Computer Architecture Assignment 2 (Semil Jain - u1417989)

Question 1) Pipelining (40 points)

Solution: ->
Unpipelined processor time taken - 14 ns
Latch time - 0.2 ns
No. of stages to pipeline - 8
Pipelined Stages time: 1.6ns; 1.8ns; 1.4ns; 1.9ns; 2.1ns; 0.9ns; 1.7ns; 2.6ns

Assuming No Stalls in Pipeline

1. **Cycle Times**
   - Unpipelined processor
     Time to execute = T + Tovh = 14 + 0.2
     Cycle time = 14.2 ns

   - Pipelined Processor
     For an N stage pipeline with variable times per stage,
     Cycle time = Max(Times per stage) + Tovh
     = 2.6 + 0.2
     Cycle time = 2.8 ns

2. **Clock speeds**
   - Unpipelined processor
     Clock Speed = 1/(Cycle time)
     = 1/ 14.2
     Clock Speed = 70.423 MHz

   - **P**ipelined processor
     Clock Speed = 1/(Cycle time)
     = 1/ 2.8
     Clock Speed = 357.143 MHz

3. **IPCs**
   - **The number of instructions executed per cycle in both the cases is 1.**
     (Although, The cycle time for the pipelined stage is quite less. Hence, this metric alone doesn't provide us with much relevant information.)

4. **Time per instruction**
   - Unpipelined Processor
     Since, an entire instruction is processed unpipelined in a single stage.
     Time per instruction = 14.2 * 1
     = **14.2 ns**
     which is also equivalent to **1 cycle.**

   - Unpipelined Processor
     Since the instruction pipeline is divided into 8 stages.
     Time per instruction = 2.8 * 8
     = **22.4 ns**
     which is also equivalent to **8 cycles** (8 stages)**.**

     Although time taken per instruction is more, but we have parallel instructions in processing.

5. **Speedup Provided**

Speedup = (Clockspeed of Pipelined / Clockspeed of unpipelined)
= 357.143 ns / 70.423 ns
= **5.07**

6. **Speedup for 1000 stage pipeline**

Time taken per instruction per stage = (14/1000) + 0.2 (latch)
= 0.214

Speedup = Execution time (Unpipelined) / Execution time (pipelined - 1000 stages)
= 14.2 / 0.214
= 66.36

Question 2) Instructions in the 5-stage Pipeline (20 points)

Solution ->

1. **What does a load instruction do in the ALU stage of the basic 5-stage pipeline?**

- The ALU stage in a load instruction is responsible for arithmetically computing the address location in the memory from where the data is to be fetched for Loading into the register. The result is then passed onto the corresponding latch, which is picked up by the next stage (DM).
- Example:

Consider - LD R2 <- 8 [R1]
Suppose address by R1 is 1000. Here, ALU computes the address by adding offset 8 to 1000, indicating the data is at memory location 1008.

2. **Provide two example instructions that do not write to registers.**

- **BEZ R1, [R6]**

This is a branching instruction and no write is required here into the register. If a certain condition (here: value at R1 == 0) is met, then increase and set the PC to point to the next branched location contained within R6.

- **ST R6 -> 8 [R2]**

Here, we need to get the data from R6 and get the location by ALU computation (R2 + 8), which gives us the final location in the memory to store the value. Hence, no write to register is performed in this case.

3. **Provide an example instruction that writes to memory.**

- **ST R6 -> 8 [R2]**

As explained previously, Here we fetch the data to store R6, compute the memory location (R2 + 8) and write value in R6 to the computed memory.

4. **Provide an example instruction that does not use the ALU stage of the pipeline.**

- **BEZ R1, [R3]**

This branching instruction does not require any ALU computation.
If value in R1 == 0, branch to instruction located at the memory indicated by R3.

5. **Provide an example instruction that does nothing in the DM stage of the pipeline.**

- **ADD R3 <- R1, R2**

Here, we don't need anything from the memory as all values required are contained in the input registers and the result is also to be stored in a register itself.

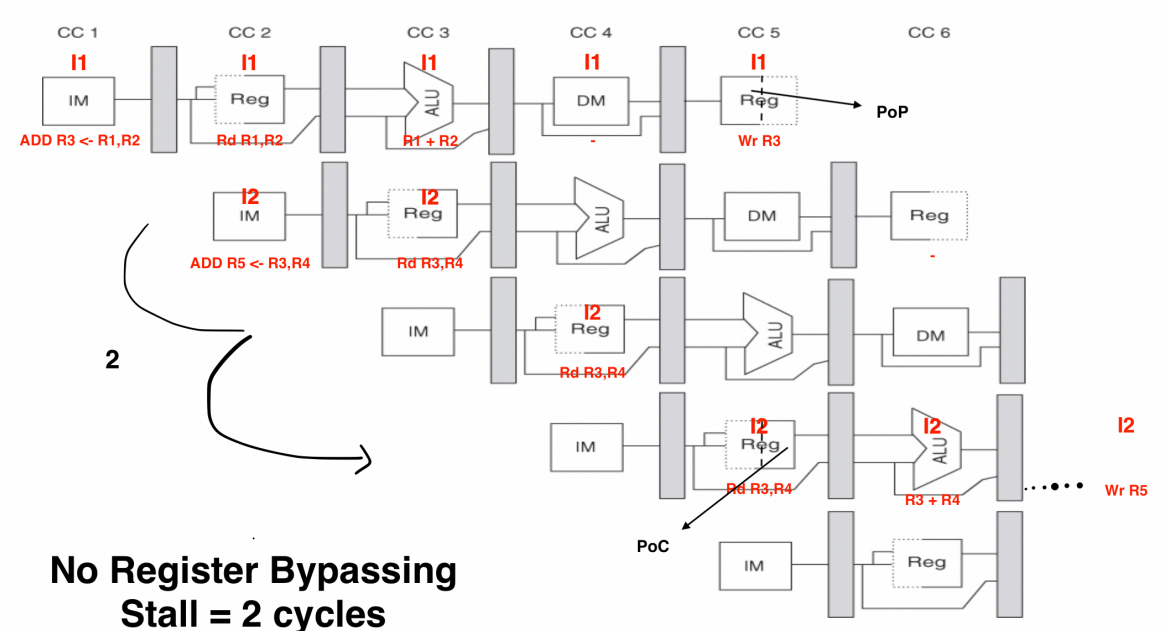
6. **Specify the number of input registers used by the following instructions: ADD, LD, ST.**

- Assuming the x86 instruction set.

a) Add - **2** (The two operands to add)
b) LD - **1** (Starting location in memory from where the data value is to be fetched)
c) ST - **2** (Data value to store and Location in memory where the data is to be stored)

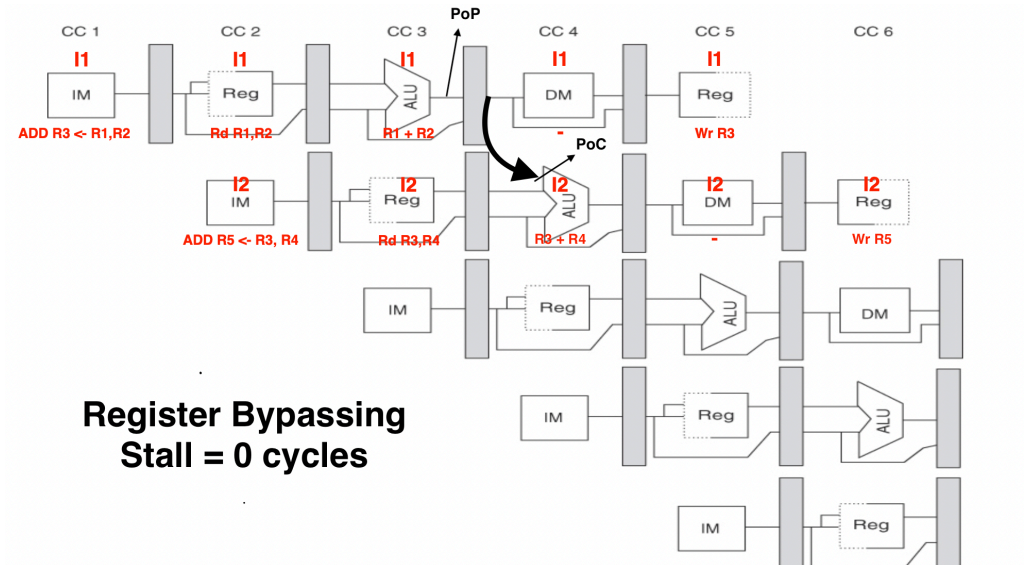
Question 3) Data Dependences (40 points) (Assuming D/RR takes 1st half of cycle to decode and next half to read. Also, Reg write happens in 1st half of the cycle)
PoP: Point of Production. PoC: Point of Consumption

1. Int-add, followed by a dependent Int-add
Example: ADD R3 <- R1,R2 ADD R5 <- R3,R4
a) No Bypassing



The operand for the second add is dependent on the result of the first instruction. Since there is no Register Bypassing, The DR stage of the 2nd instruction has to wait to read the operands until the result from the 1st instruction is stored in the register. The point of production is in the 1st half of the 5th stage and the point of consumption needs to be after that. Hence, As evident from the diagram, There are two stall cycles required before proceeding ahead.

b) Full Bypassing

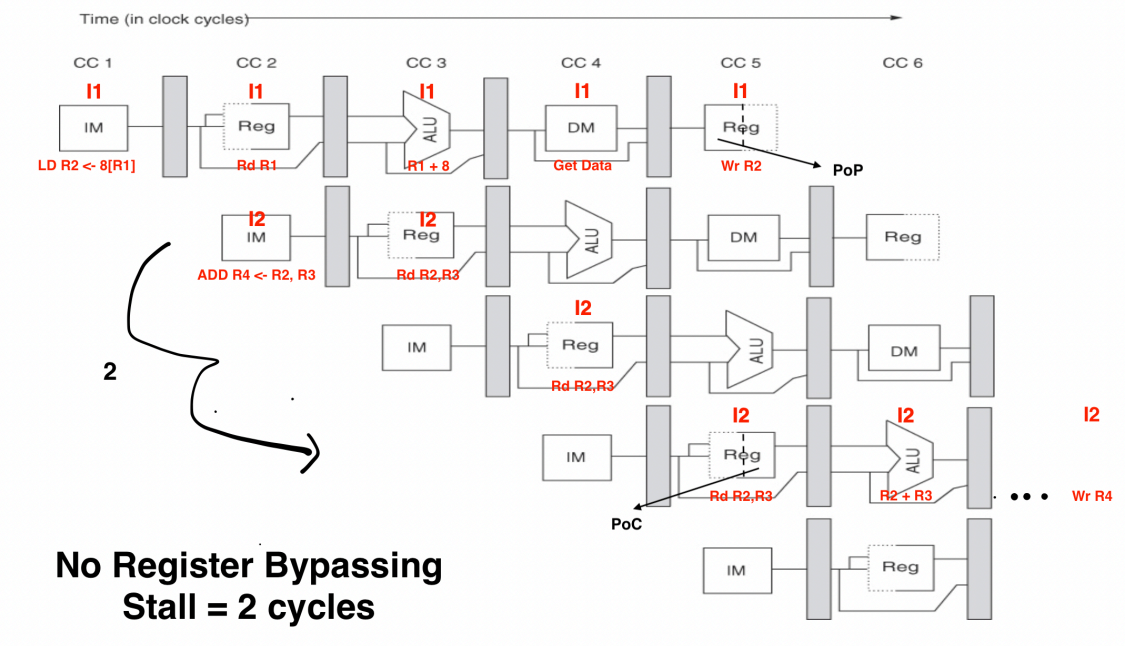


Here, Since Register bypassing is considered, instead of waiting for two cycles to get the operand from the final register, we can just take the same intermediate value from Latch L3 and use it for next computation without stalling (as seen in the diagram).

2. Load, followed by a dependent Int-add

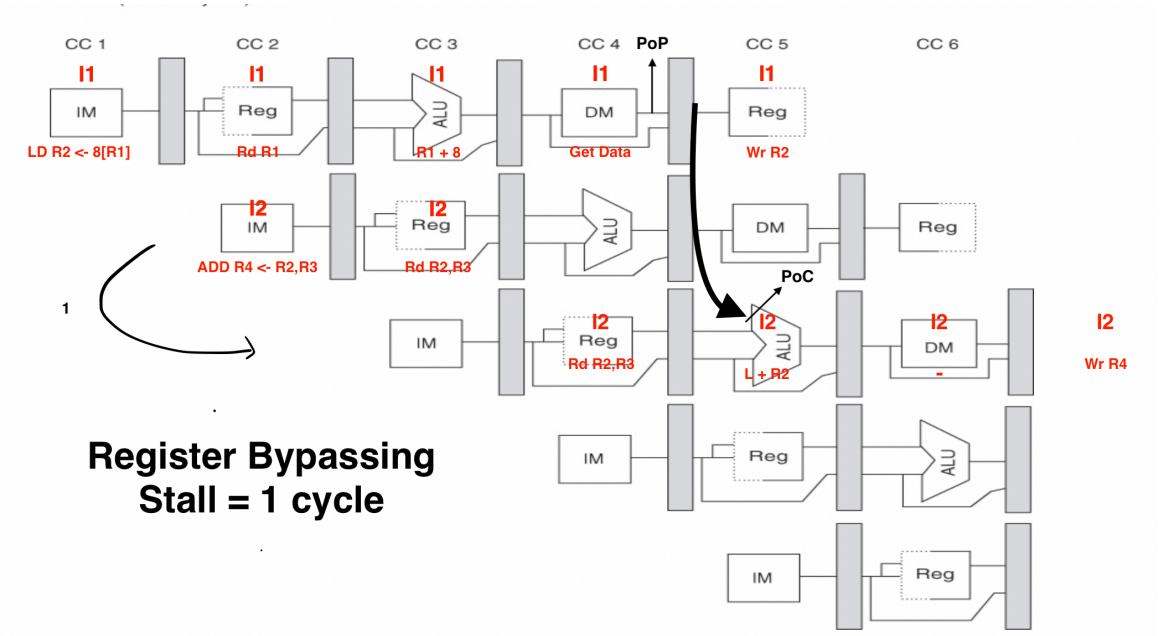
Example: LD R2 <- 8[R1] , ADD R4 <- R2,R3

a) No Bypassing



The write in register for the Load (PoP) happens at the last stage, in order to consume the value and use it in the 2nd instruction without register bypassing, we need to stall for 2 cycles until the register contains the value.

b) Full Bypassing

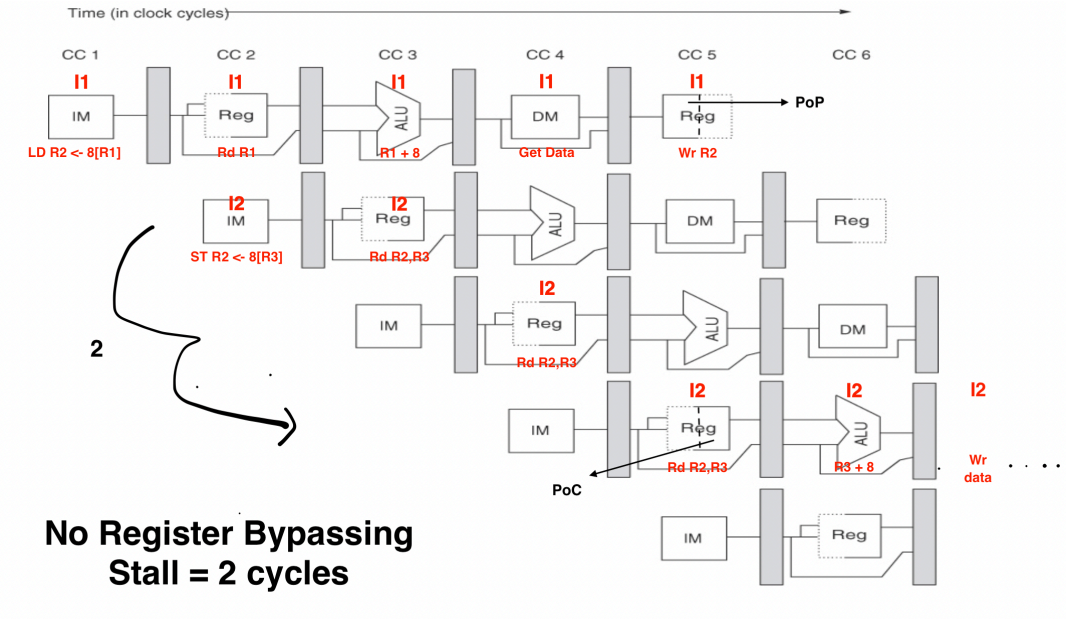


Since Bypassing is allowed, We take the R2 value for 2nd instruction from the Latch L4, instead of waiting for register write. Hence, number of stalls decreases to 2.

3. Load, providing the data for a Store

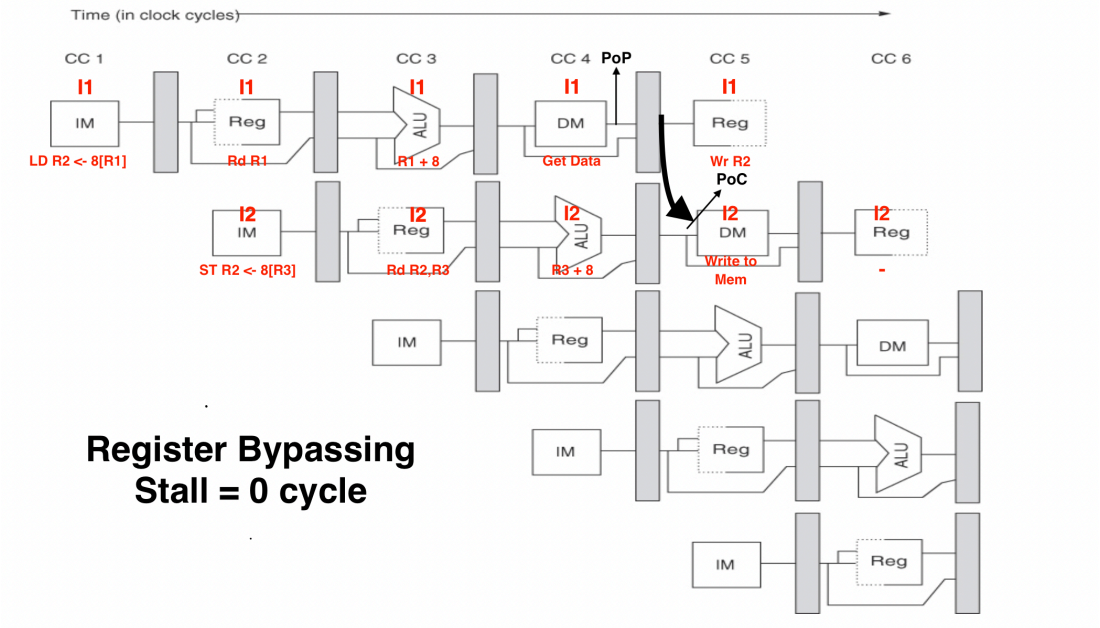
Example: LD R2 <- 8[R1] ST R2 -> 8[R3]

a) No Bypassing



Similar to the above 2 examples, In case of no stalling we have to wait for Register write to complete, which results in 2 stall cycles for PoC.

b) Full Bypassing

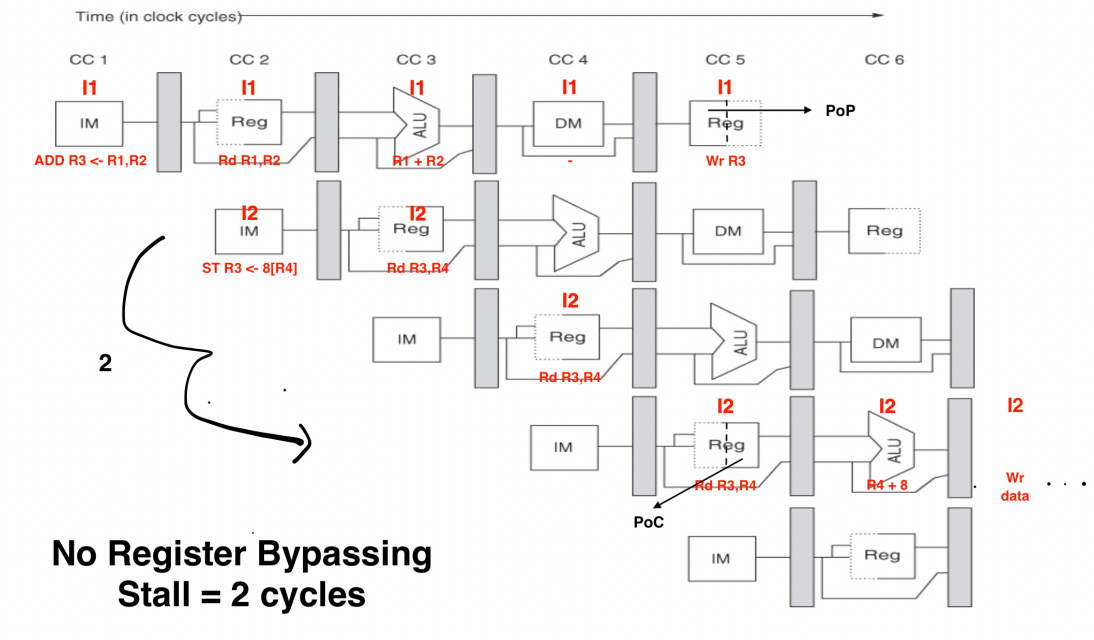


Here, we get the data in L4 and we need not wait for register write. Hence, The stall cycles is reduced to 0.

4. Int-add, providing the data for a Store

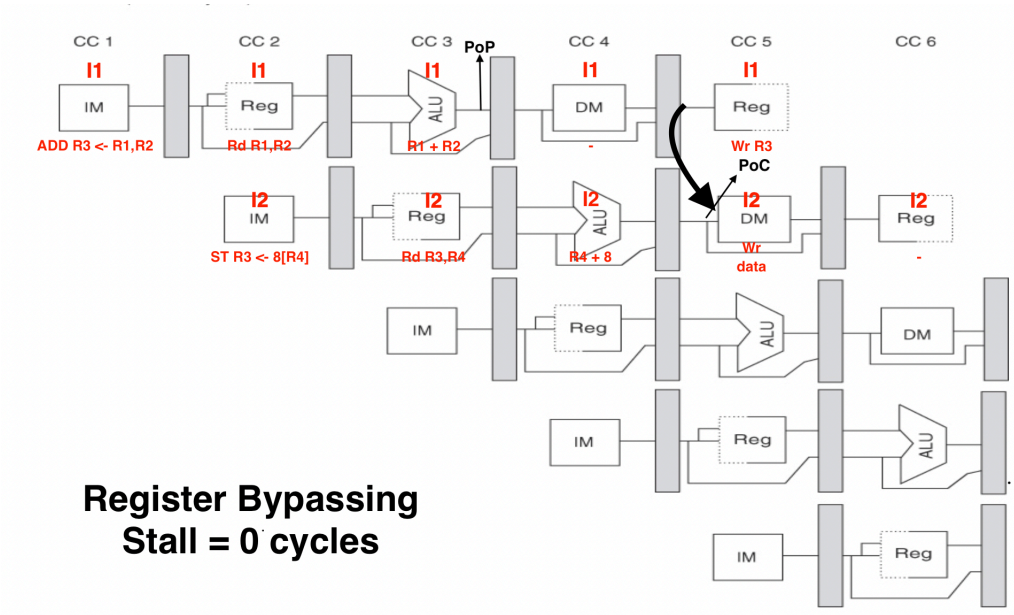
Example: ADD R3 <- R1,R2 ST R3 <- 8[R4]

a) No Bypassing



Similar to the above No Bypassing examples, There are 2 stall cycles as 2nd instruction waits for register write to complete.

b) Full Bypassing



Here, The data after add is ready in L3 after the ALU stage and passed to L4 as it is. Since the store instruction requires data to write in stage 4, we get it from L4. Hence no stalls.