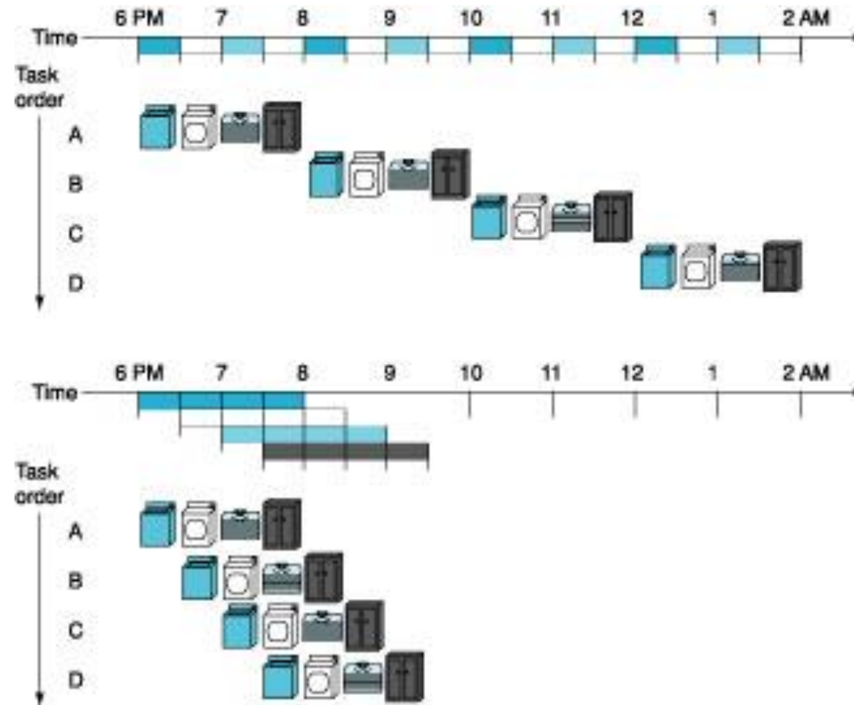
Chapter Seven

Enhancing Performance with Pipelining

6.1 An Overview of Pipelining

Example: Laundry



Pipelined laundry is four times faster than nonpipelined.

6.1 An Overview of Pipelining

- **The same principles apply to processors where we pipeline instruction execution.**
- **MIPS instructions classically take five steps:**
  1. **Fetch instruction from memory**
  2. **Read registers while decoding the instruction.**
  3. **Execute the operation or calculate an address.**
  4. **Access an operand in data memory.**
  5. **Write the result into a register.**

6.1 An Overview of Pipelining

Example: Single-Cycle versus Pipelined Performance

Compare the average time between instructions of a single-cycle implementation to a pipelined implementation. The operation time are:

200 ps for memory

200 ps for ALU

100 ps for register

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

Total time for each instruction calculated from the time for each component.

Continue

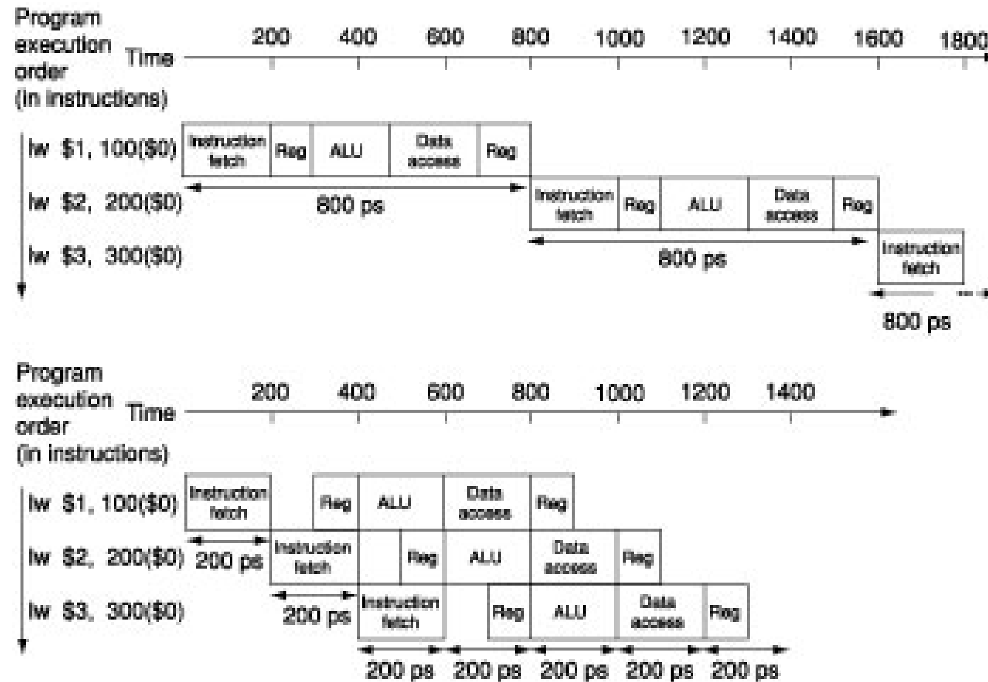


Figure 6.3 Nonpipelined and pipelined execution of three load word instructions.

The time between 1st and 4th (nonpipelined) = 3 × 800 =2400 ps.

The time between 1st and 4th (pipelined) = 3 × 200 =600 ps.

⇒ Speedup = 2400/600 = 4 < 5 ? Why? Because stages are not perfectly balanced.

The time between 1st and 2th (nonpipelined) = 800 ps.

The time between 1st and 2th (pipelined) = 200 ps.

⇒ Speedup = 800/200 = 4

Continue

- **If the stages are perfectly balanced, then:**

$$Time\ between\ instructions_{pipelined} = \frac{Time\ between\ instructions_{nonpipelined}}{Number\ of\ pipe\ stages}$$

$$= \frac{800}{5} = 160\ ps.$$

But, in Figure 6.3, clock cycle = 200 ps. not 160 ps. Why?

Moreover, for three instruction: it's 1400 ps versus 2400 ps.

$\Rightarrow$ **2400/1400=1.7 < 4 Why? because three instructions only.**

For 1,000,003 instructions:

$$\frac{1000000 \times 800 + 2400}{1000000 \times 200 + 1400} = \frac{800002400 ps}{200001400 ps} \approx 4.0 \approx \frac{800 ps}{200 ps}$$

Designing Instruction Sets for Pipelining

- What makes it easy?
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
  - Operands must be aligned in memory (a single data transfer requiring one data memory accesses).

- What makes it hard?
  - structural hazards: suppose we had only one memory
  - data hazards: an instruction depends on a previous instruction
  - control hazards: need to worry about branch instructions
- We'll build a simple pipeline and look at these issues

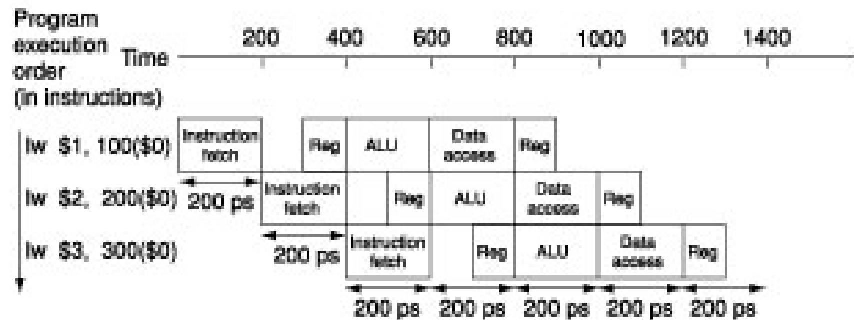
- We'll talk about modern processors and what really makes it hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

Pipeline Hazards

Hazards: when the next instruction can not executed in the following clock cycle.

- **Structural Hazards**

  The hardware cannot support the combination of instructions that we want to execute in the same clock cycle.



If we had a single memory, and if we had a fourth instruction fetched from memory ⇒ structural hazard.

Pipeline Hazards

2. Data Hazards

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

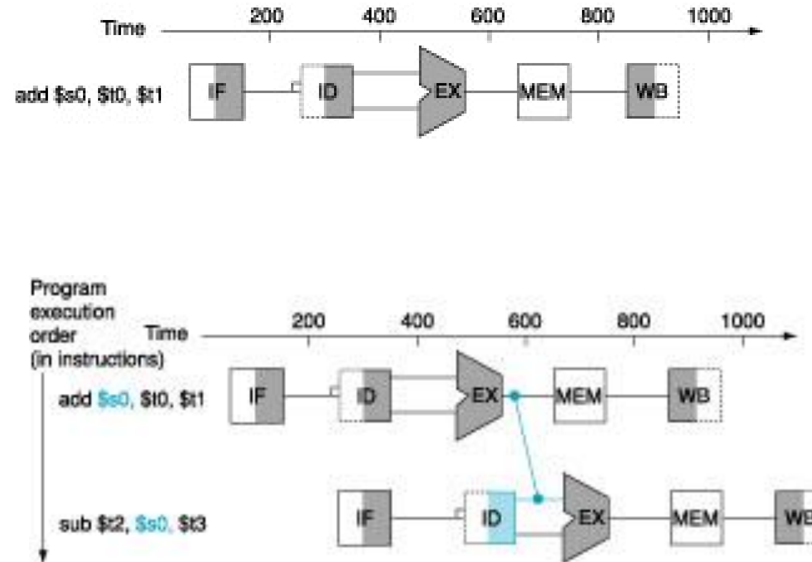
```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

- The add instruction doesn't write its result until the fifth stage ⇒ add three bubbles.
- The primary solution: forwarding or bypassing.

Example: Forwarding with Two Instructions

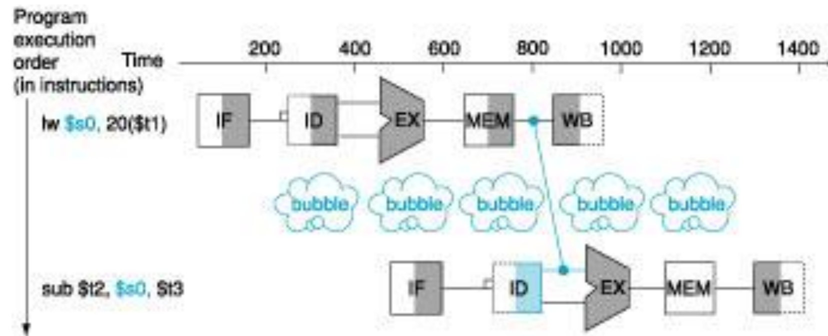
For the two instruction above, show what pipeline stage would be connected by forwarding.

Continue



- Forwarding paths are valid only if the destination stage is later in time than the source stage.
- Forwarding cannot prevent all pipeline stalls. For example, suppose the first instruction were a load of $s0 instead of an add. The desired data would be available only after the fourth stage. which is too late for the input of the third stage of sub.
- Hence, even with forwarding,, we would have to stall one stage for a load-use data hazard. see next Figure.

Continue



We need a stall even with forwarding when an R-format instruction following a load tries to use the data

Example: Reordering Code to Avoid Pipeline Stalls

Consider the following code segment in C:

