COMPUTER ARCHITECTURE

FOURTH YEAR ELECTRONIC
2016-2017

BY

Assist. Prof. Dr. EYAD I. ABBAS

جمهورية العراق
وزارة التعليم العالي و البحث العلمي
دائرة الدراسات والتخطيط والمتابعة
الدراسات والتخطيط

وزارة التعليم العالي والبحث العلمي

REPUBLIC OF IRAQ
MINISTRY OF HIGHER EDUCATION
AND SCIENTIFIC RESEARCH
STUDIES & PLANNING & FOLLOW UP DIRECTORATE
STUDIES & PLANNING DEPARTMENT

العدد : ت م٣/ ٤٢٨٤/١ التاريخ: ٢٠١٦/٩/٢٢

№ 305198

التقويم الجامعي للسنة الدراسية ٢٠١٦-٢٠١٧

تفاصيل العام الدراسي	اليوم	التاريخ	الملاحظات
بدء الفصل الدراسي الأول	الأحد	٢٠١٦/١٠/٢-٢٠١٧/١/١٢	١٥ أسبوع
تبدأ الامتحانات النهائية للفصل الدراسي الأول للكليات والمعاهد التي تتبع النظام الفصلي والامتحانات النصف سنوية للكليات والمعاهد التي تتبع النظام السنوي	الأحد	٢٠١٧/١/١٥-٢٠١٧/٢/٢	ثلاثة أسابيع
العطلة الربيعية	الأحد	٢٠١٧/٢/٥-٢٠١٧/٢/١٨	أسبوعان
بدء الفصل الدراسي الثاني	الأحد	٢٠١٧/٢/١٩-٢٠١٧/٦/١	١٥ أسبوع
بدء الامتحانات النهائية للدور الأول للكليات والمعاهد التي تتبع النظام السنوي والامتحانات النهائية للفصل الدراسي الثاني للكليات والمعاهد التي تتبع النظام الفصلي	الأحد	٢٠١٧/٦/٤-٢٠١٧/٦/٢٢	ثلاثة أسابيع
بدء العطلة الصيفية		٢٠١٧/٧/١	شهران
بدء التدريب الصيفي لطلبة الكليات والمعاهد المشمول به وحسب المدد المعتمدة في التعليمات الخاصة بالتدريب الصيفي.		٢٠١٧/٧/١-٢٠١٧/٨/٣١	
بدء امتحانات الدور الثاني		٢٠١٧/٩/١	

- أما بالنسبة لطلبة المرحلة الأولى فيتم تعويض النقص في الفصل الدراسي بالاستفادة من العطلة الربيعية وأيام السبت إذا دعت الحاجة إلى ذلك.

الصفحة الرسمية لوزارة التعليم العالي والبحث العلمي
www.facebook.com/moheiq

وزارة التعليم العالي والبحث العلمي

COMPUTER ARCHITECTURE SYLLABUS

EE 405

Year: FOURTH **Theoretical: 3 hrs/Week**

1. Register Transfer and Microoperation: **8Hrs.**

Register transfer language, Register transfer, Bus and Memory Transfer, Arithmetic and Logic Microoperations, Shift Microoperations, Arithmetic Logic Shift Unit, Control Function.

2. Basic Computer Organization and Design: **12Hrs.**

Instruction Code, Computer Register, Computer Instructions, Timing and Control, Memory – Reference Instruction, Input – Output and Interrupt, Complete Computer Description, Design of Basic Computer, Design of Accumulator Logic.

3. Micro Programmed Control: **8Hrs.**

Control Memory, Address Sequencing, Micro Program Example, Design of Control Time.

4. Central Processing Unit: **8Hrs.**

General Register and Stack Organization, Instruction Formats, Addressing Modes, Data Transfer and Manipulation, Reduced Instruction Set Computer.

5. Memory Management: **10Hrs.**

Memory Hierarchy, Auxiliary Memory, Main Memory, Associative Memory, Cache Memory, Virtual Memory Management, Memory Management Hardware.

6. Input–Output Organization: **8Hrs.**

Peripheral Devise, Asynchronous Data Transfer, Mode of Transfer, Direct Memory Access (DMA), Input – Output Processor (IOP).

7. Pipelining: **12Hrs.**

General Considerations, Comparison of Pipelined and Nonpipelined Computers, Instruction Pipeline, Example Pipeline Processors, Instruction-Level Parallelism, Arithmetic Pipeline, Structural Hazards and Data Dependencies, Branch Delay and Multicycle Instructions, Superscalar Computers.

8. Computer Arithmetic: **6Hrs.**

Addition, Subtraction, Multiplication and Division Algorithm, Decimal arithmetic operations.

9. Reduced Instruction Set Computers (RISCs) **8 Hrs.**

RISC/CISC Evolution Cycle, RISCs Design Principles, Overlapped Register Windows, RISCs Versus CISCs, Pioneer (University) RISC Machines, Advanced RISC Machines.

10. Multiprocessors and Multiple Computers **12 Hrs.**

Classification of Computer Architectures, SIMD Schemes, MIMD Schemes, Interconnection Networks, Centralized and distributed Shared Memory- Architectures, Cache Coherence.

References:

1. Mano, M. Morris, Computer System Architecture, 3rd Edition, Prentice-Hall, Inc., 1993.
2. Mostafa Abd-El-Barr, Hesham El-Rewini, "Fundamentals of Computer Organization and Architecture", A John Wiley & Sons, Inc Publication, 2005.
3. M. Morris Mano, Computer Engineering Hardware Design, 1st Edition, Prentice-Hall, Inc., 1988.

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):

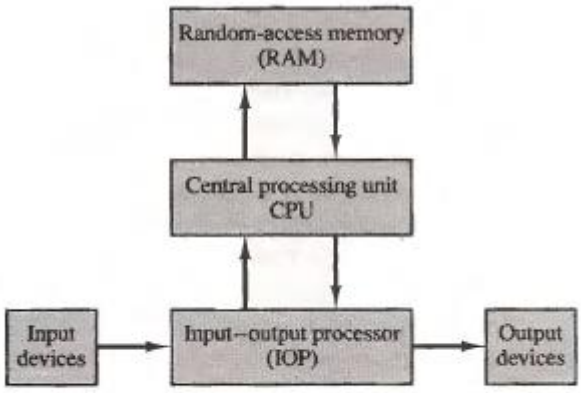


Figure 1 Block diagram of a digital computer.

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
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. 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

microoperations. A microoperation is an elementary operation 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 microoperations 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 microoperations performed on the binary information stored in the registers.
3- The control that initiates the sequence of microoperations.

The symbolic notation used to describe the microoperation 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 microoperation 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 microoperation sequences among the registers of a digital module.

Register Transfer

Computer registers are designated 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):

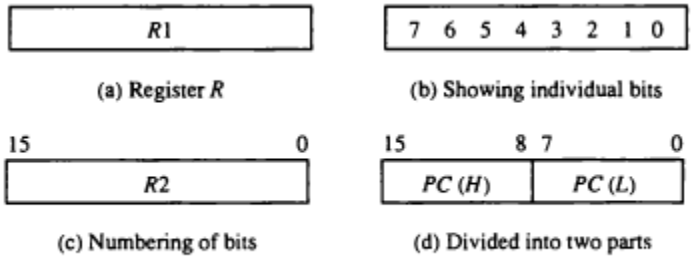


Figure 2 Block diagram of register.

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). 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.

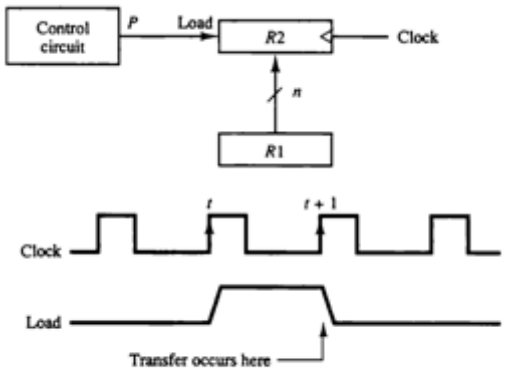
If we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

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

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.

$$P: R2 \leftarrow R1$$

The control condition is terminated with a colon. It symbolizes 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 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 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S1 and S0.

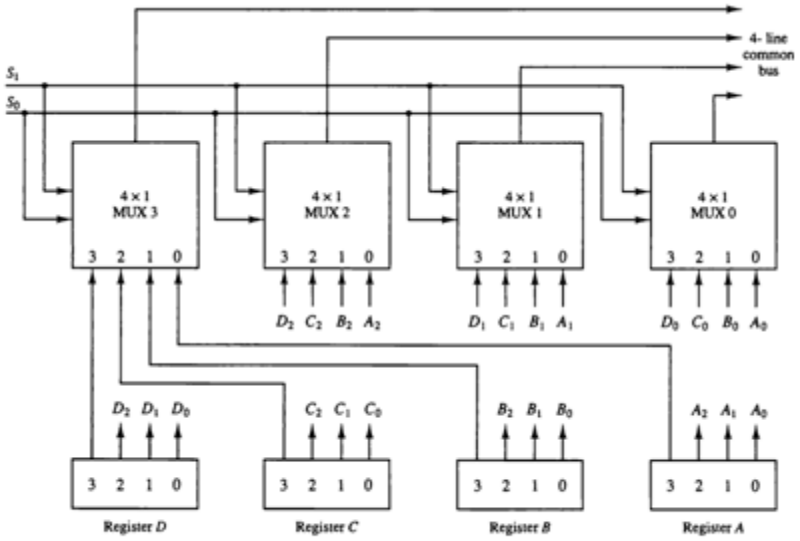


Fig 3 **Bus system for four registers**

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*

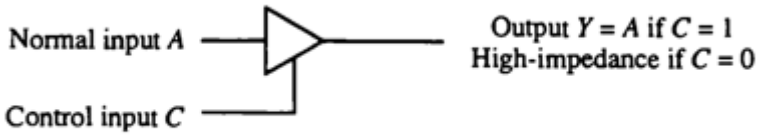
The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is includes in the statement, the register transfer is symbolized as follows:

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

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.

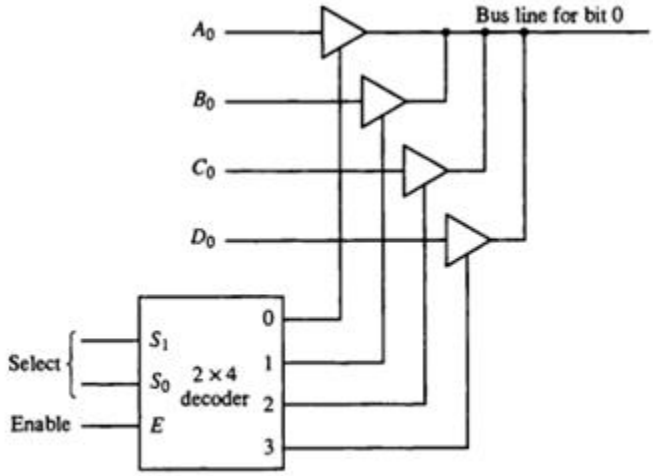


Figure 4 Bus line with three state-buffers.

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.

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

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

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

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 microoperations. In most computers, the multiplication operation is implemented with a sequence of add and shift microoperations. Division is implemented with a sequence of subtract and shift microoperations.

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).

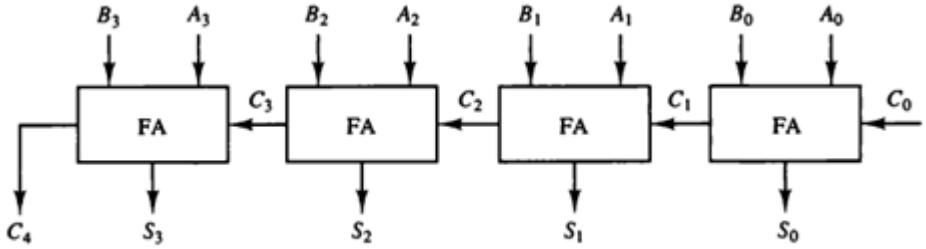


Figure 5 4-bit binary adder.

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).

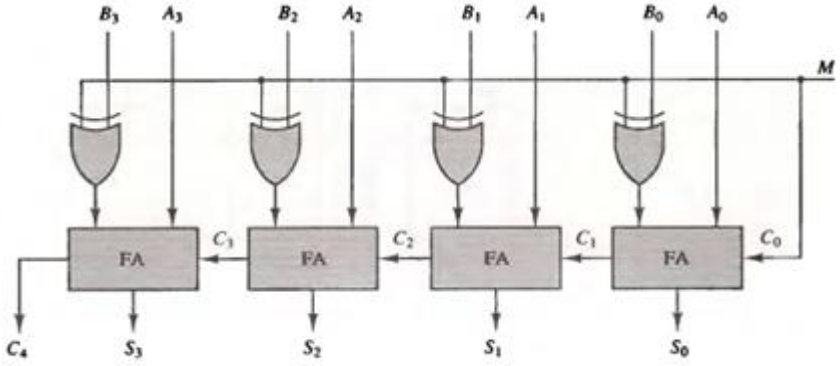


Figure 6 4-bit adder-subtractor.

The increment microoperation 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 4-bit combinational circuit incrementer is shown in Fig(7):

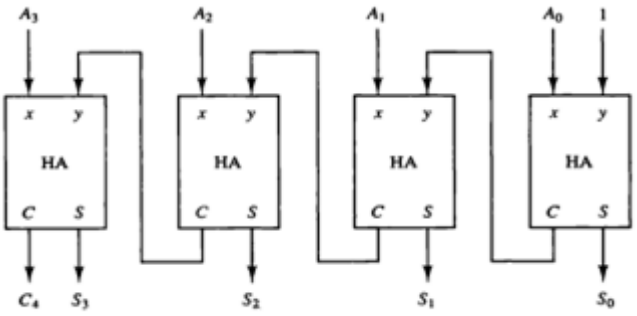


Figure 7 4-bit binary incrementer.

The arithmetic microoperations listed in the Table above 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).

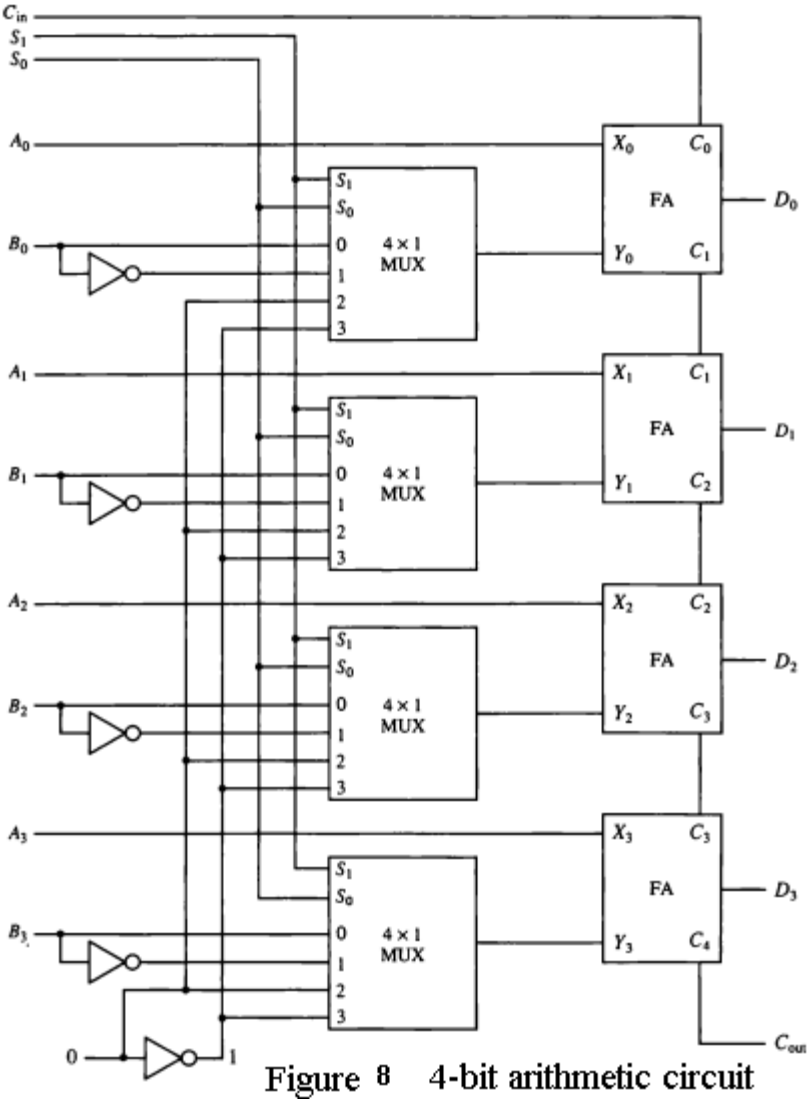


Figure 8 4-bit arithmetic circuit

It is possible to generate the eight arithmetic microoperations 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 Microoperations

Logic microoperations 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 microoperation with the contents of two registers Rl and R2 is symbolized by the statement:

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

It specifies a logic microoperation 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 microoperation stated above symbolizes the following logic computation:

$$\begin{array}{ll} 1010 & \text{Content of } R1 \\ \underline{1100} & \text{Content of } R2 \\ 0110 & \text{Content of } R1 \text{ after } P = 1 \end{array}$$

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

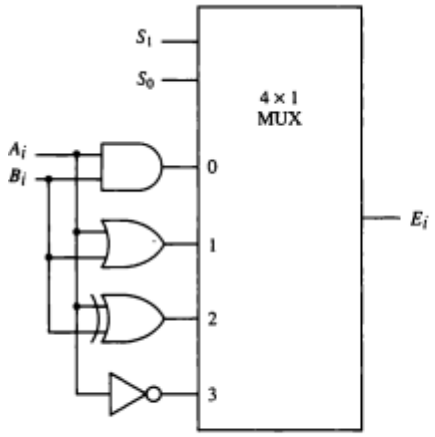
x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

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 microoperations 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	$E = \overline{A}$	Complement

(b) Function table

Figure 9 One stage of logic circuit.

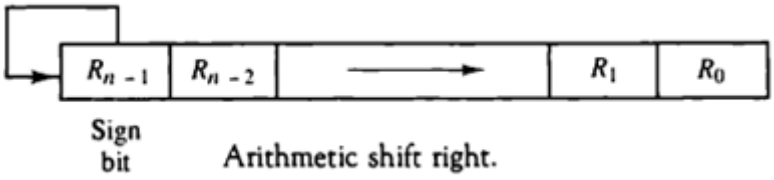
Shift Microoperations

Shift microoperations 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 microoperations 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

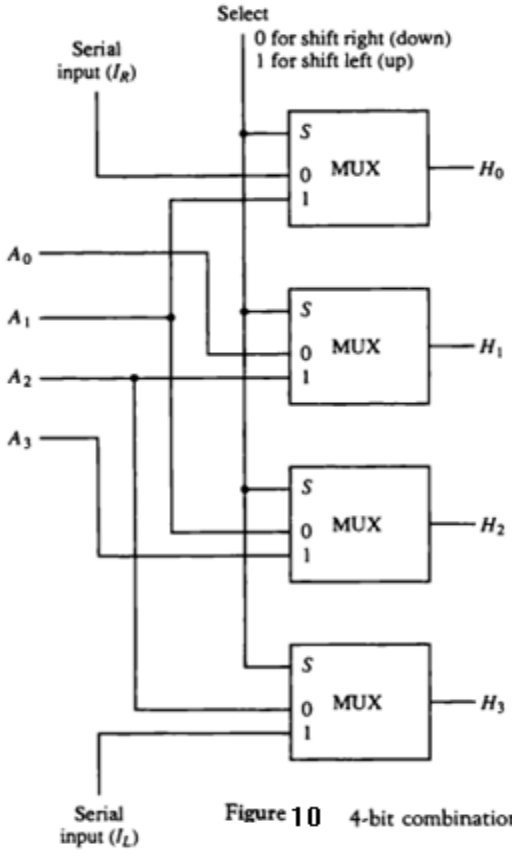
An arithmetic shift is a microoperation that shifts a signed binary number to the left or right. The arithmetic shift-left inserts a 0 into R_0, and shifts all other bits to the left. The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2} . A sign reversal occurs if the bit in R_{n-1}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

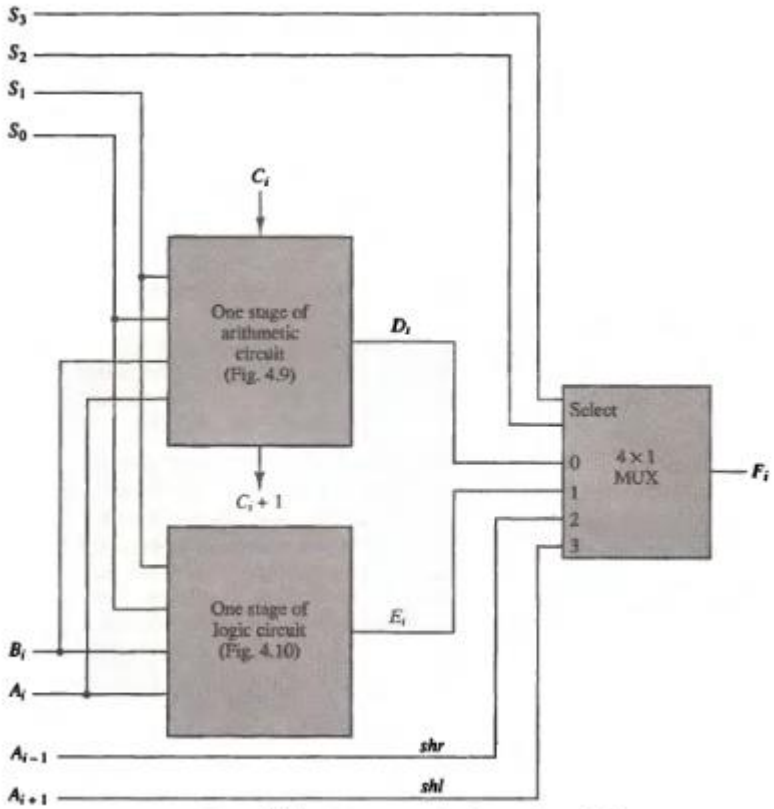


Figure 11 One stage of arithmetic logic shift unit.

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

Instruction Codes

A computer instruction is a binary code that specifies a sequence of microoperations for the computer. Instruction codes together with data are stored in memory. The computer reads each instruction from memory and places it in a control register. The control then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of microoperations. Every computer has its own unique instruction set. The ability to store and execute instructions, the stored program concept, is the most important property of a general-purpose computer. An instruction code is a group of bits that instruct the computer to perform a specific operation. It is usually divided into parts, each having its own particular interpretation. The most basic part of an instruction code is its operation part. The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement. The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer.

The simplest way to organize a computer is to have one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

For a memory unit with 4096 words we need 12 bits to specify an address since 2^{12}= 4096. If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand. The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. Fig(12) depicts this type of organization.

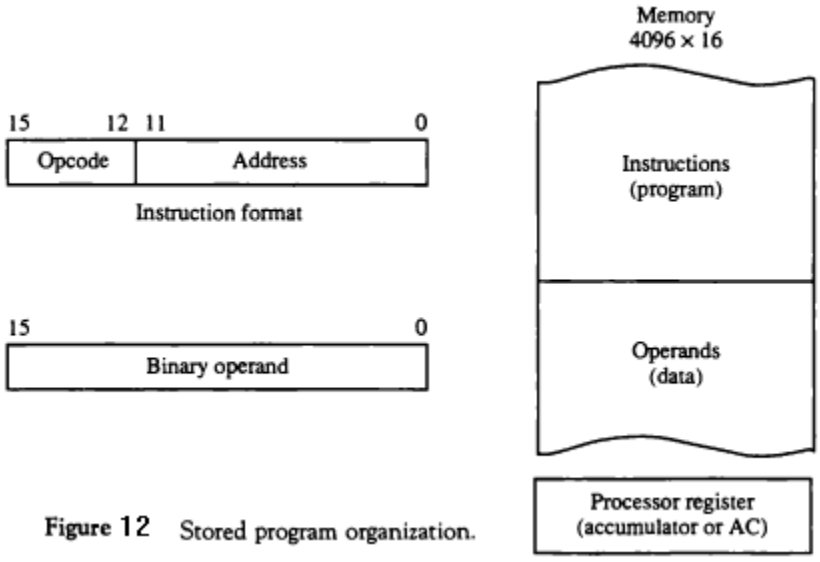


Figure 12 Stored program organization.

