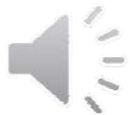


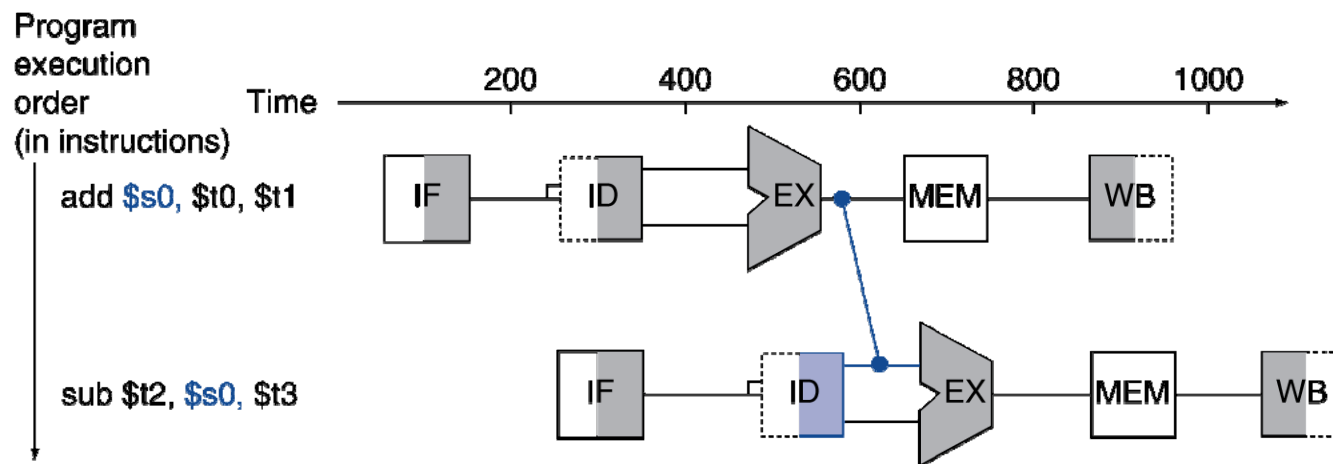
# Chapter 4

## The Processor



# 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



# Data Hazards in ALU Instructions

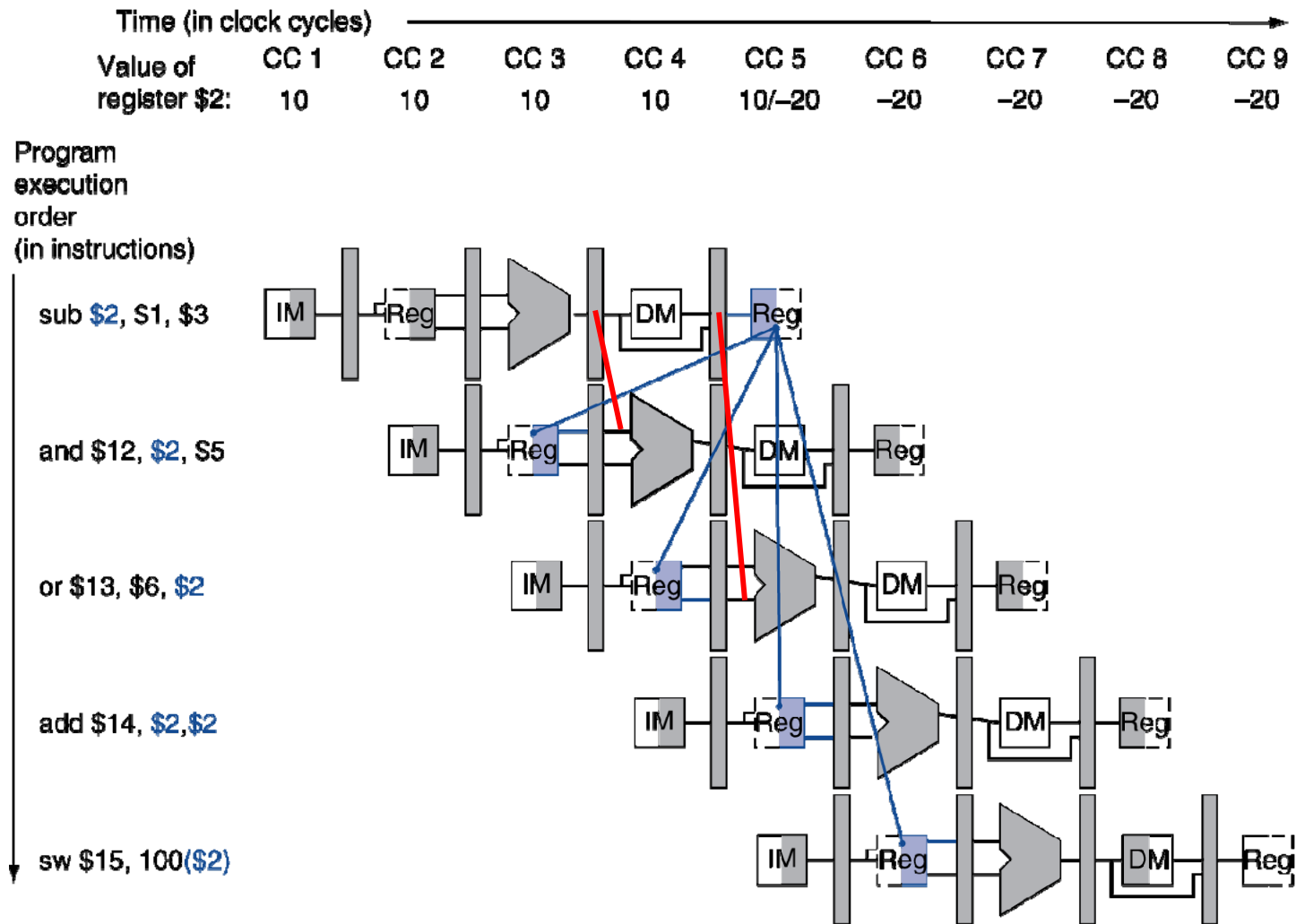
- Consider this sequence:

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

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



# Dependencies & Forwarding



# Detecting the Need to Forward

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

Fwd from  
EX/MEM  
pipeline reg

Fwd from  
MEM/WB  
pipeline reg

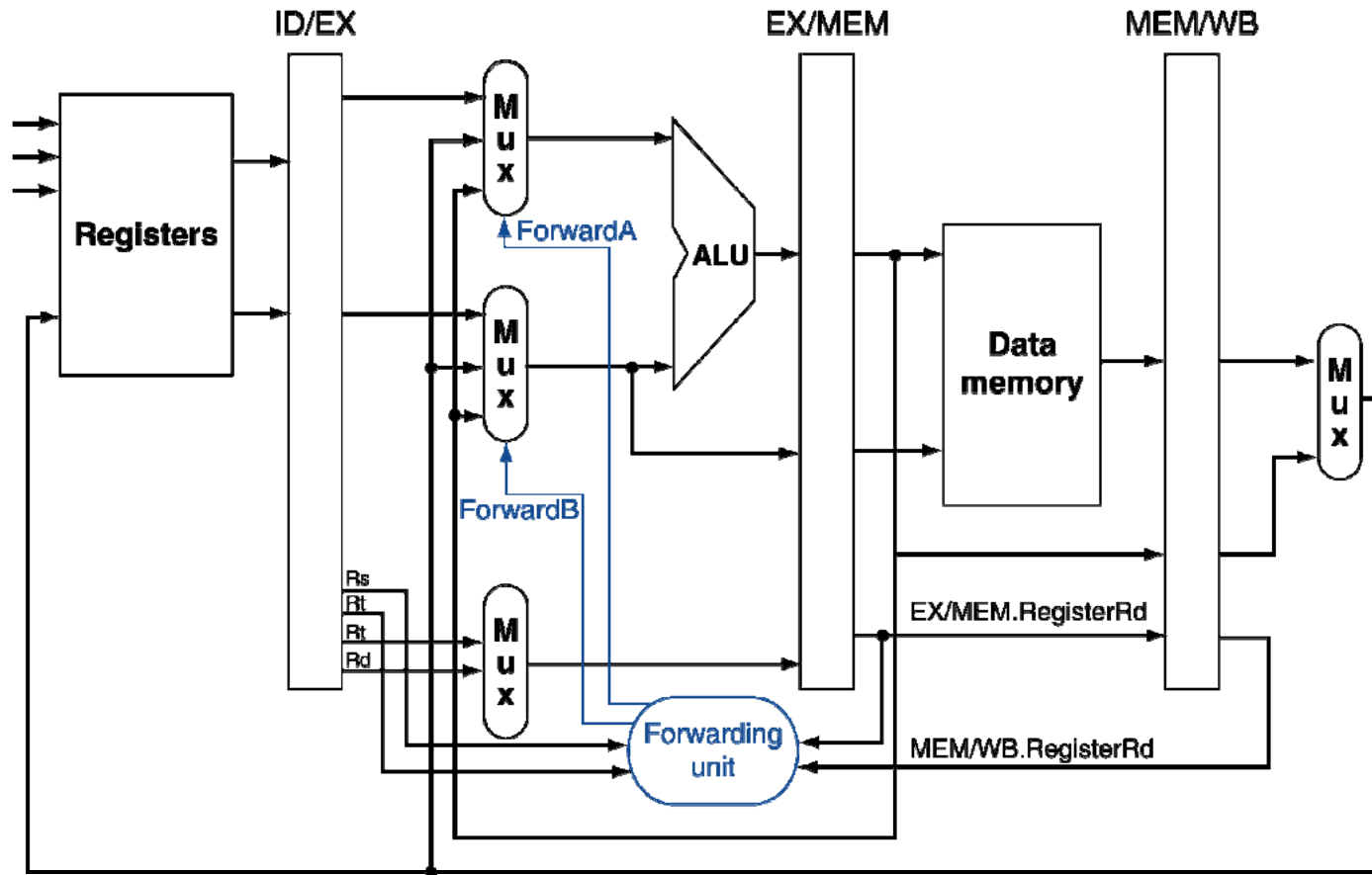


# Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not \$zero
  - EX/MEM.RegisterRd  $\neq$  0,  
MEM/WB.RegisterRd  $\neq$  0



# Forwarding Paths



b. With forwarding



# Forwarding Conditions

## ■ EX hazard

- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))

ForwardA = 10

- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))

ForwardB = 10

## ■ MEM hazard

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd  $\neq$  0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

ForwardA = 01

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd  $\neq$  0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

ForwardB = 01





# Double Data Hazard

- Consider the sequence:
  - add \$1, \$1, \$2
  - add \$1, \$1, \$3
  - add \$1, \$1, \$4
- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

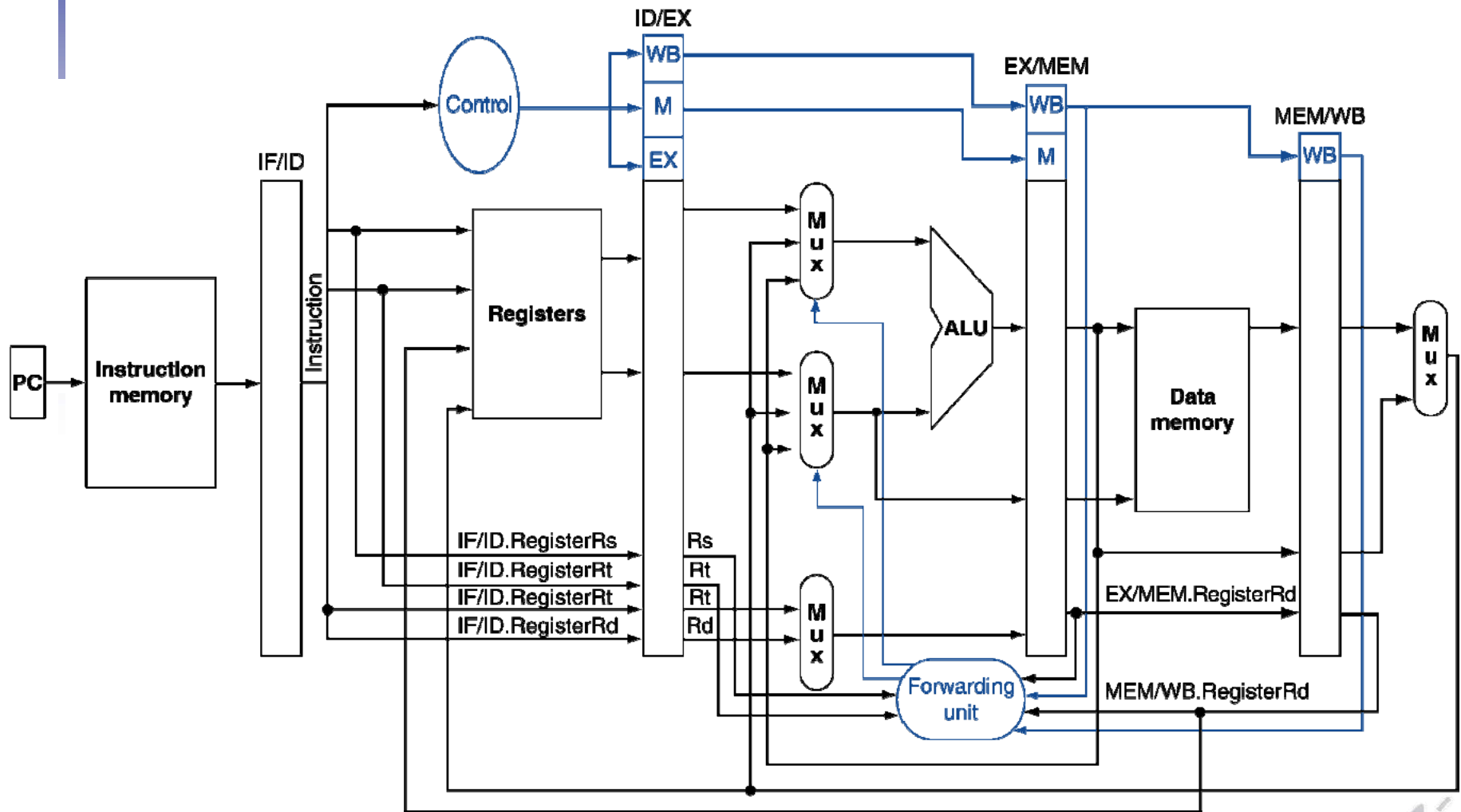


# Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd  $\neq$  0)  
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))  
ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd  $\neq$  0)  
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))  
ForwardB = 01

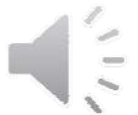


# Datapath with Forwarding



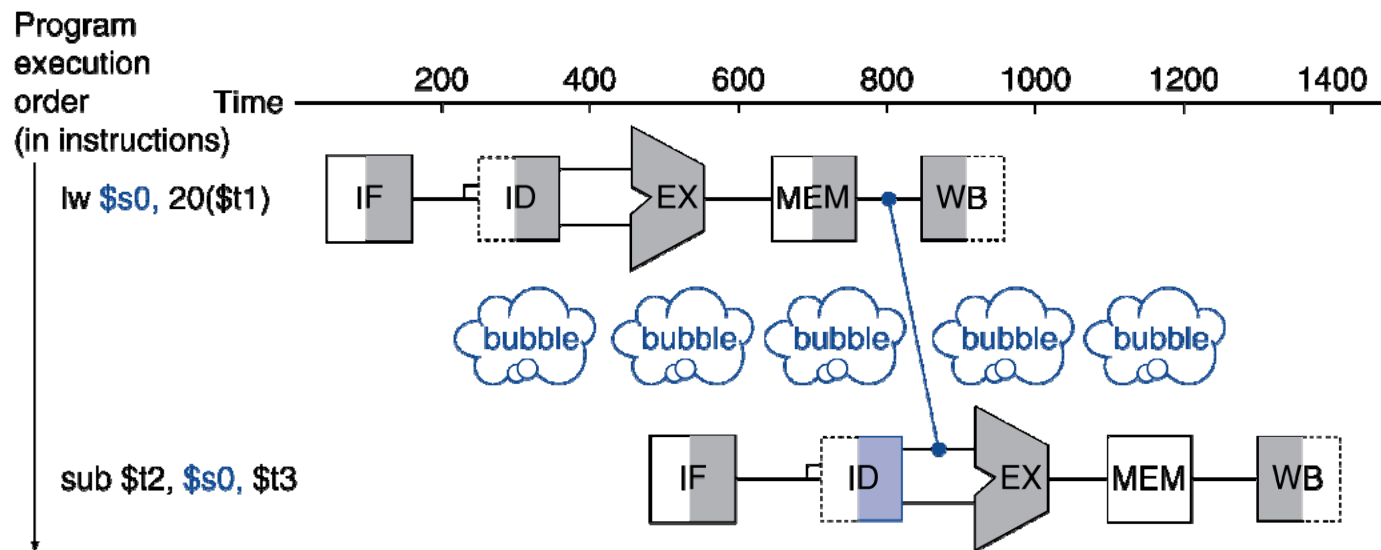
# Chapter 4

## The Processor

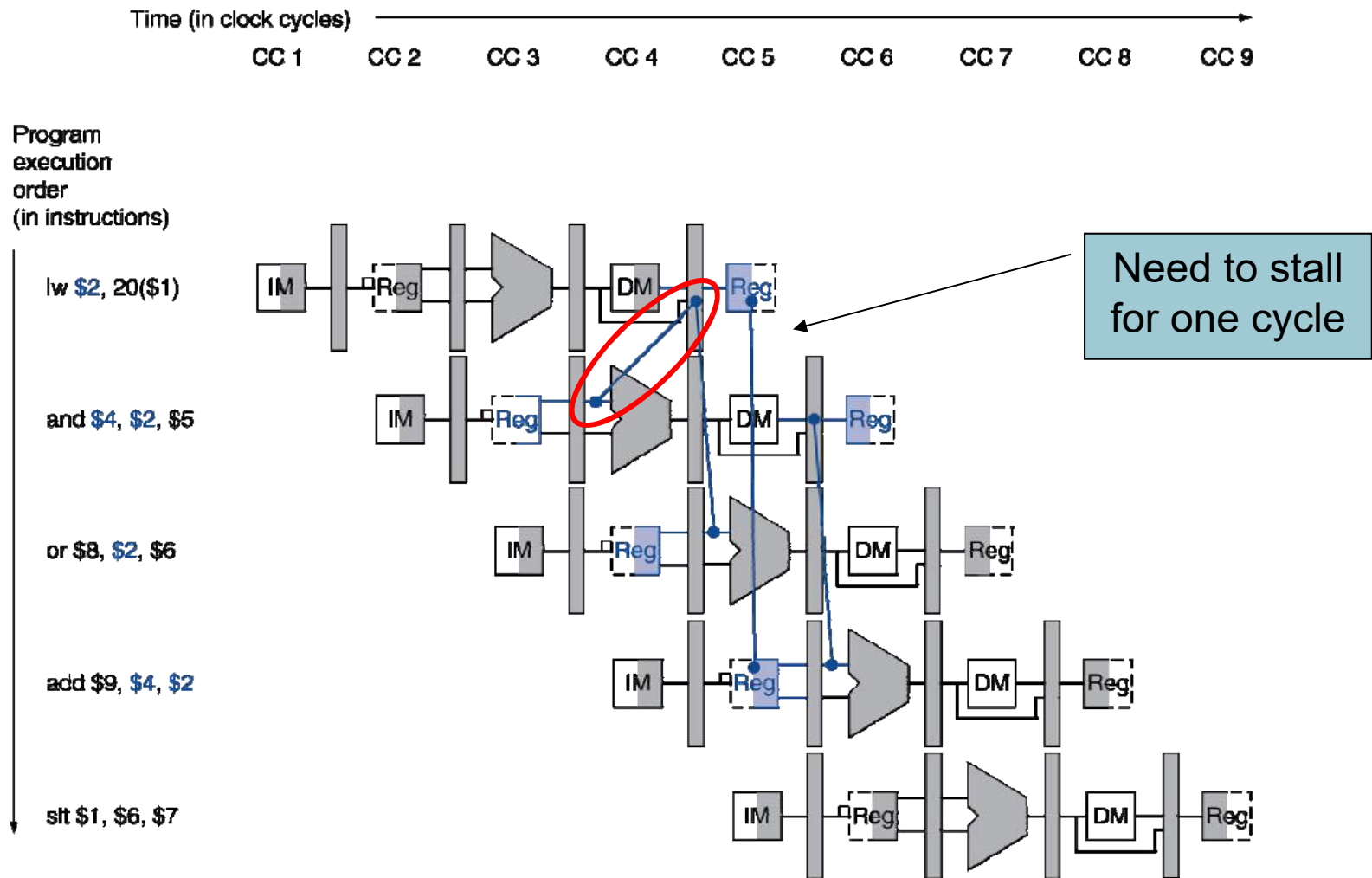


# Load-Use Data Hazard

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



# Load-Use Data Hazard



# Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and  
((ID/EX.RegisterRt = IF/ID.RegisterRs) or  
(ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble



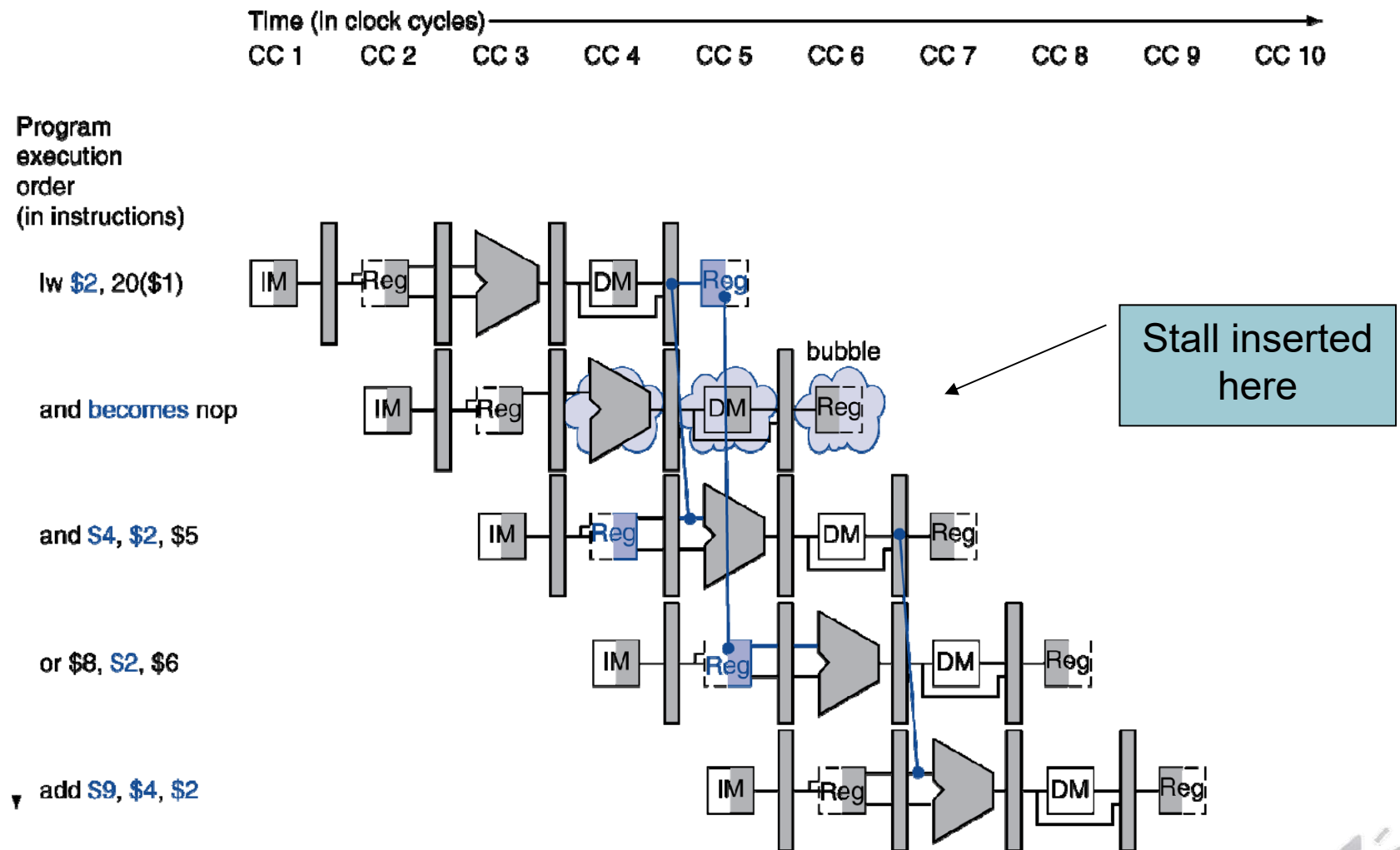
# How to Stall the Pipeline

- 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 I w
    - Can subsequently forward to EX stage

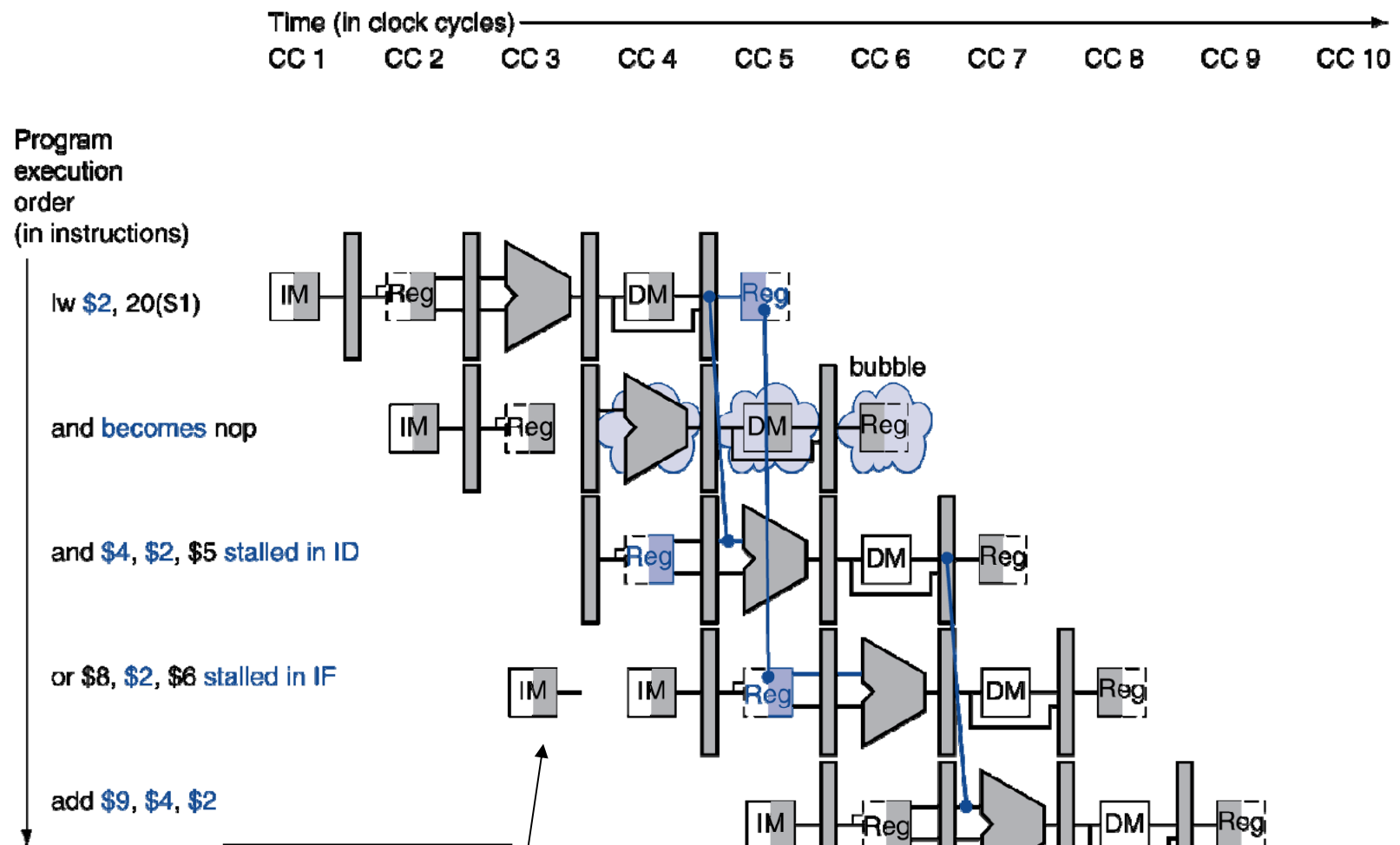




# Stall/Bubble in the Pipeline



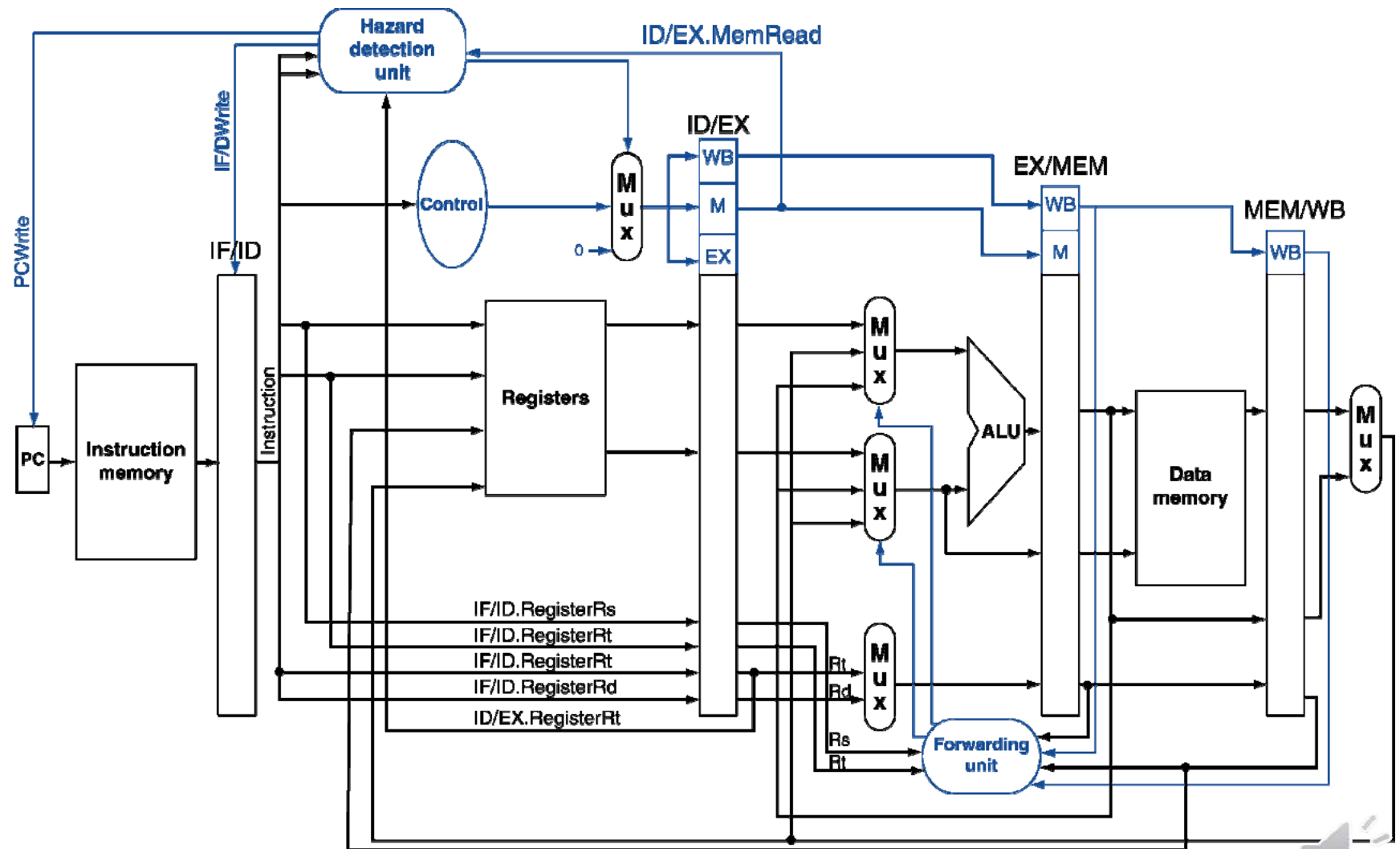
# Stall/Bubble in the Pipeline



Or, more accurately...

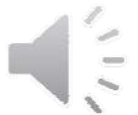


# Datapath with Hazard Detection



# Chapter 4

## The Processor



# Stalls and Performance

## The BIG Picture

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



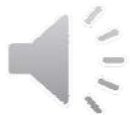
# Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage



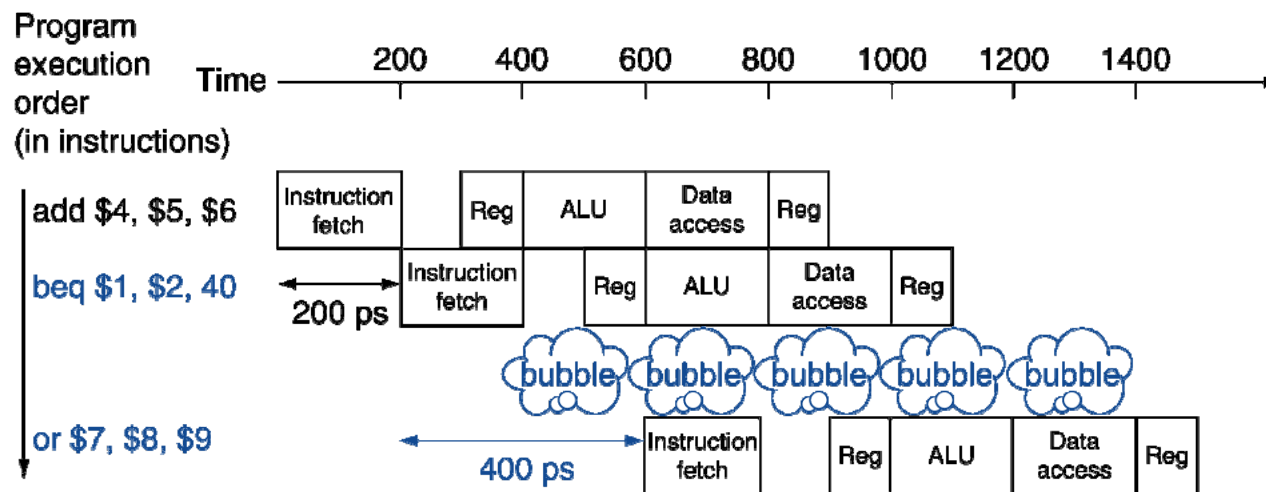
# Dealing With Branch Hazards

- Hardware solutions
  - stall until you know which direction branch goes
  - guess which direction, start executing chosen path (but be prepared to undo any mistakes!)
    - static branch prediction: base guess on instruction type
    - dynamic branch prediction: base guess on execution history
  - reduce the branch delay



# Stall on Branch

- Wait until branch outcome determined before fetching next instruction





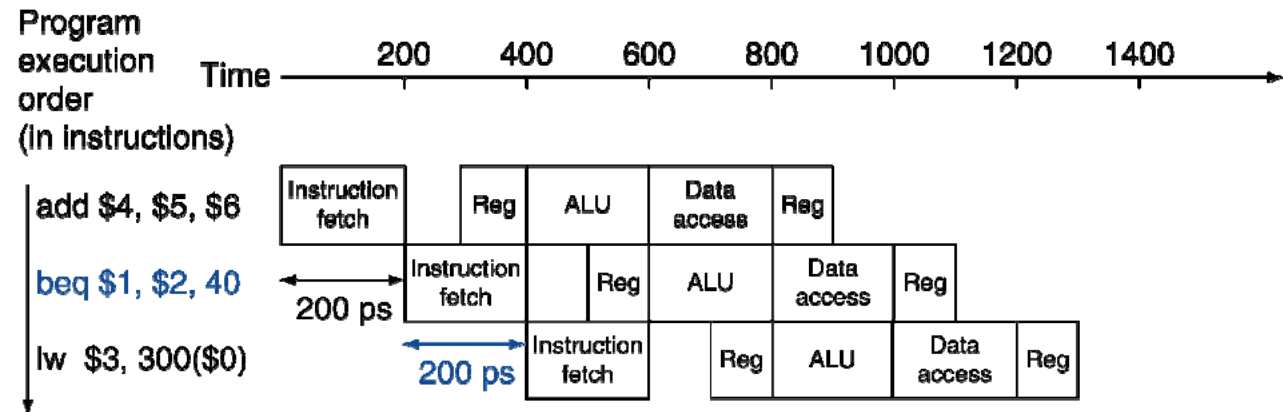
# Branch Prediction

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

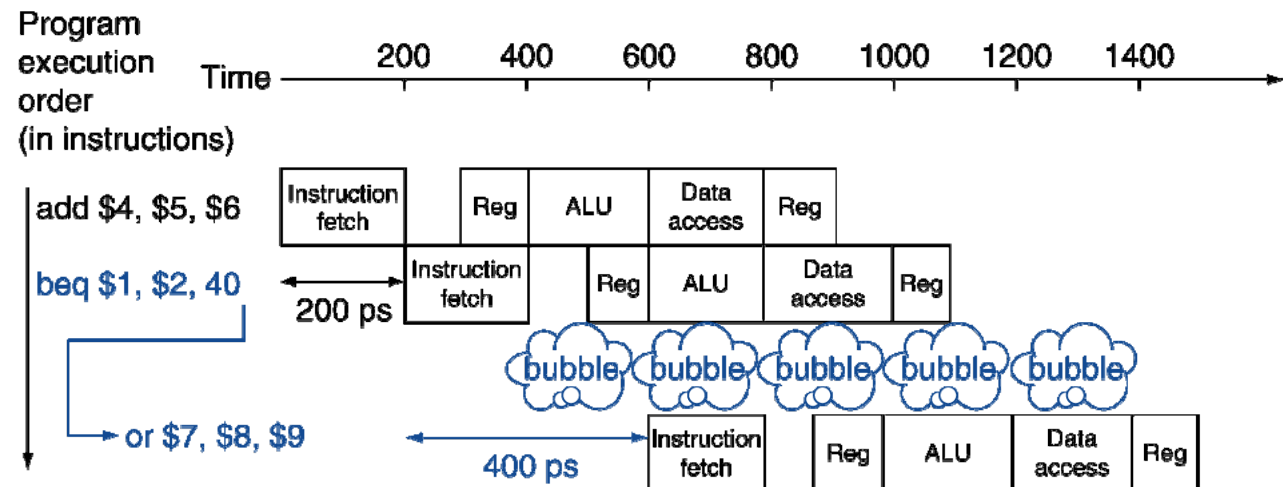


# MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



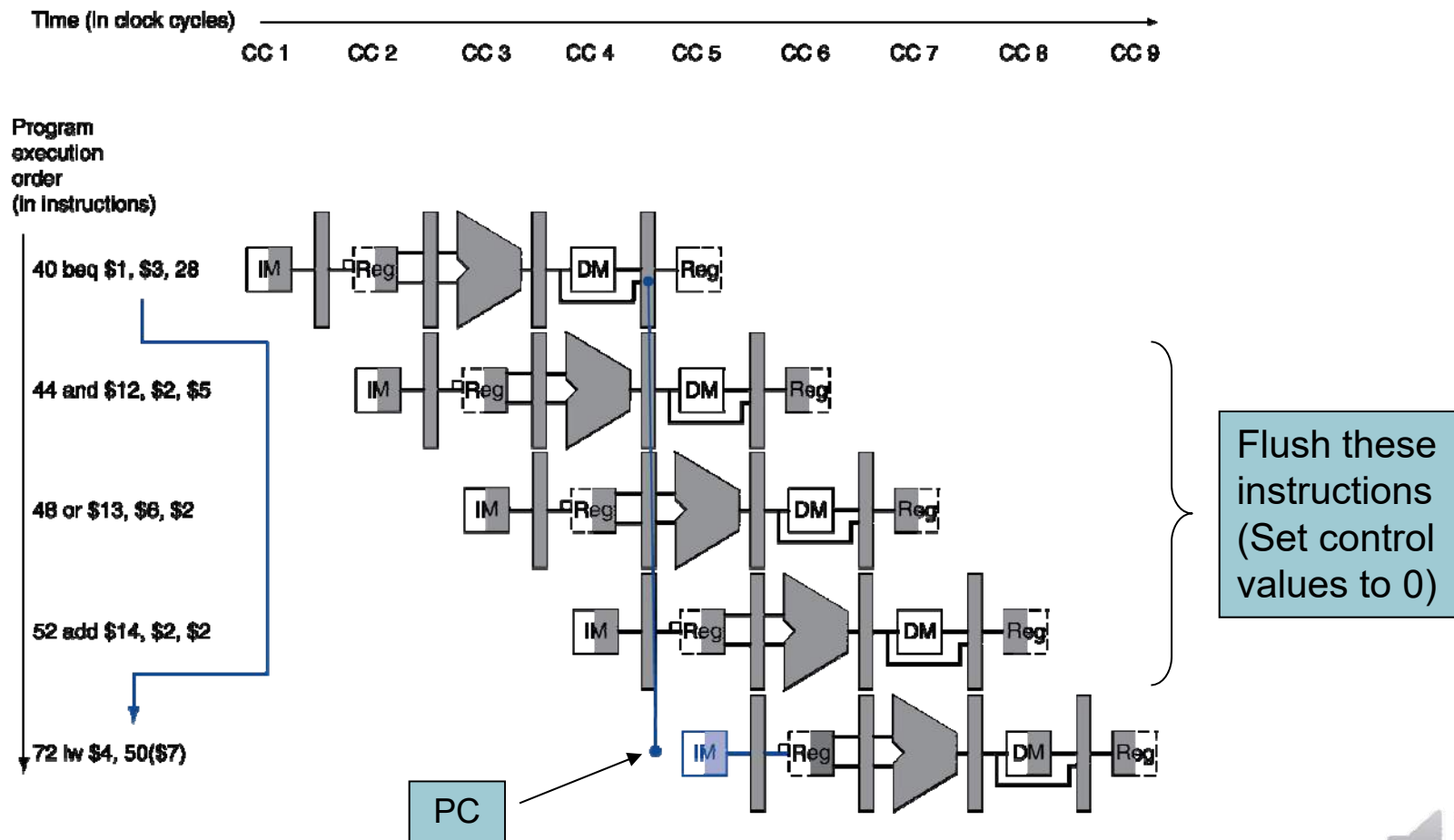
# More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history



# Branch Hazards

- If branch outcome determined in MEM



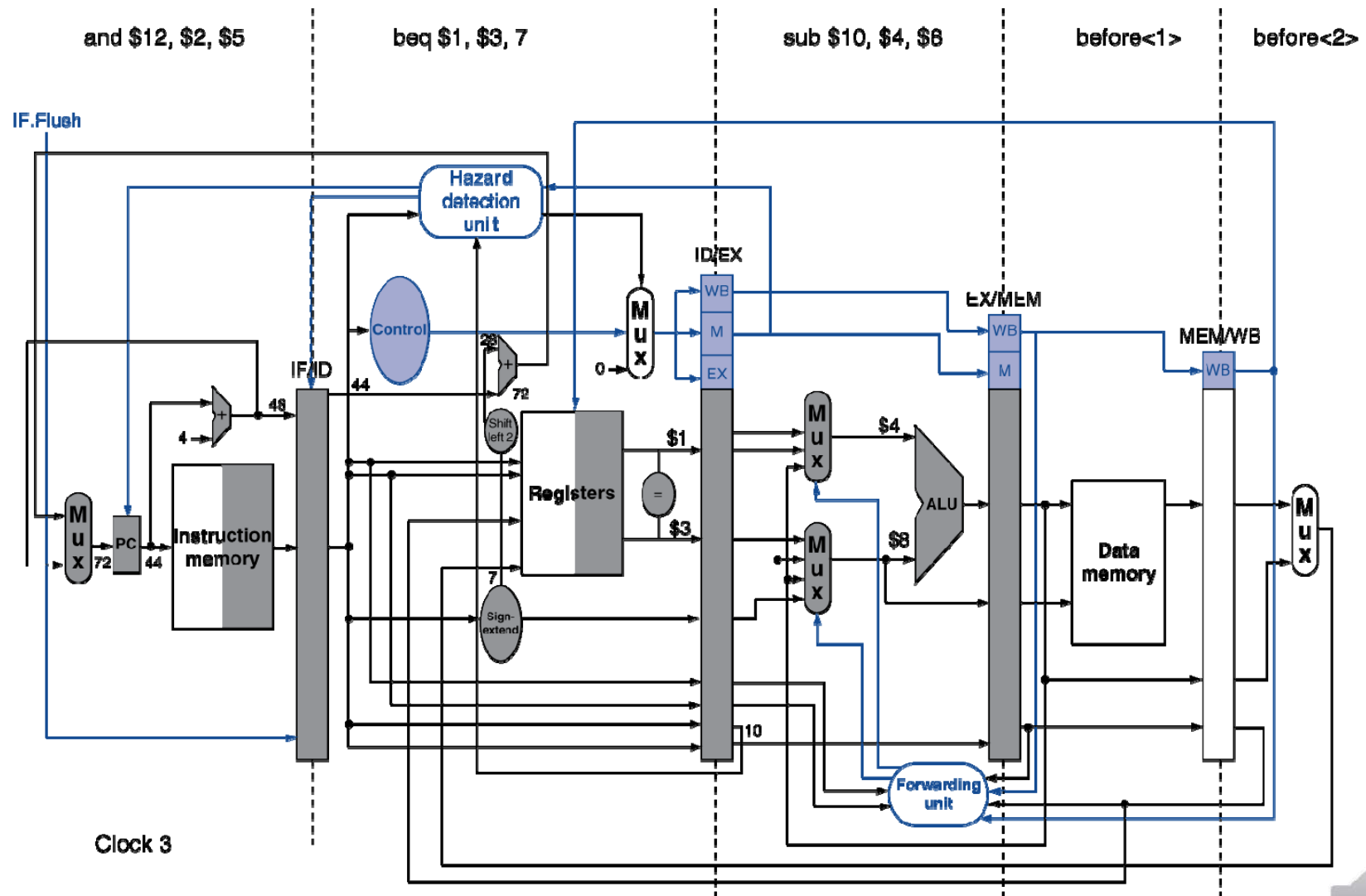
# Reducing Branch Delay

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

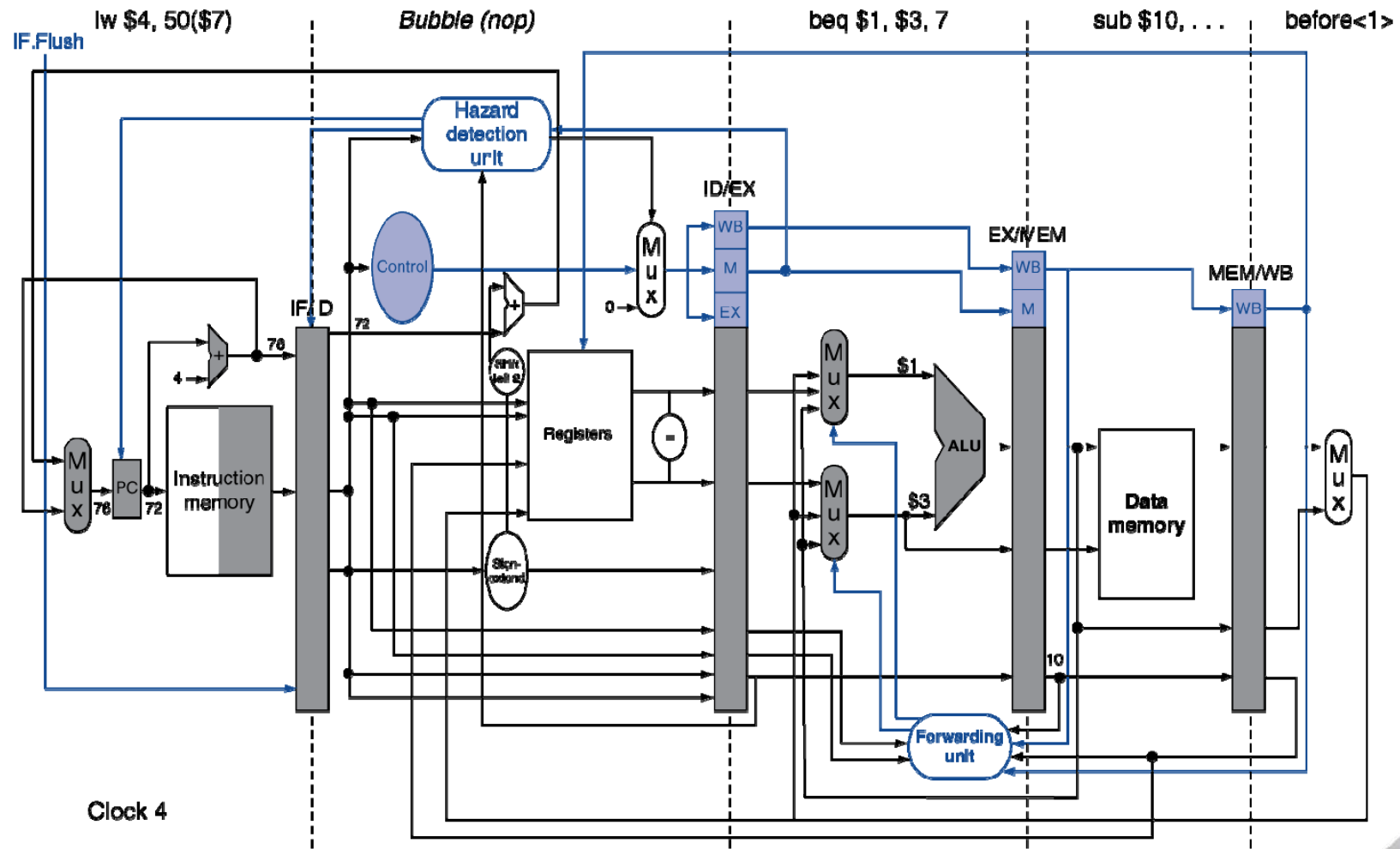
```
36:  sub  $10, $4, $8
40:  beq  $1,  $3,  7
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
    ...
72:  lw   $4,  50($7)
```



# Example: Branch Taken



# Example: Branch Taken

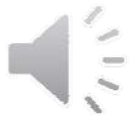


Clock 4



# Chapter 4

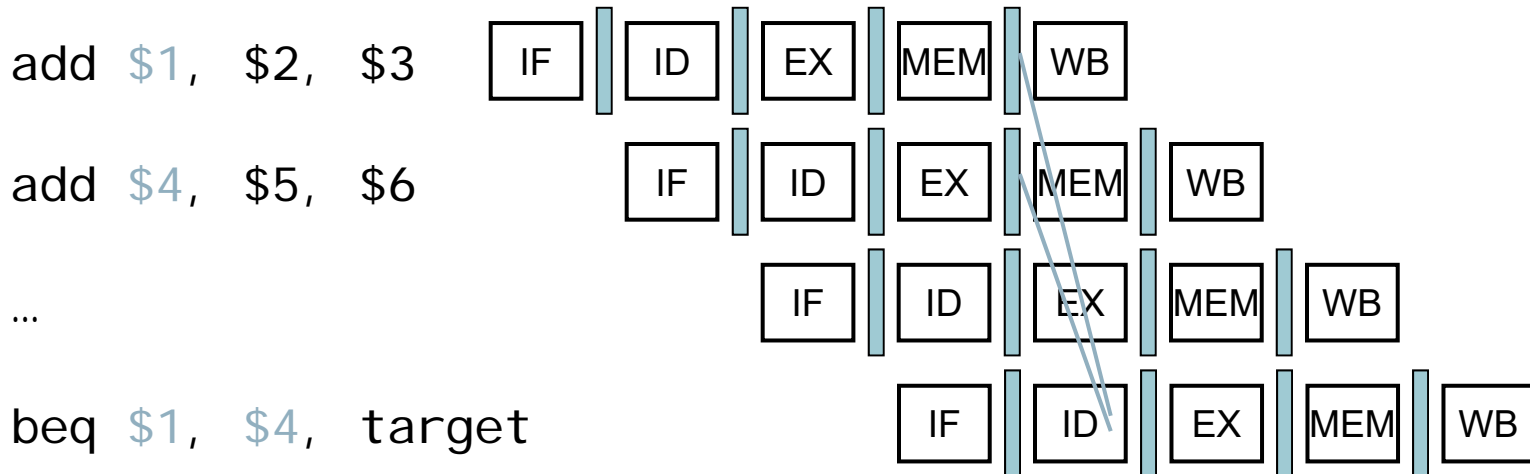
## The Processor





# Data Hazards for Branches

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

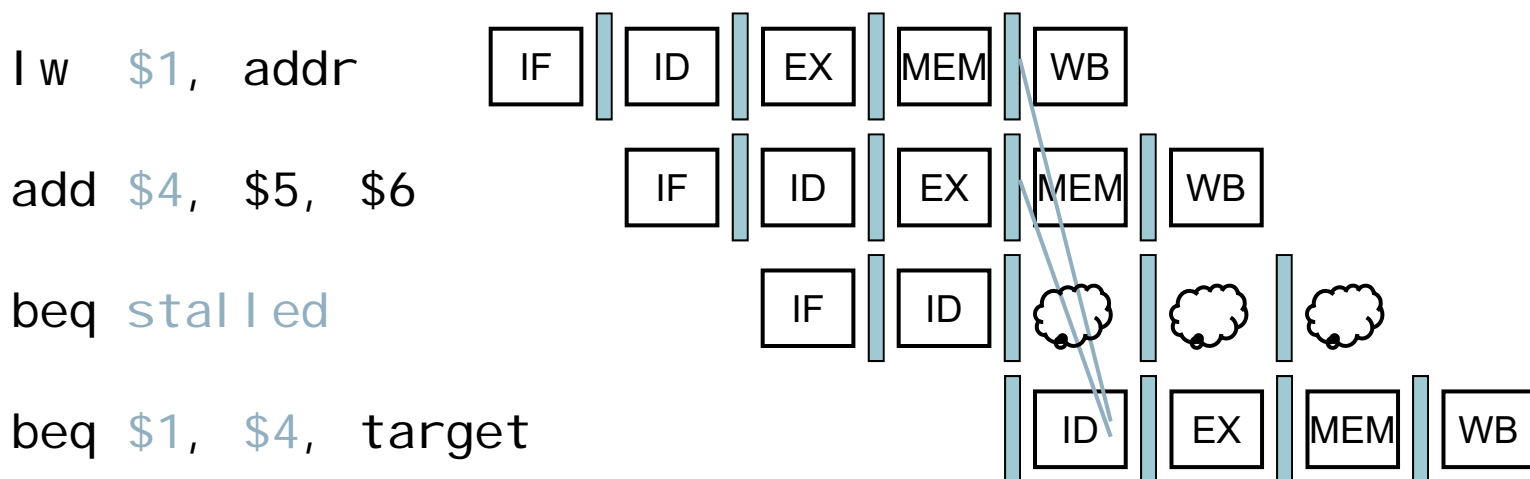


- Can resolve using forwarding



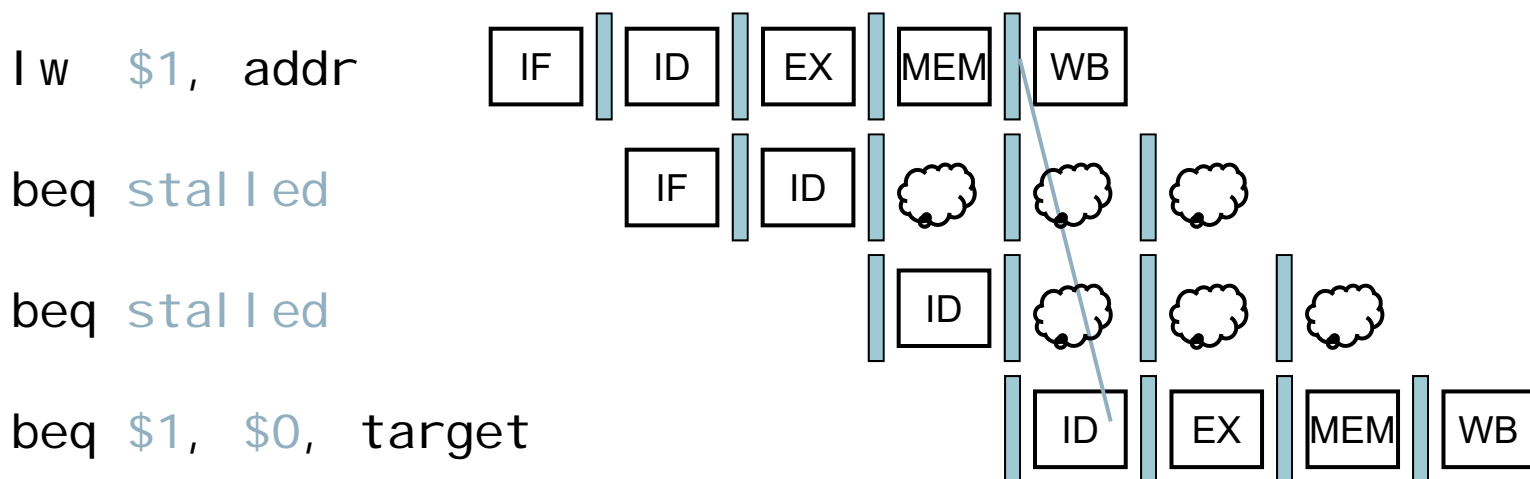
# Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2<sup>nd</sup> preceding load instruction
  - Need 1 stall cycle



# Data Hazards for Branches

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



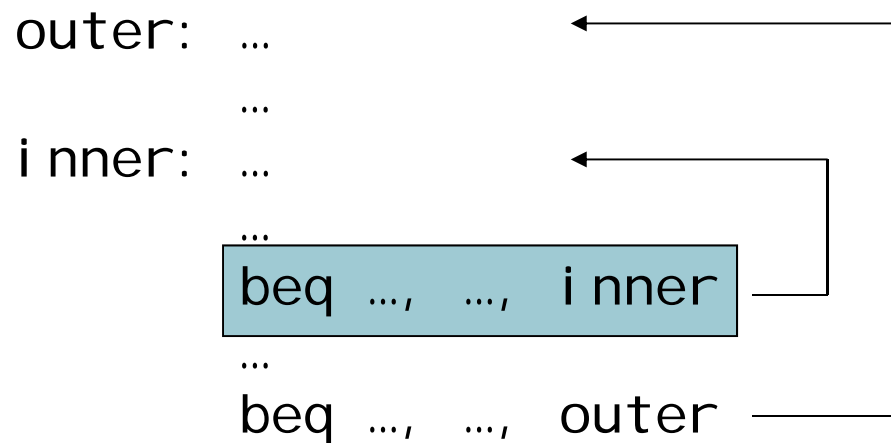
# Dynamic Branch Prediction

- 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!

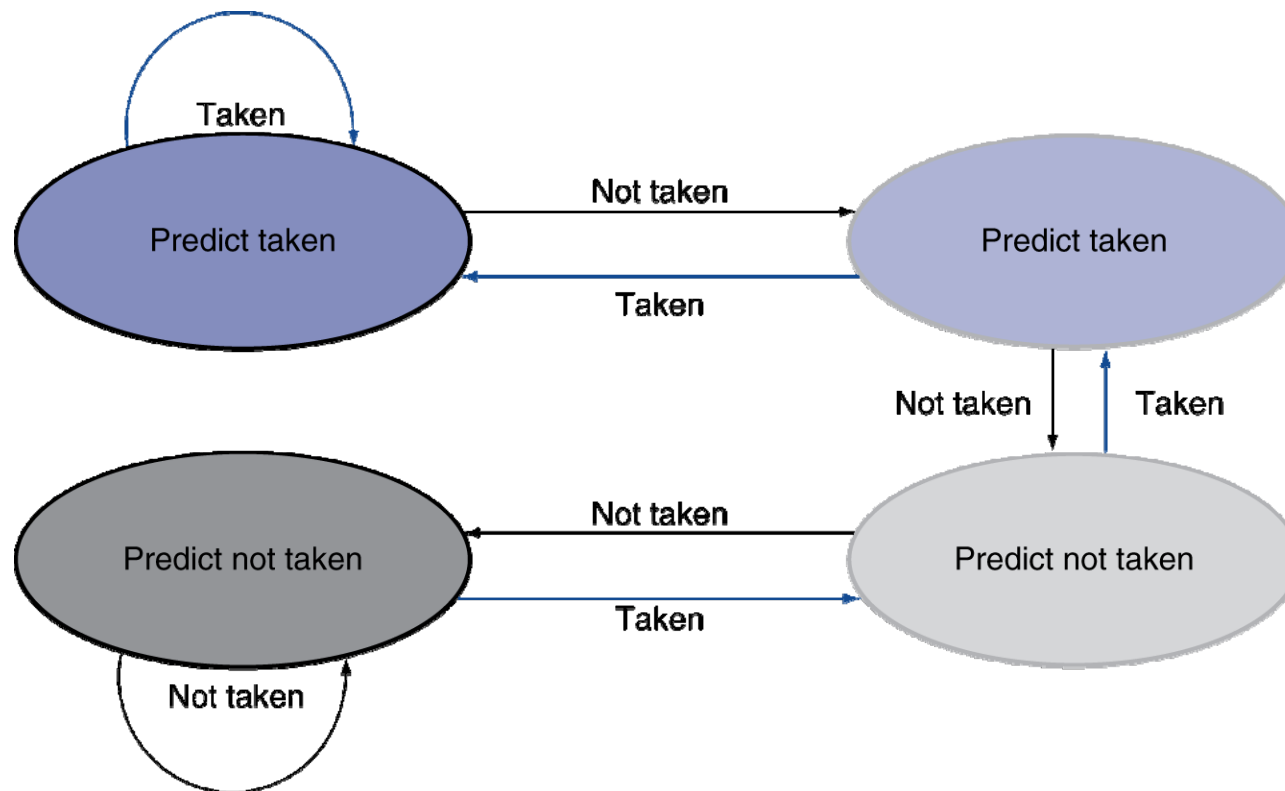


- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around



# 2-Bit Predictor

- Only change prediction on two successive mispredictions



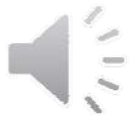
# Calculating the Branch Target

- 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



# Chapter 4

## The Processor





# Exceptions and Interrupts

- “Unexpected” events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, ...
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard



# Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180



# An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - ....: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler



# Handler Actions

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

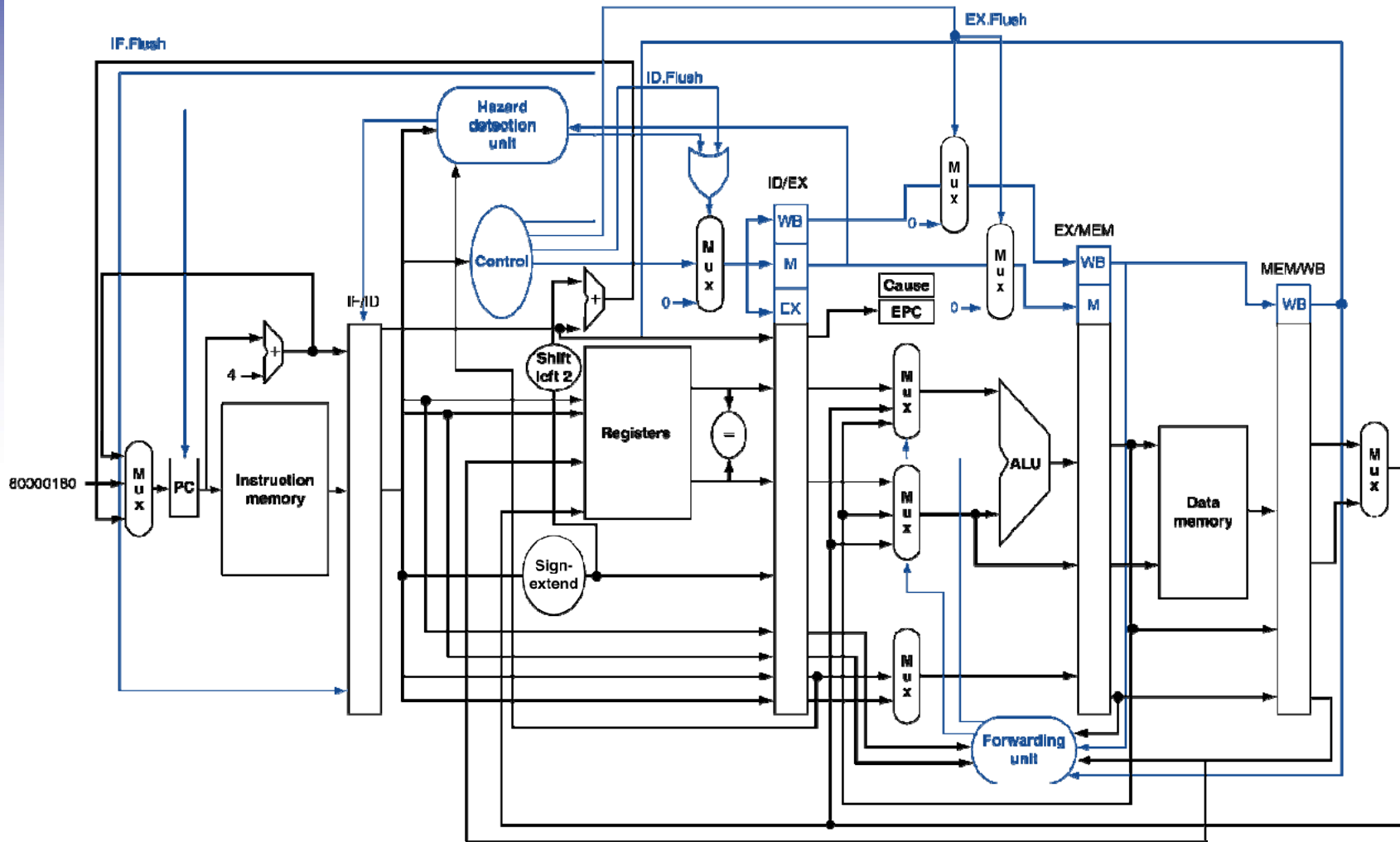


# Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage  
add \$1, \$2, \$1
  - Prevent \$1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware



# Pipeline with Exceptions



# Exception Properties

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



# Exception Example

- Exception on `add` in

40	sub	\$11,	\$2,	\$4
44	and	\$12,	\$2,	\$5
48	or	\$13,	\$2,	\$6
4C	add	\$1,	\$2,	\$1
50	sl t	\$15,	\$6,	\$7
54	lw	\$16,	50(\$7)	

...

- Handler

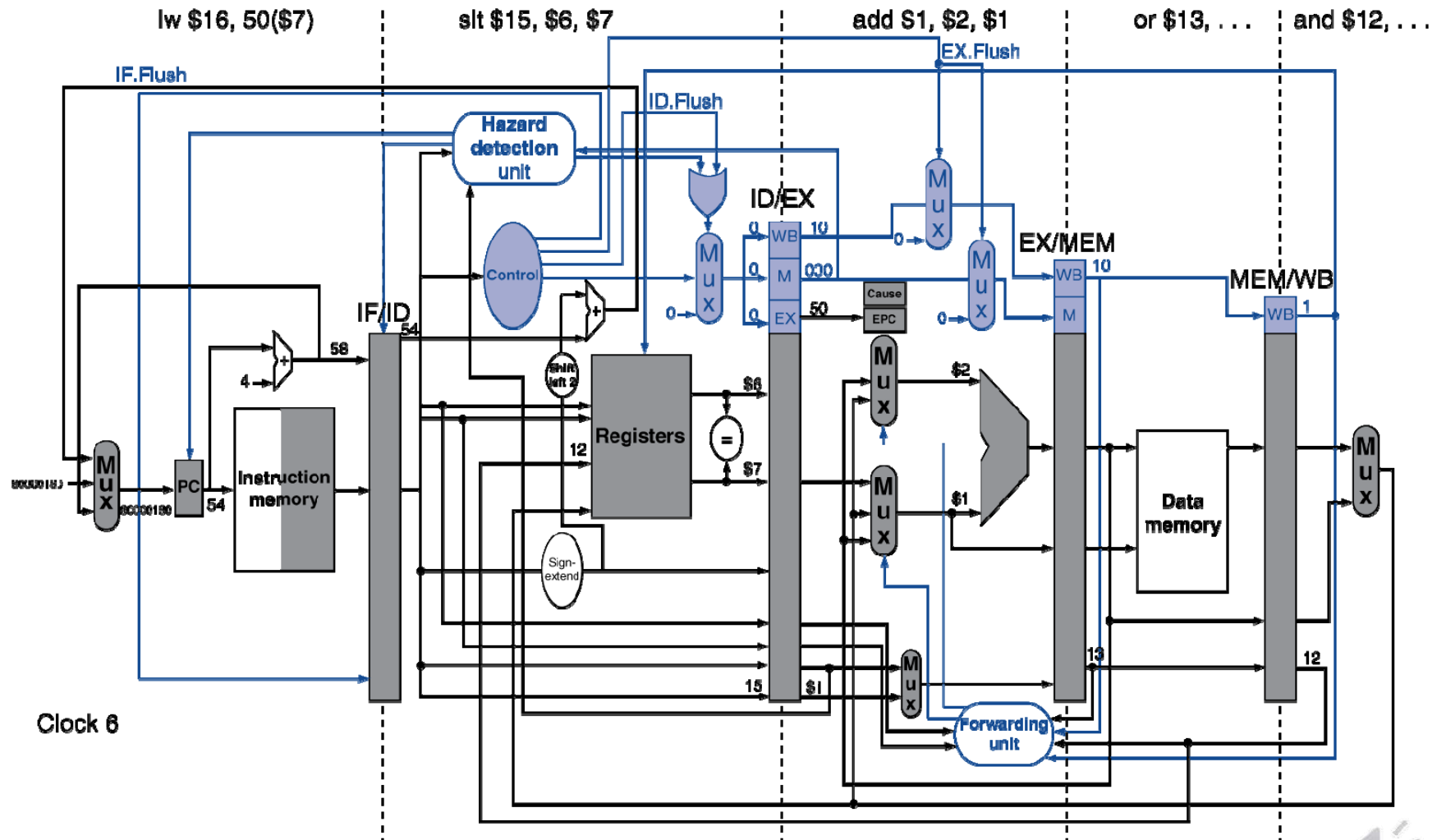
80000180	sw	\$25,	1000(\$0)
80000184	sw	\$26,	1004(\$0)

...

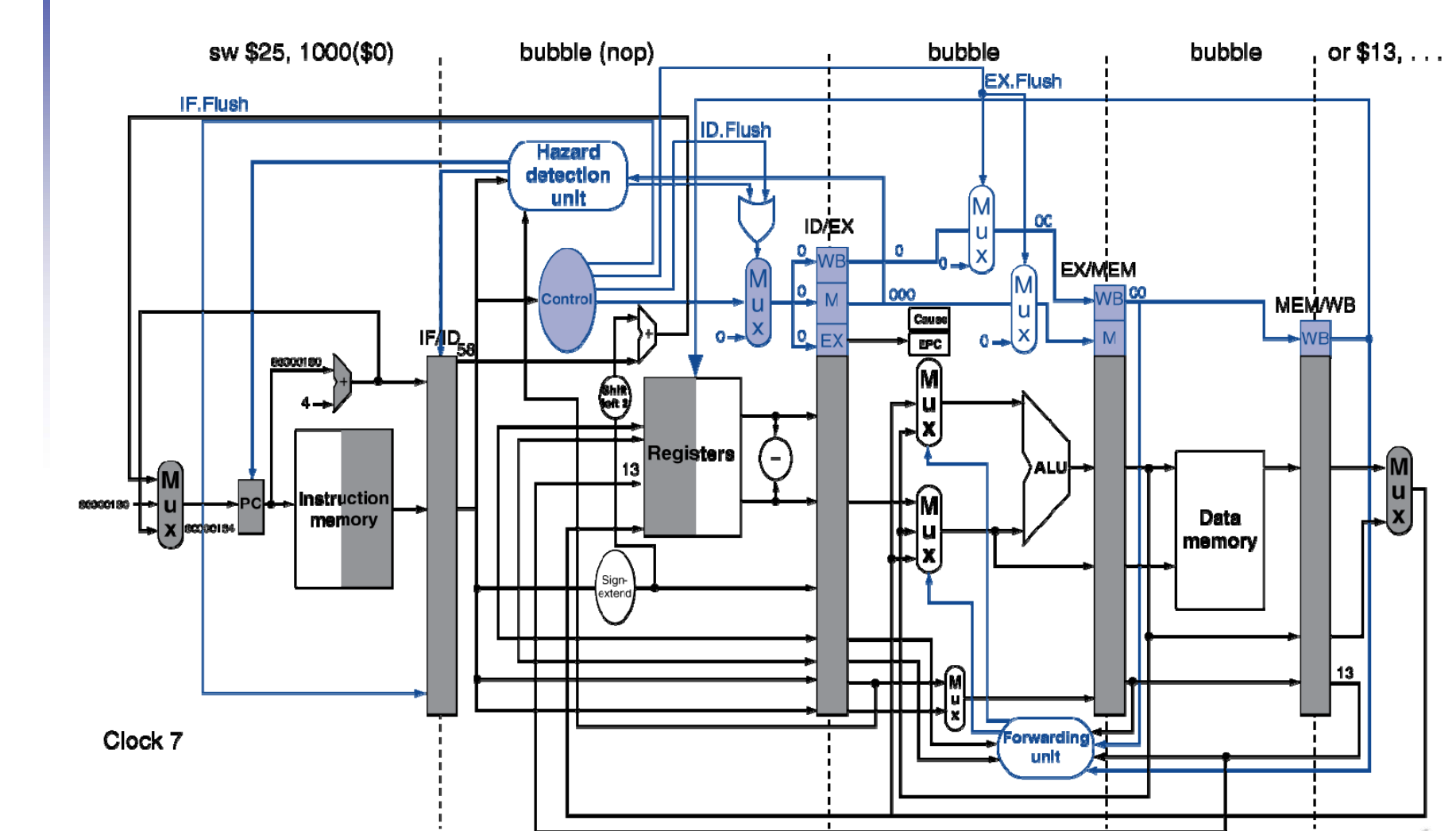




# Exception Example



Age Group	Percentage
18-24	35%
25-34	25%
35-44	15%
45-54	10%
55-64	5%
65-74	3%
75-84	2%
85+	1%



# 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

