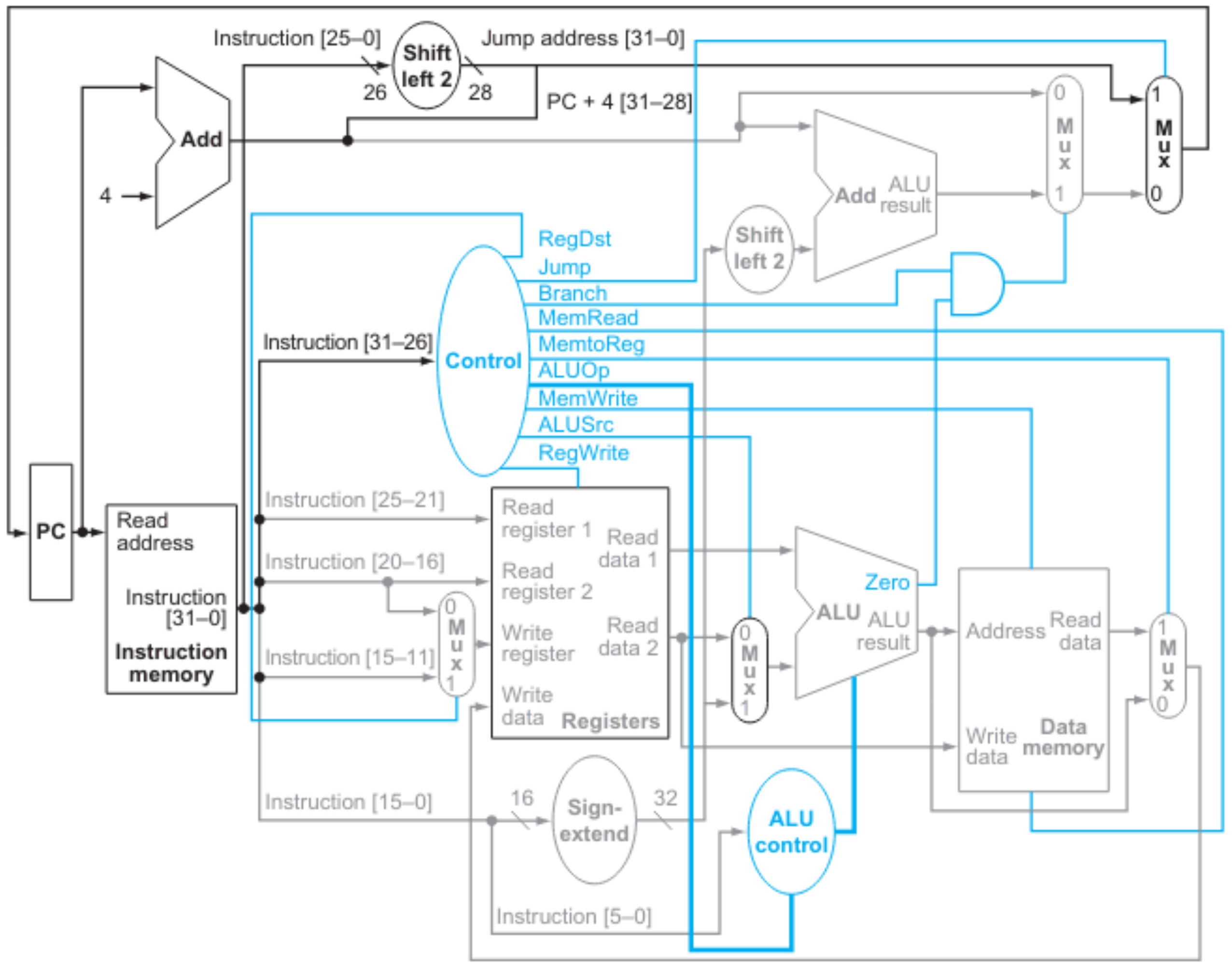
CA Tut 9

Single Cycle Datapath



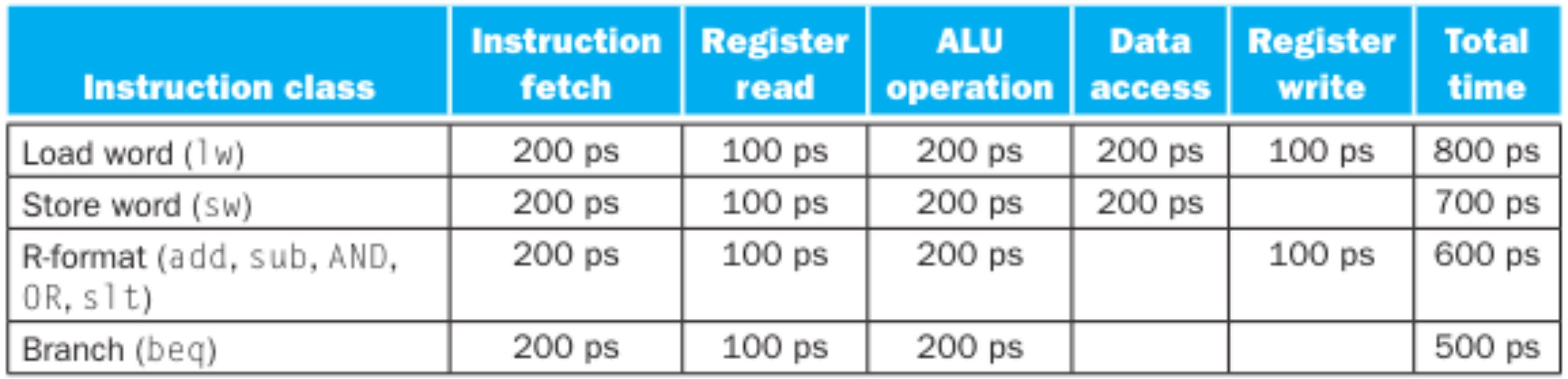
Single Cycle Datapath

- Not used nowadays
- The single cycle datapath has a fixed length for the clockcycle, which is the longest time taken by an instruction to get executed, which is the load
- The load passes through all of the phases of execution cycle: fetch, decode, execute, memory, writeback.
- Thus, the clock cycle length will have to be too long.

Pipelining

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
- Throughput= No. of tasks executed/ time.
- Pipelining doesn’t decrease the time to execute a single task but rather increases the performance by performing multiple tasks in parallel.

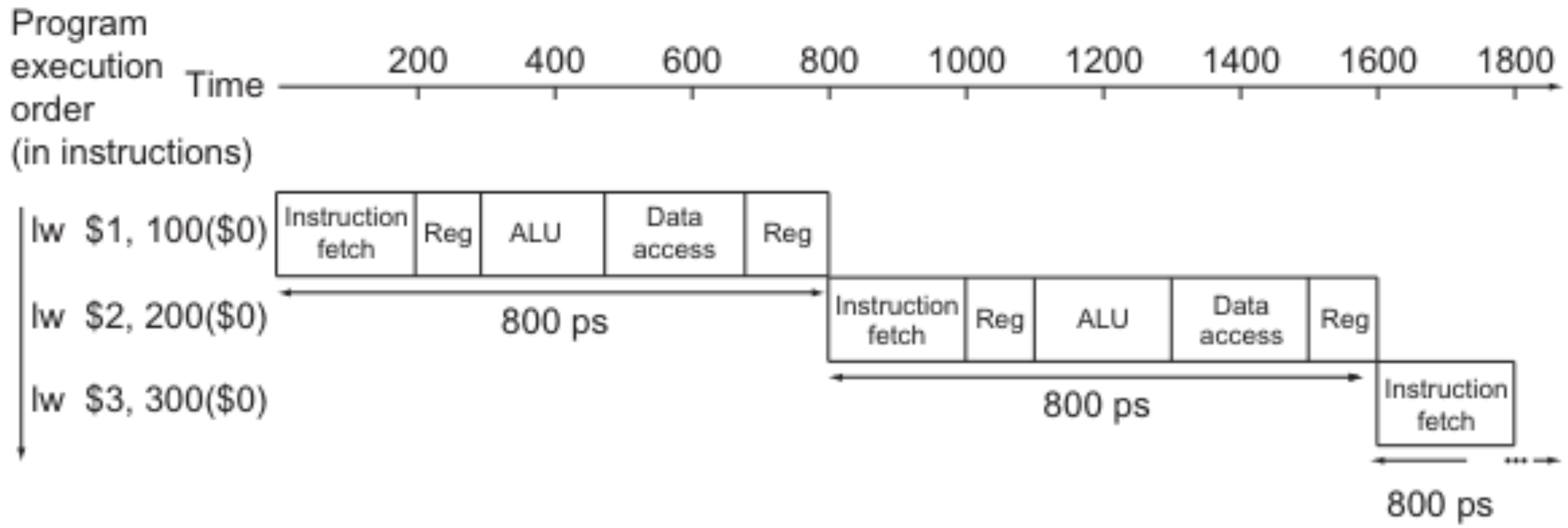
Single Cycle Vs Pipelining



Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

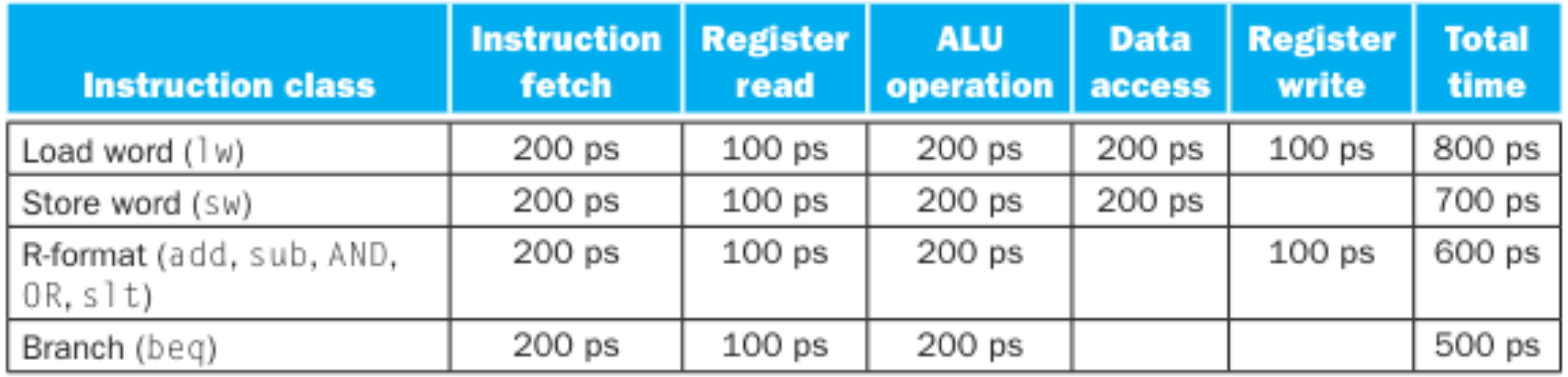
The load here takes 800 ps, the longest instruction to execute, so in a single cycle datapath, the clock cycle is 800 ps

Single Cycle Execution



Total exec time (Unpipelined)= 3*800=2400 ps

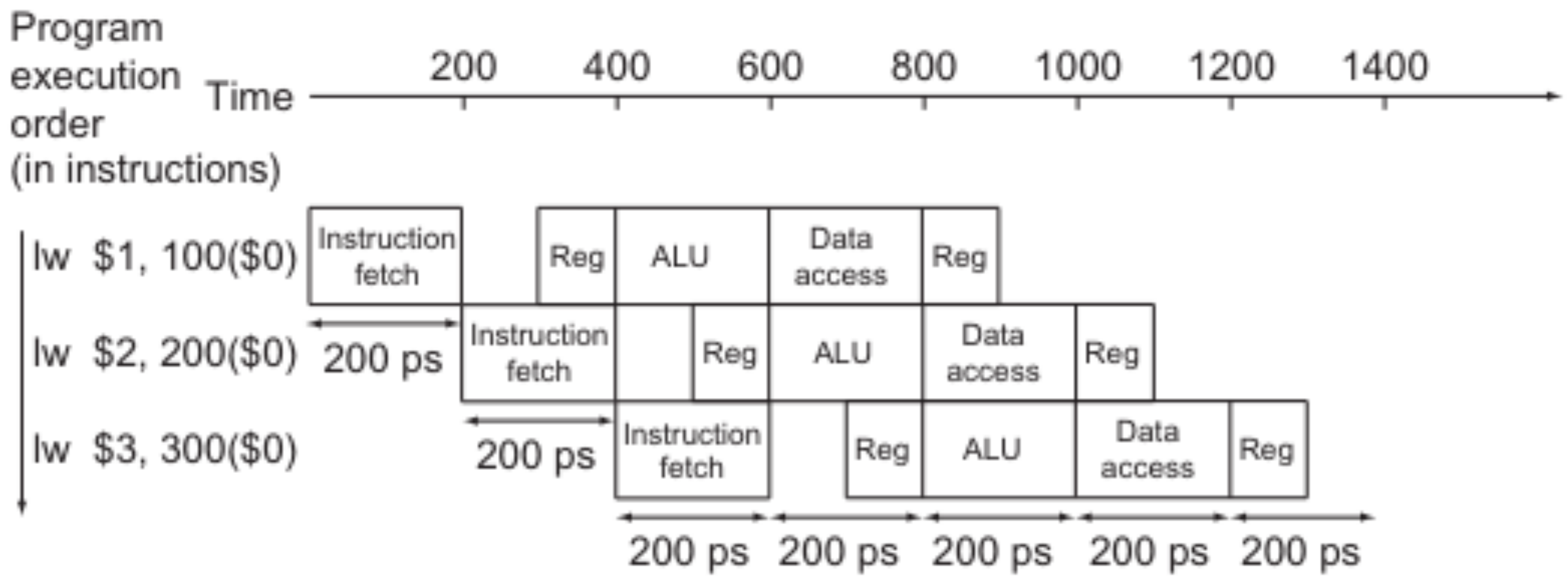
Single Cycle Vs Pipelining



Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

- In pipelining, the clock cycle’s length is the longest time taken to execute a stage, in this case it would be 200ps.

Pipelined execution



Pipeline Hazards

- There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards, and there are three different types.
  - Structural Hazards: Happens when two instructions are trying to access the same resource in the same clock cycle. In Von Neumann architectures, this does happen when an instruction is fetched from memory and another instruction is trying to access memory to read/write data.

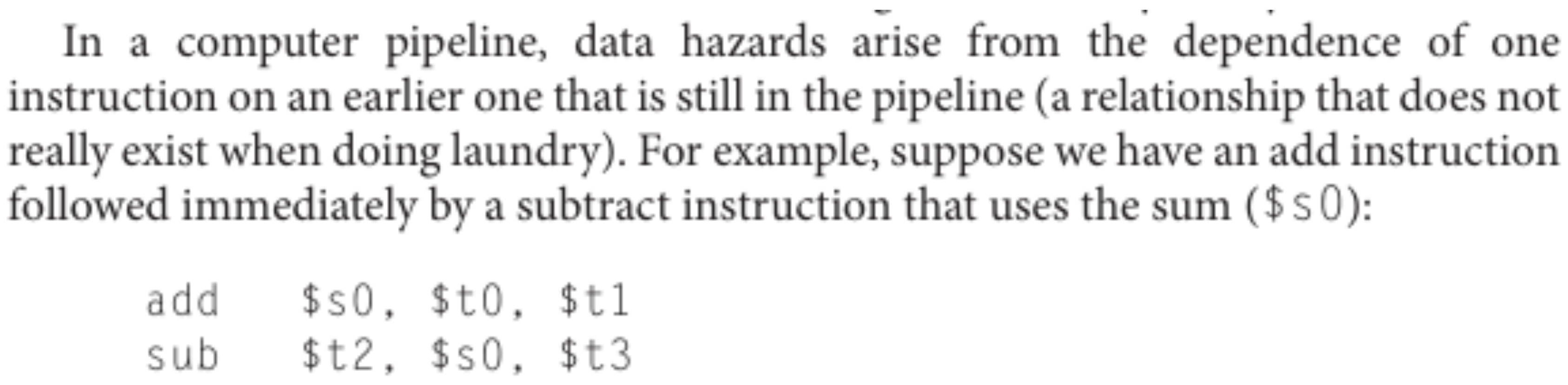
  Does this happen in MIPS??????

Pipeline Hazards

- MIPS has both a Harvard implementation and a Von Neumann Implementation.
- A structural hazard can only occur in Von Neumman Implementation
- As there is a unified memory, containing both data and instructions
- So in a single cycle, either an instruction or an operand can be accessed

Pipeline Hazards

- Data Hazards: occur when the pipeline must be stalled because one step must wait for another to complete



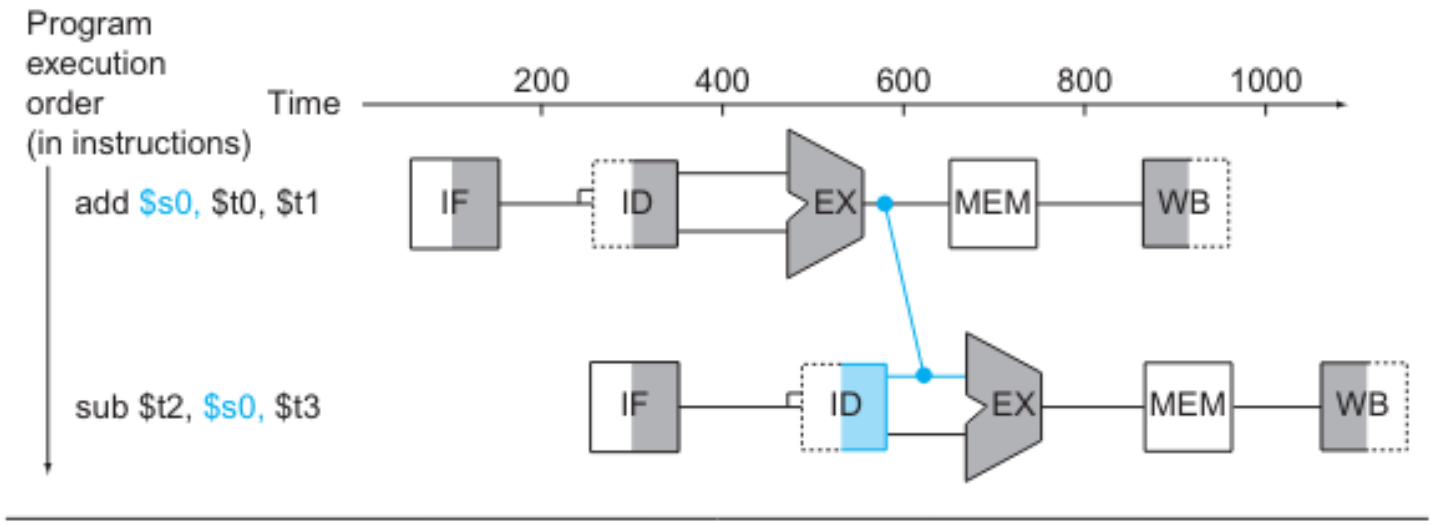
In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline (a relationship that does not really exist when doing laundry). For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum ($s0):

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

- Solution: Forwarding

Forwarding

- Supply the result from the alu for so directly as the second operand to the ALU in the next clock cycle, instead of waiting for the alu to write back the new value of s0 to the reg file.

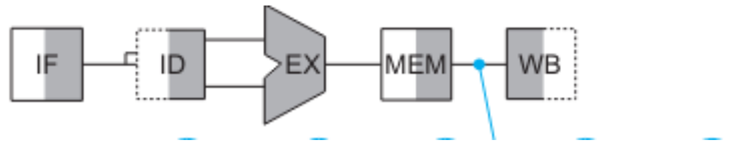


Forwarding with stall

lw $s0, 20($t1)

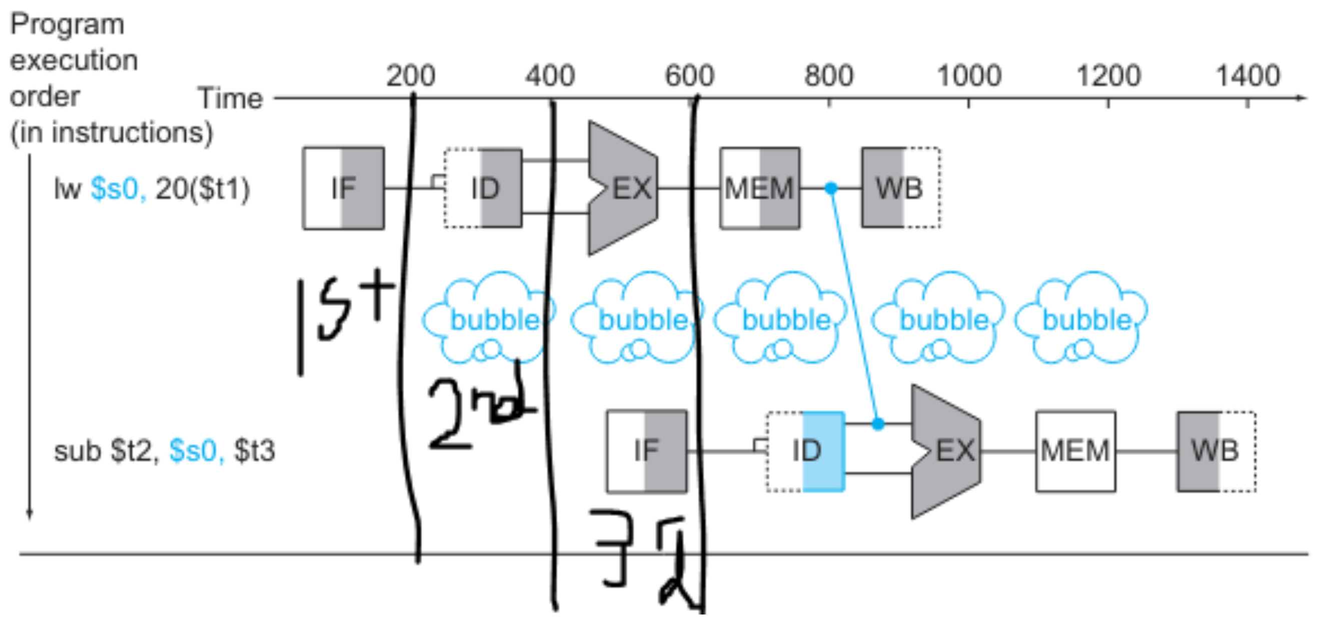
sub $t2, $s0, $t3

- Here s0’s new value is written from the memory not from the alu.



- So, the new value of s0 is only acquired after an extra clock cycle, so the execution of the sub instruction must be stalled for an extra clock cycle.

Forwarding with stall



- Here the sub instruction was fetched in the third clock cycle of the overall program execution instead of the second cycle, this is because to forward the new value of s0 from the memory directly to the ALU in the proper clock cycle.

Control Hazards

- Control hazards arise from the need to make a decision based on the results of one instruction while others are executing.
- Typically involved with branch instructions.
- Notice that we must begin fetching the instruction following the branch on the very next clock cycle.
- Nevertheless, the pipeline cannot possibly know what the next instruction should be, since it only just received the branch instruction.

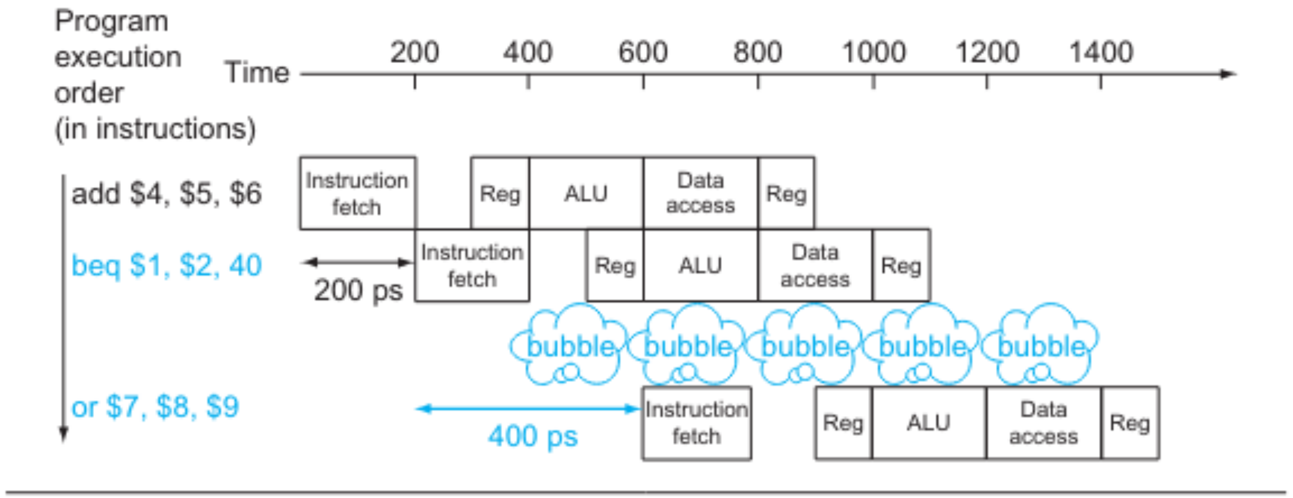
Control Hazards

- Given the following program:

```
Add $4, $5, $6
Beq $1, $2 ,40
LW   $3,300($0)
Add $1,$6,$7
……..
……
OR $7, $8, $9            //To be branched to instruction
```

Control Hazards

- First solution: Assuming that in the decode stage where get the values from the registers we have enough hardware modules compare them and take the decision of branching or not.
- In this case we just stall by one cycle.

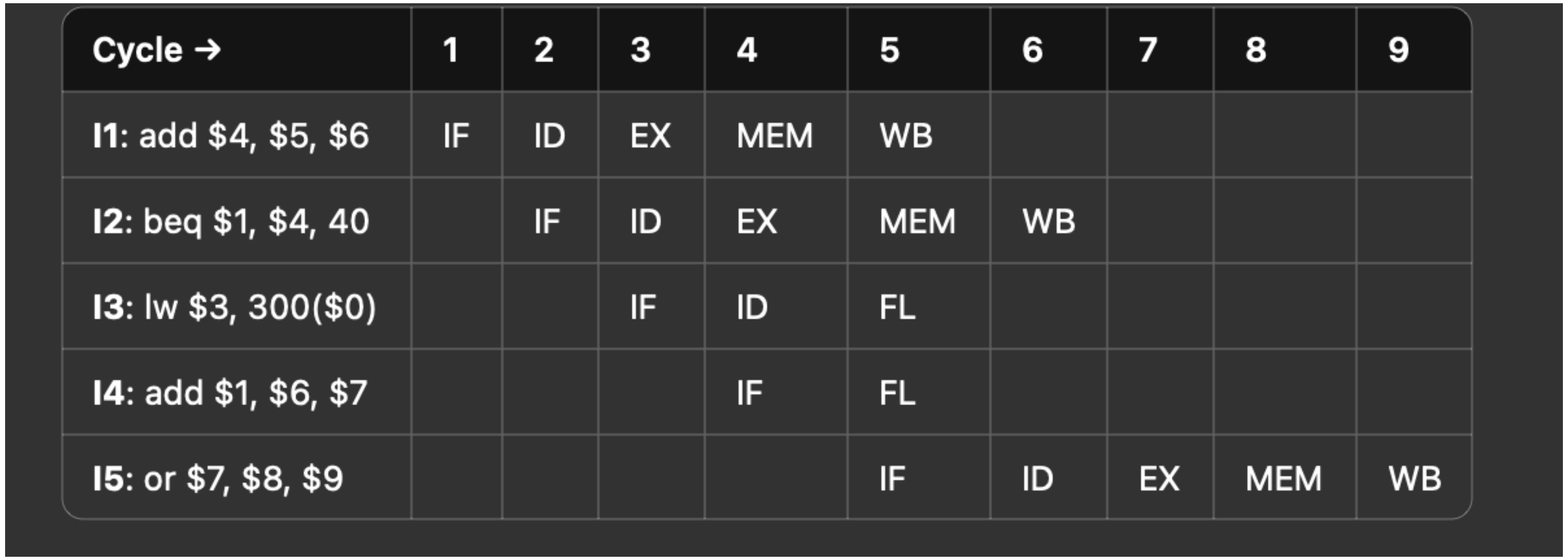


Control Hazards

- if we cannot resolve the branch in the second stage, as is often the case for longer pipelines, then we’d see an even larger slowdown if we stall on branches
- The cost of this option is too high for most computers to use and motivates a second solution, branch prediction

Branch Prediction

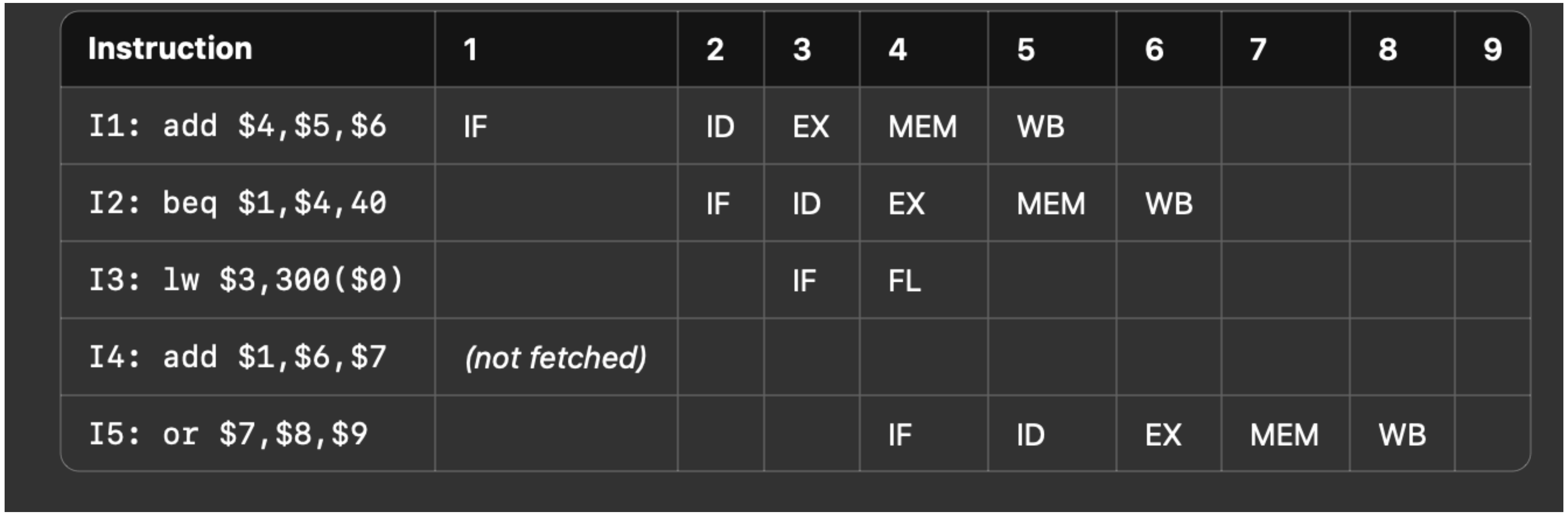
- First approach: predict that the branch will always not be taken.
- However, if it's actually taken, then whatever instructions that should be skipped, but they did enter the pipeline must be flushed out.



Cycle →	1	2	3	4	5	6	7	8	9
I1: add $4, $5, $6	IF	ID	EX	MEM	WB				
I2: beq $1, $4, 40		IF	ID	EX	MEM	WB			
I3: lw $3, 300($0)			IF	ID	FL				
I4: add $1, $6, $7				IF	FL				
I5: or $7, $8, $9					IF	ID	EX	MEM	WB

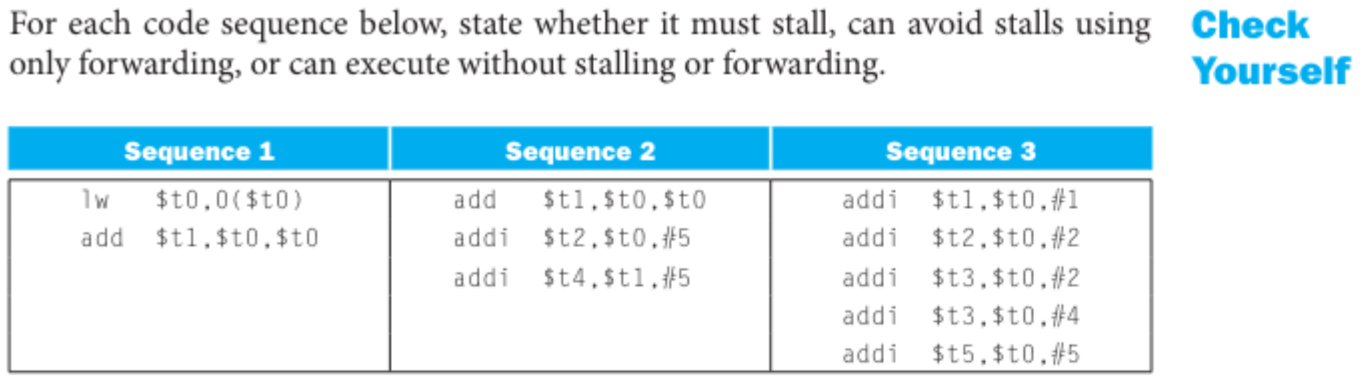
Branch Prediction + Hardware module

- Now we use branch prediction + having the hardware module that knows the decision of the branch in the decode phase.



Instruction	1	2	3	4	5	6	7	8	9
`I1: add $4,$5,$6`	IF	ID	EX	MEM	WB				
`I2: beq $1,$4,40`		IF	ID	EX	MEM	WB			
`I3: lw $3,300($0)`			IF	FL					
`I4: add $1,$6,$7`	*(not fetched)*								
`I5: or $7,$8,$9`				IF	ID	EX	MEM	WB	

Exercise

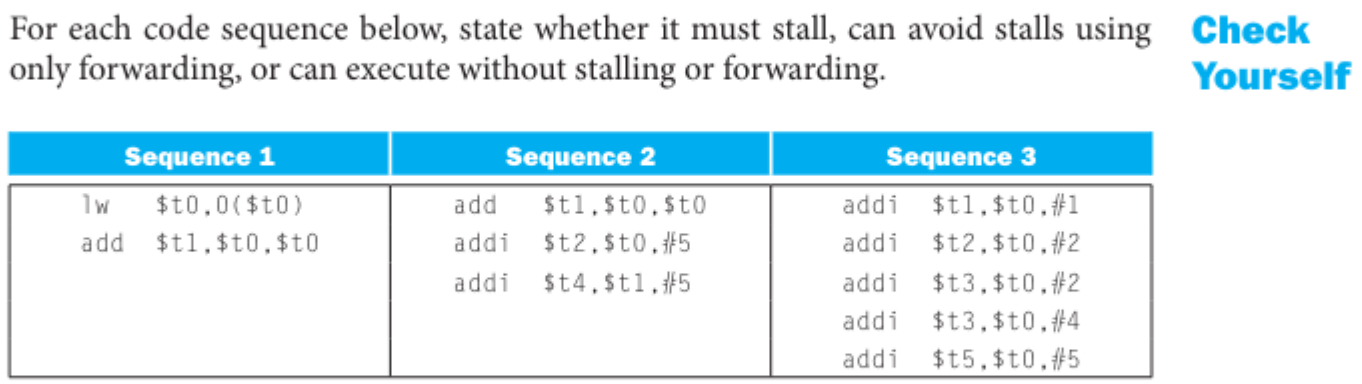


For each code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

Check Yourself

Sequence 1	Sequence 2	Sequence 3
`lw $t0,0($t0)`	`add $t1,$t0,$t0`	`addi $t1,$t0,#1`
`add $t1,$t0,$t0`	`addi $t2,$t0,#5`	`addi $t2,$t0,#2`
	`addi $t4,$t1,#5`	`addi $t3,$t0,#2`
		`addi $t3,$t0,#4`
		`addi $t5,$t0,#5`

Exercise



For each code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

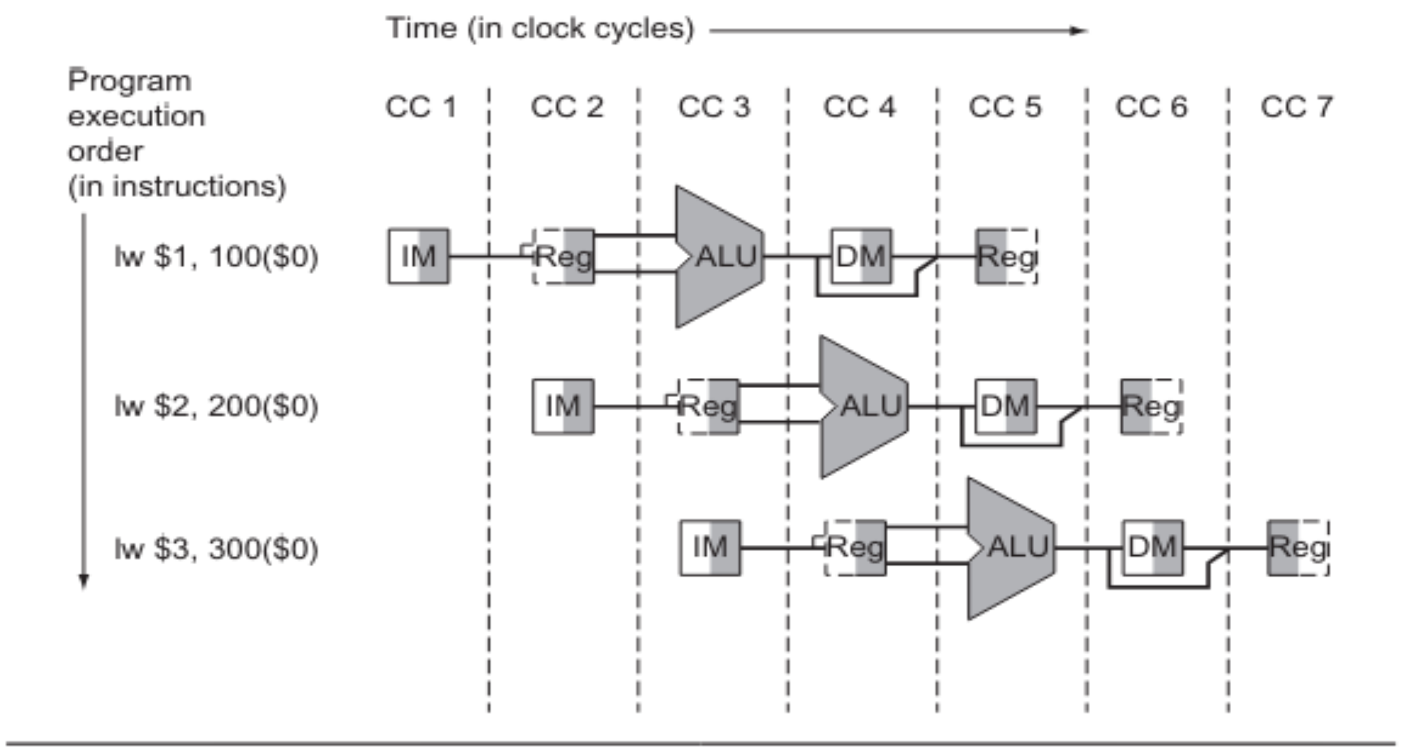
Check Yourself

Sequence 1	Sequence 2	Sequence 3
`lw $t0,0($t0)`	`add $t1,$t0,$t0`	`addi $t1,$t0,#1`
`add $t1,$t0,$t0`	`addi $t2,$t0,#5`	`addi $t2,$t0,#2`
	`addi $t4,$t1,#5`	`addi $t3,$t0,#2`
		`addi $t3,$t0,#4`
		`addi $t5,$t0,#5`

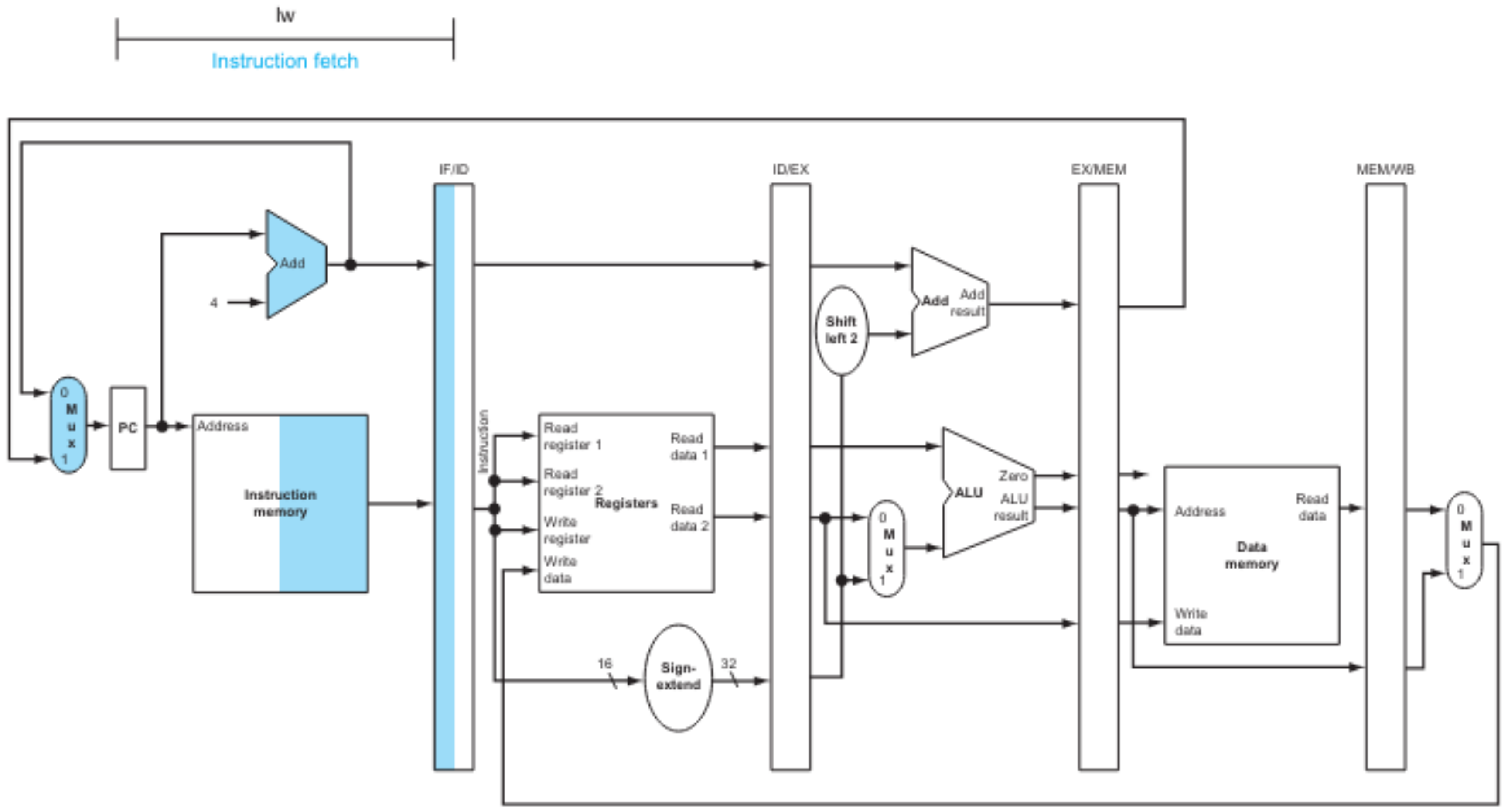
1. Forwarding with stalling
2. Forwarding only
3. Execute normally.

Pipelining Datapath

- Registers are added between every stage and the other to retain info about the instruction moving between every two stages.
- As if each instruction has its own datapath.



Pipelining Datapath



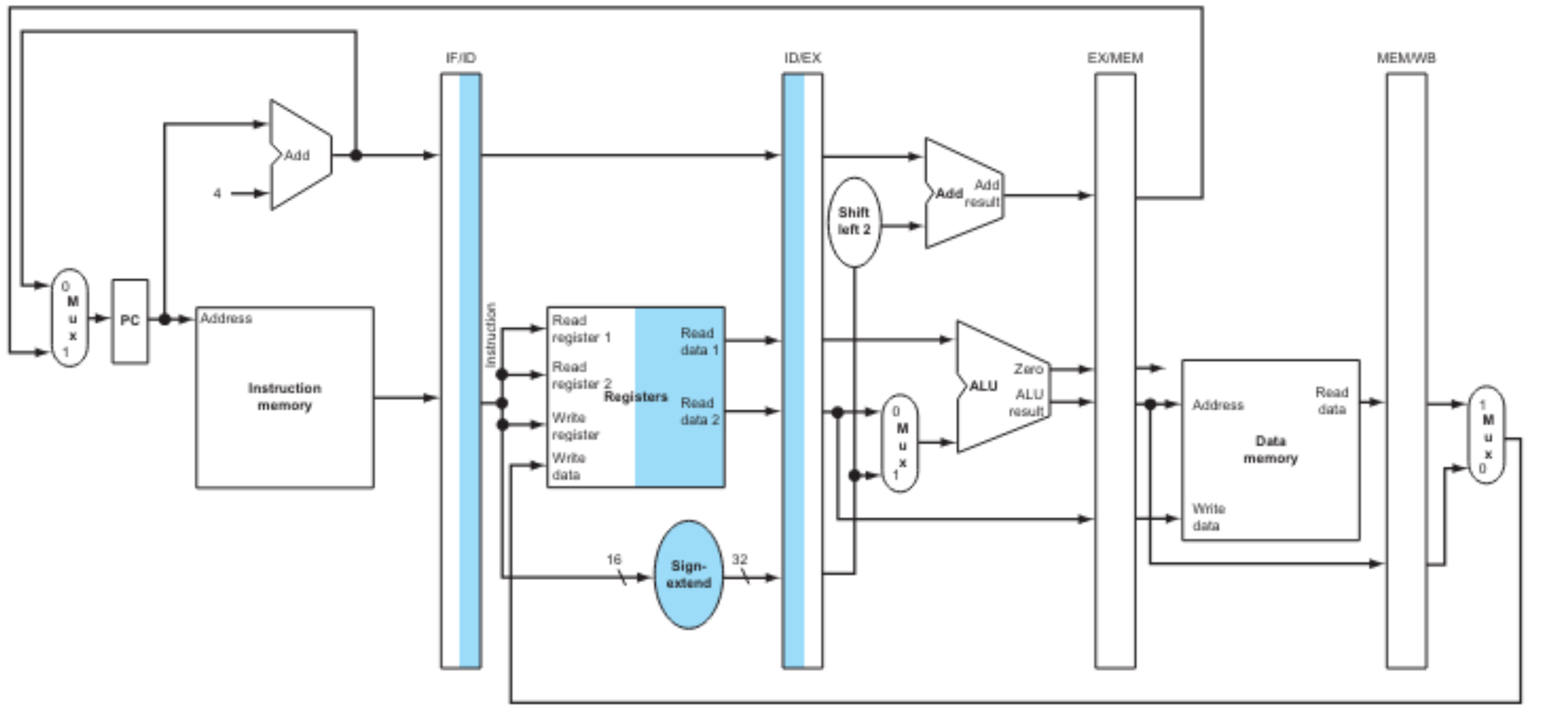
Pipelining Datapath

Fetching

- The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle
- This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq

lw

Instruction decode

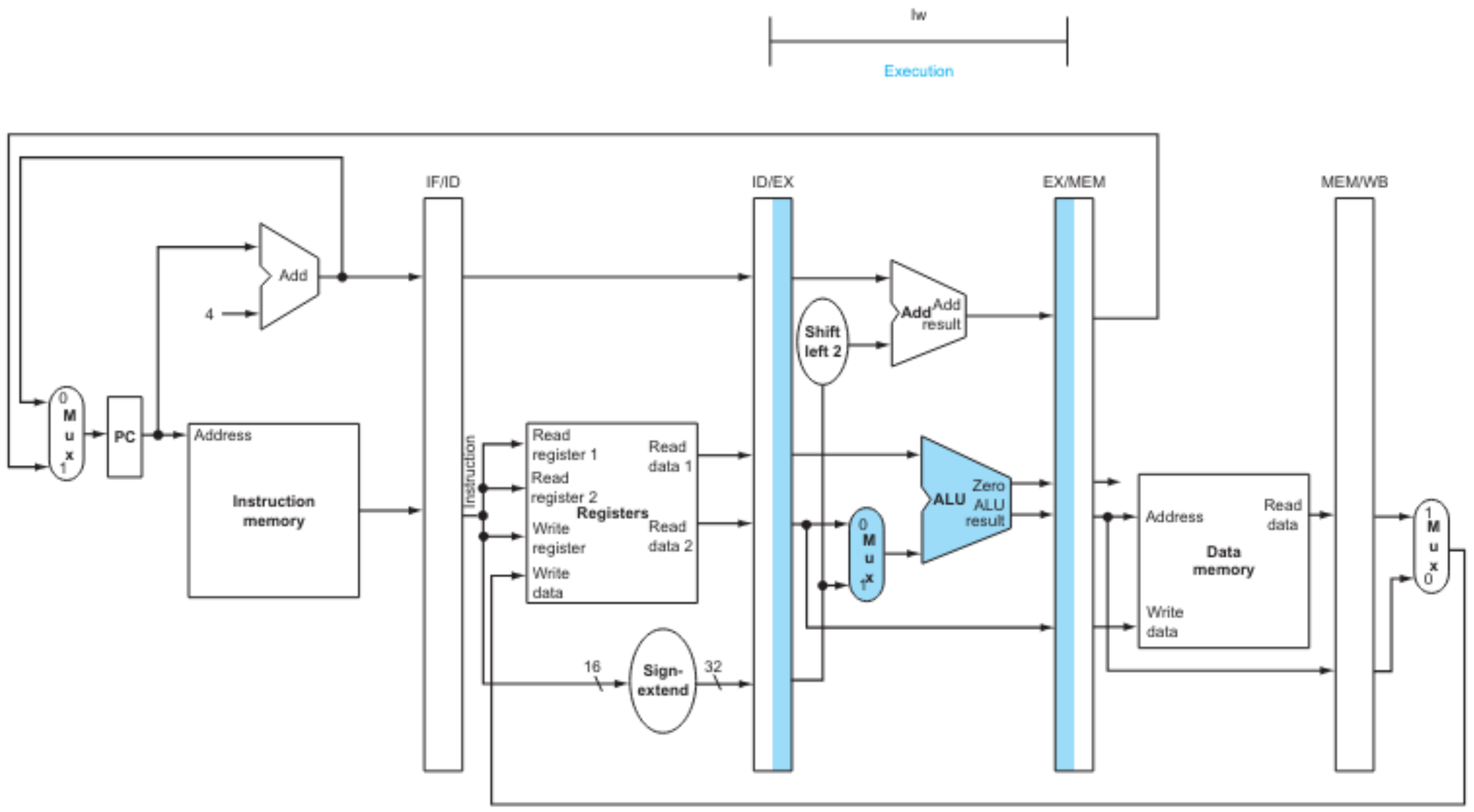


Pipelining Datapath

Decoding

- Instruction decode and register file read: IF/ID pipeline register supplies the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers.
- All three values are stored in the ID/EX pipeline register, along with the incremented PC address. We again transfer everything that might be needed by any instruction during a later clock cycle from the previous pipeline register

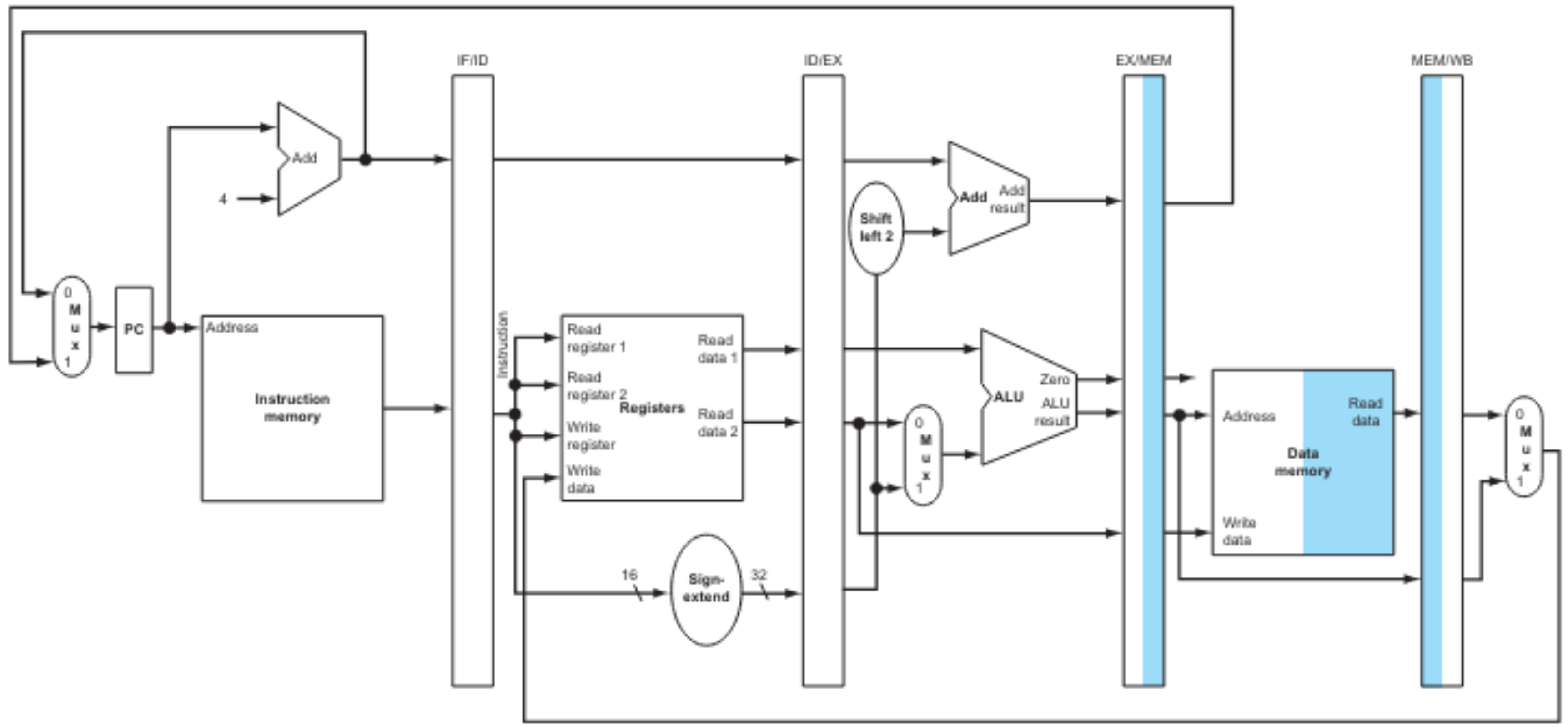
Pipelining Datapath



Pipelining Datapath

Execution

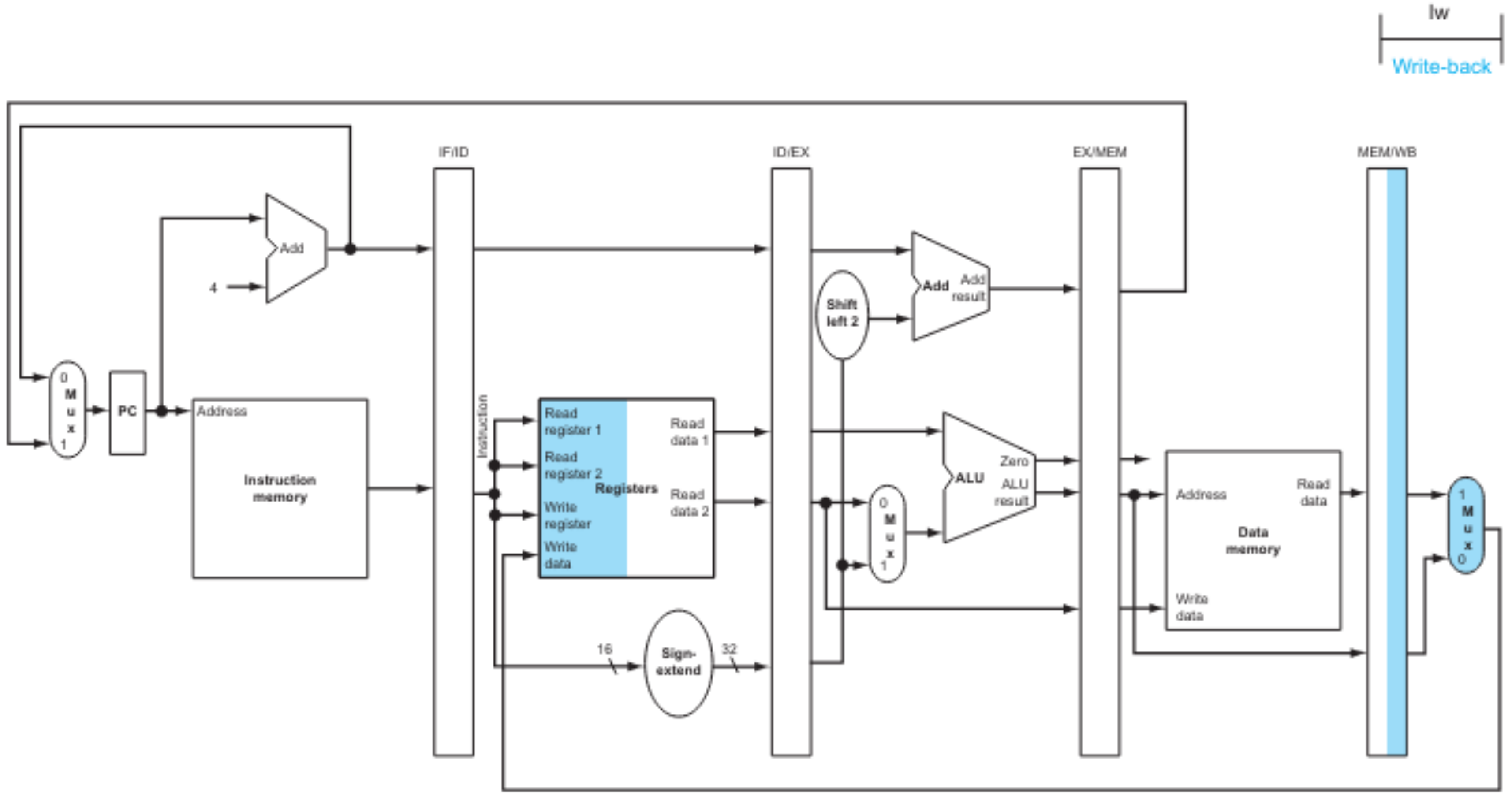
- The contents of register 1 and the sign-extended immediate are read from the ID/EX pipeline register and adds them using the ALU. The sum is placed in the EX/MEM pipeline register.
- The value of the reg in RT field is also forwarded to EX/MEM pipeline Reg in case of SW instruction.


IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
Add
Add result
Shift left 2
0 M u x 1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign-extend
32
0 M u x 1
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
0 M u x 1

Pipelining Datapath

Memory

- Load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register
- In case of R-type instruction, the all result is forwarded to the Mem/ WB pipeline register.


lw
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

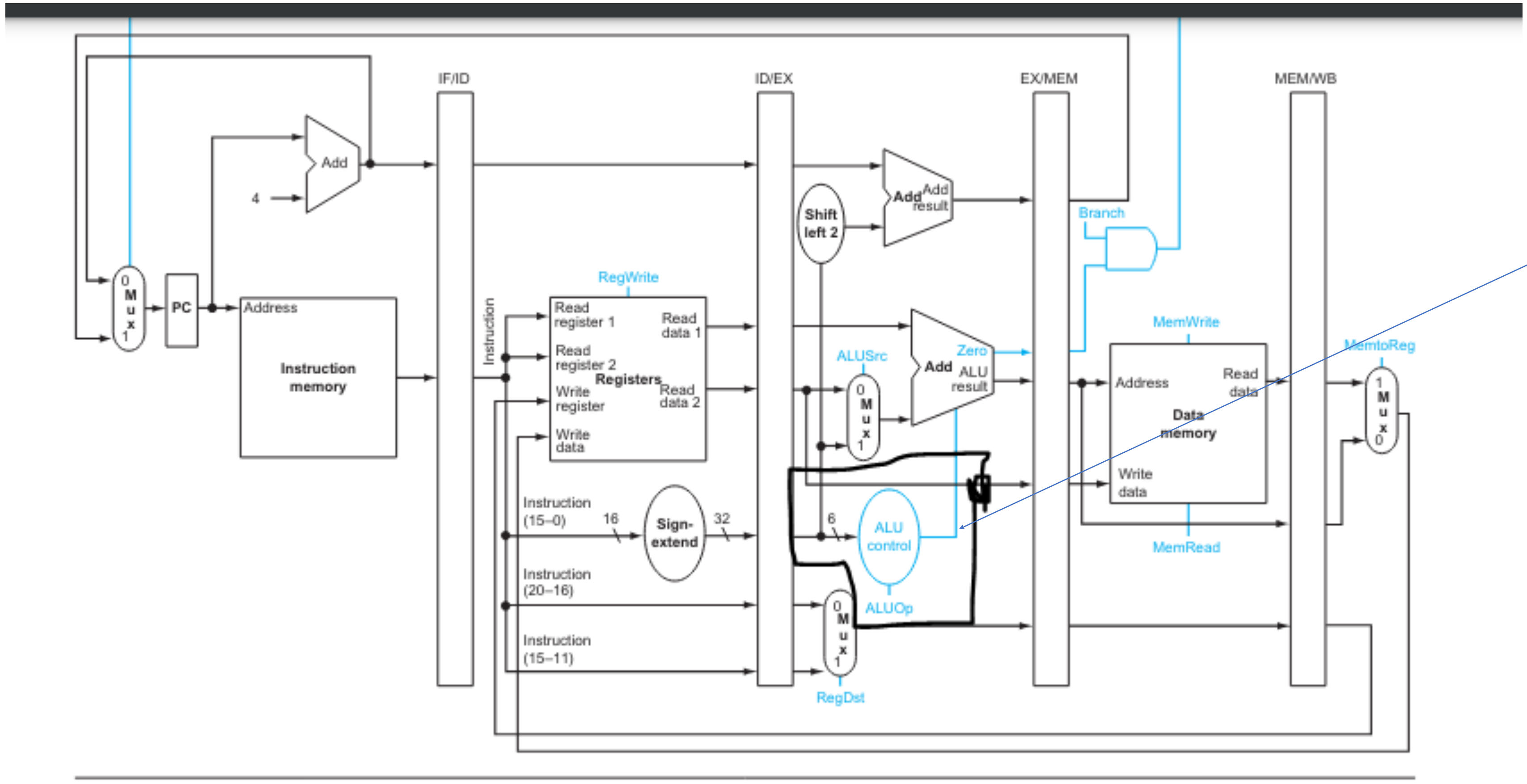
Pipelining Datapath

- Write Back

Reading the data from the MEM/WB pipeline register and writing it into the register file in

The dst reg should have been forwarded to the MEM/WB registers through every pipeline stage registers starting from the IF/ID registers.

Pipeline Control Signals



The 4 alu control bits need the ALUop + funct field from the instruction to be generated

Thus the funct needs to be included in ID/EX registers so that it can be used to determine the alu control bits

Pipeline Control Signals

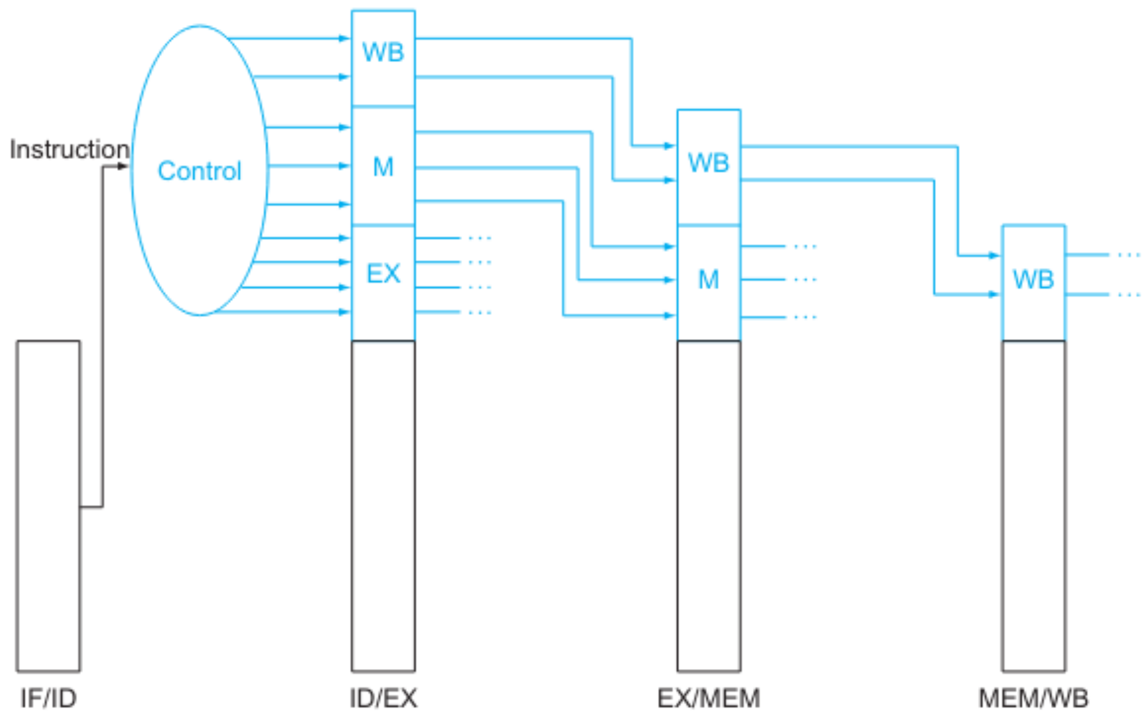


FIGURE 4.50 The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

The control signals in the pipelined datapath are outputted from the control unit and fed to ID/EX registers. Afterwards, only the signals required for upcoming stages are forwarded to the upcoming pipeline stage registers.

Pipeline Control Signals

- Forwarded to ID/EX: Alu src, Aluop, Regdst, Branch, Memread, Memwrite, MemtoReg.
- Forwarded to Ex/Mem: Memread, Memwrite, MemtoReg.
- Forwarded to Mem/WB: MemtoReg.