Computers that have a single-processor register usually assign to it the name accumulator and label it AC.

For example, operations such as clear AC, complement AC, and increment AC operate on data stored in the AC register.

When the second part of an instruction immediate code specifies an operand, this type is called **immediate operand**. When the second part specifies the address of an operand, the instruction is said to have a **direct address**. The third possibility called **indirect address**, where the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found. One bit of the instruction code can be used to distinguish between a direct and an indirect address.

As an illustration of this configuration, consider the instruction code format shown in Fig(13).

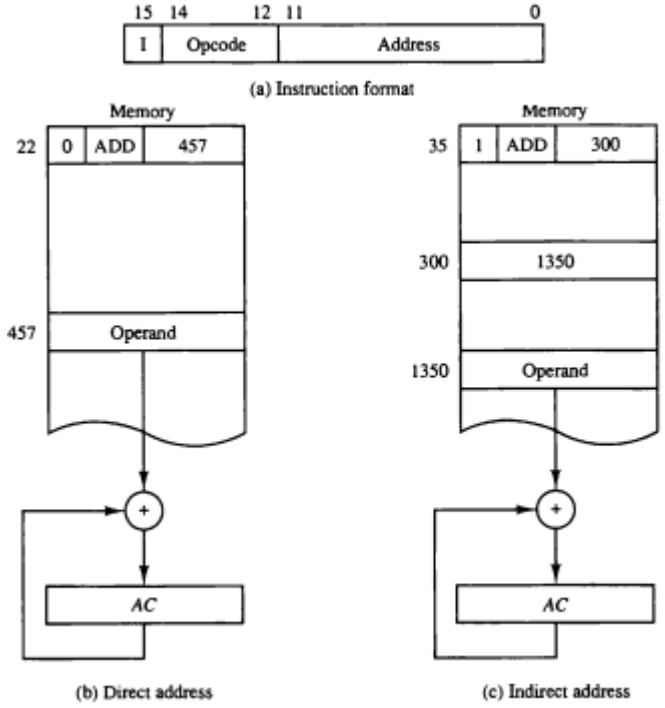


Fig 13 Demonstration of direct and indirect address.

Computer Registers

Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time. The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence and executes it, and so on. This type of instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed. The computer needs processor registers for manipulating data and a register for holding a memory address. These requirements dictate the register configuration are listed in Table(9) and shown in Fig(14).

TABLE 9 List of Registers for the Basic Computer

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

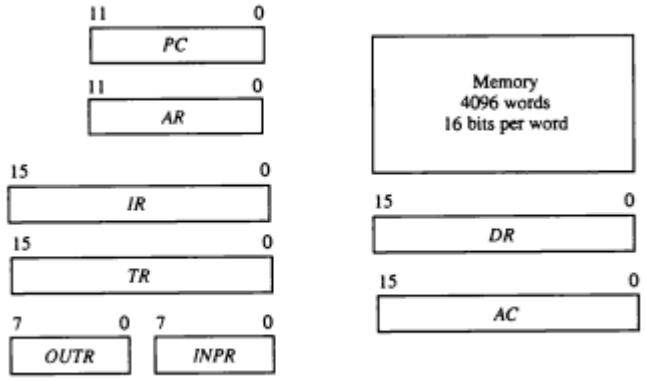


Figure 14 Basic computer registers and memory.

The basic computer has eight registers, a memory unit, and a control unit. The connection of the registers and memory of the basic computer to a common bus system is shown in Fig(15).

Two registers, AR and PC, have 12 bits each since they hold a memory address. When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to 0's. When AR or PC receive information from the bus, only the 12 least significant bits are transferred into the register.

The input register INPR and the output register OUTR have 8 bits each and communicate with the eight least significant bits in the bus. The INPR receives a character from an input device which is then transferred to AC. OUTR receives a character from AC and delivers it to an output device. There is no transfer from OUTR to any of the other registers.
Five registers have three control inputs: LD (load), INR (increment), and CLR (clear).

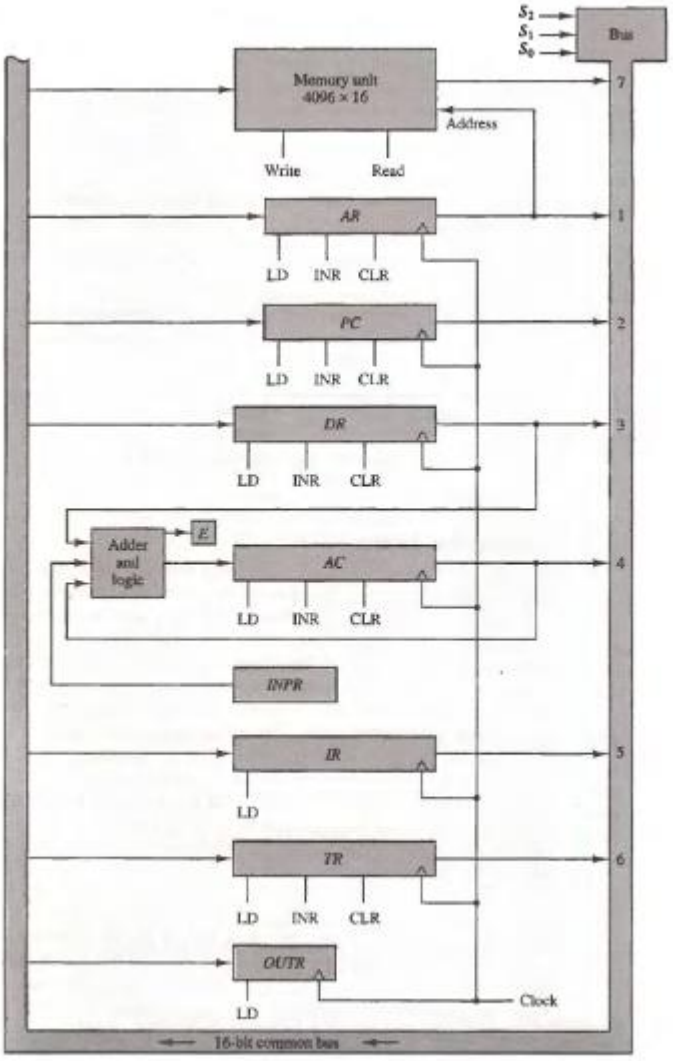


Figure 15 Basic computer registers connected to a common bus.

Computer Instructions

The basic computer has three instruction code formats, as shown in Figure. Each format has 16 bits.

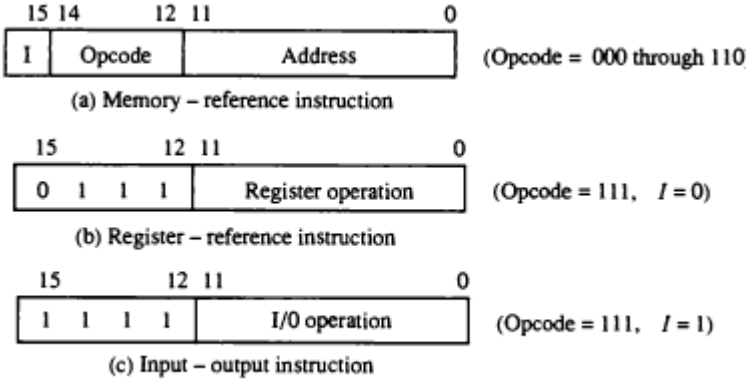


Figure 16 Basic computer instruction formats.

The type of instruction is recognized by the computer control from the four bits in positions 12through 15 of the instruction. If the three opcode bits in positions 12 through 14 are not equal to 111, the instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode J. If the 3-bit opcode is equal to 111, control then inspects the bit in position 15. If this bit is 0, the instruction is a register-reference type. If the bit is 1, the instruction is an input-output type. Note that the bit in position 15 of the instruction code is designated by the symbol J but is not used as a mode bit when the operation code is equal to 111.

The instructions for the computer are listed in Table(10):

TABLE 10 Basic Computer Instructions

Symbol	Hexadecimal code $I = 0$	Hexadecimal code $I = 1$	Description
AND	0xxx	8xxx	AND memory word to *AC*
ADD	1xxx	9xxx	Add memory word to *AC*
LDA	2xxx	Axxx	Load memory word to *AC*
STA	3xxx	Bxxx	Store content of *AC* in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear *AC*
CLE	7400		Clear *E*
CMA	7200		Complement *AC*
CME	7100		Complement *E*
CIR	7080		Circulate right *AC* and *E*
CIL	7040		Circulate left *AC* and *E*
INC	7020		Increment *AC*
SPA	7010		Skip next instruction if *AC* positive
SNA	7008		Skip next instruction if *AC* negative
SZA	7004		Skip next instruction if *AC* zero
SZE	7002		Skip next instruction if *E* is 0
HLT	7001		Halt computer
INP	F800		Input character to *AC*
OUT	F400		Output character from *AC*
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions.
2. Instructions for moving information to and from memory and processor registers.
3. Instructions that check status information to provide decision making capabilities.
4. Input and output instructions.
5. The capability of stopping the computer.

Timing and Control

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

There are two major types of control organization:

1- **hardwired control** : the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation.

2- **microprogrammed control:** the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations.

The block diagram of the control unit is shown in Fig(17):

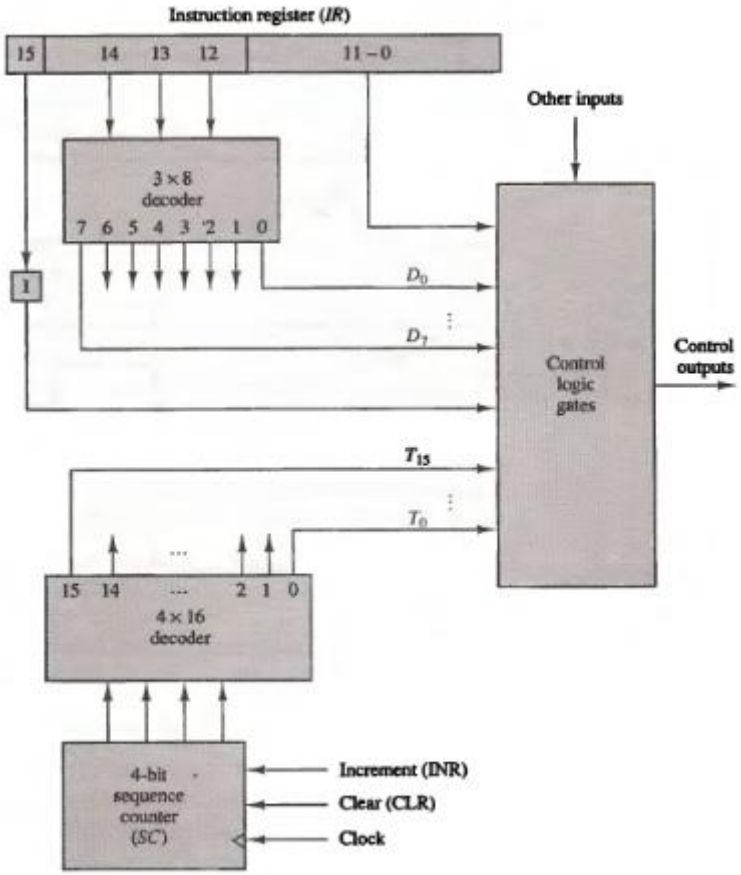


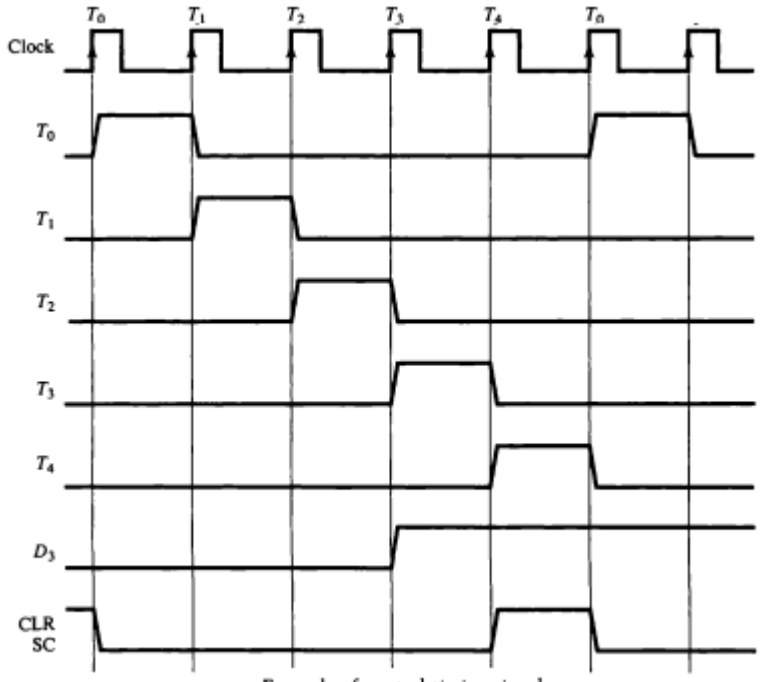
Figure 17 Control unit of basic computer.

SC is incremented with every positive clock transition, unless its CLR input is active. This produces the sequence of timing signals T_0, T_1, T_2, T_3, T_4, and so on, as shown in the diagram. (Note the relationship between the timing signal and its corresponding positive

clock transition.) If SC is not cleared, the timing signals will continue with T_5, T_6, up to T_{15} and back to T_0.

As an example, consider the case where SC is incremented to provide timing signals T_0, T_1 T_2, T_3, and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D_3 is active. This is expressed symbolically by the statement

$$D_3T_4: SC \leftarrow 0$$



Example of control timing signals.

Instruction Cycle

A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory if the instruction has an indirect address.
4. Execute the instruction.

This process continues indefinitely unless a HALT instruction is encountered. Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T_0. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2, and so on. The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

$$T_0: AR \leftarrow PC$$

$$T_1: IR \leftarrow M[AR], PC = PC + 1$$

$$T_2: D_0,,,,D_7 \leftarrow Decode\ IR(12-14), AR \leftarrow IR(0-11), I \leftarrow IR(15)$$

It is necessary to use timing signal T_1 to provide the following connections in the bus system.

1. Enable the read input of memory.
2. Place the content of memory onto the bus by making $S_2S_1S_0$ =111.
3. Transfer the content of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

Memory - Reference Instructions

The table(11) lists the seven memory-reference instructions. The decoded output D, for i = 0,1, 2, 3, 4, 5, and 6 from the operation decoder that belongs effective address to each instruction is included in the table. The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I=1. The symbolic description of each instruction is specified in the table in terms of register transfer notation. The actual execution of the instruction in the bus system will require a sequence of microoperations.

TABLE 11 Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description	Means
AND	D_0	$AC \leftarrow AC \wedge M[AR]$	AND to AC
ADD	D_1	$AC \leftarrow AC + M[AR], \quad E \leftarrow C_{out}$	ADD to AC
LDA	D_2	$AC \leftarrow M[AR]$	LOAD to AC
STA	D_3	$M[AR] \leftarrow AC$	STORE AC
BUN	D_4	$PC \leftarrow AR$	Branch Unconditional
BSA	D_5	$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$	Branch and save address
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1,$ If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$	Increment and skip if 0

Input - Output and Interrupt

A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. The input-output configuration is shown in Fig(18). The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially. The 1-bit input flag FGI is a control flip-flop. The flag bit is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer. The output register OUTR works similarly but the direction of information flow is reversed. Initially, the output flag FGO is set to 1. The computer checks the flag bit; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0.

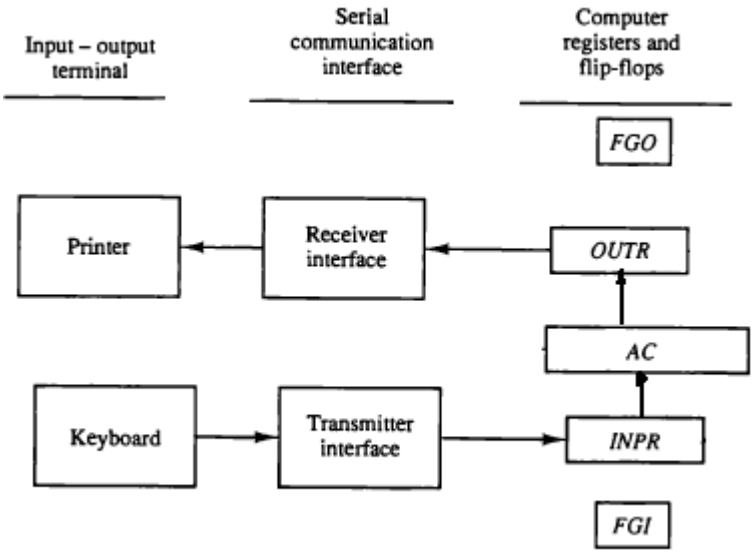


Figure 18 Input-output configuration.

Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility. Input-output instructions have an operation code 1111 and are recognized by the control when D7 = 1 and I = 1. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input-output instructions are listed in Table(12). These instructions are executed with the clock transition associated with timing signal T_3.

TABLE 12 Input-Output Instructions

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$	Output character
SKI	pB_9:	If ($FGI = 1$) then ($PC \leftarrow PC + 1$)	Skip on input flag
SKO	pB_8:	If ($FGO = 1$) then ($PC \leftarrow PC + 1$)	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

Consider a computer that can go through an instruction cycle in 1μs. Assume that the input-output device can transfer information at a maximum rate of 10 characters per second. This is equivalent to one character every 100,000μs. Two instructions are executed when the computer checks the flag bit and decides not to transfer the information. This means that at the maximum rate, the computer will check the flag 50,000 times between each transfer. The computer is wasting time while checking the flag instead of doing some other useful processing task.

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.

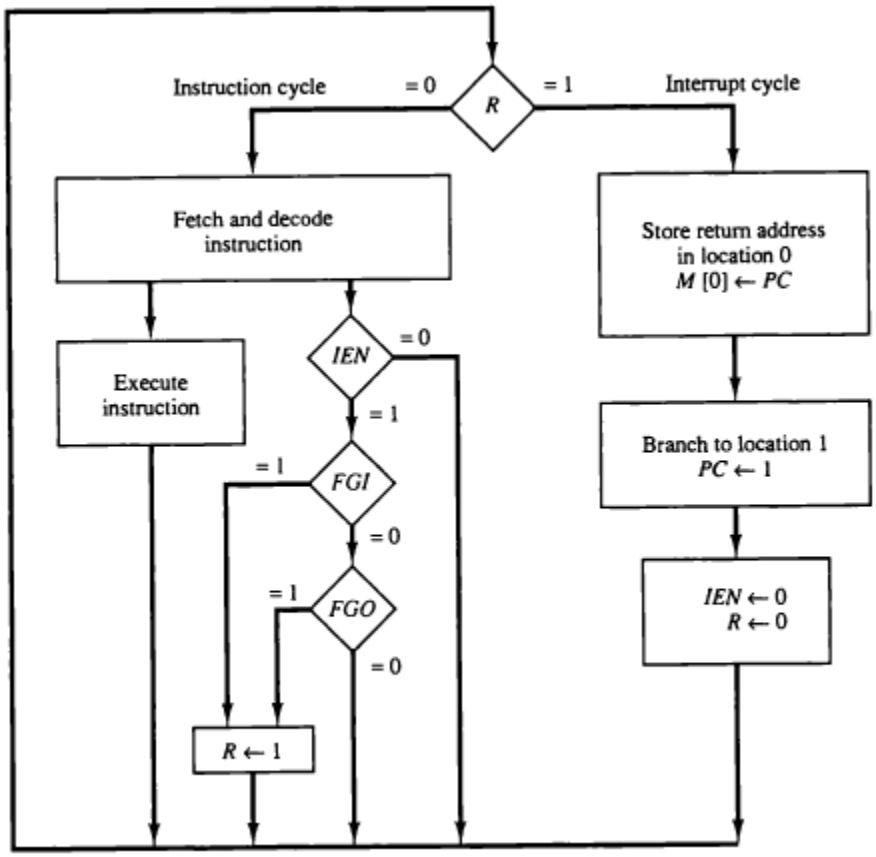


Figure 19 Flowchart for interrupt cycle.

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.

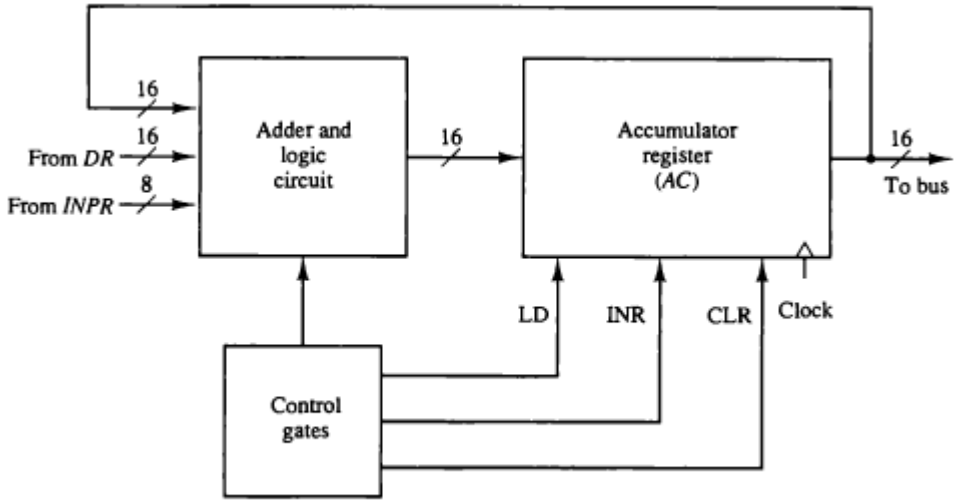


Figure 20 Circuits associated with AC.

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).

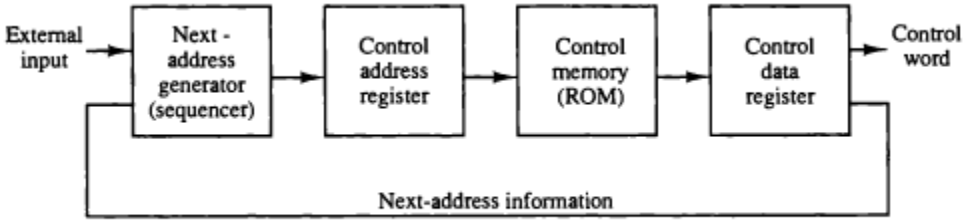


Figure 21 Microprogrammed control organization.

Address Sequencing

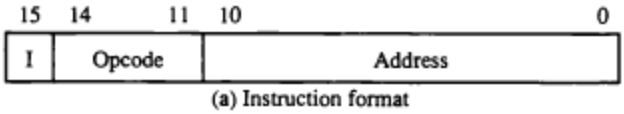
Microinstructions are stored in control memory in groups, with each group routine specifying a routine. An initial address is loaded into the control address register when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction 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 1-bit held for indirect addressing symbolized by J, a 4-bit operation code (opcode), and an 11-bit 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

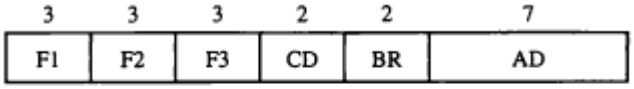
Figure 22 Computer instructions.

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).

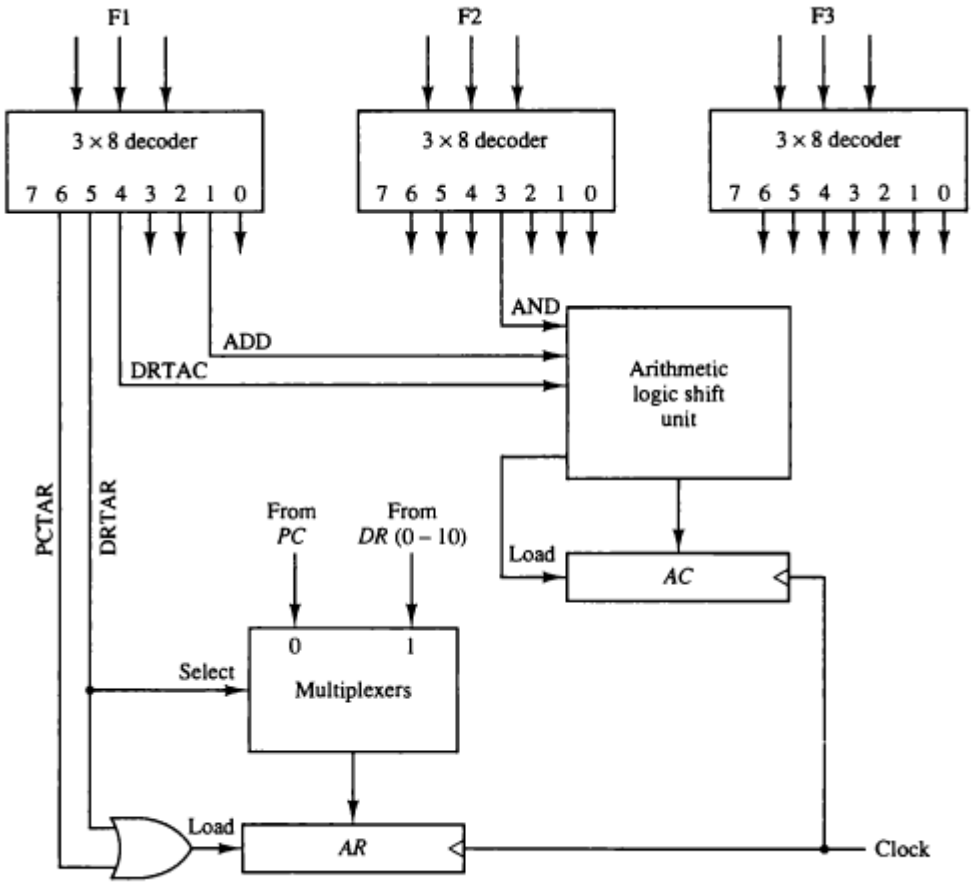


Figure 24 Decoding of microoperation fields.

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.

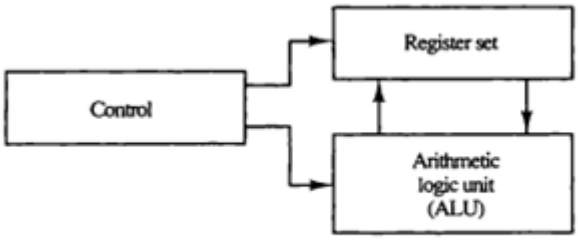


Figure 25 Major components of CPU.

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):

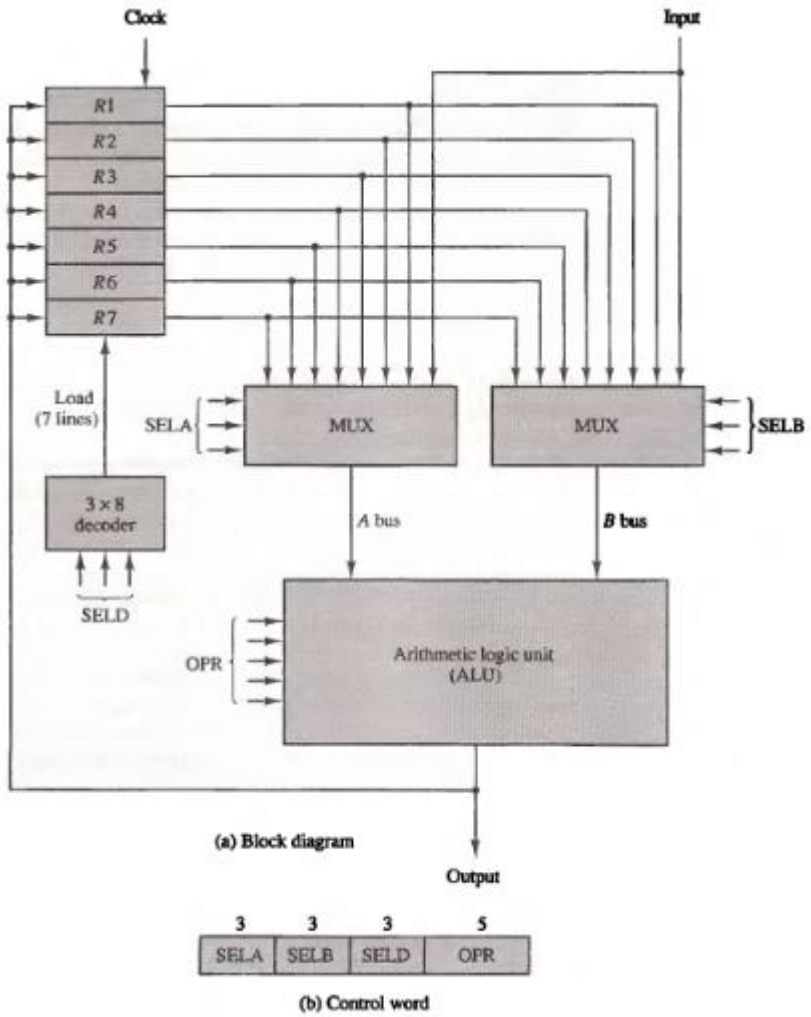


Figure 26 Register set with common ALU.

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, first-out (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:

$$ADD \;\; X$$

Where X is the address of the operand. The ADD instruction in this case results in the operation:

$$AC \leftarrow AC + M[X]$$

An example of a general register type of organization the instruction for an arithmetic addition may be written in an assembly language as:

$$ADD \;\; R1, R2, R3$$

to denote the operation:

$$R1 \leftarrow R2 + R3$$

The following is a program to evaluate $X = (A + b) * (C + D)$:
with three-address instruction formats as:

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

Two-address instructions formats as:

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

One-address instructions use an implied accumulator (AC) register for all data manipulation as:

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

Addressing Modes

Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction.

PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. An example of an instruction format with a distinct addressing mode field is shown in Fig(27).

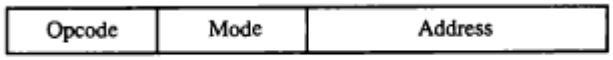


Figure 27 Instruction format with mode field.

Immediate Mode: In this mode the operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field.

Register Mode: In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction.

Register Indirect Mode: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory.

Auto-increment or Auto-decrement Mode: This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.

The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode.

Direct Address Mode: In this mode the effective address is equal to the address part of the instruction.

Indirect Address Mode: In this mode the address field of the instruction gives the address where the effective address is stored in memory.

The effective address in these modes is obtained from the following computation:

effective address = address part of instruction + content of CPU register

Relative Address Mode: In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.

Indexed Addressing Mode: In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.

Base Register Addressing Mode: In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.

Numerical Example:

1- The two-word instruction at address 200 and 201 is a "load to AC" instruction with an address field equal to 500.
2- The first word of the instruction specifies the operation code and mode, and the second word specifies the address part.
3- PC has the value 200 for fetching this instruction.
4- The content of processor register Rl is 400, and the content of an index register XR is 100.

AC receives the operand after the instruction is executed. The Fig(28) lists a few pertinent addresses and shows the memory content at each of these addresses.

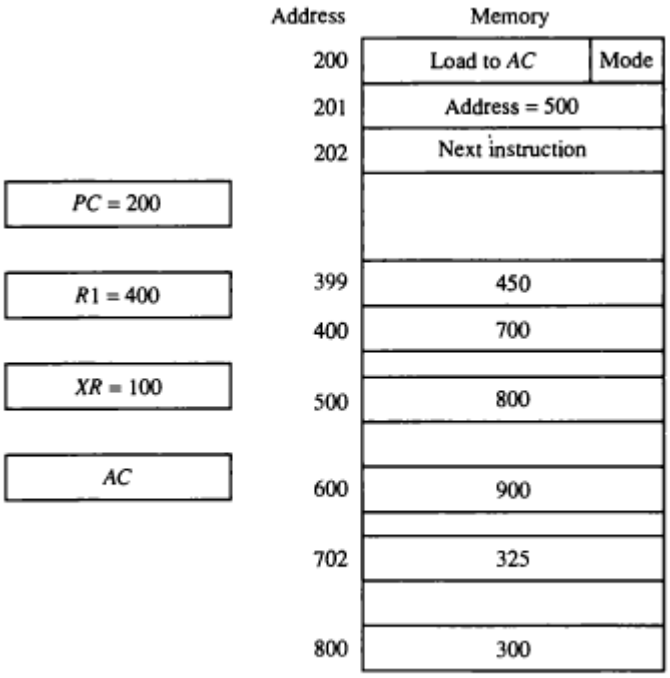


Figure 28 Numerical example for addressing modes.

Answer:

1- *In the direct address mode* the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800.

2- *In the immediate mode* the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201)

3- *In the indirect mode* the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300.

4- *In the relative mode* the effective address is 500 + 202 = 702 and the operand is 325. (Note that the value in PC after the fetch phase and during the execute phase is 202)

5- *In the index mode* the effective address is XR + 500 = 100 + 500 = 600 and the operand is 900.

6- *In the register mode* the operand is in Rl and 400 is loaded into AC. (There is no effective address in this case)

7- *In the register indirect mode* the effective address is 400, equal to the content of Rl and the operand loaded into AC is 700.

8- *The auto-increment mode* is the same as the register indirect mode except that Rl is incremented to 401 after the execution of the instruction.

9- *The auto-decrement mode* decrements Rl to 399 prior to the execution of the instruction. The operand loaded into AC is now 450.

Table (16) lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

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.

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:

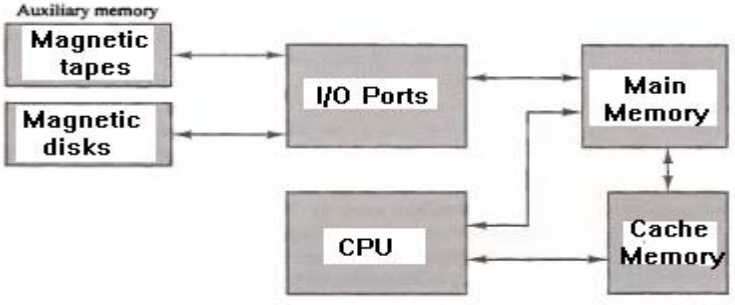


Figure 29 Memory hierarchy in a computer system.

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):

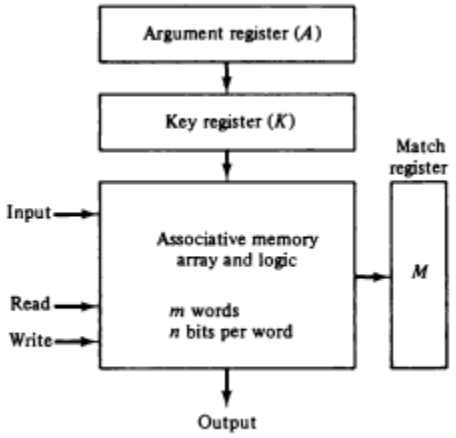


Figure 30 Block diagram of associative memory.

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 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.

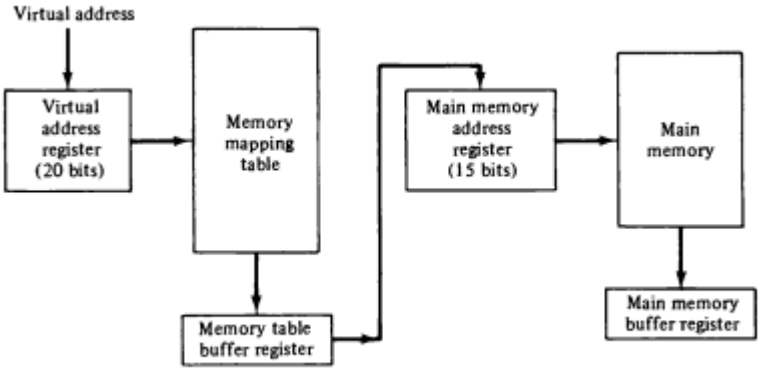


Figure 31 Memory table for mapping a virtual address.

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.

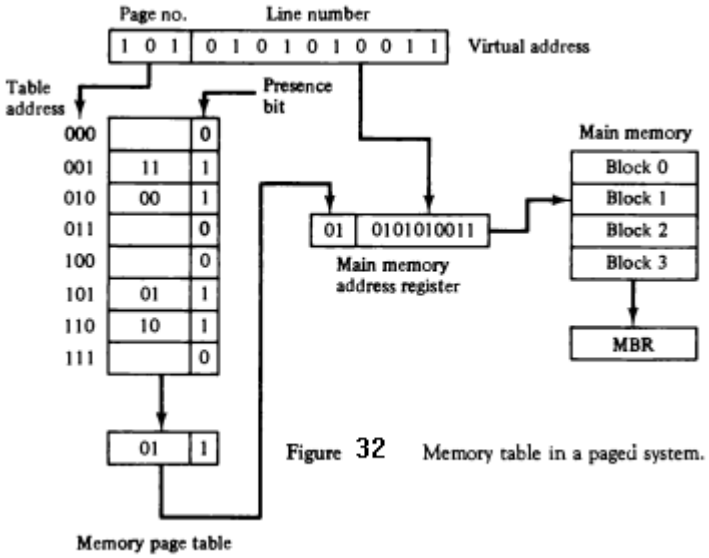


Figure 32 Memory table in a paged system.

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).

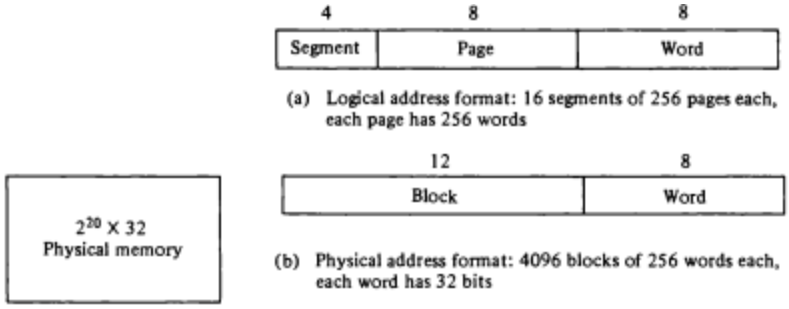
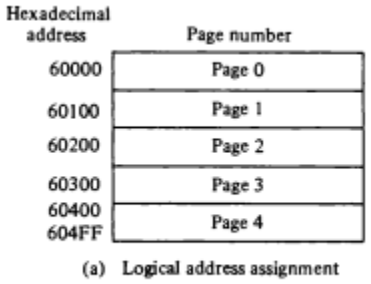


Figure 33 An example of logical and physical addresses.

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).



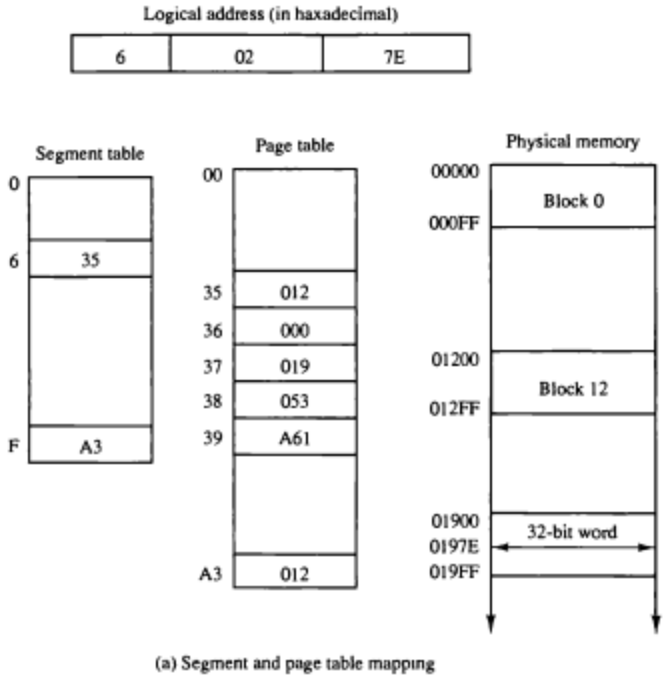
(a) Logical address assignment

Segment	Page	Block
6	00	012
6	01	000
6	02	019
6	03	053
6	04	A61

(b) Segment-page versus memory block assignment

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.

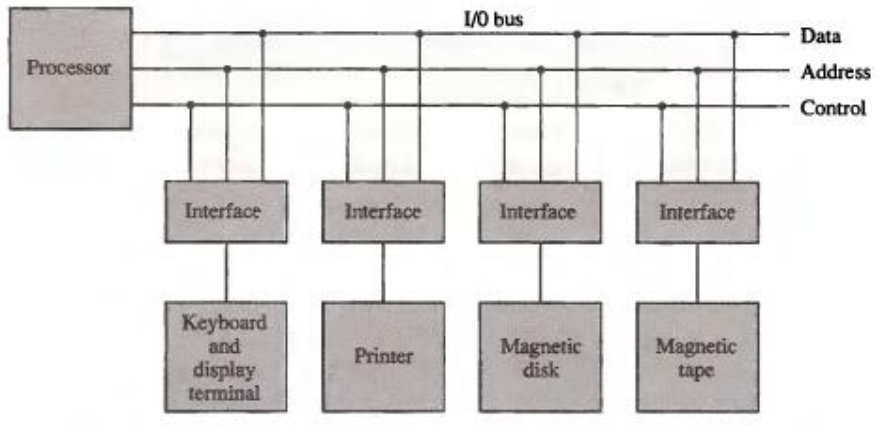
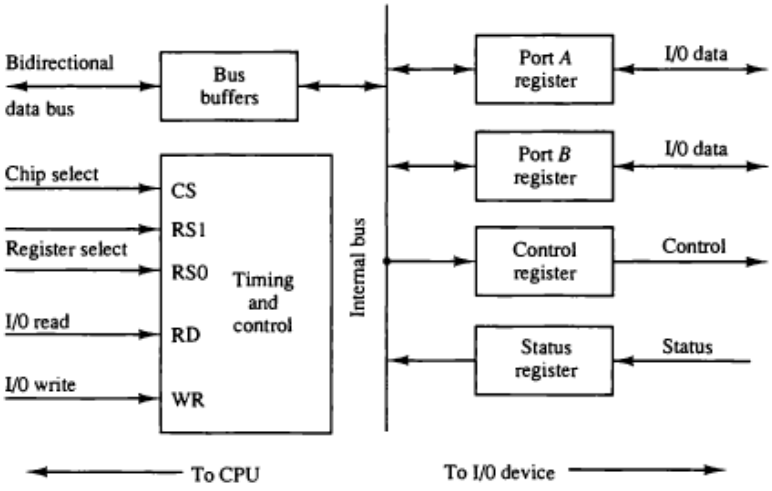


Figure 36 Connection of I/O bus to input-output devices.

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.

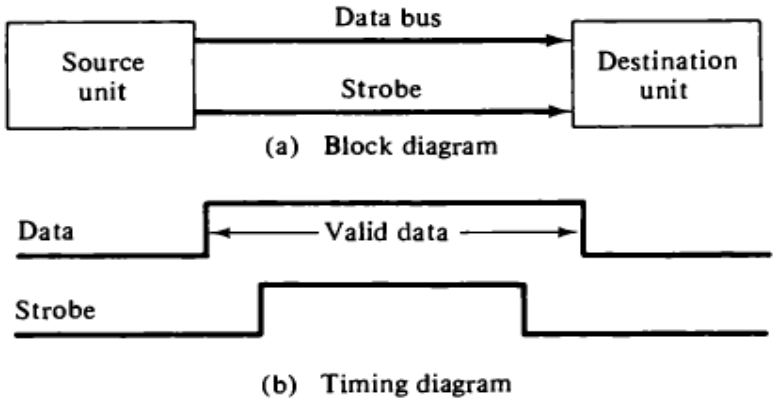


Figure 38 Source-initiated strobe for data transfer.

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

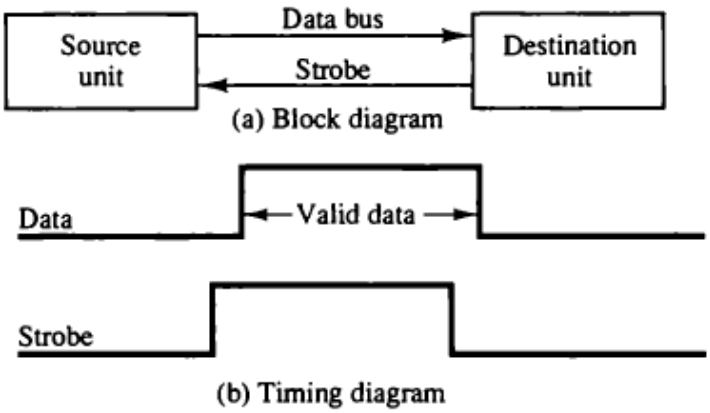


Figure 39 Destination-initiated strobe for data transfer.

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.

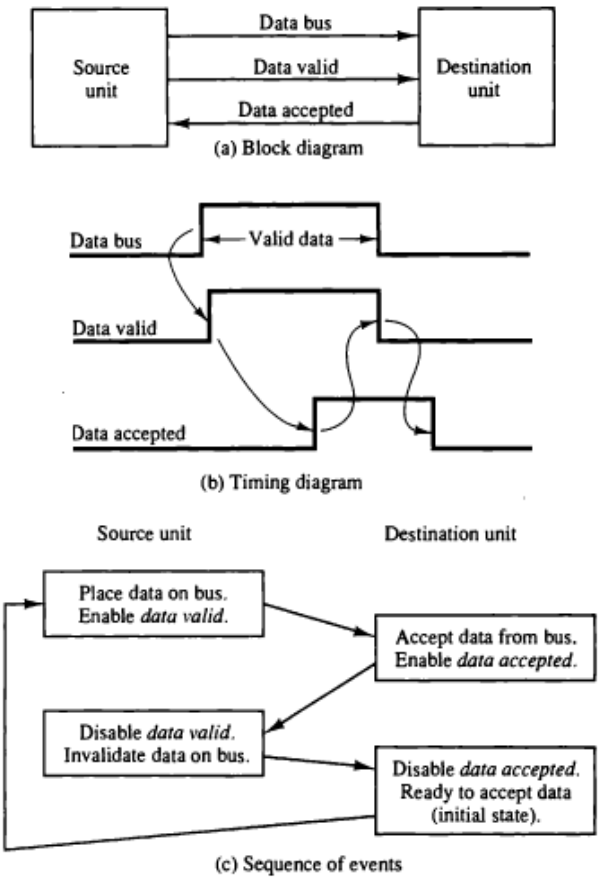


Figure 40 Source-initiated transfer using handshaking.

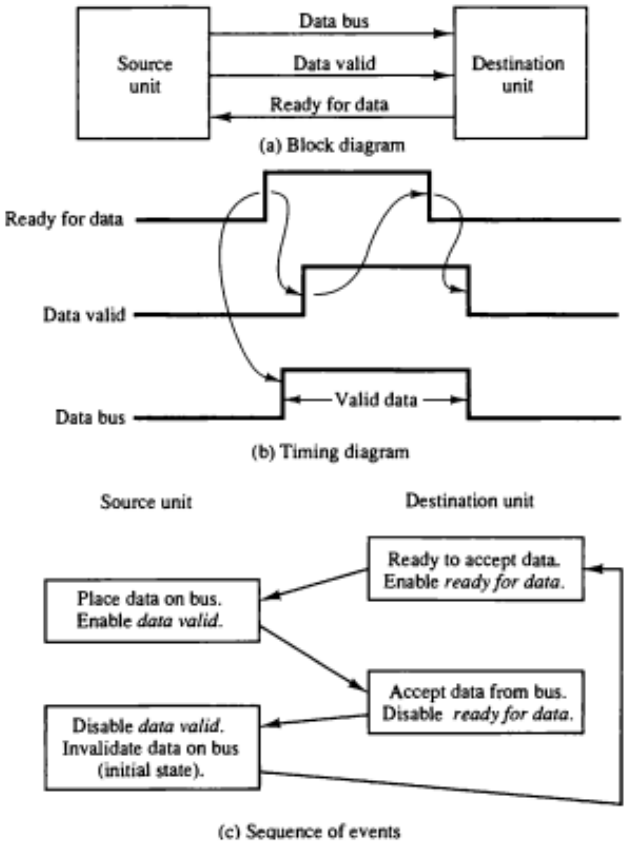


Figure 41 Destination-initiated transfer using handshaking.

An example of serial transmission format is shown in Fig. 42.

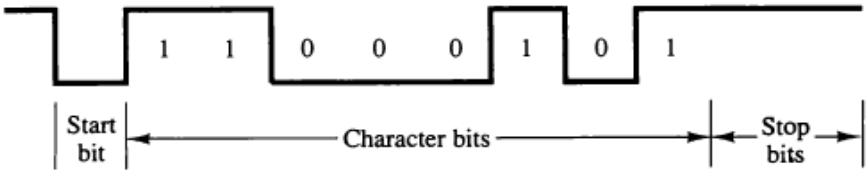


Figure 42 Asynchronous serial transmission.

As an illustration, consider the serial transmission of a terminal whose transfer rate is 10 characters per second. Each transmitted character consists of a start bit, eight information bits, and two stop bits, for a total of 11 bits. Ten characters per second means that each character takes 0.1 s for transfer. Since there are 11 bits to be transmitted, it follows that the bit time is 9.09 ms. The baud rate is defined as the rate at which serial information is transmitted and is equivalent to the data transfer in bits per second. Ten characters per second with an 11-bit format have a transfer rate of 110 baud. Integrated circuits are available which are specifically designed to provide the interface between computer and similar interactive terminals. Such a circuit is called an asynchronous communication interface or a universal asynchronous receiver-transmitter (UART).

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.

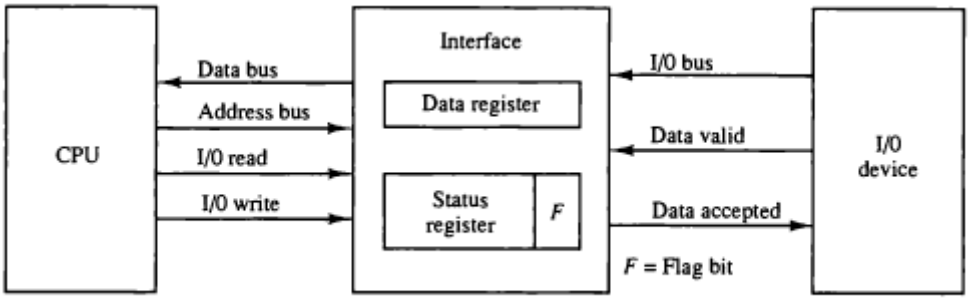


Figure 43 Data transfer from I/O device to CPU.

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.

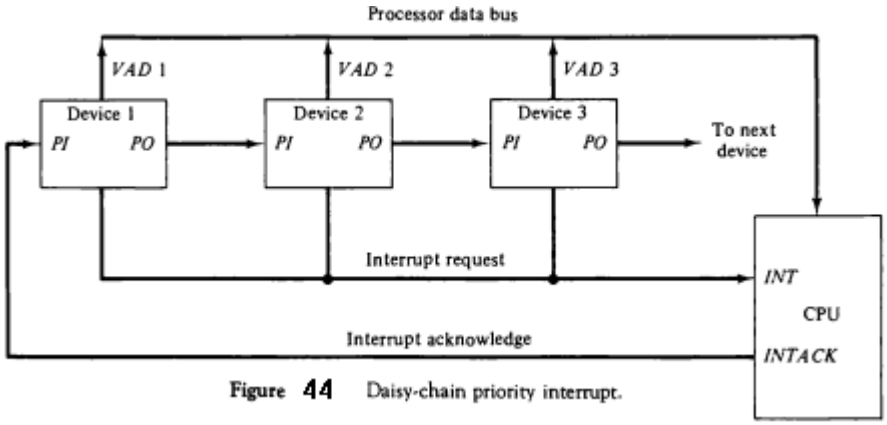


Figure 44 Daisy-chain priority interrupt.

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.

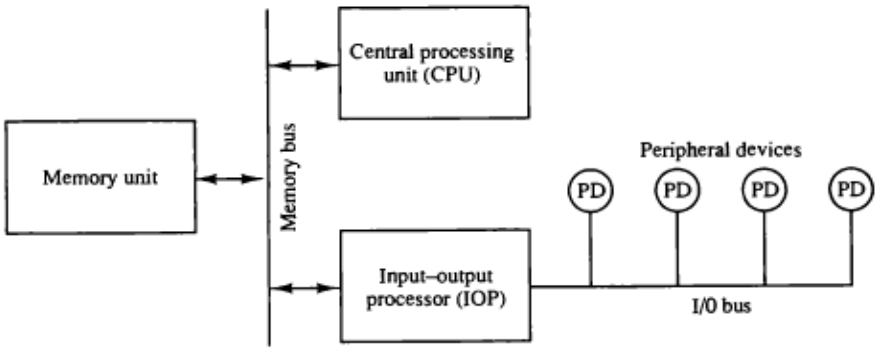


Figure 45 Block diagram of a computer with I/O processor.

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 four-segment pipeline is illustrated in Fig. 46. The operands pass through all four segments in a fixed sequence.

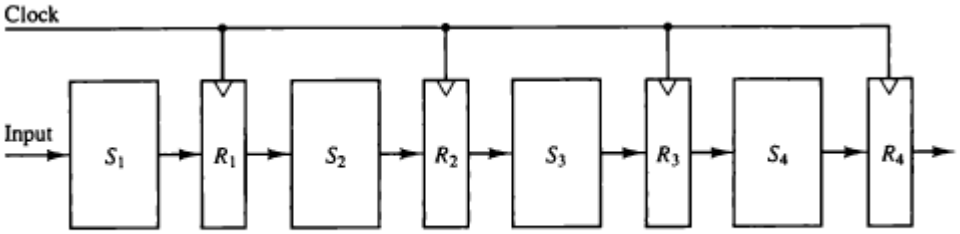


Figure 46 Four-segment pipeline.

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

Segment:	1	2	3	4	5	6	7	8	9	Clock cycles
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	

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 = 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.

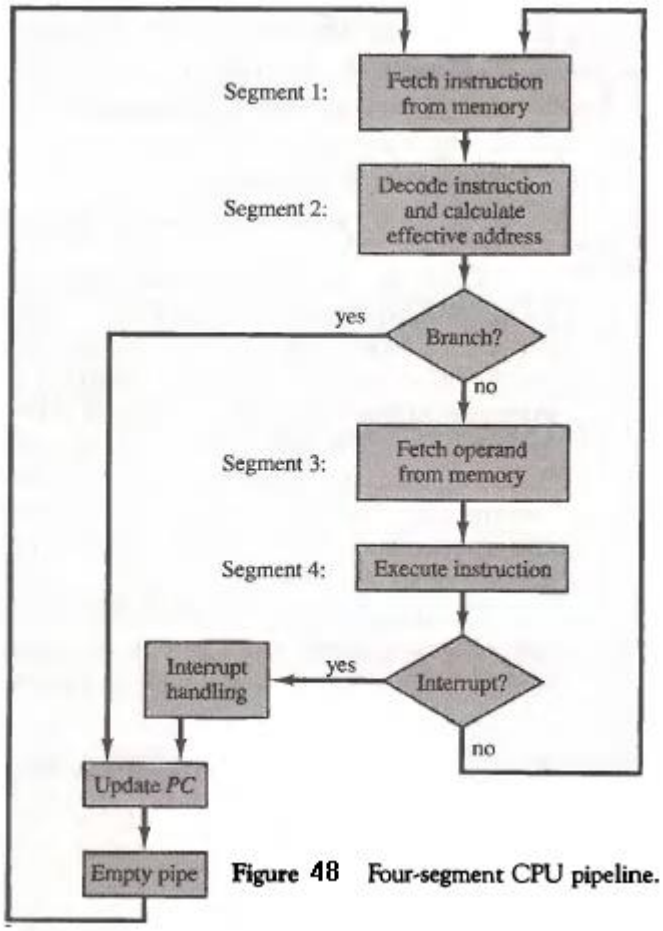


Figure 48 Four-segment CPU pipeline.

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.

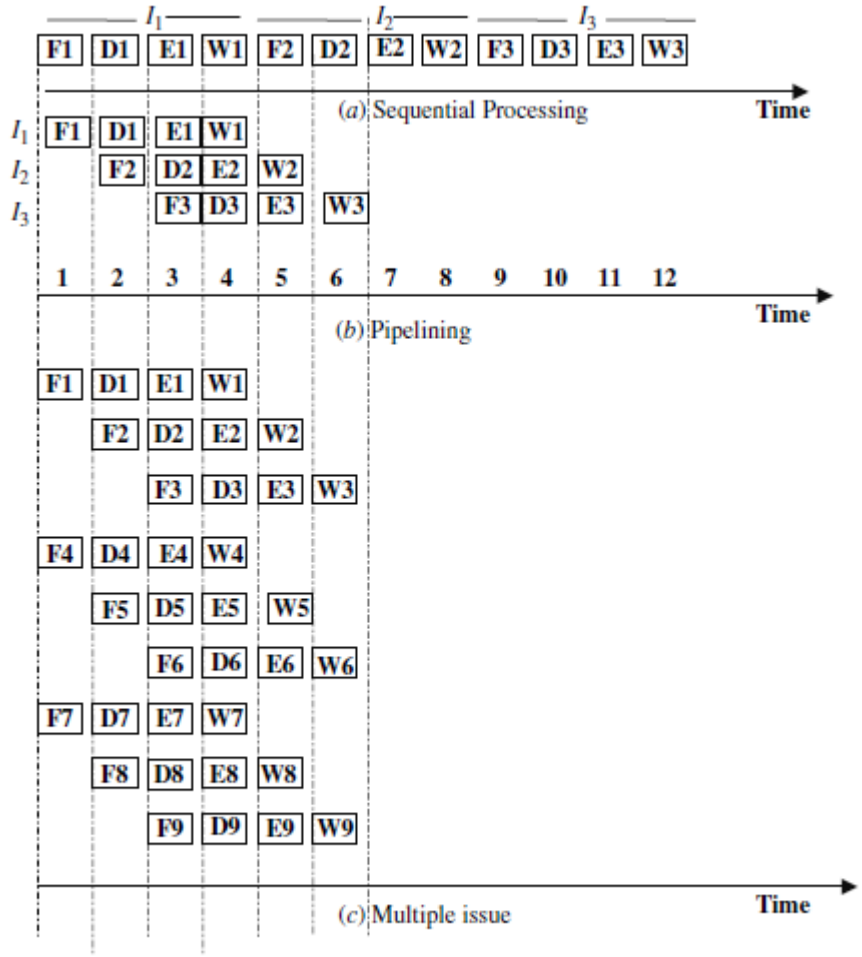


Figure 49 Multiple issue versus pipelining versus sequential processing

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 sub-operations 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:

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

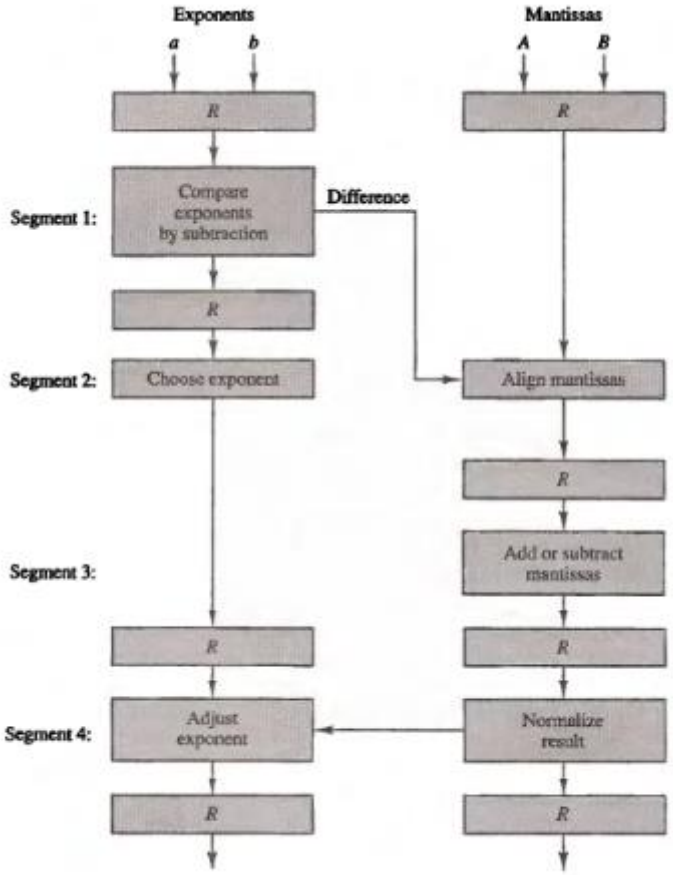


Figure 49 Pipeline for floating-point addition and subtraction.

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.

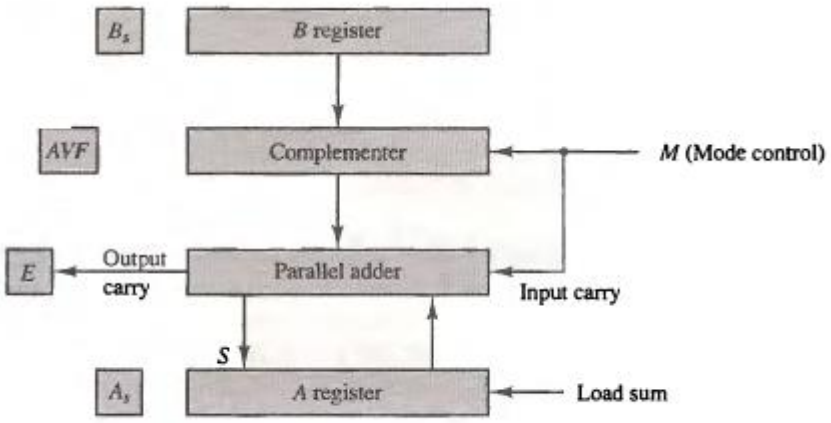


Figure 51 Hardware for signed-magnitude addition and subtraction.

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 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.

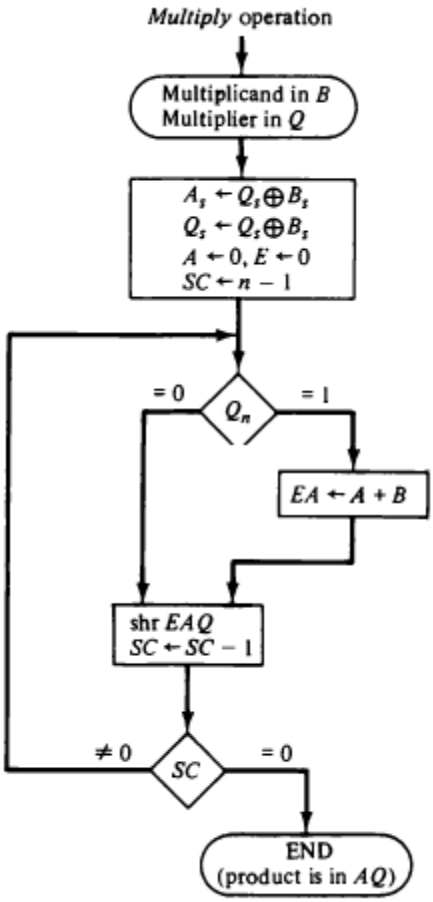


Figure 52 Flowchart for multiply operation.

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

	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.

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 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.

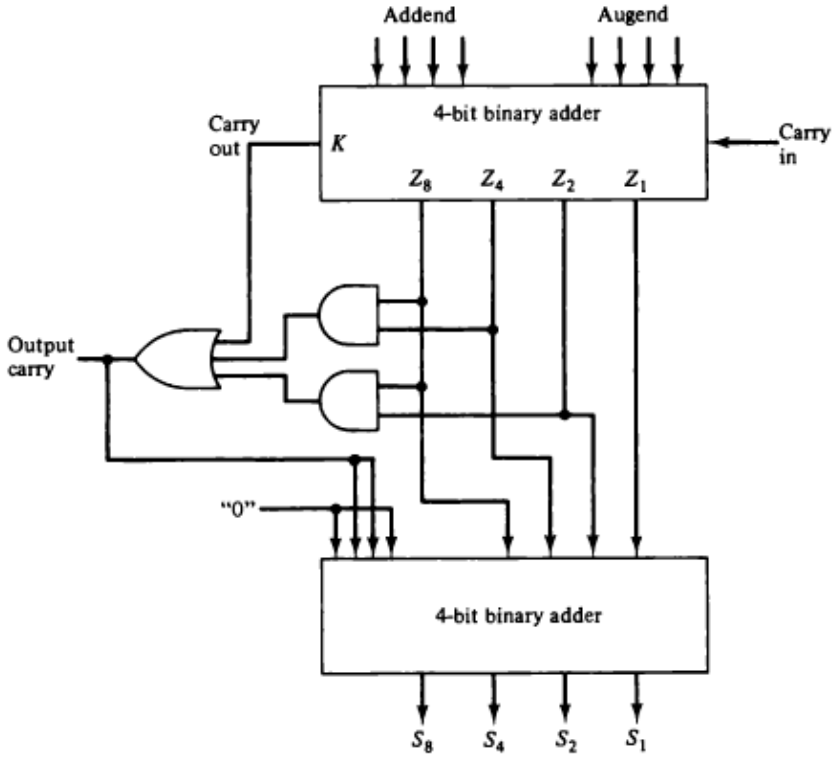


Figure 54 Block diagram of BCD adder.

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.*

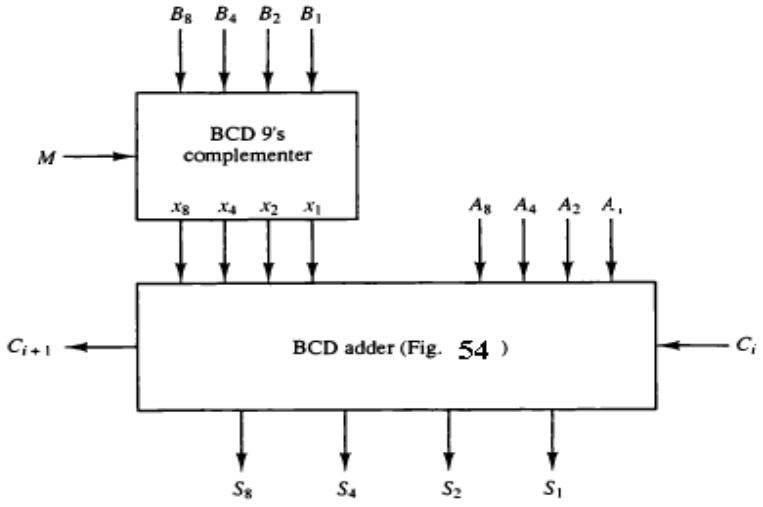


Figure 55 One stage of a decimal arithmetic unit.

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.

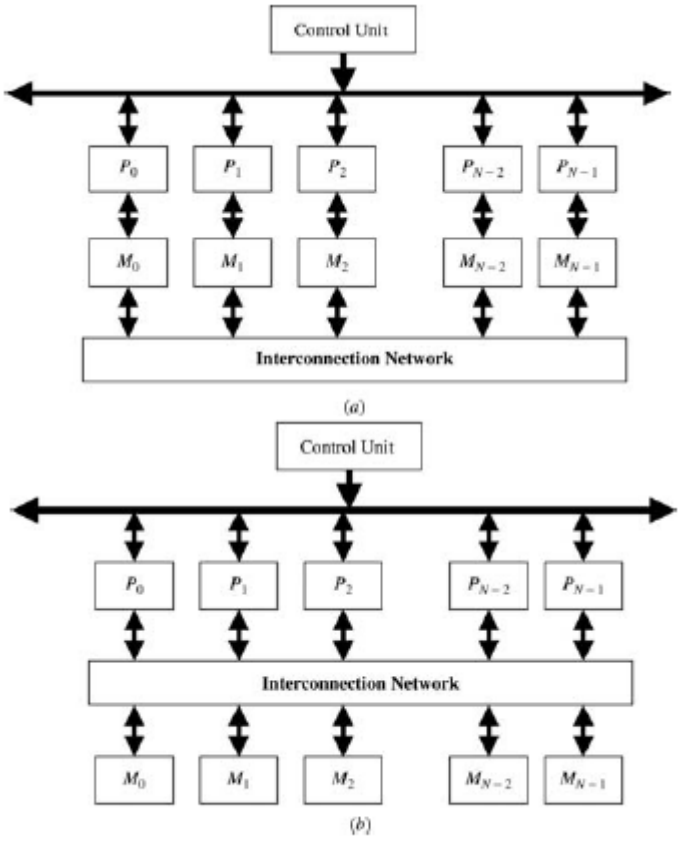
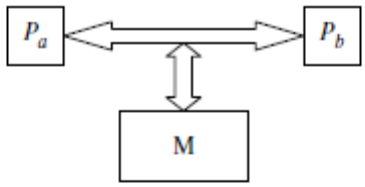


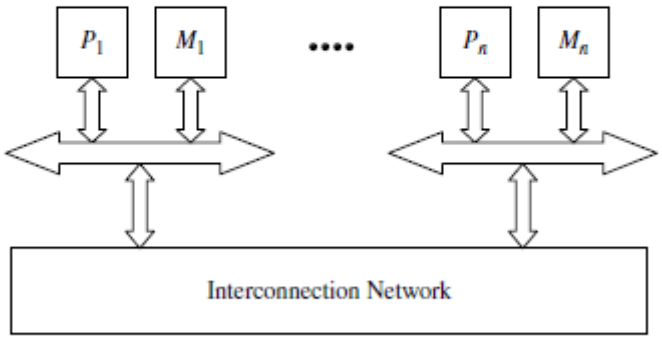
Figure 56 Two SIMD schemes. (*a*) SIMD scheme 1, (*b*) SIMD scheme 2

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.

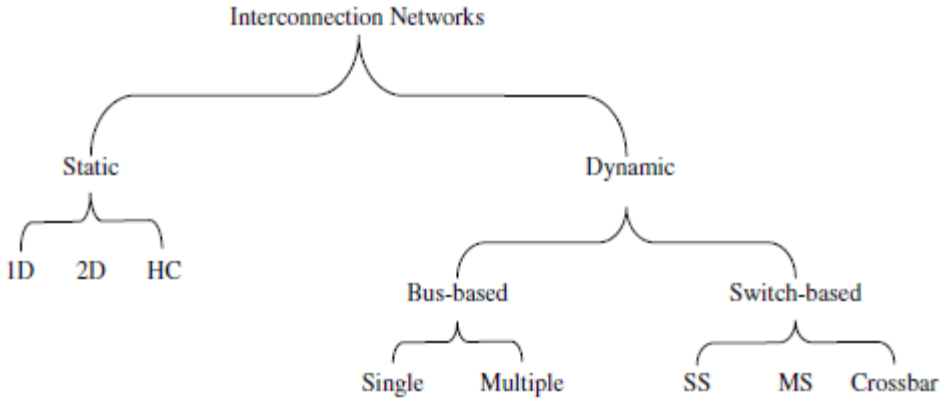


Figure 58 A topology-based taxonomy for interconnection networks