University of Technology- Computer Science

Computer Architecture

Third Class

2023-2024

Asst. Prof Dr. Raheem Abdul Sahib

Computer Architecture

Lecturers:

Asst. Prof. Dr. Raheem Abdul Sahib Ogla

2023-2024

Second Course

Part One

Computer Architecture – 2nd course (syllabus)

➢ Introduction to Computer Organization

- RISC and CISC
- I/O Organization and Peripheral Control Strategies.
- I/O Interfaces and Programming
- Asynchronous data transfer

➢ Memory Management.

- Memory types and Hierarchy
- Main Memory address map.
- Associative Memory and Content Addressable Memories.

➢ Parallel Processing

- Pipeline (general consideration).
- Arithmetic Pipeline.
- Instruction Pipeline.
- Difficulties and Solutions in Instruction Pipeline.
- Vector processing and Array Processing.

1. Introduction

Computer architecture is the organization of the components which make up a computer system and the meaning of the operations which guide its function. It defines what is seen on the machine interface, which is targeted by programming languages and their compilers.

Q1: \ What is computer architecture?

Computer architecture can be defined as a set of rules and methods that describe the functionality, management and implementation of computers. To be precise, it is nothing but rules by which a system performs and operates.

Computer Architecture can be divided into mainly three categories, which are as follows −

- **Instruction set Architecture or ISA** − Whenever an instruction is given to processor, its role is to read and act accordingly. It allocates memory to instructions and also acts upon memory address mode (Direct Addressing mode or Indirect Addressing mode).
- **Micro Architecture** − It describes how a particular processor will handle and implement instructions from ISA.
- **System design** − It includes the other entire hardware component within the system such as virtualization, multiprocessing.

Role of computer Architecture

The main role of Computer Architecture is to balance the performance, efficiency, cost and reliability of a computer system.

For Example − Instruction set architecture (ISA) acts as a bridge between computer's software and hardware. It works as a programmer's view of a machine.

Computers can only understand binary language (i.e., 0, 1) and users understand high level language (i.e., if else, while, conditions, etc). So to communicate between user and computer, Instruction set Architecture plays a major role here, translating high level language to binary language.

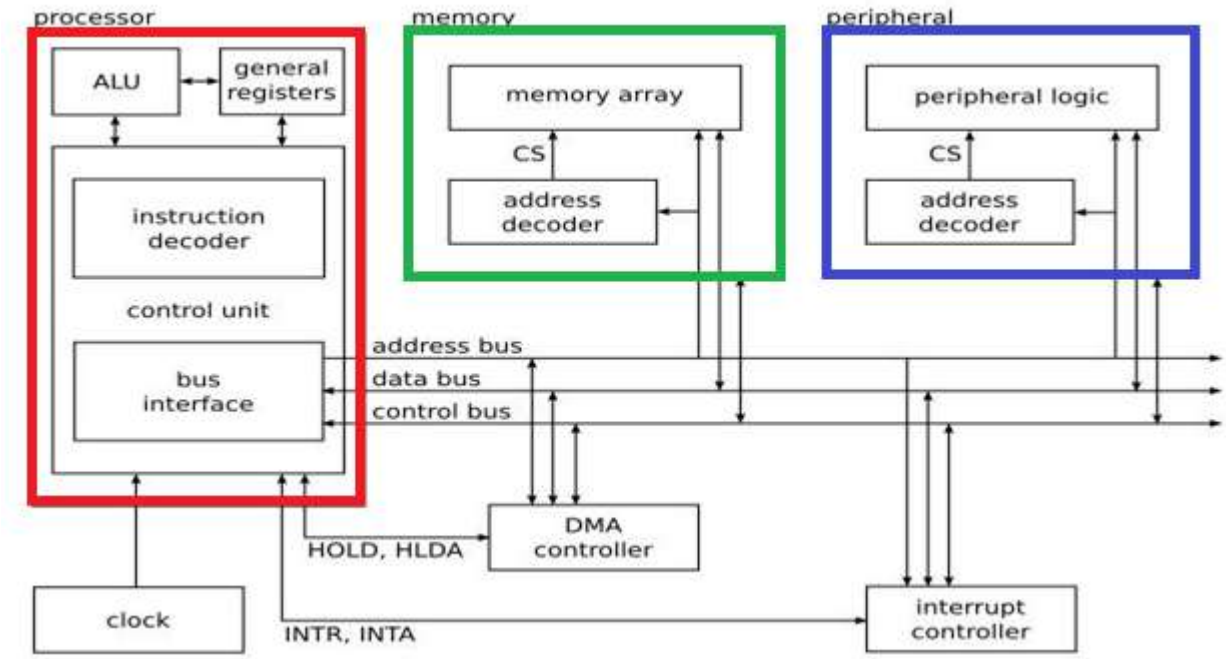
Structure Computer Architecture

Example structure of Computer Architecture as given below. Generally, computer architecture consists of the following −

- Processor
- Memory
- Peripherals

All the above parts are connected with the help of system bus, which consists of **address bus**, **data bus** and **control bus.**

The diagram given below depicts the computer architecture −



1.1. RISC and CISC Architecture

Means classification of instructions set to categories to increase the performance of Computer (Classification Instruction Set According Languages program)

1. Reduced Instruction Set Computer (RISC)

An important aspect of computer architecture is the design of the instruction set for the processor. The instruction set chosen for a particular computer determines the way that machine language programs are constructed. A computer with a large number of instructions is classified as a complex instruction set computer, abbreviated CISC. In the early 1980s, a

Reduced Instruction Set Computer or RISC Architecture

The fundamental goal of RISC is to make hardware simpler by employing an instruction set that consists of only a few basic steps used for evaluating, loading, and storing operations. A load command loads data but a store command stores data.

Characteristics of RISC:

1. It has simpler instructions and thus simple instruction decoding.
2. More general-purpose registers.
3. The instruction takes one clock cycle in order to get executed.
4. The instruction comes under the size of a single word.
5. Pipeline can be easily achieved.
6. Few data types.
7. Simpler addressing modes.

2. Complex Instruction Set Computer or CISC Architecture

The fundamental goal of CISC is that a single instruction will handle all evaluating, loading, and storing operations, similar to how a multiplication command will handle evaluating, loading, and storing data, which is why it's complicated.

Characteristics of CISC:

1. Instructions are complex, and thus it has complex instruction decoding.
2. The instructions may take more than one clock cycle in order to get executed.
3. The instruction is larger than one-word size.
4. Lesser general-purpose registers since the operations get performed only in the memory.
5. More data types.
6. Complex addressing modes.

Both CISC and RISC approaches primarily try to increase the performance of a CPU. Here is how both of these work:

1. CISC: This kind of approach tries to minimize the total number of instructions per program, and it does so at the cost of increasing the total number of cycles per instruction.

2. RISC: It reduces the cycles per instruction and does so at the cost of the total number of instructions per program.

$$CPU\ Time = \frac{Seconds}{Program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instructions} \times \frac{Seconds}{Cycle}$$

When programming was done in assembly language earlier, there was a desire to make the instructions perform more tasks. It is because assembly programming was arduous (شاق) and error-prone (معرض للخطأ) and led to the evolution of CISC architecture. But as the dependency of high-level language on assembly language decreased, RISC architecture prevailed (هو السائد).

Example

Suppose we need to add two different 8-bit numbers:

1. CISC approach: There would be a single instruction or command for this, such as ADD, that would perform the task.

2. RISC approach: In this case, the programmer would write the very first load command in order to load data in the registers. Then it would use a suitable operator and store the obtained result in the location that is desired.

The add operation here is divided into parts, namely, **operate, load, and store**. Due to this, RISC programs are much longer, and they require more memory to get stored, even though they require fewer transistors because the commands are less complex.

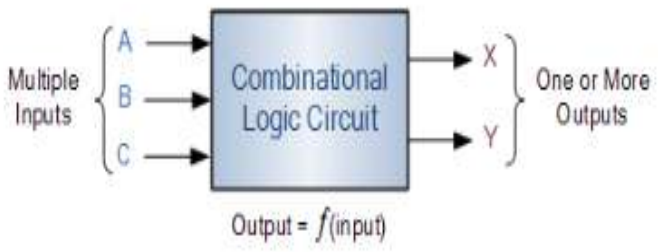
Q1: \ What is combinational circuit?

In digital electronics, a combinational circuit is **a circuit in which the output depends on the present combination of inputs**. Combinational circuits are made up of **logic gates**. The output of each logic gate is determined by its logic function.

Q2: What is difference between combinational and sequential circuit?

1. **combinational circuit** is time-independent. The output it generates does not depend on any of its previous inputs.
2. **sequential circuits** are the ones that depend on clock cycles. They depend entirely on the past as well as the present inputs for generating output.

Parameters	Combinational Circuit	Sequential Circuit
Meaning and Definition	It is a type of circuit that generates an output by relying on the input it receives at that instant, and it stays independent of time.	It is a type of circuit in which the output does not only rely on the current input. It also relies on the previous ones.
Feedback	A Combinational Circuit requires no feedback for generating the next output. It is because its output has no dependency on the time instance.	The output of a Sequential Circuit, on the other hand, relies on both- the previous feedback and the current input. So, the output generated from the previous inputs gets transferred in the form of feedback. The circuit uses it (along with inputs) for generating the next output.
Performance	We require the input of only the current state for a Combinational Circuit. Thus, it performs much faster and better in comparison with the Sequential Circuit.	In the case of a Sequential Circuit, the performance is very slow and also comparatively lower. Its dependency on the previous inputs makes the process much more complex.
Complexity	It is very less complex in comparison. It is because it basically lacks implementation of feedback.	This type of circuit is always more complex in its nature and functionality. It is because it implements the feedback, depends on previous inputs and also on clocks.
Elementary Blocks	Logic gates form the building/ elementary blocks of a Combinational Circuit.	Flip-flops form the building/ elementary blocks of a Sequential Circuit.
Operation	One can use these types of circuits for both- Boolean as well as Arithmetic operations.	You can mainly make use of these types of circuits for storing data.



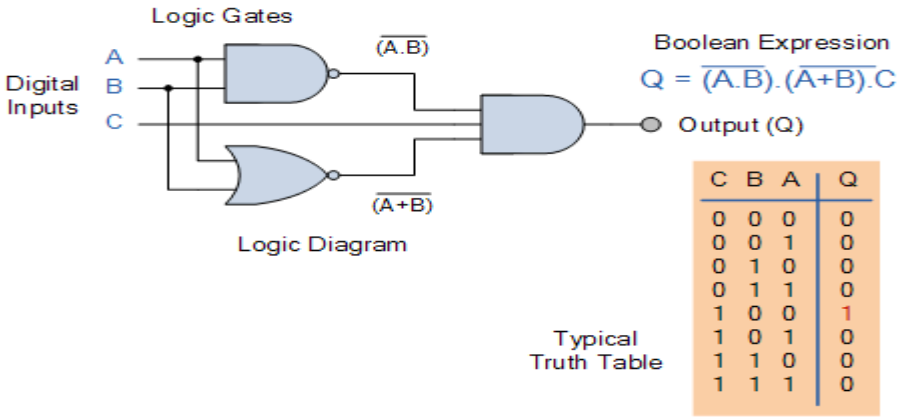
Combinational Logic Circuits

Combinational Logic Circuits are memoryless digital logic circuits whose output at any instant in time depends only on the combination of its inputs.

Unlike Sequential Logic Circuits whose outputs are dependent on both their present inputs and their previous output state giving them some form of ***Memory.***

The outputs of **Combinational Logic Circuits** are only determined by the logical function of their current input state, logic "0" or logic "1", at any given instant in time.

Q4: \ You have Three digital inputs (A, B, C) and output (Q), Design Combinational Logic Circuits (your Answer should appear (Logical diagram, Boolean Expression and Typical truth table)?
Solution



Q1: \ What are the differences between Combinational circuitry and State circuitry?

solution

1. **Combinational circuitry** behaves like a simple function. The output of combinational circuitry depends only on the current values of its input.
2. **State circuitry** behaves more like an object method. The output of state circuitry does not just depend on its inputs — it also depends on the past history of its inputs.

Q2: \What is the Difference Between Half Adder and Full Adder?

Solution

There is a primary difference between half adder and full adder. Half adder only adds the current inputs as 1-bit numbers and does not focus on the previous inputs. On the other hand, Full Adder can easily carry the current inputs as well as the output from the previous additions.

What is a Half Adder?

It is a combinational logic circuit. You can design it by connecting one AND gate and one EX-OR gate. A half-adder circuit consists of two input terminals- namely A and B. Both of these add two input digits (one-bit numbers) and generate the output in the form of a **carry and a sum**. Thus, there are two output terminals.

The output that one obtains from the EX-OR gate is the sum of both the one-bit numbers. The output obtained from the AND gate is called the carry. But you cannot forward the carry that you obtain in one addition into another addition. It is because of the absence of any logic gate to process it. Thus, it's called the Half Adder circuit.

We can write the equation of output for both the gates in the form of a logical operation that the logic gates perform. Here, we write the carry equation in the form of AND operation and the sum equation in the form of EX-OR operation.

Logical Expression of Half Adder

Sum (S) = $A \oplus B$

Carry (C) = $A \bullet B$

Truth Table

Here is a truth table representing the possible outputs obtained from the possible inputs in a Half Adder:

Input		Output	
A	B	CARRY	SUM
0	0	0	0
1	1	1	0
0	1	0	1
1	0	0	1

What is Full Adder?

A full adder is a circuit that has two AND gates, two EX-OR gates, and one OR gate. The full adder adds three binary digits. Among all the three, one is the carry that we obtain from the previous addition as C-IN, and the two are inputs A and B. It designates the input carry as the C-OUT and the normal output as S (or SUM).

Just like the Half Adder, the Full Ladder is a combinational type of logic circuit- meaning, it has no storage element. But it has additional logic gates. Thus, it adds the previous carry to generate the complete output. Thus, it is called the Full Adder.

One can also designate a Full Adder using one OR gate and two Half Adders. The OR gate here generates a carry that it obtains after the addition. We obtain the sum of these digits in the form of output from the second Half Adder.

The equation for the output that you can obtain by the EX-OR gate is the sum of all the binary digits. Here, the output that you obtain from the AND gate is the carry that you obtain by addition. This equation is in the form of a logical operation.

Logical Expression of Full Adder

CARRY-OUT = AB + BCin $\oplus$ ACin

SUM = (A $\oplus$ B) $\oplus$ Cin

Truth Table

A truth table represents the possible outputs obtained from the possible inputs. A truth table for the Full Adder is as follows:

Input			Output	
A	B	C	SUM	CARRY OUT
0	0	0	0	0
1	1	1	1	1
0	1	1	0	1
1	0	1	0	1
0	0	1	1	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1

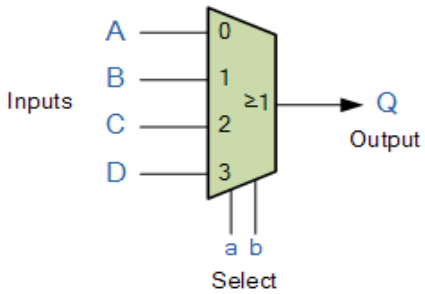
Q:\What are the main difference between Half Adder and Full Adder?
Solution

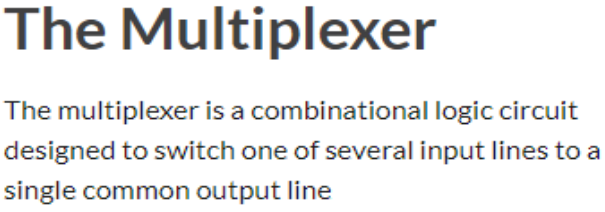
Parameter	Half Adder	Full Adder
Basics	The Half Adder is a type of combinational logic circuit that adds two of the 1-bit binary digits. It generates carry and sum of both the inputs.	The Full Adder is also a type of combinational logic that adds three of the 1-bit binary digits for performing an addition operation. It generates a sum of all three inputs along with a carry value.
Adding the Previous Carry	The Half Adder does not add the carry obtained from the previous addition to the next one.	The Full Adder, along with its current inputs A and B, also adds the previous carry.
Hardware Architecture	A Half Adder consists of only one AND gate and EX-OR gate.	A Full Adder consists of one OR gate and two EX-OR and AND gates.
Total Inputs	There are two inputs in a Half Adder- A and B.	There are a total of three inputs in a Full Adder- A. B. C-in.
Usage	The Half Adder is good for digital measuring devices, computers, calculators, and many more.	The Full Adder comes into play in various digital processors, the addition of multiple bits, and many more.
Logical Expression	Here is the logical expression of Half Adder: C = A * B S = A ⊕ B	Here is the logical expression of Full Adder: Cout = (AB) + CinA ⊕ CinB S =A ⊕ B ⊕ Cin

Q6: What Means by multiplexer?

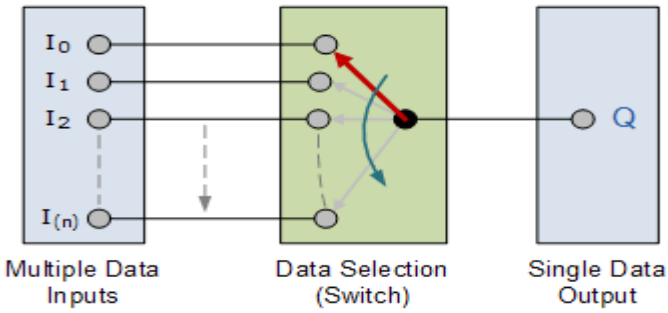
Multiplexing is the generic term used to describe the operation of sending one or more analogue or digital signals over a common transmission line at **different times** or **speeds** and as such, the device we use to do just that is called the multiplexer.

The *multiplexer*, shortened to "MUX" or "MPX", is a combinational logic circuit designed to switch one of several input lines through to a single common output line by the application of a control signal. Multiplexers operate like very fast acting multiple position rotary switches connecting or controlling multiple input lines called "channels" one at a time to the output.

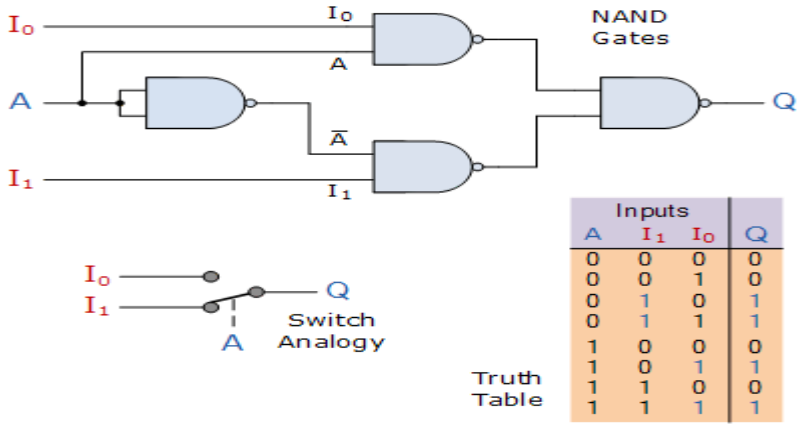




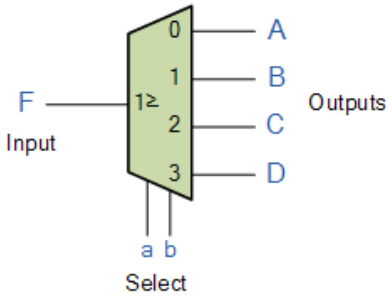
A. Basic multiplexing Switch



B. 2- multiplexing Switch



Q7: What Means by Demultiplexer



The Demultiplexer

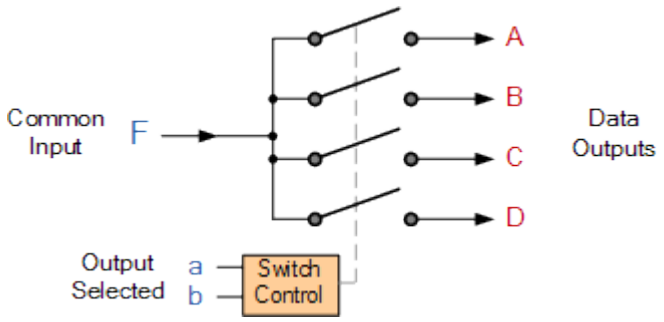
The demultiplexer is a combinational logic circuit designed to switch one common input line to one of several seperate output line

---

The data distributor, known more commonly as the demultiplexer or “Demux” for short, is the exact opposite of the Multiplexer we saw in the previous tutorial.

The *demultiplexer* takes one single input data line and then switches it to any one of a number of individual output lines one at a time. The **demultiplexer** converts a serial data signal at the input to a parallel data at its output lines as shown below.

1-to-4 Channel De-multiplexer



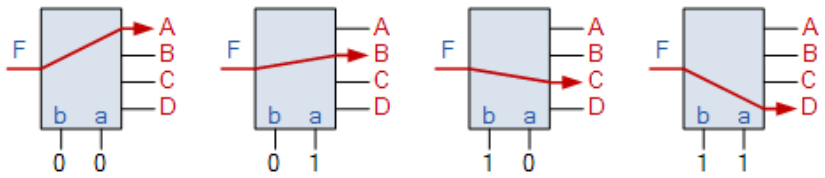
Output Select		Data Output Selected
b	a	
0	0	A
0	1	B
1	0	C
1	1	D

The Boolean expression for this 1-to-4 **Demultiplexer** above with outputs A to D and data select lines a, b is given as:

$$F = \overline{ab}A + a\bar{b}B + \bar{a}bC + abD$$

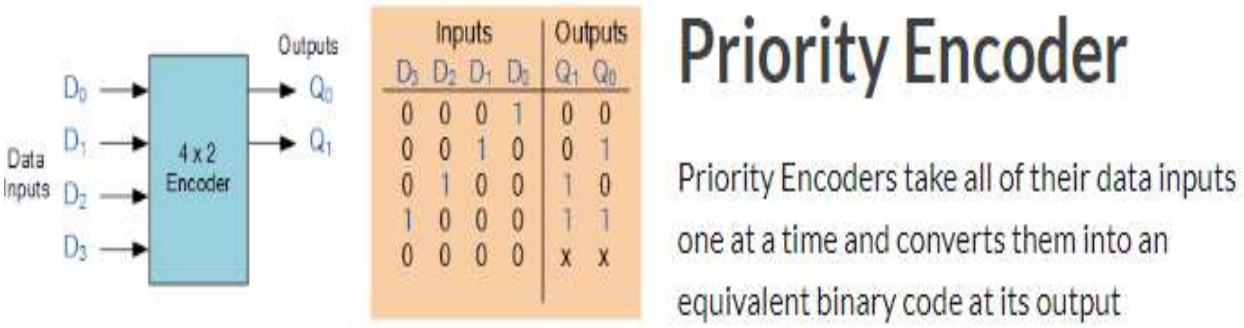
The function of the **Demultiplexer** is to switch one common data input line to any one of the 4 output data lines A to D in our example above. As with the multiplexer the individua solid state switches are selected by the binary input address code on the output select pins "a" and "b" as shown.

Demultiplexer Output Line Selection



As with the previous multiplexer circuit, adding more address line inputs it is possible to switch more outputs giving a 1-to-2^n data line outputs.

Q8: What Means by Priority Encoder



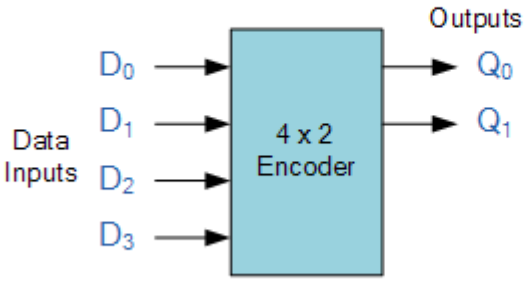
Unlike a multiplexer that selects one individual data input line and then sends that data to a single output line or switch. The job of a priority encoder is to produce a binary output address for the input with the highest priority.

The **Digital Encoder** more commonly called a **Binary Encoder** takes ALL its data inputs one at a time and then converts them into a single encoded output. So we can say that a binary encoder, is a multi-input combinational logic circuit that converts the logic level "1" data at its inputs into an equivalent binary code at its output.

Generally, digital encoders produce outputs of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines. An "n-bit" binary encoder has 2^n input lines and n-bit output lines with common types that include 4-to-2, 8-to-3 and 16-to-4 line configurations.

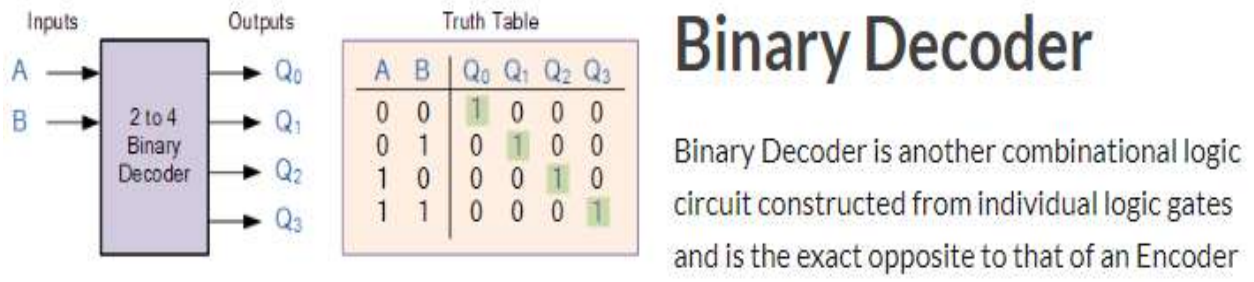
Example:

4-to-2 Bit Binary Encoder



Inputs				Outputs	
D_3	D_2	D_1	D_0	Q_1	Q_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1
0	0	0	0	x	x

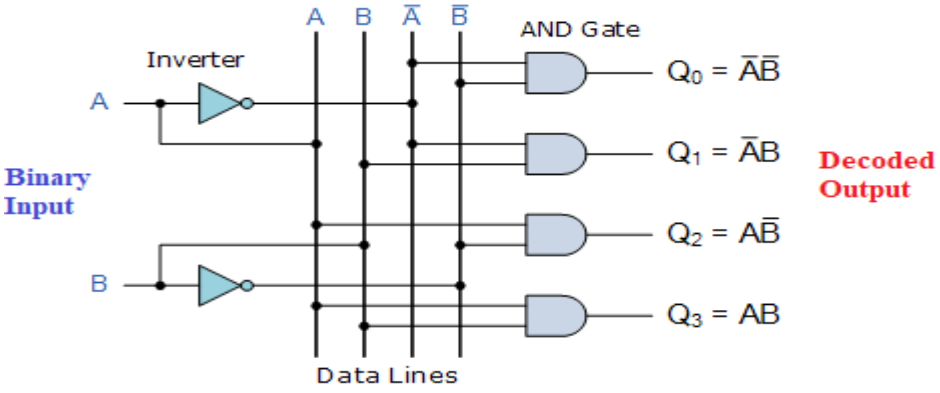
Q9: What Means by Binary Decoder?

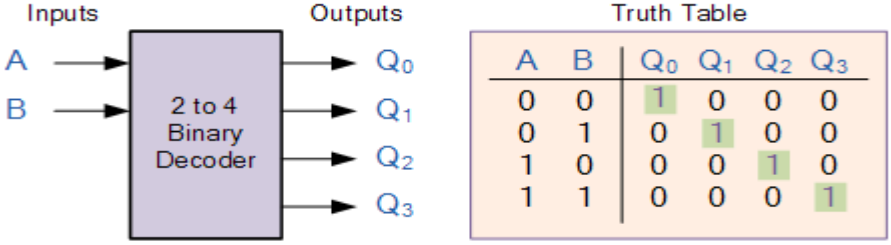


The term "**Decoder**" means to translate or decode coded information from one format into another, so a binary decoder transforms "n" binary input signals into an equivalent code using 2^n outputs.

