(23CS304) COMPUTER ORGANIZATION & ARCHITECTURE

UNIT-I

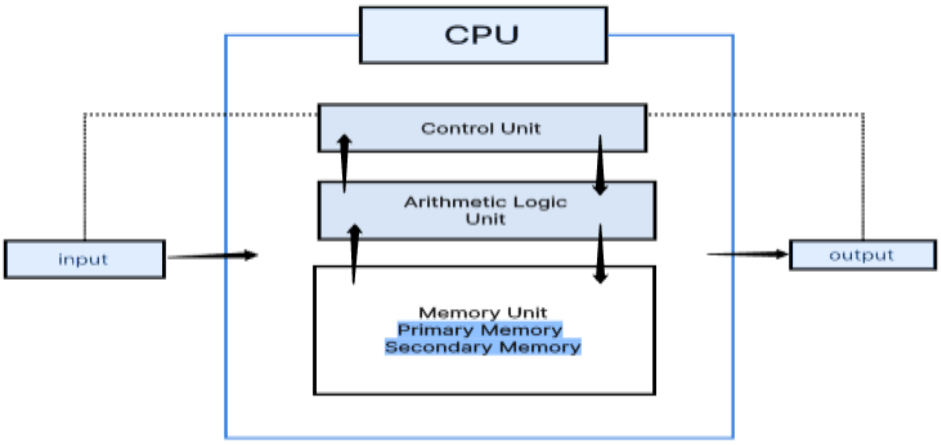
DIGITAL COMPUTERS: Introduction, Block diagram of Digital Computer, Definition of Computer Organization, Computer Design and Computer Architecture.
REGISTER TRANSFER LANGUAGE AND MICROOPERATIONS: Register Transfer language. Register Transfer Bus and memory transfers, Arithmetic Micro operations, logic micro operations, shift micro operations, Arithmetic logic shift unit.
BASIC COMPUTER ORGANIZATION AND DESIGN: Instruction codes, Computer Register Computer instructions, Timing and control, Instruction cycle, Memory – Reference Instructions, Input – Output and Interrupt.

DIGITAL COMPUTERS

A Digital computer can be considered as a digital system that performs various computational tasks.

The first electronic digital computer was developed in the late 1940s and was used primarily for numerical computations. By convention, the digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit. A computer system is subdivided into two functional entities: Hardware and Software.
The hardware consists of all the electronic components and electromechanical devices that comprise the physical entity of the device. The software of the computer consists of the instructions and data that the computer manipulates to perform various data-processing tasks.

Block diagram of Digital Computer:



- The Central Processing Unit (CPU) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and a control circuit for fetching and executing instructions.
- The memory unit of a digital computer contains storage for instructions and data.
- The Random Access Memory (RAM) for real-time processing of the data.
- The Input-Output devices for generating inputs from the user and displaying the final results to the user.
- The Input-Output devices connected to the computer include the keyboard, mouse, terminals, magnetic disk drives, and other communication devices.

Historical Perspective:

Development of technologies used to fabricate the processors, memories and I/O units of the computers has been divided into various generations as given below:

First generation: 1946 to 1955: Computers of this generation used Vacuum Tubes. The computes were built using stored program concept. Ex: ENIAC, EDSAC, IBM 701. Computers of this age typically used about ten thousand vacuum tubes. They were bulky in size had slow operating speed, short life time and limited programming facilities.

Second generation: 1955 to 1965: Computers of this generation used the germanium transistors as the active switching electronic device. Ex: IBM 7000, B5000, IBM 1401. Comparatively smaller in size About ten times faster operating speed as compared to first generation vacuum tube based computers. Consumed less power, had fairly good reliability. Availability of large memory was an added advantage.

Third generation: 1965 to 1975: The computers of this generation used the Integrated Circuits as the active electronic components. Ex: IBM system 360, PDP minicomputer etc. They were still smaller in size. They had powerful CPUs with the capacity of executing 1 million instructions per second (MIPS). Used to consume very less power consumption.

Fourth generation: 1976 to 1990: The computers of this generation used the LSI chips like microprocessor as their active electronic element. HCL horizen III, and WIPRO'S Uni plus+ HCL's Busy bee PC etc. They used high speed microprocessor as CPU. They were more user friendly and highly reliable systems. They had large storage capacity disk memories.

Beyond Fourth Generation: 1990 onwards: Specialized and dedicated VLSI chips are used to control specific functions of these computers. Modern Desktop PC's, Laptops or Notebook Computers.

BASIC COMPUTER ORGANIZATION: Most of the computer systems found in automobiles and consumer appliances to personal computers and main frames have some basic organization. The basic computer organization has three main components:

- CPU
- Memory subsystem
- I/O subsystem.

The generic organization of these components is shown in the figure below.

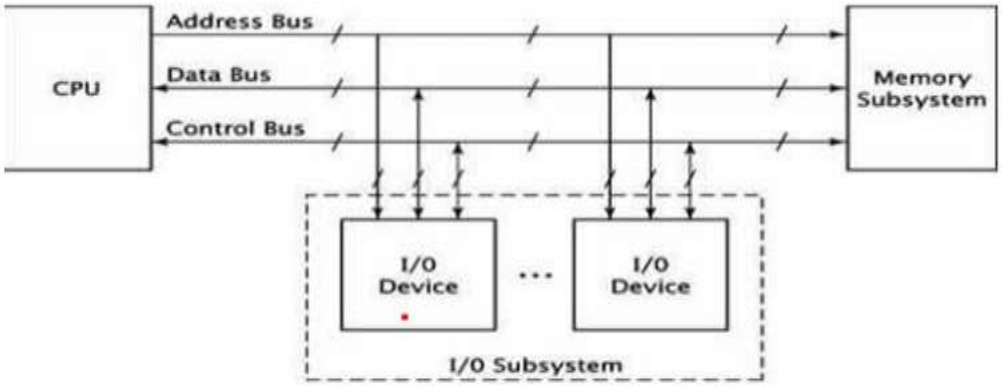


Fig 1.1 Generic computer Organization

Computer organization: Computer organization is the knowing, what the functional components of a computer are, how they work and how their performance is measured and optimized. Computer
Organization refers to the level of abstraction above the digital logic level, but below the operating system level.

Computer design: 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.

Two basic types of computer architecture are:

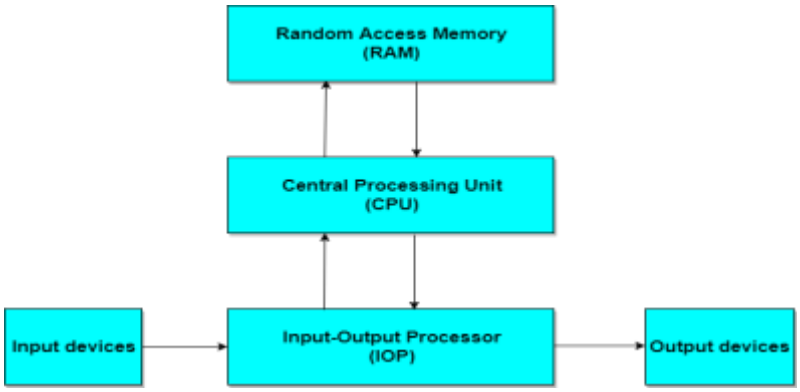
1. **Von Neumann architecture**
2. **Harvard architecture**

1. Von Neumann architecture

The **von Neumann architecture** describes a general framework, or structure, that a computer's hardware, programming, and data should follow. Although other structures for computing have been devised and implemented, the vast majority of computers in use today operate according to the von Neumann architecture.

von Neumann envisioned the structure of a computer system as being composed of the following components:

1. **ALU:** The **Arithmetic-Logic unit** that performs the computer's computational and logical functions.
2. **RAM:** Memory; more specifically, the computer's main, or fast, memory, also known as **Random Access Memory (RAM)**.
3. **Control Unit**: This is a component that directs other components of the computer to perform certain actions, such as directing the fetching of data or instructions from memory to be processed by the ALU; and
4. **Man-machine interfaces;** i.e. input and output devices, such as keyboard for input and display monitor for output.



An example of computer architecture base on the von Neumann architecture is the desktop **personal computer**.

2. Harvard architecture

The **Harvard architecture** uses physically separate **storage** and **signal pathways** for their instructions and data. The term originated from the **Harvard Mark I** and the data in relay latches (23- digits wide).

In a computer with Harvard architecture, the CPU can read both an instruction and data from memory at the same time, leading to double the memory bandwidth.

Microcontroller (single-chip microcomputer)-based computer systems and **DSP**(Digital Signal Processor)-based computer systems are examples of Harvard architecture.

BASIC LOGIC GATES WITH TRUTH TABLES

Nowadays, computers have become an integral part of life as they perform many tasks and operations in quite a short span of time. One of the most important functions of the CPU in a computer is to perform logical operations by utilizing hardware like Integrated Circuits software technologies & electronic circuits,. But, how this hardware and software perform such operations is a mysterious puzzle. In order to have a better understanding of such a complex issue, we must have to acquaint ourselves with the term Boolean Logic, developed by George Boole. For a simple operation, computers utilize binary digits rather than digital digits. All the operations are carried out by the Basic Logic gates. This article discusses an overview of what are **basic logic gates** in digital electronics and their working.

Basic Logic Gates:

A logic gate is a basic building block of a digital circuit that has two inputs and one output. The relationship between the i/p and the o/p is based on a certain logic. These gates are implemented using electronic switches like transistors, diodes. But, in practice, basic logic gates are built using (complementary metal-oxide-semiconductor) CMOS technology, Field-effect transistor FETS, and MOSFET(Metal Oxide Semiconductor FET)s. Logic gates are used in microprocessors, microcontrollers, embedded system applications, and in electronic and electrical project circuits. The basic logic gates are categorized into seven: AND, OR, XOR, NAND, NOR, XNOR, and NOT. These logic gates with their logic gate symbols and truth tables are explained below.

Basic Logic Gates Operation

What are the 7 Basic Logic Gates?

The basic logic gates are classified into seven types: AND gate, OR gate, XOR gate, NAND gate, NOR gate, XNOR gate, and NOT gate. The truth table is used to show the logic gate function. All the logic gates have two inputs except the NOT gate, which has only one input.

When drawing a truth table, the binary values 0 and 1 are used. Every possible combination depends on the number of inputs. If you don't know about the logic gates and their truth tables and need guidance on them, please go through the following infographic that gives an overview of logic gates with their symbols and truth tables.

