Computer Organization and Architecture Print this page

Web Course Developed for NPTEL
Computer Organization and Architecture

Objective:

This web course is divided into 11 Modules, which is further divided into Lectures.

Again each lectures contains along with the topic some additional resourses, references etc.

For the detail syllabus of the course, **click here..**.

Course Developer

Prof. J. K. Deka

Prof. Jatindra Kumar deka
is working as Assistant Professor in
Department of Computer Science and Engineering, IIT Guwahati

Contact Detail :

Phone(O): +91 (361) 2582354
Phone(R): +91 (361) 2584354
Fax : +91 (361) 2690762
Email-id : jatin@iitg.ernet.in

Academic Profile:

- B. E., Motilal Nehru Regional Engineering College, Allahabad.
- M. Tech., Indian Institute of Technology, Kharagpur.
- Ph. D., Indian Institute of Technology, Kharagpur.

Outline of the Course (COA)

Print this page

<< Previous | First | Last | Next >>

Outline of the Course :

The study materials provided in this web course is intended for the first level course on Computer Organization and Architecture. It can be used as a supplementary study materials for undergraduate course of Universities/Institutions in India. This web course will help the B. Tech./B.E. students for their course on Computer Organization and Architecture. It is also useful for the courses like BCA, MCA, B. Sc. (Computer Science/Information Technology), where Computer Organization and Architecture is taught as a compulsory subject.

The students who study Computer Organization and Architecture, generally study the introductory course on Digital Systems. The students should have some knowledge on Digital Logic Circuit Design course to go through this study materials.

Student should have also some preliminary idea about computer programming (in high level language), which will help them to understand how to program a computer to solve a problem; and how the program is executed in the computer.

<< Previous | First | Last | Next >>

While describing a Computer, the terms Organization and Architecture generally come together. Though a distinction is often made between Computer Organization and Architecture, it is difficult to give precise definition for these terms.

Computer Architecture refers to those attributes of a system visible to a programmer. Computer Organization refers to the operational units and their interconnections that realize the architectural specifications. As an example, it is an architectural design issue whether a computer will have a multiply instruction. It is an organizational issue whether that instruction will be implemented by a special multiply unit or by the method of repeated addition
by using the add unit of the system.

Though these concepts help us to get some idea about Organization and Architecture, in this study materials, no specific distinction has been made between organization and architecture.

The study materials available in this web course should not be treated as a replacement to text books, rather it should be treated as a lecture notes prepared with the help of some text books.

The main text books used for preparing these lecture notes are:

1. **Computer Organization and Architecture: Designing for Performance**
   Authors: William Stallings
   Publisher: Prentice-Hall India

2. **Computer Organization**
   Authors: Carl Hamacher, Zvonko Vranesic and Safwat Zaky
   Publisher: McGraw Hill

Other reference books :

1. **Computer Architecture A Quantitative Approach**
   Authors: John L Hennessy and David Patterson
   Publisher: Morgan Kaufman

2. **Structured Computer Organization**
   Authors: Andrew S. Tanenbaum
   Publisher: Prentice-Hall India

3. **Computer Organization and Design**
   Authors: P. Paul Choudhury
   Publisher: Prentice-Hall India

Outline of the Course (COA)

The course is subdivided into several modules.

Module 1 is Introduction.

In the first lecture of this module, I have started with a very tiny hypothetical computer through which I tried to introduced most of the terms that are used in computer organization and architecture, and the working principle of a computer. Also I tried to explain the concept of execution of a program in this computer. In this course material, the term computer always means digital computer, and these concepts are not related to analog computer.

How the information is stored in digital computer is explained next. It also contains some historical information regarding the evaluation of first generation computer and the changes of technologies to achieve the computer of current generation.

In **Module 2**, the implementation issues of some of the operations like addition, multiplication, etc. are explained.

Those who are familiar with the implementation issues of Arithmetic and Logic operations, they may skip Module 2.

Memory or storage unit is an important unit in digital computer. In **Module 3**, I have explained the memory module. First, the basic memory module is introduced along with its working principle. Then I moved to advanced features like Cache memory, virtual memory, etc. Some issues of memory management is also included in this module.

In **Module 4**, I have explained the architectural issues, like different addressing mode, machine instruction and instruction format. The concepts of addressing modes and machine instruction formats are explained in general. For particular machine organization, one may look into Module 11 where the instruction format of 8085 and 8086 are explained in brief.

In **Module 5**, the design issues of Central processor unit is explained. Both Hardwired-controlled and microprogrammed-controlled control unit design are explained. I have explained the concept in general, without taking any specific architecture or organization.

In **Module 6**, concept of Input/Output mechanism is explained. The issues related to interrupt and interrupt handling mechanism has been taken care in this module. The concept of DMA is also explained in this module.

In **Module 7**, the connection of Input/Output devices are explained and the concept of I/O buses are also explained. The organization of Hard Disk, the external storage device is also briefly explained in this module.

The concept of Reduced Instruction Set Computer (RISC) is explained in **Module 8**.

The advanced issue, concept of pipeline is introduced in **Module 9** and explained the design issues.

Module 10 talks about the advanced features like multi-processor and parallel processing.

In **Module 11**, I have taken two case studies: Intel 8085 and Intel 8086 microprocessor. I tried to give the concept of instruction set design with the help of these two processors. Also, it includes the concept of assembly level programming and machine level programming.

Computer Organization and Architecture Print this page

Module 1 : Introduction

In this Module, we have three lectures, viz.

1. **Introduction to computer System and its submodules**
2. **Number System and Representation of information.**
3. **Brief History of Comp. Evolution**

Click the proper link on the left side for the lectures

Representation of Basic Information

The basic functional units of computer are made of electronics circuit and it works with electrical signal. We provide input to the computer in form of electrical signal and get the output in form of electrical signal.

There are two basic types of electrical signals, namely, **analog** and **digital**. The analog signals are continuous in nature and digital signals are discrete in nature.

The electronic device that works with continuous signals is known as **analog device** and the electronic device that works with discrete signals is known as **digital device**. In present days most of the computers are digital in nature and we will deal with Digital Computer in this course.

Computer is a digital device, which works on two levels of signal. We say these two levels of signal as **High** and **Low**. The High-level signal basically corresponds to some high-level signal (say 5 Volt or 12 Volt) and Low-level signal basically corresponds to Low-level signal (say 0 Volt). This is one convention, which is known as positive logic. There are others convention also like negative logic.

Since Computer is a digital electronic device, we have to deal with two kinds of electrical signals. But while designing a new computer system or understanding the working principle of computer, it is always difficult to write or work with 0V or 5V. To make it convenient for understanding, we use some logical value, say,

LOW (L) - will represent 0V and
HIGH (H) - will represent 5V

Introduction to Computer System

Computer is used to solve mainly numerical problems. Again it is not convenient to work with symbolic representation. For that purpose we move to numeric representation. In this convention, we use 0 to represent **LOW** and 1 to represent **HIGH**.

0 means LOW
1 means HIGH

To know about the working principle of computer, we use two numeric symbols only namely 0 and 1. All the functionalities of computer can be captured with 0 and 1 and its theoretical background corresponds to two valued boolean algebra.

With the symbol 0 and 1, we have a mathematical system, which is knows as **binary number system**. Basically binary number system is used to represent the information and manipulation of information in computer. This information is basically strings of 0s and 1s.

The smallest unit of information that is represented in computer is known as Bit (Binary Digit), which is either 0 or 1. Four bits together is known as **Nibble**, and Eight bits together is known as **Byte**.

Introduction :: Computer Organization and Architecture

Computer technology has made incredible improvement in the past half century. In the early part of computer evolution, there were no stored-program computer, the computational power was less and on the top of it the size of the computer was a very huge one.

Today, a personal computer has more computational power, more main memory,more disk storage, smaller in size and it is available in effordable cost.

This rapid rate of improvement has come both from advances in the technology used to build computers and from innovation in computer design. In this course we will mainly deal with the innovation in computer design.

The task that the computer designer handles is a complex one: Determine what attributes are important for a new machine, then design a machine to maximize performance while staying within cost constraints.

This task has many aspects, including instruction set design, functional organization, logic design, and imple mentation.

While looking for the task for computer design, both the terms computer organization and computer architecture come into picture.

Introduction to Computer System

It is difficult to give precise definition for the terms Computer Organization and Computer Architecture. But while describing computer system, we come across these terms, and in literature, computer scientists try to make a distinction between these two terms.

Computer architecture refers to those parameters of a computer system that are visible to a programmer or those parameters that have a direct impact on the logical execution of a program. Examples of architectural attributes include the instruction set, the number of bits used to represent different data types, I/O mechanisms, and techniques for addressing memory.

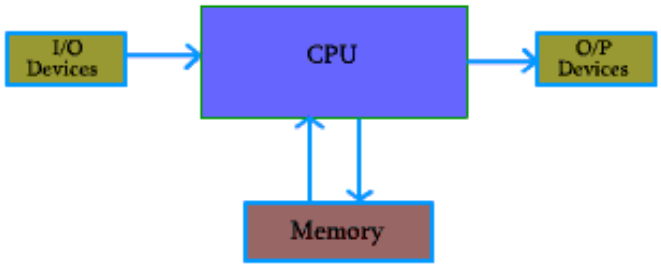
Computer organization refers to the operational units and their interconnections that realize the architectural specifications. Examples of organizational attributes include those hardware details transparent to the programmer, such as control signals, interfaces between the computer and peripherals, and the memory technology used.

In this course we will touch upon all those factors and finally come up with the concept how these attributes contribute to build a complete computer system.

Basic Computer Model and different units of Computer

The model of a computer can be described by four basic units in high level abstraction. These basic units are:

- Central Processor Unit
- Input Unit
- Output Unit
- Memory Unit



Basic Computer Model and different units of Computer

A. Central Processor Unit [CPU] :

Central processor unit consists of two basic blocks :

- The program control unit has a set of registers and control circuit to generate control signals.
- The execution unit or data processing unit contains a set of registers for storing data and an Arithmatic and Logic Unit (ALU) for execution of arithmatic and logical operations.

In addition, CPU may have some additional registers for temporary storage of data.

B. Input Unit :

With the help of input unit data from outside can be supplied to the computer. Program or data is read into main storage from input device or secondary storage under the control of CPU input instruction.

Example of input devices: Keyboard, Mouse, Hard disk, Floppy disk, CD-ROM drive etc.

C. Output Unit :

With the help of output unit computer results can be provided to the user or it can be stored in stograge device permanently for future use. Output data from main storage go to output device under the control of CPU output instructions.

Example of output devices: Printer, Monitor, Plotter, Hard Disk, Floppy Disk etc.

D. Memory Unit :

Memory unit is used to store the data and program. CPU can work with the information stored in memory unit. This memory unit is termed as primary memory or main memory module. These are basically semi conductor memories.

There ate two types of semiconductor memories -

- **Volatile Memory** : RAM (Random Access Memory).
- **Non-Volatile Memory** : ROM (Read only Memory), PROM (Programmable ROM) EPROM (Erasable PROM), EEPROM (Electrically Erasable PROM).

Secondary Memory :

There is another kind of storage device, apart from primary or main memory, which is known as secondary memory. **Secondary memories are non volatile memory** and it is used for permanent storage of data and program.

Example of secondary memories:

Hard Disk, Floppy Disk, Magenetic Tape	------	These are magnetic devices,
CD-ROM	------	is optical device
Thumb drive (or pen drive)	------	is semiconductor memory.

Basic Working Principle of a Computer

Before going into the details of working principle of a computer, we will analyse how computer works with the help of a small hypothetical computer.

In this small computer, we do not consider about Input and Output unit. We will consider only CPU and memory module. Assume that somehow we have stored the program and data into main memory. We will see how CPU can perform the job depending on the program stored in main memory.

P.S. - Our assumption is that students understand common terms like program, CPU, memory etc. without knowing the exact details.

Consider the Arithmatic and Logic Unit (ALU) of Central Processing Unit :

Consider an ALU which can perform four arithmatic operations and four logical operations

To distingish between arithmatic and logical operation, we may use a signal line,

0 - in that signal, represents an arithmatic operation and
1 - in that signal, represents a logical operation.

In the similar manner, we need another two signal lines to distinguish between four arithmatic operations.

The different operations and their binary code is as follows:

Arithmatic		Logical	
000	ADD	100	OR
001	SUB	101	AND
010	MULT	110	NAND
011	DIV	111	ADD

Consider the part of control unit, its task is to generate the appropriate signal at right moment.

There is an instruction decoder in CPU which decodes this information in such a way that computer can perform the desired task

The simple model for the decoder may be considered that there is three input lines to the decoder and correspondingly it generates eight output lines. Depending on input combination only one of the output signals will be generated and it is used to indicate the corresponding operation of ALU.

In our simple model, we use three storage units in CPU,

Two -- for storing the operand and

one -- for storing the results.

These storage units are known as register.

But in computer, we need more storage space for proper functioning of the Computer.

Some of them are inside CPU, which are known as register. Other bigger junk of storage space is known as primary memory or main memory. **The CPU can work with the information available in main memory only**.

To access the data from memory, we need two special registers one is known as **Memory Data Register (MDR)** and the second one is **Memory Address Register (MAR)**.

Data and program is stored in main memory. While executing a program, CPU brings instruction and data from main memory, performs the tasks as per the instuction fetch from the memory. After completion of operation, CPU stores the result back into the memory.

In next section, we discus about memory organization for our small machine.

Main Memory Organization

Main memory unit is the storage unit, There are several location for storing information in the main memory module.

The capacity of a memory module is specified by the number of memory location and the information stored in each location.

A memory module of capacity 16 X 4 indicates that, there are 16 location in the memory module and in each location, we can store 4 bit of information.

We have to know how to indicate or point to a specific memory location. This is done by address of the memory location.

We need two operation to work with memory.

READ Operation: This operation is to retrive the data from memory and bring it to CPU register
WRITE Operation: This operation is to store the data to a memory location from CPU register

We need some mechanism to distinguish these two operations READ and WRITE.

With the help of one signal line, we can differentiate these two operations. If the content of this signal line is

0, we say that we will do a READ operation; and if it is

1, then it is a WRITE operation.

To transfer the data from CPU to memory module and vice-versa, we need some connection. This is termed as **DATA BUS**.

The size of the data bus indicate how many bit we can transfer at a time. Size of data bus is mainly specified by the data storage capacity of each location of memory module.

We have to resolve the issues how to specify a particular memory location where we want to store our data or from where we want to retrive the data.

This can be done by the memory address. Each location can be specified with the help of a binary address.

If we use 4 signal lines, we have 16 different combinations in these four lines, provided we use two signal values only (say 0 and 1).

To distingush 16 location, we need four signal lines. These signal lines use to identify a memory location is termed as **ADDRESS BUS**. Size of address bus depends on the memory size. For a memory module of capacity of 2^n location, we need n address lines, that is, an address bus of size n.

We use a address decoder to decode the address that are present in address bus

As for example, consider a memory module of 16 location and each location can store 4 bit of information
The size of address bus is 4 bit and the size of the data bus is 4 bit
The size of address decoder is 4 X 16.

There is a control signal named R/W.
If R/W = 0, we perform a READ operation and
if R/W = 1, we perform a WRITE operation

If the contents of address bus is 0101 and contents of data bus is 1100 and R/W = 1, then 1100 will be written in location 5.

If the contents of address bus is 1011 and R/W=0, then the contents of location 1011 will be placed in data bus.

In next section, we will explain how to perform memory access operation in our small hypothetical computer.

Memory Instruction

We need some more instruction to work with the computer. Apart from the instruction needed to perform task inside CPU, we need some more instructions for data transfer from main memory to CPU and vice versa.

In our hypothetical machine, we use three signal lines to identify a particular instruction. If we want to include more instruction, we need additional signal lines.

Instruction	Code	Meaning
1000	LDAI imm	Load register A with data that is given in the program
1001	LDAA addr	Load register A with data from memory location addr
1010	LDBI imm	Load register B with data
1011	LDBA addr	Load register B with data from memory location addr
1100	STC addr	Store the value of register C in memory location addr
1101	HALT	Stop the execution
1110	NOP	No operation
1111	NOP	No operation

With this additional signal line, we can go upto 16 instructions. When the signal of this new line is 0, it will indicate the ALU operation. For signal value equal to 1, it will indicate 8 new instructions. So, we can design 8 new memory access instructions.

We have added 6 new instructios. Still two codes are unused, which can be used for other purposes. We show it as **NOP** means No Operation.

We have seen that for ALU operation, instruction decoder generated the signal for appropriate ALU operation.

Apart from that we need many more signals for proper functioning of the computer. Therefore, we need a module, which is known as control unit, and it is a part of CPU. The control unit is responsible to generate the appropriate signal.

As for example, for LDAI instruction, control unit must generate a signal which enables the register A to store in data into register A.

One major task is to design the control unit to generate the appropriate signal at appropriate time for the proper functioning of the computer.

Consider a simple problem to add two numbers and store the result in memory, say we want to add 7 to 5.

To solve this problem in computer, we have to write a computer program. The program is machine specific, and it is related to the instruction set of the machine.

For our hypothetical machine, the program is as follows

Instruction	Binary	HEX	Memory Location
LDAI 5	1000 0101	8 5	(0, 1)
LDBI 7	1010 0111	A 7	(2, 3)
ADD	0000	0	(4)
STC 15	1100 1111	C F	(5, 6)
HALT	1101	D	(7)

See the next Page for a flash Demo...

Introduction to Computer System

Consider another example, say that the first number is stored in memory location 13 and the second data is stored in memory location 14. Write a program to Add the contents of memory location 13 and 14 and store the result in memory location 15.

Instruction	Binary	HEX	Memory Location
LDAA 13	1000 0101	8 5	(0, 1)
LDBA 14	1010 0111	A 7	(2, 3)
ADD	0000	0	(4)
STC 15	1100 1111	C F	(5, 6)
HALT	1101	D	(7)

One question still remain unanswerd: How to store the program or data to main memory. Once we put the program and data in main memory, then only CPU can execute the program. For that we need some more instructions.

We need some instructions to perform the input tasks. These instructions are responsible to provide the input data from input devices and store them in main memory. For example instructions are needed to take input from keyboard.

We need some other instructions to perform the output tasks. These instructions are responsible to provide the result to output devices. For example, instructions are needed to send the result to printer.

We have seen that number of instructions that can be provided in a computer depends on the signal lines that are used to provide the instruction, which is basically the size of the storage devices of the computer.

For uniformity, we use same size for all storage space, which are known as register. If we work with a 16-bit machine, total instructions that can be implemented is 2^{16}.

The model that we have described here is known as **Von Neumann Stored Program Concept**. First we have to store all the instruction of a program in main memory, and CPU can work with the contents that are stored in main memory. Instructions are executed one after another.

We have explained the concept of computer in very high level abstraction by omitting most of the details.

As the course progresses we will explain the exact working principle of computer in more details.

Main Memory Organization: Stored Program

The present day digital computers are based on stored-program concept introduced by Von Neumann. In this stored-program concept, programs and data are stored in separate storage unit called memories.

Central Processing Unit, the main component of computer can work with the information stored in storage unit only.

In 1946, Von Neumann and his colleagues began the design of a stored-program computer at the Institute for Advanced Studies in Princeton.This computer is referred as the IAS computer.

The structure of IAS computer is shown in the next page.

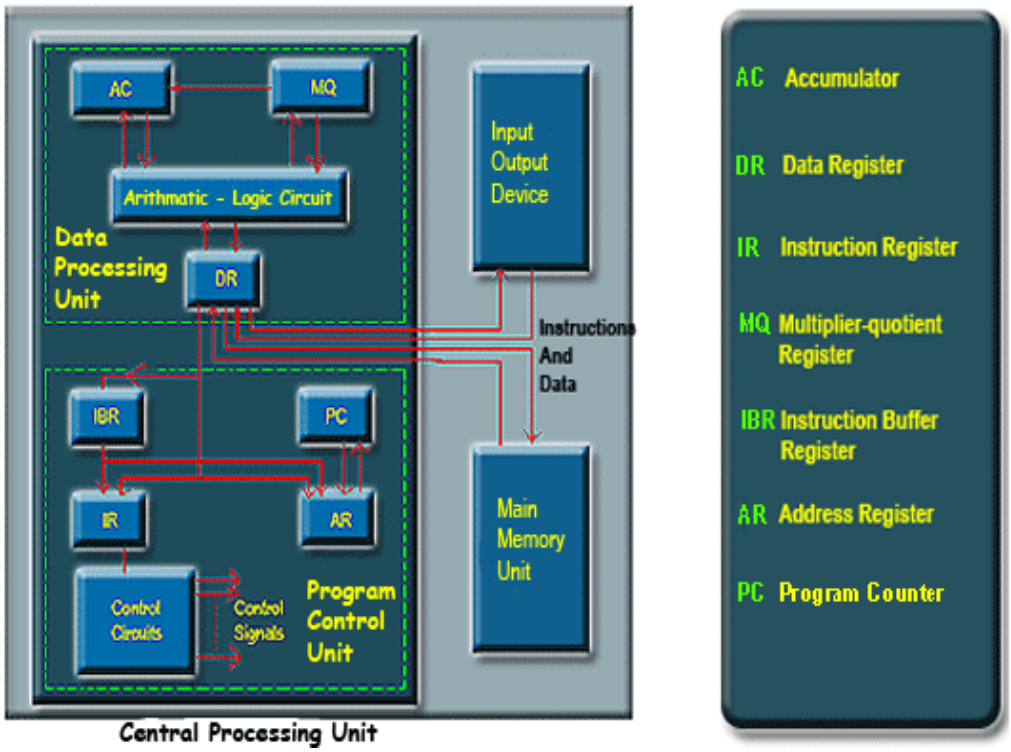


Figure : Structure of a first generation computer : IAS

The IAS computer is having three basic units:

- The Central Processing Unit (CPU).
- The Main Memory Unit.
- The Input/Output Device.

Central Processing Unit:

This is the main unit of computer, which is responsible to perform all the operations. The CPU of the IAS computer consists of a data processing unit and a program control unit.

The data processing unit contains a high speed registers intended for temporary storage of instructions, memory addresses and data. The main action specified by instructions are performed by the arithmatic-logic circuits of the data processing unit.

The control circuits in the program control unit are responsible for fetching instructions, decoding opcodes, controlling the information movements correctly through the system, and providing proper control signals for all CPU actions.

The Main Memory Unit:

It is used for storing programs and data. The memory locations of memory unit is uniquely specified by the memory address of the location. M(X) is used to indicate the location of the memory unit M with address X.

The data transfer between memory unit and CPU takes place with the help of data register DR. When CPU wants to read some information from memory unit, the information first brings to DR, and after that it goes to appropriate position.Similarly, data to be stored to memory must put into DR first, and then it is stored to appropriate location in the memory unit.

The address of the memory location that is used during memory read and memory write operations are stored in the memory register AR.

The information fetched from the memory is a operand of an instruction, then it is moved from DR to data processing unit (either to AC or MQ). If it is an operand, then it is moved to program control unit (either to IR or IBR).

Two additional registers for the temporary storage of operands and results are included in data processing units: the accumulator AC and the multiplier-quotient register MQ.

Two instructions are fetch simultaneously from M and transferred to the program control unit. The instruction that is not to be executed immediately is placed in the instruction buffer register IBR. The opcode of the other instruction is placed in the instruction register IR where it is decoded.

In the decoding phase, the control circuits generate the required control signals to perform the specified operation in the instruction.

The program counter PC is used to store the address of the next instruction to be fetched from memory.

Input Output Device :

Input devies are used to put the information into computer. With the help of input devices we can store information in memory so that CPU can use it. Program or data is read into main memory from input device or secondary storage under the control of CPU input instruction.

Output devices are used to output the information from computer. If some results are evaluated by computer and it is stored in computer, then with the help of output devices, we can present it to the user. Output data from the main memory go to output device under the control of CPU output instruction.

Binary Number System

We have already mentioned that computer can handle with two type of signals, therefore, to represent any information in computer, we have to take help of these two signals.

These two signals corresponds to two levels of electrical signals, and symbolically we represent them as 0 and 1.

In our day to day activities for arithmatic, we use the **Decimal Number System**. The decimal number system is said to be of base, or radix 10, because it uses ten digits and the coefficients are multiplied by power of 10.

A decimal number such as 5273 represents a quantity equal to 5 thousands plus 2 hundres, plus 7 tens, plus 3 units. The thousands, hundreds, etc. are powers of 10 implied by the position of the coefficients. To be more precise, 5273 should be written as:

$$5\times10^3 + 2\times10^2 + 7\times10^1 + 3\times10^0$$

However, the convention is to write only the coefficient and from their position deduce the necessary power of 10.

In decimal number system, we need 10 different symbols. But in computer we have provision to represent only two symbols. So directly we can not use deciman number system in computer arithmatic.

Number System and Representation

Print this page

<< Previous | First | Last | Next >>

For computer arithmatic we use **binary number system**. The binary number system uses two symbols to represent the number and these two symbols are 0 and 1.

The binary number system is said to be of base 2 or radix 2, because it uses two digits and the coefficients are multiplied by power of 2.

The binary number 110011 represents the quantity equal to:

$$1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 51$$ (in decimal)

We can use binary number system for computer arithmatic.

<< Previous | First | Last | Next >>

Representation of Unsigned Integers

Any integer can be stored in computer in binary form. As for example:

The binary equivalent of integer 107 is 1101011, so 1101011 are stored to represent 107.

What is the size of Integer that can be stored in a Computer?

It depends on the word size of the Computer. If we are working with 8-bit computer, then we can use only 8 bits to represent the number.The eight bit computer means the storage organization for data is 8 bits.

In case of 8-bit numbers, the minimum number that can be stored in computer is 00000000 (0) and maximum number is 11111111 (255) (if we are working with natural numbers).

So, the domain of number is restricted by the storage capacity of the computer. Also it is related to number system; above range is for natural numbers.

In general, for n-bit number, the range for natural number is from $0 \text{ to } 2^n - 1$

Any arithmetic operation can be performed with the help of binary number system. Consider the following two examples, where decimal and binary additions are shown side by side.

01101000	**104**
00110001	**49**
----------	------
10011001	**153**

In the above example, the result is an 8-bit number, as it can be stored in the 8-bit computer, so we get the correct results.

10000001	**129**
10101010	**178**
----------	------
100101011	**307**

In the above example, the result is a 9-bit number, but we can store only 8 bits, and the most significant bit (msb) cannot be stored.

The result of this addition will be stored as (00101011) which is 43 and it is not the desired result. Since we cannot store the complete result of an operation, and it is known as the overflow case.

The smallest unit of information is known as **BIT** (BInary digiT).

The binary number 110011 consists of 6 bits and it represents:

$$1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

For an n-bit number the coefficient is - a_j multiplied by 2^j where, ($0 \leq j < n$)

The coefficient $a(n-1)$ is multiplied by $2^{(n-1)}$ and it is known as most significant bit (MSB).

The coefficient a_0 is multiplied by 2^0 and it is known as least significant bit (LSB).

For our convenient, while writing in paper, we may take help of other number systems like octal and hexadecimal. It will reduce the burden of writing long strings of 0s and 1s.

Octal Number : The octal number system is said to be of base, or radix 8, because it uses 8 digits and the coefficients are multiplied by power of 8.

Eight digits used in octal system are: 0, 1, 2, 3, 4, 5, 6 and 7.

Hexadecimal number : The hexadecimal number system is said to be of base, or radix 16, because it uses 16 symbols and the coefficients are multiplied by power of 16.

Sixteen digits used in hexadecimal system are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.

Consider the following addition example:

Binary	Octal	Hexadecimal	Decimal
01101000	150	68	104
00111010	072	3A	58
----------	------	------	-----
10100010	242	A2	162

Signed Integer

We know that for n-bit number, the range for natural number is from $0 \text{ to } 2^n - 1$.

For n-bit, we have all together 2^n different combination, and we use these different combination to represent 2^n numbers, which ranges from $0 \text{ to } 2^n - 1$.

If we want to include the negative number, naturally, the range will decrease. Half of the combinations are used for positive number and other half is used for negative number.

For n-bit represenatation, the range is from $-2^{n-1} - 1 \text{ to } +2^{n-1} - 1$.

For example, if we consider 8-bit number, then range

for natural number is from 0 to 255; but
for signed integer the range is from -127 to +127.

Representation of signed integer

We know that for n-bit number, the range for natural number is from $0 \; to \; 2^n - 1$.

There are **three different schemes** to represent negative number:

- **Signed-Magnitude form.**
- **1's complement form.**
- **2's complement form.**

Signed magnitude form:

In signed-magnitude form, one particular bit is used to indicate the sign of the number, whether it is a positive number or a negative number. Other bits are used to represent the magnitude of the number.

For an n-bit number, one bit is used to indicate the signed information and remaining (n-1) bits are used to represent the magnitude. Therefore, the range is from $-2^{n-1} - 1 \; to \; +2^{n-1} - 1$.

Generally, Most Significant Bit (MSB) is used to indicate the sign and it is termed as signed bit. 0 in signed bit indicates positive numvber and 1 in signed bit indicates negative number.

For example, 01011001 represents + 169 and
11011001 represents - 169

What is 00000000 and 10000000 in signed magnitude form?

The concept of complement

The concept of complements is used to represent signed number.

Consider a number system of base-r or radix-r. There are two types of complements,

- The radix complement or the r's complement.
- The diminished radix complement or the (r - 1)'s complement.

Diminished Radix Complement :

Given a number N in base r having n digits, the (r - 1)'s complement of N is defined as $(r^n - 1) - N$.

For decimal numbers, r = 10 and r - 1 = 9, so the 9's complement of N is $(10^n - 1) - N$.

e.g., 9's complement of 5642 is 9999 - 5642 = 4357.

Radix Complement :

The r's complement of an n-digit number in base *r* is defined as $(r^N - N)$ for N != 0 and 0 for N = 0.

r's complement is obtained by adding 1 to the (r - 1)'s complement, since $(r^n - N) = [(r^n - 1) - N] + 1$

e.g., 10's complement of 5642 is 9's complement of 5642 + 1, i.e., 4357 + 1 = 4358
e.g., 2's complement of 1010 is 1's complement of 1010 + 1, i.e., 0101 + 1 = 0110.

Representation of Signed integer in 1's complement form:

Consider the eight bit number 01011100, 1's complements of this number is 10100011. If we perform the following addition:

If we add 1 to the number, the result is 100000000.

0 1 0 1 1 1 0 0	
1 0 1 0 0 0 1 1	
----------	
1 1 1 1 1 1 1 1	

Since we are considering an eight bit number, so the 9th bit (MSB) of the result can not be stored. Therefore, the final result is 00000000.

Since the addition of two number is 0, so one can be treated as the negative of the other number. So, 1's complement can be used to represent negative number.

Representation of Signed integer in 2's complement form:

Consider the eight bit number 01011100, 2's complements of this number is 10100100. If we perform the follwoing addition:

```
  0 1 0 1 1 1 0 0
  1 0 1 0 0 0 1 1
 ----------------
1 0 0 0 0 0 0 0 0
```

Since we are considering an eight bit number, so the 9th bit (MSB) of the result can not be stored. Therefore, the final result is 00000000.

Since the addition of two number is 0, so one can be treated as the negative of the other number. So, 2's complement can be used to represent negative number.

Number System and Representation

Print this page

<< Previous | First | Last | Next >>

Decimal	2's Complement	1's complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	-----	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	------	-------

<< Previous | First | Last | Next >>

Representation of Real Number

Representation of Real Number:

Binary representation of 41.6875 is 101001.1011

Therefore any real number can be converted to binary number system

There are **two schemes** to represent real number :

- Fixed-point representation
- Floating-point representation

Fixed-point representation:

Binary representation of 41.6875 is 101001.1011

To store this number, we have to store **two information**,

-- the part before decimal point and
-- the part after decimal point.

This is known as fixed-point representation where the position of decimal point is fixed and number of bits before and after decimal point are also predefined.

If we use 16 bits before decimal point and 8 bits after decimal point, in signed magnitude form, the range is

$$-2^{16} - 1 \text{ to } +2^{16} - 1 \text{ and the precision is } 2^{(-8)}$$

One bit is required for sign information, so the total size of the number is 25 bits

(1(sign) + 16(before decimal point) + 8(after decimal point)).

Floating-point representation:

In this representation, numbers are represented by a mantissa comprising the significant digits and an exponent part of Radix R. The format is:

$$mantissa * R^{exponent}$$

Numbers are often normalized, such that the decimal point is placed to the right of the **first non zero digit**.

For example, the decimal number,

5236 is equivqlent to $5.236 * 10^3$

To store this number in floating point representation, we store 5236 in mantissa part and 3 in exponent part.

IEEE standard floating point format:

IEEE has proposed two standard for representing floating-point number:

- Single precision
- Double precision

Single Precision:

S: sign bit: 0 denoted + and 1 denotes -

E: 8-bit excess -27 exponent

M: 23-bit mantissa

Double Precision:

S: sign bit: 0 denoted + and 1 denotes -

E: 11-bit excess -1023 exponent

M: 52-bit mantissa

Representation of Character

Since we are working with 0's and 1's only, to represent character in computer we use strings of 0's and 1's only.

To represent character we are using some coding scheme, which is nothing but a mapping function.

Some of standard coding schemes are: **ASCII, EBCDIC, UNICODE**.

ASCII : American Standard Code for Information Interchange.
It uses a 7-bit code. All together we have 128 combinations of 7 bits and we can represent 128 character.
As for example 65 = 1000001 represents character 'A'.

EBCDIC : Extended Binary Coded Decimal Interchange Code.
It uses 8-bit code and we can represent 256 character.

UNICODE : It is used to capture most of the languages of the world. It uses 16-bit

Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language. The Unicode Standard has been adopted by such industry leaders as Apple, HP, IBM, JustSystem, Microsoft, Oracle, SAP, Sun, Sybase, Unisys and many others.

A Brief History of Computer Architecture

Computer Architecture is the field of study of selecting and interconnecting hardware components to create computers that satisfy functional performance and cost goals. It refers to those attributes of the computer system that are visible to a programmer and have a direct effect on the execution of a program.

Computer Architecture concerns Machine Organization, interfaces, application, technology, measurement & simulation that Includes:

- **Instruction set**
- **Data formats**
- **Principle of Operation (formal description of every operation)**
- **Features (organization of programmable storage, registers used, interrupts mechanism, etc.)**

In short, it is the combination of Instruction Set Architecture, Machine Organization and the related hardware.

The Brief History of Computer Architecture

First Generation (1940-1950) :: Vacuum Tube

- **ENIAC [1945]:** Designed by Mauchly & Echert, built by US army to calculate trajectories for ballistic shells during Worls War II. Around 18000 vacuum tubes and 1500 relays were used to build ENIAC, and it was programmed by manually setting switches
- **UNIVAC [1950]:** the first commercial computer
- **John Von Neumann architecture:** Goldstine and Von Neumann took the idea of ENIAC and developed concept of storing a program in the memory. Known as the Von Neumann's architecture and has been the basis for virtually every machine designed since then.

Features:

- Electron emitting devices
- Data and programs are stored in a single read-write memory
- Memory contents are addressable by location, regardless of the content itself
- Machine language/Assemble language
- Sequential execution

Second Generation (1950-1964) :: Transistors

- William Shockley, John Bardeen, and Walter Brattain invent the **transistor** that reduce size of computers and improve reliability. Vacuum tubes have been replaced by transistors.
- **First operating Systems:** handled one program at a time
- **On-off switches** controlled by electronically.
- **High level languages**
- **Floating point arithmetic**

Third Generation (1964-1974) :: Integrated Circuits (IC)

- **Microprocessor chips** combines thousands of transistors, entire circuit on one computer chip.
- **Semiconductor memory**
- **Multiple computer models** with different performance characteristics
- The **size** of computers has been reduced drastically

Fourth Generation (1974-Present) :: Very Large-Scale Integration (VLSI) / Ultra Large Scale Integration (ULSI)

- **Combines** millions of transistors
- **Single-chip processor** and the single-board computer emerged
- Creation of the **Personal Computer (PC)**
- Use of **data communications**
- Massively **parallel machine**

Evolution of Instruction Sets

Instruction Set Architecture (ISA) Abstract interface between the Hardware and lowest-level Software

- 1950: **Single Accumulator**: EDSAC
- 1953: **Accumulator plus Index Registers:** Manchester Mark I, IBM 700 series
- **Separation of programming Model from implementation**:
  - 1963: High-level language Based: B5000
  - 1964: Concept of a Family: IBM 360
- **General Purpose Register Machines**:
  - 1977-1980: **CISC** - Complex Instruction Sets computer: Vax, Intel 432
  - 1963-1976: **Load/Store Architecture**: CDC 6600, Cray 1
  - 1987: **RISC**: Reduced Instruction Set Computer: Mips, Sparc, HP-PA, IBM RS6000

Typical RISC:

- Simple, no complex addressing
- Constant length instruction, 32-bit fixed format
- Large register file
- Hard wired control unit, no need for micro programming
- Just about every opposites of CISC

Brief History of Computer Evolution Print this page

<< Previous | First | Last | Next >>

Major advances in computer architecture are typically associated with landmark instruction set designs. Computer architecture's definition itself has been through bit changes. The following are the main concern for computer architecture through different times:

- 1930-1950: Computer arithmetic
  - Microprogramming
  - Pipelining
  - Cache
  - Timeshared multiprocessor
- 1960: Operating system support, especially memory management
  - Virtual memory
- 1970-1980: **Instruction Set Design**, especially for compilers; **Vector processing** and **shared memory multiprocessors**
  - RISC
- 1990s: Design of CPU, memory system, I/O system, multi-processors, networks
  - CC-UMA multiprocessor
  - CC-NUMA multiprocessor
  - Not-CC-NUMA multiprocessor
  - Message-passing multiprocessor

>> Previous | First | Last | Next >>

- 2000s: Special purpose architecture, functionally reconfigurable,special considerations for low power/mobile processing, chip multiprocessors, memory systems
  - Massive SIMD
  - Parallel processing multiprocessor

Under a rapidly changing set of forces, computer technology keeps at dramatic change, for example:

- **Processor clock rate** at about 20% increase a year
- **Logic capacity** at about 30% increase a year
- **Memory speed** at about 10% increase a year
- **Memory capacity** at about 60% increase a year
- **Cost per bit** improves about 25% a year
- **The disk capacity** increase at 60% a year.

A Brief History of Computer Organization

If **computer architecture** is a view of the *whole design with the important characteristics* visible to programmer, **computer organization** is *how features are implemented with the specific building blocks* visible to designer, such as control signals, interfaces, memory technology, etc. Computer architecture and organization are closely related, though not exactly the same.

A stored program computer has the following basic units:

- **Processor** -- center for manipulation and control
- **Memory** -- storage for instructions and data for currently executing programs
- **I/O system** -- controller which communicate with "external" devices:
  secondary memory, display devices, networks
- **Data-path & control** -- collection of parallel wires, transmits data, instructions, or control signal

Computer organization defines the ways in which these components are interconnected and controlled. It is the capabilities and performance characteristics of those principal functional units. Architecture can have a number of organizational implementations, and organization differs between different versions. Such, all **Intel x86** families share the same basic architecture, and **IBM system/370** family share their basic architecture.

The history of Computer Organization

Computer architecture has progressed five generation: **vacuum tubes**, **transistors**, **integrated circuits**, and **VLSI**. Computer organization has also made its historic progression accordingly.

The advance of microprocessor (Intel)

- **1977:** 8080 - the first general purpose microprocessor, 8 bit data path, used in first personal computer
- **1978:** 8086 - much more powerful with 16 bit, 1MB addressable, instruction cache, prefetch few instructions
- **1980:** 8087 - the floating point coprocessor is added
- **1982:** 80286 - 24 Mbyte addressable memory space, plus instructions
- **1985:** 80386 - 32 bit, new addressing modes and support for multitasking
- **1989 -- 1995:**
  - 80486 - 25, 33, MHz, 1.2 M transistors, 5 stage pipeline, sophisticated powerful cache and instruction pipelining, built in math co-processor.
  - Pentium - 60, 66 MHz, 3.1 M transistor, branch predictor, pipelined floating point, multiple instructions executed in parallel, first superscalar IA-32.
  - PentiumPro - Increased superscalar, register renaming, branch prediction, data flow analysis, and speculative execution
- **1995 -- 1997:** Pentium II - 233, 166, 300 MHz, 7.5 M transistors, first compaction of micro- architecture, MMX technology, graphics video and audio processing.
- **1999:** Pentium III - additional floating point instructions for 3D graphics
- **2000:** Pentium IV - Further floating point and multimedia enhancements

Evolution of Memory

- 1970: **RAM /DRAM**, 4.77 MHz
- 1987: **FPM** - fast page mode DRAM, 20 MHz
- 1995, **EDO** - Extended Data Output, which increases the read cycle between memory and CPU, 20 MHz
- 1997- 1998: **SDRAM** - Synchronous DRAM, which synchronizes itself with the CPU bus and runs at higher clock speeds, PC66 at 66 MHz, PC100 at 100 MHz
- 1999: **RDRAM** - Rambus DRAM, which DRAM with a very high bandwidth, 800 MHz
- 1999-2000: **SDRAM** - PC133 at 133 MHz, DDR at 266 MHz.
- 2001: **EDRAM** - Enhanced DRAM, which is dynamic or power-refreshed RAM, also know as cached DRAM.

Major buses and their features

A bus is a parallel circuit that connects the major components of a computer, allowing the transfer of electric impulses form one connected component to any other.

- **VESA** - Video Electronics Standard Association:
  32 bit, relied on the 486 processor to function
- **ISA** - Industry Standard Architecture:
  8 bit or 16 bit with width 8 or 16 bits. 8.3 MHz speed, 7.9 or 15.9 bandwidth accordingly.
- **EISA** - Extended Industry Standard Architecture:
  32 bits, 8.3 MHz, 31.8 bandwidth, the attempt to compete with IBM's MCA
- **PCI** - Peripheral Component Interconnect:
  32 bits, 33 MHz, 127.2 bandwidth
- **PCI-X** - Up to 133 MHz bus speed, 64 bits bandwidth, 1GB/sec throughput
- **AGP** - Accelerated Graphics Port:
  32 bits, 66 MHz, 254,3 bandwidth

Major ports and connectors/interface

- **IDE** - Integrated Drive Electronics, also know as ATA, EIDE, Ultra ATA, Ultra DMA, most widely used interface for hard disks
- **PS/2 port** - mini Din plug with 6 pins for a mouse and keyboard
- **SCSI** - Small Computer System Interface, 80 - 640 Mbs, capable of handling internal/external peripherals
- **Serial Port -** adheres to RS-232c spec, uses DB9 or DB25 connector, capable of 115kb.sec speeds
- **Parallel port** - as know as printer port, enhanced types: ECP- extended capabilities port, EPP - enhanced parallel port
- **USB - universal serial bus,** two types: 1.0 and 2.0, hot plug-and-play, at 12MB/s, up to 127 devices chain. 2.0 data rate is at 480 bits/s.
- **Firewire** - high speed serial port, 400 MB/s, hot plug-and-play, 30 times faster than USB 1.0

Computer Organization and Architecture Print this page

Module 2 : Arithmatic and Logic Unit

In this Module, we have two lectures, viz.

1. **Arithmetic and logical operation and Hardware implementation**
2. **Implementation issues of some operation**

Click the proper link on the left side for the lectures

Arithmetic and logic Unit (ALU)

ALU is responsible to perform the operation in the computer.

The basic operations are implemented in hardware level. ALU is having collection of two types of operations:

- Arithmetic operations
- Logical operations

Consider an ALU having 4 arithmetic operations and 4 logical operation.

To identify any one of these four logical operations or four arithmetic operations, two control lines are needed. Also to identify the any one of these two groups- arithmetic or logical, another control line is needed. So, with the help of three control lines, any one of these eight operations can be identified.

Consider an ALU is having four arithmetic operations. Addition, subtraction, multiplication and division. Also consider that the ALU is having four logical operations: OR, AND, NOT & EX-OR.

We need three control lines to identify any one of these operations. The input combination of these control lines are shown below:

Control line C_2 is used to identify the group: logical or arithmetic, ie

$C_2 = 0$: arithmetic operation $C_2 = 1$: logical operation.

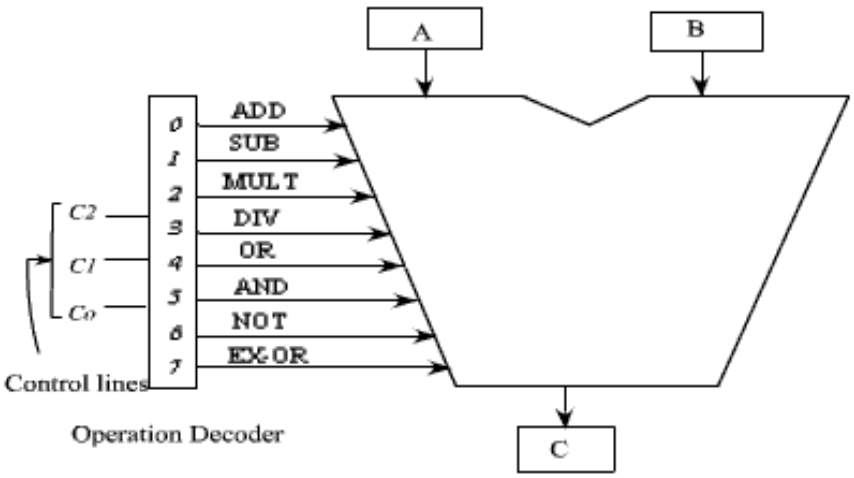
Control lines C_0 and C_1 are used to identify any one of the four operations in a group. One possible combination is given here.

C_1	C_0	Arithmetic $C_2 = 0$	Logical $C_2 = 1$
0	0	Addition	OR
0	1	Subtraction	AND
1	0	Multiplication	NOT
1	1	Division	EX-OR

Arithmetic and logic Unit (ALU) Print this page

<< Previous | First | Last | Next >>

A 3×8 decode is used to decode the instruction. The block diagram of the ALU is shown in the figure.



Block Diagram of the ALU

>> Previous | First | Last | Next >>

Arithmetic and logic Unit (ALU)

The ALU has got two input registers named as A and B and one output storage register, named as C. It performs the operation as:

$$C = A \ op \ B$$

The input data are stored in A and B, and according to the operation specified in the control lines, the ALU perform the operation and put the result in register C.

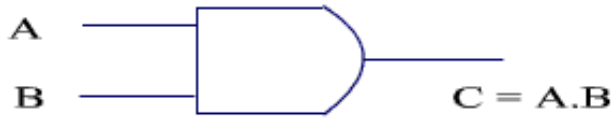
As for example, if the contents of controls lines are, 000, then the decoder enables the addition operation and it activates the adder circuit and the addition operation is performed on the data that are available in storage register A and B . After the completion of the operation, the result is stored in register C.

We should have some hardware implementations for basic operations. These basic operations can be used to implement some complicated operations which are not feasible to implement directly in hardware.

Arithmetic and logic Unit (ALU)

There are several logic gates exists in digital logic circuit. These logic gates can be used to implement the logical operation. Some of the common logic gates are mentioned here.

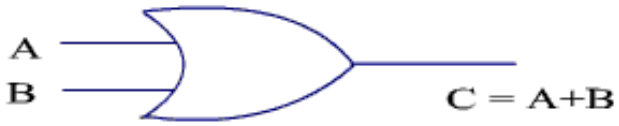
AND gate: The output is high if both the inputs are high.



A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

Arithmetic and logic Unit (ALU)

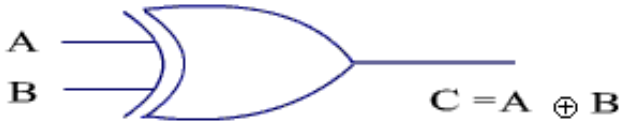
OR gate: The output is high if any one of the input is high.



A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Arithmetic and logic Unit (ALU)

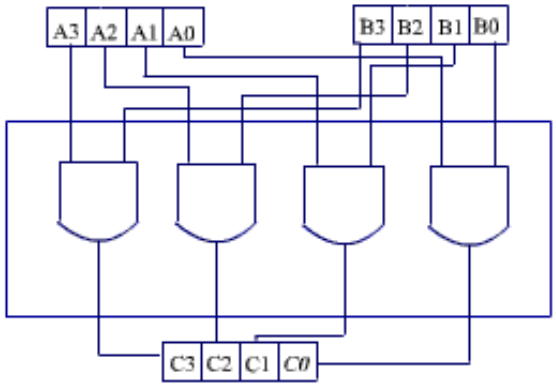
EX-OR gate: The output is high if either of the input is high.



A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

Arithmetic and logic Unit (ALU)

If we want to construct a circuit which will perform the AND operation on two 4-bit number, the implementation of the 4-bit AND operation is shown in the figure.



4-bit AND operator

Arithmetic Circuit

Binary Adder : Binary adder is used to add two binary numbers.

In general, the adder circuit needs two binary inputs and two binary outputs. The input variables designate the augends and addend bits; The output variables produce the sum and carry.

The binary addition operation of single bit is shown in the truth table

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

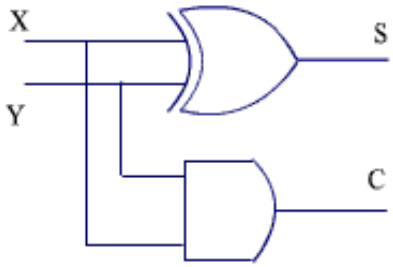
C: Carry Bit

S: Sum Bit

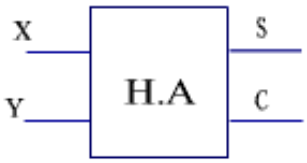
The simplified sum of products expressions are

$$S = x'y + xy'$$
$$C = xy$$

The circuit implementation is



Circuit Diagram



Block Diagram

This circuit can not handle the carry input, so it is termed as **half adder**.

Full Adder:

A full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs and two outputs.

Two of the input variables, denoted by *x* and *y*, represent the two bits to be added. The third input *Z*, represents the carry from the previous lower position.

The two outputs are designated by the symbols *S* for sum and *C* for carry.

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The simplified expression for *S* and *C* are

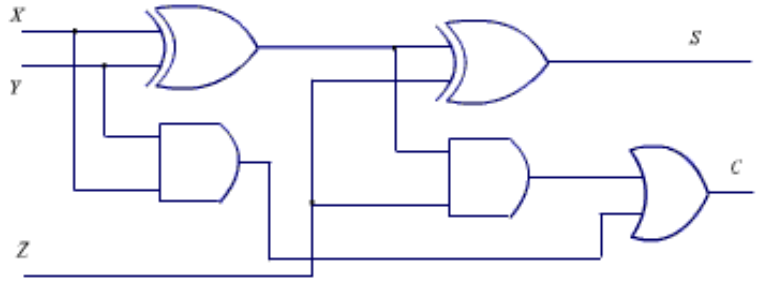
$$S = x'y'z + x'yz' + xy'z' + xyz$$
$$C = xy + xz + yz$$
$$= xy + xy'z + x'yz$$

We may rearrange these two expressions as follows:

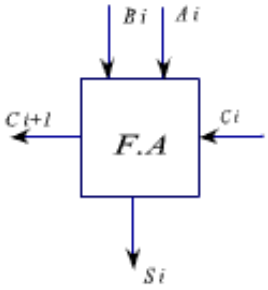
$$S = z \oplus (x \oplus y)$$
$$= z'(xy' + x'y) + z(xy' + x'y)'$$
$$= z'(xy' + x'y) + z(xy + x'y')$$
$$= xy'z' + x'yz' + xyz + x'y'z$$
$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

Arithmetic and logic Unit (ALU)

The circuit diagram full adder is shown in the figure.



Circuit Diagram



Block Diagram

n-such single bit full adder blocks are used to make *n*-bit full adder.

Arithmetic and logic Unit (ALU)

To demonstrate the binary addition of four bit numbers, let us consider a specific example.

Consider two binary numbers

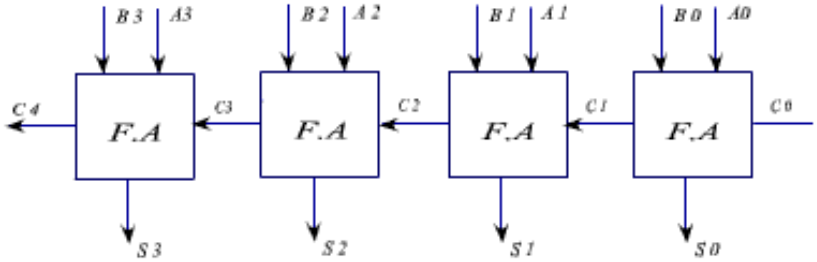
A =1 0 0 1 B = 0 0 1 1

Subscript	i	3	2	1	0
Input carry	C_i	0	1	1	0
Augend	A_i	1	0	0	1
Addend	B_i	0	0	1	1
Sum	S_i	1	1	0	0
Output Carry	C_{i+1}	0	0	1	1

Arithmetic and logic Unit (ALU)

To get the four bit adder, we have to use 4 full adder block. The carry output the lower bit is used as a carry input to the next higher bit.

The circuit of 4-bit adder shown in the figure.



Arithmetic and logic Unit (ALU)

Binary subtractor : The subtraction operation can be implemented with the help of binary adder circuit, because

$$A - B = A + (-B)$$

We know that 2's complement representation of a number is treated as a negative number of the given number.

We can get the 2's complements of a given number by complementing each bit and adding 1 to it.

The circuit for subtracting *A*-*B* consist of an added with inverter placed between each data input *B* and the corresponding input of the full adder. The input carry C_0 must be equal to 1 when performing subtraction.

The operation thus performed becomes *A* , plus the 1's complement of *B* , plus 1. This is equal to *A* plus 2's complement of *B* .

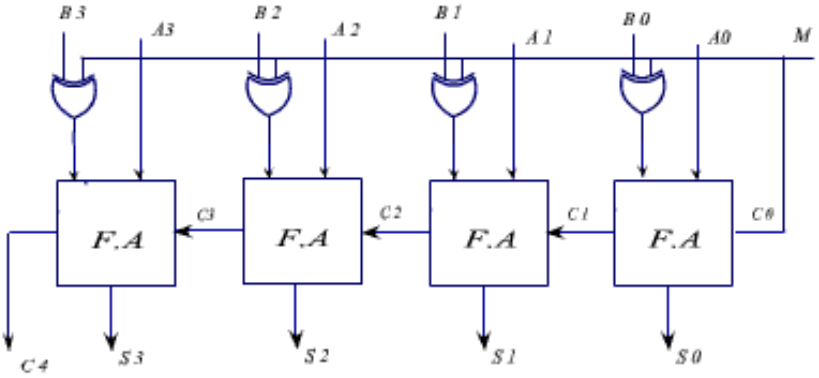
With this principle, a single circuit can be used for both addition and subtraction. The 4 bit adder subtractor circuit is shown in the figure. It has got one mode (*M*) selection input line, which will determine the operation,

Arithmetic and logic Unit (ALU)

If $M = 0$, then $A + B$

If $M = 1$ then $A - B = A + (-B)$

= A + 1's complement of $B + 1$



4-bit adder subtractor

Arithmetic and logic Unit (ALU)

The operation of OR gate:

$$x \oplus 0 = x$$
$$x \oplus 1 = x'$$

if $M = 0$, $B_i \oplus 0 = B_i$

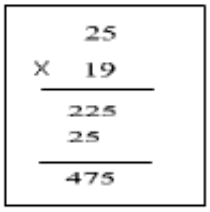
if $M = 1$, $B_i \oplus 1 = B_i'$

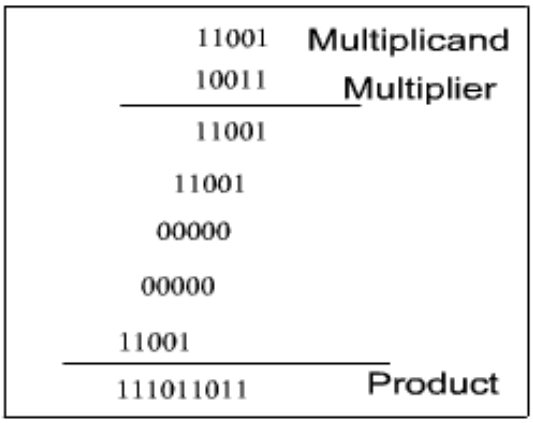
Implemental issue of some operations Print this page

Multiplication

Multiplication of two numbers in binary representation can be performed by a process of SHIFT and ADD operations. Since the binary number system allows only 0 and 1's, the digit multiplication can be replaced by SHIFT and ADD operation only, because multiplying by 1 gives the number itself and multiplying by 0 produces 0 only.

The multiplication process is illustrated with a numerical example.





Implemental issue of some operations

Print this page

<< Previous | First | Last | Next >>

The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, the multiplicand is copied down, otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product.

When multiplication is implemented in a digital computer, the process is changed slightly.

Instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers and successively accumulate the partial products in a register. It will reduce the requirements of registers.

Instead of sifting the multiplicand to the left, the partial product is shifted to right.

When the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product.

An algorithm to multiply two binary numbers. Consider that the ALU does not provide the multiplication operation, but it is having the addition operation and shifting operation. Then we can write a micro program for multiplication operation and provide the micro program code in memory. When a multiplication operation is encountered, it will execute this micro code to perform the multiplication.

<< Previous | First | Last | Next >>

Implemental issue of some operations

Print this page

<< Previous | First | Last | Next >>

The micro code is nothing but the collection of some instructions. ALU must have those operation; otherwise we must have micro code again for those operations which are not supported in ALU.

Consider a situation such that we do not have the multiplication operation in a primitive computer. Is it possible to perform the multiplication. Of course, yes, provided the addition operation is available.

We can perform the multiplication with the help of repeated addition method; for example, if we want to multiply 4 by 5 (4×5), then simply add 4 five times to get the result.

If it is possible by addition operation, then why we need a multiplication operation.

Consider a machine, which can handle 8 bit numbers, then we can represent number from 0 to 255. If we want to multiply 175×225, then there will be at least 175 addition operation.

But if we use the multiplication algorithm that involves shifting and addition, it can be done in 8 steps, because we are using an 8-bit machine.

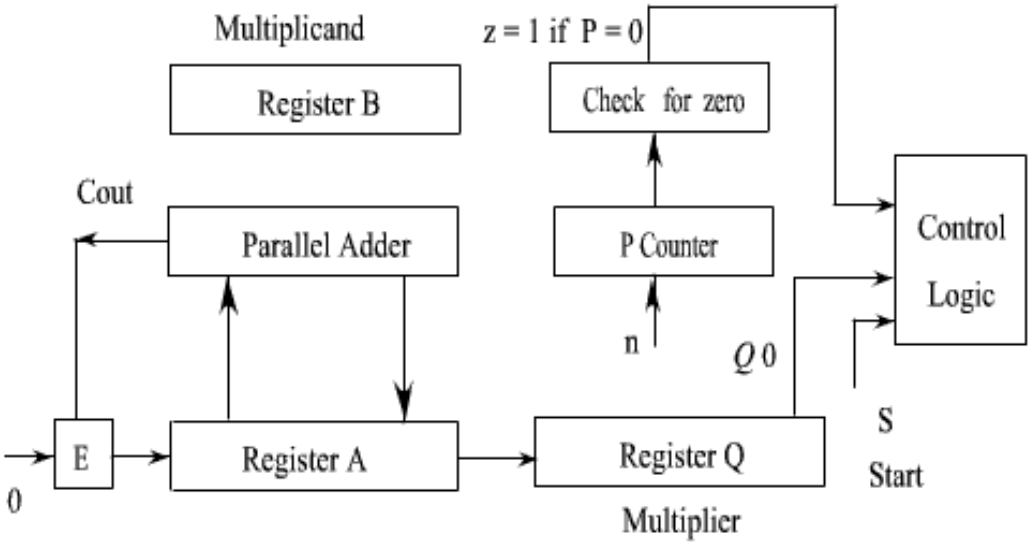
Again, the micro program execution is slightly slower, because we have to access the code from micro controller memory, and memory is a slower device than CPU.

It is possible to implement the multiplication algorithm in hardware.

<< Previous | First | Last | Next >>

Binary Multiplier, Hardware Implementation

The block diagram of binary multiplier is shown in the figure.



Block diagram of binary multiplier

Implemental issue of some operations Print this page

<< Previous | First | Last | Next >>

The multiplicand is stored in register B and the multiplier is stored in register Q.

The partial product is formed in register A and stored in A and Q

The counter P is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.

Initially, the multiplicand is in register B and the multiplier in Q. The register A is reset to 0.

The sum of A and B forms a partial product- which is transferred to the EA register.

Both partial product and multiplier are shifted to the right. The least significant bit of A is shifted into the most significant position of Q; and 0 is shifted into E.

After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right.

The right most flip flop in register Q, designated by Q_0 will hold the bit of the multiplier which must be inspected next. If the content of this bit is 0, then it is not required to add the multiplicand, only shifting is needed. If the content of this bit is 1, then both addition and shifting are needed.

<< Previous | First | Last | Next >>

Implemental issue of some operations Print this page

<< Previous | First | Last | Next >>

After each shifter, value of counter P is decremented and the process continues till the counter value becomes 0.

The final result is available in (EAQ) registers combination.

To control the operation, it is required to design the appropriate control logic that is shown in the block diagram.

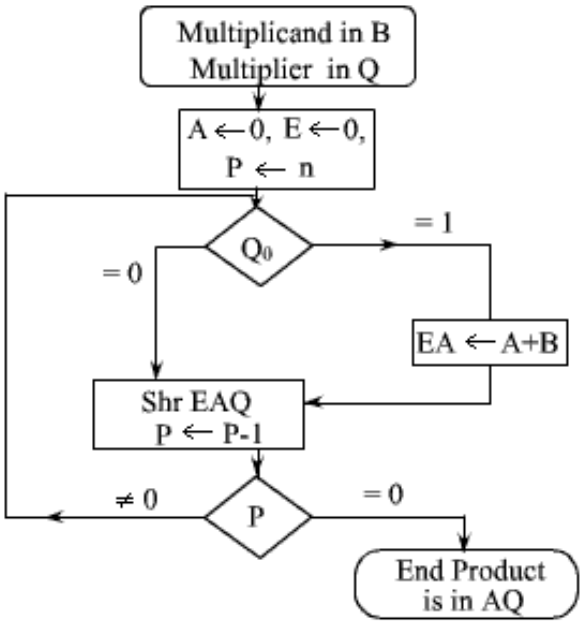
>> Previous | First | Last | Next >>

Implemental issue of some operations

Print this page

<< Previous | First | Last | Next >>

The flow chart of the multiplication operation is given in the figure.



>> Previous | First | Last | Next >>

Implemental issue of some operations

The working of multiplication algorithm is shown here with the help of an example.

Multiplicand $B = 11001$

	E	A	Q	P
Multiplier in Q	0	0 0 0 0 0	1 0 0 1 1	5
$Q_0 = 1$, Add B		1 1 0 0 1		
		---		
first partial product	0	1 1 0 0 1		
Shr EAQ	0	0 1 1 0 0	1 1 0 0 1	4
$Q_0 = 1$,Add B		1 1 0 0 1		

Second partial product	1	0 0 1 0 1		
Shr EAQ	0	1 0 0 1 0	1 1 1 0 0	3
$Q_0 = 0$, Shr EAQ	0	0 1 0 0 1	0 1 1 1 0	2

$Q_0 = 0$, Shr EAQ	0	0 1 0 0 1	0 1 1 1 0	1
$Q_0 = 1$, Add B		1 1 0 0 1		
		---		
fifth partial product	0	1 1 1 0 1		
Shr EAQ	0	0 1 1 1 0	1 1 0 1 1	0
Stop				
Final products:	0	1 1 1 0	1 1 0 1 1	

Computer Organization and Architecture Print this page

Module 3 : Memory

In this Module, we have four lectures, viz.

1. **Concept of Memory.**
2. **Cache Memory.**
3. **Memory Management**
4. **Virtual memory**

Click the proper link on the left side for the lectures.

Concept of Memory

We have already mentioned that digital computer works on stored programmed concept introduced by Von Neumann. We use memory to store the information, which includes both program and data.

Due to several reasons, we have different kind of memories. We use different kind of memory at different lavel.

The memory of computer is broadly categories into two categories:

- Internal and
- external

Internal memory is used by CPU to perform task and external memory is used to store bulk information, which includes large software and data.

Memory is used to store the information in digital form. The memory hierarchy is given by:

- Register
- Cache Memory
- Main Memory
- Magnetic Disk
- Removable media (Magnetic tape)

Register:

This is a part of Central Processor Unit, so they reside inside the CPU. The information from main memory is brought to CPU and keep the information in register. Due to space and cost constraints, we have got a limited number of registers in a CPU. These are basically faster devices.

Cache Memory:

Cache memory is a storage device placed in between CPU and main memory. These are semiconductor memories. These are basically fast memory device, faster than main memory.

We can not have a big volume of cache memory due to its higher cost and some constraints of the CPU. Due to higher cost we can not replace the whole main memory by faster memory. Generally, the most recently used information is kept in the cache memory. It is brought from the main memory and placed in the cache memory. Now a days, we get CPU with internal cache.

Main Memory:

Like cache memory, main memory is also semiconductor memory. But the main memory is relatively slower memory. We have to first bring the information (whether it is data or program), to main memory. CPU can work with the information available in main memory only.

Magnetic Disk:

This is bulk storage device. We have to deal with huge amount of data in many application. But we don't have so much semiconductor memory to keep these information in our computer. On the other hand, semiconductor memories are volatile in nature. It loses its content once we switch off the computer. For permanent storage, we use magnetic disk. The storage capacity of magnetic disk is very high.

Removable media:

For different application, we use different data. It may not be possible to keep all the information in magnetic disk. So, which ever data we are not using currently, can be kept in removable media. Magnetic tape is one kind of removable medium. CD is also a removable media, which is an optical device.

Concept of Memory

Print this page

<< Previous | First | Last | Next >>

Register, cache memory and main memory are internal memory. Magnetic Disk, removable media are external memory. Internal memories are semiconductor memory. Semiconductor memories are categoried as volatile memory and non-volatile memory.

RAM: Random Access Memories are **volatile** in nature. As soon as the computer is switched off, the contents of memory are also lost.

ROM: Read only memories are **non volatile** in nature. The storage is permanent, but it is read only memory. We can not store new information in ROM.

Several types of ROM are available:

- **PROM:** Programmable Read Only Memory; it can be programmed once as per user requirements.
- **EPROM:** Erasable Programmable Read Only Memory; the contents of the memory can be erased and store new data into the memory. In this case, we have to erase whole information.
- **EEPROM:** Electrically Erasable Programmable Read Only Memory; in this type of memory the contents of a particular location can be changed without effecting the contents of other location.

<< Previous | First | Last | Next >>

Main Memory

The main memory of a computer is semiconductor memory. The main memory unit of computer is basically consists of two kinds of memory:

RAM : Random access memory; which is volatile in nature.
ROM: Read only memory; which is non-volatile.

The permanent information are kept in ROM and the user space is basically in RAM.

The smallest unit of information is known as bit (binary digit), and in one memory cell we can store one bit of information. 8 bit together is termed as a byte.

The maximum size of main memory that can be used in any computer is determined by the addressing scheme.

A computer that generates 16-bit address is capable of addressing upto 2^{16} which is equal to 64K memory location. Similarly, for 32 bit addresses, the total capacity will be 2^{32} which is equal to 4G memory location.

In some computer, the smallest addressable unit of information is a memory word and the machine is called word-addressable.

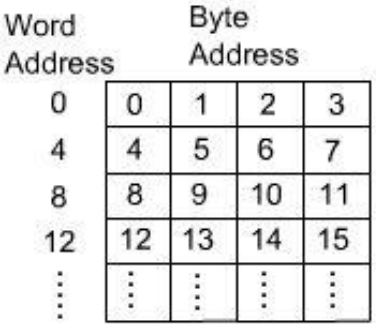
Concept of Memory

In some computer, individual address is assigned for each byte of information, and it is called **byte-addressable computer**. In this computer, one memory word contains one or more memory bytes which can be addressed individually.

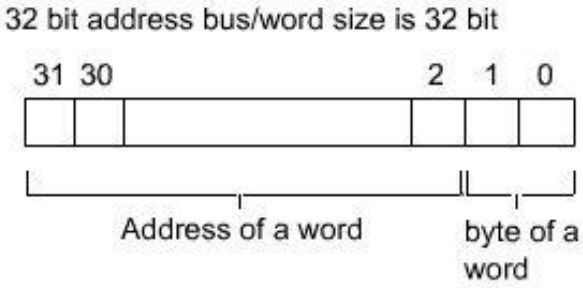
A byte addressable 32-bit computer, each memory word contains 4 bytes. A possible way of address assignment is shown in figure. The address of a word is always integer multiple of 4.

The main memory is usually designed to store and retrieve data in word length quantities. The word length of a computer is generally defined by the number of bits actually stored or retrieved in one main memory access.

Consider a machine with 32 bit address bus. If the word size is 32 bit, then the high order 30 bit will specify the address of a word. If we want to access any byte of the word, then it can be specified by the lower two bit of the address bus.



Organization of the main memory in a 32-bit byte addressable computer



Concept of Memory

The data transfer between main memory and the CPU takes place through two CPU registers.

- **MAR** : Memory Address Register
- **MDR** : Memory Data Register.

If the MAR is k-bit long, then the total addressable memory location will be 2^k.

If the MDR is n-bit long, then the n bit of data is transferred in one memory cycle.

The transfer of data takes place through memory bus, which consist of address bus and data bus. In the above example, size of data bus is n-bit and size of address bus is k bit.

It also includes control lines like Read, Write and Memory Function Complete (MFC) for coordinating data transfer. In the case of byte addressable computer, another control line to be added to indicate the byte transfer instead of the whole word.

Concept of Memory

Print this page

<< Previous | First | Last | Next >>

For memory operation, the CPU initiates a memory operation by loading the appropriate data i.e., address to MAR.

If it is a memory read operation, then it sets the read memory control line to 1. Then the contents of the memory location is brought to MDR and the memory control circuitry indicates this to the CPU by setting MFC to 1.

If the operation is a memory write operation, then the CPU places the data into MDR and sets the write memory control line to 1. Once the contents of MDR are stored in specified memory location, then the memory control circuitry indicates the end of operation by setting MFC to 1.

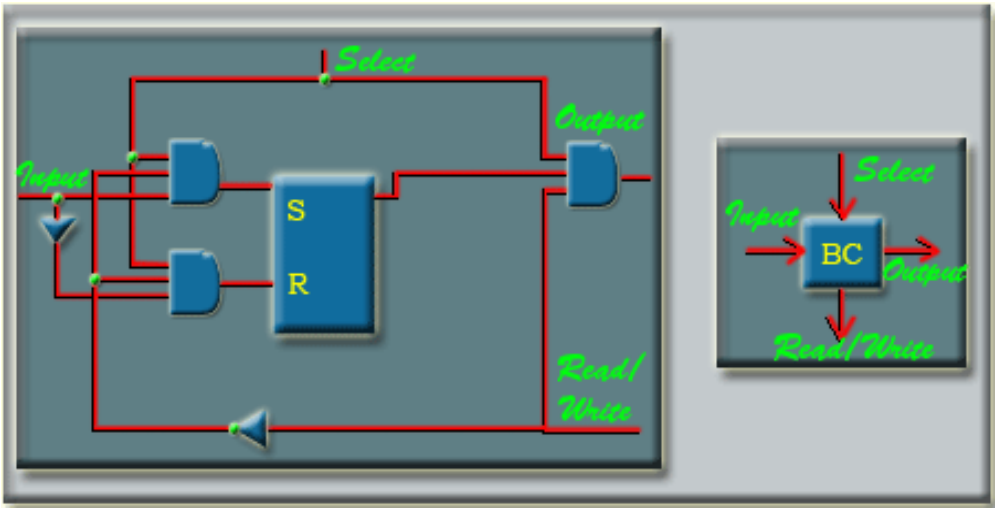
A useful measure of the speed of memory unit is the time that elapses between the initiation of an operation and the completion of the operation (for example, the time between Read and MFC). This is referred to as **Memory Access Time**. Another measure is memory cycle time. This is the minimum time delay between the initiation two independent memory operations (for example, two successive memory read operation). Memory cycle time is slightly larger than memory access time.

<< Previous | First | Last | Next >>

Binary Storage Cell:

The binary storage cell is the basic building block of a memory unit.

The binary storage cell that stores one bit of information can be modelled by an SR latch with associated gates. This model of binary storage cell is shown in the figure.



1 bit Binary Cell (BC)

The binary cell sotres one bit of information in its internal latch.

Control input to binary cell

Select	Read/Write	Memory Operation
0	X	None
1	0	Write
1	1	Read

The storage part is modelled here with SR-latch, but in reality it is an electronics circuit made up of transistors.

The memory consttucted with the help of transistors is known as semiconductor memory. The semiconductor memories are termed as Random Access Memory(RAM), because it is possible to access any memory location in random.

Depending on the technology used to construct a RAM, there are two types of RAM -

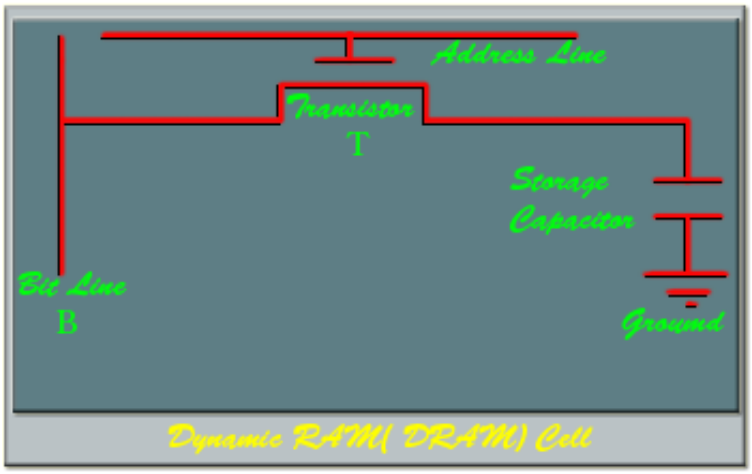
SRAM: Static Random Access Memory.
DRAM: Dynamic Random Access Memory.

Dynamic Ram (DRAM):

A DRAM is made with cells that store data as charge on capacitors. The presence or absence of charge in a capacitor is interpreted as binary 1 or 0.

Because capacitors have a natural tendency to discharge due to leakage current, dynamic RAM require periodic charge refreshing to maintain data storage. The term dynamic refers to this tendency of the stored charge to leak away, even with power continuously applied.

A typical DRAM structure for an individual cell that stores one bit information is shown in the figure.



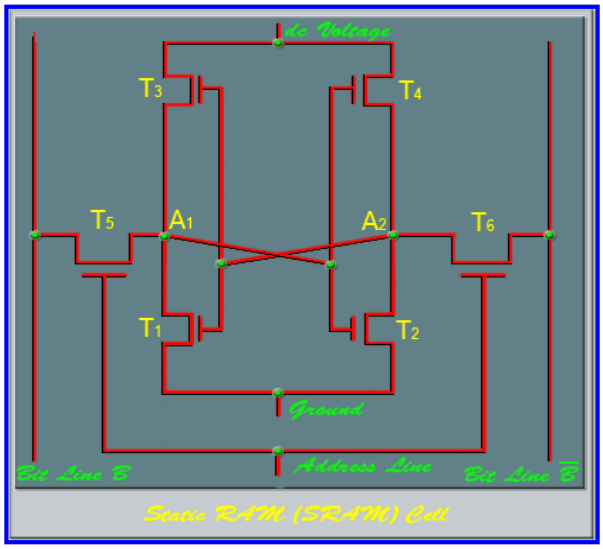
For the write operation, a voltage signal is applied to the bit line B, a high voltage represents 1 and a low voltage represents 0. A signal is then applied to the address line, which will turn on the transistor T, allowing a charge to be transferred to the capacitor.

For the read operation, when a signal is applied to the address line, the transistor T turns on and the charge stored on the capacitor is fed out onto the bit line B.

Static RAM (SRAM):

In an SRAM, binary values are stored using traditional flip-flop constructed with the help of transistors. A static RAM will hold its data as long as power is supplied to it.

A typical SRAM constructed with transistors is shown in the figure.



Click on Image To View Large Image

Concept of Memory Print this page

<< Previous | First | Last | Next >>

Four transistors (T_1, T_2, T_3, T_4) are cross connected in an arrangement that produces a stable logic state.

In logic state 1, point A_1 is high and point A_2 is low; in this state T_1 and T_4 are off, and T_2 and T_3 are on .

In logic state 0, point A_1 is low and point A_2 is high; in this state T_1 and T_4 are on, and T_2 and T_3 are off .

Both states are stable as long as the dc supply voltage is applied.

The address line is used to open or close a switch which is nothing but another transistor. The address line controls two transistors(T_5 and T_6).

When a signal is applied to this line, the two transistors are switched on, allowing a read or write operation.

For a write operation, the desired bit value is applied to line B, and its complement is applied to line $\overline{B}$. This forces the four transistors(T_1, T_2, T_3, T_4) into the proper state.

For a read operation, the bit value is read from the line B. When a signal is applied to the address line, the signal of point A_1 is available in the bit line B.

>> Previous | First | Last | Next >>

SRAM Versus DRAM :

- Both static and dynamic RAMs are volatile, that is, it will retain the information as long as power supply is applied.
- A dynamic memory cell is simpler and smaller than a static memory cell. Thus a DRAM is more dense, i.e., packing density is high(more cell per unit area). DRAM is less expensive than corresponding SRAM.
- DRAM requires the supporting refresh circuitry. For larger memories, the fixed cost of the refresh circuitry is more than compensated for by the less cost of DRAM cells
- SRAM cells are generally faster than the DRAM cells. Therefore, to construct faster memory modules(like cache memory) SRAM is used.

Internal Organization of Memory Chips

A memory cell is capable of storing 1-bit of information. A number of memory cells are organized in the form of a matrix to form the memory chip. One such organization is shown in the Figure.

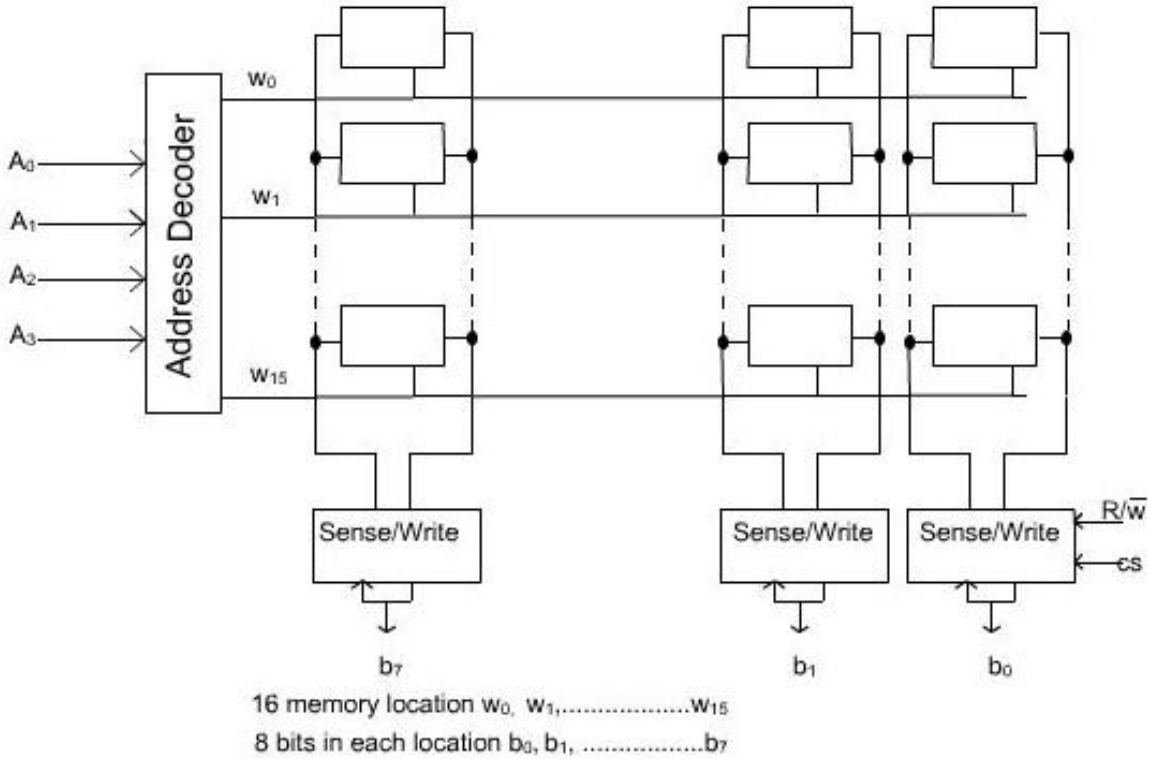


Figure : 16 X 8 Memory Organization

Concept of Memory

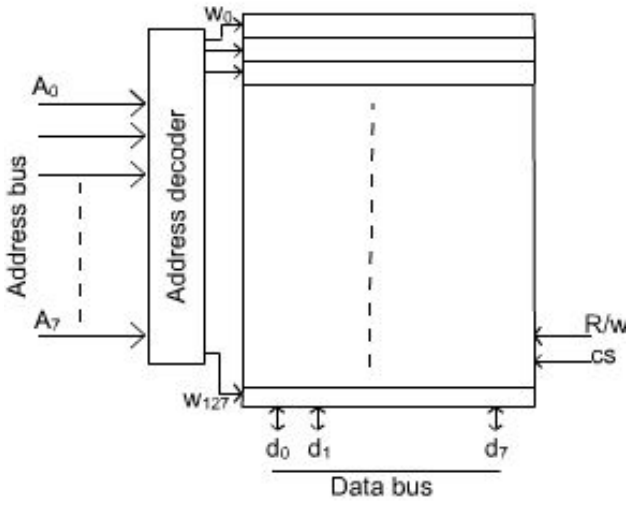
Each row of cells constitutes a memory word, and all cell of a row are connected to a common line which is referred as word line. An address decoder is used to drive the word line. At a particular instant, one word line is enabled depending on the address present in the address bus. The cells in each column are connected by two lines. These are known as bit lines. These bit lines are connected to data input line and data output line through a Sense/ Write circuit. During a Read operation, the Sense/Write circuit sense, or read the information stored in the cells selected by a word line and transmit this information to the output data line. During a write operation, the sense/write circuit receive information and store it in the cells of the selected word.

A memory chip consisting of 16 words of 8 bits each, usually referred to as 16 x 8 organization. The data input and data output line of each Sense/Write circuit are connected to a single bidirectional data line in order to reduce the pin required. For 16 words, we need an address bus of size 4. In addition to address and data lines, two control lines, $R/\overline{W}$ and CS, are provided. The $R/\overline{W}$ line is to used to specify the required operation about read or write. The CS (Chip Select) line is required to select a given chip in a multi chip memory system.

Consider a slightly larger memory unit that has 1K (1024) memory cells...

128 x 8 memory chips:

If it is organised as a 128 x 8 memory chips, then it has got 128 memory words of size 8 bits. So the size of data bus is 8 bits and the size of address bus is 7 bits ($2^7 = 128$).



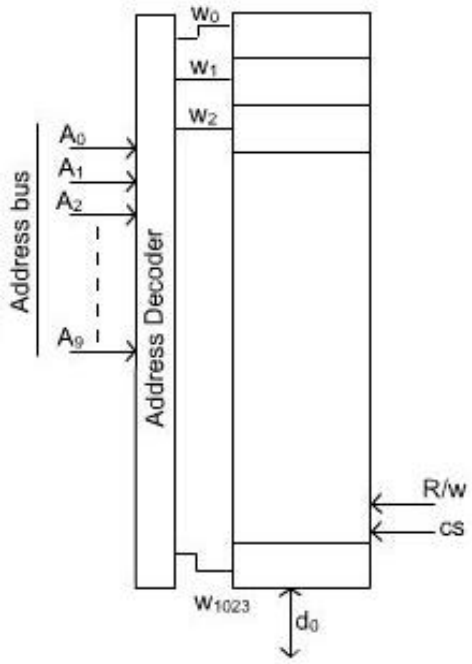
1024 x 1 memory chips:

If it is organized as a 1024 x 1 memory chips, then it has got 1024 memory words of size 1 bit only.

Therefore, the size of data bus is 1 bit and the size of address bus is 10 bits ($2^{10} = 1024$).

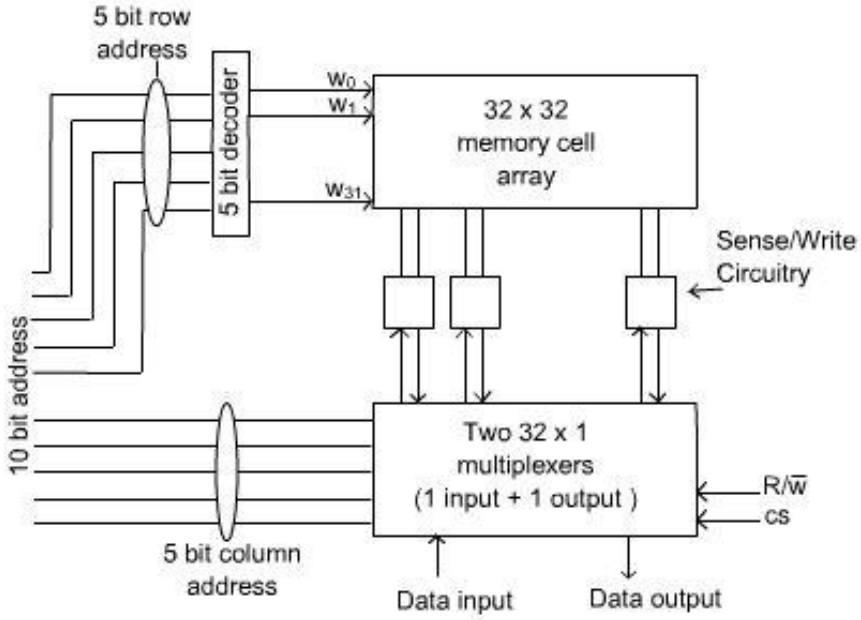
A particular memory location is identified by the contents of memory address bus. A decoder is used to decode the memory address. There are two ways of decoding of a memory address depending upon the organization of the memory module.

In one case, each memory word is organized in a row. In this case whole memory address bus is used together to decode the address of the specified location.



In second case, several memory words are organized in one row. In this case, address bus is divided into two gropus.

One group is used to form the row address and the second group is used to form the column address. Consider the memory organization of 1024 x 1 memory chip. The required 10-bit address is divided into two groups of 5 bits each to form the row and column address of the cell array. A row address selects a row of 32 cells, all of which are accessed in parallel. However, according to the column address, only one of these cells is connected to the external data line via the input output multiplexers.

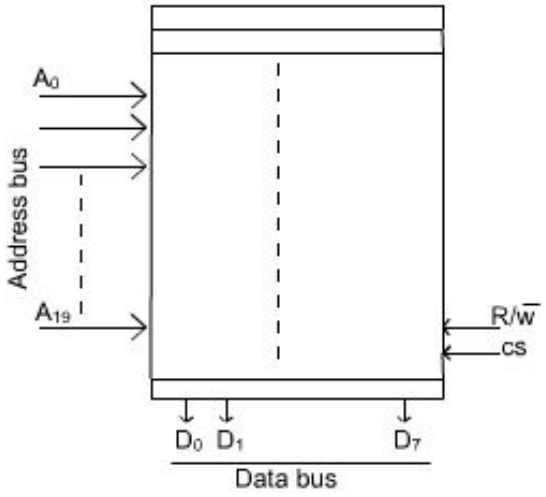


Organization of 1k x 1 memory chip

Concept of Memory

The commercially available memory chips contain a much larger number of cells. As for example, a memory unit of 1MB (mega byte) size, organised as 1M x 8, contains $2^{20} \times 8$ memory cells. It has got memory location and each memory location contains 8 bits information. The size of address bus is 20 and the size of data bus is 8.

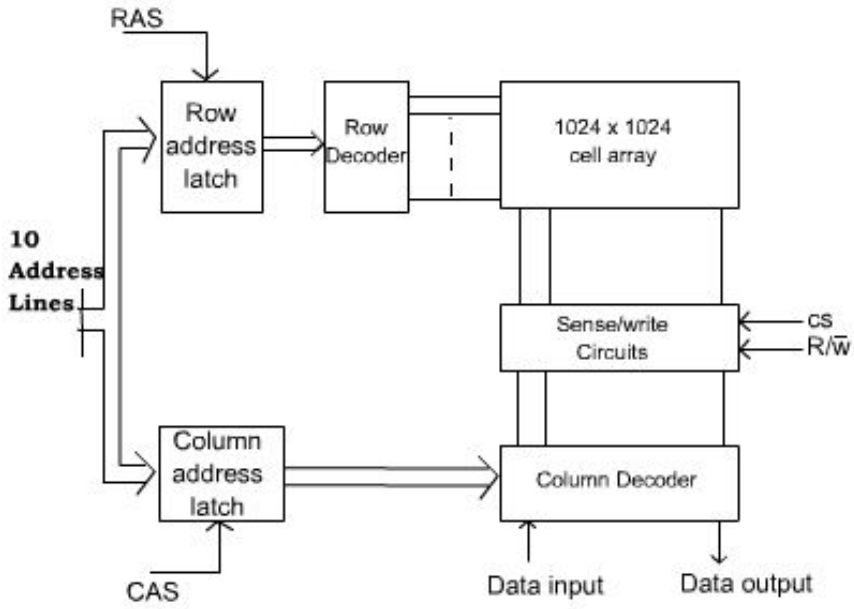
The number of pins of a memory chip depends on the data bus and address bus of the memory module. To reduce the number of pins required for the chip, we use another scheme for address decoding. The cells are organized in the form of a square array. The address bus is divided into two groups, one for column address and other one is for row address. In this case, high- and low-order 10 bits of 20-bit address constitute of row and column address of a given cell, respectively. In order to reduce the number of pin needed for external connections, the row and column addresses are multiplexed on ten pins. During a Read or a Write operation, the row address is applied first. In response to a signal pulse on the **Row Address Strobe (RAS)** input of the chip, this part of the address is loaded into the row address latch.



1 MB(Mega byte) memory chip

All cell of this particular row is selected. Shortly after the row address is latched, the column address is applied to the address pins. It is loaded into the column address latch with the help of Column Address Strobe (CAS) signal, similar to RAS. The information in this latch is decoded and the appropriate **Sense/Write** circuit is selected.

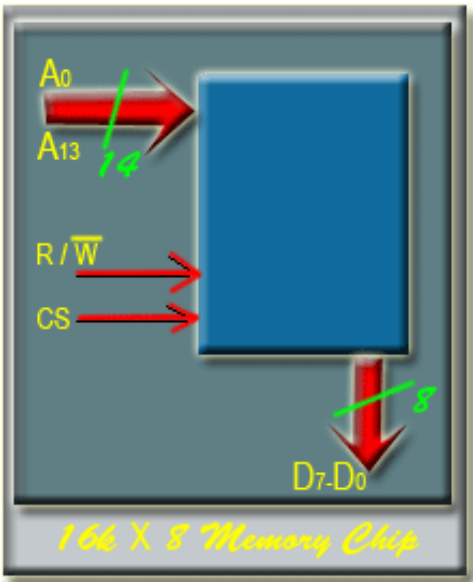
For a Write operation, the information at the input lines are transferred to the selected circuits.



Organization of a 1M x 1 Memory chip

Concept of Memory

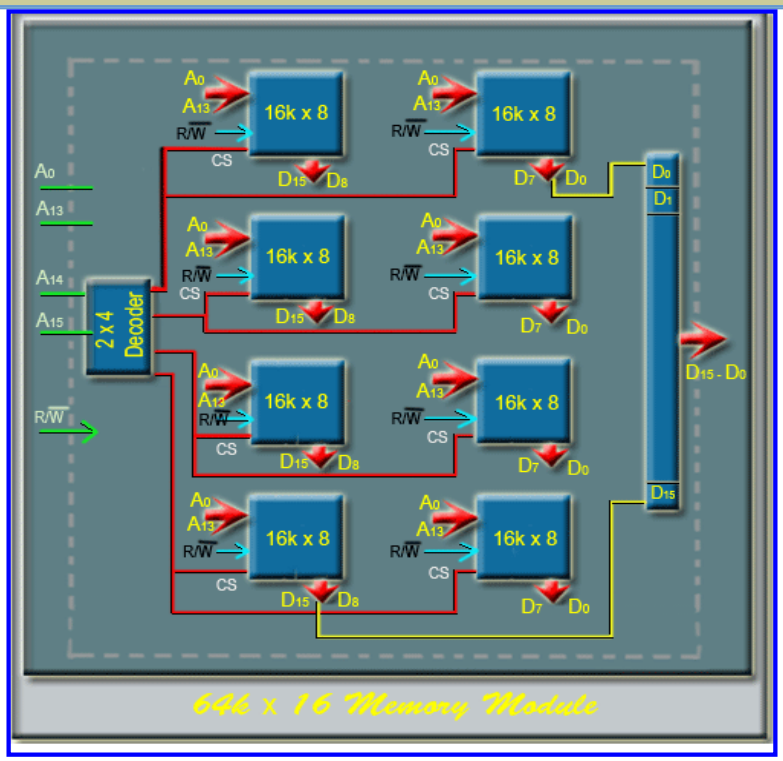
Now we discuss the design of memory subsystem using memory chips. Consider a memory chips of capacity 16K x 8. The requirement is to design a memory subsystem of capacity 64K x 16. Each memory chip has got eight lines for data bus, but the data bus size of memory subsytem is 16 bits.



The total requiremet is for 64K memory location, so four such units are required to get the 64K memory location. For 64K memory location, the size of address bus is 16. On the other hand, for 16K memory location, size of address bus is 14 bits.

Each chip has a control input line called Chip Select (CS). A chip can be enabled to accept data input or to place the data on the output bus by setting its Chip Select input to 1. The address bus for the 64K memory is 16 bits wide. The high order two bits of the address are decoded to obtain the four chip select control signals. The remaining 14 address bits are connected to the address lines of all the chips. They are used to access a specific location inside each chip of the selected row. The $R/\overline{W}$ inputs of all chips are tied together to provide a common $READ/\overline{WRITE}$ control.

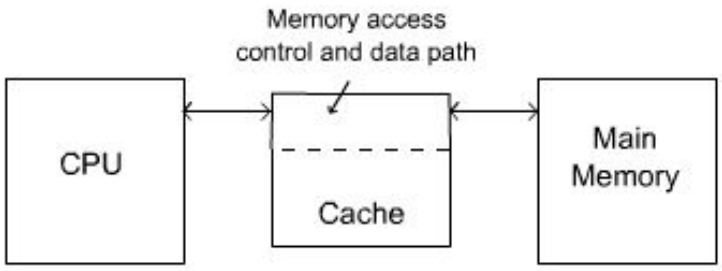
Concept of Memory



Click on Image To View Large Image

Cache Memory

Analysis of large number of programs has shown that a number of instructions are executed repeatedly. This may be in the form of a simple loops, nested loops, or a few procedures that repeatedly call each other. It is observed that many instructions in each of a few localized areas of the program are repeatedly executed, while the remainder of the program is accessed relatively less. This phenomenon is referred to as locality of reference.



Cache memory between the CPU and the main memory

Now, if it can be arranged to have the active segments of a program in a fast memory, then the tolal execution time can be significantly reduced. It is the fact that CPU is a faster device and memory is a relatively slower device. Memory access is the main bottleneck for the performance efficiency. If a faster memory device can be inserted between main memory and CPU, the efficiency can be increased. The faster memory that is inserted between CPU and Main Memory is termed as Cache memory. To make this arrangement effective, the cache must be considerably faster than the main memory, and typically it is 5 to 10 time faster than the main memory. This approach is more economical than the use of fast memory device to implement the entire main memory. This is also a feasible due to the locality of reference that is present in most of the program, which reduces the frequent data transfer between main memory and cache memory.

Operation of Cache Memory

The memory control circuitry is designed to take advantage of the property of locality of reference. Some assumptions are made while designing the memory control circuitry:

1. The CPU does not need to know explicitly about the existence of the cache.
2. The CPU simply makes Read and Write request. The nature of these two operations are same whether cache is present or not.
3. The address generated by the CPU always refer to location of main memory.
4. The memory access control circuitry determines whether or not the requested word currently exists in the cache.

When a Read request is received from the CPU, the contents of a block of memory words containing the location specified are transferred into the cache. When any of the locations in this block is referenced by the program, its contents are read directly from the cache.

The cache memory can store a number of such blocks at any given time.

The correspondence between the Main Memory Blocks and those in the cache is specified by means of a mapping function.

When the cache is full and a memory word is referenced that is not in the cache, a decision must be made as to which block should be removed from the cache to create space to bring the new block to the cache that contains the referenced word. **Replacement algorithms** are used to make the proper selection of block that must be replaced by the new one.

When a write request is received from the CPU, there are two ways that the system can proceed. In the first case, the cache location and the main memory location are updated simultaneously. This is called the **store through method** or **write through method**.

The alternative is to update the cache location only. During replacement time, the cache block will be written back to the main memory. If there is no new write operation in the cache block, it is not required to write back the cache block in the main memory. This information can be kept with the help of an associated bit. This bit it set while there is a write operation in the cache block. During replacement, it checks this bit, if it is set, then write back the cache block in main memory otherwise not. This bit is known as **dirty bit**. If the bit gets dirty (set to one), writting to main memory is required.

This write through method is simpler, but it results in unnecessary write operations in the main memory when a given cache word is updated a number of times during its cache residency period.

Consider the case where the addressed word is not in the cache and the operation is a read. First the block of the words is brought to the cache and then the requested word is forwarded to the CPU. But it can be forwarded to the CPU as soon as it is available to the cache, instead of whole block to be loaded into the cache. This is called load through, and there is some scope to save time while using load through policy.

During a write operation, if the address word is not in the cache, the information is written directly into the main memory. A write operation normally refers to the location of data areas and the property of locality of reference is not as pronounced in accessing data when write operation is involved. Therefore, it is not advantageous to bring the data block to the cache when there a write operation, and the addressed word is not present in cache.

Mapping Functions

The mapping functions are used to map a particular block of main memory to a particular block of cache. This mapping function is used to transfer the block from main memory to cache memory. Three different mapping functions are available:

Direct mapping:

A particular block of main memory can be brought to a particular block of cache memory. So, it is not flexible.

Associative mapping:

In this mapping function, any block of Main memory can potentially reside in any cache block position. This is much more flexible mapping method.

Block-set-associative mapping:

In this method, blocks of cache are grouped into sets, and the mapping allows a block of main memory to reside in any block of a specific set. From the flexibility point of view, it is in between to the other two methods.

All these three mapping methods are explained with the help of an example.

Consider a cache of 4096 (4K) words with a block size of 32 words. Therefore, the cache is organized as 128 blocks. For 4K words, required address lines are 12 bits. To select one of the block out of 128 blocks, we need 7 bits of address lines and to select one word out of 32 words, we need 5 bits of address lines. So the total 12 bits of address is divided for two groups, lower 5 bits are used to select a word within a block, and higher 7 bits of address are used to select any block of cache memory.

Let us consider a main memory system consisting 64K words. The size of address bus is 16 bits. Since the block size of cache is 32 words, so the main memory is also organized as block size of 32 words. Therefore, the total number of blocks in main memory is 2048 (2K x 32 words = 64K words). To identify any one block of 2K blocks, we need 11 address lines. Out of 16 address lines of main memory, lower 5 bits are used to select a word within a block and higher 11 bits are used to select a block out of 2048 blocks.

Number of blocks in cache memory is 128 and number of blocks in main memory is 2048, so at any instant of time only 128 blocks out of 2048 blocks can reside in cache menory. Therefore, we need mapping function to put a particular block of main memory into appropriate block of cache memory.

Direct Mapping Technique:

The simplest way of associating main memory blocks with cache block is the direct mapping technique. In this technique, block k of main memory maps into block k modulo m of the cache, where m is the total number of blocks in cache. In this example, the value of m is 128. In direct mapping technique, one particular block of main memory can be transfered to a particular block of cache which is derived by the modulo function.

Since more than one main memory block is mapped onto a given cache block position, contention may arise for that position. This situation may occurs even when the cache is not full. Contention is resolved by allowing the new block to overwrite the currently resident block. So the replacement algorithm is trivial.

The detail operation of direct mapping technique is as follows:

The main memory address is divided into three fields. The field size depends on the memory capacity and the block size of cache. In this example, the lower 5 bits of address is used to identify a word within a block. Next 7 bits are used to select a block out of 128 blocks (which is the capacity of the cache). The remaining 4 bits are used as a TAG to identify the proper block of main memory that is mapped to cache.

When a new block is first brought into the cache, the high order 4 bits of the main memory address are stored in four TAG bits associated with its location in the cache. When the CPU generates a memory request, the 7-bit block address determines the corresponding cache block. The TAG field of that block is compared to the TAG field of the address. If they match, the desired word specified by the low-order 5 bits of the address is in that block of the cache.

If there is no match, the required word must be accessed from the main memory, that is, the contents of that block of the cache is replaced by the new block that is specified by the new address generated by the CPU and correspondingly the TAG bit will also be changed by the high order 4 bits of the address. The whole arrangement for direct mapping technique is shown in the figure.

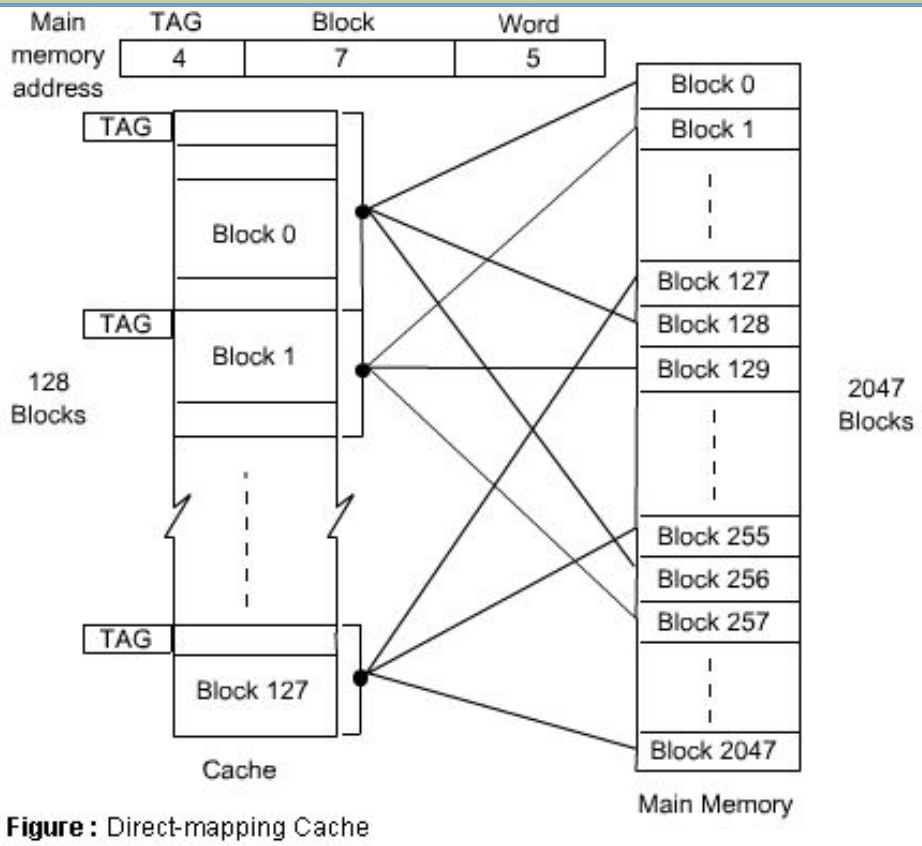


Figure : Direct-mapping Cache

Associated Mapping Technique:

In the associative mapping technique, a main memory block can potentially reside in any cache block position. In this case, the main memory address is divided into two groups, low-order bits identifies the location of a word within a block and high-order bits identifies the block. In the example here, 11 bits are required to identify a main memory block when it is resident in the cache , high-order 11 bits are used as TAG bits and low-order 5 bits are used to identify a word within a block. The TAG bits of an address received from the CPU must be compared to the TAG bits of each block of the cache to see if the desired block is present.

In the associative mapping, any block of main memory can go to any block of cache, so it has got the complete flexibility and we have to use proper replacement policy to replace a block from cache if the currently accessed block of main memory is not present in cache. It might not be practical to use this complete flexibility of associative mapping technique due to searching overhead, because the TAG field of main memory address has to be compared with the TAG field of all the cache block. In this example, there are 128 blocks in cache and the size of TAG is 11 bits. The whole arrangement of Associative Mapping Technique is shown in the **figure** (Next Page..).

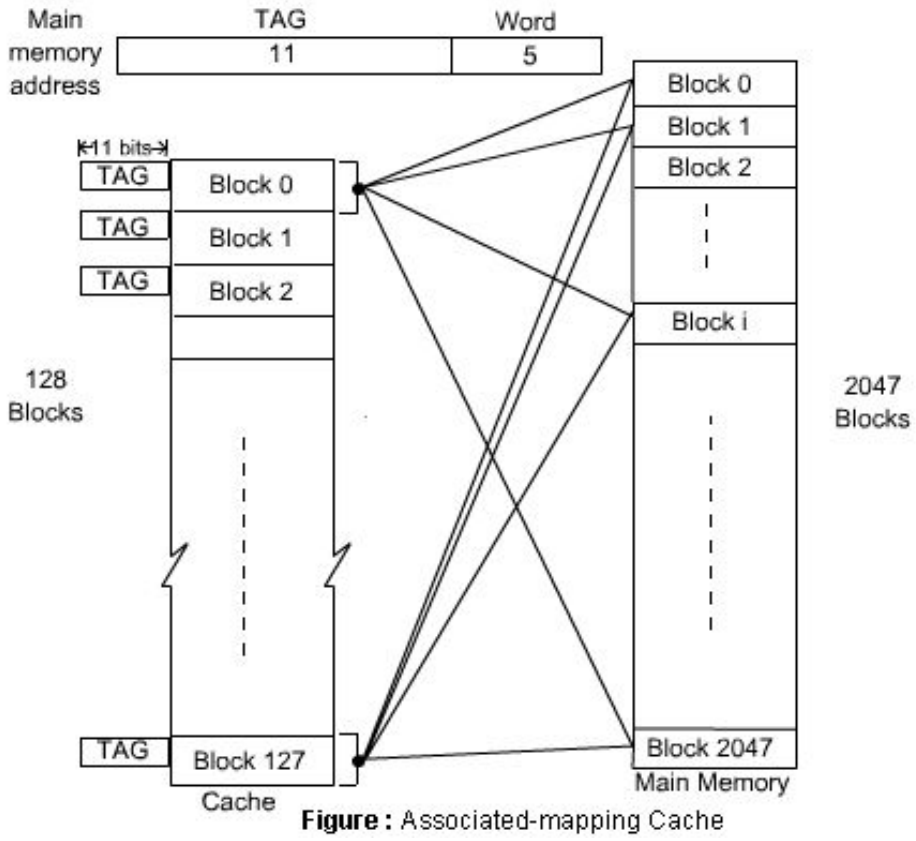


Figure : Associated-mapping Cache

Block-Set-Associative Mapping Technique:

This mapping technique is intermediate to the above two techniques. Blocks of the cache are grouped into sets, and the mapping allows a block of main memory to reside in any block of a specific set. Therefore, the flexibity of associative mapping is reduced from full freedom to a set of specific blocks. This also reduces the searching overhead, because the search is restricted to number of sets, instead of number of blocks. Also the contention problem of the direct mapping is eased by having a few choices for block replacement.

Consider the same cache memory and main memory organization of the previous example. Organize the cache with 4 blocks in each set. The TAG field of associative mapping technique is divided into two groups, one is termed as SET bit and the second one is termed as TAG bit. Since each set contains 4 blocks, total number of set is 32. The main memory address is grouped into three parts: low-order 5 bits are used to identifies a word within a block. Since there are total 32 sets present, next 5 bits are used to identify the set. High-order 6 bits are used as TAG bits.

The 5-bit set field of the address determines which set of the cache might contain the desired block. This is similar to direct mapping technique, in case of direct mapping, it looks for block, but in case of block-set-associative mapping, it looks for set. The TAG field of the address must then be compared with the TAGs of the four blocks of that set. If a match occurs, then the block is present in the cache; otherwise the block containing the addressed word must be brought to the cache. This block will potentially come to the cooresponding set only. Since, there are four blocks in the set, we have to choose appropriately which block to be replaced if all the blocks are occupied. Since the search is restricted to four block only, so the searching complexity is reduced. The whole arrangement of block-set-associative mapping technique is shown in the figure.

It is clear that if we increase the number of blocks per set, then the number of bits in SET field is reduced. Due to the increase of blocks per set, complexity of search is also increased. The extreme condition of 128 blocks per set requires no set bits and corrsponds to the fully associative mapping technique with **11 TAG** bits. The other extreme of one block per set is the direct mapping method.

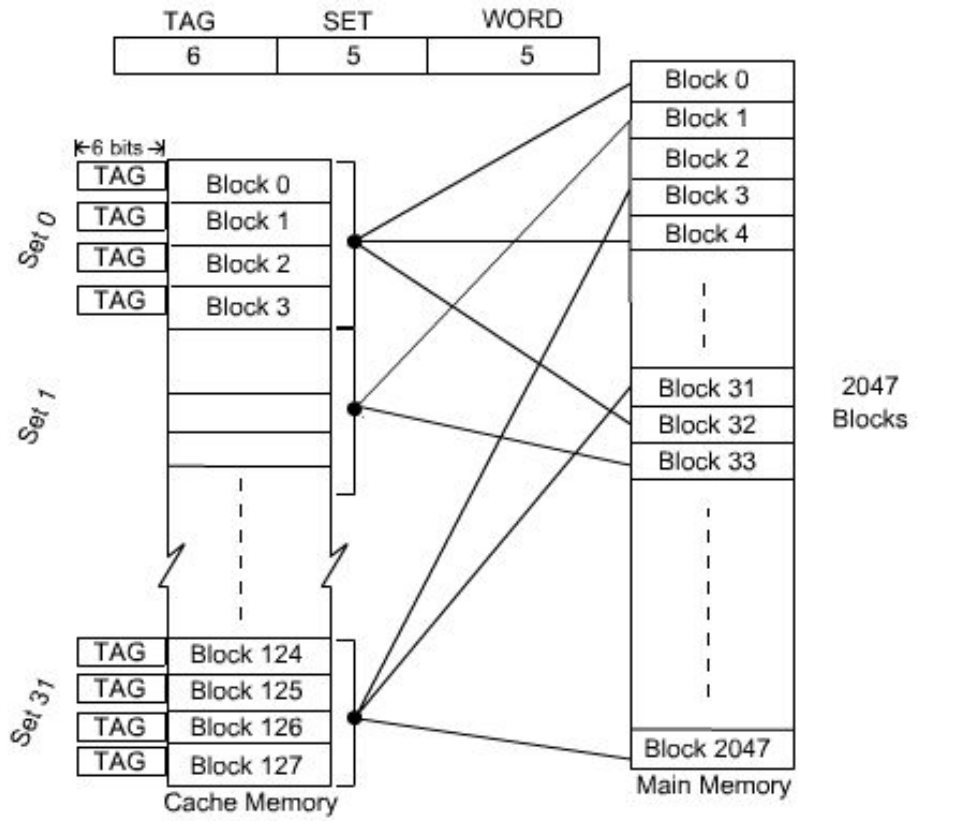


Figure : Block-set-Associated mapping Cache with 4 blocks per set

Replacement Algorithms

When a new block must be brought into the cache and all the positions that it may occupy are full, a decision must be made as to which of the old blocks is to be overwritten. In general, a policy is required to keep the block in cache when they are likely to be referenced in near future. However, it is not easy to determine directly which of the block in the cache are about to be referenced. The property of locality of reference gives some clue to design good replacement policy.

Least Recently Used (LRU) Replacement policy:

Since program usually stay in localized areas for reasonable periods of time, it can be assumed that there is a high probability that blocks which have been referenced recently will also be referenced in the near future. Therefore, when a block is to be overwritten, it is a good decision to overwrite the one that has gone for longest time without being referenced. This is defined as the least recently used (LRU) block. Keeping track of LRU block must be done as computation proceeds.

Consider a specific example of a four-block set. It is required to track the LRU block of this four-block set. A 2-bit counter may be used for each block.

When a hit occurs, that is, when a read request is received for a word that is in the cache, the counter of the block that is referenced is set to 0. All counters which values originally lower than the referenced one are incremented by 1 and all other counters remain unchanged.

When a miss occurs, that is, when a read request is received for a word and the word is not present in the cache, we have to bring the block to cache.

There are two possibilities in case of a miss:

> If the set is not full, the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are incremented by 1.
>
> If the set is full and a miss occurs, the block with the counter value 3 is removed , and the new block is put in its palce. The counter value is set to zero. The other three block counters are incremented by 1.

It is easy to verify that the counter values of occupied blocks are always distinct. Also it is trivial that highest counter value indicates least recently used block.

First In First Out (FIFO) replacement policy:

A reasonable rule may be to remove the oldest from a full set when a new block must be brought in. While using this technique, no updation is required when a hit occurs. When a miss occurs and the set is not full, the new block is put into an empty block and the counter values of the occupied block will be increment by one. When a miss occurs and the set is full, the block with highest counter value is replaced by new block and counter is set to 0, counter value of all other blocks of that set is incremented by 1. The overhead of the policy is less, since no updation is required during hit.

Random replacement policy:

The simplest algorithm is to choose the block to be overwritten at random. Interestingly enough, this simple algorithm has been found to be very effective in practice.

Main Memory

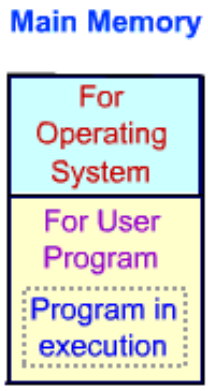
The main working principle of digital computer is *Von-Neumann* stored program principle. First of all we have to keep all the information in some storage, mainly known as main memory, and CPU interacts with the main memory only. Therefore, memory management is an important issue while designing a computer system.

On the otherhand, everything cannot be implemented in hardware, otherwise the cost of system will be very high. Therefore some of the tasks are performed by software program. Collection of such software programs are basically known as operating systems. So operating system is viewed as extended machine. Many more functions or instructions are implemented through software routine. The operating system is mainly memory resistant, i.e., the operating system is loaded into main memory.

Due to that, the main memory of a computer is divided into two parts. One part is reserved for operating system. The other part is for user program. The program currently being executed by the CPU is loaded into the user part of the memory.

In a uni-programming system, the program currently being executed is loaded into the user part of the memory.

In a multiprogramming system, the user part of memory is subdivided to accomodate multiple process. The task of subdivision is carried out dynamically by opearting system and is known as memory management.



Efficient memory management is vital in a multiprogramming system. If only a few process are in memory, then for much of the time all of the process will be waiting for I/O and the processor will idle. Thus memory needs to be allocated efficiently to pack as many processes into main memory as possible.

When memory holds multiple processes, then the process can move from one process to another process when one process is waiting. But the processor is so much faster then I/O that it will be common for all the processes in memory to be waiting for I/O. Thus, even with multiprogramming, a processor could be idle most of the time.

Due to the speed mismatch of the processor and I/O device, the status at any point in time is reffered to as a state.

There are five defined state of a process as shown in the figure.

When we start to execute a process, it is placed in the process queue and it is in the new state. As resources become available, then the process is placed in the ready queue.

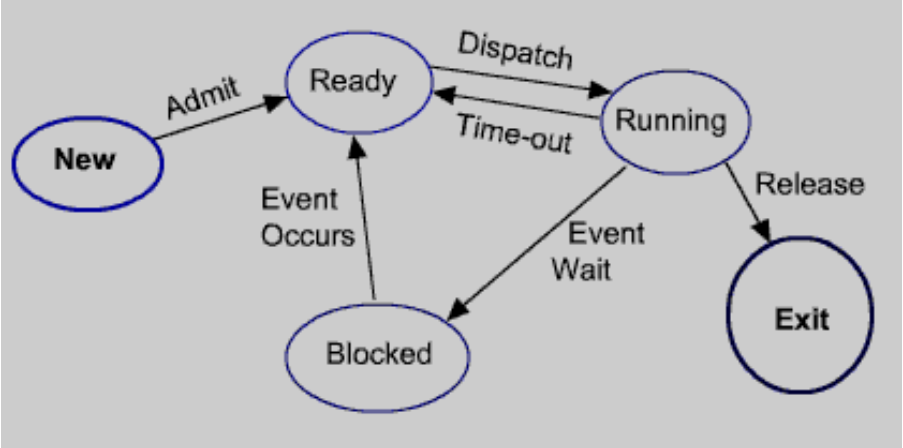


Figure : Five State process model

1. **New** : A program is admitted by the scheduler, but not yet ready to execute. The operating system will initialize the process by moving it to the ready state.
2. **Ready** : The process is ready to execute and is waiting access to the processor.
3. **Running** : The process is being executed by the processor. At any given time, only one process is in running state.
4. **Waiting** : The process is suspended from execution, waiting for some system resource, such as I/O.
5. **Exit** : The process has terminated and will be destroyed by the operating system.

The processor alternates between executing operating system instructions and executing user processes. While the operating system is in control, it decides which process in the queue sholud be executed next.

A process being executed may be suspended for a variety of reasons. If it is suspended because the process requests I/O, then it is places in the appropriate I/O queue. If it is suspended because of a timeout or because the operating system must attend to processing some of it's task, then it is placed in ready state.

We know that the information of all the process that are in execution must be placed in main memory. Since there is fix amount of memory, so memory management is an important issue.

Memory Management

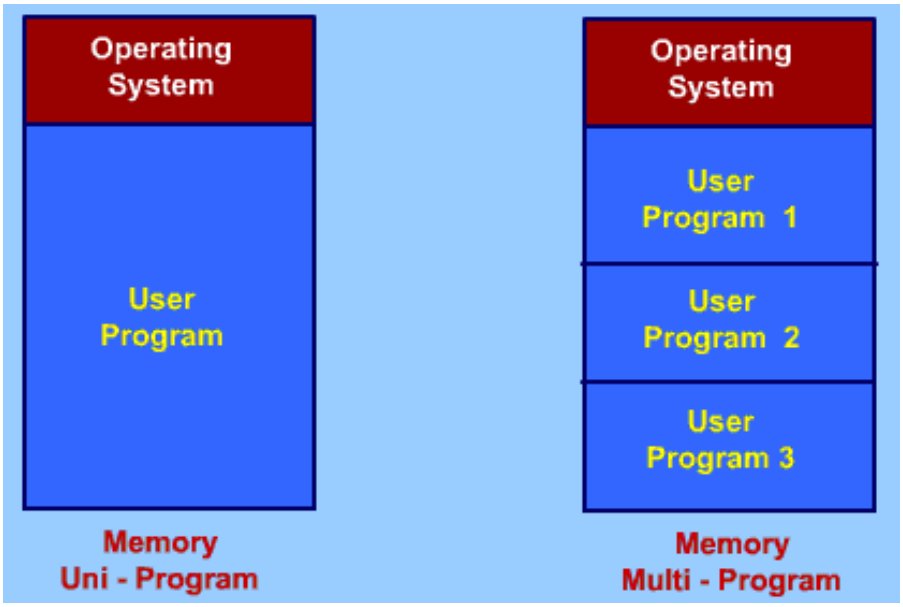
In an uniprogramming system, main memory is divided into two parts : one part for the operating system and the other part for the program currently being executed.

In multiprogramming system, the user part of memory is subdivided to accomodate multiple processes.

The task of subdivision is carried out dynamically by the operating system and is known as ***memory management***.

In uniprogramming system, only one program is in execution. After complition of one program, another program may start.

In general, most of the programs involve I/O operation. It must take input from some input device and place the result in some output device.



To utilize the idle time of CPU, we are shifting the paradigm from uniprogram environment to multiprogram environment.

Since the size of main memory is fixed, it is possible to accomodate only few process in the main memory. If all are waiting for I/O operation, then again CPU remains idle.

To utilize the idle time of CPU, some of the process must be off loaded from the memory and new process must be brought to this memory place. This is known swapping.

What is swapping :

1. The process waiting for some I/O to complete, must stored back in disk.
2. New ready process is swapped in to main memory as space becomes available.
3. As process completes, it is moved out of main memory.
4. If none of the processes in memory are ready,
   - Swapped out a block process to intermediate queue of blocked process.
   - Swapped in a ready process from the ready queue.

But swapping is an I/O process, so it also takes time. Instead of remain in idle state of CPU, sometimes it is advantageous to swapped in a ready process and start executing it.

The main question arises where to put a new process in the main memory. It must be done in such a way that the memory is utilized properly.

Partitioning

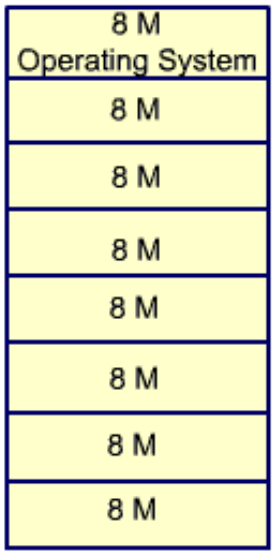
Splitting of memory into sections to allocate processes including operating system. There are two scheme for partitioning :

- Fixed size partitions
- Variable size partitions

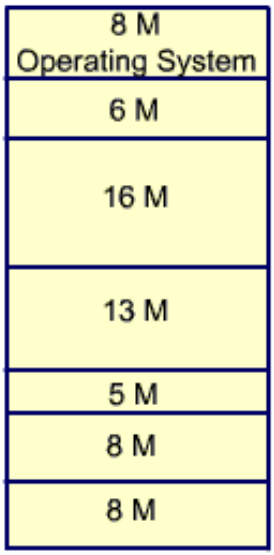
Fixed sized pertitions:

The mamory is partitioned to fixed size partition. Although the partitions are of fixed size, they need not be of equal size.

There is a problem of wastage of memory in fixed size even with unequal size. When a process is brought into memory, it is placed in the smallest available partition that will hold it.



(a) Equal Size Partitions



(b) Un-equal Size Partitions

Even with the use of unequal size of partitions, there will be wastage of memory. In most cases, a process will not require exactly as much memory as provided by the partition.

For example, a process that require 5-MB of memory would be placed in the 6-MB partition which is the smallest available partition. In this partition, only 5-MB is used, the remaining 1-MB can not be used by any other process, so it is a wastage. Like this, in every partition we may have some unused memory. *The unused portion of memory in each partition is termed as hole.*

Variable size Partition:

When a processe is brought into memory, it is allocated exactly as much memory as it requires and no more. In this process it leads to a hole at the end of the memory, which is too small to use. It seems that there will be only one hole at the end, so the waste is less.

But, this is not the only hole that will be present in variable size partition. When all processes are blocked then swap out a process and bring in another process. The new swapped in process may be smaller than the swapped out process. Most likely we will not get two process of same size. So, it will create another whole. If the swap- out and swap-in is occuring more time, then more and more hole will be created, which will lead to more wastage of memory.

There are two simple ways to slightly remove the problem of memory wastage:

Coalesce : Join the adjacent holes into one large hole , so that some process can be accomodated into the hole.

Compaction : From time to time go through memory and move all hole into one free block of memory.

During the execution of process, a process may be swapped in or swapped out many times. it is obvious that a process is not likely to be loaded into the same place in main memory each time it is swapped in. Further more if compaction is used, a process may be shiefted while in main memory.

A process in memory consists of instruction plus data. The instruction will contain address for memory locations of two types:

- Address of data item
- Address of instructions used for branching instructions

These addresses will change each time a process is swapped in. To solve this problem, a distinction is made between logical address and physical address.

- *Logical address is expressed as a location relative to the begining of the program. Instructions in the program contains only logical address.*
- *Physical address is an actual location in main memory.*

When the processor executes a process, it automatically converts from logical to physical address by adding the current starting location of the process, called it's base address to each logical address.

Every time the process is swapped in to main memory, the base address may be diffrent depending on the allocation of memory to the process.

Consider a main memory of *2-MB* out of which *512-KB* is used by the Operating System. Consider three process of size *425-KB*, *368-KB* and *470-KB* and these three process are loaded into the memory. This leaves a hole at the end of the memory. That is too small for a fourth process. At some point none of the process in main memory is ready. The operating system swaps out process-2 which leaves sufficient room for new process of size *320-KB*. Since process-4 is smaller then process-2, another hole is created. Later a point is reached at which none of the processes in the main memory is ready, but proces-2, so process-1 is swapped out and process-2 is swapped in there. It will create another hole. In this way it will create lot of small holes in the momory system which will lead to more memory wastage.

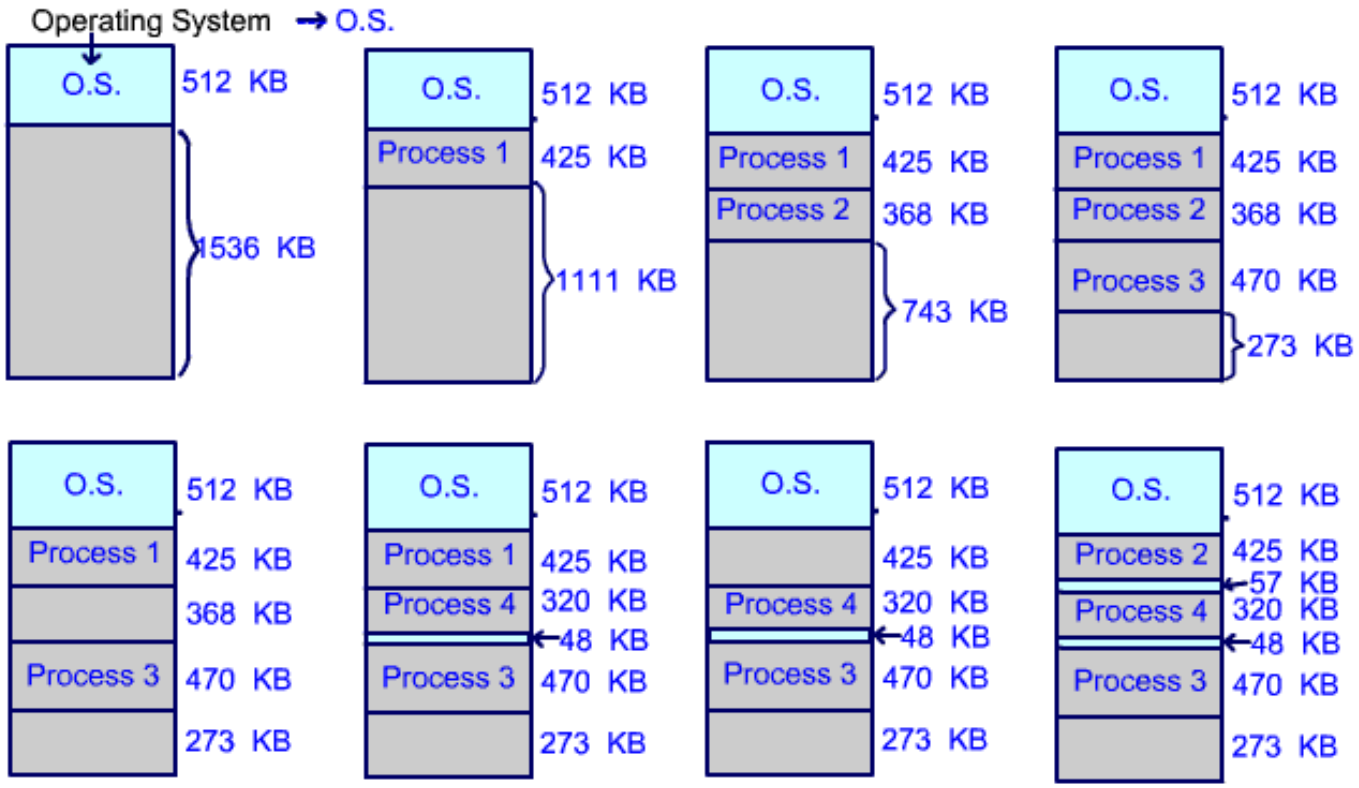


Figure : The effect of dynamic partitioning

Paging

Both unequal fixed size and variable size partitions are inefficient in the use of memory. It has been observed that both schemes lead to memory wastage. Therefore we are not using the memory efficiently.

