

# Computer Architecture

**Dr. Haroon Mahmood**

**Assistant Professor**

**NUCES Lahore**

# Hazards: problems due to pipelining

## Three Hazard types:

- **Structural**
  - same resource is needed multiple times in the same cycle
- **Data**
  - Instruction depends on result of prior instruction still in the pipeline
  - data dependencies limit pipelining
- **Control**
  - next executed instruction may not be the next specified instruction

# Can always resolve hazards by stalling

**More stall cycles = more CPU time = less performance**

**Increase performance = decrease stall cycles**

# Structural hazards

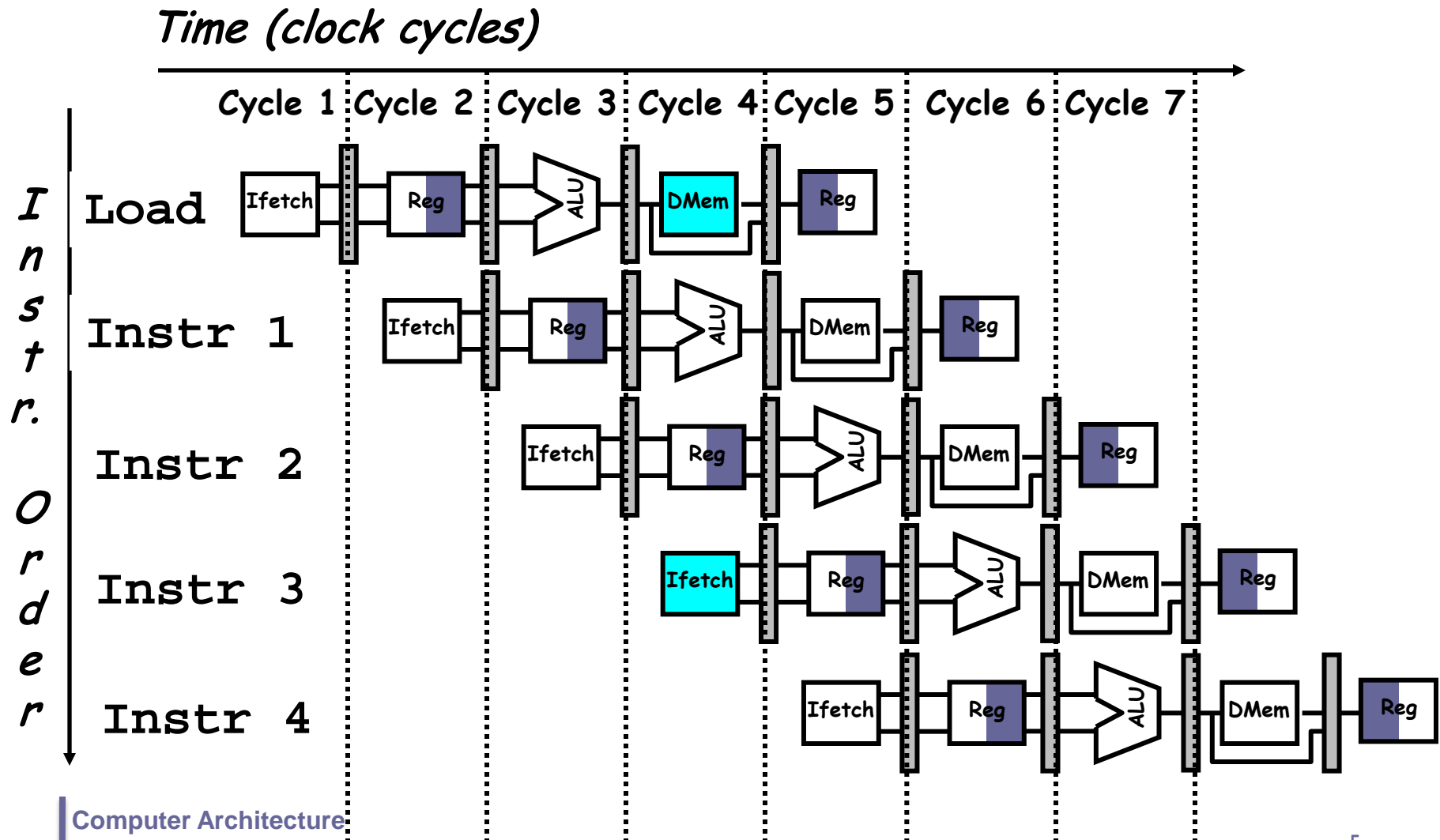
## Examples:

- Two accesses to a single ported memory
- Two operations need the same function unit at the same time
- Two operations need the same function unit in successive cycles, but the unit is not pipelined

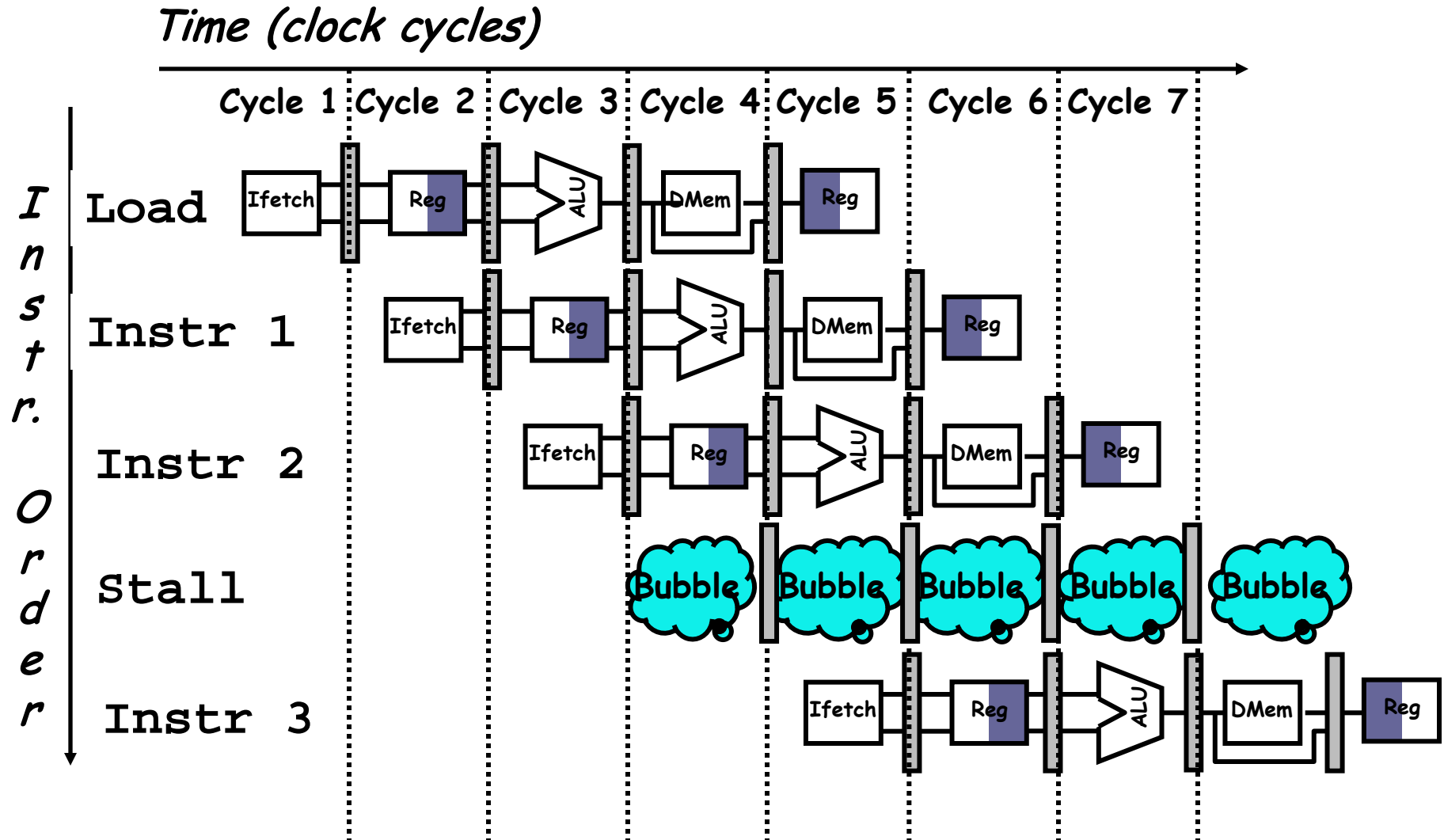
## Solutions:

- stalling
- add more hardware

# One Memory Port/Structural Hazards



# One Memory Port/Structural Hazards



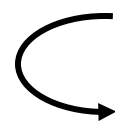
# Data Dependencies

- True dependencies and False dependencies
  - false implies we can remove the dependency
  - true implies we are stuck with it!
- Three types of data dependencies defined in terms of how succeeding instruction depends on preceding instruction
  - **RAW**: Read after Write or Flow dependency
  - **WAR**: Write after Read or anti-dependency
  - **WAW**: Write after Write

# Three Generic Data Hazards

- **Read After Write (RAW)**

Instr<sub>j</sub> tries to read operand before Instr<sub>i</sub> writes it

 I: add **r1**, r2, r3  
J: sub r4, **r1**, r3

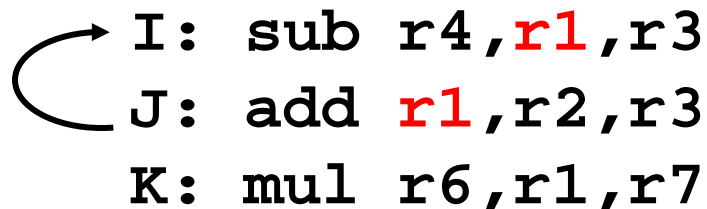
- Caused by a “**Dependence**” (in compiler nomenclature). This hazard results from an actual need for communication.



# Three Generic Data Hazards

- **Write After Read (WAR)**

Instr<sub>j</sub> writes operand before Instr<sub>i</sub> reads it



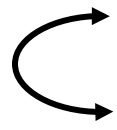
```
I:  sub  r4, r1, r3
J:  add  r1, r2, r3
K:  mul  r6, r1, r7
```

- Called an “**anti-dependence**” by compiler writers. This results from reuse of the name “**r1**”.
- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

# Three Generic Data Hazards

- **Write After Write (WAW)**

Instr<sub>j</sub> writes operand before Instr<sub>i</sub> writes it.



```
I: sub r1, r4, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

- Called an “**output dependence**” by compiler writers  
This also results from the reuse of name “**r1**”.
- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5

# WAR and WAW Dependency

- **Example program (a):**
  - i1: mul r1, r2, r3;
  - i2: add r2, r4, r5;
- **Example program (b):**
  - i1: mul r1, r2, r3;
  - i2: add r1, r4, r5;
- **both cases we have dependence between i1 and i2**
  - in (a) due to r2 must be read before it is written into
  - in (b) due to r1 must be written by i2 after it has been written into by i1

# Data hazards

- **Data dependencies:**
  - **RaW** (read-after-write)
  - **WaW** (write-after-write)
  - **WaR** (write-after-read)
- **Hardware solution:**
  - **Forwarding / Bypassing**
  - **Detection logic**
  - **Stalling**
- **Software solution: Scheduling**

# Data dependences

Three types: RaW, WaR and WaW

```
add r1, r2, 5  
sub r4, r1, r3
```

```
add r1, r2, 5  
sub r2, r4, 1
```

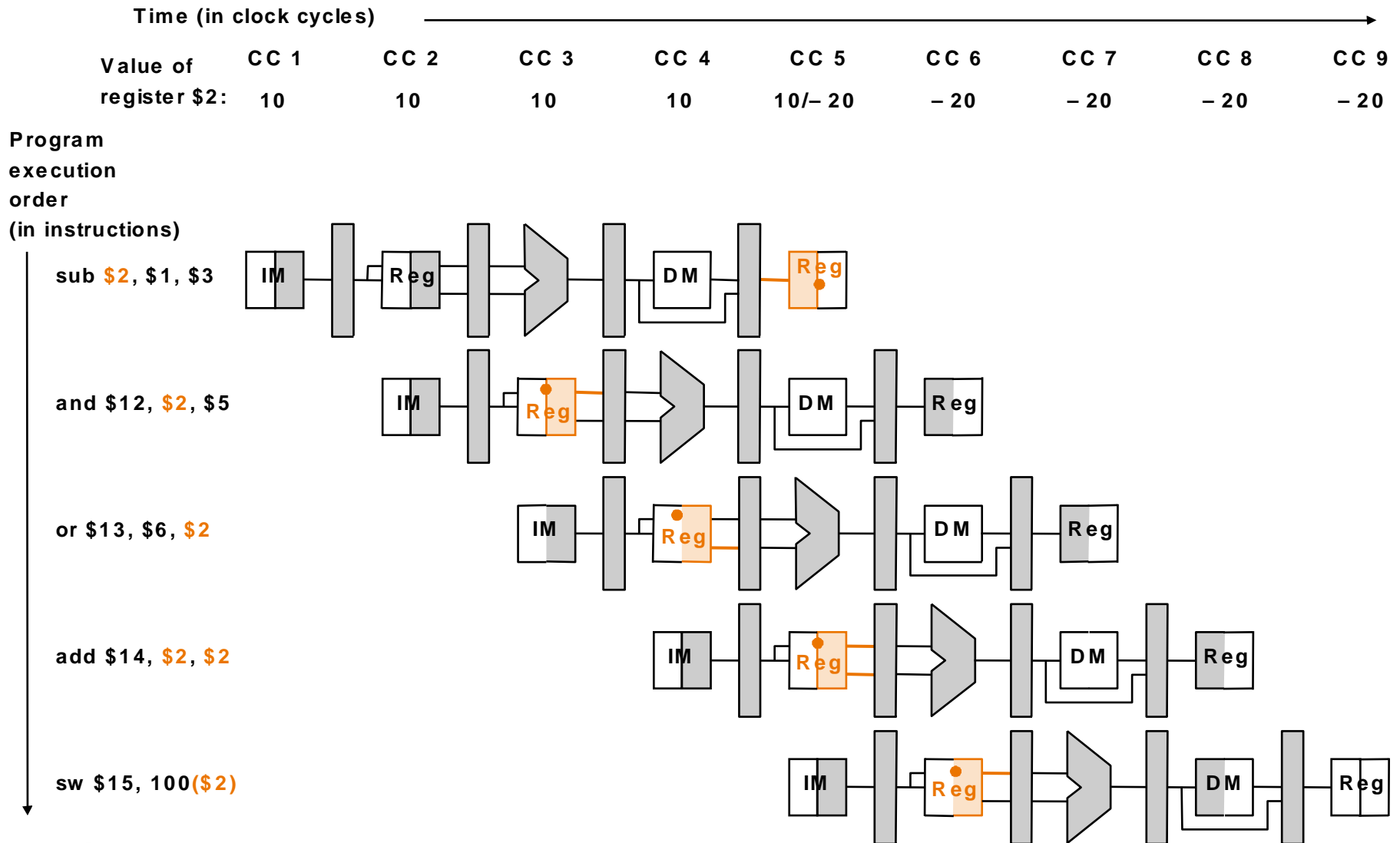
```
add r1, r2, 5  
sub r1, r1, 1
```

```
st  r1, 5(r2)  
ld  r5, 0(r4)
```

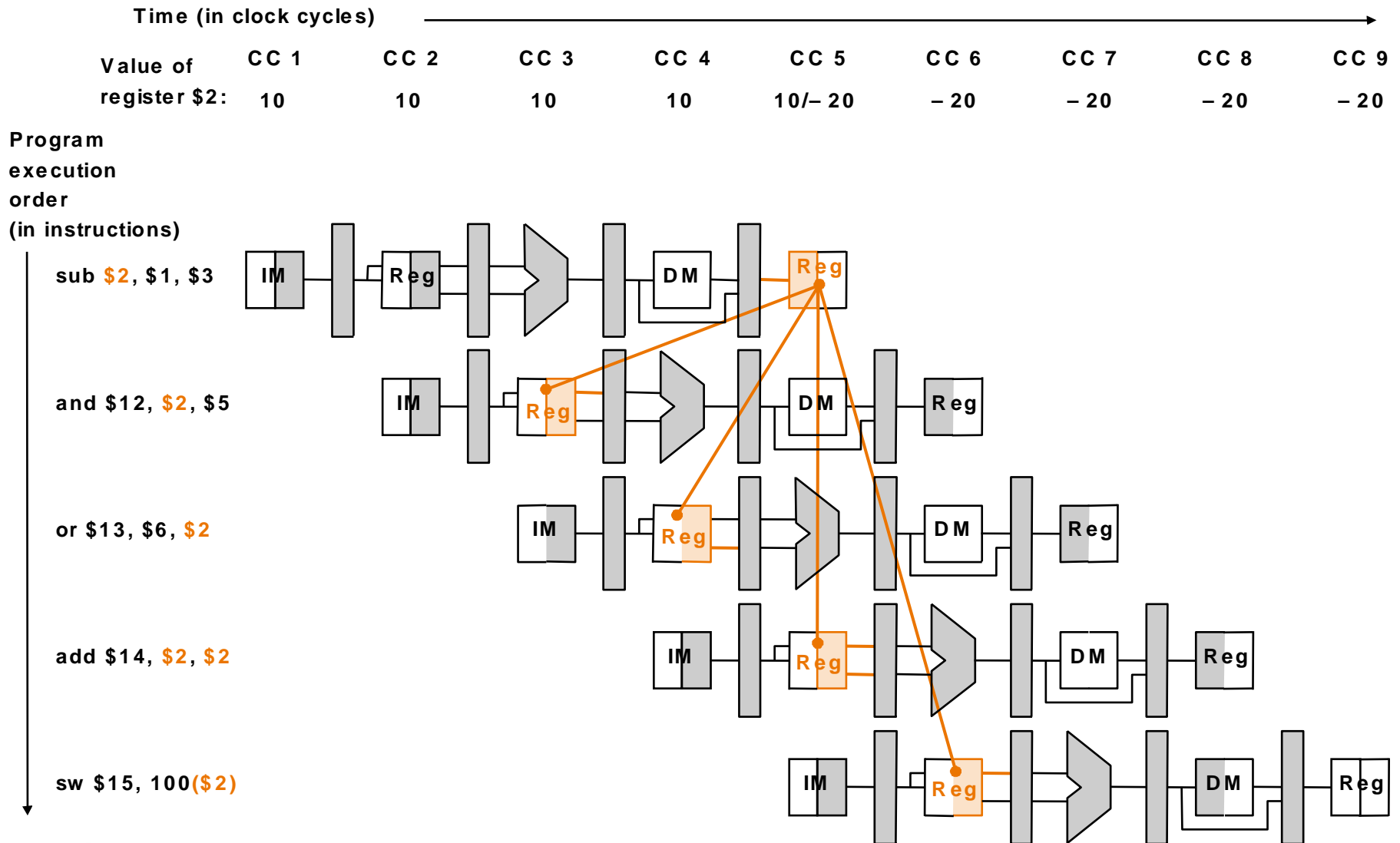
WaW and WaR do not occur in simple pipelines, but they limit scheduling freedom!

⇒ use **register renaming** to solve this!

# RaW on MIPS pipeline



# RaW on MIPS pipeline

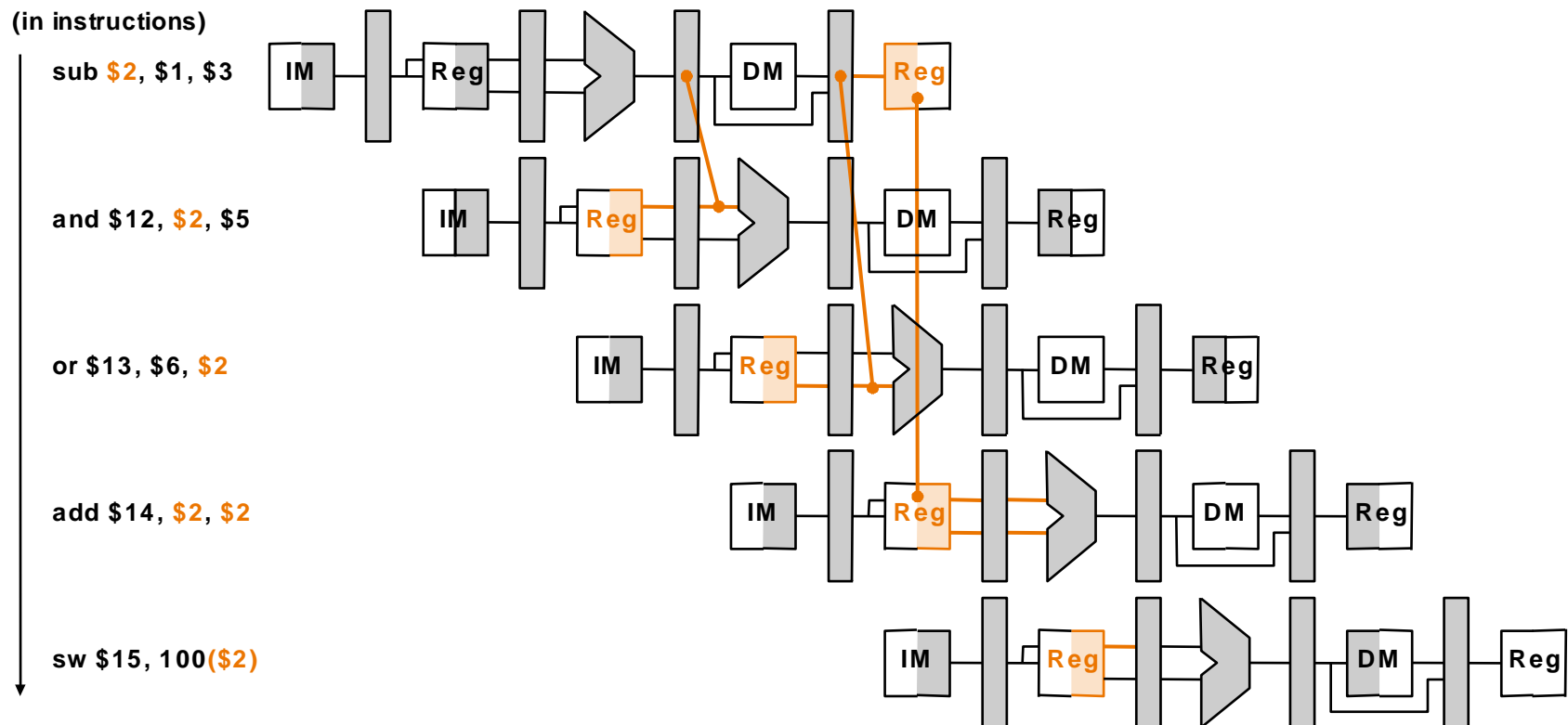


# Forwarding

	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

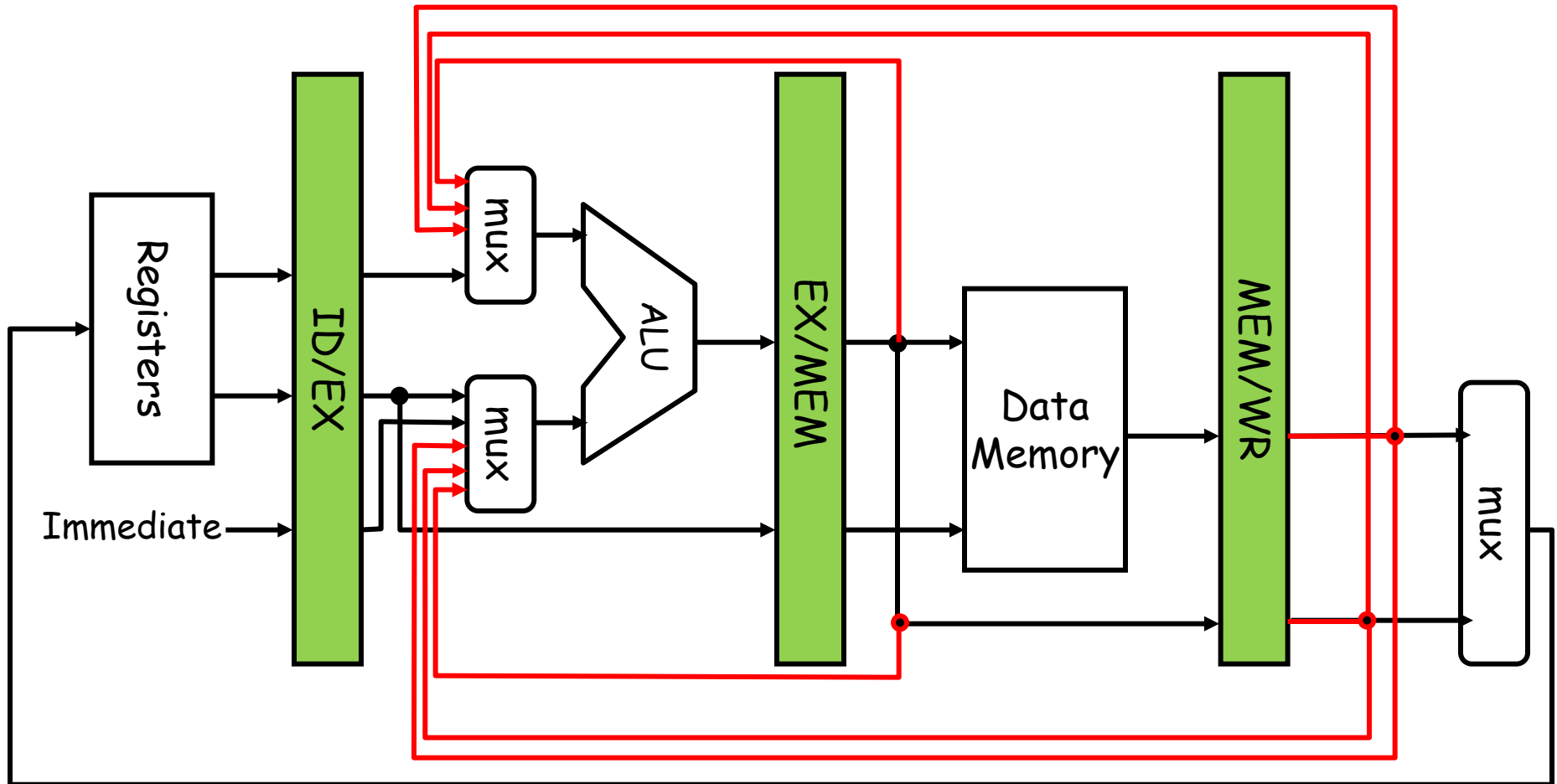
Program

execution order  
(in instructions)

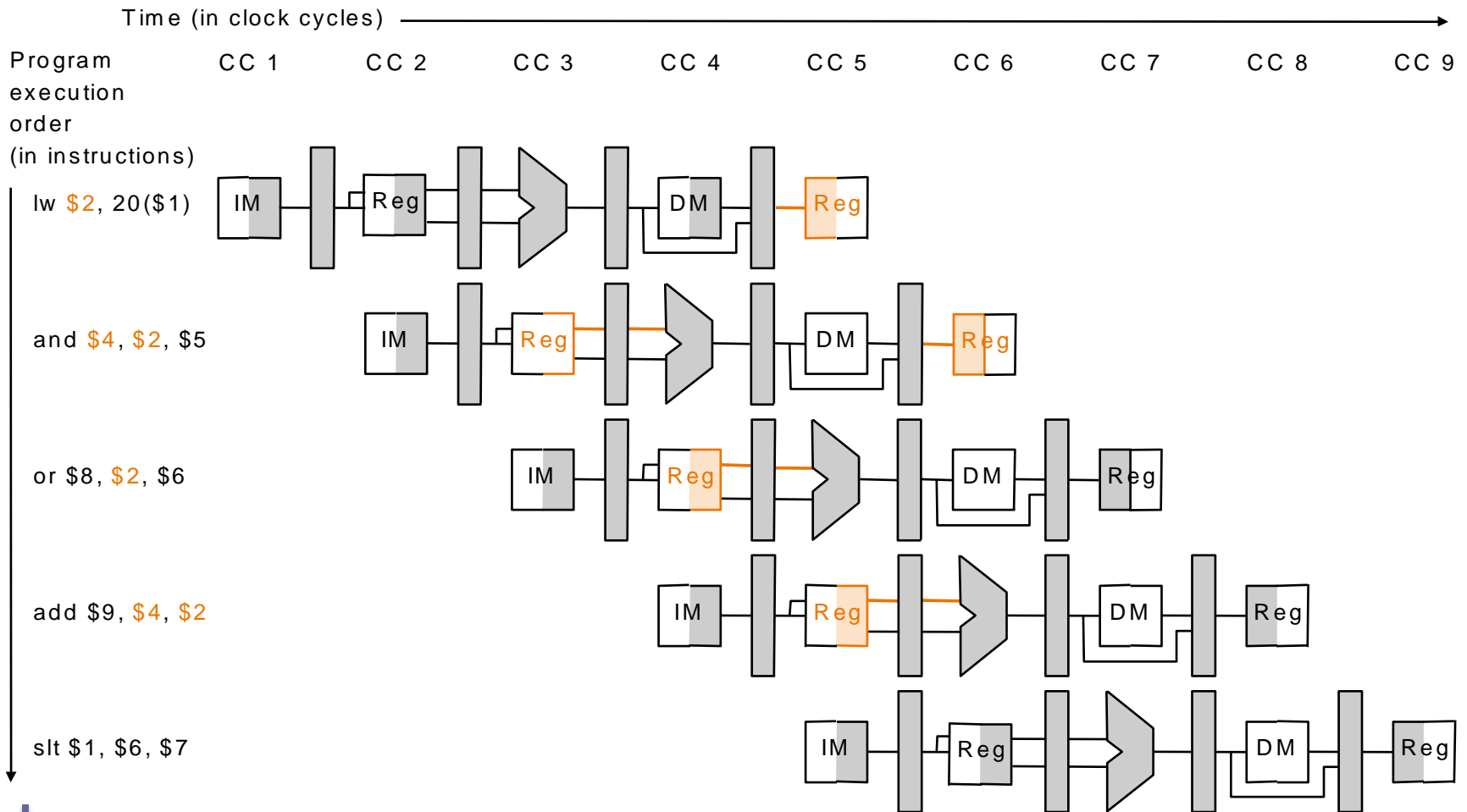




# HW Change for Forwarding

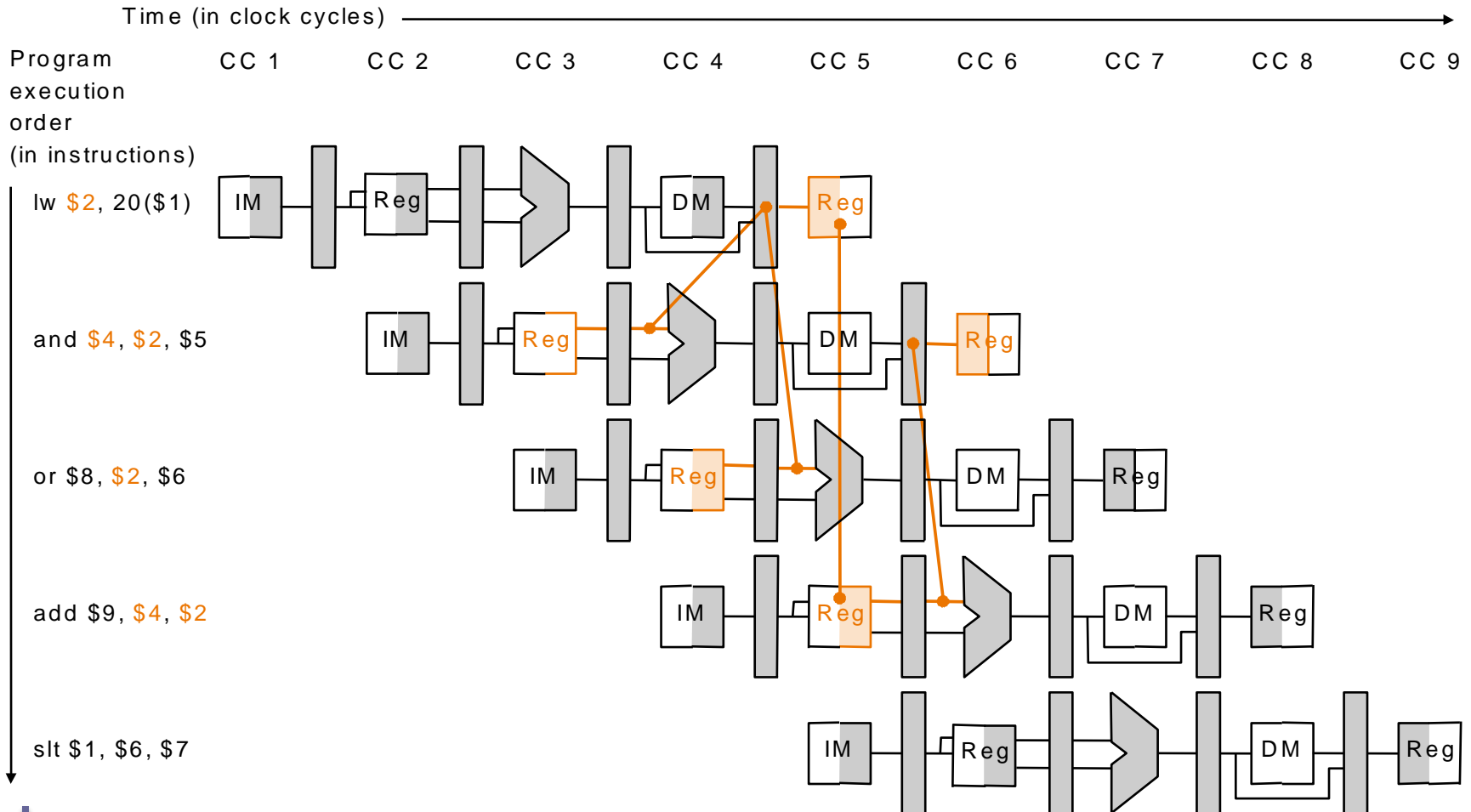


# Does forwarding always works?



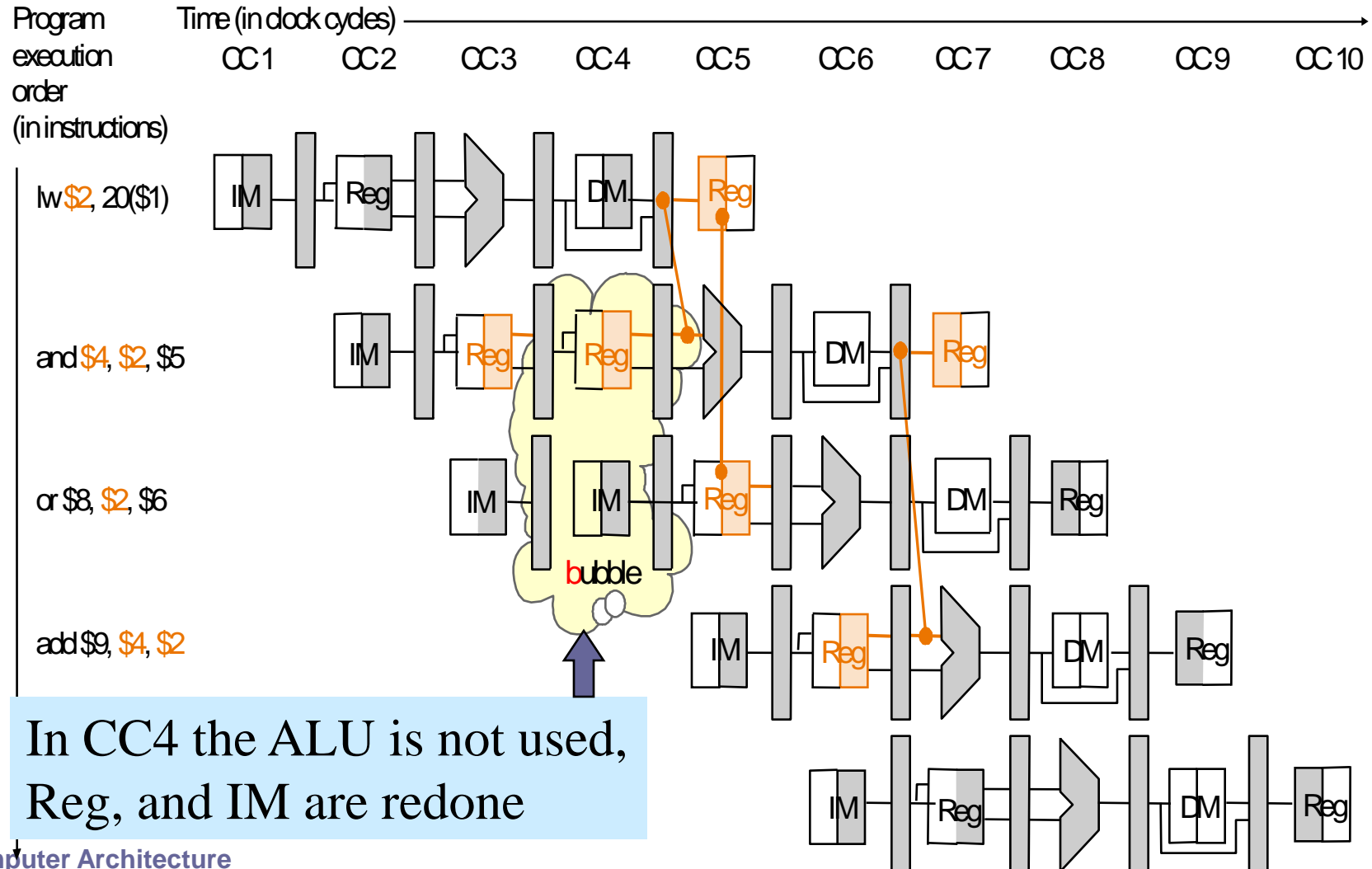
# Can't always forward

- an instruction tries to read register *r* following a load to the same register
- Need a hazard detection unit to “stall” the load instruction



# Stalling

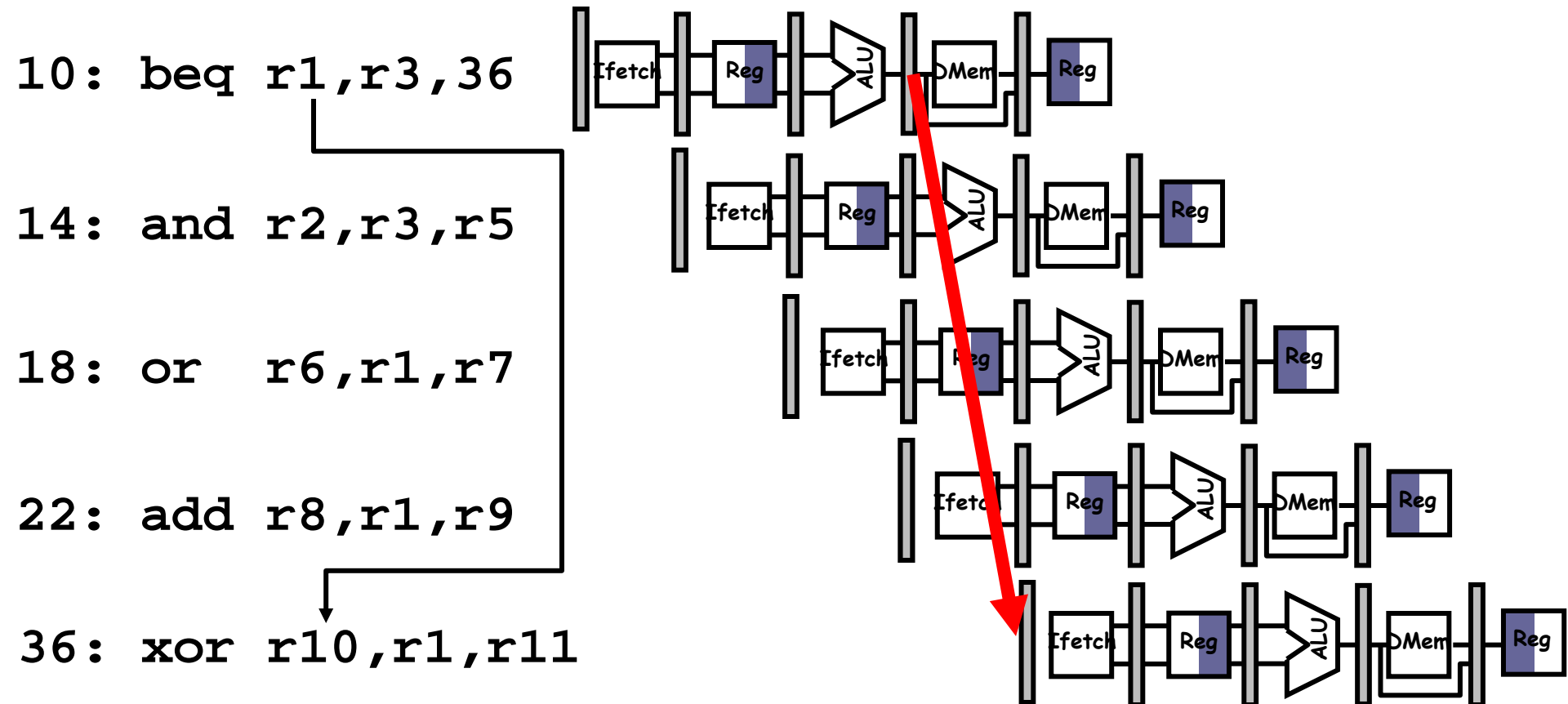
We can stall the pipeline by keeping an instruction in the same stage



# Control Hazards: Branches

- **Instruction flow**
  - Stream of instructions processed by Inst. Fetch unit
  - Speed of “input flow” puts bound on rate of outputs generated
- **Branch instruction affects instruction flow**
  - Do not know next instruction to be executed until branch outcome known
- **When we hit a branch instruction**
  - Need to compute target address (where to branch)
  - Resolution of branch condition (true or false)
  - Might need to ‘flush’ pipeline if other instructions have been fetched for execution

# Control Hazard on Branches



# Control Hazards

- **Simple techniques to handle control hazard stalls:**
  - **for every branch, introduce a stall cycle (note: every 6th instruction is a branch on average!)**
  - **assume the branch is not taken and start fetching the next instruction – if the branch is taken, need hardware to cancel the effect of the wrong-path instructions**
  - **predict the next PC and fetch that instr – if the prediction is wrong, cancel the effect of the wrong-path instructions**
  - **fetch the next instruction (branch delay slot) and execute it anyway**
    - **if the instruction turns out to be on the correct path, useful work was done**
    - **if the instruction turns out to be on the wrong path, hopefully program state is not lost**

# Software only solution?

- Have compiler guarantee that no hazards occur
- Example: where do we insert the “NOPs” ?

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

- Problem: this really slows us down!

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



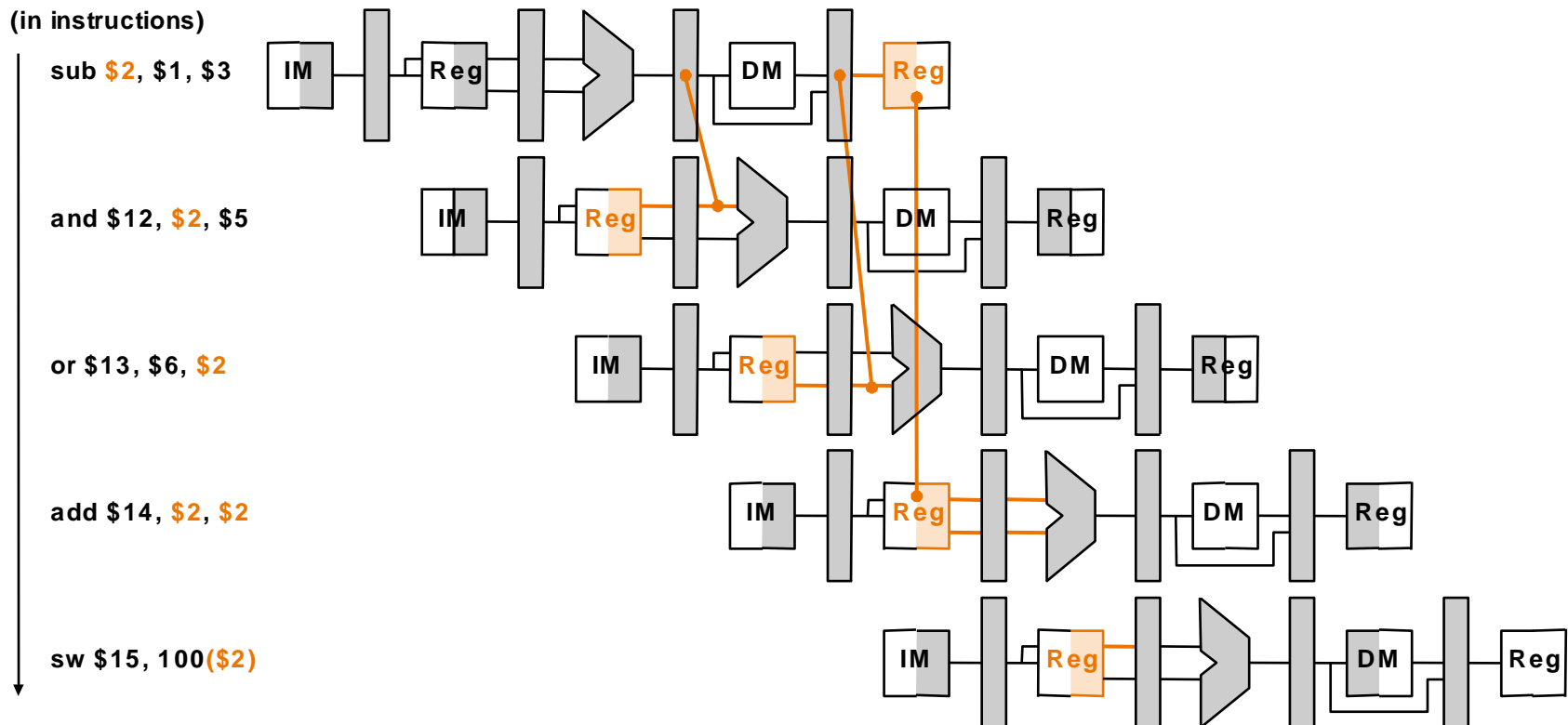
# Forwarding

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

Program

execution order  
(in instructions)



# Dependence Detection

- Notation to describe data hazards
  - 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
- Hazard between following two instructions can be detected when sub instruction is in the MEM stage and “and Instruction” is in the EXE stage
  - sub \$2, \$1,\$3
  - and \$12,\$2,\$5
  - Or \$13, \$6, \$2
  - add \$14,\$2,\$2
  - sw \$15,100(\$2)

**Data Hazard1: Sub-and:** EX/MEM.RegisterRd = ID/EX.RegisterRs = \$2

**Data Hazard2: Sub-or:** MEM/WB.RegisterRd = ID/EX.RegisterRt = \$2

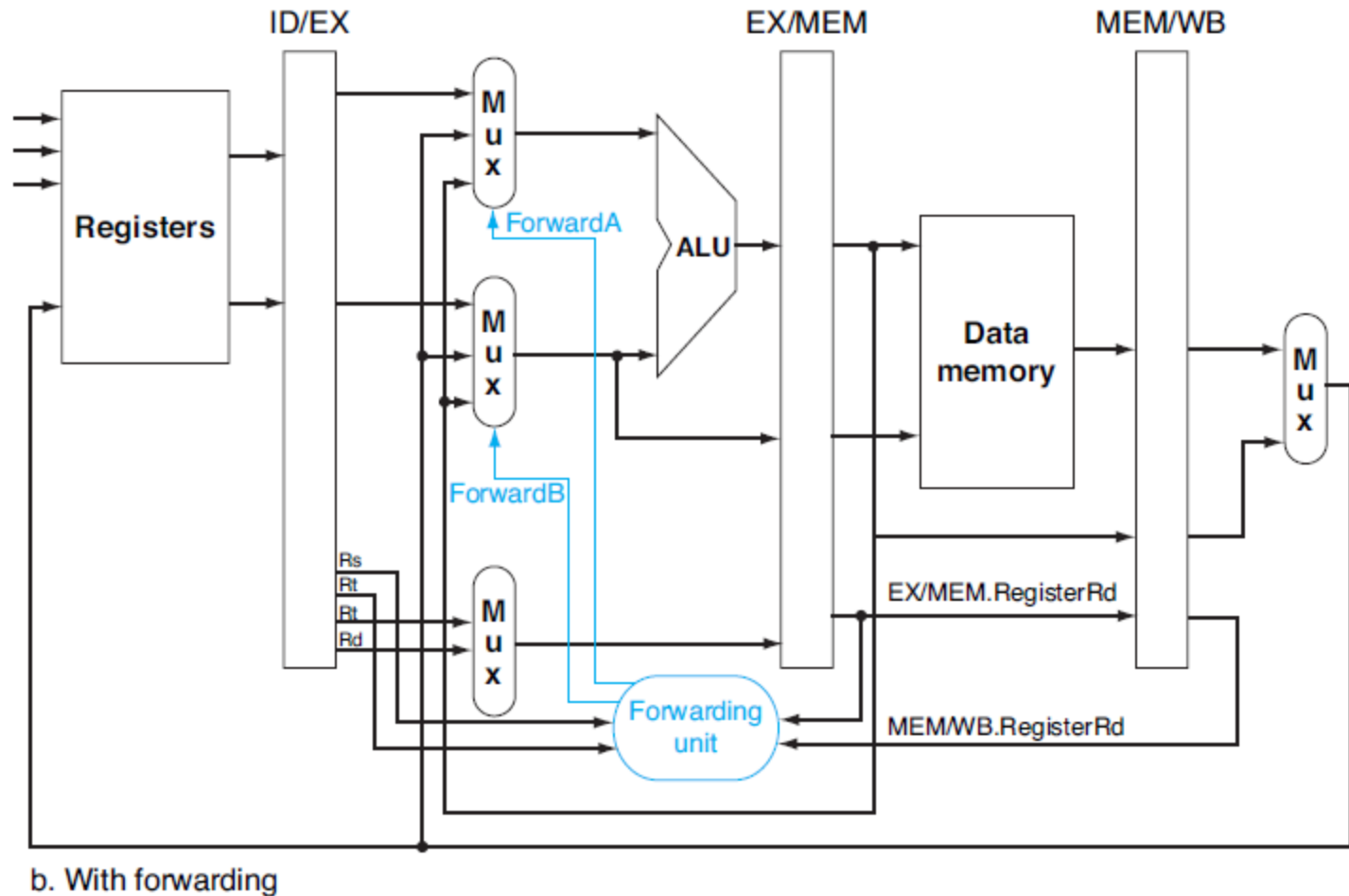
**No Hazard: Sub-add**

**No Hazard: Sub-sw**

# Writing the Forwarding Unit

- Hazard can occur with the immediately previous instruction called **EX Hazard**
- Hazard can occur with the instruction preceding the immediately previous instruction called **MEM Hazard**
- We need to design Forwarding unit such that the accurate data is forwarded to the ALU input. Forwarding unit must decide when to pass
  1. Register value read in the previous cycle
  2. ALU result of the previous instruction
  3. ALU result of the instruction previous than the last one

# Dependence Detection and Forwarding Unit



# Writing the Forwarding Unit

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.

- **We need to write separate Test-conditions inside the forwarding unit for EX and MEM hazards**

# Writing the Forwarding Unit Cont..

## ■ EX-Hazard Detection Test

```
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 Detection Test – version 1

```
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
```

# Writing the Forwarding Unit Cont...

- **Conditional Forwarding from MEM stage**
- **Consider the forwarding code**
  - `add $1,$1,$2`
  - `add $1,$1,$3`
  - `add $1,$1,$4`
- **Forwarding for instruction 3 above should be done from instruction 1 or instruction 2?**
  - Instruction 2 i.e. MEM stage rather than the WB stage of instruction 1
  - So.. Modify the MEM hazard conditions accordingly from the previous slide

# Writing the Forwarding Unit Cont...

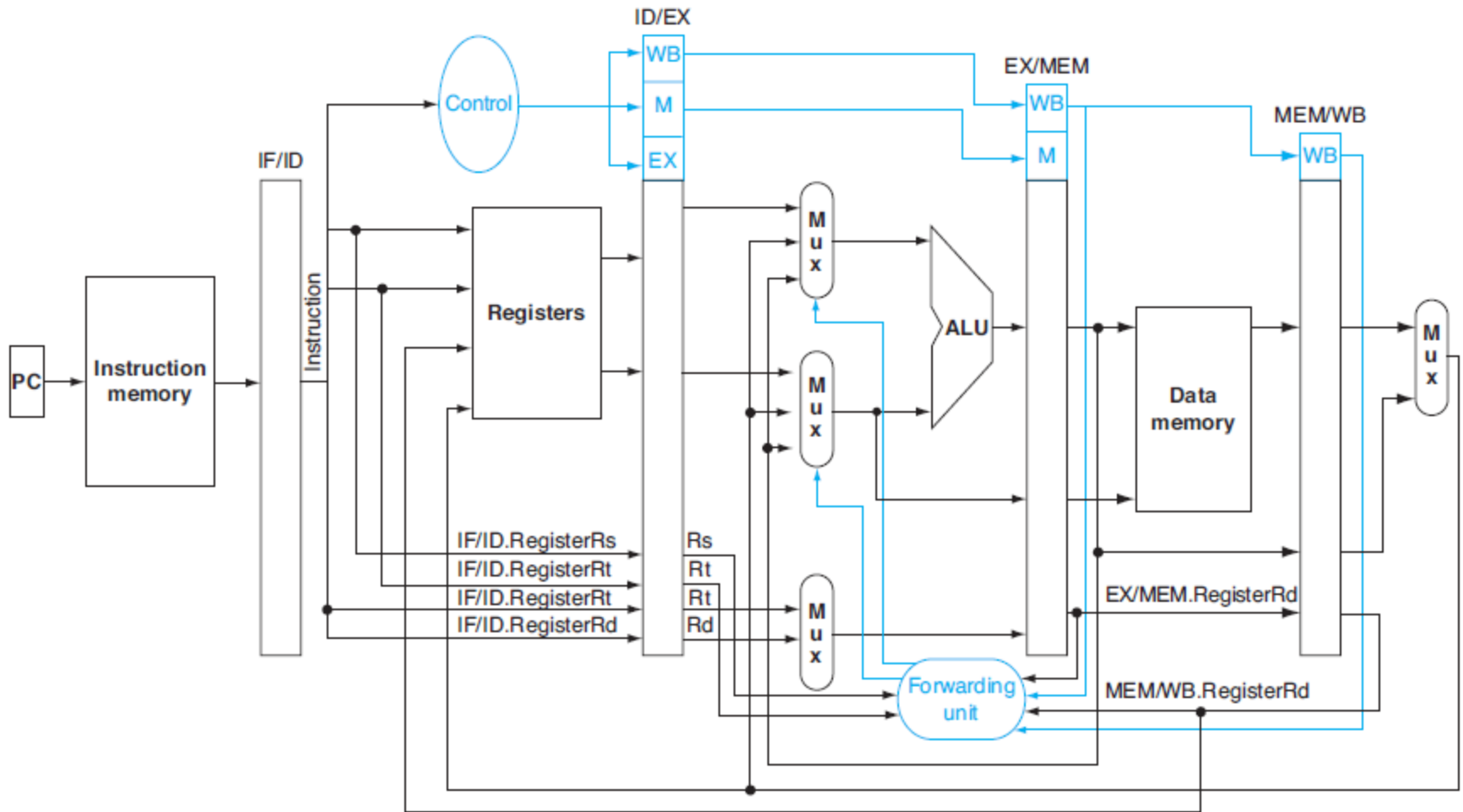
- **MEM Hazard Detection Test – Version 2**

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

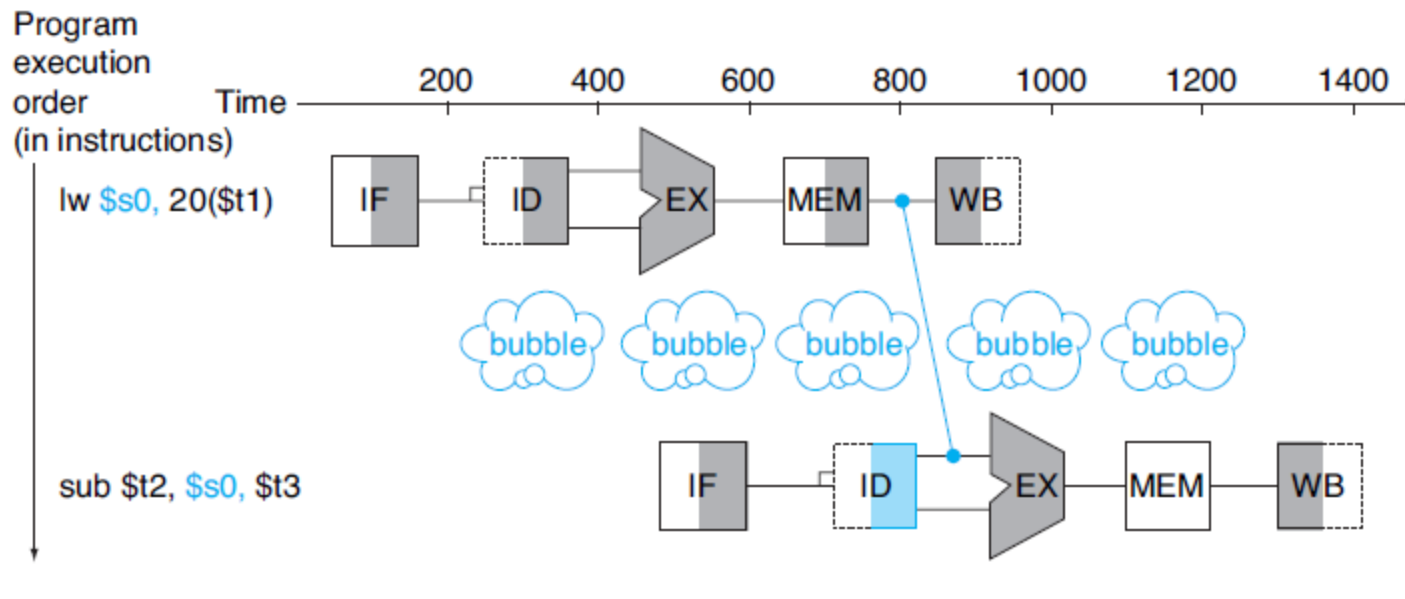
```
if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd  $\neq$  0)  
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0))  
    and (EX/MEM.RegisterRd  $\neq$  ID/EX.RegisterRt)  
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```



# Modified Data Path



# Recall Load-use data hazard



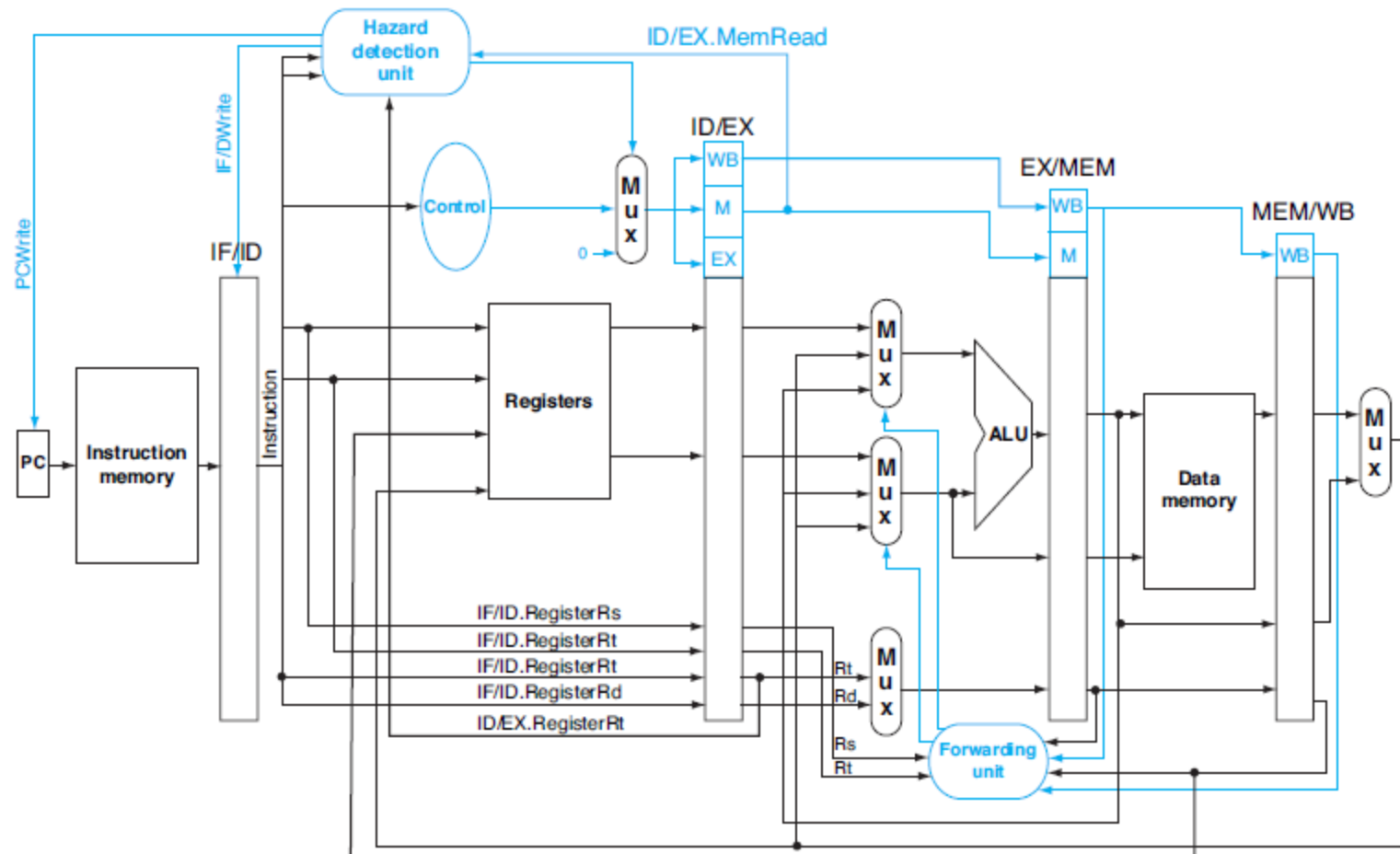
# Inserting a Stall in the Pipeline

- Two steps again
  - Hazard Detection
  - Stalling the pipeline if forwarding not possible
- Load-use hazard detection Test

```
if (ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
    stall the pipeline
```

- If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled; otherwise, we would lose the fetched instruction
- Preventing these two instructions from making progress is accomplished simply by preventing the PC register and the IF/ID pipeline register from changing
- **Base Line: Deasserting all nine control signals (setting them to 0) in the EX, MEM, and WB stages will create a “do nothing” or nop(NO OPERATION) instruction.**

# Inserting a Stall in the Pipeline Cont...

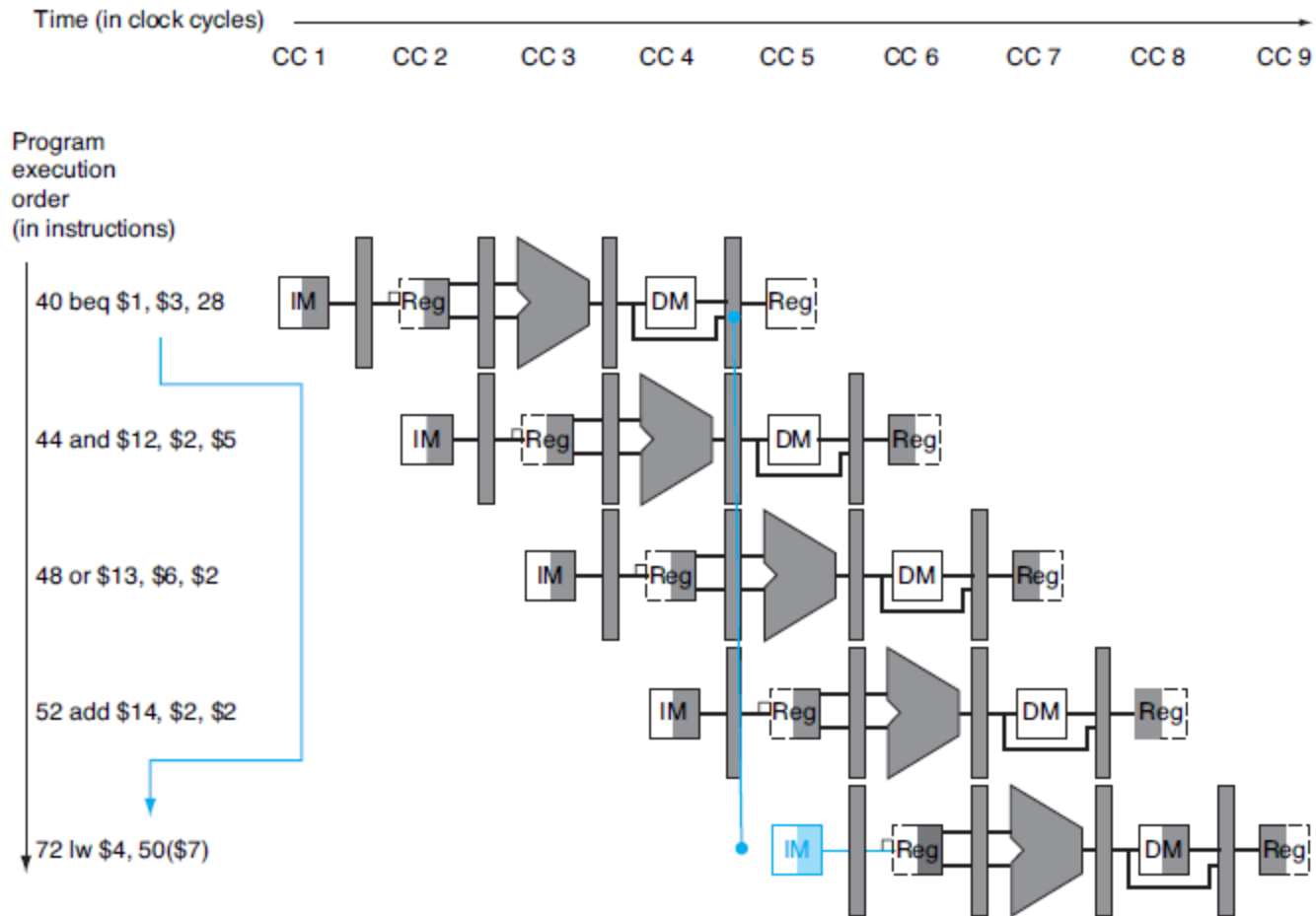


- The hazard detection unit controls the writing of the PC and IF/ID registers plus the multiplexor that chooses between the real control values and all 0s.

# Control Hazards

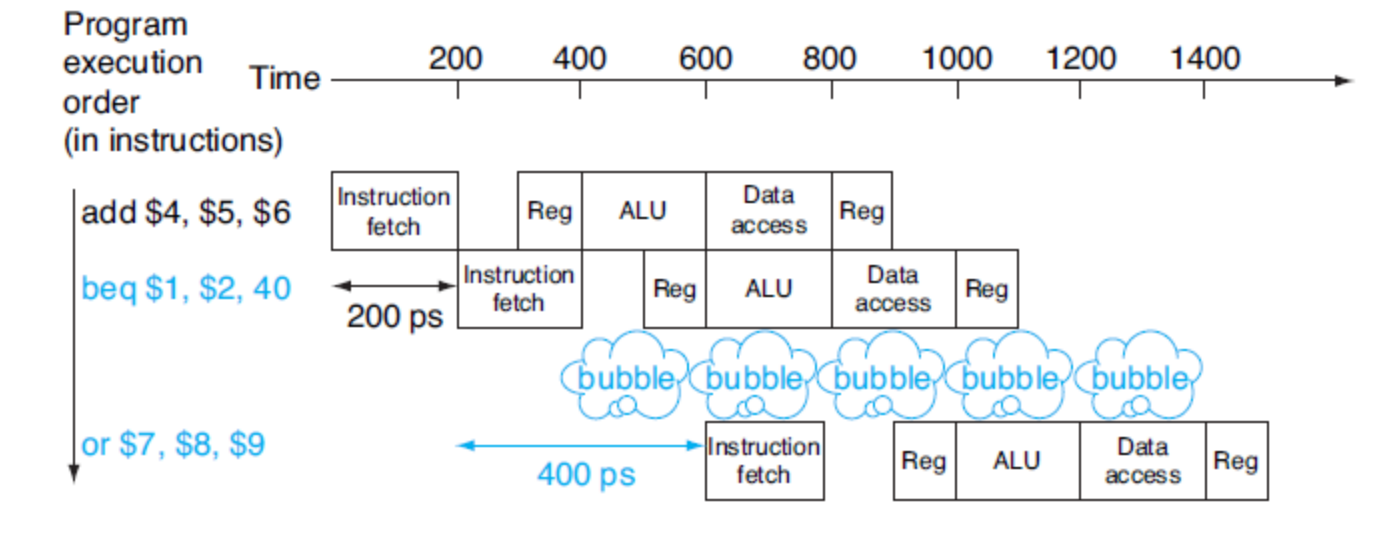
- Which instruction should be fetched after fetching the branch instruction?
- Branch decision is unknown in the next cycle
  - The pipeline cannot possibly know what the next instruction should be, since it *only just received the branch instruction from memory*

# Control Hazards Cont...



# Solution 1

- ***Solution 1: Moving the branch related hardware to ID stage***
  - ***One stall is still required i.e. the next instruction should not be fetched when the branch instruction is being decoded***



# Solution 1 Cont...

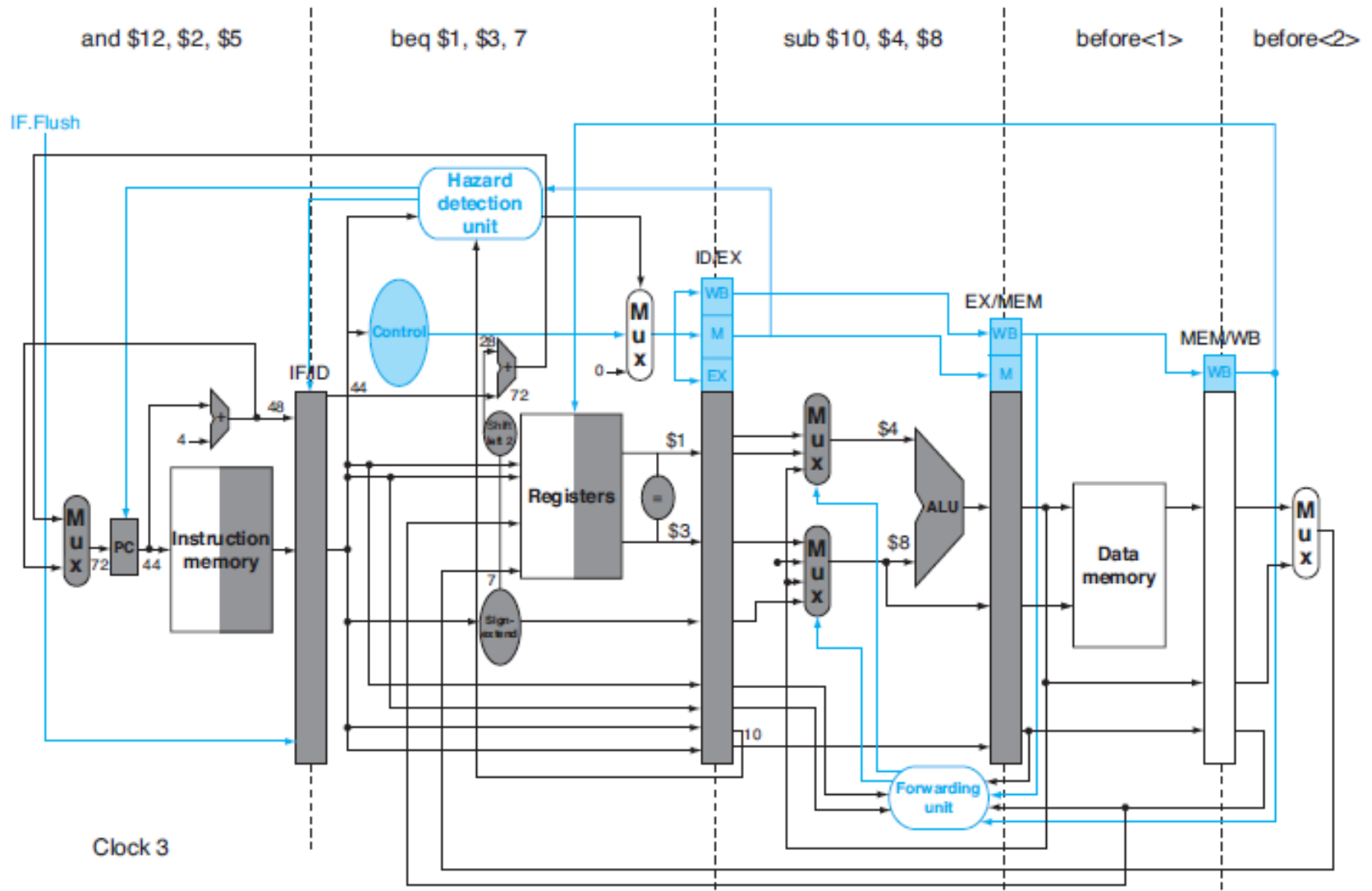
- **Moving the branch decision up requires two actions to occur earlier**
  - 1. Computing the branch target address**
    - **Move the branch adder from the EX stage to the ID stage**
  - 2. Evaluating the branch decision**
    - **Complex implementation**
      - **Forwarding and data hazard detection hardware required**



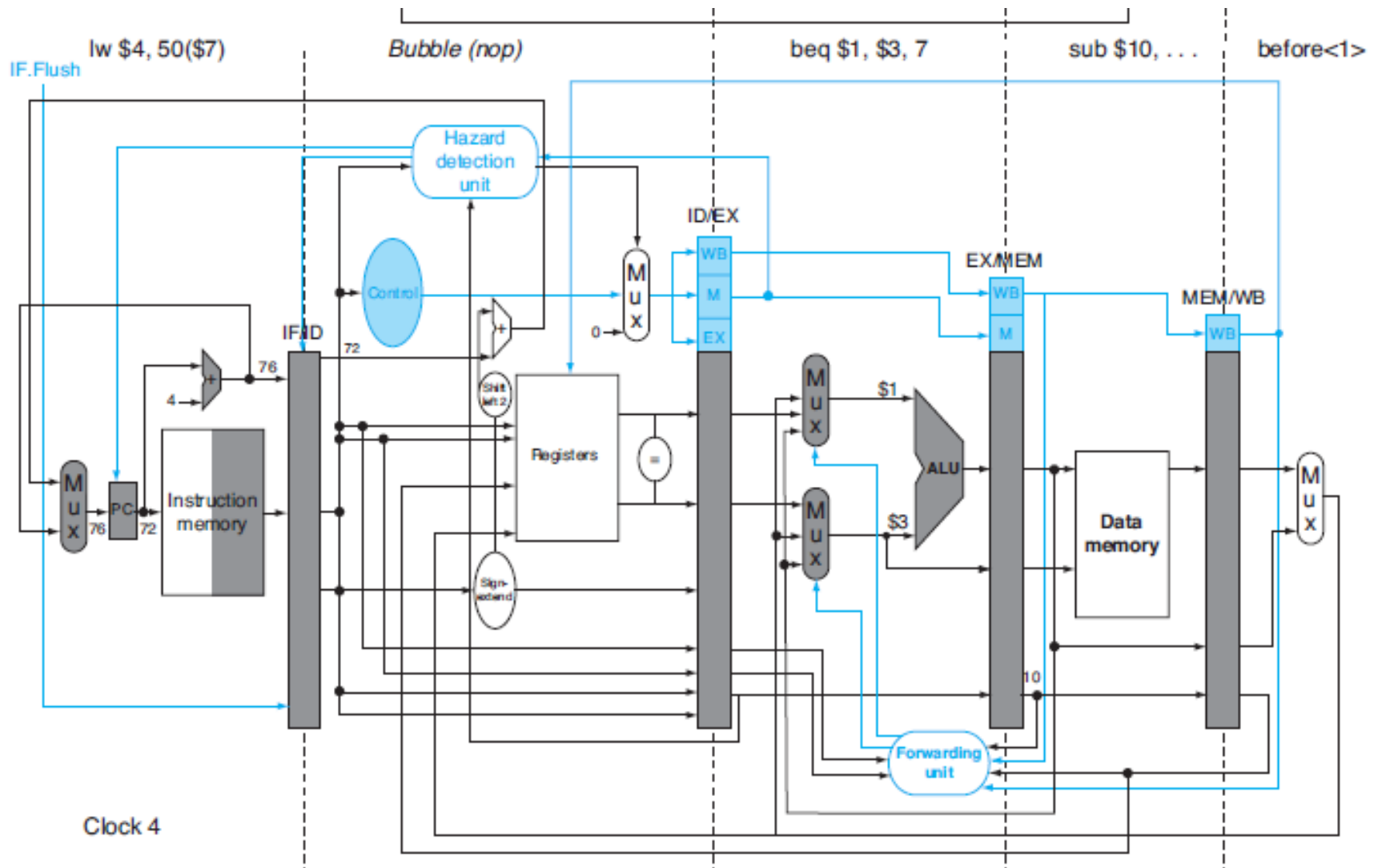
# Solution 1 Cont...

- During ID, we need complete the equality comparison so that if the instruction is a branch, we can set the PC to the branch target address.
- Operands required during the equality comparison may depend on the previous instructions executing in the other stages of the pipeline – Forwarding required again
  - Introduction of new forwarding logic
  - Operands of a branch can come from either the ALU/MEM or MEM/WB pipeline registers.
- Data hazard can occur and a stall might be required
  - R-Type instruction followed by the depending branch instruction would require 1 more stall
  - Load followed by the depending branch instruction would require 2 more stalls

# Solution 1 Cont...



# Solution 1 Cont...



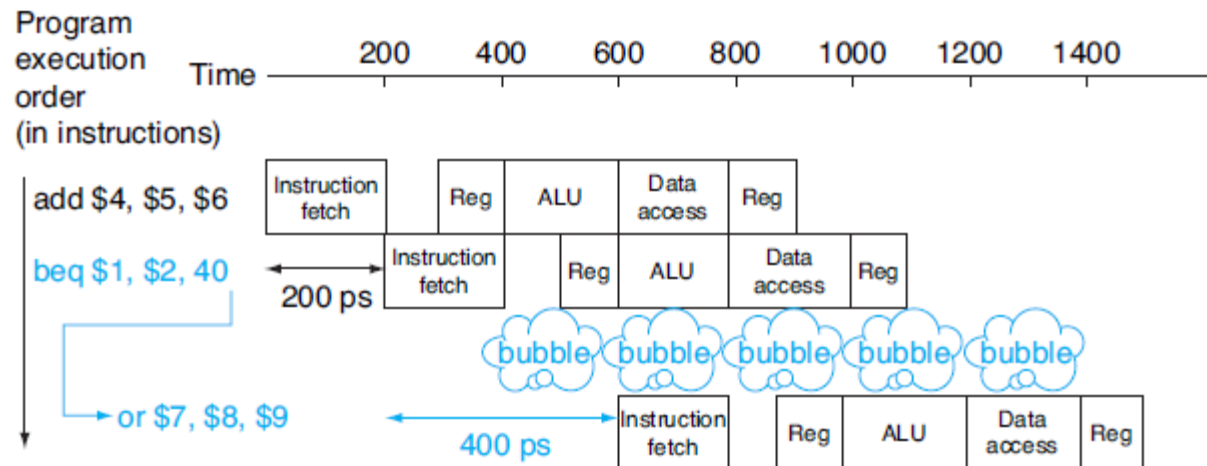
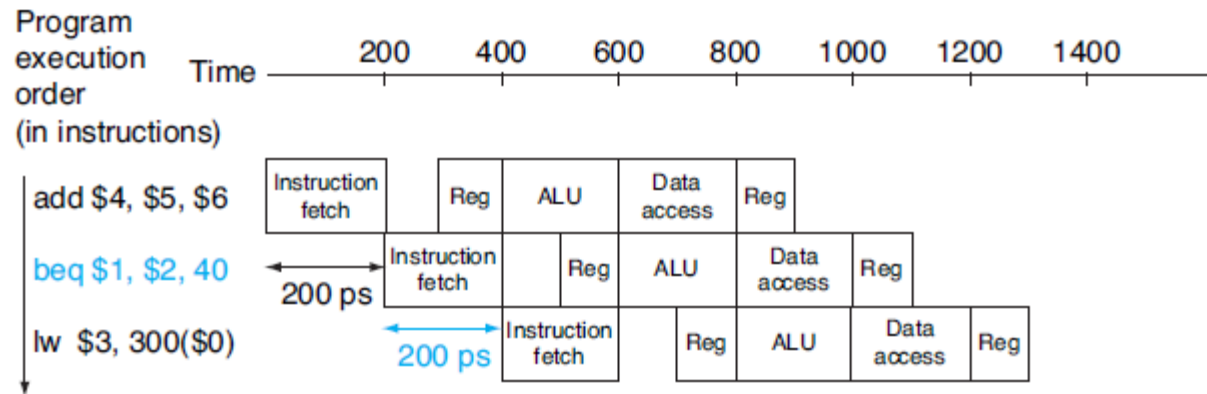
# Solution 2

- **Solution 2: Predict the branch decision Statically**

*Predict always that branches will be untaken. When you're right, the pipeline proceeds at full speed. Only when branches are taken does the pipeline stall.*

- If branches are untaken half the time this optimization halves the cost of control hazards
- *Improvement? Take predictions for some branches as taken and for other as untaken*
- *Example: At the bottom of loops are branches that jump back to the top of the loop. Since they are likely to be taken and they branch backward, we could always predict taken for branches that jump to an earlier address.*

# Solution 2 Cont...



# Solution 3

- **Solution 3: Dynamic Hardware predictors**

*Make their guesses depending on the behavior of each branch and may change predictions for a branch over the life of a program*

- Keep a history for each branch as taken or untaken, and then use the recent past behavior to predict the future
- Branch predictors have approximately 90% accuracy when amount and type of history becomes extensive

# Solution 3 Cont...

- **Dynamic branch predictor**
  - **1-bit branch prediction buffer:** Contains 1 bit indicating whether the branch was taken last time or not
  - **2-bit branch prediction buffer:** prediction must be wrong twice before it is changed
  - **Correlating predictors:** both a local branch, and the global behavior of recently executed branches together yields greater prediction accuracy
  - **Tournament predictors:** Use multiple predictors, tracking, for each branch, which predictor yields the best results. Select can operate favoring whichever of the predictors has been more accurate

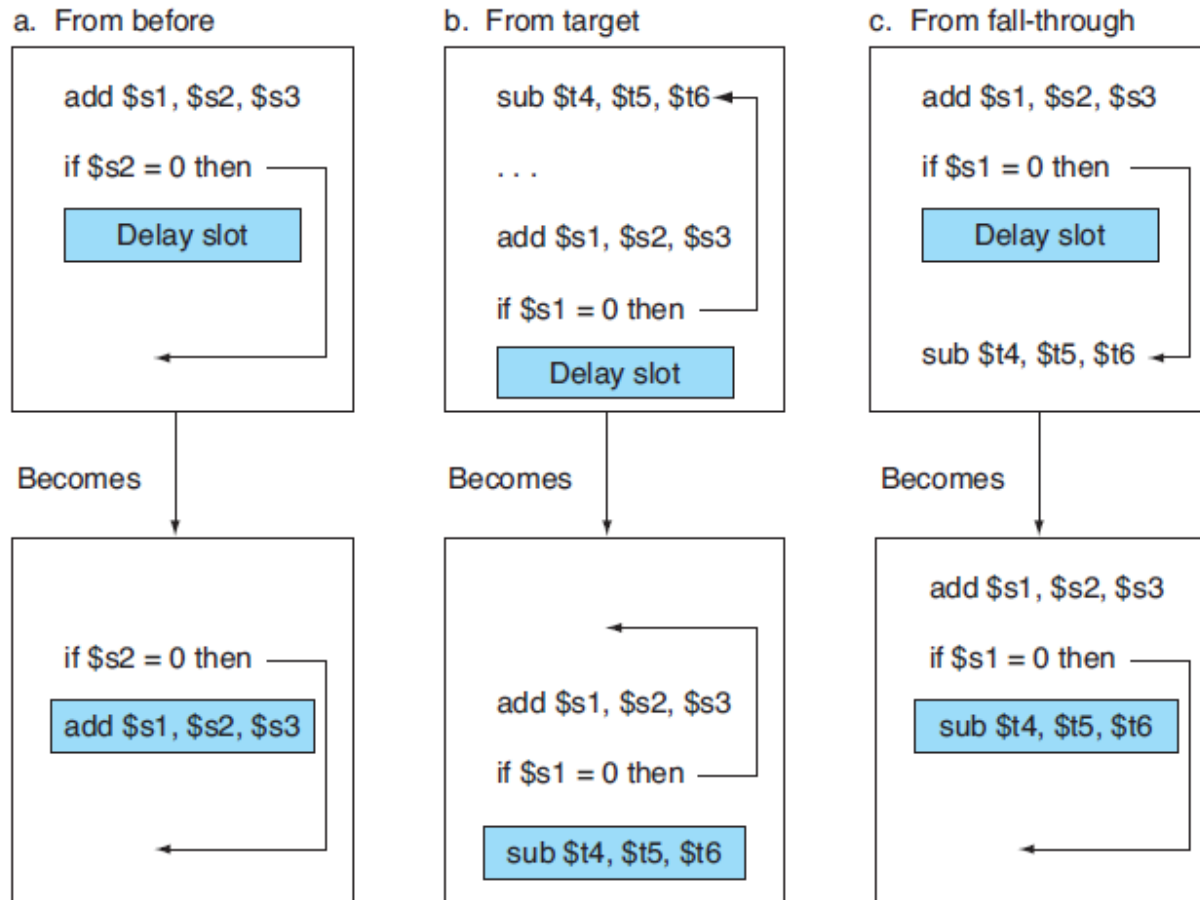
# Solution 3 Cont...

- **Branch prediction buffer can be accessed with the instruction address during the IF pipe stage.**
  - **If the instruction is predicted as taken, fetching begins from the target as soon as the PC is known after the ID stage**
  - **Else sequential fetching continues.**



# Solution 4

- **Delayed Decision:** Compilers try to place an instruction after the branch instruction which is independent of the branch decisions
  - Code reordering - Same as we discussed before



# Exceptions - “Stuff Happens”

- **Exceptions** definition: “*unexpected change in control flow*”
- Another form of control hazard.

For example:

```
add R1, R2, R1;      causing an arithmetic overflow
sw  R3, 400(R1);
add R5, R1, R2;
```

*Invalid R1 contaminates other registers or memory locations!*

# What happens during an exception

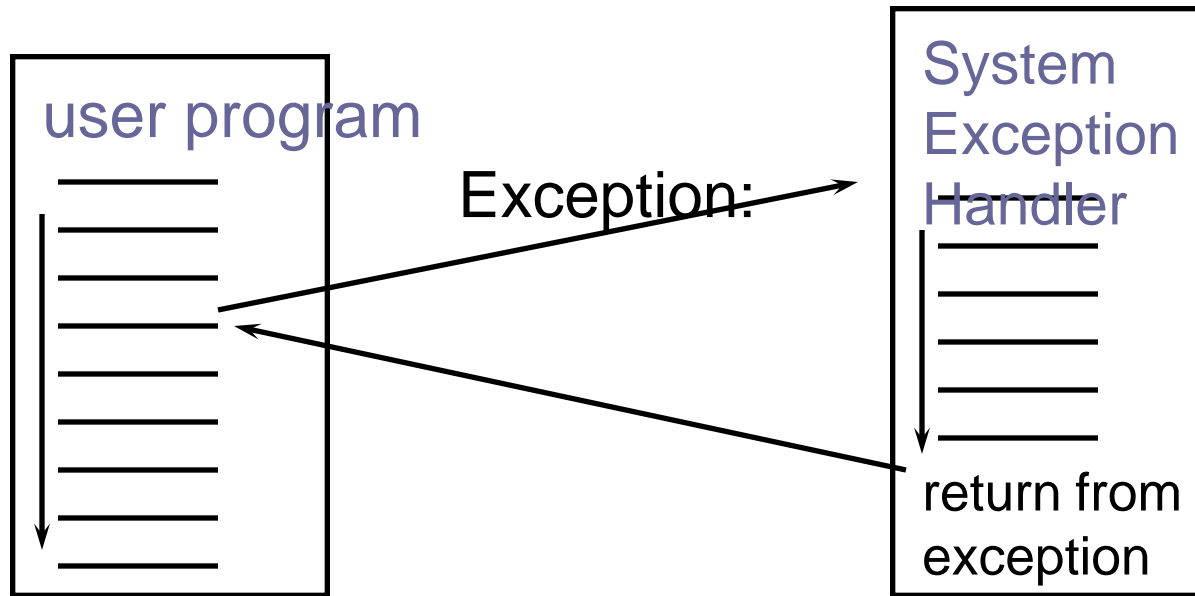
## In The Hardware

- The pipeline has to
  - 1) stop executing the **offending instruction** in midstream,
  - 2) let all **preceding instructions** complete,
  - 3) flush all **succeeding instructions**,
  - 4) set a register to show the cause of the exception,
  - 5) save the address of the offending instruction, and
  - 6) then jump to a prearranged address (the address of the exception handler code)

## In The Software

- The software (OS) looks at the cause of the exception and “deals” with it
- Normally OS kills the program

# Exceptions



## Exception = non-programmed control transfer

- system takes action to handle the exception
  - must record the address of the offending instruction
  - record any other information necessary to return afterwards
- returns control to user
- must save & restore user state

# Additions to MIPS ISA to support Exceptions

- **EPC** (Exceptional Program Counter)
  - A 32-bit register
  - Hold the address of the offending instruction
- **Cause**
  - A 32-bit register in MIPS (some bits are unused currently.)
  - Record the cause of the exception
- **Status** - interrupt mask and enable bits and determines what exceptions can occur.
- Control signals to write EPC , Cause, and Status
- Be able to write exception address into PC, increase mux set PC to exception address (MIPS uses 8000 00180<sub>hex</sub> ).
- May have to undo  $PC = PC + 4$ , since want EPC to point to offending instruction (not its successor);  $PC = PC - 4$
- What else?

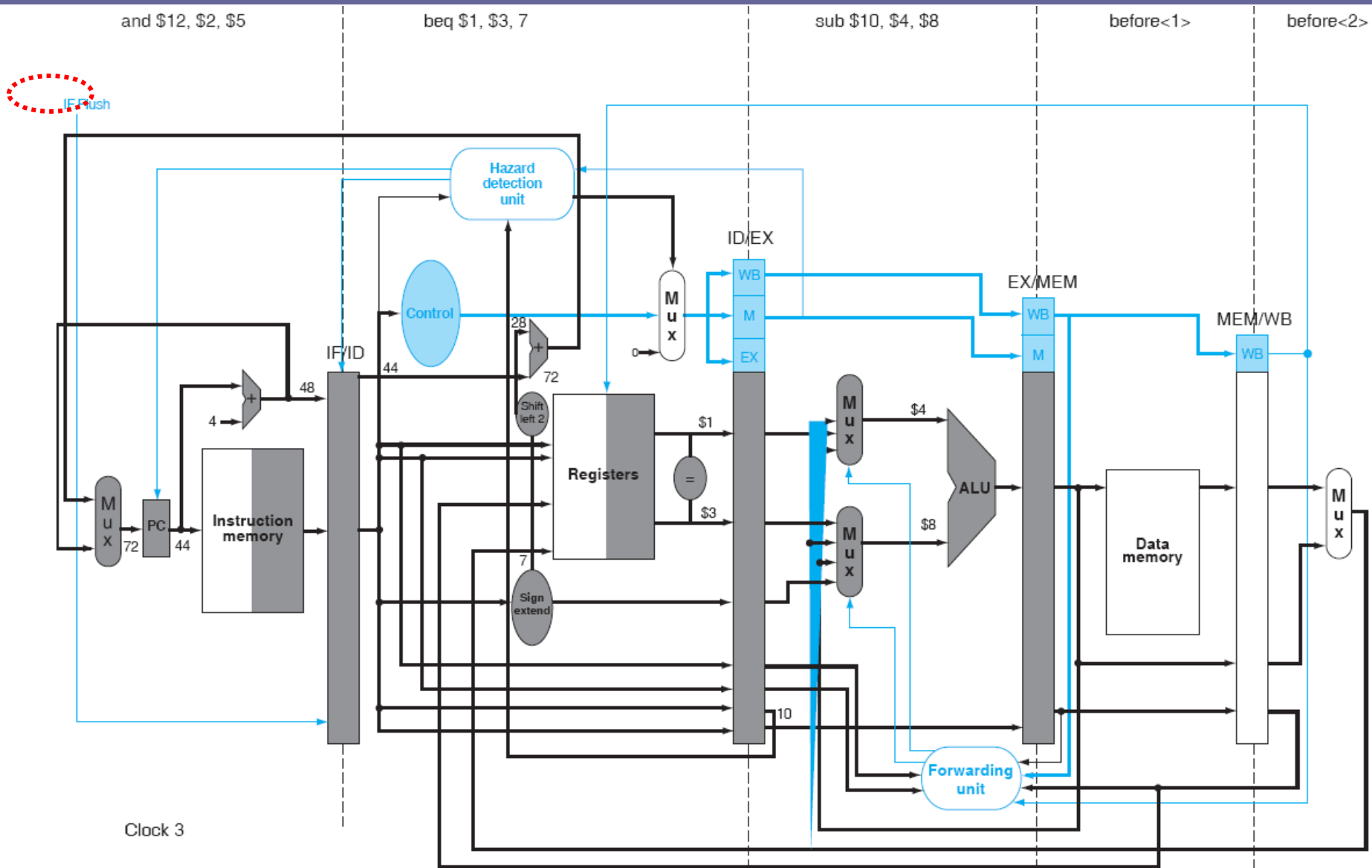
flush all succeeding instructions in pipeline

# Flush instructions in Branch Hazard

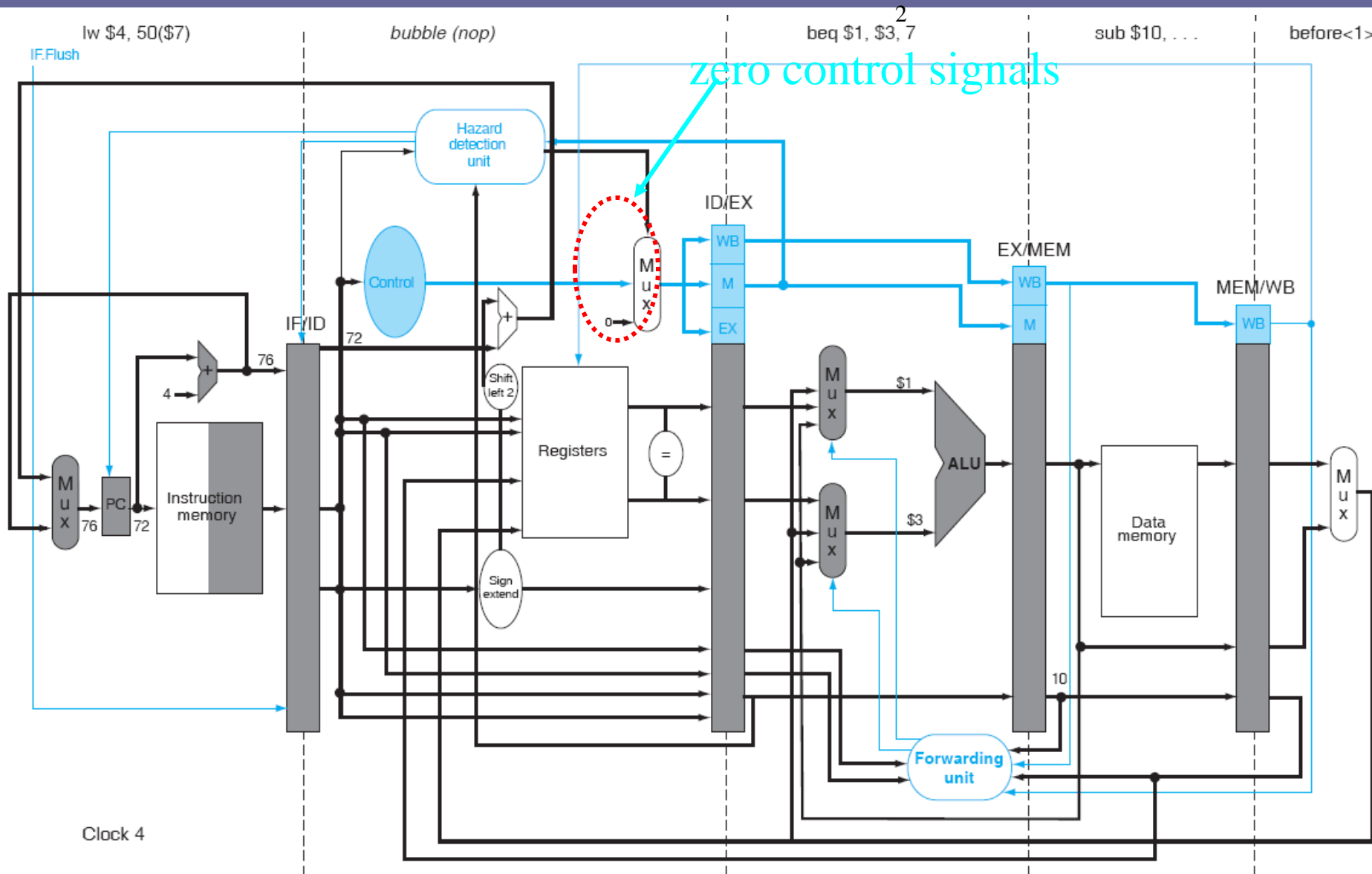
```
36 sub    $10,    $4, $8
40 beq$1, $3, 7 # taget = 40 + 4 + 7*4 = 72
44 and    $12,    $2, $5
48 or     $13,    $2, $6
52 ....

....
72 lw     $4, 50($7)
```

# Flush instructions at IF stage in Branch Hazard



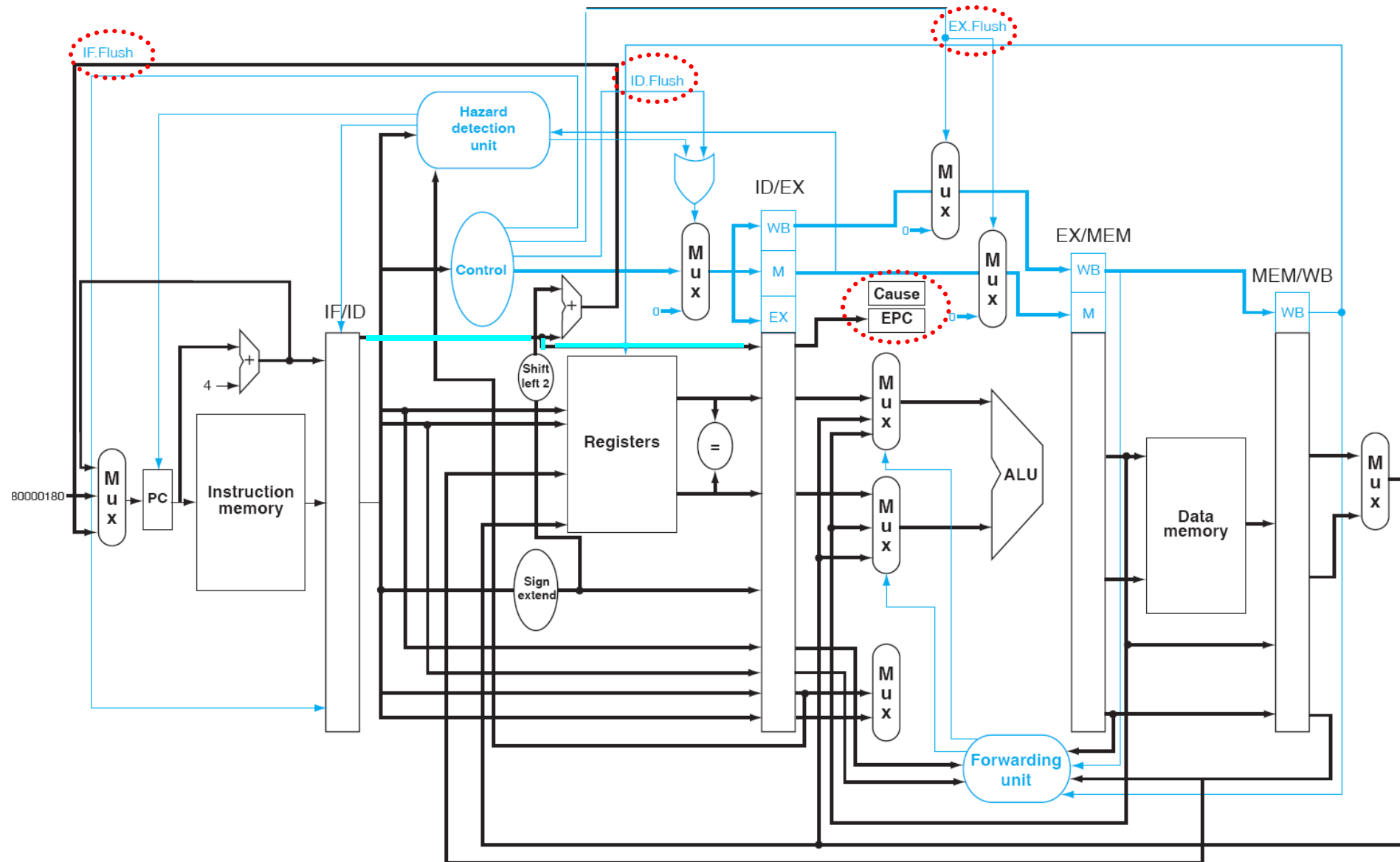
# Flush instructions at IF stage in Branch Hazard



Clock 4



# Additions to MIPS ISA to support Exceptions



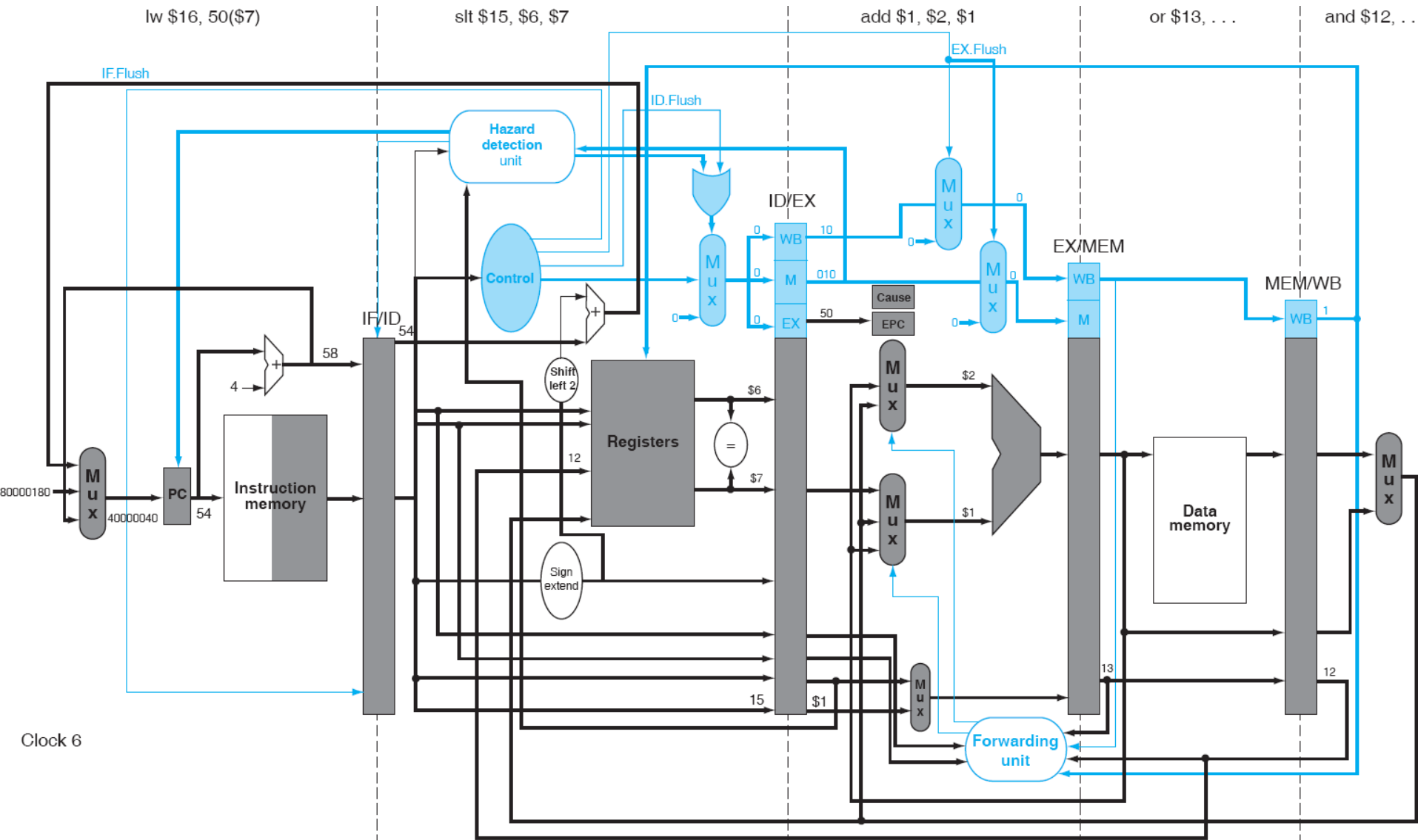
# Exceptions Example

```
40hex    sub    $11,    $2, $4
44hex    and$12,    $2, $5
48hex    or   $13,    $2, $6
4Chex    add$1, $2, $1; // arithmetic overflow
50hex    stl   $15,    %6,$7
54hex    lw   $16,    50($7)
```

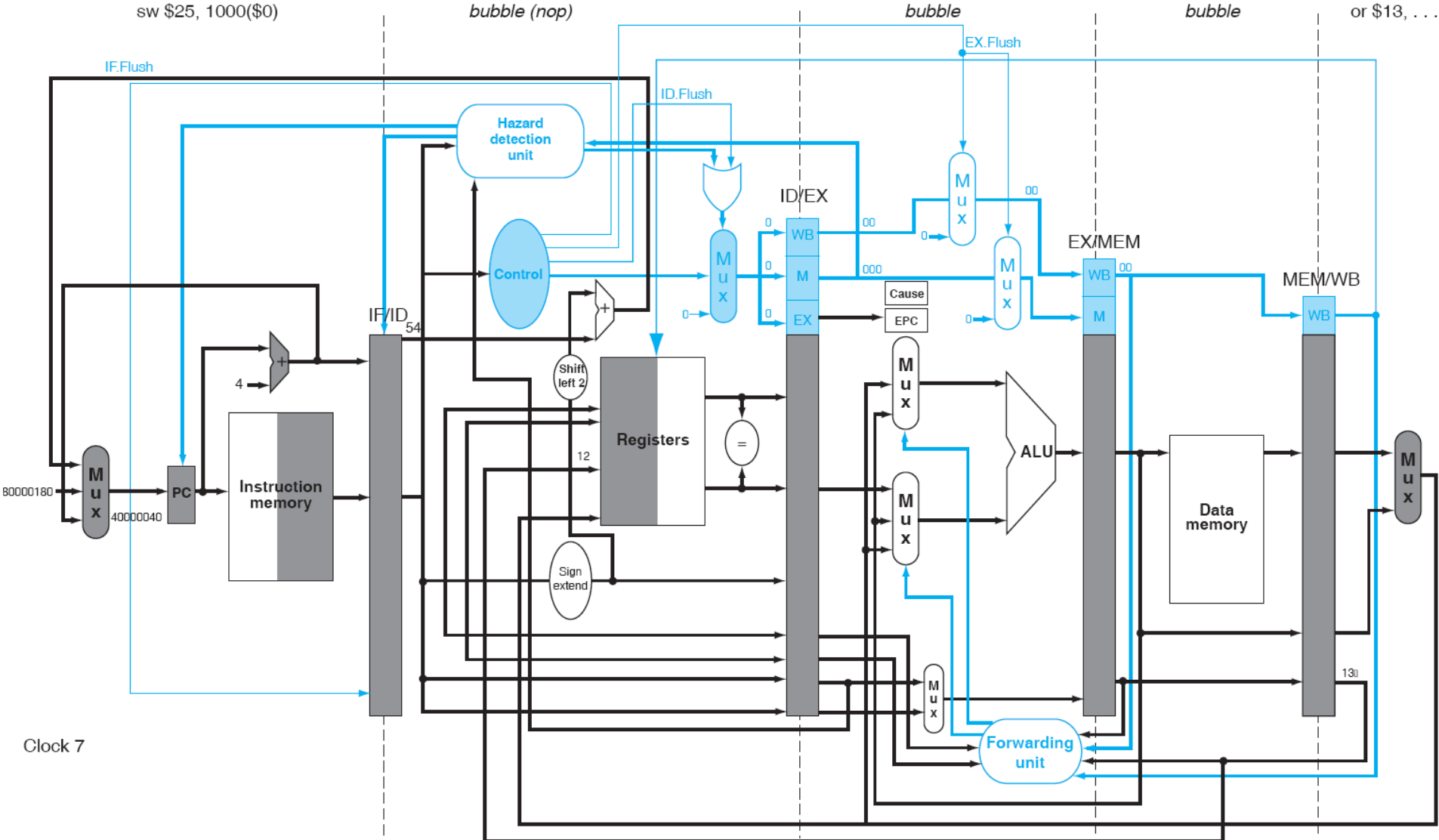
Exception handling program:

```
80000180hex  sw    $25,    1000($0)
80000184hex  sw    $12,    1004($0)
```

# Exceptions Example



# Exceptions Example



# Reordering code to avoid stalls

- **Code for C language:**

**a = b + e;**

**c = b + f;**

- **Corresponding Assembly Language code:**

**lw \$t1, 0(\$t0)**

**lw \$t2, 4(\$t0)**

**add \$t3, \$t1,\$t2**

**sw \$t3, 12(\$t0)**

**lw \$t4, 8(\$t0)**

**add \$t5, \$t1,\$t4**

**sw \$t5, 16(\$t0)**

- **Find the hazards in the preceding code segment and reorder the instructions to avoid any pipeline stalls.**

# Reordering code to avoid stalls Cont...

- **Solution**

**lw \$t1, 0(\$t0)**

**lw \$t2, 4(\$t0)**

**lw \$t4, 8(\$t0)**

**add \$t3, \$t1,\$t2**

**sw \$t3, 12(\$t0)**

**add \$t5, \$t1,\$t4**

**sw \$t5, 16(\$t0)**

# Example

- Suppose Forwarding has not been implemented
- add R5, R2, R1
- ld R3, 44(R5)
- ld R2, 40(R2)
- or R3, R5, R3
- sw R3, 0(R5)

# Example

- Suppose Forwarding has not been implemented
- add R5, R2, R1
- ld R2, 40(R2)
- Nop
- ld R3, 44(R5)
- Nop
- Nop
- or R3, R5, R3
- Nop
- Nop
- sw R3, 0(R5)



# Example

- **Loop:**

- 1. ld R4, 0(R1)**
- 2. mul R4, R4, R2**
- 3. ld R5, 0(R0)**
- 4. add R4, R4, R5**
- 5. sw R4, 0(R0)**
- 6. sub R3, R3, 1**
- 7. bne R3, 0, Loop**

# Example

- Suppose Forwarding has not been implemented
- `ld R1, 40(R6)`
- `add R2, R3, R1`
- `add R1, R6, R4`
- `sw R2, 20(R4)`
- `and R1, R1, R4`