

Module 8 : Reduced Instruction Set Programming

In this Module, we have three lectures, viz.

- 1. [Introduction to RISC](#)**
- 2. [Design issues of a RISC](#)**

Click the proper link on the left side for the lectures

Introduction

Since the development of the stored program computer around 1950, there are few innovations in the area of computer organization and architecture. Some of the major developments are:

- **The Family Concept:** Introduced by IBM with its system/360 in 1964 followed by DEC, with its PDP-8. The family concept decouples the architecture of a machine from its implementation. A set of computers are offered, with different price/performance characteristics, that present the same architecture to the user.
- **Microprogrammed Control Unit:** Suggested by Wilkes in 1951, and introduced by IBM on the S/360 line in 1964. Microprogramming eases the task of designing and implementing the control unit and provide support for the family concept.
- **Cache Memory:** First introduced commercially on IBM S/360 Model 85 in 1968. The insertion of this element into the memory hierarchy dramatically improves performance.
- **Pipelining:** A means of introducing parallelism into the essentially sequential nature of a machine instruction program. Examples are instruction pipelining and vector processing.
- **Multiple Processor:** This category covers a number of different organizations and objectives.

One of the most visual forms of evolution associated with computers is that of programming languages. Even more powerful and complex high level programming languages has been developed by the researcher and industry people.

The development of powerful high level programming languages give rise to another problem known as the semantic gap, the difference between the operations provided in HLLs and those provided in computer architecture.

The computer designers intend to reduce this gap and include large instruction set, more addressing mode and various HLL statements implemented in hardware. As a result the instruction set becomes complex. Such complex instruction sets are intended to-

- Ease the task of the compiler writer.
- Improve execution efficiency, because complex sequences of operations can be implemented in microcode.
- Provide support for even more complex and sophisticated HLLs.

To reduce the gap between HLL and the instruction set of computer architecture, the system becomes more and more complex and the resulted system is termed as **Complex Instruction Set Computer (CISC)**.

A number of studies have been done over the years to determine the characteristics and patterns of execution of machine instructions generated from HLL programs. The instruction execution characteristics involves the following aspects of computation:

- **Operation Performed:** These determine the functions to be performed by the processor and its interaction with memory.
- **Operand Used:** The types of operands and the frequency of their use determine the memory organization for storing them and the addressing modes for accessing them.
- **Execution sequencing:** This determines the control and pipeline organization.

Operations:

A variety of studies have been made to analyze the behavior of HLL programs. It is observed that

- Assignment statements predominates, suggesting that the simple movement of data is of high importance.
- There is also a presence of conditional statements (IF, Loop, etc.). These statements are implemented in machine language with some sort of compare and branch instruction. This suggest that the sequence control mechanism of the instruction set is important.

A variety of studies have analyzed the behavior of high level language program. The following table includes key results, measuring the appearance of various statement types during execution which is carried out by different researchers.

Study Language Workload	[HUCK83] Pacal Scientific	[KNUTH71] Fortran Student	[PATT82]		[TANE78] SAL System
			Pascal System	C System	
Assign	74	67	45	38	42
Loop	4	3	5	3	4
Call	1	3	15	12	12
IF	20	11	29	43	36
GOTO	20	9	--	3	--
Other	--	7	6	1	6

Table : Relative Dynamic Frequency of High-Level Language operation

RISC[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

[HUCK83] Huck, T; "Comparative analysis of computer architectures", Stanford University Technical Report No.83-243.

[KNUTH71] Knuth D; "An Empirical Study of FORTRAN programs ", Software practice and Experience, Vol. 1,1971. No.83-243

[PATT82] Patterson, D and Sequin, C; "A VLSI RISC ", Computer, September 1982.

[TANE78] Tanenbaum, A; "Implication of Structured Programming for machine architecture ", Communication of the ACM, March 1978.

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

These results are instructive to the machine instruction set designers, indicating which type of statements occur most often and therefore should be supported in an “optimal” fashion.

From these studies one can observe that though a complex and sophisticated instruction set is available in a machine architecture, common programmer may not use those instructions frequently.

Operands :

Researches also studied the dynamic frequency of occurrence of classes of variables. The results showed that majority of references are single scalar variables. In addition references to arrays/structures required a previous reference to their index or pointer, which again is usually a local scalar. Thus there is a predominance of references to scalars, and these are highly localized.

It is also observed that operation on local variables is performed frequently and it requires a fast accessing of these operands. So, it suggests that a prime candidate for optimization is the mechanism for storing and accessing local scalar variables.

Procedure Call :

The procedure calls and returns are an important aspects of HLL programs. Due to the concept of modular and functional programming, the call/return statements are becoming a predominate factor in HLL program.

It is known fact that call/return is a most time consuming and expensive statements. Because during call we have to restore the current state of the program which includes the contents of local variables that are present in general purpose registers. During return, we have to restore the original state of the program from where we start the procedure call.

Thus, it will be profitable to consider ways of implementing these operations efficiently. Two aspects are significant, the number of parameters and variables that a procedure deals with, and the depth of nesting.

Implications :

A number of groups have looked at these results and have concluded that the attempt to make the instruction set architecture close to HLL is not the most effective design strategy.

Generalizing from the work of a number of researchers three element emerge in the computer architecture.

- **First**, use a large number of registers or use a compiler to optimize register usage. This is intended to optimize operand referencing.
- **Second**, careful attention needs to be paid to the design of instruction pipelines. Because of the high proportion of conditional branch and procedure call instructions, a straight forward instruction pipeline will be inefficient. This manifests itself as a high proportion of instructions that are prefetched but never executed.

- **Third**, a simplified (reduced) instruction set is indicated. It is observed that there is no point to design a complex instruction set which will lead to a complex architecture. Due to the fact, a most interesting and important processor architecture evolves which is termed as Reduced Instruction Set Computer (RISC) architecture.

Although RISC system have been defined and designed in a variety of ways by different groups, the key element shared by most design are these:

- A large number of general purpose registers, or the use of compiler technology to optimize register usage.
- A limited and simple instruction set.
- An emphasis on optimizing the instruction pipeline

An analysis of the RSIC architecture begins into focus many of the important issues in computer organization and architecture.

The table in the next page compares several RISC and non-RISC system.

RISC[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)**Characteristics of some CISCs, RISCs and Supersclar Processors**

	Complex Instruction Set (CISC) Computer			Reduced Instruction Set (RISC) Computer		Superscaler		
Characteristics	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	Power PC	Ultra SPARC	MIPS R10000
Year Developed	1973	1978	1989	1987	1991	1993	1996	1996
Number of Instructions	208	303	235	69	94	225	--	--
Instruction Size (bytes)	2-6	2-57	1-11	4	4	4	4	4
Addressing modes	4	22	11	1	1	2	1	1
Number of general-purpose registers	16	16	8	40-520	32	32	40-520	32
Control Memory size (kbits)	420	480	246	--	--	--	--	--
Cache size (kbits)	64	64	8	32	128	16-32	32	64

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Characteristics of Reduced Instruction Set Architecture :

Although a variety of different approaches to reduce Instruction set architecture have been taken, certain characteristics are common to all of them:

1. One instruction per cycle.
2. Register-to-register operations.
3. Simple addressing modes.
4. Simple instruction formats.

1. One machine instruction per machine cycle :

A machine cycle is defined to be the time it takes to fetch two operands from registers, perform an ALU operation, and store the result in a register.

With simple, one-cycle instructions there is little or no need of microcode, the machine instructions can be hardwired. Hardware implementation of control unit executes faster than the microprogrammed control, because it is not necessary to access a microprogram control store during instruction execution.

2. Register –to– register operations

With register-to-register operation, a simple LOAD and STORE operation is required to access the memory, because most of the operation are register-to-register. Generally we do not have memory-to-memory and mixed register/memory operation.

Simple Addressing Modes

Almost all RISC instructions use simple register addressing. For memory access only, we may include some other addressing, such as displacement and PC-relative. Once the data are fetched inside the CPU, all instruction can be performed with simple register addressing.

Simple Instruction Format

Generally in most of the RISC machine, only one or few formats are used. Instruction length is fixed and aligned on word boundaries. Field locations, especially the opcode, are fixed.

With fixed fields, opcode decoding and register operand accessing can occur simultaneously. Simplified formats simplify the control unit.

The use of a large register file:

For fast execution of instructions, it is desirable of quick access to operands.

There is large proportion of assignment statements in HLL programs, and many of these are of the simple form $A \leftarrow B$. Also there are significant number of operand accesses per HLL Statement.

Also it is observed that most of the accesses are local scalars. To get a fast response, we must have an easy excess to these local scalars, and so the use of register storage is suggested.

Since registers are the fastest available storage devices, faster than both main memory and cache, so the uses of registers are preferable. The register file is physically small, and on the same chip as the ALU and Control Unit. A strategy is needed that will allow the most frequently accessed operands to be kept in registers and to minimize register-memory operations.

Two basic approaches are possible, one is based on software and the other on hardware.

- **The software approach** is to rely on the compiler to maximize register uses. The compiler will attempt to allocate registers to those variables that will be used the most in a given time period.
- **The hardware approach** is simply to use more registers so that more variables can be held in registers for longer period of time.

In hardware approach, it uses the concept of register windows.

Register Window :

The use of a large set of registers should decrease the need to access memory. The design task is to organize the registers in such a way that this goal is realized.

Due to the use of the concept of modular programming, the present day programs are dominated by call/return statements. There are some local variables present in each function or procedure.

1. On every call, local variables must be saved from the registers into memory, so that the registers can be reused by the called program. Furthermore, the parameters must be passed.
2. On return, the variables of the parent program must be restored (loaded back into registers) and results must be passed back to the parent program.
3. There are also some global variables which are used by the module or procedure.

Thus the variables that are used in a program can be categorized as follows :

- **Global variables** : which is visible to all the procedures.
- **Local variables** : which is local to a procedure and it can be accessed inside the procedure only.
- **Passed parameters** : which are passed to a subroutine from the calling program. So, these are visible to both called and calling program.
- **Returned variable** : variable to transfer the results from called program to the calling program. These are also visible to both called and calling program.

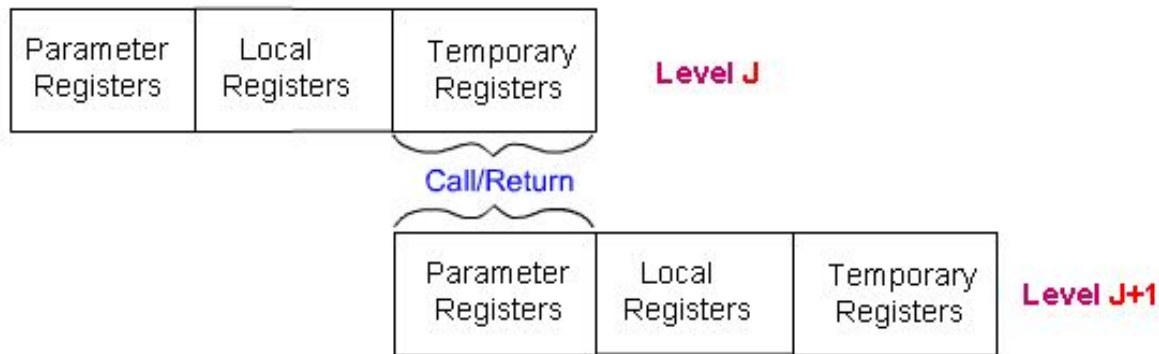
From the studies it is observed that a typical procedure employs only a few passed parameters and local variables. Also the depth of procedure activation remains within a relatively narrow range.

To exploit these properties, multiple small sets of registers are used, each assigned to a different procedure.

A procedure call automatically switches the processor to use a different fixed size window of registers, rather than saving registers in memory.

Windows for adjacent procedures are overlapped to allow parameter passing.

The concept of overlapping register window is shown in the figure.



At any time, only one window of registers is visible which corresponds to the currently executing procedure.

The register window is divided into three fixed-size areas.

- **Parameter registers** hold parameters passed down from the procedure that called the current procedure and hold results to be passed back up.
- **Local registers** are used for local variables.
- **Temporary registers** are used to exchange parameters and results with the next lower level (procedure called by current procedure)

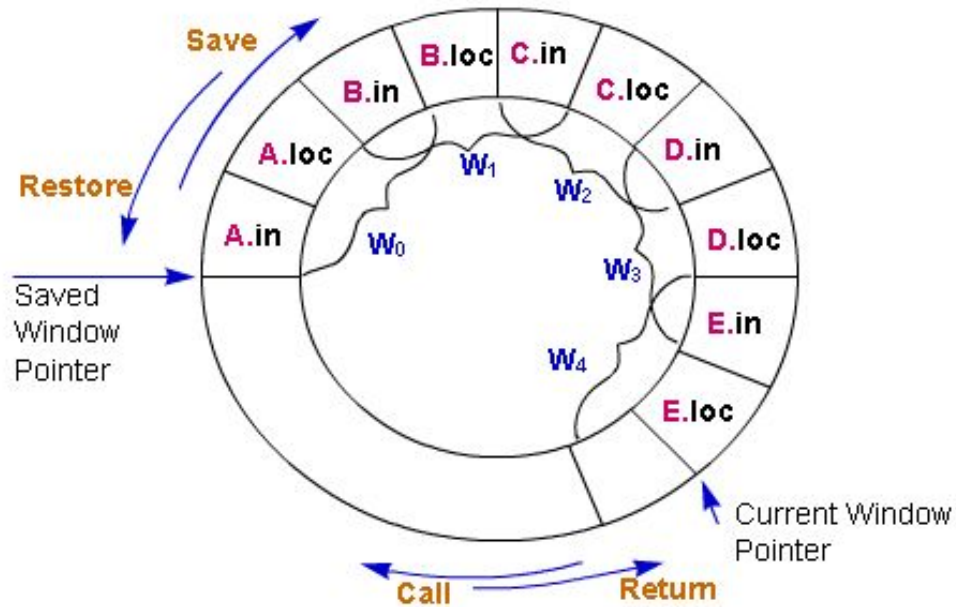
The temporary registers at one level are physically the same as the parameter registers at the next lower level. This overlap permits parameter to be passed without the actual movement of data.

To handle any possible pattern of calls and returns, the number of register windows would have to be unbounded. But we have a limited number of registers, it is not possible to provide unlimited amount of registers.

It is possible to hold the few most recent procedure activation in register windows.

Older activations must be saved in memory and later restored when the nesting depth decreases. It is observed that nesting depth is small in general.

The actual organization of the register file is a circular buffer of overlapping windows. (next..)



The circular buffer of overlapping windows is shown in the figure. The procedure call pattern is : A called B, B called C, C called D and D called E, with procedure E as active process.

The current window pointer (CWP) points to the window of currently active procedure.

The saved window pointer identifies the window most recently saved in memory.

As the nesting depth of procedure calls increases, there may not be sufficient register to accommodate the new procedure. In this case, the information of oldest procedure is stored back into memory and the saved window pointer keep tracks of the most recently saved window.

It is clear that N-Window register file can hold only N-1 procedure activations. The value of N need not be very large, because in general, the depth of procedure activation is small. In case of recursive call the depth of procedure call may increase. From survey, it is found that with 8 windows, a save or restore is needed on only 1% of the calls or returns.

Global Variables

The window scheme provides an efficient organization for storing local scalar variables in registers. Global variables are accessed by more than one procedure.

Two solutions to access the global variables:

- a. Variables declared as global in an HLL can be assigned memory location by the compiler, and all machine instructions that reference these variables will use memory reference operands. This scheme is inefficient for frequently accessed global variables.
- b. An alternative is to incorporate a set of global registers in the processor. These registers would be fixed in number and available to all procedures. In this case, the compiler must decide which global variables should be assigned to registers.

Compiler based Register Optimization

A small number of registers (e.g. 16-32) is available on the target RISC machine and the concept of registers window can not be used. In this case, optimized register usage is the responsibility of the compiler.

A program written in a high level language has no explicit references to registers. The objective of the compiler is to keep the operands for as many computations as possible in registers rather than main memory, and to minimize load and store operations.

To optimize the use of registers, the approach taken is as follows:

- Each program quantity that is a candidate for residing in a register is assigned to a symbolic or virtual register.
- The compiler then maps the unlimited number of symbolic registers into a fixed number of real registers.
- Symbolic registers whose usage does not overlap can share the same real register.
- If in a particular portion of the program, there are more quantities to deal with than real registers, then some of the quantities are assigned to the memory location.

The task of optimization is to decide which quantities are to be assigned to registers at any given point of time in the program. The technique most commonly used in RISC compiler is known as **graph coloring**.

The graph coloring problem is as follows:

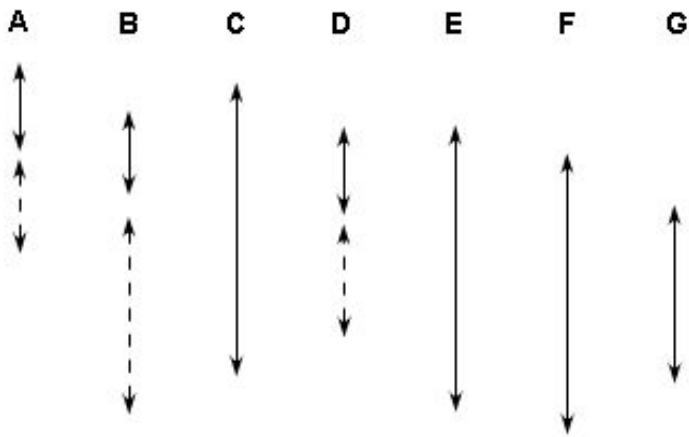
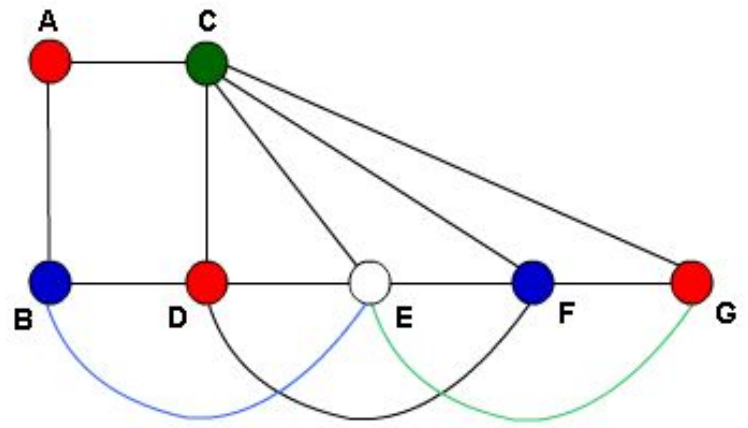
Given a graph consisting of nodes and edges, assign colors to nodes such that adjacent nodes have different colors, and do this in such a way as to minimize the number of different colors.

This graph coloring problem is mapped to the register optimization problem of the compiler in the following way:

- The program is analyzed to build a register interference graph.
- The nodes of the graph are the symbolic registers.
- If two symbolic registers are “live” during the same program fragment, then they are joined by an edge to indicate interference.
- An attempt is then made to color the graph with n colors, where n is the number of register.
- Nodes that cannot be colored are placed in memory.
- Load and store must be used to make space for the affected quantities when they are needed.

Following figures shows a simple example of a process. [\(next page..\)](#)

Assume a program with seven symbolic registers to be compiled in three actual registers. Part ‘a’ of the figure shows the register interference graph. A possible coloring with three colors is shown. Only, a symbolic register E is left uncolored and must be dealt with load and store.

**(a)** Time sequence of active use of registers**(b)** Register interference graph: Graph colouring approach

Large Register file versus cache

The Register file, organized into windows, acts as a small, fast buffer for holding a subset of all variables that are likely to be used the most heavily. From this point of view, the register file acts much like a cache memory.

The question therefore arises as to whether it would be simpler and better to use a cache and a small traditional register file instead of using a large register file.

The following table compares the characteristics of the two approaches

	Large Register File	Cache
1.	All local scalars	Recently used local scalars.
2.	Individual variables	Blocks of memory.
3.	Compiler-assigned global variables	Recently used global variables.
4.	Save/Restore based on procedure nesting depth	Save/Restore based on cache replacement algorithm.
5.	Register addressing	Memory addressing.

Module 09 : Pipeline

In this Module, we have three lectures, viz.

1. [Introduction to Pipeline Processor](#)
2. [Performance Issues](#)
3. [Branching](#)

Click the proper link on the left side for the lectures

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 accumulator, and use of cache memory improves the performance considerably. Another organizational approach, which is quite common, is instruction pipelining.

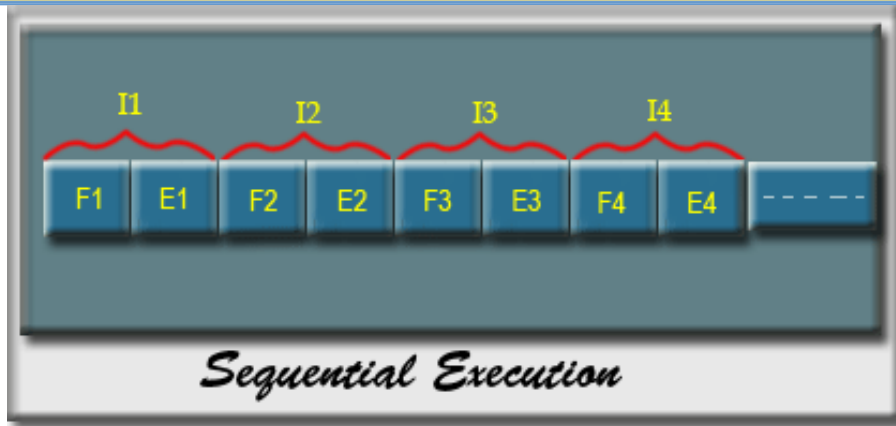
Pipelining is a particularly effective way of organizing parallel activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly line operation.

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 F_i and E_i refer to the fetch and execute steps for instruction I_i . 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 B_1

. The results of execution are stored in the destination location specified by the instruction.

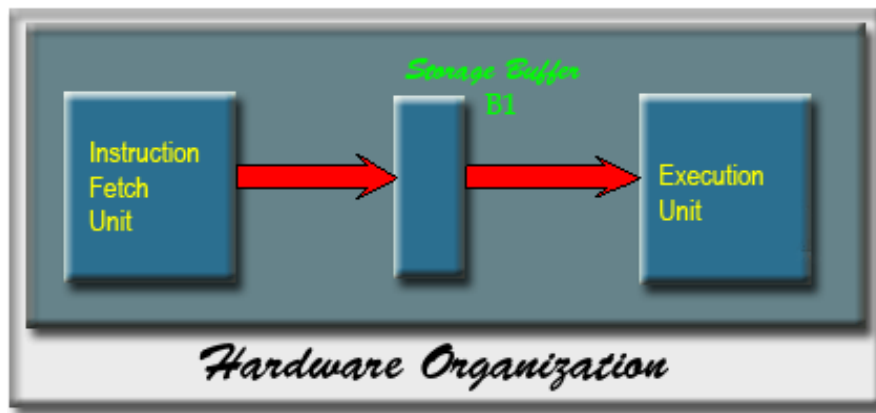
For simplicity it is assumed that fetch and execute steps of any instruction can be completed in one clock cycle.

The operation of the computer proceeds as follows:

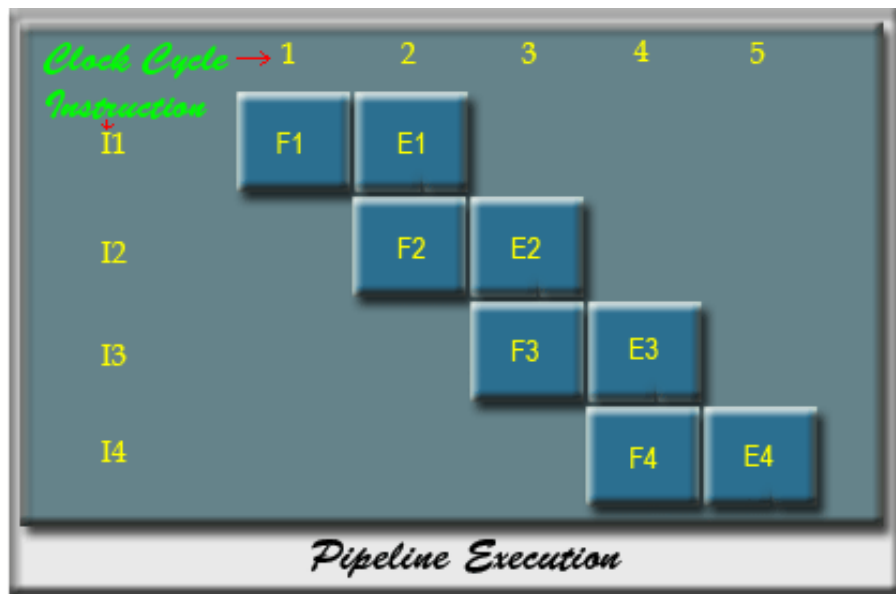
- In the first clock cycle, the fetch unit fetches an instruction (instruction I_1 , step F_1) and stored it in buffer B_1 at the end of the clock cycle.
- In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I_2 (step F_2).
- Meanwhile, the execution unit performs the operation specified by instruction I_1 which 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.



Another picture is shown on the next slide for instruction pipelining.



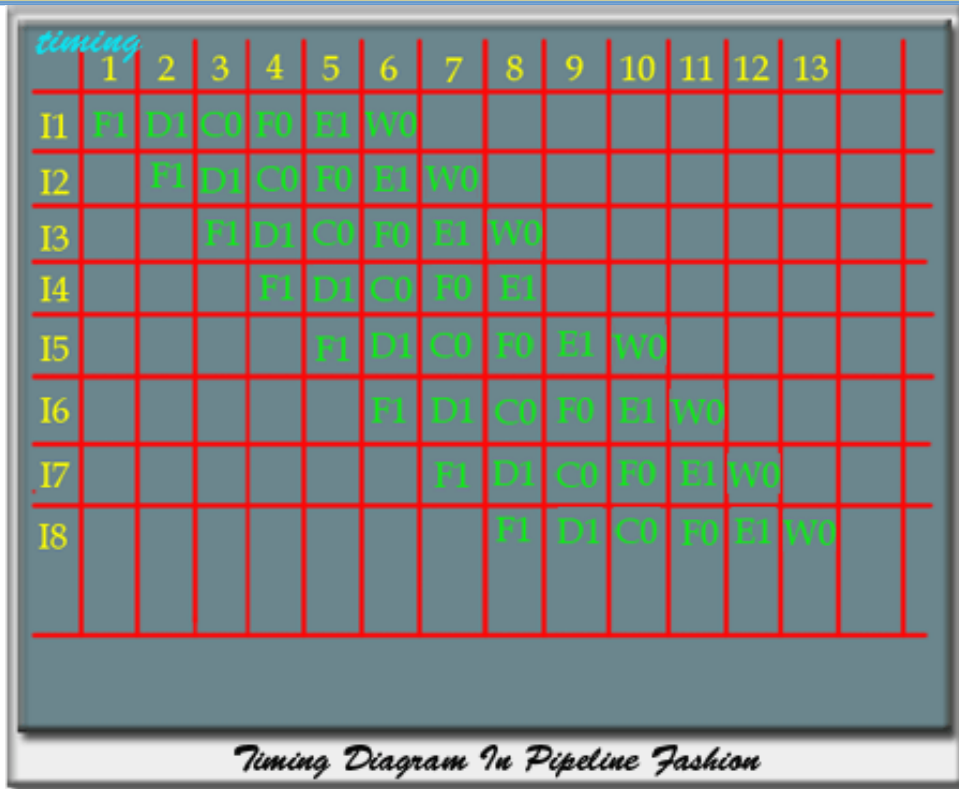
The processing of an instruction need not be divided into only two steps. To gain further speed up, the pipeline must have more stages.

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. If 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.



Pipelining[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

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.

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

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

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

where

τ_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 T_k to execute all n instructions is

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

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

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

In general, the number of instruction executed is much more higher than the number of stages in the pipeline So, the n tends to ∞ , 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;

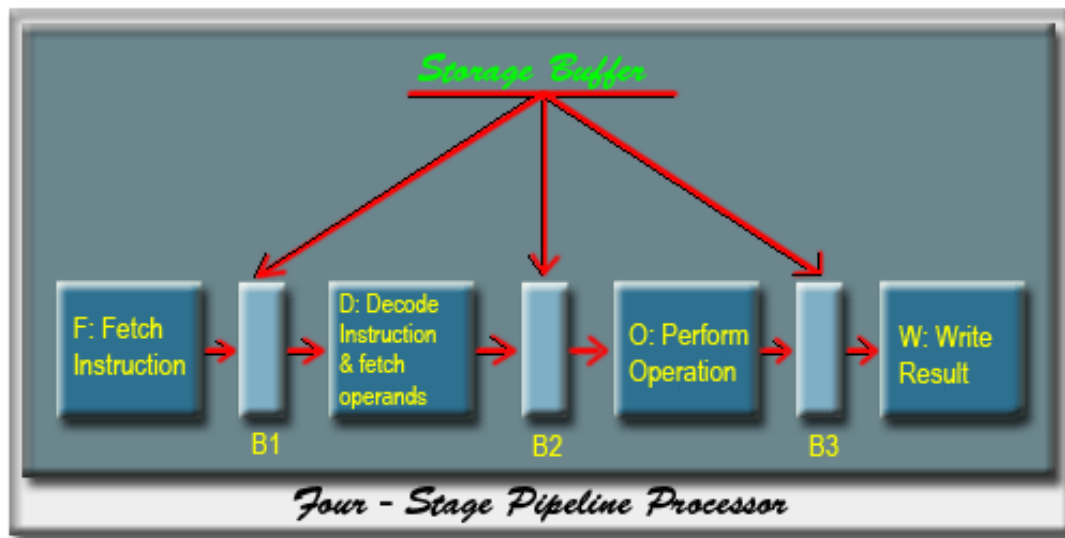
F: Fetch, Read the instruction from the memory

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 in the figure on the next slide.



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 it 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.

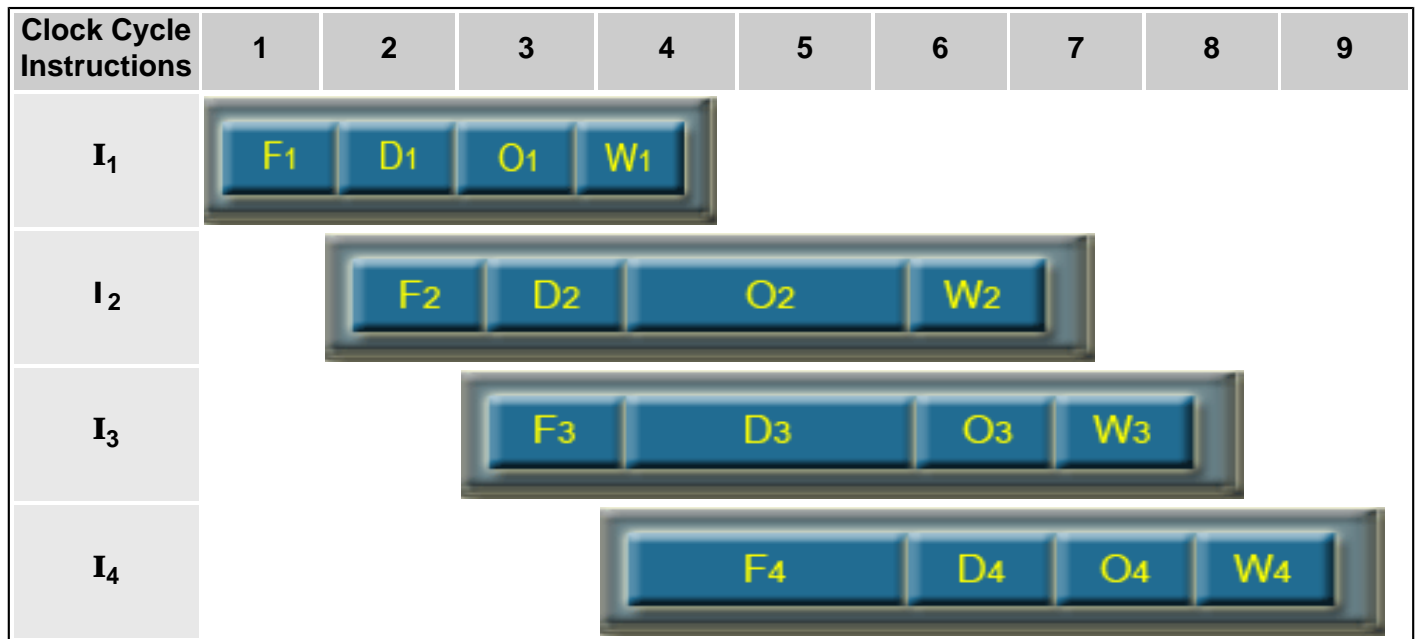
(A) Unequal time requirement to complete a subtask:

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 intact 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.

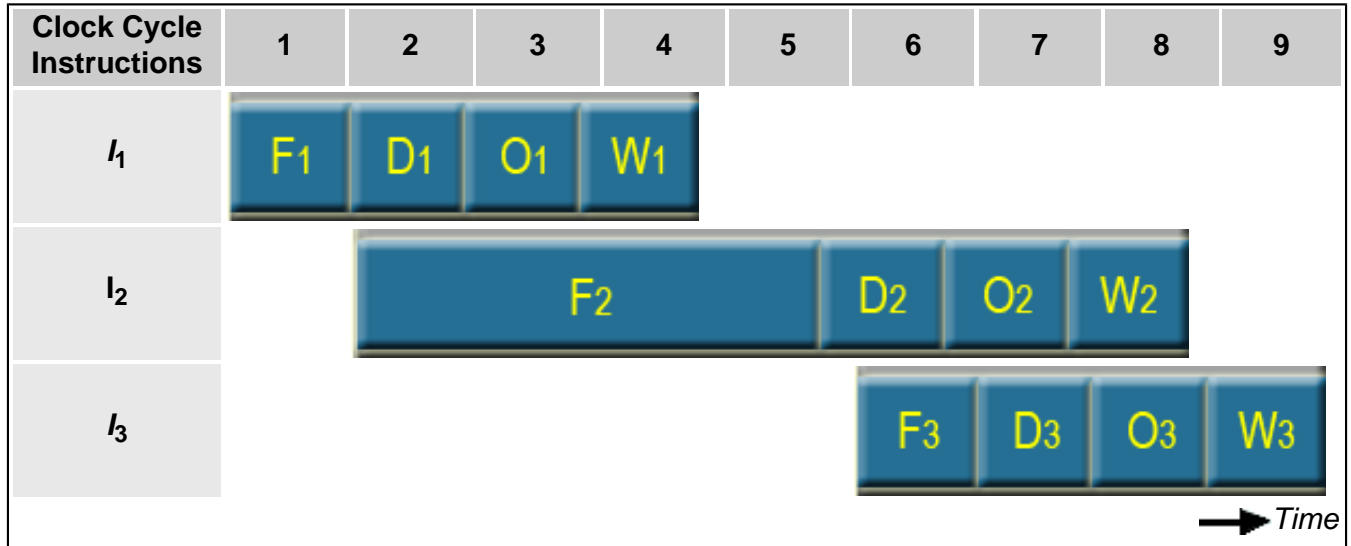
Due to that reason, pipeline operation is said to have been stalled for two clock cycle. Normal pipeline operation resumes in clock cycle 7.

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 I_2 which starts in cycle 2, results in a cache miss.

The instruction fetch unit must now suspend any further fetch requests and wait for I_2 to arrive.