```
A=B+E;
C=B+F;
```

Here is the generated MIPS code:

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($01)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($01)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

Reorder to avoid any pipeline stalls.

Pipeline Hazards

3. Control Hazards

- Arising from the need to make a decision based on the results of one instruction while others are executing.

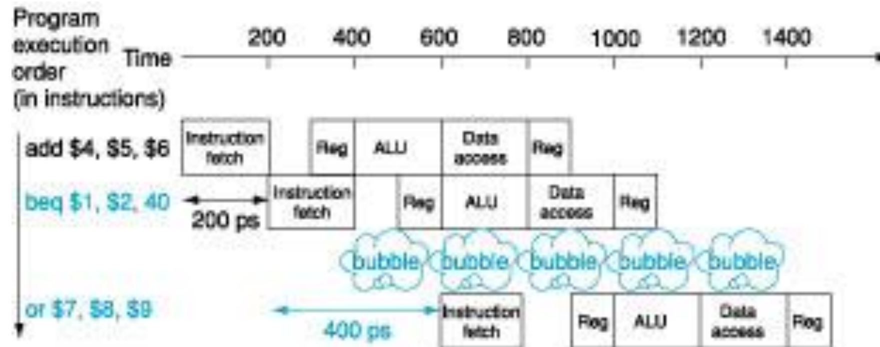
Two solutions to control hazards:

1. Stall: the cost of this option is too high
2. Predict: over 90% accuracy

Two solutions for control hazard

1. **Stall**

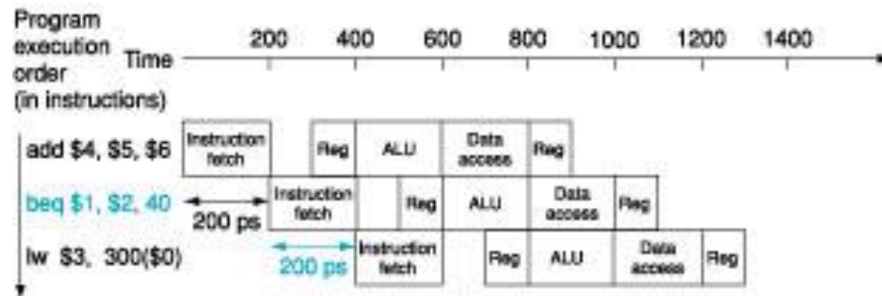
   Let's assume that we can test registers, calculate the branch address, and update the PC during the second stage of the pipeline. In the following Figure, the `lw` instruction, executed if the branch fails, is stalled one extra 200 ps clock cycle before staring.

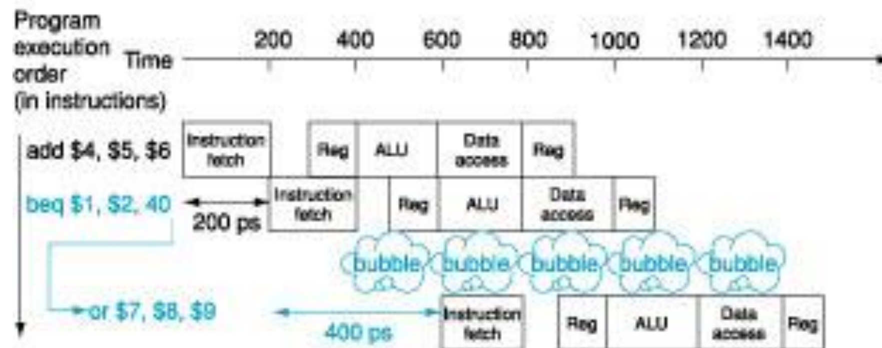


Two solutions for control hazard

2. **Predict**

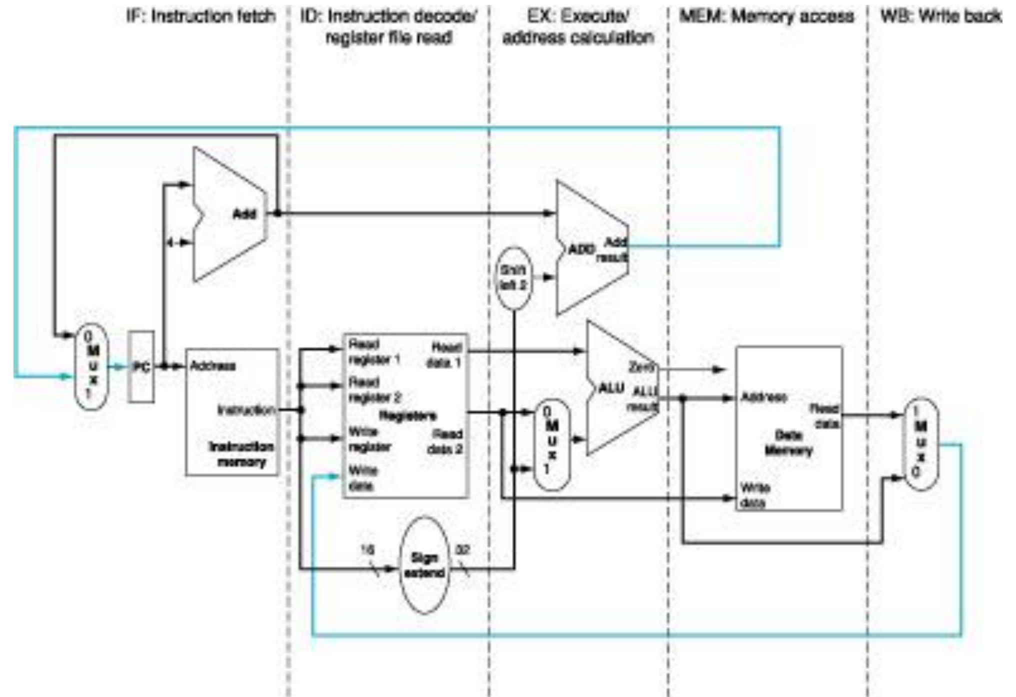