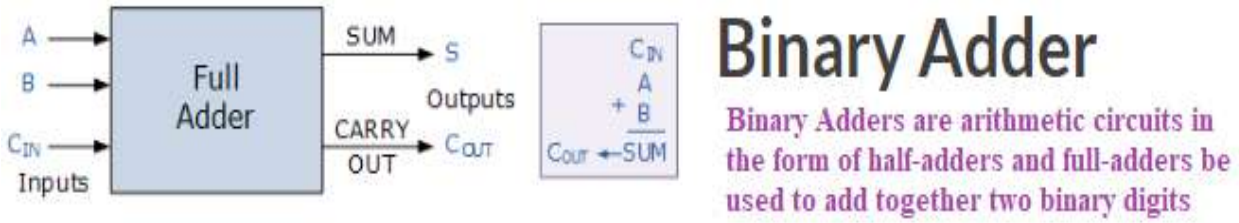
Binary Decoders are another type of digital logic device that has inputs of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines, so a decoder that has a set of two or more bits will be defined as having an *n*-bit code, and therefore it will be possible to represent 2^n possible values. Thus, a decoder generally decodes a binary value into a non-binary one by setting exactly one of its *n* outputs to logic "1".

Example: A 2-to-4 Binary Decoders





Q10: What Means by Binary Adder?



Another common and very useful combinational logic circuit which can be constructed using just a few basic logic gates allowing it to add together two or more binary numbers is the **Binary Adder**.

A basic Binary Adder circuit can be made from standard AND and Ex-OR gates allowing us to "add" together two single bit binary numbers, A and B.

The addition of these two digits produces an output called the SUM of the addition and a second output called the CARRY or Carry-out, (C_{OUT}) bit according to the rules for binary addition. One of the main uses for the ***Binary Adder*** is in arithmetic and counting circuits. Consider the simple addition of the two denary (base 10) numbers and Binary Addition of two bits below.

Decimal Addition

123	A	
+789	B	(Addend)
912	SUM	

Binary Addition of Two Bits

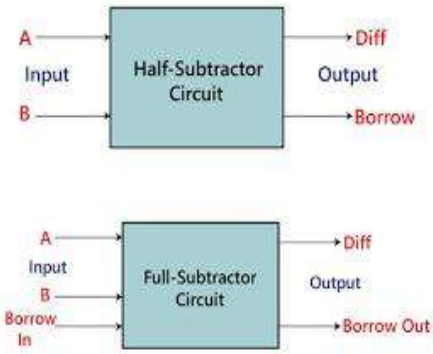
0	0	1	1
+0	+1	+0	+1
0	1	1 (carry)	1←0

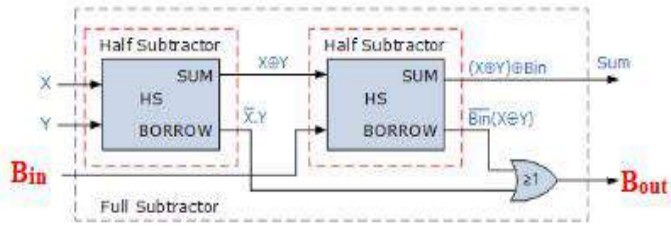
Q11: What Means by Binary Subtractor?

The Binary Subtractor is another type of combinational arithmetic circuit that produces an output which is the subtraction of two binary numbers.

As their name implies, a **Binary Subtractor** is a decision making circuit that subtracts two binary numbers from each other, for example, X – Y to find the resulting difference between the two numbers.

Unlike the Binary Adder which produces a SUM and a CARRY bit when two binary numbers are added together, the *binary subtractor* produces a DIFFERENCE, D by using a BORROW bit, B from the previous column. Then obviously, the operation of subtraction is the opposite to that of addition





COMPUTER ARCHITECTURE

Digital Computer

The digital computer is a digital system that performs various computational tasks. Digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit. Information is represented in digital computers in groups of bits. By using various coding techniques, groups of bits can be made to represent not only binary numbers but also other discrete symbols, such as decimal digits or letters of the alphabet. By judicious use of binary arrangements and by using various coding techniques, the groups of bits are used to develop complete sets of instructions for performing various types of computations.

A computer system is sometimes subdivided into two functional entities

1- The hardware of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device.
2- Computer software consists of the instructions and data that the computer manipulates to perform various data-processing tasks.

The system software of a computer consists of a collection of programs whose purpose is to make more effective use of the computer. The programs included in a systems software package are referred to as the operating system.

Computer Hardware

The hardware of the computer is usually divided into three major parts, as shown in Fig(1):

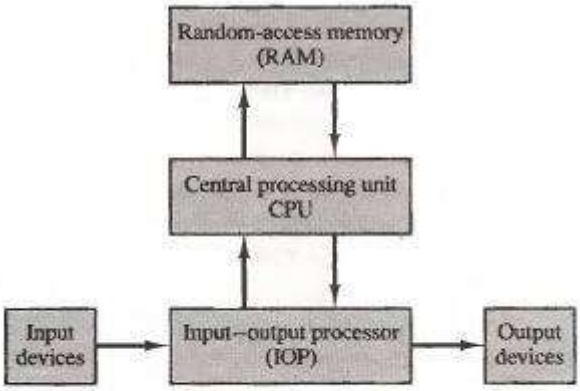


Figure 1 Block diagram of a digital computer.

The central processing unit (CPU) contains arithmetic and logic unit for manipulating data, a number of registers for storing data, and control circuits for fetching and executing instructions. The memory of a computer contains storage for instructions and data. It is called a random- access memory (RAM) because the CPU can access any location in memory at random and retrieve the binary information within a fixed interval of time. The input and output processor (IOP) contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside world. The input and output devices connected to the computer include keyboards, printers, terminals, magnetic disk drives, and other communication devices.

Computer Organization

Computer organization is concerned with the way the hardware components operate and the way they are connected together to form the computer system. The various components are assumed to be in place and the task is to investigate the organizational structure to verify that the computer parts operate as intended.

Computer Design

Computer design is concerned with the hardware design of the computer. Once the computer specifications are formulated, it is the task of the designer to develop hardware for the system. Computer design is concerned with the determination of what hardware should be used and how the parts should be connected. This aspect of computer hardware is sometimes referred to as computer implementation.

Computer Architecture

Computer architecture is concerned with the structure and behavior of the computer as seen by the user. It includes the information formats, the instruction set, and techniques for addressing memory. The architectural design of a computer system is concerned with the

specifications of the various functional modules, such as processors and memories, and structuring them together into a computer system.

Instruction Set Architecture

1. **Opcodes**: Consists of operate instructions: as logical and arithmetical instructions, Data movement instructions and Control instructions
2. **Data types**: consists of 8, 16, 32, and 64 bits
3. **Addressing modes**: consists of:
   - Operands specified
   - Next instruction to execute is specified
   - Architecture-specific
   - An instruction can use several addressing modes

Register Transfer Language

Digital systems vary in size and complexity from a few integrated circuits to a complex of interconnected and interacting digital computers. Digital system design invariably uses a modular approach(نهج معياري او نهج نمطي). The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called micro operations(MO).

A micro operation is an elementary operation (عملية ابتدائية) performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of micro operations are shift, count, dear, and load.

The internal hardware organization of a digital computer is best defined by specifying:

1- The set of registers it contains and their function.
2- The sequence of micro operations performed on the binary information stored in the registers.
3- The control that initiates the sequence of micro operations.

The symbolic notation used to describe the micro operation transfers among registers is called a ***register transfer language***.

The term "register transfer" implies the availability of hardware logic circuits that can perform a stated micro operation and transfer the result of the operation to the same or another register.

The word "language" is borrowed from programmers, who apply this term to programming languages.

A ***register transfer language*** is a system for expressing in symbolic form the micro operation sequences among the registers of a digital module.

Register Transfer

Computer registers are designated (يعين) by capital letters (sometimes followed by numerals) to denote the function of the register.

For example:

MAR: memory address register

PC: program counter

IR: instruction register

R1: processor register

The representation of registers in block diagram form is shown in Fig(2):

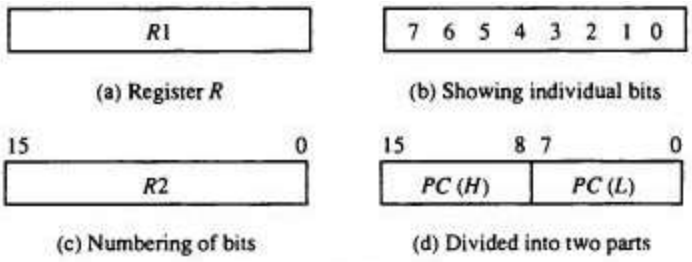


Figure 2 Block diagram of register.

a- Rectangular box with the name of the register inside.

b- The individual bits.

c- The numbering of bits in a 16-bit register can be marked on top of the box. d- 16-bit register is partitioned into two parts. Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H (for high byte).

d- The name of the 16-bit register is PC. The symbol PC (0-7) or PC(L) refers to the low-order byte and PC (8-15) or PC(H) to the high-order byte. Information transfer from one register to another is designated in symbolic form by means of a ***replacement*** operator. The statement:

$$R2 \leftarrow R1$$

Denotes a transfer of the content of register Rl into register R2. It designates a replacement of the content of R2 by the content of Rl. By definition, the content of the source register Rl does not change after the transfer.

If we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

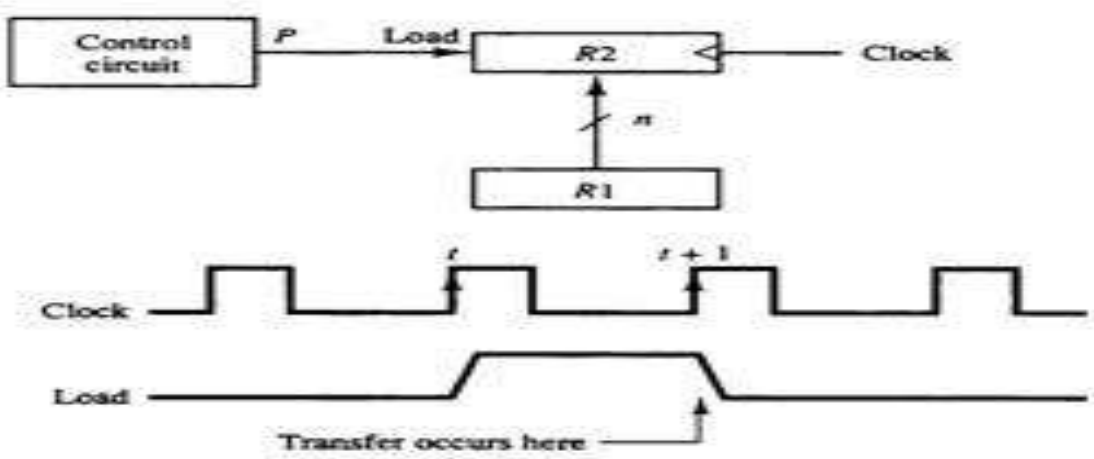
$$If\ (P = 1)\ then\ (R2 \leftarrow R1)$$

where P is a control signal generated in the control section. It is sometimes convenient

to separate the control variables from the register transfer operation control function by specifying a control function. (من الملائم أحيانًا فصل متغيرات التحكم عن وظيفة التحكم في عملية نقل السجل من خلال تحديد وظيفة التحكم.)

$$P: R2 \leftarrow R1$$

The control condition is terminated with a colon. It symbolizes (يرمز) the requirement that the transfer operation be executed by the hardware only if P = 1.



Transfer from R1 to R2 when P = 1.

To separate two or more operations that is executed at the same time by using the *comma* as the statement:

$$T: R2 \leftarrow R1, R5 \leftarrow R3$$

The basic symbols of the register transfer notation are listed in Table (1) Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register.

TABLE 1 Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR*, *R2*
Parentheses ( )	Denotes a part of a register	*R2*(0–7), *R2*(*L*)
Arrow ←	Denotes transfer of information	*R2* ← *R1*
Comma ,	Separates two microoperations	*R2* ← *R1*, *R1* ← *R2*

Bus and Memory Transfers

A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive (مُبَالَغ فيه) if separate lines are used between each register and all other registers in the system.

A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer. The multiplexers select the source register whose binary information is then placed on the bus. For example, the construction of a bus system for four registers is shown in Fig (3) Each register has four bits, numbered 0 through 3. The bus consists of four 4x1(4-input-one output) multiplexers each having four data inputs, 0 through 3, and two selection inputs, S1 and S0. (00,01,10,11)

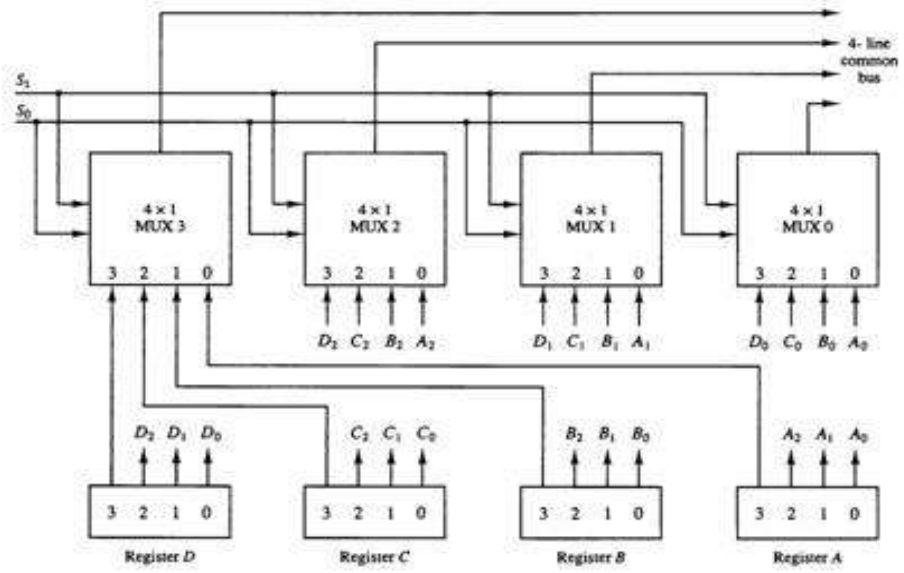


Fig 3 Bus system for four registers

The table (2) shows the register that is selected by the bus for each of the four possible binary values of the selection lines.

Table 2 Function for Bus of Fig

S_1	S_0	Register selected
0	0	*A*
0	1	*B*
1	0	*C*
1	1	*D*

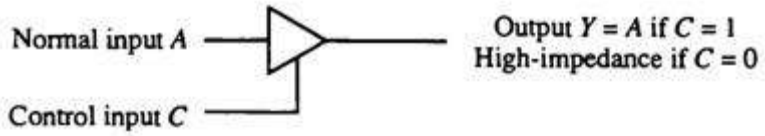
The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is including in the statement, the register transfer is symbolized as follows:

$$Bus \leftarrow C\ , \qquad R1 \leftarrow Bus$$

The content of register C is placed on the bus, and the content of the bus is loaded into register Rl by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

$$R1 \leftarrow C$$

A bus system can be constructed with three-state gates. The graphic symbol of a three state buffer gate is shown:



Graphic symbols for three-state buffer.

To construct a common bus for four registers of n bits each using three-state buffers, we need n circuits with four buffers in each as shown in Fig (4). Each group of four buffers receives one significant bit from the four registers.

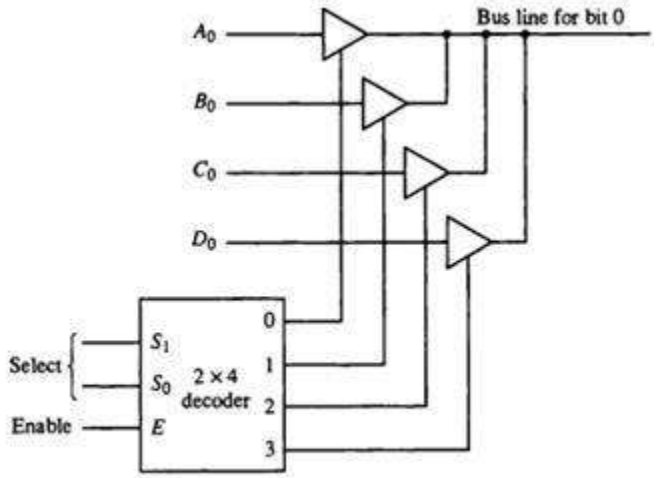


Figure 4 Bus line with three state-buffers.

The transfer of information from a memory word to the outside environment is called a read operation. The transfer of new information to be stored into the memory is called a write operation. Consider a memory unit that receives the address from a register, called the Address Register, symbolized by **AR**. The data are transferred to another register, called the Data Register, symbolized by DR.

$$Read\colon DR \leftarrow M[AR]$$

The write operation transfers the content of a data register to a memory word M selected by the address.

$$Write\colon M[AR] \leftarrow DR$$

Arithmetic Microoperations

The arithmetic operations are listed in the Table(3):

TABLE 3 Arithmetic Microoperations

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

The multiply and divide are not listed in Table (3), these two operations are valid arithmetic operations but are not included in the basic set of micro operations. In most computers, the multiplication operation is implemented with a sequence of add and shift micro operations. Division is implemented with a sequence of subtract and shift micro operations.
The digital circuit that generates the arithmetic sum of two binary numbers of any length is called a binary adder as shown in Fig (5).

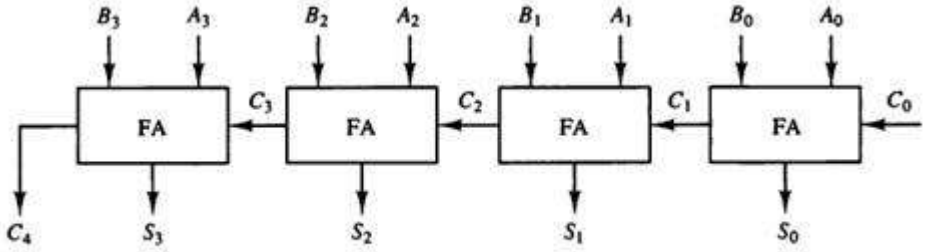


Figure 5 4-bit binary adder.

The *addition and subtraction operations* can be combined into one common circuit by including an exclusive-OR gate with each full-adder as shown in Fig (6).

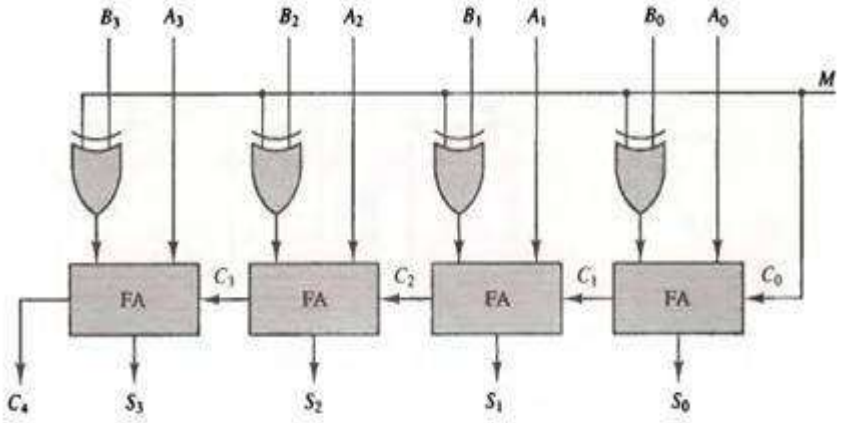


Figure 6 4-bit adder-subtractor.

The increment micro operation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. The diagram of a 4bit combinational circuit incremented is shown in Fig(7): (HA means Half Adder)

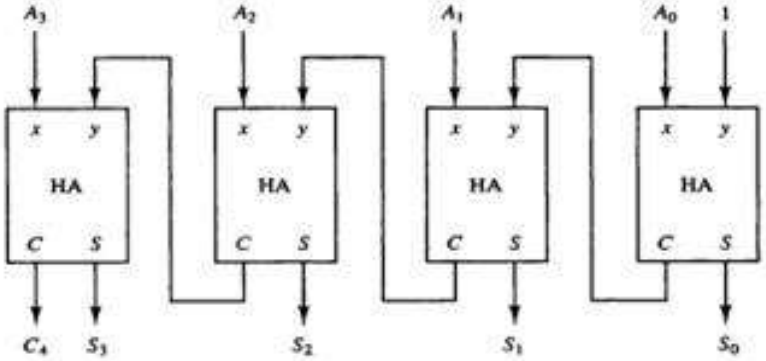


Figure 7 4-bit binary incrementer.

The arithmetic micro operations listed in the Table 3 can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations as shown in Fig (8).

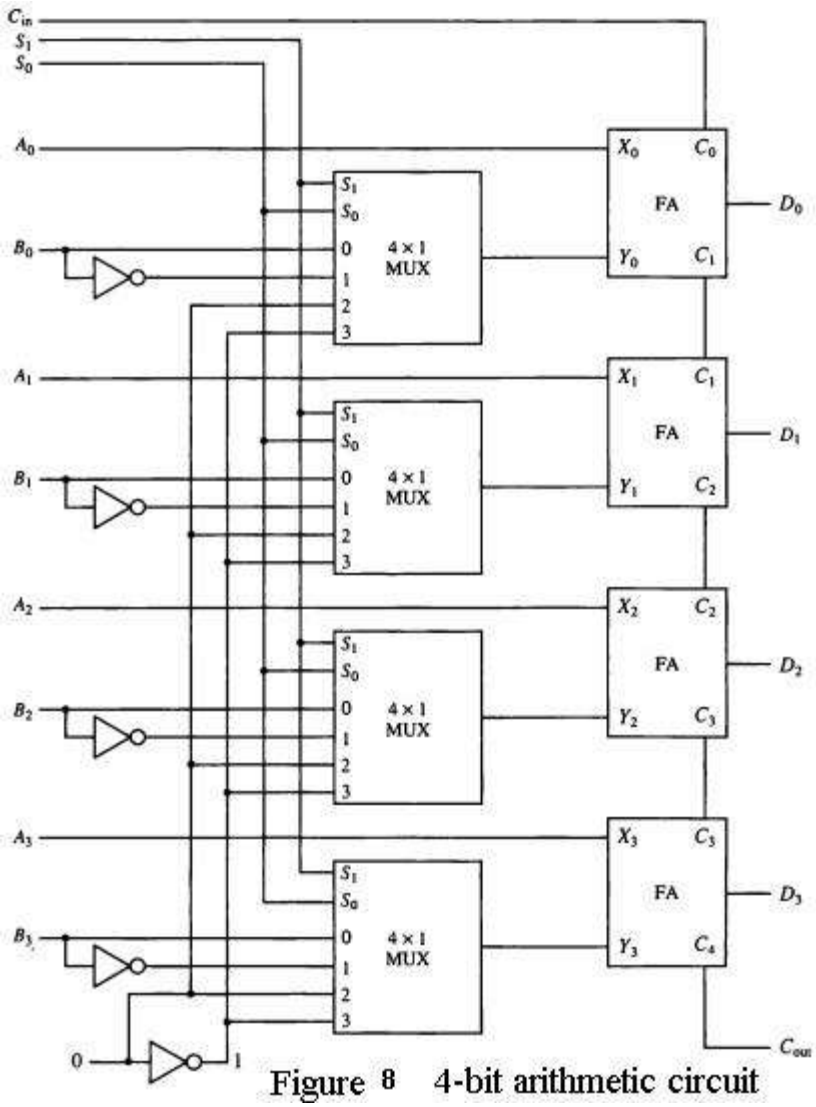


Figure 8 4-bit arithmetic circuit

It is possible to generate the eight arithmetic micro operations listed in Table (4):

TABLE 4 Arithmetic Circuit Function Table

Select			Input	Output	
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Logic Micro Operations

Logic micro operations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR micro operation with the contents of two registers Rl and R2 is symbolized by the statement:

$$P: R1 \leftarrow R1 \oplus R2$$

It specifies a logic micro operation to be executed on the individual bits of the registers provided that the control variable P = 1. As a numerical example, assume that each register has four bits. Let the content of Rl be 1010 and the content of R2 be 1100. The exclusive-OR micro operation stated above symbolizes the following logic computation:

1010	Content of $R1$
1100	Content of $R2$
0110	Content of $R1$ after $P = 1$

There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in Table (5):

TABLE 5 Truth Tables for 16 Functions of Two Variables

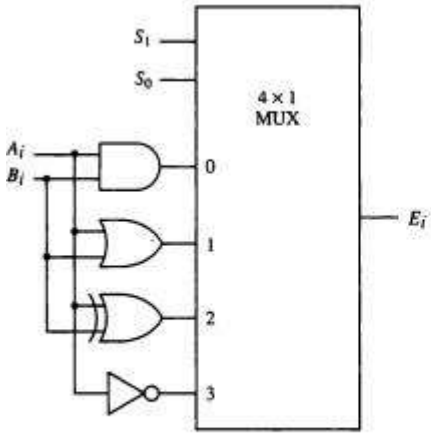
x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table (6):

TABLE 6 Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

The diagram shows (Fig 9-a) one typical stage with subscript i. For a logic circuit with n bits, the diagram must be repeated n times for i = 0,1, 2,..., n - 1. The selection variables are applied to all stages. The function table in Fig.(9-b) lists the logic micro operations obtained for each combination of the selection variables.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \bar{A}$	Complement

(b) Function table

Figure 9 One stage of logic circuit.

Shift Micro operations

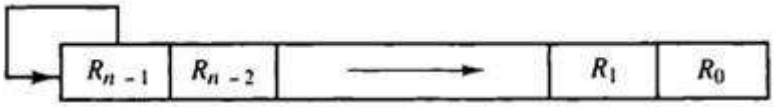
Shift micro operations are used for serial transfer of data. The contents of a register can be shifted to the left or the right. There are three types of shifts: logical, circular, and arithmetic. The symbolic notation for the shift micro operations is shown in Table (7):

TABLE 7 Shift Microoperations

Symbolic designation	Description
$R \leftarrow$ shl R	Shift-left register R
$R \leftarrow$ shr R	Shift-right register R
$R \leftarrow$ cil R	Circular shift-left register R
$R \leftarrow$ cir R	Circular shift-right register R
$R \leftarrow$ ashl R	Arithmetic shift-left R
$R \leftarrow$ ashr R	Arithmetic shift-right R

An arithmetic shift is a micro operation that shifts a signed binary number to the left or right. The arithmetic shift-left inserts a 0 into R_0, and shifts all other bits to the left. The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2} . A sign reversal occurs if the bit in R_{n-1} changes in value after the shift and caused an overflow.

The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right.



Arithmetic shift right.

Ex: If the content of 8 bits register is (10100011). What is the result of the operation after executing to the register:

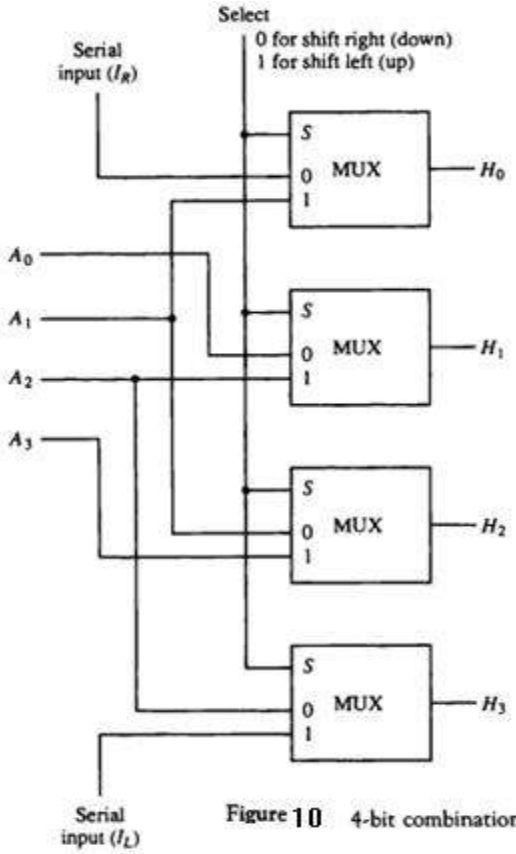
a. shl R: shift left register by 3. b. cil R : circular shift left register by 3.

c. ashl R: arithmetic shift left register by 3. d. ashr R: arithmetic shift right register by 3.

Ans:

(a) 00011000. (b) 00011101. (c) 00011000. Overflow (d) 11110100

A combinational circuit shifter can be constructed with multiplexers as shown in Fig (10). The 4-bit shifter has four data inputs, A_0 through A_3, and four data outputs, H_0 through H_3. There are two serial inputs, one for shift left (I_L) and the other for shift right (I_R).



Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 10 4-bit combinational circuit shifter.

Arithmetic Logic Shift Unit

Computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated ALU. The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig (11) with the functional table(8):

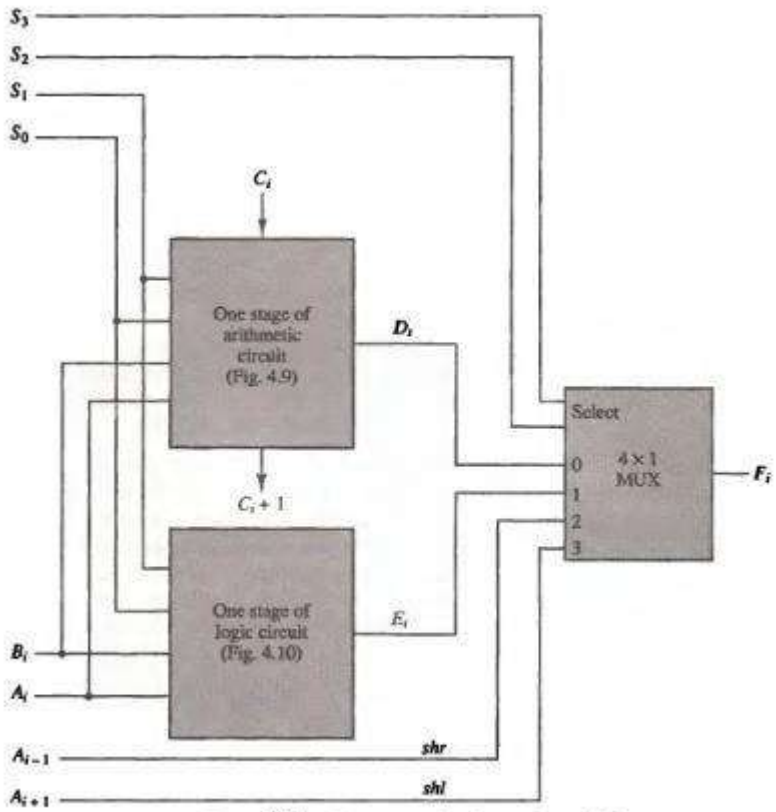


Figure 11 One stage of arithmetic logic shift unit.

TABLE 8 Function Table for Arithmetic Logic Shift Unit

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F =$ shr A	Shift right A into F
1	1	×	×	×	$F =$ shl A	Shift left A into F

The way that the interrupt is handled by the computer can be explained by means of the flowchart of Fig(19). An interrupt flip-flop R is included in the computer.

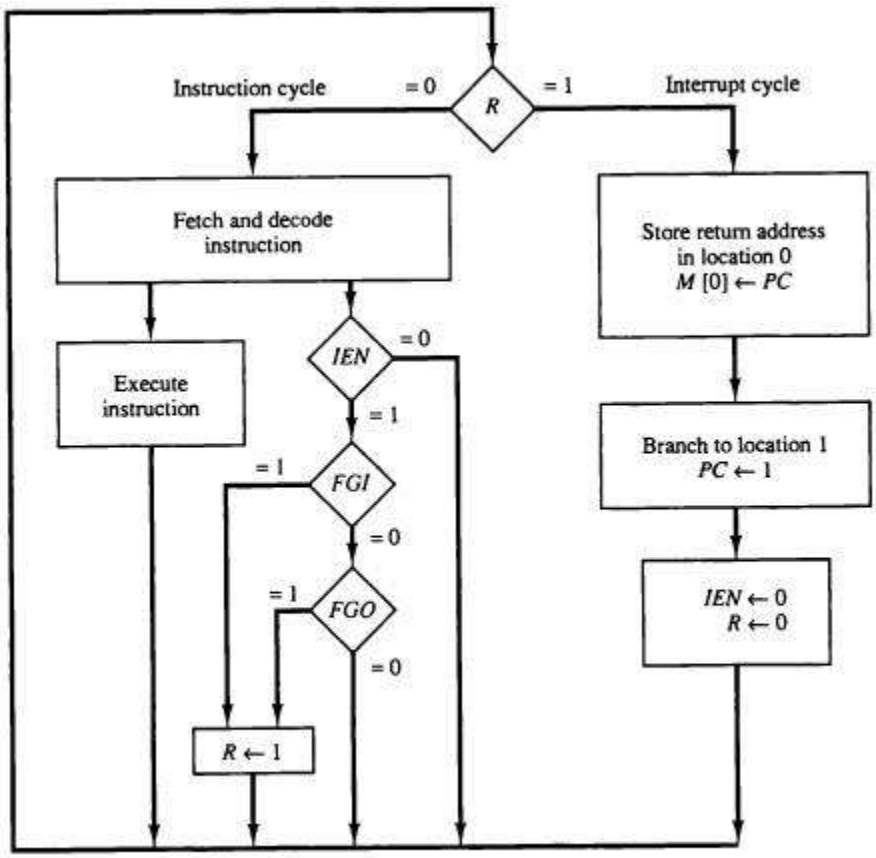


Figure 19 Flowchart for interrupt cycle.

Design of Basic Computer

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 decoders: 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 The outputs 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 S_2, S_1, and S_0 to select a register for the bus
    5. Signals to control the AC adder and logic circuit.

Design of Accumulator Logic

The circuits associated with the AC register are shown in Fig(20). The adder and logic circuit has three sets of inputs.

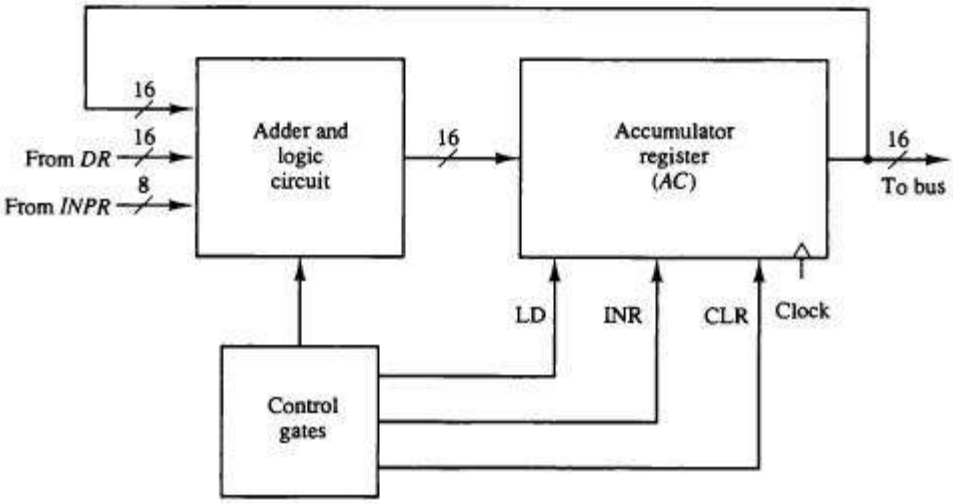


Figure 20 Circuits associated with AC.

In order to design the logic associated with AC, it is necessary to go over the register transfer statements and extract all the statements that change the content of AC.

$AC \leftarrow AC \wedge DR$	AND with *DR*
$AC \leftarrow AC + DR$	Add with *DR*
$AC \leftarrow DR$	Transfer from *DR*
$AC(0\text{–}7) \leftarrow INPR$	Transfer from *INPR*
$AC \leftarrow \overline{AC}$	Complement
$AC \leftarrow \text{shr } AC, \quad AC(15) \leftarrow E$	Shift right
$AC \leftarrow \text{shl } AC, \quad AC(0) \leftarrow E$	Shift left
$AC \leftarrow 0$	Clear
$AC \leftarrow AC + 1$	Increment

Control Memory

The function of the control unit in a digital computer is to initiate sequences of microoperations. The number of different types of microoperations that are available in a given system is finite. A control unit whose binary control variables are stored in memory is called a microprogrammed control unit. Each word in control memory contains within it a microinstruction. The microinstruction specifies one or more microoperations for the system. A sequence of microinstructions constitutes a microprogram.

A computer that employs a microprogrammed control unit will have two separate memories: a main memory and a control memory. The main memory is available to the user for storing the programs. The contents of main memory may alter when the data are manipulated and every time that the program is changed. The user's program in main memory consists of machine

instructions and data. While the control memory holds a fixed microprogram that cannot be altered by the occasional user.

The general configuration of a microprogrammed control unit is demonstrated in the block diagram of Fig(21).

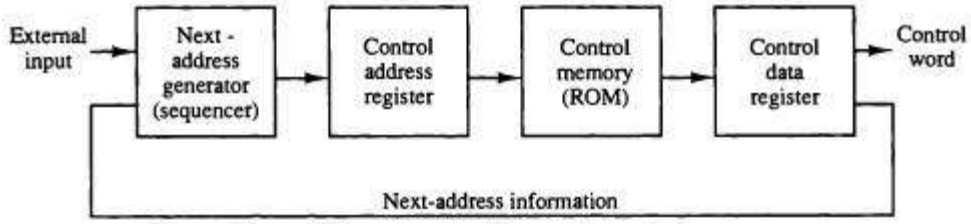


Figure 21 Microprogrammed control organization.

Address Sequencing

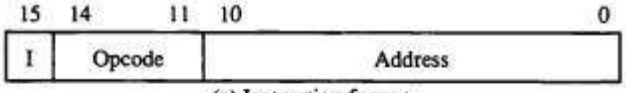
Microinstructions are stored in control memory in groups, with each group routine specifying a routine. An initial address is loaded into the control address register when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine. The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions.

In summary, the address sequencing capabilities required in a control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

Instruction format:

The computer instruction format is depicted in Fig(22-a). It consists of three fields: a 1bit held for indirect addressing symbolized by J, a 4-bit operation code (opcode), and an 11bit address field. Fig(22-b) lists four of the 16 possible memory-reference instructions.



(a) Instruction format

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions

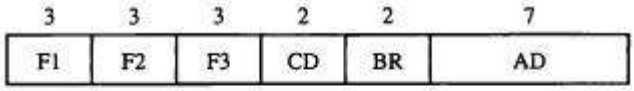
Figure 22 Computer instructions.

Microinstruction Format

The microinstruction format for the control memory is shown in Fig(23). The format 20 bits of the microiristruction are divided into four functional parts. The three fields Fl, F2, and F3 specify microoperations for the computer. The CD field selects status bit conditions. The BR field specifies the type of branch to be used. The AD field contains a branch address. The address field is seven bits wide, since the control memory has $128 = 2^7$ words.

The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microoperations as listed in Table (13). This gives a total of 21 microoperations.

The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table. The first condition is always a 1, so that a reference to CD = 00 (or the symbol U) will always find the condition to be true. When this condition is used in conjunction with the BR (branch) field, it provides an unconditional branch operation. The indirect bit I is available from bit 15 of DR after an instruction is read from memory. The sign bit of AC provides the next status bit.



F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

Figure 23 Microinstruction code format (20 bits).

TABLE 13 Symbols and Binary Code for Microinstruction Fields

F1	Microoperation	Symbol	F2	Microoperation	Symbol	F3	Microoperation	Symbol
000	None	NOP	000	None	NOP	000	None	NOP
001	$AC \leftarrow AC + DR$	ADD	001	$AC \leftarrow AC - DR$	SUB	001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow 0$	CLRAC	010	$AC \leftarrow AC \vee DR$	OR	010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow AC + 1$	INCAC	011	$AC \leftarrow AC \wedge DR$	AND	011	$AC \leftarrow$ shl AC	SHL
100	$AC \leftarrow DR$	DRTAC	100	$DR \leftarrow M[AR]$	READ	100	$AC \leftarrow$ shr AC	SHR
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR	101	$DR \leftarrow AC$	ACTDR	101	$PC \leftarrow PC + 1$	INCPC
110	$AR \leftarrow PC$	PCTAR	110	$DR \leftarrow DR + 1$	INCDR	110	$PC \leftarrow AR$	ARTPC
111	$M[AR] \leftarrow DR$	WRITE	111	$DR(0\text{-}10) \leftarrow PC$	PCTDR	111	Reserved	

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	$DR(15)$	I	Indirect address bit
10	$AC(15)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC

BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1 $CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1 $CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

Design of Control Unit

The Fig(24) shows the three decoders and some of the connections that must be made from their outputs. Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3x8 decoder to provide eight outputs. For example, when Fl = 101 (binary 5), the next clock pulse transition transfers the content of DK(0-10) to AR (symbolized by DRTAR in Table). Similarly, when Fl = 110 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR).

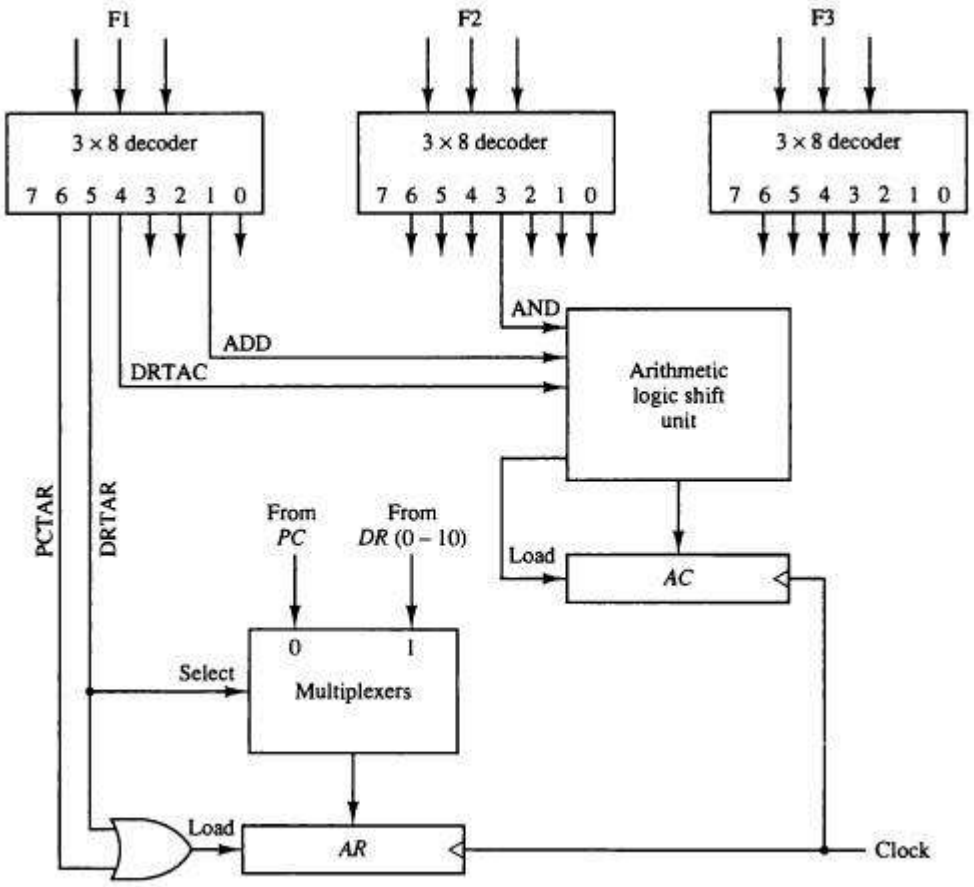


Figure 24 Decoding of microoperation fields.

Central Processing Unit

The CPU is made up of three major parts, as shown in Fig(25).

1- The register set stores intermediate data used during the execution of the instructions. The arithmetic
2- logic unit (ALU) performs the required microoperations for executing the instructions.
3- The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

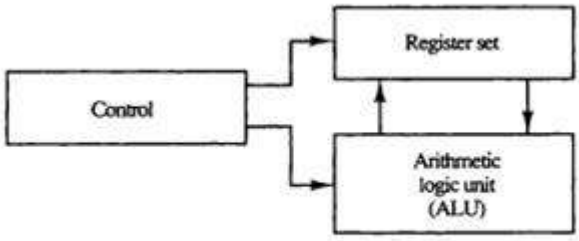


Figure 25 Major components of CPU.

General Register Organization

The memory locations are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplication. A bus organization for seven CPU registers is shown in Fig(26):

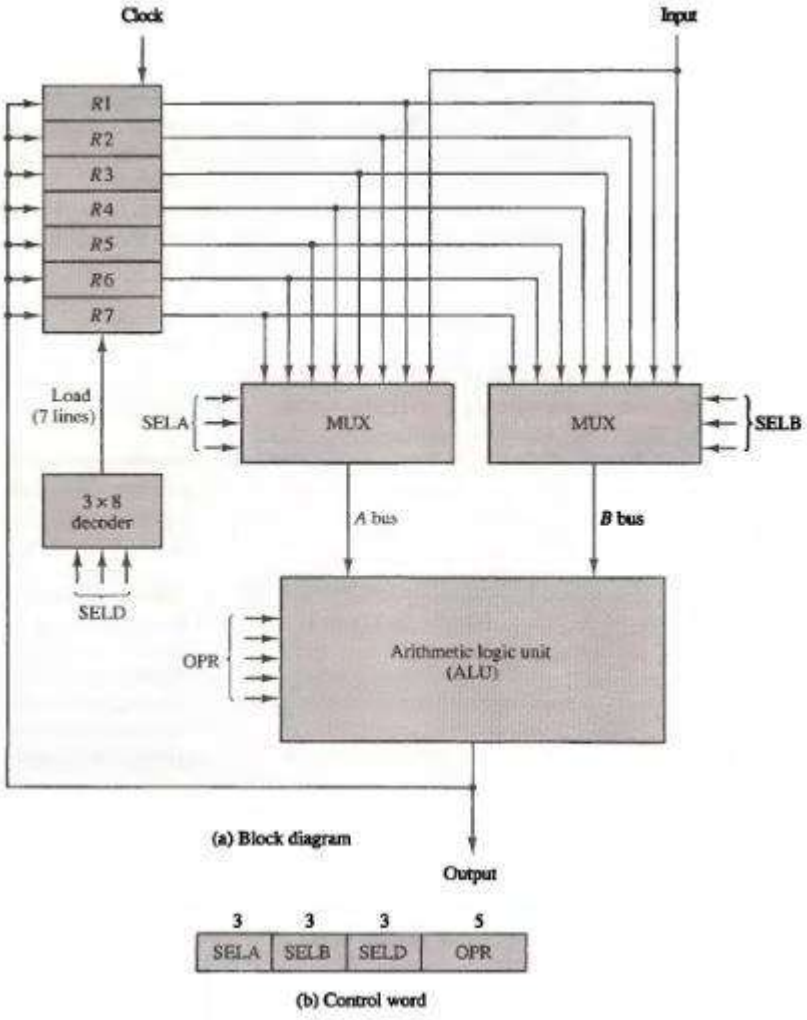


Figure 26 Register set with common ALU.

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 system. For example, to perform the operation:

$$R1 \leftarrow R2 + R3$$

The control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.
2. MUX B selector (SELB): to place the content of R3 into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition A + B.
4. Decoder destination selector (SELD): to transfer the content of the output bus into Rl. To achieve a fast response time, the ALU is constructed with high-speed circuits.

There are 14 binary selection inputs in the unit, and their combined value control word specifies a control word. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits of SELD 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 encoding of the register selections is specified in Table(14):

TABLE 14 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	*R1*	*R1*	*R1*
010	*R2*	*R2*	*R2*
011	*R3*	*R3*	*R3*
100	*R4*	*R4*	*R4*
101	*R5*	*R5*	*R5*
110	*R6*	*R6*	*R6*
111	*R7*	*R7*	*R7*

Table(15) OPR field has five bits and each operation is designated with a symbolic name.

TABLE 15 Encoding of ALU Operations

OPR Select	Operation	Symbol
00000	Transfer *A*	TSFA
00001	Increment *A*	INCA
00010	Add *A* + *B*	ADD
00101	Subtract *A* − *B*	SUB
00110	Decrement *A*	DECA
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

For example, the subtract microoperation given by the statement:

$$R1 \leftarrow R2 - R3$$

The binary control word for the subtract microoperation is 010 011 001 00101 and is obtained as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

Stack Organization

A useful feature that is included in the CPU of most computers is a stack or last-in, firstout (LIFO) list. The two operations of a stack are the insertion and deletion of items. The operation of insertion is called push, while the operation of deletion is called pop. In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$.

The push operation is implemented with the following sequence of microoperations:

$SP \leftarrow SP + 1$	Increment stack pointer
$M[SP] \leftarrow DR$	Write item on top of the stack
If $(SP = 0)$ then $(FULL \leftarrow 1)$	Check if stack is full
$EMTY \leftarrow 0$	Mark the stack not empty

The pop operation consists of the following sequence of microoperations:

$DR \leftarrow M[SP]$	Read item from the top of stack
$SP \leftarrow SP - 1$	Decrement stack pointer
If $(SP = 0)$ then $(EMTY \leftarrow 1)$	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

Instruction Formats

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1. An operation code field that specifies the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined. An example of an accumulator-type organization, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as:

TABLE 16 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

Data Transfer and Manipulation

Most computer instructions can be classified into three categories:

1. Data transfer instructions.
2. Data manipulation instructions.
3. Program control instructions.

Data transfer instructions cause transfer of data from one location to another without changing the binary information content. The table(17) list the Data transfer instructions:

TABLE 17 Typical Data Transfer Instructions

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

Data manipulation instructions are those that perform arithmetic, logic, and shift operations. The data manipulation instructions in a typical computer are usually divided into three basic types:

1- Arithmetic instructions.

2. Logical and bit manipulation instructions.

3. Shift instructions.

Computer Architecture Part2

Asst. Prof. Dr. Raheem Abdul Sahib

2023-2024

Reduced Instruction Set Computer (RISC)

An important aspect of computer architecture is the design of the instruction set for the processor. The instruction set chosen for a particular computer determines the way that machine language programs are constructed. A computer with a large number of instructions is classified as a Complex Instruction Set Computer, abbreviated CISC. In the early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often.

The RISC (Reduced Instruction Set Computer) type of computer is classified as a reduced instruction set computer or RISC.

In Summary, The Major Characteristics of CISC Architecture Are:

1. A large number of instructions—typically from 100 to 250 instructions.
2. Some instructions that perform specialized tasks and are used infrequently.
3. A large variety of addressing modes—typically from 5 to 20 different modes.
4. Variable-length instruction formats.
5. Instructions that manipulate operands in memory.

The Major Characteristics of A RISC Processor Are:

1. Relatively few instructions.
2. Relatively few addressing modes.
3. Memory access limited to load and store instructions.
4. All operations done within the registers of the CPU.
5. Fixed-length, easily decoded instruction format.
6. Single-cycle instruction execution.
7. Hardwired rather than microprogrammed control.

Memory Hierarchy

The memory unit is an essential component in any digital computer since it is needed for storing programs and data. The memory unit that communicates directly with the CPU is called the main memory. Devices that provide backup storage are called auxiliary memory. They are used for storing system programs, large data files, and other backup information. Only programs and data currently needed by the processor reside in main memory. All other information is stored in auxiliary memory and transferred to main memory when needed. A special very-high-

speed memory called a cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. Fig(29) shows the Memory Hierarchy:

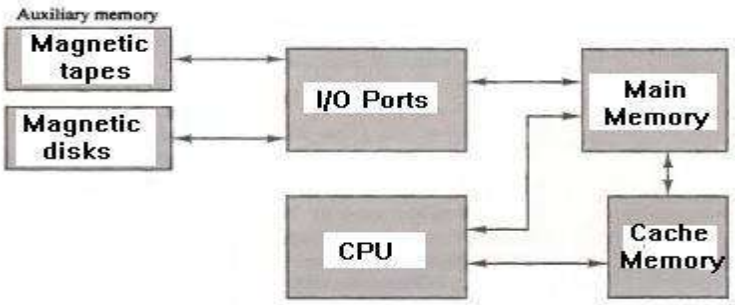


Figure 29 Memory hierarchy in a computer system.

Main Memory The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible operating modes:

The static RAM consists essentially of internal flip-flops that store the binary information. *The dynamic RAM* stores the binary information in the form of electric charges that are applied to capacitors.

Associative Memory

Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent.

A memory unit accessed by content is called an associative memory or Content Addressable Memory (CAM). When a word is written in an associative memory is capable of finding an empty unused location to store the word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locates all words which match the specified content and marks them for reading. The block diagram of an associative memory is shown in Fig (30):

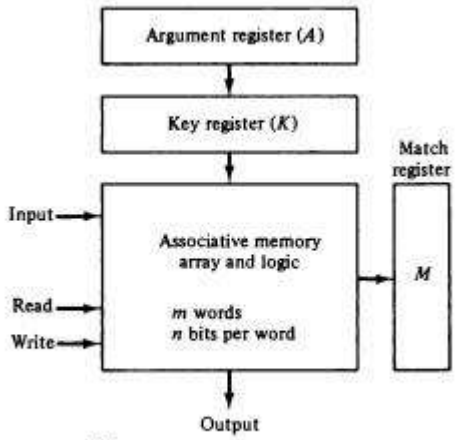


Figure 30 Block diagram of associative memory.

To illustrate with a numerical example, suppose that the argument register *A* and the key register *K* have the bit configuration shown below. Only the three left most bits of *A* are compared with memory words because K has l's in these positions.

A	101 111100	
K	111 000000	
Word 1	100 111100	no match
Word 2	101 000001	match

Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

Cache Memory

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a cache memory. It is placed between the CPU and main memory.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. The performance of cache memory is frequently measured in terms of a quantity called *hit ratio*. When the CPU refers to memory and finds the word in cache, it is said to produce a *hit*. If the word is not found in cache, it is in main memory and it counts as a *miss*.

Three types of mapping procedures are of practical interest when considering the organization of cache memory:

1. Associative mapping
2. Direct mapping
3. Set-associative mapping

Virtual Memory

Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal(تصرف), even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program generated addresses into correct main memory locations.

As an illustration, consider a computer with a main-memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since 32K = 2^{15}. Suppose that the computer has available auxiliary memory for storing 2^{20} = 1024K words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories. Denoting the address space by N and the memory space by M, we then have for this example N = 1024K and M = 32K.

The mapping table may be stored in a separate memory as shown in Fig (31) or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In the second case, the table takes space from main memory and two accesses to memory are required with the program running at half speed.

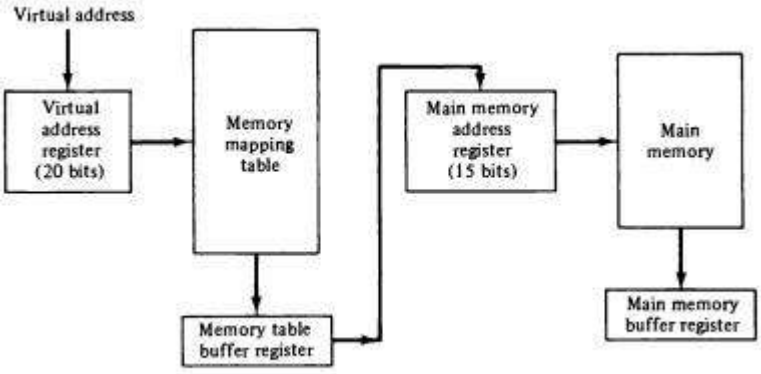


Figure 31 Memory table for mapping a virtual address.

The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size pages and blocks called blocks, which may range from 64 to 4096 words each. The term page refers to groups of address space of the same size. For example, if a page or block consists of IK words, then, using the previous example, address space is divided into 1024 pages and main memory is divided into 32 blocks.

The organization of the memory mapping table in a paged system is shown in Fig(32). The memory-page table consists of eight words, one for each page. The address in the page table denotes the page number and the content of the word gives the block number where that page is stored in main memory. The table shows that pages 1, 2, 5, and 6 are now available in main memory in blocks 0, 1, 2, and 3, respectively. A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory. A_0 in the presence bit indicates that this page is not available in main memory.

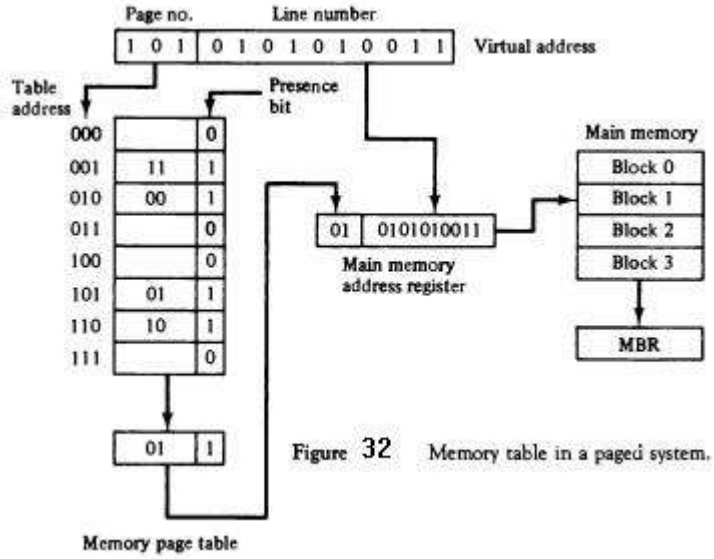


Figure 32 Memory table in a paged system.

Memory Management Hardware

A memory management system is a collection of hardware and software procedures for managing the various programs residing in memory. The memory management software is part of an overall operating system available in many computers.

The basic components of a memory management unit are:

1. A facility for dynamic storage relocation that maps logical memory references into physical memory addresses.
2. A provision for sharing common programs stored in memory by different users.
3. Protection of information against unauthorized access between users and preventing users from changing operating system functions.

The fixed page size used in the virtual memory system causes certain difficulties with respect to program size and the logical structure of programs. It is more convenient to divide programs and segment data into logical parts called segments.

A segment is a set of logically related instructions or data elements associated with a given name. Segments may be generated by the programmer or by the operating system. Examples of segments are a subroutine, an array of data, a table of symbols, or a user's program. The address generated by a segmented program is called a logical address. The logical address may be larger than the physical memory address as in virtual memory, but it may also be equal, and sometimes even smaller than the length of the physical memory address.

Numerical Example: A numerical example may clarify the operation of the memory management unit. Consider the 20-bit logical address specified in Fig(33-a). This configuration allows each segment to have any number of pages up to 256. The smallest possible segment will have one page or 256 words. The largest possible segment will have 256 pages, for a total of 256 x 256 = 64K words. The physical memory shown in Fig(33-b).

Paging splits the address space into equal sized units called pages.

While segmentation splits the memory into unequal units that may have sizes more meaningful or appropriate to the program.

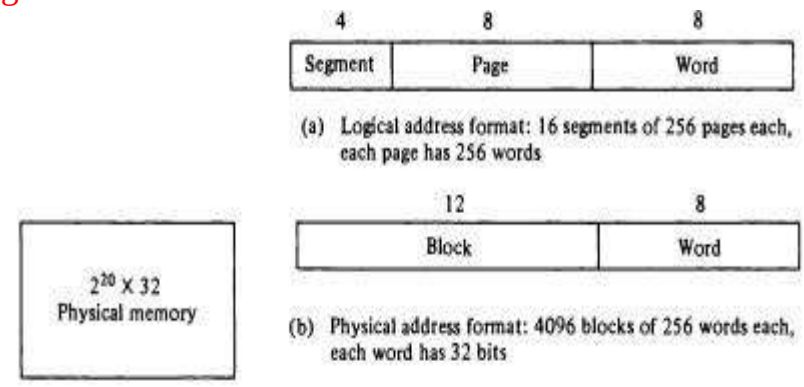


Figure 33 An example of logical and physical addresses.

Consider a program loaded into memory that requires five pages. The operating system may assign to this program segment 6 and pages 0 through 4, as shown in Fig(34-a). The total logical address range for the program is from hexadecimal 60000 to 604FF. The correspondence

between each memory block and logical page number is then entered in a table as shown in Fig(34-b).

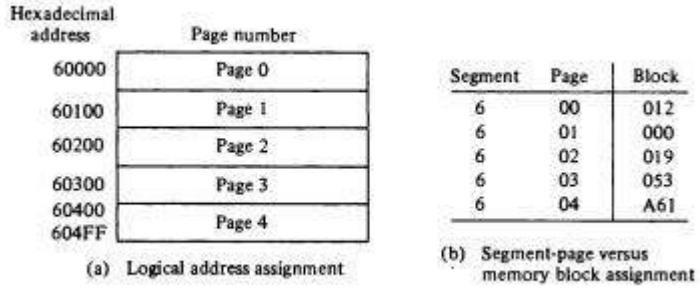
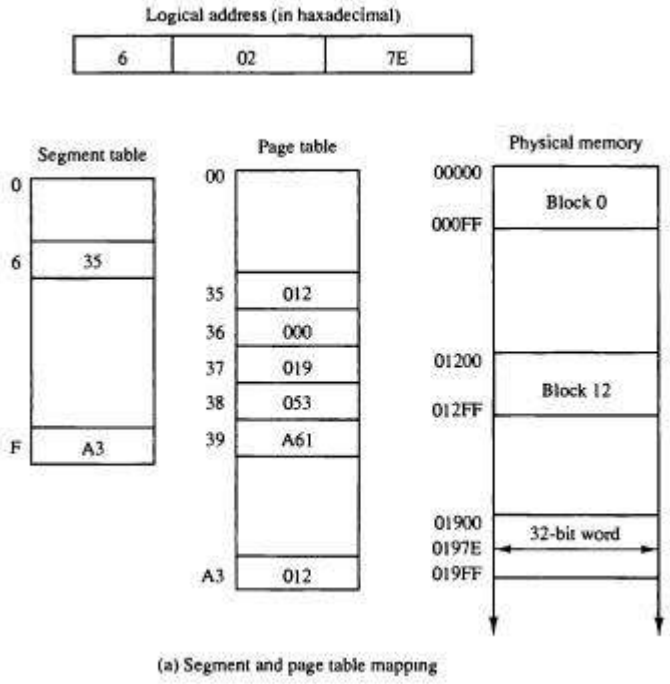


Figure 34 Example of logical and physical memory address assignment.

The information from this table is entered in the segment and page tables as shown in Fig(35-a). Now consider the specific logical address given in Fig(35). The 20-bit address is listed as a five-digit hexadecimal number. It refers to word number 7E of page 2 in segment 6. The base of segment 6 in the page table is at address 35. Segment 6 has associated with it five pages, as shown in the page table at addresses 35 through 39. Page 2 of segment 6 is at address 35 + 2 = 37. The physical memory block is found in the page table to be 019. Word 7E in block 19 gives the 20-bit physical address 0197E. Note that page 0 of segment 6 maps into block 12 and page 1 maps into block 0. The associative memory in Fig(35-b) shows that pages 2 and 4 of segment 6 have been referenced previously and therefore their corresponding block numbers are stored in the associative memory.



(a) Segment and page table mapping

Continue

Segment	Page	Block
6	02	019
6	04	A61

(b) Associative memory (TLB)

Figure 35 Logical to physical memory mapping example (all numbers are in hexadecimal).

Input-Output Organization

The input-output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the outside environment. Programs and data must be entered into computer memory for processing and results obtained from computations must be recorded or displayed for the user.

Peripheral Devices

Input or output devices attached to the computer are also called peripherals.

- The display terminal can operate in a single-character mode where all characters entered on the screen through the keyboard are transmitted to the computer simultaneously. In the block mode, the edited text is first stored in a local memory inside the terminal. The text is transferred to the computer as a block of data.
- Printers provide a permanent record on paper of computer output data.

- Magnetic tapes are used mostly for storing files of data.
- Magnetic disks have high-speed rotational surfaces coated with magnetic material.

Input-Output Interface

Input-output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The major differences are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.
2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.
3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

A typical communication link between the processor and several peripherals is shown in Fig.36. The I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The interface selected responds to the function code and proceeds to execute it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit.

There are three ways that computer buses can be used to communicate with memory and I/O:

1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and I/O but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

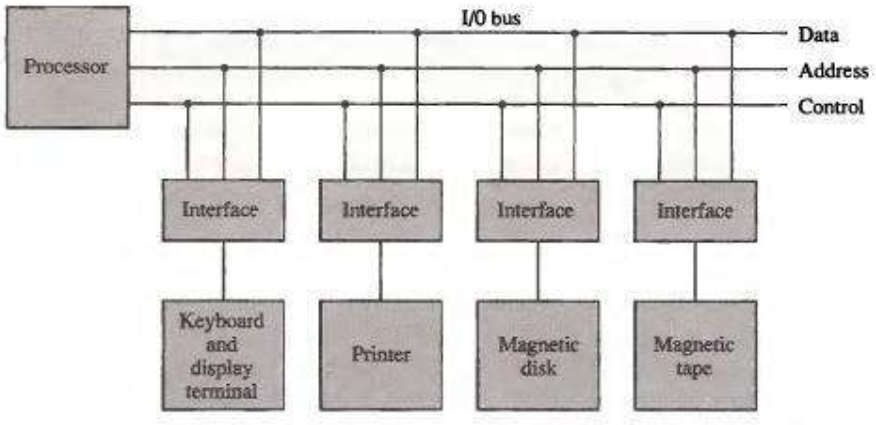
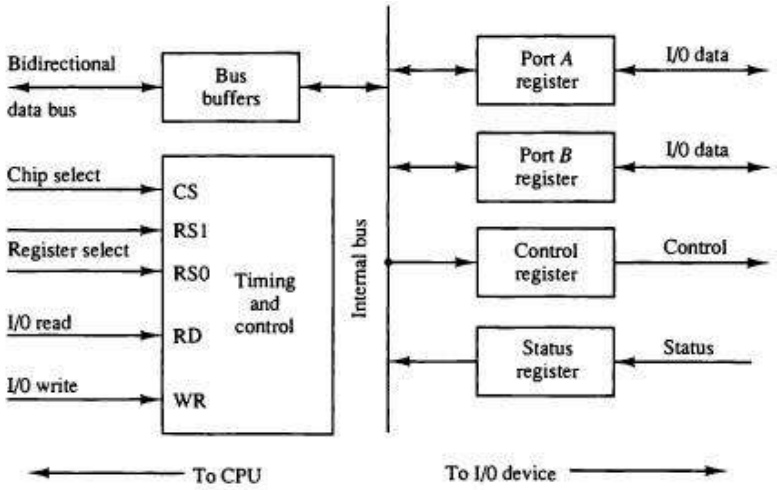


Figure 36 Connection of I/O bus to input-output devices.

Isolated I/O versus Memory-Mapped I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. In the isolated I/O configuration, the CPU has distinct input and output instructions, and

each of these instructions is associated with the address of an interface register. The isolated I/O method isolates memory and I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same address space for both memory and I/O.



CS	RS1	RS0	Register selected
0	×	×	None: data bus in high-impedance
1	0	0	Port *A* register
1	0	1	Port *B* register
1	1	0	Control register
1	1	1	Status register

Figure 37 Example of I/O interface unit.

This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as memory mapped I/O. In a memory-mapped I/O organization there is no specific input or output instructions. Computers with memory-mapped I/O can use memory-type instructions to access I/O data.

An example of an I/O interface unit is shown in block diagram form in Fig.37. It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.

Asynchronous Data Transfer

The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other. This

approach is widely used in most computer systems. Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted. Two way of achieving this:

- The strobe: pulse supplied by one of the units to indicate to the other unit when the transfer has to occur.
- The handshaking: The unit receiving the data item responds with another control signal to acknowledge receipt of the data.

The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers.

The strobe may be activated by either the source or the destination unit. Figure 38 shows a source-initiated transfer and the timing diagram.

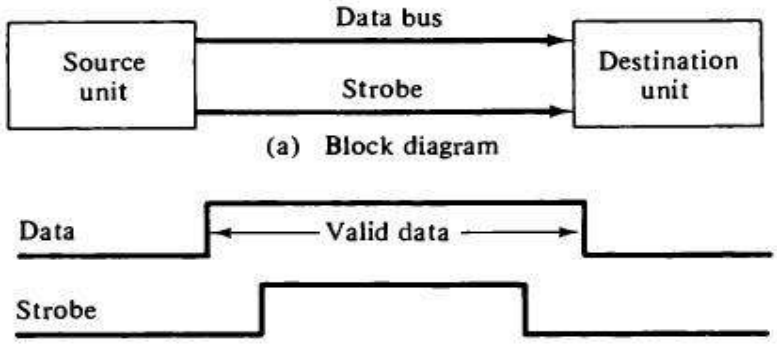


Figure 38 Source-initiated strobe for data transfer.

Fig.39 shows the strobe of a memory-read control signal from the CPU to a memory.

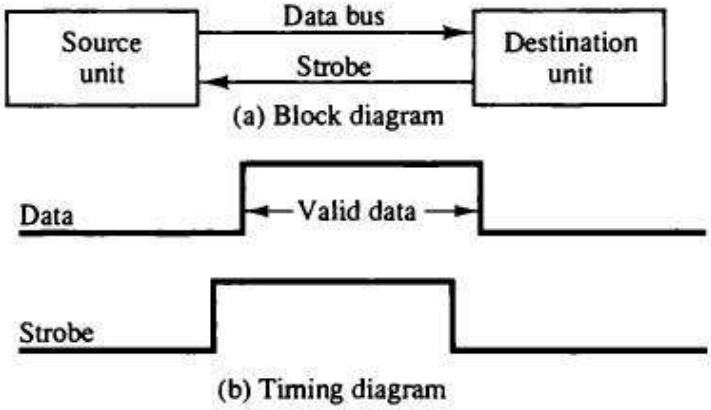


Figure 39 Destination-initiated strobe for data transfer.

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that two-wire control initiates the transfer.

Figure 40 shows the data transfer procedure when initiated by the source. The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the

destination unit. The timing diagram shows the exchange of signals between the two units. Figure 41 the destination-initiated transfer using handshaking lines. Note that the name of the signal generated by the destination unit has been changed to ready for data to reflect its new meaning.

Asynchronous Serial Transfer

The transfer of data between two units may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time. This means that an w-bit message must be transmitted through n separate conductor paths. In serial data transmission, each bit in the message is sent in sequence one at a time. This method requires the use of one pair of conductors or one conductor and a common ground. Parallel transmission is faster but requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but is less expensive since it requires only one pair of conductors. Serial transmission can be synchronous or asynchronous. A transmitted character can be detected by the receiver from knowledge of the transmission rules:

1. When a character is not being sent, the line is kept in the 1-state.
2. The initiation of a character transmission is detected from the start bit, which is always(0).
3. The character bits always follow the start bit.
4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.

Modes of Transfer

Data transfer between the central computer and I/O devices may be handled in a variety of modes. three possible modes:

1. Programmed I/O: The operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. The CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. An example of data transfer from an I/O device through an interface into the CPU is shown in Fig. 43.

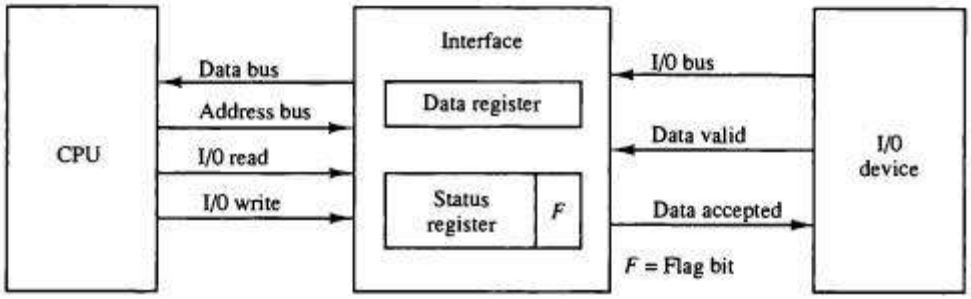


Figure 43 Data transfer from I/O device to CPU.

2. Interrupt-initiated I/O: It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime the CPU can proceed to execute another program. This method of connection between three devices and the CPU is shown in Fig. 44.

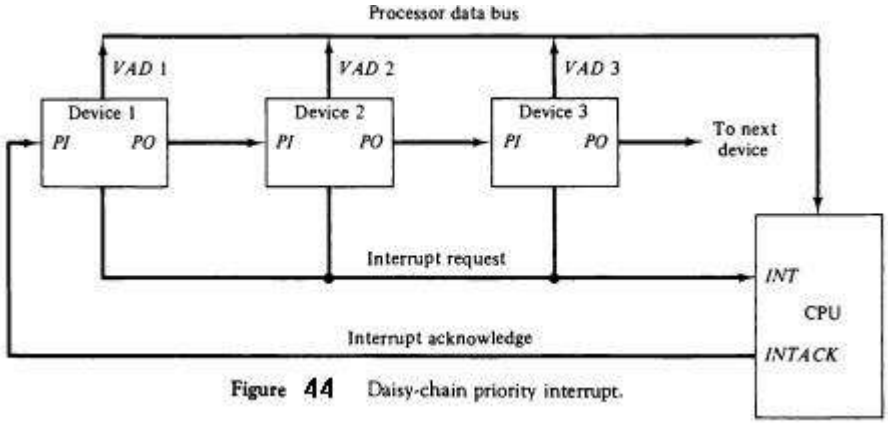


Figure 44 Daisy-chain priority interrupt.

3. Direct memory access (DMA): the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. This method of connection between devices and the memory is shown in Fig. 45.

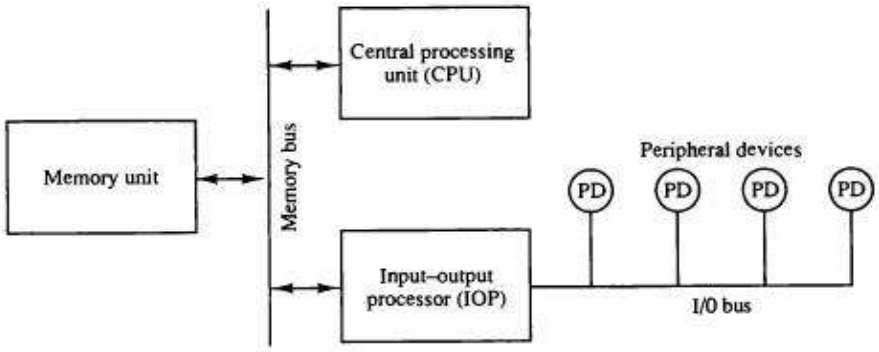


Figure 45 Block diagram of a computer with I/O processor.

Pipelining

Pipelining is a technique of decomposing a sequential process into sub-operations; with each sub-process being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows.

General Considerations

Any operation that can be decomposed into a sequence of sub-operations of about the same complexity can be implemented by a pipeline processor. The general structure of a foursegment pipeline is illustrated in Fig. 46. The operands pass through all four segments in a fixed sequence.

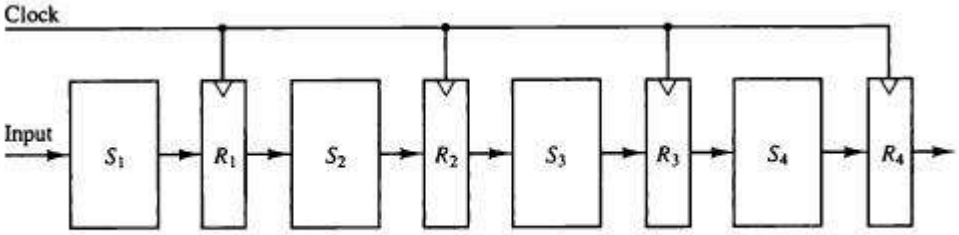


Figure 46 Four-segment pipeline.

The space-time diagram of a four-segment pipeline is demonstrated in Fig47.

Segment:	1	2	3	4	5	6	7	8	9
1	T_1	T_2	T_3	T_4	T_5	T_6			
2		T_1	T_2	T_3	T_4	T_5	T_6		
3			T_1	T_2	T_3	T_4	T_5	T_6	
4				T_1	T_2	T_3	T_4	T_5	T_6

Clock cycles

Figui 47 Space-time diagram for pipeline.

The speedup(S) of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio:

$$S = \frac{nt_n}{(k+n-1)t_p}$$

As the number of tasks increases, n becomes much larger than $k - 1$, and $k + n - 1$ approaches the value of n. Under this condition, the speedup becomes:

$$S = \frac{t_n}{t_p}$$

numerical example: Let the time it takes to process a sub-operation in each segment be equal to t_p= 20 ns. Assume that the pipeline has $k = 4$ segments and executes $n = 100$ tasks in sequence. The pipeline system will take

$$(k + n - 1)t_p = (4 + 99) \times 20 = 2060ns$$

to complete. Assuming that t = ktp = 4 x 20 = 80 ns, a

non-pipeline system requires:

$$nkt_p = 100 \times 80 = 8000ns$$

to complete the 100 tasks. The speedup ratio is equal to:

$$8000/2060 = 3.88$$

Instruction Pipeline

The computer needs to process each instruction with the following sequence of steps:

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

Figure 48 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3.

The four segments are represented in the flowchart:

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

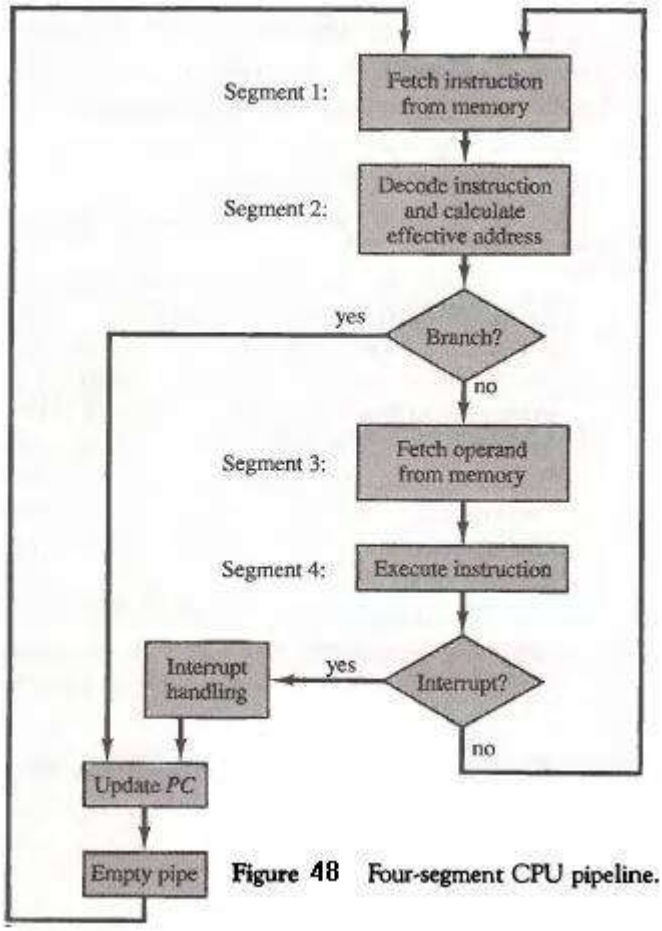


Figure 48 Four-segment CPU pipeline.

A pipeline operation is said to have been stalled if one unit (stage) requires more time to perform its function, thus forcing other stages to become idle. Consider, for example, the case of an instruction fetch that incurs a cache miss. Assume also that a cache miss requires three extra time units.

Instruction-Level Parallelism

Contrary to pipeline techniques, instruction-level parallelism (ILP) is based on the idea of multiple issue processors (MIP). An MIP has multiple pipelined datapaths for instruction execution. Each of these pipelines can issue and execute one instruction per cycle. Figure 49 shows the case of a processor having three pipes. For comparison purposes, we also show in the same figure the sequential and the single pipeline case.

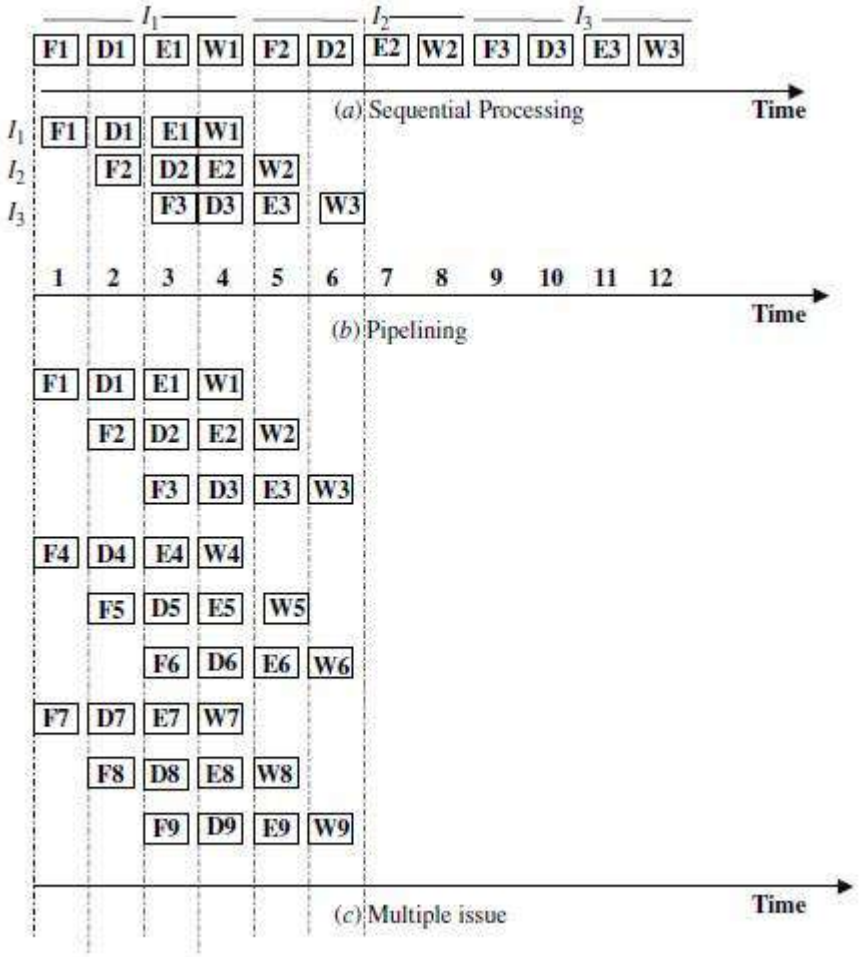


Figure 49 Multiple issue versus pipelining versus sequential processing

Arithmetic Pipeline

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems.

an example of a pipeline unit for floating-point addition and subtraction. The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$$X = A \times 2^a$$

$$Y = B \times 2^b$$

A, B are two fractions that represent the mantissas and a, b are the exponents. The suboperations that are performed in the four segments are:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas.

4. Normalize the result.

Numerical example may clarify the sub-operations performed in each segment. For simplicity, we use decimal numbers, although Fig.49 refers to binary numbers. Consider the two normalized floating-point numbers:

$$X = 0.9504 \times 10^3$$

$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to obtain$(3 - 2 = 1)$. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain:

$$X = 0.9504 \times 10^3$$

$$Y = 0.0820 \times 10^3$$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum:

$$Z = 1.0324 \times 10^3$$

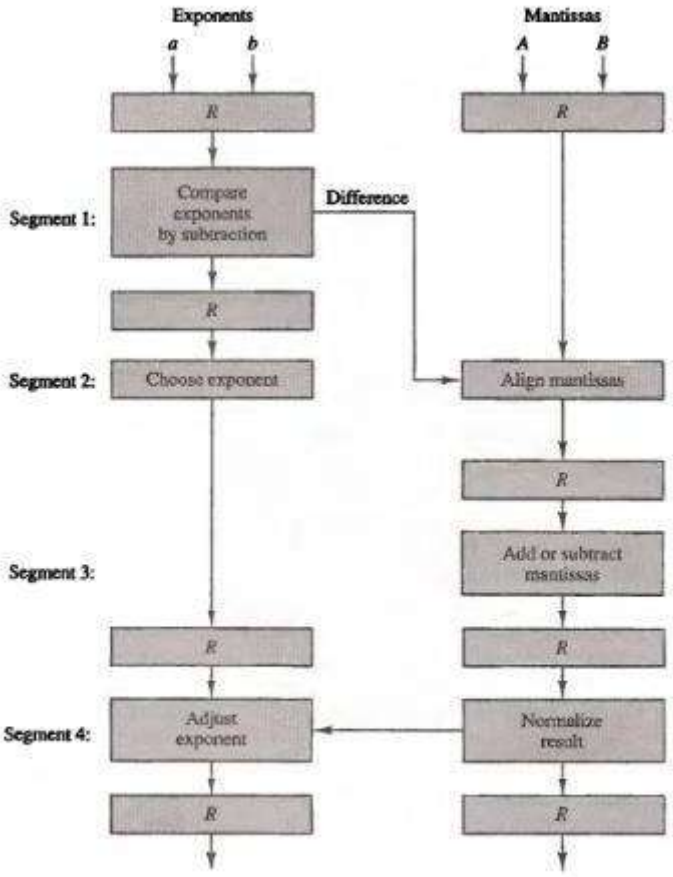


Figure 49 Pipeline for floating-point addition and subtraction.

Suppose that the time delays of the four segments are $t_1 = 60ns$, $t_2 = 70ns$, $t_3 = 100ns$, $t_4 = 80ns$, and the interface registers have a delay of $t_r = 10ns$. The clock cycle is chosen to be $t_p = t_3 + t_r = 110ns$. An equivalent non-pipeline floating point adder-subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320ns$. In this case the pipelined adder has a speedup of 320/110 = 2.9 over the non-pipelined adder.

Supercomputers

Supercomputers are very powerful, high-performance machines used mostly for scientific computations. To speed up the operation, the components are packed tightly together to minimize the distance that the electronic signals have to travel. Supercomputers also use special techniques for removing the heat from circuits to prevent them from burning up because of their close proximity.

A supercomputer is a computer system best known for its high computational speed, fast and large memory systems, and the extensive use of parallel processing.

Delayed Branch

Consider now the operation of the following four instructions:

1. LOAD: $R1 \leftarrow M$[address 1]
2. LOAD: $R2 \leftarrow M$[address 2]
3. ADD: $R3 \leftarrow R1 + R2$
4. STORE: M[address 3] $\leftarrow R3$

If the three-segment pipeline proceeds: (I: Instruction fetch, A:ALU operation, and E: Execute instruction) without interruptions, there will be a data conflict in instruction 3 because the operand in R2 is not yet available in the A segment. This can be seen from the timing of the pipeline shown in Fig. 50(a). The E segment in clock cycle 4 is in a process of placing the memory data into R2. The A segment in clock cycle 4 is using the data from R2, but the value in R2 will not be the correct value since it has not yet been transferred from memory. It is up to the compiler to make sure that the instruction following the load instruction uses the data fetched from memory. It was shown in Fig. 50 that a branch instruction delays the pipeline operation by NOP instruction until the instruction at the branch address is fetched.

Clock cycles:	1	2	3	4	5	6
1. Load $R1$	I	A	E			
2. Load $R2$		I	A	E		
3. Add $R1 + R2$			I	A	E	
4. Store $R3$				I	A	E

(a) Pipeline timing with data conflict

Clock cycle:	1	2	3	4	5	6	7
1. Load $R1$	I	A	E				
2. Load $R2$		I	A	E			
3. No-operation			I	A	E		
4. Add $R1 + R2$				I	A	E	
5. Store $R3$					I	A	E

(b) Pipeline timing with delayed load

Figure 50 Three-segment pipeline timing.

Computer Arithmetic

Arithmetic instructions in digital computers manipulate data to produce results necessary for the solution of computational problems. An arithmetic processor is the part of a processor unit that executes arithmetic operations. The data type assumed to reside in processor registers during the execution of an arithmetic instruction is specified in the definition of the instruction. The solution to any problem that is stated by a finite number of well-defined procedural steps is called an algorithm.

Addition and Subtraction with Signed-Magnitude Data: We designate the magnitude of the two numbers by A and B. When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 18. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

TABLE 18 Addition and Subtraction of Signed-Magnitude Numbers

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

Hardware Implementation: Let A and B be two registers that hold the magnitudes of the numbers, and As and Bs be two flip-flops that hold the corresponding signs. Consider now the hardware implementation of the algorithms above:

1- First, a parallel-adder is needed to perform the microoperation A + B.
2- Second, a comparator circuit is needed to establish if A > B, A = B, or A < B.
3- Third, two parallel-subtractor circuits are needed to perform the microoperations (A-B) and (B-A).
4- The sign relationship can be determined from an exclusive- OR gate with A_s and B_s as inputs.

Careful investigation of the alternatives reveals that the use of 2's complement for subtraction and comparison is an efficient procedure that requires only an adder and a complementer. Figure 51 shows a block diagram of the hardware for implementing the addition and subtraction operations. It consists of registers A and B and sign flip-flops A_s and B_s. Subtraction is done by adding A to the 2's complement of B. The output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitudes of the two numbers. The add-overflow flip-flop AVF holds the overflow bit when A and B are added.

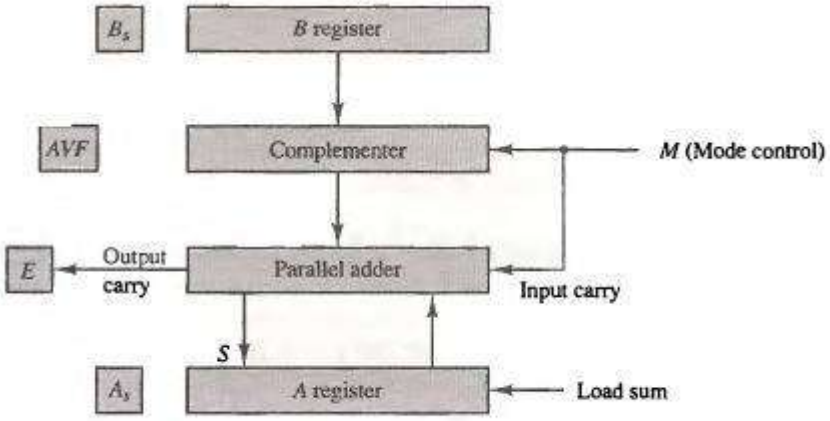


Figure 51 Hardware for signed-magnitude addition and subtraction.

The adder is equal to the sum A + B. When M = 1, the l's complement of B is applied to the adder, the input carry is 1, and output S = A + B +1. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction A - B. The signed 2's complement representation of numbers together with arithmetic algorithms for addition and subtraction are introduced as: The leftmost bit of a binary number represents the sign bit: 0 for positive and 1 for negative. If the sign bit is 1, the entire number is represented in 2's complement form. Thus +33 is represented as 00100001 and -33 as 11011111. Note that 11011111 is the 2's complement of 00100001, and vice versa. *The addition of two numbers in signed 2's complement form consists of adding the numbers with the sign bits treated the same as the other bits of the number.* A carry-out of the sign-bit position is discarded. The subtraction consists of first taking the 2's complement of the subtrahend and then adding it to the minuend.

Multiplication Algorithms

Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive shift and add operations. This process is best illustrated with a numerical example:

```
23      10111     Multiplicand
19    × 10011     Multiplier
        10111
       10111
      00000   +
     00000
    10111
437 110110101     Product
```

Figure 52 is a flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s, respectively. ***The signs are compared, and both A and Q are set to correspond to the sign of the product*** since a doublelength product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier.

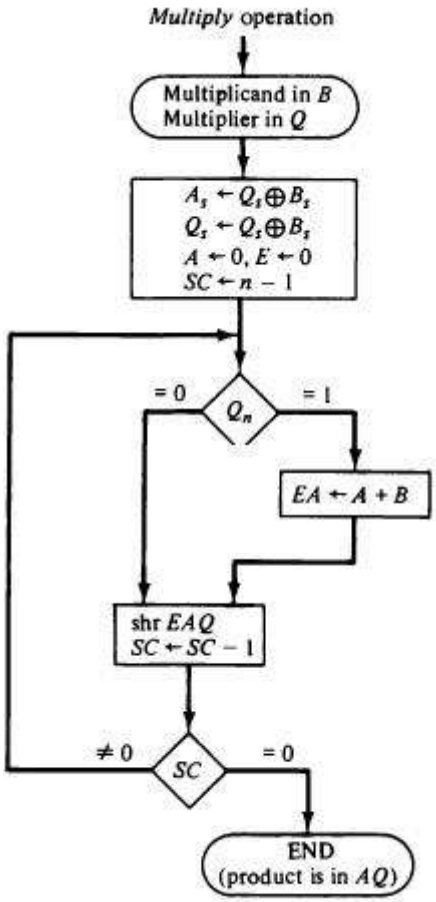


Figure 52 Flowchart for multiply operation.

The numerical example is repeated to clarify the hardware multiplication process. It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, while string of

1's in the multiplier require addition with shifting. The table 19 illustrate numerical example for multiplier 23 (which in binary equal 10111) by 19 (which binary equal 10011) gives the result 437(in binary equal 0110110101).

TABLE 19 Numerical Example for Binary Multiplier

Multiplicand $B = 10111$	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in $AQ = 0110110101$				

Division Algorithms

Division of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations. Binary division is simpler than decimal division because the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is illustrated by a numerical example in Figure 52.

```
Divisor:                    11010     Quotient = Q
B = 10001  )0111000000                Dividend = A
           01110                      5 bits of A < B, quotient has 5 bits
           011100                     6 bits of A ≥ B
           -10001                     Shift right B and subtract; enter 1 in Q
           -010110                    7 bits of remainder ≥ B
           --10001                    Shift right B and subtract; enter 1 in Q
           --001010                   Remainder < B; enter 0 in Q; shift right B
           ---010100                  Remainder ≥ B
           ----10001                  Shift right B and subtract; enter 1 in Q
           ----000110                 Remainder < B; enter 0 in Q
           -----00110                 Final remainder
```

Figure 52 Example of binary division.

The hardware for implementing the division operation is identical to that required for multiplication and consists of the components Register EAQ is now shifted to the left with 0

inserted into Q, and the previous value of E lost. The numerical example is repeated as in Figure 53:

Divisor $B = 10001$, $\bar{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\bar{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\bar{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\bar{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\bar{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\bar{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Figure 53 Example of binary division with digital hardware.

Decimal Arithmetic Unit

To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal. It can add or subtract decimal numbers, usually by forming the 9's or 10's complement of the subtrahend. Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in Fig. 54. The two decimal digits, together with the inputcarry, are first added in the top 4-bit binary adder to produce the binary sum. When the outputcarry is equal to 0, nothing is added to the binary sum.

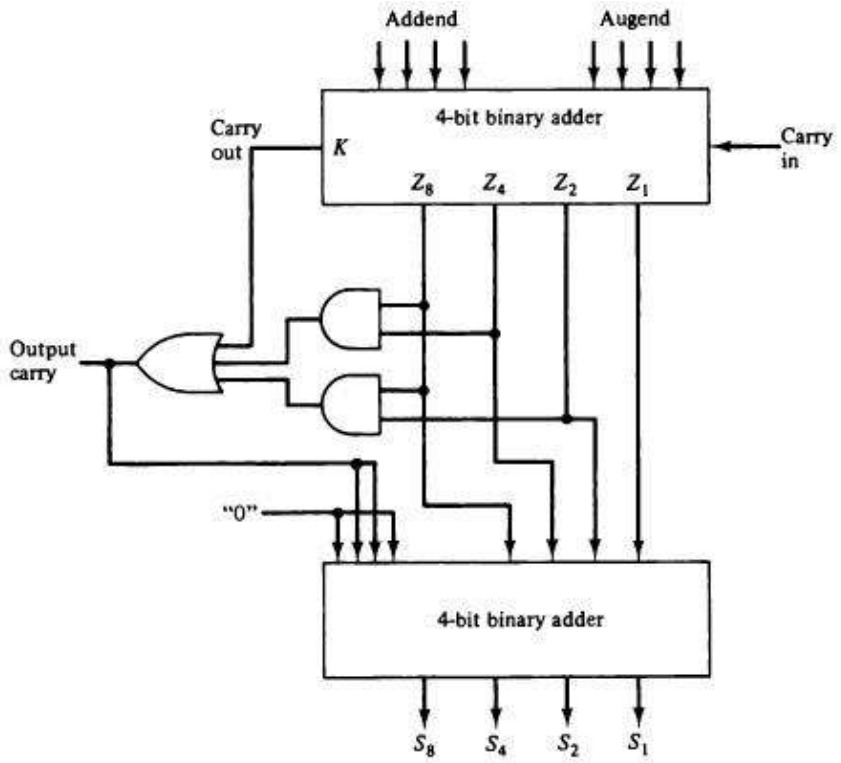


Figure 54 Block diagram of BCD adder.

A straight subtraction of two decimal numbers will require a subtractor circuit that will be somewhat different from a BCD adder. The 9's complement of a decimal digit represented in BCD may be obtained by complementing the bits in the coded representation of the digit provided a correction is included. There are two possible correction methods. In the first method, *binary 1010 (decimal 10) is added to each complemented digit and the carry discarded after each addition*. In the second method, *binary 0110 (decimal 6) is added before the digit is complemented.*

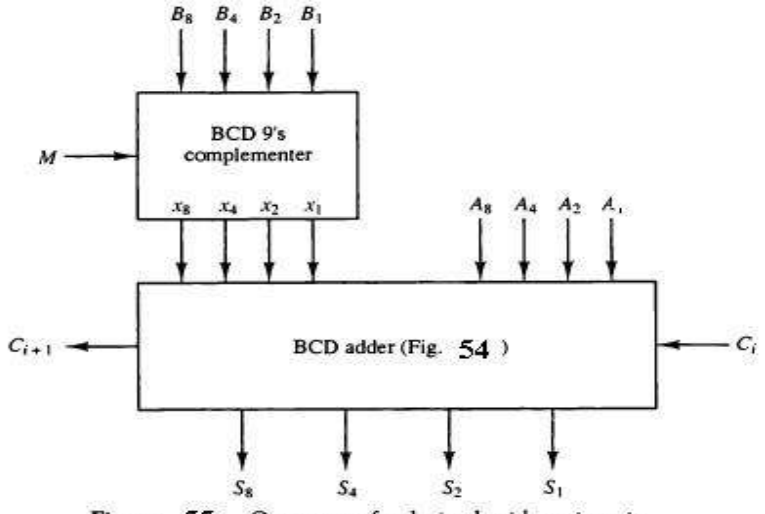


Figure 55 One stage of a decimal arithmetic unit.

One stage of a decimal arithmetic unit that can add or subtract two BCD digits is shown in Fig. 55. It consists of a BCD adder and a 9's complementer. The mode M controls the operation of

the unit. With M = 0, the S outputs form the sum of A and B. With M = 1, the S outputs form the sum of A plus the 9's complement of B. For numbers with n decimal digits we need n such stages. The output carry C_{i+1} from one stage must be connected to the input carry C_i of the next-higher-order stage. The best way to subtract the two decimal numbers is to let M = 1 and apply a 1 to the input carry C_i of the first stage. The outputs will form the sum of A plus the 10's complement of B, which is equivalent to a subtraction operation if the carry-out of the last stage is discarded.

As a numerical illustration, the 9's complement of BCD 0111 (decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 and discarding the carry, we obtain 0010 (decimal 2). By the second method, we add 0110 to 0111 to obtain 1101. Complementing each bit, we obtain the required result of 0010. One stage of a decimal arithmetic unit that can add or subtract two BCD digits is shown in Figure 55. It consists of a BCD adder and a 9's complementer.

Reduced Instruction Set Computers (RISCs)

The RISC approach is RISC-based machines are reality and they are characterized by a number of common features such as simple and reduced instruction set, fixed instruction format, one instruction per machine cycle, pipeline instruction fetch/execute units, ample number of general purpose registers (or alternatively optimized compiler code generation), Load/Store memory operations, and hardwired control unit design. While Complex Instruction Set Computers (CISCs) is became apparent that a complex instruction set has a number of disadvantages. These include a complex instruction decoding scheme, an increased size of the control unit, and increased logic delays.

RISCs 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. The 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.
4. A prime candidate for optimization is the mechanism for storing and accessing local scalar variables.

The following set of common characteristics among RISC machines is observed:

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

RISCs VERSUS CISCs

Tables 20 show a limited comparison between an example RISC and CISC machine in terms of characteristics:

TABLE 20 RISC Versus CISC Characteristics

Characteristic	(CISC)	(RISC)
Number of instructions	303	31
Instruction size (bits)	16-456	32
Addressing modes	22	3
No. general purpose registers	16	138

MULTIPROCESSORS

A multiple processor system consists of two or more processors that are connected in a manner that allows them to share the simultaneous (parallel) execution of a given computational task. Parallel processing has been advocated as a promising approach for building high-performance computer systems. The organization and performance of a multiple processor system are greatly influenced by the interconnection network used to connect them. On the one hand, a single shared bus can be used as the interconnection network for multiple processors.

CLASSIFICATION OF COMPUTER ARCHITECTURES

A number of classification schemes have been proposed, these include:

1- the Flynn's classification (1966).
2- the Kuck (1978).
3- the Hwang and Briggs (1984).

4- the Erlangen (1981).
5- the Giloi (1983).
6- the Skillicorn (1988). 7- the Bell (1992).

The instruction stream is defined as the sequence of instructions performed by the computer. The data stream is defined as the data traffic exchanged between the memory and the processing unit. This leads to four distinct categories of computer architectures:

1. Single-instruction single-data streams (SISD)
2. Single-instruction multiple-data streams (SIMD)
3. Multiple-instruction single-data streams (MISD)
4. Multiple-instruction multiple-data streams (MIMD)

SIMD SCHEMES

Two main SIMD configurations have been used in real-life machines. These are shown in Figure 56.

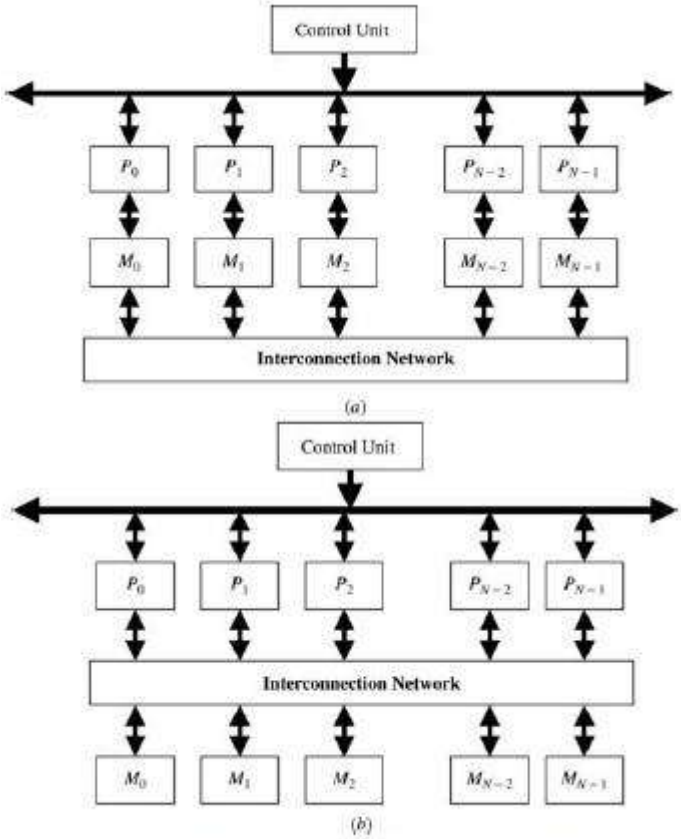
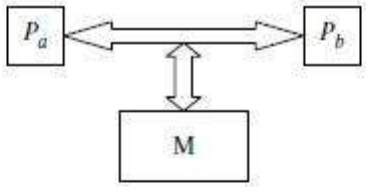


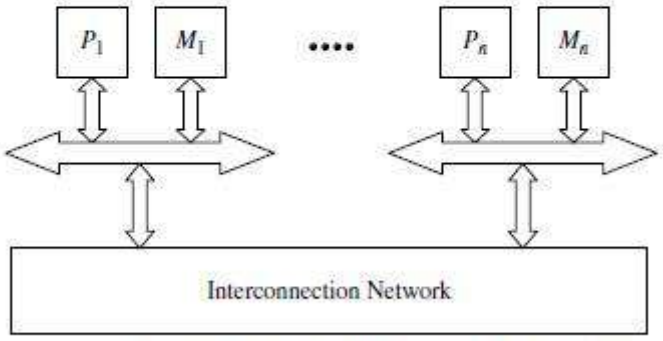
Figure 56 Two SIMD schemes. (*a*) SIMD scheme 1, (*b*) SIMD scheme 2

MIMD SCHEMES

MIMD machines use a collection of processors, each having its own memory, which can be used to collaborate on executing a given task. In general, MIMD systems can be categorized based on their memory organization into shared-memory and message-passing architectures.



a- A simple shared memory scheme



b- a message-passing multiprocessor architecture

Figure 57 MIMD Schemes

INTERCONNECTION NETWORKS

The classification of interconnection networks is based on topology. Interconnection networks are classified as either static or dynamic. In Figure 58, is provide such a taxonomy.

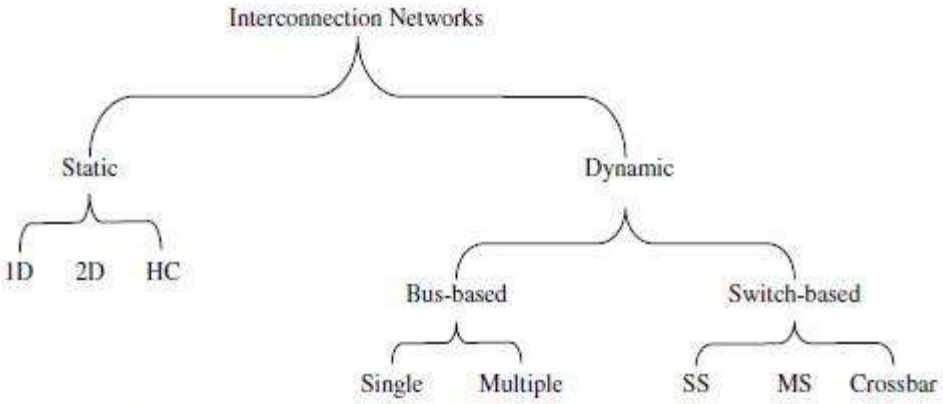


Figure 58 A topology-based taxonomy for interconnection networks

Computer Architecture Part3-2

2023-2024

Reduced Instruction Set Computer (RISC)

An important aspect of computer architecture is the design of the instruction set for the processor. The instruction set chosen for a particular computer determines the way that machine language programs are constructed. A computer with a large number of instructions is classified as a complex instruction set computer, abbreviated CISC. In the early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. This RISC type of computer is classified as a reduced instruction set computer or RISC.

In summary, the major characteristics of CISC architecture are:

1. A large number of instructions—typically from 100 to 250 instructions.
2. Some instructions that perform specialized tasks and are used infrequently.
3. A large variety of addressing modes—typically from 5 to 20 different modes.
4. Variable-length instruction formats.
5. Instructions that manipulate operands in memory.

The major characteristics of a RISC processor are:

1. Relatively few instructions.
2. Relatively few addressing modes.
3. Memory access limited to load and store instructions.
4. All operations done within the registers of the CPU.
5. Fixed-length, easily decoded instruction format.
6. Single-cycle instruction execution.
7. Hardwired rather than microprogrammed control.

Memory Hierarchy

The memory unit is an essential component in any digital computer since it is needed for storing programs and data. The memory unit that communicates directly with the CPU is called the main memory. Devices that provide backup storage are called auxiliary memory. They are used for storing system programs, large data files, and other backup information. Only programs and data currently needed by the processor reside in main memory. All other information is stored in auxiliary memory and transferred to main memory when needed. A special very-high-speed memory called a cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. Fig(29) shows the Memory Hierarchy:

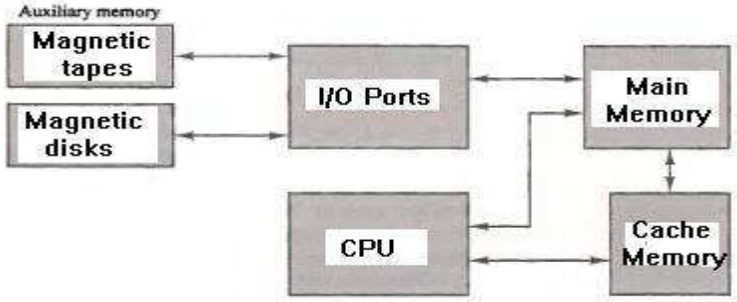


Figure 29 Memory hierarchy in a computer system.

Main Memory The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible operating modes:

The static RAM consists essentially of internal flip-flops that store the binary information. *The dynamic RAM* stores the binary information in the form of electric charges that are applied to capacitors.

Associative Memory

Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent.

A memory unit accessed by content is called an associative memory or content addressable memory (CAM). When a word is written in an associative memory is capable of finding an empty unused location to store the word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locates all words which match the specified content and marks them for reading. The block diagram of an associative memory is shown in Fig(30):

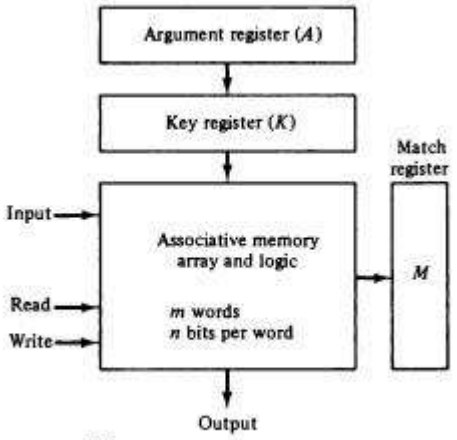


Figure 30 Block diagram of associative memory.

To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three left most bits of A are compared with memory words because K has l's in these positions.

A	101 111100	
K	111 000000	
Word 1	100 111100	no match
Word 2	101 000001	match

Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

Cache Memory

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a cache memory. It is placed between the CPU and main memory.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. The performance of cache memory is frequently measured in terms of a quantity called *hit ratio*. When the CPU refers to memory and finds the word in cache, it is said to produce a *hit*. If the word is not found in cache, it is in main memory and it counts as a *miss*.

Three types of mapping procedures are of practical interest when considering the organization of cache memory:

1. Associative mapping
2. Direct mapping
3. Set-associative mapping

Virtual Memory

Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating programgenerated addresses into correct main memory locations.

As an illustration, consider a computer with a main-memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since $32K = 2^{15}$. Suppose that the computer has available auxiliary memory for storing 2^{20} = 1024K words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories.

Denoting the address space by N and the memory space by M, we then have for this example N = 1024K and M = 32K.

The mapping table may be stored in a separate memory as shown in Fig(31) or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In the second case, the table takes space from main memory and two accesses to memory are required with the program running at half speed.

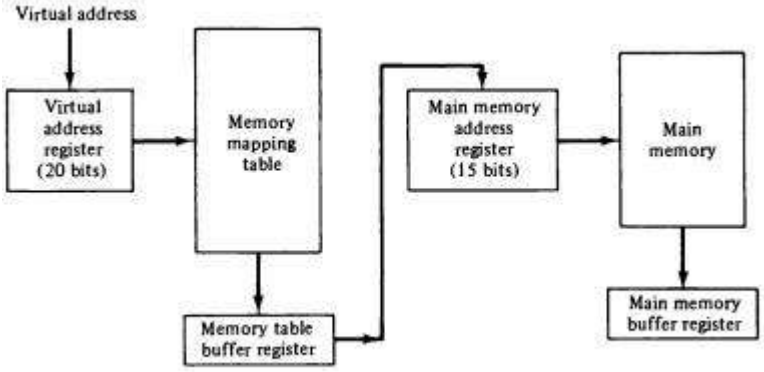


Figure 31 Memory table for mapping a virtual address.

The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size pages and blocks called blocks, which may range from 64 to 4096 words each. The term page refers to groups of address space of the same size. For example, if a page or block consists of IK words, then, using the previous example, address space is divided into 1024 pages and main memory is divided into 32 blocks.

The organization of the memory mapping table in a paged system is shown in Fig(32). The memory-page table consists of eight words, one for each page. The address in the page table denotes the page number and the content of the word gives the block number where that page is stored in main memory. The table shows that pages 1, 2, 5, and 6 are now available in main memory in blocks 0, 1, 2, and 3, respectively. A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory. A_0 in the presence bit indicates that this page is not available in main memory.

