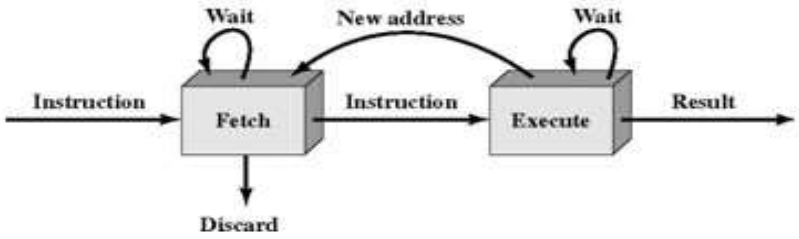
UNIT-3

❖ INSTRUCTION PIPELINING

As computer systems evolve, greater performance can be achieved by taking advantage of improvements in technology, such as faster circuitry, use of multiple registers rather than a single accumulator, and the use of a cache memory. Another organizational approach is instruction pipelining in which new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.

3.1(a) Simplified View



3.1(b) Expanded View

3.1 Two-Stage Instruction Pipeline

Figure 3.1a depicts this approach. The pipeline has two independent stages. The first stage fetches an instruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This is called *instruction prefetch* or *fetch overlap.*

This process will speed up instruction execution only if the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (Figure 3.1b), we will see that this doubling of execution rate is unlikely for 3 reasons:

1 The execution time will generally be longer than the fetch time. Thus, the fetch stage may have to wait for some time before it can empty its buffer.

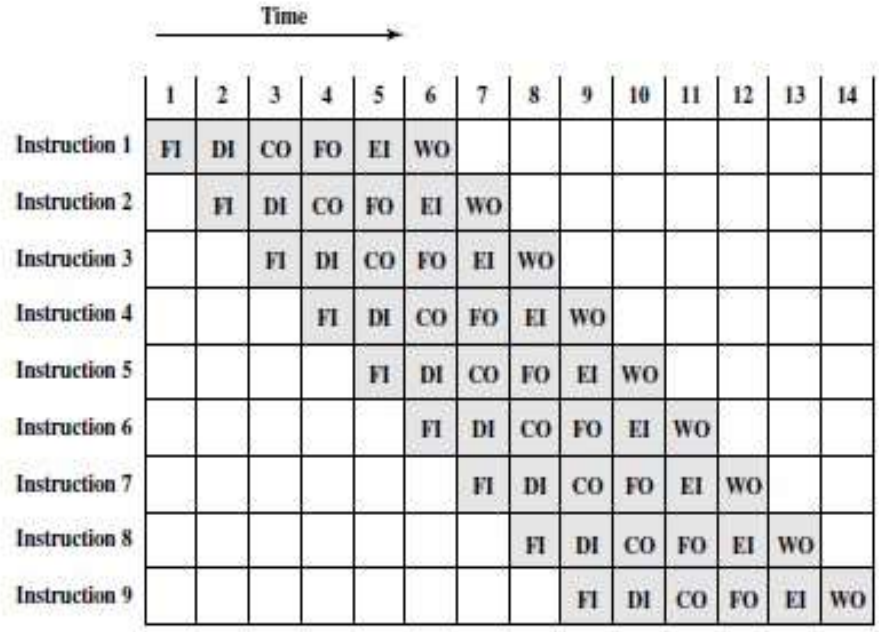
2 A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

3 When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost .If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

To gain further speedup, the pipeline must have more stages. Let us consider the following decomposition of the instruction processing.