- **One simple approach is to always predict that branches will be untaken. When you're right, the pipeline proceeds at full speed. Only when the branches are taken does the pipeline stall. See next Figure.**





6.2 A pipelined Datapath

- The single-cycle datapath:

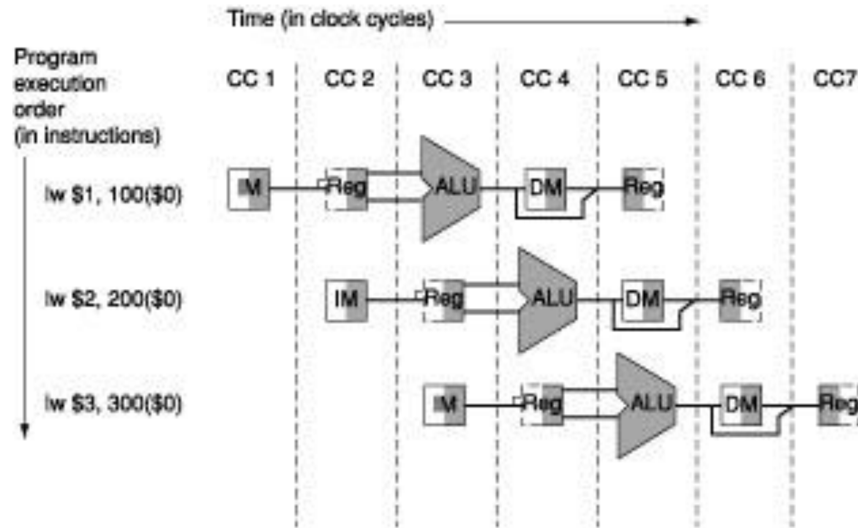


- We must separate the datapath into five pieces:
  1. IF: Instruction fetch
  2. ID: Instruction decode and register file read
  3. EX: Execute or address calculation
  4. MEM: Data memory access
  5. WB: Write back

Continue

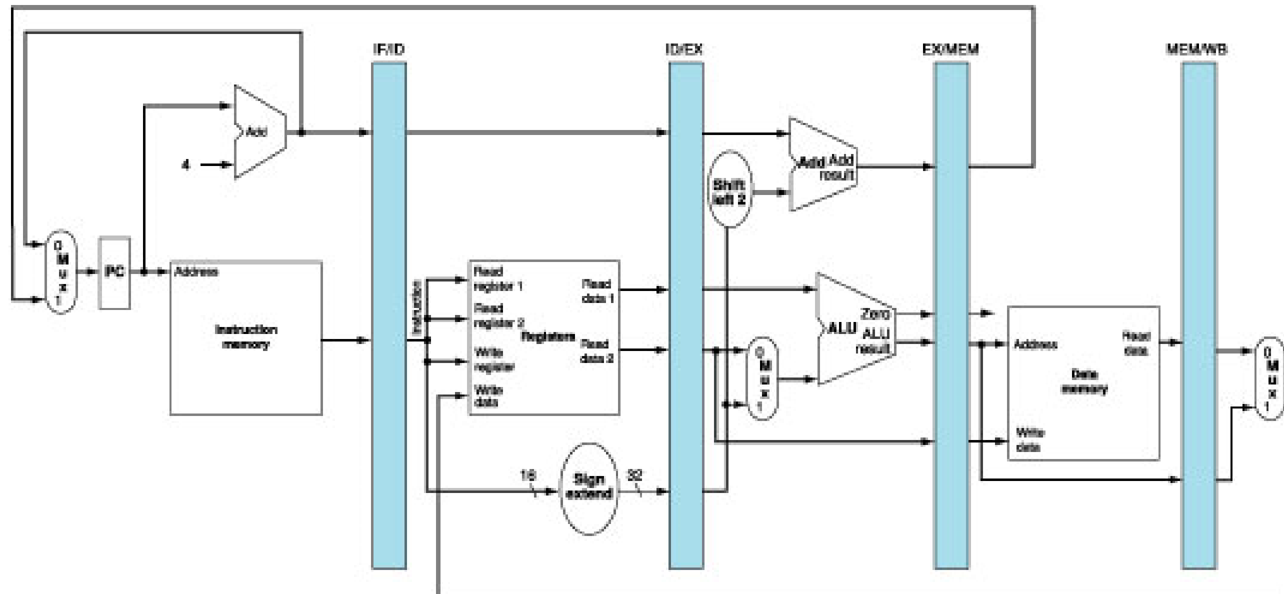
- Two exception to this left-to-right flow of instruction:
  1. The write-back stage ➔ data hazard
  2. The selection of the next value of the PC ➔ control hazard

- To show what happens in pipelined execution, pretend that each instruction has its own datapath.



Continue

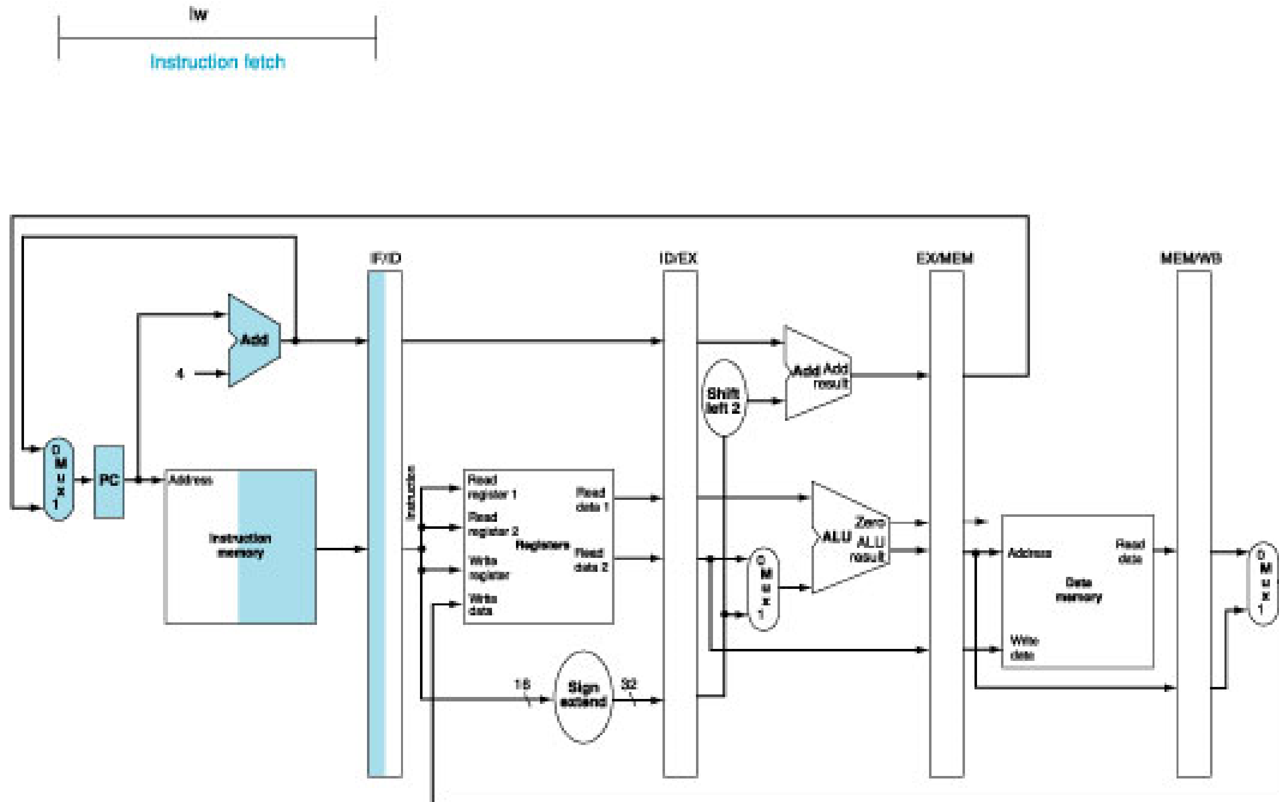
- **Use pipeline register to retain the value of an individual instruction for its other four stages.**



Continue

- **The five stages for Load Instruction are:**

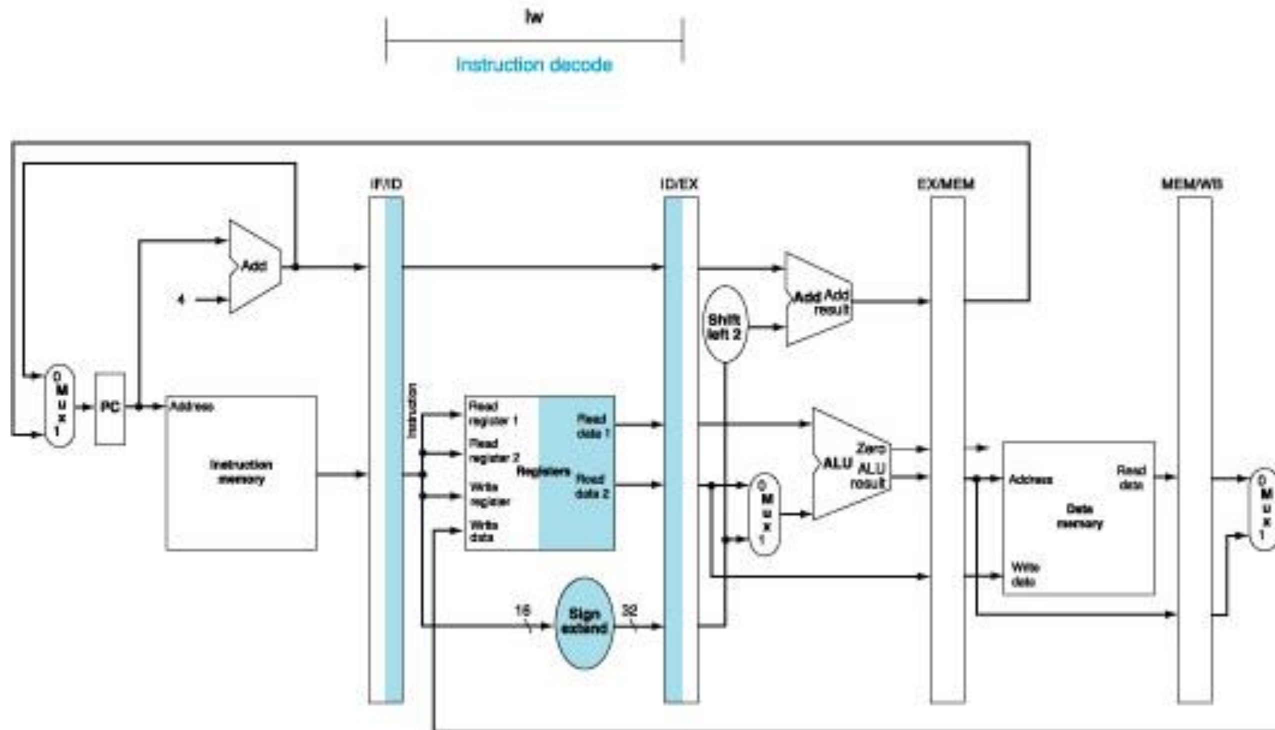
1. Instruction fetch:



- ✓ **Instruction being read and placed in the IF/ID register**
- ✓ **PC is incremented by 4 and written back into the PC. This incremented is also saved in the IF/ID.**

Continue

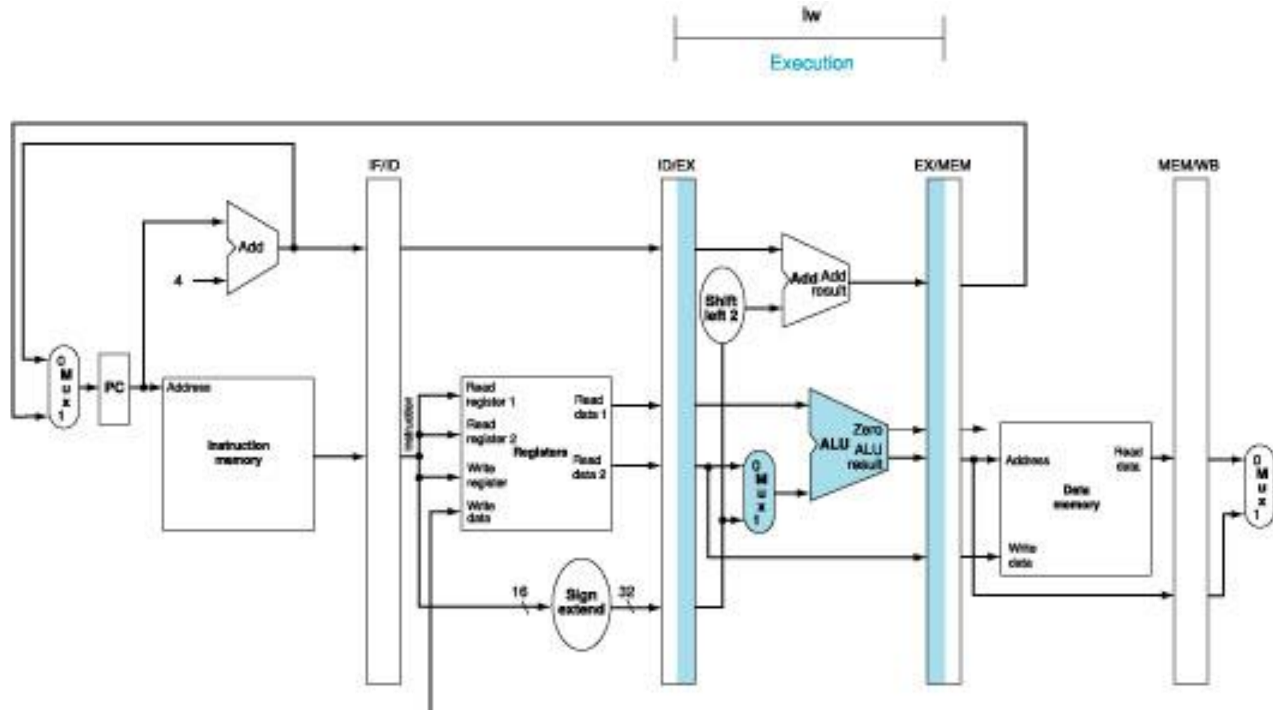
2. Instruction decode and register file read:



- ✓ IF/ID register supplying the 16-bit immediate field, and register numbers to read the two registers.
- ✓ All three values are stored in the ID/Ex register, along with the incremented PC.

Continue

3. Execute or address calculation:



✓ **Calculate the address and place it in the EX/MEM register.**

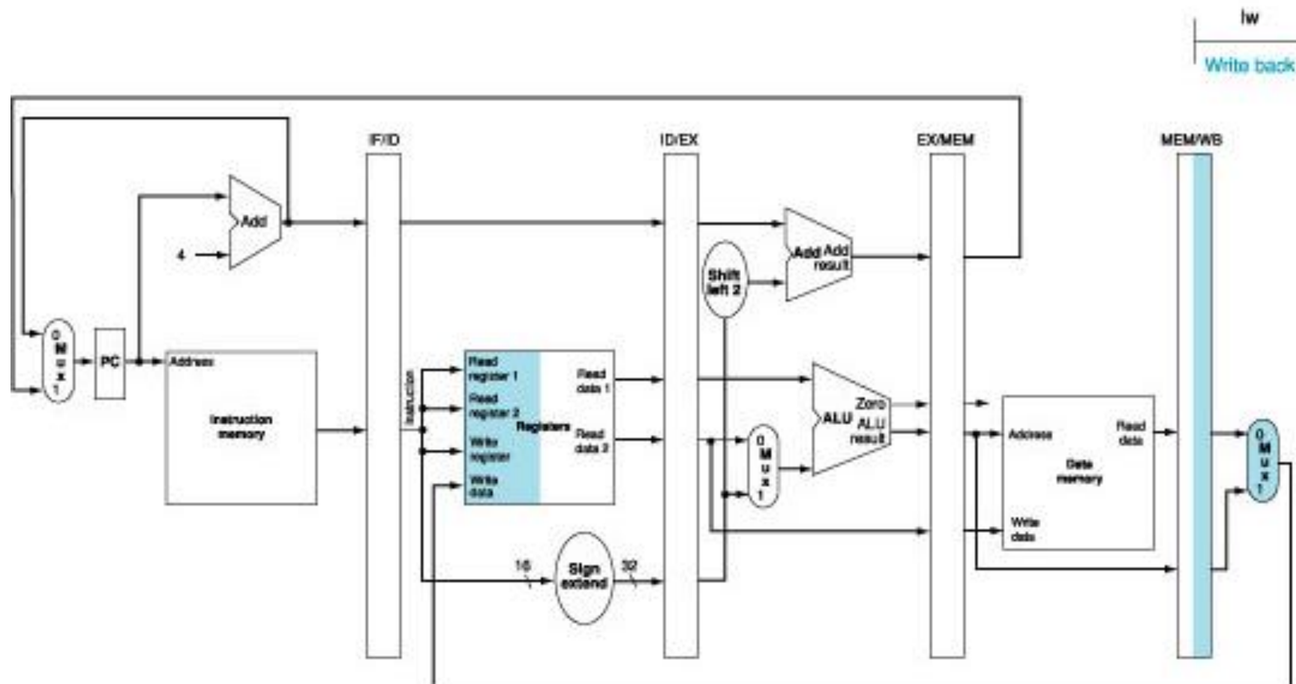
Continue

4. Memory access:

- ✓ **Read the data from the memory using the address from the EX/MEM register and load the data into the MEM/WB register.**

Continue

5. Write back:

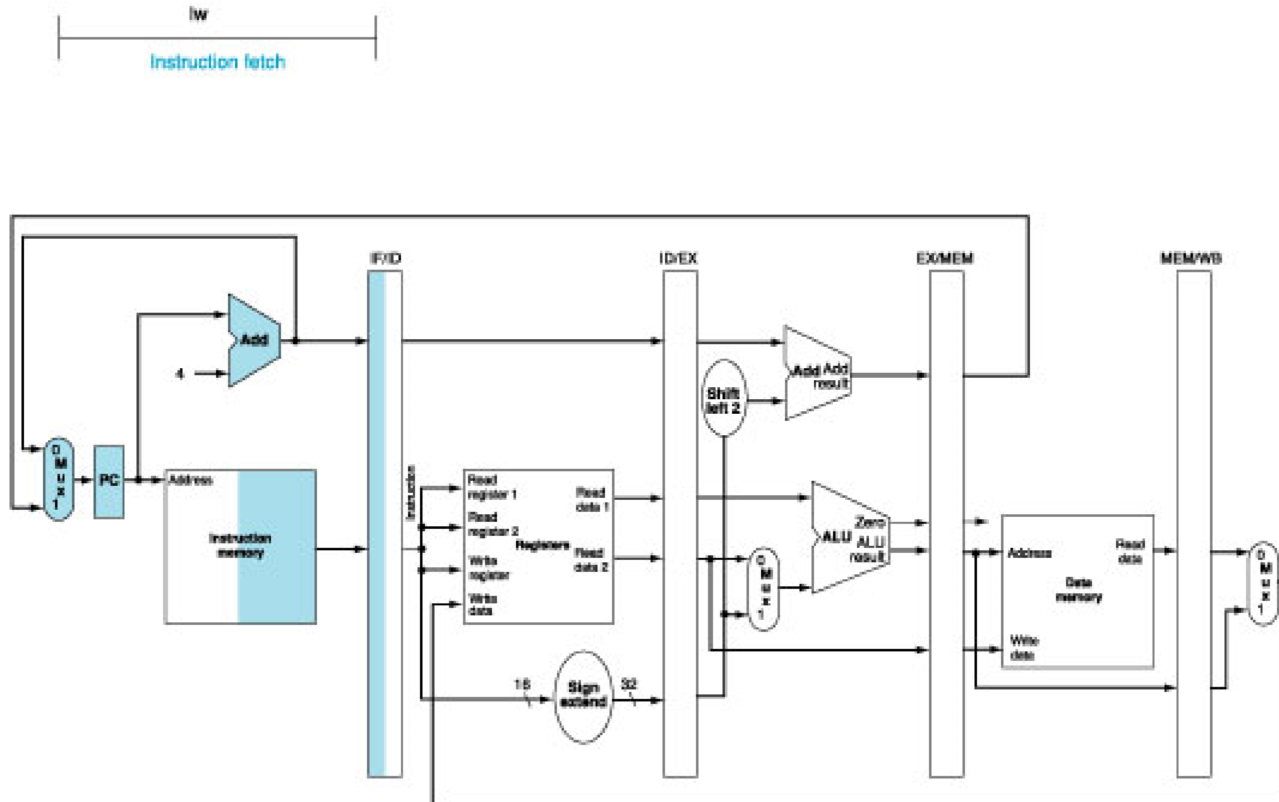


- ✓ **Reading the data from the MEM/WB register and writing it into the register file.**

Continue

- The five stages for **Store** Instruction are:

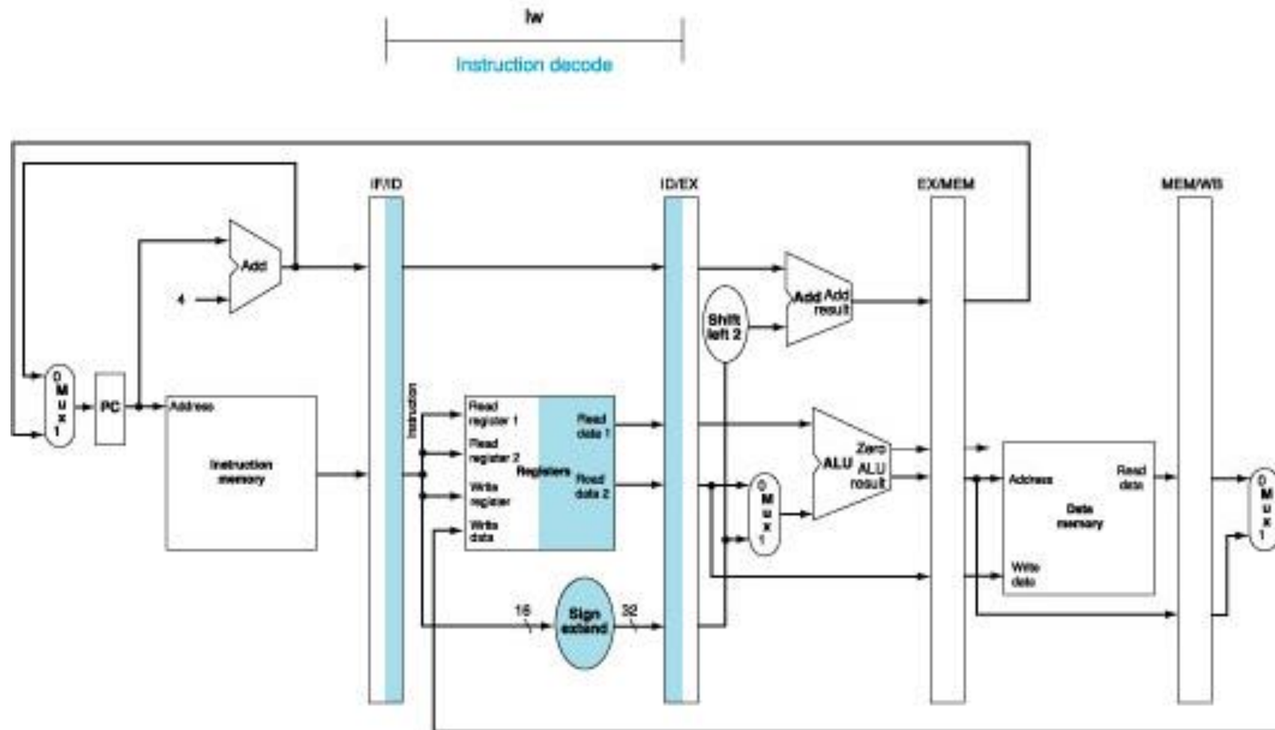
1. **Instruction fetch:**



- ✓ Instruction being read and placed in the IF/ID register
- ✓ PC is incremented by 4 and written back into the PC. This incremented is also saved in the IF/ID.

Continue

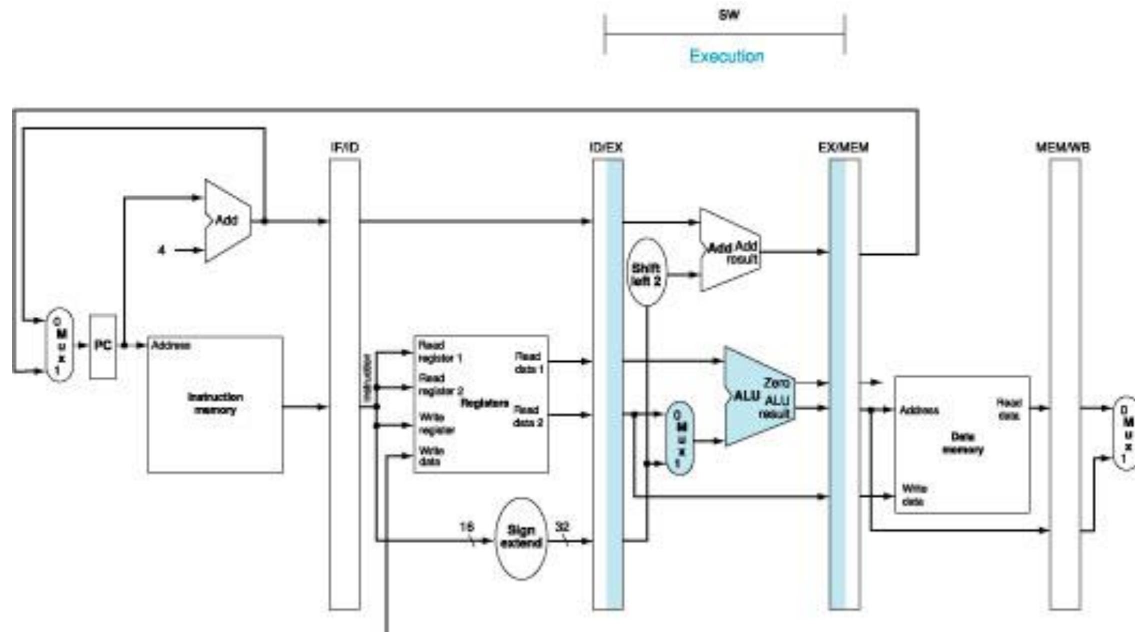
2. Instruction decode and register file read:



- ✓ IF/ID register supplying the 16-bit immediate field, and register numbers to read the two registers.
- ✓ All three values are stored in the ID/Ex register, along with the incremented PC.

Continue

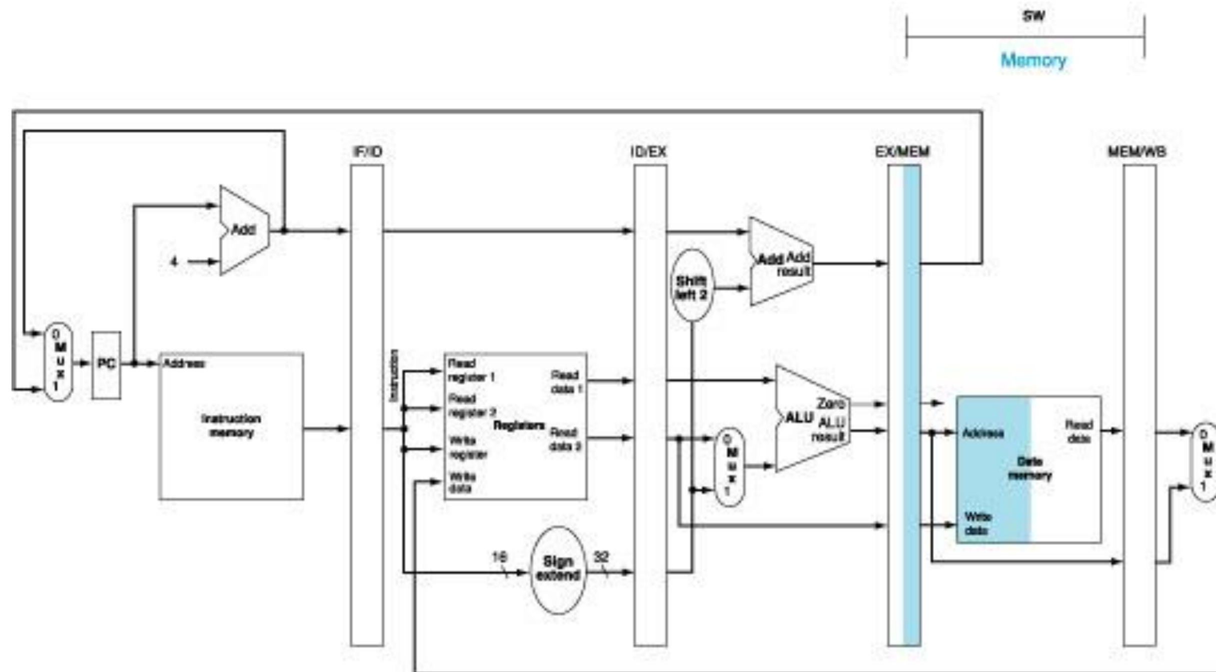
3. Execute or address calculation:



- ✓ **Calculate the address and place it in the EX/MEM register.**

Continue

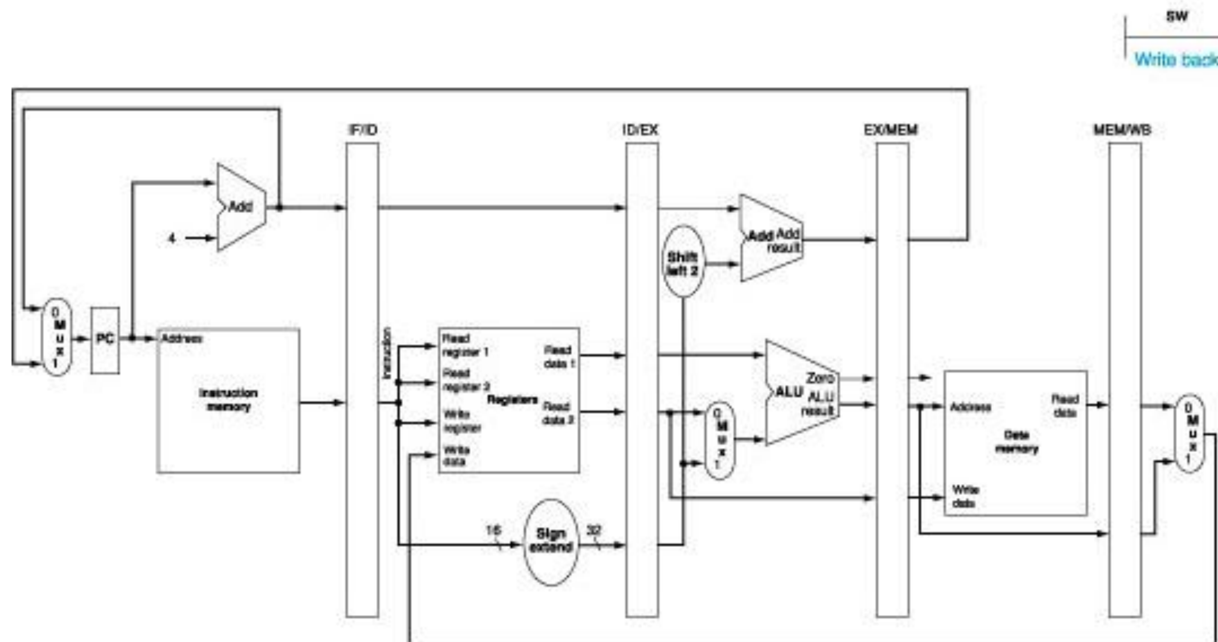
4. Memory access:



- ✓ **Write the data into the memory using the address from the EX/MEM register.**

Continue

5. Write back:



✓ **For this instruction, nothing happens in the write-back stage.**

Graphically Representing Pipelines

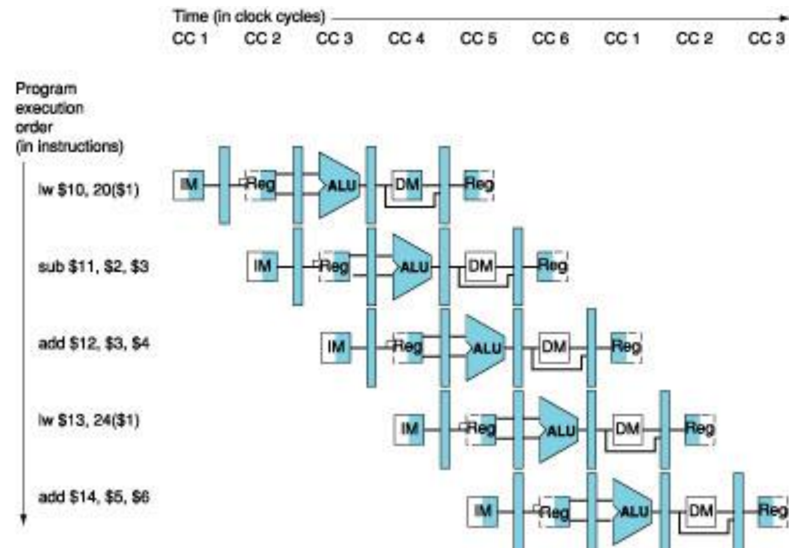
- **Two basic styles of pipeline figures:**
  1. **Multiple-clock-cycle pipeline diagrams**
  2. **Single-clock-cycle pipeline diagrams**

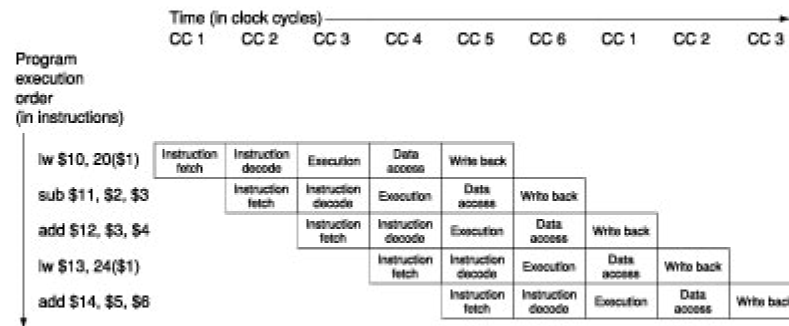
- **For Example, consider the following five-instructions sequence:**

```
lw  $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw  $13, 24($1)
add $14, $5, $6
```

Graphically Representing Pipelines

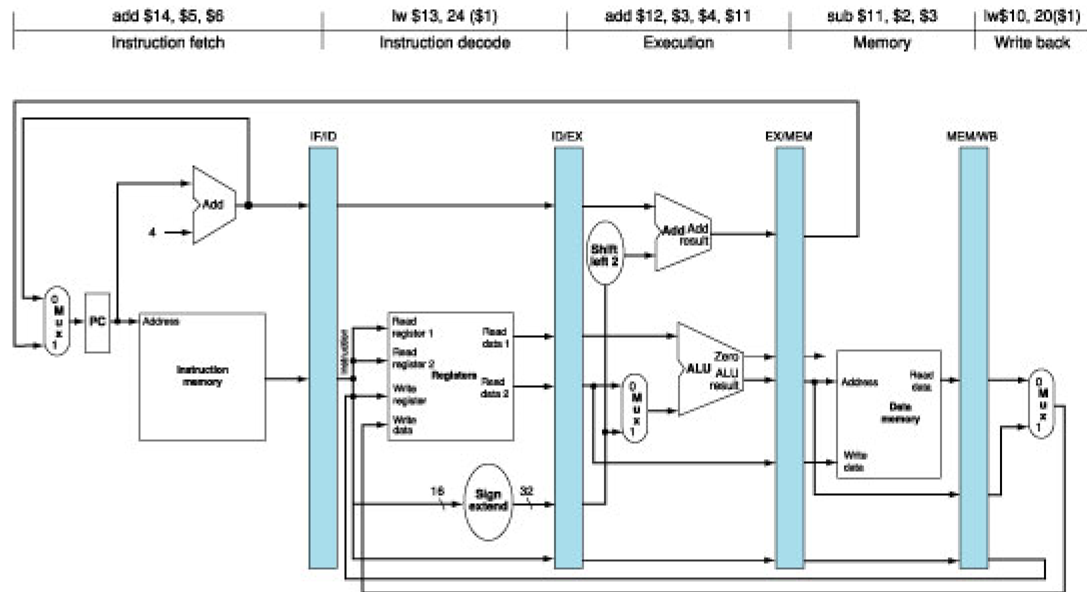
1. Multiple-clock-cycle pipeline diagrams



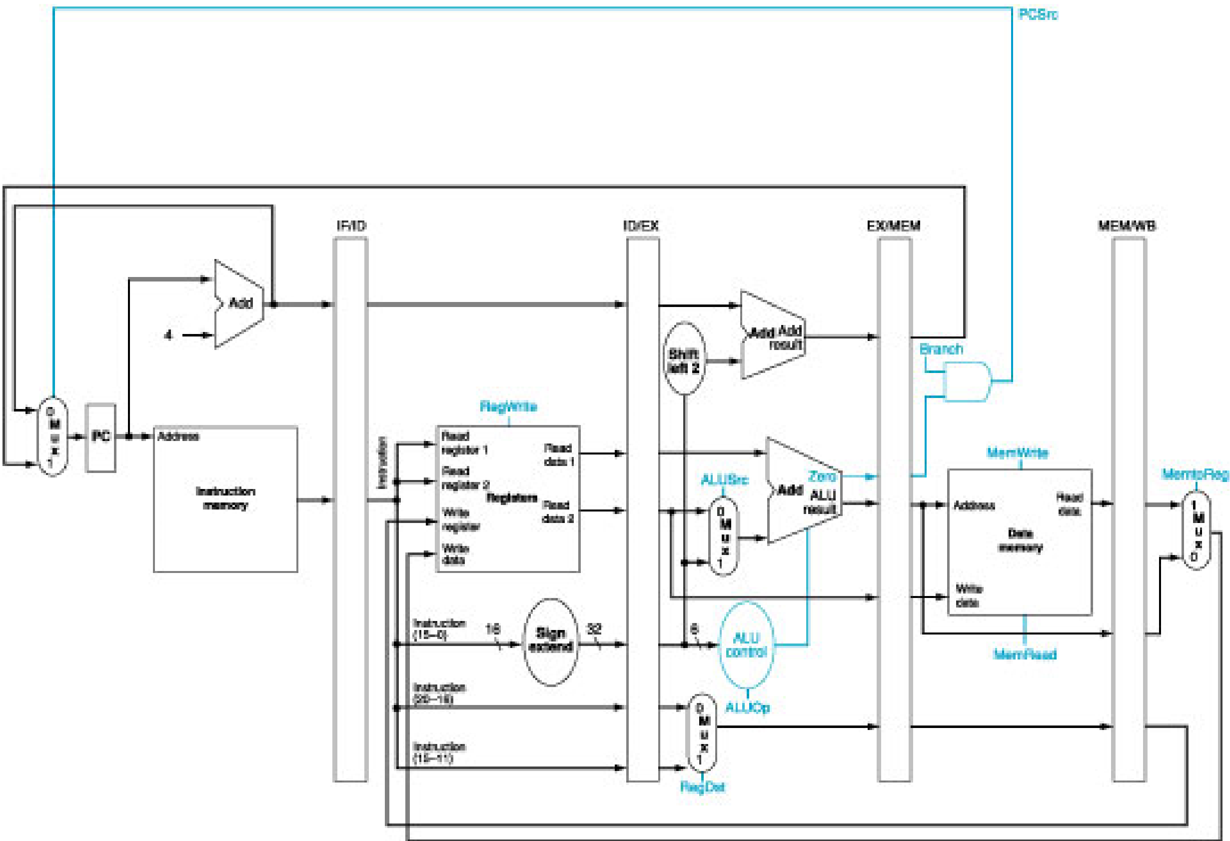


Graphically Representing Pipelines

2. Single-clock-cycle pipeline diagrams



6.3 Pipelined Control



Pipelined Control

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

FIGURE 6.25 The values of the control lines are the same as in Figure 5.18 on page 308, but they have been shuffled into three groups corresponding to the last three pipeline stages.

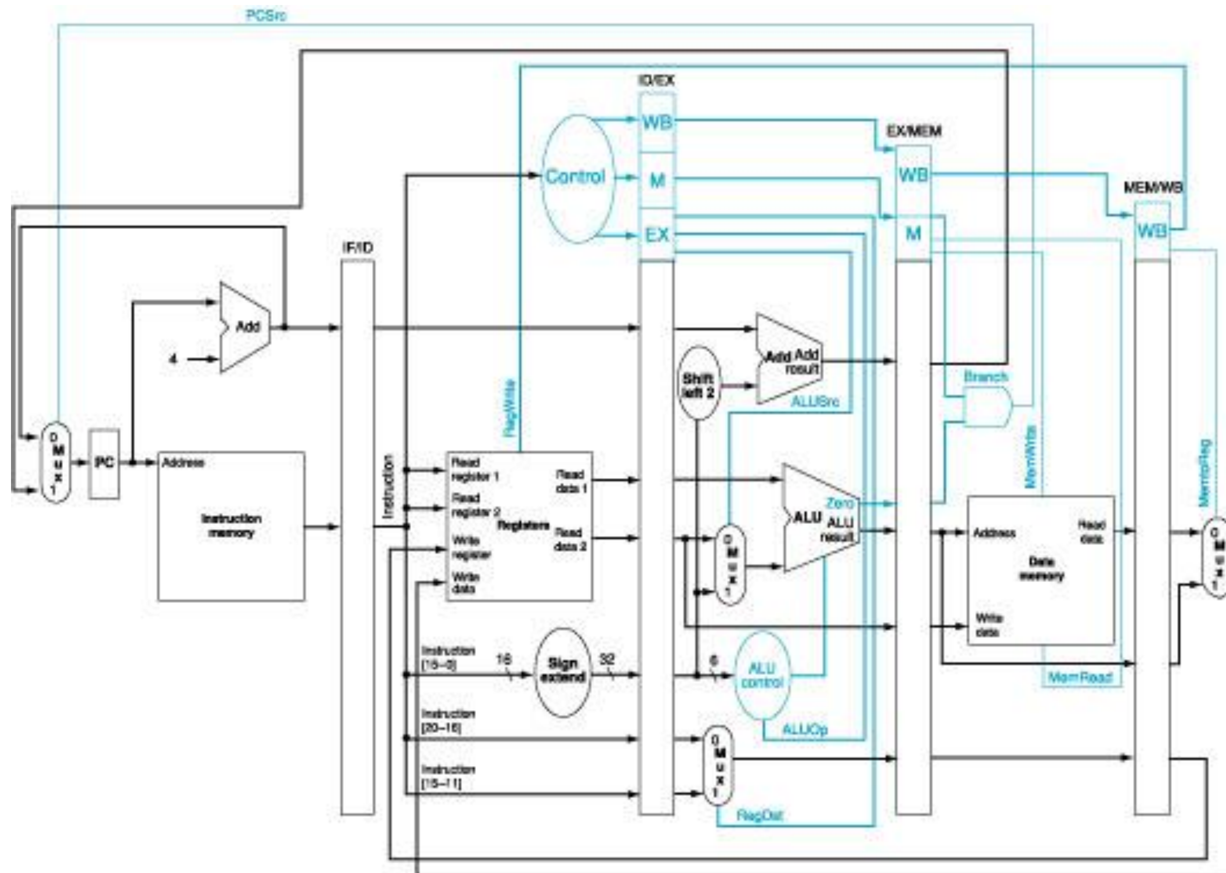
Pipelined Control

Control lines into five groups according to pipelines stage:

1. **Instruction fetch: Nothing special to set.**
2. **Instruction decode/register file read: Nothing special to set.**
3. **Execution/address calculation: signals to be set are RegDst. ALUOp, and ALUSrc.**
4. **Memory access: Branch, MemRead, and MemWrite.**
5. **Write back: MemtoReg and RegWrite.**

Instruction
Control
WB
M
EX
WB
M
WB
IF/ID
ID/EX
EX/MEM
MEM/WB

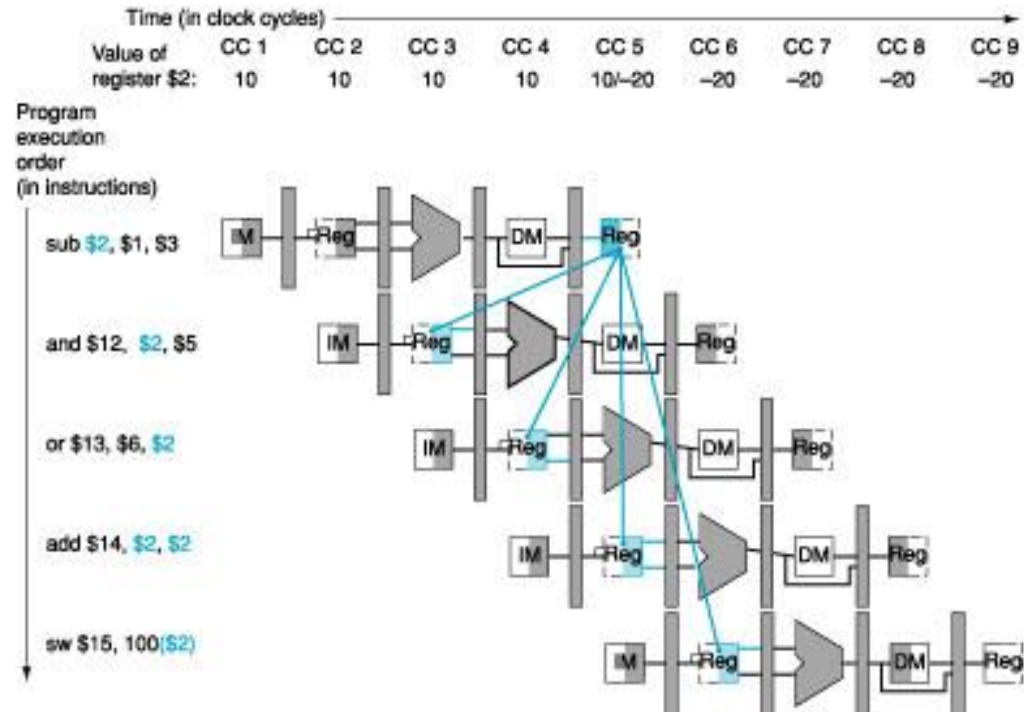
Pipelined Control



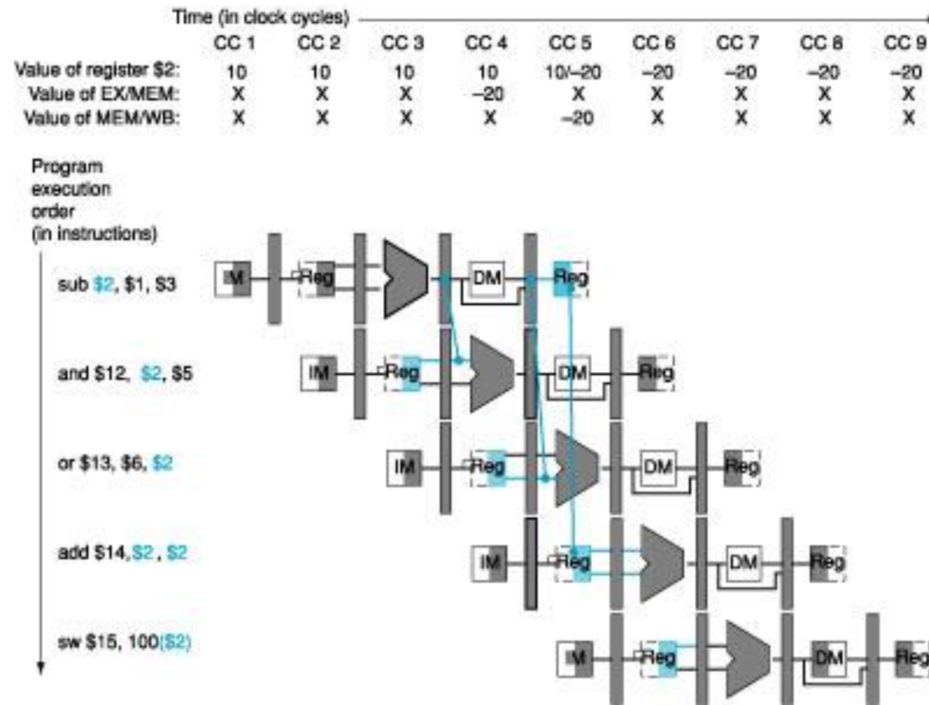
6.4 Data Hazard and Forwarding

Let's look at a sequence with many dependences:

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



Data Hazard and Forwarding



The two pairs of hazard conditions are:

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

Data Hazard and Forwarding

Example: Dependence Detection

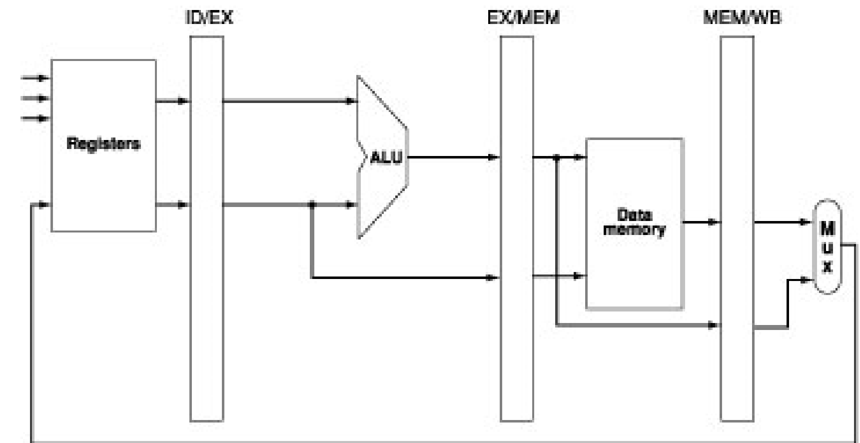
Classify the dependences in this sequence:

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

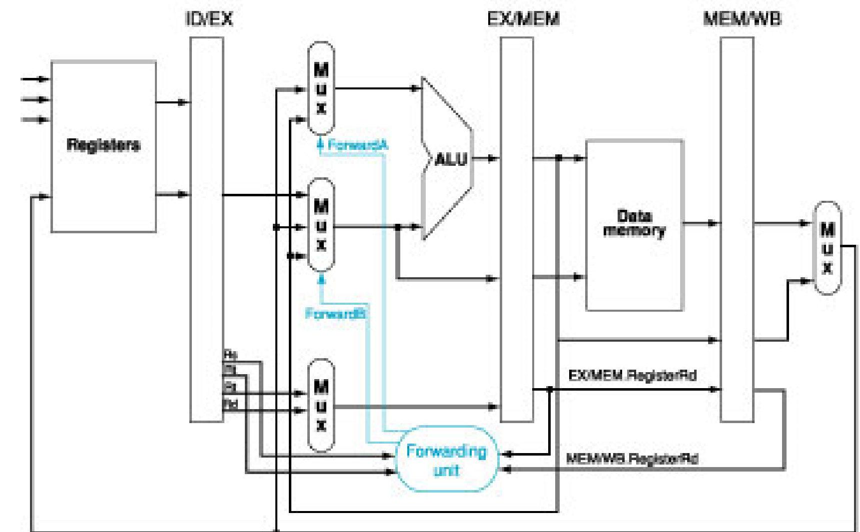
- **The `sub-and` is a type 1a hazard:**
  EX/MEM.RegisterRd = ID/EX.RegisterRs = `$2`
- **The `sub-or` is atype 2b hazard:**
  MEM/WB.RegisterRd = ID/EX.RegisterRt= `$2`
- **The two dependences on `sub-add` are not hazards because the register file supplies the proper data during ID stage of `add`.**
- **There is no data hazard between `sub` and `sw` because `sw` reads `$2` the clock after `sub` write `$2`.**

Data Hazard and Forwarding

ALU and pipeline register before and after adding forwarding



a. No forwarding



b. With forwarding

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.

FIGURE 6.31 The control values for the forwarding multiplexors in Figure 6.30. The signed immediate that is another input to the ALU is described in the elaboration at the end of this section.

Data Hazard and Forwarding

- **Some instructions do not write registers, thus add conditions:**

  `EX/MEM.RegWrite`

  `MEM/WB.RegWrite`

- **Also, if the pipeline has $0 as its destination,for example:**

  `sll $0, $1, 2`

  Thus, add conditions:

  `EX/MEM.RegisterRd ≠ 0`

  `MEM/WB.RegisterRd ≠ 0`

Data Hazard and Forwarding

Let's now write both the conditions for detecting hazards and the control signals to resolve them:

1. **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
```

2. **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
```

Data Hazard and Forwarding

Potential data hazards:

For example, when summing a vector of numbers in a single register, a sequence of instructions will all read and write to the same register:

```
add  $1, $1, $2
add  $1, $1, $3
add  $1, $1, $4
...
```

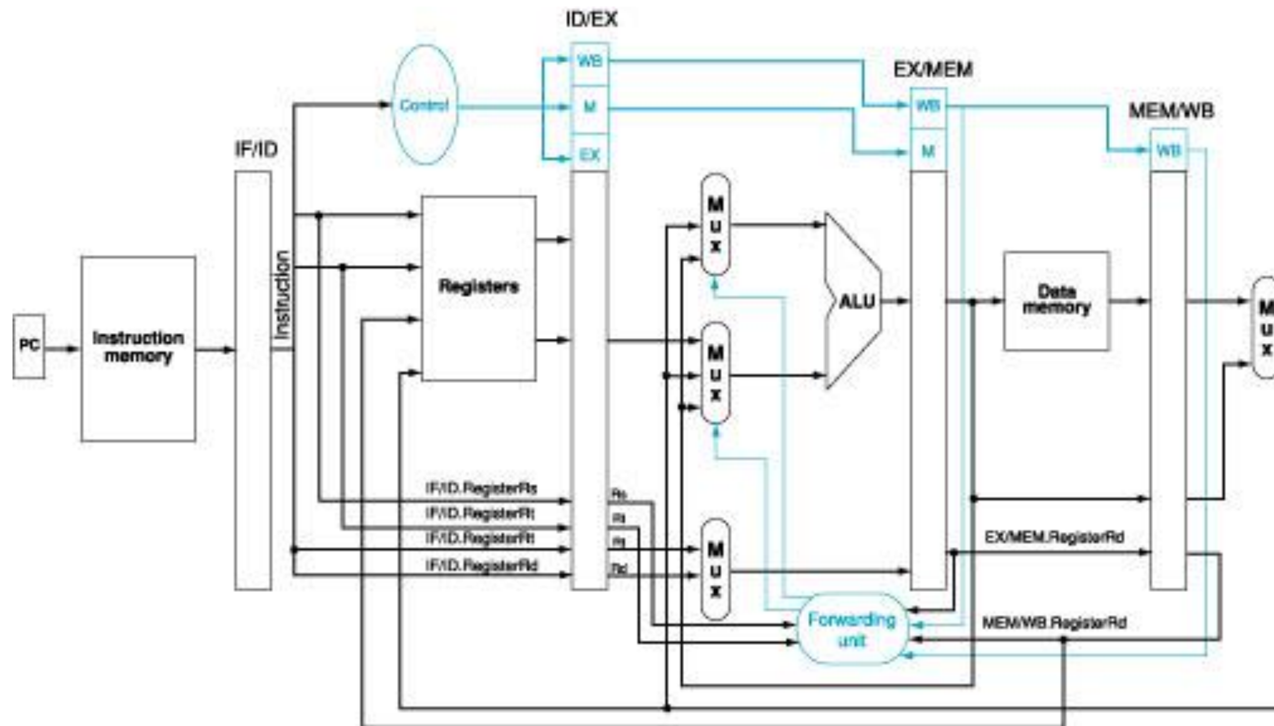
In this case, the result is forwarded from the MEM stage. Thus the control for MEM hazard would be:

```
if (MEM/WB.RegWrite and (MEM/WB.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 (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Data Hazard and Forwarding

The datapath modified to resolve hazards via forwarding



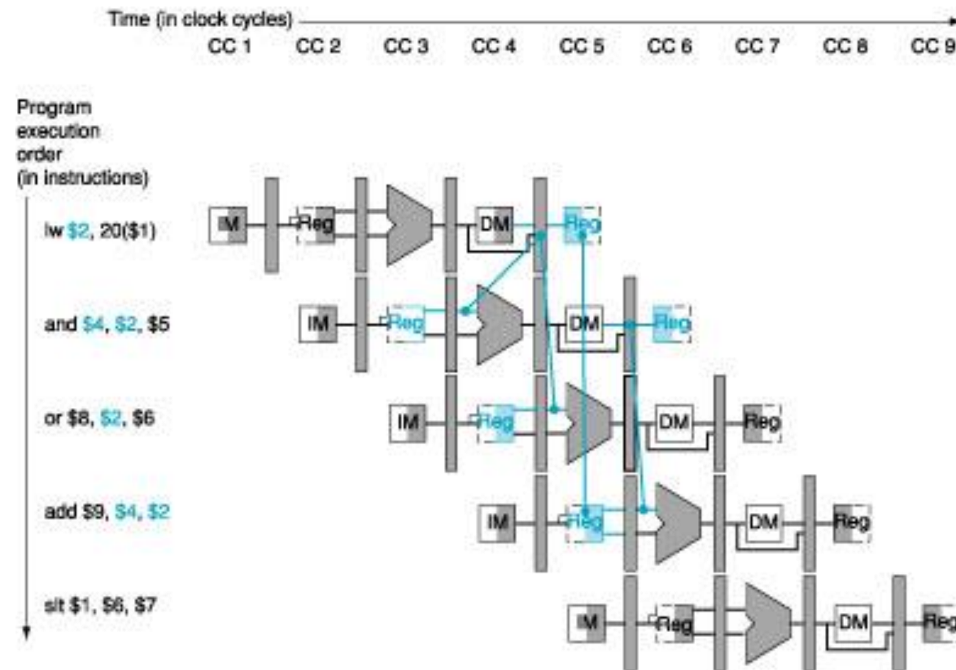
Data Hazard and Forwarding

Addition to select the signed immediate as an ALU input

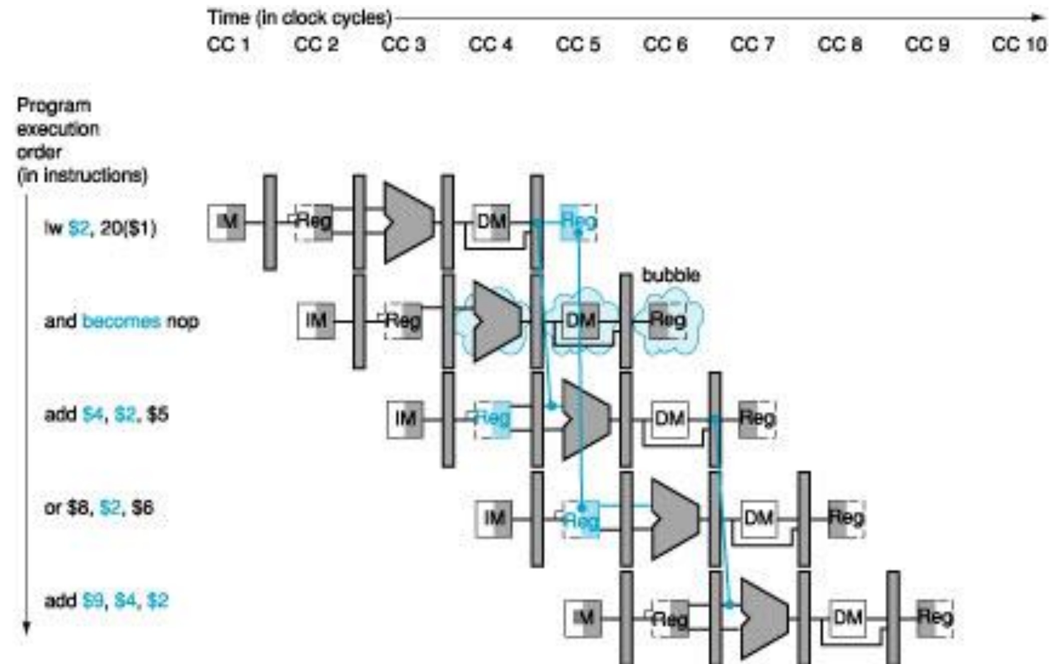
6.5 Data Hazard and Stalls

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

Data Hazard and Stalls



Data Hazard and Stalls



Data Hazard and Stalls