We assume that instruction I_2 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

$$I_1 : A \leftarrow A + 5$$

$$I_2 : B \leftarrow 3 * A$$

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 I_2 will get 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

$B=27$

$$I_1 : A \leftarrow A + 5 = 4 + 5 = 9$$

$$I_2 : B \leftarrow 3 \times A = 3 \times 9 = 27$$

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

$$I_1 : A \leftarrow A + 5 = 4 + 5 = 9$$

$$I_2 : B \leftarrow 3 \times A = 3 \times 4 = 12$$

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 :[Print this page](#)<< Previous | First | [Last](#) | Next >>

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	1	2	3	4	5	6	7	8	9	10
Instruction										
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

<< Previous | First | [Last](#) | Next >>

Branching :[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

In this instruction sequence, consider that I_3 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 I_4 and I_5

If the branch condition is false, then branch won't take place and the next instruction to be executed is I_4 which is already fetched and available for execution.

Now consider that when the condition is true, we have to execute the instruction I_{10} . 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 I_4 and I_5 . It is required to invalidate these two fetched instructions 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 to as branch penalty.

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Branching :

[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

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}

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Branching :[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

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 I_4 and I_5 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.

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Branching :[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)**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 branch 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

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

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 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,.

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 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. 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.

Branching :[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

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.

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Branching :[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

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 branch 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 [LILJ88], 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 opcode of the branch instruction. The processor assumes that the branch will be taken for certain branch opcodes and not for others. Studies reported in [LILJ88] showed that success rate is greater than 75% with the strategy.

[LILJ88] Lilja, D "Reducing the branch penalty in pipeline processors", computer, July 1988.

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Branching :[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

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.

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Branching :[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

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:

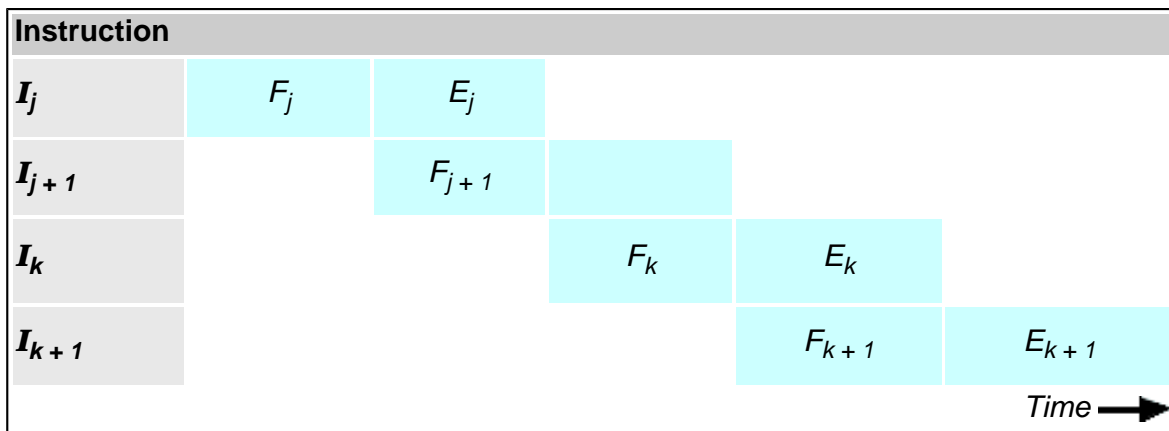
- The address of the branch instruction.
- Some member of history bits that record the state of use of that instruction.
- Information about the target instruction, it may be the address of the target instruction, or may be the target instruction itself.

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Branching :

[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Delayed Branch:

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Branching :[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Consider that the instruction I_j is a branch instruction. The processor begins fetching instruction I_{j+1} before it determine whether the current instruction, I_j , is a branch instruction.

When execution of I_j 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 place 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 filling up the delay slots with instructions, it is required to maintain the original semantics of the program.

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Branching :

[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

For example consider the consider the following code segments:

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

Original Program Loop

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Branching :

[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Here register R_2 is used as a counter to determine the number of times the contents of register R_1 are sifted left.

Consider a processor with a two-stage pipeline and one delay slot. During the execution phase of the instruction I_3 , the fetch unit will fetch the instruction I_4 . After evaluating the branch condition only, it will be clear whether instruction I_1 or I_4 will be executed next.

The nature of the code segment says that it will remain in the top depending on the initial value of R_2 and when it becomes zero, it will come out from the loop and execute the instruction I_4 . During the loop execution, every time there is a wrong fetch of instruction I_4 . The code segment can be recognized without disturbing the original meaning of the program

<i>LOOP</i>	<i>Decrement</i>	<i>R2</i>
	<i>Branch_if $\neq 0$</i>	<i>LOOP</i>
	<i>Shift_left</i>	<i>R1</i>
<i>NEXT</i>	<i>Add</i>	<i>R1,R3</i>

Reordered instructions for program loop

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

Branching :[Print this page](#)[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)

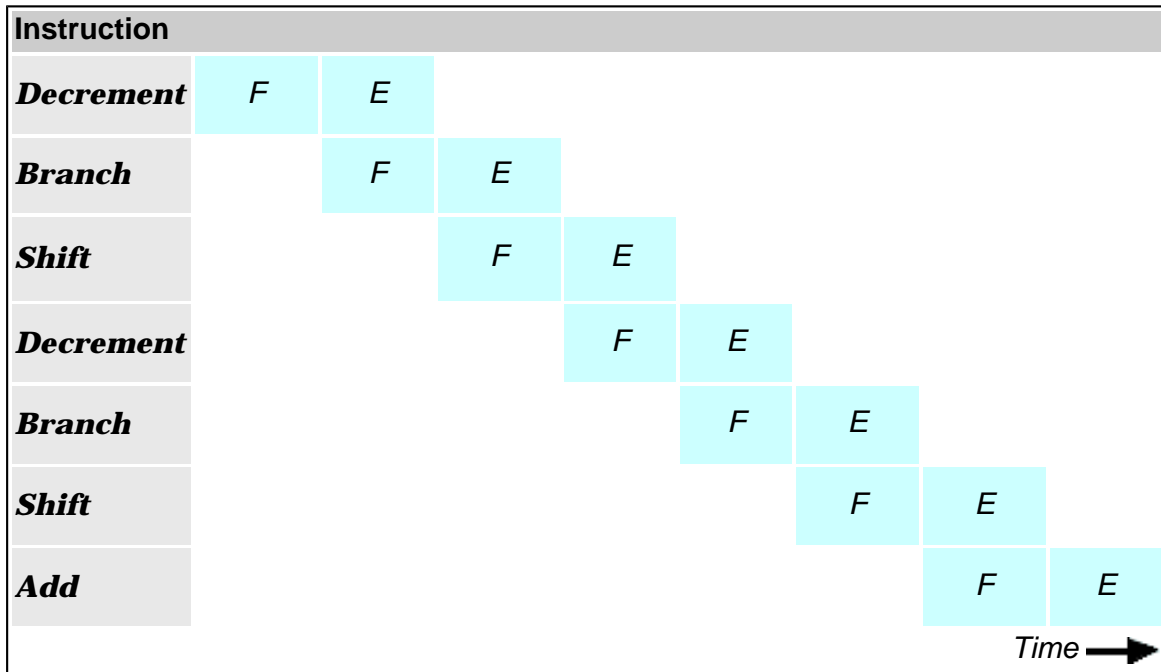
In this case, the shift instruction is fetched while the branch instruction is being executed.