There is another scheme for use of memory which is known as ***paging***.

In this scheme,

> *The memory is partitioned into equal fixed size chunks that are relatively small. This chunk of memory is known as frames or page frames.*
>
> *Each process is also divided into small fixed chunks of same size. The chunks of a program is known as pages.*

A page of a program could be assigned to available page frame.

In this scheme, the wastage space in memory for a process is a fraction of a page frame which corresponds to the last page of the program.

At a given point of time some of the frames in memory are in use and some are free. The list of free frame is maintained by the operating system.

Process A , stored in disk , consists of pages . At the time of execution of the process A, the operating system finds six free frames and loads the six pages of the process A into six frames.

These six frames need not be contiguous frames in main memory. The operating system maintains a page table for each process.

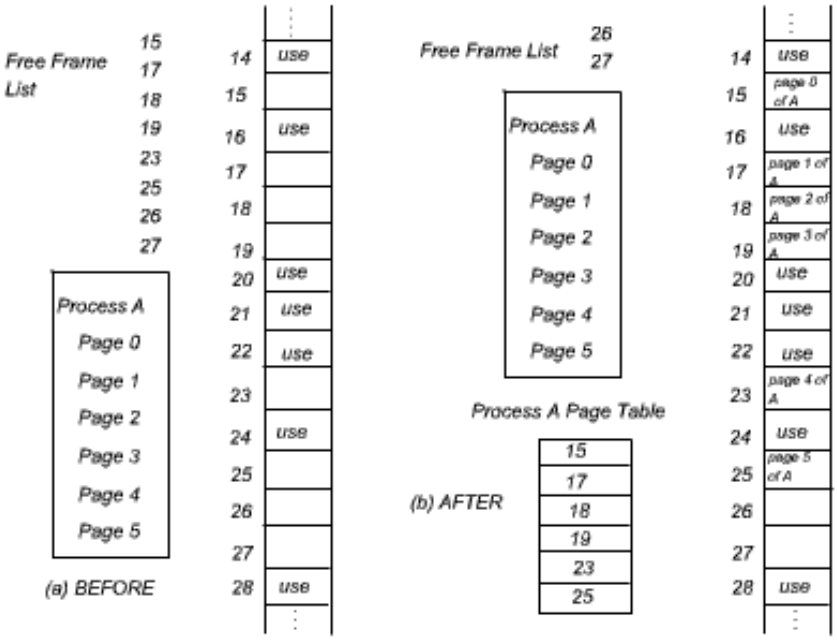
Within the program, each logical address consists of page number and a relative address within the page.

In case of simple partitioning, a logical address is the location of a word relative to the beginning of the program; the processor translates that into a physical address.

With paging, a logical address is a location of the word relative to the beginning of the page of the program, because the whole program is divided into several pages of equal length and the length of a page is same with the length of a page frame.

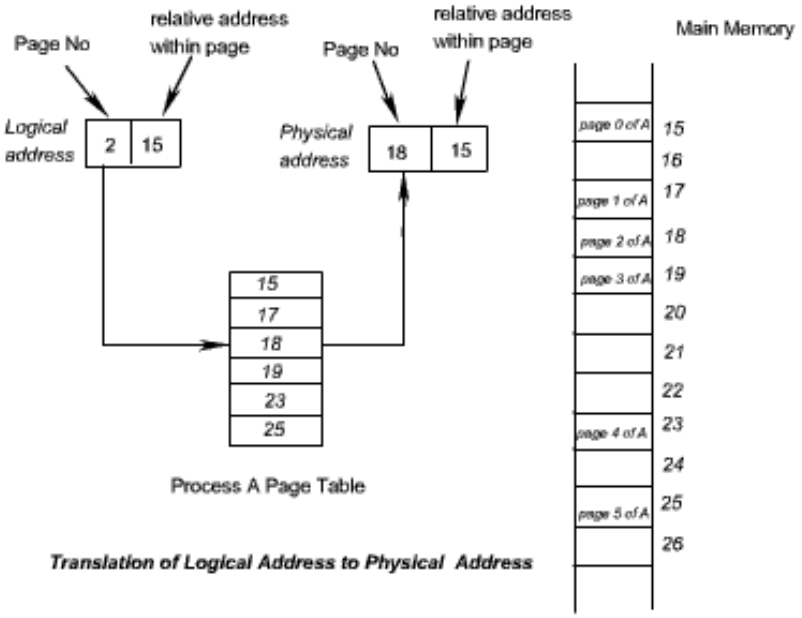
A logical address consists of page number and relative address within the page, the process uses the page table to produce the physical address which consists of frame number and relative address within the frame.

The figure on next page shows the allocation of frames to a new process in the main memory. A page table is maintained for each process. This page table helps us to find the physical address in a frame which corresponds to a logical address within a process.



Allocation Of Free Frames

The conversion of logical address to physical address is shown in the figure for the Process A.



Translation of Logical Address to Physical Address

This approach solves the problems. Main memory is divided into many small equal size frames. Each process is divided into frame size pages. Smaller process requires fewer pages, larger process requires more. When a process is brought in, its pages are loaded into available frames and a page table is set up.

Virtual Memory

The concept of paging helps us to develop truly effective multiprogramming systems.

Since a process need not be loaded into contiguous memory locations, it helps us to put a page of a process in any free page frame. On the other hand, it is not required to load the whole process to the main memory, because the execution may be confined to a small section of the program. (eg. a subroutine).

It would clearly be wasteful to load in many pages for a process when only a few pages will be used before the program is suspended.

Instead of loading all the pages of a process, each page of process is brought in only when it is needed, i.e on demand. *This scheme is known as* ***demand paging***.

Demand paging also allows us to accommodate more process in the main memory, since we are not going to load the whole process in the main memory, pages will be brought into the main memory as and when it is required.

With demand paging, it is not necessary to load an entire process into main memory.

This concept leads us to an important consequence – It is possible for a process to be larger than the size of main memory. So, while developing a new process, it is not required to look for the main memory available in the machine. Because, the process will be divided into pages and pages will be brought to memory on demand.

Virtual Memory

Because a process executes only in main memory, so the main memory is referred to as real memory or physical memory.

A programmer or user perceives a much larger memory that is allocated on the disk. This memory is referred to as virtual memory. The program enjoys a huge virtual memory space to develop his or her program or software.

The execution of a program is the job of operating system and the underlying hardware. To improve the performance some special hardware is added to the system. This hardware unit is known as Memory Management Unit (MMU).

In paging system, we make a page table for the process. Page table helps us to find the physical address from virtual address.

The virtual address space is used to develop a process. The special hardware unit , called **Memory Management Unit (MMU)** translates virtual address to physical address. When the desired data is in the main memory, the CPU can work with these data. If the data are not in the main memory, the MMU causes the operating system to bring into the memory from the disk.

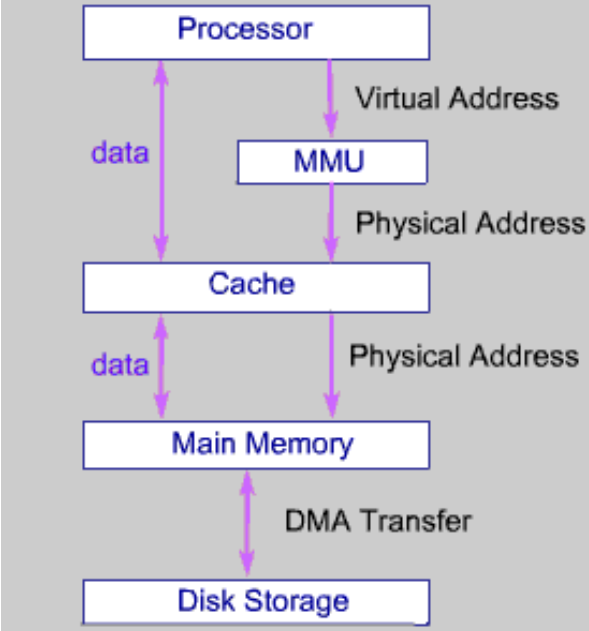


Figure : Virtual Memory Organization

Address Translation

The basic mechanism for reading a word from memory involves the translation of a virtual or logical address, consisting of page number and offset, into a physical address, consisting of frame number and offset, using a page table.

There is one page table for each process. But each process can occupy huge amount of virtual memory. But the virtual memory of a process cannot go beyond a certain limit which is restricted by the underlying hardware of the MMU. One of such component may be the size of the virtual address register.

The sizes of pages are relatively small and so the size of page table increases as the size of process increases. Therefore, size of page table could be unacceptably high.

To overcome this problem, most virtual memory scheme store page table in virtual memory rather than in real memory.

This means that the page table is subject to paging just as other pages are.

When a process is running, at least a part of its page table must be in main memory, including the page table entry of the currently executing page.

A virtual address translation method is shown in the figure on next page.

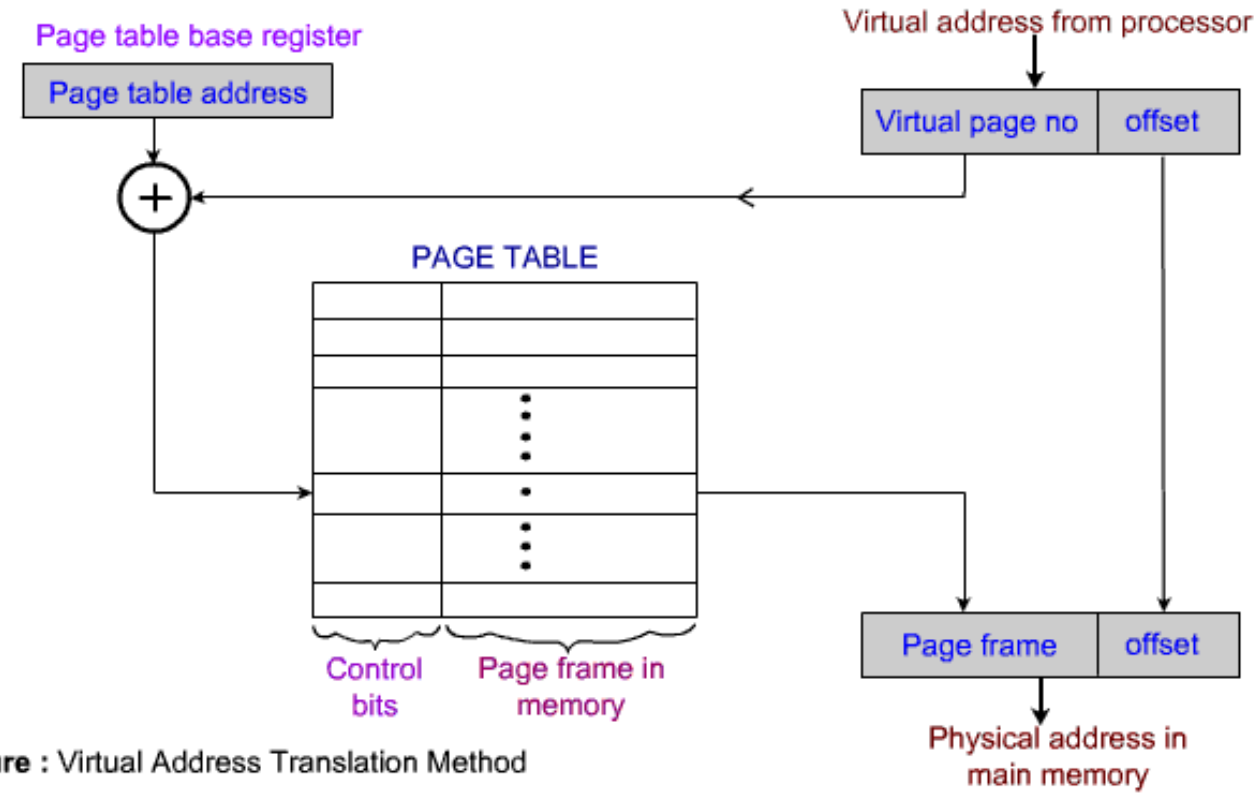


Figure : Virtual Address Translation Method

Each virtual address generated by the processor is interpreted as virtual page number (high order list) followed by an offset (lower order bits) that specifies the location of a particular word within a page. Information about the main memory location of each page kept in a page table.

Some processors make use of a two level scheme to organize large page tables.

In this scheme, there is a page directory, in which each entry points to a page table.

Thus, if the length of the page directory is X, and if the maximum length of a page table is Y, then the process can consist of up to X x Y pages.

Typically, the maximum length of page table is restricted to the size of one page frame.

Inverted page table structures

There is one entry in the hash table and the inverted page table for each real memory page rather than one per virtual page.

Thus a fixed portion of real memory is required for the page table, regardless of the number of processes or virtual page supported.

Because more than one virtual address may map into the hash table entry, a chaining technique is used for managing the overflow.

The hashing techniques results in chains that are typically short – either one or two entries.

The inverted page table in shown in the figure on next page...

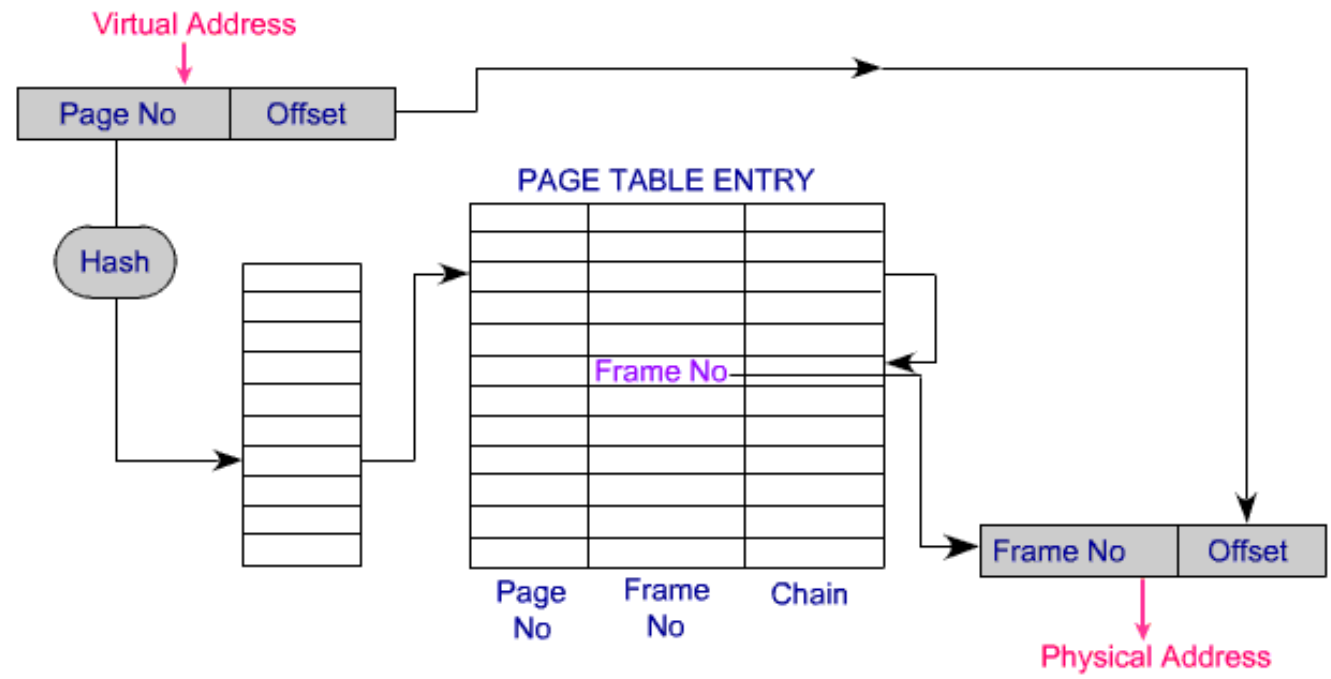


Figure : Inverted Page Table Structure

Translation Lookaside Buffer (TLB)

Every virtual memory reference can cause two physical memory accesses.

One to fetch the appropriate page table entry

One to fetch the desired data.

Thus a straight forward virtual memory scheme would have the effect of doubling the memory access time.

To overcome this problem, most virtual memory schemes make use of a special cache for page table entries, usually called *Translation Lookaside Buffer (TLB)*.

This cache functions in the same way as a memory cache and contains those page table entries that have been most recently used.

In addition to the information that constitutes a page table entry, the TLB must also include the virtual address of the entry.

The figure in next page shows a possible organization of a TLB where the associative mapping technique is used.

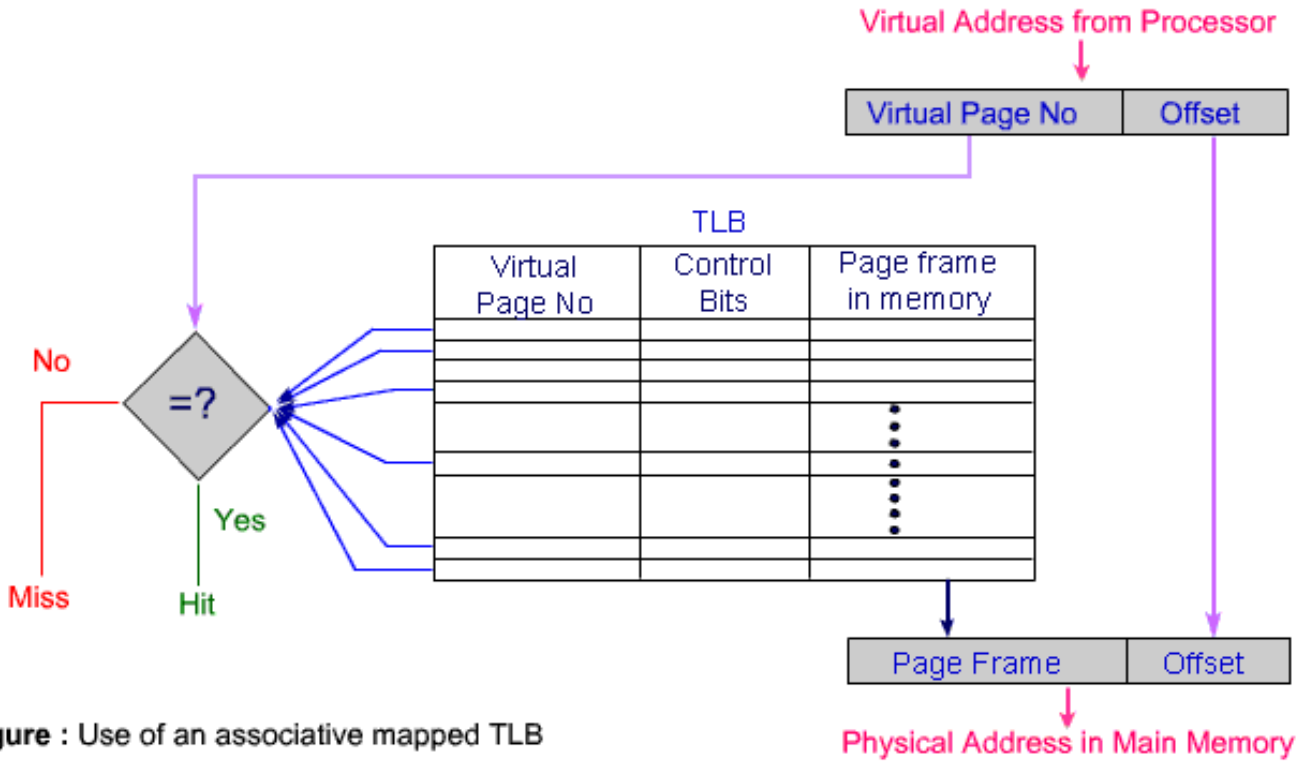


Figure : Use of an associative mapped TLB

Set-associative mapped TLBs are also found in commercial products.

An essential requirement is that the contents of the TLB be coherent with the contents of the page table in the main memory.

When the operating system changes the contents of the page table it must simultaneously invalidate the corresponding entries in the TLB. One of the control bits in the TLB is provided for this purpose.

Address Translation proceeds as follows:

- Given a virtual address, the MMU looks in the TLB for the reference page.
- If the page table entry for this page is found in the TLB, the physical address is obtained immediately.
- If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated.
- When a program generates an access request to a page that is not in the main memory, a page fault is said to have occurred.
- The whole page must be brought from the disk into the memory before access can proceed.
- When it detects a page fault, the MMU asks the operating system to intervene by raising an exception. (interrupt).
- Processing of active task is interrupted, and control is transferred to the operating system.
- The operating system then copies the requested page from the disk into the main memory and returns control to the interrupted task. Because a long delay occurs due to a page transfer takes place, the operating system may suspend execution of the task that caused the page fault and begin execution of another task whose page are in the main memory.

Module 4 : Instruction Set & Addressing

In this Module, we have three lectures, viz.

1. **Various addressing modes**
2. **Machine Instruction**
3. **Instruction Format**

Click the proper link on the left side for the lectures

We have examined the types of *operands* and *operations* that may be specified by *machine instructions*. Now we have to see how is the address of an operand specified, and how are the *bits* of an instruction organized to define the *operand addresses* and operation of that instruction.

Addressing Modes:

The most common addressing techniques are:

- **Immediate**
- **Direct**
- **Indirect**
- **Register**
- **Register Indirect**
- **Displacement**
- **Stack**

All computer architectures provide more than one of these *addressing modes*. The question arises as to how the *control unit* can determine which addressing mode is being used in a particular instruction. Several approaches are used. Often, different *opcodes* will use different addressing modes. Also, one or more bits in the *instruction format* can be used as a *mode field*. The value of the mode field determines which addressing mode is to be used.

What is the interpretation of *effective address*. In a system without virtual memory, the effective address will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the paging mechanism and is invisible to the programmer.

To explain the addressing modes, we use the following notation:

A	=	contents of an address field in the instruction that refers to a memory
R	=	contents of an address field in the instruction that refers to a register
EA	=	actual (effective) address of the location containing the referenced operand
(X)	=	contents of location X

Immediate Addressing:

The simplest form of addressing is immediate addressing, in which the operand is actually present in the instruction:

OPERAND = A

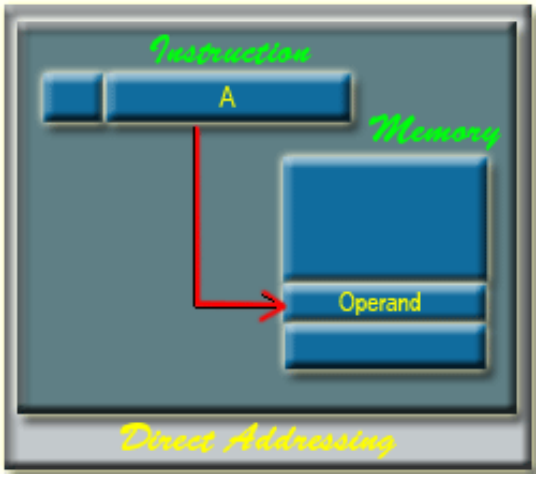
This mode can be used to define and use constants or set initial values of variables. The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the world length.

Direct Addressing:

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

EA = A

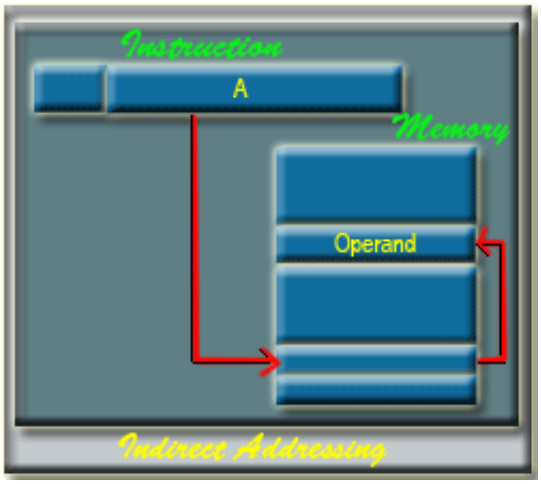
It requires only one memory reference and no special calculation.



Indirect Addressing:

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is know as indirect addressing:

EA = (A)

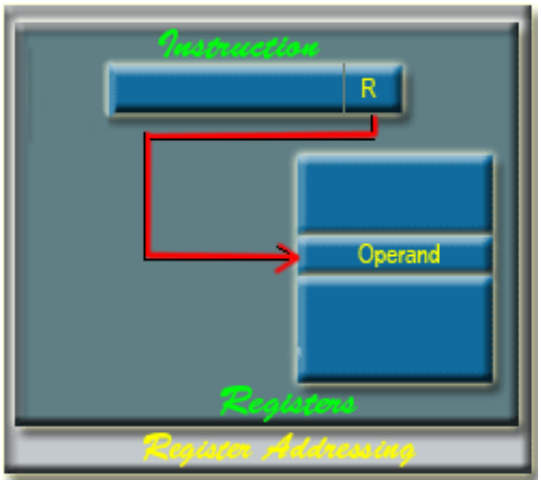


Register Addressing:

Register addressing is similar to direct addressing. The only difference is that the address field referes to a register rather than a main memory address:

EA = R

The advantages of register addressing are that only a small address field is needed in the instruction and no memory reference is required. The disadvantage of register addressing is that the address space is very limited.

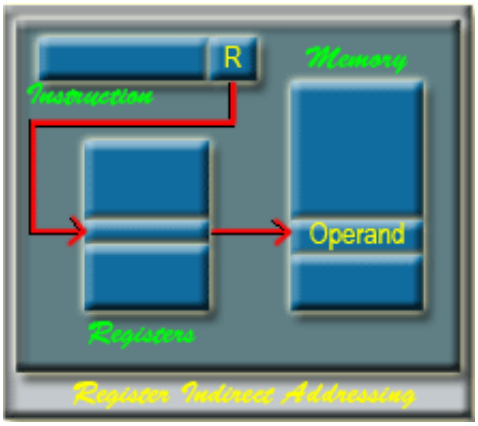


Register Indirect Addressing:

Register indirect addressing is similar to indirect addressing, except that the address field refers to a register instead of a memory location.
It requires only one memory reference and no special calculation.

EA = (R)

Register indirect addressing uses one less memory reference than indirect addressing. Because, the first information is available in a register which is nothing but a memory address. From that memory location, we use to get the data or information. In general, register access is much more faster than the memory access.



Diaplacement Addressing:

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing, which is broadly categorized as displacement addressing:

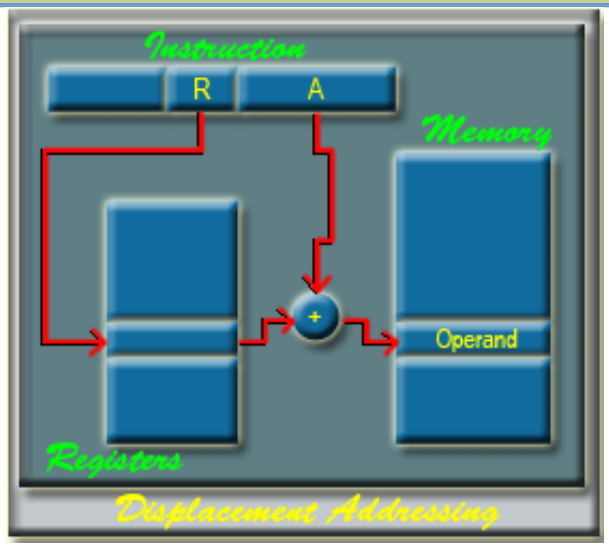
$$EA = A + (R)$$

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address.

Three of the most common use of displacement addressing are:

- Relative addressing
- Base-register addressing
- Indexing

Instruction Addressing



Relative Addressing:

For relative addressing, the implicitly referenced register is the *program counter* (PC). That is, the current instruction address is added to the address field to produce the EA. Thus, the effective address is a displacement relative to the address of the instruction.

Base-Register Addressing:

The reference register contains a memory address, and the address field contains a displacement from that address. The register reference may be *explicit* or *implicit*.

In some implimentation, a single segment/base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly.

Indexing:

The address field references a main memory address, and the reference register contains a positive displacement from that address. In this case also the register reference is sometimes explicit and sometimes implicit.

Generally index register are used for iterative tasks, it is typical that there is a need to increment or decrement the index register after each reference to it. Because this is such a common operation, some system will automatically do this as part of the same instruction cycle.

This is known as *auto-indexing*. We may get two types of auto-indexing:

-- one is auto-incrementing and the other one is
-- auto-decrementing.

If certain registers are devoted exclusively to indexing, then auto-indexing can be invoked implicitly and automatically. If general purpose register are used, the autoindex operation may need to be signaled by a bit in the instruction.

Auto-indexing using *increment* can be depicted as follows:

EA = A + (R)
R = (R) + 1

Auto-indexing using *decrement* can be depicted as follows:

EA = A + (R)
R = (R) - 1

In some machines, both *indirect addressing* and *indexing* are provided, and it is possible to employ both in the same instruction. There are two possibilities: The indexing is performed either before or after the indirection.

If indexing is performed after the indirection, it is termed *postindexing*

EA = (A) + (R)

First, the contents of the address field are used to access a memory location containing an address. This address is then indexed by the register value.

With preindexing, the indexing is performed before the indirection:

EA = (A + (R))

An address is calculated, the calculated address contains not the operand, but the address of the operand.

Stack Addressing:

A stack is a linear array or list of locations. It is sometimes referred to as a *pushdown list* or *last-in-first-out queue*. A stack is a reserved block of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.

The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.

The operation of a CPU is determine by the instruction it executes, referred to as machine instructions or computer instructions. The collection of different instructions is referred as the *instruction set of the CPU*.

Each instruction must contain the information required by the CPU for execution. The elements of an instruction are as follows:

Operation Code:
Specifies the operation to be performed (e.g., add, move etc.). The operation is specified by a binary code, know as the *operation code* or *opcode*.

Source operand reference:
The operation may involve one or more source operands; that is, operands that are inputs for the operation.

Result operand reference:
The operation may produce a result.

Next instruction reference:
This tells the CPU where to fetch the next instruction after the execution of this instruction is complete.

The next instruction to be fetched is located in main memory. But in case of virtual memory system, it may be either in main memory or secondary memory (disk). In most cases, the next instruction to be fetched immediately follow the current instruction. In those cases, there is no explicit reference to the next instruction.When an explicit reference is needed, then the main memory or virtual memory address must be given.

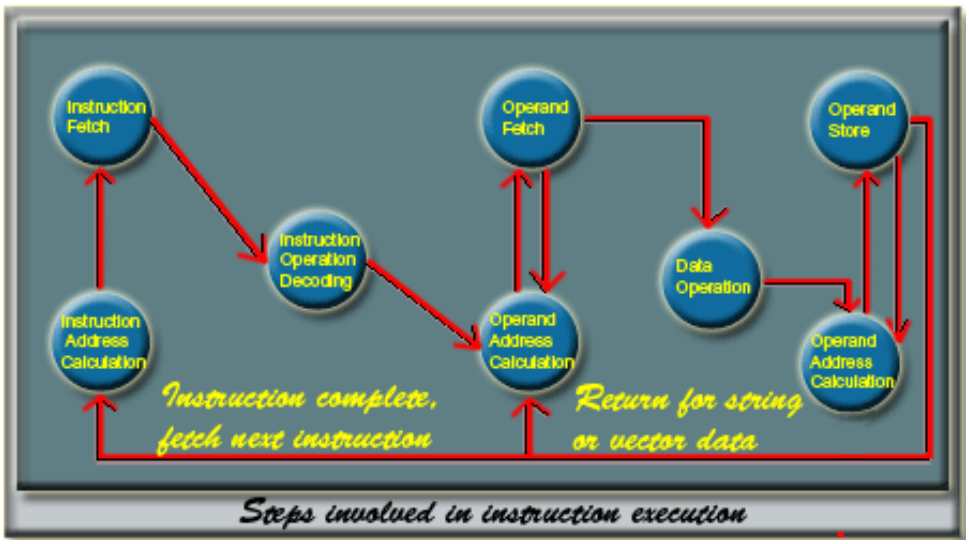
Source and result operands can be in one of the three areas:

- main or virtual memory,
- CPU register or
- I/O device.

Machine Instruction Print this page

<< Previous | First | Last | Next >>

The steps involved in instruction execution is shown in the figure-



>> Previous | First | Last | Next >>

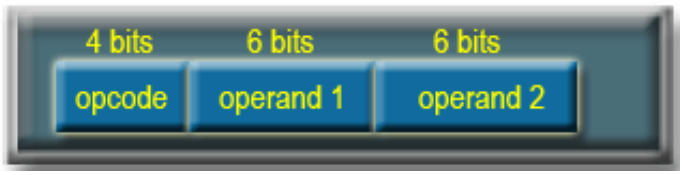
Instruction Representation

Within the computer, each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituent elements of the instruction. The instruction format is highly machine specific and it mainly depends on the machine architecture. A simple example of an instruction format is shown in the figure. It is assume that it is a 16-bit CPU. 4 bits are used to provide the operation code. So, we may have to 16 (2^4 = 16) different set of instructions. With each instruction, there are two operands. To specify each operands, 6 bits are used. It is possible to provide 64 (2^6 = 64) different operands for each operand reference.

It is difficult to deal with binary representation of machine instructions. Thus, it has become common practice to use a symbolic representation of machine instructions.

Opcodes are represented by abbreviations, called *mnemonics*, that indicate the operations. Common examples include:

ADD	Add
SUB	Subtract
MULT	Multiply
DIV	Division
LOAD	Load data from memory to CPU
STORE	Store data to memory from CPU.



A simple instruction format.

Operands are also represented symbolically. For example, the instruction

MULT R, X : R ← R x X

may mean multiply the value contained in the data location X by the contents of register R and put the result in register R

In this example, X refers to the address of a location in memory and R refers to a particular register.

Thus, it is possible to write a machine language program in symbolic form. Each symbolic opcode has a fixed binary representation, and the programmer specifies the location of each symbolic operand.

Instruction Types

The instruction set of a CPU can be categorized as follows:

Data Processing:
Arithmatic and Logic instructions Arithmatic instructions provide computational capabilities for processing numeric data. Logic (Boolean) instructions operate on the bits of a word as bits rather than as numbers. Logic instructions thus provide capabilities for processing any other type of data. There operations are performed primarily on data in CPU registers.

Data Storage:
Memory instructions Memory instructions are used for moving data between memory and CPU registers.

Data Movement:
I/O instructions I/O instructions are needed to transfer program and data into memory from storage device or input device and the results of computation back to the user.

Control:
Test and branch instructions

Test instructions are used to test the value of a data word or the status of a computation. Branch instructions are then used to branch to a different set of instructions depending on the decision made.

Number of Addresses

What is the maximum number of addresses one might need in an instruction? Most of the arithmantic and logic operations are either *unary* (one operand) or *binary* (two operands). Thus we need a maximum of two addresses to reference operands. The result of an operation must be stored, suggesting a third address. Finally after completion of an instruction, the next instruction must be fetched, and its address is needed.

This reasoning suggests that an instruction may require to contain *four address references:* two operands, one result, and the address of the next instruction. In practice, four address instructions are rare. Most instructions have one, two or three operands addresses, with the address of the next instruction being implicit (obtained from the program counter).

Instruction Set Design

One of the most interesting, and most analyzed, aspects of computer design is instruction set design. The instruction set defines the functions performed by the CPU. The instruction set is the programmer's means of controlling the CPU. Thus programmer requirements must be considered in designing the instruction set.

Most important and fundamental design issues:

Operation repertoire	: How many and which operations to provide, and how complex operations should be.
Data Types	: The various type of data upon which operations are performed.
Instruction format	: Instruction length (in bits), number of addresses, size of various fields and so on.
Registers	: Number of CPU registers that can be referenced by instructions and their use.
Addressing	: The mode or modes by which the address of an operand is specified.

Types of Operands

Machine instructions operate on data. Data can be categorised as follows :

Addresses: It basically indicates the address of a memory location. Addresses are nothing but the unsigned integer, but treated in a special way to indicate the address of a memory location. Address arithmatic is somewhat different from normal arithmatic and it is related to machine architecture.

Numbers: All machine languages include numeric data types. Numeric data are classified into two broad categories: integer or fixed point and floating point.

Characters: A common form of data is text or character strings. Since computer works with bits, so characters are represented by a sequence of bits. The most commonly used coding scheme is ASCII (American Standard Code for Information Interchange) code.

Logical Data: Normally each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data. It is sometime useful to consider an n-bit unit as consisting of n 1-bit items of data, each item having the value 0 or 1. When data are viewed this way, they are considered to be logical data. Generally 1 is treated as true and 0 is treated as false.

Types of Opearations

The number of different *opcodes* and their types varies widely from machine to machine. However, some general type of operations are found in most of the machine architecture. Those operations can be categorized as follows:

- Data Transfer
- Arithmatic
- Logical
- Conversion
- Input Output [I/O]
- System Control
- Transfer Control

Data Transfer:

The most fundamental type of machine instruction is the data transfer instruction. The data transfer instruction must specify several things. First, the location of the source and destination operands must be specified. Each location could be memory, a register, or the top of the stack. Second, the length of data to be transferred must be indicated. Third, as with all instructions with operands, the mode of addressing for each operand must be specified.

The CPU has to perform several task to accomplish a data transfer operation. If both source and destination are registers, then the CPU simply causes data to be transferred from one register to another; this is an operation internal to the CPU.

If one or both operands are in memory, then the CPU must perform some or all of the following actions:

a) Calculate the memory address, based on the addressing mode.
b) If the address refers to virtual memory, translate from virtual to actual memory address.
c) Determine whether the addressed item is in cache.
d) If not, issue a command to the memory module.

Commonly used data transfer operation:

	Operation Name		Description
	Move (Transfer)		Transfer word or block from source to destination
	Store		Transfer word from processor to memory
	Load (fetch)		Transfer word from memory to processor
	Exchange		Swap contents of source and destination
	Clear (reset)		Transfer word of 0s to destination
	Set		Transfer word of 1s to destination
	Push		Transfer word from source to top of stack
	Pop		Transfer word from top of stack to destination

Arithmatic:

Most machines provide the basic arithmatic operations like add, subtract, multiply, divide etc. These are invariably provided for signed integer (fixed-point) numbers. They are also available for floating point number.

The execution of an arithmatic operation may involve data transfer operation to provide the operands to the ALU input and to deliver the result of the ALU operation.

Commonly used data transfer operation:

	Operation Name		Description
	Add		Compute sum of two operands
	Subtract		Compute difference of two operands
	Multiply		Compute product of two operands
	Divide		Compute quotient of two operands
	Absolute		Replace operand by its absolute value
	Negate		Change sign of operand
	Increment		Add 1 to operand
	Decrement		Subtract 1 from operand

Logical:

Most machines also provide a variety of operations for manipulating individual bits of a word or other addressable units.

Most commonly available logical operations are:

	Operation Name		Description
	AND		Performs the logical operation AND bitwise
	OR		Performs the logical operation OR bitwise
	NOT		Performs the logical operation NOT bitwise
	Exclusive OR		Performs the specified logical operation Exculsive-OR bitwise
	Test		Test specified condition; set flag(s) based on outcome
	Compare		Make logical or arithmatic comparison Set flag(s) based on outcome
	Set Control Variables		Class of instructions to set controls for protection purposes, interrupt handling, timer control etc.
	Shift		Left (right) shift operand, introducing constant at end
	Rotate		Left (right) shift operation, with wraparound end

Conversion:

Conversion instructions are those that change the format or operate on the format of data. An example is converting from decimal to binary.

Input/Output :

Input/Output instructions are used to transfer data between input/output devices and memory/CPU register.

Commonly available I/O operations are:

	Operation Name		Description
	Input (Read)		Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
	Output (Write)		Transfer data from specified source to I/O port or device.
	Start I/O		Transfer instructions to I/O processor to initiate I/O operation.
	Test I/O		Transfer status information from I/O system to specified destination

System Control:

System control instructions are those which are used for system setting and it can be used only in privileged state. Typically, these instructions are reserved for the use of operating systems. For example, a system control instruction may read or alter the content of a control register. Another instruction may be to read or modify a storage protection key.

Transfer of Control:

In most of the cases, the next instruction to be performed is the one that immediately follows the current instruction in memory. Therefore, program counter helps us to get the next instruction. But sometimes it is required to change the sequence of instruction execution and for that instruction set should provide instructions to accomplish these tasks. For these instructions, the operation performed by the CPU is to upload the program counter to contain the address of some instruction in memory. The most common transfer-of-control operations found in instruction set are: branch, skip and procedure call.

Branch Instruction

A branch instruction, also called a jump instruction, has one of its operands as the address of the next instruction to be executed. Basically there are two types of branch instructions: Conditional Branch instruction and unconditionla branch instruction. In case of unconditional branch instruction, the branch is made by updating the program counter to address specified in operand. In case of conditional branch instruction, the branch is made only if a certain condition is met. Otherwise, the next instruction in sequence is executed.

There are two common ways of generating the condition to be tested in a conditional branch instruction

First most machines provide a 1-bit or multiple-bit condition code that is set as the result of some operations. As an example, an arithmetic operation could set a 2-bit condition code with one of the following four values: zero, positive, negative and overflow. On such a machine, there could be four different conditional branch instructions:

BRP X Branch to location X if result is positive
BRN X Branch to location X if result is negative
BRZ X Branch to location X is result is zero
BRO X Branch to location X if overflow occurs

In all of these cases, the result referred to is the result of the most recent operation that set the condition code.

Another approach that can be used with three address instruction format is to perform a comparison and specify a branch in the same instruction.

For example,

BRE R1, R2, X Branch to X if contents of R1 = Contents of R2.

Skip Instruction

Another common form of transfer-of-control instruction is the skip instruction. Generally, the skip imples that one instruction to be skipped; thus the implied address equals the address of the next instruction plus one instruction length. A typical example is the increment-and-skip-if-zero (ISZ) instruction. For example,

```
ISZ  R1
```

This instruction will increment the value of the register R1. If the result of the increment is zero, then it will skip the next instruction.

Procedure Call Instruction

A procedure is a self contained computer program that is incorporated into a large program. At any point in the program the procedure may be invoked, or called. The processor is instructed to go and execute the entire procedure and then return to the point from which the call took place.

The procedure mechanism involves two basic instructions: a call instruction that branches from the present location to the procedure, and a return instruction that returns from the procedure to the place from which it was called. Both of these are forms of branching instructions.

Some important points regarding procedure call:

- A procedure can be called from more than one location.
- A procedure call can appear in a procedure. This allows the nesting of procedures to an arbitrary depth.
- Each procedure call is matched by a return in the called program.

Since we can call a procedure from a variety of points, the CPU must somehow save the return address so that the return can take place appropriately. There are three common places for storing the return address:

- Register
- Start of procedure
- Top of stack

Consider a machine language instruction CALL X, which stands for call procedure at location X. If the register apprach is used, CALL X causes the following actions:

RN ← PC + IL

PC ← X

where RN is a register that is always used for this purpose, PC is the program counter and IL is the instruction length. The called procedure can now save the contents of RN to be used for the later return.

A second possibilities is to store the return address at the start of the procedure. In this case, CALL X causes

X ← PC + IL

PC ← X + 1

Both of these approaches have been used. The only limitation of these approaches is that they prevent the use of reentrant procedures. A reentrant procedure is one in which it is possible to have several calls open to it at the same time.

A more general approach is to use stack. When the CPU executes a call, it places the return address on the stack. When it executes a return, it uses the address on the stack.

It may happen that, the called procedure might have to use the processor registers. This will overwrite the contents of the registers and the calling environment will lose the information. So, it is necessary to preserve the contents of processor register too along with the return address. The stack is used to store the contents of processor register. On return from the procedure call, the contents of the stack will be popped out to appropriate registers.

In addition to provide a return address, it is also often necessary to pass parameters with a procedure call. The most general approach to parameter passing is the stack. When the processor executes a call, it not only stacks the return address, it stacks parameters to be passed to the called procedures. The called procedure can access the parameters from the stack.Upon return, return parameters can also be placed on the stack. The entire set of parameters, including return address, that is stored for a procedure invocation is referred to as stack frame.

Most commonly used transfer of control operation (Next Page)

Most commonly used transfer of control operation:

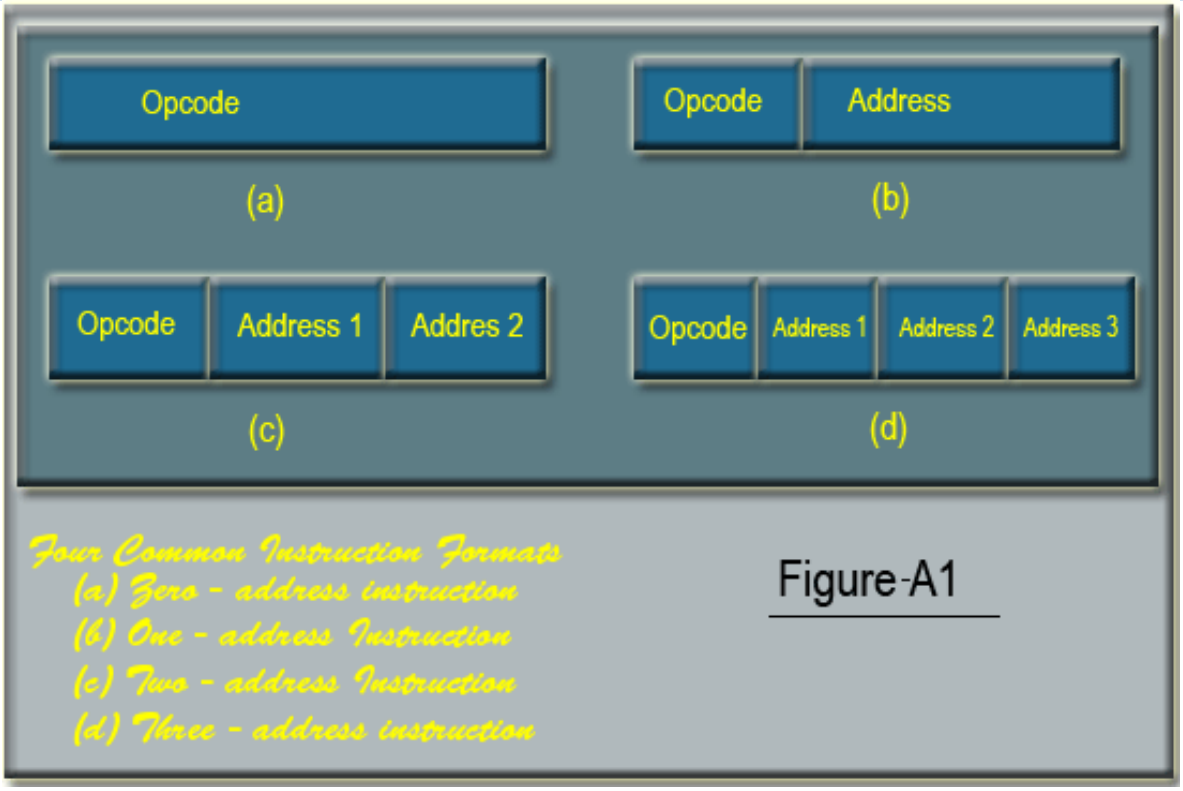
	Operation Name		Description
	Jump (branch)		Unconditional transfer, load PC with specific address
	Jump conditional		Test specific condition; either load PC with specific address or do nothing, based on condition
	Jump to subroutine		Place current program control information in known location; jump to specific address
	Return		Replace contents of PC and other register from known location
	Skip		Increment PC to skip next instruction
	Skip Conditional		Test specified condition; either skip or do nothing based on condition
	Halt		Stop program execution

Instruction Format:

An instruction format defines the layout of the bits of an instruction, in terms of its constituents parts. An instruction format must include an opcode and, implicitly or explicitly, zero or more operands. Each explit operand is referenced using one of the addressing mode that is available for that machine. The format must, implicitly or explictly, indicate the addressing mode of each operand. For most instruction sets, more than one instruction format is used. Four common instruction format are shown in the figure on the next slide .

Instruction Format Print this page

<< Previous | First | Last | Next >>



<< Previous | First | Last | Next >>

Instruction Length:

On some machines, all instructions have the same length; on others there may be many different lengths. Instructions may be shorter than, the same length as, or more than the word length. Having all the instructions be the same length is simpler and make decoding easier but often wastes space, since all instructions then have to be as long as the longest one. Possible relationship between instruction length and word length is shown in the figure.

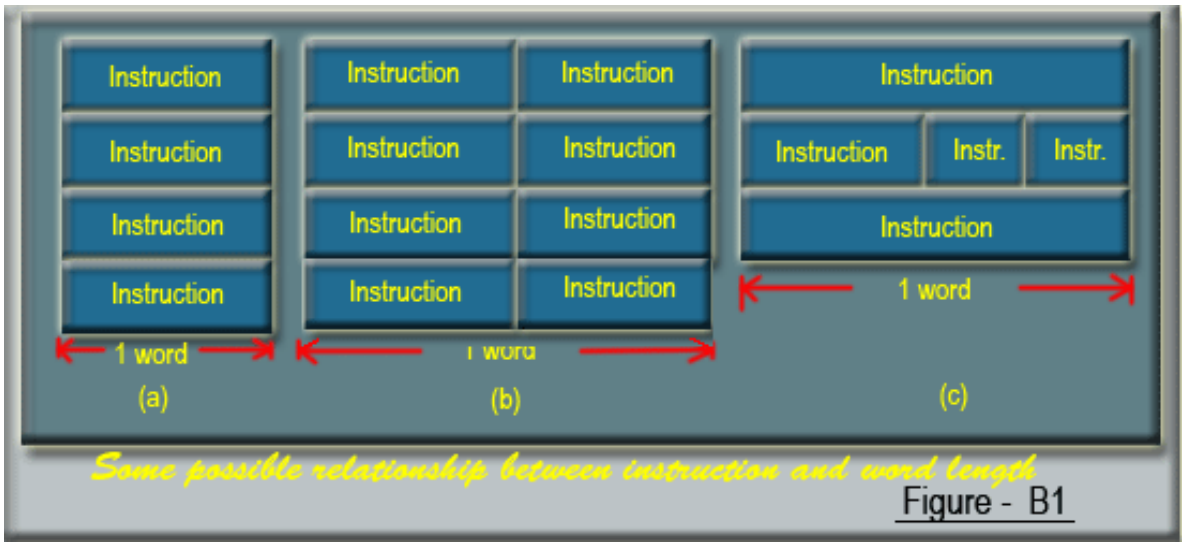


Figure - B1

Generally there is a correlation between memory transfer length and the instruction length. Either the instruction length should be equal to the memory transfer length or one should be a multiple of the other. Also in most of the case there is a correlation between memory transfer length and word length of the machine.

Allocation of Bits:

For a given instruction length, there is a clearly a trade-off between the number of opcodes and the power of the addressing capabilities. More opcodes obviously mean more bits in the opcode field. For an instruction format of a given length, this reduces the number of bits available for addressing.

The following interrelated factors go into determining the use of the addressing bits:

Number of Addressing modes:

Sometimes as addressing mode can be indicated implicitly. In other cases, the addressing mode must be explicit, and one or more bits will be needed.

Number of Operands:

Typical instructions on today's machines provide for two operands. Each operand address in the instruction might require its own mode indicator, or the use of a mode indicator could be limited to just one of the address field.

Register versus memory:

A machine must have registers so that data can be brought into the CPU for processing. With a single user-visible register (usually called the accumulator), one operand address is implicit and consumes no instruction bits. Even with multiple registers, only a few bits are needed to specify the register. The more that registers can be used for operand references, the fewer bits are needed.

Number of register sets:

A number of machines have one set of general purpose registers, with typically 8 or 16 registers in the set. These registers can be used to store data and can be used to store addresses for displacement addressing. The trend recently has been away from one bank of general purpose registers and toward a collection of two or more specialized sets (such as data and displacement).

Address range:

For addresses that reference memory, the range of addresses that can be referenced is related to the number of address bits. With displacement addressing, the range is opened up to the length of the address register.

Address granularity:

In a system with 16- or 32-bit words, an address can reference a word or a byte at the designer's choice. Byte addressing is convenient for character manipulation but requires, for a fixed size memory, more address bits.

Variable-Length Instructions:

Instead of looking for fixed length instruction format, designer may choose to provide a variety of instructions formats of different lengths. This tectic makes it easy to provide a large repertoire of opcodes, with different opcode lengths. Addressing can be more flexible, with various combinations of register and memory references plus addressing modes. With variable length instructions, many variations can be provided efficiently and compactly. The principal price to pay for variable length instructions is an increase in the complexity of the CPU.

Number of addresses :

The processor architecture is described in terms of the number of addresses contained in each instruction. Most of the arithmatic and logic instructions will require more operands. All arithmatic and logic operations are either unary (one source operand, e.g. NOT) or binary (two source operands, e.g. ADD).

Thus, we need a maximum of two addresses to reference source operands. The result of an operation must be stored, suggesting a third reference.

Three address instruction formats are not common because they require a relatively long instruction format to hold the three address reference.

With two address instructions, and for binary operations, one address must do double duty as both an operand and a result.

In one address instruction format, a second address must be implicit for a binary operation. For implicit reference, a processor register is used and it is termed as accumulator(AC). the accumulator contains one of the operands and is used to store the result.

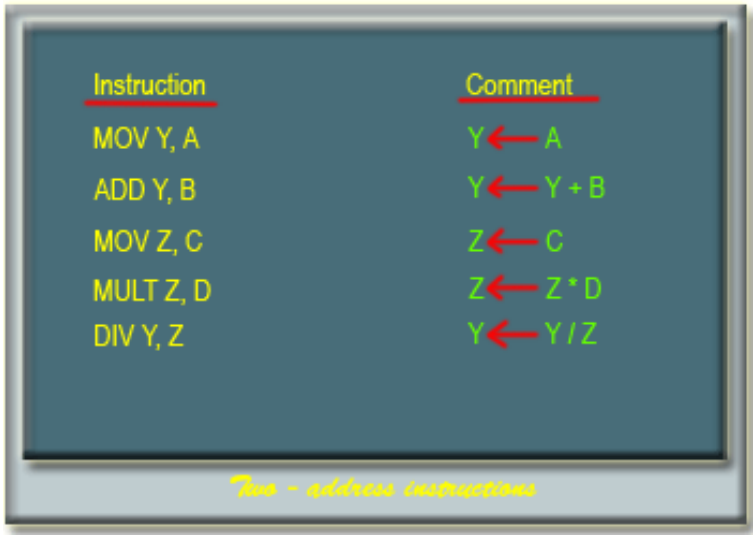
Consider a simple arithmatic expression to evalute:

$$Y= (A + B) / (C * D)$$

Instruction Addressing

Print this page

<< Previous | First | Last | Next >>



<< Previous | First | Last | Next >>

Instruction Format

Print this page

<< Previous | First | Last | Next >>

Print this page

<< Previous | First | Last | Next >>

Computer Organization and Architecture Print this page

Module 5 : CPU Design

In this Module, we have six lectures, viz.

1. **Introduction to CPU Design**
2. **Processor Organization**
3. **Execution of complete Instruction**
4. **Design of control unit**
5. **Microprogrammed control - I**
6. **Microprogrammed control - II**

Click the proper link on the left side for the lectures

Introduction to CPU

The operation or task that must perform by CPU are:

- **Fetch Instruction:** The CPU reads an instruction from memory.
- **Interprete Instruction:** The instruction is decoded to determine what action is required.
- **Fetch Data:** The execution of an instruction may require reading data from memory or I/O module.
- **Process data:** The execution of an instruction may require performing some arithmatic or logical operation on data.
- **Write data:** The result of an execution may require writing data to memory or an I/O module.

To do these tasks, it should be clear that the CPU needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is beign executed. In other words, the CPU needs a small internal memory. These storage location are generally referred as registers.

The major components of the CPU are an arithmatic and logic unit (ALU) and a control unit (CU). The ALU does the actual computation or processing of data. The CU controls the movement of data and instruction into and out of the CPU and controls the operation of the ALU.

Introduction to CPU

The CPU is connected to the rest of the system through system bus. Through system bus, data or information gets transferred between the CPU and the other component of the system. The system bus may have three components:

Data Bus:
Data bus is used to transfer the data between main memory and CPU.

Address Bus:
Address bus is used to access a particular memory location by putting the address of the memory location.

Control Bus:
Control bus is used to provide the different control signal generated by CPU to different part of the system. As for example, memory read is a signal generated by CPU to indicate that a memory read operation has to be performed. Through control bus this signal is transferred to memory module to indicate the required operation.

There are three basic components of CPU: register bank, ALU and Control Unit. There are several data movements between these units and for that an internal CPU bus is used. Internal CPU bus is needed to transfer data between the various registers and the ALU, because the ALU in fact operates only on data in the internal CPU memory.

The internal organization of CPU in more abstract level is shown in the figure A & B ....

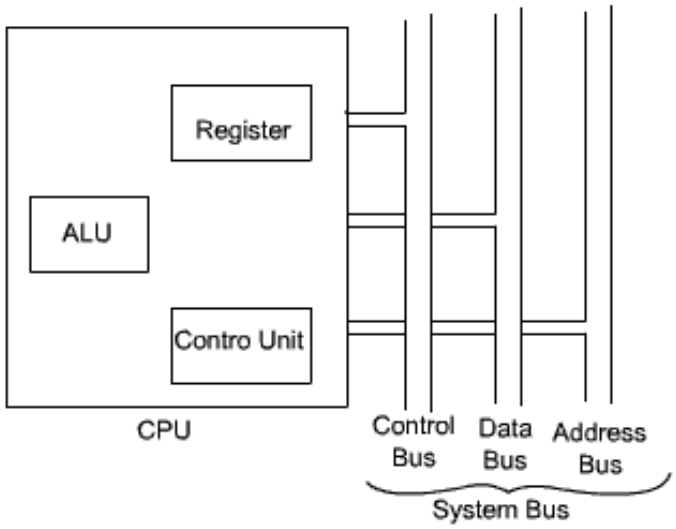


Figure A - 1 : CPU with the system Bus

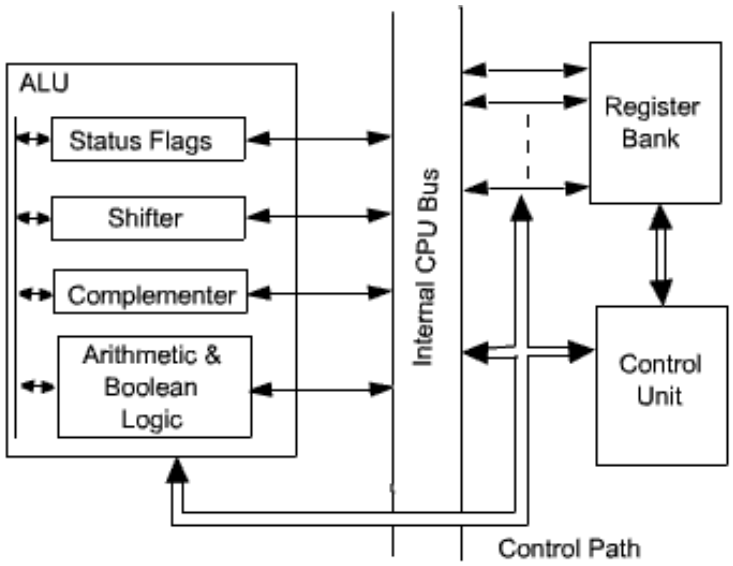


Figure A - 2 : Internal Structure of the CPU

Register Organization

A computer system employs a memory hierarchy. At the highest level of hierarchy, memory is faster, smaller and more expensive. Within the CPU, there is a set of registers which can be treated as a memory in the highest level of hierarchy. The registers in the CPU can be categorized into two groups:

- **User-visible registers:** These enables the machine - or assembly-language programmer to minimize main memory reference by optimizing use of registers.
- **Control and status registers:** These are used by the control unit to control the operation of the CPU. Operating system programs may also use these in privileged mode to control the execution of program.

User-visible Registers:

The user-visible registars can be categorized as follows:

- General Purpose Registers
- Data Registers
- Address Registers
- Condition Codes

General-purpose registers can be assigned to a variety of functions by the programmer. In some cases, general-purpose registers can be used for addressing functions (e.g., register indirect, displacement).

In other cases, there is a partial or clean separation between data registers and address registers.

Data registers may be used to hold only data and cannot be employed in the calculation of an operand address.

Address registers may be somewhat general purpose, or they may be devoted to a particular addressing mode. Examples include the following:

- **Segment pointer:** In a machine with segment addressing, a segment register holds the address of the base of the segment. There may be multiple registers, one for the code segment and one for the data segment.
- **Index registers:** These are used for indexed addressing and may be autoindexed.
- **Stack pointer:** If there is user visible stack addressing, then typically the stack is in memory and there is a dedicated register that points to the top of the stack.

Condition Codes (also referred to as flags) are bits set by the CPU hardware as the result of the operations. For example, an arithmatic operation may produce a positive, negative, zero or overflow result. In addition to the result itself beign stored in a register or memory, a condition code is also set. The code may be subsequently be tested as part of a condition branch operation. Condition code bits are collected into one or more registers.

Register Organization

There are a variety of CPU registers that are employed to control the operation of the CPU. Most of these, on most machines, are not visible to the user.

Different machines will have different register organizations and use different terminology. We will discuss here the most commonly used registers which are part of most of the machines.

Four registers are essential to instruction execution:

Program Counter (PC): Contains the address of an instruction to be fetched. Typically, the PC is updated by the CPU after each instruction fetched so that it always points to the next instruction to be executed. A branch or skip instruction will also modify the contents of the PC.

Instruction Register (IR): Contains the instruction most recently fetched. The fetched instruction is loaded into an IR, where the opcode and operand specifiers are analyzed.

Memory Address Register (MAR): Containts the address of a location of main memory from where information has to be fetched or information has to be stored. Contents of MAR is directly connected to the address bus.

Memory Buffer Register (MBR): Contains a word of data to be written to memory or the word most recently read. Contents of MBR is directly connected to the data bus.It is also known as Memory Data Register(MDR).

Apart from these specific register, we may have some temporary registers which are not visible to the user. As such, there may be temporary buffering registers at the boundary to the ALU; these registers serve as input and output registers for the ALU and exchange data with the MBR and user visible registers.

Processor Status Word

All CPU designs include a register or set of registers, often known as the processor status word (PSW), that contains status information. The PSW typically contains condition codes plus other status information. Common fields or flags include the following:

- **Sign:** Contains the sign bit of the result of the last arithmatic operation.
- **Zero:** Set when the result is zero.
- **Carry:** Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high order bit.
- **Equal:** Set if a logical campare result is equal.
- **Overflow:** Used to indicate arithmatic overflow.
- **Interrupt enable/disable:** Used to enable or disable interrupts.
- **Supervisor:** Indicate whether the CPU is executing in supervisor or user mode.
  Certain privileged instructions can be executed only in supervisor mode, and certain areas of memory can be accessed only in supervisor mode.

Apart from these, a number of other registers related to status and control might be found in a particular CPU design. In addition to the PSW, there may be a pointer to a block of memory containing additional status information (e.g. process control blocks).

Concept of Program Execution

The instructions constituting a program to be executed by a computer are loaded in sequential locations in its main memory. To execute this program, the CPU fetches one instruction at a time and performs the functions specified. Instructions are fetched from successive memory locations until the execution of a branch or a jump instruction.

The CPU keeps track of the address of the memory location where the next instruction is located through the use of a dedicated CPU register, referred to as the program counter (PC). After fetching an instruction, the contents of the PC are updated to point at the next instruction in sequence.

For simplicity, let us assume that each instruction occupies one memory word. Therefore, execution of one instruction requires the following three steps to be performed by the CPU:

1. Fetch the contents of the memory location pointed at by the PC. The contents of this location are interpreted as an instruction to be executed. Hence, they are stored in the instruction register (IR). Symbolically this can be written as:

   IR = [[PC]]

2. Increment the contents of the PC by 1.

   PC = [PC] + 1

3. Carry out the actions specified by the instruction stored in the IR.

The first two steps are usually referred to as the fetch phase and the step 3 is known as the execution phase. Fetch cycle basically involves read the next instruction from the memory into the CPU and along with that update the contents of the program counter. In the execution phase, it interpretes the opcode and perform the indicated operation. The instruction fetch and execution phase together known as instruction cycle. The basic instruction cycle is shown in the figure.

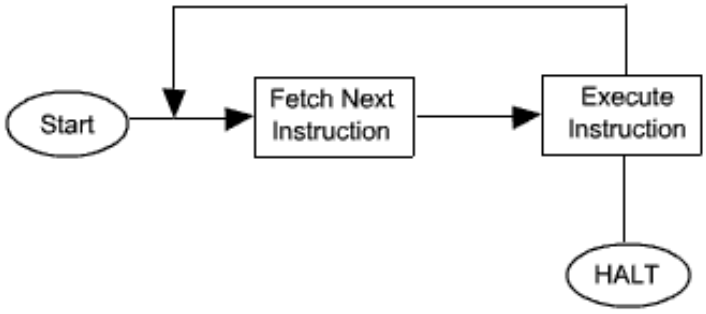


Figure B : Basic Instruction cycle

In cases, where an instruction occupies more than one word, step 1 and step 2 can be repeated as many times as necessary to fetch the complete instruction. In these cases, the execution of a instruction may involve one or more operands in memory, each of which requires a memory access. Further, if indirect addressing is used, then additional memory access are required.

The fetched instruction is loaded into the instruction register. The instruction contains bits that specify the action to be performed by the processor. The processor interpretes the instruction and performs the required action. In general, the actions fall into four categories:

- **Processor-memory:** Data may be transfrred from processor to memory or from memory to processor.
- **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- **Data processing:** The processor may perform some arithmatic or logic operation on data.
- **Control:** An instruction may specify that the sequence of execution be altered.

The main line of activity consists of alternating instruction fetch and instruction execution activities. After an instruction is fetched, it is examined to determine if any indirect addressing is involved. If so, the required operands are fetched using indirect addressing.

The execution cycle of a perticular instruction may involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation. With these additional considerations the basic instruction cycle can be expanded with more details view in this figure. The figure is in the form of a state diagram.

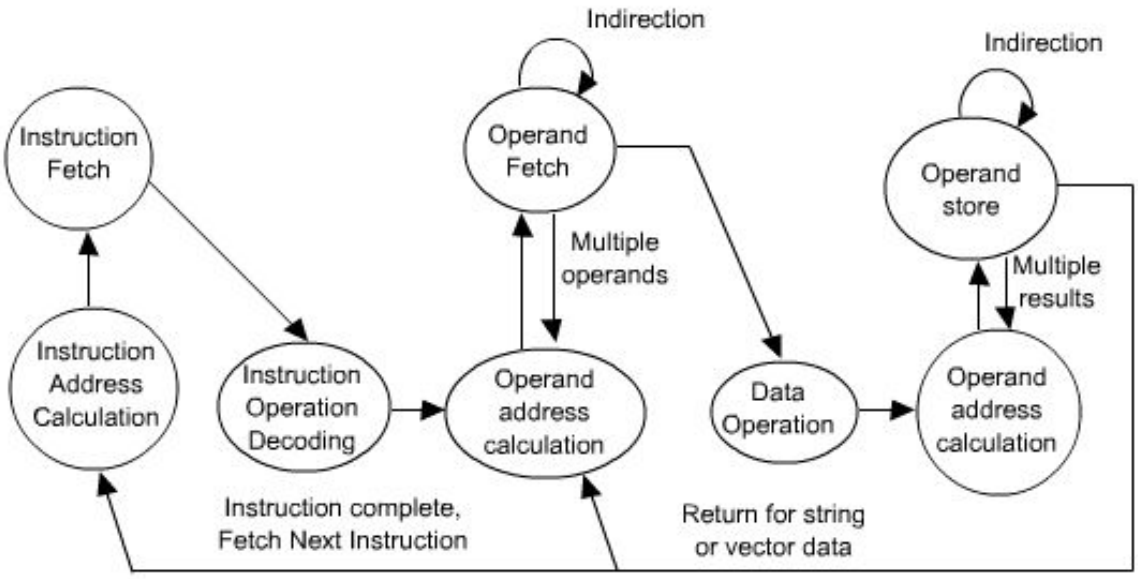


Figure C : Instruction cycle state diagram.

For any given instruction cycle, some states may be null and other may be visited more than once. The states are described below:

- **Instruction Address Calculation (IAC):** Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number of the address of the previous instruction.
- **Instruction Fetch (IF):** Read instruction from the memory location into the processor.
- **Instruction Operation Decoding (IOD):** Analyze instruction to determine type of operation to be performed and operand(s) to be used.
- **Operand Address Calculation (OAC):** If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand.
- **Operand Fetch (OF):** Fetch the operand from memory or read it from I/O.
- **Data Operation (DO):** Perform the operation indicated in the instruction.
- **Operand Store (OS):** Write the result into memory or out to I/O.

In the figure, it allows for multiple operands and multiple results, because some instructions on some machines require this. Also, in some machines, a single instruction can specify an operation to be performed on a vector (one-dimensional array) of numbers or a string of characters. This would involve repetitive operand fetch and/or store operation for a particular opcode (opcode is fetched only once).

Interrupts

Virtually all computers provide a mechanism by which other module (I/O, memory etc.) may interrupt the normal processing of the processor. The most common classes of interrupts are:

Program:	Generated by some condition that occurs as a result of an instruction execution, such as arithmatic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside the user's allowed memory space.
Timer:	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O:	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure:	Generated by a failure such as power failure or memory parity error.

The issue of interrupts is discussed in later module, but we need to introduce the concept of interrupt now to understand more clearly the nature of instruction cycle.

Interupts are provided primarily as a way to improve processing effeciency. For example, most external devices are much slower than the processor. With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress.

For I/O operation, say an output operation, like printing some information by a printer. Printer is much slower device than the CPU. The CPU puts some information on the output buffer. While printer is busy printing these information from output buffer, CPU is lying idle. During this time CPU can perform some other task which does not involve the memory bus.

When the external device becomes ready to be serviced, that is, when it is ready to accept more data from the processor, the I/O module for that external device sends an interrupt request signal to the processor. The processor responds by suspending operation of the current program, branching off to a program to service the particular I/O device (known as an interrupt handler), and resuming the original execution after the device is serviced.

From the point of view of the user program, an interrupt is just that : ***an interruption of the normal sequence of execution. When the interrupt processing is completed, execution resumes***.

To accommodate interrupts, an interrupt cycle is added to the instruction cycle, which is shown in the **figure D**. In the interrupt cycle, the processor checks if any interrupt have occurred, indicated by the presence of an interrupt signals. If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current program. If an interrupt is pending, the processor does the following:

1. It suspends the execution of the current program being executed and saves its contents. This means saving the address of the next instruction to be executed (current contents of the program counter) and any other data relevant to the processor's current activity.
2. It sets the program counter to the starting address of an interrupt handler routine.

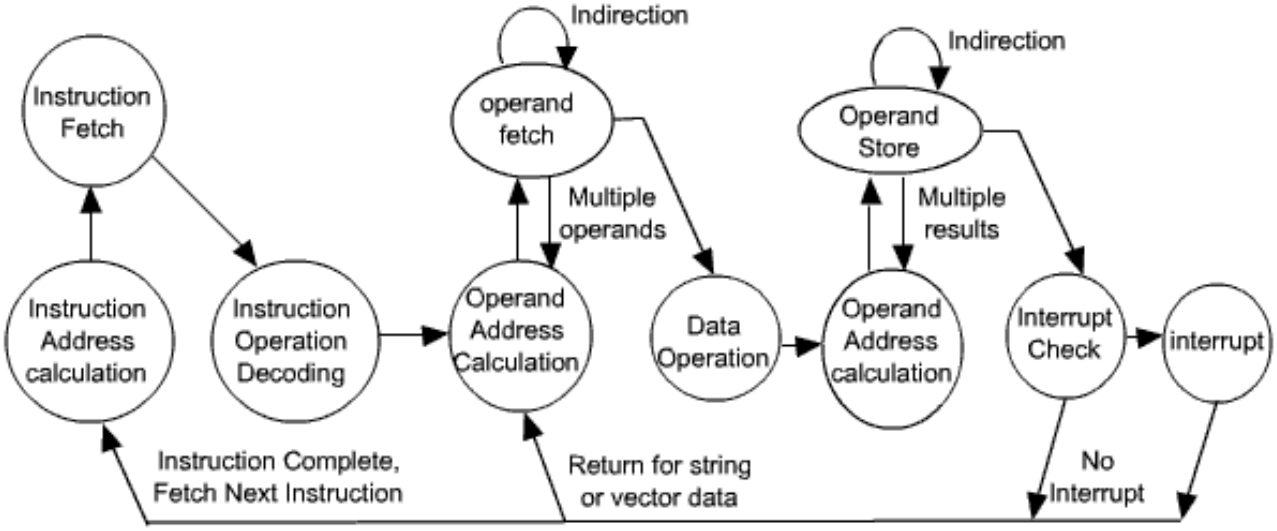


Figure D : Instruction cycle state diagram with interrupt.

The figure shows a revised instruction cycle state diagram that includes interrupt cycle processing.

Processor Organization

There are several components inside a CPU, namely, ALU, control unit, general purpose register, Instruction registers etc. Now we will see how these components are organized inside CPU. There are several ways to place these components and inteconnect them. One such organization is shown in the figure A.

In this case, the arithmatic and logic unit (ALU), and all CPU registers are connected via a single common bus. This bus is internal to CPU and this internal bus is used to transfer the information between different components of the CPU. This organization is termed as single bus organization, since only one internal bus is used for transferring of information between different components of CPU. We have external bus or buses to CPU also to connect the CPU with the memory module and I/O devices. The external memory bus is also shown in the **figure A** connected to the CPU via the memory data and address register **MDR** and **MAR**.

The number and function of registers R0 to R(n-1) vary considerably from one machine to another. They may be given for general-purpose for the use of the programmer. Alternatively, some of them may be dedicated as special-purpose registers, such as **index register** or **stack pointers**.

In this organization, two registers, namely Y and Z are used which are transperant to the user. Programmer can not directly access these two registers. These are used as input and output buffer to the ALU which will be used in ALU operations. They will be used by CPU as temporary storage for some instructions.

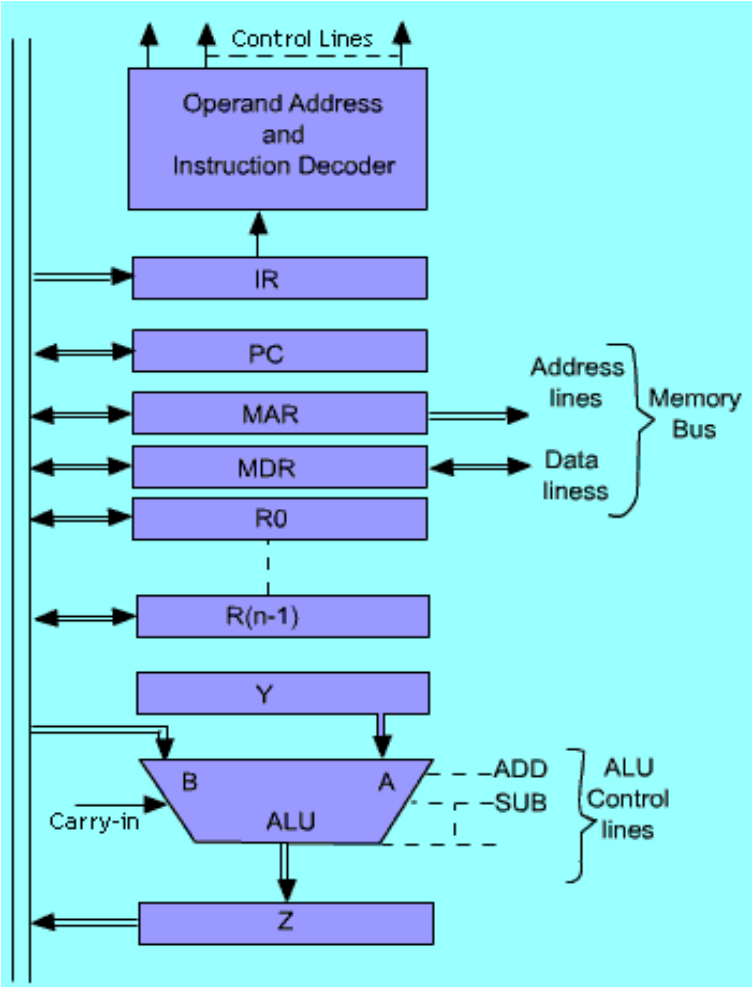


Figure A : Single bus organization of the data path inside the CPU

Processor Organization Print this page

<< Previous | First | Last | Next >>

For the execution of an instruction, we need to perform an instruction cycle. An instruction cycle consists of two phase,

- Fetch cycle and
- Execution cycle.

Most of the operation of a CPU can be carried out by performing one or more of the following functions in some prespecified sequence:

1. Fetch the contents of a given memory location and load them into a CPU register.
2. Store a word of data from a CPU register into a given memory location.
3. Transfer a word of data from one CPU register to another or to the ALU.
4. Perform an arithmatic or logic operation, and store the result in a CPU register.

Now we will examine the way in which each of the above functions is implemented in a computer. Fetching a Word from Memory:

Information is stored in memory location indentified by their address. To fetch a word from memory, the CPU has to specify the address of the memory location where this information is stored and request a Read operation. The information may include both, the data for an operation or the instruction of a program which is available in main memory.

>> Previous | First | Last | Next >>

To perform a memory fetch operation, we need to complete the following tasks:

The CPU transfers the address of the required memory location to the Memory Address Register (MAR).

The MAR is connected to the memory address line of the memory bus, hence the address of the required word is transfered to the main memory.

Next, CPU uses the control lines of the memory bus to indicate that a Read operation is initiated. After issuing this request, the CPU waits until it receives an answer from the memory, indicating that the requested operation has been completed.

This is accomplished by another control signal of memory bus known as Memory-Function-Complete (MFC).

The memory set this signal to 1 to indicate that the contents of the specified memory location are available in memory data bus.

As soon as MFC signal is set to 1, the information available in the data bus is loaded into the Memory Data Register (MDR) and this is available for use inside the CPU.

As an example, assume that the address of the memory location to be accessed is kept in register R2 and that the memory contents to be loaded into register R1. This is done by the following sequence of operations:

1. MAR ← [R2] 2. Read
3. Wait for MFC signal 4. R1 ← [MDR]

The time required for step 3 depends on the speed of the memory unit. In general, the time required to access a word from the memory is longer than the time required to perform any operation within the CPU.

The scheme that is used here to transfer data from one device (memory) to another device (CPU) is referred to as an asynchronous transfer.

This asynchronous transfer enables transfer of data between two independent devices that have different speeds of operation. The data transfer is synchronised with the help of some control signals. In this example, Read request and MFC signal are doing the synchronization task.

An alternative scheme is synchronous transfer. In this case all the devices are controlled by a common clock pulse (continously running clock of a fixed frequency). These pulses provide common timing signal to the CPU and the main memory. A memory operation is completed during every clock period. Though the synchronous data transfer scheme leads to a simpler implementation, it is difficult to accommodate devices with widely varying speed. In such cases, the duration of the clock pulse will be synchronized to the slowest device. It reduces the speed of all the devices to the slowest one.

Storing a word into memory

The procedure of writing a word into memory location is similar to that for reading one from memory. The only difference is that the data word to be written is first loaded into the MDR, the write command is issued.

As an example, assumes that the data word to be stored in the memory is in register R1 and that the memory address is in register R2. The memory write operation requires the following sequence:

1. MAR ← [R2]
2. MDR → [R1]
3. Write
4. Wait for MFC

- In this case step 1 and step 2 are independent and so they can be carried out in any order. In fact, step 1 and 2 can be carried out simultaneously, if this is allowed by the architecture, that is, if these two data transfers (memory address and data) do not use the same data path.

In case of both memory read and memory write operation, the total time duration depends on wait for the MFC signal, which depends on the speed of the memory module.

There is a scope to improve the performance of the CPU, if CPU is allowed to perform some other operation while waiting for MFC signal. During the period, CPU can perform some other instructions which do not require the use of MAR and MDR.

Register Transfer Operation

Register transfer operations enable data transfer between various blocks connected to the common bus of CPU. We have several registers inside CPU and it is needed to transfer information from one register another. As for example during memory write operation data from appropriate register must be moved to MDR.

Since the input output lines of all the register are connected to the common internal bus, we need appropriate input output gating. The input and output gates for register R_i are controlled by the signal $R_{i\ in}$ and $R_{i\ out}$ respectively.

Thus, when $R_{i\ in}$ set to 1 the data available in the common bus is loaded into R_i . Similarly when, $R_{i\ out}$ is set to 1, the contents of the register R_i are placed on the bus. To transfer data from one register to other register, we need to generate the appropriate register gating signal.

For example, to transfer the contents of register R_1 to register R_2, the following actions are needed:

- Enable the output gate of register R_1 by setting R_{1out} to 1.
  -- This places the contents of R_1 on the CPU bus.
- Enable the input gate of register R_2 by setting $R_{2\ in}$ to 1.
  -- This loads data from the CPU bus into the register R_2.

Performing the arithmetic or logic operation:

- Generally ALU is used inside CPU to perform arithmetic and logic operation. ALU is a combinational logic circuit which does not have any internal storage.

Therefore, to perform any arithmetic or logic operation (say binary operation) both the input should be made available at the two inputs of the ALU simultaneously. Once both the inputs are available then appropriate signal is generated to perform the required operation.

We may have to use temporary storage (register) to carry out the operation in ALU .

The sequence of operations that have to carried out to perform one ALU operation depends on the organization of the CPU. Consider an organization in which one of the operand of ALU is stored in some temporary register Y and other operand is directly taken from CPU internal bus. The result of the ALU operation is stored in another temporary register Z.

The organization is shown in the figure (Next Page..)

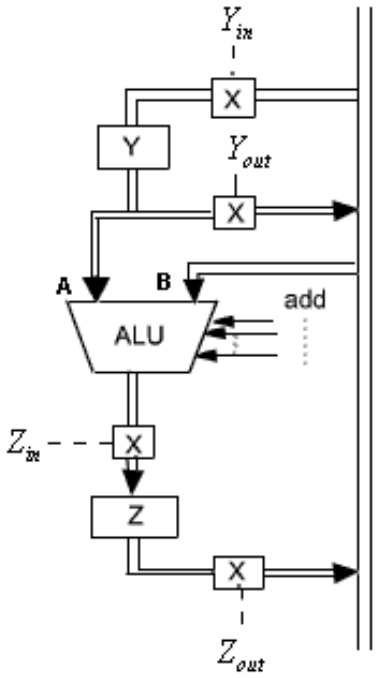


Figure B : Organization for Arithmatic & Logic Operation.

Therefore, the sequence of operations to add the contents of register R_1 to register R_2 and store the result in register R3 should be as follows:

1. R_{1out}, Y_{in}
2. R_{2out}, Add, Z_{in}
3. Z_{out}, $R3_{in}$

In step 2 of this sequence, the contents of register R_2 are gated to the bus, hence to input –B of the ALU which is directly connected to the bus. The contents of register Y are always available at input A of ALU. The function performed by the ALU depends on the signal applied to the ALU control lines. In this example, the Add control line of ALU is set to 1, which indicate the addition operation and the output of ALU is the sum of the two numbers at input A and B. The sum is loaded into register Z, since the input gate is enabled (Z_{in}). In step 3, the contents of register Z are transferred to the destination register R_3.

Question: Whether step 2 and step 3 can be carried out simultaneously.

Multiple Bus Organization

Till now we have considered only one internal bus of CPU. The single-bus organization, which is only one of the possibilities for interconnecting different building blocks of CPU.

An alternative structure is the two bus structure, where two different internal buses are used in CPU. All register outputs are connected to bus A, add all registered inputs are connected to bus B.

There is a special arrangement to transfer the data from one bus to the other bus. The buses are connected through the bus tie G. When this tie is enabled data on bus A is transfer to bus B. When G is disabled, the two buses are electrically isolated.

Since two buses are used here the temporary register Z is not required here which is used in single bus organization to store the result of ALU. Now result can be directly transferred to bus B, since one of the inputs is in bus A. With the bus tie disabled, the result can directly be transferred to destination register.

For example, for the operation, [R3] $\leftarrow$ [R1] + [R2] can now be performed as

1. $R1_{out}$, G_{enable}, Y_{in}
2. $R2_{out}$, Add, ALU_{out}, $R3_{in}$

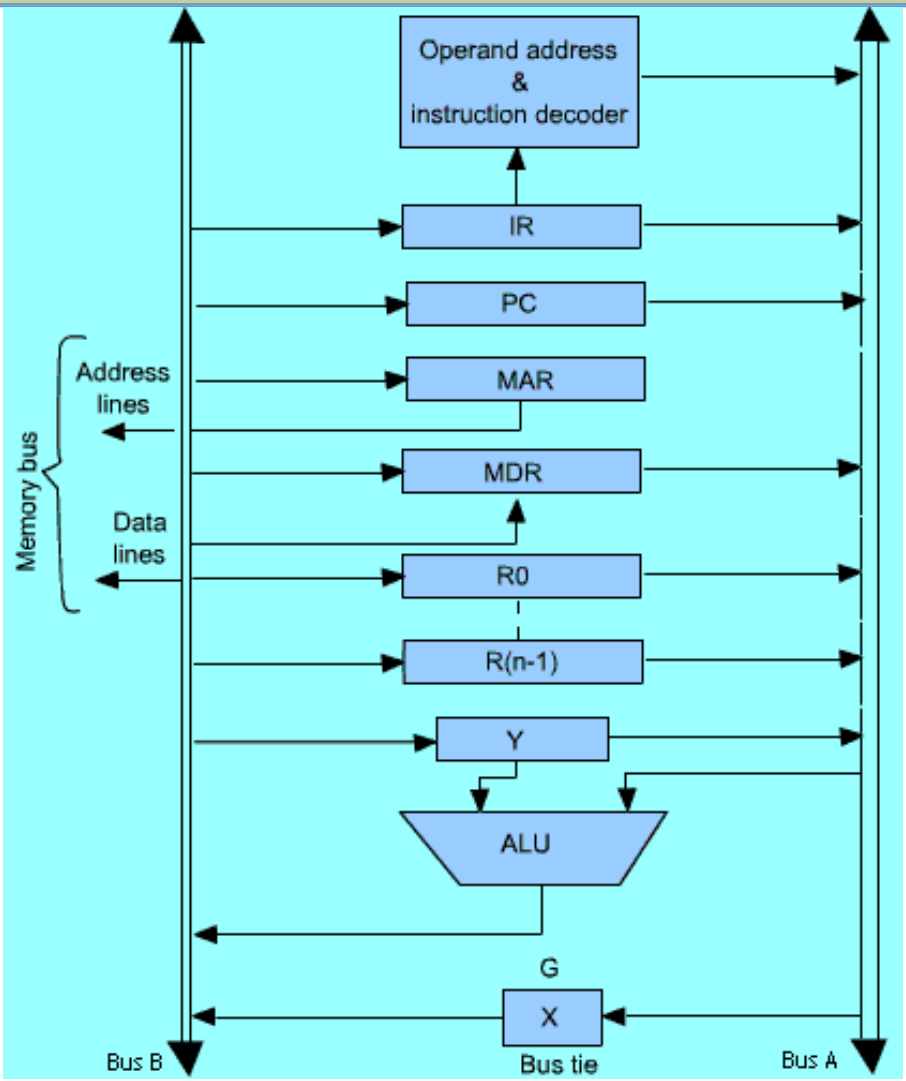


Figure C : Two bus structure

In this case source register R2 and destination register R3 has to be different, because the two operations $R2_{in}$ and $R2_{out}$ can not be performed together. If the registers are made of simple latches then only we have the restriction.

We may have another CPU organization, where three internal CPU buses are used. In this organization each bus connected to only one output and number of inputs. The elimination of the need for connecting more than one output to the same bus leads to faster bus transfer and simple control.

A simple three-bus organization is shown in the figure D (next page..).

A multiplexer is provided at the input to each of the two work registers A and B, which allow them to be loaded from either the input data bus or the register data bus.

In the diagram, a possible interconnection of three-bus organization is presented, there may be different interconnections possible.

In this three bus organization, we are keeping two input data buses instead of one that is used in two bus organization.

Two separate input data buses are present – one is for external data transfer, i.e. retrieving from memory and the second one is for internal data transfer that is transferring data from general purpose register to other building block inside the CPU.

Like two bus organization, we can use bus tie to connect the input bus and output bus. When the bus tie is enable, the information that is present in input bus is directly transferred to output bus. We may use one bus tie G1 between input data bus and ALU output bus and another bus tie G2 between register data bus and ALU output data bus.

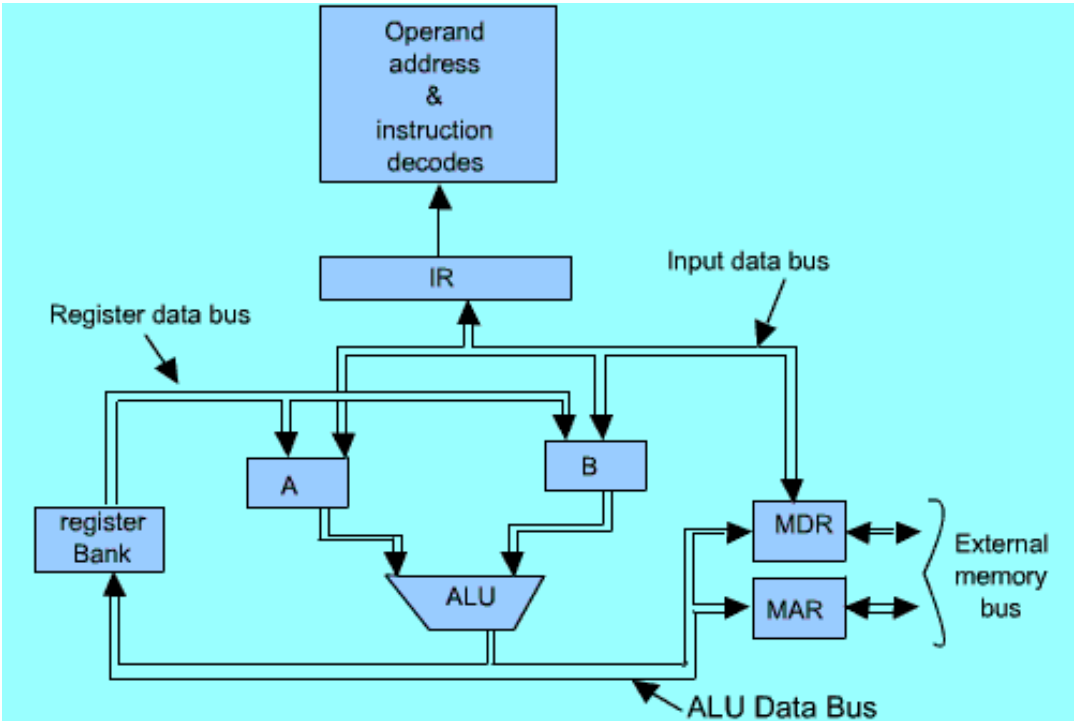


Figure D : Three Bus structure

Execution of a Complete Instructions:

We have discussed about four different types of basic operations:

- Fetch information from memory to CPU
- Store information to CPU register to memory
- Transfer of data between CPU registers.
- Perform arithmetic or logic operation and store the result in CPU registers.

To execute a complete instruction we need to take help of these basic operations and we need to execute these operation in some particular order to execute an instruction.

As for example, consider the instruction : "Add contents of memory location NUM to the contents of register R1 and store the result in register R1." For simplicity, assume that the address NUM is given explicitly in the address field of the instruction .That is, in this instruction, direct addressing mode is used.

Execution of this instruction requires the following action :

1. Fetch instruction
2. Fetch first operand (Contents of memory location pointed at by the address field of the instruction)
3. Perform addition
4. Load the result into R1.

Execution of a Complete Instructions:

Print this page

<< Previous | First | Last | Next >>

Following sequence of control steps are required to implement the above operation for the single-bus architecture that we have discussed in earlier section.

Steps	Actions
1.	PC_{out}, MAR_{in}, Read, Clear Y, Set carry -in to ALU, Add, Z_{in}
2.	Z_{out}, PC_{in}, Wait For MFC
3.	MDR_{out}, Ir_{in}
4.	Address-field- of-IR_{out}, MAR_{in}, Read
5.	$R1_{out}$, Y_{in}, Wait for MFC
6.	MDR_{out}, Add, Z_{in}
7.	Z_{out}, $R1_{in}$
8.	END

Instructruction execution proceeds as follows:

In Step1:

The instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a read request to memory.

<< Previous | First | Last | Next >>

Execution of a Complete Instructions:

To perform this task first of all the contents of PC have to be brought to internal bus and then it is loaded to MAR.To perform this task control circuit has to generate the PC_{out} signal and MAR_{in} signal.

After issuing the read signal, CPU has to wait for some time to get the MFC signal. During that time PC is updated by 1 through the use of the ALU. This is accomplished by setting one of the inputs to the ALU (Register Y) to 0 and the other input is available in bus which is current value of PC.

At the same time, the carry-in to the ALU is set to 1 and an add operation is specified.

In Step 2:

The updated value is moved from register Z back into the PC. Step 2 is initiated immediately after issuing the memory Read request without waiting for completion of memory function. This is possible, because step 2 does not use the memory bus and its execution does not depend on the memory read operation.

In Step 3:

Step3 has been delayed until the MFC is received. Once MFC is received, the word fetched from the memory is transfered to IR (Instruction Register), Because it is an instruction. Step 1 through 3 constitute the instruction fetch phase of the control sequence.

The instruction fetch portion is same for all instructions. Next step onwards, instruction execution phase takes place.

Execution of a Complete Instructions:

As soon as the IR is loaded with instruction, the instruction decoding circuits interprets its contents. This enables the control circuitry to choose the appropriate signals for the remainder of the control sequence, step 4 to 8, which we referred to as the execution phase. To design the control sequence of execution phase, it is needed to have the knowledge of the internal structure and instruction format of the PU. Secondly , the length of instruction phase is different for different instruction.

In this example , we have assumed the following instruction format :

i.e., **opcode:** Operation Code
M: Memory address for source
R: Register address for source/destination

Execution of a Complete Instructions:

In Step 5 :

The destination field of IR, which contains the address of the register R1, is used to transfer the contents of register R1 to register Y and wait for Memory function Complete. When the read operation is completed, the memory operand is available in MDR.

In Step 6 :

The result of addition operation is performed in this step.

In Step 7:

The result of addition operation is transfered from temporary register **Z** to the destination register **R1** in this step.

In step 8 :

It indicates the end of the execution of the instruction by generating End signal. This indicates completion of execution of the current instruction and causes a new fetch cycle to be started by going back to step 1.

Execution of a Complete Instructions:

Print this page

<< Previous | First | Last | Next >>

Branching

With the help of branching instruction, the control of the execution of the program is transfered from one particular position to some other position, due to which the sequence flow of control is broken. Branching is accomplished by replacing the current contents of the PC by the branch address, that is, the address of the instruction to which branching is required.

Consider a branch instruction in which branch address is obtained by adding an offset **X**, which is given in the address field of the branch instruction, to the current value of PC.

Consider the following unconditional branch instruction

JUMP **X**

i.e., the format is

<< Previous | First | Last | Next >>

Execution of a Complete Instructions:

The control sequence that enables execution of an unconditional branch instruction using the single - bus organization is as follows :

Steps	Actions
1.	PCout, MAR_{in}, Read, Clear Y, Set Carry-in to ALU, Add ,Z_{in}
2.	Z_{out}, PC_{in}, Wait for MFC
3.	MDR_{out}, IR_{in}
4.	PC_{out}, Y_{in}
5.	Address field-of IR_{out}, Add, Z_{in}
6.	Z_{out}, PC_{in}
7.	End

Execution of a Complete Instructions:

Execution starts as usual with the fetch phase, ending with the instruction being loaded into the IR in step 3. To execute the branch instruction, the execution phase starts in step 4.

In Step 4

The contents of the PC are transferred to register Y.

In Step 5

The offset **X** of the instruction is gated to the bus and the addition operation is performed.

In Step 6

The result of the addition, which represents the branch address is loaded into the PC.

In Step 7

It generates the End signal to indicate the end of execution of the current instruction.

Execution of a Complete Instructions:

Print this page

<< Previous | First | Last | Next >>

Consider now the conditional branch instruction instead of unconditional branch. In this case, we need to check the status of the condition codes, between step 3 and 4. i.e., before adding the offset value to the PC contents.

For example, if the instruction decoding circuitry interprets the contents of the IR as a branch on Negative(BRN) instruction, the control unit proceeds as follows:First the condition code register is checked. If bit N (negative) is equal to 1 , the control unit proceeds with step 4 trough step 7 of control sequence of unconditional branch instruction.

If , on the other hand , N is equal to 0, and End signal is issued .

This in effect , terminates execution of the branch instruction and causes the instruction immediately following in the branch instruction to be fetched when a new fetch operation is performed.

Therefore , the control sequence for the conditional branch instruction BRN can be obtained from the control sequence of an unconditional branch instruction by replacing the step 4 by

4. If $\overline{N}$ then End
   If N then PC_{out}, y_{in}

<< Previous | First | Last | Next >>

Execution of a Complete Instructions:

Most commonly need conditional branch instructions are

BNZ : Branch on not Zero

BZ : Branch on positive

BP : Branch on Positive

BNP : Branch on not Positive

BO : Branch on overflow

Design of Control Unit

To execute an instruction, the control unit of the CPU must generate the required control signal in the proper sequence. As for example, during the fetch phase, CPU has to generate PC_{out} signal along with other required signal in the first clock pulse. In the second clock pulse CPU has to generate PC_{in} signal along with other required signals. So, during fetch phase, the proper sequence for generating the signal to retrieve from and store to PC is PC_{out} and PC_{in}.

To generate the control signal in proper sequence, a wide variety of techniques exist. Most of these techniques, howeve, fall into one of the two categories,

1. **Hardwired Control**
2. **Microprogrammed Control.**

Hardwired Control

In this hardwired control techniques, the control signals are generated by means of hardwired circuit. The main objective of control unit is to generate the control signal in proper sequence.

Consider the sequence of control signal required to execute the ***ADD*** instruction that is explained in previous lecture. It is obvious that eight non-overlapping time slots are required for proper execution of the instruction represented by this sequence.

Each time slot must be at least long enough for the function specified in the corresponding step to be completed. Since, the control unit is implemented by hardwire device and every device is having a propagation delay, due to which it requires some time to get the *stable output signal* at the output port after giving the input signal. So, to find out the time slot is a complicated design task.

For the moment, for simplicity, let us assume that all slots are equal in time duration. Therefore the required controller may be implemented based upon the use of a counter driven by a clock.

Each state, or count, of this counter corresponds to one of the steps of the control sequence of the instructions of the CPU.

In the previous lecture, we have mentioned control sequence for execution of two instructions only (one is for add and other one is for branch). Like that we need to design the control sequence of all the instructions.

Design of Control Unit

By looking into the design of the CPU, we may say that there are various instruction for add operation. As for example,

ADD	***NUM***	R_1	Add the contents of memory location specified by ***NUM*** to the contents of register R_1 . $R_1 \leftarrow R_1 + [NUM]$
ADD	R_2	R_1	Add the contents of register R_2 to the contents of register R_1. $R_1 \leftarrow R_1 + R_2$

The control sequence for execution of these two ***ADD*** instructions are different. Of course, the fetch phase of all the instructions remain same.

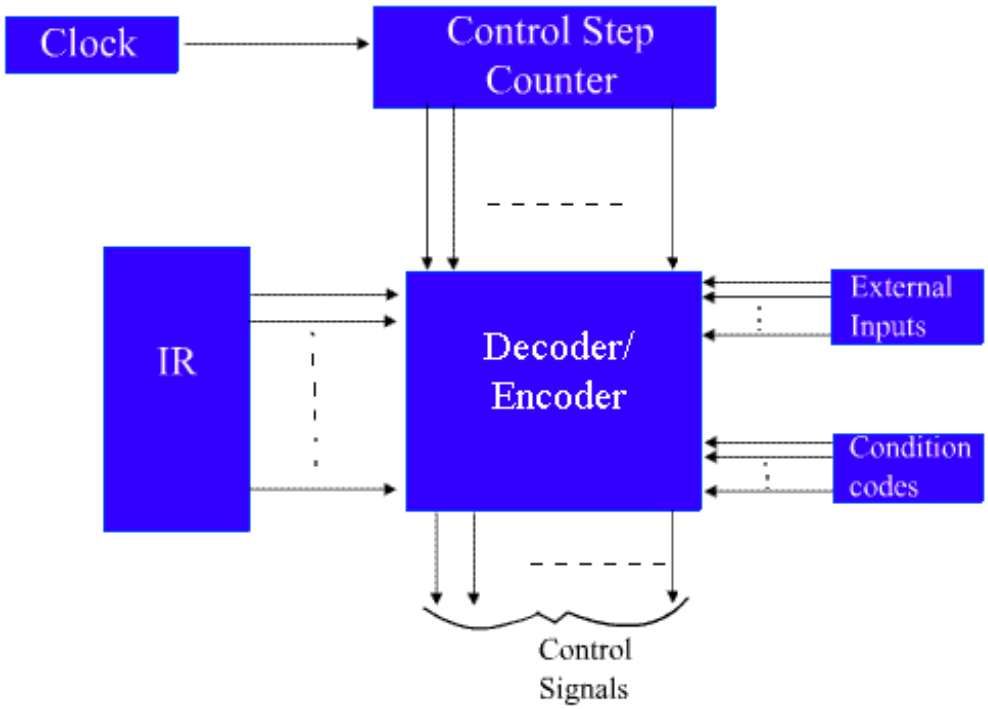
It is clear that control signals depend on the instruction, i.e., the contents of the instruction register. It is also observed that execution of some of the instructions depend on the contents of condition code or status flag register, where the control sequence depends in conditional branch instruction.

Hence, the required control signals are uniquely determined by the following information:

- **Contents of the control counter.**
- **Contents of the instruction register.**
- **Contents of the condition code and other status flags.**

The external inputs represent the state of the CPU and various control lines connected to it, such as MFC status signal. The condition codes/ status flags indicates the state of the CPU. These includes the status flags like carry, overflow, zero, etc.

Control Unit Organization



Design of Control Unit

The structure of control unit can be represented in a simplified view by putting it in block diagram. The detailed hardware involved may be explored step by step. The simplified view of the control unit is given in the **fig (A)** (prev page.)

The decoder/encoder block is simply a combinational circuit that generates the required control outputs depending on the state of all its input.

The decoder part of decoder/encoder part provide a separate signal line for each control step, or time slot in the control sequence. Similarly, the output of the instructor decoder consists of a separate line for each machine instruction loaded in the IR, one of the output line INS_1 to INS_m is set to 1 and all other lines are set to 0.

The detailed view of Control Unit organization is shown in the **fig (Next Page..)**

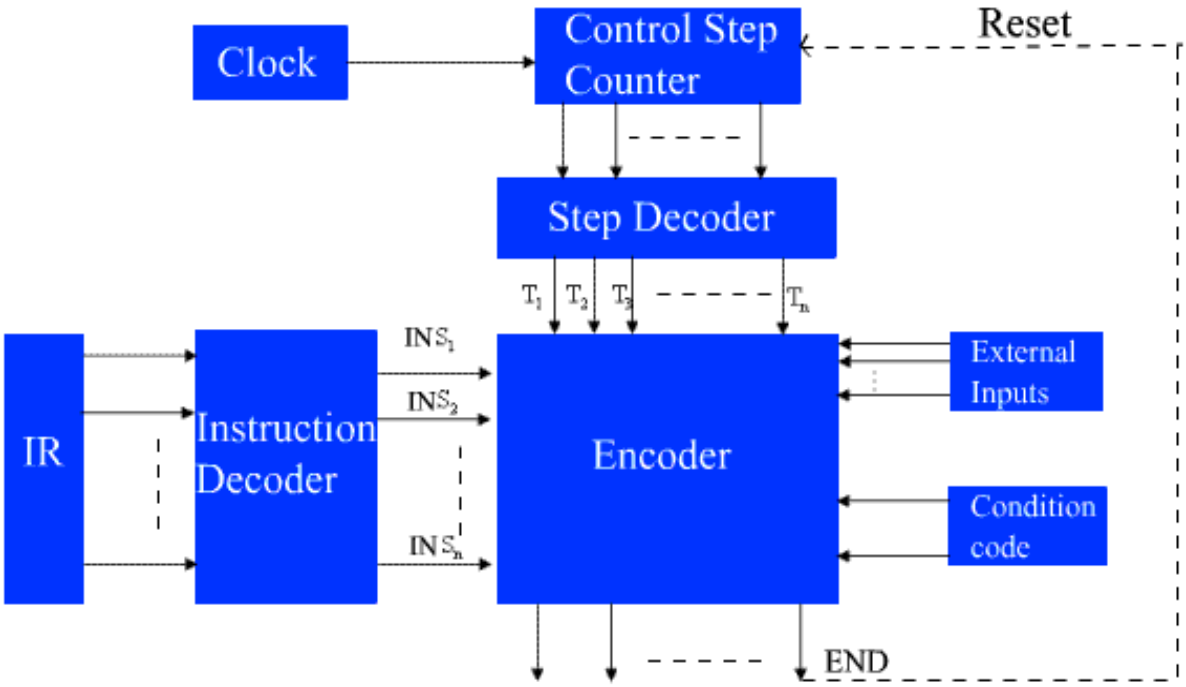


Fig (B) : Detailed view of Control Unit organization

All input signals to the encoder block should be combined to generate the individual control signals.

In the previous section, we have mentioned the control sequence of the instruction,

"Add contents of memory location address in memory direct made to register R_1 (ADD_MD)",

"Control sequence for an unconditional branch instruction (BR)",

also, we have mentioned about Branch on negative (BRN).

Consder those three CPU instruction ADD_MD, BR, BRN.

It is required to generate many control signals by the control unit. These are basically coming out from the encoder circuit of the control signal generator. The control signals are: PC_{in}, PC_{out}, Z_{in}, Z_{out}, MAR_{in}, ADD, END, etc.

By looking into the above three instructions, we can write the logic function for Z_{in} as :

$$Z_{in} = T_1 + T_6 . ADD\_MD + T_5 . BR + T_5 . BRN + \ldots\ldots\ldots\ldots$$

For all instructions, in time step1 we need the control signal Z_{in} to enable the input to register Z_{in} time cycle T_6 of ADD_MD instruction, in time cycle T_5 of BR instruction and so on.

Similarly, the Boolean logic function for ADD signal is

$$ADD = T_1 + T_6 . ADD\_MD + T_5 . BR + \ldots\ldots\ldots\ldots$$

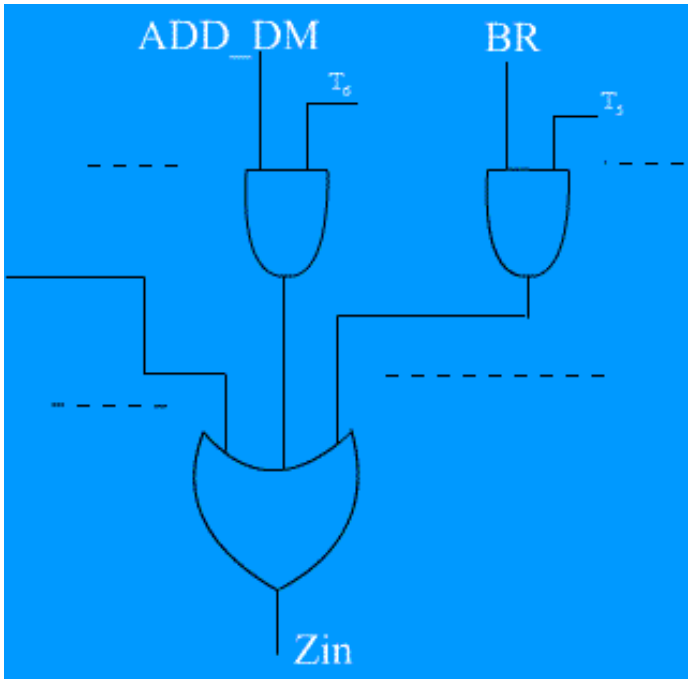
These logic functions can be implemented by a two level combinational circuit of AND and OR gates.

Similarly, the END control signal is generated by the logic function :

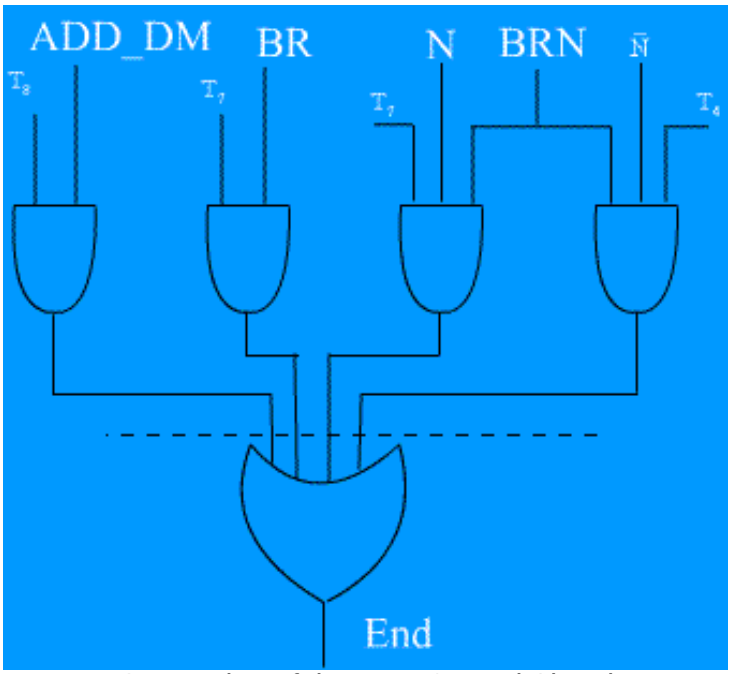
$$END = T_8 . ADD\_MD + T_7 . BR + (T_7 . N + T_4 . \overline{N}) . BRN + \ldots\ldots\ldots\ldots$$

This END signal indicates the end of the execution of an instruction, so this END signal can be used to start a new instruction fetch cycle by resetting the control step counter to its starting value.

The circuit diagram (Partial) for generating Z_{in} and END signal is shown in the diagram.



Generation of Z_{in} Control Signal



Generation of the END Control Signal

The signal *ADD_MD*, *BR*, *BRN* etc. are coming from instruction decoder circuits which depends on the contents of IR.

The signal T_1, T_2, T_3 etc are coming out from step decoder depends on control step counter.

The signal *N* (Negative) is coming from condition code register.

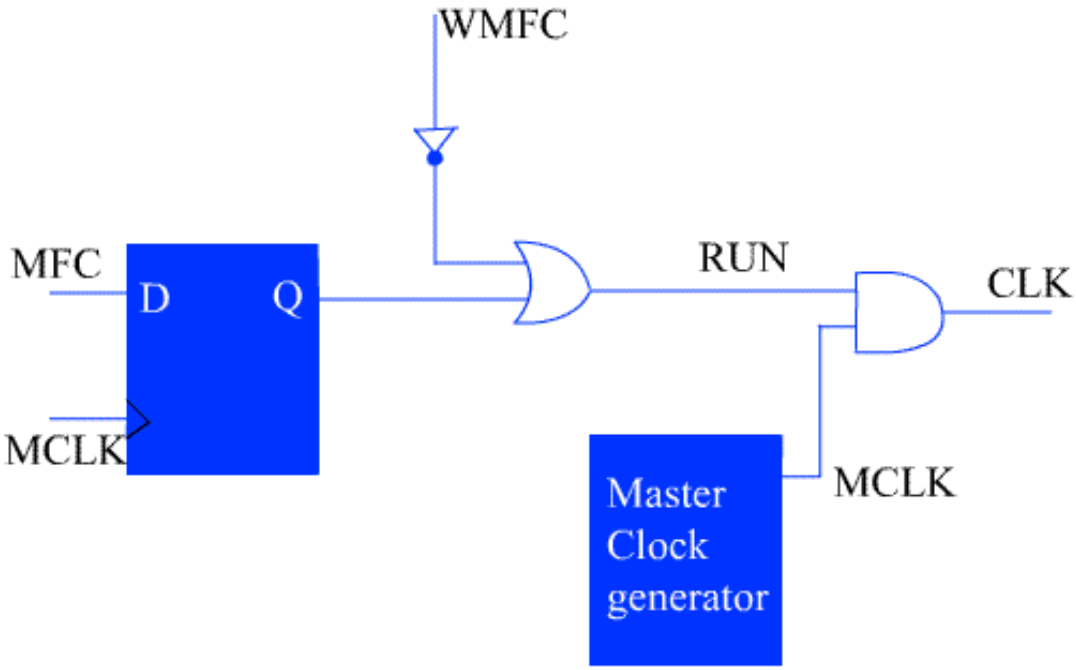
When wait for *MFC* (*WMFC*) signal is generated, then CPU does not do any works and it waits for an *MFC* signal from memory unit. In this case, the desired effect is to delay the initiation of the next control step until the *MFC* signal is received from the main memory. This can be incorporated by inhibiting the advancement of the control step counter for the required period.

Let us assume that the control step counter is controlled by a signal called *RUN*.

By looking at the control sequence of all the instructions, the *WMFC* signal is generated as:

$$WMFC = T_2 + T_5 . ADD\_MD + \ldots\ldots\ldots\ldots$$

The *RUN* signal is generated with the help of *WMFC* signal and *MFC* signal. The arrangement is shown in the figure.

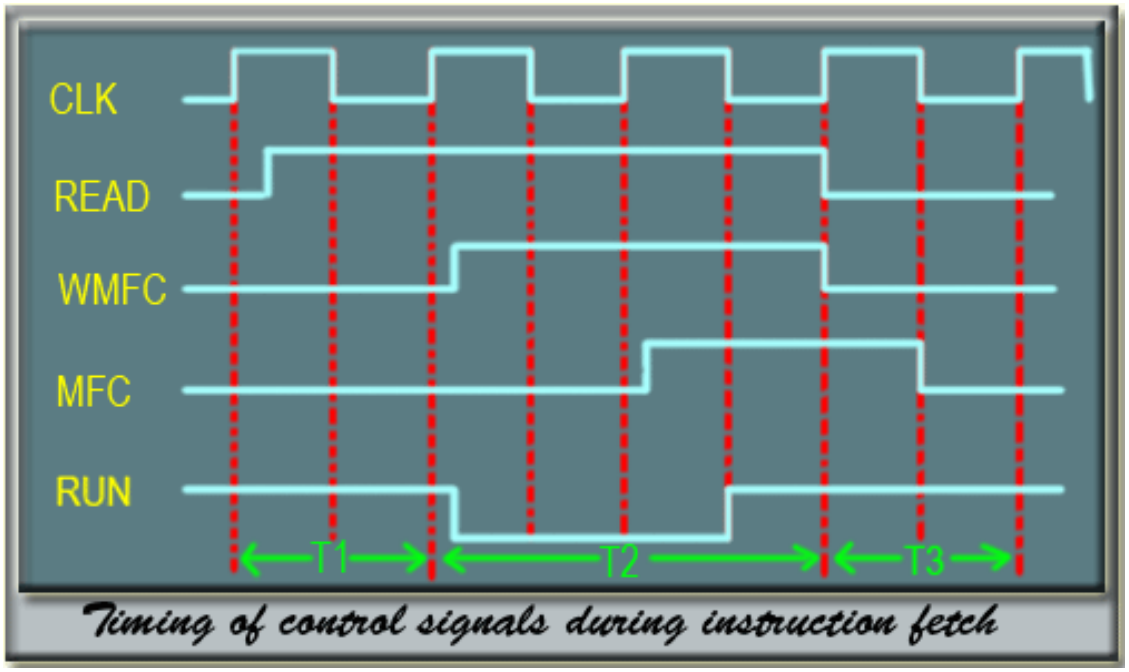


The *MFC* signal is generated by the main memory whose operation is independent of CPU clock. Hence *MFC* is an asynchronous signal that may arrive at any time relative to the CPU clock. It is possible to synchronized with CPU clock with the help of a D flip-flop.

When *WMFC* signal is high, then *RUN* signal is low. This run signal is used with the master clock pulse through an AND gate. When *RUN* is low, then the *CLK* signal remains low, and it does not allow to progress the control step counter.

When the *MFC* signal is received, the run signal becomes high and the *CLK* signal becomes same with the *MCLK* signal and due to which the control step counter progresses. Therefore, in the next control step, the *WMFC* signal goes low and control unit operates normally till the next memory access signal is generated.

The timing diagram for an instruction fetch operation is shown in the figure. (Next page..)



Timing of control signals during instruction fetch

Programmable Logic Array

In this discussion, we have presented a simplified view of the way in which the sequence of control signals needed to fetch and execute instructions may be generated.

It is observed from the discussion that as the number of instruction increases the number of required control signals will also increase.

In VLSI technology, structure that involve regular interconnection patterns are much easier to implement than the random connections.

One such regular structure is PLA (programmable logic array). PLAs are nothing but the arrays of AND gates followed by array of OR gates. If the control signals are expressed as sum of product form then with the help of PLA it can be implemented.

The PLA implementation of sequence controller is shown in the figure. (Next Page..)

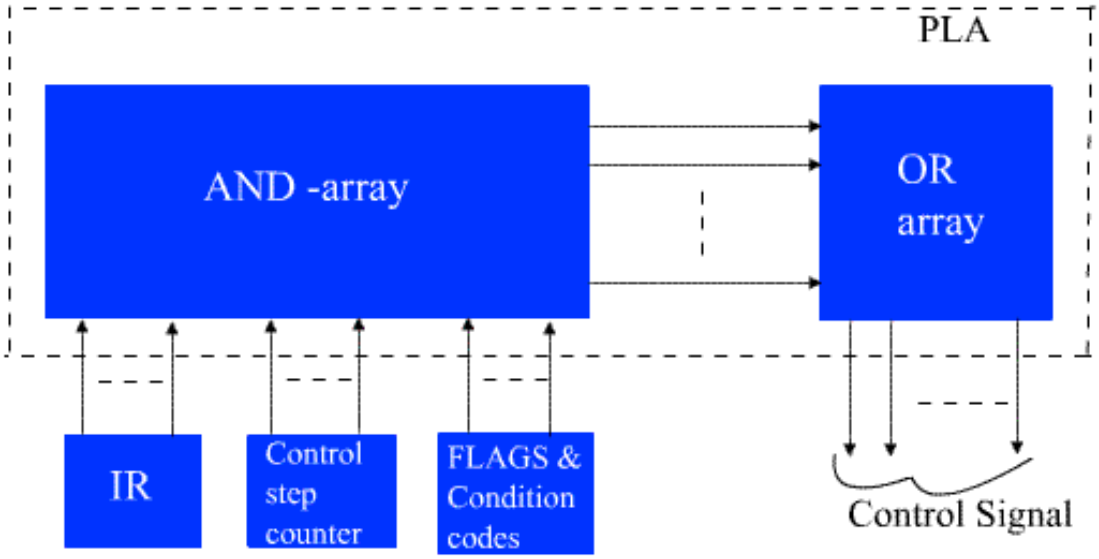


Figure D : Instruction cycle state diagram with interrupt.

Microprogrammed Control

In hardwired control, we saw how all the control signals required inside the CPU can be generated using a state counter and a PLA circuit.

There is an alternative approach by which the control signals required inside the CPU can be generated . This alternative approach is known as microprogrammed control unit.

In microprogrammed control unit, the logic of the control unit is specified by a microprogram.

A microprogram consists of a sequence of instructions in a microprogramming language. These are instructions that specify microoperations.

A microprogrammed control unit is a relatively simple logic circuit that is capable of (1) sequencing through microinstructions and (2) generating control signals to execute each microinstruction.

The concept of microprogram is similar to computer program. In computer program the complete instructions of the program is stored in main memory and during execution it fetches the instructions from main memory one after another. The sequence of instruction fetch is controlled by program counter (PC) .

Microprogram are stored in microprogram memory and the execution is controlled by microprogram counter (μPC).

Microprogram consists of microinstructions which are nothing but the strings of 0's and 1's. In a particular instance, we read the contents of one location of microprogram memory, which is nothing but a microinstruction. Each output line (data line) of microprogram memory corresponds to one control signal. If the contents of the memory cell is 0, it indicates that the signal is not generated and if the contents of memory cell is 1, it indicates to generate that control signal at that instant of time.

First let me define the different terminologies that are related to microprogrammed control unit.

Control Word (CW) :

Control word is defined as a word whose individual bits represent the various control signal. Therefore each of the control steps in the control sequence of an instruction defines a unique combination of 0s and 1s in the CW.

A sequence of control words (CWs) corresponding to the control sequence of a machine instruction constitutes the microprogram for that instruction.

The individual control words in this microprogram are referred to as microinstructions.

The microprograms corresponding to the instruction set of a computer are stored ina aspecial memory which will be referred to as the microprogram memory. The control words related to an instructions are stored in microprogram memory.

Microprogrammed Control Print this page

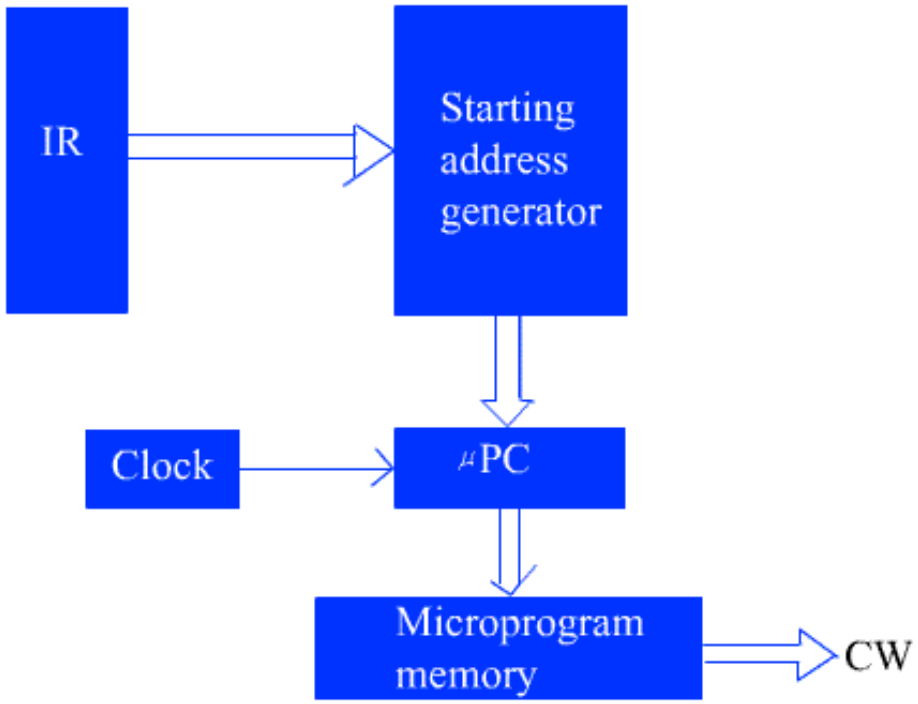
<< Previous | First | Last | Next >>

The control unit can generate the control signals for any instruction by sequencially reading the CWs of the corresponding microprogram from the microprogram memory. To read the control word sequentially from the microprogram memory a microprogram counter (μPC) is needed.

The basic organization of a microprogrammed control unit is shown in the figure.

The "starting address generator" block is responsible for loading the starting address of the microprogram into the μPC everytime a new instruction is loaded in the IR.

The μPC is then automatically incremented by the clock, and it reads the successive microinstruction from memory.



Basic organization of a microprogrammed control

<< Previous | First | Last | Next >>

Microprogrammed Control Print this page

<< Previous | First | Last | Next >>

Each microinstruction basically provides the required control signal at that time step. The microprogram counter ensures that the control signal will be delivered to the various parts of the CPU in correct sequence.

We have some instructions whose execution depends on the status of condition codes and status flag, as for example, the branch instruction. During branch instruction execution, it is required to take the decision between the alternative action.

To handle such type of instructions with microprogrammed control, the design of control unit is based on the concept of conditional branching in the microprogram. For that it is required to include some conditional branch microinstructions.

In conditional microinstructions, it is required to specify the address of the microprogram memory to which the control must direct. It is known as branch address. Apart from branch address, these microinstructions can specify which of the states flags, condition codes, or possibly, bits of the instruction register should be checked as a condition for branching to take place.

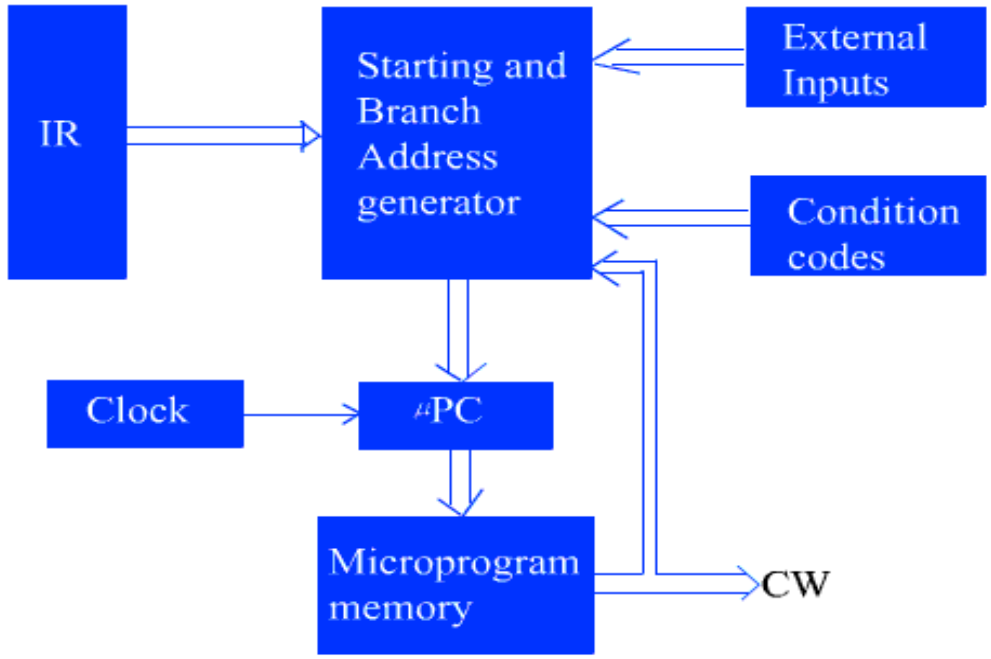
To support microprogram branching, the organization of control unit should be modified to accommodate the branching decision.

