DAR ES SALAAM INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER STUDIES
COU07302 MICROPROCESSOR AND COMPUTER ARCHITECTURE
Lecture 1 - Pipelining
by
E. Kondela

Introduction

It is observed that organization enhancements to the CPU can improve performance. We have already seen that use of multiple registers rather than a single a accumulator, and use of cache memory improves the performance considerably. Another organizational approach, which is quite common, is instruction pipelining.

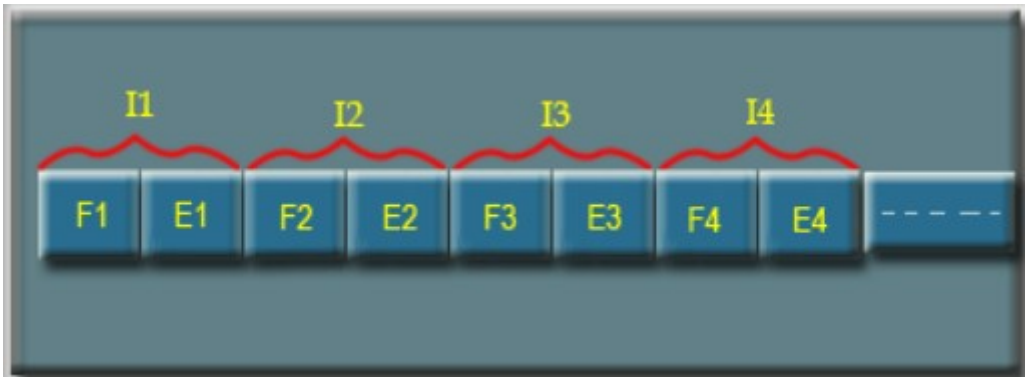
Pipelining is a particularly effective way of organizing parallel activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly line operation.

By laying the production process out in an assembly line, product at various stages can be worked on simultaneously. This process is also referred to as pipelining, because, as in a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.

To apply the concept of instruction execution in pipeline, it is required to break the instruction in different task. Each task will be executed in different processing elements of the CPU.

As we know that there are two distinct phases of instruction execution: one is instruction fetch and the other one is instruction execution. Therefore, the processor executes a program by fetching and executing instructions, one after another.

Let Fi and Ei refer to the fetch and execute steps for instruction Ii. Execution of a program consists of a sequence of fetch and execute steps is shown in the figure on the next slide.



Now consider a CPU that has two separate hardware units, one for fetching instructions and another for executing them.

The instruction fetch by the fetch unit is stored in an intermediate storage buffer B1. The results of execution are stored in the destination location specified by the instruction.

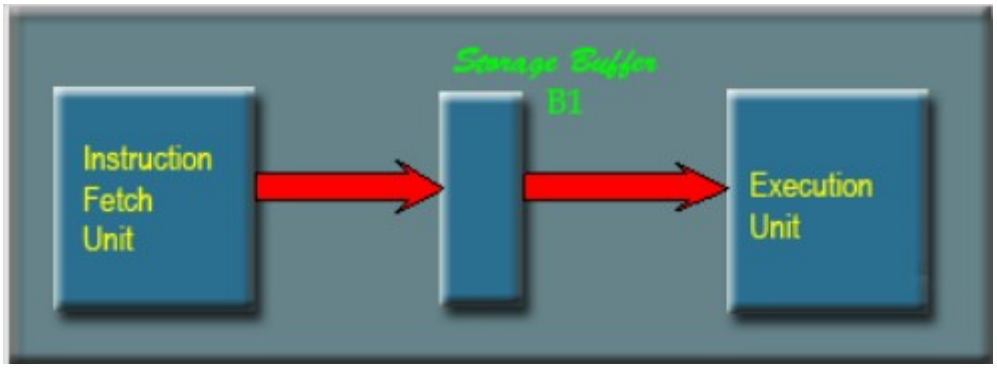
For simplicity it is assumed that fetch and execute steps of any instruction can be completed in one clock cycle.

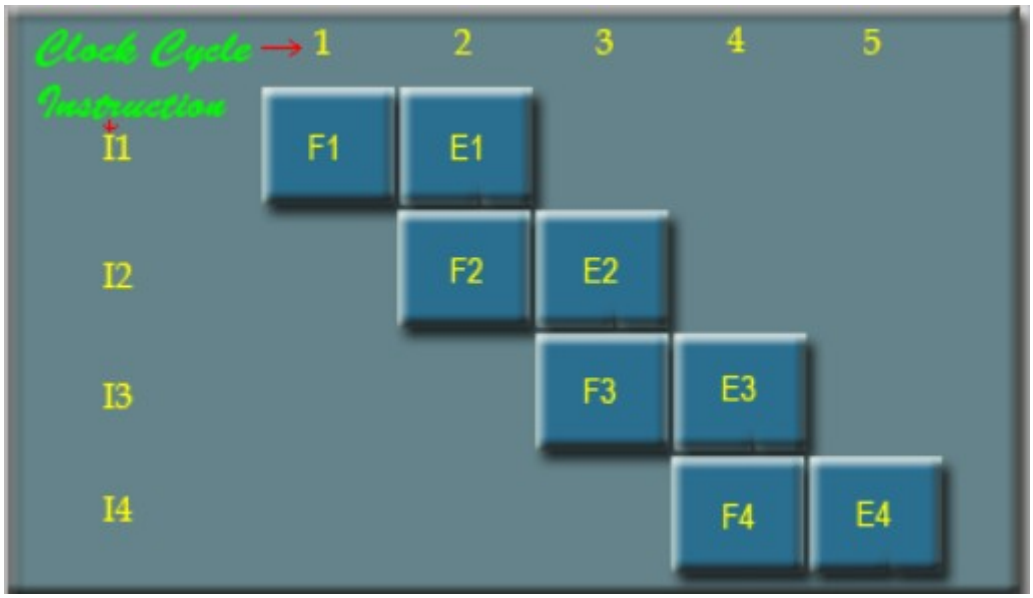
The operation of the computer proceeds as follows:

- In the first clock cycle, the fetch unit fetches an instruction (instruction *I*1, step *F*1) and stored it in buffer *B*1 at the end of the clock cycle.
- In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction *I*2 (step *F*2).
- Meanwhile, the execution unit performs the operation specified by instruction *I*1which is already fetched and available in the buffer *B*1 (step *E*1).
- By the end of the second clock cycle, the execution of the instruction *I*1 is completed and instruction *I*2 is available.
- Instruction *I*2 is stored in buffer *B*1 replacing *I*1 which is no longer needed.
- Step *E*2 is performed by the execution unit during the third clock cycle, while instruction *I*3 is being fetched by the fetch unit.

- Both the fetch and execute units are kept busy all the time and one instruction is completed after each clock cycle except the first clock cycle.
- If a long sequence of instructions is executed, the completion rate of instruction execution will be twice that achievable by the sequential operation with only one unit that performs both fetch and execute.

Basic idea of instruction pipelining with hardware organization is shown in the figure on the next slide.





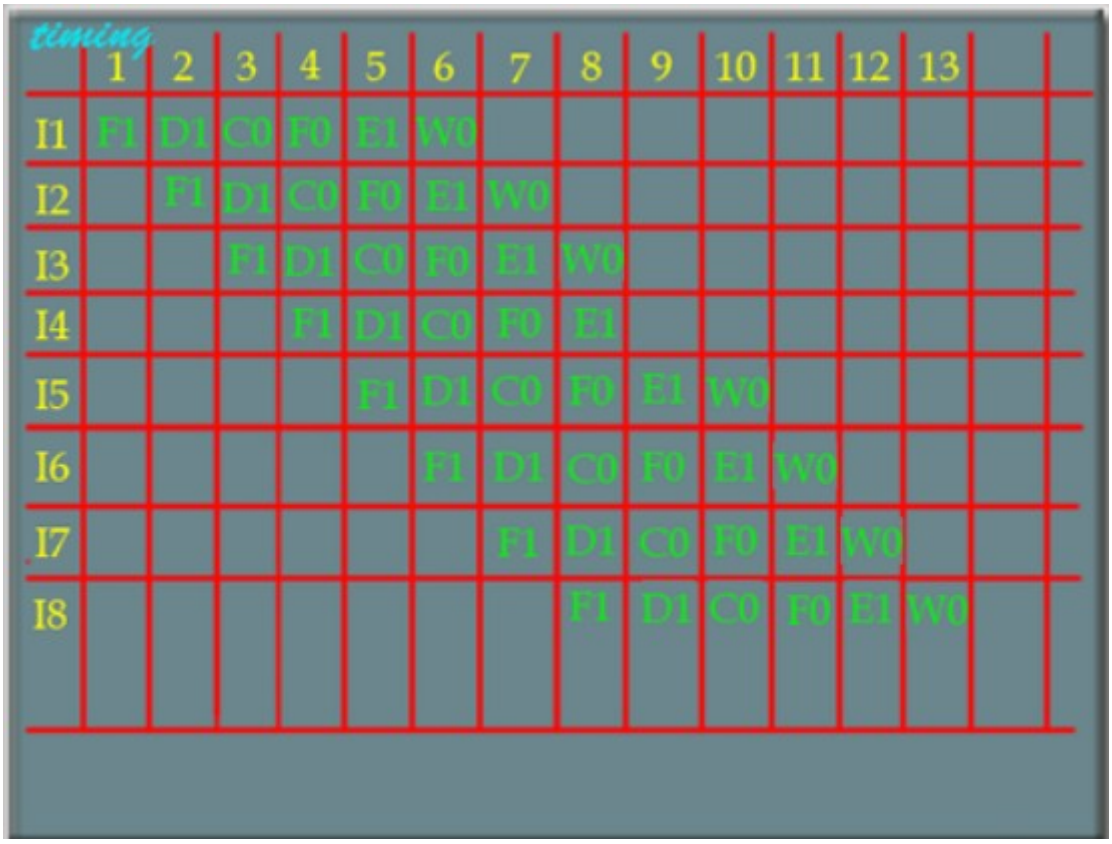
The processing of an instruction need not be divided into only two steps. To gain further speed up, the pipeline must have more stages.

Let us consider the following decomposition of the instruction execution:

- Fetch Instruction (FI): Read the next expected instruction into a buffer.
- Decode Instruction ((DI): Determine the opcode and the operand specifiers.
- Calculate Operand (CO): calculate the effective address of each source operand.
- Fetch Operands(FO): Fetch each operand from memory.
- Execute Instruction (EI): Perform the indicated operation.
- Write Operand(WO): Store the result in memory.

There will be six different stages for these six subtasks. For the sake of simplicity, let us assume the equal duration to perform all the subtasks. It the six stages are not of equal duration, there will be some waiting involved at various pipeline stages.

The timing diagram for the execution of instruction in pipeline fashion is shown in the figure on the next slide.



From this timing diagram it is clear that the total execution time of 8 instructions in this 6 stages pipeline is 13-time unit. The first instruction gets completed after 6 time unit, and there after in each time unit it completes one instruction. Without pipeline, the total time required to complete 8 instructions would have been 48 (6 X 8) time unit. Therefore, there is a speed up in pipeline processing and the speed up is related to the number of stages.

Pipeline Performance

The cycle time τ 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

$$\tau = \max[\tau_i] + d = \tau_m + d \qquad 1 \le i \le k$$

where

$\tau_m =$ maximum stage delay (delay through stage which experience 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 $\tau_m \gg d$.

Now suppose that *n* instructions are processed and these instructions are executed one after another. The total time required τ_k to execute all *n* instructions is

$$T_k = \left[k + (n-1)\right]\tau$$

A total of *k* cycles are required to complete the execution of the first instruction, and the remaining *(n-1)* instructions require *(n-1)* cycles.

The time required to execute *n* instructions without pipeline is

$$T_1 = nk\tau$$

because to execute one instruction it will take $n\tau$ cycle.

The speed up factor for the instruction pipeline compared to execution without the pipeline is defined as:

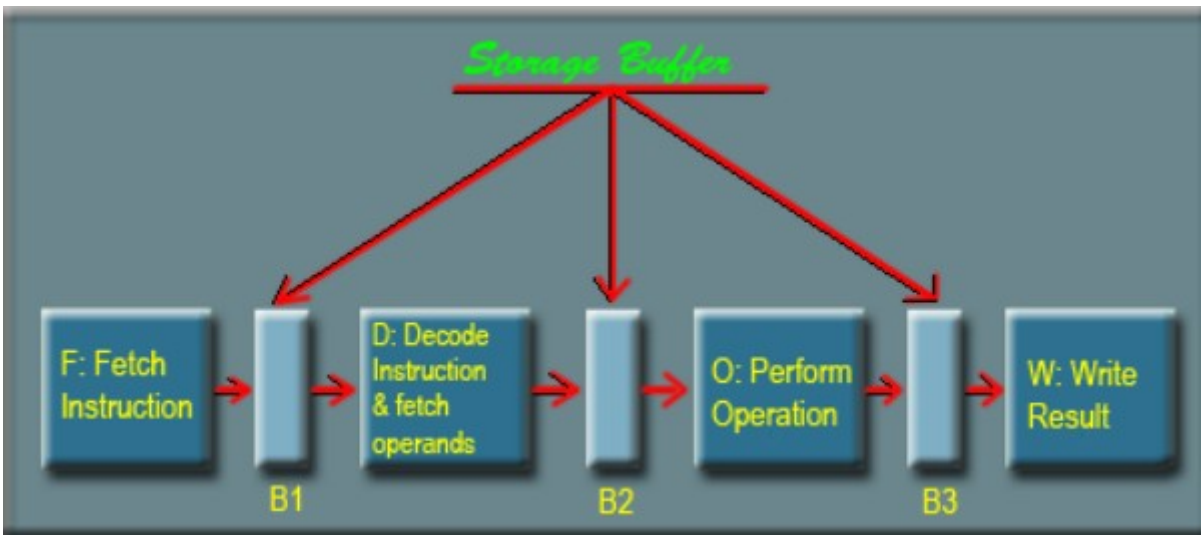
$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{\left[k+(n-1)\right]\tau} = \frac{nk}{k+(n-1)} = \frac{nk}{(k-1)+n}$$

In general, the number of instruction executed is much more higher than the number of stages in the pipeline So, the *n* tends to α, we have

$$S_k = k,$$

i.e. We have a k fold speed up, the speed up factor is a function of the number of stages in the instruction pipeline.
Though, it has been seen that the speed up is proportional to number of stages in the pipeline, but in practice the
speed up is less due to some practical reason. The factors that affect the pipeline performance is discussed next.

Effect of Intermediate storage buffer:
Consider a pipeline processor, which process each instruction in four steps;
F: Fetch, Read the instruction from the memory
D: Decode, decode the instruction and fetch the source operand (S)
O: Operate, perform the operation
W: Write, store the result in the destination location.
The hardware organization of this four-stage pipeline processor is shown next.



In the preceding section we have seen that the speed up of pipeline processor is related to number of stages in the pipeline, i.e, the greater the number of stages in the pipeline, the faster the execution rate. But the organization of the stages of a pipeline is a complex task and if affects the performance of the pipeline.

The problem related to more number of stages:

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.

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.

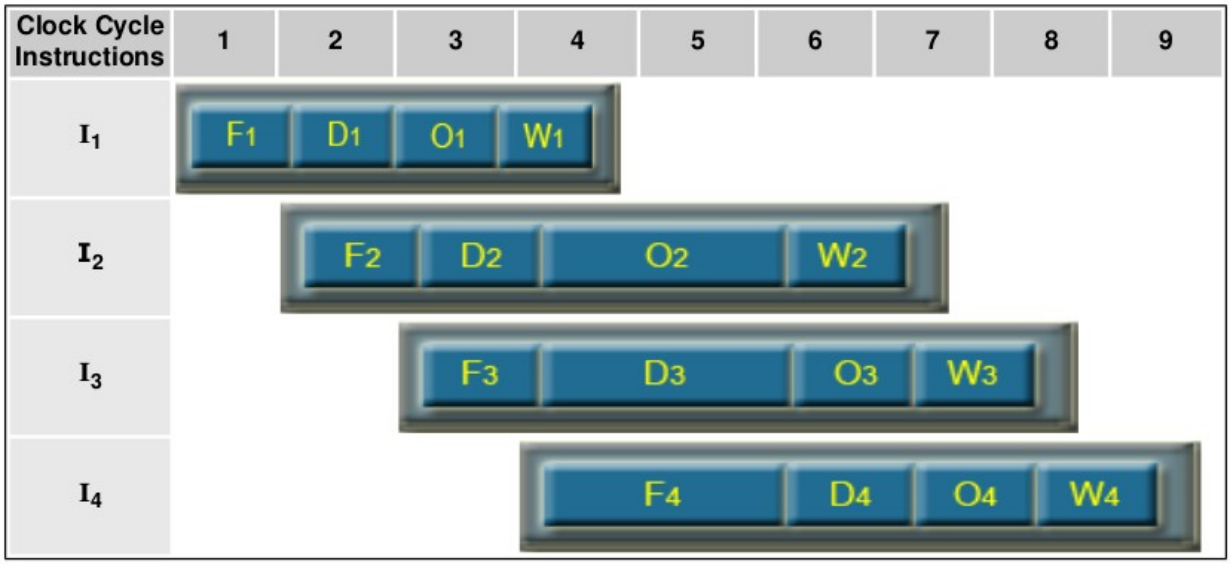
Apart from hardware organization, there are some other reasons which may effect the performance of the pipeline.

(A) Unequal time requirement to complete a subtask:

Consider the four-stage pipeline with processing step Fetch, Decode, Operand and write.

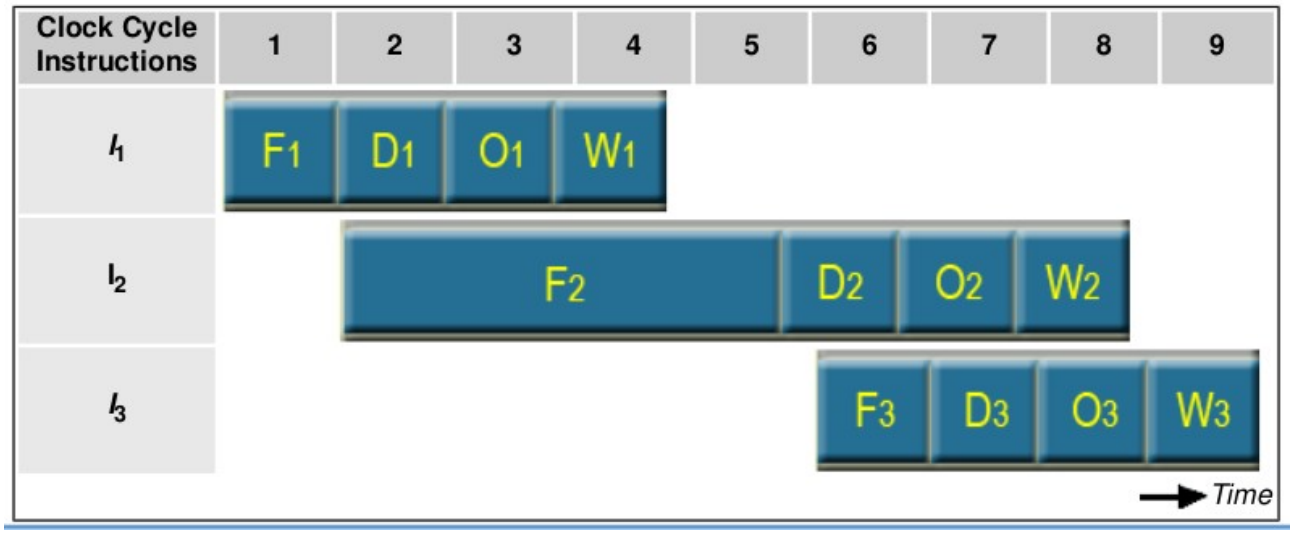
The stage-3 of the pipeline is responsible for arithmetic and logic operation, and in general one clock cycle is assigned for this task

Although this may be sufficient for most operations, but some operations like divide may require more time to complete. Following figure shows the effect of an operation that takes more than one clock cycle to complete an operation in operate stage.



The operate stage for instruction *I*2 takes 3 clock cycle to perform the specified operation. Clock cycle 4 to 6 required to perform this operation and so write stage is doing nothing during the clock cycle 5 and 6, because no data is available to write.
Meanwhile, the information in buffer B2 must remain intake until the operate stage has completed its operation. This means that stage 2 and stage 1 are blocked from accepting new instructions because the information in B1 cannot be overwritten by a new fetch instruction.
The contents of B1, B2 and B3 must always change at the same clock edge.
Due to that reason, pipeline operation is said to have been stalled for two clock cycle. Normal pipeline operation resumes in clock cycle 7. Whenever the pipeline stalled, some degradation in performance occurs.

Role of cache memory:
The use of cache memory solves the memory access problem. Occasionally, a memory request results in a cache miss. This causes the pipeline stage that issued the memory request to take much longer time to complete its task and in this case the pipeline stalls. The effect of cache miss in pipeline processing is shown in the figure.



Clock Cycle Stages	1	2	3	4	5	6	7	8	9	10
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_4	F_3			
D: Decode			D_1	*idle*	*idle*	*idle*	D_2	D_3		
O: Operate				O_1	*idle*	*idle*	*idle*	O_2	O_3	
W: Write					W_1	*idle*	*idle*	*idle*	W_2	W_3
										→ *Time*

Function performed by each stage as a function of time.

Function performed by each stage as a function of time

In this example, instruction *I*1 is fetched from the cache in cycle 1 and its execution proceeds normally. The fetch operation for instruction *I2* which starts in cycle 2, results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for *I2* to arrive.

We assume that instruction *I2* is received and loaded into buffer B1 at the end of cycle 5, It appears that cache memory used here is four time faster than the main memory.

The pipeline resumes its normal operation at that point and it will remain in normal operation mode for some times, because a cache miss generally transfer a block from main memory to cache.

From the figure, it is clear that Decode unit, Operate unit and Write unit remain idle for three clock cycle. Such idle periods are sometimes referred to as bubbles in the pipeline. Once created as a result of a delay in one of the pipeline stages, a bubble moves downstream until it reaches the last unit. A pipeline can not stall as long as the instructions and data being accessed reside in the cache. This is facilitated by providing separate on chip instruction and data caches.

Dependency Constraints:

Consider the following program that contains two instructions, *I*1 followed by *I*2

*I*1 : A← A + 5

*I*2 : B← 3 * A

When this program is executed in a pipeline, the execution of *I*2 can begin before the execution of *I*1 completes.
The pipeline execution is shown below.

Clock Cycle Stages	1	2	3	4	5	6
F: Fetch	F_1	D_1	O_1	W_1		
D: Decode		F_2	D_2	O_2	W_2	

In clock cycle 3, the specific operation of instruction *I*1 i.e. addition takes place and at that time only the new updated value of A is available. But in the clock cycle 3, the instruction *I*2 is fetching the operand that is required for the operation of *I*2. Since in clock cycle 3 only, operation of instruction *I*1 is taking place, so the instruction will get operation of the old value of A , it will not get the updated value of A , and will produce a wrong result. Consider that the initial value of A is 4. The proper execution will produce the result as

B=27
*I*1: A← A + 5 = 4 + 5 = 9
*I*2: *B← 3 x A = 3 x 9 = 27*

But due to the pipeline action, we will get the result as

*I*1: *A← A* + 5 = 4 + 5 = 9
I2:B← 3 x A = 3 x 4 = 12

Due to the data dependency, these two instructions can not be performed in parallel.

Therefore, no two operations that depend on each other can be performed in parallel. For correct execution, it is required to satisfy the following:

- The operation of the fetch stage must not depend on the operation performed during the same clock cycle by the execution stage.
- The operation of fetching an instruction must be independent of the execution results of the previous instruction.
- The dependency of data arises when the destination of one instruction is used as a source in a subsequent instruction.

Branching

In general when we are executing a program the next instruction to be executed is brought from the next memory location. Therefore, in pipeline organization, we are fetching instructions one after another.
But in case of conditional branch instruction, the address of the next instruction to be fetched depends on the result of the execution of the instruction.

Since the execution of next instruction depends on the previous branch instruction, sometimes it may be required to invalidate several instruction fetches. Consider the following instruction execution sequence:

Time / Instruction	1	2	3	4	5	6	7	8	9	10
I_1	F_1	D_1	O_1	W_1						
I_2		F_2	D_2	O_2	W_2					
I_3			F_3	D_3	O_3	W_3				
I_4				F_4	D_4	O_4	W_4			
I_5					F_5	D_5	O_5	W_5		
I_6						F_6	D_6	O_6	W_6	
I_7							F_7	D_7	O_7	W_7

In this instruction sequence, consider that I3 is a conditional branch instruction.
The result of the instruction will be available at clock cycle 5. But by that time the fetch unit has already fetched the instruction I4 and I5.
If the branch condition is false, then branch won't take place and the next instruction to be executed is I4 which is already fetched and available for execution.
Now consider that when the condition is true, we have to execute the instruction I10 after clock cycle 5, it is known that branch condition is true and now instruction I 10 has to be executed.
But already the processor has fetched instruction I4 and I5 it is required to invalidate these two fetched instruction and the pipe line must be loaded with new destination instruction I 10.

Due to this reason, the pipeline will stall for some time. The time lost due to branch instruction is often referred as branch penalty.

Time / Instruction	1	2	3	4	5	6	7	8	9	10
I_1	F_1	D_1	O_1	W_1						
I_2		F_2	D_2	O_2	W_2					
I_3			F_3	D_3	O_3	W_3				
I_4				F_4	D_4					
I_5					F_5					
I_{10}						F_{10}	D_{10}	O_{10}	W_{10}	
I_{11}							F_{11}	D_{11}	O_{11}	W_{11}

The effect of branch takes place is shown in the figure in the previous slide. Due to the effect of branch takes place, the instruction I4 and I5 which has already been fetched is not executed and new instruction I 10 is fetched at clock cycle 6.
There is not effective output in clock cycle 7 and 8, and so the branch penalty is 2. The branch penalty depends on the number of stages in the pipeline. More numbers of stages results in more branch penalty.

Dealing with Branches:
One of the major problems in designing an instruction pipe line is assuming a steady flow of instructions to the initial stages of the pipeline. The primary problem is the conditional brancho instruction 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 single pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and sometimes it 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 as additional stream.

Prefetch Branch target
When a conditional branch is recognized, the target of the branch is prefetced, 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,.

Loop Buffer:

A top buffer is a small, very high speed memory maintained by the instruction fetch stage of the pipeline and containing the most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first cheeks 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. Thus, instructions fetched in sequence will be available without the usual memory access time.
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. This is usual for the common occurrence of IF-THEN and IF-THEN-ELSE sequences.
3. This strategy is particularly well suited for 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.

The loop buffer is similar in principle to a cache dedicated to instructions. The differences are that the loop buffer only retains instructions in sequence and is much smaller in size and hence lower in cost.

Branch Prediction :
Various techniques can be used to predict whether a branch will be taken or not. The most common techniques are:

- 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 upto the time of the conditional branch instructions. The later two approaches are dynamic- they depend on the execution history.

Predict never taken always assumes that the branch will not be taken and continue to fetch instruction in sequence. Predict always taken assumes that the branch will be taken and always fetch the branet target In these two approaches it is also possible to minimize the effect of a wrong decision.
If the fetch of an instruction after the branch will cause a page fault or protection violation, the processor halts its prefetching until it is sure that the instruction should be fetched. Studies analyzing program behaviour have shown that conditional branches are taken more than 50% of the time, and so if the cost of prefetching from either path is the same, then always prefetching from the branch target address should give better performance than always prefetching from the sequential path.
However, in a paged machine, prefetching the branch target is more likely to cause a page fault than prefetching the next instruction in the sequence and so this performance penalty should be taken into account.
Predict by opcode approach makes the decision based on the opcade of the branch instruction. The processor assumes that the branch will be taken for certain branch opcodes and not for others. Studies reported in showed that success rate is greater than 75% with the strategy.

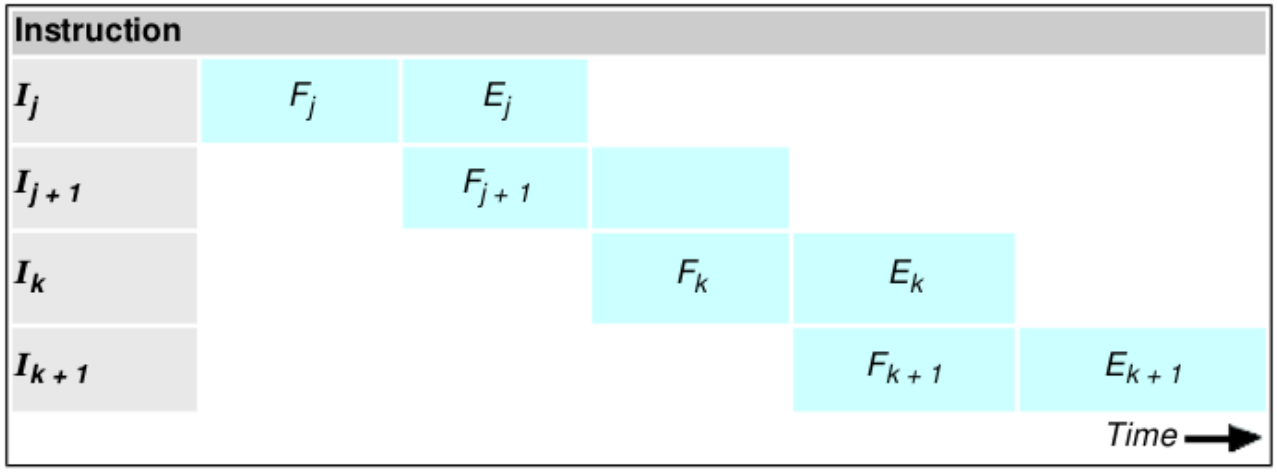
Dynamic branch strategies attempt to improve the accuracy of prediction by recording the history of conditional branch instructions in a program. Scheme to maintain the history information:

- One or more bits can be associated with each conditional branch instruction that reflect the recent history of the instruction.
- These bits are referred to as a taken/not taken switch that directs the processor to make a particular decision the next time the instruction is encountered.
- Generally these history bits are not associated with the instruction in main memory. It will unnecessarily increase the size of the instruction. With a single bit we can record whether the last execution of this instruction resulted a branch or not.
- With only one bit of history, an error in prediction will occur twice for each use of the loop: once for entering the loop. And once for exiting.

If two bits are used, they can be used to record the result of the last two instances of the execution of the associated instruction.

The history information is not kept in main memory, it can be kept in a temporary high speed memory. One possibility is to associate these bits with any conditional branch instruction that is in a cache. When the instruction is replaced in the cache, its history is lost. Another possibility is to maintain a small table for recently executed branch instructions with one or more bits in each entry.
The branch history table is a small cache memory associated with the instruction fetch stage of the pipeline. Each entry in the table consists of three elements:

- The address of the branch instruction.
- Some member of history bits that record the state of use of that instruction.
- Information about the target instruction, it may be the address of the target instruction, or may be the target instruction itself.



Consider that the instruction Ij is a branch instruction. The processor begins fetching instruction Ij+1 before it determine whether the current instruction, Ij , is a branch instruction.
When execution of is completed and a branch must be made, the processor must discard the instruction that was fetched and now fetch the instruction at the branch target.
The location following a branch instruction is called a branch delay slot. There may be more than one branch delay slot, depending on the time it takes to execute a branch instruction.
The instructions in the delay slots are always fetched and at least partially executed before the branch decision is made and the branch target address is computed.
Delayed branching is a technique to minimize the penalty incurred as a result of conditional branch instructions. The instructions in the delay slots are always fetched, so we can arrange the instruction in delay slots to be fully executed whether or not the branch is taken. The objective is to plane useful instruction in these slots. If no useful instructions can be placed in the delay slots, these slots must be filled with NOP (no operation) instructions. While feeling up the delay slots with instructions, it is required to maintain the original semantics of the program.

For example consider the following code segments

I_1	LOOP	Shift_left	R1
I_2		Decrement	R2
I_3		Branch_if $\neq 0$	LOOP
I_4	NEXT	Add	R1,R3

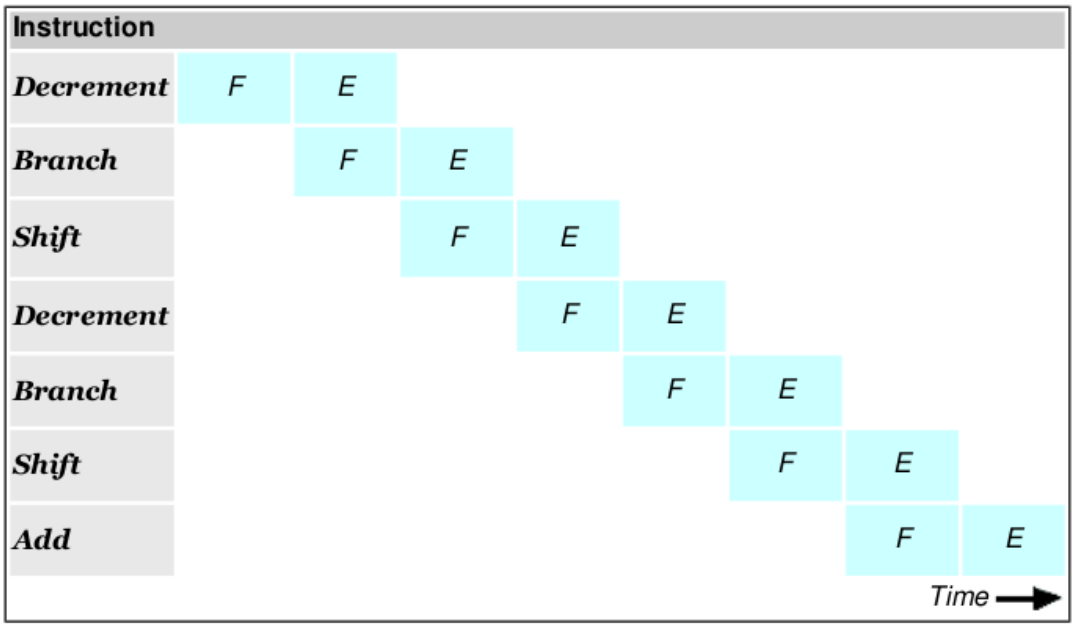
Original Program Loop

Here register R2 is used as a counter to determine the number of times the contents of register R1 are sifted left. Consider a processor with a two-stage pipeline and one delay slot. During the execution phase of the instruction I3 the fetch unit will fetch the instruction I4. After evaluating the branch condition only, it will be clear whether instruction I1 or I4 will be executed next.
The nature of the code segment says that it will remain in the top depending on the initial value of R2 and when it becomes zero, it will come out from the loop and execute the instruction I4. During the loop execution, every time there is a wrong fetch of instruction I4. The code segment can be recognized without disturbing the original meaning of the program.

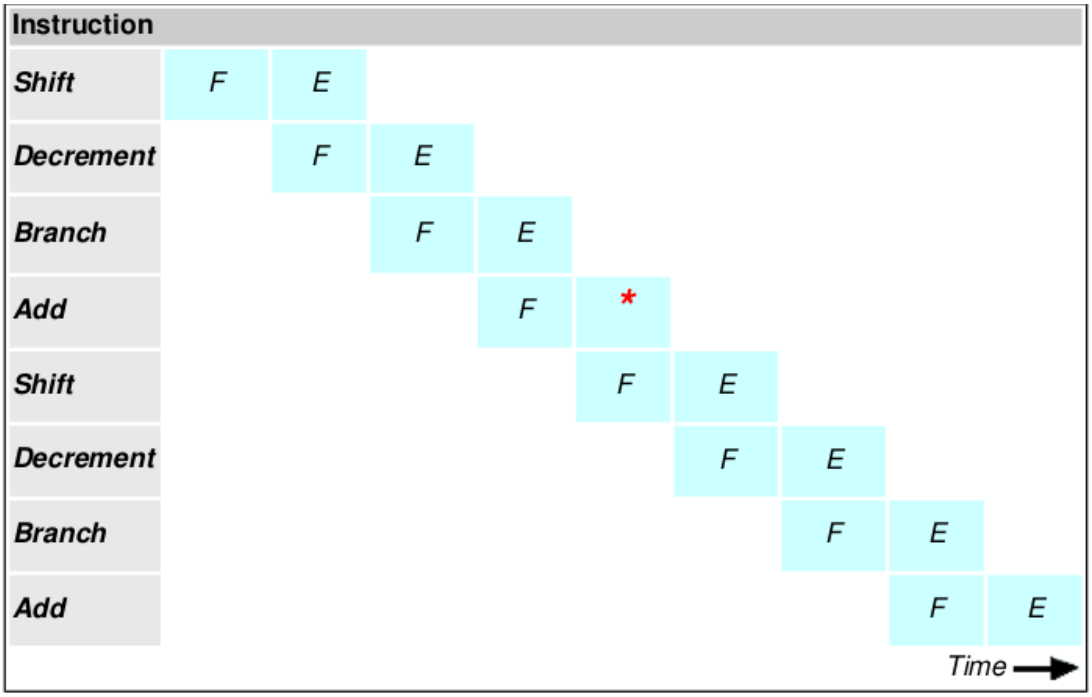
LOOP	*Decrement*	*R2*
	Branch_if ≠0	*LOOP*
	Shift_left	*R1*
NEXT	*Add*	*R1,R3*

Reordered instructions for program loop

In this case, the shift instruction is fetched while the branch instruction is being executed. After evaluating the branch condition, the processor fetches the instruction at LOOP or at NEXT, depending on whether the branch condition is true or false, respectively.
In either case, it completes execution of the shift instruction. Logically the program is executed as if the branch instruction was placed after the shift instruction. That is, branching takes place one instruction later than where the branch instruction appears in the instruction sequence in the memory, hence the name “delayed branch” .



Execution timing for last two passes through the loop of reordered instruction.



Execution timing for last two passes through the loop of the original program loop.
*Note : Execution Unit Idle