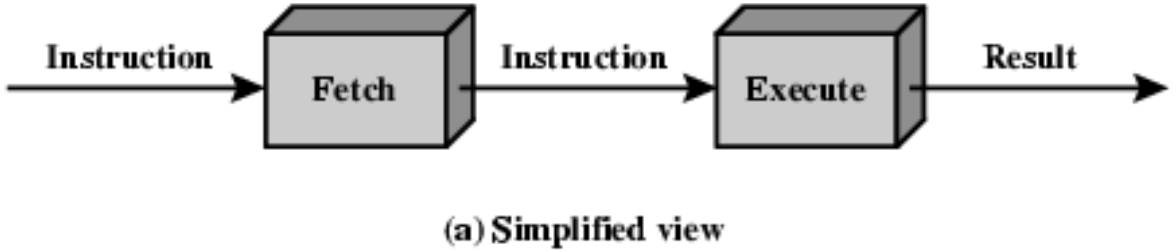
Lecture-7

Chapter-14.4

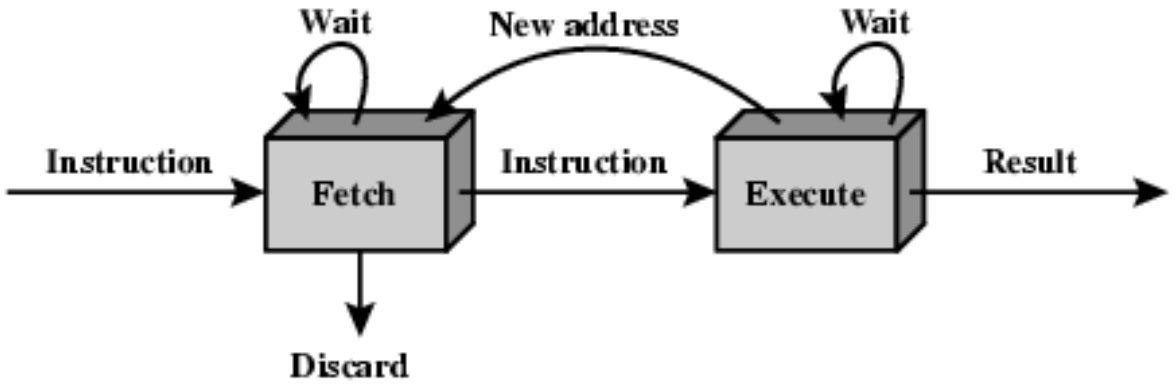
Computer Organization and Architecture Designing - William Stallings

Instruction Pipelining

Two Stage Instruction Pipeline



(a) Simplified view

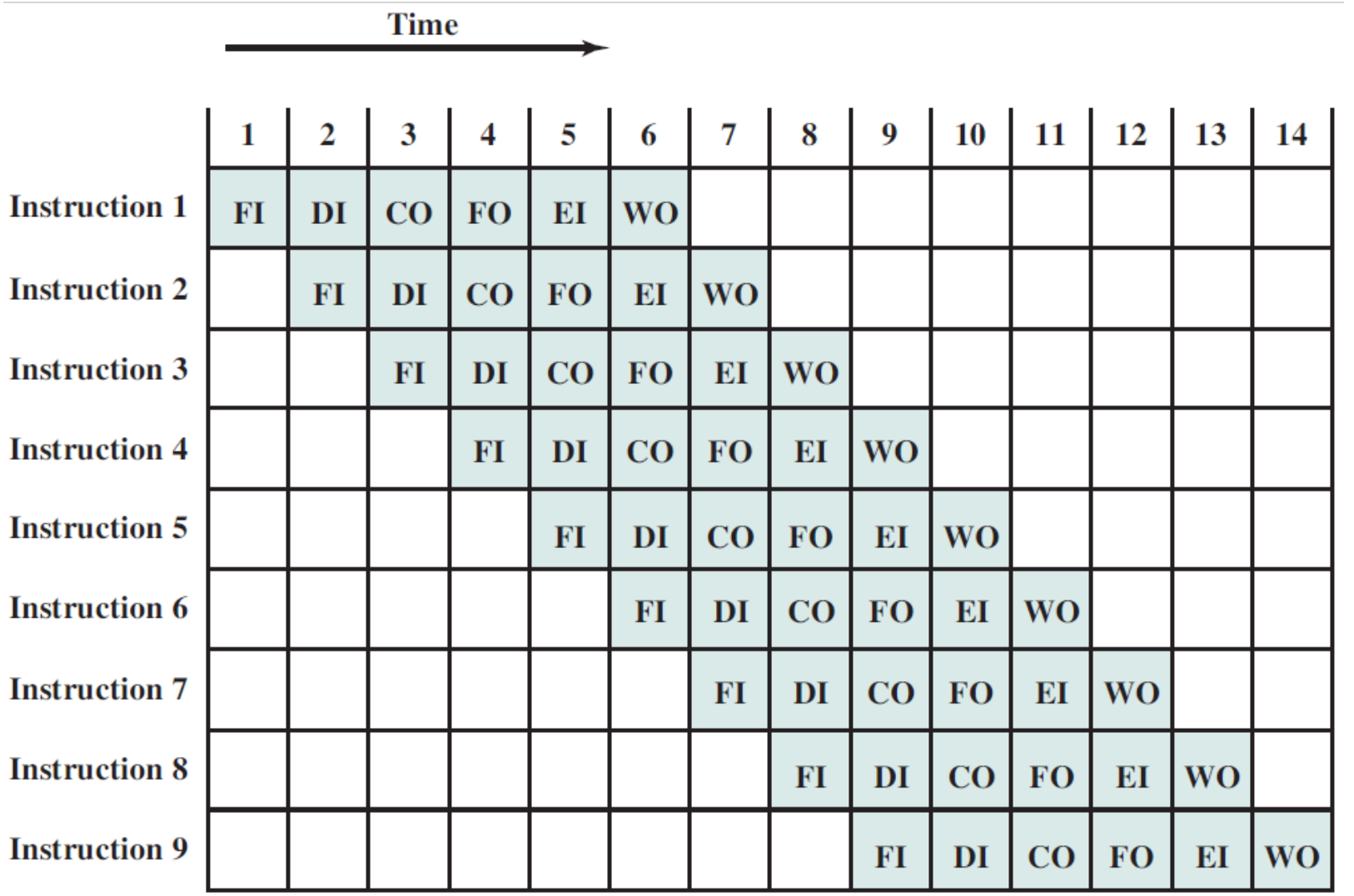


(b) Expanded view

Stages of Pipelining

- **Fetch instruction (FI):** Read the next expected instruction into a buffer.
- **Decode instruction (DI):** Determine the opcode and the operand specifiers.
- **Calculate operands (CO):** Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.
- **Fetch operands (FO):** Fetch each operand from memory. Operands in registers need not be fetched.
- **Execute instruction (EI):** Perform the indicated operation and store the result, if any, in the specified destination operand location.
- **Write operand (WO):** Store the result in memory.

Timing Diagram for Instruction Pipeline Operation



- Fetch Instruction (FI)
- Decode Instruction (DI)
- Calculate Operands (CO)
- Fetch Operands (FO)
- Execute Instructions (EI)
- Write Operands (WO)

Figure- 1.1

Timing Diagram for Instruction Pipeline Operation

- In figure 1.1 shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.
- Several comments are in order: The diagram assumes that each instruction goes through all six stages of the pipeline. This will not always be the case.
- For example, a load instruction does not need the WO stage.
- However, to simplify the pipeline hardware, the timing is set up assuming that each instruction requires all six stages.
- Also, the diagram assumes that all of the stages can be performed in parallel.
- In particular, it is assumed that there are no memory conflicts.
- For example, the FI, FO, and WO stages involve a memory access. The diagram implies that all these accesses can occur simultaneously.

The Effect of a Conditional Branch on Instruction Pipeline Operation

- Figure 1.2 illustrates the effects of the conditional branch, using the same program as previous figure.
- Assume that instruction 3 is a conditional branch to instruction 15.
- Until the instruction is executed, there is no way of knowing which instruction will come next.
- The pipeline, in this example, simply loads the next instruction in sequence (instruction 4) and proceeds.
- The branch is not determined until the end of time unit 7.
- At this point, the pipeline must be cleared of instructions that are not useful.
- During time unit 8, instruction 15 enters the pipeline.
- No instructions complete during **time units 9 through 12**; this is the performance penalty incurred because we could not anticipate the branch.