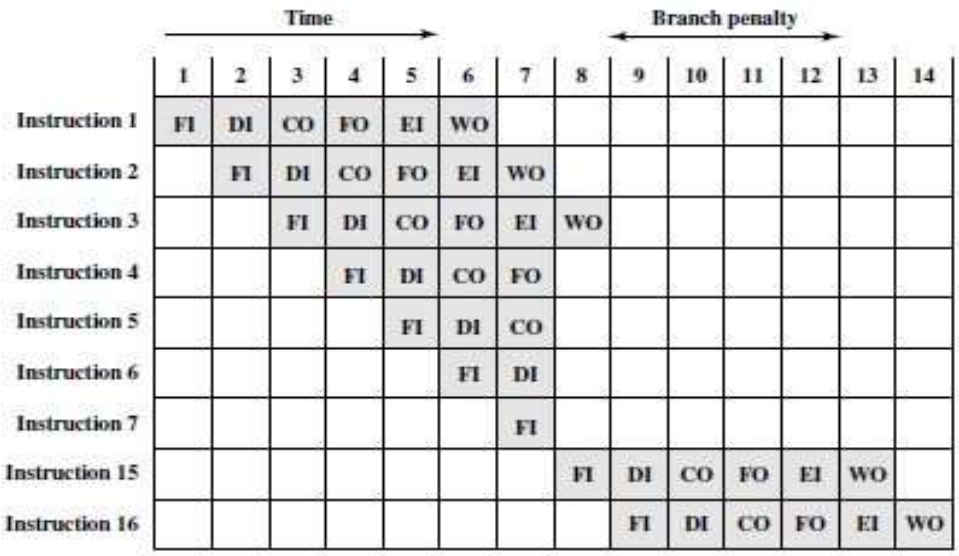
1. **Fetch instruction (FI):** Read the next expected instruction into a buffer.
2. **Decode instruction (DI):** Determine the opcode and the operand specifiers.
3. **Calculate operands (CO):** Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.
4. **Fetch operands (FO):** Fetch each operand from memory.
5. **Execute instruction (EI):** Perform the indicated operation and store the result, if any, in the specified destination operand location.
6. **Write operand (WO):** Store the result in memory.

Figure 3.2 shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.



3.2 Timing Diagram for Instruction Pipeline Operation

FO and WO stages involve a memory access. If the six stages are not of equal duration, there will be some waiting involved at various pipeline stages. Another difficulty is the conditional branch instruction, which can invalidate several instruction fetches. A similar unpredictable event is an interrupt.

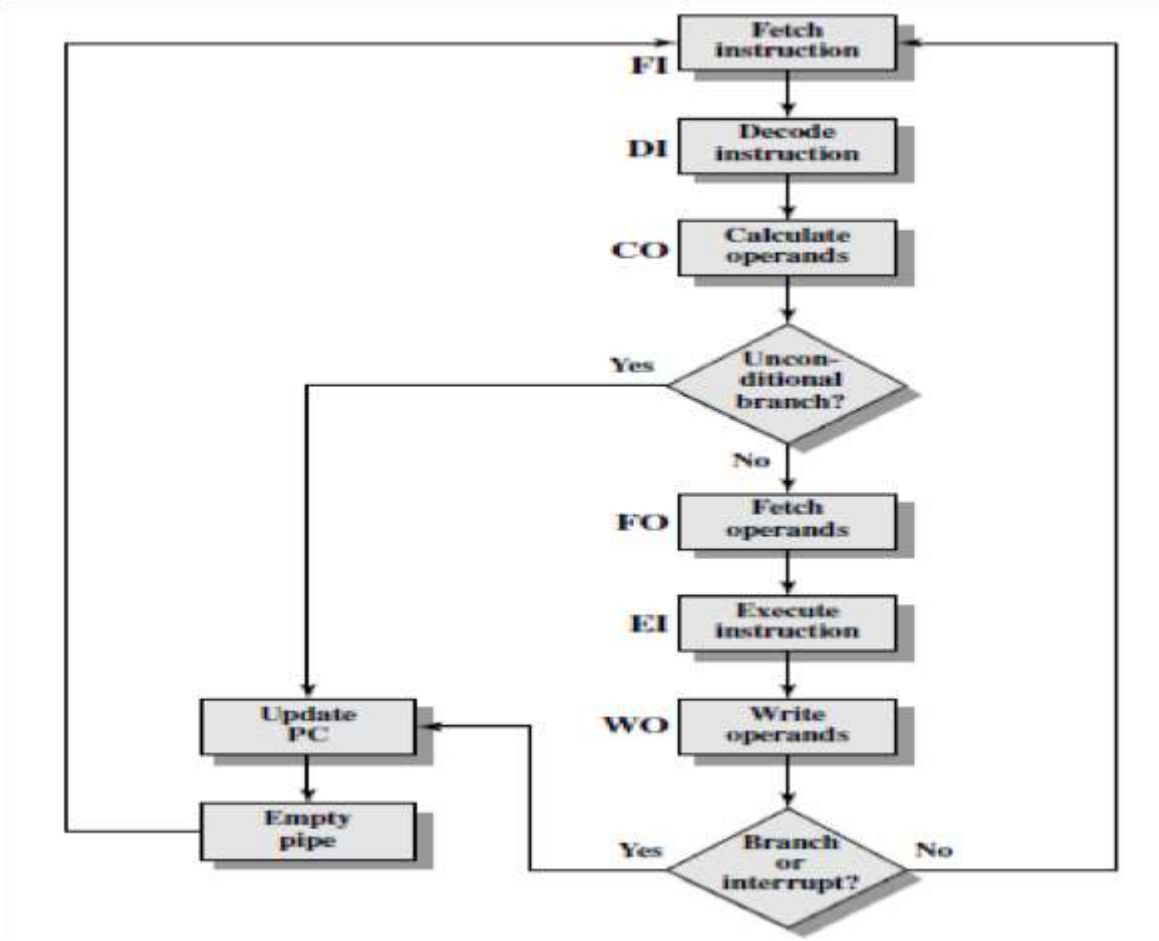


3.3 Timing Diagram for Instruction Pipeline Operation with interrupts

Figure 3.3 illustrates the effects of the conditional branch, using the same program as Figure 3.2. 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.

In Figure 3.2, the branch is not taken. In Figure 3.3, the branch is taken. This 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 3.4 indicates the logic needed for pipelining to account for branches and interrupts.



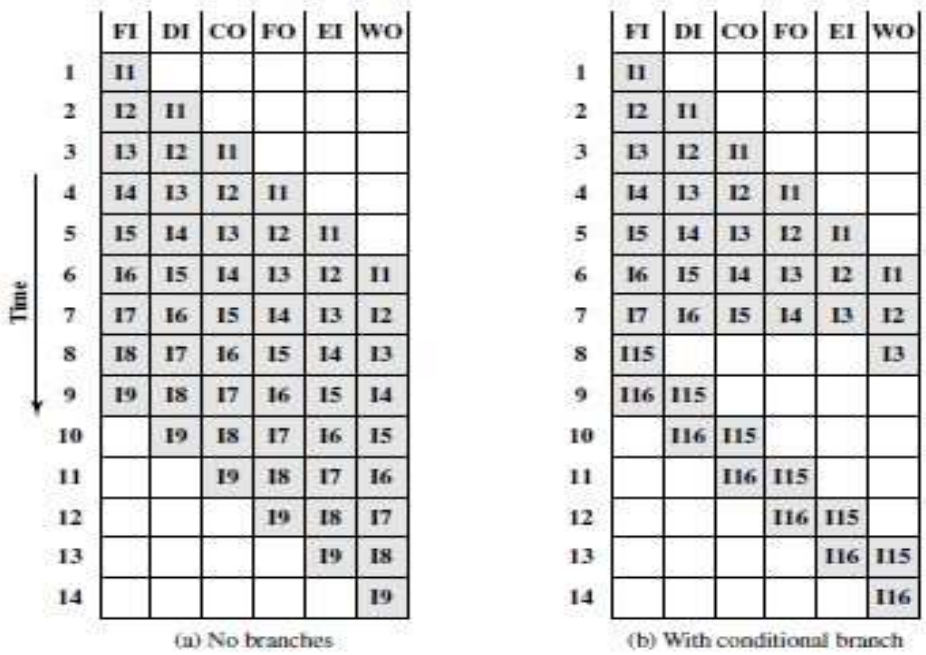
3.4 Six-stage CPU Instruction Pipeline

Figure 3.5 shows same sequence of events, with time progressing vertically down the figure, and each row showing the state of the pipeline at a given point in time. In Figure 3.5a (which corresponds to Figure 3.2), 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 3.5b, (which corresponds to Figure 3.3), 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. For high-performance in pipelining designer must still consider about :

1 At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions. This overhead can appreciably lengthen the total execution time of a single instruction.

2 The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages. This can lead to a situation where the logic controlling the gating between stages is more complex than the stages being controlled.

3 Latching delay: It takes time for pipeline buffers to operate and this adds to instruction cycle time.



3.5 An Alternative Pipeline depiction

Pipelining Performance

Measures of pipeline performance and relative speedup:

The cycle time t of an instruction pipeline is the time needed to advance a set of instructions one stage through the pipeline; each column in Figures 3.2 and 3.3 represents one cycle time. The cycle time can be determined as

$$t = \max [ti] + d = tm + d \quad 1 \ldots i \ldots k$$

where ti = time delay of the circuitry in the ith stage of the pipeline tm =

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

In general, the time delay d is equivalent to a clock pulse and tm W d. Now suppose that n instructions are processed, with no branches. Let *Tk,n* be the total time required for a pipeline with k stages to execute n instructions. Then

$$Tk,n = [k + (n -1)]t$$

A total of k cycles are required to complete the execution of the first instruction, and the remaining n

-1 instructions require *n* -1 cycles. This equation is easily verified from Figures 3.1. 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 kt. The speedup factor for the instruction pipeline compared to execution without the pipeline is defined as

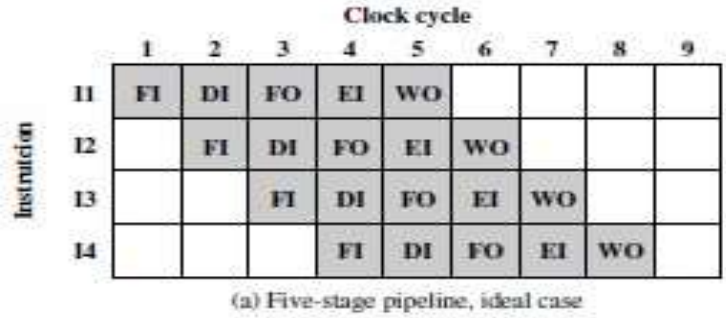
$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n-1)]\tau} = \frac{nk}{k + (n-1)}$$

❖ Pipeline Hazards

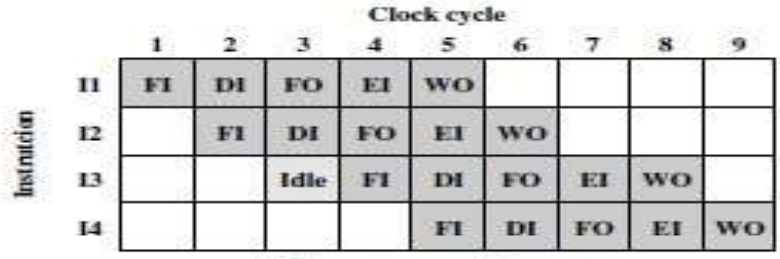
A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipeline stall is also referred to as a *pipeline bubble*. There are three types of hazards: resource, data, and control.

RESOURCE HAZARDS A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline. A resource hazard is sometime referred to as a *structural hazard*.

Let us consider a simple example of a resource hazard.Assume a simplified five-stage pipeline, in which each stage takes one clock cycle. In Figure 3.6a which a new instruction enters the pipeline each clock cycle. Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. In this case, an operand read to or write from memory cannot be performed in parallel with an instruction fetch. This is illustrated in Figure 3.6b, which assumes that the source operand for instruction I1 is in memory, rather than a register. Therefore, the fetch instruction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instruction I3. The figure assumes that all other operands are in registers.



(a) Five-stage pipeline, ideal case



(b) I1 source operand in memory

3.6 Example of Resource Hazard

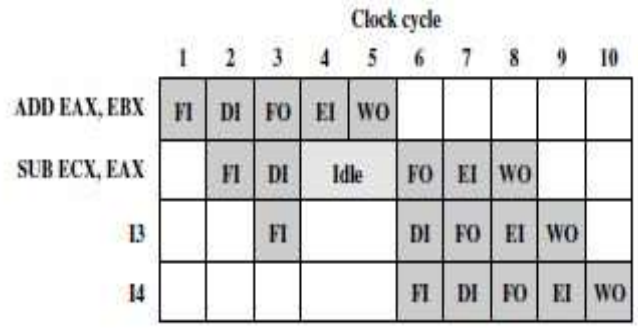
DATA HAZARDS A data hazard occurs when two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs but if the instructions are executed in a pipeline, then the operand value is to be updated in such a way as to produce a different result than would occur only with strict sequential execution of instructions. The program produces an incorrect result because of the use of pipelining.

As an example, consider the following x86 machine instruction sequence:

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

The first instruction adds the contents of the 32-bit registers EAX and EBX and stores the result in EAX. The second instruction subtracts the contents of EAX from ECX and stores the result in ECX.