Why we use Basic Logic Gates?

The basic logic gates are used to perform fundamental logical functions. These are the basic building blocks in the digital ICs (integrated circuits). Most of the logic gates use two binary inputs and generates a single output like 1 or 0. In some electronic circuits, few logic gates are used whereas in some other circuits, microprocessors include millions of logic gates.

The implementation of Logic gates can be done through diodes, transistors, relays, molecules, and optics otherwise different mechanical elements. Because of this reason, basic logic gates are used like electronic circuits.

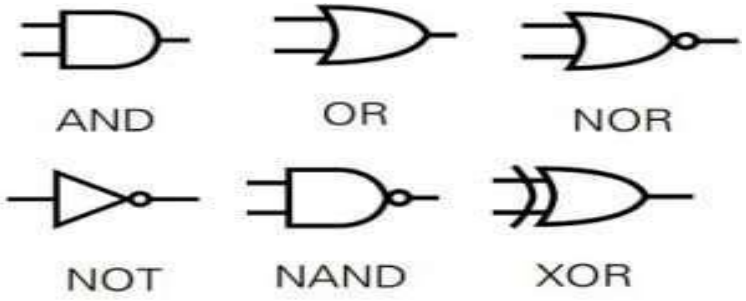
Binary & Decimal

Before talking about the truth tables of logic gates, it is essential to know the background of binary & decimal numbers. We all know the decimal numbers which we utilize in everyday calculations like 0 to 9. This kind of number system includes the base-10. In the same way, binary numbers like 0 and 1 can be utilized to signify decimal numbers wherever the base of the binary numbers is 2.

The significance of using binary numbers here is to signify the switching position otherwise voltage position of a digital component. Here 1 represents the High signal or high voltage whereas "0" specifies low voltage or low signal. Therefore, Boolean algebra was started. After that, each logic gate is discussed separately this contains the logic of the gate, truth table, and its typical symbol

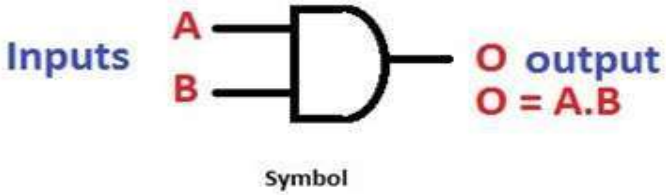
Types of Logic Gates:

The different types of logic gates and symbols with truth tables are discussed below.



AND Gate

The AND gate is a digital logic gate with 'n' i/ps one o/p, which performs logical conjunction based on the combinations of its inputs. The output of this gate is true only when all the inputs are true. When one or more inputs of the AND gate's i/ps are false, then only the output of the AND gate is false. The symbol and truth table of an AND gate with two inputs is shown below.



Symbol

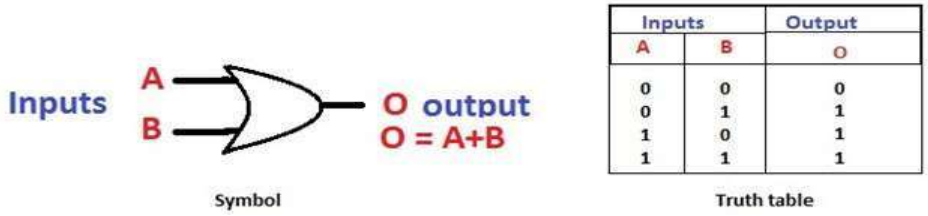
Inputs		Output
A	B	O
0	0	0
0	1	0
1	0	0
1	1	1

Truth table

AND Gate and its Truth Table

OR Gate

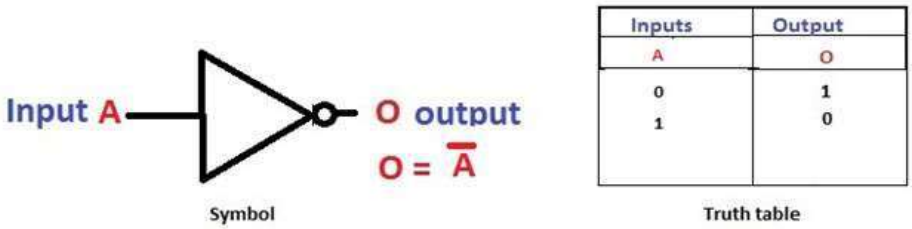
The OR gate is a digital logic gate with 'n' i/ps and one o/p, that performs logical conjunction based on the combinations of its inputs. The output of the OR gate is true only when one or more inputs are true. If all the i/ps of the gate are false, then only the output of the OR gate is false. The symbol and truth table of an OR gate with two inputs is shown below.



OR Gate and its Truth Table

NOT Gate

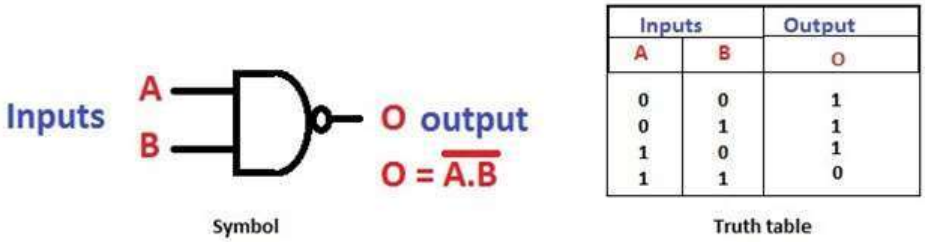
The NOT gate is a digital logic gate with one input and one output that operates an inverter operation of the input. The output of the NOT gate is the reverse of the input. When the input of the NOT gate is true then the output will be false and vice versa. The symbol and truth table of a NOT gate with one input is shown below. By using this gate, we can implement NOR and NAND gates



NOT Gate and Its Truth Table

NAND Gate

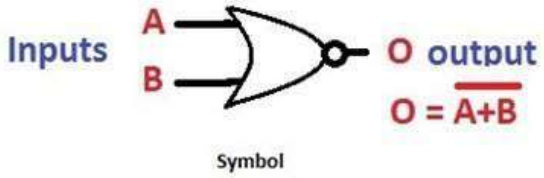
The NAND gate is a digital logic gate with ‘n’ i/ps and one o/p, that performs the operation of the AND gate followed by the operation of the NOT gate.NAND gate is designed by combining the AND and NOT gates. If the input of the NAND gate high, then the output of the gate will be low.The symbol and truth table of the NAND gate with two inputs is shown below.



NAND Gate and its Truth Table

NOR Gate

The NOR gate is a digital logic gate with n inputs and one output, that performs the operation of the OR gate followed by the NOT gate. NOR gate is designed by combining the OR and NOT gate. When any one of the i/ps of the NOR gate is true, then the output of the NOR gate will be false. The symbol and truth table of the NOR gate with the truth table is shown below.



Symbol

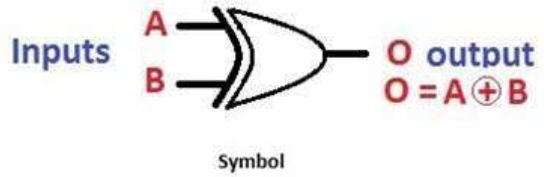
Inputs		Output
A	B	O
0	0	1
0	1	0
1	0	0
1	1	0

Truth table

NOR Gate and Its Truth Table

Exclusive-OR Gate

The Exclusive-OR gate is a digital logic gate with two inputs and one output. The short form of this gate is Ex-OR. It performs based on the operation of the OR gate. . If any one of the inputs of this gate is high, then the output of the EX-OR gate will be high. The symbol and truth table of the EX-OR are shown below.



Symbol

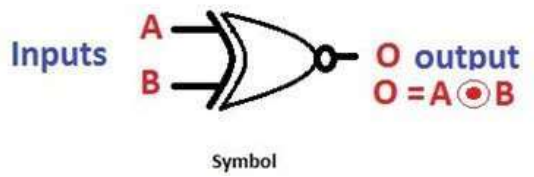
Inputs		Output
A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

Truth table

EX-OR gate and Its Truth Table

Exclusive-NOR Gate

The Exclusive-NOR gate is a digital logic gate with two inputs and one output. The short form of this gate is Ex-NOR. It performs based on the operation of the NOR gate. When both the inputs of this gate are high, then the output of the EX-NOR gate will be high. But, if any one of the inputs is high (but not both), then the output will be low. The symbol and truth table of the EX-NOR are shown below.



Symbol

Inputs		Output
A	B	O
0	0	1
0	1	0
1	0	0
1	1	1

Truth table

EX-NOR Gate and Its Truth Table

What is the Easiest Way to Learn Logic Gates?

The easiest way to learn the function of basic logic gates is explained below.

- ✓ For AND Gate – If both the inputs are high then the output is also high
- ✓ For OR Gate – If a minimum of one input is high then the output is High
- ✓ For XOR Gate – If the minimum one input is high then only the output is high
- ✓ NAND Gate – If the minimum one input is low then the output is high
- ✓ NOR Gate – If both the inputs are low then the output is high.

REGISTER TRANSFER LANGUAGE AND MICROOPERATIONS

Register Transfer Language

- Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic
- The modules are interconnected with common data and control paths to form a digital computer system
- The operations executed on data stored in registers are called *Micro operations*
- A microoperation is an elementary operation performed on the information stored in one or more registers
- Examples are shift, count, clear, and load
- Some of the digital components from before are registers that implement Micro operations
- The internal hardware organization of a digital computer is best defined by specifying
  - The set of registers it contains and their functions
  - The sequence of Micro operations performed on the binary information stored
  - The control that initiates the sequence of Micro operations
- Use symbols, rather than words, to specify the sequence of Micro operations
- The symbolic notation used is called a *register transfer language*
- A programming language is a procedure for writing symbols to specify a given computational process
- Define symbols for various types of Micro operations and describe associated hardware that can implement the Micro operations

Register Transfer

- Designate computer registers by capital letters to denote its function
- The register that holds an address for the memory unit is called MAR
- The program counter register is called PC
- IR is the instruction register and R1 is a processor register
- The individual flip-flops in an *n*-bit register are numbered in sequence from 0 to *n*-1

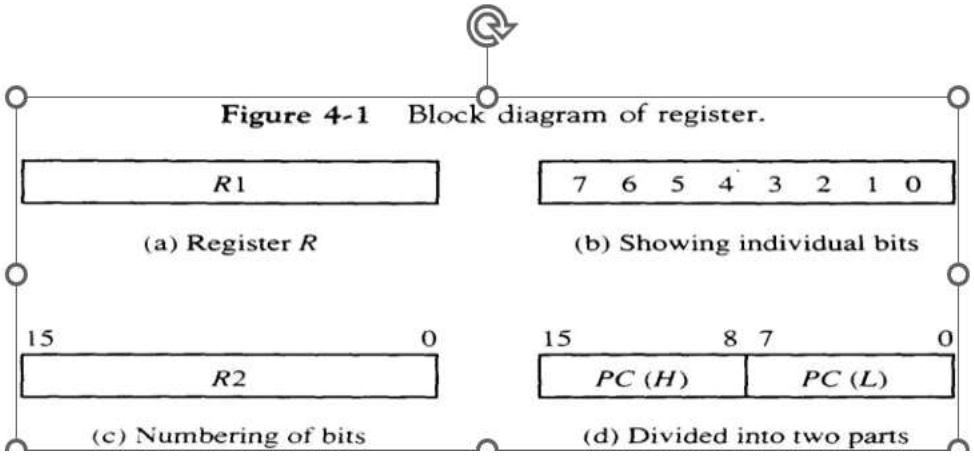


Figure 4-1 Block diagram of register.

- Designate information transfer from one register to another by R2 ← R1
- This statement implies that the hardware is available
  - The outputs of the source must have a path to the inputs of the destination
  - The destination register has a parallel load capability
- If the transfer is to occur only under a predetermined control condition, designate it by

  If (P = 1) *then* (R2 ← R1)

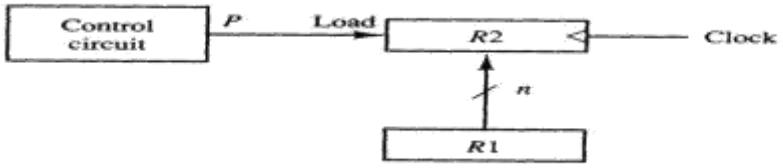
  or,

  P: R2 ← R1,

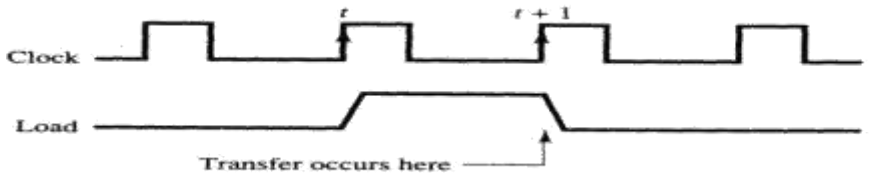
  where P is a control function that can be either 0 or 1

- Every statement written in register transfer notation implies the presence of the required hardware construction

Figure 4-2 Transfer from $R1$ to $R2$ when $P = 1$.



(a) Block diagram



(b) Timing diagram

- It is assumed that all transfers occur during a clock edge transition
- All Micro operations written on a single line are to be executed at the same time T: R2 ← R1, R1 ← R2

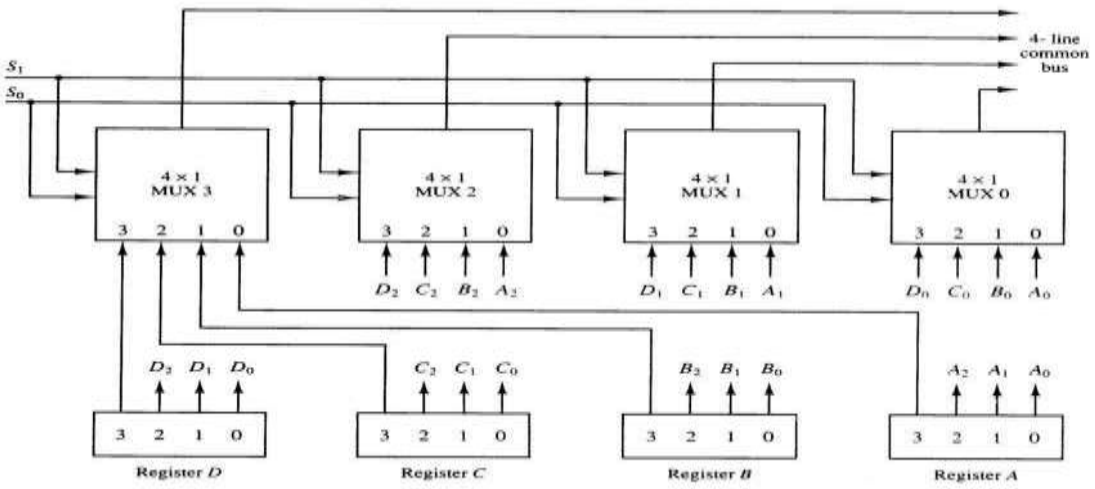
TABLE 4-1 Basic Symbols for Register Transfers

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/ Bus System for Four Registers

- Rather than connecting wires between all registers, a common bus is used
- A bus structure consists of a set of common lines, one for each bit of a register
- Control signals determine which register is selected by the bus during each transfer
- Multiplexers can be used to construct a common bus
- Multiplexers select the source register whose binary information is then placed on the bus
- The select lines are connected to the selection inputs of the multiplexers and choose the bits of one register

Figure 4-3 Bus system for four registers.



- In general, a bus system will multiplex *k* registers of *n* bits each to produce an *n*-line common bus
- This requires *n* multiplexers – one for each bit
- The size of each multiplexer must be *k* x 1
- The number of select lines required is *log k*
- To transfer information from the bus to a register, the bus lines are connected to the inputs of all destination registers and the corresponding load control line must be activated
- Rather than listing each step as

BUS ← C, R1 ← BUS, use R1 ← C, since the bus is implied

Three-State Gate Bus Buffer

- Instead of using multiplexers, *three-state gates* can be used to construct the bus system
- A three-state gate is a digital circuit that exhibits three states
- Two of the states are signals equivalent to logic 1 and 0
- The third state is a *high-impedance* state – this behaves like an open circuit, which means the output is disconnected and does not have a logic significance

Figure 4-4 Graphic symbols for three-state buffer.

- The three-state buffer gate has a normal input and a control input which determines the output state
- With control 1, the output equals the normal input
- With control 0, the gate goes to a high-impedance state
- This enables a large number of three-state gate outputs to be connected with wires to form a common bus line without endangering loading effects

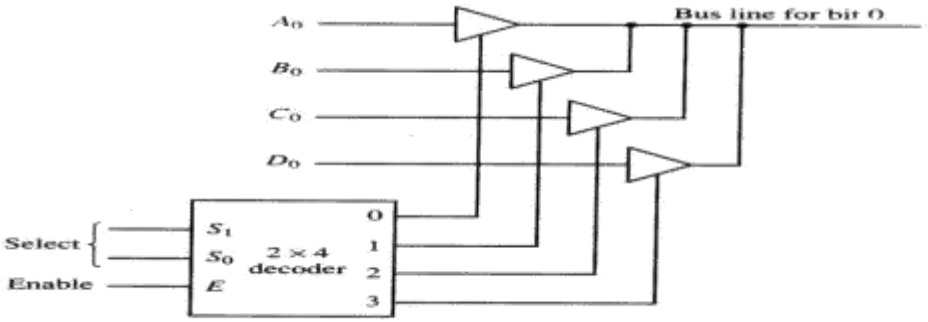


Figure 4-5 Bus line with three state-buffers.

- Decoders are used to ensure that no more than one control input is active at any given time
- This circuit can replace the multiplexer in Figure 4.3
- 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
- Only one decoder is necessary to select between the four registers

---

- Designate a memory word by the letter M
- It is necessary to specify the address of M when writing memory transfer operations
- Designate the address register by AR and the data register by DR
- The Read operation can be stated as:
  Read: DR ←M[AR]
- The write operation can be stated as:
  Write: M[AR] ← R1

MICRO-OPERATIONS:

- There are four categories of the most common Micro operations:
  - **Register transfer**: transfer binary information from one register to another
  - **Arithmetic**: perform arithmetic operations on numeric data stored in registers
  - **Logic**: perform bit manipulation operations on non-numeric data stored in registers
  - **Shift**: perform shift operations on data stored in registers

ARITHMETIC MICRO-OPERATIONS:

The basic arithmetic Micro operations are addition, subtraction, increment, decrement, and shift

- Example of addition: R3 ← R1 +R2
- Subtraction is most often implemented through complementation and addition
- Example of subtraction: R3 ← R1 + $\overline{R2}$ + 1 (strikethrough denotes bar on top – 1's complement of R2)
- Adding 1 to the 1's complement produces the 2's complement
- Adding the contents of R1 to the 2's complement of R2 is equivalent to subtracting
- Multiply and divide are not included as Micro operations
- A microoperation is one that can be executed by one clock pulse
- Multiply (divide) is implemented by a sequence of add and shift Micro operations (subtract and shift)

ADDITION:

- To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the addition
- A full-adder adds two bits and a previous carry
- A *binary adder* is a digital circuit that generates the arithmetic sum of two binary numbers of any length
- A binary added is constructed with full-adder circuits connected in cascade
- An *n*-bit binary adder requires *n* full-adders

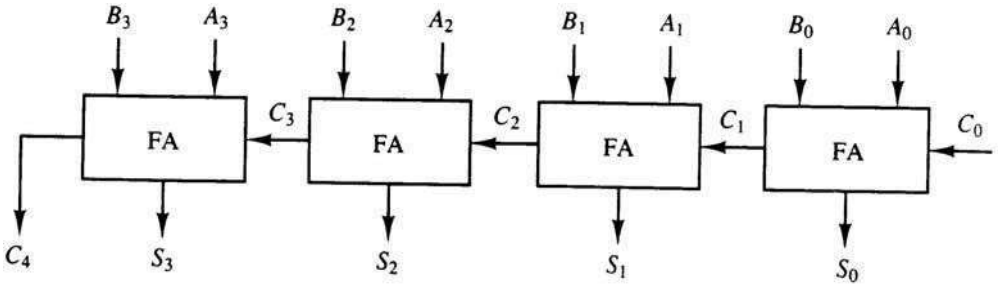


Figure 4-6 4-bit binary adder.