To generate the branch address, it is required to know the status of the condition codes and status flag. To generate the starting address, we need the instruction which is present in IR. But for branch address generation we have to check the content of condition codes and status flag.

>> Previous | First | Last | Next >>

Microprogrammed Control Print this page

The organization of control unit to enable conditional branching in the microprogram is shown in the figure.



Microprogrammed Control Print this page

<< Previous | First | Last | Next >>

The control bits of the microinstructions word which specify the branch conditions and address are fed to the "Starting and branch address generator" block.

This block performs the function of loading a new address into the μPC when the condition of branch instruction is satisfied.

In a computer program we have seen that execution of every instruction consists of two part - fetch phase and execution phase of the instruction. It is also observed that the fetch phase of all instruction is same.

In microprogrammed controlled control unit, a common microprogram is used to fetch the instruction. This microprogram is stored in a specific location and execution of each instruction start from that memory location.

At the end of fetch microprogram, the starting address generator unit calculate the appropriate starting address of the microprogram for the instruction which is currently present in IR. After the μPC controls the execution of microprogram which generates the appropriate control signal in proper sequence.

During the execution of a microprogram, the μPC is always incremented everytime a new microinstruction is fetched from the microprogram memory, except in the following situations :

1. When an End instruction is encountered, the μPC is loaded with the address of the first CW in the microprogram for the instruction fetch cycle.

<< Previous | First | Last | Next >>

2. When a new instruction is loaded into the IR, the μPC is loaded with the starting address of the microprogram for that instruction.
3. When a branch microinstruction is encountered, and the branch condition is satisfied, the μPC is loaded with the branch address.

Let us examine the contents of microprogram memory and how the microprogram of each instruction is stored or organized in microprogram memory. Consider the two example that are used in our previous lecture . First example is the control sequence for execution of the instruction "Add contents of memory location addressed in memory direct mode to register R1".

Steps	Actions
1.	PC_{out}, MAR_{in}, Read, Clear Y, Set carry-in to ALU, Add, Z_{in}
2.	Z_{out}, PC_{in}, Wait For MFC
3.	MDR_{out}, IR_{in}
4.	Address-field-of-IR_{out}, MAR_{in}, Read
5.	$R1_{out}$, Y_{in}, Wait for MFC
6.	MDR_{out}, Add, Z_{in}
7.	Z_{out}, $R1_{in}$
8.	END

Control sequence for Conditional Branch instruction (BRN) Branch on negative)

Steps	Actions
1.	PC_{out}, MAR_{in}, Read, Clear Y, Set Carry-in to ALU, Add , Z_{in}
2.	Z_{out}, PC_{in}, Wait for MFC
3.	MDR_{out}, IR_{in}
4.	PC_{out}, Y_{in}
5.	Address field-of IR_{out}, Add, Z_{in}
6.	Z_{out}, PC_{in}
7.	End

First consider the control signal required for fetch instruction , which is same for all the instruction, we are listing them in a particular order.

PC_{out}	MAR_{in}	Read	Clear Y	Set Carry to ALU	Add	Z_{in}	Z_{out}	PC_{in}	WMFC	MDR_{out}	IR_{in}

The control word for the first three steps of the above two instruction are : (which are the fetch cycle of each instruction as follows):

Step1	1	1	1	1	1	1	1	0	0	0	0	0	- - -
Step2	0	0	0	0	0	0	0	1	1	1	0	0	- - -
Step3	0	0	0	0	0	0	0	0	0	0	1	1	- - -

We are storing this three CW in memory location 0, 1 and 2. Each instruction starts from memory location 0. After executing upto third step, i.e., the contents of microprogram memory location 2, this control word stores the instruction in **IR**. The starting address generator circuit now calculate the starting address of the microprogram for the instruction which is available in **IR**.

Consider that the microprogram for add instruction is stored from memory location 50 of microprogram memory. So the partial contents from memory location 50 are as follows :

Location 50	0	1	1	0	0	0	0	0	0	0	0	0	- -	- -	- -
51	0	0	0	0	0	0	0	0	0	1	0	0	- -	- -	- -

and so on . . . .

Microprogrammed Control Print this page

<< Previous | First | Last | Next >>

The contents of the compile instruction is given below:

1 - PC_{in}, 2 - PC_{out}, 3 - MAR_{in}, 4 - Read, 5 - MDR_{out}, 6 - IR_{in}, 7 - $address_{out}$, 8 - Y_{in}, 9 - Clear Y,
10 - Carry-in, 11 - add, 12 - Z_{in}, 13 - Z_{out}, 14 -RI_{out}, 15 - RI_{in}, 16 - WMFC, 17 - END.

Memory Location	---------	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
0	----------	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0	----------
1		1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	----------
2		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	---------
| |																			
50		0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	
51		0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	
52		0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	
53		0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	
54		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

<< Previous | First | Last | Next >>

Microprogrammed Control Print this page

<< Previous | First | Last | Next >>

When the microprogram executes the End microinstruction of an instruction, then it generates the End control signal. This End control signal is used to load the μPC with the starting address of fetch instruction (In our case it is address 0 of microprogram memory). Now the CPU is ready to fetch the next instruction from main memory.

From the discussion, it is clear that microprograms are similar to computer program, but it is in one level lower, that's why it is called microprogram.

For each instruction of the instruction set of the CPU, we will have a microprogram.

While executing a computer program, we fetch instruction by instruction from main memory which is controlled by program counter(PC).

When we fetch an instruction from main memory, to execute that instruction , we execute the microprogram for that instruction. Microprograms are nothing but the collection of microinstrctions. These microinstructions will be fetched from microprogram memory one after another and its sequence is maintained by μPC. Fetching of microinstruction basically provides the required control signal at that time instant.

<< Previous | First | Last | Next >>

Microprogrammed Control : II

In the previous discussion, to design a micro programmed control unit, we have to do the following:

- For each instruction of the CPU, we have to write a microprogram to generate the control signal. The microprograms are stored in microprogram memory (control store). The starting address of each microprogram are known to the designer
- Each microprogram is the sequence of microintructions. And these microinstructions are executed in sequence. The execution sequence is maintained by microprogram counter.
- Each microinstructions are nothing but the combination of 0's and 1's which is known as control word. Each position of control word specifies a particular control signal. 0 on the control word means that a low signal value is generated for that control signal at that particular instant of time, similarly 1 indicates a high signal.
- Since each machine instruction is executed by a corresponding micro routine, it follows that a starting address for the micro routine must be specified as a function of the contents of the instruction register (IR).
- To incorporate the branching instruction, i.e., the branching within the microprogram, a branch address generator unit must be included. Both unconditional and conditional branching can be achieved with the help of microprogram. To incorporate the conditional branching instruction, it is required to check the contents of condition code and status flag.

Microprogrammed Control : II

Microprogramed controlled control unit is very much similar to CPU. In CPU the PC is used to fetch instruction from the main memory, but in case of control unit, microprogram counter is used to fetch the instruction from control store.

But there are some differences between these two. In case of fetching instruction from main memory, we are using two signals MFC and WMFC. These two signals are required to synchronize the speed between CPU and main memory. In general, main memory is a slower device than the CPU.

In microprogrammed control the need for such signal is less obvious. The size of control store is less than the size of main memory. It is possible to replace the control store by a faster memory, where the speed of the CPU and control store is almost same.

Since control store are usually relatively small, so that it is feasible to speed up their speed through costly circuits.

If we can implement the main memory by a faster device then it is also possible to eliminate the signals MFC & WMFC. But, in general, the size of main memory is very big and it is not economically feasible to replace the whole main memory by a faster memory to eliminate MFC & WMFC.

Grouping of control signals:

It is observed that we need to store the information of each control signal in control store. The status of a particular control signal is either high or low at a particular instant of time.

It is possible to reserve one bit position for each control signal. If there are *n* control signals in a CPU, them the length of each control signal is *n* . Since we have one bit for each control signal, so a large number of resources can be controlled with a single microinstruction. This organization of microinstruction is known as horizontal organization.

If the machine structure allows parallel uses of a number of resources, then horizontal organization has got advantage. Since more number of resources can be accessed parallel, the operating speed is also more in such organization. In this situation, horizontal organization of control store has got advantage.

If more number of resources can be accessed simultaneously, than most of the contents of control store is 0. Since the machine architecture does not provide the parallel access of resources, so simultaneously we cannot generate the control signal. In such situation, we can combine some control signals and group them together. This will reduce the size of control word. If we use compact code to specify only a small number of control functions in each microinstruction, then it is known as vertical organization of microinstruction.

In case of horizontal organization, the size of control word is longer, which is in one extreme point and in case of vertical organization, the size of control word is smaller, which is in other extreme.

In case of horizontal organization, the implementation is simple, but in case of vertical organization, implementation complexity increases due to the required decoder circuits. Also the complexity of decoder depends on the level of grouping and encoding of the control signal.

Horizontal and Vertical organization represent the two organizational extremes in microprogrammed control. Many intermediate schemes are also possible, where the degree of encoding is a design parameter.

We will explain the grouping of control signal with the help of an example. Grouping of control signals depends on the internal organization of CPU.

Assigning individual bits to each control signal is certain to lead to long microinstruction, since the number of required control signals is normally large.

However, only a few bits are set to 1 and therefore used for active gating in any given microinstructions. This obviously results in low utilization of the available bit space.

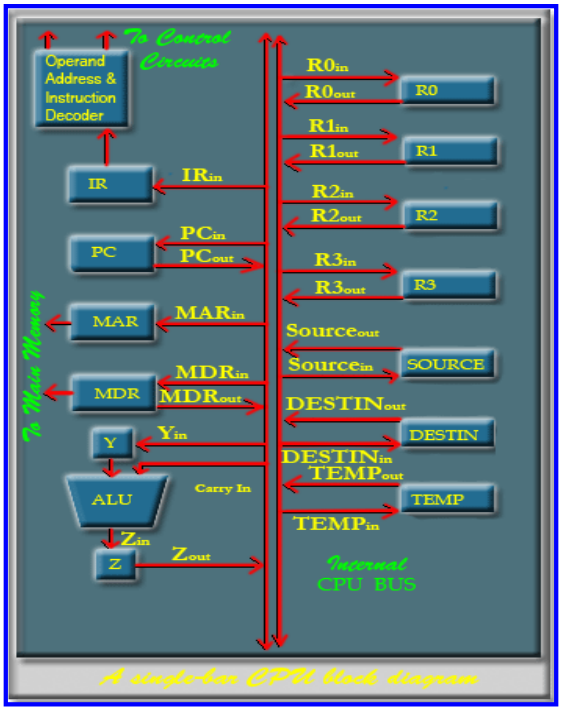
If we group the control signal in some non-over lapping group then the size of control word reduces.

Microprogrammed Control : II

Print this page

<< Previous | First | Last | Next >>

The single bus architecture of CPU is shown in the figure -



Click on Image To View Large Image

<< Previous | First | Last | Next >>

This CPU contains four general purpose registers *R0* , *R1* , *R2* and *R3* . In addition there are three other register called SOURCES, DESTIN and TEMP. These are used for temporary storage within the CPU and completely transparent to the programmer. A computer programmer cannot use these three registers.

For the proper functioning of this CPU, we need all together 24 gating signals for the transfer of information between internal CPU bus and other resources like registers.

In addition to these register gating signals, we need some other control signals which include the Read, Write, Clear Y, set carry in, WMFC, and End signal. (Here we are restricting the control signal for the case of discussion in reality, the number of signals are more).

It is also necessary to specify the function to be performed by ALU. Depending on the power of ALU, we need several control lines, one control signal for each function. Assume that the ALU that is used in the design can perform 16 different operation such as ADD, SUBSTRACT, AND, OR, etc. So we need 16 different control lines.

The above discussion indicates that 46(24+6+16) distinct signals are required. This indicates that we need 46 bits in each micro instructions, therefore the size of control word is 46.

Consider the microprogram pattern that is shown for the Add instruction. On an average 4 to 5 bits are set to 1 in each micro instruction and rest of the bits are 0. Therefore, the bit utilization is poor, and there is a scope to improve the utilization of bit.

If is observed that most signals are not needed simultaneously and many signals are mutually exclusive.

As for example, only one function of the ALU can be activated at a time. In out case we are considering 16 ALU operations. Instead of using 16 different signal for ALU operation, we can group them together and reduce the number of control signal. From digital logic circuit, it is obvious that instead of 16 different signal, we can use only 4 control signal for ALU operation and them use a 4 X 16 decoder to generate 16 different ALU signals. Due to the use of a decoder, there is a reduction in the size of control word.

Another possibilities of grouping control signal is: A sources for data transfer must be unique, which means that it is not possible to gate the contents of two different registers onto the bus at the same time. Similarly Read Write signals to the memory cannot be activated simultaneously.

This observation suggests the possibilities of grouping the signals so that all signals that are mutually exclusive are placed in the same group. Thus a group can specify one micro operation at a time.

At that point we have to use a binary coding scheme to represent a given signal within a group. As for example, for 16 ALU function, four bits are enough to decode the appropriate function.

A possible grouping of the 46 control signals that are required for the above mention CPU is given in the table.

F1 (4 bits)	F2 (3 bits)	F3 (2 bits)	F4 (2 bits)	F5 (4 bits)
0000: No Transfer	**000:** No Transfer	**00:** No Transfer	**00:** No Transfer	**0000:** Add
0001: PCout	**001:** PCin	**01:** MARin	**01:** Yin	**0001:** Sub
0010: MDRout	**001:** IRin	**10:** MDRin	**10:** SOURCEin	**0010:** MULT
0011: Zout	**011:** Zin	**11:** TEMPin	**11:** DESTINin	**0011:** Div
0100: R0out	**100:** R0in			|
0101: R1out	**101:** R1in			|
0110: R2out	**110:** R2in			|
0111: R3out	**111:** R3in			|
1000: SOURCEout				|
1001: DESTINout				|
1010: TEMPout				|
1011: ADDRESSout				**1111:** XOR

F6 (2 bits)	F7 (1 bit)	F8 (1 bit)	F9 (1 bit)	F10 (1 bit)
00: no action	**0:** no action	**0:** carry-in=0	**0:** no action	**0:** continue
01: read	**1:** clear Y	**1:** carry-in=1	**1:**WMFC	**1:** end
10: write				

A possible grouping of signal is shown here. There may be some other grouping of signal possible. Here all out-gating of registers are grouped into one group, because the contents of only one bus is allowed to goto the internal bus, otherwise there will be a conflict of data.

But the in-gating of registers are grouped into three different group. It implies that the contents of the bus may be stored into three different registers simultaneously transfer to MAR and Z. Due to this grouping, we are using 7 bits (3 +2+2) for the in-gating signal. If we would have grouped then in one group, then only 4 bits would have been enough; but it will take more time during execution. In this situation, two clock cycles would have been required to transfer the contents of PC to MAR and Z.

Therefore, the grouping of signal is a critical design parameter. If speed of operation is also a design parameter, then compression of control word will be less.

In this grouping, 46 control signals are grouped into 10 different groups (*F1* , *F2* ,……., *F10*) and the size of control word is 21. So, the size of control word is reduced from 46 to 21, which is more than 50%.

For the proper decoding, we need the following decoder:

For group *F1* & *F5* : 4 X 16 decoder,

group *F2* : 3 X 8 decoder

group *F3,F4* & *F6* : 2 X 4 decoder

Microprogram Sequencing:

In microprogrammed controlled CU,

- Each machine instruction can be implemented by a microroutine.
- Each microroutine can be accessed initially by decoding the machine instruction into the starting address to be loaded into the μPC.

Writing a microprogram for each machine instruction is a simple solution, but it will increase the size of control store.

We have already discussed that most machine instructions can operate in several addressing modes. If we write different microroutine for each addressing mode, then most of the cases, we are repeating some part of the microroutine.

The common part of the microroutine can be shared by several microroutine, which will reduse the size of control store. This results in a considerable number of branch microinstructions being needed to transfer control among various parts. So, it introduces branching capabilities within the microinstruction.

This indicates that the microprogrammed control unit has to perform two basic tasks:

- Microinstruction sequencing: Get the next microinstruction from the control memory.
- Microinstruction execution: Generate the control signals needed to execute the microinstruction.

In designing a control unit, these tasks must be considered together, because both affect the format of the microinstruction and the timing of control unit.

Design Consideration:

Two concerns are involved in the design of a microinstruction sequencing technique: the size of the microinstruction and the address generation time.

In executing a microprogram, the address of the next microinstruction to be executed is in one of these categories:

- Determined by instruction register
- Next sequential address
- Branch

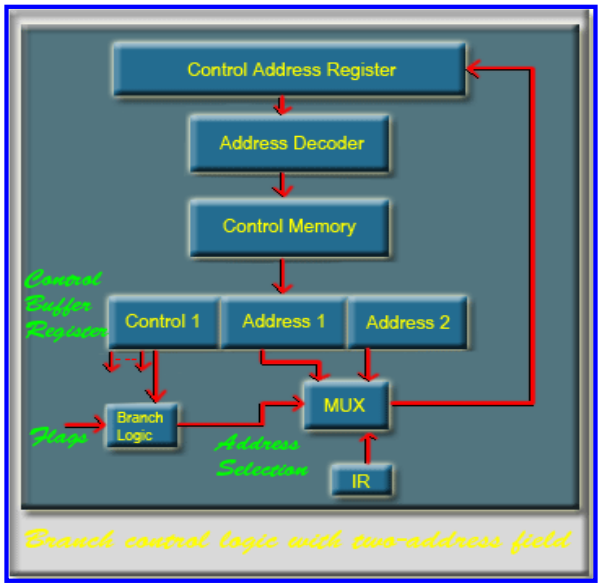
Sequencing Techniques:

Based on the current microinstruction, condition flags and the contents of the instruction register, a control memory address must be generated for the next microinstruction. A wide variety of techniques have been used and can be grouped them into three general categories:

- Two address fields
- Single address field
- Variable format.

Two Address fields:

The branch control logic with two-address field is shown in the figure below.



Click on Image To View Large Image

Microprogrammed Control : II Print this page

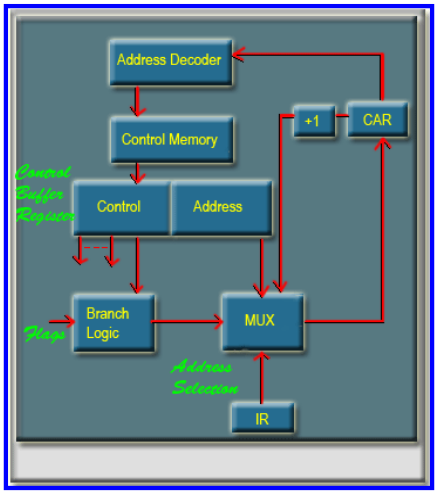
<< Previous | First | Last | Next >>

A multiplier is provided that serves as a destination for both address fields and the instruction register. Based on an address selection input, the multiplexer selects either the opcode or one of the two addresses to the control address register (CAR). The CAR is subsequently decoded to produce the next microinstruction address. The address selection signals are provided by a branch logic module whose input consists of control unit flags plus bits from the control portion of the microinstruction.

<< Previous | First | Last | Next >>

Single address field:

The two address approach is simple but it requires mere bits in the microinstruction. With some additional logic, savings can be achieved. The approach is shown in the figure:



Click on Image To View Large Image

Microprogrammed Control : II Print this page

<< Previous | First | Last | Next >>

In this single address field branch control logic, the options for next address are as follows:

- Address field
- Instruction register code
- Next sequential address

The address selection signals determine which option to be selected. This approach reduce the number of address fields to one.

>> Previous | First | Last | Next >>

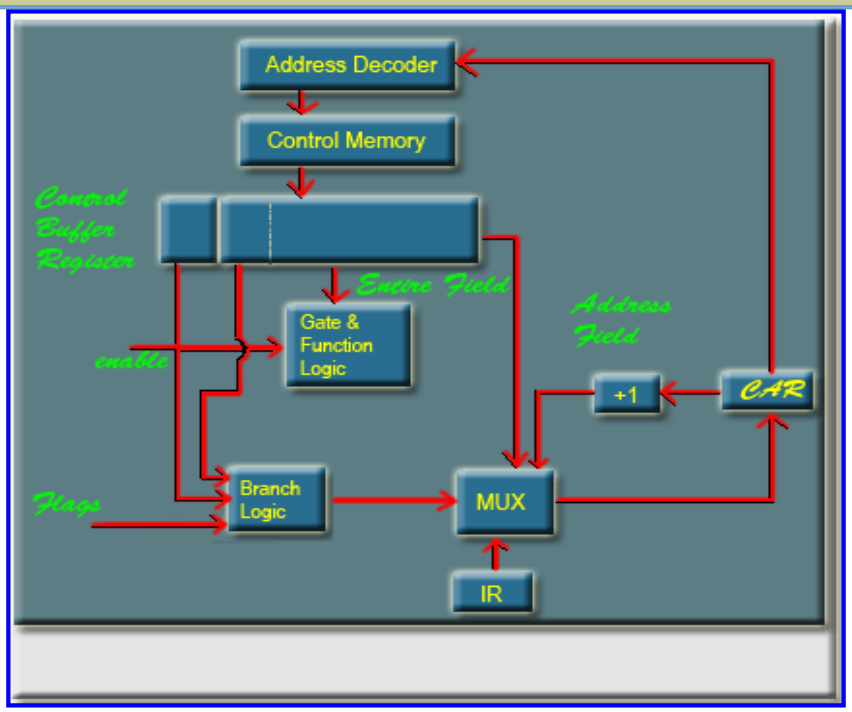
Variable format:

In variable format branch control logic one bit designates which format is being used. In one format, the remaining bits are used to active control signals. In the other format, some bits drive the branch logic module, and the remaining bits provide the address. With the first format, the next address is either the next sequential address or an address derived from the instruction register. With the second format, either a conditional or unconditional branch is being specified. The approach is shown in the figure of the next slide .

Microprogrammed Control : II

Print this page

<< Previous | First | Last | Next >>



Click on Image To View Large Image

<< Previous | First | Last | Next >>

Address Generation:

We have looked at the sequencing problem from the point of view of format consideration and general logic requirements. Another viewpoint is to consider the various ways in which the next address can be derived or computed.

Various address generation Techniques	
Explicit	*Implicit*
Two-field	Mapping
Unconditional branch	Addition
Conditional branch	Residual control

The address generation technique can be divided into two techniques: explicit & implicit.

In explicit technique, the address is explicitly available in the microinstruction.

In implicit technique, additional logic circuit is used to generate the address.

Microprogrammed Control : II Print this page

<< Previous | First | Last | Next >>

In two address field approach, signal address field or a variable format, various branch instruction can be implemented with the explicit approaches.

In implicit technique, mapping is required to get the address of next instruction. The opcode portion of a machine instruction must be mapped into a microinstruction address.

<< Previous | First | Last | Next >>

Computer Organization and Architecture Print this page

Module 06 : Input / Output

In this Module, we have four lectures, viz.

1. **Introduction to I/O**
2. **Program Controlled I/O**
3. **Interrupt Controlled I/O**
4. **Direct Memory Access**

Click the proper link on the left side for the lectures

Input/Output Organization

- The computer system's *input/output* (I/O) architecture is its interface to the outside world.
- Till now we have discussed the two important modules of the computer system -
  - **The processor** and
  - **The memory** module.

- The third key component of a computer system is a set of **I/O modules**

- Each I/O module interfaces to the system bus and controls one or more peripheral devices.

Introduction to I/O

There are several reasons why an I/O device or peripheral device is not directly connected to the system bus. Some of them are as follows -

- There are a wide variety of peripherals with various methods of operation. It would be impractical to include the necessary logic within the processor to control several devices.
- The data transfer rate of peripherals is often much slower than that of the memory or processor. Thus, it is impractical to use the high-speed system bus to communicate directly with a peripheral.
- Peripherals often use different data formats and word lengths than the computer to which they are attached.

Thus, an I/O module is required.

Input/Output Modules

The major functions of an I/O module are categorized as follows –

- Control and timing
- Processor Communication
- Device Communication
- Data Buffering
- Error Detection

During any period of time, the processor may communicate with one or more external devices in unpredictable manner, depending on the program's need for I/O.

The internal resources, such as main memory and the system bus, must be shared among a number of activities, including data I/O.

Control & timings:

The I/O function includes a control and timing requirement to co-ordinate the flow of traffic between internal resources and external devices.

For example, the control of the transfer of data from an external device to the processor might involve the following sequence of steps –

a. The processor interacts with the I/O module to check the status of the attached device.
b. The I/O module returns the device status.
c. If the device is operational and ready to transmit, the processor requests the transfer of data, by means of a command to the I/O module.
d. The I/O module obtains a unit of data from external device.
e. The data are transferred from the I/O module to the processor.

If the system employs a bus, then each of the interactions between the processor and the I/O module involves one or more bus arbitrations.

Processor & Device Communication

During the I/O operation, the I/O module must communicate with the processor and with the external device.

Processor communication involves the following -

Command decoding :

The I/O module accepts command from the processor, typically sent as signals on control bus.

Data :

Data are exchanged betweeen the processor and the I/O module over the data bus.

Status Reporting :

Because peripherals are so slow, it is important to know the status of the I/O module. For example, if an I/O module is asked to send data to the processor(read), it may not be ready to do so because it is still working on the previous I/O command. This fact can be reported with a status signal. Common status signals are ***BUSY*** and ***READY***.

Address Recognition :

Just as each word of memory has an address, so thus each of the I/O devices. Thus an I/O module must recognize one unique address for each peripheral it controls.

One the other hand, the I/O must be able to perform device communication. This communication involves command, status information and data.

Data Buffering:

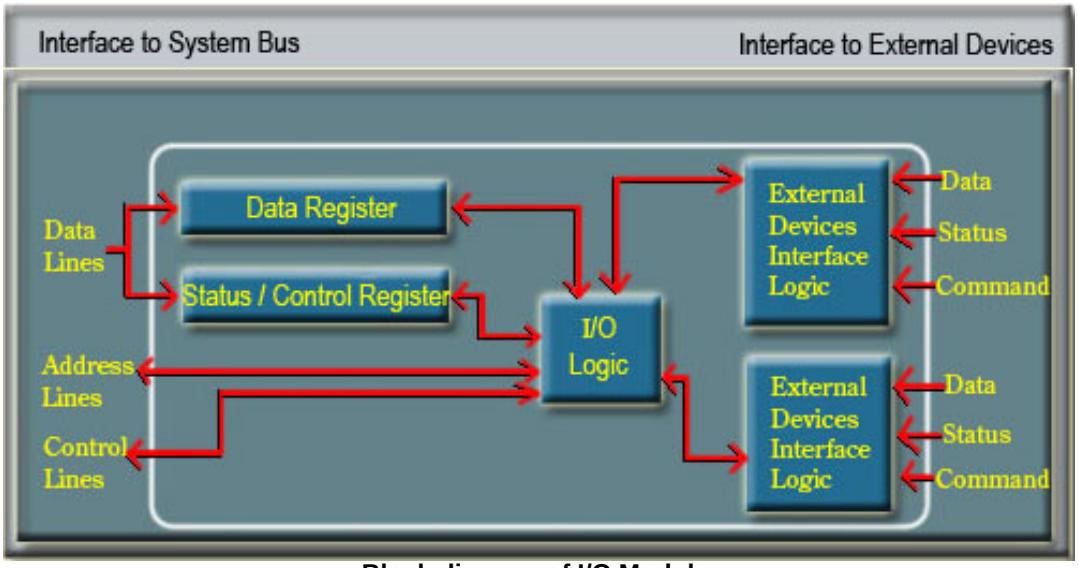
An essential task of an I/O module is data buffering. The data buffering is required due to the mismatch of the speed of CPU, memory and other peripheral devices. In general, the speed of CPU is higher than the speed of the other peripheral devices. So, the I/O modules store the data in a data buffer and regulate the transfer of data as per the speed of the devices.

In the opposite direction, data are buffered so as not to tie up the memory in a slow transfer operation. Thus the I/O module must be able to operate at both device and memory speed.

Error Detection:

Another task of I/O module is error detection and for subsequently reporting error to the processor. One class or error includes mechanical and electrical malfunctions reported by the device (e.g. paper jam). Another class consists of unintentional changes to the bit pattern as it is transmitted from devices to the I/O module.

Block diagram of I/O Module is shown in the figure.



Block diagram of I/O Module.

Introduction to I/O

There will be many I/O devices connected through I/O modules to the system. Each device will be indentified by a unique address.

When the processor issues an I/O command, the command contains the address of the device that is used by the command. The I/O module must interpret the addres lines to check if the command is for itself.

Generally in most of the processors, the processor, main memory and I/O share a common bus(data address and control bus).

Two types of addressing are possible -

- Memory-mapped I/O
- Isolated or I/O mapped I/O

Memory-mapped I/O:

There is a single address space for memory locations and I/O devices.

The processor treats the status and address register of the I/O modules as memory location.

For example, if the size of address bus of a processor is 16, then there are 2^{16} combinations and all together 2^{16} address locations can be addressed with these 16 address lines.

Out of these 2^{16} address locations, some address locations can be used to address I/O devices and other locations are used to address memory locations.

Since I/O devices are included in the same memory address space, so the status and address registers of I/O modules are treated as memory location by the processor. Therefore, the same machine instructions are used to access both memory and I/O devices.

Isolated or I/O -mapped I/O:

In this scheme, the full range of addresses may be available for both.

The address refers to a memory location or an I/O device is specified with the help of a command line.

In general $IO/\overline{M}$ command line is used to identify a memory location or an I/O device.

if $IO/\overline{M}$ =1, it indicates that the address present in address bus is the address of an I/O device.

if $IO/\overline{M}$ =0, it indicates that the address present in address bus is the address of a memory location.

Since full range of address is available for both memory and I/O devices, so, with 16 address lines, the system may now support both 2^{16} memory locations and 2^{16} I/O addresses.

Input / Output Subsystem

There are three basic forms of input and output systems –

- **Programmed I/O**
- **Interrupt driven I/O**
- **Direct Memory Access(DMA)**

With programmed I/O, the processor executes a program that gives its direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data.

With interrupt driven I/O, the processor issues an I/O command, continues to execute other instructions, and is interrupted by the I/O module when the I/O module completes its work.

In Direct Memory Access (DMA), the I/O module and main memory exchange data directly without processor involvement.

Program controlled I/O Print this page

<< Previous | First | Last | Next >>

With both programmed I/O and Interrupt driven I/O, the processor is responsible for extracting data from main memory for output operation and storing data in main memory for input operation.

To send data to an output device, the CPU simply moves that data to a *special memory location* in the I/O address space if I/O mapped input/output is used or to an address in the memory address space if memory mapped I/O is used.

Data	I/O Address Space (in memory)	if I/O mapped input/output is used
	memory address space	if memory mapped I/O is used

To read data from an input device, the CPU simply moves data from the address (I/O or memory) of that device into the CPU.

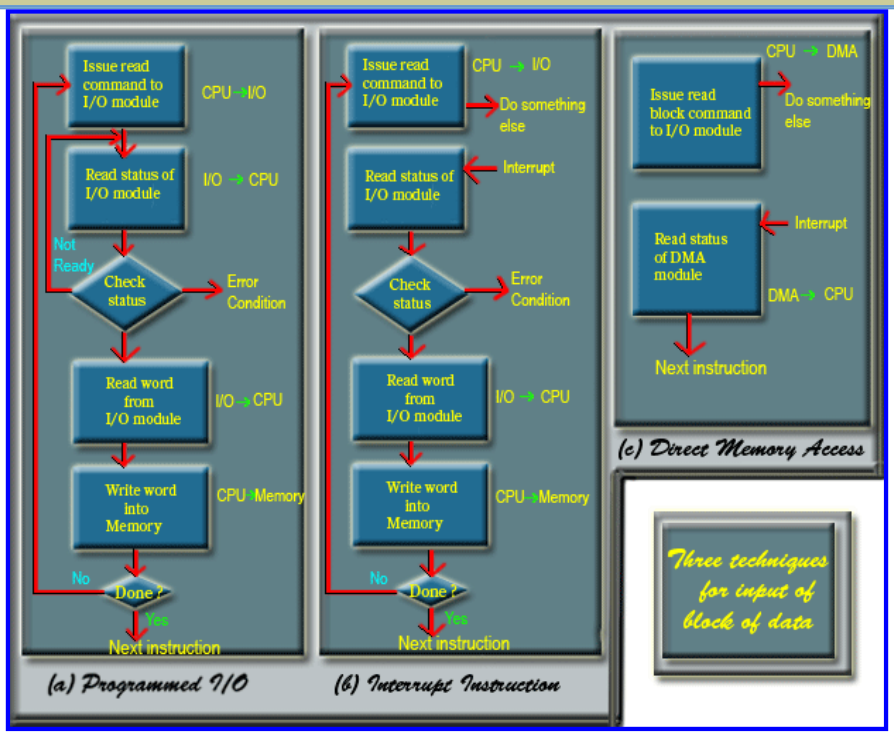
Input/Output Operation: The input and output operation looks very similar to a memory read or write operation except it usually takes *more time* since peripheral devices are slow in speed than main memory modules.

>> Previous | First | Last | Next >>

Program controlled I/O

Print this page

<< Previous | First | Last | Next >>



Click on Image To View Large Image

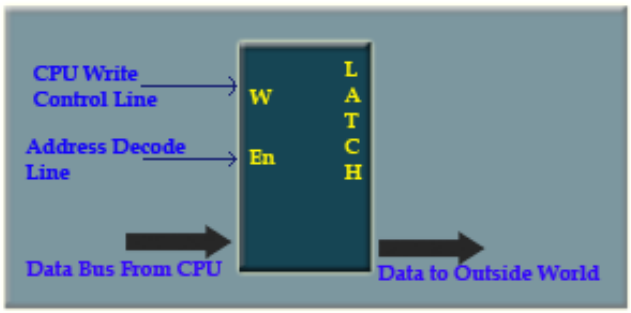
<< Previous | First | Last | Next >>

Input/Output Port

An *I/O port* is a device that looks like a memory cell to the computer but contains connection to the outside world.

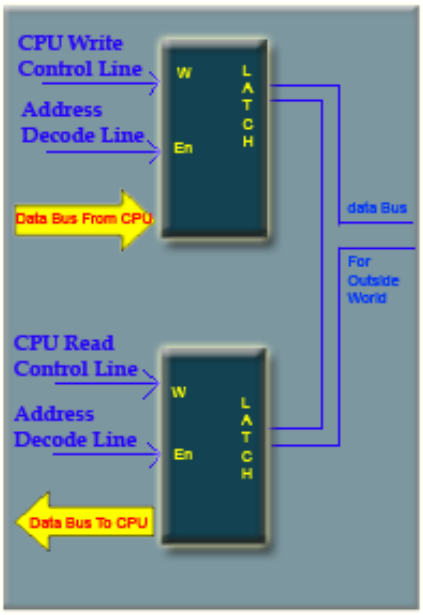
An *I/O port* typically uses a *latch*. When the CPU writes to the address associated with the latch, the latch device captures the data and makes it available on a set of wires external to the CPU and memory system.

The *I/O ports* can be *read-only*, *write-only*, or *read/write*. The *write-only* port is shown in the figure.



Program controlled I/O

First, the CPU will place the address of the device on the *I/O address bus* and with the help of *address decoder* a signal is generated which will enable the latch.



Next, the CPU will indicate the operation is a write operation by putting the appropriate signal in CPU write control line.

Then the data to be transferred will be placed in the CPU bus, which will be stored in the latch for the onward transmission to the device.

Both the address decode and write control lines must be active for the latch to operate.

The *read/write* or *input/output* port is shown in the figure.

The device is identified by putting the appropriate address in the I/O address lines. The address decoder will generate the signal for the address decode lines. According to the operation, *read* or *write*, it will select either of the latch.

If it is a write operation, then data will be placed in the latch from CPU for onward transmission to the output device.

If it is in a read operation, the data that are already stored in the latch will be transferred to the CPU.

A *read only* (input) port is simply the lower half of the figure.

In case of I/O mapped I/O, a different address space is used for I/O devices. The address space for memory is different. In case of memory mapped I/O, same address space is used for both memory and I/O devices. Some of the memory address space are kept reserved for I/O devices.

To the programmer, the difference between I/O-mapped and memory-mapped input/output operation is the instruction to be used.

For memory-mapped I/O, any instruction that accessed memory can access a memory-mapped I/O port.

I/O-mapped input/output uses special instruction to access I/O port.

Program controlled I/O Print this page

<< Previous | First | Last | Next >>

Generally, a given peripheral device will use more than a single I/O port. A typical *PC* parallel printer interface, for example, uses three ports, a *read/write port*, and *input port* and an *output port*.

The read/write port is the data port (it is read/write to allow the CPU to read the last ASCII character it wrote to the printer port).

The input port returns control signals from the printer.

- These signals indicate whether the printer is ready to accept another character, is off-line, is out of paper, etc.

The output port transmits control information to the printer such as

- whether data is available to print.

Memory-mapped I/O subsystems and I/O-mapped subsystems both require the CPU to move data between the peripheral device and main memory.

For example, to input a sequence of 20 bytes from an input port and store these bytes into memory, the CPU must send each value and store it into memory.

<< Previous | First | Last | Next >>

Programmed I/O:

In programmed I/O, the data transfer between CPU and I/O device is carried out with the help of a software routine.

When a processor is executing a program and encounters an instruction relating to I/O, it executes that I/O instruction by issuing a command to the appropriate I/O module.

The I/O module will perform the requested action and then set the appropriate bits in the I/O status register.

The I/O module takes no further action to alert the processor.

It is the responsibility of the processor to check periodically the status of the I/O module until it finds that the operation is complete.

In programmed I/O, when the processor issuses a command to a I/O module, it must wait until the I/O operation is complete.

Generally, the I/O devices are slower than the processor, so in this scheme CPU time is wasted. CPU is checking the status of the I/O module periodically without doing any other work.

I/O Commands

To execute an I/O-related instruction, the processor issues an address, specifying the particular I/O module and external device, and an I/O command. There are four types of I/O commands that an I/O module will receive when it is addressed by a processor –

- **Control :** Used to activate a peripheral device and instruct it what to do. For example, a magnetic tape unit may be instructed to rewind or to move forward one record. These commands are specific to a particular type of peripheral device.
- **Test :** Used to test various status conditions associated with an I/O module and its peripherals. The processor will want to know if the most recent I/O operation is completed or any error has occurred.
- **Read :** Causes the I/O module to obtain an item of data from the peripheral and place it in the internal buffer.
- **Write :** Causes the I/O module to take an item of data (byte or word) from the data bus and subsequently transmit the data item to the peripheral.

Interrupt driven I/O

The problem with programmed I/O is that the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data. The processor, while waiting, must repeatedly interrogate the status of the I/O module.

This type of I/O operation, where the CPU constantly tests a part to see if data is available, is polling, that is, the CPU Polls (asks) the port if it has data available or if it is capable of accepting data. Polled I/O is inherently inefficient.

The solution to this problem is to provide an interrupt mechanism. In this approach the processor issues an I/O command to a module and then go on to do some other useful work. The I/O module then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer. Once the data transfer is over, the processor then resumes its former processing.

Let us consider how it works

A. From the point of view of the I/O module:

- For input, the I/O module services a READ command from the processor.
- The I/O module then proceeds to read data from an associated peripheral device.
- Once the data are in the modules data register, the module issues an interrupt to the processor over a control line.
- The module then waits until its data are requested by the processor.
- When the request is made, the module places its data on the data bus and is then ready for another I/O operation.

B. From the processor point of view; the action for an input is as follows: :

- The processor issues a READ command.
- It then does something else
  (e.g. the processor may be working on several different programs at the same time)
- At the end of each instruction cycle, the processor checks for interrupts
- When the interrupt from an I/O module occurs, the processor saves the context
  (e.g. program counter & processor registers) of the current program and processes the interrupt.
- In this case, the processor reads the word of data from the I/O module and stores it in memory.
- It then restores the context of the program it was working on and resumes execution.

Interrupt Processing

The occurrence of an interrupt triggers a number of events, both in the processor hardware and in software.

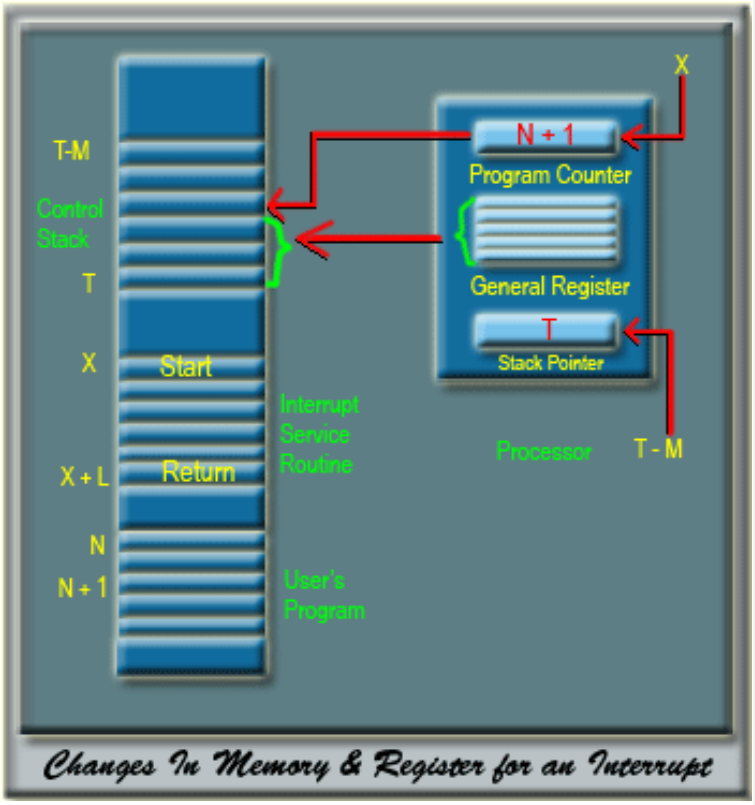
When an I/O device completes an I/O operation, the following sequences of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt.
3. The processor tests for the interrupt; if there is one interrupt pending, then the processor sends an acknowledgement signal to the device which issued the interrupt. After getting acknowledgement, the device removes its interrupt signals.
4. The processor now needs to prepare to transfer control to the interrupt routine. It needs to save the information needed to resume the current program at the point of interrupt. The minimum information required to save is the processor status word (PSW) and the location of the next instruction to be executed which is nothing but the contents of program counter. These can be pushed into the system control stack.
5. The processor now loads the program counter with the entry location of the interrupt handling program that will respond to the interrupt.

Interrupt Control I/O

Print this page

<< Previous | First | Last | Next >>



Changes In Memory & Register for an Interrupt

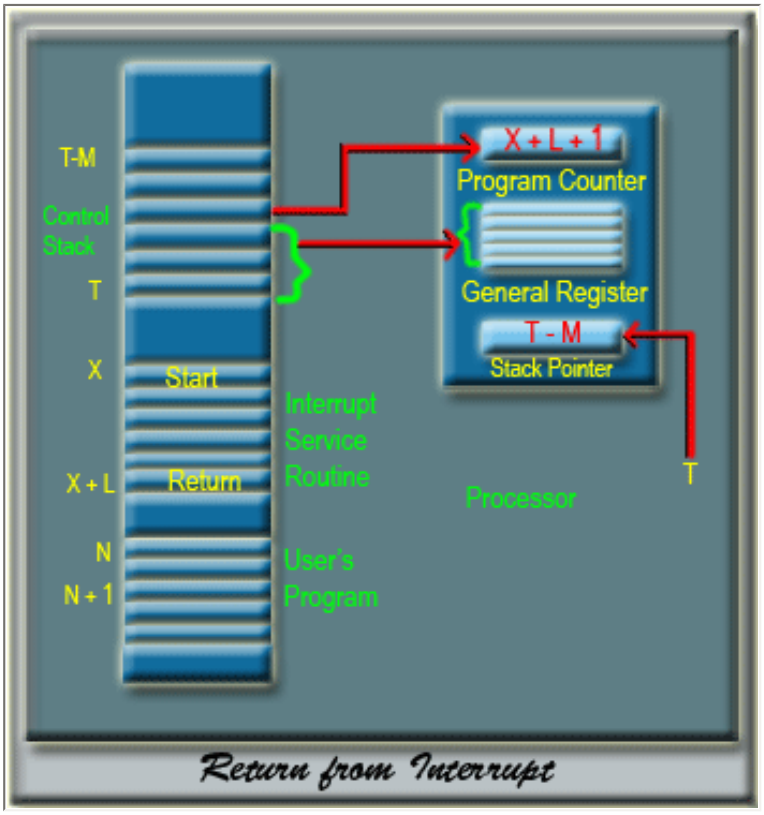
Interrupt Processing:

- An interrupt occurs when the processor is executing the instruction of location *N*.
- At that point, the value of program counter is *N+1* .
- Processor services the interrupt after completion of current instruction execution.
- First, it moves the content of general registers to system stack.
- Then it moves the program counter value to the system stack.
- Top of the system stack is maintained by stack pointer.
- The value of stack pointer is modified to point to the top of the stack.
- If M elsments are moved to the system stack, the value of stack pointer is changed from *T* to *T-M*.

<< Previous | First | Last | Next >>

- Next, the program counter is loaded with the starting address of the interrupt service routine.
- Processor starts executing the interrupt service routine.

Interrupt Control I/O



Return from Interrupt :

- Interrupt service routine starts at location *X* and the return instruction is in location *X + L*.
- After fetching the return instruction, the value of program counter becomes *X + L + 1*.
- While returning to user's program, processor must restore the earlier values.
- From control stack, it restores the value of program counter and the general registers.
- Accordingly it sets the value of the top of the stack and accordingly stack pointer is updated.
- Now the processor starts execution of the user's program (interrupted program) from memory location *N + 1*.

Interrupt Control I/O

Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an interrupt fetch. The control will transfer to interrupt handler routine for the current interrupt.

The following operations are performed at this point.

6. At the point, the program counter and PSW relating to the interrupted program have been saved on the system stack. In addition to that some more information must be saved related to the current processor state which includes the control of the processor registers, because these registers may be used by the interrupt handler. Typically, the interrupt handler will begin by saving the contents of all registers on stack.

7. The interrupt handles next processes the interrupt. This includes an examination of status information relating to the I/O operation or, other event that caused an interrupt.

8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers.

9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

Design Issues for Interrupt

Two design issues arise in implementing interrupt I/O.

- There will almost invariably be multiple I/O modules, how does the processor determine which device issued the interrupt?
- If multiple interrupts have occurred how the processor does decide which one to process?

Device Identification

Four general categories of techniques are in common use:

- **Multiple interrupt lines**
- **Software poll**
- **Daisy chain (hardware poll, vectored)**
- **Bus arbitration (vectored)**

Multiple Interrupts Lines:

The most straight forward approach is to provide multiple interrupt lines between the processor and the I/O modules.

It is impractical to dedicate more than a few bus lines or processor pins to interrupt lines.

Thus, though multiple interrupt lines are used, it is most likely that each line will have multiple I/O modules attached to it. Thus one of the other three techniques must be used on each line.

Software Poll :

When the processor detects an interrupt, it branches to an interrupt service routine whose job is to poll each I/O module to determine which module caused the interrupt.

The poll could be implemented with the help of a separate command line (e.g. TEST I/O). In this case, the processor raises TEST I/O and place the address of a particular I/O module on the address lines. The I/O module responds positively if it set the interrupt.

Alternatively, each I/O module could contain an addressable status register. The processor then reads the status register of each I/O module to identify the interrupting module.

Once the correct module is identified, the processor branches to a device service routine specific to that device.

The main disadvantage of software poll is that it is time consuming. Processor has to check the status of each I/O module and in the worst case it is equal to the number of I/O modules.

Daisy Chain :

In this method for interrupts all I/O modules share a common interrupt request lines. However the interrupt acknowledge line is connected in a daisy chain fashion. When the processor senses an interrupt, it sends out an interrupt acknowledgement.

The interrupt acknowledge signal propagates through a series of I/O module until it gets to a requesting module.

The requesting module typically responds by placing a word on the data lines. This word is referred to as a vector and is either the address of the I/O module or some other unique identification.

In either case, the processor uses the vector as a pointer to the appropriate device service routine. This avoids the need to execute a general interrupt service routine first. This technique is referred to as a *vectored interrupt*.

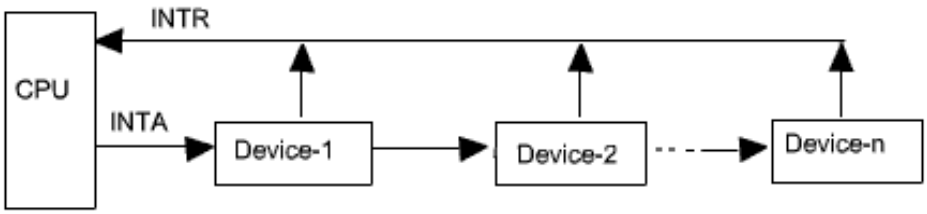


Fig : Daisy chain arrangement

Bus Arbitration :

In bus arbitration method, an I/O module must first gain control of the bus before it can raise the interrupt request line. Thus, only one module can raise the interrupt line at a time. When the processor detects the interrupt, it responds on the interrupt acknowledge line. The requesting module then places it vector on the data line.

Handling multiple interrupts

There are several techniques to identify the requesting I/O module. These techniques also provide a way of assigning priorities when more than one device is requesting interrupt service.

With multiple lines, the processor just picks the interrupt line with highest priority. During the processor design phase itself priorities may be assigned to each interrupt lines.

With software polling, the order in which modules are polled determines their priority.

In case of daisy chain configuration, the priority of a module is determined by the position of the module in the daisy chain. The module nearer to the processor in the chain has got higher priority, because this is the first module to receive the acknowledge signal that is generated by the processor.

In case of bus arbitration method, more than one module may need control of the bus. Since only one module at a time can successfully transmit over the bus, some method of arbitration is needed. The various methods can be classified into two group – centralized and distributed.

Interrupt Control I/O

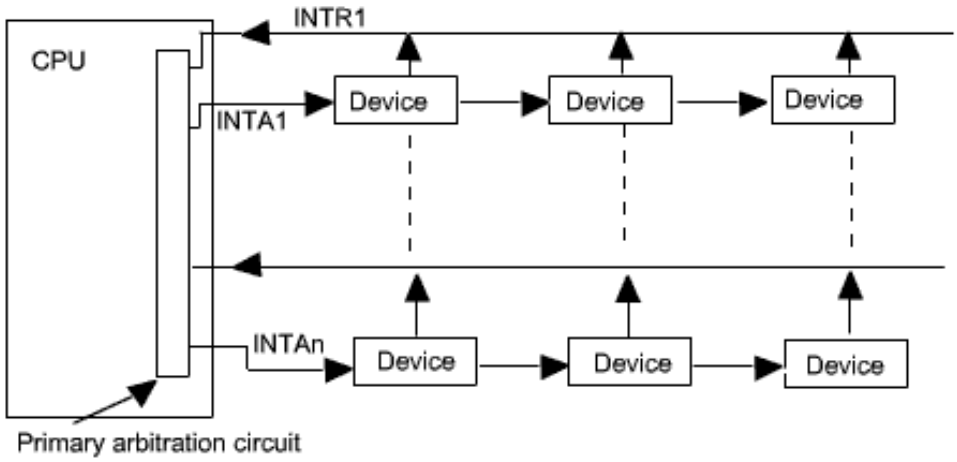
In a centralized scheme, a single hardware device, referred to as a bus controller or arbiter is responsible for allocating time on the bus. The device may be a separate module or part of the processor.

In distributed scheme, there is no central controller. Rather, each module contains access control logic and the modules act together to share the bus.

It is also possible to combine different device identification techniques to identify the devices and to set the priorities of the devices. As for example multiple interrupt lines and daisy chain technologies can be combined together to give access for more devices.

In one interrupt line, more than one device can be connected in daisy chain fashion. The High priorities devices should be connected to the interrupt lines that has got higher priority.

A possible arrangement is shown in the figure.



Interrupt Nesting

The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program and starts the execution of another. The execution of this another program is nothing but the interrupt service routine for that specified device.

Interrupt may arrive at any time. So during the execution of an interrupt service routine, another interrupt may arrive. This kind of interrupts are known as *nesting of interrupt*.

Whether interrupt nesting is allowed or not? This is a design issue. Generally nesting of interrupt is allowed, but with some restrictions. The common notion is that a high priority device may interrupt a low priority device, but not the vice-versa.

To accomodate such type of restrictions, all computer provide the programmer with the ability to enable and disable such interruptions at various time during program execution. The processor provides some instructions to enable the interrupt and disable the interrupt. If interrupt is disabled, the CPU will not respond to any interrupt signal.

On the other hand, when multiple lines are used for interrupt and priorities are assigned to these lines, then the interrupt received in a low priority line will not be served if an interrupt routine is in execution for a high priority device. After completion of the interrupt service routine of high priority devices, processor will respond to the interrupt request of low priority devices

Direct Memory Access

We have discussed the data transfer between the processor and I/O devices. We have discussed two different approaches namely *programmed I/O* and *Interrupt-driven I/O*. Both the methods require the active intervention of the processor to transfer data between memory and the I/O module, and any data transfer must transverse a path through the processor. Thus both these forms of I/O suffer from two inherent drawbacks.

- The I/O transfer rate is limited by the speed with which the processor can test and service a device.
- The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

To transfer large block of data at high speed, a special control unit may be provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor. This approach is called *direct memory access* or DMA.

DMA transfers are performed by a control circuit associated with the I/O device and this circuit is referred as DMA controller. The DMA controller allows direct data transfer between the device and the main memory without involving the processor.

Direct Memory Access [DMA]

Print this page

<< Previous | First | Last | Next >>

To transfer data between memory and I/O devices, DMA controller takes over the control of the system from the processor and transfer of data take place over the system bus. For this purpose, the DMA controller must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily. The later technique is more common and is referred to as cycle stealing, because the DMA module in effect steals a bus cycle.

The typical block diagram of a DMA controller is shown in the figure.

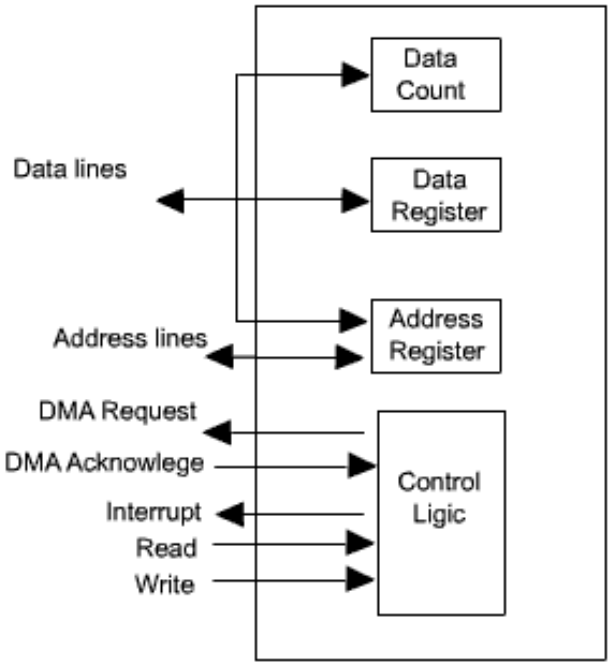


Fig : Typical DMA block diagram

<< Previous | First | Last | Next >>

Direct Memory Access [DMA]

When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information.

- Whether a read or write is requested, using the read or write control line between the processor and the DMA module.
- The address of the I/O devise involved, communicated on the data lines.
- The starting location in the memory to read from or write to, communicated on data lines and stored by the DMA module in its address register.
- The number of words to be read or written again communicated via the data lines and stored in the data count register.

The processor then continues with other works. It has delegated this I/O operation to the DMA module.

The DMA module checks the status of the I/O devise whose address is communicated to DMA controller by the processor. If the specified I/O devise is ready for data transfer, then DMA module generates the DMA request to the processor. Then the processor indicates the release of the system bus through DMA acknowledge.

The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor.

When the transfer is completed, the DMA module sends an interrupt signal to the processor. After receiving the interrupt signal, processor takes over the system bus.

Thus the processor is involved only at the beginning and end of the transfer. During that time the processor is suspended.

It is not required to complete the current instruction to suspend the processor. The processor may be suspended just after the completion of the current bus cycle. On the other hand, the processor can be suspended just before the need of the system bus by the processor, because DMA controller is going to use the system bus, it will not use the processor.

The point where in the instruction cycle the processor may be suspended shown in the figure.

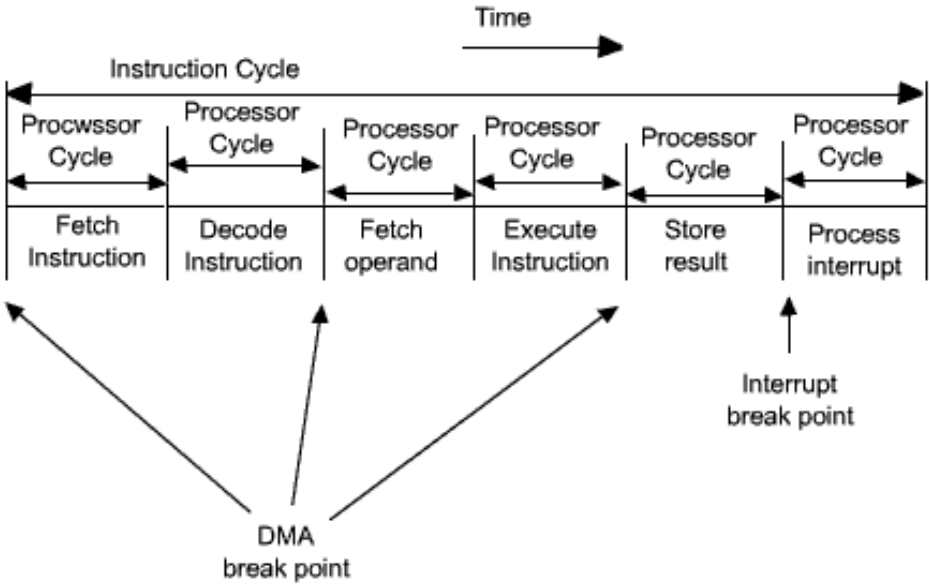


Fig : DMA break point

Direct Memory Access [DMA]

When the processor is suspended, then the DMA module transfer one word and return control to the processor.

Note that, this is not an interrupt, the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle.

During that time processor may perform some other task which does not involve the system bus. In the worst situation processor will wait for some time, till the DMA releases the bus.

The net effect is that the processor will go slow. But the net effect is the enhancement of performance, because for a multiple word I/O transfer, DMA is far more efficient than interrupt driven or programmed I/O.

Direct Memory Access [DMA]

The DMA mechanism can be configured in different ways. The most common amongst them are:

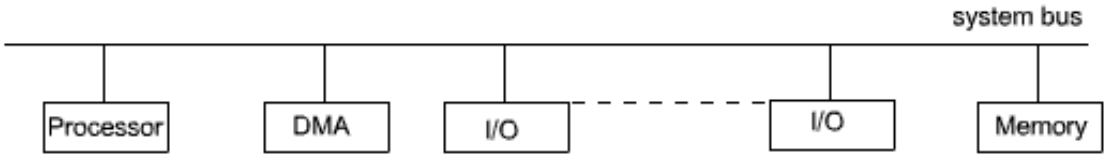
- **Single bus, detached DMA - I/O configuration.**
- **Single bus, Integrated DMA - I/O configuration.**
- **Using separate I/O bus.**

Single bus, detached DMA - I/O configuration

In this organization all modules share the same system bus.The DMA module here acts as a surrogate processor. This method uses programmed I/O to exchange data between memory and an I/O module through the DMA module.

For each transfer it uses the bus twice. The first one is when transferring the data between I/O and DMA and the second one is when transferring the data between DMA and memory. Since the bus is used twice while transferring data, so the bus will be suspended twice. The transfer consumes two bus cycle.

The interconnection organization is shown in the figure.



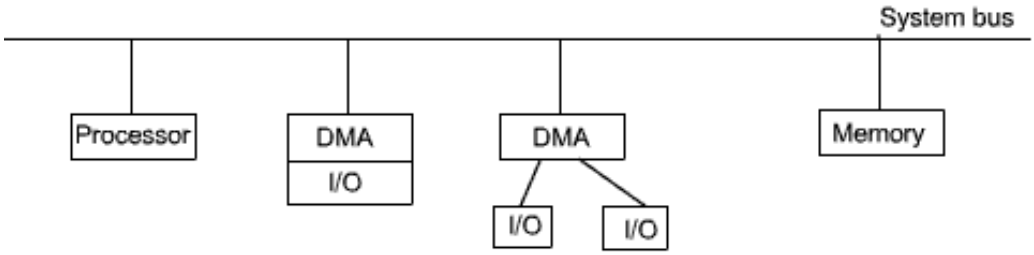
Single bus, Integrated DMA - I/O configuration

By integrating the DMA and I/O function the number of required bus cycle can be reduced. In this configuration, the DMA module and one or more I/O modules are integrated together in such a way that the system bus is not involved. In this case DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules.

The DMA module, processor and the memory module are connected through the system bus. In this configuration each transfer will use the system bus only once and so the processor is suspended only once.

The system bus is not involved when transferring data between DMA and I/O device, so processor is not suspended. Processor is suspended when data is transferred between DMA and memory.

The configuration is shown in the figure.

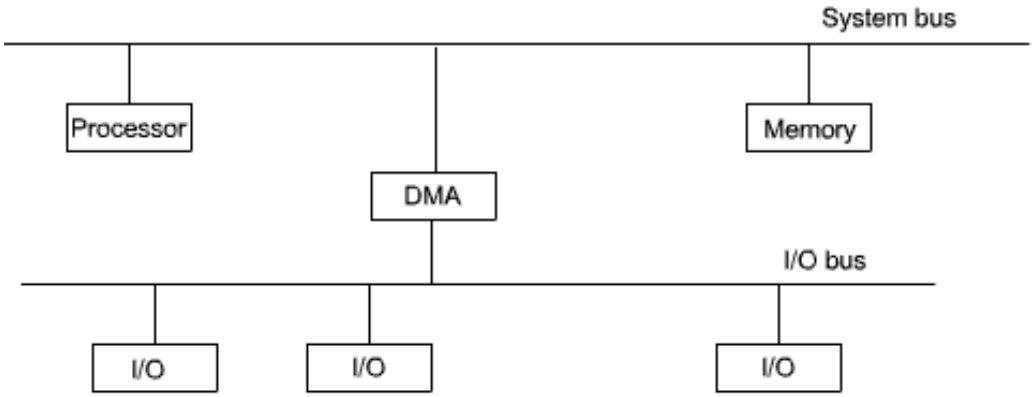


Using separate I/O bus

In this configuration the I/O modules are connected to the DMA through another I/O bus. In this case the DMA module is reduced to one.

Transfer of data between I/O module and DMA module is carried out through this I/O bus. In this transfer, system bus is not in use and so it is not needed to suspend the processor.

There is another transfer phase between DMA module and memory. In this time system bus is needed for transfer and processor will be suspended for one bus cycle. The configuration is shown in the figure.



Module 07 : Connecting I/O Devices

In this Module, we have two lectures, viz.

1. **I/O Buses**
2. **External Storage Devices**
3. **Disk Performance**

Click the proper link on the left side for the lectures

Buses

The *processor*, *main memory*, and *I/O devices* can be interconnected through common data communication lines which are termed as *common bus*.

The primary function of a common bus is to provide a communication path between the devices for the transfer of data. The bus includes the control lines needed to support interrupts and arbitration.

The bus lines used for transferring data may be grouped into three categories:

- data,
- address
- control lines.

A single $R/\overline{W}$ line is used to indicate Read or Write operation. When several sizes are possible like byte, word, or long word, control signals are required to indicate the size of data.

The bus control signal also carry timing information to specify the times at which the processor and the I/O devices may place data on the bus or receive data from the bus.

There are several schemes exist for handling the timing of data transfer over a bus. These can be broadly classified as

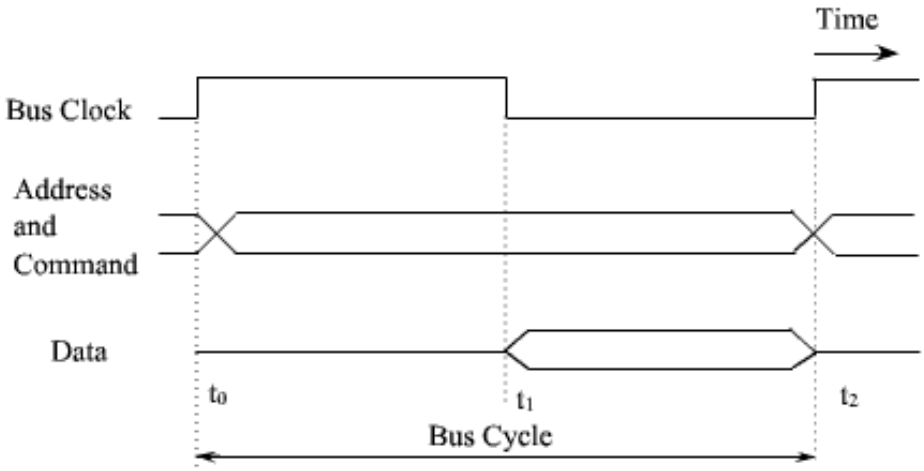
- Synchronous bus
- Asynchronous bus

Synchronous Bus :

In a synchronous bus, all the devices are synchronised by a common clock, so all devices derive timing information from a common clock line of the bus. A clock pulse on this common clock line defines equal time intervals.

In the simplest form of a synchronous bus, each of these clock pulse constitutes a bus cycle during which one data transfer can take place.

The timing of an input transfer on a synchronous bus is shown in the figure (Next Page..).



Timing of an input transfer on a synchronous bus

Let us consider the sequence of events during an input (read) operation.

At time t_0, the master places the device address on the address lines and sends an appropriate command (read in case of input) on the command lines.

In any data transfer operation, one device plays the role of a master, which initiates data transfer by issuing read or write commands on the bus.

Normally, the processor acts as the master, but other device with DMA capability may also becomes bus master. The device addressed by the master is referred to as a slave or target device.

The command also indicates the length of the operand to be read, if necessary.

The clock pulse width, t_1 - t_0, must be longer than the maximum propagation delay between two devices connected to the bus.

After decoding the information on address and control lines by slave, the slave device of that particular address responds at time t_1. The addressed slave device places the required input data on the data line at time t_1.

At the end of the clock cycle, at time t_2, the master strobes the data on the data lines into its input buffer. The period t_2- t_1 must be greater than the maximum propagation delay on the bus plus the set up time of the input buffer register of the master.

A similar procedure is followed for an output operation. The master places the output data on the data lines when it transmits the address and command information. At time t_2, the addressed device strobe the data lines and load the data into its data buffer.

Multiple Cycle Transfer

The simple design of device interface by synchronous bus has some limitations.

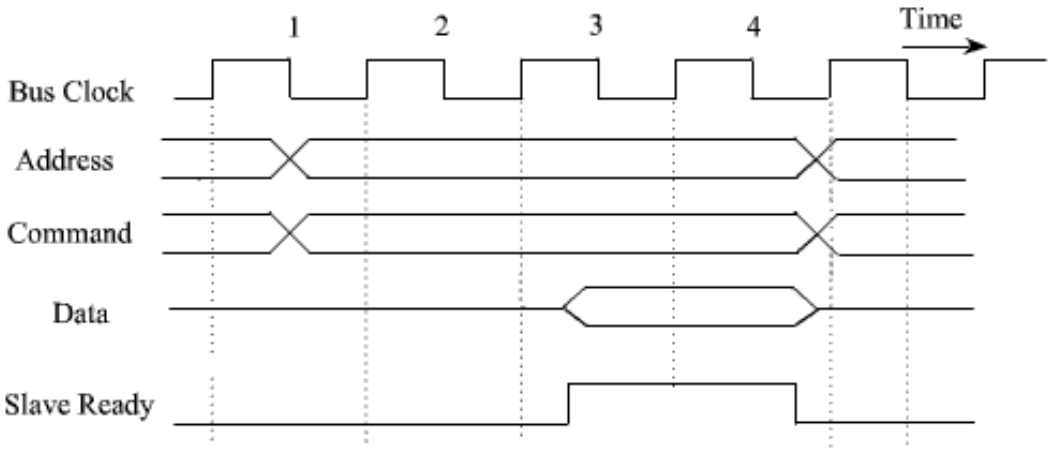
- A transfer has to be completed within one clock cycle. The clock period, must be long enough to accommodate the slowest device to interface. This forces all devices to operate at the speed of slowest device.
- The processor or the master has no way to determine whether the addressed device has actually responded. It simply assumes that, the output data have been received by the device or the input data are available on the data lines.

To solve these problems, most buses incorporate control signals that represent a response from the device. These signals inform the master that the target device has recognized its address and it is ready to participate in the data transfer operation.

They also adjust the duration of the data transfer period to suit the needs of the participating devices.

A high frequency clock pulse is used so that a complete data transfer operation span over several clock cycles. The numbers of clock cycles involved can vary from device to device

An instance of this scheme is shown in the figure.



An input transfer using multiple clock cycle

In *clock cycle 1*, the master sends address and command on the bus requesting a read operation.

The target device responded at *clock cycle 3* by indicating that it is ready to participate in the data transfer by making the slave ready signal high.

Then the target device places the data on the data line.

The target device is a slower device and it needs two clock cycle to transfer the information. After two clock cycle, that is at *clock cycle 5*, it pulls down the slave ready signal down.

When the slave ready signal goes down, the master strobes the data from the data bus into its input buffer.

If the addressed device does not respond at all, the master waits for some predefined maximum number of clock cycles, then aborts the operation.

Asynchronous Bus

In asynchronous mode of transfer, a *handshake signal* is used between *master* and *slave*.

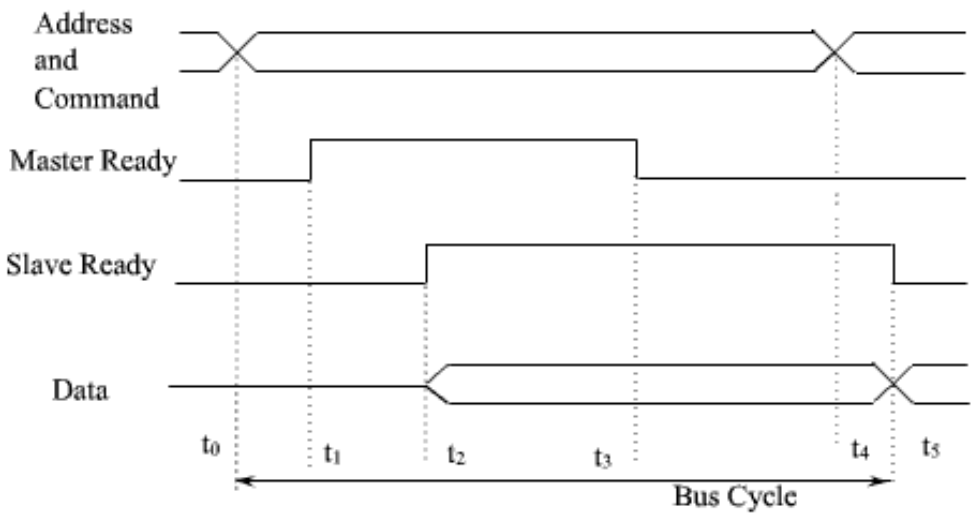
In *asynchronous bus*, there is no common clock, and the common clock signal is replaced by two timing control signals: *master-ready* and *slave-ready*.

Master-ready signal is assured by the master to indicate that it is ready for a transaction, and slave-ready signal is a response from the slave.

The handshaking protocol proceeds as follows:

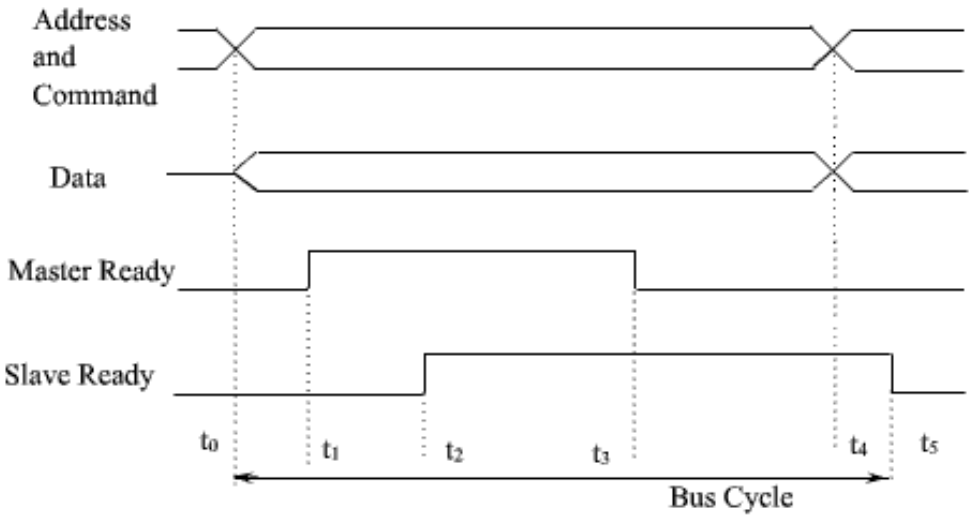
- The master places the address and command information on the bus. Then it indicates to all devices that it has done so by activating the master-ready signal.
- This causes all devices on the bus to decode the address.
- The selected target device performs the required operation and inform the processor (or master) by activating the slave-ready line.
- The master waits for slave-ready to become asserted before it remove its signals from the bus.
- In case of a read operation, it also strobes the data into its input buffer.

The timing of an input data transfer using the handshake scheme is shown in the figure.



Handshake control of data transfer during an input operation

The timing of an output operation using handshaking scheme is shown in the figure.



Handshake control of data transfer during an output operation

External Memory

Main memory is taking an important role in the working of computer. We have seen that computer works on *Von-Neuman* stored program principle. We have to keep the information in main memory and CPU access the information from main memory.

The main memory is made up of semiconductor device and by nature it is volatile. For permanent storage of information we need some non volatile memory. The memory devices need to store information permanently are termed as *external memory*. While working, the information will be transferred from external memory to main memory.

The devices need to store information permanently are either magnetic or optical devices.

Magnetic Devices:

- Magnetic disk (Hard disk)
- Floopy disk
- Magnetic tape

Optical Devices:

- CD- ROM
- CD-Recordable(CD –R)
- CD-R/W
- DVD

Magnetic Disk

A disk is a *circular platter* constructed of metal or of plastic coated with a magnetizable material.

Data are recorded on and later retrieved from the disk via a conducting coil named the *head*.

During a read or write operation, the head is stationary while the platter rotates beneath it.

The *write mechanism* is based on the fact that electricity flowing through a coil produces a magnetic field.Pulses are sent to the head, and magnetic patterns are recorded on the surface below. The pattern depends on the *positive* or *negative* currents. The direction of current depends on the information stored , i.e., positive current depends on the information '1' and negative current for information '0'.

The *read mechanism* is based on the fact that a magnetic field moving relative to a coil produces on electric current in the coil. When the surface of the disk passes under the head, it generates a current of the same polarity as the one already recorded.

Read/ Write head detail is shown in the figure:

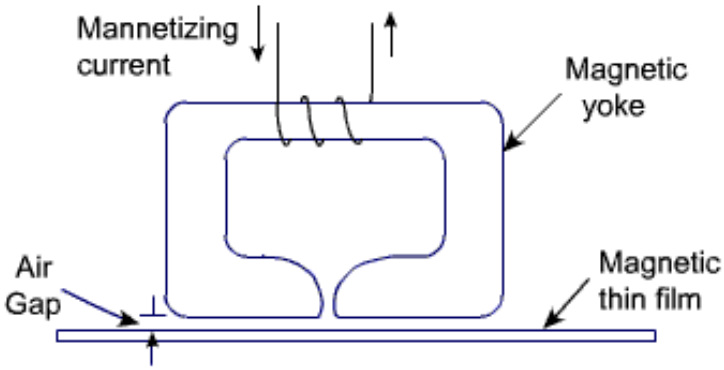


Figure : Read/Write head detail

The head is a relatively small device capable of *reading from* or *writing to* a portion of the platter rotating beneath it.

The data on the disk are organized in a concentric set of rings, called *track*. Each track has the same width as the head. Adjacent tracks are separated by gaps. This prevents error due to misalignment of the head or interference of magnetic fields.

For simplifying the control circuitry, the same number of bits are stored on each track. Thus the density, in bits per linear inch, increases in moving from the outermost track to the innermost track.

Data are transferred to and from the disk in blocks. Usually, the block is smaller than the capacity of the track. Accordingly, data are stored in block-size regions known as sector.

A typical disk layout is shown in the figure:

To avoid, imposition unreasonable precision requirements on the system, adjacent tracks (sectors) are separated by *intratrack* (intersector) gaps.

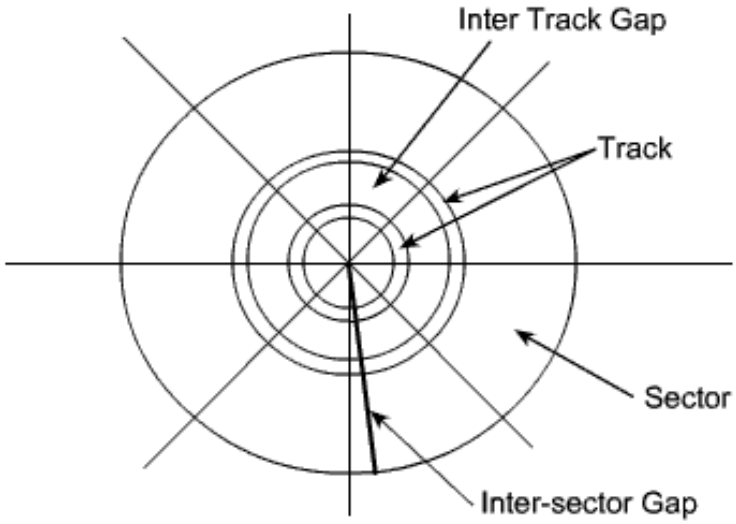
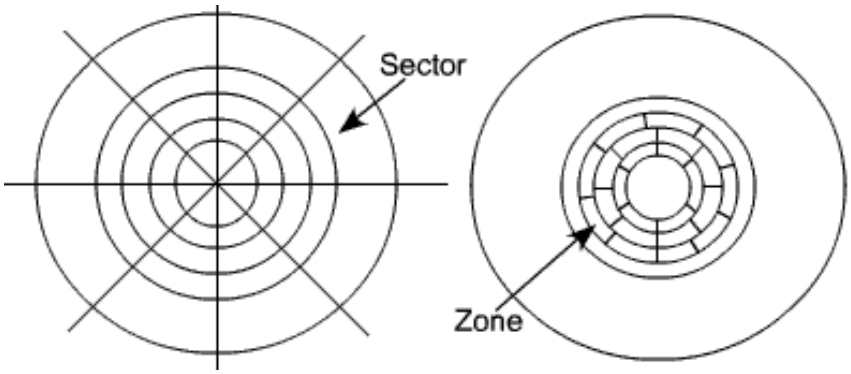


Figure : Disk Data Layout

Some means are needed to locate sector positions within a track. Clearly there must be some starting points on the track and a way of identifying the *start* and *end* of each sector. These requirements are handled by means of a control data recorded on the disk. Thus, the disk is formatted with some *extra data* used only by the disk drive and not accessible to the user.

Since data density in the outermost track is less and data density is more in inner tracks so there are wastage of space on outer tracks.

To increase the capacity, the concept of *zone* is used instead of *sectors*. Each track is divided in zone of equal length and fix amount of data is stored in each zone. So the number of zones are less in innermost track and number of zones are more in the outermost track. Therefore, more number of bits are stored in outermost track. The disk capacity is increasing due to the use of zone, but the complexity of control circuitry is also more.



Physical characteristics of disk

The head may be either fixed or movable with respect to the radial direction of the platter.

In a fixed-head disk, there is one read-write head per track. All of the heads are mounted on a rigid arm that extends across all tracks.

In a movable-head disk, there is only one read-write head. Again the head is mounted on an arm. Because the head must be able to be positioned above any track, the arm can be extended or retracted for this purpose.

The disk itself is mounted in a disk drive, which consists of the arm, the shaft that rotates the disk, and the electronics circuitry needed for input and output the binary data and to control the mechanism.

A non removable disk is permanently mounted on the disk drive. A removable disk can be removed and replaced with another disk.

For most disks, the magnetizable coating is applied to both sides of the platters, which is then referred to as double sided. If the magnetizable coating is applied to one side only, then it is termed as *single sided disk*.

Some disk drives accommodate multiple platters stacked vertically above one another. Multiple arms are provided for read write head. The platters come as a unit known as a disk pack.

The physical organization of disk is shown in the figure:

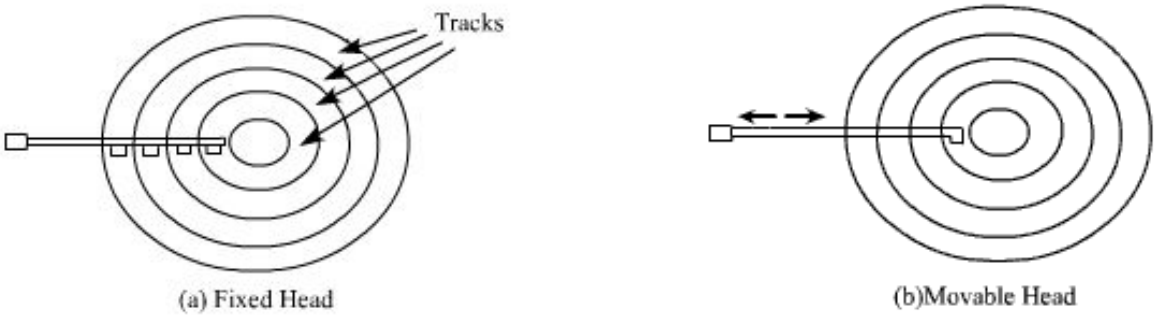


Figure: Fixed and Movable head disk

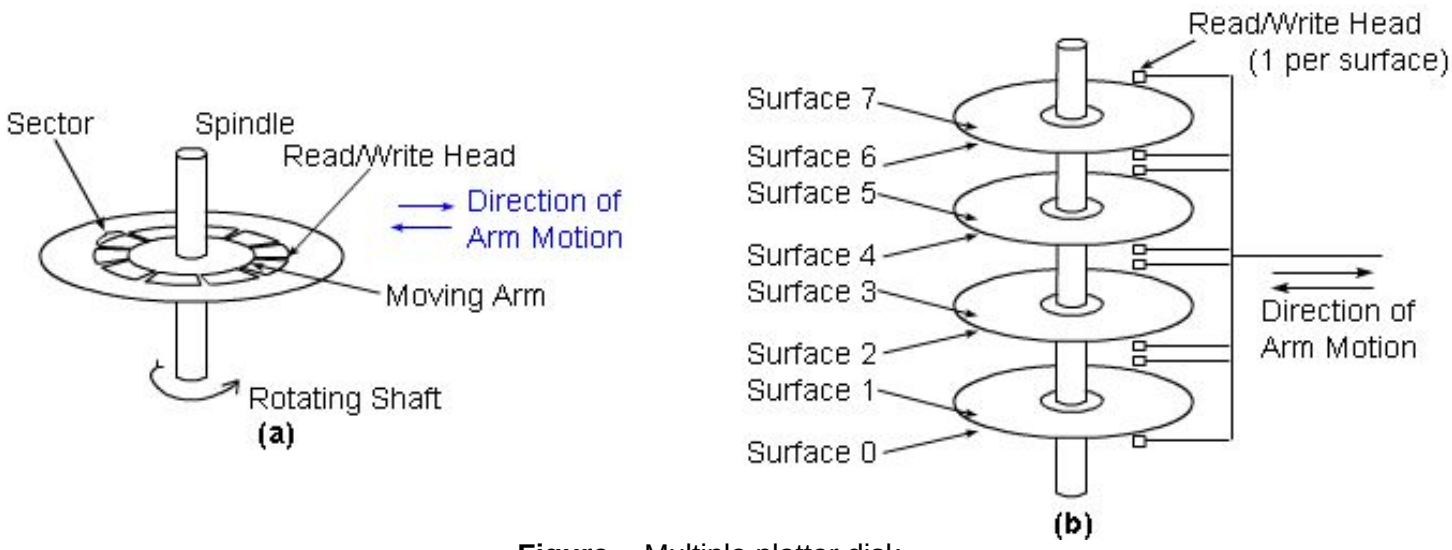
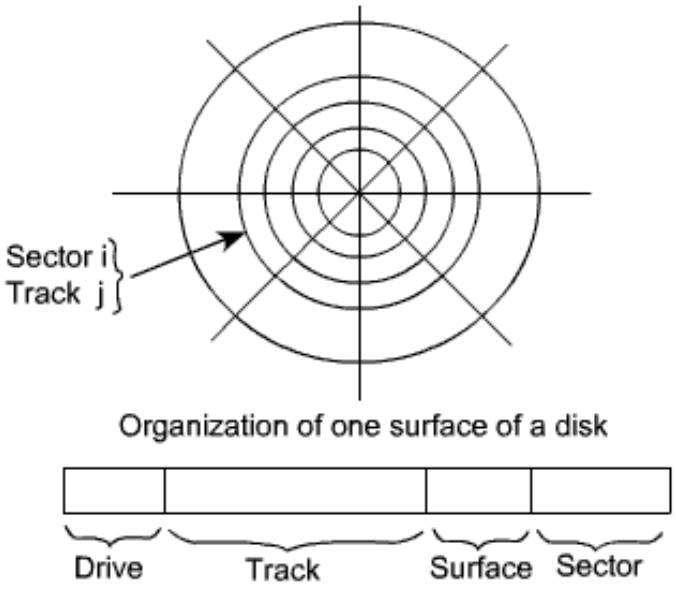


Figure: Multiple platter disk

Organization and accessing of data on a disk

The organization of data on a disk is shown in the figure below.



Organization of one surface of a disk

Each surface is divided into concentric tracks and each track is divided into sectors. The set of corresponding tracks on all surfaces of a stack of disks form a logical cylinder. Data bits are stored serially on each track.

Data on disks are addressed by specifying the surface number, the track number, and the sector number.

In most disk systems, read and write operations always start at sector boundaries. If the number of words to be written is smaller than that required to fill a sector, the disk controller repeats the last bit of data for the remaining of the sector.

During read and write operation, it is required to specify the starting address of the sector from where the operation will start, that is the *read/write head* must positioned to the correct track, sector and surface. Therefore the address of the disk contains *track no*., *sector no*., and *surface no*. If more than one drive is present, then drive number must also be specified.

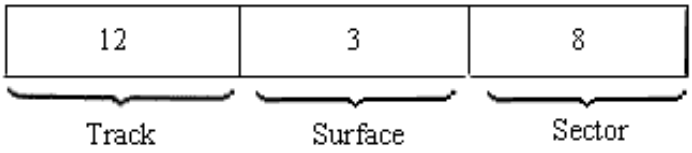
The format of the disk address word is shown in the figure. It contains the *drive no*, *track no*., *surface no*. and *sector no*.

The *read/write head* will first positioned to the correct track. In case of *fixed head system*, the correct head is selected by taking the *track no.* from the address. In case of *movable head system*, the head is moved so that it is positioned at the correct track.

By the surface no, it selects the correct surface.

To get the correct sector below the *read/write head*, the disk is rotated and bring the correct sector with the help of sector number. Once the correct sector, track and surface is decided, the *read/write operation* starts next.

Suppose that the disk system has 8 data recording surfaces with 4096 track per surface. Tracks are divided into 256 sectors. Then the format of disk address word is:



Suppose each sector of a track contains 512 bytes of disk recorded serially, then the total capacity of the disk is:

$$8 \times 4096 \times 256 \times 512 = 2^3 \times 2^{12} \times 2^8 \times 2^9$$
$$= 4 \text{ Gigabyte (GB)}$$

For moving head system, there are two components involved in the time delay between receiving an address and the beginning of the actual data transfer.

Seek Time:

Seek time is the time required to move the read/write head to the proper track. This depends on the initial position of the head relative to the track specified in the address.

Rotational Delay:

Rotational delay, also called the *latency time* is the amount of time that elapses after the head is positioned over the correct track until the starting position of the addressed sector comes under the Read/write head.

Disk Operation

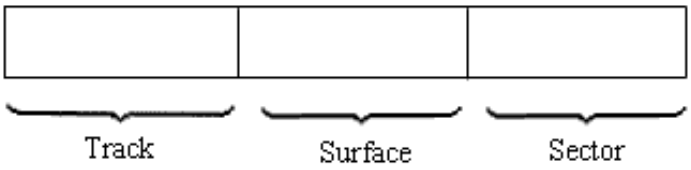
Communication between a disk and the main memory is done through DMA. The following information must be exchanged between the processor and the disk controller in order to specify a transfer.

Main memory address :
The address of the first main memory location of the block of words involved in the transfer.

Disk address :
The location of the sector containing the beginning of the the desired block of words.

Word count :
The number of words in the block to be transferred.

The disk address format is:



The word count may corresponds to fewer or more bytes than that are contained in a sector. When the data block is longer than a track:

The disk address register is incremented as successive sectors are read or written. When one track is completed then the surface count is incremented by 1.

Thus, long data blocks are laid out on cylinder surfaces as opposed to being laid out on successive tracks of a single disk surface.

This is efficient for moving head systems, because successive sector areas of data storage on the disk can be accessed by electrically switching from one *Read/Write head* to the next rather than by mechanically moving the arm from track to track.

The *track-to-track* movement is required only at *cylinder-to-cylinder* boundaries.

Disk Performance Parameter

When a disk drive is operating, the disk is rotating at constant speed.

To *read* or *write*, the head must be positioned at the desired tack and at the beginning of the desired *sector* on the *track*.

Track selection involves moving the head in a *movable-head system* or electronically selecting one head on a fixed head system.

On a *movable-head system*, the time taken to position the head at the track is known a *seek time*.

Once the track is selected, the disk controller waits until the appropriate sector rotates to line up with the head. The time it takes to reach the beginning of the desired sector is known as *rotational delay* or *rotational latency*.

The sum of the *seek time*, (for movable head system) and the *rotational delay* is termed as ***access time*** of the disk, the time it takes to get into appropriate position (track & sector) to read or write.

Once the head is in position, the read or write operation is then performed as the sector moves under the head, and the data transfer takes place.

Seek Time:

Seek time is the time required to move the disk arm to the required track. The seek time is approximated as

$$T_s = m \times n + s$$

where

T_s = estimated seek time

n = number of tracks traversed

m = constant that depends on the disk drive

s = startup time

Rotational Delay:

Disk drive generally rotates at 3600 *rpm*, i.e., to make one revolution it takes around 16.7 *ms*. Thus on the average, the rotational delay will be 8.3 *ms*.

Transfer Time:

The transfer time to or from the disk depends on the rotational speed of the disk and it is estimated as

$$T = \frac{b}{rN}$$

where

T = Transfer time
b = Number of bytes to be transferred.
N = Numbers of bytes on a track
r = Rotational speed, in revolution per second.

Thus,

the total ***average access time*** can be expressed as

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

where T_s is the average seek time.

Issues with disks:

Disks are *potential bottleneck* for system performances and storage system reliability.

The *disk access time* is relatively higher than the time required to access data from main memory and performs CPU operation. Also the disk drive contains some mechanical parts and it involves mechanical movement, so the failure rate is also high.

The disk performance has been improving continuously, microprocessor performance has improved much more rapidly.

A *disk array* is an arrangement of several disk, organized to increase performance and improve reliability of the resulting storage system. Performance is increased through *data striping*. Reliability is improved through *redundancy*.

Disk arrays that implement a combination of data striping and redundancy are called ***Redundant Arrays of Independent Disks*** (RAID).

Data Striping

In data striping, he data is segmented in equal-size partitions distributed over multiple disks. The size of the partition is called the *striping unit*.

The partitions are usually distributed using a *round-robin algorithm*:

if the disk array consists of D *disks*, then partition i is written in to disk ($i \bmod D$).

Consider a striping unit equal to a disk block. In this case, I/O requests of the size of a disk block are processed by one disk in the array.

If many I/O requests of the size of a disk block are made, and the requested blocks reside on different disks, we can process all requests in parallel and thus reduce the average response time of an I/O request.

Since the striping unit are distributed over several disks in the disk array in round robin fashion, large I/O requests of the size of many continuous blocks involve all disks. We can process the request by all disks in parallel and thus increase the transfer rate.

Redundancy

While having more disks increases storage system performance, it also lower overall storage system reliability, because the probability of failure of a disk in disk array is increasing.

Reliability of a disk array can be increased by storing redundant information. If a disk fails, the redundant information is used to reconstruct the data on the failed disk.

One design issue involves here - where to store the redundant information. There are two choices-either store the redundant information on a same number of check disks, or distribute the redundant information uniformly over all disk.

In a RAID system, the disk array is partitioned into reliability group, where a reliability group consists of a set of data disks and a set of check disks. A common redundancy scheme is applied to each group.

RAID levels

RAID Level 0 : Nonredundant

A **RAID level 0** system is not a true member of the RAID family, because it does not include redundancy, that is, no redundant information is maintained.

It uses data striping to increase the I/O performance.

For **RAID 0**, the user and system data are distributed across all of the disk in the array, i.e. data are striped across the available disk.

If two different I/O requests are there for two different data block, there is a good probability that the requested blocks are in different disks. Thus, the two requests can be issued in parallel, reducing the I/O waiting time.

RAID level 0 is a low cost solution, but the reliability is a problem since there is no redundant information to retrieve in case of disk failure.

RAID level 0 has the best write performance of all RAID levels, because there is no need of updation of redundant information.

RAID Level 1 : Mirrored

RAID level 1 is the most expensive solution to achieve the redundancy. In this system, two identical copies of the data on two different disks are maintained. This type of redundancy is called mirroring.

Data striping is used here similar to **RAID 0**.

Every write of a disk block involves two write due to the mirror image of the disk blocks.

These writes may not be performed simultaneously, since a global system failure may occur while writing the blocks and then leave both copies in an inconsistent state. Therefore, write a block on a disk first and then write the other copy on the mirror disk.

A read of a block can be scheduled to the disk that has the smaller access time. Since we are maintaining the full redundant information, the disk for mirror copy may be less costly one to reduce the overall cost.

RAID Level 2 :

RAID levels 2 and **3** make use of a parallel access technique where all member disks participate in the execution of every I/O requests.

Data striping is used in **RAID levels 2** and **3**, but the size of strips are very small, often a small as a single byte or word.

With **RAID 2**, an error-correcting code is calculated across corresponding bits on each data disk, and the bits of the cods are stored in the corresponding bit positions on multiple parity disks.

RAID 2 requires fewer disks than **RAID 1**. The number of redundant disks is proportional to the log of the number of data disks. For error-correcting, it uses Hamming code.

On a single read, all disks are simultaneously accessed. The requested data and the associated error correcting code are delivered to the array controller. If there is a single bit error, the controller can recognize and correct the error instantly, so that read access time is not slowed down.

On a single write, all data disks and parity disks must be accessed for the write operation.

RAID level 3 :

RAID level 3 is organized in a similar fashion to **RAID level 2**. The difference is that **RAID 3** requires only a single redundant disk.

RAID 3 allows parallel access, with data distributed in small strips.

Instead of an error correcting code, a simple parity bit is computed for the set of individual bits in the same position on all of the data disks.

In this event of drive failure, the parity drive is accessed and data is reconstructed from the remaining drives. Once the failed drive is replaced, the missing data can be restored on the new drive.

RAID level 4 :

RAID levels 4 through **6** make use of an independent access technique, where each member disk operates independently, so that separate I/O request can be satisfied in parallel.

Data stripings are used in this scheme also, but the data strips are relatively large for **RAID levels 4** through **6**.

With **RAID 4**, a bit-by-bit parity strip is calculated across corresponding strips on each data disks, and the parity bits are stored in the corresponding strip on the parity disk.

RAID 4 involves a write penalty when an I/O write request of small size is occurred. Each time a write occurs, update is required both in user data and the corresponding parity bits.

RAID level 5 :

RAID level 5 is similar to **RAID 4**, only the difference is that **RAID 5** distributes the parity strips across all disks.

The distribution of parity strips across all drives avoids the potential I/O bottleneck.

RAID level 6 :

In **RAID level 6**, two different parity calculations are carried out and stored in separate blocks on different disks.

The advantage of **RAID 6** is that it has got a high data availability, because the data can be regenerated even if two disk containing user data fails. It is possible due to the use of Reed-Solomon code for parity calculations.

In **RAID 6**, there is a write penalty, because each write affects two parity blocks.

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

RISC Print this page

<< Previous | First | Last | Next >>

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.

<< Previous | First | Last | Next >>

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

[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.

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. . . . . .

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

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 ← 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.

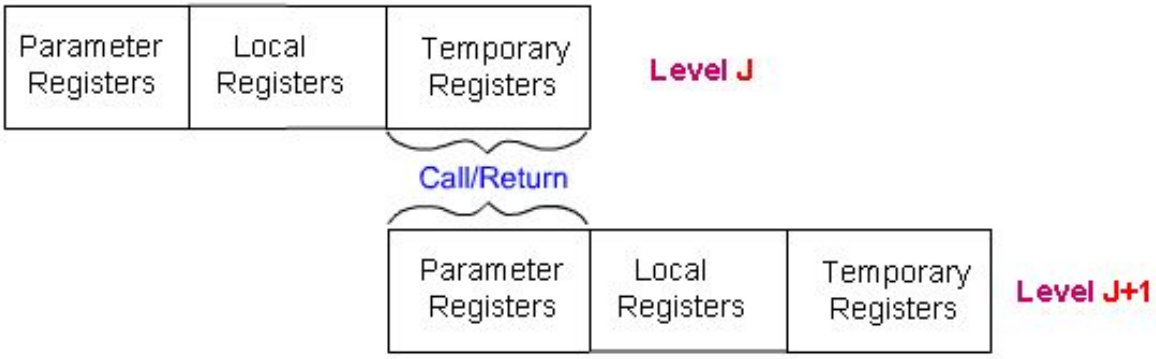
From the studies it is observed that a typical procedure employs only a few passed parameters and local variables. Also the depth of procedure activation remains within a relatively narrow range.

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.

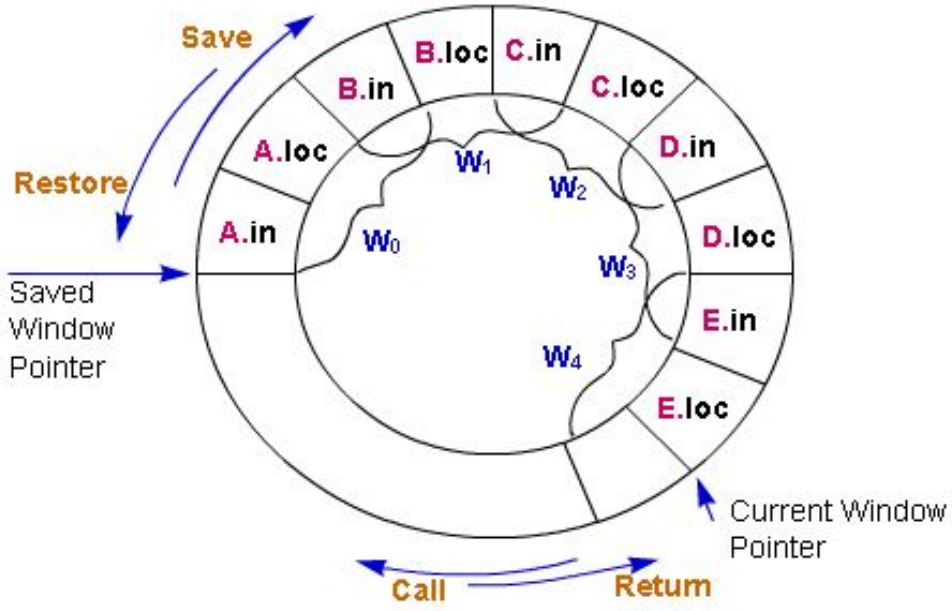
To handle any possible pattern of calls and returns, the number of register windows would have to be unbounded. But we have a limited number of registers, it is not possible to provide unlimited amount of registers.

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..)

Design Issues of RISC



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 survery, 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 is 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.

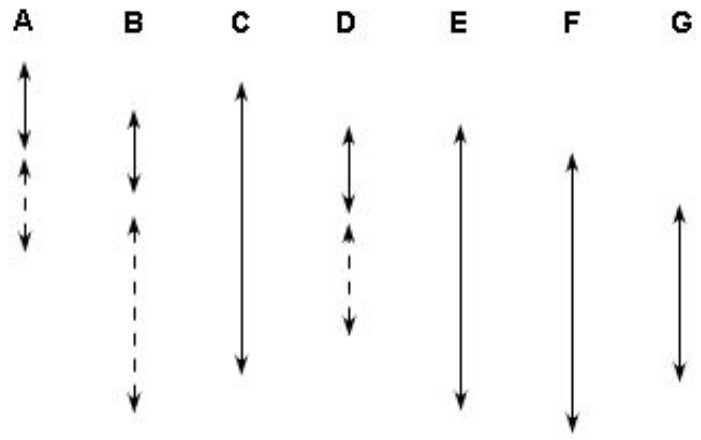
This graph coloring problem is mapped to the register optimization problem of the compiler in the following way:

- 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.

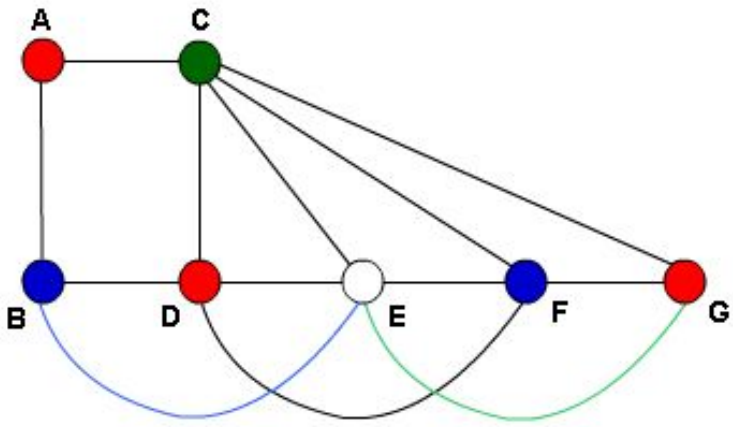
Following figures shows a simple example of a process. (next page..)

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.

Design Issues of RISC



(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 charistics 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 golbal 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

Pipelining

Print this page

<< Previous | First | Last | Next >>

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.

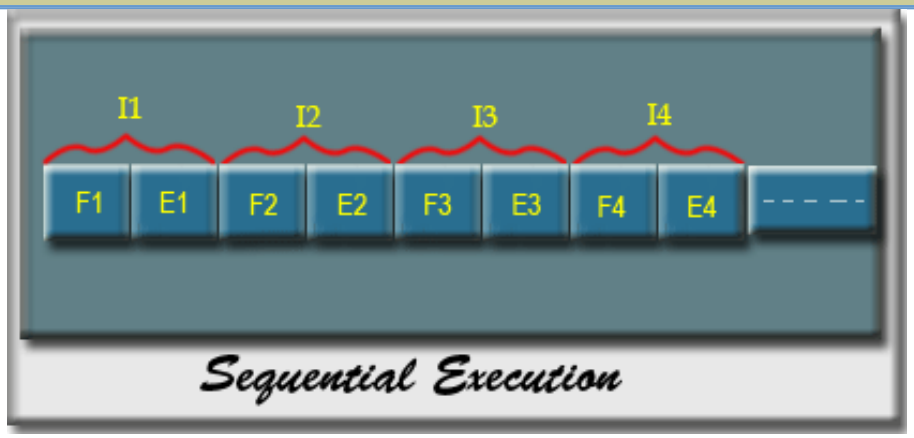
As we know that there are two distinct phases of instruction execution: one is instruction fetch and the other one is instruction execution. Therefore, the processor executes a program by fetching and executing instructions, one after another.

Let 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.

>> Previous | First | Last | Next >>

Pipelining Print this page

<< Previous | First | Last | Next >>



>> Previous | First | Last | Next >>

Pipelining Print this page

<< Previous | First | Last | Next >>

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.

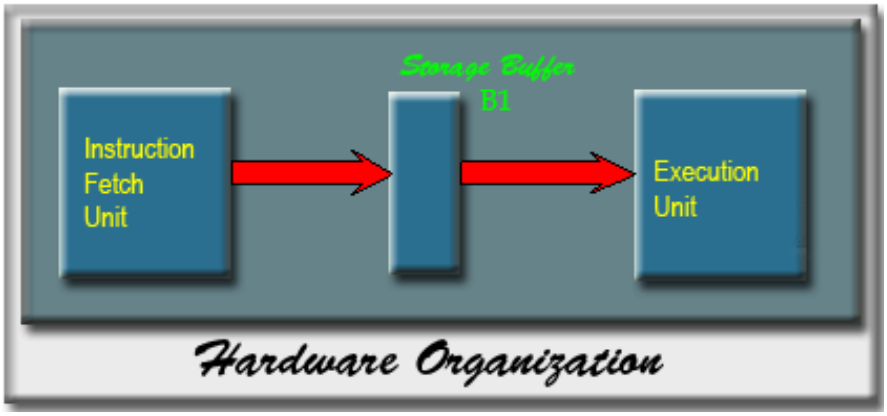
The operation of the computer proceeds as follows:

- 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_1 which is already fetched and available in the buffer B_1 (step E_1).

<< Previous | First | Last | Next >>

- 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.

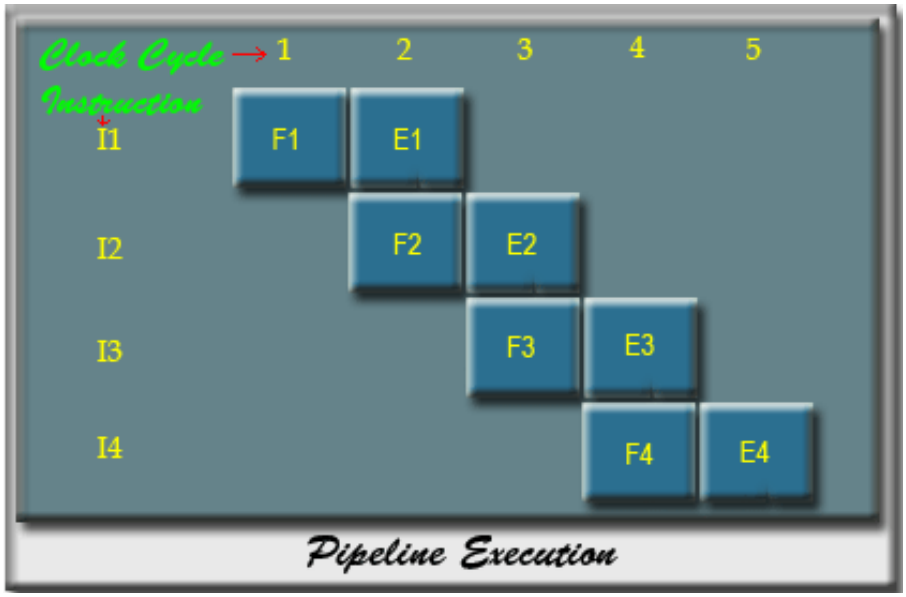


Another picture is shown on the next slide for instruction piplining.

Pipelining

Print this page

<< Previous | First | Last | Next >>



>> Previous | First | Last | Next >>

Pipelining Print this page

<< Previous | First | Last | Next >>

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.

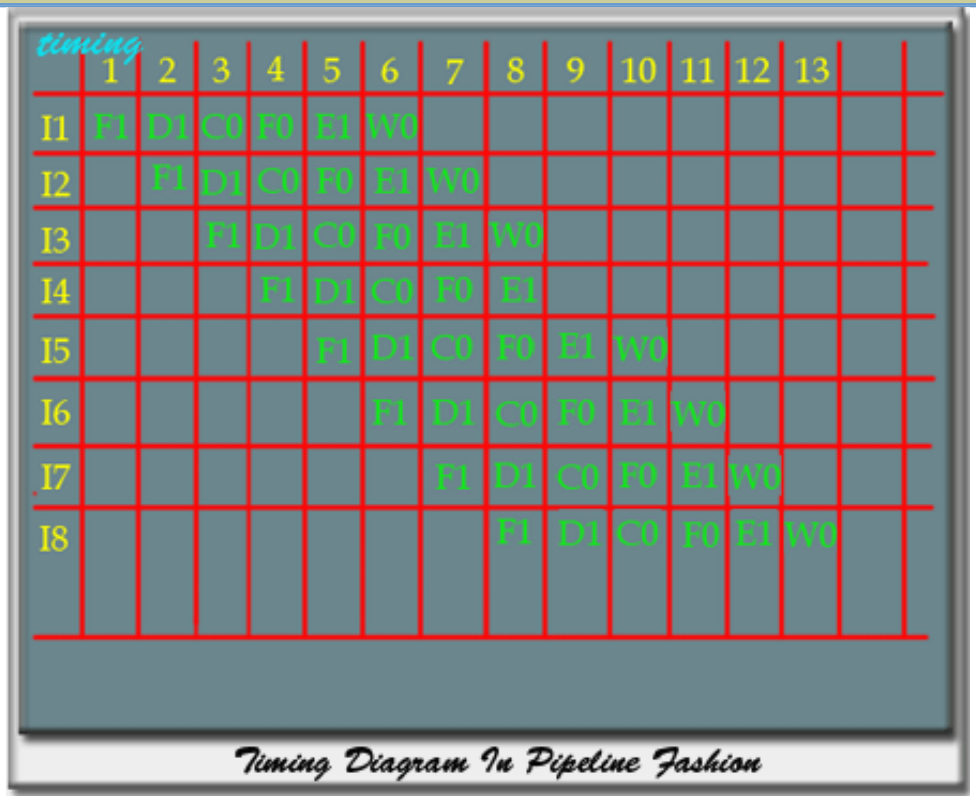
There will be six different stages for these six subtasks. For the sake of simplicity, let us assume the equal duration to perform all the subtasks. It the six stages are not of equal duration, there will be some waiting involved at various pipeline stages.

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

<< Previous | First | Last | Next >>

Pipelining Print this page

<< Previous | First | Last | Next >>



Timing Diagram In Pipeline Fashion

<< 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.

Pipeline Performance : Print this page

<< 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

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

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 T_k to execute all *n* instructions is

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

<< Previous | First | Last | Next >>

Pipeline Performance : Print this page

<< Previous | First | Last | Next >>

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 nt 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

$$S_k = k,$$

>> Previous | First | Last | Next >>

Pipeline Performance : Print this page

<< Previous | First | Last | Next >>

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.

<< Previous | First | Last | 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)

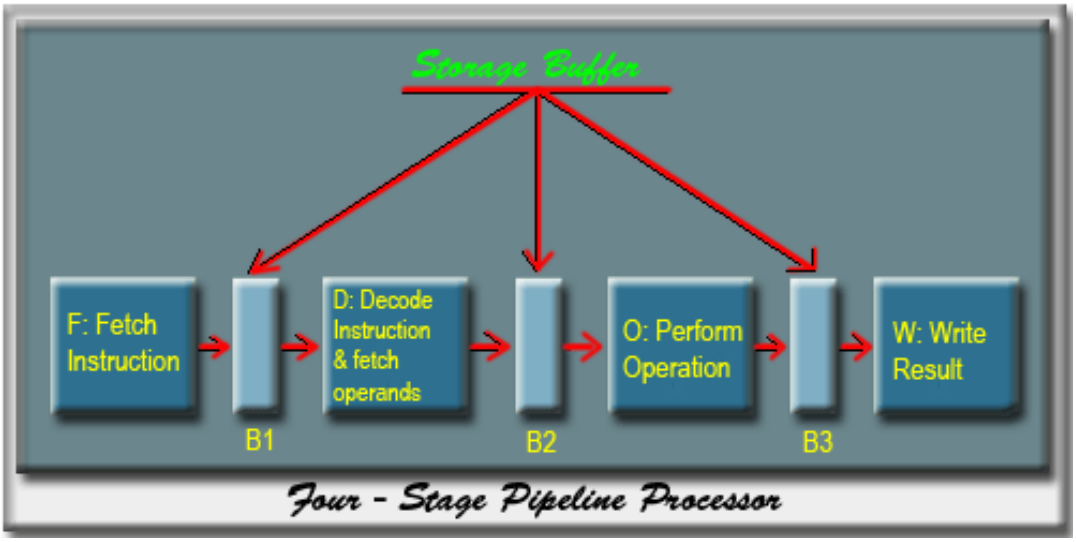
O: Operate, perform the operation

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.

Pipeline Performance : Print this page

<< Previous | First | Last | Next >>



>> Previous | First | Last | Next >>

Pipeline Performance : Print this page

<< Previous | First | Last | 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

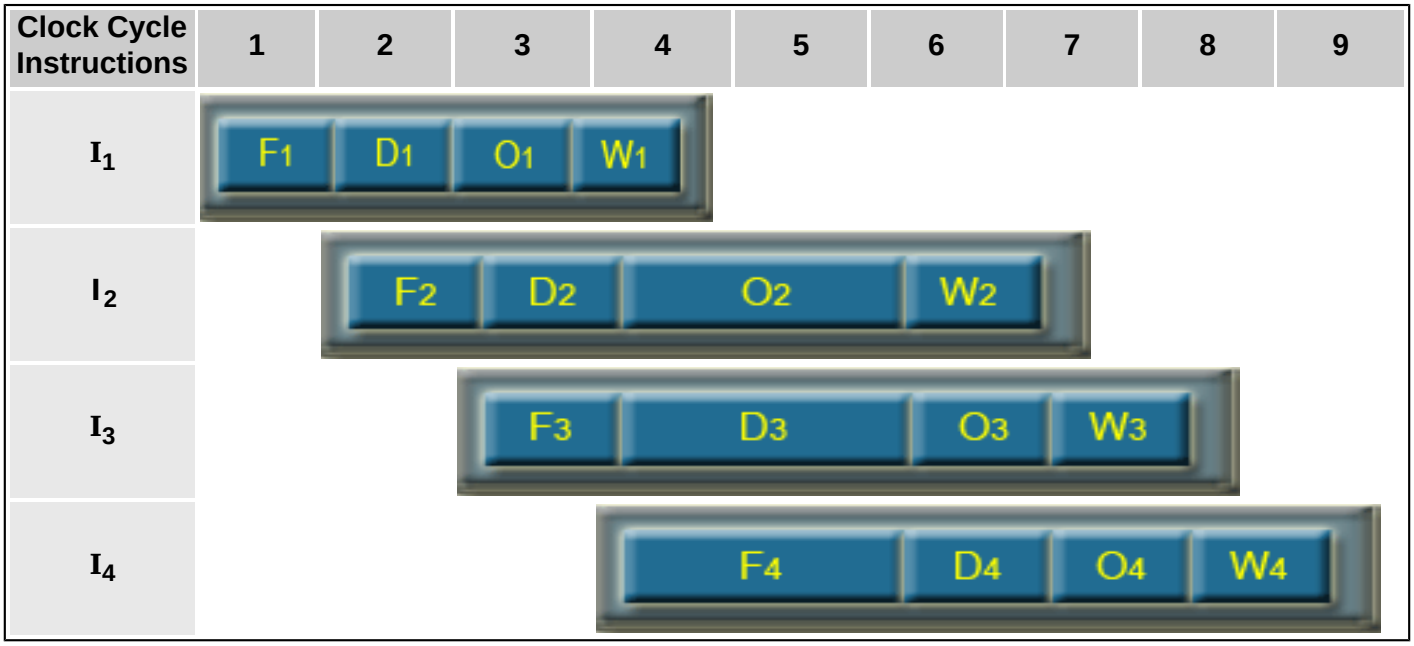
>> Previous | First | Last | Next >>

Pipeline Performance : Print this page

<< Previous | First | Last | Next >>

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.



<< Previous | First | Last | Next >>

Pipeline Performance : Print this page

<< Previous | First | Last | Next >>

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.

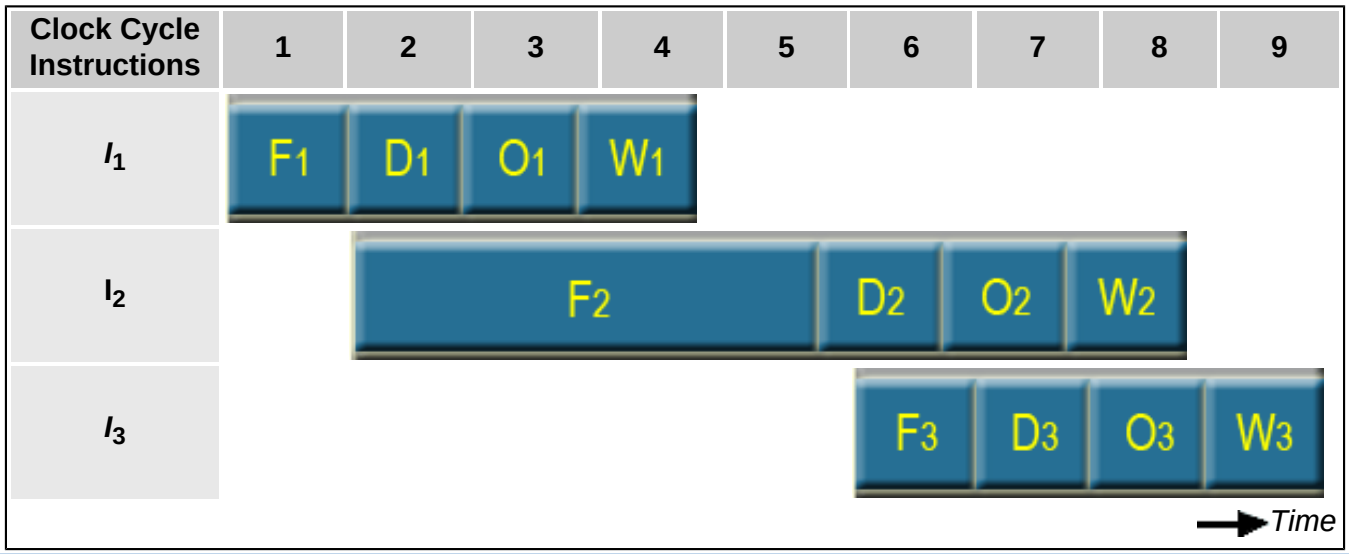
Whenever the pipeline stalled, some degradation in performance occurs.

>> Previous | First | Last | Next >>

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.



Pipeline Performance : Print this page

<< Previous | First | Last | Next >>

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.

>> Previous | First | Last | Next >>

Pipeline Performance : Print this page

<< Previous | First | Last | Next >>

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.

>> Previous | First | Last | Next >>

Pipeline Performance : Print this page

<< Previous | First | Last | Next >>

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	

<< Previous | First | Last | Next >>

Pipeline Performance : Print this page

<< Previous | First | Last | Next >>

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.

<< Previous | First | Last | Next >>

Pipeline Performance : Print this page

<< Previous | First | Last | Next >>

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.

>> Previous | First | Last | Next >>

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 / 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

<< 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 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.

<< 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 wh ich 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 >>

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

o

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 contitional branch instructions.

The later two approaches are dynamic- they depend on the execution history.

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 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 [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 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 [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:

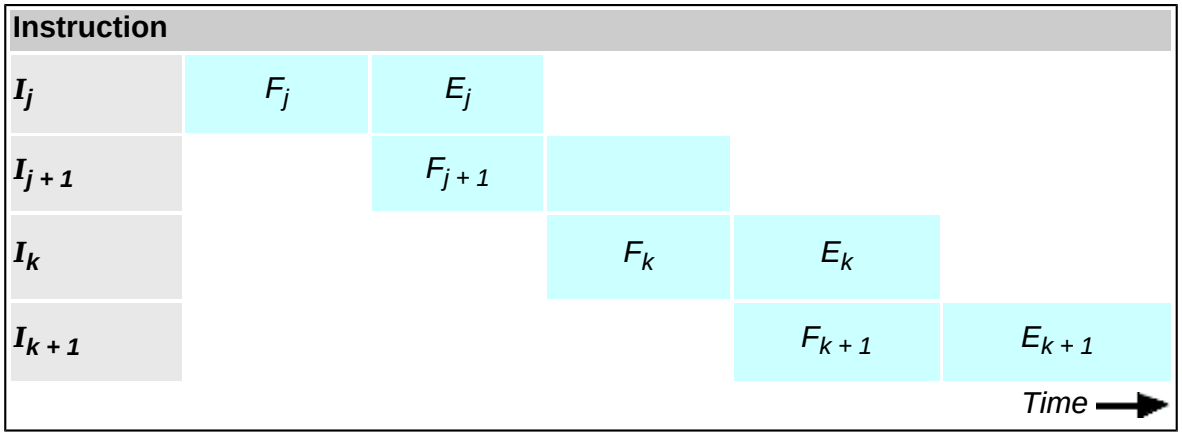
- The address of the branch instruction.
- Some member of history bits that record the state of use of that instruction.
- Information about the target instruction, it may be the address of the target instruction, or may be the target instruction itself.

<< 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 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.

>> Previous | First | Last | Next >>

Branching : Print this page

<< Previous | First | Last | Next >>

For example consider the 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

>> 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

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

Reordered instructions for program loop

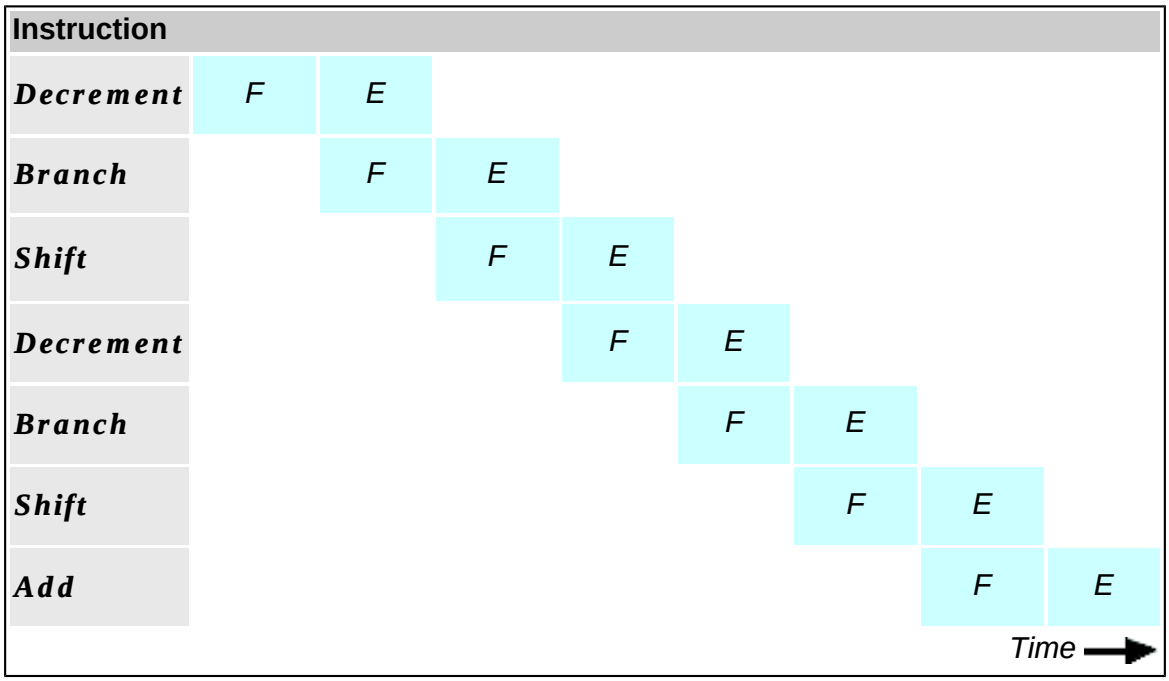
<< Previous | First | Last | Next >>

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.

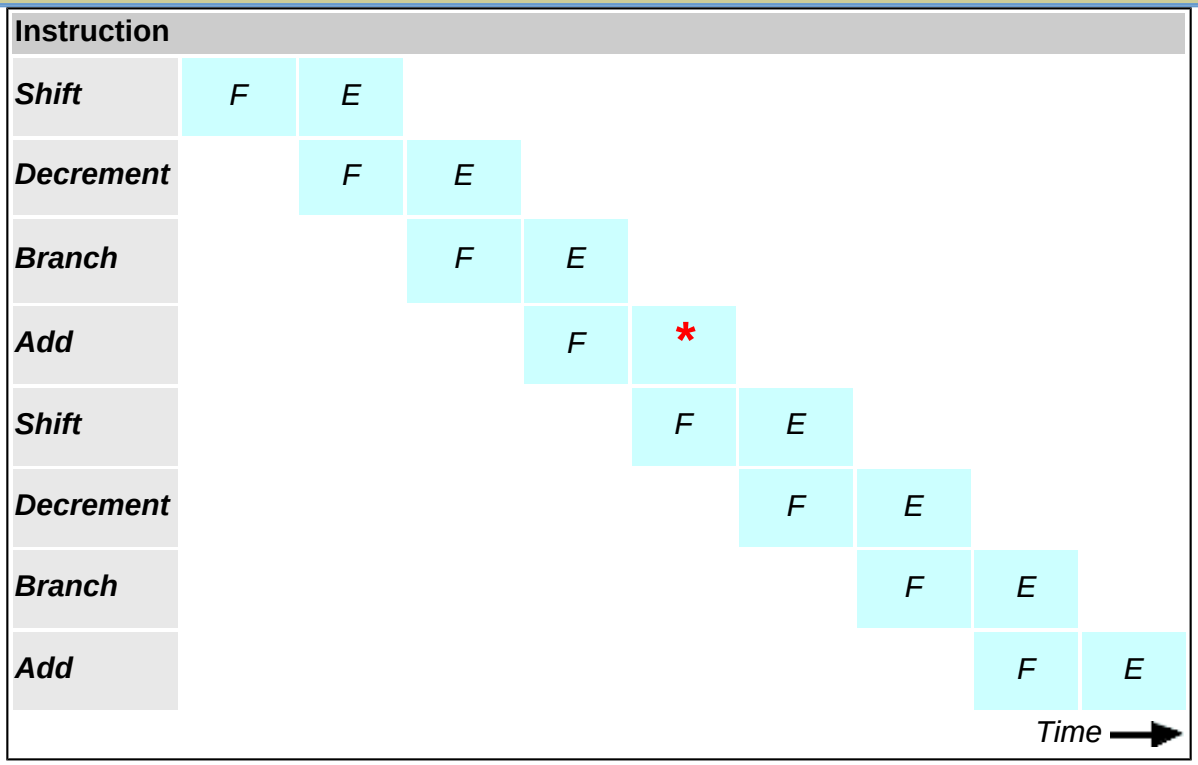
Logically the program is executed as if the branch instruction was placed after the shift instruction. That is, branching takes place one instruction later than where the branch instruction appears in the instruction sequence in the memory, hence the name “delayed branch” .



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

Branching : Print this page

<< Previous | First | Last | Next >>



Execution timing for last two passes through the loop of the original program loop.

*Note : Execution Unit Idle

<< Previous | First | Last | Next >>

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.

Symmetric Multiprocessors:

A symmetric multiprocessor (SMP) can be defined as a standalone computer system with the following characteristic:

1. There are two or more similar processor of comparable capability.
2. These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme.
3. All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.
4. All processors can perform the same functions.
5. The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file and data element levels.

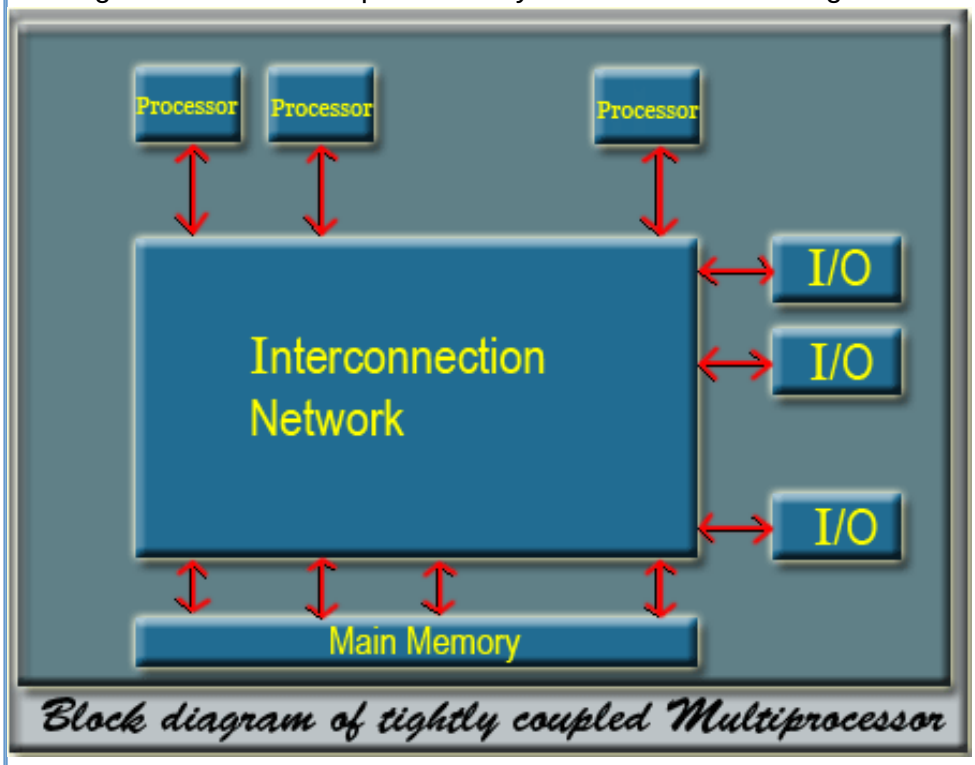
The operating system of a SMP schedules processors or thread across all of the processors. SMP has a potential advantages over uniprocessor architecture:

- Performance: A system with multiple processors will perform in a better way than one with a single processor of the same type if the task can be organized in such a manner that some portion of the work done can be done in parallel.

- Availability: Since all the processors can perform the same function in a symmetric multiprocessor, the failure of a single processor does not stop the machine. Instead, the system can continue to function at reduce performance level.
- Incremental growth: A user can enhance the performance of a system by adding an additional processor.
- Sealing: Vendors can offer a range of product with different price and performance characteristics based on number of processors configured in the system.

Organization:

The organization of a multiprocessor system is shown in the figure



Parallel Processing Print this page

<< Previous | First | Last | Next >>

- There are two or more processors. Each processor is self sufficient, including a control unit, ALU, registers and cache.
- Each processor has access to a shared main memory and the I/O devices through an interconnection network.
- The processor can communicate with each other through memory (messages and status information left in common data areas).
- It may also be possible for processors to exchange signal directly.
- The memory is often organized so that multiple simultaneous accesses to separate blocks of memory are possible.
- In some configurations each processor may also have its own private main memory and I/O channels in addition to the shared resources.

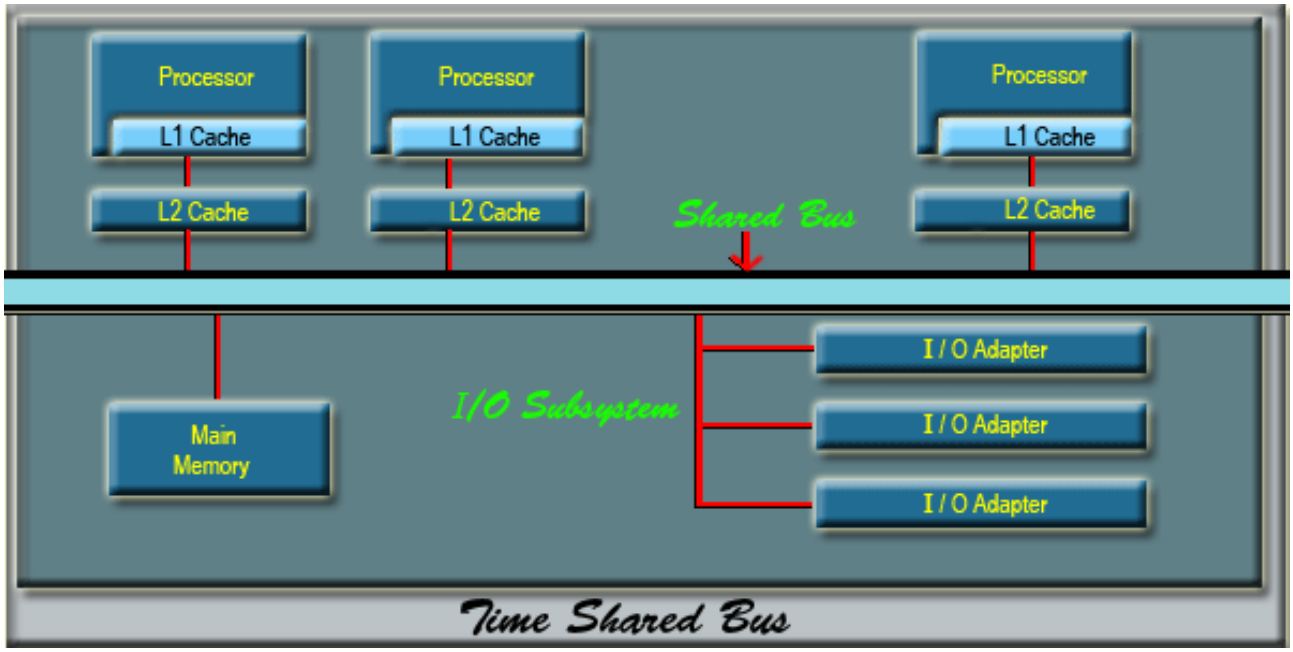
The organization of multiprocessor system can be classified as follows:

- Time shared or common bus
- Multiport memory
- Central control unit.

>> Previous | First | Last | Next >>

Time shared Bus:

Time shared bus is the simplest mechanism for constructing a multiprocessor system. The bus consists of control, address and data lines. The block diagram is shown in the figure



The following features are provided in time-shared bus organization:

- Addressing: It must be possible to distinguish modules on the bus to determine the source and destination of data
- Arbitration: Any I/O module can temporarily function as “master”. A mechanism is provided to arbitrate competing request for bus control, using some sort of priority scheme.
- Time shearing: when one module is controlling the bus, other modules are locked out and if necessary suspend operation until bus access in achieved.

The bus organization has several advantages compared with other approaches:

- Simplicity: This is the simplest approach to multiprocessor organization. The physical interface and the addressing, arbitration and time sharing logic of each processor remain the same as in a single processor system.
- Flexibility: It is generally easy to expand the system by attaching more processor to the bus.
- Reliability: The bus is essentially a passive medium and the failure of any attached device should not cause failure of the whole system.

The main drawback to the bus organization is performance. Thus, the speed of the system is limited by the bus cycle time.

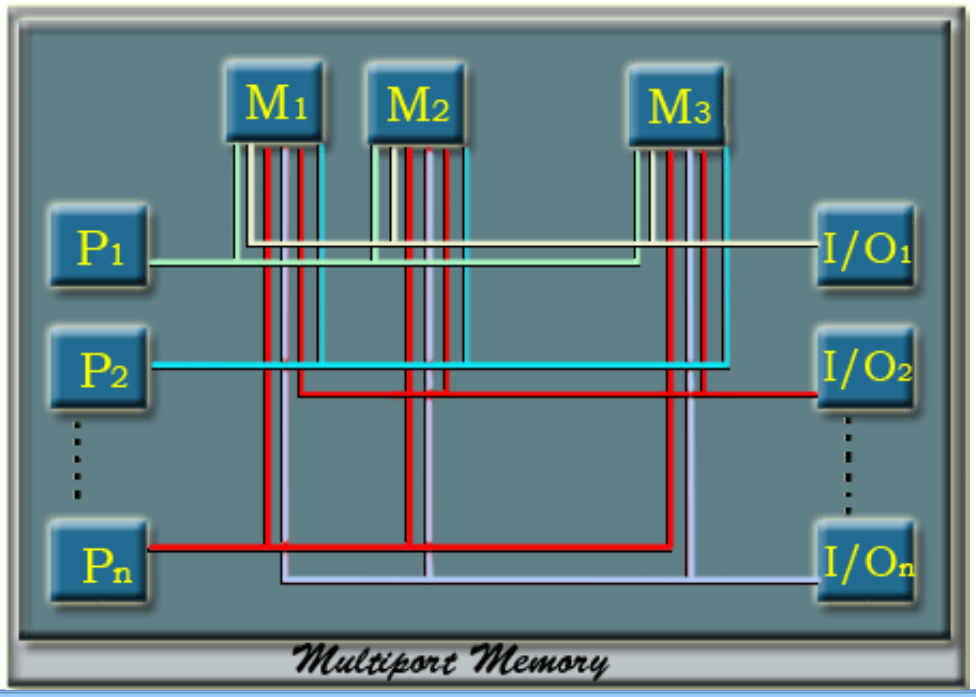
To improve performance, each processor can be equipped with local cache memory.

The use of cache leads to a new problem which is known as cache coherence problem. Each local cache contains an image of a portion of main memory. If a word is altered in one cache, it may invalidate a word in another cache. To prevent this, the other processors must perform an update in its local cache.

Multiport Memory:

The multiport memory approach allows the direct, independent access of main memory modules by each processor and io module.

The multiport memory system is shown in the figure



The multiport memory approach is more complex than the bus approach, requiring a fair amount of logic to be added to the memory system. Logic associated with memory is required for resolving conflict. The method often used to resolve conflicts is to assign permanently designated priorities to each memory port.

Non-uniform Memory Access(NUMA)

In NUMA architecture, all processors have access to all parts of main memory using loads and stores. The memory access time of a processor differs depending on which region of main memory is accessed. The last statement is true for all processors; however, for different processors, which memory regions are slower and which are faster differ.

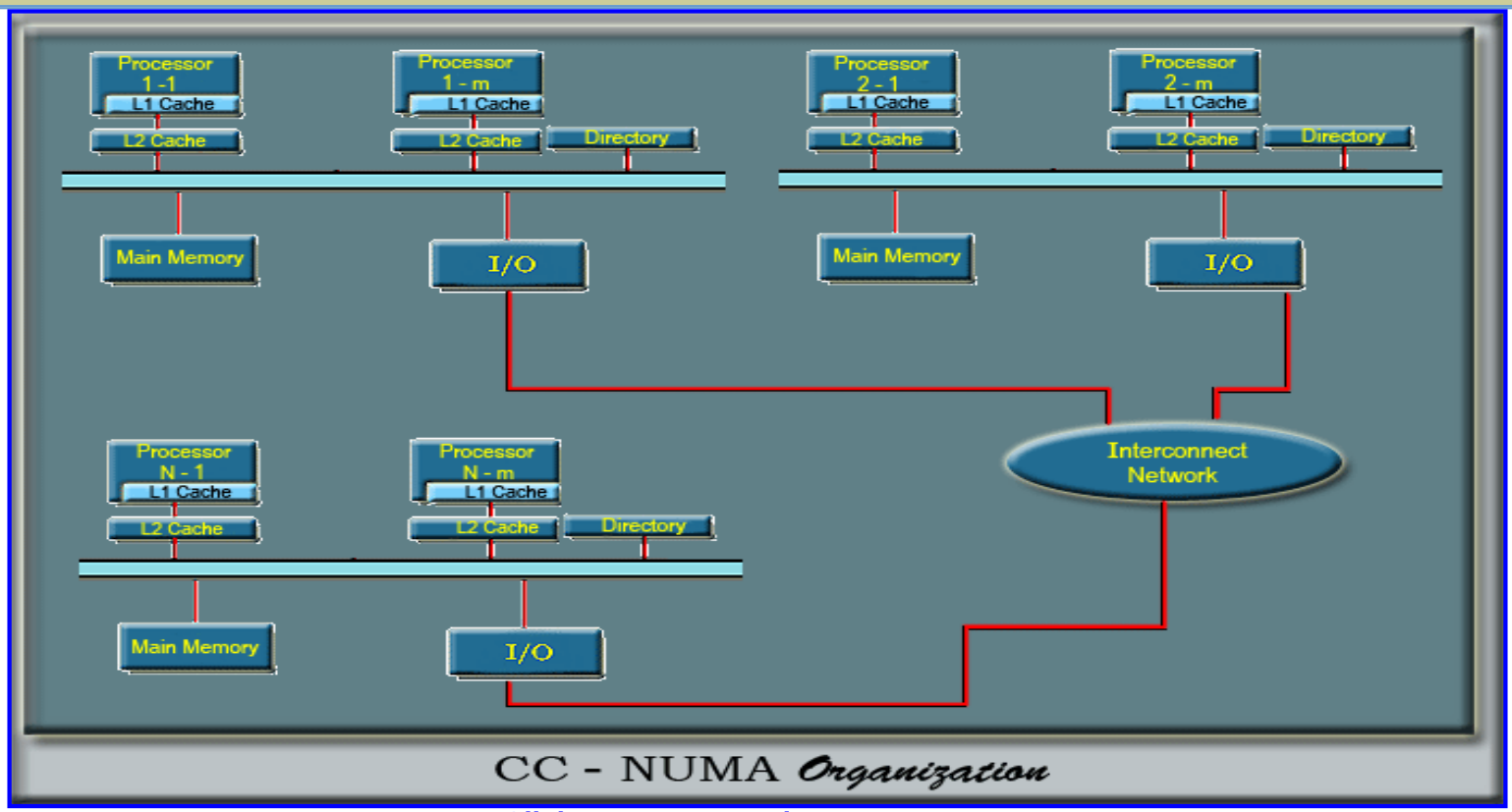
A NUMA system in which cache coherence is maintained among the cache of the various processors is known as cache-cohence NUMA (CC-NUMA)

A typical CC-NUMA organization is shown in the figure on the next slide.

Parallel Processing

Print this page

<< Previous | First | Last | Next >>



Click on Image To View Large Image

<< Previous | First | Last | Next >>

There are multiple independent nodes, each of which is, in effect, an SMP organization.

Each node contains multiple processors, each with its own *L1* and *L2* caches, plus main memory.

The node is the basic building block of the overall CC NUMA organization

The nodes are interconnected by means of some communication facility, which could be a switching mechanism, a ring, or some other networking facility.

Each node in the CC-NUMA system includes some main memory.

From the point of view of the processors, there is only a single addressable memory, with each location having a unique system-wide address.

When a processor initiates a memory access, if the requested memory location is not in the processors cache, then the *L2* cache initiates a fetch operation.

If the desired line is in the local portion of the main memory, the line is fetch across the local bus.

If the desired line is in a remote portion of the main memory, then an automatic request is send out to fetch that line across the interconnection network, deliver it to the local bus, and then deliver it to the requesting cache on that bus.

All of this activity is atomic and transparent to the processors and its cache.

In this configuration, cache coherence is a central concern. For that each node must maintain some sort of directory that gives it an indication of the location of various portion of memory and also cache status information.

Interconnection Networks:

In a multiprocessor system, the interconnection network must allow information transfer between any pair of modules in the system. The traffic in the network consists of requests (such as read and write), data transfers, and various commands.

Single Bus:

The simplest and most economical means of interconnecting a number of modules is to use a single bus.

Since several modules are connected to the bus and any module can request a data transfer at any time, it is essential to have an efficient bus arbitration scheme.

In a simple mode of operation, the bus is dedicated to a particular source-destination pair for the full duration of the requested transfer. For example, when a processor uses a read request on the bus, it holds the bus until it receives the desired data from the memory module.

Since the memory module needs a certain amount of time to access the data bus, the bus will be idle until the memory is ready to respond with the data.

Then the data is transferred to the processors. When this transfer is completed, the bus can be assigned to handle another request.

A scheme known as the split- transaction protocol makes it possible to use the bus during the idle period to serve another request.

Consider the following method of handling a series of read requests possibly from different processor.

After transferring the address involved in the first request, the bus may be reassigned to transfer the address of the second request; assuming that this request is to a different memory module.

At this point, we have two modules proceeding with read access cycle in parallel.

If neither module has finished with its access, the bus may be reassigned to a third request and so on.

Eventually, the first memory module completes its access cycle and uses the bus to transfer the data to the corresponding source.

As other modules complete their cycles, the bus is needed to transfer their data to the corresponding sources.

The split transaction protocol allows the bus and the available bandwidth to be used more efficiently. The performance improvement achieved with this protocol depends on the relationship between the bus transfer time and the memory access time.

In split- transaction protocol, performance is improved at the cost of increased bus complexity.

There are two reasons why complexity increases:

- Since a memory module needs to know which source initiated a given read request, a source identification tag must be attached to the request.
- Complexity also increases because all modules, not just the processor, must be able to act as bus muster.

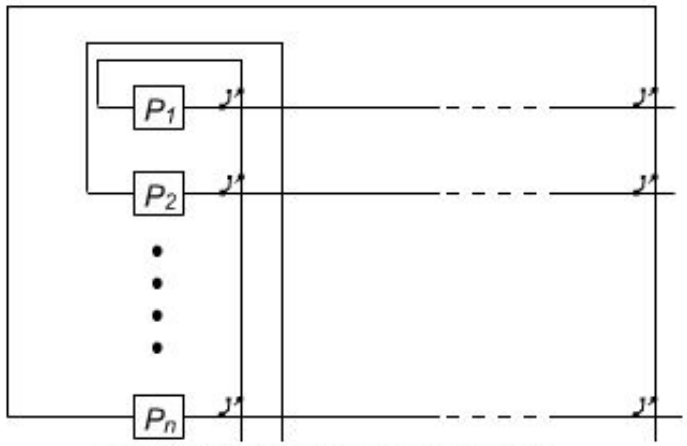
The main limitation of a single bus is that the number of modules that can be connected to the bus is not that large. Networks that allow multiple independent transfer operations to proceed in parallel can provide significantly increased data transfer rate.

Crossbar Network:

Crossbar switch is a versatile switching network. It is basically a network of switches. Any module P_i can be connected to any other module P_j by closing the appropriate switch. Such networks, where there is a direct link between all pairs of nodes are called fully connected networks.

In a fully connected network, many simultaneous transfers are possible. If *n* sources need to send data to *n* distinct destinations then all of these transfers can take place concurrently. Since no transfer is prevented by the lack of a communication path, the crossbar is called a **nonblocking switch**.

In the figure of crossbar interconnection network, a single switch is shown at each cross point. In actual multiprocessor system, the paths through the crossbar network are much wider.



Crossbar Interconnection Network

Interconnection Networks

If there are n modules in a network, than the number of cross point is n^2 in a network to interconnect n modules. The total number of switches becomes large as n increases.

In a crossbar switch, conflicts occur when two or more concurrent requests are made to the same destination device. These conflicting requests are usually handled on a predetermined priority basis.

The crossbar switch has the potential for the highest bandwidth and system efficiency. However, because of its complexity and cost, it may be cost effective for a large multiprocessor system.

Multistage Network:

The bus and crossbar systems use a single stage of switching to provide a path from a source to a destination.

In multistage network, multiple stages of switches are used to setup a path between source and destination.

Such networks are less costly than the crossbar structure, yet they provide a reasonably large number of parallel paths between source and destinations.

In the figure it shows a three-stage network that called a shuffle network that interconnects eight modules.

The term "shuffle" describes the pattern of connections from the outputs of one stage to the inputs of the next stage.

The switchbox in the figure is a 2×2 switch that can route either input to either output.

If the inputs request distinct outputs, they can both be routed simultaneously in the straight through or crossed pattern.

If both inputs request the same output, only one request can be satisfied. The other one is blocked until the first request finishes using the switch.

Interconnection Networks

Print this page

<< Previous | First | Last | Next >>

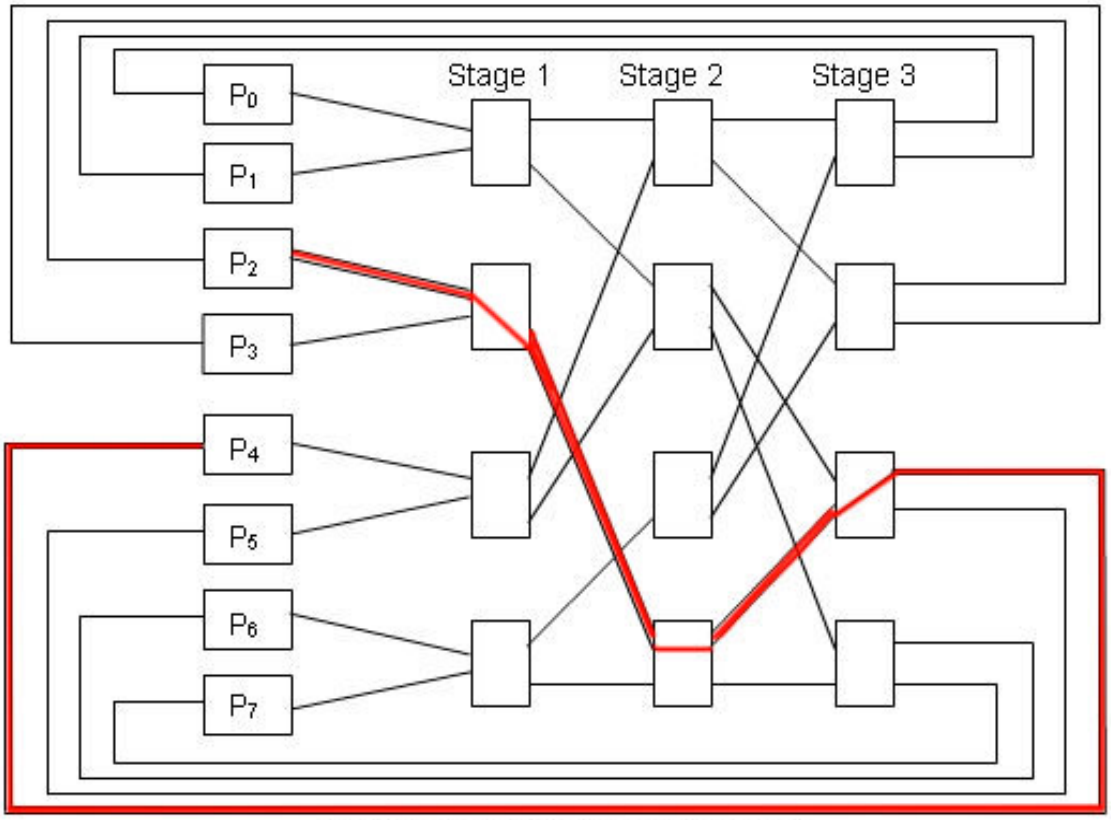


Figure : Multistage Shuffle Network

<< Previous | First | Last | Next >>

Interconnection Networks Print this page

<< Previous | First | Last | Next >>

A network consisting of s stages can be used to interconnect 2^s modules. In this case, there is exactly one path through the network from any module P_i to any module P_j. Therefore, this network provides full connectivity between sources and destinations.

Many request patterns cannot be satisfied simultaneously. For example, the connection from P_2 to P_7 can not be provided at the same time as the connection from P_3 to P_6.

A multistage network is less expansive to implement than a crossbar network. If n nodes are to be interconnected using this scheme, then we must use $s = \log_2 n$ stages with $n/2$ switches per stage. Since each switches contains four switches, the total number of switches is

$$4 \times \frac{n}{2} \times \log_2 n = 2n \times \log_2 n$$

which, for a large network, is considerably less than the n^2 switches needed in a crossbar network.

Multistage networks are less capable of providing concurrent connection than crossbar switches. The connection path between P_2 and P_4 is indicated by RED lines in the figure.

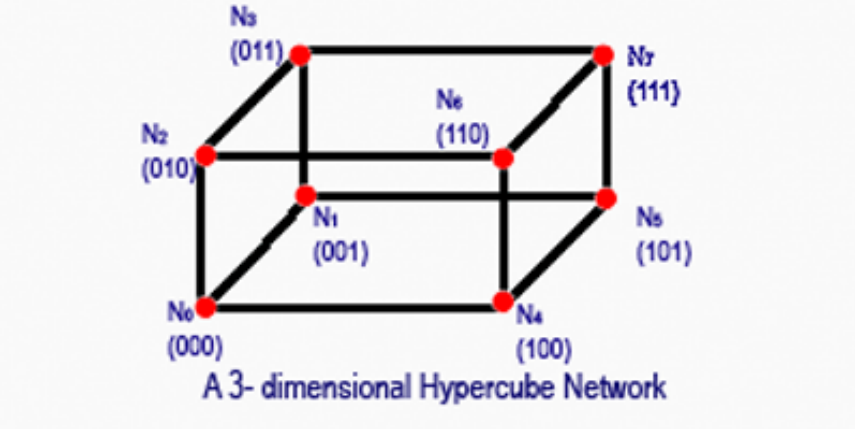
>> Previous | First | Last | Next >>

Hypercube Networks:

A hypercube is an n-dimensional cube that interconnects 2^n nodes. In addition to the communication circuit, each node usually includes a processor and a memory module as well as some I/O capability.

The figure shows a three dimensional hypercube. The small circles represent the communication circuits in the nodes. The edge of the cube represent bi-directional communication links between neighboring nodes.

In an an n-dimensional hypercube each node is directly connected to n neighbors. A useful way to label the nodes is to assign binary addresses to them in such a way that the addresses of any two neighbors differs in exactly one bit position. The functional units are attached to each node of the hypercube.



A 3- dimensional Hypercube Network

Routing messages through the hypercube is easy. If the processor at node N_i wishes to send a message to node N_j, it proceeds as follows:

- The binary addresses of the source, i, and the destination, j, are compared from least to most significant bits.
- Suppose that they differ first in position P
- Node N_i then sends the message to its neighbor whose address, k, differs from i in bit position P
- Node N_k forwards the message to the appropriate neighbor using the same address comparison scheme
- The message gets closer to destination node N_j with each of these hops from one node to another.
- For example, a message from node N_0 to N_5 transverse the following way:

$$N_0: 0\ 0\ 0 \qquad N_5: 1\ 0\ 1$$

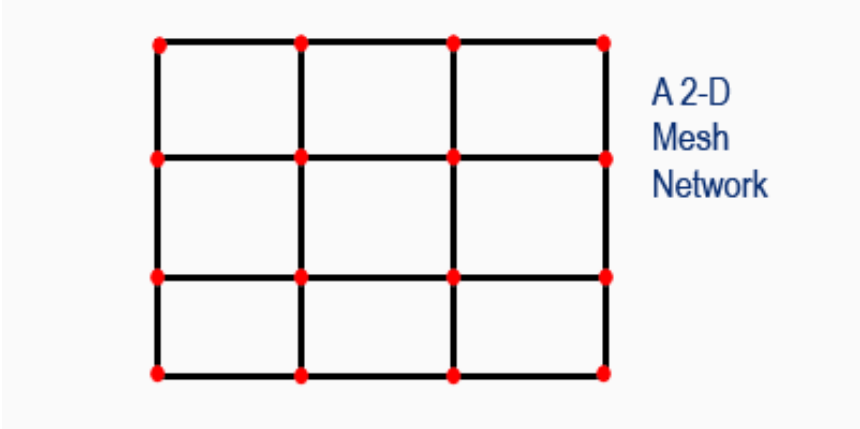
Message traverses from node N_0 to N_1, they differ in 1st bit position.

Then message traverses from N_1 to N_5, they differ in 3 rd bit position.

Therefore, it takes two hops. The maximum distance that any message needs to travel in an n - dimensional hypercube is n - hops.

Mesh networks:

Mesh network is another way to interconnect a large number of nodes in a multiprocessor system. An example of a mesh with 16 nodes is given in the figure.



The link between the nodes are bi-directional.

The functional unit are attached to the each node of the mesh network.

Routing in a mesh network can be done in several ways.

One of the simplest and most effective possibilities is to choose the path between a source node N_i and a destination node N_j such that the transfer first takes place in the horizontal directional from N_i towards N_j.

When the column in which N_j resides is reached, the transfer proceeds in the vertical direction along this column. If a wraparound connection is made between the nodes at the opposite edges of a mesh network, the result is a network that comprises a set of bi-directional rings in the X direction connected by a set of rings in the Y direction.

This network is called a torus. The average latency of information transfer is reduced in a torus, but the complexity increases.

Tree Networks:

A hierarchically structured network implemented in the form of a tree is another interconnection topology. A four way tree that interconnects 16 modules is shown in the figure.

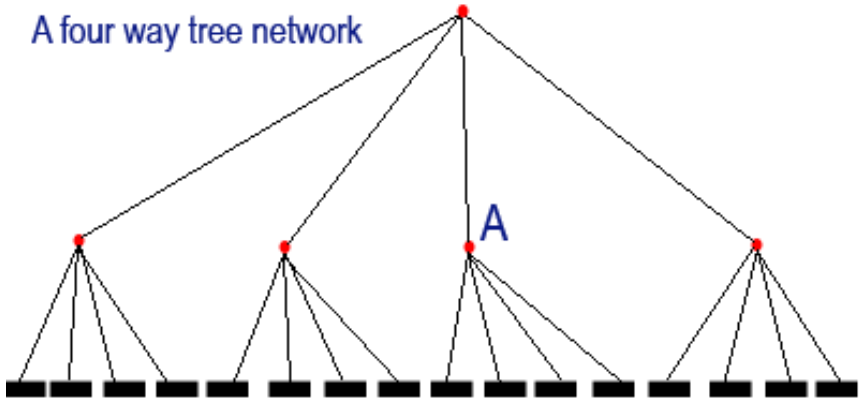
In this tree, each parent node allows communication between two of its children at a time.

An intermediate-level node, for example node A in the figure, can provide a connection from one of its child node to its parent.

This enables two leaf nodes that are any distance apart to communicate.

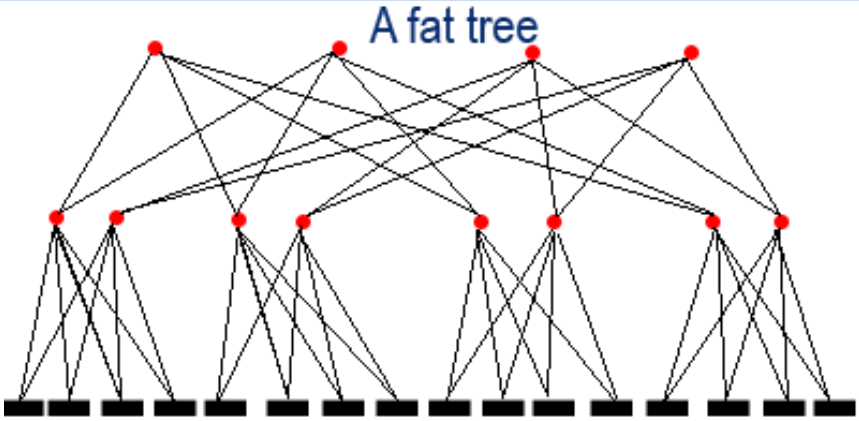
Only one path at any time can be established through a given node in the tree.

Interconnection Networks



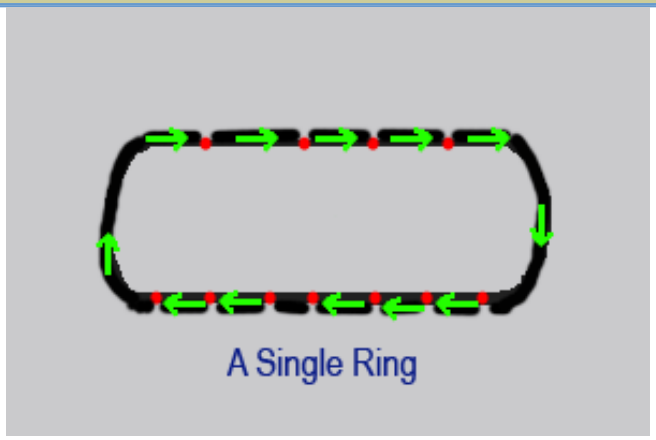
To reduce the possibility of a bottleneck, the number of links in the upper levels of a tree hierarchy can be increased. This is done in a fat tree network, in which each node in the tree (except at the top level) has more than one parent.

In the figure shown a fat tree in which each node has two parents.



Ring Networks:

One of the simplest network topologies uses a ring to interconnect the nodes in the system. A single ring is shown in the figure.



The main advantage of the arrangement is that the ring is easy to implement. Links in the ring can be wide, because each node is connected to only two neighbors. It is not useful to construct a very long ring to connect many nodes because the latency of information transfer would be unacceptably large

The simple possibility of using ring in a tree structure; this results in a hierarchy of rings.

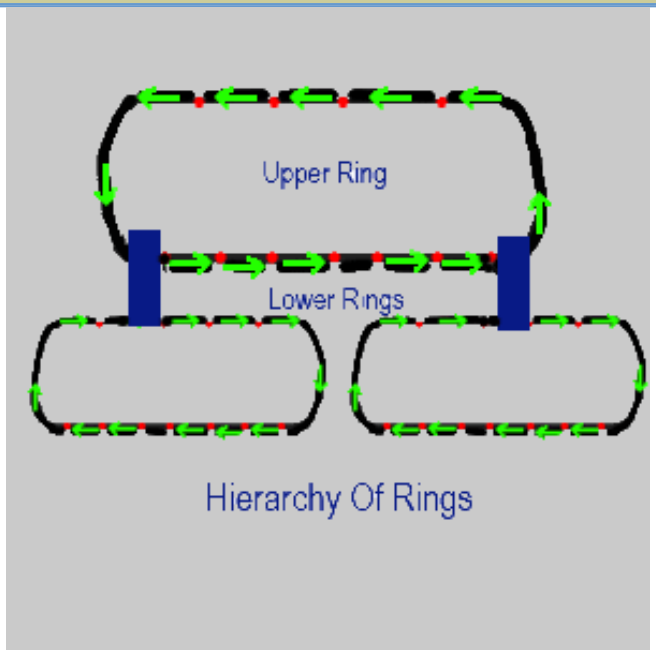
Having short rings reduces substantially the latency of transfers that involve nodes on the same ring.

The latency of transfers between two nodes on different rings is shorter than if a single ring were used.

Interconnection Networks

Print this page

<< Previous | First | Last | Next >>



Hierarchy Of Rings

<< Previous | First | Last | Next >>

Cache Coherence:

In contemporary multiprocessor system, it is customary to have one or two levels of cache associated with each processor. This organization is essential to achieve high performance.

Cache creates a problem, which is known as the cache coherence problem. **The cache coherence problem is : Multiple copies of the same data can exist in different caches simultaneously, and if processors are allowed to update their own copies freely, an inconsistent view of memory can result.**

There are two write policies:

- **Write back :** Write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is flushed from the cache.
- **Write through :** All write operations are made to main memory as well as to the cache, ensuring that main memory is always valid.

It is clear that a write back policy can result in inconsistency.

If two caches contain the same line, and the line is updated in one cache, the other cache will unknowingly have an invalid value. Subsequently read to that invalid line produce invalid results.

Cache Coherence

Even with the write through policy, inconsistency can occur unless other cache monitor the memory traffic or receive some direct notification of the update.

For any cache coherence protocol, the objective is to let recently used local variables get into the appropriate cache and stay there through numerous reads and write, while using the protocol to maintain consistency of shared variables that might be in multiple caches at the same time.

Write through protocol:

A write through protocol can be implemented in two fundamental versions.

(a) Write through with update protocol
(b) Write through with invalidation of copies

Write through with update protocol:

When a processor writes a new value into its cache, the new value is also written into the memory module that holds the cache block being changed. Some copies of this block may exist in other caches, these copies must be updated to reflect the change caused by the write operation.

The simplest way of doing this is to broadcast the written data to all processor modules in the system.

As each processor module receives the broadcast data, it updates the contents of the affected cache block if this block is present in its cache.

Write through with invalidation of copies :

When a processor writes a new value into its cache, this value is written into the memory module, and all copies in the other caches are invalidated. Again broadcasting can be used to send the invalidation requests through the system.

Write back protocol:

In the write-back protocol, multiple copies of a cache block may exist if different processors have loaded (read) the block into their caches.

If some processor wants to change this block, it must first become an exclusive owner of this block.

When the ownership is granted to this processor by the memory module that is the home location of the block. all other copies, including the one in the memory module, are invalidated.

Now the owner of the block may change the contents of the memory.

Cache Coherence

Print this page

<< Previous | First | Last | Next >>

When another processor wishes to read this block, the data are sent to this processor by the current owner.

The data are also sent to the home memory module, which requires ownership and updates the block to contain the latest value.

There are software and hardware solutions for cache coherence problem.

Software solution:

In software approach, the detecting of potential cache coherence problem is transferred from run time to compile time, and the design complexity is transferred from hardware to software

On the other hand, compile time software approaches generally make conservative decisions, leading to inefficient cache utilization.

Compiler-based cache coherence mechanism perform an analysis on the code to determine which data items may become unsafe for caching, and they mark those items accordingly. So, there are some non cacheable items, and the operating system or hardware does not cache those items.

The simplest approach is to prevent any shared data variables from being cached.

<< Previous | First | Last | Next >>

This is too conservative, because a shared data structure may be exclusively used during some periods and may be effectively read-only during other periods.

It is only during periods when at least one process may update the variable and at least one other process may access the variable then cache coherence is an issue.

More efficient approaches analyze the code to determine safe periods for shared variables. The compiler then inserts instructions into the generated code to enforce cache coherence during the critical periods.

Hardware solution:

Hardware solution provides dynamic recognition at run time of potential inconsistency conditions. Because the problem is only dealt with when it actually arises, there is more effective use of caches, leading to improved performances over a software approach.

Hardware schemes can be divided into two categories: **(a) directory protocol** and **(b) snoopy protocols**.

(a) Directory protocols:

Directory protocols collect and maintain information about where copies of lines reside. Typically, there is a centralized controller that is part of the main memory controller, and a directory that is stored in main memory.

Cache Coherence

Print this page

<< Previous | First | Last | Next >>

The directory contains global state information about the contents of the various local caches.

When an individual cache controller makes a request, the centralized controller checks and issues necessary commands for data transfer between memory and caches or between caches themselves.

It is also responsible for keeping the state information up to date, therefore, every local action that can effect the global state of a line must be reported to the central controller.

The controller maintains information about which processors have a copy of which lines.

Before a processor can write to a local copy of a line, it must request exclusive access to the line from the controller.

Before granting this exclusive access, the controller sends a message to all processors with a cached copy of this time, forcing each processors to invalidate its copy.

After receiving acknowledgement back from each such processor, the controller grants exclusive access to the requesting processor.

When another processor tries to read a line that is exclusively granted to another processors, it will send a miss notification to the controller.

<< Previous | First | Last | Next >>

The controller then issues a command to the processor holding that line that requires the processors to do a write back to main memory.

Directory schemes suffer from the drawbacks of a **central bottleneck** and the **overhead of communication** between the various cache controllers and the central controller.

(b) Snoopy protocols:

Snoopy protocols distribute the responsibility for maintaining cache coherence among all of the cache controllers in a multiprocessor system.

A cache must recognize when a line that it holds is shared with other caches.

When an update action is performed on a shared cache line, it must be announced to all other caches by a broadcast mechanism.

Each cache controller is able to "snoop" on the network to observed these broadcasted notification and react accordingly.

Snoopy protocols are ideally suited to a bus-based multiprocessor, because the shared bus provides a simple means for broadcasting and snooping.

Cache Coherence

Two basic approaches to the snoopy protocol have been explored: **Write invalidates** or **write-update (write-broadcast).**

(i) Write invalidates:

With a write-invalidate protocol, there can be multiple readers but only one write at a time.

Initially, a line may be shared among several caches for reading purposes.

When one of the caches wants to perform a write to the line it first issues a notice which invalidates the line in the other caches, making the line exclusive to the writing cache.

Once the line is exclusive, the owning processor can make local writes until some other processor requires the same line.

(il) Write update (write broadcast):

With a write update protocol, there can be multiple writers as well as multiple readers. When a processors wishes to update a shared line, the word to be updated is distributed to all others, and caches containing that line can update it.

Module 11 : Intel 8085/8086 Micro-Processor

In this Module, we have five lectures, viz.

1. **Organization of Intel 8085 Micro-Processor**
2. **Instruction set of Intel 8085 Micro-Processor**
3. **Organization of Intel 8086 Micro-Processor**
4. **Instruction formats of Intel 8086**
5. **Programming of Micro-processor**

Click the proper link on the left side for the lectures

Organization of Intel 8085 Microprocessors :

The microprocessors that are available today came with a wide variety of capabilities and architectural features. All of them, regardless of their diversity, are provided with at least the following functional components, which form the central processing unit (CPU) of a classical computer.

a. Register Section : A set of registers for temporary storage of instructions, data and address of data .

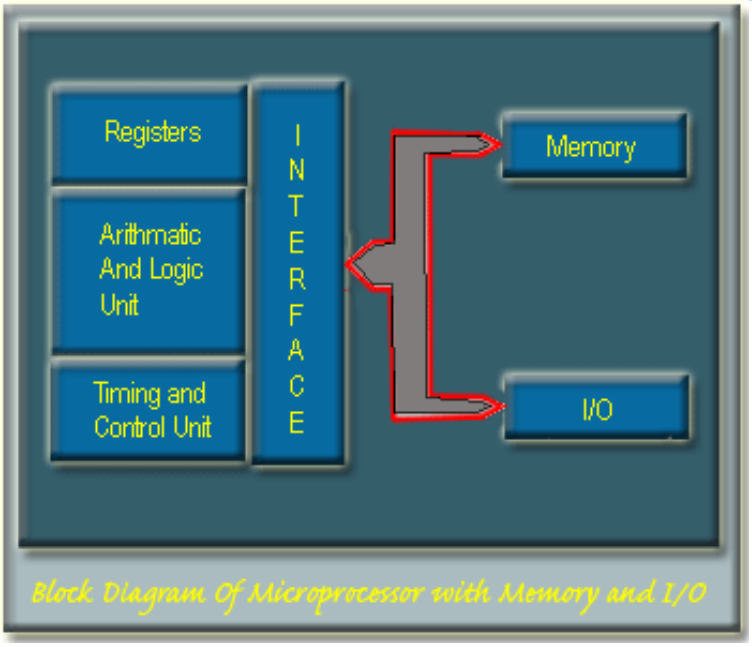
b. Arithmetic and Logic Unit : Hardware for performing primitive arithmetic and logical operations .

c. Interface Section : Input and output lines through which the microprocessor communicates with the outside world .

d. Timing and Control Section : Hardware for coordinating and controlling the activities of the various sections within the microprocessor and other devices connected to the interface section .

The block diagram of the microprocessor along with the memory and Input/Output (I/O) devices is shown in the figure on the next slide.

Organization of Intel 8085 Microprocessors



Block Diagram Of Microprocessor with Memory and I/O

Intel Microprocessors:

Intel 4004 is the first 4-bit microprocessor introduced by Intel in 1971. After that Intel introduced its first 8-bit microprocessor 8088 in 1972.

These microprocessors could not last long as general-purpose microprocessors due to their design and performance limitations.

In 1974, Intel introduced the first general purpose 8-bit microprocessor 8080 and this is the first step of Intel towards the development of advanced microprocessor.

After 8080, Intel launched microprocessor 8085 with a few more features added to its architecture, and it is considered to be the first functionally complete microprocessor.

The main limitations of the 8-bit microprocessors were their low speed, low memory capacity, limited number of general purpose registers and a less powerful instruction set .

To overcome these limitations Intel moves from 8-bit microprocessor to 16-bit microprocessor.

In the family of 16-bit microprocessors, Intel's 8086 was the first one introduced in 1978 .

8086 microprocessor has a much powerful instruction set along with the architectural developments, which imparted substantial programming flexibility and improvement over the 8-bit microprocessor.

Microprocessor Intel 8085 :

Intel 8085 is the first popular microprocessor used by many vendors. Due to its simple architecture and organization, it is easy to understand the working principle of a microprocessor.

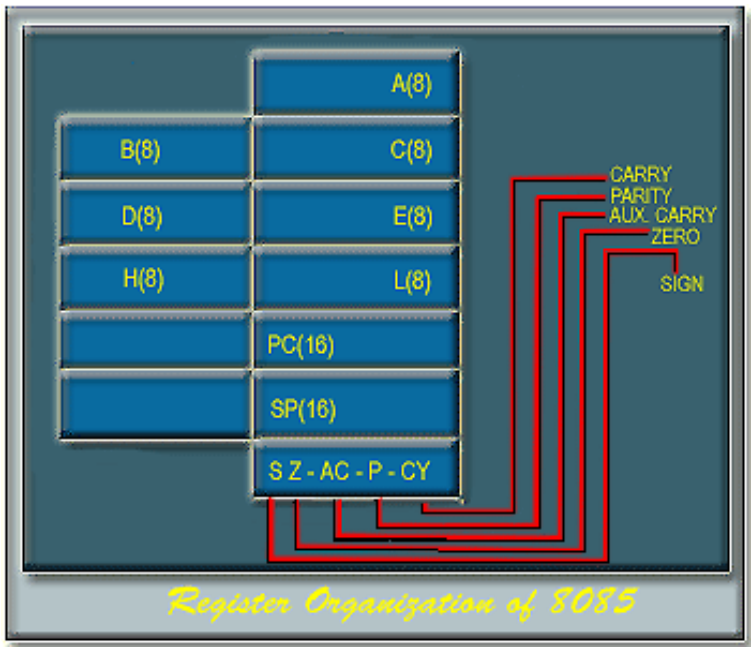
Register in the Intel 8085:

The programmable registers of 8085 are as follows -

- One 8-bit accumulator A.
- Six 8-bit general purpose register (GPR's)
  B, C, D , E , H and L.
- The GPR's are also accessible as three 16-bit register pairs BC, DE and HL.
- There is a 16-bit program counter(PC), one 16-bit stack pointer(SP) and 8-bit flag register . Out of 8 bits of the flag register , only 5 bits are in use.

Organization of Intel 8085 Microprocessors

The programmable registers of the 8085 are shown in the figure -



Apart from these programmable registers , some other registers are also available which are not accessible to the programmer . These registers include -

- Instruction Register(IR).
- Memory address and data buffers(MAR & MDR).
  - MAR: Memory Address Register.
  - MDR: Memory Data Register.
- Temporary register for ALU use.

ALU of 8085 :

The 8-bit parallel ALU of 8085 is capable of performing the following operations –

Arithmetic : Addition, Subtraction, Increment, Decrement, Compare.

Logical : AND, OR, EXOR, NOT, SHIFT / ROTATE, CLEAR.

Because of limited chip area , complex operations like multiplication, division, etc are not available, in earlier processors like 8085.

The operations performed on binary 2's complement data.

The five flag bits give the status of the microprocessor after an ALU operation.

The carry (C) flag bit indicates whether there is any overflow from the MSB.

The parity (P) flag bit is set if the parity of the accumulater is even.

The Auxiliary Carry (AC) flag bit indicates overflow out of bit –3 (lower nibble) in the same manner, as the C-flag indicates the overflow out of the bit-7.

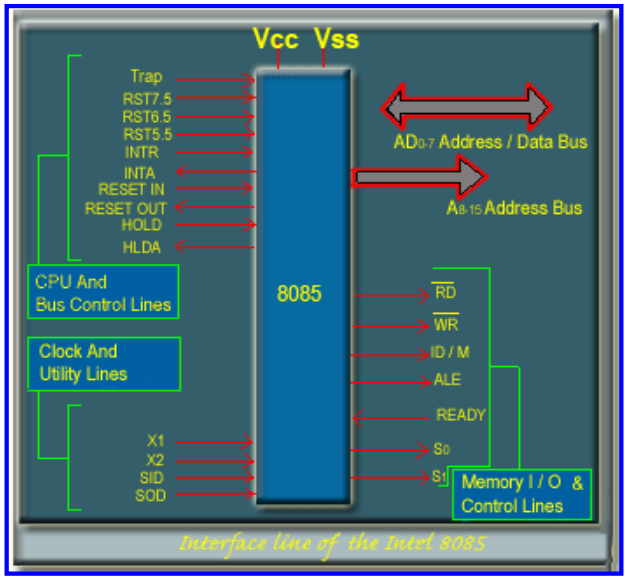
The Zero (Z) flag bit is set if the content of the accumulator after any ALU operations is zero.

The Sign(S) flag bit is set to the condition of bit-7 of the accumulator as per the sign of the contents of the accumulator(positive or negative).

The Interface Section:

Microprocessor chips are equipped with a number of pins for communication with the outside world. This is known as the system bus.

The interface lines of the Intel 8085 microprocessor are shown in the figure-



Click on Image To View Large Image

Address and Data Bus

The AD0 - AD7 lines are used as lower order 8-bit address bus and data bus , in time division multiplexed manner .

The A8 - A15 lines are used for higher order 8 bit of address bus.

There are seven memory and I/O control lines -

RD : indicates a READ operation when the signal is LOW .

WR : indicates a WRITE operation when the signal is LOW .

IO/M : indicates memory access for LOW and I/O access for HIGH .

ALE : ALE is an address latch enable signal , this signal is HIGH when address information is present in AD0-AD7 . The falling edge of ALU can be used to latch the address into an external buffer to de-multiples the address bus .

READY : READY line is used for communication with slow memory and I/O devices .

S0 and S1 : The status of the system bus is difined by the S0 and S1 lines as follows -

S1	S0	Operation Specified
0	0	Halt
0	1	Memory or I/O WRITE
1	0	Memory or I/O READ
1	1	Instruction Fetch

There are ten lines associated with CPU and bus control-

- TRAP , RST7.5 , RST6.5 , RST5.5 and INTR to acknowledge the INTA output.
- RESET IN : This is the reset input signal to the 8085.
- RESET OUT : The 8085 generates the RESET-OUT signal in response to RESET-IN signal , which can be used as a system reset signal .
- HOLD : HOLD signal is used for DMA request.
- HLDA : HLDA signal is used for DMA grant .

Organization of Intel 8085 Microprocessors

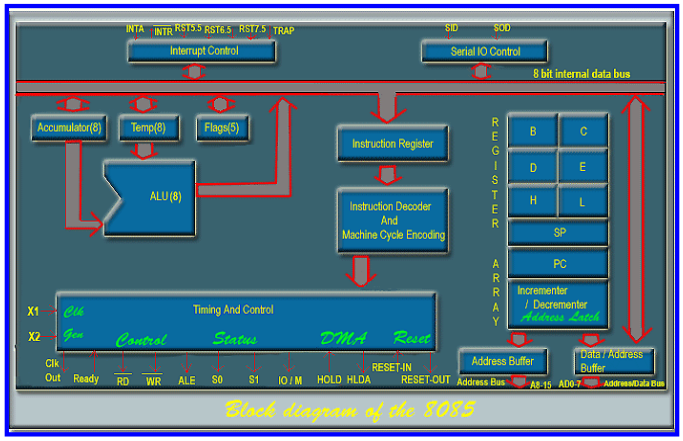
Print this page

<< Previous | First | Last | Next >>

Clock and Utility Lines :

X1 and X2: X1 and X2 are provided to connect a crystal or a RC network for generating the clock internal to the chip.
SID: input line for serial data communication.
SOD: output line for serial data communication.
V_{CC} and V_{SS}: Power supply.

The block diagram of the Intel 8085 is shown in the figure -



Click on Image To View Large Image

<< Previous | First | Last | Next >>

Instruction and data formats :

Memory used in the Intel 8085 microprocessor is organized in 8-bit, i.e., It is byte organized.

Every byte has a unique location in physical memory.

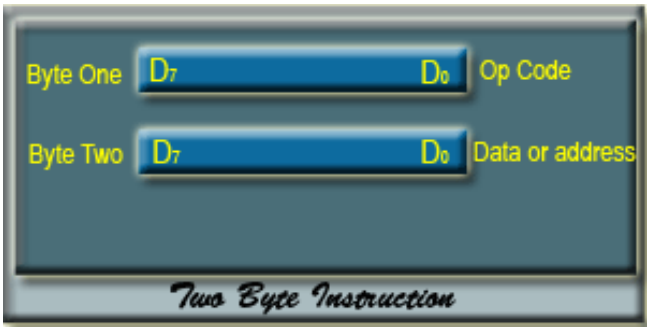
The address bus of 8085 is 16-bit wide , so the location of memory is described by one of a sequence of 16-bit binary address.

The 8085 can address up to 64k(i.e. 65,535) bytes of memory , which may consist of both RAM and ROM.

Data in 8085 is stored in the form of 8-bit binary integers-

And 8085 program instruction may be one , two or three bytes in length.

Multiple byte instructions must be stored in successive memory locations; the address of the first byes is always used as the address of the instruction.



Organization of Intel 8085 Microprocessors

Addressing Modes :

The 8085 has four different modes for addressing data stored in memory or in registers -

Direct: Bytes 2 and 3 of the instruction contains the exact memory address of the data item(the low-order bits of the address are in byte 2 , the high-order bits in byte 3).

Register: The instruction specifies the register or register pair in which the data are located.

Register Indirect: The instruction specifies a register pair which contains the memory address where the data are located .(the high-order bits of the address are in the first register of the pair and the low order bits in the second).

Immediate: The instruction contains the data itself . This is either and 8-bit quantity or a 16-bit quantity (least significant byte first , most significant byte second).

Unless directed by an interrupt or branch instruction the execution of instructions proceeds through consecutively increasing memory locations.

A branch instruction can specify the address of the next instruction to be executed in one of two ways -

Direct: The branch instruction contains the address of the next instruction to be executed .

Register Indirect : The branch instruction indicates a register pair which contains the address of the next instruction to be executed .

Instruction Set :

The complete instruction set of 8085 can be grouped in five different functional groups -

1. **Data Transfer group:** Moves data between registers, or between memory location and registers, includes moves, loads, stores and exchanges.
2. **Arithmetic group:** Adds, subtracts, increments , or decrements data in registers or memory .
3. **Logic group :** AND’s , OR’s , XOR’s , compares , rotates , or complements data in registers or between memory and a register .
4. **Branch group :** Initiates conditional or unconditional jumps , calls , returns , and restarts .
5. **Stack , I/O and machine control group :** Includes instructions for maintaining the sack , reading from input ports , writing to output ports , setting and reading interrupt masks , and setting and clearing flags .

Instruction Set of Intel 8085 microprocessor :
Data Transfer Group:

This group of instructions transfers data to and from registers and memory. Condition flags are not affected by any instruction in this group.

The bit pattern designating one of the registers *A, B, C, D, E, H, L.*

D D D or S S S	Register Name
111	*A*
000	*B*
001	*C*
010	*D*
011	*E*
100	*H*
101	*L*

Instruction Set of Intel 8085 microprocessor

Move Register : MOV r_1, r_2 $(r_1) \leftarrow (r_2)$

The content of register r_2 is moved to register r_1

These are single byte instructions and the format is

For example :

This machine instruction indicates

MOV A, B

i.e. the contents of register B will be moved to accumulator A.

The op-code of this machine instruction in hexadecimal is

7 8

and this is a single byte instruction

MOV r , *M* (Move from memory)

$(r) \leftarrow ((H)(L))$

The content of the memory location, whose address is in registers *H* and *L* , is moved to register *r* .

It is observed that there is no register available for the encoding 1 1 0 . So, in source field *S S S* , the contents 1 1 0 indicates a memory access.

Example :

Assume that the contents of *H* is 10_H and *L* is 00_H. Then the instruction

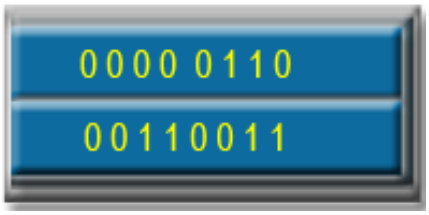
Moves the contents of memory location 1000_H to accumulator. The op-code of this machine instruction in hexadecimal is $7E$

MVI r , data (Move immediate)

$(r) \leftarrow (\text{byte } 2)$

It is a 2-byte instruction, the content of the byte 2 of the instruction is moved to register *r* .

Example



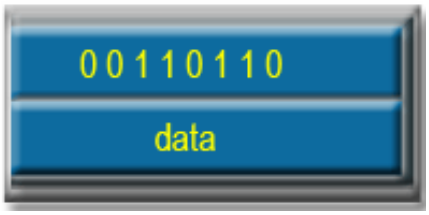
The data 0 0 1 1 0 0 1 1 i.e 33_H will be moved to register *B* .

The machine instruction in hexadecimal is $06_H 33_H$

MVI M , data (Move to memory immediate)

$((H)(L)) \leftarrow (\text{byte } 2)$

The content of byte 2 of the instruction is moved to the memory location whose address is in register *H* and *L* .



LXI rp , data 16 (Load register pair immediate)

$(rh) \leftarrow (\text{byte } 3)$

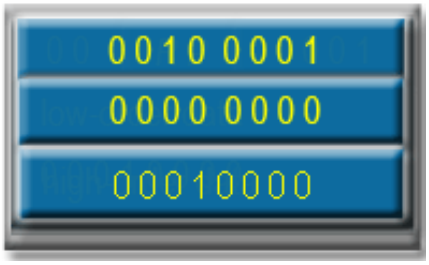
$(rl) \leftarrow (\text{byte } 2)$

Byte 3 of the instruction is moved in to the higher order register (rh) of the register pair *rp*. Byte 2 of the instruction is moved into the low-order register (rl) of the register pair rp .

The bit pattern in RP designating one of the register pairs *B* , *D* , *H* & *SP* :

RP	Register - Pair
00	*B-C*
01	*D-E*
10	*H-L*
11	*S-P*

For example :



This instruction loads the register pair *H-L* by 1000_H

The machine instruction in hexadecimal format is
$21_H\ 00_H\ 10_H$

LDA addr (Load accumulator direct)

$A \leftarrow$ ((byte 3) (byte 2))

The contents of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register *A* . It is a three byte instruction.

The instruction $3A_H00_H20_H$ moves the contents of memory location 2000_H to accumulator.

STA addr (store accumulator direct)

((byte-3) (byte-2)) $\leftarrow (A)$

The content of the accumulator is moved to the memory location whose address is specified in byte 2 and byte 3 of the instruction.

The instruction $32_H 10_H 20_H$ moves the contents of accumulator into memory location 2010_H

LHLD addr (Load *H* and *L* direct)

(*L*) ⇐ ((byte 3)(byte 2)
(*H*) ⇐ ((byte 3)(byte 2) +1)

The contents of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register *L* . The content of the next memory location is moved to register *H* .

SHLD addr (store *H* and *L* direct)

((byte 3) (byte 2)) ⇐ (*L*)
((byte 3) (byte 2) +1) ⇐ (*H*)

The content of register *L* is moved to the memory location whose address is specified in byte 2 and byte 3. The content of register *H* is moved to the next memory location.

For example:

$22_H\ 00_H\ 10_H$

This instruction moves the content of register *L* to the memory location 1000_H and moves the content of register *H* to the memory location 1001_H

LDAX ((*rp*)) (Load Accumulator Indirect)

The content of memory location, whose address is in the register pair *rp* , is moved to accumulator.

STAX rp(Store Accumulator Indirect)

$((rp)) \leftarrow (A)$

The content of register A is moved to the memory location whose address is in the register pair rp .

XCHG (Exchange *H* and *L* with *D* and *E*)

$(H) \leftrightarrow (D)$

$(L) \leftrightarrow (E)$

The contents of register *H* and *L* are exchanged with the contents of register *D* and *E* .

Arithmetic Group:

This group of instructions performs arithmetic operations on data in registers and memory.
Most of the instructions in this group affect the flag bits according to the standard rules.
All subtraction operations are performed via 2's complement arithmetic and set the carry flag to one to indicates a borrow and clear it to indicate no borrow.

ADD r (Add Register)

$(A) \leftarrow (A) + (r)$

The content of register *r* is added to the content of the accumulator. The result is placed in the accumulator.

e.g.

1 0 0 0 0 0 0 1 indicates the operation $(A) \leftarrow (A) + (C)(ADD\ C)$ because the binary coding for register C is 001.

The op-code for *ADD C* in hexadecimal is 81_H

The content of accumulator is added to the content of register *C* , and the result is stored in accumulator.

Depending on the result of the operation, the flags bits are set or reset.

If the result of the operation is zero, then the *Z* flag is set to 1. Generally it will follow the standard rule to set the flags.

The other operation in arithmetic group is listed below.

Now you are in a position to interprets the meaning of each instruction.

Addition group

Operation	Operation Performed	OP code	Flags affected
ADD r (Add register)	$(A) \leftarrow (A) + (r)$	1 0 0 0 0 *S S S*	Z, S, P, CY, AC
ADD M (Add memory)	$(A) \leftarrow (A) + ((H)(L))$	1 0 0 0 0 1 1 0	*Z, S, P, CY, AC*
ADI data (Add Immediate)	$(A) \leftarrow (A) + \text{byte}2$	1 1 0 0 0 1 1 0 (data)	*Z, S, P, CY, AC*
ADC r (Add register with carry)	$(A) \leftarrow (A) + (r) + (CY)$	1 0 0 0 1 *S S S*	*Z, S, P, CY, AC*
AMC M (Add memory with carry)	$(A) \leftarrow (A) + ((H)(L)) + (CY)$	1 0 0 0 1 1 1 0	*Z, S, P, CY, AC*
ACI data (Add Immediate with carry)	$(A) \leftarrow (A) + \text{byte}2 + (CY)$	1 1 0 0 1 1 1 0 (data)	*Z, S, P, CY, AC*

Subtraction group

Operation	Operation Performed	OP code	Flags affected
SUB r (Subtract Register)	$(A) \leftarrow (A) - (r)$	1 0 0 10 *S S S*	Z, S, P, CY, AC
SUB M (Subtract Memory)	$(A) \leftarrow (A) - ((H)(L))$	1 0 0 1 0 1 1 0	*Z, S, P, CY, AC*
SUI data (Subtract Immediate)	$(A) \leftarrow (A) - \text{byte}2$	1 1 0 1 0 1 1 0 (data)	*Z, S, P, CY, AC*
SBB r (Subtract Register with borrow)	$(A) \leftarrow (A) - (r) - (CY)$	1 0 0 1 0 *S S S*	*Z, S, P, CY, AC*
SBB M (Subtract Memory with borrow)	$(A) \leftarrow (A) - ((H)(L)) - (CY)$	1 0 0 1 1 1 1 0	*Z, S, P, CY, AC*
SBI data (Subtract immediate with borrow)	$(A) \leftarrow (A) - \text{byte}2 - (CY)$	1 1 0 1 1 1 1 0 (data)	*Z, S, P, CY, AC*

Increment/Decrement group

Operation	Operation Performed	OP code	Flags affected
INR r (Increment register)	$(r) \leftarrow (r)+(1)$	0 0 D D D 1 0 0	Z, S, P, AC
INR M (Increment memory)	$((H)(L)) \leftarrow ((H)(L))+1$	0 0 1 1 0 1 0 0	Z, S, P, AC
DCR r (Decrement register)	$(r) \leftarrow (r)-1$	0 0 D D D 1 0 1	Z, S, P, AC
DCR M (Decrement memory)	$((H)(L)) \leftarrow ((H)(L))-1$	0 0 1 1 0 1 0 1	Z, S, P, AC
INX rp (Increment register pair)	$(rh)(rl) \leftarrow (rh)(rl)+1$	0 0 R P 0 0 1 1	None
DCX rp (Decrement register pair)	$(rh)(rl) \leftarrow (rh)(rl)-1$	0 0 R P 1 0 1 1	None
DAD rp (Add register pair to H & L)	$(H)(L) \leftarrow (H)(L)+(rh)(rl)$	0 0 R P 1 0 0 1	*CY*
DAA (Decimal Adjust Accumulator)	The eight bit number in the accumulator is adjusted to form two four-bit Binary-coded decimal digits.	0 0 1 0 0 1 1 1	*Z,S,P,AC,C*

Logical group:

This group of instructions perform logical (Boolean) operations on data in registers and memory and on condition flags.

All instructions in this group affect the Zero, Sign, Parity, Auxiliary carry and carry flags according to the standard rules.

Operation	Operation Performed	Op-code
ANA r (*AND* Register)	$(A) \leftarrow (A) \wedge (r)$	1 0 1 0 0 *S S S*
ANA M (*AND* memory)	$(A) \leftarrow (A) \wedge ((H)(L))$	1 0 1 0 0 1 1 *D*
ANI data (*AND* immediate)	$(A) \leftarrow (A) \wedge \text{byte } 2$	1 1 1 0 0 1 1 0 (data)
XRA r (Exclusive *OR* Register)	$(A) \leftarrow (A) \oplus (r)$	1 0 1 0 1 *S S S*
XRA M (Exclusive *OR* memory)	$(A) \leftarrow (A) \oplus ((H)(L))$	1 0 1 0 1 1 1 0

XRI data (Exclusive *OR* immediate)	$(A) \leftarrow (A) \oplus$ byte 2	1 1 1 0 1 1 1 0 (data)
ORA r (*OR* Register)	$(A) \leftarrow (A) \vee (r)$	1 0 1 1 0 *S S S*
ORA M (*OR* memory)	$(A) \leftarrow (A) \vee ((H)(L))$	1 0 1 1 0 1 1 0
ORI data (*OR* immediate)	$(A) \leftarrow (A) \vee$ byte 2	1 1 1 1 0 1 1 0
CMP r (Compare register)	$(A) - (r)$ the accumulator remains unchanged. *Z* flag is set to 0 if $(A) = (r)$ CY flag is set to 1 if $(A) < (r)$	1 0 1 1 1 *S S S*
CMP M (Compare memory)	$(A) - ((H)(L))$	1 0 1 1 1 1 1 0

CPI data (Compare immediate)	$(A) - \text{byte } 2$	1 1 1 1 1 1 1 0 data
RLC (Rotate left)	$(A_{n+1}) \leftarrow (A_n), (A_0) \leftarrow (A_7)$ $(CY) \leftarrow (A_7)$	0 0 0 0 0 1 1 1
RRC (Rotate right)	$(A_n) \leftarrow (A_{n+1}), (A_7) \leftarrow (A_0)$ $(CY) \leftarrow A_0$	0 0 0 0 1 1 1 1
RAL (Rotate left through carry)	$(A_{n+1}) \leftarrow (A_n), (CY) \leftarrow (A_7)$ $(A_0) \leftarrow (CY)$	0 0 0 1 0 1 1 1
RAR (Rotate right through carry)	$(A_n) \leftarrow (A_{n+1}), (CY) \leftarrow (A_0)$ $(A_7) \leftarrow (CY)$	0 0 0 1 1 1 1 1
CMA (Complement accumulator)	$(A) \leftarrow (\overline{A})$	0 0 1 0 1 1 1 1
CMC (complement carry)	$(CY) \leftarrow (\overline{CY})$	0 0 1 1 1 1 1 1
STC (Set carry)	$(CY) \leftarrow 1$	0 0 1 1 0 1 1 1

Branch group :

This group of instructions alter normal sequential flow of the program.
Condition flags are not affected by any instruction in this group.

There are two types of branch instructions- unconditional and conditional.
Unconditional transfers simply load the program counter with the new address of the instruction from where the execution starts again.
Conditional transfer examine the status of one of the four processor flags to determine if the specified branch is to be executed.

The conditions that may be specified are as follows:

Condition	C C C
NZ - not zero (*Z* = 0)	0 0 0
Z - zero (*Z* =1)	0 0 1
NC - no carry (*CY* = 0)	0 1 0
C - carry (*CY* = 1)	0 1 1
PO - Parity odd (*P* = 0)	1 0 0
PE – parity even (*P* = 1)	1 0 1
P = plus (*S* = 0)	1 1 0
M – Minus (*S* = 1)	1 1 1

JMP addr (Jump)

$(PC) \leftarrow (\text{byte 3})(\text{byte 2})$

J condition addr (Conditional jump)

If (*C C C*) then

$(PC) \leftarrow (\text{byte 3})(\text{byte2})$

If the specified condition is true, control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction, otherwise control continuous sequentially.

Example:

1 1 0 0 1 0 1 1, this op-code indicates JZ, ie. Jump on zero. It will check the zero flag bit, and if the Z flag is 1, control will jump to specified address.

In hex, this op-code is CB_H

Therefore, $CB_H\ 00_H\ 20_H$, for this instruction,

It will check the *Z* flag, if *Z* flag is set to 1, then program counter will be loaded with 2000_H , otherwise control continues sequentially.

CALL addr (Call)

$((SP)-1) \leftarrow (PCH)$

$((SP)-2) \leftarrow (PCL)$

$(SP) \leftarrow (SP)-2$

$(PC) \leftarrow (\text{byte } 3)(\text{byte } 2)$

C condition addr (condition call)

If (CCC)

$((SP)-1) \leftarrow (PCH)$

$((SP)-2) \leftarrow (PCL)$

$(SP) \leftarrow (SP)-2$

$(PC) \leftarrow (\text{byte } 3)(\text{byte } 2)$

RET (Return)

$$(PCL) \leftarrow ((SP))$$
$$(PCH) \leftarrow ((SP)+1)$$
$$(SP) \leftarrow (SP)+2$$

R condition (conditional Return)

If (CCC)

$(PCL) \leftarrow ((SP))$

$(PCH) \leftarrow ((SP)+1)$

$(SP) \leftarrow (SP)+2$

***RST n* (Restart)**

$((SP)-1) \leftarrow (PCH)$

$((SP)-2) \leftarrow (PCL)$

$(SP) \leftarrow (SP)-2$

$(PC) \leftarrow 8 \times (NNN)$

Control is transferred to the instruction whose address is eight times the content of *NNN.*

PCHL (Jump *H* and *L* indirect – move *H* and *L* tp *PC*)

$(PCH) \leftarrow (H)$

$(PCL) \leftarrow (L)$

Stack, I/O and machine control group

This group of instructions performs I/O, manipulates the stack, and alters internal control flags.

PUSH rp (Push)

$$((SP)-1) \leftarrow (rh)$$
$$((SP)-2) \leftarrow (rl)$$
$$(SP) \leftarrow (SP)-2$$

PUSH PSW (Push processor status word)

$$((SP)-1) \leftarrow (A)$$
$$((SP)-2)_0 \leftarrow (CY), ((SP)-2)_1 \leftarrow X$$
$$((SP)-2)_2 \leftarrow (P), ((SP)-2)_3 \leftarrow X$$
$$((SP)-2)_4 \leftarrow (AC), ((SP)-2)_5 \leftarrow X$$
$$((SP)-2)_6 \leftarrow (Z), ((SP)-2)_7 \leftarrow (S)$$
$$(SP) \leftarrow (SP)-2$$

FLAG WORD

POP rp (Pop)

$(rl) \leftarrow ((SP))$

$(rh) \leftarrow ((SP)+1)$

$(SP) \leftarrow (SP)+2$

POP PSW (Pop processor status word)

$$(CY) \leftarrow ((SP))_0, \quad (P) \leftarrow ((SP))_2 \quad (AC) \leftarrow ((SP))_4$$
$$(Z) \leftarrow ((SP))_6 \quad (S) \leftarrow ((SP))_7$$
$$(A) \leftarrow ((SP)+1)$$
$$(SP) \leftarrow (SP)+2$$

XTHL (Exchange stack top with *H* and *L*)

$$(L) \leftrightarrow ((SP))$$
$$(H) \leftrightarrow ((SP)+1)$$

SPHL Move *HL* to *SP*

$(SP) \leftarrow (H)(L)$

IN port (Input)

$(A) \leftarrow (\text{data})$

The data placed on the eight bit bi-directional data bus by the specified port is moved to register *A.*

OUT port (output)

$(\text{data}) \leftarrow (A)$

The content of register *A* is placed on the eight bit bi-directional data bus for transmission to the specified port.

EI (Enable Interrupt)

The interrupt system is enabled following the execution of the next instruction. Interrupts are not recognised during the *EI* instruction.

DI (Disable interrupt)

The interrupt system is disabled immediately following the execution of th *D* I instruction. Interrupts are not recognized during the *DI* instruction.

HLT (Halt)

The processor is stopped.

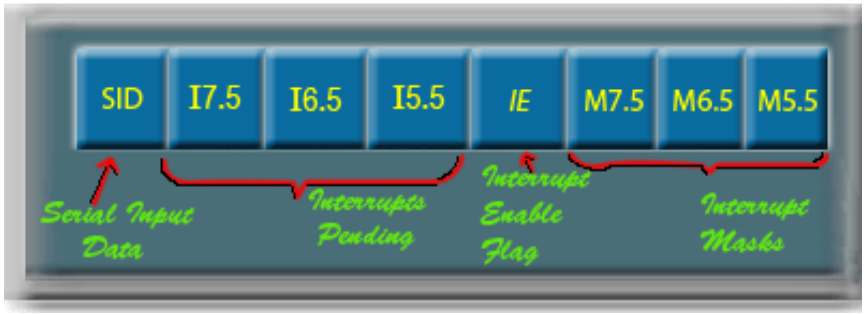
NOP (No operation)

No operation is performed. The registers and flags are unaffected.

RIM (Read Interrupt Masks)

The *RIM* instruction loads data into the accumulator relating to interrupts and serial input.

Accumulator content after *RIM*



SIM (Set interrupt masks)

The execution of *SIM* instruction uses the content of the accumulator to perform the following functions -

- Program the interrupt masks for the RST 7.5, 6.5 and 5.5 hardware interrupts
- Reset the edge-triggered RST 7.5 input latch
- Load the SOD output latch.

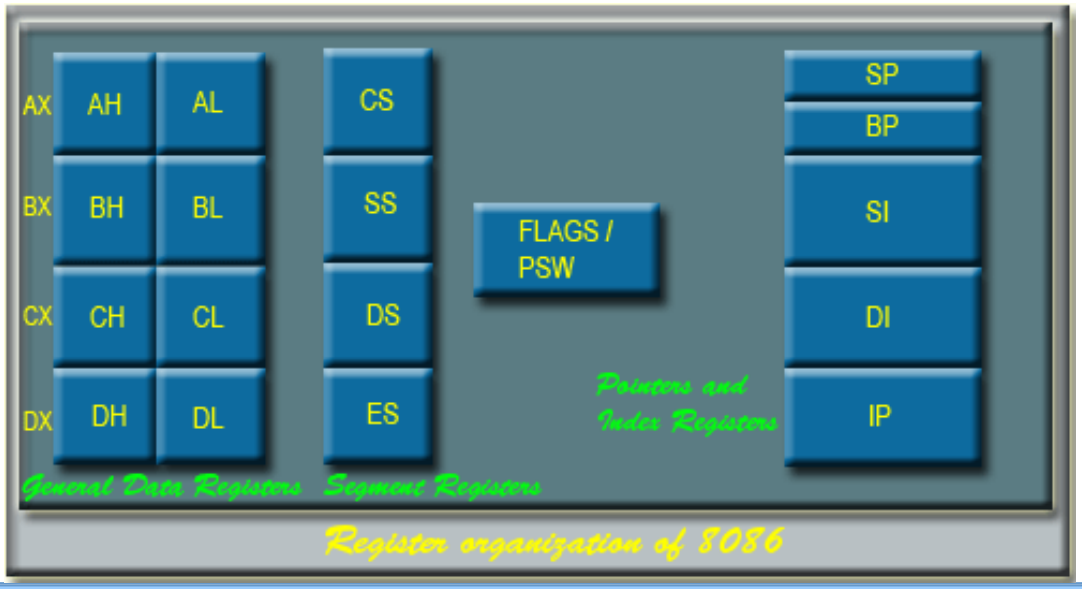
Organization of Intel 8086 Microprocessor :

Intel 8086 was the first 16-bit microprocessor introduced by Intel in 1978.

Register Organization of 8086

All the registers of 8086 are 16-bit registers. The general purpose registers can be used as either 8-bit registers or 16-bit registers.

The register set of 8086 can be categorized into 4 different groups. The register organization of 8086 is shown in the figure.



General Data Registers :

The registers *AX, BX,CX* and *DX* are the general purpose 16-bit registers.

AX is used as 16-bit accumulator. The lower 8-bit is designated as *AL* and higher 8-bit is designated as *AH* . *AL* can be used as an 8-bit accumulator for 8-bit operation.

All data register can be used as either 16 bit or 8 bit. *BX* is a 16 bit register, but *BL* indicates the lower 8-bit of *BX* and *BH* indicates the higher 8-bit of *BX* .

The register *CX* is used default counter in case of string and loop instructions.

The register *BX* is used as offset storage for forming physical address in case of certain addressing modes.

DX register is a general purpose register which may be used as an implicit operand or destination in case of a few instructions.

Segment Registers :

The 8086 architecture uses the concept of segmented memory. 8086 able to address to address a memory capacity of 1 megabyte and it is byte organized. This 1 megabyte memory is divided into 16 logical segments. Each segment contains 64 kbytes of memory.

There are four segment register in 8086

- Code Segment register (CS)
- Data Segment register (DS)
- Extra Segment register (ES)
- Stack Segment register (SS)

Code Segment Register (CS): is used for addressing memory location in the code segment of the memory, where the executable program is stored.

Data Segment Register (DS): points to the data segment of the memory where the data is stored.

Extra Segment Register (ES) : also refers to a segment in the memory which is another data segment in the memory.

Stack Segment Register (SS): is used for addressing stack segment of the memory. The stack segment is that segment of memory which is used to store stack data.

While addressing any location in the memory bank, the physical address is calculated from two parts:

- The first is segment address, the segment registers contain 16-bit segment base addresses, related to different segment.
- The second part is the offset value in that segment.

The advantage of this scheme is that in place of maintaining a 20-bit register for a physical address, the processor just maintains two 16-bit registers which is within the memory capacity of the machine.

Pointers and Index Registers .

The pointers contain offset within the particular segments.

- The pointer register *IP* contains offset within the code segment.
- The pointer register *BP* contains offset within the data segment.
- He pointer register *SP* contains offset within the stack segment.

The index registers are used as general purpose registers as well as for offset storage in case of indexed, base indexed and relative base indexed addressing modes.

The register *SI* is used to store the offset of source data in data segment.

The register *DI* is used to store the offset of destination in data or extra segment.

The index registers are particularly useful for string manipulation.

Flag Register

The 8086 flag register contents indicate the results of computation in the *ALU* . It also contains some flag bits to control the *CPU* operations.

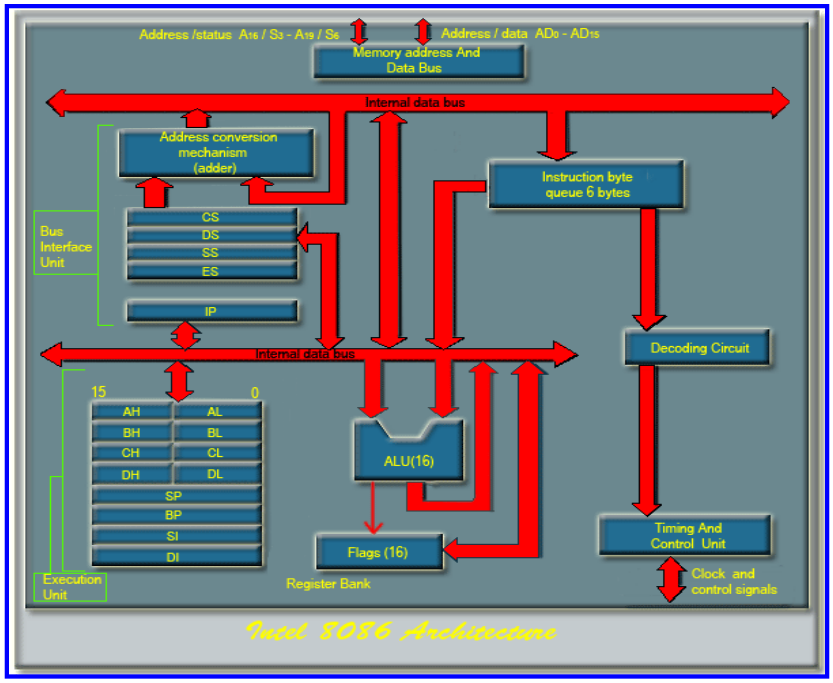
8086 Architecture :

The 8086 architecture supports -

- a 16-bit ALU.
- a set of 16 bit registers
- provides segmented memory addressing scheme
- a rich instruction set.
- Powerful interrupt structure
- Fetched instruction queue for overlapped fetching and execution step.

Organization of Intel 8086 Microprocessor

The internal block diagram units inside the 8086 microprocessor is shown in the figure.



Click on Image To View Large Image

Organization of Intel 8086 Microprocessor

The architecture of 8086 can be divided into two parts

a. Bus Interface Unit (BIU).

b. Execution Unit (EU).

The bus interface unit is responsible for physical address calculations and predecoding instruction byte queue(6 byte long).

The bus interface unit makes the system bus signals available for external devices.

The 8086 addresses a segmented memory. The complete physical address which is 20-bit long is generated using segment and offset registers, each 16-bit long.

Generating a physical address :

- The content of segment register (segment address) is shifted left bit-wise four times.
- The content of an offset register (offset address) is added to the result of the previous shift operation.

These two operations together produce a 20-bit physical address.

For example, consider the segment address is 2010 *H* and the offset address is 3535 *H* .

The physical address is calculated as:

Segment Address	2010H		0010	0000	0001	0000
Shifted left by 4 bit positions		0010	0000	0001	0000	0000
Offset address	3535H		0011	0101	0011	0101
Physical address		0010	0011	0110	0011	0101
		2	3	6	3	5H

The segment address by the segment value 2010H can have offset value from 0000 *H* to *FFFFH* within it, ie. Maximum 64K locations may be accommodated in the segment.

The physical address range for this segment is from 20100 *H* to 300*FFH* .

The segment register indicates the base address of a particular segment and *CS* , *DS* , *SS* and *ES* are used to keep the segment address.

The offset indicates the distance of the required memory location in the segment from the base address, and the offset may be the content of register *IP* , *BP* , *SI* , *DI* and *SP* .

Once the opcode is fetched and decoded, the external bus becomes free while the Execution Unit is executing the instruction.

While the fetched instruction is executed internally, the external bus is used to fetch the machine code of the next instruction and arrange it in a queue called as predecoded instruction byte queue.

This is a 6 byte long queue, works in first-in first-out policy.

While the opcode is fetched by the bus interface unit (BIU), the execution unit (EU) executes the previously decoded instruction concurrently.

Organization of Intel 8086 Microprocessor

The execution unit contains.

- the register set of 8086 except segment registers and IP.
- a 16-bit ALU to perform arithmetic & logic operation
- 16-bit flag register reflects the results of execution by the ALU.
- the decoding units decodes the op-code bytes issued from the instruction byte queue.
- the timing and control unit generates the necessary control signals to execute the instruction op-code received from the queue.

The execution unit may pass the results to the bus interface unit for storing them in memory.

Memory Segmentation :

The size of address bus of 8086 is 20 and is able to address 1 Mbytes (2^{20}) of physical memory.

The compete 1 Mbytes memory can be divided into 16 segments, each of 64 Kbytes size.

The addresses of the segment may be assigned as 00000*H* to *F*0000*H* respectively.

The offset values are from 00000*H* to *FFFFH* so that the physical address range from 00000H to FFFFFH.

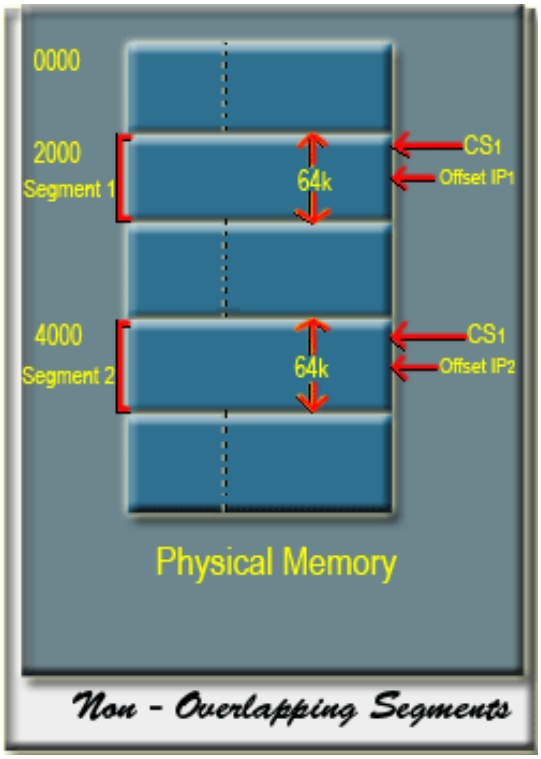
If the segmentation is done as per above mentioned way, the segments are called non-overlapping segments.

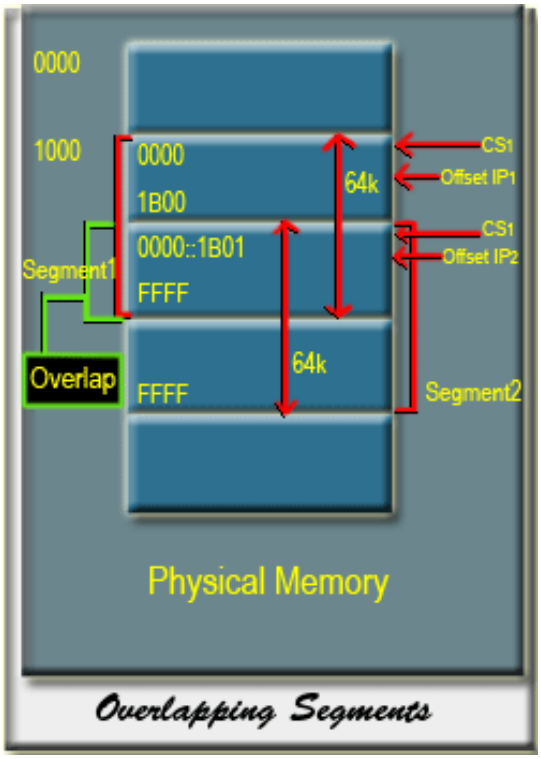
In some cases segment may overlap also. Suppose a segment starts at a particular address and its maximum size can go up to 64 Kbytes. But if another segment starts before this 64 Kbytes location of the first segment, the two segments are said to be overlapping segment.

The area of memory from the start of the second segment to the possible end of the first segment is called as overlapped segment.

Non-overlapping segments and overlapping segments are shown in the figure on the next slide .

Organization of Intel 8086 Microprocessor





The main advantages of the segmented memory scheme are as follows:

a. Allows the memory capacity to be 1 Mbyte although the actual addresses to be handled are of 16-bit size

b. Allows the placing of code data and stack portions of the same program in different parts (segments) of memory, for data and code protection.

c. Permits a program and/or its data to be put into different areas of memory each time program is executed, ie, provision for relocation may be done.

Flag Register

A 16 flag register is used in 8086. It is divided into two parts .

a. Condition code or status flags

b. Machine control flags

The condition code flag register is the lower byte of the 16-bit flag register. The condition code flag register is identical to 8085 flag register, with an additional overflow flag.

Organization of Intel 8086 Microprocessor

Print this page

<< Previous | First | Last | Next >>

The control flag register is the higher byte of the flag register. It contains three flags namely direction flag(*D*), interrupt flag (*I*) and trap flag (*T*).

The complete bit configuration of 8086 is shown in the figure.

<< Previous | First | Last | Next >>

S - Sign Flag : This flag is set, when the result of any computation is negative.

Z - Zero Flag: This flag is set, if the result of the computation or comparison performed by the previous instruction is zero.

P - Parity Flag: This flag is set to 1, if the lower byte of the result contains even number of 1's.

C - Carry Flag: This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

T - Tarp Flag: If this flag is set, the processor enters the single step execution mode.

I - Interrupt Flag: If this flag is set, the maskable interrupt are recognized by the CPU, otherwise they are ignored.

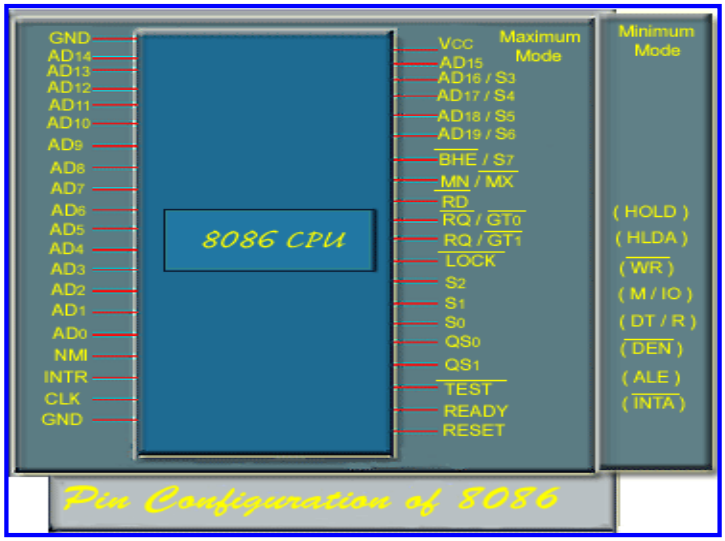
D - Direction Flag: This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto incrementing mode.

AC -Auxilary Carry Flag: This is set, if there is a carry from the lowest nibble, i.e, bit three during addition, or borrow for the lowest nibble, i.e, bit three, during subtraction.

O - Over flow Flag: This flag is set, if an overflow occurs, i.e, if the result of a signed operation is large enough to accommodate in a destination register. The result is of more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit sign operations, then the overflow will be set.

Signals of 8086

The 8086 is a 16-bit microprocessor. This microprocessor operate in single processor or multiprocessor configurations to achieve high performance. The pin configuration of 8086 is shown in the figure. Some of the pins serve a particular function in minimum mode (single processor mode) and others function in maximum mode (multiprocessor mode).



Click on Image To View Large Image

Description of signals of 8086

$AD_7 - AD_0$ -The address/ data bus lines are the multiplexed address data bus and contain the right most eight bit of memory address or data. The address and data bits are separated by using *ALE* signal.

$AD_{15} - AD_8$ -The address/data bus lines compose the upper multiplexed address/data bus. This lines contain address bit $A_{15} - A_8$ or data bus $D_{15} - D_8$. The address and data bits are separated by using *ALE* signal.

$A_{19} / S_6 - A_{18} / S_3$ - The address/status bus bits are multiplexed to provide address signals $A_{19} - A_{16}$ and also status bits $S_6 - S_3$. The address bits are separated from the status bits using the ALE signals. The status bit S_6 is always a logic 0, bit S_5 indicates the condition of the interrupt flag bit. The S_4 and S_3 combinedly indicate which segment register is presently being used for memory access.

S_4	S_3	Funtion
0	0	Extra segment
0	1	Stack segment
1	0	Code or no segment
1	1	Data Segment

$\overline{BHE}/S_7$ - The bus high enable (BHE) signal is used to indicate the transfer of data over the higher order $(D_{15} - D_8)$ data bus. It goes low for the data transfer over $D_{15} - D_8$ and is used to derive chip select of odd address memory bank or peripherals.

$\overline{BHE}$	A_0	Indication
0	0	Whole word
0	1	Upper byte from or to odd address
1	0	Lower byte from or to even address
1	1	None

$\overline{RD}$ - : Read : whenever the read signal is at logic 0, the data bus receives the data from the memory or I/O devices connected to the system

READY - This is the acknowledgement from the slow devices or memory that they have completed the data transfer operation. This signal is active high.

INTR - Interrupt Request: Interrupt request is used to request a hardware interrupt. If *INTR* is held high when interrupt enable flag is set, the 8086 enters an interrupt acknowledgement cycle after the current instruction has completed its execution.

$\overline{TEST}$ - : This input is tested by " *WAIT* " instruction. If the *TEST* input goes low; execution will continue. Else the processor remains in an idle state.

NMI - Non-maskable Interrupt: The non-maskable interrupt input is similar to *INTR* except that the *NMI* interrupt does not check for interrupt enable flag is at logic 1, i.e, *NMI* is not maskable internally by software. If *NMI* is activated, the interrupt input uses interrupt vector 2.

RESET : The reset input causes the microprocessor to reset itself. When 8086 reset, it restarts the execution from memory location *FFFF0H* . The reset signal is active high and must be active for at least four clock cycles.

CLK : Clock input: The clock input signal provides the basic timing input signal for processor and bus control operation. It is asymmetric square wave with 33% duty cycle.

VCC $+5V$ power supply for the operation of the internal circuit

*GND:*Ground for the internal circuit

$MN/\overline{MX}$: The minimum/maximum mode signal to select the mode of operation either in minimum or maximum mode configuration. Logic 1 indicates minimum mode.

Minimum mode Signals:

The following signals are for minimum mode operation of 8086.

$M/\overline{IO}$ - Memory/ *I/O* $M/\overline{IO}$ signal selects either memory operation or *I/O* operation. This line indicates that the microprocessor address bus contains either a memory address or an *I/O* port address. Signal high at this pin indicates a memory operation. This line is logically equivalent to $\overline{S_2}$ in maximum mode.

$\overline{INTA}$ - - Interrupt acknowledge: The interrupt acknowledge signal is a response to the INTR input signal. The $\overline{INTA}$ signal is normally used to gate the interrupt vector number onto the data bus in response to an interrupt request.

ALE - Address Latch Enable: This output signal indicates the availability of valid address on the address/data bus, and is connected to latch enable input of latches.

$DT/\overline{R}$ - : Data transmit/Receive: This output signal is used to decide the direction of date flow through the bi-directional buffer. $DT/\overline{R}=1$ indicates transmitting and $DT/\overline{R}=0$ indicates receiving the data.

$\overline{DEN}$ - Data Enable: Data bus enable signal indicates the availability of valid data over the address/data lines.

HOLD - The hold input request a direct memory access(DMA). If the hold signal is at logic 1, the microprocessor stops its normal execution and places its address, data and control bus at the high impedance state.

HLDA : Hold acknowledgement indicates that 8086 has entered into the hold state.

Maximum mode signal:

The following signals are for maximum mode operation of 8086.

$\overline{S_2}, \overline{S_1}, \overline{S_0}$ - Status lines: These are the status lines that reflect the type of operation being carried out by the processor.

These status lines are encoded as follows -

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Function
0	0	0	Interrupt Acknowledge
0	0	1	Read *I/O* port
0	1	0	Write *I/O* port
0	1	1	Halt
1	0	0	Code Access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	passive

$\overline{LOCK}$ - The lock output is used to lock peripherals off the system, ie, the other system bus masters will be prevented from gaining the system bus.

QS_1 and QS_0 - Queue status: The queue status bits shows the status of the internal instruction queue. The encoding of these signals is as follows

QS_1	QS_0	Function
0	0	No operation, queue is idle
0	1	First byte of opcode
1	0	Queue is empty
1	1	Subsequent byte of opcode

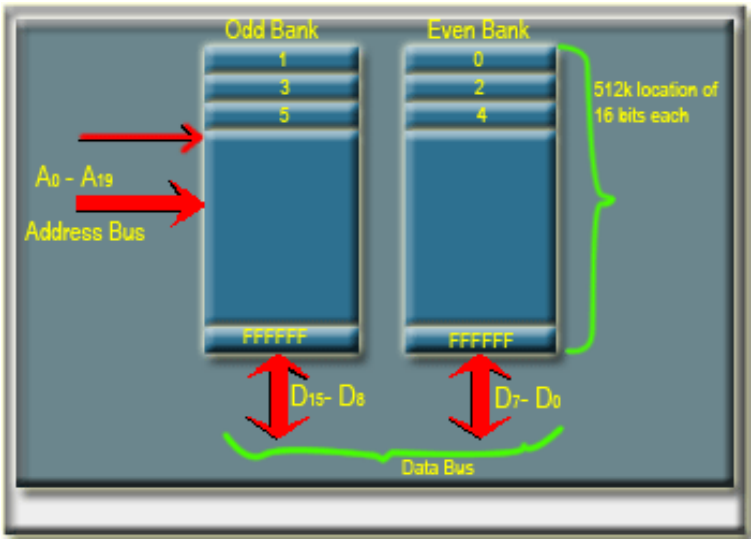
$\overline{RQ}/\overline{GT1}$ and $\overline{RQ}/\overline{GT0}$ - request/Grant: The request/grant pins are used by other local bus masters to force the processor to release the local bus at the end of the processors current bus cycle. These lines are bi-directional and are used to both request and grant a *DMA* operation. $\overline{RQ}/\overline{GT0}$ is having higher priority than $\overline{RQ}/\overline{GT1}$.

Physical Memory Organization

In an 8086 based system, the 1Mbyte memory is physically organized as odd bank and even bank, each of 512kbytes, addressed in parallel by the processor.

Byte data with even address is transferred on $D_7 - D_0$ and byte data with odd address is transferred on $D_{15} - D_8$.

The processor provides two enable signals, $\overline{BHE}$ and A_0 for selecting of either even or odd or both the banks.



$\overline{BHE}$	A_0	Function
0	0	Whole word
0	1	Upper byte/ odd address
1	0	Lower byte/even address
1	1	none

$D_{15} - D_0 / A_{15} - A_0$ are the common signal line in 8086 $AD_{15} - AD_0$

If the address bus contains $FFFFO_H$

If $\overline{BHE} = 0$, then it reads the 16 bits data from memory location $FFFFO_H$ and $FFFF1_H$.

If $\overline{BHE} = 1$, it reads the 8 bits data from memory location $FFFFO_H$.

If the address bus contains *00001H.*

If $\overline{BHE} = 0$, then it reads the 8 bits data from memory location *00001H*.

Instruction formats of 8086 :

The instruction format of 8086 has one or more number of fields associated with it.
The first filled is called operation code field or opcode field, which indicates the type of operation.
The instruction format also contains other fields known as operand fields.
There are six general formats of instructions in 8086 instruction set. The length of an instruction may vary from one byte to six bytes.

a) **One byte Instruction** : This format is only one byte long and may have the implied data or register operands. The least significant 3 bits of the opcode are used for specifying the register operand, if any. Otherwise, all the eight bits form an opcode and the operands are implied.

For example :

1 1 1 1 1 0 0 0 $F8_H$ *CLC* : clear carry

This is an operation without any operand, which clear the carry flag bit.

Depending on the register (reg = *RRR*), the contents of the specified register will be exchanged with the accumulator. This operation is having one operand which is specified in a register.

ASC || Adjust for addition *AAA* 00110111 37_H

Here the operand to this instruction is implicit and it take the contents of register *AL* .

b) **Register to Register** : This format is 2 bytes long. The first byte of the code specifies the operation code and the width of the operand specifies by *w* bit. The second byte of the opcode shows the register operands and *RIM* field.

The register represented by the *REG* field is one of the operands. The *RIM* field specifies another register or memory location, ie., the other operand. The register specified by *REG* is a source operand if $D = 0$, else it is a destination operand.

Instruction formats of 8086

For example:
MOV : data transfer operation from Register to Register.
Op-code is

10001000 11000001 88_H $C1_H$

REG = 0 0 0 indicates Register *AL*

REG = 0 0 1 indicates Register *CL*

$w = 0$ indicates it is a byte operation (8 bit)

$d = 0$ indicates *AL* is a source register.

This instruction indicates *MOV CL* , *AL* , i.e *CL* ← *AL*

C) **Register to/from memory with no displacement** : This format is also 2 bytes long and similar to the register to register format except for the *MOD* field.

The *MOD* field shows the *MOD* of addressing. In case of no displacement. *MOD* = 00

For example :

MOV : Data transfer Register/memory to/from register.

Instruction formats of 8086 Print this page

<< Previous | First | Last | Next >>

This format is similar to register to register transfer. The difference is in *MOD* field.

For register to register, *MOD* = 11

For register to/from memory with no displacement, *MOD* = 00.

When *MOD*= 0 0, the r/m fields indicates the address to memory location.

As for example *r/m* = 1 1 1 indicates (*BX)*

The instruction

1 0 0 0 1 0 1 0 0 0 0 0 0 1 1 1 indicates the instruction *MOV AX, [BX]*

In hexadecimal, the instruction is $8A_H07_H$

Here the data is present in a memory location in *DS* whose offset address is in *BX* . The effective address of the data is given as $10H \times DS + [BX]$

>> Previous | First | Last | Next >>

d) **Register to/from Memory with Displacement** : This type of instruction format contains one or two additional bytes for displacement along with 2-byte the format of the register to/from memory without displacement.

Here *MOD* fields indicates the size of displacement.

MOD = 0 1 indicates displacement of 8 bytes (instruction is of size 3 bytes)

MOD = 1 0 indicates displacement of 16 bytes. (instruction is of size 4 bytes)

Already we have seen the other two options of *MOD*

MOD = 1 1 indicates register to register transfer

MOD = 0 0 indicates memory without displacement

In this case, *R/M* fields indicates a memory when *MOD* is not 1 1

R/M = 1 1 1 indicates (*BX*)

When *MOD* = 0 1, the offset address is $(BX) + D8$

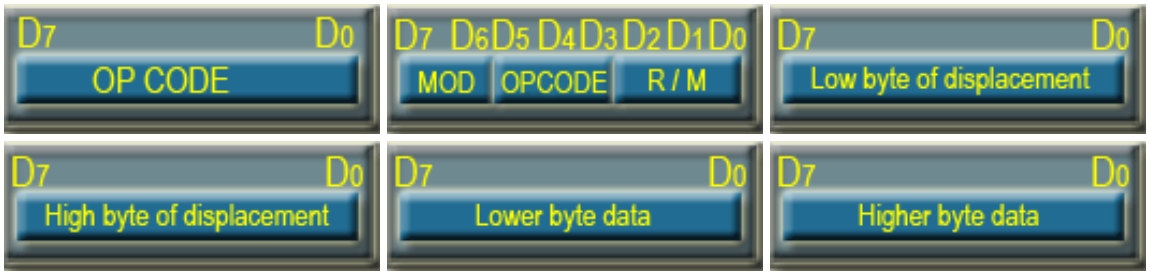
When MOD = 1 0, the offset address is $(BX) + D16$

e) **Immediate operand to register** : In this format, the first byte as well as the 3 bits from the second byte which are used for *REG* field in case of register to register format are used for opcode. It also contains one or two bytes of immediate data.

When $w = 0$, the size of immediate data is 8 bits and the size of instruction is 3 bytes.

When $w = 1$, the size of immediate data is 16 bits and the size of instruction is 4 bytes.

f) **Immediate operand to memory with 16-bit displacement** : This type of instruction format requires 5 to 6 bytes for coding. The first two bytes contain the information regarding *OPCODE* , *MOD* and *R/M* fields. The remaining 4 bytes contain 2 bytes of displacement and 2 bytes of data.



Instruction formats of 8086 Print this page

<< Previous | First | Last | Next >>

The REG code of the different registers (either a source or destination operands) in the opcode byte are assigned with binary code, which is shown below-

w bit	Register code (3 bit)	Registers	Segment register code (2 bit)	Segment register
0	0 0 0	A L	0 0	E S
0	0 0 1	*CL*	0 1	*C S*
0	0 1 0	*DL*	1 0	*S S*
0	0 1 1	*BL*	1 1	*D S*
0	1 0 0	*AH*		
0	1 0 1	*CH*		
0	1 1 0	*DH*		
0	1 1 1	*BH*		
1	0 0 0	*AX*		
1	0 0 1	*CX*		
1	0 1 0	*DX*		
1	0 1 1	BX		
1	1 0 0	*SP*		
1	1 0 1	*BP*		
1	1 1 0	*SI*		
1	1 1 1	*DI*		

Coding of different registers.

>> Previous | First | Last | Next >>

Addressing Modes of 8086 :

The different addressing modes of the 8086 instructions along with corresponding *MOD* , *REG* and *R/M* field are given in the table.

operands	Memory operands			Register operands	
	No Displacement	Displacement 8 bits	Displacement 16 bits		
MOD	0 0	0 1	1 0	11	
R/M					
0 0 0	$(BX)+(SI)$	$(BX)+(SI)+D8$	$(BX)+(SI)+D16$	AL	*AX*
0 0 1	$(BX)+(DI)$	$(BX)+(DI)+D8$	$(BX)+(DI)+D16$	*CL*	*CX*
0 1 0	$(BP)+(SI)$	$(BP)+(SI)+D8$	$(BP)+(SI)+D16$	*DL*	*DX*
0 1 1	$(BP)+(DI)$	$(BP)+(DI)+D8$	$(BP)+(DI)+D16$	*BL*	*BX*
1 0 0	(SI)	$(SI)+D8$	$(SI)+D16$	*AH*	*SP*
1 0 1	(DI)	$(DI)+D8$	$(DI)+D16$	*CH*	*BP*
1 1 0	$D16$	$(BP)+D8$	$(BP)+D16$	*DH*	*SS*
1 1 1	(BX)	$(BX)+D8$	$(BX)+D16$	*BH*	*DI*

$D8$ and $D16$ represent 8 and 16 bit displacement respectively.

The default segment for the addressing modes using *BP* and *SP* is *SS* . For all other addressing modes the default segments are *DS* or *ES* .

Addressing mode indicates a way of locating data or operands.

Different addressing modes of 8086 :

1. **Immediate** : In this addressing mode, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

ex. *MOV AX, 0050 H*

Here *0050 H* is the immediate data and it is moved to register *AX* . The immediate data may be 8-bit or 16-bit in size.

2. **Direct** : In the direct addressing mode, a 16 bit address (offset) is directly specified in the instruction as a part of it.

ex. *MOV AX* [1000 *H*]

Here data resides in a memory location in the data segment, whose effective address is $10H \times DS + 1000H$.

3. **Register** : In register addressing mode, the data is stored in a register and it is referred using the particular register. All the registers except IP may be used in this mode.

ex. *MOV AX,BX*

4. **Register Indirect** : In this addressing mode, the address of the memory location which contains data or operand is determined in an indirect way using offset registers. The offset address of data is in either *BX* or *SI* or *DI* register. The default segment register is either *DS* or *ES* .

e.g. *MOV AX,* $[BX]$

The data is present in a memory location in *DS* whose offset is in *BX* . The effective address is $10H \times DS + [BX]$.

5. **Indexed** : In this addressing mode offset of the operand is stored in one of the index register. *DS* and *ES* are the default segments for index registers *SI* and *DI* respectively

e.g. *MOV AX,* $[SI]$

The effective address of the data is $10H \times DS + [SI]$.

6. **Register Relative** : In this addressing mode the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers *BX* , *BP* , *SI* and *DI* in the default either *DS* or *ES* segment.

e.g. *MOV AX* , *50H* $[BX]$

The effective address of the data is $10H \times DS + 50H + [BX]$.

7. **Based Indexed** : In this addressing mode the effective address of the data is formed by adding the content of a base register (any one of *BX* or *BP*) to the content of an index register (any one of *SI* or *DI*). The default segment register may be *ES* or *DS* .

e.g *MOV* , $[BX][SI]$

The effective address is $10H \times DS + [BX] + [SI]$.

8. **Relative Based Indexed** : The effective address is formed by adding an 8-bit or 16-bit displacement with the sum of contents of any one of the base register (*BX* or *BP*) and any one of the index registers in a default segment.

e.g. *MOV AX , 50H* $[BX][SI]$

Here 50H is an immediate displacement. The effective address is $10H \times DS + [BX] + [SI] + 50H$.

9. **Intrasegment Direct Mode** : In this mode, the address to which the control is to be transferred lies in the segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value. The displacement is computed relative to the content of the instruction pointer *IP* .

10. **Intrasegment Indirect Mode** : This mode is similar to intrasegment direct mode except the displacement to which control is to be transferred is passed to the instruction indirectly. Here the branch address is found as the content of a register or a memory location.

11. **Intersegment Direct Mode** : In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the *CS* and *IP* of the destination address are specified directly in the instruction.

12. **Intersegment Indirect Mode** : This mode is similar to intersegment direct mode except the address to which the control is to be transferred is passed to the instruction indirectly. This information is kept in a memory block of 4 bytes: *IP* (*LSB*), *IP* (*MSB*), *CS* (*LSB*) and *CS* (*MSB*) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

Instruction set of 8086

The 8086 instructions are categorized into eight different groups.

a) **Data Transfer Instruction** : This type of instructions are used to transfer data from source operand to destination operand. All the store, load, move, exchange, input and output operations belong to this category.

b) **Arithmatic and Logical Instructions** : All the instructions performing arithmetic , logical, increment, decrement, compare and scan instructions belong to this category.

c) **Branch Instructions** : These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instructions belong to this category.

d) **Loop Instructions** : The *LOOP* , *LOOPNZ* and *LOOPZ* instructions belong to this category. These are useful to implement different loop structures.

e) **Machine control Instructions** : These instructions control the machine status. *NOP* , *HLT* , *WAIT* and *LOCK* instructions belong to this category.

f) **Flag Manipulation Instructions:** All instructions which directly affect the flag register belong to this category. The instructions *CLD* , *STD* , *CLI* , *STI,* etc. belong to this category.

g) **Shift and Rotate Instructions** : These instructions involve the bitwise shifting or rotation in either direction with or without a count in *CX* .

h) **String Instructions** : These instruction involve string manipulation operations like load, move, scan. Compare, store, etc. These instructions are only to be operated upon the string.

Data Transfer operation :

MOV **:**	move *PUSH* : Push to stack
POP **:**	pop from stack *XCHG* : Exchange
IN **:**	Input from the port *OUT* : Output to the port
XLAT **:**	Translate *LEA* : Load effective address
LDS/ LES **:**	Load pointer to *DS/ES*
LAHF **:**	Load *AH* from lower byte of Flag register
SAHF **:**	Store *AH* to lower byte of Flag register
PUSHF **:**	push Flags to stack
POPF **:**	pop Flags from stack

Arithmatic operations :

ADD **:**	add	***DAS*** **:**	Decimal Adjust After Subtraction
ADC **:**	Add with carry	***NEG*** **:**	Negative
INC **:**	Increment	***MUL*** **:**	Unsigned multiplication Byte or word.
DEC **:**	Decrement	***IMUL*** **:**	Signed multiplication
SUB **:**	Subtract	***CBW*** **:**	Convert signed Byte to Word
SBB **:**	Subtract with Borrow	***CWD*** **:**	Convert signed word to double word
CMP **:**	compare	***DIV*** **:**	Unsigned Division
AAA **:**	*ASCII* Adjust After Addition	***IDIV*** **:**	Signed Division
AAS **:**	*ASCII* Adjust After Subtraction		
AAM **:**	*ASCII* Adjust for multiplication		
AAD **:**	*ASCII* Adjust for division		
DAA **:**	Decimal Adjust Accumulator		

Logical Instructions:

AND **:**	Logical AND
OR **:**	Logical *OR*
NOT **:**	Logical Invert
XOR **:**	Logical Exclusive *OR*
TEST **:**	Logical compare Instruction
SHL/SAL **:**	Shift logical/Arithmatic left
SHR **:**	Shift logical Right
SAR **:**	Shift Arithmatic Right
ROR **:**	Rotate Right without carry
ROL **:**	Rotate Left without carry
RCR **:**	Rotate Right through carry
RCL **:**	Rotate Left through carry

String Manipulation Instructions :

PEP **:**	Repeat Instruction prefix
MOVSB / MOVSW **:**	Move string Byte or string word.
CMPS **:**	Compare sting Byte or String word.
SCAS **:**	Scan string Byte or string word
LODS **:**	Load string Byte or String word
STOS **:**	Store String Byte or String word

Control transfer or Branching Instructions :

A) Unconditional Branch Instructions.

CALL **:**	Unconditional Call
RET **:**	Return from the procedure
INT N **:**	Interrupt type *N*
INTO **:**	Interrupt on overflow
JMP **:**	Unconditional Jump
IRET **:**	Return from ISR
LOOP **:**	Loop unconditionally

B) Conditional Branch Instruction:

***JZ* Label :**	Transfer control to 'Labe' if $ZF = 1$
***JNZ* Label :**	Transfer control to 'Labe' if $ZF = 0$
***JS* Label :**	Transfer control to 'Label' if $SF = 1$
***JNS* Label :**	Transfer control to 'Label' if $SF = 0$
***JO* Label :**	Transfer control to 'Label' if $OF = 1$
***JNO* Label :**	Transfer control to 'Label' if $OF = 0$
***JP* Label :**	Transfer control to 'Label' if $PF = 1$
***JNP* Label :**	Transfer control to 'Label' if $PF = 0$

***JB* Label :**	Transfer control to 'Label' if $CF = 1$
***JNB* Label :**	Transfer control to 'Label' if $CF = 0$
***JBE* Label :**	Transfer control to 'Labe'' if $CF = 1$ or $ZF = 1$
***JNBE* Label :**	Transfer control to 'Labe' if $CF = 0$ or $ZF = 0$
***JL* Label :**	Transfer control to 'Labe' if $SF = 1$ or $OF = 1$
***JNL* Label :**	Transfer control to 'Labe' if $SF = 0$ or $OF = 0$
***JLE* Label :**	Transfer control to 'Labe' if $ZF = 1$ or neither *SF* nor *OF* is 1
***JNLE* Label :**	Transfer control to 'Labe' if $ZF = 0$ or at least any one of *SF* and *OF* is 1
***LOOPZ* label :**	Loop through a sequence of instructions from 'Labe' while $ZF = 1$ and $CX \neq 0$.
***LOOPNZ* label :**	Loop through a sequence of instructions from 'Labe' while $ZF = 0$ and $CX \neq 0$.

Flag manipulator and Processor control instructions :

CLC **:**	Clear carry flag
CMC **:**	Complement carry flag
STC **:**	Set carry flag
CLD **:**	Clear direction flag
STD **:**	Set direction flag
CLI **:**	Clear interrupt flag
STI **:**	Set Interrupt flag
WAIT **:**	wait for test input pin to go low
HLT **:**	Halt the processor
NOP **:**	No operation
ESC **:**	Escape to external device like NDP (numeric co-processor)
LOCK **:**	

Computer Programming :

From the discussion of microprocessor 8085 and 8086, it is clear that all the instructions are nothing but the combination of 0s and 1s. ‘0' indicates a low signal (may be voltage 0V) and ‘1' indicates a high signal (may be voltage 5V).

On the other hand all the computers work on Von Neuman stored program principle. We have to store the computer program, which is nothing but the set of instruction and computer executes them one after another as per the program requirement.

We have to store the program (i.e. set of instructions) in computer memory. The starting address of the program must be specified while executing the program and it is loaded into the program counter. After that program counter keeps track of the execution of the program till the end of the program.

Every program must be end with a stop or terminating instruction, otherwise the control unit will keep on fetching information from memory. Once it encounters a halt or stop instruction, execution stops.

Simple Example :

Consider that we want to add two numbers 75 and 97.
The first requirement is to keep this two numbers in memory.
Next we have to write a program to fetch this two numbers from memory to CPU and add this two numbers. Finally the result has to be stored in memory.

Again we have to load the program in memory. We must know the memory address of the first instruction where we have stored the program.

While executing the program counter must be loaded with the starting address of the program. The control unit will generate appropriate signal to perform the required task.

Consider that we are using Intel 8085 microprocessor to solve this problem. The task required to perform this operation:

- Get the two numbers from memory to general purpose register.
- Perform the addition operation
- Store the result back into the memory.

Finer details with respect to 8085:

Assume that we store number 75 in memory location $1000H$ and 97 in memory location *1001H*.The result will be stored in memory location *1002H*.

One possible solution :

Step 1: First load the content of memory location $1000H$ to accumulator.

$LDA\ 1000H$ $3A\ 00\ 10$ (machine instruction in hexadecimal)

The format of this instruction is opcode Low-order address high order address opcode of LDA is 00 11 10 10 i.e. $3A_H$.

Step 2: Move the contents of accumulator to register *B* .

$MOV\ B\ A$.

The format of this instruction is

0 1	**D D D**	**S S S**
	Destination register	Source register

i.e., 0 1 0 0 0 1 1 1 *47H* (Machine instruction in hexadecimal)

Step 3: Load the content of memory location *1001H* to accumulator

LDA 1001*H* 3*A* 01 10 (Machine instruction in hexadecimal)

Step 4: Add the content of register *B* to accumulator and store the result in accumulator.

ADD B

The format of this instruction is

1 0 0 0 0 S S S

Source Register

i.e. 1 0 0 0 0 0 0 0 80_H (Machine instruction in hexadecimal)

Step 5: Store the result that is present in accumulator to memory location 1002*H*

STA 1002*H* 3 2 0 2 1 0 (Machine instruction in hexadecimal)

opcode Lower order address higher order address

opcode of *STA* is 0 0 1 1 0 0 1 0 i.e.

32_H

Step 6 : *HALT* : to indicate the end of program
Opcode of HALT 0 1 1 1 0 1 1 0 $76H$

Therefore to carry out this addition, we have to perform these five operation.

In this example we assume that data are available in memory and storing the result in memory.

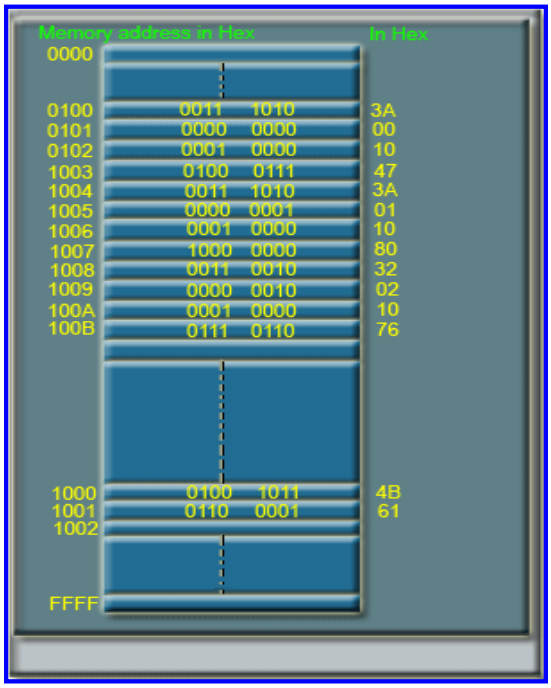
But if we want to take the input from some input device (like key board), first we have to accept the input from keyboard and stored in some memory location. Similarly to display the result in monitor, we have to get the result from memory and display in monitor with the help of some io instruction.

The complete program is :

Memory location

Memory Location	**Machine Code**	**opcode**
0100*H*	3 *A* 00 10	LDA *1000H*
0103*H*	47	MOV B, A
0104*H*	3 *A* 01 10	*LDA 1001H*
0107*H*	80	ADD B
0108*H*	32 02 10	STA *1002H*
010*BH*	76	HLT

If we store this program from memory location $100H$, then the contents of memory is shown below:



Click on Image To View Large Image

While executing the program, the program counter (PC) will be loaded with the starting memory address of this program, i.e. 0100*H* .

The processor will start execution this program starting from memory location 0100*H* and it keeps on doing the execution job till it encounters a halt instruction.

If we program the microprocessor in this way, i.e writing the machine code directly, it is known as machine language programming.

The main advantage of machine language programming is that the memory control is directly in the hands of the programmer, so that, he/she may able to manage the memory of the system more efficiently.

The disadvantages of machine language programming are more prominent.

The programming, coding and resource management techniques are tedious. The programmer has to take care of al these functions.

The programs are difficult to understand unless one has a thorough technical knowledge of the processor architecture and the instruction set.

Also it is difficult to remember the machine code of each and every instruction,.

Assembly Language programming:

The assembly language programming is simpler as compared to the machine language programming.

The instruction mnemonics are directly used in the assembly language programming. There is an one-to –one correspondence between instruction mnemonics and machine instruction.

An assembler is used to translate the assembly language programming to machine code.

The program written in assembly language programming is more readable than machine language programming.

The main improvement in assembly language over machine language is that the address value and the contents can be identified by labels.

Programming of microprocessor Print this page

<< Previous | First | Last | Next >>

The assembly language instruction sequence for 8085 microprocessor of the previously discussed program will look like:

```
LDA  1000H      // Load accumulator from memory address 1000H
MOV  B,A        // Move the content from accumulator to B
LDA  1001H      // Load accumulator from memory address 1001H
ADD  B          // Add the content of B to accumulator and store the result in accumulator
STA 1002H       // Store the content of accumulator in memory location 1002 H
HLT             // Halt
```

>> Previous | First | Last | Next >>

Another assembly language instruction sequence to solve the same problem.

```
MVI  L,00H      // Load lower order byte of address 00H to register L
MVI  H,10H      // Load higher order byte of address 10H to register H
MVI  B,M        // Move the contents of memory location addressed by H - L pair to B
LDA 01H 10H     // Load the accumulator from memory location 1001H
ADD B           // Add the content of B to accumulator
STA  02H 10H    // Store the content of accumulator to memory location 1002H
HLT             // Halt.
```

The *H*-*L* register pair is loaded with 1000 *H* which is the address of first input. This is done in first two instruction.

Moving the contents of memory location 1000 *H* to the register *B* . It is done in third instruction.

Next we are loading the accumulator from the memory location 1001 *H* , which is the second input. This is done in fourth instruction.

Te content of register *B* is added with the content of accumulator and store the result in accumulator. It is done in fifth instruction.

The content of accumulator is stored in memory location 1002 *H* . It is done in sixth instruction.
Seventh instruction is to halt the processor, i.e, to stop the program execution.

The first two instructions can be replaced

```
LXI    H,1000H
```

Assembly language instruction sequence for the same program for 8086 microprocessor.
Consider that data segment starts from 20000*H* and code segment starts from *10000H*.

```
MOV  CX,2000H        // Initialize DS at 2000H
MOV  DS,CX           // Move the content of CS to DS
MOV  AX,[1000H]      // Get first operand in AX
MOV  BX,[1001H]      // Get second operand in BX
ADD  AX,BX           // Perform addition
MOV  [1002H],AX      // Store the result in location 1002H
HLT                  // Stop
```

Since the immediate data cannot be loaded into a segment register, the data is transferred to one of the general purpose register, i.e. *CX* , and then the register content is moved to the segment register *DS* .

The data segment register contains 2000*H* .

The effective address of the operands are 2000:1001 i.e. 21000*H* and 21001*H*

The result is stored in memory location 21002*H*.

Programming of microprocessor Print this page

<< Previous | First | Last | Next >>

Problem : Find out the largest number from an unordered array of sixteen 8 bit numbers stored sequentially in the memory locations starting at 0200*H*.
Assembly language instruction sequence for 8085 microprocessor.

	$MVI \quad B,OFH$	// put OFH in register B to count the number of input elements
	$LXI \quad H,0200H$	// Load the register pair H-L by 0200*H* which is address of the first element.
	$MOV \quad A,M$	// The content of memory location pointed by H-L register pair is moved to accumulator.
BACK:	*INXH*	// Increment the register pair H-L to get the next element
	$CMP \quad M$	//The content of the memory location whose address is in H-L pair is subtracted from the accumulator. The accumulator remain unchanged. $CY = 1$ if $A < \left((H)(L)\right)$
	$JNC \quad NEXT$	// Jump to the level 'NEXT' if the carry is not set, i.e, content of accumulator is biggest so far.
	$MOV \quad A,M$	// if carry is set, the content of the memory addressed by (*H* - *L*)pair is biggest so far, so putting it in accumulator.
NEXT :	$DCR \quad B$	// Decrement the content *B* , to indicate we have checked one more element
	$JNZ \quad BACK$	// jump on not zero, content of *B* indicates numbers of elements to read, and repeat the process.
	HLT	// Finishes the input and stop.

>> Previous | First | Last | Next >>

Programming of microprocessor Print this page

<< Previous | First | Last | Next >>

Assembly language instruction sequence for 8086 microprocessor.
Here assume that the data segment stands from memory location 10000*H*.

```
        MOV  CX, OFH      // initialize counter for number of iteration
        MOV  AX,1000H     // initialize data segment
        MOV  DS,AX
        MOV  SI  0200H    // initialize source pointer
        MOV  AX,[SI]      // take first number in AX
BACK:   INC  SI           // increment source pointer.
        CMP  AX,[SI]      // compare next number with the previous
        JNC  NEXT         // If the next number is smaller, jump to NEXT .
        MOV  A×[SI]       // If the next number is bigger, replace the previous one with
                          the next element
NEXT :  LOOP BACK         // Repeat the procedure for 15 times.
        HLT
```

>> Previous | First | Last | Next >>

CMP operation subtract the source operand from the destination operand, but does not store the result any where. The carry flag is set if the source operand is greater than the destination operand.

LOOP : This instruction executes the part of the program from the label in the instruction to the loop instruction, *CX* number of times. At each iteration *CX* is decremented automatically. This instruction basically implements decrement counter and jump if not zero structure.

Example: Write a program to move a string of data bytes from offset 1000*H* to 2000*H*. The length of the string is *FFH* .

Assembly language instruction sequence for 8085 microprocessor:

```
        LXI H, 1000H
        LXI D, 2000H
        MVI C, FFH
LOOP:   MOV  A, M
        STAX  D          // The content of register A is moved to the
                         memory location whose address is in register pair
                         D-E
        INX  H
        INX  D
        DCR  C
        JNZ  LOOP        // Jump on not zero, i.e. the result of decrement is
                         not zero.
        HLT
```

Assembly language instruction sequence for 8086 microprocessor.
In 8086, we have to define the data segment by setting the *DS* register. Assume that *DS* is set appropriately

```
        MOV SI, 1000H
        MOV DI, 2000H
        MOV CX, FFH
LOOP:   MOV AX,[SI]
        MOV [DI],AX
        INC SI
        INC DI
        DEC CX
        JNZ LOOP
        HLT
```

The program listing is similar to the program of 8085, every instruction of 8085 is replaced by an equivalent instruction of 8086.
Therefore, the above program listing is correct. But this is not an efficient implementation for 8086, because we have not used any advance feature of 8086.

Alternate program listing for 8086 microprocessor

Assume that the data segment register and extra segment registers are set appropriately.

```
        MOV SI, 1000H
        MOV DI, 2000H
        MOV CX, FFH
        CLD
REP     MOVSB
        HLT
```

CLD instruction clears the direction flag.

If the direction flag bit is '0', the string is processed in auto increment mode.

REP : This instruction is used as a prefix to other instructions. The instruction to which the *REP* prefix is provided, is executed repeatedly until the *CX* register becomes zero. At each iteration *CX* is decremented by one automatically. When *CX* becomes zero, the execution proceeds to the next instruction in sequence.

MOVSB : A string of bytes stored in a set of consecutive memory locations is moved to another set of destination locations.