Figure 3.7 shows the pipeline behaviour. The ADD instruction does not update register EAX until the end of stage 5, which occurs at clock cycle 5. But the SUB instruction needs that value at the beginning of its stage 2, which occurs at clock cycle 4. To maintain correct operation, the pipeline must stall for two clocks cycles. Thus, in the absence of special hardware and specific avoidance algorithms, such a data hazard results in inefficient pipeline usage. There are three types of data hazards;



3.7 Example of Resource Hazard

- **Read after write (RAW), or true dependency:**.A hazard occurs if the read takes place before the write operation is complete.
- **Write after read (RAW), or antidependency:** A hazard occurs if the write operation completes before the read operation takes place.
- **Write after write (RAW), or output dependency:** Two instructions both write to the same location. A hazard occurs if the write operations take place in the reverse order of the intended sequence. The example of Figure 3.7 is a RAW hazard.

CONTROL HAZARDS A control hazard, also known as a *branch hazard*, occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded.

❖ **<u>Dealing with Branches</u>**

Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not. A variety of approaches have been taken for dealing with conditional branches:

- Multiple streams
- Prefetch branch target
- Loop buffer
- Branch prediction
- Delayed branch

MULTIPLE STREAMS A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may make the wrong choice.

A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams. There are two problems with this approach:

- With multiple pipelines there are contention delays for access to the registers and to memory.
- Additional branch instructions may enter the pipeline (either stream) before the original branch decision is resolved. Each such instruction needs an additional stream.

Examples of machines with two or more pipeline streams are the IBM 370/168 and the IBM 3033.

PREFETCH BRANCH TARGET When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched.

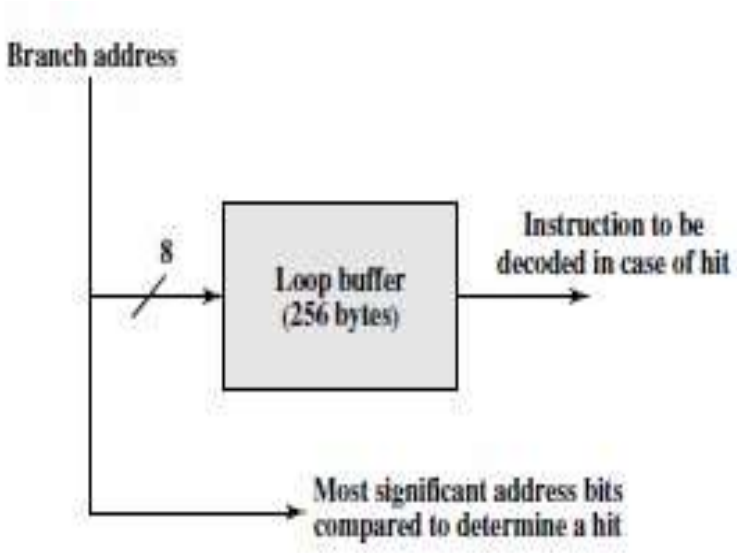
Example- The IBM 360/91 uses this approach.

LOOP BUFFER A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline and containing the n most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer.

The loop buffer has three benefits:

1. With the use of prefetching, the loop buffer will contain some instruction sequentially ahead of the current instruction fetch address.
2. If a branch occurs to a target just a few locations ahead of the address of the branch instruction,the target will already be in the buffer.
3. This strategy is particularly well suited to dealing with loops, or iterations; hence the name *loop buffer*. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.

Figure 3.8 gives an example of a loop buffer



3.8 Loop Buffer

BRANCH PREDICTION Various techniques can be used to predict whether a branch will be taken. Among the more common are the following:

- Predict never taken
- Predict always taken
- Predict by opcode
- Taken/not taken switch
- Branch history table

The first three approaches are static: they do not depend on the execution history up to the time of the conditional branch instruction. The latter two approaches are dynamic: They depend on the execution history.

The first two approaches are the simplest. These either always assume that the branch will not be taken or continue to fetch instructions in sequence, or they always assume that the branch will be taken and always fetch from the branch target. The predict-never-taken approach is the most popular of all the branch prediction methods.

DELAYED BRANCH It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired.