

# Chapter 4

---

## The Processor

# Introduction



- › CPU performance factors
  - Instruction count
    - › Determined by ISA and compiler
  - CPI and Cycle time
    - › Determined by CPU hardware
  
- › We will examine two RISC-V implementations
  - A simplified version
  - A more realistic pipelined version

# Introduction



## › Simple subset, shows most aspects

- Memory reference: `ld, sd`

`ld x5, 40(x6)`

`sd x5, 40(x6)`

- Arithmetic/logical: `add, sub, and, or`

`add x9, x20, x21`

- Control transfer: `beq`

`beq rs1, rs2, L1`

Name (Bit position)	Fields					
	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

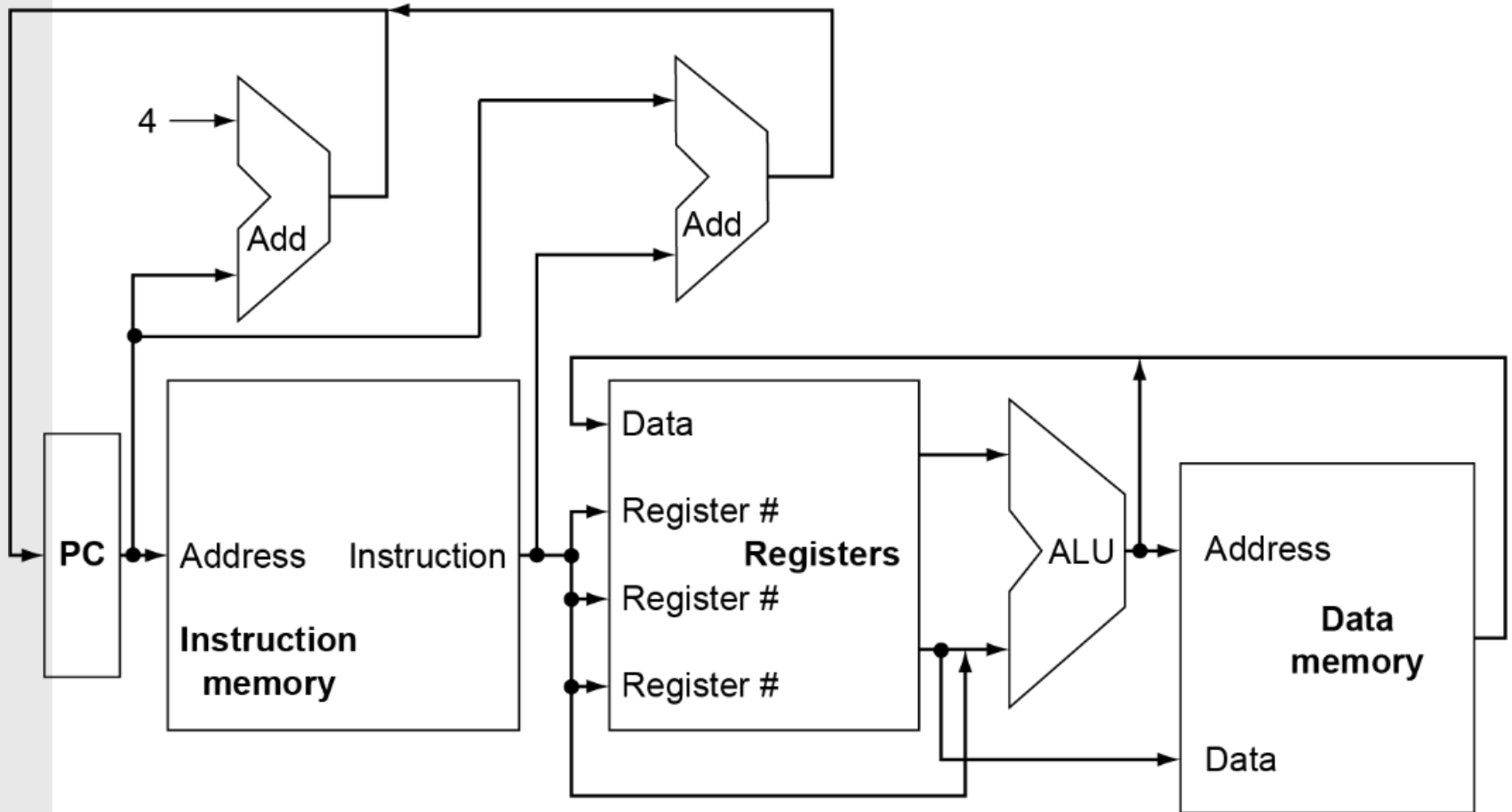
# Instruction Execution

---

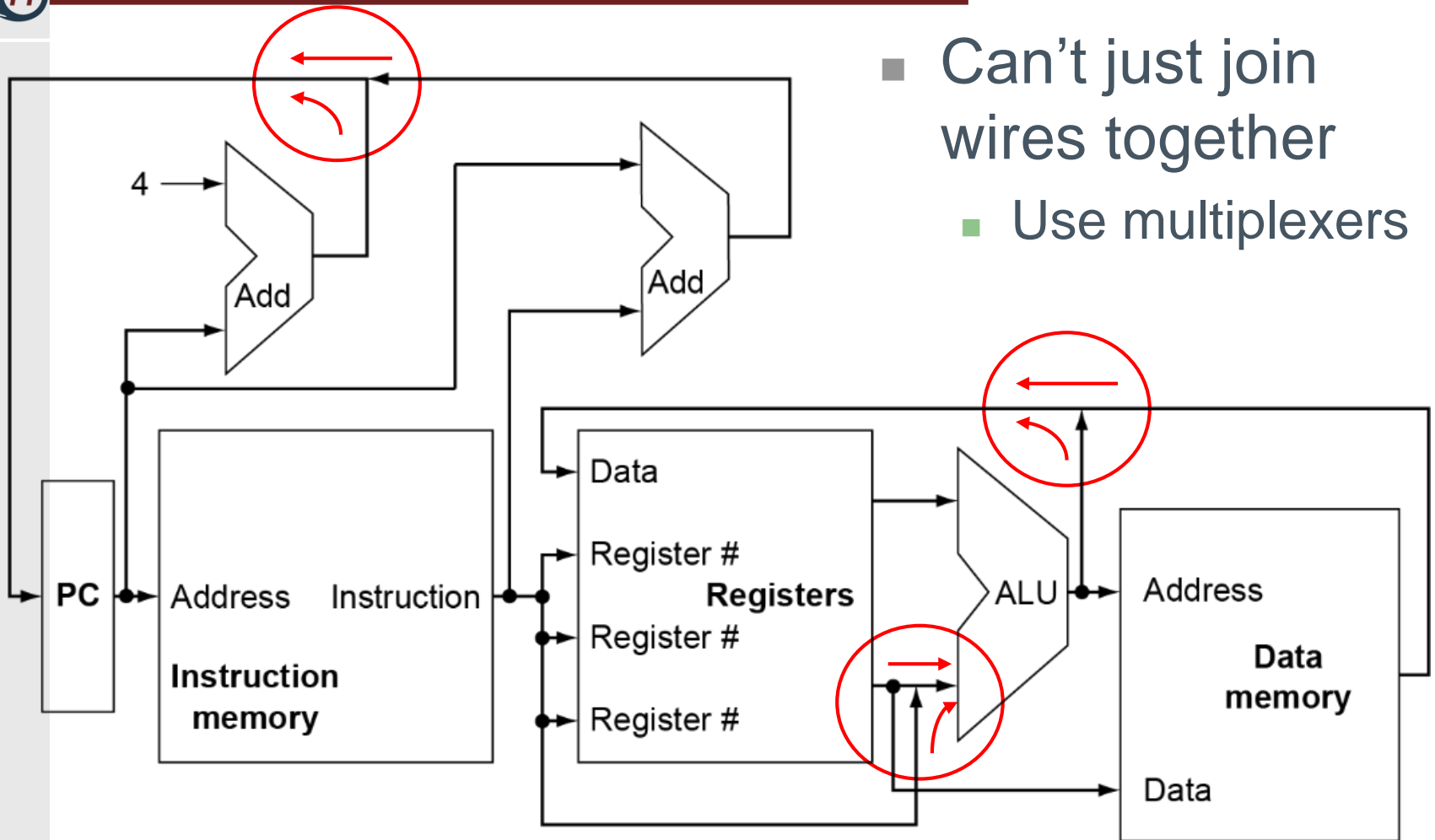


- › PC → instruction memory, fetch instruction
- › Register numbers → register file, read registers
- › Depending on instruction class
  - Use ALU to calculate
    - › Arithmetic result
    - › Memory address for load/store
    - › Branch comparison
  - Access data memory for load/store
  - PC ← target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

# CPU Overview

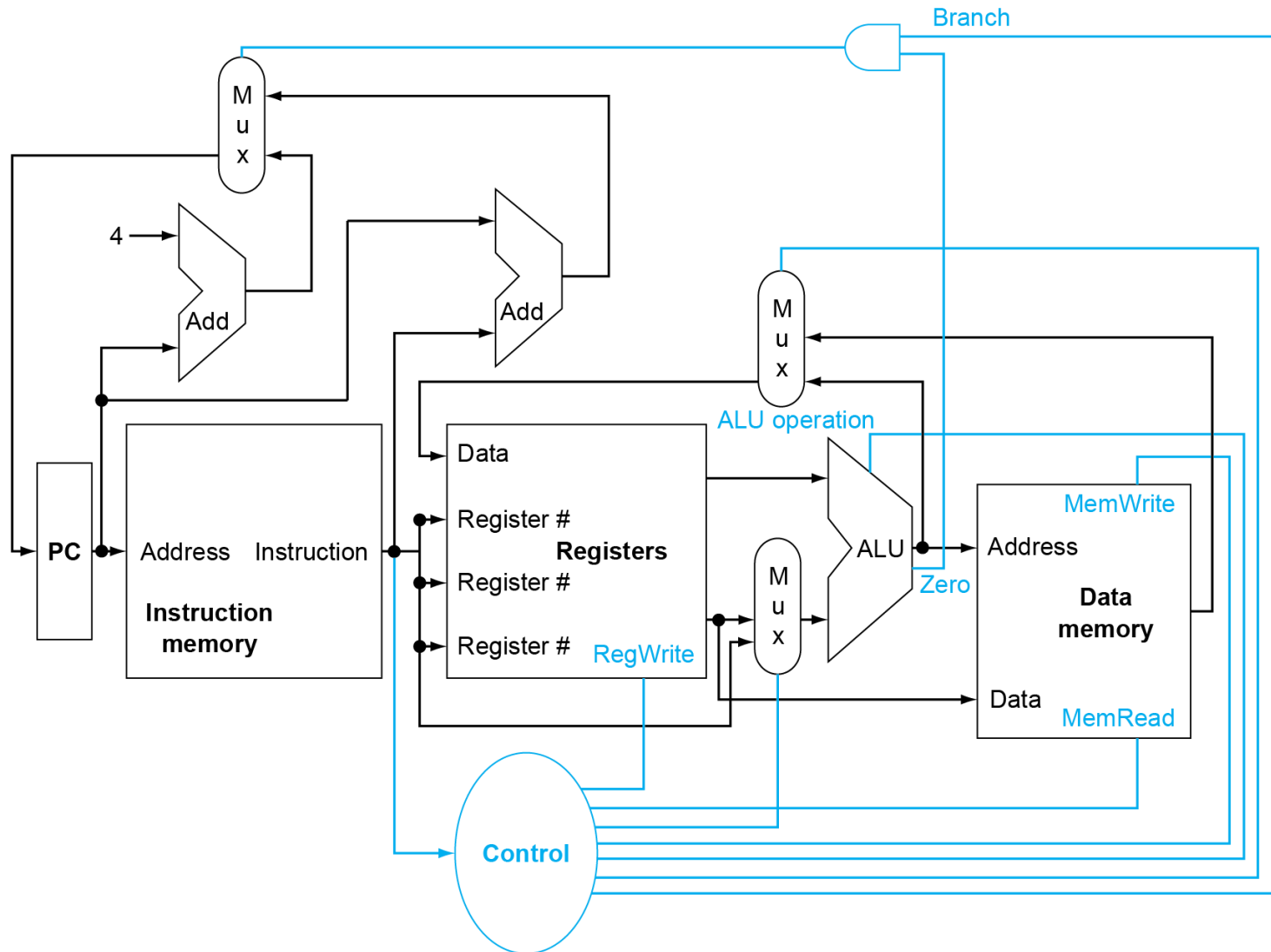


# Multiplexers



- Can't just join wires together
  - Use multiplexers

# Control



# Logic Design Basics

---



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



# Combinational Elements



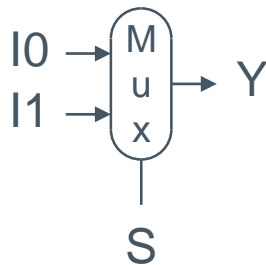
## > AND-gate

–  $Y = A \& B$



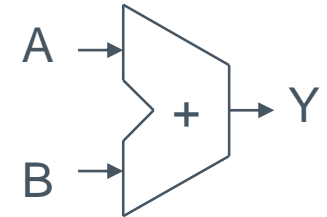
## ■ Multiplexer

■  $Y = S ? I1 : I0$



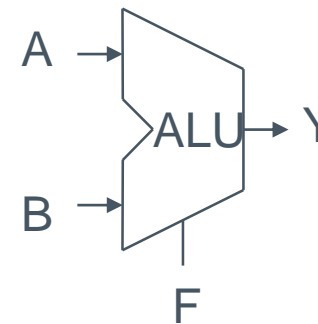
## ■ Adder

■  $Y = A + B$



## ■ Arithmetic/Logic Unit

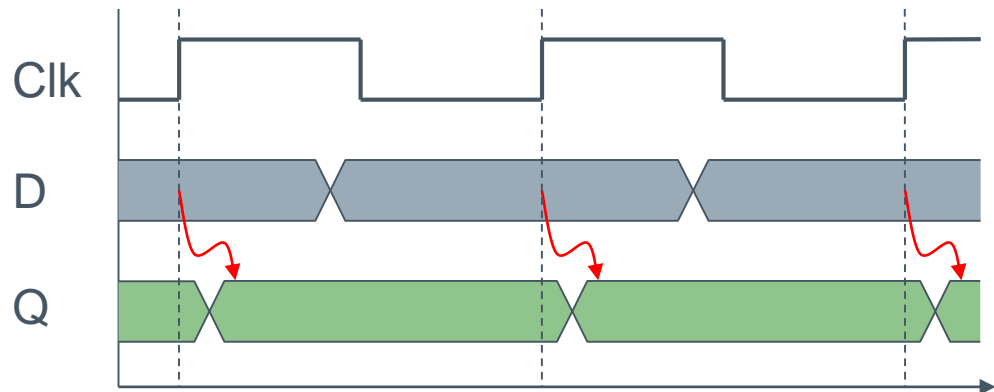
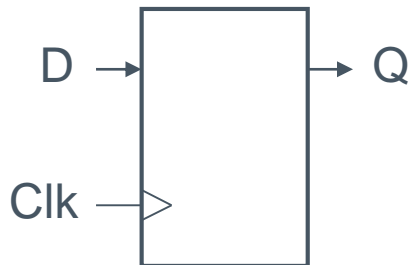
■  $Y = F(A, B)$



# 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

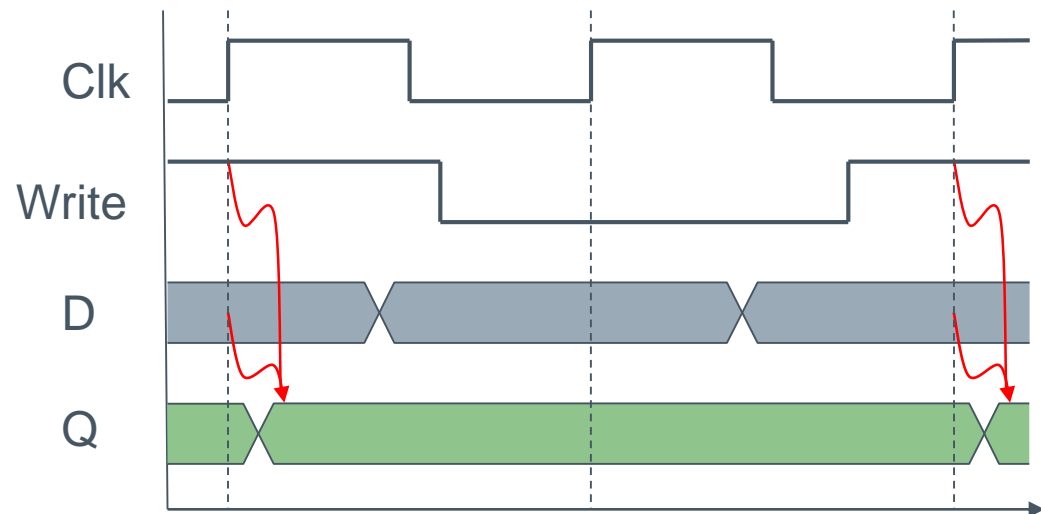
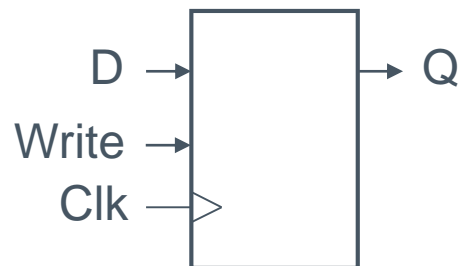


# Sequential Elements



## > Register with write control

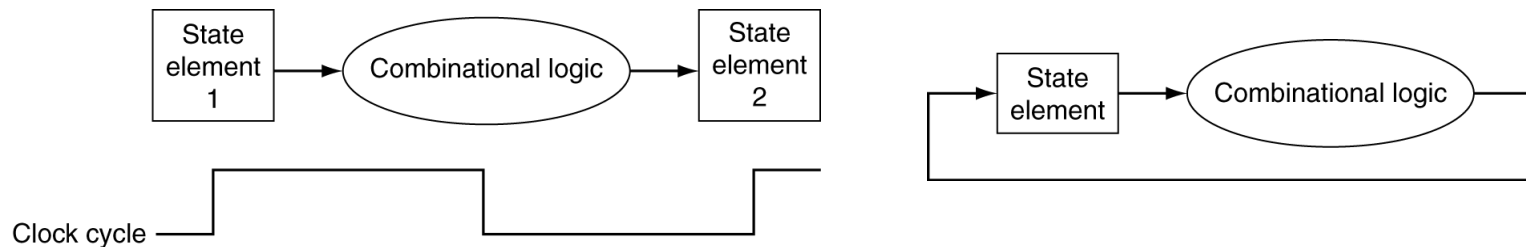
- Only updates on clock edge when write control input is 1
- Used when stored value is required later



# 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



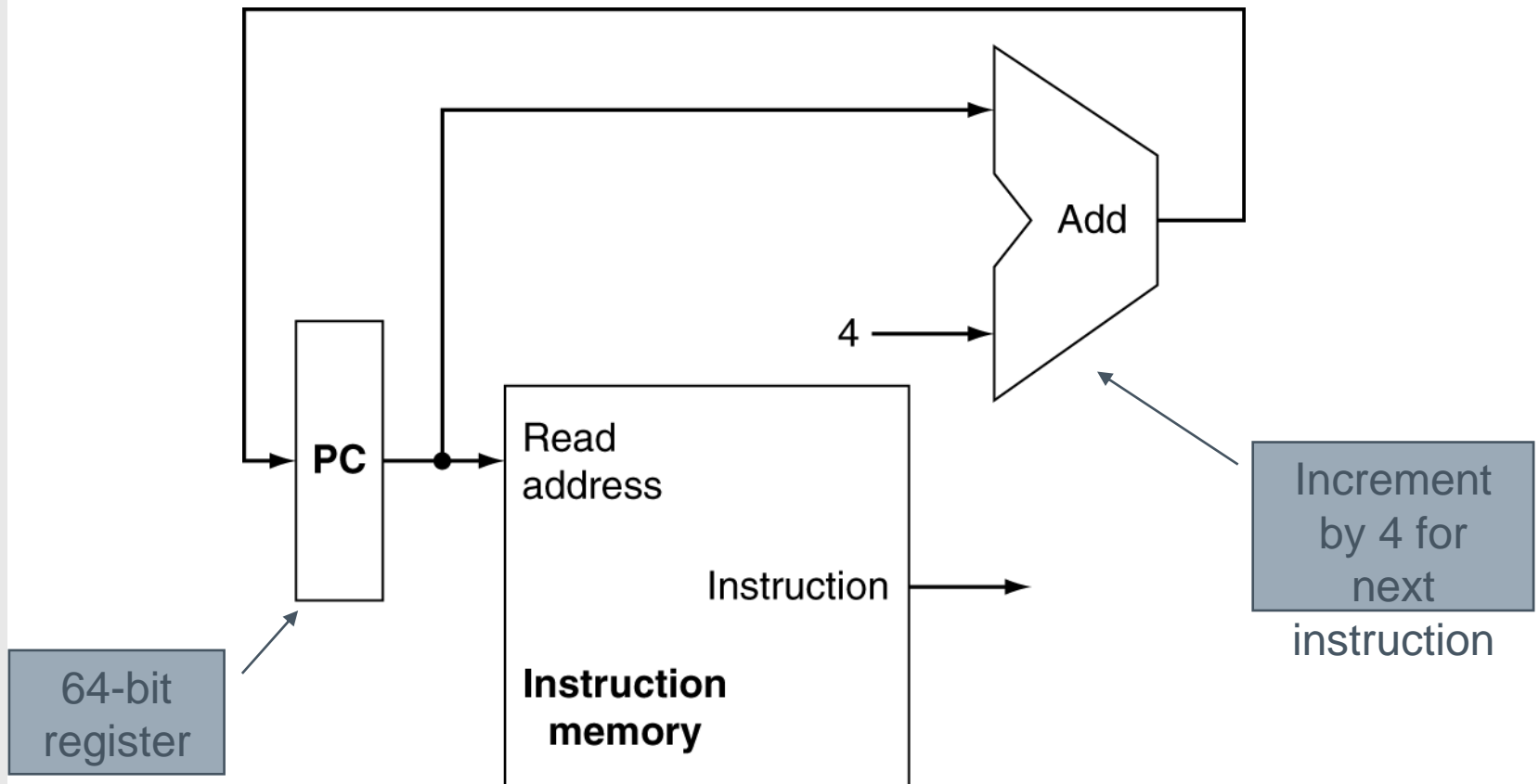
# Building a Datapath

---



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

# Instruction Fetch

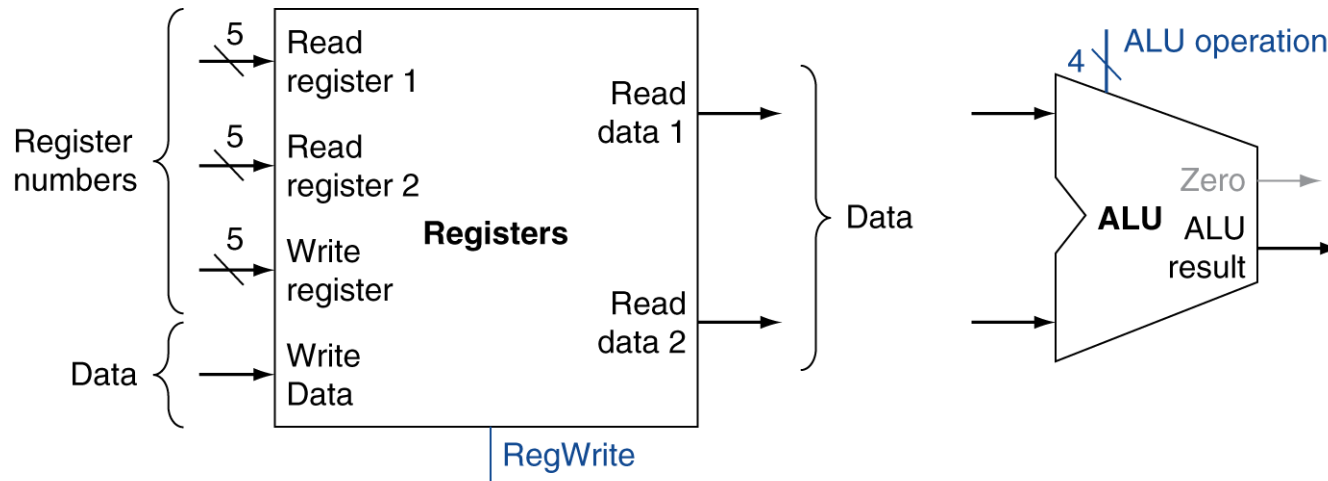


# R-Format Instructions

`add x9,x20,x21`



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



a. Registers

b. ALU

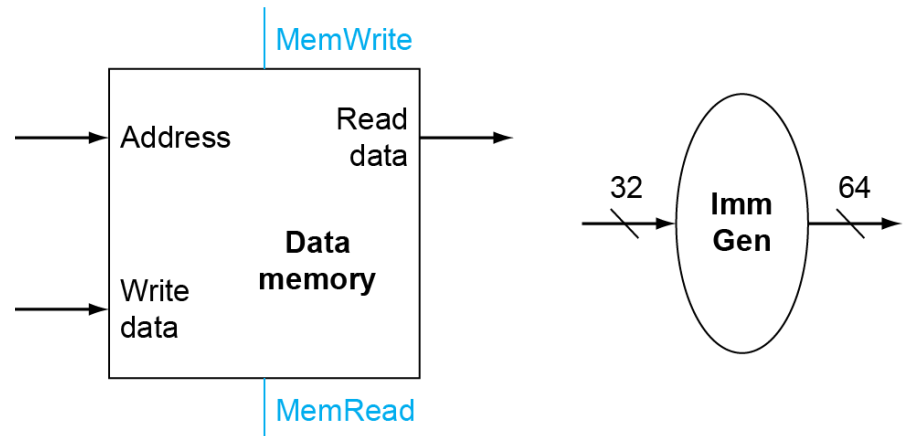
Name (Bit position)	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode

# Load/Store Instructions

ld x5, 40(x6)  
sd x5, 40(x6)



- › Read register operands
- › Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset



a. Data memory unit

b. Immediate generation unit

- › Load: Read memory and update register
- › Store: Write register value to memory

Name (Bit position)	31:25	24:20	19:15	14:12	11:7	6:0
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode



# Branch Instructions

beq rs1, rs2, L1

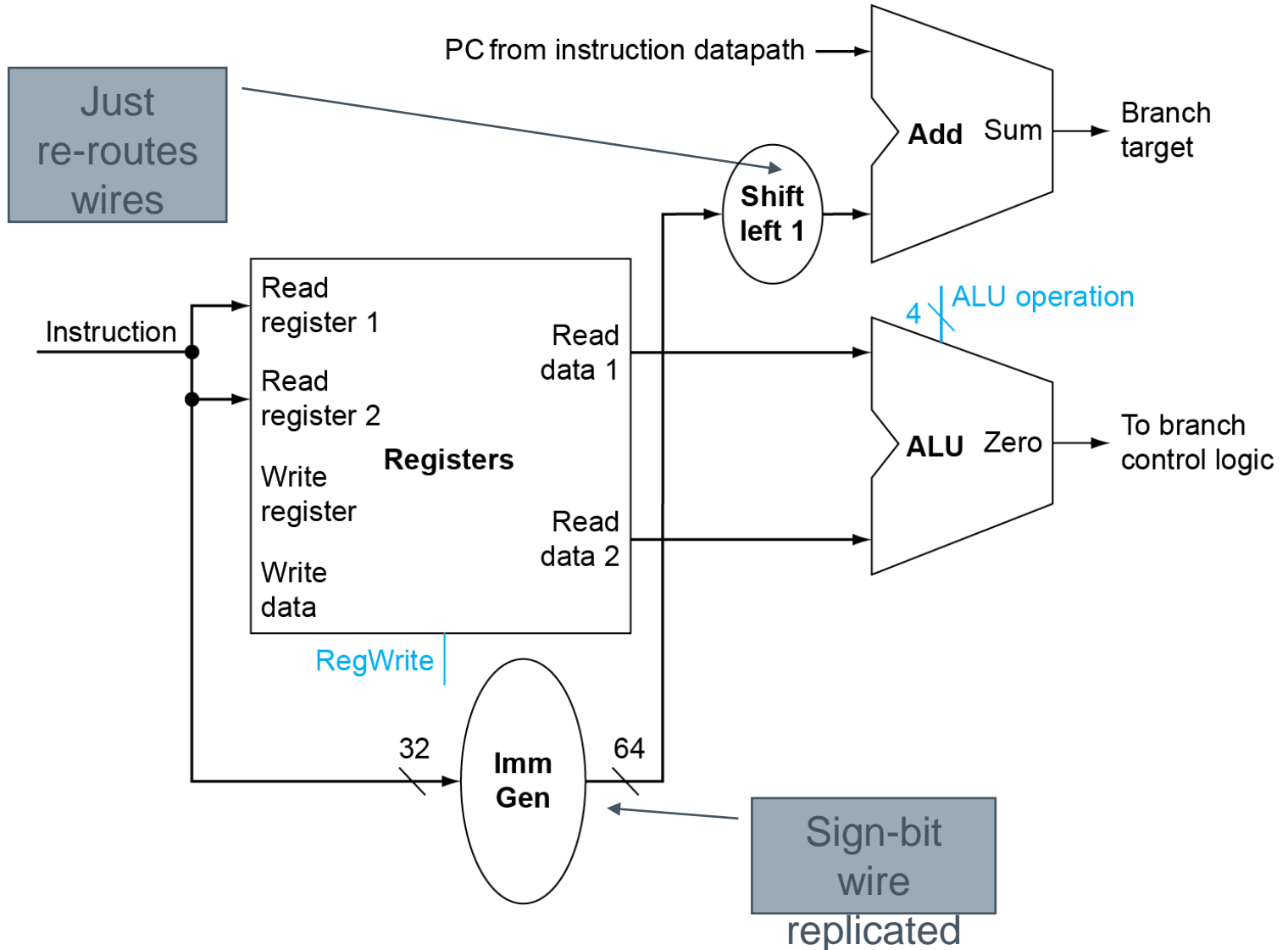


- › Read register operands
- › Compare operands
  - Use ALU, subtract and check Zero output
- › Calculate target address
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value

Name (Bit position)	Fields					
	31:25	24:20	19:15	14:12	11:7	6:0
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

# Branch Instructions

beq rs1, rs2, L1



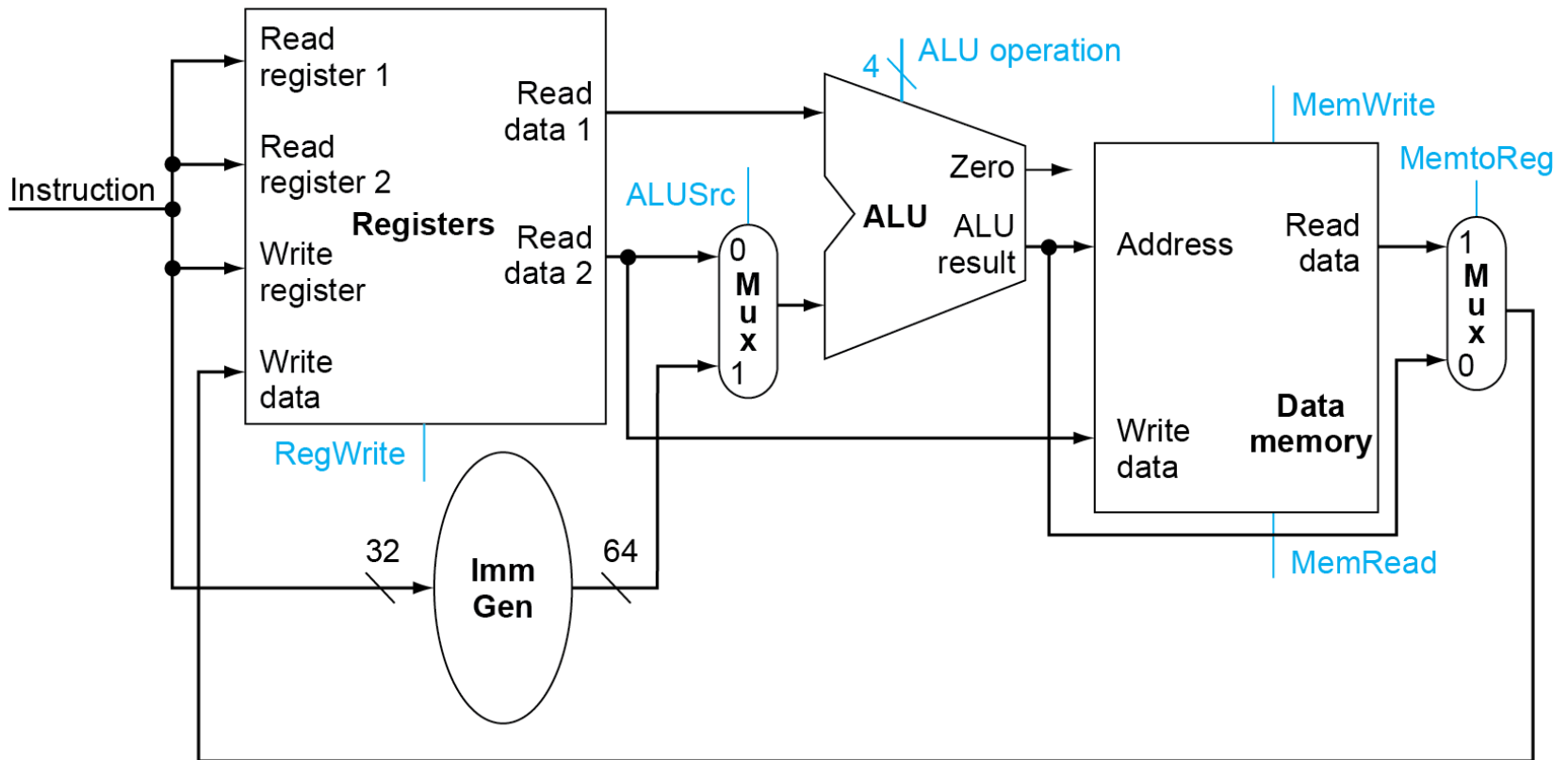
# Composing the Elements

---

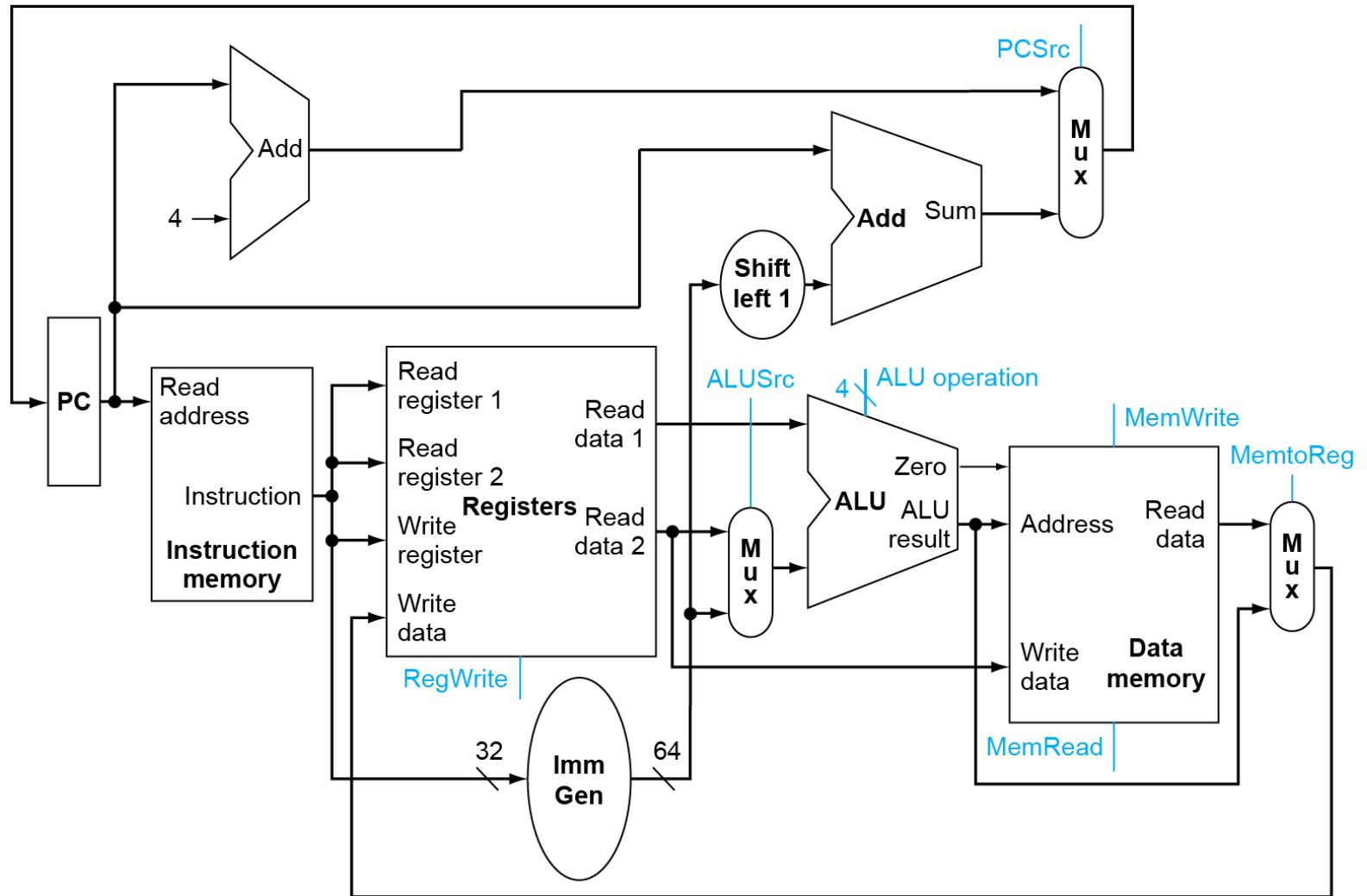


- › First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- › Use multiplexers where alternate data sources are used for different instructions

# R-Type/Load/Store Datapath



# Full Datapath



# ALU Control



- › ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

# ALU Control



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

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

**FIGURE 4.12** How the ALU control bits are set depends on the ALUOp control bits and the different opcodes for the R-type instruction. The instruction, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the funct7 or funct3 fields; in this case, we say that we “don’t care” about the value of the opcode, and the bits are shown as Xs. When the ALUOp value is 10, then the funct7 and funct3 fields are used to set the ALU control input. See [Appendix A](#).

# The Main Control Unit



## › Control signals derived from instruction

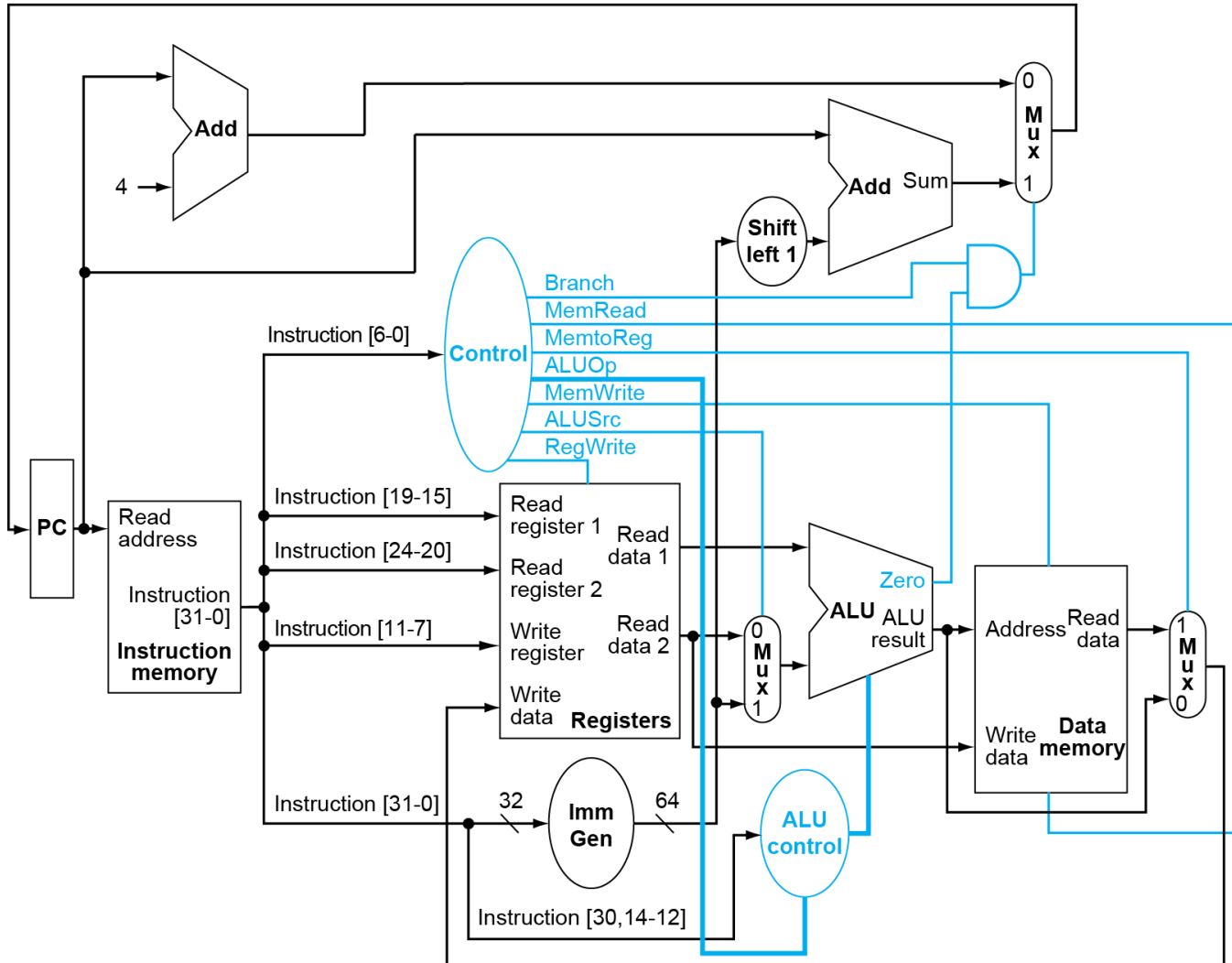
Name (Bit position)	Fields					
	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

ALUOp		Funct7 field							Funct3 field			Operation
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

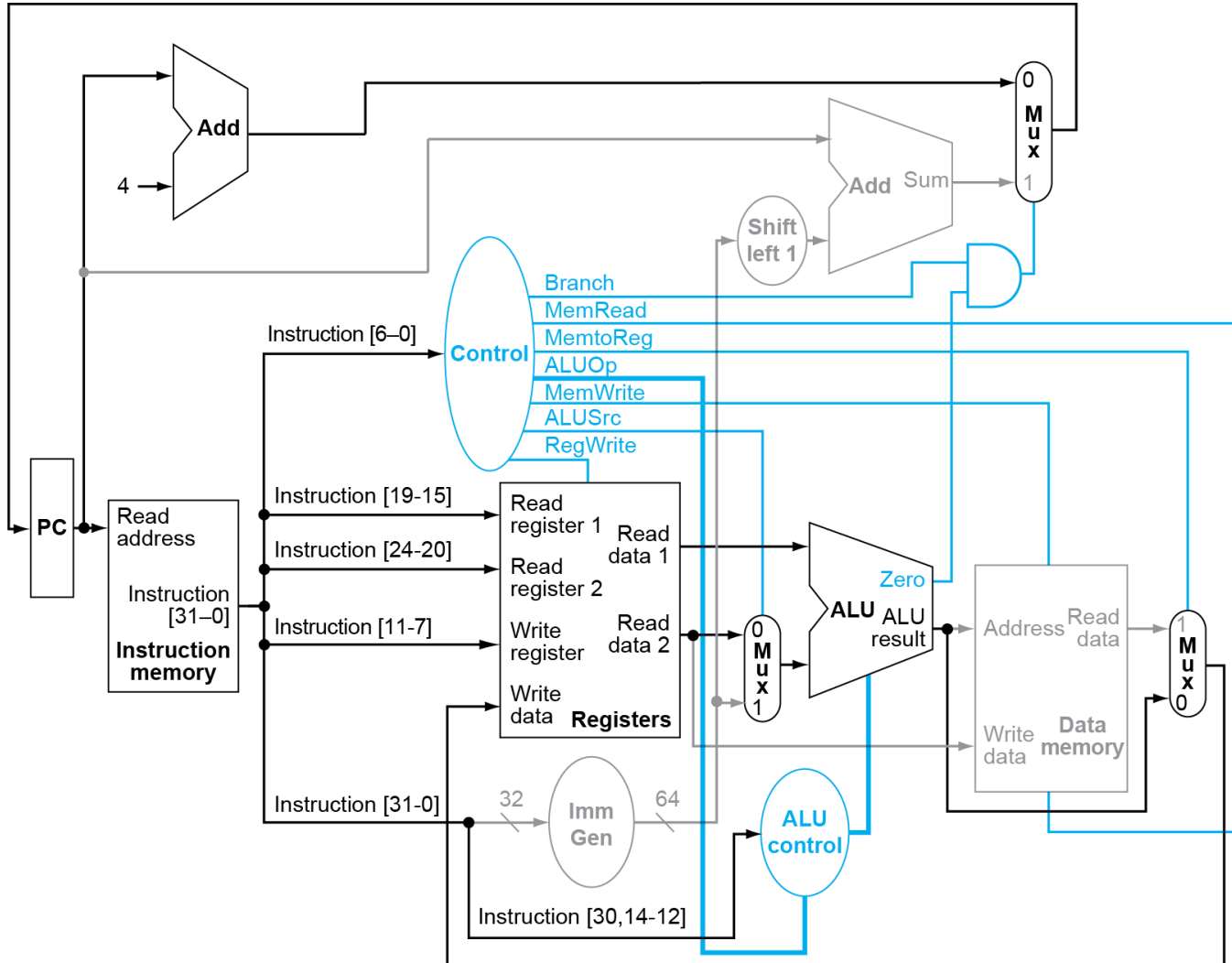
**FIGURE 4.13 The truth table for the 4 ALU control bits (called Operation).** The inputs are the ALUOp and funct fields. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. While we show all 10 bits of funct fields, note that the only bits with different values for the four R-format instructions are bits 30, 14, 13, and 12. Thus, we only need these four funct field bits as input for ALU control instead of all 10.



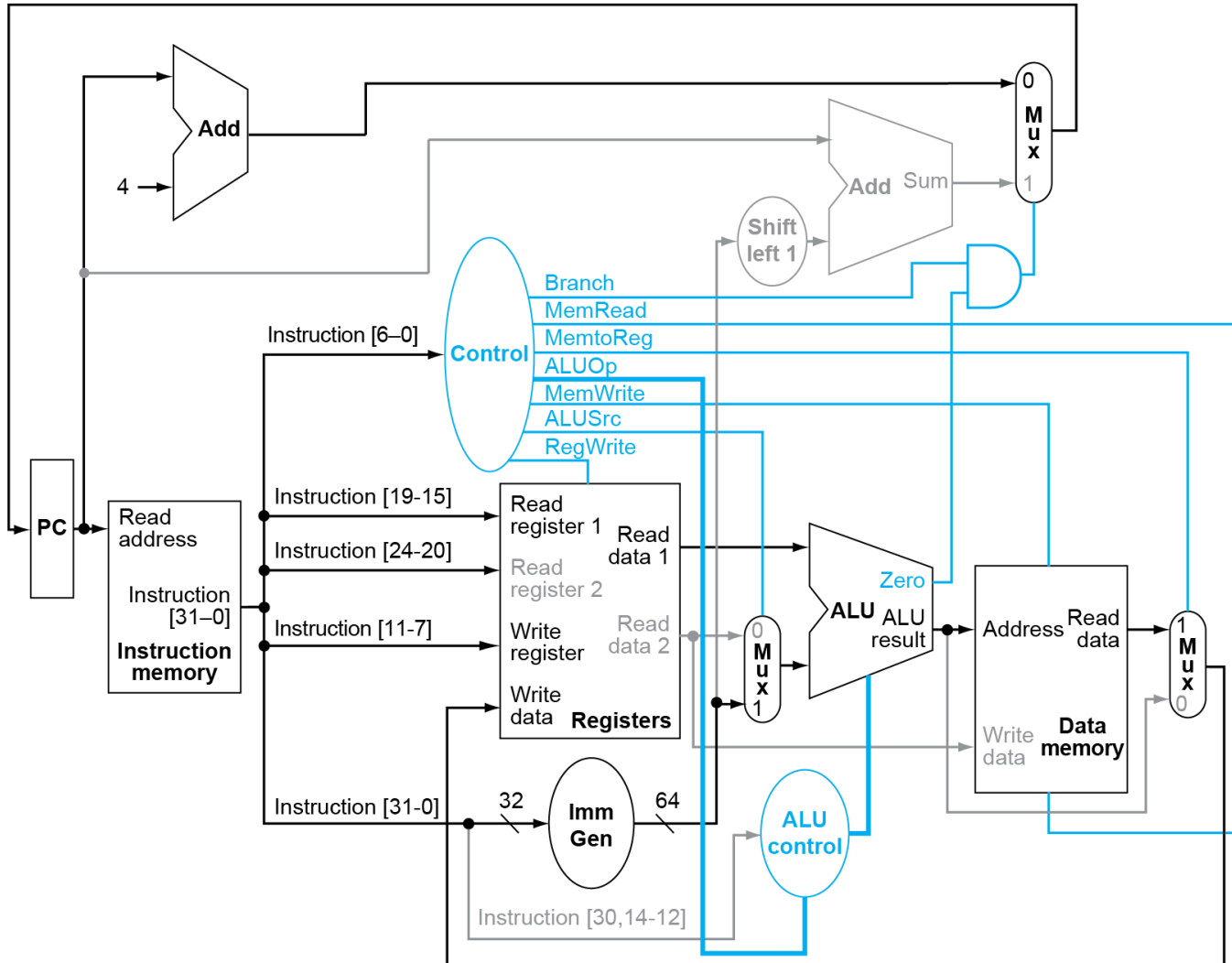
# Datapath With Control



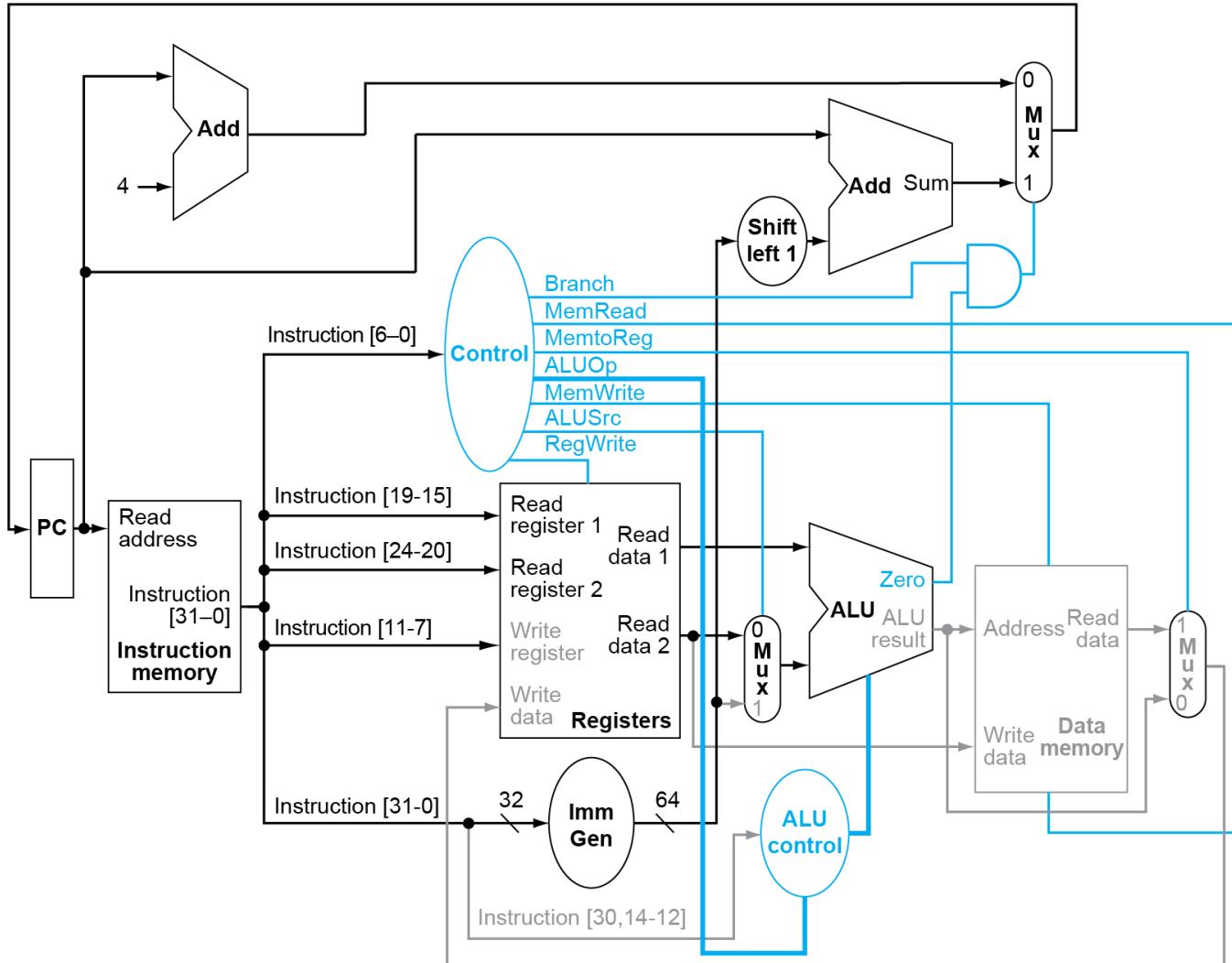
# R-Type Instruction



# Load Instruction



# BEQ Instruction



# Performance Issues

---

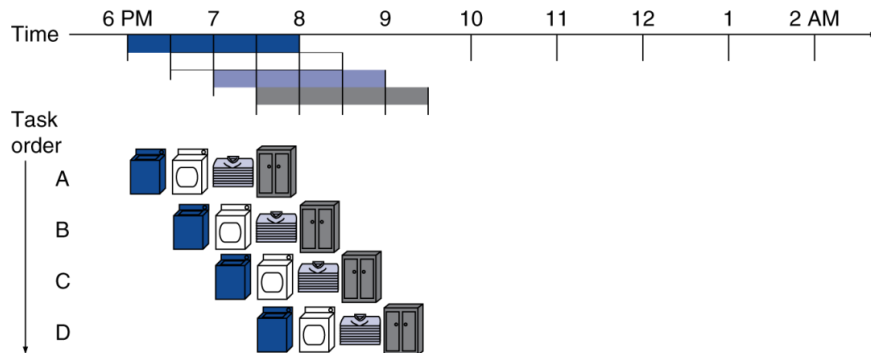
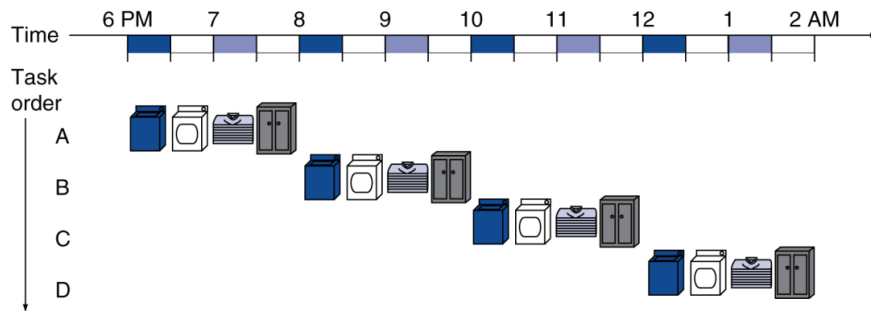


- › Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- › Not feasible to vary period for different instructions
- › Violates design principle
  - Making the common case fast
- › We will improve performance by pipelining

# Pipelining Analogy



- › Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:

- Speedup  
=  $8/3.5 = 2.3$

- Non-stop:

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

# 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
- › Compare pipelined datapath with single-cycle datapath

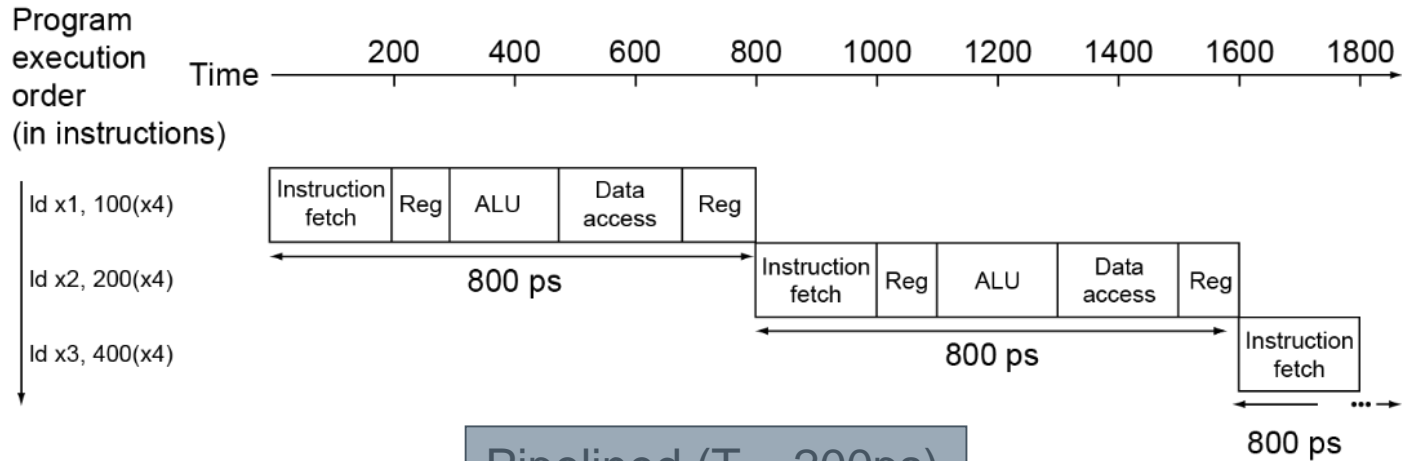
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps



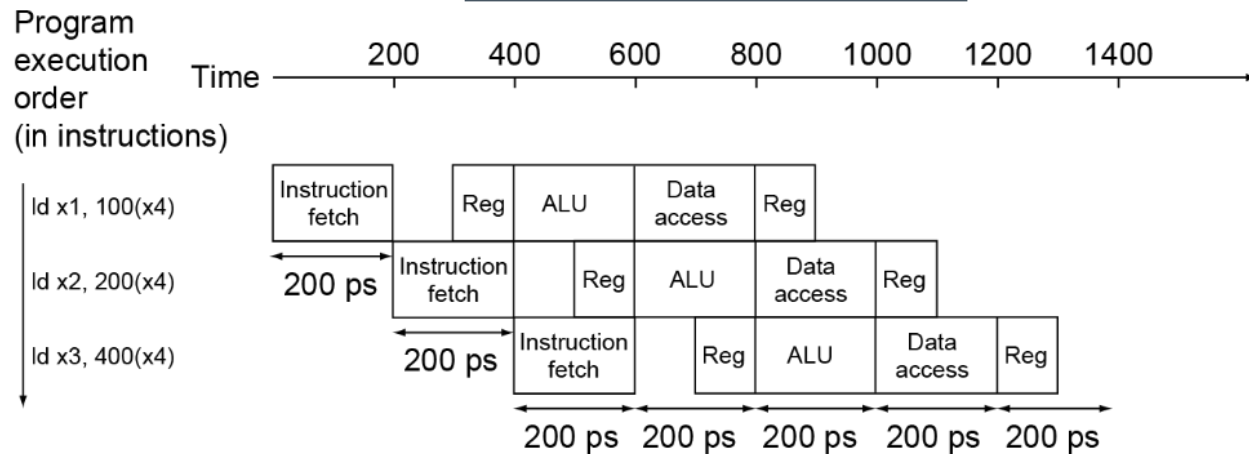
# Pipeline Performance



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
  - Time between instructions<sub>pipelined</sub>  
=  $\frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$
- › If not balanced, speedup is less
- › Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

# Pipelining and ISA Design

---



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

# 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

---



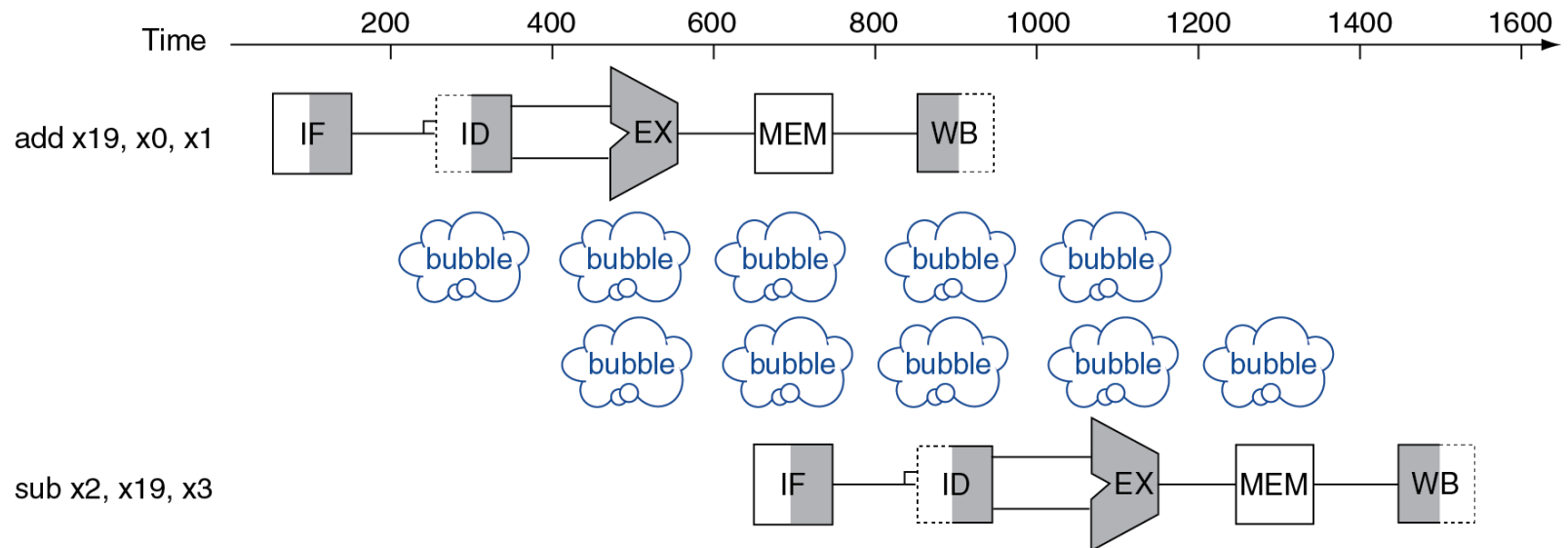
- › Conflict for use of a resource
- › In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - › Would cause a pipeline “bubble”
- › Hence, pipelined datapaths require separate instruction/data memories
  - Or separate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

# Data Hazards



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

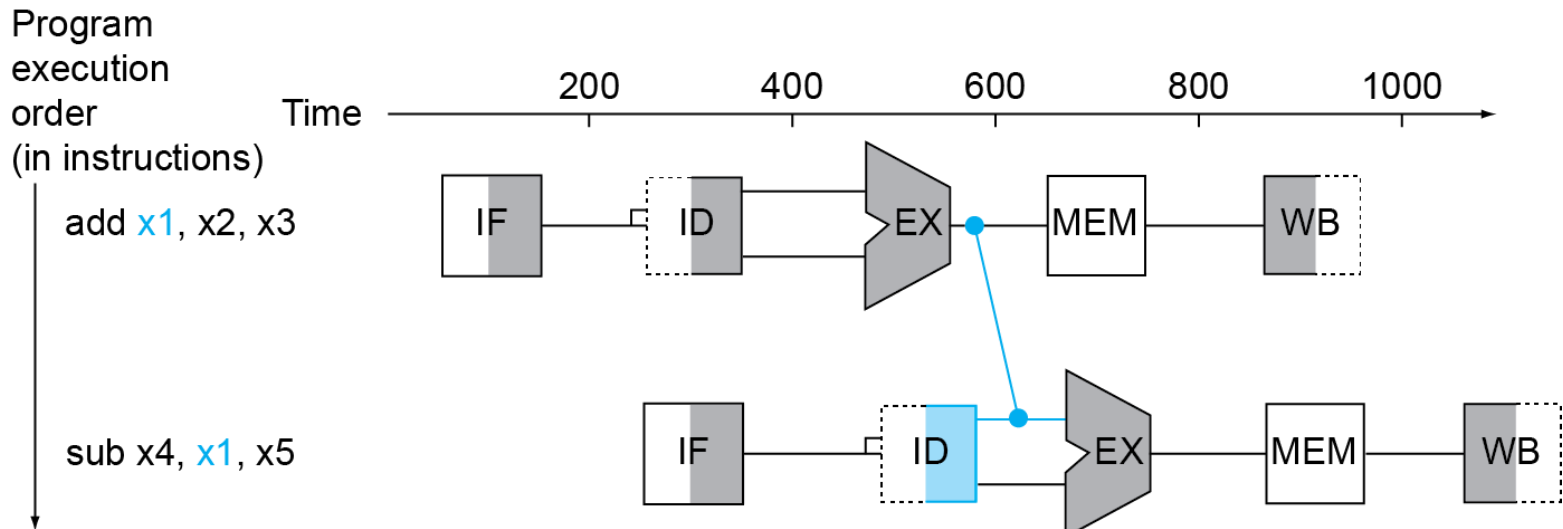
– add        **x19**, x0, x1  
  sub        x2, **x19**, x3



# Forwarding (aka Bypassing)



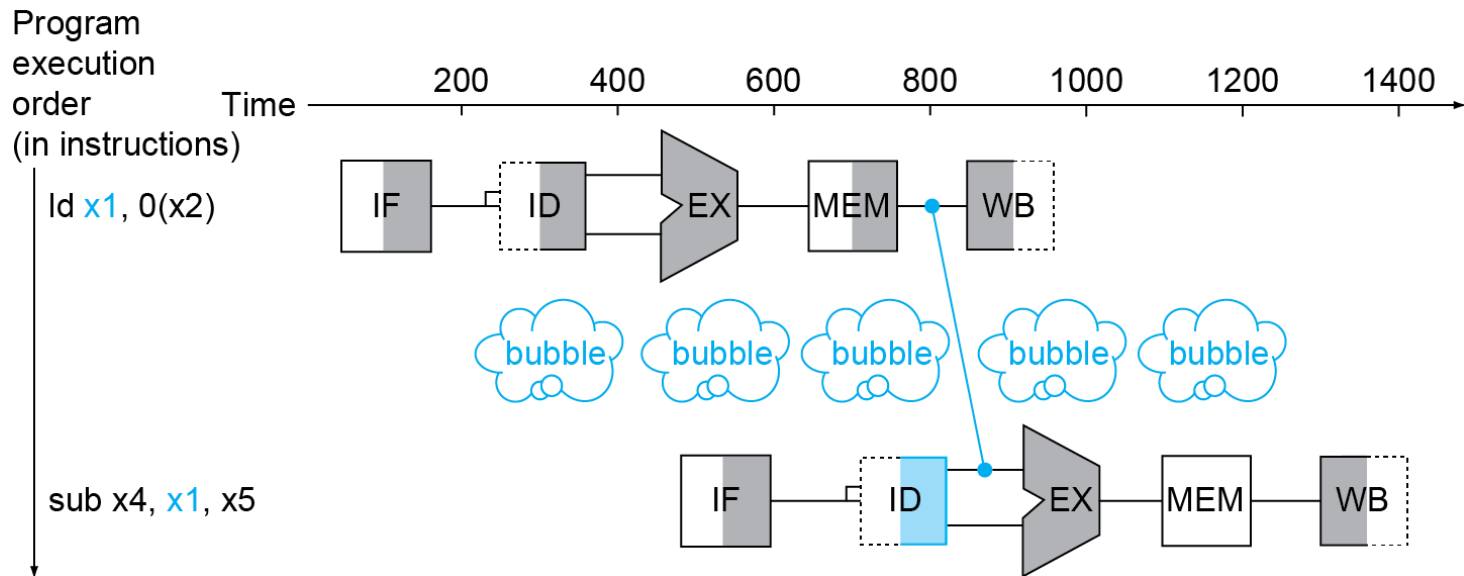
- › Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



# Load-Use Data Hazard



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

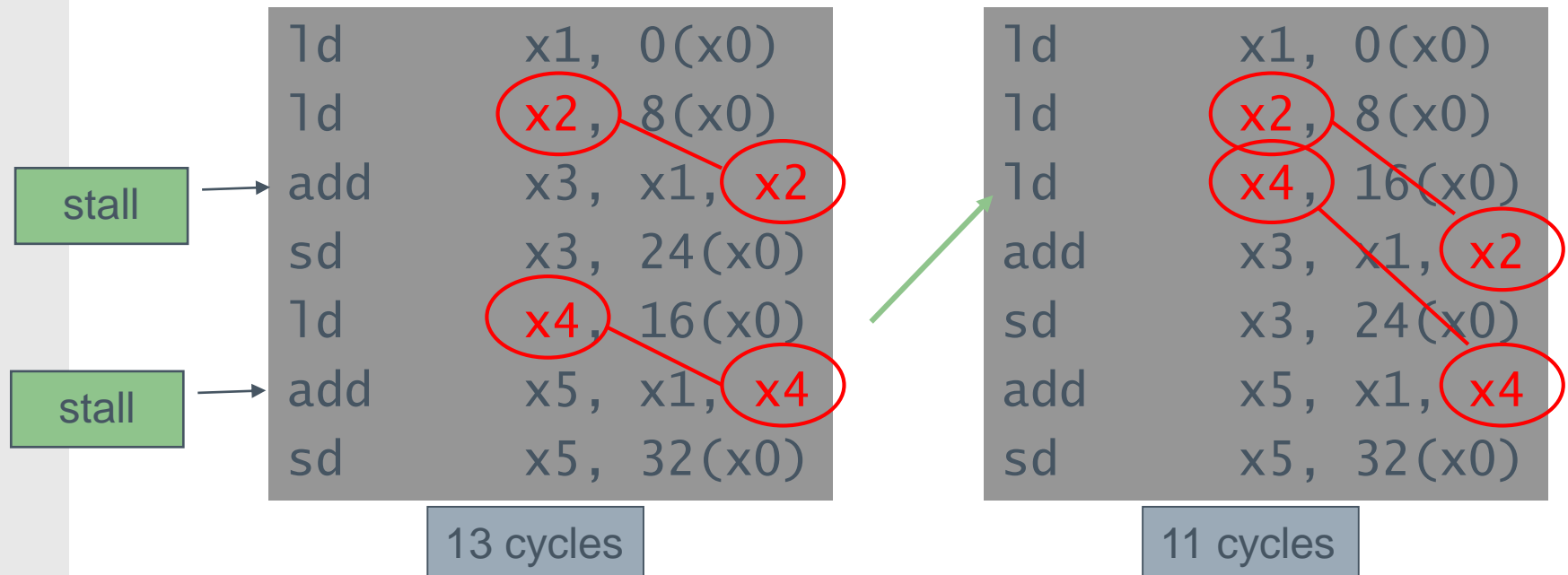




# Code Scheduling to Avoid Stalls



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



# Control Hazards

---

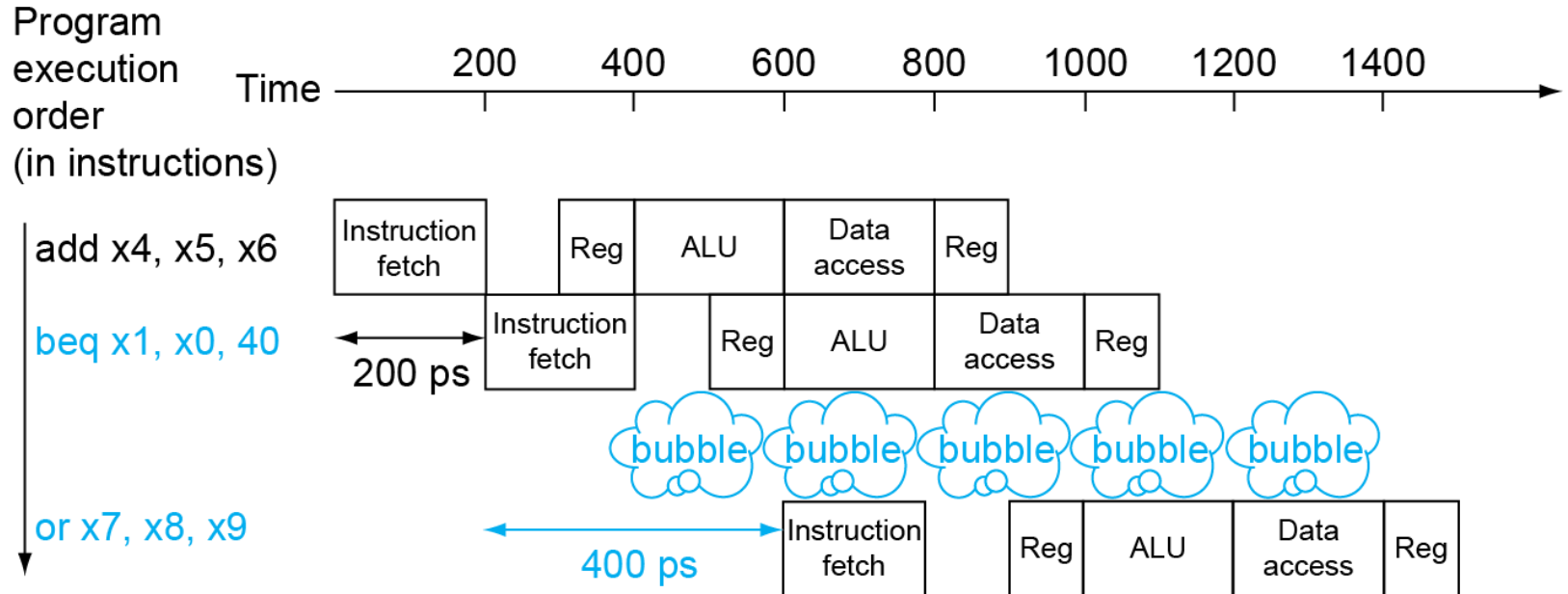


- › Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - › Still working on ID stage of branch
  
- › In RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware 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 RISC-V pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