SUBTRACTION:

- The subtraction A-B can be carried out by the following steps
  - Take the 1's complement of B (invert each bit)
  - Get the 2's complement by adding 1
  - Add the result to A
- The addition and subtraction operations can be combined into one common circuit by including an XOR gate with each full-adder

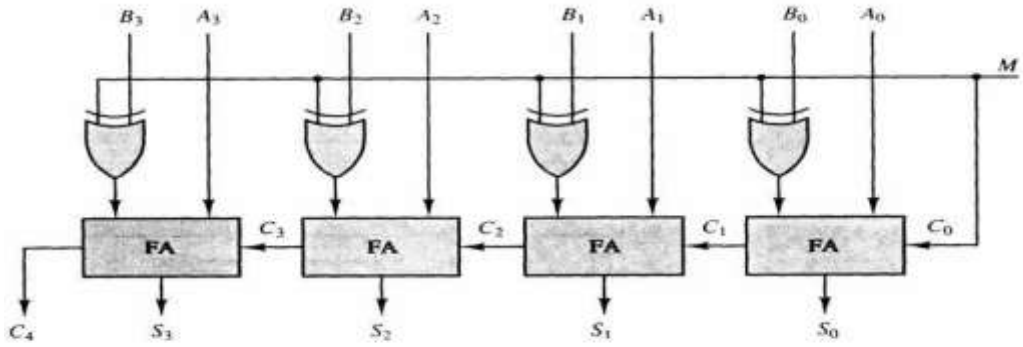


Figure 4-7 4-bit adder-subtractor.

INCREMENT:

- The increment microoperation adds one to a number in a register
- This can be implemented by using a binary counter – every time the count enable is active; the count is incremented by one
- If the increment is to be performed independent of a particular register, then use half-adders connected in cascade
- An *n*-bit binary incrementor requires *n* half-adders

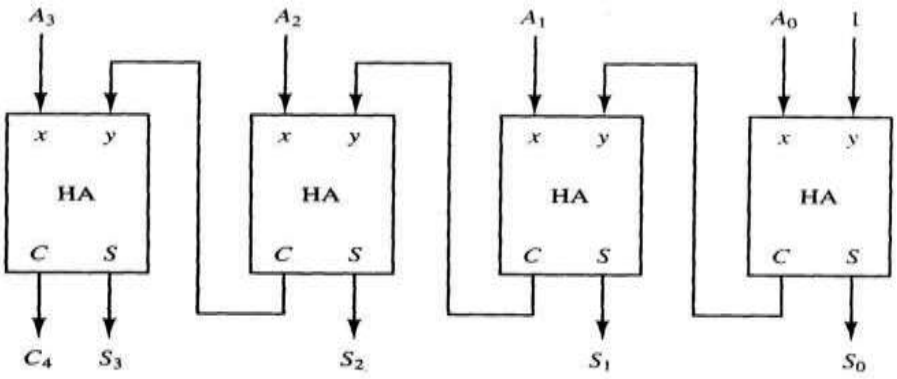


Figure 4-8 4-bit binary incrementer.

ARITHMETIC CIRCUIT:

- Each of the arithmetic Micro operations can be implemented in one composite arithmetic circuit
- The basic component is the parallel adder
- Multiplexers are used to choose between the different operations
- The output of the binary adder is calculated from the following sum:
  $D = A + Y + C_{in}$

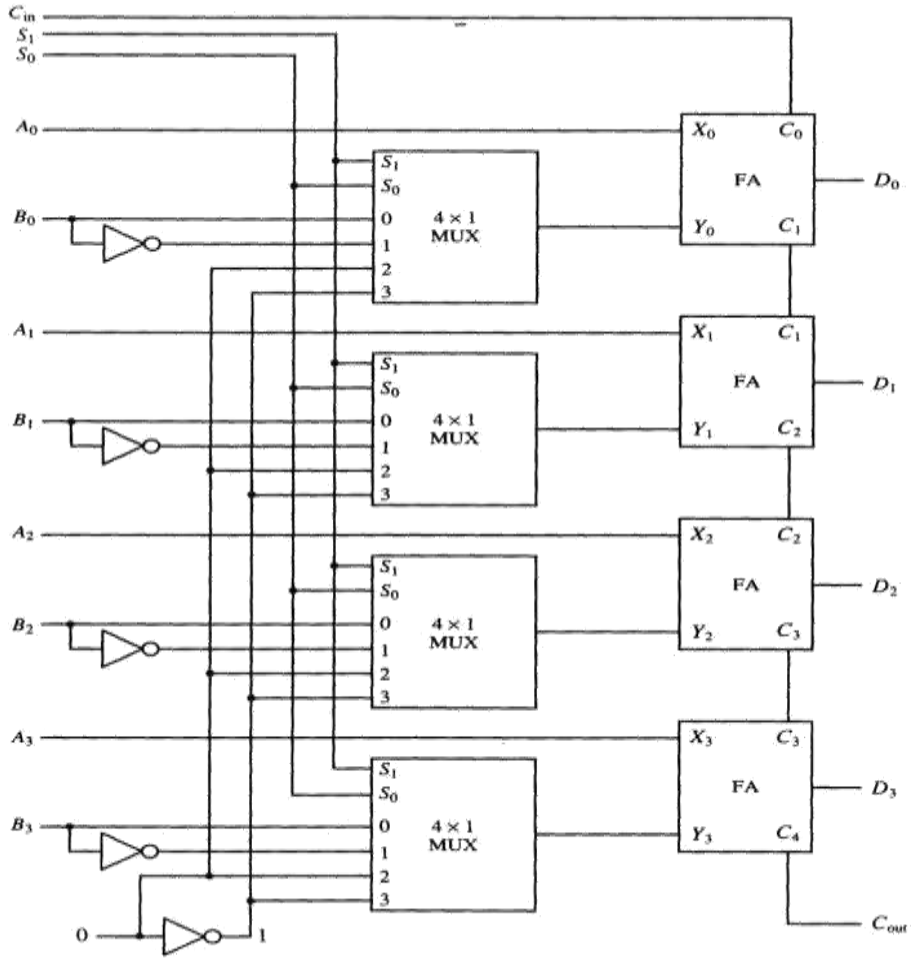


Figure 4-9 4-bit arithmetic circuit.

TABLE 4-4 Arithmetic Circuit Function Table

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

LOGIC MICRO-OPERATIONS:

- Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately
- Example: the XOR of R1 and R2 is symbolized by

  P: R1 ← R1 ⊕ R2
- Example: R1 = 1010 and R2 = 1100

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

  - Symbols used for logical Micro operations:
    - OR: ∨
    - AND: ∧
    - XOR: ⊕
  - The + sign has two different meanings: logical OR and summation
  - When + is in a microoperation, then summation
  - When + is in a control function, then OR

    Example:

    There are 16 different logic operations that can be performed with two binary variables

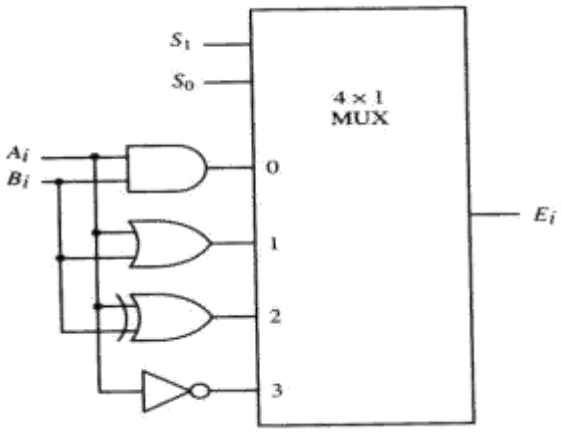
TABLE 4-5 Truth Tables for 16 Functions of Two Variables

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

TABLE 4-6 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

Figure 4-10 One stage of logic circuit.



(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

- The hardware implementation of logic Micro operations requires that logic gates be inserted for each bit or pair of bits in the registers
- All 16 Micro operations can be derived from using four logic gates

- Logic micro operations can be used to change bit values, delete a group of bits, or insert new bit values into a register
- The *selective-set* operation sets to 1 the bits in A where there are corresponding 1's in B

1010 A before
<u>1100</u> B (logic operand)

1110 A after

$A \leftarrow A \vee B$

- The *selective-complement* operation complements bits in A where there are corresponding 1's in B

1010 A before
<u>1100</u> B (logic operand)

0110 A after

$A \leftarrow A \oplus B$

- The *selective-clear* operation clears to 0 the bits in A only where there are corresponding 1's in B

1010 A before
<u>1100</u> B (logic operand)
0010 A after

$A \leftarrow A \wedge \overline{B}$

- The *mask* operation is similar to the selective-clear operation, except that the bits of A are cleared only where there are corresponding 0's in B

  1010 A before
  1100 B (logic operand)

  1000 A after

  $A \leftarrow A \wedge B$

- The *insert* operation inserts a new value into a group of bits
- This is done by first masking the bits to be replaced and then Oring them with the bits to be inserted

  0110 1010 A before
  0000 1111 B (mask)
  0000 1010 A after masking

  0000 1010 A before
  1001 0000 B (insert)
  1001 1010 A after insertion

- The *clear* operation compares the bits in A and B and produces an all 0's result if the two number are equal

  1010 A
  1010 B

  0000 $A \leftarrow A \oplus B$

SHIFT MICRO-OPERATIONS:

- Shift Micro operations are used for serial transfer of data
- They are also used in conjunction with arithmetic, logic, and other data-processing operations
- **There are three types of shifts**: logical, circular, and arithmetic
- **A logical shift** is one that transfers 0 through the serial input
- The symbols *shl* and *shr* are for logical shift-left and shift-right by one position R1 ← shl R1

- **The circular shift** (aka rotate) circulates the bits of the register around the two ends without loss of information
- The symbols *cil* and *cir* are for circular shift left and right

TABLE 4-7 Shift Microoperations

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

- **The arithmetic shift** shifts a signed binary number to the left or right
- To the left is multiplying by 2, to the right is dividing by 2
- Arithmetic shifts must leave the sign bit unchanged
- A sign reversal occurs if the bit in R_{n-1} changes in value after the shift
- This happens if the multiplication causes an overflow
- An overflow flip-flop V_s can be used to detect the overflow

  $V_s = R_{n-1} \oplus R_{n-2}$

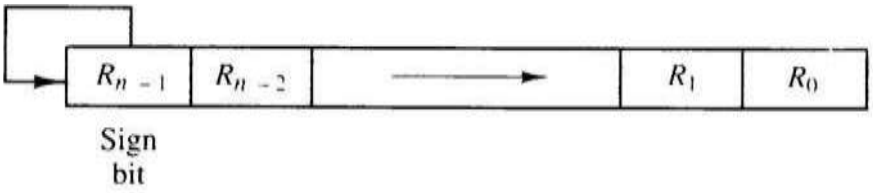
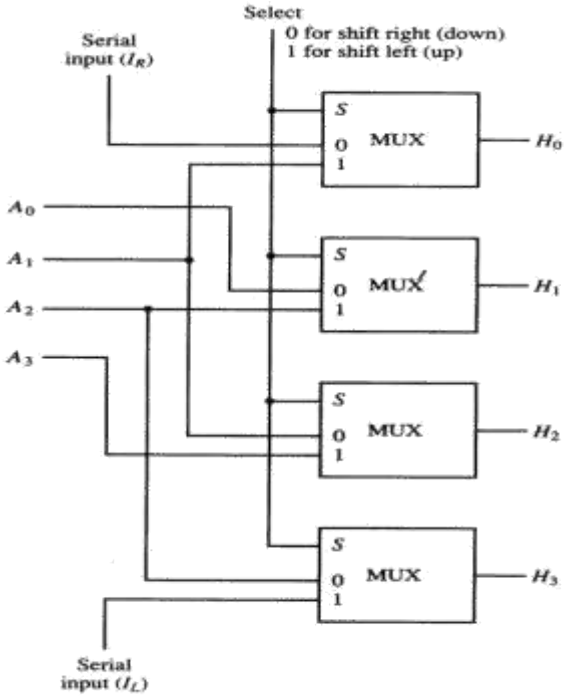


Figure 4-11 Arithmetic shift right.

- A bi-directional shift unit with parallel load could be used to implement this
- Two clock pulses are necessary with this configuration: one to load the value and another to shift
- In a processor unit with many registers, it is more efficient to implement the shift operation with a combinational circuit
- The content of a register to be shifted is first placed onto a common bus and the output is connected to the combinational shifter, the shifted number is then loaded back into the register
- This can be constructed with multiplexers



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 4-12 4-bit combinational circuit shifter.

ARITHMETIC LOGIC SHIFT UNIT:

- The *arithmetic logic unit (ALU)* is a common operational unit connected to a number of storage registers
- To perform a microoperation, the contents of specified registers are placed in the inputs of the ALU
- The ALU performs an operation and the result is then transferred to a destination register
- The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period

Figure 4-13 One stage of arithmetic logic shift unit.

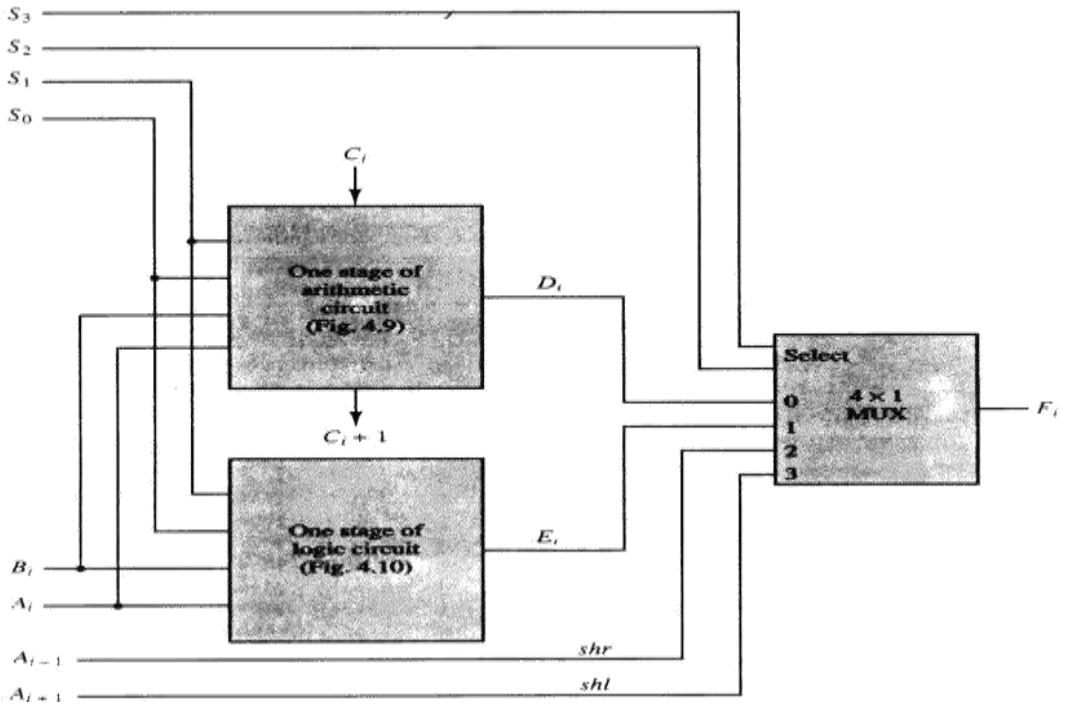


TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

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

BASIC COMPUTER ORGANIZATION AND DESIGN

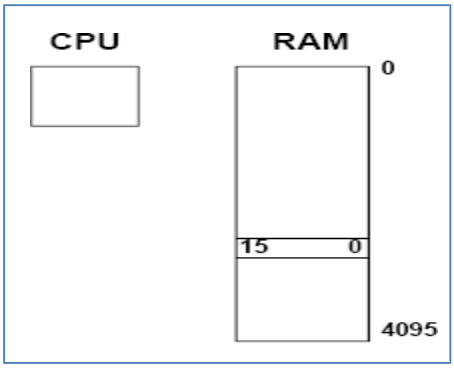
- Every different processor type has its own design (different registers, buses, micro-operations, machine instructions, etc)
- Modern processor is a very complex device
- It contains
  - Many registers
  - Multiple arithmetic units, for both integer and floating point calculations
  - The ability to pipeline several consecutive instructions to speed execution
  - Etc.
- However, to understand how processors work, we will start with a simplified processor model
- We will use this to introduce processor organization and the relationship of the RTL model to the higher level computer processor

THE BASIC COMPUTER

The Basic Computer has two components, a processor and memory

- The memory has 4096 words in it
  - 4096 = 212, so it takes 12 bits to select a word in memory

• Each word is 16 bits long



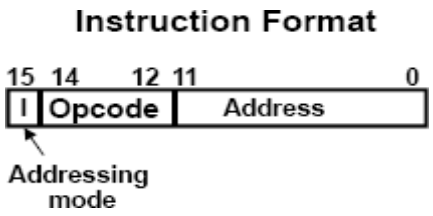
• **Program:** A sequence of (machine) instructions

- **(Machine) Instruction**: – A group of bits that tell the computer to perform a specific operation (a sequence of micro-operation)
- The instructions of a program, along with any needed data are stored in memory
- The CPU reads the next instruction from memory
- It is placed in an Instruction Register (IR)
- Control circuitry in control unit then translates the instruction into the sequence of micro operations necessary to implement it

INSTRUCTION FORMAT

- A computer instruction is often divided into two parts
  - An op code (Operation Code) that specifies the operation for that instruction
  - An address that specifies the registers and/or locations in memory to use for that operation
- In the Basic Computer, since the memory contains 4096 (= 212) words, we needs 12 bit to specify which memory address this instruction will use
- In the Basic Computer, bit 15 of the instruction specifies the addressing mode (0: direct addressing, 1: indirect addressing)
- Since the memory words, and hence the instructions, are 16 bits long, that leaves 3 bits for the instruction's op code

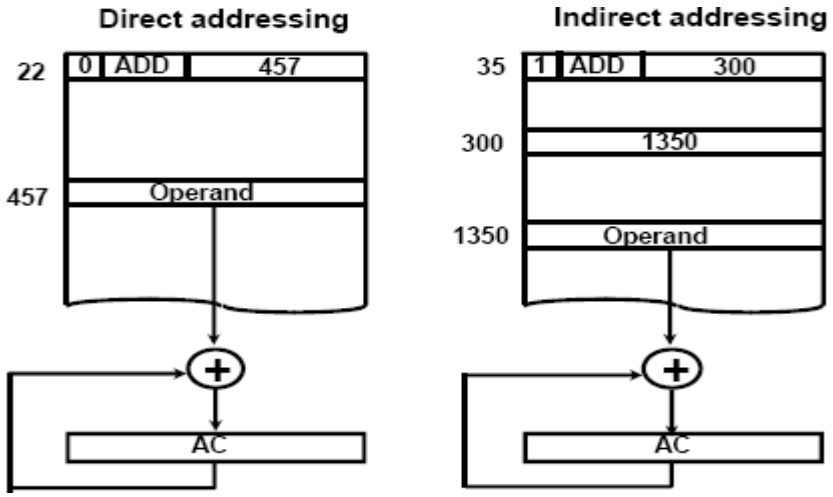
---



ADDRESSING MODES

• The address field of an instruction can represent either

- Direct address: the address in memory of the data to use (the address of the operand), or
- Indirect address: the address in memory of the address in memory of the data to use



Effective Address (EA): – The address, that can be directly used without modification to access an Operand for a computation-type instruction, or as the target address for a branch-type instruction

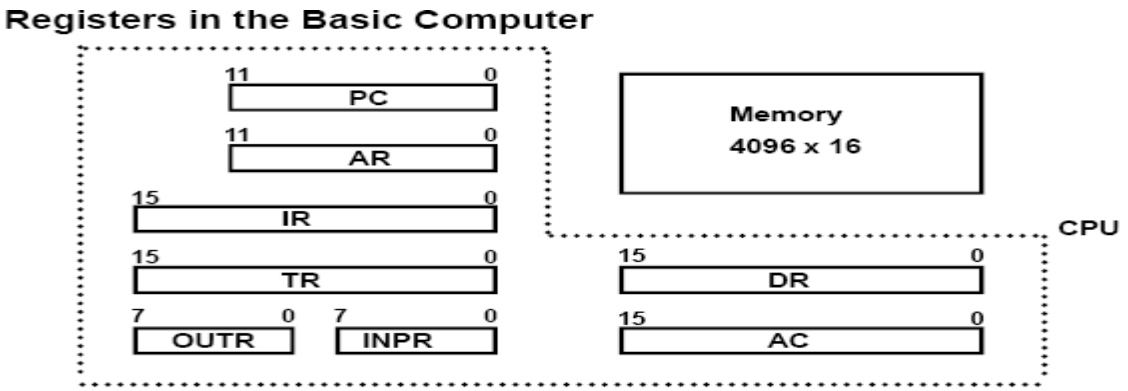
PROCESSOR REGISTERS

- A processor has many registers to hold instructions, addresses, data, etc
- The processor has a register, the Program Counter (PC) that holds the memory address of the next instruction to get
  - Since the memory in the Basic Computer only has 4096 locations, the PC only needs 12 bits
- In a direct or indirect addressing, the processor needs to keep track of what locations in memory it is addressing: The
- Address Register (AR) is used for this
  - The AR is a 12 bit register in the Basic Computer
- When an operand is found, using either direct or indirect addressing, it is placed in the Data Register (DR). The processor then uses this value as data for its operation
- The Basic Computer has a single general purpose register – the Accumulator (AC)
- The significance of a general purpose register is that it can be referred to in instructions
  - e.g. load AC with the contents of a specific memory location; store the contents of AC into a specified memory location
- Often a processor will need a scratch register to store intermediate results or other temporary data; in the Basic Computer this is the Temporary Register (TR)
- The Basic Computer uses a very simple model of input/output (I/O) operations

- Input devices are considered to send 8 bits of character data to the processor
- The processor can send 8 bits of character data to output devices

- The Input Register (INPR) holds an 8 bit character gotten from an input device
- The Output Register (OUTR) holds an 8 bit character to be send to an output device

BASIC COMPUTER REGISTERS

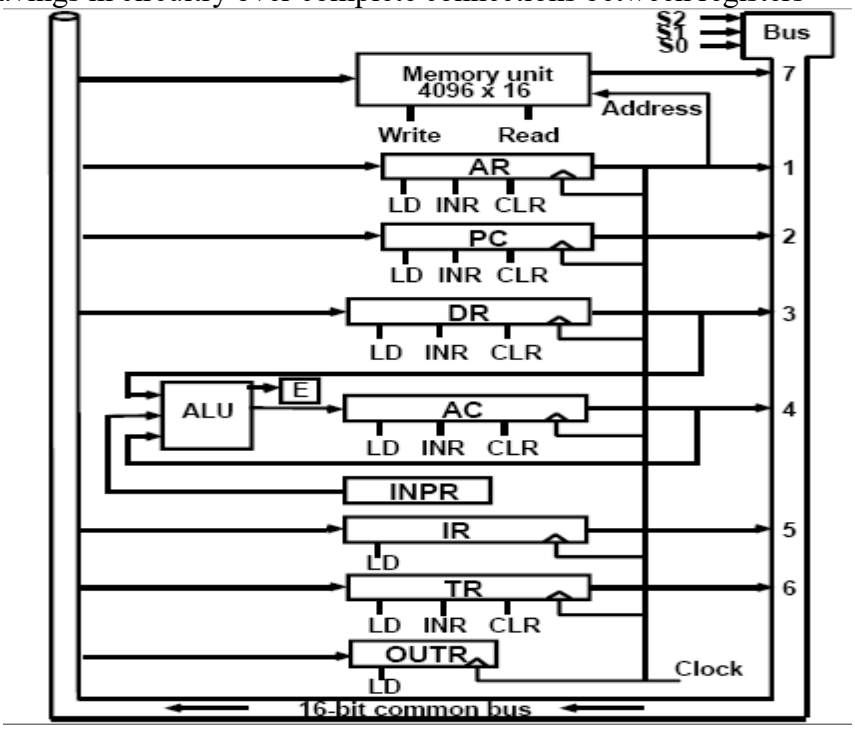


List of BC Registers

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

COMMON BUS SYSTEM:

- The registers in the Basic Computer are connected using a bus
- This gives a savings in circuitry over complete connections between registers



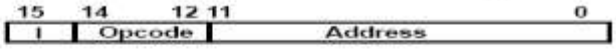
Three control lines, S2, S1, and S0 control which register the bus selects as its input

S_2	S_1	S_0	Register
0	0	0	x
0	0	1	AR
0	1	0	PC
0	1	1	DR
1	0	0	AC
1	0	1	IR
1	1	0	TR
1	1	1	Memory

- Either one of the registers will have its load signal activated, or the memory will have its read signal activated
  - Will determine where the data from the bus gets loaded
- The 12-bit registers, AR and PC, have 0's loaded onto the bus in the high order 4 bit positions
- When the 8-bit register OUTR is loaded from the bus, the data comes from the low order 8 bits on the bus

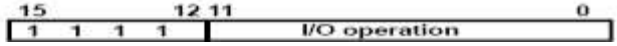
BASIC COMPUTER INSTRUCTIONS: Basic Computer Instruction Format

Memory-Reference Instructions (OP-code = 000 ~ 110)



Register-Reference Instructions (OP-code = 111, I = 0)

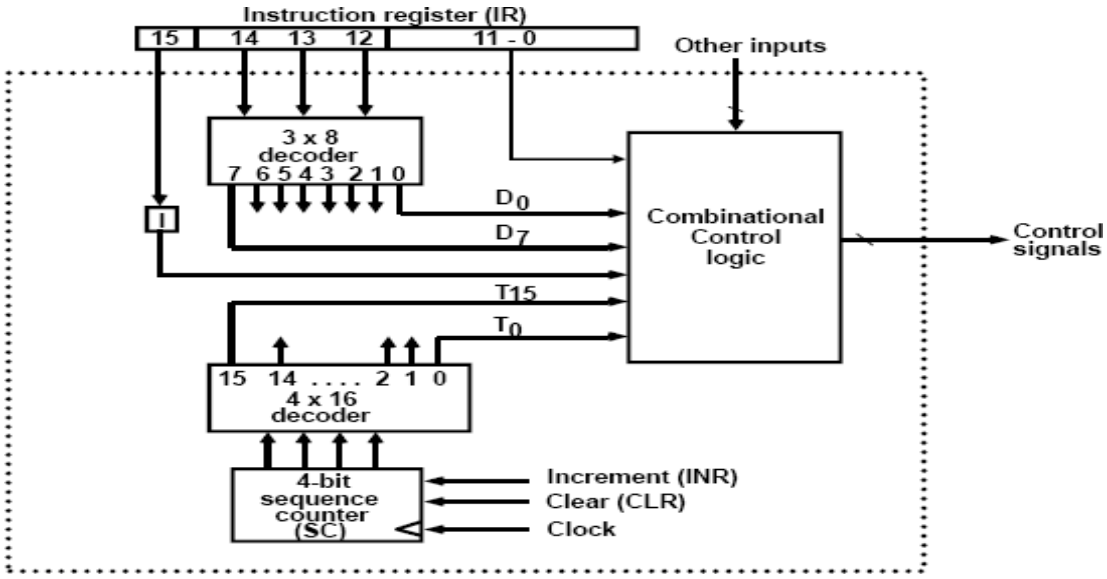
Input-Output Instructions (OP-code =111, I = 1)



Symbol	*Hex Code* *I = 0*	*I = 1*	*Description*
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into 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 instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
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

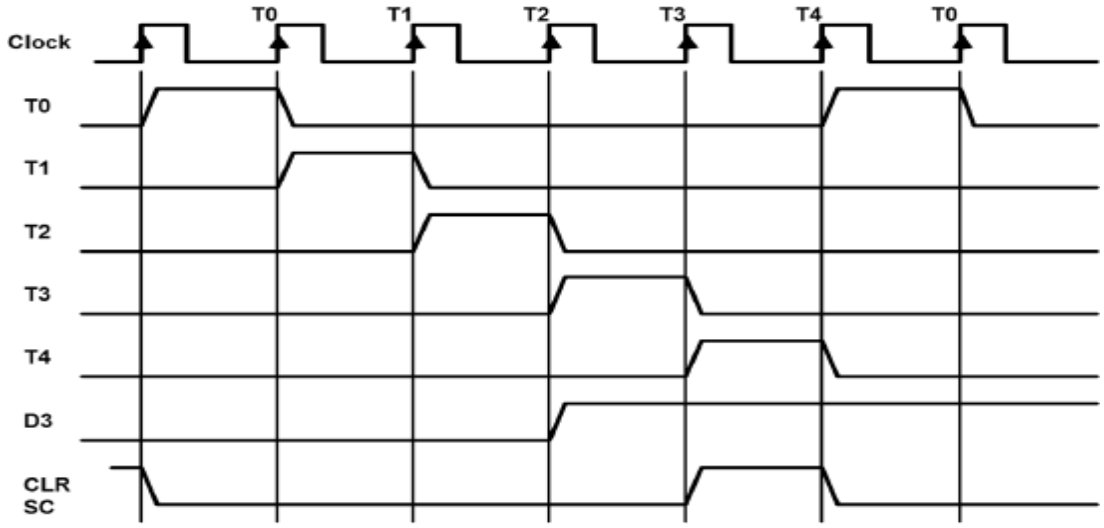
TIMING AND CONTROL

Control unit of Basic Computer



- Generated by 4-bit sequence counter and 4 x 16 decoder
- The SC can be incremented or cleared.
- Example: T0, T1, T2, T3, T4, T0, T1 . . .

Assume: At time T4, SC is cleared to 0 if decoder output D3 is active. D3T4: SC ←0



INSTRUCTION CYCLE

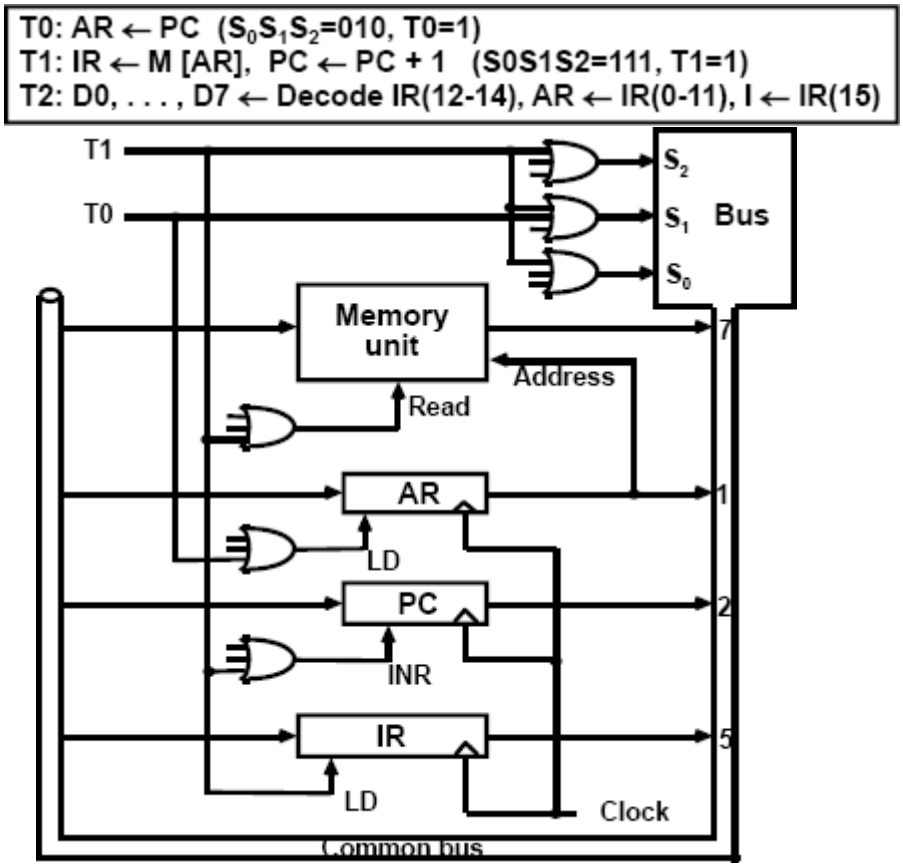
• In Basic Computer, a machine instruction is executed in the following cycle:

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

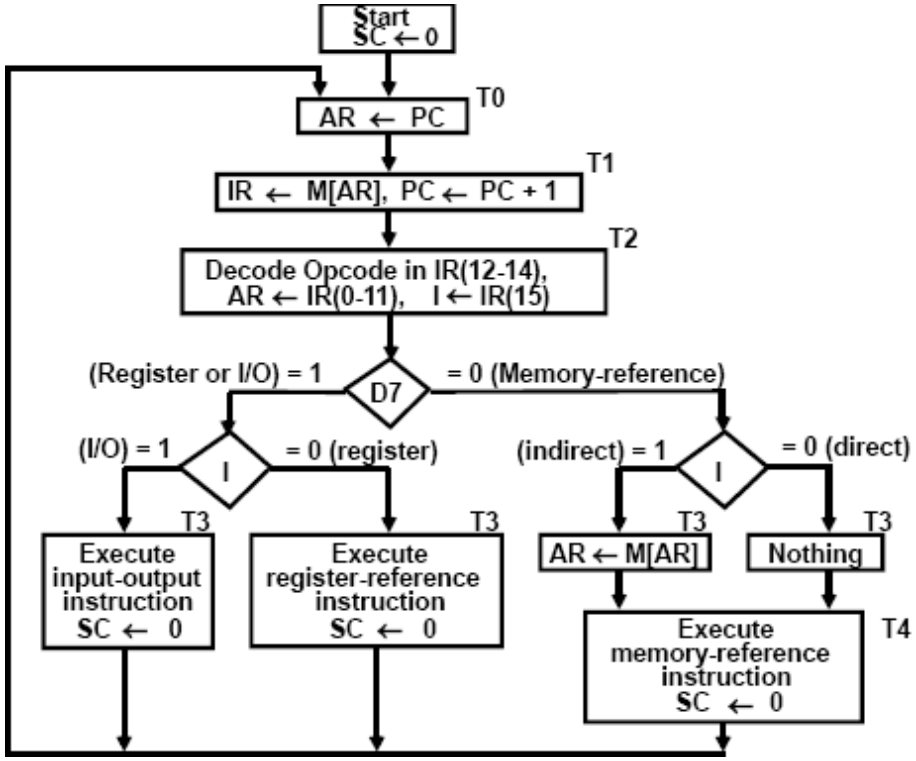
• After an instruction is executed, the cycle starts again at step 1, for the next instruction

---

FETCH and DECODE



DETERMINE THE TYPE OF INSTRUCTION



D'_7IT_3: $AR \leftarrow M[AR]$
$D'_7I'T_3$: Nothing
$D_7I'T_3$: Execute a register-reference instr.
D_7IT_3: Execute an input-output instr.

REGISTER REFERENCE INSTRUCTIONS

Register Reference Instructions are identified when

- $D_7 = 1$, $I = 0$
- Register Ref. Instr. is specified in $b_0 \sim b_{11}$ of IR
- Execution starts with timing signal T_3

$r = D_7 I'T_3$ => Register Reference Instruction
$B_i = IR(i)$, i=0,1,2,...,11

	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow AC'$
CME	rB_8:	$E \leftarrow E'$
CIR	rB_7:	$AC \leftarrow shr\ AC, AC(15) \leftarrow E, E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow shl\ AC, AC(0) \leftarrow E, E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	if (AC(15) = 0) then ($PC \leftarrow PC+1$)
SNA	rB_3:	if (AC(15) = 1) then ($PC \leftarrow PC+1$)
SZA	rB_2:	if (AC = 0) then ($PC \leftarrow PC+1$)
SZE	rB_1:	if (E = 0) then ($PC \leftarrow PC+1$)
HLT	rB_0:	$S \leftarrow 0$ (S is a start-stop flip-flop)

MEMORY REFERENCE INSTRUCTIONS

Symbol	Operation Decoder	Symbolic Description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, if $M[AR] + 1 = 0$ then $PC \leftarrow PC+1$

- The effective address of the instruction is in AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I = 1
- Memory cycle is assumed to be short enough to complete in a CPU cycle
- The execution of MR instruction starts with T_4

AND to AC

D_0T_4: DR ← M[AR] Read operand

D_0T_5: AC ← AC ∧ DR, SC ← 0 AND with AC

ADD to AC

D_1T_4: DR ← M[AR] Read operand

D_1T_5: AC ← AC + DR, E ← C_{out}, SC ← 0 Add to AC and store carry in E

LDA: Load to AC

D_2T_4: DR ← M[AR]

D_2T_5: AC ← DR, SC ← 0

STA: Store AC

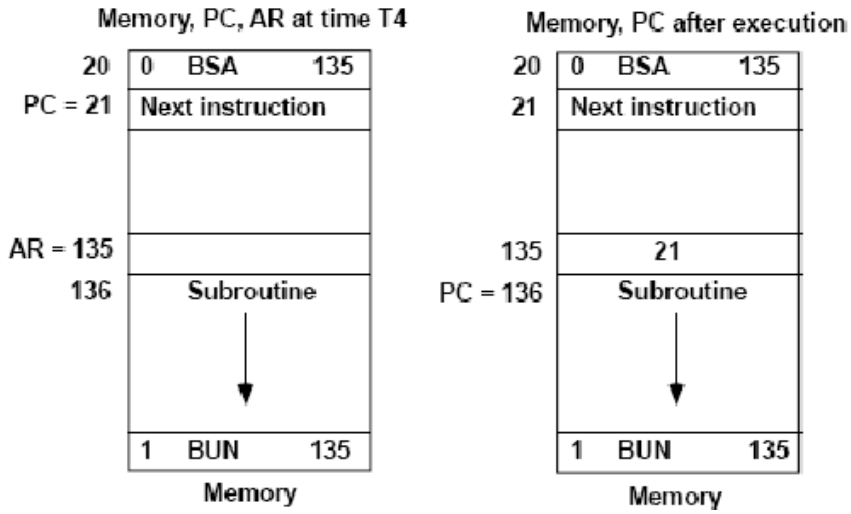
D_3T_4: M[AR] ← AC, SC ← 0

BUN: Branch Unconditionally

D_4T_4: PC ← AR, SC ← 0

BSA: Branch and Save Return Address

M[AR] ← PC, PC ← AR + 1



BSA:

D_5T_4: M[AR] ← PC, AR ← AR + 1

D_5T_5: PC ← AR, SC ← 0

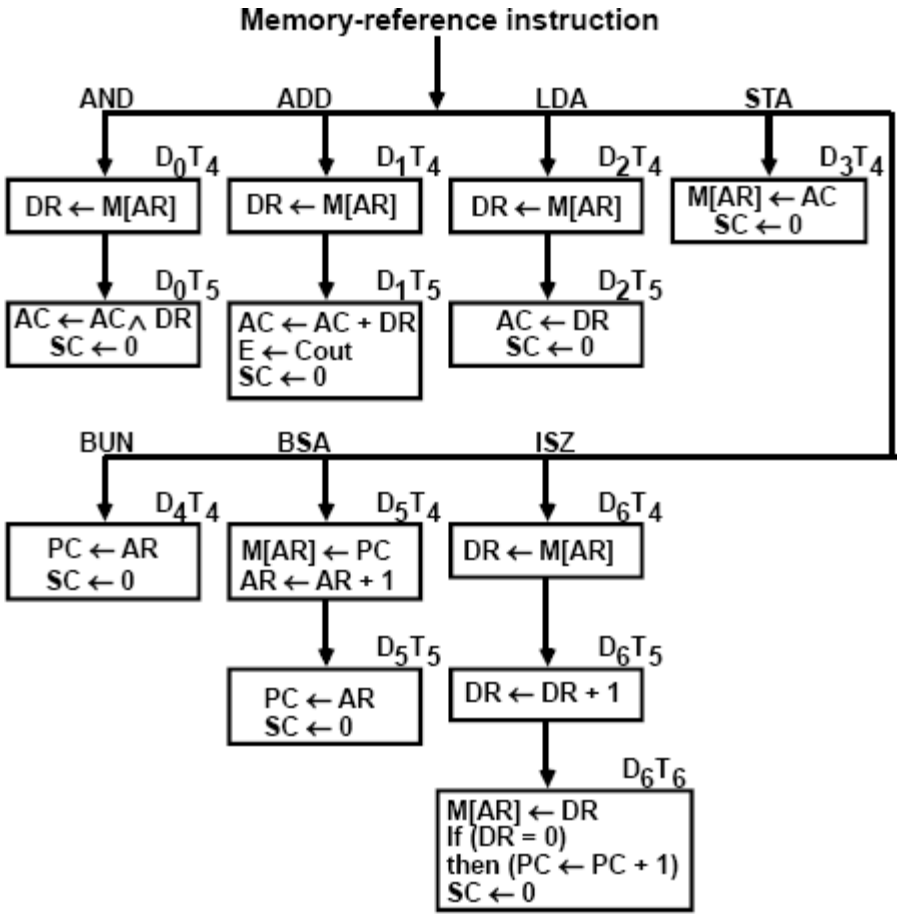
ISZ: Increment and Skip-if-Zero

D_6T_4: DR ← M[AR]

