DIGITAL LOGIC AND COMPUTER ORGANIZATION

II B.TECH I SEMESTER - CSE (AR 23)

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

LENDI INSTITUTE OF ENGINEERING AND TECHNOLOGY

(An Autonomous Institute, Approved by AICTE & Permanently Affiliated to JNTU-GV, Vizianagaram)

(Accredited By NAAC with A Grade and Accredited by NBA)

Jonnada (Village), Denkada (Mandal), Vizianagaram District – 535 005

Phone No. 08922-241111, 241112

E-Mail: lendi_2008@yahoo.com website: www.lendi.org

UNIT – III

Introduction to Programmable Logic Devices (PLDs): PLA, PAL, PROM, Realization of Switching Functions using PROM, PAL and PLA.

Basic Structure of Computers: Functional unit, Basic Operational concepts, Bus structures, System Software, Performance, Register Transfer Notation, Assembly Language Notation, Basic Instruction Types.

INDEX

UNIT – III
(A) Introduction to Programmable Logic Devices (PLDs)

Introduction to Programmable Logic Devices (PLDs): PROM, PLA, PAL, Realization of Switching Functions using PROM, PAL and PLA.

INTRODUCTION

A memory unit is a device to which binary information is transferred for storage and from which information is retrieved when needed for processing. There are **two types of memories** that are used in digital systems: random-access memory **(RAM)** and read-only memory **(ROM)**.

ROM is a programmable logic device (PLD). The binary information that is stored within such a device is specified in some fashion and then embedded within the hardware in a process is referred to as programming the device. The word "**programming**" here refers to a **hardware procedure which specifies the bits that are inserted** into the hardware configuration of the device.

ROM is one example of a PLD. Other such units are the programmable logic array (PLA), programmable array logic (PAL), and the field-programmable gate array (FPGA). **A PLD is an integrated circuit with internal logic gates connected through electronic paths that behave similarly to fuses.** In the original state of the device, all the fuses are intact. Programming the device involves blowing those fuses along the paths that must be removed in order to obtain the particular configuration of the desired logic function.

A typical PLD may have hundreds to millions of gates interconnected through hundreds to thousands of internal paths. The below figure shows the conventional and array logic symbols for a multiple input OR gate. Instead of having multiple input lines into the gate, we draw a single line entering the gate. The input lines are drawn perpendicular to this single line and are connected to the gate through internal fuses.

Figure: Conventional and array logic diagrams for OR gate

The term "programmable" means changing either hardware or software configuration of an internal logic and interconnects. The configuration of the internal logic is done by the user. PROM, EPROM, PAL, PLA are some of the examples of Programmable Logic devices.

- Programmable Read Only Memory (PROM) - a fixed array of AND gates and a programmable array of OR gates.
- Programmable Array Logic (PAL) - a programmable array of AND gates feeding a fixed array of OR gates.
- Programmable Logic Array (PLA) - a programmable array of AND gates feeding a programmable array of OR gates.

READ-ONLY MEMORY (ROM)

A read-only memory (ROM) is essentially a memory device in which permanent binary information is stored. The binary information must be specified by the designer and is then embedded in the unit to form the required interconnection pattern. Once the pattern is established, it stays within the unit even when power is turned off and on again.

A block diagram of a ROM consisting of k inputs and n outputs is shown in below figure. The inputs provide the address for memory, and the outputs give the data bits of the stored word that is selected by the address. The number of words in a ROM is determined from the fact that k address input lines are needed to specify 2^k words. Note that ROM does not have data inputs, because it does not have a write operation. Integrated circuit ROM chips have one or more enable inputs and sometimes come with three-state outputs to facilitate the construction of large arrays of ROM.

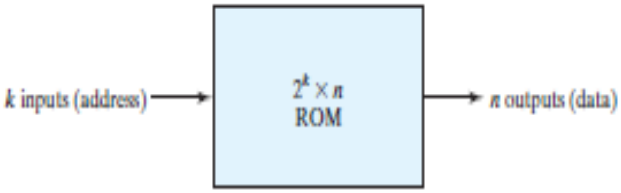


Figure: ROM block diagram

Consider, for example, a 32 X 8 ROM. The unit consists of 32 words of 8 bits each. There are five input lines that form the binary numbers from 0 through 31 for the address. Below figure shows the internal logic construction of this ROM. The five inputs are decoded into 32 distinct outputs by means of a 5 X 32 decoder. Each output of the decoder represents a memory address. The 32 outputs of the decoder are connected to each of the eight OR gates. Each OR gate must be considered as having 32 inputs. Each output of the decoder is connected to one of the inputs of each OR gate. Since each OR gate has 32 input connections and there are 8 OR gates, the ROM contains 32 X 8 = 256 internal connections. In general, **a 2^k X n ROM will have an internal k X 2^k decoder and n OR gates.** Each OR gate has 2^k inputs, which are connected to each of the outputs of the decoder.

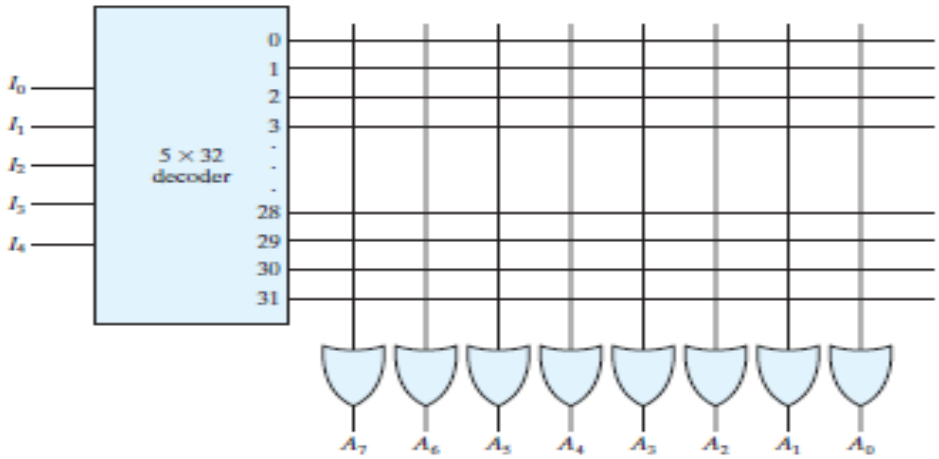


Figure: Internal logic of a 32 * 8 ROM

The above 256 intersections are programmable. A programmable connection between two lines is logically equivalent to a switch that can be altered to be either closed (meaning that the two lines are connected) or open (meaning that the two lines are disconnected). The

programmable intersection between two lines is sometimes called a **cross point**. Various physical devices are used to implement cross point switches. One of the simplest technologies employs a **fuse** that normally connects the two points, but is **opened or "blown**" by the application of a **high-voltage pulse into the fuse**.

The internal binary storage of a ROM is specified by a truth table that shows the word content in each address. For example, the content of a 32 X 8 ROM may be specified with a truth table similar to the one shown in below Table.

Table: 32 X 8ROM Truth Table (Partial)

Inputs					Outputs							
I_4	I_3	I_2	I_1	I_0	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	1
0	0	0	1	0	1	1	0	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0
		⋮							⋮			
1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	1	1	0	0	0	1	0
1	1	1	1	0	0	1	0	0	1	0	1	0
1	1	1	1	1	0	0	1	1	0	0	1	1

The hardware procedure that programs the ROM blows fuse links in accordance with a given truth table. For example, programming the ROM according to the above truth table results in the configuration shown in below figure. Every **0** listed in the truth table specifies the **absence of a connection**, and every **1** listed specifies a **path that is obtained by a connection.**

For example, the table specifies the eight-bit word 10110010 for permanent storage at address 3. The four 0's in the word are programmed by blowing the fuse links between output 3 of the decoder and the inputs of the OR gates associated with outputs A_6, A_3, A_2, and A_0. The four 1's in the word are marked with a '**x**' to denote a **temporary connection**, in place of a **dot used for a permanent connection in logic** diagrams. When the input of the ROM is 00011, all the outputs of the decoder are 0 except for output 3, which is at logic 1. The signal equivalent to logic 1 at decoder output 3 propagates through the connections to the OR gate outputs of A_7, A_5, A_4, and A_1. The other four outputs remain at 0. The result is that the stored word 10110010 is applied to the eight data outputs.

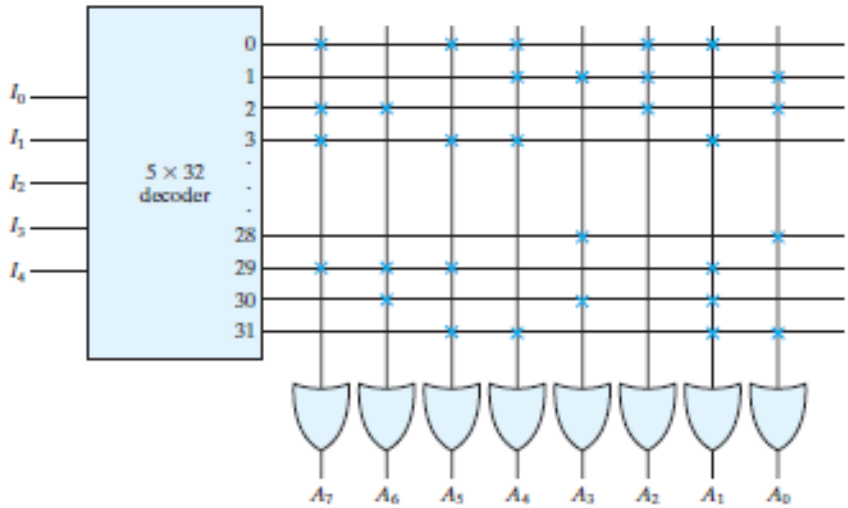
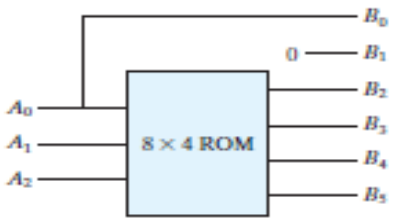


Figure: Programming the ROM according to above mentioned table

Example: Design a combinational circuit using a ROM. The circuit accepts a three-bit number and outputs a binary number equal to the square of the input number.

Solution: The first step is to derive the truth table of the combinational circuit. The below table is the truth table for the combinational circuit with three inputs and six outputs are needed to accommodate all possible binary numbers. We note that output **B_0 is always equal to input A_0**, so there is no need to generate B_0 with a ROM, since it is equal to an input variable. Moreover, output **B_1 is always 0**, so this output is a known constant. We actually need to generate only four outputs with the ROM; the other two are readily obtained. **The minimum size of ROM needed must have three inputs and four outputs.** Three inputs specify eight words, so the ROM must be of size 8 X 4. The ROM implementation is shown below. The three inputs specify eight words of four bits each.

Inputs			Outputs						
A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0	Decimal
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	1	0	0	0	1	49



(a) Block diagram

A_2	A_1	A_0	B_5	B_4	B_3	B_2
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	1	0	0

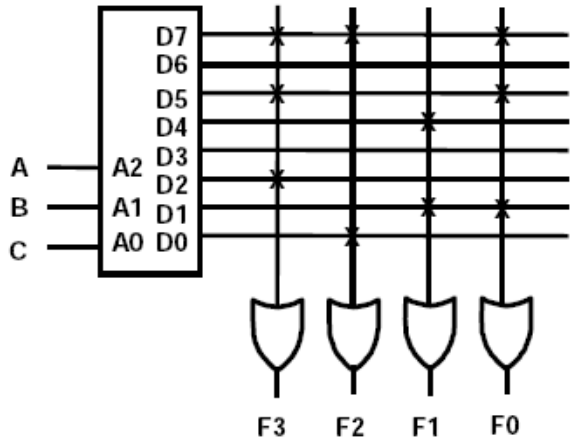
(b) ROM truth table

Example: Implement the following functions using PROM

$F_0 = \Sigma\, m\,(1, 5, 7)$
$F_1 = \Sigma\, m\,(1, 4)$
$F_2 = \Sigma\, m\,(0, 7)$
$F_3 = \Sigma\, m\,(2, 5, 7)$

Solution: The fixed "AND" array is a "decoder" with 3 inputs and 8 outputs implementing minterms which is shown below:

A 8 X 4 PROM (N = 3 input lines, M= 4 output lines)



Types of ROMs

The required paths in a ROM may be programmed in four different ways. The **first** is called **mask programming and is done by the semiconductor company** during the last fabrication process of the unit. The mask programming is economical only if a large quantity of the same ROM configuration is to be ordered.

For small quantities, it is more economical to use a **second** type of ROM called **programmable read-only memory, or PROM.** When ordered, PROM units **contain all the fuses intact,** giving all 1's in the bits of the stored words. The fuses in the PROM are blown by the application of a high-voltage pulse to the device through a special pin. A **blown fuse defines a binary 0 state and an intact fuse gives a binary 1 state.** The hardware procedure for programming **ROMs or PROMs is irreversible**, and **once programmed, the fixed pattern is permanent and cannot be altered**. Once a bit pattern has been established, the unit must be discarded if the bit pattern is to be changed.

A **third** type of ROM is the **erasable PROM**, or EPROM, which can be restructured to the initial state even though it has been programmed previously. When the EPROM is placed under a **special ultraviolet light for a given length of time**, the shortwave radiation discharges the internal floating gates that serve as the programmed connections. After erasure, the EPROM returns to its initial state and can be reprogrammed to a new set of values.

The **fourth** type of ROM is the **electrically erasable PROM** (EEPROM or E2PROM). This device is like the EPROM, except that the previously programmed connections can be **erased with an electrical signal instead of ultraviolet light. The advantage is that the device can be erased without removing it from its socket.**

Combinational PLDs

There are three major types of combinational PLDs, differing in the placement of the programmable connections in the AND–OR array. The below figure shows the configuration of the three PLDs. The **PROM has a fixed AND array constructed as a decoder and a programmable OR array.** The programmable OR gates implement the Boolean functions in sum-of-minterms form. The **PAL has a programmable AND array and a fixed OR array**. The AND gates are programmed to provide the product terms for the Boolean functions, which are logically summed in each OR gate. The most flexible PLD is the **PLA, in which both the AND and OR arrays can be programmed.** The product terms in the AND array may be shared by any OR gate to provide the required sum-of-products implementation.

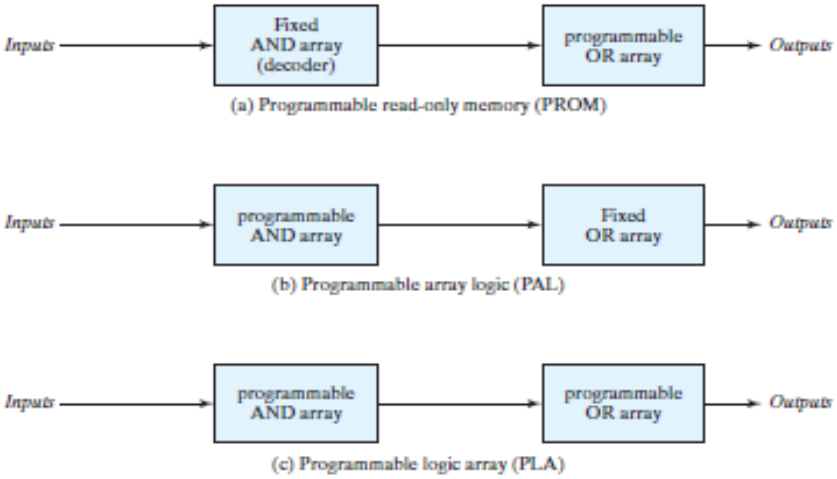


Figure: Basic configuration of three PLDs

PROGRAMMABLE LOGIC ARRAY (PLA)

The PLA is similar in concept to the PROM, except that the **PLA does not provide full decoding of the variables and does not generate all the minterms. The decoder is replaced by an array of AND gates that can be programmed to generate any product term of the input variables. The product terms are then connected to OR gates to provide the sum of products for the required Boolean functions.**

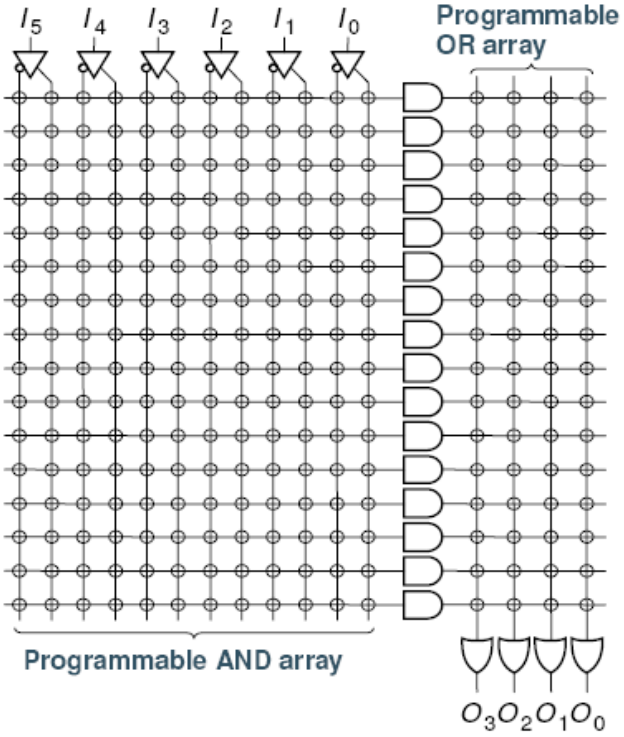


Figure: PLA Structure

The internal logic of a PLA with three inputs and two outputs is shown in the below figure. Such a circuit is too small to be useful commercially, but is presented here to demonstrate the typical logic configuration of a PLA. The diagram uses the array logic graphic symbols for complex circuits. Each input goes through a buffer–inverter combination, shown in the diagram with a composite graphic symbol, that has both the true and complement outputs. Each input and its complement are connected to the inputs of each AND gate, as indicated by the intersections between the vertical and horizontal lines. The outputs of the AND gates are connected to the inputs of each OR gate. The output of the OR gate goes to an XOR gate, where the other input can be programmed to receive a signal equal to either logic 1 or logic 0. The output is inverted when the XOR input is connected to 1 (since $x \oplus 1 = x'$). The output does not change when the XOR input is connected to 0 (since $x \oplus 0 = x$). The particular Boolean functions implemented in the PLA of below figure are

F1 = AB’ + AC + A’BC’
F2 = (AC + BC)’

The product terms generated in each AND gate are listed along the output of the gate in the diagram. The product term is determined from the inputs whose cross points are connected and marked with a (**x**). The output of an OR gate gives the logical sum of the selected product

terms. The output may be complemented or left in its true form, depending on the logic being realized.

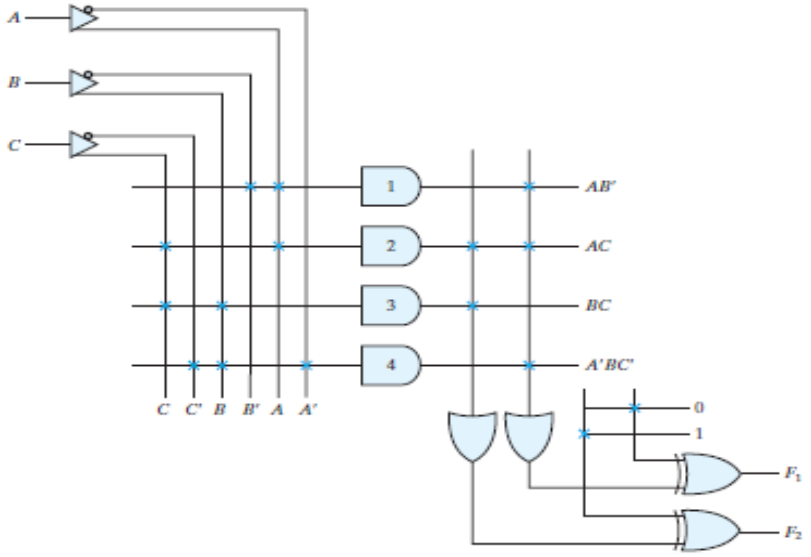


Figure: PLA with three inputs, four product terms, and two outputs

The **fuse map of a PLA** can be specified in a tabular form. For example, the programming table that specifies the PLA of above figure is listed in below table. The PLA programming table consists of three sections. The **first section lists the product terms** numerically. The **second section specifies the required paths between inputs and AND gates**. The **third section specifies the paths between the AND and OR gates.** For each output variable, we may have a T (for true) or C (for complement) for programming the XOR gate. The product terms listed on the left are not part of the table; they are included for reference only. For each product term, the inputs are marked with 1, 0, or — (dash). If a variable in the product term appears in the form in which it is true, the corresponding input variable is marked with a 1. If it appears complemented, the corresponding input variable is marked with a 0. **If the variable is absent from the product term, it is marked with a dash.**

Table: PLA Programming Table

		Inputs			Outputs (T)	(C)
	Product Term	A	B	C	F_1	F_2
AB'	1	1	0	—	1	—
AC	2	1	—	1	1	1
BC	3	—	1	1	—	1
A'BC'	4	0	1	0	1	—

The paths between the inputs and the AND gates are specified under the column head "Inputs" in the programming table. A 1 in the input column specifies a connection from the input variable to the AND gate. A 0 in the input column specifies a connection from the complement of the variable to the input of the AND gate. A dash specifies a blown fuse in both the input variable and its complement. It is assumed that an open terminal in the input of an AND gate behaves like a 1.

The paths between the AND and OR gates are specified under the column head "Outputs." The output variables are marked with 1's for those product terms which are included in the function. Each product term that has a 1 in the output column requires a path from the output of the AND gate to the input of the OR gate. Those marked with a dash specify a blown fuse. It is assumed that an open terminal in the input of an OR gate behaves like a 0. Finally, a T (true) output dictates that the other input of the corresponding XOR gate be connected to 0, and a C (complement) specifies a connection to 1.

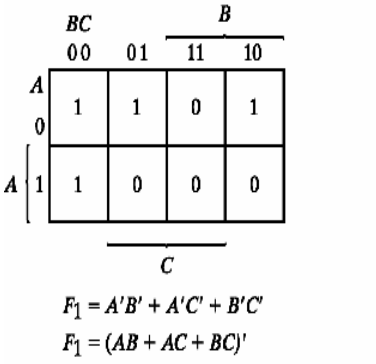
The size of a PLA is specified by the number of inputs, the number of product terms, and the number of outputs. **For n inputs, k product terms, and m outputs, the internal logic of the PLA consists of n buffer–inverter gates, k AND gates, m OR gates, and m XOR gates. There are 2n * k connections between the inputs and the AND array, k * m connections between the AND and OR arrays, and m connections associated with the XOR gates**.

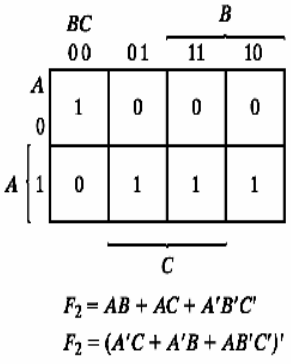
Example: Implement the following two Boolean functions with a PLA:

$F_1 (A, B, C) = \Sigma (0, 1, 2, 4)$

$F_2 (A, B, C) = \Sigma (0, 5, 6, 7)$

Solution: The two functions are simplified in the maps given below. Both the true value and the complement of the functions are simplified into sum-of-products form. The combination that gives the minimum number of product terms is $F_1 = (AB + AC + BC)'$ and $F_2 = AB + AC + A'B'C'$. This combination gives four distinct product terms: AB, AC, BC, and A'B'C'. The PLA programming table for the combination is shown below. Note that output F_1 is the true output, even though a C is marked over it in the table. This is because F_1 is generated with an AND–OR circuit and is available at the output of the OR gate. The XOR gate complements the function to produce the true F_1 output.



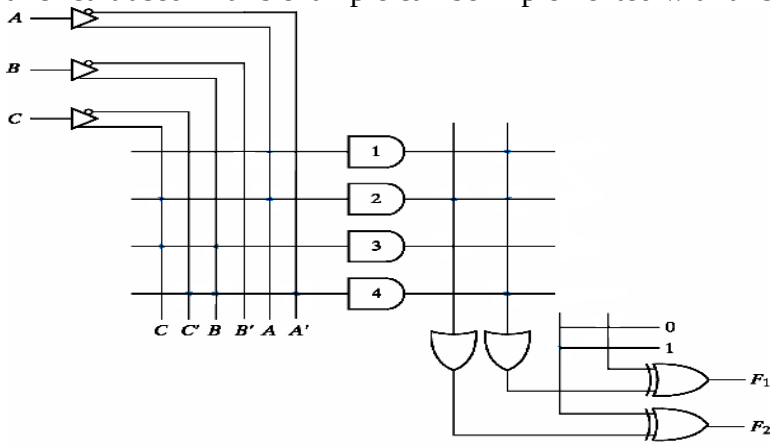


PLA programming table

Product term	Inputs A	B	C	Outputs (C) F_1	(T) F_2
1	1	1	–	1	1
2	1	–	1	1	1
3	–	1	1	1	–
4	0	0	0	–	1

AB
AC
BC
A'B'C'

The combinational circuit used in this example can be implemented with this PLA.



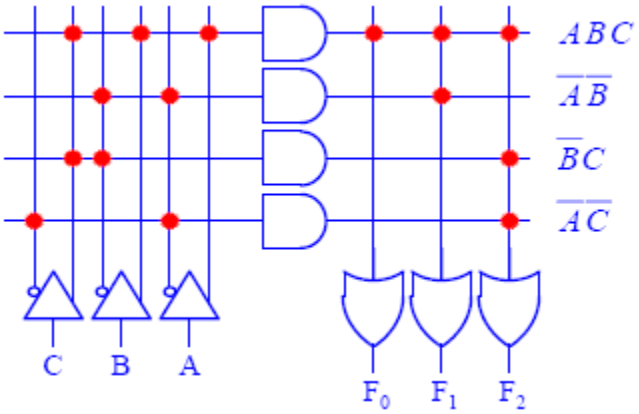
Example: Implement the following functions using PLA

$F_0 = ABC$

$F_1 = ABC + A'B'$

$F_2 = ABC + B'C + A'C'$

Solution:



PROGRAMMABLE ARRAY LOGIC (PAL)

The PAL is a programmable logic device with a **fixed OR array and a programmable AND array.** Because only the AND gates are programmable, the PAL is easier to program than, but is not as flexible as, the PLA.

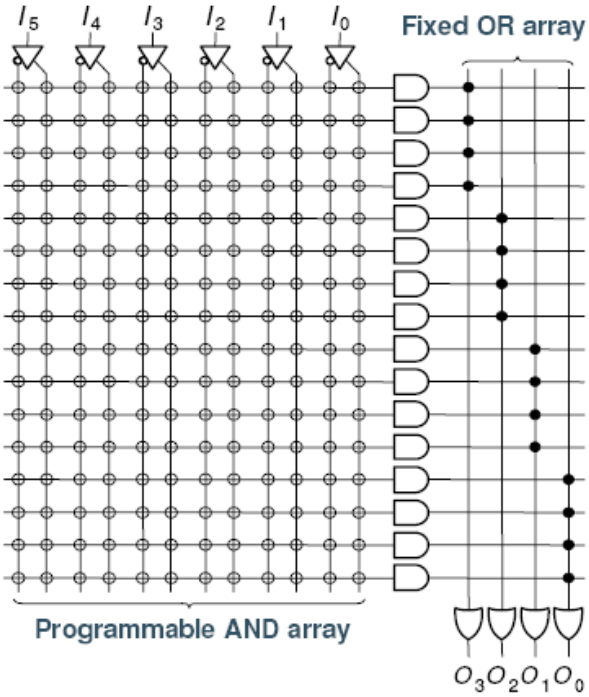


Figure: PAL Structure

The below figure shows the logic configuration of a typical PAL with four inputs and four outputs. Each input has a buffer–inverter gate, and each output is generated by a fixed OR gate. There are four sections in the unit, each composed of an AND–OR array that is three wide, the term used to indicate that there are three programmable AND gates in each section

and one fixed OR gate. Each AND gate has 10 programmable input connections, shown in the diagram by 10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple-input configuration of the AND gate. One of the outputs is connected to a buffer–inverter gate and then fed back into two inputs of the AND gates.

Commercial PAL devices contain more gates than the one shown below. A typical PAL integrated circuit may have eight inputs, eight outputs, and eight sections, each consisting of an eight-wide AND–OR array. The output terminals are sometimes driven by three-state buffers or inverters.

In designing with a PAL, the Boolean functions must be simplified to fit into each section. **Unlike the situation with a PLA, a product term cannot be shared among two or more OR gates.** Therefore, each function can be simplified by itself, without regard to common product terms. The number of product terms in each section is fixed, and if the number of terms in the function is too large, it may be necessary to use two sections to implement one Boolean function.

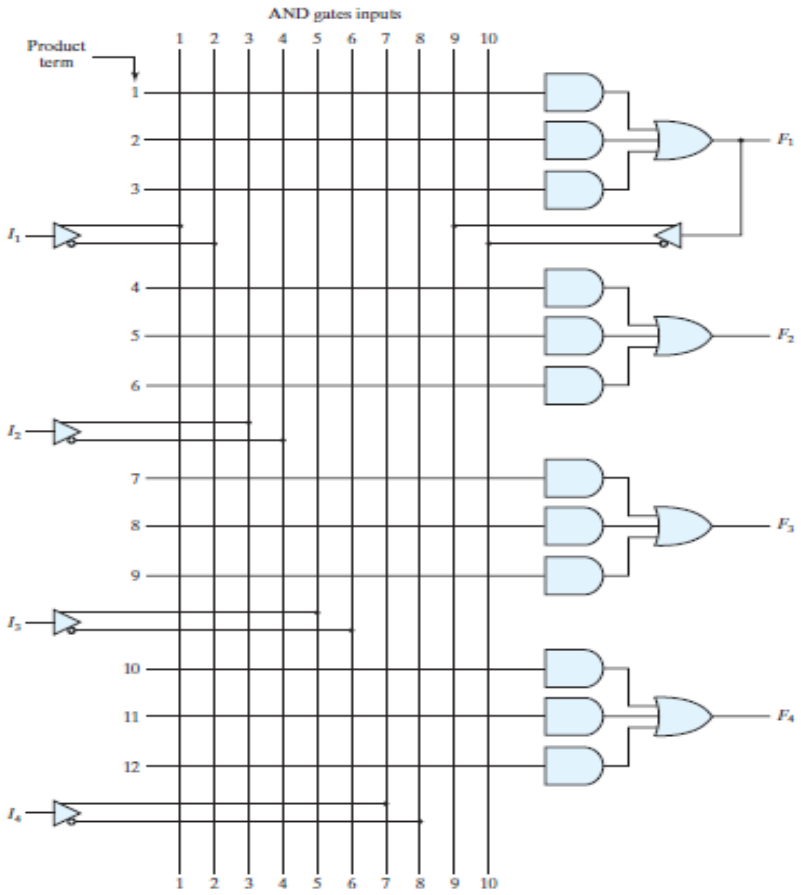


Figure: PAL with four inputs, four outputs, and a three-wide AND–OR structure

Example: Consider the following Boolean functions, given in sum-of-minterms form, design of a combinational circuit using a PAL,

w (A, B, C, D) = Σ(2, 12, 13)
x (A, B, C, D) = Σ (7, 8, 9, 10, 11, 12, 13, 14, 15)
y (A, B, C, D) = Σ (0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)
z (A, B, C, D) = Σ (1, 2, 8, 12, 13)

Solution: Simplifying the four functions to a minimum number of terms results in the following Boolean functions:

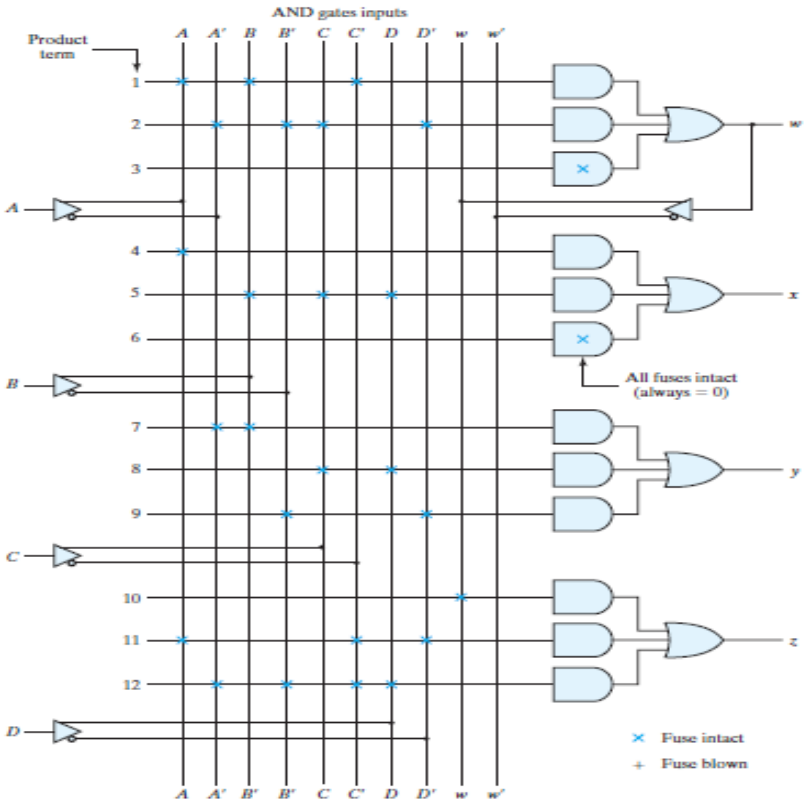
w = ABC’ + A’B’CD’
x = A + BCD
y = A’B + CD + B’D’
z = ABC’ + A’B’CD’ + AC’D’ + A’B’C’D = w + AC’D’ + A’B’C’D

Note that the function for z has four product terms. The logical sum of two of these terms is equal to w. By using w, it is possible to reduce the number of terms for z from four to three. The below table lists the PAL programming table for the four Boolean functions. The table is divided into four sections with three product terms in each, to conform to the PAL of below figure. The first two sections need only two product terms to implement the Boolean function. The last section, for output z, needs four product terms. Using the output from w, we can reduce the function to three terms.

Table: PAL Programming Table

Product Term	AND Inputs A	B	C	D	W	Outputs
1	1	1	0	–	–	$w = ABC'$
2	0	0	1	0	–	$+ A'B'CD'$
3	–	–	–	–	–	
4	1	–	–	–	–	$x = A$
5	–	1	1	1	–	$+ BCD$
6	–	–	–	–	–	
7	0	1	–	–	–	$y = A'B$
8	–	–	1	1	–	$+ CD$
9	–	0	–	0	–	$+ B'D'$
10	–	–	–	–	1	$z = w$
11	1	–	0	0	–	$+ AC'D'$
12	0	0	0	1	–	$+ A'B'C'D$

The fuse map for the PAL as specified in the programming table is shown in below figure.



Example: Implement the following functions using PAL.

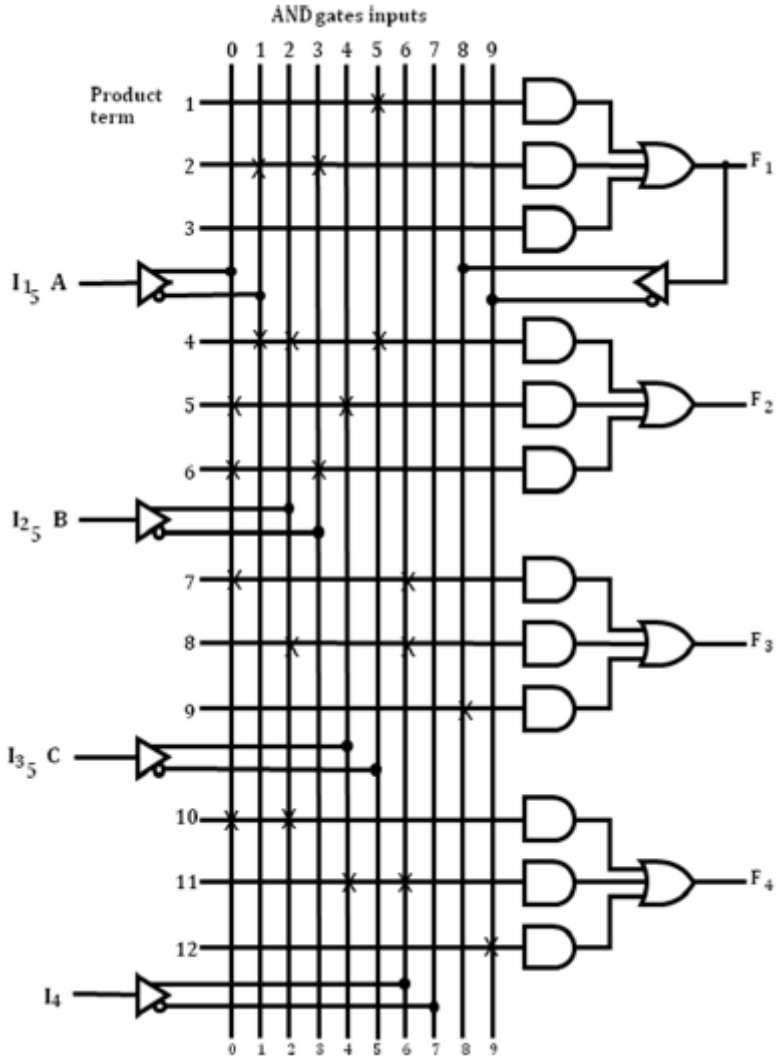
F1 = A'B' + C'

F2 = A'BC' + AC + AB'

F3 = A'I4 + BI_4 + F1

F4 = AB+CI_4+F1'

Solution:



Comparison of PROM, PLA and PAL

PROM	PLA	PAL
AND array is fixed and OR array is programmable.	Both AND and OR arrays are programmable.	OR array is fixed and AND array is programmable.
Cheaper and simple to use.	Costliest and complex than PAL and PROMs,	Cheaper and simpler.
All minterms are decoded.	AND array can be programmed to get desired minterms.	AND array can be programmed to get desired minterms.
Only Boolean functions in Standard SOP form can be implemented using PROM	Any Boolean functions in SOP form can be implemented using PLA.	Any Boolean functions In SOP form can be implemented using PAL.

(B) BASIC STRUCTURE OF COMPUTERS

Functional unit, Basic Operational concepts, Bus structures, System Software, Performance, Register Transfer Notation, Assembly Language Notation, Basic Instruction Types.

INTRODUCTION TO COMPUTER ARCHITECTURE AND ORGANIZATION

Computer Architecture is a functional description of requirements and design implementation for the various parts of a computer. It deals with the functional behavior of computer systems. It comes before the computer organization while designing a computer. Computer architecture and computer organization are related but distinct concepts in the field of computer science.

Computer architecture refers to the design of the internal workings of a computer system, including the CPU, memory, and other hardware components. It involves decisions about the organization of the hardware, such as the instruction set architecture, the data path design, and the control unit design. Computer architecture is concerned with optimizing the performance of a computer system and ensuring that it can execute instructions quickly and efficiently.

On the other hand, **computer organization** refers to the operational units and their interconnections that implement the architecture specification. It deals with how the components of a computer system are arranged and how they interact to perform the required operations. Computer organization is concerned with the physical implementation of the architecture design and includes decisions about the interconnection and communication between components, such as the bus structure, memory hierarchy, and input/output systems.

In summary, **computer architecture** is focused on the design of the internal workings of a computer system, while computer organization is focused on the implementation of that design. Computer architecture is concerned with the high-level design decisions, while computer organization deals with the low-level implementation details. **Architecture describes what the computer does.**

Computer Organization comes after the decision of Computer Architecture first. Computer Organization is how operational attributes are linked together and contribute to realizing the architectural specification. Computer Organization deals with a structural relationship. **The organization describes how it does it.**

FUNCTIONAL UNIT

A computer consists of five functionally independent main parts: **input, memory, arithmetic and logic, output, and control units,** as shown in figure below.

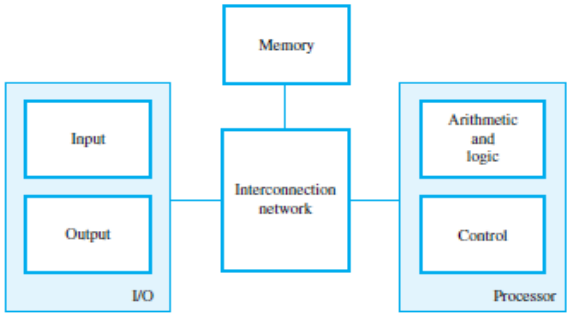


Figure: Basic functional units of a computer

The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines. The information received is stored in the computer's memory, either for later use or to be processed immediately by the arithmetic and logic unit. The processing steps are specified by a program that is also stored in the memory. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit. An interconnection network provides the means for the functional units to exchange information and coordinate their actions. We refer to the arithmetic and logic circuits, in conjunction with the main control circuits, as the **processor**.

Input and output equipment is often collectively referred to as the input-output (I/O) unit. The information handled by a computer is convenient to categorize this information as either instructions or data. Instructions are explicit commands that

- Govern the transfer of information within a computer as well as between the computer and its I/O devices.
- Specify the arithmetic and logic operations to be performed.

A program is a list of instructions which performs a task. Programs are stored in the memory. The processor fetches the program instructions from the memory, one after another, and performs the desired operations. The computer is controlled by the stored program, except for possible external interruption by an operator or by I/O devices connected to it. Data are numbers and characters that are used as operands by the instructions. Data are also stored in the memory.

Input Unit

Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor.

Many other kinds of input devices for human-computer interaction are available, including the **touchpad, mouse, joystick, and trackball**. These are often used as graphic input devices in conjunction with displays. **Microphones** can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, **cameras** can be used to capture video input.

Digital communication facilities, such as the Internet, can also provide input to a computer from other computers and database servers.

Memory Unit

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary Memory

Primary memory, also called **main memory**, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed. The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually. Instead, they are handled in groups of fixed size called **words**. The memory is organized so that one word can be stored or retrieved in one basic operation. The number of bits in each word is referred to as the **word length** of the computer, typically 16, 32, or 64 bits.

To provide easy access to any word in the memory, a distinct **address** is associated with each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. A particular word is accessed by specifying its address and issuing a control command to the memory that starts the storage or retrieval process.

Instructions and data can be written into or read from the memory under the control of the processor. It is essential to be able to access any word location in the memory as quickly as possible. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a **random-access memory** (RAM). The time required to access one word is called the **memory access time**. This time is independent of the location of the word being accessed. It typically ranges from a few nanoseconds (ns) to about 100 ns for current RAM units.

Cache Memory

As an adjunct to the main memory, **a smaller, faster RAM unit,** called a **cache**, is used to hold sections of a program that are currently being executed, along with any associated data. The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip. The purpose of the cache is to facilitate high instruction execution rates.

At the start of program execution, the cache is empty. All program instructions and any required data are stored in the main memory. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an instruction requires data located in the main memory, the data are fetched and copies are also placed in the cache.

Now, suppose a number of instructions are executed repeatedly as happens in a program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. Similarly, if the same data locations are accessed repeatedly while copies of their contents are available in the cache, they can be fetched quickly.

Secondary Storage

Although **primary memory** is essential, it tends to be **expensive** and does not retain information when power is turned off. Thus additional, less expensive, permanent **secondary storage** is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. Access times for secondary storage are longer than for primary memory. A wide selection of secondary storage devices is available, including **magnetic disks, optical disks (DVD and CD), and flash memory devices**.

Arithmetic and Logic Unit

Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication, division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. For example, if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use.

When operands are brought into the processor, they are stored in high-speed storage elements called **registers.** Each register can store one word of data. Access times to registers are even shorter than access times to the cache unit on the processor chip.

Output Unit

The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. A familiar example of such a device is a **printer**. Most printers employ either photocopying techniques, as in laser printers, or ink jet streams. Such printers may generate output at speeds of 20 or more pages per minute. However, printers are mechanical devices, and as such are quite slow compared to the electronic speed of a processor.

Some units, such as graphic displays, provide both an output function, showing text and graphics, and an input function, through touchscreen capability. The dual role of such units is the reason for using the single name **input/output** (I/O) unit in many cases.

Control Unit

The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve centre that sends control signals to other units and senses their states.

I/O transfers, consisting of input and output operations, are controlled by program instructions that identify the devices involved and the information to be transferred. Control circuits are responsible for generating the *timing signals* that govern the transfers and determine when a given action is to take place. Data transfers between the processor and the memory are also managed by the control unit through timing signals. It is reasonable to think of a control unit as a well-defined, physically separate unit that interacts with other parts of the computer. Much of the control circuitry is physically distributed throughout the computer. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

The operation of a computer can be summarized as follows:

- The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
- Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed.
- Processed information leaves the computer through an output unit.
- All activities in the computer are directed by the control unit.

BASIC OPERATIONAL CONCEPTS

The activity in a computer is governed by instructions. To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be used as instruction operands are also stored in the memory.
A typical instruction might be

Load R2, LOC

This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2. The original contents of location LOC are preserved, whereas those of register R2 are overwritten. Execution of this instruction requires several steps. First, the instruction is fetched from the memory into the processor. Next, the operation to be performed is determined by the control unit. The operand at LOC is then fetched from the memory into the processor. Finally, the operand is stored in register R2.

After operands have been loaded from memory into processor registers, arithmetic or logic operations can be performed on them. For example, the instruction

Add R4, R2, R3

adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum.

After completing the desired operations, the results are in processor registers. They can be transferred to the memory using instructions such as

Store R4, LOC

This instruction copies the operand in register R4 to memory location LOC. The original contents of location LOC are overwritten, but those of R4 are preserved.

For Load and Store instructions, transfers between the memory and the processor are initiated by sending the address of the desired memory location to the memory unit and asserting the appropriate control signals. The data are then transferred to or from the memory.

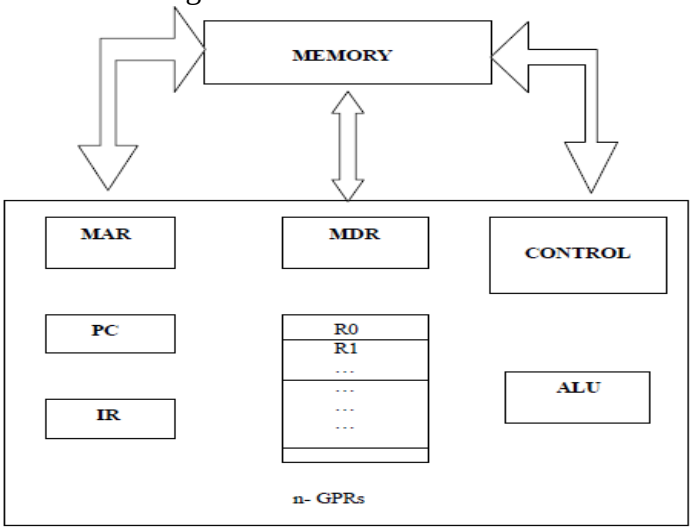


Figure: Connection between processor and main memory

This figure shows how the memory and the processor can be connected. It also shows some components of the processor. In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The **instruction register** (IR) **holds the instruction that is currently being executed**. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction. The **program counter** (PC) is another specialized register. It **contains the memory address of the next instruction to be fetched and executed.** During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed. It is customary to say that the PC points to the next instruction that is to be fetched from the memory. In addition to the IR and PC, processor also contains **general-purpose registers** R0 through Rn−1, often called **processor registers**. They serve a variety of functions, including holding operands that have been loaded from the memory for processing.

The other two registers which facilitate communication with memory are: -

1. MAR – (Memory Address Register): It holds the address of the location to be accessed.

2. MDR – (Memory Data Register): It contains the data to be written into or read out of the address location.

Operating steps are

1. Programs reside in the memory and usually get these through the input unit.
2. Execution of the program starts when the PC is set to point at the first instruction of the program.
3. Contents of PC are transferred to MAR and a Read Control Signal is sent to the memory.
4. After the time required to access the memory elapses, the address word is read out of the memory and loaded into the MDR.
5. Now contents of MDR are transferred to the IR and now the instruction is ready to be decoded and executed.
6. If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.
7. An operand in the memory is fetched by sending its address to MAR and initiating a read cycle.
8. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU.
9. After one or two such repeated cycles, the ALU can perform the desired operation.
10. If the result of this operation is to be stored in the memory, the result is sent to MDR.
11. Address of location where the result is stored is sent to MAR and a write cycle is initiated.
12. The contents of PC are incremented so that PC points to the next instruction that is to be executed.

Normal execution of a program may be pre-empted (temporarily interrupted) if some devices require urgent servicing, to do this one device raises an Interrupt signal. An interrupt is a request signal from an I/O device for service by the processor. The processor provides the requested service by executing an appropriate interrupt service routine. The diversion may change the internal stage of the processor its state must be saved in the memory location before interruption. When the interrupt-routine service is completed the state of the processor is restored so that the interrupted program may continue.

BUS STRUCTURES

The bus shown below is a simple structure that implements the interconnection network. Only one source/destination pair of units can use this bus to transfer data at any one time.

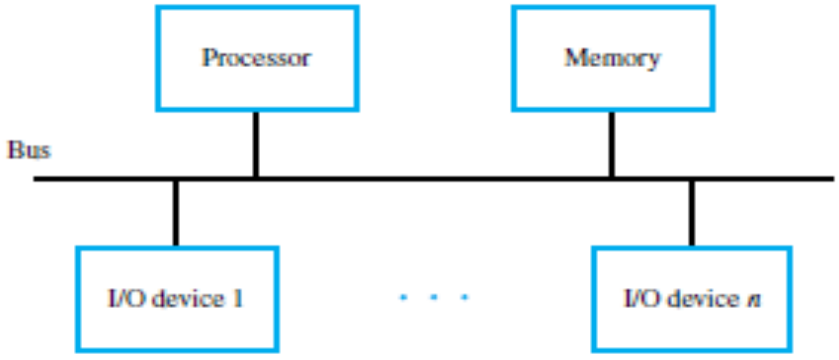


Figure: A single bus structure

The bus consists of three sets of lines used to carry address, data, and control signals. I/O device interfaces are connected to these lines, as shown below for an input device.

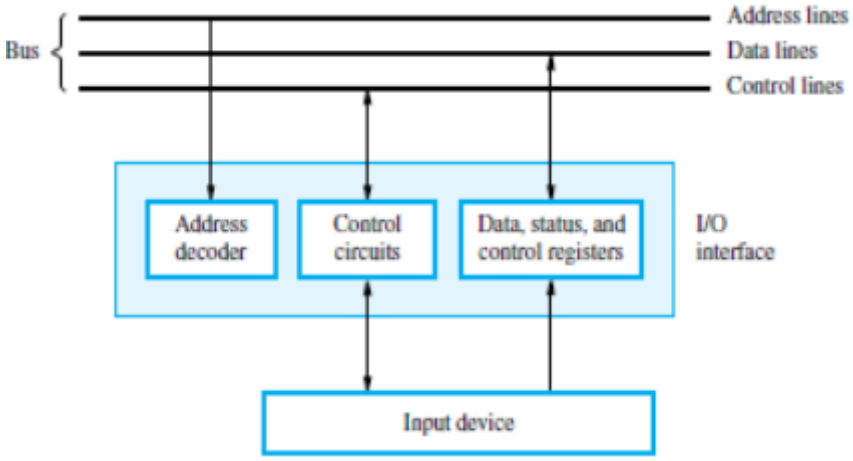


Figure: I/O interface for an input device

Each I/O device is assigned a unique set of addresses for the registers in its interface. When the processor places a particular address on the address lines, it is examined by the address decoders of all devices on the bus. The device that recognizes this address responds to the commands issued on the control lines. The processor uses the control lines to request either a Read or a Write operation, and the requested data are transferred over the data lines.

When I/O devices and the memory share the same address space, the arrangement is called **memory-mapped I/O**. Any machine instruction that can access memory can be used to transfer data to or from an I/O device.

Single bus structure is

- Low cost
- Very flexible for attaching peripheral devices

Multiple bus structure certainly increases, the performance but also increases the cost significantly.

All the interconnected devices are not of same speed and time, leads to a bit of a problem. This is solved by using cache registers (i.e. buffer registers). These buffers are electronic registers of small capacity when compared to the main memory but of comparable speed.

The instructions from the processor at once are loaded into these buffers and then the complete transfer of data at a fast rate will take place.

SYSTEM SOFTWARE

If a user wants to enter and run an application program, he/she needs a System Software. **System software** is a collection of programs that are executed as needed to perform functions such as:

1. Receiving and interpreting user commands
2. Entering and editing application programs and storing then as files in secondary storage devices
3. Managing the storage and retrieval of files in secondary storage devices
4. Running standard application programs such as word processors, spread sheets, games, with data supplied by the user

5. Controlling I/O units to receive input information and produce output results
6. Translating programs from source form prepared by the user into object form consisting of machine instructions
7. Linking and running user- written application programs with existing standard library routines, such as numerical computation packages.

Various components of system software:

Compiler: A system software program which translates the high level language program into a suitable machine language program.

Text editor: Another important system program that all programmers use is a text editor.

- It is used for entering and editing application programs.
- The user of this program interactively executes commands that allow statements of a source program entered at a keyboard to be accumulated in a file.

Operating system (OS): It is a key system software component.

- This is a large program, or actually a collection of routines, that is used to control the sharing of and interaction among various computer units as they execute application programs.
- The OS routines perform the tasks required to assign computer resources to individual application programs.
- These tasks include assigning memory to program and data files, moving data between memory and disk units, and handling I/O operations.

Types of software:

A layer structure showing where Operating System is located on generally used software systems on desktops

System software: System software helps run the computer hardware and computer system. It includes a combination of the following:

- device drivers
- operating systems
- servers
- utilities
- windowing systems
- compilers
- debuggers
- interpreters
- linkers

The purpose of systems software is to unburden the applications programmer from the often complex details of the particular computer being used, including such accessories as communications devices, printers, device readers, displays and keyboards, and also to partition the computer's resources such as memory and processor time in a safe and stable manner. Examples are- Windows XP, Linux and Mac.

Application software: Application software allows end users to accomplish one or more specific (not directly computer development related) tasks. Typical applications include:

- Business software
- Computer games
- Quantum chemistry and solid state physics software
- Telecommunications (i.e. internet and everything that flows on it)
- Databases
- Educational software
- Medical software
- Military software
- Molecular modeling software
- Image editing
- Spreadsheet
- Simulation software
- Word processing
- Decision making software

Application software exists for and has impacted a wide variety of topics.

PERFORMANCE

Performance of computer:

The most important measure of the performance of a computer is how quickly it can execute programs. The speed with which a computer executes program is affected by the design of its hardware. For best performance, it is necessary to design the compiler, the machine instruction set, and the hardware in a coordinated way.

The total time required to execute the program is elapsed time is a measure of the performance of the entire computer system. It is affected by the speed of the processor, the disk and the printer. The time needed to execute an instruction is called the processor time.

Just as the elapsed time for the execution of a program depends on all units in a computer system, the processor time depends on the hardware involved in the execution of individual machine instructions. This hardware comprises the processor and the memory which are usually connected by the bus.

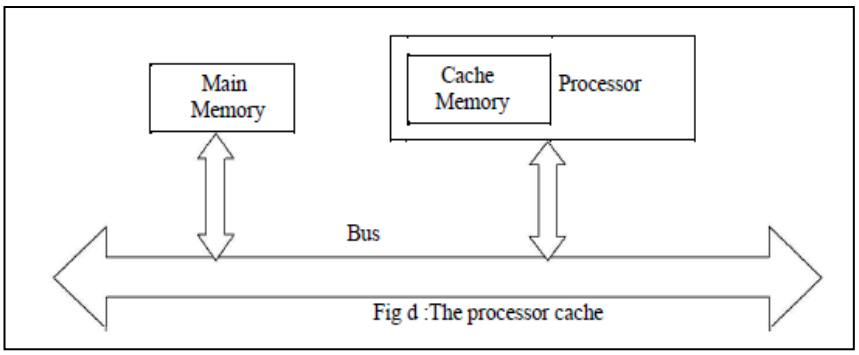


Fig: The processor cache

Let us examine the flow of program instructions and data between the memory and the processor. At the start of execution, all program instructions and the required data are stored in the main memory. As the execution proceeds, instructions are fetched one by one over the bus into the processor, and a copy is placed in the cache later if the same instruction or data item is needed a second time, it is read directly from the cache.

The processor and relatively small cache memory can be fabricated on a single IC chip. The internal speed of performing the basic steps of instruction processing on chip is very high and is considerably faster than the speed at which the instruction and data can be fetched from the main memory. A program will be executed faster if the movement of instructions and data between the main memory and the processor is minimized, which is achieved by using the cache.

For example: Suppose a number of instructions are executed repeatedly over a short period of time as happens in a program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. The same applies to the data that are used repeatedly.

The system performance can be estimated by the following factors:

1) Memory type
2) Processor clock
3) Pipelining and super scalar operation
4) Clock rate
5) Instruction set CISC & RISC

Processor clock:

Processor circuits are controlled by a timing signal called **clock**. Each machine cycle is divided into the regular time intervals called clock cycles. To execute a machine instruction the processor divides the action to be performed into a sequence of basic steps that each step can be completed in one clock cycle. The length P of one clock cycle is an important parameter that affects the processor performance.

Processor used in today's personal computer and work station has a clock rates that range from a few hundred million to over a billion cycles per second.

Basic performance equation:

Let 'T' be the processor time required to execute a program that has been prepared in some high-level language. The compiler generates a machine language object program that corresponds to the source program. Assume that complete execution of the program requires the execution of 'N' machine cycle language instructions. The number N is the actual number of instruction execution and is not necessarily equal to the number of machine cycle instructions in the object program. Some instruction may be executed more than once, which in the case for instructions inside a program loop others may not be executed all, depending on the input data used.

Suppose that the average number of basic steps needed to execute one machine cycle instruction is S, where each basic step is completed in one clock cycle. If clock rate is 'R' cycles per second, the program execution time is given by

$$T = \frac{N \times S}{R}$$

this is often referred to as the basic performance equation.

We must emphasize that N, S & R are not independent parameters changing one may affect another. Introducing a new feature in the design of a processor will lead to improved performance only if the overall result is to reduce the value of T. We assume that instructions

are executed one after the other. Hence the value of S is the total number of basic steps, or clock cycles, required to execute one instruction.

Pipelining and super scalar operation:

A substantial improvement in performance can be achieved by overlapping the execution of successive instructions using a technique called **pipelining.** Consider

ADD R1 R2 R3

this instruction adds the contents of R1 & R2 and places the sum into R3.

The contents of R1 & R2 are first transferred to the inputs of ALU. After the addition operation is performed, the sum is transferred to R3.

The processor can read the next instruction from the memory, while the addition operation is being performed. Then of that instruction also uses, the ALU, its operand can be transferred to the ALU inputs at the same time that the add instructions is being transferred to R3.

A higher degree of concurrency can be achieved if multiple instructions pipelines are implemented in the processor. This means that multiple functional units are used creating parallel paths through which different instructions can be executed in parallel with such an arrangement, it becomes possible to start the execution of several instructions in every clock cycle. This mode of operation is called **superscalar execution**.

Clock rate:

The two possibilities for increasing the clock rate 'R' are:

1. Improving the IC technology makes logical circuit faster, which reduces the time of execution of basic steps. This allows the clock period P, to be reduced and the clock rate R to be increased.

2. Reducing the amount of processing done in one basic step also makes it possible to reduce the clock period P. however if the actions that have to be performed by an instructions remain the same, the number of basic steps needed may increase.

Increase in the value 'R' that are entirely caused by improvements in IC technology affects all aspects of the processor's operation equally with the exception of the time it takes to access the main memory. In the presence of cache the percentage of accesses to the main memory is small. Hence much of the performance gain excepted from the use of faster technology can be realized.

CISC and RISC instruction set:

Simple instructions require a small number of basic steps to execute. Complex instructions involve a large number of steps. For a processor that has only simple instruction a large number of instructions may be needed to perform a given programming task. This could lead to a large value of 'N' and a small value of 'S' on the other hand if individual instructions perform more complex operations, a fewer instructions will be needed, leading to a lower value of N and a larger value of S. It is not obvious if one choice is better than the other.

But complex instructions combined with pipelining (effective value of S ≈ 1) would achieve one best performance. However, it is much easier to implement efficient pipelining in processors with simple instruction sets.

SPEC

It is very important to be able to access the performance of a computer, computer designers use performance estimates to evaluate the effectiveness of new features. A non-profit organization called SPEC- **System Performance Evaluation Corporation** selects and publishes bench marks.

The program selected range from game playing, compiler, and data base applications to numerically intensive programs in astrophysics and quantum chemistry. In each case, the program is compiled under test, and the running time on a real computer is measured. The same program is also compiled and run on one computer selected as reference.

The 'SPEC' rating is computed as follows.

$$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

If the SPEC rating = 50 means that the computer under test is 50 times as fast as reference computer. This is repeated for all the programs in the SPEC suite, and the geometric mean of the result is computed.

Let $SPEC_i$ be the rating for program 'i' in the suite. The overall SPEC rating for the computer is given by **SPEC rating** = $\prod_{i=1}^{n}(\mathbf{SPEC_i})^{1/n}$ where n is the number of programs in the suite.

REGISTER TRANSFER NOTATION

In computer data will transfer from one location to another. Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. In general, location will be identified by a symbolic name standing for its hardware binary address. For example, names for the addresses of memory locations may be LOC, PLACE, A, VAR2; processor register names may be R0, R5; and I/O register names may be DATAIN, OUTSTATUS, and so on.

The contents of a location are denoted by placing square brackets around the name of the location. Thus, the expression R1← [LOC] means that the contents of memory location LOC are transferred into processor register R1.

As another example, consider the operation that adds the contents of registers R1 and R2, and then places their sum into register R3. This action is indicated as R3 ← [R1] + [R2]. This type of notation is called **Register Transfer Notation (RTN)**.

The right side of RTN expression always denotes a value, and the left hand side is the name of the location where the value is to be placed, overwriting the old contents of the location.

ASSEMBLY LANGUAGE NOTATION

Assembly language notation is used to represent machine instructions and programs. For example, an instruction, the contents of memory location LOC are transferred into processor register R1, is specified by the statement:

MOVE LOC, R1

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

As another example, consider the operation that adds the contents of registers R1 and R2, and then places their sum into register R3. It can be represented in assembly language statement as **ADD R1, R2, R3**

BASIC INSTRUCTION TYPES (3-address, 2-address, 1-address, 0-address)

The operation of adding two numbers is a fundamental capability in any computer. The statement

C = A + B

in a high-level language program instructs the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C. When the program containing this statement is compiled, the three variables, A, B, and C, are assigned to distinct locations in the memory. Consider addresses of these locations as A, B, and C, respectively. The contents of these locations represent the values of the three variables. Hence, the above high-level language statement requires the action

C ← [A] + [B]

to take place in the computer. To carry out this action, the contents of memory locations A and B are fetched from the memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location C.

The required action can be accomplished by a sequence of simple machine instructions. Assume that instruction contains the memory address of the three operands- A, B, C. The **three-address instruction** can be represented symbolically as

Add A, B, C

Operands A and B are called source operands, C is called the destination operand and Add is the operation performed on the operands. A general instruction of this type has the format **Operation source1, source2, destination**

If k bits are needed to specify the memory address of each operand, the encoded form of 3-address instruction must contain 3k bits for addressing purpose in addition to the bits needed to denote the Add operation. For a modern processor with a 32-bit address space, a 3-address instruction is too large to fit in one word.

An alternative approach is to use a sequence of simpler instructions to perform the same task, with each instruction having only one or two operands. **Two- address instruction** of the form

Operation source, destination

are available. An add instruction of the type is **Add A, B**

which performs the operation **B ← [A] + [B].** This means that operand B is both a source and destination.

A single two- address instruction cannot be used to add the contents of locations A and B, without destroying either of them, and to place the result in location C. This problem can be solved by using another two-address instruction that copies the contents of one memory location into another.

Move B, C

which performs the operation **C ← [B],** leaving the contents of location B unchanged.
The operation **C ← [A] + [B]** can now performed by the two- instruction sequence

Move B, C
Add A, C

Even two- address instructions will not normally fit into one word for usual word lengths and address sizes. Another possibility is to have machine instructions that specify only one memory operand. When a second operand is needed, as in the case of an Add instruction, it is understood implicitly to be in a unique location. A processor register called **accumulator** may be used for this purpose. Thus, the one-address instruction

Add A

means, add the contents of memory location A to the contents of the accumulator register and place the sum back into the accumulator.

Load A

instruction copies the contents of memory location A into the accumulator and

Store A

instruction copies the contents of the accumulator into memory location A.

Using only one address instructions, the operation **C ← [A] + [B]** can now performed by executing the sequence of instructions:

Load A
Add B
Store C

Here the operand specified in the instruction may be a source or destination, depending on the instruction. In Load A instruction, address A specifies the source operand, and the destination location is the accumulator and in Store C instruction, C denotes the destination location where as accumulator is the source register.

It is also possible to use instructions in which locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a **push down stack**. Such instructions are called **zero- address instructions**.

Example: Implement the given function X = (A + B) * (C + D) using 3-address, 2-address, 1-address and 0- address instructions. Where X, A, B, C and D are memory locations.

Solution:

Three address instructions:

```
ADD   R1, A, B    R1 ← M [A] + M [B]
ADD   R2, C, D    R2 ← M [C] + M [D]
MUL   X, R1, R2   M [X] ← R1 * R2
```

Two address instructions:

```
MOV   R1, A       R1 ← M [A]
ADD   R1, B       R1 ← R1 + M [A]
MOV   R2, C       R2 ← M [C]
ADD   R2, D       R2 ← R2 + M [B]
MUL   R1, R2      R1 ← R1 * R2
MOV   X, R1       M [X] ← R1
```

One address instructions:

```
LOAD    A     AC ← M [A]
ADD     B     AC ← AC + M [B]
STORE   T     M [T] ← AC
LOAD    C     AC ← M [C]
ADD     D     AC ← AC + M [D]
MUL     T     AC ← AC * M [T]
STORE   X     M [X] ← AC
```

Zero address instructions:

```
PUSH    A     TOS ← A
PUSH    B     TOS ← B
ADD           TOS ← (A+ B)
PUSH    C     TOS ← C
PUSH    D     TOS ← D
ADD           TOS ← (C+ D)
MUL           TOS ← (A+ B) * (C+ D)
POP     X     M [X] ← TOS
```

TOS: Top of the stack