Lecture 4-6: August 16 and 18, 2021

Computer Architecture and Organization-II

Biplab K Sikdar

Pipeline Structural and Data Hazard

The issues that limit pipelining called pipeline hazards. Hazards prevent the next instruction from execution during its designated clock cycles.

Speed up in K-stage pipeline

$$Speedup = \frac{CPUtime\ non-pipelined}{CPUtime\ pipelined} = \frac{n \times K \times CT}{((n-1)+K) \times CT}$$

n is the number of instructions.

If K = 5 (Figure 1) and n = 5,

$$Speedup = \frac{5 \times 5 \times CT}{(4+5) \times CT} = \frac{25}{9} = 2.77.$$

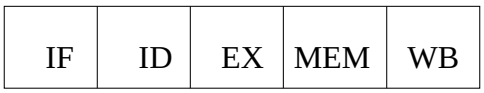


Figure 1: 5-stage instruction pipeline

For 10000 instructions,

$$Speedup = \frac{5 \times 10000 \times CT}{(4+10000) \times CT} = \frac{50000}{10004} = 4.998.$$

If n $>>$ K, the speedup is close to K, the number of pipeline stages.

Speedup is affected due to hazards.

0.1 Linear and Non-linear Pipeline

In linear pipeline structure, there is no feedback or feedforwad path (Figure 2).

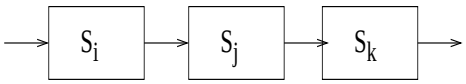


Figure 2: Linear pipeline

Pipeline of Figure 3

Can realize function F1 utilizing stages S_1, S_2, and S_3 in 3 consecutive clock cycles. On the other hand, F2 may follow S_1, S_2, S_3, S_1, and S_2. It requires 5 clock cycles. Pipeline structure that allows feedback/feedforward is a *nonlinear pipeline*.

a. Feedback in Figure 3: From stage S_3 to S_1.

b. Feedforward in Figure 4: From stage S_1 to S_3).

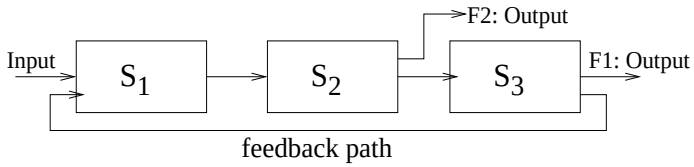


Figure 3: Feedback

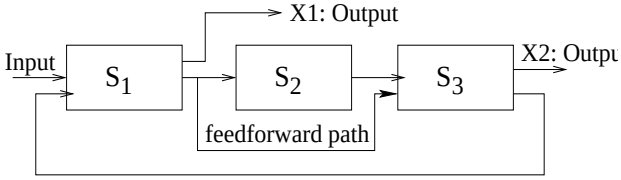


Figure 4: Feedforward

0.2 Reservation Table

Reservation table describes occupancy of pipeline stages in different clock cycles.

A row represents one resource/pipeline stage. A column represents pipeline cycle.

Figure 5(a) corresponds to linear function X. X utilizes a pipeline stage only once.

Figure 5(b) represents reservation tables for nonlinear pipeline functions.

Each pipeline function (X_1/ X_2) shown in Figure 5(b.i) / 5(b.ii) requires 6 cycles.

Multiple entries in a row denote feedback path.

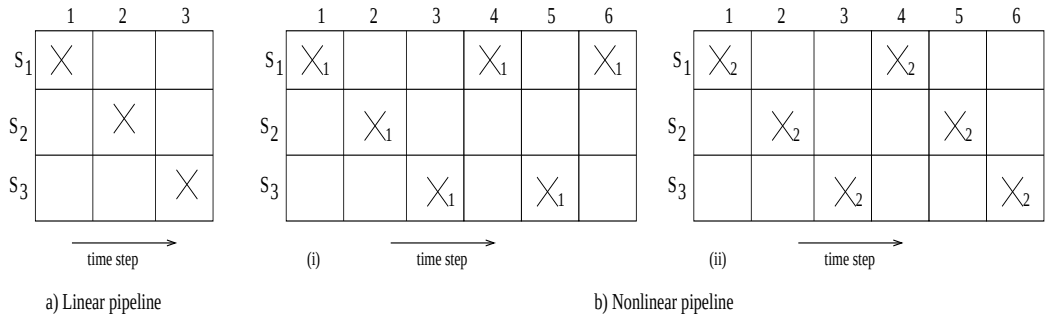


Figure 5: Reservation table a) Linear pipeline b) Nonlinear pipeline

0.3 Instruction Pipeline

In this discussion, we consider three structures of instruction pipeline.

1. 4-stage instruction pipeline:

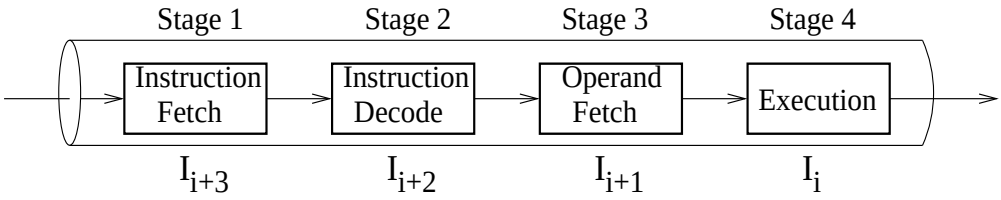


Figure 6: 4-stage instruction Pipeline

IF: instruction fetch, ID: instruction decode,

OF: operand fetch, EX: arithmetic/logic execution.

2. 5-stage instruction pipeline:

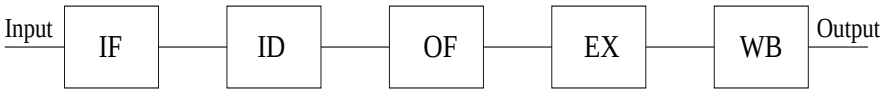


Figure 7: 5-stage instruction pipeline

3. 5-stage DLX-like instruction pipeline:

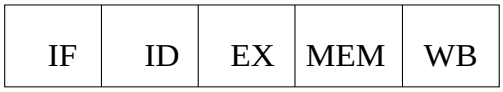
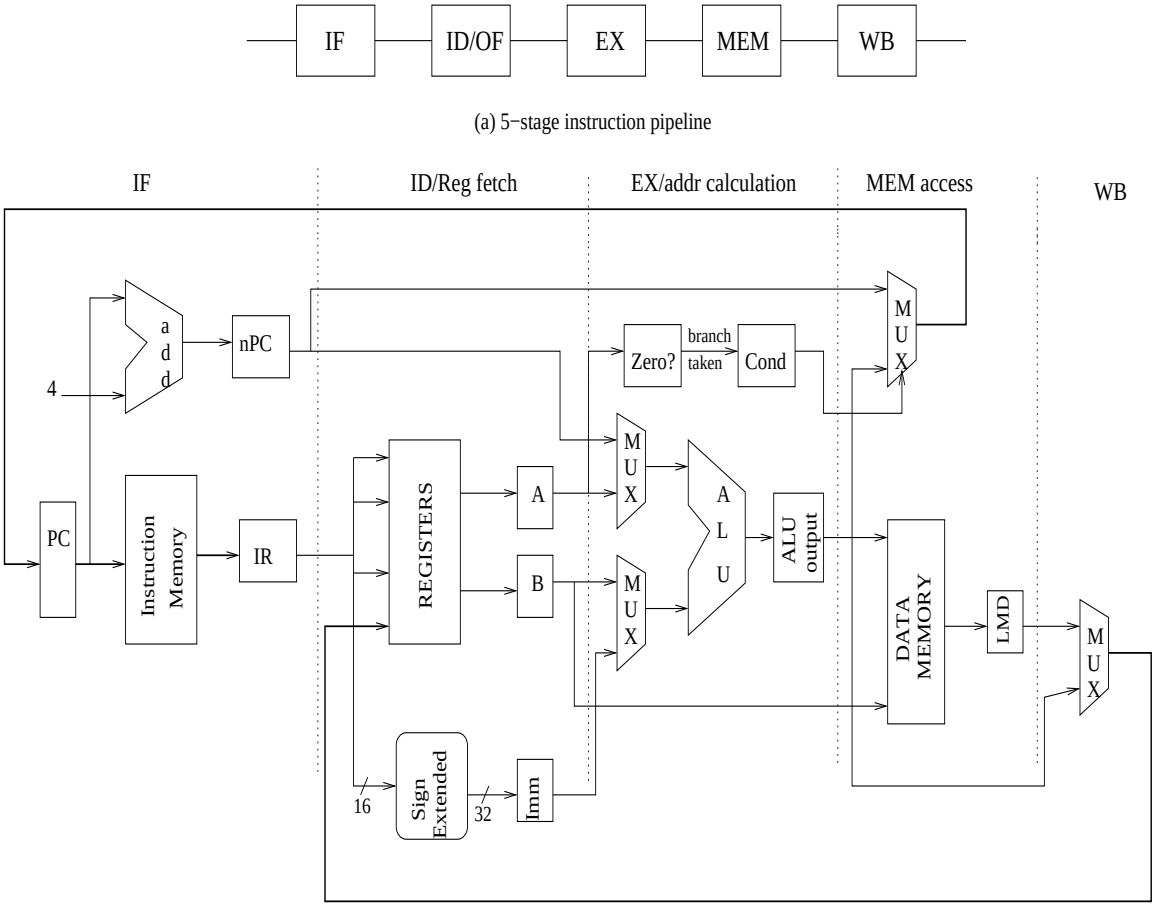


Figure 8: 5-stage DLX-like instruction pipeline

DLX (Deluxe) is a simple load-store architecture.

DLX instruction can be implemented in at most five clock cycles.

DLX is in Figure 9.



(a) 5−stage instruction pipeline

(b) 5−stage instruction pipeline datapath

Figure 9: 5-stage instruction pipeline architecture (as in H. Patterson)

1. The function IF is

$$\begin{aligned} \text{IR} &\leftarrow \text{M(PC)}, \\ \text{NPC} &\leftarrow \text{PC + 4(k)}. \end{aligned}$$

NPC holds the next sequential PC. When the current instruction is a branch, the content of PC is either the NPC or the target address computed at ALU.

2. Instruction decode/register fetch cycle (ID) performs

$$A \leftarrow Reg[IR_{6\cdots10}]$$

$$B \leftarrow Reg[IR_{11\cdots15}]$$

$$I_{mm} \leftarrow IR_{16\cdots31}$$

A, B are the temporary registers. The 16 bits $IR_{16\cdots31}$ of IR are also stored in I_{mm}. The decoding is done in parallel with reading of registers.

3. In EX, ALU operates on operands prepared in the prior cycle. It performs any one of the following four functions.

   a) Memory reference

$$ALU_{output} \leftarrow A + I_{mm}$$

   ALU_{output} is a temporary register.

   b) Register to register transfer

$$ALU_{output} \leftarrow A \text{ op } B$$

   c) Register immediate operation

$$ALU_{output} \leftarrow A \text{ op } I_{mm}$$

   d) Branch

   $ALU_{output} \leftarrow NPC + I_{mm}$; it computes branch target.

   Condition $\leftarrow$ (A op 0). The *op* is ==, ...

4. Memory access or branch completion (MEM): Only the load, store, and branch instructions are active in this stage.

   a) Memory reference

   $$\text{LMD} \leftarrow \text{Mem}[\text{ALU}_{output}]$$

   or,

   $$\text{Mem}[\text{ALU}_{output}] \leftarrow \text{B};$$

   LMD is the load memory data register.

   b) Branch

   $$\text{if (cond) PC}\leftarrow\text{ALU}_{output}\text{ else PC}\leftarrow\text{NPC.}$$

5. Write-back cycle (WB)

   a) Register-Register ALU instruction

   $$\text{Regs}[\text{IR}_{16\cdots20}]\leftarrow\text{ALU}_{output};$$

   b) Register-Immediate ALU instruction

   $$\text{Regs}[\text{IR}_{11\cdots15}]\leftarrow\text{ALU}_{output};$$

   c) Load Instruction

   $$\text{Regs}[\text{IR}_{11\cdots15}]\leftarrow\text{LMD;}$$

   Writes the result into register file.

Limit to pipelining (hazard) is due to

1. Hardware resources may not support concurrent processing of a combination of instructions. (structural hazards, Figure 10)

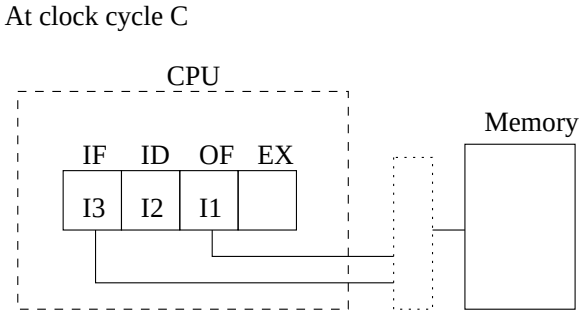


Figure 10: Structural hazard

2. Processing of instruction depends on results of prior instr. (data hazards)

   ADD R1, R2, R3
   MULT R4, R1, R5
   OR R6, R1, R7

   Instructions MULT and OR, use result (R1) generated by ADD.

3. Delay between instruction fetch and decision on control flow (branch/jump) prevents next instruction from execution. (control hazards, Figure 11)

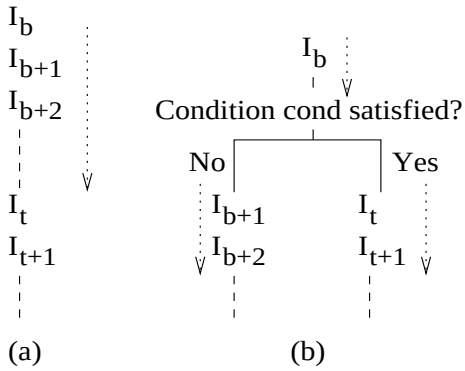


Figure 11: Branch control

0.4 Structural Hazards

Structural hazard is arisen out of resource conflicts when hardware can't support all possible combinations of instructions in simultaneous overlapped execution.

Figure 12 explains structural hazard in 4-stage instruction pipeline.

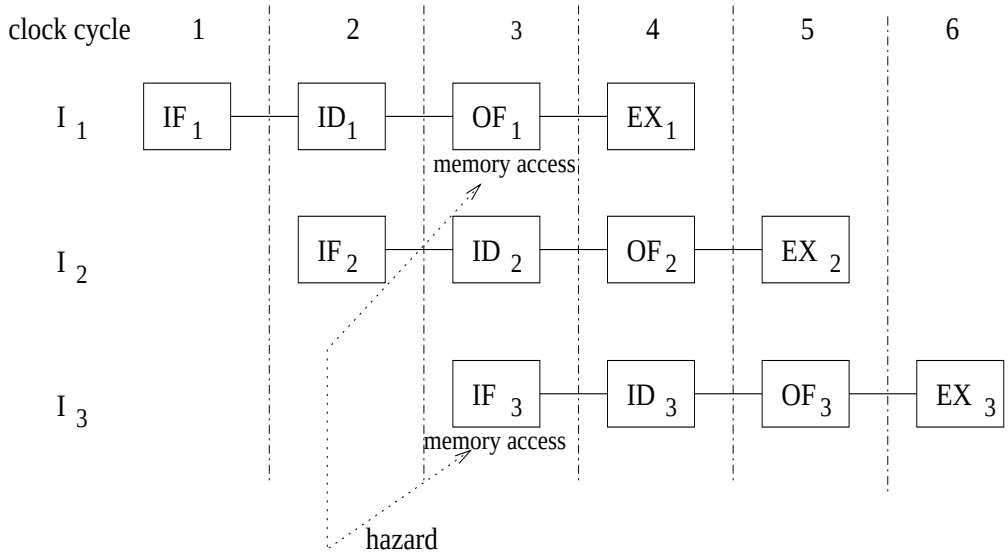


Figure 12: Structural hazard in 4-stage instruction pipe

The hazard is due to single memory shared by data (operand) and instruction.

