

# Chapter 4 – part II

## Pipelining

**Tien-Fu Chen**

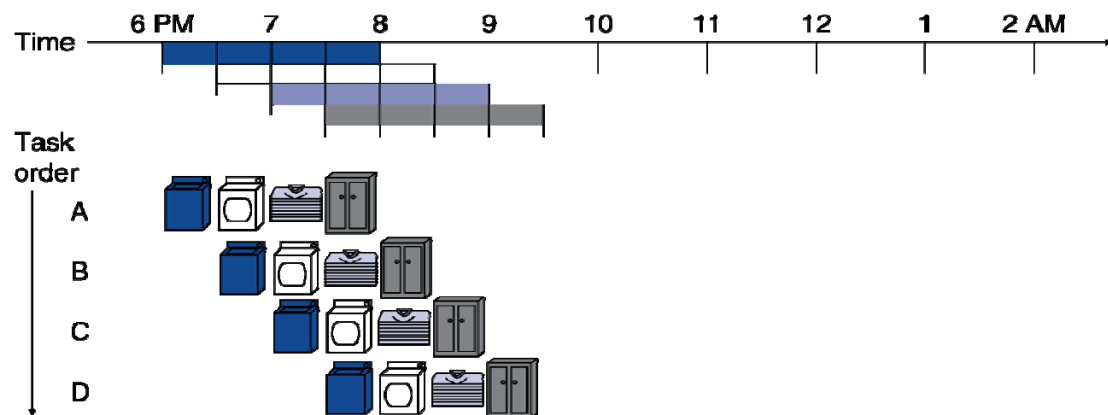
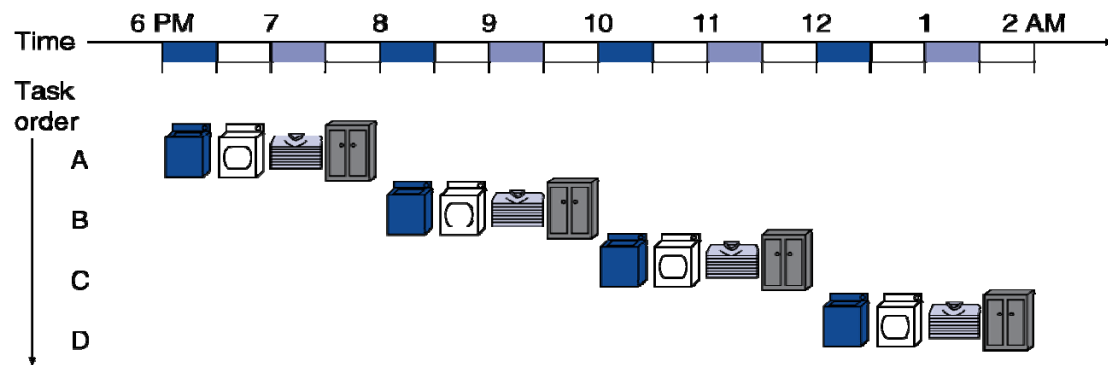
Dept. of Computer Science and  
Information Engineering

**National Chiao Tung Univ.**

Material source:  
COD RISC-V slides

# Pipelining Analogy

- ❑ Pipelined laundry: overlapping execution
  - Parallelism improves performance



- ❑ Four loads:

- Speedup  
 $= 8 / 3.5 = 2.3$

- ❑ Non-stop:

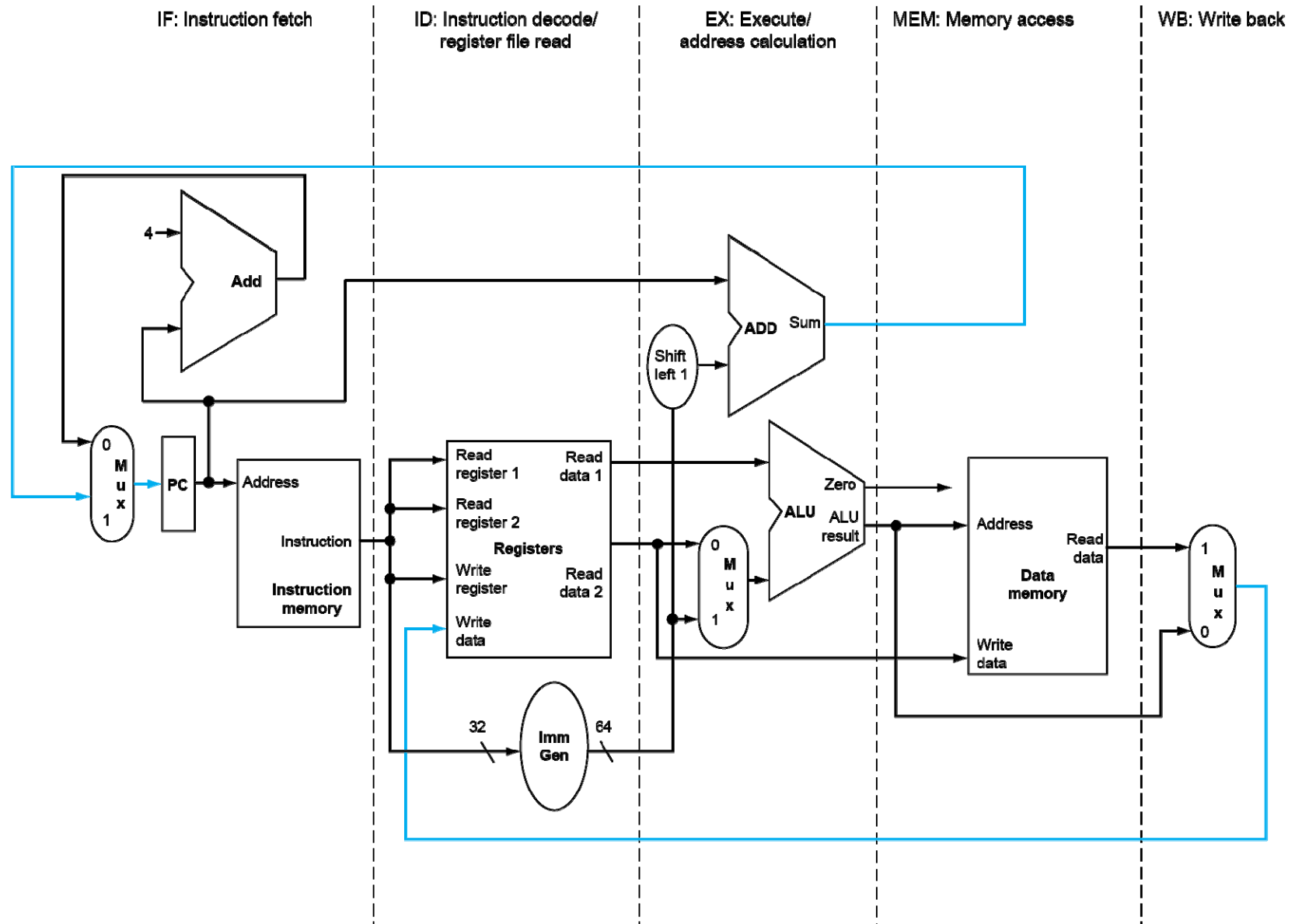
- Speedup  
 $= 2n / 0.5n + 1.5 \approx 4$   
 $= \text{number of stages}$

# RISC-V Pipeline

---

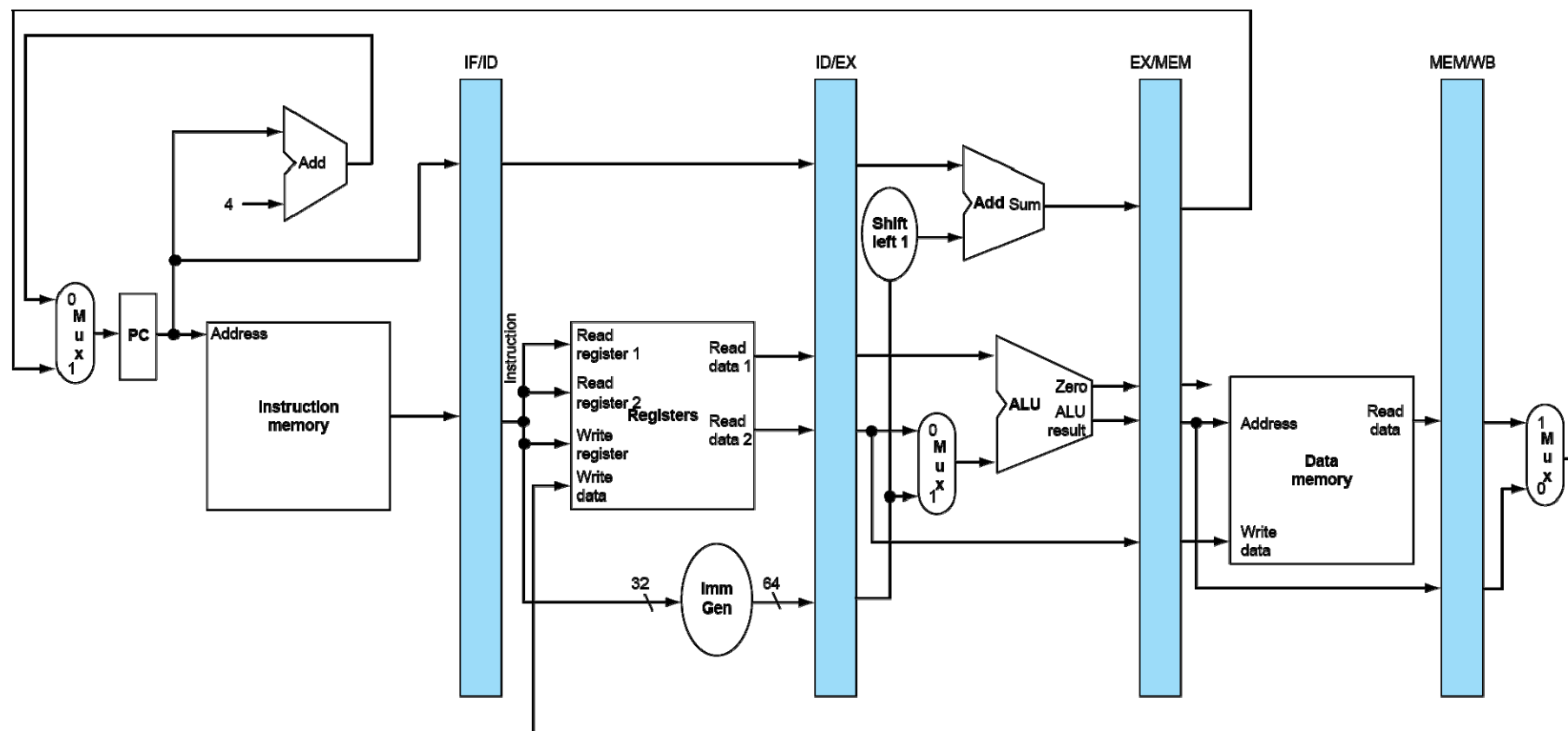
- ❑ Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

# RISC-V Pipelined Datapath



# Pipeline registers

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



# Pipeline Performance

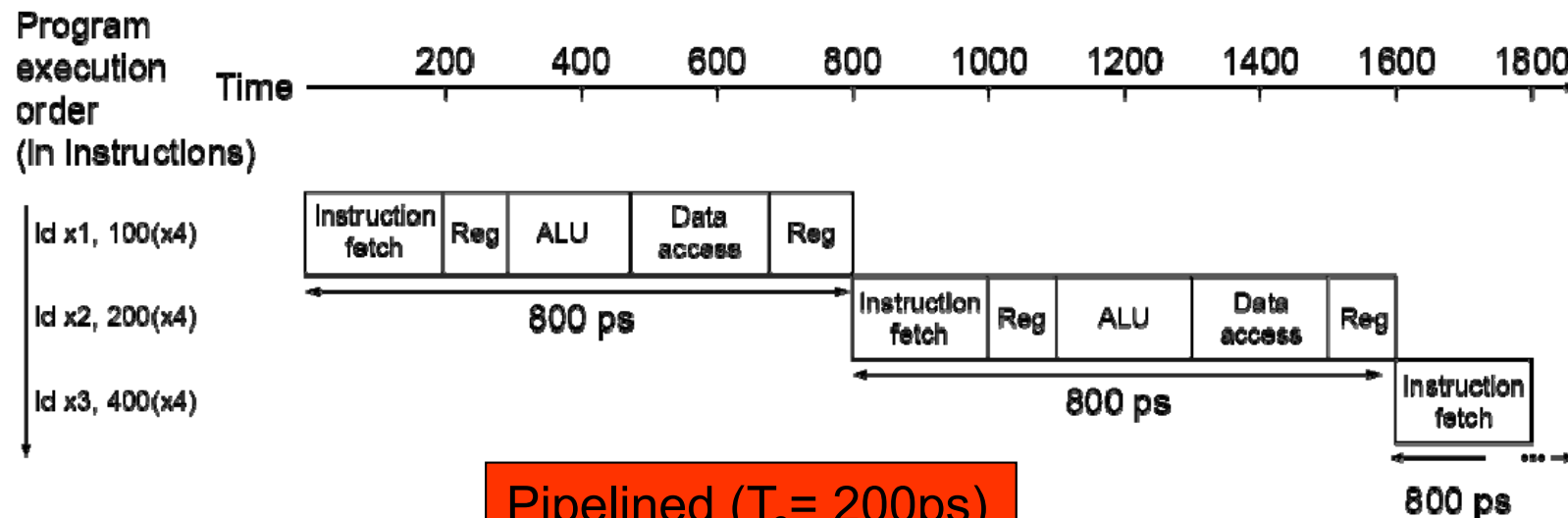
---

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

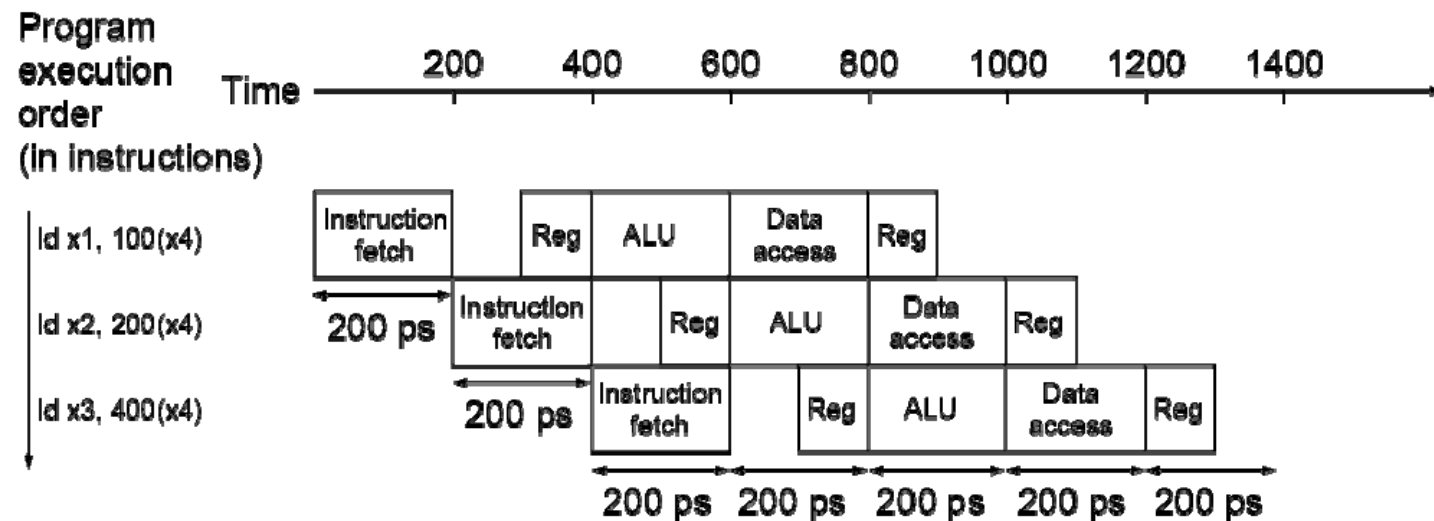
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

# Pipeline Performance

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

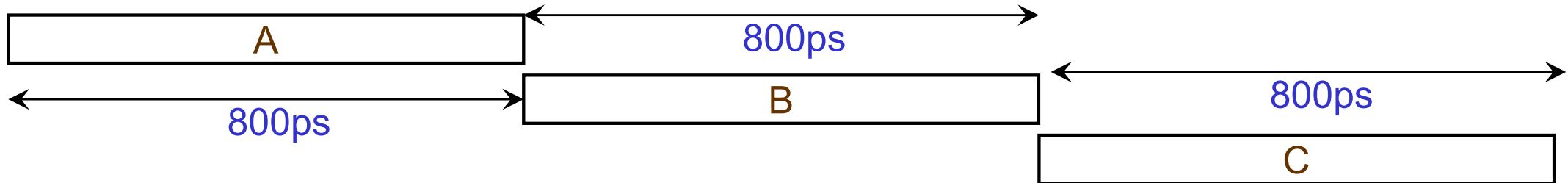


Pipelined ( $T_c = 200\text{ps}$ )

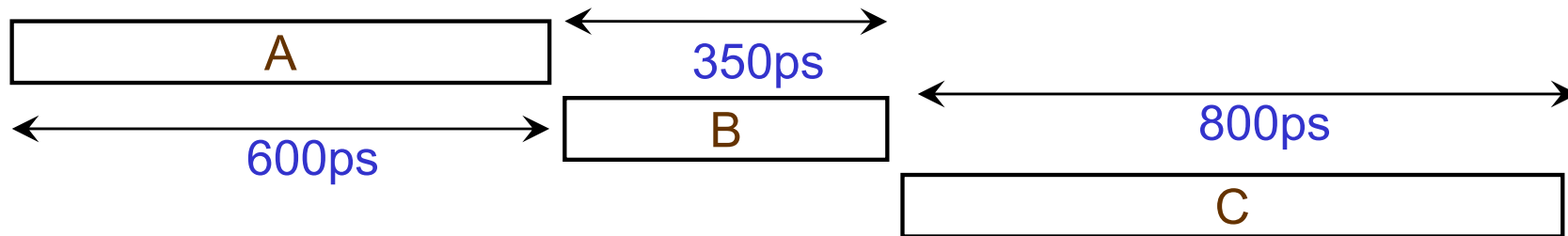


# Comparisons

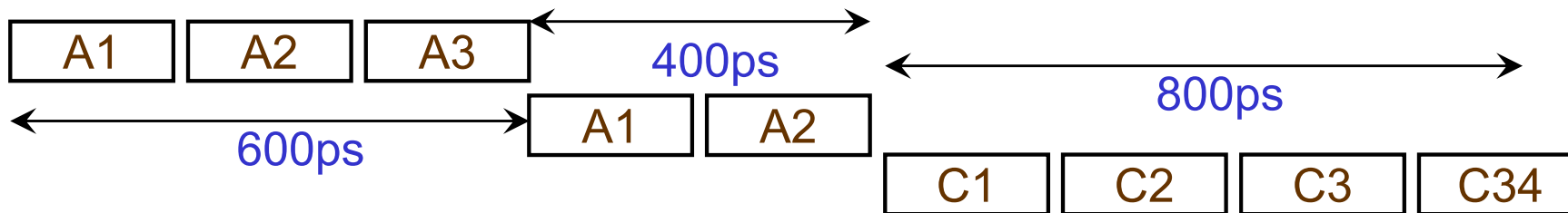
## ❑ Single-cycle MIPS



## ❑ Variable-clock MIPS



## ❑ Multi-cycle MIPS





# Designing Pipelining

---

## ❑ Instruction Set for pipelining

- all instructions are the same length
- just a few instruction formats, symmetry register fields
- memory operands appear only in loads and stores
- operands must be aligned in memory

## ❑ What makes it hard?

- structural hazards: resource conflicts, e.g. only one memory
  - ❑ sol: adding more resource
- control hazards: need to worry about branch instructions
  - ❑ sol: resolve earlier, predict branch
- data hazards: an instruction depends on a previous instruction
  - ❑ sol: forwarding, bypassing

# More about Pipeline

---

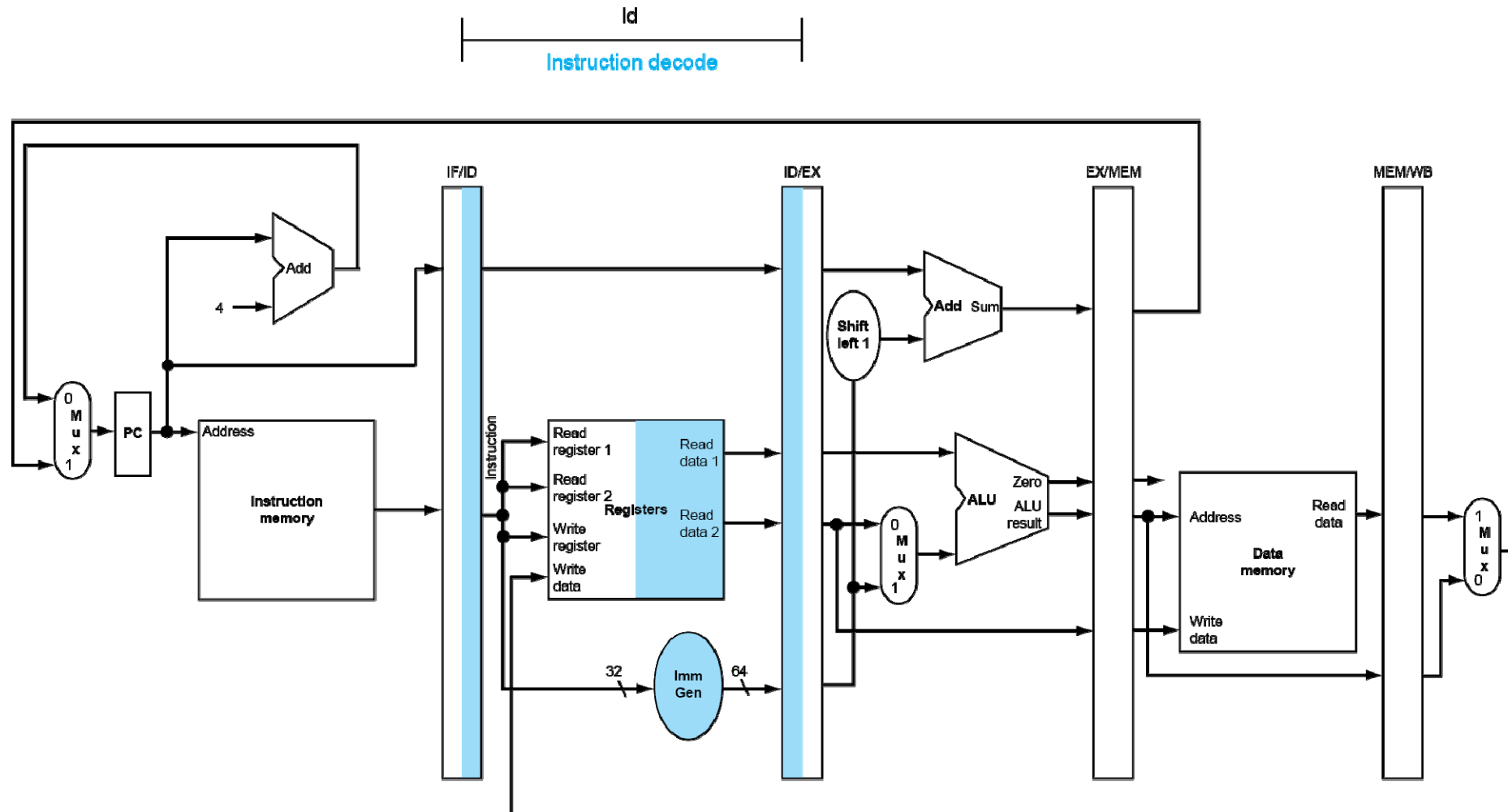
## ❑ Pipeline technique

- to exploit instruction-level parallelism (ILP)
- invisible to the programmers

## ❑ Pipeline Lessons

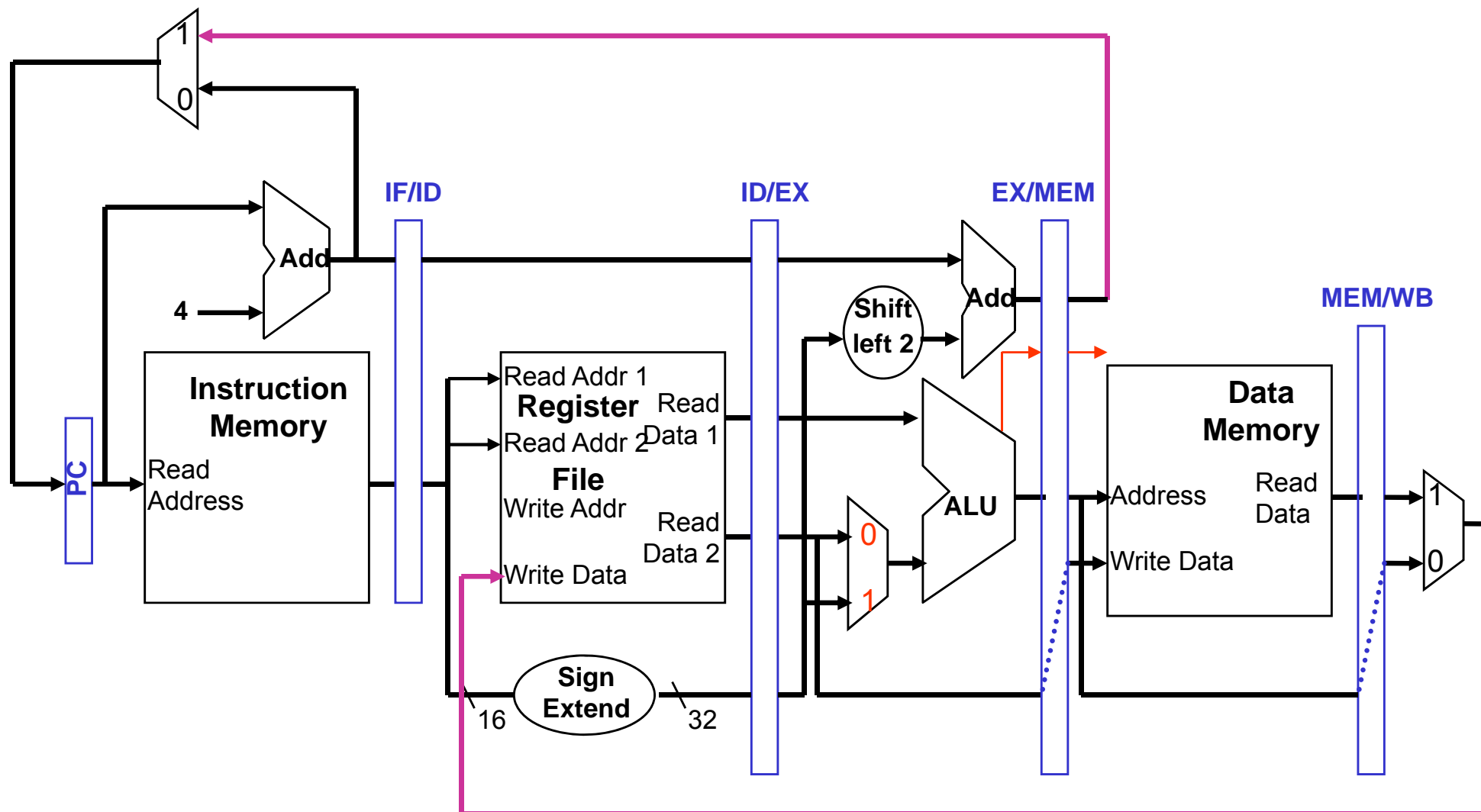
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to fill?pipeline and time to drain?it reduces speedup
- Stall for Dependences

