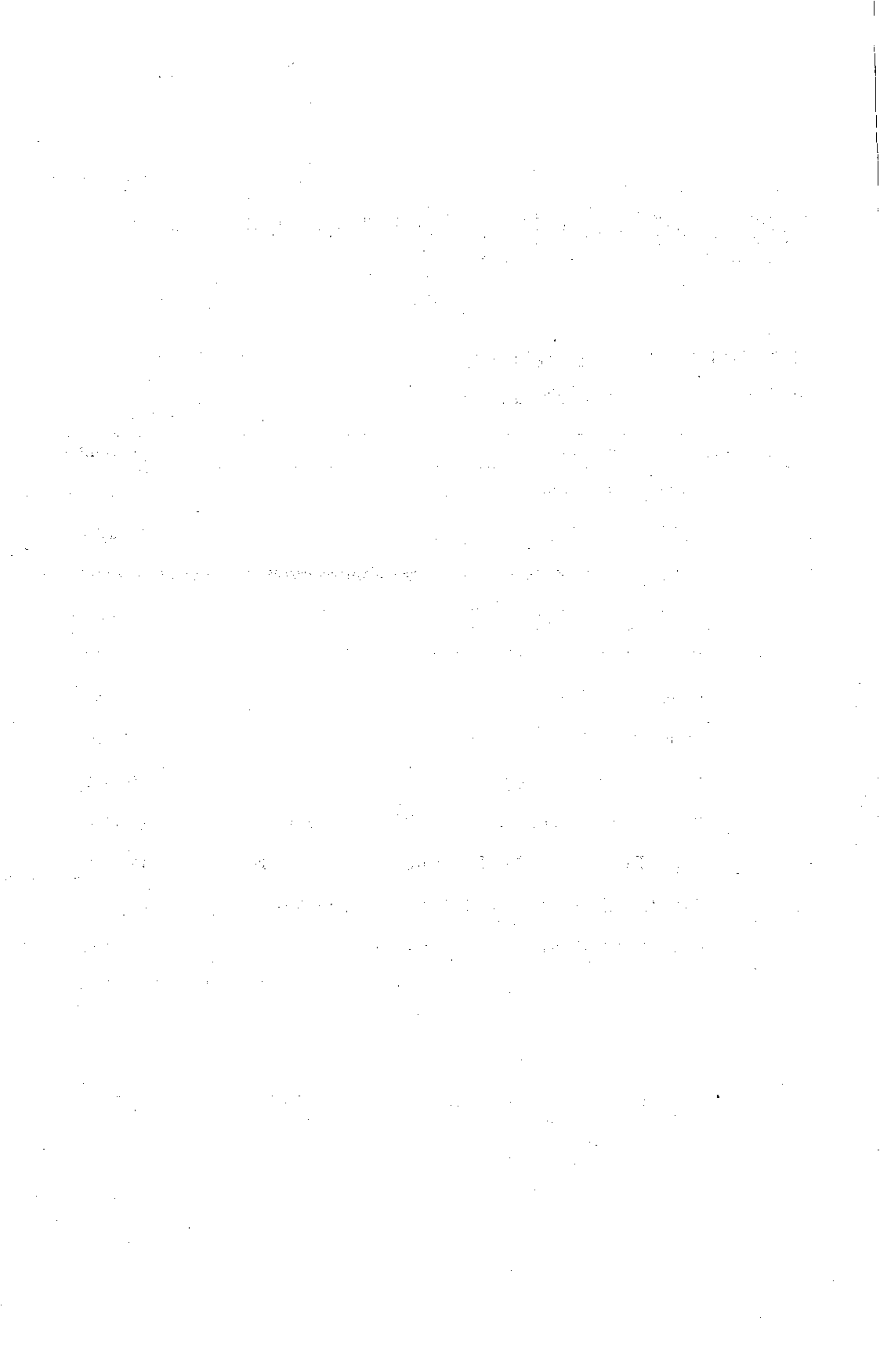
M.Sc. (C.S.) - 08

VARDHAMAN MAHAVEER OPEN UNIVERSITY, KOTA

Index

COMPUTER ARCHITECTURE AND MICRO PROCESSOR

Unit Number	Unit Name	Page Number
1.	PROCESSOR BASICS	1-13
2.	ORGANIZATION AND ARCHITECTURE	14-25
3.	DATAPATH DESIGN	26-37
4.	PROCESSOR ORGANIZATION	38-54
5.	REDUCED INSTRUCTION SET COMPUTER	55-72
6.	CONTROL DESIGN	73-86
7.	MEMORY ORGANIZATION	87-103
8.	SYSTEM ORGANIZATION	104-117
9.	INTRODUCTION TO MICRO COMPUTER SYSTEM	118-131
10.	INTRODUCTION TO MICRO COMPUTER SYSTEMS	132-158
11.	PERIPHERALS & THEIR INTERFACING WITH 8085	159-176
12.	COMPARATIVE STUDY OF 8085, 8086 & 8086	177-196



Unit-1
Processor basics

Structure of unit

1.0 Objective
1.1 Introduction
1.2 CPU organization
1.2.1 Fundamentals
1.2.2 Additional Features
1.2.3 Self learning Exercises
1.3 Data representation
1.3.1 Basic Formats
1.3.2 Fixed point numbers
1.3.3 Floating Point numbers
1.3.4 Self learning Exercises
1.4 Instruction sets
1.4.1 Instruction Formats
1.4.2 Instruction types
1.4.3 Programming consideration
1.4.4 Self learning Exercises
1.5 Summary
1.6 Glossary
1.7 Further Readings
1.8 Answer to Self Learning Exercise
1.9 Unit End Questions

1.0 Objective

This unit is concerned with computer's CPU and its related topics. The components of CPU and their functions are discussed. Instruction set and formats explained. Different ways of storage of data are discussed.

1.1 Introduction

The part of computer that perform the bulk of data processing is called the Central Processing Unit(CPU). The CPU fetches one instruction of program from the memory, decode it and then execute it. After decoding the instruction the CPU comes to know what operation is to be performed and whether the data to be processed is in memory or register of CPU. After executing one instruction it fetches the next instruction for execution till the end of program reached. The result is placed in the memory or sent to an output device according to the instruction given in the program. Besides executing the program, the CPU also control input and output devices and other component of the computer.

The main function of CPU is to execute sequence of instructions that is program. Steps involved in program execution by CPU is as follows :-

1. The CPU transfers instruction and when necessary their input data (operands) from main memory to registers of CPU.
2. CPU executes the instruction in their stored sequence except when execution sequence is explicitly altered by branch instruction.

When necessary, the CPU transfers output data(results) from CPU registers to main memory.

1.2 CPU Organization

Internal structure of CPU having three major parts as shown in figure 1.1

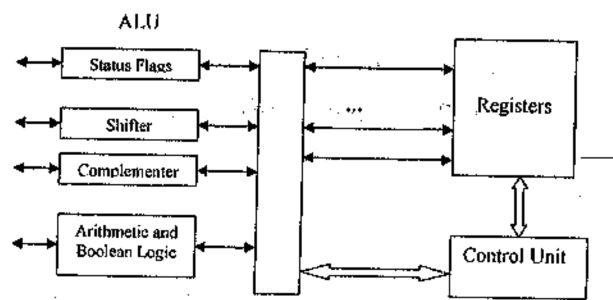


Figure 1.1 The internal Structure of CPU

1.1.1 Fundamentals

Register Set stores intermediate data used during the execution of the instruction.

The Arithmetic Logic Unit(ALU) perform arithmetic and logic operations such as addition, subtraction, multiplication , division AND, OR, NOT, Ex-OR, Left or Right Shift, Clear. Other mathematical operations such as exponentiation, logarithmic, trigonometric and floating point operations are not performed by ALU. These operations are performed by special purpose math processor called Floating Point Unit(FPU).They are either software or employ a math processor IC in micro processor based system. The use of software for such operation makes execution slow, math processor speed up program execution and reduce programming complexity. The choice depends on actual requirement and cost involved in a particular application.

The Control Unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform. It control entire operation and other devices such as memory, input and output devices of computer. It fetches instruction from memory, decode the instruction, interprets the instruction to know what tasks are to be performed and send suitable control signals to other components to perform further necessary steps to execute the instruction.

For fetching and executing an instruction the following steps are performed-

1. The address of the memory location where instruction lies, is placed on the address bus.
2. instruction is read from memory.
3. instruction is sent to the decoding circuitry for decoding.
4. addresses and data required for the execution for the instruction are read from memory.
5. these data/addresses are sent to the other section for processing.
6. the results are sent to the memory or kept in some register.
7. necessary steps are taken to fetch next instruction. For this the content of program counter is incremented.

Three different types of bus used to interconnect the CPU, memory and I/O devices

Control bus (*bidirectional*)–carries the necessary commands and control signals to the various parts of the system.

Address bus (*unidirectional*)–specifies the address of the memory word or I/O device that the CPU wants to communicate.

Data bus (*bidirectional*)–carries the data transmitted between CPU, memory and I/O devices.

Registers.

A CPU contains a number of registers to store data temporarily during the execution of a program. Registers are classified as follows:

Accumulator is register which hold one of the operands prior to the execution of an instruction, and receives the result of most of the arithmetic and logical operation.

General Purpose Registers store data and intermediate results during execution of a program. They are accessible to users through instruction if the user s are working in assembly language.

Special Purpose Registers All CPU do not contains all of these special purpose registers. A powerful CPU contains most of them. The brief description of these registers are as follows-

Program Counter(PC) holds address of the memory location which contains the next instruction to be fetched from memory.

Stack Pointers(SP) holds the address of last occupied memory location of the stack. Thus it indicate upto what memory locations the stack is already filled up.

Status Register holds 1-bit flags to indicate certain conditions that arise during arithmetic and logical operation. The important indications shown by computer are :

Carry – it indicates whether there is overflow or not.

Zero – it indicates whether the result is zero or nonzero.

Sign – it indicates whether the result is plus or minus.

Parity – it indicates whether the result contains odd number of 1s or even number of 1s.

Instruction Register hold an instruction until it is decoded.

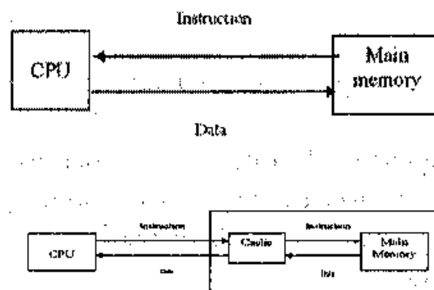
Index Register are used for addressing. The address of an operand is the sum of the contents of the index register and a constant. Instructions involves index register contain constant. The constant is added to content of index register to form the effective address, i.e. address of operands.

Memory Address Register(MAR) hold address of the instruction or data to be fetched from the memory. The CPU transfers the address of the next instruction from program counter to the memory address register. From MAR it is sent to the memory through address bus.

Memory Buffer Register(MBR) or Data Register(DR) hold the instruction code or data received from or sent to memory. The data which are written into the memory are held in this register until the write operation is completed.

External communication of CPU

As in figure 1.2 if no cache memory is available CPU communicate directly with main memory. CPU is significantly faster than main memory. It can read from or write to registers perhaps 5 to 10 times faster than it can read from or write to main memory. To increase the processor main memory speed disparity cache memory come introduced between CPU and main memory. Cache memory if smaller and faster than main memory and in some system wholly or in part it reside in CPU chip. Using cache memory, CPU can perform memory load or store instruction in single clock cycle. Where as in a system does not have cache require many cycles for these operation. Cache is designed such that it is transparent to CPU instruction that is cache and main memory forms a single seamless memory space that is called external memory.



The CPU communicates with IO devices in the same way as it communicates with external memory.

1.2.2 Additional features

Architecture extensions - The basic design of accumulator based CPU can be improved. Following architecture extensions improve their performance and ease of programming.

1) Multipurpose register set for storing data address

These will replace accumulator AC and Auxiliary registers DR and AR of our basic CPU resulting CPU has 32 such registers.

2) Additional data, instruction and address types

Most CPU have instructions to handle data and address add and subtract instruction, little extra circuitry is required for (fixed point) multiply and divide instruction, which simplify many programming tasks. CALL and RETURN instruction also simplify program design.

3) Register to indicate computation status

Status register indicate user and supervise states and exceptional conditions resulting from the instruction execution. Conditional branch instruction can test the status register, which simplifies the programming of conditional actions.

4) Program Control stack

Various special registers and instruction facilitate the transfer of control among program due to procedure calling or external interrupts. Many CPU use a flexible scheme for program control transfer. The stack memory is intended for solving key information about an interrupted program. CPU address register called a stack pointer automatically keeps track of the stack's entry point.

5) Pipelining

Modern CPUs employ a variety of speed up techniques, including cache memories and several forms of instruction level parallelism. These features add to CPU 's complexity. If some activity do not share a resource such as the system bus, they can be carried out at the same time. By merging the execution part of each instruction cycle with the fetch part of the following instruction cycle. We can reduce the overall execution time from six clock cycle to four. this overlapping of instruction fetching and execution is an example of instruction pipelining, which is an important speed up feature Fig 1.3 illustrate graphically the two stage pipelining. Each instruction can be thought of as passing through. Two consecutive stages of processing : a fetch stage implement mainly by PCU(Program Control Unit and execution stage implemented by DPU(Data Processing Unit)

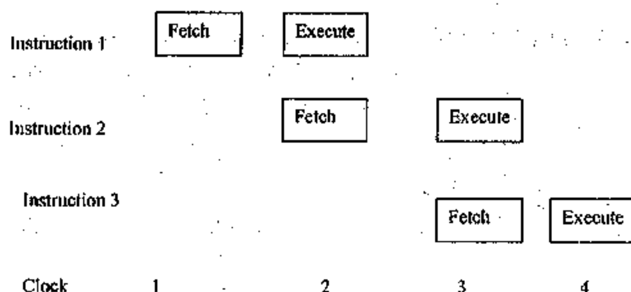


Figure 1.3 Pipelining

Two instruction can be processed simultaneously in every CPU clock cycle, with one completing its fetch phase and the other completing its execution phase. A two stage pipelining can therefore double the CPU's performance from one instruction every two clock cycles to one instruction every clock cycle.

1.2.3 Self learning Exercises

True/ False

a. Data bus is unidirectional.

b. Control bus carries the necessary commands and control signals to the various parts of the system.

Fill in the blanks

c. .... is register which hold one of the operands prior to the execution of an instruction, and receives the result of most of the arithmetic and logical operation.

d. .... is a technique used in modern CPUs to speed up processing,

1.2 Data Representation

The manner in which the data is expressed symbolically by binary digit in a computer is known as data representation.

Following are some type of numbers to be represent-

Binary is a number system using only ones and zeros (or two states).

Decimal is a number system based on ten digits (including zero).

Hexadecimal is a number system based on sixteen digits (including zero).

Octal is a number system based on eight digits (including zero).

Duodecimal is a number system based on twelve digits (including zero).

Figure 1.4 show fundamental division of information into instruction and data. Data can be further subdivided into numerical and non numerical.

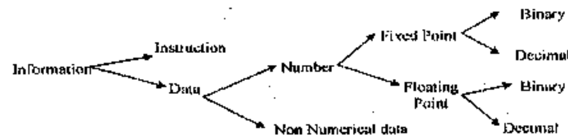


Figure 1.4 Division of instruction and data

In selecting a number representation to be used in computer, the following factor should be taken into account :

- Ø The number type to be represented; for example integer or real numbers.
- Ø The range of values likely to be encountered.
- Ø The precision of the number, which refers to the maximum accuracy of the representation.

There are two types of arithmetic operations available in computer. These are :

1. Integer(Number without fractional part) arithmetic
2. Real(number that contain a fractional part) arithmetic

therefore are two type of numbers exists integer and real numbers and computer have to have representation technique for both of them.

1.3.1 Basic Formats

Integer representations

Sign-magnitude is the simplest method for representing signed binary numbers. One bit (by universal convention, the highest order or leftmost bit) is the sign bit, indicating positive or negative, and the remaining bits are the absolute value of the binary integer.

In one's complement representation, positive numbers are represented in the "normal" manner (same as unsigned integers with a zero sign bit), while negative numbers are represented by complementing all of the bits of the absolute value of the number.

In two's complement representation, positive numbers are represented in the "normal" manner (same as unsigned integers with a zero sign bit), while negative numbers are represented by complementing all of the bits of the absolute value of the number and adding one.

In unsigned representation, only positive numbers are represented. Instead of the high order bit being interpreted as the sign of the integer, the high order bit is part of the number. An unsigned number has one power of two greater range than a signed number (any representation) of the same number of bits.

Real Representation

There are two methods of representing real numbers

Fixed point representation

Floating Point representation.

1.3.2 Fixed Point Number

Computers are designed such that each location(all known as *word*) in memory stores only a finite number of digits. That's why operands in arithmetic operation have only a finite number of digits.

The number, in which decimal point is always fixed in one position is known as Fixed point number and representation technique is called Fixed Point Representation. In this representation decimal point is not actually present, but its presence is only assumed (see figure1.5).

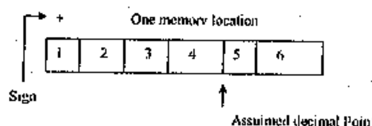


Figure 1.5 Fixed point representation

Fixed point formats allow a limited range of values and have relatively simple hardware requirements. So this representation is generally used when hardware cost, speed, or complexity is important
Floating point number, on the other hand allow a much larger range of values but require either costly processing hardware or lengthy software implementation.

1.3.3 Floating point Number

There is no fixed number of digits before and after decimal point; that is the decimal point can float in floating point number. Floating point representation has two parts: first part resent a signed fixed point number called mantissa. The second part designates position of decimal point is called exponent. The fixed part mantissa can be fraction or integer.
For example decimal number 4172.123 is represented in floating point as follows –

Mantissa	Exponent
+.4172123	+04

Decimal shown above is an assumed decimal point. It is not physically indicated in register. If we use integer system of representation for mantissa the number 4112.123 will be represented as shown below

Mantissa	Exponent
+4112123	-02

In this representation sign of exponent has been shown negative to indicate the actual position of decimal point lies two decimal position to the left of the assumed decimal point.

Value of exponent indicate actual position of decimal point is four position to the right of the indicated decimal point fraction. This representation is equivalent to the scientific notation $+.4172123 \times 10^{+4}$.
Floating point is always interpreted to represent a number into following form-

$$M \times R^E$$

Floating point representation uses a second register to store a number that designate the position of the decimal point in first register. Only the mantissa and exponent are physically stored in register(including their signs). The radix R and radix point position of the mantissa are always assumed.
Floating point binary number is represented in similar manner except that it uses base 2 for the exponent. For example binary number +10011.11 is represented with 8-bit fraction and 6-bits exponent as follows-

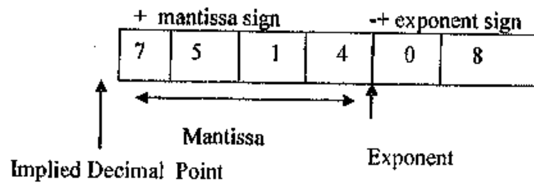
Fraction	Exponent
01001110	000100

Left most zero in fraction part denote positive sign. The floating point number is equivalent to –

$$M \times 2^E \quad = \quad +(.1001110) \times 2^{+4}$$

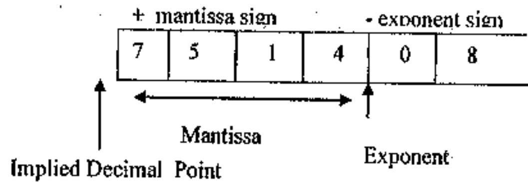
A normalize floating point number provide maximum possible precision. A floating point number is said to be *normalized* it the most significant digit of the mantissa is non zero. For example 350 and 11010 are normalized but 000350and 00011010 are not normalized. The number can be normalized by shifting three position to left and discard leading zeros to obtain11010000 three shift multiply the number by $2^3 = 8$, to keep same value for the Floating point number exponent must be subtracted by 3. The mantissa and exponent will have their own independent signs.

For example 75.14×10^6. is represented in our computer as .7514E8 as given in following figure.



Since leading zeros serves only to locate the decimal point therefore for that number shifting of the mantissa till its most significant digit is non zero performed. So the mantissa contains maximum possible number of significant digits. We can not normalize a zero, since it does not have non zero digits. It is represented in floating point by all zeros in the mantissa and exponent.

For example .004854 will be stored as –



Fixed point format require simple hardware but the range of values it allows is very limited whereas Floating point format allow a wide range of values but require expensive hardware or large software implementation.

Floating point number can represent are just approximations. So floating point represented are less accurate and slower.

In floating point representation some number can be represented in many ways for example 3.0×10^{14}, 0.3×10^{15} and 0.03×10^{16} represent same number.

4.4.4 Self learning Exercises

True/ False

e. Floating Point representation cover higher range of numbers that can be stored compared to fixed Point representation.

f. Floating Point numbers provide high accuracy.

Fill in the blanks

g. The manner in which the data is expressed symbolically by binary digit in a computer is known as ....

h. In Floating point representation number has two parts .... and ....

1.4 Instruction set :

The essential elements of a computer instruction are opcode and operand. opcode specified operation to be performed which can be either arithmetic and logical operation: movement of data between two register, register and memory, or two memory location ; I/O and control. Operand reference specify a register or memory location of operand data. The type of data may be address, number, character or logical data.

Elements of Machine Instruction :-

Operation code : specifies the operation to be performed (eg. ADD, SUB)

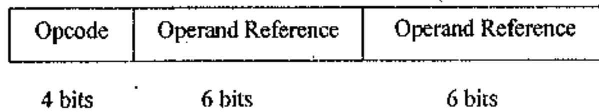
Source operand reference : operand that are input for the operation.

Result operand reference : The operand 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.

Instruction representation :

During instruction execution, an instruction is read into an instruction register (IR) in CPU. The CPU must be able to extract the data from the various instruction fields to perform the required operation. Simple instruction format is given below –



Opcode	Operand Reference	Operand Reference
4 bits	6 bits	6 bits

Figure 1.6 A Simple Instruction Format

The large number of instruction of a given computer, gives the flexibility for the user and the programmer to carry out various computational task. The instruction set provide mechanism to decide location of operand mechanism of locating an operand are decided by the mode of addressing.
The instruction of a computer broadly classified into following categories -

1. Arithmetic Instruction
2. Logical Instruction
3. Shift Instruction
4. Data Movement Instruction
5. Program control Instruction

Arithmetic Instruction

The Instruction include operation of addition, subtraction, multiplication, and division. Simple processor do not have hardware circuitry for multiplication and division. So they perform by programming technique which takes larger time. But all advanced processor have hardware solution for multiplication and division operation.

Brief description of these instruction are as follows -

ADD : This instruction is used to add two values. This instruction is used as

ADD destination, Source

content of source and destination is added and result stored in destination.

SUB : This Instruction is used similar to ADD Instruction except that subtraction is performed instead of addition.

MUL : MUL Instruction is used to multiply and used as

MUL Destination, Source

product of source and destination evaluated and result stored in destination

DIV : perform division operation. and used as follows

DIV destination, source

means destination divide by content of source.

Logical Instruction

There are some logical instruction, some of them are of two operand and some are of one. Some of them are explained below -

AND AND destination, source perform the logical AND operation between content of register or memory specified as destination and source in instruction and result store in destination.

OR OR destination, source this instruction is similar to AND except that instead of logical AND it perform logical OR operation.

XOR XOR destination, source perform exclusive OR between content of source and destination and result stored in destination.

NEG/NOT used for negating the operand.

CLR/SET used for clearing and setting the one bit flip flop usually flag register to zero or one.

Shift instruction

Some of shift instruction are listed below -

SHL/SHR - used to shift the content of specified register Left and right respectively.

ROR/ROL - this instruction rotate all the bits in a specified word to the right / left some number position.

RCL/ RCR - this instruction rotate all the bits in a specified register some number of bits positions to left/ right.

SAR - this instruction shift each bit in the specified register some number of bit position to right.

Data Movement Instruction

These instruction moves the data from one location to another without modifying data in the process. some of them are

MOV - MOV destination, source instruction moves the data from source to destination.

STORE - This Instruction stores the content of an impled register usually accumulator, into specified memory location.

LOAD - This Instruction copies the content of specified location into register specified in the instruction.

Program Control Instruction

These instructions are used for controlling the sequence of program execution. Since many situation program is not executed in sequence therefore some widely use instruction for deviation of execution of program in sequence are described briefly below-

CALL This Instruction is used to transfer execution to subroutine, or a procedure. When this Instruction executed control of execution is transferred to procedure use after all relevant data is moved to stack as soon as execution of procedure completed control return to location where it has left.

JMP used to deviate from the usual sequence of execution. We can also associate condition with instruction to transfer execution control when condition is specified.

RETURN it is last executed Instruction of procedure. it restore all relevant data stored in stack at the time of entering the procedure.

IN used to enter data from peripheral device into computer.

OUT used to transfer the data from computer into peripheral device.

HLT is an instruction used to stop further execution of computer instruction.

1.4.1 Instruction Format

Instruction format contains number of parts like opcode, mode bit address fields(s), register field etc. The instruction format specifies that given operand and/or mentioned in the opcode is executed on a data whose location is specified by the contents of the address field and mode bits. An instruction format is normally stored in one word of memory. However, it is possible that an instruction format can not accommodated in one memory word in this case it is stored in number of consecutive memory location. A computer has a large variety of instruction formats of variable length and the control unit decode the instruction and facilitate the execution of the instruction.

F irst part of instruction format is operation code which specifies the operation to be performed by the computer.

Second part mode bits gives the mechanism of calculating the address of operand

Third part contains address of operands

Instruction format may have zero, one, two, three fields to specify an address. Instructions are labeled as zero address instruction, one address instruction, two address instruction or three address instruction depends on number of arguments in instruction.

There are three basic types of CPU organization.

1. The single accumulator organization uses an implied accumulator register and the instruction format uses one address such as ADD AX.

2. General register organization instruction format require three register address fields to identify the two operands and the result operand for example ADD AX, BX, CX in which content of Bx and CX added and stored in AX. Number of registers reduced to two if result is stored in one of the source register.

3. In stack oriented organization address of operand is always implied hence does not use any address field, such as ADD implied that content of top two location of stack removed into registers, content added and result is stored in the new top of the stack.

Instruction in stack organization uses no address that's why length of instruction is smallest where as in register oriented instruction with three address is largest. While writing program in assembly stack oriented program have large number of instruction where as register oriented program have small number of instructions.

Three Address Instructions :

Instruction formats use three address fields to specifies the location of two source operands and the result operand. These address can be memory or registers. Program to evaluate Z = (P + Q) * (R - S) / (T + V) using three address scheme in computer which have four registers AX, BX, CX, DX to hold intermediate data is given below :-

```
ADD DX, P, Q
SUB CX, R, S
ADD Bx, T, V
MUL AX, CX, DX
DIV Z, BX, AX
```

The three address format provide short length program. But length of instruction format registers three address hence fairy long.

Two Address Instruction :

The instruction which are of the source behave as a destination also is required only two address is known as Two address instruction. In this scheme program for evaluation of Z = (P + Q) * (R - S) / (T + V) will be as follows -

```
MOV DX, P
ADD DX, Q
MOV CX, R
SUB CX, S
MOV BX, T
ADD BX, V
MUL CX, DX
DIV CX, BX
MOV Z, CX
```

MOV instruction transfer the operand from memory to register or from register to memory.

One address Instructions:

These instructions use implied register, accumulator, for data manipulation. Program is written neglecting the second register and assuming that accumulator is long enough to take care of the multiplication and division operation. Program for instruction Z = (P + Q) * (R - S) / (T + V) is given below-

```
LOAD P
ADD Q
STORE M
LOAD R
SUB S
MUL M
STORE M
LOAD T
ADD V
DIV M
STORE Z
```

It is assumed that all the operations are performed between accumulator and memory operand.

Zero Address Instructions:

A computer organized using only stack does not use any address. However the PUSH and POP instruction require one address fields to specify the operand that from/ to the top of the stack. Arithmetic expression are evaluated using top two content of stack and the result is stored on the top of the stack. Program for Z = (P + Q) * (R - S) / (T + V) will be as follows in zero address instructions -

PUSH P
PUSH Q
ADD
PUSH R
PUSH S
SUB
MUL
PUSH T
PUSH V
DIV
POP Z

The zero-address machine needs more instructions than the 3-address machine, but they are *shorter* instructions so the 0-address machine does not necessarily have a performance disadvantage, and the 3-address machine does not necessarily have a performance advantage

1.4.2 Instruction Type

Let an instruction Y = X + Y, add value of X and Y and put the result in Y. Let us assume that variable X and Y correspond to location 114 and 214. This operation could be accomplished as follows -

1. Load register with value of location 114.
2. Add the value of location 214 to the register.
3. Store the content of register in the memory location 114.

So you can see that single high level language instruction require three machine language instruction. High level language express operation in a concise algebraic from using variable where as machine language express operation in basic from which involve movement of data to/from register.

A computer should have a set of instruction that allows the user to formulate any data processing task. Any program written in high level language must be translated into machine language to be executed. Thus set of machine instruction must be sufficient to express any of the instruction from a high level language. We can categorize instruction type as follows -

Data processing : Arithmetic and logical instruction
Data Storage : Memory instruction
Data Movement : I/O instruction
Control : Test and branch instruction.

Arithmetic instruction provide computational capabilities for processing numeric data.
Logic(Boolean) instruction operate on the bits of word as bits rather than as number.
Memory instruction for moving data between memory and registers.
I/O instruction are used to test the value of data word or the status of a computation.
Branch instruction are used to branch to different set of instruction depending on the decision made.

1.4.3 Programming consideration

Data processing operations require three operands such that
X := A + B   that is addition require X, A and B
Each instruction involve set of low level instructions which actually run at the time of execution of single high level instruction. For example above addition instructions consists following low level instructions to run-
AC := M (A)

DR := AC
AC := M (B)
AC := AC + DR
M(X) := AC

Single address instructions (i.e. instruction with one address explicit memory address) can implement by sequence of instruction.

For example set of instructions for addition with single addressing will be as follows -

LOAD A	Load A from M into accumulator
MOVE DR AC	Move content of accumulator to data register
LOAD B	Load B from M into accumulator
ADD	Add contents of DR to AC
STORE X	Store content of AC in M.

Above program fragment use only load and store instruction to access memory a feature called load/ store architecture. Memory referencing ADD B instruction take longer time than ADD instruction that reference only CPU register and memory reference complicate the instruction decoding logic in the CPU.

1.1.1 Self learning Exercises

True/ False

i. Instruction set allows the user to formulate any data processing task.

j. MOV instruction transfer the operand from memory to register or from register to memory.

Fill in the blanks

k. During instruction execution, an instruction is read into an .... in CPU.

l. The essential elements of a computer instruction are .... and ....

1.2 Summary

The main task of CPU is to fetch instruction from an external memory and execute them. This task require program counter to keep track of the active instruction and registers to store the instruction and data.

The simplest CPU consists central data registers, accumulator with ALU capable of addition, subtraction and logic operations. Most CPU consists 32 or more general purpose registers replaces the accumulator.

The arithmetic capabilities of simpler processor are limited to the fixed point(inter also) instruction. More powerful CPU have built-in hardware to execute floating point instruction.

Computer store and process information in various formats, two major formats are fixed point and floating point. The two most common binary number representation are signed magnitude and two's complement. Each representation simplifies the implementation of addition and subtraction.

Floating point numbers is a pair of fixed point called Mantissa(M) and exponent(E) and represent number $M \times B^E$ where B is implicit base. Floating point number greatly increase the numerical range obtainable using given word size. but require complex arithmetic circuit than fixed point.

The function performed by a CPU are defined by instruction set. An instruction consists of an opcode and a set of operands an address fields. Various techniques called instruction formats are used to specify operands. An instruction set should be complete, efficient and easy to use in some broad sense

1.6 Glossary

Arithmetic Logic Unit(ALU)

A Part of a computer that perform arithmetic operations, logic operations, and related operations.

Central Processing Unit (CPU)

That portion of a computer that fetches and executes instructions. It consists ALU, control Unit and registers.

Control Unit

The part of CPU that control CPU operations, including ALU operations, the movement of data within CPU, and the exchange of data and control signals across external interfaces.

Fixed Point Representation

A radix numeration system in which the radix point is implicitly fixed in the series of digital places by some

convention upon which agreement has been reached.

Floating Point Representation

A numeration system in which a real number is represented by a pair of distinct numerals, the real number being the product of the fixed point part, one of the numeral, and a value obtained by raising the implicit floating point base to a power denoted by the exponent in the floating point representation, indicated by second numeral.

Instruction

An instruction that can be recognized by the processing unit of the computer for which it is designed.

Instruction Set

A complete set of the operators of the instructions of a computer together with a description of the types of meaning that can be attributed to their operands.

Instruction Formats

The layout of a computer instruction as a sequence of bits. The format divides the instruction into fields, corresponding to the constituent elements of the instruction.(e.g. operand, opcode).

1.7 Further Readings

- J.P.Hayes: Computer Architecture and Organization, McGraw-Hill International.
- William Stallings: Computer Organization & Architecture, Pearson Education.
- M.Morris Mano: Computer System Architecture, Prentice Hall of India.

1.8 Answer to self learning Exercises

Question No.	Answer	Question No.	Answer
A	False	G	Data representation
B	True	H	Mantissa, Exponent
C	Accumulator	I	True
D	Pipelining	J	True
E	True	K	Instruction Register
F	False	L	Opcode , Operand

1.9 Unit End Questions

1. Why are transfer of control instruction needed.?
2. What are the typical elements of a Machine instruction.?
3. What is the need of floating Point representation.?
4. Write use of each components of CPU.
5. Describe Instruction set and Instruction formats.

Unit-2
Organization and architecture

Structure of unit

2.0 Objective
2.1 Introduction
2.2 Buses
2.2.1 System Buses
2.2.2 Self learning Exercises
2.3 Computer Components
2.3.1 The Control Logic gates
2.3.2 Adder and Logic Circuit
2.3..3 Self learning Exercises
2.4 Computer Function
2.5 Interconnection Structure
2.5.1 Time Shared Common bus
2.5.2 Multiport Memory
2.5.3 Crossbar Switch
2.5.4 Multistage Switching Network
2.5.5 Hypercube Interconnection
2.5.6 Self learning Exercises
2.6 Bus Interconnection
2.6.1 Single Bus
2.6.2 Multiple Bus Hierarchies
2.6.3 Self Learning Exercise
2.8 PCI (Peripheral Component Interconnect)
2.9 Input/Output – External Devices.
2.10 Summary
2.11 Glossary
2.12 Further Readings
2.13 Answer to Self Learning Exercise
2.14 Unit End Questions

2.0 Objective

This unit is concerned with computer's Buses and its interconnection. Various computer components and computer functions are also discussed. In last PCI (Peripheral Component Interconnect) and Input/ Output External device are explained briefly.

2.1 Introduction

Computer architecture in computer engineering is the conceptual design and fundamental operational structure of a computer system. It is a blueprint and functional description of requirements and design implementations for the various parts of a computer, focusing largely on the way by which the central processing unit (CPU) performs internally and accesses addresses in memory. It may also be defined as the science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals.

Computer architecture refers to those attributes of a system visible to a programmer, or put another way, those attributes that have a direct impact on the logical execution of a program. Computer organization refers to the operational units and their interconnection that realize the architecture specification.

Examples of architecture attributes include the instruction set, the number of bit to represent various data types (e.g.., numbers, and characters), I/O mechanisms, and technique for addressing memory. Organization attributes include those hardware details transparent to the programmer, such as control signals, interfaces between the computer and peripherals, and the memory technology used.

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 a mechanism that makes repeated use of the add unit of the system. The organization decision may be bases on the anticipated frequency of use of the multiply instruction, the relative speed of the two approaches, and the cost and physical size of a special multiply unit.

2.2 Buses

A bus is communication pathway connecting two or more devices is known as bus.

- Address bus- identify source or destination
- Data bus – carries data or instruction
- Control Bus – control and timing information such as memory read/write, interrupt request, clock signal

Bus can have following characteristics –

Dedicated - a line is permanently assigned either to one function or to a physical subset of computer components.

Time multiplexed - using the same lines for multiple purposes (different purposes at different times)

Physical dedication - the use of multiple buses, each of which connects to only a subset of modules, with an adapter module to connect buses and resolve contention at the higher level

Centralized - a single hardware device called the bus controller or arbiter allocates time on the bus.

Distributed - each module contains access control logic and the modules act together to share the bus.

Computer system contain a number of buses at various levels to facilitate the transfer of information between components. The CPU contains a number of internal buses for transferring information between processor register and ALU. Some of them are as follows-

A memory bus consists of lines for transferring data address and read/write information.

A I/O bus is used to transfer information to and from input and output devices.

system bus A bus connects major components(CPU, I/O, memory) in a multiprocessor system.

The processor in a shared memory multiprocessor system request access to common memory or other common resources through the system bus. If no other processor is currently utilizing the bus, the requesting processor may be granted access immediately. But requesting processor must wait if another processor is currently utilizing the system bus. And other processor may request the system bus at the same time. Arbitration must be performed to resolve this multiple contention for the shared resources. The arbitration logic would be part of the system bus controller placed between the local bus and the system bus.

2.2.1 System Bus

A typical system bus consists of approximately 100 signal lines. these lines are divided into three functional groups data address and control. In addition there are power distribution lines that supply power to components. For example the IEEE standard 796 multibus system has 16 data lines, 24 address lines, 26 control lines and 20 power lines for total of 86 lines.

The data lines provide a path for the transfer of data between processors and common memory. The number of data lines are usually multiple of 8, with 16 and 32 being most common.

The address lines are used to identify memory address or any other source or destination such as input or output ports. The number of address lines determine, the maximum possible memory capacity in the system. For example an address of 24 lines can access up to 224(superscript 16 mega) words of memory. Address lines are unidirectional from processor to memory and data lines are bidirectional, allowing the transfer of data in either direction.

Data transfer over system bus can be either synchronous or asynchronous. In synchronous bus each data item is transferred during a time slice known in advance to both source and destination units. Synchronization achieved by driving both units from a common clock or to have separate clocks of approximately the same frequency in each unit. Synchronization signals are transmitted periodically in order to keep all clocks in the system in step each other.

In asynchronous bus each data item being transferred is accompanied by handshaking control signals to

indicate when the data are transferred from the source and received by the destination.

The control lines provide signals for controlling the information transfer between units. Timing signals indicate the validity of data and address information. Command signal specify operation to be performed. Typical control lines include transfer signals such as memory read and write, acknowledge of transfer, interrupt request, bus control signals such as bus request and bus grant, and signals for arbitration procedures.

Typical physical arrangement of a system bus

- A number of parallel electrical conductors
- Each system component (usually on one or more boards) taps into some or all of the bus lines (usually with a slotted connector)
- System can be expanded by adding more boards
- A bad component can be replaced by replacing the board where it resides

2.2.2 Self learning Exercises

True/Falsse

A. The data lines provide a path for the transfer of data between processors and common memory.

B. In Asynchronous bus each data item is transferred during a time slice known in advance to both source and destination units

Fill In the Blanks

C. The .... are used to identify memory address or any other source or destination

D. In .... each data item being transferred is accompanied by handshaking control signals to indicate when the data are transferred from the source and received by the destination.

2.3 Computer Components

The basic computer consists of the following hardware components :

1. A memory unit with 4096 words of 16 bits each.
2. Nine Registers : AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC
3. Seven flip flops : I, S, E, R, IEN, FGI, and FGO
4. Two decoder : a 3 x 8 operation decoder and a 4 x 16 timing decoder
5. A 16-bit common bus
6. Control Logic gates
7. Adder and logic circuit connected to the input of AC

2.3.1 The Control Logic gates :

The block diagram of control logic gates are shown in Fig 2.1. The inputs to this circuit come from the two decoders, the I flip-flop, and bits 0 through 11 of IR. The other inputs to the control logic gates are : AC bits 0 through 15 to check if AC = 0 and to detect the sign bit in AC(15); DR bits 0 through 15 to check if DR = 0; and the values of seven flip-flops.

The output of the control logic circuit are :

1. Signals to control the inputs of the nine registers.
2. Signals to control the read and write inputs of memory
3. Signals to set, clear, or complement the flip-flops
4. Signals for S2, S1, and S0 to select a register for the bus.
5. Signals to control the AC adder and logic circuit.

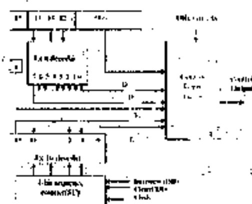


Figure 2.1 Control Logic gates

2.3.2 Adder and Logic Circuit:

Adder and logic circuit has three set of inputs. One set of 16 inputs comes from the outputs of AC. Another set of 16 inputs comes from data register (DR). A third set of eight inputs comes from the input register INPR. The outputs of adder and logic circuit provide the data inputs for the register. In addition, it is necessary to include logic gates for controlling the LD, INR and CLR in the register and for controlling the operation of the adder and logic circuit.

The adder and logic circuit can be divided into 16 stages, with each stage corresponding to one bit of AC. Each stage has a JK flip-flop, two OR gates and two AND gates. The load(LD) input is connected to the inputs of the AND gates. One stage of adder and logic circuit consists of seven AND gates, one OR gate and a full-adder.

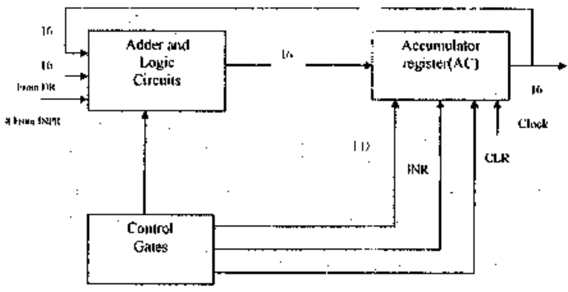


Figure 2.2 Adder and Logic Circuits

2.2.3 Self learning Exercises

True/ False

E. A basic computer consists 4 flip-flops.

F. Adder and logic circuit has three set of inputs.

Fill in the blanks-

G. A basic computer consists .... registers.

H. In Adder and Logic circuit one set of 16 inputs comes from .... Another set of 16 inputs comes from .... A third set of eight inputs comes from the input register ....

2.4 Computer Functions

In general terms, there are four main functions of a computer:

- Gather Data
- Data processing
- Data storage
- Data movement
- Control

Gather Data All computers, no matter what their size, must gather data before they can process the data. data can be gathered— manually, automatically, or a combination of both. In manually a operator or technician will input the data to the computer. Automatically gathering data means the computer receives data from another system, subsystem, or equipment. Many computer systems are designed to gather data using a combination of both the manual and automatic methods.

The computer, of course, must be able to process data. The data may take a wide variety of forms, and the range of processing requirements is broad.

It is also essential that a computer store data. Event if the computer is processing data on the fly (i.e., data come in and get processed, and the results go right out), the computer must temporarily store at least those pieces of data that are being worked on at any given moment. Thus, there is at least a short-term data storage function. Files of data are stored on the computer for subsequent retrieval and update.

The computer must be able to move data between itself and the outside world. The computer's operating

environment consists of devices that serve as either sources or destinations of data. When data are received from or delivered to a device that is directly connected to the computer, the process id known as input-output (I/O), and the device is referred to as a peripheral. When data are moved over longer distances, to or from a remote device, the process is known as data communications.

Finally, there must be control of there three functions. Ultimately, this control is exercised by the individual who provides the computer with instructions. Within the computer system, a control unit manages the computer's resources and orchestrates the performance of its functional parts in response to those instructions.

2.5 The Interconnection Structures

The components that form a multiprocessor system are CPUs, IOPs connected to input-output devices and a memory units that may be partitioned into a number of separate modules The interconnection between the components can have different physical configuration, depending on the number of transfer paths that are available between the processor and memory in shared memory system. Some of these schemes are presented below.

2.5.1 Time Shared Common bus

A common bus multiprocessor system consist of a number of processors connected through a common path to a memory unit. Only one processor can communicate with memory or other processor at any given time. Transfer operation are conducted by processor in control of the bus. Any other processor want to transfer must first check availability of bus, and as it become available processor can address destination to initiate transfer. A command is issued to inform destination what operation is to be performed. The receiving unit recognize its address in bus and responds to the control signal from sender, after that transfer is initiated.

Advantages and Disadvantages

1. Since one common bus is shared by all processors, transfer conflict can be resolved easily. These conflict must resolved by incorporating a bus controller that establishes priorities among requesting units.
2. A single common bus system is restricted to one transfer at a time, therefore total overall transfer rate within system limited by speed of single path.

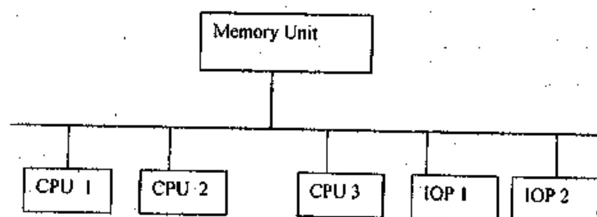


Figure 2.3 Time-shared Common Bus

2.5.2 Multiport Memory

A multiport system employ separate buses between each memory module and each CPU. For 4-CPU and 4-memory modules(MMs) are shown in Fig. 2.4 . Each processor bus is connected to each memory module and it consists address, data and control lines required to communicate with memory. The memory module have 4-ports and each port accommodates one of the buses. The module have internal control logic to determine which port will have access to memory at given time. Memory access conflicts are resolved by assigning fixed priorities to each memory port. The priority for memory access associated with each processor may be established by the physical port position that its bus occupies in each module.

Advantages and Disadvantages

1. It have high transfer rate that can be achieved due to multiple paths between processor and memory.
2. It requires expensive memory control logic.

3. It require large number of cables and connectors.
4. This is usually appropriate with small number of processors.

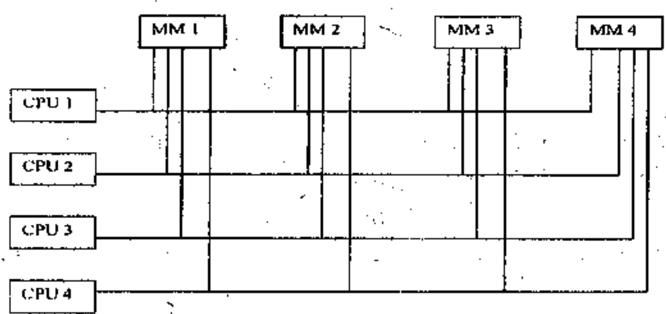


Figure 2.4 Multiport Memory

2.5.3 Crossbar Switch

The crossbar switch organization consists of a number of cross points that are placed at intersection between processor bus and memory module paths. Crossbar switch interconnection 4-CPU and 4-memory modules is shown in Figure 2.5

1. The small square in each cross point is a switch that determines the path from a processor to a memory module.
2. Each switch point has control logic to set up the transfer path between a processor and memory.
3. It examine the address that is placed in the bus to determine whether its particular module is being addressed.
4. It also resolve multiple requests for access to the same memory module on a predetermined priority basis.

Since there is separate path associated with each module therefore crossbar switch organization support simultaneous transfer from all memory modules.

The hardware required to implement the switch can become quite large and complex.

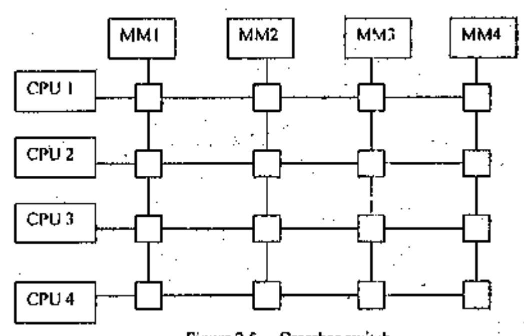


Figure 2.5 Crossbar switch

2.5.4 Multistage Switching Network

The basic component of a multistage network is a two-input, two-output interchange switch as shown in Fig 2.6 The 2 x 2 switch has two inputs, labeled A and B and two outputs labeled 0 and 1. There are control signals associated with the switch that establish the interconnection between the input and output terminals. The switch has capability of connecting input A to either of the outputs. Terminal B of the switch behaves in a similar fashion. The switch also has the capability to arbitrate between conflicting requests. If input A and B both request the same output terminal, only one of them will be connected; other will be blocked.

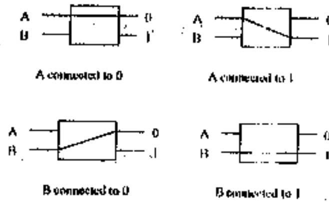


Figure 2.6 two-input, two-output interchange switch

Using the 2 x 2 switch as a building block it is possible to build a multistage network to control communication between a number of sources and destinations. One of the arrangement is shown in Fig 2.7. The two processors P_1 and P_2 are connected through switches to eight memory modules marked in binary from 000 through 111.

The path of source to destination is determined from binary bits of the destination. The first second, third bits specifies switch output in first, second and third level respectively. For example to connect P_1 to memory 101, it is necessary to form a from P_1 to output 1 in first level, output 0 in second level and output 1 in third level.

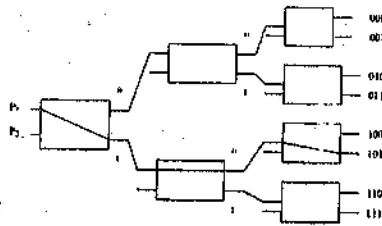


Fig 2.7 Binary Tree with 2 x 2 switches

2.5.5 Hypercube Interconnection

The hypercube or binary n-cube multiprocessor structure is loosely coupled system composed of $N = 2^n$ processors interconnected in n-dimensional binary cube. Each processor form a node of cube. Although it is customary to refer to each node as having a processor, in effect it contains not only a CPU but also local memory and I/O interface. Each processor has direct communication path to n other neighbor processors. These path correspond edge of the cube. There are 2^n distinct n-bit binary addresses that can be assigned to the processors. Each processor address differs from that of each of its n neighbors by exactly one bit position.

Fig 2.8 Shows the hypercube structure for n = 1,2 and 3. A one-cube structure has n= 1 and $2^n = 2$. A two-cube structure has n= 2 and $2^n = 4$ that means four nodes interconnected as a square. A three cube structure has eight node interconnected as a cube. Each node as assigned a binary address in such a way that the address of two neighbors differ in exactly one bit position. For example the three neighbors of the node with address 100 in a three cube structure are 000, 110 and 101. Each of these binary numbers differs from address 100 by one bit value.

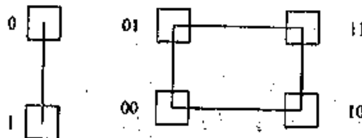


Figure 2.8 The hypercube structure

2.5.6 Self learning Exercises

True/False

I. The interconnection between the components can have same physical configuration.

J. Crossbar switch organization support simultaneous transfer from all memory modules.

Fill in the blanks

K. A multiport system employ separate .... between each memory module and each CPU.

L. There are .... associated with the switch that establish the interconnection between the input and output terminals.

2.6 Bus Interconnection

2.6.1 Single Bus

If two or more devices connect with single bus then signal transmitted by one of the device is available for reception by all other devices attached to the bus. But if two devices transmit during the same time their signals will overlap and become garbled. Therefore single bus raise following problem.

- Single device can successfully transmit.
- Lots of devices on one bus leads to propagation delay.
- Bus may become bottleneck of the system.

2.6.2 Multiple Bus Hierarchies

Since if a great number of devices are connected to single bus, performance will suffer therefore most system use multiple buses, generally laid out in a hierarchy.

traditional structure is given below-

There is a local bus connecting CPU to cache and that may also support one or more local devices. Cache is also connected to system bus which connect main memory to cache. Now with this structure I/O transfer to and from the main memory across the system bus do not interfere with the processor activity.

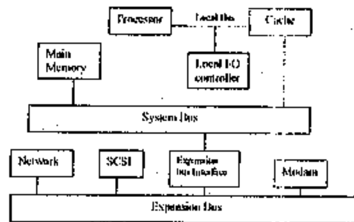


Figure . 2.9 Traditional bus architecture

This traditional bus architecture is reasonably efficient but begins to break down as higher and higher performance is seen in I/O devices. In response to these growing demands a common approach is used to build a high speed bus that is closely integrated with the rest of the system, requiring only a bridge between the processor's bus and high speed bus. Typical example of this arrangement is shown in –

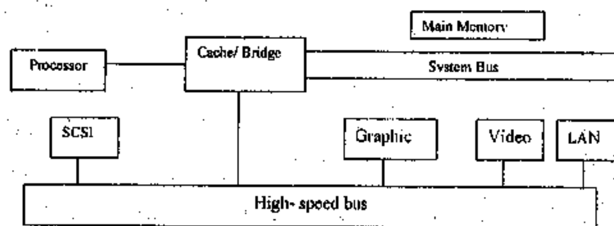


Figure 2.10 Multiple Bus architecture

Where SCSI : small computer system interface to support local disk drives, CD-ROMs, and other

peripherals.

Serial: serial port to support a printer or scanner

- The high speed bus brings high demand devices closer integration with the processor and at the same time is independent of the processor.
- Changes in processor architecture do not affect the high speed bus and vice versa.
- It is possible to connect I/O controllers directly onto the system bus, A more efficient solution is to make use of one or more expansion buses for this purpose
- Allows system to support wide variety of I/O devices
- Insulates memory-to-process traffic from I/O traffic

2.6.3 Self Learning Exercise

True/ False

M. If two or more devices connect with single bus then signal transmitted by one device at a time.

N. SCSI stands for small computer system interface.

Fill in the blanks.

O. Lots of devices on one bus leads to ....

P. Two types of bus interconnection are .... and ....

2.7 PCI (Peripheral Component Interconnect)

Currently by far the most popular local I/O bus, the *Peripheral Component Interconnect (PCI)* bus was developed by Intel and introduced in 1993. It is geared specifically to fifth- and sixth-generation systems, although the latest generation 486 motherboards use PCI as well.

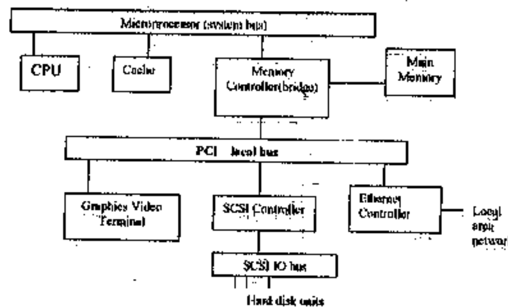


Fig 2.11 Computer system organized around PCI bus

PCI (Peripheral Component Interconnect) is a type of computer bus for attaching or inserting peripheral devices into a computer.

PCI (Peripheral Component Interconnect) is an interconnection system between a microprocessor and attached devices in which expansion slots are spaced closely for high speed operation. Using PCI, a computer can support both new PCI cards while continuing to support Industry Standard Architecture (ISA) expansion cards, However, PCI 2.0 is no longer a local bus and is designed to be independent of microprocessor design. PCI is designed to be synchronized with the clock speed of the microprocessor. PCI is now installed on most new desktop computers, not only those based on Intel's Pentium processor but also those based on the PowerPC. PCI transmits 32 bits at a time in a 124-pin connection (the extra pins are for power supply and grounding) and 64 bits in a 188-pin connection in an expanded implementation. PCI uses all active paths to transmit both address and data signals, sending the address on one clock cycle and data on the next.

The PCI specifications define two different card lengths. The full-size PCI form factor is 312 millimeters long; short PCIs range from 119 to 167 millimeters in length to fit into smaller slots where space is an issue. Like the full-size PCI, the short PCI is a high-performance I/O bus that can be configured dynamically for use in devices with high bandwidth requirements.

PCI is described as high-bandwidth and processor-independent data path between the CPU and high-

speed peripherals. The PCI spec allows for the capability to transfer up to 132 megabytes per second at a bus clock speed of 33 MHz (although the current rates being claimed by manufacturers are more commonly in the 30 Mb/sec range). This speed makes it especially suitable for high data rate applications like digital audio and video. PCI slots are found in the current generations of both PC and Macintosh personal computers.

Technically, PCI is not a bus but a bridge or mezzanine. It includes buffers to decouple the CPU from relatively slow peripherals and allow them to operate asynchronously.

PCI Bus Performance

The PCI bus provides superior performance to the VESA local bus; in fact, PCI is the highest performance general I/O bus currently used on PCs. This is due to several factors:

- Burst Mode: The PCI bus can transfer information in a burst mode, where after an initial address is provided multiple sets of data can be transmitted in a row.
- Bus Mastering: bus mastering is the capability of devices on the PCI to take control of the bus and perform transfers directly. PCI supports full bus mastering, which leads to improved performance.
- High Bandwidth Options: The PCI bus specification version 2.1 calls for expandability to 64 bits and 66 MHz speed;.
- The speed of the PCI bus can be set synchronously or asynchronously, depending on the chipset and motherboard. In a synchronized setup (used by most PCs), the PCI bus runs at half the memory bus speed; since the memory bus is usually 50, 60 or 66 MHz, the PCI bus would run at 25, 30 or 33 MHz respectively. In an asynchronous setup the speed of the PCI bus can be set independently of the memory bus speed. This is normally controlled through jumpers on the motherboard, or BIOS settings.
- Overclocking refers to changing the settings of a computer system so that the hardware runs at a faster speed than the manufacturer rated it for. Every piece of hardware in a computer system is tested and is supposed to be rated to run at a particular clock speed. When you overclock, you change the settings of the hardware so that it runs faster than what the manufacturer originally intended. Overclocking the system bus on a PC that uses synchronous PCI will cause PCI peripherals to be overclocked as well, often leading to system stability problems.

2.8 Input/Output Devices (Externals)

Input and output devices are similar in operation but perform opposite functions. It is through the use of these devices that the computer is able to communicate with the outside world. Input data may be in any one of three forms:

- Manual inputs from a keyboard or console
- Analog inputs from instruments or sensors
- Inputs from a source on or in which data has previously been stored in a form intelligible to the computer.

Computers can process hundreds of thousands of computer words or characters per second. Thus, a study of the first method (manual input) reflects the inability of human-operated keyboards or keypunches to supply data at a speed that matches the speed of digital computers. A high average speed for keyboard operation is two or three characters per second, that, when coded to form computer words, would reduce the data input rate to the computer to less than a computer word per second. Since mainframe computers are capable of reading several thousand times this amount of information per second, it is clear that manual inputs should be minimized to make more efficient use of computer time. However, as a rule, the keyboard is the normal input media for microcomputers.

Input data that has previously been recorded on paper tapes, magnetic tapes, magnetic disks, or floppy disks in a form understood by the program may also be entered into the computer. These are much faster methods than entering data manually from a keyboard. The most commonly used input devices in this category are magnetic tape units, magnetic disk drive units, and floppy disk drive units.

Output information is also made available in three forms:

- Displayed information: codes, numbers, words, or symbols presented on a display device like a cathode-ray screen

Control signals: information that operates a control device, such as a lever, aileron, or actuator

Recordings: information that is stored in a machine language or human language on tapes, disks, or printed media

Devices that display, store, or read information include magnetic tape units, magnetic disk drive units, floppy disk drive units, printers, and display devices.

2.9 Summary

The difficulty in transferring information among the units of computers largely depends on the physical distances separating them. Communication within signal computer system involves information transfer over distances of less than a meter is primarily implemented by groups of electrical wires called buses.

System Bus consists of electrical pathways, which move information between basic components of the motherboard, including between RAM and the CPU.

Main components of computer that compute all the operation are memory, registers, circuits, buses, control.

Main functions of a computer are Gather Data, Data processing, Data storage, Data movement, Control.

The interconnection between the components can have different physical configuration, depending on the number of transfer paths that are available between the processor and memory in shared memory system

Bus interconnection is the scheme of connecting various devices. There are two ways single bus in which devices are connected through one bus but due to propagation delay lies in this scheme since only one device can communicate at a time multiple bus interconnection is used.

PCI (Peripheral Component Interconnect) is a type of computer bus for attaching peripheral devices into a computer.

1.11 Glossary

Address Bus

That portion of a system bus used for the transfer of an address. Typically the address identifies a main memory location or an IO devices

Bus

A shared communication path consisting of one or a collection of lines.

Bus interconnection

Bus interconnection is the scheme of connecting various devices

Input-Output

Pertaining to either input or output or both. Refers to the movement of data between a computer and a directly attached peripheral.

Interconnection Structure

The interconnection between the components can have different physical configuration, depending on the number of transfer paths that are available between the processor and memory in shared memory system

Peripheral Component Interconnect(PCI)

Peripheral Component Interconnect is a type of computer bus for attaching peripheral devices into a computer.

System Bus

A bus used to interconnect major computer components.

2.12 Further Readings

J.P.Hayes: Computer Architecture and Organization, McGraw-Hill International.

William Stallings: Computer Organization & Architecture, Pearson Education.

M.Morris Mano: Computer System Architecture, Prentice Hall of India.

2.13 Answer to Self Learning Exercise

Question No.	Answer	Question No.	Answer
A	True	I	False

B	False	J	True
C	Address Line	K	Buses
D	Asynchronous Bus	L	Control Signals
E	False	M	True
F	True	N	True
G	Nine	O	Propagation Delay
H	AC, DR, INPR		Single and Multiple Buses

2.14 Unit End Questions

1. What is the purpose of System Bus?.
2. Write short notes on the following
    - PCI
    - Interconnection Structure
    - Computer Functions
3. What are benefits of using multiple bus architecture compared to single bus architecture.

Unit-3
DataPath Design

Structure of unit

3.0 Objective
3.1 Introduction
3.2 Datapath Design
3.3 Fixed Point arithmetic
3.3.1 Addition and Subtraction
3.3.2 Multiplication and Division
3.3.3 Self learning Exercises
3.4 Floating Point Arithmetic
3.4.1 Addition and Subtraction
3.4.2 Multiplication and Division
3.4.3 Self learning Exercises
3.5 Arithmetic Logic Unit
3.5.1 Combinational ALU
3.5.2 Sequential ALU
3.5.3 Self Learning Exercises
3.6 Summary
3.7 Glossary
3.8 Further Readings
3.9 Answer to Self Learning Exercise
3.10 Unit End Questions

3.0 Objective

This unit is concerned with computer's ALU and its arithmetic operations. ALU and its combinational and sequential implementation are discussed. There are two types o numbers – fixed point and Floating Point numbers. Integers are comes in fixed point where decimal is fixed at after last digit. Technique of arithmetic operations perform is different in two numbers. These techniques and its implementation are explained.

3.1 Introduction

Arithmetic instruction in digital computer manipulate data to produce results necessary for the solution of computational problems. These instructions perform arithmetic operations and are responsible for bulk of activity involved in processing data in a computer. Four basic operations are addition, subtraction, multiplication and division. It is possible to formulate other arithmetic functions using these basic operations. An arithmetic instruction may specify binary or decimal data. Each data may be fixed-point or floating point number. The collection of state elements, computation elements, and interconnections that together provide a conduit for the flow and transformation of data in the processor during execution is called data path .Data path or data processing part of CPU is responsible foe executing arithmetic and logical (nonnumeric) instruction on various operand. Designer of data path must be familiar with sequence of steps in particular operations that should carried out to achieve correct result. Well defined procedure steps for solution of problem is known as an algorithm.

Algorithm for addition of two fixed point numbers when negative number is in signed 2's complement representation is simple and require parallel binary adder. If negative number is in signed magnitude representation algorithm is more complex and require circuitry to add, subtract and to compare sign and magnitude of number.

3.2 Data Path Design

A datapath is a collection of functional units, such as ALUs or multipliers, that perform data processing operations.

Elements of data path.

- ALUs are just one datapath building block
- Computational Elements
- Combination Circuits
- Outputs follow inputs
- Example - ALU
- State Elements
- Sequential Circuits
- Outputs change on clock edge
- Example - Register
- Control

The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program."

Two principal concerns for computer arithmetic are

- way in which number represent
- Algorithm used for basic arithmetic operations.

Integer Arithmetic Operations

The addition of two binary numbers is computed in the same way as we compute the sum of two decimal numbers. Using the relation 0+0=0, 0+1=1+0=1 and 1+1=10, we can easily compute the sum of two binary numbers.

The subtraction of two binary numbers is computed in the same way as we compute the sum of two decimal numbers. Using the relation 0 – 0 = 0, 0 – 1 = 1 (borrow) 1 – 0 = 1 and 1 – 1 = 0, we can easily compute the subtraction of two binary numbers.

3.3 Fixed point arithmetic

In computing, a fixed-point number representation is a real data type for a number that has a fixed number of digits after (and sometimes also before) the radix point (*e.g.*, after the decimal point '.' in English decimal notation). Fixed-point number representation can be compared to the more complicated (and more computationally demanding) floating point number representation. Floating-point numbers allow you to deal with an extremely wide range of numbers: from the very small to the very large. They do this by storing the number as some digits and the position of the decimal point

Fixed-point numbers are useful for representing fractional values, usually in base 2 or base 10, when the executing processor has no floating point unit (FPU) or if fixed-point provides improved performance or accuracy for the application at hand. Most low-cost embedded microprocessors and microcontrollers do not have an FPU.

However, if you don't have a floating-point co-processor, this sort of arithmetic can be slow. If you're prepared to lose the wide range of numbers that floating-point gives, you can speed things up by fixing the position of the decimal point and using integer arithmetic operations. This is called fixed-point arithmetic If the result is out of the range that can be properly represented in the given data size it is the case of overflow(if value exceed positive limit) or underflow(if value exceed negative limit). *Saturation* is a way of handling the overflow (and underflow) in this approach value closest to the true result is taken in the range representable.

3.3.1 Fixed Point Addition and subtraction

There are three ways of representing negative fixed point binary number signed representation, one's and two's complement. Most of the computers uses two's complement method.

Addition and subtraction with signed representation of data :-

Following example shows how addition and subtraction perform -

```
    00000110 (+6)                     11111010 (-6)
+   00001101 (+13)                    00001101 (+13)
_________________                 _________________
```

00010011 (+19) 00000111 (+7)

Let two numbers are represented by a and b. If number is represented with sign then there are eight different cases which are given below –

Table : Addition and subtraction of signed magnitude numbers

Operation	Add magnitudes	Subtract Magnitude		
		when a > b	when a < b	when a = b
(+a) + (+b)	+ (a + b)			
(+a) + (-b)		+ (a - b)	- (b - a)	+ (a - b)
(-a) + (+b)		- (a - b)	+ (b - a)	+ (a - b)
(-a) + (-b)	- (a + b)			
(+a) - (+b)		+ (a - b)	- (b - a)	+ (a - b)
(+a) - (-b)	+ (a + b)			
(-a) - (+b)	- (a + b)			

first column represent eight conditions, Last column to prevent from negative zero and other columns show actual operation performed.

Steps included in addition and subtraction are given below

Algorithm :

1. if any one operand is zero then operation terminate with non zero as a result.
2. Check sign of two numbers.
3. if both have identical sign

   Add the number and attach sign of any one to result.
4. if both have different sign

   Compare the magnitudes

   if both have different magnitude then

   subtract smaller from large number.

   Attach sign of large number.

   if both have same magnitude then

   Subtract them and attach positive sign to result.

Hardware Implementation require following -

1. Two registers to hold magnitude of two numbers say A and B
2. Two flip flops As, Bs for sign of both numbers.
3.. parallel adder is needed to perform micro operation A + B
4. a comparator circuit to find A > B, A = B and A < B
5. Two parallel subtractor circuits for micro operation A - B
6. exclusive - OR gate to determine sign between As, Bs

Addition and Subtraction with 2's complement data :-

In signed representation leftmost bit of binary number is zero for positive and one for negative. So +33 and -33 are represented by 00100001 and 1010001 respectively. But in 2's complement data -33 is represented as 11011111 and +33 is represented similarly as above.

Subtraction is done by adding 2's complement of second operand and add overflow carry if generated. Hardware implementation is shown in Figure 3.1

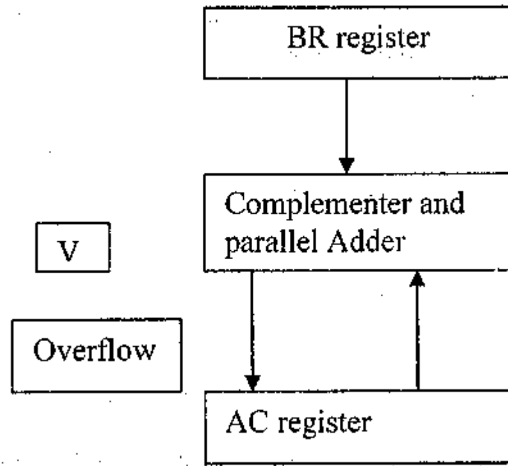


Figure3.1 Hardware for signed 2's complement additions and subtraction

Sum is obtained by adding content of register AC and BR(including sign bit also). Overflow bit V set if carry generated by addition of most significant bits otherwise V set to 0.
Subtraction operation is accomplished by adding content of AC and 2's complement of BR register. The Overflow must be checked during this operation because the two number added could have same sign.
It is much simpler to add and subtract numbers if negative number is maintained in signed 2's complement representation compared to signed magnitude addition and subtraction.

3.3.2 Mutiplication and Division

Multiplication involves the generation of partial product, one for each digit in the multiplier. These partial products are summed to produce the final product.
The partial products are easily defined when multiplier bit is zero partial product is zero. When multiplier is one the partial product is multiplicand.
The total product is produced by summing the partial product. For this operation, each successive partial product is shifted one position to left relative to proceeding partial product.
The multiplication of two n-bit binary integers result in product of up to 2n bit in length.
Product of two fixed point number in signed representation is evaluated as follows-

```
         0.110  0.75
X        1.110  -0.25
---------------------------
         0000
         0110          partial Products
         0110
         1010
---------------------------
      1110100   -0.1875
```

If least significant bit of multiplier is 1 then multiplicand is copied. otherwise zeros are copied down. The number copied in successive lines are shifted one position to left from previous number. Finally all numbers are added. and result is find sum the sign of product is positive if both multiplicand and multiplier are of same sign otherwise attach negative sign to product.

Hardware implementation

Some points should keep in mind while evaluating product-

1. It is convenient to provide an adder for summation of only two binary numbers and successively accumulate the partial product in a register.
2. Instead of shifting the multiplicand to the left, partial product is shifted to the right. When corresponding bit of multiplier is zero, there is no need to add all zeros to partial product since it will not alter the value. Hardware requirement is shown in Figure 3.2

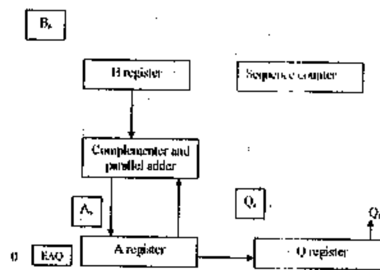


Figure 3.2 Hardware implementation of Multiplication

Initially multiplicand is in register B and multiplier in Q. The Sum of A and B form a partial product which is transferred to the EA register. Both partial product and multiplier are shifted to right. This shift will denoted by statement shr EAQ designate right shift. The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and zero 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.

Algorithm

1. Initially multiplicand is in B and multiplier in Q and Sign of multiplicand and multiplier is Bs, Qs respectively.
2. Signs are compared.
3. Register A and E are cleared.
4. Sequence counter SC is set to number of bits of multiplier.
5. After initialization, the low order bit of multiplier in Q_n is tested.
6. If it is one multiplicand in B is added to the present partial product in A.
7. If it is a zero, do nothing.
8. Register EAQ is then shifted once to the right to form the new partial product.
9. The process stops when sc = 0

Division

Binary division is simpler than decimal division since quotient digits are either 0 or 1. Example of division is given below in Figure 3.3

```
    0011
 11)1001
    0
    10
     0
    100
     11
      11
```

Hardware implementation of division is same as multiplication see figure 3.2.

-> EAQ register is shifted to left with zero inserted into Q_n and the previous value of E lost.

-> divisor is stored in B register and the double length dividend is stored in register A and B.

-> dividend is shifted to the left and the division is subtracted.

-> If E = 1 then it says A > B, quotient bit 1 is inserted into Q_n and partial remainder is shifted to the left to repeat the process.

-> if E = 0 it says A < B quotient in Q_n remains zero.

-> the value of B is then added to restore the partial remainder in A to its previous value.

-> partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed.

-> if sign of both are same, sign of quotient is plus.

-> If they are of different sign quotient sign is minus, sign of remainder is same as sign of the dividend.

Algorithm

1. Dividend is in A and q and Divisor in B.
2. Sign of result is transferred into Qs.
3. A constant is set into sequence counter SC to specify the number of bits in the quotient.
4. If A>=B Divide over-flow bit is set and operation terminated.
5. If A < B no divide overflow occur so value of dividend is restored by adding B to A.
6. division of the magnitude starts by shifting the dividend in AQ to the left with high order bit shifted into E.
7. If E = 1 then EA will be greater than EA consists 1 followed by n-1 bits while B consists of only n-1 bits.
8. Evaluate EA - B and insert 1 into Q_n.
9. If shift left operation insert 0 into E, divisor subtracted by adding its 2's complement value and carry transferred into E.
10 If E = 0it says A < B and original number is restored by adding B to A, otherwise leave 0 in Q_n.
11. This process repeat with register A holding partial remainder.
12. After n-1 times quotient is formed in register Q and remainder is in register A. sign of quotient in Q_n and sign of remainder in As.

3.3.3 Self learning Exercises

True/ False

A. There are three ways of representing negative fixed point binary number.

B. Binary division is simpler than decimal division

Fill in the blanks

C. Multiplication involves the generation of ....

D. Saturation is a way of handling the ....

3.4 Floating Point Arithmetic

Floating point arithmetic can be implemented by two loosely connected fixed point data path circuits. An exponent unit and mantissa unit.

A generic fixed point arithmetic circuit can implemented as -

Mantissa unit is responsible for addition, subtraction, multiplication and division of mantissa.

Exponent Unit can be implemented by simple circuit for addition, subtraction and comparison of exponents. Exponent comparison can be done by a comparator or by subtracting the exponent. Datapath of floating point unit is given as follows-

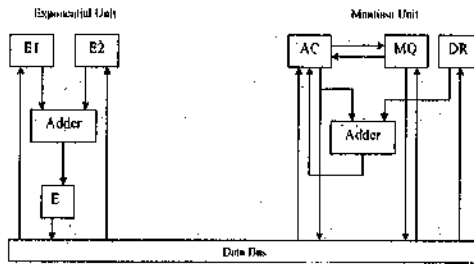


Figure 3.4 Datapath of FPU

Exponent of two number put in register E1 and E2 respectively. Adder take content of those register and compute E1 + E2 and put in register E. And mantissa is stored in AC and DR registers and adder compute sum of both of them.

All computers with floating point instruction also have fixed point instruction so it is expected to design a single ALU to execute both fixed and floating point arithmetic. This design takes the form of a fixed point arithmetic unit in which registers and adder can be partitioned into exponent and mantissa when floating point operation are being performed. In recent years it has become more common to implement fixed point and floating point unit. This separation makes it possible for fixed point and floating point instruction to be executed in parallel.

3.4.1 Floating Point Addition and Subtraction

In many high level language Floating point number specify by real declaration and fixed point number specify by integer declaration statement. Computer which have compiler of theses high level language should provide provision of floating point operation. One option for this is hardware for these operations, but if no hardware is available for these operations compiler must have package of floating point software. Hardware solution is costly than software solution.

Arithmetic operation with floating point number are more complicated and take more time than fixed point number. Floating point operations require complex hardware.

Addition or Subtraction

Addition and subtraction operation involves following phases-

1. Zero Check : A floating point number that is zero can not be normalize. If zero is used, result may also be zero. We check for zero at the beginning and terminate the process if necessary.

Addition and subtraction identical to check for zero. If either operand is zero other is reported as the result.

2. Significant Alignment: for alignment exponent make equal by shifting larger mantissa to left and increase exponent and smaller mantissa shift to right (decrement exponent). It is better to shift mantissa of smaller number. This shifting will repeat till exponent are equal.

3. Addition : Now two adjusted mantissa are added because the sign may differ the result may be zero. There is also possibility of mantissa overflow by one digit. If so, mantissa shifted right and exponent is incremented An exponent overflow could occur as a result. This will be reported and operation halted.

4. Normalization : The final step is normalize result. Normalization is done by shifting mantissa left until most significant digit is non zero. Each shift cause a decrement of exponent as a result of this exponent can underflow. Finally result is rounded and reported.

For example two numbers are -

$$.4241500 \times 10^{-2}$$
$$.1280000 \times 10^{-1}$$

Before addition/ subtraction there exponents have to made equal for this requirement. Alignment of mantissa is done by shifting one mantissa and exponent is adjusted until it is equal to the other exponent. To equalize exponent either first number shift three position left or shift second number three position right. Since shifting to right involve loss of most significant digits and shifting to left involve loss of least significant digits so shifting to left is most preferable. It only reduce accuracy where is shifting to left may cause an error.

Commonly mantissa of lesser exponent is shifted to right by the number of place equal to difference between the exponents. As soon as exponent become equal, mantissa are added after summation sum may contain overflow digit which can be corrected by shifting sum to right by single place and increment exponent.
for example if we get sum 1.9654 X 10 5
after normalization sum will be .19654 X 10 6
But in case of subtraction result may have most significant zeros as given in following example

```
  .4287 X 10 5
- .4232 X 10 5
--------------
  .0055 X 10 5
```

for full utilization of storage cells we require to store all significant digits. But in this case leading zeros are insignificant digits(since they are just to display position of decimal) so solution is to shift mantissa to right till most significant digit after decimal become non zero. For this normalize mantissa by shifting mantissa to left and decrement the exponent until non zero digit appear in first position.
Detail algorithm is given below-
For addition of mantissa first exponent are compared and shifting of mantissa required. and due to shifting exponent either incremented or decremented by one. Once mantissa align but mantissa added.

Algorithm (For addition)

1. If both arguments are zero then return zero
2. if First argument is zero then return second argument
3. if second argument is zero then return first argument
4. If both arguments are not zero then
   if exponents are equal then
   add mantissa and store resultant mantissa
   if resultant mantissa is zero then
   return zero
      otherwise
         if mantissa overflow then
            Shift mantissa to right.
            Increment exponent
         if exponent overflow then
            report overflow and return
   otherwise (if exponents are not equal)
   align the mantissa (multiply and divide mantissa by ten)by shifting one of them until the difference between exponent reduced to zero.
5. if no mantissa overflow and no exponent overflow then
      if result normalized then
         Round resultant mantissa and return
      otherwise
         repeat following steps till result not normalize
         shift mantissa left
         decrement exponent
         if exponent underflow then
            report underflow and return
         otherwise
            report result and return.

Algorithm (For Subtraction)

1) If both arguments are zero then
   return zero
2) if First argument is zero then
   return second argument with minus sign
3) if second argument is zero then
   return first argument
4) If both arguments are not zero then
   if exponents are equal then
      Subtract mantissa and store resultant mantissa
      if resultant mantissa is zero then
         return zero

otherwise

if mantissa is not zero and normalized then
Round resultant mantissa and return

otherwise

repeat following steps till result normalize
shift mantissa left
decrement exponent
if exponent underflow then
report underflow and return
otherwise
report result and return.

otherwise (if exponents are not equal)

align the mantissa (multiply and divide mantissa by ten)by shifting one of them until the difference between exponent reduced to zero.

3.4.2 Multiplication and Division

Multiplication and division do not require alignment of mantissa. Product calculated by multiplication of mantissa and addition of exponents. Division is calculated by dividing mantissa and subtracting exponents. Operation performed with mantissa are same as in fixed point numbers. So two can share the registers and circuits and operation with exponent are compare, increment, decrement, added and subtracted so exponent can be represented either signed representation, one's and two's complement.

Algorithm (For Multiplication of two normalize Floating point Number)

If single or both arguments is zero then
Return zero
otherwise
Add exponents
If exponent overflow then
Report Overflow and Return
If exponent underflow then
Report Underflow and Return
otherwise
Multiply mantissa
Normalize product mantissa(shift mantissa and update exponent).
Round normalize mantissa and return.

Algorithm (For Division of two normalize Floating point Number)

If both arguments is zero then
Return zero

If first argument is zero then
Return zero
If second argument is zero then
Return error : divide by zero
otherwise
Subtract exponents
If exponent overflow then
Report Overflow and Return
if exponent underflow then
Report Underflow and Return
otherwise
Divide mantissa
Normalize product mantissa(shift mantissa and update exponent).
Round normalize mantissa and return.

3.4.3 Self learning Exercises

True/False

E. FPU is responsible for floating point operation.

F. Floating Point operations are more complex and time consuming than Fixed-point operations

Fill in the blanks

G. Multiplication and division do not require .... of mantissa

H. Shifting mantissa left until most significant digit is non zero is known as ....

3.5 ALU

Combination of various circuits used to execute data processing instruction is Arithmetic Logic Unit(ALU). The complexity of an ALU is determined by the way in which its arithmetic instruction are realized. Two type of circuits that can be used to design ALU are given below-

Combinational Circuits: Circuits whose outputs depend only on the current inputs; hence they appear to *combine* the inputs in some way to produce the outputs.

Sequential Circuits: Circuits whose outputs depend on the both the current and past inputs; hence they use the *sequence* of inputs over time to determine the output. For example, memory circuits are inherently sequential, and a circuit which takes two binary numbers, adds them, and outputs the sum, would be combinational.

A Combinational circuit is a special case of a sequential circuit that does not have a storage capability. Synonymous with combinational circuit.

1Combinational circuits can realize ALU which perform simple fixed point addition, subtraction and word based logical operation. Much more extensive data processing and control logic is necessary to implement floating point arithmetic in hardware. Some processor having fixed point ALU employs special purpose auxiliary unit called arithmetic processor to perform floating point and other complex numerical function.

3.5.1 Combinational ALU

The ALU is a combinational logic circuit. This means it can be described using a truth table, and it can be implemented using a functionally complete set of logic gates.

Combinational ALU implement most of the CPU's fixed point operation. It combine functions of twos complement adder – subtractor with those of circuit and generates word based logic function of the form f(x,y) for example AND, XOR and NOT. They can thus implement most of CPU's fixed point data processing instructions.

3.5.2 Sequential ALU

Combinational logic can implement multiplication and division but it is generally impractical to merge these operations with addition and subtraction into a single combinational ALU reason is as follows-

- Combinational multiplication and division are costly in terms of hardware.
- They also slower than addition and subtraction circuits due to many logic levels. Since n bits combinational multiplication and division composed of n or more levels of addition and subtraction logic, therefore multiplication and division n times slower than addition or subtraction.
- number of gates in multiplication and division is also greater by factor of n.

Structure of basic sequential ALU are given below-

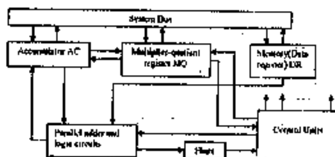


Figure 3.5 Structure of a basic sequential ALU

It is intended to implement multiplication and division using one of the sequential digit by digit shift and add/ subtract algorithm. There are word registers are used for operand storage

Accumulator AC.

Multiplier – quotient register MQ

Data Register (DR)

Accumulator mostly used to hold one of the operand of operation and result.

Multiplier – Quotient register store multiplier in case of multiplication operation and quotient in case of division operation.

Data register store multiplicand or divisor.

AC and MQ capable of left and right shifting. Additional data processing is provided by combinational ALU which can perform addition, subtraction and logical operations. Which accepts its input from AC and DR and Places result in AC.

Role of all above registers in different operation is given below-

Addition	AC := AC + DR
Subtraction	AC := AC – DR
Multiplication	AC.MQ := DR * MQ
Division	AC.MQ := MQ/DR
AND	AC := AC and DR
OR	AC := AC or DR
ex-OR	AC := AC xor DR
NOT	AC := not(AC)

DR can serve as a memory register to store data addressed by an instruction address field(ADR) then DR can be replaced by m(ADR).

3.5.3 Self Learning Exercises

True/False

I. Output of a sequential circuit depends only on current input.

J. Number of gates in multiplication and division is also greater by factor of n.

Fill in the blanks

K. Combination of various circuits used to execute data processing instruction is ....

L. Circuits whose outputs depend only on the current inputs is known as ....

3.6 Summary

Datapath or data processing part of CPU is responsible for executing arithmetic and logical (non-numerical) instruction on various operation types, including fixed point and floating point numbers. The arithmetic functions of simple machine may be limited to the addition and subtraction of fixed point numbers. Most powerful processors incorporate multiply and divide instruction and in many cases have the hardware

needed to process floating point instruction as well.

Fixed point adder and subtractors are easily constructed from combinational logic. Fixed point multiplication and division can be implemented by shift and add/subtract algorithms that resemble manual methods. The product or quotient of two km-bit numbers is found in k sequential steps. Where each step involves an m-bit shift and possibly km-bit addition or subtraction. Division is inherently more difficult then multiplication due to problem of determining quotient digits. Both multiplier and divider can be implemented by combinational logic array circuit but at a substantial increase in amount of hardware required.

Floating point operations can be implemented by an autonomous execution unit within CPU. A floating point processor is typically composed of a pair of fixed point ALUs. One of them process exponent and other to process mantissas. Special circuits are need to normalization and in case of addition and subtraction This special circuit will also perform exponents comparison and mantissa aligned.

3.7 Glossary

Algorithm

Well defined steps for solution of problem is known as an algorithm.

Combinational Circuit

A logic device whose output values, at any given instant depends only on input values at that time.

Datapath

A datapath is a collection of functional units, such as ALUs or multipliers, that perform data processing operations.

Fixed-point number

Fixed-point number is number that has a fixed number of digits after the radix point.

Saturation

Saturation is a way of handling the overflow (and underflow)

Sequential Circuit

A digital logic circuit whose output depends on the current input plus state of circuit. Sequential circuit thus posses the attribute of memory.

3.8 Further Readings

- J.P.Hayes: Computer Architecture and Organization, McGraw-Hill International.
- William Stallings: Computer Organization & Architecture, Pearson Education.
- M.Morris Mano: Computer System Architecture, Prentice Hall of India.

3.9 Answer to Self Learning Exercise

Question No.	Answer	Question No.	Answer
A	True	G	Alignment
B	True	H	Normalization
C	Partial Product	I	False
D	Underflow	J	True
E	True	K	ALU
F	True	L	Combinational Circuit

3.10 Unit End Questions

1. What are the essential elements of a number in floating point notations.?
2. How floating point operations perform.?
3. Write an algorithm for fixed point operations.
4. Describe combinational and sequential ALU in brief.

Unit-4
Processor Organization

Structur Of The Unit

4.0 Objective
4.1 Introduction
4.2 Cpu Structure And Function
4.3 Basic Processor Organization
4.3.1. Type Of Register4
4.4 General Register Or Ganization
4.5 Stack Organization
4.6 Instruction Format
4.7 Addressing Mode
4.8 Type Of Instructions
4.9 Instruction Cycle
4.10 The Pentium Processor
4.12 The Prowerpc Procerssor
4.13 Summary
4.14 Glossary
4.15 Further Readings
4.16 Answers To Self Learning Exepceses
4.17 Unit And Questions

4.0 OBJECTIVE:

In this unit you are able to learn CPU operations how they work and different registers of CPU. Different phase of instruction cycle, mode of addressing and computer instruction types. Pentium processor primary operating modes, RISC microprocessor architecture Evaluation of Pentium processor and PowerPC.

4.1 INTRODUCTION:

The central processing unit or CPU for short is the brain of a computer. It reads and executes program instructions, performs calculations, and makes decisions. The CPU is responsible for storing and retrieving information on disks and other media. It also handles information on from one part of the computer to another like a central switching station that directs the flow of traffic throughout the computer system.

4.2 CPU STRUCTURE AND FUNCTION

The part of a computer that perform bulk of data processing operation is called CPU . The primary function of a processor, such as the CPU of a computer, is to execute sequence of instructions stored in main memory, which is external to the CPU.

The CPU must first fetch the instruction from this memory before it can be executed. The sequence of operations involved in processing and instruction constitutes an instruction cycle the instruction cycle can be sub divided into two major phases the fetch cycle and the execute cycle the instruction is obtain from the main memory during the fetch cycle.

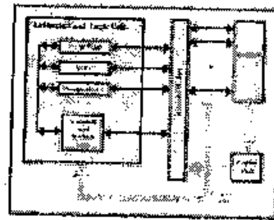
The execute cycle includes decoding the instruction, fetching any required operands, and performing the operations specified by the instruction operation code (opcode). The behavior of the CPU during the instruction cycle is defined by sequence of microoperations . An operation performed in register is called microoperation.

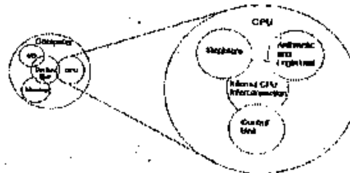
The time tcpu required for the shortest well defined CPU microoperation is defined to be the CPU cycle time and is the basic unit of the time for majoring all the CPU actions the reciprocal of tcpu is the CPU clock rate and generally major in megahertz.

In addition to execute programs the CPU supervise other system components by special control lines. The CPU directly or indirectly controls IO operations such as data transfer between IO device and main memory.

In an interrupt the CPU suspend the execution of the program that it is executing and transfer the control to the appropriate interrupt-handling program.

4.3 BASIC PROCESSOR ORGANIZATION:





Simplified view of a CPU

o Arithmetic and logic unit – perform computation or processing of data

o Control unit - controls the movement of data instructions in and out of the CPU, and controls the operation of the ALU.

o Registers - Is small amount of internal memory.

Despite the great improvement in the circuit technology over the years, almost all CPU design based on the following factors—

1. The CPU should be fast(measured by the CPU cycle time tcpu).
2. The main memory of large capacity is needed to store program and data required by the CPU.

The MM(main memory) speed is measured by memory cycle time t_M. which is the minimum time elapse between two successive read or write operations. The ratio t_M/t_{CPU} is typically ranges from 1 to 10. The CPU consist storage devices called register. The program execution is implemented as follows:

1. Fetch the instruction from the memory.
2. Decode the instruction.
3. Read the effective address from the memory.
4. Executive the instruction.

Self learning exercise

1. What are the main components of CPU?
2. What is BUS?
3. What is Micro operation?

4.3.1 TYPE OF REGISTERS

1. Data registers (DR)

Used only to hold data and cannot be used in the calculation of the operand address.

2. Address registers (ADR)

Hold the address of the instruction. May be somewhat general-purpose, or may be devoted to a particular addressing mode.

Control and Status Registers

There are a variety of CPU registers that are used to control the operation of the CPU.

Most of these, on most machines, are not visible to the user. Four registers are essential to instruction execution:

Program counter (PC): contains the address of an instruction to be fetched.

Instruction register (IR): contains the instruction most recently fetched.

Memory address register (MAR): contains the address of a location in memory.
Memory buffer register (MBR): contains a word of data to be written to memory or the word most recently read.
Program Counter (PC): keeps track of the instruction in the program stored in memory. PC holds the address of the instruction to be executed next and in incremented each time an instruction is fetched from memory. All CPU designs include a register or set of registers, often known as the program status word (PSW). The PSW typically contains condition codes plus other status information.
Common flags include the following:

- Sign
- Zero
- Carry
- Equal
- Overflow
- Interrupt enable/disable
- Supervisor

It is common to dedicate the lowest few hundred or thousand words of memory for control purposes. The designer must decide how much control information should be in registers and how much in memory. The usual trade-off of cost vs. speed arises. CPU design and operating system design are closely linked. If the CPU designer has a functional understanding of the operating system to be used, than the register organization can be tailored to the operating system.

Figure 2: Different types of register organization

4.4 GENERAL REGISTER ORGANIZATION:

Generally memory location are needed for storing pointers, return address, temporary results and partial product during multiplication. Having to refer to memory locations for such applications is a time consuming because memory assess is such a time consuming process. So it is more convenient and efficient to store this intermediate value in processor registers.
When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various micro-operations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic and shift micro-operation in the processor.
A Bus organization for seven CPU registers: —

Figure 3: General register organization based CPU.

In the general register organization based CPU most of the operation are performed in general purpose register. In this organization we use 7 general purpose register R1 to R7 . one external input . The output of each register is connected to true multiplexer (MUX) to form the two buses A & B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses forms the input to a common ALU. The operation selected in the ALU determines the arithmetic or logic micro-operation that is to be performed. The result of the micro-operation is available for output and also goes

into the inputs of the registers. A decoder selects the register that receives the information from the output bus. The decoder activates one of the register load inputs, thus providing a transfer both between the data in the output bus and the inputs of the selected destination register.

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the systems.

R_1 ® $R_2 + R_3$

(1) MUX A selection (SEC A): to place the content of R2 into bus A

(2) MUX B selection (sec B): to place the content of R3 into bus B

(3) ALU operation selection (OPR): to provide the arithmetic addition (A + B)

(4) Decoder destination selection (SEC D): to transfer the content of the output bus into R_1

These form the control selection variables are generated in the control unit and must be available at the beginning of a clock cycle. The data from the two source registers propagate through the gates in the multiplexer and the ALU, to the output bus, and into the destination registers, all during the clock cycle intervals.

Control Word:

There are 14 bit binary selection inputs in the units, and their combined value specified a control word. It consists of four fields three fields contain three bits each, and one field has five bits. The three bits of SEL A select a source register for the A input of the ALU. The three bits of SEL B select a source register for the B input of the ALU. The three bit of SEC D select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specifies a particular micro-operation.

Symbolic Designation

Micro Operation	SEC A	SEC B	SEL D	OPR	Control Word

Table: Encoding of Register selection fields.

Binary	Code	SEL A	SEL B SEL D
000	Input	Input	None
001	R_1		
010	R_2		
011	R_3	S	S
100	R_4	A	A
101	R_5	M	M
110	R_6	E	E
111	R_7		

Table: Encoding of ALU operation

OPR & elect	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	Add A + B	ADD
00101	Subtract A-B	SUB
00110	Decrement A	DEC A
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA

11000	Shift left A	SHLA

Examples of Micro-operation for the CPU

R_1 ® $R_2 - R_3$	R_2	R_3	R_1	SUB	010	011	001	00101
R_4 ® R_5 V R_5	R_4	R-	R_4	OR	100	101	100	0101
R_6 ® R_6 + 1	R_6	-	R_6	MCA	110	000	110	00001
R_7 ® R_1	R_1	-	R_7	TSFA	001	000	111	00000
Output ® R_2	R_2	-	None	TSFA	010	000	000	00000
Output ® Input	Input	-	None	TSFA	000	000	000	00000
R_4 ® SHL R4	R_4	-	R_4	SHLA	100	000	100	11000
R_5 ® 0	R_5	R_5	R_5	XOR	101	101	101	01100

4.5 STACK ORGANIZATION

A useful feature that is included in the CPU of most computers is a stack or last-in first out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation a stack can be companied to a stack of trays.

The stack in Digital Computer is essentially a memory unit with an address register that can count only (after an initial value is loaded into it.) The register that holds the address for the stack is called a Stack Pointer (SP) because its values always points at the top item in the stack.

The two operations – PUSH (insert)
- POP (delete)

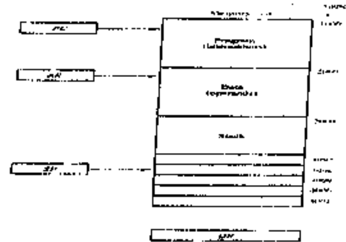


Figure 4: Stack organization based CPU

Register Stack:- A stack can be placed in a portion of a large memory as it can be organized as a collection of a finite number of memory words as register.

In a 64- word stack, the stack pointer contains 6 bits because $2^6 = 64$.

The one bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty. DR is the data register that holes the binary data to be written into on read out of the stack.

Initially, SP is decide to 0, EMTY is set to 1, FULL = 0, so that SP points to the word at address 0 and the stack is masked empty and not full.

PUSH	SP ® SP + 1	increment stack pointer
	M [SP] ® DR	unit item on top of the Stack
	It (SP = 0) then (FULL ® 1) check it stack is full	
	EMTY ® 0	mask the stack not empty.
POP	DR ® [SP]	read item trans the top of stack
	SP ® SP –1	decrement SP
	It (SP = 0)	then (EMTY ® 1) check it stack is empty
	FULL ® 0	mask the stack not full

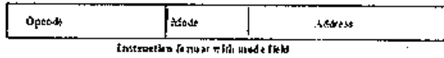
4.6 INSTRUCTION FORMATS:

The most common fields found in instruction format are: -

(1) An operation code field that specified the operation to be performed

(2) An address field that designates a memory address or a processor registers.

(3) A mode field that specifies the way the operand or the effective address is determined.



Computers may have instructions of several different lengths containing varying number of addresses. The number of address field in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organization.

(1) Single Accumulator organization Example : ADD X AC ® AC + M [×]

(2) General Register Organization Example : ADD R1, R2, R3 R ® R2 + R3

(3) Stack Organization Example : PUSH X and POP X

Three address Instruction

Computer with three addresses instruction format can use each address field to specify either processor register are memory operand.

To Evaluate following expression using. Three address instruction use following step

X = (A + B) * (C + A)

ADD R_1, A, B A_1 ® M [A] + M [B]

ADD R_2, C, D R_2 ® M [C] + M [B]

MUL X, R_1, R_2 M [X] R_1 * R_2

The advantage of the three address formats is that it results in short program when evaluating arithmetic expression. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Two Address Instruction

Most common in commercial computers. Each address field can specify either a processes register on a memory word.

To evaluate following expression using two address instruction use following step

X = (A + B) * (C + A)

MOV	R_1, A	R_1 ® M [A]
ADD	R_1, B	R_1 ® R_1 + M [B]
MOV	R_2, C	R_2 ® M [C]
ADD	R_2, D	R_2 ® R_2 + M [D]
MUL	R_1, R_2	R_1 ® R_1 * R_2
MOV	X_1 R_1	M [X] R_1

It used an implied accumulator (AC) register for all data manipulation. For multiplication/division, there is a need for a second register. All operations are done between the AC register and a memory operand. It's the address of a temporary memory location required for storing the intermediate result.

To Evaluate following expression using One address instruction use following step

X = (A + B) * (C + A)

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)
ML	T	AC ® AC + M (T)
STORE	X	M [X] AC

Zero – Address Instruction

A stack-organized computer does not use an address field for the instruction ADD and MUL. The PUSH & POP instruction, however, need an address field to specify the operand that communicates with the stack (TOS ® top of the stack)

To Evaluate following expression using Zero address instruction use following step

```
        X = (A + B) * (C + A)
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 ® (C + D) * (A + B)
POP         X       M [X]  TOS
```

4.7 ADDRESSING MODES

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer register as memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction between the operand is activity referenced. Computer use addressing mode technique for the purpose of accommodating one or both of the following provisions.

(1) To give programming versatility to the uses by providing such facilities as pointer to memory, counters for top control, indexing of data, and program relocation.

(2) To reduce the number of bits in the addressing fields of the instruction.

There are following types of addressing mode techniques are used –

1. Implied mode : In this mode the address field is not required in the instruction format .
2. Immediate mode : operand are implicitly use in the definition so no need of address field ,stack operation are the example of this type of mode .
3. Register mode : operand are reside in the register and directly use from the register so also no address field is use .
4. Register indirect mode: the value of the register gives the address of the operand.
5. Direct Address mode : The address part of the instruction format give the address of the operand where it reside in the memory .
6. Indirect Address mode : the address part of the instruction format give address where the address is store .
7. Auto increment/ Auto decrement mode : this mode is similar to register indirect mode but register value is increment in case of auto increment after operation and incase of auto decrement register value is decrement after operation .

Similarly other modes are Relative Address mode ,Indexed Addressing mode in which PC value is updated .

Relative Address: The content of PC is added to the address part of the instruction in order to obtain the effective address Ex :- PC = 825 + 1 + 24

Indexed addressing Mode:- The content of index register is added to the address part of the instruction in order to obtain the effective address.

Base Register Addressing Mode:-The content of a base register is added to the address bank of the instruction to obtain the effective address.

Self learning exercise

4. What are the different type of addressing mode ?

4.8 TYPE OF INSTRUCTIONS

Computer provides an extensive set of instructions to give the user the flexibility to carryout various computational task. Most computer instruction can be classified into three categories.

(1) Data transfer instruction

(2) Data manipulation instruction

(3) Program control instruction

Data transfer instruction cause transferred data from one location to another without changing the binary instruction content. Data manipulation instructions are those that perform arithmetic logic, and shift operations. Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer.

(1) Data Transfer Instruction

Data transfer instruction move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processes registers, between processes register & input or output, and between processes register themselves.

(Typical data transfer instruction)

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

(2) Data Manipulation Instruction

It performs operations on data and provides the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types.

(a) Arithmetic Instruction
(b) Logical bit manipulation Instruction
(c) Shift Instruction.

(a) Arithmetic Instruction

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	Add
Subtract	Sub
Multiply	MUL
Divide	DIV
Add with Carry	ADDC
Subtract with Basses	SUBB

(b) Logical & Bit Manipulation Instruction

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-Or	XOR
Clear Carry	CLRC
Set Carry	SETC
Complement Carry	COMC
Enable Interrupt	ET

(c) Shift Instruction

Instructions to shift the content of an operand are quite useful and one often provided in several variations. Shifts are operation in which the bits of a word are moved to the left or right. The bit-shifted in at the and

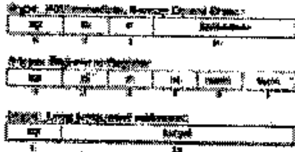
of the word determines the type of shift used. Shift instruction may specify either logical shift, arithmetic shifts, or rotate type shifts.

Name	Mnemonic
Logical Shift right	SHR
Logical Shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate mgmt through carry	RORC
Rotate left through carry	ROLC

(3) Program control instruction:

Program control type instruction include jump, branch, test compare types instructions in which program counter value updated.

In the given diagram immediate ,register to register and Jump instruction format is shown.



4.9 INSTRUCTION CYCLE

An instruction cycle includes the following sub cycles:

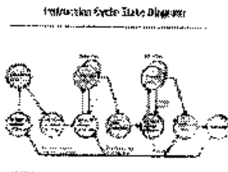
1. Fetch: read the next instruction from memory into the CPU.
2. Execute: interpret the opcode and perform the indicated operation.
3. Interrupt: if interrupts are enable and in interrupt has occurred: save the current process state and service the interrupt. The instruction cycle will now be elaborated upon.

The Indirect Cycle

The execution of an instruction may involve one or more operands in memory, each of which requires a memory access. Further, if indirect addressing is used (effective address = [memory]), the additional memory accesses are required. We can think of fetching indirect addresses as one more instruction subcycle the main line of activity consists of alternating instruction fetch and instruction execute activities. After an instruction is fetched, it is examined to see if any indirect addressing is involved. If so, the required operands are fetched using indirect addressing.

Following execution, an interrupt may be processed before the next instruction fetch.

1. Once an instruction is fetched, its operand specifiers must be identified.
2. Each input operand in memory is then fetched, and this process may require indirect addressing.
3. Register based operands need not be fetched.
4. Once the opcode is executed, a similar process may be needed to store the result in memory.



Data Flow

The exact sequence of events during an instruction cycle depends on the design of the CPU.

Assume that a CPU employs a memory address register (MAR), a memory buffer register (MBR), a

program counter (PC), and an instruction register (IR). Instruction Fetch Cycle: an instruction is read from memory.

a. PC contains address of next instruction to be fetched.
b. This address moved to MAR
c. Address placed on address bus
d. Control unit requests a memory read
e. Result placed on the data bus, copied to MBR, and then moved to IR
f. PC is incremented by 1, preparing for next fetch

Once the instruction fetch cycle is over, the control unit examines the contents of the IR to determine if it contains an operand specifies using indirect addressing. If so, an indirect cycle is performed.

Indirect Cycle

The rightmost N bits of MBR, which contains the memory reference portion of the instruction, are transferred to the MAR.The control unit requests a memory read. Result (the address of operand) moved to MBR.

Execute Cycle

The fetch and indirect cycles are simple and predictable. The execute cycle takes many forms, because the forms depends on which of the various machine instructions is in the IR. This cycle may involve transferring data between registers, read or write from memory or I/O, and/or the invocation of the ALU.

Interrupt Cycle

Like the fetch and indirect cycles, the interrupt cycle is simple and predictable. Current contents of PC saved to allow resumption after interrupt. Thus, contents of PC are transferred to MBR to be written to memory. The special memory location reserved for this purpose is loaded into the MAR from the control unit (i.e. stack pointer).

PC is loaded with the address of the interrupt routine. Next instruction will be at beginning of interrupt routine.

4.10 THE PENTIUM PROCESSORS

The Pentium series is an excellent example of Complex Instruction Set Computer (CISC) design. The PowerPC is a direct descendant of IBM 801, one of the best designed RISC systems on the market.

Pentium

Intel has ranked the number one maker of microprocessors for decades.

8080: The first's general purpose microprocessor . this is 8-bit machine ; with an 8 bit data path in memory .

8086: A far more powerful 16-bit machine. In addition to wide data path and larger registers, the 8086 supported an instruction cache ,or queue ,that perfected a few instructions before they are executed.

80286 The extension of 8086 enabled addressing a 16-Mbit memory instead of just 1Mbyte.

IA-32 80386 Processor

Eight 32-bit register mark with (R0 – R7) are general purpose register .It is used to hold operand Architecture based on memory model. Different area combination in memory is called *segment* Code

Segment (CS)
–Hold program instruction ,Stack Segment (SS)
–Contains processor stack, Data Segment (DS)
–To hold operand data. Processor used segment register CS,SS, DS and ES to fetch code, stack and two data segment.

80386 and 80486 Processor

1.1 80386 is the first processor which implement IA-32 architecture
80386 is the first processor which implement IA-32 architecture .80486 contains integer processing unit and floating point .80486 memory support is similar to 80386
80486 allows *parallelism* and *pipelining*

- 486 contains integer processing unit and floating point
- 80486 memory support is similar to 80386
- 80486 allows *parallelism* and *pipelining*

Pentium Processor

Upgraded 80486

The power is twice 80486 CISC architecture which achieved high performance by using RISC characteristic processor, two instruction be executed in one clock cycle
Use simple dynamic branch prediction form Select direction either to choose last branch executed Maximum instruction number can be executed in one cycle is twice

- The power is twice 80486

Pentium Pro Processor

Increase super scalar and the ability to execute instruction without sequential Super scalar factor, is also known as maximum instruction can be execute in one clock cycle is in Pentium Pro Processor
Super scalar operation is supported by many execution unit including for integer operation and floating point unit. Ability to execute instruction in different sequence with data stated in program is fetch from memory.
Pentium Pro to be used in multiprocessor system
Pentium II and III Processor
Pentium II Processor add MMX instruction .MMX instruction prepare processor as parallel in multimedia operation towards pixel which describe graphical data.
Pentium III Processor introduce vector instruction (SIMD) .Is an instruction to process vector operation at floating point data

Pentium 4 Processor

- Clock rate between 1.3 to 1.5 GHz
- IA-32 instruction set fully supported including MMX and SSE instruction
- Cache instruction to hold path segment execution, decoded instruction known as trace
- Trace can be more than one branch in original program
- If the path is repeated, execution will be much faster
- Test will be done to ensure that the same branch is fetched when trace is repeated
- CISC architecture which achieved high performance by using RISC characteristic processor
- It include additional floating point and other enhancements for multimedia.

Itanium : The new generation Intel Processor make use of 64 bit organization with the IA –64 architecture

Data type of Pentium

- General - byte (8), word (16), doubleword (32), and quadword (64). signed integers are in 2's complement representation. Pentium uses little endian style representation.
- Floating point - single precision (32), double precision (64), extended double precision (80)

Registers Pentium

- General - Eight 32 bit general purpose registers - EAX, EBX, ECX, EDX, ESP, EBP, ECI, and EDI. The low 16 bits of each of these registers act as 16 bit registers - AX, BX, CX, DX, SP, BP, CI, and

DI. The lower and higher 8 bits of each of these 16 bit registers are also identified as registers - AL, BL, CL, DL, AH, BH, CH, and DH.

- Floating Point - Eight registers of 64 bit floating point numbers FP0 to FP7.
- Multimedia - Eight 64 bit multimedia registers MM0 to MM7.
- Segment - Six 16 bit segment selectors that index into segment tables - CS, SS, DS, ES, FS, and GS. CS register references the segment containing the instruction being executed. SS register references the segment containing the a user-visible stack. The remaining segment registers enable the user to reference upto four separate data segments at a time.
- Flags register contains condition codes and various mode bits.
- Instruction Pointer (IP) - address of the current instruction

Addressing mode of Pentium

- Immediate: Operand = A
- Register operand: LA = R
- Displacement: LA = (SR) + A
- Base: LA = (SR) + (B)
- Base with displacement: LA = (SR) + (B) + A
- Scaled index with displacement: LA = (SR) + (I) x S + A
- Base with index and displacement: LA = (SR) + (B) + (I) + A
- Base with scaled index and displacement: LA = (SR) + (I) x S + (B) + A
- Relative: LA = (PC) + a

where

LA = linear address

(X) = contents of X

SR = segment register

PC = program counter

A = contents of an address field in the instruction

R = register

B = base register

I = index register

S = scaling factor

Instruction Format

Pentium

This is a two address ISA, which means one of the source operands in some operations is also the destination. The length of the instruction is not fixed. It has a variable number of bytes.

- Instruction Prefix: Each instruction can have zero to four prefixes. The prefix overrides the ususal interpretation of the instruction.
- Opcode - one or two bytes
- ModR/m: This byte provide addressing information. The ModR/m byte specifies whether an operand is in a register or in memory.
- SIB: SIB byte specifies the full addressing mode. The SIB byte consists of three fields: The Scale field (2 bits) specifies the scale factor; the Index field (3 bits) specifies the index register; the Base field (3 bits) specifies the base register.
- Displacement: If the ModR/M specifies that the address calculation requires a displacement, the one, two, or four bytes is contained in the instruction.
- Immediate: Provides the value of an 8, 16, 32 bit operand.
- Figure... Pentium numeric data formats-

Operations Pentium

Data Movement

- MOV: Move operand between registers or between register and memory
- PUSH: Push operand onto stack
- PUSHA: Push all registers on stack
- MOVSX: Move byte, word, dword, sign extended
- LEA: Load effective address
- IN, OUT: Input, output operand from I/O space

Arithmetic

- ADD: Add operands
- SUB: Subtract operands
- MUL: Unsigned integer multiplication
- IDIV: Divide operands

Logical

- AND: And operands

604: Intended for desktop computer computers and low end servers . again it is 32 bit machine but it use much more advance super scale design technique to achieve greater performance .

620: Intended for high end servers . the first member of PowerPC family to implement a full 64 bit architecture ,including 64 bit registers and data path .

740/750 (G3): This processor integrate two levels of cache In the main processor chip, providing significant performance improvement over a comparable machine with off chip cache organization.

G4 and G5 : Increase parallelism ad internal speed of processor chip. A complete description of the PowerPC ISA can be obtained from the IBM site.

And other Examples are MIPS, and SPARC.

Data Types of powerPC

- General - byte (8), halfword (16), word (32), and double word (64). PowerPC can operate in little endian or big endian mode.
- Floating point - single precision (32), double precision (64)
- Byte string - 0 to 128 bytes in length .

Registers

PowerPC

- General: Thirty two 64 bit general purpose registers R0 to R31.

- Exception Register (XER): Reports exceptions in integer arithmetic operations.
- General: Thirty two 64 bit general purpose registers for all floating point operations FPR0 to FPR31.
- Floating point status and control register (FPSCR): 32-bit register that control the operation of floating point quantities.
- Condition register: Consists of eight 4-bit condition code fields.
- Link register: Used in conditional branch instruction and for call / return.
- Count: Used to control an iteration loop.

Addressing Modes

Load/Store Addressing

- Indirect: EA = (BR) + D
- Indirect indexed: EA = (BR) + (IR)

Branch Addressing

- Absolute EA = I
- Relative: EA = (PC) + I
- Indirect: EA = (L / CR)

Fixed-point Computation

- Register: EA = GPR
- Immediate: Operand = I

Floating Point Computation

- Register: EA = FPR

EA = effective address
(X) = contents of X
BR = base register
IR = index register
L / CR = link or count register
GPR = general purpose register
FPR = floating point register
D = displacement
I = immediate value
PC = program counter

Instruction format

All instructions in the PowerPC are 32 bits long and follow a regular format. The first 6 bits of an instruction specify the operation to be performed. For all load/store, arithmetic, and logical instructions, the opcode is followed by two 5-bit register references, enabling 32 general purpose registers to be used.

Operations of PowerPC

Data Movement

- lwzu: Load word and zero extend to left; update source register
- ld: Load double word

2. bl: branch to target address and place effective address of instruction following the branch into the Link Register
3. bc: Branch conditional on Count Register and'or on bit in Condition Register
4. sc: System call to invoke an operating system service
5. trap: Compare two operands and invoke system trap handler if specified conditions are met

4.12 SUMMARY:

Processor organizations are three type, accumulator base CPU, general register organization and stack organization. The main register of accumulator based CPU is accumulator (AC), in case of general registration all the operation perform in general purpose registers. stack organization use memory Stack. Instruction cycle uses general two phases fetch cycle and execute cycle. In fetch cycle instruction are fetch

from memory and in execute cycle instruction are executed.

4.13 GLOSSARY :

Accumulator - A special storage register associated with the arithmetic logic unit for storing the results of steps in a calculation or data transfer.

Arithmetic Logic Unit (ALU) - The part of the CPU where arithmetic and logic operations are performed. Sometimes called the arithmetic unit.

Arithmetic Operator - The arithmetical signs of addition, subtraction, division and multiplication as used by a given programming language.

Central Processing Unit (CPU) - The main part of the computer, its 'brain', consisting of the central memory, arithmetic logic unit and control unit. Also called the central processor.

Chip - An alternative name for an integrated circuit.

Clock - A special circuit that sends pulses of current to the CPU and other computer components.

Control Unit (CU) - That part of the computer which accesses instructions in sequence, interprets them and then directs their implementation.

Fetch-Execute Cycle - Refers to the process whereby the control unit must first fetch an instruction from main memory before it can execute (interpret) that instruction.

Instruction Decoder - Complex circuitry in the CU designed to decode (interpret) any instruction in the computer's machine code repertoire.

Instruction Register (IR) - A special register in the CPU that holds the bit pattern corresponding to the next instruction to be performed within the CPU. The Control Unit accesses this register to decide which circuits need to be activated.

Instruction Set - A set of assembly language mnemonics which represent the machine code of a particular computer.

Logical Operator - Name given to the logical symbols for "greater than", "less than", etc., as used by a particular high-level language. Also called relational operators.

Memory Address Register (MAR) - When another instruction is needed in the IR, or a value is to be loaded into the accumulator, or an operand is needed to perform some arithmetic or logic instruction, this register contains the memory address where the desired information can be found. It also serves as a pointer to the location in memory where the contents of some CPU register is to be stored.

Memory Buffer Register (MBR) - This register serves as an interface between the CPU and main memory. Anything needed by the CPU (instruction or data) is first placed here before it goes to its final destination (such as the accumulator, IR, PC or other registers). Also, anything in the CPU that is to be stored in main memory comes here first before being copied into the main memory at the location specified by the address containd in the MAR.

Memory Unit - Part of the computer where data and instructions are held. (Also known as main memory, main store, central memory, immediate access memory.)

Program Counter (PC) - This CPU register always contains the memory address where the next instruction to be performed by the CPU can be found. Its contents is copied into the MAR before an instruction is fetched from the main memory. While the instruction is being fetched, the Control Unit updates the contents of the PC so that it will again point to the next instruction to be performed.

Register - A special location, which is sometimes protected, used for specific purposes only. Example: accumulator, program counter.

4.14 FURTURE READING

J. P. Hayes, Computer Architecture and Organization, McGraw-Hill, New York, 1998.

W. Stallings, Computer Organization and Architecture: Designing for Performance, NJ, 1996.

Morris mano ; Computer system Architecture

4.15 ANSWERS TO SELF LEARNING EXERCISES

1. The main component of CPU are ALU, CU and register set .
2. Bus is a group of wire use to transfer data from one component to another .

3. operation perform in register is called microoperation.

4. direct ,indirect, immediate ,implied ,register direct ,register indirect, auto increment auto decrement are the different addressing mode.

4.16 UNIT END QUESTIONS

1. Solve the following Expression using one address ,two address ,three address ,Zero address X=(A*B)+(C*D)*E
2. Explain different type of registers.
3. Explain different types of addressing mode. Explain each with example.
4. Explain General register organization based CPU.
5. What is instruction cycle? Explain different stages of instruction cycle.
6. Explain power PC and Pentium processor. Give example of each type of processor.
7. What are the different types of CPU organization?
8. What is micro operation? Explain logical micro operations.
9. How we represent number in fixed point and floating-point numbers.

- BTS: Bit test and set. The instruction copies the current value of a bit to flag CF and sets the original bit to 1
- BSF: Bit scan forward. Scans a word or dword for a 1-bit and stores the number of the first 1-bit into a register
- SHL / SHR: Shift logical left or right
- SAL / SAR: Shift arithmetic left or right
- ROL / ROR: Rotate left or right
- SETcc: Sets a byte to zero or one depending on any of the 16 conditions defined by status flags

Control Transfer

- JMP: Unconditional jump
- CALL: Transfer control to another location. The address of the next instruction following the CALL is placed on the stack
- JE / JZ: Jump if equal / zero
- LOOPE / LOOPZ: Loops if equal / zero. Conditional jump using a value stored in register ECX. The instruction first decrements ECX before testing ECX for the branch condition
- INT / INTO: Interrupt / Interrupt if overflow. Transfer control to an interrupt service routine

String Operations

- MOVS: Move byte, word, dword string. The instruction operates on one element of a string, indexed by registers ESI and EDI. After each string operation, the registers are automatically incremented or decremented to point to the next element of the string.

Load byte, word, dword of string (1 byte per digit) and packed (1 byte per 2 digits) representation.

4.11 THE POWERPC PROCESSOR:

In 1975, IBM started the 801-minicomputer project that launched the RISC movement. In 1986, IBM developed a RISC workstation, the RT PC, which was not a commercial success. In 1990, introduced the IBM RISC/6000 was a RISC like super scale machine and marketed that as a high performance workstation. IBM began to refer to this as the POWER architecture.

IBM then entered into an alliance with Motorola the developer of the 68000 series for Apple computers. The result of this alliance was the series of microprocessors that implement the PowerPC architecture. This architecture derive as a power architecture. The resulting power PC architecture is a super scalar RISC system. The processors in the series were:

601 : It is a 32 bit machine,

603 : Intended for low end desktop and portable computer s. it is also 32 bit machine ,compatible in performance with 601, but low cost and more efficient implementation.

- lmw: Load multiple word; load consecutive words into contiguous registers from the target register through general purpose register

- lswx: Load a string of bytes into registers beginning with target register; 4 bytes per register; wrap around from register 31 to register0

Arithmetic

- add: Add contents of two registers and place in third register
- subf: Subtract contents of two registers and place in third register
- mullw: Multiply low order 32 bit contents of two registers and place 64 bit product in third register
- divd: Divide 64-bit contents of two registers and place in quotient in third register
- lfs: Load 32 bit floating point number from memory, convert to 64 bit format, and store in floating point register
- fadd: Add contents of two registers and place in third register
- fmadd: Multiply contents of two registers, add the contents of a third, and place result in fourth register
- fcmpu: Compare two floating point oerands and set condition bits

Logical

- cmp: Compare two operands and set four condition bits in the specified
- crand: Condition register AND: two bits of the Condition Register are ANDed and the resut placed in one of the two bit positions
- and: AND contents of two registers and place in third register
- cntlzd: Count number of consecutive 0 bits starting at bit zero in source register and place count in destination register
- rldic: Rotate left doubleword register, AND with mask, and store in destination register
- sld: Shift left bits in source register and store in destination register

Control

1. b: Unconditional branch

4.16 UNIT END QUESTIONS

1. Solve the following Expression using one address ,two address ,three address ,Zero address X=(A*B)+(C*D)*E
2. Explain different type of registers.
3. Explain different types of addressing mode . Explain each with example.
4. Explain General register organization based CPU.
5. What is instruction cycle? Explain different stages of instruction cycle.
6. Explain power PC and Pentium processor. Give example of each type of processor.
7. What are the different types of CPU organization?
8. What is micro operation? Explain logical micro operations.
9. How we represent number in fixed point and floating-point numbers.
10. Evaluate expression X=(A*B)+ (C*D) using
   a. TWO ADDRESS INSTRUCTION
   b. ONE ADDERSS INSTRUCTIONS
11. Explain the main operation of a stack.
12. What is the main function of SP,PC,MBR;MAR.

UNIT-5
Reduced Instruction Set Computer

Structure Of The Unit

5.0 Objective
5.1. Introduction
5.2. Risc/Cisc Evolution Cycle
5.3. Riscs Design Principles
5.4. Risc Characteristics
5.5. Overlapped Register Windows
5.6. Instruction Execution
5.7. Risc Pipelining
5.8. Riscs Vs Ciscs
5.9. Risc Architecture And Cisc Architecture
5.9.1 Sparc
5.9.2 Mips
5.9.3 Mips R4000
5.9.4 Powerpc
5.9.5 Pentium
5.10. Summary
5.11. Glossary
5.12. Further Readings
5.13. Answers To Self Learning Exercises
5.14. Unit End Questions

5.0 Objective:

In this unit we study reduced instruction set computers (RISCs). These machines represent a noticeable shift in computer architecture paradigm This paradigm promotes simplicity rather than complexity. The RISC approach is substantiated by a number of studies indicating that assignment statement, conditional branching, and procedure calls/return represent. These studies showed also that among all operations, procedure calls/return are the most time-consuming. Based on such results, the RISC approach calls for enhancing architectures with the resources needed to make the execution of the most frequent and the most time-consuming operations most efficient.

5.1 INTRODUCTION:

In the 50 years since the development of the first electronic computer, the architecture of computers the structure of the machine's hardware and software and the interactions between the two and with the world has evolved. Computer designers have worked with the available technology and information theory to create ever faster, ever more capable machines. These machines represent a noticeable shift in computer architecture paradigm. This paradigm promotes simplicity rather than complexity. The RISC approach is substantiated by a number of studies indicating that assignment statements, conditional branching, and procedure calls/return represent more than 90% and that complex operations such as long division represent only about 2% of the operations performed in a typical set of benchmark programs. These studies showed also that among all operations, procedure calls/return are the most time-consuming. Based on such results, the RISC approach calls for enhancing architectures with the resources needed to make the execution of the most frequent and the most time-consuming operations most efficient. The seed for the RISC approach started as early as the mid- 1970s. Its real-life manifestation appeared in the Berkeley RISC-I and the Stanford MIPS machines, which were introduced in the mid-1980s.

5.2 RISC/CISC EVOLUTION CYCLE

In the mid-1970's advances in semiconductor technology began to reduce the difference in speed between main memory and processor chips. As memory speed increased, and high-level languages displaced as-

sembly language, the major reasons for CISC (complex instruction set computer) began to disappear, and computer designers began to look at ways computer performance could be optimized beyond just making faster hardware. One of their key realizations was that a sequence of simple instructions produces the same results as a sequence of complex instructions, but can be implemented with a simpler (and faster) hardware design.

A RISC (reduced instruction set computer) is a microprocessor that is designed to perform a smaller number of types of computer instruction so that it can operate at a higher speed (million of instructions per second). Since each instruction type that a computer must perform requires additional transistors and circuitry, a larger list or set of computer instructions tends to make the microprocessor more complicated and slower in operation.

John Cocke of IBM Research in Yorktown, New York, originated the RISC concept in 1974 by proving that about 20% of theinstructions in a computer did 80% of the work. The first computer to benefit from this discovery was IBM's PC/XT in 1980. Later, IBM's RISC System/6000, made use of the idea. The term itself (RISC) is credited to David Patterson, a teacher at the University of California in Berkeley. The concept was used in Sun Microsystems' SPARC microprocessors and led to the founding of what is now MIPS Technologies, part of Silicon Graphics. DEC's Alpha microchip also uses RISC technology.

The RISC concept has led to a more thoughtful design of the microprocessor. Among design considerations are how well an instruction can be mapped to the clock speed of the microprocessor (ideally, an instruction can be performed in one clock cycle); how "simple" an architecture is required; and how much work can be done by the microchip itself without resorting to software help. Besides performance improvement, some advantages of RISC and related design improvements are: A new microprocessor can be developed and tested more quickly if one of its aims is to be less complicated.

Operating system and application programmers who use the microprocessor's instructions will find it easier to develop code with a smaller instruction set.

The simplicity of RISC allows more freedom to choose how to use the space on a microprocessor.

5.3 RISC DESIGN PRINCIPLES

A computer with the minimum number of instructions has the disadvantage that a large number of instructions will have to be executed in realizing even a simple function. This will result in a speed disadvantage. On the other hand, a computer with large number of instructions has the disadvantage of complex decoding and hence a speed disadvantage. It is then natural to believe that a computer with a carefully selected reduced set of instructions should strike a balance between the above two design alternatives. The question then becomes what constitutes a carefully selected reduced set of instructions? In order to arrive at an answer to this question, it is necessary to conduct in-depth studies on a number of aspects of computation. These aspects should include (a) operations that are most frequently performed during execution of typical (benchmark) programs, (b) operations that are most time consuming, and (c) the type of operands that are most frequently used. A number of early studies were conducted in order to find out the typical break down of operations that are performed in executing benchmark programs. The estimated distribution of operations is shown in Table 1. careful look at the estimated percentage of operations performed reveals that assignment statements, conditional branches, and procedure calls constitute about 90% of the total operations performed, while other operations, however complex they may be, make up the remaining 10%.

TABLE 1 Estimated Distribution of Operations

Operations	Estimated percentage
Assignment statements	35
Loops	5
Procedure calls	15
Conditional branches	40
Unconditional branches	3
Others	2

The studies on time–performance characteristics of operations revealed that among all operations, procedure calls/return are the most time-consuming. With regards to the type of operands used during typical computation, it was noticed that the majority of references (no less than 60%) are made to simple scalar variables and that no less than 80% of scalars are local variables (to procedures).

The above observations about typical program behavior have led to the following conclusions:

1. Simple movement of data (represented by assignment statements), rather than complex operations, are substantial and should be optimized.

2. Conditional branches are predominant and therefore careful attention should be paid to the sequencing of instructions. This is particularly true when it is known that pipelining is indispensable to use.

3. Procedure calls/return are the most time-consuming operations and therefore a mechanism should be devised to make the communication of parameters among the calling and the called procedures cause the least number of instructions to execute.

5.4 RISC CHARACTERISTICS

1. Simple instruction set.

In a RISC machine, the instruction set contains simple, basic instructions, from which more complex instructions can be composed.

2. Same length instructions.

Each instruction is the same length, so that it may be fetched in a single operation.

3. Machine-cycle instructions.

One instruction per clock cycle. Most instructions complete in one machine cycle, which allows the processor to handle several instructions at the same time. This pipelining is a key technique used to speed up RISC machines.

4. Fixed-length instructions.
5. Limited number of instructions (128 or less).
6. Limited set of simple addressing modes (minimum of two: indexed and PC-relative).
7. All operations are performed on registers; no memory operations. Only two memory operations: Load and Store.
8. Pipelined instruction execution.
9. Large number of general-purpose registers or the use of advanced compiler technology to optimize register usage.
10. Hardwired control unit design rather than microprogramming.
11. RISC use Overlapped registered window.

Self learning exercise

1. What do you mean by hardwired control?
2. what is the difference between hardwired control and micro-program control ?
3. What is microprogram?

5.5 OVERLAPPED REGISTER WINDOWS

The main idea behind the use of register windows is to minimize memory accesses .In order to achieve that; a large number of CPU registers are needed. For example, the number of CPU general-purpose registers available in the original SPARC machine (one of the earliest RISCs) was 120. However, it is desirable to have only a subset of these registers visible at any given time and to have them addressed as if they were the only set of registers available. Therefore, CPU registers are divided into multiple small sets, each assigned to a different procedure. A procedure call will automatically switch the CPU to use a different. Fixed-size window of registers. In order to minimize the actual movement of parameters between the calling and the called procedures, each set of registers is divided into three subsets: parameter registers, local registers, and temporary registers. When a procedure call is made, a new overlapping window will be created such that the temporary registers of the caller are physically the same as the parameter registers of the called procedure. This overlap allows parameters to be passed among procedure without actual

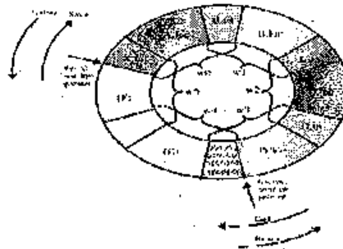
movement of data (Fig. 1).

Operation of Circular Buffer:

When a call is made, a current window pointer is moved to show the currently active register window .If all windows are in use, an interrupt is generated and the oldest window (the one furthest back in the call nesting) is saved to memory. A saved window pointer indicates where the next saved windows should restore.

TABLE 2 Different Register Windows Characteristics

Architecture	Number of windows	Number of registers per window
Berkeley RISC-I	8	16
Pyramids	16	32
SPARC	32	32



In addition, a set of a fixed number of CPU registers is identified as global registers and is available to all procedures. For example, references to registers 0 through 7 in the SPARC architecture refer to unique global registers, and references to registers 8 through 31 indicate registers in the current window. The current window is pointed to using what is normally called the current window pointer (CWP). Upon having all windows filled, the register window wraps around, thus acting like a "circular buffer." Table .2 shows the number of windows and the window size for a number of architectures. It should be noted that a study was conducted in 1985 to find out the impact of using register window on the performance of the Berkeley RISC. In this study, two versions of the machine were studied. The rest is design with register windows and the second was a hypothetical Berkeley RISC implemented without windows. The results of the study indicated a decrease by a factor of 2 to 4 (depending On specified benchmark) in the memory traffic due to the use of register windows.

Example: Suppose a system has total 74 registers R0 to R7 are global registers

Rest 64 registers are divided in 4 windows to accommodate procedure A, B, C, D each register window consist 10 local registers two set of 6 register are common to adjacent window. The common overlap register permit parameters to be passed without the actual movement . And only one register window is activated at a time. With a pointer indicating current window. Each procedure call is activated by incrementing the pointer. the high register of calling procedure called the low register of calling procedure .There fore the parameter automatically transferred from calling procedure to called procedure .If procedure A call procedure B then register R26 through R31 are common to both procedures procedure use local R32 to R41 register for local variable storage .

So for each procedure total 32 registers are available.

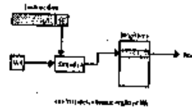
Which is 10 global +10 local +6 low +6 high =32

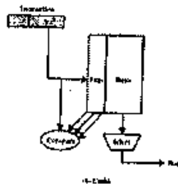
In general

suppose global register= G
number of local register are =L
number of common register are =C
number of window are = W
then number of available register are calculated as follows
Window size = L+2C+G
The total number of register need in the processor (register file)
Register file = (L+C)W +G
If G=10 ,L=10, C=6 and W=4 then
Register file =(10+6)X4 +10 = 74
Window size = L +2C +G = 10+12+10=32
Registers v_s Cache:

REGISTER	CACHE
Large Register File	
All local scalars	Recently used local scalars
Individual variables	Blocks of memory
Compiler assigned global variables	Recently used global variables
Save/restore based on procedure nesting	Save/restore based on caching algorithm
Register addressing	Memory addressing

Referencing a Scalar - Window Based Register File





Referencing a Scalar – Cache:

In cache memory address is divided into two parts tag and index. so in case of 15 bit memory address n . Then k bit index and n-k bit tag field. Each word in cache consist of data word and its associated tag. The CPU generate a request , the index field is used for the address to access the cache .the tag field of the CPU address is compared with the tag in the word read from the cache , if two tag match a hit and the desired word is in the cache .

5.6 INSTRUCTION EXECUTION

Instruction and the assembly language definition are given in the given table

Table 3 *Assembly Language Definition for RISC I*

Instr.	Operands	Comments	
ADD	Rs,S2,Rd	Rd ← Rs + S2	integer add
ADDC	Rs,S2,Rd	Rd ← Rs + S2 + carry	add with carry
SUB	Rs,S2,Rd	Rd ← Rs - S2	integer subtract
SUBC	Rs,S2,Rd	Rd ← Rs - S2 - carry	subtract with carry
SUBR	Rs,S2,Rd	Rd ← S2 - Rs	integer subtract
SUBCR	Rs,S2,Rd	Rd ← S2 - Rs - carry	subtract with carry
AND	Rs,S2,Rd	Rd ← Rs & S2	logical AND
OR	Rs,S2,Rd	Rd ← Rs | S2	logical OR
XOR	Rs,S2,Rd	Rd ← Rs xor S2	logical EXCLUSIVE OR
SLL	Rs,S2,Rd	Rd ← Rs shifted by S2	shift left
SRL	Rs,S2,Rd	Rd ← Rs shifted by S2	shift right logical
SRA	Rs,S2,Rd	Rd ← Rs shifted by S2	shift right arithmetic
LDL	(Rx)S2,Rd	Rd ← M[Rx+S2]	load long
LDSU	(Rx)S2,Rd	Rd ← M[Rx+S2]	load short unsigned
LDSS	(Rx)S2,Rd	Rd ← M[Rx+S2]	load short signed
LDBU	(Rx)S2,Rd	Rd ← M[Rx+S2]	load byte unsigned
LDBS	(Rx)S2,Rd	Rd ← M[Rx+S2]	load byte signed
STL	(Rx)S2,Rm	M[Rx+S2] ← Rm	store long
STS	(Rx)S2,Rm	M[Rx+S2] ← Rm	store short
STB	(Rx)S2,Rm	M[Rx+S2] ← Rm	store byte
JMP	CON,S2(Rx)	pc ← Rx+S2	conditional jump
JMPR	CON,Y	pc ← pc + Y	conditional relative
CALL	S2(Rx),Rd	CWP--; Rd ← pc, next pc ← Rx+S2	call reg.-indexed and change window
CALLR	Y,Rd	CWP--; Rd ← pc, next pc ← pc + Y	call relative and change window
RET	(Rx)S2	pc ← Rx+S2, next CWP++	return, change window
RETINT	(Rx)S2	pc ← Rx+S2; next CWP++	also enable interrupts
CALLINT	Rd	CWP--; Rd ← last pc	also disable interrupts
LDHI	Y,Rd	Rd<31:13> ← Y; Rd<12:0> ← 0	load immediate high

5.7 RISC PIPELINING :

Pipelining is a technology of decomposing a sequential process into sub-process with each sub-process is executed in a special dedicated segment that operates concurrent with all other segments.

In RISC most instructions are register to register so no need to calculate the effective address of operand or fetching the operand from memory the instruction cycle is divided into three phases

I: Instruction fetch

E: Execute

A: ALU operation with register input and output For load and store

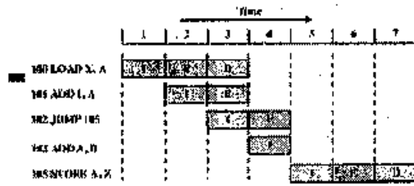
I: Instruction fetch

E: Execute Calculate memory address

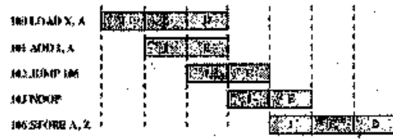
D: Memory Register to memory or memory to register operation

Normal and delayed branch

Address	Normal	Delayed	Optimized
100	LOAD X,A	LOAD X,A	LOAD X,A
101	ADD 1,A	ADD 1,A	JUMP 105
102	JUMP 105	JUMP 105	ADD 1,A
103	ADD A,B	NOOP	ADD A,B
104	SUB C,B	ADD A,B	SUB C,B
105	STORE A,Z	SUB C,B	STORE A,Z
106		STORE A,Z	

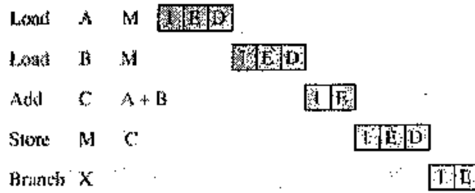


(a) Traditional Pipeline

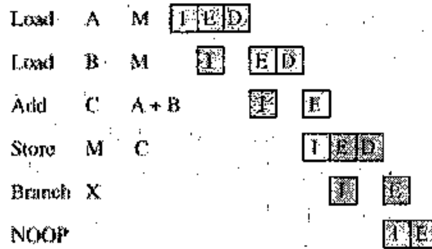


(b) RISC Pipeline with Inserted NOOP

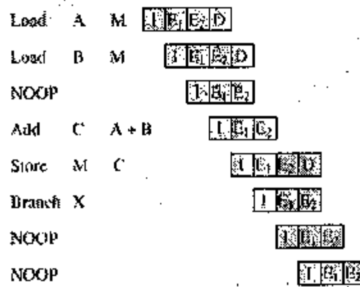
(c) Reversed Instructions



(a) Sequential execution



(b) Two-way pipelined timing



(d) Four-way pipelined timing

There is a problem called data dependency arise if it may caused by degradation of performance in instruction pipeline due to the collision of data and instructions.
A data dependency occurs if an instruction is needed data and data is not yet available. Similarly if an instruction is needed address and address is not yet available
Delayed branch and use of NOP (no operation to resolve pipeline conflict).

Figure 3 The 21264 instruction pipeline

5.8 RISC V_S CISC

TABLE 4

	Complex Instruction Set (CISC) Computer			Reduced Instruction Set (RISC) Computer		Superscalar		
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	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 (KB)	64	64	8	32	128	[illegible]	32	64

The choice of RISC versus CISC depends totally on the factors that must be considered by a computer designer. These factors include size, complexity, and speed. RISC architecture has to execute more instructions to perform the same function performed by CISC architecture. To compensate for this drawback, RISC architectures must use the chip area saved by not using complex instruction decoders in providing a large number of CPU registers, additional execution units, and instruction caches. The use of these resources leads to a reduction in the traffic between the processor and the memory. On the other hand, CISC architecture with richer and more complex instructions will require a smaller number of instructions than its RISC counterpart. However, CISC architecture requires a complex decoding scheme and hence is subject to logic delays. It is therefore reasonable to consider that the RISC and CISC paradigms differ primarily in the strategy used to trade off different design factors. There is very little reason to believe that an idea that improves performance for RISC architecture will fail to do the same thing in CISC architecture and vice versa. For example, one key issue in RISC development is the use of optimizing the compiler to reduce the complexity of the hardware and to optimize the use of CPU registers. These same ideas should be applicable to CISC compilers.

TABLE 5 RISC Versus CISC Performance

Application	MIPS CPI (RISC)	VAX CPI (CISC)	CPI ratio	Instruction ratio
Spice 2G6	1.80	8.02	4.44	2.48
Matrix300	3.06	13.81	4.51	2.37
Nasa 7	3.01	14.95	4.97	2.10
Espresso	1.06	5.40	5.09	1.70

Increasing the number of CPU registers could very much improve the performance of a CISC machine. This could be the reason behind not. Finding a pure commercially available RISC (or CISC) machine. It is not unusual to see a RISC machine with complex floating-point instructions (see the details of the SPARC architecture in the next section). It is equally expected to see CISC machines making use of the register

windows RISC idea. In fact there have been studies indicating that a CISC machine such as the Motorola 680xx with a register window will achieve a 2 to 4 times decrease in the memory traffic. This is the same factor that can be achieved by RISC architecture, such as the Berkeley RISC, due to the use of a register window. It should, however, be noted that most processor developers (except for Intel and its associates) have opted for RISC processors. Computer system manufacturers such as Sun Microsystems are using RISC processors in their products. However, for compatibility with the PC-based market, such companies are still producing CISC-based products. Tables.4 and 5 show a limited comparison between an example RISC and CISC machine in terms of performance and characteristics, respectively. An elaborate comparison among a number of commercially available RISC and CISC machines.

5.9 RISC AND CISC ARCHITECTURES:

5.9.1 SPARC

Sun SPARC(scalable processor architecture) the SPARC make use of register windows.
SPARC is a RISC architecture with some unusual quirks such as register windows. The architecture is open but its main use if by its originators Sun Microsystems in their Sparcstation line. Thus it is the architecture of the 3rd year laboratory machine Pompeii.

Memory and data types

SPARC is a 32-bit architecture. The general purpose registers are 32 bits, as are memory addresses. Thus 2^32 bytes can be addressed. In addition, instructions are all 32 bits long. SPARC instructions support a variety of integer data types from single bytes to double words (eight bytes) and a variety of different precision floating-point types. Each of these values can only be accessed when aligned to a memory address, which is a multiple of its size. Integer values are stored in twos-complement representation and floating-point values are encoded using the IEEE standard.

Registers

Most of the instruction are register to register

R_d and R_{s1} are register references S_2 are register or a 13 bit immediate operand.

At any point during execution the SPARC has provides access to 32 general purpose integer registers (8 global registers and 24 register window registers) and 32 general purpose floating-point registers. The integer registers are numbered 0-31 and the following aliases are used to distinguish the different purposes of different registers.

regs 0-7	%g0-%g7	! global registers
regs 8-15	%o0-%o7	! output registers
regs 16-23	%l0-%l7	! local registers
regs 24-31	%i0-%i7	! input registers

The floating-point registers are also numbered 0-31 and are referred to as %f0-%f31.
The global registers refer to the same set of physical registers in all procedures. The other registers are stored in a register stack that provides the ability to manipulate register windows. The local registers are only accessible to the current procedure.
The architecture constrains %g0 to always be 0. When used as a source register it will always return 0; when used as a result destination the result will be thrown away. Register 15 (%o7) is used by the call instruction to hold the return address during procedure calls.
The other output registers are used to pass parameters and return values from procedures. When a procedure is called, parameters are passed in the out registers and the register window is shifted 16

registers further into the register stack. This makes the input registers of the called procedure the same as the output registers of the calling procedure. Thus the parameters are accessed in the called procedure via its in registers. Also, word values to be returned by the procedure can be left in the in registers. When the procedure returns the caller can access the returned values in its out registers.

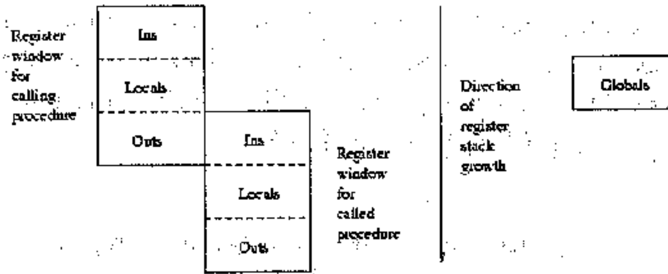


Figure : 4 Overlap register in SPARC .

The detail of SPARC register window operation is not important for the Obr compiler because Obr has no procedures and we only make use of procedure calls to utilize C run-time routines. The parameters we pass to these routines will be passed in the out registers %o0 and %o1, and the return value (if any) will be available in %o0 when the routine returns. Local registers will be used to store the address of the variable memory block and to hold temporary values computed in expressions. We will not use the in registers.

Instruction set

For our purposes the SPARC architecture has instructions that can be categorised as follows:

- load/store (memory access)
- integer/floating-point arithmetic and conversion
- control transfer

Load/store instructions are the only instructions that access memory. The ld and st instructions load and store word/single quantities from/to integer and floating-point registers.

ld mem, reg ! Load word from address mem into reg.
st reg, mem ! Store contents of reg to address mem.

The memory address can be specified either by two registers or by one register and a 13-bit immediate value (which can be omitted if zero). Indirect addressing is indicated by square brackets. The following examples load from various addresses into %l1.

ld [%l0], %l1 ! Load from the address stored in %l0.
ld [%l0+4], %l1 ! Load the word from a four byte offset from the
! address stored in %l0.
ld [%l0+%l4], %l1 ! Load from the address that is the sum of the
! contents of %l0 and %l4.

Most *arithmetic* instructions operate on two source registers and write their result to a register (which may be one of the source registers). One of the registers can be replaced with an immediate integer operand. The Obr compiler makes use of the following integer operations: add, sub, smul, and sdiv. The 's' in the last two instruction names indicates 'signed' multiplication and division. The following single-precision floating-point operations are used: fadds, fsubs, fmuls, and fdivs

add %l1, %l2, %l3 ! Add the word values in %l1 and %l2 and put result in %l3.
add %l4, 4, %l4 ! Increment %l4 by four.

The neg and not integer arithmetic instructions implement integer negation and complement. They operate on a single source register and write their result to a register (which may be the same as the source register). Similarly, fnegs negates single-precision floating-point values.

neg %l1, %l2 ! Negate the word value in %l1 and put result input %l2.

The mov and set instructions can be used to transfer values between registers, to load literal values into registers, or to load symbolic addresses into registers.

```
mov %l1, %o0     ! Copy the value in %l1 to %o0.
mov 1, %l2       ! Put the value 1 in %l2.
set ifmt, %l4    ! Put the address of the symbolic constant ifmt in %l4.
```

Loading literal floating-point values into registers is harder because floating-point values cannot be immediate instruction operands. Thus it is necessary to encode the value in memory and load it from there. The following fragment illustrates a typical sequence. The operand of the .word pseudo-op encodes the floating-point value.

```
.seg    "data"
L29:
        .word  0x40133333
        .seg   "text"
        set    L29, %o0
        ld     [%o0], %f2
```

Conversion of word integer values to/from single-precision floating-point values is achieved using the fitos and fstoi instructions. A gotcha is that the integer value to be converted or produced as a result of conversion must reside in a floating-point register. There is no way to transfer directly from the floating-point registers to/from the integer registers so memory must be used as an intermediary if the value is needed subsequently or comes from elsewhere. The following code shows a sequence converting the integer value 1 to a floating-point value via a temporary memory address.

```
mov     1, %l1
st      %l1, [%l0+12]
ld      [%l0+12], %f1
fitos   %f1, %f1
```

Control transfer instructions alter the program counter. They are used to implement source-level control statements, procedure calls. The SPARC has delayed branches so in most cases we will insert nop (no operation) instructions in the delay slots of branches.

The ba (branch always) instruction is an unconditional transfer of control.

```
ba L1         ! Transfer control to label L1.
nop           ! Do nothing in the delay slot.
```

Conditional transfers are implemented with the aid of a set of *condition codes* that are set by the cmp, fcmps and tst instructions.

```
cmp %l2, %l5          ! Compare the values of %l2 and %l5 and set condition codes.
tst %l3               ! Compare %l3 to zero and set condition codes.
```

Once the condition codes have been set, branch instructions can be used to transfer control if particular condition code bits are set. The Obr compiler uses the following branch instructions: be (branch equal), bne (branch not equal), bg (branch greater than), and bl (branch less than).

```
cmp %l1, %l4          ! Compare the values of %l1 and %l4 and set condition codes.
bg L2                 ! Branch to label L2 if %l1 > %l4
nop                   ! Do nothing in the delay slot.
```

Although Obr has no procedures or functions, our compiler must generate procedure calls and returns in order to be able to interface properly with the C run-time library. The following instructions are used: call (call subroutine), ret (return from subroutine), save, (save register window), and restore (restore register window).

Modulus operations

The SPARC does not have an instruction that can perform a modulus (remainder) operation. To generate code for the MOD operation in the Obr compiler we make use of a C run-time routine called .rem'. A modulus operation %l1 MOD %l2 leaving the result in %l3 is implemented as follows.

```
mov %l1, %o0        ! Pass value in %l1 as first parameter.
mov %l2, %o1        ! Pass value in %l2 as first parameter.
call .rem           ! Call the .rem run-time routine.
nop                 ! Do nothing in the delay slot.
mov %o0, %l3        ! Move the return value from %o0 to %l3.
```

Self learning exercise

1. Give the example of RISC and CISC.
2. What does the SPARC mean?

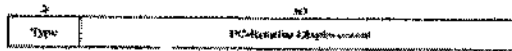
5.9.2 MIPS (Microprocessor without interlocked pipeline stages)

The first commercial RISC chip developed by MIPS technology.

Series of MIPS are MIPS2000, MIPS3000, and MIPS4000and MIPS6000.

The MIPS4000 uses 32 registers of 64 bits registers

MIPS is a 32-bit pipelined LOAD/STORE machine. It uses a five-stage pipeline consisting of Instruction Fetch (IF), Instruction Decode (ID), Operand Decode



(OD), Operand Store/Execution (OS/EX), and Operand Fetch (OF). The first three stages perform respectively instruction fetch, instruction decode, and operand fetch. The OS/EX stage sends operand to memory in the case of a store instruction or use the ALU in case of instruction execution. The OF stage receives the operand in case of a load instruction. MIPS uses a mechanism called pipeline interlock in order to prevent an instruction from continuing until the needed operand is available. Unlike the Berkeley RISC, MIPS have a single set of sixteen 32-bit general-purpose registers. The MIPS compiler optimizes the use of registers in whatever way is best for the program currently being purpose registers, MIPS provided four additional registers in order to hold the four previous PC values (to support backtracking and restart in case of a fault). A register is used to hold the future PC value (to support branch instructions). Four addressing modes are used in MIPS. These are immediate, indexed, based with offset, and base shifted. Four instruction groups were identified in MIPS. These are ALU, Load/Store, Control, and Special instructions. A total of 13 ALU instructions were provided. These include all register-to-register two- or three operand formats. (Fig. 6). A total of 10 LOAD/STORE instructions were provided. They use 16 or 32 bits. In the latter case, adding a 16-bit signed constant to a register using the second format uses indexed addressing. A total of six control flow instructions were provided. These include

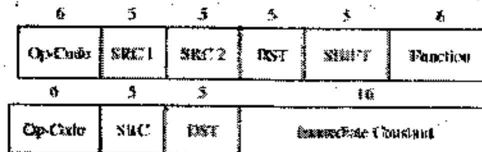
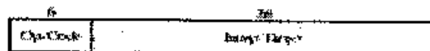


Figure 6 Three-operand instructions used in MIPS



jumps, relative jumps, and compare instructions. They support procedure and interrupt linkage. Some examples of MIPS instructions are:

1. ALU: Add src1, src2, dst; dst src1 þ src2
2. Load/Store: Ld [src1 þ src2], dst; dst M[src1 þ src2]
3. Control: Jmp dst; PC dst
4. Special Function: SavePC A; M[A] PC

5.9.3 MIPS R4000

- General Purpose registers: R0 - R31, 64 bits. R0 is constant 0, R31 is a link register for link and jump instructions.
- Multiply and divide High/Low registers — 64 bits each.
- Program Counter (64 bits)
- Floating Point registers: FPR0 - FPR31 (32 or 64 bit)
- Control/Status register (32 bit)
- Implementation/Revision register (32 bit)
- Control Coprocessor registers: 32 named registers, each 32 bits. Cache and MMU control, exception processing, debugging.

The MIPS family has four addressing modes:

- Base + immediate offset (loads and stores)
- Register direct (arithmetic)
- Immediate (jumps)
- PC relative (branches)

Memory accesses in the MIPS architecture are to any multiple between 1 and 8 bytes.

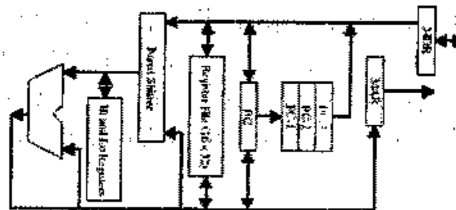


Figure 2 MIPS organization

5.9.4 POWERPC

In 1975, IBM started the 801 minicomputer project that launched the RISC movement. In 1986, IBM developed a RISC workstation, the RT PC, which was not a commercial success. In 1990, introduced the RISC/6000 and marketed that as a high performance workstation. IBM began to refer to this as the POWER architecture.

IBM then entered into an alliance with Motorola the developer of the 68000 series for Apple computers. The result of this alliance was the series of microprocessors that implement the PowerPC architecture. The processors in the series were: 601, 603, 604, 620, 740/750 (G3), G4, and G5. A complete description of the PowerPC ISA can be obtained from the IBM site.

Example of CISC

5.9.5 PENTIUM

The Pentium series is an excellent example of Complex Instruction Set Computer (CISC) design. The PowerPC is a direct descendant of IBM 801, one of the best designed RISC systems on the market.

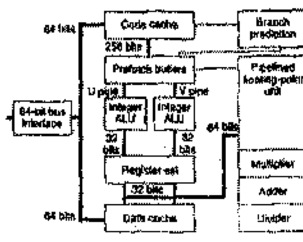


Figure . Pentium processor block diagram.

The most important enhancements over the 486 are the separate instruction and data caches, the dual integer pipelines (the U-pipeline and the V-pipeline, as Intel calls them), branch prediction using the branch target buffer (BTB), the pipelined floating-point unit, and the 64-bit external data bus. Even-parity checking is implemented for the data bus and the internal RAM arrays (caches and TLBs).

As for new functions, there are only a few; nearly all the enhancements in Pentium are included to improve performance, and there are only a handful of new instructions. Pentium is the first high-performance microprocessor to include a system management mode like those found on power-miserly processors for notebooks and other battery-based applications; Intel is holding to its promise to include SMM on all new CPUs. Pentium uses about 3 million transistors on a huge 294 mm 2 (456k mils 2). The caches plus TLBs use only about 30% of the die. At about 17 mm on a side, Pentium is one of the largest microprocessors ever fabricated and probably pushes Intel's production equipment to its limits. The integer data path is in the middle, while the floating-point data path is on the side opposite the data cache. In contrast to other superscalar designs, such as SuperSPARC, Pentium's integer data path is actually bigger than its FP data path. This is an indication of the extra logic associated with complex instruction support. Intel estimates about 30% of the transistors were devoted to compatibility with the x86 architecture. Much of this overhead is probably in the microcode ROM, instruction decode and control unit, and the adders in the two address generators, but there are other effects of the complex instruction set. For example, the higher frequency of memory references in x86 programs compared to RISC code led to the implementation of the dual-ac.

Register set

The purpose of the Register is to hold temporary results, and control the execution of the program. General-purpose registers in Pentium are EAX, ECX, EDX, EBX, ESP, EBP,ESI, or EDI.

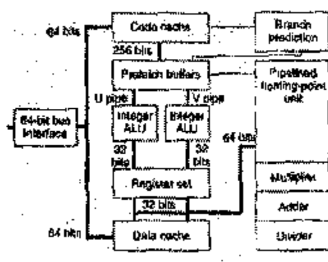


Figure . Pentium processor block diagram.

The most important enhancements over the 486 are the separate instruction and data caches, the dual integer pipelines (the U-pipeline and the V-pipeline, as Intel calls them), branch prediction using the branch target buffer (BTB), the pipelined floating-point unit, and the 64-bit external data bus. Even-parity checking is implemented for the data bus and the internal RAM arrays (caches and TLBs).

As for new functions, there are only a few; nearly all the enhancements in Pentium are included to improve performance, and there are only a handful of new instructions. Pentium is the first high-performance microprocessor to include a system management mode like those found on power-miserly processors for notebooks and other battery-based applications; Intel is holding to its promise to include SMM on all new CPUs. Pentium uses about 3 million transistors on a huge 294 mm 2 (456k mils 2). The caches plus TLBs

use only about 30% of the die. At about 17 mm on a side, Pentium is one of the largest microprocessors ever fabricated and probably pushes Intel's production equipment to its limits. The integer data path is in the middle, while the floating-point data path is on the side opposite the data cache. In contrast to other superscalar designs, such as SuperSPARC, Pentium's integer data path is actually bigger than its FP data path. This is an indication of the extra logic associated with complex instruction support. Intel estimates about 30% of the transistors were devoted to compatibility with the x86 architecture. Much of this overhead is probably in the microcode ROM, instruction decode and control unit, and the adders in the two address generators, but there are other effects of the complex instruction set. For example, the higher frequency of memory references in x86 programs compared to RISC code led to the implementation of the dual-ac.

Register set

The purpose of the Register is to hold temporary results, and control the execution of the program. General-purpose registers in Pentium are EAX, ECX, EDX, EBX, ESP, EBP,ESI, or EDI.

The 32-bit registers are named with prefix E, EAX, etc, and the least 16 bits 0-15 of these registers can be accessed with names such as AX, SI Similarly the lower eight bits (0-7) can be accessed with names such as AL & BL. The higher eight bits (8-15) with names such as AH & BH. The instruction pointer EAP known as program counter(PC) in 8-bit

microprocessor, is a 32-bit register to handle 32-bit memory addresses, and the lower 16 bit segment IP is used for 16-bi memory address.

The flag register is a 32-bit register , however 14-bits are being used at present for 13 different tasks; these flags are upward compatible with those of the 8086 and 80286. The comparison of the available flags in 16-bit and 32-bit microprocessor is may provide some clues related to capabilities of these processors. The 8086 has 9 flags, the 80286 has 11 flags, and the 80286 has 13 flags. All of these flag registers include 6 flags related to data conditions (sign, zero, carry, auxiliary, carry , overflow, and parity) and three flags related to machine operations.(interrupts, Single-step and Strings). The 80286 has two additional : I/O Privilege and Nested Task. The I/O Privilege uses two bits in protected mode to determine which I/O instructions can be used, and the nested task is used to show a link between two tasks.

The processor also includes control registers and system address registers , debug and test registers for system and debugging operations.

Addressing mode & Types of instructions

Instruction set is divided into 9 categories of operations and has 11 addressing modes. In addition to commonly available instructions in a 8 bit microprocessor and this set includes operations such as bit manipulation and string operations, high level language support and operating system support. An instruction may have 0-3 operands and the operand can be 8, 16, or 32- bits long. The 80386 handles various types of data such as Single bit , string of bits , signed and unsigned 8-, 16-, 32- and 64- bit data, ASCII character and BCD numbers.

Pentium

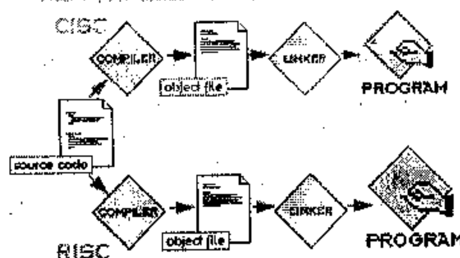
Intel has ranked the number one maker of microprocessors for decades.

1985 - IA-32 80386 Processor
1989 - IA-32 80486 Processor
1993 - Pentium
1995 - Pentium Pro
1997 - Pentium II
1999 - Pentium III
2000 - Pentium 4
2001 - IA-64 Itanium
2002 - current - Itanium 2

Transistor count of some processors

Processor	Technology	Number of transistors
SPARC	[illegible]	0.8 micron CMOS
MIPS R3000	110000	2 micron CMOS
MIPS R6000	360000	ECL
M88000	175000	CMOS
(M88200)	[illegible]	CMOS
IBM RS/6000	2040000	[illegible] micron
Intel 80486	1200000	CMOS
Motorola 68040	1200000	CMOS

Since CISC machines perform complex actions with a single instruction, where RISC machines may require multiple instructions for the same action, code expansion can be a problem



Code expansion refers to the increase in size that you get when you take a program that had been compiled for a CISC machine and re-compile it for a RISC machine. The exact expansion depends primarily on the quality of the compiler and the nature of the machine's instruction set. Fortunately for us, the code expansion between a 68K processor used in the non-PowerPC Macintoshes and the PowerPC seems to be only 30-50% on the average, although size-optimized PowerPC code can be the same size (or smaller) than corresponding 68K code.

5.10 SUMMARY:

Both RISCs and CISCs try to solve the same problem: to cover the semantic gap. They do it in different ways. CISCs are going the traditional way of implementing more and more complex instructions. RISCs try to simplify the instruction set.

- Innovations in RISC architectures are based on a close analysis of a large set of widely used programs.
- The main features of RISC architectures are: reduced number of simple instructions, few addressing modes, load-store architecture, instructions are of fixed length and format, a large number of registers is available.
- One of the main concerns of RISC designers was to maximize the efficiency of pipelining.
- Present architectures often include both RISC and CISC features.

5.11 GLOSSARY

Addressing mode :Addressing modes are an aspect of the instruction set architecture in most central processing unit (CPU) designs.

CISC ; complex instruction set computer.

Instruction Set : The set of machine instructions that a particular CPU can execute; the corresponding set of assembly language mnemonics

MIPS : Microprocessor without interlocked pipeline stages.

Register : Register are the group of flip flop .

RISC : Reduce instruction set computer.

SPARC : Scalable processor architecture.

5.12 FURTHER READINGS

J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann: San Mateo, San Francisco, CA, 1996.

D. Patterson and R. Ditzel, The case for the reduced instruction set computer, Comput. Architecture

A. Tanenbaum, Structured Computer Organization, 3rd ed., Prentice-Hall: Englewood Cliffs

5.13 ANSWERS TO SELF LEARNING EXERCISES

1. Hard wired control contain fixed logical circuit.
2. microprogram control unit is constructed by microinstruction use control memory which is costly and hardwired control unit is a fixed logical circuit.
3. microprogram is a set of instructions.
4. SPARC,PowerPC,MIPS are example of RISC and Pentium processor are example of CISC.
5. scalable processor architecture.

5.14 UNIT END QUESTIONS

1. Contrast the two approaches (the software and the hardware) used in RISC machines to minimize memory operations.
2. What are the main principles used to construct a RISC machine?
3. Explain, with examples, the concept of register window and window overlapping. Suggest a different approach to achieve the same results as those achieved using register window and window overlapping.
4. Explain overlap register window.
5. What do RISC and CISC stand for and what are the differences in practice?
6. Some instruction sets use a register file and others use an operand stack for intermediate storage. How does the code density compare between these two approaches?
7. Early computers used just an accumulator rather than a register file or stack. How does the code density compare between accumulator and stack machines?
8. Compact RISC instruction sets typically use a fixed 16-bit instruction size. If three operands are to be specified, each of n bits, then $(16 - 3 \times n)$ bits are left for the opcode. If $n = 3$ then we cannot access enough registers and if $n = 4$ we do not have enough opcode bits. In practice we usually need 5 bits for the opcode leaving 11 bits to specify three registers. Using 11 bits to store three operands, how many registers can be specified and how might these three operands be decoded?

UNIT 6
Control Design

Structure Of The Unit

6.0 Objective
6.1. Introduction
6.2. Basic Concept
6.3. Design Control Unit
6.3.1. Hardwired Control Unit
6.3.1.1. Classical Method
6.3.1.2. One Hot Method
6.3.2. Microprogrammed Control
6.4. Parallelism In Microinstructions
6.5. Horizontal Versus Vertical
6.6. Advantages & Applications Of Microprogramming
6.7. Multiplier Control Unit
6.8. Summary
6.9. Glossary
6.10. Further Readings
6.11. Unit End Questions

6.0 Objective:

In this unit we study register level design of control part of instruction set processor. we design the control unit which is two type hardwired control and micro programmed control.

6.1 INTRODUCTION:

A processor is composed of data path (data processing) and control unit. The data path is a network of functional and storage units capable of performing certain microoperation on data word. Data path of a processor is the execution unit such as ALU, shifter, registers and their interconnects. Control unit is considered to be the most complex part of a processor. The purpose of control unit is to issue control signals to the data path or its function is to control various units in the data path. These control signals enter the datapath at "control points" where they select the function Control unit realizes the behavior of a processor as specified by its micro operations.

6.2 BASIC CONCEPTS:

A CPU's datapath contains circuits to perform arithmetic and logical operations on words such as fixed-point or floating-point numbers. "It contains a register file (RF) for temporary storage of operands, two functional units F_1 and F_2 responsible for data processing, and multiplexers to allow the data to be steered through DP. Typical functional units are an ALU performing addition, subtraction, and logical operations; a shifter; or a multiplier. The control unit CU receives external instructions or commands, which it converts into a sequence of control signals that the CU applies to DP to implement a sequence of register-transfer operations.

Figure:1 Processor with control unit and datapath unit

The control signals that implement an addition instruction of the form ADD A,B, which we write as:A := A + B; in our HDL notation. Assume that this operation can be executed in a single clock cycle, whose timing details are not of concern at this level of abstraction. The input variables A and B are obtained from registers of the same name in RF, and the result is stored back into register A. Observe that the registers of RF permit their contents to be read from and written into in the same clock cycle, a basic property of the (edge-triggered) flip-flops from which such registers are constructed. RF is configured with one input and two output ports to support operations like with two or three addresses. Besides selecting the data registers to be used, the control unit CU must also select the operation to be performed on the data, in this case, functional unit F1 perform ADD operation. Finally the necessary logical connections for the data to flow through DP must be established by applying appropriate control signals to the multiplexers.

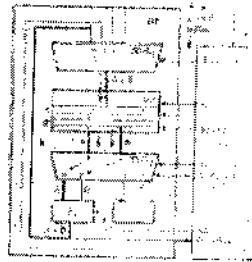


Figure2 : Implement the ADD operation (A:=A+B)

Thus we see that CU must activate the following three types of control signals during the clock cycle in which the ADD A,B instruction is executed.

- Function select: Add.
- Storage control: Read A, Read B, Write A.
- Data routing: Select *p-t*, Select *u-w*, Select *v-x*.

There is usually some feedback of control information from DP to CU to indicate exceptional conditions encountered during instruction execution. In the above given example the functional unit F_i performing the addition sends an overflow signal to CU whenever the sum A + B exceeds the normal word size.

Cycle	Function select	Storage control	Data routing
1	Add	Read A, Read B, Write A	Select *p-t*, Select *u-w*, Select *v-x*
2	Add with carry	Read A, Read B, Write A	Select *p-t*, Select *u-w*, Select *v-x*

A CPU's data path contains circuit to perform arithmetic and logic microoperations on words such as fixed point and floating point numbers

6.3 DESIGN CONTROL UNIT :

To generate the control signal in proper , a wide variety of techniques exist. Most of these techniques, however, fall into one of the two categories.

a. Hardwired control unit

b. Microprogramed control

Control unit can be implemented by hardwired or by microprogram. Computer engineering strives to optimize three aspects of control unit design:

1. The complexity (hence cost) of the control unit
2. The speed of control unit
3. The engineering cost of the design (time, correctness etc)

6.3.1 HARDWIRED CONTROL UNIT

In the past, hardwired control unit is very difficult to design hence its engineering cost is very high. Presently, the emphasis of computer design is the performance therefore hardwired is the choice. Also the CAD tool for logic design has improved to the point that a complex design can be mostly automated. Therefore almost all processors of today use hardwired control unit.

The behavioral description of the control unit, shown by State diagram .Most states are simply driven by clock and only transition to next state.

event : go to next state

event : go to state 1 or state 2 depends on conditionals

event : branch to many states by decoding

Design method: The control unit design involves various factors like the amount of hardware used, speed of operation, cost of design process. We consider two systematic methods to design the hardwired controllers.

1. Method 1 : The classical method of the sequential circuit design . It attempt to minimize the amount of hardware in particular, by using $[\log_2 p]$ flip flops to realize a P-state circuit.
2. Method 2: An approach that use one flip flop per state and is known as one hot method. While expensive in terms of flip flops, this method simplifies CU design and debugging.

State table:The most general approach to hardwired control unit design is based on the state table method. In the state table method, the design of a hardwired control unit is just like the design of other sequential circuits. The first step is to construct a state table that describes the behavior of the control unit. The state table is actually a more detailed state transition and control signal activation description than the control flow graph. The Moore machine model is usually more desirable than the Mealy machine model because the control signals generated are independent of the input of the present cycle. This allows the control unit state to directly determine the control signals. Eliminating input condition signals from the path that delivers control signals often improves the operation speed of the control unit.

The rows of the state table correspond to the set of internal states $\{S_i\}$ .these states are determine by the information stored in the machine at discrete points of time (clock cycle) . let X and Z denote the input and output variables. The column correspond to the combination of the X signals that can be applied to the machine and denoted by $\{ I_j \}$ .The entry in row Si and column Ij has the form Si,j,Oi,j where Si,j is the next state of machine that result from the application of input combination Ij and Oi,j denote the output signal that appear on Z whenever the machine is in state Si with input Ij applied .If output value depend only on current state and independent of input combination . If all the output of this type the circuit is call Moore machine.

GCD processor using Classical method : gcd is greatest common divisor gcd(X,Y) of two positive integer that divide exactly into both X and Y . for example gcd(12,18) =6

```
gcd(in: X,Y ;out : Z);
registe XR, YR ,TEMPR;
XR:=X; YR:=Y;              (INPUT THE DATA )
While XR>0 do begin
If XR<= YR then begin (SWAP XR AND YR)
TEMPR:=YR;YR:=XR;
XR:=TEMPR ; end
XR:=XR-YR;                 (SUBTRACT YR FROM XR)
End
Z:=YR;                     (OUTPUT THE RESULT)
End gcd;
```

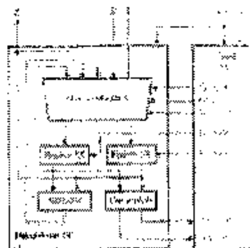


Figure 3: control unit for gcd processor

We can identify the set of state of control unit by examining the behavior definition .A start state S0 is entered when Reset become 1; this state also load X and Y in to DP registers. The subsequent action of gcd processor are swap and subtract ,for which we define the states S1and S2 respectively. A final state is S3 is entered when gcd(X,Y) has been computed.
If the input control signal (XR>0)=0, indicate that the while loop should be skipped , a transaction state So to S3 .If (XR>0)=1 ,the while loop is entered , and a transaction state is made to S1 to perform a swap if (XR>=YR) =0; other wise perform subtraction . the later case defines third entry in the state table ,whose input combination is (XR>0)(XR>=YR)=11.because subtraction is always followed by swap so the second row entries are S2.

State	Input (XR>0)(XR>=YR) 0-	10	11	Output Subtract	Swap	Select XY	Load XR	Load YR
S_0(begin)	S_3	S_1	S_2	0	0	1	1	1
S_1(swap)	S_2	S_2	S_2	0	1	0	1	1
S_2(subtract)	S_3	S_1	S_2	1	0	0	1	1
S_3(End)	S_3	S_3	S_3	0	0	0	0	0

Fig : State table defining the control unit of the GCD processor.

6.3.1.1 CLASSICAL METHOD: The major steps of classical method are as follows:

1. Construct a P-row state table that defines desire input output behavior.
2. Select the minimum number p of D-type flip-flops and assign a –bit binary code to each state .
3. Design a combinational circuit C that generate the primary output signals{zi} and S the secondary outputs {Di} that must be applied to flip flops.

We use this method to design control unit. Since there are four states , we require two flip flops, whose outputs $D_1D_0 = Y_0Y_1$ define control unit 's internal state . we assign the binary patterns to the four states in the following way:

S_0 =00, S_1=01,S_2=10,S_3=11

The D flip flop characteristic equation $Di^+(t+1) = Di(t)$ define the inputs D_1^+ and D_0^+ to the flip flops. CU's combinational logic C and be derived from the excitation table using any manual and automatic method. Suppose two level Sum of products(SOP) minimization .It is easily checked that C is define by the following SOP equations . All the AND-OR- SOP circuit can be changed to NAND s to produce NAND –NAND realization of original function.

$$D_1^+ = \overline{(XR>0)} + (XR>=YR) + D_0$$

$$D_0^+ = D_1 . D_0 + \overline{(XR>=YR)} . D_0 + \overline{(XR>0)} . \overline{D_0}$$

$$Subtract = D_1 . \overline{D_0}$$

$$Swap = \overline{D_1} . D_0$$

$$select\ XY = \overline{D_1} . \overline{D_0}$$

$$Load\ XR = \overline{D_1} + \overline{D_0}$$

$$Load\ YR = \overline{D_1}$$

FIGURE 4: all NAND classical design for the control unit of the gcd processor

6.3.1.2. ONE HOT METHOD :

An alternative approach that simplifies the design processes and gives C a regular and predictable structure ,is the one hot method , so called because its binary state assignment always contains a single 1 the hot bit while all the remaining bits while all the remaining bits are 0. Thus the state assignment for a four-state machine like the gcd processor takes the following form:

S_0=0001, S_1=0010, S_2=0100, S_3=1000 in general p flip flops are need to represent P states ,so the one hot method is restricted to fairly small values of P. Suppose that state S_i in a one hot design has the hot variable Di. further ,suppose that $I_{j,1}, I_{j,2 \ldots\ldots} I_{j,n}$ denote all input combinations that causes the state transitions to Sj to Si . then each AND combinations that causes transitions to Si . we can write

$$Di^+ = D_1(I_{1,1} + I_{1,2} + \ldots\ldots I_{1,n1}) + D_2(I_{2,1} + I_{2,2} + \ldots\ldots, I_{2,n2}) + \ldots\ldots$$

This immediate yield the SOP form

$$Di^+ = D_1 I_{1,1} + D_1 I_{1,2} + \ldots\ldots D_1 I_{1,n1}) + D_2 I_{2,1} + D_2 I_{2,2} + \ldots\ldots D_2 I_{2,n2}) + \ldots\ldots$$

Which is practical implicated with AND –OR and NAND-NAND circuit . for gcd processor's CU . State S_1 as next state only for S_0 and S_2 , in each case with the input combination

$$(XR>0).\ \overline{(XR >= YR)}\ .$$

The entire state of next state and the output equation obtained by applying to gcd processor's CU follows

[illegible]

Using De-Morgan law

[illegible]

[illegible]

[illegible]

Subtract = [illegible]

Swap = [illegible]

Select XY = [illegible]

Load XR = [illegible]

Load YR = [illegible]

Excitation table for the control unit of GCD processor'

The steps of the one hot method for moore machine can be summarize as follows

1. Construct a P-row state table that defines the desired input –output behavior.
2. Associate a separate D-type flip-flop D_i with each state Si, and assign the P-bit one hot binary code $D_0, D_1, \ldots, D_{i-1}, D_i, D_{i+1}, \ldots D_p$ =0,0….0,1,0,….0 to Si .
3. Design a combinational circuit C that generates the primary and secondary output signals { D_i } and {Z_k} , respectively. D_i^+ define by logical equation

$$D_i^+ = \sum_{i=1}^{p} D_i (I_{j,1} + I_{j,2} + \ldots\ldots I_{j,nj})$$

Where $I_{j,1}, I_{j,2}, \ldots\ldots I_{j,nj}$ denotes all the input combinations that causes transition from Sj to Si . If Z_k =1 (active) only in rows k,h for h=1,2,….m_k then Z_k is define by

$$Z_k = D_{k1} + D_{k2} + D_{k3} \ldots + D_{k\,mk} = \overline{(\overline{D_{k1}}\,\overline{D_{k2}} \ldots\ldots \overline{D_{kmk}})}$$

6.3.2 MICRO PROGRAMMED CONTROL UNIT

The idea of microprogrammed control units was introduced by M. V. Wilkes in the early 1950s. Microprogramming is a method control of control unit design in which control signals selection and sequencing information stored in ROM or RAM called control memory CM . The control signals are activated at any time are specified by a microinstruction , which is fetch by control memory same way as fetched from main memory. A set of related microinstruction forms a micro program .Micro program can be changed easily by changing the content of CM. So microprogramming is more flexible as hardwired control. But it required extra hardware cost due to the control memory. There is also the performance penalty due to the time require to access the microinstruction from CM. microprogramming is use in CISC as Pentium and 680X0.A micro assembler is necessary to translate micro programs into executable program that can be stored in control memory .

The reason for microprogramming

Microcode was originally developed as a simpler method of developing the control logic for a computer. Initially CPU instruction sets were "hard wired". Each step needed to fetch, decode and execute the machine instructions (including any operand address calculations, reads and writes) was controlled directly by combinatorial logic and rather minimal sequential state machine circuitry. While very efficient, the need for powerful instruction sets with multi-step addressing and complex operations (see below) made such "hard-wired" processors difficult to design and debug; highly encoded and varied-length instructions can contribute to this as well, especially when very irregular encodings are used.

Microcode simplified the job by allowing much of the processor's behaviour and programming model be defined via microprogram routines rather than by dedicated circuitry. Even late in the design process, microcode could easily be changed, whereas hard wired CPU designs were very cumbersome to change, so this greatly facilitated CPU design.

Architectures with instruction sets implemented by complex microprograms included the IBM System/360 and Digital Equipmen Corporation VAX. The approach of increasingly complex microcode-implemented instruction sets was later called CISC. A middle way, used in many microprocessors , is to use PLAs and/or ROMs (instead of combinatorial logic) mainly for instruction decoding, and let a simple state machine (without much, or any, microcode) do most of the sequencing. The various practical uses of microcode and related techniques (such as PLAs) have been numerous over the years, as well as approaches to where, and to which extent, it should be used. It is still used in modern CPU designs.

Other benefits

A processor's microprograms operate on a more primitive, totally different and much more hardware-oriented architecture than the assembly instructions visible to normal programmers. In coordination with the hardware, the microcode implements the programmer-visible architecture. The underlying hardware need not have a fixed relationship to the visible architecture. This makes it possible to implement a given instruction set architecture on a wide variety of underlying hardware micro-architectures.

Doing so is important if binary program compatibility is a priority. That way previously existing programs can run on totally new hardware without requiring revision and recompilation. However there may be a performance penalty for this approach. The tradeoffs between application backward compatibility vs CPU performance are hotly debated by CPU design engineers.

The IBM System/360 has a 32-bit architecture with 16 general-purpose registers, but most of the System/360 implementations actually use hardware that implemented a much simpler underlying microarchitecture; for example, the System/360 Model 30 had 8-bit data paths to the arithmeti logic unit (ALU) and main memory and implemented the general-purpose registers in a special unit of higher-speed core memory, and the System/360 Model 40 had 8-bit data paths to the ALU and 16-bit data paths to main memory and also implemented the general-purpose registers in a special unit of higher-speed core memory. The Model 50 and Model 65 had full 32-bit data paths and implemented the general-purpose registers in faster transistor circuits. In this way, microprogramming enabled IBM to design many System/360 models with substantially different hardware and spanning a wide range of cost and performance, while making them all architecturally compatible. This dramatically reduced the amount of unique system software that had to be written for each model.

A similar approach was used by Digital Equipment Corporation in their VAX family of computers. Initially a 32-bit TTL processor in conjunction with supporting microcode implemented the programmer-visible architecture. Later VAX versions used different microarchitectures, yet the programmer-visible architecture didn't change.

Microprogramming also reduced the cost of field changes to correct defects (bugs) in the processor; a bug could often be fixed by replacing a portion of the microprogram rather than by changes being made to hardware logic and wiring.

Examples of microprogrammed systems

- Most models of the IBM System/360 series were microprogrammed:
- The Model 25 was unique among System/360 models in using the top 16k bytes of core storage to hold the control storage for the microprogram. The 2025 used a 16-bit microarchitecture with seven control words (or microinstructions). At power up, or full system reset, the microcode was loaded from the card reader. The IBM 1410 emulation for this model was loaded this way.
- The Model 30, the slowest model in the line, used an 8-bit microarchitecture with only a few hardware registers; everything that the programmer saw was emulated by the microprogram. The microcode for this model was also held on special punched cards, which were stored inside the machine in a dedicated reader per card, called "CROS" units (Capacitor Read-Only Storage). A second CROS reader was installed for machines ordered with 1620 emulation.
- The Model 40 used 56-bit control words. The 2040 box implements both the System/360 main processor and the multiplex channel (the I/O processor). This model used "TROS" dedicated readers similar to "CROS" units, but with an inductive pickup (Transformer Read-only Store).
- The Model 50 had two internal datapaths which operated in parallel: a 32-bit datapath used for

arithmetic operations, and an 8-bit data path used in some logical operations. The control store used 90-bit microinstructions.

- The Model 85 had separate instruction fetch (I-unit) and execution (E-unit) to provide high performance. The I-unit is hardware controlled. The E-unit is microprogrammed with 108-bit control words.
- The Digital Equipment Corporation PDP-11 processors, with the exception of the PDP-11/20, were microprogrammed.
- The Burroughs B700 "microprocessor" executed application-level opcodes using sequences of 16-bit microinstructions stored in main memory, each of these was either a register-load operation or mapped to a single 56-bit "nanocode" instruction stored in read-only memory. This allowed comparatively simple hardware to act either as a mainframe peripheral controller or to be packaged as a standalone computer.
- The Burroughs B1700 was implemented with radically different hardware including bit-addressable main memory but had a similar multi-layer organisation.
- The NCR 315 was microprogrammed with hand wired ferrite cores (a ROM) pulsed by a sequencer with conditional execution. Wires routed through the cores were enables for various data and logic elements in the processor.
- In common with many other complex mechanical devices Charles Babbage's analytical engine used banks of cams to control each operation, i.e. it had a read-only control store. As such it deserves to be recognised as the first microprogrammed computer to be designed, even if it has not yet been realised in hardware.

The VU0 and VU1 vector units in the Sony Playstation 2 are micro programmable; in fact, VU1 was *only* accessible via microcode for the first several generations of the SDK.

BASIC LAYOUT OF A MICROPROGRAMMED CONTROL UNIT

MICROPROGRAMMED CONTROL UNIT WITH A MICROPROGRAM COUNTER (μPC)

Microcode is a layer of lowest-level instructions involved in the implementation of machine code instructions in many computers and other processors; it resides in a special high-speed memory and translates machine instructions into sequences of detailed circuit-level operations. It helps separate the machine instructions from the underlying electronics so that instructions can be designed and altered more freely. It also makes it feasible to build complex multi-step instructions while still reducing the complexity of the electronic circuitry compared to other methods. Writing microcode is called microprogramming and the microcode for a given processor is often called a microprogram.

The microcode is normally written by the CPU engineer during the design phase. It is generally not meant to be visible or changeable by a normal programmer, nor even an assembly programmer. Unlike machine

code which often retains backwards compatibility, microcode only runs on the exact CPU model for which it's designed. Microcode can be used to let one microarchitecture emulate another, usually more powerful, architecture.

Some hardware vendors, especially IBM, also use the term *microcode* as a synonym for *firmware*, whether or not it actually implements the microprogramming of a processor. Even simple firmware, such as the one used in a hard drive, is sometimes described as microcode. Such use is not discussed here.

MICROCODE EXECUTION

1. Op-code is decoded.
2. Microinstructions are retrieved from control memory (control address register and the decoder serve as the address register and selection mechanism or control unit).
3. The control address register locates the microinstruction to be retrieved from control memory.
4. The microinstruction register holds the retrieved microinstruction - micro opcode and address of the next microinstruction in the control memory
5. Current microinstruction is executed.
6. The address of the next microinstruction is entered into the control memory to retrieve the next micro-instruction.
7. If all microinstructions were executed, then store next op-code of conventional instruction in the control address register, if not, execute remaining microinstruction.
8. Conditional jumps are implemented by letting the states of some conditional flip-flops modify the address of the next microinstruction to be retrieved.

Control unit organization: A microinstruction has two parts a control field that specify the control signals and address field that contain address in CM of the next instruction to be executed. Maurice V. Wilkes, the inventor of microprogramming, each bit Ki of a control field corresponding to a distinct control line C_i. When K_i =1 in the current microinstruction, C_i is activated other wise Ci remain inactive. A ROM implements the control memory CM. The left part of ROM decodes an address obtained from a control memory address register (CMAR). Each address select a particular row in the right part (OR plane) of the ROM IT contain 6bit control field and three bit address. When address 000 is selected the control signals C_0 C_2,C_4 are activated, as indicated by Xs in the control field. At the same time content of $a_2a_1a_0$ =001 are sent to CMAR. Where they are stored the address of the next instruction to be executed. CMAR is loaded from external source as well as from the address from the address field of the microinstruction. The external source provide the starting address of micro program in CM. Many modifications to the preceding design have been proposed over the year. A major area of concern is the microinstruction word length. The microinstruction word length is determined by three factors :

1. The maximum number of simultaneous micro operation that must be specified that is the degree of parallelism required at the micro operation level.
2. The way in which control information is represented or encode.
3. The way in which next microinstruction address is specified.

Figure 5 : .Basic structure of a microprogrammed control unit.

The control memory are usually ROM. so their content are cannot be change. So there is no need to

change the CM except to correct the design error to make minor enhancements to the system. The CM could be RAM (read –write memory). Wilkes observe that such a device called a write able control memory. (WCM). A processor with WCM is said dynamically micro programmable because the control memory content can be changed .

6.4 PARALLELISM IN MICROINSTRUCTIONS

- Microinstruction formats take advantage of the fact that, at the microprogramming, many operations can be performed in parallel.
- If all useful combinations of parallel microoperations were specified by a single opcode,the number of opcodes would, in most cases, be enormous.
- An opcode decoder of considerable complexity would be needed. To avoid these difficulties, it is usual to divide the microoperation specification part of a microinstruction into k disjoint control fields.
- Each control field handles a limited set of microoperations,any one of which can be performed simultaneously with the microoperations specified by the remaining control fields.
- A control field often specifies the control-line values for a single device such as an adder, a register, or a bus.
- There is a 1 –bit control field for every control line in the system.

- The scheme with a control field for every control signal is wasteful of control memory space because most of the possible combinations of control signals are never used.

Consider, for instance, the register R which can be loaded from any of four independent sources under the control of the four separate signals c0,c1,c2,c3, as indicated.

A straightforward implementation of the associated control points using an encoder and a multiplexer .

Suppose that the ci's are derived from a microinstruction control field in which there is 1 bit for each control signal. This results in the 4-bit control field .

Only the five control-field patterns ,since any other pattern will create a conflict by attempting to load R from two or more independent sources simultaneously.

The uuencoded format fig (a), has the advantage that all the control signals are individually identified in, and can be obtained directly from, the microinstruction.

The encoded control signals k_0,k_1,k_2 of fig.(b) must be passed through a decoder if we wish to extract the four original control signals c_0,c_1,c_2,c_3.

Often we can use the encoded control signals directly so that no decoding is needed.

Example: We can connect the two signals k_1,k_2 of fig.(b) directly to the select inputs S of the multiplexer in fig.(b),thereby eliminating the priority encoder.

The complemented control signal k_0 can then be connected directly to the LOAD input of the register R to complete the design.

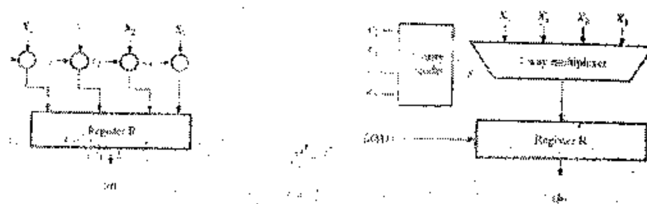


Fig : Control field for the circuit of (a) unencoded format and (b)encoded format

6.5 HORIZONTAL VERSUS VERTICAL

Microinstructions can be classified as horizontal or vertical. Individual bits in horizontal microinstructions correspond to individual control lines.

Horizontal microinstructions have the following characteristic :

1. Long formats
2. Ability to express high degree of parallelism
3. Each bit controls a single control line.
4. Little encoding of control information.

Vertical microinstructions have following characteristic :

1. Short formats.
2. Little ability to express parallelism.
3. Little encoding of control information

control lines are coded into specific fields within a microinstruction. Decoders are needed to map a field of k bits to 2k possible combinations of control lines. For example, a 3-bit field in a microinstruction could be used to specify any one of eight possible lines. Because of the encoding, vertical microinstructions are much shorter than horizontal ones. Control lines encoded in the same field cannot be activated simultaneously. Therefore, vertical microinstructions allow only limited parallelism. It should be noted that no decoding is needed in horizontal microinstructions while decoding is necessary in the vertical case.

Definition based on the degree of encoding:

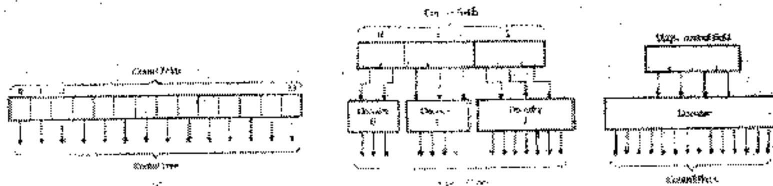
- A horizontal microinstruction format allows no encoding of control information, whereas a vertical format does.

Definition based on degree of parallelism:

- A vertical microinstruction can specify only one microoperation(no parallelism).

These definitions are not independent, since a large amount of parallelism implies little encoding, and vice versa.

Example: The format of fig.6(a)is *horizontal* and that of fig.6(c) is *vertical* under both of the preceding definition.



6.6 ADVANTAGES & APPLICATIONS OFMICROPROGRAMMING

1) THE SYSTEMATIZATION OF CONTROL

2) IMPROVEMENT IN PERFORMANCE

a) a high degree of parallelism in data paths e.g., multiple bit microinstructions are performed in one cycle

b) a high degree of decision logic (in table search and sorting routines)

3) COMPUTER-SERIES COMPATIBILITY

Compatibility of instruction sets between smaller and larger machines of a series, e.g., Intel286, 386, Pentium, IBM Systems/309x, Motorola 68000 series.

4) EMULATION

Emulation is the combined software/hardware interpretation of the machine instruction of one machine by another. Target's machine architecture is mapped onto the host machine.

EMULATOR - a set of microprograms that interpret a particular instruction set or language L1. Computer C1 emulates computer C2 if it can interpret machine language L2.

5) MICRODIAGNOSTICS

Microprogramming diagnostic routines have allowed refinements and increased the speed of detecting and localizing faults,including error detection and correction of microstorage itself.

a) software diagnostics

b) hardware diagnostics (test generation methods)

c) microdiagnostics

6) SOFTWARE SUPPORT: eases programming

7) SPECIAL-PURPOSE DEVICES

e.g, special processors for data communication, data aquisition, device controllers

8) DYNAMIC MICROPROGRAMMING

This allows routines to be easily microprogrammed. Computer can be restructured to represent any instruction vocabulary by use of writable control memory (WCM). It allows the instruction set of the machine to be changed and be TAILORED to specific

6.7 MULTIPLIER CONTROL UNIT

Fixed-point multiplication requires substantially more hardware than fixed-point addition. Multiplication is usually implemented by some form of repeated addition. A simple but slow method to compute $X \times Y$ is to add the multiplicand Y itself to X times, where X is multiplier. The multiplication of 2s-compltement numbers presents some more difficulties. The Figure-1 shows flow of data of 2s-complement multiplier. The HDL description of the multiplier for 8-bit 2s-complement numbers are given .

The block diagram of the multiplier's data path unit shown in Figure redrawn in expanded form in Figure-7 to show a set of control points, which represent abstractly the control signals and associated logic gates needed to link CU and DP. These control signals are derived from the multiplication algorithm in Figure-9. The statement labeled BEGIN in Figure-9, for instance, requires the register A, COUNT, and F to be reset simultaneously to the all-zero state. A single control signal c_{10} is therefore provided for this purpose, it can be connected directly to the CLEAR inputs of the three registers in question, and so no additional logic is needed to implement the c_{10} control point. Control signal c_8 and c_9 transfer a data word from the input bus INBUS to register Q and M, respectively, and are shown in the corresponding data paths of Figure-9; these signals may be connected to the registers' LOAD inputs. Figure-10 introduces a control signal called COUNT7, which is set to 1 when COUNT = 111_2 and is set to 0 otherwise. COUNT7, the right-most bit Q[0] of multiplier register Q and the external BEGIN signal serve as the primary inputs to CU.

The flowchart resembles a state transition graph that describes the behavior of both the control and datapath units. To obtain a state table for the control unit CU, we associate a state S_i with every operation block in Figure-10, leading to the seven states labeled S_1:S_7. An additional state S_0 represents the reset or waiting condition of the control unit. CU has three primary inputs signals—BEGIN, Q[0], and COUNT7; hence there are eight possible input combination. Figure-10 shows a eight-state state table, which is derived directly from Figure-9

CU can be directly implemented from the state table of Figure-10, or, equally easily, from the flowchart of Figure-7 by the one-hot method. Eight flip-flops are needed to accommodate CU's eight states S_0:S_7. The next state equations are

$$D_0^+ = D_0.\overline{BEGIN} + D_7$$

$D_1^+ = D_0 . BEGIN$

$D_2^+ = D_1$

$D_3^+ = D_2 .Q[0] + D_4 .Q[0] . \overline{COUNT7}$

$D_4^+ = D_2 .Q[0] + D_3 + D_4 .Q[0] . \overline{COUNT7}$

$D_5^+ = D_4 . Q[0] . COUNT7$

$D_6^+ = D_5 + D_4 .Q[0] . \overline{COUNT7}$

$D_7^+ = D_6$

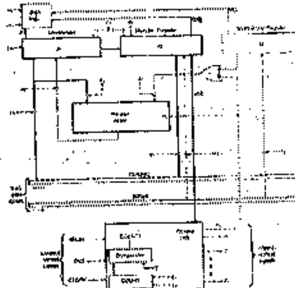


Figure-7: Twos-complement multiplier with a set of control points

The output equations are

$c_0 = c_1 = c_{11} = D_4$

$c_2 = c_3 = c_4 = D_3 + D_5$

$c_5 = D_5$

$c_6 = D_6$

$c_7 = D_7$

$c_8 = D_2$

$c_9 = c_{10} = D_1$

$END = D_0$

Control Signal Operation controlled

c_0 Set sign bit of A to F.

c_1 Right-shift register-pair A.Q.

c_2 Transfer adder output to A.

c_3 Transfer A to left input of adder.

c_4 Transfer M to right input of adder.

c_5 Perform subtraction (correction). Clear Q[0].

c_6 Transfer A to output bus.

c_7 Transfer Q to output bus.

c_8 Transfer word on input bus to Q.

c_9 Transfer word on input bus to M.

c_{10} Clear A, COUNT, and F registers.

c_{11} Increment COUNT.

END Completion signal (CU idle).

Control signals for two's complement multiplier

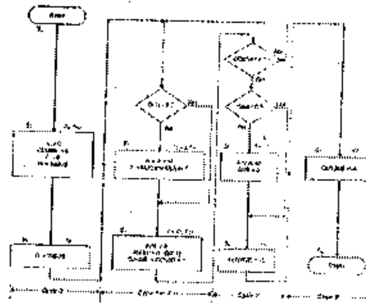


Figure-9: Flowchart for the twos-complement multiplier

State	Inputs: BEGIN Q[0] Count7 000	001	010	011	100	101	110	111	Outputs c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	END
S_0	S_0	S_0	S_0	S_0	S_1	S_1	S_1	S_1	0	0	0	0	0	0	0	0	0	0	0	0	1
S_1	S_2	S_2	S_2	S_2	S_2	S_2	S_2	S_2	0	0	0	0	0	0	0	0	0	1	1	0	0
S_2	S_3	S_3	S_3	S_3	S_3	S_3	S_3	S_3	0	0	0	0	0	0	0	0	1	0	0	0	0
S_3	S_4	S_4	S_4	S_4	S_4	S_4	S_4	S_4	0	0	1	1	1	0	0	0	0	0	0	0	0
S_4	S_5	S_5	S_5	S_5	S_5	S_5	S_5	S_5	1	1	0	0	0	0	0	0	0	0	0	1	0
S_5	S_6	S_6	S_6	S_6	S_6	S_6	S_6	S_6	0	0	1	1	1	1	0	0	0	0	0	0	0
S_6	S_7	S_7	S_7	S_7	S_7	S_7	S_7	S_7	0	0	0	0	0	0	1	0	0	0	0	0	0
S_7	S_0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0	0	0	0	0	0	1	0	0	0	0	0

Figure :10 State table for multiplier control unit

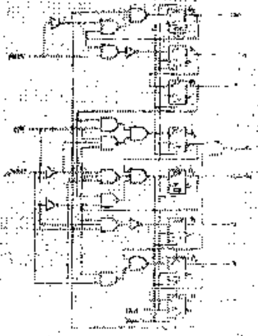


Figure 11: All NAND classical design for multiplier control unit

Fig : 11 show the NAND circuit is implement by the output equations. Despite having more flip-flop ,one hot method which is show in Fig 12 is better in many ways than the classical design one hot design has generally smaller umber of gates .the hot design is easy to design and understand .

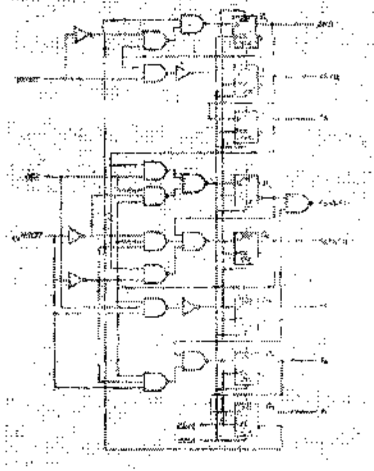


Figure 12: All NAND one hot design for multiplier control unit

6.8 SUMMARY

The CPU is the part of a computer that interprets and carries out the instructions contained in the programs we write. The CPU's main components are the register .file,ALU, and the control unit. The register .file contains general-purpose and special registers .General-purpose registers may be used to hold operands and intermediate results. The special registers may be used for memory access, sequencing, status information, or to hold the fetched instruction during decoding and execution. Arithmetic and logical operations are performed in the ALU. Internal to the CPU,
data may move from one register to another or between registers and ALU. Data may also move between the CPU and external components such as memory and I/O. The control unit is the component that controls the state of the instruction cycle. As long as there are instructions to execute, the next instruction is fetched from main memory. The instruction is executed based on the operation speci.ed in the op-code .eld of the instruction. The control unit generates signals that control the .ow of data within the CPU and between the CPU and external units such as memory and I/O. The control unit can be implemented using hardwired or microprogramming techniques.

6.9 GLOSSARY

Control Unit (CU) - That part of the computer which accesses instructions in sequence, interprets them and then directs their implementation.
Memory Unit - Part of the computer where data and instructions are held. (Also known as main memory, main store, central memory, immediate access memory.)
Hard wired control : hardwired control unit contain fixed logical circuit .

Microprogram control unit: Microprogram control unit is constructed by microinstruction use control memory which is costly.

6.10 FURTHER READINGS

J. P. Hayes, Computer Architecture and Organization, McGraw-Hill, New York, 1998.
W. Stallings, Computer Organization and Architecture: Designing for Performance, NJ, 1996.

6.11 UNIT END QUESTIONS

1. Design a hardwired control unit to perform A-B operation.
2. Design a hardwired control unit to find GCD(greatest common factor).
3. What are the main differences between the following pairs?
   (a) Vertical and horizontal microinstructions
   (b) Microprogramming and hardwired control
4. Using the single-bus architecture, generate the necessary control signals, in the proper order (with minimum number of micro-instructions), for conditional branch instruction.
5. Write a micro-program for the fetch instruction using the one-bus datapath and the two-bus datapath.

Unit-07
Memory Organization

Structure of the Unit

7.0 Objectives
7.1 Introduction
7.2 Memory Technology
7.2.1 Memory Device Characteristics
7.2.2 Random-Access Memories
7.2.3 Serial-Access Memories
7.3 Memory Systems
7.3.1 Multilevel Memories
7.3.2 Address Translation
7.3.3 Memory Allocation
7.4 Computer Memory
7.4.1 Internal Memory
7.4.1.1 Computer Memory System Overview
7.4.1.2 Semi Conductor Main Memory
7.4.1.3 Cache Memory
7.4.1.3.1 Main Features
7.4.1.3.2 Address Mapping
7.4.1.3.3 Structure versus Performance
7.4.1.4 Advanced Dram Organization
7.4.2 External Memory
7.4.2.1 Magnetic Disk
7.4.2.2 RAID
7.4.2.3 Optical Memory
7.4.2.4 Magnetic Tape
7.5 Summary
7.6 Glossary
7.7 Further Readings
7.8 Answer to Self Learning Exercise
7.9 Unit End Questions

7.0 Objective

This unit is concerned with computer's memory system and its impact on performance. The characteristics of the most important storage-device technologies are discussed. The behavior and management of multilevel hierarchical memory system are explained. Different types of memories are also explained.

7.1 Introduction

From the previous unit we have learned the basic concepts of control design with design examples. We have also discussed concepts of micro programmed control. We have discussed micro control unit as well as CPU control unit.

7.2 Memory Technology

Every computer contains several types of devices to store the instructions and data required for its operation. These storage devices plus the algorithms implemented by hardware and/or software-needed to manage the stored information form the memory system of the computer.

7.2.1 Memory Device Characteristics

A CPU should have rapid, uninterrupted access to the external memories where its programs and the data they process are stored so that the CPU can operate at or near its maximum speed. Unfortunately, memories that operate at speeds comparable to processor speeds are expensive, and generally only very small

systems can afford to employ a single memory using just one type of technology. Instead, the stored information is distributed, often in complex fashion, over various memory units that have very different performance and cost.

Memory types The information-storage components of a computer can be placed in four groups.

· CPU registers- These high-speed registers in the.CPU serve as the working memory for temporary storage of instruction and data. They usually form a general purpose register file for storing data as it is processed. A capacity of 32 data words is typical of a register file, and each register can be accessed, that is read from or written into within a single clock cycle (a few nanosecond)

· Main (primary) memory This large, fairly fast external memory stores programs and data that are in active use. Storage location in main memory are addressed directly by the CPU's load and store instruction. Access times of five or more clock cycles are usual.

· Secondary memory. This memory type is much larger in capacity but also much slower than main memory. Secondary memory stores system programs, large data files, and the like that are not continually required by the CPU. It also acts as .an overflow memory when the capacity of main memory is exceeded. Information in secondary storage is considered to be on-line but is accessed indirectly via input/output programs that transfer information between main and secondary memory.

· Cache The cache is much faster than main memory because some or all of it can reside on the same IC as the CPU. Unlike the three other memory types, caches are normally transparent to the programmer. Together, a computer's caches and main memory implement the external memory M addressed directly by CPU instructions.

Performance and cost The most meaningful measure of the cost of a memory device is the purchase price to the user of a complete unit. The price should include not only the cost of the information storage medium itself but also the cost of the peripheral equipment: (access circuitry) needed to operate the memory. Let C be the price in dollars of a complete memory system with S bits of storage capacity. We define the cost c of the memory as follows: .

$c = C/S$ dollars/bit

The performance of an individual memory device is primarily determined by the rate at which information can be read from or written into the memory. A basic performance measure is the average time to read a fixed amount of information, for instance, one word, from the memory. This parameter is called the *read access time,* or simply the *access time,* of the memory and is denoted by t_A. The write access time is defined similarly; it is often, but not always, equal to the read access time. The access time depends on the physical nature of the storage medium and on the access mechanisms used. It is calculated from the time the memory receives a read request to the time at which the requested information becomes available at the memory's output terminals.

Clearly, low cost and short access time are desirable memory characteristics; unfortunately, they also tend to be incompatible. Memory units with fast access are expensive, while low-cost memories are slow.

Access modes A fundamental characteristic of a memory is the order or sequence in which information can be accessed. If storage locations can be accessed in any order and access time is independent of the location being accessed.

Each storage location in a RAM can be accessed independently of the other locations. In serial memories the access mechanism is shared by storage locations and must be assigned to different locations at different times by moving the stored information, the read-write head or both.

In serial access memories however, the time required to bring the desired location into correspondence with a read-write head increases the effective access time, so serial access tends to be slower than random access. Thus the type of access mode contributes significantly to the inverse relationship between cost and access time.

Memory devices such as magnetic hard disks and CD-ROMs contain many rotating storage tracks. If each track has its own read-write head, the tracks can be accessed randomly, but access within each track is serial in such cases access mode is semirandom.

Memory retention The method of writing information into a memory can be permanent or irreversible in that once information has been written, it cannot be altered while the memory is in use or on-line. Printing on paper is an example of a permanent storage technique. Memories whose contents cannot be altered on-line- if they can be altered at all- are read-only memories (ROMs). A ROM is therefore a nonerasable storage device. ROMs are widely used to store control Programs such as microprograms. Compact Disk (CD) ROMs are a class of nonerasable secondary memory devices.
Semiconductor ROMs whose contents can be changed off-line-and with some difficulty-are called programmable read only memories (PROMs).
Memories in which reading or writing can be done with impunity on-line are called read-write memories to differentiate them from ROMs. All memories used temporary storage purposes are read-write memories. In some technologies the stored information is lost over a period of time unless corrective action is taken. Three characteristics of memories that destroy information in this way are destructive readout, dynamic storage and volatility. In some memories the method of reading the memory destroys the information; this phenomenon is called destructive readout (DRO). Memories in-which reading does not affect the stored data have nondestructive read(NRDO).
Another physical process that can destroy the contents of a memory is the removal or failure of its power supply. A memory is volatile if the loss of power destroys the stored information. Information can be stored indefinitely in a volatile memory by providing battery backup or other means to maintain a continuous supply of power. Most IC- memories are volatile, while most magnetic and optic memories are nonvolatile Figure 7.1 summarizes these characteristics for some important contemporary memory technologies.

Technology	Primary Storage medium	Access Mode	Alterability	Permanence	Typical access time TA
Bipolar Semiconductor	Electronic	Rando-m	Read/Wri-te	NDRO, volatile	10 ns
Metal Oxide Semiconductor (MOS)	Electronic	Rando-m	Read/Wri-te	DRO or NDRO, volatile	60 ns
Magnetic (hard) disk	Magnetic	Semira-ndom	Read/Wri-te	NDRO, nonvolatile	10 ms
Magnetic-optical disk	Optical	Semira-ndom	Read/Wri-te	NDRO, nonvolatile	50 ms
Compact disk ROM	Optical	Semira-ndom	Read Only	NDRO, nonvolatile	100 ms
Magnetic tape cartridge	Magnetic	Serial	Read/Wri-te	NRDO nonvolatile	1s

Figure 7.1 Characteristics of some common memory technologies
Finally we mention reliability, which is measured by the mean time before failure (MTBF). In general memories with no moving parts have much higher reliability than memories such as magnetic disks, which involve considerable mechanical motion. Even in memories without moving parts, reliability problems arise, particularly when very high storage densities or data transferred rates are used. Error dectecting and error-correcting codes can Increase the reliability of any memory.

7.2.2 Random-Access Memories

RAMs are distinguished by the fact that each storage location can be accessed independently with fixed access and cycle times that are independent of the position of the accessed location.
Organization The main components of a RAM device is DRAM IC. At its heart is a storage unit composed of a large number *(2m)* of addressable locations, each of which stores a w-bit word. Individual bits are not directly addressable unless w = 1. A RAM of this sort is referred to as a 2m X w-bit or 2m-word memory. The RAM operates as follows: First the address of the target location to be accessed is transferred via the address bus to the RAM's address buffer. The address is then processed by the address decoder, which

selects the required location in the storage cell unit. A control line indicates the type *of* access *to* be performed. If a read operation (load) is requested, the contents *of* the addressed location are transferred from the storage cell unit *to* the data buffer and from there *to* the data bus. If a write (store) is requested, the word *to* be stored is transferred from the data bus *to* the selected location in the storage unit.
The storage unit is made up *of* many identical 1-bit memory cells and their- interconnections. The actual number *of* lines connected *to* the cell and their functions depend on the memory technology and the addressing scheme in use. Each cell is connected *to* a set *of* data, address, and control signals. One physical line often has several logical functions; for example, it can serve as both an address and data line. In each line connected *to* the storage cell unit, we can expect *to* find a driver that acts as either an amplifier or a transducer *of* physical signals. The drivers, decoders, and control circuits *form* the *access circuitry* *of* the RAM and can have a significant impact on the total size and *cost of* the memory.
A RAM's storage cells are physically arranged into regular arrays *to* reduce the *cost of* the connections between the cells and the access circuitry. The memory address is partitioned into *d* components *so* that the address A_i *of* cell C_i becomes a d-dimensional vector $(A_{i,1}, A_{i,2}, .., A_{i,d}) = A_i$. Each *of* the *d* parts *of* the address word goes *to* a separate address decoder and a separate set *of* address drivers. A cell is selected by simultaneously activating all *d of* its address lines. A memory unit with this kind *of* addressing is said *to* be *d-dimensional.*
The most common RAM organization is the *two-dimensional* (2-D) or *row- column* scheme,In this scheme *m-bit* address word is divided into two parts, X and *Y,* consisting of m_x and m_y bits, respectively. The cells are arranged in a rectangular array of $N_{x<=}2^{m_x}$ rows and $N_{y<=}2^{m_y}$ columns, so the total number of cells is $N = N_x N_y$. A cell is selected by the coincidence of signals applied to its X and Y address lines. The 2-D organization requires much less access circuitry than a 1-D organization for the same storage capacity. For example, if $N_x = N_y =$ "*N,* the number of address drivers needed is *2"N.*

Semiconductor RAMs Semiconductor memories in which the storage cells are small transistor circuits have been used for high-speed CPU registers since the 1950s. It was not until the development of VLSI in the 1970s that producing large RAM ICs suitable for main-memory and cache applications became economical. Single-chip RAMs can be manufactured in sizes ranging from a few hundred bits to 1 Gb or more. Both bipolar and MOS transistor circuits are used in RAMs, but MOS is the dominant circuit technology for large RAMs semiconductor memories fall into two categories- SRAMs and DRAMs- whose data-retention methods are static and dynamic, respectively.SRAMs consist of memory cells that resembles the flip-flops used in processor design. SRAM cells differ from flip-flops primarily in the methods used to address the cells and transfer data to and from them. Multifunction lines minimize storage-cell complexity and the number of cell connections, thereby facilitating the manufacture of very large 2-D arrays of storage cells.
In DRAM cell the 1 and 0 states correspond to the presence and absence of a stored charge in a capacitor controlled by a transistor switching circuit. Since a DRAM cell can be constructed around a single transistor, whereas a static cell requires up to six transistors, higher storage density is achieved with DRAMs.

7.2.3 Serial-Access Memories

The data in a serial-access memory must be accessed in a predetermined order via read-write circuitry that is shared by different Storage locations. Large Serial memories typically store information in a fixed set of tracks, each consisting of a sequence of I-bit storage cells. A track has one or more access points at which a read-write "head" can transfer information to or from the track. A stored item is accessed by moving either the stored information or the read–write heads or both. Functionally, a storage track in a serial memory resembles a shift register, so data transfer to and from a track is essentially serial.
Serial-access memories find their main application as secondary computer memories because of their low cost per bit and cost is achieved by using very simple and small storage cells. Long access time is due to several factors:

- The read-write head positioning time.

- The relatively slow speed at which the tracks move.
- The fact that data transfer to and from the memory is serial rather than parallel.

Access methods Serial memories such as magnetic hard disks can be divide into those where each track has one or more whose read-write heads are shared among differ read-write heads, the need to move the heads between tracks introduces a delay. The average time to move a head from another is the seek time t_s of the memory. Once the head is in position, the desired cell may be in the wrong part of the moving storage track. Some time is required for this cell t reach the read-write head so that data transfer can begin. The averag time for this movement to take place is the latency t_L of the memory. In memories where information rotates around a closed track, t_L is called the rotational Latency.
Each storage cell in a track stores a single bit. A w-bit word may be stored in two different ways. It can consist of w consecutive bits along a single track. Alternatively, w tracks may be used to store the word, with each track storing a different bit. By synchronizing the w tracks and providing a separate read-write head for each track, all w bits can be accessed simultaneously. In either case it is inefficient to read or write just one word per serial access, since the seek time and the rotational latency consume so much time. Words are therefore grouped into larger units called blocks. All the words in a block are stored in consecutive locations so that the time to access an entire block includes only one seek and one latency time.

Once the read-write head is positioned at the start of the requested word a block, data is transferred at a rate that depends on two factors: the speed of the stored information relative to the read-write head and the storage density along the track. The speed at which data can be transferred continuously to or from track under these circumstances is the data-transfer rate. If a track has a storage density of T bits/cm and moves at a velocity of V cm/s past the read-write head, then and a data-transfer rate is TV bits/s.
The time t_B needed to access a block of data in a serial-access memory can be estimated as follows. Let each track have a fixed (average) capacity of N words and rotate at r revolutions per second. Let n be the number of words per block. The data-transfer rate of the memory is then rN words/s. Once the read-write head is positioned at the start of the desired block, its data can be transferred in approximately n/(rN) seconds. The average latency is 1/(2r) seconds, which is the time needed for half a revolution. If ts is the average seek time, then an appropriate a formula for t_B is

$$t_B = t_s + 1/2r + n/rN$$

Memory organization Figure 7.2 shows the overall organization of a serial memory unit. Assume that each word is stored along a single track and that each access results in the transfer of a block of words. The address of the data to be accessed is applied to the address decoder, whose output determines the track to be used (the track address) and the location of the desired block of information within the track(the block address). The track address determines the particular read-write head to be selected. Then, If necessary, the selected head is moved into position to transfer data to or from the target track. The desired block cannot be accessed until it coincides with the selected head.

To determine when this condition occurs, a track-position indicator generates the address of block that is currently passing the read-write head. The generated address is compared with the block address produced by the address decoder. When they match, the selected head is enabled and data transfer between the buffer registers begins. The read-write head is disabled when a complete block of information has been transferred.

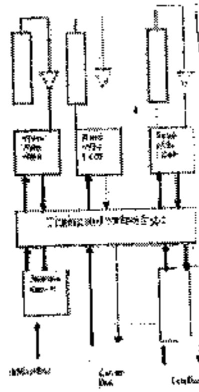


Figure 7.2 Organization of serial-access memory unit

7.2.4 Self Learning Exercise

True/False

A. A CPU should have rapid, uninterrupted access to the external memories where its program and the data they process are stored.

B. Storage locations in main memory are addressed directly by the CPU's load and store instructions.

Fill In the Blanks

C. ____________ memory stores system programs and large data files.

D. ____________ storage cells are physically arranged into regular arrays to reduce the cost of connection between cells and the access circuitry.

E. Serial access memories store information on _______ that behave like shift registers.

7.3 Memory Systems

This section examines the general characteristics of memory systems that have a multilevel, hierarchical organization. Two key design issues are considered in detail automatic translation of addresses and dynamic relocation of data.

7.3.1 Multilevel Memories

A computer's memory units form a hierarchy of different memory types in which each member is in some sense subordinate to the next-highest member of the hierarchy. The object of this organization is to achieve a good trade-off between cost, storage capacity and performance for the memory system as a whole. General characteristics Typical technologies used in these hierarchies are semiconductor SRAMs for cache memory, semiconductor DRAMs for main memory, and magnetic-disk units for secondary memory. I-cache has separate areas for storing instructions and D-cache stores the data. The following relations normally hold between adjacent memory levels M_i and M_{i+1} in a memory hierarchy:

Cost per bit	$C_i > C_{i+1}$
Access time	$t_A < t_A i+1$.
Storage capacity	$S_i < S_{j+1}$

The differences in cost, access time, and capacity between M_i and M_{i+1} can be several orders of magnitude. Considerable system resources are devoted to shielding the CPU from these differences, so it almost always seems a very large and inexpensive memory space and rarely seems an access time greater than that of M_1, the first (highest) level of the memory hierarchy.

The CPU and other processors can communicate directly with M_1 only, M_1 can communicate with M_2, and so on. Consequently, for the CPU to read information held in some memory level M_i requires a sequence of i data transfers of the form

$M_{i-1} := M_i; M_{i-2} := M_{i-1}; M_{i-3} := M_{i-2}; ... M_1 := M_2; CPU := M_1$.

An exception is allowed in the case of caches; the CPU is designed to bypass the cache level(s) and go directly to main memory, as we will see later. In general, all the information stored in M_i at any time is also stored in M_{i+1} but not vice versa.

During program execution the CPU produces a steady stream of memory addresses. At any time these addresses are distributed in some fashion throughout the memory hierarchy. If an address is generated that is currently assigned only to M_i where i'•1, the address must be reassigned to M_1, the level of the memory hierarchy that the CPU can access directly. This relocation of addresses involves the transfer of data between levels M_i and M_1-a relatively slow process. For a memory hierarchy to work efficiently, the addresses generated by the CPU should be found in M_1 as often as possible. This approach requires that future addresses be to some extent predictable so that information can be transferred to M_1 before it is actually referenced by the CPU. If the desired data cannot be found in M_1 then the program originating the memory request must be suspended until an appropriate reallocation of storage is made.

Cache and Virtual memory The various parts of a memory hierarchy are controlled in vary different fashions. Cache and main memory form a distinct sub hierarchy whose design objective is to support CPU access with a minimum of delay

Main and secondary memory form another distinct two level sub hierarchy. This interaction is managed by operating system, however so is not transparent to system software although it is somewhat transparent to user code. The term virtual memory is applied when the main and secondary memories appear to a user program like a single, large and directly addressable memory. Traditionally there are three reasons for using virtual memory.

- To free user programs from the need to carry out storage allocation and to permit efficient sharing of the available memory space among different users.
- To make programs independent of the configuration and capacity of the physical memory present for their execution; for example, to allow seamless overflow into secondary memory when the capacity of main memory is exceeded.
- To achieve the very low access time and cost per bit that are possible with a memory hierarchy.

7.3.2 Address Translation

The set of abstract locations that a program Q can reference is Q' s virtual address space V. Such addresses can be explicitly or implicitly named by identifiers that a programmer assigns to data variables, instruction labels, and so forth. The addresses can also be constructed or modified by the system software that controls Q. To execute Q on a particular computer, its virtual addresses must be mapped onto the real address space R, defined by the addressable (external) memory M that is physically present in the computer. This process is called address translation or address mapping.

Address translation can be viewed abstractly as a function f: V ->R.

A compiler transforms the symbolic identifiers of a program into binary addresses. If the program is sufficiently simple, the compiler can completely map virtual addresses to real addresses. Address translation can also be completed when the program is first loaded for execution. This process is called static translation, since the real address space of the program is fixed for the duration of its execution. It is often desirable to vary the virtual space of a program dynamically during execution; this process is dynamic translation. For example, a recursive procedure one that calls itself is typically controlled by a stack containing the linkage between successive calls. The size of this stack cannot be predicted in advance because it depends on the number of times the procedure is called; therefore, it is desirable to allocate stack addresses on the fly. Hardware-implemented memory management units (MMUs) have come into widespread use for run-time

address translation.

Base Addressing An executable program comprises a set of instruction and data blocks each of which is a sequence of words to be stored in consecutive memory locations during execution. A word W within a block has its own effective address A_{eff}, which the CPU must know to access W. (For the moment, we will ignore the distinction between the real and virtual address spaces.) W is also specified by the address B, called the base address, of the block that contains it, along with W's relative address or displacement D (also called an offset or index) within the block

$$A_{eff} = B + D$$

often the address is designed so that B supplies the high-order bits of A_{eff} while D applies the low-order bits thus:

$$A_{eff} = B.D$$

row A_{eff} is formed simply by concatenating B and D, a process that does not significantly increase the time for address generation.

A simple way to implement static and dynamic address mapping is to put base addresses in a memory map or memory address table controlled by the memory management system. The table can be stored in memory, in CPU registers, or in both. The address-generation logic of the CPU computes an effective address A_{eff} by combining the displacement D with the corresponding base address B_j

Blocks are easily relocated in memory by manipulating their base addresses. Suppose that two blocks are allocated to main memory M It is desired to load a third block K_3 into M; however, a contiguous empty space, or hole of sufficient size is unavailable. A solution to this problem is to move block K_2 by assigning it a new base address B_2 and reloading it into memory. This creates a gap into which block K_3 can be loaded by assigning to it appropriate base address.

With dynamic memory allocation, we must control the references made by a block to locations outside the memory area currently assigned to it. The block can be permitted to read from certain locations, but writing outside its assigned area must be prevented. A common way of doing this is by specifying the highest address L_i, called the limit address, that the block can access. Equivalently, the size of the block may be specified. The base address B_i and the limit address L_i are stored in the memory map. Every real address A_r generated by the block is compared to B_i and L_i; the memory access is completed if and only if the condition following is satisfied.

$$B_i d" A_r d" L_i$$

Translation look-aside buffer The input address A_v is a virtual address consisting of a (virtual) base address B_v concatenated with a displacement D. A_v contains an effective address computed in accordance with some program-defined addressing mode (direct, indirect, indexed, and so on) for the memory item being accessed. It also can contain specific control information-a segment address. The real address $B_R = f(B_v)$ assigned to B_v *is* stored in a memory map somewhere in the memory system; this map can be quite large. To speed up the Mapping process, part (or occasionally all) of the memory map is placed in a small high speed memory in the CPU called a *translation look-aside buffer (TLB)*. The TLB's input is thus the base-address part B_v of A_v; its output is the corresponding address B_R. This address is then concatenated with the D part of A_v to obtain the full physical address A_R.

If the virtual address B_v is not currently assigned to the TLB, then the part of the memory map that contains B_v is first transferred from the external memory into the TLB. Hence the TLB itself forms a cache like level within a multilevel address storage system for memory maps. For this reason, the TLB is sometimes referred to as an address cache.

Segments The basic unit of information for swapping purposes in a multilevel memory is a fixed-size block called a page. Pages are allocated to page-sized storage regions (page frames), whose fixed size and address formats make paging System easy to implement. Pages are convenient blocks for the physical partitioning swapping of the information stored in a multilevel memory. It is often desirable to have higher-

level information blocks, termed segments, that correspond to logical entities such as programs or data sets. Segments facilitate the mapping of individual programs, as well as the assignment and checking of different storage properties. For example, write operations may not be permitted into certain region of the virtual address space in order to protect critical items. It is easier to protect information in question by making it a read-only segment S, rather than assigning access restrictions to the possibly large number of pages that compose S. Formally, a segment is a set of logically related, contiguous words. A word in a segment to by specifying a base address-the segment address and a displacement within the segment. A program and its data can be viewed as a collection of linked segments. The links arise from the fact that a program segment uses, or calls, other segments. Some computers have a memory management technique that allocates main memory by M_1 segments alone. When a segment not currently resident in M_1 is required, the entire segment is transferred from secondary memory M_2. The Physical addresses assigned to the segments are kept in a memory map called segment table (which can itself be a relocatable segment).

Pages A page is a fixed-length block that can be assigned to fixed regions of physical memory called page frames, The chief advantage of paging is that data transfer between memory levels is simplified: an incoming page can be assigned to , available page frame. In a pure paging system, each virtual address consists of two parts: a page address and a displacement. The memory map, now referred to as page table, Each (virtual) page address has a corresponding (real) address of a page frame in main or secondary memory. When the presence bit P = 1, the page in question is present in memory, and the page table contains the base address of the page frame to which the page has been assigned. If P = 0, a page fault occurs and a page swap ensues. The change bit C indicates whether or not the page has been changed since it was last loaded into main memory. If a change has occurred (C = 1), the page must be copied onto secondary memory when it is preempted. The page table can also contain memory protection data that specifies the access rights of the current program to read from, write into, or execute the page in question. Page tables differ from segment tables primarily in the fact that they contain no block size information.

7.3.3 Memory Allocation

Various levels of a memory system are divided into sets of contiguous locations, variously called regions, segments, or pages, which store blocks of data. Blocks are swapped automatically among the levels in order to minimize the access time seen by the processor. Swapping generally occurs in response to processor requests (demand swapping), However, to avoid making a processor wait while a requested item is being moved to the fastest level of memory M_1, some kind of anticipatory swapping must be implemented, which implies transferring blocks to M_1 in anticipation that they will be required soon, Good short-range prediction of access-request patterns is possible because of locality of reference.

The placement of blocks of information in a memory system is called memory allocation. The method of selecting the part of M, in which an incoming block K is to be placed is the replacement policy, Simple replacement policies assign K to M_1 only when an unoccupied or inactive region of sufficient size is available. More aggressive policies preempt occupied blocks to make room for K. In general, successful memory allocation methods result in a high hit ratio and a low average access time. If the hit ratio is low, an excessive amount of swapping between memory levels occurs, a phenomenon known as thrashing, Good memory allocation also minimizes the amount of unused or underused space in M_1

Nonpreemptive allocation Suppose a block K_j of n_j words is to be transferred from M_2 to M_1. If none of the blocks already occupying M_1 can be preempted (overwritten or moved) by K_j, then it is necessary to find or create an "available" religion of n_j, or more words to accommodate K_j. This process is termed as preemptive allocation.

Two widely used algorithms for nonpreemptive allocation of variable-sized blocks-unpaged segments, for example-are first fit and best fit. The first-fit method scans the memory map sequentially until an available region R_j of n_j or more words is found, where n_j is the size of the incoming block K_j. It then allocates K_j to R_j. The best-fit approach requires searching the memory map completely and assigning K_j to a region of n_j e•n_i words such that n_j- n_i is minimized.

Preemptive allocation Nonpreemptive allocation cannot make efficient use of memory in all situations.

Memory overflow, that is, rejection of a memory allocation request due to insufficient space, can be expected to occur with M_1 only partially full. Much more efficient use of the available memory space is possible if the occupied space can be reallocated to make room for incoming blocks. Reallocation may be done in two ways:

- The blocks already in M_1 can be relocated within M_1 to create a gap large enough for the incoming block.
- One or more occupied regions can be made available by deallocating the blocks they contain. This method requires a rule-a replacement policy-for selecting blocks to be deallocated and replaced.

7.3.4 Self Learning Exercise

True/ False

F. Dynamic allocation means determining the regions of memory assigned to program before the execution.

G. Swapping generally occurs in response to processor requests is known as demand swapping.

Fill In the Blanks

H. A _______ is a fixed-length block that can be assigned to fixed regions of physical memory called page frames.

I. The memory units of a computer are organized as __________ hierarchy.

J. A ___________ transforms the symbolic identifiers of a program into binary addresses.

7.4 Computer memory

7.4.1 Internal memory

7.4.1.1 Computer memory System Overview

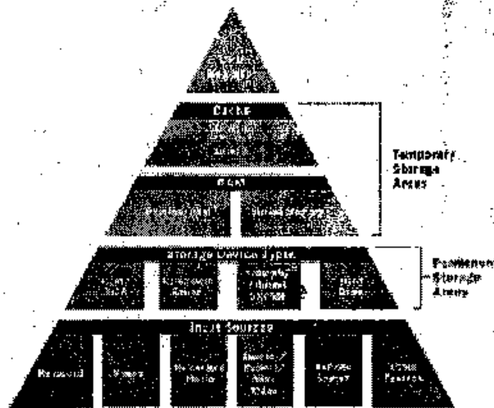


Figure 7.3 Computer Memory System Overview

2.2.2.2 Semiconductor Main Memory

Semiconductor random access memory, or RAM, as it is often referred to, is used in all types of computers. RAM is also called a read/write memory or a scratch-pad memory. Semiconductor RAM refers to semiconductor IC memories that can be used in a read mode as well as a write mode. Semiconductor memories use either a read cycle or a write cycle depending on the type of request, independent of each other. The read cycle is normally a shorter time period than the write cycle.

Semiconductor memories are normally nondestructive readout and volatile memories. In a nondestructive readout memory, the data stored in memory is not destroyed by the procedure used to read the data from the memory cells. Volatile memories require electrical power to maintain storage. If the power goes away for some reason, the data stored in the memory cells is lost. For this reason, an uninterruptible power supply (UPS) and a battery backup system are used in many semiconductor memory applications to maintain constant power and prevent loss of information because of power fluctuations or failures. This is especially important in microcomputers where configuration data is maintained in special devices such as a complementary metaloxide semiconductor (CMOS). The

battery backup and a filter capacitor provide the required power when the microcomputer has been powered down. Computers that use an UPS system have an established time in which data will be retained for momentary power losses. The term random access memory (RAM) is consistently used for read/write devices. Although RAM only describes one characteristic of read/write devices, it is used and understood by most people to mean read/ write devices. RAM means random addresses can be presented to the memory which means data can be written and read in any desired order from any location.
Note: The term RAM is not used for read-only memories (ROM), although a ROM can also be random access. Semiconductor RAM itself is made up of variable numbers of these RAM chips. Each chip contains large numbers of memory cells and the logic to support them. Each memory cell is an electronic circuit with at least two stable states. With the advent of large and very large scale integration (LSI/VLSI), literally thousands or hundreds of thousands of memory cell circuits can be placed on a single chip.
Each of the two-state memory cell circuits is capable of storing a single binary digit orbit (0 or 1).
These chips are mounted on logic boards or circuit card assemblies in some sort of memory array, also called gate arrays, based on the memory capabilities required or desired by the equipment designer.
The capabilities of individual chips determine the array organization for the memory capabilities desired.
On RAM chips, memory cells are organized based on two factors, the number of memory words or addresses and the number of bits per word. Most memory logic chips are rated by these values. For instance, a 4K by 16 chip would provide 4,096 16-bit memory addresses. This 4K by 16 chip will not support a 32-bit word for 4,096 addresses.

2.2.2.3 Cache Memory

The term cache refers to a fast intermediate memory within a larger memory system Caches directly address the von Neumann bottleneck by providing the CPU with fast, single-cycle access to ite external memory. They also provide an efficient way to place a small portion of memory on the same chip as a microprocessor
A cache serves as a buffer between a CPU and its main memory. The translation look-aside buffers (TLBs) used within memory management system are specialized caches that permit very fast translation of memory addresses. Data buffers built into high speed secondary memory device such as hard disk drives are also called caches.

2.2.2.3.1 Main Features

Cache Organization: Memory words are stored in a cache data memory and are grouped into small pages called cache blocks or lines. The contents of the cache's data memory are thus copies of a set of main-memory blocks. Each cache block is marked with its block address, referred to as a tag, so the cache knows what part of memory space the block belongs. The collection of tag addresses currently assigned to the cache, which can be non-contiguous, is stored in a special memory, the cache tag memory or directory.

Two-level hierarchy (Mi-1, Mi)	Cache Main memory (M1,M2)	Main- secondary memory (M2,M3)
Typical access time ratios tA/tAi-1	5/1	1000/1
Memory Management System	Mainly implemented by hardware	Mainly implemented by software
Typical page size	8B	4KB
Access of processor to second level Mi	Processor has direct access to M2	All access to M3 via M2

Figure 7.4 Major Differences between cache and main-secondary memory hierarchies

Cache operation In 4 bytes long cache block each memory address is 12 bits long, so the 10 high-order bits from the tag or block address, and 2 low-order bits define a displacement address within the block. When a block is assigned to M_1's data memory, its tag is also placed in M_1's tag memory. To read the word, its address is sent to M_1, which compares A_i's tag part to its stored tags and finds a match (hit). The stored tag pinpoints the corresponding block in M_1's data memory, and the 2-bit displacement is used to output the target word to the CPU.

A Cache write operation employs the same addressing technique. The tag part of the target address M_i is again presented to M_1, along with the data word to be stored. When a hit occurs, the new data is stored at the location pointed to by A_i in the data memory of M_1. Now a new problem arises: The data in M_1 with address A_i differs from the data in M_2 with the same address. A temporary inconsistency of this sort is acceptable as long as no device-another processor, for instance-attempts to read the old or stale data. Preventing the improper use of stale data is the cache coherence or cache consistency problem.

When a block with C = 1 is replaced, its data contents are then written back to main memory M_2 This technique is referred to as write-back or copy-back.

2.2.2.3.2 Address Mapping

When a tag address is presented to the cache, it must be quickly compared to the stored tags to determine whether a matching tag is currently assigned to the cache. The obvious approach of scanning all the tags in sequence is unacceptably slow. The fastest technique for implementing tag comparison is *associative* or *content addressing* which permits the input tag to be compared simultaneously to all tags in cache tag memory.

Associative addressing In an associative memory any stored item can be accessed by using the contents of the item in question, generally some specified subfield as an address. Associative memories are also commonly known as content-addressable-memories (CAMs). The subfield chosen to address the memory is called the key. Items stored in an associative memory can viewed as having the two- field format

KEY, DATA

Where the key is the stored address and DATA is the information to be accessed.

An associative cache employs a tag, that is, a block address, as the key. At the start of memory access, the incoming tag is compared simultaneously to all the tags stored in the cache's tag memory. If a match (cache hit) occurs, a match-indicating signal triggers the cache to service the requested memory access. A no-match signal identifies a cache miss, and the memory access requested is forwarded to main memory for service. A cache block containing the target address is then sent from main memory to the cache, and at the same time, a data word is sent to the CPU or transferred from the CPU to the cache, in response to the original access request.

Associative memory Figure 7.5 shows the general structure of an associative memory. Each unit of stored information is a fixed-length word. Any subfield of the word can be chosen as the key. The current key is compared simultaneously with all stored words those that match the key output a match signal, which enters a select circuit, which enables the data field to be accessed. If several entries have the same key, then the select circuit determines which data field is to be read out.

2.2.2.3.3 Structure versus Performance

Cache types Caches are distinguished by the kinds of information they store. An instruction or I-cache stores instructions only, while a data or D-cache stores data only. Separating the stored data in this way recognizes the different access behavior patterns of instructions and data. Caches are also classified by the level they occupy in the memory hierarchy. A level 1 (L1) or primary cache is an efficient way to implement an on-chip memory. additional memory level can be introduced via an off-chip, level 2 (L2) or secondary cache.

Performance The cache is the fastest component in the memory hierarchy so it is desirable to make the average memory access time t_A seen by the CPU as close as possible to access time t_{A1} of the cache. To achieve this goal, M_1 should satisfy a very high percentage of all memory references; that is, the cache hit ratio H should be almost one.

Design process The parameters like the block replacement and write policies, influence the cache's hit ratio H in ways that are hard to quantify because they depend on the workloads used with the cache. Such workloads, in turn, are application dependent. As a result, potential cache designs are evaluated by extensive trace-driven simulation experiments with address traces derived from representative programs or benchmarks for the target applications. Experiments involving billions of simulated address references are often carried out in the design of the caches for a new microprocessor.

7.1.4 Advanced DRAM Organization

One of the most critical system bottlenecks when using high-performance processors is the interface to main internal memory. This interface is the most important pathway in the entire computer system. The basic building block of main memory remains the DRAM chip, as it has for decades; until recently, there had been no significant changes in DRAM architecture since the early 1970s. The traditional DRAM chip is constrained both by its internal architecture and by its interface to the processor's memory bus.

One attack on the performance problem of DRAM main memory has been to insert one or more levels of high-speed SRAM cache between the DRAM main memory and the processor. But SRAM is much costlier than DRAM, and expanding cache size beyond a certain point yields diminishing returns.

In recent years, a number of enhancements to the basic DRAM architecture have been explored, and some of these are now on the market. The two schemes that currently dominate the market are SDRAM and RDRAM. CDRAM has also received considerable attention. We examine each of these approaches in this section.

Synchronous DRAM :One of the most widely used forms of DRAM is the synchronous DRAM (SDRAM). Unlike the traditional DRAM, which is asynchronous, the SDRAM exchanges data with the processor synchronized to an external clock signal and running at the full speed of the processor/memory bus without imposing wait states.

In a typical DRAM, the processor presents addresses and control levels to the memory, indicating that a set of data at a particular location in memory should be either read from or written into the DRAM. After a delay, the access time, the DRAM either writes or reads the data. During the access-time delay, the DRAM performs various internal functions, such as activating the high capacitance of the row and column lines, sensing the data, and routing the data out through the output buffers. The processor must simply wait through this delay, slowing system performance.

With synchronous access, the DRAM moves data in and out under control of the system clock. The processor or other master issues the instruction and address information, which is latched by the DRAM. The DRAM then responds after a set number of clock cycles. Meanwhile, the master can safely do other tasks while the SDRAM is processing the request.

The SDRAM employs a burst mode to eliminate .the address setup time and row and column line precharge time after the first access. In burst mode, a series of data bits can be clocked out rapidly after the first bit has been accessed. This mode is useful when all the bits to be accessed are in sequence and in the same row of the array as the initial access. In addition, the SDRAM has a multiple-bank internal architecture that improves opportunities for on-chip parallelism.

The mode register and associated control logic is another key feature differentiating SDRAMs from conventional DRAMs. It provides a mechanism to customize the SDRAM to suit specific system needs. The mode register specifies the burst length, which is the number of separate units of data synchronously fed onto the bus. The register also allows the programmer to adjust the latency between receipt of a read request and the beginning of data transfer.

The SDRAM performs best when it is transferring large blocks of data serially, such as for applications like word processing, spreadsheets, and multimedia.

Rambus DRAM : RDRAM, developed by Rambus [FARM92, CRIS97], has been adopted by Intel for its Pentium and Itanium processors. It has become the main competitor to SDRAM. RDRAM chips are vertical packages, with all pins on one side. The chip exchanges data with the processor over 28 wires no more than 12 centimeters long. The bus can address up to 320 RDRAM chips and is rated at 1.6 Gbps. The special RDRAM bus delivers address and control information using an asynchronous block-oriented

protocol. After an initial 480 ns access time, this produces the 1.6 GBps data rate. What makes this speed possible is the bus itself, which defines impedances, clocking, and signals very precisely. An RDRAM gets a memory request over the high-speed bus. This request contains the desired address, the type of operation, and the number of bytes in the operation.

The configuration of SDRAM consists of a controller and a number of RDRAM modules connected together via a common bus. The controller is at one end of the configuration, and the far end of the bus is a parallel termination of the bus lines. The bus includes 18 data lines (16 actual data, two parity) cycling at twice the clock rate; that is, one bit is sent at the leading and following edge of each clock signal. This results in a signal rate on each data line of 800 Mbps. There is a separate set of 8 lines (RC) used for address and control signals. There is also a clock signal that starts at the far end from the controller propagates to the controller end and then loops back. A RDRAM module ends data to the controller synchronously to the clock to master, and the controller sends data to an RDRAM synchronously with the clock signal in the opposite direction. The remaining bus lines include a reference voltage, ground, and power source.

Cache DRAM : Cache DRAM (CDRAM), developed by Mitsubishi, integrates a small SRAM cache (16Kb) onto a generic DRAM chip. The SRAM on the CDRAM can be used in two ways. First, it can be used as a true cache, consisting of a number of 64-bit lines. The cache mode of the CDRAM is effective for ordinary random access to memory. The SRAM on the CDRAM can also be used as a buffer to support the serial access of a block of data. For example, to refresh a bit-mapped screen, the CDRAM can prefetch the data from the DRAM into the SRAM buffer. Subsequent accesses to the chip result in accesses solely to the SRAM.

7.4.2 External Memory

7.4.2.1 Magnetic disks

The magnetic disks are the foundation of external memory on virtually all computer system. Both removable and fixed disk or hard disk are used in computer system from personal computer to mainframe or supercomputer.

Principle: The disk is a metal or plastic platter coated with magnetizable material. Data is recorded onto and later read from the disk using a conducting coil. Data is organized into concentric rings, called tracks, on the platter. Tracks are separated by gap. Disk rotates at a constant speed. The number of data bits per track is a constant. The data density is higher on the inner tracks. Logical data transfer unit is the sector. Sectors are identified on each track during the formatting process.

Disk characteristics: Each platter has its own read/write head. Fixed head has a head per track. Movable head uses one head per platter. Removable platter can be removed from disk drive for storage of transfer to another machine

Data accessing times:

Seek time : position the head over the correct track

Rotational latency : wait for the desired sector to come under the head

Access time : seek time plus rotational latency

Block transfer time : time to read the block (sector) off of the disk and transfer it to main memory.

7.4.2.2 RAID Technology

The RAID (Redundant Array of Independent Disk) technology can obtain greater performance and higher availability. RAID refers to a family of techniques for using multiple disks as a parallel array of data storage devices with redundant built in to compensate for disk failure. Disk drive performance has not kept pace with improvements in other parts of the system It is limited in many cases by the electromechanical transport means. Its capacity is of a high performance disk drive can be duplicated by operating many (much cheaper) disks in parallel with simultaneous access. Data is distributed across all disks with many parallel disks operating as if they were a single unit, redundancy techniques can be used to guard against data loss in the unit (due to aggregate failure rate being higher)

7.4.2.3 Optical Memory

In Optical Memory, data is stored on an optical medium (i.e., CD-ROM or DVD), and read with a laser beam. While not currently practical for use in computer processing, optical memory is an ideal solution for storing large quantities of data very inexpensively, and more importantly, transporting that data between computer devices.

7.4.2.4 Magnetic Tape

This is the first kind of secondary memory. And still widely used. It is very cheap and very slow and has sequential access. In this data is organized as records with physical air gaps between records. one word is stored across the width of the tape and read using multiple read/write heads.

7.4.3.Self Learning Exercise

True/ False

K. The I-cache stores instructions only

L. To reduce the speed disparity between CPU and main memory, one or more intermediate memories called caches are used.

Fill in the Blanks

M. The optical memory read with ________ beam.

N. The RAID stands for __________________.

O. The _________ memory is much faster than main memory.

P. In an ___________ memory any stored item can be accessed by using the contents of the item in question

7.5 Summary

In this unit we have discussed that main memory is of the random-access type where the access time of every location is constant. DRAMs are based on single transistor cells. Secondary memories require a lower cost per bit and a higher storage density. We can achieve these goals by using serial access memory technologies that share access mechanism and have access times that vary with location. To reduce the speed disparity between CPU and main memory, one or more intermediate memories called caches are used.

7.6 Glossary

Address Translation/ Address mapping	It is the process by which the virtual address maps on to real address, defined by addressable memory that is physically present on the computer.
Cache memory	It is a fast intermediate memory which serves as buffer between a CPU and its main memory.
DRAM	Dynamic Random-Access Memory, is the medium that is used for the temporary storage of information by today's personal computers and mainframes.
Magnetic Disk	A memory device, such as a floppy disk, a hard disk, or a removable cartridge, that is covered with a magnetic coating on which digital information is stored in the form of microscopically small, magnetized needles.
Magnetic tape	It is a medium for magnetic recording generally consisting of a thin magnetizable coating on a long and narrow strip of plastic.
Memory allocation	The placement of blocks of information in memory system is called memory allocation.
Optical memory	In Optical Memory, data is stored on an optical medium (i.e., CD-ROM or DVD), and read with a laser beam.
RAM	A memory that stores binary information during the operation of computer. This memory is used as a writing pad to write user programs and data. The information stored in this memory can be read and altered easily.
Serial Access Memories	These are the memories which store the information on tracks that behave like shift registers.

7.7 Further Readings

- J.P.Hayes: Computer Architecture and Organization, McGraw-Hill International
- R.S.Goankar: Microprocessor Architecture, Programming and Applications with the 8085/8080, 2nd Edition, New Age International Publishers Limited, ISBN-81-224-0710-2.
- K.L.Short: Microprocessors and Programmed Logic, 2nd Edition, Prentice Hall of India Pvt. Ltd. 1988, ISBN-0-07-100462-9.

7.8 Answer to self learning Exercises

Question No.	Answer	Question No.	Answer
A	True	I	Multilevel
B	True	J	Compiler
C	Secondary	K	True
D	RAM	L	True
E	Tracks	M	Laser
F	False	N	Redundant Array of Independent Disk
G	True	O	Cache
H	Page	P	Associative

7.9Unit End Questions:

1. What should be the characteristics of memory devices?
2. Write short note on
   a. Magnetic tape memories
   b. Optical memories
   c. Magnetic disk memories
   d. Cache memories
3. What do you understand by address translation? Explain the structure of dynamic address translation system.
4. State the difference between Non pre-emptive and pre-emptive memory allocation?
5. What is cache? Explain its features.

Unit-08
System Organization

Structure of the Unit

8.0 Objectives
8.1 Introduction
8.2 System Organization
8.3 IO and System Control
8.3.1 Programmed IO
8.3.2 DMA and Interrupts
8.3.3 IO Processors
8.4 Parallel Processing
8.4.1 Processor-Level-Parallelism
8.4.2 Multiprocessors
8.5 Pipeline Control
8.5.1 Instructions Pipelines
8.5.2 Arithmetic Pipelines
8.5.3 Pipeline Performance
8.5.4 Super-Scalar Processing
8.6 Summary
8.7 Glossary
8.8 Further Readings
8.9 Answer to Self Learning Exercise
8.10 Unit End Questions

8.0 Objectives

This unit explains about the interconnection of computer and its major components. It explains their management at processor level or system level. It discusses the use of multiprocessor to achieve high performance. Pipeline control and super scalar processing are also discussed in detail.

8.1 Introduction

In the previous unit we have studied the memory organization, its technology and the characteristics of memory devices. We have an overview of different types of computer memories i.e. Internal and external memory. We have also discussed address mapping and advanced DRAM organization.

8.3 IO and System Control

The main data-processing functions of a computer involve its CPU and external (cache-main) memory M. The CPU fetches instructions and data from M, processes them, and eventually stores the results back in M. The other system components-secondary memory, user interface devices, and so on—constitute the input-output (IO) system.

IO control methods

Input-output operations are distinguished by the extent to which the CPU is involved in their execution. (Unless otherwise stated, IO *operation* refers to a data transfer between an IO device and M, or between an IO device and the CPU). If such operations are completely controlled by the CPU, that is, the CPU executes programs that initiate, direct and terminate the IO Operations, the computer is said to be using *programmed IO. This type of IO control can be* implemented with little or no special hardware, but causes the CPU to spend a lot of time performing relatively trivial IO-related functions. One such function is testing the status of IO devices to determine if they require servicing by the CPU.

A modest increase in hardware enables an IO device to transfer a block of information to or from M without CPU intervention. This task requires the IO device to generate memory addresses and transfer data to or from the bus (system or local) connecting it to M via its interface controller.

The DMA controller can also be provided with circuits enabling it to request service from the CPU, that is

execution of a specific program to service an IO device. This type of request called an interrupted, and it frees the CPU from the task of periodically testing the status of IO device. Unlike a DMA request, which merely requests temporary access to the system bus, an interrupt request causes the CPU to switch programs by saving its previous program state and transferring control to a new interrupt-handling program. After the interrupt has been serviced, the CPU can resume execution of the interrupted program.

DMA controller has partial control of IO operations. Essentially complete control of IO operations can be relinquished by the CPU if an IO *processor* (IOP) is introduced. Like a DMA controller, an IOP has direct access to main memory and can interrupt the CPU; however, an IOP can also execute programs directly. These programs, called IO *programs,* may employ an instruction set different from the CPU's *one* that is oriented toward IO operations. It is common for larger systems to use general-purpose microprocessors as IOPs. An IOP can perform several independent data transfers between main memory and one or more IO devices recourse to the CPU. Usually the IOP is connected to the devices it controls by a separate bus system, the IO bus.

8.3.1 Programmed IO

A method included in every computer for controlling IO operations. It is most useful in small, low-speed systems where hardware cost must be minimized. Programmed IO requires that all IO operations be executed under the direct control of the CPU; in other words, every data-transfer operation involving an IO device requires the execution of an instruction by the CPU, Typically the transfer is between two programmable registers: one a CPU register and the other attached to the IO device. The IO device does not have direct access to main memory M. A data transfer from the IO device to M requires the CPU to execute several instructions, including an input instruction to transfer a word from the IO device to the CPU and a store instruction to transfer the word from CPU to M. One or two additional instructions may be needed for address computation and data-word counting.

IO addressing

In systems employing programmed IO, the CPU, M, and IO usually communicate via the system bus. The address lines of the system bus that are used to select memory locations can also be used to select IO devices. An IO device is connected to the bus via an IO port, which, from the CPU's perspective, is an addressable data register, thus making it little different from a main memory location.

This technique is called memory mapped IO. A memory referencing instruction that causes data to be fetched from or stored at address X automatically becomes an IO instruction, if X is made the address of an IO port. The usual memory load and store instructions are used to transfer data words to or from IO ports , no special IO instructions are needed.

The control lines *READ* and *WRITE,* which are activated by the CPU when processing a memory reference instruction, are used to initiate either a memory access cycle or an IO transfer .

In the organization of IO-*mapped IO,* the memory and IO address spaces are separate. This scheme is used, for example, in the Intel 80X86 microprocessor series. A memory-referencing instruction activates the READ M or WRITE M control line which does not affect the IO devices. The CPU must execute separate IO instructions to activate the *READ IO* and *WRITE* IO lines, which cause a Word to be transferred between the addressed IO port and the CPU. An IO device and a memory location can have the same address bit pattern without conflict.

IO instructions

As few as two instructions can implement programmed IO. For example, members of intel 80X86 series have two IO instructions called IN and OUT. The instruction IN X causes a word to be transferred from IO port X to the 80X86's accumulator register A. The instruction OUT X transfers a word from the A register to IO port X.

When the CPU executes an IO instruction such as IN or OUT, the addressed IO port is expected to be ready to respond to the instruction. Therefore, the IO device must transfer data to or from the CPU-IO data bus within a specified period. To prevent loss of information or an indefinitely long IO instruction execution time, the CPU must know the IO device status so that the transfer is carried out only when the

device is ready. With the programmed IO the CPU can be programmed to test the IO device's status before initiating an IO data transfer. Often the status is specified by a single bit of information that the IO device make available on a continuous basis.

The CPU must perform the following steps to determine the status of an IO device:

1. Read the IO device's status bit.
2. Test the status bit to determine if the device is ready to begin transferring data.
3. If not ready, return to step 1; otherwise, proceed with the data transfer.

If programmed IO is the primary method of input-output control in a computer, additional IO instructions can be provided to augment the IN and OUT instructions.

A common IO programming task is the transfer of a block of words between an IO device and a contiguous region of memory.

IO interface circuits

The task of connecting an IO device to a computer system is greatly eased by the use of standard ICs variously known as IO interface circuits, peripheral interface adapters, and the like. These circuits allow the IO devices of widely different characteristics to be connected to a standard bus with a minimum special purpose hardware or software. The simplest interface circuit is a one-word, addressable register that serves as an IO port. The major microprocessor families contain various general-purpose and special-purpose IO interface circuits. They are called programmable if they can be modified under program control to match the characteristics of different IO devices.

Among the most basic IO interface circuits are programmable circuits intended to act as serial or parallel ports. Serial ports accommodate many types of slow peripheral devices ranging from secondary memory units to network connections. Parallel ports are designed to interface with IO devices employing multibit, bi-directional data paths.

3.3.2 DMA and Interrupts:

The programmed IO method section has two limitations:

The speed with which the CPU can test and service IO devices limits IO data transfer rates.

The time that the CPU spends testing IO device status and executing IO data transfers can often be better spent on other tasks.

The influence of the CPU on IO transfer rates is twofold.

First, a delay occurs while an IO device needing service waits to be tested by the CPU. If there are many IO devices in the system, each device may be tested infrequently.

Second, programmed IO transmits data through the CPU rather than allowing it to be passed directly from main memory to the IO device, and vice-versa.

DMA and Interrupt circuits increase the speed of IO operations by eliminating most of the role played by the CPU in such operations. In each case special control lines, to which we assign the generic names DMA REQUEST and INTERRUPT REQUEST, connect the IO devices to the CPU. Signals on these lines cause the CPU to suspend its current activities at appropriate breakpoints and attend to the DMA or interrupt request. Thus these special request lines eliminate the need for the CPU to execute routines that determine IO device status. DMA further allows IO data transfer to take place without the execution of IO instruction by the CPU.

A DMA request by an IO device only requires the CPU to grant control of the memory (system) bus to the requesting device. The CPU can yield control at the end of any transactions involving the use of this bus. The instruction cycle is composed of a number of CPU cycles, several of which require use of the system bus. A common technique is to allow the machine to respond to a DMA request at the end of any CPU clock cycle. Thus during the instruction cycle there are five points in time (breakpoints) when the CPU can respond to a DMA request. When such a request is received by the CPU, it waits until the next breakpoint, releases the system bus, and signals the requesting IO device by activating a DMA ACKNOWLEDGE control line.

Interrupts are requested and acknowledged in much the same way as DMA requests. However, an interrupt

is not a request for bus control; rather, it asks the CPU to begin executing an interrupt service program. The interrupt program performs tasks such as initiating an IO operation or responding to an error encountered by the IO device. The CPU transfers control to this program in essentially the same way it transfers control to a subroutine. The CPU responds to interrupts only between instruction cycles.

Direct Memory Access

The IO device IO connected to the system bus via a special interface circuit, a DMA controller, which contains a data buffer register IODR, as in programmed IO case; it also controls an address register IOAR and a data count register DC. These registers enable the DMA controller to transfer data to or from a contiguous region of memory. IOAR stores the address of the next word to be transferred. It is automatically incremented or decremented after each word transfer. The data counter DC stores the number of words that remain to be transferred. It is automatically decremented after each transfer and tested for zero. When the data count reaches zero the DMA transfer halts. The DMA controller is normally provided with an interrupt to the CPU to signal the end of the IO data transfer. The logic necessary to control DMA can easily be placed in single IC with other IO control circuits. A DMA controller can be designed to supervise DMA transfers involving several IO devices, each with a different priority of access to the system bus. Data can be transferred in several different ways under DMA control. In a DMA block transfer a data-word sequence of arbitrary length is transferred in a single burst while the DMA controller is master of the memory bus. This DMA mode is needed by the disk drives, where data transmission cannot be stopped or slowed without loss of data, and block transfers are the norm. Block DMA transfer supports the fastest IO data-transfer rates, but it can make the CPU inactive for relatively long periods by tying up the system bus. An alternative technique called cycle stealing allows the DMA controller to use the system bus to transfer one data word, after which it must return control of the bus to the CPU. Consequently, long blocks of IO data are transferred by a sequence of DMA bus transaction interspersed with CPU bus transactions. Cycle stealing reduces the maximum IO transfer rate, but it also reduces the interference by the DMA controller in the CPU's memory access. It is possible to eliminate this interference completely by designing the DMA interface so that bus cycle are stolen only when the CPU is not actually using the system bus, this is transparent DMA. Many different data-transfer characteristics of IO devices are as follows-

1. The CPU executes two IO instructions, which load the DMA register IOAR and DC with their initial values. IOAR should contain the base address of the memory region to be used in the data transfer. DC should contain the number of to be transferred to or from that region.
2. When the DMA controller is ready to transmit or receive data, it activates the REQUEST line to the CPU. The CPU waits for the next DMA breakpoint. It then relinquishes control of the data and address lines and activates DMA ACKNOWLEDGE. Note that DMA REQUEST and DMA ACKNOWLEDGE are essentially BUS REQUEST and BUS GRANT lines for control of the system bus. Simultaneous DMA requests from several DMA controllers are resolved by bus-priority control techniques.
3. The DMA controller now transfers data directly to or from main memory. After a word is transferred, IOAR and DC are updated.
4. If DC has not yet reached zero but the IO device is not ready to send or receive the next batch of data, the DMA controller releases the system bus to the CPU by deactivating the DMA REQUEST line. The CPU responds by deactivating the DMA ACKNOWLEDGE and resuming control of the system bus.
5. If DC is decremented to zero, the DMA controller again relinquishes control of system bus; it may also send an interrupt request signal to the CPU. The CPU responds by halting the IO device or by initiating a new DMA transfer.

By reducing the CPU's need to access main memory, a cache can greatly reduce conflicts between CPU and IO data transfers. High performance microprocessors often have separate cache-CPU and IO main-memory access paths, which means that a DMA transfer involving main memory can proceed in parallel with CPU cache operations. DMA operations use the PCI local bus, while the CPU communicates with the cache via the system bus. Only when the CPU needs access to main memory in response to a cache miss, for example -does it come into conflict with DMA controllers; such conflicts are resolved by the PCI bridge unit.

Interrupts

The word interrupt is used in a broad sense for any infrequent or exceptional event that causes a CPU to temporarily transfer control from its current program to another program-an interrupt handler-that services the event in question. Interrupts are the primary means by which IO devices obtain the services of the CPU. They significantly improve a computer's IO performance by giving IO devices direct and rapid access to the CPU and by freeing the CPU from the need to check the status of its IO devices.

The basic method of interrupting the CPU is by activating a control line with the generic name INTERRUPT REQUEST that connects the interrupt source to the CPU. An Interrupt indicator is then stored in a CPU register that the CPU tests periodically, usually at the end of every instruction cycle.

Interrupt selection

The problem of selecting one IO device to service from several that have generated interrupts strongly resembles the arbitration process for bus control. Some interrupt methods require that interrupting device be given control of the system bus. The techniques employed for bus arbitration-daisy chaining, polling, and independent requesting can all be readily adapted to interrupt handling and can be realized by software, hardware, or a combination of both.

The interrupt selection method requiring the least hardware is the single-line method. All IO ports share a single INTERRUPT REQUEST line. On responding to a single interrupt request, the CPU must scan all the IO devices to determine the source of the interrupt. This procedure requires activating an INTERRUPT ACKNOWLEDGE line (corresponding to BUS GRANT) that is connected in daisy-chain fashion to all IO devices. The connection sequence of this line determines the interrupt priority of each device.

Vectored Interrupts

The most flexible response to interrupts is obtained when an interrupt request from a particular device causes a direct, hardware-implemented transition to the correct interrupt-handling program. The interrupting device must then supply the CPU with the starting address or interrupt vector of that program.

Each interrupt request line generates a unique fixed address, which is used to modify the CPU's program counter PC. Interrupt requests are stored on receipt in an interrupt register. The interrupt mask register can disable any or all of the interrupt request lines under program control. By setting bit i of the register to 1(0), interrupt request line i is disabled (enabled). The k masked interrupt signals are fed into a priority encoder that produces a %$\log_2 k$% -bit address, which is then transferred into PC.

PCI interrupts

The PCI local bus provides general support for interrupt handling details such as the vectoring method used are architecture specific and depend on the particular devices using bus. The PCI bus has four interrupt request lines named INTA:D among its optional lines. A single-function IO device with interrupt capability must use INTA , as its interrupt request line; multifunction IO devices can use all four lines. A particular pattern on the PCI bus's command lines denotes interrupt acknowledge. Together, the INTx interrupt request lines and the interrupt acknowledge command can implement the request-acknowledge signal exchange needed during an i interrupt transaction over the PCI bus.

Every PCI-compatible device must have a standard set of addressable configuration registers CR that identify the device and its communication needs. When the system is powered up, the system controller (operating system) reads the CR registers to determine, among other things, the device's interrupt connections. Its 8-bit "interrupt pin" register in CR tells the system controller which interrupt request line INTx the IO device is using. A second 8-bit register in CR called the "interrupt line" register specifies the system controller's input line that is connected to INTx so that the routing of the interrupt request lines is programmable. The system controller can use this fact to determine the IO device's interrupt request priority and to access its interrupt vectors. The CR registers form a small address space that is separate from the main-memory and IO address spaces, as indicated by the existence of configuration read and configuration write in the command set specified for the PCI bus.

Pipeline interrupt

After an interrupt occurs, the controlling CPU must be able to identify the interrupting instruction and the

register contents needed for any corrective actions. This is not a problem when instructions are executed in sequence and only one is active at any time. However in a pipelined processor with several instructions in process concurrently, it is possible for instructions to finish out of sequence; that is, an instruction can finish sooner than another instruction that was issued earlier. where three floating-point instructions, a 7-cycle multiply and two 4-cycle adds, are being processed by one or more pipelined units. Assuming no hazards occur due to data dependencies, this completion order is acceptable as far as the main computation is concerned. We can solve the imprecise-interrupt problem in several ways. The most direct is to make all interrupts precise by forcing all instructions to complete in the order in which they are issued.
The undesirable result of this forced, in-order execution method is that the combined processing time for the three instructions increases from seven to nine cycles, so some of the performance benefit of pipelining is lost.

8.3.3. IO Processors

The IO processor (IOP) is a logical extension of the IO control methods considered so far. In the systems with programmed IO, peripheral devices are controlled directly by the CPU. The DMA concept extends limited control over data transfer to IO devices. An IOP has the ability to execute instructions, which gives it fairly complete control over IO operations. Like a CPU, an IOP is an instruction set processor, but it has a more restricted instruction set. IOPs are primarily communication control units designed to link IO devices to a computer. They have also been peripheral processing units (PPUs) to emphasize their subsidiary role with respect to the central processing unit(CPU).

IO instruction types

In a computer with an IOP, the CPU does not normally execute IO data-transfer instruction. Such instructions are contained in IO programs that are stored in M and are fetched and executed by the IOP. The CPU does execute a few IO instructions that allow it to initiate and terminate the execution of IO programs via the IOP and also to test the status of the IO system. The IO instructions executed by the IOP are primarily associated with data-transfer operations. A typical IO operation has the form: READ(WRITE) a block of n words from (to) device X to (from) memory region Y. the IOP is provided with direct access to M(DMA) and so can control the memory bus when the CPU does not require that bus. Other instructions types such as arithmetic, logical and branch are included in the IOP's instruction set to facilitate the calculation of addresses, IO device priorities and so on. The third category of IO instructions are those executed by IO devices. These instruction control functions such as REWIND, SEEK ADDRESS or PRINT PAGE. Instruction of this type are fetched by the IOP as data and passed on to the appropriate IO device for execution.
The instructions executed by the IOP are called channel command words (CCW). They are of three types
· Data-transfer instructions These include input (read), output (write), and sense (read status). They cause the number of bytes in the data count field to be transferred between the specified memory region and the previously selected IO device.
· Branch Instruction These cause the IOP to fetch the next CCW from the specified memory address rather than from the next sequential location.
· IO device control instruction These are transmitted to the IO device and specify functions peculiar to that device
The opcode of a data-transfer instruction can be transmitted directly to the IO device as the "command" byte while the IO operation is being set up. If the IO device requires more control information, it is supplied via an output data transfer.
The flag field of the CCW modifies the operation specified by the opcode. For example, a program control flag PCI can be set to instruct the IOP to generate an IO interrupt and make the current IOP status available to the CPU. Another flag specifies command chaining,, which means that the current CCW is followed by another CCW that is to be executed immediately. If this flag is not set the IOP ceases IO program execution after executing the current CCW.
IOP organization The IOP and CPU share access to a common memory M via the system bus. M stores separate programs for execution by the CPU and the IOP; it also contains a communication region IOCR

for passing information in the form of messages between the two processors. The CPU can place there the parameters of an IO task, for example, the addresses of the IO programs to be executed, and the identity of IO devices to be used. The CPU and IUOP also communicate with each other directly via control lines. Standard DMA or bus grant/acknowledge lines are used for arbitration of the system bus between the two processors. The CPU can attract the IOP's attention, for instance, when executing an IO instruction like START IO, by activating the ATTENTION line. In response the IOP begins execution of an IOP program whose specification have been placed in the IOCR communication area.

8.3.4 Self Learning Exercises

True/False

A. IO devices have direct access to main memory.

Fill In the Blanks

B. The use of CPU programs to control all phases of an IO operations is called ___________.

C. The technique to assign a part of main memory address space to IO ports is called as _________.

33393 D. The ___________ is a logical extension of the IO control methods.

E. The instructions executed by the IOP are called ___________.

8.4 Parallel processing

Computer performance can be increased by executing many instructions simultaneously or in parallel. This section describes processor-level parallelism in computers, focusing on the use of multiple CPUs to achieve very high throughput and fault tolerance.

Processor-Level Parallelism

Performance has increased steadily thanks to faster hardware technologies or designs, many important computational problems remain beyond the capabilities of the fastest current machines. Some computer designers believe that processor and memory technologies are approaching physical limits on their size and speed. Size reductions and speed increases well beyond present levels are feasible, but their cost may not be acceptable. One way to address these issues is to exploit processor-level-parallelism, for example, by building computers containing large numbers- perhaps hundreds or thousands-of low-cost processors that can work in parallel on common tasks.

A further advantage of processor-level-parallelism is tolerance of hardware and software faults. While failure of its CPU is almost always fatal to a sequential computer, a parallel computer can be designed to continue functioning, perhaps at a reduced performance level, in the presence of defective CPU.

Dependencies The main benefit of parallel processing is faster computation. A price is paid, however, in the need for a significant amount of extra hardware. We can say that increasing the number of processors by a factor of n makes an n-fold increase in computing performance. This maximum speed up is rarely achieved because it is difficult to keep all the members of a set of parallel processors continually working at their maximum rates. Dependencies among sub tasks can force a processor to wait until other processors supply results that it needs.In the parallel summation algorithm for the linear processors array, the processors must wait for data from their left neighbors. The processors in a parallel computer often share resources such as memory banks, IO devices or operating system routines which can be used by only one processor at a time. A major issue in designing and programming parallel systems is to avoid conflicts in the use of shared resources. The extent to which all processors can kept busy depends on the computer architecture, the tasks being performed , and the way in which the tasks are programmed.

Classification methods

A processor such as CPU operates by fetching instructions and operand from memory M(main memory or cache), executing the instructions and placing the final results in M. The instructions from an instruction stream flowing from M to the processor, while the operands from another stream, the data stream, flowing to and from the processor.

Flynn's qualification divides computers into four broad groups based on the values of m_I and m_D associated with their CPUs.

Single instruction stream single data stream (SISD) $m_I = m_D = 1$. Conventional machines with a

single CPU capable only of scalar arithmetic fall into this category. SISD computers and sequential computers are synonymous.

- Single instruction stream multiple data stream (SIMD) $m_I = 1$, $m_D > 1$. This category includes such early parallel computers as ILLIAC IV that have a single program-control unit and many independent execution units.
- Multiple instruction stream single data stream (MISD) $m_I > 1$, $m_D = 1$. Few parallel computers fit well in this class. Fault tolerant computers where several CPUs process the same data using different programs are MISD.
- Multiple instruction stream Multiple data stream (MIMD) $m_I > 1$, $m_D > 1$. This category covers multiprocessors, which are computers with more than one CPU and the ability to execute several programs simultaneously.

8.4.2 Multiprocessors

A multiprocessor is an MIMD computer containing two or more CPUs that cooperate on common computational tasks. Multiprocessors are distinguished from multicomputers and computer networks, which are systems with multiple CPUs operating largely independently on separate tasks. The various processors making up a multiprocessor typically share resources such as communication facilities, IO devices, program libraries and databases and are controlled by a common operating system.

Motivation

The main reasons for including multiple CPUs in a computer system to improve performance and reliability. Performance is improved either by distributing the computation of a large task among several CPUs or by performing many small tasks in parallel using separate CPUs. A multiprocessor with n identical processors can, in principle, provide n times the performance of a comparable SISD system or uniprocessor. A major goal, therefore, in designing an n-CPU multiprocessor is to achieve a speedup S(n) as close to n as possible. By enabling such resources as secondary memory to be shared, a multiprocessor can reduce overall system costs. Many multiprocessors also have the advantage of scalability; that is,the system size can be increased incrementally by adding processors to meet growing computation needs. Scalability is facilitated by making all CPUs identical and allowing each to execute either operating system (kernel) or user code; multiprocessors with these properties are said to be symmetric. Finally, system reliability is improved by the fact that the failure of one CPU need not cause the entire system to fail. The functions of the faulty CPU can be taken over by the other CPUs; consequently, multiprocessors enable fault tolerance to be incorporated into the system.

Shared-memory and distributed-memory multiprocessors are sometimes referred to as tightly coupled and loosely coupled, respectively, reflecting the speed and ease with which they can interact on common tasks. Multiprocessors are also classified by the number of processors they contain: Massively parallel machines can contain thousands of processors. Most multiprocessors, however, are modestly parallel, containing from 2 to about 30 processors; such multiprocessors have existed since the 1960.

Shared-bus systems

Most commercial multiprocessors have been built around a single shared system bus B because of B' s relative simplicity and low cost. The CPUs, memory, and IO units are attached directly to B and time-share its communication facilities. Only one pair of units can use B at a time, either for CPU- memory or IO-memory communication. The memory units and IO devices on B are global to all the processors; hence single-bus multiprocessors are of the shared memory class. If the access time to the shared memory is the same for each processor the multiprocessor is said to be of the uniform-memory access (UMA) type.

The global bus B is clearly a communication bottleneck in shared-bus multiprocessor, leading to contention and delay whenever two or more units request access to main memory. In practice, memory contention limits to about 30 the number of CPUs that can be included in the system without an unacceptable degradation in performance

Consider a situation in which two CPUs share a region R of global memory where A mutual exclusion

applies; that is, only one processor should have access to the shared region at a time. Access to R is conveniently controlled by semaphore (flag) F that indicates whether R is currently being used by some other process (F=1) or is available for use by a new process (F = 0). Before it attempts to access R, a CPU first reads F, which must be stored in global memory. If F=0, the CPU then changes F to1 and proceeds to use R. If it finds that F is already 1, then it does not attempt to use R. The mutual exclusion requirement can be violated if it is possible for two CPUs to independently access the semaphore at the same time and find F=0. This violation can occur if a second processor CPU_2 can read F after the first processor CPU_1 has read it, but before CPU_1 has changed F to 1.The problem in the fact that semaphore flag test-and-set instructions issued by the CPUs can be broken down into interleaved bus cycles as follows:

Global bus cycle	**Action**
I	CPU_1 fetches semaphore F=0
i+1	CPU_2 fetches semaphore F=0
i+2	CPU_1 sets F to 1
i+3	CPU_2 sets f to 1

At time i + 4, both CPU_1 and CPU_2 assume they have exclusive control over the critical region R, with potentially catastrophic consequences.

Cache coherence

In shared-bus multiprocessors like the Symmatry, caches play a vital role in reducing the contention for the shared system bus. Without caches, connecting more than two or three CPUs to the same bus might be impratical. Typically, each CPU has a private one or two-level cache, which forms a local memory and allows the CPU to access data and instructions without using the system bus. With an independent cache in each CPU, the possibility exists for two or more caches to contain different (inconsistent) versions of the same information, that is the cache-coherence problem. This problem causes both the cache and main (global) memory to be updated whenever a memory write operation occurs. Suppose, for example, that one CPU updates variable X in both its cache and the global memory. If another CPU then changes X, the new value of X will be written into main memory, but the two caches will contain different values for X. Subsequent reads from these caches can lead to inconsistent results. Thus to ensure coherence we need a mechanism that informs each cache about changes to shared information stored in other caches.

We can solve the cache-coherence problem with either hardware or software. One software-based solution is to mark (tag) information during program compilation as either cacheable or noncacheable. All writable shared items are marked as noncacheable, meaning they can be accessed directly only from main memory. A write-through policy that requires a processor to mark a shared cache item X as invalid, or to be deallocated, whenever the processor writes into X can then ensure cache coherence. When the processor references X again, it is forced to bypass the cache and access main memory, thereby always acquiring the most recent version of X. This approach can significantly degrade system performance.

Hardware-based methods of maintaining cache coherence offer the advantages of higher speed and program transparency, but they tend to be expensive. One possible approach is for a processor to broadcast its write operation to all the caches and the global memory via the shared bus. Every cache controller in the system then examines its assigned addresses to see if the broadcast item is presently allocated to it. If it is, the cache block (line) in question is either updated or marked as dirty (modified). The drawback of this technique is that every cache write forces all caches to check the broadcast data, making the caches unavailable for normal processing.

A related, but less costly, hardware-based method known as cache snooping equips each CPU with circuitry to continuously monitor or "snoop" on system-bus activity in order to detect references by other processors to memory addresses currently in its cache. The CPU can also signal other CPUs that it has a copy of the referenced item and, when necessary, modify or delay the other CPUs' main-memory accesses. If CPU_2 attempts to read (write) memory data with an address that is currently assigned to CPU_1's cache, CPU_1 detects this attempt in what is called a snoop read (write) hit by CPU_1. On making a snoop hit,

CPU_1 determines whether actual or potential incoherence exists and then takes appropriate steps to eliminate it.

Message-passing computers

As developments in VLSI technology during the 1980s ushered 10 powerful one-chip, microprocessors and memory (RAM) chips with capacities in the multimegabit range, it has become feasible to build massively parallel multiprocessors, with hundreds or thousands of processor. Multiprocessor architectures with distributed memory systems, where interprocessor communication is by message-passing, avoid most of the contention problems inherent in the use of single shared memories and buses. Such computers can provide extremely high performance, but they also pose problems in algorithm and program design that are far from being satisfactorily solved.

Various static and dynamic interconnection structures have been proposed for massively parallel multiprocessors. Static structures like hypercubes abd trees are easier to build and control when many processors are involved. Dedicated buses or IO communication lines typically serve as interprocessor links. Neighboring processors can then interact at the maximum possible rate, with little interference from other processors. Interconnection networks are selected to trade hardware costs for communication speed in some class of applications. The hypercube structure achieves a good balance between these parameters. An n-dimensional hypercube computer is characterized by the presence of 2^n nodes, each consisting of a processor and its local memory. Each processor P_i has direct links to n other processors (its neighbors); these links form the edges of the hypercube. A set of 2^n distinct n-bit binary addresses can be assigned to the processor in such a way that P_i's address differs from each of its neighbors in exactly 1 bit. Hypercubes have several attractive features:

A hypercube can be expanded or scaled up while maintaining a good balance between the number of nodes and the cost of internode communication as n is incremented by one, the number of nodes doubles, but the node degree and the maximum internode distance both increase only by one (from n to n+1)

A hypercube is homogeneous in that the system appears the same when viewed from any of its nodes. This feature simplifies programming because all node can executes the same programs on different data when collaborating on a common task.

We can embed other useful interconnection structures, such as rings and meshes, efficiently in the hypercube. We say that (graph) G is embeddable in H if and only if every node in G can be mapped into a distinct node in H such that all nodes that are neighbors in G are also neighbors in H. In other words, G is embeddable in H if we can find an exact (isomorphic) copy of G inside H.

A large hypercube can support multiple concurrent users with each user program assigned to a private embedded hypercube or subcube that is disjoint from other users' subcubes. For example, in a four-dimensional hypercube four-node subcubes can be assigned to two users, and an eight-node subcube can be assigned to third user.

8.4.3 Self Learning Exercises

True/False

F. Computer performance can be increased by executing many instructions simultaneously or in parallel.

G If the access time to the shared memory is same for each processor, the multiple processor is said to be of the uniform-memory access (UMA) type.

Fill In the Blanks

H. A computer containing more than one CPU is a ________.

I. A multiprocessor is an ________ computer containing two or more CPUs that operate on common computational tasks.

J. MINs are ________.

8.5 Pipeline Control

Pipelining provides a basic way to speed up arithmetic operations as well as is also used to implement the entire instruction-processing behavior of high-performance CPUs.

Instruction Pipelines

During program execution, instructions pass through a sequence of processing steps that lend themselves naturally to pipelining. Consequently, a CPU can be organized as one or more pipelines, whose various stages fetch opcodes and operands, execute instruction, and store results in local registers or external memory. In general, an instruction pipeline is a multifunction, reconfigurable pipeline designed to speed up a computer's performance by efficiently overlapping the processing of instructions. An instruction pipeline is normally invisible to programmers and managed automatically by program compilers and the CPU's internal program-control unit. Instruction pipelines were first used in IBM 7030 (also known as Stretch) and a few other computers of the 1960s. They reemerged in the 1980s as key contributors to the high performance achieved by RISCs. Instruction pipelining has also been successfully incorporated into CISCs such as the 80X86/Pentium series, beginning with the 80486 microprocessor in 1989.

Pipeline structure

The general structure of a pipeline of *m* stages $S_1, S_2 \ldots S_m$ appears in

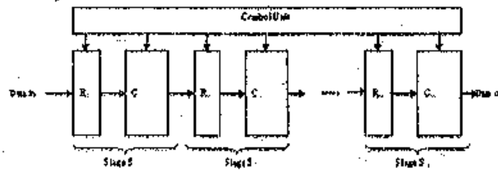


Figure 8.1 Structure of m-stage pipeline

When S_i has computed its results, it passes them, along with any unprocessed input operands, to S_{i+1} for further processing, and S_i receives a new set of operands from S_{i-1}. Thus the pipeline can contain up to *m* independent data sets, all in different stages of computation. Buffer registers and other synchronization logic are placed between stages so the stages do not interfere with one another. The performance speedup of an instruction pipeline derives from the fact that up to *m* independent instruction be in progress simultaneously in the *m* stages.

The simplest instruction pipeline breaks instruction processing into two parts: a fetch stage S_1 and an execute stage S_2. Thus a two-stage pipeline increases throughput by overlapping instruction fetching and instruction execution. While instruction I_i with address A_i is being executed by stage S_2, the instruction I_{i+1} with next consecutive address A_{i+1} is fetched from memory by stage S_1. If on executing I_i in S_2 is determined that a branch must be made to a nonconsecutive address A_j ,A_{i+1}, then the prefetched instruction I_{i+1} in S_1 has to be discarded

Multistage pipeline An m-stage instruction pipeline can overlap the processing of up to m instructions, so it is desirable to use more than two stages to maximize instruction throughput. The value of m depends on the maximum number of stages into which instruction processing can be efficiently broken. This number in turn depends on the complexity of the instruction set, the organization of the external memory M, and the way in which the CPU's datapath is implemented. In practice the number of pipeline stages ranges from three to a dozen or more.

8.5.2 Arithmetic Pipeline

It is used in floating point operations

- Floating point = (mantissa, exponent)

$X=(m_x,e_x)=m_x*2^{e_x}$

$Y=(m_y,e_y)=m_y*2^{e_y}$

- Floating point sum

$X+Y=(m_x*2^{e_x-e_y}+m_y)*2^{e_y}$

8.5.3 Pipeline Performance

A Pipeline performance can be measured by its throughput in terms of millions of instruction executed per second (MIPS) or number of clock cycle per instruction (CPI). These quantities are related by the equation:

$$CPI=f/MIPS$$

Where f is the pipeline's clock frequency in MHZ, and the values of CPI and MIPS are average figures that can be determined experimentally by processing suits of representative programs. The maximum value of CPI for a single program is one, making the pipeline's maximum possible throughput equal to f. this throughput is attained only when the pipeline is supplied with a continuous stream of instructions that keep all its stages busy. Superscalar machines reduce CPI below one by executing several instruction streams simultaneously using multiple pipelines. A space diagram for an m-stage pipeline has the form of an m x n grid, where n is the number of clock cycle to complete the processing of some sequence of N instructions of interest.

Another general measure of pipeline performance is the speedup S(m) defined by

$$S(m)=T(1)/T(m)$$

Where T(m) is the execution time for some target workload on an m-stage pipeline and T(1) is the execution time for same workload on a similar non-pipelined processor.

It is reasonable to assume that T(1) £ mT(m), in which case S(m) £m. A pipeline's efficiency and speedup are related as follows

$$S(m)= m \times E(m)$$

An easy way to improve a pipeline's performance is to increase the number of stages *m*. This assumes that the pipeline's processing tasks can be subdivided in a useful way and that the cost of doing so is acceptable. Each new stage S_i introduces some new hardware cost and delay due to its buffers register R_i and associated control logic. In particular, we will determine the pipeline's *performance/cost ratio PCR* defined as

$$PCR = f/K$$

Where f is pipeline's clock frequency and *K* is its hardware cost.

Control Scheme

An elegant way to control a pipeline for collision-free operation is by computing collision vectors. A *collision vector* CV for a reservation table R at time t is a binary vector $C_1C_2,...,C_{M-1}C_M$ where the *ith* bit C_i is 1 if initiating a pipeline instruction at $t+i$ results in a collision; C_i is 0 otherwise. An *initial collosion vector* CV_0 is obtained from the forbidden list *F* of *R* as follows. Element C of CV_0 is set to 1 if *i* is in *F*, and C is set to 0 otherwise, for $i = 1,2,...,M$, where M is the maximum element in *F*. A convenient way to store CV is in a shift register $CR=CR_1:CR_M$ called a *collision register*. By inspecting CR_1 at time *t*, we can determine whether issuing a new instruction in the next clock cycle $t+1$ will result in a collision. A simple left shift of CR, with the right-most bit CR_M set to 0 prepares CR_1 for inspection in the next clock cycle. If we decide to initiate a new instruction at t+1 then CR is left shifted and its contents are replaced by CR *or* CV_0 is the initial collision vector obtained from *F* as specified above and or denotes the bitwise OR operations. These actions ensure that CR defines all the collision possibilities due either to ongoing pipeline operations or to the newly initiated one.

The progress of an instruction stream through a pipeline can be delayed by various unfavorable dependency relationships among instructions and their data operands, which are collectively referred as hazards.

8.5.4 Superscalar Processing

Microprocessor that reach performance levels greater than one instruction per cycle by fetching, decoding and executing several instructions concurrently. This mode of operation is called superscalar. A superscalar computer has a single CPU that attempts to exploit the parallelism that is implicit in ordinary computer programs.

Characteristics

Superscalar operation requires a processor to detect and exploit instruction-level parallelism hidden in the programs it executes. A super scalar CPU has multiple execution units (E-units), each of which is usually pipelined, so that they constitute a set of independent instruction pipelines. The CPU's control unit PCU is designed to fetch and decode several instructions concurrently. It can issue or dispatch up to *k* instructions simultaneously to the various E-units where k, the instruction-issue degree, can be six or more using current technology. The need to process so many instructions simultaneously without performance-degrading conflicts greatly complicates the design of the PCU. The PCU of a superscalar machine is responsible for determining when each instruction can be executed and for providing it with access to the resources it

needs, such as memory operands, E-units, and CPU registers, in a prompt and efficient manner. To do so, it must take the following factors into account

- Instruction type: For example, a floating-point add instruction has to be issued to a floating-point E-unit and not to an integer E-unit.
- E-unit availability: An instruction can be Issued to a pipelined E-unit only if no collisions will result, as determined by the pipeline's reservation table.
- Data dependencies: To avoid conflicting use of registers, data-dependency constraints among the operands of the active instructions must be satisfied.
- Control dependencies: To maintain high performance levels, techniques are needed to reduce the impact of branch instructions on pipeline efficiency.
- Program order: Instructions must eventually produce results in the order specified by the program being executed. The results may, however, be computed out-of-order internally to improve the CPU' s performance.

8.5.5 Self Learning Exercises

True/False

K. A pipeline's performance can be measured by its throughput in terms of MIPS.

L. A superscalar computer has a multiple CPU that attempts to exploit parallelism.

Fill In the Blanks

M. A superscalar CPU has multiple ________

8.6 Summary

Input-output systems are distinguished by the extent of CPU involvement in IO operations. The use of CPU programs to control all phases of an IO operation is called programmed IO. By providing IO devices with DMA and IO interrupt control, data transfer can be implemented independently of the CPU. Multiprocessors have been designed around various interconnection networks of which the shared bus is most common. We can improve the performance of a CPU by structuring its program control and execution logic in form of one or more pipelines.

8.7 Glossary

DMA Controller	It is an external device which controls the DMA i.e. the process of high speed data transfer.
Instruction pipeline	It is multifunctional, re-configurable pipeline designed to speed up a computer's performance by efficiently overlapping the processing of instructions
Interrupts	It is an asynchronous process of communication with the microprocessor, initiated by an external peripheral.
IO Processor	It is a logical extension of the IO control methods.
Multiprocessor	It is an MIMD (Multiple Instruction stream Multiple Data stream) computer, which contains two or more CPUs that cooperate the common computational tasks.
Superscaler processing	It is the process of a superscalar computer, which has a single CPU that attempts to exploit the parallelism.

8.8 Further Readings

- J.P.Hayes: Computer Architecture and Organization, McGraw-Hill International
- R.S.Goankar: Microprocessor Architecture, Programming and Applications with the 8085/8080, 2nd Edition, New Age International Publishers Limited, ISBN-81-224-0710-2.
- K.L.Short: Microprocessors and Programmed Logic, 2nd Edition, Prentice Hall of India Pvt. Ltd. 1988, ISBN-0-07-100462-9.

8.9 Answer to Self Learning Exercises

Question	Answer	Question	Answer
A	False	H	Multiprocessor
B	programmed IO	I	MIMD
C	memory mapped IO	J	multistage interconnection networks
D	IO processor (IOP)	K	True
E	channel command words(CCW)	L	False
F	True	M	execution units (E-units)
G	True		

8.10 Unit End Questions:

1. Define Each of the following IO control methods.

a. Programmed IO

b. DMA Controllers

c. IO Processors.

List the advantages and disadvantages of each method with respect to program design complexity, Io band width and interface hardware costs.

2. What do you understand by parallel processing? Explain the terms SISD, SIMD, MISD, MIMD with reference to it.

3. Write short note on following-

a. Multiprocessor

b. Processor-level-parallelism

c. IOP organization

d. Instruction pipeline

Unit-09

Introduction to Micro Computer Systems

Structure of the Unit

9.0 Objectives
9.1 Introduction
9.2 Microprocessors
9.2.1 Binary Digits
9.2.2 Memory
9.2.3 Input/Output
9.2.4 Microprocessor as a CPU(MPU)
9.2.5 Organization of a Microprocessor-Based System
9.3 Microcontrollers
9.4 Microcomputer Devices
9.4.1 Microprocessor
9.4.2 Memory
9.4.3 Input
9.4.4 Output
9.5 Machine Language
9.6 Assembly Language
9.6.1 Advantage of assembly language
9.6.2 Difference between machine language and assembly language
9.7 Bus Concept
9.7.1 Address Bus
9.7.2 Data Bus
9.7.3 Control Bus
9.8 Architecture of 8085A
9.8.1 Pinout of 8085A
9.9 Summary
9.10 Glossary
9.11 Further Readings
9.12 Answer to Self Learning Exercise
9.13 Unit End Questions

9.0 Objectives

Objective of this unit is to draw a block diagram of a microprocessor based system and explain the functions of each component: microprocessor, memory and I/O and their lines of communication (the bus). Architecture and pinout of 8085A is also discussed.

9.1 Introduction

We have studied the organization of system in previous unit where we have discussed different modes of transfer i.e. programmed IO, DMA and Interrupts etc. We have also discussed different features and types of pipelines.

9.2 Microprocessor

A microprocessor is a multipurpose, programmable clock-driven, register-based electronic device that reads binary instructions from a storage device called memory, accepts binary data as input and processes data according to those instructions, and provides results as output.

A typical programmable machine can be represented with four components: microprocessor, memory, input, and output. These four components work together or interact with each other to perform a given task; thus, they comprise a system. The physical components of this system are called hardware. A set of instructions written for the microprocessor to perform a task is called a program, and a group of programs is called software.

The microprocessor applications are classified primarily in two categories: reprogrammable systems and embedded systems In reprogrammable systems, such as microcomputers, the microprocessor is used for computing and data processing. In embedded systems, the microprocessor is a part of a final product and is not available for reprogramming to end use.

9.2.1 Binary Digits

The microprocessor operates in binary digits, 0 and 1, also known as bits. Bit is an abbreviation for the term binary digit. These digits are represented in terms of electrical voltages in the machine, Generally, 0 represents one voltage, level, and 1 represents another. The digits 0 and 1 are also synonymous with low and high, respectively. Each microprocessor recognizes and processes a group of bits called the word, *and* microprocessors are classified according to their word length. For example, a processor with an 8-bit word is known as 8-bit microprocessor and a processor with an 32-bit word is known as 32-bit microprocessor.

Microprocessor is designed to understand and execute many instructions. It is multipurpose machine. It can be used to perform various sophisticated functions, as well as simple tasks such as turning devices on or off. A programmer can select appropriate instructions and ask the microprocessor to perform various tasks on given set of data.

9.2.2 Memory

Memory is like the pages of notebook with space for a fixed number of binary numbers on each line. However, these pages are generally made of semiconductor material. Typically, each line is an 8-bit register that can store eight binary bits and several of these registers are arranged in a sequence called memory. These registers are always grouped together in powers of two. For example, a group of 1024(2^{10}) 8-bit registers on a semiconductor chip is known as 1K byte of memory. The user writes the necessary instructions and data in memory through an input device, and asks the microprocessor to perform the given task and find an answer. The answer is generally displayed at an output device or stored in memory.

9.2.3 Input/Output

The user can enter instructions and data into memory through devices such as keyboard or simple switches. These devices are called input devices. The microprocessor reads the instructions from the memory and processes the data according to those instructions. The result can be displayed by a device such as seven segments LEDs (Light Emitting Diodes) or printed by a printer. These devices are called output devices.

9.2.4 Microprocessor As A CPU (MPU)

We can also view the microprocessor as a primary component of a computer. Computer has four components: memory, input, output and the central processing unit (CPU), which consists of the arithmetic/logic unit (ALU) and the control unit. The CPU contains various registers to store data, ALU to perform arithmetic and logical operations, instruction decoders, counters, and control lines. The CPU reads instructions from the memory and performs the tasks specified. It communicates with input/output devices either to accept or to send data. These devices are known as peripherals. The CPU is the primary and central player in communicating with devices such as memory, input, and output. However, the timing of the communication process is controlled by the group of circuits called the control unit.

A computer with a microprocessor as its CPU is known as a microcomputer. The terms microprocessor and microprocessor unit (MPU) are often used synonymously. MPU implies a complete processing unit with the necessary control signals. Because of the limited number of available pins on a microprocessor package, some of the signals (such as control and multiplexed signals) need to be generated by using discrete devices to make the microprocessor a complete functional unit or MPU.

A semiconductor fabrication technology became more, advanced, manufacturers were able to place not only MPU but also memory and I/O interfacing circuits on a single chip; this is known as a microcontroller or microcontroller unit (MCU). A microcontroller is essentially an entire computer on a single chip.

9.2.5 Organization of a Microprocessor-Based System

A microcomputer is one among many microprocessor-based systems. It includes three components: microprocessor; I/O.(input/output) and memory (read/write m.emory and read-only memory). These

components are organized around a common communication path called a bus. The entire group of components is also referred to as a system or a microcomputer System, and the components themselves are referred to as sub-systems. At the outset, it is necessary to differentiate between the terms *microprocessor* and *microcomputer* because of the common misuse of these terms in popular literature. The microprocessor is one component of the microcontroller. On the other hand, themicrocomputer is a complete computer, similar to any other computer, except that CPU functions of the microcomputer are performed by the microprocessor. Similarly, the term peripheral is used for input/output devices.

9.3 Microcontroller

A microcontroller (also MCU or μC) is a functional computer system on a chip. It contains a processor core, memory, and programmable input/output peripherals.

Microcontrollers include an integrated CPU, memory (a small amount of RAM, program memory, or both) and peripherals capable of input and output.

It emphasizes high integration, in contrast to a microprocessor which only contains a CPU. In addition to the usual arithmetic and logic elements of a general purpose microprocessor, the microcontroller integrates additional elements such as read-write memory for data storage, read-only memory for program storage, Flash memory for permanent data storage, peripherals, and input/output interfaces. At clock speeds of as little as 32KHz, microcontrollers often operate at very low speed compared to microprocessors, but this is adequate for typical applications. They consume relatively little power (milliwatts or even microwatts), and will generally have the ability to retain functionality while waiting for an event such as a button press or interrupt. Power consumption while sleeping (CPU clock and peripherals disabled) may be just nanowatts, making them ideal for low power and long lasting battery applications.

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, remote controls, office machines, appliances, power tools, and toys. By reducing the size, cost, and power consumption compared to a design using a separate microprocessor, memory, and input/output devices. Microcontrollers make it economical to electronically control many more processes.

A microcontroller is a single integrated circuit, commonly with the following features:

- central processing unit - ranging from small and simple 4-bit processors to complex 32- or 64-bit processors
- discrete input and output bits, allowing control or detection of the logic state of an individual package pin
- serial input/output such as serial ports (UARTs)
- other serial communications interfaces like I²C, Serial Peripheral Interface and Controller Area Network for system interconnect
- peripherals such as timers, event counters, PWM generators, and watchdog
- volatile memory (RAM) for data storage
- ROM, EPROM, EEPROM or Flash memory for program and operating parameter storage
- clock generator, often an oscillator for a quartz timing crystal, resonator or RC circuit
- many include analog-to-digital converters
- in-circuit programming and debugging support

This integration drastically reduces the number of chips and the amount of wiring and circuit board space that would be needed to produce equivalent systems using separate chips. Furthermore, and on low pin count devices in particular, each pin may interface to several internal peripherals, with the pin function selected by software. This allows a part to be used in a wider variety of applications than if pins had dedicated functions. Microcontrollers have proved to be highly popular in embedded systems since their introduction in the 1970s.

Some microcontrollers use a Harvard architecture: separate memory buses for instructions and data, allowing accesses to take place concurrently. Where a Harvard architecture is used, instruction words for the processor may be a different bit size than the length of internal memory and registers; for example: 12-bit instructions used with 8-bit data registers.

The decision of which peripheral to integrate is often difficult. The microcontroller vendors often trade operating frequencies and system design flexibility against time-to-market requirements from their customers and overall lower system cost. Manufacturers have to balance the need to minimize the chip size against additional functionality.

Microcontroller architectures vary widely. Some designs include general-purpose microprocessor cores, with one or more ROM, RAM or I/O functions integrated onto the package. Other designs are purpose built for control applications. A microcontroller instruction set usually has many instructions intended for bit-wise operations to make control programs more compact. For example, a general purpose processor might require several instructions to test a bit in a register and branch if the bit is set, where a microcontroller could have a single instruction that would provide that commonly-required function.

Microcontrollers typically do not have a math coprocessor, so floating point multiplication and division are carried out using a standard library.

Microcontrollers are sometimes called *embedded microcontrollers,* which just means that they are part of an embedded system — that is, one part of a larger device or system.

A microcontroller differs from a microprocessor, which is a general-purpose chip that is used to create a multi-function computer or device and requires multiple chips to handle various tasks. A microcontroller is meant to be more self-contained and independent, and functions as a tiny, dedicated computer.

The great advantage of microcontrollers, as opposed to using larger microprocessors, is that the parts-count and design costs of the item being controlled can be kept to a minimum. They are typically designed using CMOS (complementary metal oxide semiconductor) technology, an efficient fabrication technique that uses less power and is more immune to power spikes than other techniques.

There are also multiple architectures used, but the predominant architecture is CISC (Complex Instruction Set Computer), which allows the microcontroller to contain multiple control instructions that can be executed with a single macro instruction. Some use a RISC (Reduced Instruction Set Computer) architecture, which implements fewer instructions, but delivers greater simplicity and lower power consumption.

Early controllers were typically built from logic components and were usually quite large. Later, microprocessors were used, and controllers were able to fit onto a circuit board. Microcontrollers now place all of the needed components onto a single chip. Because they control a single function, some complex devices contain multiple microprocessors.

Microcontrollers have become common in many areas and can be found in home appliances, computer equipment, and instrumentation. They are often used in automobiles, and have many industrial uses as well and have become a central part of industrial robotics. Because they are usually used to control a single process and execute simple instructions, microcontrollers do not require significant processing power.

The automotive market has been a major driver of microcontrollers, many of which have been developed for automotive applications. Because automotive microcontrollers have to withstand harsh environmental conditions, they must be highly reliable and durable. Nonetheless, automotive microcontrollers, like their counterparts, are very inexpensive and are able to deliver powerful features that would otherwise be impossible, or too costly to implement.

9.4 Microcomputer Devices

In addition to the microprocessor and memory, we need various input and output devices. The system needs a temperature sensor as an input device to sense room temperatures, and three output devices-a fan, a heater, and an LCD panel for display.

9.4.1 Microprocessor

The processor will read the binary instructions from memory and execute those instructions continuously. It will read the temperature, display it at the LCD display panel, and turn on/off the fan and the heater based on the temperature

9.4.2 Memory

The system includes two types of memory. ROM (read-only memory) will be used to store the program, called the monitor program, that is responsible for providing the necessary instructions to the processor to

monitor the system. This will be a permanent program stored in ROM and will not be altered. The R/W (read-write) memory is needed for temporary storage of data.

9.4.3 Input

In this system, we need a device that can translate temperature (measurement of heat) into an equivalent electrical signal; a device that translates one form of energy into another form is called a transducer. For example a microphone is a transducer that converts sound energy into an electrical signal, and a thermocouple is a transducer that converts heat into an electrical signal. These days temperature sensors are available as integrated circuits. A temperature sensor is a three-terminal semiconductor electronic device that generates a voltage signal that is proportional to the temperature. However, this is an analog signal and our processor is capable of handling only binary bits. Therefore, this signal must be converted into digital bits. The analog-to-digital (A/D) converter performs that function. The A/D converter is also an electronic semiconductor chip that converts an input analog signal into the equivalent eight binary output signals. In microprocessor based systems, devices that provide binary inputs(data) are connected to the processor using devices such as buffers called input ports. This A/D converter is an input port, and it will be assigned a binary number called an address. The microprocessor reads this digital signal from the input port.

9.4.4 Output

There are three output devices: fan, heater and liquid crystal display (LCD). These devices are connected to the processor using latches called output ports.

Fan: This is an output device, identified as port1, that is turned on by the processor when the temperature reaches a set higher limit.

Heater: This is also an output device, identified as port2, that is turned on by the processor when the temperature reaches a set lower limit.

Liquid Crystal Display (LCD): This display is made of crystal material placed between two plates in form of dot matrix or segments. It can display letters, decimal digits or graphic characters. The LCD is used to display temperatures.

9.4.5 Self Learning Exercise

True/False

A. A microprocessor is a multipurpose, programmable clock-driven, register-based electronic device that reads binary instructions from a storage device called memory.

B. A microcontroller (also MCU or μC) is a functional computer system on multiple chips

C. Microcontrollers are used in automatically controlled products and devices.

Fill In the Blanks

D. The microprocessor operates in binary digits, 0 and 1, also known as ________.

E. Computer has four components: _________, input, output and the central processing unit (CPU).

F. Microcontroller contains a processor core, memory, and programmable________ peripherals.

G Three micro computer output devices are-a fan, a _________ , and an LCD panel for display.

9.5 Machine language

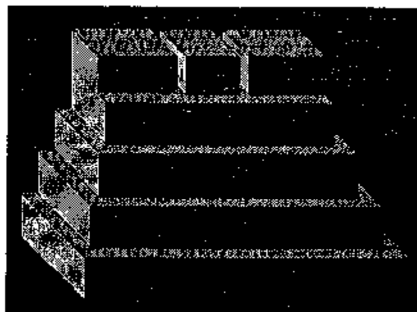


Figure 9.1 Arrangement of different languages

It is the lowest-level programming language (except for computers that utilize programmable microcode) Machine languages are the only languages understood by computers. While easily understood by computers, machine languages are almost impossible for humans to use because they consist entirely of numbers. Programmers, therefore, use either a high-level programming language or an assembly language. An assembly language contains the same instructions as a machine language, but the instructions and variables have names instead of being just numbers.

Programs written in high-level languages are translated into assembly language or machine language by a compiler. Assembly language programs are translated into machine language by a program called an assembler.

Every CPU has its own unique machine language. Programs must be rewritten or recompiled, therefore, to run on different types of computers. Sometimes referred to as machine code or object code, machine language is a collection of binary digits or bits that the computer reads and interprets. Machine language is the only language a computer is capable of understanding.

9.6 Assembly Language

Assembly language, commonly referred to as assembly, is a more human readable form of machine language. Every computer architecture uses its own assembly language thus processors using an architecture based on the x86, PowerPC, or TI DSP will each use their own language. *Machine language* is the pattern of bits encoding a processor's operations. Assembly will replace those raw bits with a more readable symbols call *mnemonics.*

For example, the following code is a single operation in machine language.

```
0001110010000110
```

For practical reasons, a programmer would rather use the equivalent assembly representation for the previous operation.

```
ADD     R6,R2,R6     ; Add $R2 to $R6 store in $R6
```

This is a typical line of assembly. The op code ADD instructs the processor to add the operands R2 and R6, which are the contents of register R2 to register R6, and store the results in register R6. The ";" indicates that everything after that point is a comment, and is not used by the system.

Assembly has a one-to-one mapping to machine language. Therefore, each line of assembly corresponds to an operation that can be completed by the processor. This is not the case with high-level languages. The assembler is responsible for the translation from assembly to machine language. The reverse operation is completed by the dissassembler.

Assembly instructions are very simple, unlike high-level languages. Often they only accomplish a single operation. Functions that are more complex must be built up out of smaller ones.

The following are common types of instructions:

- Moves:
  - o Set a register to a fixed constant value
  - o Move data from a memory location to a register (a load) or move data from a register to a memory location (a store). All data must be fetched from memory before a computation may be performed. Similarly, results must be stored in memory after results have been calculated.
  - o Read and write data from hardware devices and peripherals
- Computation:
  - o Add, subtract, multiply, or divide. Typically, the values of two registers are used as parameters and results are placed in a register
  - o Perform bitwise operations, taking the conjunction/disjunction (and/or) of corresponding bits in a pair of registers, or the negation (not) of each bit in a register
  - o Compare two values in registers (>, <, >=, or <=)
- Control Flow:
  - o Jump to another location in the program and execute instructions there
  - o Jump (branch) to another location if a certain condition holds
  - o Jump to another location, but save the location of the next instruction as a point to return to (a call)

9.6.1 Advantages of Assembly

The greatest advantage of assembly programming is raw speed. A diligent programmer should be able to optimize a piece of code to the minimum number of operations required. Less waste will be produced by extraneous instructions. However, in most cases, it takes an in-depth knowledge of the processor's instruction set in order to produce better code than the compiler writer does. Compilers are written in order to optimized your code as much as possible, and in general, it is hard to write more efficient code than it.

Low-level programming is simply easier to do with assembly. Some system-dependent tasks performed by operating systems simply cannot be expressed in high-level languages. Assembly is often used in writing device drivers, the low level code that is responsible for the interaction between the operating system and the hardware.

Processors in embedded space, such as the ez430, have the potential for the greatest gain in using assembly. These systems have very limited computational resources and assembly allows the maximum functionality from these processors. However, as technology advances, even the lowest power microcontroller is able to become more powerful for the same low cost.

9.6.2 Difference between machine language and assembly language

Machine language is the actual bits used to control the processor in the computer, usually viewed as a sequence of hexadecimal numbers (typically bytes). The processor reads these bits in from program memory, and the bits represent "instructions" as to what to do next. Thus machine language provides a way of entering instructions into a computer (whether through switches, punched tape, or a binary file).

Assembly language is a more human readable view of machine language. Instead of representing the machine language as numbers, the instructions and registers are given names (typically abbreviated words, or mnemonics, eg ld means "load"). Unlike a high level language, assembler is very close to the machine language. The main abstractions (apart from the mnemonics) are the use of labels instead of fixed memory addresses, and comments.

An assembly language program (ie a text file) is translated to machine language by an *assembler*. A *disassembler* performs the reverse function (although the comments and the names of labels will have been discarded in the assembler process).

9.6.3 Self Learning Exercise

True/False

H. Machine languages are the only languages understood by computers.

I. Assembly language programs are translated into programming language by a program called an assembler.

J. Computation means Compare two values in registers (>, < , >=, or <=)

K. A disassembler performs the same function as assembler.

Fill In the Blanks

L. Assembly language is a more human readable form of ________ language.

M. Assembly has a _________ mapping to machine language.

N. ______ is used to set a register to a fixed constant value .

O. Machine language is the actual bits used to control the ___________ in the computer.

9.7 Bus

A set of parallel conductors, which allow devices attached to it to communicate with the CPU is called as Bus

The bus consists of three main parts:

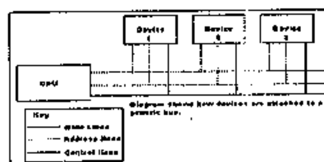


Figure 9.2 Device Attachment to a generic bus

· Control lines :
These allow the CPU to control which operations the devices attached should perform, i.e. read or write.
· Address lines
Allows the CPU to reference certain (Memory) locations within the device.
· Data lines
The meaningful data which is to be sent or retrieved from a device is placed on to these lines.
The Bus is set to run at a specified speed which is measured in MHz

9.7.1 Address Bus

In this information transfer takes place in only one direction, from the microprocessor to the memory or I/O elements, there this called a unidirectional bus. For 8-bit microprocessors, this bus is typically 16 bits long. The CPU can generate $2^{16} = 65536$ different possible addresse on this bus. A memory location or an I/O element can be represented by each one of these addresses. For example, an 8-bit data word can be stored in address 2000_{16}

When the microprocessor wants to transfer information between itself and a certain memory location or I/O device, it generates the 16-bit address from an internal register on its 16 address pins a0-a15, which then appears on the address bus. These 16 address bits are decoded to determine the desired memory location or I/O device. This decoding process normally requires hardware.

9.7.2 Data Bus

In this data can flow in both directions, to or from the microprocessor, therefore this is a bidirectional bus. In some microprocessor, the datapins are used to send other information such as bits in addition to data. This means the data pins are time-shared or multiplexed. The intel 8085 microprocessor, used as the CPU in the intel SDK-85 microcomputer, is an example where the lower 8 bits of the address are multiplexed on the data bus.

9.7.3 Control Bus

The bus consists of a number of signals that are used to synchronize the operation of the individual microcomputer elements. The microprocessor sends some of these control signals to the other elements to indicate the type of operation being performed. Each microcomputer has a unique set of control signals. However, there are some control signals that are common to most microprocessor.

9.7.4 Self Learning Exercise

True/False

P. Address Bus is called as unidirectional bus.

Q. Address lines allow the CPU to reference certain (Memory) locations within the device.

Fill In the Blanks

R. The bus consists of three main parts: control lines, address lines and ________.

S. In ________ data can flow in both directions, to or from the microprocessor, therefore this is a bidirectional bus.

9.8 Architecture of the 8085A

The internal architecture of the 8085A contains a register array with both dedicated and general purpose registers.

- A 16 bit program counter (PC)
- A 16 bit stack pointer (SP)
- Six 8-bit general purpose registers arranged in pairs: BC, DE, HL
- A temporary register pair: WZ

A 16-bit program counter addresses instructions in anyone of 65,536 possible memory locations. When the RESET IN pin of the 8085A is made logic 0, the program counter is reset to zero; when the RESET IN pin is returned to logic 1, the control unit transfers the contents of the PC to the address latch, providing the address of the first instruction to be executed. Thus, program execution in the 8085A begins with the instruction in memory location zero.

8085A Istructions are 1 to 3 bytes in length. The first byte always contains the operation code (OP code). During the instruction fetch, the first byte is transferred from the memory by way of the external data bus through the data bus buffer latch into the instruction register. The PC is automatically incremented so it contains the address of the next instruction if the instruction contains only 1 byte, or the address of the next byte of the present instruction if the instruction consists of 2 or 3 bytes.

In case of a multibyte instruction, the timing and control section provides additional operations to read in the additional bytes. The timing and control section uses the instruction decoder output and external control signals to generate the state and cycle timing signals and signals for the control of external devices. After all the bytes of an instruction have been fetched into the microprocessor, the instruction is executed. Execution may require transfer of data between the microprocessor and memory or an I/O device. For these transfers, the memory or I/O device address placed in the address latch comes from the instruction which was fetched or from one of the register pairs used as a data pointer: HL, BC, or DE.

six general purpose registers in the register array can be used as single 8-bit registers or as 16-bit register pairs. The temporary register pair, WZ, is not program addressable and is only used by the control unit for the internal execution of instructions. For example, to address an external register for a data transfer, WZ is used to temporarily hold the address from an instruction read into the microprocessor until the address is transferred to the address and address/data latch.

The 16-bit stack pointer, SP, maintains a pointer to the top of the stack allocated in external memory. The stack, as previously indicated, primarily supports interrupt and subroutine programming.

The 8085A's arithmetic logic unit performs arithmetic and logic operations on data. The operands are stored in two registers associated with the ALU: the 8-bit accumulator and the 8-bit *temporary register.* The accumulator is loaded from the internal bus and can transfer data to the internal bus. Thus, it serves both as a destination and *source* register for data. The temporary register temporarily holds one of the operands during a binary operation. For example, if the contents of register B are to be added to the contents of the accumulator and the result left in the accumulator, the temporary register holds a copy of the contents of register B while the arithmetic operation is taking place.

Associated with the ALU is the 5-bit flag register, F, which indicates conditions associated with the results of arithmetic or logic operations. The flags indicate zero, a carry out of the high order bit, the sign (most significant bit), parity, and auxiliary carry (carry out of the fourth bit).

8085A's internal data bus is 8 bits wide and transfers instructions and data among the various internal registers or to external devices through the multiplexed address/ data bus buffer latch. The bidirectional, three-state *address/data bus buffer latch isolates the* microprocessor's internal data bus from the external system address/data bus. In the output mode, the information on the internal bus is loaded into the 8-bit data latch that drives the address/data bus output buffer. The output buffers are floated during input or nontransfer operations. During the input mode, data from the external data bus is transferred over the internal data bus to an internal register.

9.8.1 Pinout of 8085A

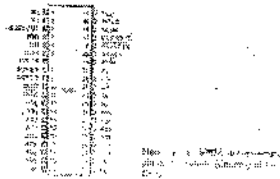


Figure 9.3 8085A microprocessor pin configuration

The 8085A is an 8-bit microprocessor suitable for a wide range of applications. It is a single chip, NMOS device implemented with approximately 6200 transistors on a 164 X 222 mil chip contained in a 40-pin dual-in-line package. The package pins their configuration are shown in Figure 9.3. The instruction set of the 8085Aconists of 74 instructions.

The 8085A operates on a single 5 V power *supply* connected at *V*cc; power supply ground is connected

to *V*ss. The frequency of the internal clock generator, which synchronizes the operation of the 8085A, is determined by a crystal or *RC network* connected at pins X_1 and X_2. The internal clock generator oscillates at twice the basic microprocessor frequency. A 50 percent duty cycle two phase, nonoverlapping clock is derived from oscillator. A 6.25 MHz crystal provides a 3.125 MHz internal clock frequency. A TTL level clock output, CLK(OUT),derived from one phase of the internal clock, provides a clock signal that can be used for synchronizing external devices. Instead of a crystal or *RC* network, an external clock can be connected to X_1. The remaining package pins provide the address, data, and control signals.

Intel's later version of the 8085A, the 8085AH, uses an NMOS technology called HMOS. This implementation of the 8085A has improved electrical characteristics, including 20 percent lower power consumption, wider voltage margins, and higher speed versions. CMOS implementations of the 8085A, such as OKI Semiconductor's MSM80C85A, are also available. Architecturally and functionally, these implementations of the 8085A are identical. Where the distinction is not important, the various implementations will simply be referred to generically as the 8085A. Detailed timing calculations in this text use the parameters for the 8085AH.

An 8085A microcomputer system can be constructed with standard ROM and RWM or with specially designed ICs that contain memory and I/O ports. Figure 9.4 shows a microcomputer system using standard ROM and RWM. The 8085A is capable of directly addressing up to 64 K memory locations with its 16-bit address. Eight of the 16 bits, A_8-A_{15} are provided directly on the three-state address pins, A_8-A_{15}. The other eight bits, A A_0-A_7 are provided on the bidirectional, three-state address/data pins, AD_0-AD_7. The address/data pins are time multiplexed: at times carrying addresses, at other times carrying data. Address information is provided on the address/data pins by the 8085A at the beginning of each memory reference, and is externally latched and held during the remainder of the memory reference to provide address bits A_0-A_7. The 8-bit latch in Figure 9.4 latches the address information from the address/data pins when clocked by the address latch enable signal, ALE. The 8085A generates this signal at the appropriate time when providing address information on its address/data pins. At other times during a memory reference, a byte of data is transferred to or from the memory on the address/ data pins. The 8085A generates two control pulses to indicate whether it is reading, RD or writing, WR, an external register. Time multiplexing of the address and data reduces the number of pins on the microprocessor package.

I/O ports are basically external registers. They can be interfaced as memory and written to and read from by any instruction that references memory. Alternatively, special instructions can read or write an I/ O port. The 8085A directly addresses up to 256 input and 256 output ports, using special I/O instructions with an 8-bit address. This 8-bit address is repeated on pins AD_0-AD_7 and A_8-A_{15} when an I/O device is addressed, and, therefore, the I/O device need only decode one set of these identical address bits. Another control signal, IO/M, generated by the 8085A indicates whether the microprocessor wants to read or write memory or I/O. When this signal is logic 0, memory is being referenced; when it is logic 1, I/O is being referenced. The signals RD, WR, and IO/M are used together in the system design to control the reading and writing of external memory and I/O ports. Table 9.1 summarizes the functions of the 8085A pins discussed in this section.

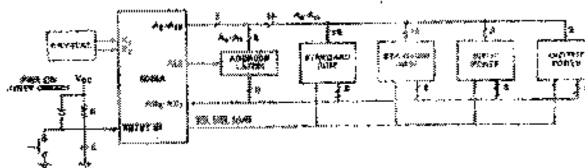


Figure 9.4 8085 A microprocessor system using standard ROM and R/WM

Symbol	Type	Name and Functions
A_8-A_{15}	O	**Address Bus:** The most significant 8 bits of the memory address or the 8 bits of the I/O address, 3-stated during Hold and Halt modes and during RESET.
$AD_{0\text{-}7}$	I/O	**Multiplexed Address / Data Bus:** Lower 8 bits of memory address(or I/O address) appear on the bus during the first clock cycle. It then becomes the Data bus during the second and third cycles.
A	O	**Address Latch Enable:** It occurs during the first clock state of a machine cycle and enables the address to get latched into the on-chip latch of peripherals. The falling edge of ALE is set to guarantee setup and hold times for the address information. The falling edge of ALE can also be used to strobe the status information. ALE is never 3-stated.
$\overline{RD}$	O	**Read Control:** A low level on RD indicates the selected memory or I/O device is to be read and that the Data bus is available for the data transfer, 3-stated during Hold and Halt modes and during RESET.
$\overline{WR}$	O	**Write Control:** A low level on WR indicates the data on the Data Bus is to be written into the selected memory or I/O location. Data is set up at the trailing edge of wR. 3-stated during Hold and Halt modes and during RESET.
S_0,S_1 and IO/$\overline{M}$	O	**Machine Cycle Status:** **IO/$\overline{M}$ — S_1 — S_2 — Status** 0 0 1 Memory Write 0 1 0 Memory read 1 0 1 I/O write 1 1 0 I/O read 0 1 1 Opcode fetch 1 1 1 Opcode fetch 1 1 1 Interrupted Acknowledge * 0 0 Halt * x x Hold * x x Reset * = 3-state (high impedance) x= unspecified S_1 can also be used as an advanced R/W status. IO/M, S_0 and S_1 become valid at the beginning of a machine cycle and remain stable throughout the cycle. The falling edge of ALE may be used to latch the state of these lines.
X1,X2	I	**X_1 and X_2:** are connected to a crystal, LC or RC network to drive the internal clock generator. X1 can also be an external clock input from a logic gate. The input frequency is divided by 2 to give the processor's internal operating frequency.
CLK	O	**Clock:** clock output for use as a system clock. The period of CLK is twice the X_1, X_2 input period.
Vcc		**Power:** +5 volt supply.
Vss		**Ground** reference

Table 9.1 Description of some 8085A pins

Address Bus	A group of lines that are used to send a memory address or a device address from the MPU to the memory location or the peripheral. The 8085 microprocessor has 16 address lines.
ALU	The group of circuits that performs arithmetic and logic operations. The ALU is a part of the CPU.
Assembly Language	A medium of communication with a computer in which programs are written in mnemonics. An assembly language is specific to a given computer.
Bus	A group of lines used to transfer bits between the microprocessor and other components of the computer system.
Control Bus	Single lines that are generated by the MPU to provide timing of various operations.
CPU	The Central Processing Unit. The group of circuits that processes data and provides control signals and timing. It includes the arithmetic/logic unit, registers, instruction decoder, and the control unit.
CU	The group of circuits that provides timing and signals to all operations in the computer and controls data flow.
Data Bus	A group of bi-directional lines used to transfer data between the MPU and peripherals (or memory). The 8085 microprocessor has eight data lines.
Input	A device that transfers information from outside world to the computer.
Instruction	A command in binary that is recognized and executed by the computer to accomplish a task. Some instructions are designed with one word, and some require multiple words.
Machine Language	The binary medium of communication with a computer through a designed set of instructions specific to each other.
Memory	A medium that stores binary information (instructions and data)
Microcontroller	A device that includes microprocessor, memory, and I/O signal lines on a single chip, fabricated using VLSI technology.
Microprocessor	A semiconductor device (integrated circuit) manufactured by using the LSI technique. It includes the ALU, register arrays, and control circuits on a single chip. The term (MPU) is also synonymous with the microprocessor.
Output	A device that transfers information from computer to the outside world.
RAM	A memory that stores binary information during the operation of computer. This memory is used as a writing pad to write user programs and data. The information

9.8.2 Self Learning Exercise

True/False

T. In case of a multibyte instruction, the timing and control section provides additional operations to read in the additional bytes.

U. The unidirectional, three-state address/data bus buffer latch isolates the microprocessor's internal data bus from the external system address/data bus.

Fill In the Blanks

V. The 8085A's ____________ unit performs arithmetic and logic operations on data.

W. The ___________ register holds a copy of the contents of a register while the arithmetic operation is taking place.

X. During the input mode, data from the external data bus is transferred over the _______ data bus to an internal register.

Y. The instruction set of the 8085A conists of ____ instructions.

9.9 Summary

In this unit we have discussed various terms like CPU, ALU, CU, memory, Input and Output. The microprocessor reads the instructions from main memory and communicates with all peripherals using the system bus. The various components of microprocessor are memory, I/O devices and bus. In machine language instructions are written in form of 0s and 1s where as in assembly language English like words are used to present binary instructions. We have also discussed Bus concept and architecture and pinout of 8085 A.

9.10 Glossary

9.11 Further Readings

- J.P.Hayes: Computer Architecture and Organization, McGraw-Hill International
- R.S.Goankar: Microprocessor Architecture, Programming and Applications with the 8085/8080, 2nd Edition, New Age International Publishers Limited, ISBN-81-224-0710-2.
- K.L.Short: Microprocessors and Programmed Logic, 2nd Edition, Prentice Hall of India Pvt. Ltd. 1988, ISBN-0-07-100462-9.

9.12 Answer to Self Learning Exercises

Question	Answer	Question	Answer
A	True	N	Moves
B	False	O	Processor
C	True	P	True
D	Bits	Q	True
E	Memory	R	Data lines
F	Input/output	S	Data bus
G	Heater	T	True
H	True	U	False
I	False	V	Arithmetic logic
J	True	W	Temporary
K	False	X	Internal
L	Machine	Y	74
M	One-to-one	Z	74

9.13 Unit End Questions

1. Write short note on

a. Microprocessor

b. Microcontroller

c. Microcomputer devices
2. State the difference between Machine Language and Assembly Language
3. Explain the architecture of 8085A.

Unit-10
Introduction to Micro Computer Systems
Assembly Language And Programming In 8085
Structure Of The Unit

10.0 Objective
10.1 Introduction
10.1.1 What is an Assembler?
10.1.2 What the Assembler does?
10.2 Programming model of 8085 microprocessor
10.3 Addressing modes
10.4 Instruction set classification
10.4.1 Data Transfer (Copy) Operations
10.4.2 Arithmetic Operations
10.4.3 Logical Operations
10.4.4 Branching Operations
10.4.5 Machine Control Operations
10.5 Instruction format
10.6 Conditional Assembly
10.7 Subroutines
10.7.1 Transferring data to subroutines
10.8 MACROS
10.8.1 Why use Macros?
10.8.2 What is a Macro?
10.8.3 Macro Vs Subroutine
10.8.4 Using Macros
10.9 Interrupts
10.9.1 Interrupt Concepts
10.9.2 Writing Interrupt Subroutines
10.10 Summary
10.11 Glossary
10.12 Further Readings
10.13 Answers to Self Learning Exercises
10.14 Unit End Questions

10.0 Objective

After studying this unit, you will learn

- Instruction set of 8085
- Program structures of 8085
- Macros and subroutines, Stack, Counter and timing delay
- Interrupt structure and its programming.

10.1 Introduction

Almost every line of source coding in an assembly language source program translates directly into a machine instruction for a particular processor. Therefore, the assembly language programmer must be familiar with both the assembly language and the processor for which he is programming.

Before moving to assembly language programming of 8085. We will start with the assembly language programming and assembler concept to better understand 8085 programming.

10.1.1 What is an Assembler ?

An assembler is a software tool - a program - designed to simplify the task of writing computer programs. If you have ever written a computer program directly in a machine-recognizable form such as binary or

hexadecimal code, you will appreciate the advantages of programming in a symbolic assembly language. Assembly language operation codes (opcodes) are easily remembered (MOV for move instructions, JMP for jump). You can also symbolically express addresses and values referenced in the operand field of instructions. Since you assign these names, you can make them as meaningful as the mnemonics for the instructions. For example, if your program mis-manipulates a date as data, you can assign it the symbolic name DATE. If your program contains a set of instructions used as a timing loop (a set of instructions executed repeatedly until a specific amount of time has passed), you can name the instruction group TIMER.

10.1.2 What the Assembler Does ?

To use the assembler, you first need a source program. The source program consists of programmer-written assembly language instructions. These instructions are written using mnemonic opcodes and labels as described previously. Assembly language source programs must be in a machine-readable form when passed to the assembler. The Intellect development system includes a text editor that will help you maintain source programs as paper tape files or diskette files. You can then pass the resulting source program file to the assembler, The assembler program performs the clerical task of translating symbolic code into object code which can be executed by the 8080 and 8085 microprocessors. Assembler output consists of three possible files: the object file containing your program translated into object code; the list file printout of your source code, the assembler generated object code, and the symbol table; and the symbol-cross-reference file, a listing of the symbol-cross-reference records.

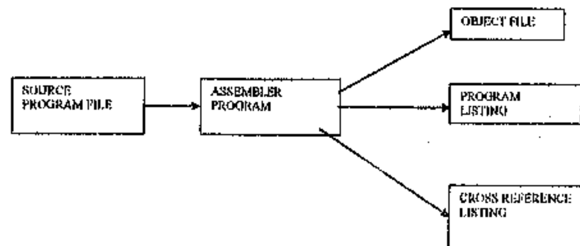


Figure 10.1 : Structure of Assembler

Object Code

For most microcomputer applications, you probably will eventually load the object program into some form of read only memory, However, do not forget that the Intellect development system is an 8080 microcomputer system with random access memory. In most cases you can load and execute your object program on the development system for testing and debugging. This allows you to test your program before your prototype application system is fully developed.

A special feature of this assembler is that it allows you to request object code in a relocatable format. This frees the programmer from worrying about the eventual mix of read only and random access memory in the application system; individual portions of the program can be relocated as needed when the application design is final. Also a large program can be broken into a number of separately assembled modules. Such modules are both easier to code and to teat.

Program Listing

The program Listing provides a permanent record of both the source program and the object code. The assembler also provides diagnostic messages for common programming errors in the program listing. For example, if you specify a 16-hi value for an instruction that can use only an 8-bit value, the assembler tells you that the value exceeds the permissible range.

Symbol-Cross-Reference Listing

The symbol-cross-reference listing is the diagnostic tools provided by the assembler. Assume, for example, that your program manipulates a data field named DATE, and that testing reveals a program logic error in the handling of this data. The symbol-cross-reference listing simplifies debugging this error because it points you to each instruction that references the symbol DATE.

10.2 Programming Model of 8085 Microprocessor

The 8085 programming model includes six registers, one accumulator, and one flag register, as shown in Figure. In addition, it has two 16-bit registers: the stack pointer and the program counter. They are described briefly as follows.

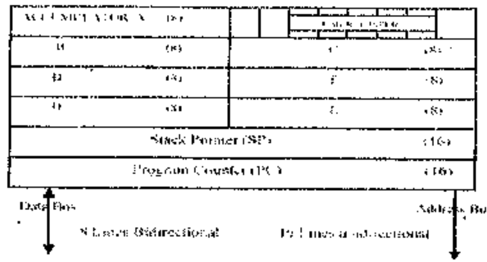


Figure 10.2: 8085 Programming Model

Registers

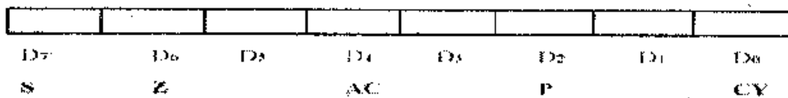
The 8085 has six general-purpose registers to store 8-bit data; these are identified as B,C,D,E,H, and L as shown in the figure. They can be combined as register pairs - BC, DE, and HL - to perform some 16-bit operations. The programmer can use these registers to store or copy data into the registers by using data copy instructions.

Accumulator

The accumulator is an 8-bit register that is a part of arithmetic/logic unit (ALU). This register is used to store 8-bit data and to perform arithmetic and logical operations. The result of an operation is stored in the accumulator. The accumulator is also identified as register A.

Flags

The ALU includes five flip-flops, which are set or reset after an operation according to data conditions of the result in the accumulator and other registers. They are called Zero(Z), Carry (CY), Sign (S), Parity (P), and Auxiliary Carry (AC) flags; their bit positions in the flag register are shown in the Figure below. The most commonly used flags are Zero, Carry, and Sign. The microprocessor uses these flags to test data conditions.



For example, after an addition of two numbers, if the sum in the accumulator id larger than eight bits, the flip-flop uses to indicate a carry — called the Carry flag (CY) — is set to one. When an arithmetic operation results in zero, the flip-flop called the Zero(Z) flag is set to one. The first Figure shows an 8-bit register, called the flag register, adjacent to the accumulator. However, it is not used as a register; five bit positions out of eight are used to store the outputs of the five flip-flops. The flags are stored in the 8-bit register so that the programmer can examine these flags (data conditions) by accessing the register through an instruction.

These flags have critical importance in the decision-making process of the microprocessor. The conditions (set or reset) of the flags are tested through the software instructions. For example, the instruction JC (Jump on Carry) is implemented to change the sequence of a program when CY flag is set. The thorough understanding of flag is essential in writing assembly language programs.

Program Counter (PC)

This 16-bit register deals with sequencing the execution of instructions. This register is a memory pointer. Memory locations have 16-bit addresses, and that is why this is a 16-bit register.

The microprocessor uses this register to sequence the execution of the instructions. The function of the program counter is to point to the memory address from which the next byte is to be fetched. When a byte (machine code) is being fetched, the program counter is incremented by one to point to the next memory

location

Stack Pointer (SP)

The stack pointer is also a 16-bit register used as a memory pointer. It points to a memory location in R/W memory, called the stack. The beginning of the stack is defined by loading 16-bit address in the stack pointer.

This programming model will be used in subsequent tutorials to examine how these registers are affected after the execution of an instruction.

Self Learning Exercises

1. What are the various flags used in 8085 ?
2. What is Stack Pointer?
3. In 8085 name the 16 bit registers?
4. What is Program counter?

10.3 Addressing Modes

The instructions MOV B, A or MVI A, 82H are to copy data from a source into a destination. In these instructions the source can be a register, an input port, or an 8-bit number (00H to FFH). Similarly, a destination can be a register or an output port. The sources and destination are operands. The various formats for specifying operands are called the ADDRESSING MODES. For 8085, they are:

1. Immediate addressing.
2. Register addressing.
3. Direct addressing.
4. Indirect addressing.

Immediate addressing

Data is present in the instruction. Load the immediate data to the destination provided. Example: MVI R,data

Register addressing

Data is provided through the registers.

Example: MOV Rd, Rs

Direct addressing

Used to accept data from outside devices to store in the accumulator or send the data stored in the accumulator to the outside device. Accept the data from the port 00H and store them into the accumulator or Send the data from the accumulator to the port

01H.

Example: IN 00H or OUT 01H

Indirect Addressing

This means that the Effective Address is calculated by the processor. And the contents of the address (and the one following) is used to form a second address. The second address is where the data is stored. Note that this requires several memory accesses; two accesses to retrieve the 16-bit address and a further access (or accesses) to retrieve the data which is to be loaded into the register.

10.4 Instruction Set Classification

An instruction is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions, called the instruction set, determines what functions the microprocessor can perform. These instructions can be classified into the following five functional categories: data transfer (copy) operations, arithmetic operations, logical operations, branching operations, and machine-control operations.

10.4.1 Data Transfer (Copy) Operations

This group of instructions copy data from a location called a source to another location called a destination, without modifying the contents of the source. In technical manuals, the term data transfer is used for this copying function. However, the term transfer is misleading; it creates the impression that the contents of the

source are destroyed when, in fact, the contents are retained without any modification. The various types of data transfer (copy) are listed below together with examples of each type:

Types	Examples
1. Between Registers	1. Copy the contents of the register B into register D.
2. Specific data byte to a register or a memory location.	2. Load register B with the data byte 32H.
3. Between a memory location and a register. B.	3. From a memory location 2000H to register
4. Between an I/O device and the accumulator.	4.From an input keyboard to the accumulator.

10.4.2 Arithmetic Operations

These instructions perform arithmetic operations such as addition, subtraction, increment, and decrement.

Addition - Any 8-bit number, or the contents of a register or the contents of a memory location can be added to the contents of the accumulator and the sum is stored in the accumulator. No two other 8-bit registers can be added directly (e.g., the contents of register B cannot be added directly to the contents of the register C). The instruction DAD is an exception; it adds 16-bit data directly in register pairs.

Subtraction - Any 8-bit number, or the contents of a register, or the contents of a memory location can be subtracted from the contents of the accumulator and the results stored in the accumulator. The subtraction is performed in 2's compliment, and the results if negative, are expressed in 2's complement. No two other registers can be subtracted directly.

Increment/Decrement - The 8-bit contents of a register or a memory location can be incremented or decrement by 1. Similarly, the 16-bit contents of a register pair (such as BC) can be incremented or decrement by 1. These increment and decrement operations differ from addition and subtraction in an important way; i.e., they can be performed in any one of the registers or in a memory location.

10.4.3 Logical Operations

These instructions perform various logical operations with the contents of the accumulator.

AND, OR Exclusive-OR - Any 8-bit number, or the contents of a register, or of a memory location can be logically ANDed, Ored, or Exclusive-ORed with the contents of the accumulator. The results are stored in the accumulator.

Rotate- Each bit in the accumulator can be shifted either left or right to the next position.

Compare- Any 8-bit number, or the contents of a register, or a memory location can be compared for equality, greater than, or less than, with the contents of the accumulator.

Complement - The contents of the accumulator can be complemented. All 0s are replaced by 1s and all 1s are replaced by 0s.

10.4.4 Branching Operations

This group of instructions alters the sequence of program execution either conditionally or unconditionally.

Jump - Conditional jumps are an important aspect of the decision-making process in the programming. These instructions test for a certain conditions (e.g., Zero or Carry flag) and alter the program sequence when the condition is met. In addition, the instruction set includes an instruction called unconditional jump.

Call, Return, and Restart - These instructions change the sequence of a program either by calling a subroutine

or returning from a subroutine. The conditional Call and Return instructions also can test condition flags.

10.4.5 Machine Control Operations

These instructions control machine functions such as Halt, Interrupt, or do nothing.

The microprocessor operations related to data manipulation can be summarized in four functions:

1. copying data
2. performing arithmetic operations
3. performing logical operations
4. testing for a given condition and alerting the program sequence

Some important aspects of the instruction set are noted below:

1. In data transfer, the contents of the source are not destroyed; only the contents of the destination are changed. The data copy instructions do not affect the flags.
2. Arithmetic and Logical operations are performed with the contents of the accumulator, and the results are stored in the accumulator (with some expectations). The flags are affected according to the results.
3. Any register including the memory can be used for increment and decrement.
4. A program sequence can be changed either conditionally or by testing for a given data condition.

10.5 Instruction Format

An instruction is a command to the microprocessor to perform a given task on a specified data. Each instruction has two parts: one is task to be performed, called the operation code (opcode), and the second is the data to be operated on, called the operand. The operand (or data) can be specified in various ways. It may include 8-bit (or 16-bit) data, an internal register, a memory location, or 8-bit (or 16-bit) address. In some instructions, the operand is implicit.

Instruction word size

The 8085 instruction set is classified into the following three groups according to word size:

1. One-word or 1-byte instructions
2. Two-word or 2-byte instructions
3. Three-word or 3-byte instructions

In the 8085, "byte" and "word" are synonymous because it is an 8-bit microprocessor.
However, instructions are commonly referred to in terms of bytes rather than words.

One-Byte Instructions

A 1-byte instruction includes the opcode and operand in the same byte. Operand(s) are internal register and are coded into the instruction.

For example:

Task	Op code	Operand	Binary Code	Hex Code
Copy the contents of the accumulator in the register C.	MOV	C,A	0100 1111	4FH
Add the contents of register B to the contents of the accumulator.	ADD	B	1000 0000	80H
Invert (compliment) each bit in the accumulator.	CMA		0010 1111	2FH

These instructions are 1-byte instructions performing three different tasks. In the first instruction, both operand registers are specified. In the second instruction, the operand B is specified and the accumulator is assumed. Similarly, in the third instruction, the accumulator is assumed to be the implicit operand. These instructions are stored in 8-bit binary format in memory; each requires one memory location. MOV rd, rs
rd <— rs copies contents of rs into rd.

Coded as 01 ddd sss where ddd is a code for one of the 7 general registers which is the destination of the data, sss is the code of the source register.

Example: MOV A,B

Coded as 01111000 = 78H = 170 octal (octal was used extensively in instruction design of such processors).

ADD r

A <— A + r

Two-Byte Instructions

In a two-byte instruction, the first byte specifies the operation code and the second byte specifies the operand. Source operand is a data byte immediately following the opcode. For example:

Task	Opcode	Operand	Binary Code	Hex Code	
Load an 8-bit data byte in the accumulator	MVI	A, Data	0011 1110	3E	First Byte
			DATA	Data	Second Byte

Assume that the data byte is 32H. The assembly language instruction is written as

Mnemonics	Hex code
MVI A, 32H	3E 32H

The instruction would require two memory locations to store in memory.

MVI r,data

r <— data

Example: MVI A,30H coded as 3EH 30H as two contiguous bytes. This is an example of immediate addressing.

ADI data

A <— A + data

OUT port

where port is an 8-bit device address. (Port) <— A. Since the byte is not the data but points directly to where it is located this is called direct addressing.

Three-Byte Instructions

In a three-byte instruction, the first byte specifies the opcode, and the following two bytes specify the 16-bit address. Note that the second byte is the low-order address and the third byte is the high-order address.

opcode + data byte + data byte

Task	Opcode	Operand	Binary code	Hex Code	
Transfer the program sequence to the memory location 2085H.	JMP	2085H	1100 0011	C3	First byte
			1000 0101	85	Second Byte
			0010 0000	20	Third Byte

This instruction would require three memory locations to store in memory.

Three byte instructions - opcode + data byte + data byte

LXI rp, data16

rp is one of the pairs of registers BC, DE, HL used as 16-bit registers. The two data bytes are 16-bit data in L H order of significance.

rp <— data16

Example: LXI H,0520H coded as 21H 20H 50H in three bytes. This is also immediate addressing.

LDA addr A <— (addr) Addr is a 16-bit address in L H order. Example: LDA 2134H coded as 3AH 34H

21H. This is also an example of direct addressing.
Lets see the use of the above defined instructions with the help of some programs

Example-1 Write an assembly program to add two numbers

```
MVI   D, 8BH
MVI   C, 6FH
MOV   A, C
ADD   D
OUT   PORT1
HLT
```

Example-2 Write an assembly program to find greatest between two numbers Program

```
MVI   B, 30H
MVI   C, 40H
MOV   A, B
CMP   C
OUT   PORT1
HLT   JZ EQU
JC    GRT
OUT   PORT1
HLT   EQU:   MVI A, 01H
OUT   PORT1
HLT
GRT: MOV   A, C
OUT    PORT1
HLT
```

10.6 Conditional Assembly

The IF, ELSE. and ENDIF directives enable you to assemble portions of your program conditionally, that is, only if certain conditions that you specify are satisfied.

Conditional assembly is especially useful when your application requires custom programs for a number of common options. As an example, assume that a basic control program requires customizing to accept input from one of six diffrent sensing devices and to drive one of five different control devices. Rather than code some thirty separate programs to account for all the possibilities, you can code a single program. The conditional directives must enclose the code for the individual sensors and drivers. When you need to generate a custom program, you can insert SET directives near the beginning of the source program to select the desired sensor and driver routine.

10.6.1 IF, ELSE, ENDIF Directives

Because these directives are used in conjunction, they are described together here

Label	*Opcode*	*Operand*
optional:	IF	expression
optional:	ELSE	---
optional:	ENDIF	---

The assembler evaluates the expression in the operand field of the IF directive. If bit 0 of the resulting value is one (TRUE), instructions between the IF directive and the next ELSE or ENDIF directive are assembled. When bit 0 is zero (FALSE) these instructions are ignored. (A TRUE expression evaluates to OFFFFH and FALSE to OH; only bit zero need be tested.) All statements included between an IF

directive and its required associated ENDIF directive are defined as an IF-ENDIF block. The ELSE directive is optional, and only one ELSE directive may appear in an IF-ENDIF block. When included, ELSE is the converse of IF. When bit 0 of the expression in the IF directive is zero, all statements between ELSE and the next ENDIF are assembled. If bit 0 is one, these statements are ignored.

Operands are not allowed with the ELSE and ENDIF directives. An IF-ENDIF block may appear within another IF-ENDIF block. These blocks can be nested to eight levels. Macro definitions may appear within an IF-ENDIF block. Conversely, IF-ENDIF blocks may appear within macro definitions. In either case, you must be certain to terminate the macro definition or IF-ENDIF block so that it can be assembled completely, For example, when a macro definition begins in an IF block but terminates after an ELSE directive, only a portion of the macro can be assembled. Similarly, an IF-ENDIF block begun within a macro definition must terminate within that same macro definition.

Example 1. Simple IF-ENDIF Block:

```
COND1:  IF TYPE EQ 0
        .
        .                   ;ASSEMBLED IF 'TYPE = 0'
        .                   ;IS TRUE
        .
        ENDIF
```

Example 2. IF-ELSE-ENDIF Block:

```
COND2:  IF TYPE EQ 0
        .                   ;ASSEMBLED IF 'TYPE = 0'
        .                   ;IS TRUE
        .
        ELSE
        .
        .                   ;ASSEMBLED IF 'TYPE = 0'
        .                   ;IS FALSE
        ENDIF
```

10.7 Subroutines

Suppose a program consists of several separate routines, any of which may be executed depending upon some initial condition (such as a number passed in a register). One way to code this would be to check each condition sequentially and branch to the routines accordingly as follows:

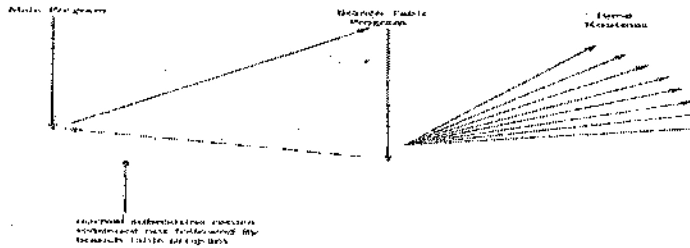
```
CONDITION = CONDITION 1?
IF YES BRANCH TO ROUTINE 1
CONDITION = CONDITION 2?
IF YES BRANCH TO ROUTINE 2
        .
        .
BRANCH TO ROUTINE N
```

A sequence as above is inefficient, and can be improved by using a branch table.

The logic at the beginning of the branch table program loads the starting address of the branch table into the H and L registers. The branch table itself consists of a list of starting addresses for the routines to be branched to. Using the H and L registers as a pointer, the branch table program loads the selected routine's starting address into the program counter, thus effecting a jump to the desired routine. For example, consider a program that executes one of eight routines depending on which bit of the accumulator is set:

Jump to routine 1 if the accumulator holds 00000001
" " " 2 " " " " 00000010
" " " 3 " " " " 00000100
" " " 4 " " " " 00001000
" " " 5 " " " " 00010000
" " " 6 " " " " 00100000
" " " 7 " " " " 01000000
" " " 8 " " " " 10000000

A program that provides such logic follows. The program is termed a 'pseudo-subroutine' because it is treated as a subroutine by the programmer (i.e., it appears just once in memory), but is entered via a regular JUMP instruction rather than via a CALL instruction.



```
Label     Code    Operand

START:    LXI     H,BTBL     ;REGISTERS H AND L WILL
                             ;POINT TO BRANCH TABLE
GTBIT:    RAR
          JC      GETAD
          INX     H          ;(H,L)=(H,L)+2 TO
          INX     H          ;POINT TO NEXT ADDRESS
                             ;IN BRANCH TABLE
          JMP     GTBIT
GETAD:    MOV     E,M        ;BIT FOUND
          INX     H          ;LOAD JUMP ADDRESS
                             ;INTO D AND E REGISTERS
          MOV     D,M
          XCHG               ;EXCHANGE D AND E
                             ;WITH H AND L
          PCHL               ;JUMP TO ROUTINE
                             ;ADDRESS
.
.
.
BTBL:     DW      ROUT1      ;BRANCH TABLE. EACH
          DW      ROUT2      ;ENTRY IS A TWO-BYTE
          DW      ROUT3      ;ADDRESS
          DW      ROUT4      ;HELD LEAST SIGNIFICANT
          DW      ROUT5      ;BYTE FIRST
          DW      ROUT6
          DW      ROUT7
          DW      ROUT8
```

The control routine at START uses the H and L registers as a pointer into the branch table (BTBL) corresponding to the bit of the accumulator that is set. The routine at GETAD then transfers the address held in the corresponding branch table entry to the H and L registers via the D and E registers, and then uses a PCHL instruction, thus transferring control to the selected routine.

10.7.1 TRANSFERRING DATA TO SUBROUTINES

A subroutine typically requires data to perform its operations. In the simplest case, this data may be transferred in one or more registers.

Sometimes it is more convenient and economical to let the subroutine load its own registers. One way to do this is to place a list of the required data (called a parameter list) in some data area of memory, and pass the address of this list to the subroutine in the H and L registers.

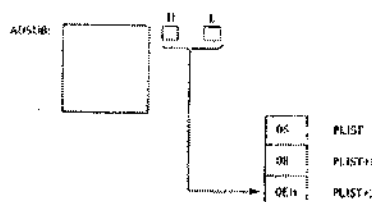
For example, the subroutine ADSUB expects the address of a three-byte parameter list in the H and L registers. It adds the first and second bytes of the list, and stores the result in the third byte of the list:

```
Label      Code    Operand    Comment

           LXI     H,PLIST    ;LOAD H AND L WITH
                              ;ADDRESSES OF THE PARAM-
                              ;ETER LIST
           CALL    ADSUB      ;CALL THE SUBROUTINE
RET1:
           .
PLIST:     DB      6          ;FIRST NUMBER TO BE ADDED
           DB      8          ;SECOND NUMBER TO BE
                              ;ADDED
           DS      1          ;RESULT WILL BE STORED HERE
           LXI     H,LIST2    ;LOAD H AND L REGISTERS
           CALL    ADSUB      ;FOR ANOTHER CALL TO ADSUB
RET2:
           .
LIST2:     DB      10
           DB      35
           DS      1
           .
ADSUB:     MOV     A,M        ;GET FIRST PARAMETER
           INX     H          ;INCREMENT MEMORY
                              ;ADDRESS
           MOV     B,M        ;GET SECOND PARAMETER
           ADD     B          ;ADD FIRST TO SECOND
           INX     H          ;INCREMENT MEMORY
                              ;ADDRESS
           MOV     M,A        ;STORE RESULT AT THIRD
                              ;PARAMETER STORE
           RET                ;RETURN UNCONDITIONALLY
```

The first time ADSUB is called, it loads the A and B registers from PLIST and PLIST+1 respectively, adds them, and stores tFe result in PLIST+2. Return is then made to the instruction at RET1.

First call to ADSUB:



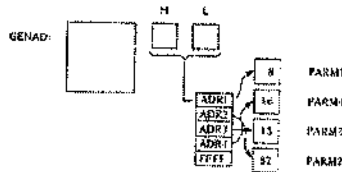
second time ADSUB is called, the H and L registers point to the parameter list LIST2. The A and B registers are loaded with 10 and 35 respectively, and the sum is stored at LIST2+2. Return is then made to the instruction at RET2.

Note that the parameter lists PLIST and LIST2 could appear anywhere in memory without altering the results produced by ADSUB.

This approach does have its limitations, however. As coded, ADSUB must receive a list of two and only two numbers to be added, and they must be contiguous in memory. Suppose we wanted a subroutine (GENAD) which would add an arbitrary number of bytes, located anywhere in memory, and leave the sum in the accumulator.

This can be done by passing the subroutine a parameter list which is a list of addresses of parameters, rather than the parameters themselves, and signifying the end of the parameter list be a number whose first

byte is FFH (assuming that no parameters will be stored above address FFOOH).
Call to GENAD:



As implemented below, GENAD saves the current sum (beginning with zero) in the C register. It then loads the address of the first parameter into the D and E registers. If this address is greater than or equal to FFOOH, it reloads the accumulator with the sum held in the C register and returns to the calling routine. Otherwise, it loads the parameter into the accumulator and adds the sum in the C register to the accumulator. The routine then loops back to pick up the remaining parameters.

```
Label     Code    Operand    Comment

          LXI     H,PLIST    ;LOAD ADDRESS OF
          CALL    GENAD      ;PARAMETER ADDRESS LIST
          .
          .
          HALT
PLIST:    DW      PARM1      ;LIST OF PARAMETER ADDRESSES
          DW      PARM2
          DW      PARM3
          DW      PARM4
          DW      OFFFFH     ;TERMINATOR


          PARM1:     DB      6
          PARM4:     DB      16
                     .
                     .
          PARM3:     DB      13
                     .
                     .
          PARM2:     DB      82


GENAD:    XRA     A          ;CLEAR ACCUMULATOR
LOOP:     MOV     C,A        ;SAVE CURRENT TOTAL IN C
          MOV     E,M        ;GET LOW ORDER ADDRESS BYTE
                             ;OF FIRST PARAMETER
          INX     H
          MOV     A,M        ;GET HIGH ORDER ADDRESS BYTE
                             ;OF FIRST PARAMETER
          CPI     0FFH       ;COMPARE TO FFH
          JZ      BACK       ;IF EQUAL, ROUTINE IS COMPLETE
          MOV     D,A        ;D AND E NOW ADDRESS PARAMETER
          LDAX    D          ;LOAD ACCUMULATOR WITH PARAMETER
          ADD     C          ;ADD PREVIOUS TOTAL
          INX     H          ;INCREMENT H AND L TO POINT
                             ;TO NEXT PARAMETER ADDRESS
          JMP     LOOP       ;GET NEXT PARAMETER
BACK:     MOV     A,C        ;ROUTINE DONE - RESTORE TOTAL
          RET                ;RETURN TO CALLING ROUTINE
          END
```

Note that GEND could add any combination of the parameters with no change to the parameters themselves. The sequence

```
        LXI     H,PLIST
        CALL    GENAD
        .
        .
PLIST:  DW      PARM4
        DW      PARM1
        DW      OFFFFH
```

would cause PARAM1 and PARAM4 to be added, no matter where in memory they might be located (excluding address above 0FFFFH).

Many variations of parameter passing are possible. For example, if it is necessary to allow parameters to be stored at any address, a calling program can pass the total number of parameters as the first parameter; the subroutine then loads this first parameter into a register and uses it as a counter to determine when all parameters had been accepted

10.8 MACROS

10.8.1 Why Use Macros?

A macro is essentially a facility for replacing one set of parameters with another. In developing your program, you will frequently find that many instruction sequences are repeated several times with only certain parameters changed. As an example, suppose that you code a routine that moves five bytes of data from one memory location to another. A little later, you find yourself coding another routine to move four bytes from a different source field to a different destination field. If the two routines use the same coding techniques, you will find that they are identical except for three parameters: the character count, the source field starting address, and the field starting address. Certainly it would be handy if there were some way to regenerate that original routine substituting the new parameters rather than rewrite that code yourself. The macro facility provides this capability and offers several other advantages over writing code repetitiously:

- The tedium of frequent rewrite (and the probability of error) is reduced.
- Symbols used in macros can be restricted so that they have meaning only within the macro itself. Therefore, as you code your program, you need not worry that you will accidentally duplicate a symbol used in a macro. Also, a macro can be used any number of times in the same program without duplicating any of its own symbols.
- An error detected in a macro need be corrected only once regardless of how many times the macro appears in the program. This reduces debugging time.
- Duplication of effort between programmers can be reduced. Useful functions can be collected in a library to allow macros to be copied into different programs.

In addition, macros can be used to improve program readability and to create structured programs. Using macros to segment code blocks provides clear program notation and simplifies tracing the flow of the program.

10.8.2 What Is A Macro?

A macro can be described as a routine defined in a formal sequence of prototype instructions that, when called within a program, results in the replacement of each such call with a code expansion consisting of the actual instructions represented.

The concepts of macro definition, call, and expansion can be illustrated by a typical business form letter, where the prototype instructions consist of preset text. For example, we could define a macro CNFIRM with the text

Air Flight welcomes you as a passenger.

Your Flight number FNO leaves at DTIME and arrives in DEST at ATIME.

This macro has four dummy parameters to be replaced, when the macro is called, by the actual flight number, departure time destination, and arrival time. Thus the macro call might look like

CNFIRM 123, '10:45', 'Ontario',11:52'

A second macro, CAR, could be called if the passenger has requested that a rental car be reserved at the destination airport. This macro might have the text

Your automobile reservation has been confirmed with MAKE rent-a-car agency,

Finally, a macro GREET could be defined to specify the passenger name.

Dear NAME:

The entire text of the business letter (source file) would then look like

GREET 'Ms. Scannel'

CNFIRM 123, '10:45', 'Ontario', '11:52'

CAR 'Blotz'

We trust you will enjoy your flight.

Sincerely,

When this source file is passed through a macro processor, the macro calls are expanded to produce the following letter.

Dear Ms. Scannel:

Air Flight welcomes you as a passenger. Your flight number 123 leaves at 10:45 and arrives in Ontario at 1 1 : 52. Your automobile reservation has been confirmed with Blotz rent-a-car agency.

We trust you will enjoy your flight.

Sincerely,

While this example illustrates the substitution of parameters in a macro, it overlooks the relationship of the macro processor and the assembler. The purpose of the macro processor is to generate source code which is then assembled.

10.8.3 Macros Vs. Subroutines

At this point, you may be wondering how macros differ from subroutines invoked by the CALL instruction. Both aid program structuring and reduce the coding of frequently executed routines. One distinction between the two is that subroutines necessarily branch to another part of your program while macros generate in-line code. Thus, a program contains only one version of a given subroutine, but contains as many versions of a given macro as there are calls for that macro.

Notice the emphasis on 'versions' in the previous sentence, for this is a major difference between macros and subroutines. A macro does not necessarily generate the same source code each time it is called. By changing the parameters in a macro call, you can change the source code the macro generates. In addition, macro parameters can be tested at assembly-time by the conditional assembly directives. These two tools enable a general-purpose macro definition to generate customized source code for a particular programming situation. Notice that macro expansion and any code customization occur at assembly-time and at the source code level. By contrast, a generalized subroutine resides in your program and requires execution time.

It is usually possible to obtain similar results using either a macro or a subroutine. Determining which of these facilities to use is not always an obvious decision. In some cases, using a single subroutine rather than multiple in-line macros can reduce the overall program size. In situations involving a large number of parameters, the use of macros may be more efficient. Also, notice that macros can call subroutines, and subroutines can contain macros.

10.8.4 Using Macros

The assembler recognizes the following macro operations:

- MACRO directive
- ENDM directive
- LOCAL directive
- REPT directive
- IRP directive

- IRPC directive
- EXITM directive
- Macro call

All of the directives listed above are related to macro definition. The macro call initiates the parameter substitution (macro expansion) process.

Macro Definition

Macros must be defined in your program before they can be used. A macro definition is initiated by the MACRO assembler directive, which lists the name by which the macro can later be called, and the dummy parameters to be replaced during macro expansion. The macro definition is terminated by the ENDM directive. The prototype instructions bounded by the MACRO and ENDM directives are called the macro body.

When label symbols used in a macro body have 'global' scope, multiply-defined symbol errors result if the macro is called more than once. A label can be given limited scope using the LOCAL directive. This directive assigns a unique value :o the symbol each time the macro is called and expanded. Dummy parameters also have limited scope. Occasionally you may wish to duplicate a block of code several times, either within a macro or in line with other source :ode. This can be accomplished with minimal coding effort using the REPT (repeat block), IRP (indefinite repeat), and IRPC (indefinite repeat character) directives. Like the MACRO directive, these directives are terminated by ENDM.

The EXITM directive provides an alternate exit from a macro. When encountered, it terminates the current macro just as if ENCM had been encountered.

Macro Definition Directives

MACRO Directive

Label	*Opcode*	*Operand*
name	MACRO	optional dummy parameter(s)

The name in the label field specifies the name of the macro body being defined. Any valid user-defined symbol name can be used as a macro name. Note that this name must be present and must not be terminated by a colon. dummy parameter can be any valid user-defined symbol name or can be null. When multiple parameters are listed, they must be separated by commas. The scope of a dummy parameter is limited to its specific macro definition. If a reserved symbol is used as a dummy parameter, its reserved value is not recognized. For example, if you code A.B,C as a dummy parameter list, substitutions will occur properly. However, you cannot use the accumulator or the B and registers within the macro. Because of the limited scope of dummy parameters, the use of these registers is not affected outside the macro definition. Dummy parameters in a comment are not recognized. No substitution occurs for such parameters.

Dummy parameters may appear in a character string. However, the dummy parameter must be adjacent to an ampersand character (&).

Any machine instruction or applicable assembler directive can be included in the macro body. The distinguishing feature of macro prototype text is that parts of it can be made variable by placing substitutable dummy parameters in instruction fields. These dummy parameters are the same as the symbols in the operand field of the MACRO directive.

Example:

Define macro MAC with dummy parameters G1, G2, and G3

```
MAC1    MACRO   G1,G2,G3   ;MACRO DIRECTIVE
MOVES:  LHLD    G1         ;MACRO BODY
        MOV     A,M
        LHLD    G2
        MOV     B,M
        LHLD    G3
        MOV     C,M
        ENDM               ;ENDM DIRECTIVE
```

ENDM Directive

Label	Opcode	Operand
—	ENDM	—

The ENDM directive is required to terminate a macro definition and follows the last prototype instruction. It is also required to terminate code repetition blocks defined by the REPT, IRP, and IRPC directives. Any data appearing in the label or operand fields of an ENDM directive causes an error

LOCAL Directive

Label	Opcode	Operand
—	LOCAL	label name(s)

The specified label names are defined to have meaning only within the current macro expansion. Each time the macro is called and expanded, the assembler assigns each local symbol a unique symbol in the form ??nnnn. The assembler assigns ??0001 to the first local symbol, ??0002 to the second, and so on. The most recent symbol name generated always indicates the total number of symbols created for all macro expansions. The assembler never duplicates these symbols. The user should avoid coding symbols in the form ??nnnn so that there will not be a conflict with these assembler-generated symbols.
Dummy parameters included in a macro call cannot be operands of a LOCAL directive. The scope of a dummy parameter is always local to its own macro definition. Local symbols can be defined only within a macro definition. Any number of LOCAL direct ves may appear in a macro definition, but they must all follow the macro call and must precede the first line of prototype code. A LOCAL directive appearing outside a macro definition causes an error. Also, a name appearing in the label field of a LOCAL directive causes an error.
Example:
The definition of MAC1 (used as an example in the description of the MACRO directive) contains a potential error because the symbol MOVES has not been declared local. This is a potential error since no error occurs if MAC] is called only once in the program, and the program itself does not use MOVES as a symbol. However, if MAC1 is called more than once, or if the program uses the symbol MOVES, MOVES is a multiply-defined symbol. This potential error is avoided by naming MOVES in the operand field of a LOCAL directive:

```
MAC1    MACRO   G1,G2,G3
        LOCAL   MOVES
MOVES:  LHLD    G1
        MOV     A,M
        LHLD    G2
        MOV     B,M
        LHLD    G3
        MOV     C,M
        ENDM
```

Assume that MAC1 is the only macro in the program and that it is called twice. The first time MAC1 is expanded, MOVES is replaced with the symbol ??0001; the second time, MOVES is replaced with ??0002. Because the assembler encounters only these special replacement symbols, the program may contain the symbol MOVES without causing a multiple definition.

REPT Directive

Label	Opcode	Operand
optional:	REPT	expression

The REPT directive causes a sequence of source code lines to be repeated 'expression' times. All lines appearing between the REPT directive and a subsequent ENDM directive constitute the block to be repeated. When 'expression' contains symbolic names, the assembler must encounter the definition of he symbol prior to encountering the expression. The insertion of repeat blocks is performed in-line when the assembler encounters the REPT directive. No explicit call is required to cause the code insertion since the definition is an implied call for expansion.

Example 1:

Rotate accumulator right six times.

```
ROTR6:  REPT   6
        RRC
        ENDM
```

Example 2:

The following REPT directive generates the source code for a routine that fills a five-byte field with the haracter stored in the accumulator:

PROGRAM CODE		GENERATED CODING	
LHLD	CNTR1	LHLD	CNTR1
REPT	5	MOV	M,A
MOV	M,A	INX	H
INX	H	MOV	M,A
ENDM		INX	H
		MOV	M,A
		INX	H
		MOV	M,A
		INX	H
		MOV	M,A
		INX	H

ample 3:

: following example illustrates the use of REPT to generate a multiplication routine. The multiplication is omplished through a series of shifts. If this technique is

amiliar, refer to the example of multiplication in Chapter 6. The example in Chapter 6 uses a program p for the multiplication. This example replaces the loop with seven repetitions of the four instructions losed by the REPT-ENDM directives. Notice that the expansion specified by this REPT directive ses the label SKIPAD to be generated seven times. Therefore, SKIPAD must be declared local to this ro

```
FSTMUL:   MVI     D,0      ;FAST MULTIPLY ROUTINE
          LXI     H,0      ;MULTIPLY E*A - 16-BIT RESULT
                           ;IN H&L
          REPT    7
          LOCAL   SKIPAD
          RLC              ;;GET NEXT MULTIPLIER BIT
          INC     SKIPAD   ;;DON'T ADD IF BIT = 0
          DAD     D        ;;ADD MULTIPLICAND INTO ANSWER
SKIPAD:   DAD     H
          ENDM
          RLC
          RNC
          DAD     D
          RET
```

This example illustrates a classic programming trade-off: speed versus memory

IRP Directive

Label	*Opcode*	*Operand*
optional:	IRP	dummy param, <list>

The operand field for the IRP (indefinite repeat) directive must contain one macro dummy parameter followed by a list of actual parameters enclosed in angle brackets. IRP expands its associated macro prototype code substituting the fist actual parameter for each occurrence of the dummy parameter. IRP then expands the proto type code again substituting the second actual parameter from the list. This process continues until the list is exhausted.

The list of actual parameters to be substituted for the dummy parameter must be enclosed in angle brackets (<>). Individual items in the list must be separated by commas. The number of actual parameters in the list controls the number of times the macro body is repeated; a list of n items causes n repetitions. An empty list (one with no) parameters coded) specifies a null operand list. IRP generates one copy of the macro body substituting a null for each occurrence of the dummy parameter. Also, two commas with no intervening character create a null parameter within the list.

Example:

The following code sequence gathers bytes of data from different areas of memory and then stores them in consecutive bytes beginning at the address of STORIT:

```
PROGRAM CODE                      GENERATED CODING

LXI   H,STORIT                    LXI   H,STORIT
IRP   X,<FLD1,3E20H,FLD3>         LDA   FLD1
LDA   X                           MOV   M,A
MOV   M,A                         INX   H
INX   H                           LDA   3E20H
ENDM                              MOV   M,A
                                  INX   H
                                  LDA   FLD3
                                  MOV   M,A
                                  INX   H
```

IRPC Directive

Label	*Opcode*	*Operand*
optional:	IRPC	dummy param,text

The IRPC (indefinite repeat character) directive causes a sequence of macro prototype instructions to be repeated for each text character of the actual parameter specified. If the text string is enclosed in optional angle brackets, any delimiters appearing in the text string are treated simply as text to be substituted into the prototype code.

The assembler generates one iteration of the prototype code for each character in the text string. For each iteration, the assembler substitutes the next character from the string for each occurrence of the dummy parameter. A list of n text characters generates n repetitions of the IRPC macro body, An empty string specifies a null actual operand. IRPC generates one copy of the macro body substituting a null for each occurrence of the dummy parameter.

Example:

```
          PROGRAM CODE          GENERATED CODING

          LHLD   DATE-1         LHLD    DATE-1
MVDATE:   IRPC   X,1977         INX     H
          INX    H              MVI     M,1
          MVI    M,X            INX     H
          ENDM                  MVI     M,9
                                INX     H
                                MVI     M,7
                                INX     H
                                MVI     M,7
```

IRPC provides the capability to treat each character of a string individually; concatenation provides the capability for building text strings from individual characters.

EXITM Directive

Label	*Opcode*	*Operand*
optional:	EXITM	—

EXITM provides an alternate method for terminating a macro expansion or the repetition of a REPT, IRP, or IRPC code sequence. When EXITM is encountered, the assembler ignores all macro prototype instructions located between the EXITM and ENDM directive for this macro. Notice that EXITM may be used in addition to ENDM, but not in place of ENDM. When used in nested macros, EXITM causes an exit to the previous level of macro expansion. An EXITM within a REPT, IRP, or IRPC terminates not only the current expansion, but all subsequent iterations as well. Any data appearing in the operand field of an EXITM directive causes an error.

Example:

EXITM is typically used to suppress unwanted macro expansion. In the following example, macro expansion is terminated when the EXITM directive is assembled because the condition X EQ 0 is true.

```
MAC3    MACRO   X,Y
          .
          .
        IF X EQ 0
        EXITM
          .
          .
        ENDM
```

MACRO CALLS

Once a macro has been defined, it can be called any number of times in the program. The call consists of the macro name and any actual parameters that are to replace dummy parameters during macro expansion. During assembly, each macro call is replaced by the macro definition code; dummy parameters are replaced by actual parameters.

Macro Call Format

Label	*Opcode*	*Operand*
optional:	macro name	optional actual parameter(s)

The assembler must encounter the macro definition before the first call for that macro. Otherwise, the macro call is assumed to be an illegal opcode. The assembler inserts the macro body identified by the macro name each time it encounters a call to a previously defined macro in your program.

The positioning of actual parameters in a macro call is critical since the substitution of parameters is based solely on position. The first-listed actual parameter replaces each occurrence of the first-listed dummy parameter; the second actual parameter replaces the second dummy parameter, and so on. When coding a macro call, you must be certain to list actual parameters in the appropriate sequence for the macro.

Notice that blanks are usually treated as delimiters. Therefore, when an actual parameter contains blanks (passing the instruction MOV A,M, for example) the parameter must be enclosed in angle brackets. This is also true for any other delimiter that is to be passed as part of an actual parameter. Carriage returns cannot be passed as actual parameters.

If a macro call specifies more actual parameters than are listed in the macro definition, the extra parameters are ignored. If fewer parameters appear in the call than in the definition, a null replaces each missing parameter.

Example:

The following example shows two calls for the macro LOAD. LOAD is defined as follows:

```
LOAD      MACRO     G1,G2,G3
          LOCAL     MOVES
MOVES:    LHLD      G1
          MOV       A,M
          LHLD      G2
          MOV       B,M
          LHLD      G3
          MOV       C,M
          ENDM
```

LOAD simply loads the accumulator with a byte of data from the location specified by the first actual parameter, the B register with a byte from the second parameter, and the C register with a byte from the third parameter.

The first time LOAD is called, it is used as part of a routine that inverts the order of three bytes in memory. The second time LOAD is called, it is part of a routine that adds the contents of the B register to the accumulator and then compares the result with the contents of the C register

```
MAIN PROGRAM                              SUBSTITUTION

JNZ NEXT                                  JNZ    NEXT
LOAD   FLD,FLD+1,FLD+2          ??0001:   LHLD   FLD
MOV    M,A      ;INVERT BYTES             MOV    A,M
DCX    H                                  LHLD   FLD+1
MOV    M,B                                MOV    B,M
DCX    H                                  LHLD   FLD+2
MOV    M,C                                MOV    C,M
LOAD   3E0H,BYTE,CHECK                    MOV    M,A      ;INVERT BYTES
ADD    B        ;CHECK DIGIT              DCX    H
CMP    C                                  MOV    M,B
CNZ    DGTBAD                             DCX    H
                                          MOV    M,C
                                ??0002:   LHLD   3E0H
                                          MOV    A,M
                                          LHLD   BYTE
                                          MOV    B,M
                                          LHLD   CHECK
                                          MOV    C,M
                                          ADD    B        ;CHECK DIGIT
                                          CMP    C
                                          CNZ    DGTBAD
```

Nested Macro Calls

Macro calls (including any combination of nested IRP, IRPC, and REPT constructs) can be nested within macro definitions up to eight levels. The macro being called need not be defined when the enclosing macro is defined; however, it mist be defined before the enclosing macro is called.

A macro definition can also contain nested calls to itself (recursive macro ca//s) up to eight levels, as long as the recursive macro expansions can be terminated eventually, This operation can be controlled using the conditional assembly directives described in Chapter 4 (IF, ELSE, ENDIF).

Example:

Have a macro call itself five times after it is called from elsewhere in the program.

```
PARAM1    SET      5
RECALL    MACRO
          .
          .
          .
          IF       PARAM1 NE 0
PARAM1    SET      PARAM1-1
          RECALL                  ;RECURSIVE CALL
          ENDIF
          .
          .
          ENDM
```

Macro Expansion

When a macro is called, the actual parameters to be substituted into the prototype code can be passed in one of two modes. Normally, the substitution of actual parameters for dummy parameters is simply a text substitution. The parameters are not evaluated until the macro is expanded.

If a percent sign (%) precedes the actual parameter in the macro call, however, the parameter is evaluated immediately, before expansion occurs, and is passed as a decimal number representing the value of the parameter. In the case of I RPC, a '%' preceding the actual parameter causes the entire text string to be treated as a single parameter. One IRPC iteration occurs for each digit in the decimal string passed as the result of immediate evaluation of the text string.

The normal mechanism for passing actual parameters is adequate for most applications. Using the percent sign to pre-evaluate parameters is necessary only when the value of the parameter is different within the local context of the macro definition as compared to its global value outside the macro definition.

Example:

The macro shown in this example generates a number of rotate instructions. The parameters passed in the

macro call determine the number of positions the accumulator is to be rotated and whether rotate right or rotate left instructions are to be generated. Some typical calls for this macro are as follows:

```
SHIFTR    'R',3
SHIFTR    L,%COUNT-1
```

The second call shows an expression used as a parameter. This expression is to be evaluated immediately rather than passed simply as text.
The definition of the SHIFTR macro is shown below. This macro uses the conditional IF directive to test the validity of the first parameter. Also, the REPT macro directive is nested within the SHIFTR macro.

```
SHIFTR  MACRO     X,Y
        IF X EQ 'R'
          REPT Y
            RAR
          ENDM
        ENDIF
        IF X NE 'L'
          EXITM
        ELSE
          REPT Y
            RAL
          ENDM
        ENDIF
        ENDM
```

The indentation shown in the definition of the SHIFTR macro graphically illustrates the relationships of the IF, ELSE, ENDIF directives and the REPT, ENDM directives. Such indentation is not required in your program, but may be desirable as documentation.
The SHIFTR macro generates nothing if the first parameter is neither R nor L. Therefore, the following calls produce no code. The result in the object program is as though the SHIFTR macro does not appear in the source program.

```
SHIFTR    5
SHIFTR    'B',2
```

The following call to the SHIFTR macro generates three RAR instructions

```
SHIFTR    'R',3
```

Assume that SET directive elsewhere in the source program has given COUNT the value 6. The following call generates five RAL instructions

```
SHIFTR    'L',%COUNT-1
```

The following is a redefinition of the SHIFTR macro. In this definition, notice that concatenation is used To form the RAF: or RAL operation code. If a call to the SHIFTR macro specifies a character other than R or L, illegal operation codes are generated. The assembler flags all illegal operation codes as errors

```
SHIFTR    MACRO    X,Y
          REPT     Y
          RA&X
          ENDM
          ENDM
```

NULL MACROS

A macro may legally comprise only the MACRO and ENDM directives. Thus, the following is a legal macro definition

```
NADA      MACRO     P1,P2,P3,P4
          ENDM
```

A call to this macro produces no source code and therefore has no effect on the program.

Although these is no reason to write such a macro, the null (or empty) macro body has a practical application. For example, all the macro prototype instructions might be enclosed with IF-ENDIF conditional directives. When none of the specified conditions is satisfied, all that remains of the macro is the MACRO directive and the ENDM directive

10.9 INTERRUPTS

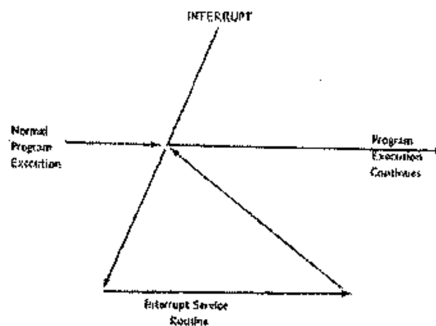
10.9.1 Interrupt Concepts

The following is a general description of interrupt handling and applies to both the 8080 and 8085 processors. However, the 8085 processor has some additional hardware features for interrupt handling.

Often, events occur external to the central processing unit which require immediate action by the CPU. For example, suppose a device is sending a string of 80 characters to the CPU, one at a time, at fixed intervals. There are two ways to handle such a situation:

A. A program could be written which accepts the first character, waits until the next character is ready (e.g., executes a timeout by incrementing a sufficiently large counter), then accepts the next character, and proceeds in this fashion until the entire 80 character string has been received. This method is referred to as programmed Input/Output.

B. The device controller could interrupt the CPU when a character is ready to be input, forcing a branch from the executing program to a special interrupt service routine. The interrupt sequence may be illustrated as follows:



1. The instruction currently being executed is completed.
2. The interrupt enable bit, INTE, is reset = 0.
3. The interrupting device supplies, via hardware, one instruction which the CPU executes. This instruction does not appear anywhere in memory, and the programmer has no control over it, since it is a function of the interrupting device's controller design. The program counter is not incremented before this instruction. The instruction supplied by the interrupting device is normally an RST instruction since this is an efficient one byte call to one of 8 eight-byte subroutines located in the first 64 words of memory. For instance, the device may supply the instruction:

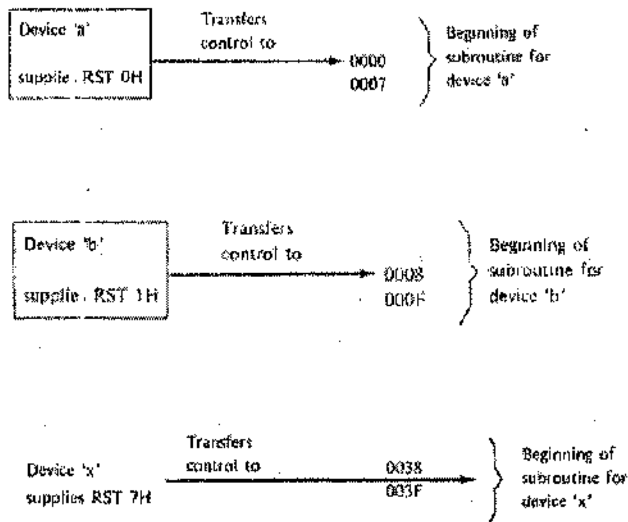
RST OH

with each input interrupt. Then the subroutine which processes data transmitted from the device to the CPU will be called into execution via an eight-byte instruction sequence at memory locations OOOOH to 0007H.

A digital input device may supply the instruction:

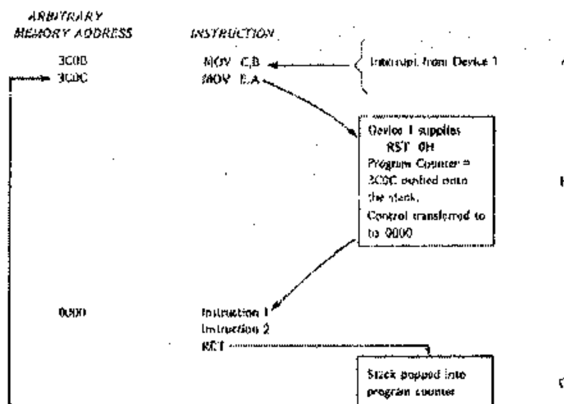
RST 1H

Then the subroutine that processes the digital input signals will be called via a sequence of instructions occupying memory locations 0008H to 000FH.



Note that any of these 8-byte subroutines may in turn call longer subroutines to process the interrupt, if necessary.

Any device may supply an RST instruction (and indeed may supply any one-byte 8080 instruction). The following is an example of an Interrupt sequence:



Device 1 signals an interrupt as the CPU is executing the instruction at 3COB. This instruction is completed. The program counter remains set to 3000, and the instruction RST OH supplied by device 1 is executed. Since this is a call to location zero, 3000 is pushed onto the stack and program control is transferred to location OOOOH. (This subroutine may perform jumps, calls, or any other operation.) When the RETURN is executed, address 3000 is popped off the stack and replaces the contents of the program counter, causing execution to continue at this point.

10.9.2 Writing Interrupt Subroutines

In general, and registers or condition bits changed by an interrupt subroutine must be restored before returning to the interrupted program, or errors will occur.

For example, suppose a program is interrupted just prior to the instruction:

JC LOC

and the carry)it equals 1, If the interrupt subroutine happens to reset the carry bit before returning to the interrupted pr)gram, the jump to LOC which should have occurred will not, causing the interrupted program to produce erroneous results.

Like any other subroutine then, any interrupt subroutine should save at least the condition bits and restore them before performing a RETURN operation. (The obvious and most convenient way to do this is to save

the data in the stack, using PUSH and POP operations.)

Further, the interrupt enable system is automatically disabled whenever an interrupt is acknowledged. Except in special cases, therefore, an interrupt subroutine should include an EI instruction somewhere to permit detection and handling the future interrupts. One instruction after an EI is executed, the interrupt subroutine may itself be interrupted. This process may continue to any level, but as long as all pertinent data are saved and restored, correct program execution will continue automatically

A typical interrupt subroutine, then, could appear as follows:

```
Code    Operand    Comment

PUSH    PSW        ;SAVE CONDITION BITS AND ACCUMULATOR
EI                 ;RE-ENABLE INTERRUPTS
.                  ;
.                  ;PERFORM NECESSARY ACTIONS TO SERVICE
.                  ;THE INTERRUPT
.
POP     PSW        ;RESTORE MACHINE STATUS
RET                ;RETURN TO INTERRUPTED PROGRAM
```

Self learning exercises

1. Define interrupt.
2. Which interrupt has the highest priority?
3. How many interrupts are there in 8085?

10.10 Summary

In this module we have discussed the assembly language of 8085 Microprocessor.

- An Assembler is a software tool or program , design to simplify the task of writing computer programs.
- The 8085 programming model includes six registers, one accumulator, and one flag register. In addition, it has two 16-bit registers: the stack pointer and the program counter.
- The various formats for specifying operands are called addressing modes. For 8085 they are : Immediate Addressing, Register addressing , Direct Addressing and Indirect Addressing.
- The instruction set of 8085 are classified into these categories: Arithmetic , Logical, Branching and Machine Control Operations.
- Describing different instruction including conditional assembly , Subroutines, Macros and Interrupts.

10.11 Glossary

Assembly Language : Assembly Language is a low level programming language using the human readable instructions of the CPU.

Addressing Mode : Addressing modes are an aspect of the instruction set architecture in most central processing unit (CPU) designs. ...

Instruction Set : The set of machine instructions that a particular CPU can execute; the corresponding set of assembly language mnemonics.

Macro : a single computer instruction that results in a series of instructions in machine language

Subroutine : A portion of a program that performs a specific task.

10.12 Further Readings

1. Ramesh S.Gaonkar, "Microprocessor Architecture, Programming and Applications with the 8085", 4th edition, Penram International Publishing (India) Pvt. Ltd., 1999.

2. Yu-cheng Liu and Glenn A.Gibson, "Microcomputer Systems: The 8086/8088 Family Architecture,

Programming & Design", 2nd Edition, Prentice Hall of India Pvt. Ltd., 2001.
3. Barry B.Brey, "The Intel Microprocessors – 8086/8088, 80186, 286, 386, 486, Pentium and Pentium Pro processor", Prentice Hall of India Pvt. Ltd., 1998.
4. Douglas V. Hall, "Microprocessors and Interfacing", Tata McGraw Hill, 1999.
5. Peter Abel, "IBM PC Assembly Language and Programming", Prentice Hall of India Private Limited, 1998.
6. B. Ram," Fundamentals of Microprocessors and Microcomputere",5th rev ed.2001,Dhanpat Rai, New Delhi
7. webphysics.davidson.edu/faculty/dmb/py310/8085.pdf

10.13 Answers to Self Learning Exercises

1. Sign flag, Zero flag, Auxiliary flag, Parity flag, Carry flag.
2. Stack pointer is a special purpose 16-bit register in the Microprocessor, which holds the address of the top of the stack.
3. Stack pointer and Program counter all have 16 bits
4. Program counter holds the address of either the first byte of the next instruction to be
5. fetched for execution or the address of the next byte of a multi byte instruction, which has not been completely fetched. In both the cases it gets incremented automatically one by one as the instruction bytes get fetched. Also Program register keeps the address of the next instruction.
6. External devices or internal abnormal conditions can interrupt the normal program execution of a microprocessor.
7. TRAP
8. 12

10.14 Unit End Questions

1. (a)Specify the contents of the registers and the flag status as the following instructions are executed.

   i. MVI A, 00H
   ii. MVI B, F8H
   iii. MOV C, A
   iv. MOV D, B
   v. HLT

   (b)Write instructions to load the hexadecimal number 65H in register C and 92H in accumulator A. Display the number 65H at PORT0 and 92H at PORT1.

2. Draw and explain the block diagram of a microprocessor 8085.
3. (a)Why the lower order address bus is multiplexed with data bus? How they will be de-multiplexed?
   (b) Differentiate between maskable and non-maskable interrupts.
4. Write an 8085 assembly language program using minimum number of instructions to add the 16 bit no. in BC, DE & HL. Store the 16 bit result in DE pair.
5. (a)Explain in detail the following instructions:-
   (i) ADC (ii) LHLD (iii) RLC (iv) DI
   (b) Define & explain the term addressing modes.

UNIT 11

Peripherals And Their Interfacing With 8085

Structure Of The Unit

11.0 Objective
11.1 Introduction
11.2 Address Space Partitioning
11.2.1 Memory Mapped I/O Scheme
11.2.2 I/O Mapped I/O Scheme
11.3 Memory and I/O Interfacing
11.3.1 Memory Interfacing
11.3.2 I/O Interfacing
11.4 Data Transfer Schemes
11.4.1 Synchronous data transfer
11.4.2 Asynchronous Data Transfer
11.4.3 Interrupt Driven Data Transfer
11.4.4 Multiple Interrupts
11.5 Interrupts
11.5.1 The 8085 Interrupts
11.6 Interfacing Devices and I/O Devices
11.6.1 Generation of Control Signals for Memory and I/O Devices
11.7 I/O Ports
11.8 Programmable DMA Controller
11.8.1 Intel 8257
11.9 Sample Program I/O Interfacing
11.10 Summary
11.11 Glossary
11.12 Further Readings
11.13 Answers to Self Learning Exercises
11.14 Unit End Questions

11.0 Objective

After studying this unit, you will learn

- Peripherals and their interfacing with 8085
- Memory Interfacing
- Interfacing I/O ports
- Data transfer schemes (Synchronous, asynchronous, interrupt driven),
- Architecture & interfacing of- DMA controller 8257.

11.1 Introduction

A microprocessor combined with memory and input/output devices, forms a microcomputer. The microprocessor is the heart of a microcomputer. Memories and input/output devices are interfaced to microprocessor to form a microcomputer. In case of large and minicomputers the memories and input/output devices are interfaced to CPU by the manufacturer. In a microprocessor-based system the designer has to select suitable memories and input/output devices for his task and interface them to the microprocessor. The selected memories and input/output devices should be compatible with microprocessor. If a particular device is not compatible, an additional electronic circuit has to be designed through which the device may be interfaced to the CPU.

11.2 Address Space Partitioning

The Intel 8085 uses a 16-bit wide address bus for addressing memories and I/O devices. Using 16-bit wide address bus it can access 2^{16} = 64 K bytes of memory and I/O devices. The 64 K addresses are to

be assigned to memories and I/O devices for their addressing. There are two schemes for the allocation of addresses to memories and input/output devices:

1. Memory mapped I/O scheme
2. I/O Mapped I/O scheme

11.2.1 Memory Mapped I/O Scheme

In memory mapped I/O scheme there is only one address space. Address space is defined as the set of all possible addresses that a microprocessor can generate. Some addresses are assigned to memories and some addresses to I/O devices. An I/O device is also treated as a memory location and one address is assigned to it. Suppose that memory locations are assigned the addresses 2000 to 24FF. One address is assigned to each memory location. Any one of these addresses cannot be assigned to an I/O device. The addresses for I/O devices are different from the addresses which have been assigned to memories. The addresses which have not been assigned to memories can be assigned to I/O devices. For example, 2500, 2501, 2502 etc. may be assigned to I/O devices. One address is assigned to each I/O device. In this scheme all the data transfer instructions of the microprocessor can be used for both memory as well as I/O devices. For example, MOV A, M will be valid for data transfer from the memory location or I/O device whose address is in H-L pair. If the H-L pair contains the address of a memory location, data will be transferred from the memory location to the accumulator. If the H-L pair contains the address of an I/O device, data will be moved from the I/O device to the accumulator. The memory mapped I/O scheme is suitable for a small system.

11.2.2 I/O Mapped I/O Scheme

In this scheme the addresses assigned to memory locations can also be assigned to I/O devices. Since the same address may be assigned to a memory location or an I/O device, the microprocessor must issue a signal to distinguish whether the address on the address bus is for a memory location or an I/O device. The Intel 8085 issues an IO/M -signal for this purpose. When this signal is high address on the address bus is for an I/O device. When this signal is low, the address on the ad bus is for a memory location. Two extra instructions IN and OUT are used to address I/O device The IN instruction is used to read to data of an input device. The OUT instruction is used to send to an output device. This scheme is suitable for a large system.

11.3 Memory and I/O Interfacing

Several memory chips and I/O devices are connected to a microprocessor. Fig.11.1 shows schematic diagram to interface memory chips or I/O devices to a microprocessor. An address decoding circuit is employed to select the required I/O device or a memory chip. Fig. 11.2 shows a schematic diagram of a decoding circuit. If IO/M is high the decoder 2 is activated and the required I/O device is selected. If IO/M is low, the decoder 1 is activated and the required memory chip is selected. A few MSBs of the address lines are applied to the decoder to select a memory chip or an I/O device.

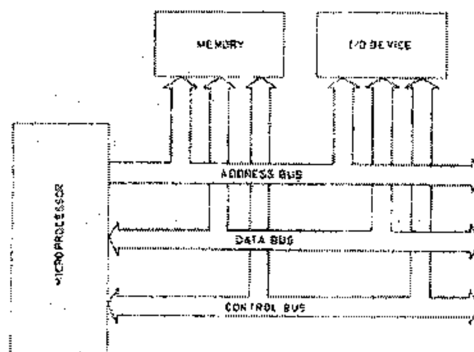


Figure 11.1 Schematic Diagram for Memory and I/O Interfacing

11.3.1 Memory Interfacing

The address of a memory location or an I/O device is sent out by the microprocessor. The corresponding memory chip or I/O device is selected by a decoding circuit. The decoding task can be performed by a decoder, a comparator, a bipolar PROM or PLA Programmed logic array).

In this section the application of 74LS138, a 1 to 8 lines decoder will be illustrated. Fig. 11.3 shows the interface of memory chips through 74LS138. G1, G2A and G2B are enable signals. To enable 74L5138,Gl should be high, and G2A and G2B should below. A, B and C are select lines. By applying proper logic to select lines any one of the outputs can be selected. Y0, Y1 Y7 are 8 output lines. An output line goes low when it is selected. Other output lines remain high. Table11.1 shows the truth table for 74LS138. When C1 is low or G2A is high or G2B is high, all output lines become high. Thus 74LS1 acts as decoder only when G1 is high, and G2A and G2B are low. The memory locations for EPROM 1 will lie in the range 0000 to 1FFF. These are the memory locations for ZONE 0 for the memory chip which is connected to the output line Y0 of the decoder. Similarly for ZONE 1 is 2000 to 3FFF and for ZONE 7 is E000 to FFFF. Table 11.2 shows the memory locations for various zones

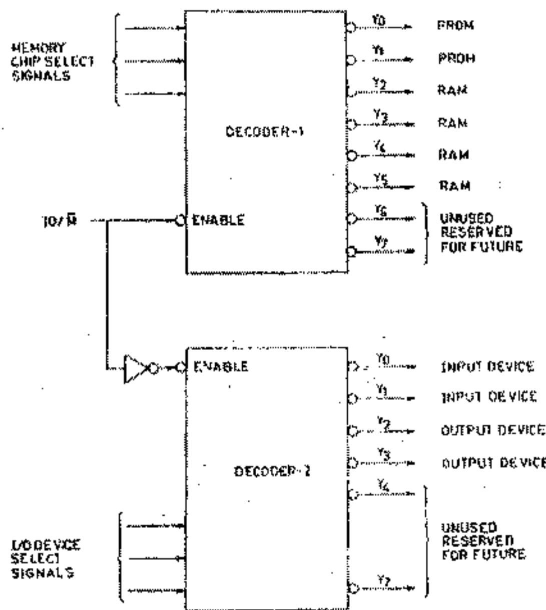


Figure 11.2 Interfacing of Memory and I/O Devices

INPUTS						OUTPUTS							
ENABLE			SELECT										
G1	G2A	G2B	C	B	A	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
X	H	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	L	H	H	H	H	H	L	H	H	H	H
H	L	L	H	L	L	H	H	H	H	L	H	H	H
H	L	L	H	L	H	H	H	H	H	H	L	H	H
H	L	L	H	H	L	H	H	H	H	H	H	L	H
H	L	L	H	H	H	H	H	H	H	H	H	H	L

X Denotes irrelevant

Table 1: Truth Table for 74LS138

Decoder Output	Memory Device	Zones of the Address Space	Memory Locations Address
Y_0	EPROM 1	ZONE 0	0000 to 1FFF
Y_1	EPROM 2	ZONE 1	2000 to 3FFF
Y_2	RAM 1	ZONE 2	4000 to 5FFF
Y_3	RAM 2	ZONE 3	6000 to 7FFF
Y_4	RAM 3	ZONE 4	8000 to 9FFF
Y_5	RAM 4	ZONE 5	A000 to BFFF
Y_6	RAM 5	ZONE 6	C000 to DFFF
Y_7	RAM 6	ZONE 7	E000 to FFFF

Table 2: Memory Locations for various Zones

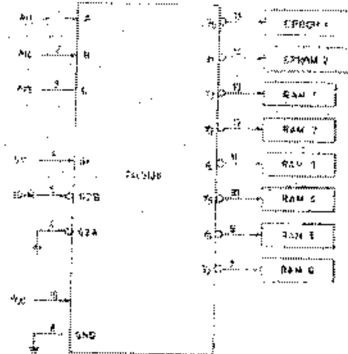


Figure 11.3: Interfacing Memory Chips using 74LS138

The entire memory address (64K for 8085) has been divided into 8 zones. Address lines A15, A14 and A13 have been applied to the select lines A, B and C of the 74LS138. The logic applied to these lines selects a particular memory device, an EPROM or a RAM. Other address lines A0, A1, A2 ...and A12 go directly to memory chip. They decide the address of the memory location within a selected memory chip. IO/M is connected to G2B. When IO/M goes low for memory read/write operation, G2B goes low. G1 is connected to + 5 $V_{d.c.}$ supply and G2A is grounded.

11.3.2 I/O Interfacing

Fig.11.4 shows the interface of I/O devices through decoder 74LS138. As the address of an I/O device is of 8 bits, only A8 — A15 lines of address bus are used for I/O addressing. The address lines A8, A9 and A10 have been applied to select lines A, B and C of the 74LS138. The address lines A11 — A15 are applied to G2B through a NAND gate. G2B becomes low only when all address lines A11 — A15 are high. G2A is grounded. IO/M is connected to G1. When IO/M goes high for I/O read/ write operation, Gl goes high. Table 11.3 shows the addresses of I/O devices connected to 74LS138.

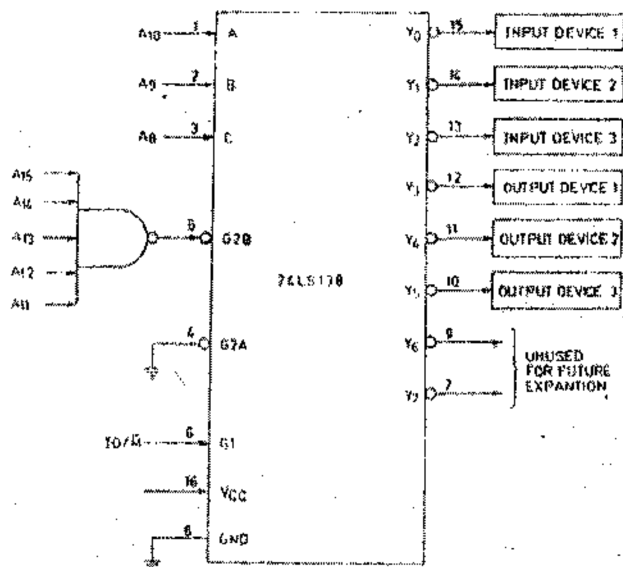


Figure 11.4 Interfacing of I/O Devices Using 74LS138

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	*Selected Output Lines*	*Corres-ponding Address*	*I/O Device*
1	1	1	1	1	0	0	0	Y_0	F8	Input Device 1
1	1	1	1	1	0	0	1	Y_1	F9	Input Device 2
1	1	1	1	1	0	1	0	Y_2	FA	Input Device 3
1	1	1	1	1	0	1	1	Y_3	FB	Output Device 1
1	1	1	1	1	1	0	0	Y_4	FC	Output Device 2
1	1	1	1	1	1	0	1	Y_5	FD	Output Device 3
1	1	1	1	1	1	1	0	Y_6	FE	Unused
1	1	1	1	1	1	1	1	Y_7	FF	Unused

Table 3: Address of I/O Devices connected to 74LS138

11.4 Data Transfer Scheme

In a microprocessor-based system or in a computer data transfer takes place between two devices such as microprocessor and memory, microprocessor and I/O devices, and memory and I/O device. Usually, semiconductor memories are compatible with microprocessor because the same technology is employed in the manufacturing of both semiconductor memories and microprocessors. Hence, there is less problem associated with the interfacing of memory. A wide variety of I/O devices having wide range of speed and other different characteristics are available. They use different manufacturing technology such as electronic, electrical, mechanical, electro-mechanical optical etc. Due to these reasons designers face difficulties in interfacing I/O devices with microprocessor. Special interfacing circuitries have to be designed for the purpose. A microprocessor-based system or a computer may have several I/O devices of different speed. A slow I/O device can not transfer data when microprocessor issues instruction for the same because it takes some time to get ready. To solve the problem of speed mismatch between a microprocessor and I/O devices a number of data transfer techniques have been developed. The data transfer schemes are classified into the following two broad categories.

1. Programmed data transfer schemes

2. DMA (Direct Memory Access) data transfer scheme.

Programmed Data Transfer Schemes

Programmed data transfer schemes are controlled by the CPU. Data are transferred from an I/O. device to the CPU (or to the memory through the CPU) or vice versa under the control of programs which reside in the memory. These programs are executed by the CPU when an I/O device is ready to transfer data. The microprocessor executes the program to transfer data. Programmed data transfer schemes are employed when small amount of data are to be transferred. The programmed data transfer schemes are classified into the following three categories.

(i) Synchronous data transfer scheme

(ii) Asynchronous data transfer scheme

(iii) Interrupt driven data transfer scheme

These schemes will be discussed in subsequent sections.

DMA Data Transfer Scheme

In DMA data transfer scheme CPU does not participate. Data are directly transferred from an I/O device to the memory or vice versa. The data transfer is controlled by the I/O device or a DMA controller. This scheme is employed when large amount of data are to be transferred. If bulk data are transferred through the CPU, it takes appreciable time and the process becomes slow. An I/O device which wants to send data using DMA technique, sends the HOLD signal to the CPU. On receiving a HOLD signal from an I/O device the CPU gives up the control of buses as soon as the current machine cycle is completed. The CPU sends a hold acknowledge signal to the I/O device to indicate that it has received the HOLD request and it has released the buses. The I/O device takes over the control of buses and directly transfers data to the memory or reads data from the memory.

DMA data transfer scheme is a faster scheme as compared to programmed data transfer scheme. It is used to transfer data from mass storage devices such as hard disks, floppy disks etc. It is also used for high-speed printers. When data transfer is over, the CPU regains the control over the buses.

DMA data transfer schemes are of the following two types:

(i) Burst mode of DMA data transfer

(ii) Cycle stealing techniques of DMA data transfer.

Burst Mode of DMA Data Transfer. A scheme of DMA data transfer, in which the I/O device withdraws the DMA request only after all the data bytes have been transferred, is called burst mode of data transfer. By this technique a block of data is transferred. This technique is employed by magnetic disks drives. In case of magnetic disks data transfer can not be stopped or slowed down without loss of data. Hence, block transfer is a must.

Cycle Stealing Technique. In this technique a long block of data is transferred by a sequence of DMA cycles. In this method after transferring one byte or several bytes the I/O device withdraws DMA request. This method reduces interference in CPU's activities. The interference can be eliminated completely by designing an interfacing circuitry which can steal bus cycle for DMA data transfer only when the CPU is not using the system bus.

In DMA data transfer schemes I/O devices control data transfer and hence the I/O devices must have registers to store memory addresses and byte count. It must also have electronic circuitry to generate necessary control signals required for DMA operations. Usually, I/O devices do not have these facilities. To solve this problem DMA controllers have been designed and developed. Examples of DMA controller chips are : Intel 8237A, 8257 etc. which will be discussed in subsequent sections.

11.4.1 Synchronous data transfer

Synchronous means "at the same time." The device which sends data and the device which receives data are synchronized with the same clock. When the CPU and I/O devices match in speed, this technique of the data transfer is employed. The data transfer with I/O devices is
performed executing IN or OUT instructions for I/O mapped I/O devices or using memory read/write instructions for memory mapped I/O devices. The IN instruction is used to read data from an input device

or input port. The OUT instruction is used to send data from the CPU to an output device or output port. As the CPU and the I/O device match in speed, the I/O device is ready to transfer data when IN or OUT instruction is issued by the CPU. The status of the I/O device *i.e.*, whether it is ready or not, is not examined before data are transferred, as it is not needed.

The I/O devices compatible with microprocessors in speed are usually not available. Hence, this technique of data transfer is rarely used for I/O devices. However, memories compatible with microprocessors are available, and therefore, this technique is invariably used with compatible memory devices.

11.4.2 Asynchronous Data Transfer

Asynchronous means "at irregular intervals". In this method data transfer is not based on per- determined timing pattern. This technique of data transfer is used when the speed of an I/O device does not match the speed of the microprocessor, and the timing characteristic of I/O device is not predictable. In this technique the status of the I/O device *i.e.* whether the device is ready or not, is checked by the microprocessor before the data are transferred. The microprocessor initiates the I/O device to get ready and then continuously checks the status of the I/O device till the I/O device becomes ready to transfer data. When I/O device becomes ready the microprocessor sends instruction to transfer data. This mode of data transfer is called *handshaking mode* of data transfer because some signals are exchanged between the I/O device and microprocessor before the actual data transfer takes place. The microprocessor issues an initiating signal to the I/O device to get ready (or to start). When I/O device becomes ready it sends signals to the processor to indicate that it is ready. Such signals are called *handshake signals.*

Fig. 11.5 shows a schematic diagram for asynchronous data transfer. Asynchronous data transfer is used for slow I/O devices. This technique is an inefficient technique because the precisious

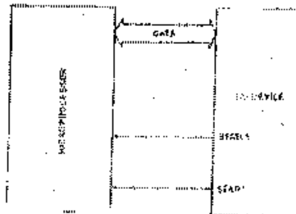


Figure 11.5 Asynchronous Data Transfer

time of the microprocessor is wasted in waiting. Fig; 11.6. shows a simple example of asynchronous data transfer. An A/D converter has been interfaced to the microprocessor to transfer data in asynchronous mode. The microprocessor sends a start of conversion signal, S/C to the A/D

converter. The A/D converter being a slow device as compared to a microprocessor, takes some time to convert analog signal to digital signal. When conversion is over the A/D converter makes end of conversion signal, E/C high. The microprocessor goes on checking E/C till it becomes high. When F/C becomes high, the microprocessor issues instructions for data transfer. Some simple I/O devices may not have status signal. In such a case the microprocessor goes on checking whether data are available on the port or not. A keyboard interfaced to a. microprocessor through a port is an example of this type of data transfer scheme. Asynchronous data transfer discussed so far use software approach. It can also be implemented by hardware approach employing READY signal. An I/O device is interfaced to the microprocessor through READY signal. I/O device (or memory chip) and microprocessor have READY pins. When an I/O device or memory chip becomes ready to transfer data, it makes READY signal high. 11 microprocessor checks READY signal before data are transferred. If READY is low the microprocessor enters a wait state. The status of the READY signal is sensed by the microprocessor in the T2 state of a machine cycle. The microprocessor remains in wait state till READY becomes high. This technique is commonly used by slow memory devices.

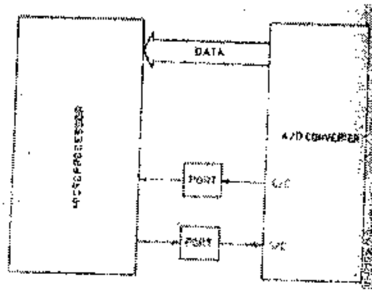


Fig. 11.6 Asynchronous Data Transfer Scheme for an A/D Converter

11.4.3 Interrupt Driven Data Transfer

In this scheme the microprocessor initiates an I/O device to get ready, and then it executes its main program instead of remaining in a program loop to check the status of the I/O device. When the I/O device becomes ready to transfer data, it sends a high signal to the microprocessor through a special input line called an interrupt line. In other words it interrupts the normal processing sequence of the microprocessor. On receiving an interrupt the microprocessor completes the current instruction at hand, and then attends the I/O device. It saves the contents of the program counter on the stack first, and then takes up a subroutine called ISS (Interrupt Service Subroutine). It executes ISS to transfer data from or to the I/O device. Different ISS are to be provided for different I/O devices. After completing the data transfer the microprocessor returns back to the main program which it was executing before the interrupt occurred. Interrupt driven data transfer is used for slow I/O devices.

It is an efficient technique as compared to asynchronous data transfer scheme because precious time of the microprocessor is not wasted in waiting while an I/O device is getting ready. Fig. 11.7 show the interfacing of an A/D converter to transfer data employing interrupt drive data transfer scheme. The microprocessor sends first the start of conversion signal, S/C to the A/D converter. Thereafter, the microprocessor executes its main program. A/D converter is a slow device compared to a microprocessor. It takes some time to convert analog signal to its equivalent digit quantity When A/D converter completes the task of conversion, it makes an end of conversion signal, E/C high. The E/C signal is connected to an interrupt line of the microprocessor. When interrupt line goes high, the microprocessor takes all necessary steps to transfer data from the A/D converter. After completing the data transfer the microprocessor returns back to execute the main program that it was executing prior to the interrupt.

11.4.4 Multiple Interrupts

In a microcomputer system several I/O devices can use interrupt driven data transfer scheme. While interfacing 1/0 devices using in- MIcR0PR0cE SSOR AID CONVERTER interrupts the following situations may arise:

1. A microprocessor may have only one interrupt level and several I/O devices are to be connected to it.
2. A microprocessor may have several interrupt levels and one I/O device is to be connected to each interrupt level.

The schemes which are used to tackle such situations are described in the following subsections.

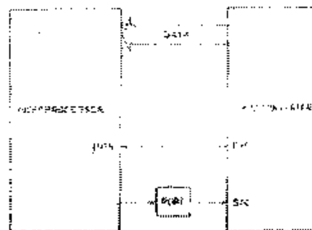


FIg. 11.7 Interrupt Driven Data Transfer Scheme for an A/D converter

Several I/O Devices Connected to a Single Interrupt Level When several I/O devices are to be connected

to a single interrupt level, they are connected through interrupt controller, 8259. Up to 8I/Odevices can be connected to the microprocessor through an8259. If more than 8 I/O devices are to be connected, more 8259 ICS are used in series.

One Device Connected to Each Level of Interrupt When a microprocessor has several interrupt levels and the number of I/O devices is equal to or less than the interrupt levels one device may be connected to each level of interrupt, in this scheme when a device interrupts the microprocessor, the microprocessor immediately knows which device has interrupted. The processor automatically transfers its program to a specific memory location that has been assigned

to the interrupt level. It executes ISS for the device which has interrupted. Such an interrupt scheme is known as *vectored interrupt.*

11.5 Interrupts

Interrupt is a process where an external device can get the attention of the microprocessor.

- The process starts from the I/O device
- The process is asynchronous.

Classification of Interrupts

Interrupts can be classified into two types:

- Maskable Interrupts (Can be delayed or Rejected)
- Non-Maskable Interrupts (Can not be delayed or Rejected)

Interrupts can also be classified into:

- Vectored (the address of the service routine is hard-wired)
- Non-vectored (the address of the service routine needs to be supplied externally by the device)

An interrupt is considered to be an emergency signal that may be serviced.

The Microprocessor may respond to it as soon as possible.

What happens when MP is interrupted ?

When the Microprocessor receives an interrupt signal, it suspends the currently executing program and jumps to an Interrupt Service Routine (ISR) to respond to the incoming interrupt.

Each interrupt will most probably have its own ISR.

RESPONDING TO INTERRUPTS

Responding to an interrupt may be immediate or delayed depending on whether the interrupt is maskable or non-maskable and whether interrupts are being masked or not.

There are two ways of redirecting the execution to the ISR depending on whether the interrupt is vectored or non-vectored.

- Vectored: The address of the subroutine is already known to the Microprocessor
- Non Vectored: The device will have to supply the address of the subroutine to the Microprocessor

11.5.1 The 8085 Interrupts

- When a device interrupts, it actually wants the MP to give a service which is equivalent to asking the MP to call a subroutine. This subroutine is called ISR (Interrupt Service Routine)
- The 'EI' instruction is a one byte instruction and is used to Enable the non-maskable interrupts.
- The 'DI' instruction is a one byte instruction and is used to Disable the non-maskable interrupts.
- The 8085 has a single Non-Maskable interrupt.
- The non-maskable interrupt is not affected by the value of the Interrupt Enable flip flop.
- The 8085 has 5 interrupt inputs.
- The INTR input.
- The INTR input is the only non-vectored interrupt.
- INTR is maskable using the EI/DI instruction pair.
- RST 5.5, RST 6.5, RST 7.5 are all automatically vectored.
- RST 5.5, RST 6.5, and RST 7.5 are all maskable.

- TRAP is the only non-maskable interrupt in the 8085
- TRAP is also automatically vectored

Interrupt name	Maskable	Vectored
INTR	Yes	No
RST 5.5	Yes	Yes
RST 6.5	Yes	Yes
RST 7.5	Yes	Yes
TRAP	No	Yes

INTERRUPT VECTORS AND THE VECTOR TABLE

An interrupt vector is a pointer to where the ISR is stored in memory.
All interrupts (vectored or otherwise) are mapped onto a memory area called the Interrupt Vector Table (IVT).

- The IVT is usually located in memory page 00 (0000H - 00FFH).
- The purpose of the IVT is to hold the vectors that redirect the microprocessor to the right place when an interrupt arrives.

Example: Let, a device interrupts the Microprocessor using the RST 7.5 interrupt line.

- Because the RST 7.5 interrupt is vectored, Microprocessor knows, in which memory location it has to go using a call instruction to get the ISR address. RST7.5 is knows as Call 003Ch to Microprocessor. Microprocessor goes to 003C location and will get a JMP instruction to the actual ISR address. The Microprocessor will then, jump to the ISR location

THE 8085 NON-VECTORED INTERRUPT PROCESS

1. The interrupt process should be enabled using the EI instruction.
2. The 8085 checks for an interrupt during the execution of every instruction.
3. If INTR is high, MP completes current instruction, disables the interrupt and sends INTA (Interrupt acknowledge) signal to the device that interrupted
4. INTA allows the I/O device to send a RST instruction through data bus.
5. Upon receiving the INTA signal, MP saves the memory location of the next instruction on the stack and the program is transferred to 'call' location (ISR Call) specified by the RST instruction
6. Microprocessor Performs the ISR
7. ISR must include the 'EI' instruction to enable the further interrupt within the program.
8. RET instruction at the end of the ISR allows the MP to retrieve the return address from the stack and the program is transferred back to where the program was interrupted.
9. The 8085 recognizes 8 RESTART instructions: RST0 - RST7. each of these would send the execution to a predetermined hard-wired memory location:

ESTART SEQUENCE

The restart sequence is made up of three machine cycles
In the 1st machine cycle:
The microprocessor sends the INTA signal.
While INTA is active the microprocessor reads the data lines expecting to receive, from the interrupting device, the opcode for the specific RST instruction.
In the 2nd and 3rd machine cycles:
the 16-bit address of the next instruction is saved on the stack.
Then the microprocessor jumps to the address associated with the specified RST instruction.

THE 8085 MASKABLE/VECTORED INTERRUPTS

The 8085 has 4 Masked/Vectored interrupt inputs.
RST 5.5, RST 6.5, RST 7.5
- They are all maskable.

They are automatically vectored according to the following table:

The vectors for these interrupt fall in between the vectors for the RST instructions. That's why they have names like RST 5.5 (RST 5 and a half).

MASKING RST 5.5, RST 6.5 AND RST 7.5

These three interrupts are masked at two levels:

Through the Interrupt Enable flip flop and the EI/DI instructions.

The Interrupt Enable flip flop controls the whole maskable interrupt process.

Through individual mask flip flops that control the availability of the individual interrupts.

These flip flops control the interrupts individually.

THE 8085 MASKABLE/VECTORED INTERRUPT PROCESS

1.The interrupt process should be enabled using the EI instruction.

2.The 8085 checks for an interrupt during the execution of every instruction.

3.If there is an interrupt, and if the interrupt is enabled using the interrupt mask, the microprocessor will complete the executing instruction, and reset the interrupt flip flop.

4.The microprocessor then executes a call instruction that sends the execution to the appropriate location in the interrupt vector table.

5.When the microprocessor executes the call instruction, it saves the address of the next instruction on the stack.

6.The microprocessor jumps to the specific service routine.

7.The service routine must include the instruction EI to re-enable the interrupt process.

8.At the end of the service routine, the RET instruction returns the execution to where the program was interrupted.

MANIPULATING THE MASKS

The Interrupt Enable flip flop is manipulated using the EI/DI instructions.

The individual masks for RST 5:5, RST 6.5 and RST 7.5 are manipulated using the SIM instruction.

This instruction takes the bit pattern in the Accumulator and applies it to the interrupt mask enabling and disabling the specific interrupts.

Self Learning Exercises

1. Can an input port and an output port have the same port address?

2. How will the port number be affected if we decode the high order address lines A_{15}-A_8 rather than A_7–A_0.

3. If high order lines are partially decoded, how can one determine whether it is peripheral I/O or memory mapped I/O.

4. In a Memory mapped I/O how does the microprocessor differentiate between an I/O and memory? Can an I/O have the same address as a memory register?

5. Why is a 16 bit address (data) stored in memory in the reverse order- the low order byte first, followed by the high order byte?

11.6 Interfacing Devices and I/O Devices

To communicate with the outside world microcomputers use peripherals (I/O devices). Commonly used peripherals are : A/D converter, D/A converter, CRT, printers, hard disks, floppy disks, magnetic tapes etc. Peripherals are connected to the microcomputer through electronic circuits known as interfacing circuits. Generally, each I/O device requires a separate interfacing circuit. The interfacing circuit converts the data available from an input device into compatible format for the computer. [he interface associated with the output device converts the output of the microcomputer Into the desired peripheral format. To simplify the work of the designer microprocessor

manufacturers have developed a member of general purpose and special purpose single chip interfacing devices. Some of the general purpose devices are:

(i) I/O Port

(ii) Programmable Peripheral Interface (PPI)

(iii) DMA Controller
(iv) Interrupt Controller
(v) Communication Interface
Special purpose interfacing devices are designed to interface a particular type of I/O device t the microprocessor. Examples of such devices are:
(i) CRT Controller.
(ii) Floppy Disk Controller.
(iii) Key Board and Display Interface.
A large number of interfacing devices have been developed by various manufacturers. I'lw detailed description of these devices is beyond the scope of the book. Here we will discuss some important Memory and I/O devices.

11.6.1 Generation of Control Signals for Memory and I/O Devices

Intel 8085 issues control signals RD, WR for read and write operation of memory and I/O devices. It also issues a status signal I0/M to distinguish whether read/write operation is -
to be performed by memory or I/O device. Memory and I/O devices require control signals in modified form shown below:
MEMR - Memory read.
MEMW - Memory write.
IOR—I/Oread.
lOW— 1/0 write
— These control signals are generated using RD, WR and IO/M using logic gates. OR
logic and inverters are used for the purpose as shown in Fig. 11.11.

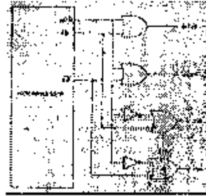


Figure 11.11 Control Signals for Memory I/O Read/Write Operation

To get MEMR, use 10/M v RD

v is a symbol for logical OR operation. Memory read operation takes place when I0/M and RD both are low. IO/M and RD are applied to an OR gate. The output of the OR
gate is MEMR. When both I0/M and RD are low and MEMR goes low and it activates memory for read operation. Similarly, other control signals are obtained as shown below:
To get MEMW, use IO/M v WR. —
I/O read/write operation takes place when IO/M is high. To get IOR and I0W signals
IO/M is inverted and then applied to OR gate
Get IOR using (inverted I0/M v RD)
Get lOW using (inverted I0/M v WR)

11.7 I/O Ports

An input device is connected to the microprocessor through an input t port. An input port i*s a* place for uploading data. Au input device unloads data into the port. The microprocessor reads data from the input port. Thus data are transferred from the input device to the accumulator via input port. Similarly, an output device is connected to the microprocessor through an output port. The microprocessor unloads data into an output port. As output port is connected to the output device, Figure 11.12 shows a schematic connection of the CPU, I/O port and I/O devices. An I/O port may be programmable or non-programmable. A non-programmable port behaves as an input port if it has been designed and connected in input mode. .Similarly, a port connected in output mode acts as an output port. But a programmable I/O port can be programmed

to act either as an input port or output port; the electrical connections remain same.

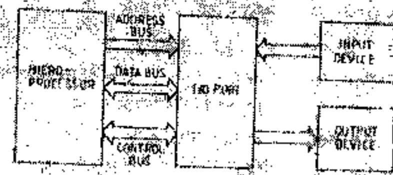


Figure 11.12 Interfacing of I/O device through I/O port

The Intel 8212 is an 8-bit non-programmable I/O port. It can be connected to the microprocessor either as an input port or an output port. If we require one input port and one output port, two units of 8212 will be required. One of them will be connected in input mode and the other in the output mode. Figure 11.13 shows the connections of 8212 in input mode and output mode.

Figure 11.13 Interfacing of Intel 8212

The Intel 8155 is a RAM with I/O ports. It contains a 256 byte RAM, 3 I/O ports and a 14-bit timer/counter. There are three ports A, B and C. The port A and port B are of 8-bits and port C of 6 bits Each port can be programmed either as an input port or output port. The port C may also be programmed as a control port for the port A and port B.

11.8 Programmable DMA Controller

The bulk data transfer from fast I/O devices to the memory or from the memory to I/O devices through the accumulator is a time consuming process. For such a situation the direct memory access (DMA) technique is preferred. In DMA data transfer scheme, data are directly transferred from an I/O device to RAM or from RAM to an I/O device. For DMA data transfer, the data and address buses come under the control of the peripheral device which wants DMA data transfer. The microprocessor has to relinquish the control of the address and data buses for DMA operation on the request of the I/O device. For DMA data transfer the I/O device must have its own registers to store byte count and memory address. It must also be able to generate control signals required for DMA data transfer. Generally such facilities are nor available with I/O devices. Single chip programmable DMA controllers have been developed by several manufactures for interfacing of I/O devices to the microprocessor for DMA data transfer. Such controllers provide all the facilities for DMA data transfer. Intel 8257 is described below.

11.8.1 Intel 8257

The Intel 8257 is a programmable DMA controller. Fig. 11.14 shows its schematic diagram. It is a 4-channel programmable direct memory access (DMA) controller. It is in a 40 pin I.C. package and requires a single + 5 V supply for its operation. Four I/O devices can be interfaced to the microprocessor through this device. It is capable of performing three operations, namely read, write and verify. During the read operation data are directly transferred from the memory to the I/O device. During the write operation data are transferred from the I/O device to the memory. On receiving a request from an I/O device, the 8257 generates a sequential memory address which allows the I/O device to read or write directly to or from the memory. Each channel incorporates two 16-bit registers, namely (i) DMA address register and (ii) byte count register. These registers are initialized before a channel is enabled. Initially, the DMA address register is loaded with the address of the first memory location to be accessed. During DMA operation it stores the next memory location to be accessed in the next DMA cycles. 14-LSBs of the byte count register store the number of bytes to be transferred. 2^{14} (16384) bytes of data can directly be transferred to the memory from the I/O device or from the memory to the I/O device. 2 MSBs of the byte count register indicate the operation which will be performed by the controller on that channel. Besides these registers the 8257 also

includes a mode set register and a status register.

Important pins of Intel 8257 are as follows:

DRQ0 — DRQ3. These are DMA request lines. An I/O device sends its DMA request on one of these lines. A HIGH status of the line generates a DMA request.

DACK0 — DACK3. These are DMA acknowledge lines. The Intel 8257 sends an acknowledge signal through one of these lines informing an I/O device that it has been selected for DMA data transfer. A LOW on the line acknowledges the I/O device.

lines are input lines. The inputs select one of the registers to be read or programmed. A4 — A7 lines give tristated outputs which carry 4 through 7 of the 16-bit memory address generated by the 8257.

A0 — A7. These are address lines. A0 — A3 are bidirectional lines. In the master mode these lines carry 4 LSBs of 16-bit memory address generated by the 8257. In the slave mode these lines are input lines. The inputs select one of the registers to be read or programmed. A4 — A7 lines give tristated outputs which carry 4 through 7 of the 16-bit memory address generated by the 8257.

Do — D7 These are data lines. These are bidirectional three-state lines. While programming the controller the CPU sends data for the DMA address register, the byte count register and the mode set register through these data lines. During DMA cycle, the 8257 sends the 8 MSBs of the memory address through these lines at the beginning of the DMA cycle. These 8 MSBs are then latched in 8212 latch. Tlwreafter the data bus is made available to handle memory data transfer during rest of (he I)MA cycle.

AEN Address latch enable.

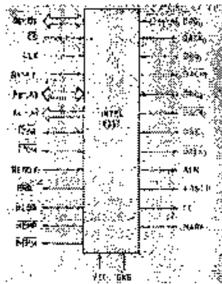


Figure 11.14 Schematic diagram of 8257

ADSTB A High on this line latches the 8MSBs of the address, which are sent on D-bus, into Intel 8212 connected for this purpose.

CS It is chip select.

I/OR I/O read. It is a bidirectional line. In output mode it is used to access data from the device during the DMA write cycle.

IO/W I/O write. It is a bidirectional line. In output mode it allows the transfer of data to the I/O device during the DMA read cycle. Data is transferred from the memory.

MEMR Memory read.

MEMW Memory write.

TC Byte count (Terminal count).

MARK Modulo 128 Mark.

CLK Clock

HRQ Hold request

HLDA Hold acknowledge.

An I/O device sends its request for DMA transfer through one of the four DRQ lines. On receiving the DMA request for DMA data transfer from an I/O device, the Intel 8257 sends the hold request hi the CPU through the HRQ line. The 8257 receives the hold acknowledge signal from the CPU through HLDA line. After receiving the hold acknowledge from the CPU, it sends DMA acknowledge to the I/O device through DACK line. The memory address is sent out on address and data lines. The 8257 sends 8 MSBs of the memory address over D-Bus. These 8 MSBs of the memory address are latched into 8212 using

ADSTB signal. ADSTB is similar to ALE of Intel 8085. For DMA read cycle, in which data are transferred from memory to I/O devices, two control signals MEMR arid 1/OW are issued by 8257. The MEMR enables the addressed memory for reading data from it. The I/OW enables the I/O device to accept data. Similarly, for DMA write cycle, in which data are transferred from the I/O device to the memory, two control signals MEMW and I/OR are issued by ii controller. The MEMW enables the addressed memory for writing data to it. The I/OR enables the I/O device to output data. The byte count is decremented by one after the transfer of one byte of data. When byte count becomes zero, TC goes high indicating that the data transfer using DMA is complete. The four DMA channels are programmed either in a fixed priority mode of operation. READY line is used by slow memory or I/O devices.

11.9 Sample Program : I/O INTERFACING

We shall connect a simple output device that has one buffer register. The microprocessor transfer the data to this buffer and the device consumes. Next byte must be transferred after the earlier byte is used. To do this called synchronization, we have what is called a device flag which is set when the data is put in the buffer. In this case microprocessor outputs the data so the flag is set by it and device when consumes it clears the flag. Microprocessor check this 0 (buffer empty) then puts new byte. This is actually done by a program that reads the flag and waits for flag to become 0. We can connect this flag to the system. Assume that out flag is connected to LSB of the 8 bit bus and it is addressed by 8 bit IO address of say 61hex then IN 61H will read the flag in A register with its LSB having the flag value. Assume that output device buffer address is 60H Then the IN 60 will read the buffer in A register. Program schematic for outputting N bytes to the device as subrouting. This way of doing IO is called programmed IO which involves waiting for flag.

```
NBYTEOUT:
PUSH B
PUSH D
PUSH H
PUSH PSW
LDA N
MOV D, A — counter in D
LOOP:                MVI H, X — address of X in HL pair
IN 61H
ANI 01 — extract the flag bit (LSB) A= 0 if flag =0 and non zero otherwise
JNZ WAIT
MOV A, M — Fa byte from X is put in A reg.
OUT 60H — data from A (from X) is put in IO buffer
INX H
DCR D
JNZ LOOP
POP PAW
POP H
POP D
POP B
RET — Return to calling program.
N DB 1 — Defines 1 byte location for N
```

Assume that we have a input device connected to ort 70H and its flag at port 71H

For input device the data is produced by the device so it sets the flag and CPU consume the data and clears the flag. The program thus will wait for flag to become 1 and then read the byte from buffer.

```
NBYTEOUT:
PUSH B
PUSH D — stores the program calling program state
PUSH H
```

```
PUSH PSW
LDA N
MOV D, A — counter in D
LOOP:                    MVI H, X — address of X in HL pair
IN 71H
ANI 01 — extract the flag bit (LSB) A= 0 if flag =0 and non zero otherwise
JZ LOOP
IN 70H — data is read from buffer
MOV M, A it is moved to locations X
INX H
DCR D
JNZ LOOP
POP PAW
POP H
POP D Restores the calling program state
POP B
RET Return to calling program
```

HARDWARE REQUIRED TO SUPPORT THE ABOVE PROGRAMS IS GIVEN BELOW

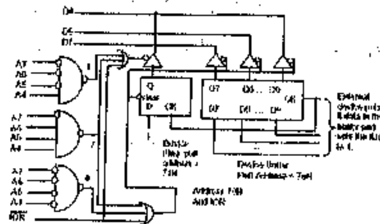


Figure: INPUT Device Interface with data port as 70 Hex and Status port as 71 H

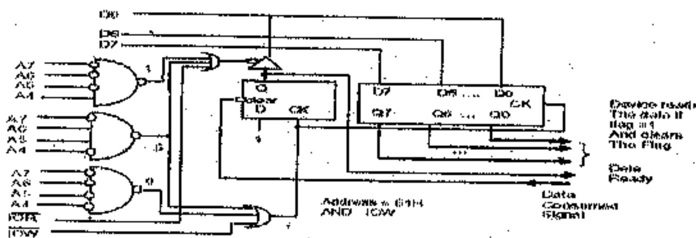


Figure: OUTPUT Device Interface Data output port as 60H and Status port (1 bit input port))as 61H

11.10 Summary

The interfacing concept can be summarized as follows

Peripheral-Mapped I/O

- The Out is a 2 byte instruction.
- A Latch is commonly used to interface output devices.
- The IN is a 2 byte instruction.
- A tri-state buffer is commonly used to interface input devices.
- To interface an output or input device, the low order address bus A7-A0 needs to be decoded to generate the device address pulse, which must be combined with the control signal IOR (or IOW) to select the device.

Memory Mapped I/O

- Memory related instructions are used to transfer data.
- To interface I/O devices, the entire bus must be decoded to generate the device address pulse, which must be combined with the control signal MEMR (or MEMW) to generate the I/O select pulse. This

pulse is used to enable the I/O device and transfer the data.

11.11 Glossary

CISC - Complex Instruction Set Computer. Describes the architecture of a processor family. CISC processors generally feature variable-length instructions, multiple addressing formats, and contain only a small number of general-purpose registers. Intel's 80x86 family is the quintessential example of CISC Contrast with RISC.

DMA-Direct memory access; a process in which circuits other than the CPU can read from or write to memory without processor intervention. Displacement-the amount by which a location differs from a reference point. When using data tables, the reference point is usually the base address.

Flag-A flip-flop used to indicate the status of an operation. For example, the zero flag will indicate if an operation results in zero if it is set.

Handshaking-The exchange of control and status information between two circuits. Handshaking is used to coordinate the transfer of data between circuits.

Interrupt- An asynchronous electrical signal from a peripheral to the processor. When the peripheral asserts this signal, an interrupt occurs. When an interrupt occurs, the current state of the processor is saved and an interrupt service routine is executed. When the interrupt service routine exits, control of the processor is returned to whatever part of the software was previously running.

Interrupt vector-A special code that identifies the circuit requesting an interrupt.

Port-an interface circuit capable of receiving from or placing information on the bus.

Software Interrupt- An interruption of a program that is initiated by a software instruction. Software interrupts are commonly used to implement breakpoints and operating system entry points. Unlike true interrupts, they occur synchronously with respect to program execution. In other words, software interrupts always occur at the beginning of an instruction execution cycle.

11.12 Further Readings

1.R. S. Goanker,"Microproccesor Architecture, Programming and Applications with the 8085/8080", 2nd Edition, New Age International Publishers Limited, ISBN-81-224-0710-2.

2.K.L. Short," Microprocessors and Programmed Logic",2nd Edition, Prentice Hall of India Pvt.Ltd.1988, ISBN-0-07-100462-9

3.Douglas V. Hall," Microprocessor and Interfacing",Mc-Graw Hill Book Company, 1987, ISBN-0-07-100462-9.

4.B. Ram," Fundamentals of Microprocessors and Microcomputere",5th rev ed.2001,Dhanpat Rai, New Delhi

11.13 Answers to Self Learning Exercises

1. Yes, They will be differentiated by control signal.
2. The port address will remain the same because the I/O port address is duplicated on both segments of the address bus.
3. To recognize the type of I/O , examine the control signal. If the control signal is LOW it must be peripheral I/O and if control signal is MEMW it must be memory mapped I/O.
4. In memory mapped I/O the microprocessor can not differentiate between an I/O and memory: it treats an I/O as if it is the memory. Therefore, an I/O and memory register can not have the same address; the entire memory map (64K) of the system has to be shared between memory and I/O.
5. This has to do with the design of the 8085 microprocessor. The instruction decoder or the associated micro program is designed to recognize the second byte as the low order byte in a three byte instruction.

11.14 Unit End Questions

1. Explain what is (a) Memory mapped I/O Scheme (b) I/O Mapped I/O Scheme
2. What is interrupt? Explain enabling, disabling and masking of interrupts. Discuss with suitable example how to transfer data using interrupts.
3. Discuss Why an Interrupt controller is required.

4. What are I/O ports? What are programmable and nonprogrammable ports.
5. Show interface connection for I/O devices employing 74LS138 and explain how to determine address for each I/O device.
6. Explain what is vectored interrupt.
7. Discuss how memory chips and I/O devices are interfaced to a microprocessor.
8. If the speed of I/O devices do not match the speed of microprocessor, what type of data transfer techniques are used? Describe them briefly.
9. What is interfacing?
10. Explain the concept of Direct Memory Access (DMA).
11. Explain the functions of Handshake signals.

UNIT-12
Comparative Study Of 8085, 8086 & 8088

Structure Of The Unit

12.0 Objective
12.1 Introduction
12.1.1 Evolution from 8080/8085 to 8086
12.1.2 Evolution from 8086 to 8088
12.2 8085 Microprocessor
12.2.1 Pin diagram of 8085
12.2.2 Signal group of 8085
12.2.3 Block diagram of 8085
12.2.4 Description of 8085
12.3 8086 Microprocessor
12.3.1 Pin diagram of 8086
12.3.2 Signal group of 8086
12.3.3 Internal Organization of 8086
12.4 8088 Microprocessor
12.4.1 The Basic Architecture of 8088
12.5 Pentium processor
12.5.1 Brief History of Pentium
12.5.2 Block Diagram of Pentium
12.6 Dual Core processor
12.7 Summary
12.8 Glossary
12.9 Further Readings
12.10 Answers to Self Learning Exercises
12.11 Unit End Questions

12.0 Objective

After studying this unit, you will learn

- 8085 Pinout, internal architecture and block diagram
- 8086 Pinout, internal architecture and block diagram
- 8088 Pinout, internal architecture and block diagram
- Features of Pentium processor

12.1 Introduction

12.1.1 Evolution from 8080/8085 to 8086

- Intel introduced 8086 microprocessor in 1978. This 16-bit microprocessor was a major improvement over the previous generation of 8080/8085 series of microprocessors.

8086	**8080/8085**
1 megabyte (20-bit add. bus) 16-bit Data bus Pipelined processor (first single-chip µpr)	Memory of 64 kilobyte (16-bit add. bus) 8-bit data bus Non-pipelined µpr

In a system with pipelining, the data and the address bus are busy transferring data while the CPU

is processing information.

12.2.2 Evolution from 8086 to 8088

8086 was with 16-bit data bus internally and externally. All registers and the data bus carrying data in/out of the CPU were 16-bit.

- That time all the peripherals were designed around 8-bit microprocessor.
- It was expensive to built PCB with 16-bit data bus.

So Intel introduced 8088, which was;

- Identical to 8086 internally, but externally 8-bit data bus instead of 16-bit.
- It had 1 megabyte of memory like 8086.

IBMs decision to pick up 8088 as their choice of microprocessor in designing the IBM PC.

- 8088-based IBM PC was enormous success, because IBM and Microsoft made it an open system.
- This enabled the cloning of this system and resulted a huge growth in both hardware and software designs based on IBM PC.
- In contrast IBMs main competitor Apple computer introduced a closed system and blocked all attempts of cloning.

Product	8080	8085	8086	8088	80286	80386	80486
Year Introduced	1974	1976	1978	1979	1982	1985	1989
Clock rate (MHz)	2-3	3-8	5-10	5-8	6-16	16-33	25-50
No. of transistors	4500	6500	29,000	29,000	130,000	275,000	1.2 million
Physical memory	64K	64K	1M	1M	16M	4G	4G
Internal data bus	8	8	16	16	16	32	32
External data bus	8	8	16	8	16	32	32
Address bus	16	16	20	20	24	32	32
Data type (bits)	8	8	8,16	8,16	8,16	8,16,32	8,16, 32

Table 12.1: Evolution of Intel's Microprocessors

12.2 8085 Microprocessor

The salient features of 8085 μp are:

- It is a 8 bit microprocessor.
- It is manufactured with N-MOS technology.
- It has 16-bit address bus and hence can address up to 216 = 65536 bytes (64KB) memory locations through A0-A15.
- The first 8 lines of address bus and 8 lines of data bus are multiplexed AD0 – AD7.
- Data bus is a group of 8 lines D0 – D7.
- It supports external interrupt request.
- A 16 bit program counter (PC)
- A 16 bit stack pointer (SP)
- Six 8-bit general purpose register arranged in pairs: BC, DE, HL.
- It requires a signal +5V power supply and operates at 3.2 MHZ single phase clock.

• It is enclosed with 40 pins DIP (Dual in line package

12.2.1 PIN DIAGRAM OF 8085—

Following figure shows the pin diagram of 8085 microprocessor.

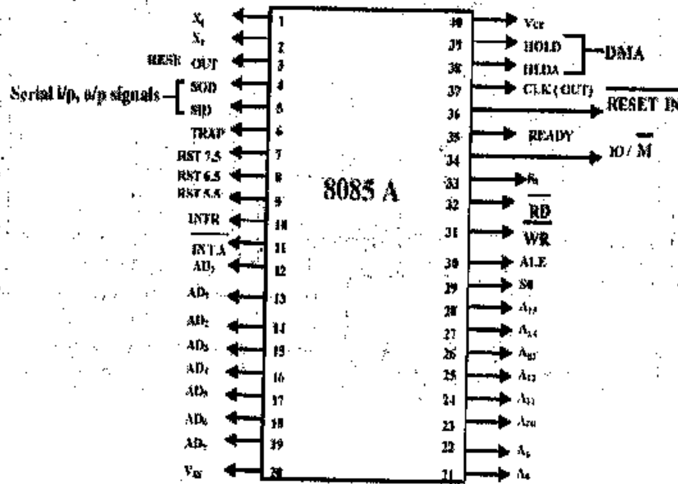


Figure 12.1 PIN DIAGRAM OF 8085

12.2.2 SIGNAL GROUP OF 8085 :

Following figure displays the signal group of 8085 processor.

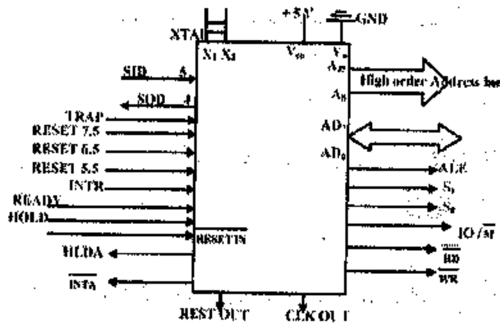


Figure 12.2 SIGNAL GROUP OF 8085

12.2.3 BLOCK DIAGRAM OF 8085

Block diagram of 8085 is displayed in the following figure.

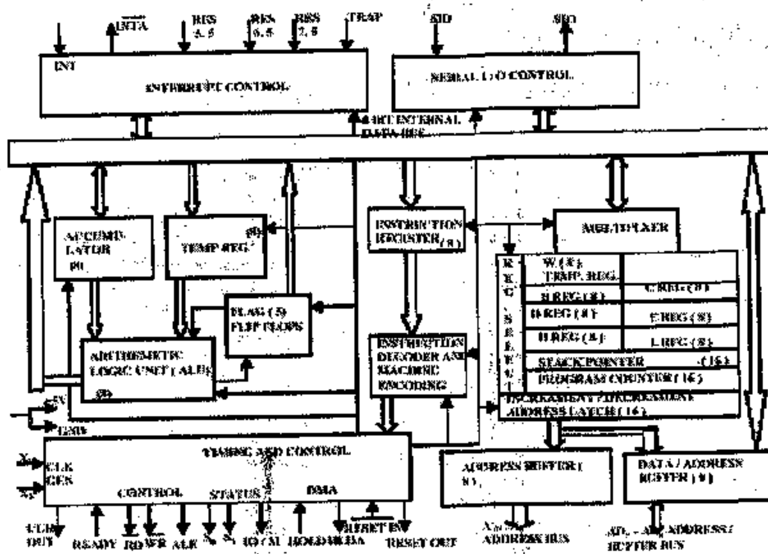


Figure 12.3 : BLOCK DIAGRAM OF 8085

The flag registers and general purpose registers of 8085 are shown below

FLAG REGISTERS

INDIVIDUAL	B, C, D, E, H, L
COMBINATION	B&C, D&E, H&L

GENERAL PURPOSE REGISTERS

12.2.4 Description of 8085

Memory

• Program, data and stack memories occupy the same memory space. The total addressable memory size is 64 KB.

• Program memory - program can be located anywhere in memory. Jump, branch and call instructions use 16-bit addresses, i.e. they can be used to jump/branch anywhere within 64 KB. All jump/branch instructions use absolute addressing.

•Data memory - the processor always uses 16-bit addresses so that data can be placed anywhere.

•Stack memory is limited only by the size of memory. Stack grows downward First 64 bytes in a zero memory page should be reserved for vectors used by RST instructions.

Interrupts

•The processor has 5 interrupts. They are presented below in the order of their priority (from lowest to highest):

•INTR is maskable 8080A compatible interrupt. When the interrupt occurs the processor fetches from the bus one instruction, usually one of these instructions:

•One of the 8 RST instructions (RST0 - RST7). The processor saves current program counter into stack and branches to memory location N * 8 (where N is a 3-bit number from 0 to 7 supplied with the RST instruction).

•CALL instruction (3 byte instruction). The processor calls the subroutine, address of which is specified in the second and third bytes of the instruction.

•RST5.5 is a maskable interrupt. When this interrupt is received the processor saves the contents of the

PC register into stack and branches to 2CH (hexadecimal) address.

- RST6.5 is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 34H (hexadecimal) address.
- RST7.5 is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 3CH (hexadecimal) address.
- TRAP is a non-maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 24H (hexadecimal) address.
- All maskable interrupts can be enabled or disabled using EI and DI instructions. RST 5.5, RST6.5 and RST7.5 interrupts can be enabled or disabled individually using SIM instruction.

Reset Signals

- RESET IN: When this signal goes low, the program counter (PC) is set to Zero, µp is reset and resets the interrupt enable and HLDA flip-flops.
- The data and address buses and the control lines are 3-stated during RESET and because of asynchronous nature of RESET, the processor internal registers and flags may be altered by RESET with unpredictable results.
- RESET IN is a Schmitt-triggered input, allowing connection to an be kept low a minimum of three clock periods.

Upon power-up, RESET IN must remain low for at least 10 ms after minimum Vcc has been reached.

- For proper reset operation after the power - up duration, RESET IN should be kept low a minimum of three clock periods.
- The CPU is held in the reset condition as long as RESET IN is applied Typical Power-on RESET RC values R_1 = 75K&!, C_1 = 1µF.
- RESET OUT: This signal indicates that µp is being reset. This signal can be used to reset other devices. The signal is synchronized to the processor clock and lasts an integral number of clock periods.

Serial communication Signal

- SID - Serial Input Data Line: The data on this line is loaded into accumulator bit 7 whenever a RIM instruction is executed.
- SOD - Serial Output Data Line: The SIM instruction loads the value of bit 7 of the accumulator into SOD latch if bit 6 (SOE) of the accumulator is 1.

DMA Signals

- HOLD: Indicates that another master is requesting the use of the address and data buses. The CPU, upon receiving the hold request, will relinquish the use of the bus as soon as the completion of the current bus transfer.
- Internal processing can continue. The processor can regain the bus only after the HOLD is removed.
- When the HOLD is acknowledged, the Address, Data RD, WR and IO/M lines are 3-stated.
- HLDA: Hold Acknowledge: Indicates that the CPU has received the HOLD request and that it will relinquish the bus in the next clock cycle.
- HLDA goes low after the Hold request is removed. The CPU takes the bus one half-clock cycle after HLDA goes low.
- READY: This signal Synchronizes the fast CPU and the slow memory, peripherals.
- If READY is high during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data.
- If READY is low, the CPU will wait an integral number of clock cycle for READY to go high before completing the read or write cycle.
- READY must conform to specified setup and hold times.

Registers

- Accumulator or A register is an 8-bit register used for arithmetic, logic, I/O and load/store operations.
- Flag Register has five 1-bit flags.
- Sign - set if the most significant bit of the result is set.

• Zero - set if the result is zero.
•Auxiliary carry - set if there was a carry out from bit 3 to bit 4 of the result.
•Parity - set if the parity (the number of set bits in the result) is even.
•Carry - set if there was a carry during addition, or borrow during subtraction/comparison/rotation.

General Registers

• 8-bit B and 8-bit C registers can be used as one 16-bit BC register pair. When used as a pair the C register contains low-order byte. Some instructions may use BC register as a data pointer.
• 8-bit D and 8-bit E registers can be used as one 16-bit DE register pair. When used as a pair the E register contains low-order byte. Some instructions may use DE register as a data pointer.
• 8-bit H and 8-bit L registers can be used as one 16-bit HL register pair. When used as a pair the L register contains low-order byte. HL register usually contains a data pointer used to reference memory addresses.
•Stack pointer is a 16 bit register. This register is always decremented/incremented by 2 during push and pop.
• Program counter is a 16-bit register

Instruction Set

•8085 instruction set consists of the following instructions:
• Data moving instructions.
• Arithmetic - add, subtract, increment and decrement.
• Logic - AND, OR, XOR and rotate.
•Control transfer - conditional, unconditional, call subroutine, return from subroutine and restarts.
• Input/Output instructions.
•Other - setting/clearing flag bits, enabling/disabling interrupts, stack operations, etc.

Addressing mode

• Register - references the data in a register or in a register pair.
•Register indirect - instruction specifies register pair containing address, where the data is located.
•Direct, Immediate - 8 or 16-bit data.

12.3 8086 Microprocesoor

•It is a 16-bit μp.
•8086 has a 20 bit address bus can access up to 220 memory locations (1 MB).
•It can support up to 64K I/O ports.
•It provides 14, 16 -bit registers.
•It has multiplexed address and data bus AD0- AD15 and A16 - A19.
•It requires single phase clock with 33% duty cycle to provide internal timing.
•8086 is designed to operate in two modes, Minimum and Maximum.
•It can prefetches upto 6 instruction bytes from memory and queues them in order to speed up instruction execution.
•It requires +5V power supply.
•A 40 pin dual in line package

Minimum and Maximum Modes:

•The minimum mode is selected by applying logic 1 to the MN / MX input pin. This is a single microprocessor configuration.
•The maximum mode is selected by applying logic 0 to the MN / MX input pin. This is a multi micro processors configuration.

12.3.1 Pin diagram of 8086

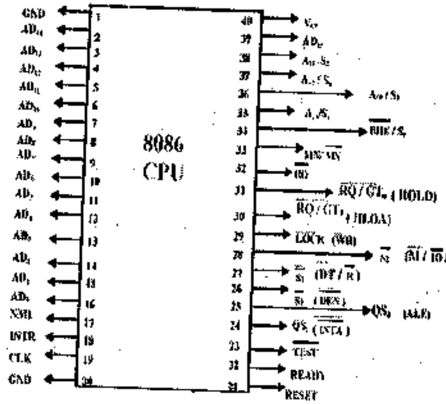


Figure 12.4 PIN DIAGRAM OF 8086

12.3.2 SIGNAL GROUP OF 8086

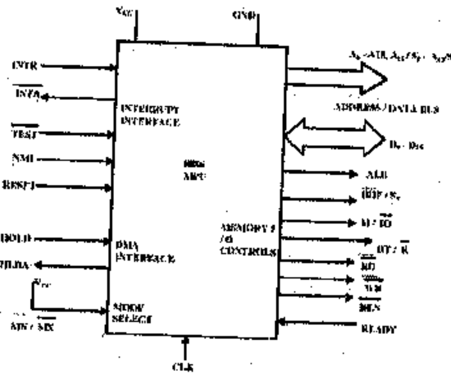


Figure 12.5 SIGNAL GROUP OF 8086

12.3.3 : Internal Organization of 8086

Now we will study the internal organization, pipelining and registers of 8086.

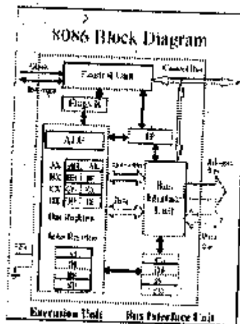
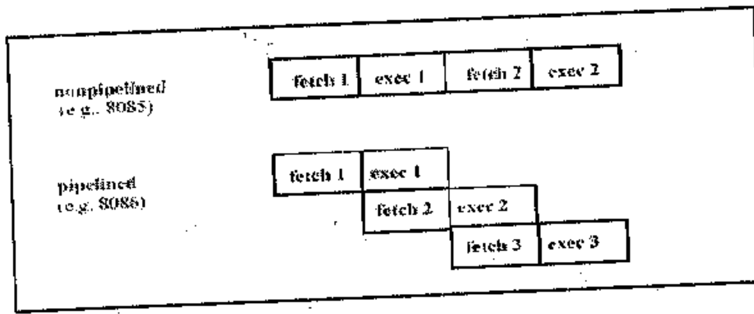


Figure 12.6 : Block diagram of 8086

Pipelining

•In the 8085 microprocessor, the CPU could either fetch or execute at a given time. CPU had to fetch an instruction from the memory, then execute it, then fetch again and execute it and so on.

•Pipelining is the simplest form to allow the CPU to fetch and execute at the same time. Note that the fetch and execute times can be different.



Intel implemented the concept of pipelining by splitting the internal structure of 8088/86 into two sections.
•the execution unit (EU)
•the bus interface unit (BIU)
•These two sections work simultaneously. BIU accesses memory and peripherals while the EU executes the instructions previously fetched.
•It only works if BIU keeps ahead of EU. Thus BIU has a buffer of queue. (8088 has 4 byte, and 8088 has 6 bytes).
•If the execution of any instruction takes to long, the BIU is filled to its maximum capacity and busses will stay idle. It starts to fetch again whenever there is 2-byte room in the queue.
•When there is a jump instruction, the microprocessor must flush out the queue. When a jump instruction is executed BIU starts to fetch information from the new location in the memory. In this situation EU must wait until the BIU starts to fetch the new instruction. This is known as branch penalty.

Registers of 8086 Microprocessor

- In the CPU, registers are used store information temporarily. The information can be one or two bytes of data, or the address of data.
- In 8088/8086 general-purpose registers can be accessed as either16-bit or 8-bit registers. All other registers can be accessed as full 16-bit registers.

AX 16 bit Register	
AH 8 bit register	AL 8 bit register

The bits of the registers are numbered in descending order:

8-bit register:

D7	D6	D5	D4	D3	D2	D1	D0

16-bit register:

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

- Different registers are used for different functions. Registers will be explained later within the context of instructions and their applications.
- The first letter of each general register indicates its use.
- AX is used for the accumulator.
- BX is used for base addressing register.
- CX is used for counter loop operations.
- DX is used to point out data in I/O operations.

These registers are in described in detail in the following table.

Category	Bits	Register Names
General	16	**AX, BX, CX, DX**
	8	**AH, AL, BH, BL, CH, CL, DH, DL**
Pointer	16	**SP** (stack pointer), **BP**(base pointer)
Index	16	**SI** (source index), **DI**(destination index)
Segment	16	**CS** (code segment), **DS**(data segment)
Instruction	16	**IP** (instruction pointer)
Flag	16	**FR** (flag register)

Note: the general registers can be accessed as full 16 bits (such as AX), or as the high byte only (AH) or low byte only (AL). The others are not!!

12.4 8088 Microprocessor

Intel 8088 microprocessor was released in 1979, or one year after the Intel 8086 CPU. Both processors have the same architecture, and the only difference of the 8088 CPU from the 8086 is the external data bus width - it was reduced from 16 bits to 8 bits. The 8088 CPU uses two consecutive bus cycles to read or write 16 bit data instead of one bus cycle for the 8086, which makes the 8088 processor to run slower. On the plus side hardware changes to the 8088 CPU made it compatible with 8080/8085 support chips. This was an important factor in choosing the 8088 processor for IBM PC line of computers because at that time 8-bit support chips were cheaper than 16-bit support chips, and there was better selection of 8-bit chips.

The 8088 microprocessor has 16-bit registers, 16-bit internal data bus and 20-bit address bus, which allows the processor address up to 1 MB of memory. The 8088 uses the same segmented memory addressing as the 8086: the processor can address 64 KB of memory directly, and to address more than 64 KB of memory the CPU has to break the update into a few parts - update up to 64 KB of memory, change segment register, update another block of memory, update segment register again, and so on.

Like to 8086, the 8088 microprocessor supports Intel 8087 numeric co-processor. The CPU recognizes all Floating-Point (FP) instructions, and, when necessary, it calculates memory address for FP instruction operand and does a dummy memory read. The FPU captures the calculated address and, possibly, the data, and proceeds to execute FP instruction. The CPU at the same time starts executing the next instruction. Thus, both integer and floating-point instructions can be executed concurrently.

Original Intel 8088 microprocessor was manufactured using HMOS technology. There were also CHMOS versions of the chip - 80C88 and 80C88A. These microprocessors had much lower power consumption and featured standby mode

12.4.1 The Basic Architecture of 8088

Below is a block diagram of the organizational layout of the Intel 8088 processor. It includes two main sections: the Execution Unit (EU) and the Bus Interface Unit (BIU). The EU takes care of the processing including arithmetic and logic. The BIU controls the passing of information between the processor and the devices outside of the processor such as memory, I/O ports, storage devices, etc.

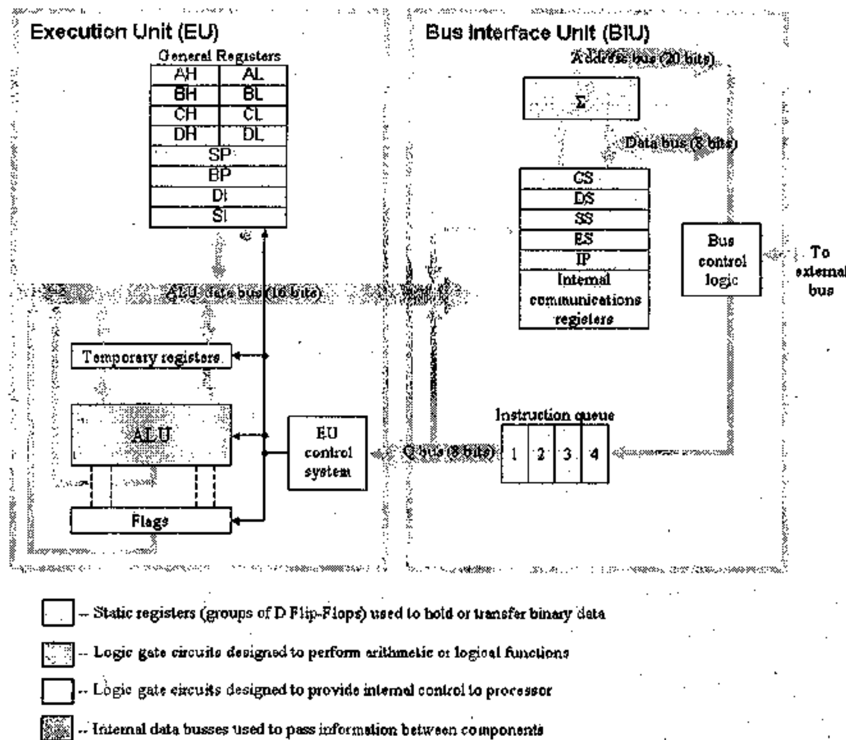


Figure 12.7: Basic Architecture of 8088

The rest of this document will describe the purpose of the different portions of the processor within these two units.

General Registers

The general registers are categorized into two sets: data and address. The data registers are for calculations; the address registers contain memory addresses and are used to point to the locations in memory where data will be retrieved or stored.

Examining the diagram shows that there are four pairs of registers at the top labeled AH, AL, BH, BL, CH, CL, DH, and DL. These are the data registers. Each of these registers is 8 bits long. Each pair, however, can also operate as a single 16 bit register. AH and AL can operate as a pair referred to as AX. This combining of registers is simply a concatenation, the 8 bits of AL simply tacked to the end of the 8 bits of AH. For example, if AH contains 101100002 (B016) and AL contains 010111112 (5F16), then the virtual register AX contains 10110000010111112 (B05F16).

Intel has given each of these computational registers a name. These names are listed below:

AX - Accumulator register

BX - Base register

CX - Counter register

DX - Data register

Below the data registers in the block diagram are the address registers: SP, BP, DI, and SI. These are officially referred to as the pointer (SP and BP) and index registers (DI and SI). These registers are used with the segment registers to point to specific addresses in the memory space of the processor. We will address their operation in the section on the segment registers. It is sufficient at this point to say that they act like pointers in the programming language C or C++. Their basic function is as follows:

SP is the stack pointer and it points to the "top plate" or last piece of data placed on the stack.

BP (base pointer), SI (source index), and DI (destination index) are all pointers that the programmer has for their own use.

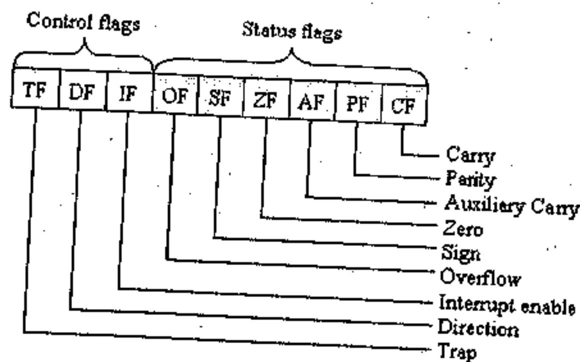
The Flags

Imagine the instrumentation on the dash board of a car. Blinking on and off occasionally behind the speedometer, tachometer, fuel gauge, and such, are a number of lights informally called "idiot lights". Each of these lights has a unique purpose. One comes on when the fuel is low. Another lights up when the high beams are on. Another warns the driver of low coolant. There are many more lights, and depending on the type of car you drive, some may even replace a gauge such as oil pressure.

Now let's go back to the processor. There are a number of "idiot lights" that the processor can use, each one based on the result of the previous operation. For example, the addition of two number might produce a negative sign, an erroneous overflow, a carry, or a value of zero. Well, that would be four idiot lights: sign, overflow, carry, and zero.

Each of these idiot lights, otherwise known as flags, can be represented with a single bit. If the resulting number had a negative sign, the sign flag would equal 1. If the result was not a negative number, (zero or greater than zero) the sign flag would equal 0. (Side note: Motorola processors more correctly refer to this flag as the negative flag.)

For the sake of organization, these flags are grouped together to form a single number. That number is the *flags register* shown at the bottom of the EU section of the processor diagram. The individual bits of the flags are arranged as shown in the figure below



The group of flags in the figure identified as *control flags* are used to control how the processor runs. These are typically controlled by the user's software. The group of flags in the figure identified as *status flags* are usually set by the previous operation as in our addition example.

Arithmetic Logic Unit

As implied by the name, the Arithmetic Logic Unit (ALU) is the computation portion of the EU. Any time arithmetic or logic needs to be performed on numbers, the numbers are sent from the general registers to the ALU, the ALU performs the function, and the result is sent back to the general registers.

EU Control System

The EU Control System is a set of gates that control the timing, passing of data, and other items within the execution unit. It's analogous to a manager in business who doesn't necessarily know the details of the operation, but they plan what happens, where it happens, and when it happens.

Instruction Pointer

The Instruction Pointer (IP) can be found toward the bottom of the group of registers in the center of the BIU. This register is an address register just like the SP, BP, DI, and SI registers in the EU. The difference is its purpose. The IP points to the next instruction to execute from memory. I will elaborate on this in the section on segment registers.

Segment Registers

In the center of the BIU section of the processor organizational block diagram is a set of registers labeled CS, DS, SS, and ES. These four registers are the *segment registers* and are used in conjunction with the *pointer and index registers* to store and retrieve items from the memory space.

You probably noticed that our address registers are 16 bits wide while the address space of the 8088 is 20 bits. (The memory space of the original 8088 is 2^{20} = 1 Meg.) So how does this work? Are four of the

address lines just ignored since we can only send 16 bits of information from our addressing registers? Of course not.

Next time your Windows operating system throws up an error, look to see if it gives you the address where the error occurred. If it does, you should see a number that looks something like:

3241:A34E

This number is actually the combination of *2* registers: a segment register (the number to the left of the colon) and a pointer or index register (the number to the right of the colon). Note that a four digit hexadecimal number results in a 16 bit binary number. It is the combination of these two 16-bit registers that creates the 20-bit address line.

To do this, take the value in the segment register and shift if left four places, i.e., add four zeros to the right side of the number. In our example above, 324116 = 0011 0010 0100 00012 becomes 3241016 = 0011 0010 0100 0001 00002. This value is then added to the pointer or index register. This makes the value from our example:

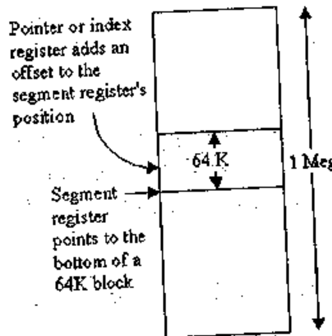
0011 0010 0100 0001 0000 (hexadecimal 32410)
\+ 1010 0011 0100 1110 (hexadecimal A34E)
0011 1100 0111 0101 1110 (hexadecimal 3C75E)

This computation takes place in the "Address Summing Block" located directly above the segment registers in the BIU in the organizational block diagram.

Therefore, the process of trying to access a single location in the 8088 processor's memory space takes three things:

a 16-bit *segment address* contained in one of the segment registers;
a 16-bit *offset address* contained in a pointer or index register; and
a 20-bit *physical address* which is the output from the address summing block.

If we look at this from the memory space point of view, the segment register shifted left four places so that four zeros are filled in from the right points to an address somewhere in the memory space. The offset address is then added to it to point to an address within the 216 = 65,535 (64K) locations above where the segment register is pointing.



There are two purposes for this summation of segment and pointer registers to access a single, physical memory address. First, it allows the processor to access more address lines (20 in the case of the 8088) than it has bits in its address registers (16 in the case of the 8088). There is, however, a more significant reason, a reason that allows you to load multiple programs at one time. It is called *relocatable code*. Let's put it this way, if you're loading both Microsoft Word and Netscape at the same time, does it matter which one you load first? Of course not.

The way this works is that the program itself has control of the pointers and index registers. It's as if they have a 64K block of memory to jump around in wherever they want. The operating system, however, has control of the segment registers. That way it can force a program to reside in a specific segment of memory. As long as the segment value stays the same for that program, and the program only manipulates the pointer and index registers, then there will be no errors. When something messes up one of these registers so that the physical address being pointed to is outside the allowed range for the program, that's when the "blue screen of death" appears.

Each segment register has its own name as shown below

- CS - Code Segment - A register pointing to the area of memory where the code is stored
- DS - Data Segment - A register pointing to the area of memory where the data is stored
- SS - Stack Segment - A register pointing to the area of memory where the processor temporarily stores register values in case they get messed up
- ES - Extra Segment - A register pointing to where ever the user wants it to point

Some of the segment registers and pointer registers are set up to operate in pairs for a specific purpose. These are:

- CS:IP — *code segment:instruction pointer* points to the physical address of the next instruction in memory to execute.
- SS:SP — *stack segment:stack pointer* points to the stack in memory, a temporary storage place for data.
- DS:DI — *data segment:destination index* points to the physical address in memory where data is to be *stored* using a pointer.
- DS:SI — *data segment:source index* points to the physical address in memory where data is to be *retrieved* using a pointer.

Bus Control Logic

The Bus Control Logic is a set of gates that control access to the external bus of the 8088. This includes all external memory devices, I/O ports, and other resources that communicate with the processor through the bus.

Pipelining

Microprocessor designers, in an attempt to squeeze every last bit of speed from their designs, try to make sure that every circuit is doing something productive at all times. If their is an idle circuit, then try to see if it can't predict what it should be doing and perform that function. If it predicted wrongly, then through the result away. If it predicted correctly, then time was saved and code was executed faster.

The most popular application of this theory is with the execution of the machine code instructions. In general, the execution of a machine code instruction can be broken into three stages:

fetch — retrieving the next instruction to execute from its location in memory

decode — determining which circuits to energize in order to execute the fetched instruction

execute — executing the instruction

When examining the architecture of the 8088 processor, you may notice that there are three separate circuits which perform these three tasks.

The bus control logic performs the fetch.

The EU control system performs the decode.

The ALU performs the execute.

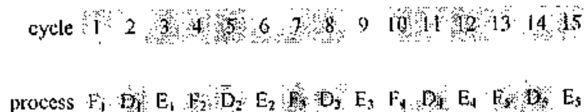
If the bus control logic is done fetching the current instruction, what's to keep it from fetching the next instruction? It may have to guess what the next instruction is, but if it guesses right, then the EU control system won't have to wait for the next instruction to be fetched once it's completed the execution of the current instruction.

And once the EU control system has finished telling the ALU what to do to execute the current instruction, what's to keep it from decoding the next instruction while it's waiting for the ALU to finish? If the bus control logic guessed right about what the next instruction is, then the ALU won't have to wait for a fetch and subsequent decode in order to execute the next instruction.

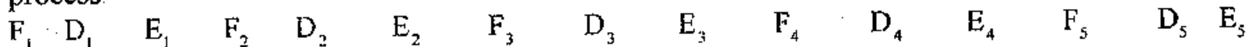
This is called *pipelining*, and it is an important method for speeding up the operation of a processor.

Keeping with our simple three part process for executing an instruction, the example below shows how much time can be saved with pipelining. (Note: "F" represents the fetch cycle, "D" represents the decode cycle, and "E" represents the execute cycle. The subscript after the letter indicates the instruction number.)

Processor operation without pipelining

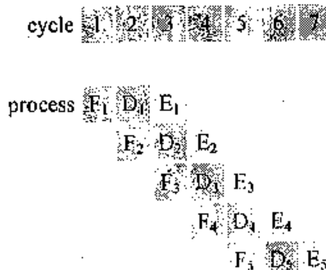


process



Without pipelining, five instructions take 15 cycles to execute. Now let's see how fast those same five instructions are executed using a pipelined architecture.

Processor operation with pipelining



Fifteen cycles reduced to seven. That's quite an improvement. If this pipelining thing works, it can make the processor appear a great deal faster. In fact, the following equations represent the difference.

number of cycles(non-pipelined) = 3 * number of instructions

number of cycles(pipelined) = 2 + number of instructions

Therefore, if the number of instructions is quite high, the number of cycles required of a pipelined architecture is almost 1/3 of that of the non-pipelined.

Instruction Queue

The Instruction Queue is the mechanism in the Intel 8088 processor that handles the pipelining function.

Self Learning Exercises

1. If CX contains the binary value 0110 1101 0110 1011_2, what value does CH have?

2. Assume the flag register is set as shown below after an addition. Using these flags, what can you tell us about the result?

TF	DF	IF	OF	SF	ZF	AF	PF	CF
0	0	0	0	1	0	0	0	1

3. If you were to add the binary number 10110101_2 and 10010110_2, how would the flags be set?

4. What is the difference in the number of cycles required to execute 50 instructions between a pipelined and a non-pipelined processor?

12.5 Pentium Processor

First we will discuss the history of Pentium processor and then we will study block diagram of Pentium processor and its main features.

12.5.1 Brief History of the Pentium Processor

The Pentium family of processors, which has its roots in the Intel486(TM) processor, uses the Intel486 instruction set (with a few additional instructions). The term "Pentium processor" refers to a family of microprocessors that share a common architecture and instruction set. The first Pentium processors (the P5 variety) were introduced in 1993. This 5.0-V processor was fabricated in 0.8-micron bipolar complementary metal oxide semiconductor (BiCMOS) technology. The P5 processor runs at a clock

frequency of either 60 or 66 MHz and has 3.1 million transistors.
The next version of the Pentium processor family, the P54C processor, was introduced in 1994. The
P54C processors are fabricated in 3.3-V, 0.6-micron BiCMOS technology. The P54C processor also
has System Management Mode (SMM) for advanced power management
The Intel Pentium processor, like its predecessor the Intel486 microprocessor, is fully software compatible
with the installed base of over 100 million compatible Intel architecture systems. In addition, the Intel
Pentium processor provides new levels of performance to new and existing software through a
reimplementation of the Intel 32-bit instruction set architecture using the latest, most advanced, design
techniques. Optimized, dual execution units provide one-clock execution for "core" instructions, while
advanced technology, such as superscalar architecture, branch prediction, and execution pipelining, enables
multiple instructions to execute in parallel with high efficiency. Separate code and data caches combined
with wide 128-bit and 256-bit internal data paths and a 64-bit, burstable, external bus allow these
performance levels to be sustained in cost-effective systems. The application of this advanced technology
in the Intel Pentium processor brings "state of the art" performance and capability to existing Intel architecture
software as well as new and advanced applications.

12.5.2 Block diagram of the Pentium

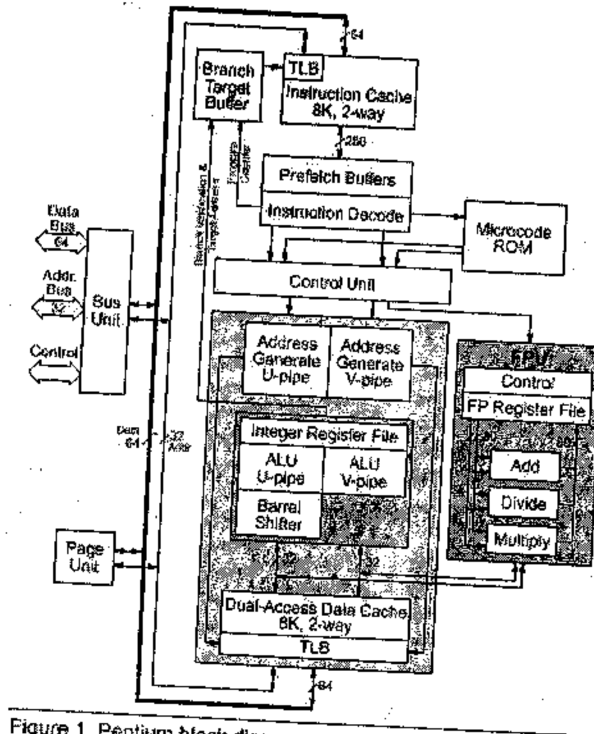


Figure 1. Pentium block diagram.

Above figure shows a block diagram of the Pentium design.
The most important enhancements over the 486 are the separate instruction and data caches, the dual
integer pipelines (the U-pipeline and the V-pipeline, as Intel calls them), branch prediction using the branch
target buffer (BTB), the pipelined floating-point unit, and the 64-bit external data bus. Even-parity checking
is implemented for the data bus and the internal RAM arrays (caches and TLBs).
As for new functions, there are only a few; nearly all the enhancements in Pentium are included to improve
performance, and there are only a handful of new instructions. Pentium is the first high-performance micro-
processor to include a system management mode like those found on power-miserly processors for
notebooks and other battery-based applications; Intel is holding to its promise to include SMM on all new

CPUs. Pentium uses about 3 million transistors on a huge 294 mm 2 (456k mils 2). The caches plus TLBs use only about 30% of the die. At about 17 mm on a side, Pentium is one of the largest microprocessors ever fabricated and probably pushes Intel's production equipment to its limits. The integer data path is in the middle, while the floating-point data path is on the side opposite the data cache. In contrast to other superscalar designs, such as SuperSPARC, Pentium's integer data path is actually bigger than its FP data path. This is an indication of the extra logic associated with complex instruction support. Intel estimates about 30% of the transistors were devoted to compatibility with the x86 architecture. Much of this overhead is probably in the microcode ROM, instruction decode and control unit, and the adders in the two address generators, but there are other effects of the complex instruction set. For example, the higher frequency of memory references in x86 programs compared to RISC code led to the implementation of the dual-ac.

Register set

The purpose of the Register is to hold temporary results, and control the execution of the program. General-purpose registers in Pentium are EAX, ECX, EDX, EBX, ESP, EBP,ESI, or EDI.

The 32-bit registers are named with prefix E, EAX, etc, and the least 16 bits 0-15 of these registers can be accessed with names such as AX, SI Similarly the lower eight bits (0-7) can be accessed with names such as AL & BL. The higher eight bits (8-15) with names such as AH & BH. The instruction pointer EAP known as program counter(PC) in 8-bit microprocessor, is a 32-bit register to handle 32-bit memory addresses, and the lower 16 bit segment IP is used for 16-bi memory address.

The flag register is a 32-bit register , however 14-bits are being used at present for 13 different tasks; these flags are upward compatible with those of the 8086 and 80286. The comparison of the available flags in 16-bit and 32-bit microprocessor is may provide some clues related to capabilities of these processors. The 8086 has 9 flags, the 80286 has 11 flags, and the 80286 has 13 flags. All of these flag registers include 6 flags related to data conditions (sign, zero, carry, auxiliary, carry , overflow, and parity) and three flags related to machine operations.(interrupts, Single-step and Strings). The 80286 has two additional : I/O Privilege and Nested Task. The I/O Privilege uses two bits in protected mode to determine which I/O instructions can be used, and the nested task is used to show a link between two tasks.

The processor also includes control registers and system address registers , debug and test registers for system and debugging operations.

Addressing mode & Types of instructions

Instruction set is divided into 9 categories of operations and has 11 addressing modes. In addition to commonly available instructions in a 8 bit microprocessor and this set includes operations such as bit manipulation and string operations, high level language support and operating system support. An instruction may have 0-3 operands and the operand can be 8, 16, or 32- bits long. The 80386 handles various types of data such as Single bit , string of bits , signed and unsigned 8-, 16-, 32- and 64- bit data, ASCII character and BCD numbers.

High level language support group includes instructions such as ENTER and LEAVE. The ENTER instruction is used to ENTER from a high level language and it assigns memory location on the stack for the routine being entered and manages the stack. On the other hand the LEAVE generates a return procedure for a high level language. The operating system support group includes several
instructions , such as APRL.(Adjust Requested Privilege Level) and the VERR/W (Verify Segment for Reading or Writing). The APRL is designed to prevent the operating system from gaining access to routines with a higher priority level and the instructions VERR/W verify whether the specified memory address can be reached from the current privilege level.

Operating mode and system management mode of Pentium

The Pentium processor has two primary operating modes and a "system management mode."

The operating mode determines which instructions and architectural features are accessible.

These modes are:

Protected Mode :This is the native state of the microprocessor. In this mode all instructions and architectural features are available, providing the highest performance and capability. This is the recommended mode that all new applications and operating systems should target. Among the capabilities of protected mode is

the ability to directly execute "real-address mode" 8086 software in a protected, multi-tasking environment. This feature is known as Virtual-8086 "mode" (or "V86 mode"). Virtual-8086 "mode" however, is not actually a processor "mode," it is in fact an attribute which can be enabled for any task (with appropriate software) while in protected mode.

·Real-Address Mode (also called "real mode") This mode provides the programming environment of the Intel 8086 processor, with a few extensions (such as the ability to break out of this mode). Reset initialization places the processor in real mode where, with a single instruction, it can switch to protected mode.

· System Management Mode The Pentium microprocessor also provides support for System Management Mode (SMM). SMM is a standard architectural feature unique to all new Intel microprocessors, beginning with the Intel386 SL processor, which provides an operating-system and application independent and transparent mechanism to implement system power management and OEM differentiation features. SMM is entered through activation of an external interrupt pin (SMI#), which switches the CPU to a separate address space while saving the entire context of the CPU. SMM-specific code may then be executed transparently. The operation is reversed upon returning.

Advanced Features

The Pentium P54C processor is the product of a marriage between the Pentium processor's architecture and Intel's 0.6-micron, 3.3-V BiCMOS process The Pentium processor achieves higher performance than the fastest Intel486 processor by making use of the following advanced technologies.

·Superscalar Execution: The Intel486 processor can execute only one instruction at a time. With superscalar execution, the Pentium processor can sometimes execute two instructions simultaneously.

·Pipeline Architecture: Like the Intel486 processor, the Pentium processor executes instructions in five stages. This staging, or pipelining, allows the processor to overlap multiple instructions so that it takes less time to execute two instructions in a row. Because of its superscalar architecture, the Pentium processor has two independent processor pipelines.

·Branch Target Buffer: The Pentium processor fetches the branch target instruction before it executes the branch instruction.

•Dual 8-KB On-Chip Caches: The Pentium processor has two separate 8-kilobyte (KB) caches on chip—one for instructions and one for data—which allows the Pentium processor to fetch data and instructions from the cache simultaneously.

• Write-Back Cache: When data is modified; only the data in the cache is changed. Memory data is changed only when the Pentium processor replaces the modified data in the cache with a different set of data

• 64-Bit Bus: With its 64-bit-wide external data bus (in contrast to the Intel486 processor's 32-bit-wide external bus) the Pentium processor can handle up to twice the data load of the Intel486 processor at the same clock frequency.

• Instruction Optimization: The Pentium processor has been optimized to run critical instructions in fewer clock cycles than the Intel486 processor.

• Floating-Point Optimization: The Pentium processor executes individual instructions faster through execution pipelining, which allows multiple floating-point instructions to be executed at the same time.

• Pentium Extensions: The Pentium processor has fewer instruction set extensions than the Intel486 processors. The Pentium processor also has a set of extensions for multiprocessor (MP) operation. This makes a computer with multiple Pentium processors possible.

A Pentium system, with its wide, fast buses, advanced write-back cache/memory subsystem, and powerful processor, will deliver more power for today's software applications, and also optimize the performance of advanced 32-bit operating systems (such as Windows 95) and 32-bit software applications.

12.6 Dual Core Processor

A dual core processor is a CPU with two separate cores on the same die, each with its own cache. It's the equivalent of getting two microprocessors in one.

In a single-core or traditional processor the CPU is fed strings of instructions it must order, execute, then selectively store in its cache for quick retrieval. When data outside the cache is required, it is retrieved.

through the system bus from random access memory (RAM) or from storage devices. Accessing these slows down performance to the maximum speed the bus, RAM or storage device will allow, which is far slower than the speed of the CPU. The situation is compounded when multi-tasking. In this case the processor must switch back and forth between two or more sets of data streams and programs. CPU resources are depleted and performance suffers.

In a dual core processor each core handles incoming data strings simultaneously to improve efficiency. Just as two heads are better than one, so are two hands. Now when one is executing the other can be accessing the system bus or executing its own code. Adding to this favorable scenario, both AMD and Intel's dual-core flagships are 64-bit.

To utilize a dual core processor, the operating system must be able to recognize multi-threading and the software must have simultaneous multi-threading technology (SMT) written into its code. SMT enables *parallel* multi-threading wherein the cores are served multi-threaded instructions in parallel. Without SMT the software will only recognize one core. Adobe Photoshop is an example of SMT-aware software. SMT is also used with multi-processor systems common to servers.

A dual core processor is different from a multi-processor system. In the latter there are two separate CPUs with their own resources. In the former, resources are shared and the cores
reside on the same chip. A multi-processor system is faster than a system with a dual core processor, while a dual core system is faster than a single-core system, all else being equal.

An attractive value of dual core processors is that they do not require a new motherboard, but can be used in existing boards that feature the correct socket. For the average user the difference in performance will be most noticeable in multi-tasking until more software is SMT aware. Servers running multiple dual core processors will see an appreciable increase in performance.

Multi-core processors are the goal and as technology shrinks, there is more "real-estate" available on the die. In the fall of 2004 Bill Siu of Intel predicted that current accommodating motherboards would be here to stay until 4-core CPUs eventually force a changeover to incorporate a new memory controller that will be required for handling 4 or more cores.

12.7 Summary

- The 8085 is a 8 bit microprocessor , manufactured with N-MOS technology. It t has 16-bit address bus. It supports external interrupt request. A 16 bit program counter (PC). A 16 bit stack pointer (SP). Six 8-bit general purpose register arranged in pairs: BC, DE, HL. It requires a signal +5V power supply and operates at 3.2 MHZ single phase clock.. It is enclosed with 40 pins DIP (Dual in line package).
- The 8086 is a 16-bit μp. It has a 20 bit address bus can access up to 220 memory locations (1 MB). It can support up to 64K I/O ports. It provides 14, 16 -bit registers. It has multiplexed address and data bus AD0- AD15 and A16 - A19. It requires single phase clock with 33% duty cycle to provide internal timing. 8086 is designed to operate in two modes, Minimum and Maximum. It can prefetches upto 6 instruction bytes from memory and queues them in order to speed up instruction execution. It requires +5V power supply. A 40 pin dual in line package
- The 8088 microprocessor has 16-bit registers, 16-bit internal data bus and 20-bit address bus, which allows the processor address up to 1 MB of memory. The 8088 uses the same segmented memory addressing as the 8086: the processor can address 64 KB of memory directly, and to address more than 64 KB of memory the CPU has to break the update into a few parts - update up to 64 KB of memory, change segment register, update another block of memory, update segment register again, and so on.
- A Pentium system, with its wide, fast buses, advanced write-back cache/memory subsystem, and powerful processor, will deliver more power for today's software applications, and also optimize the performance of advanced 32-bit operating systems (such as Windows 95) and 32-bit software applications.

12.8 Glossary

Address Bus - A set of electrical lines connected to the processor and all of the peripherals with which itcommunicates. The address bus is used by the processor to select a specific memory location or

registerwithin a particular peripheral

Addressing modes-Techniques used by software or hardware in calculating an address.

Index Register-A special purpose register used by a processor when performing indexed addressing. Thevalue in the index register is usually the reference location to which a displacement will be added.

I/O-Input/output; the process by which the computer communicates with the outside world.

Instruction-A Microprocessor command.

Instruction Cycle-The time and activities associated with the performance of an instruction.

Instruction Fetch-A machine cycle used by the processor to obtain instructions from memory.

Instruction Register-The part of the CPU that stores the instruction while it is being decoded.

Instruction Set- Is referred to the set of instructions that the microprocessor can execute. The *instruction set* specifies the types of instructions (such as load/store, integer arithmetic, and branch instructions), the specific instructions, and the encoding used for the instructions. The instruction set definition also specifies the addressing modes used for accessing memory.

Interrupt- An asynchronous electrical signal from a peripheral to the processor. When the peripheral asserts this signal, an interrupt occurs. When an interrupt occurs, the current state of the processor is saved and an interrupt service routine is executed. When the interrupt service routine exits, control of the processor is returned to whatever part of the software was previously running.

Physical Address -The actual address that is placed on the address bus when accessing a memory location or register. The address used to access physically-implemented memory. This address can be translated from the effective address. When address translation is not used, this address is equal to the effective address.

12.9 Further Readings

1. Microprocessor, Architecture, Programming and Application with 8085-Gaonkar, John Wiley Eastern , Ltd, Publication

2. Microprocessors and interfacing-Douglas V Hall, Tata Mc-Graw Hill publication

3. Microcomputer Systems: The 8086/8088 family-Yu-Chen Lin, Glen A Gibson, Prentice Hall of India Publication

4. The 8086 Microprocessor : programming and interfacing the PC-Kenneth J Ayala, Penram publication

5. The 8086 family: John Uffenbeck, Prentice Hall of India publication.

12.10 Answers to Self Learning Exercises

1. CH contains $0110\ 1101_2$.

2. As a result of the addition, there was no overflow (OF=0), the result is negative (SF=1), it isn't zero (ZF=0, but you could've also told us that because it is negative), and there was a carry.

3. First, let's add the two numbers to see what the result is.

```
  1 0 1 1 0 1 0 1
+ 1 0 0 1 0 1 1 0
1 0 1 0 0 1 0 1 1
```

Now just go from left to right through the status flags.

1. OF=1 — There was an overflow, i.e., adding two negative numbers resulted in a positive number.
- SF=0 — The result is positive.
- ZF=0 — The result does not equal zero.
- AF=0 — For now we won't worry about the auxiliary flag.
- PF=0 — For now we won't worry about the parity flag.
- CF=1 — There was a carry.

4. number of cycles(non-pipelined) = 3 * 50 = 150 cycles

number of cycles(pipelined) = 2 + 50 = 52 cycles

12.11 Unit End Questions

1. What are important signals of Intel 8086?
2. How many operating modes does 8086 have?
3. How many functional units does 8086 contain?
4. What is the function of a segment register in 8086?
5. What are conditional and control flags in 8086?
6. How many interrupt lines does 8086 have?
7. What physical address is represented by:
   (i) 4370 : 561E H (ii) 7A32 : 0028 H
8. Describe the difference between the instructions:
   (i) MOV AL, 0DB H (ii) MOV AL,DB H