CCSYA

The Processor

Pipelining – Part II

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

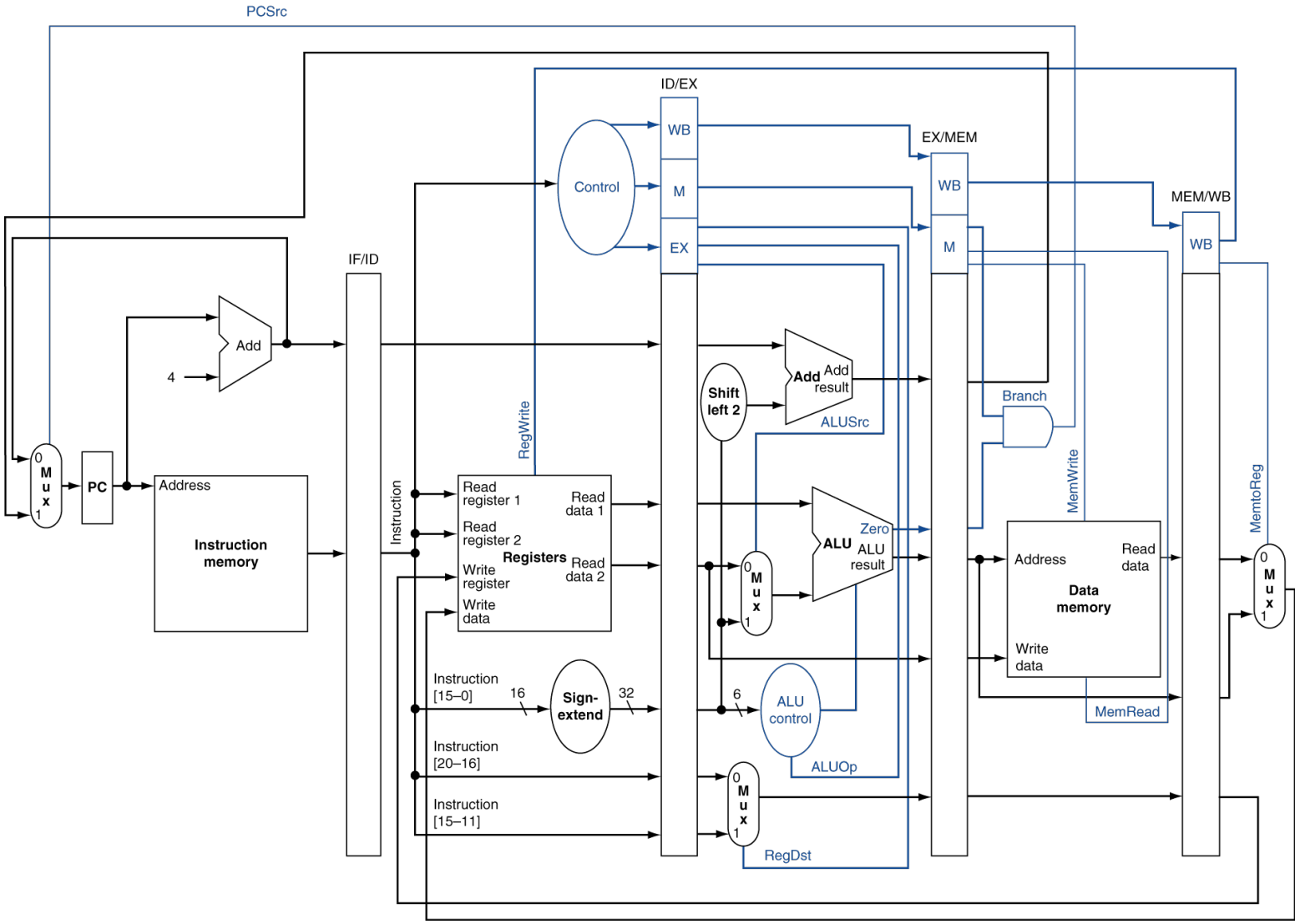
Luís Nogueira (lmn@isep.ipp.pt)

Pipeline Summary

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

- **Instruction set design affects complexity of pipeline implementation**
  - MIPS ISA designed for pipelining

Pipelined Datapath



Pipeline Hazards

- **There are situations when the next instruction cannot execute in the following clock cycle**
  - These events are called **hazards**
- **Structural hazard**
  - 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

Dealing with Hazards

- **Without intervention, pipeline must always be stalled**
  - Stalling the pipeline usually lets some instruction(s) proceed, another/others wait for data, resource, etc
- **Stalls impede progress of a pipeline and result in deviation from 1 instruction executing/clock cycle**
  - If no stalls, speedup equal to number of pipeline stages in ideal case
- **CPI pipelined = Ideal CPI + Pipeline stall cycles per instruction**
  - 1 + Pipeline stall cycles per instruction

Structural Hazards

- **Conflict for use of a resource**

- **If our MIPS pipeline had a single memory**
  - Load/store requires data access
  - Instruction fetch would have to stall for that cycle
    - Would cause a pipeline “bubble”

Clock Number

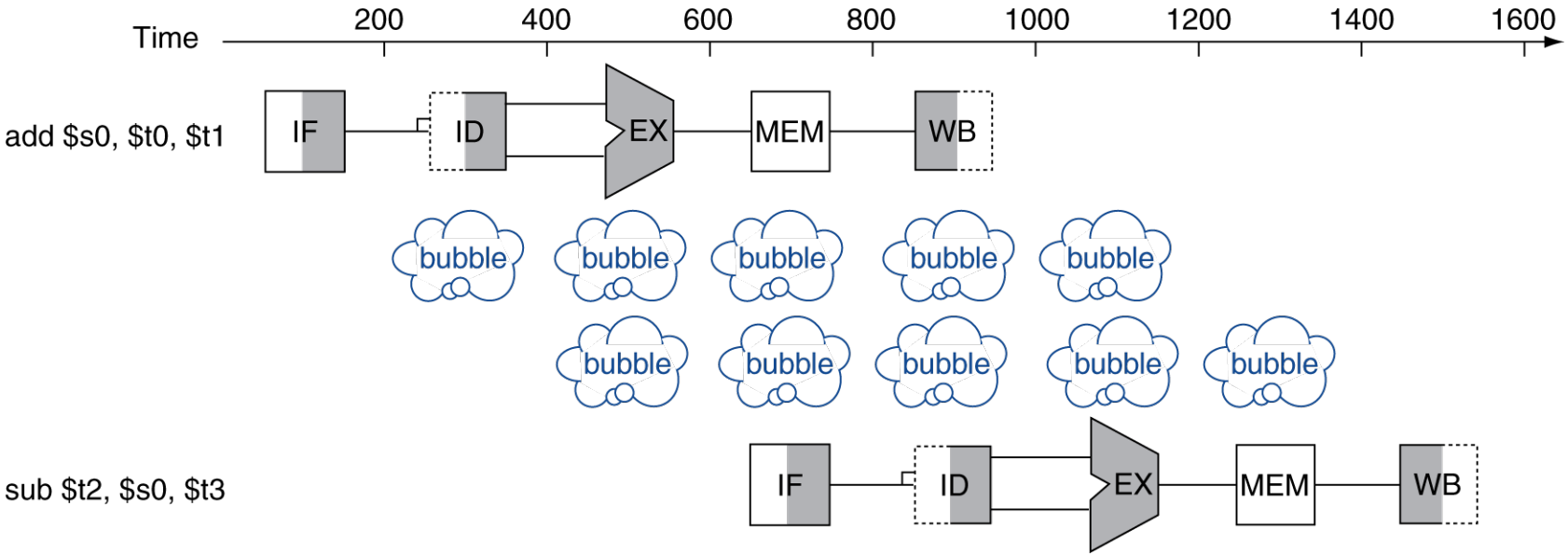
Inst. #	1	2	3	4	5	6	7	8	9	10
LOAD	IF	ID	EX	MEM	WB					
Inst. i+1		IF	ID	EX	MEM	WB				
Inst. i+2			IF	ID	EX	MEM	WB			
Inst. i+3				stall	IF	ID	EX	MEM	WB	
Inst. i+4						IF	ID	EX	MEM	WB
Inst. i+5							IF	ID	EX	MEM
Inst. i+6								IF	ID	EX

Structural Hazards

- **Solution is to replicate resource**
  - Otherwise, CPI degrades quickly from our ideal ‘1’ for even the simplest of cases
- **MIPS instruction set was designed to be pipelined**
  - Making it easy for designers to avoid structural hazards when designing a pipeline
- **Examples**
  - Separate instruction/data memories
  - Separate instruction/data caches
  - An ALU to perform an arithmetic operation and an adder to increment PC

Data Hazards

- **An instruction depends on completion of data access by a previous instruction**
  - `add  $s0, $t0, $t1`
    `sub  $t2, $s0, $t3`



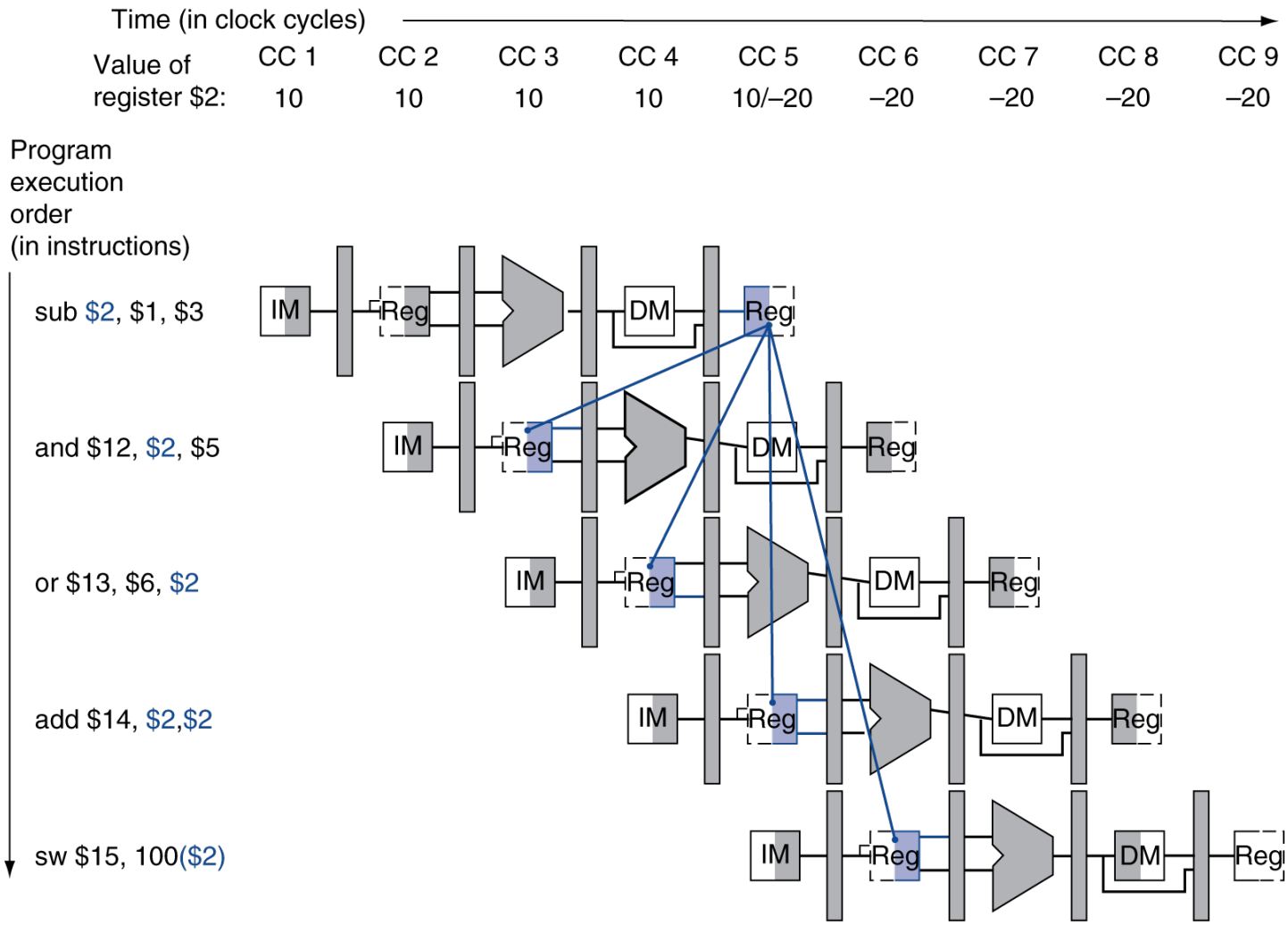
Data Hazards

- **Dependence of instructions**
  - When an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage

- **Let's look at a sequence with many dependences**

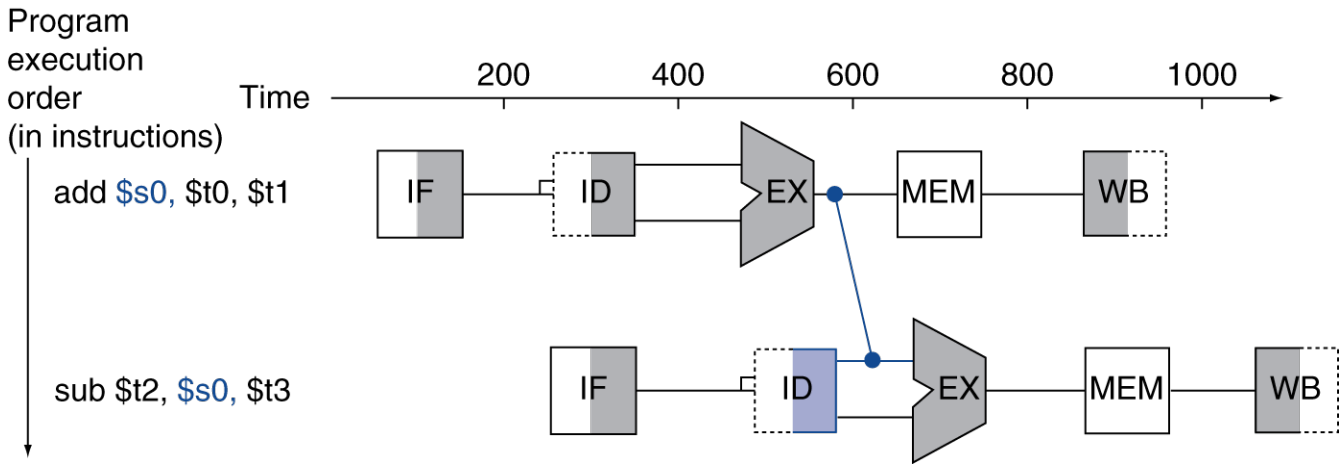
```
sub    $2, $1,$3          # Register $2 written by sub
and    $12,$2,$5          # 1st operand($2) depends on sub
or     $13,$6,$2          # 2nd operand($2) depends on sub
add    $14,$2,$2          # 1st($2) & 2nd($2) depend on sub
sw     $15,100($2)        # Base ($2) depends on sub
```

Data Hazards



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



Forwarding

- **Dependence is solved from a pipeline register, rather than waiting for the WB stage to write the register file**
  - Thus, the required data exists in time for later instructions, with the pipeline registers holding the data to be forwarded

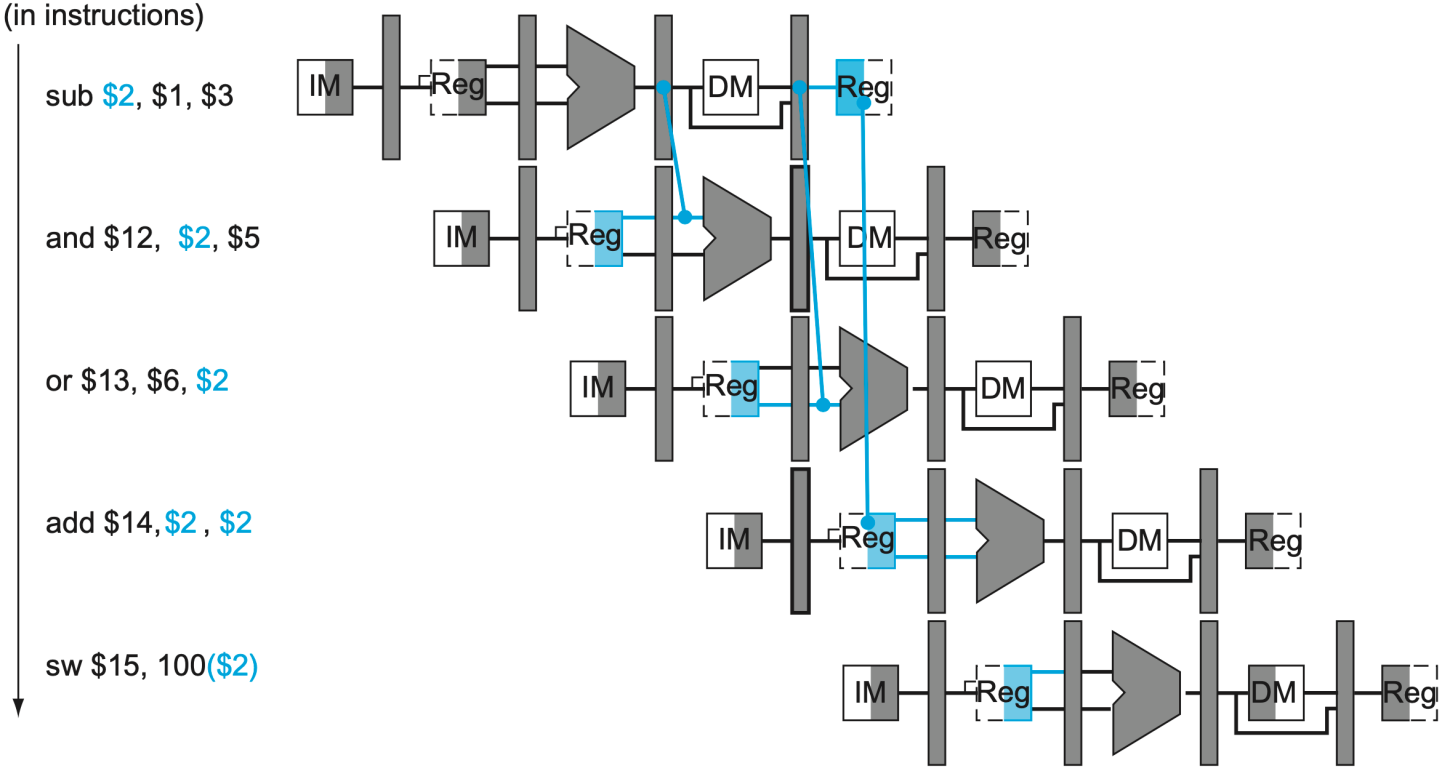
- **If we can take the inputs to the ALU from any pipeline register rather than just ID/EX, then we can forward the proper data**
  - By adding multiplexors to the input of the ALU, and with the proper controls, we can run the pipeline at full speed in the presence of these data dependences
  - This forwarding control will be in the EX stage, because the ALU forwarding multiplexors are found in that stage

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)



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

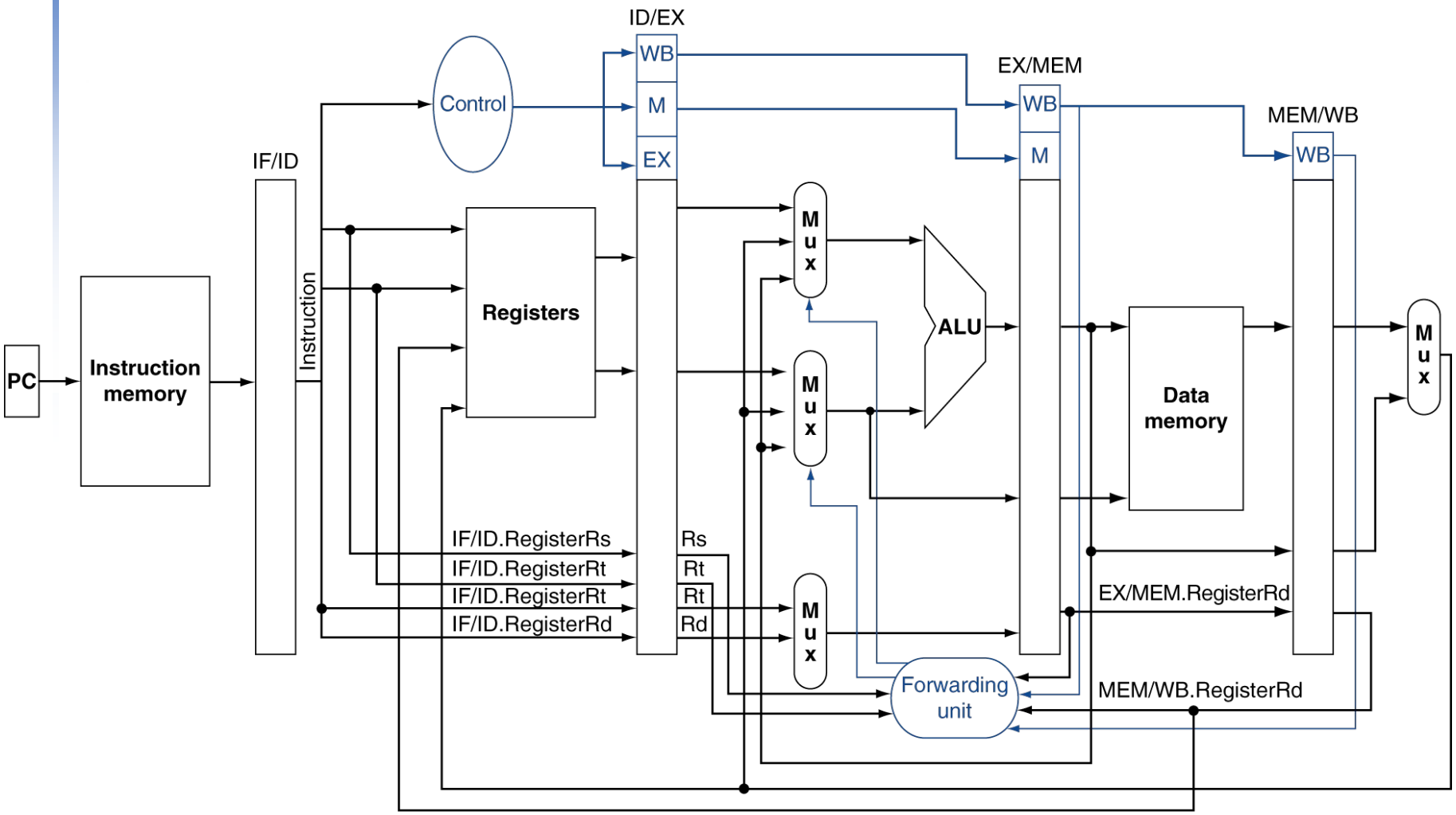
  → Forward from EX/MEM pipeline register
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

  → Forward from MEM/WB pipeline register

Detecting the Need to Forward

- **Some instructions do not write registers**
  - No need to forward!
- **One solution is simply to check to see if the RegWrite control signal is active**
  - Examining the WB control field of the pipeline register during the EX and MEM stages determines whether RegWrite is asserted
- **And only if Rd for that instruction is not $zero**
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0

Datapath with Forwarding



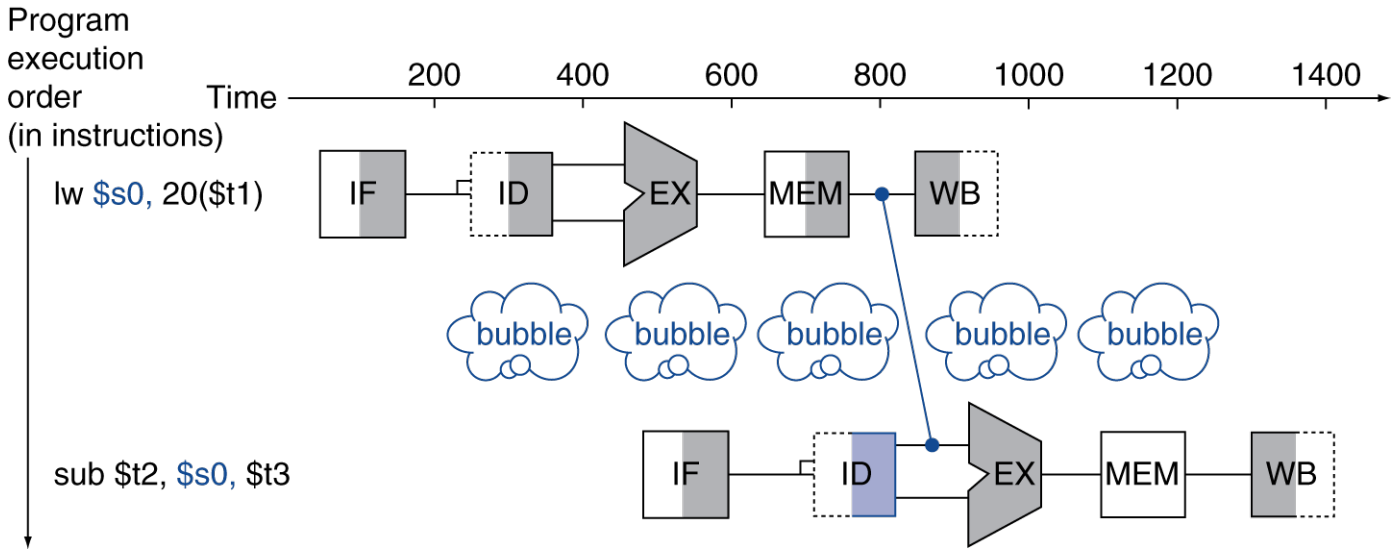
Forwarding Conditions

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

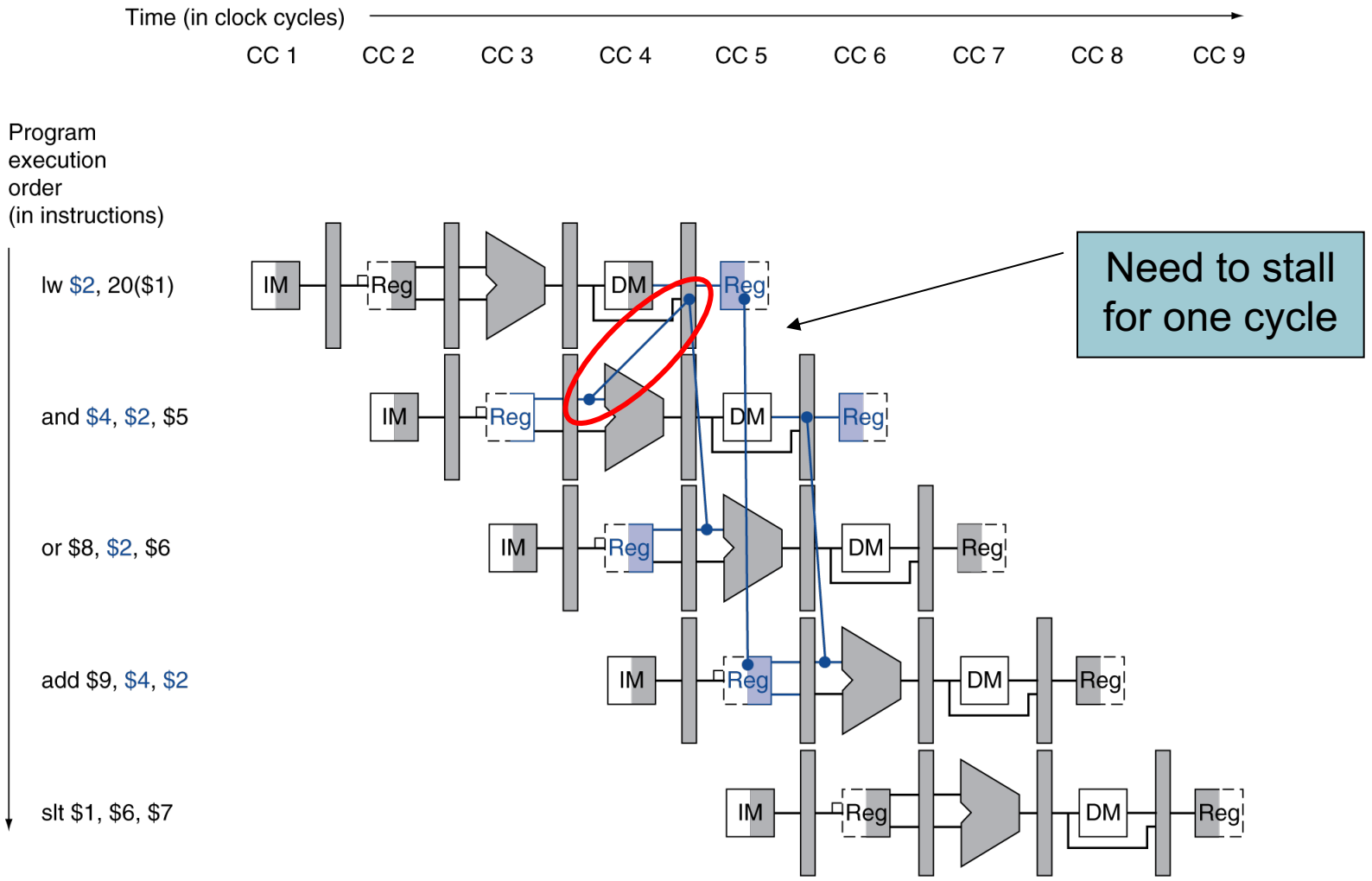
Load-Use Data Hazard

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

- **Load has a latency that forwarding can't solve**



Load-Use Data Hazard



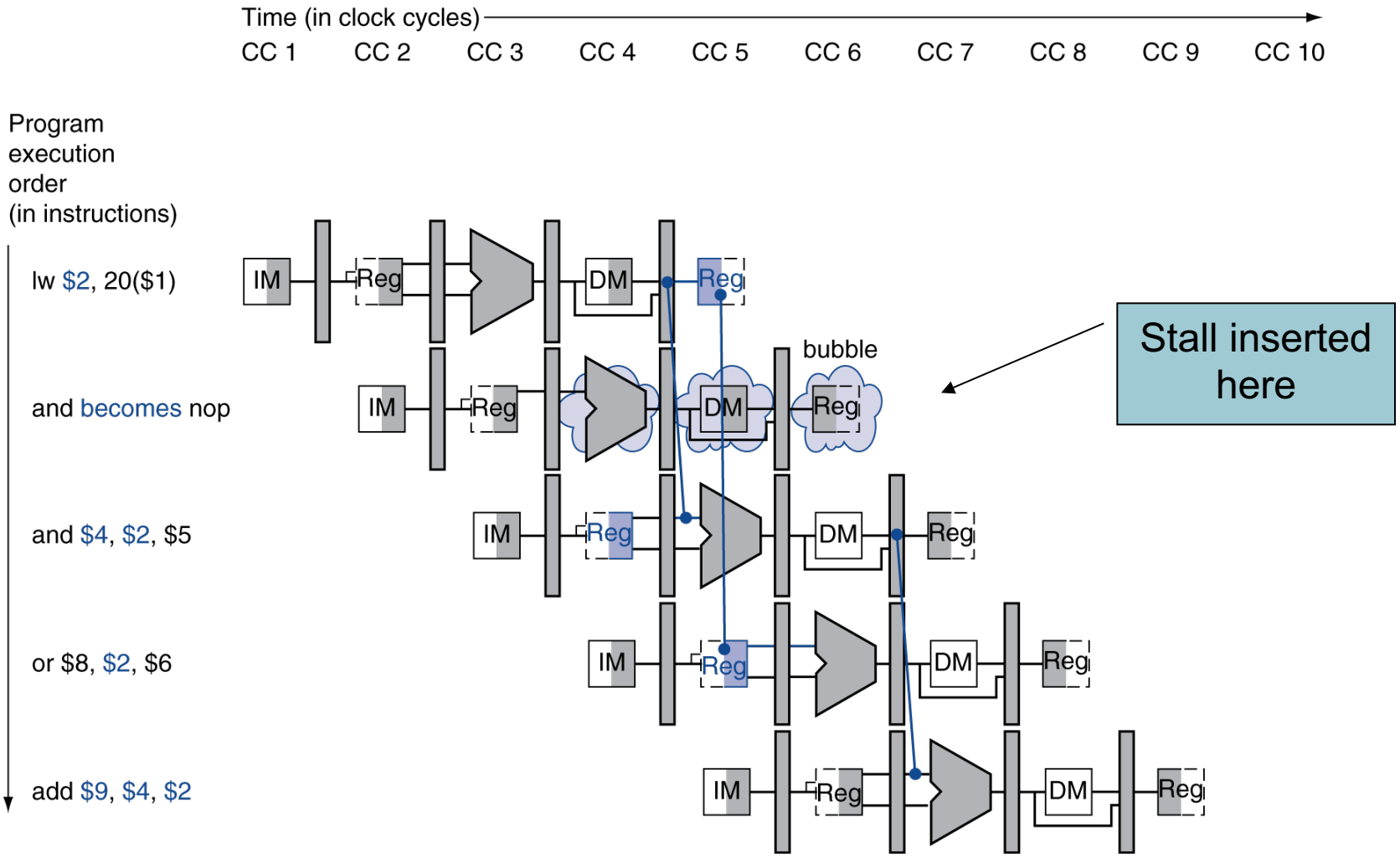
Load-Use Hazard Detection

- **We need a hazard detection unit operating during the ID stage**
  - So that it can insert the stall between the load instruction and its use
- **The hazard detection unit must check**
  - If a load is in execution and…
  - …if the destination register of the load in the EX stage matches either source register of the instruction in the ID stage
- **Stall the pipeline one clock cycle**
  - if (ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt)))

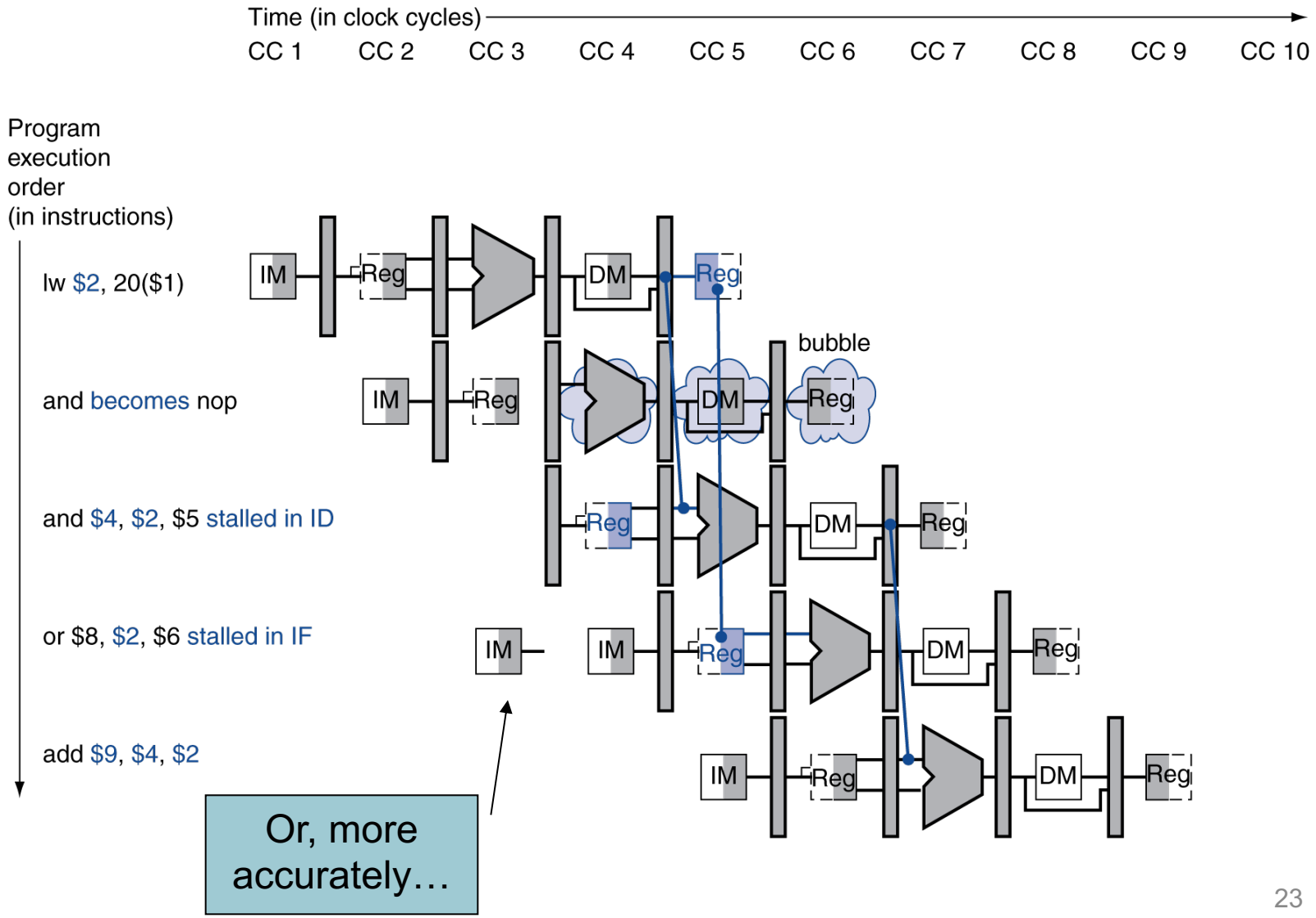
How to Stall the Pipeline

- **Force control values in ID/EX register to 0**
  - EX, MEM and WB do a **no-operation** (`nop`)
- **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 `lw`
    - Can subsequently forward to EX stage

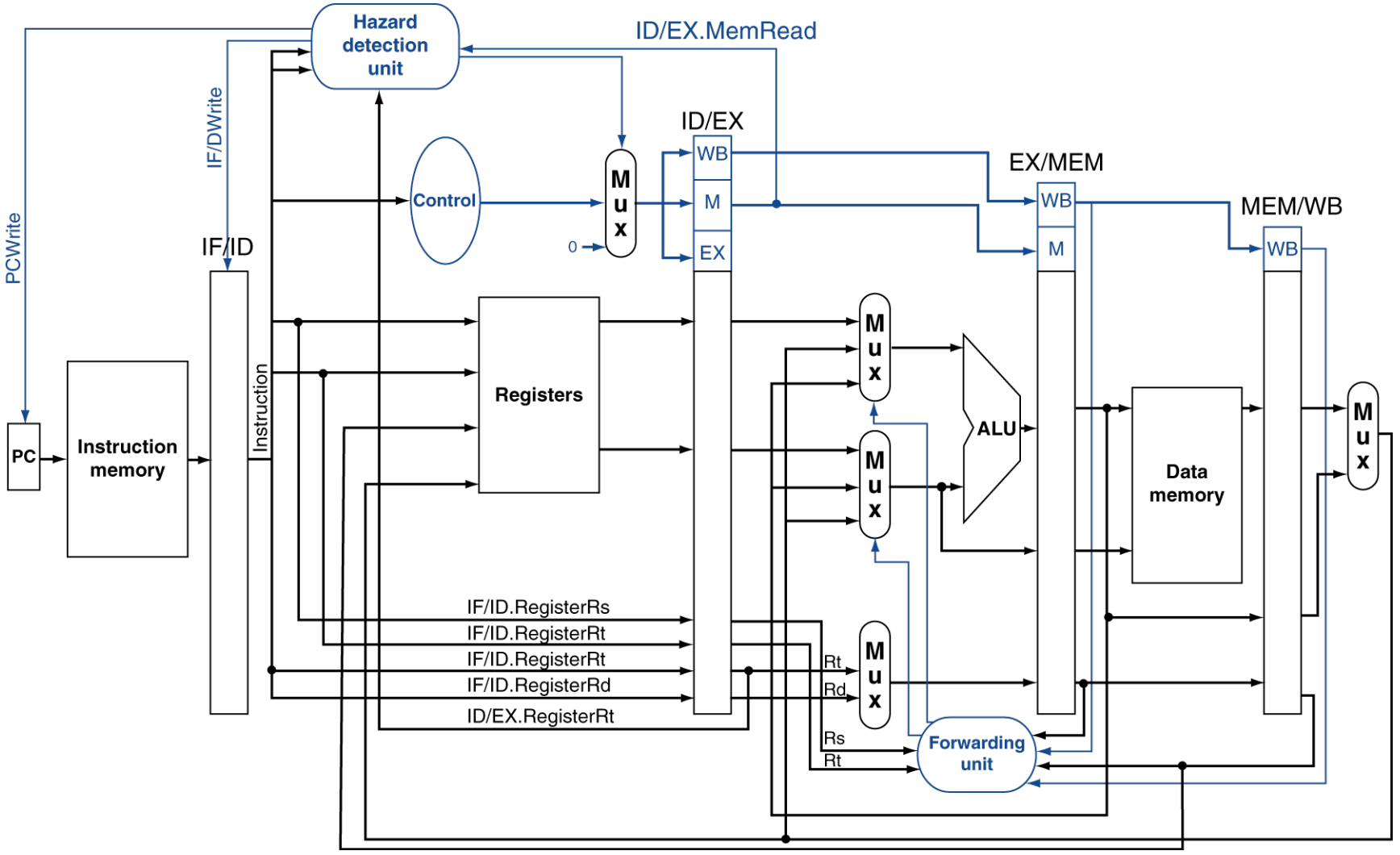
Load-Use Data Hazard



Load-Use Data Hazard



Datapath with Hazard Detection



Stalls and Performance

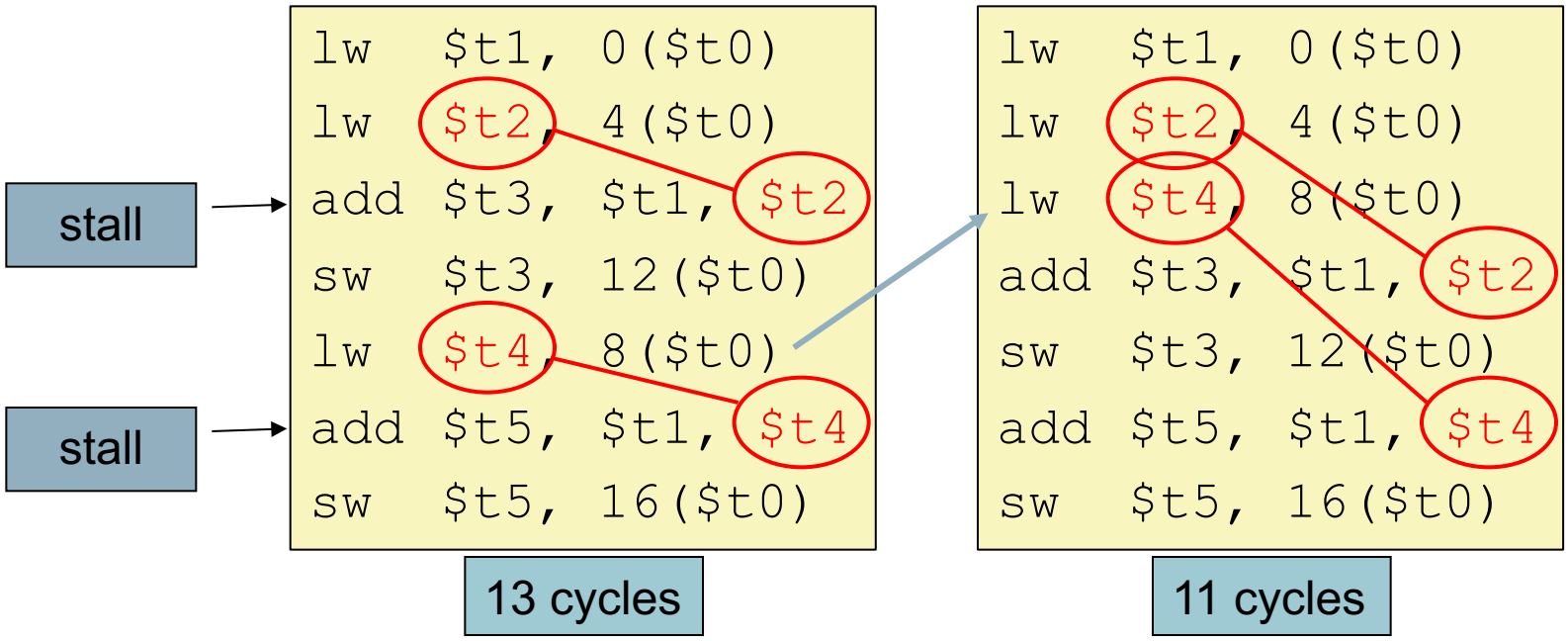
- **Stalls reduce performance but are required to get correct results**

- **Compiler can arrange code to avoid hazards and stalls**
  - Although it generally relies upon the hardware to resolve hazards and thereby ensure correct execution
  - Requires knowledge of the pipeline structure

Code Reordering to Avoid Stalls

- **Reorder code to avoid use of load result in the next instruction**

- **Consider the code for `A = B + E; C = B + F;`**



Data hazards: Summary

Situation	Example	Action

No dependence	`lw $t1, 45($t2)`
	`add $t5, $t6, $t7`
	`sub $t8, $t6, $t7`
	`or $t9, $t6, $t7`
		No hazard possible because no dependence exists on `$t1` in the immediately following three instructions

Dependence requiring stall	`lw $t1, 45($t2)`
	`add $t5, $t1, $t7`
	`sub $t8, $t6, $t7`
	`or $t9, $t6, $t7`
		Comparators detect the use of `$t1` in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX

Dependence overcome by forwarding	`lw $t1, 45($t2)`
	`add $t5, $t6, $t7`
	`sub $t8, $t1, $t7`
	`or $t9, $t6, $t7`
		Comparators detect the use of `$t1` in SUB and forward the result of LOAD to the ALU in time for SUB to begin with EX

Dependence with accesses in order (read after write)	`lw $t1, 45($t2)`
	`add $t5, $t6, $t7`
	`sub $t8, $t6, $t7`
	`or $t9, $t1, $t7`
		No action is required because the read of `$t1` by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half

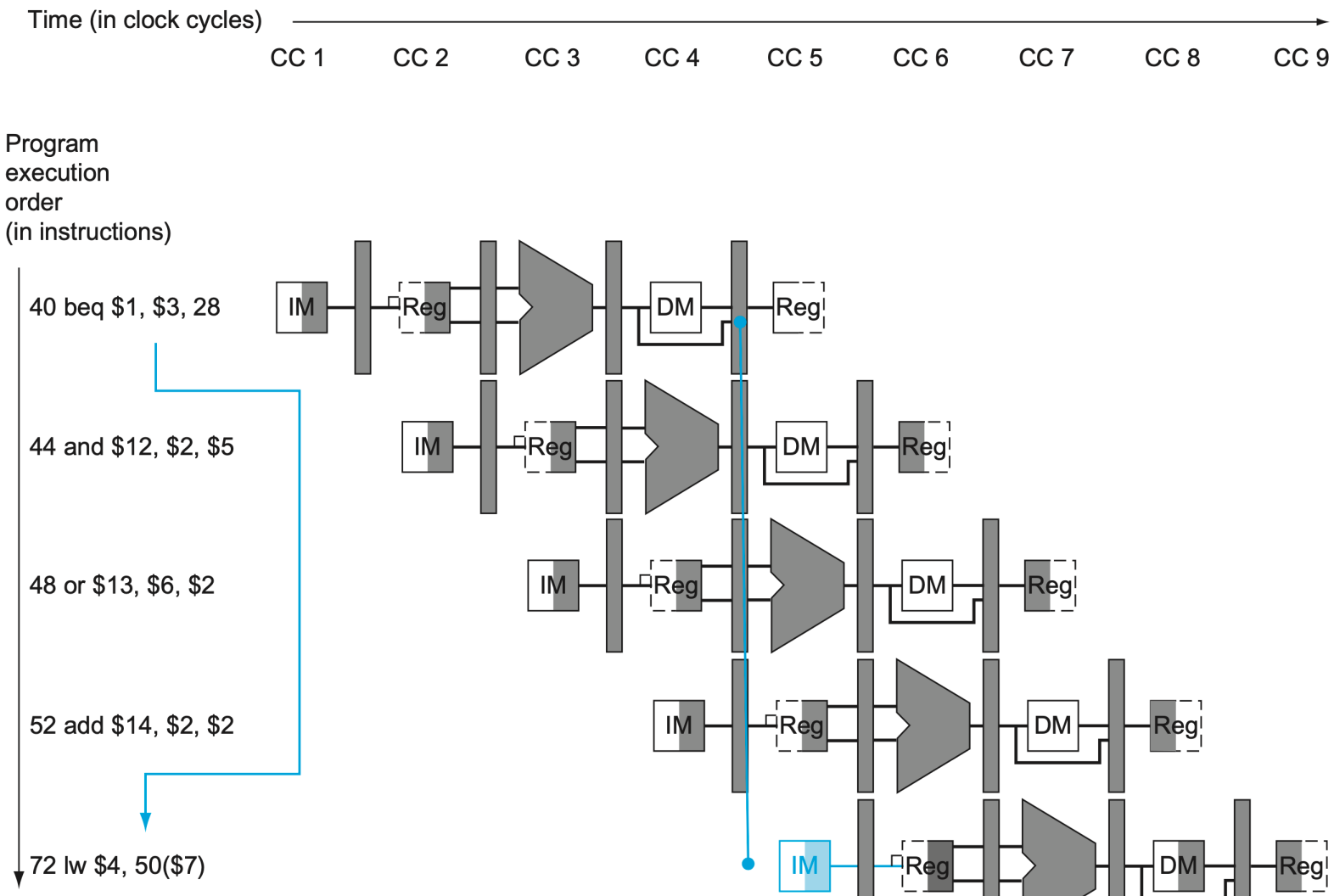
Control Hazards

- **An instruction must be fetched at every clock cycle to sustain the pipeline**
  - Yet, in our design, the decision about whether to branch doesn't occur until the MEM pipeline stage

- **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

```
36:   sub   $10, $4, $8
40:   beq   $1,  $3, 7      # PC-relative addressing
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 Hazards



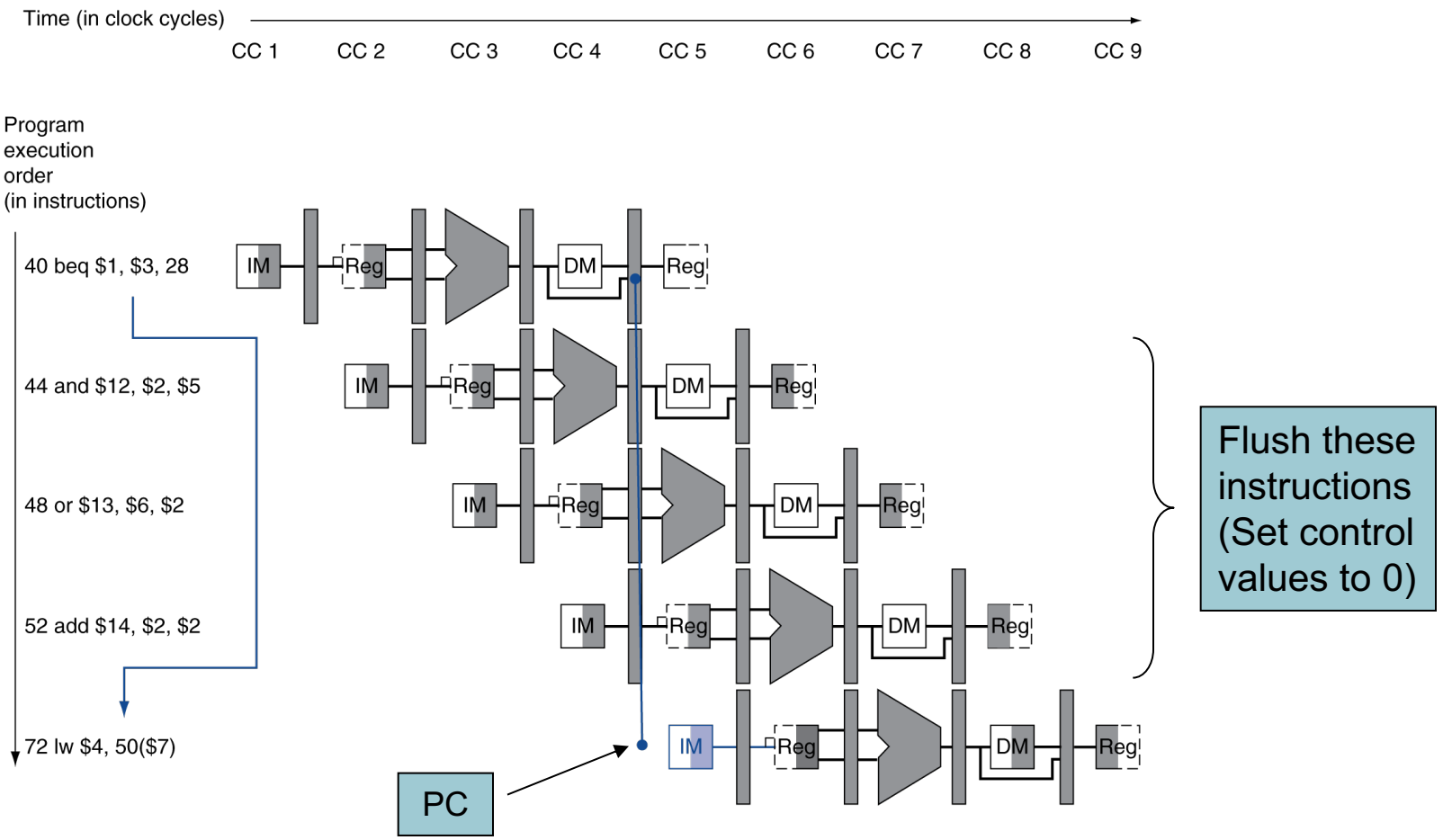
Stall on Branch

- **Wait until branch outcome is determined before fetching next instruction**
  - Makes all branches cost 4 cycles
- **Seems wasteful, particularly when the branch isn't taken**
  - Could have spent clock cycles fetching, decoding and executing next instructions
- **The cost of this option is too high for most computers to use and motivates another solutions**

Assume Branch Not Taken

- **Always assume branch will NOT be taken**
  - On average, branches are taken half the time
- **Cuts overall time for branch processing in half**
  - If prediction is correct, continue execution down the sequential instruction stream
  - If prediction is incorrect, just discard the instructions
- **Discarding instructions means we must be able to flush instructions in the IF, ID, and EX stages**
  - Just change control to 0

Assume Branch Not Taken



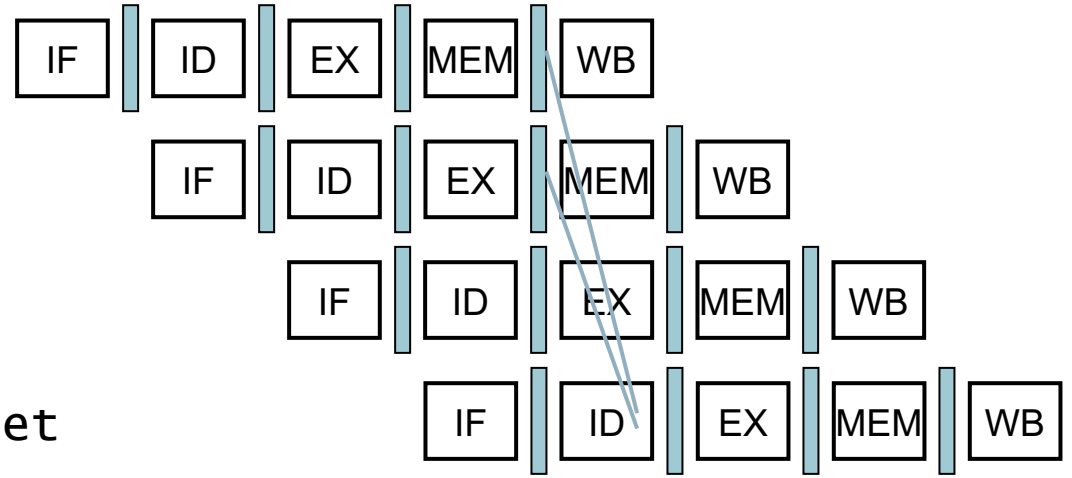
Reducing Branch Delay

- **Move hardware to determine outcome to ID stage**
  - Reduces the **penalty of a branch to only one instruction** if the branch is taken, namely, the one currently being fetched
- **Branch address calculation (easy part)**
  - We already have the PC value and the immediate field in the IF/ID pipeline register
  - Just move the branch adder from the EX stage to the ID stage
- **Moving the branch test implies additional forwarding and hazard detection hardware**
  - Since a branch dependent on a result still in the pipeline must still work properly with this optimization

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
```

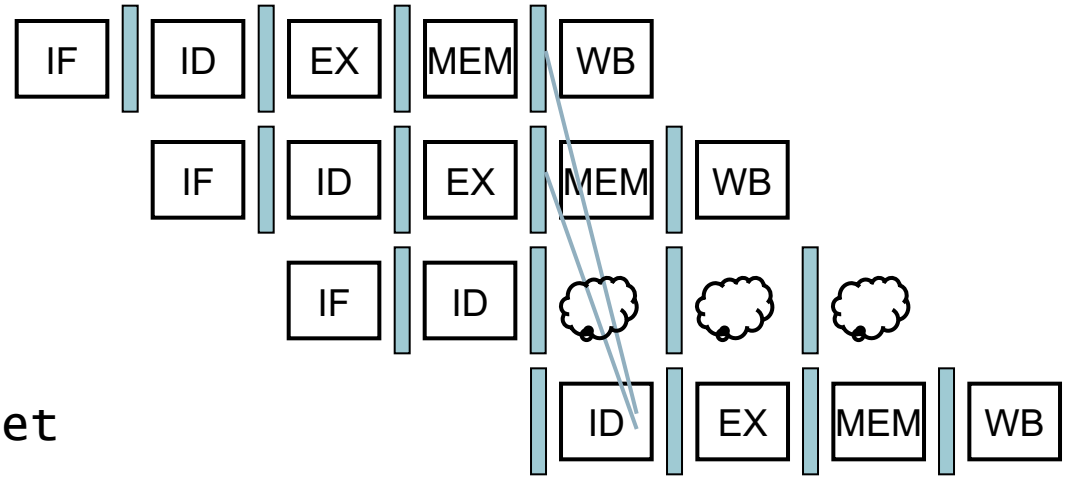


- **Can resolve using forwarding**

Data Hazards for Branches

- **If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction**

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

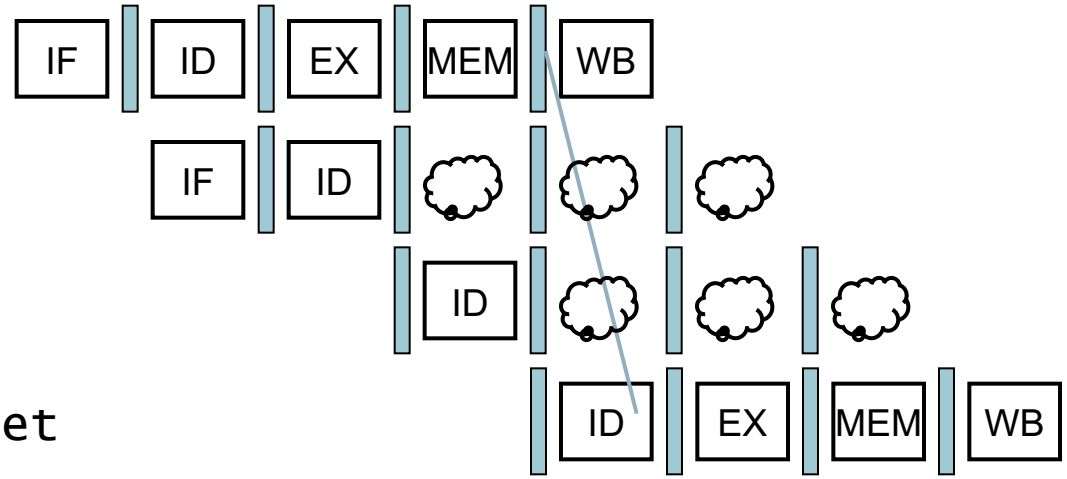


- **Need 1 stall cycle**

Data Hazards for Branches

- **If a comparison register is a destination of immediately preceding load instruction**

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



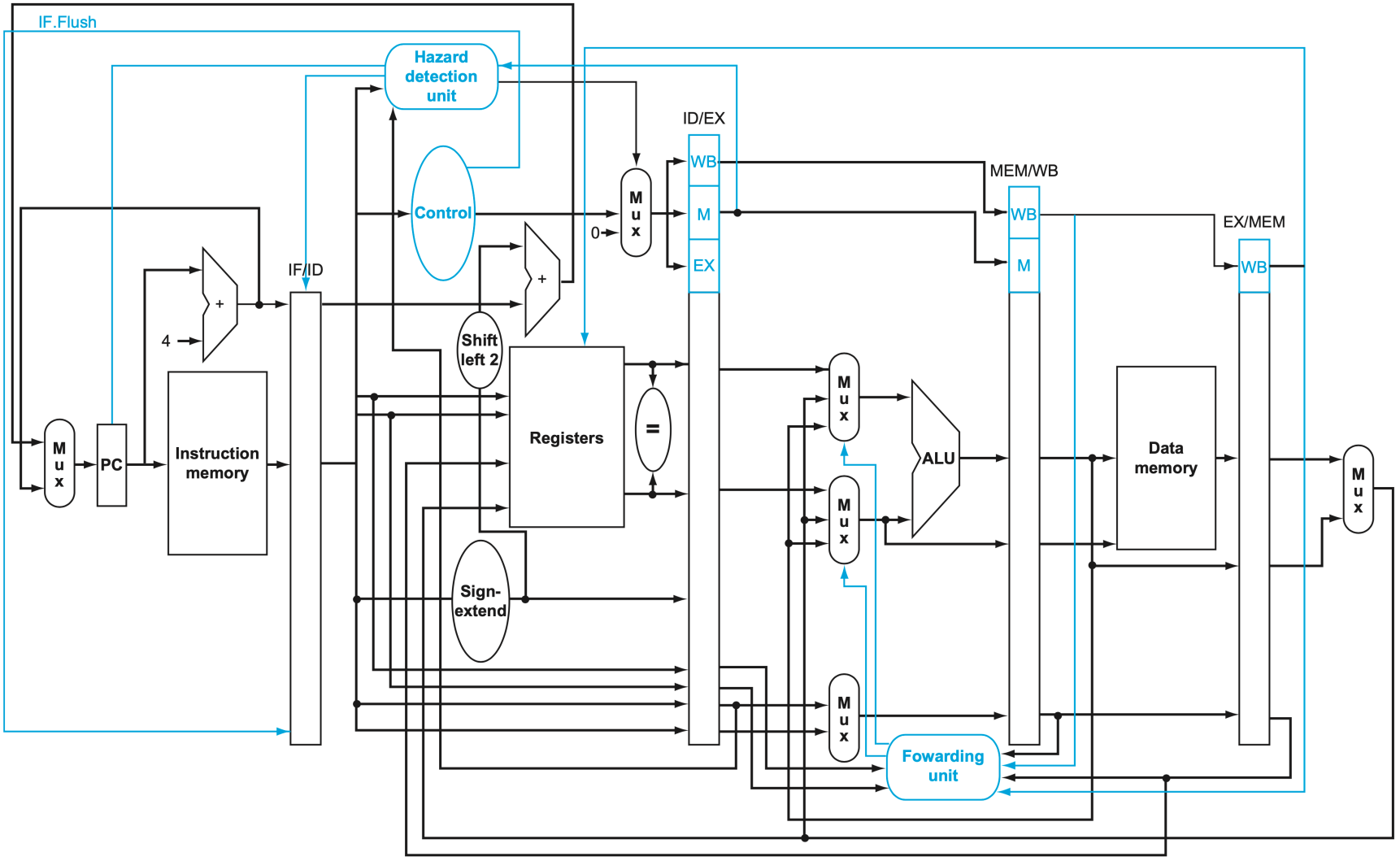
- **Need 2 stall cycles**

Reducing Branch Delay

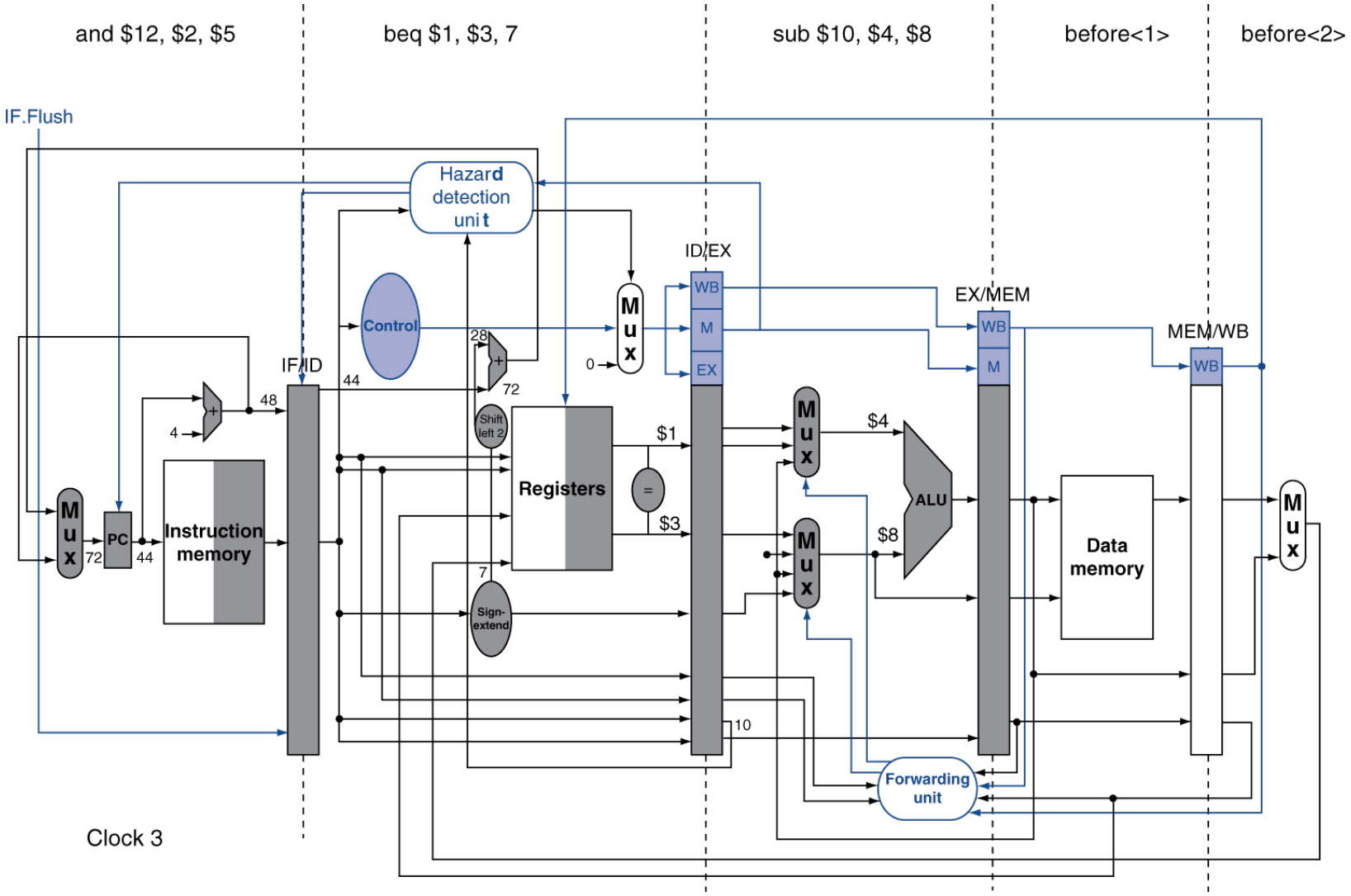
- **Despite these difficulties, moving the branch execution to the ID stage is an improvement**
  - Reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched

- **To flush instructions in the IF stage, we add a control line, called IF.Flush**
  - That zeros the instruction field of the IF/ID pipeline register
  - Clearing the register transforms the fetched instruction into a `nop`

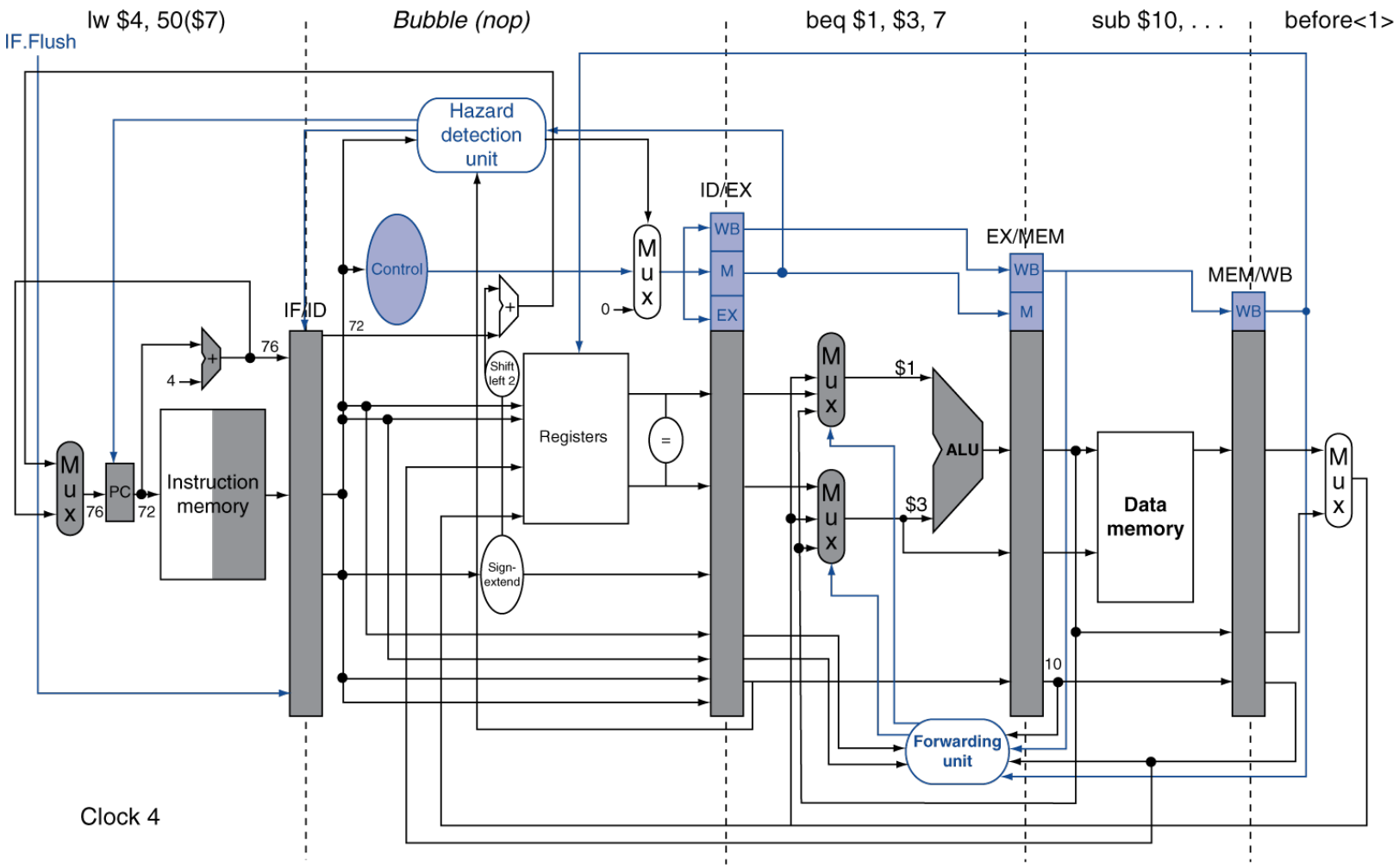
Reducing Branch Delay



Example: Branch Taken

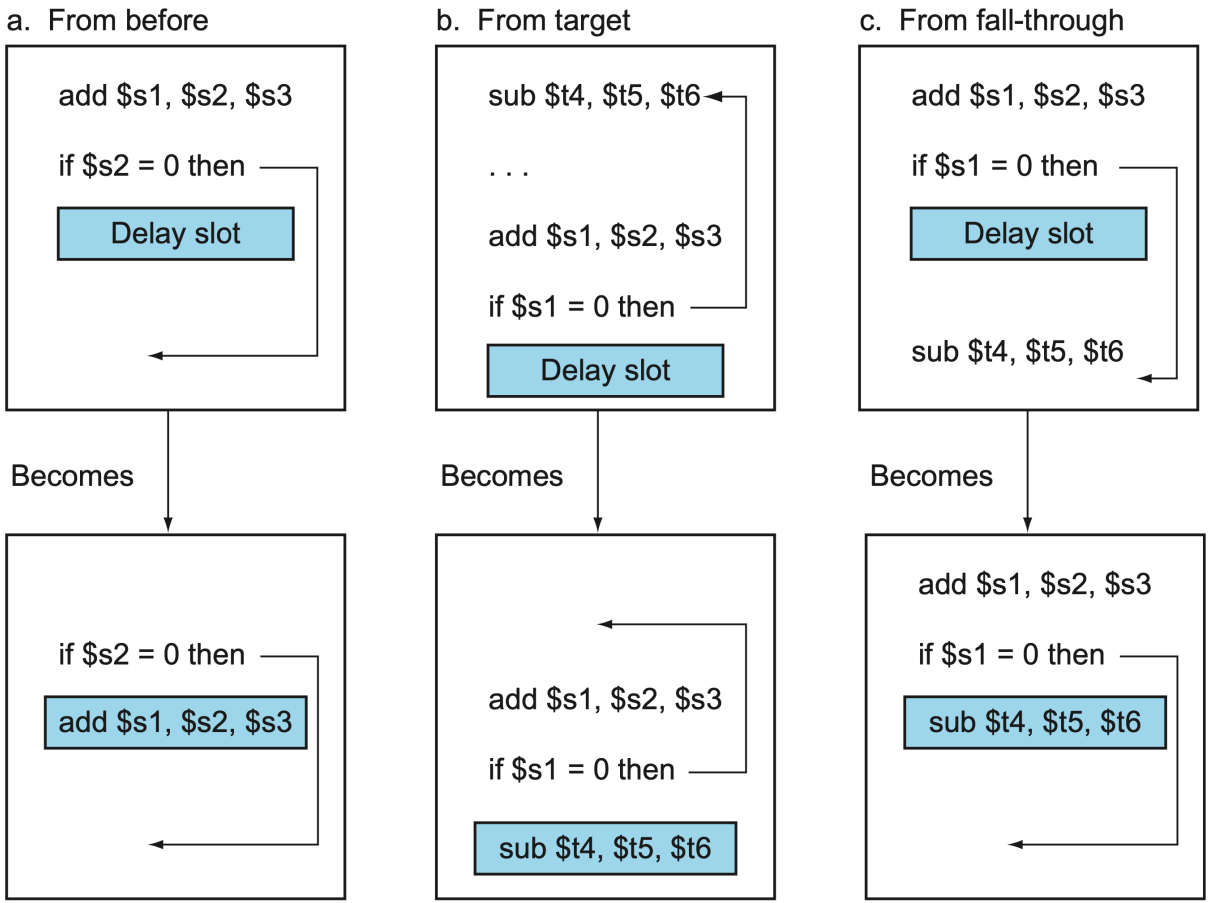


Example: Branch Taken



Branch Delay Slot

- **Compiler tries to place an instruction that always executes after the branch in the branch delay slot**
  - b) and c) are used when a) is not possible



Branch Delay Slot

- **Limitations on delayed branch scheduling arise from**
  1. The restrictions on the instructions that are scheduled into the delay slots
  2. The ability to predict at compile time whether a branch is likely to be taken or not

- **Is a simple and effective solution for a five-stage pipeline issuing one instruction each clock cycle**
  - As processors go to longer pipelines, the branch delay becomes longer, and a single delay slot is insufficient
  - Hence, delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches

Concluding Remarks

- **Pipelining improves instruction throughput using parallelism**
  - More instructions completed per second
  - Latency for each instruction not reduced
- **There are situations when the next instruction cannot execute in the following clock cycle**
  - These events are called hazards
- **Structural hazards**
  - Occur when the hardware cannot support the combination of instructions that we want to execute in the same clock cycle
  - Avoided by replicating functional units

Concluding Remarks

- **Data hazards**
  - Occur when the pipeline must be stalled because one step must wait for another to complete
  - Partially solved through forwarding
    - Demands adding extra hardware to retrieve the missing item early from the internal resources

- **Control hazards**
  - Arising from the need to decide based on the results of one instruction while others are executing
  - Partially solved through reducing the branch delay and prediction
    - Demands moving hardware to determine outcome of branch to ID stage and adding hardware to support prediction