# Go through the datapath with load/store

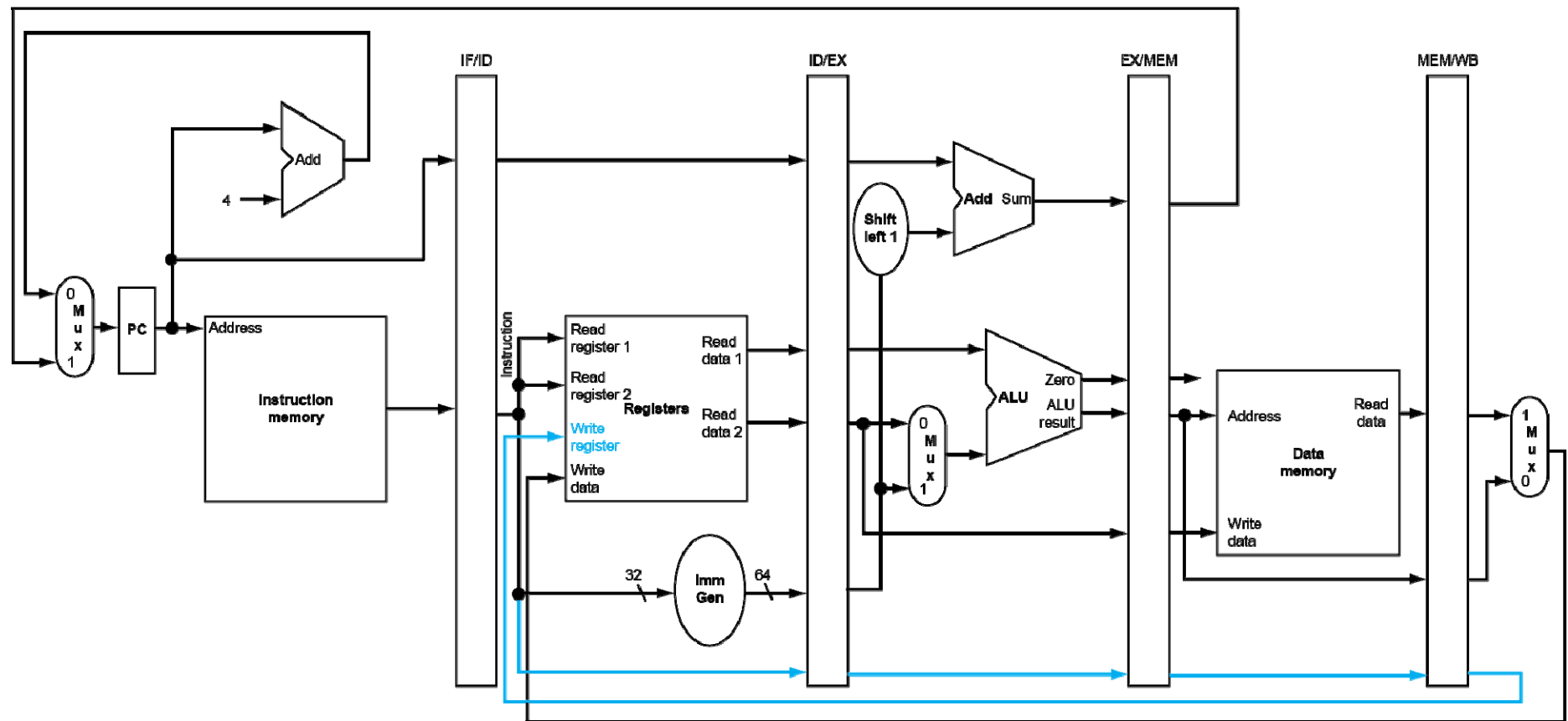


# A bug in the datapath?

- Which instruction gives the write register number?

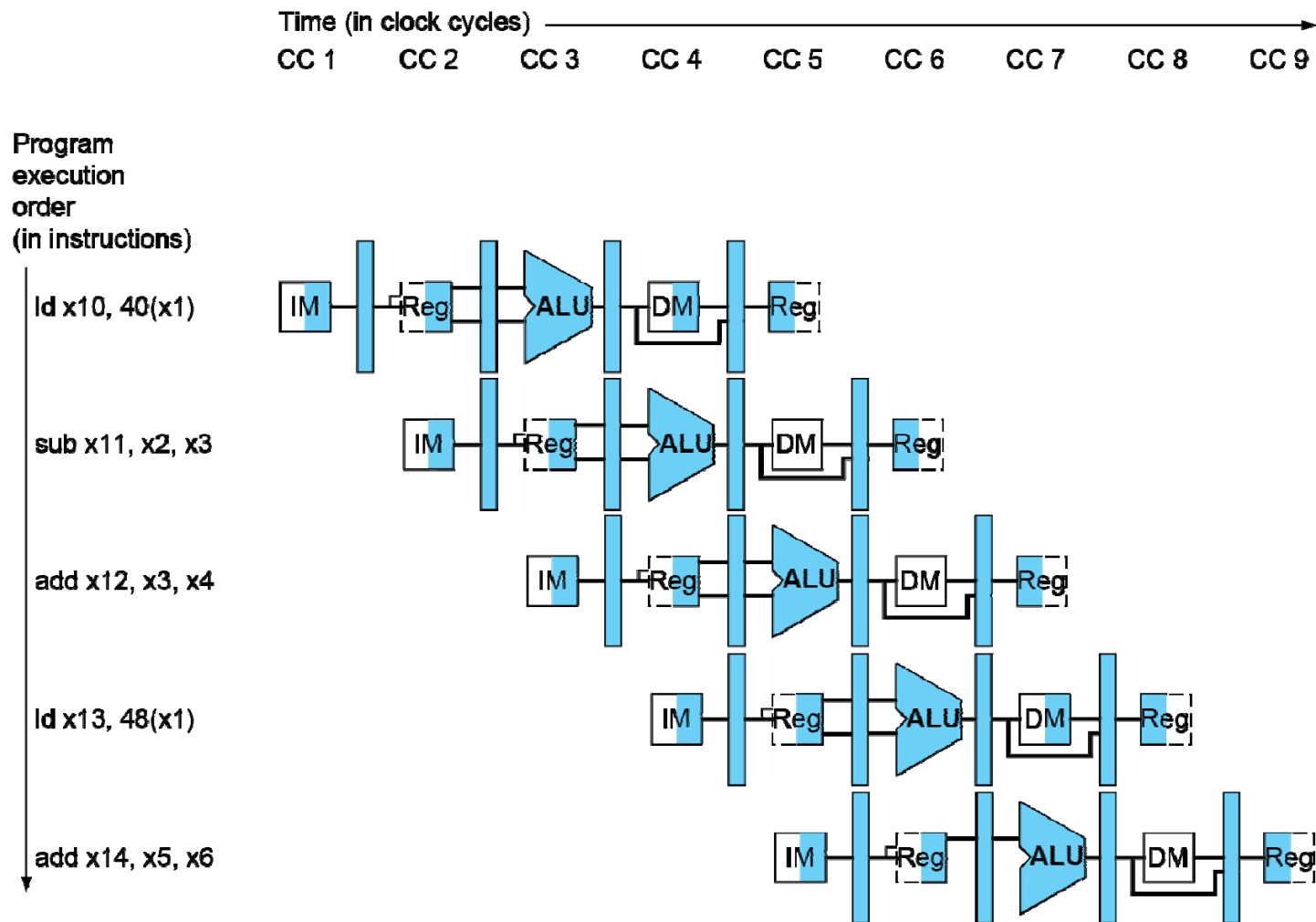


# Corrected Datapath for Load

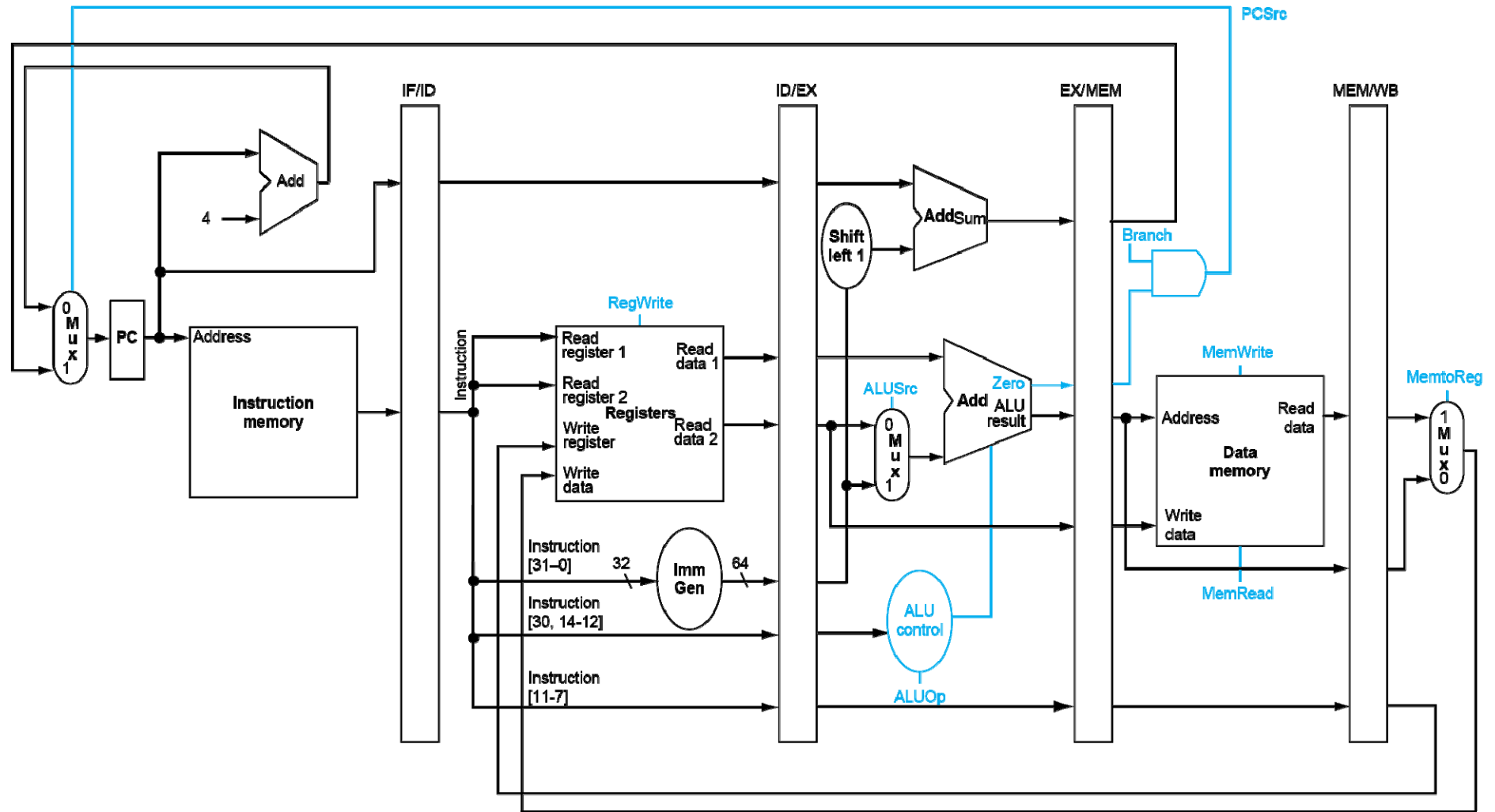


# Multi-Cycle Pipeline Diagram

- Form showing resource usage

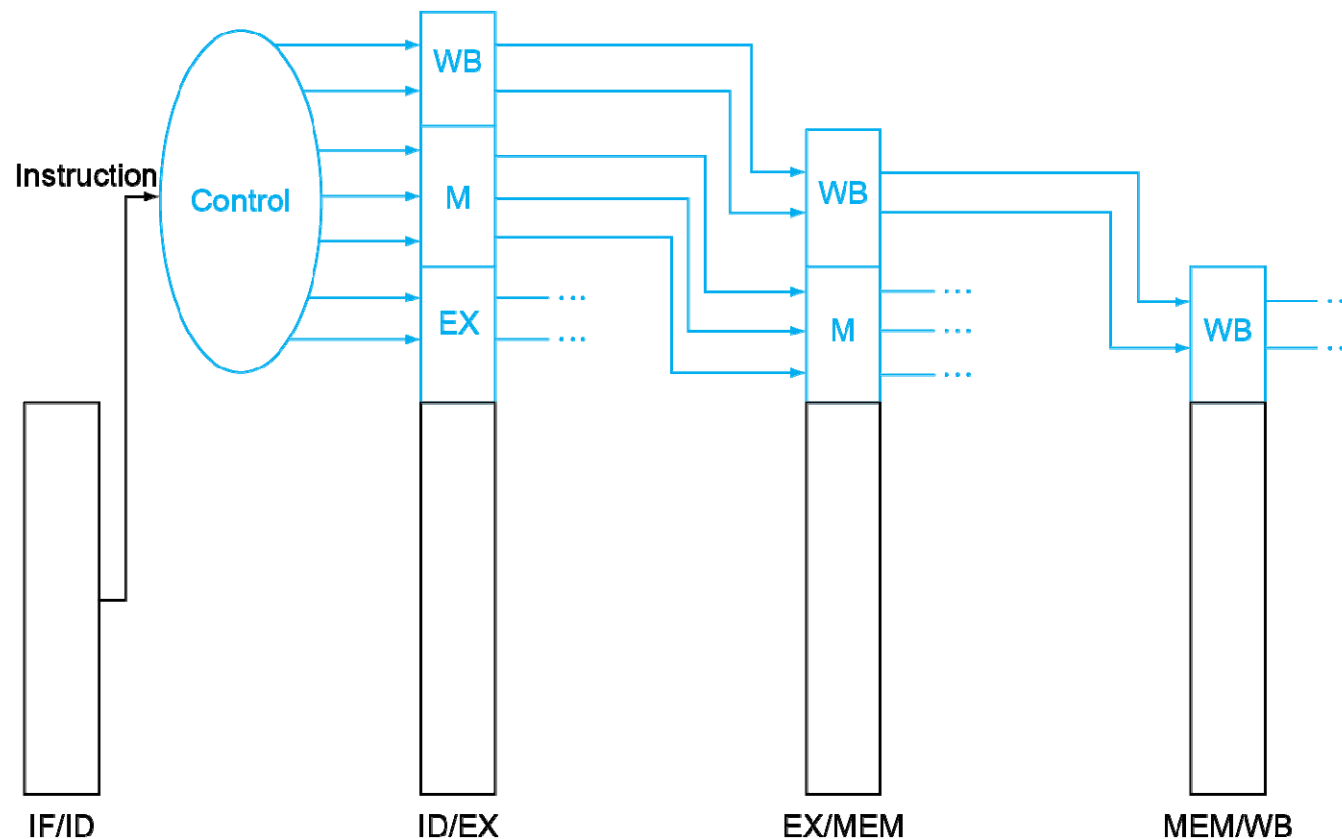


# Pipelined Control (Simplified)



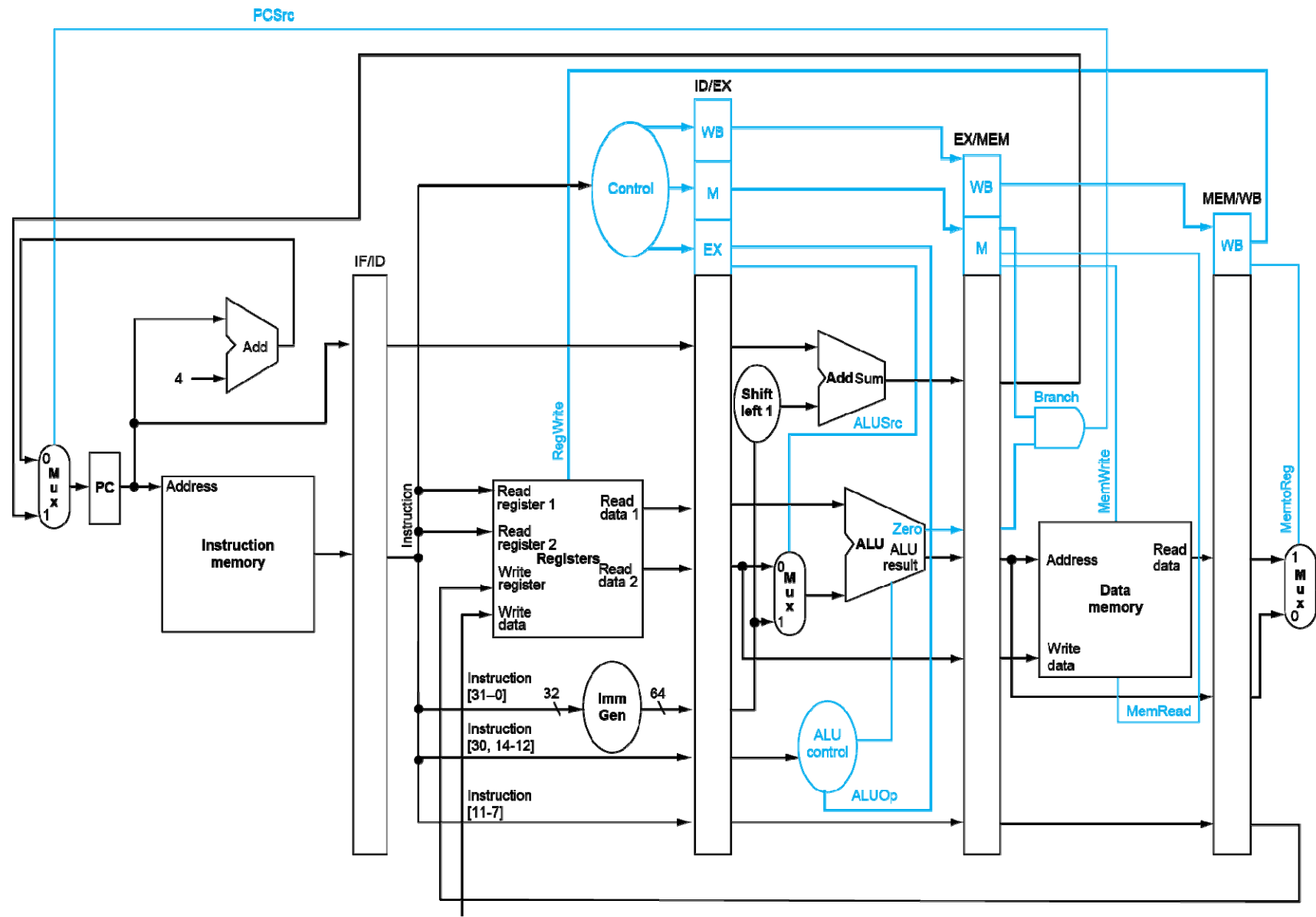
# Pipelined Control

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





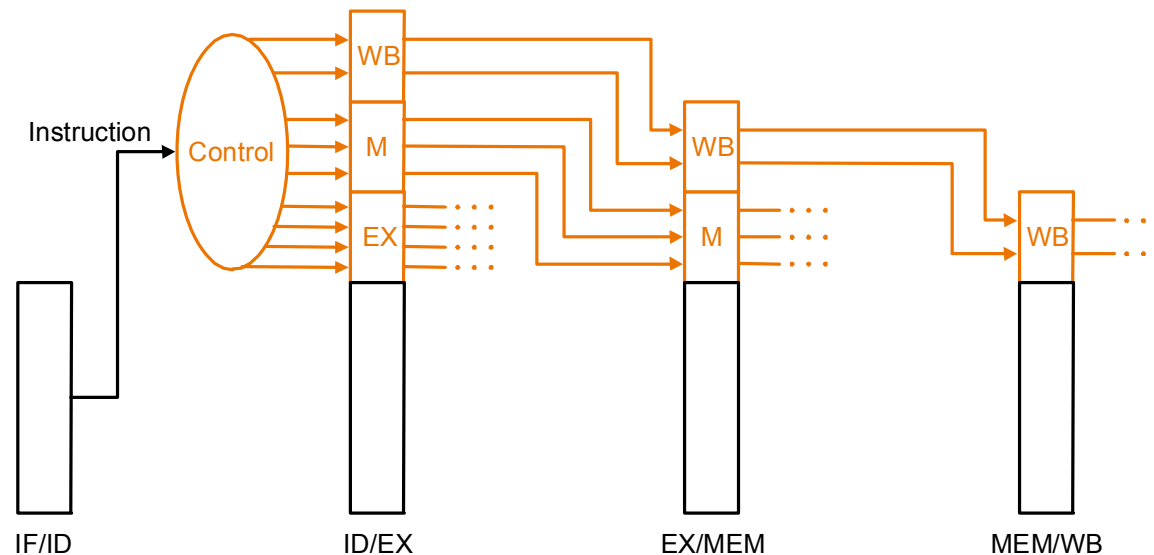
# Pipelined Control



# Pipelined Control

- Pass control signals along in pipeline registers just like the data

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X



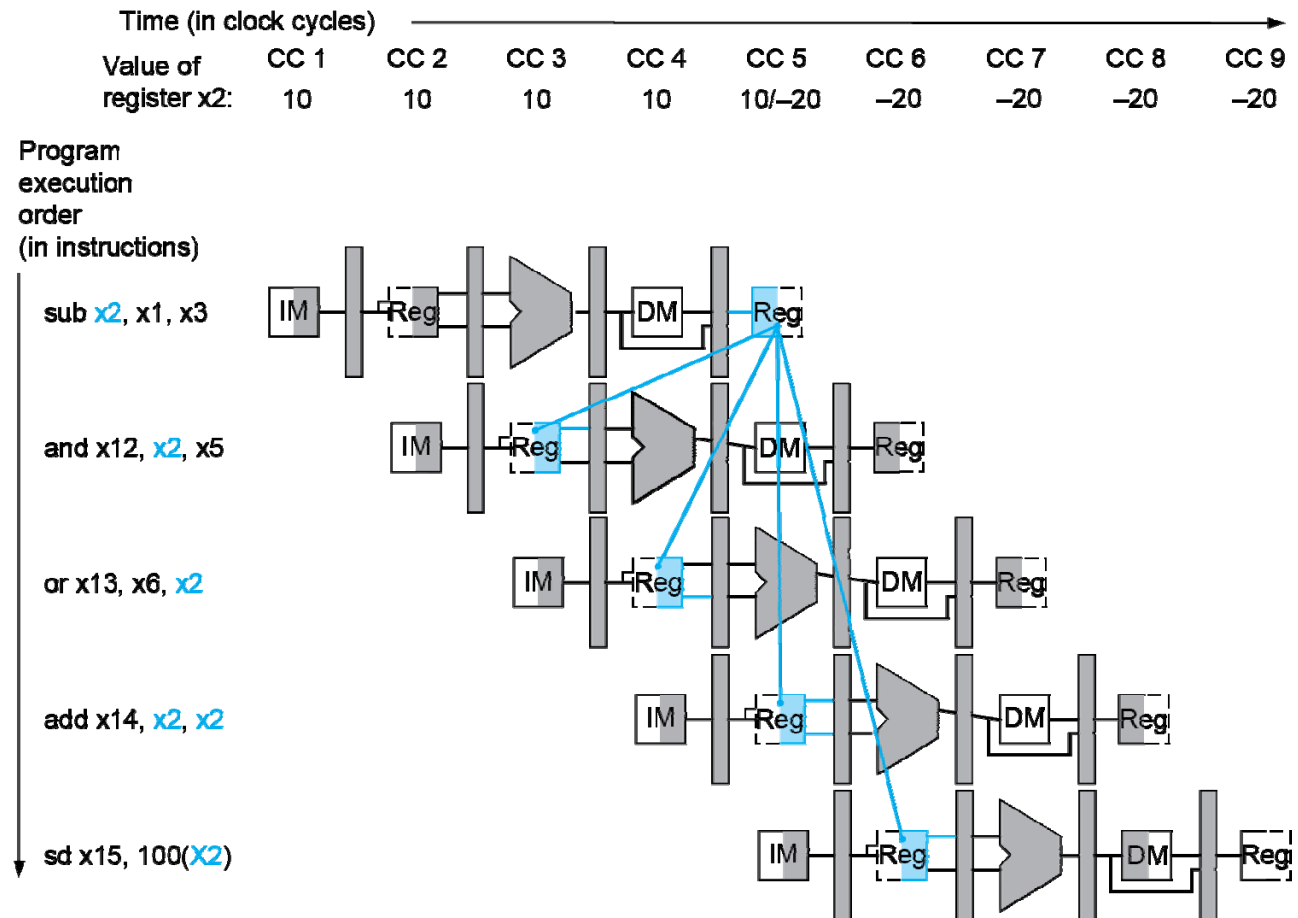
# Data Hazards in ALU Instructions

❑ 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?



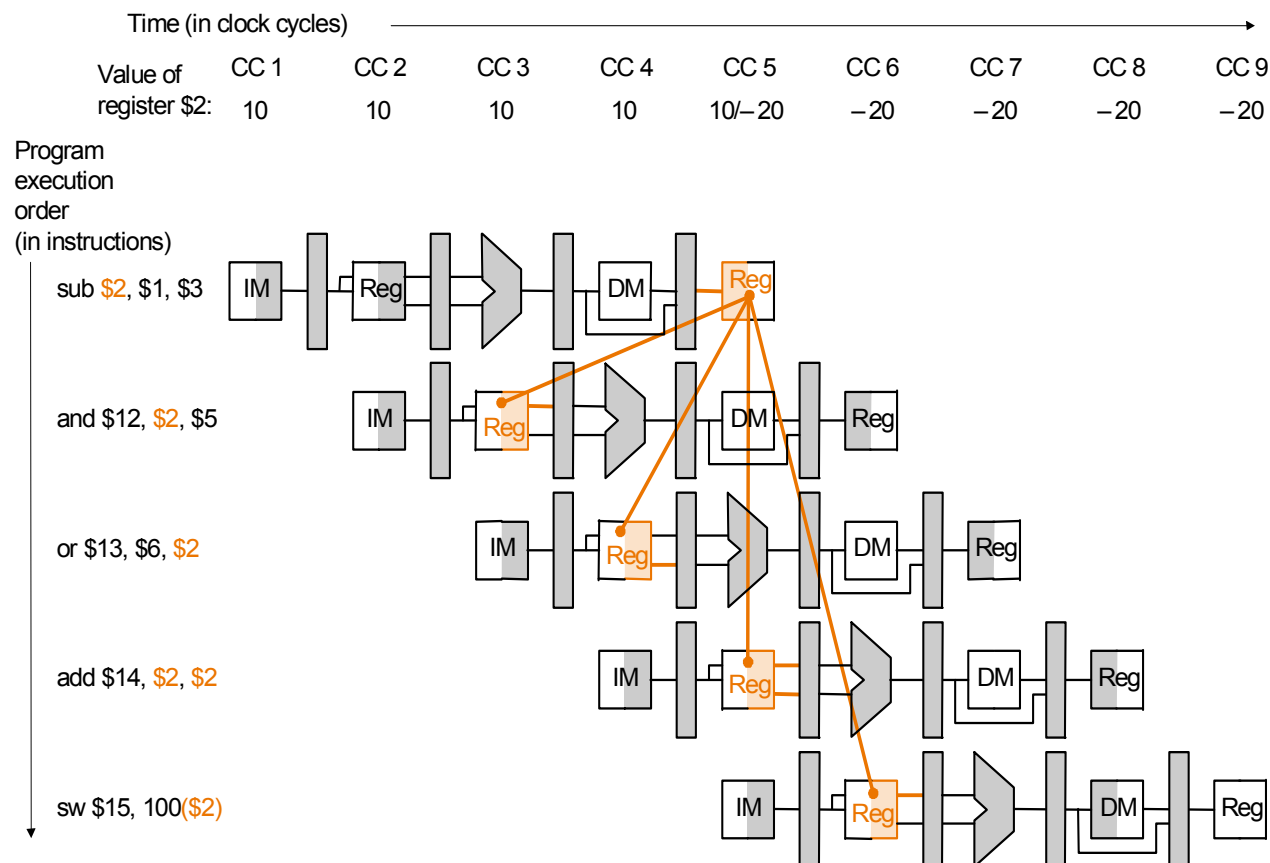
# Data Hazard on registers:

- Dependencies backwards in time are hazards

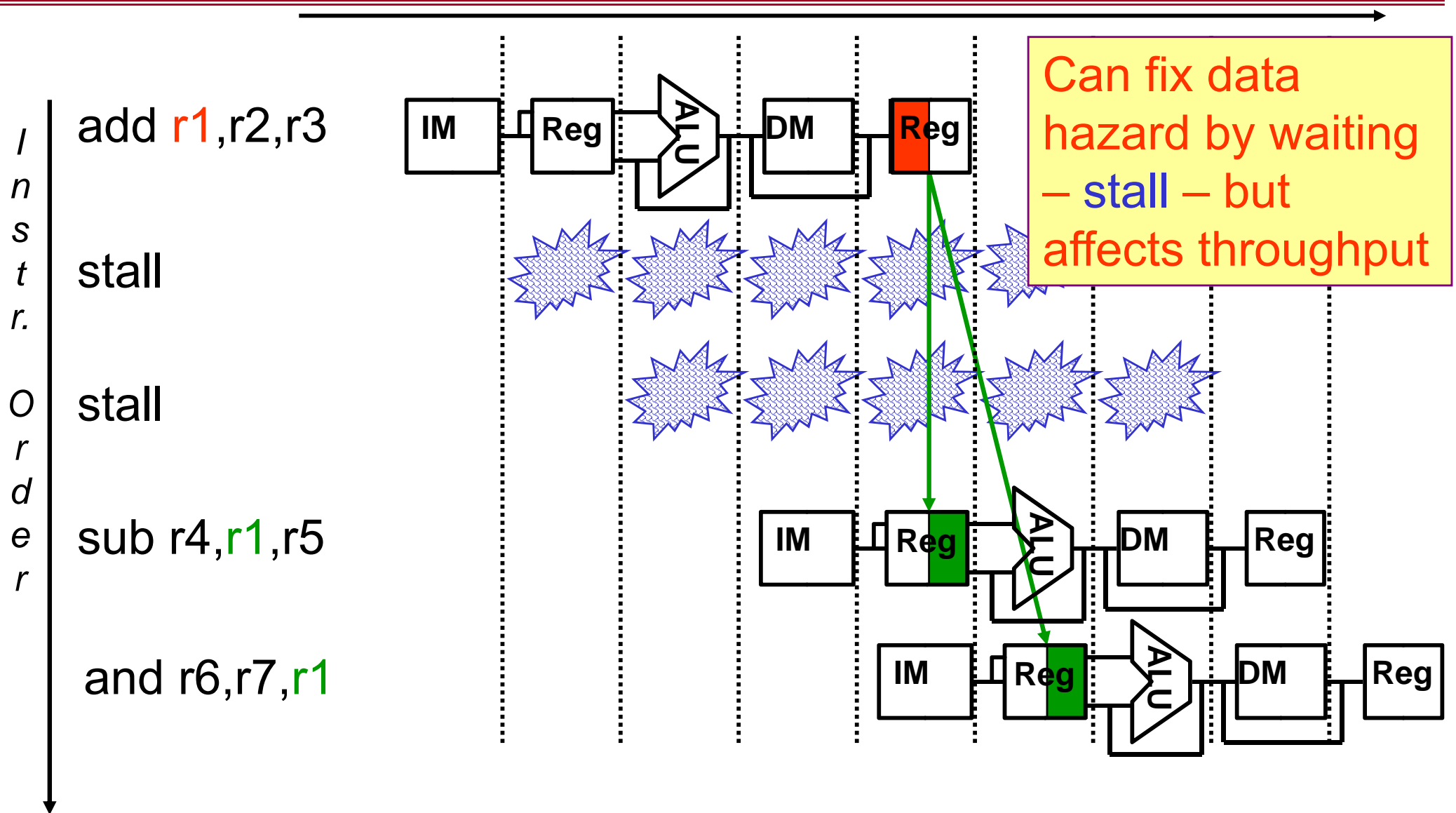
## Software solution

```

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

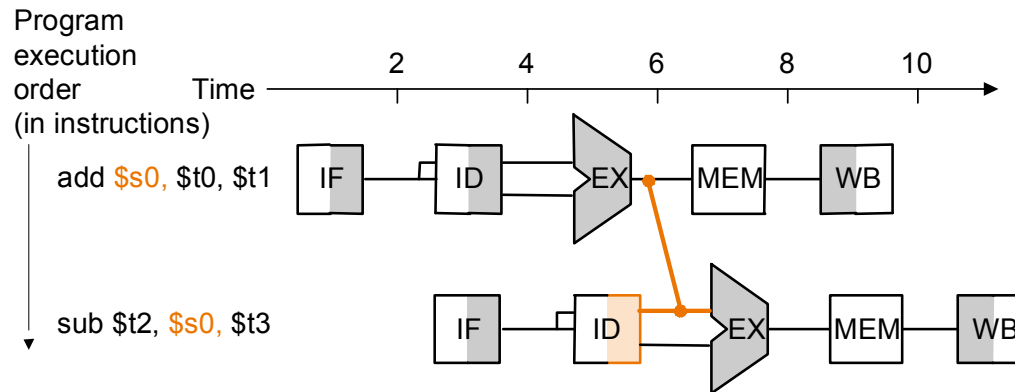


# Fix a data hazard by hardware stall

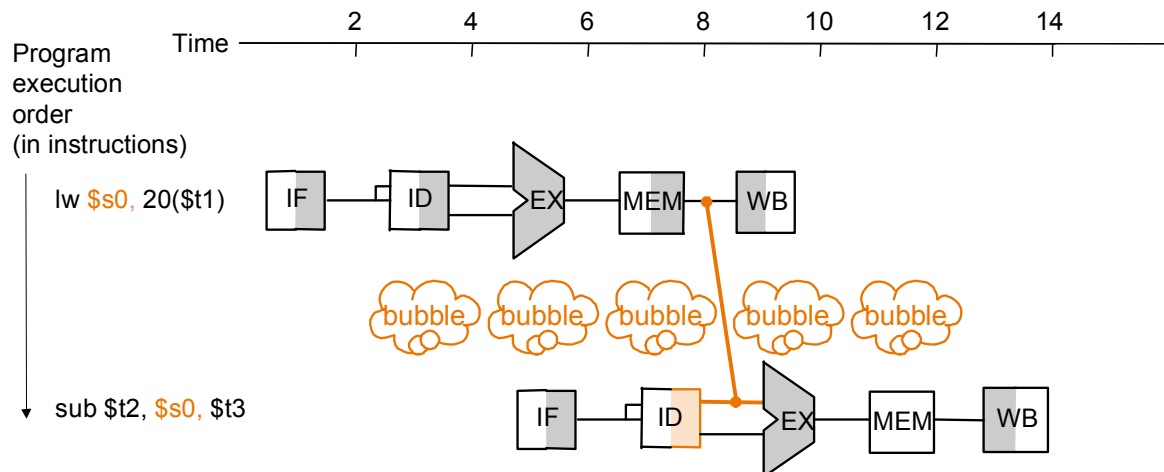


# Solving data hazards by forwarding

## Forwarding with two R-type instructions

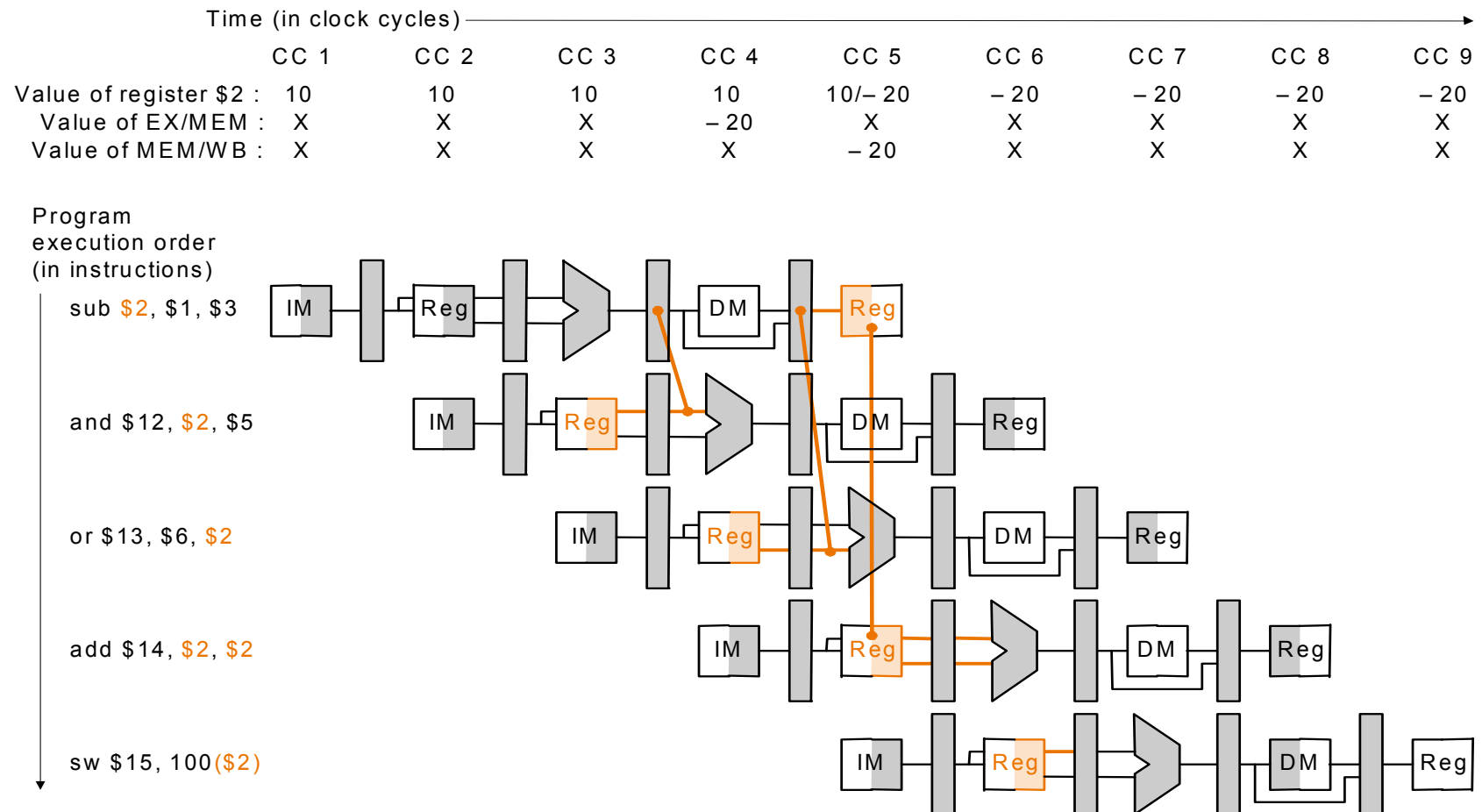


## Need a stall with forwarding when an R-type following a load



# Forwarding (or Bypassing):

- ❑ Use temporary results, don't wait for them to be written
  - register file forwarding to handle read/write to same register
  - ALU forwarding



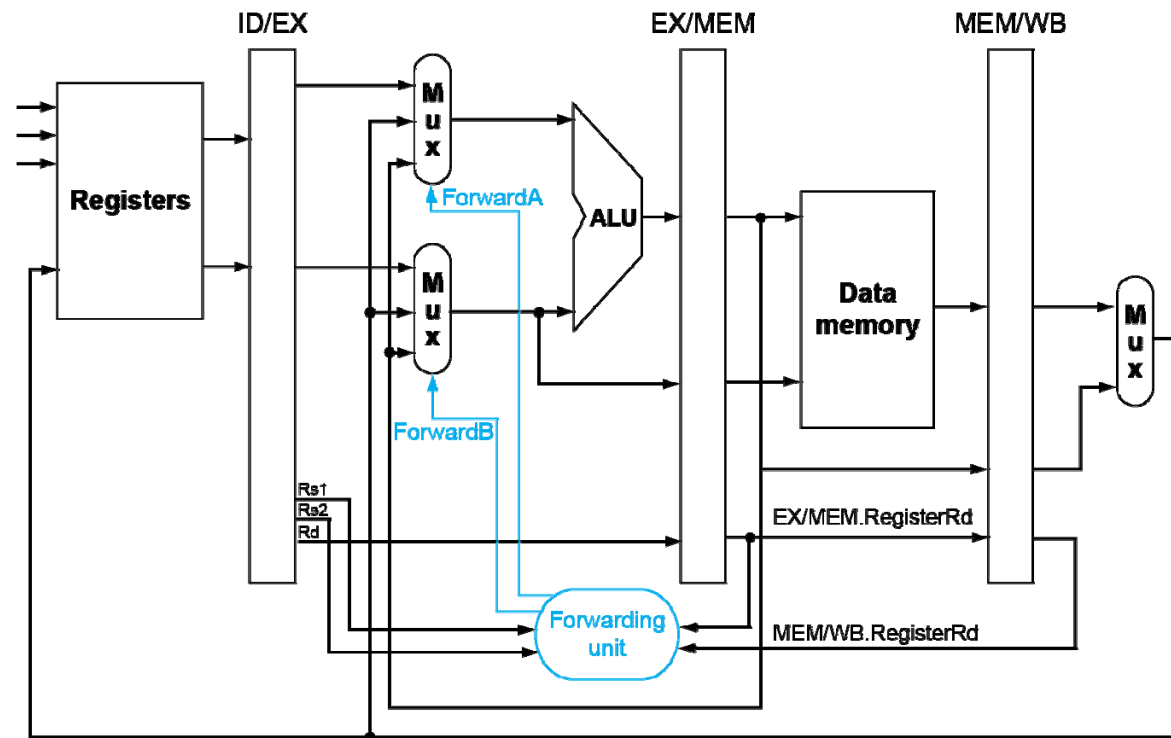
# Dependency detection and control

## ❑ Control value

Mux control	Source
ForwardA=00	ID/EX
ForwardA=10	EX/MEM
ForwardA=01	MEM/WB
ForwardB=00	ID/EX
ForwardB=10	EX/MEM
ForwardB=01	MEM/WB

## ❑ Notation

- EX/Mem  
Pipeline register between EXE and Memory
- “RegisterRd” is number of register to be written (RD)
- “RegisterRs1” is number of RS1 register
- “RegisterRs2” is number of RS2 register
- “ForwardA, ForwardB” controls forwarding muxes





# Data Forwarding Control Conditions (1/5)

## ❑ EX/MEM hazard:

if (EX/MEM.RegisterRd == ID/EX.RegisterRs1))

ForwardA = 10

if (EX/MEM.RegisterRd == ID/EX.RegisterRs2))

ForwardB = 10

## ❑ MEM/WB hazard:

if (MEM/WB.RegisterRd == ID/EX.RegisterRs1))

ForwardA = 01

if (MEM/WB.RegisterRd == ID/EX.RegisterRs2))

ForwardB = 01

Running with:

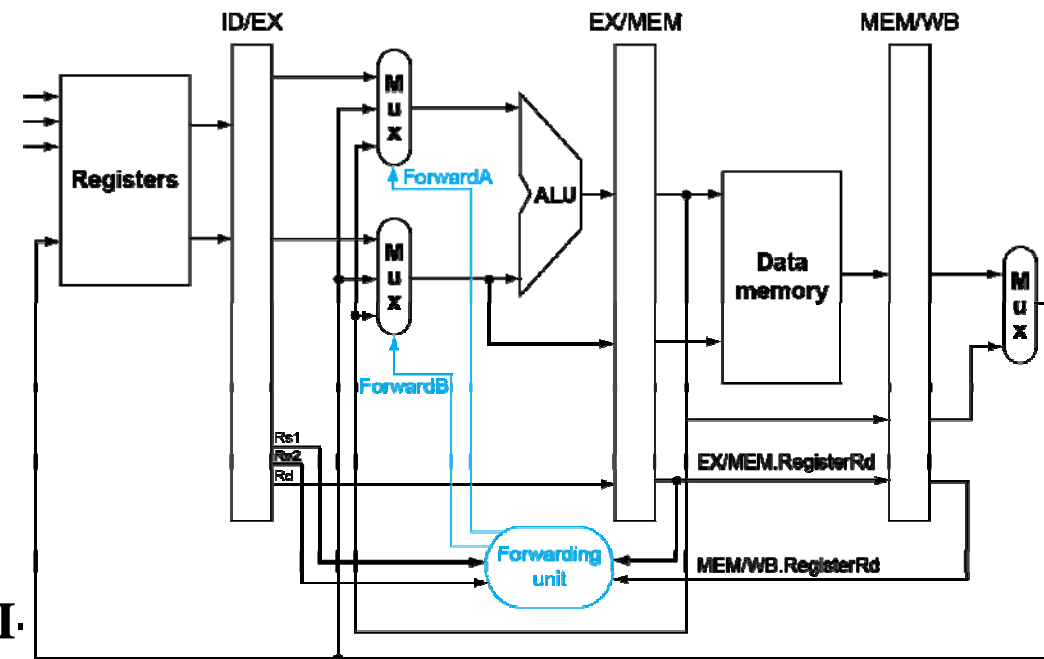
sub x2, x1, x3

and x12, x2, x5

or x13, x6, x2

add x14, x2, x2

sw x15, 100(x2)



# Data Forwarding Control Conditions (2/5)

---

## ❑ EX/MEM hazard:

```
if (EX/MEM.RegWrite
    && (EX/MEM.RegisterRd == ID/EX.RegisterRs1))
    ForwardA = 10
if (EX/MEM.RegWrite
    && (EX/MEM.RegisterRd == ID/EX.RegisterRs2))
    ForwardB = 10
```

Running with:

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

## ❑ MEM/WB hazard:

```
if (MEM/WB.RegWrite && (MEM/WB.RegisterRd == ID/EX.RegisterRs1))
    ForwardA = 01
if (MEM/WB.RegWrite
    && (MEM/WB.RegisterRd == ID/EX.RegisterRs2))
    ForwardB = 01
```

# Data Forwarding Control Conditions (3/5)

---

## ❑ EX/MEM hazard:

if (**EX/MEM.RegWrite** and (**EX/MEM.RegisterRd != 0**

&& (EX/MEM.RegisterRd == ID/EX.RegisterRs))

ForwardA = 10

if (**EX/MEM.RegWrite** and (**EX/MEM.RegisterRd != 0**

&& (EX/MEM.RegisterRd == ID/EX.RegisterRt))

ForwardB = 10

Running with:

sub x2, x1, x3

and x12, x2, x5

or x13, x6, x2

add x14, x2, x2

sw x15, 100(x2)

## ❑ MEM/WB hazard:

if (**MEM/WB.RegWrite** and (**MEM/WB.RegisterRd != 0**

&& (MEM/WB.RegisterRd == ID/EX.RegisterRs))

ForwardA = 01

if (**MEM/WB.RegWrite** and (**MEM/WB.RegisterRd != 0**

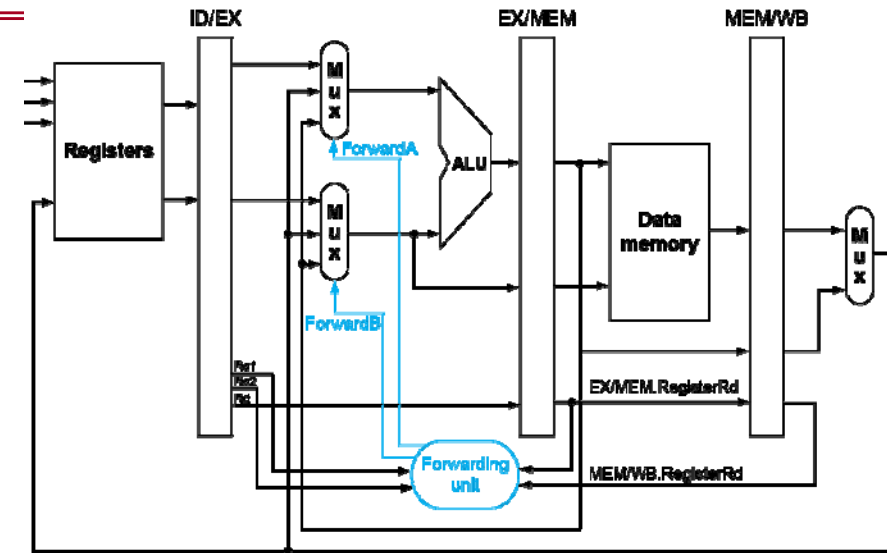
&& (MEM/WB.RegisterRd == ID/EX.RegisterRt))

ForwardB = 01

# Data Forwarding Control Conditions (4/5)

## Control value

Mux control	Source
ForwardA=00	ID/EX
ForwardA=10	EX/MEM
ForwardA=01	MEM/WB
ForwardB=00	ID/EX
ForwardB=10	EX/MEM



## Detection

### EX hazard

- if (Ex/Mem.Regwrite) && (Ex/Mem.RegisterRd!=0) && (Ex/Mem.RegisterRd ==ID/Ex.RegisterRs1))  
ForwardA=10
- if (Ex/Mem.Regwrite) && (Ex/Mem.RegisterRd!=0) && (Ex/Mem.RegisterRd ==ID/Ex.RegisterRs2))  
ForwardB=10

### Mem hazard

- if (Mem/Wb.Regwrite) && (Mem/Wb.RegRd!=0) && (Ex/Mem.RegRd != ID/Ex.RegRs1) &&  
(Mem/Wb.RegRd ==ID/Ex.RegRs1))  
ForwardA=01

# Data Forwarding Control Conditions (5/5)

## Control value

Mux control	Source
ForwardA=00	ID/EX
ForwardA=10	EX/MEM
ForwardA=01	MEM/WB
ForwardB=00	ID/EX
ForwardB=10	EX/MEM
ForwardB=01	MEM/WB

## Detection

### EX hazard

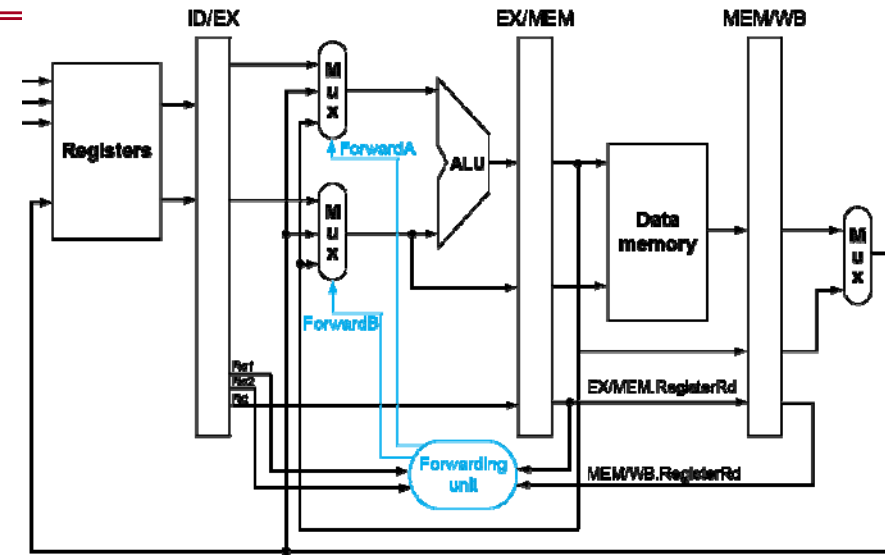
- if (Ex/Mem.Regwrite) && (Ex/Mem.RegisterRd!=0) && (Ex/Mem.RegisterRd ==ID/Ex.RegisterRs1))  
ForwardA=10

### Mem hazard

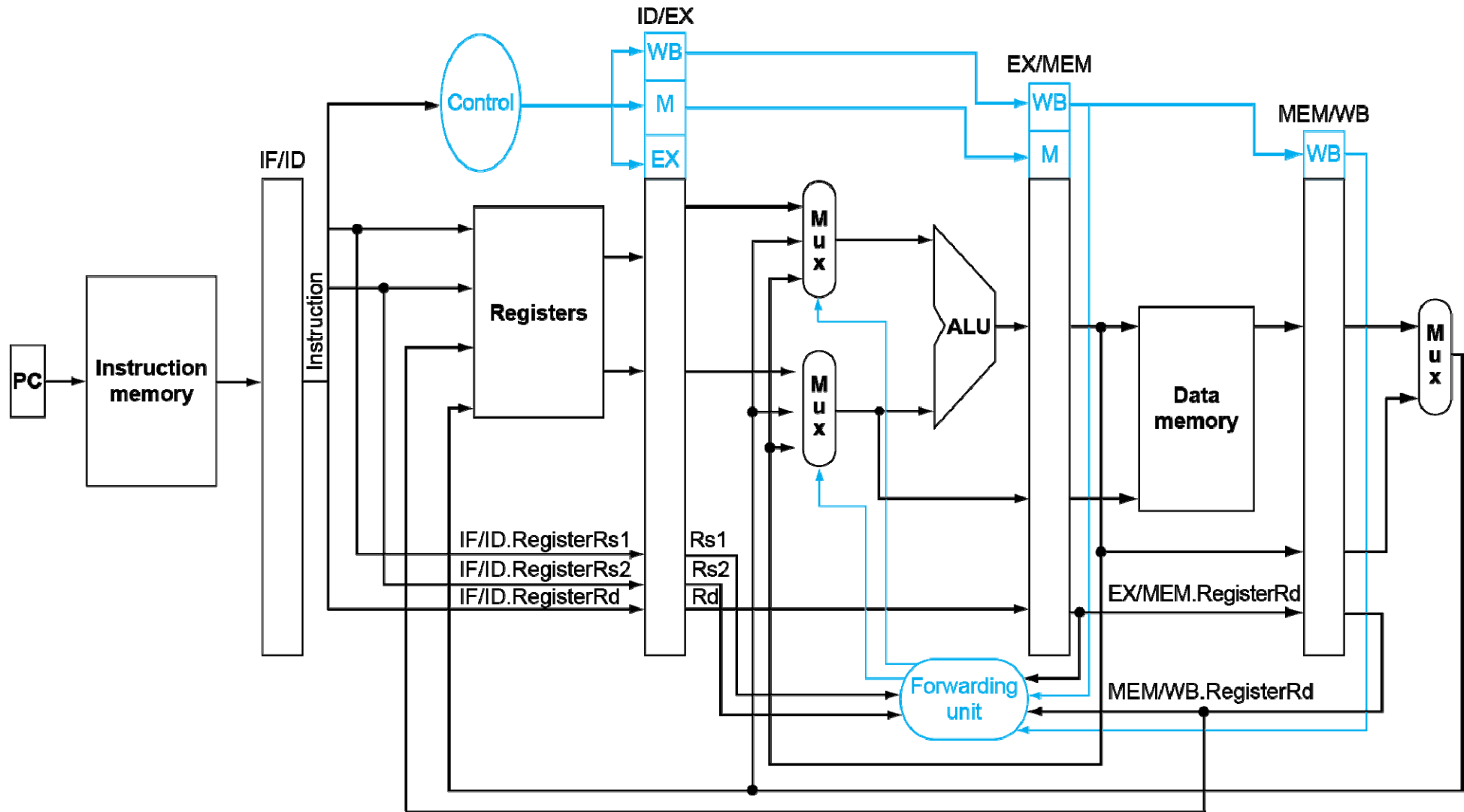
- if (Mem/Wb.Regwrite) && (Mem/Wb.RegRd!=0) &&

**!(Ex/Mem.Regwrite) && (Ex/Mem.RegisterRd!=0)) && (Ex/Mem.RegisterRd !=ID/Ex.RegisterRs1))**  
**&& (Mem/Wb.RegRd ==ID/Ex.RegRs1))**

ForwardA=01

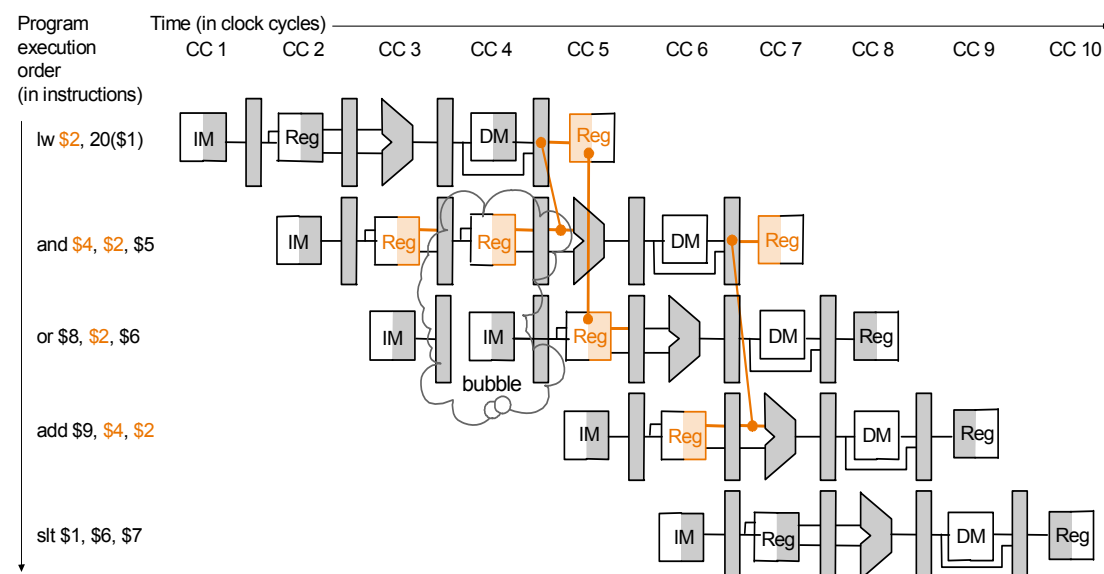
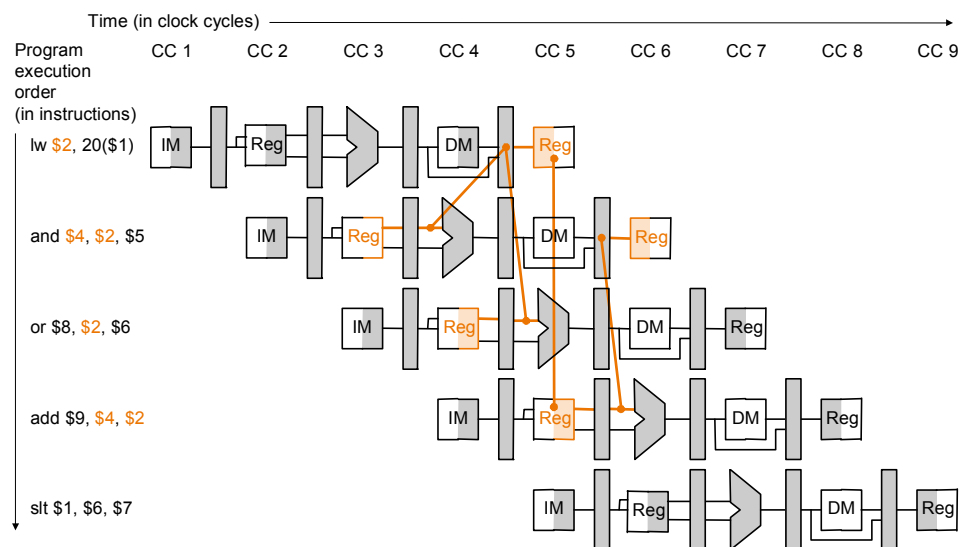


# Datapath with forwarding

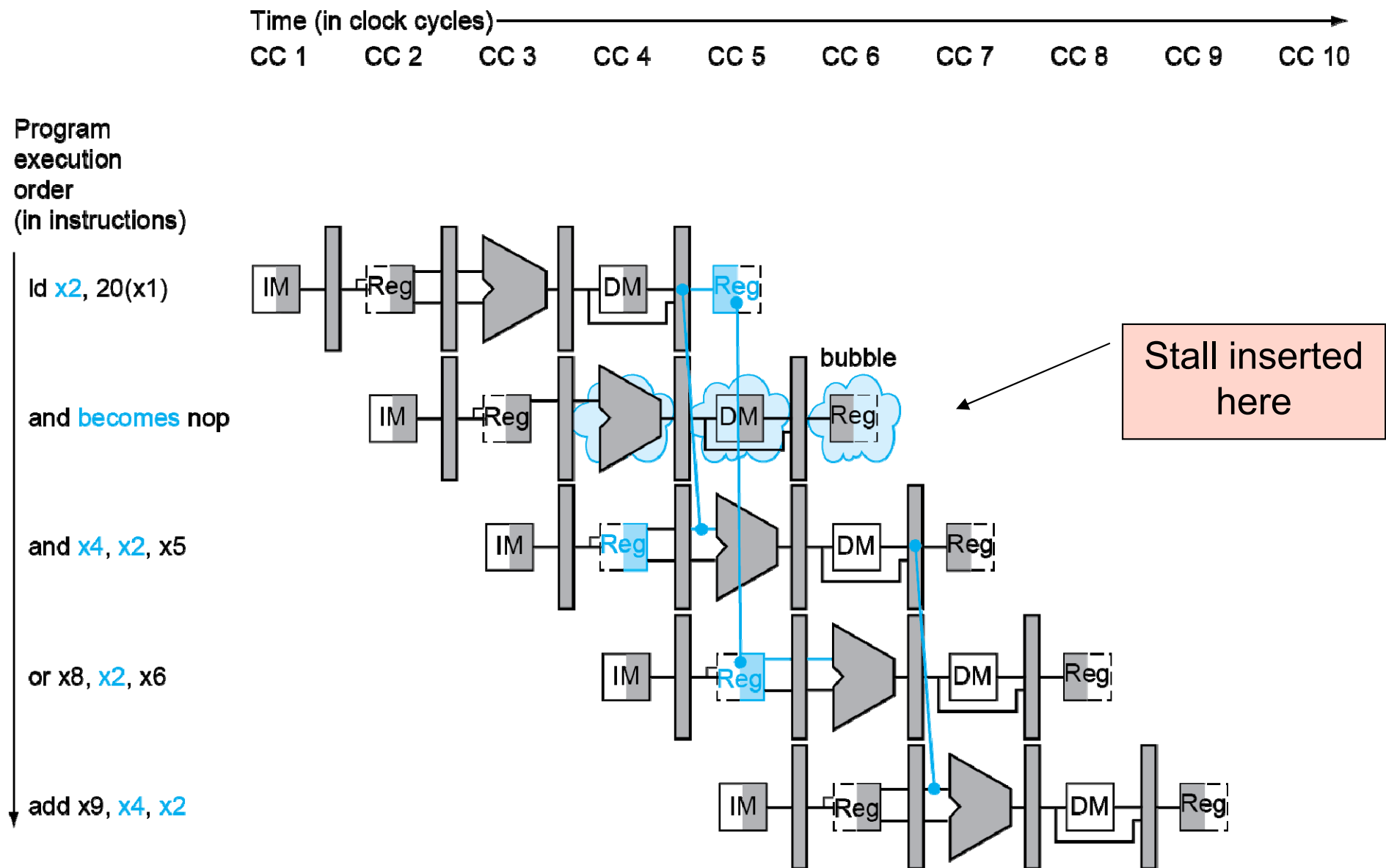


# Data Hazards and stall

- ❑ Load word can still cause a hazard



# Load-Use Data Hazard





# Load-use Hazard Detection Unit

---

- ❑ Need a **hazard detection unit** in the ID stage that inserts a stall between the load and its use

ID Hazard Detection

```
if (ID/EX.MemRead
    and ((ID/EX.RegisterRd == IF/ID.RegisterRs1)
        or (ID/EX.RegisterRd == IF/ID.RegisterRs2)))
    stall the pipeline
```

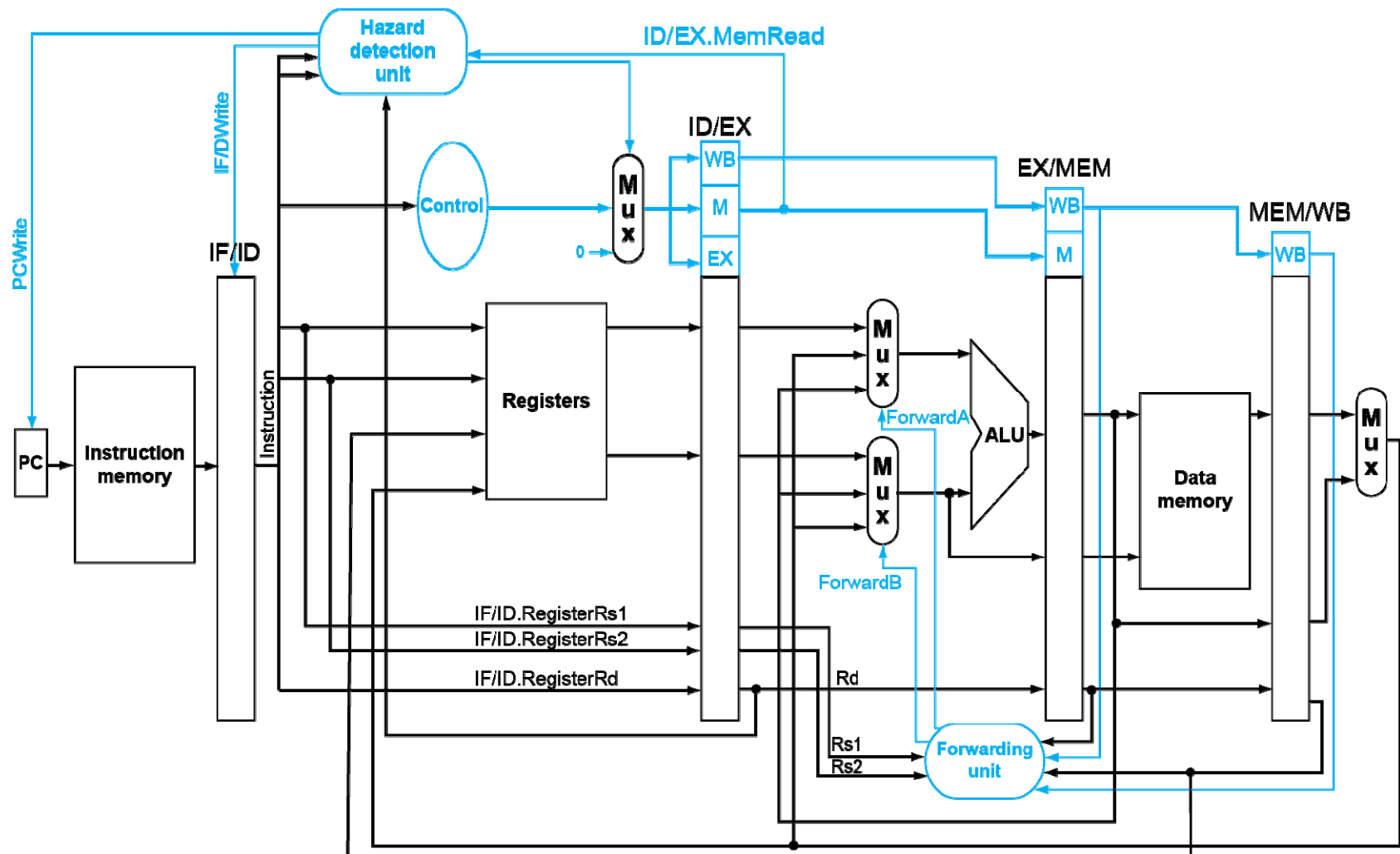
- ❑ The first line tests to see if the instruction is a load;
- ❑ Next two lines check to see if the destination register of the load in the EX stage matches either source registers of the instruction in the ID stage
- ❑ After this 1-cycle stall, the forwarding logic can handle the remaining data hazards

# Stall Hardware

---

- ❑ Prevent the IF and ID stage instructions from making progress down the pipeline, done by preventing the PC register and the IF/ID pipeline register from changing
  - Hazard detection unit controls the writing of the PC and IF/ID registers
- ❑ The instructions in the back half of the pipeline starting with the EX stage must be flushed (execute `noop`)
  - Must deassert the control signals (setting them to 0) in the EX, MEM, and WB control fields of the ID/EX pipeline register.
  - Hazard detection unit controls the multiplexor that chooses between the real control values and 0's.
  - Assume that 0's are benign values in datapath: nothing changes

# Datapath with forwarding and hazard detection

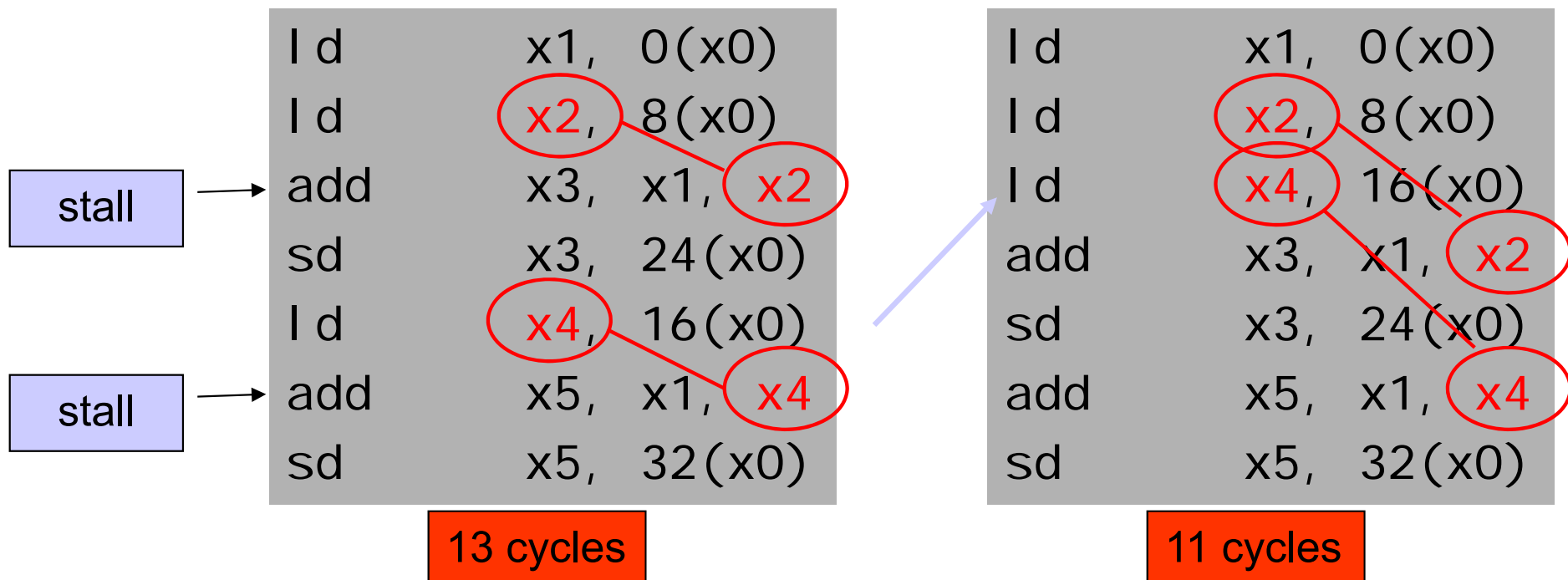


## – Stall detection

- if  $(ID/EX.MemRead \ \&\& \ (ID/Ex.RegRd == IF/ID.RegRs1 \ || \ ID/Ex.RegRd == IF/ID.RegRs2))$   
stall the pipeline

# Code Scheduling to Avoid Stalls

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



# Mapping the control by breaking executions into 5 steps – Appendix D

---

## ❑ Instruction Fetch

$IR = \text{Memory}[PC];$   
 $PC = PC + 4;$

## ❑ Instruction Decode and Register Fetch

$A = \text{Reg}[IR[25-21]];$   
 $B = \text{Reg}[IR[20-16]];$   
 $ALUOut = PC + (\text{sign-ext}(IR[15-0]) \ll 2)$

## ❑ Execution,

Memory Reference:  
 $ALUOut = A + \text{sign-extend}(IR[15-0]);$

R-type:  
 $ALUOut = A \text{ op } B;$

Branch:  
if  $(A == B)$   $PC = ALUOut;$

## ❑ Memory Access or R-type instruction completion

Loads/stores access memory

$MDR = \text{Memory}[ALUOut];$   
or  
 $\text{Memory}[ALUOut] = B;$

R-type instructions finish  
 $\text{Reg}[IR[15-11]] = ALUOut;$

## ❑ Write-back step

$\text{Reg}[IR[20-16]] = MDR;$

# Summary of steps

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	$IR = \text{Memory}[PC]$ $PC = PC + 4$			
Instruction decode/register fetch	$A = \text{Reg}[IR[25-21]]$ $B = \text{Reg}[IR[20-16]]$ $ALUOut = PC + (\text{sign-extend}(IR[15-0]) \ll 2)$			
Execution, address computation, branch/jump completion	$ALUOut = A \text{ op } B$	$ALUOut = A + \text{sign-extend}(IR[15-0])$	if $(A == B)$ then $PC = ALUOut$	$PC = PC[31-28] \parallel (IR[25-0] \ll 2)$
Memory access or R-type completion	$\text{Reg}[IR[15-11]] = ALUOut$	Load: $MDR = \text{Memory}[ALUOut]$ or Store: $\text{Memory}[ALUOut] = B$		
Memory read completion		Load: $\text{Reg}[IR[20-16]] = MDR$		

## ❑ In multicycle implementation

- empty entries indicate taking fewer cycles
- a new instruction will be started as soon as the current instruction completes

# Defining the control

---

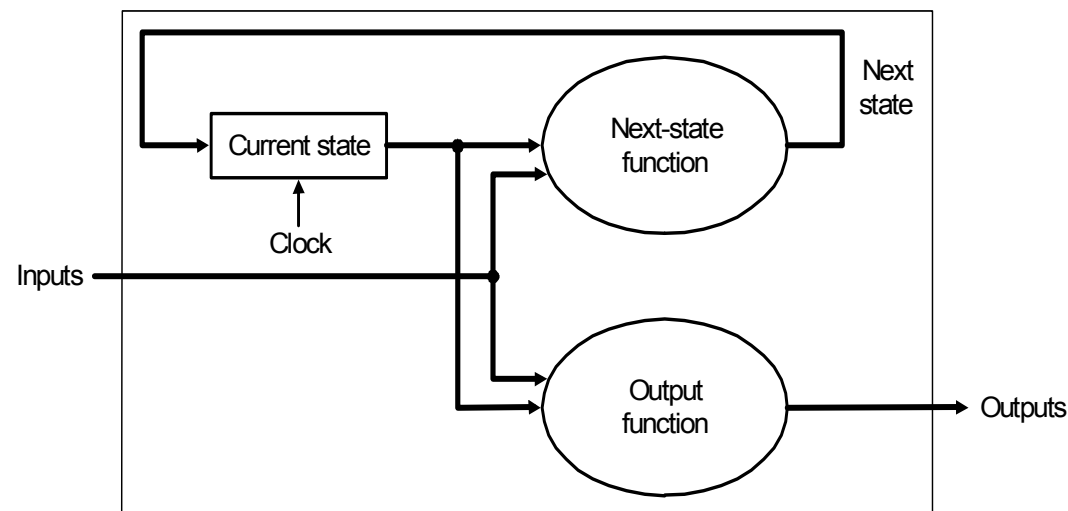
- ❑ Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- ❑ Implementation can be derived from specification
- ❑ Two approaches
  - Finite state machine
    - ❑ a set of states
    - ❑ direction - next-state functions
  - Microprogramming
- ❑ Both allow implementation using gates, ROM, PLA

# Review: finite state machines

---

## □ Finite state machines:

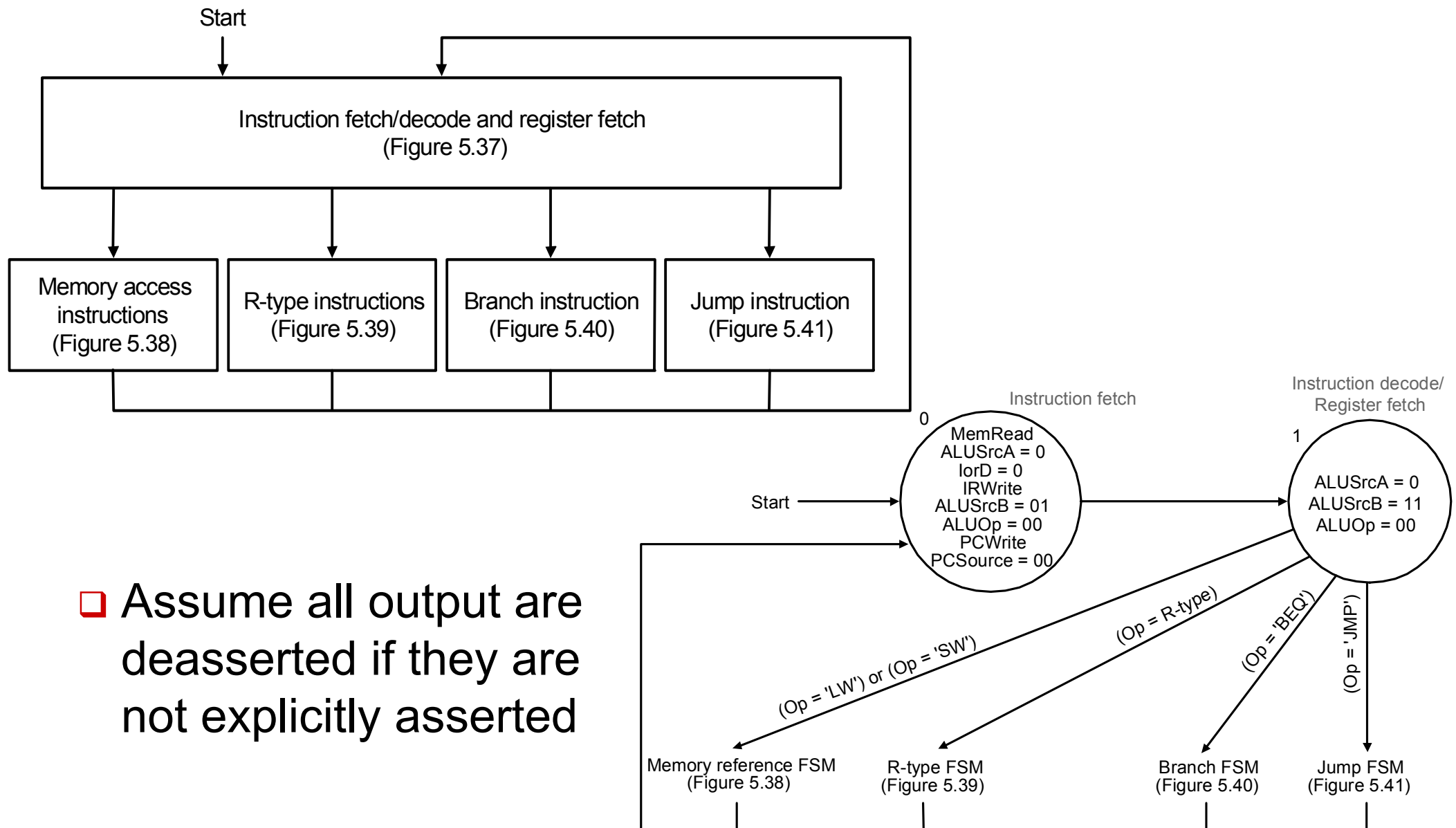
- a set of states and
- next state function (determined by current state and the input)
- output function (determined by current state and possibly input)



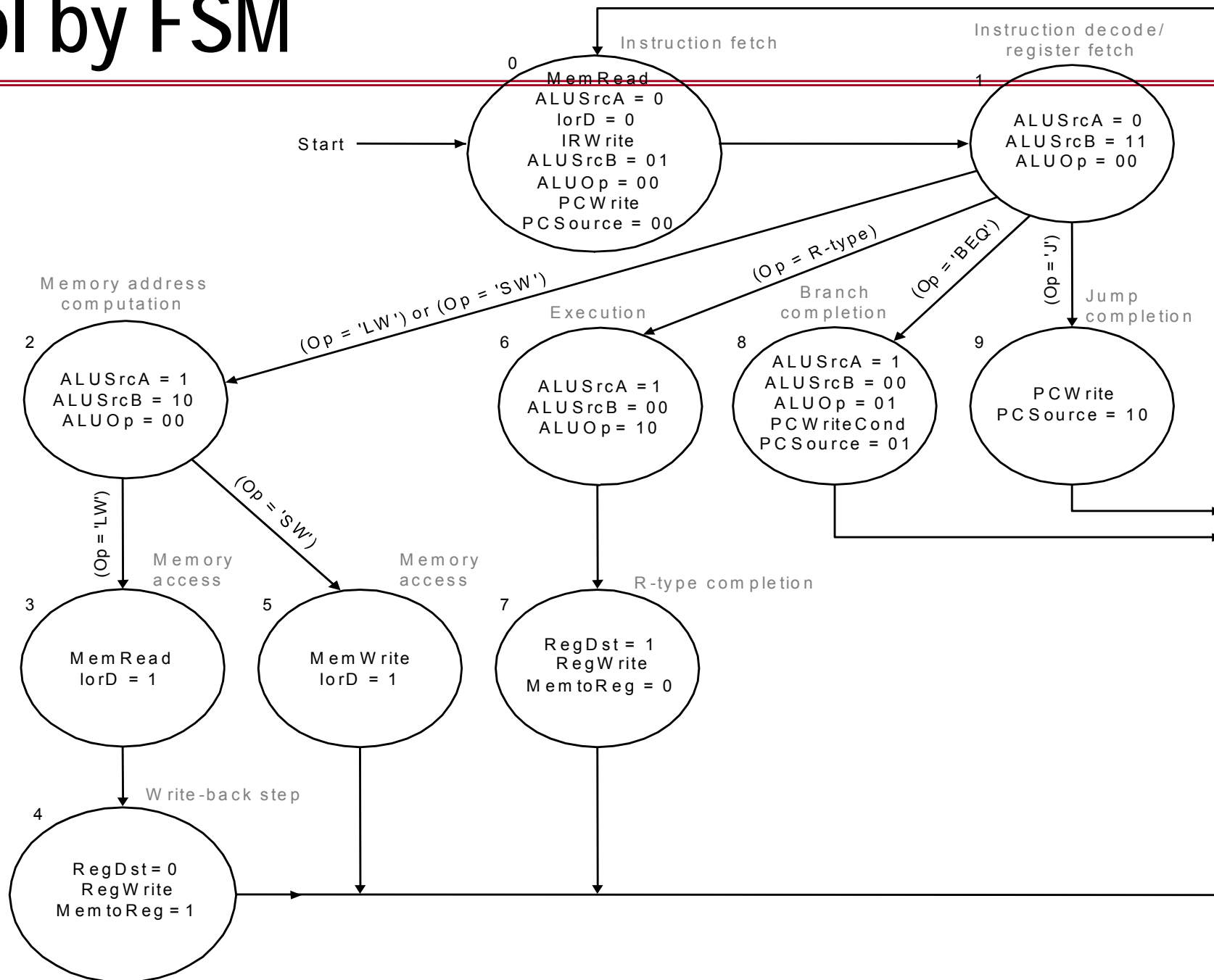
- We'll use a Moore machine (output based only on current state)



# Defined by finite state machine (FSM)



# Control by FSM



# FSM Implementation

## ❑ FSM can be implemented with

- a temporary register  
hold current state
- a block of combination logic that determines:
  - ❑ signals to be asserted
  - ❑ the next state

## ❑ Moore machine

- output depends only on current states

## ❑ Mealy machine

- use input and current state to determine output