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.

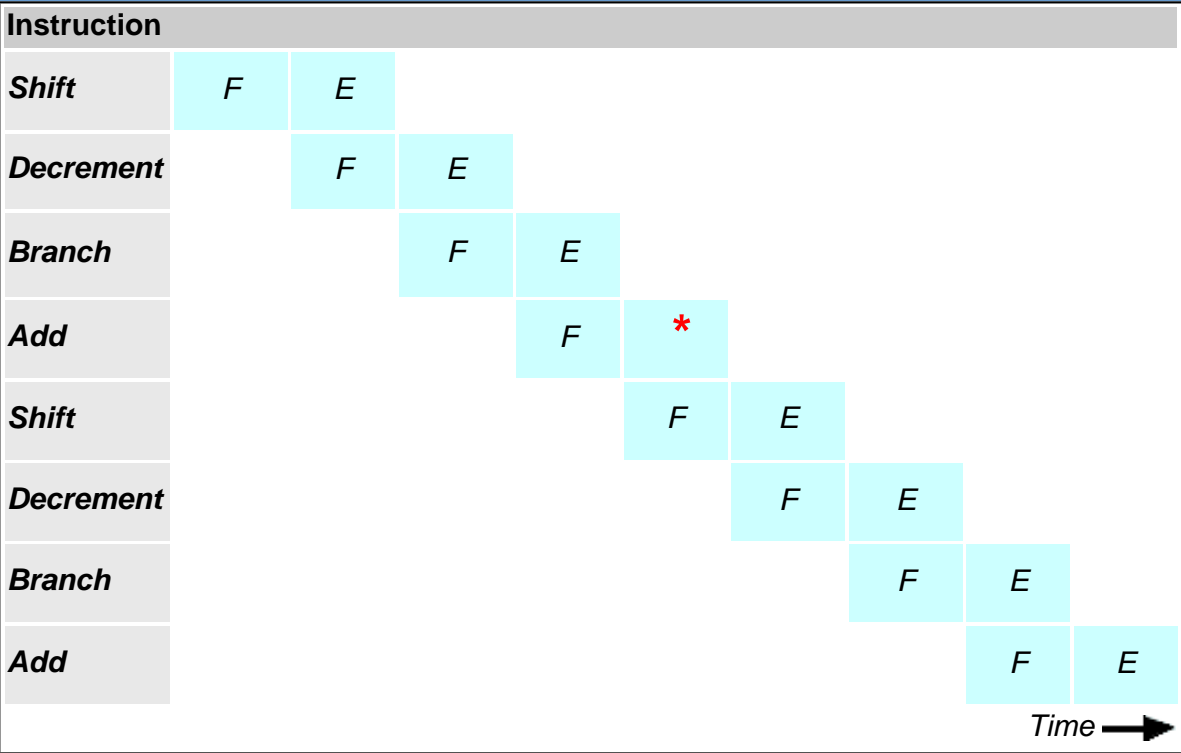
In either case, it completes execution of the shift instruction.

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” .

[<< Previous](#) | [First](#) | [Last](#) | [Next >>](#)



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

Module 10 : Multi-Processor / Parallel Processing

In this Module, we have three lectures, viz.

1. [Introduction to Parallel Processing](#)
2. [Introduction to Network](#)
3. [Cache Coherence](#)

Click the proper link on the left side for the lectures

Parallel Processing :

Originally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequence of instruction.

Processor executes programs by executing machine instructions in a sequence and one at a time.

Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation store result.)

It is observed that, at the micro operation level, multiple control signals are generated at the same time.

Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for long time.

By looking into these phenomenons, researcher has look into the matter whether some operations can be performed in parallel or not.

As computer technology has evolved, and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usual to enhance performance and, in some cases, to increase availability.

The taxonomy first introduced by Flynn is still the most common way of categorizing systems with parallel processing capability. Flynn proposed the following categories of computer system:

- **Single Instruction, Multiple Data (SIMD) system:** A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data memory, so that each instruction is executed on a different set of data by the different processors. Vector and array processors fall into this category
- **Multiple Instruction, Single Data (MISD) system** A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure has never been implemented.
- **Multiple Instruction, Multiple Data (MIMD) system:** A set of processors simultaneously execute different instruction sequences on different data sets. SMPs, clusters, and NUMA systems fits into this category.

With the MIMD organization, the processors are general purpose; each is able to process all of the instructions necessary to perform the appropriate data transformation.

Further MIMD can be subdivided into two main categories:

- **Symmetric multiprocessor (SMP):** In an SMP, multiple processors share a single memory or a pool of memory by means of a shared bus or other interconnection mechanism. A distinguish feature is that the memory access time to any region of memory is approximately the same for each processor.

- Nonuniform memory access (NUMA): The memory access time to different regions of memory may differ for a NUMA processor.

The design issues relating to SMPs and NUMA are complex, involving issues relating to physical organization, interconnection structures, inter processor communication, operating system design, and application software techniques.