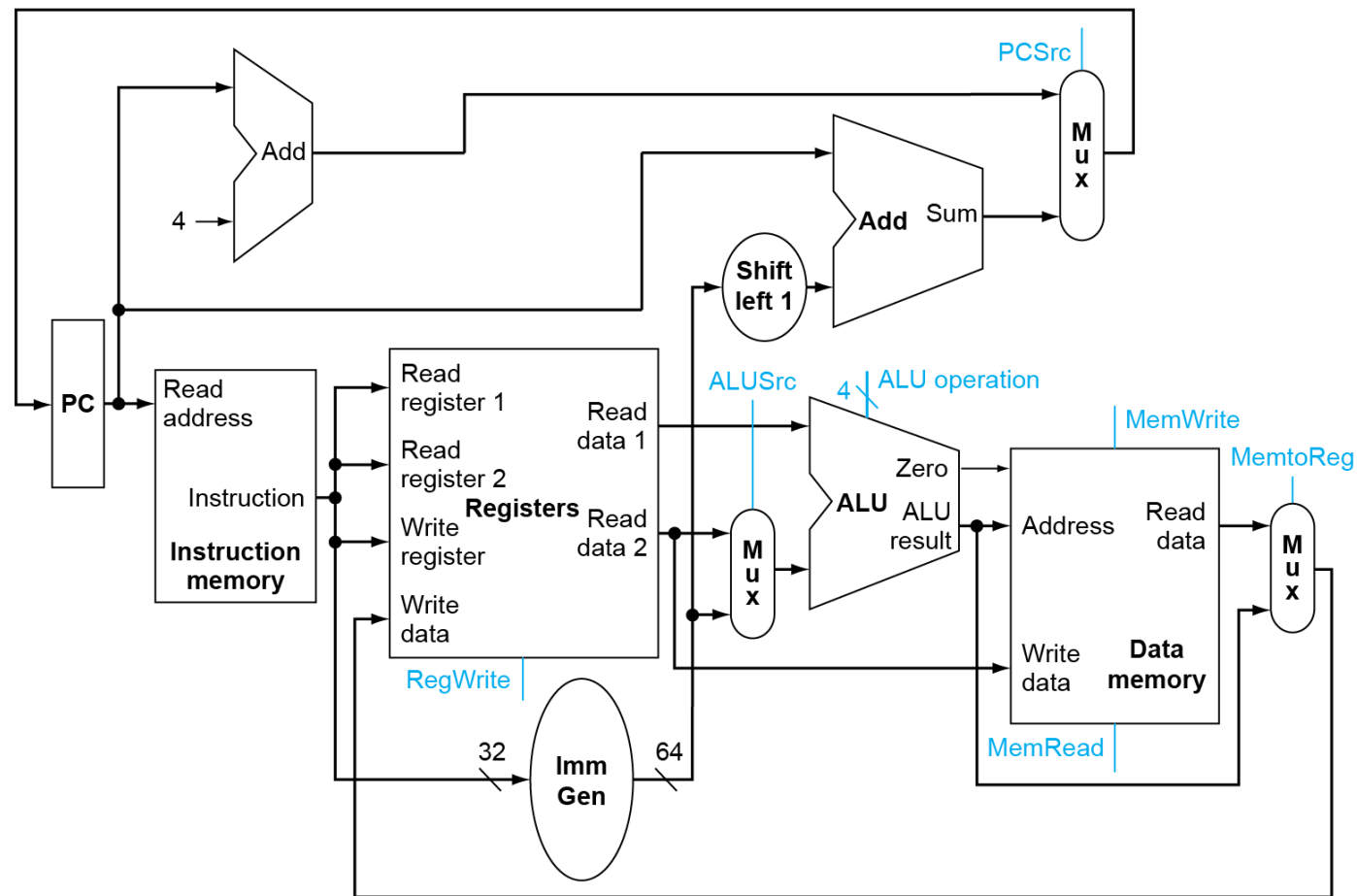
# Structural Hazards

- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle



# Structural Hazards

- **Solution:** Pipelined datapaths require separate instruction/data memories

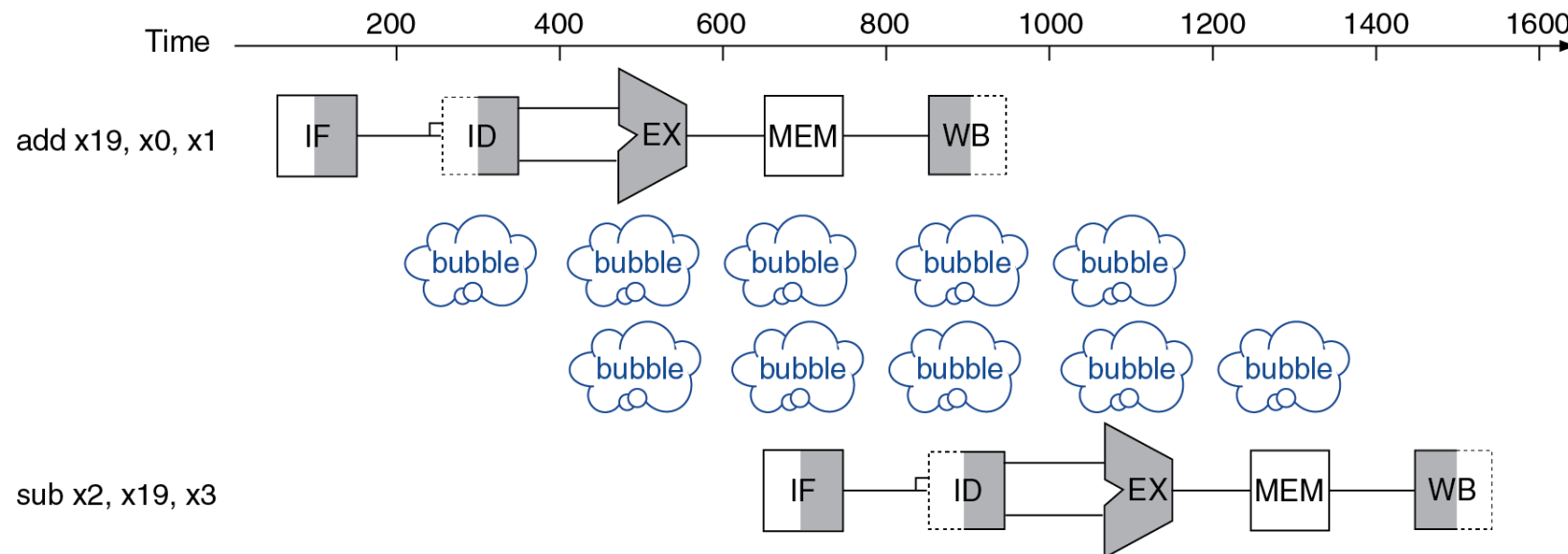


# Pipeline Hazards

- Situations that prevent starting the next instruction in the next cycle
  1. Structure hazards
    - When a required resource is busy for another instruction
  2. Data hazard
    - Due to data dependance. Need to wait for previous instruction to complete its data read/write

# Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add **x19**, x0, x1
  - sub x2, **x19**, x3



# Pipeline Hazards

- Situations that prevent starting the next instruction in the next cycle
  1. Structure hazards
    - When a required resource is busy for another instruction
  2. Data hazard
    - Due to data dependance. Need to wait for previous instruction to complete its data read/write
  3. Control hazard
    - Due to jump and branch instructions. Need to wait for the address of the next instruction to fetch



# Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on the branch outcome
  - Need to find whether the branch is taking or not
  - Need to calculate the branch target address for taken branch instruction
  - Therefore, fetching the next instruction should wait for the branch instruction result

# 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

# Contents

- 4.6 An Overview of Pipelining
- 4.7 Pipelined Datapath and Control
- 4.8 Data Hazards: Forwarding versus Stalling
- 4.9 Control Hazards
- 4.10 Exceptions
- 4.11 Parallelism via Instructions
- 4.12 Putting it All Together: The Intel Core i7 6700 and ARM Cortex-A53
- 4.15 Fallacies and Pitfalls
- 4.16 Concluding Remarks

# Contents

---

## 4.7 Pipelined Datapath and Control

- Five-Stage Pipeline

- Pipeline Operation

- Pipeline Diagrams

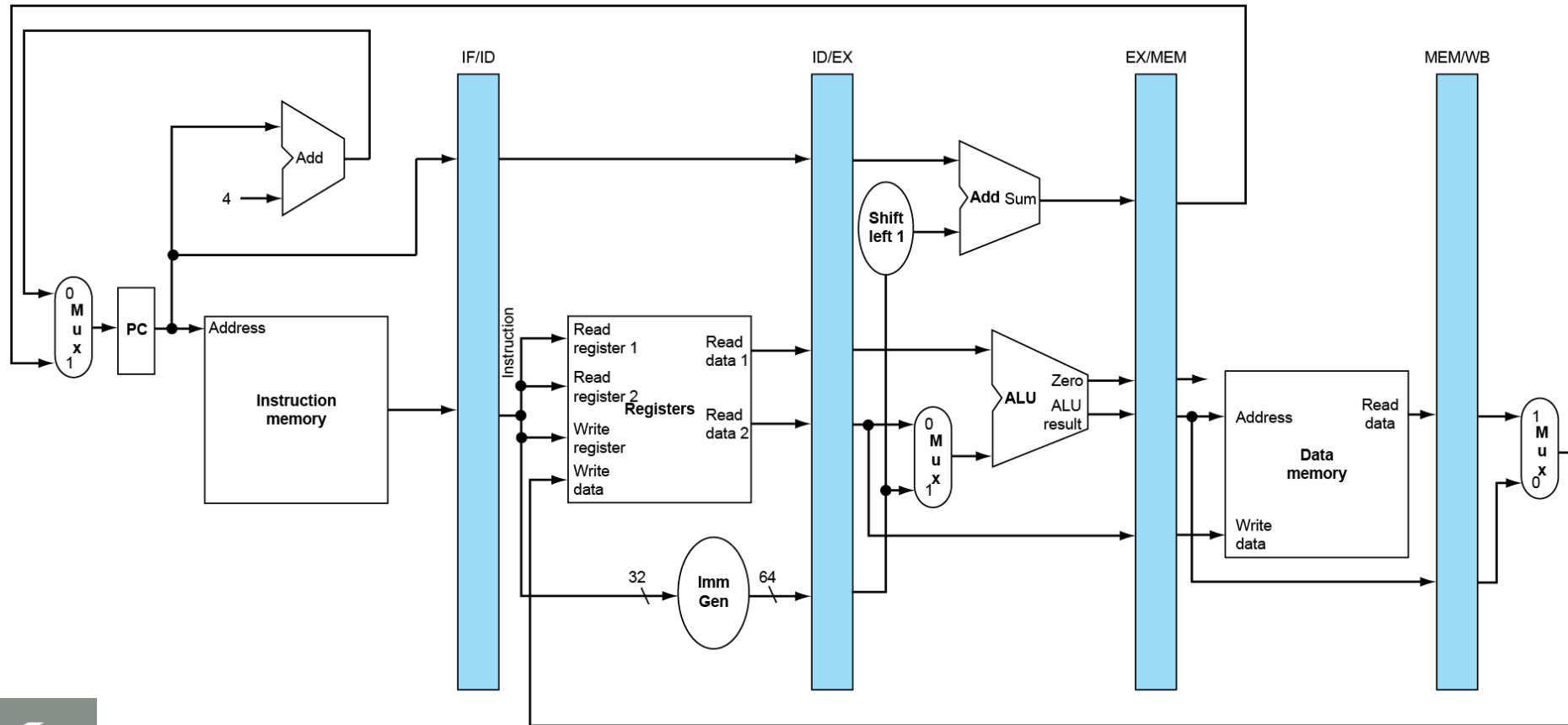
- Pipeline Control

# Five-Stage Pipeline

1. **IF (F)**: Fetch instruction from the instruction memory
2. **ID (D)**: Decode instruction and read operands
3. **EX (E)**: Execute operation or calculate address
4. **MEM (M)**: Memory access
5. **WB (W)**: Write result to the register

# Pipeline registers

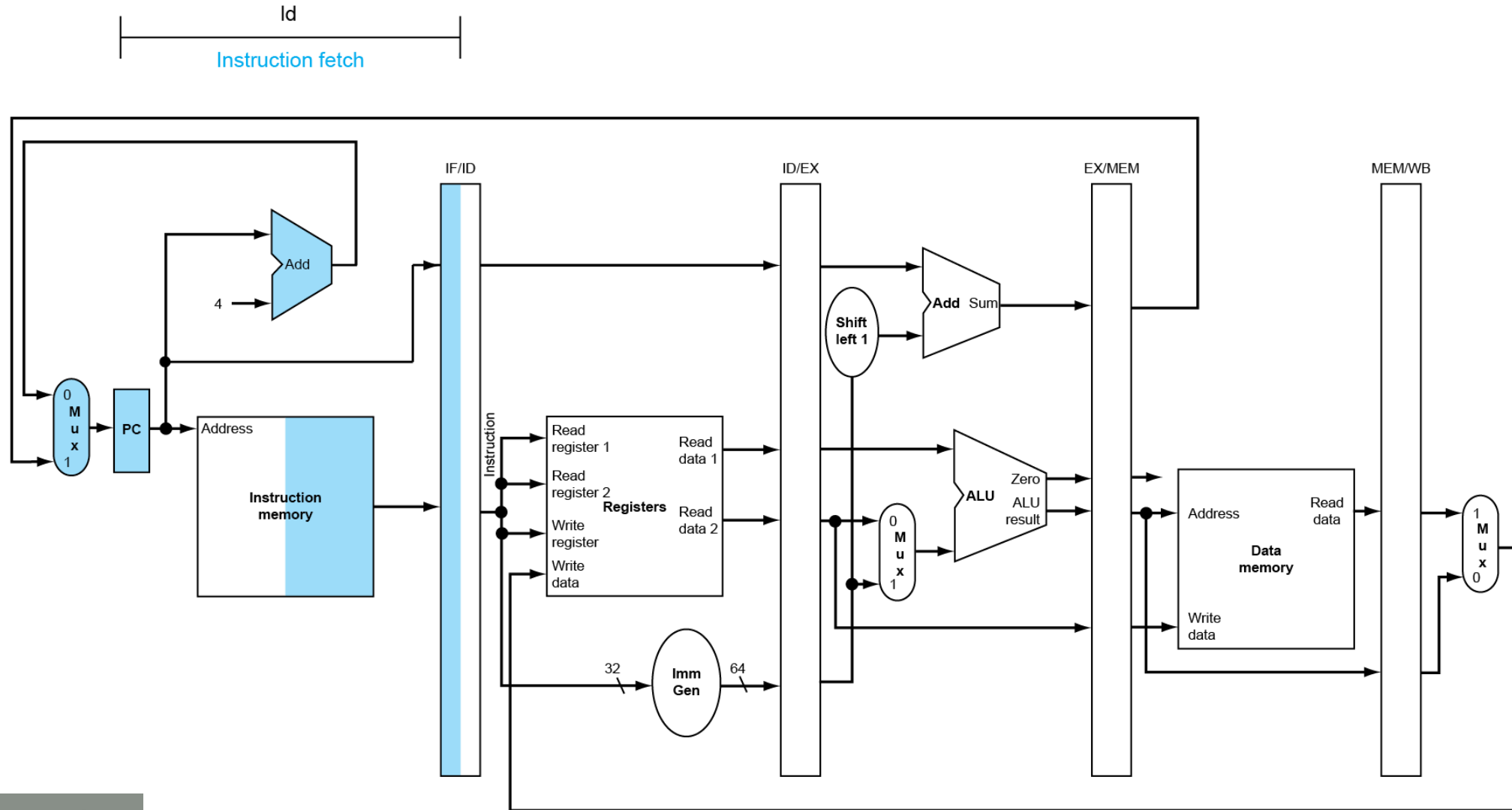
- Need registers between stages
  - To hold information produced in previous cycle



# Pipeline Operation

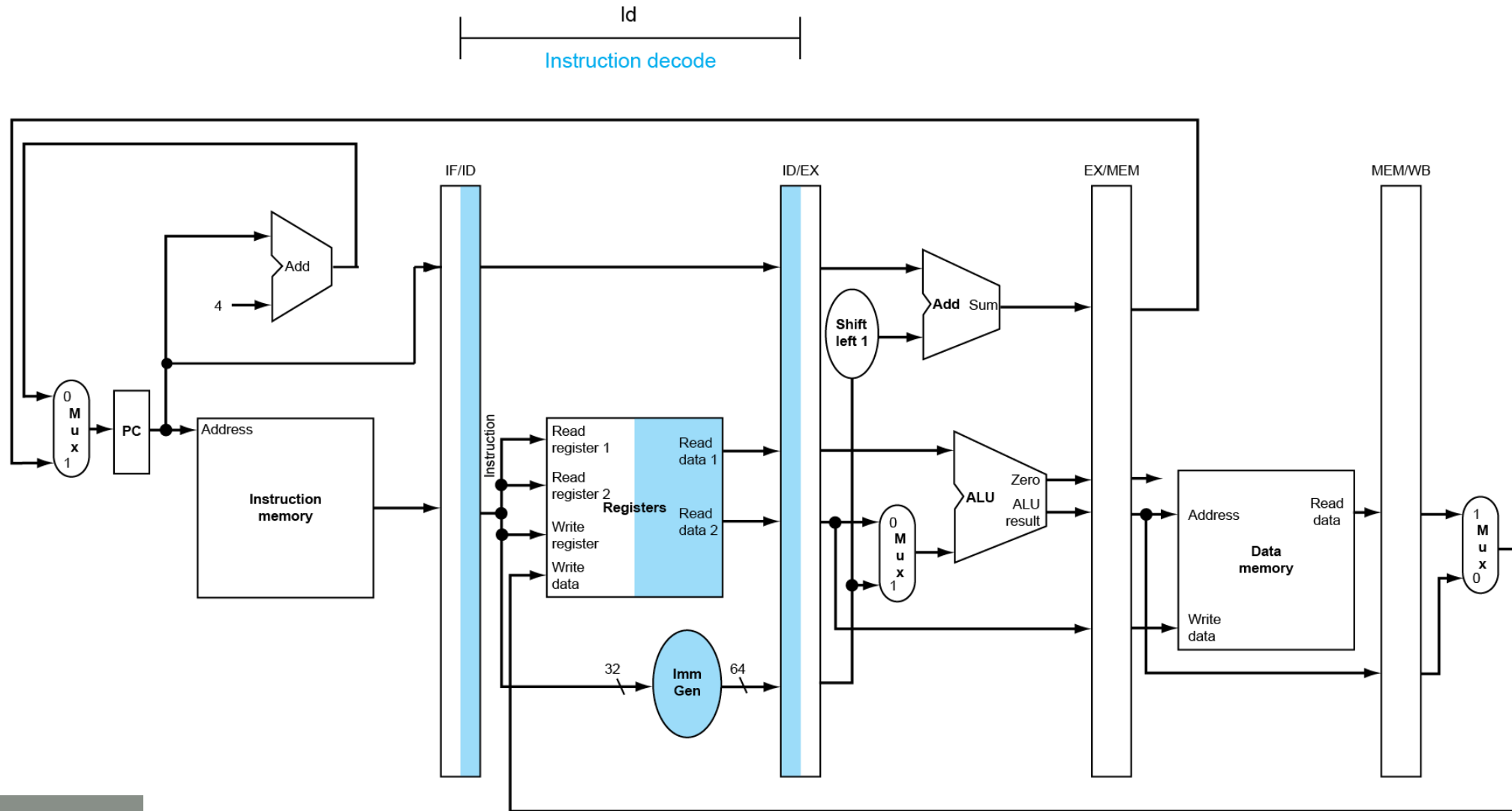
- We'll study cycle-by-cycle flow of instructions through the pipelined datapath.
- We'll look at “single-cycle” diagrams for load & store.

# IF for Load, Store, ...

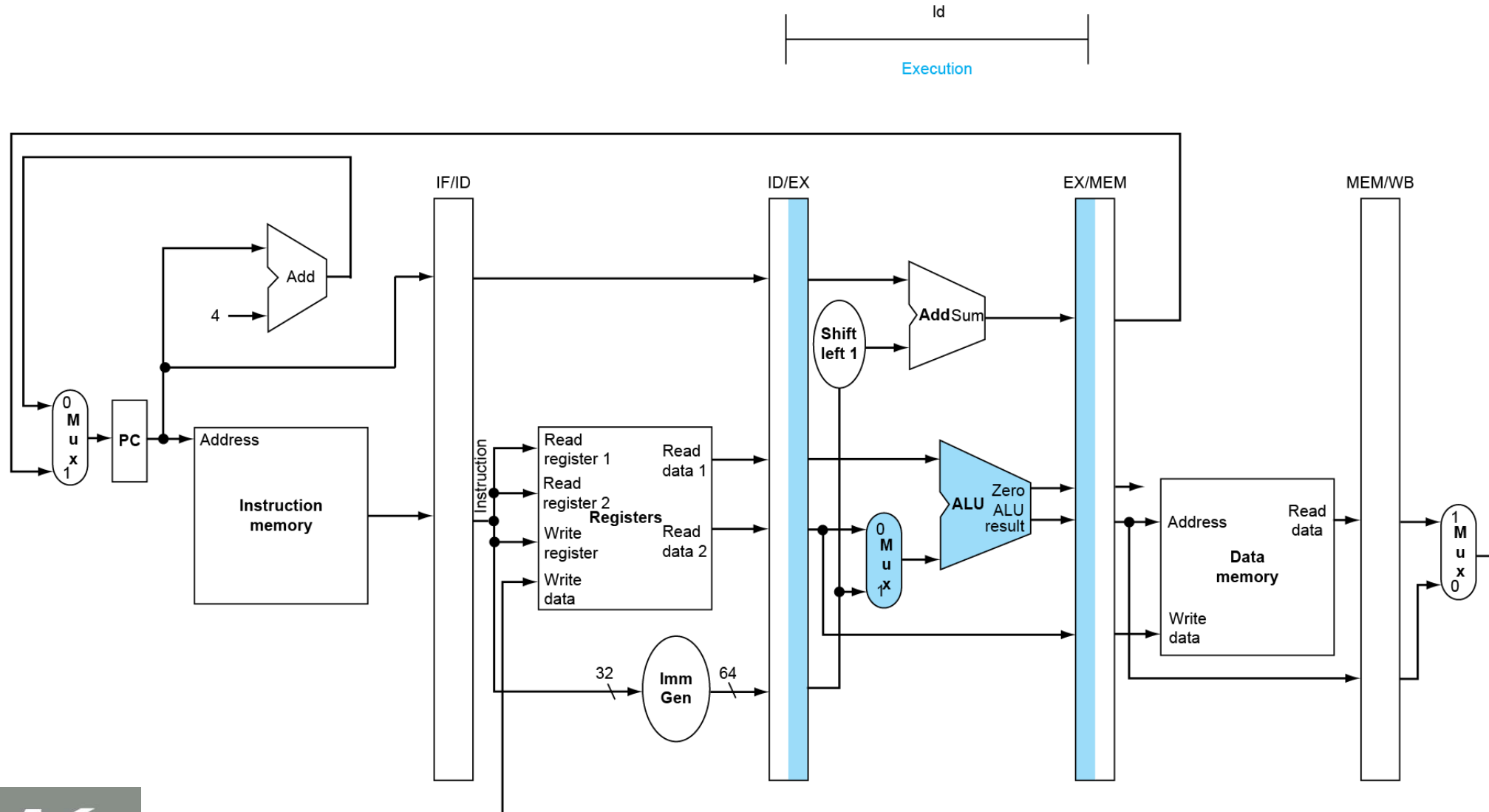




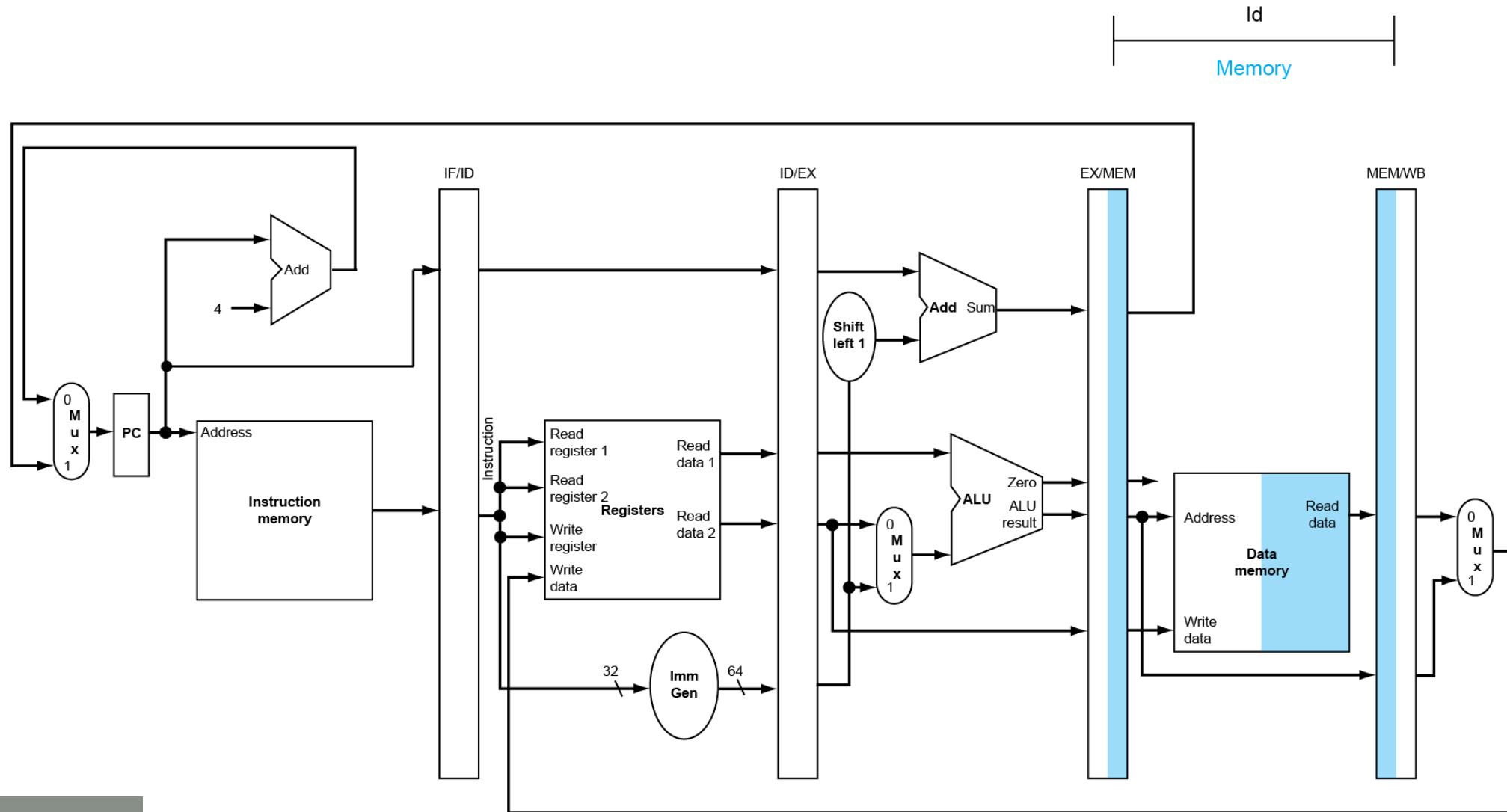
# ID for Load, Store, ...



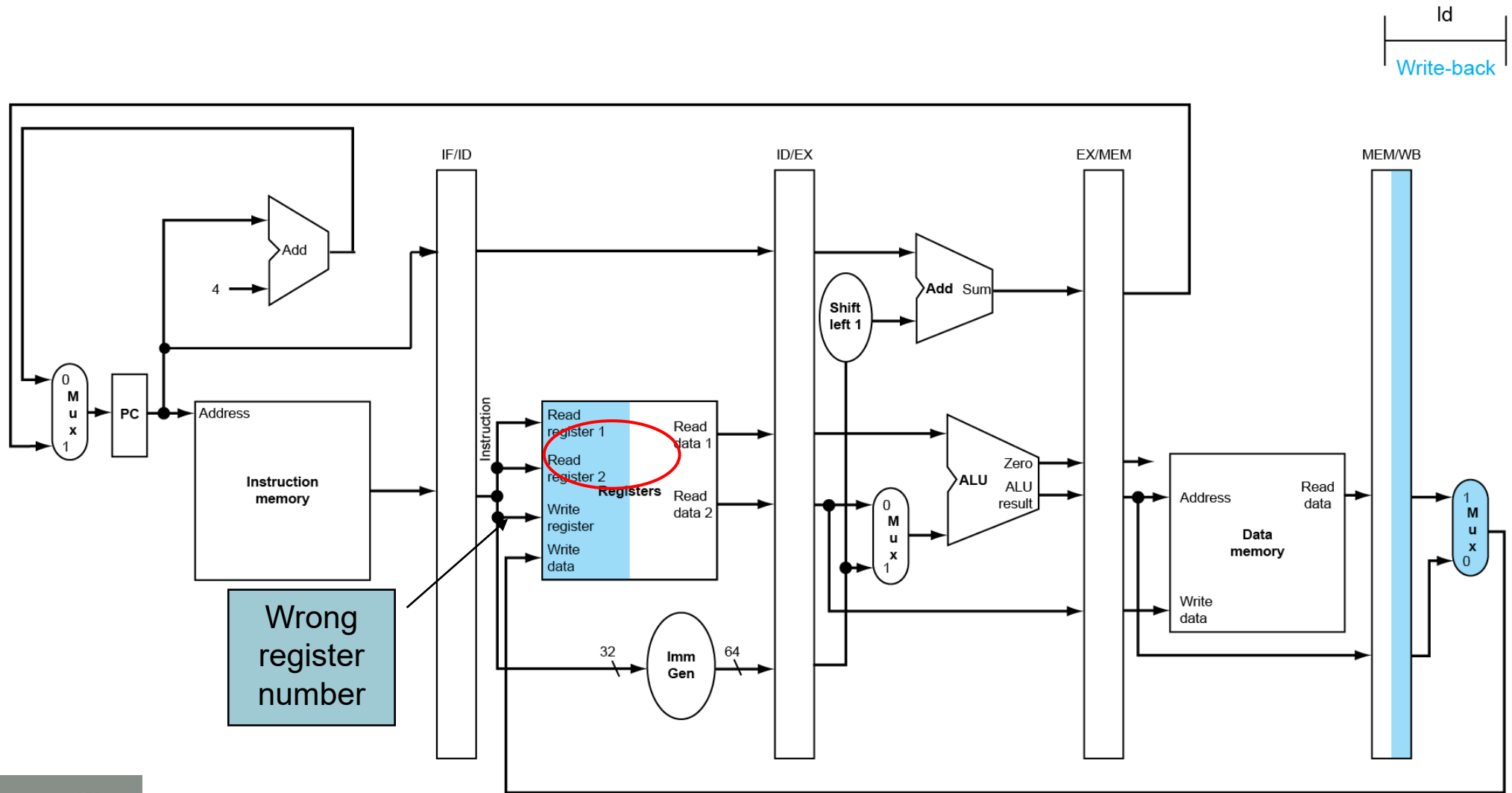
# EX for Load



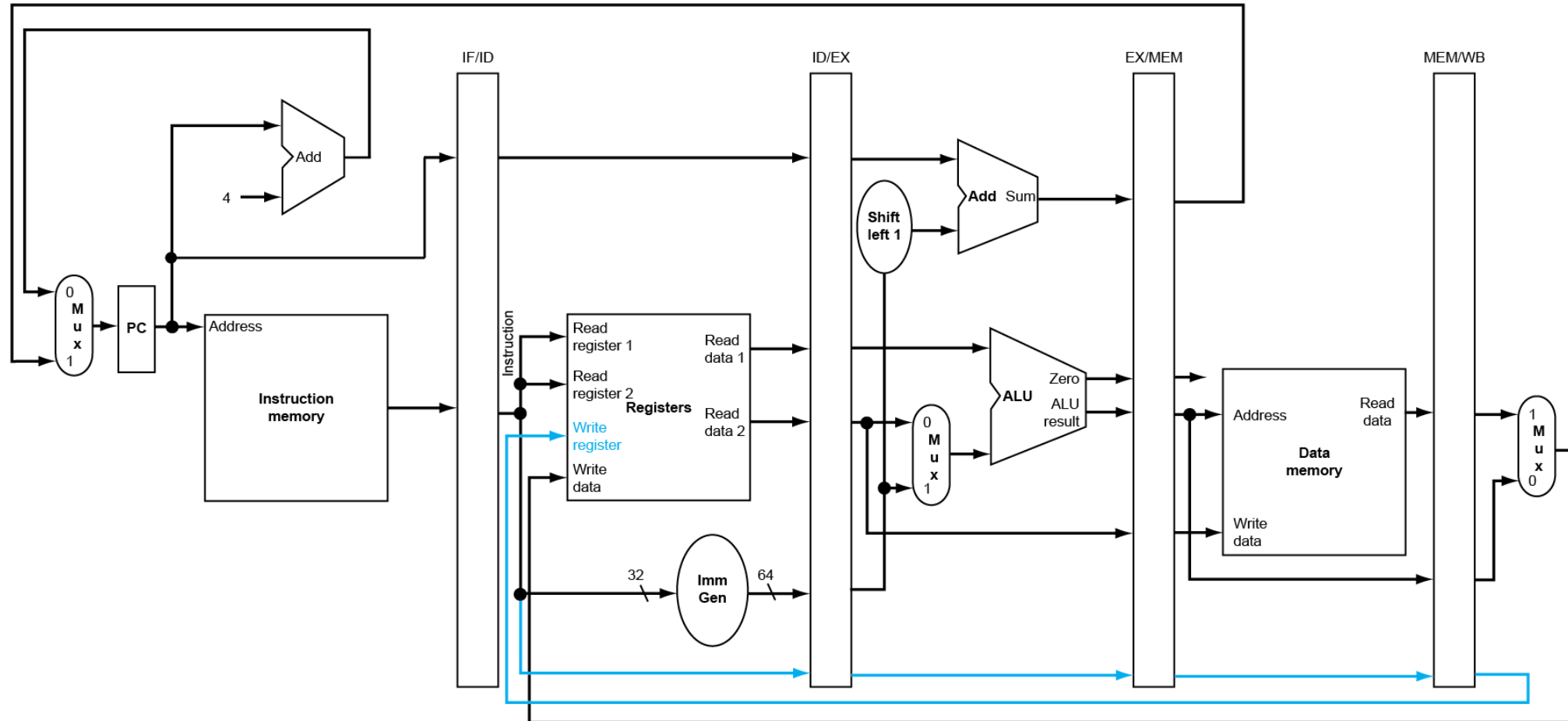
# MEM for Load



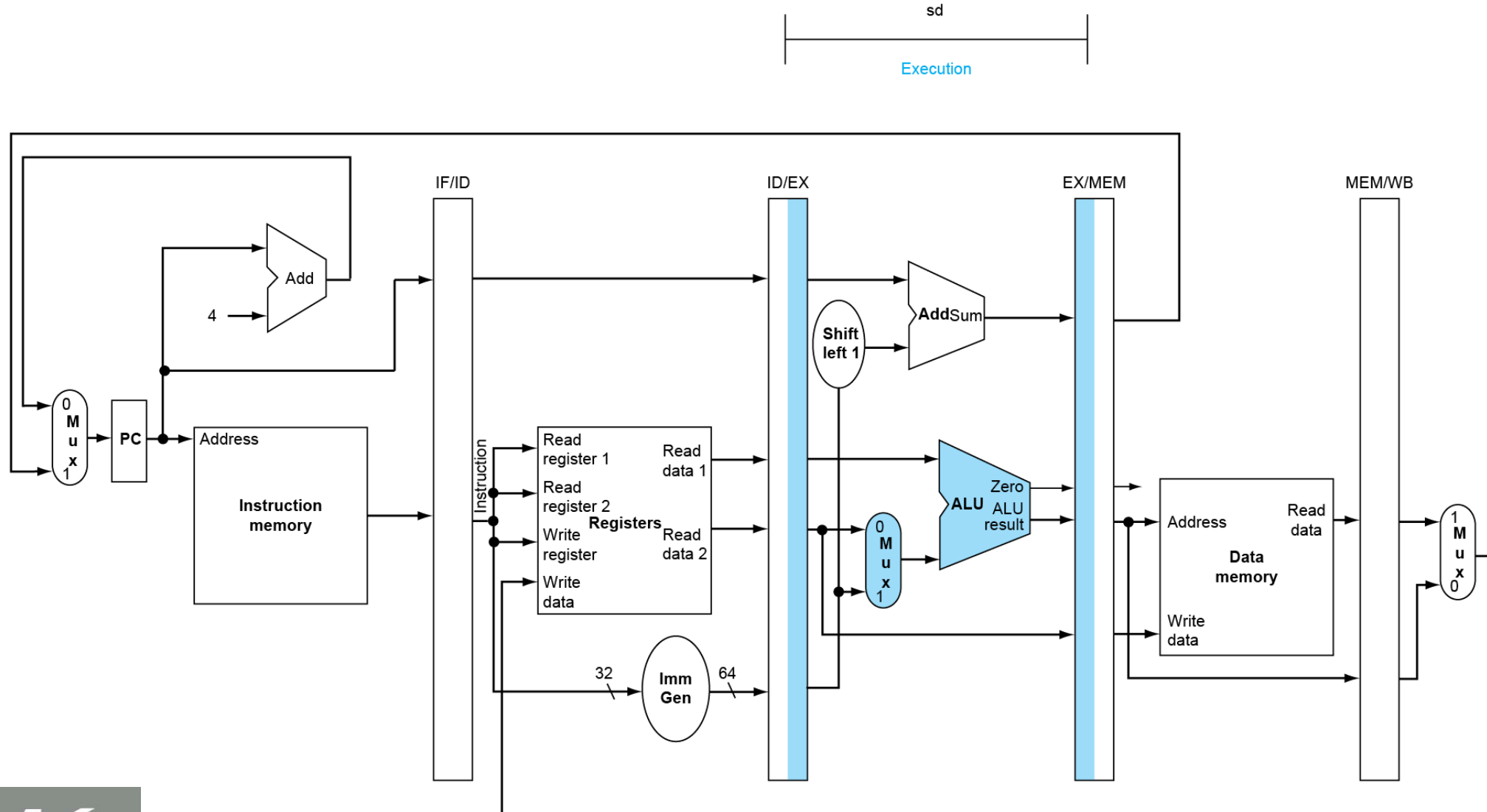
# WB for Load



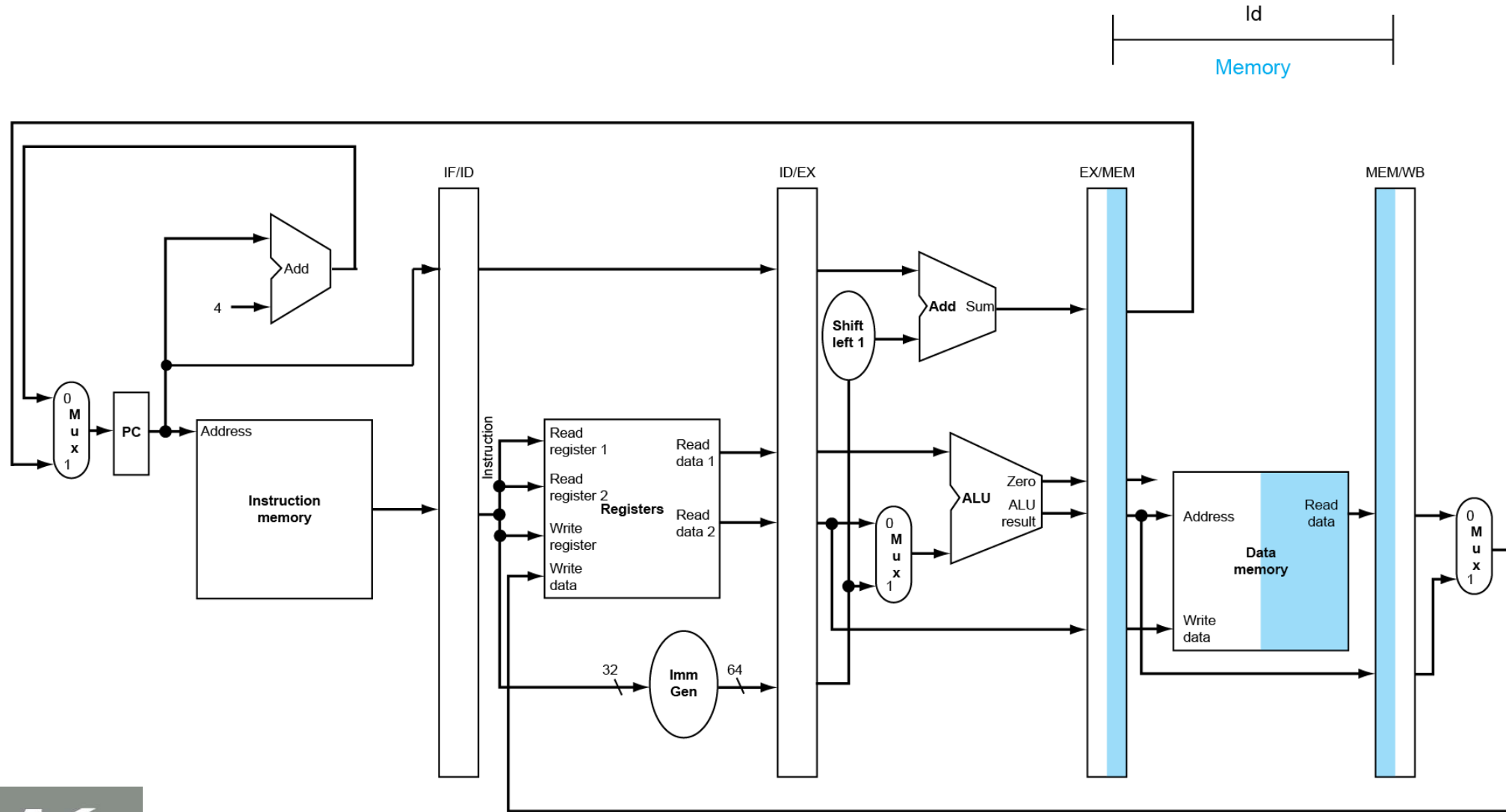
# Corrected Datapath for Load



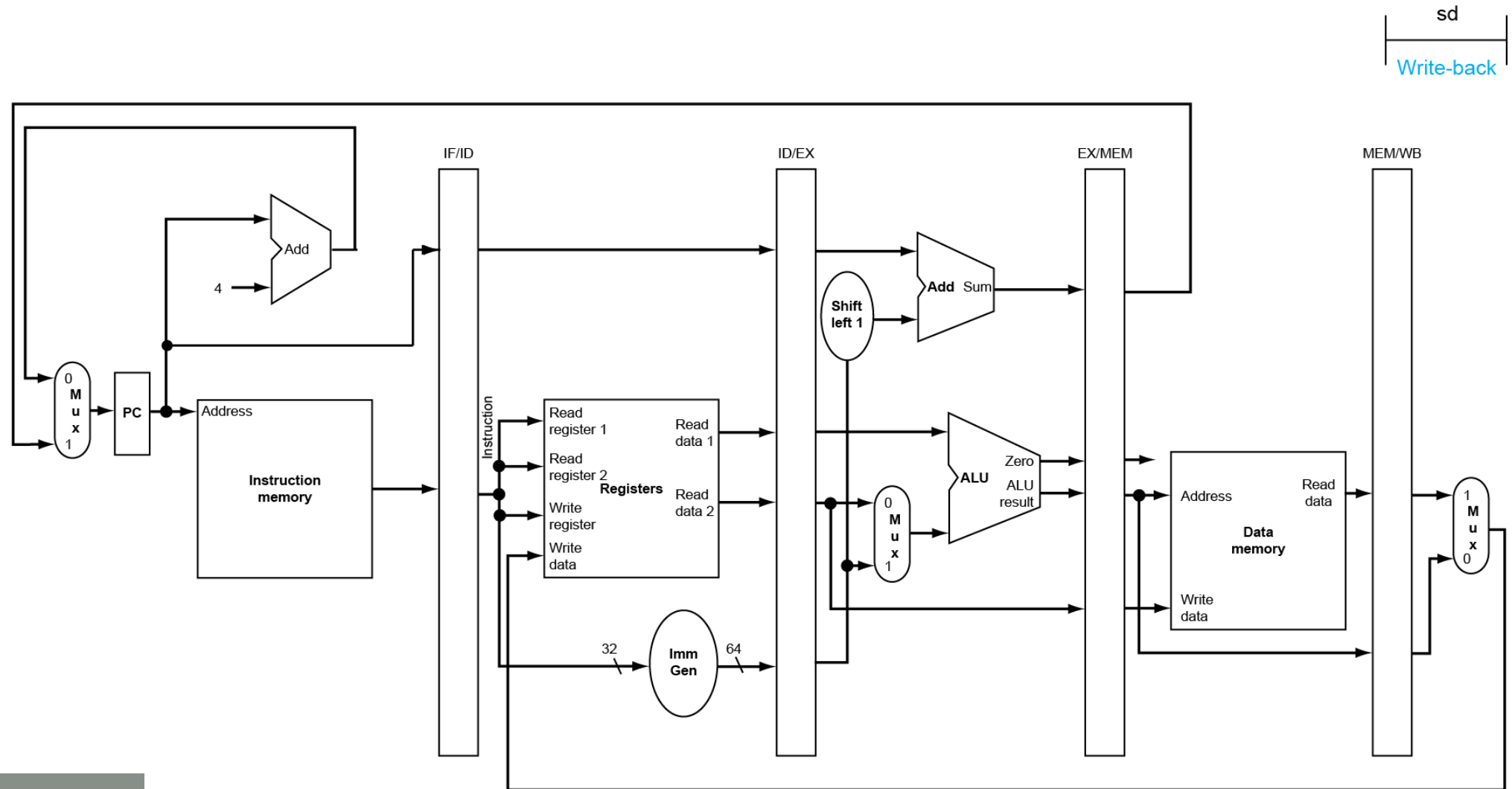
# EX for Store



# MEM for Store



# WB for Store



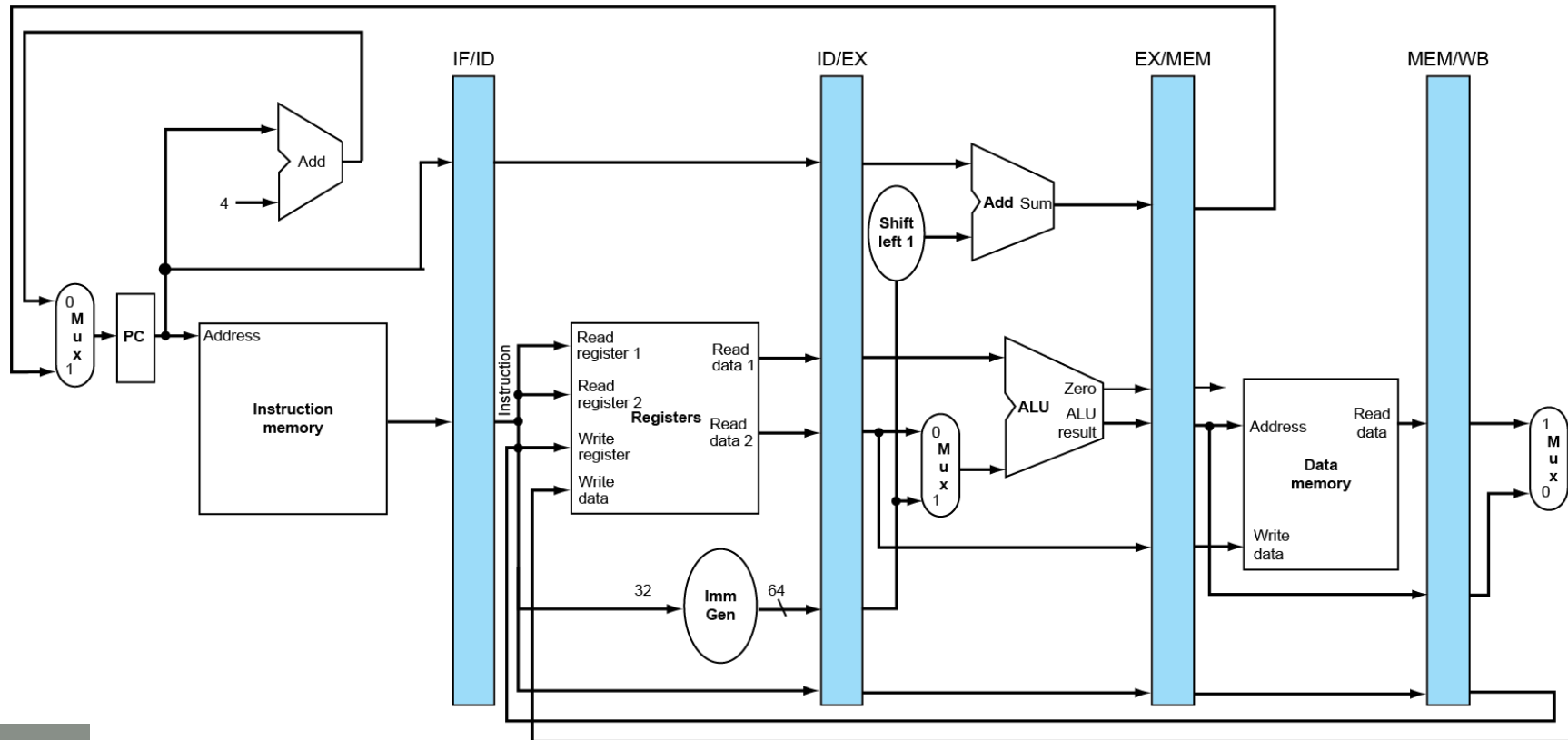
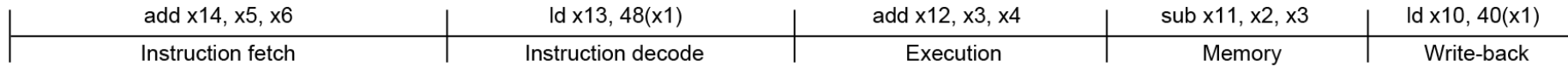


# Pipeline Diagrams

1. Single-clock-cycle pipeline diagram
  - Shows pipeline usage in a single cycle
  - Highlight resources used
2. Multi-clock-cycle diagram
  - Graph of operation over time

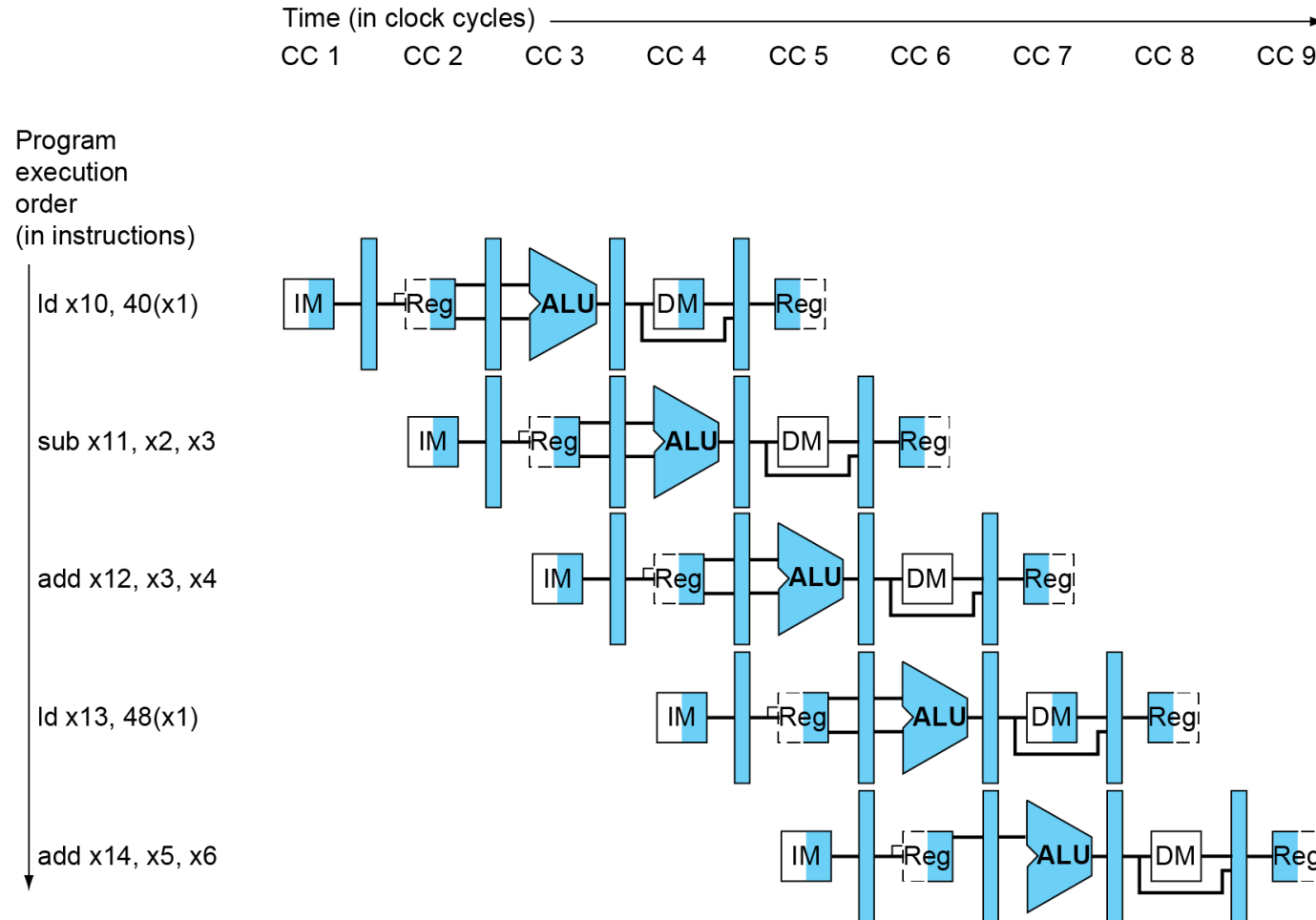
# Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle



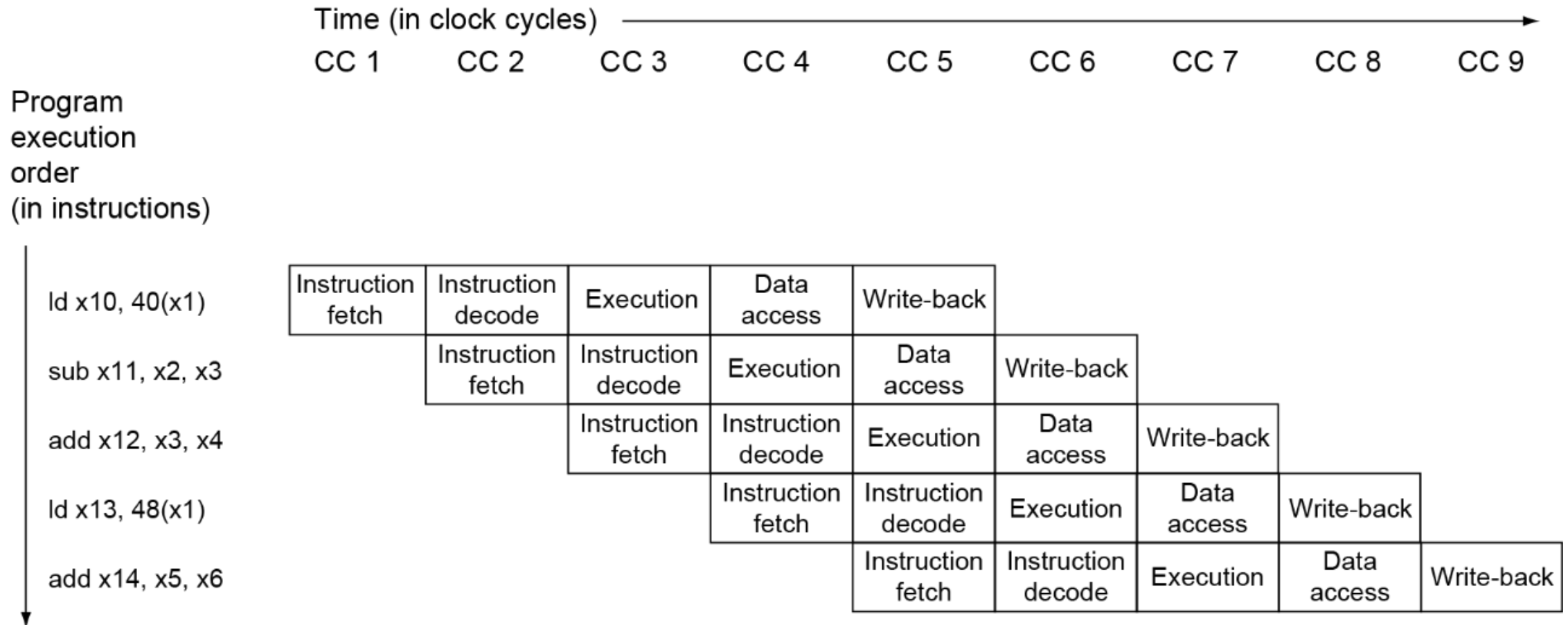
# Multi-Cycle Pipeline Diagram

- Form showing resource usage



# Multi-Cycle Pipeline Diagram

## ■ Traditional form

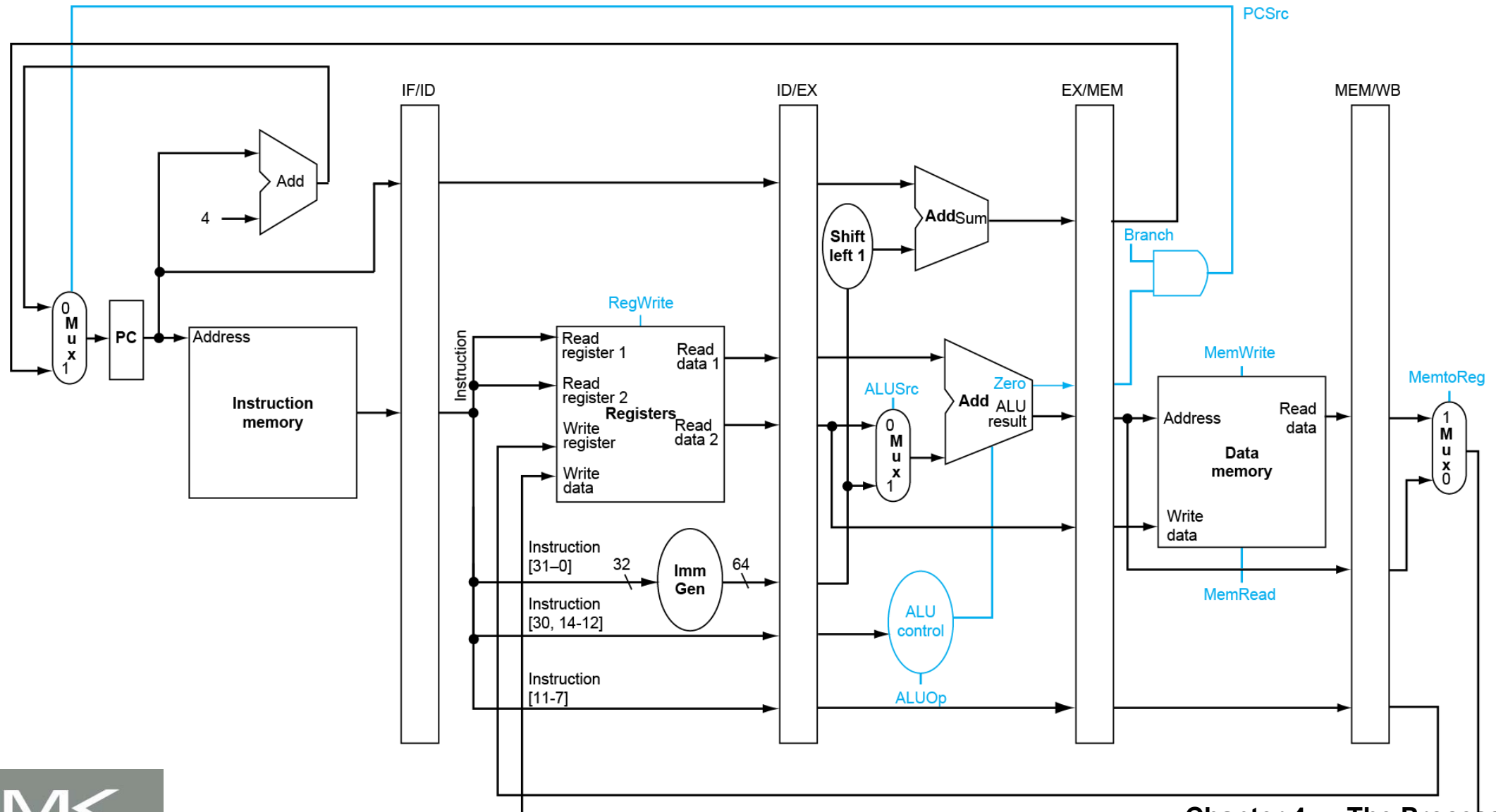


# Multi-Cycle Pipeline Diagram

- Form used in this class

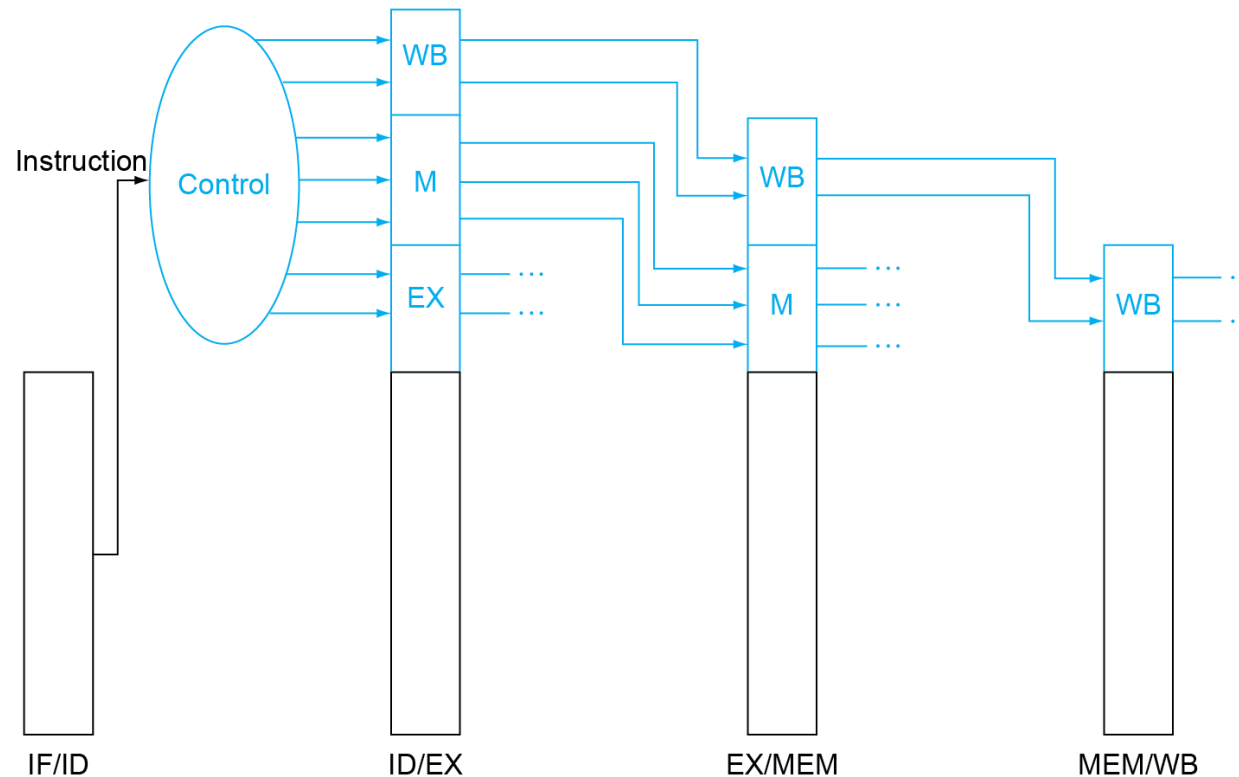
		1	2	3	4	5	6	7	8	9
ld	x10, 40(x1)	F	D	E	M	W				
sub	x11, x2, x3		F	D	E	M	W			
add	x12, x3, x4			F	D	E	M	W		
ld	x13, 48(x1)				F	D	E	M	W	
add	x14, x5, x6					F	D	E	M	W

# Pipelined Control (Simplified)

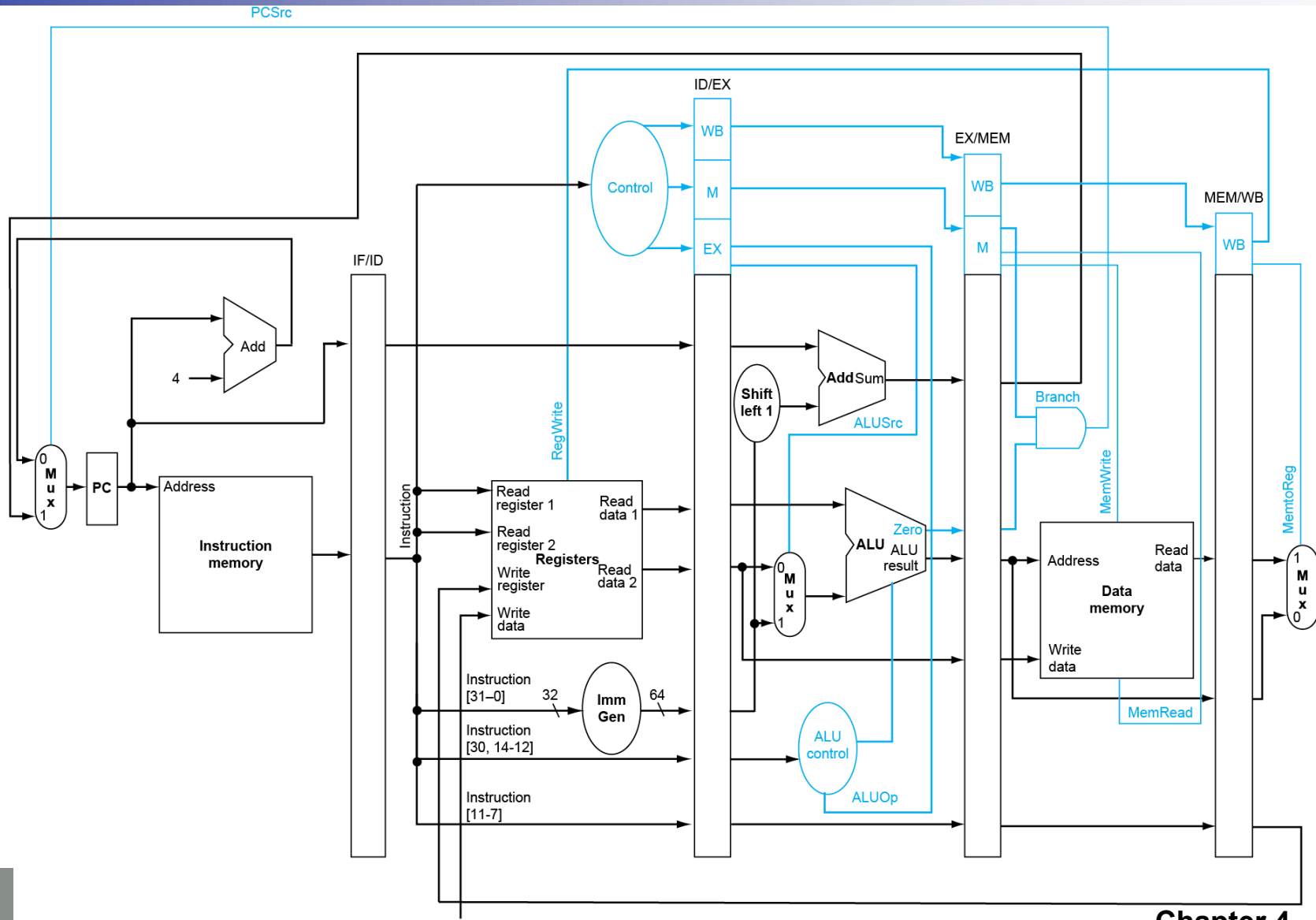


# Pipelined Control

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



# Pipelined Control





# Contents

---

4.6 An Overview of Pipelining

4.7 Pipelined Datapath and Control

4.8 Data Hazards: Forwarding versus Stalling

4.9 Control Hazards

4.10 Exceptions

4.11 Parallelism via Instructions

4.12 Putting it All Together: The Intel Core i7 6700 and ARM Cortex-A53

4.15 Fallacies and Pitfalls

4.16 Concluding Remarks

# Contents

---

## 4.8 Data Hazards: Forwarding versus Stalling

Data Hazards in ALU Instructions

Load-Use Data Hazard

Code Scheduling

# Data Hazards in ALU Instructions

- Consider this sequence:

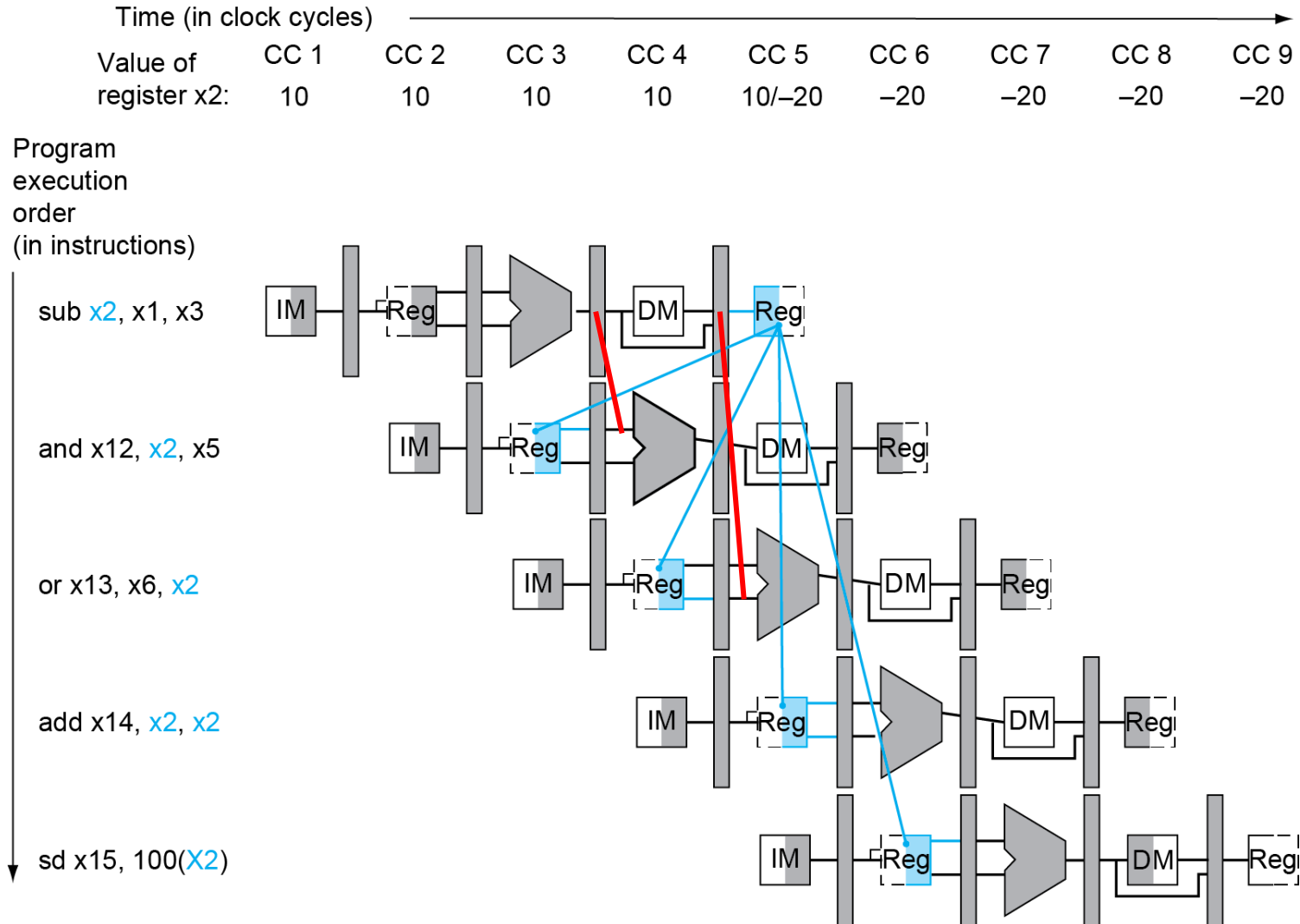
```
sub  x2, x1, x3
and  x12, x2, x5
or   x13, x6, x2
add  x14, x2, x2
sd   x15, 100(x2)
```

- There are multiple true data dependencies, read-after-write (RAW), on register x2.
- We can resolve hazards with stalls or forwarding.

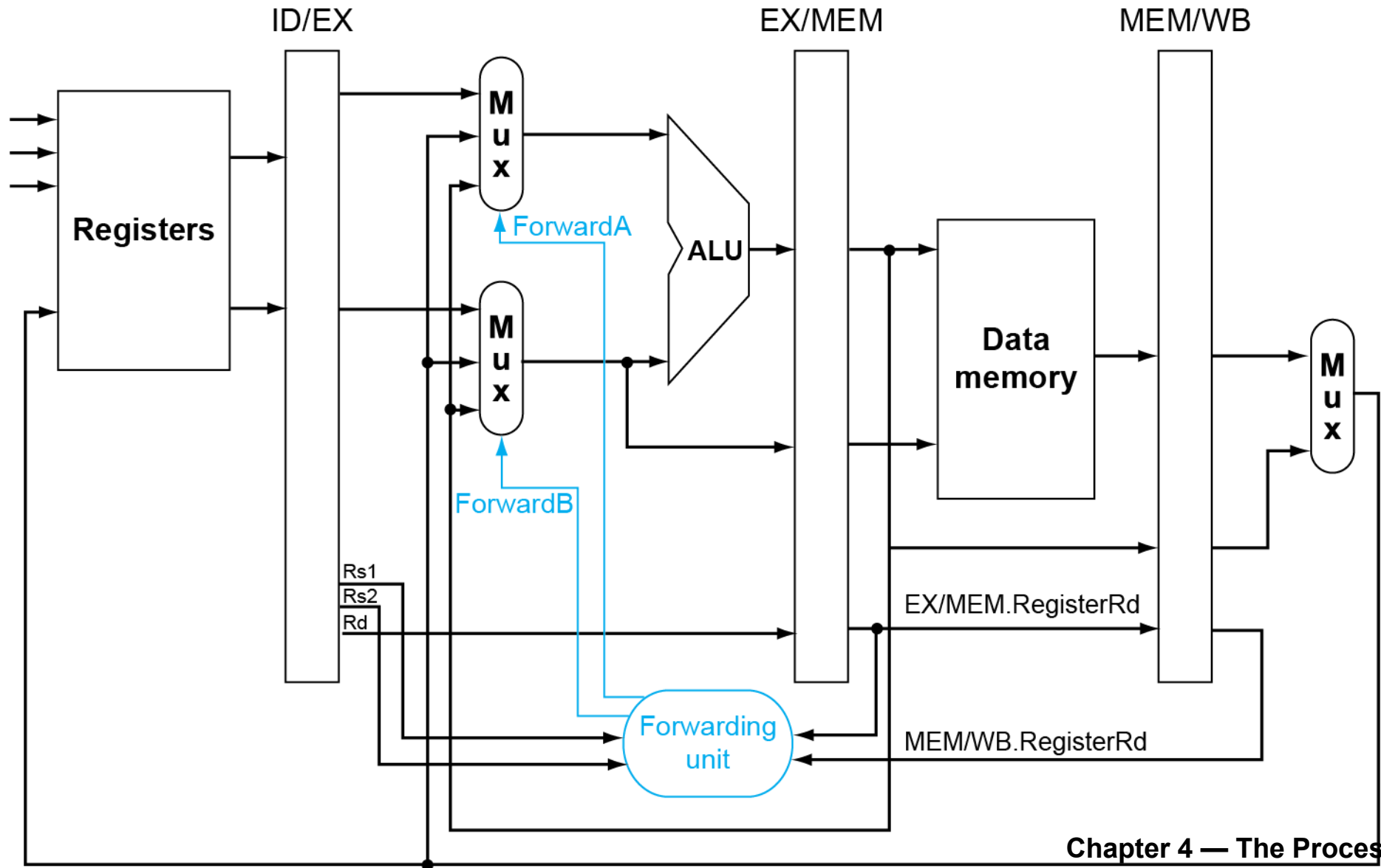
## Assume no forwarding (except through the Register File) and hazards are solved by stalls

	1	2	3	4	5	6	7	8	9	10	11
sub    x2, x1, x3	F	D	E	M	W						
and    x12, x2, x5		F									
or     x13, x6, x2											
add    x14, x2, x2											
sd     x15, 100(x2)											

# Dependencies & Forwarding



# Forwarding Paths

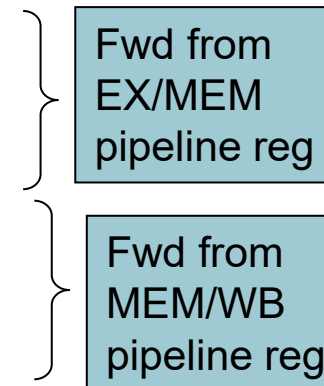


## With Forwarding

	1	2	3	4	5	6	7	8	9	10
sub x2, x1, x3	F	D	E	M	W					
and x12, x2, x5		F								
or x13, x6, x2										
add x14, x2, x2										
sd x15, 100(x2)										

# Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., E.RegisterRs1 = register number for Rs1 sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - E.RegisterRs1, E.RegisterRs2
- Data hazards when
  - 1a. M.RegisterRd = E.RegisterRs1
  - 1b. M.RegisterRd = E.RegisterRs2
  - 2a. W.RegisterRd = E.RegisterRs1
  - 2b. W.RegisterRd = E.RegisterRs2





# Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - M.RegWrite, W.RegWrite
- And only if Rd for that instruction is not x0
  - M.RegisterRd  $\neq$  0,  
W.RegisterRd  $\neq$  0

# Forwarding Conditions

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

# Condition for Forwarding from Memory

- if (M.RegWrite  
and (M.RegisterRd  $\neq$  0)  
and (M.RegisterRd = E.RegisterRs1)) ForwardA = 10
- if (M.RegWrite  
and (M.RegisterRd  $\neq$  0)  
and (M.RegisterRd = E.RegisterRs2)) ForwardB = 10

# Double Data Hazard

- Consider the sequence:

add x1, x1, x2

add x1, x1, x3

add x1, x1, x4

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

# Condition for Forwarding from Write back

- if (W.RegWrite  
and (W.RegisterRd  $\neq$  0)  
and not(M.RegWrite and (M.RegisterRd  $\neq$  0)  
and (M.RegisterRd  $\neq$  E.RegisterRs1))  
and (W.RegisterRd = E.RegisterRs1)) ForwardA = 01
- if (W.RegWrite  
and (W.RegisterRd  $\neq$  0)  
and not(M.RegWrite and (M.RegisterRd  $\neq$  0)  
and (M.RegisterRd  $\neq$  E.RegisterRs2))  
and (W.RegisterRd = E.RegisterRs2)) ForwardB = 01

# Contents

---

## 4.8 Data Hazards: Forwarding versus Stalling

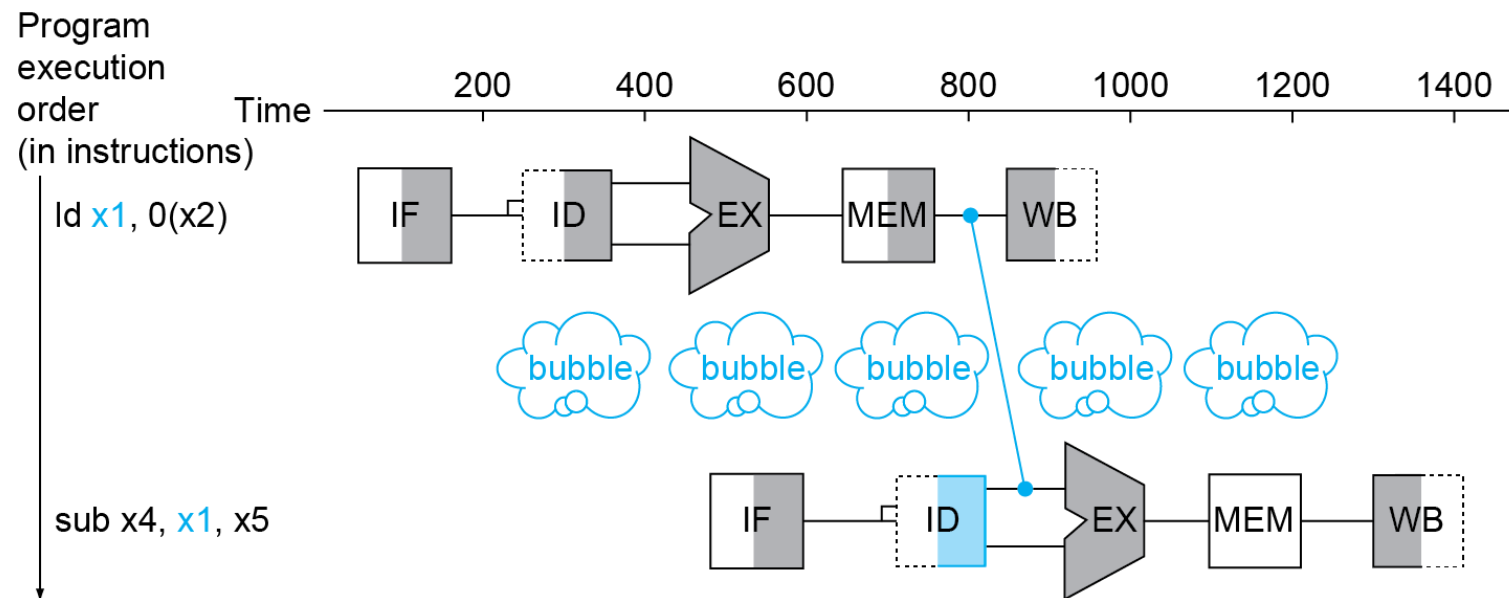
Data Hazards in ALU Instructions

Load-Use Data Hazard

Code Scheduling

# 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

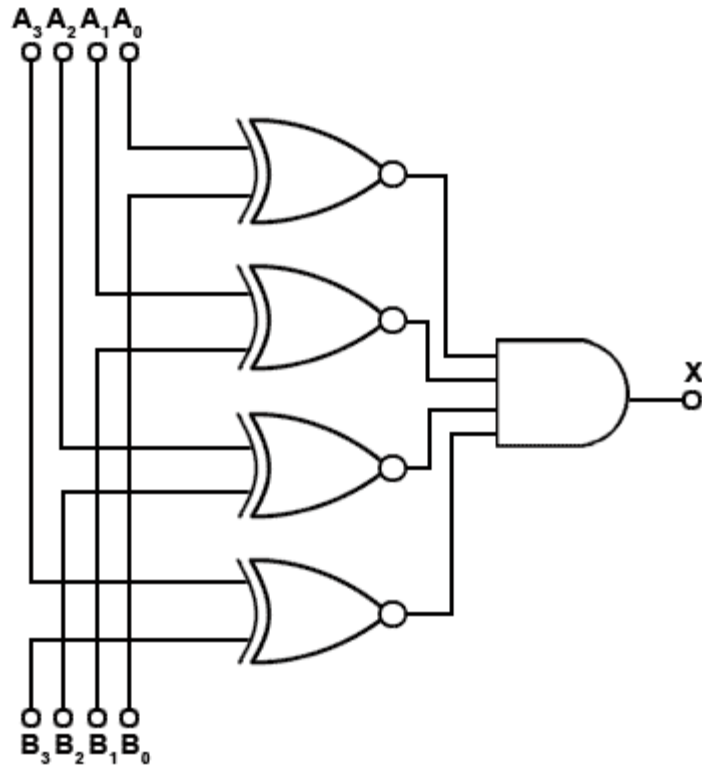
	1	2	3	4	5	6	7	8	9	10
ld x1, 0(x2)	F	D	E	M	W					
sub x4, x1, x5		F	D							



# Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - D.RegisterRs1, D.RegisterRs2
- Load-use hazard when
  - E.MemRead and
    - ((E.RegisterRd = D.RegisterRs1) or (E.RegisterRd = D.RegisterRs2))
- If detected, stall and insert bubble

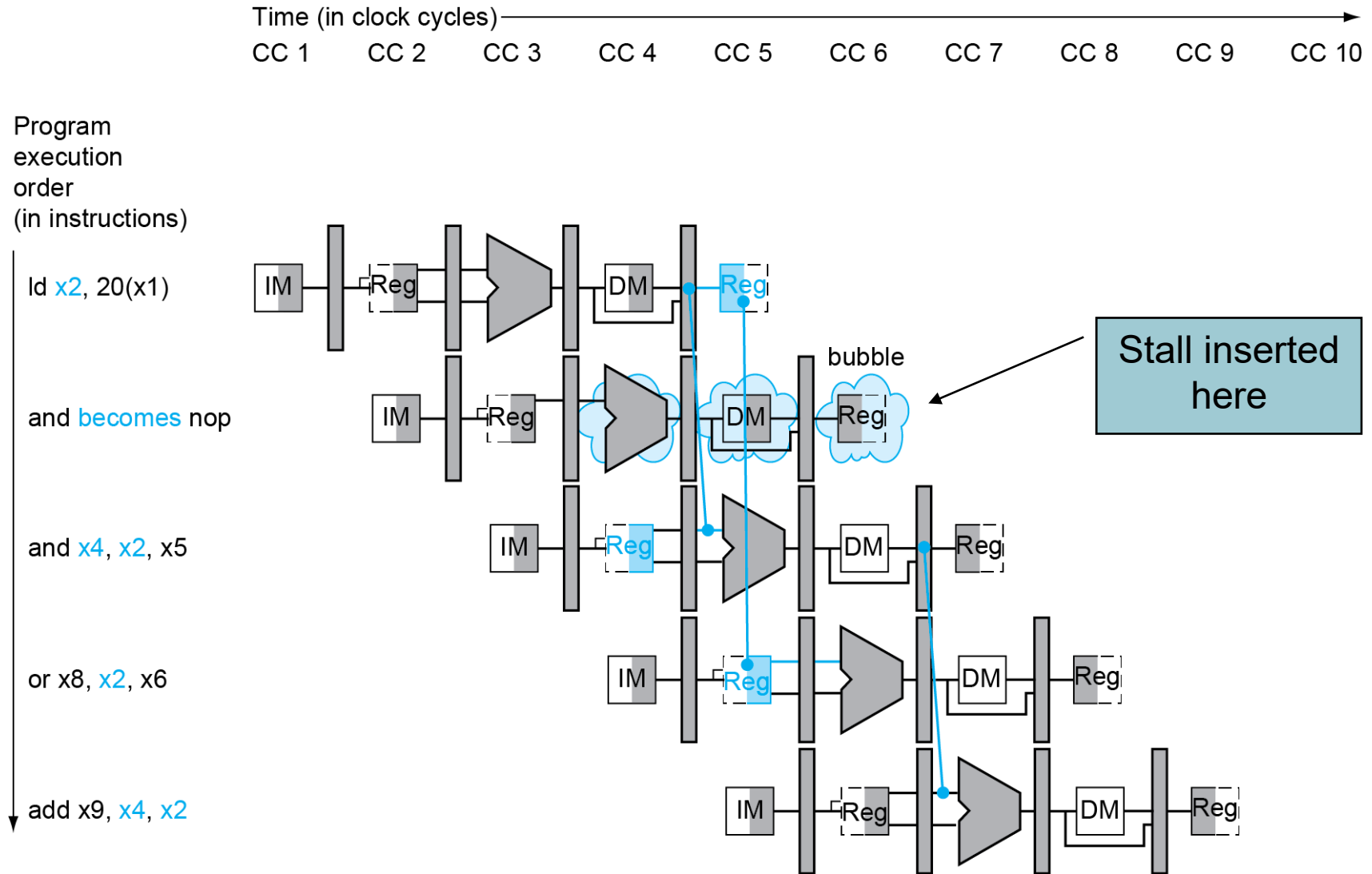
# Stall Circuit



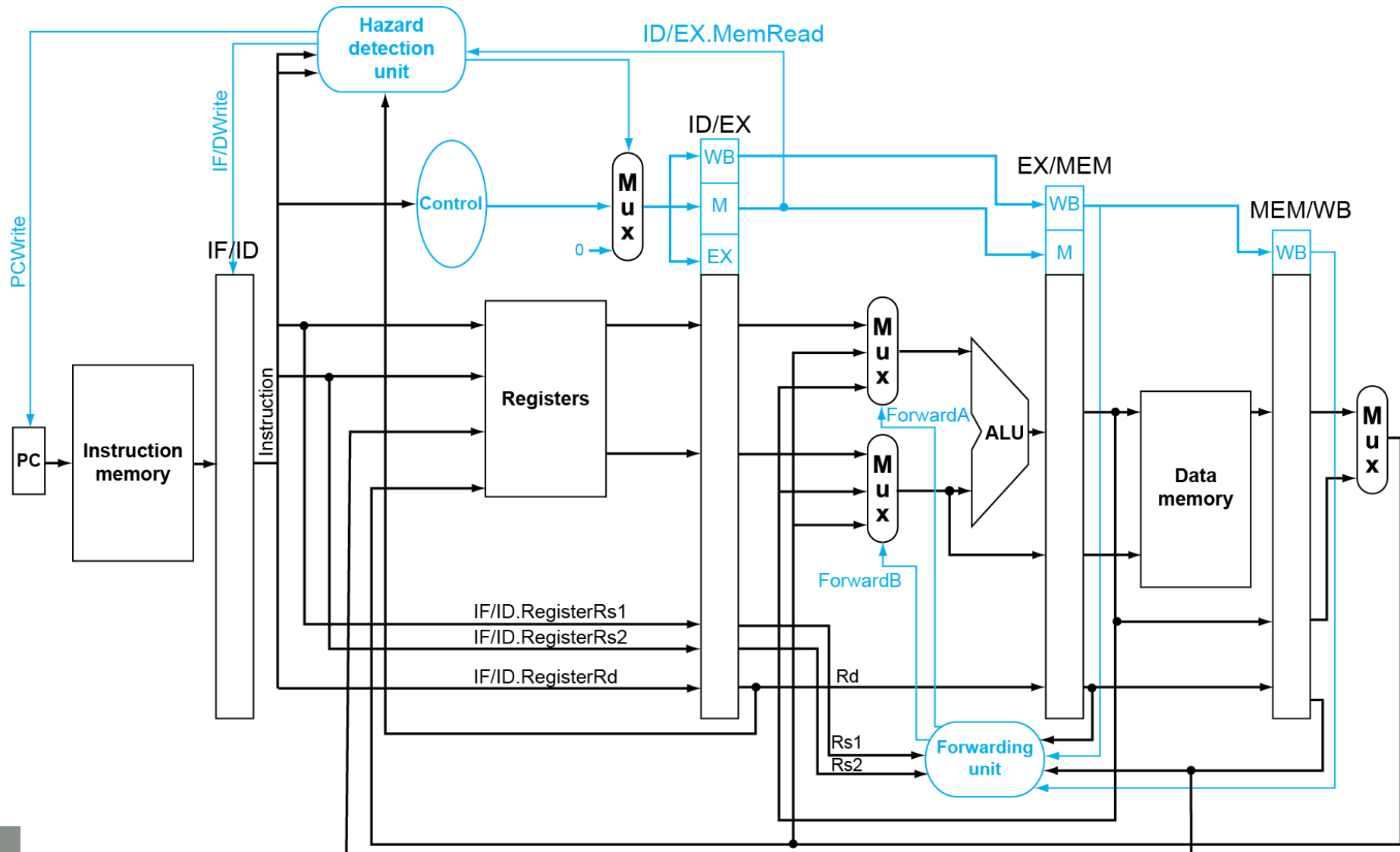
# How to Stall the Pipeline

- 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 1 d
    - Can subsequently forward to EX stage

# Load-Use Data Hazard



# 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

# Rearranging to solve Load-Use Data Hazard

	1	2	3	4	5	6	7	8	9	10
ld x1, 0(x2)	F	D	E	M	W					
sub x4, x1, x5		F	D	D	E	M	W			
add x7, x5, x6			F	F	D	E	M	W		

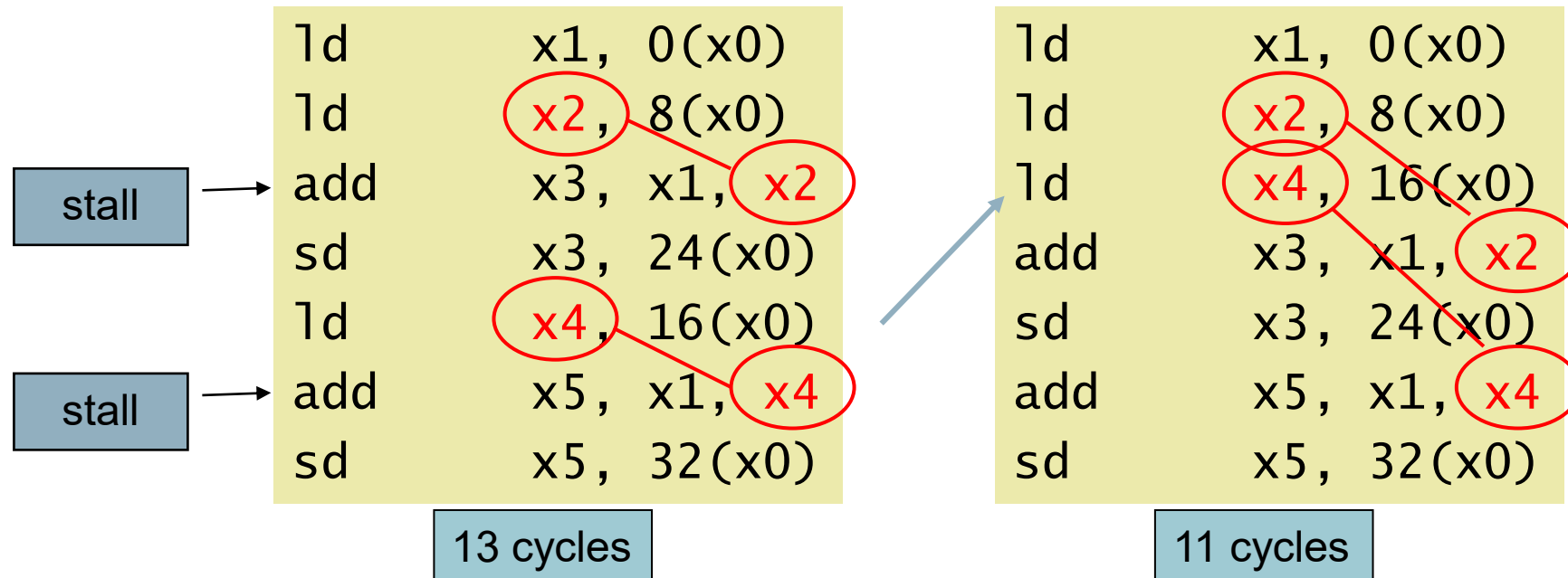
# Rearranging to solve Load-Use Data Hazard

	1	2	3	4	5	6	7	8	9	10
ld x1, 0(x2)	F	D	E	M	W					
add x7, x5, x6		F	D							
sub x4, x1, x5										



# Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for  $a = b + e; c = b + f;$



# Contents

---

4.6 An Overview of Pipelining

4.7 Pipelined Datapath and Control (Review)

4.8 Data Hazards: Forwarding versus Stalling

4.9 Control Hazards

4.10 Exceptions

4.11 Parallelism via Instructions

4.12 Putting it All Together: The Intel Core i7 6700 and ARM Cortex-A53

4.15 Fallacies and Pitfalls

4.16 Concluding Remarks

# Contents

---

## 4.9 Control Hazards

- Branch Hazards

- Reducing Branch Delay

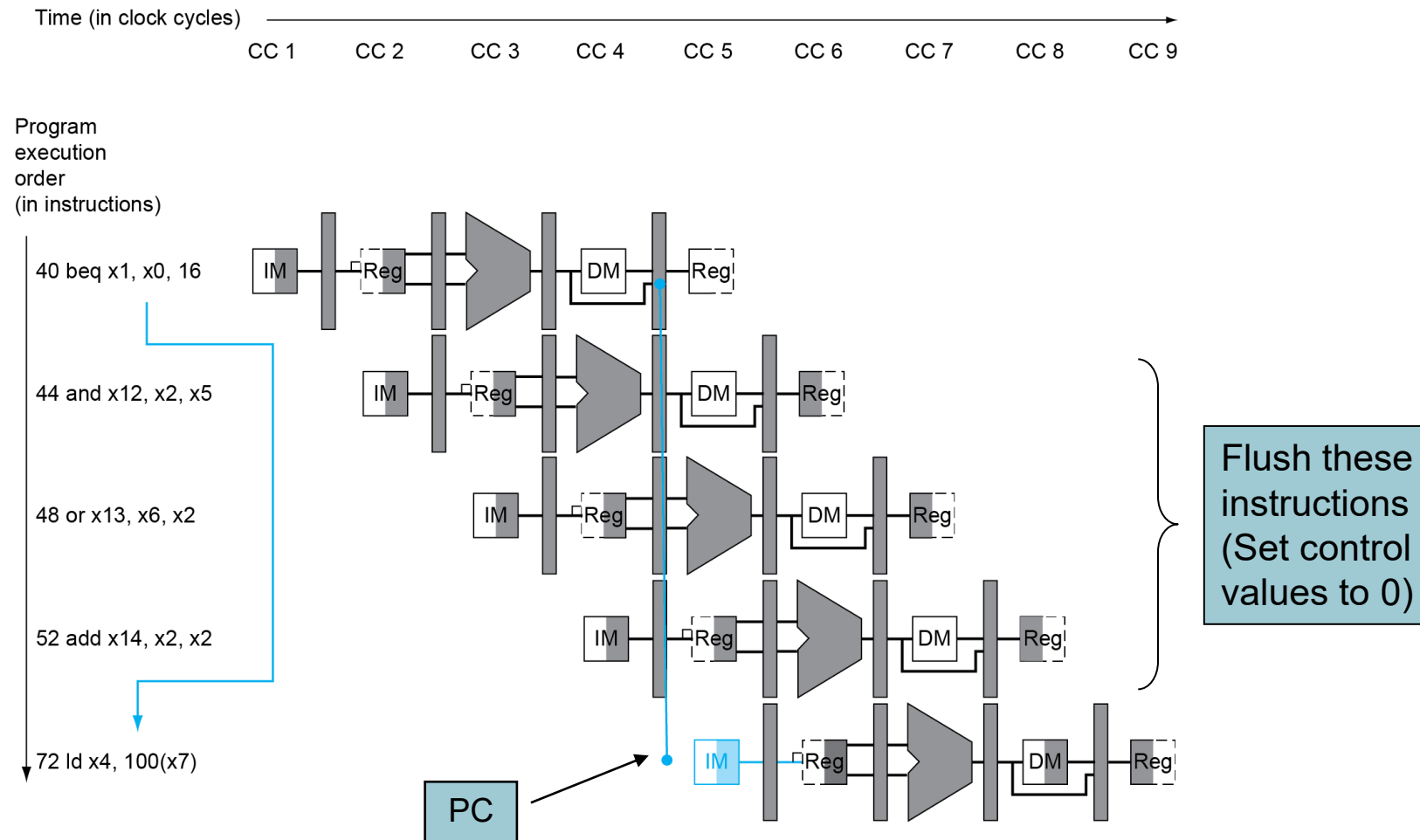
- Branch Prediction

- Dynamic Branch Prediction

- Calculating Branch Target

# Branch Hazards

- If branch outcome determined in MEM



# Solving branches in the **Memory** stage

Assume taken branch

		1	2	3	4	5	6	7	8	9	10
40	beq x1, x0, 16	F	D	E	M	W					
44	and x12, x2, x5		F								
48	or x13, x6, x2										
52	add x14, x2, x2										
72	ld x4, 100(x7)										

# Reducing Branch Delay

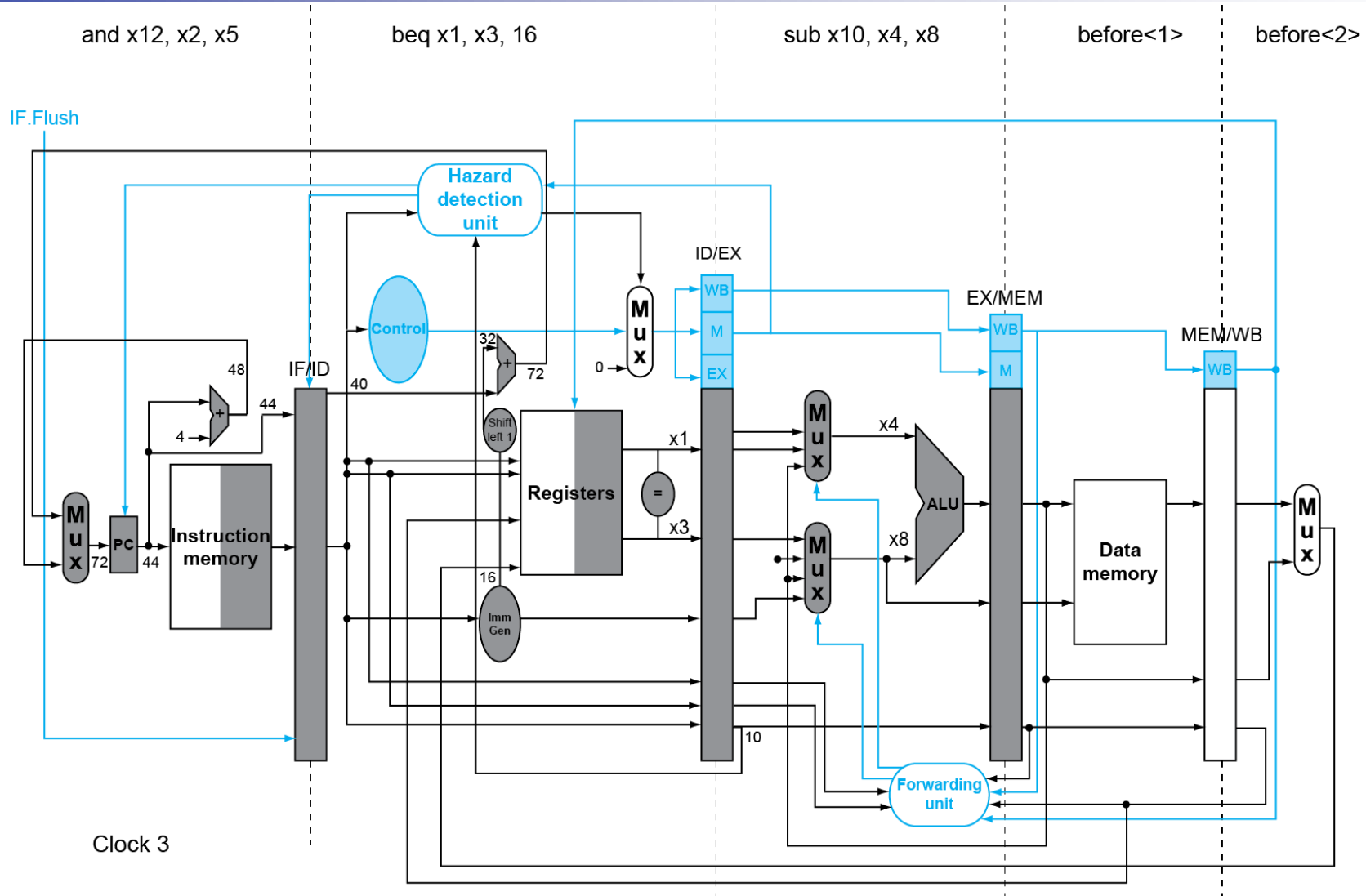
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:  sub  x10, x4, x8
40:  beq  x1,  x3, 16 // PC-relative branch
                        // to 40+16*2=72

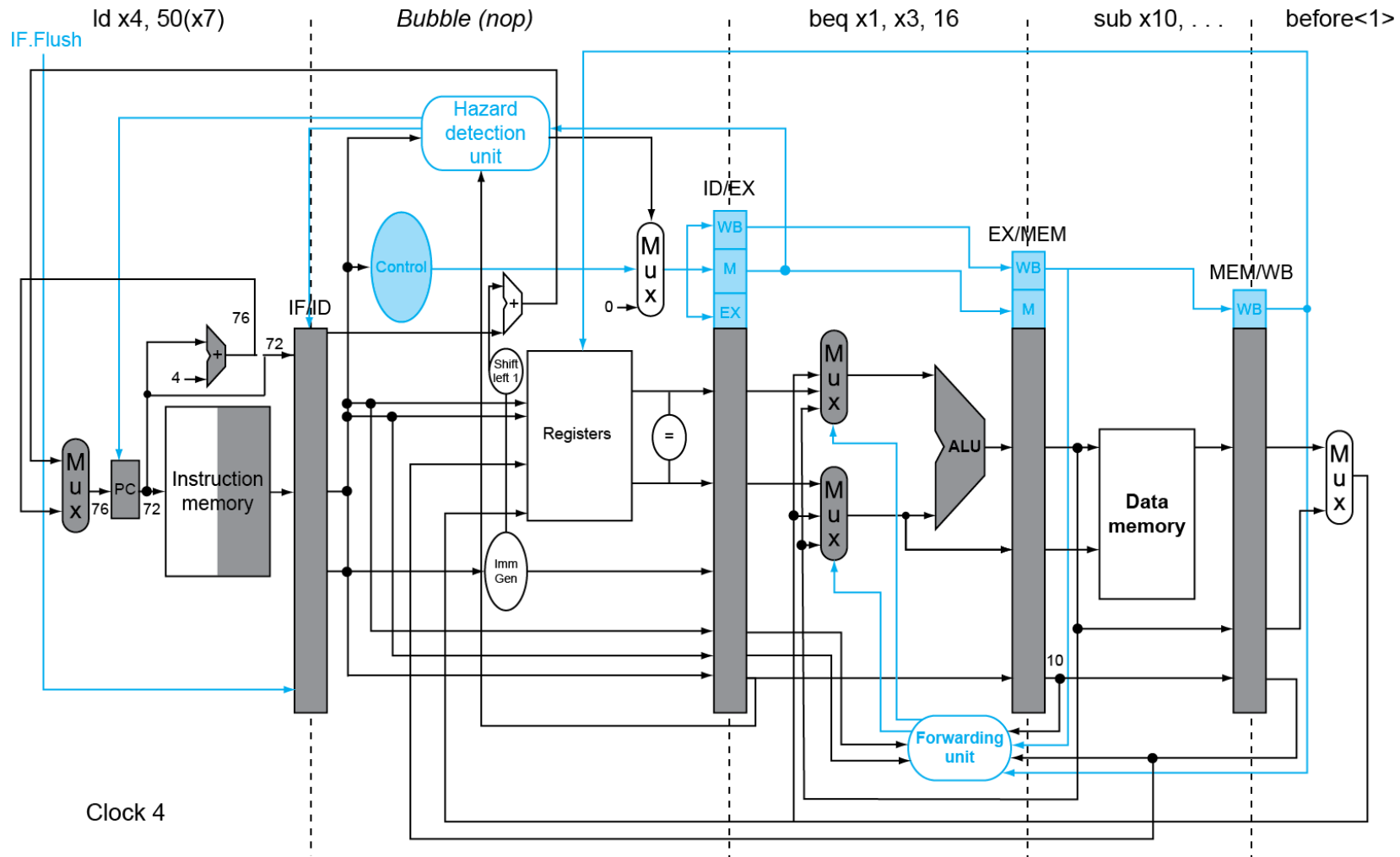
44:  and  x12, x2, x5
48:  or   x13, x2, x6
52:  add  x14, x4, x2
56:  sub  x15, x6, x7

    ...
72:  ld   x4, 50(x7)
```

# Example: Branch Taken



# Example: Branch Taken





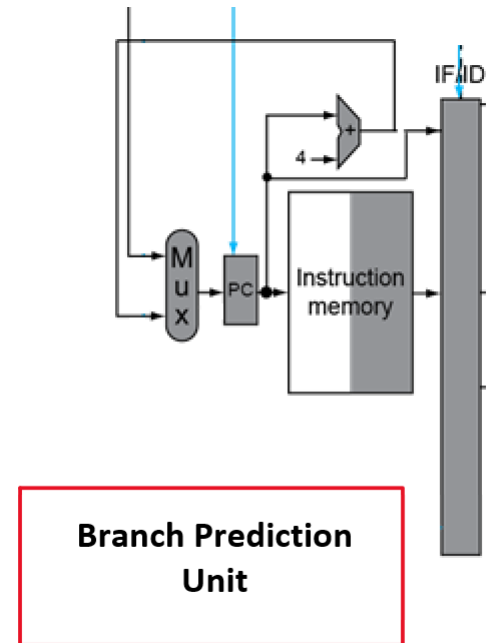
# Solving branches in the **Decode** stage

Assume taken branch

		1	2	3	4	5	6	7	8	9	10
40	beq x1, x0, 16	F	D	E	M	W					
44	and x12, x2, x5		F								
48	or x13, x6, x2										
52	add x14, x2, x2										
72	ld x4, 100(x7)										

# 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 RISC-V pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay



# Predict Not Taken

- Solving branches in the **Decode** stage
- Assume branch is **not taken**.

		1	2	3	4	5	6	7	8	9	10
	beq x1, x0, L	F	D	E	M	W					
	I <sub>2</sub>		F								
L	I <sub>T</sub>										

# Predict Not Taken

- Solving branches in the **Decode** stage
- Assume branch is **taken**.

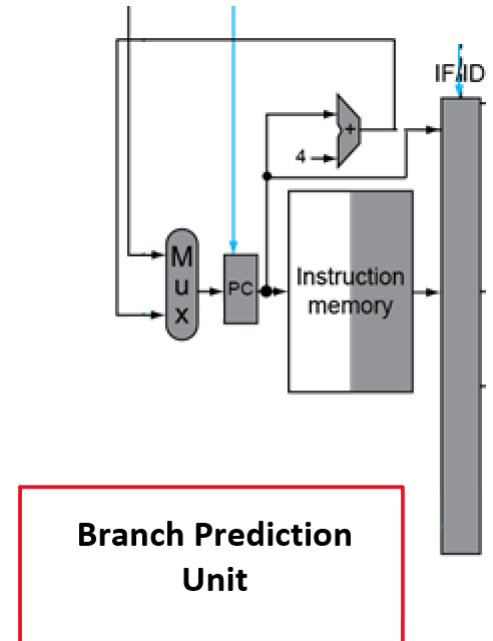
		1	2	3	4	5	6	7	8	9	10
	beq x1, x0, L	F	D	E	M	W					
	I <sub>2</sub>		F								
L	I <sub>T</sub>										

# 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

# 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



# Branch History Table (BHT)

## One-Level Branch Predictor

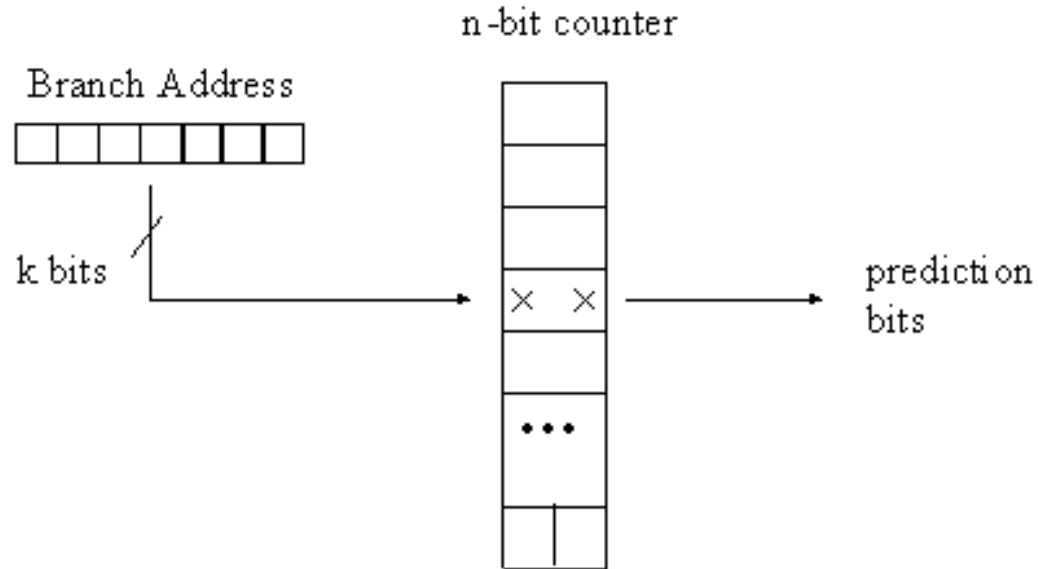
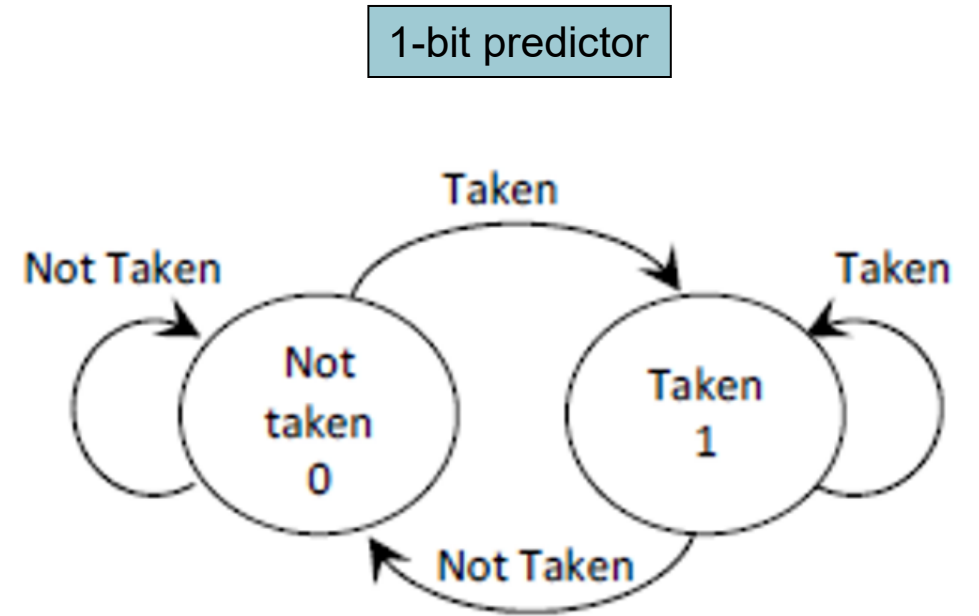
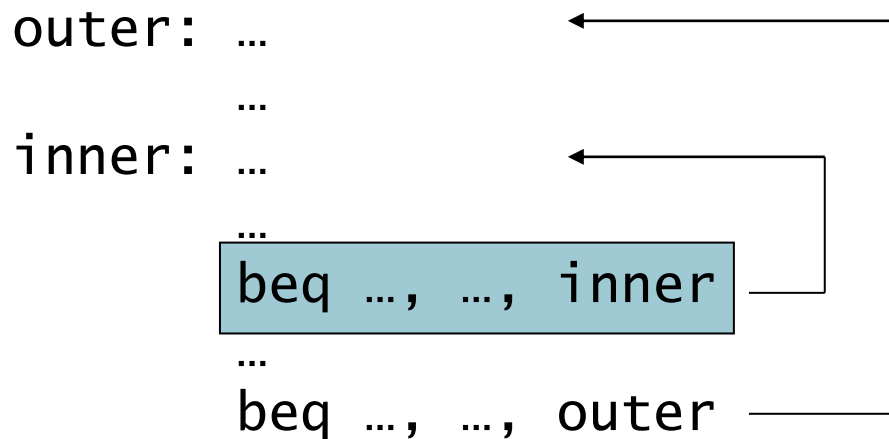


Table size =  $n \times 2^k$  bits



# 1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!



Iteration	997	998	999	0	1
Prediction	T	T	T	NT	T
Result	C	C	I	I	C

**T: Taken**

**NT: Not Taken**

**C: Correct prediction**

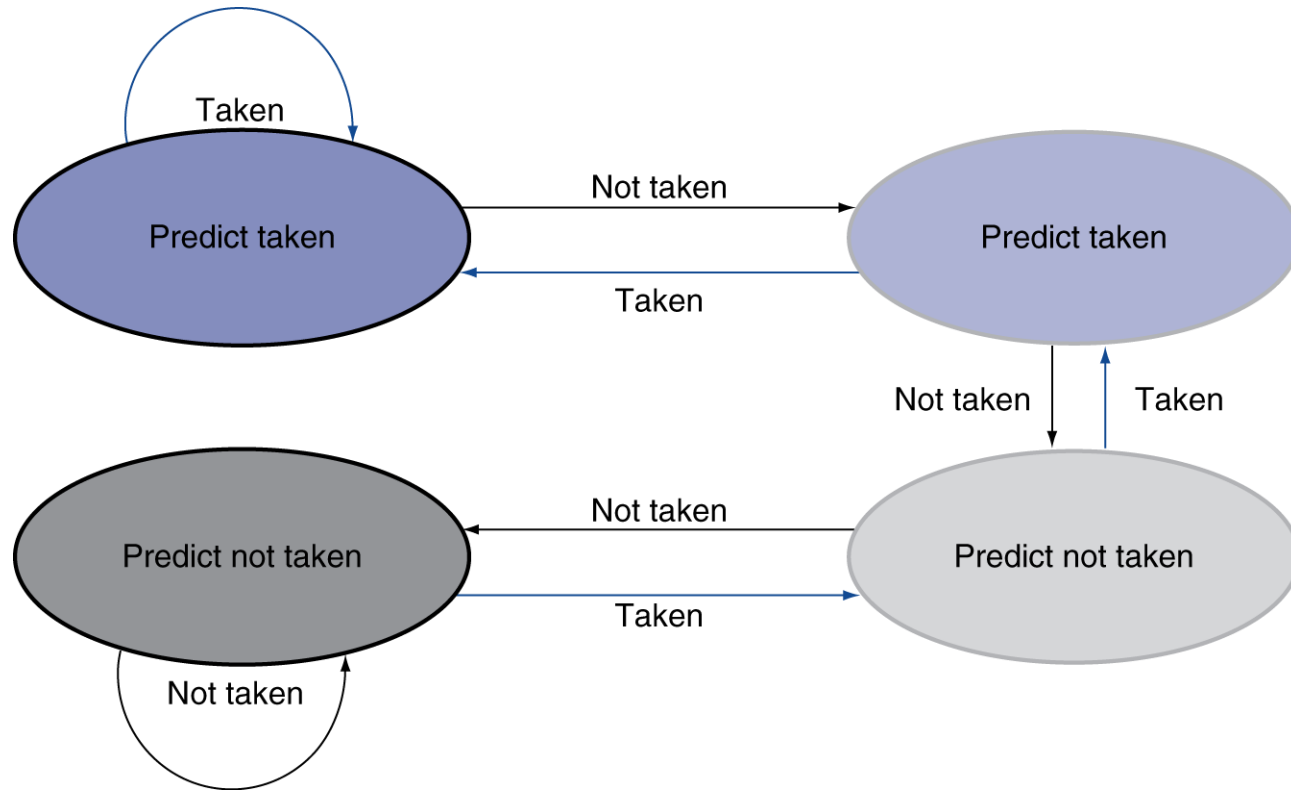
**I: Incorrect prediction**

- 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



Iteration	997	998	999	0	1
Prediction	T	T	T	T	T
Result	C	C	I	C	C

**T: Taken**

**NT: Not Taken**

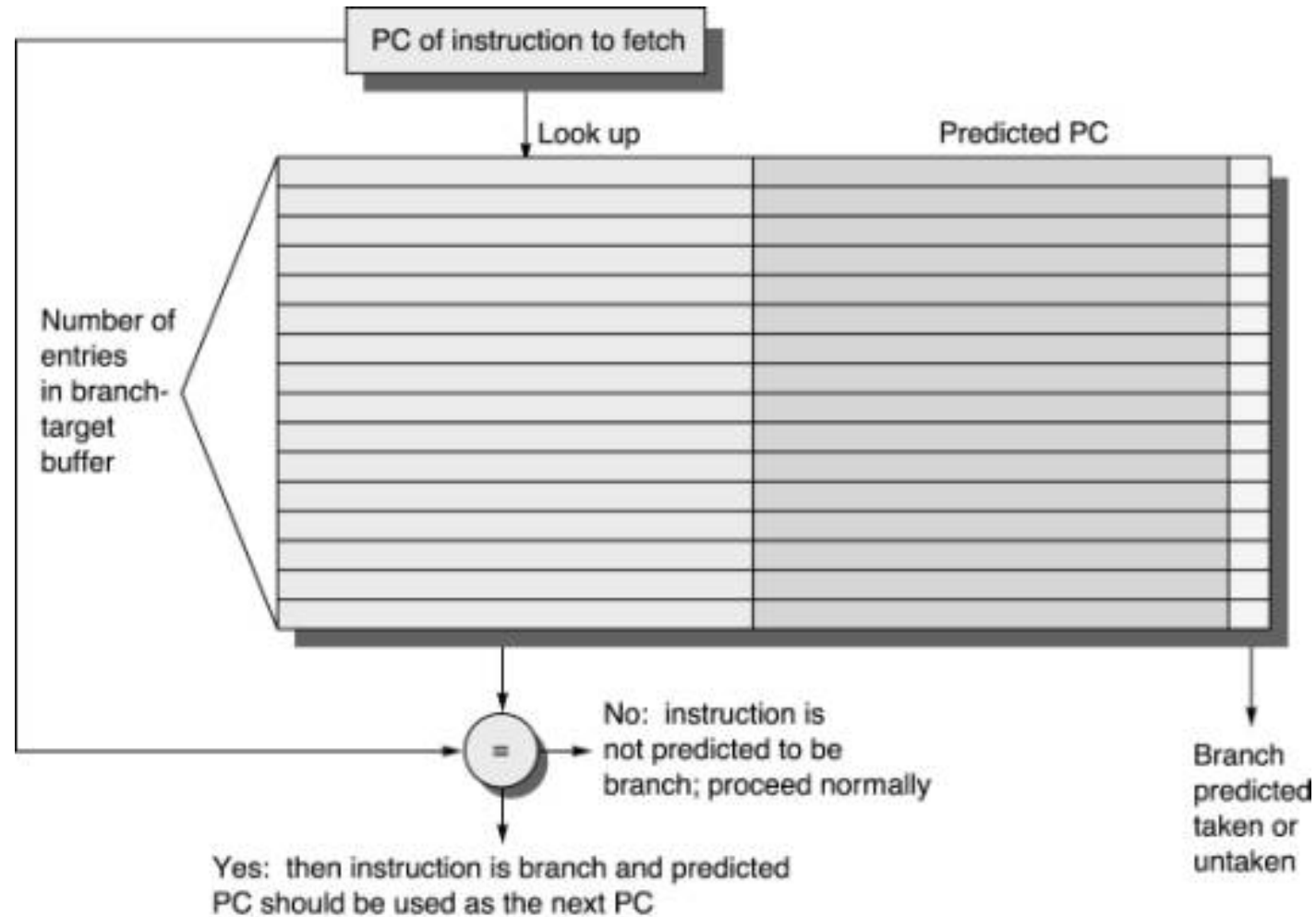
**C: Correct prediction**

**I: Incorrect prediction**

# 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

# Branch Target Buffer (BTB)



# Contents

---

4.6 An Overview of Pipelining

4.7 Pipelined Datapath and Control (Review)

4.8 Data Hazards: Forwarding versus Stalling

4.9 Control Hazards

4.10 Exceptions

4.11 Parallelism via Instructions

4.12 Putting it All Together: The Intel Core i7 6700 and ARM Cortex-A53

4.15 Fallacies and Pitfalls

4.16 Concluding Remarks

# Contents

---

## 4.10 Exceptions

Exceptions and Interrupts

Handling Exceptions

Exceptions in a Pipeline

Exception Example

Multiple Exceptions

Imprecise Exceptions

# 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, syscall, ...
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

# Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In RISC-V: Supervisor Exception Program Counter (SEPC)
- Save indication of the problem
  - In RISC-V: Supervisor Exception Cause Register (SCAUSE)
  - 64 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, ...
- Jump to handler
  - Assume at `0000 0000 1C09 0000hex`

# Handling Exceptions

I1  
I2  
I3  
I4  
I5

0000 0000 1C09 0000<sub>hex</sub>

**Exception  
Handling  
Routine**

**SEPC, SCAUSE**



# An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Exception vector address to be added to a vector table base register:
  - Undefined opcode      00 0100 0000<sub>two</sub>
  - Hardware malfunction: 01 1000 0000<sub>two</sub>
  - ...:                      ...
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

# An Alternate Mechanism

I1  
I2  
I3  
I4  
I5

Undefined opcode	00 0100 0000 <sub>two</sub>
Hardware malfunction:	01 1000 0000 <sub>two</sub>
...:	...

**SEPC, SCAUSE**

# Handler Actions

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

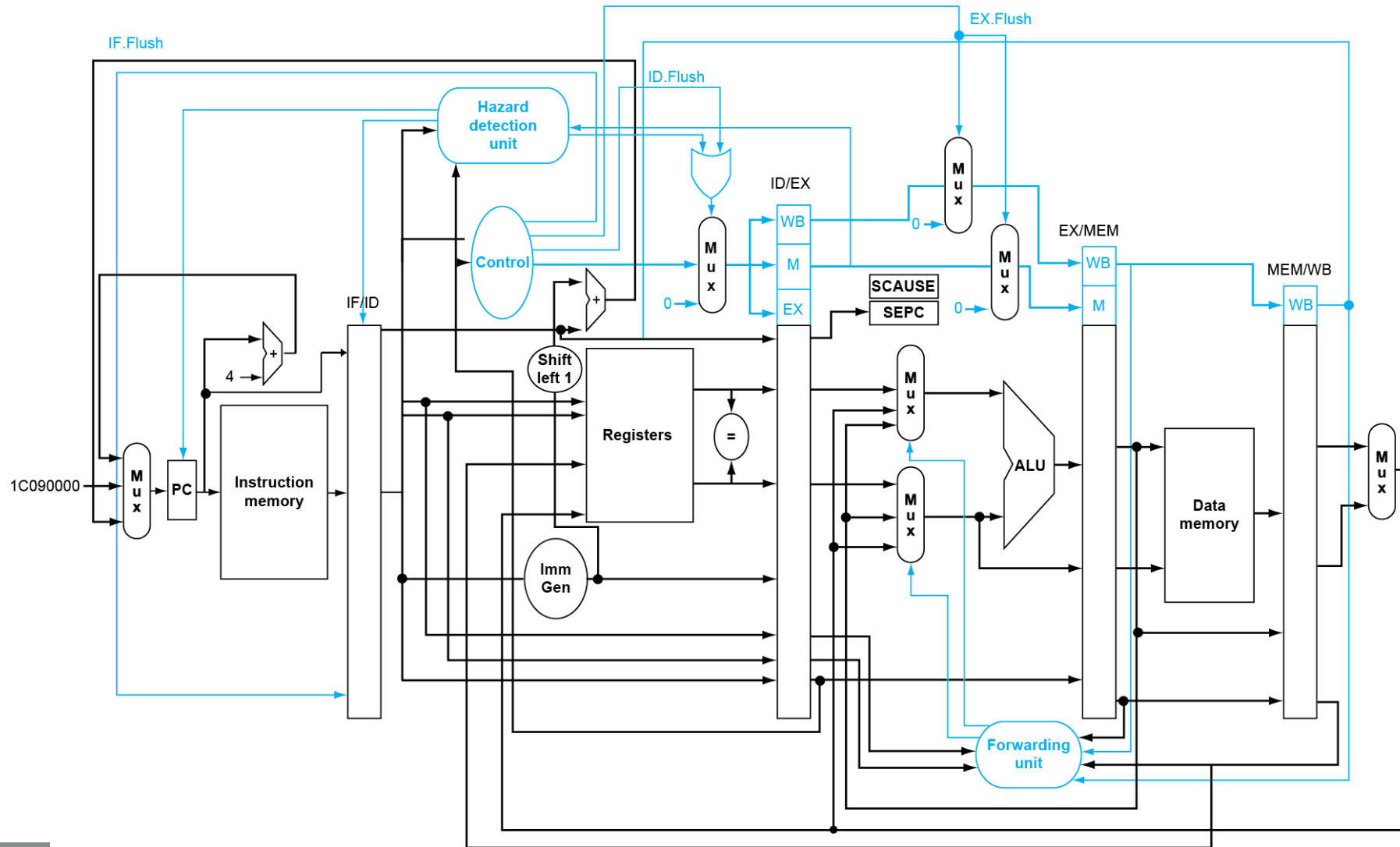
# Exceptions in a Pipeline

- Another form of control hazard
- Consider malfunction on add in EX stage  
add x1, x2, x1
  - Prevent x1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set SEPC and SCAUSE register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

# Exceptions in a Pipeline

		1	2	3	4	5	6	7	8	9	10	11	12	13						
	<b>I1</b>	F	D	E	M	W														
	add x1, x2, x1		F																	
	<b>I3</b>																			
	<b>I4</b>																			
	<b>I5</b>																			
	<b>IHS</b>																			

# 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 SEPC register
  - Identifies causing instruction

# Exception Example

- Exception on `add` in

```
40    sub    x11, x2, x4
44    and    x12, x2, x5
48    or     x13, x2, x6
4c    add    x1,  x2, x1
50    sub    x15, x6, x7
54    ld     x16, 100(x7)
```

...

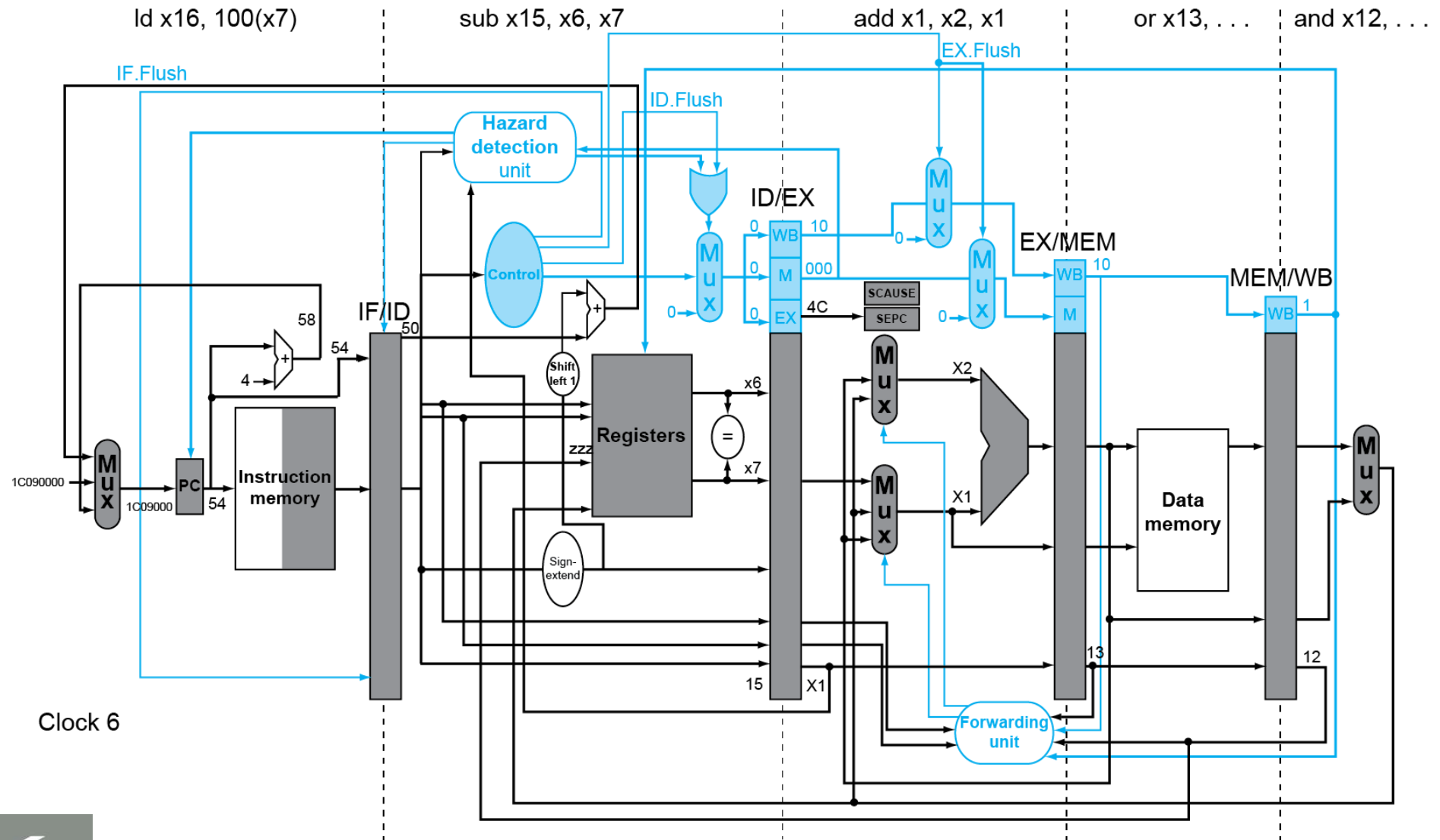
- Handler

```
1c090000    sd    x26, 1000(x10)
1c090004    sd    x27, 1008(x10)
```

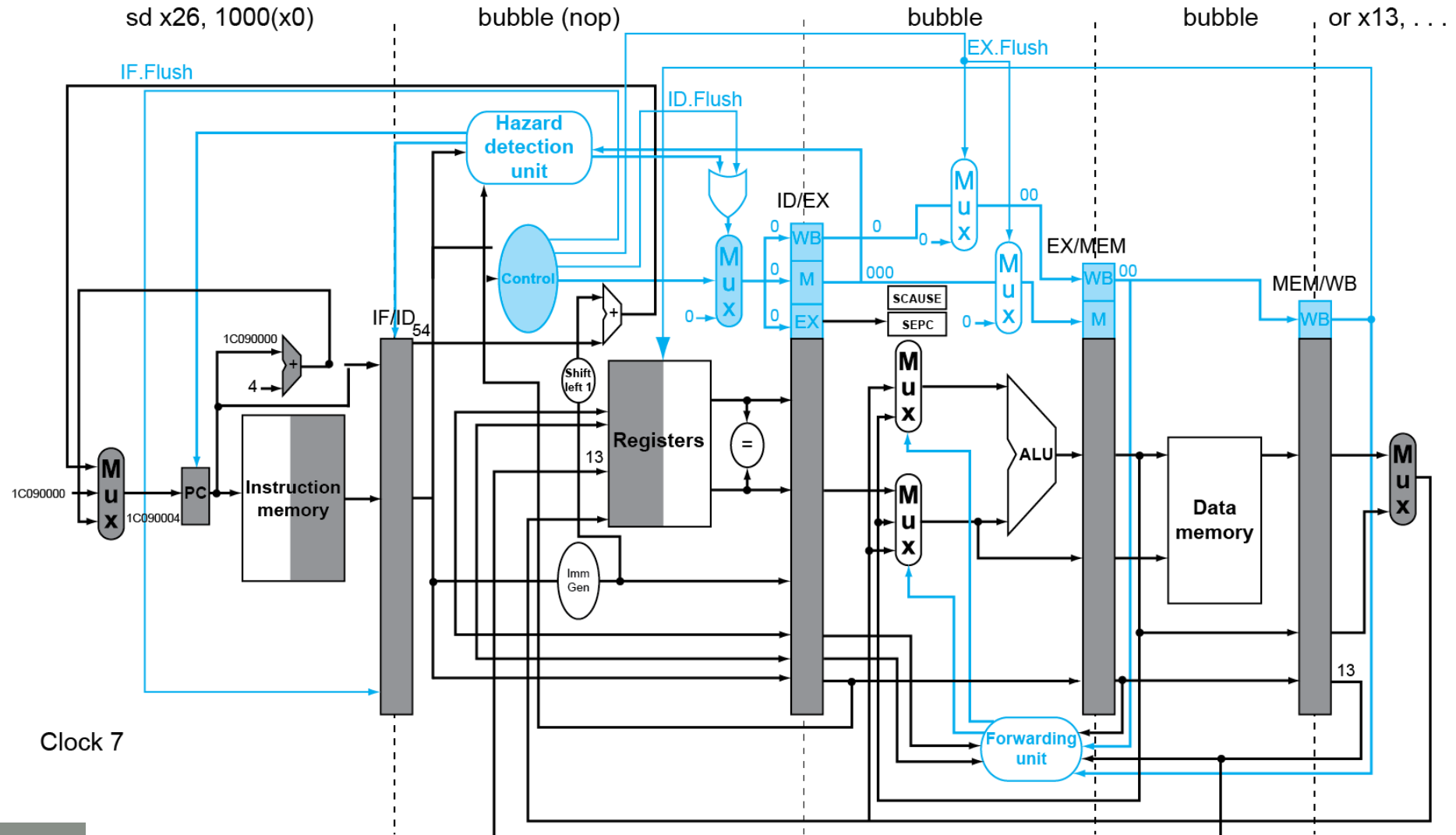
...



# Exception Example



# Exception Example



# Multiple Exceptions

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

# Multiple Exceptions

		1	2	3	4	5	6	7	8	9	10	11	12	13					
	I1	F	D	E	M	W													
	add x1,x2,x1		F																
	I3 (bad)																		
	I4																		
	I5																		
IHS																			

# Imprecise Exceptions

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

# Contents

4.6 An Overview of Pipelining

4.7 Pipelined Datapath and Control (Review)

4.8 Data Hazards: Forwarding versus Stalling

4.9 Control Hazards

4.10 Exceptions

4.11 Parallelism via Instructions

4.12 Putting it All Together: The Intel Core i7 6700 and ARM Cortex-A53

4.15 Fallacies and Pitfalls

4.16 Concluding Remarks

# Contents

## 4.11 Parallelism via Instructions

Instruction-Level Parallelism (ILP)

Multiple Issue

Static Multiple Issue

VLIW

Scheduling Static Multiple Issue

Loop Unrolling

Dynamic Multiple Issue

Register Renaming

Speculation

Why Do Dynamic Scheduling

# 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

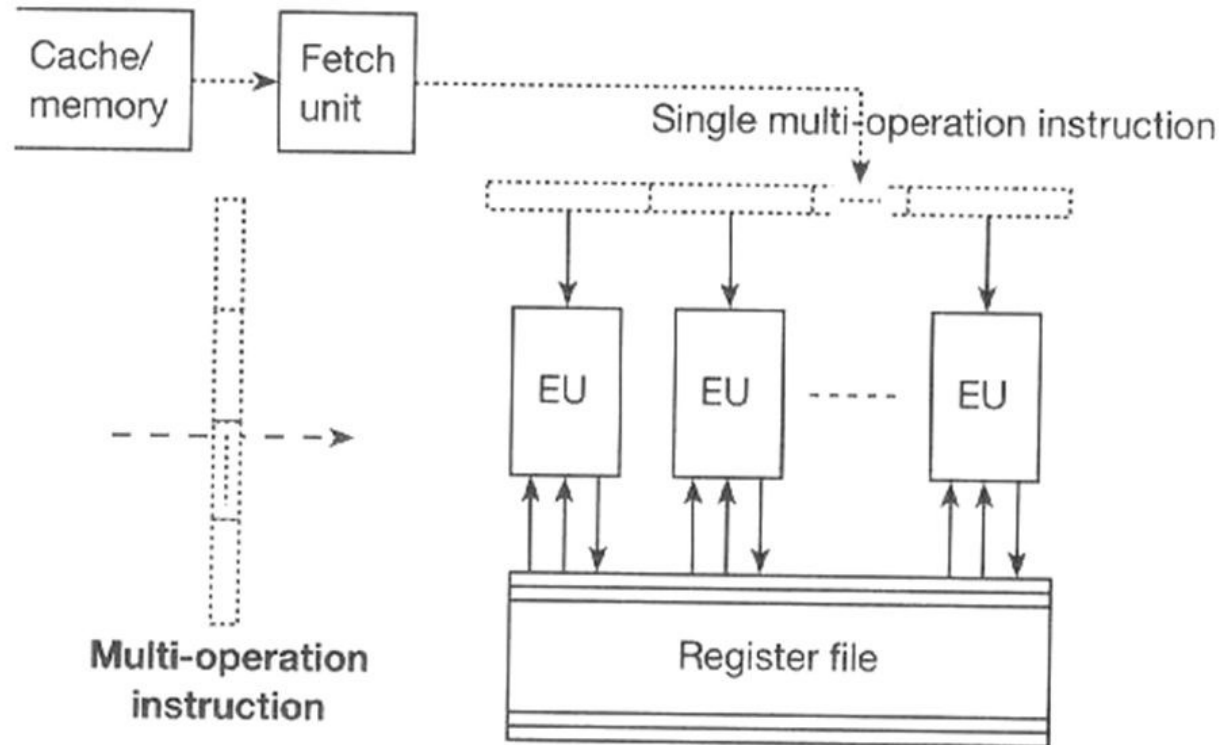
- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions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

# Static Multiple Issue

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

## VLIW

(very long instruction word, 1024 bits!)



VLIW approach

# Scheduling Static Multiple Issue

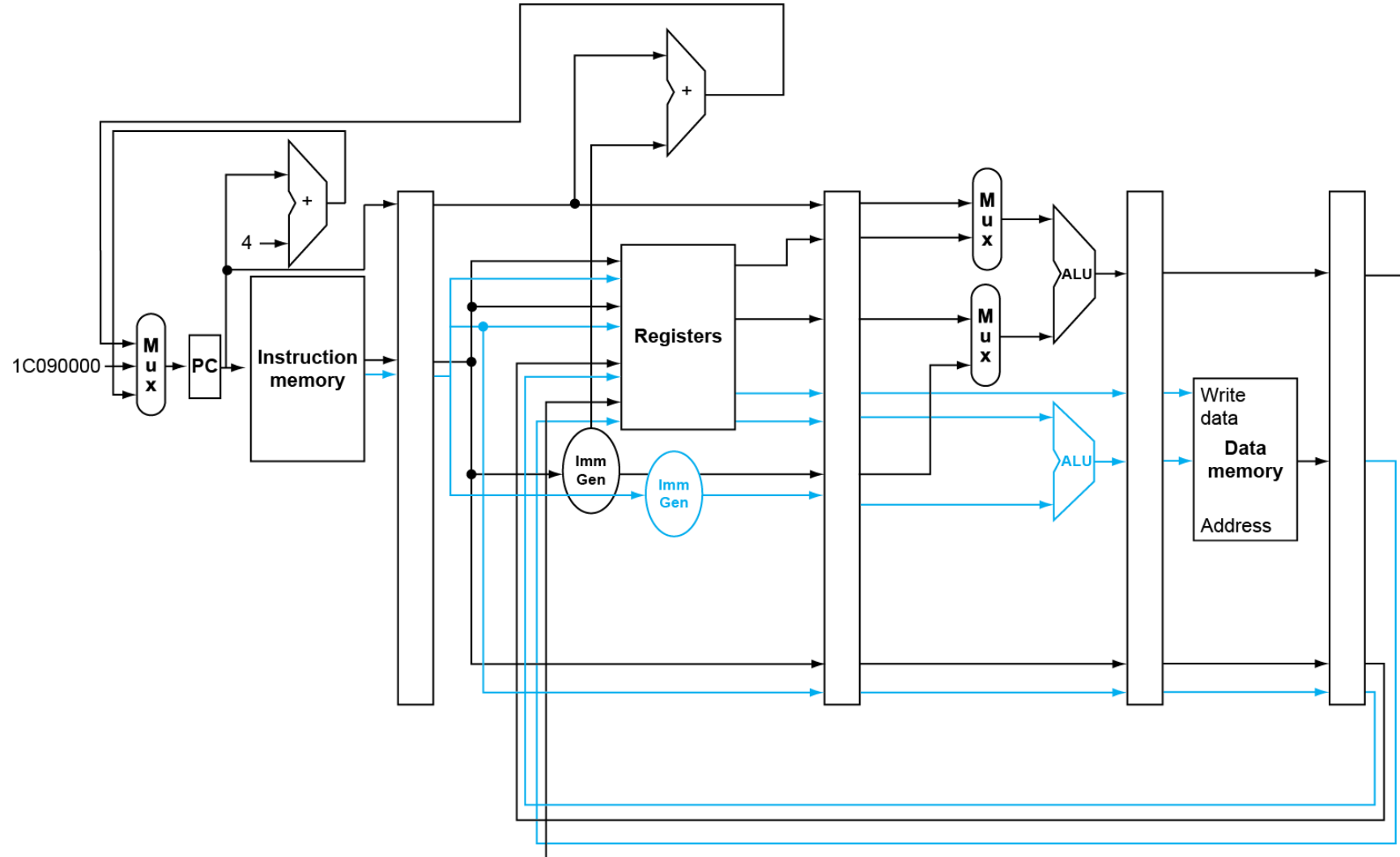
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

# RISC-V 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	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

# RISC-V with Static Dual Issue



# Hazards in the Dual-Issue RISC-V

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - `add x10, x0, x1`  
`ld x2, 0(x10)`
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

# Hazards in the Dual-Issue RISC-V

	1	2	3	4	5	6	7	8	9	10
add <b>x10</b> , x0, x1	F	D	E	M	W					
nop	<b>F</b>	<b>D</b>	<b>E</b>	<b>M</b>	<b>W</b>					
nop										
ld x2, 0( <b>x10</b> )										



# Hazards in the Dual-Issue RISC-V

- Load-use hazard
  - `ld x31, 0(x20)`  
`add x31, x31, x21`
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

# Hazards in the Dual-Issue RISC-V

	1	2	3	4	5	6	7	8	9	10
nop	<b>F</b>	<b>D</b>	<b>E</b>	<b>M</b>	<b>W</b>					
ld <b>x31</b> , 0(x20)	<b>F</b>	<b>D</b>	<b>E</b>	<b>M</b>	<b>W</b>					
nop										
nop										
add x31, <b>x31</b> , 21										

# Forwarding in Dual-Issue RISC-V

- In addition to forwarding from M and W to E, there are additional forwarding paths among the two pipelines, e.g.:
  - From W in memory pipeline to E in ALU pipeline
    - `ld x31, 0(x20)`  
`add x31, x31, x21`
    - *Refer to the previous slide*
  - From W in ALU pipeline to M in memory pipeline
    - `add x31, x31, x21`  
`sd x31, 0(x20)`

# Forwarding in Dual-Issue RISC-V

From W in ALU pipeline to M in memory pipeline

	1	2	3	4	5	6	7	8	9	10
add <b>x31</b> , x31, x21	<b>F</b>	<b>D</b>	<b>E</b>	<b>M</b>	<b>W</b>					
nop										
nop										
sd <b>x31</b> , 0(x20)										

# Scheduling Example

- Schedule this for dual-issue RISC-V

```
Loop: ld    x31,0(x20)    // x31=array element
      add  x31,x31,x21    // add scalar in x21
      sd   x31,0(x20)    // store result
      addi x20,x20,-8     // decrement pointer
      blt  x22,x20,Loop  // branch if x22 < x20
```

	ALU/branch	Load/store	cycle
Loop:			1
			2
			3
			4

# Scheduling Example

- Schedule this for dual-issue RISC-V

```
Loop: ld    x31,0(x20)    // x31=array element
      add  x31,x31,x21    // add scalar in x21
      sd   x31,0(x20)    // store result
      addi x20,x20,-8    // decrement pointer
      blt  x22,x20,Loop  // branch if x22 < x20
```

	ALU/branch	Load/store	cycle
Loop:	nop	ld x31,0(x20)	1
	addi x20,x20,-8	nop	2
	add x31,x31,x21	nop	3
	blt x22,x20,Loop	sd x31,8(x20)	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”, write-after-read
    - Or “output dependence”, write-after-write
      - Reuse of a register name

# Unrolling Steps

1. Replicate the loop instructions  $n$  times
2. Remove unneeded loop overhead
3. Modify instructions
4. Rename registers
5. Schedule instructions



# Example

Loop:

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
addi  x20, x20, -8
blt   x22, x20, Loop
```

# 1. Replicate the loop instructions 4 times

Loop:

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
addi  x20, x20, -8
blt   x22, x20, Loop
```

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
addi  x20, x20, -8
blt   x22, x20, Loop
```

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
addi  x20, x20, -8
blt   x22, x20, Loop
```

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
addi  x20, x20, -8
blt   x22, x20, Loop
```

## 2. Remove unneeded loop overhead

Loop:

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
addi  x20, x20, -8
blt   x22, x20, Loop
```

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
addi  x20, x20, -8
blt   x22, x20, Loop
```

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
addi  x20, x20, -8
blt   x22, x20, Loop
```

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
addi  x20, x20, -8
blt   x22, x20, Loop
```

## 2. Remove unneeded loop overhead

Loop:

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
```

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
```

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
```

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
addi  x20, x20, -8
blt   x22, x20, Loop
```

## 3. Modify instructions

Loop:

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
```

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
```

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
```

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
addi  x20, x20, -8
blt   x22, x20, Loop
```

## 3. Modify instructions

Loop:

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
```

```
ld    x31, -8(x20)
add   x31, x31, x21
sd    x31, -8(x20)
```

```
ld    x31, -16(x20)
add   x31, x31, x21
sd    x31, -16(x20)
```

```
ld    x31, -24(x20)
add   x31, x31, x21
sd    x31, -24(x20)
addi  x20, x20, -32
blt   x22, x20, Loop
```

## 4. Rename registers

Loop:

```
ld    x31, 0(x20)
add   x31, x31, x21
sd    x31, 0(x20)
```

```
ld    x31, -8(x20)
add   x31, x31, x21
sd    x31, -8(x20)
```

```
ld    x31, -16(x20)
add   x31, x31, x21
sd    x31, -16(x20)
```

```
ld    x31, -24(x20)
add   x31, x31, x21
sd    x31, -24(x20)
addi  x20, x20, -32
blt   x22, x20, Loop
```

## 4. Rename registers

Loop:

```
ld    x28, 0(x20)
add   x28, x28, x21
sd    x28, 0(x20)
```

```
ld    x29, -8(x20)
add   x29, x29, x21
sd    x29, -8(x20)
```

```
ld    x30, -16(x20)
add   x30, x30, x21
sd    x30, -16(x20)
```

```
ld    x31, -24(x20)
add   x31, x31, x21
sd    x31, -24(x20)
addi  x20, x20, -32
blt   x22, x20, Loop
```



## 5. Schedule instructions

Loop:

```
ld    x28, 0(x20)
add   x28, x28, x21
sd    x28, 0(x20)
ld    x29, -8(x20)
add   x29, x29, x21
sd    x29, -8(x20)
ld    x30, -16(x20)
add   x30, x30, x21
sd    x30, -16(x20)
ld    x31, -24(x20)
add   x31, x31, x21
sd    x31, -24(x20)
addi  x20, x20, -32
blt   x22, x20, Loop
```

	ALU/branch	Load/store	cycle
Loop:			1
			2
			3
			4
			5
			6
			7
			8

# Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi x20,x20,-32	ld x28, 0(x20)	1
	nop	ld x29, 24(x20)	2
	add x28,x28,x21	ld x30, 16(x20)	3
	add x29,x29,x21	ld x31, 8(x20)	4
	add x30,x30,x21	sd x28, 32(x20)	5
	add x31,x31,x21	sd x29, 24(x20)	6
	nop	sd x30, 16(x20)	7
	blt x22,x20,Loop	sd x31, 8(x20)	8

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

# Dynamic Multiple Issue

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

# Dynamic Pipeline Scheduling

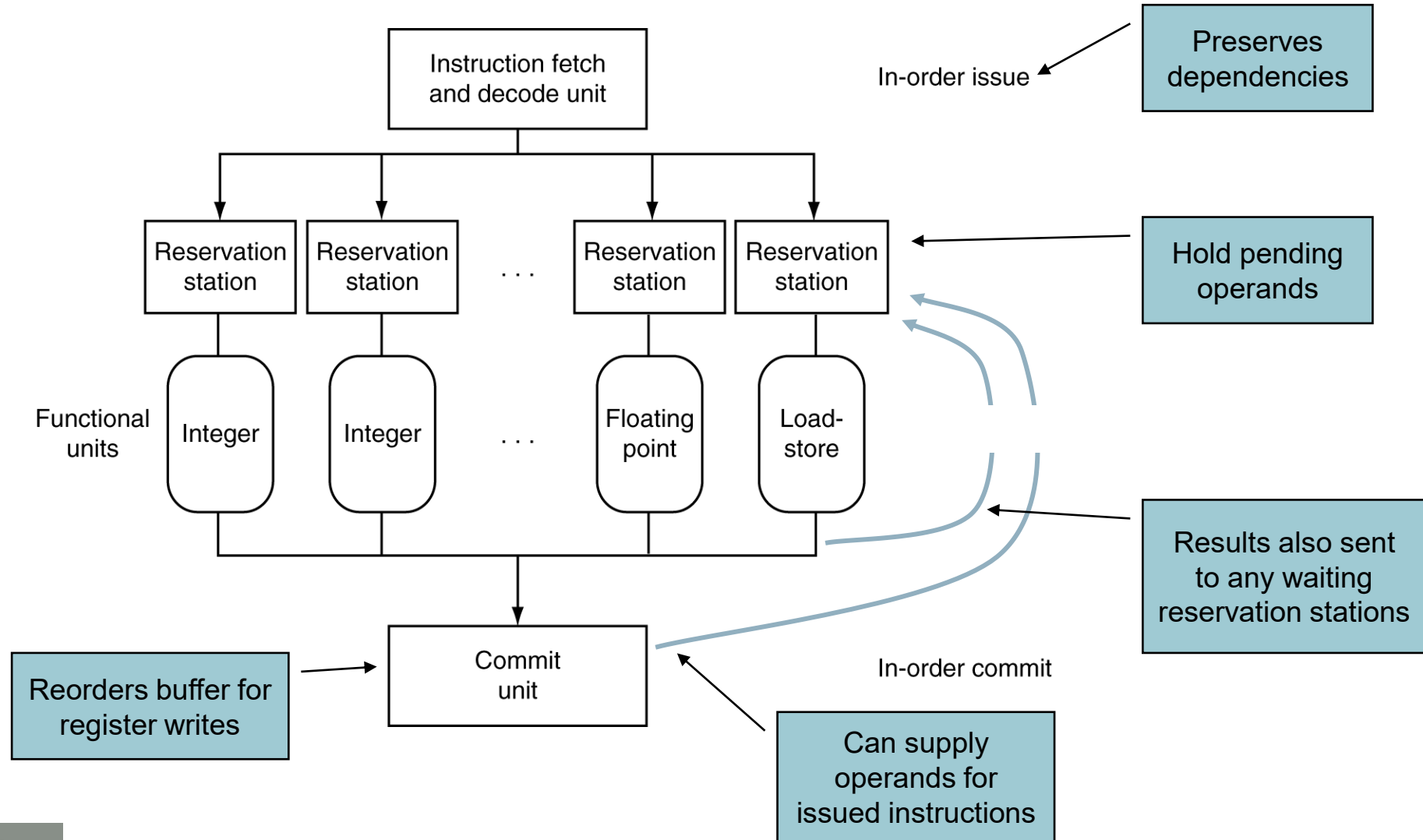
- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order

- Example

```
ld    x31, 20(x21)
add   x1, x31, x2
sub   x23, x23, x3
andi  x5, x23, 20
```

- Can start sub while add is waiting for ld

# Dynamically Scheduled CPU



# Pipeline Stages

**F:** Fetch from instr. memory (IM) to instr. queue (IQ).

**I:** Issue from IQ to reservation stations (RS), reading ready operands from register file (RF).

**E:** Execute when functional unit (FU) is free and instr. in RS has ready operands.

**W:** Write result from FU through common data bus (CDB) to reorder buffer (ROB) and RS.

**C:** Commit results in order from ROB to RF and memory.

■ Loads have **FIAMWC**, stores have **FIAC**. **A:** Address calculation

# Single-issue Example

	1	2	3	4	5	6	7	8	9	10
ld x31, 20(x21)										
add x1, x31, x2										
sub x23, x23, x3										
andi x5, x23, 20										

# 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



# Examples

- Assume superscalar processor of degree 3
- Name dependence (WAR)

```
mul    x1, x2, x3
add    x4, x1, x5
ld     x5, 16(x21)
```

- Output dependence (WAW)

```
mul    x1, x2, x3
add    x4, x1, x5
ld     x1, 16(x21)
```

# Triple Issue: Name dependence (WAR)

Assume multiplication latency is 3 cycles

		1	2	3	4	5	6	7	8	9	10
mul	x1, x2, x3										
add	x4, x1, x5										
ld	x5, 16(x21)										

# Triple Issue: Output Dependence (WAW)

Assume multiplication latency is 3 cycles

		1	2	3	4	5	6	7	8	9	10
mul	x1, x2, x3										
add	x4, x1, x5										
ld	x1, 16(x21)										

# Speculation

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

# Compiler/Hardware Speculation

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

# Branch Speculation

- Predict branch and continue issuing
  - Don't commit until branch outcome determined
- **Example:** Assume a superscalar processor of degree 2 and the branch prediction is not taken.

```
1d    x1, 0(x20)
beq   x1, x2, skip
I3
I4
```

# Example: Assume a superscalar processor of degree 2 and the branch prediction is not taken. (Correct prediction)



		1	2	3	4	5	6	7	8	9	0
ld	x1, 0(x20)	F	I								
beq	x1, x2, Skip	F	I								
I3			F	I							
I4			F	I							

...

Skip:

# Example: Assume a superscalar processor of degree 2 and the branch prediction is not taken. (**Incorrect prediction**)



		1	2	3	4	5	6	7	8	9	0
ld	<b>x1</b> , 0(x20)	F	I								
beq	<b>x1</b> , x2, Skip	F	I								
I3			F	I							
I4			F	I							

...

**Skip:**



# Load Speculation

- Avoid load and cache miss delay
  - Load before completing outstanding stores
  - Predict the effective address or loaded value
  - Bypass stored values to load unit
- Don't commit load until speculation cleared
- **Example:** Superscalar of degree 3.

ld x1, 0(x20)

sd x2, 0(x1)

ld x3, 0(x21)

**Example: Load speculation. Assume a superscalar processor of degree 3. Predict the second load does not depend on the store. (Correct prediction)**

		1	2	3	4	5	6	7	8	9	0
ld	x1, 0(x20)	F	I								
sd	x2, 0(x1)	F	I								
ld	x3, 0(x21)	F	I								

**Example: Load speculation. Assume a superscalar processor of degree 3.  
Predict the second load does not depend on the store. (Incorrect prediction)**

		1	2	3	4	5	6	7	8	9	0
ld	x1, 0(x20)	F	I								
sd	x2, 0(x1)	F	I								
ld	x3, 0(x21)	F	I								

# Speculation and Exceptions

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

# Exceptions Examples

- Assume superscalar processor of degree 3 with 2 address calculation units
- **E1**: Predict branch as not take, but resolve to taken. **The Td has exception in M.**

		1	2	3	4	5	6	7	8	9	0
beq	x1, x2, L1	F	I								
ld	x5, 16(x21)	F	I								

# Exceptions Examples

- Assume superscalar processor of degree 3 with 2 address calculation units
- **E2:** Assume **the first sd has exception in C.**

		1	2	3	4	5	6	7	8	9	0
ld	<b>x1</b> , 0(x20)	F	I								
sd	<b>x1</b> , 0(x21)	F	I								
sd	x2, 16(x21)	F	I								

# Why Do Dynamic Scheduling?

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

# Does Multiple Issue Work?

## The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit 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/ Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5W
Intel Pentium	1993	66 MHz	5	2	No	1	10W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103W
Intel Core	2006	3000 MHz	14	4	Yes	2	75W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Yes	2-4	87W
Intel Core Westmere	2010	3730 MHz	14	4	Yes	6	130W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Yes	6	130W
Intel Core Broadwell	2014	3700 MHz	14	4	Yes	10	140W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Yes	14	165W
Intel Ice Lake	2018	4200 MHz	14	4	Yes	16	185W

# Contents

---

4.6 An Overview of Pipelining

4.7 Pipelined Datapath and Control (Review)

4.8 Data Hazards: Forwarding versus Stalling

4.9 Control Hazards

4.10 Exceptions

4.11 Parallelism via Instructions

**4.12 Putting it All Together: The Intel Core i7 6700 and ARM Cortex-A53**

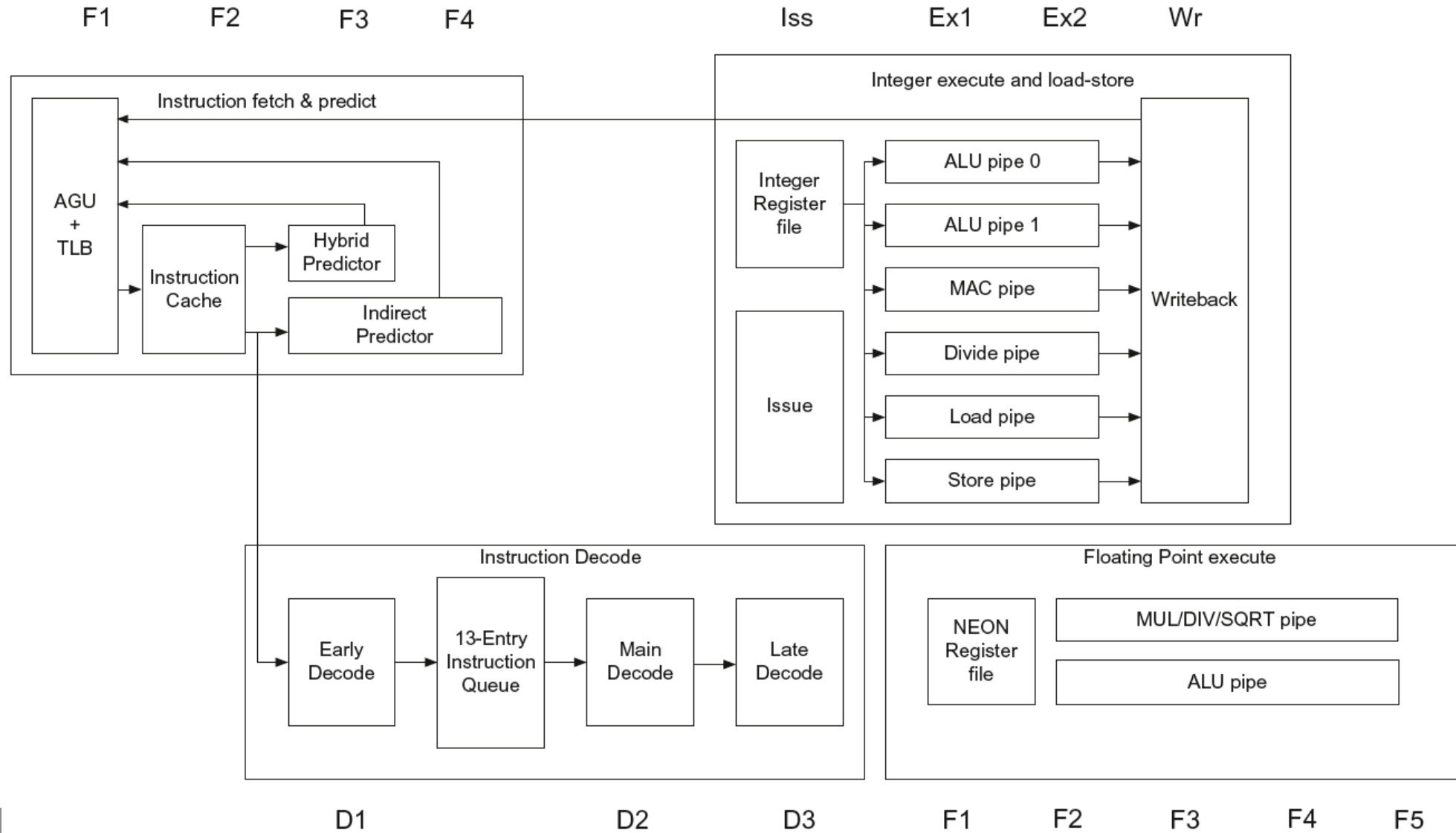
**4.15 Fallacies and Pitfalls**

**4.16 Concluding Remarks**

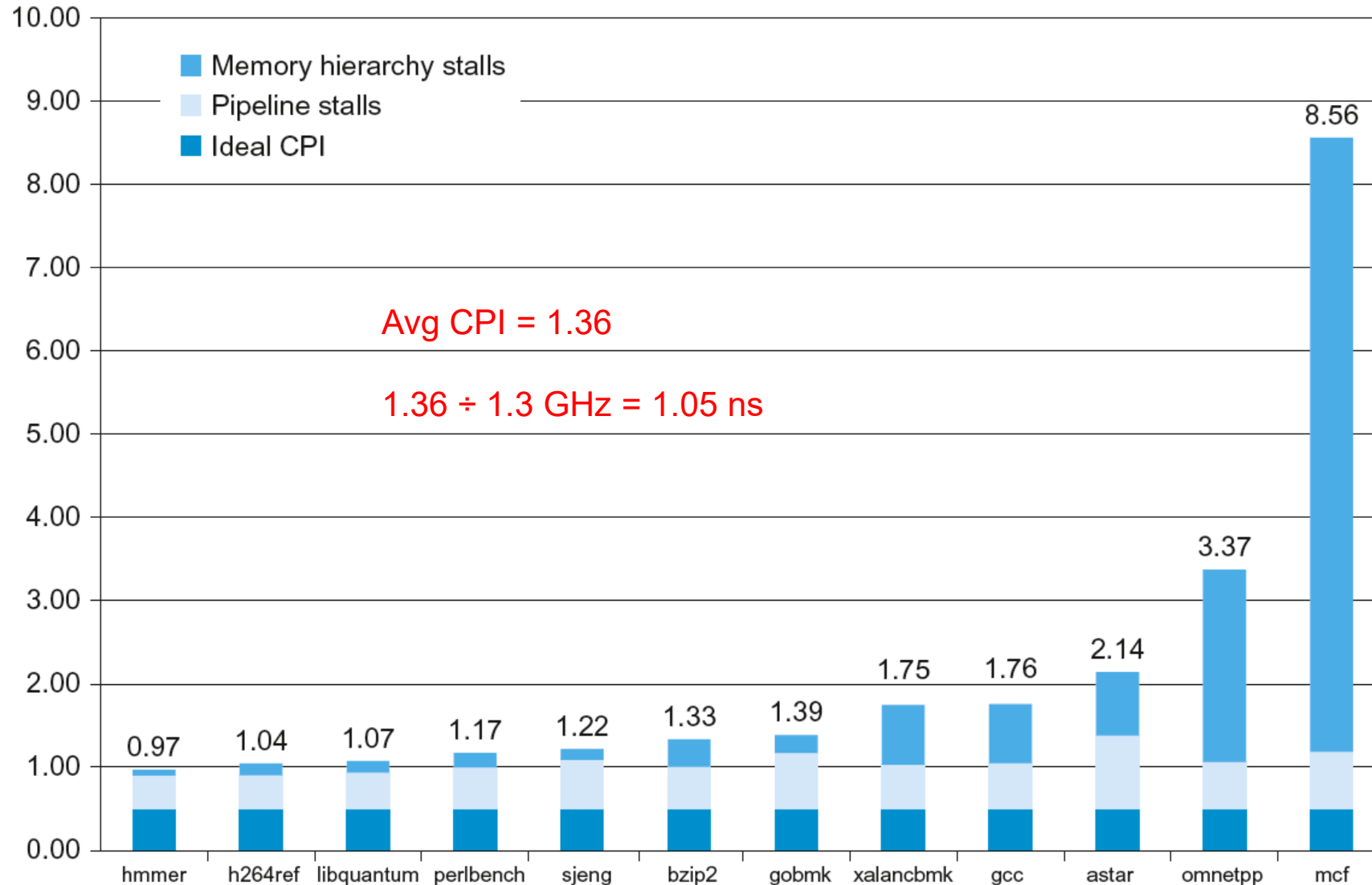
# Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 6700
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.3 GHz	3.4 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	Multi-level
1 <sup>st</sup> level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2 <sup>nd</sup> level caches/core	128-2048 KiB	256 KiB (per core)
3 <sup>rd</sup> level caches (shared)	(platform dependent)	8 MB

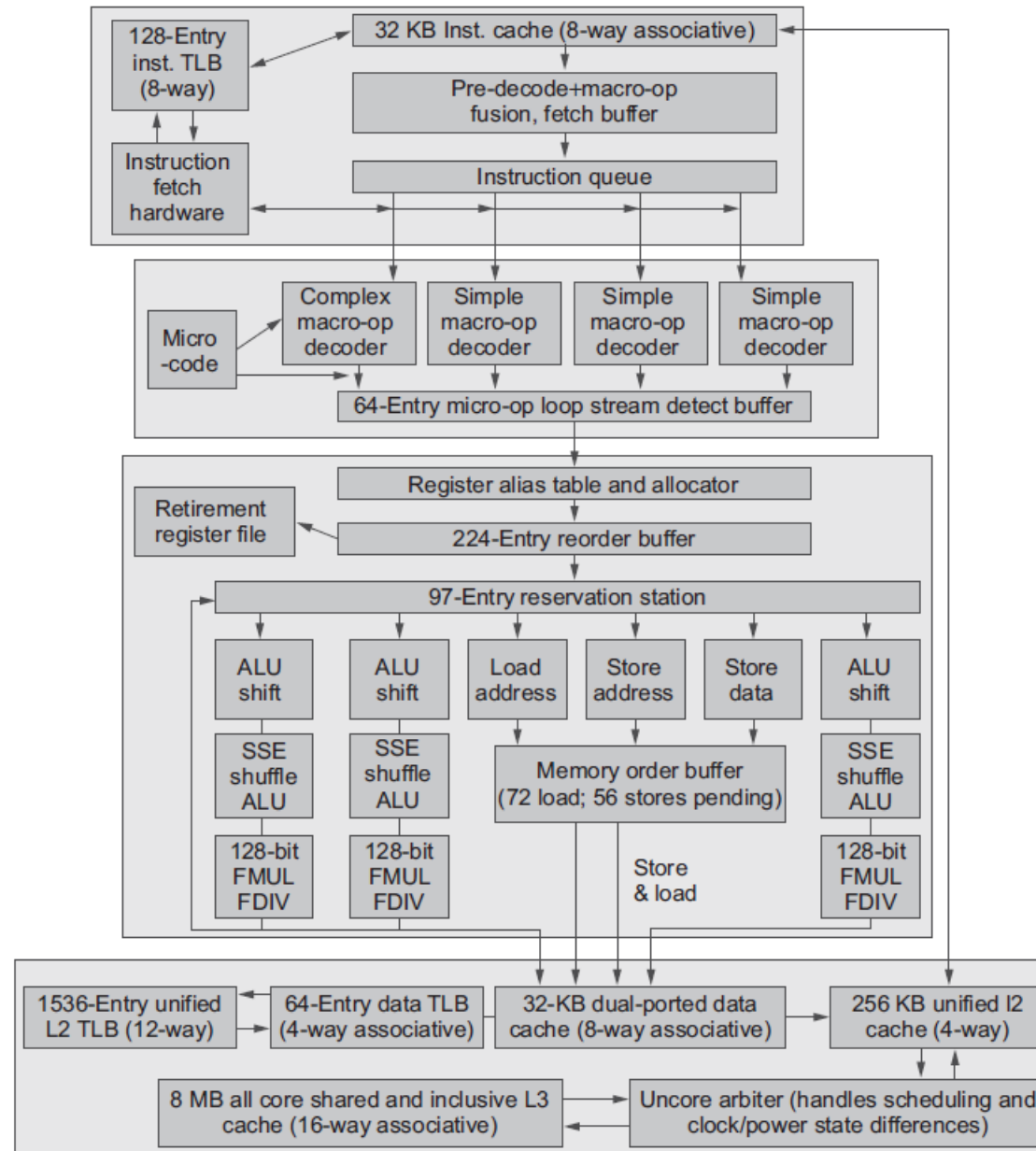
# ARM Cortex-A53 Pipeline



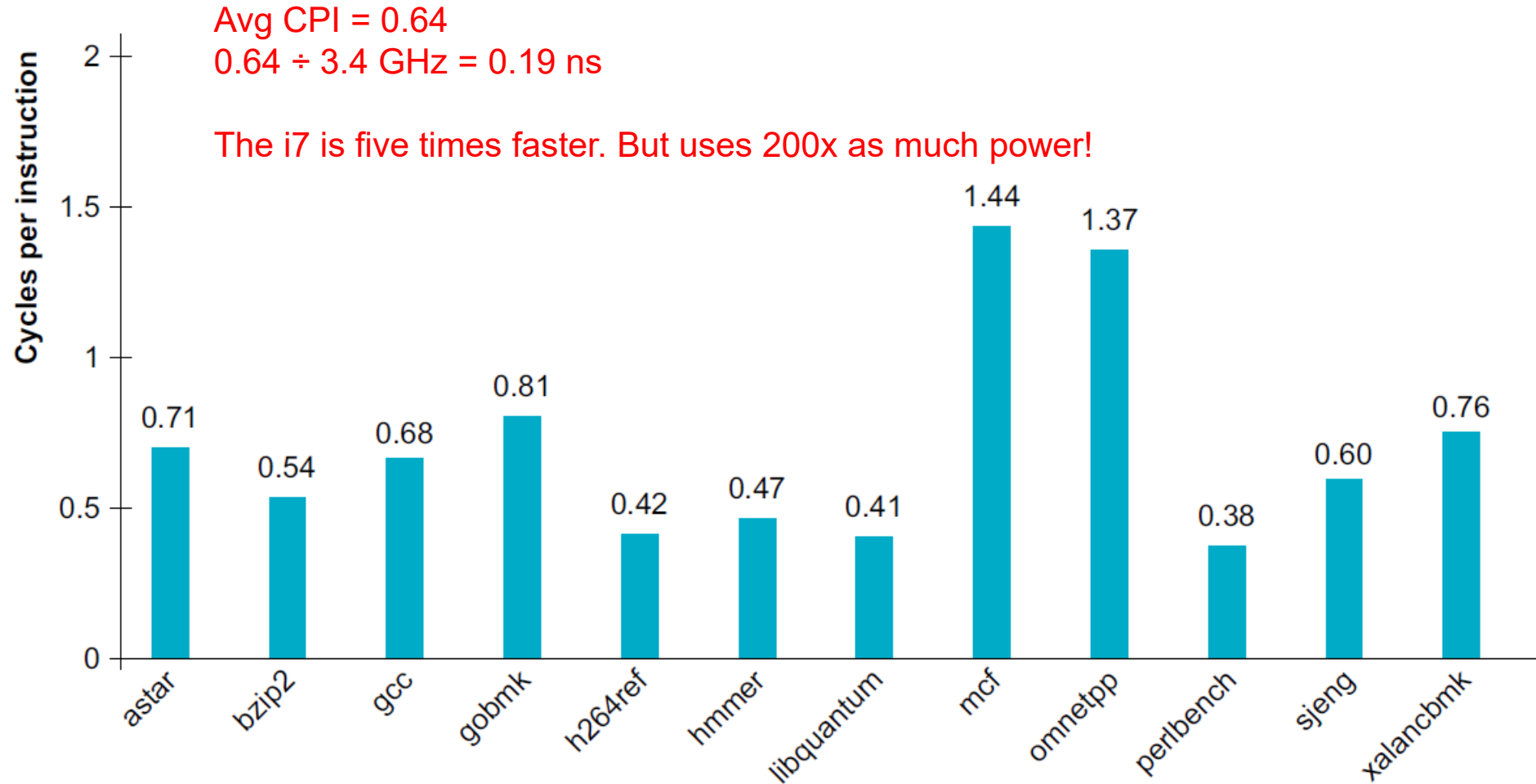
# ARM Cortex-A53 Performance



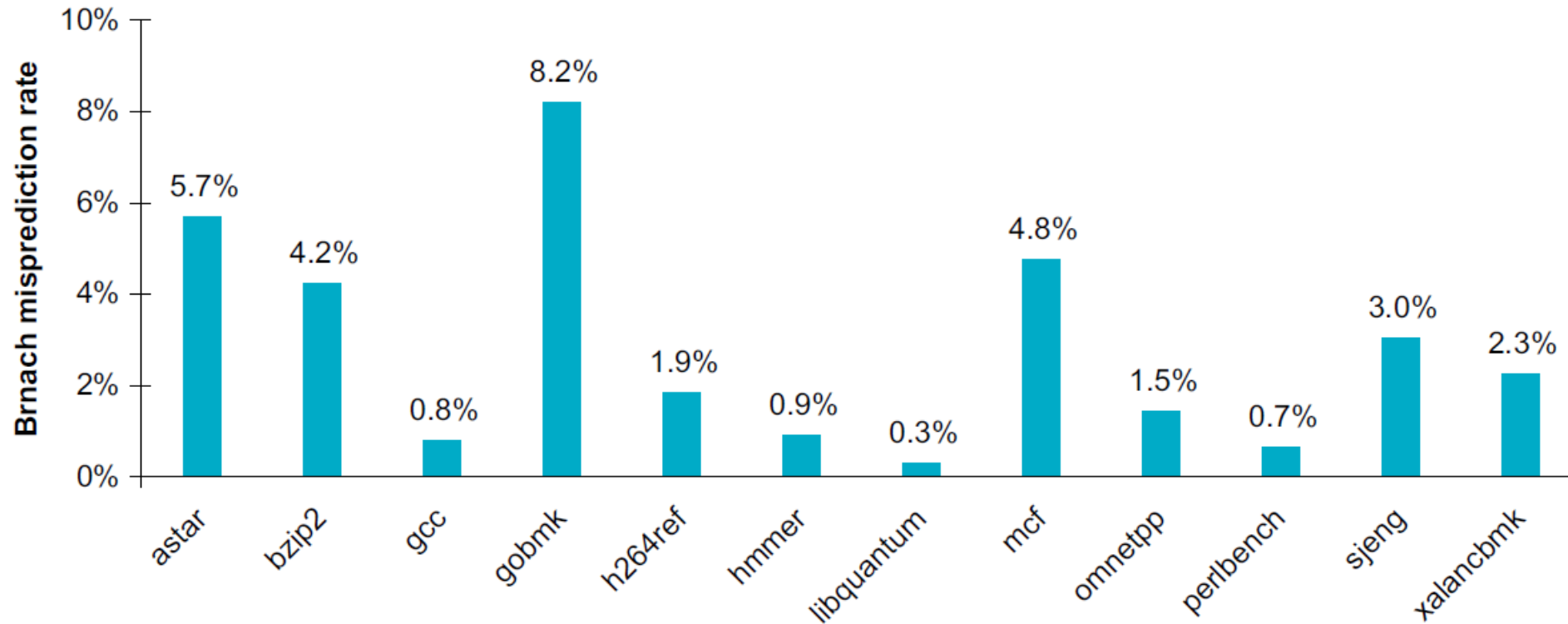
# Core i7 Pipeline



# Core i7 Performance



# Core i7 Performance





# Contents

---

4.6 An Overview of Pipelining

4.7 Pipelined Datapath and Control (Review)

4.8 Data Hazards: Forwarding versus Stalling

4.9 Control Hazards

4.10 Exceptions

4.11 Parallelism via Instructions

4.12 Putting it All Together: The Intel Core i7 6700 and ARM Cortex-A53

**4.15 Fallacies and Pitfalls**

**4.16 Concluding Remarks**

# 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

# Contents

---

- 4.6 An Overview of Pipelining
- 4.7 Pipelined Datapath and Control (Review)
- 4.8 Data Hazards: Forwarding versus Stalling
- 4.9 Control Hazards
- 4.10 Exceptions
- 4.11 Parallelism via Instructions
- 4.12 Putting it All Together: The Intel Core i7 6700 and ARM Cortex-A53
- 4.15 Fallacies and Pitfalls
- 4.16 Concluding Remarks**

# Concluding Remarks

- 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