# 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

---



- › 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

# RISC-V Pipelined Datapath



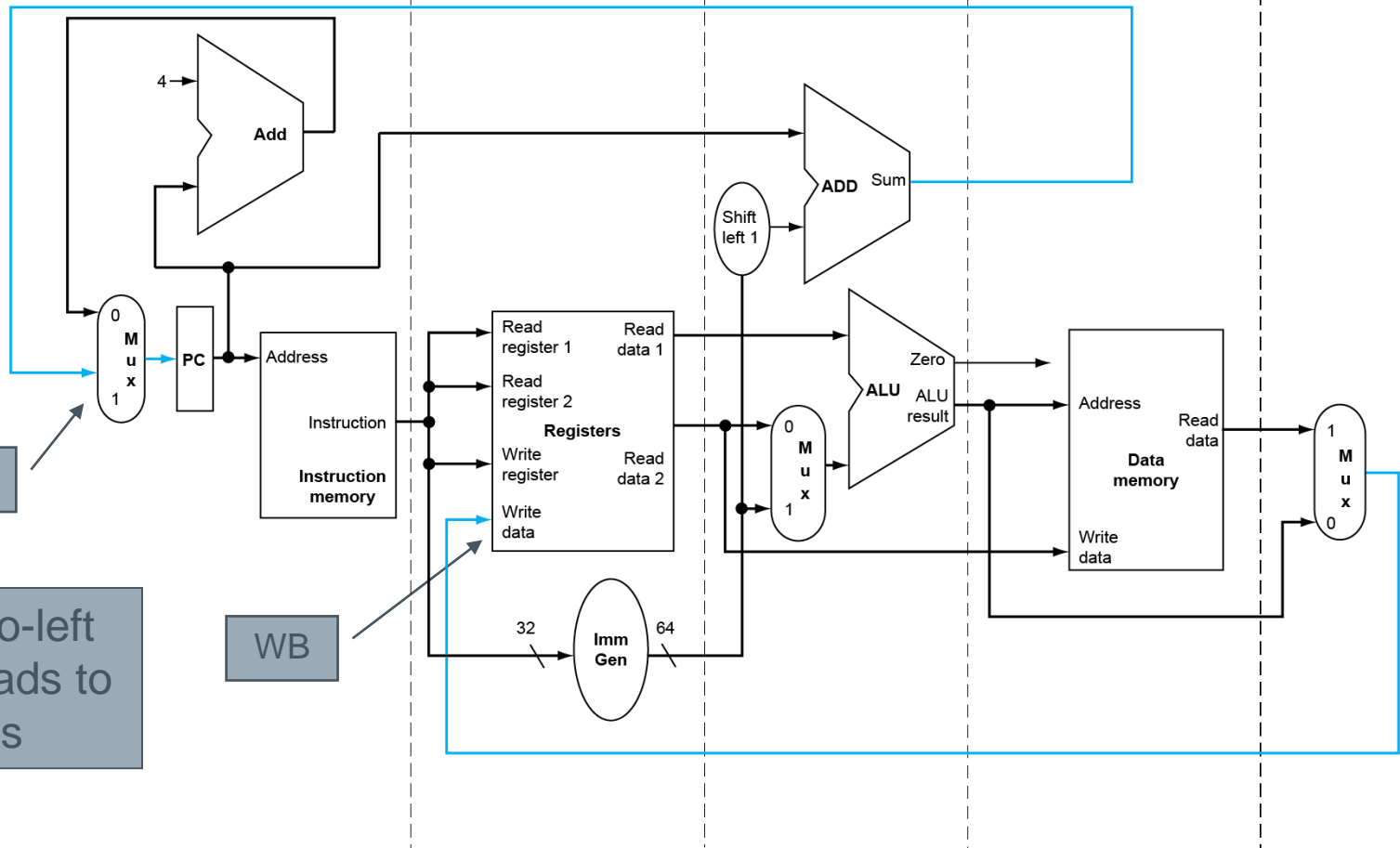
IF: Instruction fetch

ID: Instruction decode/  
register file read

EX: Execute/  
address calculation

MEM: Memory access

WB: Write back



MEM

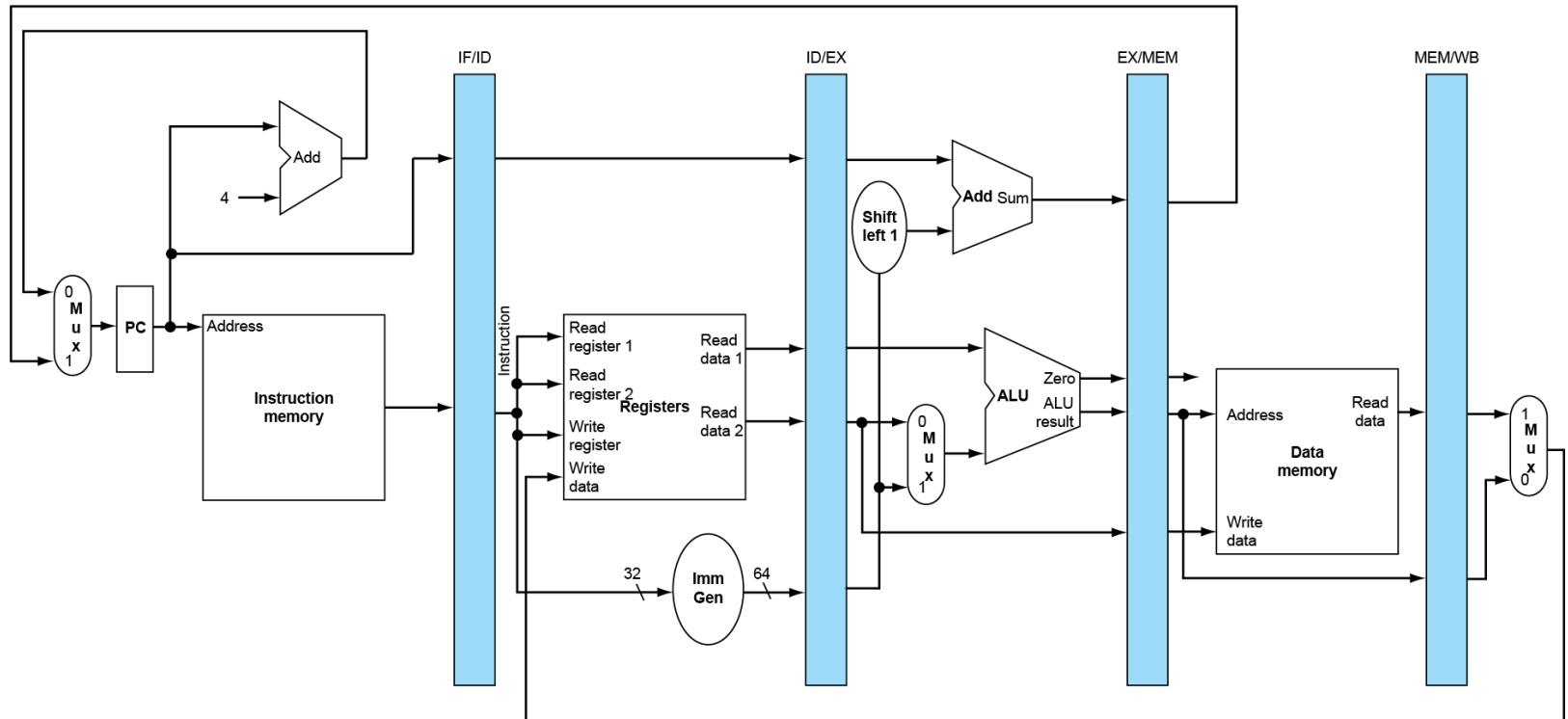
Right-to-left  
flow leads to  
hazards

WB

# Pipeline registers



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





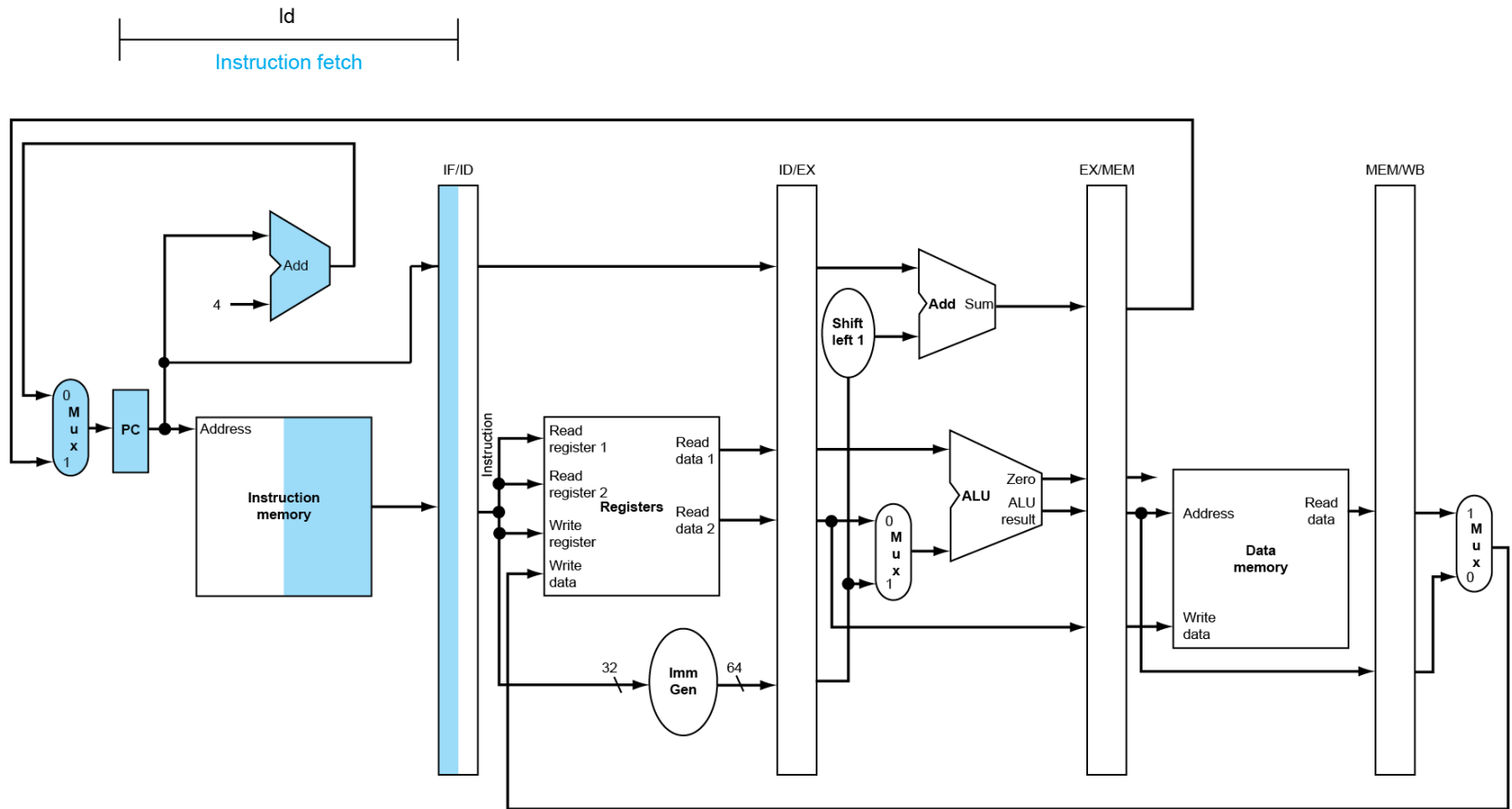
# Pipeline Operation

---

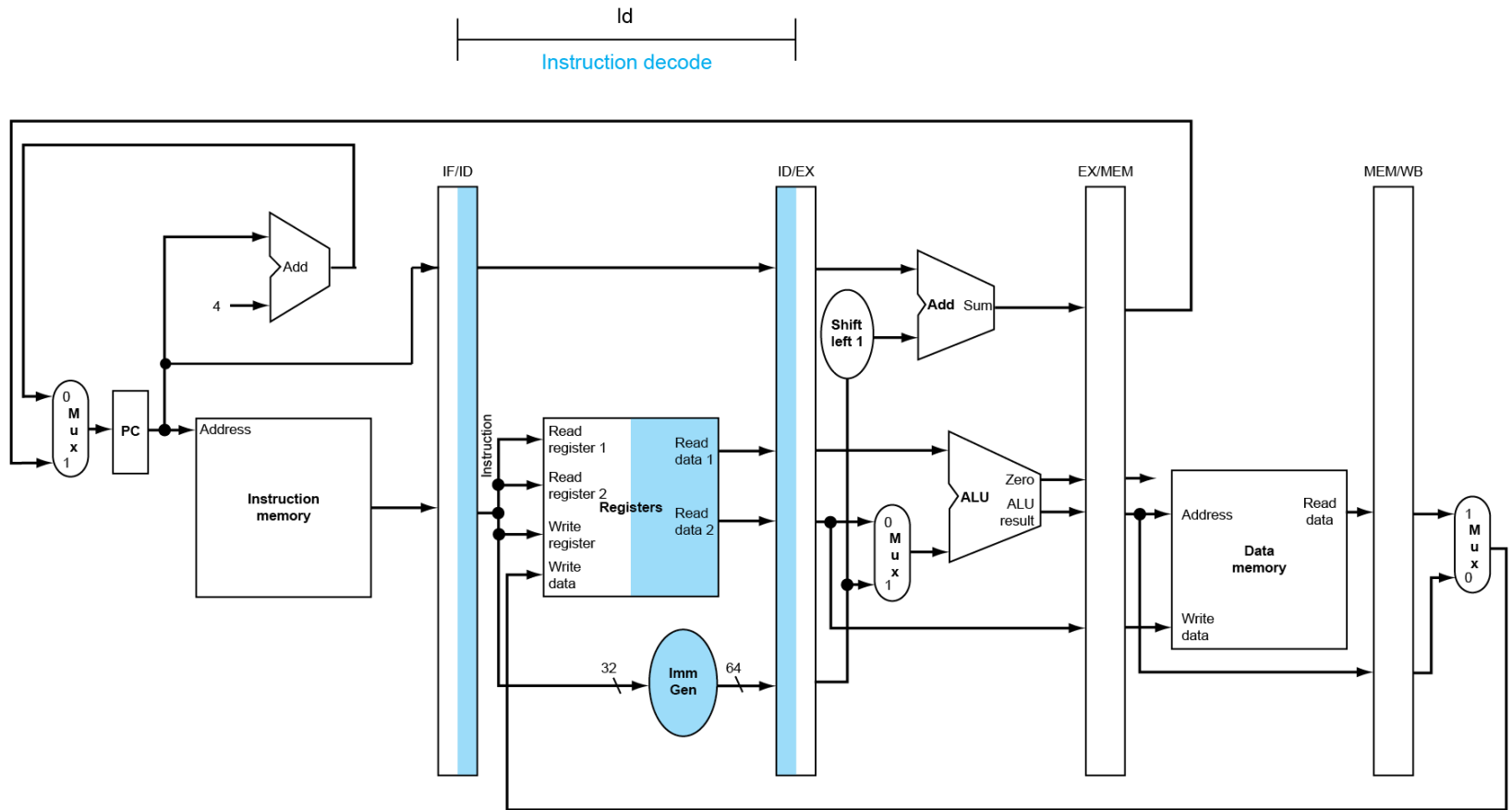


- › Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - › Shows pipeline usage in a single cycle
    - › Highlight resources used
  - c.f. “multi-clock-cycle” diagram
    - › Graph of operation over time
- › We’ll look at “single-clock-cycle”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

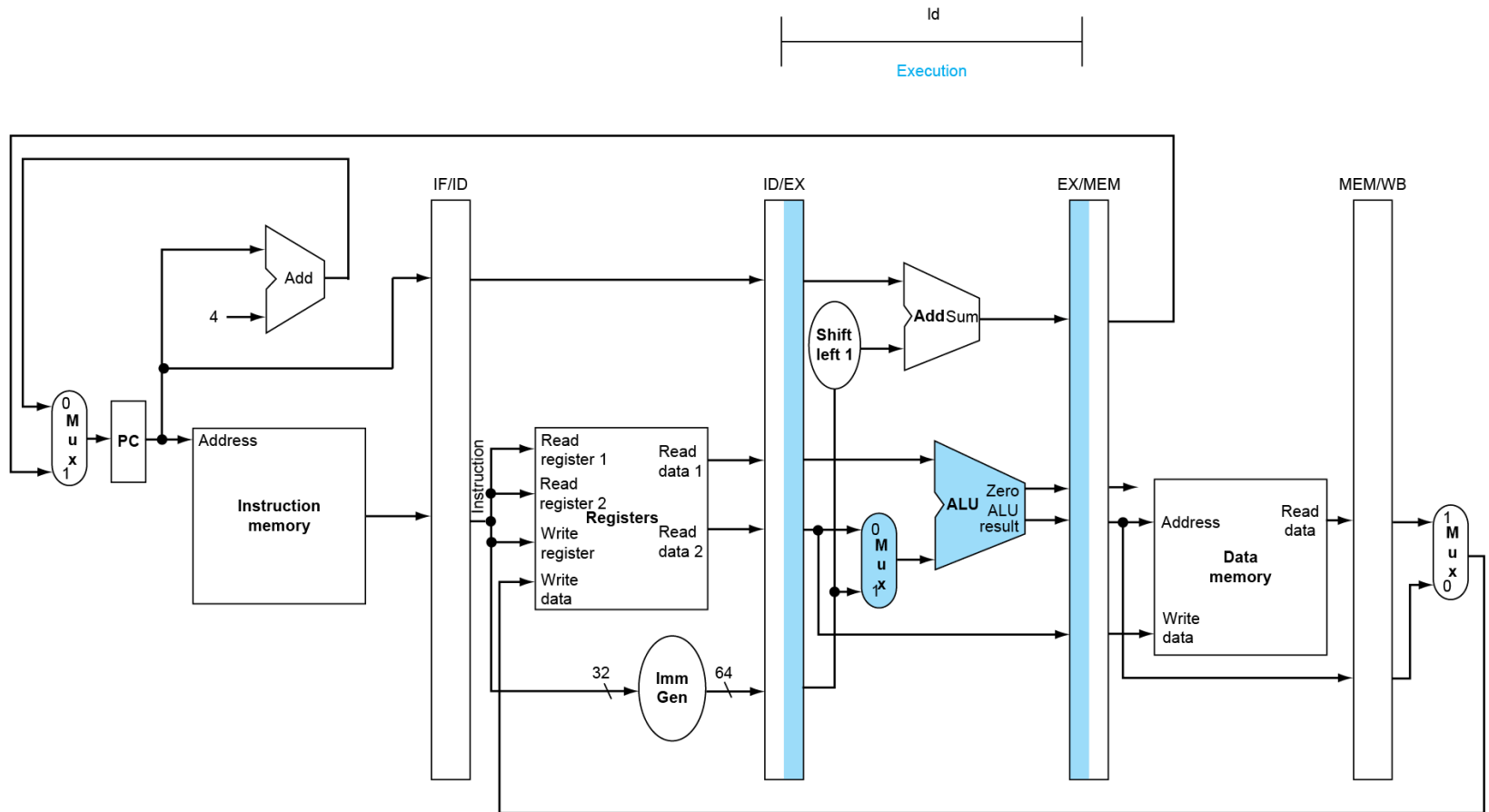
# IF for Load, Store, ...



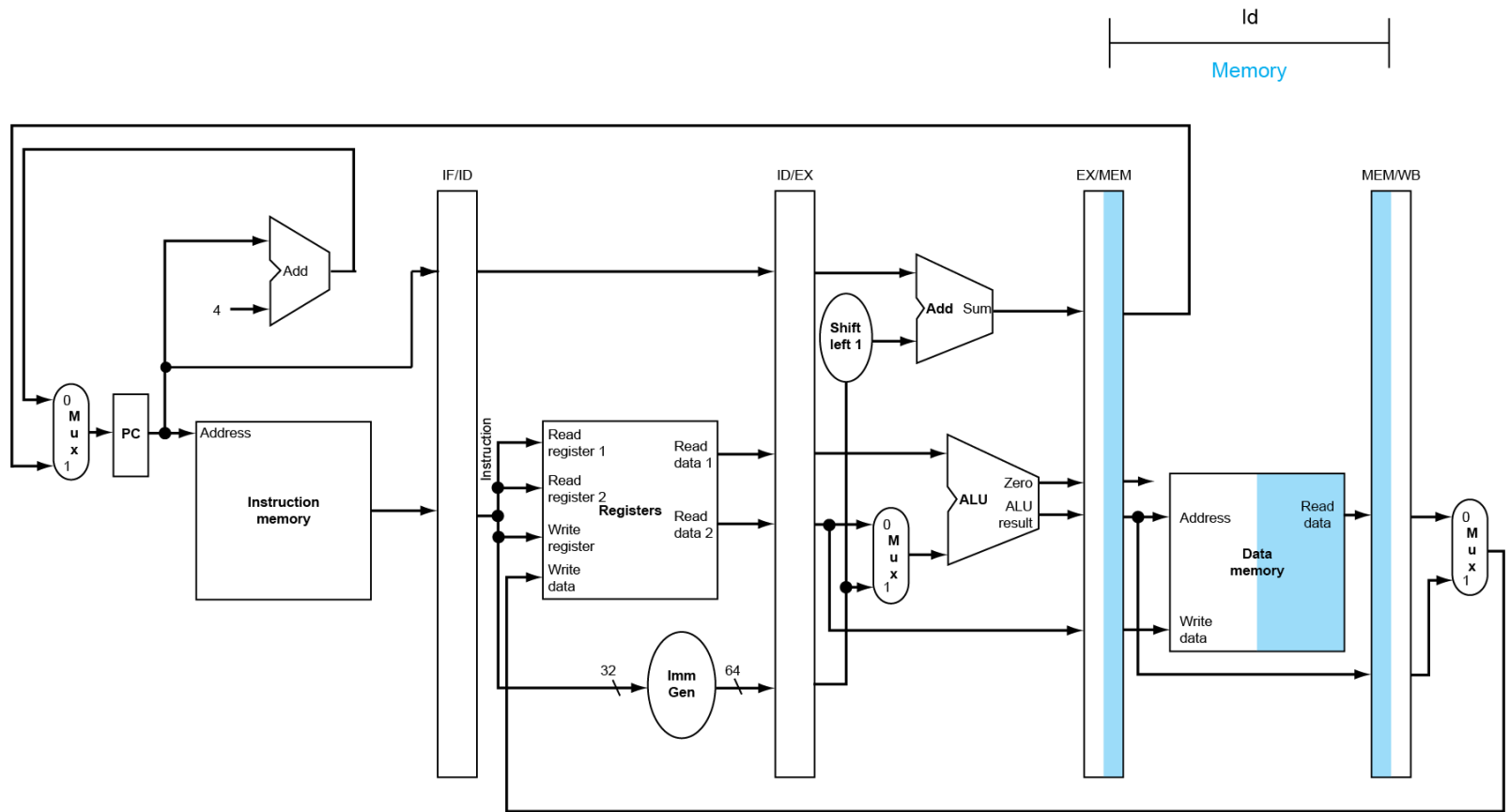
# ID for Load, Store, ...



# EX for Load



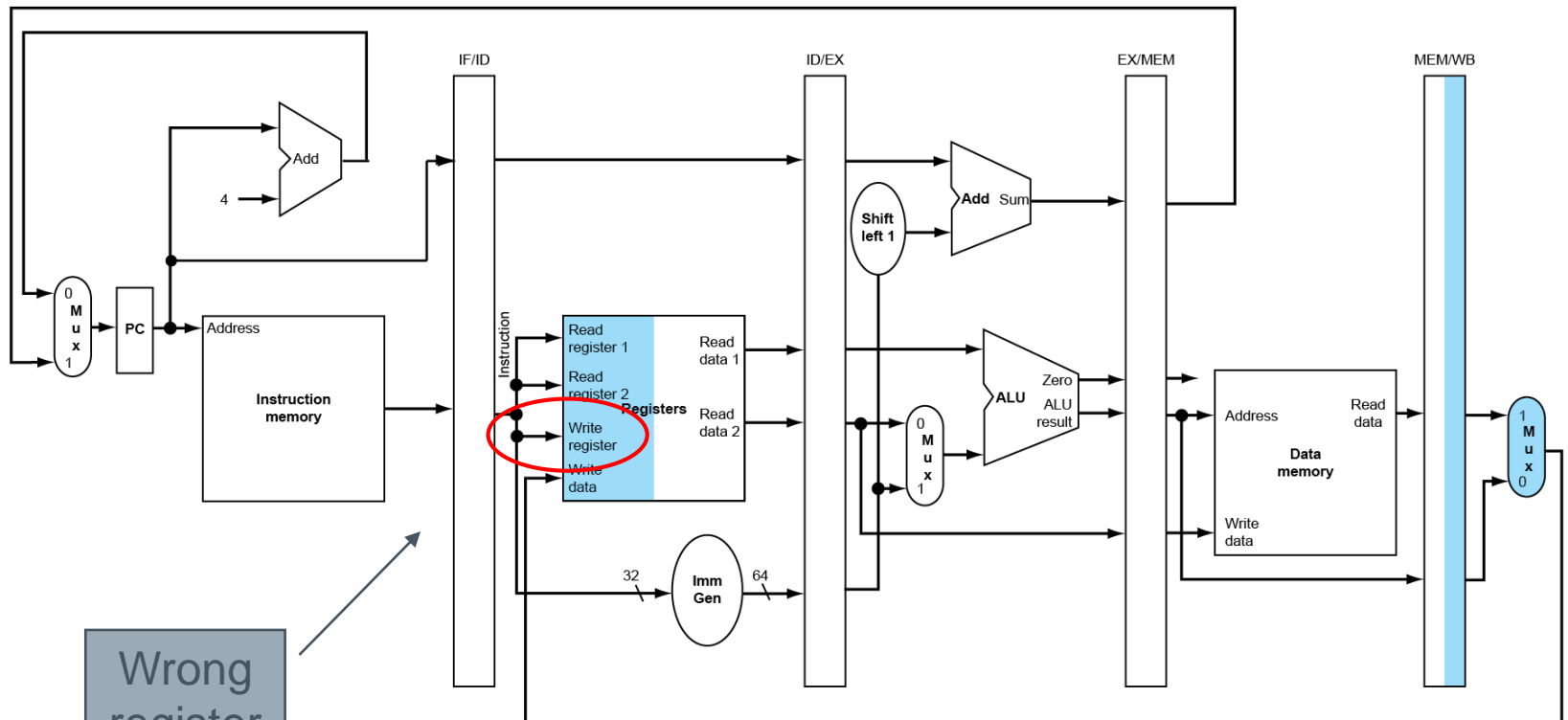
# MEM for Load



# WB for Load

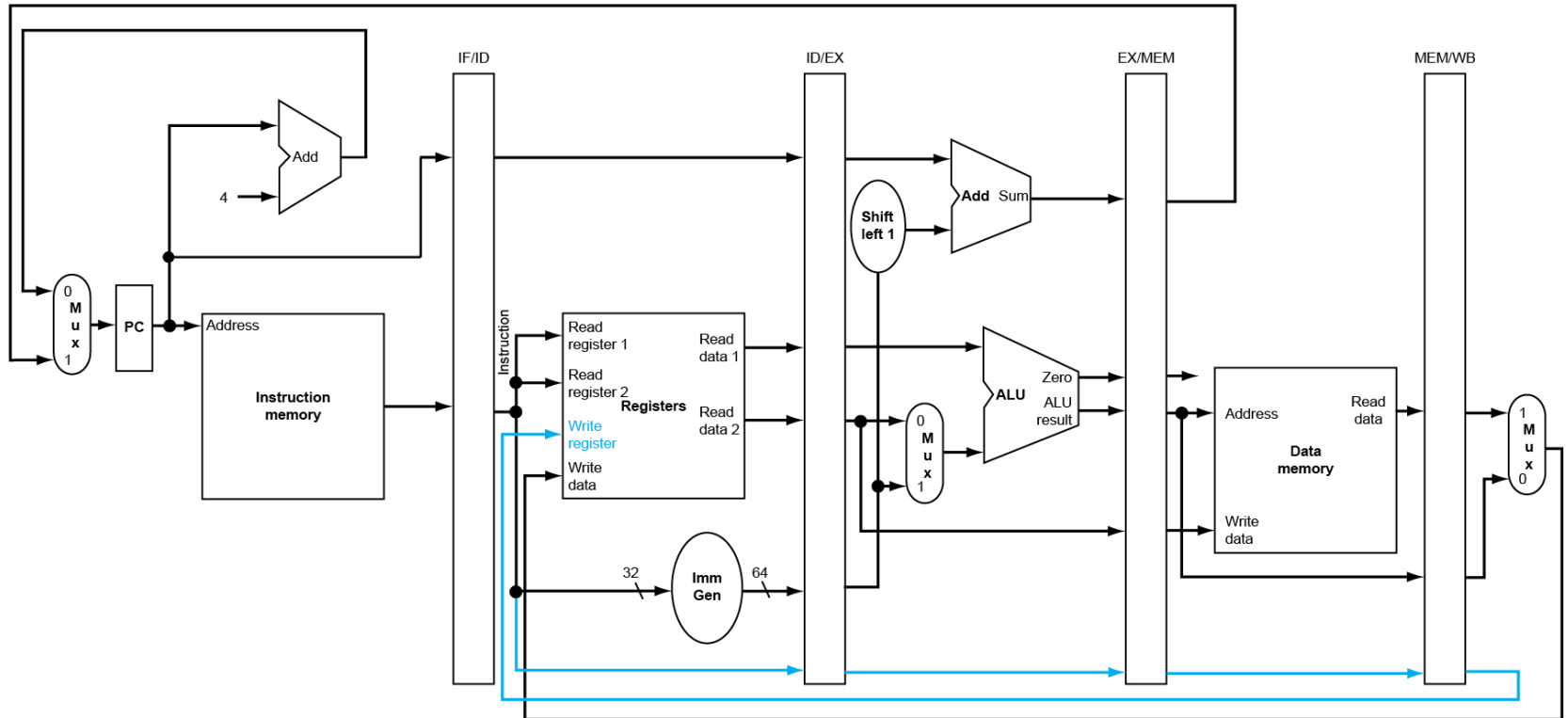


Id  
Write-back



Wrong register number

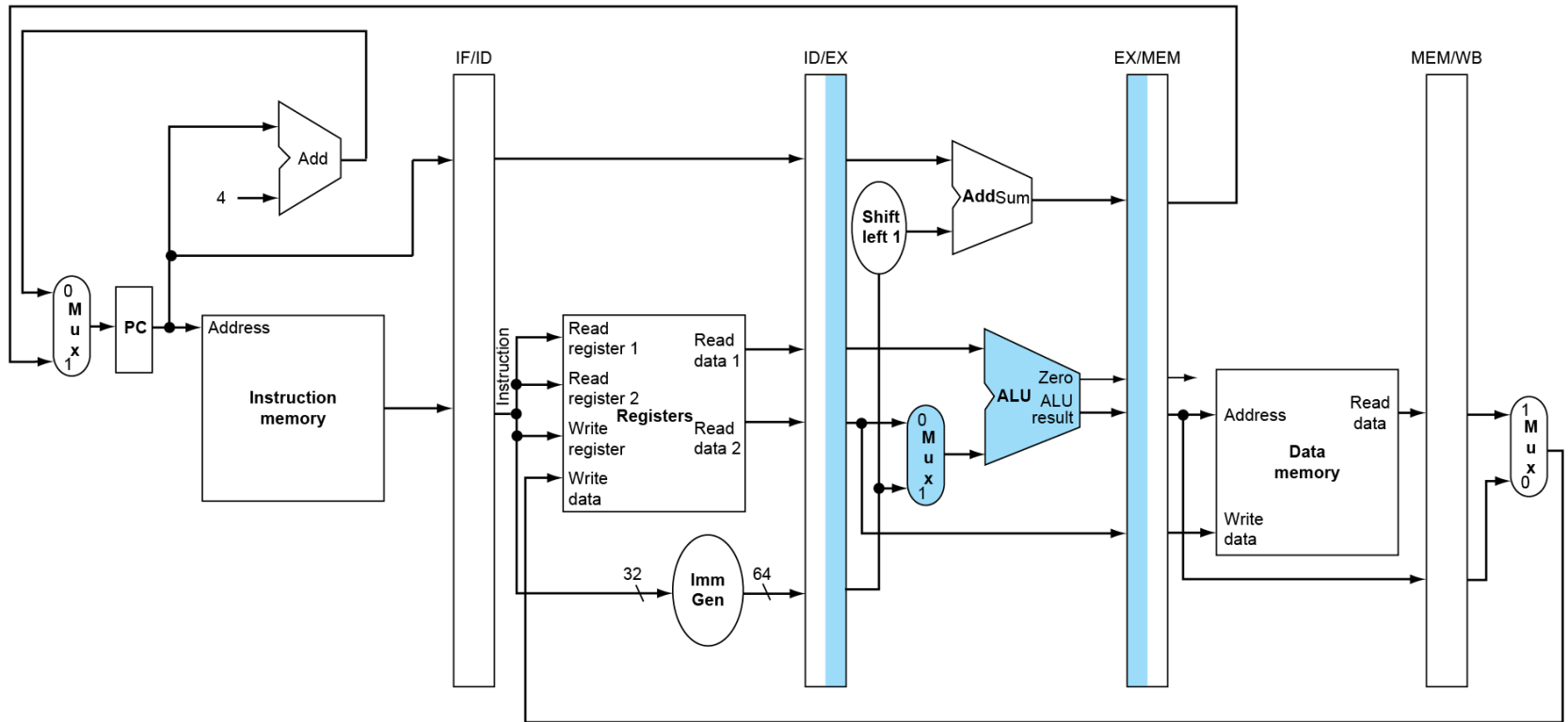
# Corrected Datapath for Load



# EX for Store

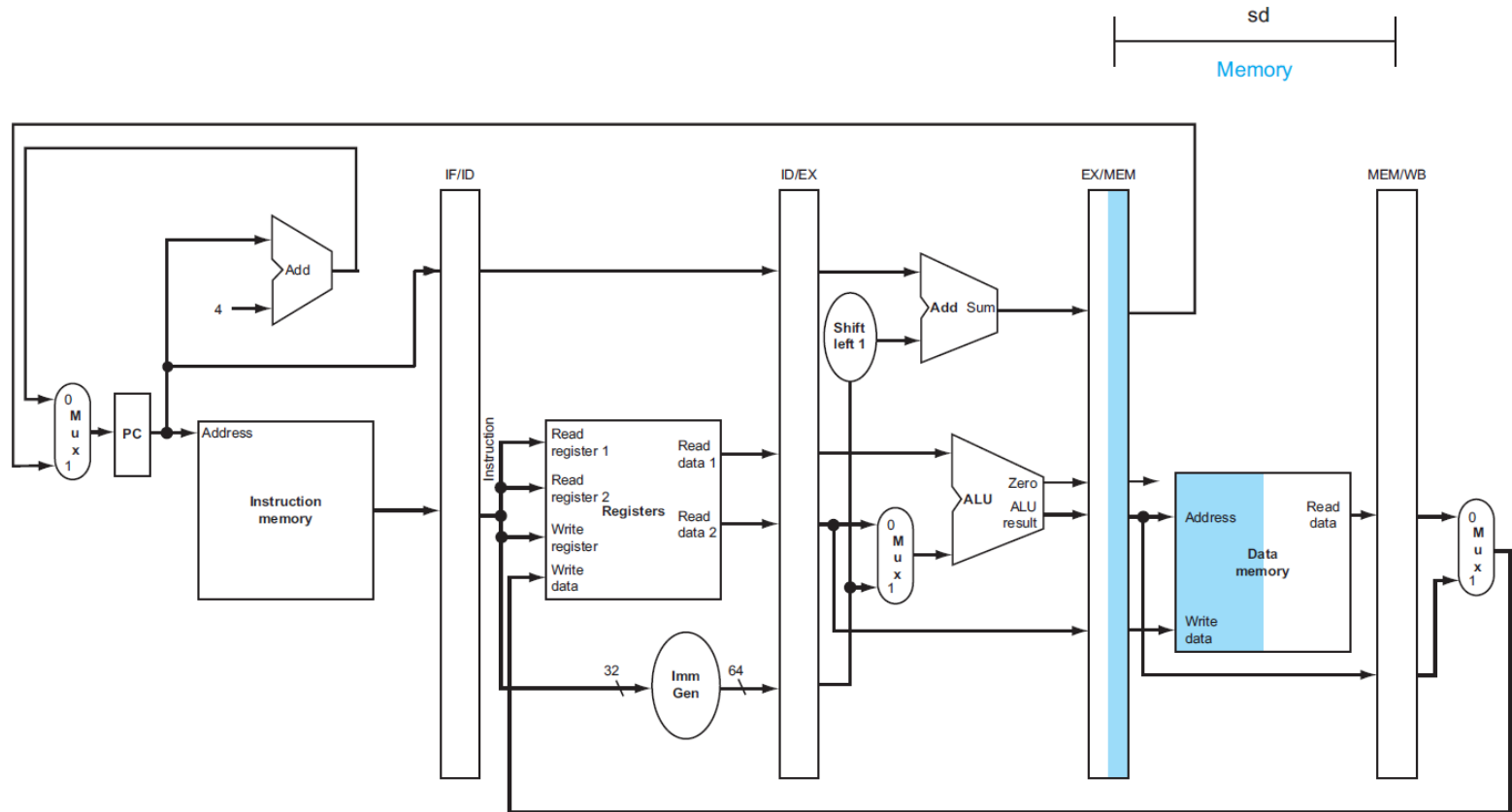


sd  
Execution





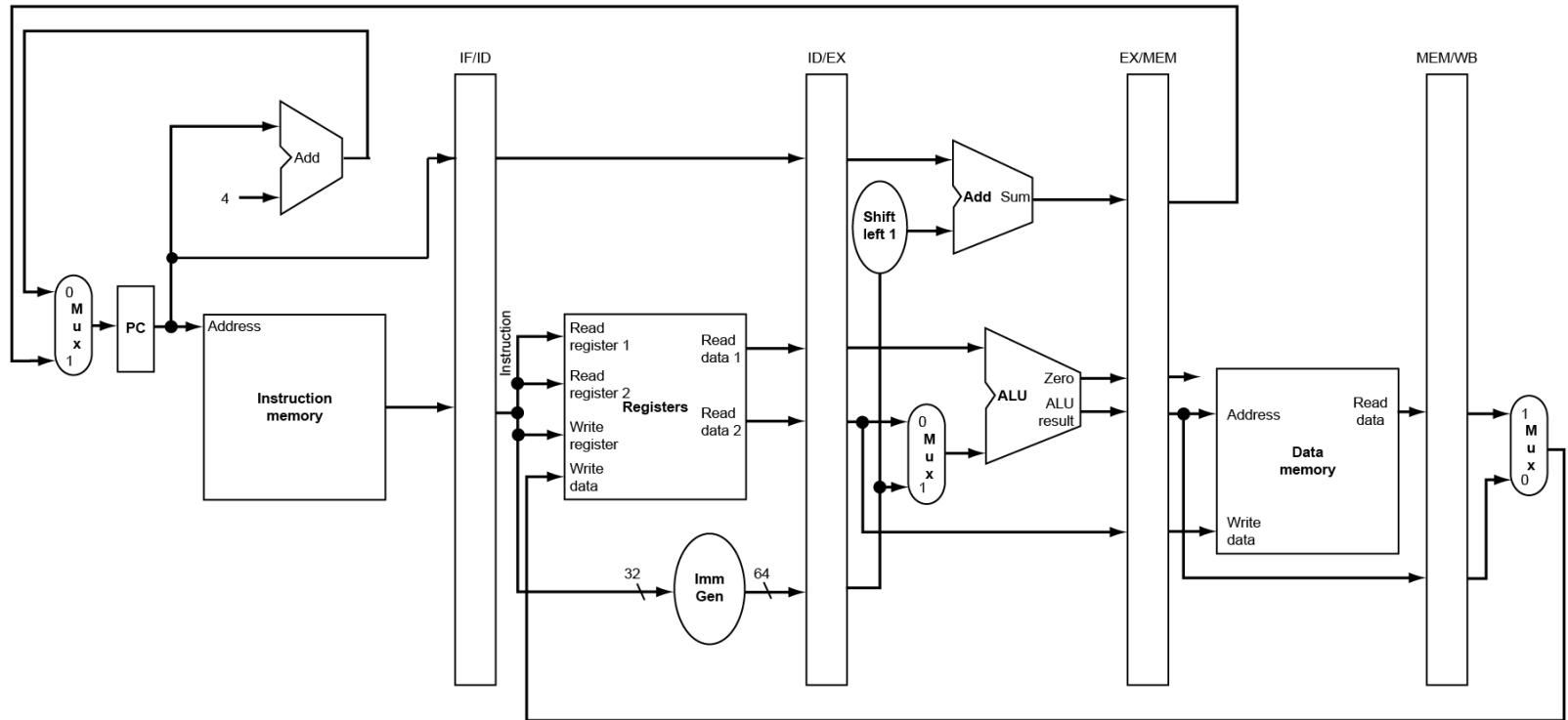
# MEM for Store



# WB for Store



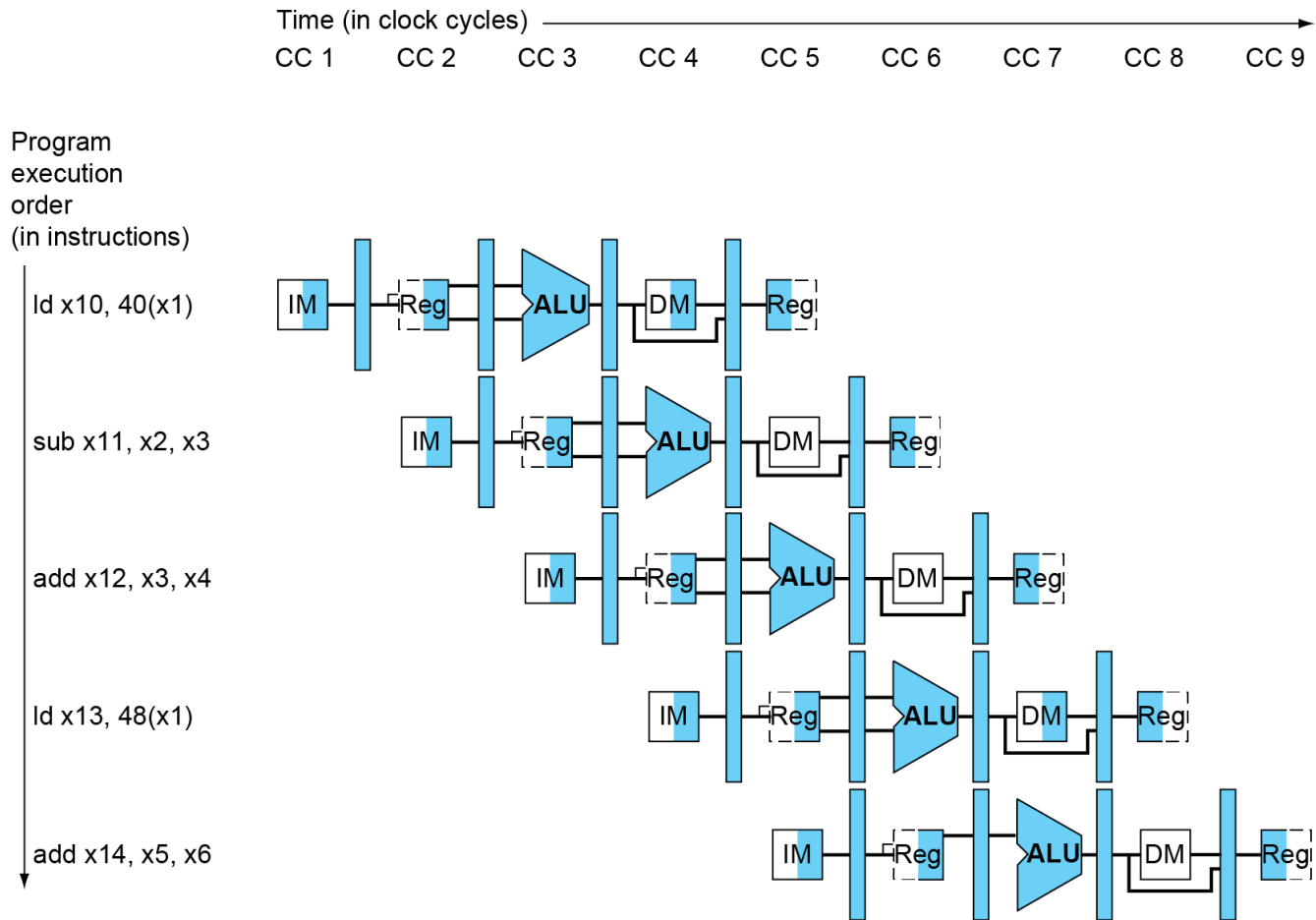
sd  
Write-back



# Multi-Cycle Pipeline Diagram



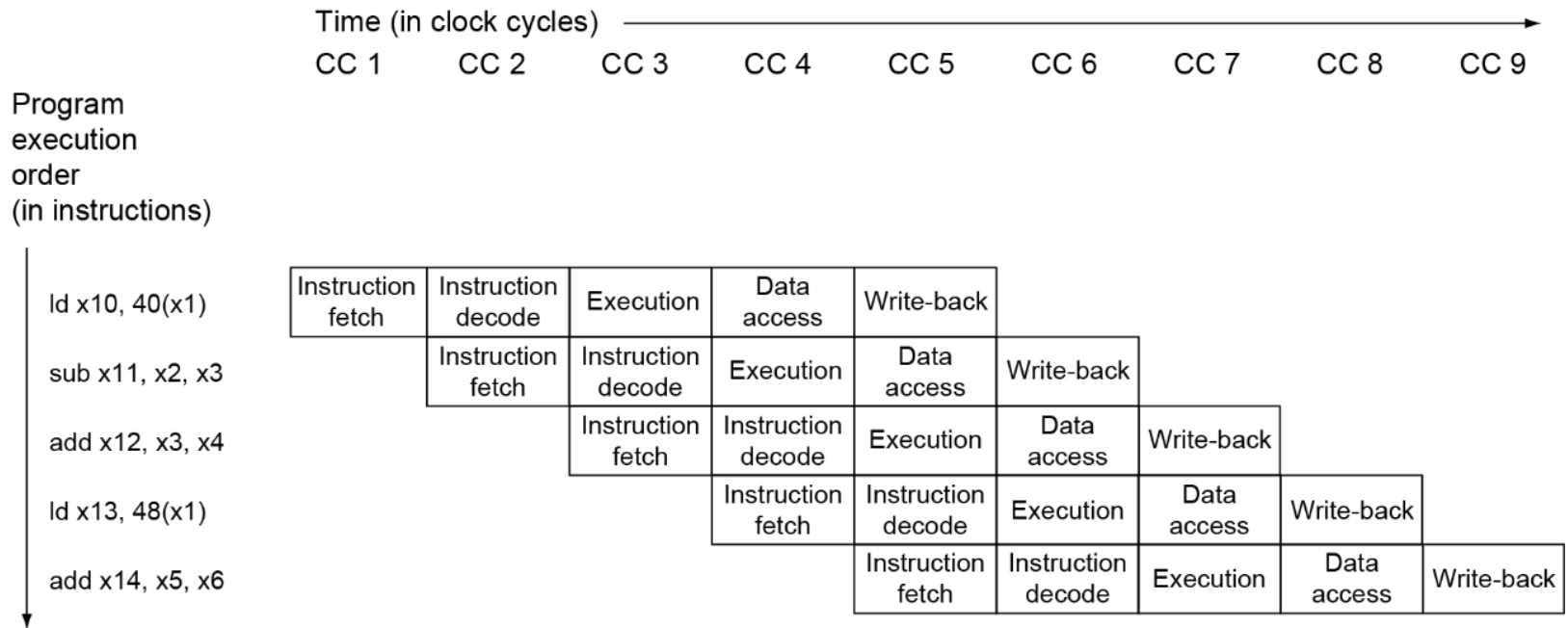
› Form showing resource usage



# Multi-Cycle Pipeline Diagram



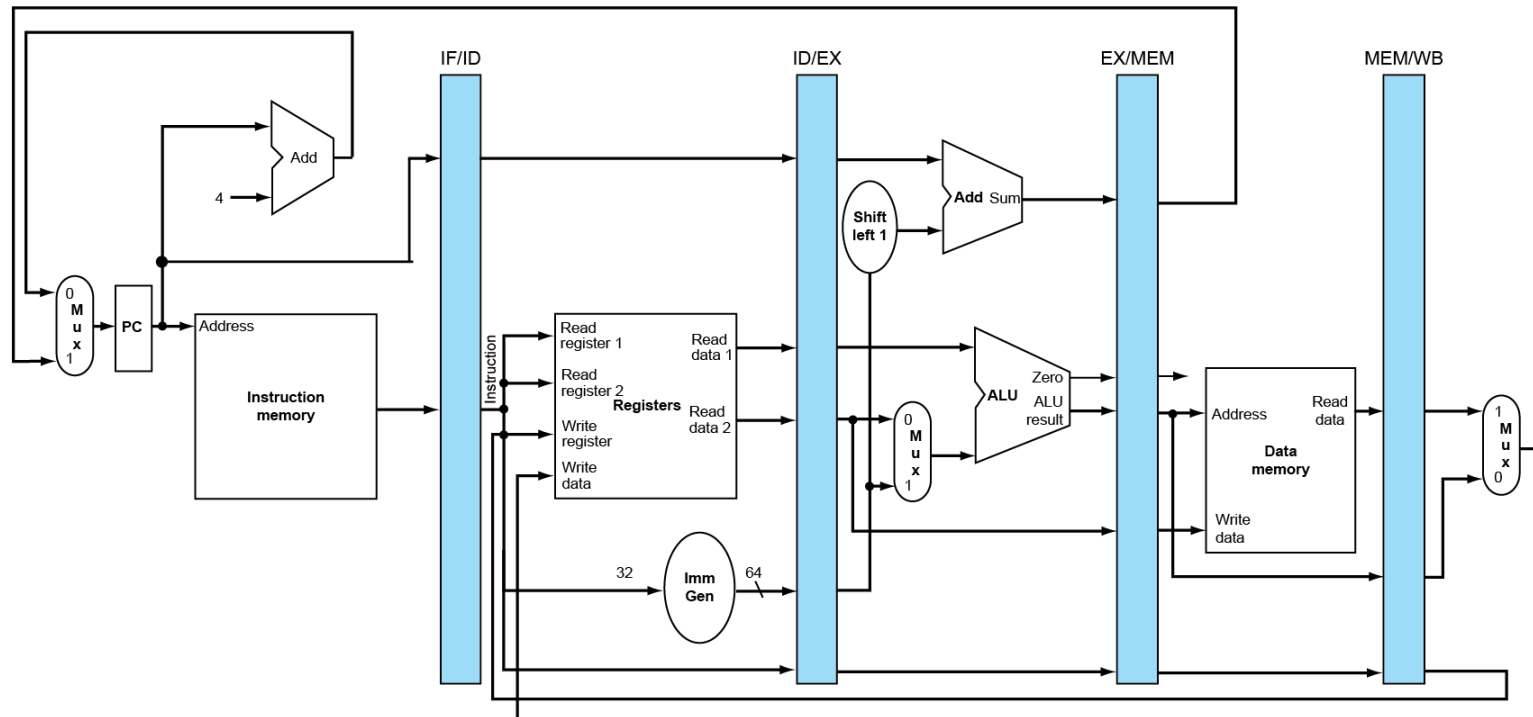
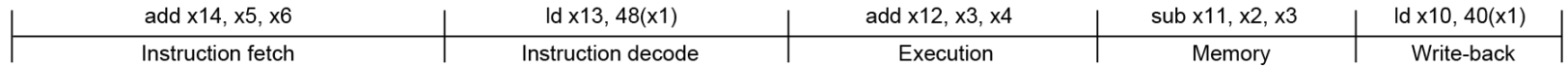
## › Traditional form



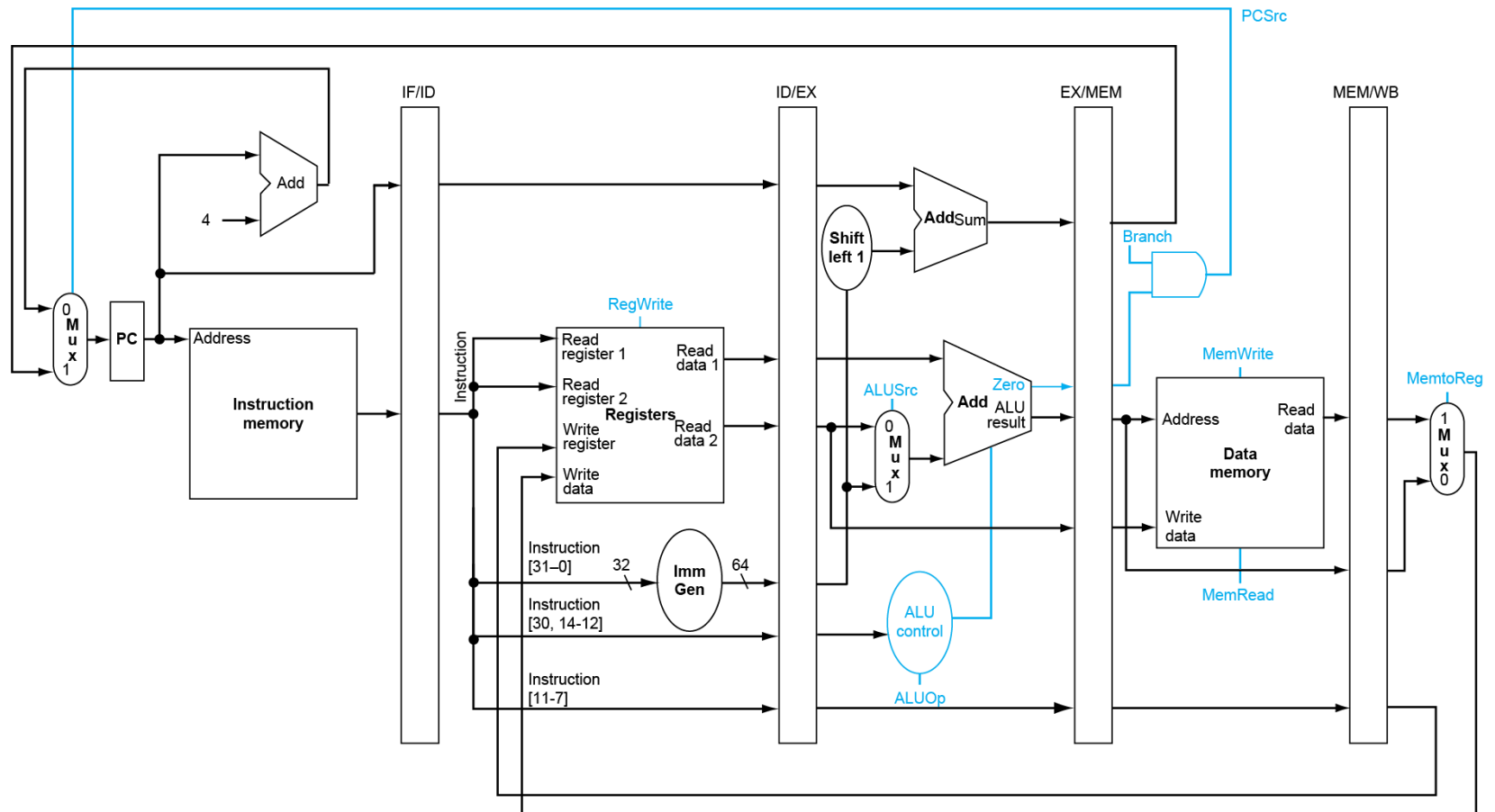
# Single-Cycle Pipeline Diagram



› State of pipeline in a given cycle



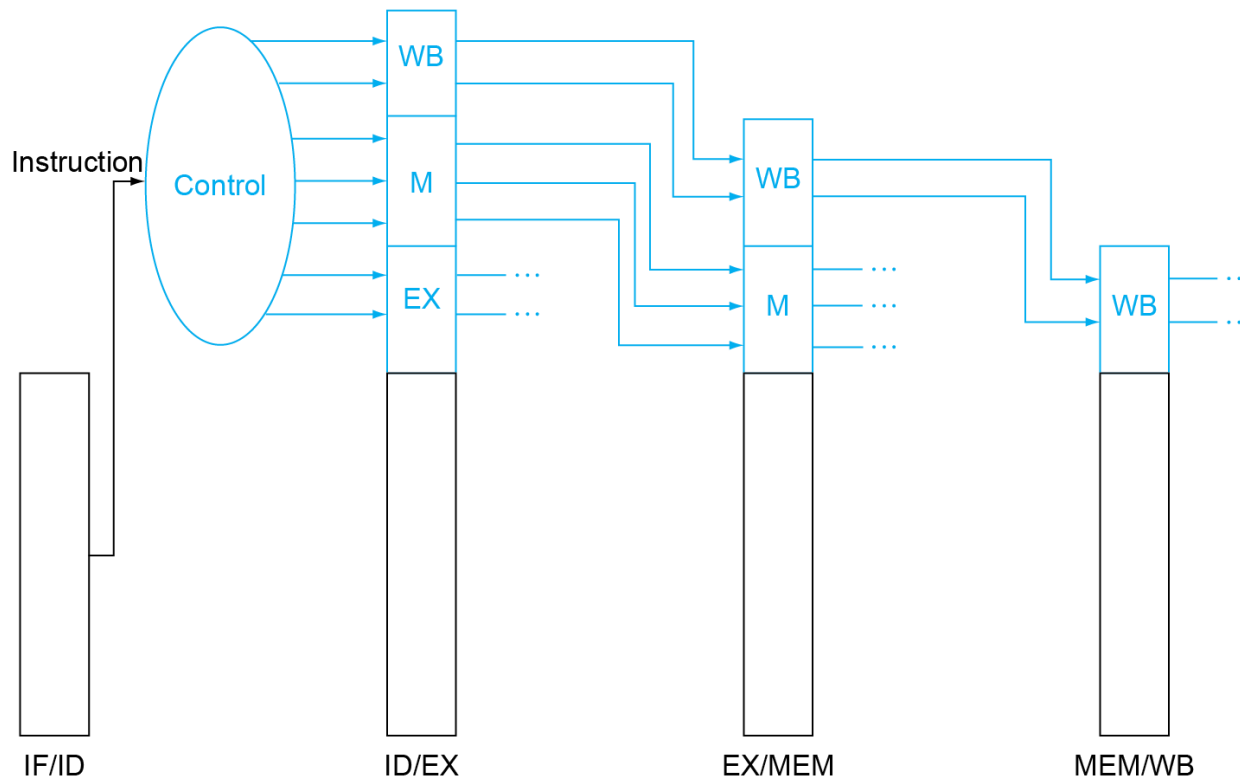
# Pipelined Control (Simplified)