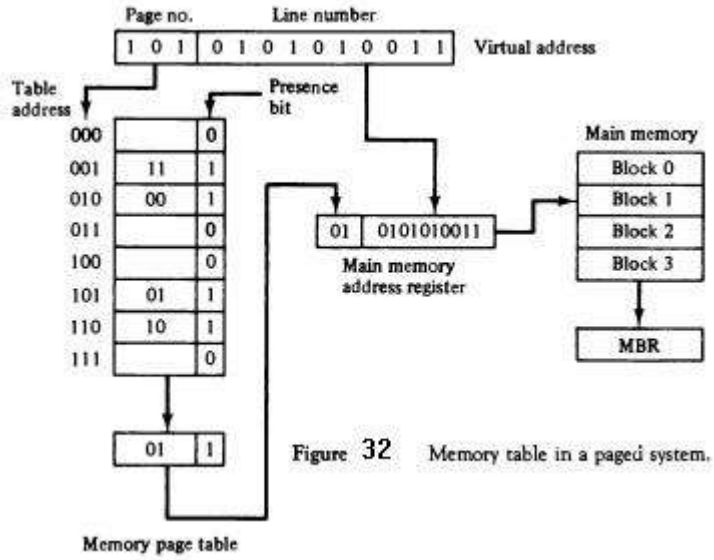


Figure 32 Memory table in a paged system.

Memory Management Hardware

A memory management system is a collection of hardware and software procedures for managing the various programs residing in memory. The memory management software is part of an overall operating system available in many computers.

The basic components of a memory management unit are:

1. A facility for dynamic storage relocation that maps logical memory references into physical memory addresses.
2. A provision for sharing common programs stored in memory by different users.
3. Protection of information against unauthorized access between users and preventing users from changing operating system functions.

The fixed page size used in the virtual memory system causes certain difficulties with respect to program size and the logical structure of programs. It is more convenient to divide programs and segment data into logical parts called segments.

A segment is a set of logically related instructions or data elements associated with a given name. Segments may be generated by the programmer or by the operating system. Examples of segments are a subroutine, an array of data, a table of symbols, or a user's program. The address generated by a segmented program is called a logical address. The logical address may be larger than the physical memory address as in virtual memory, but it may also be equal, and sometimes even smaller than the length of the physical memory address.

Numerical Example: A numerical example may clarify the operation of the memory management unit. Consider the 20-bit logical address specified in Fig(33-a).This configuration allows each segment to have any number of pages up to 256. The smallest possible segment will have one page or 256 words. The largest possible segment will have 256 pages, for a total of 256 x 256 = 64K words. The physical memory shown in Fig(33-b).

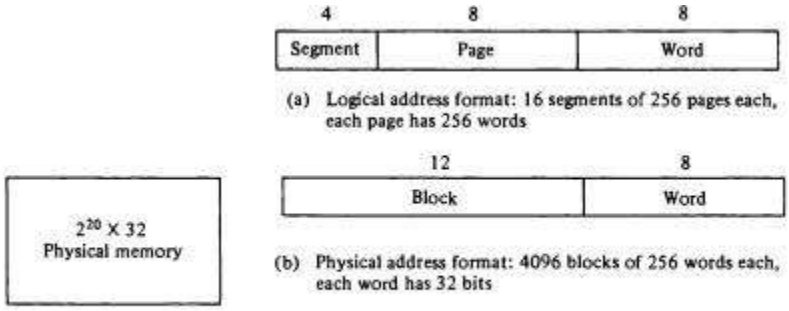


Figure 33 An example of logical and physical addresses.

Consider a program loaded into memory that requires five pages. The operating system may assign to this program segment 6 and pages 0 through 4, as shown in Fig(34-a). The total logical address range for the program is from hexadecimal 60000 to 604FF. The correspondence between each memory block and logical page number is then entered in a table as shown in Fig(34-b).

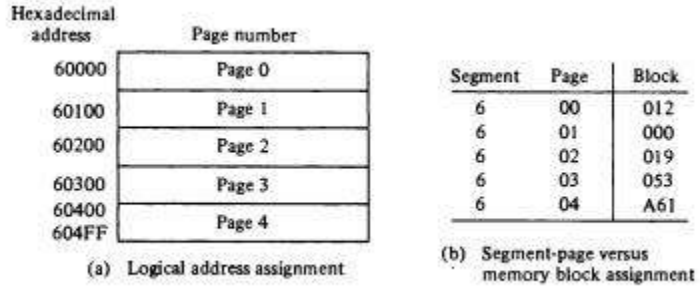
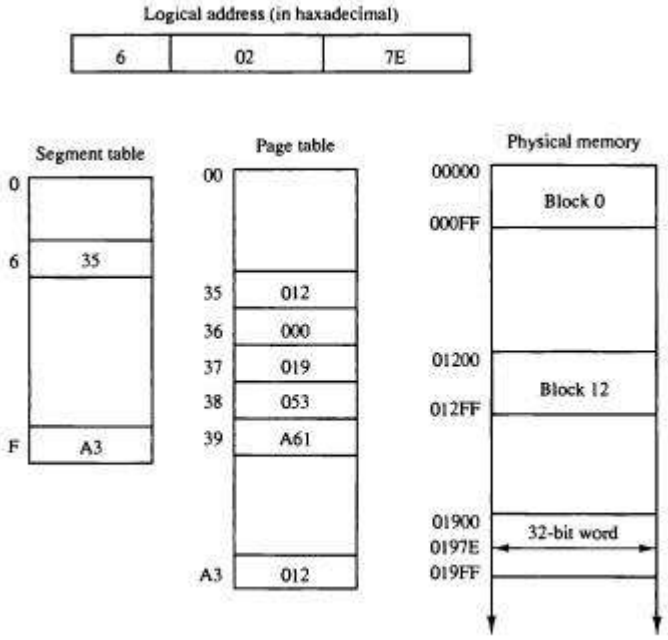


Figure 34 Example of logical and physical memory address assignment.

The information from this table is entered in the segment and page tables as shown in Fig(35-a). Now consider the specific logical address given in Fig(35). The 20-bit address is listed as a five-digit hexadecimal number. It refers to word number 7E of page 2 in segment 6. The base of segment 6 in the page table is at address 35. Segment 6 has associated with it five pages, as shown in the page table at addresses 35 through 39. Page 2 of segment 6 is at address 35 + 2 = 37. The physical memory block is found in the page table to be 019. Word 7E in block 19 gives the 20-bit physical address 0197E. Note that page 0 of segment 6 maps into block 12 and page 1 maps into block 0. The associative memory in Fig(35-b) shows that pages 2 and 4 of segment 6 have been referenced previously and therefore their corresponding block numbers are stored in the associative memory.



(a) Segment and page table mapping

Continue

Segment	Page	Block
6	02	019
6	04	A61

(b) Associative memory (TLB)

Figure 35 Logical to physical memory mapping example (all numbers are in hexadecimal).

Input-Output Organization

The input-output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the outside environment. Programs and data must be entered into computer memory for processing and results obtained from computations must be recorded or displayed for the user.

Peripheral Devices

Input or output devices attached to the computer are also called peripherals.

- The display terminal can operate in a single-character mode where all characters entered on the screen through the keyboard are transmitted to the computer simultaneously. In the block mode, the edited text is first stored in a local memory inside the terminal. The text is transferred to the computer as a block of data.
- Printers provide a permanent record on paper of computer output data.
- Magnetic tapes are used mostly for storing files of data.
- Magnetic disks have high-speed rotational surfaces coated with magnetic material.

Input-Output Interface

Input-output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The major differences are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.
2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.
3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

A typical communication link between the processor and several peripherals is shown in Fig.36. The I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The interface selected responds to the function code and proceeds to execute it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit.

There are three ways that computer buses can be used to communicate with memory and I/O:

1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and I/O but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

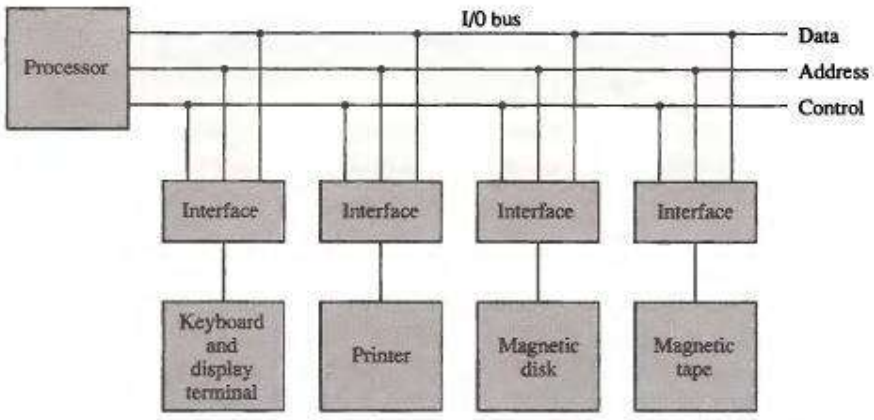
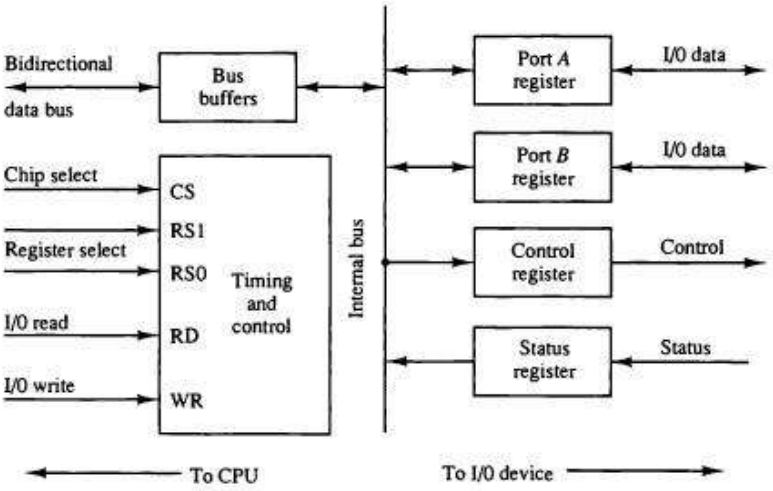


Figure 36 Connection of I/O bus to input-output devices.

Isolated I/O versus Memory-Mapped I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. The isolated I/O method isolates memory and I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same address space for both memory and I/O.



CS	RS1	RS0	Register selected
0	×	×	None: data bus in high-impedance
1	0	0	Port *A* register
1	0	1	Port *B* register
1	1	0	Control register
1	1	1	Status register

Figure 37 Example of I/O interface unit.

This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as memory mapped I/O. In a memory-mapped I/O organization there is no specific input or output instructions. Computers with memory-mapped I/O can use memory-type instructions to access I/O data.

An example of an I/O interface unit is shown in block diagram form in Fig.37. It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.

Asynchronous Data Transfer

The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems. Asynchronous data transfer between two

independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted. Two way of achieving this:

- The strobe: pulse supplied by one of the units to indicate to the other unit when the transfer has to occur.
- The handshaking: The unit receiving the data item responds with another control signal to acknowledge receipt of the data.

The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers.

The strobe may be activated by either the source or the destination unit. Figure 38 shows a source-initiated transfer and the timing diagram.

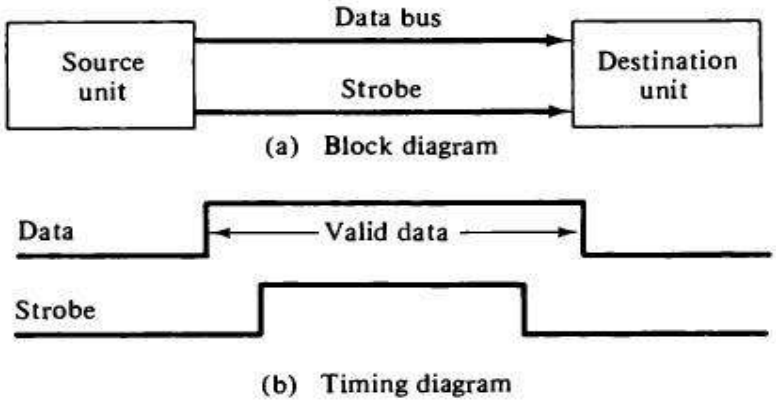


Figure 38 Source-initiated strobe for data transfer.

Fig.39 shows the strobe of a memory-read control signal from the CPU to a memory.

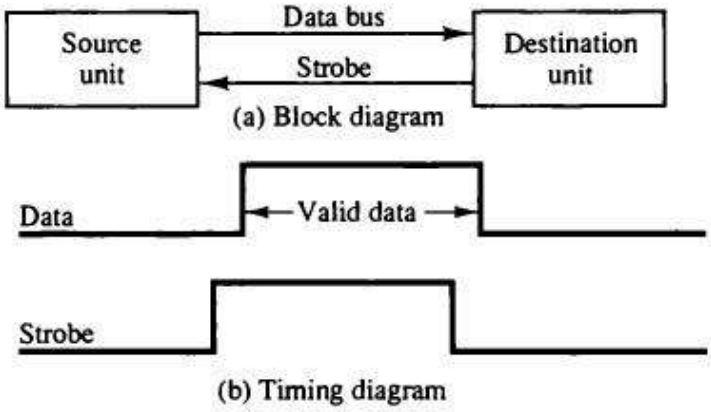


Figure 39 Destination-initiated strobe for data transfer.

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that two-wire control initiates the transfer.

Figure 40 shows the data transfer procedure when initiated by the source. The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the destination unit. The timing diagram shows the exchange of signals between the two units. Figure 41 the destination-initiated transfer using handshaking lines. Note that the name of the signal generated by the destination unit has been changed to ready for data to reflect its new meaning.

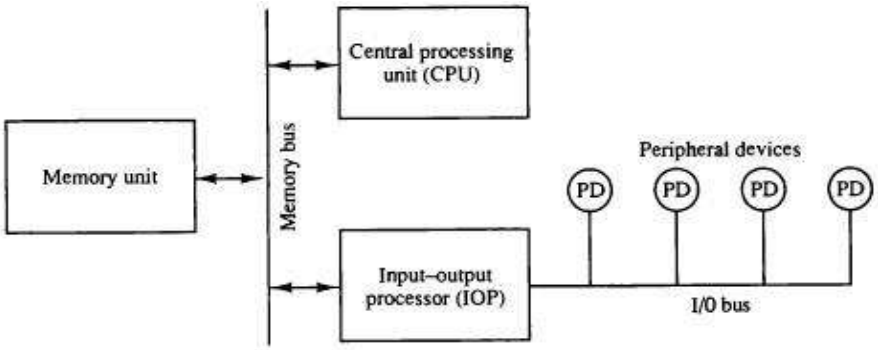


Figure 45 Block diagram of a computer with I/O processor.

Pipelining

Pipelining is a technique of decomposing a sequential process into sub-operations; with each sub-process being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows.

General Considerations

Any operation that can be decomposed into a sequence of sub-operations of about the same complexity can be implemented by a pipeline processor. The general structure of a four segment pipeline is illustrated in Fig. 46. The operands pass through all four segments in a fixed sequence.

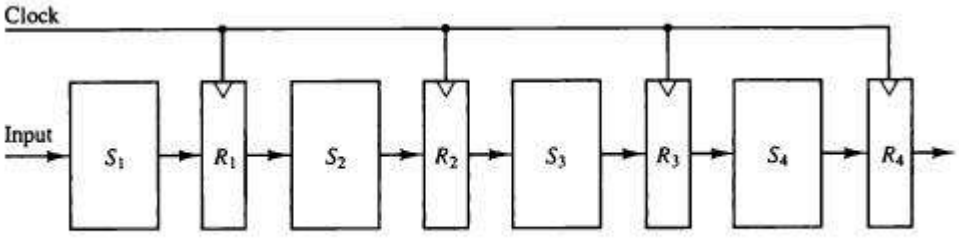


Figure 46 Four-segment pipeline.

The space-time diagram of a four-segment pipeline is demonstrated in Fig47.

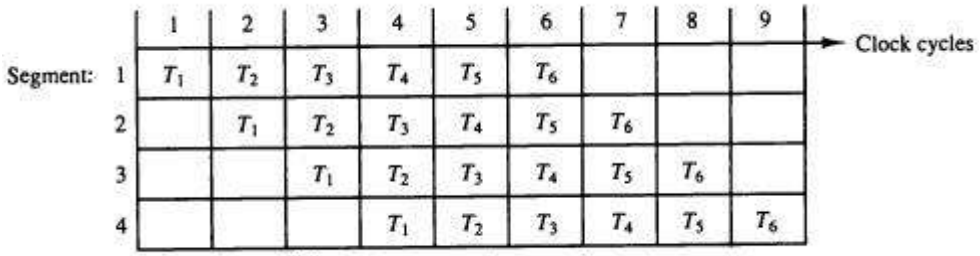


Figure 47 Space-time diagram for pipeline.

The speedup(S) of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio: $$S = \frac{nt_n}{(k+n-1)t_p}$$

As the number of tasks increases, n becomes much larger than $k - 1$, and $k + n - 1$ approaches the value of n. Under this condition, the speedup becomes:

$$S = \frac{t_n}{t_p}$$

numerical example: Let the time it takes to process a sub-operation in each segment be equal to t_p= 20 ns. Assume that the pipeline has $k = 4$ segments and executes $n = 100$ tasks in sequence. The pipeline system will take

$$(k + n - 1)t_p = (4 + 99) \times 20 = 2060ns$$

to complete. Assuming that $t = k \times t_p = 4 \times 20 = 80$ ns, a non-pipeline system requires:

$$nkt_p = 100 \times 80 = 8000ns$$

to complete the 100 tasks. The speedup ratio is equal to:

$$8000/2060 = 3.88$$

Instruction Pipeline

The computer needs to process each instruction with the following sequence of steps:

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

Figure 48 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3.

The four segments are represented in the flowchart:

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

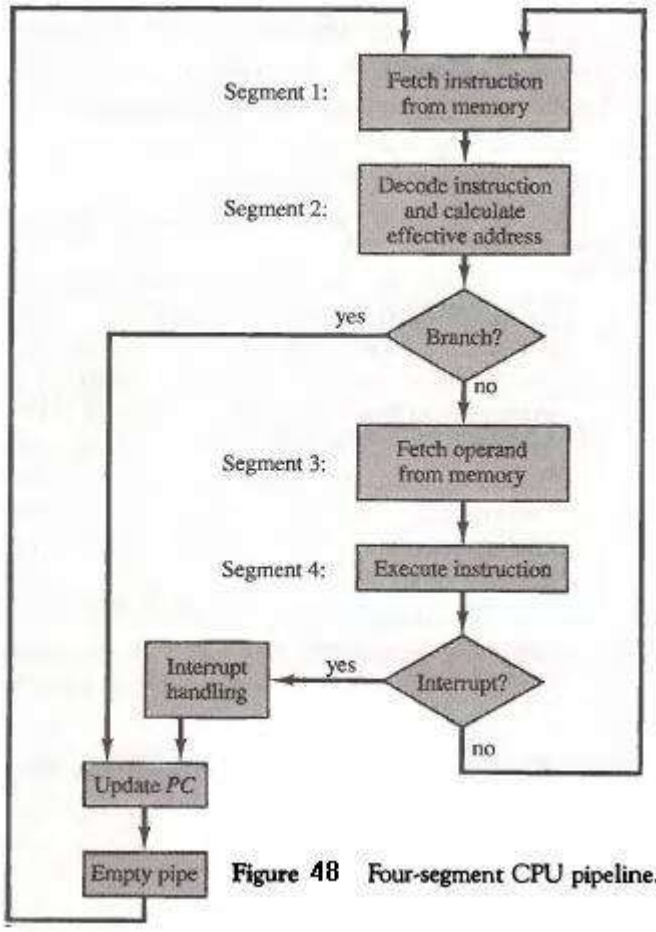


Figure 48 Four-segment CPU pipeline.

A pipeline operation is said to have been stalled if one unit (stage) requires more time to perform its function, thus forcing other stages to become idle. Consider, for example, the case of an instruction fetch that incurs a cache miss. Assume also that a cache miss requires three extra time units.

Instruction-Level Parallelism

Contrary to pipeline techniques, instruction-level parallelism (ILP) is based on the idea of multiple issue processors (MIP). An MIP has multiple pipelined data paths for instruction

execution. Each of these pipelines can issue and execute one instruction per cycle. Figure 49 shows the case of a processor having three pipes. For comparison purposes, we also show in the same figure the sequential and the single pipeline case. Instruction-level parallelism (ILP) is the parallel or simultaneous execution of a sequence of instructions in a computer program. More specifically ILP refers to the average number of instructions run per step of this parallel execution.

(ILP): هو التنفيذ المتوازي أو المتزامن لسلسلة من التعليمات في برنامج كمبيوتر. بشكل أكثر تحديدًا ، يشير ILP إلى متوسط عدد التعليمات التي يتم تشغيلها لكل خطوة من هذا التنفيذ المتوازي.

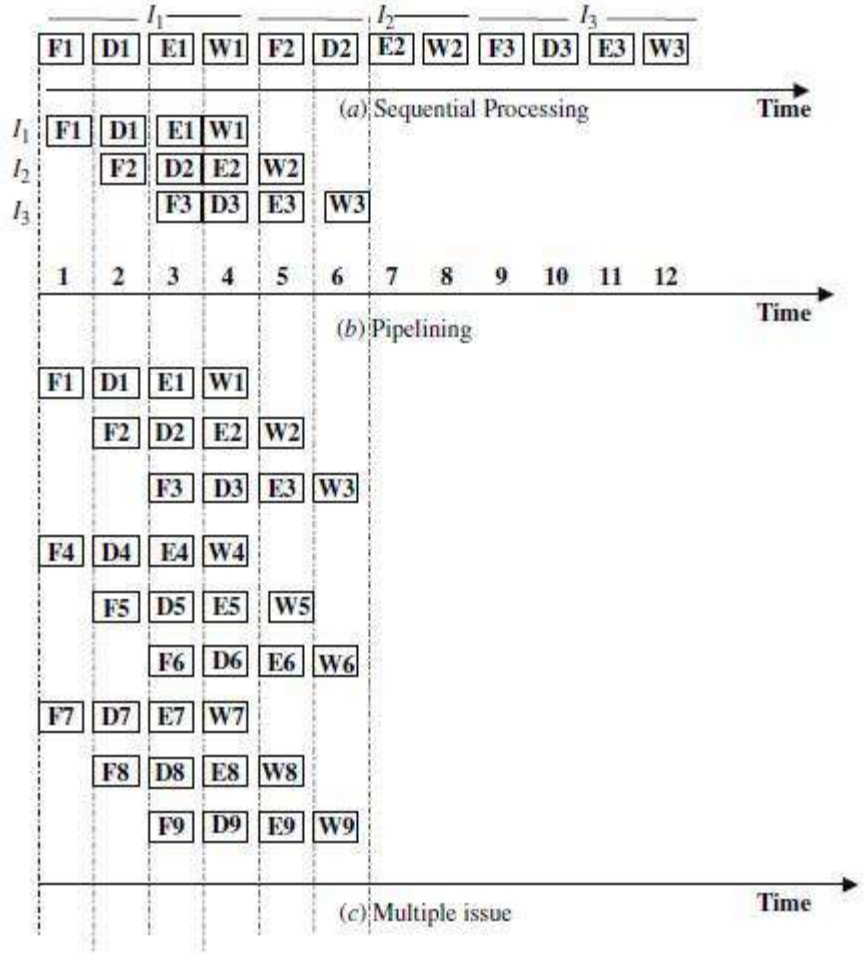


Figure 49 Multiple issue versus pipelining versus sequential processing

Computer Arithmetic

Arithmetic instructions in digital computers manipulate data to produce results necessary for the solution of computational problems. An arithmetic processor is the part of a processor unit that executes arithmetic operations. The data type assumed to reside in processor registers during the execution of an arithmetic instruction is specified in the definition of the instruction. The solution to any problem that is stated by a finite number of well-defined procedural steps is called an algorithm.

Addition and Subtraction with Signed-Magnitude Data: We designate (عين) the magnitude of the two numbers by A and B. When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 18. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

TABLE 18 Addition and Subtraction of Signed-Magnitude Numbers

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

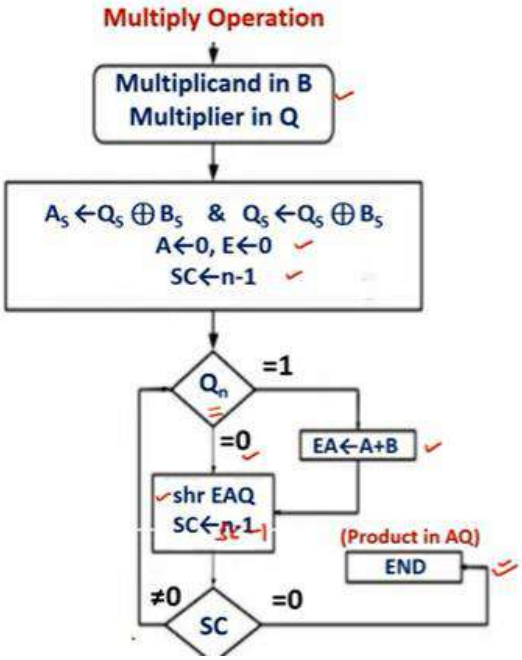
Multiplication Algorithms

Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive shift and add operations. This process is best illustrated with a numerical example:

Note: The product of multiplying any binary number x by a single binary digit is always either 0 or x. Therefore, the multiplication of two binary numbers comes down to shifting the multiplicand left appropriately for each non-zero bit in the multiplier, and then adding the shifted numbers together

```
23      10111    Multiplicand
19    × 10011    Multiplier
        10111
       10111
      00000   +
     00000
    10111
437 110110101    Product
```

Figure 52 is a flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s, respectively. ***The signs are compared, and both A and Q are set to correspond to the sign of the product*** since a double length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier.



Hardware Algorithm for Multiply Operation

- Initially multiplicand is stored in B register and multiplier is stored in Q register
- Sign of registers B (Bs) and Q (Qs) are compared using **XOR** functionality (if both the signs are alike, output of XOR operation is 0 unless 1)
- Output is stored in As (sign of A register)
- Initially 0 is assigned to register A and E flip flop
- Sequence counter (SC) is initialized with value n-1, n is the number of bits in the Multiplier
- Now least significant bit of multiplier is checked. If it is 1 add the content of register A with Multiplicand (register B) and result is assigned in A register with carry bit in flip flop E
- Content of E A Q is shifted to right by one position
- If Qn = 0, only shift right operation on content of E A Q is performed in a similar fashion
- Content of Sequence counter is decremented by 1
- Check the content of SC, if it is 0, end the process and the final product is present in register A and Q, else repeat the process

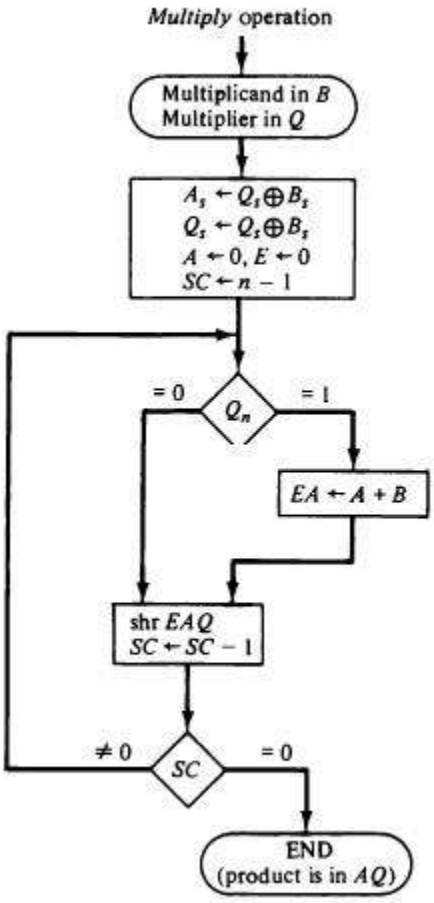
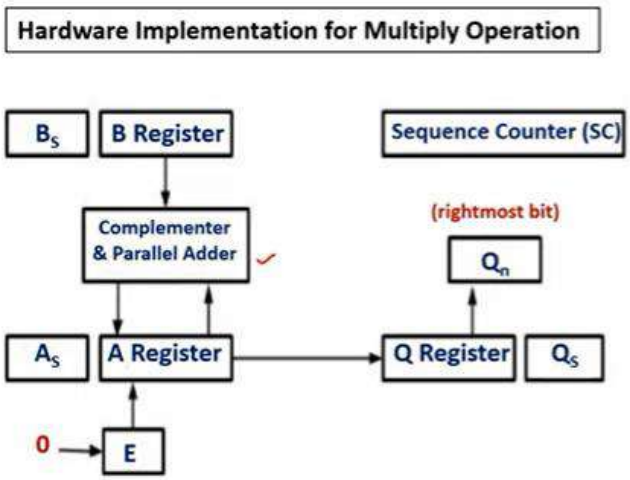
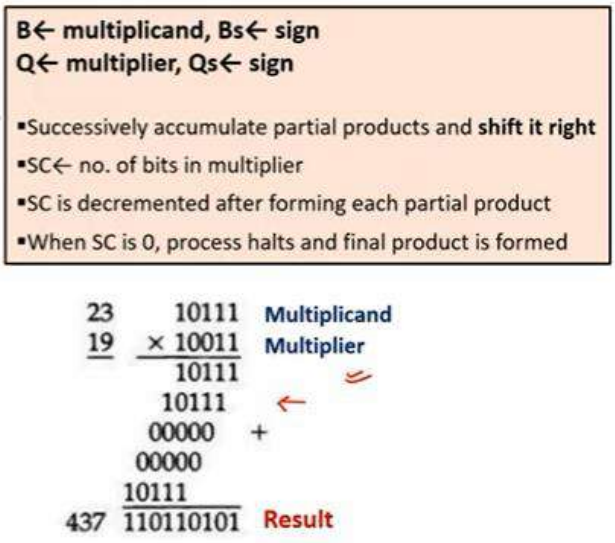


Figure 52 Flowchart for multiply operation.

https://www.youtube.com/watch?v=2IxktI0BOu4





58

The hardware for implementing the division operation is identical to that required for multiplication and consists of the components Register EAQ is now shifted to the left with 0 inserted into Q, and the previous value of E lost. The numerical example is repeated as in Figure 53:

Multiplication Process

The multiplier and multiplicand are loaded into two registers Q and M.

A Third Register A is initially set to zero.

C is the 1-bit register holds the carry bit resulting from addition. Now

The control logic reads bits of the multiplier at one time.

If Q_0 is 1, the multiplicand is added to register A and stored back in register A with C bit used for carry.

Then all the bits of CAQ are shifted to right 1 bit, so that C bit goes to A_{n-1} A_0 goes to Q_{n-1} and Q_n is lost.

If Q_0 is 0 no addition is performed just do the shift.

The process is repeated for each bit of the original multiplier.

The result 2n bit product is contained in QA registers

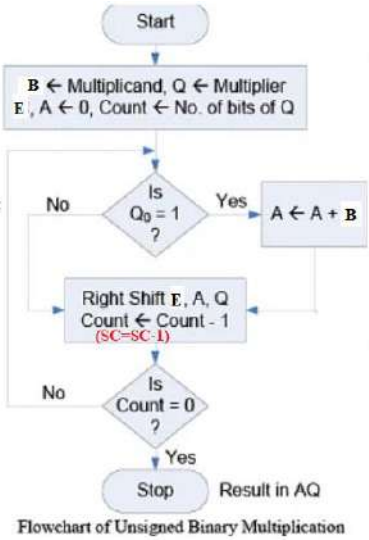
Divisor $B = 10001$, $\overline{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Figure 53 Example of binary division with digital hardware.

Unsigned Binary Multiplication

```
   1011   Multiplicand 11
 x 1101   Multiplier 13
   1011
   0000   Partial Product
  1011
+ 1011
 10001111 Product (143)
```



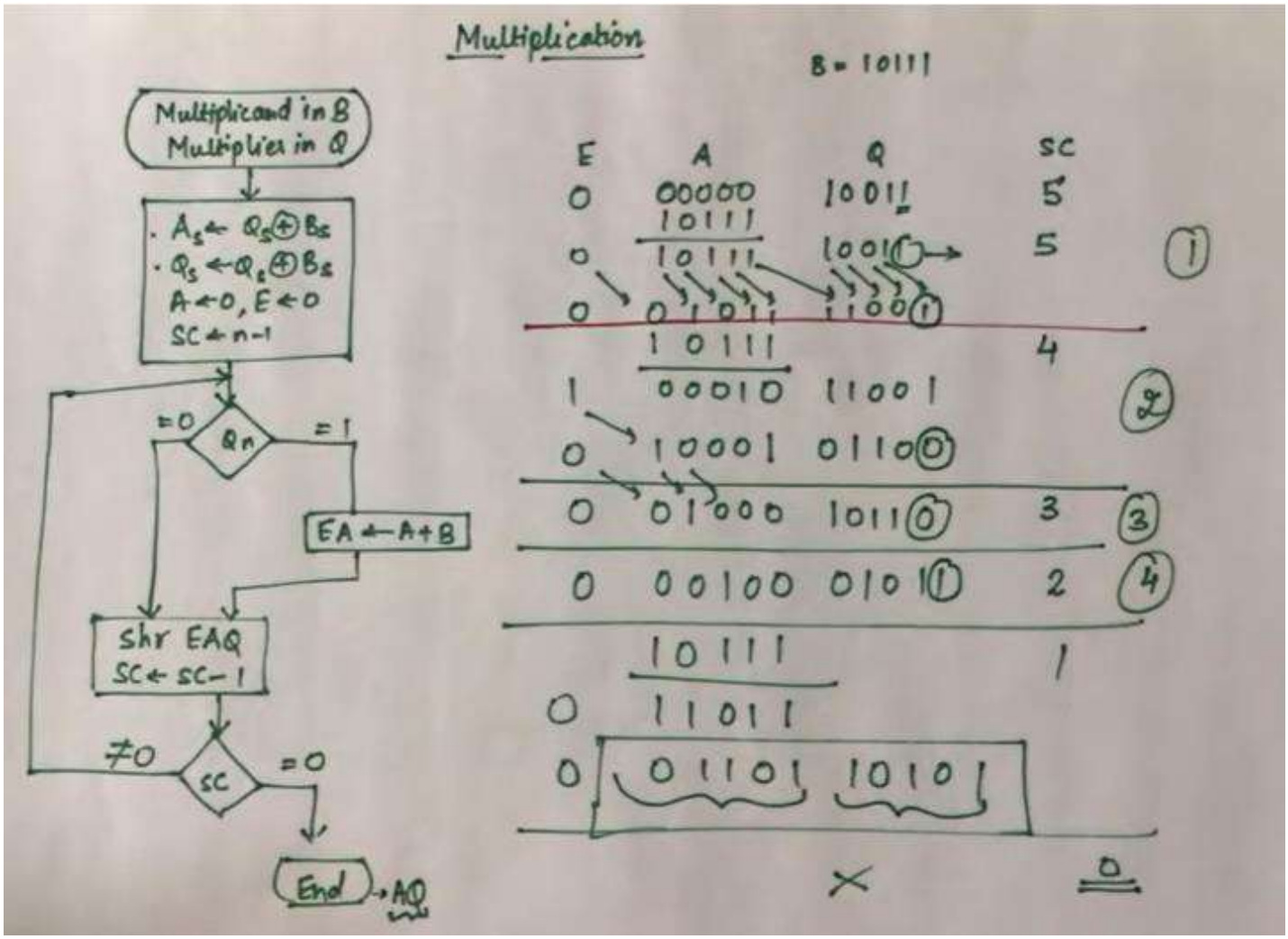
Flowchart of Unsigned Binary Multiplication

Example: Multiply 15 x 11 using unsigned binary method 1011 * 1111

E	A	Q	B	SC	Remarks
0	0000	1011	1111	4	Initialization
0	1111	1011	-	-	Add (A ← A + B)
0	0111	1101	-	3	Logical Right Shift E, A, Q
1	0110	1101	-	-	Add (A ← A + B)
0	1011	0110	-	2	Logical Right Shift E, A, Q
0	0101	1011	-	1	Logical Right Shift E, A, Q
1	0100	1011	-	-	Add (A ← A + B)
0	1010	0101	-	0	Logical Right Shift E , A, Q

Result = 1010 0101 = $2^7 + 2^5 + 2^2 + 2^0 = 165$

Method of Unsigned Binary Multiplication



Decimal Arithmetic Unit

To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal. It can add or subtract decimal numbers, usually by forming the 9's or 10's complement of the subtrahend(المطروح). Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in Fig. 54. The two decimal digits, together with the input- carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output carry is equal to 0, nothing is added to the binary sum.

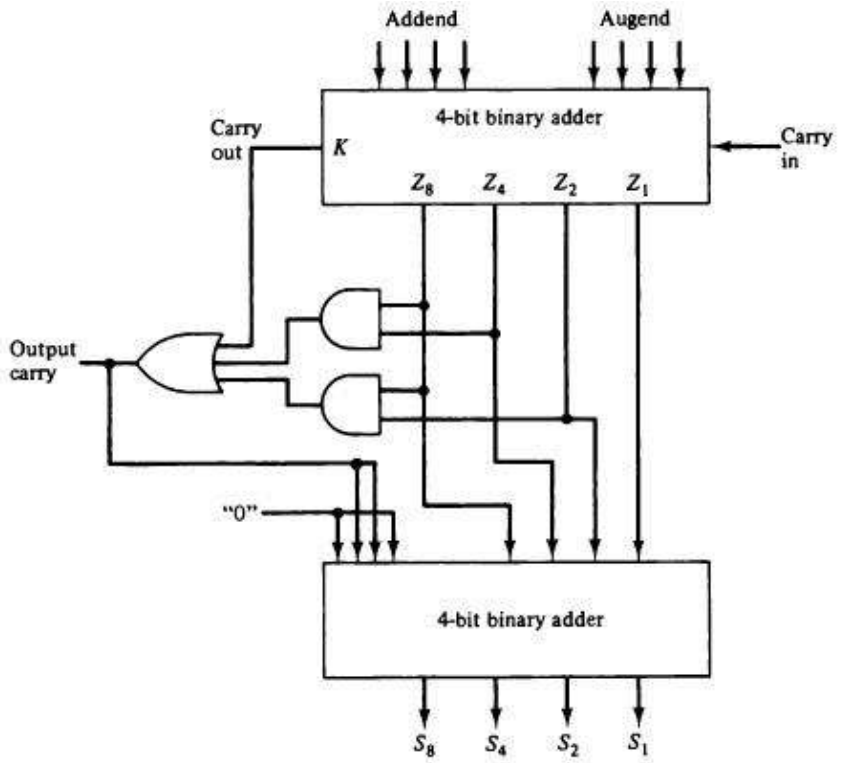


Figure 54 Block diagram of BCD adder.

Reduced Instruction Set Computers (RISCs)

The RISC approach is RISC-based machines are reality and they are characterized by a number of common features such as simple and reduced instruction set, fixed instruction format, one instruction per machine cycle, pipeline instruction fetch/execute units, ample number of general purpose registers (or alternatively optimized compiler code generation), Load/Store memory operations, and hardwired control unit design. While Complex Instruction Set Computers (CISCs) is became apparent that a complex instruction set has a number of disadvantages. These include a complex instruction decoding scheme, an increased size of the control unit, and increased logic delays.

RISCs 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. The 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.
4. A prime candidate for optimization is the mechanism for storing and accessing local scalar variables.

The following set of common characteristics among RISC machines is observed:

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

RISCs VERSUS CISCs

Tables 20 show a limited comparison between an example RISC and CISC machine in terms of characteristics:

TABLE 20 RISC Versus CISC Characteristics

Characteristic	(CISC)	(RISC)
Number of instructions	303	31
Instruction size (bits)	16-456	32
Addressing modes	22	3
No. general purpose registers	16	138

MULTIPROCESSORS

A multiple processor system consists of two or more processors that are connected in a manner that allows them to share the simultaneous (parallel) execution of a given computational task. Parallel processing has been advocated as a promising approach for building high-performance computer systems. The organization and performance of a multiple processor system are greatly influenced by the interconnection network used to connect them. On the one hand, a single shared bus can be used as the interconnection network for multiple processors.

CLASSIFICATION OF COMPUTER ARCHITECTURES

A number of classification schemes have been proposed, these include:

1- the Flynn's classification (1966).
2- the Kuck (1978).
3- the Hwang and Briggs (1984).
4- the Erlangen (1981).
5- the Giloi (1983).
6- the Skillicorn (1988). 7- the Bell (1992).

The instruction stream is defined as the sequence of instructions performed by the computer. The data stream is defined as the data traffic exchanged between the memory and the processing unit. This leads to four distinct categories of computer architectures:

1. Single-instruction single-data streams (SISD)
2. Single-instruction multiple-data streams (SIMD)
3. Multiple-instruction single-data streams (MISD)
4. Multiple-instruction multiple-data streams (MIMD)

SIMD SCHEMES

Two main SIMD configurations have been used in real-life machines. These are shown in Figure 56.

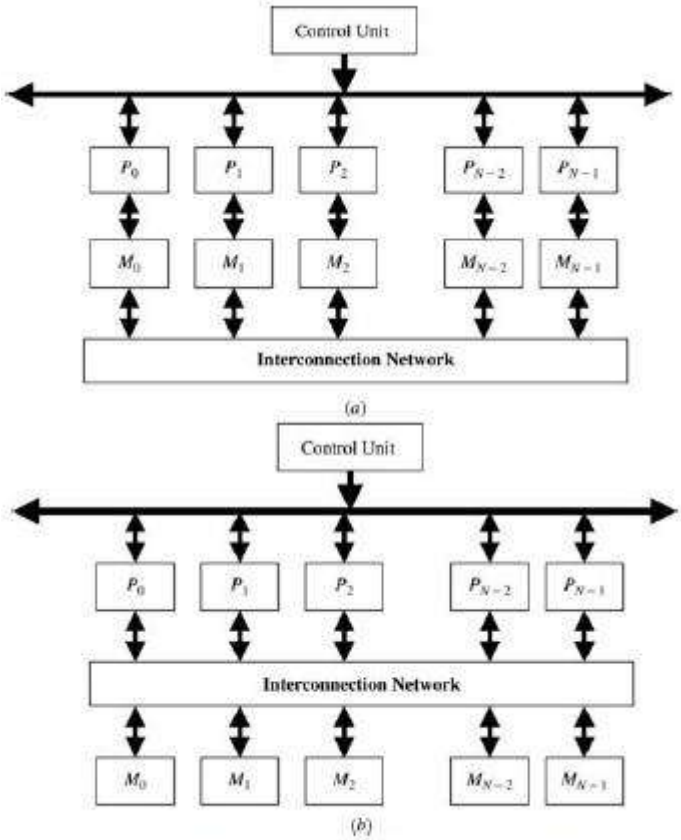
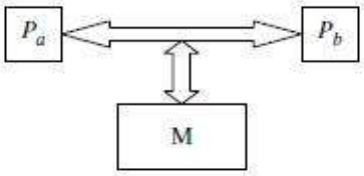


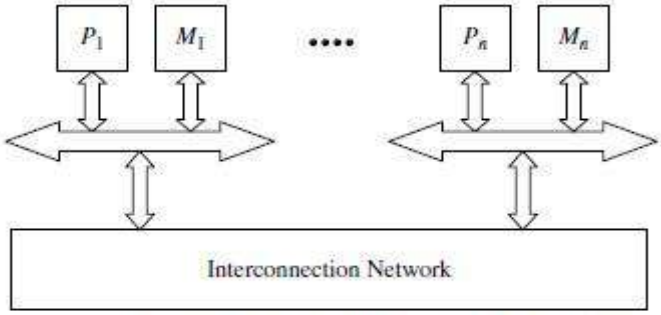
Figure 56 Two SIMD schemes. (*a*) SIMD scheme 1, (*b*) SIMD scheme 2

MIMD SCHEMES

MIMD machines use a collection of processors, each having its own memory, which can be used to collaborate on executing a given task. In general, MIMD systems can be categorized based on their memory organization into shared-memory and message-passing architectures.



a- A simple shared memory scheme



b- a message-passing multiprocessor architecture

Figure 57 MIMD Schemes

INTERCONNECTION NETWORKS

The classification of interconnection networks is based on topology. Interconnection networks are classified as either static or dynamic. In Figure 58, is provide such a taxonomy.

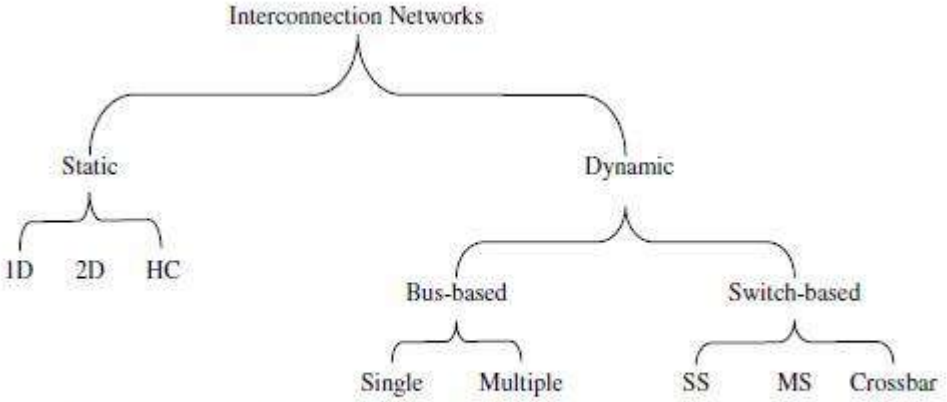


Figure 58 A topology-based taxonomy for interconnection networks

Parallel Processing

Execution of Concurrent Events in the computing process to achieve faster Computational Speed

The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time.

The amount of hardware increases with parallel processing, and with it, the cost of the system increases.

However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

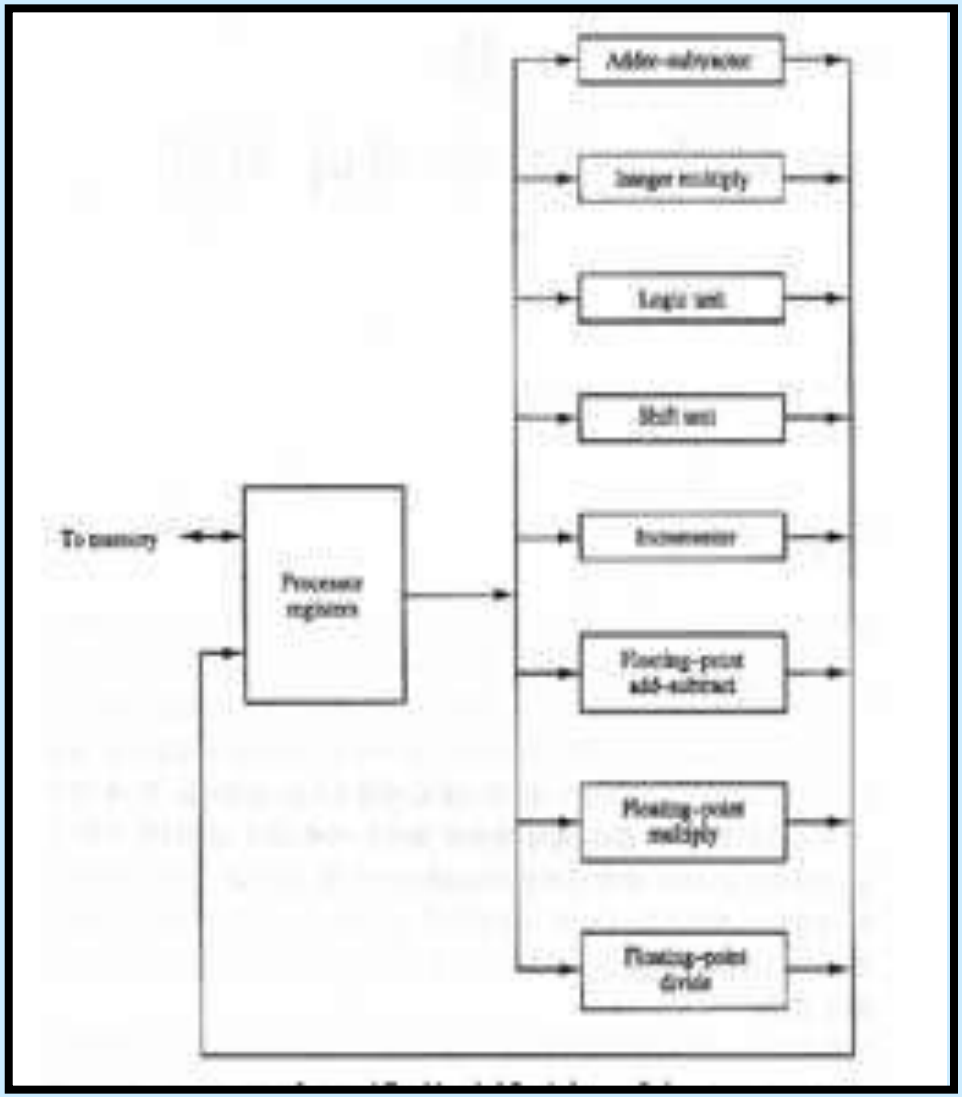
Parallel processing according to levels of complexity

At the lower level

Serial Shift register VS parallel load registers

At the higher level

Multiplicity of functional units that performs identical or different operations simultaneously.



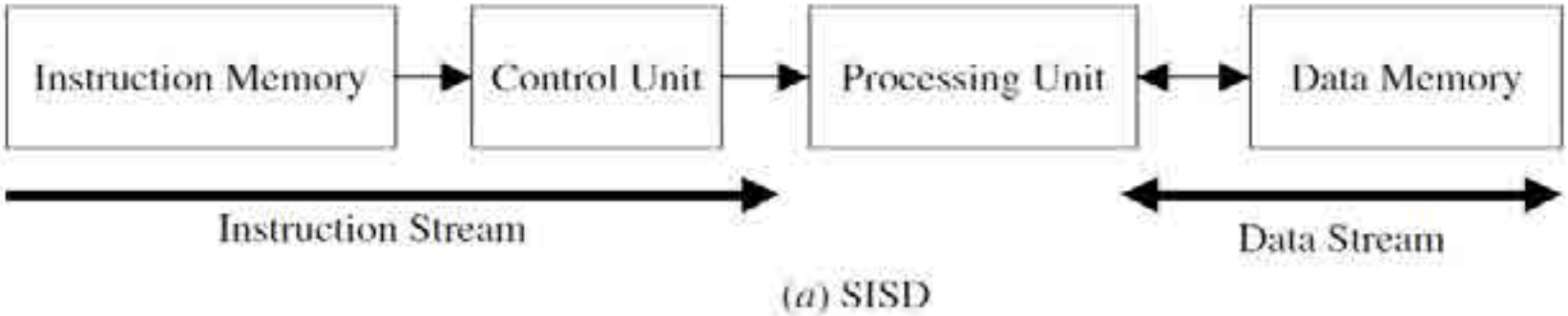
Parallel Computers

Architectural Classification

- Flynn's classification
  - » Based on the multiplicity of *Instruction Streams* and *Data Streams*
  - » Instruction Stream
    - Sequence of Instructions read from memory
  - » Data Stream
    - Operations performed on the data in the processor

		Number of *Data Streams*	
		Single	Multiple
Number of *Instruction Streams*	Single	SISD	SIMD
	Multiple	MISD	MIMD

SISD COMPUTER SYSTEMS



(*a*) SISD

- Standard von Neumann machine
- Instructions and data are stored in memory
- One operation at a time

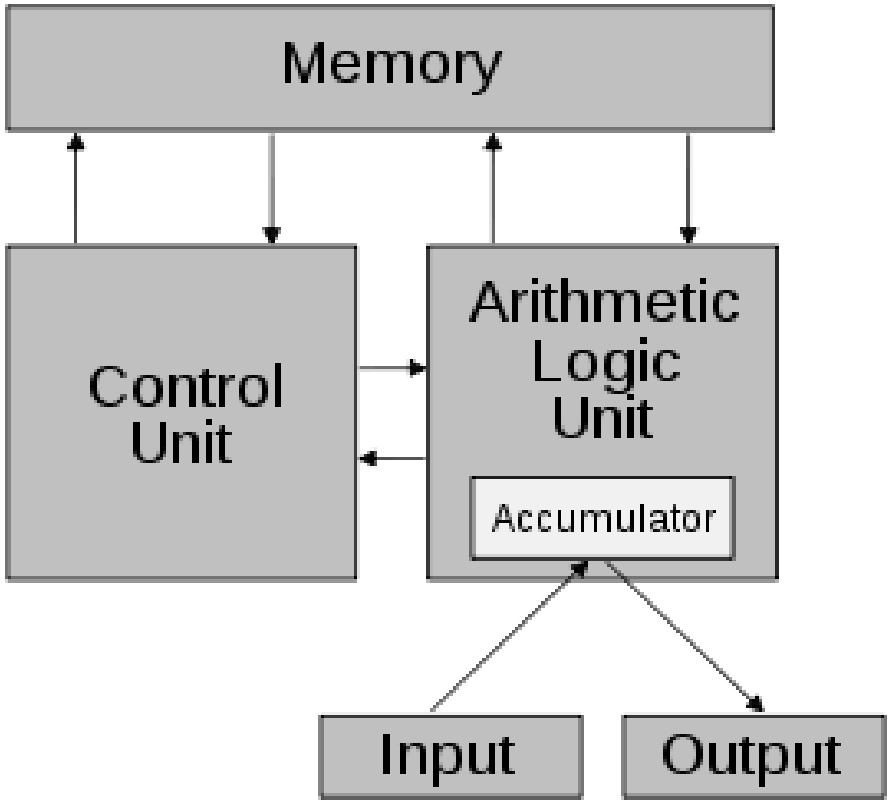
Limitations

Von Neumann bottleneck

Maximum speed of the system is limited by the ***Memory Bandwidth*** (bits/sec or bytes/sec)

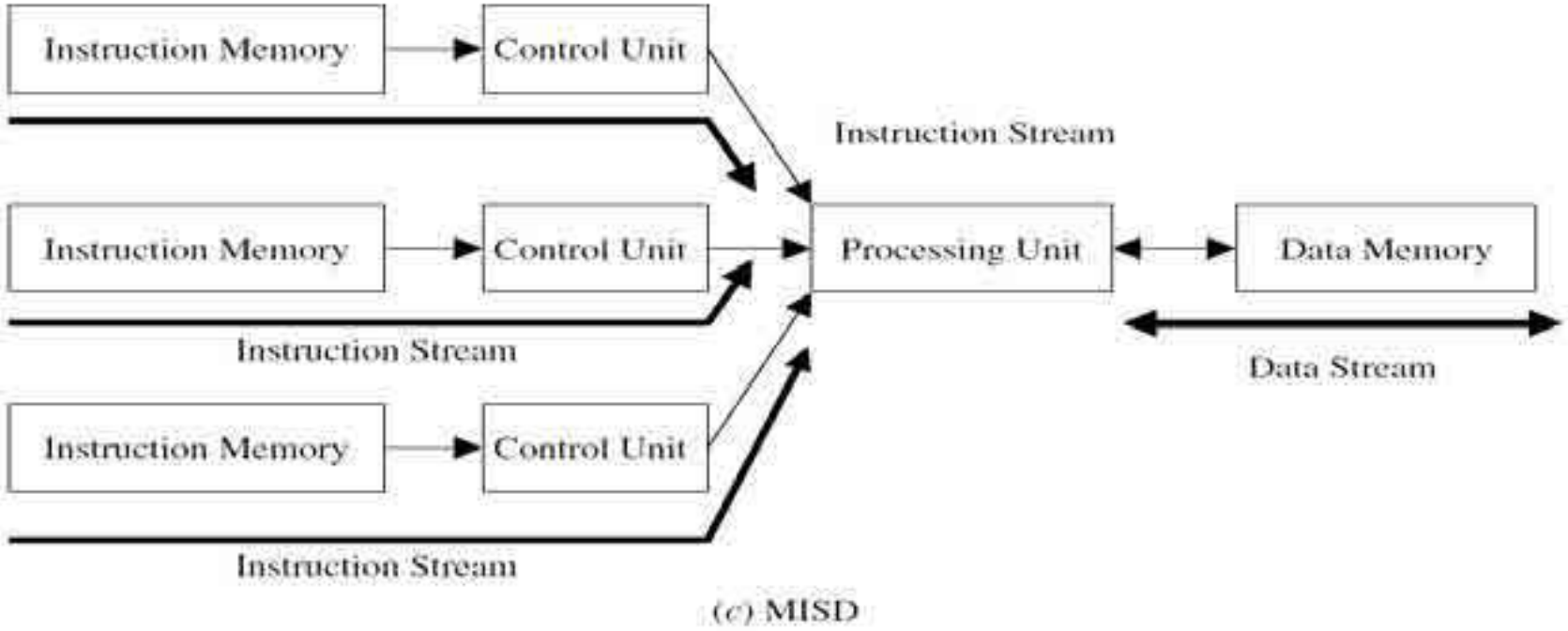
- Limitation on ***Memory Bandwidth***
- Memory is shared by CPU and I/O

Von Neumann Architecture



Schematic of the von Neumann architecture. The Control Unit and Arithmetic Logic Unit form the main components of the Central Processing Unit (CPU)

MISD COMPUTER SYSTEMS

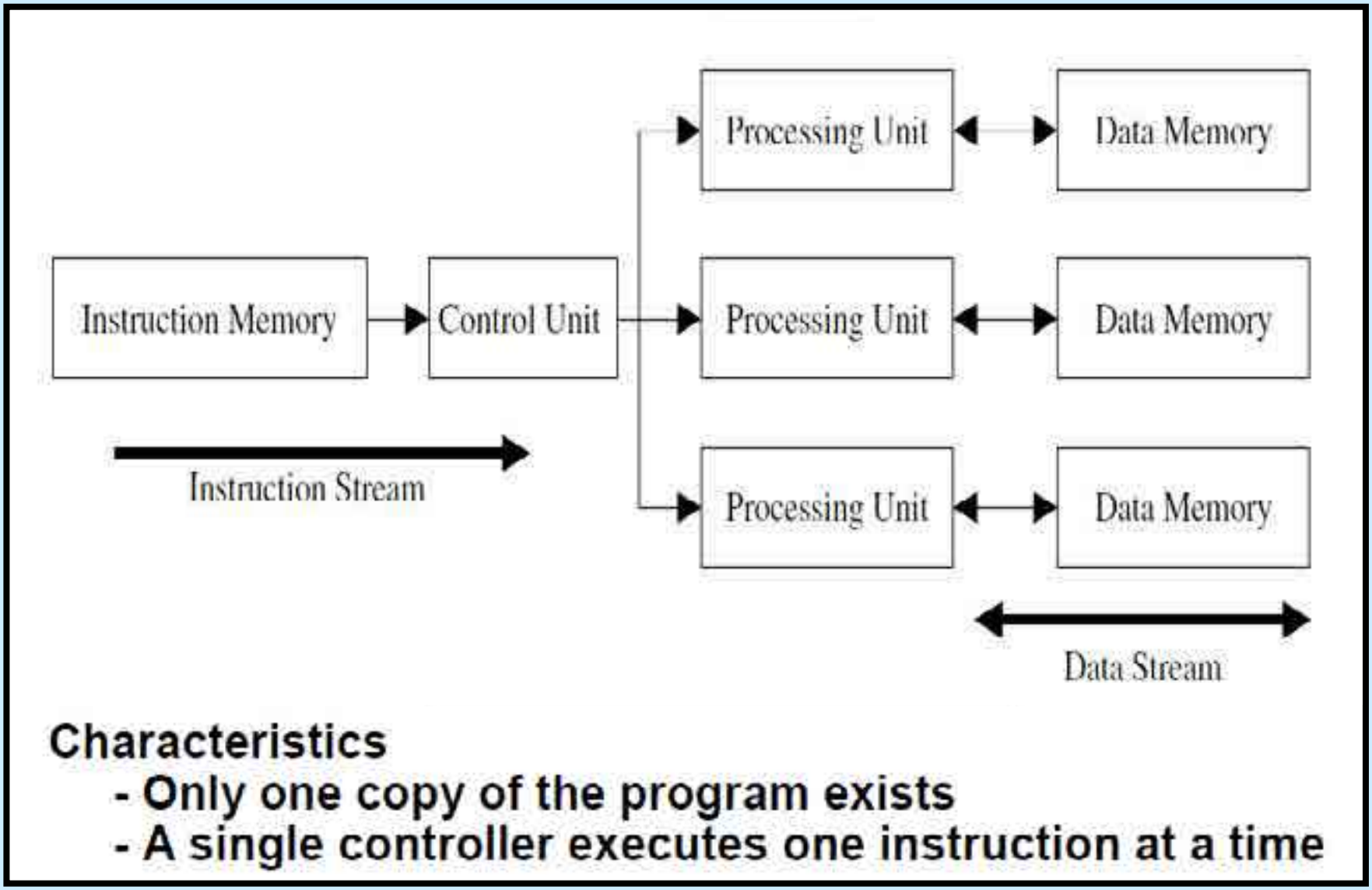


(c) MISD

Characteristics

- There is no computer at present that can be classified as MISD

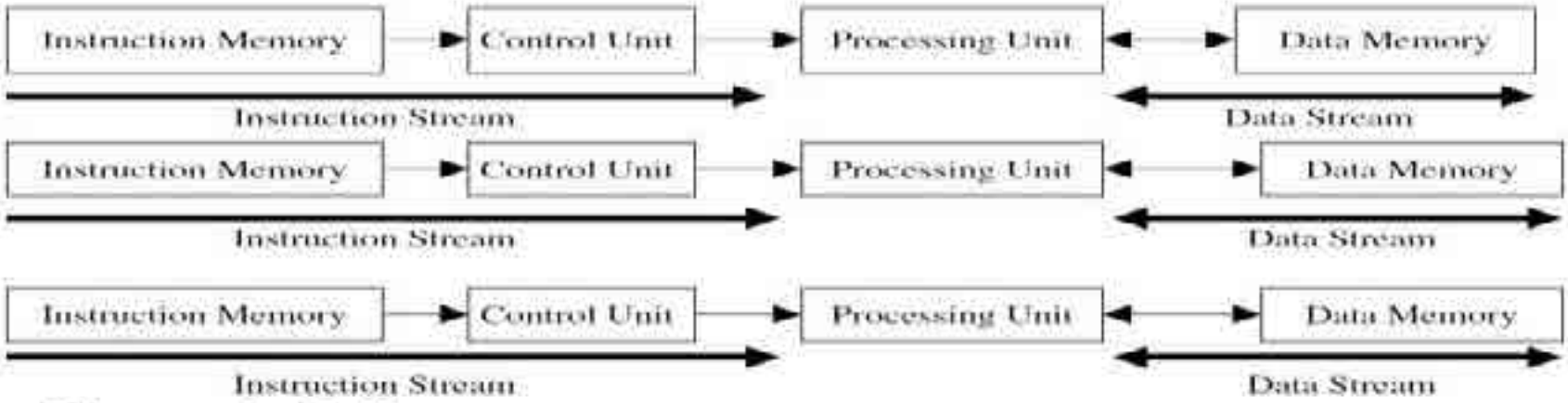
SIMD COMPUTER SYSTEMS



Characteristics

- Only one copy of the program exists
- A single controller executes one instruction at a time

MIMD COMPUTER SYSTEMS



Characteristics

- Multiple processing units
- Execution of multiple instructions on multiple data

Types of MIMD computer systems

- Shared memory multiprocessors
- Message-passing multicomputers

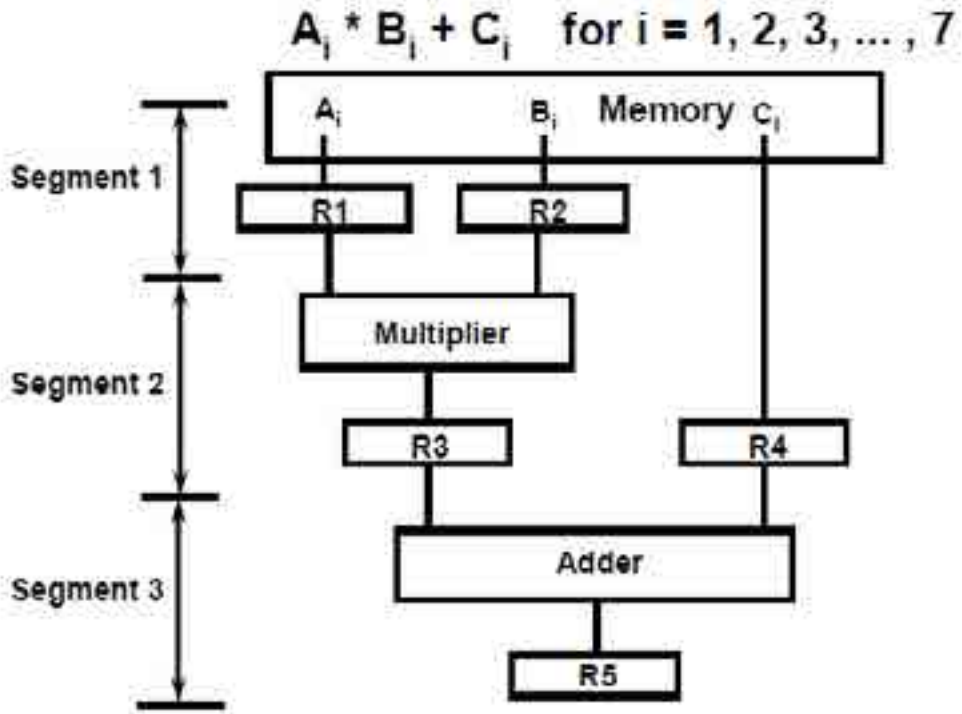
PIPELINING

A technique of decomposing a sequential process into suboperations, with each subprocess being executed in a partial dedicated segment that operates concurrently with all other segments.

A pipeline can be visualized as a collection of processing segments through which binary information flows.

The name "pipeline" implies a flow of information analogous to an industrial assembly line.

Example of the Pipeline Organization



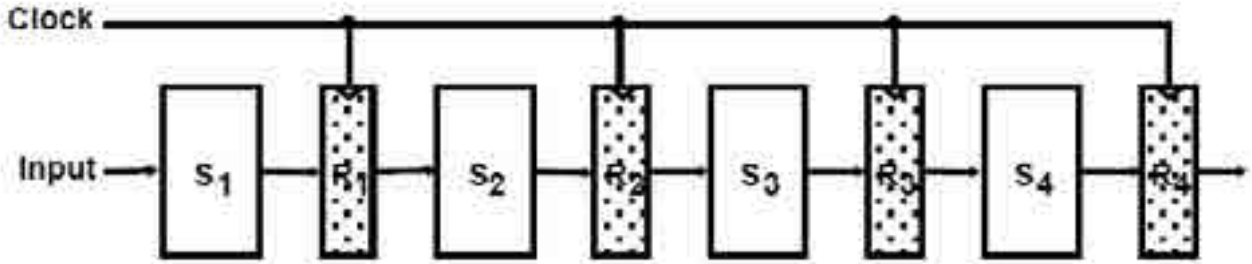
$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	Load A_i and B_i
$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$	Multiply and load C_i
$R5 \leftarrow R3 + R4$	Add

OPERATIONS IN EACH PIPELINE STAGE

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1			
2	A2	B2	A1 * B1	C1	
3	A3	B3	A2 * B2	C2	A1 * B1 + C1
4	A4	B4	A3 * B3	C3	A2 * B2 + C2
5	A5	B5	A4 * B4	C4	A3 * B3 + C3
6	A6	B6	A5 * B5	C5	A4 * B4 + C4
7	A7	B7	A6 * B6	C6	A5 * B5 + C5
8			A7 * B7	C7	A6 * B6 + C6
9					A7 * B7 + C7

GENERAL PIPELINE

General Structure of a 4-Segment Pipeline



Space-Time Diagram

Segment	1	2	3	4	5	6	7	8	9
	1	T1	T2	T3	T4	T5	T6		
	2		T1	T2	T3	T4	T5	T6	
	3			T1	T2	T3	T4	T5	T6
	4				T1	T2	T3	T4	T5	T6

Clock cycles

Behavior of the pipeline is illustrated with a space time diagram.

Space time diagram:

This shows the segment utilization as a function of time.

Cont.

Space Time diagram:

- The horizontal axis displays the time in clock cycle and vertical axis gives the segment number
- Diagram shows 6 task (T1 to T6)executed in four segment

Task :

is defined as the total operation performed going through all the segment in the pipeline

Speedup ratio of pipeline

Consider

- k: segment pipeline with clock cycle time t_p to execute n tasks
- first task T1 requires a time equal kt_p to complete its operation since there are k segments in the pipe .
- Remaining n-1 tasks emerge from the pipe at the rate of one task per clock cycle and they will complete after a time equal to (n-1)tp.
- Therefore to complete n task using k-segement pipeline requires
  K+(n-1) clock cycle
- Example 4 segment , 6task
  time required to complete op. 4+(6-1)=9
  clock cycle

Cont.

- For nonpipeline unit that perform the same operation and takes a time equal to t_n to complete each task.
- The total time required for n tasks $=nt_n$
- Speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio
- $S=nt_n / (K+n-1)t_p$
- As the number of tasks increases , n beomes larger the k-1, and k+n-1 approaches the value of n under this condition ,the speedup becomes $S=t_n / t_p$
- If we assume that the time it takes to process a task is the same in the pipeline and nonpipeline circuit, tn=ktp
- Including the assumption speedup reduces to $S=Kt_p/t_p=K$
- This shows that the theoretical max. speedup that a pipeline can provide is k, where k is the no. of segment in the pipeline

PIPELINE AND MULTIPLE FUNCTION UNITS

Example

- 4-stage pipeline
- subopertion in each stage; t_p = 20nS
- 100 tasks to be executed
- 1 task in non-pipelined system; 20*4 = 80nS

Pipelined System

$$(k + n - 1)*t_p = (4 + 99) * 20 = 2060nS$$

Non-Pipelined System

$$tn = n*k*t_p = 100 * 80 = 8000nS$$

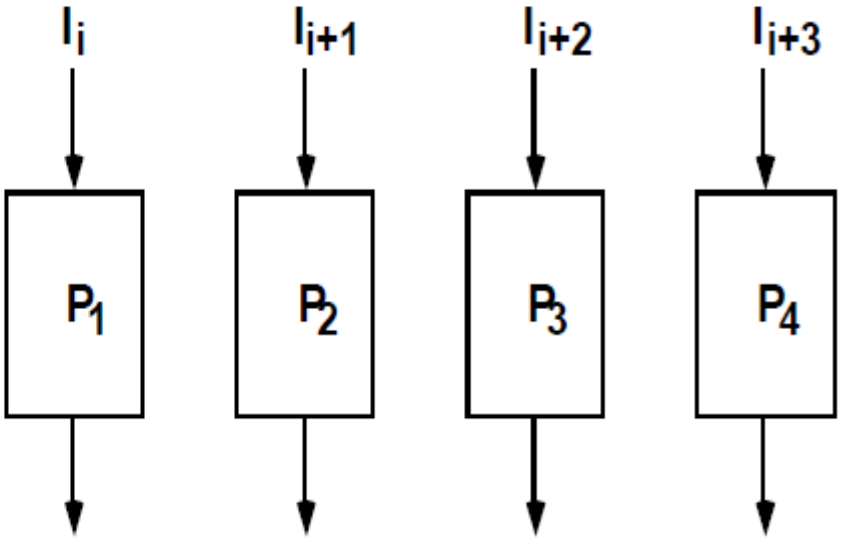
Speedup

$$S_k = 8000 / 2060 = 3.88$$

4-Stage Pipeline is basically identical to the system with 4 identical function units

Cont.

Multiple Functional Units

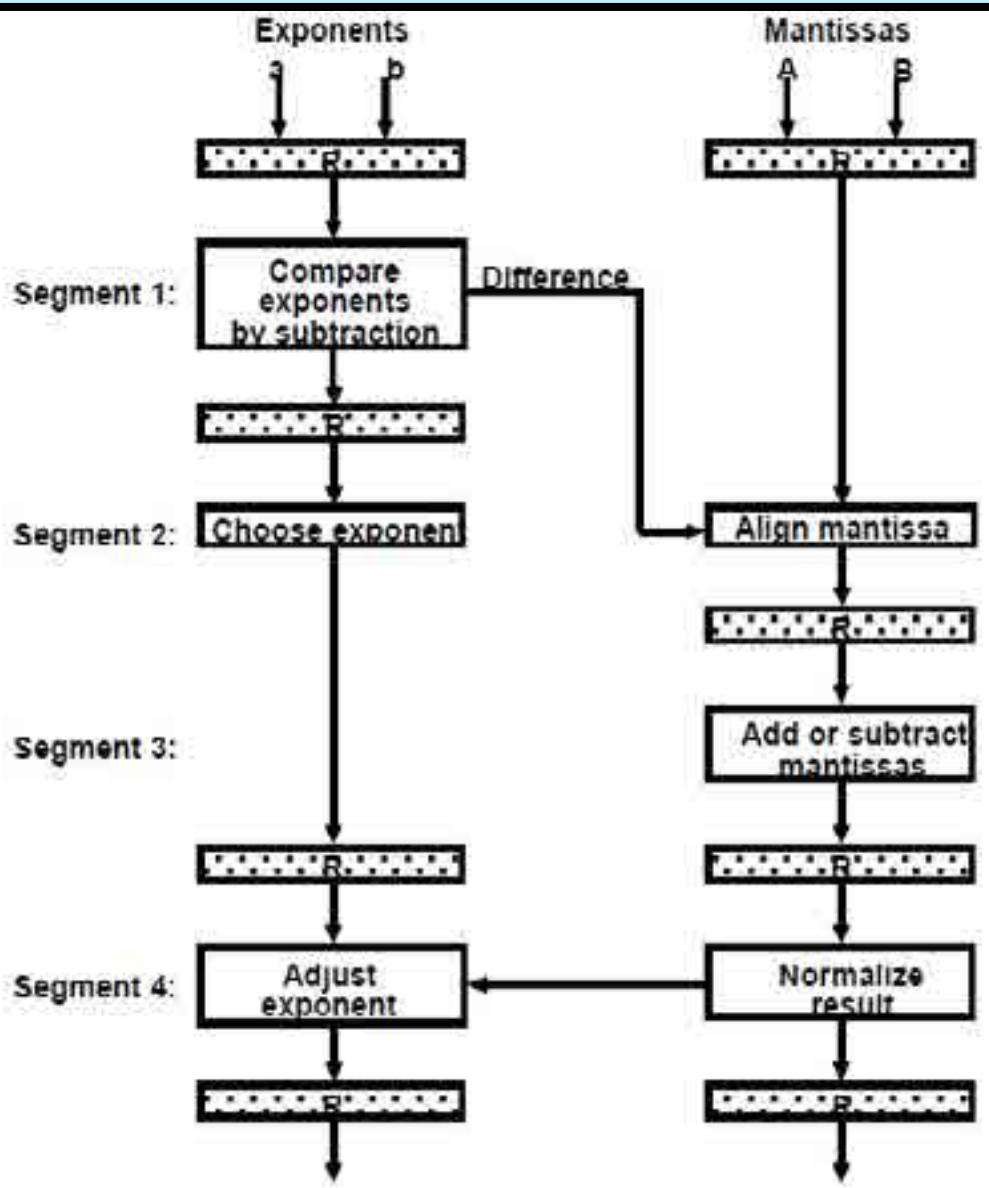


ARITHMETIC PIPELINE

Floating-point adder

$X = A \times 2^{a}$
$Y = B \times 2^{b}$

[1] Compare the exponents
[2] Align the mantissa
[3] Add/sub the mantissa
[4] Normalize the result



INSTRUCTION CYCLE

Six Phases* in an Instruction Cycle

[1] Fetch an instruction from memory
[2] Decode the instruction
[3] Calculate the effective address of the operand
[4] Fetch the operands from memory
[5] Execute the operation
[6] Store the result in the proper place

* Some instructions skip some phases
* Effective address calculation can be done in the part of the decoding phase
* Storage of the operation result into a register is done automatically in the execution phase

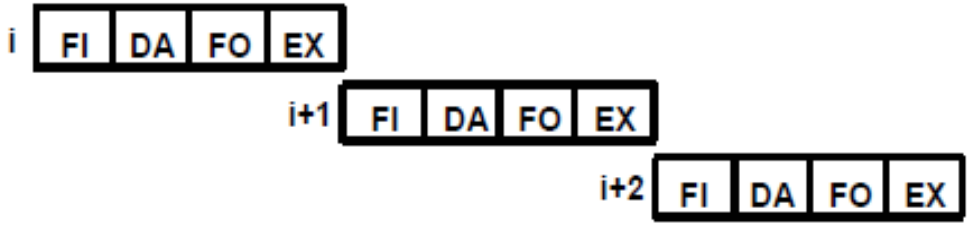
==> 4-Stage Pipeline

[1] FI: Fetch an instruction from memory
[2] DA: Decode the instruction and calculate the effective address of the operand
[3] FO: Fetch the operand
[4] EX: Execute the operation

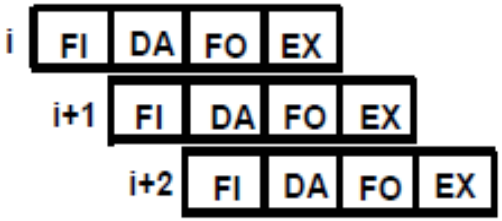
INSTRUCTION PIPELINE

Execution of Three Instructions in a 4-Stage Pipeline

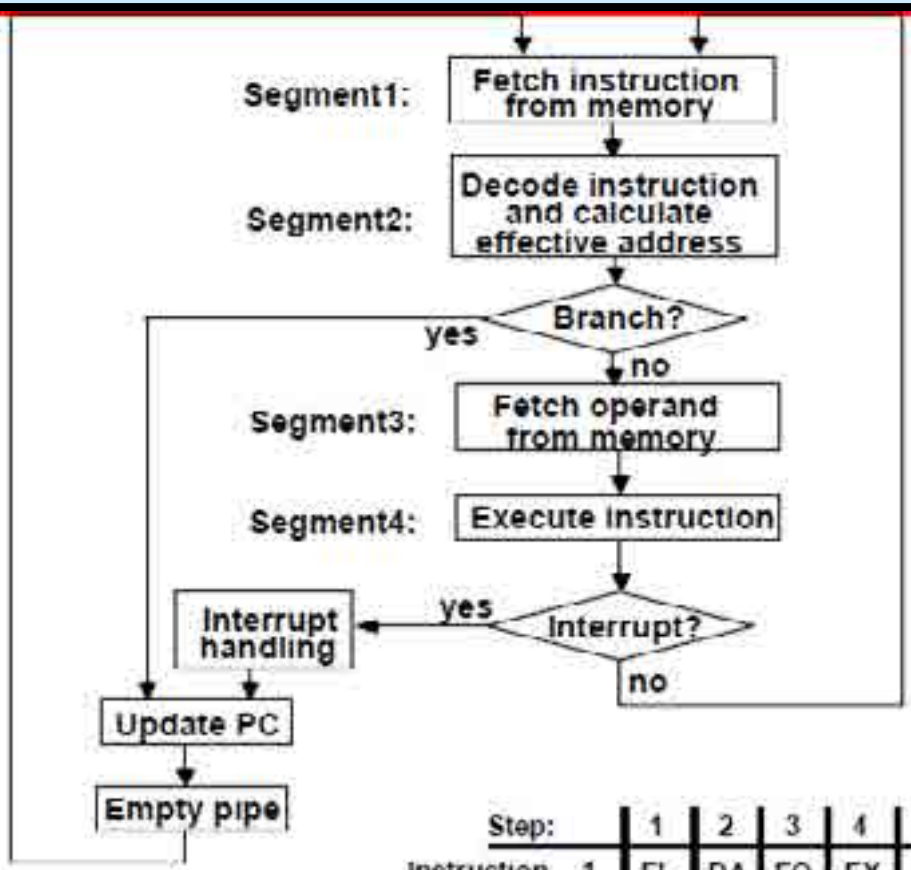
Conventional



Pipelined

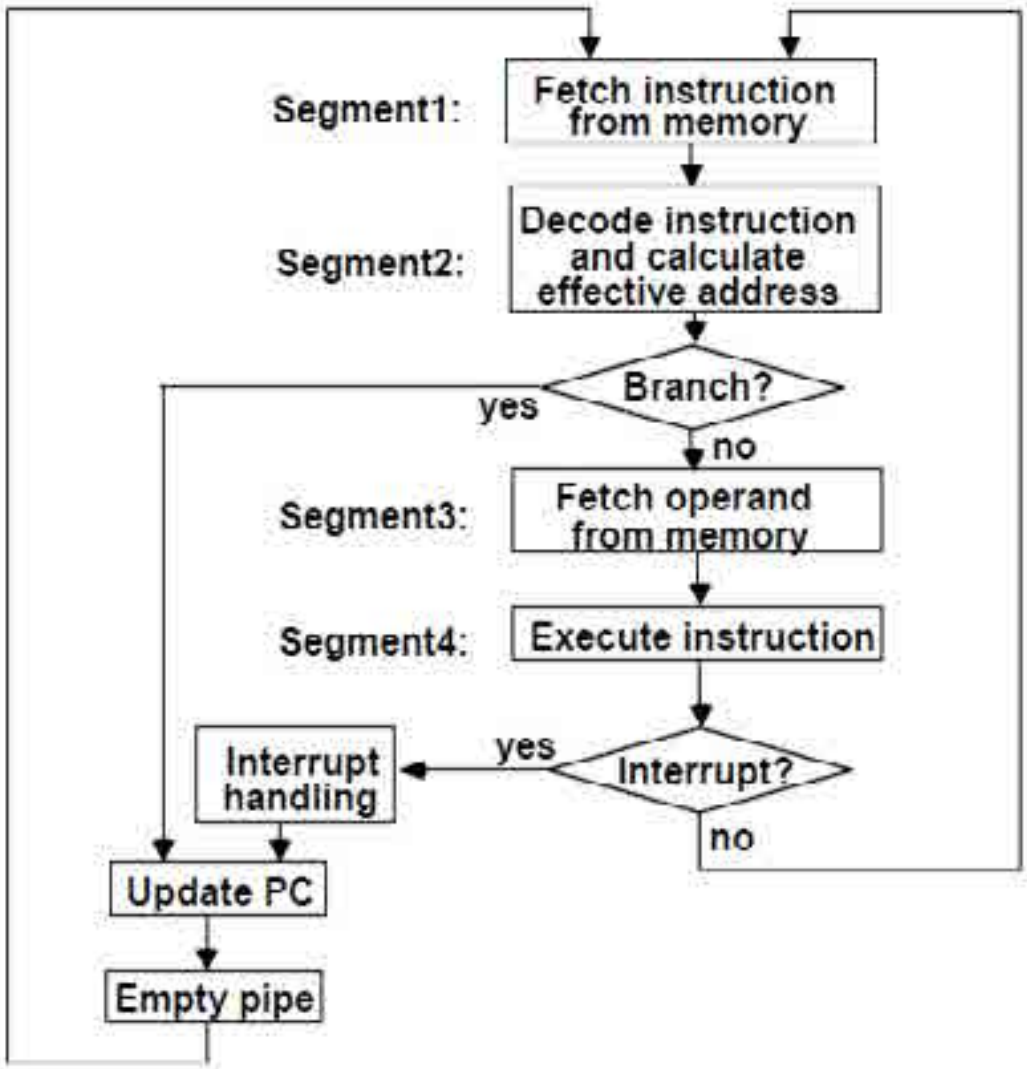


INSTRUCTION EXECUTION IN A 4-STAGE PIPELINE



	Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	-	-	FI	DA	FO	EX			
	5					-	-	-	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Pipeline


Segment1:
Fetch instruction from memory
Segment2:
Decode instruction and calculate effective address
Branch?
yes
no
Segment3:
Fetch operand from memory
Segment4:
Execute instruction
Interrupt?
yes
Interrupt handling
no
Update PC
Empty pipe

Space time diagram

	Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	-	-	FI	DA	FO	EX			
	5					-	-	-	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX