



# Computer Architecture

**Lecturer: Mihai Negru**

2<sup>nd</sup> Year, Computer Science

**Lecture 8: Pipeline CPU Design**

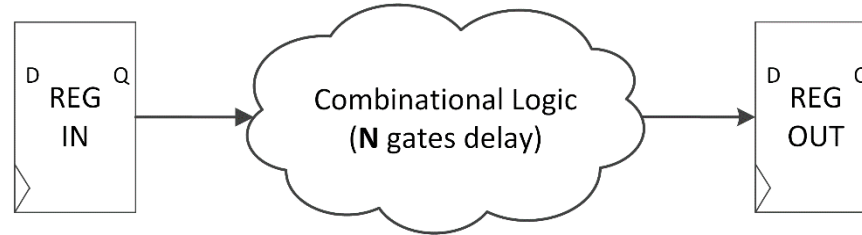
<http://users.utcluj.ro/~negrum/>



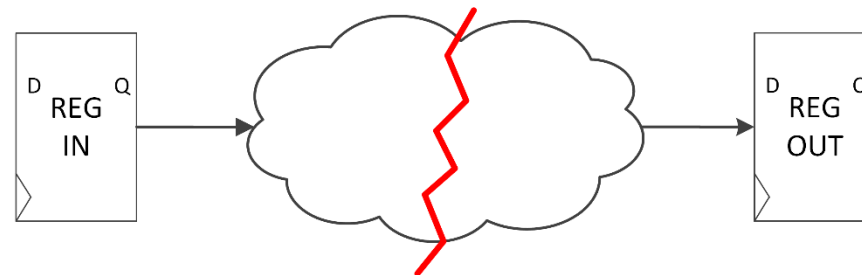
# The problem with long critical paths



- The critical path limits the clock frequency for the circuit



- The clock period must cover at least the equivalent N gates delay (plus the setup/hold time of the registers)
  - Time period for a new result:  $\approx N \times (\text{delay/gate}) = D$
  - Throughput (number of results/time interval):  $\text{Throughput}_{\text{init}} = 1/D$
- Solution to reduce the clock period (greater frequency):
    - Partition the critical path with synchronous storage elements (registers)

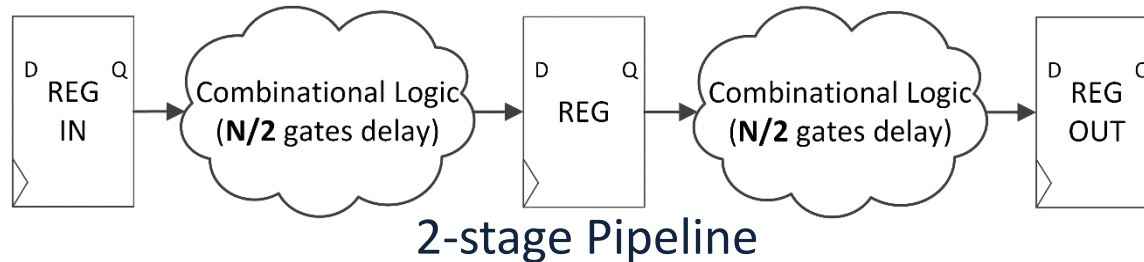




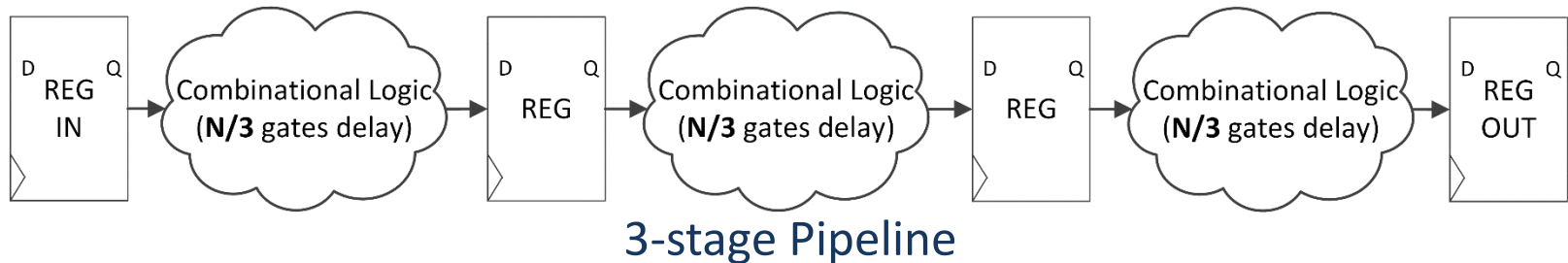
# Partitioning the Critical Path



- Pipeline: a set of data processing elements (**stages**) connected in series; the output of one stage is the input of the next one



- The clock period must cover at least the equivalent  $N/2$  gates delay
- Time period for a new result:  $\approx (N/2) \times (\text{delay/gate}) = D/2$
- Potential throughput is double:  $2/D = 2 \times (1/D) = 2 \times \text{Throughput}_{\text{init}}$



- The clock period must cover at least the equivalent  $N/3$  gates delay
- Time period for a new result:  $\approx (N/3) \times (\text{delay/gate}) = D/3$
- Potential throughput is double:  $3/D = 3 \times (1/D) = 3 \times \text{Throughput}_{\text{init}}$



# Partitioning the Critical Path

## Performance vs. Cost



- Does the throughput grow indefinitely?
  - No – the intermediate registers introduce supplemental delays (cumulative)
- If the duration without pipeline is  $D$  and the duration introduced by an intermediate register is  $D_{reg}$  → throughput for  $k / k+1$  stages:

$$Throughput(k) = \frac{1}{\frac{D}{k} + (k-1) \cdot D_{reg}}$$

$$Throughput(k+1) = \frac{1}{\frac{D}{k+1} + k \cdot D_{reg}}$$

- Throughput decreases ( $Throughput(k+1) < Throughput(k)$ ) when:

$$D_{reg} > \frac{D}{k} - \frac{D}{k+1}$$

- In reality problems of additional cost → balance between cost and performance in order to determine the optimum number of stages  $k_{opt}$ :
  - $k < k_{opt}$  – under-pipelined design
  - $k > k_{opt}$  – over-pipelined design



# Pipelining – General Aspects



- Pipelines allow overlapping execution of multiple instructions, exploiting the ILP (instruction level parallelism)
- The pipeline increases the instruction throughput (the number of instructions that can be executed in a unit of time)
- Pipeline decreases the clock cycle time
  - Ideal case: duration without pipeline / pipeline stages
  - In reality:
    - The stages will not be perfectly balanced
    - The pipeline rate is limited by the slowest pipeline stage
    - Other delays may appear due to hazards (stalls)
    - The setup/hold times of the intermediate registers
    - Time to “fill” pipeline and time to “drain” – reduce speedup
- In general **the pipeline is not visible to the programmer** (compiler)
- Pipelining yields a reduction in the **average execution time per instruction**.



# Pipelining – General Aspects



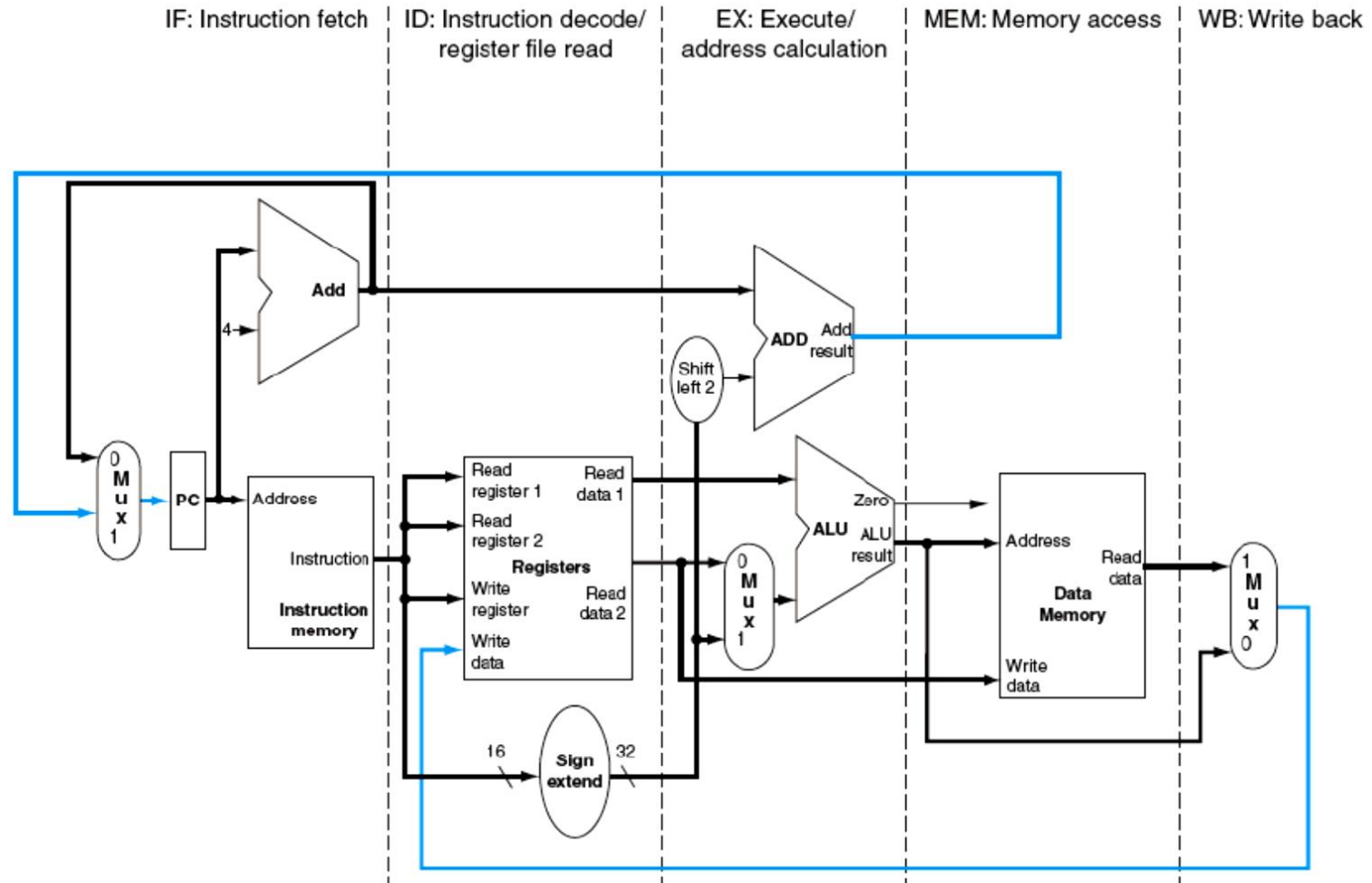
- Pipeline designer's goals
  - To balance the length of pipeline stages
  - To reduce the effects of the possible hazards
- ISA MIPS advantages for a pipeline implementation
  - All instructions are the same length
  - Just a few instruction formats, registers located in same place in instructions
  - Memory operands appear only in loads and stores
- What makes pipelining hard? → Hazards
  - Structural hazards: suppose we had only one memory.
  - Control hazards: need to worry about branch/jump instructions
  - Data hazards: an instruction depends on a previous instruction
- We will build a simple pipeline CPU starting from the Single-Cycle CPU, look at the possible problems and find solutions



# Pipeline CPU Design



- Start from the single-cycle implementation, 5 stages



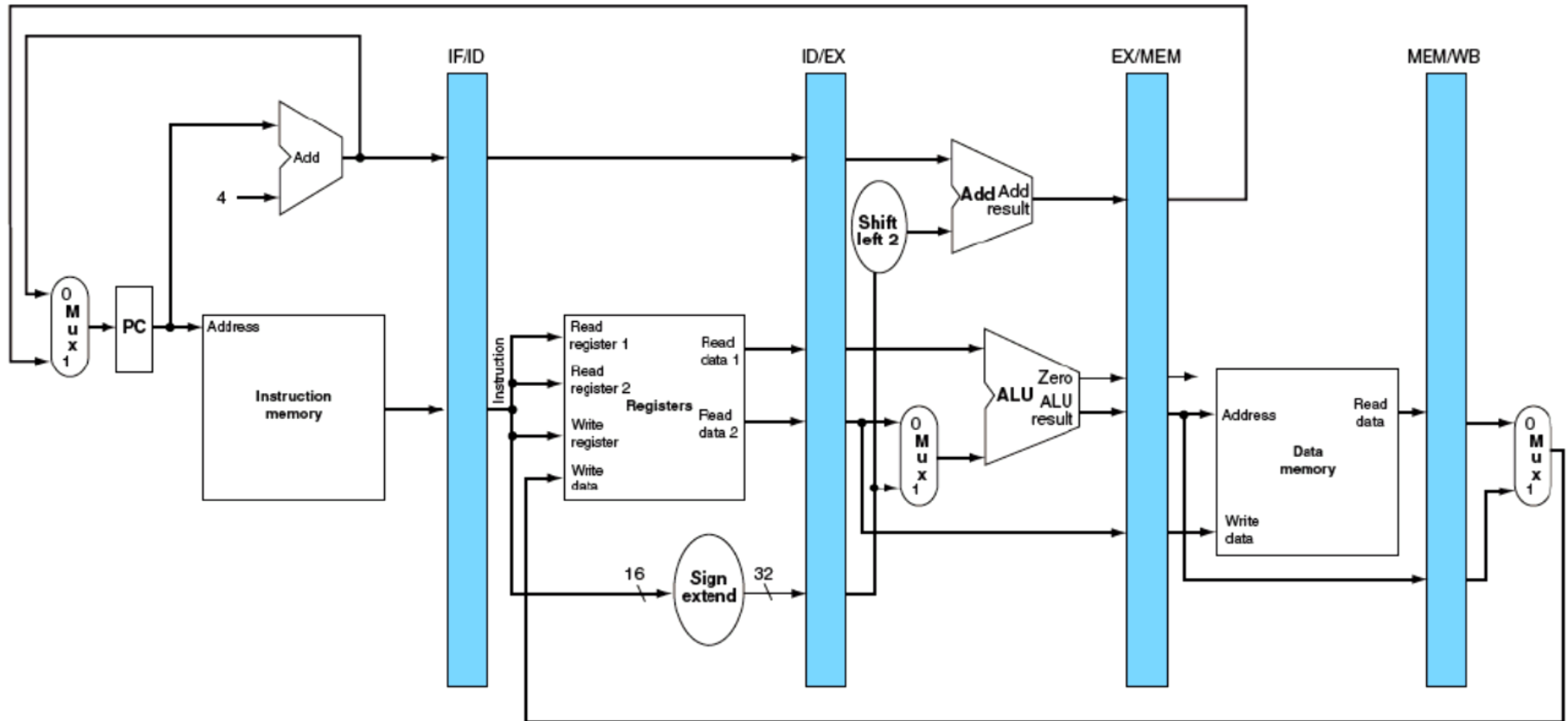


- Basic idea
  - Single-cycle data path divided into IF, ID, EX, MEM, WB stages
  - Up to 5 instructions will be in execution during each clock cycle
  - Normal data flow is from left to right
  - The paths flowing from right to left (register write-back and PC on a branch) introduce complications into pipeline (forwarding, branch delay)
- How to actually split the data-path into stages?
  - Add registers between each pair of pipe stages.
  - The registers transfer data values and control information from one stage to the next.
  - PC: a pipeline register before the IF stage → one pipeline register for each stage. Practically PC is a part of the IF!





# Pipeline CPU Design



Single-Cycle data-path, with added registers between execution stages

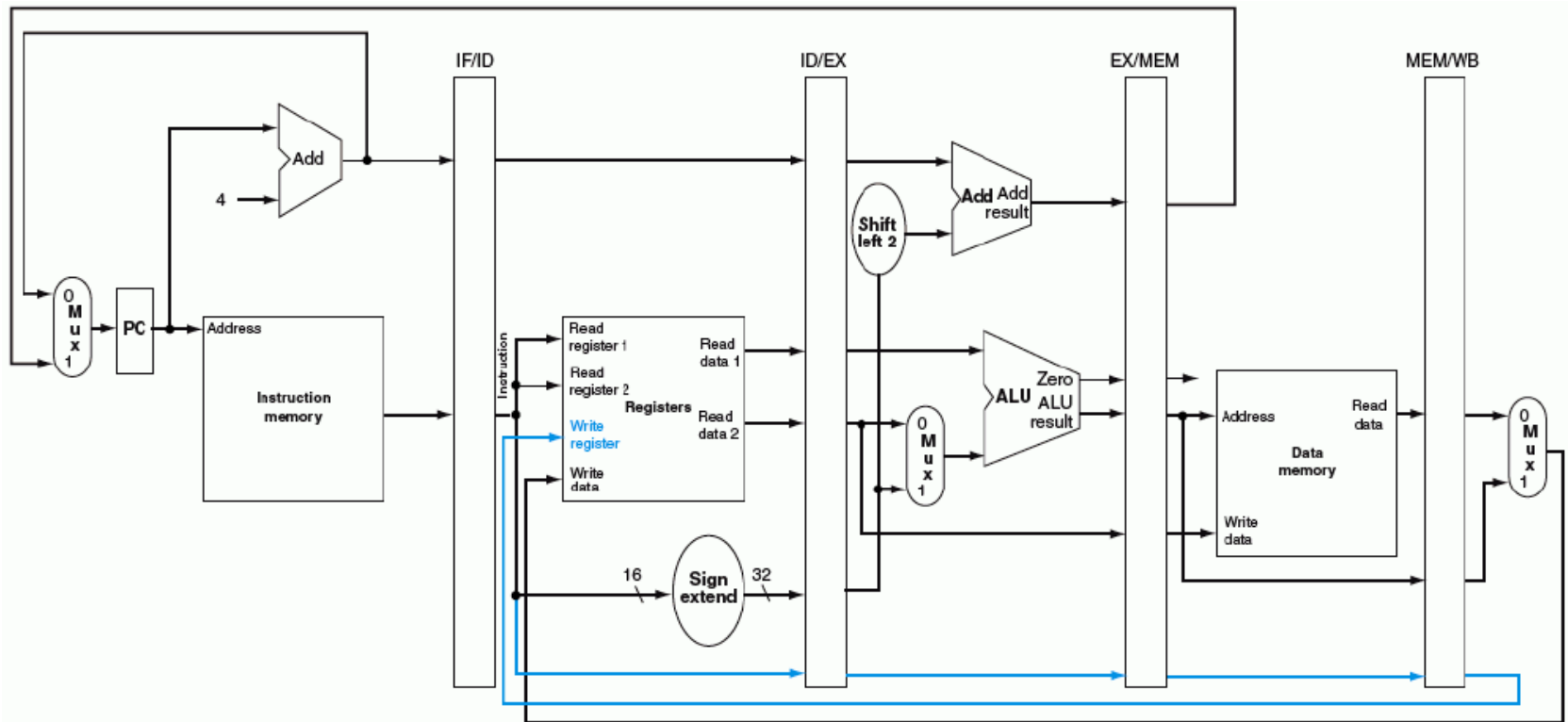
- Can you find a problem even if there are no dependencies?
- What type of instructions present this problem?



# Pipeline CPU Design



- Answer:
  - Register File: write address and write data come from different instructions



Corrected Pipeline: The destination address and data are pipelined simultaneously



# Actions in Pipeline Stages



- IF – Instruction Fetch Stage
  - PC – Address of Instruction Memory (IM); fetch the current instruction
    - $IF/ID.IR \leftarrow IM[PC]$
  - Increment the PC by 4
    - $IF/ID.PC \leftarrow PC + 4$
  - Write the PC using either the branch-target address or the incremented PC.
    - $PC \leftarrow PC + 4$  or  $PC \leftarrow EX/MEM.PC$  (PC + 4 + S\_Ext(Imm) << 2 – branch target)
- ID (ID&OF) – Instruction Decode and Operand Fetch Stage
  - OF: Read operands from the RF → in the second half of a clock cycle.
    - $ID/EX.A \leftarrow RF[rs]$
    - $ID/EX.B \leftarrow RF[rt]$
  - Sign or Zero extend the immediate field of the instruction
    - $ID/EX.Imm \leftarrow S\_Ext(Imm)$  or  $ID/EX.Imm \leftarrow Z\_Ext(Imm)$
  - The possible write data address for Register File is transmitted
    - $ID/EX.WriteRegRd \leftarrow rd$ ,  $ID/EX.WriteRegRt \leftarrow rt$  (the write register is rd or rt)
  - Copy the PC from the IF/ID stage
    - $ID/EX.PC \leftarrow IF/ID.PC$  (PC + 4)



# Actions in Pipeline Stages



- EX – Execute Stage
  - Memory reference instructions (LW & SW): calculate the effective address
    - $EX/MEM.ALUResult \leftarrow ID/EX.A + ID/EX.Imm$  (RF[rs] + S\_Ext(Imm))
  - Transmit Write data for SW
    - $EX/MEM.WriteData \leftarrow ID/EX.B$  (RF[rt])
  - Register-type ALU instructions:
    - $EX/MEM.ALUResult \leftarrow ID/EX.A (ALUOp-func) ID/EX.B$  (RF[rs] ALUOp-func R[rt])
  - Register-Immediate ALU instructions:
    - $EX/MEM.ALUResult \leftarrow ID/EX.A (ALUOp-codop) ID/EX.Imm$  – Arithmetic
    - $EX/MEM.ALUResult \leftarrow ID/EX.A (ALUOp-codop) ID/EX.Imm$  – Logical
  - Branch instructions: perform the equality test in ALU
    - $EX/MEM.Zero \leftarrow ALUZero$  (ID/EX.A – ID/EX.B)
  - Compute the branch-target address in the additional Adder
    - $EX/MEM.PC \leftarrow ID/EX.PC + ID/EX.Imm \ll 2$  (PC + 4 + S\_Ext(Imm) << 2)
  - The Write data address for Registers is transmitted
    - $EX/MEM.WriteReg \leftarrow ID/EX.WriteRegRD$  or  $ID/EX.WriteRegRt$



# Actions in Pipeline Stages



- MEM – Memory Stage
  - LW: read data memory at the address computed in EX stage.
    - $MEM/WB.LMD \leftarrow DM[EX/MEM.ALUResult]$
    - LMD – Load Memory Data register
  - SW: write the value of the RF[rt] register into the data memory.
    - $DM[EX/MEM.ALUResult] \leftarrow EX/MEM.WriteData$
  - The Write data address for Registers is transmitted
    - $MEM/WB.WriteReg \leftarrow EX/MEM.WriteReg$
  - ALU result is transmitted
    - $MEM/WB.ALUResult \leftarrow EX/MEM.ALUResult$
    - $BEQ: (EX/MEM.Zero) \rightarrow PC \leftarrow EX/MEM.PC$  (PC + 4 + S\_Ext(Imm) << 2)
- WB – Write Back Stage
  - Register – type and Register – Immediate ALU instructions:
    - Write the ALU result into the register file.
    - $RF[MEM/WB.WriteReg] \leftarrow MEM/WB.ALUResult$  (RF[rd] or RF[rt])
  - LW: Write the data read from Data Memory into the Register File
    - $RF[MEM/WB.WriteReg] \leftarrow MEM/WB.LMD$  (RF[rt])
  - Write to Register File → in the **first half** of the clock cycle



# Actions in Pipeline Stages



## Pipeline RTL Summary

	R – R (R-type)	R-I (I-type, aritm/logic)	LW	SW	BEQ
IF	$IF/ID.IR \leftarrow IM[PC]$ $IF/ID.PC \leftarrow PC + 4$ $PC \leftarrow PC + 4$ or $(EX/MEM.Zero \& Branch) \rightarrow PC \leftarrow EX/MEM.PC$				
ID	$ID/EX.A \leftarrow RF[rs]$ $ID/EX.B \leftarrow RF[rt]$ $ID/EX.Imm \leftarrow S\_Ext(Imm)$ or $ID/EX.Imm \leftarrow Z\_Ext(Imm)$ $ID/EX.WriteRegRd \leftarrow rd, ID/EX.WriteRegRt \leftarrow rt$ $ID/EX.PC \leftarrow IF/ID.PC$				
EX	$EX/MEM.ALUResult \leftarrow ID/EX.A$ $(ALUOp-Func) ID/EX.B$	$EX/MEM.ALUResult \leftarrow ID/EX.A$ $(ALUOp-codop) ID/EX.Imm$	$EX/MEM.ALUResult \leftarrow ID/EX.A + ID/EX.Imm$	$EX/MEM.ALUResult \leftarrow ID/EX.A + ID/EX.Imm$	$EX/MEM.Zero \leftarrow ALUZero (ID/EX.A - ID/EX.B)$
	$EX/MEM.WriteReg \leftarrow ID/EX.WriteRegRd$ or $ID/EX.WriteRegRt$			$EX/MEM.WriteData \leftarrow ID/EX.B$	$EX/MEM.PC \leftarrow ID/EX.PC + ID/EX.Imm \ll 2$
MEM	$MEM/WB.ALUResult \leftarrow EX/MEM.ALUResult$	$MEM/WB.ALUresult \leftarrow EX/MEM.ALUresult$	$MEM/WB.LMD \leftarrow DM[EX/MEM.ALUresult]$	$DM[EX/MEM.ALUResult] \leftarrow EX/MEM.WriteData$	$EX/MEM.Zero \rightarrow PC \leftarrow EX/MEM.PC$
	$MEM/WB.WriteReg \leftarrow EX/MEM.WriteReg$				
WB	$RF[MEM/WB.WriteReg] \leftarrow MEM/WB.ALUResult$	$RF[MEM/WB.WriteReg] \leftarrow MEM/WB.ALUResult$	$RF[MEM/WB.WriteReg] \leftarrow MEM/WB.LMD$		

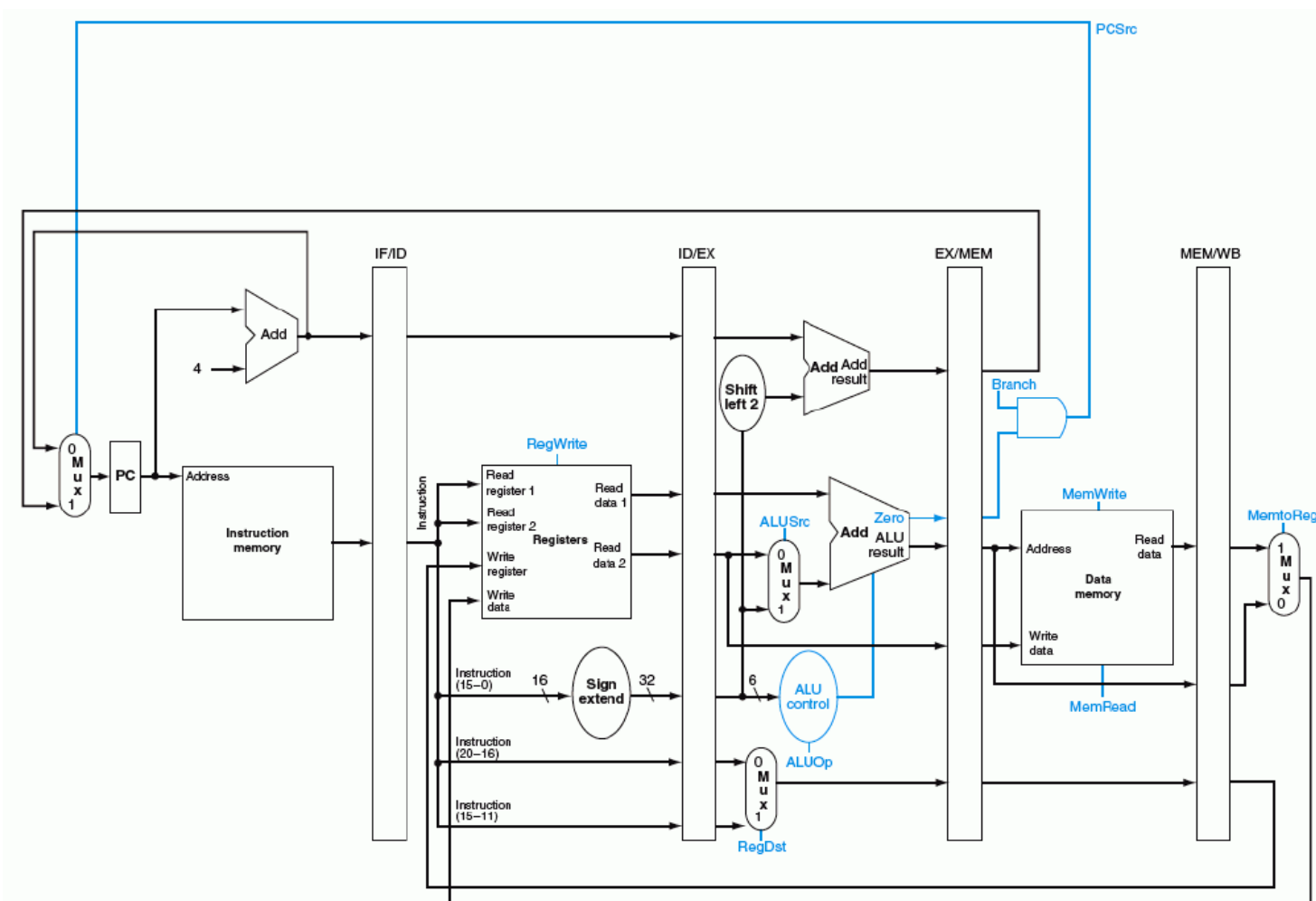
What else must be transferred along with the data in order for the instructions to work?



# Pipeline CPU Design – Control



What needs to be controlled in each stage?



MIPS Pipeline with Control Signals



# Pipeline CPU Design – Control



Instr	EX Stage			MEM Stage			WB Stage	
	RegDst	ALUOp	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
R-type	1	10	0	0	0	0	1	0
LW	0	00	1	0	1	0	1	1
SW	X	00	1	0	0	1	0	X
BEQ	X	01	0	1	0	0	0	X

MIPS Pipeline Control Settings in different stages (IF and ID are common)

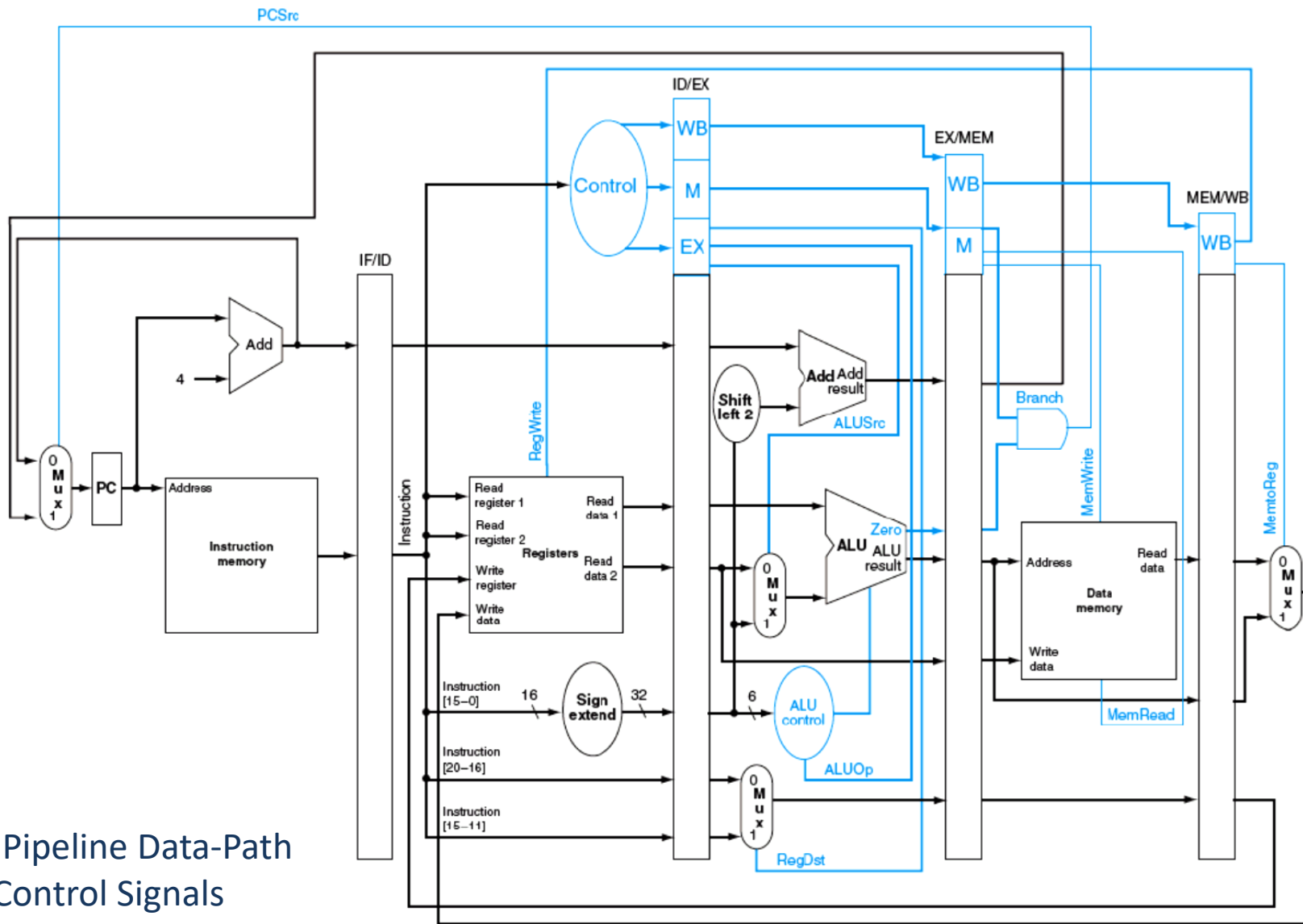
- Pipeline Control Signals must “travel” together with the intermediate results
  - The Main Control Unit generates the control signals during ID Stage
  - Control signals for EX Stage (RegDst, ALUOp, ALUSrc) are used 1 clock cycle later
  - Control signals for MEM (MemWrite, Branch, ...) are used 2 clock cycles later
  - Control signals for WB (MemtoReg, RegWrite) are used 3 clock cycles later
  - Pass control signals along just like the data from one stage to another







# Pipeline CPU Design – Control



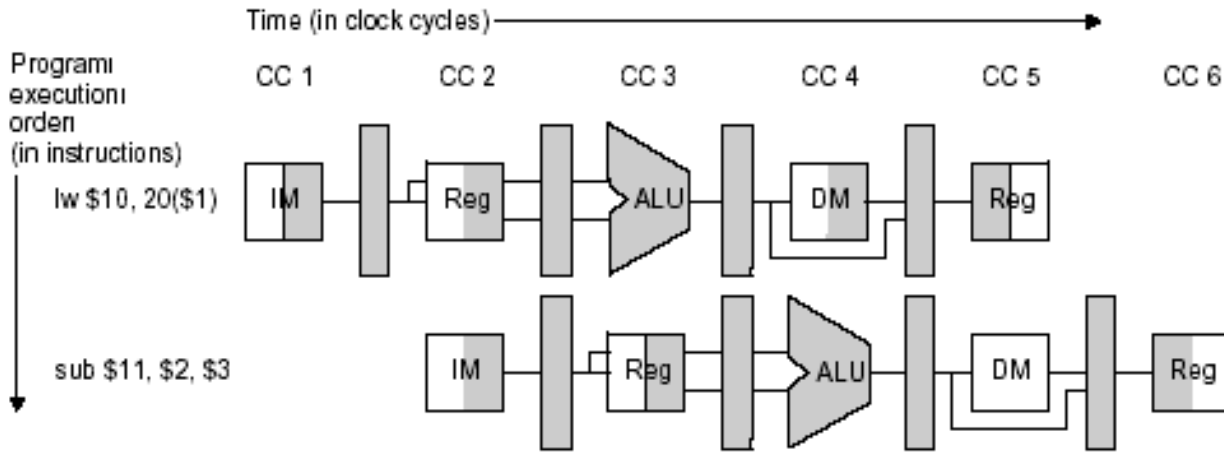
MIPS Pipeline Data-Path with Control Signals



# Pipeline Representations



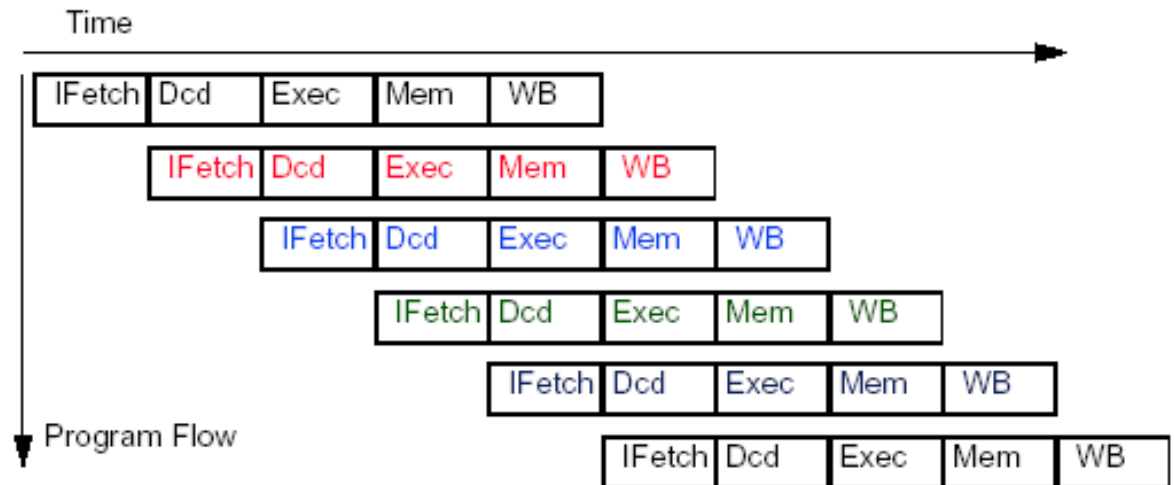
## Graphically Representing Pipelines



### Utility:

- How many cycles does it take to execute this code?
- What is the ALU doing during cycle 4?

## Conventional Pipelined Execution Representation





# Pipeline CPU Design – Hazards



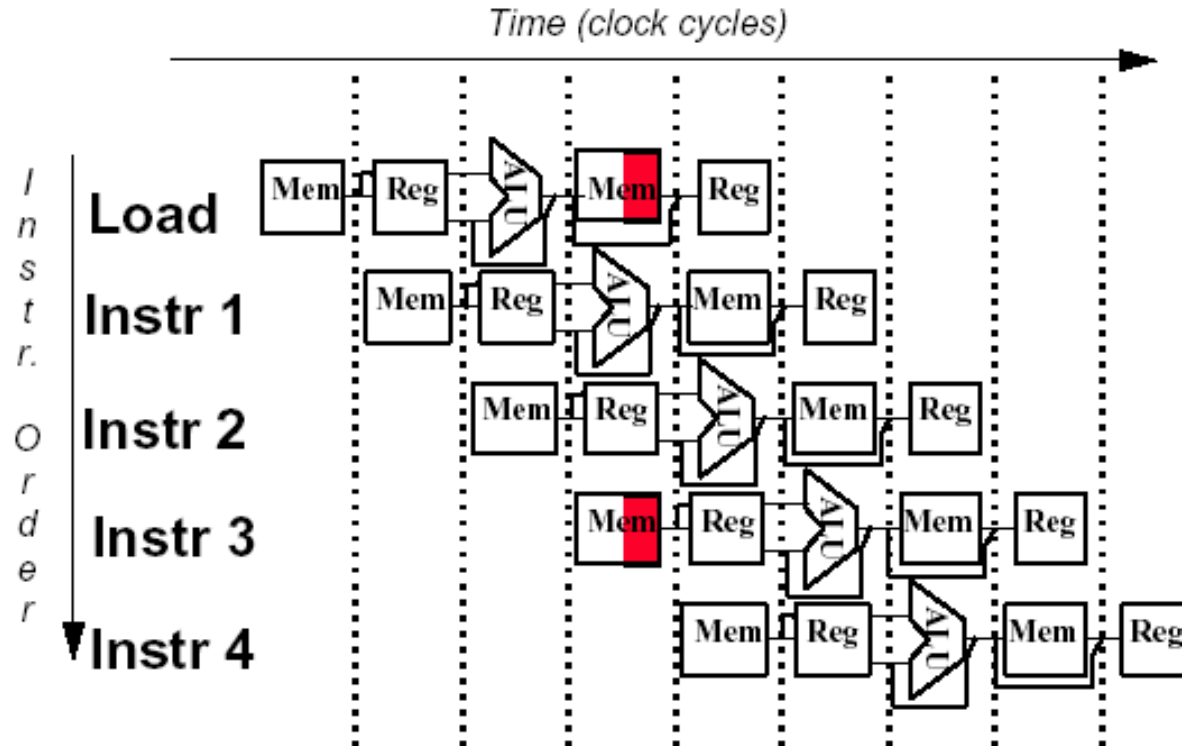
- Hazards – Situations in pipelining when the next instruction cannot be executed in the following clock cycle
- 3 types of Hazards
  - **Structural Hazards** (resource dependency) – two instructions attempt to use the same resource at the same time → resource constraints
  - **Data Hazards** (data dependency)
    - Attempt to use data before it is ready (data availability)
    - For an instruction in ID stage the source operand(s) are produced by a prior instruction still in the pipeline
  - **Control Hazards** (flow control)
    - Attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - Pipelining of branches, jumps & other instructions that change the PC
- Can always resolve hazards by waiting
  - Pipeline control must detect the hazard and take action to resolve hazards
  - Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” (NoOps) in the pipeline



# Pipeline CPU Design – Hazards



- Structural Hazards
  - Single Memory → Structural Hazard



Simultaneous Memory access of Load and Instruction 3, IF

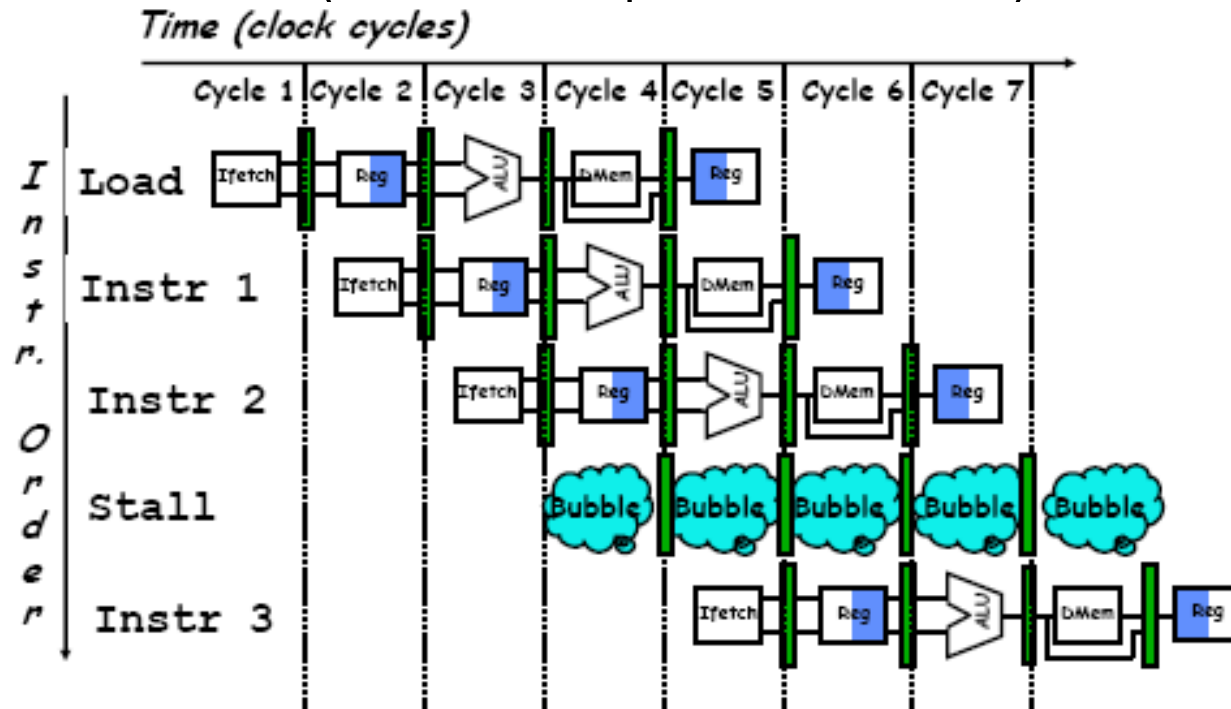
Is there a solution?



# Pipeline CPU Design – Hazards



- Structural Hazard
  - Single Memory
  - Wait solution – slow (reduces the speed of execution)



A single memory: Wait – Stall – Bubble – NoOp

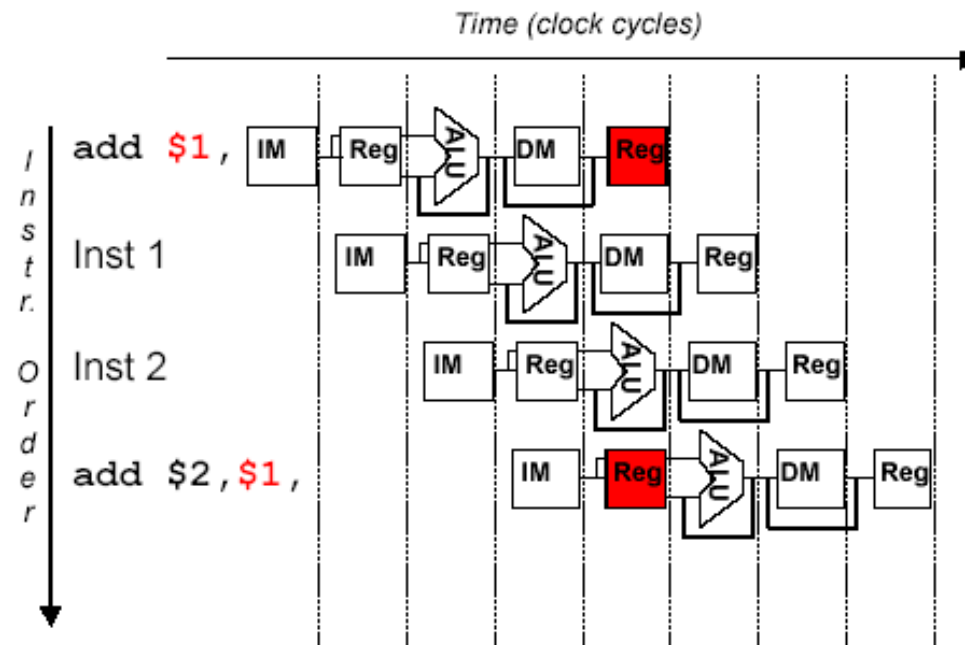
The used solution – Independent Instruction and Data (cache in real processors) memories; IM & DM are already “inherited” from the single-cycle processor



# Pipeline CPU Design – Hazards



- Structural Hazard
  - Simultaneous access to the Register File → Structural hazard
  - Read operands in ID stage and write result in WB stage



Simultaneous Register File access in WB (write) and ID (read) stages

- Solution:
  - Register file writes in the first half and reads in the second half of the cycle
  - Reads are asynchronous

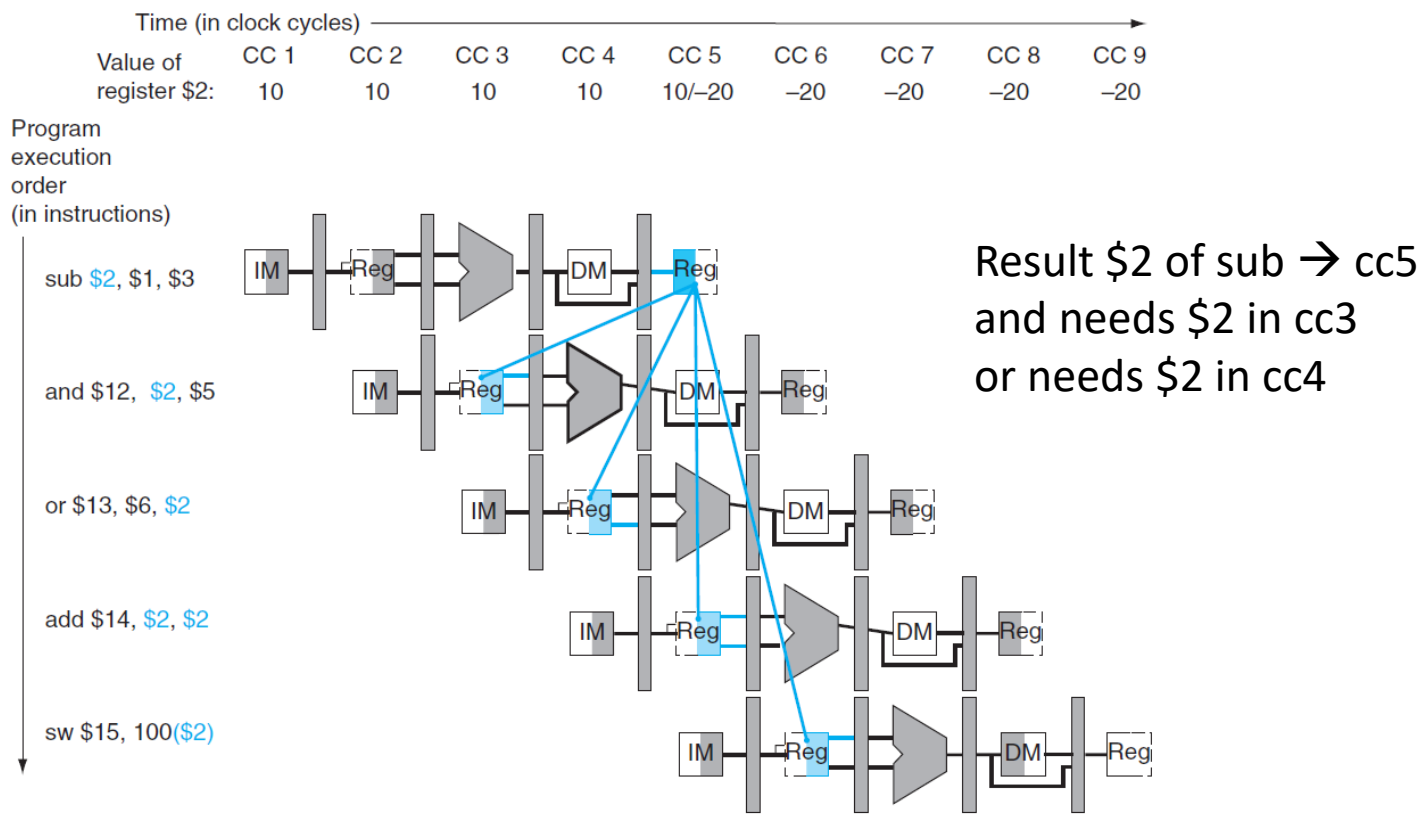


# Pipeline CPU Design – Hazards



- Data Hazards and Forwarding

- The operands of the instructions are not available yet (they will be produced by previous instructions in the pipeline)



Dependencies that “go backward in time” cause data hazards





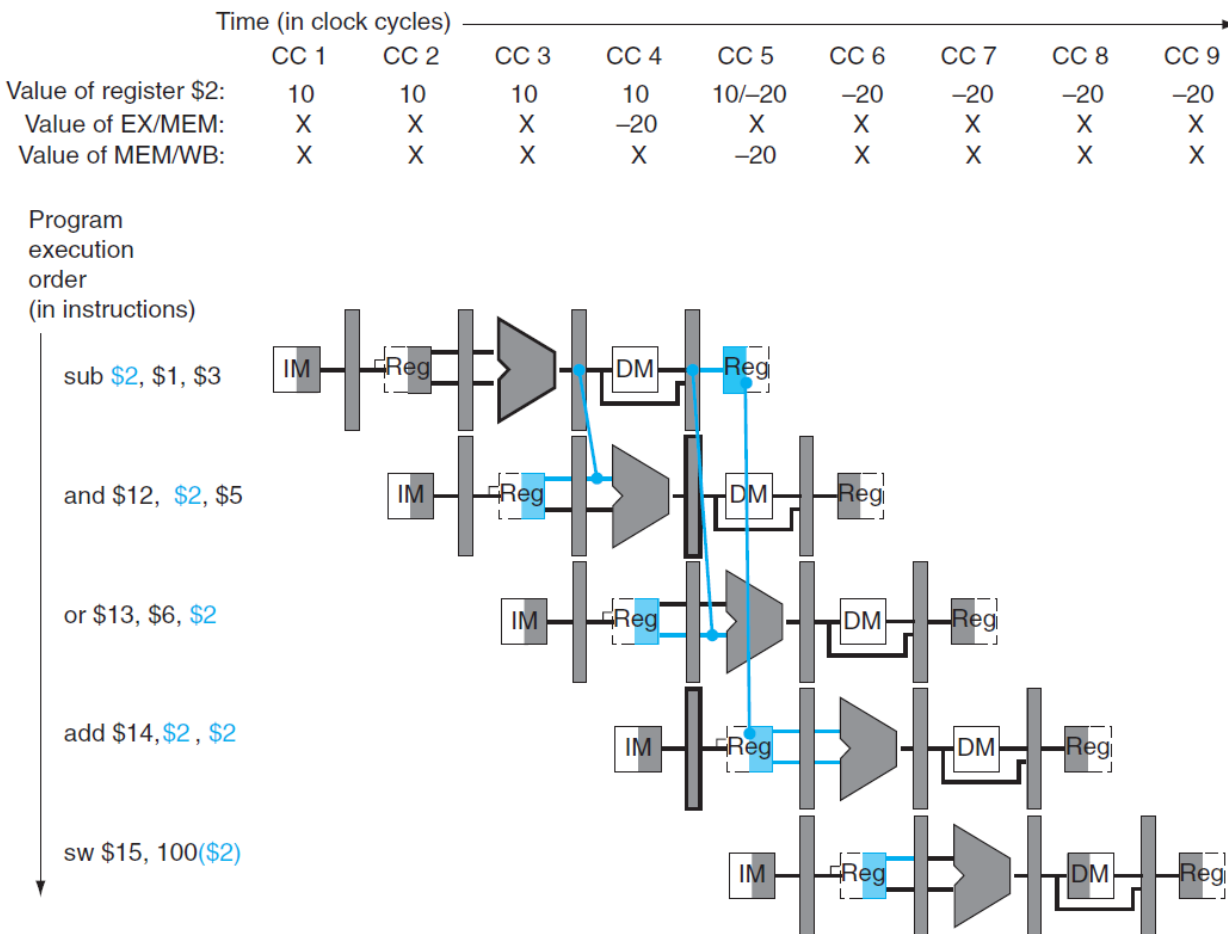
- Data Hazards and Forwarding
  - This hazard appears in ID stage, when an operand is in other pipeline stages
  - The usual name of this type of data hazard is **RAW (read after write)**
  - Software solution (programmer or compiler) – insert **NoOps**
  - Hardware solution – use **Forwarding**
- Forwarding – use temporary results, don't wait for them to be written in the Register File
  - “Forward” result from one stage to another to avoid the data hazard
  - The forwarded result can go to either ALU input; both ALU inputs can use forwarded inputs from the same or from different pipeline registers



# Pipeline CPU Design – Hazards



## • Data Hazards and Forwarding



- **Forwarding** – the use of the result from the EX and MEM stages, before they are written in the Register File

- **Forwarding** of **sub** result from **EX/MEM**, **MEM/WB** to the input of the **ALU** for **and** & **or** instructions in **cc4**, **cc5**

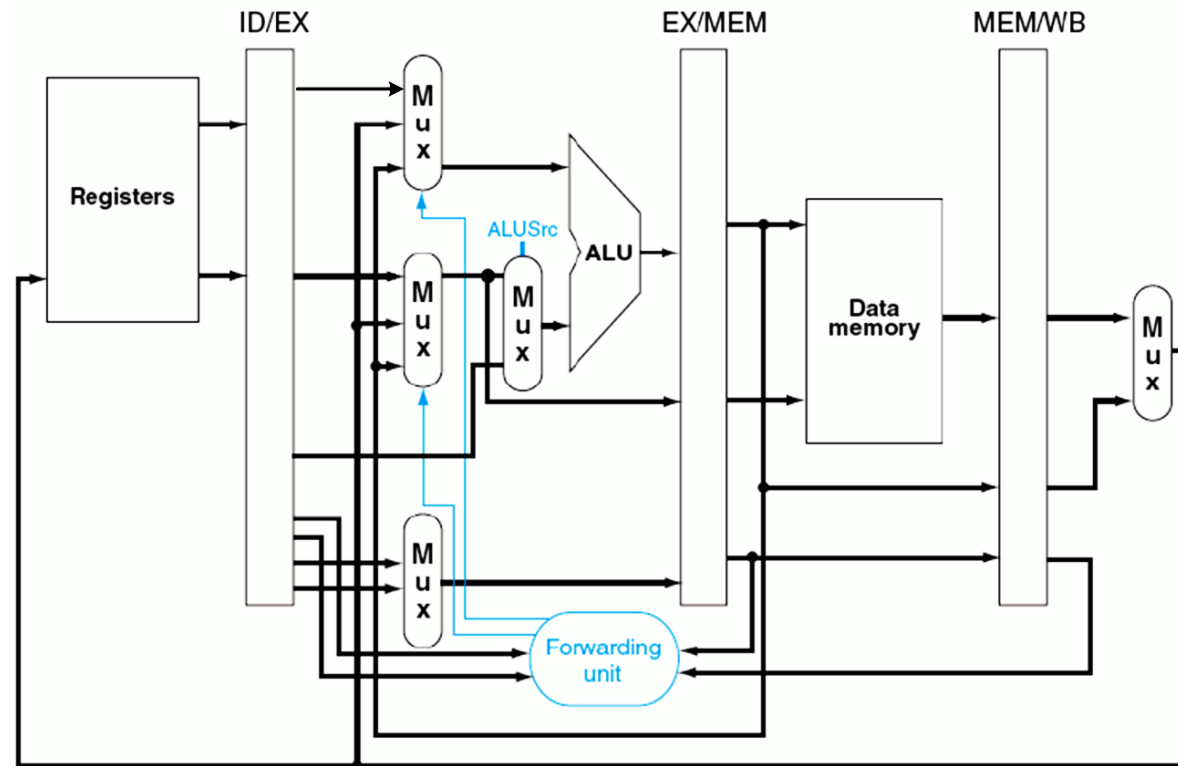
- The **forwarding** for the **add** instruction through the Register File is **implicit**



# Pipeline CPU Design – Hazards



- Hazard Detection for Forwarding
  - Hazard must be detected before execution
  - Detect if  $rs, rt$  at EX stage equals  $rd, rt$  in MEM or WB stage
  - In case of hazard, the data should be forwarded to the input of the ALU



Forwarding Unit and Multiplexers added to implement Forwarding



- Hazard Detection for Forwarding
  - Three sources for each MUX
    - ID/EX: no forwarding, just from the register file (00)
    - EX/MEM: forwarded data from the prior ALU results (10)
    - MEM/WB: from data memory or an earlier ALU result (01)
  - Notation: ID/EX.RegisterRs = The rs address field of the Register File in ID/EX pipeline register
  - Detect the Hazard condition
  - Do not forward when it is unnecessary, check the RegWrite signal
  - The control logic for Forwarding is implemented by the **Forwarding Unit**



- Data Forwarding Control Conditions

- MEM Stage Hazard (consecutive instructions, distance = 1)

if (EX/MEM.RegWrite and

(EX/MEM.RegisterRd != 0) and (no forwarding for reg.nr = 0)

(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

- Forwards the result from the previous instruction to either input of the ALU





# Pipeline CPU Design – Hazards



- Data Forwarding Control Conditions

- WB Stage Hazard (consecutive instructions, distance = 2)

Typical Problem, <b>WB</b> hazard	Atypical problem, <b>MEM</b> hazard
<pre>add \$1, \$1, \$2 add \$5, \$2, \$3 add \$1, \$1, \$4</pre> 	<pre>add \$1, \$1, \$2 add \$1, \$1, \$3 add \$1, \$1, \$4</pre> 

- The most recent result must be forwarded
- The hazard in MEM Stage has priority, if it exists
- The hazard in WB Stage is resolved only if there is no MEM Stage hazard with the previous instruction



- Data Forwarding Control Conditions

- WB Stage Hazard

if (MEM/WB.RegWrite

    and (MEM/WB.RegisterRd != 0)

    and (EX/MEM.RegisterRd != ID/EX.RegisterRs)     (No MEM Hazard)

    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))     (WB Hazard condition)

        ForwardA = 01

if (MEM/WB.RegWrite

    and (MEM/WB.RegisterRd != 0)

    and (EX/MEM.RegisterRd != ID/EX.RegisterRt)     (No MEM Hazard)

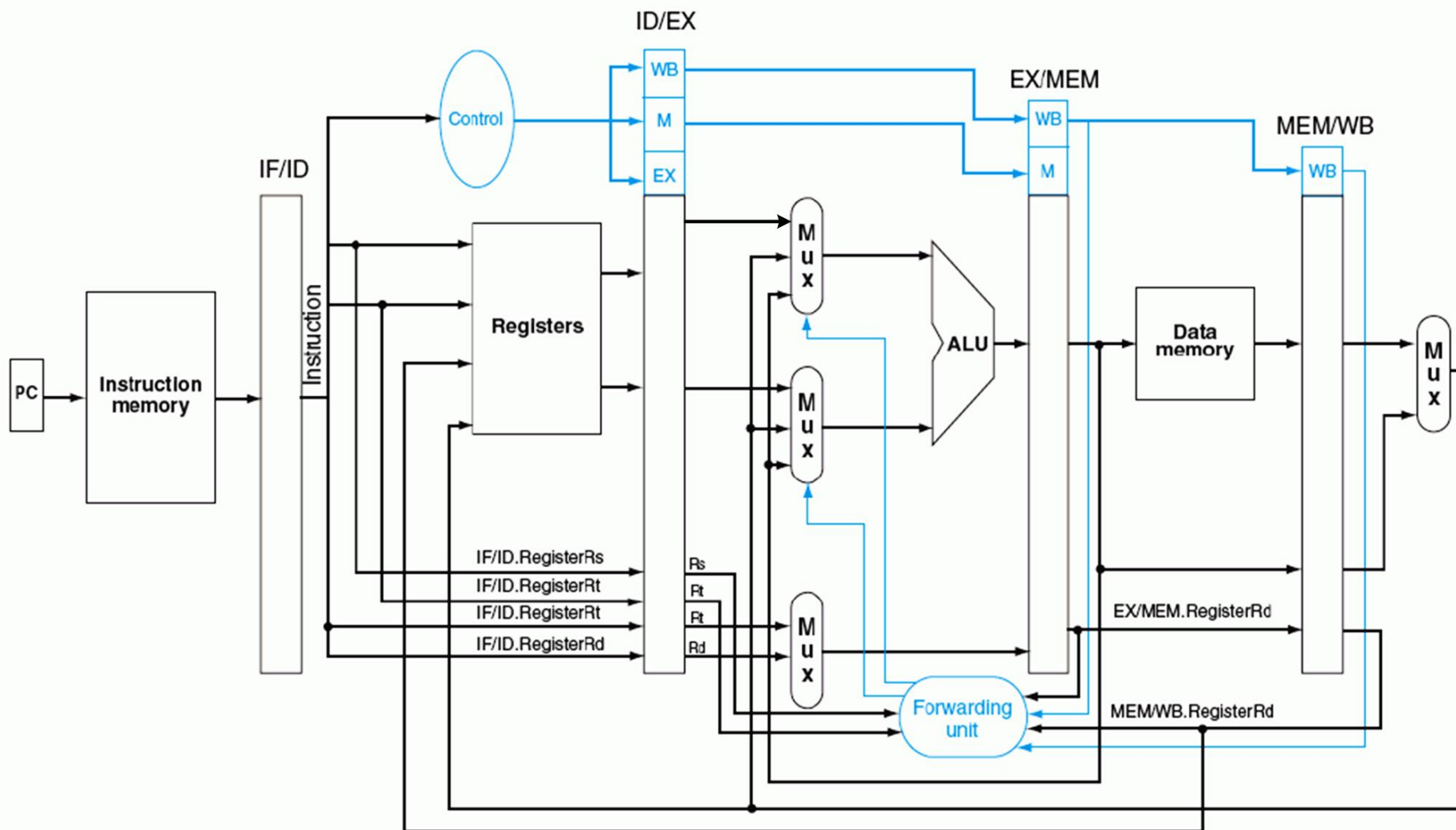
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))     (WB Hazard condition)

        ForwardB = 01

- Forwards the result from the previous instruction to either input of the ALU



# Pipeline CPU Design – Hazards



MIPS Pipeline Data-Path with forwarding (incomplete – some things are missing)

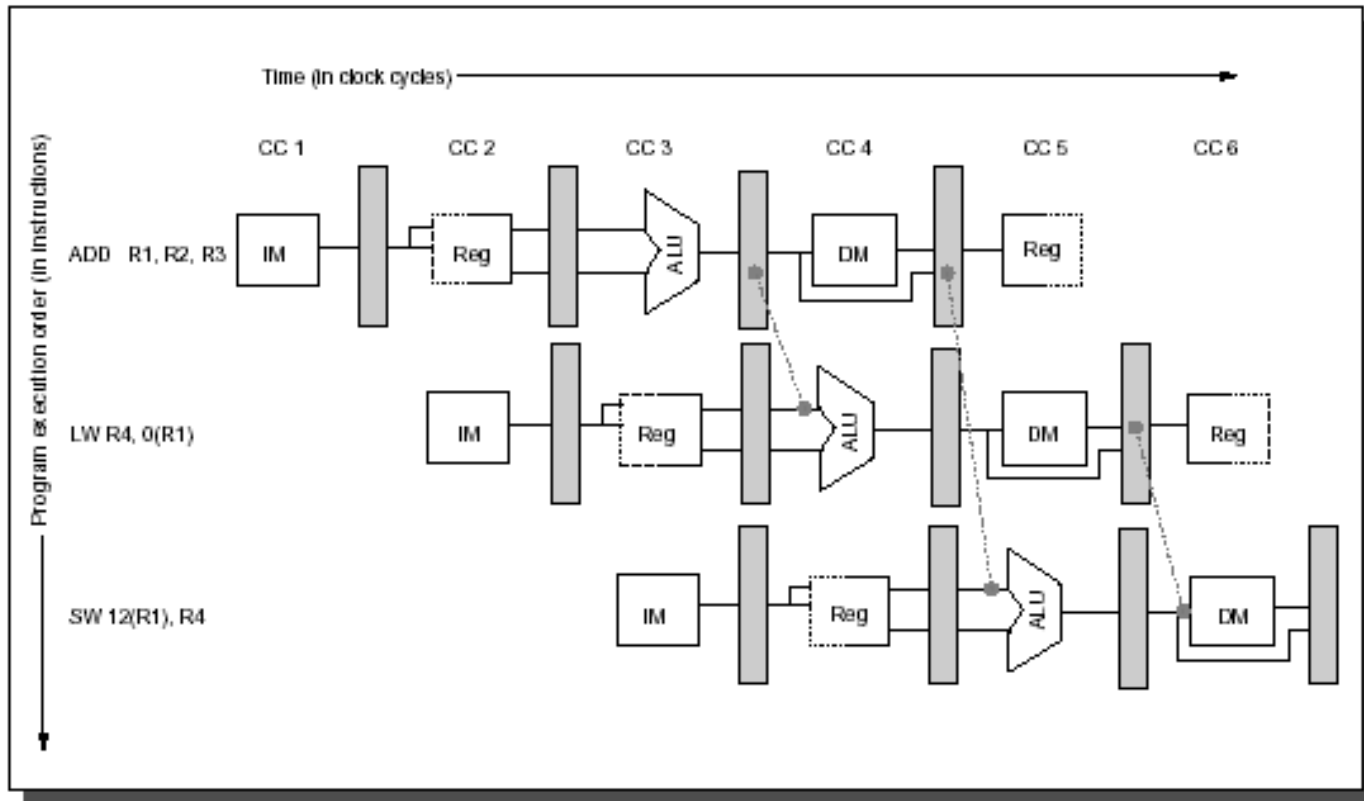




# Pipeline CPU Design – Hazards



- Forwarding can also be applied to memory reference instructions



## Example for LW-SW instructions:

- Forwarding in CC4 and CC5 – the same as before
- Forwarding in CC6 – we need another multiplexer at the input of the memory in order to allow the forwarding of the output from the memory from the previous clock cycle

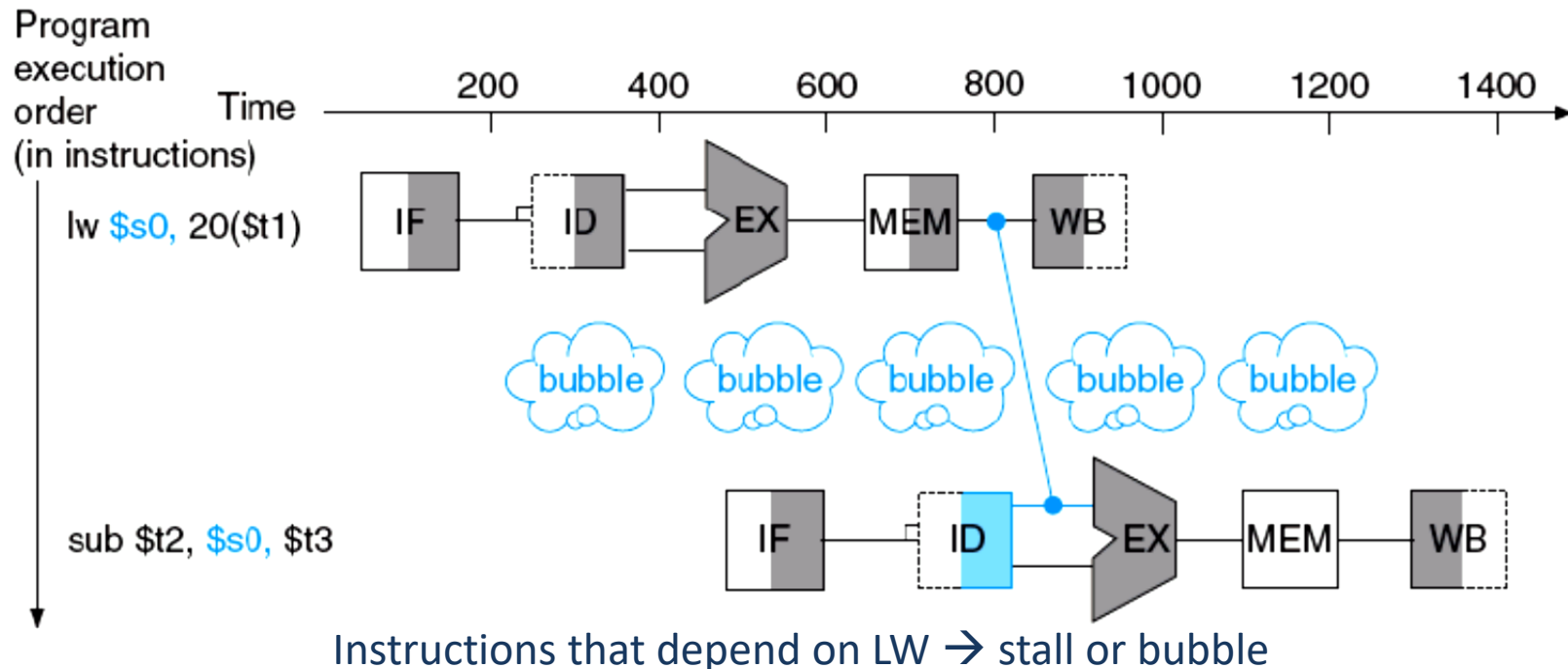


# Pipeline CPU Design – Hazards



- Data Hazard and Stalls

- Some hazards can only be resolved by **stalling** – forwarding does not help



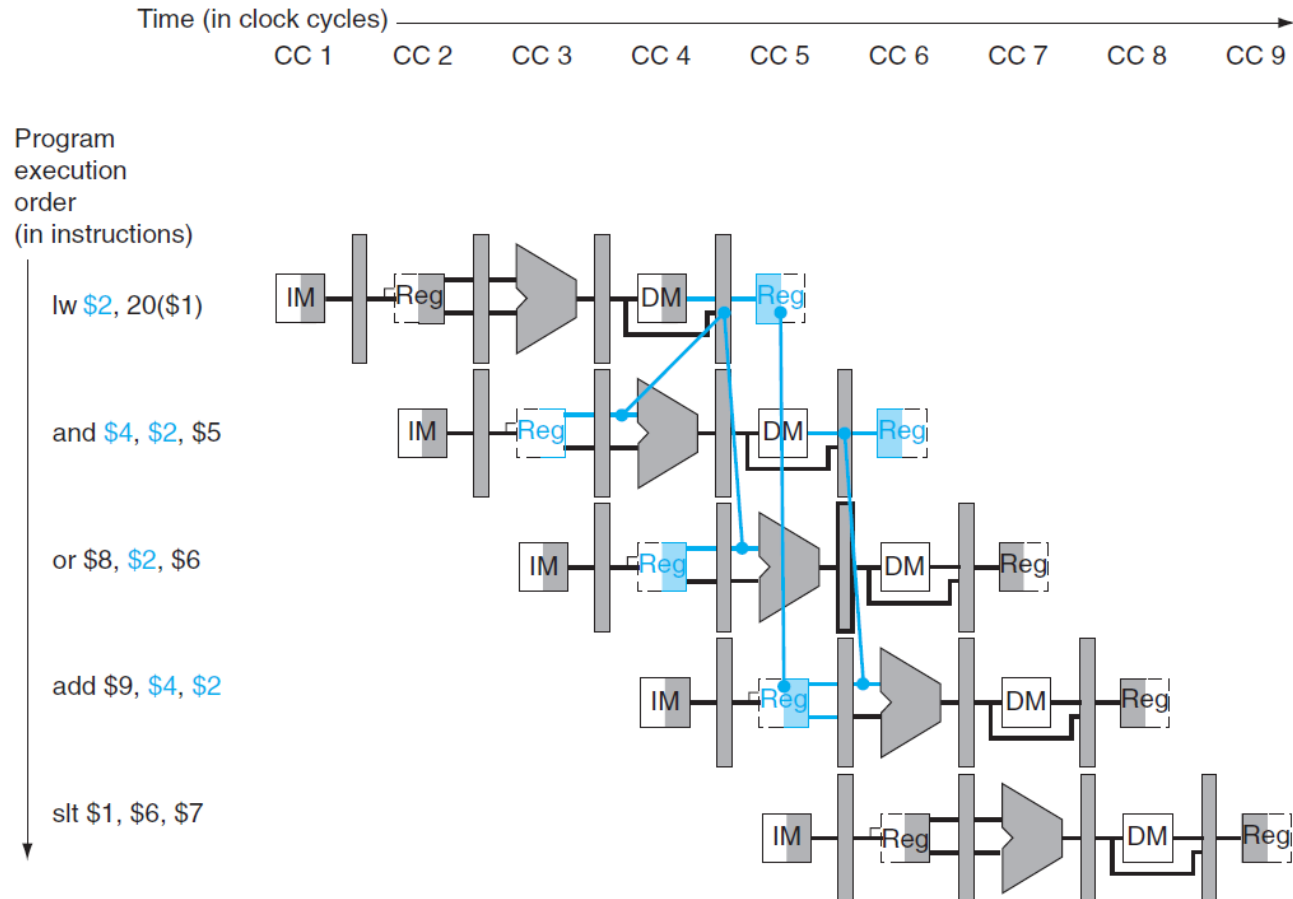
- Forwarding is not possible until the data is read from memory
- LW can cause hazards
  - The instruction that follows after a LW reads the register written by LW
- Named: Load Data Hazard



# Pipeline CPU Design – Hazards



- Data Hazard and Stalls



The hazard between LW and AND cannot be solved by forwarding, AND has to wait

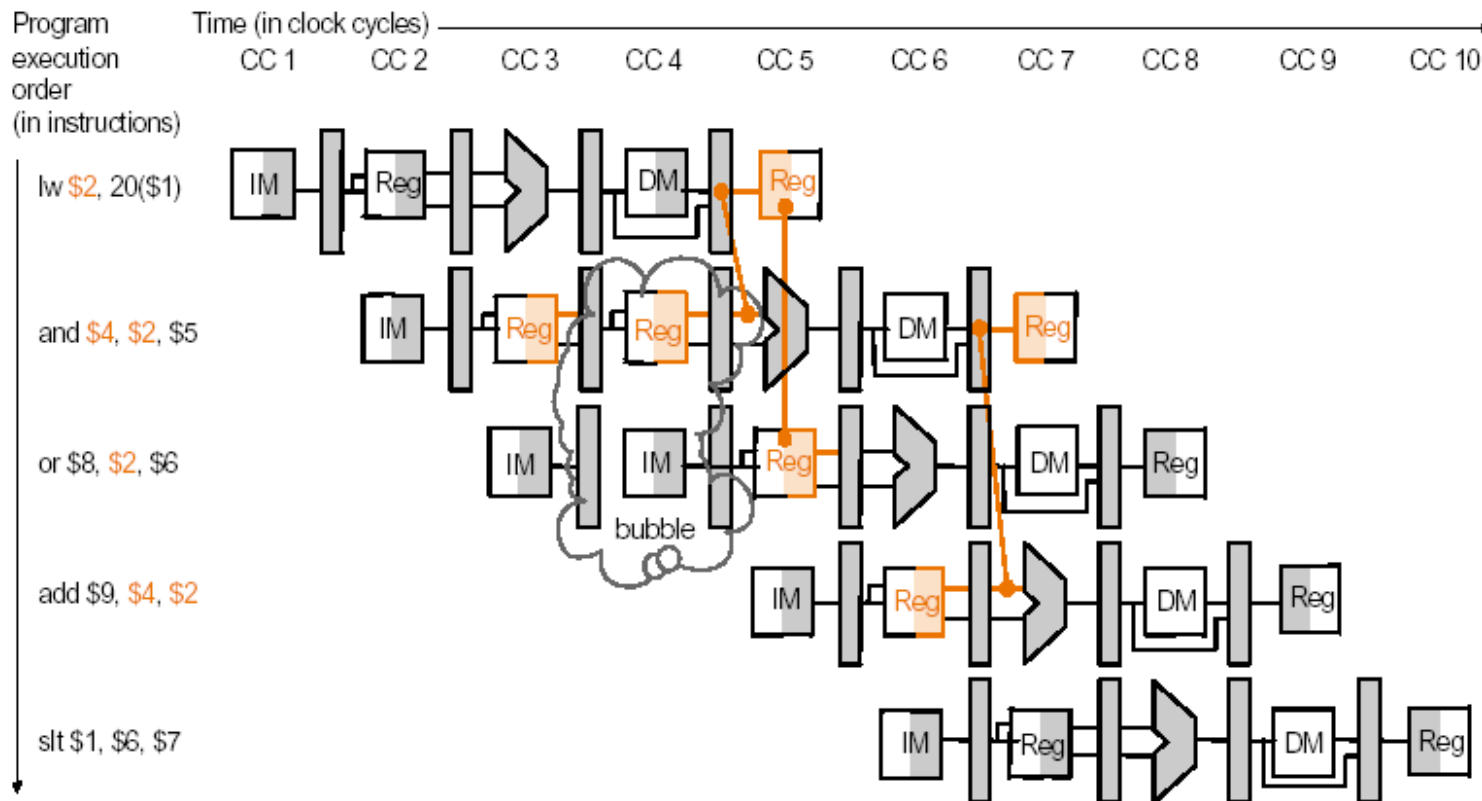
**Solution:** We introduce a wait cycle (bubble / stall / NoOp) in cc4 → NOP instead of AND



# Pipeline CPU Design – Hazards



- Data Hazard and Stalls



For Load Data Hazard, even if forwarding is applied for LW, we need a stall (bubble)

- The instruction dependent on LW must be stalled
- A **Hazard Detection Unit** in ID stage to insert a “stall” between the LW & AND instructions
- The hardware equivalent to introducing a NoOp instruction after the LW



- Data Hazard and Stalls

- Hazard Detection Unit for LW (in ID stage)

- Verifies if the instruction in EX stage is LW
- If yes and if the destination of the LW instruction is one of the sources for the instruction in the ID stage, then the pipeline will be blocked in the IF and ID stages (the instructions are “on hold”, delayed with one clock cycle)

If (ID/EX.MemRead

and ((ID/EX.RegisterRt = IF/ID.RegisterRs)

or (ID/EX.RegisterRt = IF/ID.RegisterRt)))

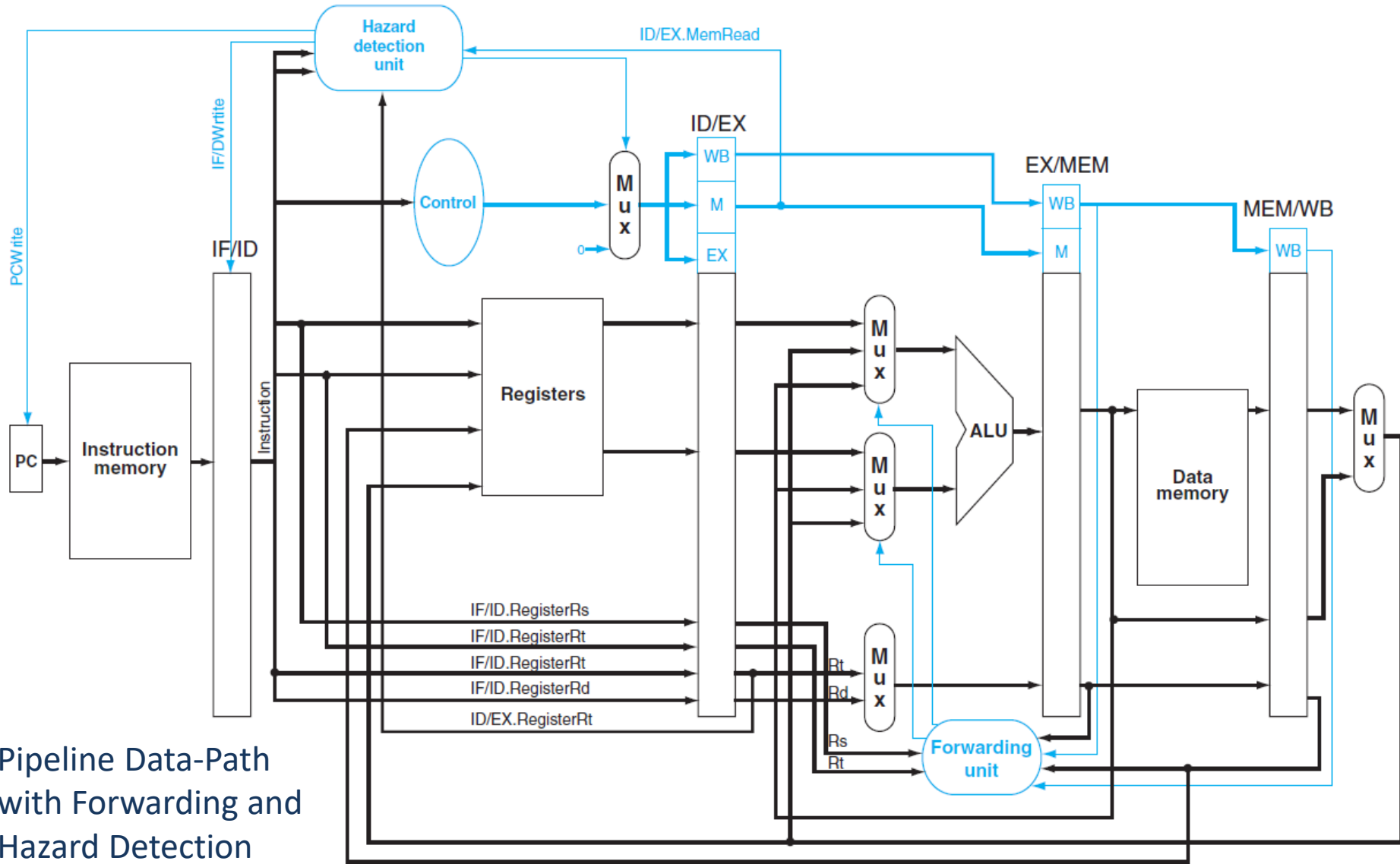
→ stall the pipeline

(LW INSTRUCTION?)

(destination in EX= source in ID?)



# Pipeline CPU Design – Hazards



Pipeline Data-Path with Forwarding and Hazard Detection



- Data Hazard and Stalls

- A NoOp (stall/bubble) implemented in hardware implies:

- to block the PC and the pipeline registers (that are located before the current instruction, i.e. ID stage) in order to hold their values for one more clock cycle;
- Force the control signals from the ID/EX register to 0 (**FLUSH**) in order to further insert the NoOp in the pipeline

- **Concrete:**

- We stall the pipeline by keeping an instruction in the ID stage: PCWrite = 0 and IF/IDWrite = 0, and
- We insert a bubble into the pipeline by writing '0' to EX, MEM and WB control fields of the ID/EX register (NoOp)
- The '0' control values are transmitted forward in the pipeline at every clock cycle  
→ no writes to registers or memory, no jump or branch



# Load Data Hazard solutions



- HW Stalls to Resolve Data Hazard
  - “Interlock”: checks for hazard & stalls
- SW inserts independent instructions
  - Worst case – inserts NoOP instructions
  - MIPS I solution: No HW checking
- Compiler solutions to avoid Load Hazards
  - Compiler will detect data dependency and inserts NoOp instructions until data is available
  - Compiler will find independent instructions to fill in the *Load-delay slots*
  - Software Scheduling can help to Avoid Load Data Hazards
    - reordering of the instructions



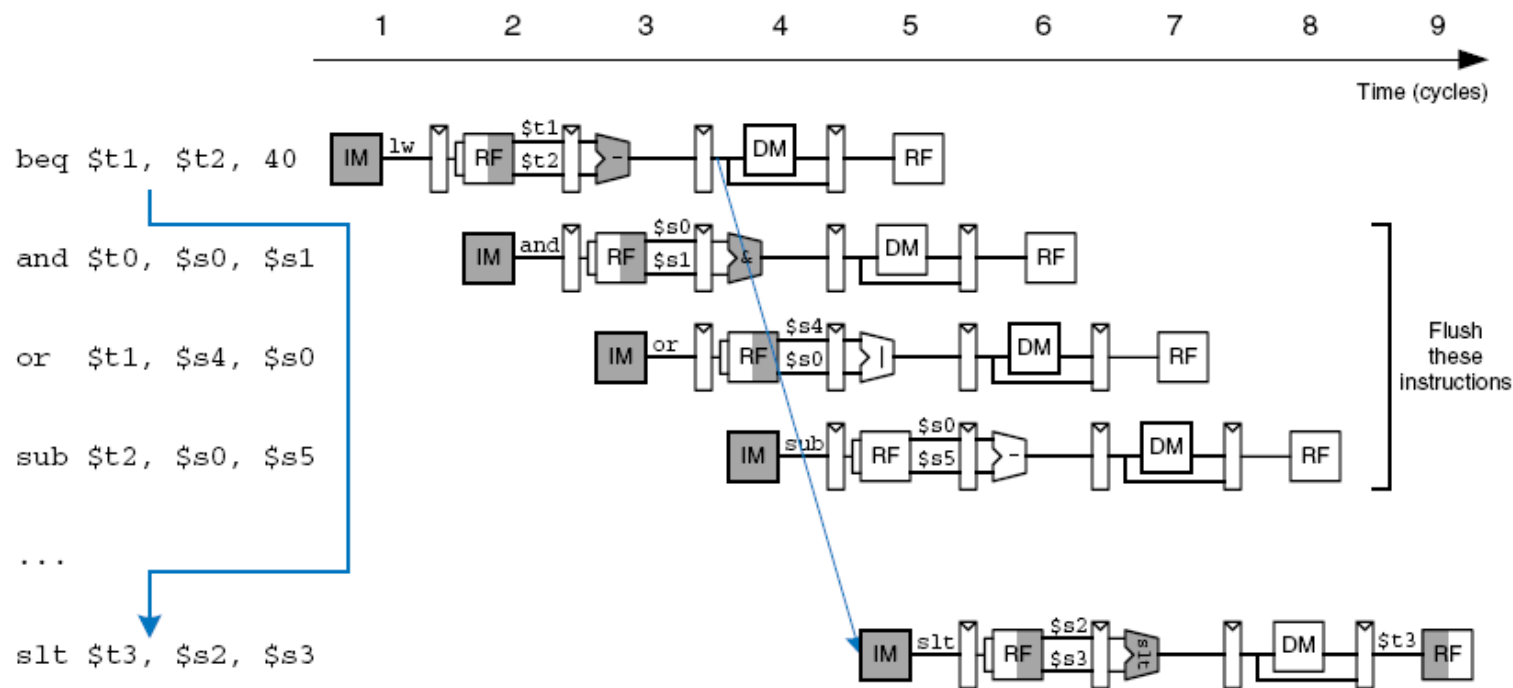


# Pipeline CPU Design – Hazards



- Control Hazards (Branch, Conditional Jumps)

- Branch decision only in **MEM Stage**
- The next 3 instructions after branch are already in IF, ID, EX stages
- If branch is taken these instructions **must not write the results!**



BEQ condition is evaluated in cc4, if it is true:

- The new PC value will be written in cc5 → slt

What can be done to reduce the number of instructions?



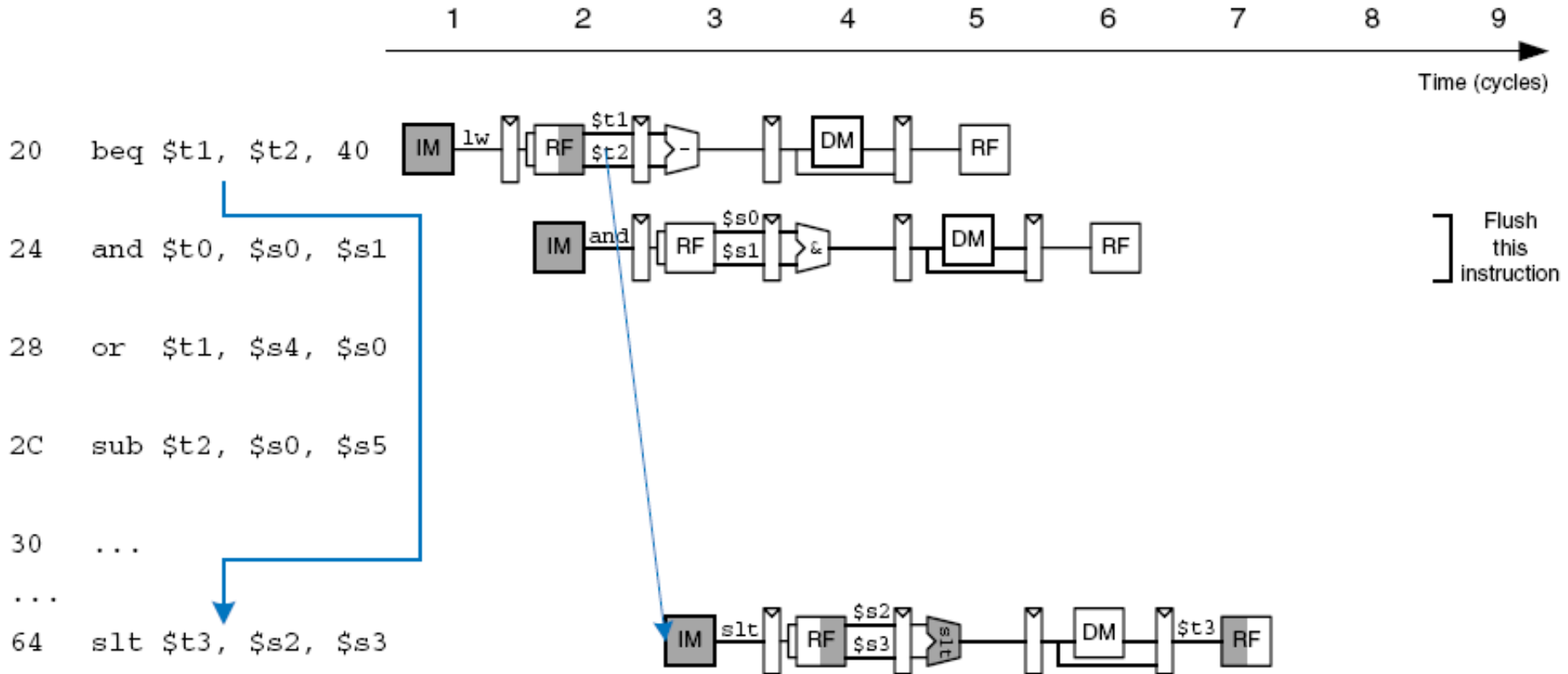
- Control Hazards (Branch, Conditional Jumps)
  - Branch instruction in pipeline
    - We are predicting “branch not taken” (execute the next instructions)
    - Need to add hardware for flushing instructions if we are wrong
  - Clever design techniques can reduce the delay to **ONE instruction**
  - Move the branch execution hardware earlier in the pipeline, fewer instructions need to be flushed
  - In the simple MIPS pipeline we have selected the next PC for a branch in MEM stage
  - We can move the Branch address calculation and condition detection in the ID stage → Only one instruction in the pipeline after a Branch
  - Branch condition test in ID implies additional forwarding and hazard detection, the branch can be dependent on a result still in pipeline
  - Forward data from EX/MEM or MEM/WB pipeline registers
  - Stall if a data hazard that can not be resolved by forwarding occurs (LW before the branch)
  - **Branch Delay now 1 clock cycle!**



# Pipeline CPU Design – Hazards



- Control Hazards (Branch, Conditional Jumps)

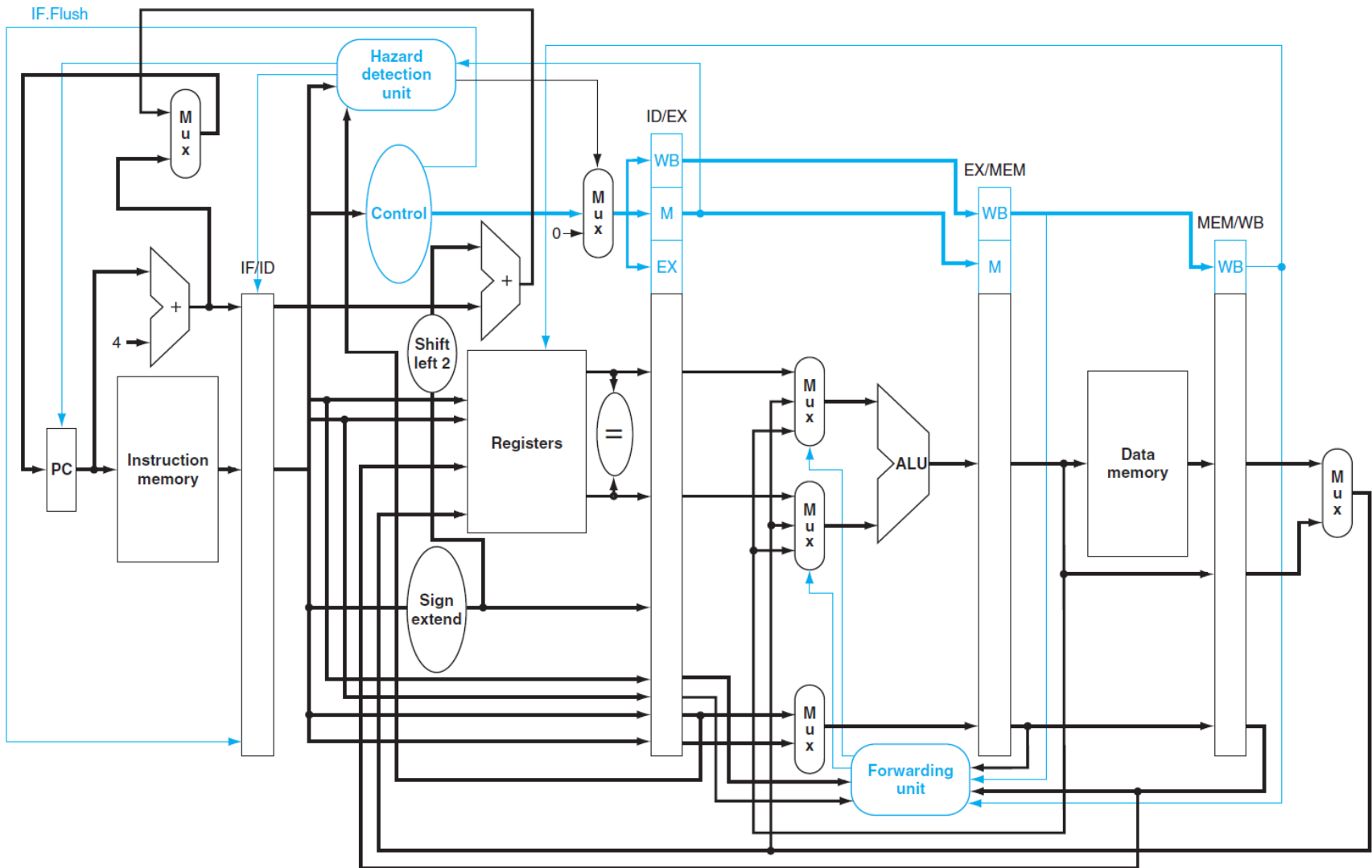


Branch resolved in ID stage, only one instruction enters the pipeline after a branch

We must add hardware to cancel (flush) the instruction that entered the pipeline in case the branch is taken!



# Pipeline CPU Design – Hazards



MIPS Pipeline with Branch Logic resolved in ID Stage



- Handling Hazards (Summary)

- Flush

- Cancels an instruction by transforming it into a NoOp (at the hardware level) by annulling the next pipeline register
- The canceling can be complete – reset the pipeline register, or minimal by setting to '0' the Write control signals for the Register File and Data Memory
- Flush for branch: canceling the next instruction after the branch – Clear the IF/ID register (NoOp)

- Stall – Bubble

- keeps the content of the pipeline registers for “younger” instructions (located before the current instruction) and wait for the “older” instructions (located after the current instruction) to finish
- Block the writing in the previous pipeline registers
- Transmit '0' (NoOp) to the next pipeline stage where the depending instruction is located (flush)
  - Software example NoOp: `sll $0, $0, 0` (bubble)



- Four Branch Hazard Alternatives
  - Stall until branch decision is clear
  - Predict Branch Not Taken
    - Execute successor instructions in sequence
    - FLUSH instructions in pipeline if branch actually taken
    - PC+4 already calculated, so use it to get next instruction
    - Must be careful not to alter state of registers until actual branch target is known
    - Only slightly more complicated than pipeline freeze to implement
    - Compiler can modify loops to favor branches not taken
  - Predict Branch Taken
    - Treat every branch as taken
    - Begin fetch at target when the branch is decoded and target address known
    - No advantage: because target address and branch outcome are in ID
    - Only makes sense for machines that compute target address before determining branch outcome
  - Delayed Branch
    - Define branch to take place **AFTER** the following instruction



# Pipeline CPU Design – Hazards



- Branch delay of length  $n$ 
  - 1 slot delay ( $n = 1$ ) allows proper decision and branch target address in 5 stage pipeline (MIPS)
  - Define a “branch delay slot”
    - The next instruction after a branch is always executed
    - Rely on compiler to “fill” the slot with something useful
    - Worst case, Software inserts NoOp into branch delay slot
  - Static Branch Prediction Using Compiler Technology – Delayed branch
    - The instruction in the delay slot (there is only one delay slot) is executed
    - If the branch is not taken, execution continues with the instruction after the branch-delay instruction
    - If the branch is taken, execution continues at the branch target.
    - When the instruction in the branch delay slot is also a branch, the meaning is unclear.
    - Because of this confusion, architectures with delay branches often disallow putting a branch in the delay slot



# Pipeline CPU Design – Hazards



- Delayed Branch implementation
  - Fetch subsequent instruction independent on branch outcome
  - The **compiler** must fill the branch delay slot with a valid/useful instruction
    - Otherwise a NoOp is used
  - Where to find instructions to fill the branch delay slot?
    - Before branch instruction
    - From the target address: only valuable when branch taken
    - From fall through: only valuable when branch not taken

Scheduling Strategy	Requirements	Improves Performance: When?
From before	Branch must not depend on the rescheduled instructions	Always
From target	Must be ok to execute rescheduled instructions if branch is not taken. May need to duplicate instructions	When branch is taken. May enlarge programs if instructions are duplicated
From fall through	Must be ok to execute instructions if branch is taken	When branch is not taken.

Delayed-branch scheduling schemes and their requirements

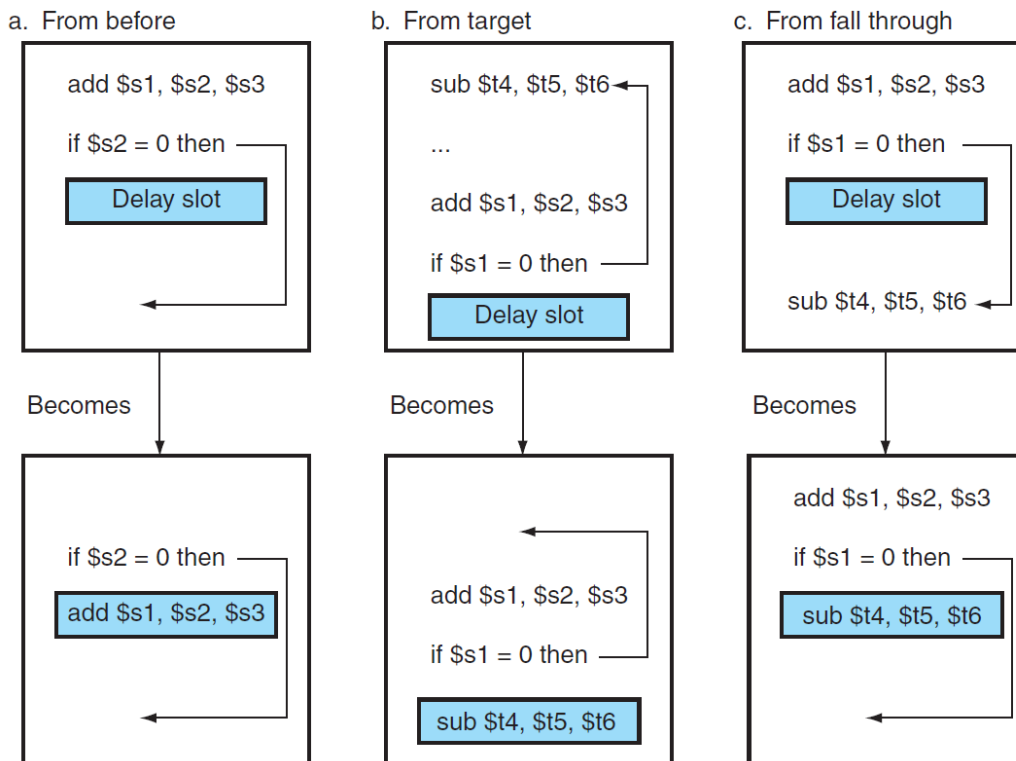




# Pipeline CPU Design – Hazards



- Delayed Branch implementation



## Static Prediction (Compiler) of conditional branches: Taken / Not Taken

- Improves strategy for placing instructions in delay slot
- Two strategies: **Backward** branch predict taken, **forward** branch not taken
- Profile-based prediction: record branch behavior, predict branch based on prior runs



## Problems – Homework



- Implementing instructions in pipeline – same as previous lectures
- Draw the pipeline diagram with and without forwarding for the following code sequences:

<pre>lw \$6, 4000(\$7) add \$9, \$6, \$3 or \$2, \$9, \$6 lw \$2, 2000(\$2) add \$3, \$9, \$2 sw \$9, 2000(\$3)</pre>	<pre>loop:   lw \$6, 4000(\$7)   add \$9, \$6, \$3   or \$5, \$9, \$6   lw \$2, 2000(\$5)   add \$3, \$9, \$2   subi \$5, \$5, 12   sw \$9, 2000(\$3)   bne \$9, \$0, loop</pre>	<pre>loop:   lw R10, X(R20)   lw R11, Y(R20)   subu R10, R10, R11   sw Z(R20), R10   addiu R20, R20, 4   subu R5, R23, R20   bnez R5, LOOP   nop; 1 delay slot</pre>	<pre>add \$r1, \$r5, \$r3 lw \$r2, 0(\$r1) add \$r1, \$r2, \$r3 sw \$r1, 0(\$r5)</pre>
---	--	--	--

- The Branch can be resolved in ID or MEM stage
- Compute the number of clock cycles for the code sequence
- Consider 2 iterations for the loop examples



# References



1. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 3<sup>rd</sup> edition, ed. Morgan–Kaufmann, 2005.
2. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5<sup>th</sup> edition, ed. Morgan–Kaufmann, 2013.
3. MIPS32™ Architecture for Programmers, Volume I: “Introduction to the MIPS32™ Architecture”.
4. MIPS32™ Architecture for Programmers Volume II: “The MIPS32™ Instruction Set”.