MIPS Pipeline

Hazards

Hazards

- There are situations in pipelining when the next instruction cannot execute in the following clock cycle.
- *Three different* types
  - **Structural hazards:** HW cannot support this combination of instructions
  - **Data hazards:** Instruction depends on result of prior instruction still in the pipeline
  - **Control hazards:** Pipelining of branches & other instructions that change the PC
  - Common solution is to **stall** the pipeline until the hazard is resolved, inserting one or more "**bubbles**" in the pipeline

Hazards

- Structure hazards
  - A required resource is busy
  - MIPS instruction and data memory helps pipelining

I1	IF	ID	EX	MEM	WB			
I2		IF	ID	EX	MEM	WB		
I3			IF	ID	EX	MEM	WB	
I4				IF	ID	EX	MEM	WB

  - When I1 reads memory, I4 reads instruction
  - Separate Instruction and data memory will help I1 and I4 to continue in parallel

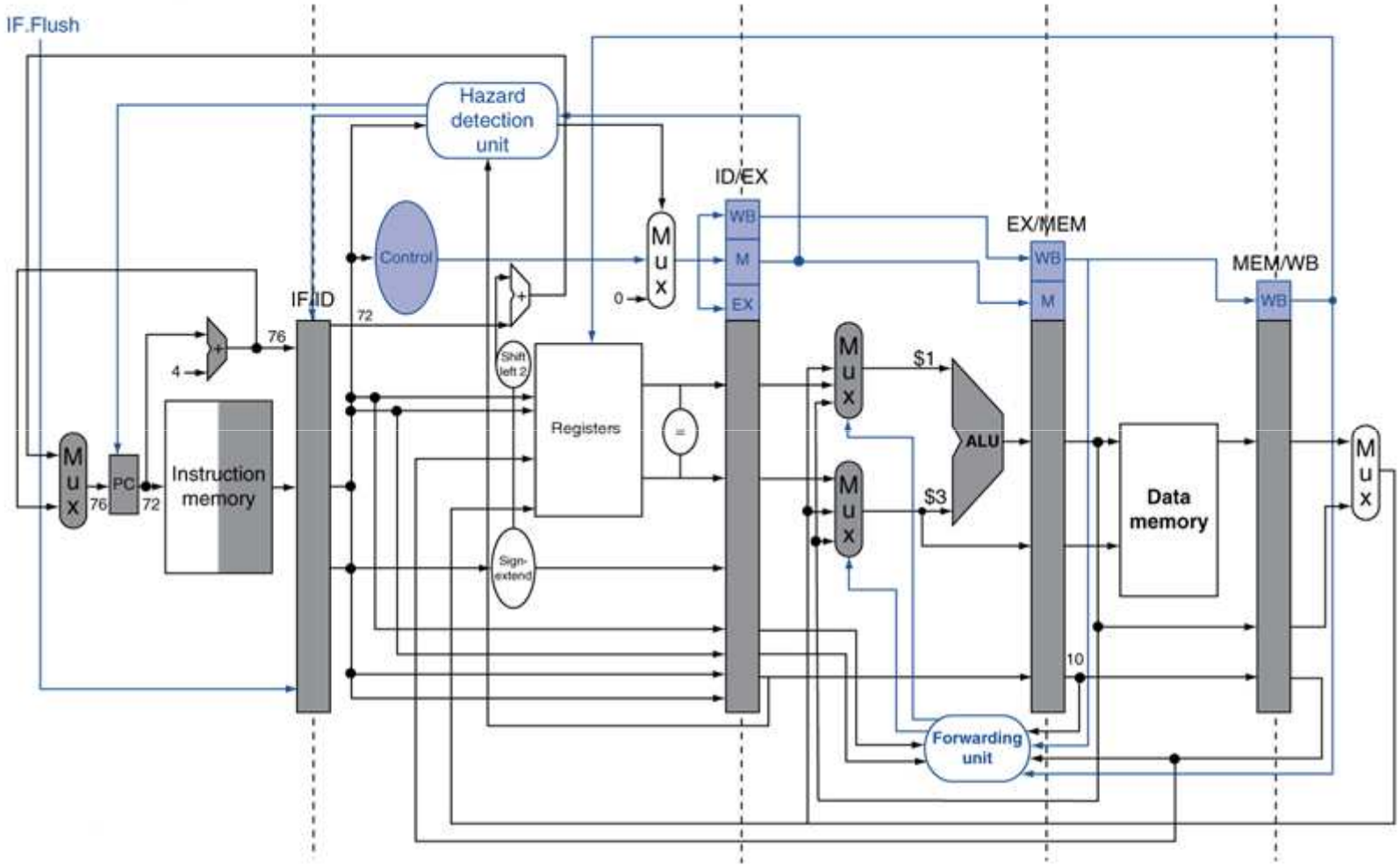
Data Hazard

Data hazard

- One instruction waiting for the result of previous instruction
- Solution
  - Stall
    - Need to wait for previous instruction to complete its data read/write
  - Forward

- Instruction type : Hazard solutions
  - R-type - arithemetic : Forwarding Unit
  - Lw – data transfer : Hazard Detection Unit

Data Hazard

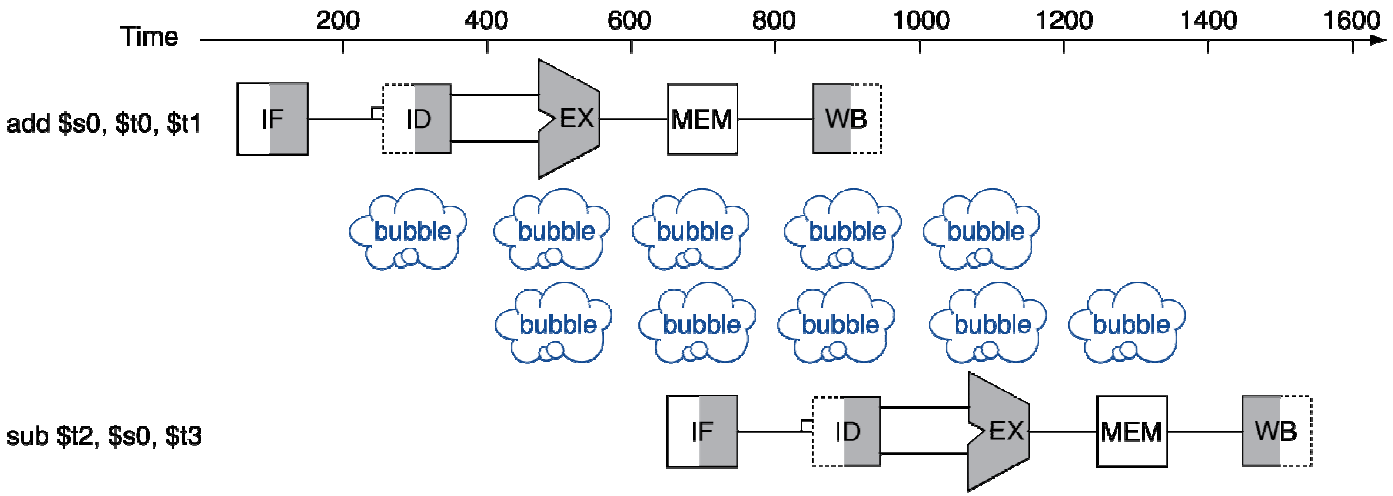

IF.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
WB
M
EX
IF/ID
72
76
4
0
Mux
Shift left 2
Registers
=
Sign-extend
PC
Instruction memory
$1
$3
ALU
Data memory
10
Forwarding unit

Stall

- Need to wait for previous instruction to complete its data read/write
- Source reg of 2nd instruction is destination reg of 1st instruction

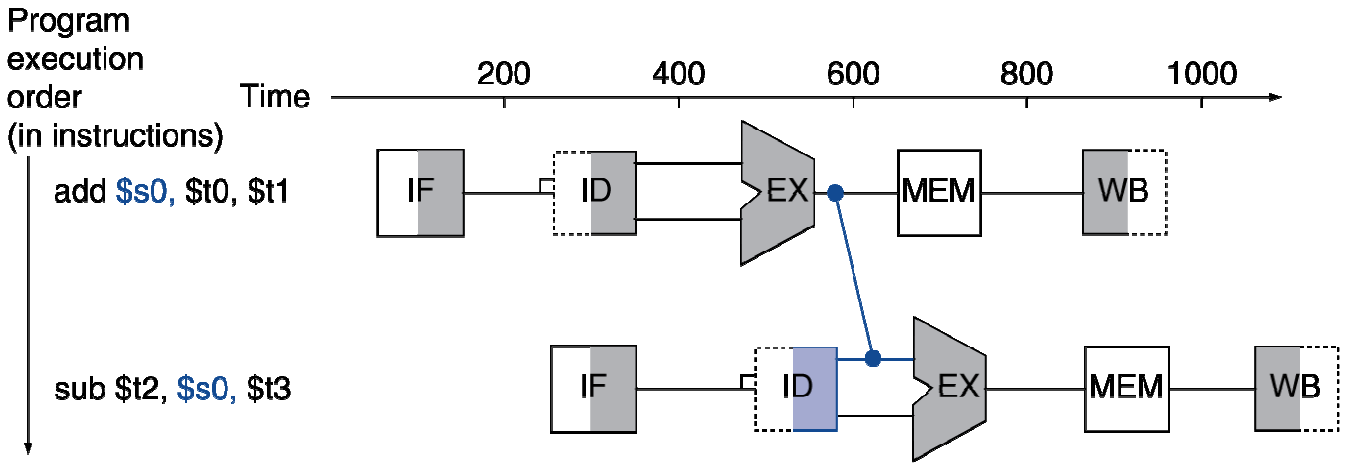
- add $s0, $t0, $t1
- sub $t2, $s0, $t3

add	IF	ID	EX	MEM	WB			
Sub		IF	ID	ID	ID	EX	MEM	WB
			IF	IF	IF	ID	EX	MEM
						IF	ID	EX



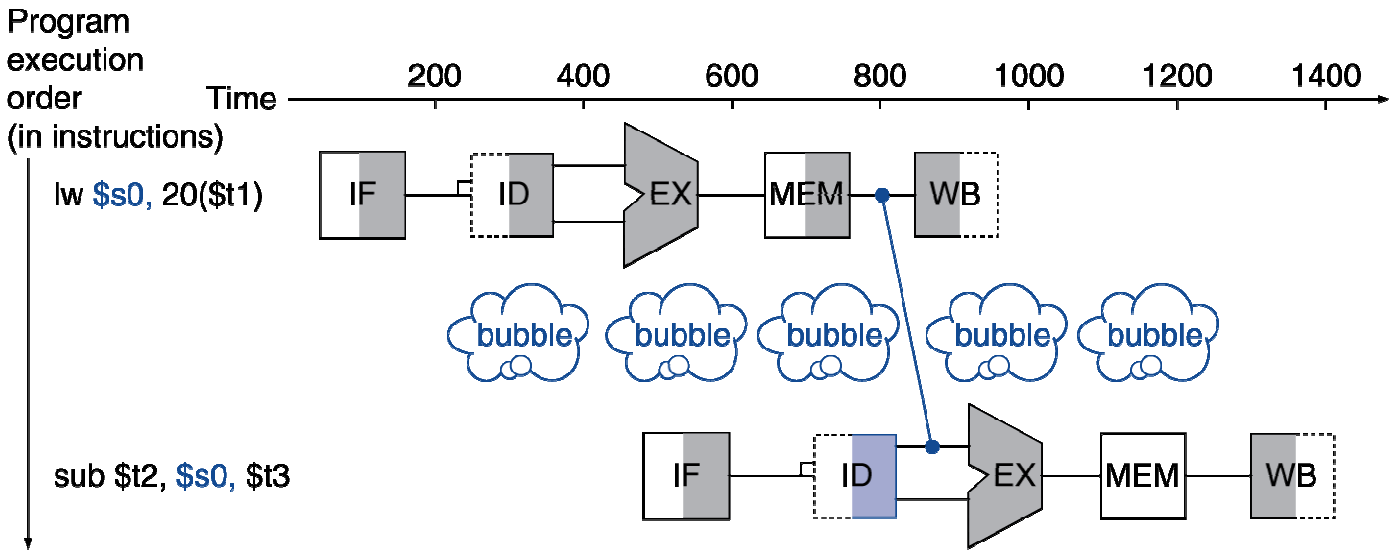
Forwarding

- Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding or bypassing**
- As soon as the ALU creates the sum for the add, we can supply it as an input for the subtract
- 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
- Data from lw instruction is available only after MEM
- So there is a stall



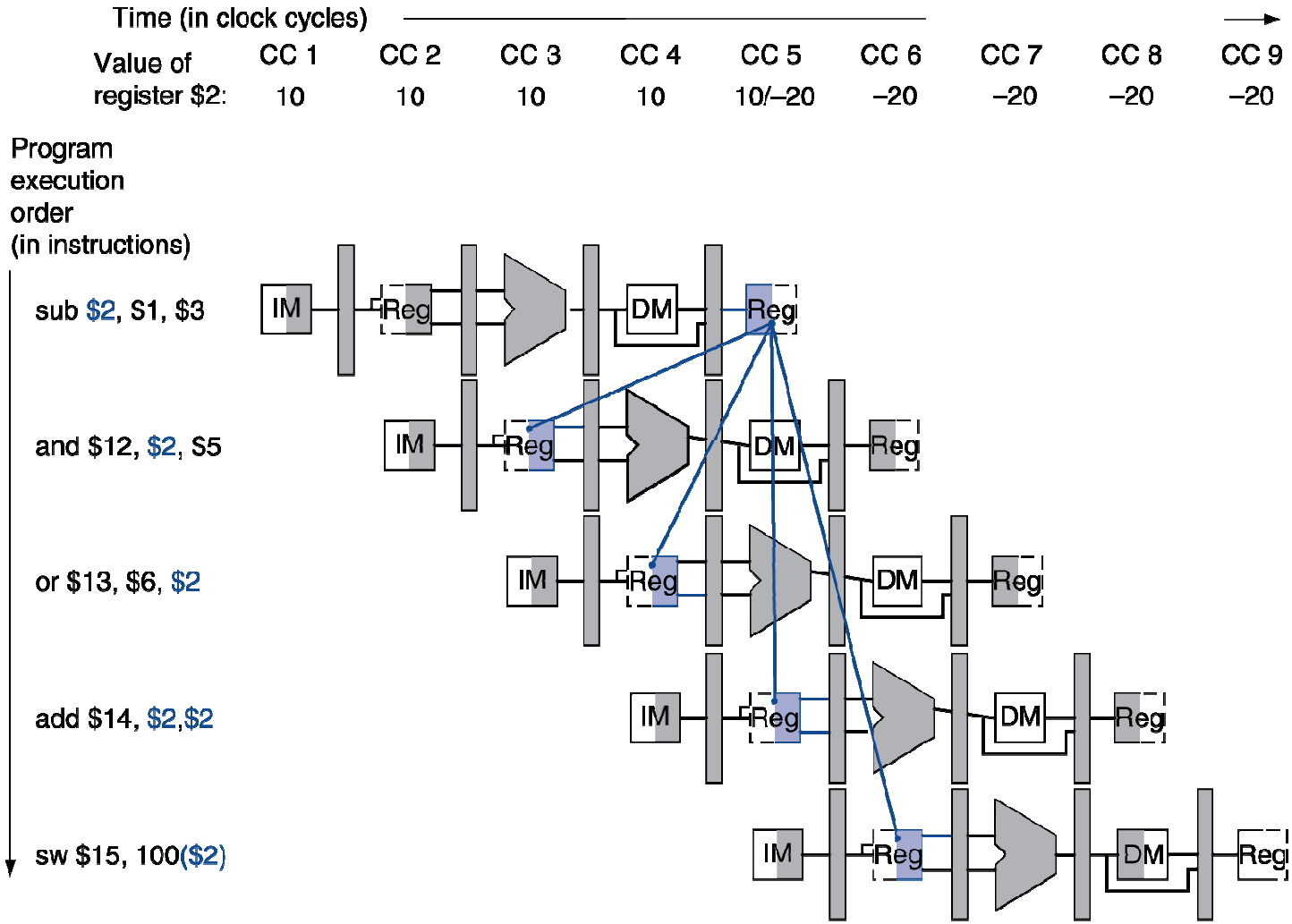
I: Data Hazards in ALU Instructions

- Consider this sequence:

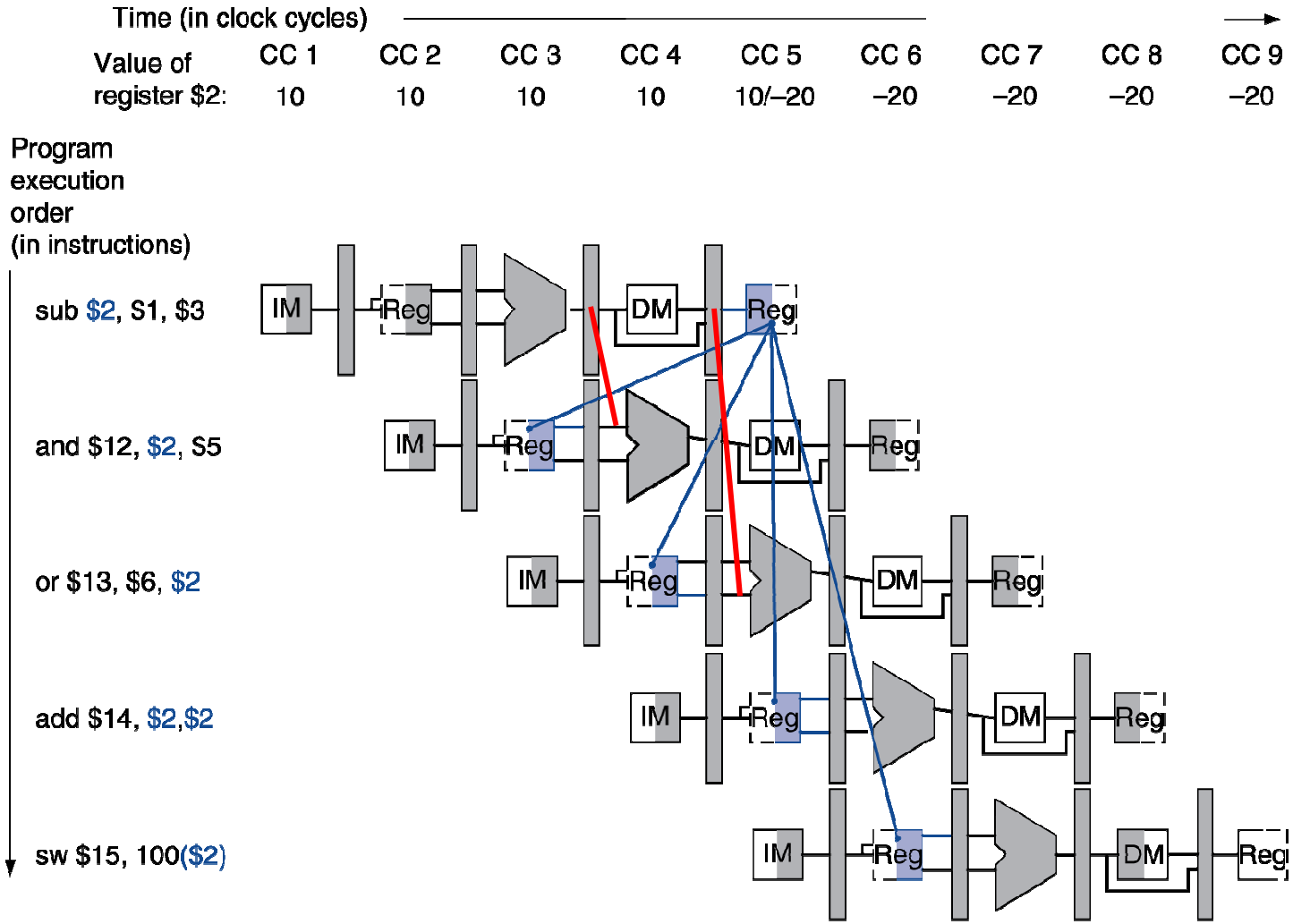
```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

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

Dependencies



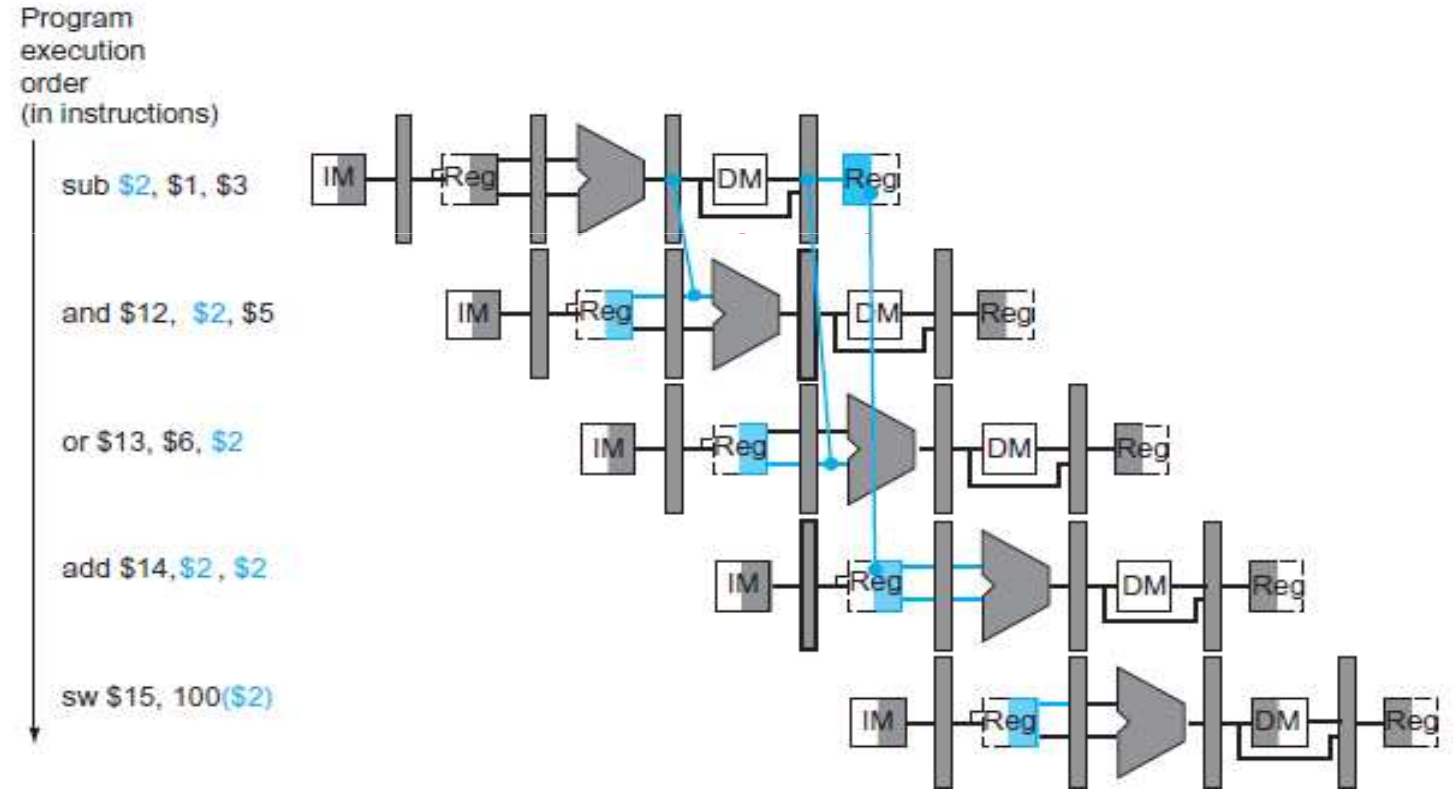
Dependencies & Need for Forwarding



Dependencies & Forwarding

Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2:	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X



Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10

Forwarding Conditions 1

- Data hazards when

  1a. If(EX/MEM.RegisterRd = ID/EX.RegisterRs)

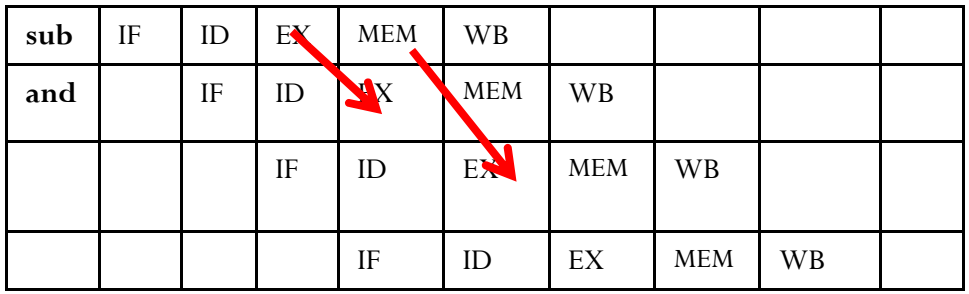
  1b.if(EX/MEM.RegisterRd = ID/EX.RegisterRt)

  Fwd from EX/MEM pipeline reg

  2a. If(MEM/WB.RegisterRd = ID/EX.RegisterRs)

  2b. If(MEM/WB.RegisterRd = ID/EX.RegisterRt)

  Fwd from MEM/WB pipeline reg



Forwarding Conditions 1

- Data hazards when

  1a. If(EX/MEM.RegisterRd = ID/EX.RegisterRs)

  1b. if(EX/MEM.RegisterRd = ID/EX.RegisterRt)

  2a. If(MEM/WB.RegisterRd = ID/EX.RegisterRs)

  2b. If(MEM/WB.RegisterRd = ID/EX.RegisterRt)

Example
•1a hazard $2
sub $2, $1,$3 EX/MEM.RegisterRd
and $12,$2,$5 ID/EX.RegisterRs

Example
•1b hazard $2
sub $2, $1,$3 EX/MEM.RegisterRd
and $12,$5,$2 ID/EX.RegisterRt

Example
•2a hazard $2
sub $2, $1,$3 MEM/WB.RegisterRd
and $12, $2, $5
or $13, $2, $6 ID/EX.RegisterRs

Example
•2b hazard $2
sub $2, $1,$3 MEM/WB.RegisterRd
and $12, $2, $5
or $13, $6, $2 ID/EX.RegisterRt

Forwarding Conditions 2

- Some instructions do not write registers (sw,beq)
- Need to forward only when the previous instrution write the register (RegWrite)
- Forward only when RegWrite control signal is active (R type, lw)
  - If (EX/MEM.RegWrite)
  - If(MEM/WB.RegWrite)

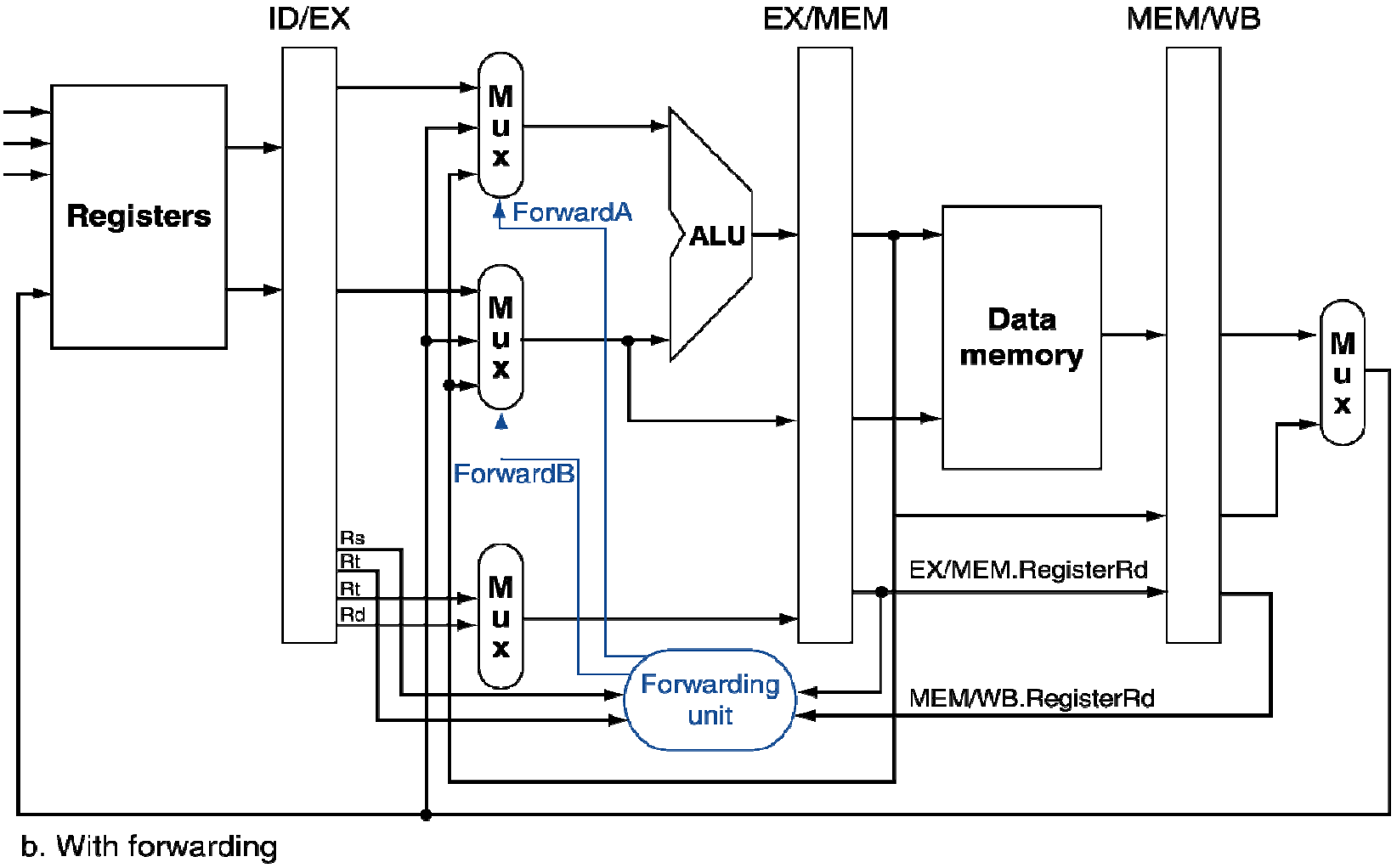
Forwarding Conditions 3

- And only if Rd for that instruction is not $zero
  - If Rd is zero need not write as this register will always have 0
    - If(EX/MEM.RegisterRd ≠ 0)
    - If(MEM/WB.RegisterRd ≠ 0)

Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Forwarding Paths



b. With forwarding

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

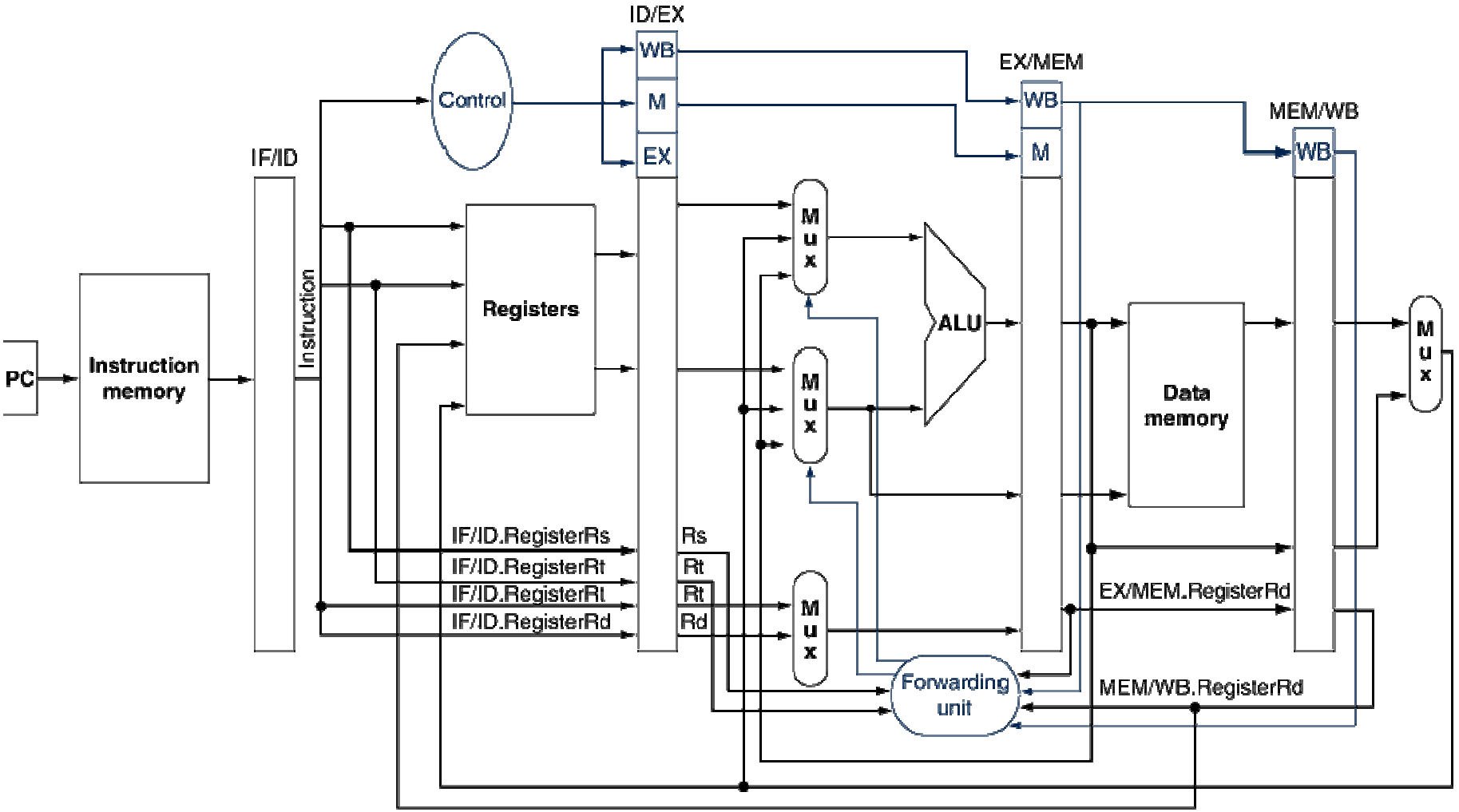
Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 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 ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

The control values for the forwarding multiplexors

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.

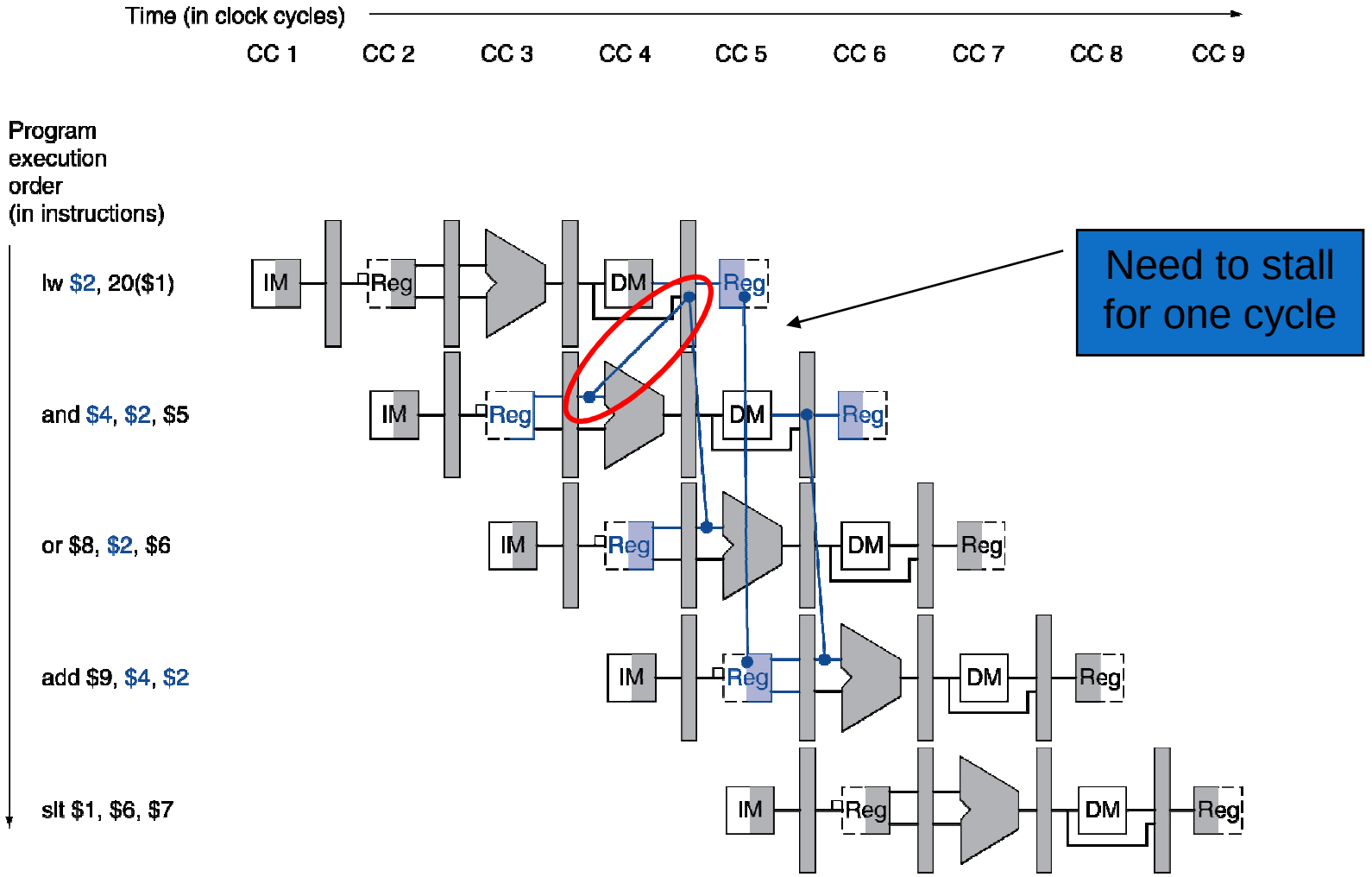
Datapath with Forwarding



multiplexors to the inputs to the ALU consider data forwarding

II: Data Hazard

Load-Use Data Hazard



Eg:

lw $2, 20($1) # lw instruction's reg rt is next and instructions's rs

and $4, $2, $5

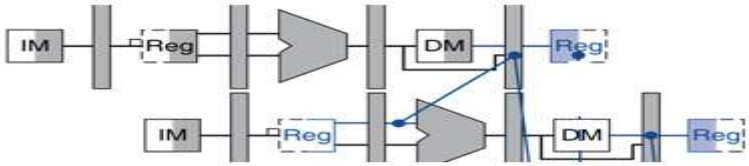
Load-Use Hazard Detection

- *Use* ***"hazard detection unit"***
- *It* operates during the ID stage
- It insert stall

Eg:

lw $2, 20($1

and $4, $2, $5



1: ID/EX.MemRead : checks if instruction is a load instruction

2: ID/EX.RegisterRt : Lw instructions destination reg

IF/ID.RegisterRs, IF/ID.RegisterRt : ALU instructions operand

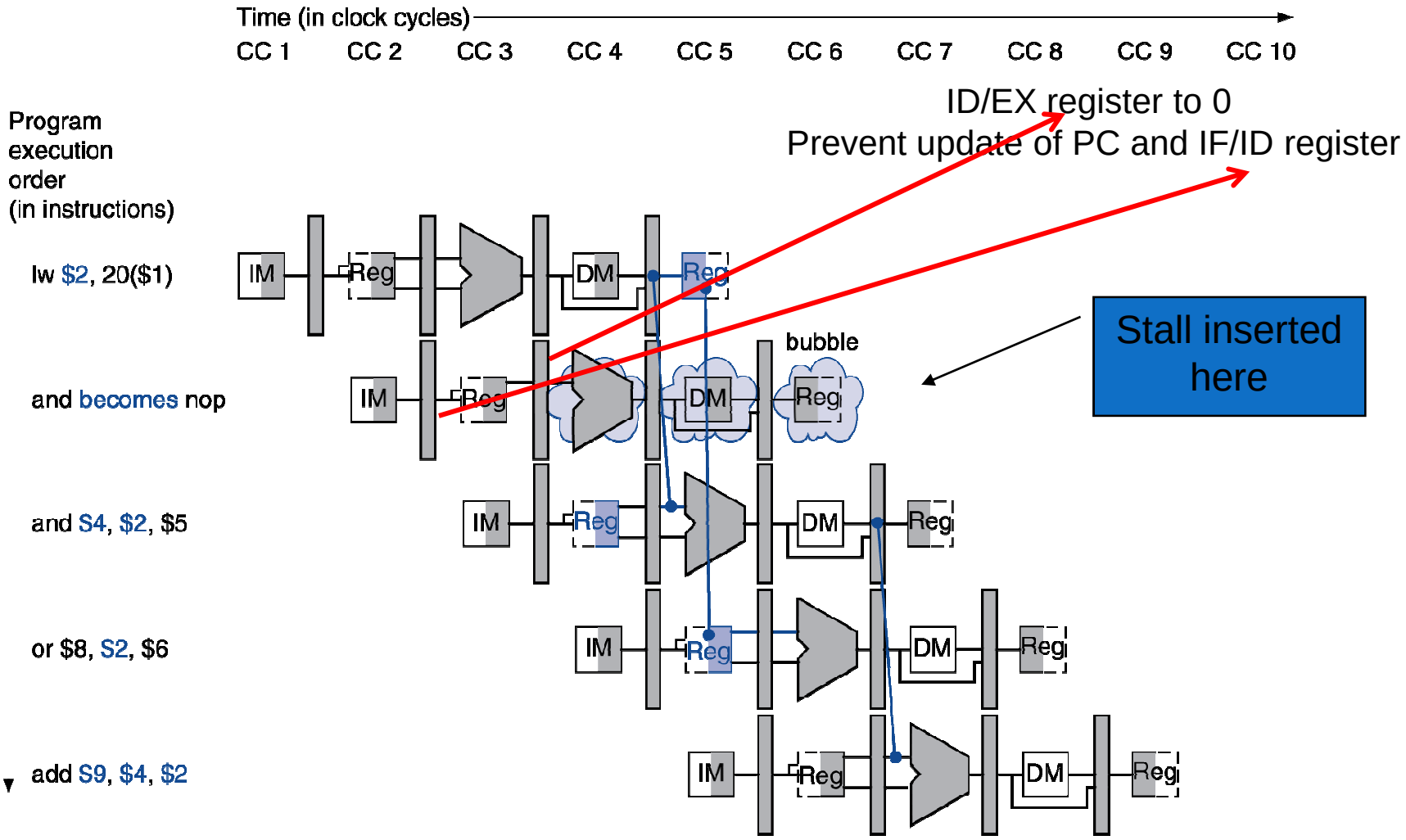
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
    stall

How to Stall the Pipeline

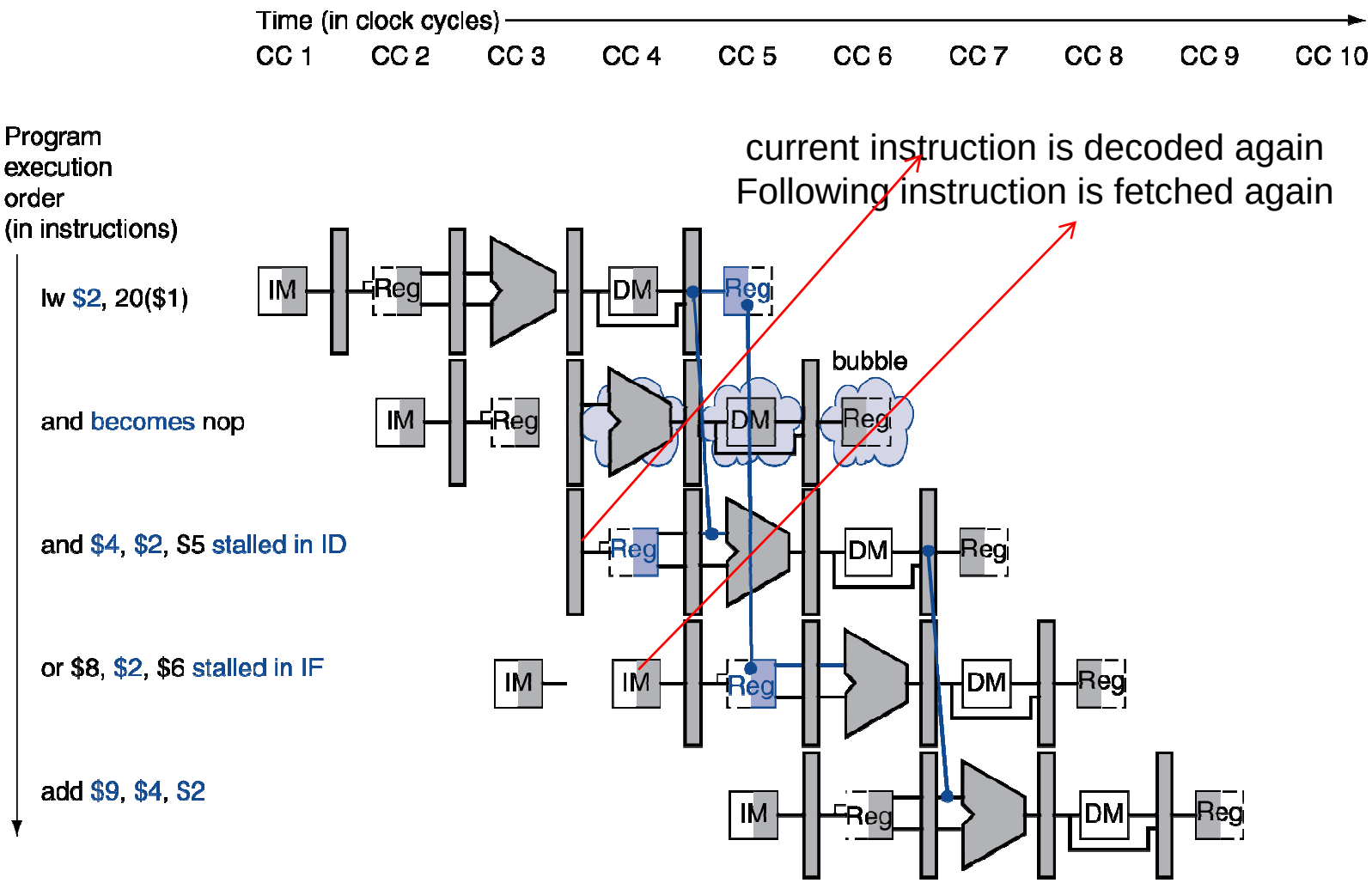
Stall : using a `nop` (no-operation) instruction

1. Force control values in ID/EX register to 0
   - EX, MEM and WB do `nop` (no-operation)
2. Prevent update of PC and IF/ID register
   - current instruction is decoded again
   - Following instruction is fetched again
   - 1-cycle stall allows MEM to read data for `lw`

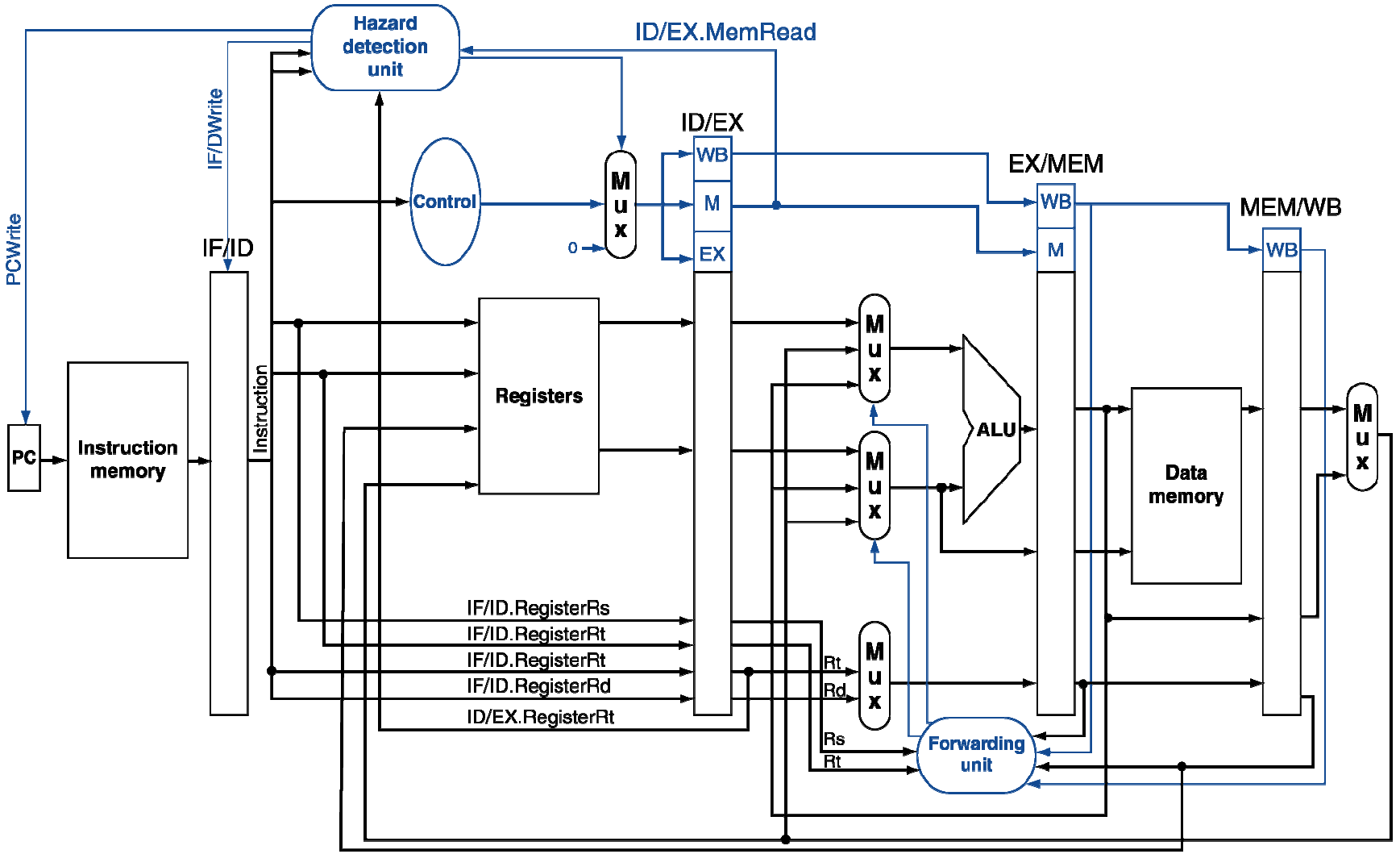
Stall/Bubble in the Pipeline



Stall/Bubble in the Pipeline



Datapath with Hazard Detection



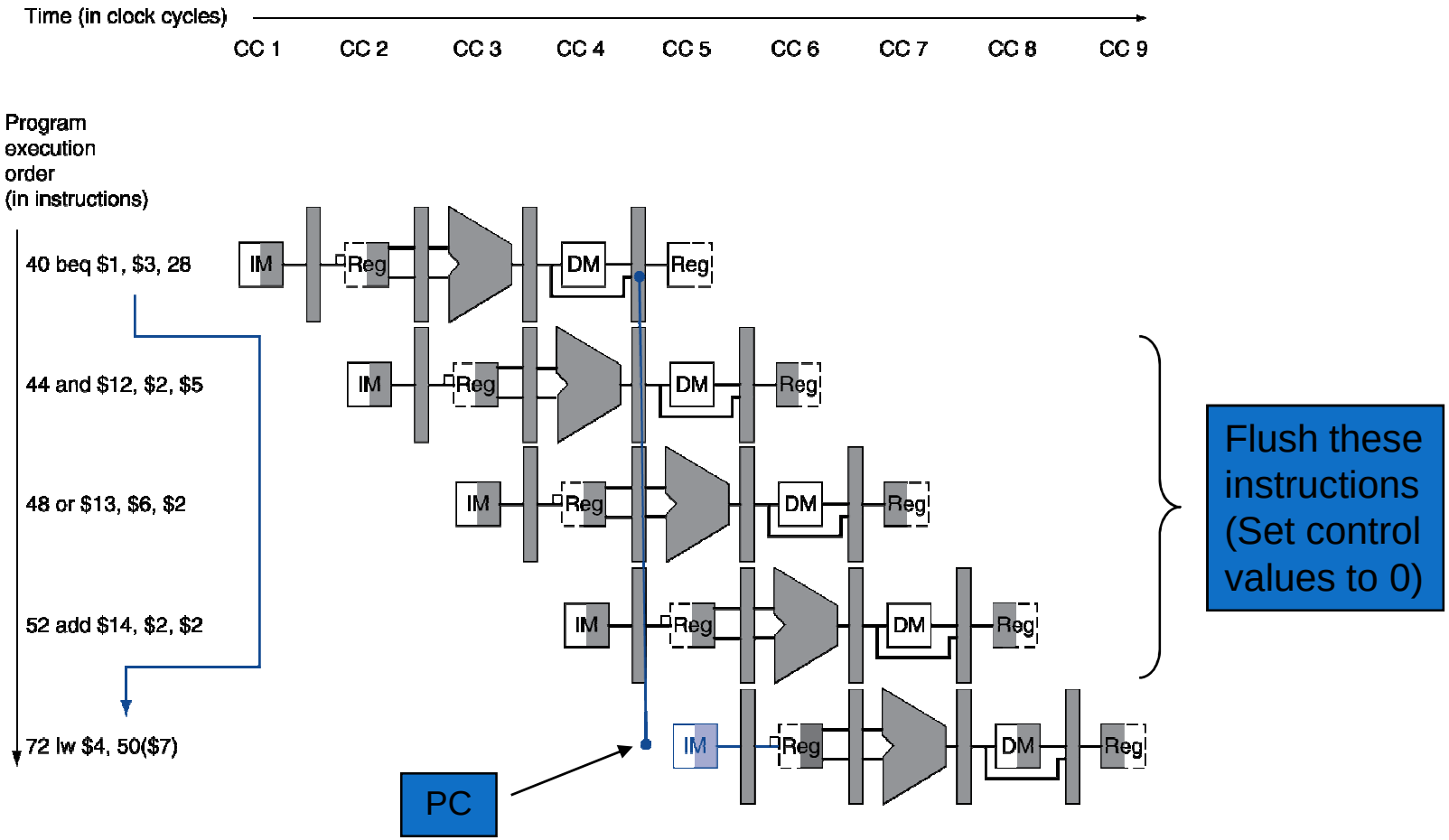
Control Hazard

Control/Branch Hazards

- If branch outcome determined in MEM

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)
```

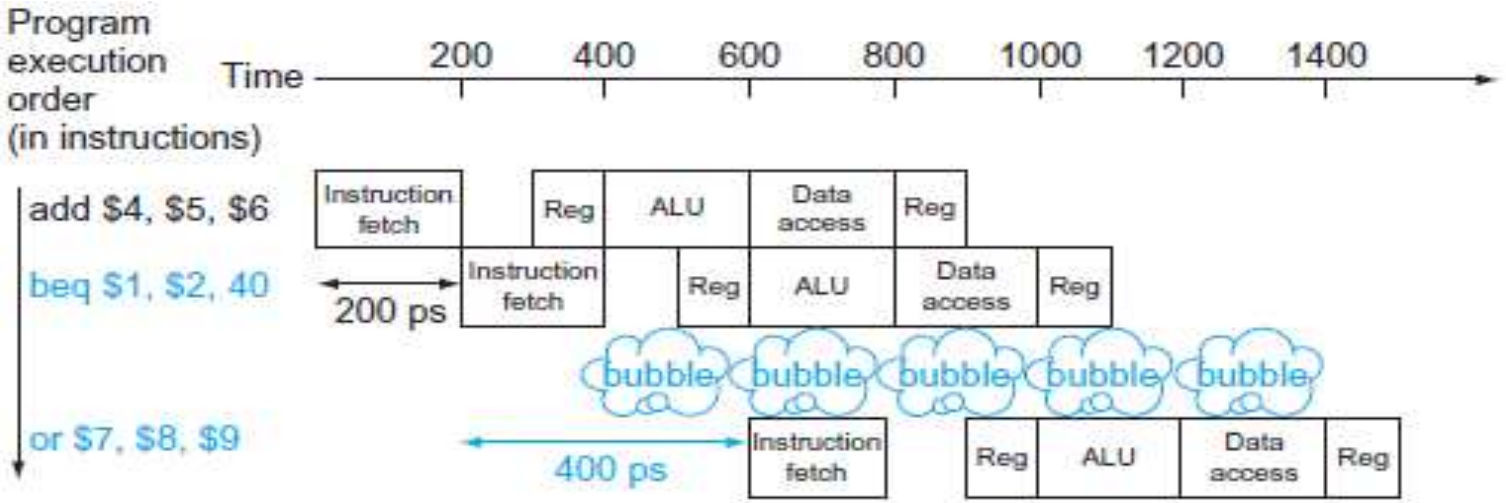


Control Hazard

- Need to make a branch decision based on the results of one instruction while others are execution
  - Stall
  - Predict

Control Hazard

- **Stall**
  - **Wait** until the pipeline determines the outcome of the branch and knows what instruction address to fetch from
  - Put in enough **extra hardware** so that we can test registers, calculate the branch address, and update the PC during the **second stage (ID)** of the pipeline



Reducing Branch Delay

- Next PC for a branch is selected in the MEM stage
- If we move the branch execution earlier in the pipeline, then only fewer instructions need be flushed
- Needs more hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator

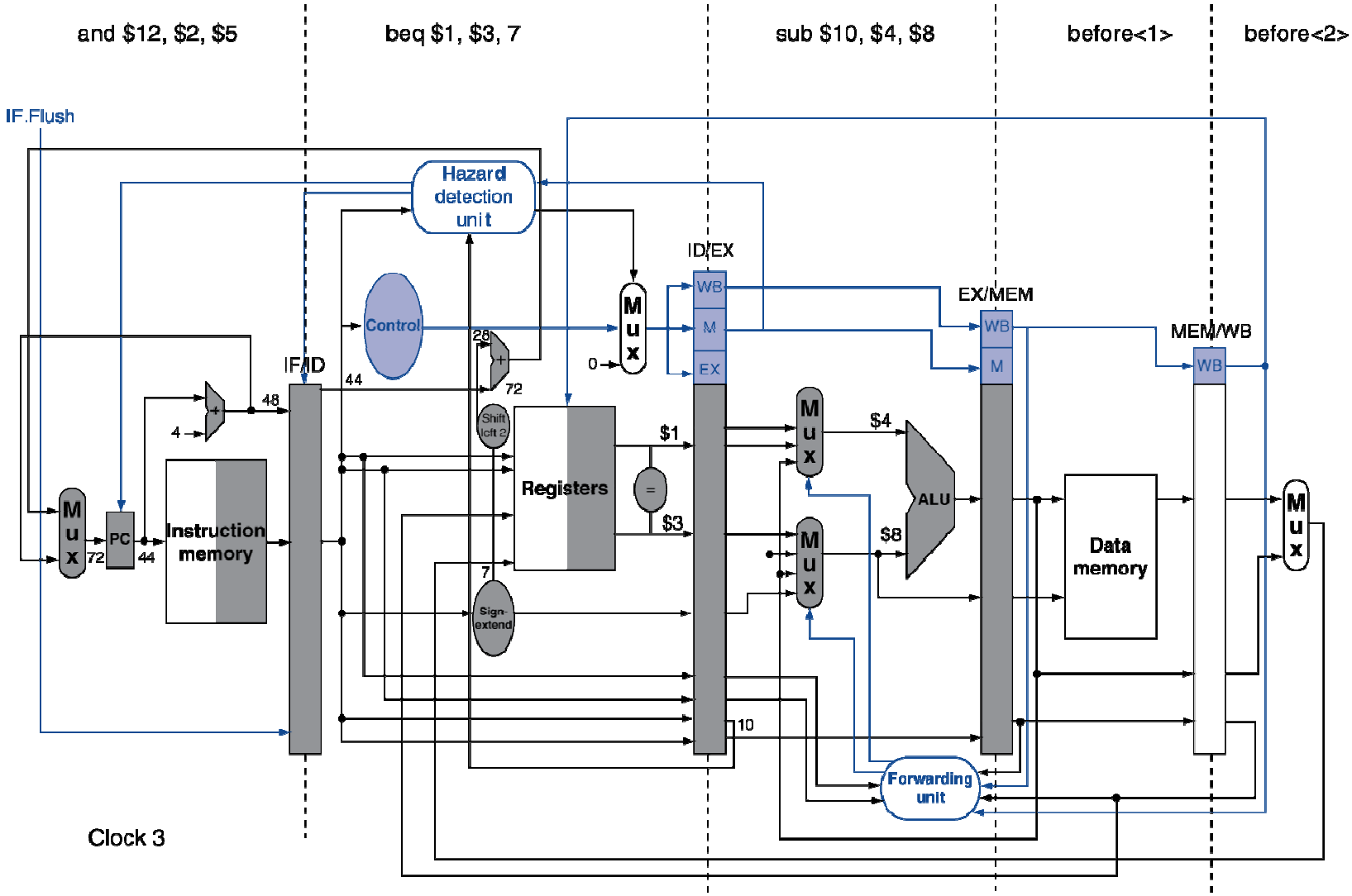
Reducing Branch Delay

- Needs more hardware to determine outcome to ID stage
  - Target address adder
    - Calculation is easy with PC and immediate value so this can be moved from EX stage to ID stage
  - Register comparator
    - Branch decision is the harder part
    - Simple branch test can be done with few gates without ALU
    - Eg: Equality test

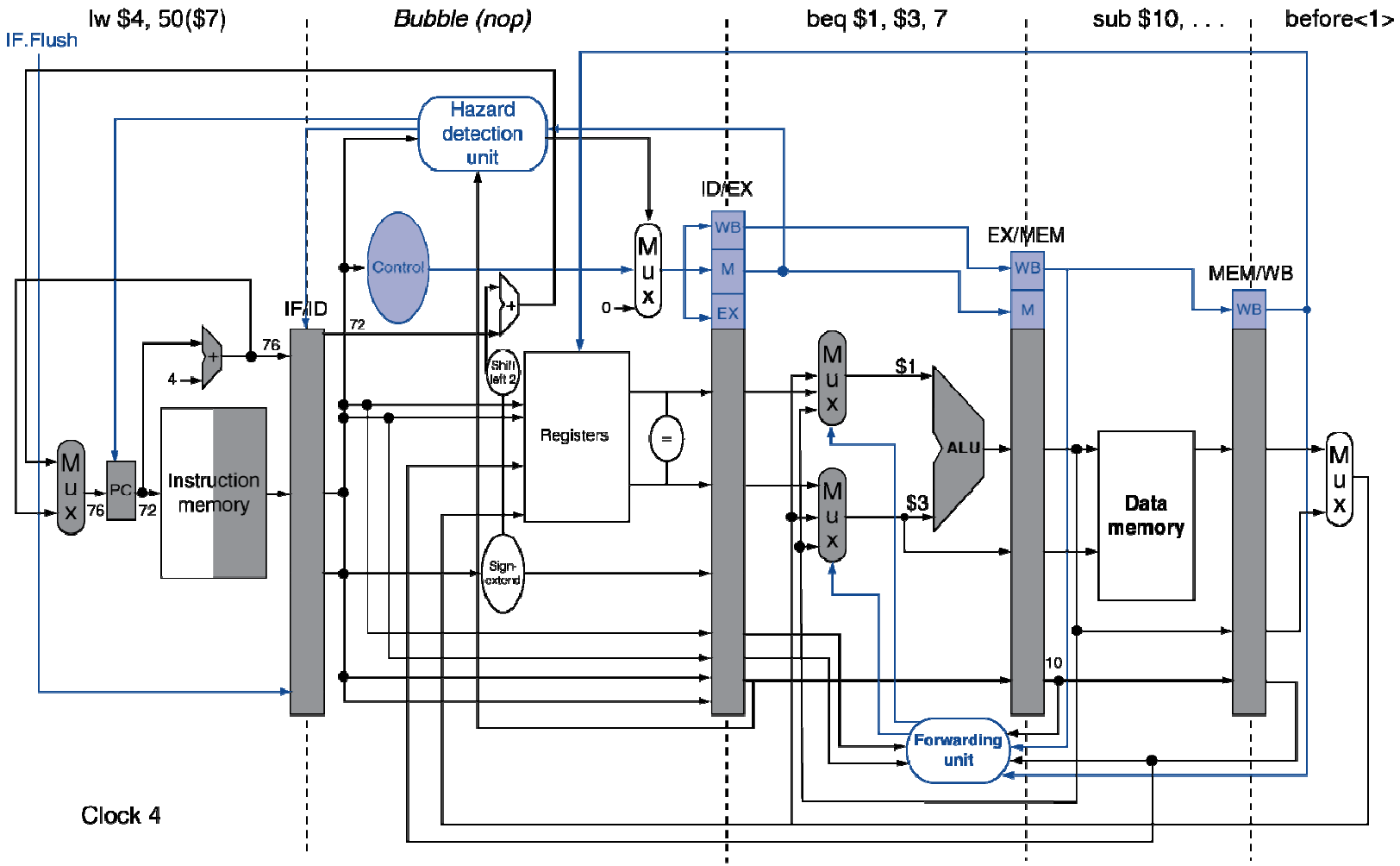
OP1	**0110**
OP2	0111
Xor	0001
Or all bits	1 (not equal)

OP1	**0110**
OP2	0110
Xor	0000
Or all bits	0 (equal)

Example: Branch Taken



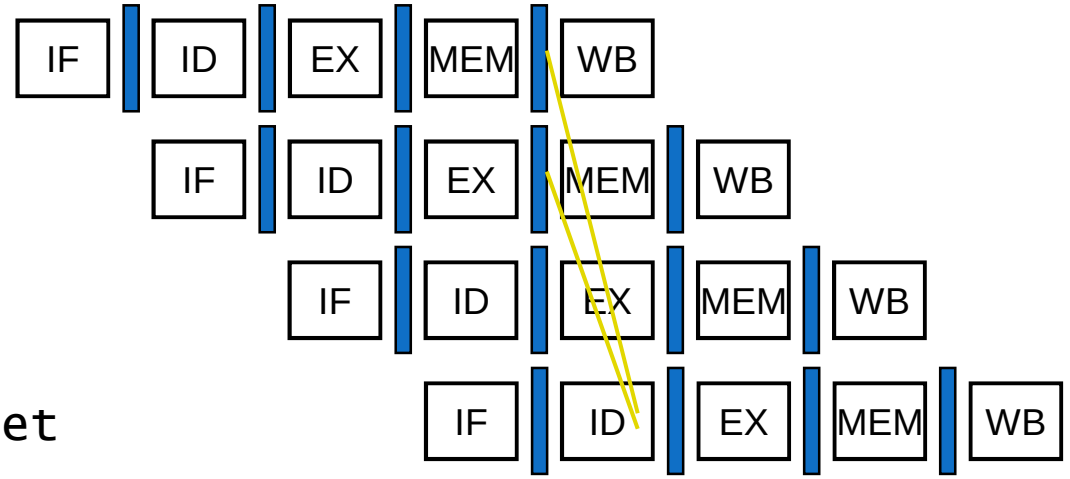
Example: Branch Taken



Data Hazards for Branches

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

```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```

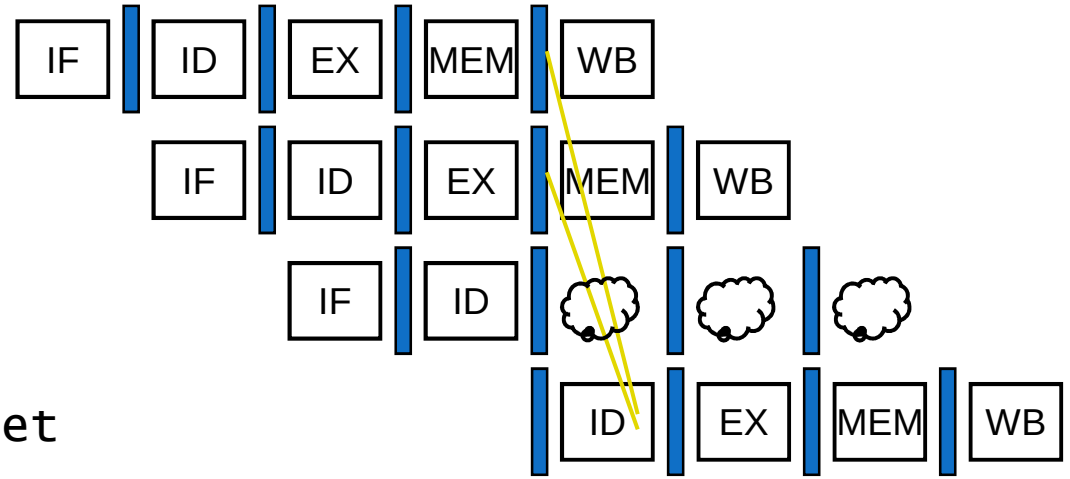


- Can resolve using forwarding

Data Hazards for Branches

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

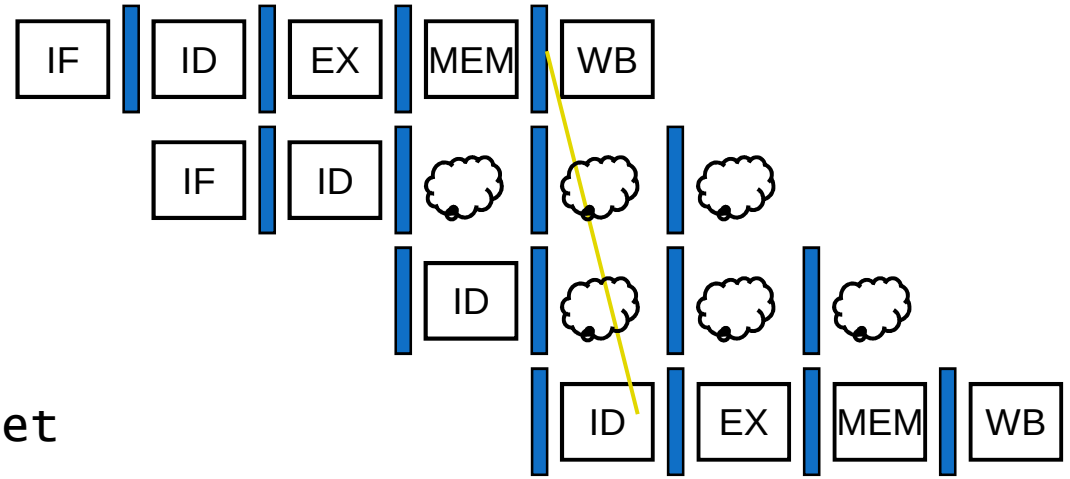
```
lw   $1, addr
add  $4, $5, $6
beq  stalled
beq  $1, $4, target
```



Data Hazards for Branches

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

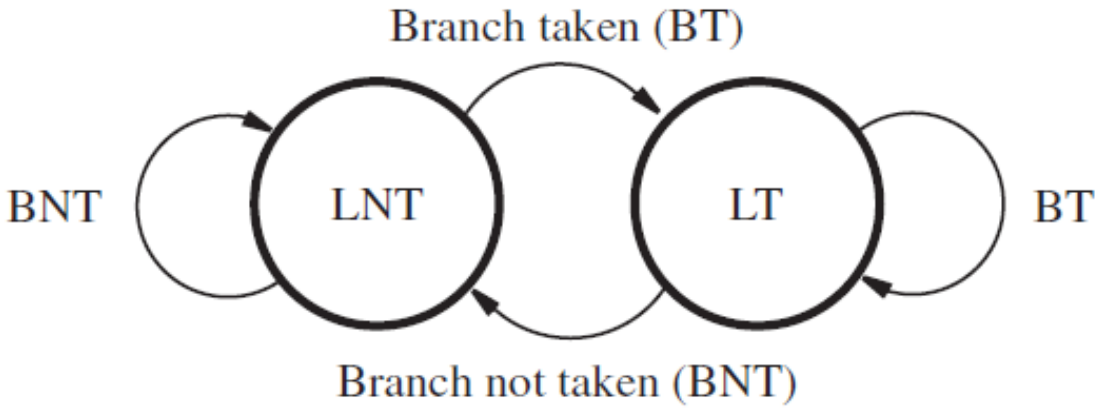
```
lw   $1, addr
beq  stalled
beq  stalled
beq  $1, $0, target
```



Control Hazard

- **Static Prediction**
  - Assume branch will take
  - Assume branch will not take
    - When prediction is right, the pipeline proceeds at full speed
    - If the prediction is wrong, the instructions that are being fetched and decoded must be discarded
    - To discard instructions change the original control values to 0s
    - Discard instruction in IF, ID, and EX stages
- **Dynamic Prediction**
  - Branch prediction made based on history

1-Bit Dynamic Predictor



(a) A 2-state algorithm

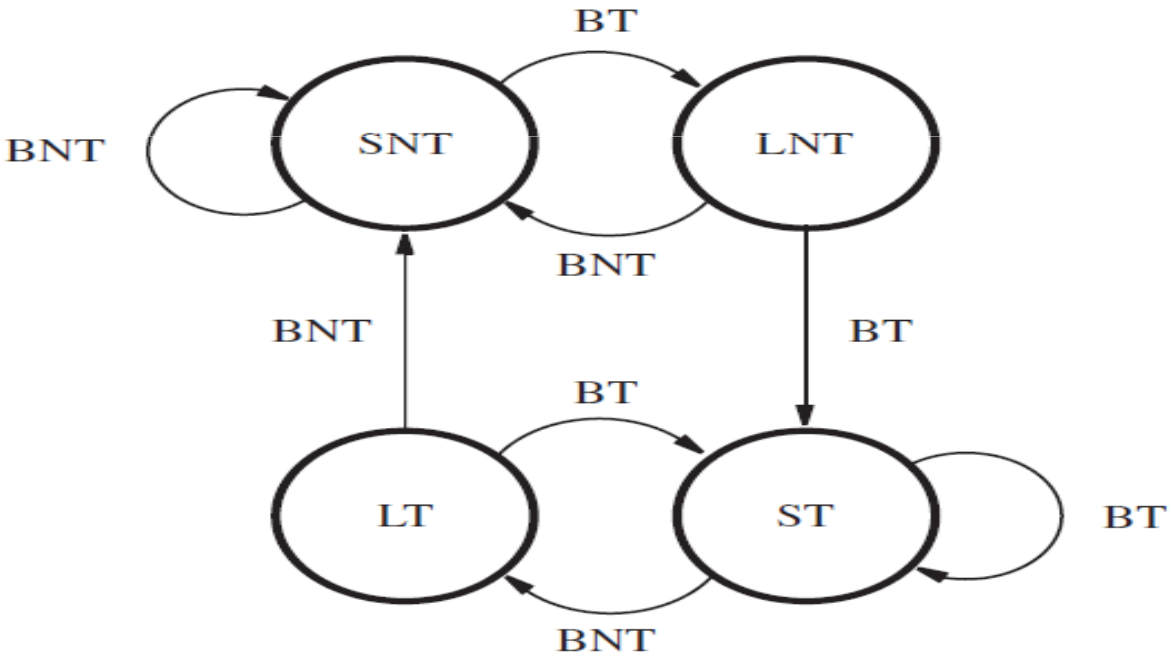
1-Bit Predictor: Shortcoming

- 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

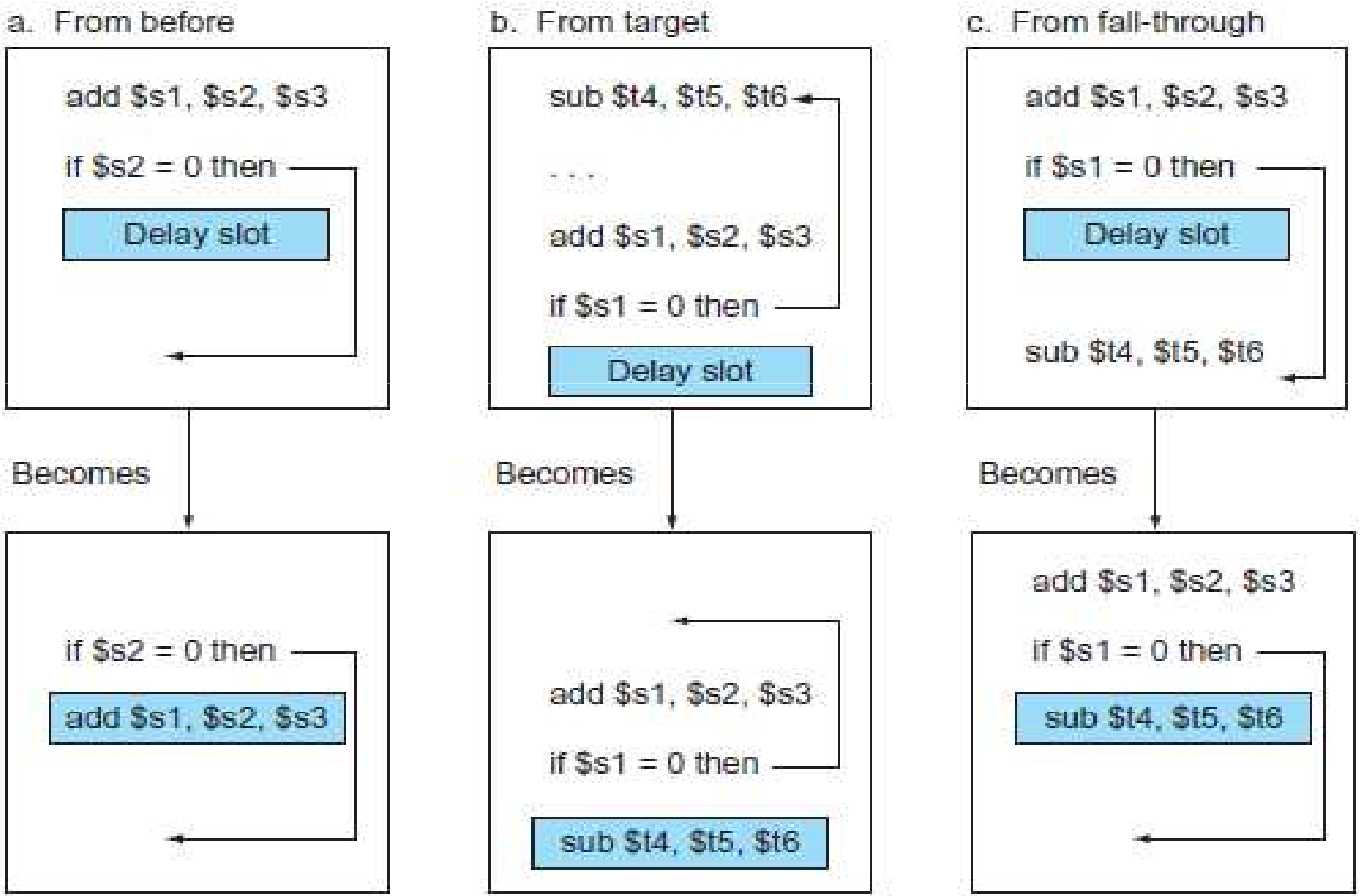
2-Bit Dynamic Predictor

- Only change prediction on two successive mispredictions
- ST - Strongly likely to be taken
- LT - Likely to be taken
- LNT - Likely not to be taken
- SNT - Strongly likely not to be taken



(b) A 4-state algorithm

Scheduling the branch delay slot.



Exception

Exceptions and Interrupts

- Flow of instructions are changed by
  - branch or jump instruction
  - "Unexpected" events requiring change in flow of control
- Exception is another form of control hazard

1. **Exception**
   - Arises within the CPU (Internal)
2. **Interrupt**
   - From an external I/O controller (External)

- Dealing with them without sacrificing performance is hard

Exceptions and Interrupts

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Handling Exceptions

- The basic action that the processor must perform when an exception occurs is
  1. Save the address of the offending instruction in the exception program counter **(EPC)**
  2. **Transfer control** to the operating system at some specified address
  3. Operating system then take the appropriate action,
     1. providing some service to the user program
     2. taking some predefined action in response to an overflow
     3. stopping the execution of the program
     4. reporting an error
  4. Then operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program

Handling Exceptions

- OS must know the
  - Reason for the exception
  - Instruction that caused it

- There are two main methods used to communicate the reason for an exception
  1. status register (called the *Cause register)*
  2. **vectored interrupts**

Handling Exceptions

Cause register

- used in the MIPS architecture
- holds a field that indicates the reason for the exception
- Use single entry point for all exceptions
- the operating system decodes the status register to find the cause

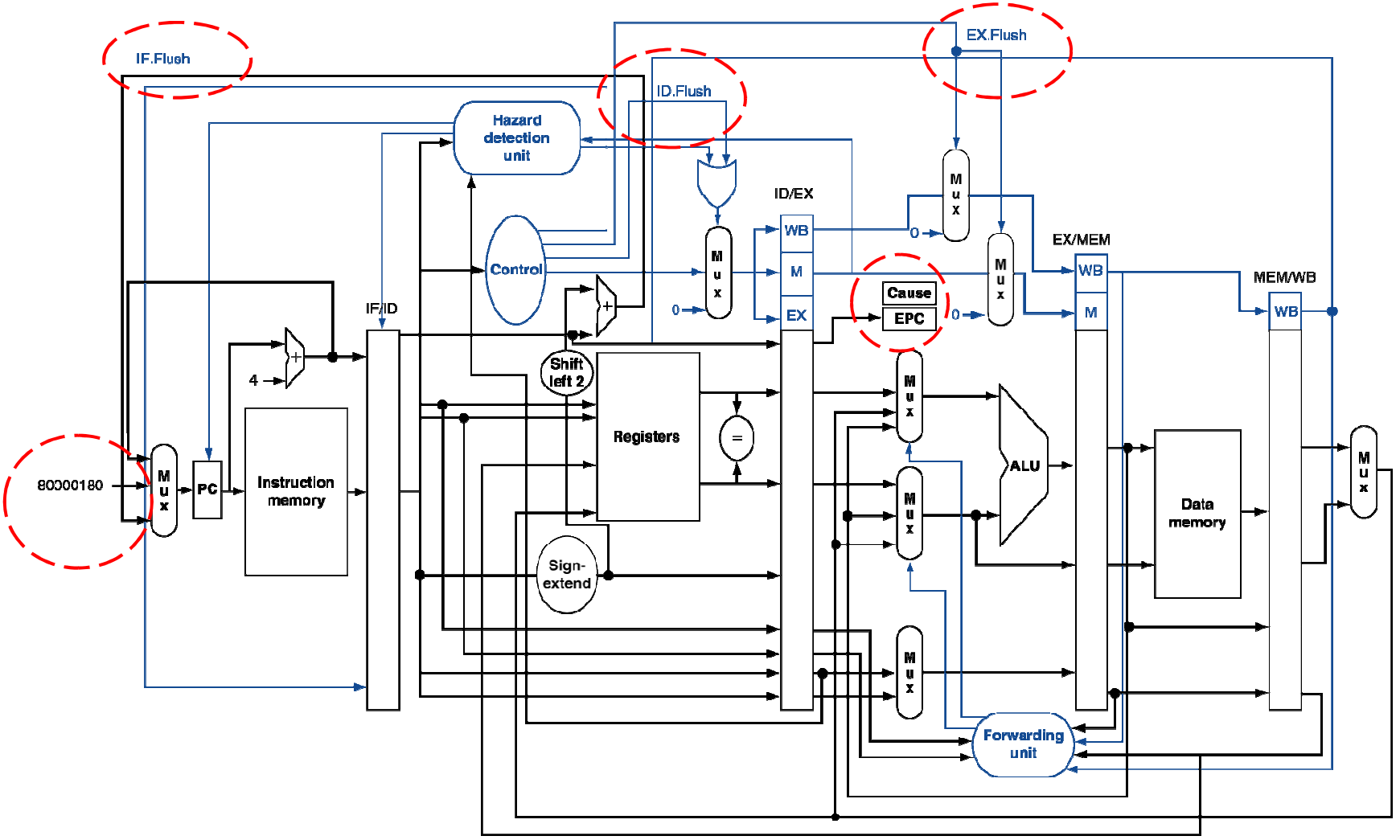
- **vectored interrupts**
  - address to which control is transferred is determined by the cause of the exception

Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

Handling Exceptions in MIPS

- *Cause register is used inMIPS*
- Single entry point being the address 8000 0180hex.
- Handled by adding a few extra registers and control signals
- Add two additional registers to our current MIPS implementation
  - *EPC: A 32-bit register used to hold the address of the affected instruction*
  - *Cause: A register used to record the cause of the exception.*
    - 32 bits register
    - 10 representing an undefined instruction
    - 12 representing arithmetic overflow

Pipeline with Exceptions



Exceptions in a Pipeline

- Consider overflow on add in EX stage

  `add $1, $2, $1`

  - Complete previous instructions
  - Flush `add` and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler

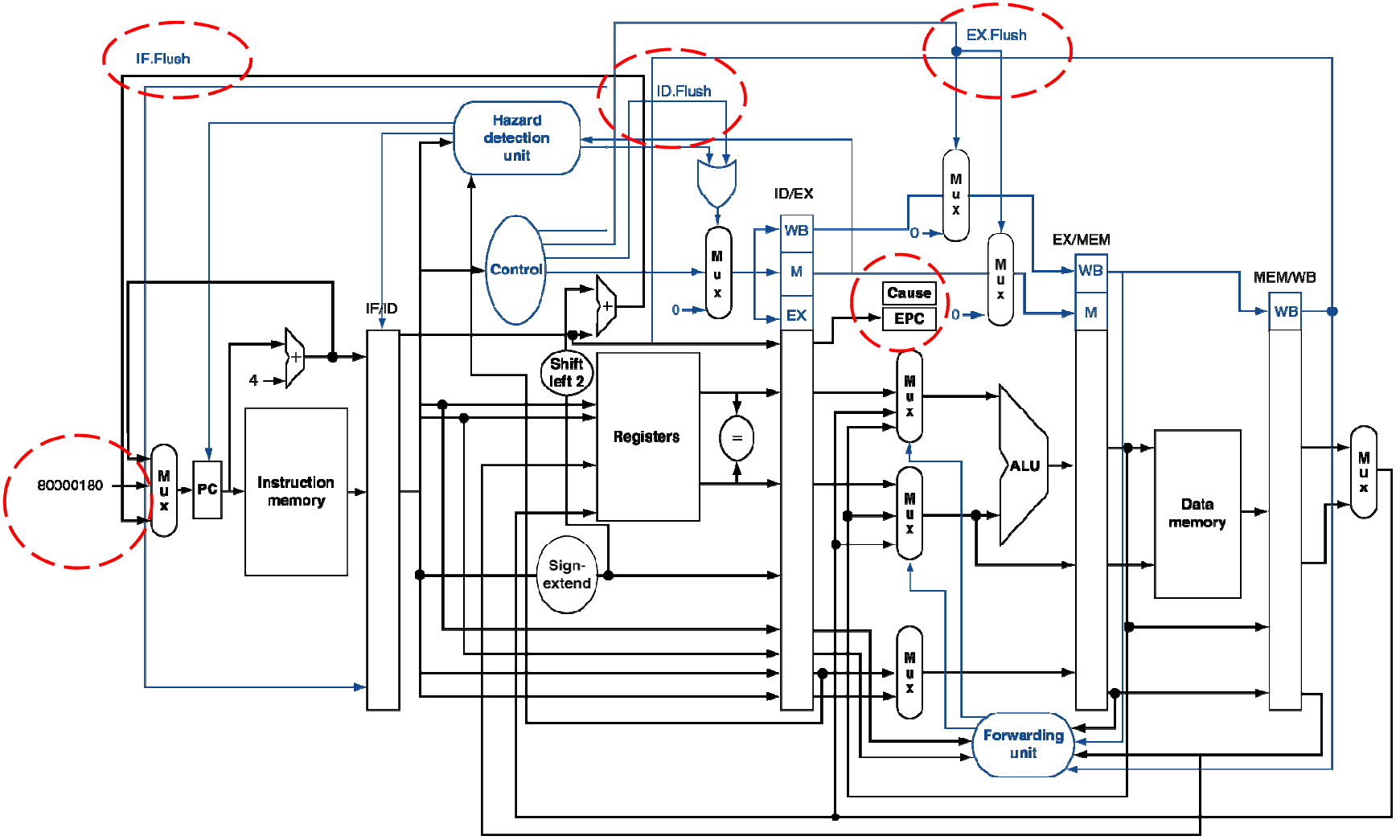
- Similar to mispredicted branch

Exceptions in a Pipeline

Control Signals

- Flush will set all control signa;s to 0 so that the instruction becomes **nop**
- Similar to mispredicted branch
  - branch uses **IF.Flush**
- Two new control signals are used to handle Exceptions
  - **ID.Flush**
  - **EX.Flush**
    - Arithmetic exceptions are found in ALU EX stage
    - So the instructions that precede this exception causing instruction can proceed but not the instruction that succeeds
    - The succeeding instruction will be to its EX and ID stage so use flush

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

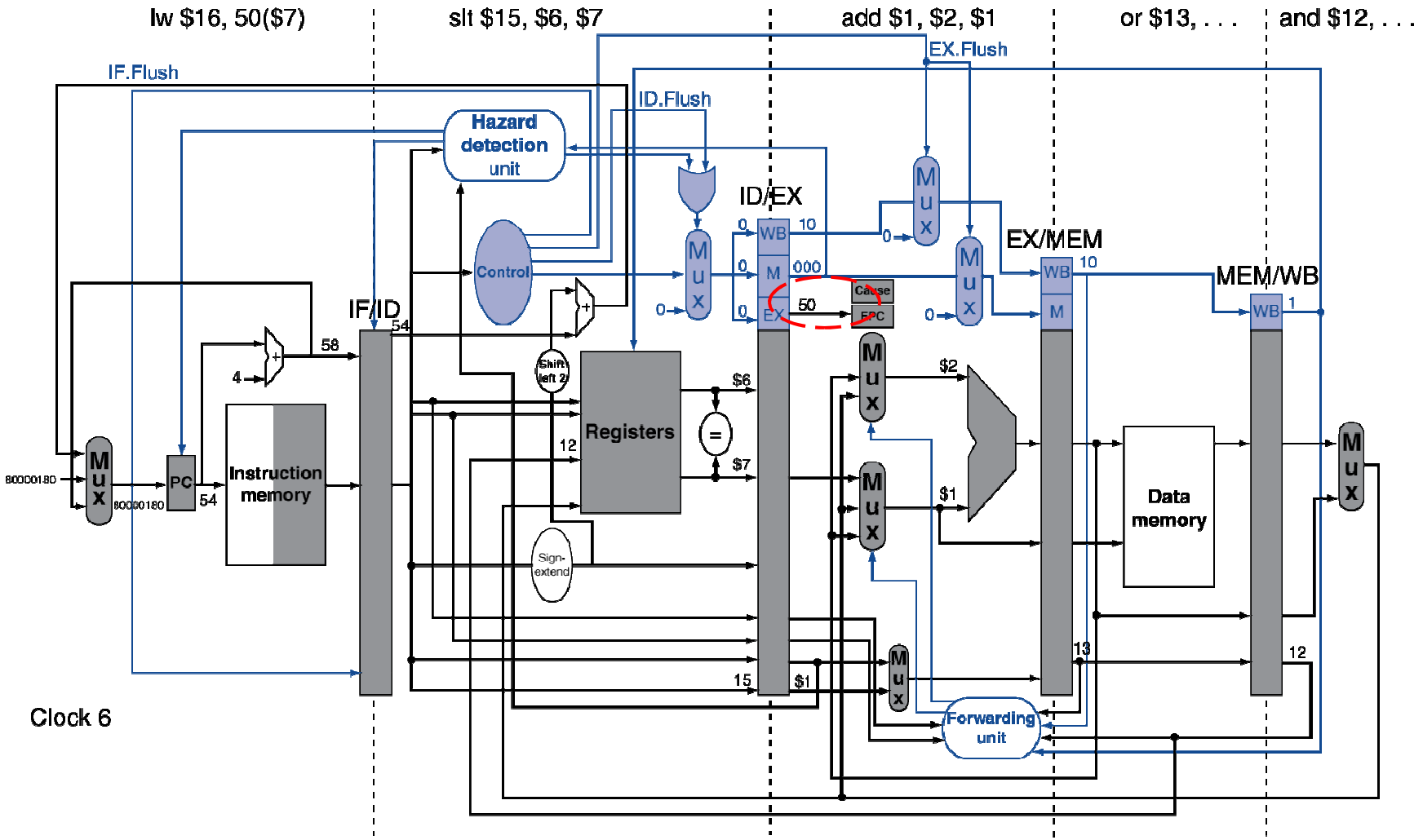
- Exception on `add` in

```
40      sub   $11, $2, $4
44      and   $12, $2, $5
48      or    $13, $2, $6
4c      add   $1,  $2, $1
50      slt   $15, $6, $7
54      lw    $16, 50($7)
…
```

- Handler

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

Exception Example



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      slt     $15, $6, $7
54      lw      $16, 50($7)
                ...
        Handler
80000180        sw   $25, 1000($0)
 80000184       sw   $26, 1004($0)
```