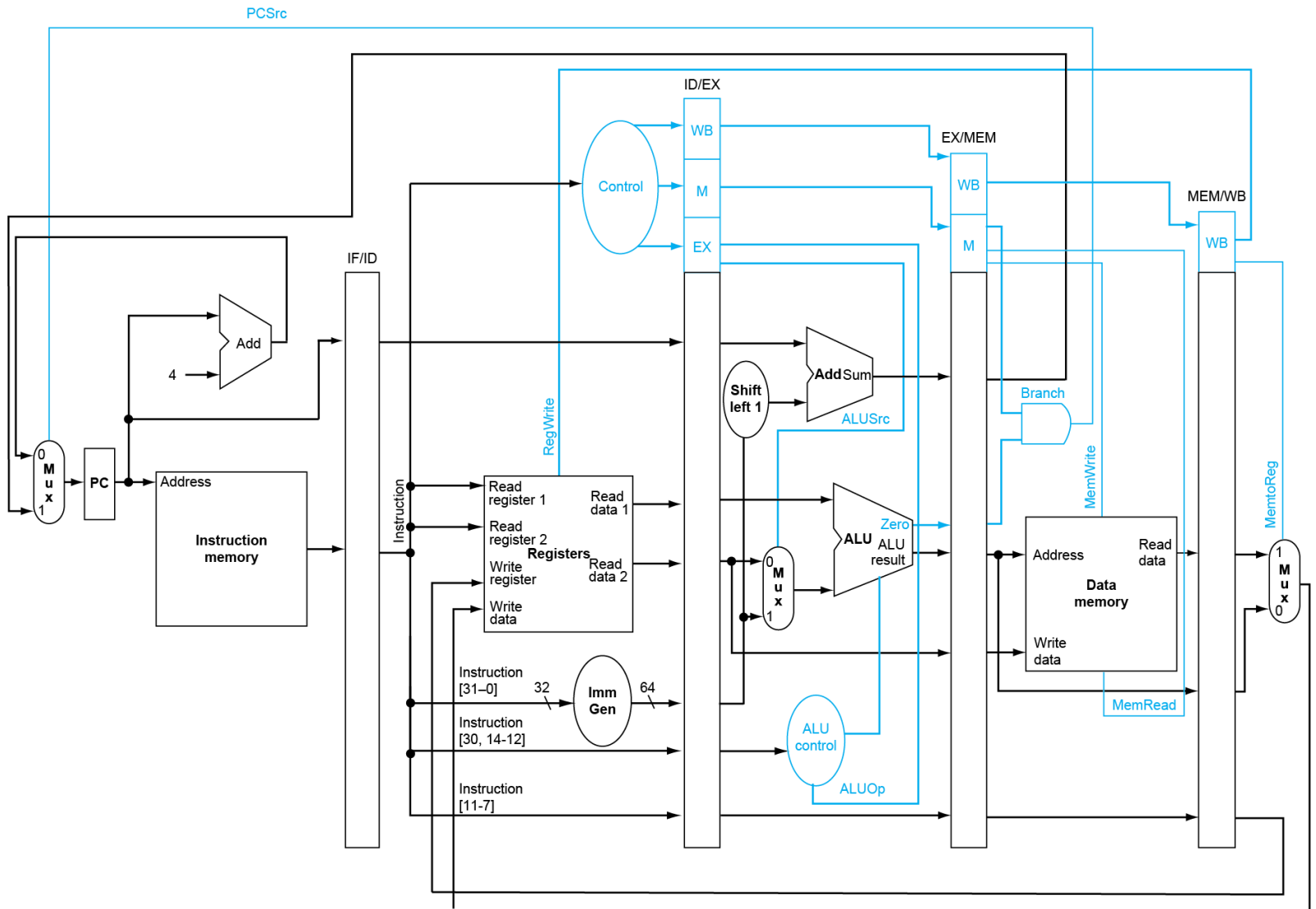
# Pipelined Control



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



# Pipelined Control





# 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)
```

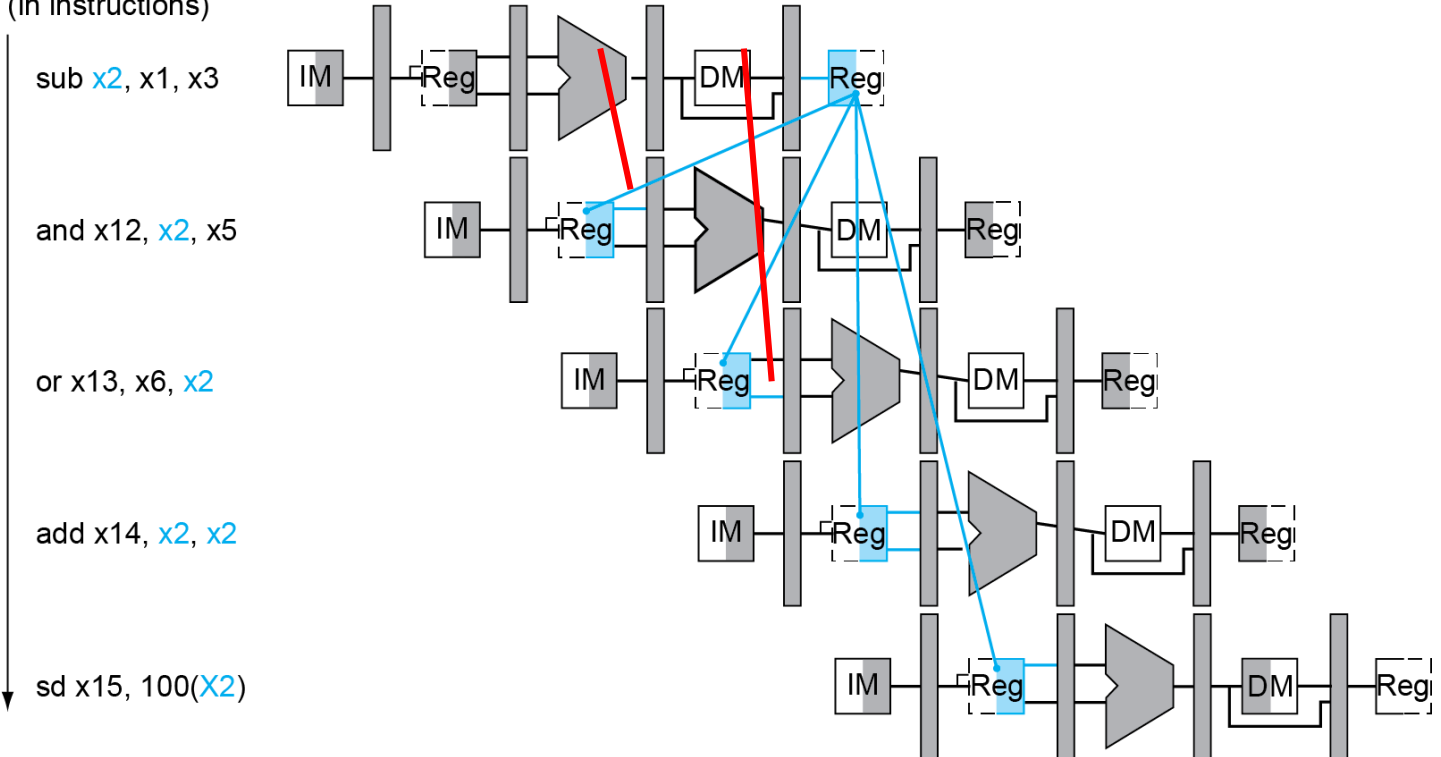
- › We can resolve hazards with forwarding
  - How do we detect when to forward?

# Dependencies & Forwarding



	Time (in clock cycles) →								
Value of register x2:	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
	10	10	10	10	10/-20	-20	-20	-20	-20

Program execution order (in instructions)



# Detecting the Need to Forward



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

Fwd from  
EX/MEM  
pipeline reg

Fwd from  
MEM/WB  
pipeline reg

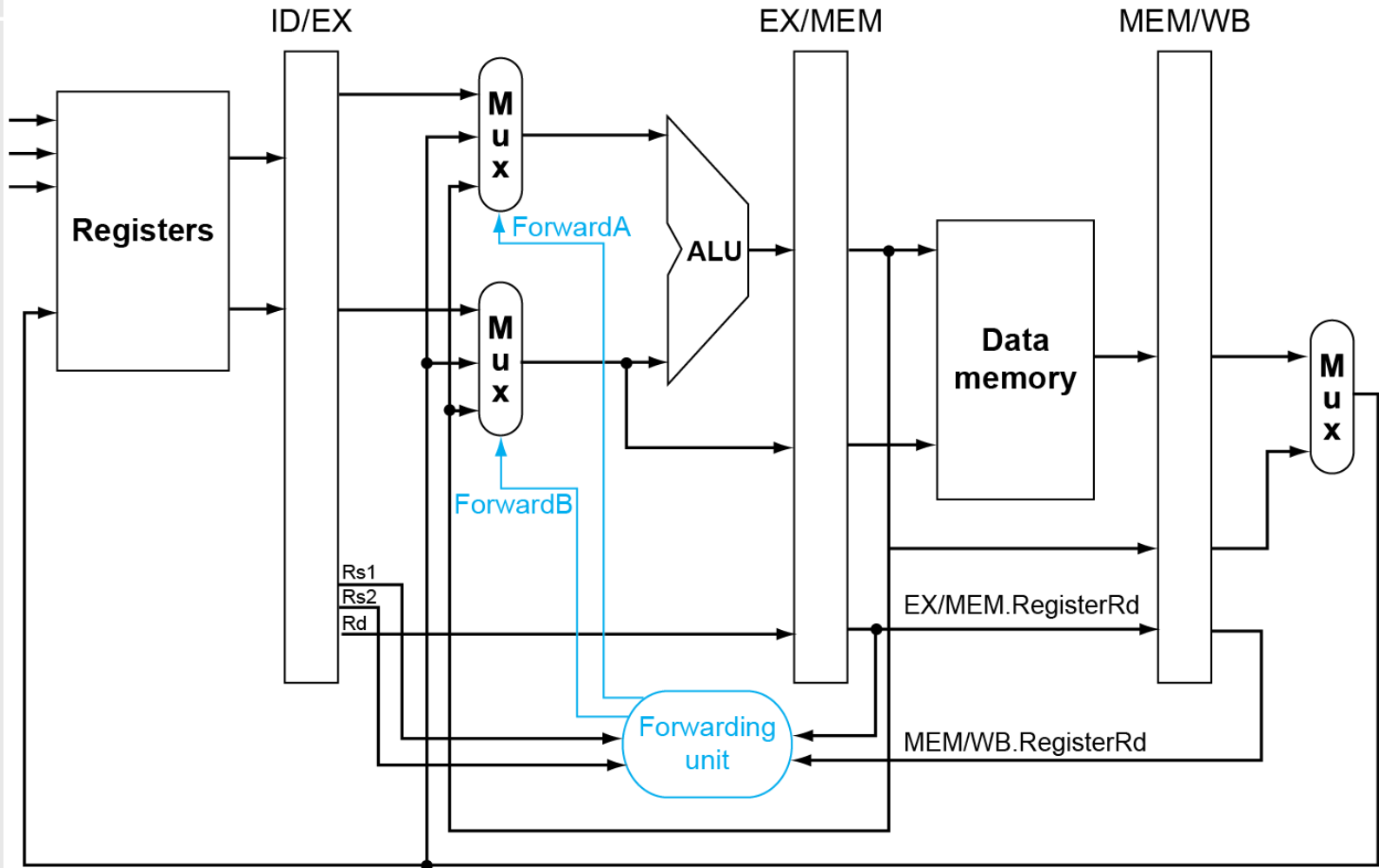
# Detecting the Need to Forward

---



- › 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 x0
  - EX/MEM.RegisterRd  $\neq$  0,  
MEM/WB.RegisterRd  $\neq$  0

# Forwarding Paths



# 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.

# 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

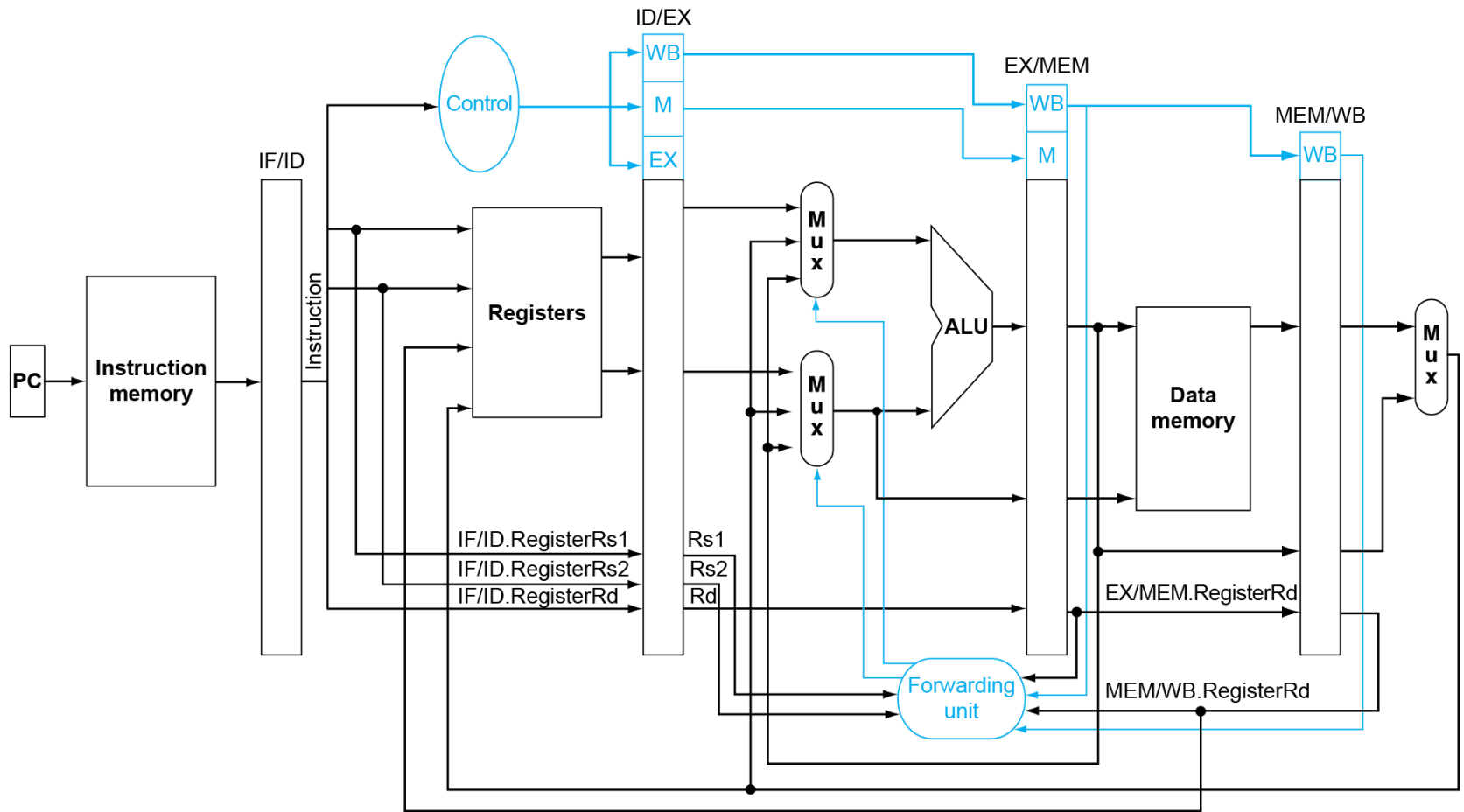
# Revised Forwarding Condition



- › 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  $\neq$  ID/EX.RegisterRs1))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) 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  $\neq$  ID/EX.RegisterRs2))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01



# Datapath with Forwarding



# 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.RegisterRs1, IF/ID.RegisterRs2
- › Load-use hazard when
  - ID/EX.MemRead and
    - ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or (ID/EX.RegisterRd = IF/ID.RegisterRs1))
- › If detected, stall and insert bubble

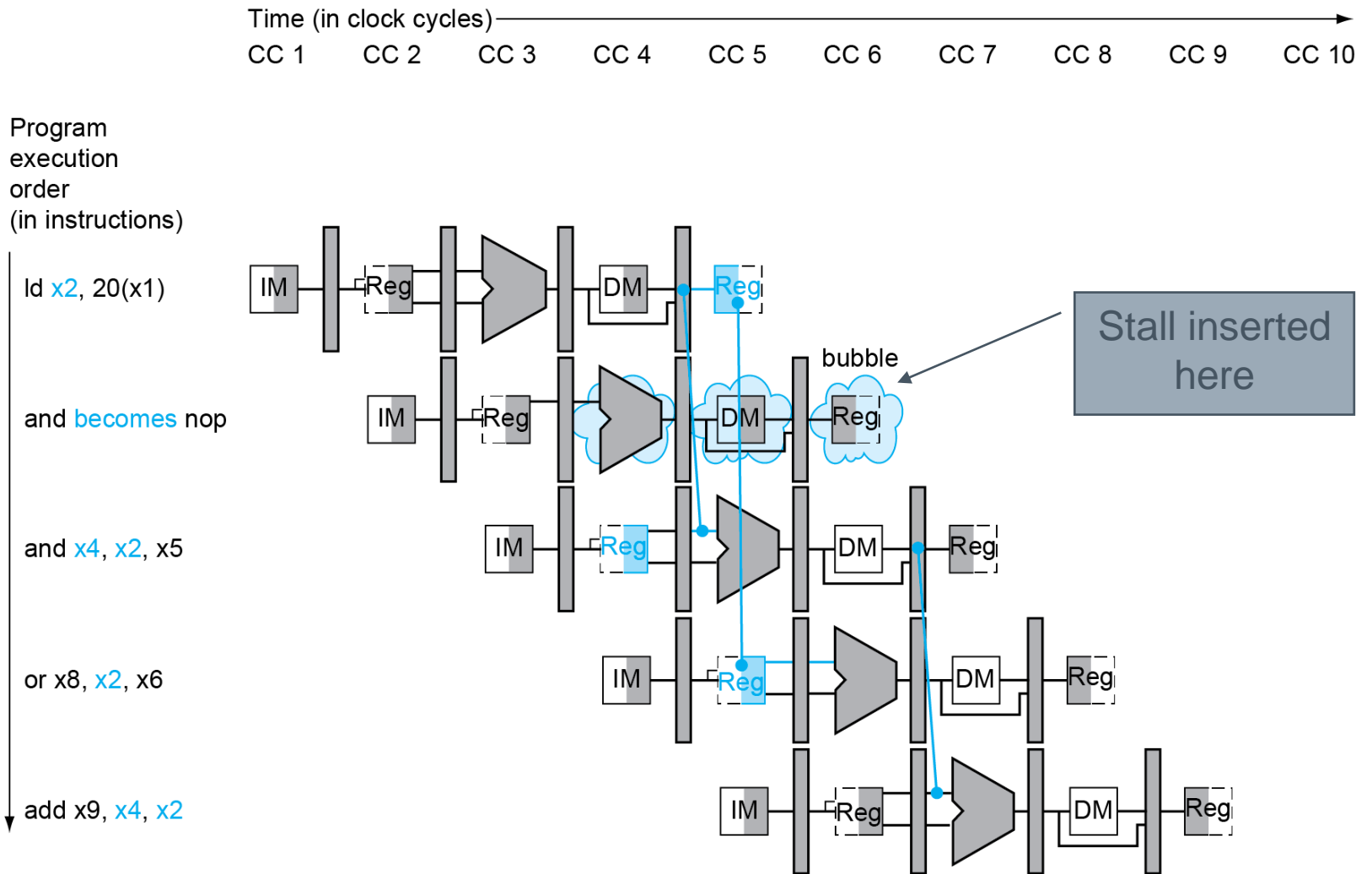
# How to Stall the Pipeline

---

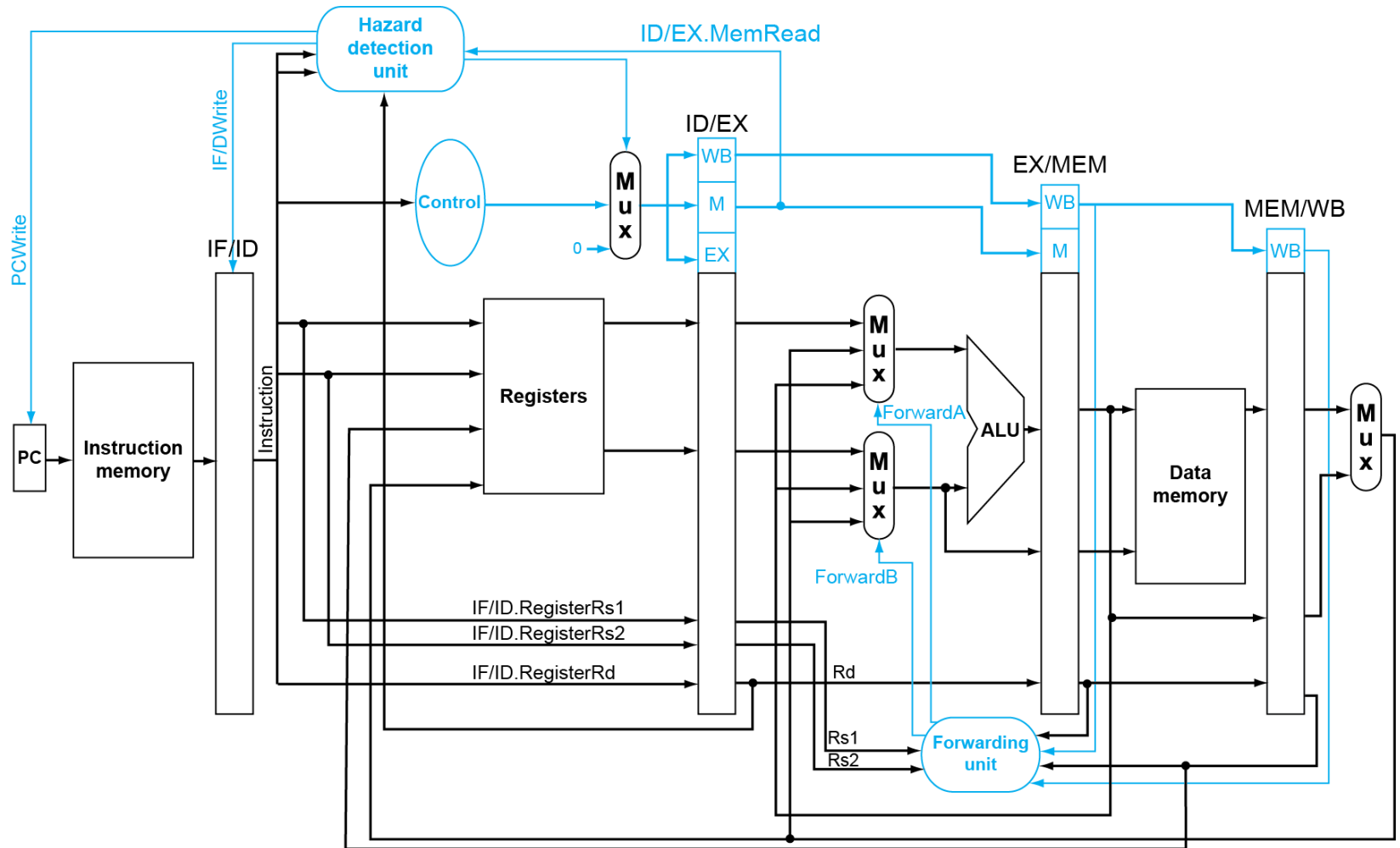


- › Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- › Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for 1d
    - › Can subsequently forward to EX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

# Load-Use Data Hazard



# Datapath with Hazard Detection



# Stalls and Performance

---

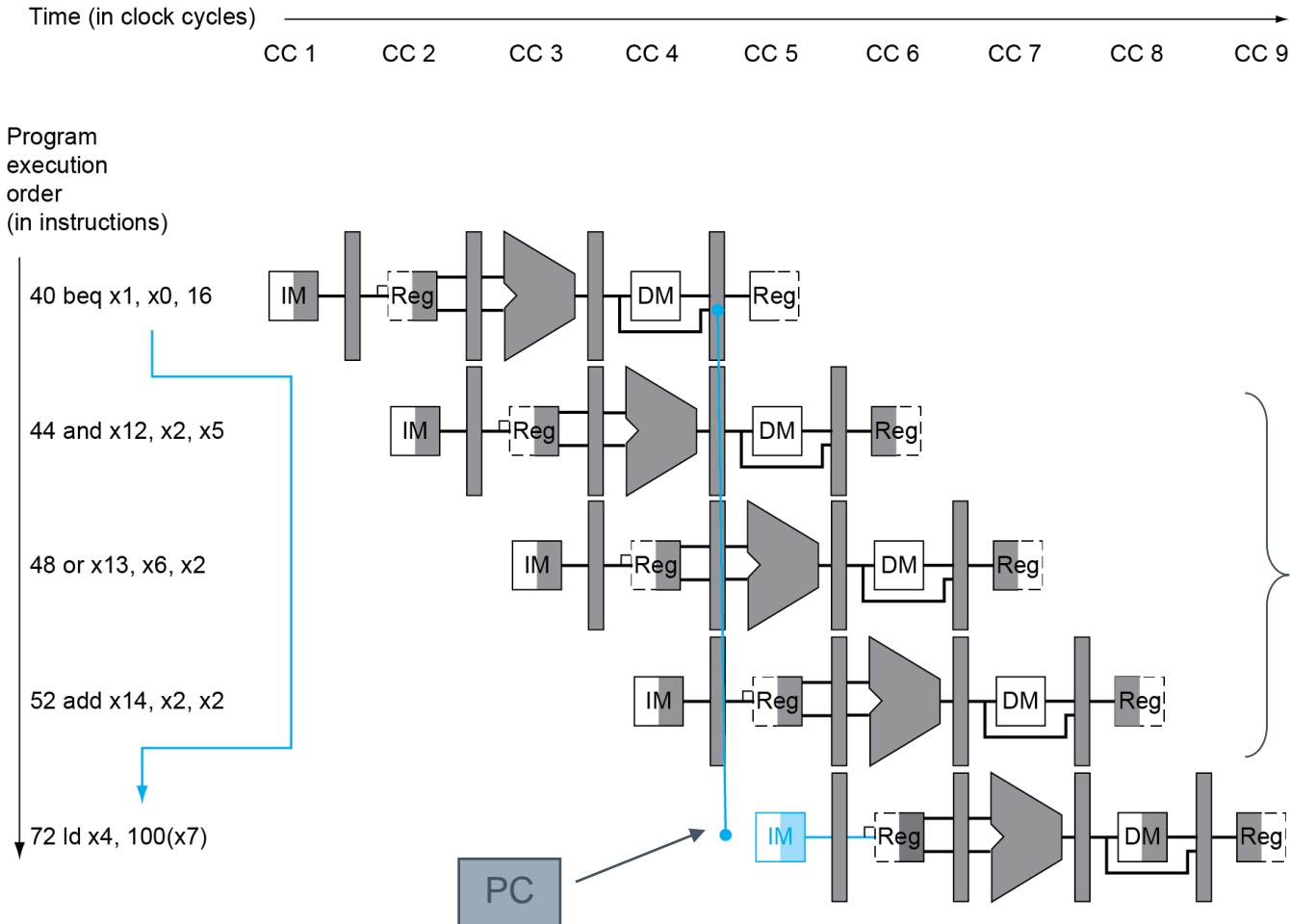


- › Stalls reduce performance
  - But are required to get correct results
- › Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

# Branch Hazards



› If branch outcome determined in MEM



Flush these instructions (Set control values to 0)

# Reducing Branch Delay



- › Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator

- › Example: branch taken

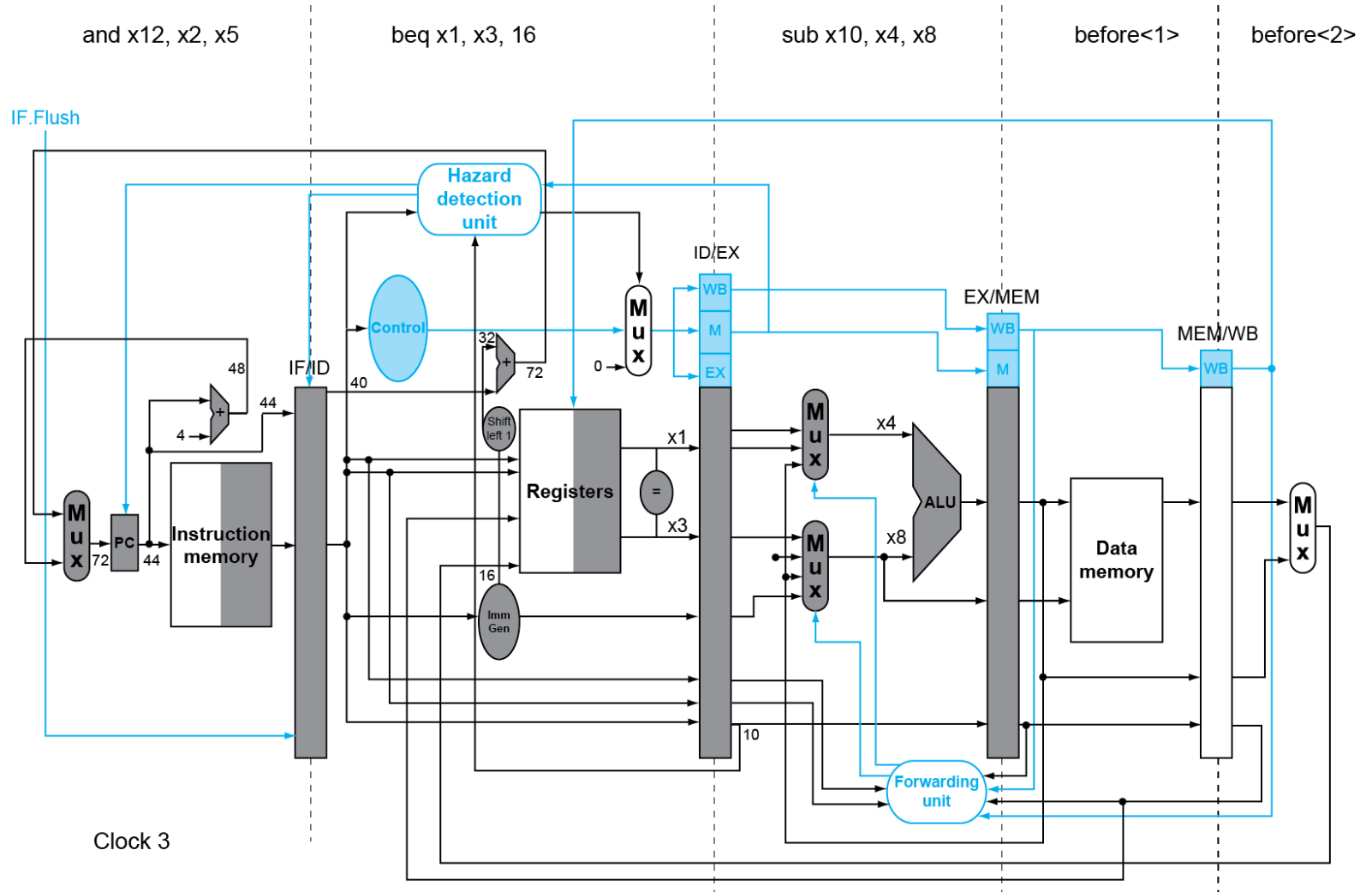
```
36:  sub   x10, x4, x8
40:  beq   x1,  x3, 16    // PC-relative
    branch                // to 40+16*2=72

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

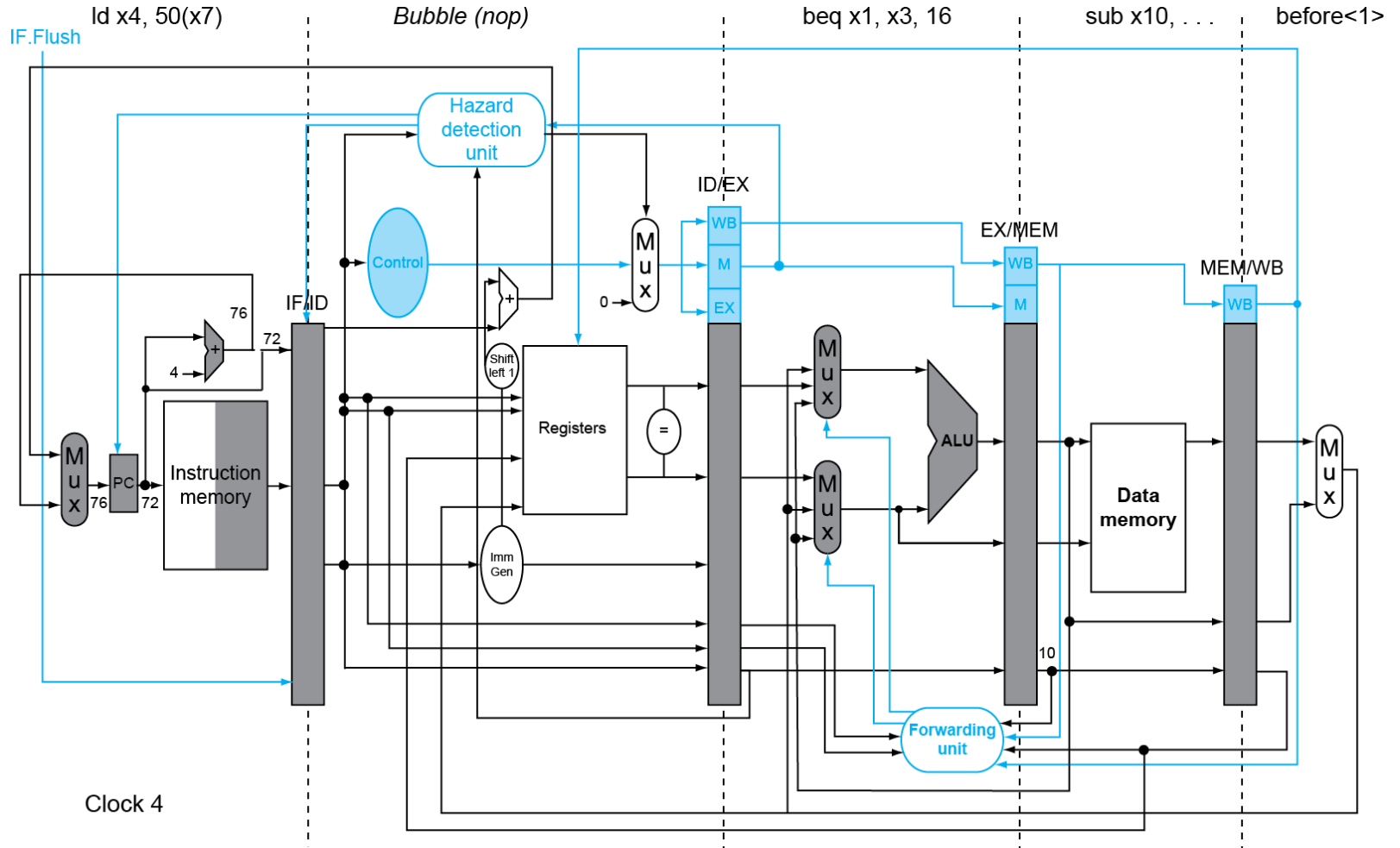
72:  ld    x4, 50(x7)
```



# Example: Branch Taken



# Example: Branch Taken



# Dynamic Branch Prediction

---



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

# 1-Bit Predictor: Shortcoming



- › Inner loop branches mispredicted twice!

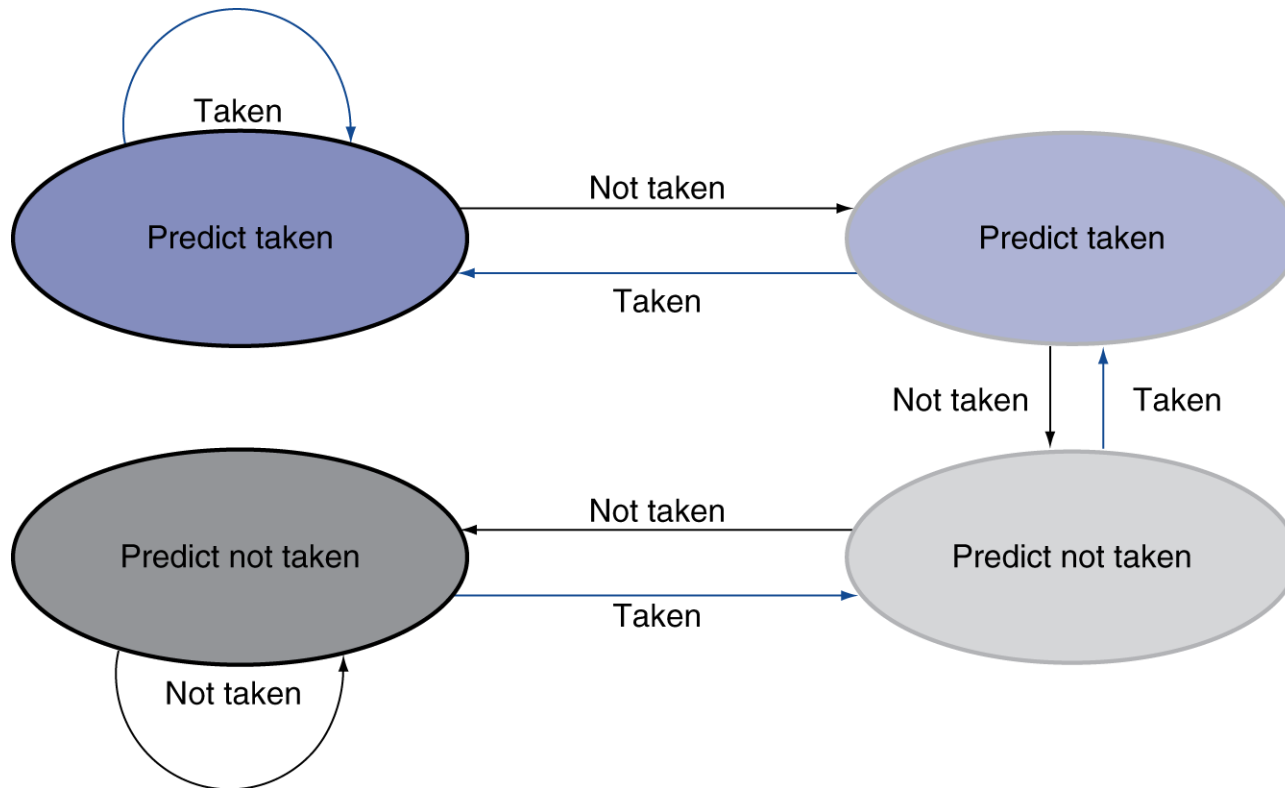


- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

# 2-Bit Predictor



- › Only change prediction on two successive mispredictions



# Calculating the Branch Target

---



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

# Exceptions and Interrupts



- › “Unexpected” events requiring change in flow of control
  - Different ISAs use the terms differently
- › Exception
  - Arises within the CPU
    - › e.g., undefined opcode, 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`



# 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

# 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, ...

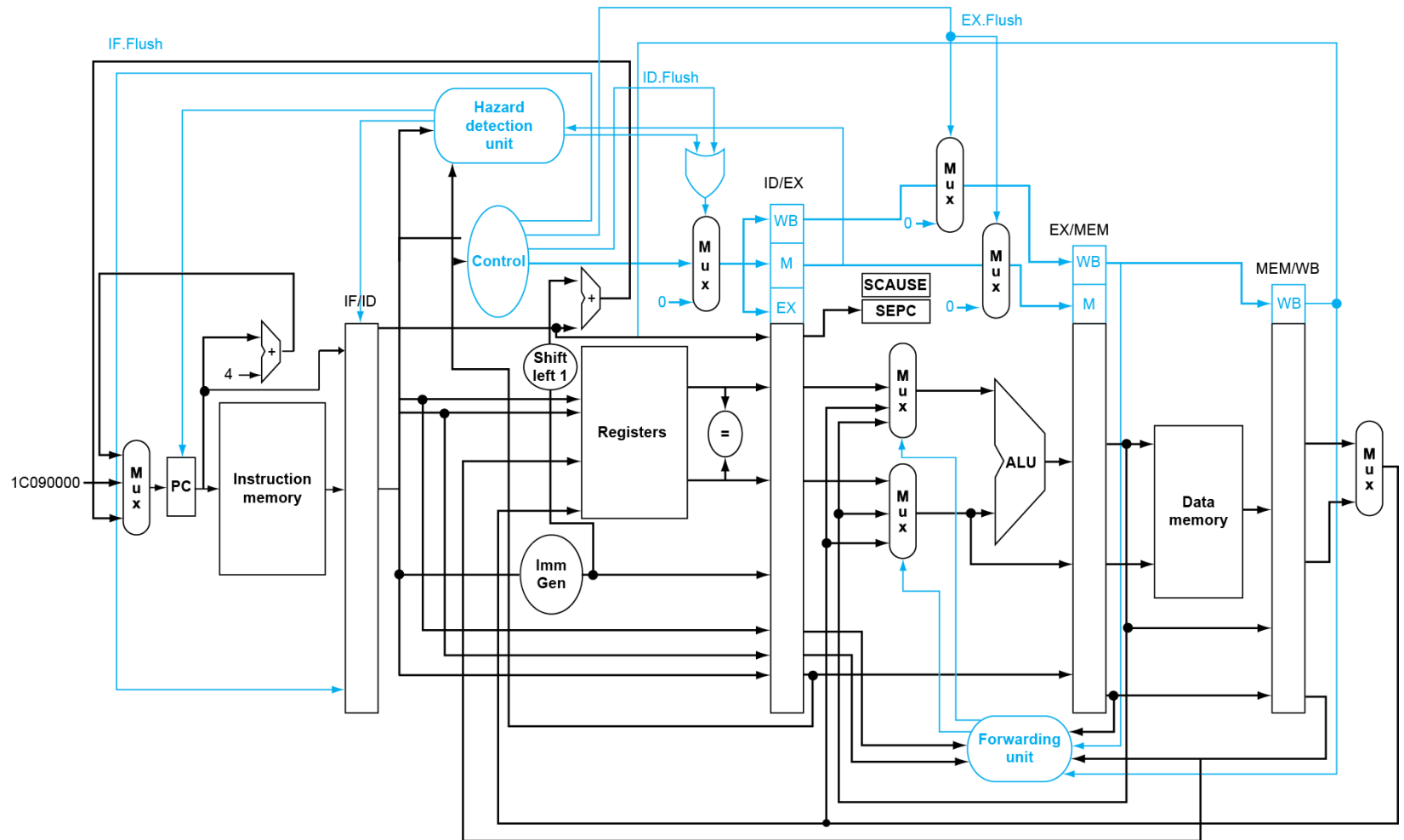
# Exceptions in a Pipeline

---



- › 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

# 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      orr    x13, x2, x6
4c      add    x1,  x2, x1
50      sub    x15, x6, x7
54      ld     x16, 100(x7)
```

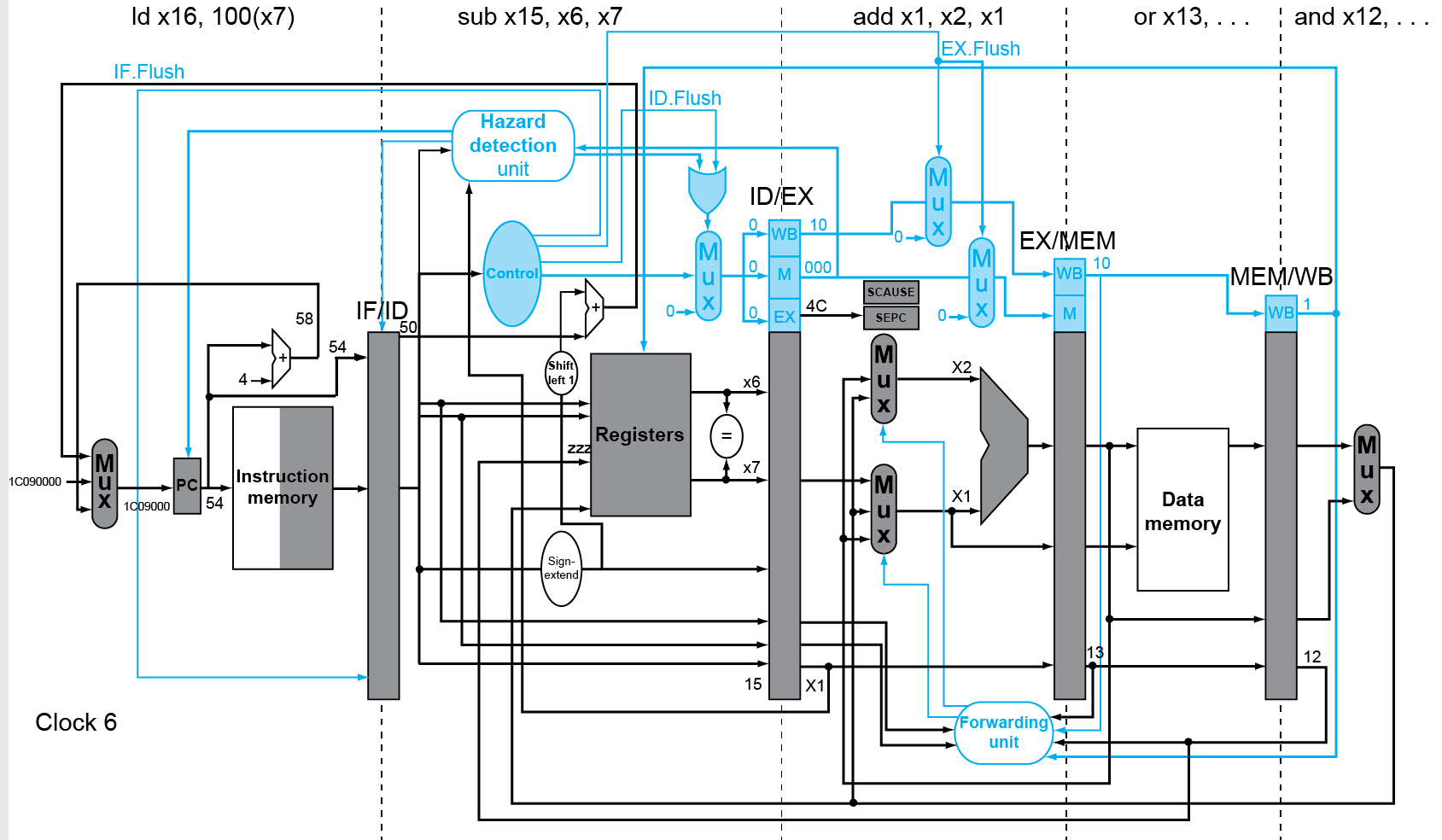
...

## › Handler

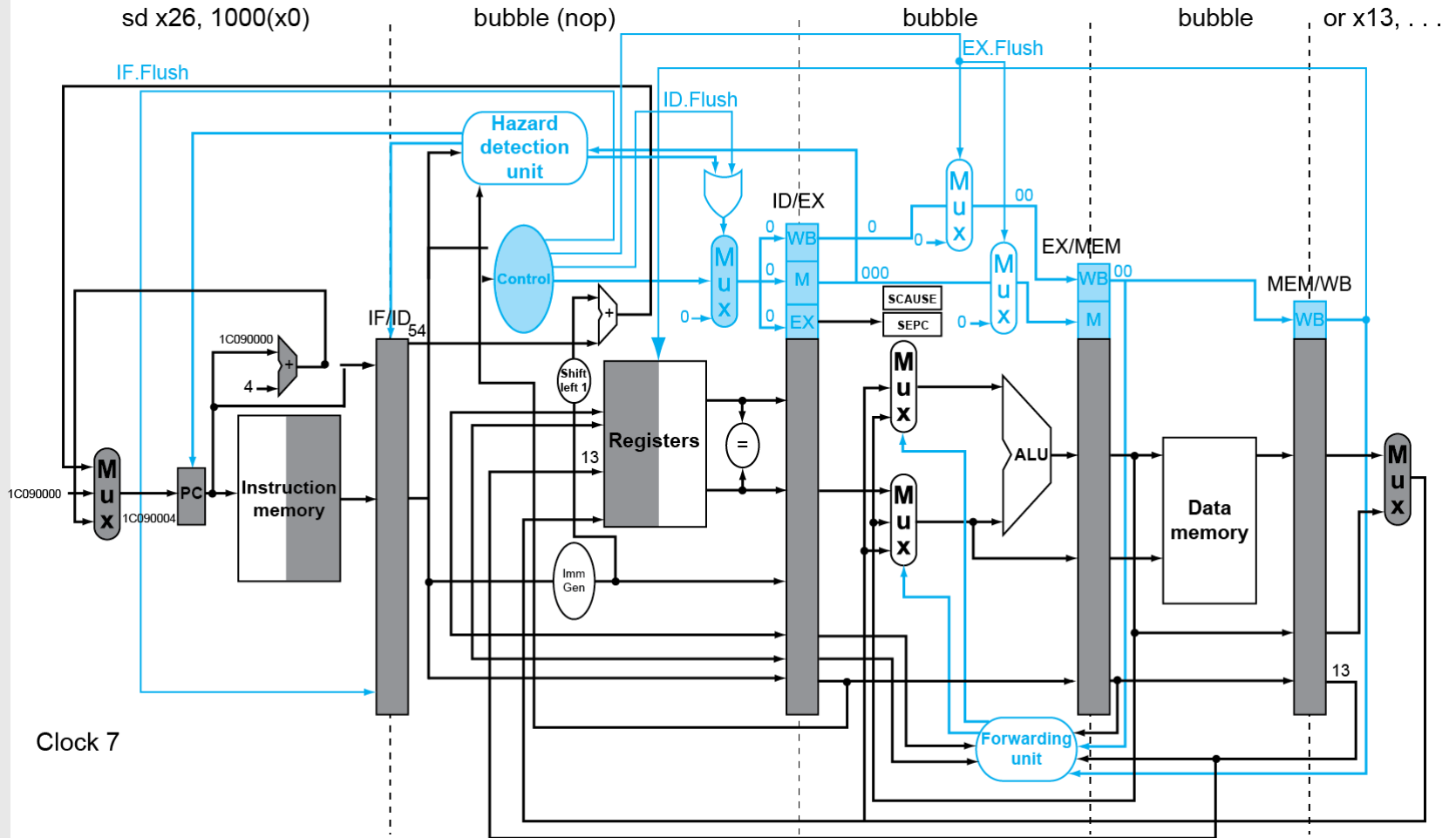
```
1c090000    sd    x26, 1000(x10)
1c090004    sd    x27, 1008(x10)
```

...

# 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!

# Imprecise Exceptions

---



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

# Instruction-Level Parallelism (ILP)



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

# Multiple Issue

---



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

# Speculation



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

# Compiler/Hardware Speculation

---



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

# Speculation and Exceptions

---



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

# Static Multiple Issue

---



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



# 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

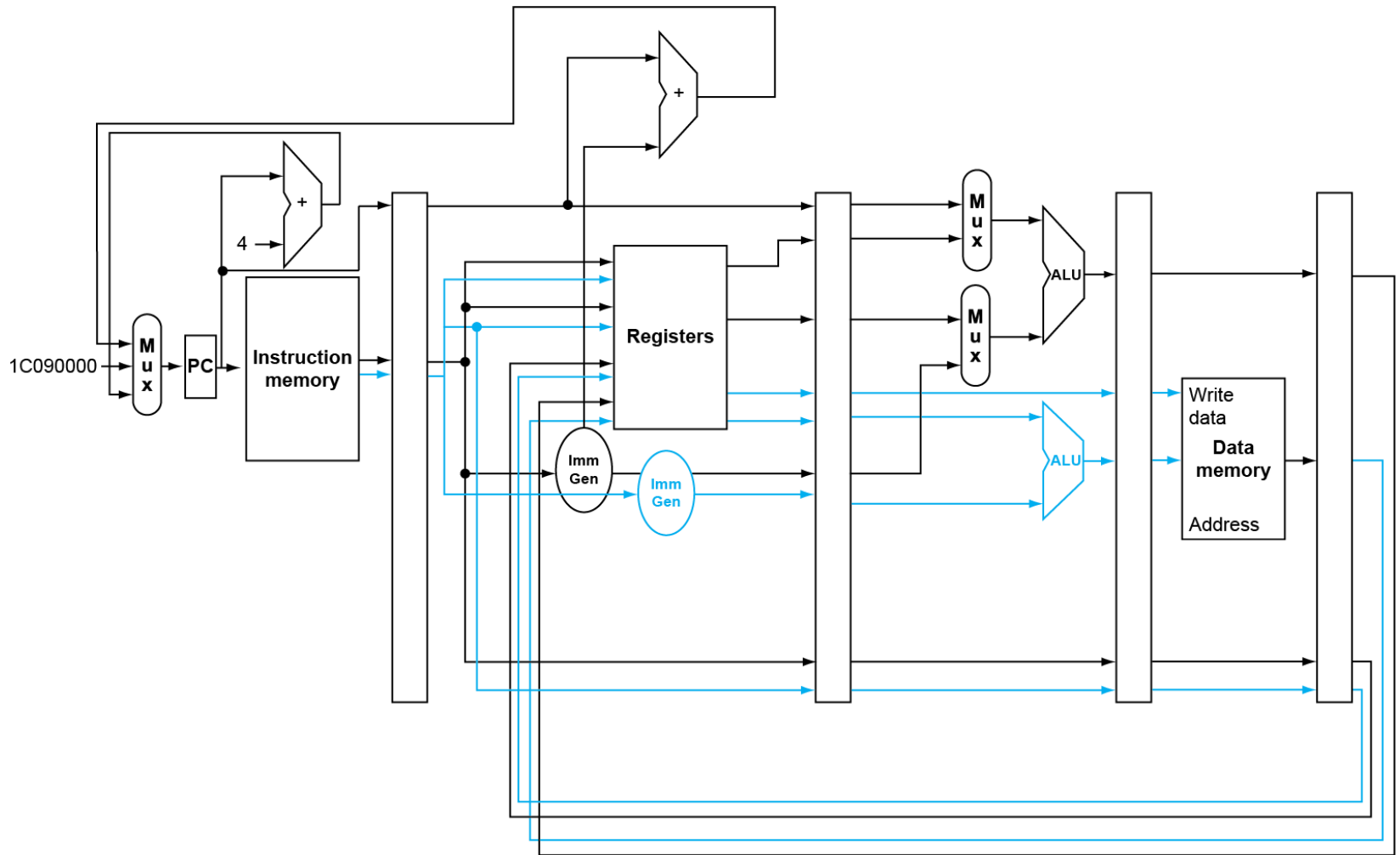
# RISC-V with Static Dual Issue



- › 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						
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

# 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  $IPC = 2$ )

need to add 8 because content of x20  
was decreased in the meantime

# Loop Unrolling

---



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

# Loop Unrolling Example



need to rename registers to avoid anti-dependencies

- › Loop unrolled by a factor 4

```
Loop:  ld    x31, 0(x20)           // x31=array element
iteration 0  add    x31, x31, x21      // add scalar in x21
          sd    x31, 0(x20)        // store result
-----
iteration 1  ld    x31, -8(x20)      // x31=array element
          add    x31, x31, x21      // add scalar in x21
          sd    x31, -8(x20)       // store result
-----
iteration 2  ld    x31, -16(x20)    // x31=array element
          add    x31, x31, x21      // add scalar in x21
          sd    x31, -16(x20)     // store result
-----
iteration 3  ld    x31, -24(x20)    // x31=array element
          add    x31, x31, x21      // add scalar in x21
          sd    x31, -24(x20)     // store result
-----

          addi   x20, x20, -32      // decrement pointer
          blt    x22, x20, Loop     // branch if x22 < x20
```

# Loop Unrolling Example



- › Loop unrolled by a factor 4

```
Loop:  ld    x28, 0(x20)           // x31=array element
iteration 0  add    x28, x28, x21       // add scalar in x21
          sd    x28, 0(x20)       // store result
-----
iteration 1  ld    x29, -8(x20)      // x31=array element
          add    x29, x29, x21     // add scalar in x21
          sd    x29, -8(x20)     // store result
-----
iteration 2  ld    x30, -16(x20)    // x31=array element
          add    x30, x30, x21    // add scalar in x21
          sd    x30, -16(x20)    // store result
-----
iteration 3  ld    x31, -24(x20)    // x31=array element
          add    x31, x31, x21    // add scalar in x21
          sd    x31, -24(x20)    // store result
-----

          addi   x20, x20, -32     // decrement pointer
          blt    x22, x20, Loop    // branch if x22 < x20
```



# 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



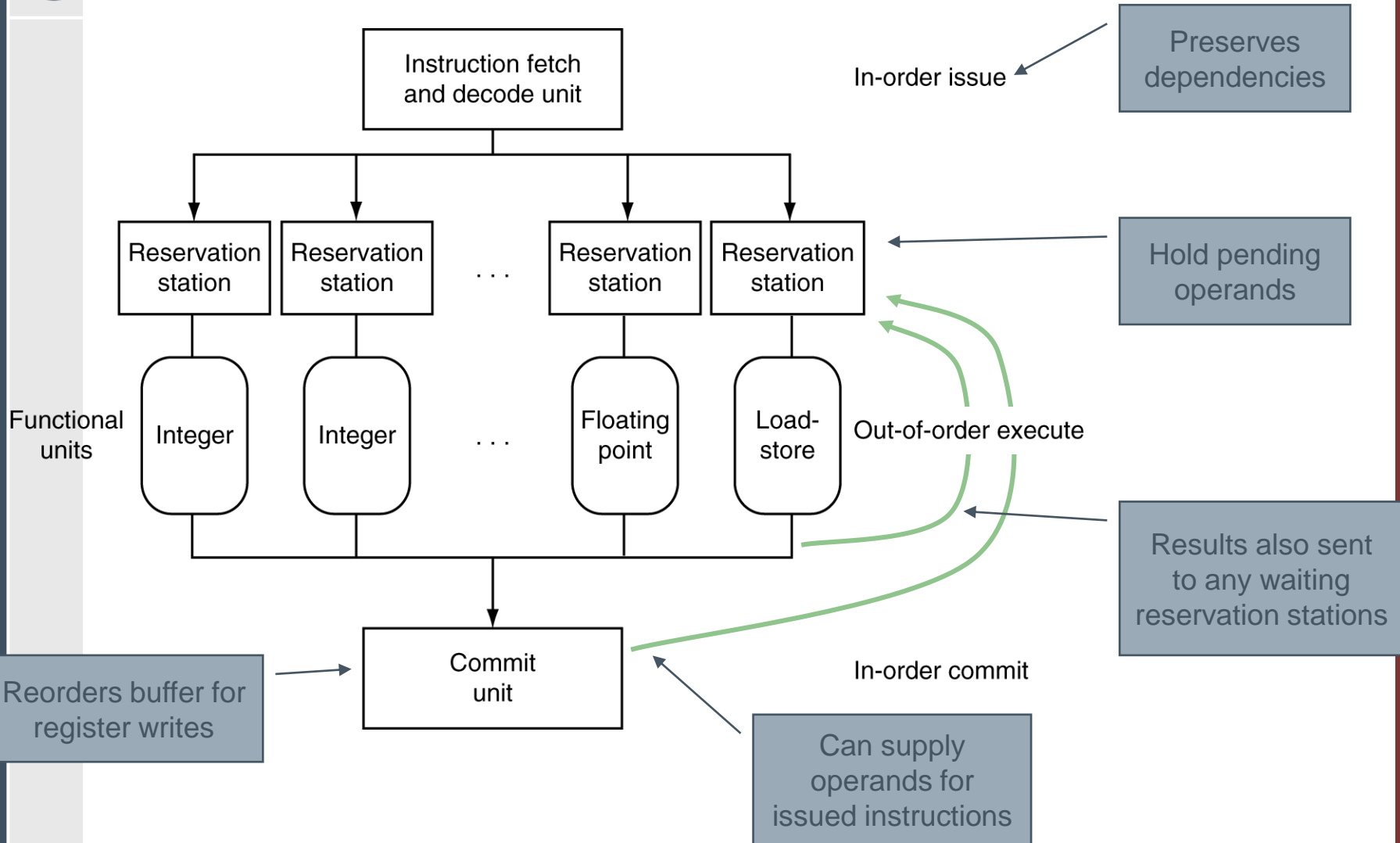
- › Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order

- › 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



# Register Renaming

---



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

# Speculation

---



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

# Why Do Dynamic Scheduling?

---



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

# Does Multiple Issue Work?



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



# Power Efficiency



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

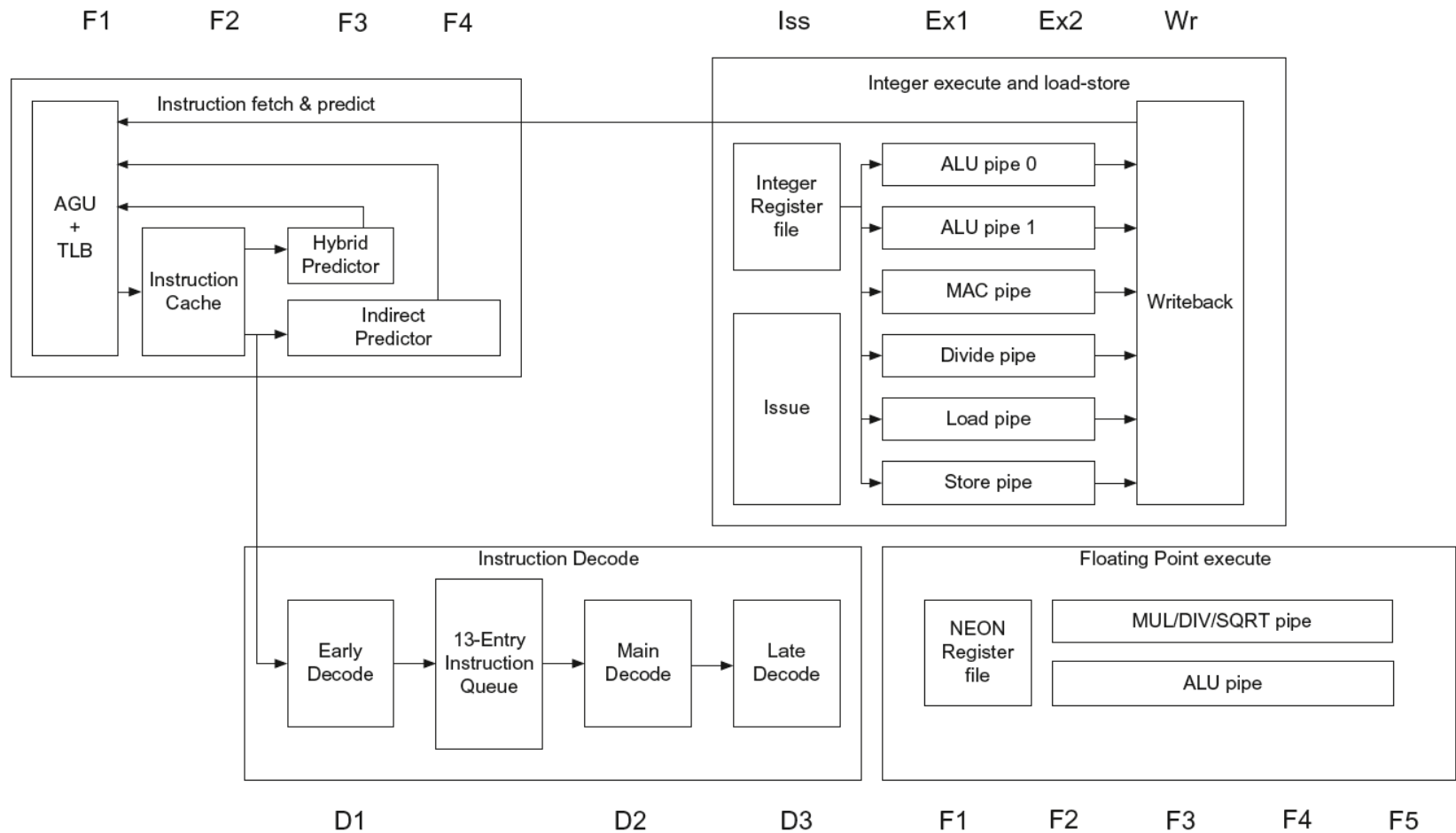
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4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

# Cortex A53 and Intel i7

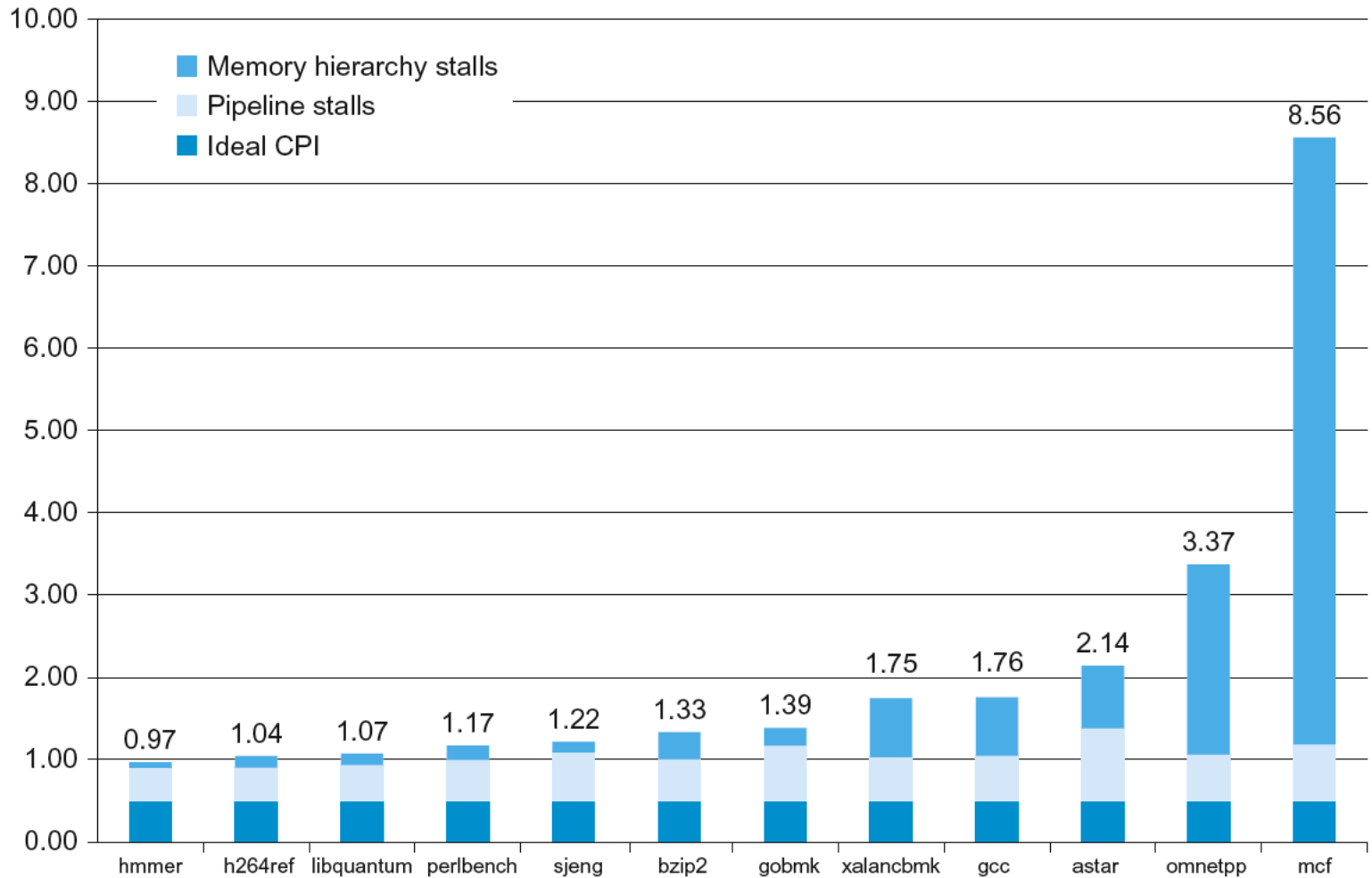


Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-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)	2-8 MB

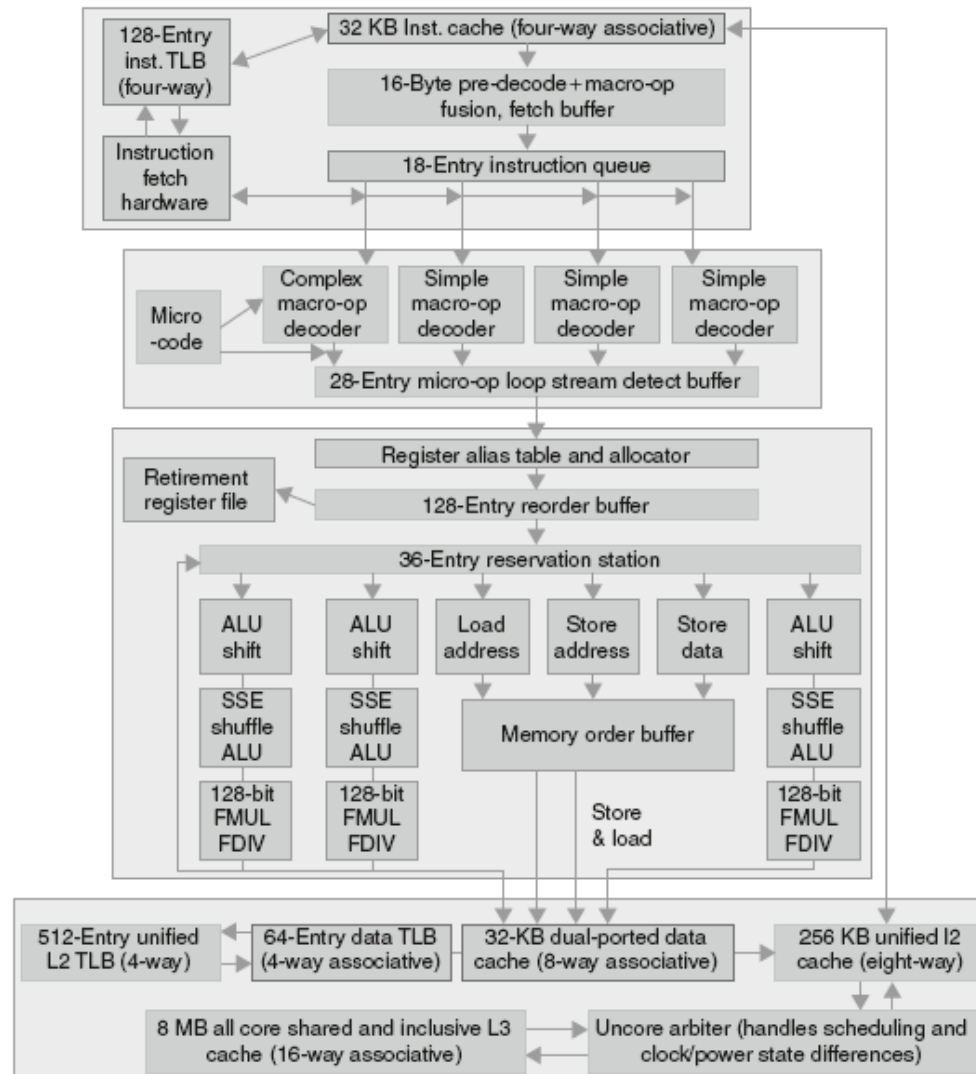
# ARM Cortex-A53 Pipeline



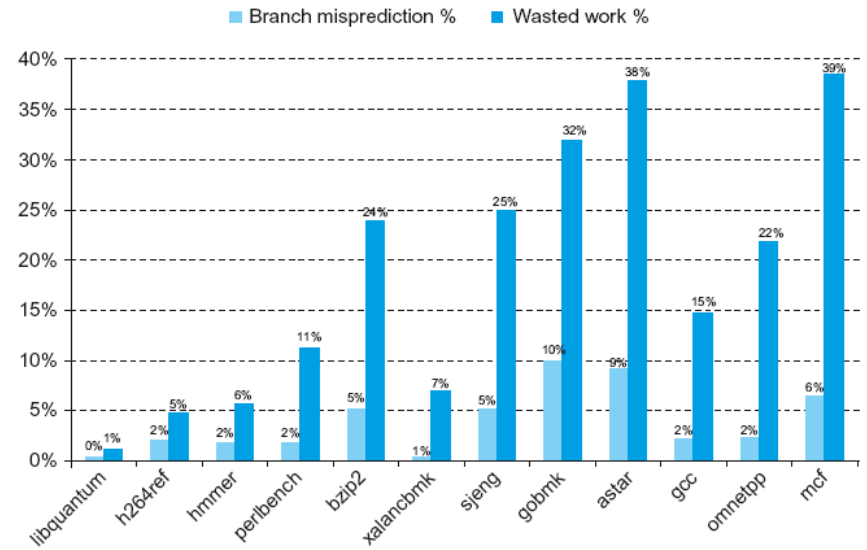
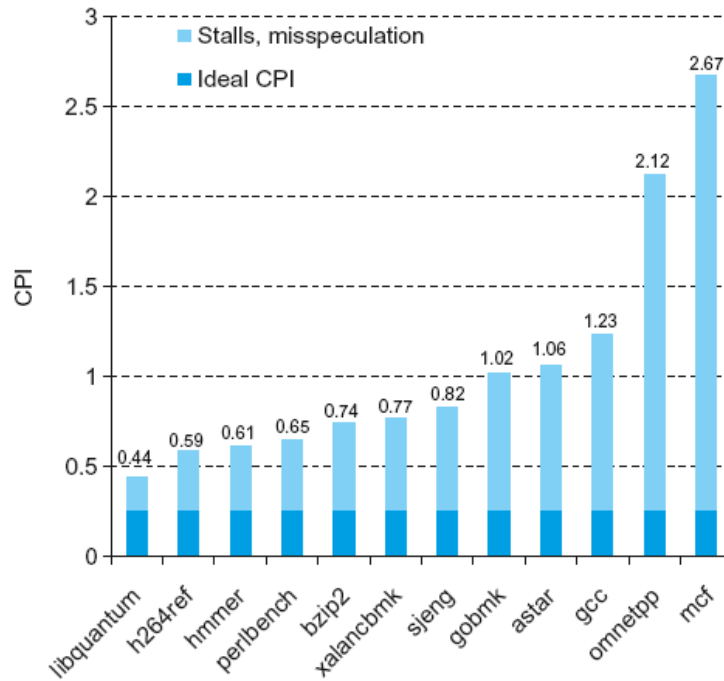
# ARM Cortex-A53 Performance



# Core i7 Pipeline



# Core i7 Performance



# Matrix Multiply



## › Unrolled C code

```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6   for ( int i = 0; i < n; i+=UNROLL*4 )
7     for ( int j = 0; j < n; j++ ) {
8       __m256d c[4];
9       for ( int x = 0; x < UNROLL; x++ )
10        c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12      for( int k = 0; k < n; k++ )
13        {
14          __m256d b = _mm256_broadcast_sd(B+k+j*n);
15          for (int x = 0; x < UNROLL; x++)
16            c[x] = _mm256_add_pd(c[x],
17                               _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18        }
19
20      for ( int x = 0; x < UNROLL; x++ )
21        _mm256_store_pd(C+i+x*4+j*n, c[x]);
22    }
23 }
```

# Matrix Multiply

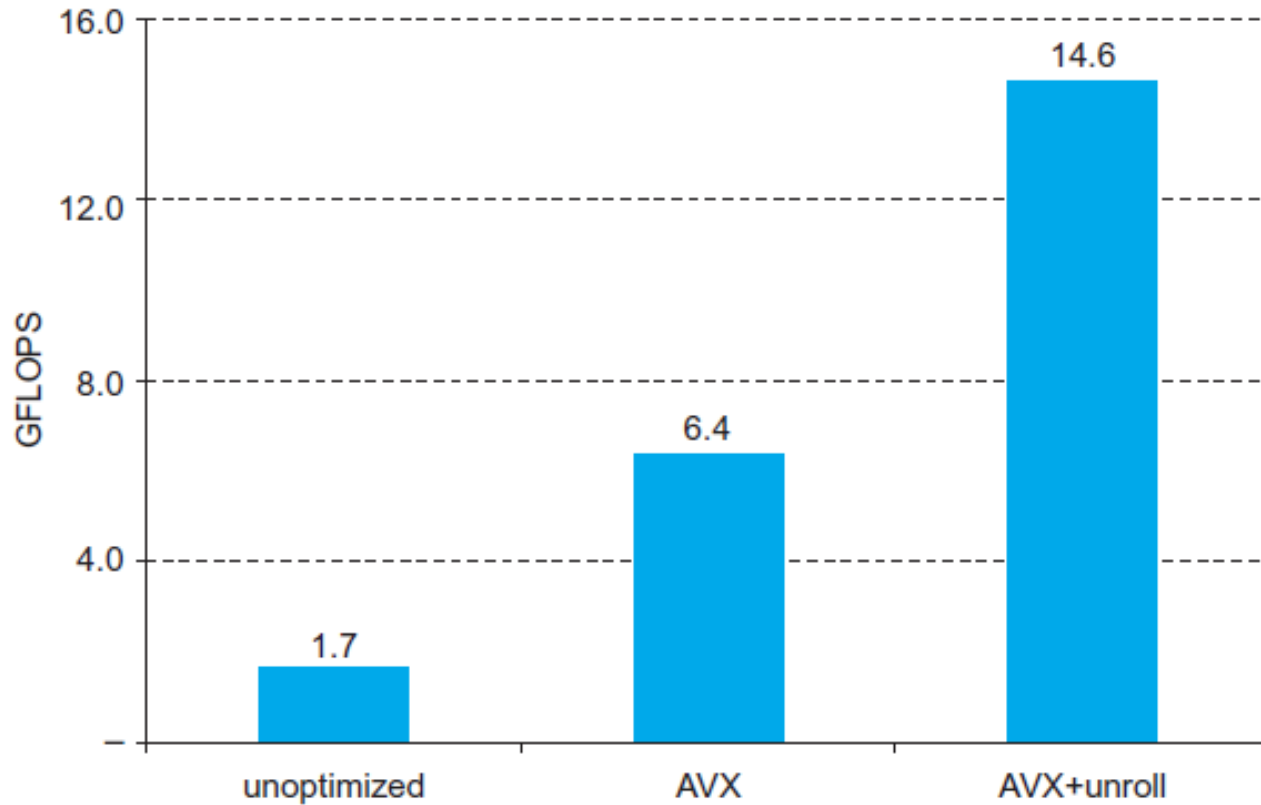


## > Assembly code:

```
1 vmovapd (%r11),%ymm4           # Load 4 elements of C into %ymm4
2 mov %rbx,%rax                 # register %rax = %rbx
3 xor %ecx,%ecx                 # register %ecx = 0
4 vmovapd 0x20(%r11),%ymm3      # Load 4 elements of C into %ymm3
5 vmovapd 0x40(%r11),%ymm2      # Load 4 elements of C into %ymm2
6 vmovapd 0x60(%r11),%ymm1      # Load 4 elements of C into %ymm1
7 vbroadcastsd (%rcx,%r9,1),%ymm0 # Make 4 copies of B element
8 add $0x8,%rcx # register %rcx = %rcx + 8
9 vmulpd (%rax),%ymm0,%ymm5     # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4     # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5 # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3     # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5 # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0 # Parallel mul %ymm1,4 A elements
15 add %r8,%rax                 # register %rax = %rax + %r8
16 cmp %r10,%rcx               # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2     # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1     # Parallel add %ymm0, %ymm1
19 jne 68 <dgemm+0x68>          # jump if not %r8 != %rax
20 add $0x1,%esi                # register % esi = % esi + 1
21 vmovapd %ymm4,(%r11)         # Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)     # Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)     # Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)     # Store %ymm1 into 4 C elements
```



# Performance Impact



# Fallacies



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

# Pitfalls

---



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

# Concluding Remarks

---



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