UNIT III

8. PIPELINING

8.1 Basic Concepts

Performance of the CPU can be enhanced basically by two methods:

1. Using the faster Hardware technology for processing
2. Arrange the hardware such that more than one operation is performed concurrently.

Pipelining is one of the effective ways to implement concurrency in the execution of the instruction.

8.1.1 Definition

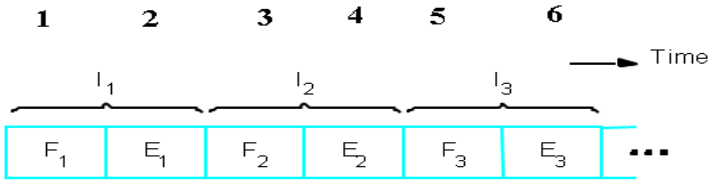
Overlapping the execution of the instruction such that different hardware units are used in a particular clock cycle.

We know that execution of any instruction has two different phases:

1. Fetch: fetching of the instruction performed by fetch unit.
2. Execute: Execution of the instruction is done by the combination of various units collectively called as execution unit.

Next we will know how pipelining technique is advantageous when compared to the traditional sequential execution.

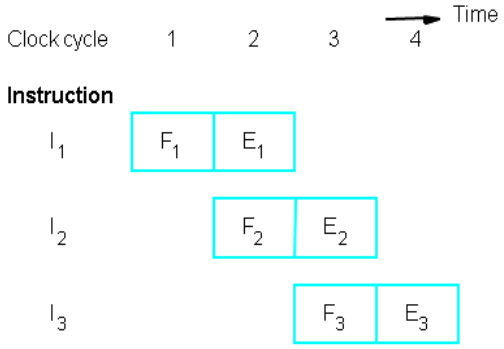
Assume that there are three instructions I1, I2 and I3 to be executed by a processor. Following are the sequence of operations performed when it is carried sequentially in the processor. Assume that each of the unit takes only one clock cycle for completing the operation.



(a) Sequential execution

As you can see from the above figure that sequential execution takes 6 clock cycles all together for three instructions taking one clock cycle for each of the unit.

When the same sequences of instructions are being overlapped in the pipeline, following are the sequence of execution taking place:

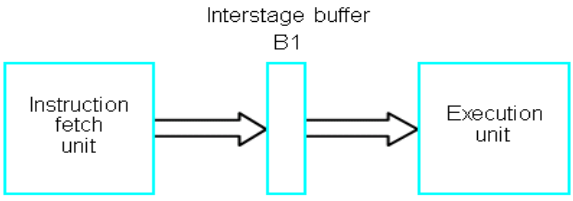


(c) Pipelined execution

When the same sets of instructions are executed in pipeline, it is executed in just 4 clock cycles due to the overlapping of the operation. Hence the operation completes 2 cycles faster.

The above illustrated pipeline is a two stage pipeline which has only two stages fetch and execute.

Hardware arrangement of two stage pipeline



(b) Hardware organization

There is a buffer in between the stages of pipeline called as an **Interstage buffer.** This buffer is used to store the value that is being processed in the fetch stage and from this buffer execution unit accessed required data. Since the stages are taken up by the new instruction in every clock cycle, it is very much important to store the data to retain it to the next stage.

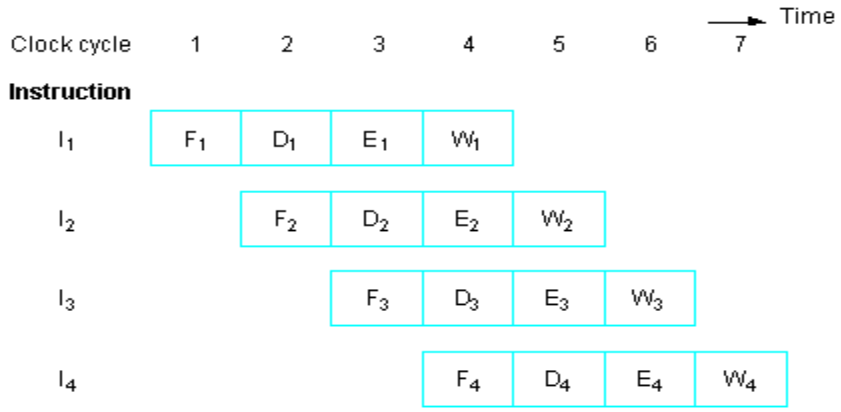
8.1.2 Four Stage pipeline

In the last section, we have seen two stage pipelines which had only two stages fetch and execute. The execute stage over there is a internal combination of many other units.