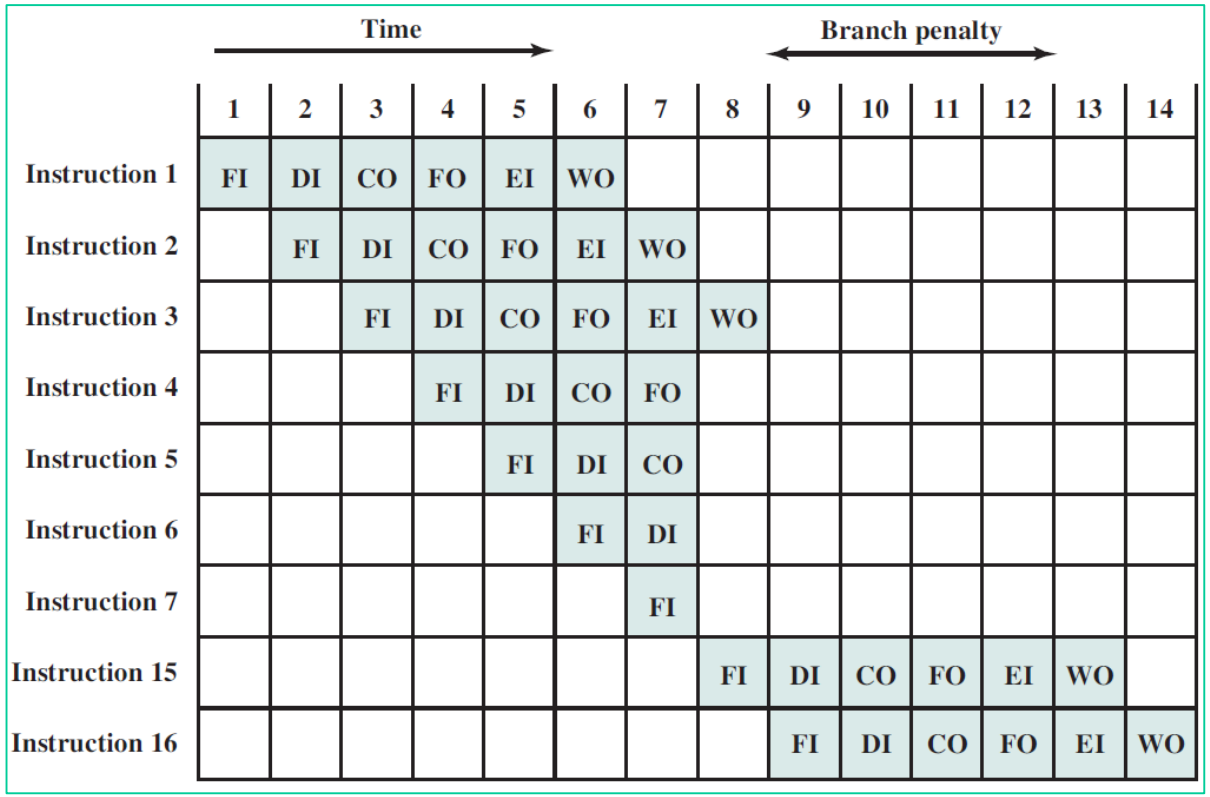
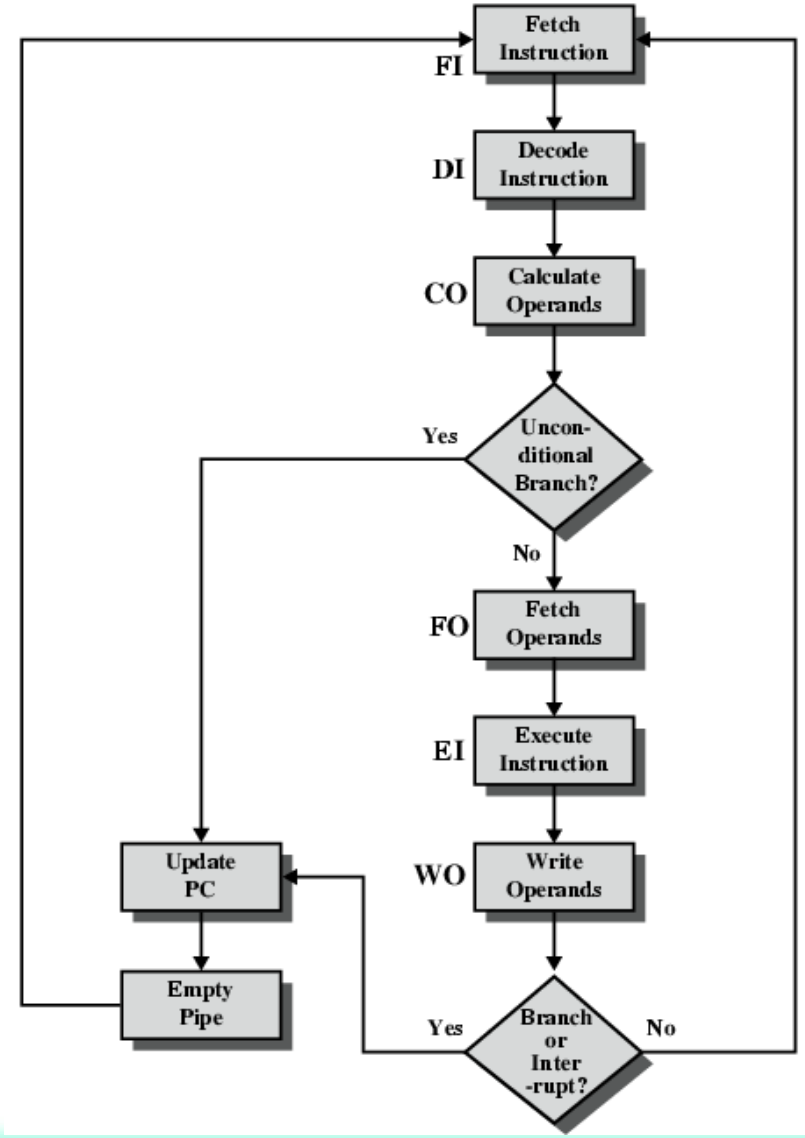


Figure- 1.2

Six Stage Instruction Pipeline



Alternative Pipeline Depiction

- Fetch Instruction (FI)
- Decode Instruction (DI)
- Calculate Operands (CO)
- Fetch Operands (FO)
- Execute Instructions (EI)
- Write Operands (WO)

Figure 1.3 shows same sequence of events, with time pr ogressing vertically down the figure, and each row show ing the state of the pipeline at a given point in time.

- In Figure **1.3a** (which corresponds to **Figure 1.1**), the pipeline is full at time 6, with 6 different instructions in various stages of execution, and remains full through time 9; we assume that instruction I9 is the last instruction to be executed.
- In Figure **1.3b,** (which corresponds to **Figure 1.2**), the pipeline is full at times 6 and 7.
- At time 7, instruction 3 is in the execute stage and executes a branch to instruction 15.
- At this point, instructions I4 through I7 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15

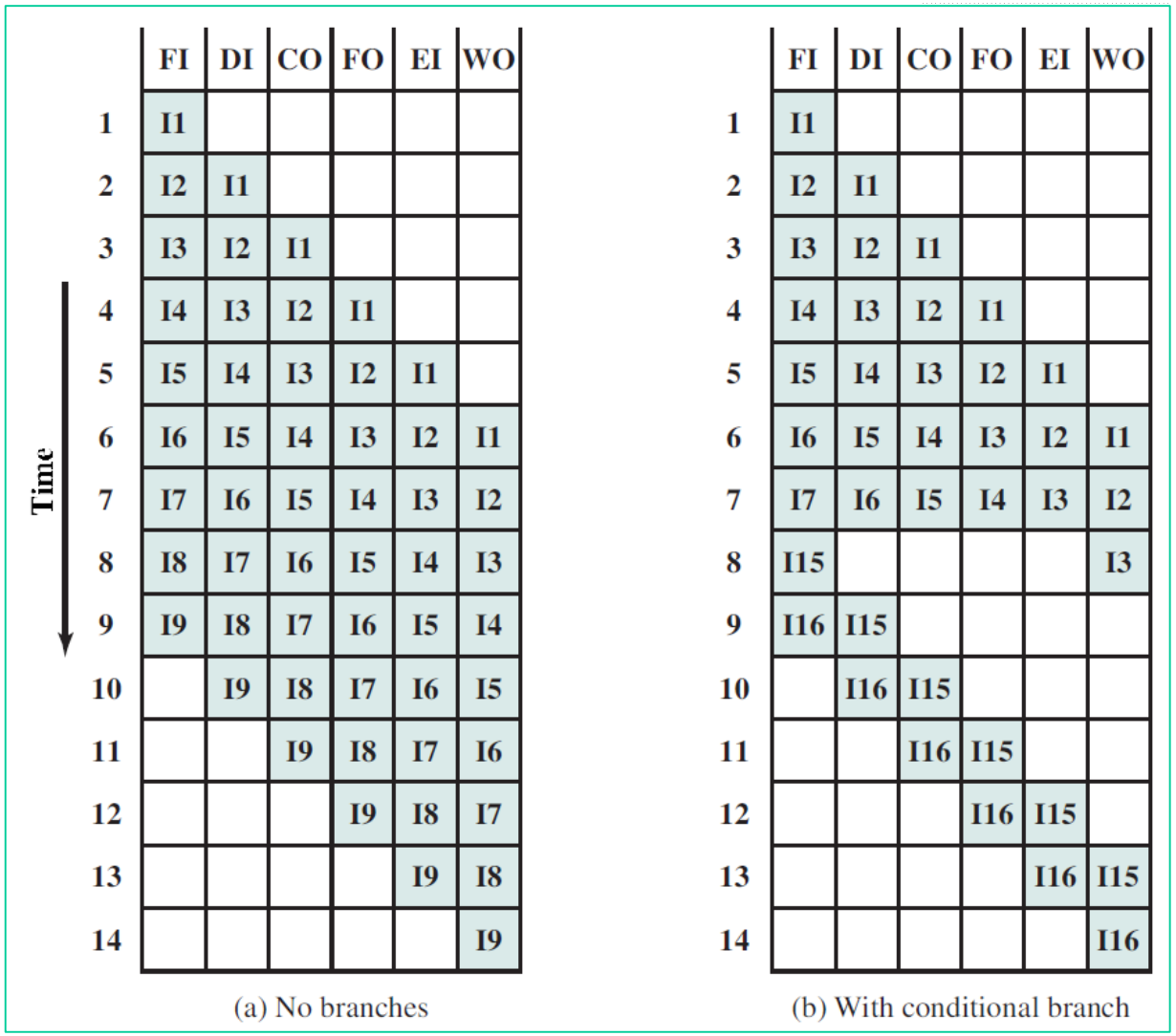


Figure- 1.3

Pipeline Performance

- The cycle time of an instruction pipeline is the time needed to advance a set of instructions one stage through the pipeline
- The cycle time can be determined as

$$\tau = \max_i[\tau_i] + d = \tau_m + d \quad 1 \leq i \leq k$$

where

τ_i = time delay of the circuitry in the ith stage of the pipeline

τ_m = maximum stage delay (delay through stage which experiences the largest delay)

k = number of stages in the instruction pipeline

d = time delay of a latch, needed to advance signals and data from one stage to the next

Pipeline Performance

- Now suppose that n *instructions* are processed, with no branches. Let $T_{k,n}$ be the total time required for a pipeline with ***k stages*** to execute ***n instructions***. Then

$$T_{k,n} = [k + (n - 1)]\tau$$

[A total of k cycles are required to complete the execution of the first instruction, and the remaining n – 1 instructions require n – 1 cycle. Therefore; The ninth instruction completes at time cycle 14:14 = [6 + (9 - 1)]]

- Now consider a processor with equivalent functions but no pipeline, and assume that the instruction cycle time is K_t. The **speedup factor** for the instruction pipeline compared to execution without the pipeline is defined as:

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)}$$

$$= 9*6 / 6+(9-1)$$

$$= 3.85$$

Pipeline Hazards

Pipeline hazards refer to situations where the pipeline is unable to maintain its **normal flow of instructions** due to dependencies or conflicts between instructions.

There are **three types** of pipeline hazards:

1. **Resource/Structural Hazards**: These occur when two instructions require the same hardware resource at the same time. For example, if one instruction needs to write to a register while another instruction is reading from the same register, a structural hazard occurs.
2. **Data Hazards**: These occur when an instruction depends on the result of a previous instruction that has not yet been completed. For example, if one instruction performs a calculation and the next instruction requires the result of that calculation as an input, a data hazard occurs.
3. **Control Hazards**: These occur when the pipeline encounters a branch instruction (e.g., if-else or loop), and it is not clear which instruction to fetch next until the branch is resolved. This causes a delay in the pipeline.

Resource Hazards

There are several common causes of structural hazards:

- **Resource conflicts**: When two or more instructions require the same hardware resource at the same time, a structural hazard occurs. ***For example***, if two instructions need to write to the same register at the same time, the pipeline cannot execute them simultaneously.
- **Limited resources**: If the processor has a limited number of functional units or registers, it may not be possible to satisfy all instruction requests at the same time.
- **Resource fragmentation**: Sometimes the pipeline has a sufficient number of resources, but they are not organized optimally to accommodate certain types of instructions.

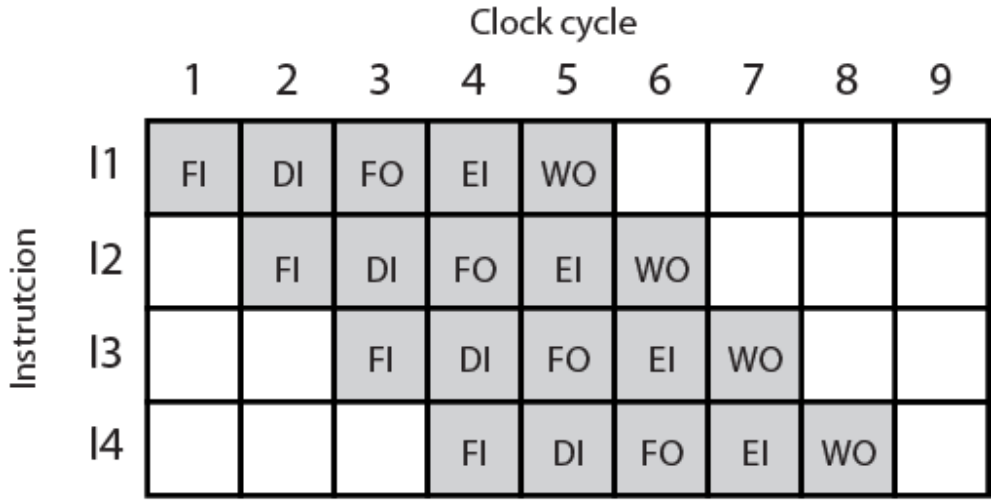
One solution: increase available resources

  – Multiple main memory ports, Multiple ALUs

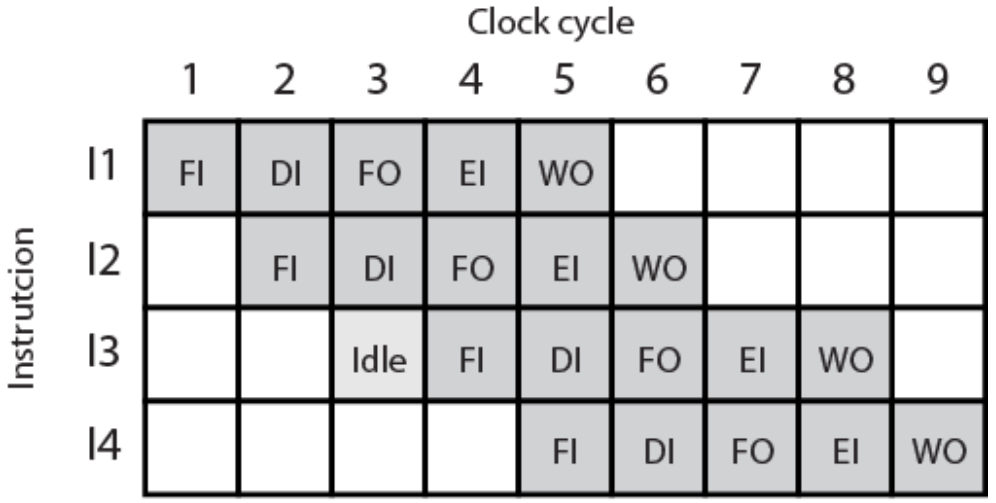
Other solutions can be: Resource Sharing, Instruction Scheduling, Compiler Optimization,

By using a combination of these solutions, processors can minimize the impact of structural hazards and achi eve higher performance and efficiency

Resource Hazard Diagram



(a) Five-stage pipeline, ideal case



(b) I1 source operand in memory

Data Hazards

Data hazards occur when an instruction depends on the results of a previous instruction that has not yet been completed. These dependencies arise when an instruction tries to read from or write to a register or memory location that is still being used by a previous instruction.

Example:

- **ADD** EAX, EBX /* EAX = EAX + EBX
- **SUB** ECX, EAX /* ECX = ECX – EAX

Here: **ADD instruction** does not update EAX until end of stage 5, at clock cycle 5. **SUB instruction** needs value at beginning of its stage 2, at clock cycle 4. Pipeline must stall for two clocks cycles.

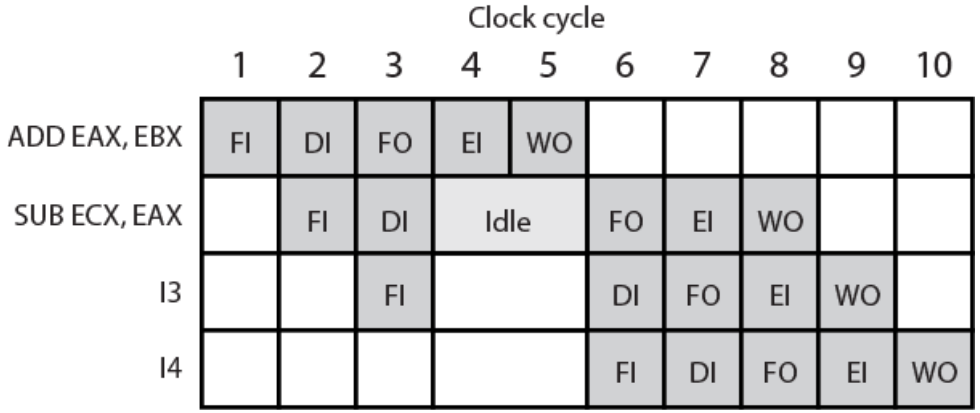


Fig: Data Hazard Diagram

Types of Data Hazard

There are **three types** of data hazards:

- **RAW (Read After Write) Hazard**: Causes when an instruction tries to read a value before it is written by a prior instruction.
  - **Example**:
```
ADD EAX, EBX  ; Writes EAX in stage 5 (Execute).
SUB ECX, EAX  ; Reads EAX in stage 2 (Decode).
```
  - **Result:** The SUB instruction reads stale data.
  - **Solution**: **Stalling**: Insert 2 idle cycles (as shown in Figure 14.16).

- **WAR (Write After Read) Hazard**: Causes when an instruction writes to a register before a prior in struction reads it.
  - **Example**:
```
LOAD R1, [MEM]  ; Reads R1 in stage 2 (Decode).
ADD  R1, R2, R3 ; Writes R1 in stage 5 (Execute).
```
  - **Result:** The LOAD reads an incorrect value if ADD writes first.
  - **Solution**: **Register Renaming**: Use temporary registers to isolate dependencies.

Types of Data Hazard

There are **three types** of data hazards:

- **WAW (Write After Write) Hazard**: Causes when two instructions write to the same register out of order.
  - **Example**:
```
MUL R1, R2, R3  ; Writes R1 in stage 5.
ADD R1, R4, R5  ; Writes R1 in stage 5.
```
  - **Result:** Incorrect final value in R1.
  - **Solution**: Reordering Instructions: Ensure writes occur in program order.

Control Hazard

Also known as *branch hazard* where Pipeline makes wrong decision on branch prediction and brings instructions into pipeline that must subsequently be discarded.

There are several techniques to mitigate control hazards in pipelined processors:

Multiple Streams: Multiple streams is a technique that involves maintaining multiple instruction streams in the pipeline, one for the predicted path and one for the alternative path. When a branch instruction is encountered, the pipeline can switch to the appropriate stream without incurring any pipeline stalls.

Prefetch Branch Target: Prefetch branch target is a technique that involves predicting the target address of a branch instruction and prefetching the instructions at that target address into the instruction cache. This can reduce the number of pipeline stalls that occur when a branch instruction is encountered.

also there are **Loop Buffer, Branch Prediction**, and **Delayed Branch**

Thank You