



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

Simple subset, shows most aspects

Memory reference: ld, sd

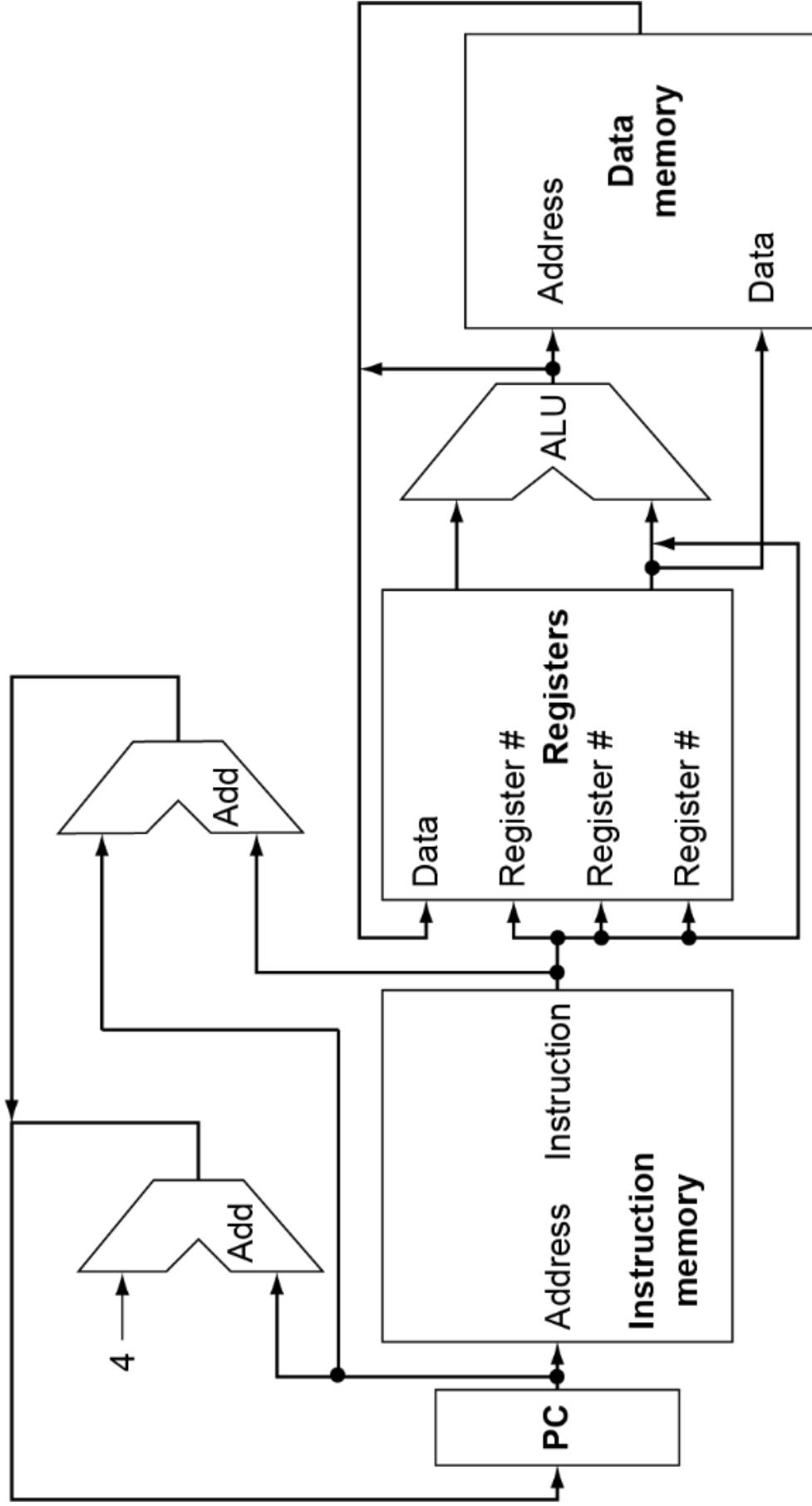
Arithmetic/logical: add, sub, and, or

Control transfer: beq

Instruction Execution

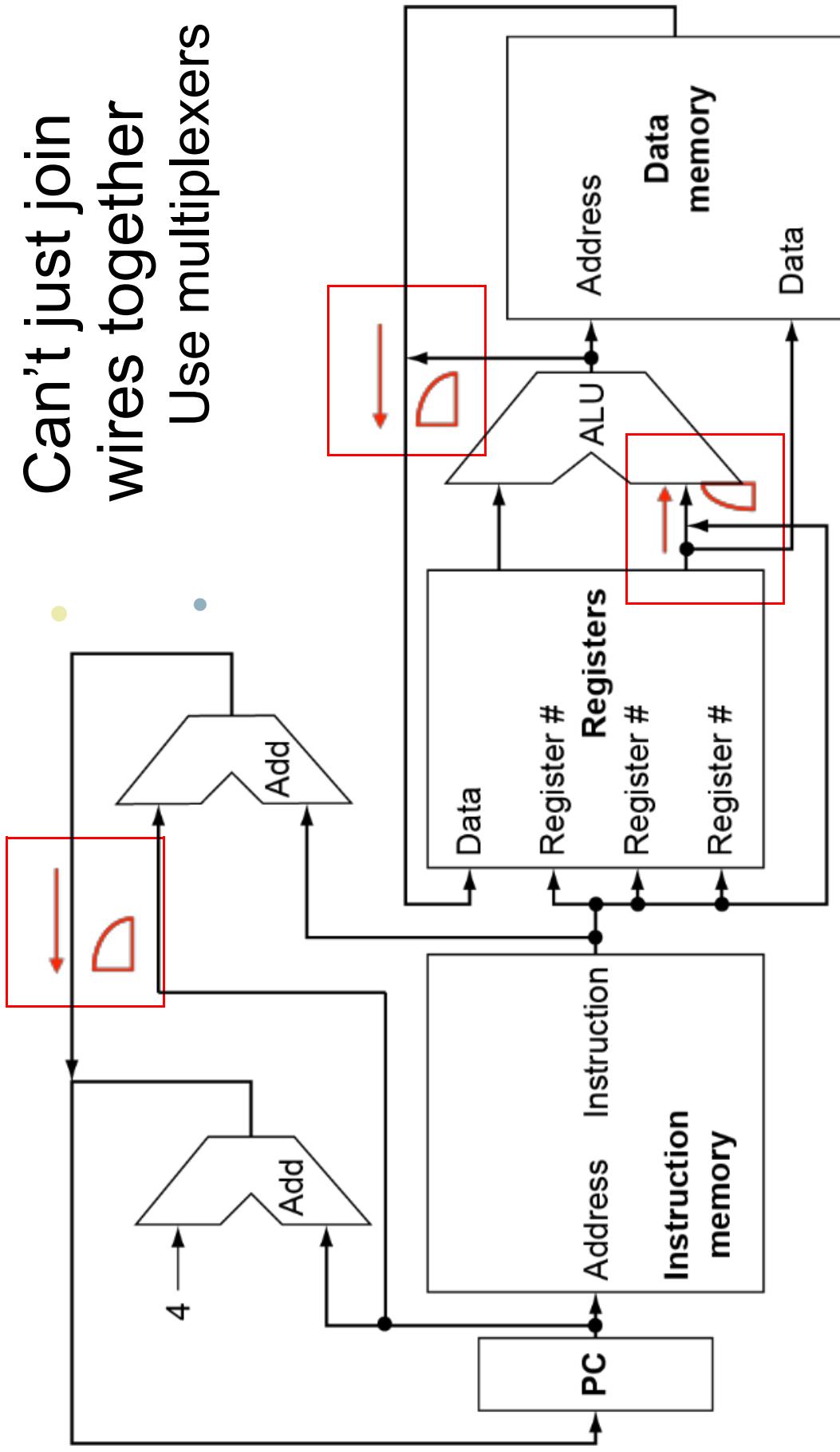
- PC instruction memory, fetch instruction
- Register numbers register file, read registers
- Depending on instruction class
 - Use ALU to calculate
 - Arithmetic result
 - Memory address for load/store
 - Branch comparison
 - Access data memory for load/store
 - PC target address or PC + 4

CPU Overview

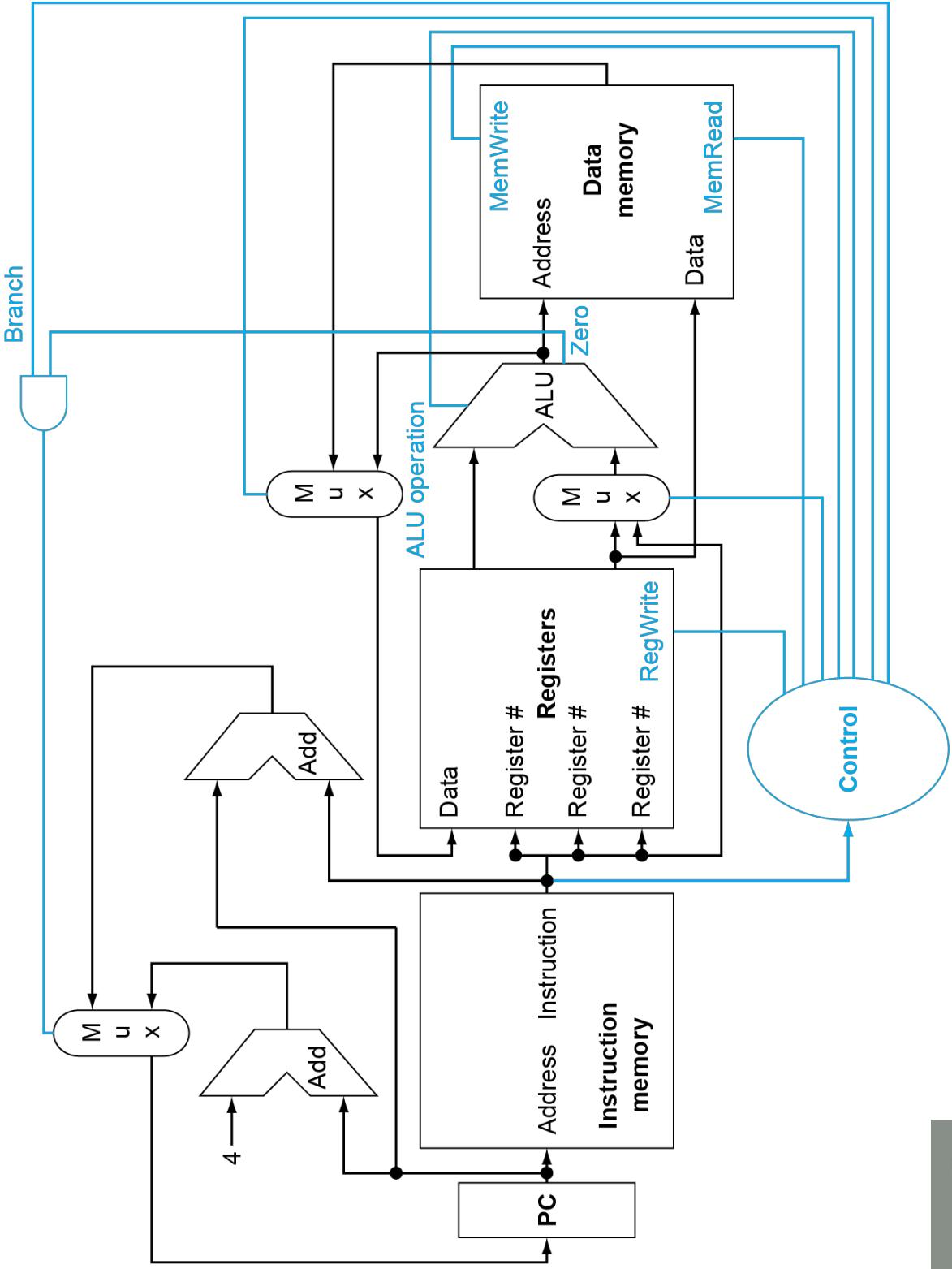


Multiplexers

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

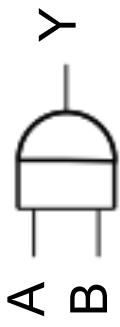
State (sequential) elements

Store information

Combinational Elements

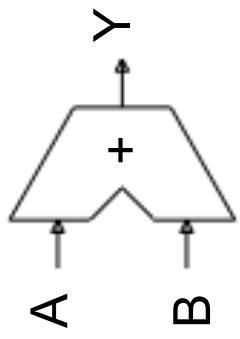
AND-gate

$$Y = A \& B$$



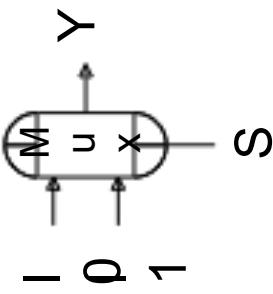
Adder

$$Y = A + B$$



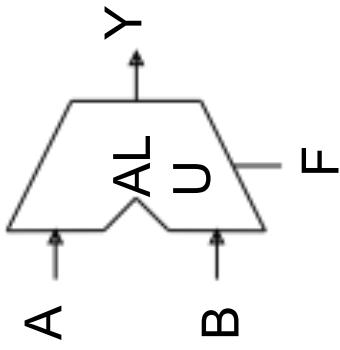
Multiplexer

$$Y = S ? I_1 : I_0$$



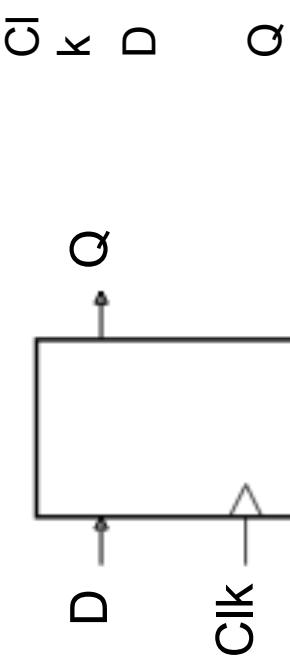
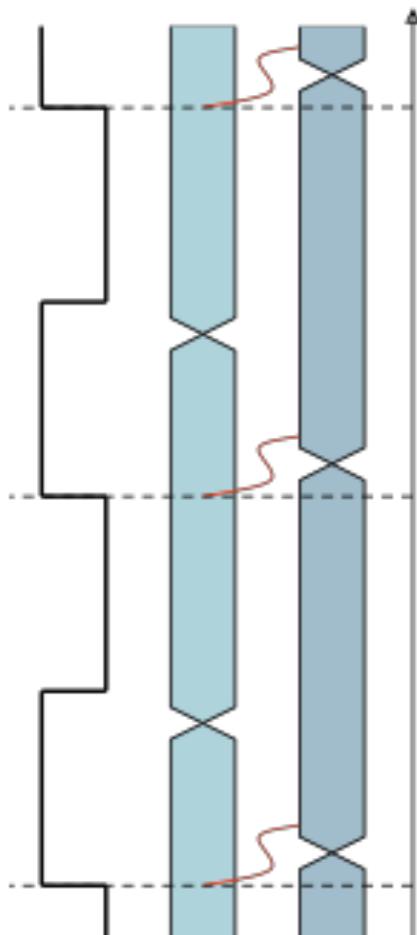
Arithmetic/Logic Unit

$$Y = F(A, B)$$



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

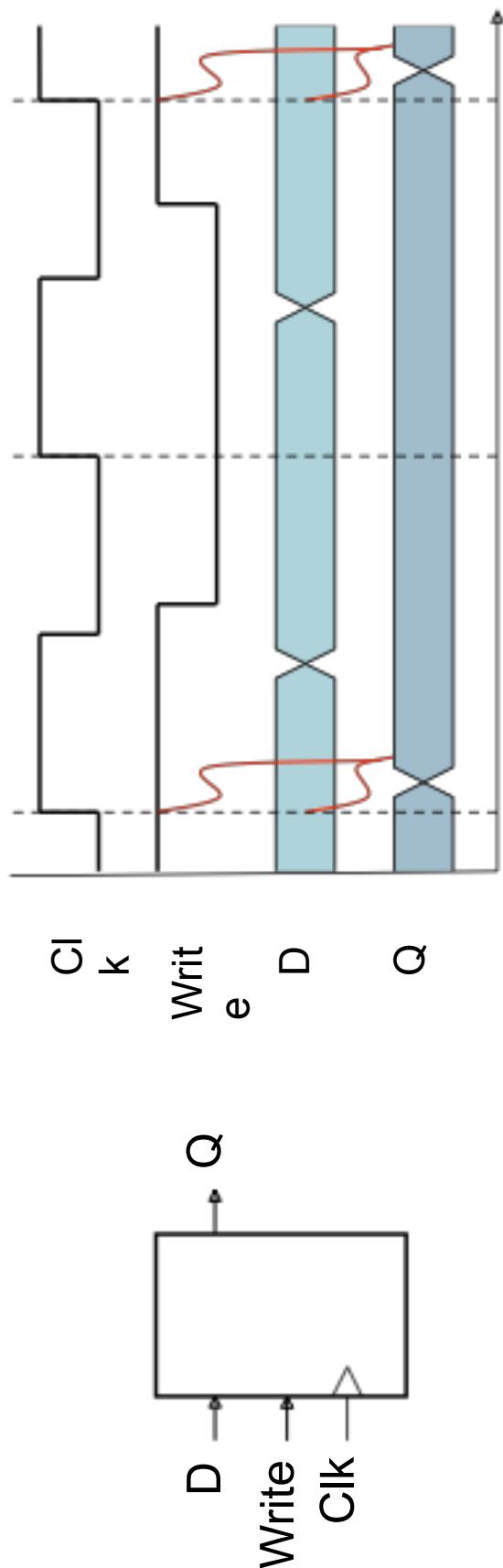


Sequential Elements

Register with write control

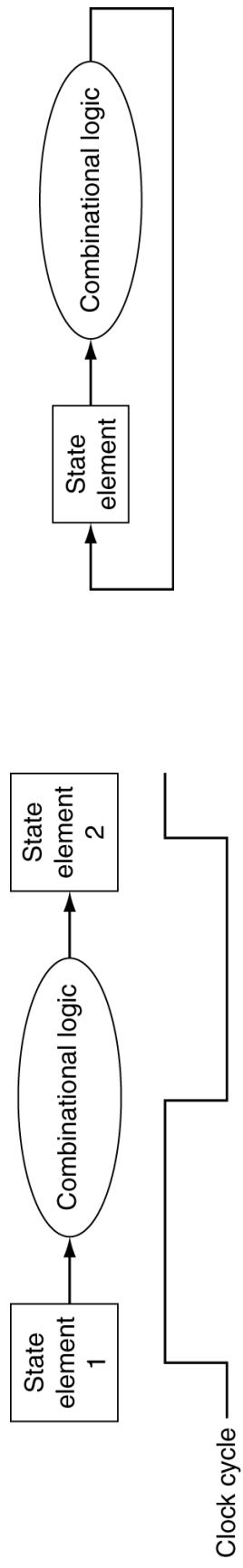
Only updates on clock edge when write control input is 1

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

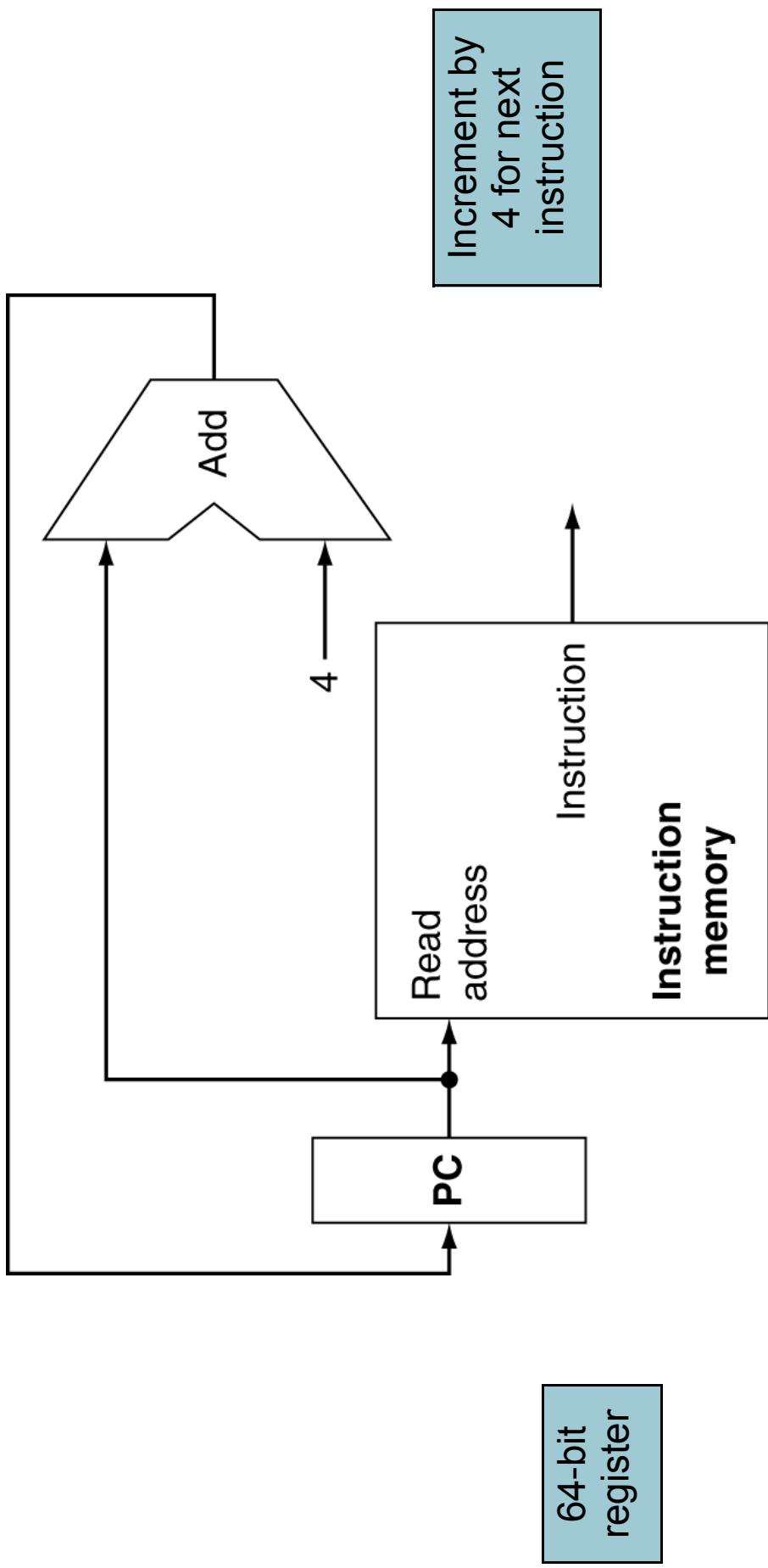
Elements that process data and addresses
in the CPU

Registers, ALUs, mux's, memories, ...

We will build a RISC-V datapath
incrementally

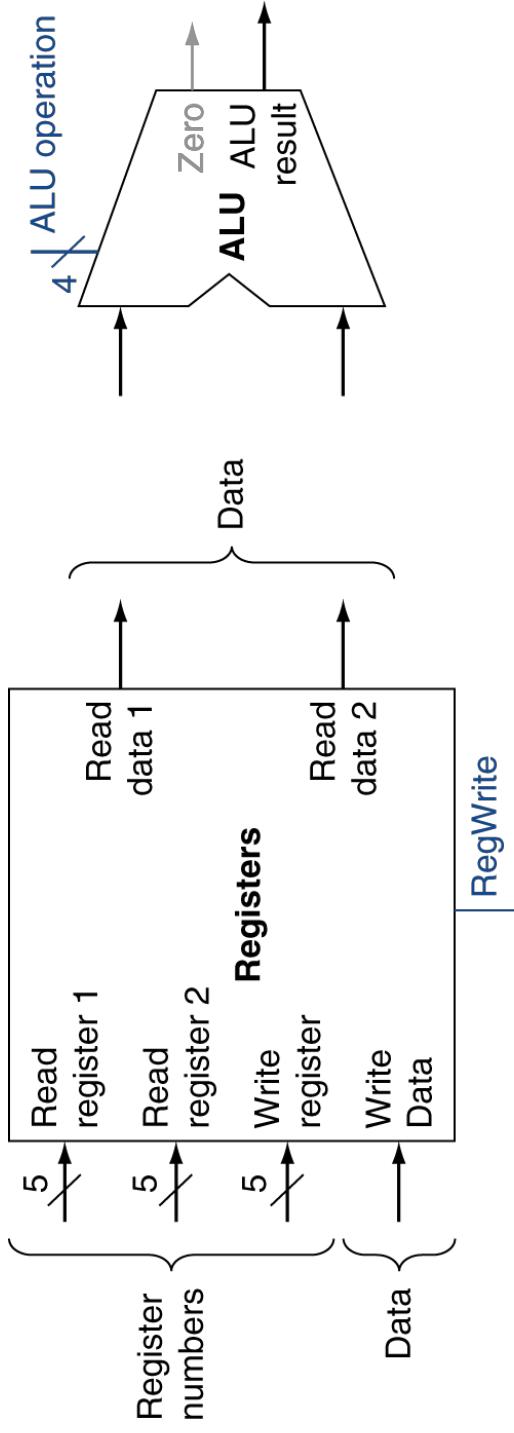
Refining the overview design

Instruction Fetch



R-Format Instructions

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



a. Registers

b. ALU

Load/Store Instructions

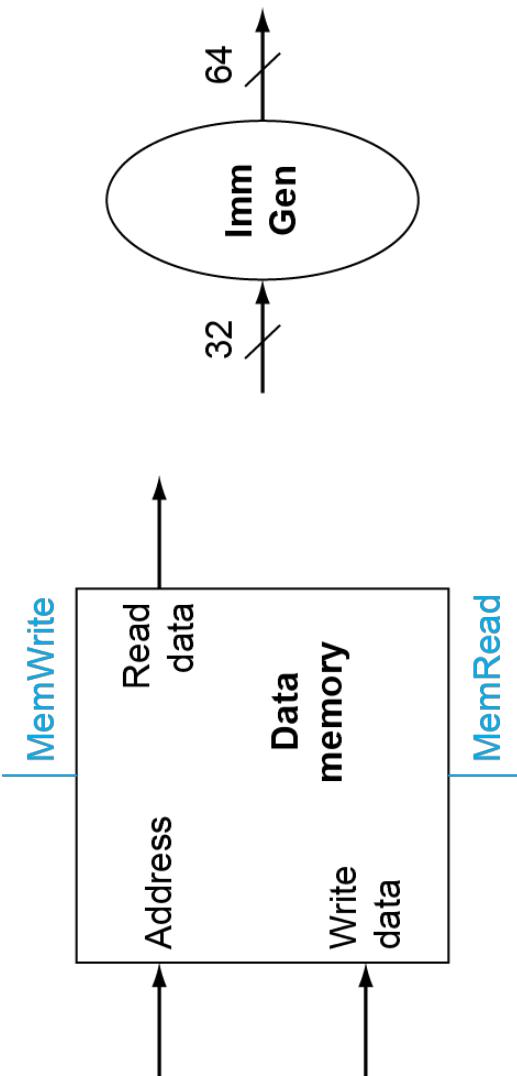
Read register operands

Calculate address using 12-bit offset

Use ALU, but sign-extend offset

Load: Read memory and update register

Store: Write register value to memory



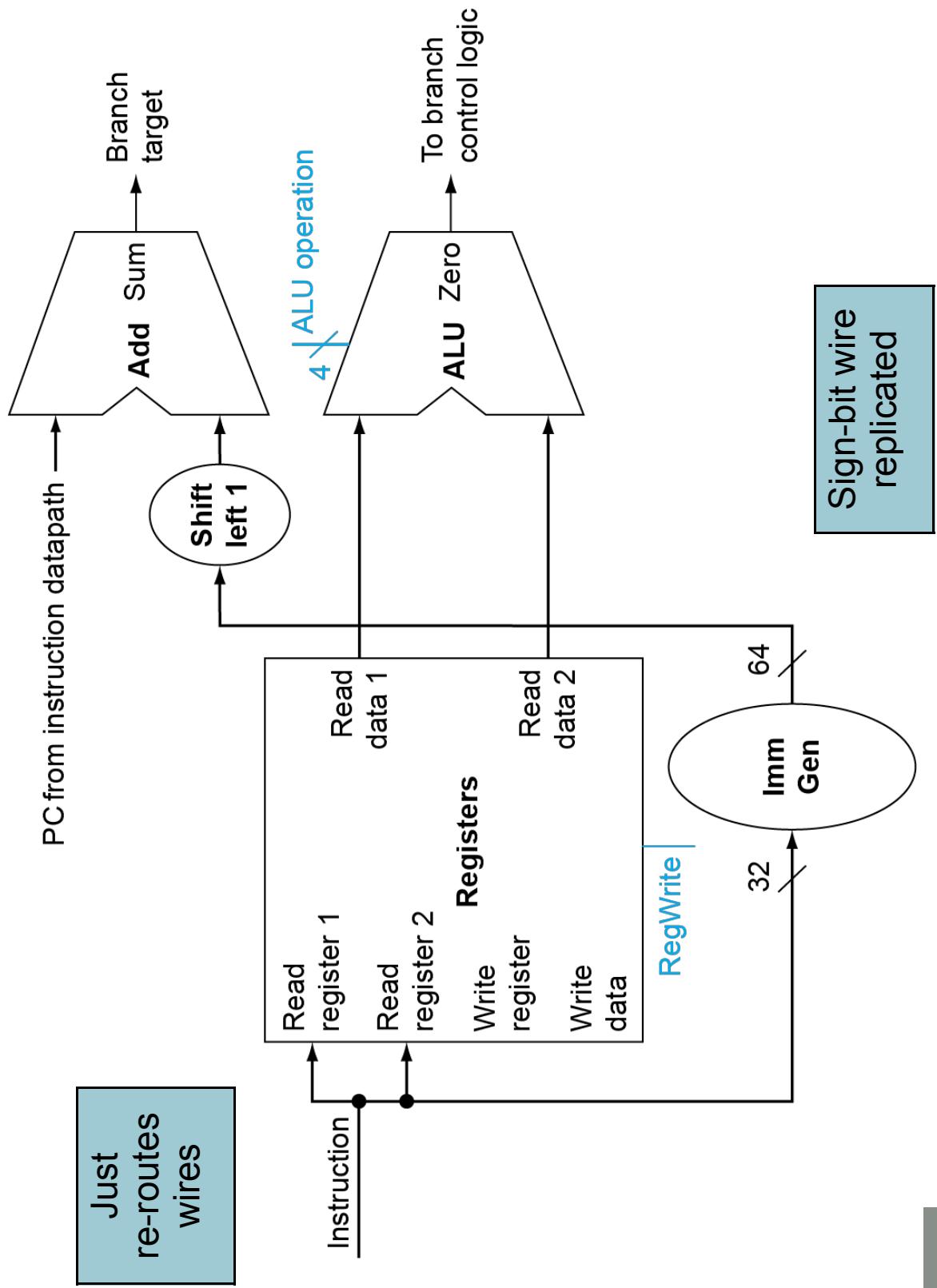
a. Data memory unit

b. Immediate generation unit

Branch Instructions

- 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

Branch Instructions

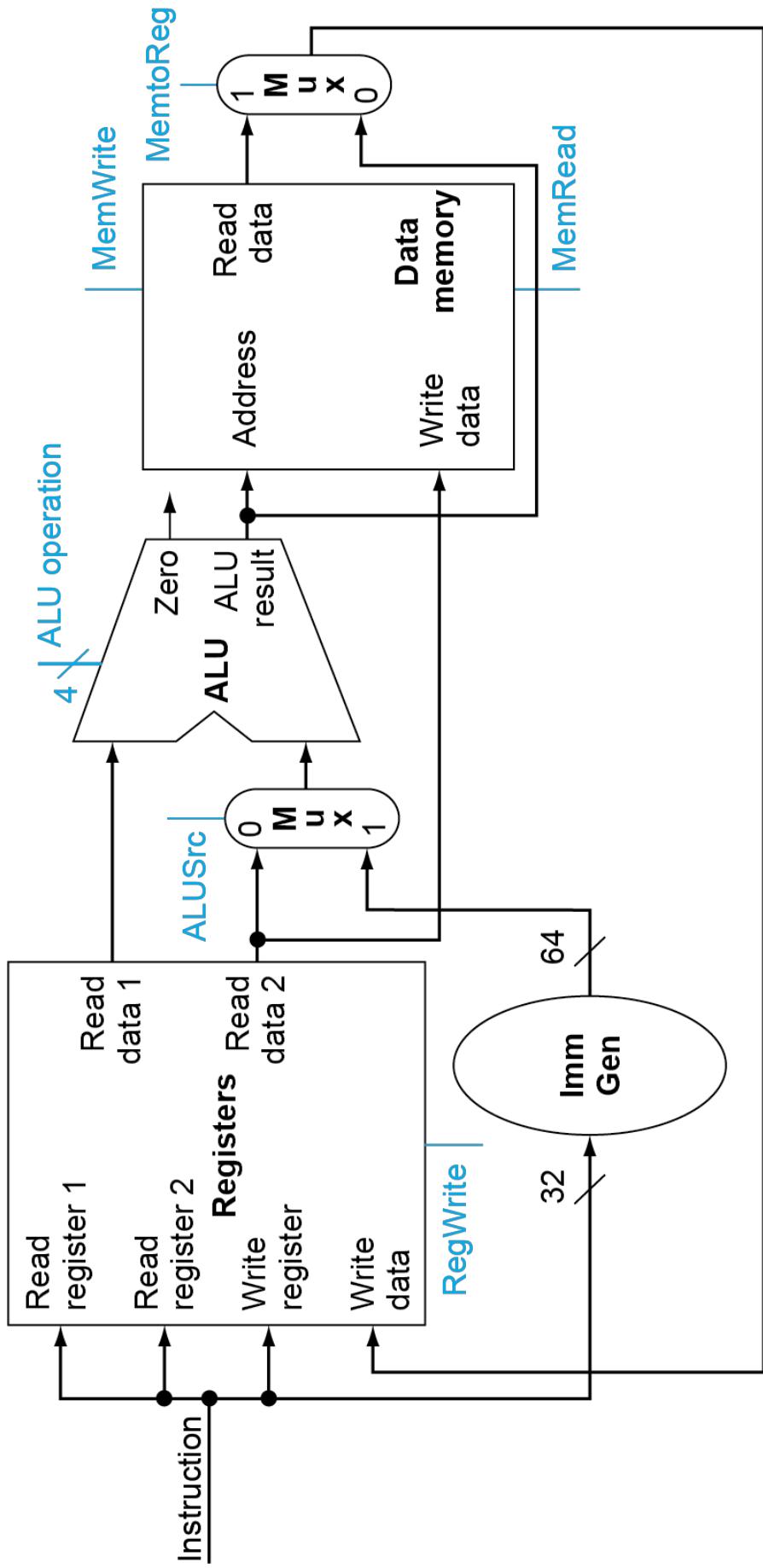


Composing the Elements

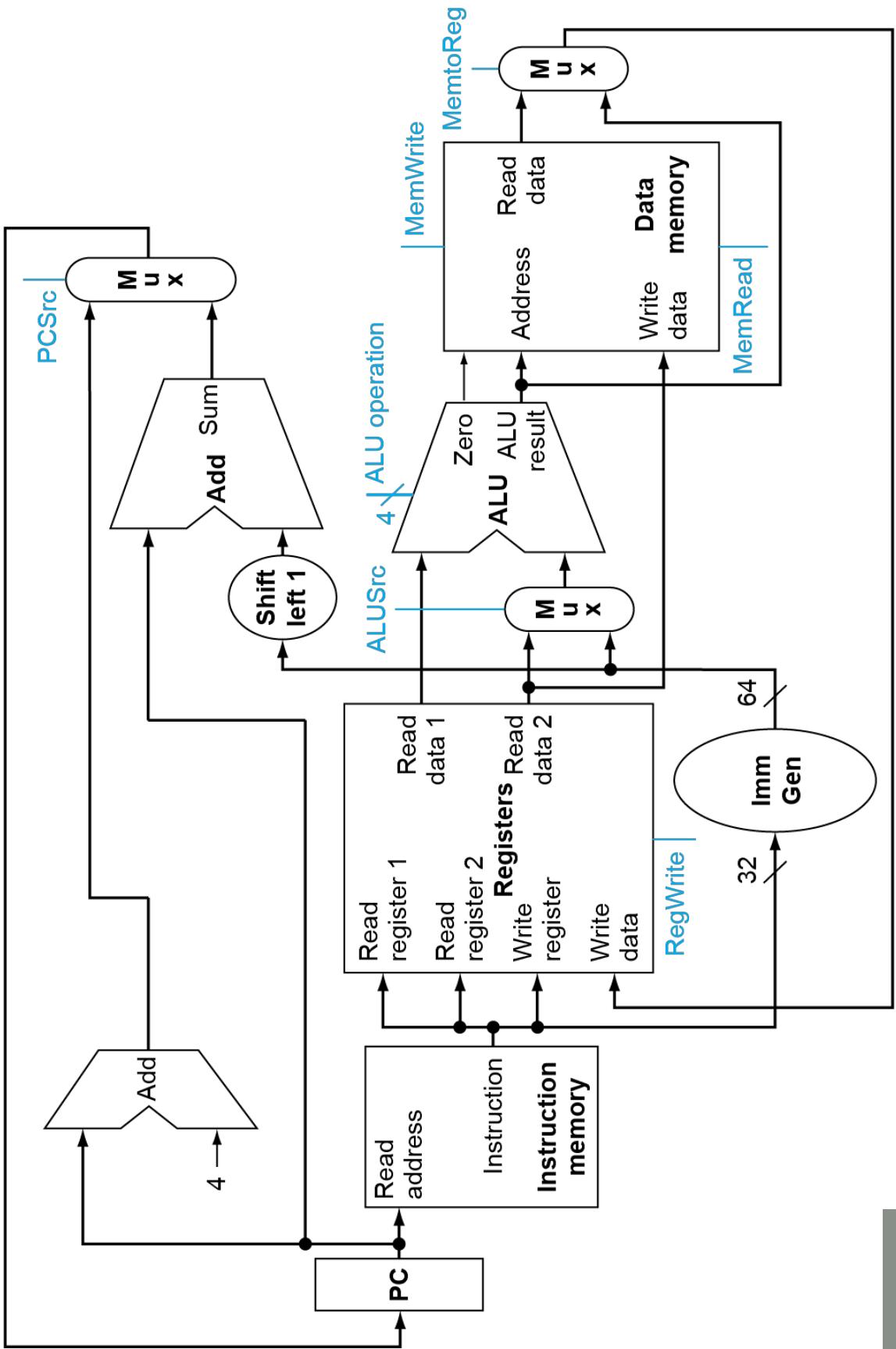
- First-cut data path does an instruction in one clock cycle
 - Each datapath element can only do one function at a time
 - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions



R-Type/Load/Store Datapath



Full Datapath



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

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
ld	00	load register	XXXXXX	add	0010
sd	00	store register	XXXXXX	add	0010
beq	01	branch on equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001

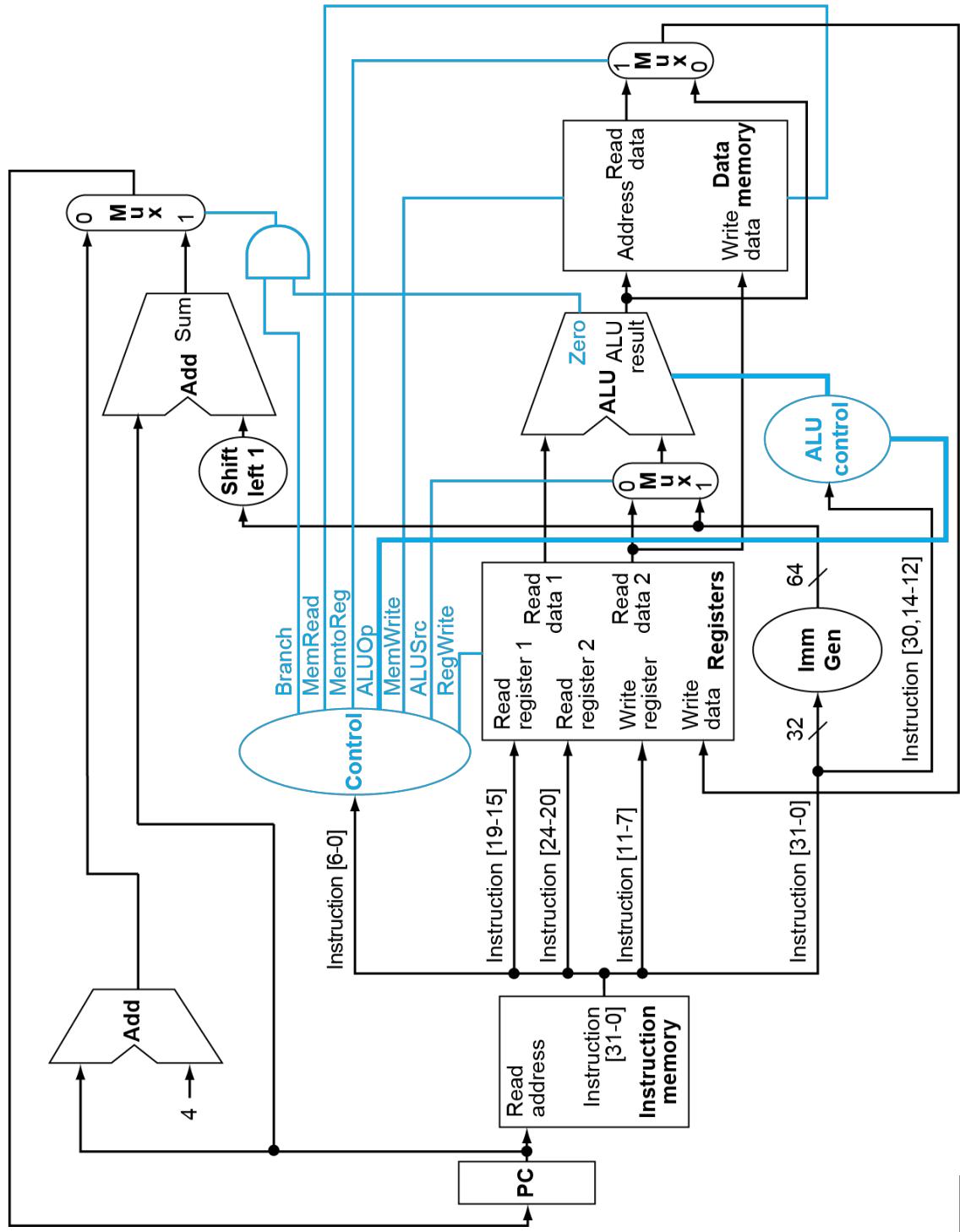
The Main Control Unit

Control signals derived from instruction

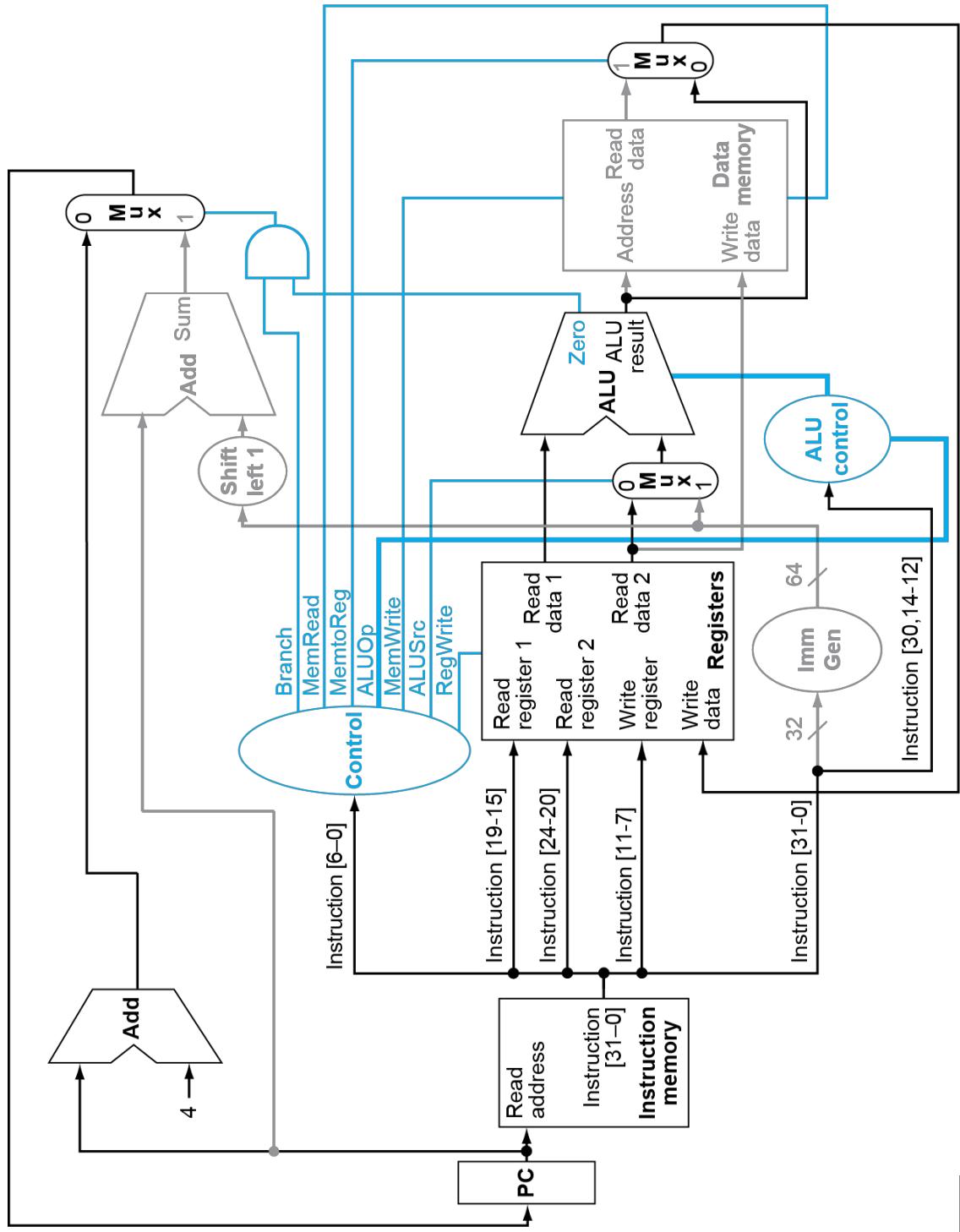
	Name (Bit position)	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode	
(b) I-type	immed[11:0]		rs1	funct3	rd	opcode	
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	

ALUOp	Funct7 field	Funct3 field	Operation						
	I[31]	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	1	X	X	X	X	X	X	X	0010
1	X	0	0	0	0	0	0	0	0110
1	X	0	1	0	0	0	0	0	0010
1	X	0	0	0	0	0	1	1	0000
1	X	0	0	0	0	0	1	1	0001

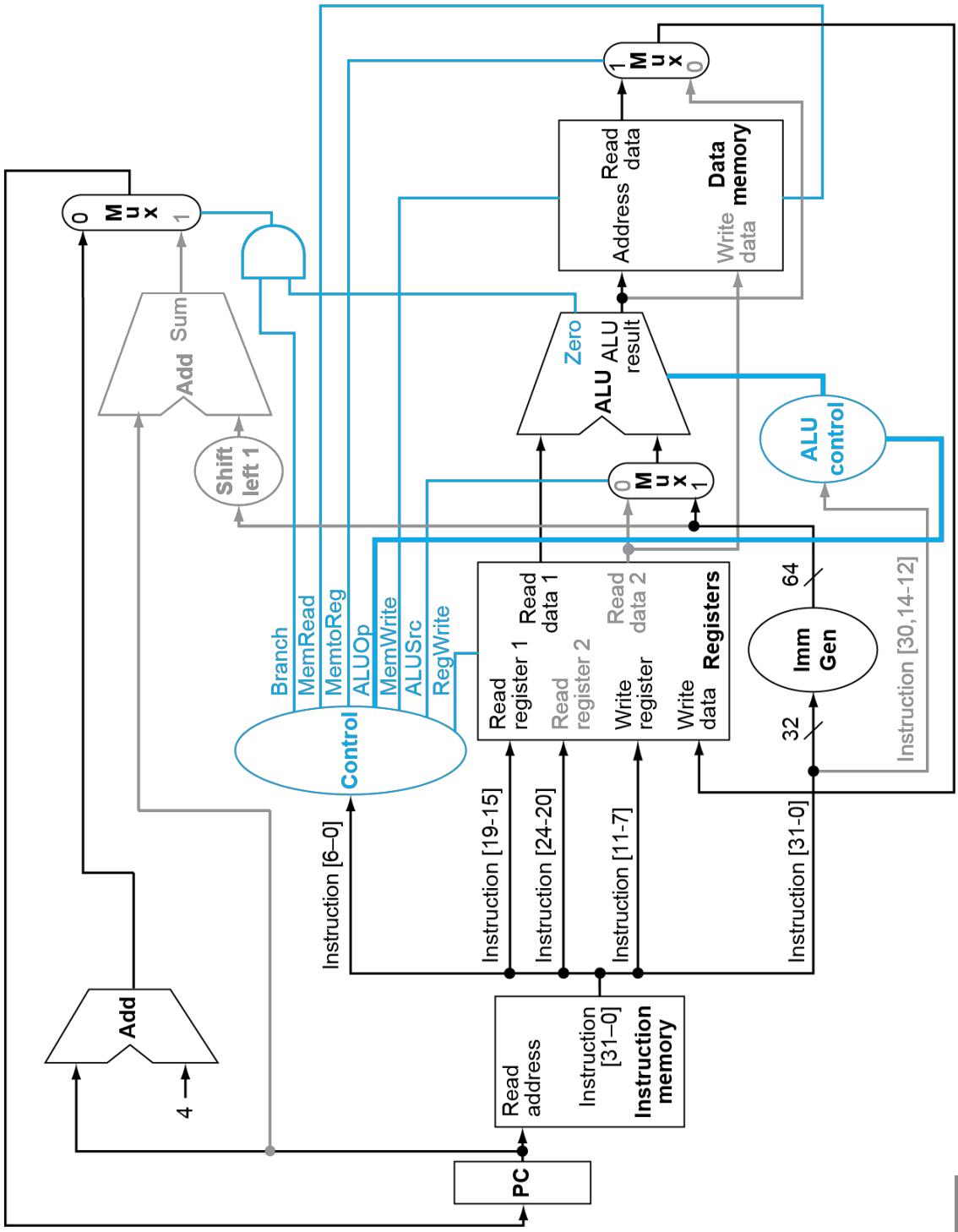
Datapath With Control



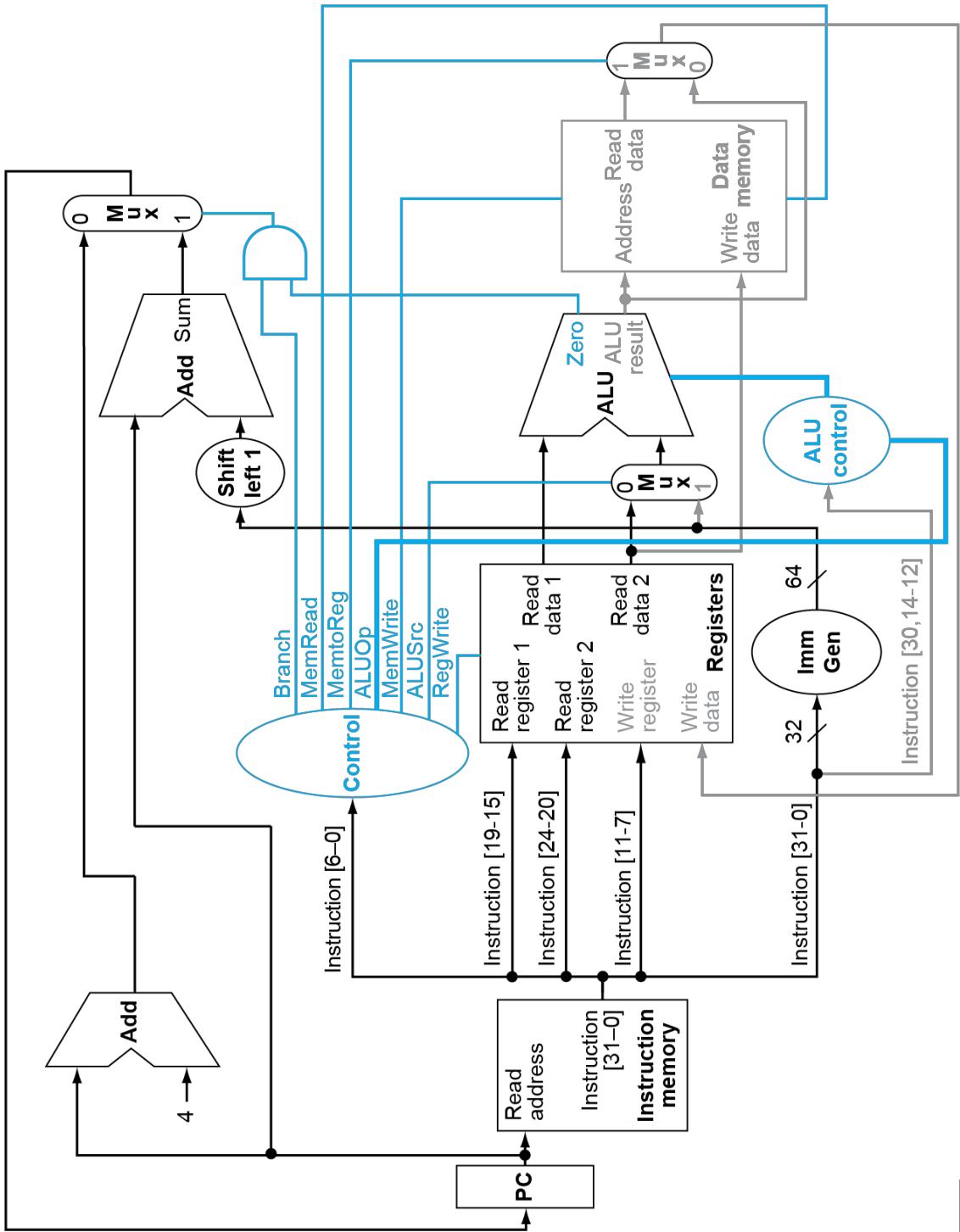
R-Type Instruction



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

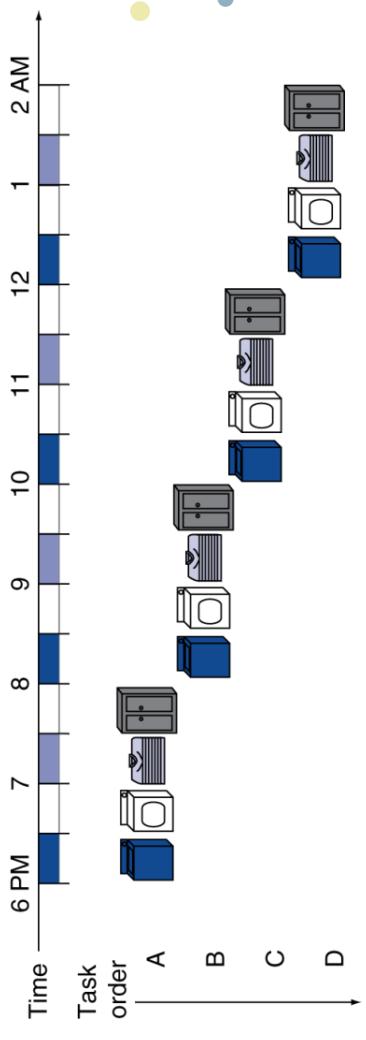
Violates design principle

Making the common case fast

We will improve performance by pipelining

Pipelining Analogy

Pipelined laundry: overlapping execution
Parallelism improves performance



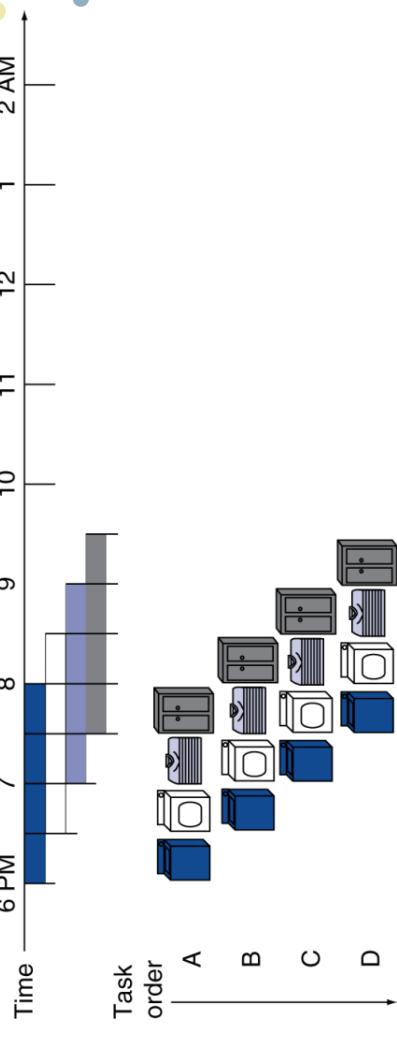
Four loads:

$$\text{Speedup} = 8/3.5 = 2.3$$

Non-stop:

$$\text{Speedup} = 2n/0.5n + 1.5 \approx 4$$

= number of stages



RISC-V Pipeline

Five stages, one step per stage

- IF: Instruction fetch from memory
- ID: Instruction decode & register read
- EX: Execute operation or calculate address
- MEM: Access memory operand
- WB: Write result back to register



Pipeline Performance

Assume time for stages is

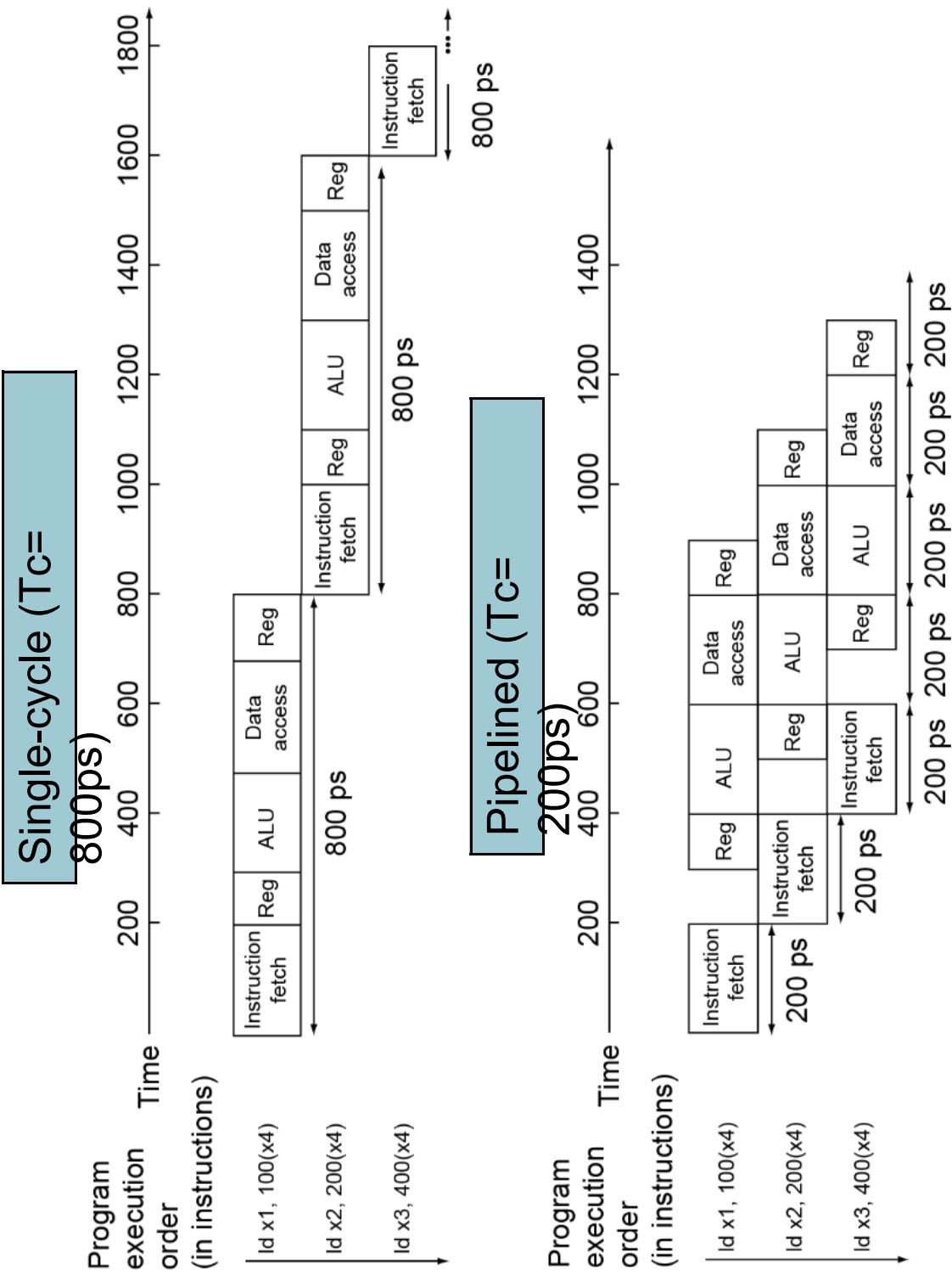
100ps for register read or write

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			600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance



Pipeline Speedup

- If all stages are balanced
 - i.e., all take the same time
 - Time between instructions pipelined
 - = Time between instructions non-pipelined
 - 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 3rd stage, access
memory in 4th 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”

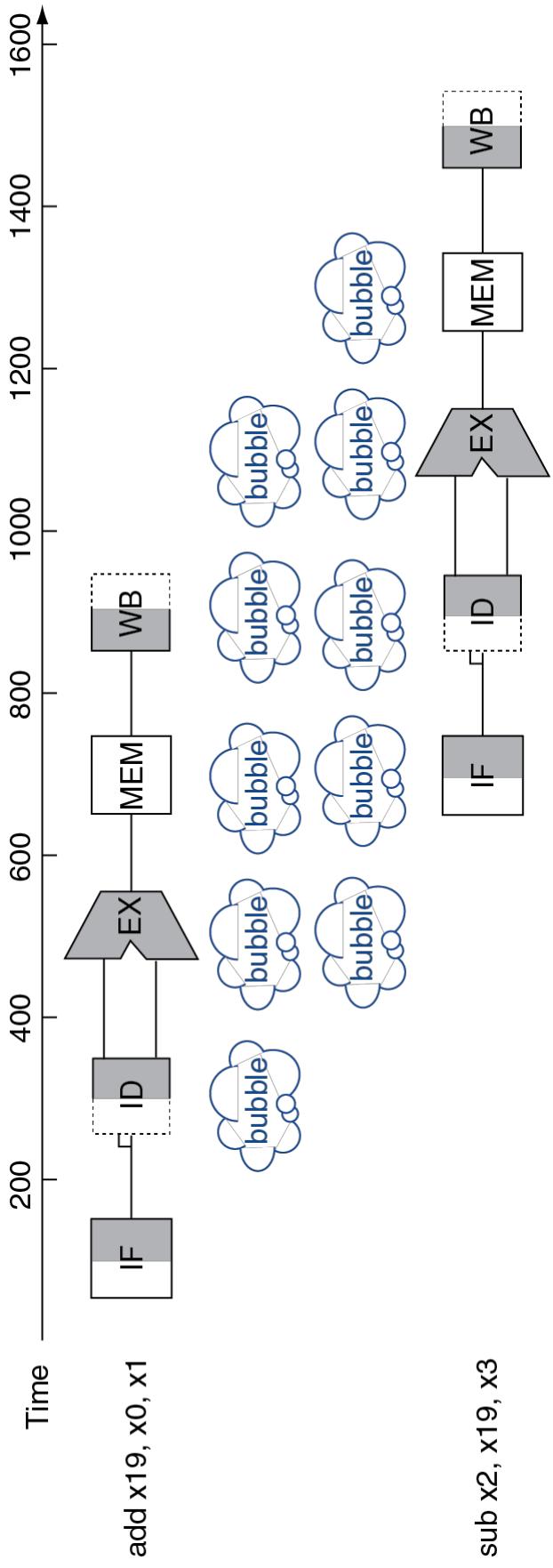
Hence, pipelined datapaths require separate instruction/data memories
Or separate instruction/data caches



Data Hazards

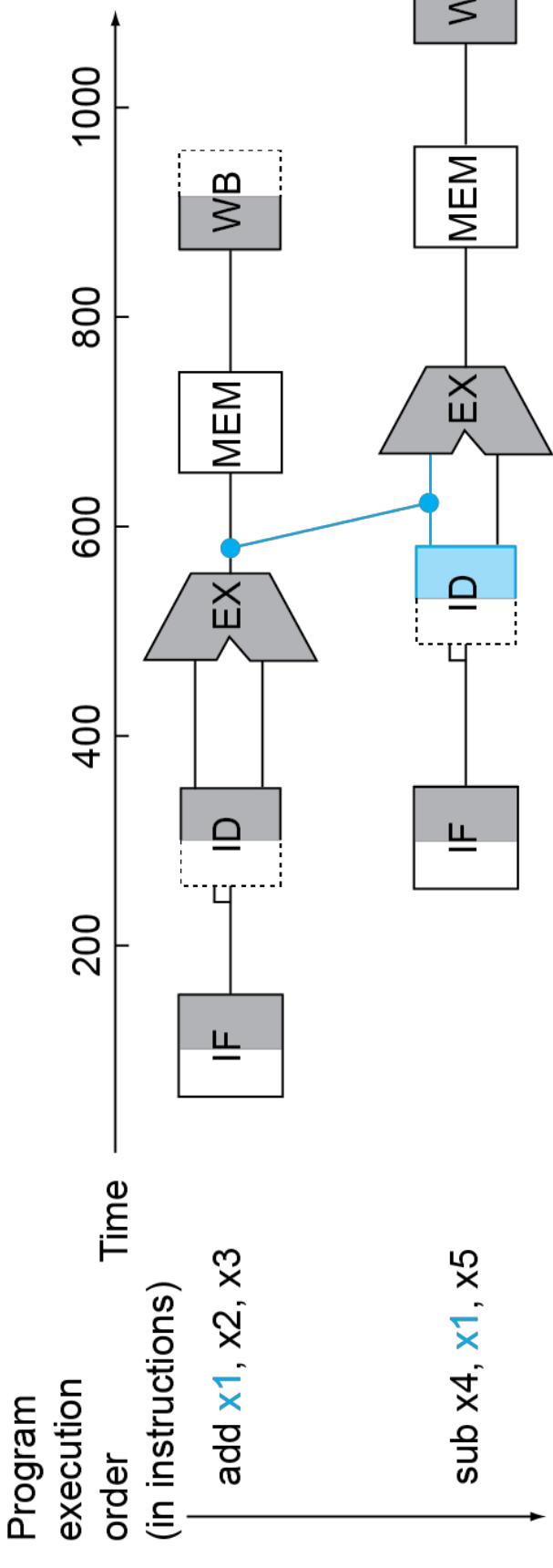
An instruction depends on completion of data access by a previous instruction

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



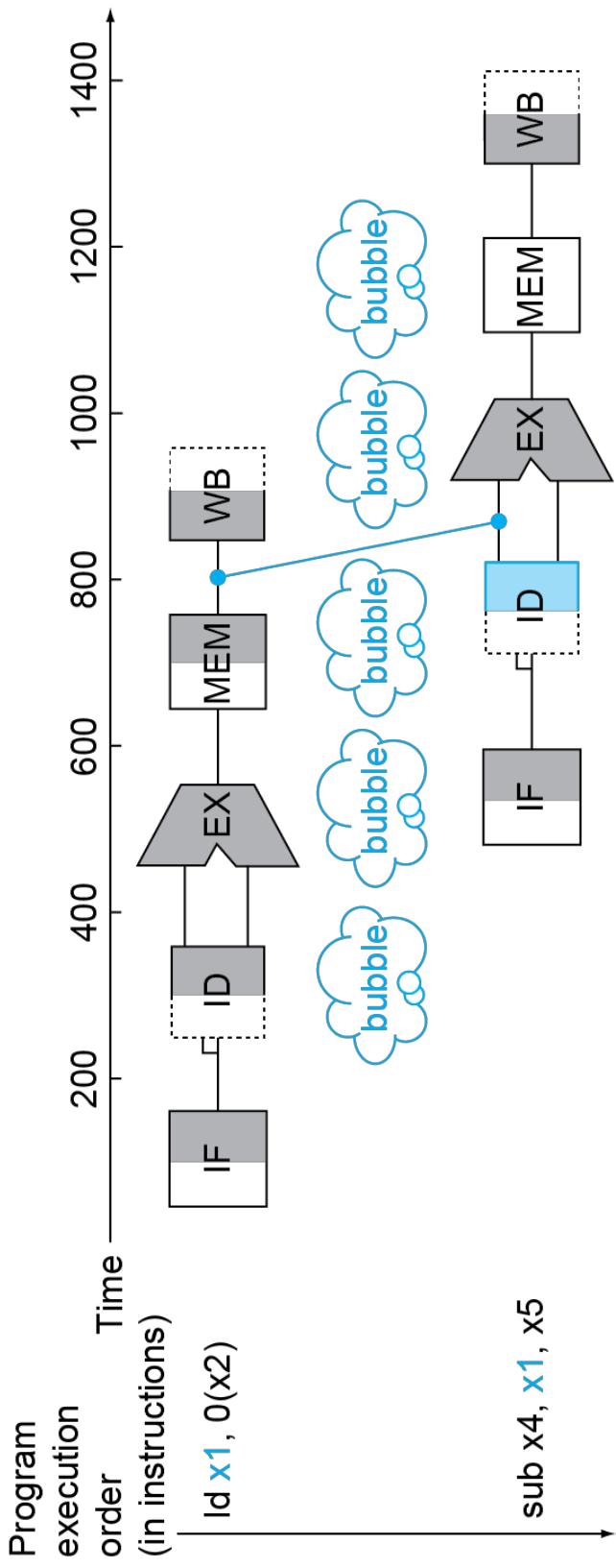
Forwarding (aka Bypassing)

- 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

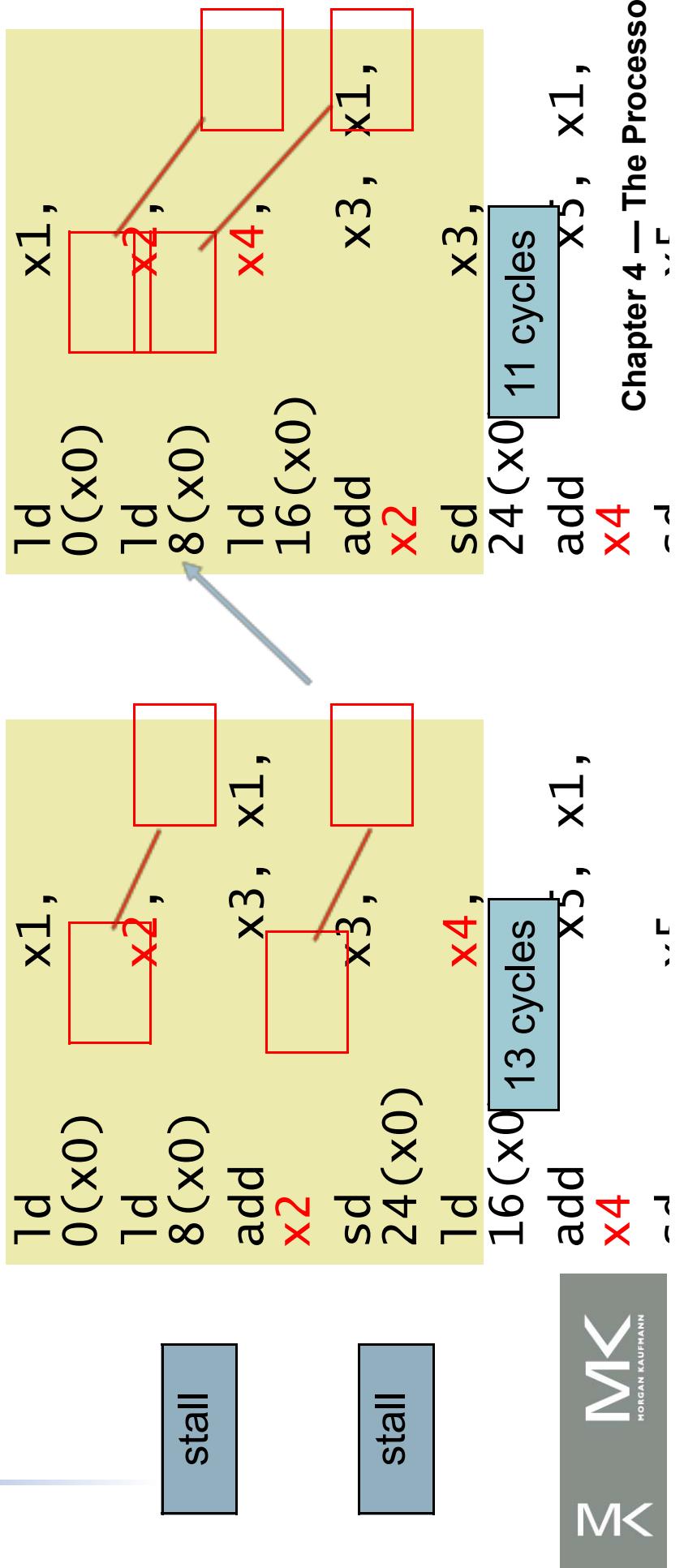
Can't always avoid stalls by forwarding
If value not computed when needed
Can't forward backward in time!



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

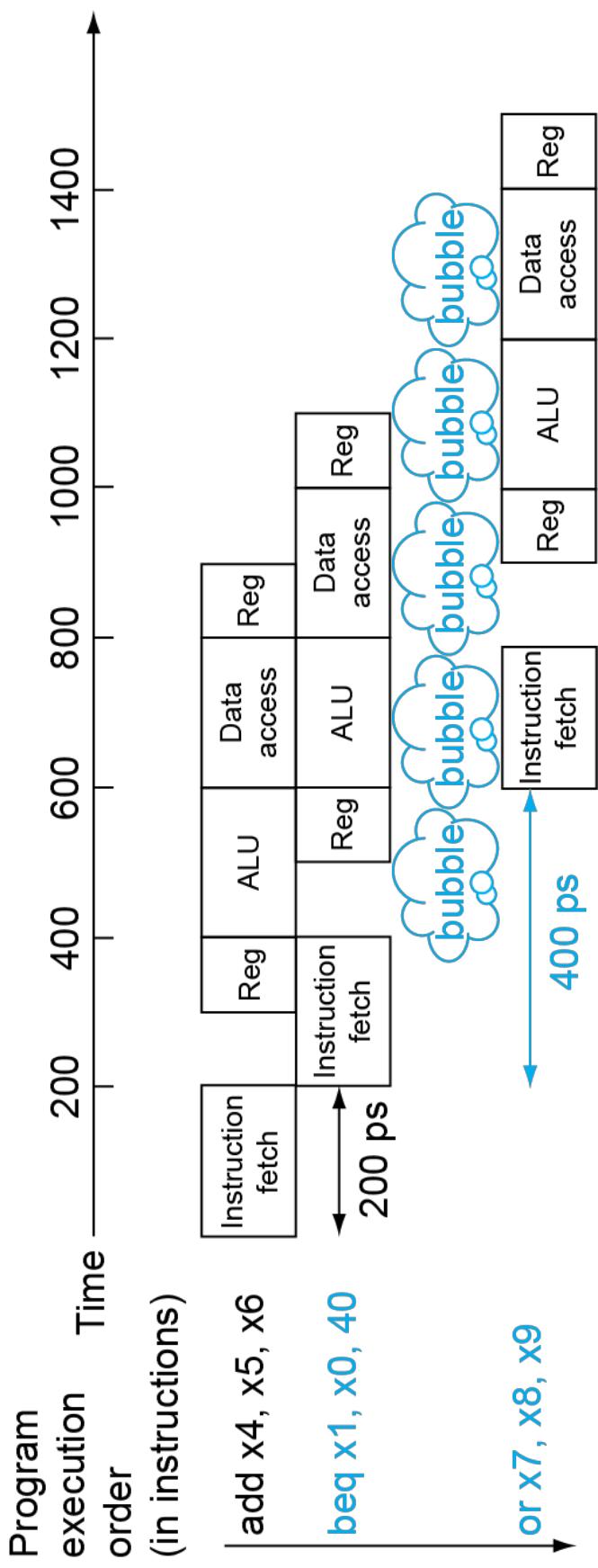
- Branch determines flow of control
- Fetching next instruction depends on branch outcome
- Pipeline can't always fetch correct instruction
- Still working on ID stage of branch

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

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
- Executes multiple instructions in parallel
- Each instruction has the same latency

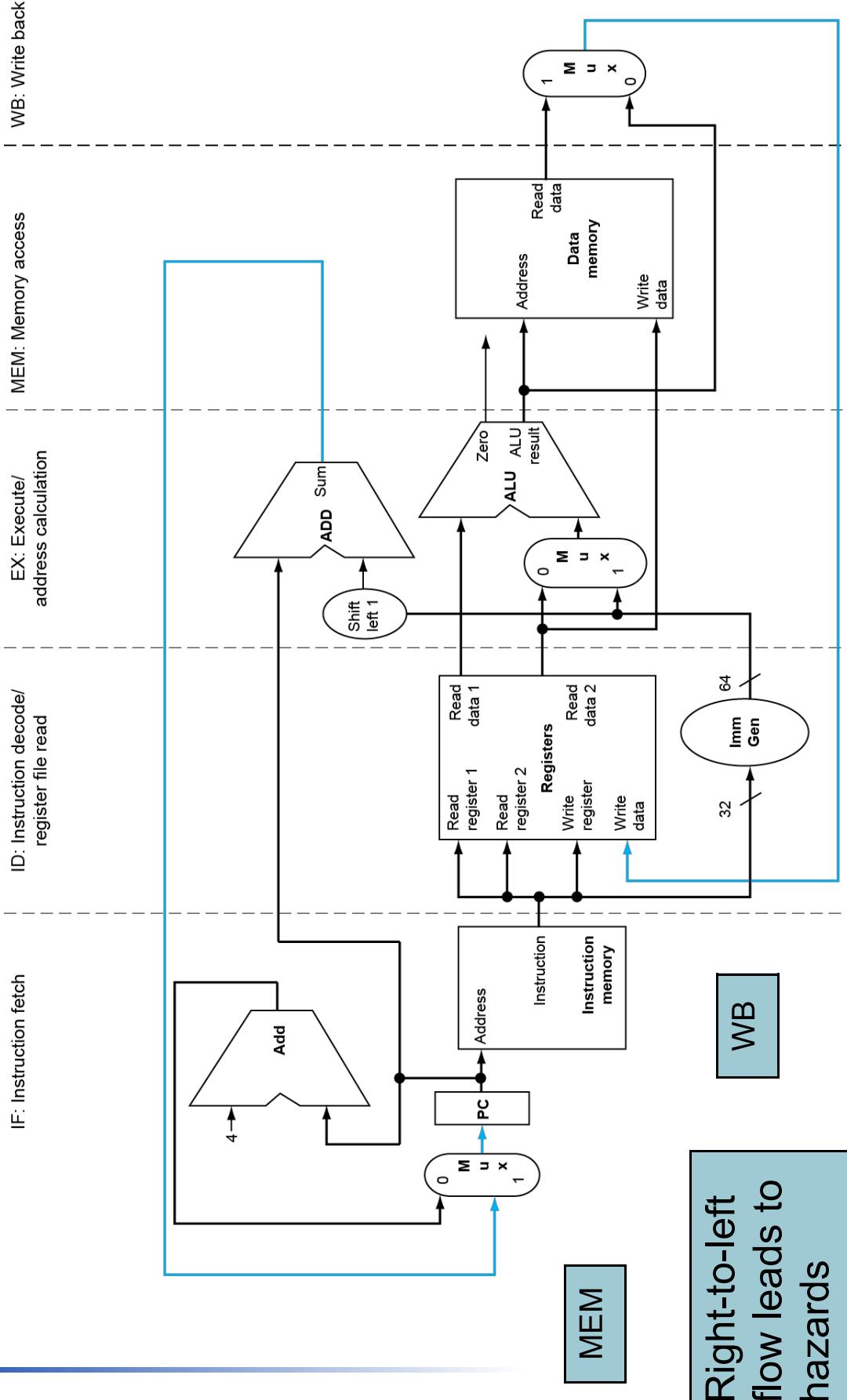
Subject to hazards

Structure, data, control

Instruction set design affects complexity of pipeline implementation

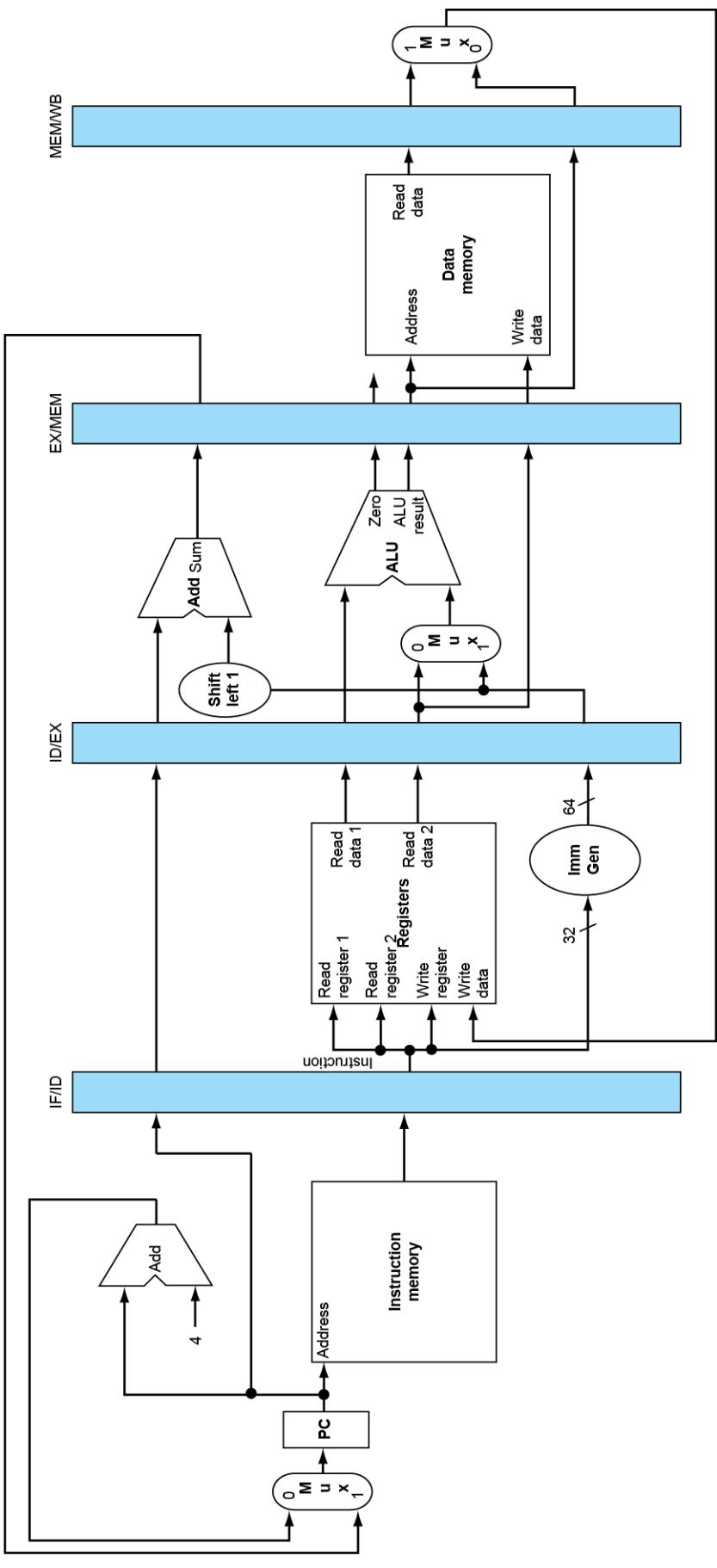


RISC-V Pipelined Datapath



Pipeline registers

Need registers between stages
To hold information produced in previous cycle



Pipeline Operation

Cycle-by-cycle flow of instructions through the pipelined datapath

“Single-clock-cycle” pipeline diagram

Shows pipeline usage in a single cycle

Highlight resources used

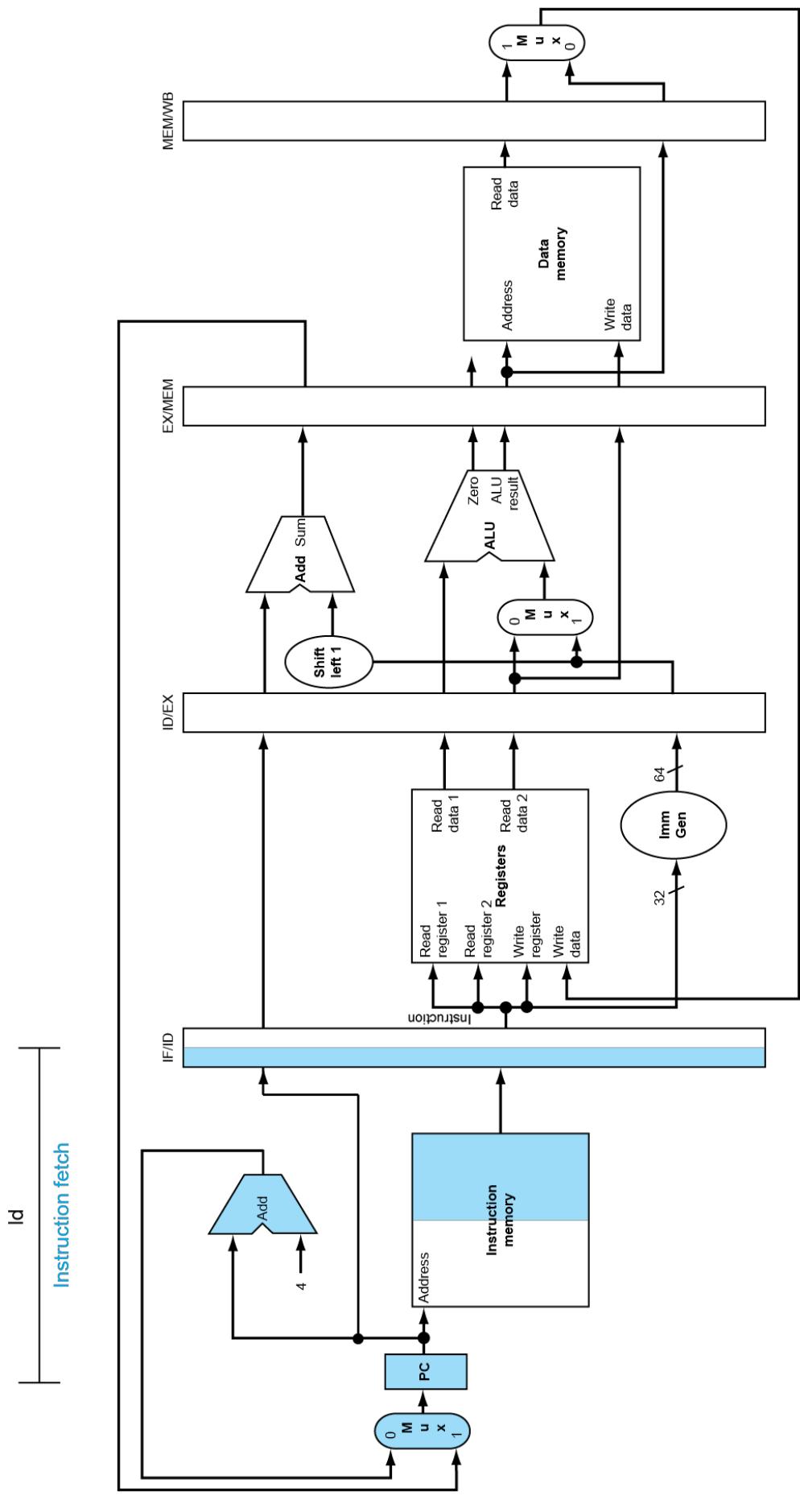
c.f. “multi-clock-cycle” diagram

Graph of operation over time

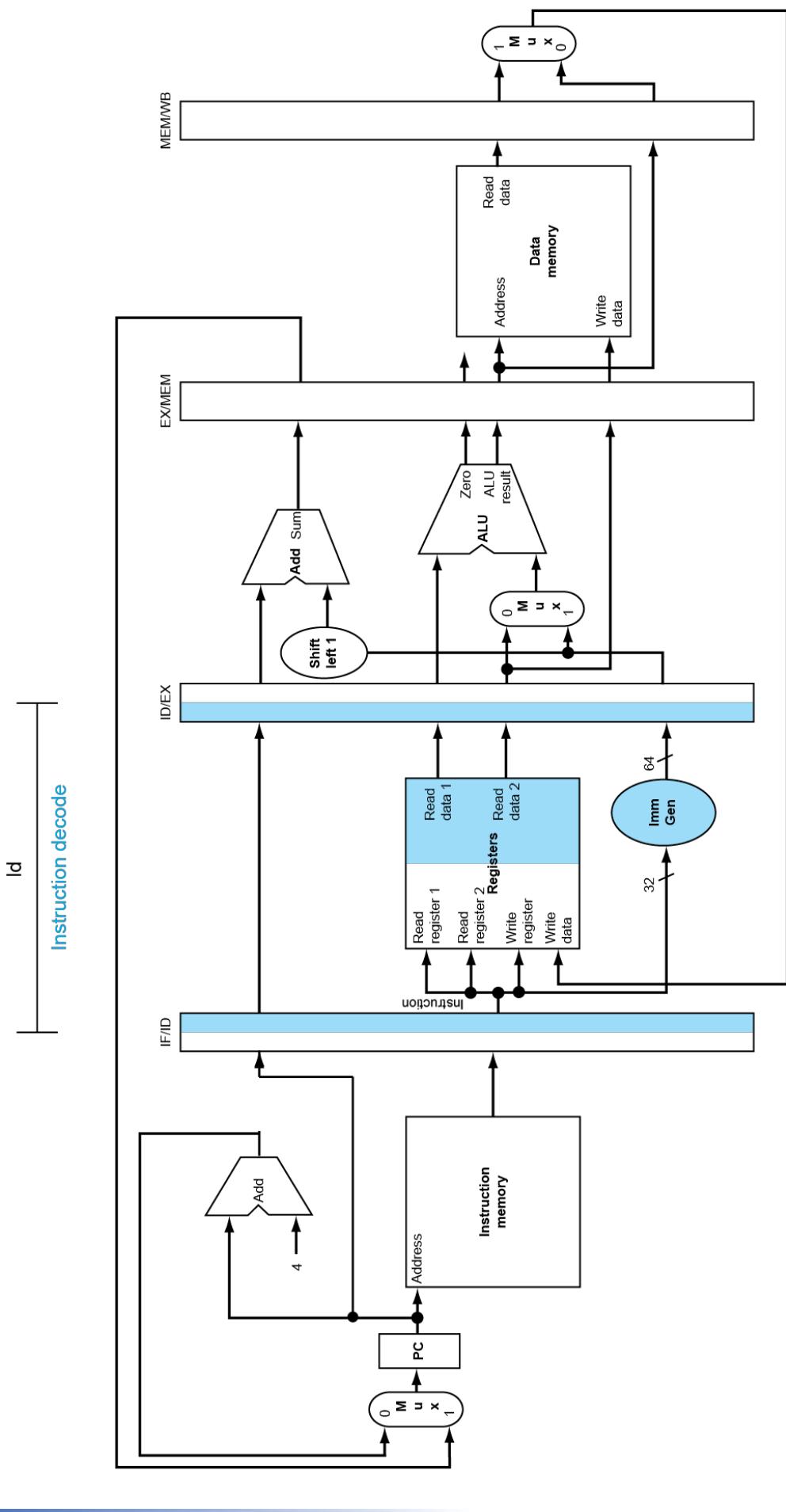
We'll look at “single-clock-cycle” diagrams for load & store



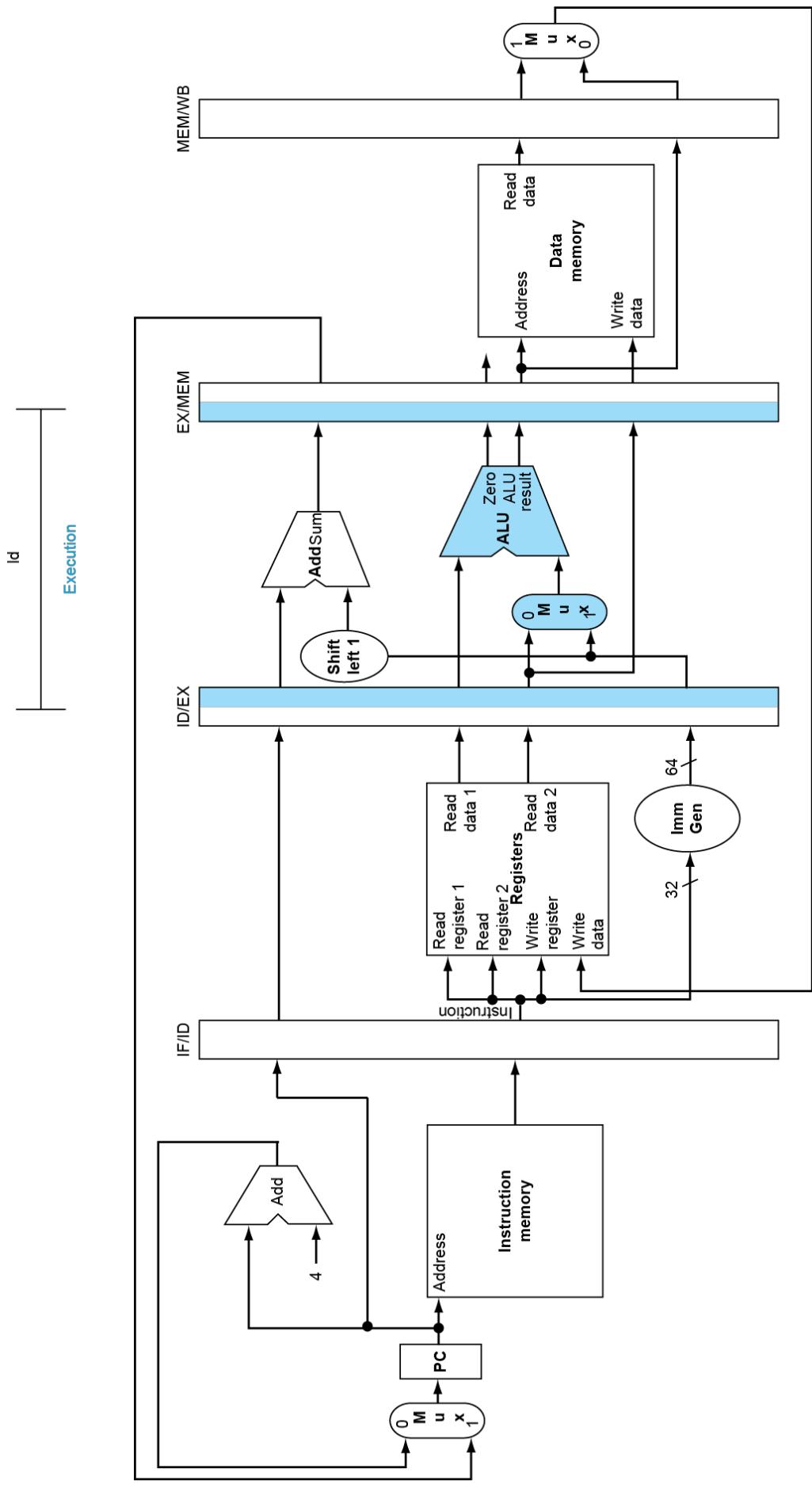
IF for Load, Store, ...



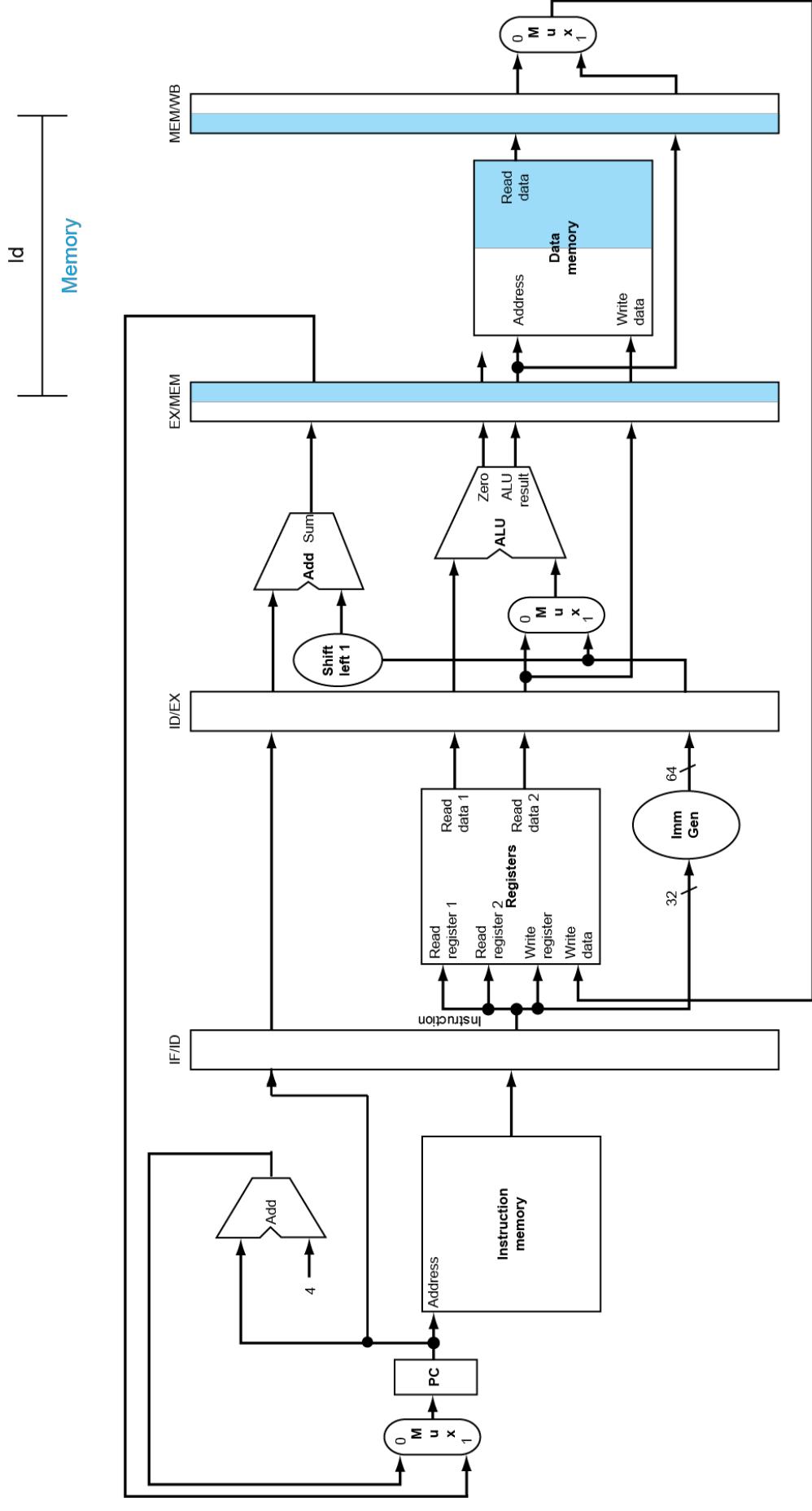
ID for Load, Store, ...



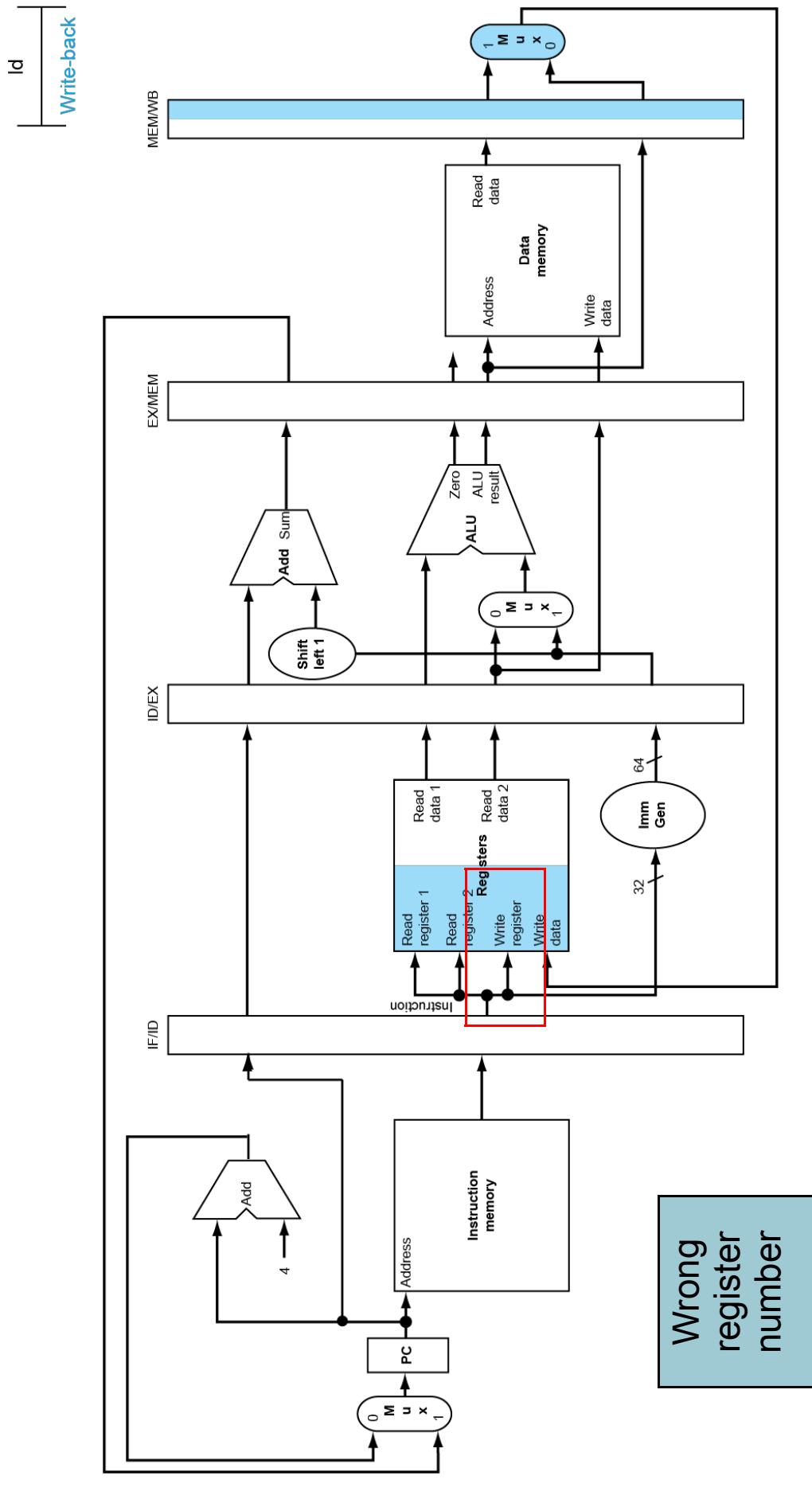
EX for Load



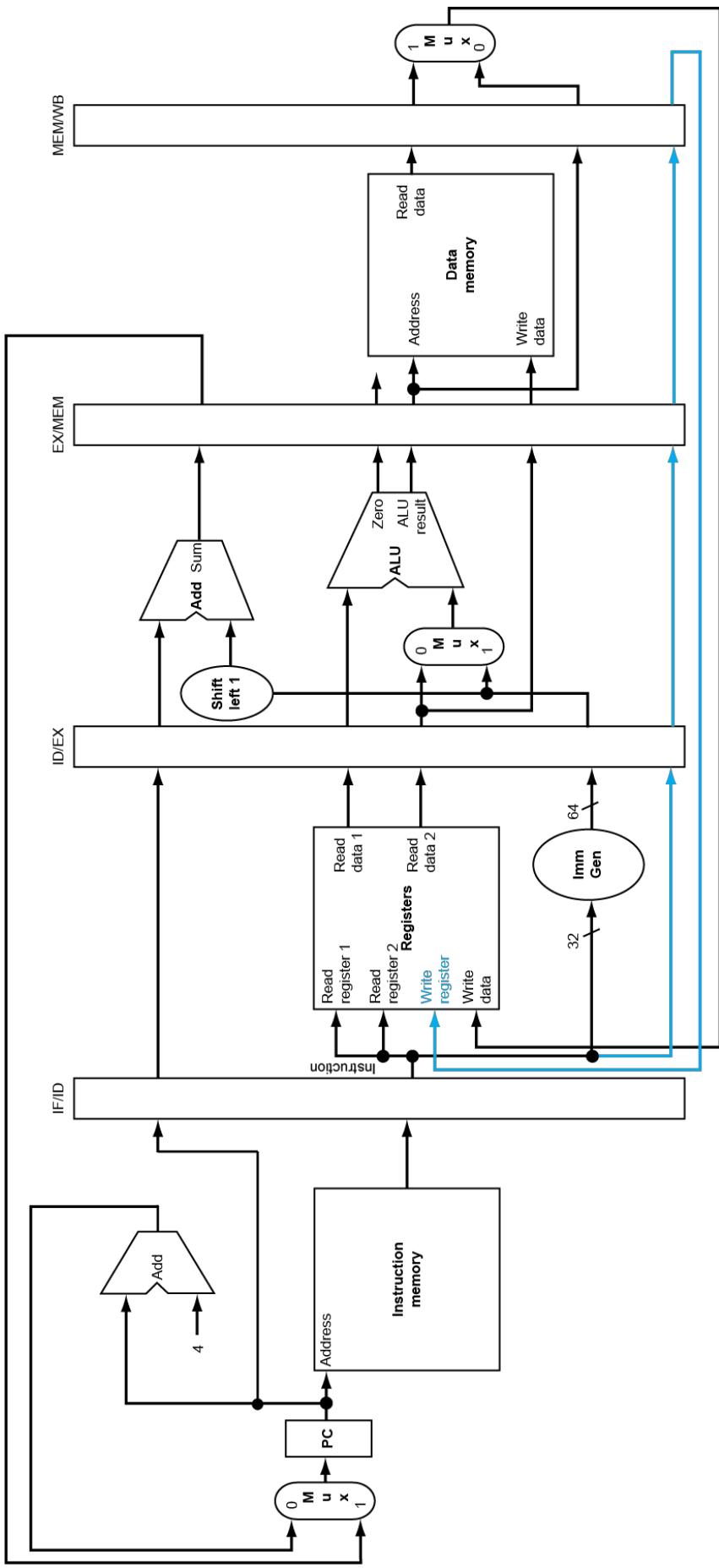
MEM for Load



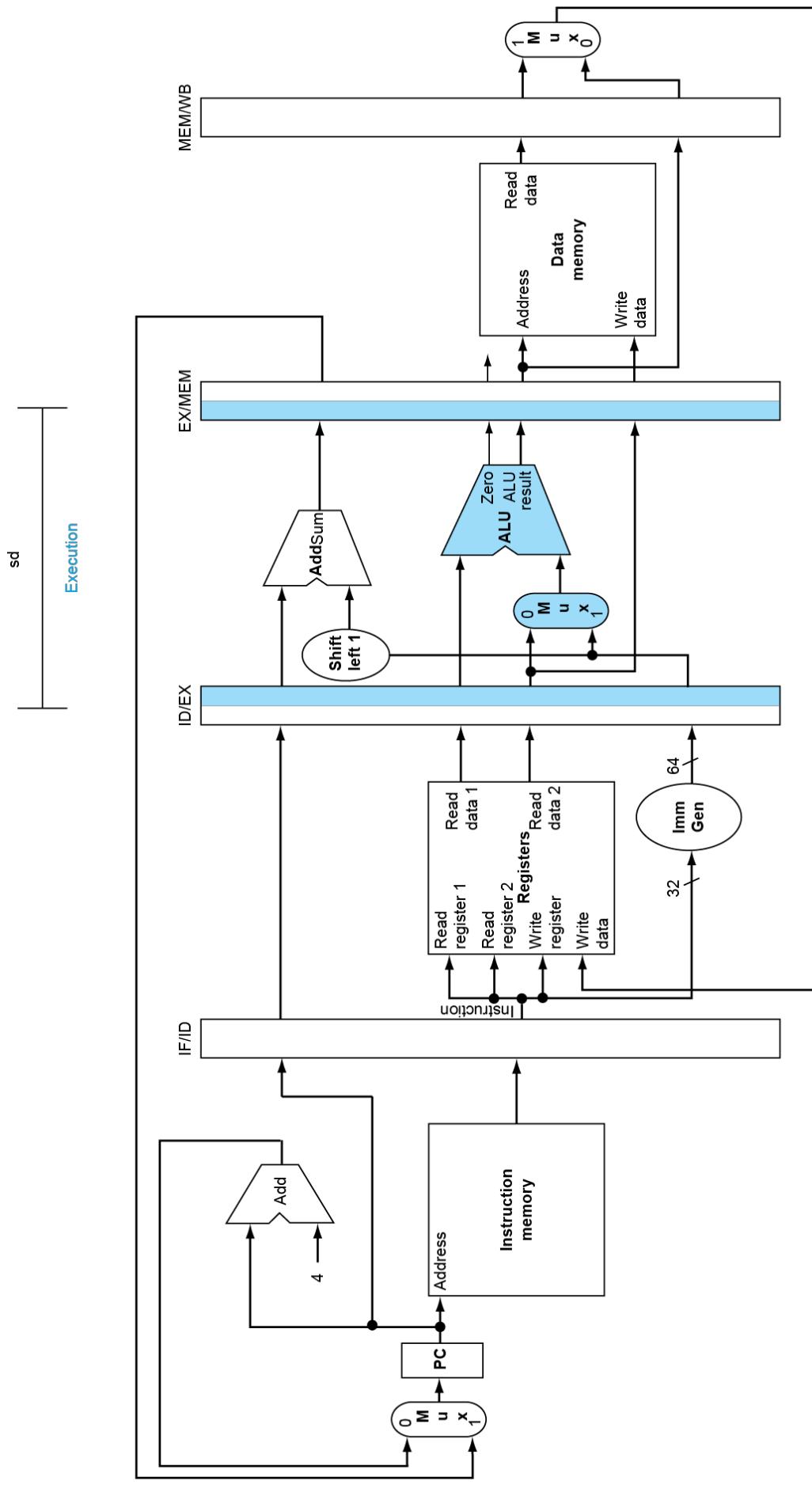
WB for Load



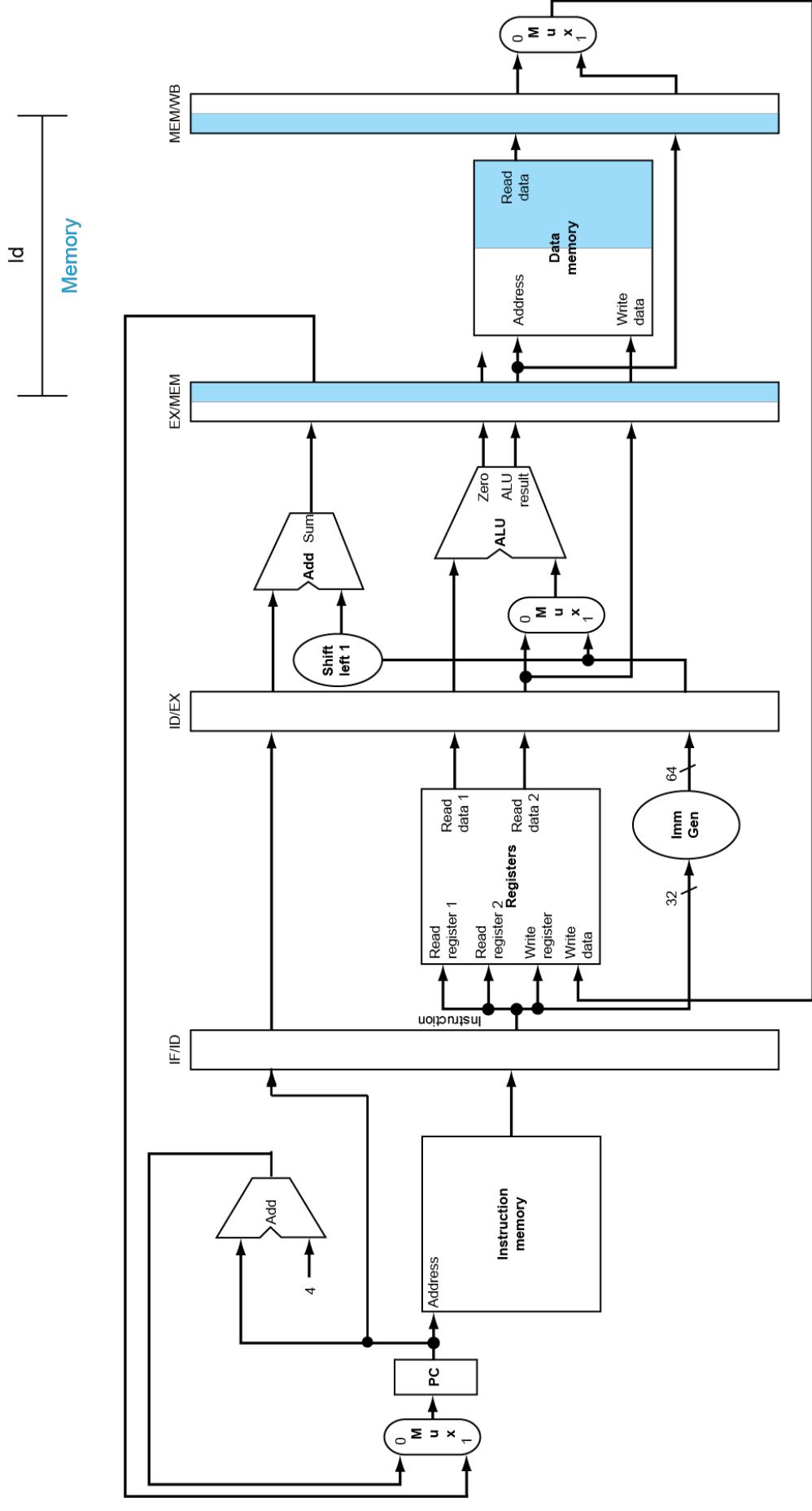
Corrected Datapath for Load



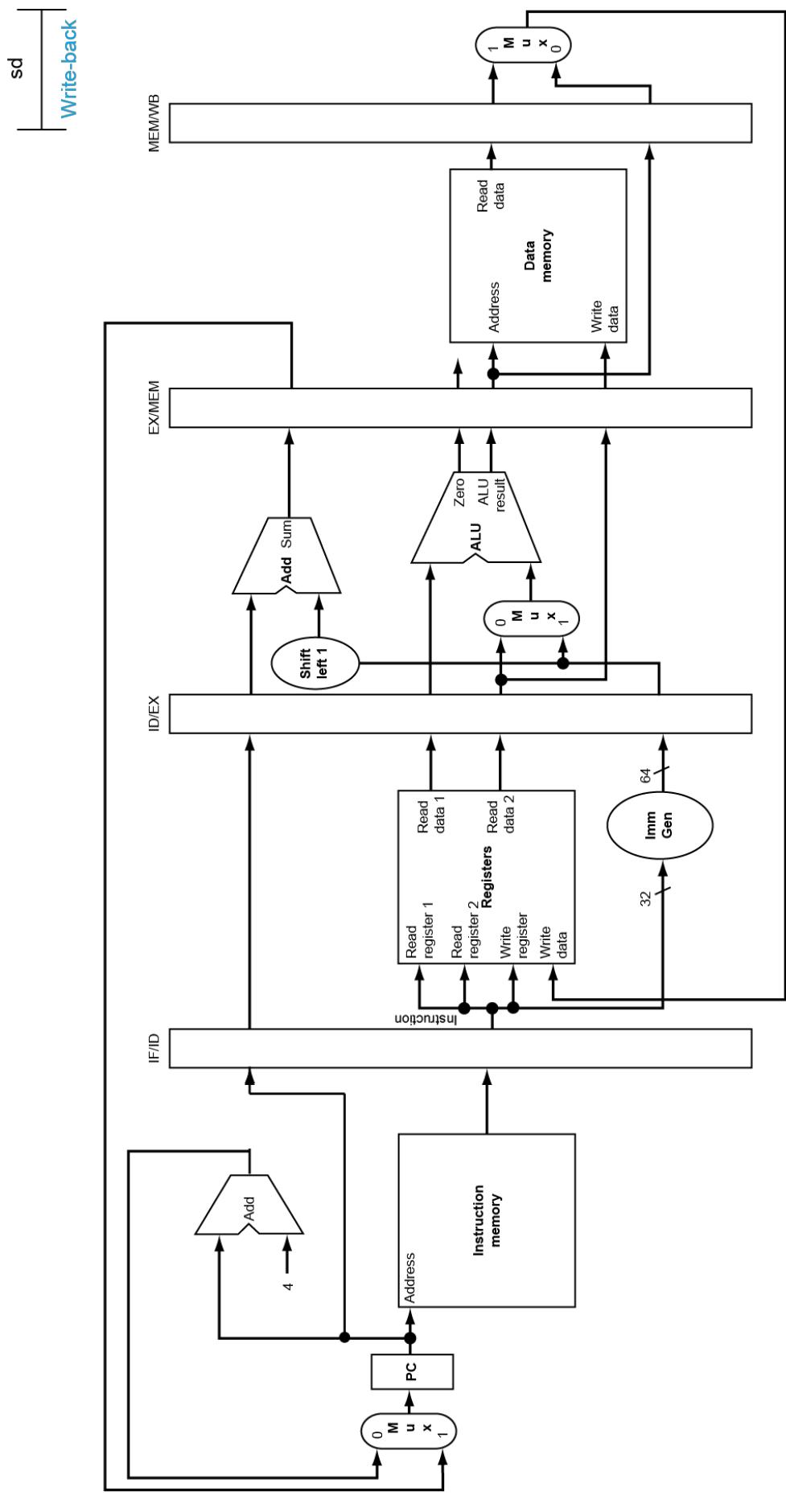
EX for Store



MEM for Store

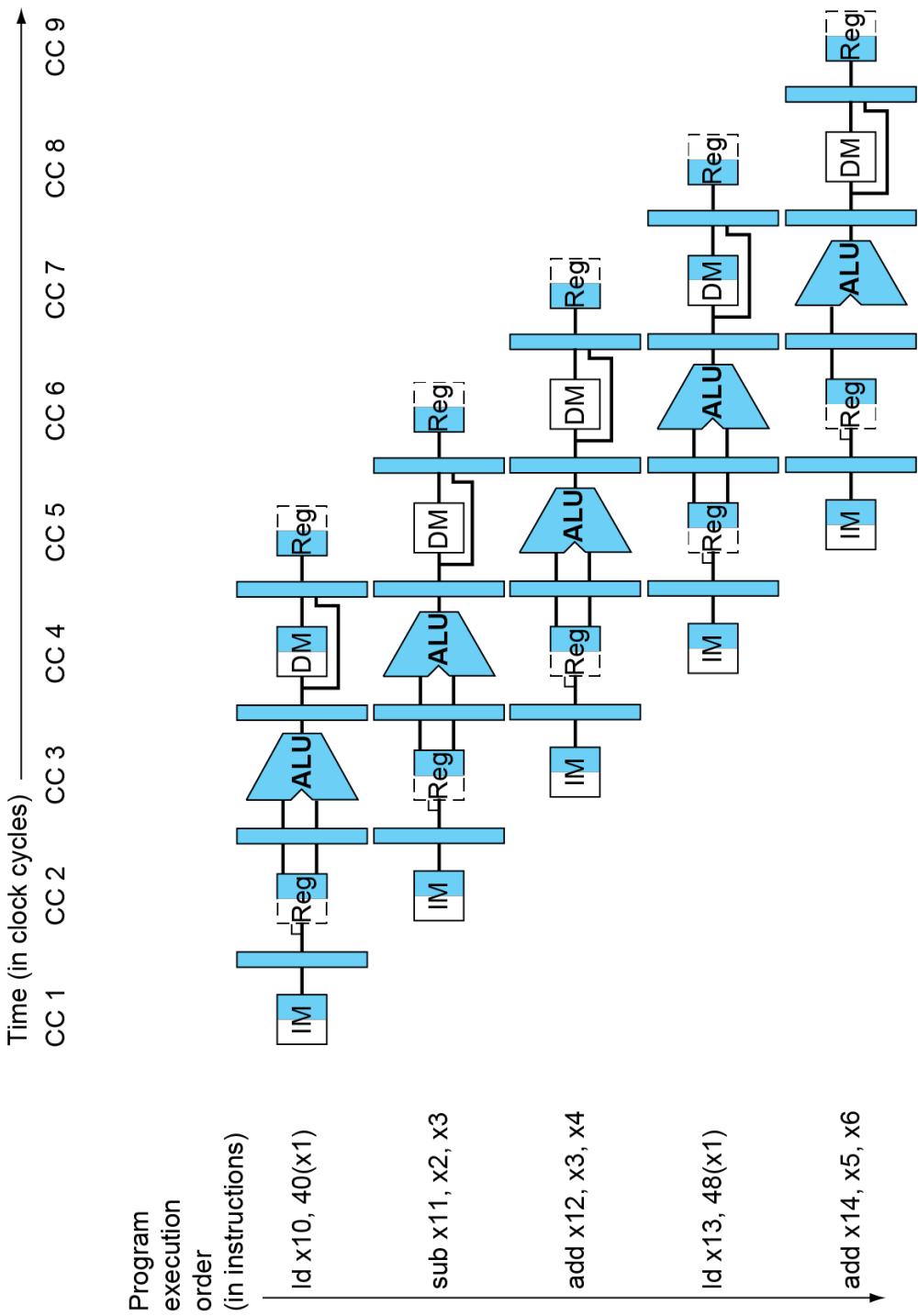


WB for Store



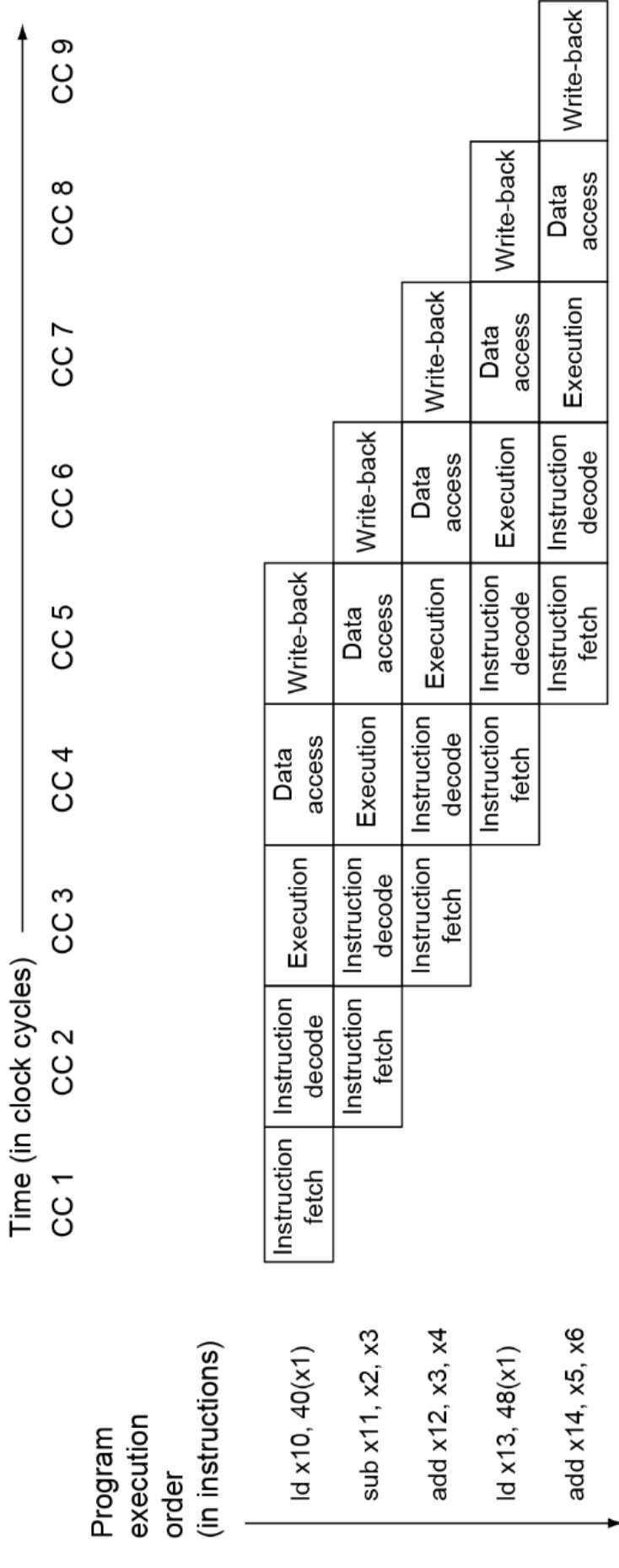
Multi-Cycle Pipeline Diagram

Form showing resource usage



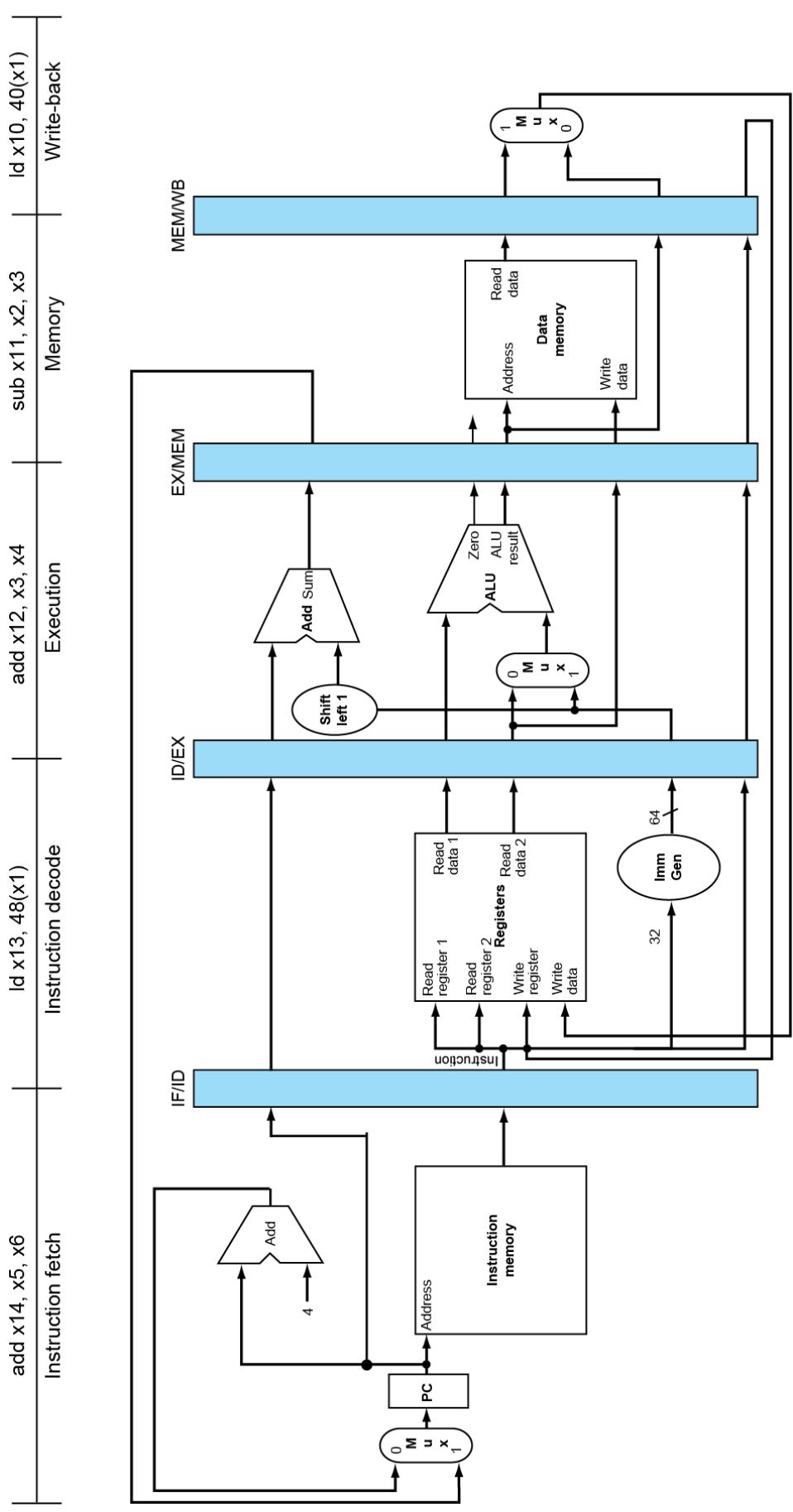
Multi-Cycle Pipeline Diagram

Traditional form

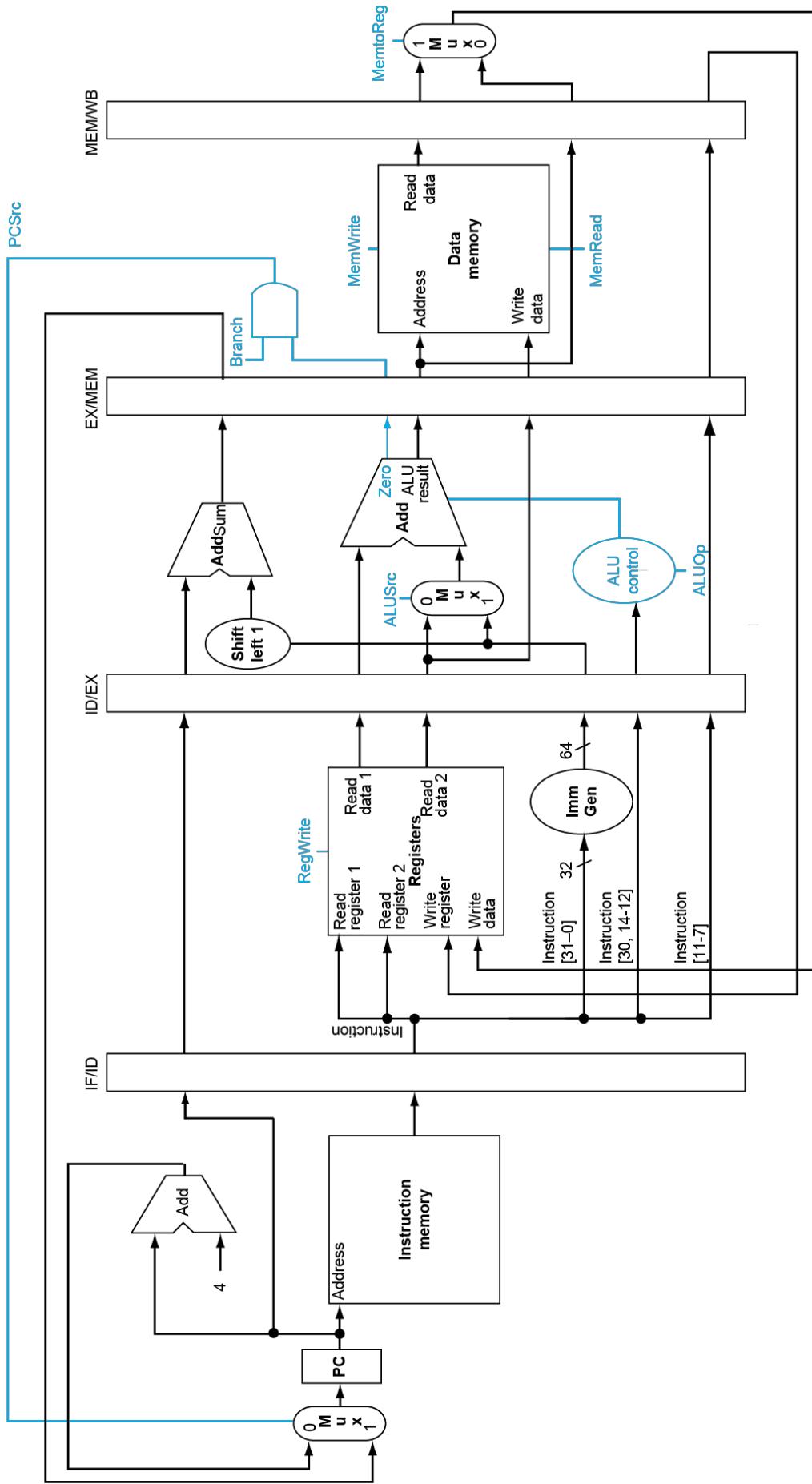


Single-Cycle Pipeline Diagram

State of pipeline in a given cycle

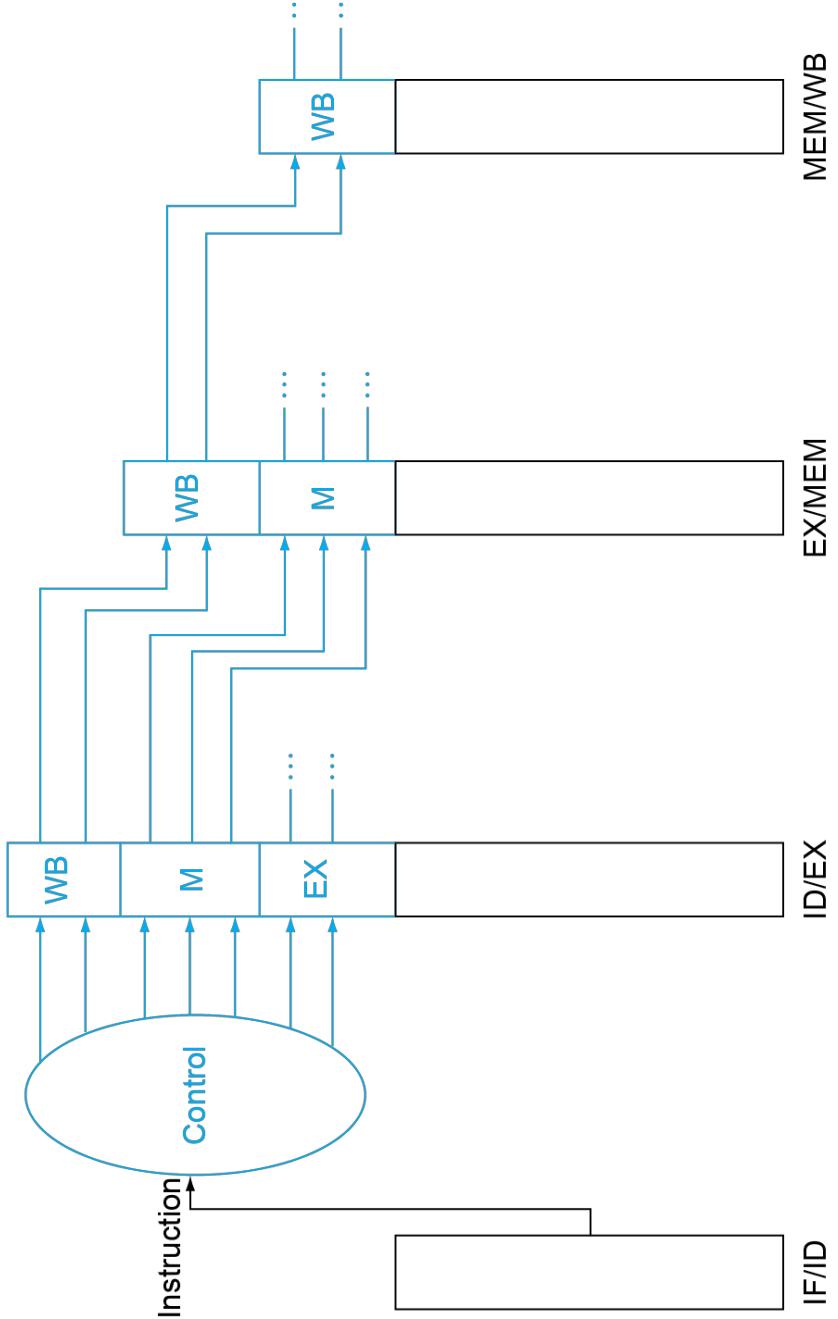


Pipelined Control (Simplified)

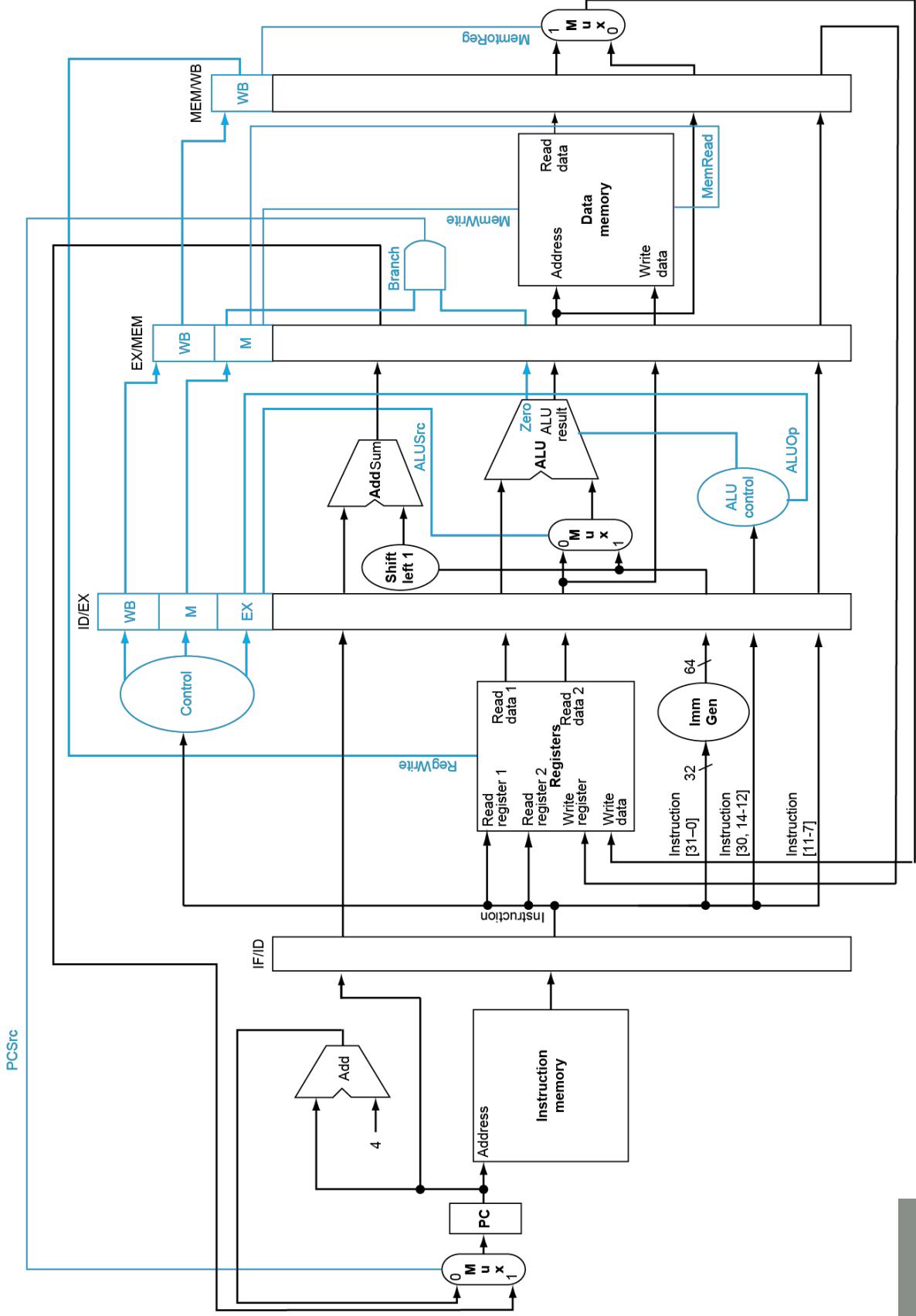


Pipelined Control

Control signals derived from instruction
As in single-cycle implementation



Pipelined Control



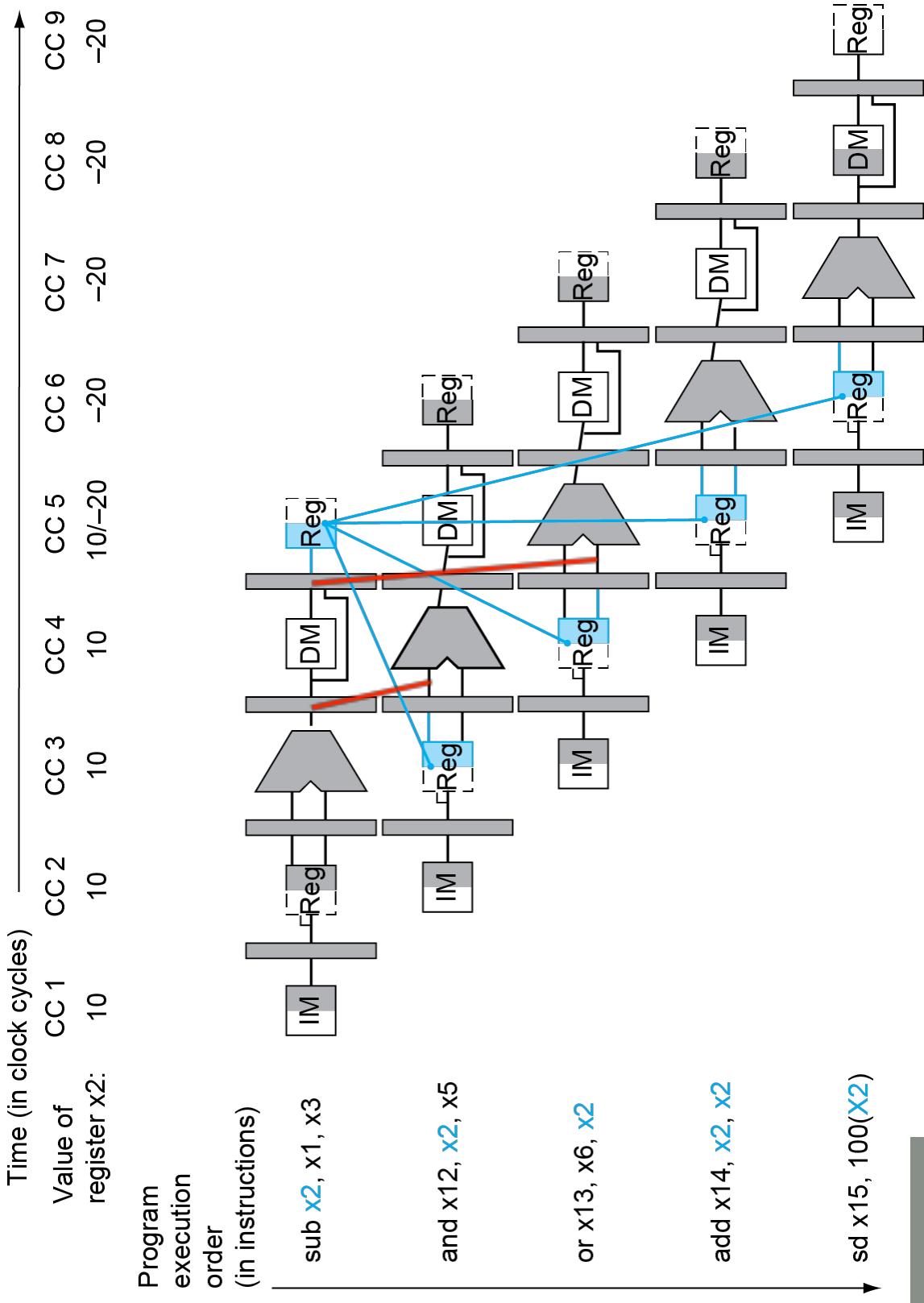
Data Hazards in ALU Instructions

Consider this sequence:

sub x_2 , x_1 , x_3
and x_{12} , x_2 , x_5
or x_{13} , x_6 , x_2
add x_{14} , x_2 , x_2
sd x_{15} , 100(x_2)

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

Dependencies & Forwarding



Detecting the Need to Forward

Pass register numbers along pipeline
e.g., ID/EX.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 Fwd from EX/MEM pipeline reg
- 1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2 Fwd from MEM/WB pipeline reg
- 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
- 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

Detecting the Need to Forward

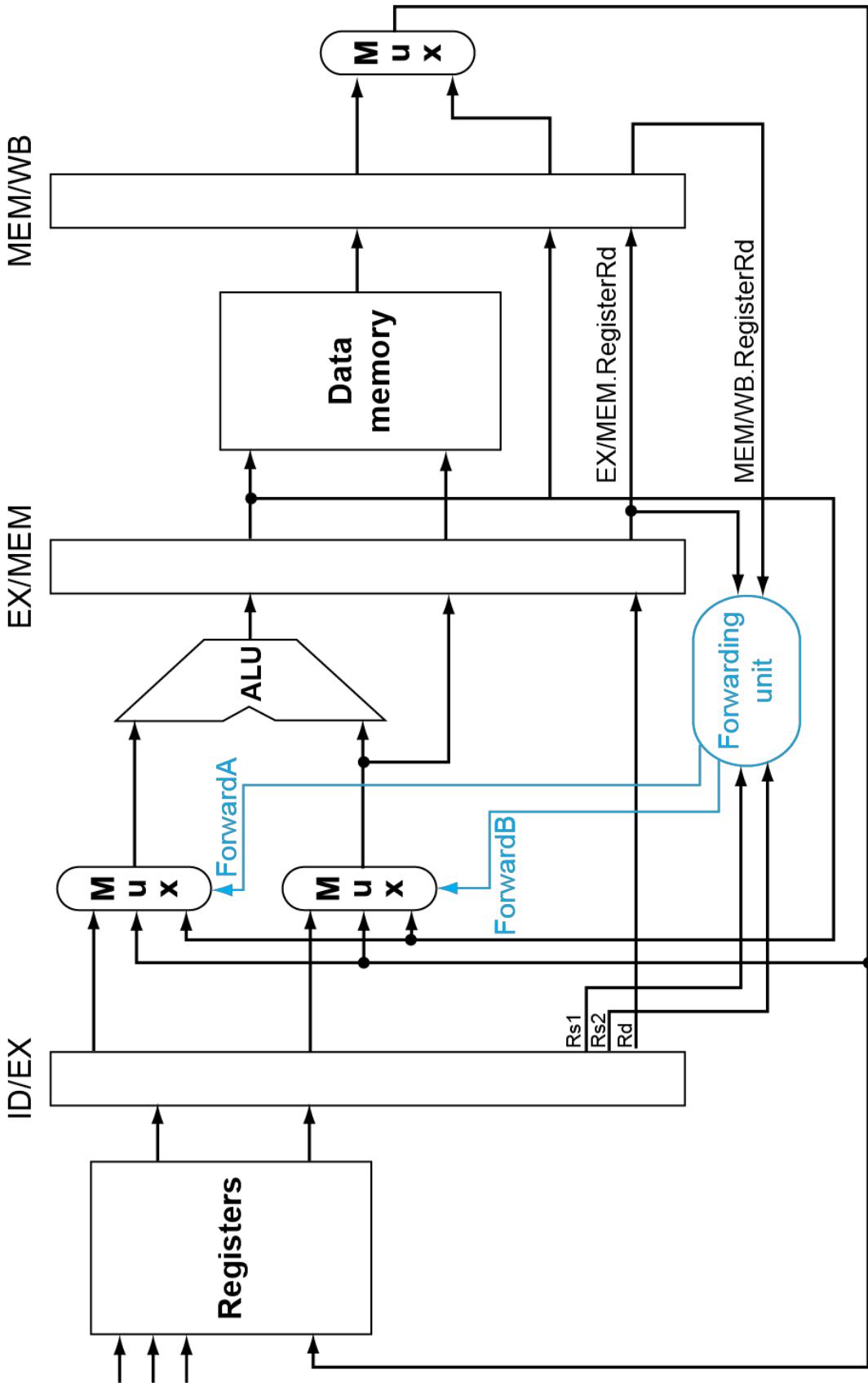
But only if forwarding instruction will write to a register!

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 x_1, x_1, x_2

add x_1, x_1, x_3

add x_1, x_1, x_4

Both hazards occur

Want to use the most recent

Revise MEM hazard condition

Only fwd if EX hazard condition isn't true



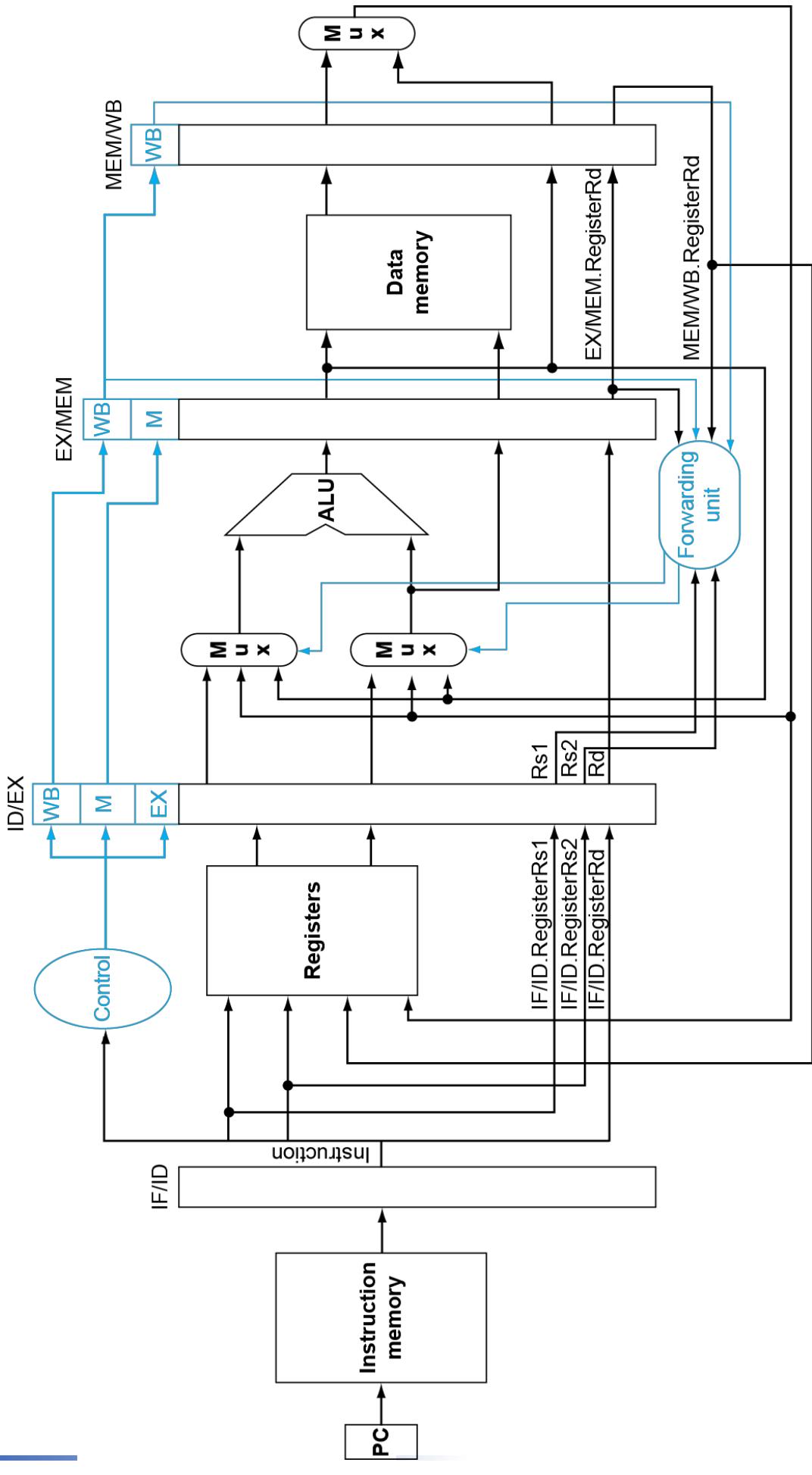
Revised Forwarding Condition

MEM hazard

```
if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd ≠ 0)  
and not(EX/MEM.RegisterRd and (EX/MEM.RegisterRd ≠ 0)  
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs1))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01  
if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd ≠ 0)  
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)  
and (EX/MEM.RegisterRd ≠ ID/EX.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

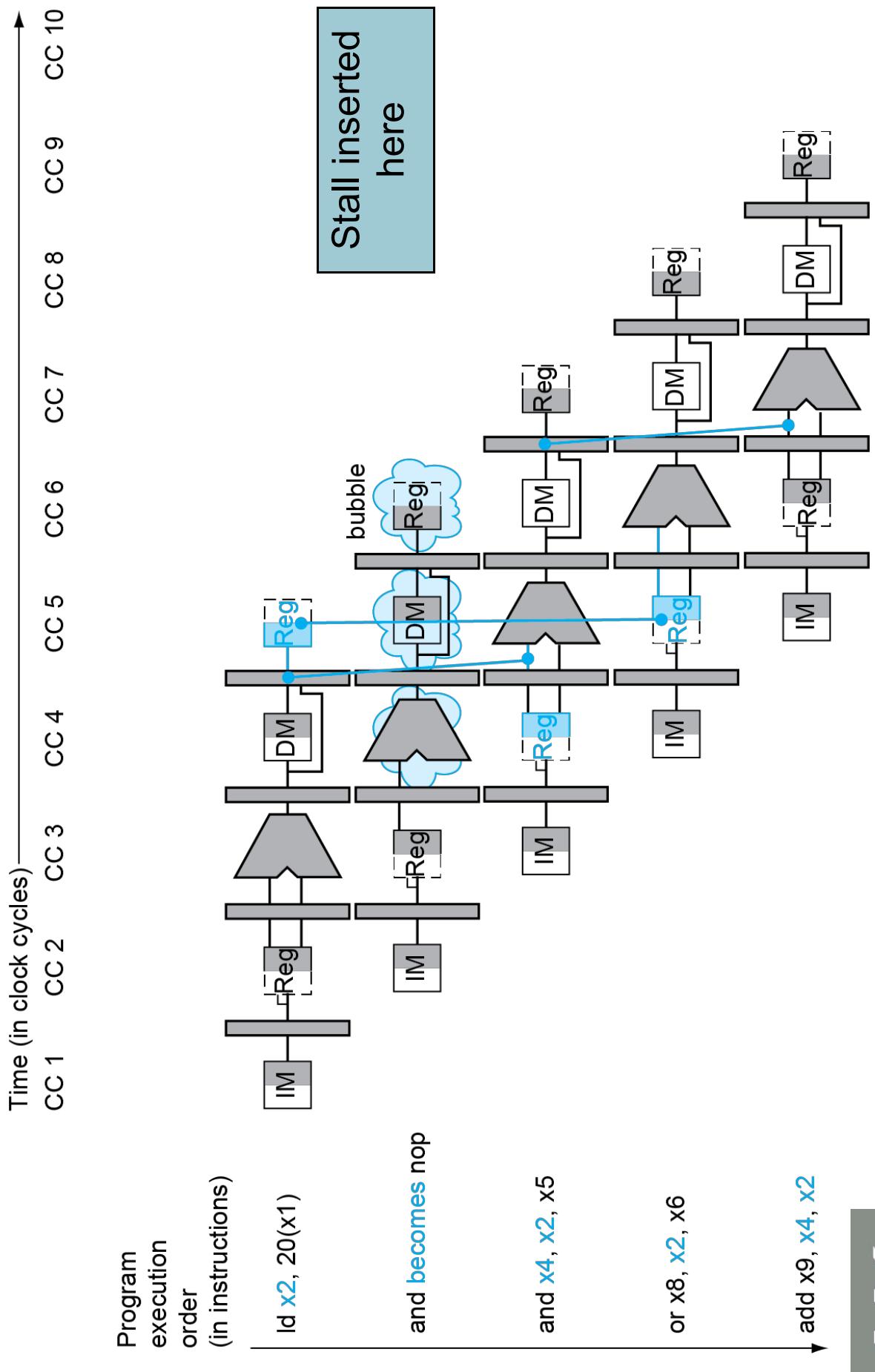
EX, MEM and WB do nop (no-operation)

Prevent update of PC and IF/ID register

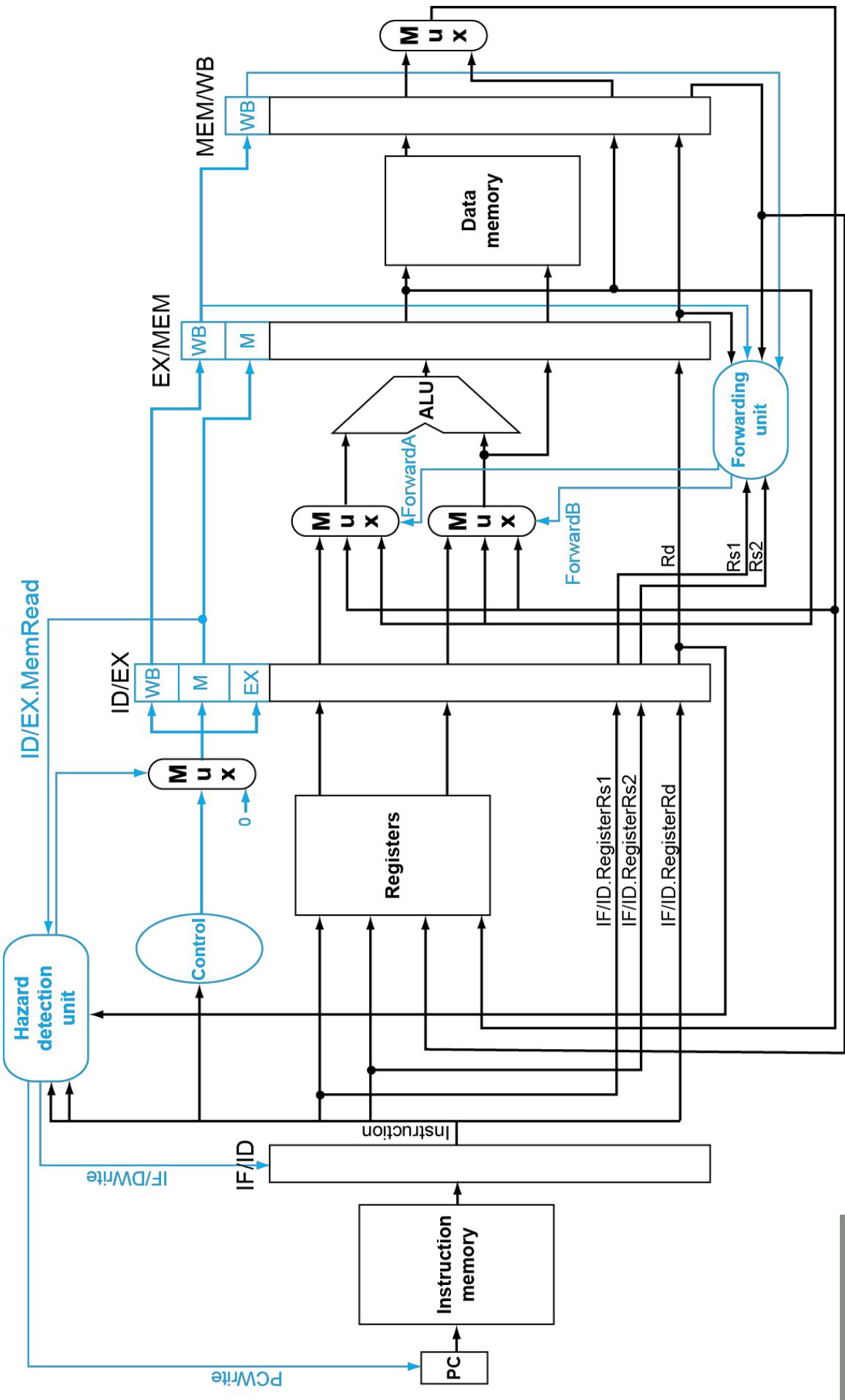
Using instruction is decoded again

Following instruction is fetched again
1-cycle stall allows MEM to read data for 1 d
Can subsequently forward to EX stage

Load-Use Data Hazard



Datapath with Hazard Detection



Stalls and Performance

The BIG Picture

Picture stalls reduce performance

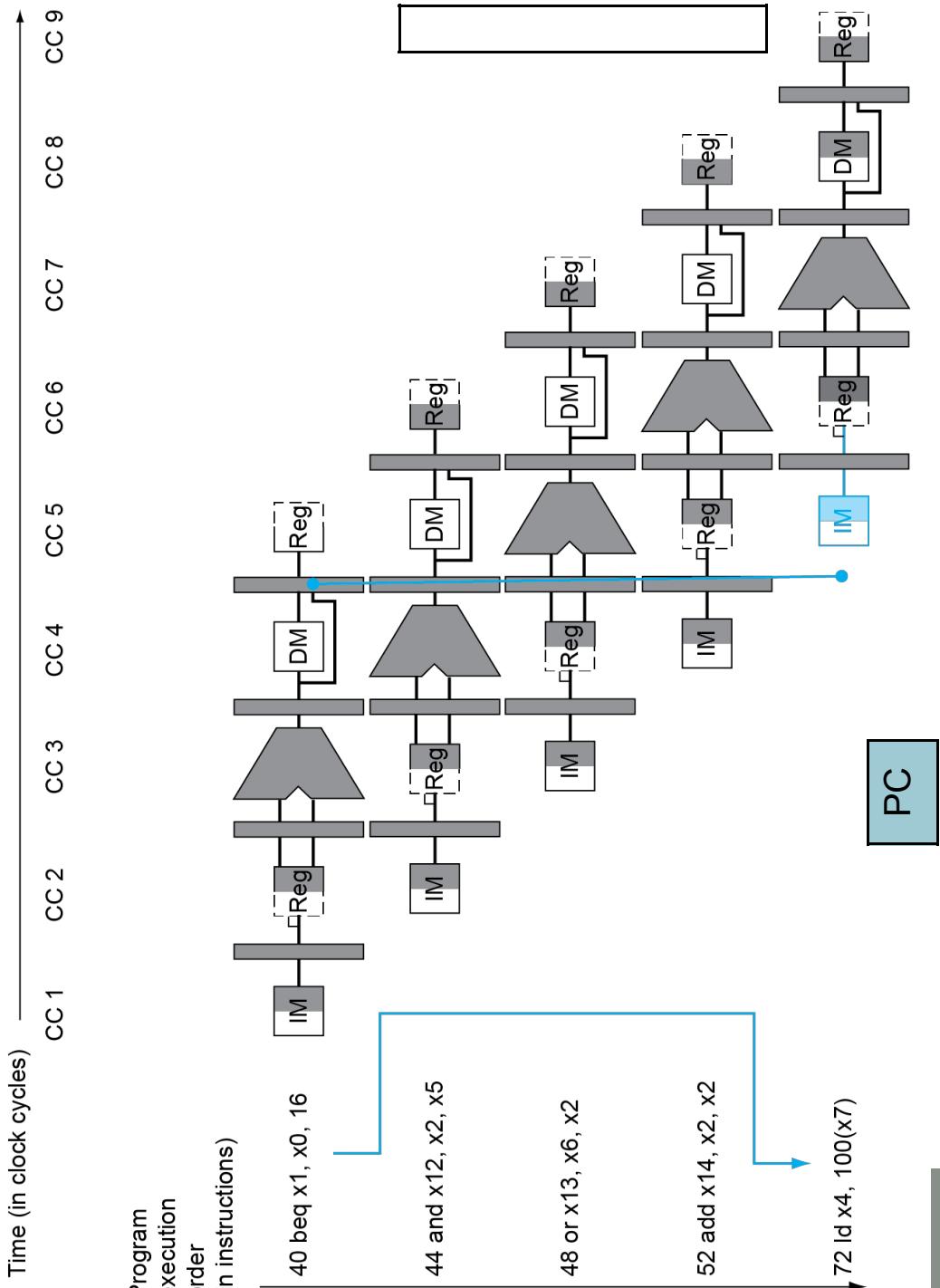
But are required to get correct results

Compiler can arrange code to avoid
hazards and stalls

Requires knowledge of the pipeline structure

Branch Hazards

If branch outcome determined in MEM



Reducing Branch Delay

Move hardware to determine outcome to ID stage

Target address adder

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)
```

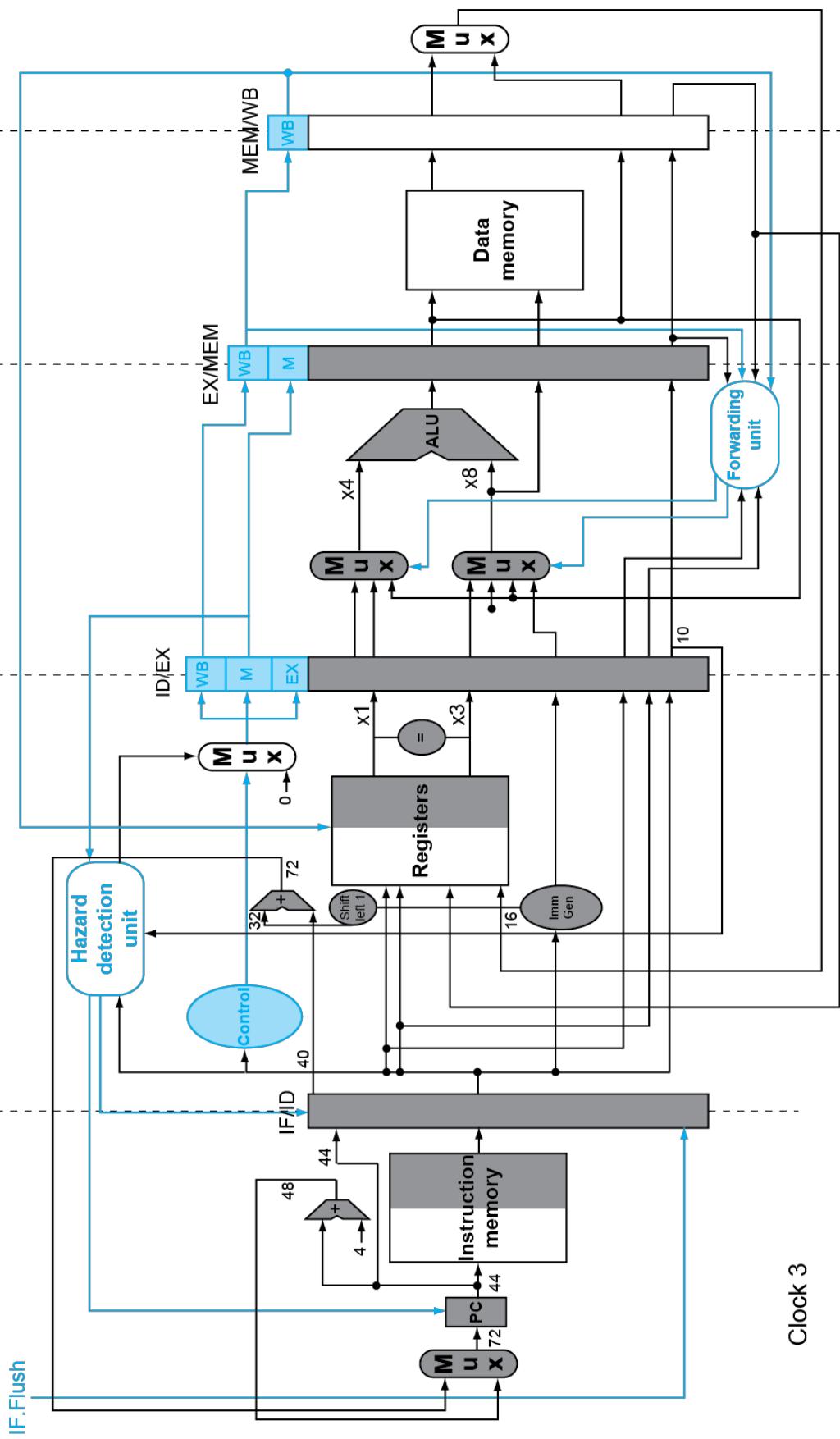
Example: Branch Taken

and x12, x2, x5

beq x1, x3, 16

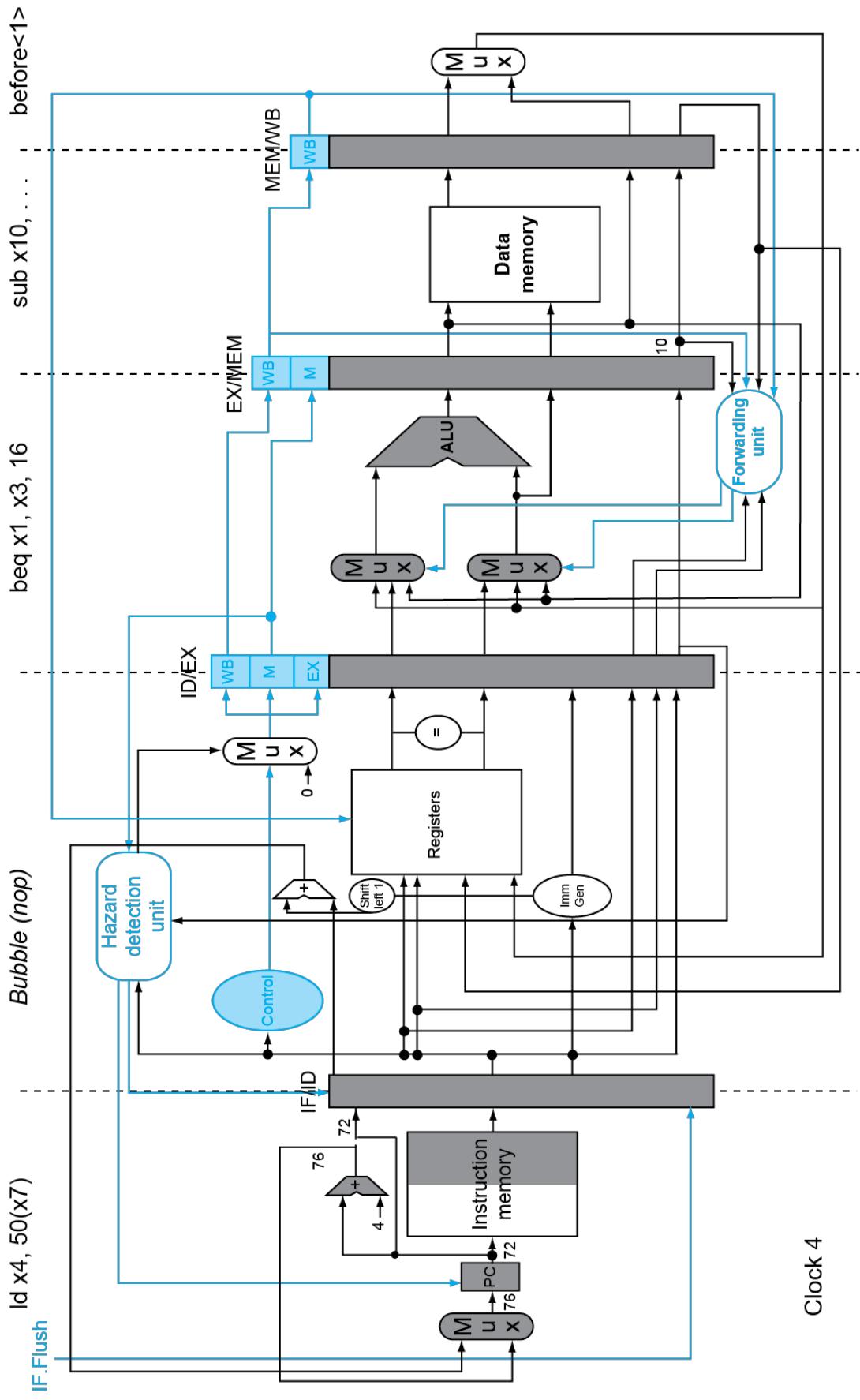
sub x10, x4, x8

before <2>



Clock 3

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

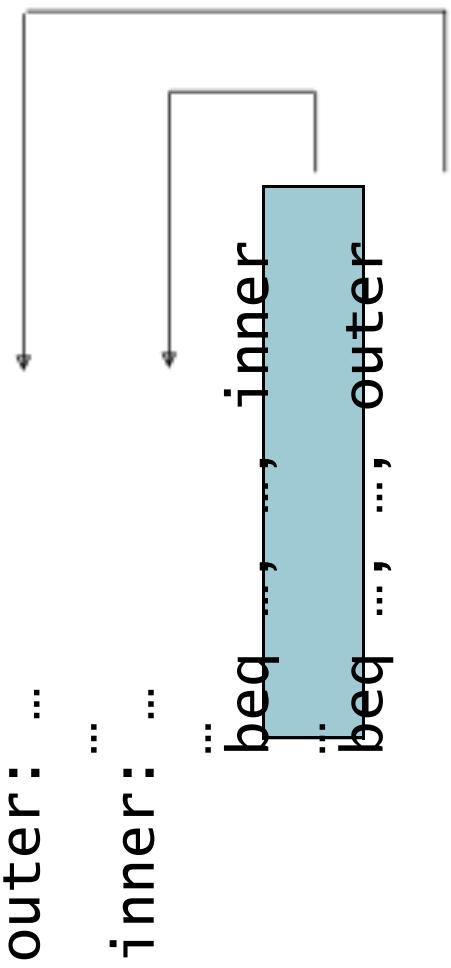
Start fetching from fall-through or target

If wrong, flush pipeline and flip prediction



1-Bit Predictor: Shortcoming

Inner loop branches mispredicted twice!

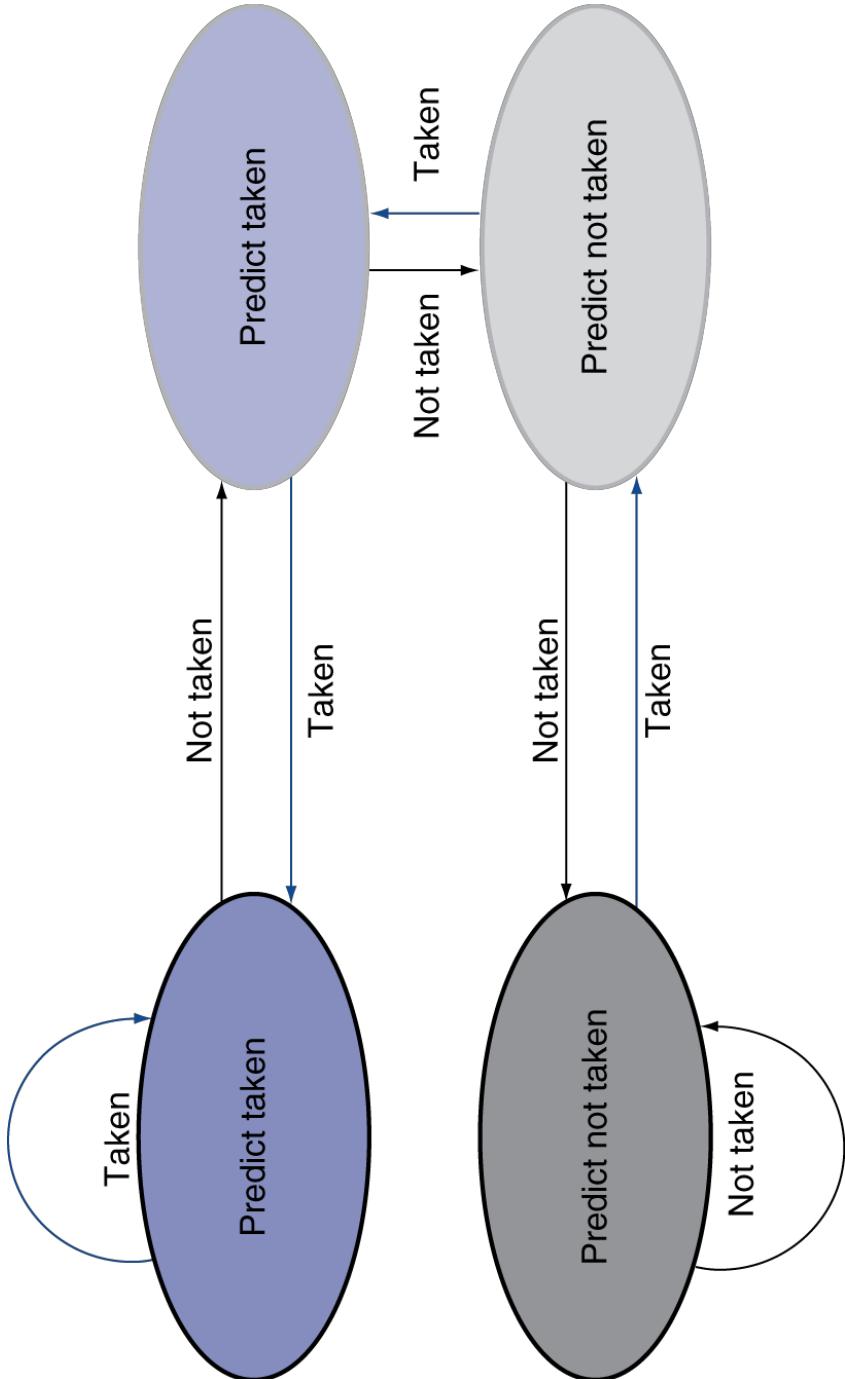


Mispredict as taken on last iteration of
inner loop

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 00000two

Hardware malfunction: 01 1000 00000two

⋮
⋮

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

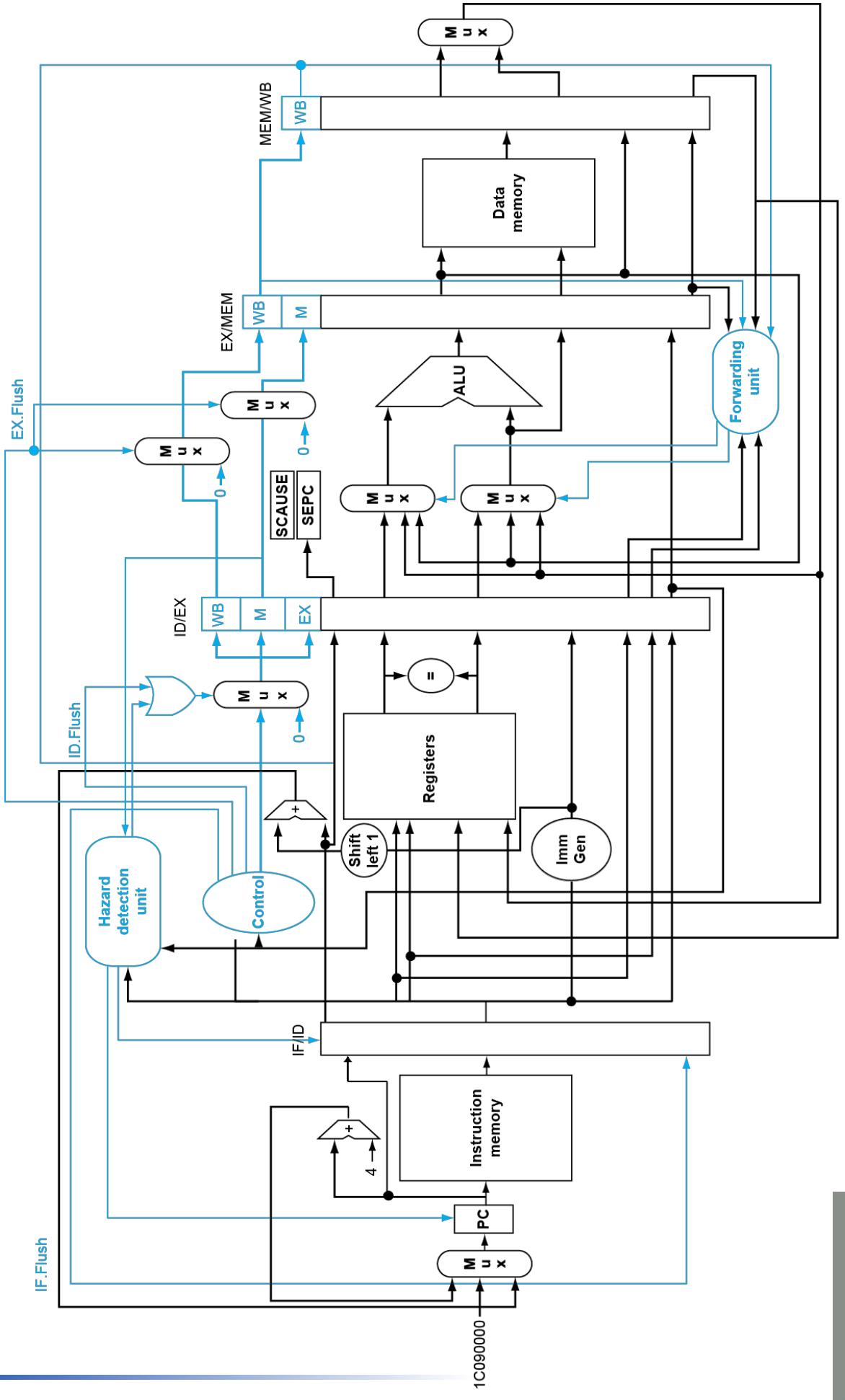
Terminate program

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)
...

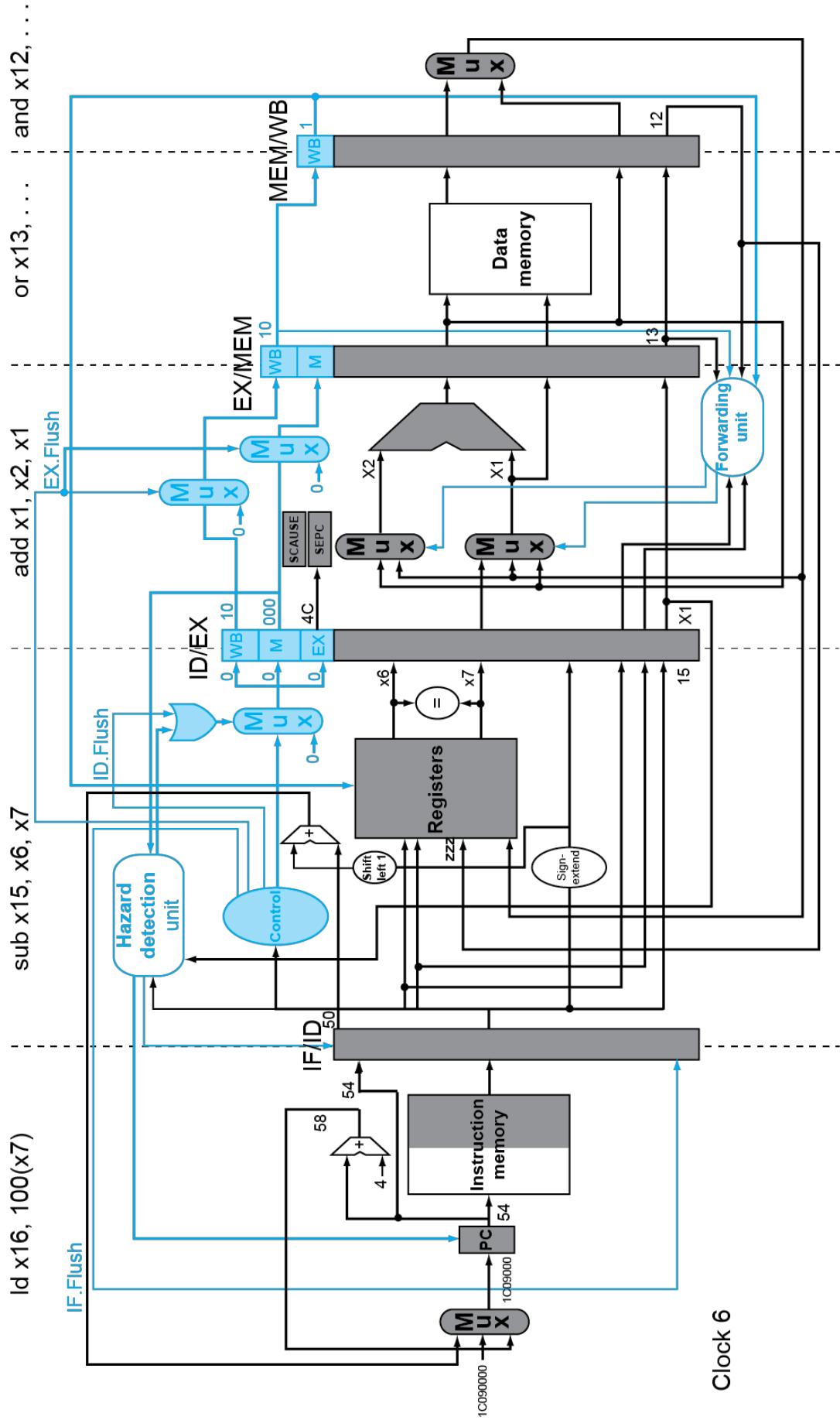
```

Handler

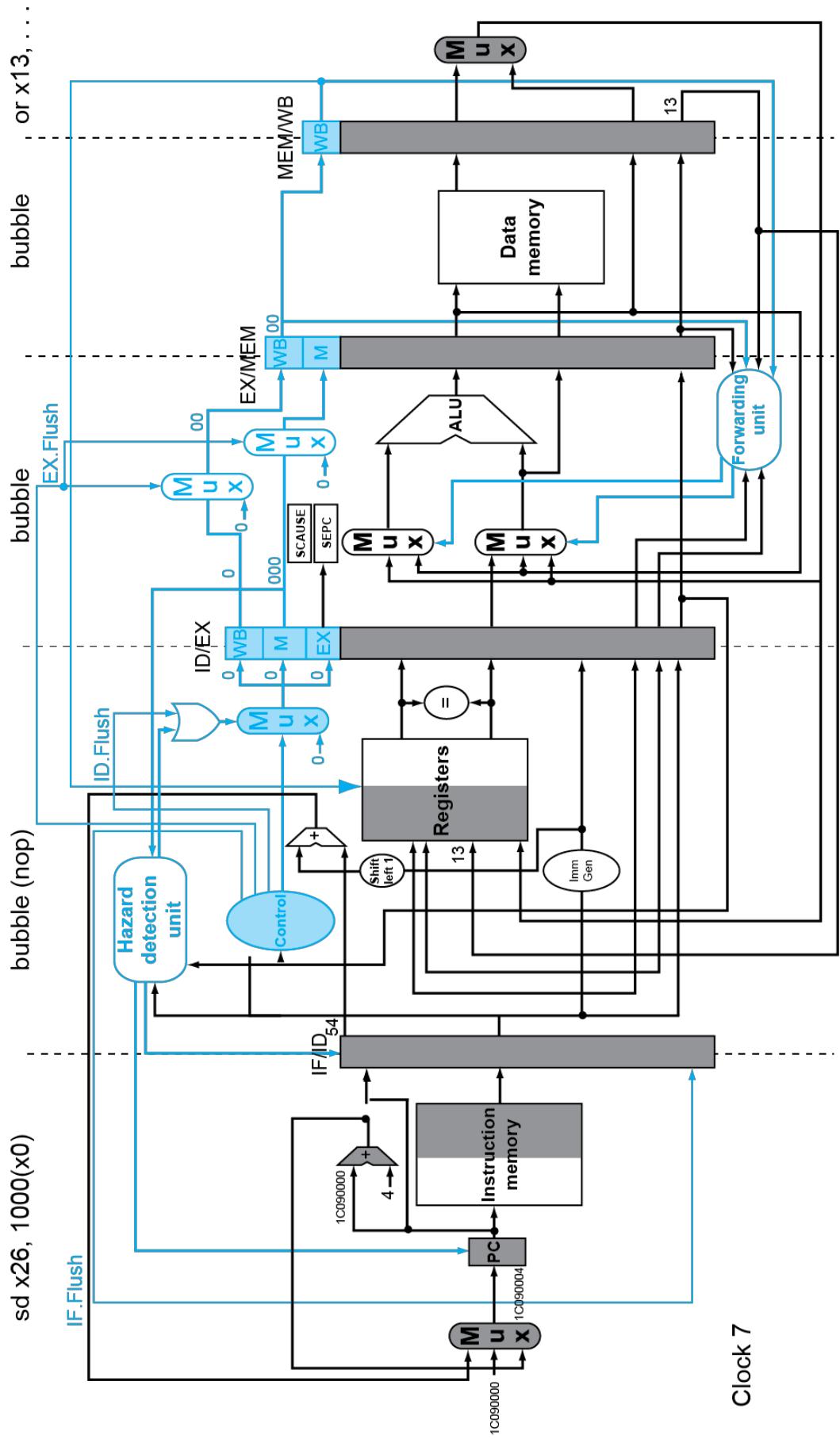
```
1c0900000 sd x26, 1000(x10)
1c0900004 sd x27, 1008(x10)
...

```

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!



Inprecise 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 □ shorter clock cycle

Multiple issue

Replicate pipeline stages □ 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
□ 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

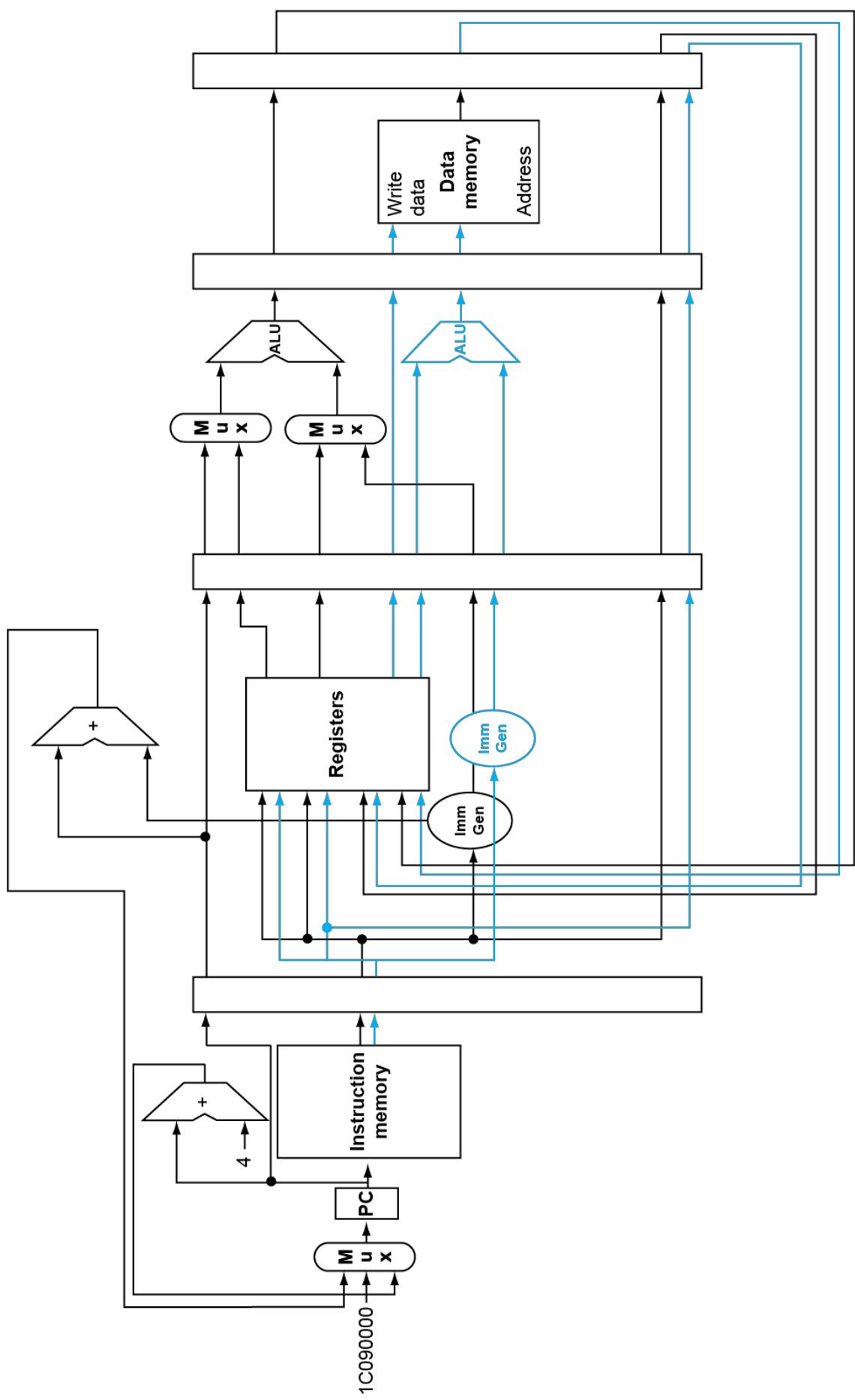
ALU/branch, then load/store

Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages				
		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 x_{10} , x_0, x_1
1d $x_2, 0(x_{10})$

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
      b1t 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
.	b1t x22,x20,Loop	sd x31,8(x20)	4

$$IPC = 5/4 = 1.25 \text{ (c.f. peak IPC = 2)}$$

Loop Unrolling

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



Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop	addi x20, x20, -32	1d x28 , 0(x20)	1
:	nop	1d x29 , 24 (x20)	2
	add x28, x28, x21	1d x30 , 16 (x20)	3
	add x29, x29, x21	1d 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
	b1t 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

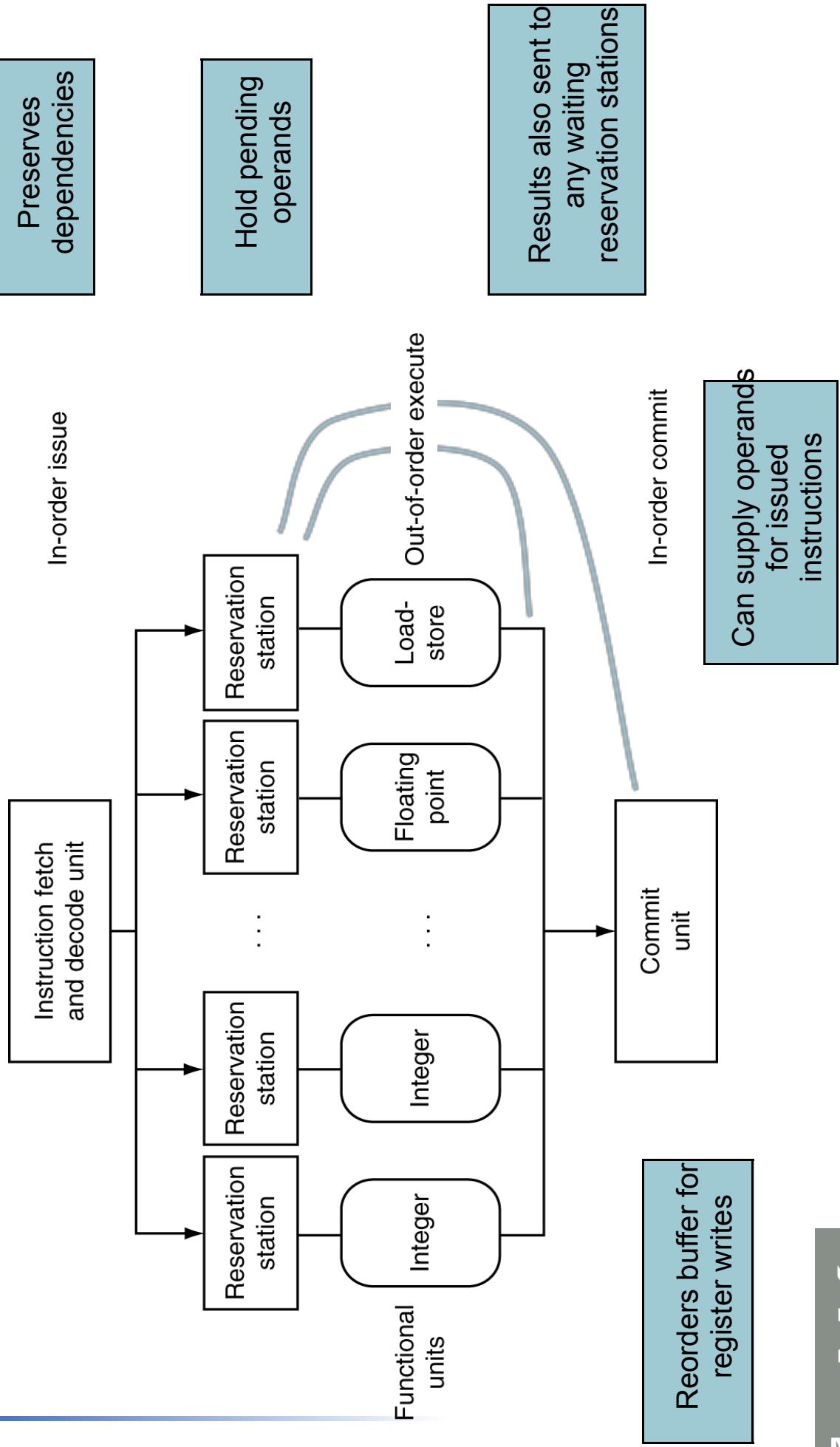
Example

```
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?

The BIG Picture

Yes, but not as much as we'd like

Programs have real dependencies that limit ILP

Some dependencies are hard to eliminate
e.g., pointer aliasing

Some parallelism is hard to expose

Limited window size during instruction issue

Memory delays and limited bandwidth
Hard to keep pipelines full

Speculation can help if done well

Power Efficiency

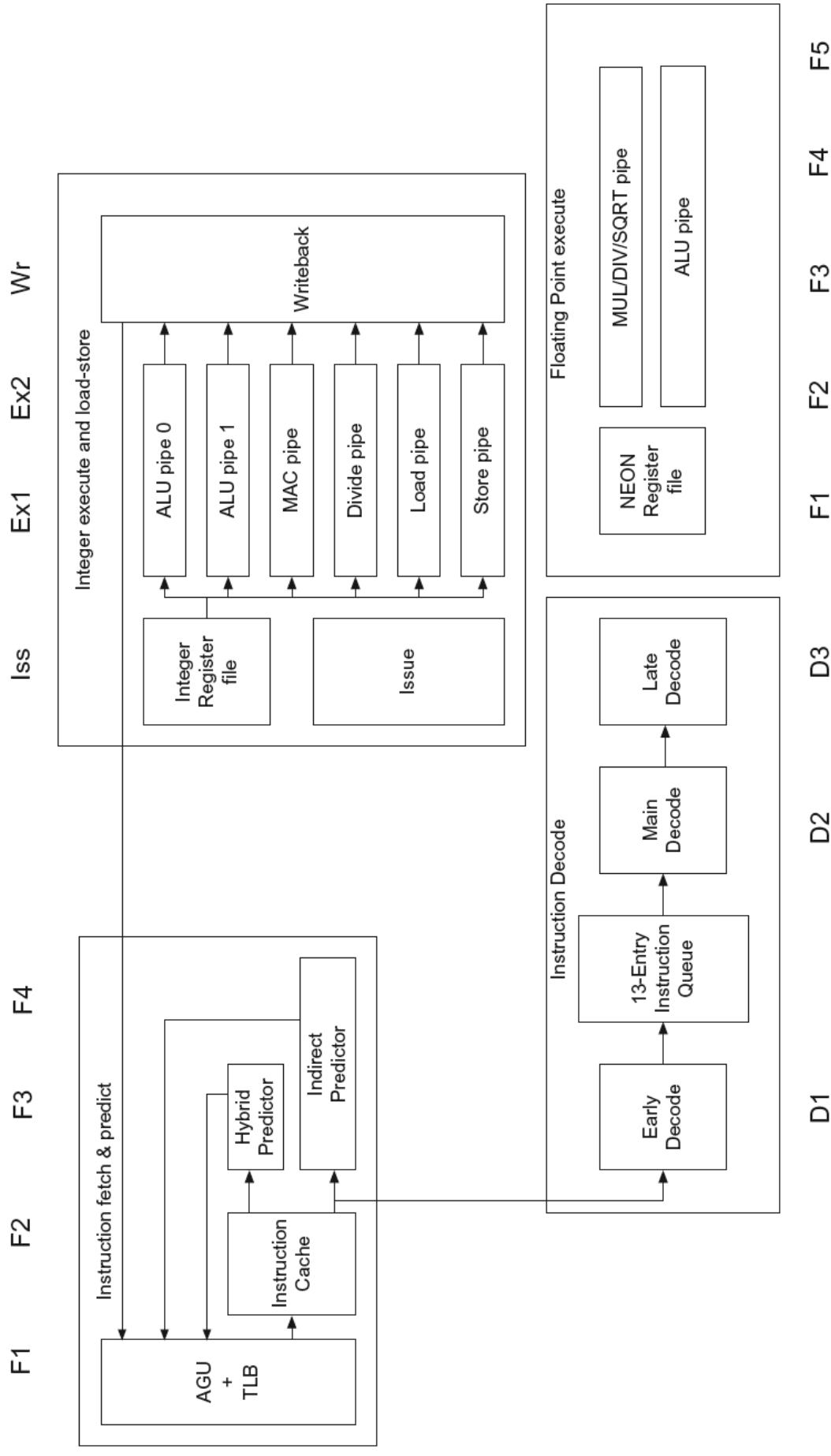
Complexity of dynamic scheduling and speculations requires power
Multiple simpler cores may be better

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

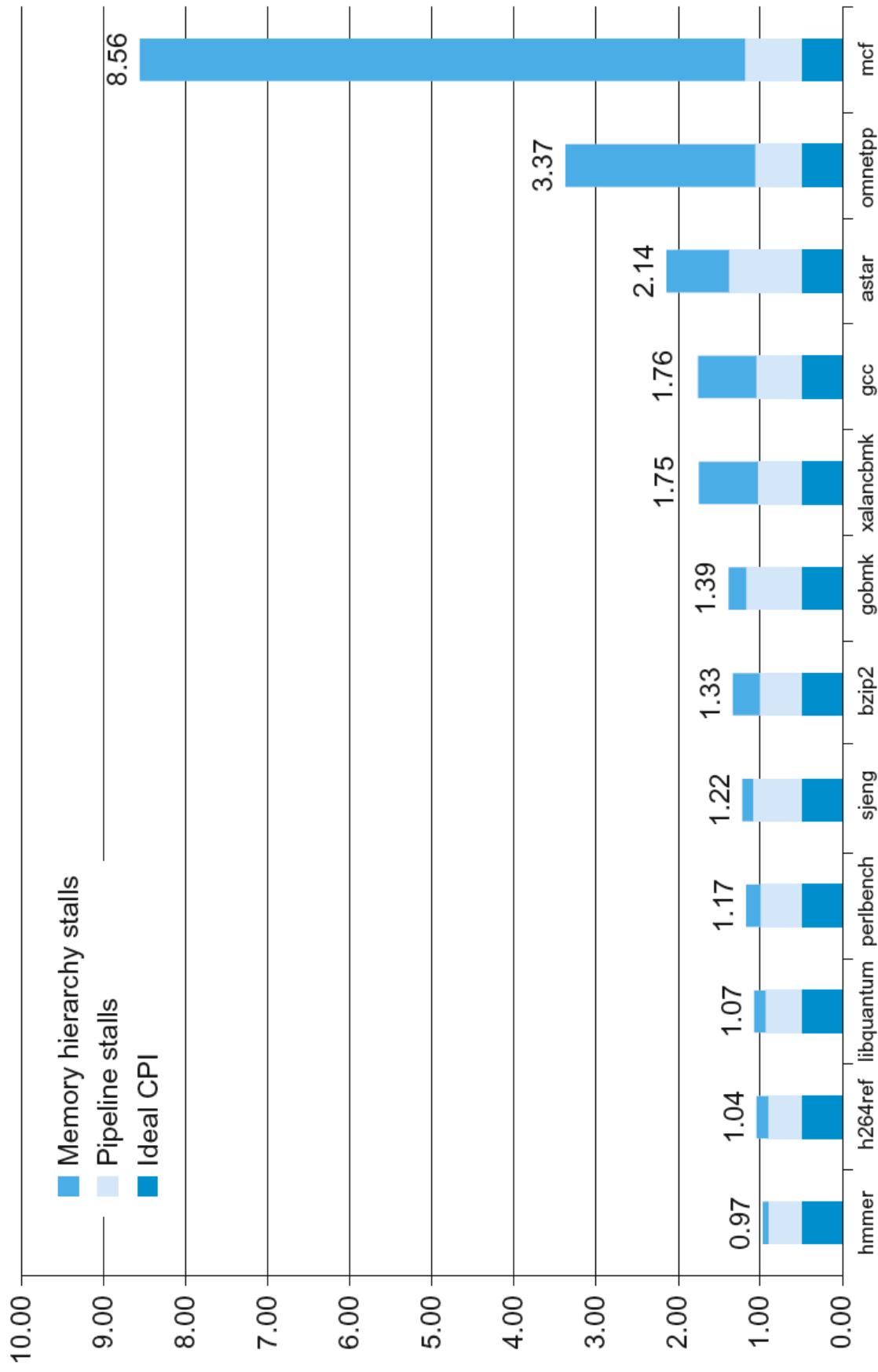
Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile	Server, cloud
Thermal design power	100 mWatts (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	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB L1, 16-64 KiB	32 KiB L1, 32 KiB D
2nd level caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

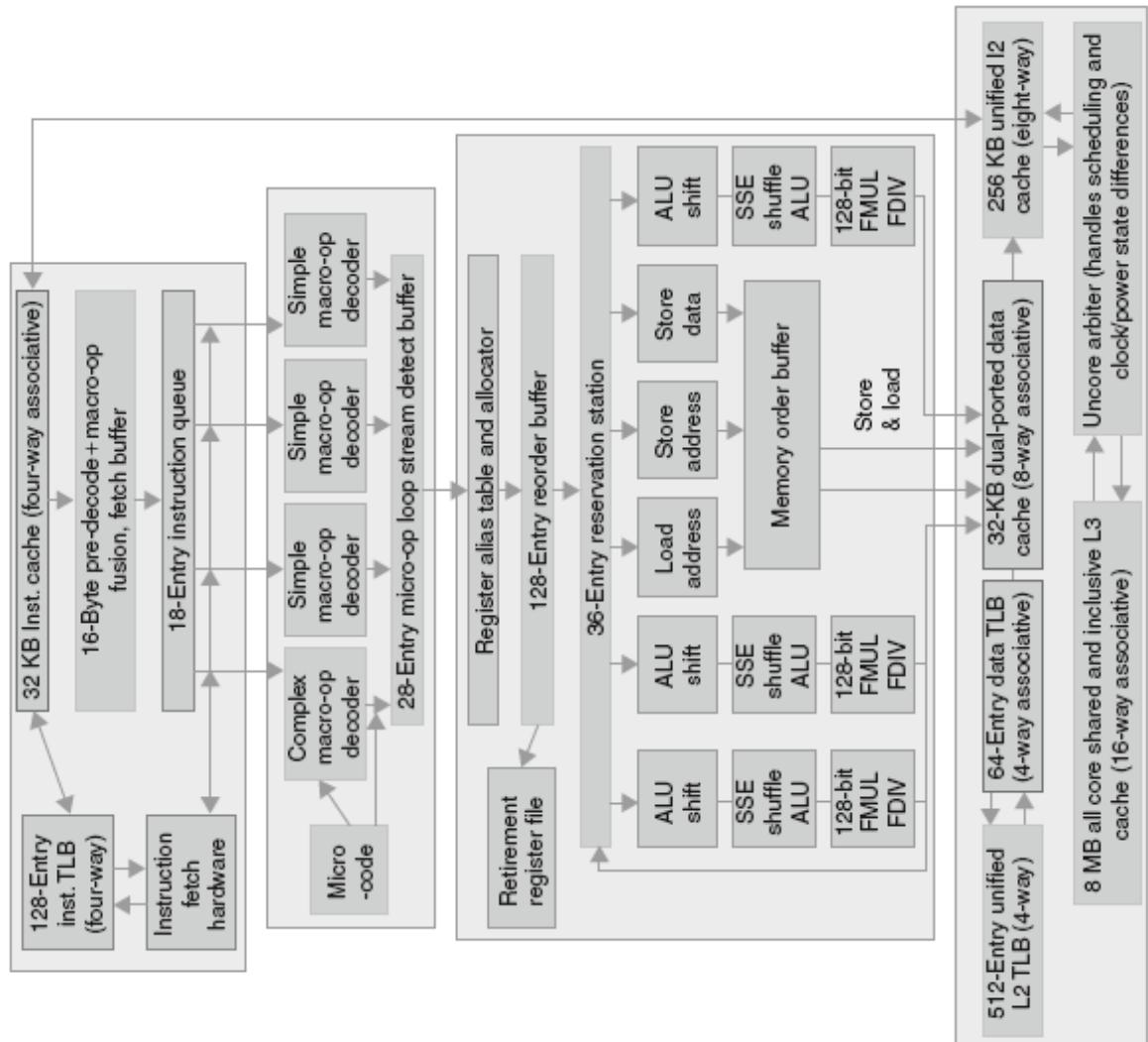
ARM Cortex-A53 Pipeline



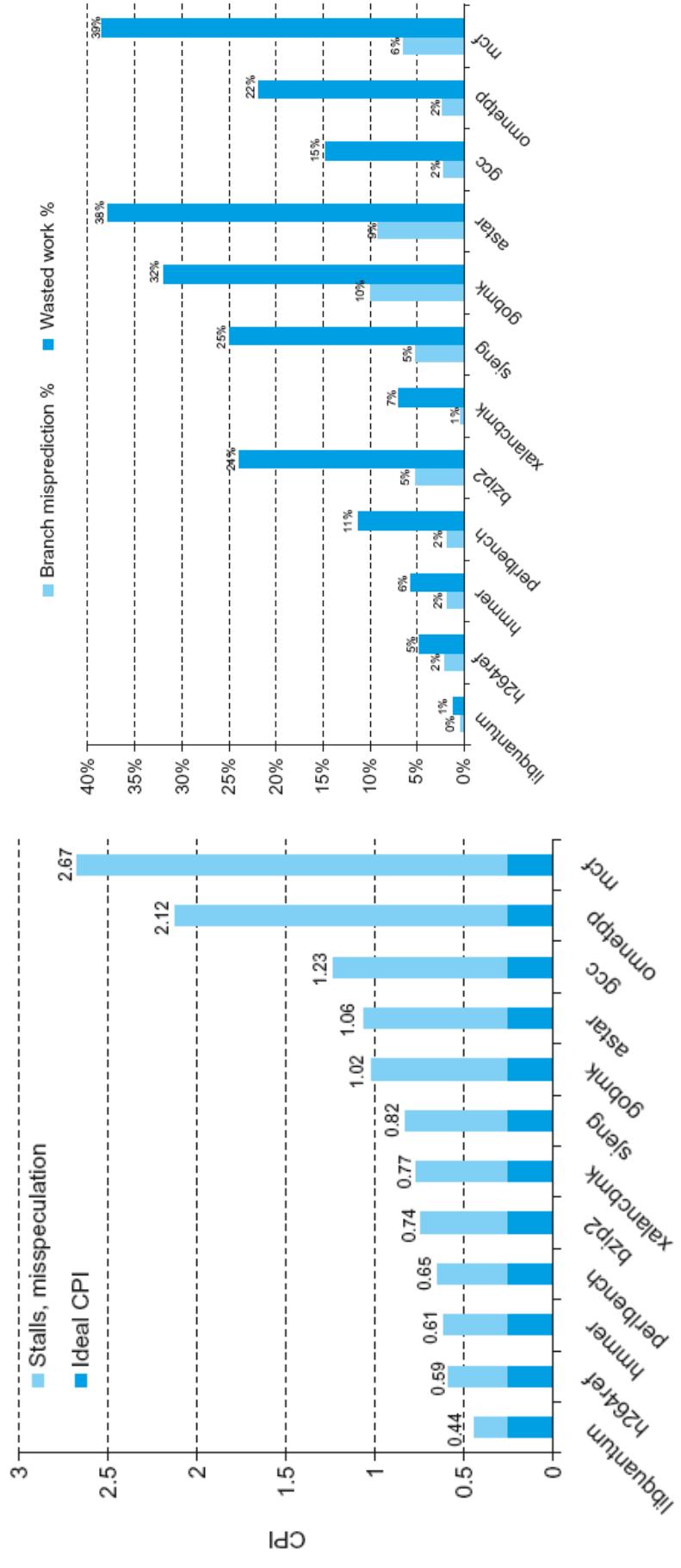
ARM Cortex-A53 Performance



Core i7 Pipeline



Core i7 Performance



Matrix Multiply

Unrolled C code

```
1 #include <xx86intrin.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 }
```

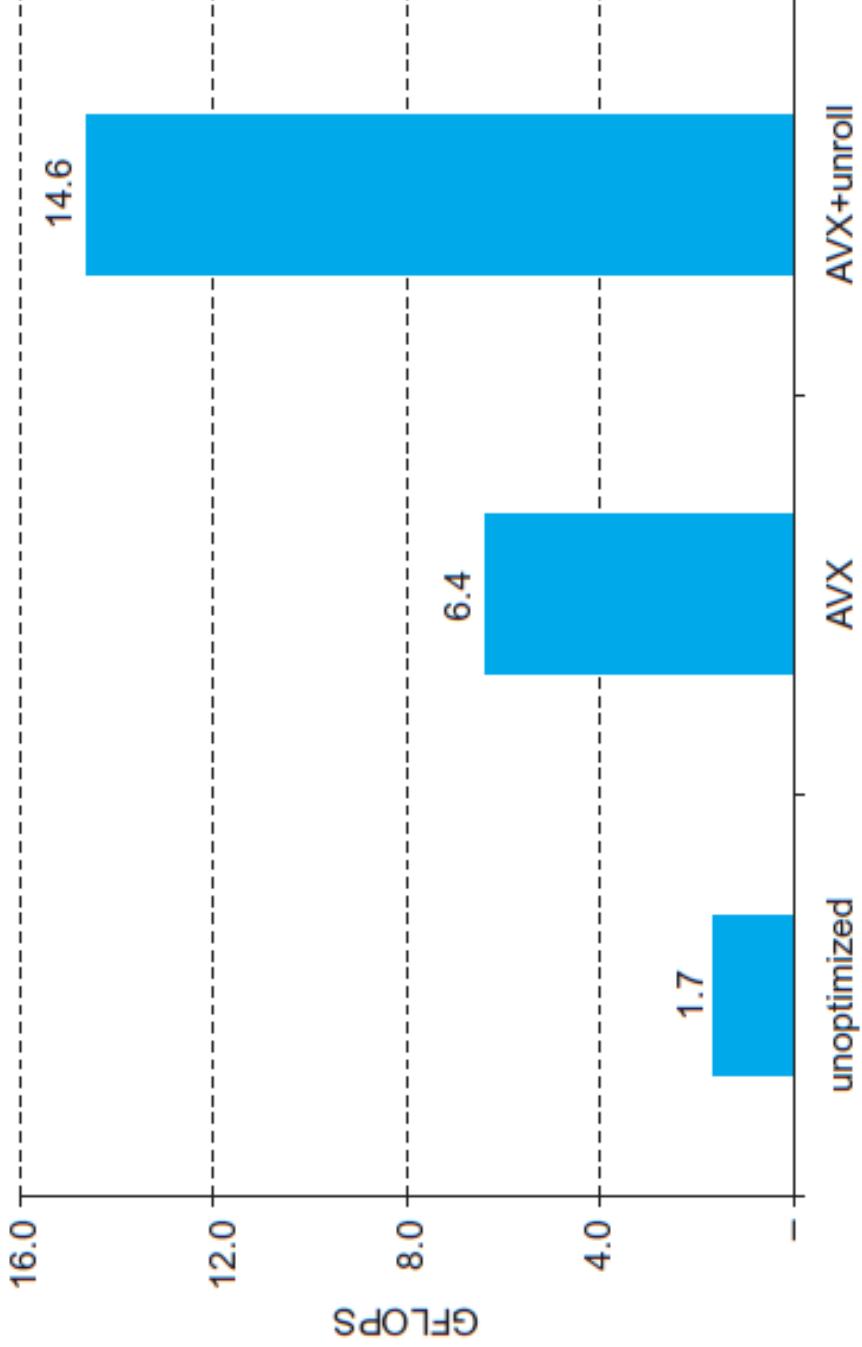
Matrix Multiply

Assembly code:

```
1 vmovapd (%r11), %ymm4
2 mov %rbx, %rax
3 xor %ecx, %ecx
4 vmovapd 0x20(%r11), %ymm3
5 vmovapd 0x40(%r11), %ymm2
6 vmovapd 0x60(%r11), %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 (LLP)
 - Dependencies limit achievable parallelism
 - Complexity leads to the power wall