CSE 4305
Computer Organization and Architecture

Lecture 5
Pipelining

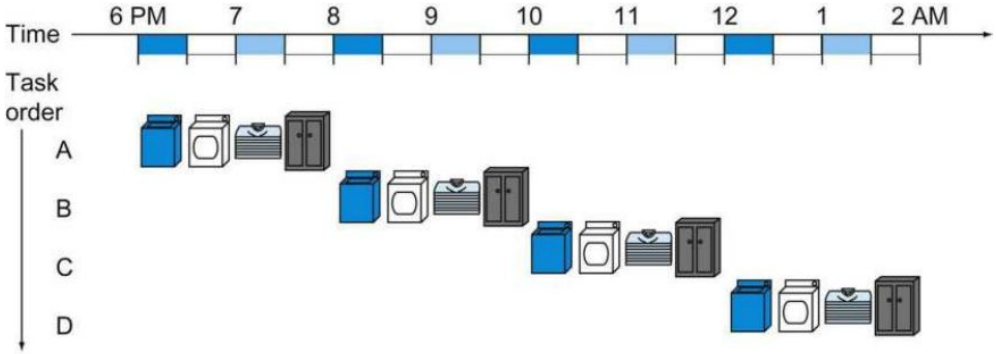
Sabrina Islam
Lecturer, CSE, IUT
Contact: +8801832239897
E-mail: sabrinaislam22@iut-dhaka.edu

Pipelining

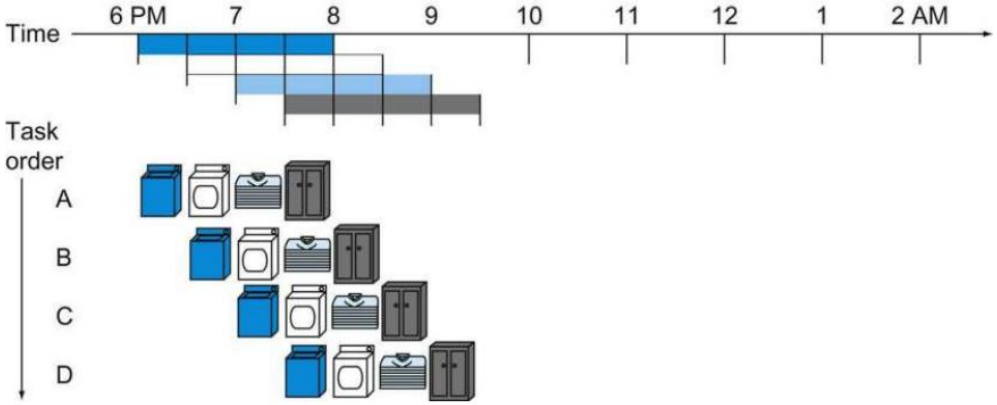
- Multiple instructions are overlapped in execution.
- Pipelining makes a system lot faster than the sequential execution of instructions.

The laundry analogy to understand Pipelining

Sequential implementation



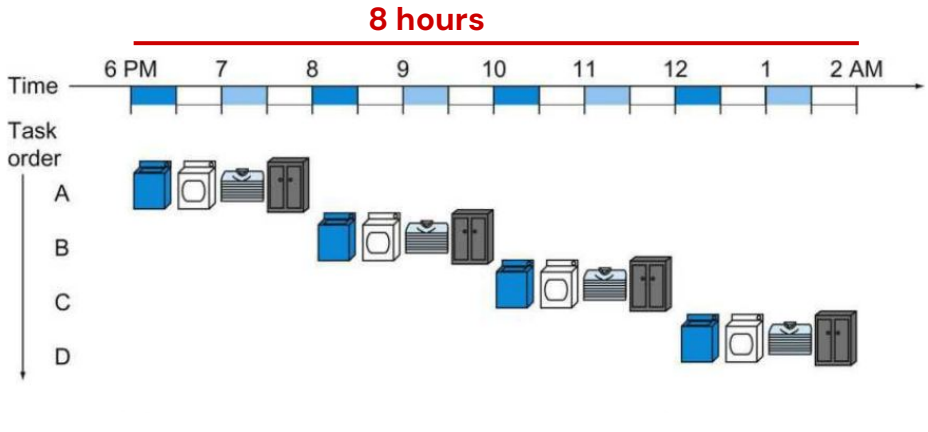
Pipelined implementation

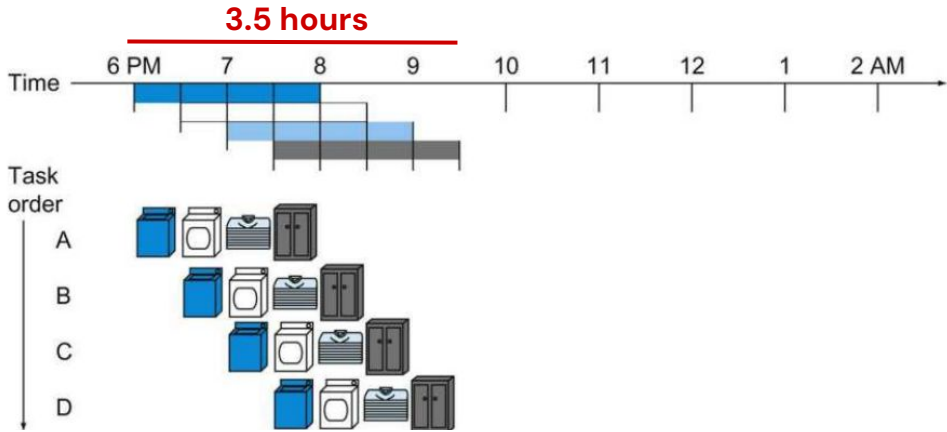


Stages:

1. Place clothes in the washer.
2. Place the wet load in the dryer.
3. Fold the dry load.
4. Place folded clothes in cupboard.

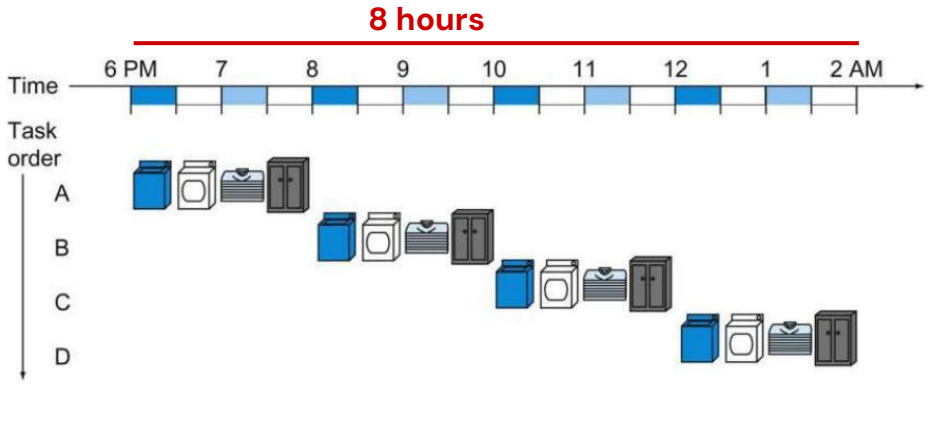
The laundry analogy to understand Pipelining

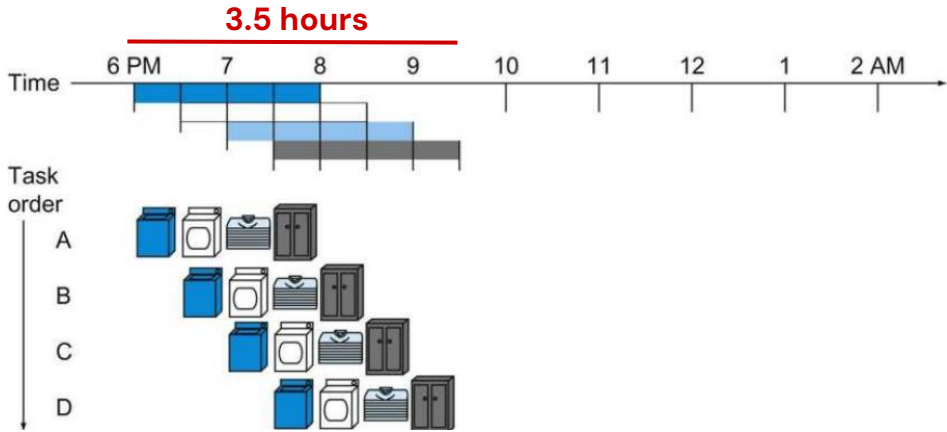




- Pipelining **does not decrease** the time to complete one task or instruction. Each of the tasks (A, B, C, D) take **2 hours** in both implementation.
- Pipelining **increases** throughput by decreasing the total time to complete all tasks in hand. The total time decreases from **8 hours to 3.5 hours**.

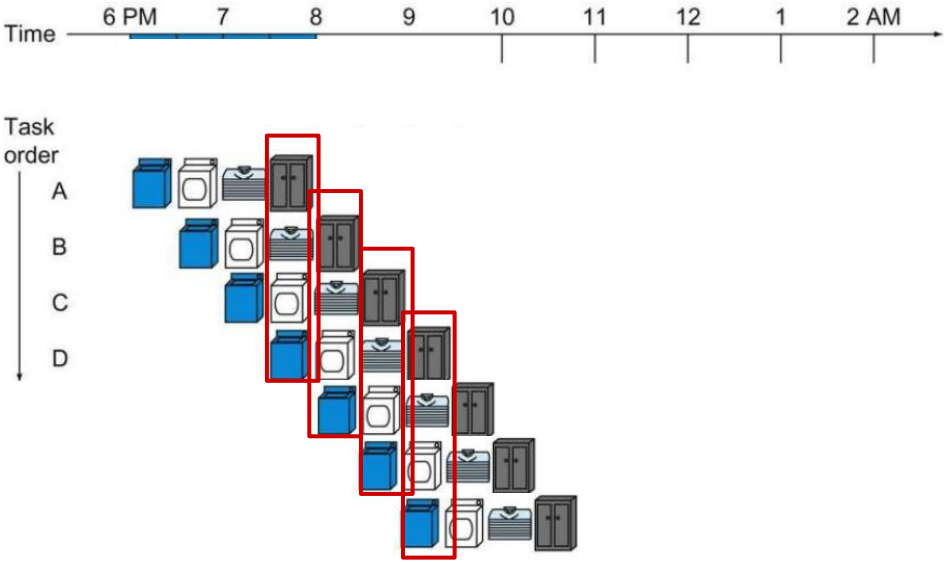
The laundry analogy to understand Pipelining





Here, speed-up = 8/3.5 or 2.3

Pipelining



If the stage times are equal and the number of tasks at hand is huge,

Speed-up = Number of stages

The stages of RISC-V instructions

1. Fetch instruction from memory.
2. Read registers and decode the instruction.
3. Execute the operation or calculate an address.
4. Access an operand in data memory (if necessary).
5. Write the result into a register (if necessary).

Example

Let's consider a RISC-V implementation and assume the execution time for each instructions!

1. Fetch instruction from memory - 200ps
2. Data access from memory - 200ps
3. ALU operation - 200ps
4. Register read - 100ps
5. Register write - 100ps

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load doubleword (ld)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store doubleword (sd)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Example

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load doubleword (ld)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store doubleword (sd)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

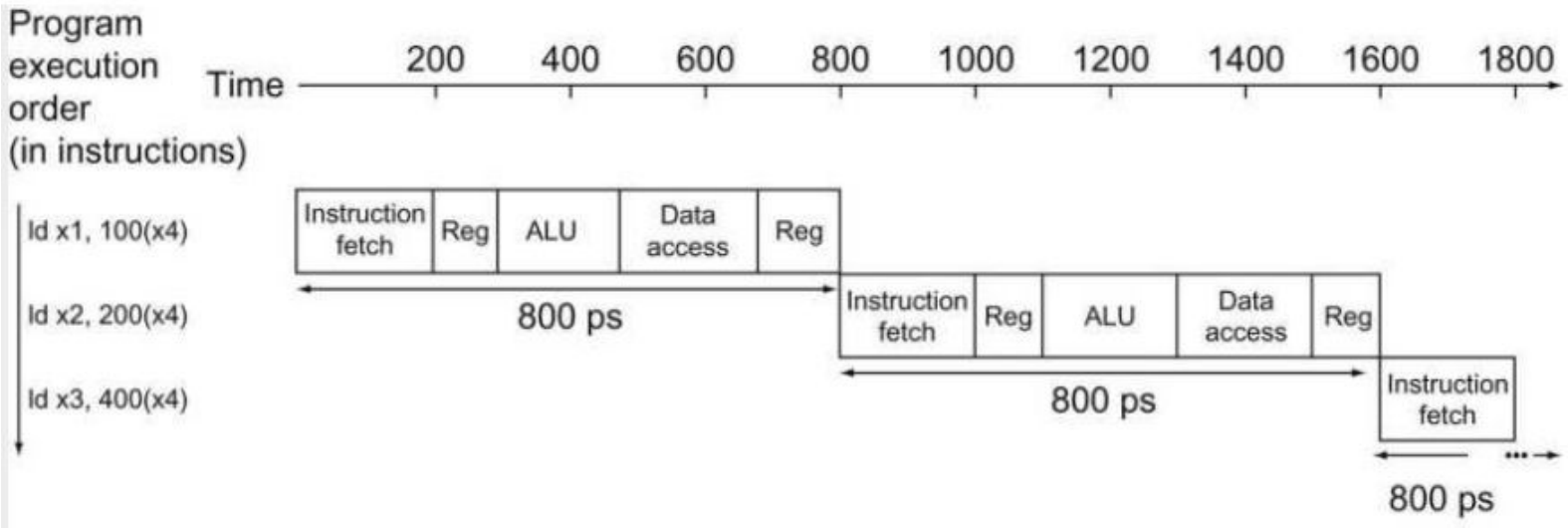
Longest time

In a single cycle implementation,

- The clock needs to be the longest time taken to execute an instruction. So the **clock time** for every instruction will be **800 ps**.
- The pipeline **stage time** is also limited by the slowest resource. Here it is **200ps**.

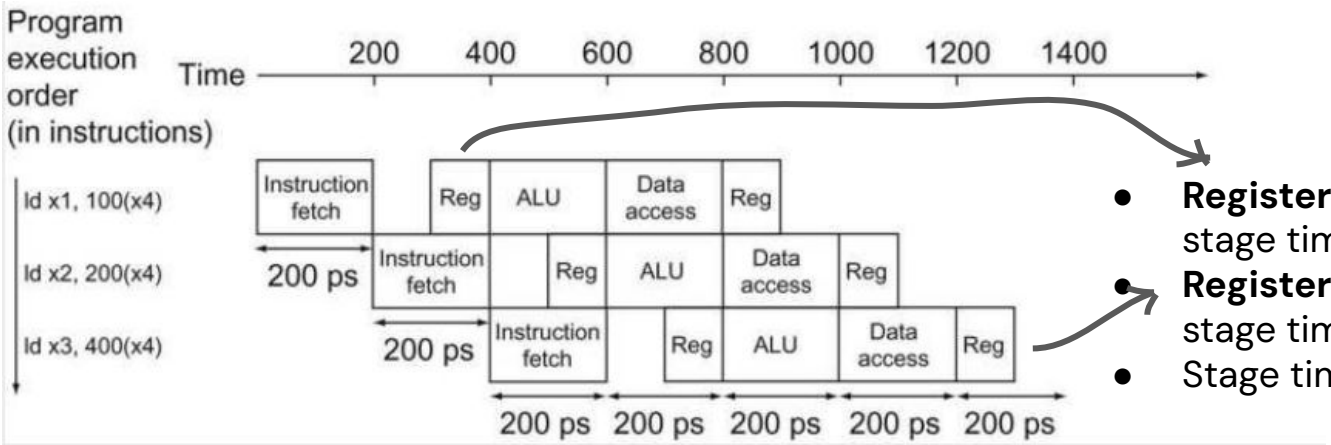
Example

Sequential execution of 3 load instructions



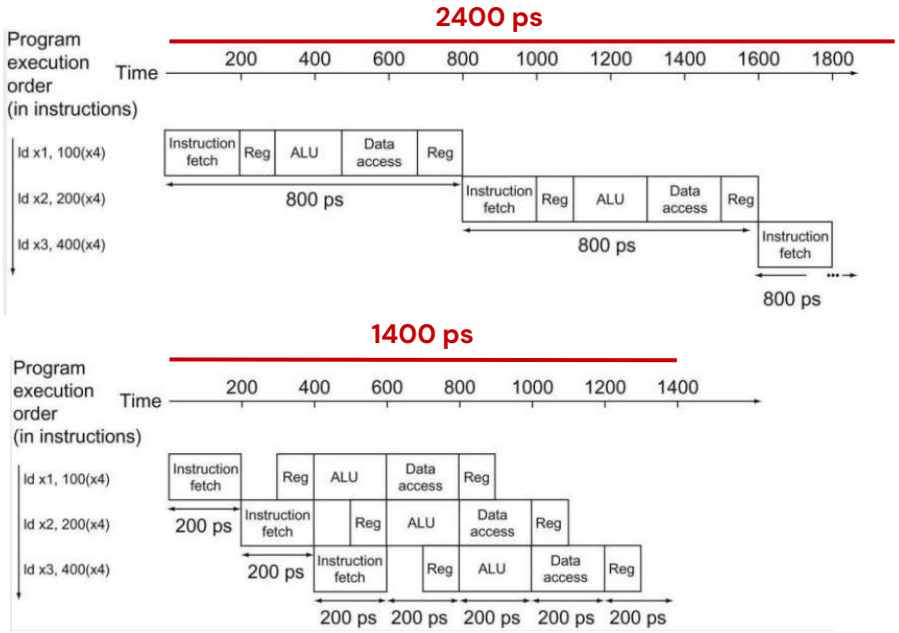
Example

Pipelined execution of 3 load instructions



- **Register read**: 2nd half of the stage time.
- **Register write**: 1st half of the stage time.
- Stage time here is 200ps.

Example



As shown earlier,

Speed-up = Number of stages

Performance Improvement:

In this example, speed-up = 2400/1400
= 1.71

Let's consider 1,000,003 instructions

In the non-pipelined example,

time = 1,000,000 ×800 ps +2400 ps
= 800,002,400 ps

In the pipelined example,

time = 1,000,000 ×200 ps +1400 ps
= 200,001,400 ps

Improvement = 800,002,400/ 200,001,400
= **4**

Designing instructions for Pipelining

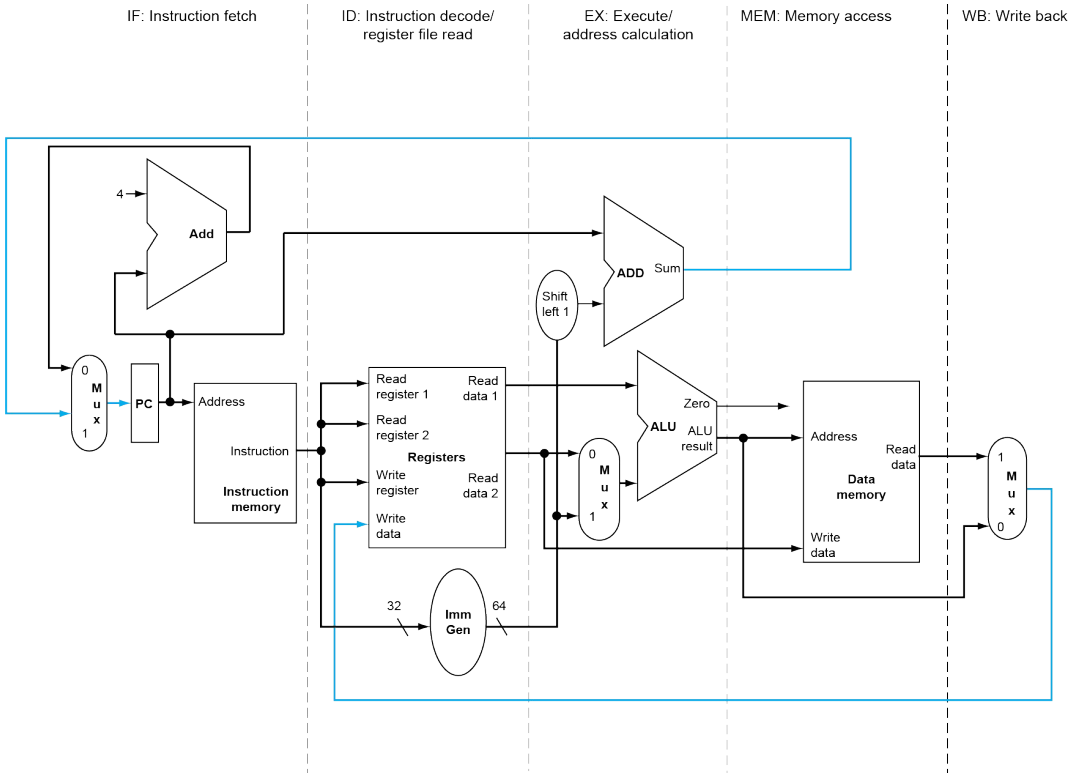
Characteristics of the RISC-V design that are helpful in pipelining:

- All RISC-V instructions are the same length.
- Only a few instruction formats, with the source and destination register fields being located in the same place in each instruction.
- Memory operands only appear in loads or stores in RISCV.

A brief of pipeline hazards

- **Structural hazard**
  - Hardware cannot support the combination of instructions that we want to execute in the same clock cycle.
  - Ex: Having a single memory instead of two. If there were 4 load instructions, we would see that in the same clock cycle, the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory.
- **Data Hazard**
  - Occurs when the pipeline must be stalled because one step must wait for another to complete.
  - Data that are needed to execute the instruction are not yet available.
  - Can be solved by **forwarding or bypassing.**

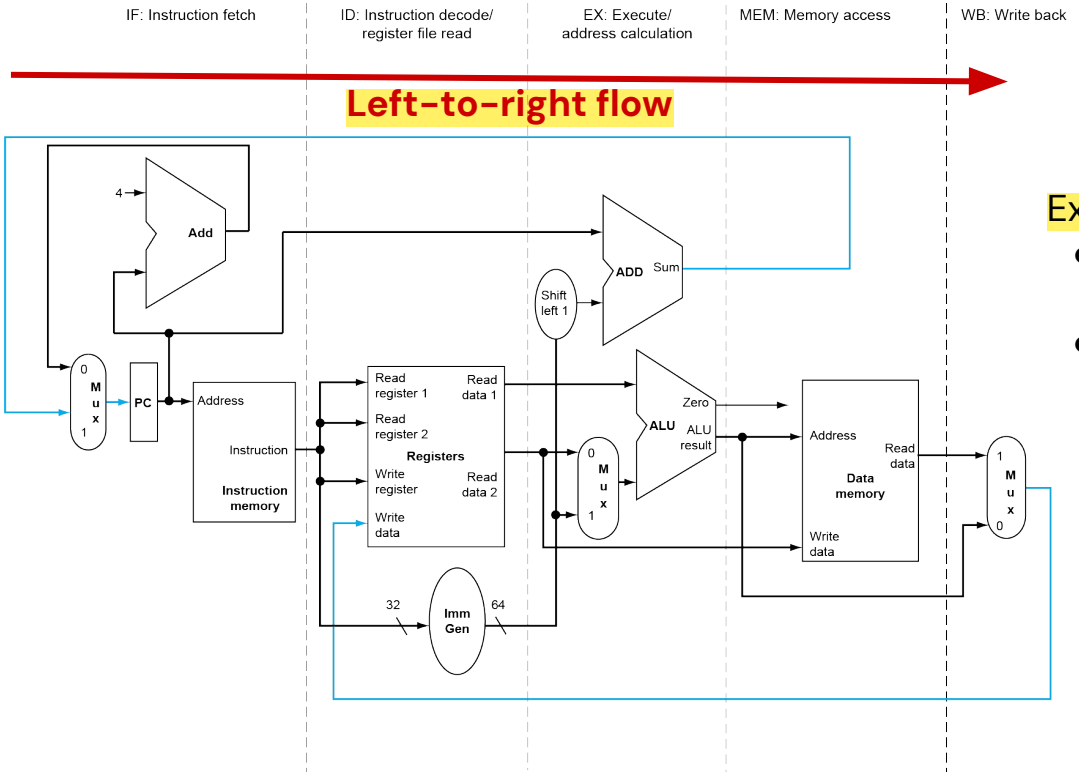
Datapath divided into 5 stages



- **IF**: Instruction fetch
- **ID**: Instruction decode and register file read
- **EX:** Execution or address calculation
- **MEM:** Data memory access
- **WB:** Write back

5-stage pipeline

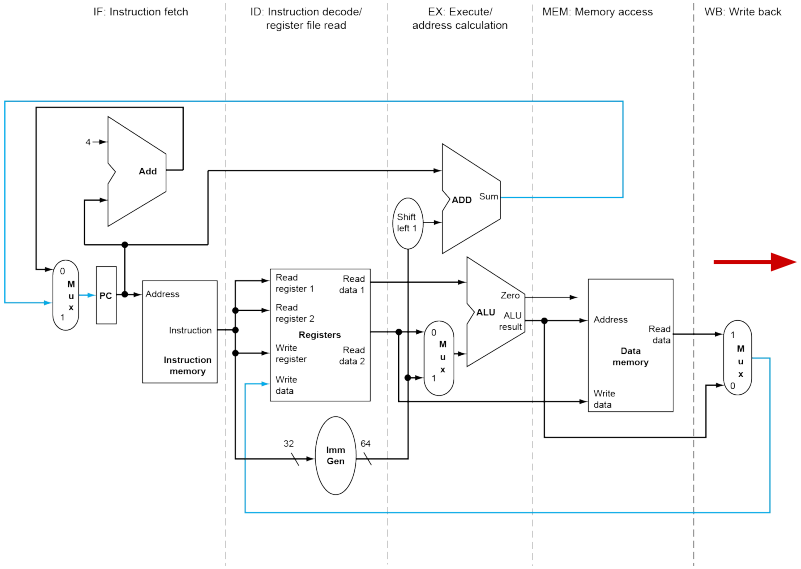
Datapath divided into 5 stages

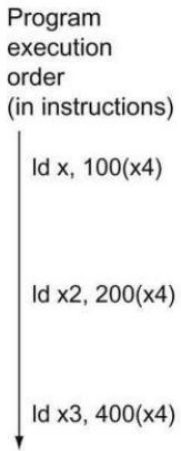


Exception to this left-to-right flow

- **Write-back:**
  - Result back to register
- **Instruction fetch:**
  - PC = PC + 4 or,
  - PC = PC + offset (from MEM)

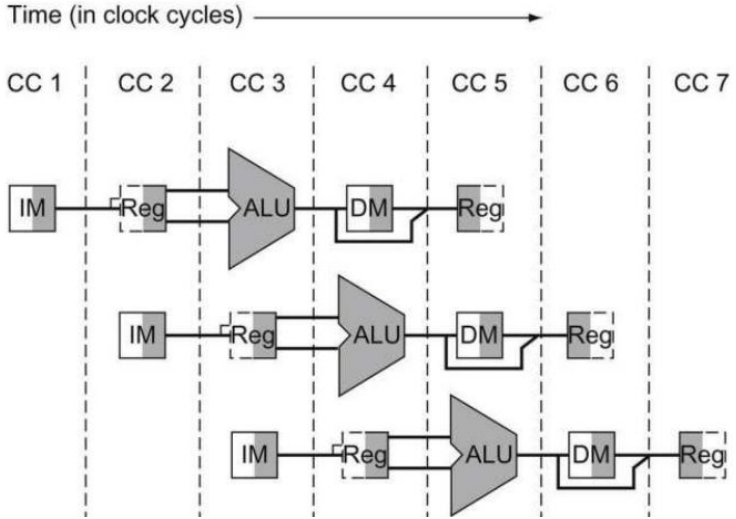
Pipelined Datapath





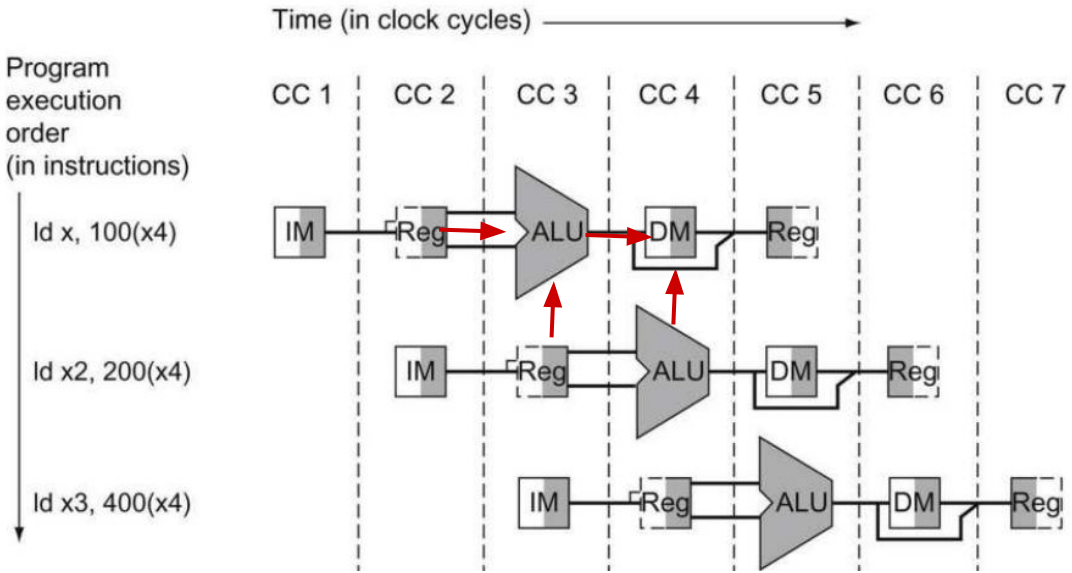
In case of the registers and memory,

- right half shaded denotes being read.
- left half shaded denotes being written



Assuming each instruction has its own datapath. But one datapath is shared in reality.

Pipelined Datapath

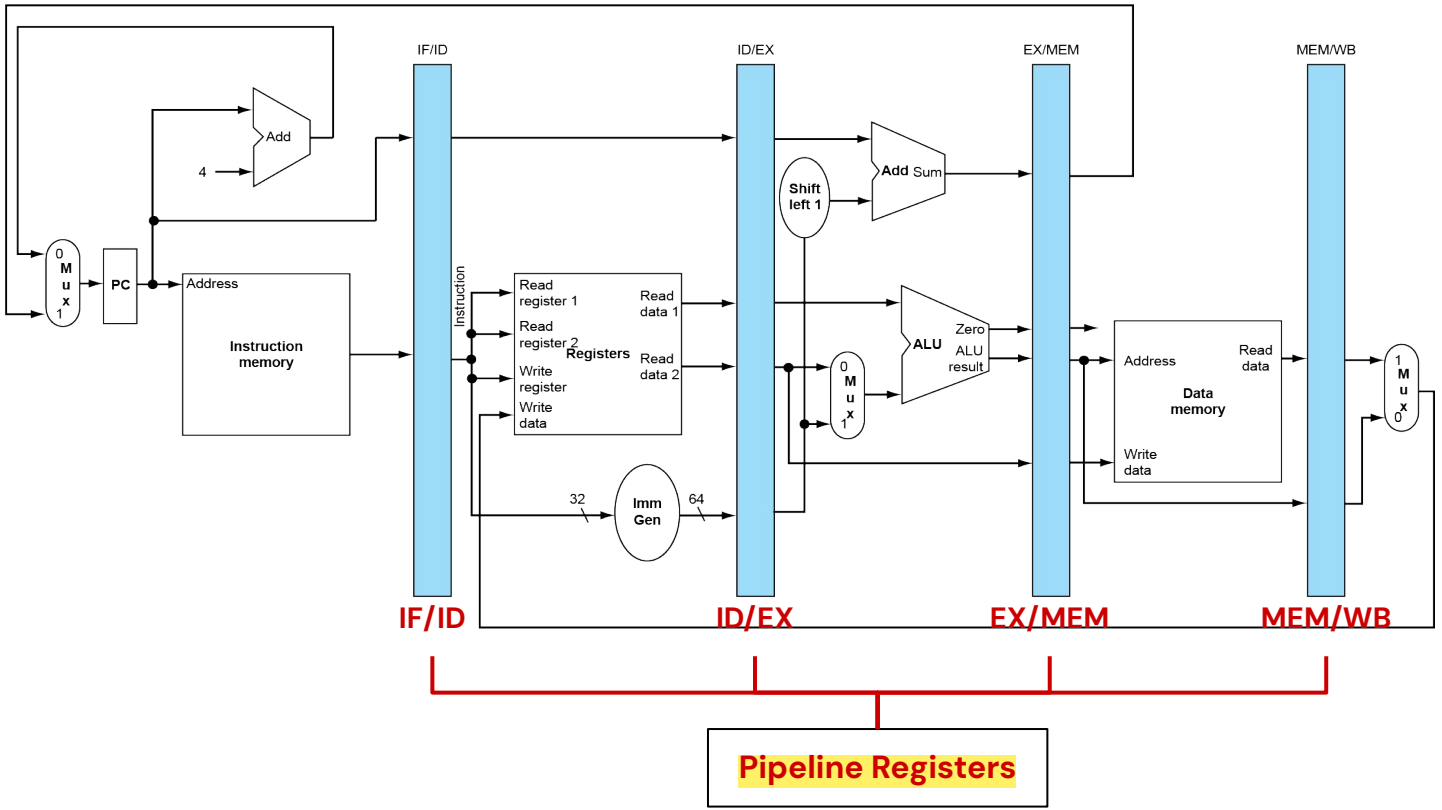


While sharing the datapath elements, **overlap may occur!**

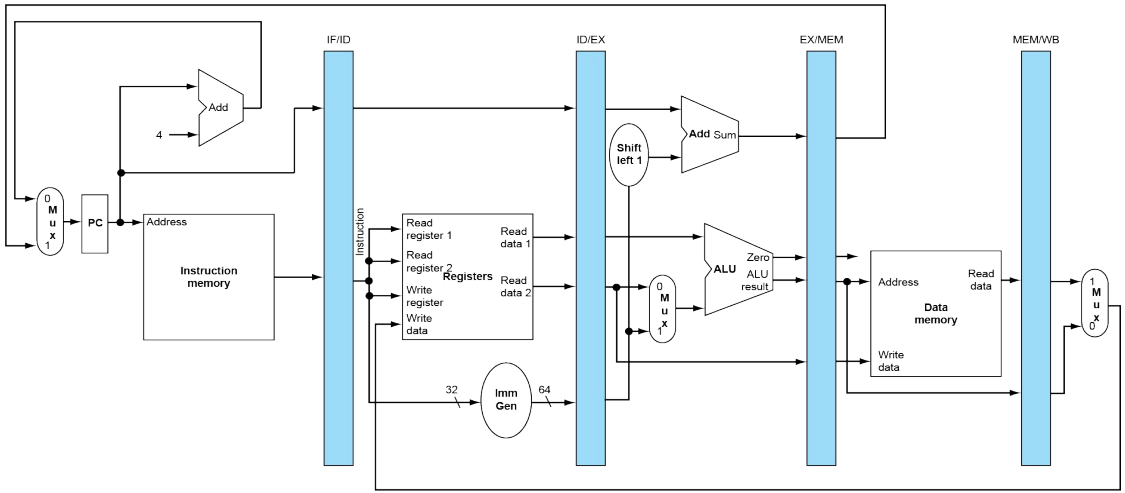
For example, in CC3 ALU may get reg value from the 2nd load instruction instead of the 1st instruction. Same thing can happen in other stages too.

Add registers between stages to avoid this conflict!

Pipelined Datapath



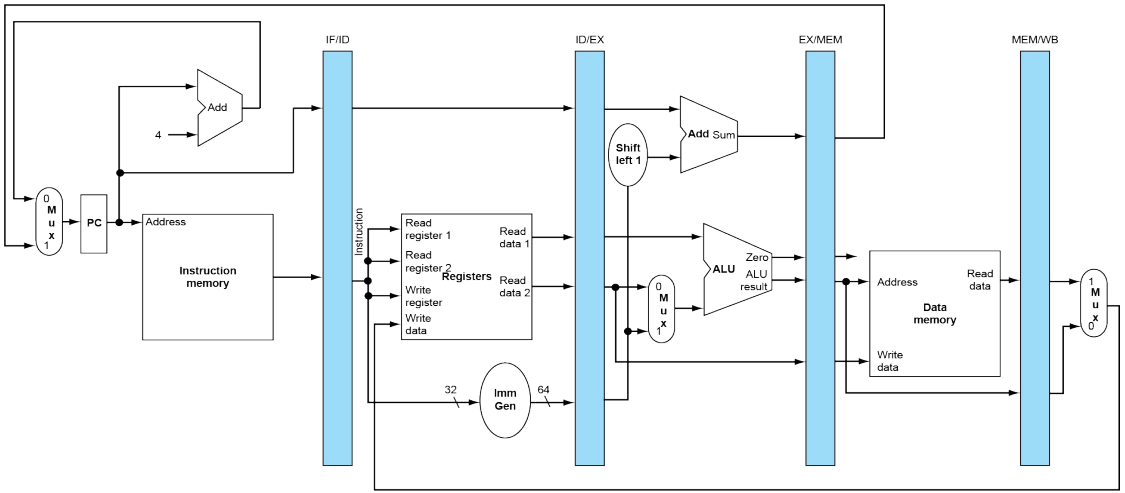
Pipelined Datapath



No pipeline after the write-back stage. Why?

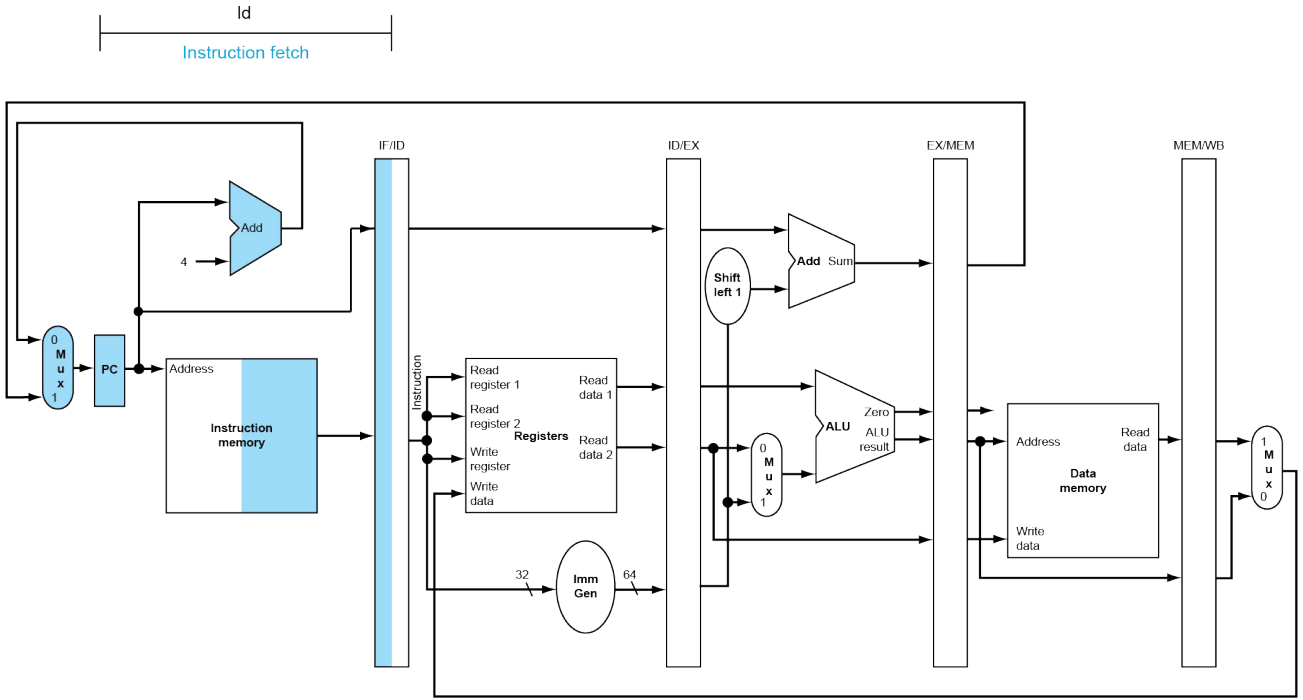
- The value from that stage is stored in a particular register already.

Pipelined Datapath



- To understand this pipelined execution, we'll next see the **load** instruction as it is **active in all stages** (IF, ID, EX, MEM, WB).
- The highlighted regions of the registers or memory
  - **right half**: being read
  - **left half**: being written

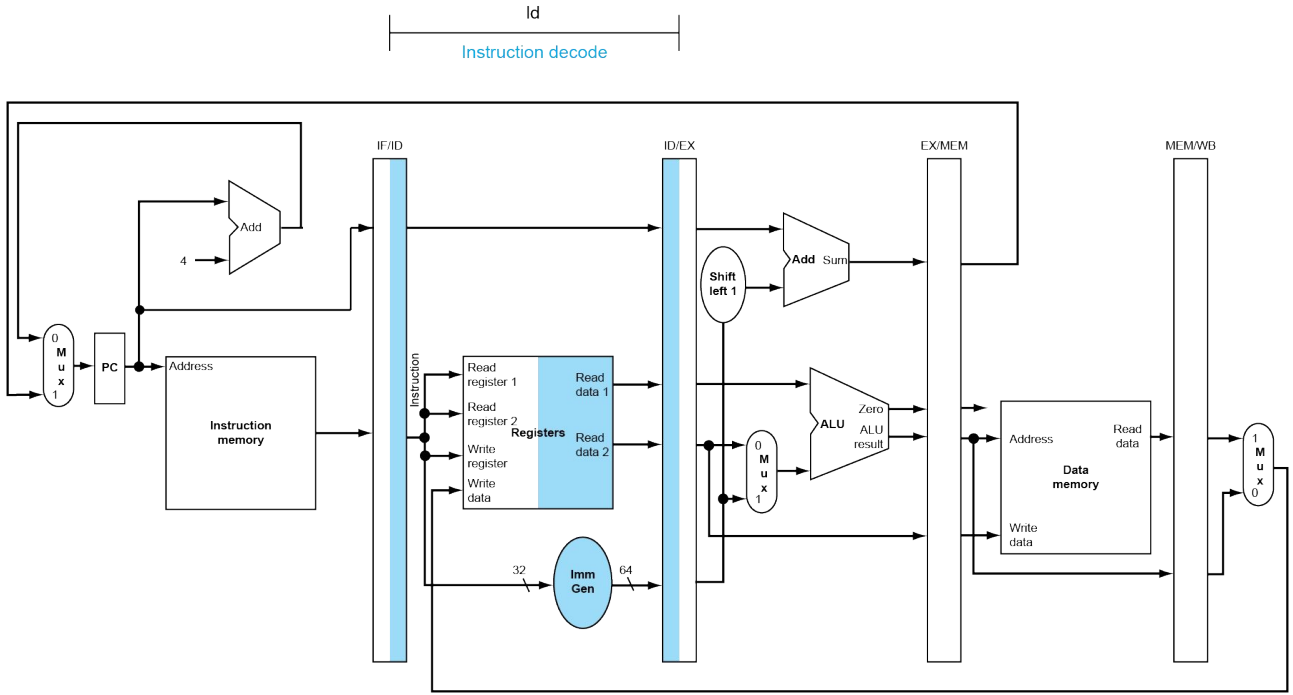
Pipelined datapath for a Load instruction (IF stage)



1. Instruction is read from memory.
2. It is placed in the IF/ID register.
3. PC updated (incremented by 4).
4. PC is also placed in the IF/ID register.

** PC needs to be stored as we don't know the instruction type yet!

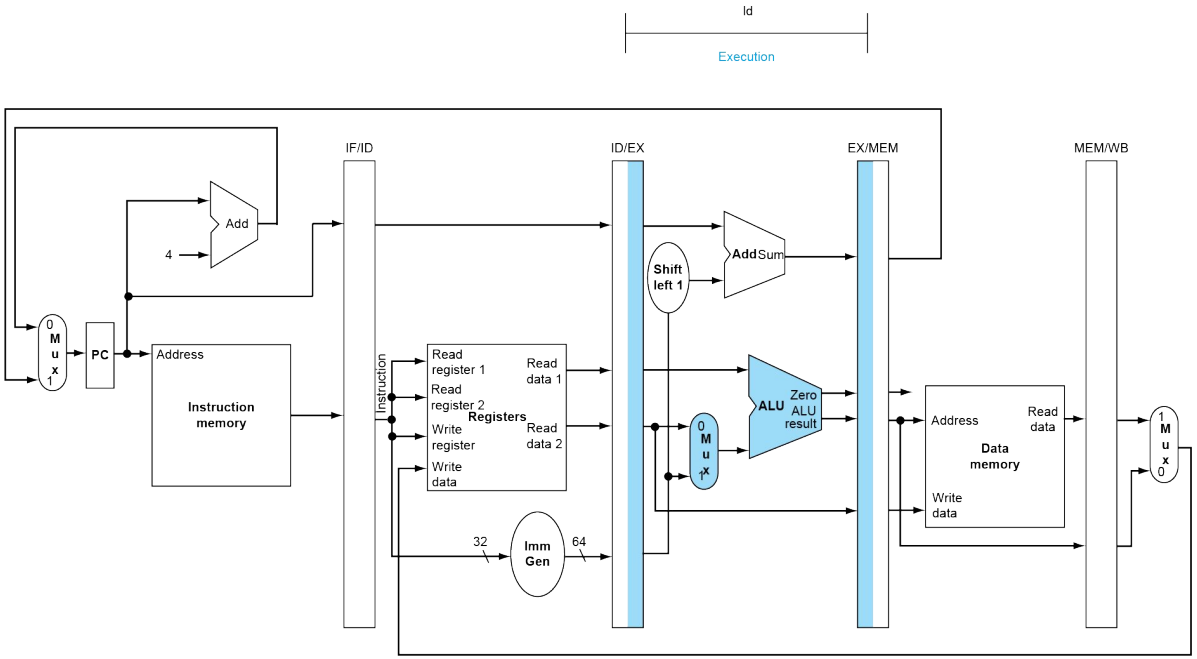
Pipelined datapath for a Load instruction (ID stage)



The values that are written in ID/EX register:

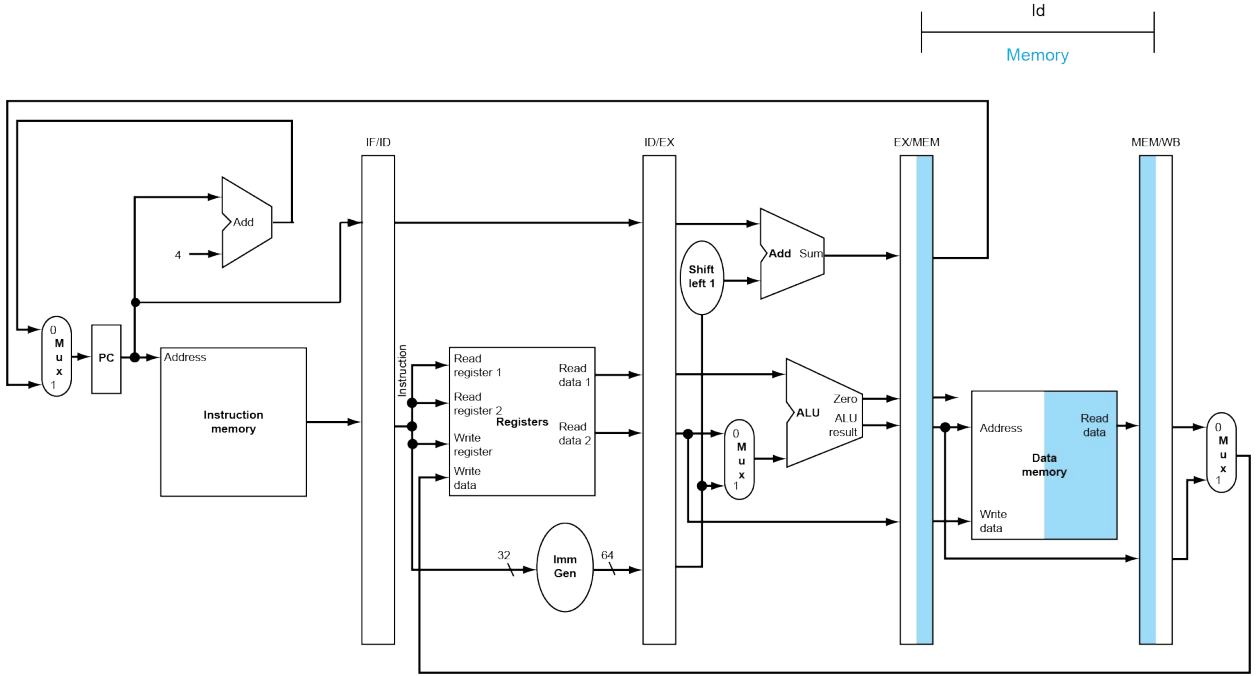
- Sign-extended immediate value
- Register numbers to read the two registers
- Value of PC

Pipelined datapath for a Load instruction (EX stage)



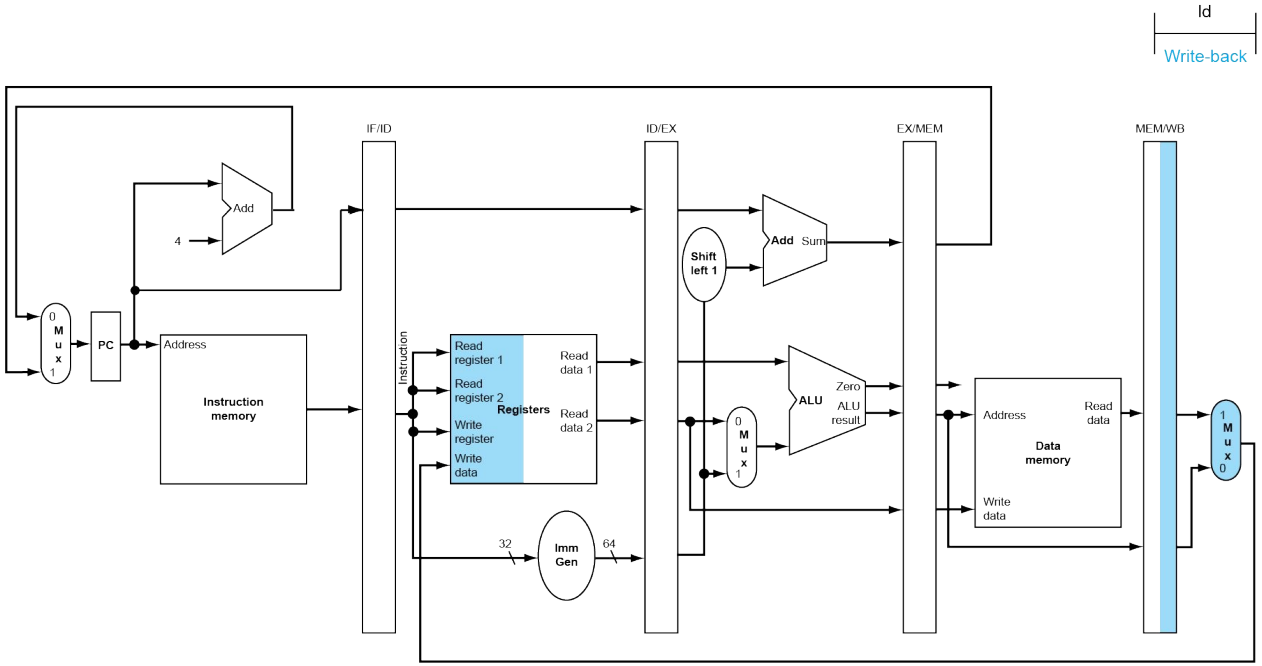
- Reads the source register and the sign-extended immediate from the ID/EX.
- Adds them using the ALU.
- Sum is placed in the EX/MEM

Pipelined datapath for a Load instruction (MEM stage)



- Address stored in the EX/MEM is used to access data memory.
- data read from memory is stored in the MEM/WB.

Pipelined datapath for a Load instruction (WB stage)

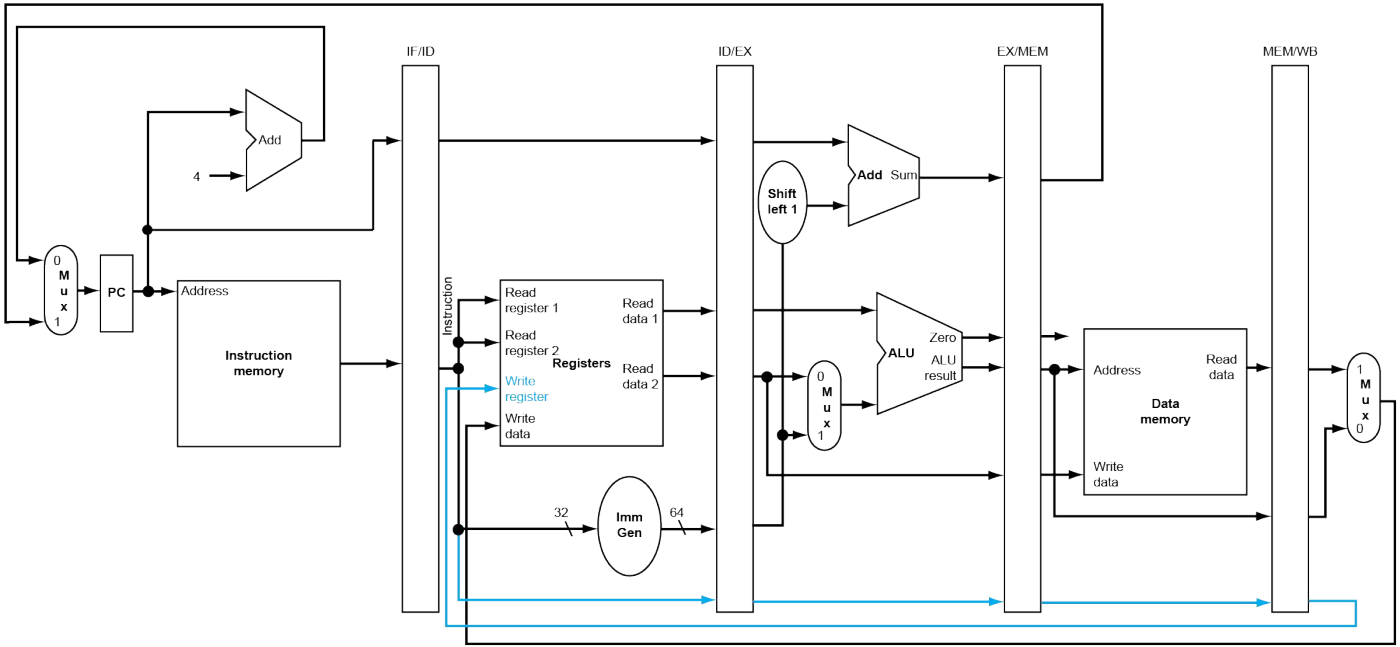


- Read the data from the MEM/WB register
- Write it back to the register file in the middle of the datapath.

There's a catch!

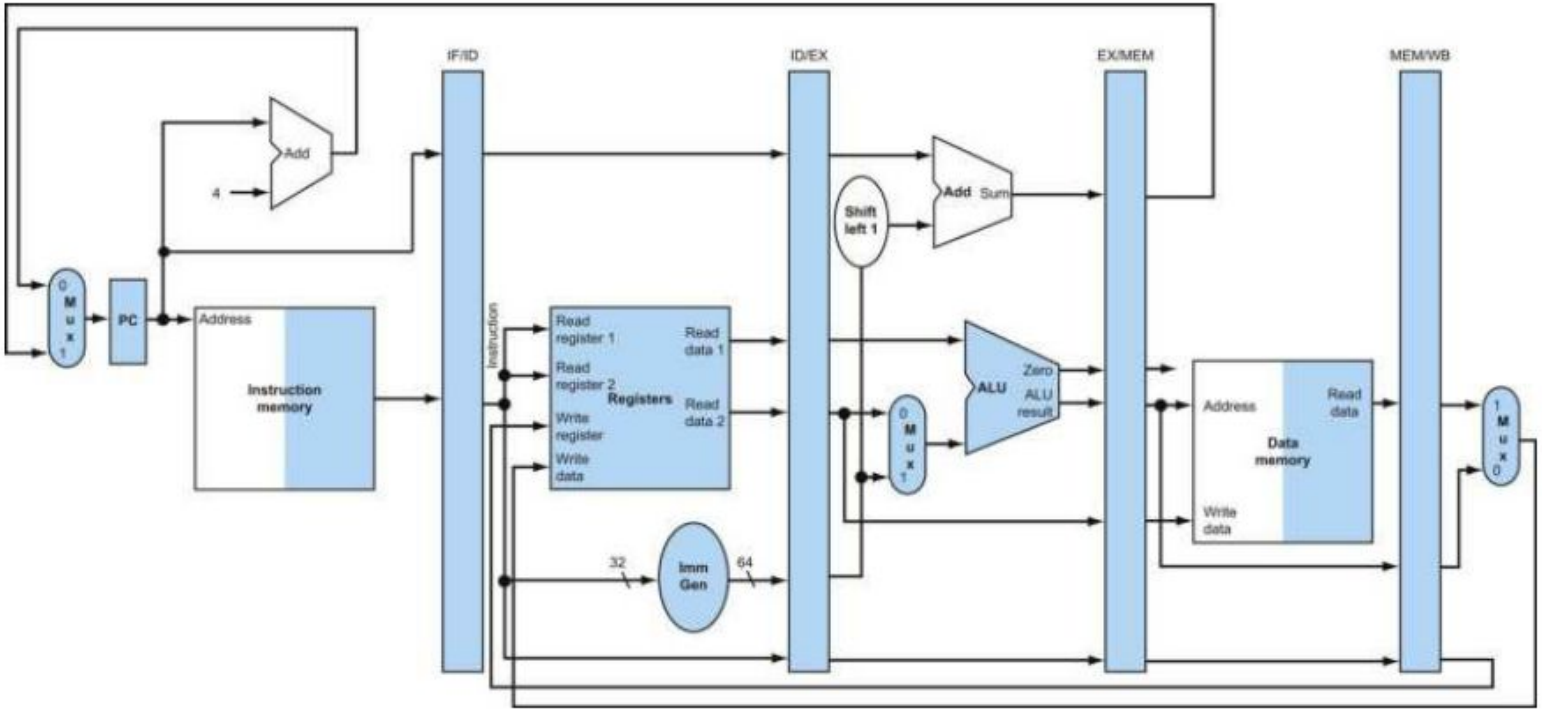
How does the WB stage know which register to write to???

Corrected Pipelined datapath for a Load instruction



- The write register number is passed through all the pipeline registers.
- Then it is passed with the data in the WB stage.

The entire datapath highlighting all the 5 stages

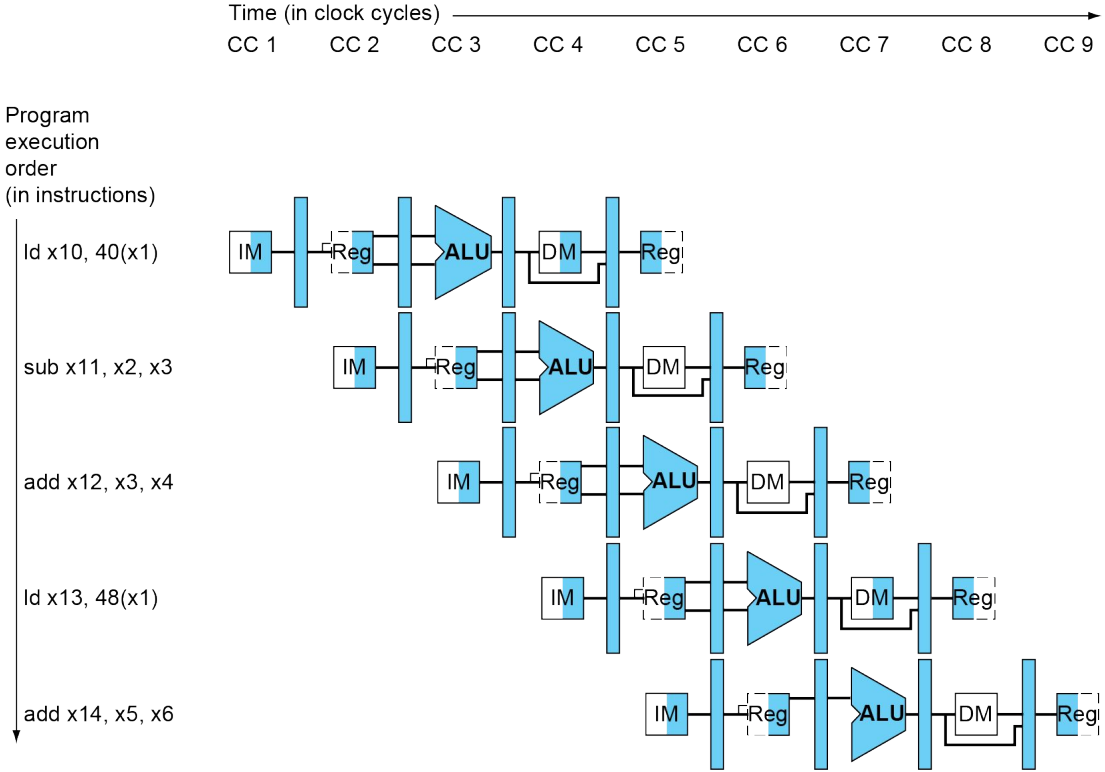


What will be the steps in each stages of the pipelined datapath of a Store instruction?

Try it yourself!

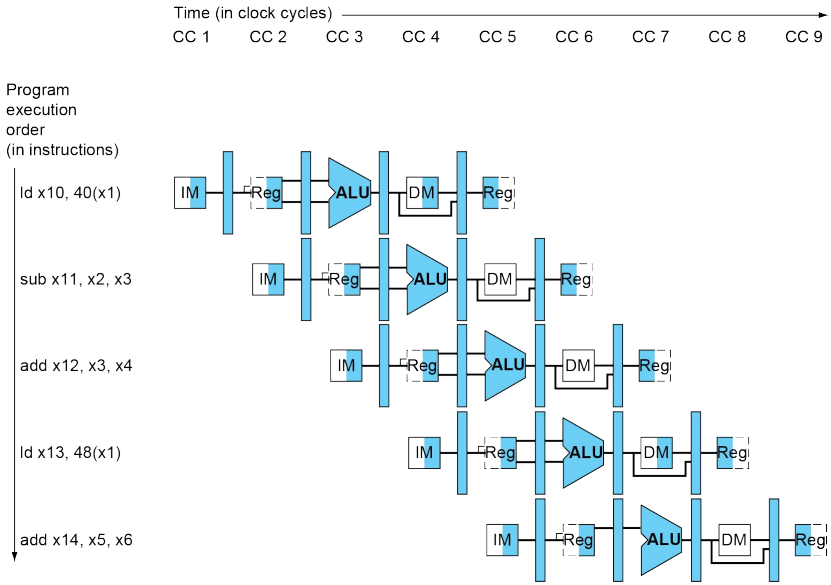
Example

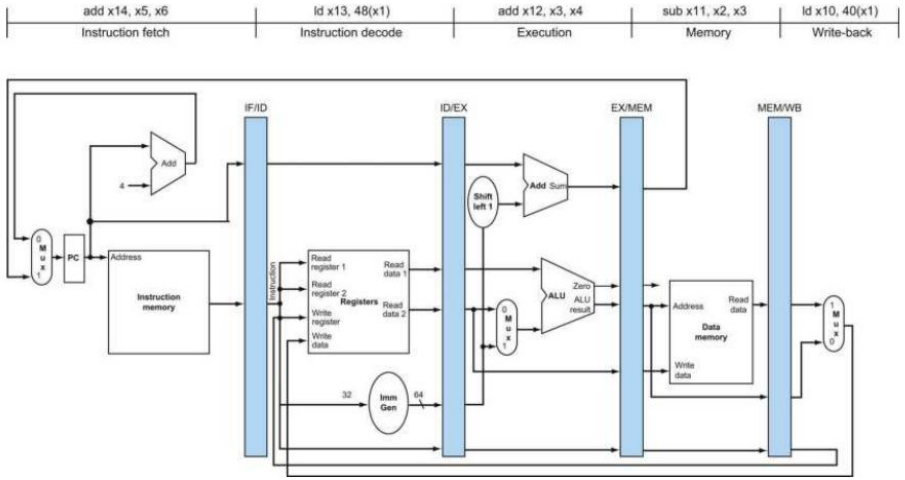
```
ld x10, 40(x1)
sub x11, x2, x3
add x12, x3, x4
ld x13, 48(x1)
add x14, x5, x6
```



Multiple-clock-cycle pipeline diagram of five instructions.

Multiple-clock-cycle VS single-clock-cycle diagram





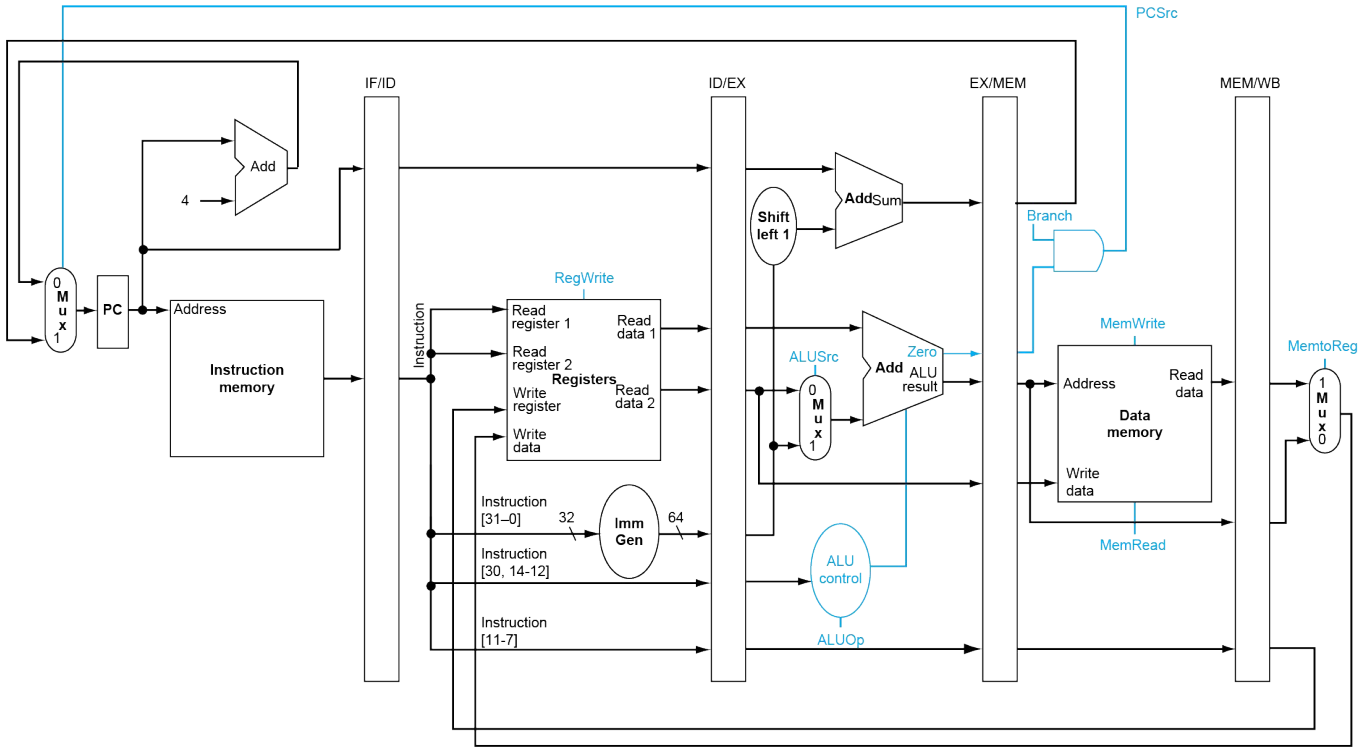
Simpler but contains no details

Highly detailed datapath of single clock cycle

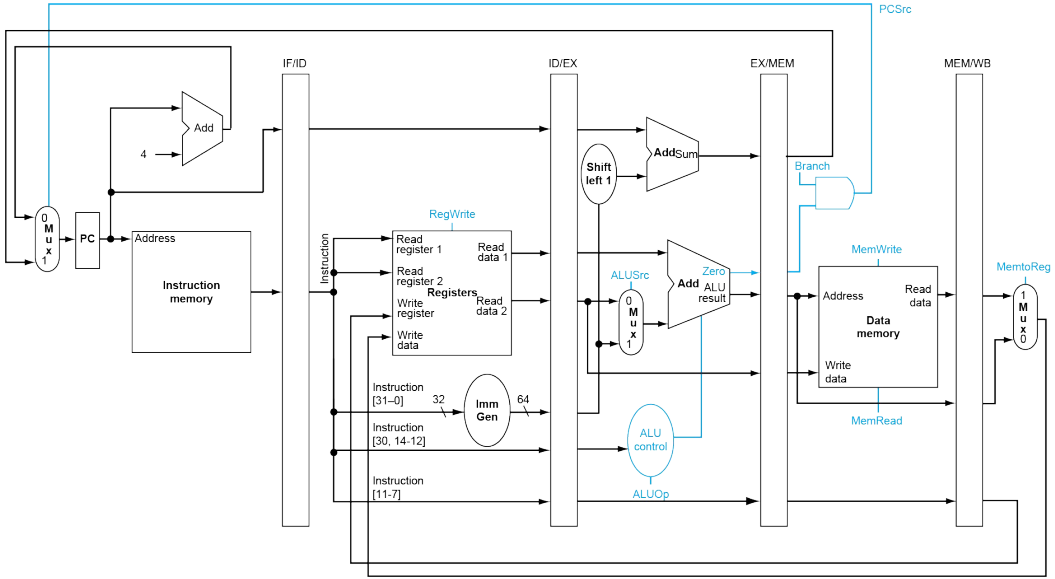
Multiple-clock-cycle pipeline diagram of five instructions.

Single-clock-cycle pipeline diagram of CC5.

Pipelined Control (Simplified)



Pipelined Control



No write signal is needed for:

- PC
- Pipeline Registers (IF/ID, ID/EX, EX/MEM, and MEM/WB)

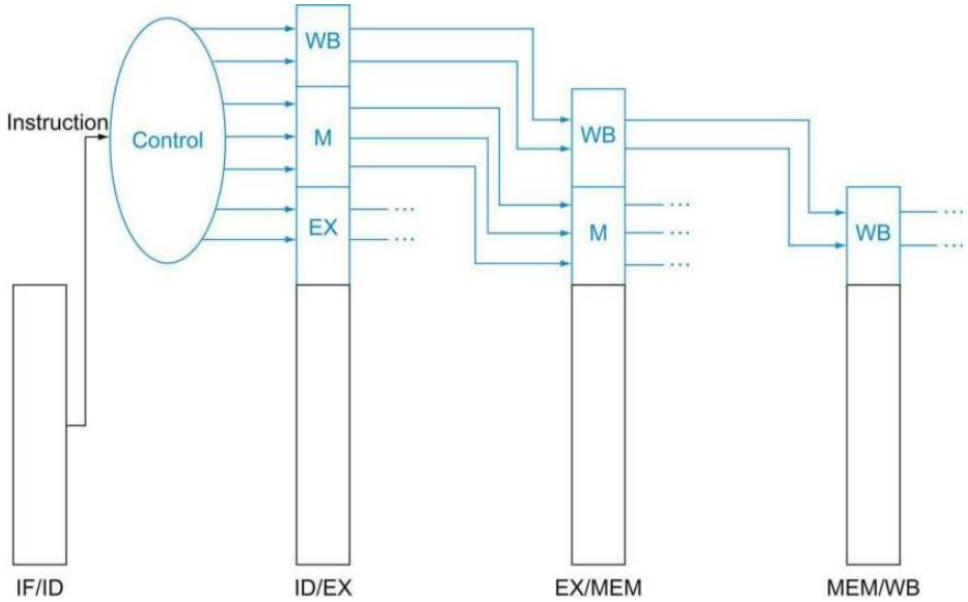
as they are written during each clock cycle.

Pipelined Control

- No control line is required for the first 2 stages.
- Control lines start from the EX stage.
- So, create the 7 control signals during ID stage.

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

Pipelined Control

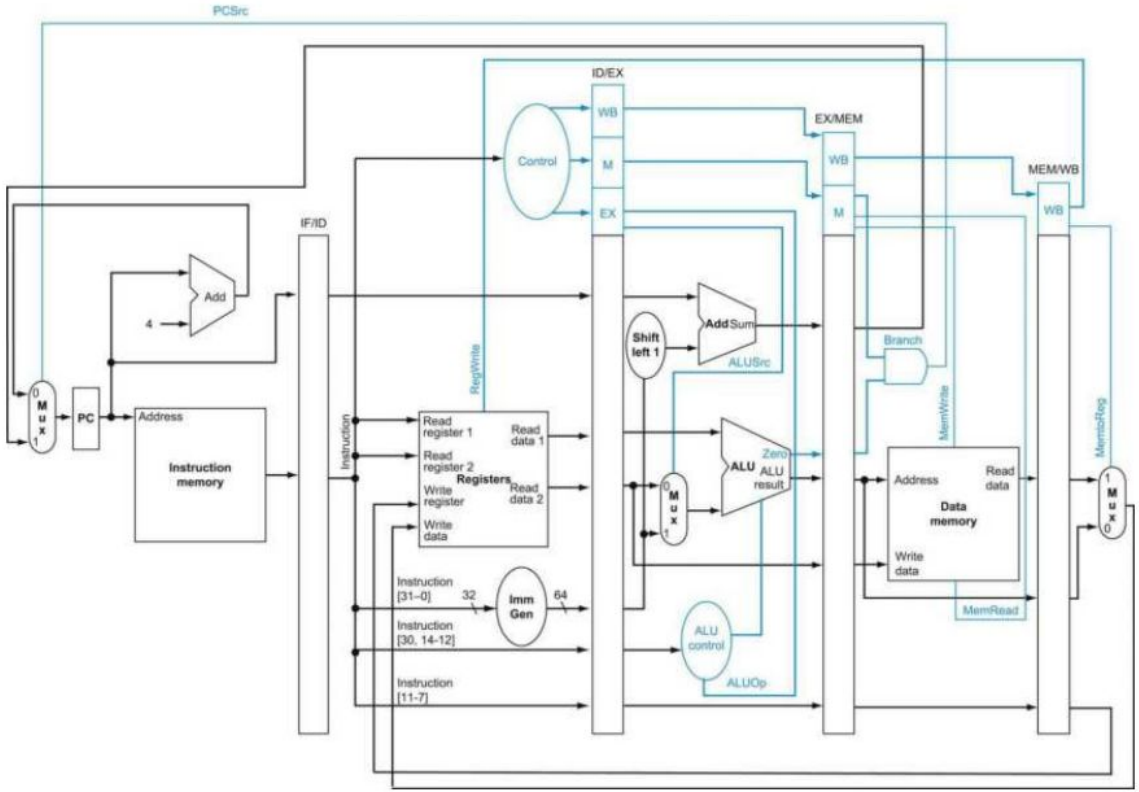


To pass these control signals through the stages

- Extend the pipeline registers to include control information

Fig: The seven control lines for the final three stages.

Full Pipelined Datapath with Control Lines

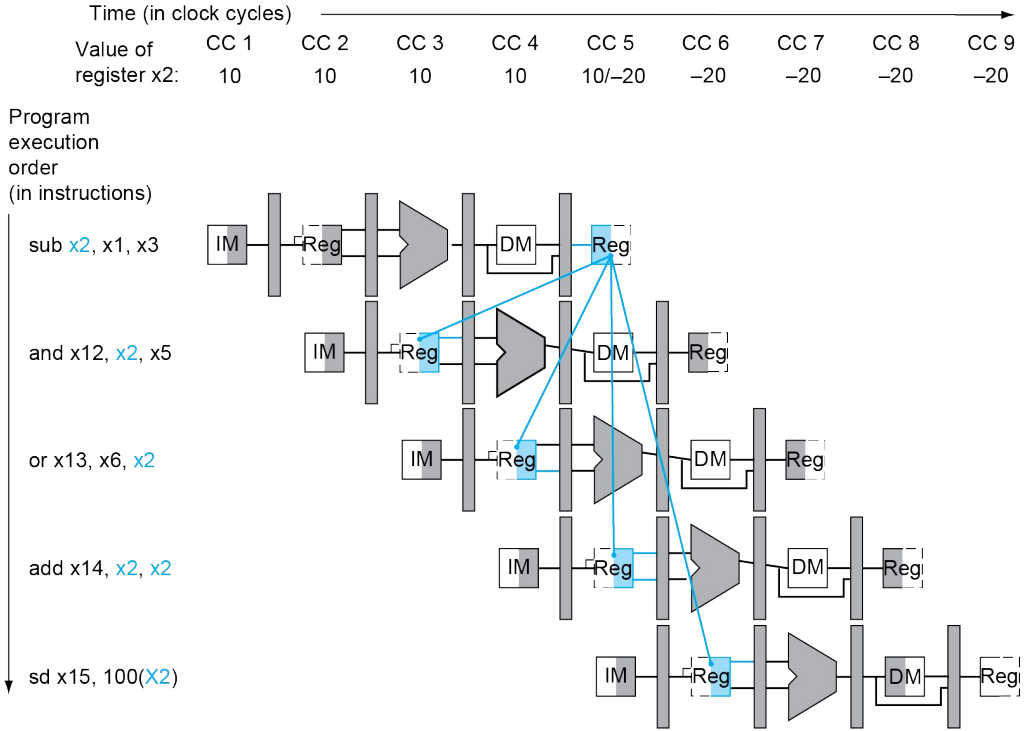


Data Hazards

- Occurs when one instruction is dependent on another.
- The 5 instructions shown are dependent on one another. How to deal with this situation?

```
sub x2, x1, x3 // Register z2 written by sub
and x12, x2, x5    // 1st operand(x2) depends on sub
or x13, x6, x2 // 2nd operand(x2) depends on sub
add x14, x2, x2    // 1st(x2) & 2nd(x2) depend on sub
sd x15, 100(x2)    // Base (x2) depends on sub
```

Data Hazards



The two pairs of hazard conditions are:

1a. EX/MEM.RegisterRd =ID/EX.RegisterRs1
1b. EX/MEM.RegisterRd =ID/EX.RegisterRs2

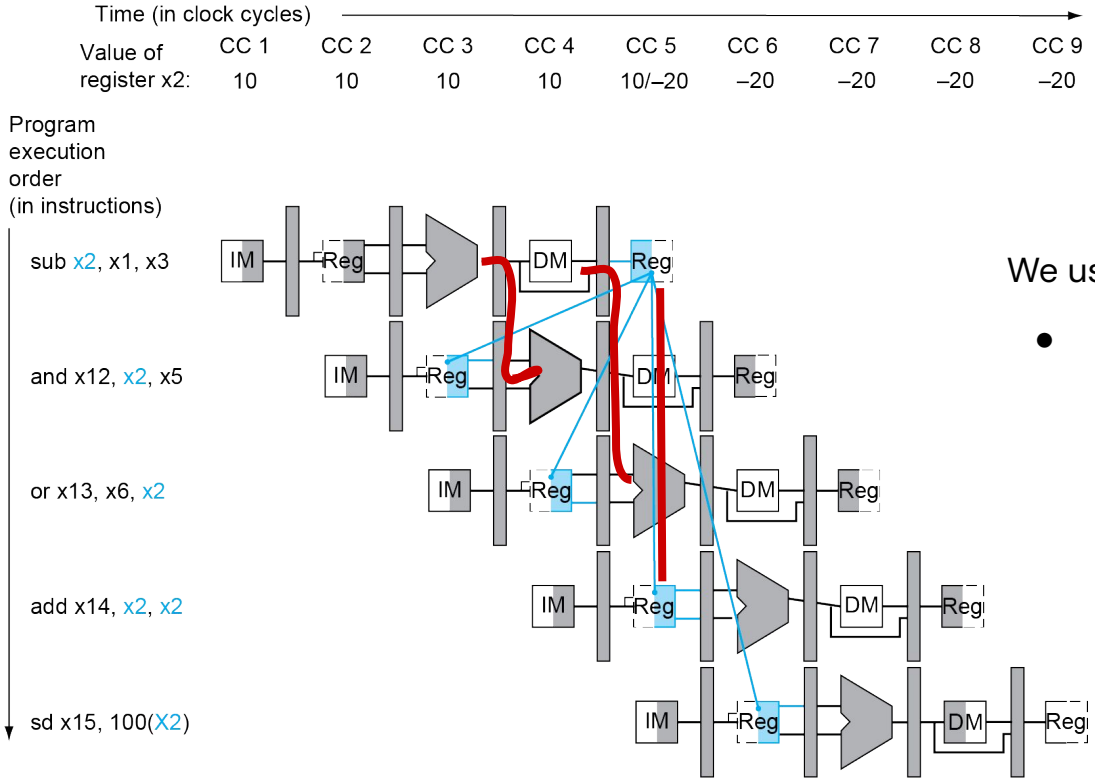
2a. MEM/WB.RegisterRd =ID/EX.RegisterRs1
2b. MEM/WB.RegisterRd =ID/EX.RegisterRs2

sub-and: hazard 1 occurs
sub-or: hazard 2 occurs

** Hazard doesn't occur for the 4th instruction bcz write occurs in the 1st half of the cycle and read occurs in the 2nd half

How to handle the hazards where data has to go backward in time?

Data Hazards (Forwarding)

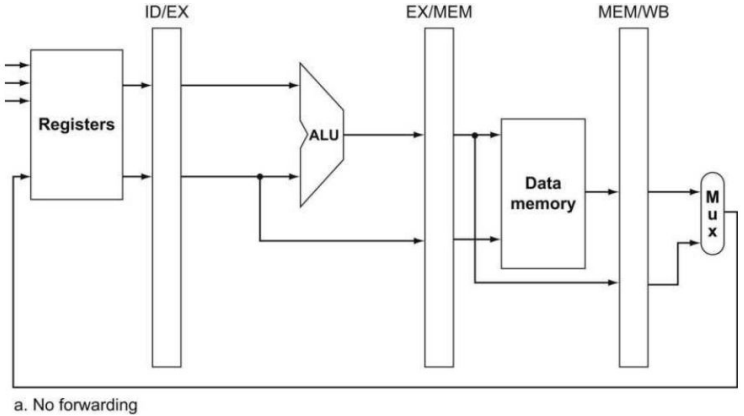


We use **Forwarding** to solve this issue.

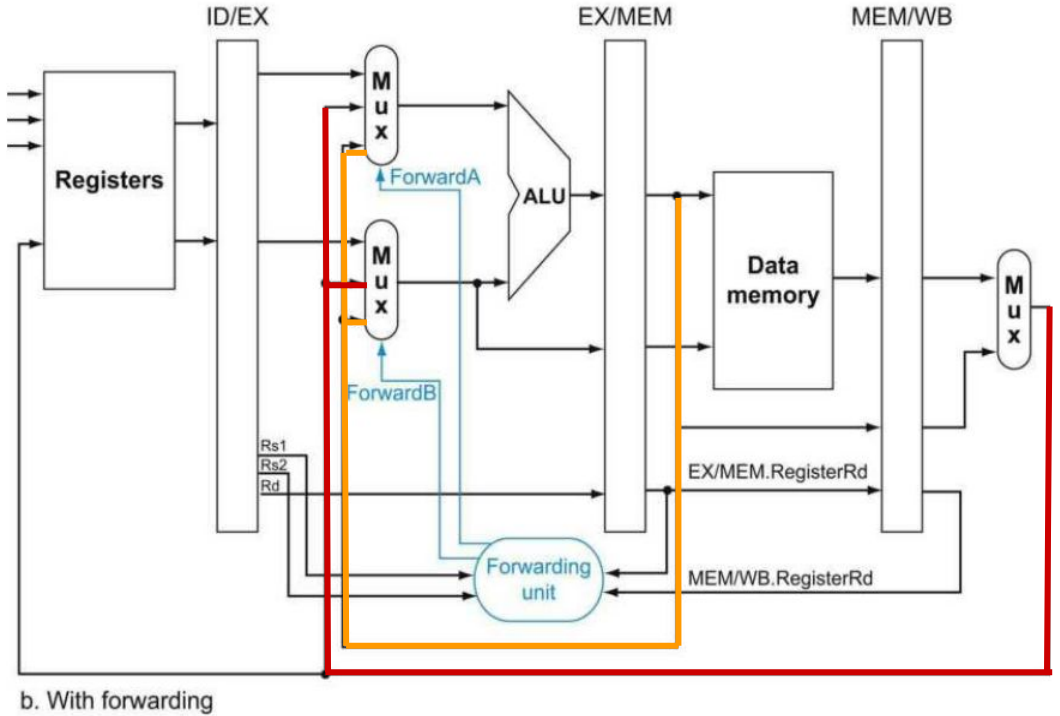
- by retrieving the missing data element from internal buffers rather than programmer-visible registers or memory.

Data Hazards (Forwarding)

- If we can take the inputs to the ALU from any pipeline register rather than just ID/EX, then we can forward the correct data.
- How do we determine whether to choose the register value or the forwarded value in the ALU?
  - Adding multiplexors to the input of the ALU.
  - Proper controls signals.

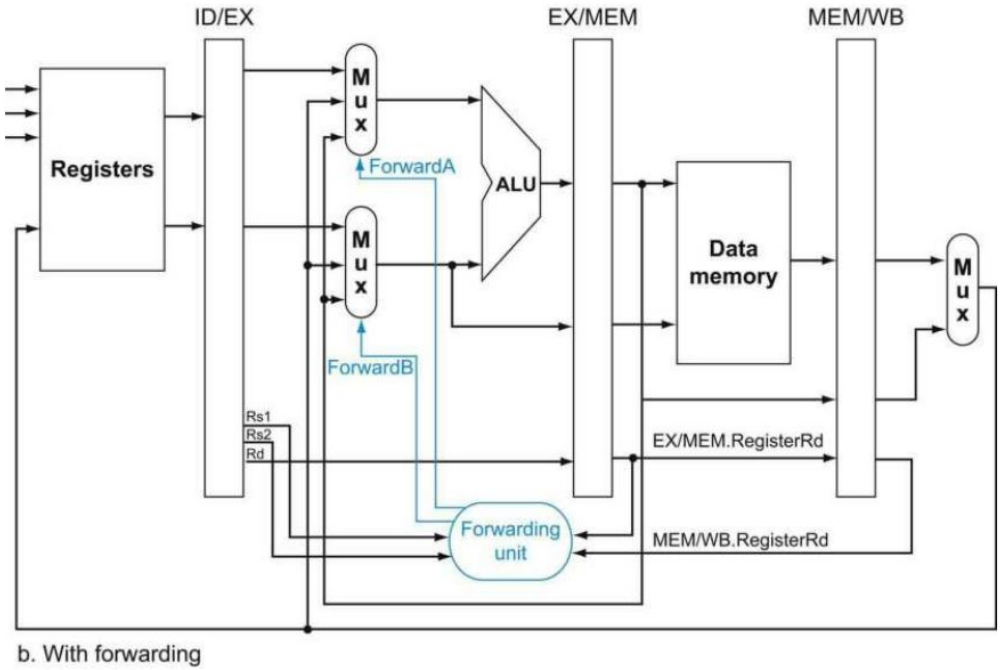


a. No forwarding



b. With forwarding

Data Hazards (Forwarding)



- For now, we're assuming the only instructions we need to forward are the four **R-format** instructions: `add`, `sub`, `and`, and `or`.
- Some instructions do not write registers. In those case we cannot forward data. **But how do we know if the instruction is writing register or not?**
  - **RegWrite** signal.
    - EX/MEM.RegWrite == 1
      MEM/WB.RegWrite == 1
- **No forwarding if Rd is x0.**
  - EX/MEM.RegisterRd ≠0 (1st hazard)
    MEM/WB.RegisterRd ≠0 (2nd hazard)

Data Hazards (Forwarding)

1a. EX/MEM.RegisterRd =ID/EX.RegisterRs1
1b. EX/MEM.RegisterRd =ID/EX.RegisterRs2

2a. MEM/WB.RegisterRd =ID/EX.RegisterRs1
2b. MEM/WB.RegisterRd =ID/EX.RegisterRs2

The control signals for 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.

Data Hazards (Forwarding)

Now let's write 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.RegisterRs1)) ForwardA = 10

if (EX/MEM.RegWrite
  and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10

2. MEM hazard:

if (MEM/WB.RegWrite
  and (MEM/WB.RegisterRd ≠ 0)
  and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
if (MEM/WB.RegWrite
  and (MEM/WB.RegisterRd ≠ 0)
  and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

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.

Data Hazards (Forwarding)

```
add x1, x1, x2
add x1, x1, x3
add x1, x1, x4
```

There's a catch! What if double data hazard occurs?

- EX and Mem hazard both occurs.

EX hazard:

if (EX/MEM.RegWrite
  and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10

if (EX/MEM.RegWrite
  and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10

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.

Corrected MEM hazard:

if (MEM/WB.RegWrite
  and (MEM/WB.RegisterRd ≠ 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
  and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01

if (MEM/WB.RegWrite
  and (MEM/WB.RegisterRd ≠ 0)
  and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
  and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

Data Hazards (Forwarding)

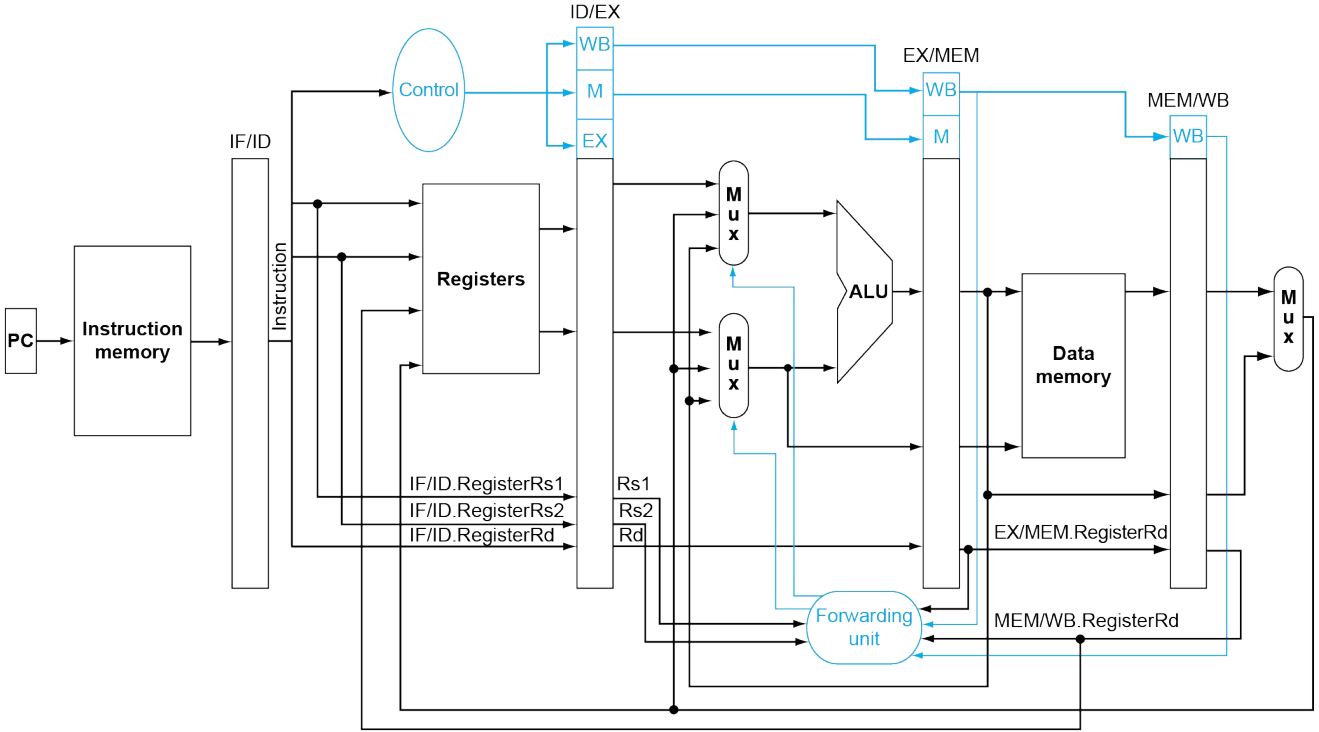


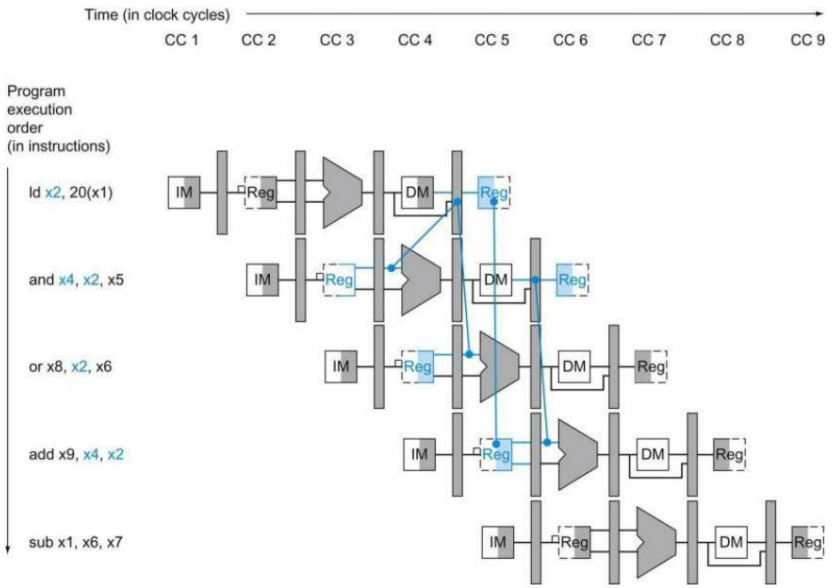
Fig: The datapath modified to resolve hazards via forwarding.

Data Hazards

Can we solve all data hazards by forwarding?

- What happens when an instruction tries to read a register following a load instruction that writes the same register?
  - Forwarding doesn't work!
  - **Stalling** is the solution!

```
lw x2, 20(x1)
and x4, x2, x5
or x8, x2, x6
add x9, x4, x2
sub x1, x6, x7
```



Data Hazards (Stalling)

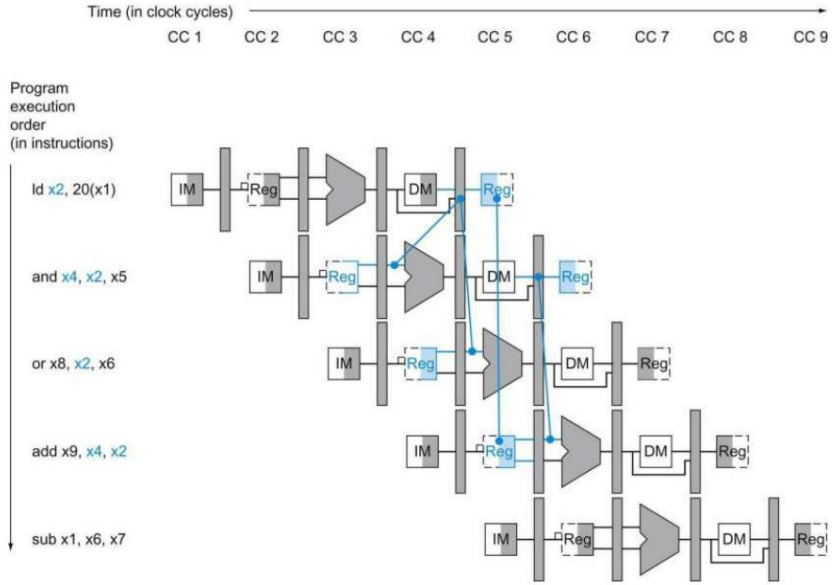
We need a hazard detection unit.

- It operates during the ID stage.
- Inserts the stall between the load and the instruction dependent on it.

Stall hazard detection condition:

```
if (ID/EX.MemRead
    and ((ID/EX.RegisterRd = IF/ID.RegisterRs1)
    or (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
        stall the pipeline
```

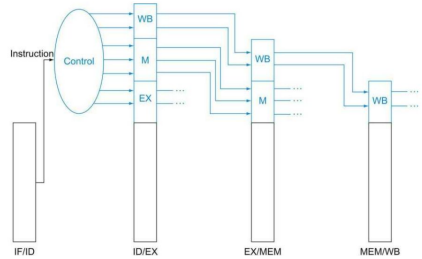
Stalling is done for 1 CC

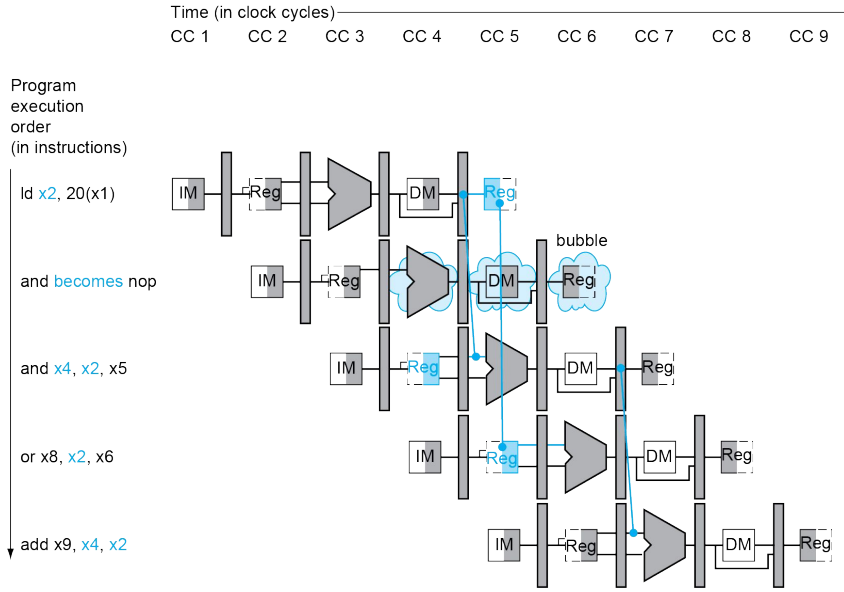


Data Hazards (Stalling)

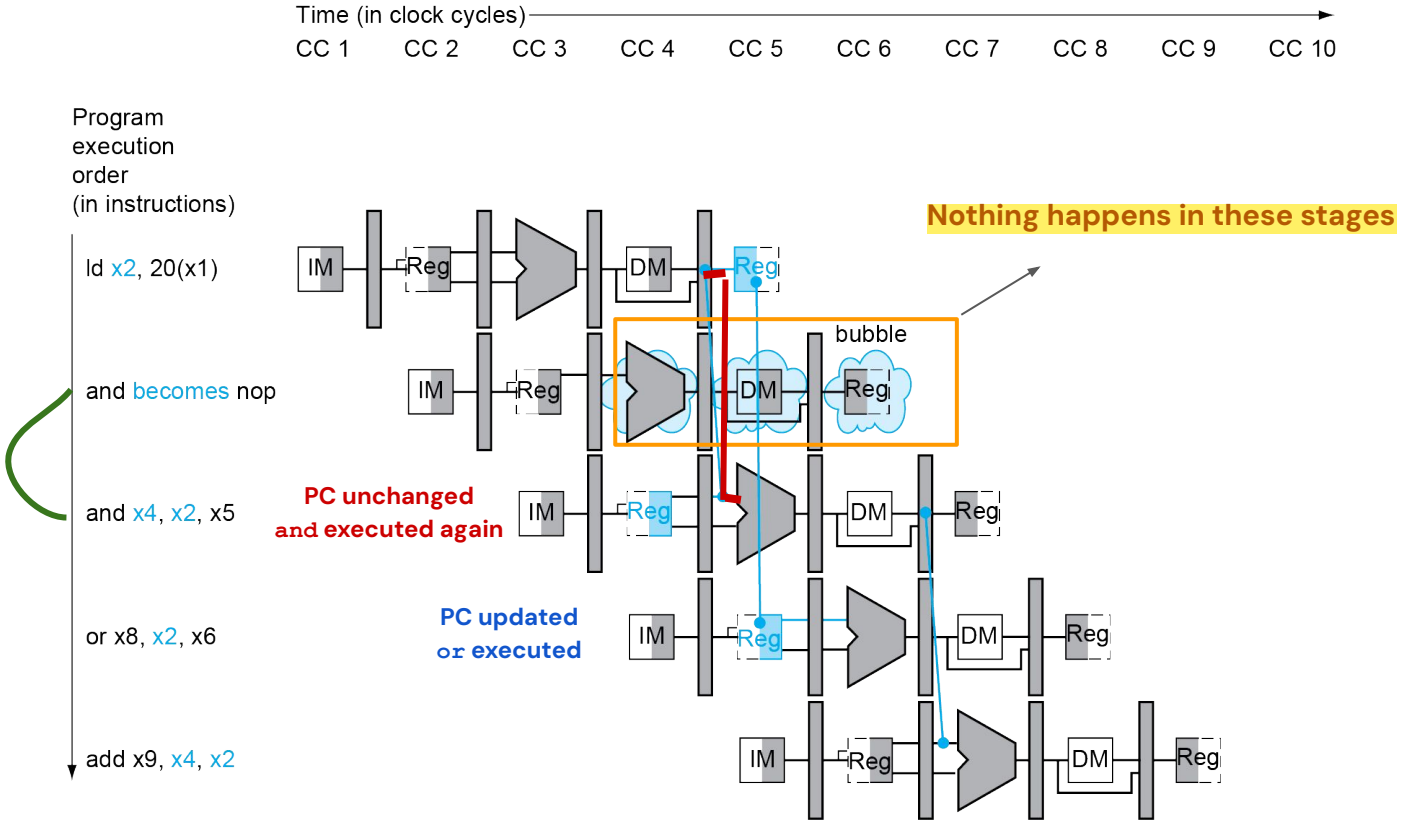
Stalling:

- When the hazard is identified in the ID stage we'll **deassert all 7 control signals** (value set to 0). It makes that instruction **nops.**
  - **nops**: An instruction that does no operation to change state.
- Keep the **PC** register and the **IF/ID** pipeline register **unchanged**.





Data Hazards (Stalling)



Reference:

1. Computer Architecture and Design RISC-V Edition, by David A Patterson and John L. Hennessy.

Thank You!