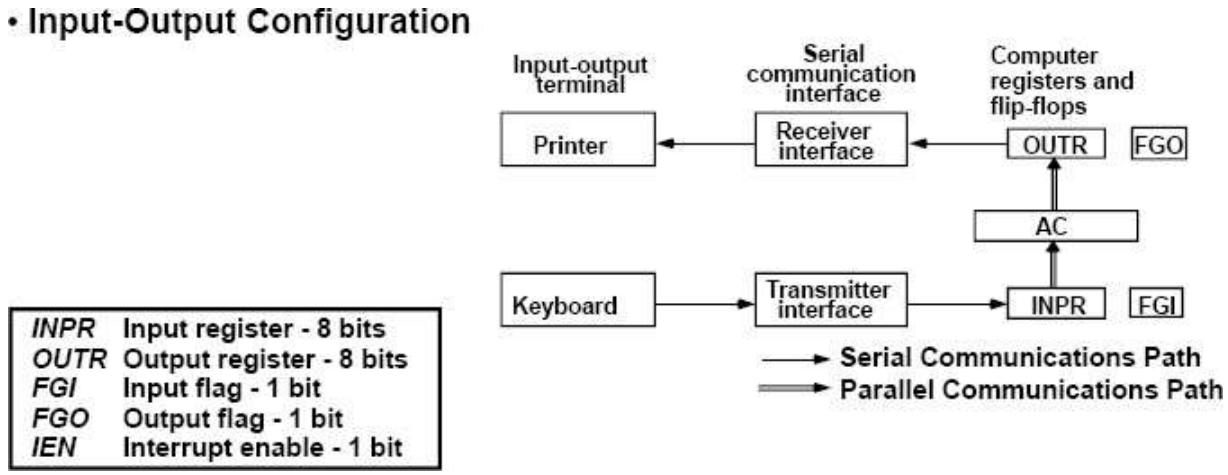
D_6T_5: DR ← DR + 1

D_6T_4: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

FLOWCHART FOR MEMORY REFERENCE INSTRUCTIONS



INPUT-OUTPUT AND INTERRUPT



- The terminal sends and receives serial information
- The serial info. From the keyboard is shifted into INPR
- The serial info. for the printer is stored in the OUTR
- INPR and OUTR communicate with the terminal serially and with the AC in parallel.
- The flags are needed to synchronize the timing difference between I/O device and the computer

---

INPUT-OUTPUT INSTRUCTIONS

$D_7IT_3 = p$
$IR(i) = B_i,\ i = 6, \ldots, 11$

	p:	SC ← 0	Clear SC
INP	pB_{11}:	AC(0-7) ← INPR, FGI ← 0	Input char. to AC
OUT	pB_{10}:	OUTR ← AC(0-7), FGO ← 0	Output char. from AC
SKI	pB_9:	if(FGI = 1) then (PC ← PC + 1)	Skip on input flag
SKO	pB_8:	if(FGO = 1) then (PC ← PC + 1)	Skip on output flag
ION	pB_7:	IEN ← 1	Interrupt enable on
IOF	pB_6:	IEN ← 0	Interrupt enable off

- Program-controlled I/O
  - Continuous CPU involvement
    I/O takes valuable CPU time
  - CPU slowed down to I/O speed
  - Simple
  - Least hardware

Input

```
LOOP,   SKI   DEV
        BUN   LOOP
        INP   DEV
```

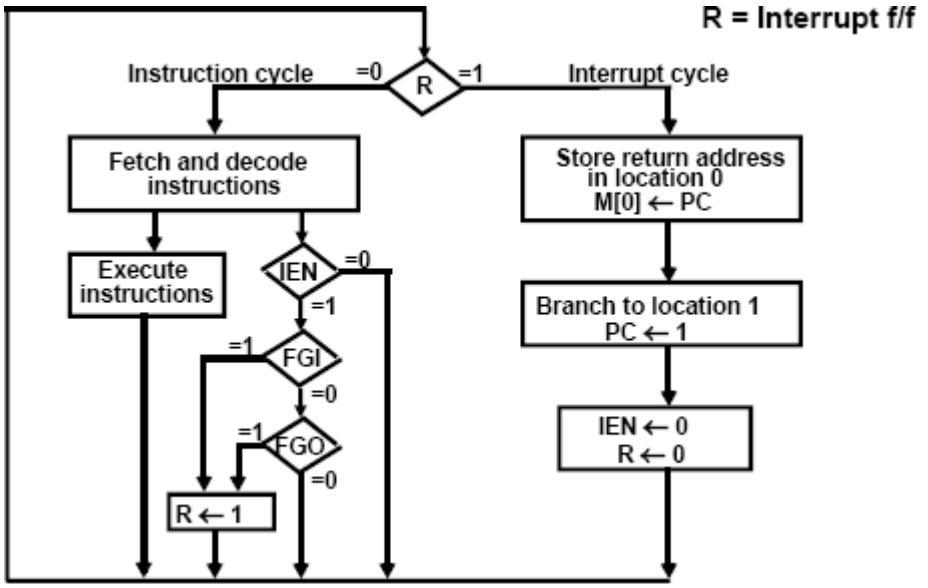
Output

```
LOOP,   LDA   DATA
LOP,    SKO   DEV
        BUN   LOP
        OUT   DEV
```

INTERRUPT INITIATED INPUT/OUTPUT

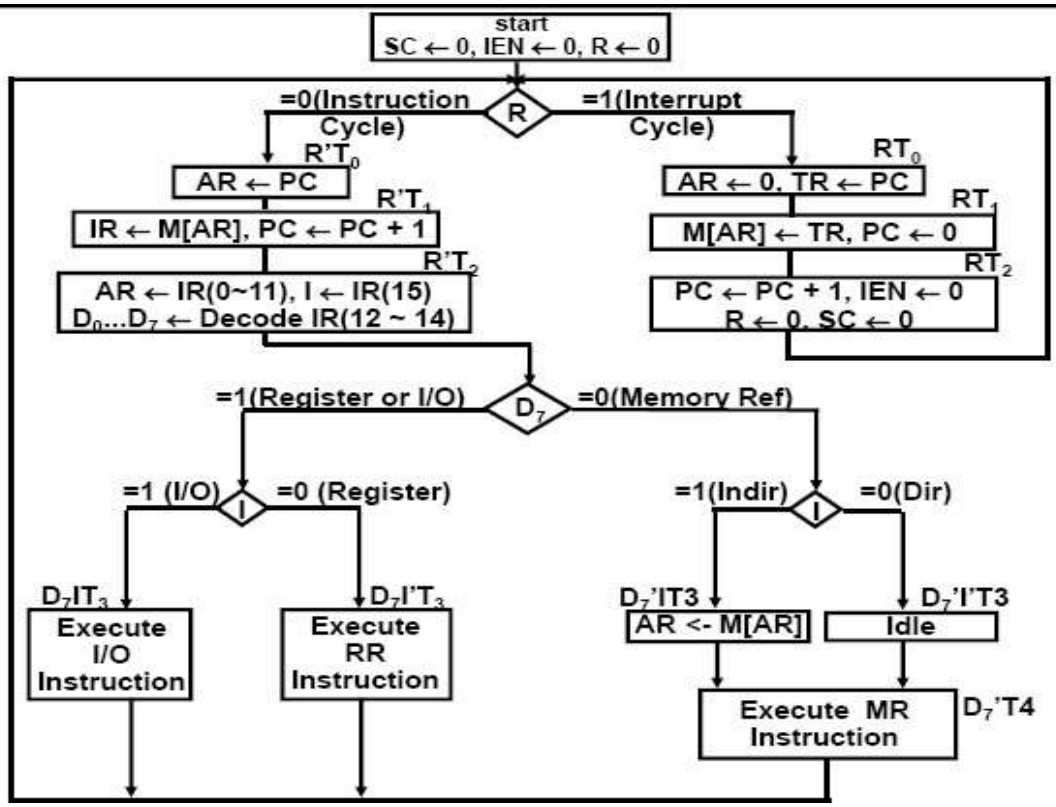
- Open communication only when some data has to be passed --> interrupt.
- The I/O interface, instead of the CPU, monitors the I/O device.
- When the interface founds that the I/O device is ready for data transfer, it generates an interrupt request to the CPU
- Upon detecting an interrupt, the CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing.

* IEN (Interrupt-enable flip-flop)
- can be set and cleared by instructions
- when cleared, the computer cannot be interrupted



- The interrupt cycle is a HW implementation of a branch and save return address operation.
- At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1.
- At memory address 1, the programmer must store a branch instruction that sends the control to an interrupt service routine
- The instruction that returns the control to the original program is "indirect BUN 0"

COMPLETE COMPUTER DESCRIPTION



Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR], PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D0, \ldots, D7 \leftarrow$ Decode $IR(12 \sim 14)$, $AR \leftarrow IR(0 \sim 11), I \leftarrow IR(15)$
Indirect	$D_7'IT_3$:	$AR \leftarrow M[AR]$
Interrupt		
$T_0'T_1'T_2'(IEN)(FGI + FGO)$:		$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0, TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR, PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$
Memory-Reference		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR, SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR, SC \leftarrow 0$
STA	D_3T_4:	$M[AR] \leftarrow AC, SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR, SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC, AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR, SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$, if$(DR=0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$

Register-Reference		
	$D_7I'T_3 = r$	(Common to all register-reference instr)
	$IR(i) = B_i$	$(i = 0,1,2, \ldots, 11)$
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow AC'$
CME	rB_8:	$E \leftarrow E'$
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E, E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E, E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	If$(AC(15) =0)$ then $(PC \leftarrow PC + 1)$
SNA	rB_3:	If$(AC(15) =1)$ then $(PC \leftarrow PC + 1)$
SZA	rB_2:	If$(AC = 0)$ then $(PC \leftarrow PC + 1)$
SZE	rB_1:	If$(E=0)$ then $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
Input-Output	$D_7IT_3 = p$	(Common to all input-output instructions)
	$IR(i) = B_i$	$(i = 6,7,8,9,10,11)$
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{-}7) \leftarrow INPR, FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{-}7), FGO \leftarrow 0$
SKI	pB_9:	If$(FGI=1)$ then $(PC \leftarrow PC + 1)$
SKO	pB_8:	If$(FGO=1)$ then $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$