1.3. COMPUTER ARCHITECTURE AND ORGANIZATION

Computer Architecture is concerned with the structure and behavior of the computer as seen by the user. It includes the information formats, the instruction sets, 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.

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

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.

1.4. ELEMENTS OF A CPU

The function of the CPU is executing the program stored in the memory. For doing this, the CPU fetches one instruction at a time, executes it and then takes up next instruction. This action is done repeatedly and is known as instruction cycle. As shown in Fig. 1.7, the instruction cycle consists of two phases: fetch phase and execute phase. In fetch phase, an instruction is fetched from memory. In execute phase, the instruction is analyzed and relevant operations are performed.

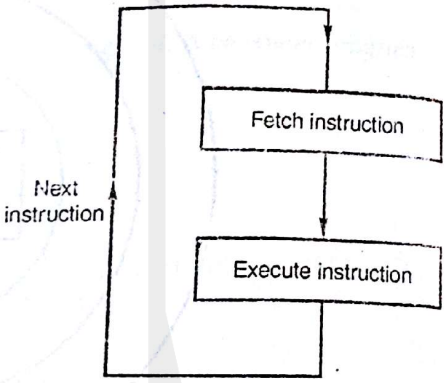


Fig. 1.7. Instruction cycle phases

1.4.1. CPU Registers

The CPU consists of following major registers (Fig. 1.8) :

1. Accumulator (AC)

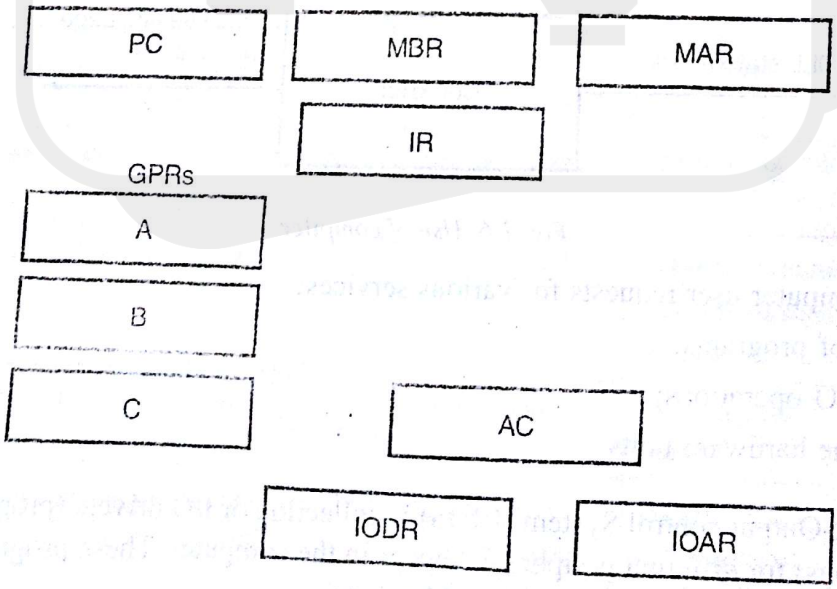


Fig. 1.8. CPU registers

2. Program Counter (PC)
3. Memory Address Register (MAR)
4. Memory Buffer Register (MBR)
5. Instruction Register (IR)
6. General Purpose Registers (GPR)
7. I/O Data Register (IODR)
8. I/O Address Register (IOAR)

The adder performs arithmetic operations such as addition, subtraction etc. The accumulator register holds the result of previous operation in ALU. It is also used as an input register to the adder. The program counter (instruction address counter) contains the address of the memory location from where next instruction has to be fetched. As soon as one instruction fetch is complete, the contents of PC are incremented so as to point to the next instruction address. The instruction register stores the current instruction fetched from memory. The MAR contains the address of memory location during a memory read/write operation. The MBR contains the data read from memory (during read) or data to be written into memory (during write). THE GPRs are used for multiple purposes: storing operands, addresses, constants etc. In addition to GPRs, the CPU also has some working registers known as scratch pad memory. These are used to keep the intermediate results within an instruction cycle for complex instructions such as MULTIPLY, DIVIDE etc.

1.4.2. Clock

The clock unit generates and supplies a continuous sequence of clock pulses. The clock signal is used as a timing reference by the control unit. The clock signal is a periodic waveform since the waveform repeats itself. The rate at which the periodic waveform repeats is known as the frequency (f). It is specified as cycles per second (cps) or Hz. The clock frequency is an indication of the internal operating speed of the processor. The fixed interval at which the periodic signal repeats is called as its time period (T). The relationship between the frequency and the period is

$$f = \frac{1}{T}$$

The pulse width (tpw) indicates the duration of the (clock) pulse. Another related term is duty cycle. The duty cycle is the ratio (expressed in percentage) of pulse width to period, *i.e.*, duty cycle = (tpw/T) × 100%. Figure 1.9 illustrates the terms period, frequency, and duty cycle.

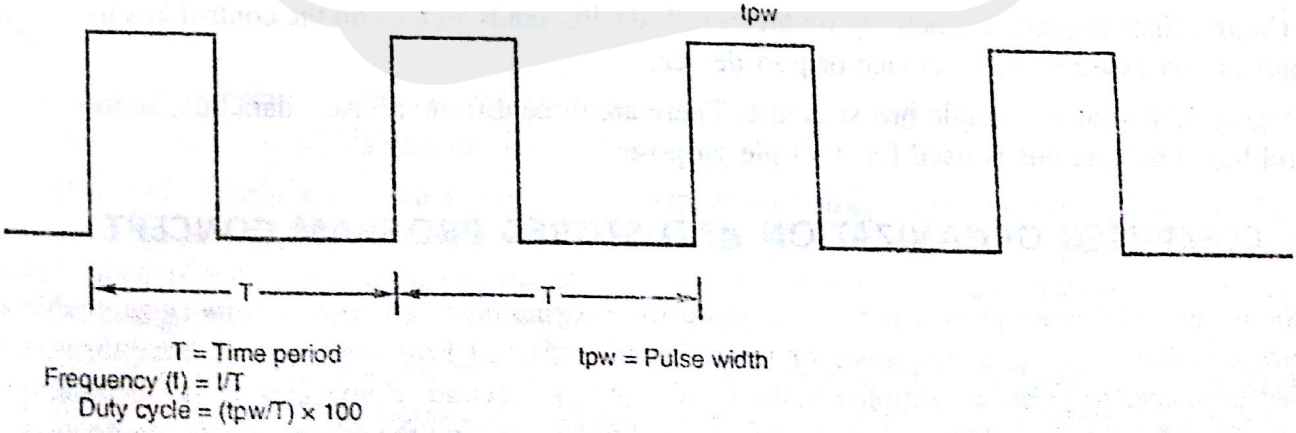


Fig. 1.9. Periodic waveforms

Example 1.1. *A clock signal has a frequency of 10 MHz with a duty cycle of 50%. Calculate its period and pulse width.*

Data : $f = 10\ MHz$; *duty cycle* $= 50\%$;

Solution. Time period $(T) = \frac{1}{f} = \frac{1}{(10 \times 10^6)} = 100\ \eta s$;

$$\text{Duty cycle} = \left(\frac{tpw}{T}\right) \times 100 = 50\%;$$

$$tpw = 0.5\ T = 50\ \eta s.$$

1.4.3. CPU Busses

A group of wires connecting two or more devices and providing a path to perform communication is called Bus. A bus that connects major computer components/modules (CPU, Memory, I/O) is called system bus. These system buses are separated into three functional groups:

1. Data bus.
2. Address bus.
3. Control bus.

Data Bus

The data bus consists of 8, 16, 32 or more parallel lines. The data bus lines are bidirectional. This means that CPU can read data on these lines from memory or from a port as well as data out on these lines to a memory location or to a port.

Address Bus

It is a unidirectional bus. The address bus consists of 16, 20, 24 or more parallel lines. The CPU sends out the address of the memory location or I/O port that is to be written or read from by using this address bus.

Control Bus

Control lines regulate the activity on the bus. The CPU sends signals on the control bus to enable the outputs of addressed memory device or port device.

Figure 1.10 shows a simple bus structure. There are three different buses: data bus, address bus and control bus. The data bus is used for multiple purposes:

1.5. COMPUTER ORGANIZATION AND STORED PROGRAM CONCEPT

Stored program concept is a process to store the instruction in computer memory to enable it to perform a variety of tasks in sequence or intermittently. 'Stored Program' was the breakthrough that enabled computer to perform complex tasks in fractions of second. Being able to do arithmetic and evaluate formulas fast would be useless if a person had to keep telling the computer what to do next. The

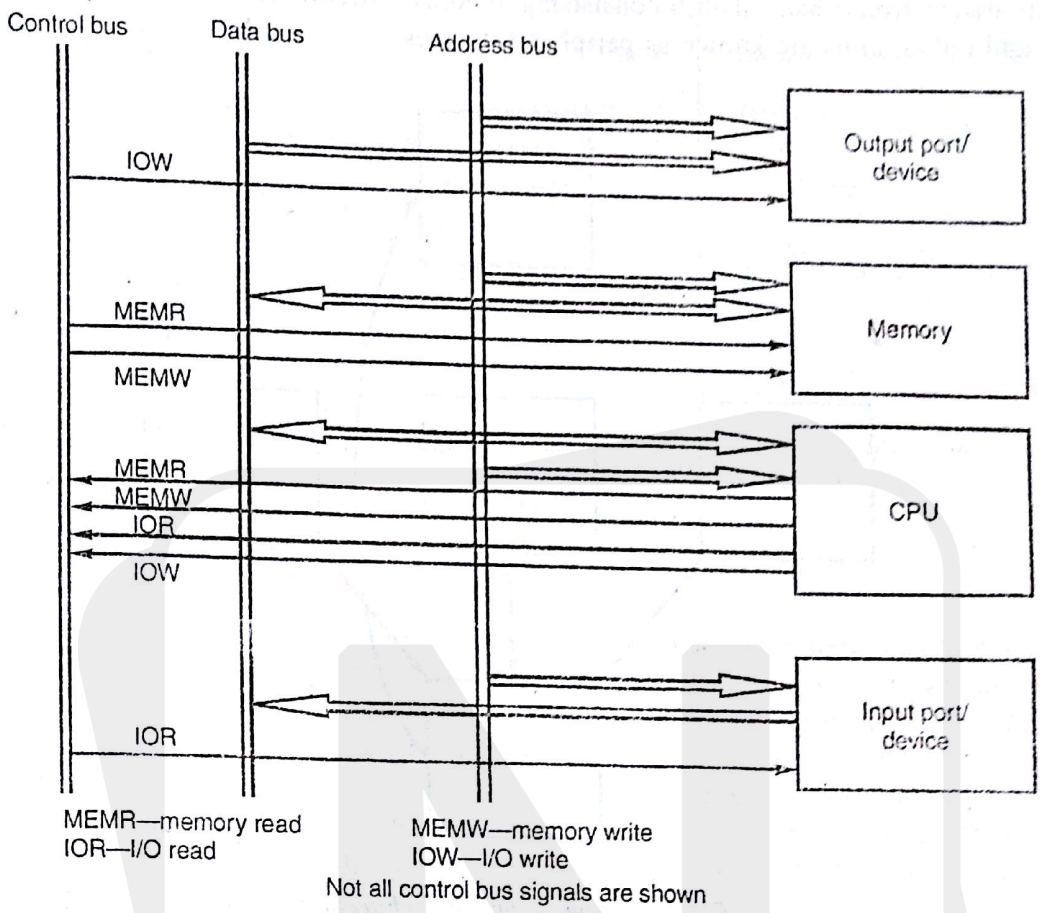


Fig. 1.10. Simple bus structure

trick was to encoding the list of instructions into numbers that could coexist in the computer's memory along with data, and having a 'processor' build into the machine to decode and execute the instructions. Now the computer could do both, calculations and control of the sequence at electronic speed. The idea was introduced by **John Von Neumann**, who proposed that a program can be electronically stored in binary-number format in a memory device so that instructions could be modified by the computer as determined by intermediate computational results.

A **Von Neumann Architecture** computer have five parts *i.e.*,

(*a*) An arithmetic-logic-unit

(*b*) A control unit

(*c*) A memory

(*d*) Some form of input/output and

(*e*) a bus that provides a data path between these parts.

The ALU and control unit have usually some temporary storage units known as registers. Each register can be considered as a fast memory with single location. Such registers temporarily store certain information such as instruction, data, address, etc. Storing in registers is advantageous since these can be read quickly compared to fetching them from external memory.

The ALU and control unit are together known as Central Processing Unit (CPU) or processor. The memory and CPU consist of electronic circuits and form the nucleus of the computer. The input and

output units are electromechanical units consistsing of both electronic circuits and mechanical assemblies. The input and output units are known as peripheral devices.

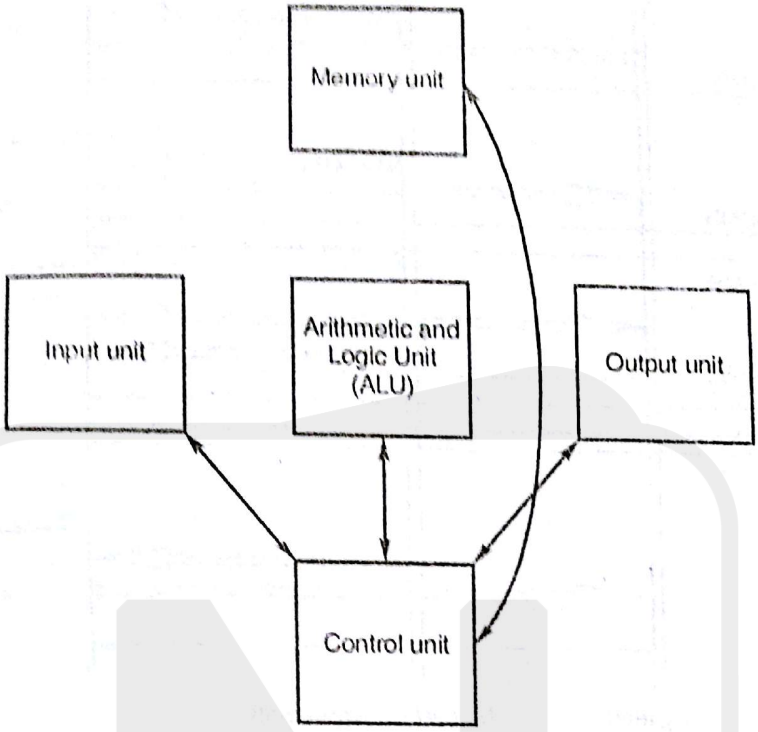


Fig. 1.11. Von Neumann Architecture

A von Neumann Architecture computer performs following sequence of steps:

1. Fetch the next instruction from memory at the address in the program counter.
2. ADD 1 to the program counter.
3. Decode the instruction using control unit. The control unit commands the rest of the computer to perform some operation.
4. Go back to step 1.

Limitation in Von Neumann Model

Very few computers have a pure von Neumann architecture. Most computers add another step to check for interrupts, electronic events that could occur at any time. An interrupt resembles the ring of a telephone, calling a person away from some lengthy task. Von Neumann computers spend a lot of time moving data to and from the memory, and this slows the computer. So, often we separate the bus into two or more busses, usually one for instruction and the other for data.

Difference between Von Neumann architecture and Harvard architecture?

Harvard architecture has separate data and instruction busses, allowing transfers to be performed simultaneously on both busses.

Von Neumann architecture has only one bus which is used for both data transfers and instruction fetches, and therefore data transfers and instruction fetches must be scheduled - they cannot be performed at the same time.

3.7. BUS AND MEMORY TRANSFER

A computer has many registers and paths must be provided to transfer the information from one register to another register. This transfer done by group wires called bus. To achieve a reasonable speed of operation, a computer must be recognised so that all its units can handle one full word of data. When data is transferred between units, all its bits are transferred in parallel, that is, the bits are transferred simultaneously over different wires. Many wires are required to establish the necessary connections. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A scheme for transferring information between the registers is a common bus system. Some control lines are required to manage the common bus system.

Bus: A group of wires that connected several devices to carry the data (or) information is called as a bus.

Bus Transfer: The data transfer between various blocks connected to the common bus is called bus transfer.

Memory Transfer: The data is stored (write) into memory or read from memory is called memory transfer.

3.7.1. Bus Transfer

A more efficient scheme for transferring information between registers in a multiple register configuration is a common bus system. It is shared by all the units. Switches are required to enable paths to be shared as shown in Fig. 3.14. These switches are implemented by multiplexer or tristate non-inverting buffers. One common control signal is used to control all the switches, the switches are closed (connection enabled) or open (connections disabled).

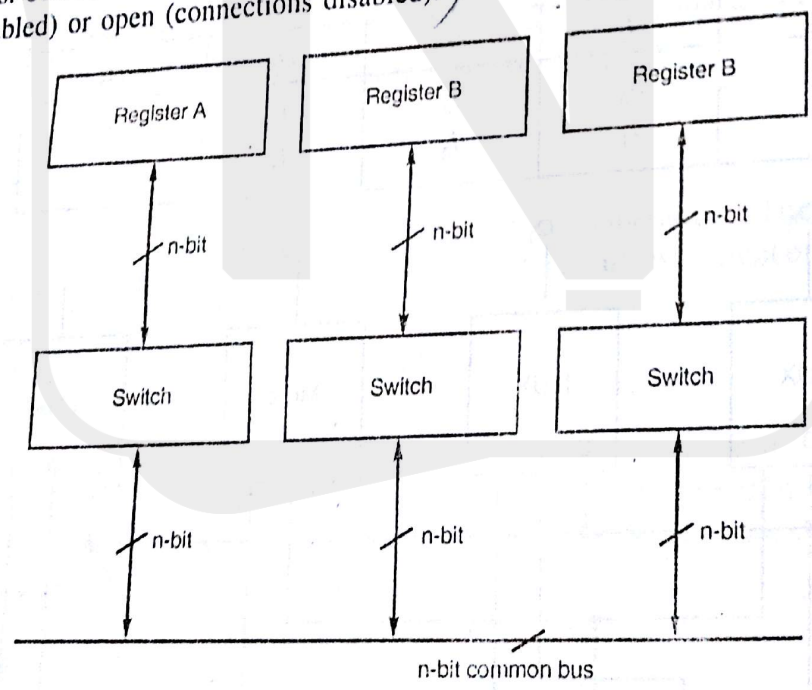


Fig. 3.14. Shared common bus system.

3.7.1.1. Shared Data Path using MUX

Figure 3.15 shows the shared data path using multiplexers. The required multiplexers is based on the number of registers to be used in the data path.

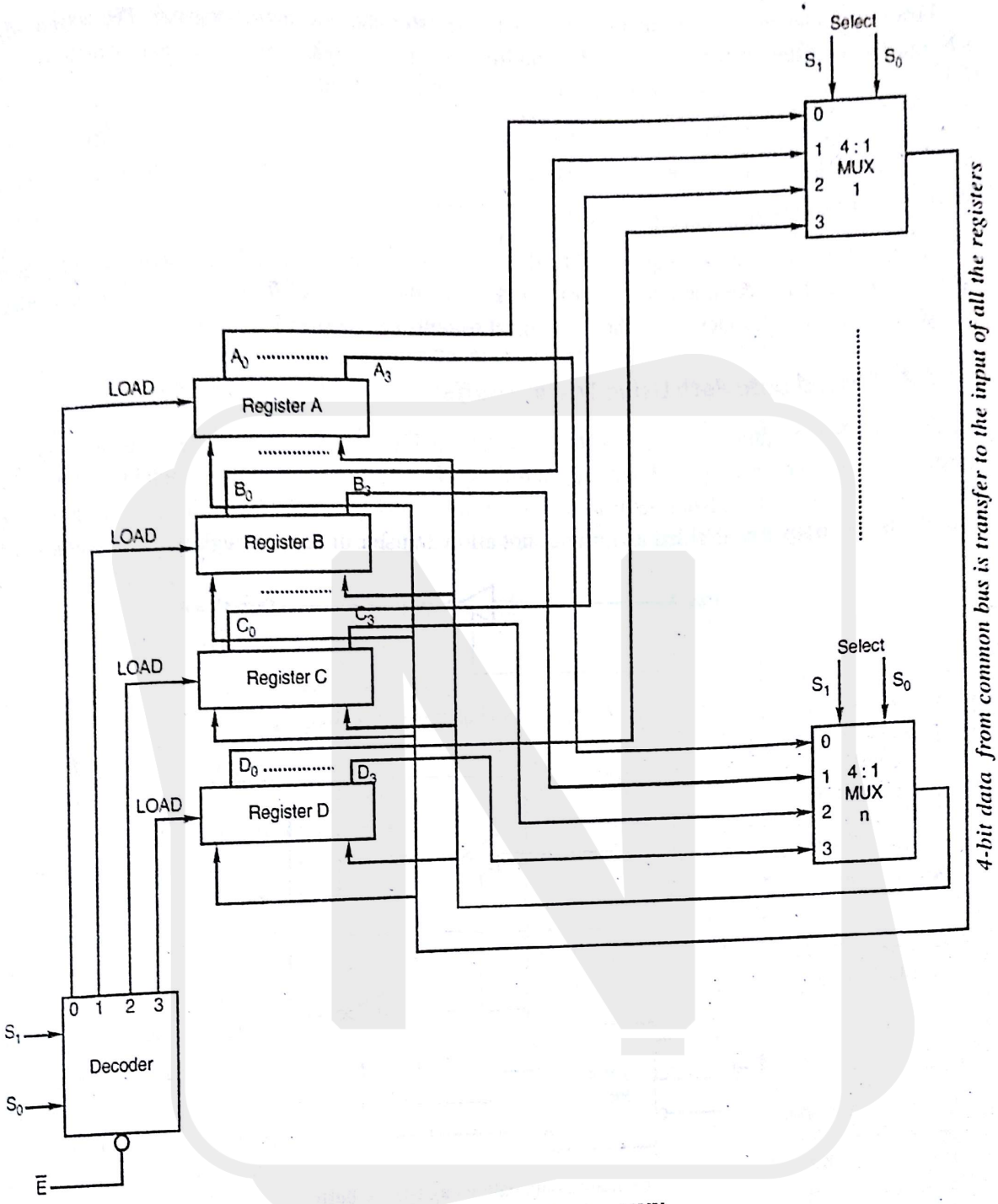


Fig. 3.15. Shared data path using MUX.

The multiplexers select one source register bus, and the decoder selects one destination register to transfer the information. The 4 bits in the same significant position in the register group go through a 4 to 1 multiplexer to form one line of bus. For n bit registers, n multiplexers are needed to produce an n-common line bus (or) shared data path. The transfer of information from the bus into its destination register is accomplished by activbating load control of that register. If the decoder is not enabled, no information will be transferred between the registers.

For example, consider the following statement

$$C \leftarrow B$$

From the statement, we can identify the source register and destination register. The source register is B and the destination register is C. The multiplexer and decoder select the lines which satisfy the following condition to transfer the data from register B into register C.

MUX select line $S_1S_0 = 01$

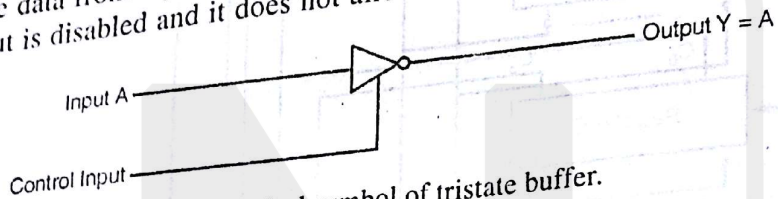
Decoder select line $S_1S_0 = 10$

Decoder enable $\overline{E} = 0$

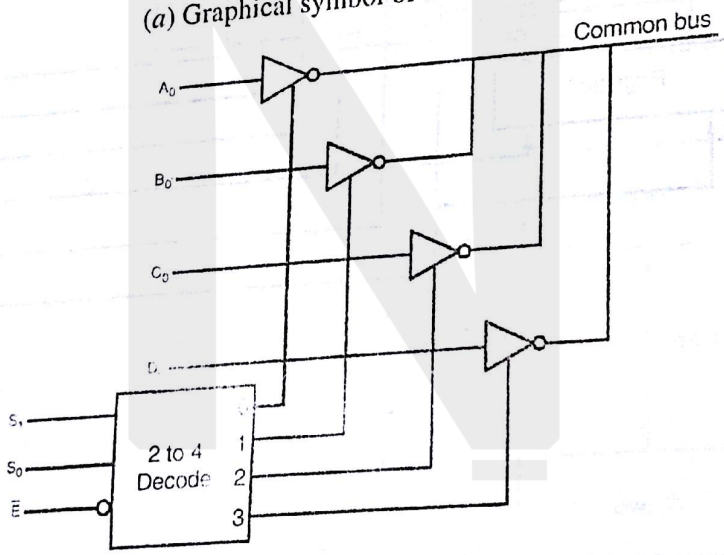
When the select lines are equal to 01, all the MUXs are enabled and the second input places the content of B register to common bus. At next clock pulse, the content of B, being on the bus, is loaded into register C, when the decoder sends the load signal to register C, using $S_1S_0 = 10$.

3.7.1.2. Shared Data Path Using Tristate Buffer

The graphic symbol of tristate buffer is shown in Fig. 3.16(*a*). It has one input and one output controlled by one control input. When the control inpuyt is 1, the buffer output is enabled and the buffer allows to transfer the data from input to output terminal through them. When the control input is equal to 0, the buffer output is disabled and it does not allow transfer of data through it.



(*a*) Graphical symbol of tristate buffer.



(*b*) Shared data path using tristate buffer.

Fig. 3.16.

When more switches are required for sharing the data path line, a shared data path is formed by using tristate buffer in place of multiplexer. Figure 3.16(*b*) shows the 4 bit shared data path using tristate buffer. The output of four buffers are connected together to form a single/common bus line. The connected buffers must be controlled by the control input of each buffer so that only one output of the four tristate buffers has access to the bus line. While all the other buffers act on high impedance (open circuited) state. A decoder is needed to activate the buffer to generate the control signals. When the enable input of decoder is 0, any one of its buffers are enabled based on the selection line of the decoder. When the enable input is high all of its four buffers are disabled.

3.7.2. Memory Transfer

The basic operation of memory transfer is Fetch (or Read) and Store (or Write). The Read operation transfers a copy of the contents from memory location to CPU. The word in memory remains unchaned. The store operation transfers the word information from the CPU to a specific memory location, destroying previous contents of that location. The memory word is symbolised by the letter M. The required information is selected from the memory lcoation by address, it is stored in the address register symbolised by AR. The data is transferred to another register called data register, symbolised by DR. A simple read operation is stated as follows :

$$\text{Read} : DR \leftarrow M[AR]$$

This **read** operation transfer the information into data register (DR) from the memory word M selected by the address in address register (AR).

The **write** operation transfers the content of data register to a memory word M selected by the address. The write operation is stated as follows :

$$\text{Write} : M[AR] \leftarrow B$$

The contents of register B is transferred to selected memory location. When it receives the control signal which is high.

3.7.3. Bus Organization

A bidirectional bus used to carry data between two units is data bus. A unidirectional bus used to carry memory addresses is called memory bus. The way in which different bus are connected to form common bus, so that CPU, memory and I/O devices can used common bus when required is called bus organization.

A basic computer consists of a memory unit, a control unit and registers. There must be a path that can be used to transfer information between memory and registers or among registers. A most efficient way of transferring information from source to destination in a system with many registers is to use a common bus. Fig. 3.17 shows the connection of eight registers and memory unit of 4096 × 16 to a common bus system. The eight registers are Address Register (AR), Program Counter (PC), Data Register (DR), Accumulator (AC), Instruction register (IR), Temporary Register (TR), Input Register (INPR) and Output Register (OUTR). The Fig. 3.10 consist of 16-bit common bus.

The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined by the binary value of the selection lines S_2, S_1 and S_0 is shown in Table 3.7. The numbers along each output lines shows the decimal equivalent of the required binary selection.

Table 3.7. Function Table.

S_2	S_1	S_0	*Register*
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

For example, when $S_2S_1S_0 = 011$, the 16-bit outputs of DR are placed on the bus lines.

The lines from the common-bus are connected to the input of each registers and the data inputs of the memory. The particular register whose LD (Load) input is enabled receives the data from the bus. The memory receives the data from the bus when its write input is enabled.

The memory places its output onto the bus when the read input is activated and $S_2S_1S_0 = 111$.

The registers DR, AC, IR, TR are of 16-bits Two Register PC and AR have 12-bits since they store address.

When the contents of AR and PC are placed on the bus the four most-significant bits are set to 0's. When AR and PC receives the data from the bus only 12 least significant bits are transferred to the register.

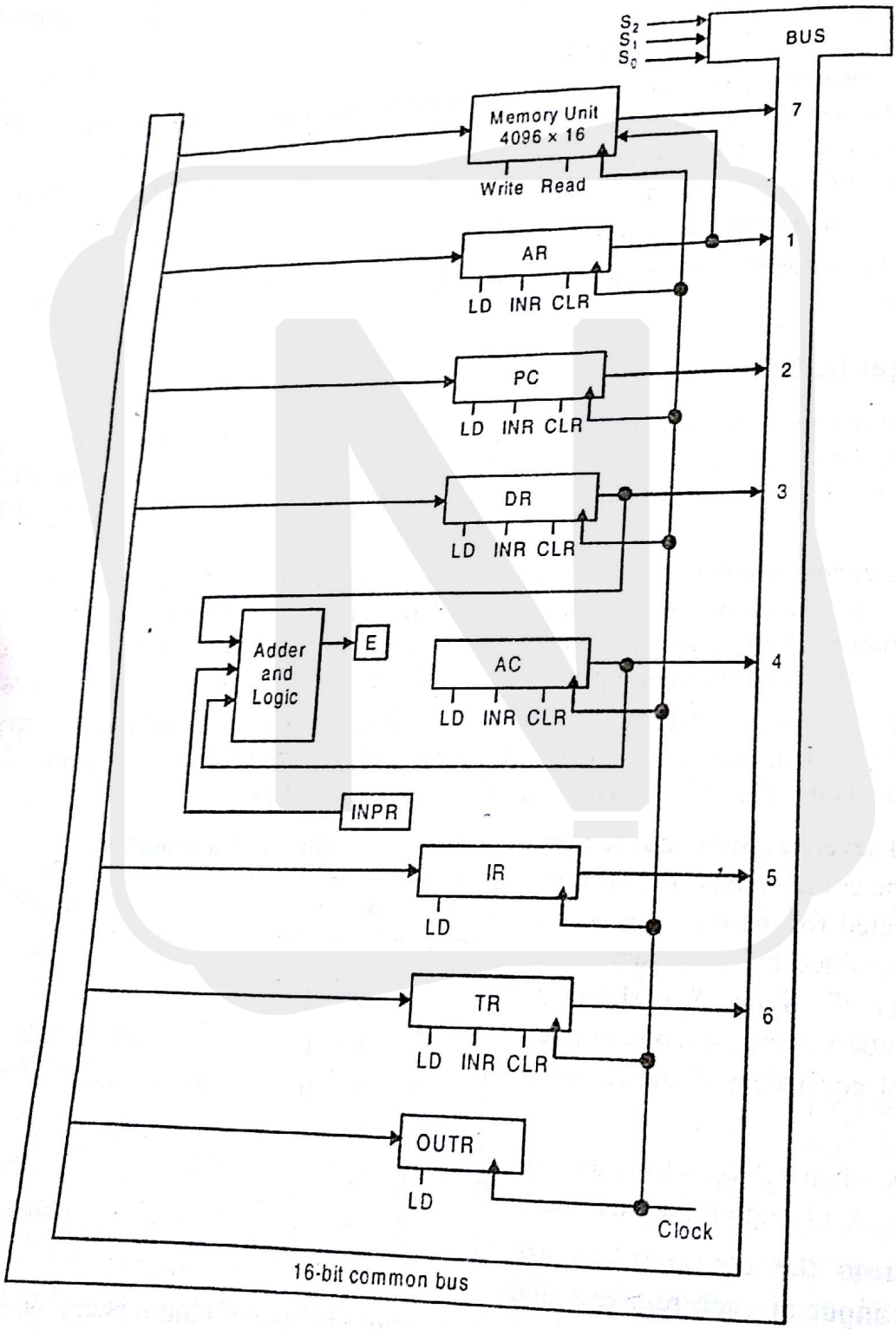


Fig. 3.17. Common Bus Organization.

The input register INPR and OUTR have 8 bits hence they communicate only with the 8 least significant bits in the bus. INPR is connected to bus to provide information but OUTR is connected to bus only to receive information from the bus. INPR receives a character from I/O device which is transferred to AC and OUTR receives a character from AC and delivers to an output device. No transfer is there from OUTR to any of the other registers.

The 16-bit common bus receives information from six registers and the memory unit. Also, the 16-bit common bus are connected to the inputs of six registers and the memory unit. The five registers have three control signals LD (Load), INR (increment) and CLR (clear). Two registers have only LD (load) control signal connected to the common bus. AR is also connected to the memory address. Thus, AR always specify the memory address. During a memory write operation the content of any register can be specified for the memory data and similarly during memory read operation any register except AC receives the memory data. The 16-bit AC receives inputs from adder and logic circuit which receives input from three registers. These three registers are 16-bit AC, 16 bit data register DR and 8-bit inputs come from input register INPR. The inputs from DR and AC are used for arithmetic and logic micro-operations. The function table shown in Table 3.8 shows the binary value of selection line $S_2S_1S_0$ that selects one of the register.

For example, in order to transfer contents of PC to Address Register, the computer requires the following instructions :

(1) Set the selection variables $S_2S_1S_0 = 010$

(2) Transfer the contents of PC to the bus

(3) Enable LD input of AR

(4) Transfer contents of bus into AR

3.7.4. Bus Architecture

A group of wires connecting two or more devices and providing a path to perform communication is called bus. A bus that connects major computer components/modules (CPU, Memory, I/O) is called a system bus. These system buses are separated into three functional groups.

1. Data Bus
2. Address bus
3. Control bus.

Data Bus

The data bus consists of 8, 16, 32 or more parallel lines. The data bus lines are bidirectional. This means that CPU can read data on these lines from memory or from a port as well as data out on these lines to a memory location or to a port.

Address Bus

It is a unidirectional bus. The address bus consists of 16, 20, 24 or more parallel lines. The CPU sends out the address of the memory location or I/O port that is to be written or read from by using this address bus.

Control Bus

Control lines regulate the activity on the bus. The CPU sends signals on the control bus to enable the outputs of addressed memory device or port device.

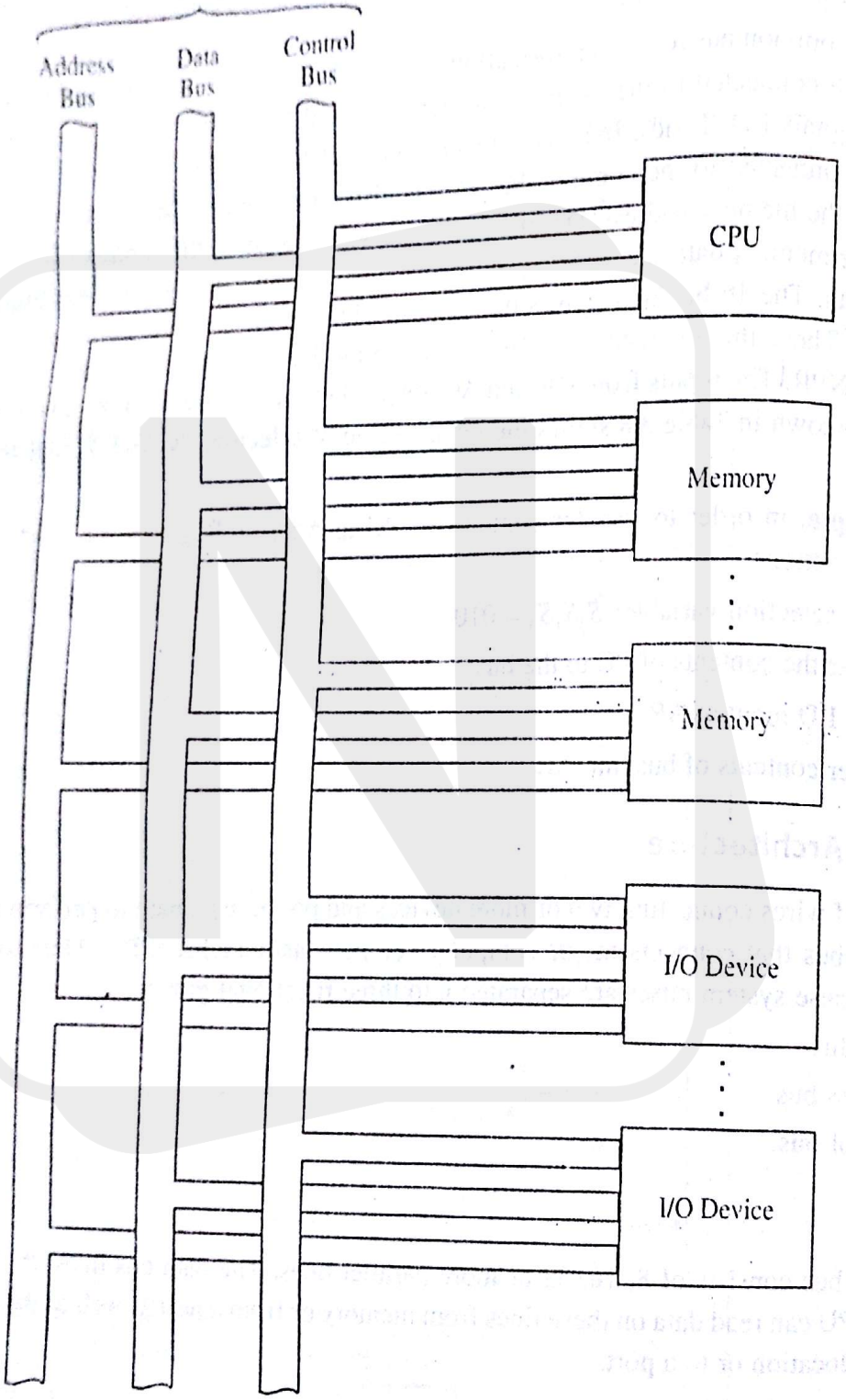


Fig. 3.18. Bus interconnection scheme.

Figure 3.18 shows the bus interconnection scheme. The complexity of bus control logic depends on the amount of translation needed between the system bus and CPU, the timing requirements, whether or not interrupt management is included and size of the overall system. For smaller systems, control signals of the CPU can be used directly to reduce the complexity. But large systems with several interfaces would

need bus driver and receiver circuits connected to the bus in order to maintain adequate signal quality. In most of the processors, multiplexed address and data buses are used to reduce the number of pins. During the first part of the bus cycle, address is present on this bus, afterwards, the same bus is used for data transfer purpose. So latches are required to hold the address sent by the CPU initially.

IMPORTANT NOTES

- **Data Bus:** It is the bidirectional bus used to carry the data between the system units.
- **Address Bus:** It is a unidirectional bus. Its function to select the particular device (or) memory location etc.
- **Control Bus:** It is also unidirectional bus. Its function is to control the units based on the request.

3.7.5. Bus Arbitration

Amongst several masters and slave units are connected to a shared bus, it may happen that more than one master or slave units will request access to the bus at a same time. In such situations bus access is give to the master having highest priority. A mechanism which decides the selection of current master to access bus is known as bus arbitration. Three different mechanisms are commonly used for this.

1. Daisy chaining
2. Parallel arbitration
3. Independent requesting

IMPORTANT NOTES

- **Bus Arbitration:** A mechanism which decides the selection of current master to access bus is known as bus arbitration.

3.7.5.1. Daisy Chaining

The system connection for daisy chaining method are shown in Fig. 3.23. Daisy chaining method is cheaper and simple method. All masters make use of the same line for bus request. The bus grant signal serially propagates through each master until it encounters the first one that is requesting access to the bus. This master blocks the propagation of the bus grant signal, activates the busy line and gains control of the bus. Therefore, any other requesting module will not receive the grant signal and hence cannot get the bus access.

Advantages

1. It is a simple and cheapter method.
2. It requires the least number of lines and this number is independent of the number of masters in the system.

Disadvantages

1. The propagation time delay of bus grant signal is proportional to the number of masters in the system. This makes arbitration time slow and hence limits the number of masters in the system.
2. The priority of the master is fixed by the physical location of master.
3. Failure of any one master causes the whole system to fail.

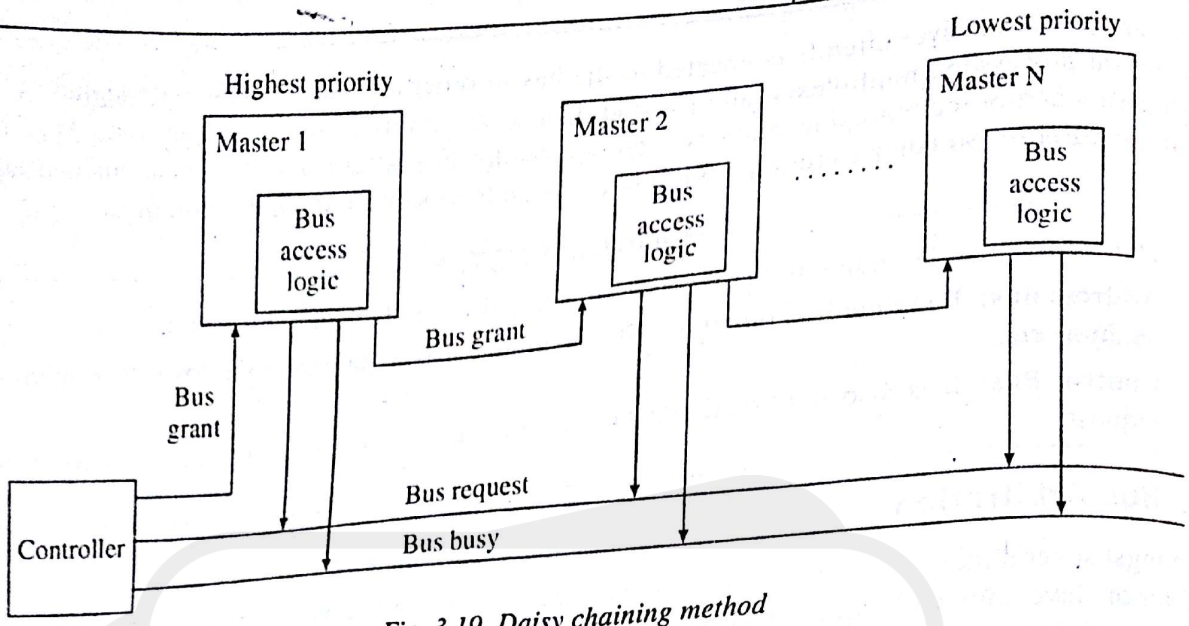


Fig. 3.19. Daisy chaining method

3.7.5.2. Parallel Arbitration

Figure 3.20 shows the parallel arbitration logic, it consists of priority encoder and a decoder. In this mechanism, each bus arbiter has a bus request output line and a bus acknowledges input line. Each arbiter enables the request line when its device (processor) is requesting access to the system bus.

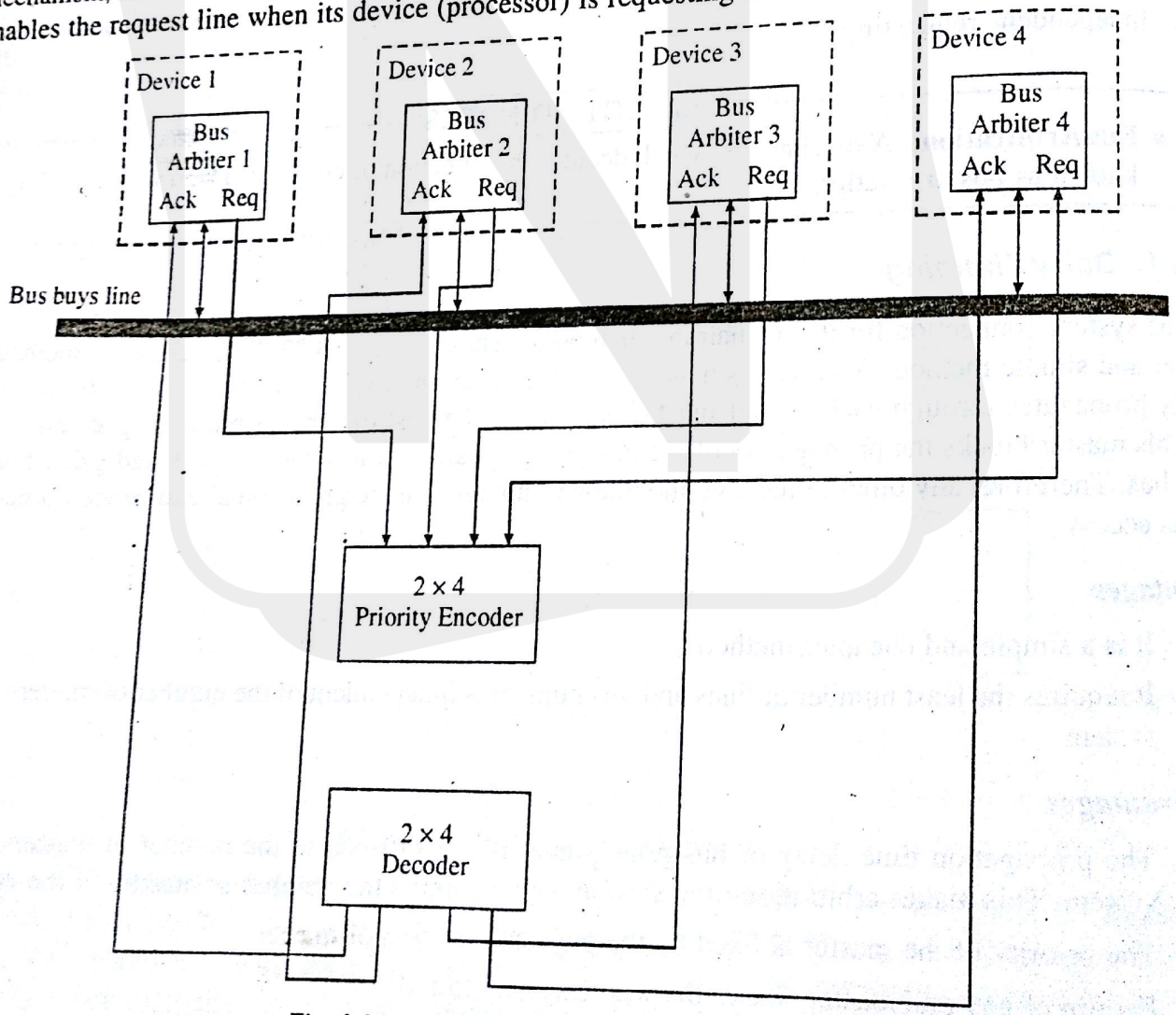


Fig. 3.20. Parallel arbiter for four devices.

Each arbiter bus request output line is connected to the input of priority encoder. The output of the encoder generates a 2-bit code which represents the highest-priority unit among those requesting the bus. This 2-bit code is given to the input of 2 × 4 decoder which enables the proper acknowledge line to grant bus access to the highest-priority unit. A device is utilised in the system only after it receives the acknowledged signal.

Advantages

1. The priority can be changed by altering the priority sequence stored in the controller.
2. If one module fails entire system does not fail.

3.7.5.3. Independent Priority

Figure 3.21 shows the system connections for the independent priority scheme. In this scheme each master has a separate pair of bus request and bus grant lines and each pair has a priority assigned to it. The built in priority encoder within the controller selects the highest priority and asserts the corresponding bus grant signal.

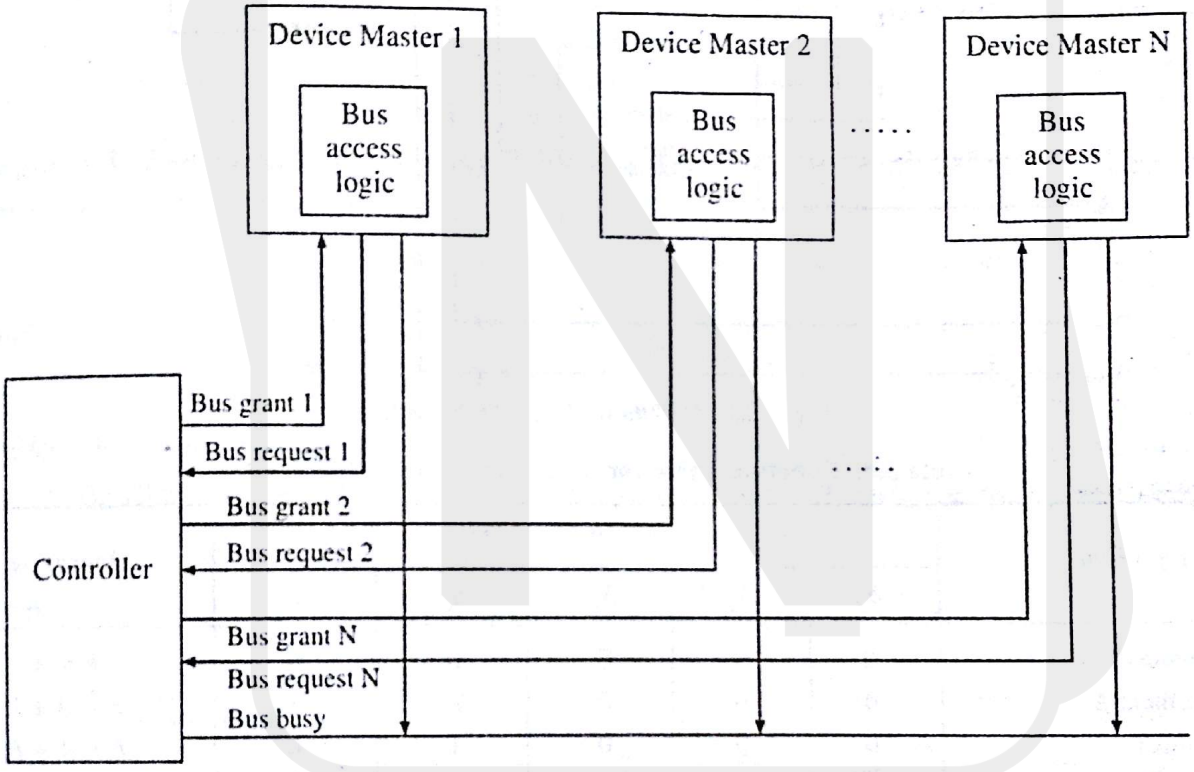


Fig. 3.21. Independent priority method.

3.8. DESIGN OF AN ARITHMETIC LOGIC SHIFT UNIT

In the earlier sections, we discussed the logic circuits for arithmetic operations and logic operations. These circuits can be combined to form one ALU with common selection variables. Figure 3.22 shows one stage of a complete arithmetic logic shift unit. The arithmetic, logic and shift operations are performed on the contents of two registers A and B. A_i stands for the ith bit of register A and B_i stands for the ith bit of register B. Inputs A_i and B_i are applied to both the arithmetic and shift units.

S_1 and S_0 selects a particular microoperation. A 4 × 1 multiplexer choose between the arithmetic output D_i and logic output E_i. The data input of the multiplexer are selected by the selection lines S_3

and S_2. A_{i-1} and A_{i+1} are the two data inputs of multiplexer for shift-right operation and shift-left operation, respectively. The output carry C_{i+1} is connected to the input carry C_i of the next stage in sequence.

Table 3.8 lists eight arithmetic operation, four logic operation and two shift operations. S_3, S_2, S_1, S_0 and C_{in} selects a particular operation. C_{in} is used to select the arithmetic operation only.

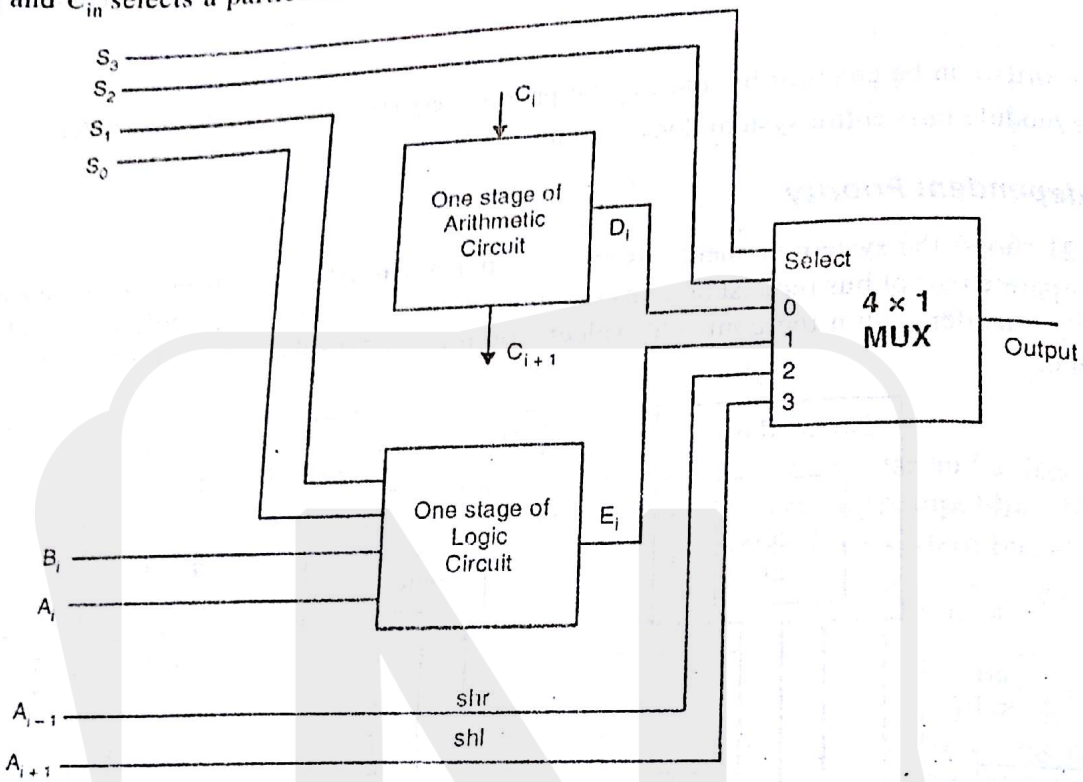


Fig. 3.22. Arithmetic Logic Shift Unit.

Table 3.8. Function Table for Arithmetic Logic Shift Unit.

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

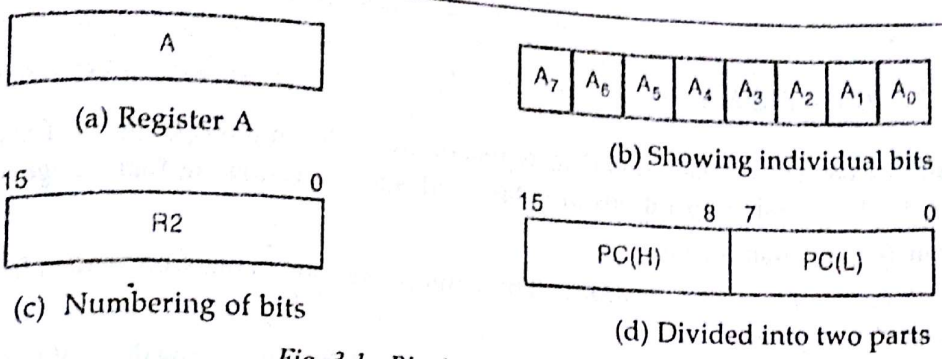


Fig. 3.1. Block Diagram of Register.

The symbolic notation used to describe the microoperation transfer among registers is called a *register transfer language*. This is a way of expressing the different microoperations sequence in the symbolic form among registers of a computer.

For example, the statement.

$$R2 \longleftarrow R1$$

simply denotes that the content of register $R1$ is to be transferred into register $R2$. After this transfer operation takes place the previous content of $R2$ will be replaced by the content of $R1$ but the content of register $R1$ does not change after the transfer. $R1$ is the source register and $R2$ is the destination register. If we want that the transfer can take place only when certain control condition satisfies, this can be shown by means of an if then statement as

$$\text{if } (P = 1) \text{ then } (R2 \leftarrow R1)$$

where P is a control signal, generated in the control section. Also, some time we use control function to separate the control variables from the register transfer operation. A control function is a Boolean variable that is equal to 1 or 0. Thus, the above statement is equivalent as

$$P : R2 \longleftarrow R1$$

where colon is used to terminate the control condition.

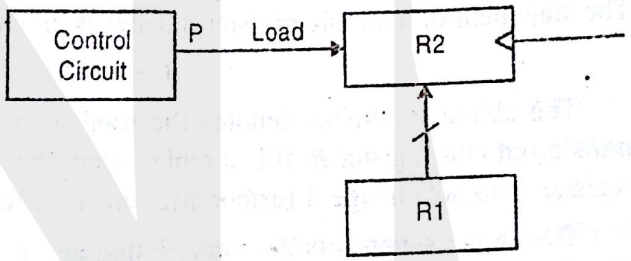


Fig. 3.2. Hardware Implementation of $P : R2 \longleftarrow R1$

Every statement written in a register transfer notation implies a hardware implementation. The block diagram that depicts the transfer from $R1$ to $R2$ is shown in Fig. 3.2.

The basic symbols used for the register transfer notation is shown in Table 3.1.

Table 3.1. Basic Symbols for Register-Transfer Logic.

Symbol	*Description*	*Examples*
Letters (and numerals)	Denotes a register	$R1$, $R2$, PC, IR
Parentheses ( )	Denotes a part of register	PC (0 – 7), PC (L)
Arrow ←	Denotes a transfer of information	$R2 \leftarrow R1$
Comma,	Separates two Microoperation	$R2 \leftarrow R1, R3 \leftarrow R1$
Subscript	Denotes a bit of a register	A_2, B_6
Colon :	Terminates a control function	$x' T_0$:
Square brackets [ ]	Specifies an address for memory transfer	MBR ← M[MAR]

3.2. MICROOPERATIONS

The operations executed on data stored in registers are called microoperations, for example shift, count, clear, and load. The microoperations are classified into the following four categories:

1. **Data transfer microoperation.**
   (*a*) *Register transfer microoperations.* These micro operations transfer the data from one register to another register.
   (*b*) *Memory transfer microoperations.* These microoperations transfer data from register to memory or memory to register.
2. **Arithmetic microoperations.** These microoperations perform arithmetic operations on data stored in registers. Addition, subtraction, multiplication, division all are arithmetic microoperations.
3. **Logic microoperations.** These microoperations perform bit manipulation operations on nonnumeric data stored in registers.
4. **Shift microoperations.** These microoperations shift the data in left or right direction stored registers.

3.3. REGISTER/DATA TRANSFER MICROOPERATIONS

Register transfer means that the contents of one register is transferred to another register. The information transfer from one register to another is designated in symbolic form by using the operators. The statement of a simple register transfer is given by

$$B \leftarrow A$$

The above statement denotes the transfer between two registers. The contents of register A are transferred into register B. It is a replacement the contents of B by the contents of A, but the contents of register A do not change A further after this transfer.

The above statements are very simple and it specifies a register transfer of the contents of source register (A is source register for the given statement) to destination register. This transfer does not depend on any control function. However, some transfers depend on some specified conditions. The control function is a Boolean function that can be equal to 1 (logic HIGH) or 0 (logic LOW). The control function is terminated by colon. This colon indicates that the transfer operation by executed by the hardware only when the control function is equal 0 or 1 (depending on the logic level).

For example, the register transfer statement with control function is given below.

$$xy' : A \leftarrow B$$

The control statement is xy' and the register transfer statement is $A \leftarrow B$. The register transfer is based on this control statement xy'. When this control hardware issues control signal 1, the content of register B is loaded into register A. The hardware implementation is shown in Fig. 3.3.

The output of register B is connected to the input of regsiter A, and the number of connecting wires (bus) is equal to the number of bits in the registers. Registers A must have load control input so that it can be enabled when the control function $xy' = 1$. The register B also has some control input (it is not shown in Fig. 3.3). It is assumed that register B has a synchronized clock pulse input, the content of register B is transferred to register A when the synchronised clock pulse input, the conten of register B is transferred to register A when the synchronized clock plus is HIGH.

The content of register B appears at the input of register A, but the content is not loaded to register A. The register A is loaded when the control function is HIGH.

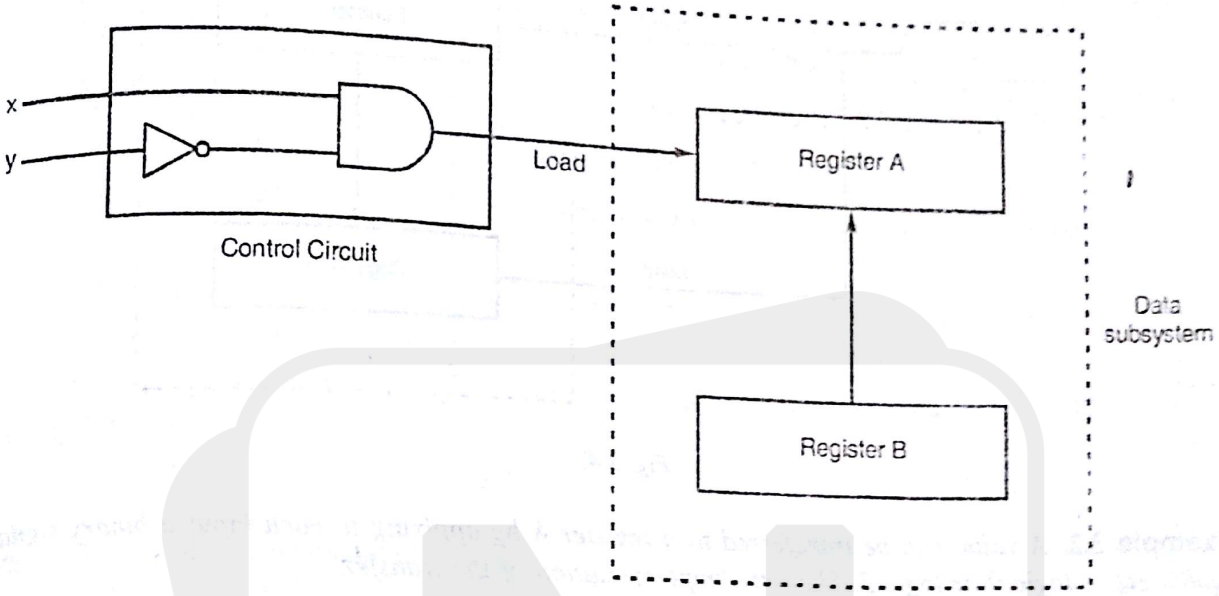


Fig. 3.3. Hardware implementation of data transfer between A and B.

Example 3.1. *Show the block diagram that executes the statement*

$$xT_3 : A \leftarrow B,\ B \leftarrow A$$

Solution. Control function is xT_3

Register transfer statements are $A \leftarrow B$, $B \leftarrow A$.

We have the two statements for register transfer, so we cannot perform the transfer function simultaneously. The control function is xT_3. It has two possible values 0 for one register transfer statement and 1 for another register transfer statement.

The function table is given below, it summarises the register transfer with control function.

Function Table

Control function	*Register transfer*	*Comments*
0	$A \leftarrow B$	Load register A
1	$B \leftarrow A$	Load register B

Hardware Implementation

After receiving the load signal 1 from the control subsystem the contents of register A is loaded register B. After receiving the load 0 signal from control sub subsystem the contents of register B is loaded into register A.

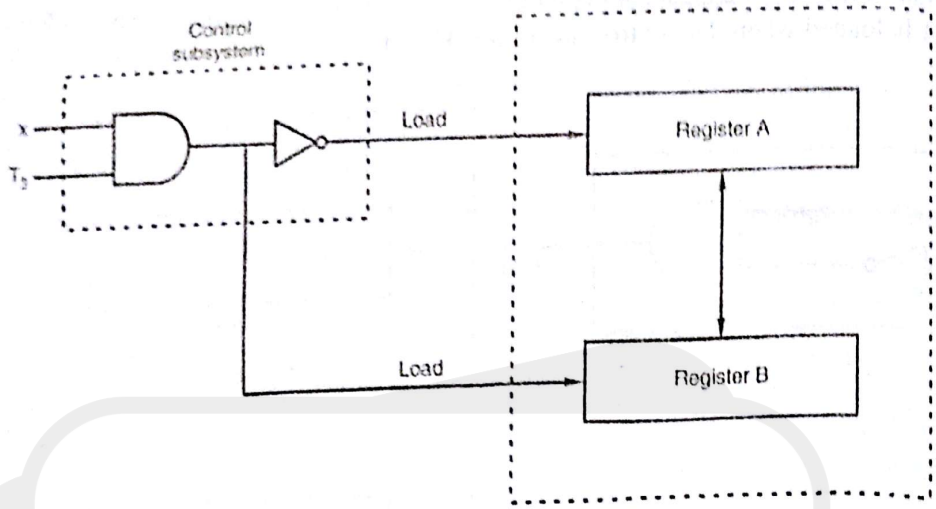


Fig. 3.4.

Example 3.2. *A value can be transferred to a register A by applying to each input a binary signal equivalent to logic 0 or logic 1. Show the implementation of the transfer.*

$$X : A \leftarrow 10010110$$

Solution. The register is loaded when the control function X equals 1. Each bit is stored in each flip-flop of the register.

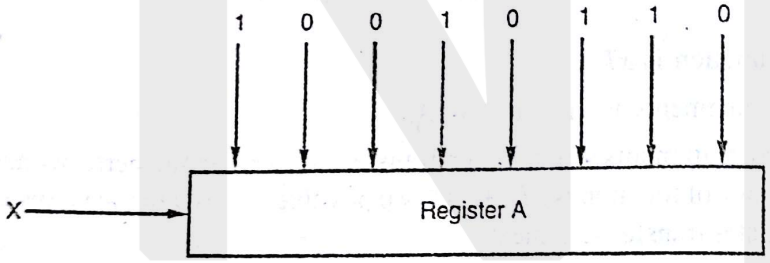


Fig. 3.5.

3.4. ARITHMETIC MICROOPERATIONS

The basic arithmetic microoperations are addition, subtraction, increment, decrement and shift.

Let the arithmetic microoperations defined by the statement

$$R3 \longleftarrow R1 + R2$$

specifies an add microoperation. It states that add the content of register $R1$ and that of register $R2$ and stored the result in register $R3$. Thus, to implement this statement with hardware we need three registers and the digital component that performs the addition operation. Other basic arithmetic microoperations are listed in the Table 3.2.

Arithmetic microoperations are those microoperations that are used to perform arithmetic operations.

Table 3.2. Arithmetic Microoperations.

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

Subtraction is basically implemented through complementation and addition and can be specified by the statement

$$R \leftarrow A + \overline{B} + 1$$

where $\overline{B}$ is the 1's complement of B. When we add 1 to the 1's complement it will give 2's complement of B and adding A to the 2's complement of B gives A minus B *i.e.*, $(A - B)$. The increment and decrement microoperation are implemented with the help of combinational circuit or with a binary up-down counter. They are symbolized by plus-one or minus-one operation executed on the contents of a register. The multiplication operation and division are valid arithmetic operations. Multiplication operation is basically implemented with a sequence of add and shift microoperations and division is implemented with a sequence of subtract and shift microoperations.

The block diagram of various arithmetic circuits is discussed in chapter 2 like Half Adder, Full Adder, Adder-Subtractor.

Arithmetic Circuit

The basic arithmetic microoperation listed in Table 3.2 can be implemented by means of a single arithmetic circuit where the basic component of the circuit is parallel adder. Figure 3.6 shows the logic circuit of 4-bit arithmetic circuit.

The arithmetic circuit consists of four full adder that makes the 4-bit adder and four multiplexers for choosing different operations. A and B are the two 4-bit inputs and D is the output. S_1 and S_0 are two selection lines. C_{in} is the input carry connected to the carry input of the full adder in the least significant position and other carries are connected to the next full adder with C_{out} the output carry. The four inputs from A goes directly to the X inputs of the binary adder and four inputs from B are connected to the data inputs of the multiplexers which also receives the complement of B. Other two data inputs of multiplexers are connected to logic 0 and logic 1 respectively.

The output of the circuit is calculated from the following arithmetic sum :

$$D = A + Y + C_{in}$$

where A is the 4-bit input at the X inputs and Y is the 4-bit binary number at the Y input of the circuit. With the help of selection lines S_1 and S_0 and by making input carry C_{in} equal to 0 or 1, the arithmetic microoperation listed in Table 3.3 can be generated.

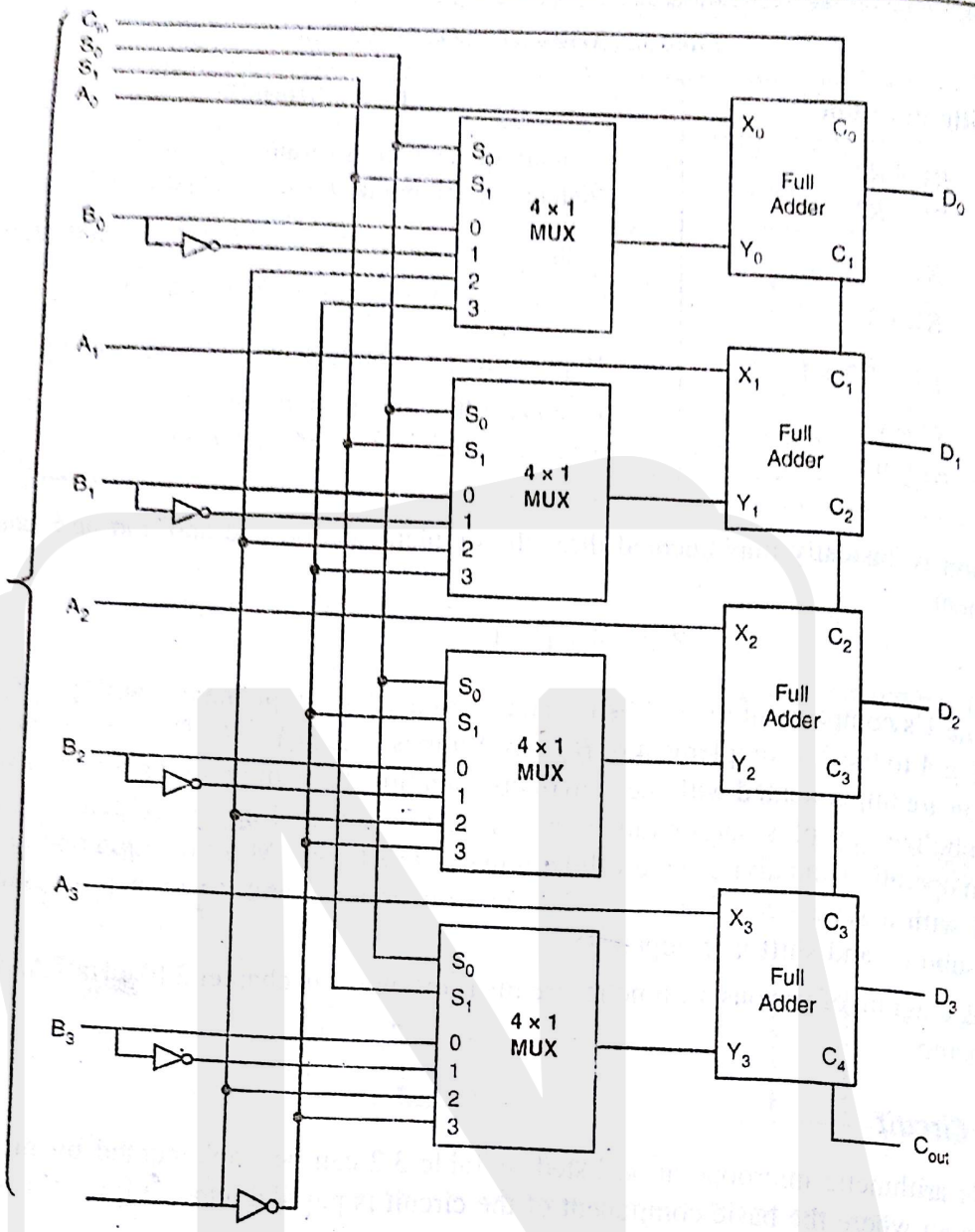


Fig. 3.6. 4-bit Arithmetic Circuit.

Table 3.3. Arithmetic Circuit Function Table.

Select			Input	Output	Microoperation
S_1	S_0	C_{in}	Y	$D = X + Y + C_{in}$	
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

Case 1 : When $S_1S_0 = 00$: In this case, the value of B is applied to the Y inputs of the adder. If $C_{in} = 1$, output $D = A + B + 1$ and if $C_{in} = 0$, output $D = A + B$, are the addition microoperation with or without adding the carry input C_{in}.

Case 2 : When $S_1S_0 = 01$: In this case the complement of B is applied to the Y inputs. If $C_{in} = 1$, the output $D = A + \overline{B} + 1$, which produces A plus the 2's complement of B and hence the subtraction $A - B$. If $C_{in} = 0$, the output $D = A + \overline{B}$, which is equivalent to $A - B - 1$ *i.e.* subtract with borrow.

Case 3 : When $S_1S_0 = 10$: In this case, the inputs from B are neglected, and 0's are inputed into the Y inputs. If $C_{in} = 1$, $D = A + 1$, *i.e.* the value of A is incremented by 1 and when $C_{in} = 0$, $D = A$ and hence we have a direct transfer from input A to output D.

Case 4 : When $S_1S_0 = 11$: In this case, all 1's are inserted into the Y inputs. A number with all 1's is basically equal to the 2's complement of 1. Thus, if $C_{in} = 1$, the output $D = A - 1 + 1 = A$, and hence a direct transfer from input A to output D. When $C_{in} = 0$, $D = A - 1$, *i.e.* the value of A is decremented by 1.

Ripple Carry Adder or Parallel Adder

As we know that a single full-adder is capable of adding three one bit binary numbers or two one bit numbers and one previous carry, in order to add binary numbers with more than one bit, additional full adders must be emploted. A four bit parallel adder can be constructed using four full adders as shown in Fig. 3.7. These four full adders are connected in cascade *i.e.*, the carry output of each adder is connected to the carry input of the next higher-order adder. An n-bit parallel adder is constructed using 'n' number of full adders.

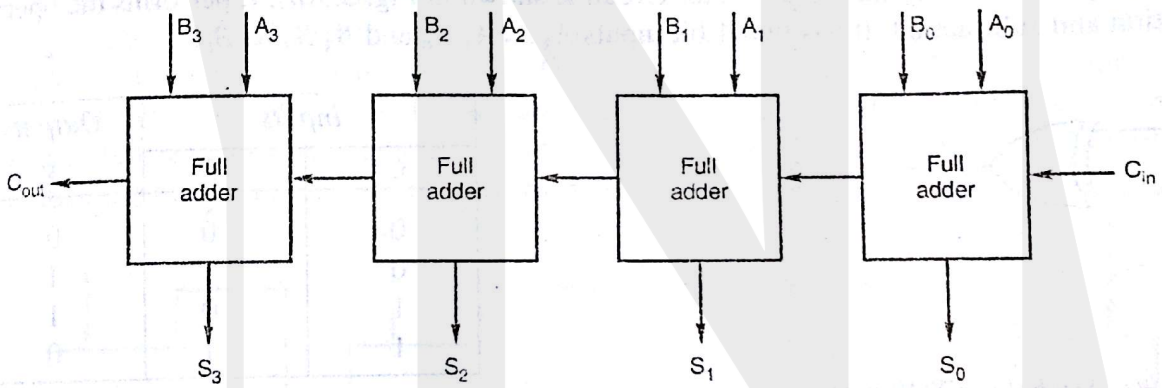


Fig. 3.7. Block diagram of parallel adder.

The carry from the right most position (LSB) has to propagate through all the bit positions. If t_d is the time delay for each full-adder stage. For n-bit parallel adder has the maximum propogation delay is nt_d. This is the major drawback of parallel adder.

Serial Adder

The speed of the adder is an important design parameter of adder circuits. If the speed is not important, then the cost effective option is to use a serial adder, in which bits are added one pair at a time. The serial adder method uses only one full adder circuit and a storage device (flip-flop) to hold the generated output carry. Let $A = A_{n-1}\ A_{n-2} \ldots A_0$ and $B = B_{n-1}\ B_{n-2} \ldots B_0$ be two unsigned numbers that have to be added to produce sum $S_{n-1}\ S_{n-2} \ldots S_0$. The two number A and B are stored in the register A and register B respectively. The pair of bits in A_0 and B_0 (which are stored in right shift register) are transferred serially one at a time, through the single full adder to produce a sum and carry and this carry is stored in the flip-flop. The stored output carry from one pair of bits is used as an input carry for the next pair of bits as

shown in Fig. 3.8. The addition of two n-bit numbers starts from the least significant bits and progresses step by step to the most significant bits in different time sequences. Initially the D flip-flop is cleared

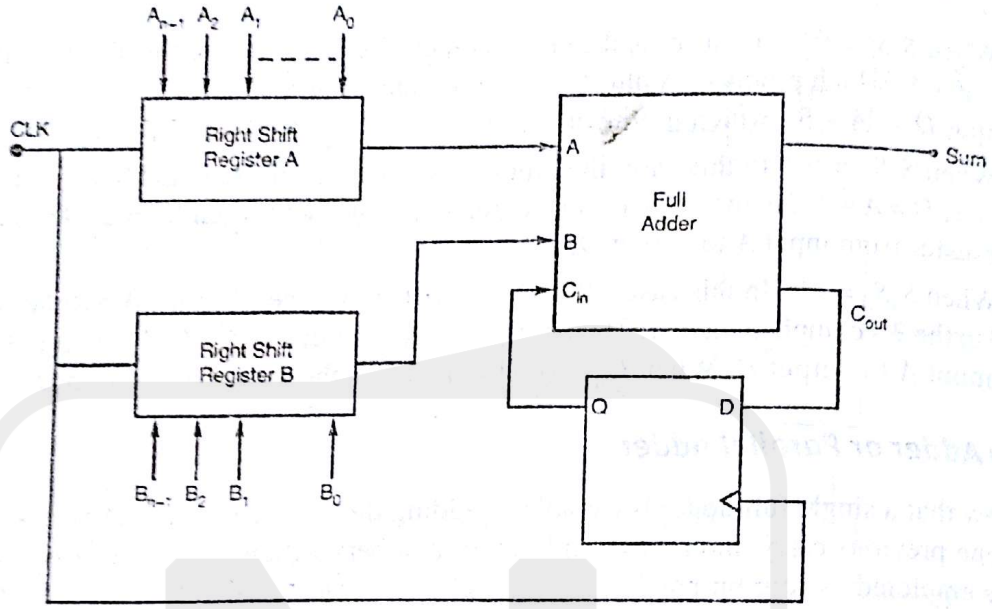
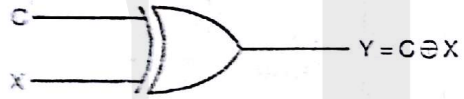


Fig. 3.8. Block diagram of Serial adder.

4-bit Parallel Adder/Subtractor

The 4-bit parallel binary adder/subtractor circuit is shown in Fig. 3.9(*a*). It performs the operation of both addition and subtraction. It has two 4-bit inputs $A_3\ A_2\ A_1\ A_0$ and $B_3\ B_2\ B_1\ B_0$.



Inputs		Outputs
C	X	Y
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 3.9(a) Symbol of EXOR gate.

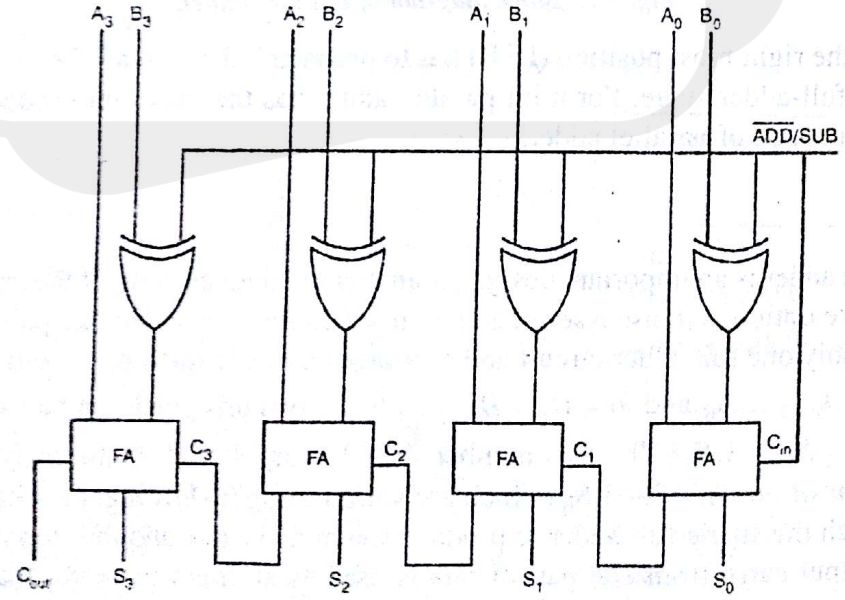


Fig. 3.9(b) A bit parallel binary adder/subtractor

We can not perform the two operations simultaneously, so we need one control signal to perform this operation. The $\overline{ADD}/SUB$ control line, connected with C_{in} of LSB of the full adder, is used to perform the operations of addition and subtraction. The EXOR gates are used as controlled inverters. From the EXOR gate truth table we know that when one of the inputs is LOW the output is of the same value of the other input and when one of the inputs is HIGH the output is the complement of the other input. Fig. 3.9(*a*) shows the symbol EXOR gate and above table shows the truth table.

If $C = 0$, the input variable X, either 0 or 1 wil be complemented and transferred to output. By using these EXOR gate properties, we use this gate in the 4 bit adder/subtractor circuit which is shown in Fig. 3.9(*b*).

Case 1 : $\overline{ADD}/SUB = 1$. Now the controlled inverter (EXOR gate) produces the 1's complement of $B_3\ B_2\ B_1\ B_0$. Since 1 is given to C_{in} of the LSB bit of the adder, it is added to the complemented output of EXOR gate output, it is equal to 2's complement of $B_3\ B_2\ B_1\ B_0$. The 2's complemented $B_3\ B_2\ B_1\ B_0$ will be added to $A_3\ A_2\ A_1\ A_0$ to produce the sum, the produced output of $S_3\ S_2\ S_1\ S_0$ is the difference between $A_3\ A_2\ A_1\ A_0$ and $B_3\ B_2\ B_1\ B_0$.

Case 2 : $\overline{ADD}/SUB = 0$. Now the controlled inverter is transferred $B_3\ B_2\ B_1\ B_0$ four bit to full adder, this 4 bit is added with $A_3\ A_2\ A_1\ A_0$ to produce sum and carry.

Binary Incrementor

The increment microoperation adds one to a number in a register. For example, if a 4-bit number 1110, it will be 1111 after it incremented.

The diagram of a 4-bit combinational circuit incrementer is shown in Fig. 3.10. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder. The circuit receives the four bits from A_0 through A_3, adds one to it, and generates the incremented output in S_0 through S_3. The output carry C_4 will be 1 only after incrementing binary 1111. This also causes outputs S_0 through S_3 to go to 0.

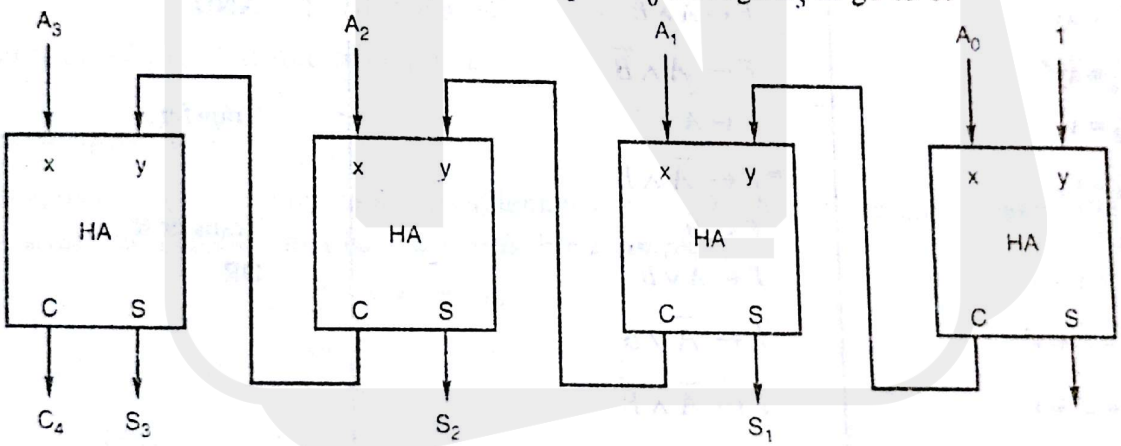


Fig. 3.10. 4-bit binary incrementer

3.5. LOGICAL MICROOPERATIONS

Logic microoperations are those microoperations that are used to perform logical operations, such as AND, OR, NOT etc. The microoperations which perform these operations on individual pairs of bits stored in registers are said to be logic microoperations. These operations consider each bit of the register separately and treat it as a binary variable. The XOR (exclusive OR) operation between the content of two register $R1$ and $R2$ can be represented as

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

The symbol ⊕ represents the XOR operation. Also, special symbols are used for logic microoperations OR, AND and complement operations. The symbol ∨ is used to denote an OR microoperation, and symbol ∧ is used to denote AND microoperation.

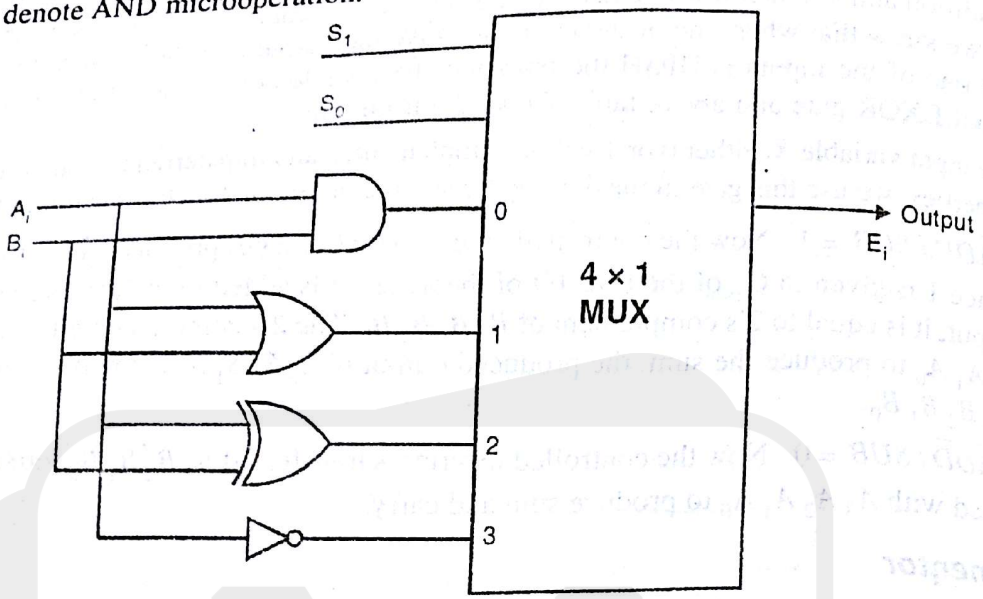


Fig. 3.11. One Stage of Logic Circuit.

There are 16 different logic microoperations that can be performed with two binary variables and are listed below in the Table 3.4.

Table 3.4. Sixteen Logic Microoperations.

Boolean Function	*Logic Microoperation*	*Meaning*
$F_0 = 0$	$F \leftarrow 0$	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
$F_6 = x + y$	$F \leftarrow A \vee B$	OR
$F_7 = x + y'$	$F \leftarrow \overline{A} \vee B$	
$F_8 = x' + y$	$F \leftarrow \overline{A} \wedge B$	
$F_9 = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{10} = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_{11} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{12} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{13} = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_{14} = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

Although there are 16 logic microoperations most of the computers uses only four— AND, OR, XOR, and complement from which all others can be derived. Figure 3.11 depicts the hardware implementation of four basic logic microoperations.

The output of the four gates (AND, OR, XOR, NOT) are applied to the data input of the 4 × 1 multiplexer which will give output according to the function Table 3.5.

Table 3.5. Function Table.

S_1	S_0	*Output*	*Operator*
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

The selection lines S_1S_0 choose one of the data inputs of the multiplexer and directs it to the output. The diagram shows one stage of circuit. For n-bits the diagram must be repeated n times, for $i = 0, 1, 2, ..., n - 1$.

3.5.1. Selective Microoperations

Selective-Set Operation

The *selective-set* operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation :

```
1010   A before
1100   B (logic operand)
----
1110   A after
```

It is similar to Logic-OR microoperation.

Selective-complement

The *selective-complement* operation complements bits in A where there are corresponding 1's in B. It does not affect bit positions that have 0's in B. For example :

```
1010   A before
1100   B (logic operand)
----
0110   A after
```

It is similar to Exclusive-OR microoperation.

Selective-clear

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

```
1010   A before
1100   B (logic operand)
----
0010   A after
```

It is similar to $A \wedge B'$ microoperation.

Mask Operation

The *mask* operation is similar to the selective-clear operation except that the bits of A are cleared only where there are correspoinding 0's in B. The mask operation is an AND microoperation as seen from the following numerical example :

```
1010  A before
1100  B (logic operand)
1000  A after masking
```

It is similar to Logic-AND operation.

3.6. SHIFT MICROOPERATIONS

Shift microoperations are used for serial transfer of data. The shifting of bits of register can be in both direction left or right, during which the first flip-flop receives its binary information from the serial input. When the operation is shift-left, the serial input transfers a bit into the right most position and when the operation is shift-right the bit from the serial input is transferred into the left most position.

Shift microoperation can be classified as :

(*a*) Logical, (*b*) Circular, (*c*) Arithmetic.

Shift microoperations are those microoperations that are used to perform shifting of bits in either left or right direction.

1. Logical Shift

It is the easiest to perform as the bits are simply shifted to the right or to the left. If it is shift-left a 0 bit is transferred at the right most position through serial input and the leftmost bit is lost. Similarly, if it is shift-right the right most bit is lost and a 0 bit is added at the leftmost place.

For example,

$$R1 \leftarrow \text{shl } R1$$
$$R2 \leftarrow \text{shr } R2.$$

shl and shr are the symbols used for shift-left and shift-right microoperations. The above two statements specifies a 1-bit shift to the left of the content of register $R1$ and a 1-bit shift to the right of the content of register $R2$ respectively.

2. Circular Shift

Also known as rotate operation, simply circulates the bits of the register around the two ends. No bit is added or lost. If it is left circular shift, the bits are shifted in left direction and the rightmost bit takes the left most bit position. Similarly, if it is right circular shift, the bits are shift in right direction and the leftmost bit takes the rightmost bit position.

3. Arithmetic Shift

An arithmetic shift is a microoperation that shifts a signed binary number to the left or right. In arithmetic left or arithmetic right shift operations, we need to consider the sign bit also, because arithmetic shift requires that the sign bit (the leftmost bit) remains unchanged. 0 is the sign bit for positive and 1 is sign bit for negative numbers.

The Fig. 3.12 shows a register of n-bits, where bit b_{n-1} in the leftmost position contains the sign bit of a number. b_{n-2} is the most significant bit and b_0 is the least significant bit of number.

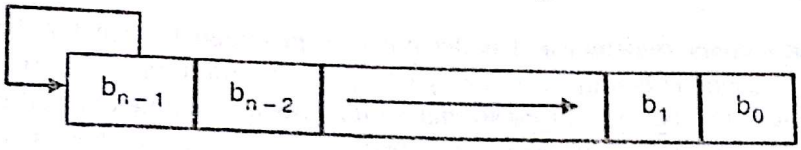


Fig. 3.12.

In case of arithmetic shift-right (ashr) the sign bit remains unchanged and shifts the number, including the sign bit, to the right. The bit b_0 is lost. Thus, b_{n-1} remains unchanged, b_{n-2} receives b_{n-1} and so on.

In case of arithmetic shift-left (ashl), 0 is inserted into bit position b_0 and shifts all other bits to the left. The initial bit b_{n-1} is lost and replaced by bit b_{n-2}. If the bit in b_{n-1} changes the value after the shift a sign reversal occurs.

4. 4-bit Shifter

The 4-bit combinational circuit called shifter constructed with the help of multiplexers is explained in Fig. 3.13. It consists of four data inputs I_0 to I_3 and four data output O_0 to O_3. The two serial inputs, one for shift left and other for shift right are S_L and S_R respectively. The function table for the circuit is given in Table 3.6. When the selection input $S = 0$, the input data are shifted right and when $S = 1$, the input data are shifted left. The function table explains which input goes to each output after the shift operation. In general, a shifter having n data inputs and n data outputs requires n multiplexers.

Table 3.6. Function Table.

Select	*Output*			
S	O_0	O_1	O_2	O_3
0	S_R	I_0	I_1	I_2
1	I_1	I_2	I_3	S_L

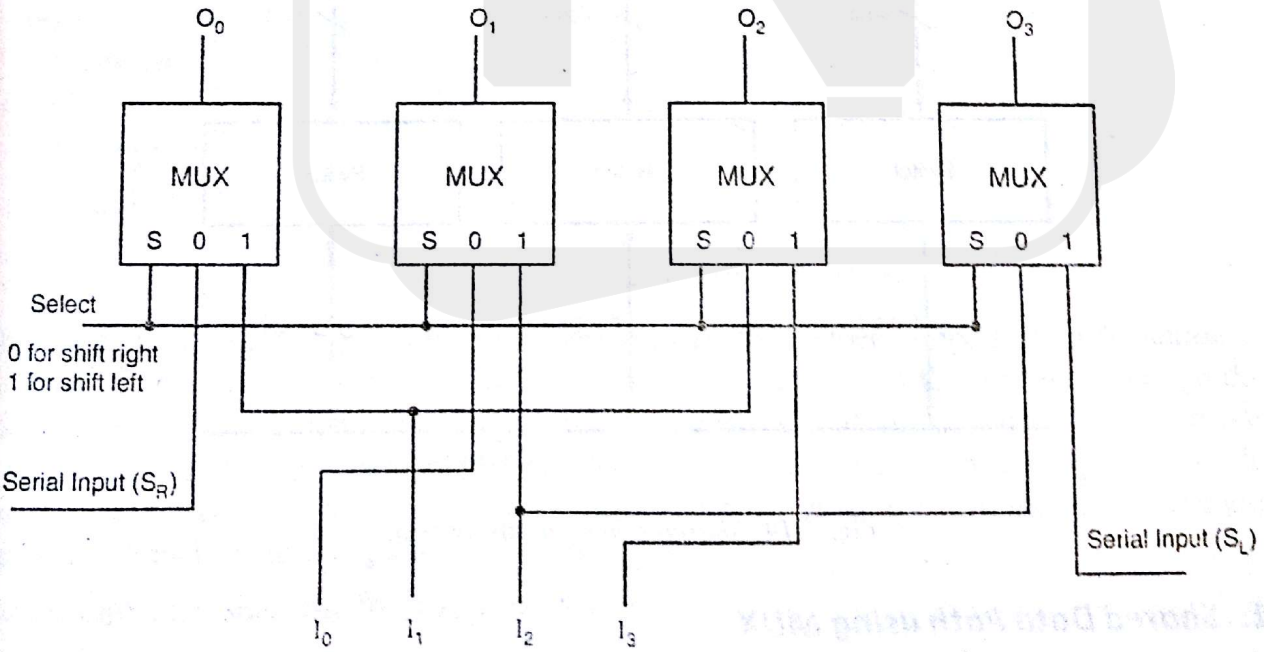


Fig. 3.13. 4-bit Shifter.

SOLVED EXAMPLES

Example 3.3. *Show the block diagram of the hardware (similar to Fig. 3.23) that implements the following register transfer statement :*

$$yT_2: R2 \leftarrow R1,\ R1 \leftarrow R2$$

Solution.

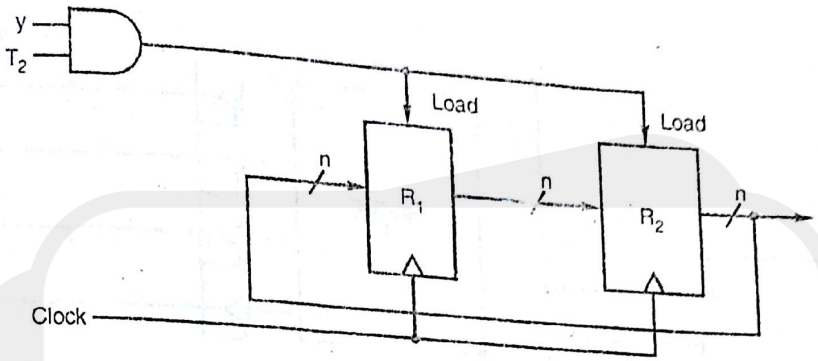


Fig. 3.23

Example 3.4. *Represent the following conditional control statement by two register transfer statements with control functions.*

$$If\ (P = 1)\ then\ (R1 \leftarrow R2)\ else\ if\ (Q = 1)\ then\ (R1 \leftarrow R3)$$

Solution.

$$P : R1 \leftarrow R2$$
$$P'Q : R1 \leftarrow R3$$

Example 3.5. *Show the hardware that implements the following statement. Include the logic gates for the control function and a block diagram for the binary counter with a count enable input*

$$xyT_0 + T_1 + y'T_2:\ AR \leftarrow AR + 1$$

Solution.

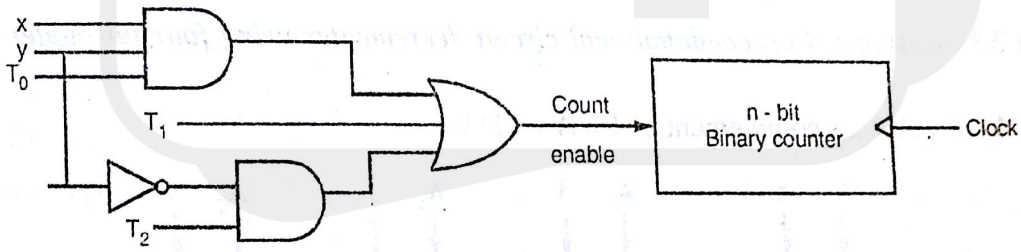


Fig. 3.24

Example 3.6. *A digital computer has a common bus system for 16 registers of 32 bits each. The bus is constructed with multiplexers.*

(a) How many selection inputs are there in each multiplexer ?

(b) What size of multiplexers are needed ?

(c) How many multiplexers are there in the bus ?

Solution.

(*a*) 4 selection lines to select one of 16 registers.

(*b*) 16 × 1 multiplexers.

(*c*) 32 multiplexers, one for each bit of the registers.

Example 3.7. *Draw a diagram of a bus system similar to the one shown in Fig. 3.25, but use three state buffers and a decoder instead of the multiplexers.*

Solution.

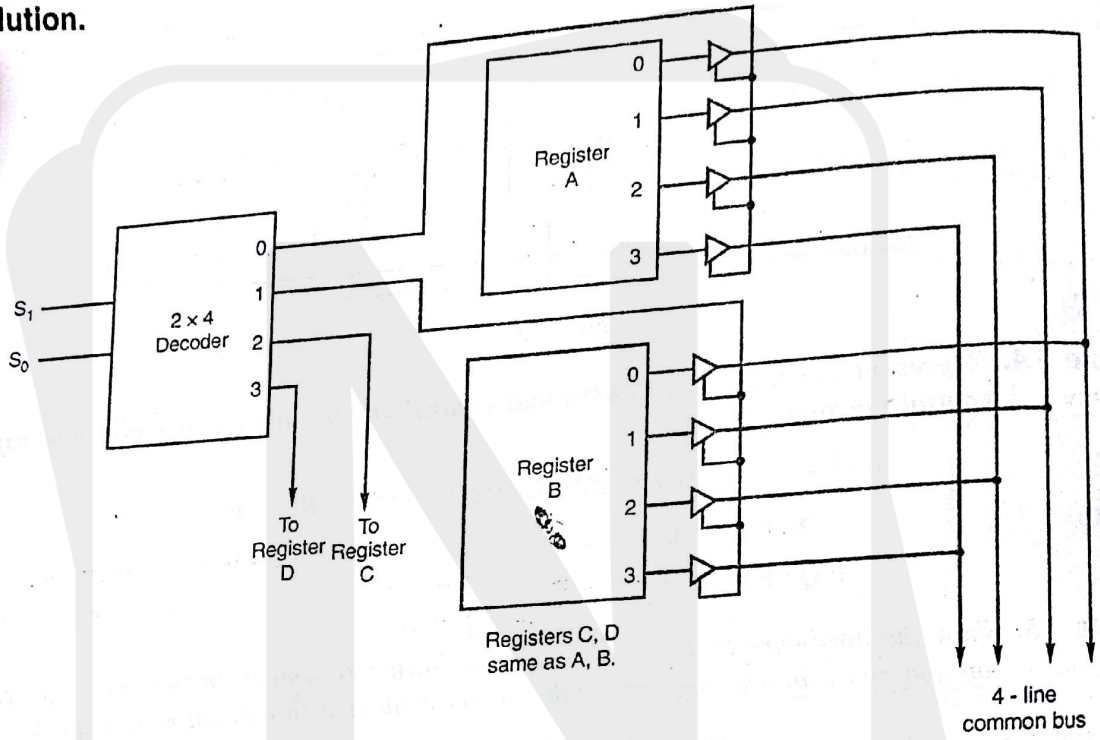


Fig. 3.25

Example 3.8. *Design a 4-bit combinational circuit decrementer using four full-adder circuits.*

Solution. $A - 1 = A + 2$'s complement of $1 = A + 1111$

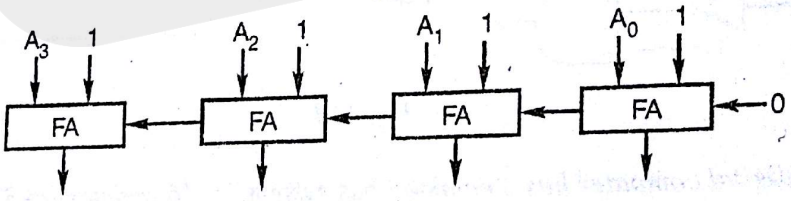


Fig. 3.26

Example 3.9. *Design an arithmetic circuit with one selection variable S and two n-bit data inputs A and B. The circuit generates the following four arithmetic operations in conjunction with the input carry C_{in}. Draw the logic diagram for the first two stages.*

S	Cin = 0	Cin = 1
0	D = A + B (add)	D = A + 1 (increment)
1	D = A – 1 (decrement)	D = D = A + $\overline{B}$ + 1 (subtract)

Solution.

S	C_{in}	X	Y	
0	0	A	B	(A + B)
0	1	A	0	(A + 1)
1	0	A	1	(A – 1)
1	1	A	$\overline{B}$	(A – B)

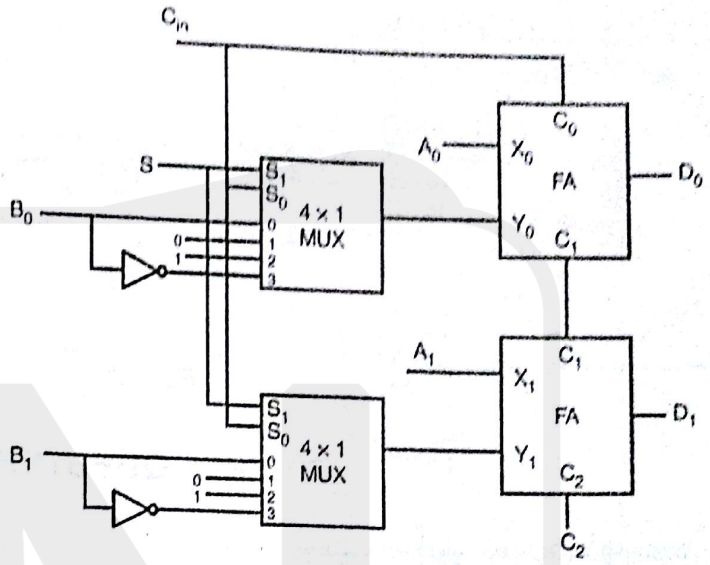


Fig. 3.27

Example 3.10. *An 8-bit register contains the binary value 10011100. What is the register value after an arithmetic shift right ? Starting from the initial number 10011100, determine the register value after an arithmetic shift left, and state whether there is an overflow.*

Solution. R = 10011100

Arithmetic shift right : 11001110

Arithmetic shift left : 00111000 overflow because a negative number changed to positive.

Example 3.11. *The following transfer statements specify a memory. Explain the memory operation in each case.*

(a) $R_2 \leftarrow M[AR]$

(b) $M[AR] \leftarrow R_3$

(c) $R_5 \leftarrow M[R5]$

Solution.

(a) Read memory word specified by address in AR into register R_2.

(b) Write content of register R_3 into the memory word specified by the address *n* AR.

(c) Read memory word specified by the address in R_5 and transfer content to R_5 (destroys previous value).

Example 3.12. *Draw a block diagram for the hardware that implements the following statement*

$$x + yz : AR \leftarrow AR + BR$$

where AR, BR are two n-bit register and x, y, z are conrol variables.

Solution.

(*a*) Read memory word specified by address in AR into register R_2.

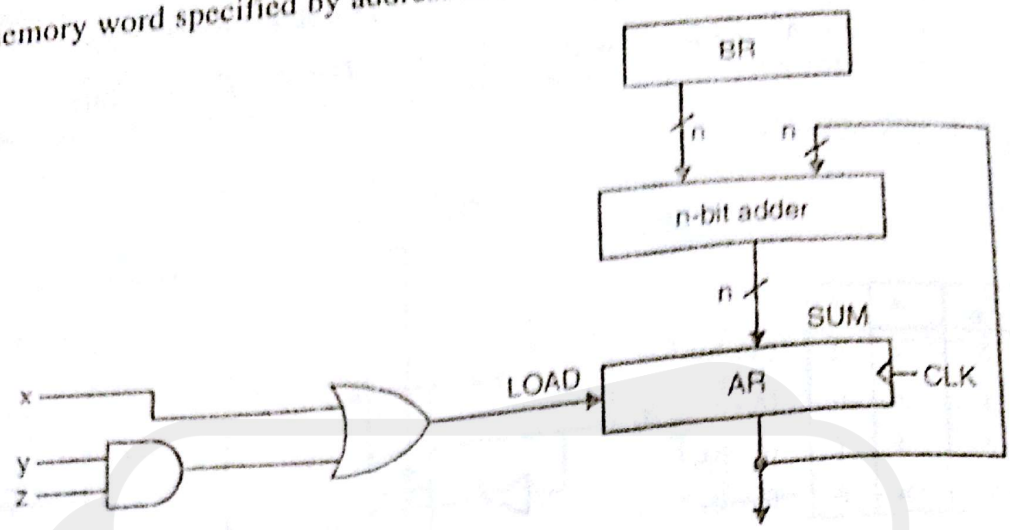


Fig. 3.28.

REVIEW QUESTIONS

1. Write short note on micro-operation.
2. Define bus.
3. Define register transfer language (RTL).
4. What is main advantage of RTL.
5. Explain the concept of memory and bus transfer.
6. Draw the block diagram of ALU to perform the following operstions (use MUX)
   (*i*) AND (*ii*) OR
   (*iii*) NOR (*iv*) NAND
7. Explain the different shift operations with examples.
8. Design a 4-bit arithmetic circuit which is capable of performing the following micro-operation
   (*a*) Addition (with and without carry)
   (*b*) Subtraction (with or without borrow)
   (*c*) Increment
   (*d*) Decrement
   (*e*) Transfer
9. A digital computer has a common bus system for 8 registers of 8 bit each. The bus is constructed with
   (*i*) How many select inputs are there in each MUX.
   (*ii*) What size of MUX are needed.
   (*iii*) How many MUX's are there in bus.

in which it is possible to write a computer program. All other languages are said to be *high level* or *low level* according to how closely they can be said to resemble machine code.

In this context, a low-level language corresponds closely to machine code, so that a single low-level language instruction translates to a single machine-language instruction. A high-level language instruction typically translates into a series of machine-language instructions.

Low-level languages have the advantage that they can be written to take advantage of any peculiarities in the architecture of the central processing unit (CPU) which is the "brain" of any computer. Thus, a program written in a low-level language can be extremely efficient, making optimum use of both computer memory and processing time. However, to write a low-level program takes a substantial amount of time, as well as a clear understanding of the inner workings of the processor itself. Therefore, low-level programming is typically used only for very small programs, or for segments of code that are highly critical and must run as efficiently as possible.

High-level languages permit faster development of large programs. The final program which is executed by the computer is not efficient, but the savings in programmer time.

This is because the cost of writing a program is nearly constant for each line of code, regardless of the language. Thus, a high-level language where each line of code translates to 10 machine instructions costs only one tenth as much in program development as a low-level language where each line of code represents only a single machine instruction.

In addition to the distinction between high-level and low-level languages, there is a further distinction between *compiler languages* and *interpreter languages*.

4.5.1. Machine Language

The very lowest possible level at which one can program a computer is in its own native machine code, consisting of strings of 1's and 0's and stored as binary numbers. The main problems with using machine code directly are that it is very easy to make a mistake, and very hard to find it once you realize the mistake has been made.

4.5.2. Assembly Language

Assembly language is the symbolic representation of machine code, which also allows symbolic designation of memory locations. Thus, an instruction to add the contents of a memory location to an internal CPU register called the *accumulator* might be add a number instead of a string of binary digits (*bits*).No matter how close assembly language is to machine code, the computer still cannot understand it.

The assembly-language program must be translated into machine code by a separate program called an *assembler*. The assembler program recognizes the character strings that make up the symbolic names of the various machine operations, and substitutes the required machine code for each instruction. At the same time, it also calculates the required address in memory for each symbolic name of a memory location, and substitutes those addresses for the names. The final result is a machine-language program that can run on its own at any time; the assembler and the assembly language program are no longer needed.

To help distinguish between the "before" and "after" versions of the program, the original assembly language program is also known as the *source code*, while the final machine language program is designated the *object code*.

If an assembly-language program needs to be changed or corrected, it is necessary to make the changes to the source code and then re-assemble it to create a new object program. Assembly Language is a Symbolic Language.

4.5.3. Compiler Language

Compiler languages are the high-level equivalent of assembly language. Each instruction in the compiler language can correspond to many machine instructions. Once the program has been written, it is translated to the equivalent machine code by a program called a *compiler*. Once the program has been compiled, the resulting machine code is saved separately, and can be run on its own at any time.

As with assembly-language programs, updating or correcting a compiled program requires that the original (source) program be modified appropriately and then recompiled to form a new machine-language (object) program.

Typically, the compiled machine code is less efficient than the code produced when using assembly language. This means that it runs a bit more slowly and uses a bit more memory than the equivalent assembled program. To offset this drawback, however, we also have the fact that it takes much less time to develop a compiler-language program, so it can be ready to go sooner than the assembly-language program.

4.5.4. Interpreter Language

An interpreter language, like a compiler language, is considered to be high level. However, it operates in a totally different manner from a compiler language. Rather, the interpreter program resides in memory, and directly executes the high-level program without preliminary translation to machine code.

This uses an interpreter program to directly execute the user's program has both advantages and disadvantages. The primary advantage is that one can run the program to test its operation, make a few changes, and run it again directly. There is no need to recompile because no new machine code is ever produced. This can enormously speed up the development and testing process.

On the down side, this arrangement requires that both the interpreter and the user's program reside in memory at the same time. In addition, because the interpreter has to scan the user's program *one line at a time* and execute internal portions of itself in response; execution of an interpreted program is much slower than for a compiled program.

4.6. INTRODUCTION TO 8085