At clock cycle 3, both operand fetching (OF_1) and instruction fetching (IF_3) units require access to memory. It can't be realizable in single port memory.

The solution can be multiport memory.

Figure 13 describes execution of the set of instructions in a 5-stage pipeline.
In clock cycle 4, there is a structural hazard after instruction I_3.

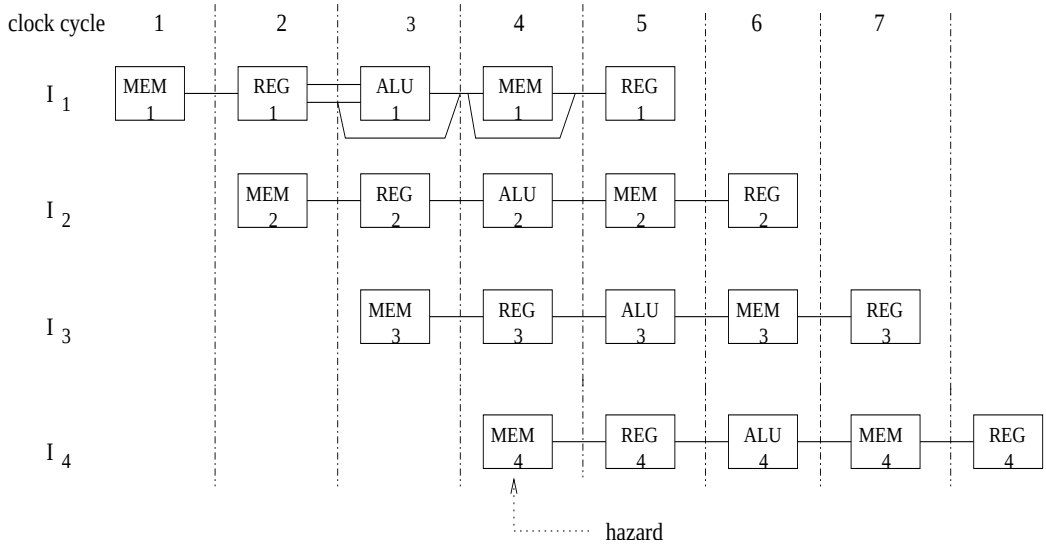
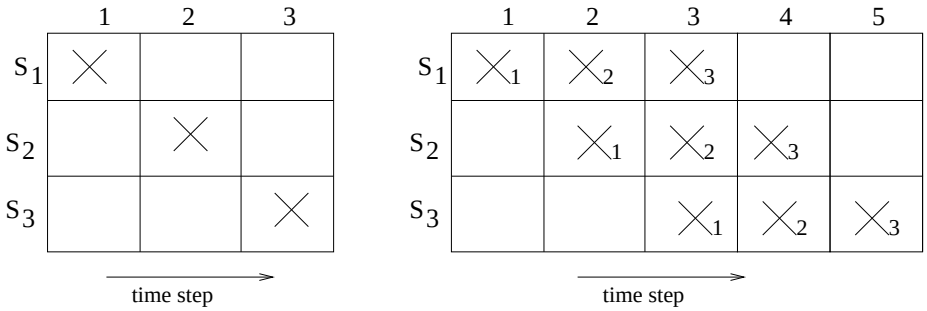


Figure 13: Structural hazard in 5-stage instruction pipe

Reservation table indicates possibility of structural hazard in pipelined system.

Multiple entries (X) in a row of RT indicate possibility of structural hazard.

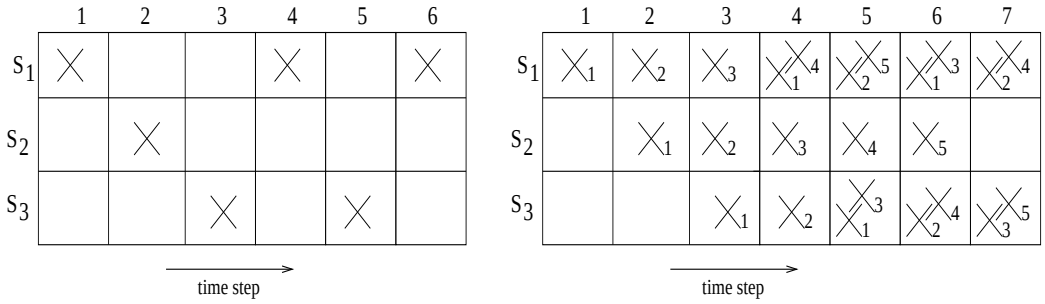


a) Reservation table b) Pipeline stages occupancy for latency 1

Figure 14: Reservation table denotes no structural hazards

3-stage pipeline corresponding to RT of Figure 14(a) is free from structural hazard.
It shows no collision (Figure 14(b)).

However, RT of 3-stage pipeline structure shown in Figure 15(a) indicates -
there are possibilities of structural hazards at S_1 and S_3 (Figure 15(b)).
There are multiple entries in row for S_1 and S_3, when latency is 1.



a) Reservation table b) Pipeline stages occupancy for latency 1

Figure 15: Reservation table denotes structural hazards

0.4.1 Solution 1: Stalls

This can be realizable with the introduction of stalls in pipeline.

When I_j requires resource that also accessed by its former instruction I_i at time t, then stalls introduce delay d for I_j so that resource can be available for I_i at $t + d$.

A stall is commonly called a pipeline bubble.

Stalls (2 nos) to avoid structural hazards in 4-stage pipeline are shown in Figure 16.

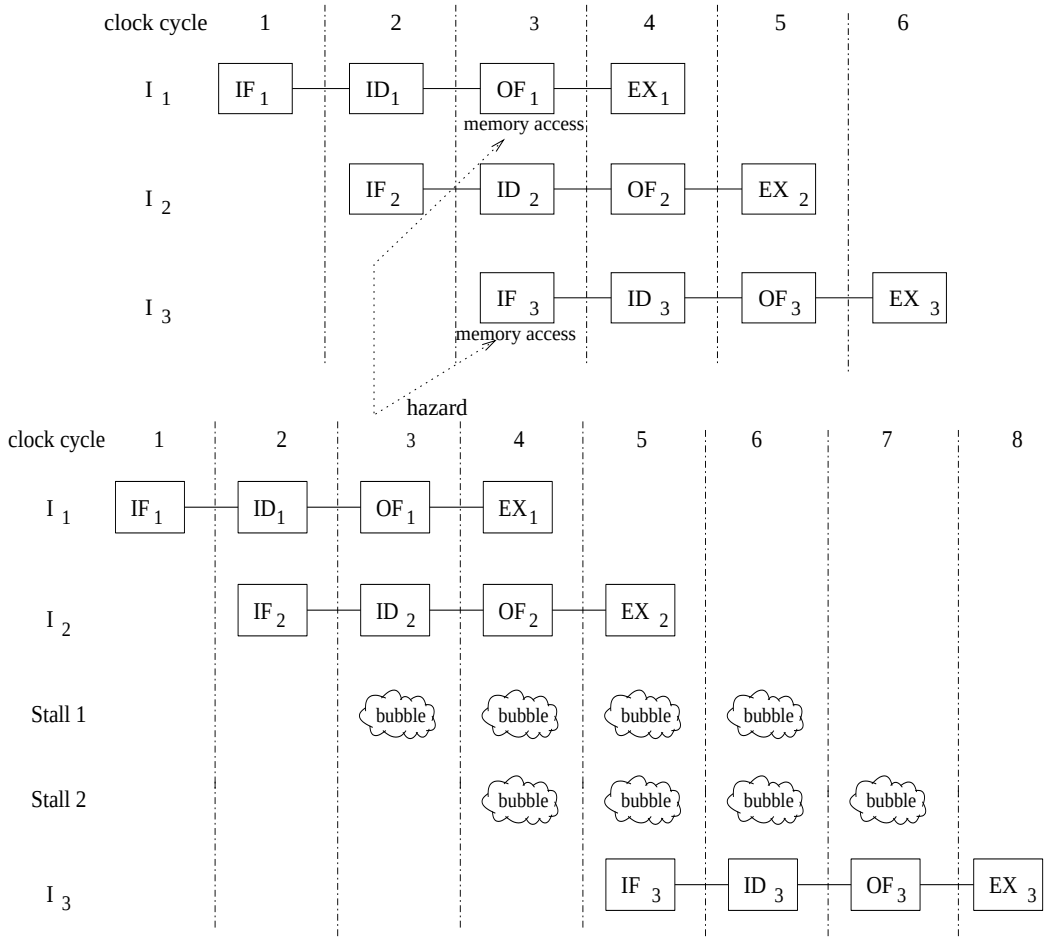


Figure 16: Stalls to avoid structural hazards in 4-stage instruction pipe

For the 5-stage pipeline, number of such stalls is 3 (Figure 17).

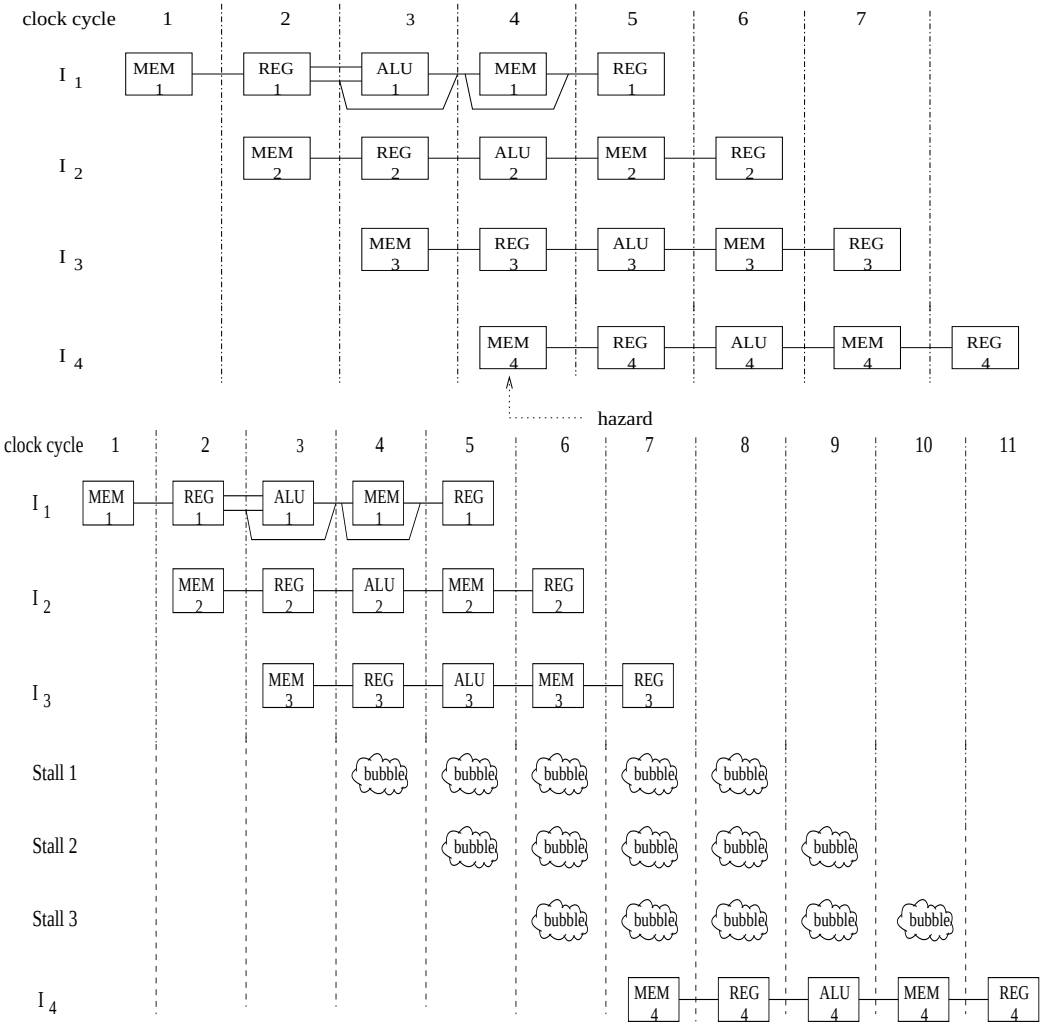


Figure 17: Stalls to avoid structural hazards in 5-stage instruction pipe

Stalls increases CPI than its usual ideal value. The CPI in stall

$$\text{CPI}_{pipeline} = \text{Ideal CPI} + \text{Average stall cycles per instruction.}$$

Stalls, therefore, reduce speed up in a pipeline.

$$\text{The speed up with stall} = \frac{Pipeline\ depth\ (number\ of\ stages)}{1 + pipeline\ stall\ cycles\ per\ instruction}.$$

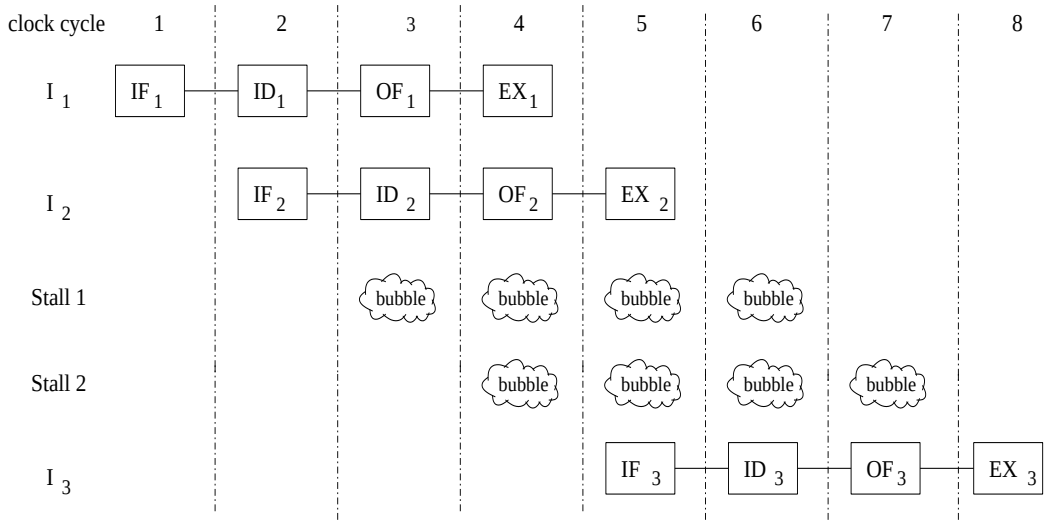


Figure 18: Stalls in the pipeline

In Figure 18, between 4 to 7 clock cycles -that is, in 4 clock cycles

We get two instructions executed.

That is, CPI = 2.

Efficiency is reduced to half.

0.4.2 Solution 2: Throw more hardware

Use multiport memory (Figure 19).

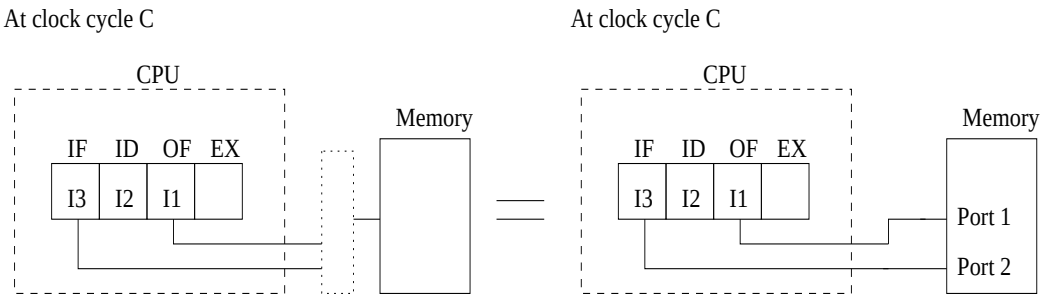


Figure 19: Structural hazard

Use I-cache and D-cache (Figure 20).

I-cache stores instructions and D-cache stores operands.

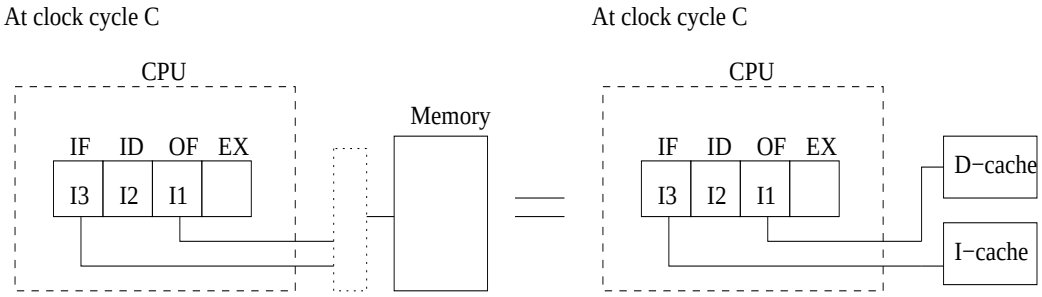


Figure 20: Structural hazard

It avoids simultaneous access to the same resource (memory).

0.5 Data Hazards

A pipeline stage of instruction I_j may require data of former instruction I_i at time t.

Former instruction I_i has not produced it at t - the data hazards may occur.

ADD R1, R2, R3
MULT R4, R1, R5
OR R6, R1, R7

Instructions MULT and OR, after the ADD, use the result generated by ADD.

1. Execution in 4-stage pipeline (Figure 21).

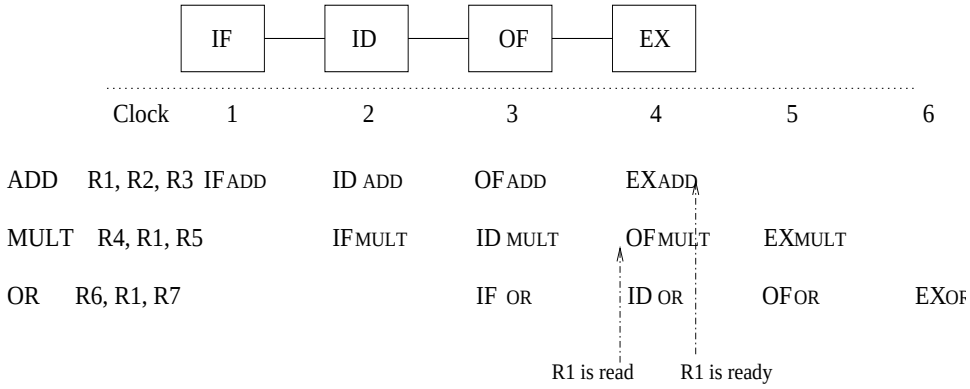


Figure 21: Data hazard in 4-stage instruction pipe

2. Execution in 5-stage pipeline (Figure 22).

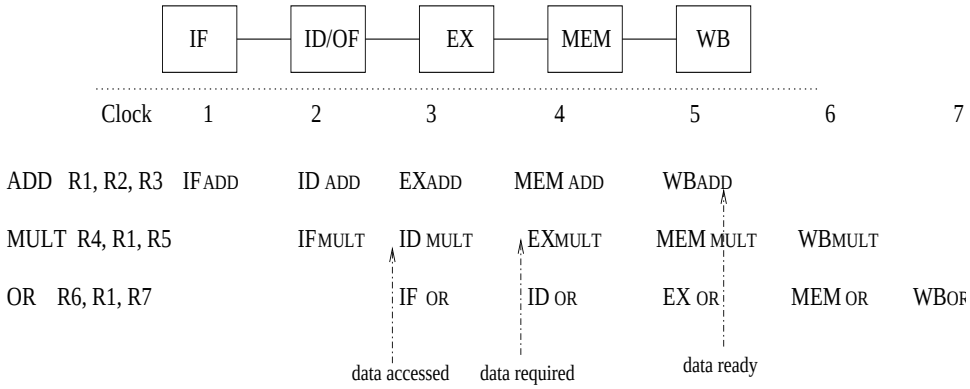
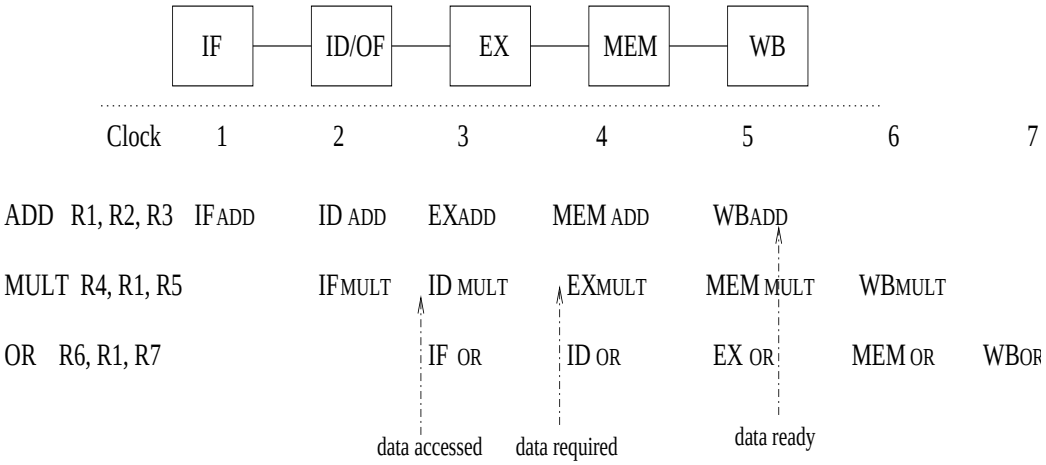


Figure 22: Data hazard in 5-stage instruction pipe



In Figure 22, operand fetching for MULT instruction is to be done at clock 3.

However, operand R1 is updated not before clock 5 (WB stage).

That is, MULT operation assumes stale value for operand R1.

This is *data hazard*.

OR operation gets affected by similar hazard.

R1/R7 are to be fetched at clock 4 but desired value of R1 is available after clock 5.

Data forwarding is considered to resolve data hazards in a pipeline.

0.5.1 Data forwarding

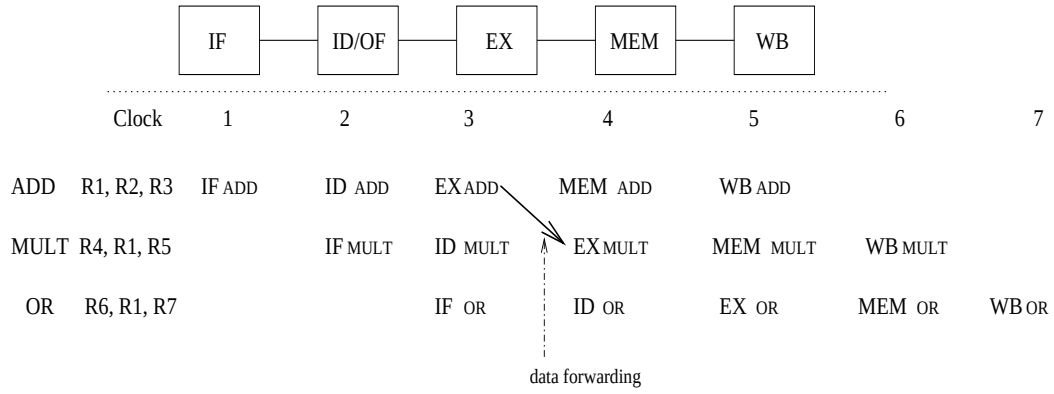


Figure 23: Data forwarding in 5-stage instruction pipe to avoid data hazard

As shown in Figure 23, MULT needs R1 at clock 4 and ADD produces it at clock 3. So, result of ADD -that is, R1 is moved the place where it is needed for MULT.

The *forwarding* works as follows:

A. ALU results on ALU_{output} register is fed back to ALU inputs (Figure 24). That is, before writing to register, output of ALU is sent to the input of ALU.

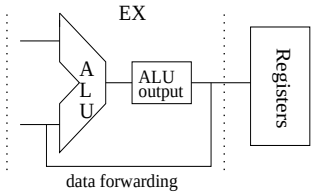


Figure 24: Data forwarding hardware

B. If forwarding hardware detects that a previous ALU operation has written register corresponding to a source for the current ALU operation, control logic selects the forwarded result as ALU input.

Generalized form of data forwarding is called *internal forwarding*

Internal forwarding

It replaces unnecessary memory access. A data that is already in register of a processor is directly sent to operand registers assigned for the instructions.

read-read forwarding: Consider following sequence of two instructions that read data from memory location M1.

$$R1 \leftarrow (M1)$$
$$R2 \leftarrow (M1)$$

In internal forwarding, these two instructions can be replaced by

$$R1 \leftarrow (M1)$$
$$R2 \leftarrow R1$$

It saves one memory access.

write-read forwarding: Consider following sequence of two instructions. First instruction writes to memory location M1 and second one reads data from M1.

$$\begin{aligned} (M1) &\leftarrow R1 \\ R2 &\leftarrow (M1) \end{aligned}$$

In internal forwarding, these two instructions can be replaced by

$$\begin{aligned} (M1) &\leftarrow R1 \\ R2 &\leftarrow R1. \end{aligned}$$

write-write forwarding: Consider following sequence of two instructions. Each of these writes content of a register to same memory location M1.

$$(M1) \leftarrow R1$$
$$(M1) \leftarrow R2$$

In internal forwarding, these two instructions can be replaced by a single instruction

$$(M1) \leftarrow R2.$$

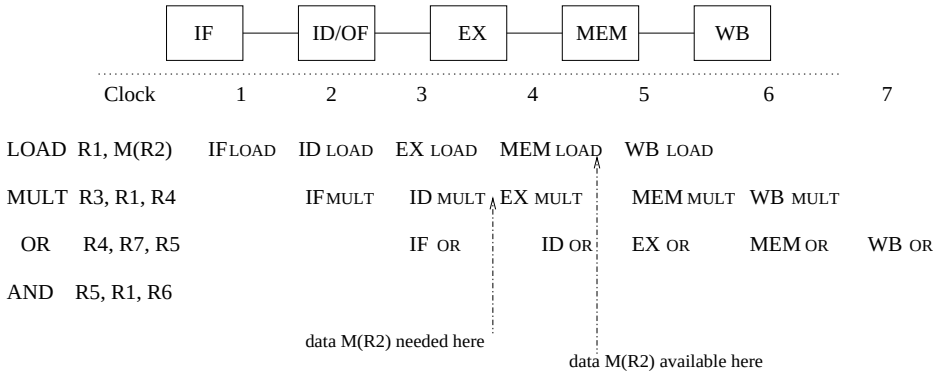
0.5.2 Stalls

Data forwarding can not solve all data hazard problems (Figure 25(a)).

It is effective if execution of I_i (preceeds I_j) is completed before execution of I_j.

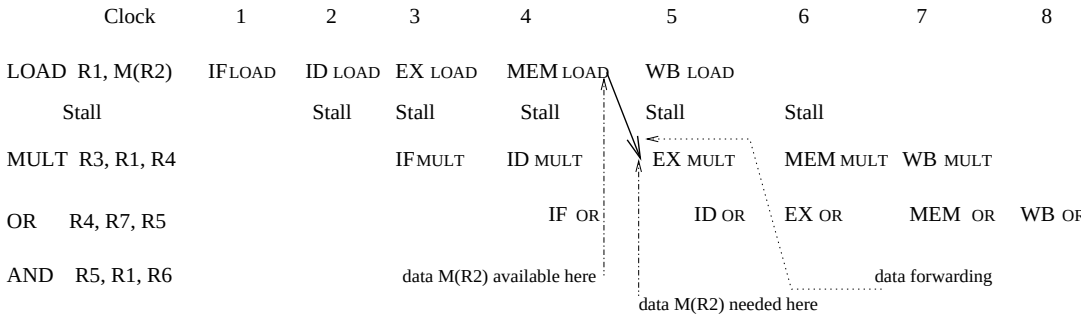
Consider execution of following program segment

LOAD R1, M(R2)
MULT R3, R1, R4
OR R4, R7, R5
AND R6, R1, R7



Forwarding of R1 that is M(R2) can't be done

(a)



Insertion of stall cycle with forwarding of R1 that is M(R2) avoids hazard

(b)

Figure 25: Data forwarding with stalls in 5-stage pipe to avoid data hazard

To resolve this, we introduce stalls (Figure 25(b)). Compiler inserts stalls.

0.5.3 Instruction reordering

There are instructions (say, I_k to I_{k+r}) that may not be dependent on output of I_i.

So, I_k to I_{k+r} instructions can be reordered to avoid data hazards.

Reordering of instructions is done during compilation time.

Let consider following program segment

I_i: LOAD R1, M(R2)
I_j: MULT R3, R1, R4
I_k: OR R4, R7, R5

Figure 26(a) shows execution with data forwarding and stall.

Figure 26(b) describes execution after instruction reordering. It avoids stall.

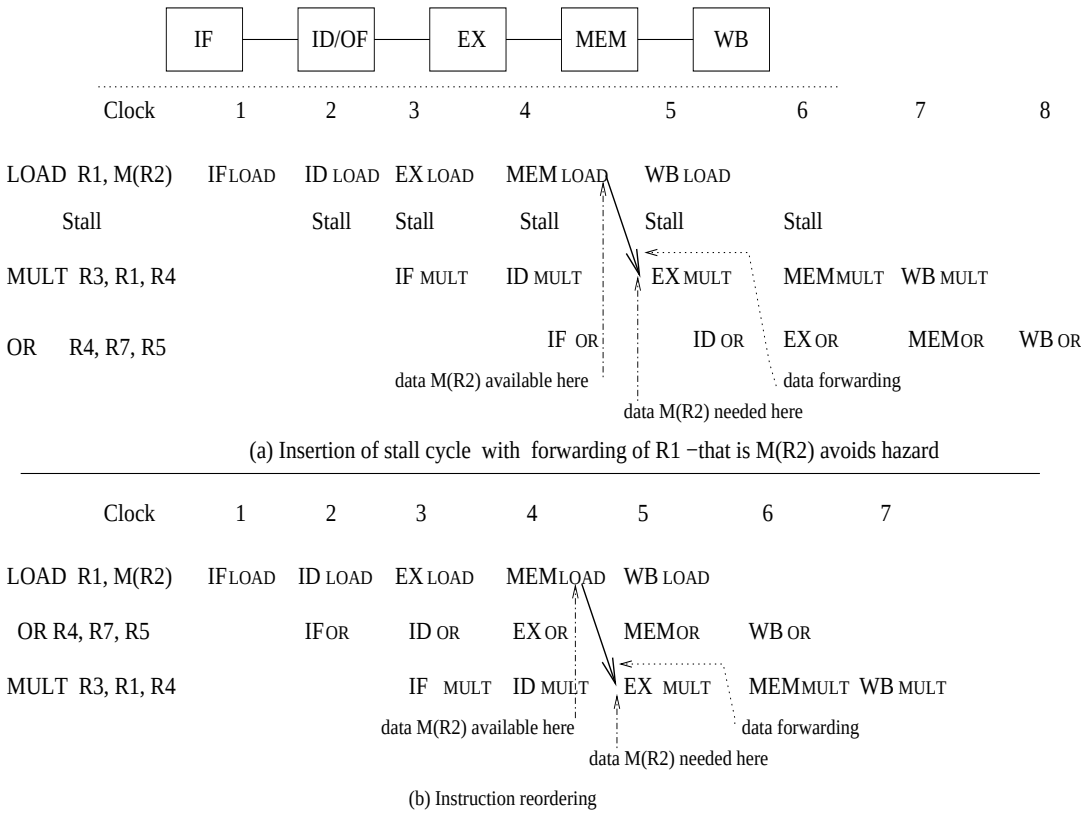


Figure 26: Instruction reordering to avoid data hazard

It can be possible as output set R(I_i) and input set D(I_k) are mutually exclusive.

Introduction of stall cycles, and instruction reordering are software solutions.
Data forwarding is a hardware solution and requires some additional hardware.
As forwarding hardware is placed on the pipeline datapath, it introduces a delay Δt.
Stall cycles or bubbles can be represented by a compiler as *nop* (no operation).
Instruction reordering is one-time investment of compilation time - serves better.
Data hazards occur due to data dependence - explained next.

0.5.4 Generic data hazards

The data hazards occur due to data dependence.

Three categories of dependence can affect pipeline performance.

RAW (read after write)
WAR (write after read)
WAW (write after write)

Let I_j follows I_i in the program.

First operation in RAW/WAR/WAW is associated with I_j.

Second operation is defined for I_i.

Example: In RAW, 'R' (read) corresponds to I_j and 'W' (write) corresponds to I_i.

1. **Read after write (RAW)**

It is also called *real* dependency (Figure 27).

It implies - programmer's intention is I_j will read an operand v after I_i writes to v.

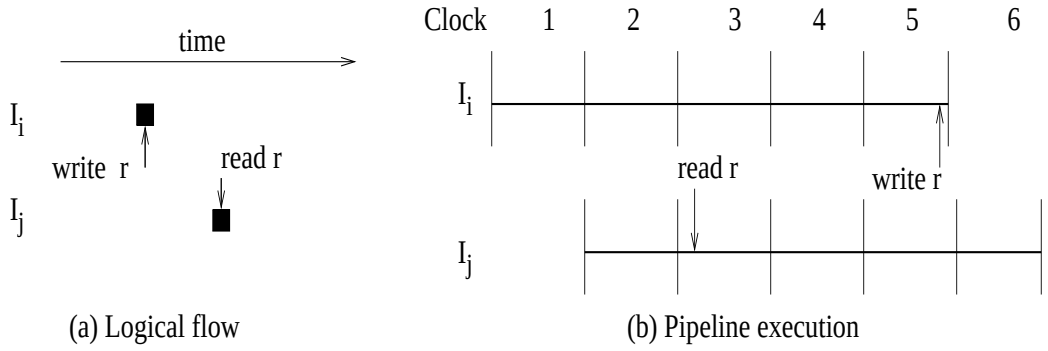


Figure 27: RAW

RAW: Let consider execution of following program segment.

I_i: R1 ← R2 + R3
I_j: R4 ← R1 * R5
I_k: R6 ← R1 or R7

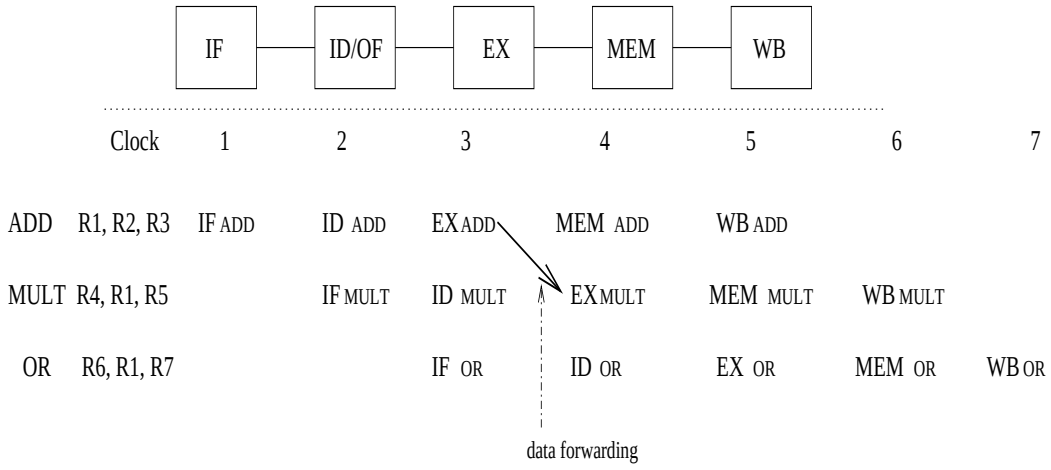


Figure 28: Data forwarding in 5-stage instruction pipe to avoid data hazard

Dependence between I_i and I_j (also I_i and I_k) on R1 is of type RAW. Here

$$R(I_i) \cap D(I_j) \neq \phi \text{ (also } R(I_i) \cap D(I_k) \neq \phi).$$

R(I) is the range (output set of instruction I) and D(I) is the domain (input set) of I.

2. **Write After Read (WAR)**

Also referred to as *antidependency* (Figure 29).

i) Instruction I_j logically follows instruction I_i,

ii) Logically, I_j writes to an operand after I_i reads it (Figure 29(a)), and

ii) In pipeline execution, I_j tries to write operand before I_i reads (Figure 29(b)).

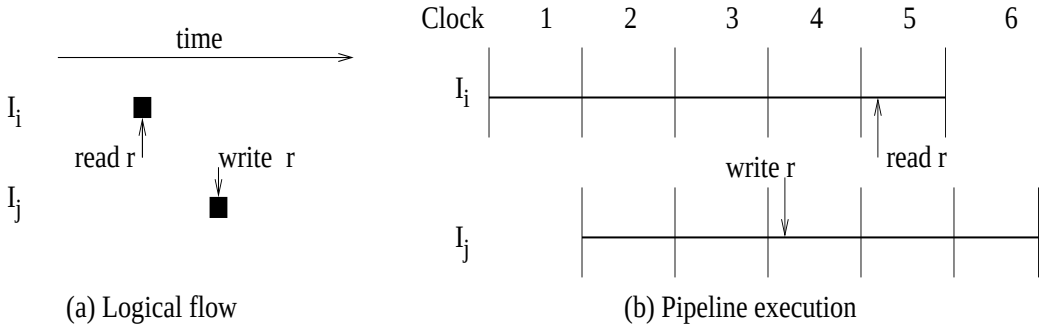


Figure 29: WAR

Let take following instruction sequence

$$\begin{aligned} I_i&: \quad R3 \leftarrow R1 + R2 \\ I_j&: \quad R2 \leftarrow R1 + R3 \end{aligned}$$

There is WAR dependence between I_i and I_j on R2. Here

$$D(I_i) \cap R(I_j) \neq \phi.$$

$R(I_j)$ is the range (output set of I_j) and $D(I_i)$ is the domain (input set) of I_i.

WAR cannot be an issue for pipeline of Figure 30, Figure 31 and Figure 32.

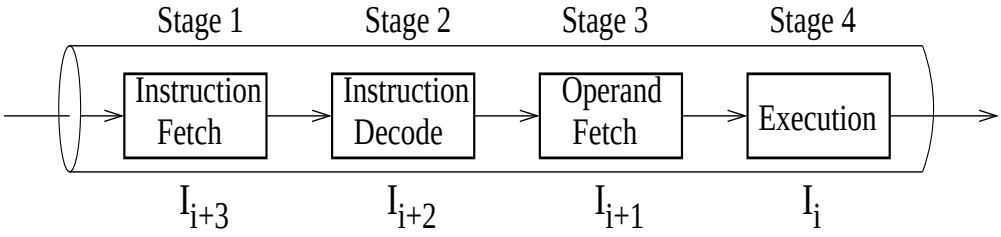


Figure 30: 4-stage instruction pipeline

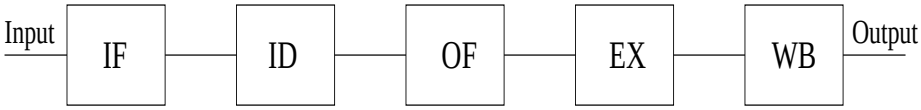


Figure 31: Instruction pipeline

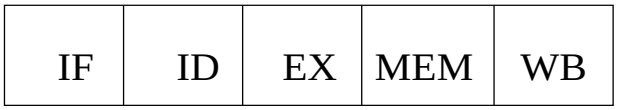


Figure 32: 5-stage DLX-like instruction pipeline

WAR hazard represents a problem with concurrent execution.

3. **Write After Write (WAW)**

Is also called output-dependency. This class of dependency is also artificial.

i) Logically, I_j writes to an operand after I_i writes to it (Figure 29(a)), and

ii) In pipeline execution, I_j tries to write operand before I_i writes (Figure 29(b)).

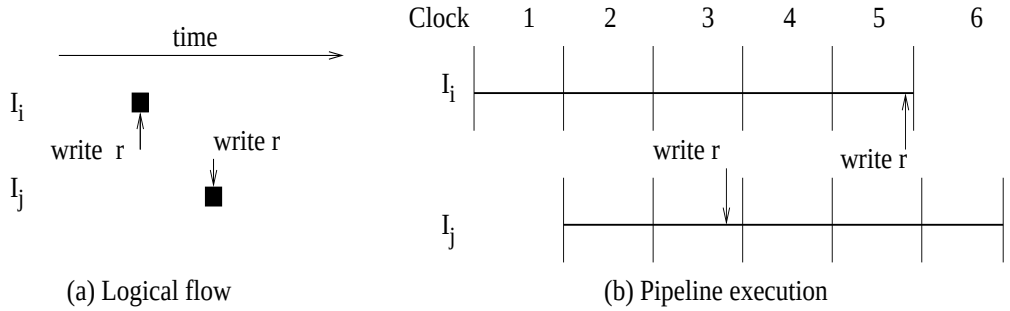


Figure 33: WAW

WAW data hazard is due to

$$R(I_i) \cap R(I_j) \neq \phi.$$

R(I_i) is the range (output set) of I_i and R(I_j) is the range of I_j.

Let take following program segment

I_i: R2 $\leftarrow$ R1 + R2
I_j: R2 $\leftarrow$ R3 + R4

Logical execution demands - both I_i and I_j write to R2.

Final desired value for R2 is written by I_j.

To avoid WAW data hazard, Write Back of I_j is to be delayed till completion of I_i.

4. **Read After Read (RAR)** This is known as input dependence. It is not a true hazard since register value does not change. Multiple reads of same operand can be performed in any order without affecting the domain of an instruction.

Problems

1. Consider the following program segment

$$
\begin{array}{l}
\vdots \\
\text{LOAD} \quad \text{R1, MEM1} \\
\text{LOAD} \quad \text{R2, MEM2} \\
\text{MULT} \quad \text{R1, R2} \\
\text{ADD} \quad \text{R1, R2} \\
\vdots
\end{array}
$$

   Prior to first LOAD instruction, contents of R1 = 0F, R2 = 1F, MEM1 = 08 and MEM2 = 21.

   a) Show operation of the pipeline of Figure 34 for the above sequence.
   b) During operand fetching for MULT (at cycle 6) what is the content of R2?
   c) Is there any possibility of data hazard at cycle 7?

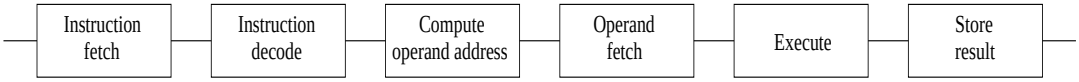


Figure 34: 6-stage instruction pipeline

2. Following assembly code is to be executed in a 3-stage pipelined processor with hazard detection and resolution in each stage. Pipeline stages are shown in Figure 35. Find possible structural and data hazards for execution of the code

$$
\begin{array}{lcl}
\text{INC R0} & \Rightarrow & \text{R0} \leftarrow \text{R0} + 1 \\
\text{MUL AC, R0} & \Rightarrow & \text{AC} \leftarrow \text{AC*R0} \\
\text{STORE X, AC} & \Rightarrow & \text{M(X)} \leftarrow \text{AC} \\
\text{ADD AC, R0} & \Rightarrow & \text{AC} \leftarrow \text{AC} + \text{R0} \\
\text{STORE Y, AC} & \Rightarrow & \text{M(Y)} \leftarrow \text{AC}
\end{array}
$$

   M(X) defines the content of memory location X.

Figure 35: 3-stage instruction pipeline

3. Let consider execution of the following code segment in a 5-stage (IF, ID/OF, EX, MEM, and WB) instruction pipeline.

LOAD	R1, M(R2)
MULT	R3, R1, R4
OR	R4, R1, R5
AND	R6, R1, R7

a) Clearly indicate the data dependencies and their type.
b) If data forwarding is not implemented in this pipelined processor, then indicate data hazards and show *stalls* introduced to eliminate those hazards.
c) Indicate hazards and show *stalls* introduced to eliminate those hazards when data forwarding is implemented.
d) What speed-up is achieved in data forwarding in comparison to without data forwarding?

4. Consider execution of following code segment in a 5-stage (IF, ID/OF, EX, MEM, and WB) instruction pipeline.

LW	R_a, $address_a$
LW	R_b, $address_b$
ADD	R_c, R_a, R_b
SW	$address_c$, R_c
LW	R_d, $address_d$
LW	R_e, $address_e$
SUB	R_f, R_d, R_e
SW	$address_f$, R_f

Assume load/store have latency of one clock cycle. Reorder the code sequence to avoid data hazard and stalls.