Modern pipeline exposes the internal units of those execute stage and treat all those units as separate stages which are made of three stages: Decode, Execute and write All together there are four different stages in a four stage pipeline:

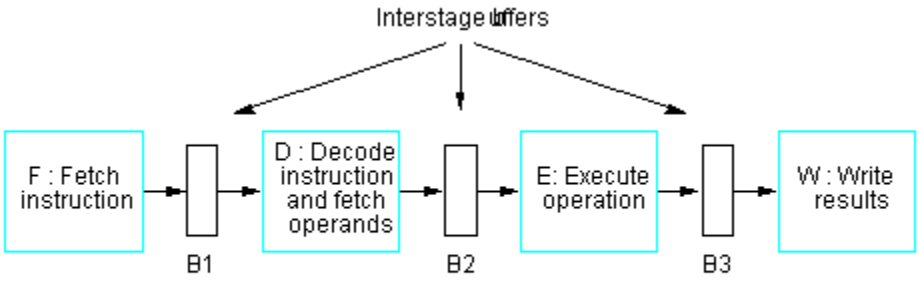
1. Fetch (F): Unit that reads the instruction from the memory.
2. Decode (D): Unit that identifies the operation and operand in the instruction.
3. Execute (E): It is a ALU that performs the required operation.
4. Write (W): Unit that stores the result into required destination location.

Below shown is the pipeline sequence for the execution of instruction I1, I2, I3 and I4 which takes one clock cycle for processing in every stage.



(a) Instruction execution divided into four steps

Hardware arrangement of Four stage pipeline



(b) Hardware organization

There has to be a small buffer called as interstage buffer in between each and every stages. Hence there three interstage buffers B1, B2 and B3. B1 is a buffer between fetch and decode stage, B2 is a buffer between decode and execute stage and b3 is a buffer between execute and write stage.

One of the most important criteria over is choosing the clock period because all the stages should complete their operation in one clock cycle. Some of the stages may take more time and some other may take less time. Hence while choosing the clock period for a single cycle, it is chosen

such that the longest stage also completes operation in a single cycle. So clock period is chosen to be the period of longest stage.

For example: if fetch takes 40ns , decode takes 20ns , execute takes 30ns and write takes 15 ns. The clock period is chose to be 40ns which is longest among all the stages.

8.2 Role of Cache memory in pipelining

Cache is a fastest primary memory unit which has less memory capacity when compared with the main memory unit and is built on top of the processor chip.

Speed of the internal processor operation is very high when compared with the external operations. There will be a reasonable amount of delay when the operation is done external to the processor.

In pipeline, ideal condition says that every stage has to complete its operation in one clock cycle and cycle time is chosen such that it is the longest time of all stages so that every stage can complete its operation within that limit. According to this principle fetch stage is considered to be the longest stage as it requires fetching the instruction from external memory device and cycle time is very much large when compared to all other stages and if it is chosen as cycle time, all other stages has to wait for a long period. So to overcome this problem, few instructions before execution are placed on the small and fastest memory chip called as **cache,** which is built on top of processor chip and time to complete operation is very small and can operate in a speed equal to processor speed.

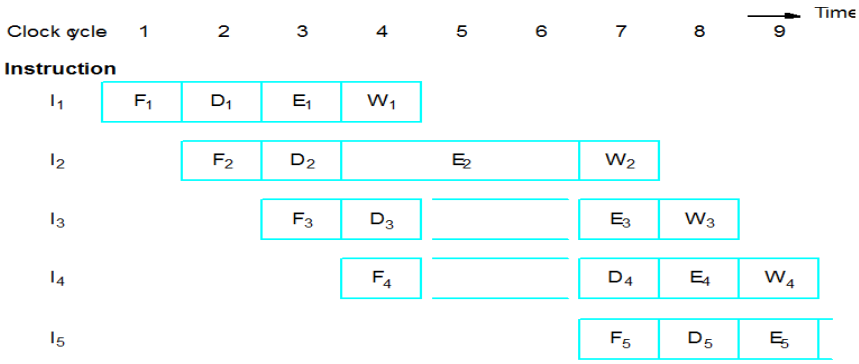
Hence to save the delay in terms of clock cycle cache memories play a very vital role especially in pipeline systems.

8.3 Pipeline Hazards

What are hazards in pipeline?

Ideally each and every stage in a pipeline has to complete its operation in one clock cycle. Unfortunately for variety of reasons, a stage may take more than one clock cycle to complete its operation.

As an example, consider the sequence of pipeline operations were in execution stage takes 3 clock cycles to complete its operation. When a timing diagram is plotted for the operation, following is the sequence.



In this condition, as according to the above timing diagram, pipeline is said to have stalled for **2 clock cycles.** I.e. the execution stage of I2 takes extra cycles, clock 5 and clock 6. Because of that all other stages are delayed by 2 clock cycles for successive instructions. This delay is called as stall. **Any condition that causes pipeline to stall is called as hazard.**

Basically it is a condition which causes pipeline to deviate from its normal functioning and work abnormally opposite to the principles.

8.3.1 Types of Hazards

Hazards in pipeline are very broadly classified into three different categories based on the situation of occurrence:

1. Data Hazard
2. Control Hazard
3. Structural Hazard

We will have a very detailed discussion on each of these hazards.

8.4 Data Hazard

It is a condition which causes stall in the pipeline due to the delay in the availability of required data operand of an instruction at the expected time in the pipeline.

Remember these kinds of hazards are caused for producing the correct result in the operation and are not occurred due to the functioning of the pipeline.

We will consider a simple example and let us analyze the need of the data hazard or the situation when exactly the data hazard is required to occur in the pipeline. Consider two instructions:

$A \leftarrow 3 + A$

$B \leftarrow 4 \times A$, were A=5

We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially. When executed in sequence, the operation is:

$A \leftarrow 3 + 5 = 8$

$B \leftarrow 4 \times 8 = 32$

If the same sets of instructions are executed in pipeline overlapping the stages, then the result would be:

$A \leftarrow 3 + 5 = 8$

$B \leftarrow 4 \times 5 = 20$

The above operation on an ideal pipeline produces a wrong result. So to overcome this problem, hazards are created in the pipeline when you have the instructions of above type.

To be precise, data hazards are mainly caused when there is a data dependency between the instructions.

Data Dependency: If source operand of the current instruction is the destination operand of the previous instruction, then it is called as dependent instructions and the process is called as **data dependency.**

To get a clear picture on data dependency, consider the set of instructions I1 and I2 shown below

I1: Mul R2, R3, R4

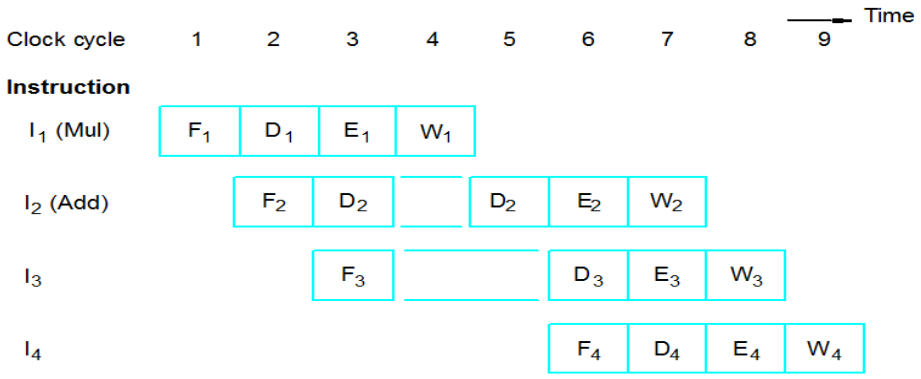
I2: Add R5, R4, R6

As shown in the above instructions, in the instruction I2 R4 is one of the source operand and in I1 R4 is the destination operand. So instruction I2 is data dependent on instruction I1 which means that instruction I2 has to execute only after instruction I1 produces the result. I.e. after instruction I1 completes its **write stage.**

In the case of data dependency, basic constraint that must be enforced to guarantee correct results is causing the stalls (data hazards) in the pipeline till the required result from the previous instruction is obtained.

There is a special hardware in the decoding stage to detect the data dependency.

Timing diagram for the pipeline operation of the same with data hazard is shown below:



Pipeline stalled by data dependency between D_2 and W_1.

As shown in the timing diagram, a stall of 2 cycles is created which is nothing but the data hard occurred. Decode stage of I2 takes two extra cycles. So there is a stall of 2 clock cycles.

8.4.1 Operand Forwarding

Operand forwarding is one of the techniques that reduce stalls in the pipeline caused due to data hazard. Data hazard basically occurs when there is a data dependency between the instructions where current instruction requires the destination operand of previous instruction as the source operand.

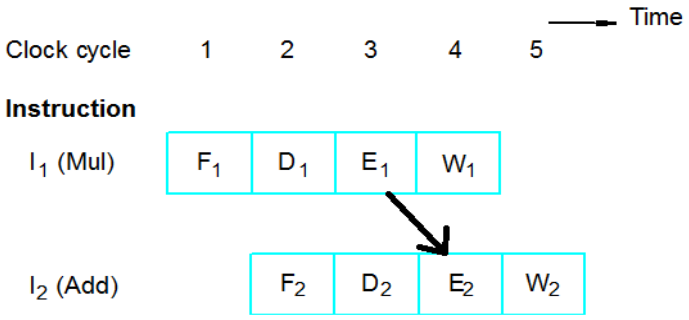
To illustrate this technique, let us consider the same example that is been discussed in the previous section:

I1: Mul R2, R3, R4

I2: Add R5, R4, R6

Here at the outset we know that source R4 of I2 is dependent on the destination R4 of the I1. Hence it is required of I2 to wait until I1 completes write. But according to the actual condition, it can be interpreted as instruction I2 requires the result produced by I1 in the execution stage. And when it is put in a pipeline, result is available immediately after the execution stage of I1 which is again used in the execution stage of I2.

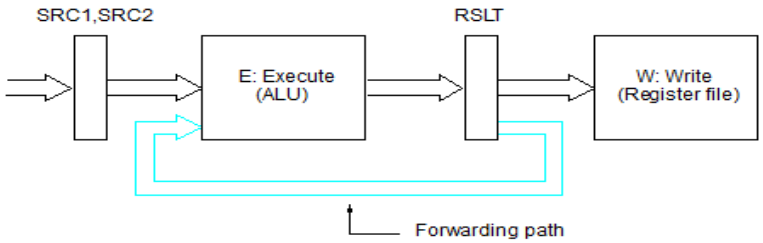
Hence the delay is reduced by forwarding (bypassing) the executed result back as operand into the execution stage of next clock cycle. Typical timing representation for this is can be plotted as shown below:



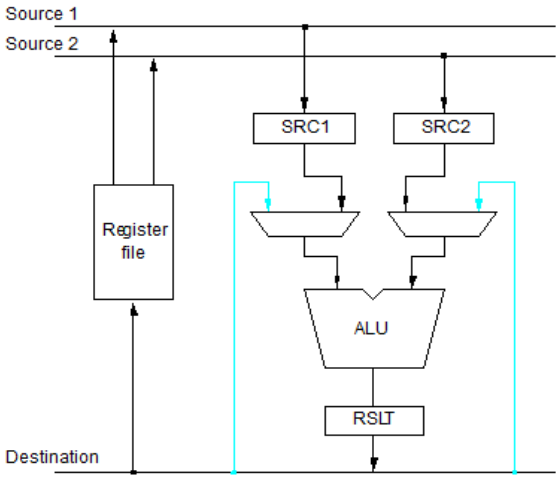
As shown in the timing diagram above, the arrow mark into the execution stage in cycle 3 to cycle 4 suggests the operand forwarding technique. As observed there is no stall in the pipeline when this technique is implemented.

Hardware of Operand forwarding

To implement this technique, it requires a little modification into the existing hardware of the pipeline which is shown below:



Detailed functional unit of this hardware is arranged according to the one shown below:



(a) Datapath

Hardware looks almost similar to three bus architecture were in two output lines into the bus source1 and source2 and one input line from the destination bus. The interstage buffer between decoder and execution unit is split up into two registers holding the source operand SRC1 and SRC2 respectively. Interstage buffer RSLT between execution stage and write stage collects the result operated by the ALU.

We know that there is hardware in the decoding unit which detects the data dependency between the instructions. If there is a data dependency, operand forwarding technique works according to the following steps:

1. It will first which of the two operands is data dependent among SRC1 and SRC2.
2. Once after detecting that, it will activate the required line through the multiplexer.
3. Data from the activated line goes back into the ALU through multiplexer.

8.4.2 Handling Data Hazards through software

Instead of discovering the data dependency through hardware in the decoding unit, it can be detected and controlled through the software, were compiler takes care of detecting the dependency during the compile time itself.

To ensure the proper execution, compiler inserts No operation or null instruction to the required number of clock cycles to insert stalls or idle cycles. Inserting a single no op is equal to inserting 1 cycle delay.

For example for the instruction

I1: Mul R2, R3, R4

I2: Add R5, R4, R6

Above instruction requires 2 cycles delay between instruction I1 and instruction I2. So two no operation instructions are inserted as shown below:

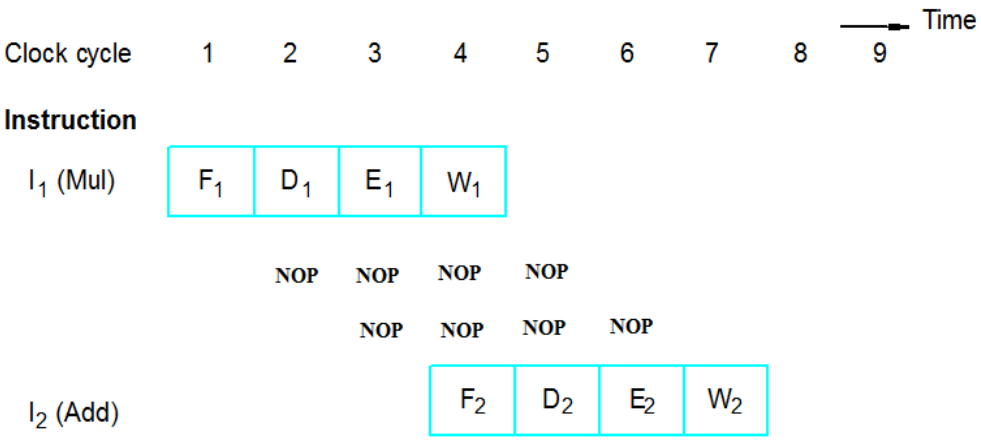
I1: Mul R2, R3, R4

NOP

NOP

I2: Add R5, R4, R6

Following is the timing diagram which ensures the correct working



8.4.3 Side Effects

When the operand other than the one explicitly named in the instruction as a destination operand is affected, then the instruction is said to have the side effect. As there is no method to detect the dependency in that case which leads to wrong output.

As an example consider the following instruction

Add (R2)+, R1

Mul R2, R3

According to the basic definition of the data dependency, there is no dependency in the above instruction. But in add instruction it is a source as well as destination. Because after accessing it, it changes the value which effects wrong value to be read by instruction Mul.

Solution: There is no such effective solution to overcome this problem. All that can be done is that whenever it is pipeline operation, these kind of instructions are avoided.

8.5 Instruction Hazards

Stalling of the pipeline due to the delay in the availability of the instruction for execution. It can basically occur in two conditions:

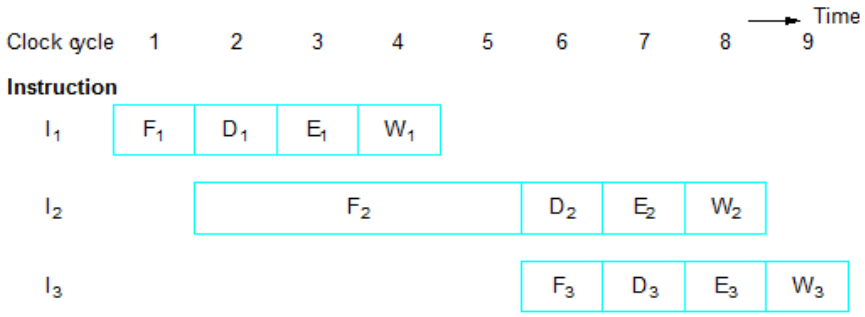
1. Due to cache Miss
2. Due to Branch instruction

We will discuss each one of this in detail.

8.5.1 Instruction Hazard due to Cache Miss

When the instruction that needs to be fetched is not present in the cache memory, the scenario is called as **cache miss.** When there is cache miss, instructions need to be fetched from the main memory unit which takes lot of delay and create stalls. This stall due to cache miss is called instruction hazard.

Consider that there are three instructions I!, I2 and I3 were I2 is not present in the cache. Let us analyze how the hazard is caused according to the timing signals of the pipeline. Assume that fetching from main memory takes 4 clock cycles.



(a) Instruction execution steps in successive clock cycles

In the above case the stall created is 3 clock cycles.

8.5.2 Instruction Hazard due to Branch instruction

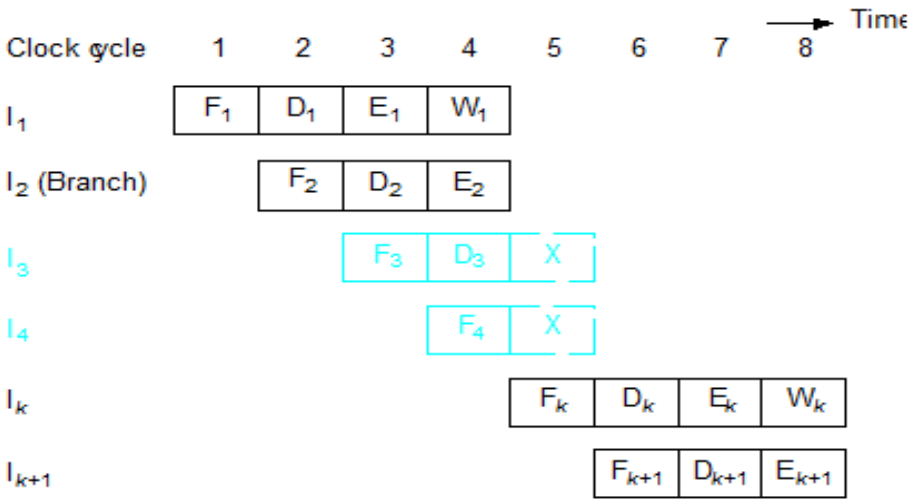
There are two types of branch instructions, both of them contributes to the instruction hazard:

1. Unconditional Branches
2. Conditional Branches

8.5.2.1 Unconditional Branches

We know that unconditional branches are the instructions that change the sequence of fetching without any condition. To see how exactly it causes hazards in pipeline, let us consider the execution sequence of following instructions.

Let I1, I2 (unconditional branch), I3,….In be the instructions in sequence. Let I_k be the branch target address of the unconditional branch instruction I2. Following are the execution sequence when put up in a pipeline



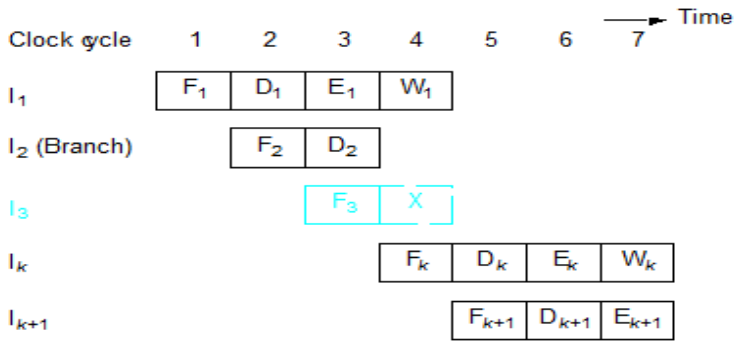
In the above pipeline operation, after instruction I2, in cycle 3 it should have fetched instruction I_k as it is a branch instruction. But due to the delay of the execution unit, it fetched instruction I_k in the cycle 5 were in it took two cycle delay in fetching the actual instruction. In the course of delay there are two unwanted instructions I3 and I4 fetched in. The slots were these instructions are fetched in is called as **Branch Delay slots.**

Time lost as a result of branch instruction is called **Branch Penalty.** In the above case branch penalty is 2 as it has fetched two unwanted instruction in two cycles.

Reducing Branch Penalty

One of the simplest methods to reduce the branch penalty is to compute the BTA earlier in the pipeline itself. To implement this technique, there will be dedicated hardware unit the fetch unit to identify the branch instruction which will compute BTA through its special decoder as soon as branch instruction is fetched.

Following is the sequence when previous set of instructions is put up in the pipeline



(b) Branch address computed in Decode stage

As observed in the above timing diagram, there is only one clock cycle penalty and one delay slot which means it has reduced one cycle delay.

8.6 Conditional Branch and Branch Prediction

When it is conditional branch instruction, there are some differences when compared with unconditional branch instructions. One of them is that it executes the branch target address only if the condition is satisfied. Another thing is that when it is a conditional branch instruction, it depends on the result of the previous instruction also. Dependency can be easily ruled out by operand forwarding technique.

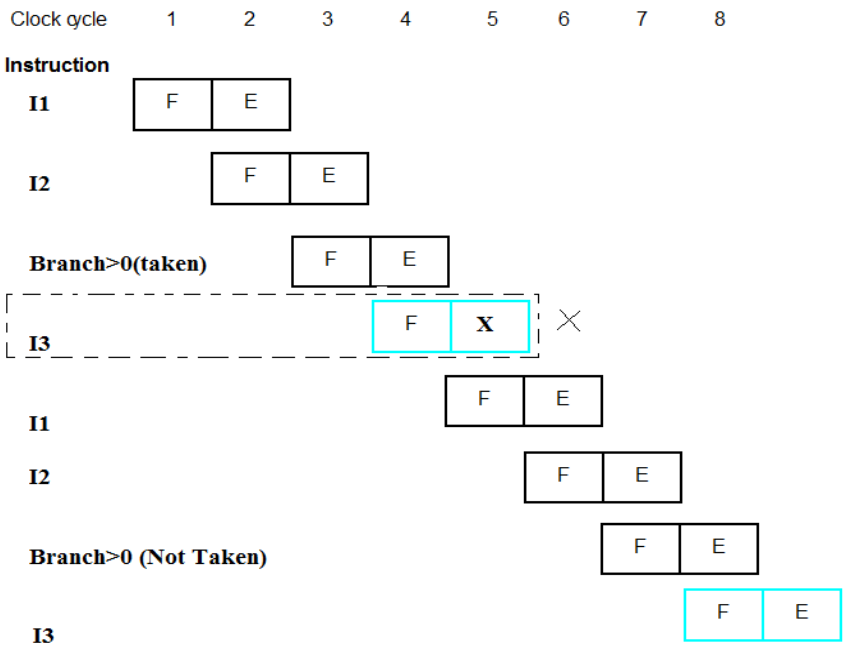
Another most important thing to be analyzed is that when executed branch may be taken or it may not be taken based on the condition. If branch is taken wrongly fetched instruction needs to be discarded or else they have to be executed.

When branch is taken, decoder hardware detects that the wrongly fetched instruction needs to be discarded and it stops its execution. This process is called as **Flushing.**

To illustrate its working, let us consider the following sequence of instructions:

```
Loop:  I1
       I2
       Branch > 0      Loop
       I3
       I4
```

Here according to the above sequence, if branch is taken, it has to go back to the Loop or else it should continue with the sequence. Let us assume that first time it is taken and next time it is not taken and let's see the sequence of action in a two stage pipeline.



In the above timing sequence, it can be observed that when taken, I3 is a wrong instruction fetched and it needs to be flushed out from the execution as shown and if it is not taken the execution in the sequence is accepted.

8.6.1 Delayed Branch

This is one of the techniques to minimize penalty in case of conditional branch instructions. Implementation technique is very simple. We know that location following the branch instruction is called as **branch delay slot** and instruction is always fetched into the delay slot and are executed based on the outcome of the decision of the branch instruction. If branch is taken then fetched instruction needs to be flushed as it is an unwanted instruction fetched.

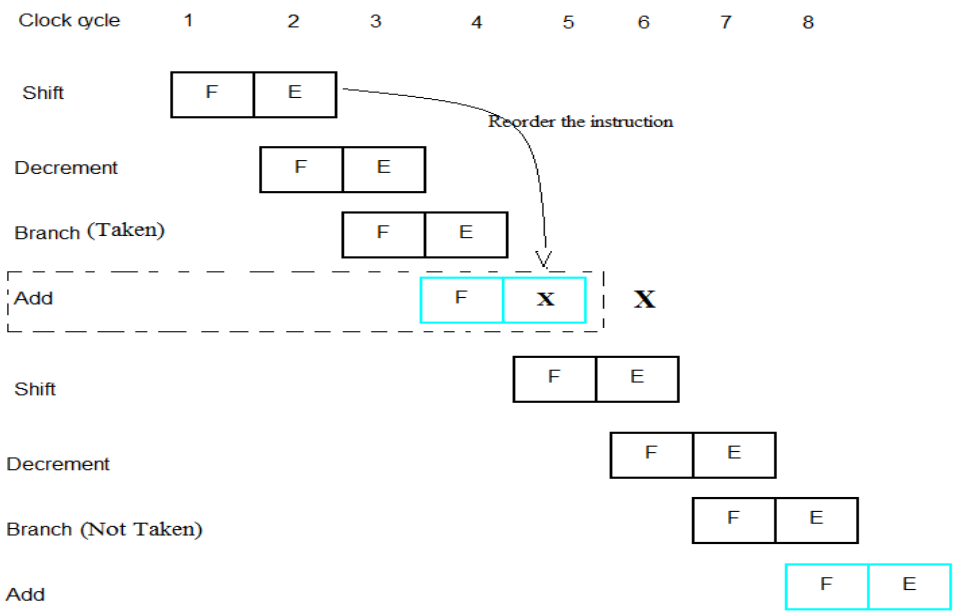
The idea is very simple over here, instruction in delay slot are always fetched and whether the branch is taken or not taken, they are always made to execute by reordering the instructions such that the delay slot is filled with some useful instruction that has to be executed without depending on the outcome of the branch instruction.

Reordering technique is implemented by the compiler and it reorders instruction such that instruction following the branch is some useful instruction that doesn't depend on the outcome of the branch.

In case if there are no useful instructions, they are filled with No operations.

To illustrate the technique, let's consider the following sequence of instructions

```
LOOP    Shift_left    R1
        Decrement     R2
        Branch=0      LOOP
NEXT    Add           R1,R3
```



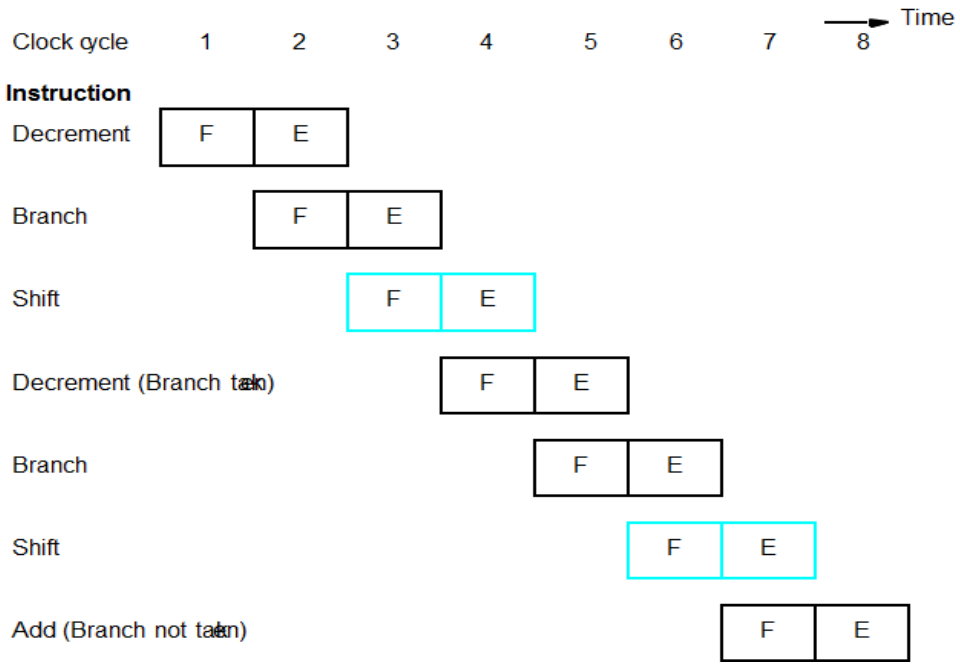
As shown in the above timing diagram, there is a wrong instruction Add fetched in the cycle 4 as the branch is taken. The fetched instruction needs to be flushed from its execution. In this case some useful instruction that has to be executed without depending on branch outcome can be placed so that flushing can be avoided.

To insert an useful instruction independent of branch instruction, compiler reorders the instruction such a way that it finds some independent instruction prior to branch which should be executed and places that instruction inside the delay slot.

In the above considered example, shift is and\ independent instruction that is kept in a delay slot. So irrespective of the branch instruction result, it should always be executed.

Following is the re arrangement of code and its corresponding timing diagram:

```
LOOP    Decrement     R2
        Branch=0      LOOP
        Shift_left    R1
NEXT    Add           R1,R3
```



8.7 Branch Prediction

Another technique for reducing the branch penalty if delayed branch technique doesn't work well. The technique is based on prediction rule were in it predicts if a branch is taken or not taken. There are two types of prediction:

1. Static prediction
2. Dynamic prediction

8.7.1 Static Branch Prediction

As the name indicates, the technique is based on fixed prediction scheme. There will be fixed prediction based on the practical observation for the outcome of every branch instructions. It can be predicted as Taken or Not Taken. When they come for the execution decision is made on the prediction. This type prediction is taken care by the compiler were it maintains a prediction table and does the execution accordingly.

Even though simple not a effective prediction scheme as it goes wrong most of the times.

8.7.2 Dynamic Prediction

The main objective of this prediction scheme is to reduce making the wrong predictions. Here for the effective prediction, processor keeps track of the branch decisions made every time when a particular branch instruction is executed.

Recent branch decisions are maintained in a special table called branch history table or sometimes they are even attached as a bit value within the branch instructions.

Every time prediction is made on the outcome of the most recent history that is kept track.

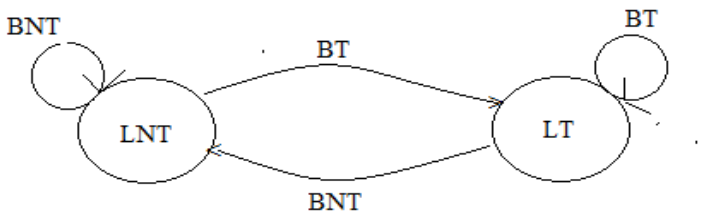
There are two different types of dynamic prediction scheme:

1. Two state machine
2. Four State machine

Two State Machine

As the name indicates, there are two states Likely Not Taken (LNT) and Likely Taken (LT) to predict the current state of the branch. If the state is LNT, branch is predicted to be not taken and if the state is LT, branch is predicted to be taken. In this case a history of the branch is maintained in terms of single bit. i.e. 0 indicates LNT and 1 indicates LT. Any time if the prediction changes history value immediately changes. If it is 0, it becomes 1 and if it is 1, it becomes 0.

Corresponding is the sate machine model for the prediction scheme:



Four State machine

Here there are four different states:

ST: Strongly Taken

LT: Likely Taken

LNT: Likely Not Taken

SNT: Strongly Not Taken

Two states ST and SNT indicate prediction to be taken. And another two states LT and LNT predicts the branch as Not taken. Here in this method, prediction value changes only if the consecutive predictions go wrong. History value in this scheme is maintained in 4 bits were:

00: Strongly Not Taken

01: Likely Not Taken

11: Strongly Taken

10: Likely Taken

Corresponding timing diagram gives a clear picture of this scheme: