BLOCK I:
INSTRUCTION SET ARCHITECTURE AND PROCESSOR DESIGN

Unit 1 : Instruction Set Design and Architecture
Unit 2 : Combinational Circuit and its Applications
Unit 3 : Computer Arithmetic
Unit 4 : Register Transfer Language and Processor Logic Design

UNIT 1:INSTRUCTION SET DESIGN AND ARCHITECTURE

Unit Structure:

Space for learners:

Space for learners:

1.1 INTRODUCTION

In this unit, we will discuss addressing types, addressing modes and representation of characters. The organization of computer processor as well as various registers is explained in brief. Here, machine language program using different addressing type is elaborated. We will also know about the instruction cycle. At the end of the chapter integer, fixed point representation, floating point representation and character representation inside computer are discussed.

1.2 UNIT OBJECTIVES

The objectives of the unit are:

- To know the addressing type
- To know the addressing mode
- Overview of processor
- Overview of registers
- To know about instruction cycle
- Data representation in computer

1. 3 INSTRUCTION SET DESIGN

An instruction set is a collection of machine language or assembly language instructions that are understood by central processing unit (CPU). The following issues are considered in instruction set design:

- Whether operands are to be stored in registers, memory, stack or accumulator

- How many operands are present in instructions 0, 1, 2, or 3
- Whether access modes of operands are register, immediate, indirect and so on.
- What are the operations that are supported in instruction add, sub, mul etc.

Space for learners:

1.3.1 How many addresses

Let us assume the statement in a high level programming language given bellow

a = a + b + a * c

It is clear that the value of *a* multiply with *c*is added with *a, b* and the final result is stored in the variable *a*. You know the precedence and associativity rules of high level languages. However, you cannot expect the computer hardware to directly understand these rules.

It requires operations to be performed in small steps. The desired result will be produced after going through **sequence of simple steps**. Hence, it eliminates the necessity for the machine to understand about these rules. In most of the cases operands name i.e. **address** is used rather than value. The machine may be following types depending on addresses:

- 3-address machines
- 2-address machines
- 1-address machines
- 0-address machines

Here number 0, 1, 2, 3 indicates maximum number of address/operand the machine can have.

Here we will use the convention that 'first operand is destination' in an instruction. This means we will consider that the result of operation will be stored in first operand of the instruction.

STEP TO CONSIDER

The address may be either memory or computer registers. In a particular machine final result of operation may be stored in first, or last operand. Here, we consider that the first operand will hold the result of the operation.

1.3.1.1 *3-address machines*

Space for learners:

The general format of a 3-address machine instruction is:

Operation dst, op1, op2

Here, *operation* indicates opcode of the operation to be performed, the first operand *dst* represent destination operand i.e. where the result of operation will be stored, op1 and op2 indicates two source operands between which operation is to be performed. Thus the following instruction means:

ADD R2, R1, R0

Add the values stored in register *R1* and *R0*, and store result in the register *R2*.

When all operands of instructions are only in register then we call it a **register-register** machine or a **load-store** machine. Instead of that if all operands of instructions are only in memory then we call it a **memory-memory** machine. The following is such an example:

ADD X, Y, Z

Add the value of variable *y* to the value of variable *z* and then store the result in the memory location *x*. In a memory-memory machine the CPU has to get the operands from memory prior to execution of the operation. After that it has to store the result back in memory. There are several ways to specify the address of an operand. We will discuss this topic in addressing mode section.

Let us now see how to implement a 3-address instruction for the statement

$a = a + b + a * c$

Answer:

```
MUL R4, a, c          # store a*c in R4
ADD R1, a, b          # store a + b in R1
ADD R1, R1, R4        # Store result in R1
```

The final result of the expression can be found in register R1.

1.3.1.2 2-address machines

The general format of 2-address machine instruction is:

Operation dst, op

where, *operation* is opcode of the operation, *dst* represent the source operand as well as destination, *op* represent the second source operand. Let us see the following instruction

ADD R1, R2

The meaning of this instruction is to add the values stored in registers R1 and R2, and then store the result back in register R1.

The advantage of 2-address instructions over three-address instructions is that it helps in preserving memory, since they are shorter. More over shorter instructions take less time for fetching. The drawback having two-address instructions is that one of the source operands is destroyed. It requires extra moves *to retain the operand* as sometimes operand may be needed later.

Let us now see how to implement a 2-address instruction for the statement

$a = a + b + a * c$

Answer:

MUL c, a	# multiply *a, c* and store in *c*
MOV R1,c	# move content of c to R1
ADD b,a	# add a, b and store in b
MOV R2,b	# move content of b to R2
ADD R1, R2	#add R1, R2 and store in R1

The final result of the expression can be found in register R1.

1.3.1.3 1-address machines

In a 1-address machine accumulator has a source operand and result of operation is put back implicitly in the accumulator. The instruction needs to indicate the second source operand. The format of a 1-address instruction is as follows:

Space for learners:

operation op

The opcode '*operation*' is the name of the operation to be done, *op* indicates either source or destination operand. Here the instruction:

ADD a

It means addition of value of variable *a* with the content of accumulator. The result of addition is put in the accumulator. The accumulator is a special purpose register.

Let us now see how to implement a 1-address instruction for the statement.

$a = a + b + a * c$

Answer:

LOAD a	# load content of *a* in accumulator
MUL c	# multiply accumulator i.e.*a* and *c*
ADD b	# add *b* to previous contents of the accumulator i.e. $a * c + b$
ADD a	# $a * c + b + a$
STO a	# store the final result in location *a*

The final result of the expression can be found in the memory location a.

STOP TO CONSIDER

As the number of addresses reduced the number of instructions increases to do the same task.

1.3.1.4 Zero-address machines

The zero-address machines are implemented using stack. A stack is last in first out (LIFO) data structure that is operated by using PUSH and POP. PUSH moves an operand from computer memory into top of stack, on the other hand POP gets out the last item from top of the stack. Only PUSH and POP indicates an operand. No other opcode specify any operand. This is the reason why it is called a zero address machine. The question is how then operands are handled by the machine for the operation. It is done by

Space for learners:

extracting top two elements of stack and putting the result back into stack.

Let us see how to implement a zero-address instruction for the statement

*a = a + b + a * c*

Answer:

```
PUSH a          # push the value of a
PUSH c          # push the value of c
MUL             # multiply top two value a * c
PUSH b          # push the value of b;
ADD             # add top two value b + a * c
PUSH a          # push the value of a
ADD             # add top two value a + b + a * c
POP  a          # store in top of stack in a
```

The final result of the expression can be found in the memory location a.

Space for learners:

1.3.2 Types of Instructions

The computer supports the following types of instructions:

- Data Transfer Instructions
- Arithmetic Instructions
- Bit Manipulation Instructions
- Program Execution Transfer Instructions
- Processor Control Instructions
- Iteration Control Instructions
- Interrupt Instructions

1.3.2.1 Data Transfer Instructions

These instructions transfer data from the source to the destination location inside the computer. The common data transfers are among registers or between registers and memory or between the register (s) and the input/output devices. Different computer uses various mnemonics for the same instruction. The following are some of the data transfer mnemonics with their meaning.

- **MOV**: Transfer data from register to resister or resister to memory.
- **ST:** Store from register (accumulator) to memory
- **LD:** Load data from memory to register
- **PUSH:** Transfer data from CPU register to top of the stack.
- **POP:** Transfer data from top of stack to CPU register
- **XCHG:** Exchange data between two given locations.
- **IN:** Read data from an input port to accumulator.
- **OUT:** Transfer data from accumulator to particular output port.

Space for learners:

1.3.2.2 Arithmetic Instructions

The basic arithmetic operations are addition, subtraction, multiplication and division between two numbers. These arithmetic operations are performed between two operands. Some of the arithmetic operations may be performed on a single operand too.

Following a few arithmetic instructions:

- **ADD**: Add the contents of two source locations.
- **MUL**: Multiply the contents of two source locations.
- **DIV**: Divide content of one source locations with the other.
- **SUB**: Subtract content of one source locations from the other.
- **ADC**: Add the contents of two source locations with carry.
- **INC**: Increment the content of source location by 1.

1.3.2.3 Bit Manipulation Instructions

These instructions manipulate data in bit level i.e. operations like shift or logical. Below isa few instructions of this group with meaning are given:

- **NOT**: This invert each bit of source bit pattern.

- **AND**: Logical AND operation between each corresponding bit of both source operand.
- **OR:** Logical OR operation between each corresponding bit of both source operand.
- **XOR** –Perform logical Exclusive-OR operation between each corresponding bit of both source operand.
- **SHL:** Perform bits shift towards left and fill zero in LSBs.
- **SHR**: Perform bits shift towards left and fill zero in MSBs.

Space for learners:

1.3.2.4 Program Execution Transfer Instructions

These instructions transfer the control during an execution of instructions. The transfer of control during execution of instruction may be conditional or unconditional. A few such examples are listed below:

- **CALL**: It calls a subprogram and saves the return address on top stack.
- **RET:** Returns from subprogram/function to the main program.
- **JMP:** Jumps to the given address and process the next instruction.
- **JC:** Jumps when value of carry flag is 1
- **JNC:** Jumps when value of carry flag is 0

1.3.2.5 Processor Control Instructions

These instructions set or reset the flag values and thus control the actions of the processor. Following are the instructions under this group:

- **STC:** Set the carry flag (CF) to 1
- **CLC:** Reset the carry flag i.e. CF = 0
- **CMC:** Complement state of carry flag.
- **STI:** Set the interrupt flag to 1.
- **CLI:** Reset the interrupt flag to 0.

1.3.2.6 Iteration Control Instructions

These instructions can execute a group of instructions repeatedly. A few list of iteration control instructions are:

- **LOOP**: Execute a group of instructions repeatedly until the condition is true.
- **JCXZ:** Jump to a given address if CX = 0

1.3.2.7 Interrupt Instructions

These instructions call an interrupt during execution of instructions.

- **INT:** Interrupt the process and call service routine.
- **INTO:** Interrupt the process if OF = 1
- **IRET:** Return to main program from interrupt service.

CHECK YOUR PROGRESS-I

1. When all operands of instructions are only in register then we call it a _____machine**.**

2. If all operands of instructions are only in memory then we call it a _____ machine.

3. The drawback having two-address instructions is that one of the source operands is _____.

4. In a one-address machine the result of operation is put back implicitly in the _____.

5. The zero-address machines are implemented using _____.

State TRUE or FALSE:

6.The processor has three types of organization.

7. The advantage of two-address instructions over three-address instructions is that it helps in preserving memory.

8. The accumulator is a special purpose register.

9. MOV is control transfer instruction.

10. POP insert an operand from computer memory into top of stack.

Space for learners:

1.4 ADDRESSING MODES

In a typical instruction, we see the address fields are relatively small. The purpose of addressing mode is to reference main memory locations as large as possible. This is the reason why a variety of addressing modes have been implemented. The most commonly used addressing modes are:

- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement
- Stack

Mode	Algorithm	Advantage	Disadvantage
Immediate	Operand=A	No memory reference	Limited operand magnitude
Direct	EA=A	Simple	Limited address space
Indirect	EA=(A)	Large address place	Multiple memory reference
Register	EA=R	No memory reference	Limited address space
Register indirect	EA=(R)	Large address place	Extra memory reference
Displacement	EA=A+(R)	Flexibility	Complexity
Displacement	EA= top of stack	No memory reference	Limited applicability

Table 1.1Basic Addressing Modes

The Table 1.1depicts the address calculation procedure for each addressing mode. Each of the addressing modes will be represented with different opcodes. The opcode may be one or more bits in the instruction format.

STOP TO CONSIDER

The effective address of operand is calculated after decoding the opcode.

Space for learners:

1.4.1 Immediate Addressing

The immediate addressing holds the operand value in the instruction.

Operand = A

This addressing mode is generally used to set initial values of variables or constants. The primary advantage is that there is no need of memory reference. Thus it saves one memory or cache cycle in the instruction cycle. The disadvantage of immediate addressing mode is that size of the number is limited to size of the address field.

1.4.2 Direct Addressing

In direct addressing mode the address field holds the effective address of the operand:

EA = A

The advantage of direct addressing mode is that it needs only one memory reference. The disadvantage this addressing mode is limited address space accessibility.

1.4.3 Indirect Addressing

In direct addressing mode usually length of the address field is less than word length. It causes limitation in address range. If the address field refers to address of a word in memory, it can access a full-length address of the operand. This way of accessing memory word is known as indirect addressing. In indirect addressing mode the address field contains address of another memory location where the value of actual operand remains.

EA = (A)

The parenthesis interpreted as contents of '*A*' is another address. The disadvantage of indirect addressing is that it requires two

Space for learners:

memory references to fetch actual operand value, first to get its address and next to get its value.

Space for learners:

1.4.4 Register Addressing

The register addressing mode has similarity to direct addressing. The difference here is that address field indicates a register instead of main memory address:

EA = R

The register *R* specifies the address where the operand value contains. The advantages of this mode are that a small address field is needed and no memory references needed means less time required for fetching instruction. The disadvantage of this mode is that the available address space is limited to registers only.

1.4.5 Register Indirect Addressing

The register indirect addressing mode is similar to indirect addressing mode. The only difference is that address field refers to a register instead of memory location. Let us see the register indirect address.

EA = (R)

The advantages and disadvantages of register indirect addressing mode are similar to indirect addressing mode. But, register indirect addressing mode has one more advantage since it uses one less memory reference it saves one cycle time when it is executed.

1.4.6 Displacement Addressing

The displacement addressing mode combines the direct addressing with register indirect addressing. The effective address in this mode looks like as:

EA = A + (R)

This addressing mode the instruction contains two address fields, out of which at least one of it is explicit. The value stored in one of

the addresses field (i.e. *A*) is used directly. The contents of second address field i.e. register is added to *A* to obtain the effective address. We will discuss three most commonly used displacement addressing:

- Relative addressing
- Base-register addressing
- Indexing

RELATIVE ADDRESSING: The relative addressing is also known as PC-relative addressing. In this mode of addressing the register that implicitly referenced is program counter (PC). As we know PC contains the address next instruction to be executed. Hence, it is added to the address field in order to produce the *EA*. This is how the effective address in this addressing mode is a displacement relative to the address of the instruction.

EA= PC + address field value

BASE-REGISTER ADDRESSING: In the base-register addressing mode, the referenced register contains a main memory address. The address field indicates a displacement from that address, which is usually an unsigned integer.

EA=base register + address field value

INDEXING: In this addressing mode, the effective address of the operand is calculated by adding content of index register with address field value.

EA= IR+ address field value

The indexing mechanism is extensively used for implementing iterative operations. Suppose a list of numbers present in memory location starting from A and we want to add 1 to each number on that list. Here, we have to fetch each number and after adding 1 to it, store it back to that location. The effective addresses that requires are *A, A + 1, A + 2, . . .,* and so on to last location. It can be done easily with indexing. The value of A is stored in the instruction's address field value, and the index

Space for learners:

register is initialized to 0. At the end of each operation, index register is incremented by 1.

Space for learners:

1.4.7 Stack Addressing

The stack addressing is also referred to as a *last-in-first-out* or *queue pushdown list.* In this addressing mode items are placed to the top of the stack. Hence, the stack is partially filled at any given time.

The stack is associated with a pointer called stack pointer (SP) whose value refers to the top address of the stack. If the top two item of the stack is in processor registers, the SP references the third item of the stack. The stack pointer is a dedicated special purpose register. It is a form of implied addressing. The instructions do not require a memory reference; it always implicitly indicates the top of the stack.

CHECK YOUR PROGRESS-II

11. The purpose of addressing mode is to reference _____ as large as possible**.**

12. The immediate addressing mode generally used to set initial values of _____ or _____.

13. In direct addressing mode address field holds _____address of the operand.

14. In indirect addressing mode the address field contains _____ of another memory location.

15. The displacement addressing mode combines the direct addressing with _____ addressing.

State TRUE or FALSE:

16.The advantage of direct addressing mode is that it needs only one memory reference.

17. In register addressing mode one memory references needed.

18. The stack is associated with a pointer called stack pointer.

19. Effective address is calculated after decoding an instruction.

20. In stack addressing two memory references needed.

Space for learners:

1.5 PROCESSOR ORGANISATION

The computer processor needs to do the following things to execute an instruction:

• **Fetch instruction:** The processor *has* to read instructions from memory i.e. from register, cache or main memory.

• **Interpret instruction:** After reading an instruction it is decoded to know what action to be performed.

• **Fetch data:** During the execution of an instruction it may need to read data from computer memory or input/output (I/O) module.

• **Process data:** In execution time of an instruction, it may have to perform either arithmetic or logical operation on data.

• **Write data:** At the end of an instruction execution, the results may need to write data to main memory or an I/O module. In order to do these, it clears that the processors sometimes have to store intermediate data. Hence, the processor requires a small internal memory.

Figure 1.1 is a block diagram of a processor depicting its connection to the rest of the system through system bus. The vital components of the central processing unit are

- Arithmetic and logic unit (ALU)
- Control unit (CU).
- Registers

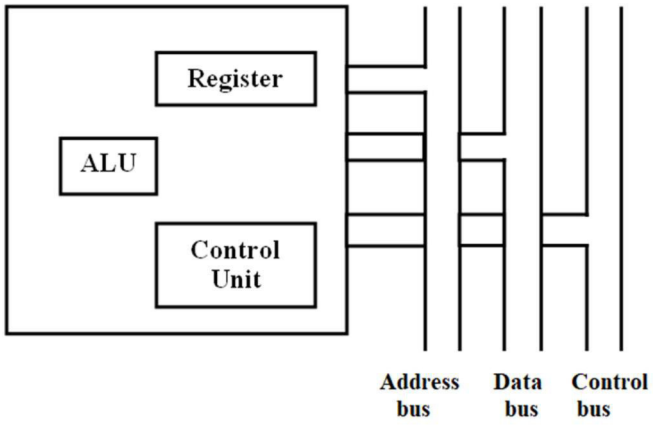


Figure1.1: The block diagram of CPU

The ALU performs the actual processing of data. The CU controls the data and instructions movement in the processor. It also controls the operations of the ALU. The figure also depicts internal memory of processor, called registers.

In general, CPU or processor organization has three categories depending on the number of address fields:

- Single Accumulator organization
- General register organization
- Stack organization

In accumulator based organization, a special purpose register called accumulator is used for performing the operations. In general, register organization involves different registers in the computation tasks. In the stack organization the calculations performed on top of the stack. The instruction of stack organization does not contain any address field. In general, a combination of different organizations is mostly used.

Space for learners:

1.6 REGISTER ORGANISATION

The computer system consists of memory in different level called hierarchy. At top levels of the hierarchy means memory is faster than the below level. In this level it is smaller as well as more expensive. The register inside the processor is top level memory followed by cache memory and main memory respectively. The registers have two categories:

• User-visible registers
• Control and status registers

STOP TO CONSIDER

The address bus, data bus and control bus are together called system bus. Operand address bits can travel through address bus, data bits travel through data bus and CPU generated signal travel through control bus. The processor interaction with main memory is done through these buses.

1.6.1 User-Visible Registers

Space for learners:

The user-visible registers are used by assembly language programmer in order to minimize main memory references. It can be in the following types:

- General purpose register
- Data
- Address
- Condition codes

General-purpose registers are used to store temporary data during execution of instruction. For a given *opcode* the general-purpose register can hold the operand. This is true use of general-purpose registers. The general-purpose registers sometimes can be used for addressing purpose (e.g., register indirect, displacement).

Data registers can be used to hold data only. It cannot be used for calculating of operand address.

Address registers may either general purpose or devoted to an individual addressing mode. The following are examples of it:

- **Segment pointers:** The segment register is used to hold the address of the base of the segment.

- **Index registers:** These registers are used for auto indexing in indexed addressing.

- **Stack pointer:** In stack addressing a dedicated register is used called stack pointer.

Condition codes (*flags*): These are bits set by the processor depending on result of an operation. As we know, result of arithmetic operation may be positive, negative, zero, or overflow. In this case a condition code *(flag)* is set and result is stored in memory or register. Subsequently the code may be tested during execution of conditional branch operation.

1.6.2 Control and Status Registers

Space for learners:

The operations of processor are controlled by variety of internal registers. In general, these registers are not visible to programmer or user. Here, we will discuss four such essential registers.

• **Program counter (PC):** It holds the address of the next instruction to be executed.

• **Instruction register (IR):** It holds the address of currently executed instruction.

• **Memory address registers (MAR):** It holds the address of an instruction to be fetched.

• **Memory buffer registers (MBR):** Holds data that needs the current instruction or result produced by the instruction. Another register that is included in a processor is called the *program status word* (PSW). It contains condition code and other status information. The followings are status flags:
• **Sign:** It holds sign bit of the recent arithmetic operation.
• **Zero:** It is set when the result of operation is 0.
• **Carry:** It is set if addition operation produce a carry or borrow (for subtraction) from lower order bit.

• **Equal:** Set if a logical comparison of two operands is equal.

• **Overflow:** When arithmetic operation produces overflow it is set.

• **Interrupt Enable/Disable:** This flag is used to enable or disable the interrupts.

• **Supervisor:** It indicates the execution mode of processor (supervisor or user). Some of the privileged instructions are executed only in supervisor mode. Similarly, certain memory location can be accessed through supervisor mode only.

1.7 INSTRUCTION CYCLE

An instruction cycle goes through the following stages:
•**Fetch:** The processor reads the next instruction from PC

•**Execute:** Decode the *opcode* and perform the required operation.
• **Interrupt:** If interrupt occurs, pause the current process, save status of it and go to the interrupt.

Before elaborating instruction cycle it's important to know one additional stage called indirect cycle.

CHECK YOUR PROGRESS-III

21. The _____ performs the actual processing of data.

22. The CPU organization has _____ categories.

23. The computer system consists of memory in different level called _____.

24. _____registers are used to store temporary data during execution of instruction.

25. The execution mode of processor either _____or _____.

State TRUE or FALSE:

26. The CU controls the data and instructions movement in the processor.

27. The segment register is used to hold the address of the base of the segment.

28. PC holds the address of current instruction executing.

29. MBR holds the address of an instruction to be fetched.

30. Carry flag is set if addition operation produces a carry.

1.7.1 The Indirect Cycle

During instruction execution it may have one or more operands that need memory access. In case of indirect addressing additional memory accesses are needed. The Figure 1.2 depicts instruction cycle.

After fetching the instruction it is checked to see if it involves any indirect addressing. If indirect addressing involves, the operands are fetched according to indirect addressing. After execution, an interrupt will occur before fetching the next instruction.

Space for learners:

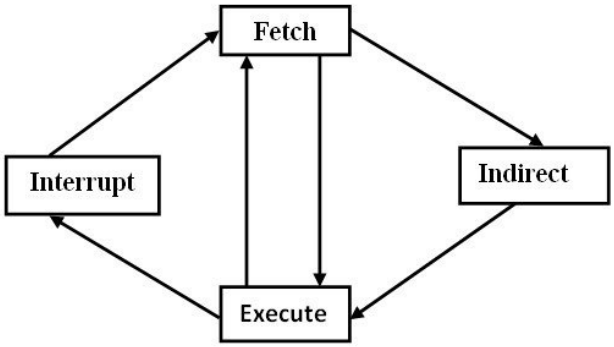


Figure 1.2 Instruction Cycle

After fetching the instruction the operand are fetched from memory. If the operand is in register then fetching is not required. Once execution of instruction is completed the result may be needed to store in main memory.

1.7.2 Data Flow

In an instruction cycle sequence of events occurs according to the design of processor. Suppose, a processor consists of a program counter (PC), a memory address register (MAR), a memory buffer register (MBR), and an instruction register (IR).

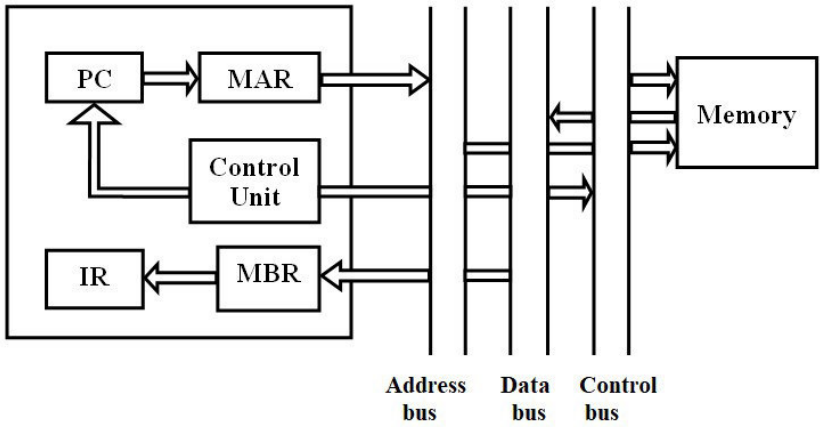


Figure 1.3 Data Flow, Fetch cycle

Figure 1.3 depicts the data flow during fetch cycle. The PC holds the address of the next instruction to be fetched. This address is placed on the address bus through the MAR.

Space for learners:

The CU requests a main memory read to fetch the required data for the instruction. The requested result is placed on the data bus and goes to the MBR and finally reached the IR. In the mean time, the PC is incremented for fetching the next instruction. At the end of fetch cycle, the CU checks the IR to know whether it's holding an operand specifier using indirect addressing. If indirect addressing is found, indirect cycle is performed after fetch cycle. Figure 1.4 depicts this simple cycle. The address reference bits of the MBR are transferred to the MAR. After that the CU places a memory read request. Then desired address of the operand is placed in MBR through address bus.

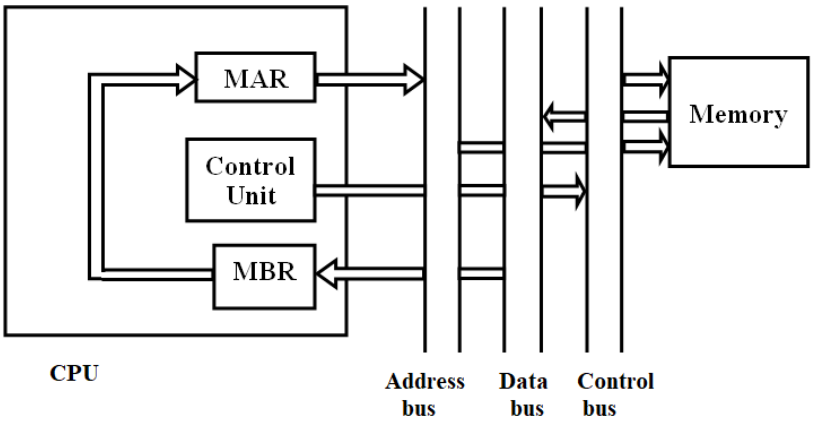


Figure 1.4 Data Flow, Indirect cycle

The fetch and indirect cycles are very simple. The *execute cycle* may have various stages. It many involve ALU operation, register transfer of data, read/write operation from memory or I/O. On the other hand the *interrupt cycle* is as simple as fetch and indirect cycle. It is depicted in figure 1.5. Before going to interrupt, current status of the PC must be in order to resume normal activity after the interrupt. So, the contents of the PC is transferred to the MBR and written to memory. For this purpose special memory location is reserved and it is loaded into the MAR from the CU. The special memory may be a stack pointer. The PC is filled with the address of interrupt routine. Henceforth, the next instruction cycle will fetch the desired instruction.

Space for learners:

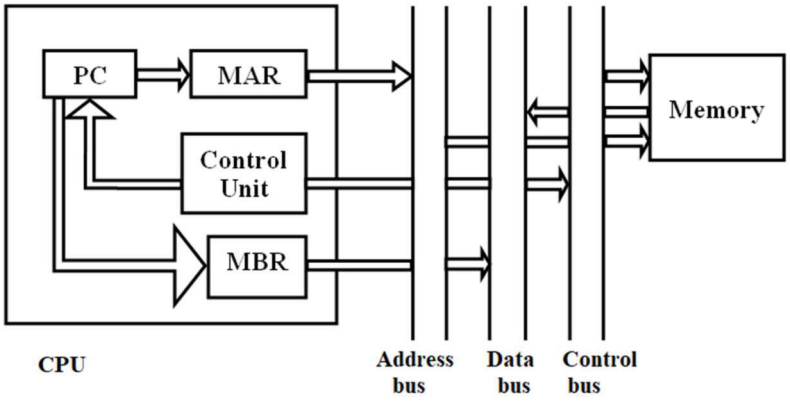


Figure 1.5 Data Flow, Interrupt Cycle

STOP TO CONSIDER

The instruction cycle has different stages fetching, decoding opcode, effective address calculation, execution of operation on data and writing data in memory that are executed in sequence.

1.8 DATA REPRESENTATION

A digital computer represents all types of information in binary number system due to following reasons:

• In digital computers all electronic components operate in binary mode.
• Computers use binary system where only two digits present.
• Whatever can be done using decimal number system can also be done using a binary number system.

1.8.1 Number representation

The numbers in computer are represented using binary number system. An r base number system uses r distinct digits. The decimal number has 10 digits. So, decimal numbers are 10-base number system. The binary numbers system has two digits '0' and '1'. It is called base 2 number system. The octal numbers system has eight digits 0, 1, 2, 3, 4, 5, 6 and 7. The decimal number 831.6 can be written as follows with power of base 10.

Space for learners:

$8 \times 10^2 + 3 \times 10^1 + 1 \times 10^0 + 6 \times 10^{-1}$

When a binary number 101101 is written in this way with power of base 2, it provides decimal equivalent.

STOP TO CONSIDER
The hexadecimal numbers system has 16 digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.

$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45$

The decimal number can be converted to r base number system by using the steps:

- At first the number is separated into its integer and fraction parts and then each part converted separately.
- The integer part is converted to r base by dividing it successively with r until it becomes zero.
- The remainders in reverse order give the r base equivalent.
- The fraction part is converted to r base by multiply it repeatedly by r until its fraction part becomes zero.

Suppose, decimal number 112.8125 has to convert into binary. Here integer part is 112 and fraction part is 0.8125. At first, we will convert integer part 112 into binary then fraction part according to above rules. Since binary number system is 2 base we will divide 112 by 2 until it become zero. The following table depicts the process.

Division	Remainder
112 / 2 = 56	0
56 / 2 = 28	0
28 / 2 = 14	0
14 / 2 = 7	0
7 / 2 = 3	1
3 / 2 = 1	1
1 / 2 = 0	1

Now write down the remainder in reverse order i.e. 1110000 which is binary equivalent number of decimal integer 112. Next, the fraction part 0.81252 is multiplied by 2. The fraction of that result is again multiplied by 2until fraction part become zero.

Space for learners:

Multiplication	Resultant integer part (R)
0.81252 x 2= 1.625	1
0.6252 x 2= 1.25	1
0.252 x 2= 0.50	0
0.50 x 2= 1.0	1
0 x 2 = 0	0

The binary equivalent of fraction will be 0.11010. Using the same rules we can convert a decimal number to any base system.

1.8.1.1 Complements

Complements simplify the subtraction and logical manipulation in digital computer. There are two types of complements present in *r* base system namely *r's* and *(r – 1)'s* complement. If a number *N* in *r* base contains n digits, the *(r – 1)'s* complement of *N* is calculated as $(r^n - 1) - N$. For a decimal number, the 9's complement of *N* is $(10^n - 1) - N$. Thus, 9's complement of 545700 is 999999 – 545700 = 454299. In case of binary number, the 1's complement of *N* is calculated as $(2^n - 1) - N$. Thus, 1's complement of 1011000 is 1111111 – 1011000 = 0100111. Simply, 1's complement is obtained by just toggling all bits. The *r's* complement of a number *N* with n-digit is calculated as $n - N$. This is like adding 1 to the *(r – 1)'s* complement of the number. Thus, 10's complement of 2389 is 7610 + 1 = 7611. Similarly, 2's complement of 101100 is 010011 + 1 = 010100.

CHECK YOUR PROGRESS-IV

31. Computers use _____ system where only two digits present.

32. The octal numbers system has _____ digits.

33. The hexadecimal numbers system has _____ digits.

34. Complements simplify the _____ operation.

State TRUE or FALSE:

35.The decimal integer part is converted to r base by dividing it successively with r until it becomes zero.

36. There are two types of complements present in r base system namely r's and (r + 1)'s complement

37. 1's complement is obtained by just toggling all bits.

38. The binary number system has base 2.

Space for learners:

All positive integer numbers and zero can be considered as unsigned number. In order to represent negative numbers in computer signed numbers must be used. Because + and – signs are not present, rather these sign are represented by either '0' or '1'. The most significant bit of signed number is 0 for positive and 1 for negative. The fixed-point number representation has three parts as depicts in figure 1.6.

- Sign field
- Integer field
- Fractional field.

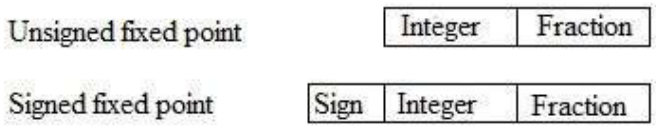


Figure 1.6: Fixed-point number representation

The 2's complementation representation is common in computer due to easier for arithmetic operations.

In a 32 bit register 1 bit reserved for the sign. Assume 15 bits are reserved for the integer part and 16 bits for the fractional part. The number -43.625 can be represented in register as depicted in figure 1.7.

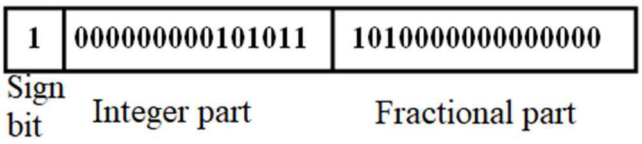


Figure 1.7: Representation of -43.625

The sign bit 1 represent - and 000000000101011 is 15 bit binary equivalent for decimal 43 and 1010000000000000 represent 16 bit binary equivalent for fraction 0.625.

1.8.3 Floating-Point Representation

The floating number consists of two parts. The first part is a signed fixed point number that is called mantissa. The second part exponent represents the position of the decimal (or binary) point. The fixed

Space for learners:

point mantissa is either fraction or integer. The floating point number always represent in the form $M \times r^e$.

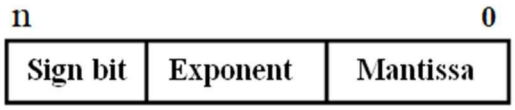


Figure 1.8: Floating point representation in register

The mantissa *M* and the exponent *e* present in the register with their sign as depicted in figure 1.8. A floating-point decimal number use base 10 for the exponent and binary number use base 2 for the exponent. A floating-point number is called normalized if the most significant bit (MSB) of the mantissa is 1. For positive integer, the MSB, or sign bit, is 0 and the remaining bits represent the magnitude. On the other hand for negative number, the MSB, or sign bit, is 1. The rest of the number can be represented in one of three ways

Signed-magnitude representation

Signed-1's complement representation

Signed-2's complement representation

Using floating point representation any non-zero number can be represented in the normalized form. Suppose, in 32-bit register 1 bit use as sign bit, 8 bits use for signed exponent, and remaining 23 bits represents fractional part. Now the decimal number −53.5 can represented as depicted in figure 1.9. The binary equivalent of -53.5 is $(-110101.1)_2$ and normalized representation is $(-1.101011)x2^5$

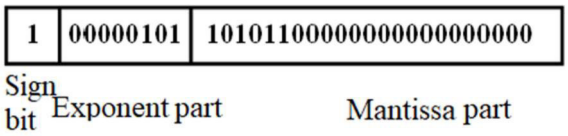


Figure 1.9: Floating point representation of -53.5

1.8.4 Character Representation

Different character codes are used to represent alphanumeric characters in bits 0 and 1. The most commonly used character code is American standard Code for Information Interchange (ASCII).

Space for learners:

ASCII uses 7-bits that provides 128 bit-patterns. In ASCII there are 26 lowercase and uppercase letters, 10 digits, and 32 punctuation marks. The remaining represents whitespace characters and special *control characters*. The uppercase A-Z, lowercase a-z and the digits 0-9 are in continuous series.

Space for learners:

Bit positions 654								Bit positions
000	**001**	**010**	**011**	**100**	**101**	**110**	**111**	**3210**
NUL	*DLE*	*SP*	0	@	P	‘	p	0000
SOH	*DC1*	!	1	A	Q	a	q	0001
STX	*DC2*	“	2	B	R	b	r	0010
ETX	*DC3*	#	3	C	S	c	s	0011
EOT	*DC4*	$	4	D	T	d	t	0100
ENQ	*NAK*	%	5	E	U	e	u	0101
ACK	*SYN*	&	6	F	V	f	v	0110
BEL	*ETB*	‘	7	G	W	g	w	0111
BS	*CAN*	(	8	H	X	h	x	1000
HT	*EM*	)	9	I	Y	i	y	1001
LF	*SUB*	*	:	J	Z	j	z	1010
VT	*ESC*	+	;	K	[	k	{	1011
FF	*FS*	,	<	L	\	l	|	1100
CR	*GS*	-	=	M	]	m	}	1101
SO	*RS*	.	>	N	^	n	~	1110
SI	*US*	/	?	O	_	o	*DEL*	1111

1.9 SUMMING UP

- An instruction set is collection of machine language or assembly language instructions that are understood by central processing unit (CPU).
- The machine may be 3-address machines, 2-address machines, 1-address machines and 0-address machines

- The computer supported instructions types are Data Transfer Instructions, Arithmetic, Bit Manipulation, Program Execution Transfer, Processor Control, Iteration Control and Interrupt Instructions.
- The most commonly used addressing modes are Immediate, Direct, Indirect, Register, Register indirect, Displacement and Stack addressing.
- CPU or processor organization has three categories: Single Accumulator organization, General register organization and Stack organization.
- The register inside the processor is in top level memory hierarchy followed by cache memory and main memory respectively.
- The registers have two categories: user-visible registers and control and status registers
- **General-purpose registers** are used to store temporary data during execution of instruction.
- **Data registers** can be used to hold data only. It cannot be used for calculating of operand address.
- **Address registers** may either general purpose or devoted to an individual addressing mode.
- **PC** holds the address of the next instruction to be executed.
- **IR** holds the address of currently executed instruction.
- **MAR** holds the address of an instruction to be fetched.
- **MBR** holds data that needs the current instruction or the result produced by the instruction.
- The use of status flags:
  Sign: It holds sign bit of the recent arithmetic operation. **Zero:** It is set when the result of operation is 0. **Carry:** It is set if addition operation produce a carry or borrow (for subtraction) from lower order bit.
- The numbers in computer are represented using binary number system.
- The floating number consists of two parts. The first part is a signed fixed point number that is called mantissa. The second part exponent represents the position of the decimal (or binary) point.
- In ASCII there are 26 lowercase and uppercase letters, 10 digits, and 32 punctuation marks. The remaining represents whitespace characters and special *control characters*.

Space for learners:

1.10 ANSWERS TO CHECK YOUR PROGRESS

1. Register-register
2. Memory-memory
3. Destroyed
4. Accumulator
5. Stack
6. True
7. True
8. True
9. False
10. False
11. Memory location
12. Variable, constant
13. Effective
14. Address
15. Register indirect
16. True
17. False
18. True
19. True
20. False
21. ALU
22. Three
23. hierarchy
24. General
25. Supervisor, user
26. True
27. True
28. False
29. False
30. True
31. Binary
32. Eight
33. Sixteen
34. Subtraction
35. True
36. False
37. True
38. True

Space for learners:

1.11 POSSIBLE QUESTIONS

Short answer type questions:

1. What is an instruction set?
2. Write the type of instruction for the following:
   JUMP, ADD
3. What are the types of CPU organization?
4. Arrange the followings in ascending order of access time: Secondary memory, Register, Main Memory, Cache Memory
5. What type of buses the system bus has?
6. What is the use of immediate addressing?
7. What is the Indirect Addressing? Give examples.
8. What is an accumulator?
9. Write assembly language code to evaluate X = (A-B) + (C-D) for stack based CPU
10. What are the categories of registers?
11. What happens to PC when interrupt occurs?

12. What is floating point representation?
13. What is 1's complement of 10011010?
14. What is 2's complement of 11000111?
15. Convert the decimal number 26.578 into binary number.

Long answer type questions:

1. Briefly explain the various addressing modes.
2. Briefly explain the instruction cycle.
3. List any five instruction types with adequate examples.
4. Convert decimal number 56.789 into binary, octal and hexadecimal number.
5. Briefly explain the data flow process with block diagram.

Space for learners:

1.12 REFERENCES AND SUGGESTED READINGS

- Computer Architecture and Organization by B. Govindarajalu.; TMH publication.
- Advanced Computer Architecture A systems Design Approach by Richard Y. Kain; PHI Publication
- Computer Organization and Architecture Designing for Performance by William Stallings; Pearson Education
- Computer System Architecture by M. Morris Mano, PHI Publication.

UNIT 2: COMBINATIONAL CIRCUITS AND ITS APPLICATIONS

Space for learners:

Unit Structure

Space for learners:

2.1 INTRODUCTION

This chapter describes the combinational circuits and the applications of combinational circuits. Sum of Product (SOP) and Product of Sum (POS) forms are the basic building blocks of the combinational circuits. When the logic gates are connected together to produce some specific output the resulting electronic circuit is known as combinational circuits. The output of the circuit always depends on the combination of the input variables.

2.2UNIT OBJECTIVES

The unit is describing the designing and applications of combinational logic circuits. After completing the unit students′ will able to:

- Analyze and apply different combinations of the logic gates.
- Design combinational circuits from the Boolean expressions.
- Design combinational circuits from the truth table.
- Describe the universal behaviour of NAND and NOR logic gates.
- Explain and describe adder circuits.
- Analyze the comparator circuits.
- Describe decoders and encoders
- Describe multiplexers and demultiplexers

2.3 AND-OR LOGIC COMBINATIONAL CIRCUIT:

The Figure 2.1 shows an AND-OR circuit consisting of two input AND gates and one two input OR gate. The Boolean expression for the AND gate outputs and the resulting SOP expression for the

output Y are shown on the circuit diagram. The AND-OR circuit can have any number of AND and OR with any number of inputs.

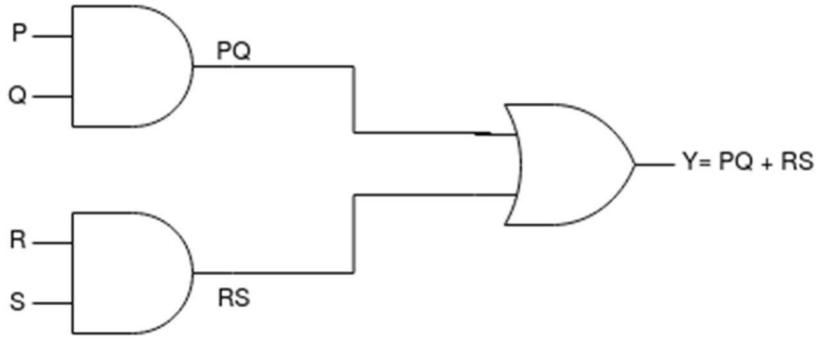


Figure 2.1 AND-OR logic diagram

The truth table for the above combinational circuit is shown in Table-2.1. The outputs of the AND gates are also shown in the table.

INPUTS				PQ	RS	OUTPUT Y
P	Q	R	S			
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	1	1
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	1	1	1	0	1	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	1	1
1	1	0	0	1	0	1
1	1	0	1	1	0	1
1	1	1	0	1	0	1
1	1	1	1	1	1	1

Table 2.1 Truth table for the logic circuit of Figure 2.1

2.4 AND-OR-INVERT LOGIC COMBINATIONAL CIRCUIT

If the output of the AND-OR circuit is complemented i.e. inverted, the resultant circuit is called AND-OR-Inverted circuit. The AND-OR logic implements the SOP expression and the corresponding

Space for learners:

POS expressions can be obtained using the AND-OR-Inverted logic. The logic circuit diagram Figure 2.2 shows an AND-OR-Inverted circuit and development of the POS output expression.

$Y = (P'+Q')(R'+S') = (PQ)'(RS)' = (((PQ)'(RS)')')' = (((PQ)')' +((RS)'))')' = (PQ + RS)'$

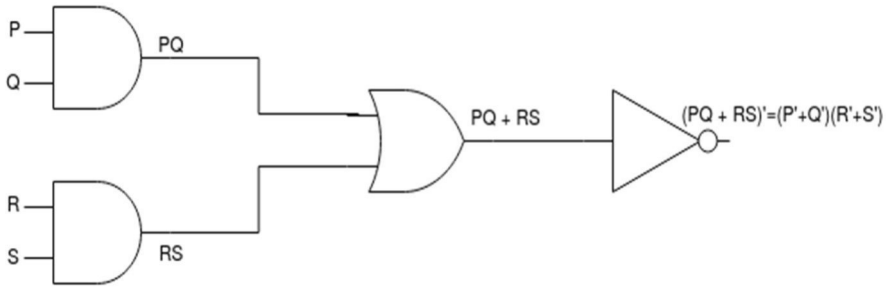


Figure 2.2 AND-OR Invert logic

In general, an AND-OR-Invert circuit can have any number of AND gates each with any number of inputs. A truth table can be developed from the AND-OR truth table in Table 2.1 by simply changing all 1s to 0s and all 0s to 1s in the output column.

STOP TO CONSIDER

- The AND-OR logic implements the SOP expressions, in other words, the SOP expressions are implemented using AND-OR logic.
- The AND-OR-Inverted logic implements POS expressions, in other words, the POS expressions are implemented using AND-OR-Inverted logic

2.5 EXCLUSIVE-OR LOGIC:

The exclusive-OR gate is considered a type of logic gate with its own unique symbol; it is actually a combination of two AND gates, one OR gate, and two inverters (NOT gate) as shown in Figure 2.3. The output is 1 only if the two inputs are at opposite levels.

Space for learners:

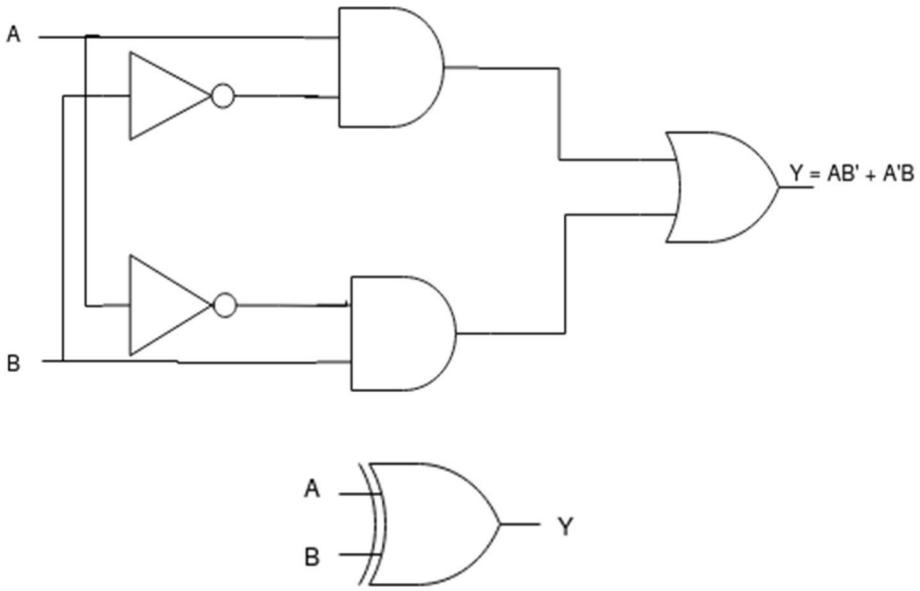


Figure 2.3 Exclusive-OR logic

The output expression for the circuit in Figure 2.3 is Y = AB′ + A′B
i.e. Y = A ⊕ B

The truth table for exclusive-OR is shown in Table 2.2.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.2: Truth table of exclusive-OR

2.6 EXCLUSIVE-NOR LOGIC

The complement of exclusive-OR is known as exclusive-NOR, which is derived as follows:

Y = (AB′+A′B)′ = (AB′)′(A′B)′ = (A′ + B)(A+B′) = A′B′ + AB

The output Y is 1 only if the two inputs A and B are at the same level. The exclusive-NOR can be implemented by simply inverting the output of an exclusive-OR. The following Figure 2.4 (a) shows

Space for learners:

the exclusive-NOR and Figure 2.4(b) shows the direct implementation of the expression A′B′+AB.

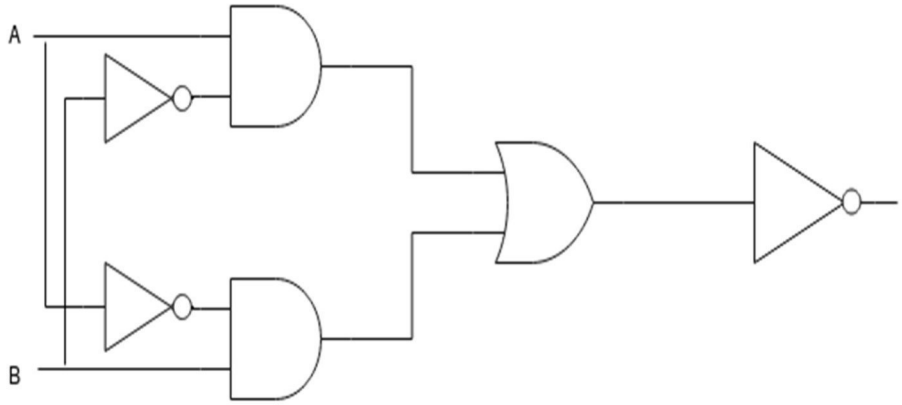


Figure 2.4(a) Y = (AB′+A′B)′

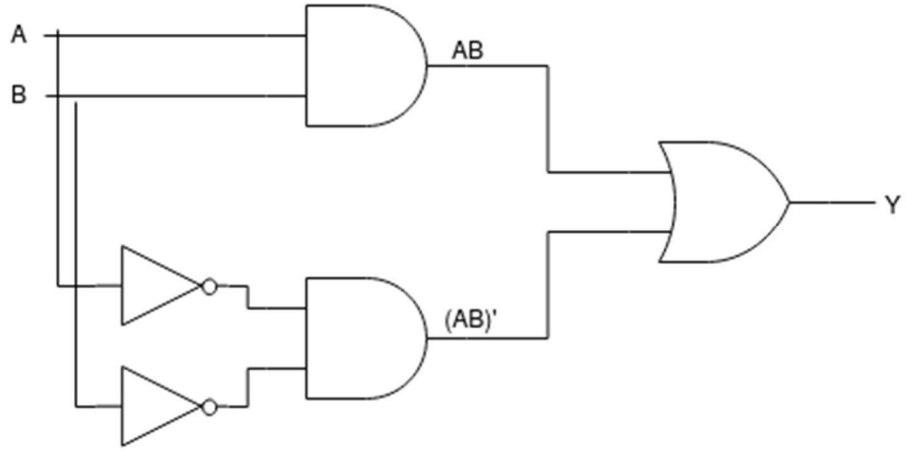


Figure 2.4(b) Y = A′B′ + AB

STOP TO CONSIDER

- Exclusive-OR (XOR) logic is a combination of two AND gates, one OR gate, and two inverters (NOT gate)
- Exclusive-NOR (XNOR) logic is a combination of two AND gates, one OR gate, and three inverters (NOT gate) or XNOR is obtained by applying an inverter at the output of XOR.

2.7 IMPLEMENTING COMBINATIONAL LOGIC

This section will describe the methods of implementing the logic circuits. The first method describes the implementation from the

Space for learners:

Boolean expression and the second method describes the implementation from the truth table.

Space for learners:

2.7.1 Logic circuit design from Boolean expression

Let us consider the following Boolean expression:

$$Y = (A+B)(C+D+E)$$

A closer observation shows that the above expression ′Y′ consists of two terms.

The first term is formed by doing OR operation between A and B, and the second term is formed by doing OR operation among C, D, and E. The two terms are then AND together to produce the final output Y. The OR operations must be performed before the AND operation.

To design the combinational circuit, a 2-input OR gate is required to form the term A+B and a 3-input OR gate is required to form the term C+D+E. A 2-input AND gate is then required to combine the two OR terms. The resulting logic circuit is shown in Figure 2.5.

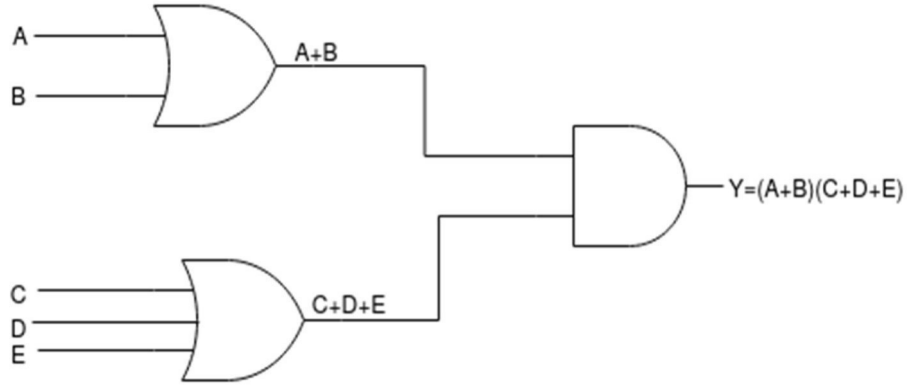


Figure 2.5. Logic circuit for the expression Y =(A+B)(C+D+E)

Let us implement the following expression as another example.

$$Y = (A+B)(C′D′+EF)$$

Like the previous example, let′s have a closer look at the expression. The terms A+B and (C′D′+EF) are AND together to form Y. The term C′D′+EF is first formed by doing AND between C′ and D′, E and F and then performs OR operation between these two terms. Before getting the expression C′D′ +EF, you must have the C′D′ and EF, before these two terms you must have C′ and D′. So, the logic operation must be performed in proper order. The logic gates

required to implement the expression Y = (A+B)(C′D′+EF) are as follows:

a. Two NOT gates to get C′ and D′
b. Two 2-input AND gates to form C′D′ and EF
c. Two 2-input OR gates to form A+B and C′D′+EF
d. One 2-input AND gate to form Y.

The logic circuit of this expression is shown in Figure 2.6

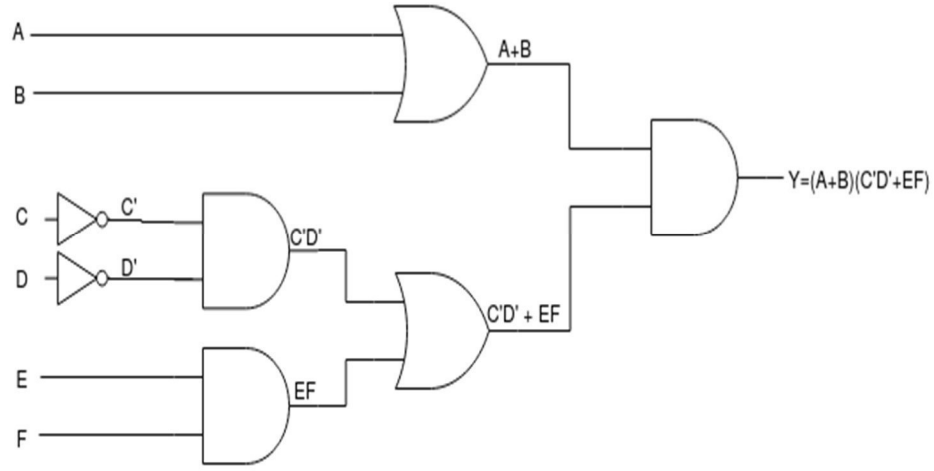


Figure 2.6. Logic diagram for the expression Y = (A+B)(C′D′+EF)

2.7.2. Logic circuit design from truth table

Instead of using the SOP expression to design the combinational circuit you can use the truth table and from the truth table that you can derive using the SOP expression. Table 2.3 shows one example of such an implementation.

Inputs			Output Y	Product Terms
A	B	C		
0	0	0	0	
0	0	1	0	
0	1	0	1	A′BC′
0	1	1	0	
1	0	0	1	AB′C′
1	0	1	0	
1	1	0	0	
1	1	1	1	ABC

Table 2.3: Truth table for logic function

Space for learners:

The Boolean expression obtained for the Table 2.3 is given below:

$$Y = A'BC'+AB'C'+ABC$$

The expression Y is obtained by doing OR operations among the product terms for which the output is 1. The first, second, and third are formed by doing AND operations among (A′, B, C′), (A, B′, C′), and (A, B, C) respectively. The logic gates required to implement the circuit are as follows:

a. Three NOT gates.
b. Three 3-input AND gates.
c. One 3-input OR gate.

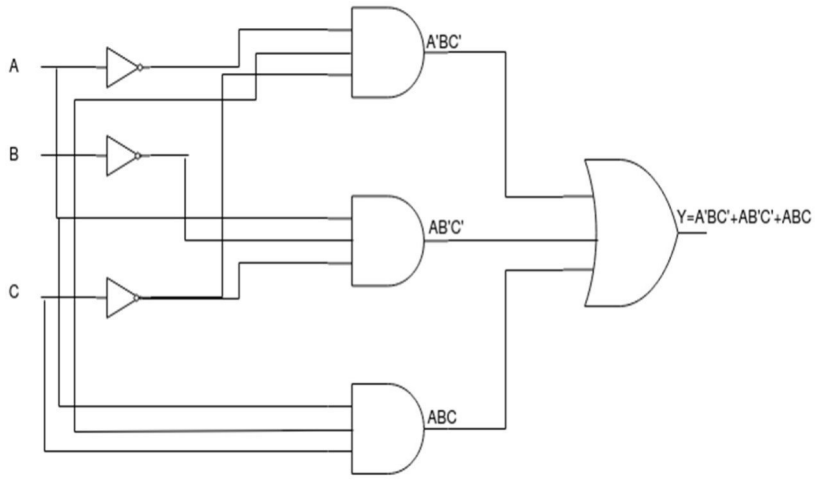


Figure 2.7 Logic diagram for the expression Y = A′BC′+AB′C′+ABC

Reduce the combinational logic circuit shown in Figure 2.8 to a minimum form.

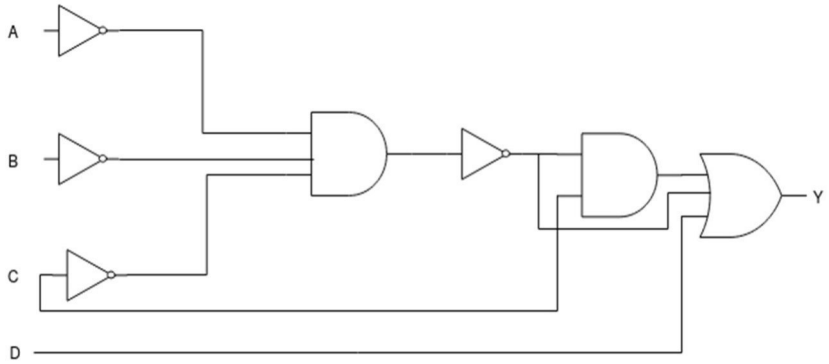


Figure 2.8 Combinational logic circuit to be reduced

The expression for the output of the circuit is Y = (A′B′C′)′C+(A′B′C′)′+D

Space for learners:

Applying D′ Morgan′s theorem and Boolean algebra,

$Y = ((A')'+(B')'+(C')')C+(A')'+(B')'+(C')'+D$

$= AC+BC+CC+A+B+C+D$

$=C(A+B+1)+A+B+C+D$

$Y=A+B+C+D$

The simplified circuit is a 4-input OR gate as shown in the Figure 2.9

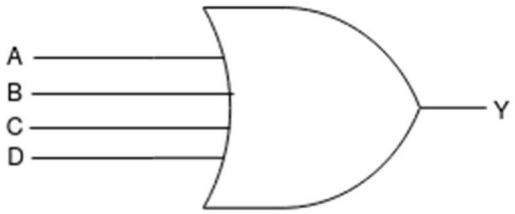


Figure 2.9 Reduced form the logic circuit of Figure 2.7

Note: Before implementing the logic circuit directly it is better to reduce the algebraic expressions to its minimized form so that the number of gates required to implement the circuit is minimum. This leads to reduction of propagation delay among the gates. More the number of gates, the more the propagation delay and also the heat produced by the circuit will increase.

STOP TO CONSIDER

- The implementation of combinational logic circuits is either from the Boolean expression or truth table.
- The expression should be carefully observed and has to identify the number of AND, OR, inverters required.
- Before implementing the logic circuit, it is advisable to reduce the expression by applying the boolean algebra
- If the number of gates are less in the final combinational circuit then the propagation delay will also be minimal.

CHECK YOUR PROGRESS-I

1. POS stands for _____
2. SOP stands for _____
3. An Exclusive-OR can be represented as _____
4. The number of AND gates required to implement the boolean expression ABC is ____

Space for learners:

2.8 THE UNIVERSAL PROPERTY OF NAND AND NOR GATES

Till now, you have studied combinational circuits designing with AND and OR, and NOT gates. This section will describe the universal property of NAND and NOR gates. The universality of NAND means it can be used as an inverter and the combinations of NAND gates can be used to implement the AND, OR, and NOR operations. Similarly, the NOR gate can be used to implement the inverter, AND, OR, and NAND operations.

2.8.1 The NAND gate as a universal logic element

The NAND gate is a universal gate because it can be used to produce the NOT, the AND, the OR, and the NOR functions. An inverter can be made from a NAND gate by connecting all of the inputs together and creating, in effect, a single input as shown in the Figure 2.10(a) for a 2-input gate. An AND function can be generated by the use of NAND gates alone as shown in Figure 2.10(b). An OR function can be implemented with only NAND gates, as shown in Figure 2.10(c). Similarly, the NOR function can also be produced as shown in Figure 2.10(d).

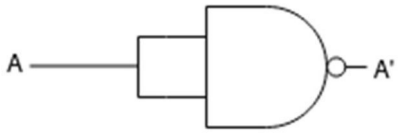


Figure 2.10(a) NAND gate as inverter or NOT

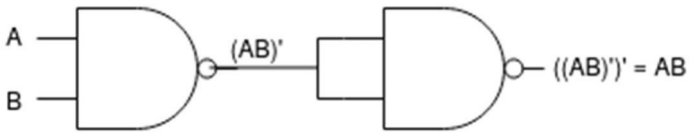


Figure 2.10(b) Two NAND gates are combined to produce AND operation

Space for learners:

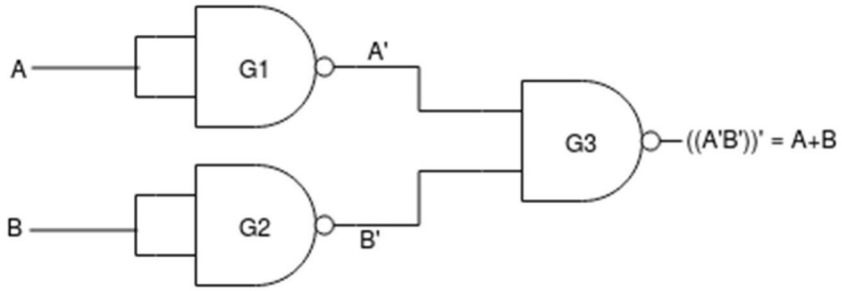


Figure 2.10(c) Three NAND gates are combined to produce OR operation

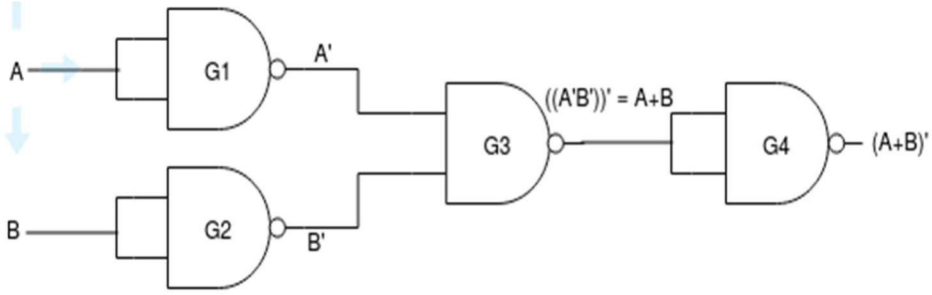


Figure 2.10(d) Four NAND gates are combine to produce NOR operation

In Figure 2.10(b), a NAND gate is used to invert a NAND output to form the AND function which is given below:

$$Y = ((AB)')' = AB$$

In Figure 2.10(c), NAND gates G1 and G2 are combined to invert the two input variables before they are applied to NAND gate G3. The OR gate output is derived as follows by applying DeMorgans′s theorem:

$$Y = ((A'B')' = A + B$$

In Figure 2.10(d), NAND gate G4 is used as an inverter connected to the circuit of part (c) to produce the NOR operation (A+B)′.
Finally, we can conclude that using the NAND gate it is possible to implement any logic function.

2.8.2 The NOR gate as a Universal logic element

The NOR gate can also be used to produce the NOT, AND, OR, and NAND functions. A NOT circuit, or inverter, can be made from NOR gate by connecting all of the inputs together to effectively create a single input, as shown in Figure 2.11(a) with a 2-input example. Also, an OR gate can be produced from NOR gates as shown in Figure 2.11(b). An AND gate can be produced using the

Space for learners:

NOR gates as shown in the Figure 2.11(c), the NOR gates G1 and G2 are used as inverters and the final output is derived by the use of DeMorgan′s theorem as follows:

$$Y = (A'+B')' = AB$$

Figure 2.11(d) shows the implementation of NAND function using NOR gates. Hence we can conclude that the NOR gate can also work as a universal gate like the NAND gate.

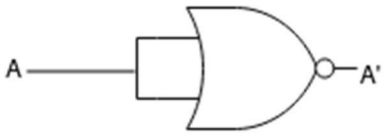


Figure 2.11(a) NOR gate used as inverter

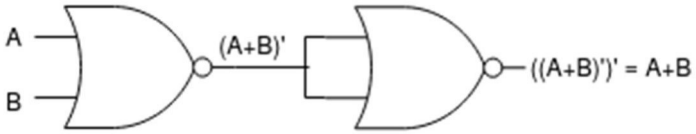


Figure 2.11(b) NOR gates are combined to produce OR operation

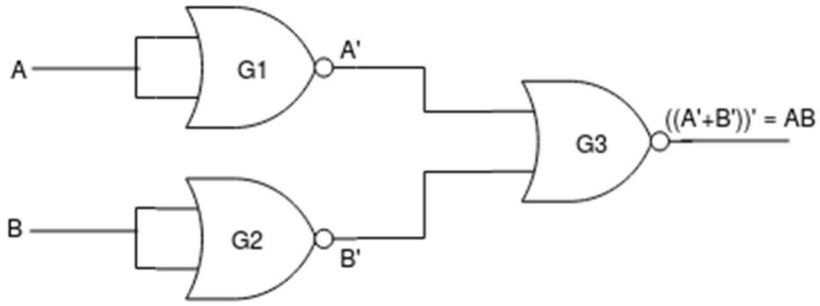


Figure 2.11(c) NOR gates are combined to produce AND operation

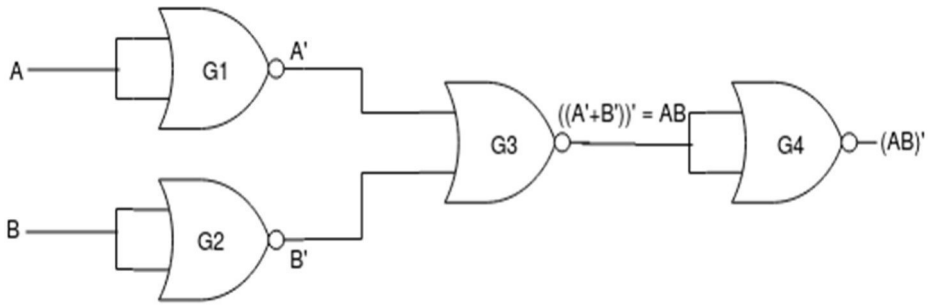


Figure 2.11(d) NOR gates are combined to produce NAND operation

2.8.3 Combinational circuit using NAND gate

NAND gates can work as either NAND or negative OR by applying DeMorgan′s theorem.

$$(AB)' = A' + B'$$

Space for learners:

Consider the NAND logic as shown in Figure 2.12. The output expression is developed in the following steps:

$Y = ((AB)'(CD)')'$
$= ((A'+B')(C'+D'))'$
$= (A'+B')'+(C'+D')'$
$= (A')'(B')' + (C')'(D')'$
$= AB + CD$

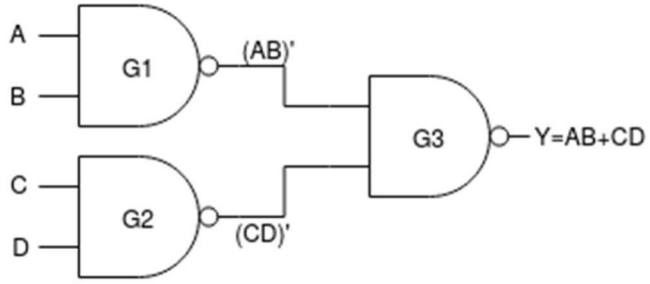


Figure 2.12 Implementation of the Boolean expression Y= AB+CD using NAND gate

2.8.4 Combinational circuit using NOR gate

The NOR gate can work as either a NOR or negative AND, as shown by DeMorgan′s theorem.

$(A+B)' = A'B'$

Consider the NOR logic in Figure 2.13. The output expression is developed as follows:

$Y = ((A+B)'+(C+D)')' = ((A+B)')'((C+D)')' = (A+B)(C+D)$

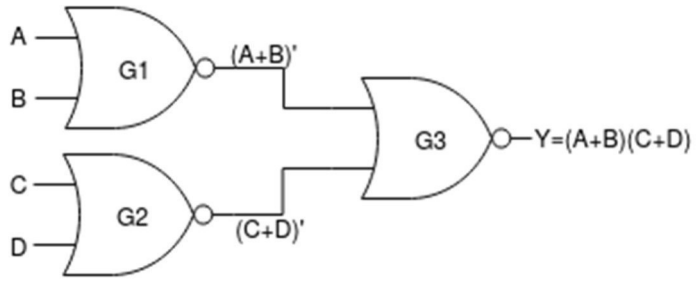


Figure 2.13 Implementation of the boolean expression Y= (A+B)(C+D) using NOR gate

STOP TO CONSIDER

- NAND and NOR gates are called universal gates.
- NAND and NOR can be used to implement all the primary logic like AND, OR, NOT(invert).
- NAND can produce NOR and, similarly, NOR can produce NAND.

Space for learners:

Space for learners:

CHECK YOUR PROGRESS-II

5. The number of NOR gate(s) required to implement OR is/are _____
6. The number of NAND gates(s) required to implement AND is/are _____
7. The number of NAND gates(s) required to implement Y = A′+B is/are _____
8. The number of NOR gates(s) required to implement Y = A′+B is/are _____

2.9 COMBINATIONAL LOGIC CIRCUIT FUNCTIONALITIES

In this section, many types of fixed functions of combinational circuits are introduced, viz. Adders, comparator, decoders, encoders, code converters, multiplexer, demultiplexer etc.

2.9.1 The comparison function

The magnitude of comparison performed by a logic circuit is called a comparator. A comparison compares two quantities and indices whether or not they are equal. Figure 2.14 represents a comparison function, one number in binary form is applied to input A, and the other number in binary form is applied to input B. The outputs indicate the relationship of the two numbers by producing 1 on the proper output line. Suppose that the binary representation of 3 is applied to input A and a binary representation of the number 6 is applied to input B. A ′1′ (HIGH) will appear on the A<B output, indicating the relationship between the two numbers.

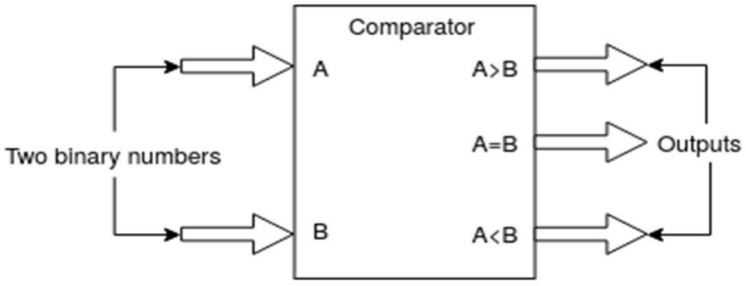


Figure 2.14 Basic magnitude comparator

2.9.2 The Arithmetic functions

Addition is performed by a logic circuit called adder. An adder adds two binary numbers on inputs A and B with a carry input (C_{in}) and generates a sum (Σ) and a carry output (C_{out}) as shown in Figure 2.15.

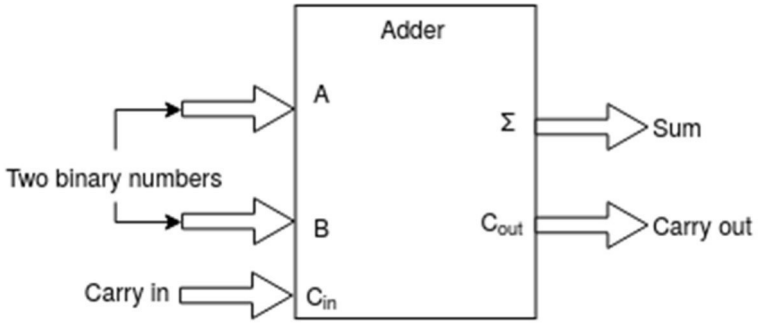


Figure 2.15 Basic Adder

Subtraction is also performed by a logic circuit. A subtractor requires three inputs, viz.,the two numbers that are to be subtracted and a borrow input. The two outputs are the difference and the borrow output. The subtraction operation is a special case of addition operation.

Multiplication: A multiplier is a logic circuit that performs multiplication. Because numbers are always multiplied in twos, two inputs are necessary. The product is the multiplier′s output. Multiplication can be achieved using an adder in conjunction with other circuits since it is merely a series of additions with shifts in the positions of the partial products.

Division: Division can be achieved with a series of subtraction, comparisons, and shifts, therefore an adder can be used in conjunction with other circuits. The division requires two inputs, and the quotient and remainder are generated as outputs.

Code conversion: The logic circuits can also be used for code conversion. A code is a collection of bits arranged in a specific pattern and used to represent data. A code converter converts one type of coded data into another type of coded data. For example, Conversion from binary to Binary Coded Decimal (BCD) or Gray code.

Encoder: The **encoder** is a logic circuit that performs the encoding function. The encoder turns data into a coded representation, such as

Space for learners:

a decimal number or an alphanumeric letter. One form of encoder, for example, turns all of the decimal digits, 0 through 9, to binary code.

Decoder: A logic circuit called a *decoder* performs the decoding operation. The decoder translates coded data, such as binary numbers, to uncoded data, such as decimal numbers. One form of decoder, for example, translates a 4-bit binary code into the appropriate decimal digits.

Data selection function: The multiplexer and the demultiplexer are two types of circuits that select data in the data selection function. A multiplexer, often known as a MUX, is a logic circuit that transfers digital data from many input lines to a single output line in a predetermined time sequence. A multiplexer can be thought of as an electronic switch that links each of the input lines to the output line in a sequential manner. A demultiplexer (DEMUX) is a logic circuit that converts digital data from one input line to multiple output lines in a predetermined order. The demux is a reverse mux. When data from numerous sources needs to be sent across one line to a distant place and then redistributed to multiple recipients, multiplexing and demultiplexing are utilized.

STOP TO CONSIDER

- The AND, OR, and NOT can be used to design the complex logic circuits to perform specific operations.

2.9.3 Basic Adders

Adders are essential not only in computers, but also in a wide range of digital systems that process numerical data. The study of digital systems requires a basic understanding of the adder action. The half-adder and full-adder are described in this section.

2.9.3.1 The Half-Adder

Recall the basic rules for binary addition
0+0 = 0
0+1 = 0

Space for learners:

1+0 = 1
1+1 = 10
A logic circuit known as a half-adder performs the operations.
The half-adder takes two binary digits as inputs and produces two binary digits, a sum bit and a carry bit, as outputs. Figure 2.16 shows a half-adder represented by the logic symbol.

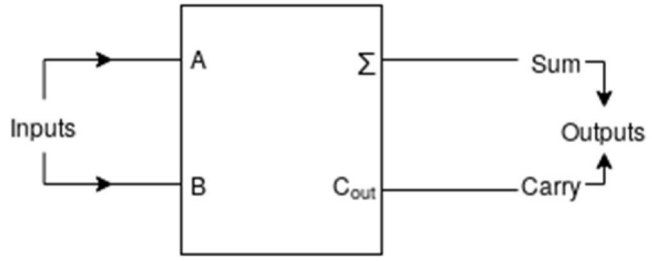


Figure 2.16 Logic symbol for a half-adder

Half-Adder Logic from the operation of the half-adder as stated in Table 2.3, expressions can be derived for the sum and the output carry as functions of the inputs. Note that the output carry (C_{out}) is a 1 only when both A and B are 1s, therefore, C_{out} can be expressed as the AND of the input variables. $C_{out} = AB$.

Table 2.3 Half-adder truth table

A	B	C_{out}	Σ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Now, observe that the sum (Σ) is a 1 only if the input variables, A and B, are not equal. The sum can therefore be expressed as the exclusive-OR of the input variables. $\Sigma = A \oplus B$. The logic implementation required for the half-adder function can be developed using Σ and C_{out}. The output carry is produced with AND gate with A and B on the inputs, and the sum output is generated with an exclusive-OR (XOR) gate, as shown in Figure 2.17. Remember, the XOR is implemented with AND gates, an OR gate, and inverters.

Space for learners:

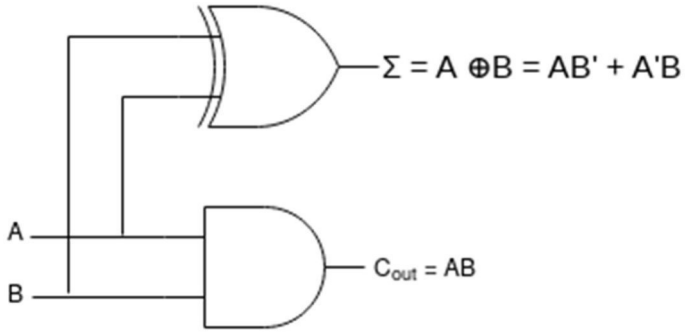


Figure 2.17 Half-adder logic diagram

2.9.3.2 The Full Adder

The second category of adder is the full-adder. The full-adder accepts two input bits and an input carry and generates a sum output and an output carry. The basic difference between full-adder and a half-adder is that the full-adder accepts an input carry. A logic symbol for a full-adder is shown in Figure 2.18, and the truth table in Table 2.4 shows the operation of a full-adder.

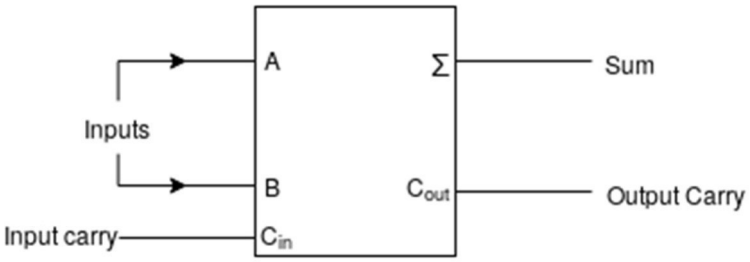


Figure 2.18 Logic symbol for a full-adder

Table 2.4 Full-adder truth table

A	B	C_{in}	C_{out}	Σ
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Space for learners:

Full-Adder Logic The full-adder must add the two input bits and input carry. From the half-adder you know that the sum of the input bits A and B is exclusive-OR of those variables. A⊕B. For the input carry (C_{in}) to be added to the input bits, it must be exclusive-ORed with A⊕B, yielding the equation for the sum output of the full-adder.

$\Sigma = (A \oplus B) \oplus C_{in}$. This means that to implement the full-adder sum function, two 2-input exclusive-OR gates can be used. The first must generate the term A⊕B, and the second has as its inputs the output of the first XOR gate and the input carry, as shown in Figure 2.19(a).

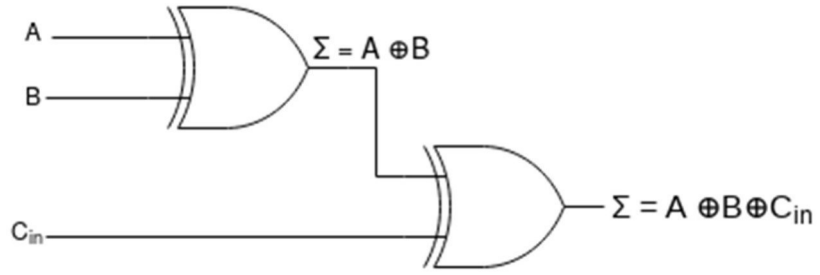


Figure 2.19(a) Logic required to form the sum of three bits

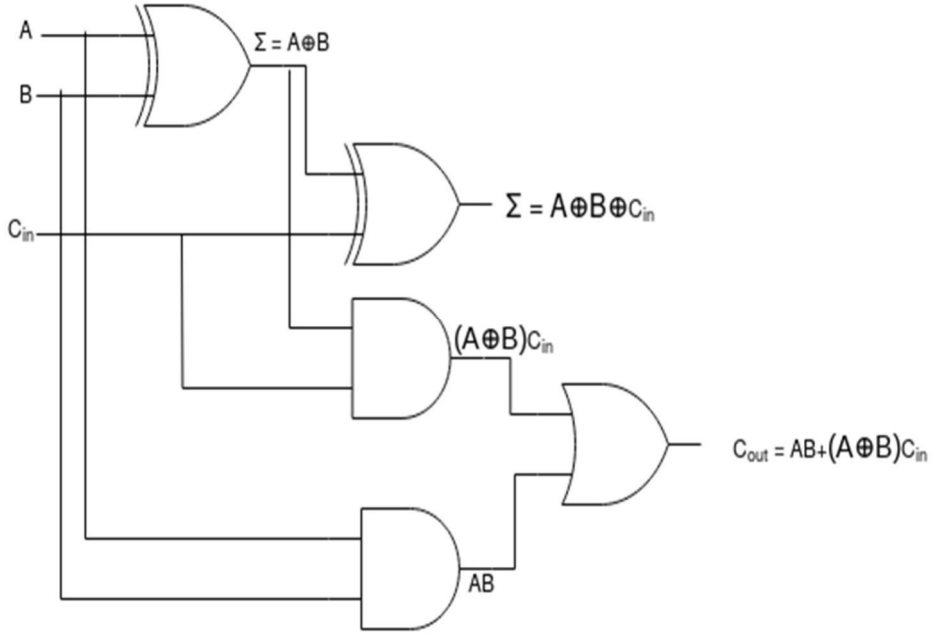


Figure 2.19(b) Complete logic circuit for a full-adder

The output carry is a 1 when both inputs to the first XOR gate are 1s or when both inputs to the second XOR gate are 1s. You can verify this fact by studying Table 2.4. The output carry of full-adder is therefore produced by the inputs A ANDed with B and A⊕B

Space for learners:

ANDed with C_{in}. These two terms are ORed, as expressed in the expression of C_{out}. This function is implemented and combined with the sum logic to form a complete full-adder circuit, as shown in Figure 2.19(b). Notice that in Figure 2.19(b) there are two half-adders, connected as shown in the block diagram of Figure 2.20(a), with their output carries ORed. The logic symbol shown in Figure 2.20(b) will normally be used to represent the full-adder.

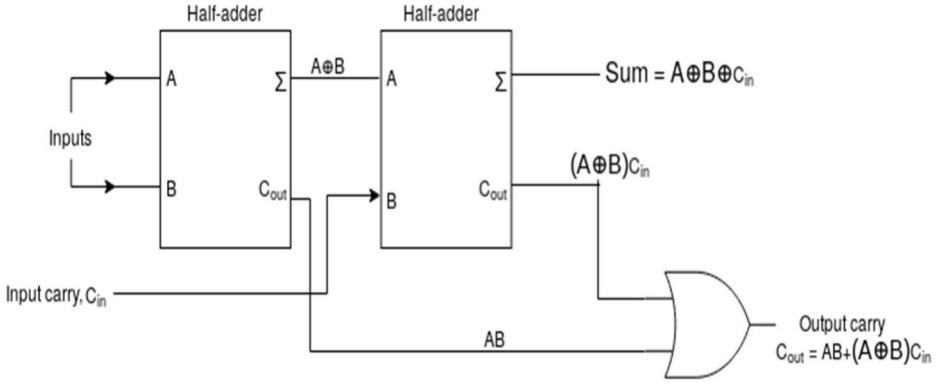


Figure 2.20(a) Arrangement of two half-adders to form a full-adder

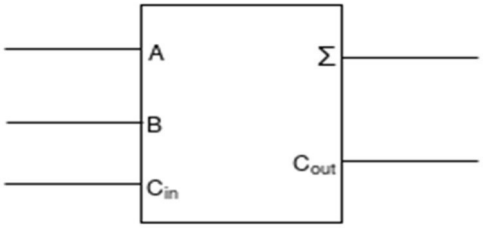


Figure 2.20(b) Full-adder logic symbol

2.9.3.3 Parallel Binary Adders

Parallel binary adders are formed by connecting two or more full-adders. The basic operations of such adders, as well as their associated input and output functions, are described in this section. A single full-adder can add two one-bit numbers as well as an input carry. Additional full-adders must be used to add binary numbers with more than one bit. As shown above with 2-bit integers, when one binary number is added to another, each column creates a sum bit and a 1 or 0 carry bit to the next column to the left.

10
+10

100

Space for learners:

Space for learners:

In this case, the second column′s carry bit becomes the third column′s sum bit. A full adder is required for each bit in two binary numbers to be added. So two adders are required for 2-bit numbers, four adders are required for 4-bit values, and so on. Each adder′s carry output is connected to the next higher-order adder′s carry input, as shown in Figure 2.21 for a 2-bit adder. Because there is no carry input to the least significant bit location, either a half-adder or the carry input of a full-adder can be set to 0 (grounded). In Figure 2.21 the least significant bits (LSB) of the two numbers are represented by A_1 and B_1. The next higher-order bits are represented by A_2 and B_2. The three sum bits are Σ_1, Σ_2, and Σ_3. Notice that the output carry from the left-most full-adder becomes the most significant bit (MSB) in the sum Σ_3.

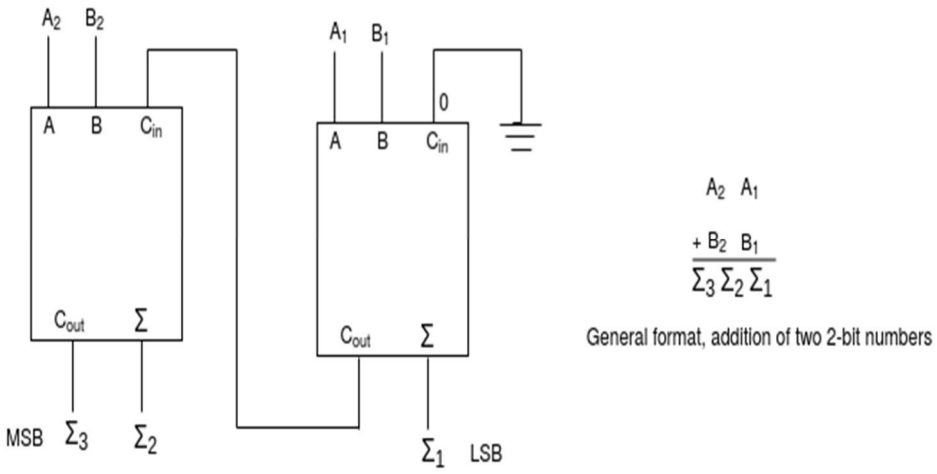


Fig 2.21. A 2-bit adder

Four-bit Parallel Adders

A nibble is a collection of four bits. As demonstrated in Figure 2.22, a basic 4-bit parallel adder is developed with four full-adder stages. The LSBs (A_1 and B_1) of each number being added are applied to the right-most full-adder; the higher-order bits are applied to the gradually higher-adders as illustrated; and the MSBs (A_4 and B_4) of each number are applied to the left-most full-adder. The carry output of each adder is connected to the carry input of the next higher-order adder as indicated. These are called internal carries.

In terms of the method used to handle carries in a parallel adder, there are two types: the *ripple carry* adder and *carry look-ahead* adder. A *ripple carry* adder is one in which the carry output of each full-adder is connected to the carry input of the next higher-order stage (a stage is one full-adder). The sum and the output carry of any stage cannot be produced until the input carry occurs. This causes a time delay in the addition process. The carry propagation delay for each full-adder is the time from the application of the input carry until the output carry occurs, assuming that the A and B inputs are already present.

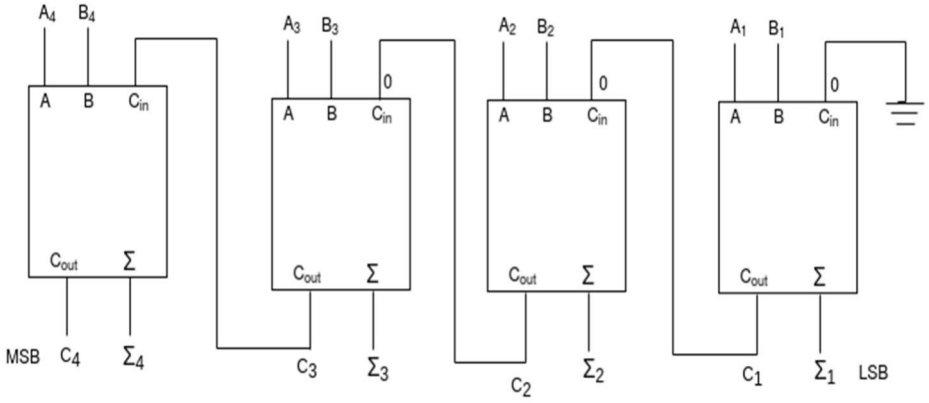


Figure 2.22 A 4-bit Adders

Look-ahead carry addition is a technique for speeding up the addition process by eliminating the *ripple carry* delay. The *look-ahead carry* adder predicts each stage′s output carry and produces it using either *carry generation* or *carry propagation* based on the input bits of each stage.

Carry generation occurs when an output carry is produced (generated) internally by the full-adder. A carry is generated only when both input bits are 1s. The generated carry, C_g, is expressed as the AND function of the two input bits, A and B. C_g=AB.

Carry Propagation occurs when the input carry is rippled to become the output carry. An input carry may be propagated by the full-adder when either or both of the input bits are 1s. The propagated carry, Cp, is expressed as the OR function of the input bits. C_p = A + B.

2.9.3.4 Truth table for 4-bit parallel adder

Table 2.5 is the truth table for a 4-bit adder. On Some data sheets, truth tables may be called *function tables or functional truth tables*. The subscript *n* represent the adder bits and can be 1, 2, 3, or 4 for

Space for learners:

the 4-bit adder. C_{n-1} is the carry from the previous adder. Carries C_1, C_2, and C_3 are generated internally. C_0 is an external carry input and C_4 is an output.

Table 2.5 Truth table for 4-bit parallel adder

C_{n-1}	A_n	B_n	Σ_n	C_n
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

STOP TO CONSIDER

- A half-adder has two inputs and two outputs.
- A full-adder has three inputs and two outputs.
- A 4-bit parallel adder can add two 4-bit binary numbers.
- Two half-adders can be used to design a full adder.

CHECK YOUR PROGRESS-III

State whether true or false

9. The sum expression for a half adder is A+B
10. The carry out C_{out}expression for a full adder is $AB+C_{in}$
11. A 4-bit parallel adder has four full adders.
12. There are two types of carry, they are ripple carry and look ahead carry
13. Carry generation occurs when an output carry is produced.

2.9.4 Binary Subtractor

Binary subtractors are special circuits which subtract two binary numbers from each other. Binary subtractor produces a difference and borrow output after the completion of the subtraction operation. Binary subtraction has two digits, subtracting a "0" from a "0" or a "1" leaves the result unchanged as 0-0 = 0 and 1-0 = 1. Subtracting

Space for learners:

a “1” from a “1” results in a “0”, but subtracting a “1” from a “0” requires a borrow. In other words, 0-1 requires a borrow and if you borrow 1 then the minuend 0 becomes 10 and the operation 0-1 becomes 10-1 which will give the output 1, this also leads to set the borrow bit 1. The half-subtractor and full-subtractor are discussed below.

Space for learners:

2.9.4.1 The Half-Subtractor

A half subtractor is a logical circuit that performs a subtraction operation on two binary digits. The half subtractor produces a difference (D) and a borrow out (B_{out}) bit for the next stage. The Figure 2.23 shows the logic symbol of a half-subtractor circuit.

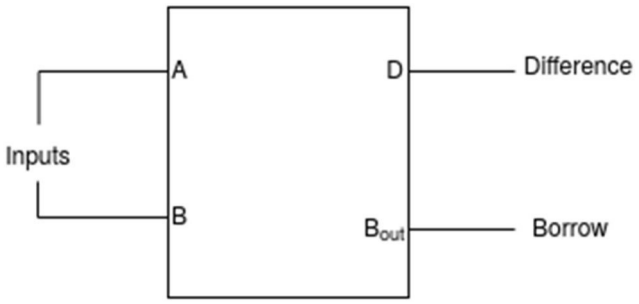


Figure 2.23 Logic symbol of Half-Subtractor

Table 2.6: Truth Table of a Half-Subtractor

Inputs		**Outputs**	
A	**B**	**Difference(D=A-B)**	**Borrow (B_{out})**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

From the Table 2.6 of the half subtractor, the difference (D) output can be obtained by doing exclusive-OR between A and B and the Borrow (B_{out}) can be obtained by doing AND operation between A′ and B. The Boolean expression for a half subtractor is as follows.

$$D = A \oplus B$$
$$B_{out} = A'B$$

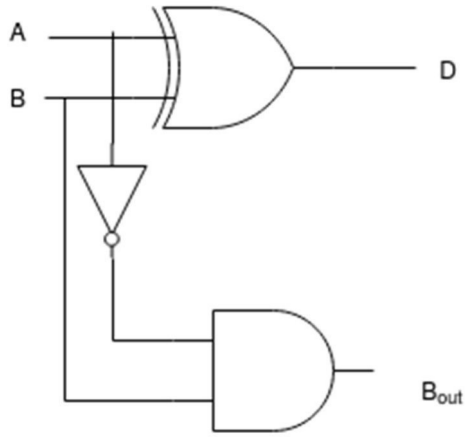


Figure 2.24 Logic circuit for Half-subtractor

The Boolean expressions for 'sum' in half-adder and 'difference' in half-subtractor are exactly the same. The only difference is the output carry of the half-adder and the borrow out of the subtractor circuit, difference between these two quantities is the inversion of the minuend input A.

The disadvantage of the half-subtractor is that if you subtract multiple bits there is no option for 'borrow-in' from its earlier stages. So, we need a full subtractor circuit to take into account this borrow-in input from the earlier stages.

2.9.4.2 The Full-Subtractor

The full-subtractor has three inputs. The two single bit data inputs A (minuend) and B (subtrahend) are the same as before plus an additional *Borrow-in* (B_{in}) input to receive the borrow generated by the subtraction process from a previous stage as shown in Figure 2.25.

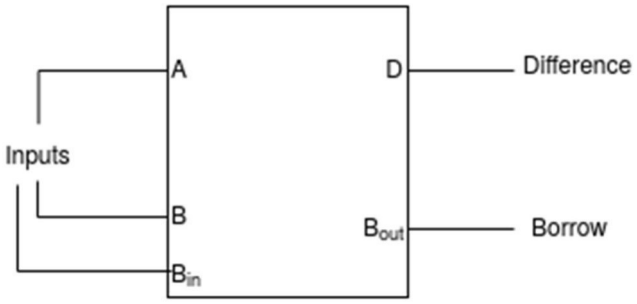


Figure 2.25 : Logic symbol of a Full-Subtractor

Space for learners:

Space for learners:

The "full subtractor" combinational circuit performs the subtraction operation on three binary bits resulting in outputs for the difference D and borrow B_{out}. Like the adder circuit, the full subtractor can also be thought of as two half subtractors connected together, with the first half subtractor passing its borrow to the second half subtractor as shown in the Figure 2.26 and the complete logic circuit of the full-subtractor is shown in the Figure 2.27.

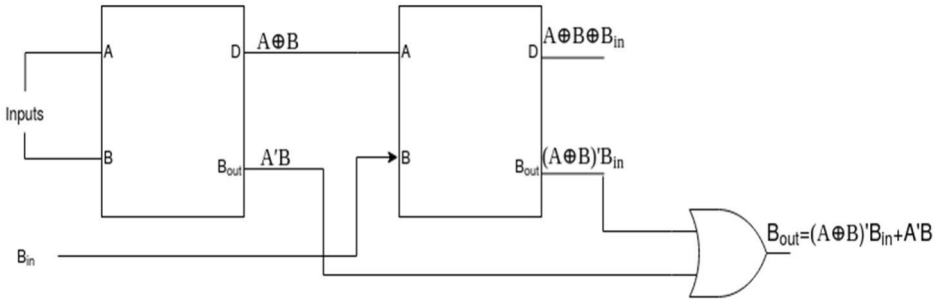


Figure 2.26: Arrangement of two half-subtractors to form a full-subtractor

Table 2.7: Truth for the Full-Subtractor

Inputs			**Outputs**	
A	**B**	**B_{in}**	**D**	**B_{out}**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

The truth table Table 2.7 shows the subtraction operation between A and B, the truth table operations are explained below:

a. If A = 0, B = 0, and B_{in}= 0, then the output D and B_{out} both are 0.

b. In the second set of inputs the A = 0, B = 0, but the B_{in} =1, so before performing the subtraction operation, first, you have to increment the B by 1 unit then the B will change to 1

(B_{in}=1 indicates there was a borrow in the *previous step* of the series of operations here *previous step* is not referring the first set of input operation of the Table 2.7), now if you perform the A-B i.e. 0-1 you need a borrow then only the operation will be possible, so, if you borrow 1 then the A will change to 10 and the subtraction operation 10-1 will give 1, i.e. D=1, since the operation was performed using borrow, the B_{out}=1.

c. In the third set of inputs A=0, B=1, B_{in}= 0, since B_{in}= 0, so, it is not necessary to increment the B, but A-B i.e. 0-1 in this step needs a borrow, so you have to borrow 1 to A, the A will change to 10 and the operation will change to 10-1=1, so, D=1, B_{out}=1.

d. In the fourth set of inputs A=0, B=1, B_{in}= 1, this time B_{in}= 1, so, you have to increment the B by one unit, then B will change to 10, now, if you perform the subtraction operation A-B, i.e. 0-10 then you need a borrow, if you borrow 1 the A will change to 10 and the operation becomes 10-10=0, therefore D=0, since the operation was completed using a borrow so, B_{out}= 1.

e. In the fifth set of inputs A=1, B=0, B_{in}= 0, since B_{in}= 0, so the B will not change and the A-B i.e. 1-0 does not need any borrow therefore D=1 and B_{out}= 0.

f. In the sixth set of inputs A=1, B=0, B_{in}= 1, since B_{in}= 1, so, B will be incremented by 1 unit and B becomes 1 i.e. B=1 and the operation A-B will be 1-1=0, hence D=0 and B_{out}= 0.

g. In the seventh set of inputs A=1, B=1, B_{in}= 0, since B_{in}= 0, so, B will not change and the A-B i.e. 1-1 does not need any borrow therefore D=0 and B_{in}= 0.

h. In the eighth set of inputs A=1, B=1, B_{in}= 1, since B_{in}= 1, so, B will be incremented by 1 unit and the value of B becomes 10, now, the operation A-B becomes 1-10 which is not possible therefore A needs borrow to complete the operation, after 1 borrow to A the A becomes 11 and A-B = 11-10. Hence D=1 and B_{out}= 1.

Space for learners:

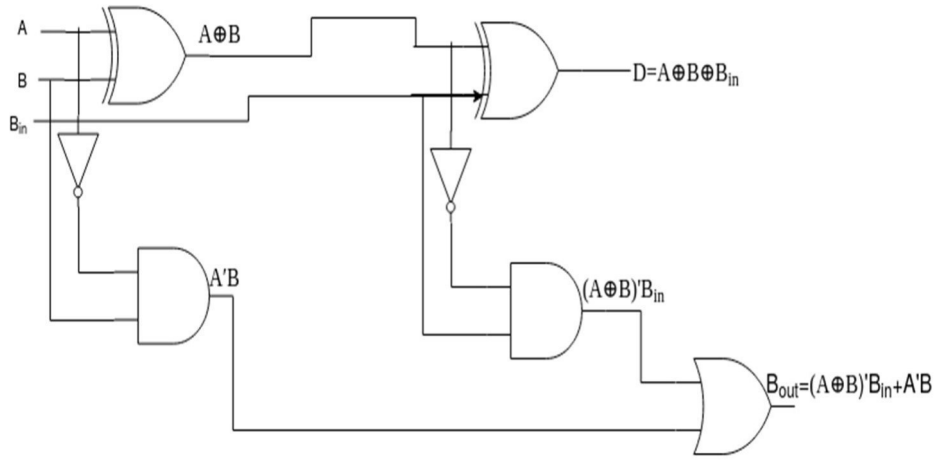


Figure 2.27: Complete logic circuit for a full-subtractor

STOP TO CONSIDER

- Subtractor circuits are similar to the adder circuits.
- Subtraction operation in binary works in the same pattern that works in normal mathematics.
- The difference expression of the subtractor circuit is the same as the sum expression of the adder circuit.
- The Full-subtractor supports borrow in from the previous stages.
- B_{in}= 1 indicates there is a borrow in the previous step.
- If A<B+B_{in}then A needs borrow from its

2.9.5 Comparators

A comparator′s primary role is to compare the magnitudes of two binary quantities in order to identify their relationship. A comparator circuit, in its most basic form, determines if two integers are equal.

2.9.5.1 Equality

Because its output is 1 when the two input bits are not equal and 0 when they are equal, the exclusive-OR gate can be used as a basic comparator. As a 2-bit comparator, Figure 2.28 depicts the exclusive-OR gate.

Space for learners:

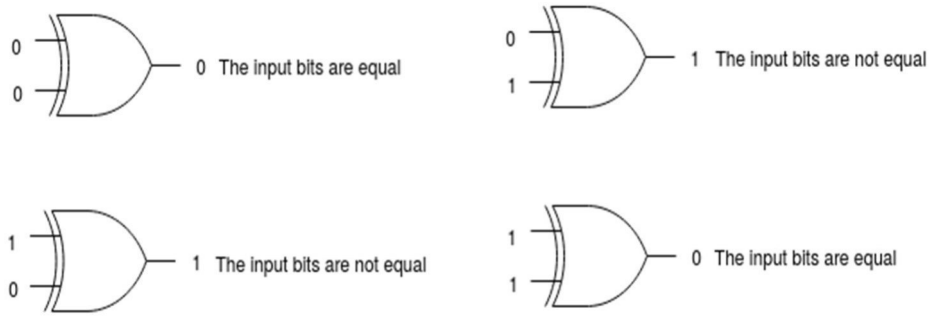


Figure 2.28 Basic comparator operation

An additional exclusive-OR gate is required to compare binary values comprising two bits each. Gate G_1 compares the two numbers′ least significant bits (LSBs), while gate G_2 compares the two most significant bits (MSBs), as seen in Figure 2.29. If the two numbers are equal, their corresponding bits are also equal, and each exclusive-OR gate′s output is a 0. If the corresponding sets of bits are not equal, the exclusive-OR gate output is set to 1.

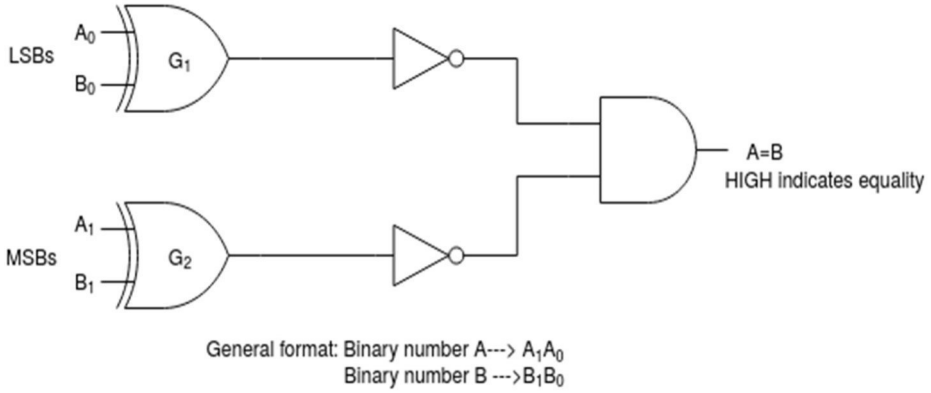


Figure 2.29 2-bits binary number comparison

Two inverters and an AND gate can be used to produce a single output representing the equality or inequality of two values, as shown in Figure 2.29. Each exclusive-OR gate′s output is inverted and applied to the AND gate′s input. When each exclusive-two OR′s input bits are identical. The numbers′ corresponding bits are equal, resulting in a 1 on both AND gate inputs and a 1 on the output. When the two numbers are not equal, one or both sets of corresponding bits are different, and a 0 appears on at least one AND gate input, resulting in a 0 on the AND gate′s output. As a result, the AND gate′s output indicates whether the two numbers are equal (1) or unequal (0).

Space for learners:

2.9.5.2 Inequality

Many IC comparators have additional outputs in addition to the equality output that show which of the two binary integers being compared is greater. As indicated in the logic symbol for a 4-bit comparator in Figure 2.30, there is an output that indicates when number A is larger than number B (A>B) and an output that indicates when number A is smaller than number B (A<B).

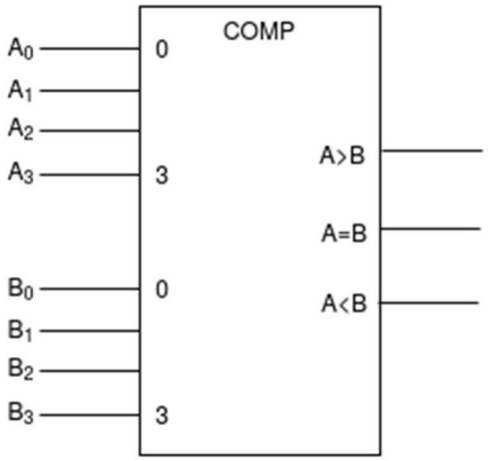


Figure 2.30 Logic symbol for a 4-bit comparator

To determine an inequality of binary numbers A and B, you first examine the highest-order bit in each number. The following conditions are possible:

1. If A_3=1 and B_3=0, number A is greater than number B.
2. If A_3=0 and B_3=1, number A is less than number B.
3. If A_3= B_3, then you must examine the next lower bit position for an inequality.

These three operations are valid for every bit position in the number. The general procedure used in the comparator is to check the inequality of the bit position, starting from the most significant bit (MSB). When such an inequality is found, the relationship of the two numbers is established, and any other inequalities in lower-order bit positions must be ignored because it is possible for an opposite indication to occur; the highest-order indication must take precedence.

CHECK YOUR PROGRESS-IV

State whether true or false

14. The exclusive-OR gate is a basic comparator.
15. The HIGH output will appear if we compare 11_2 and 11_2.
16. LSB stands for Least Significant Bit.

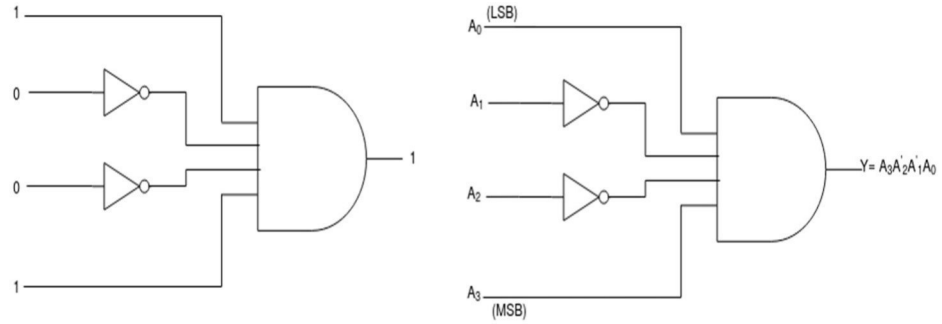
Space for learners:

2.9.6 Decoders

A decoder′s basic task is to identify the presence of a specific combination of bits (code) on its inputs and to signify that presence with a specific output level. A decoder contains n input lines to handle *n* bits and 1 to 2^n output lines to signal the presence of one or more *n*-bit combinations in its most basic form.

2.9.6.1 The Basic Binary Decoder

Assume you need to figure out when a binary 1001 appears on a digital circuit′s inputs. Because it provides a HIGH output only when all of its inputs are HIGH, an AND gate can be utilized as the basic decoding element. As a result, when the binary number 1001 occurs, you must ensure that all of the AND gate′s inputs are HIGH; this may be done by inverting the two middle bits (the 0s) as shown in Figure 2.31.



(a) (b)

Figure 2.31 Binary decoder

The logic equation for the decoder of Figure 2.31(a) is developed as illustrated in Figure 2.31(b). You should verify that the output is 0 except when A_0=1, A_1=0, A_2=0, and A_3=1 are applied to the inputs. A_0 is the LSB and A_3 is the MSB. In the representation of a binary number, the LSB is the right-most bit in a horizontal arrangement and the top-most bit in a vertical arrangement, unless specified otherwise. If the NAND gate is used in place of the AND gate in Figure 2.31, a LOW output will indicate the presence of the proper binary code, which is 1001 in this case.

Space for learners:

2.9.6.2 3 to 8 line Decoder

Figure 2.32 shows a decoder circuit with three inputs and $2^3 = 8$ outputs. It uses all AND gates, so the output is active high. Please note that for a given input code, the only valid output (HIGH) is the output corresponding to the decimal equivalent of the binary input code (for example, only when CBA = $101_2 = 5_{10}$, the O_5 output will become HIGH). The decoder can be referenced in various ways. It can be called a 3-8 line decoder because it has 3 input lines and 8 output lines. It can also be called a binary octal decoder or converter because it takes a 3-digit binary input code and activates one of the eight (octal) outputs corresponding to that code. It is also called a 1 of 8 decoder because only 1 of the 8 outputs is activated at a time.

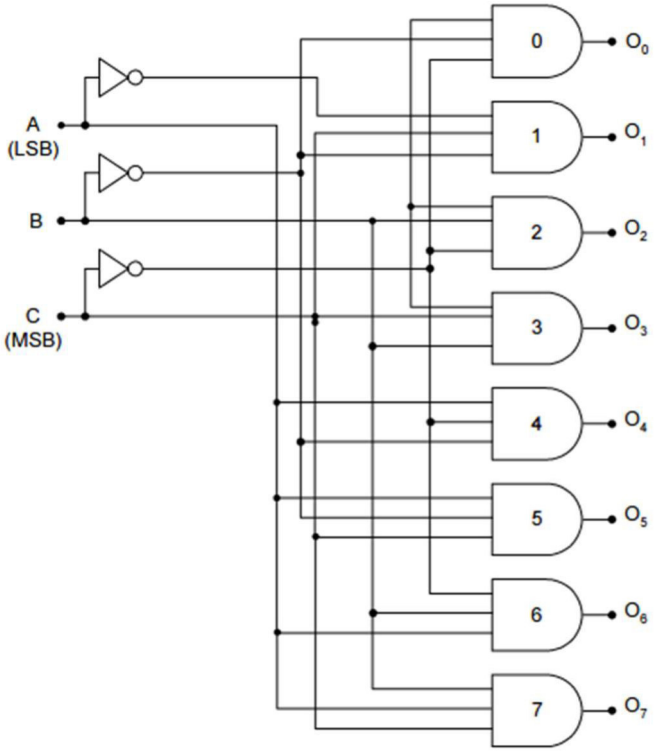


Figure 2.32 3-to-8-line decoder

Table 2.8: 3-to-8-line Decoder truth table with decoding function

Inputs			Decoding Function	Outputs							
C	B	A		O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
0	0	0	C′B′A′	1	0	0	0	0	0	0	0
0	0	1	C′B′A	0	1	0	0	0	0	0	0
0	1	0	C′BA′	0	0	1	0	0	0	0	0
0	1	1	C′BA	0	0	0	1	0	0	0	0

Space for learners:

1	0	0	CB′A′	0	0	0	0	1	0	0	0
1	0	1	CB′A	0	0	0	0	0	1	0	0
1	1	0	CBA′	0	0	0	0	0	0	1	0
1	1	1	CBA	0	0	0	0	0	0	0	1

CHECK YOUR PROGRESS-V

17. An n-bit decoder can have ____ output lines
18. Determine the logic expression for the input 0111 by producing HIGH level on the output

2.9.7 Encoders

An encoder is essentially a combinatorial logic circuit that does the opposite of a decoder. An encoder accepts an active level from one of its inputs representing a number such as decimal or octal and converts it to a coded output such as BCD and binary. You can devise an encoder for encoding various symbols or characters. The process of converting the familiar symbols and numbers into a coded form is called encoding.

2.9.7.1 Decimal to BCD Encoder

As shown in Figure 2.33, this form of encoder has 10 inputs, one for each decimal digit, and four outputs that correspond to the BCD code. This is a simple ten-to-four line encoder.

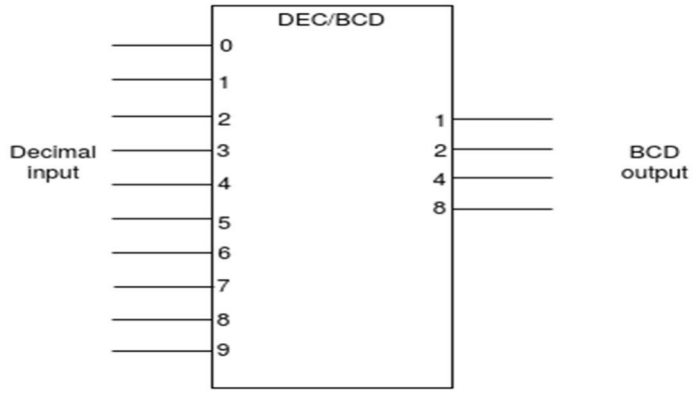


Figure 2.33 Logic symbol for a decimal to BCD encoder

Space for learners:

Space for learners:

Table 2.7 lists the BCD (8421) code. To evaluate the logic, you can use this table to explore the relationship between each BCD bit and the decimal digits. For instance, the most significant bit of the BCD code, A_3, is always a 1 for decimal digit 8 or 9. An OR expression for bit A_3 in terms of the decimal digits can therefore be written as $A_3 = 8+9$. Bit A_2 is always a 1 for decimal digit 4, 5, 6 or 7 can be expressed as an OR function as follows:

$A_2 = 4+5+6+7$

Bit A_1 is always 1 for decimal digit 2, 3, 6, or 7 and can be expressed as

$A_1 = 2+3+6+7$

Finally, A_0 is always a 1 for decimal digits 1, 3, 5, 7, or 9. The expression for A_0 is

$A_0 = 1+3+5+7+9$

Table 2.9: Decimal to BCD encoder truth table

Decimal Digit	**BCD code**			
	A_3	**A_2**	**A_1**	**A_0**
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

Let us now use the logic expression we just generated to implement the logic circuitry required for encoding each decimal digit to a BCD code. Each BCD output is easily formed by ORing the relevant decimal digit input lines. Figure 2.34 depicts the basic encoder logic that results from these expressions. The circuit in Figure 2.34 has the following fundamental operation: The appropriate levels are displayed on the four BCD output lines when HIGH occurs on one of the decimal digit input lines. If input line 9 is HIGH (and all other input lines are LOW), for instance, this

condition will result in HIGH on outputs A_0 and A_3 and LOW on outputs A_1 and A_2, which is the BCD code (1001) for decimal 9.

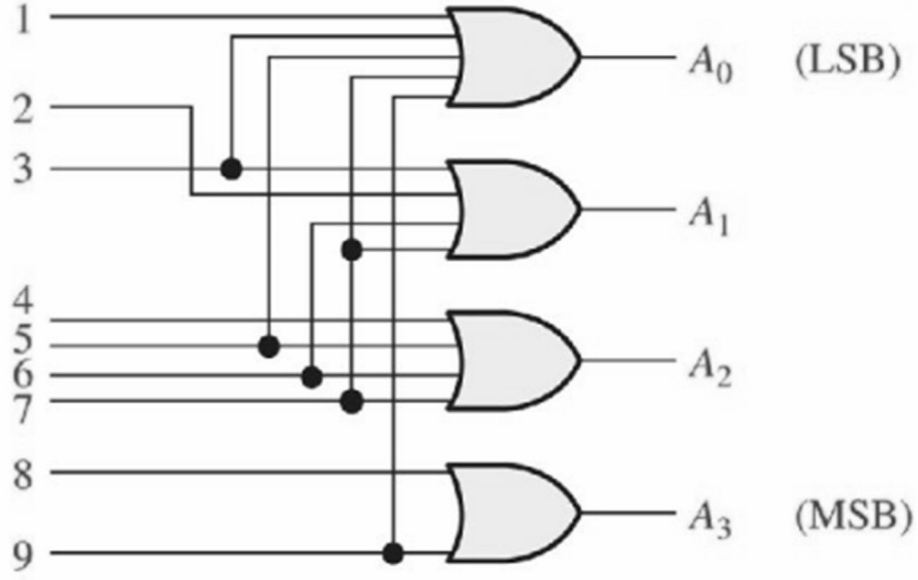


Figure 2.34 BCD encoder logic circuit

STOP TO CONSIDER

- Encoders perform the reverse operation of the decoders
- Encoders convert familiar symbols or numbers to coded forms.
- An encoder having 2^n input lines in the input will have n output lines in the output.

2.9.8 Multiplexers

The multiplexer (MUX) is a device that allows digital information from multiple sources to be routed to a single line for transmission over that line to a common destination. The basic multiplexer has several data input lines and one output line. It also has a data select input, allowing you to change digital data from any input to the line out. The multiplexer is also called a data selector.

Figure 2.35 shows a logic symbol for a 4-input multiplexer (MUX). Because there are two data select lines, any one of the four data input lines can be picked with two select bits.

Space for learners:

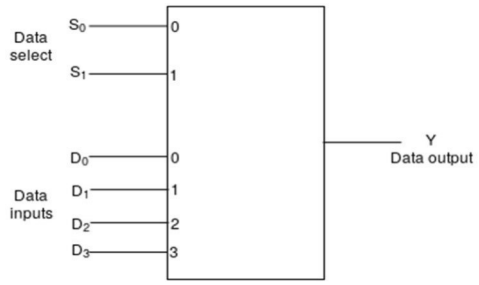


Figure 2.35 Logic symbol of a 4-input multiplexer

In Figure 2.35, a 2-bit code on the data-select (S) inputs will allow the data on the selected data input to pass through to the data output. If a binary 0 (S_1=0 and S_0=0) is applied to the data-select lines, the data on input D_0 appear on the data-output line. If a binary 1 (S_1=0 and S_0=1) is applied to the data-select lines, the data on input D_1 appear on the data output. If a binary 2 (S_1=1 and S_0=0) is applied, the data on D_2 appear on the output. If a binary 3 (S_1=1 and S_0=1) is applied, the data on D_3 are switched to the output line. A summary of this operation is shown in Table 2.8.

Table 2.10: 4 to 1 line multiplexer truth table

Data select inputs		**Input selected**
S_1	**S_0**	
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Let′s have a look at the logic circuits that this multiplexing process requires. The status of the selected data input is replicated in the data output. As a result, you can construct a logic expression for the output from the data input and the inputs you choose.

The data output is equal to D_0 only if S_1=0 and S_0=0; $Y = D_0S'_1S'_0$
The data output is equal to D_1 only if S_1=0 and S_0=1; $Y = D_1S'_1S_0$
The data output is equal to D_2 only if S_1=1 and S_0=0; $Y = D_2S_1S'_0$
The data output is equal to D_3 only if S_1=1 and S_0=1; $Y = D_3S_1S_0$

When these terms are ORed, the total expression for the data output is

Space for learners:

$Y = D_0S'_1S'_0 + D_1S'_1S_0 + D_2S_1S'_0 + D_3S_1S_0$

The implementation of this equation requires four 3-input AND gates, a 4-input OR gate, and two inverters to generate the complements of S_1 and S_0 as shown in Figure 2.36. Because data can be selected from any one of the input lines, this circuit is also referred to as a *data selector*.

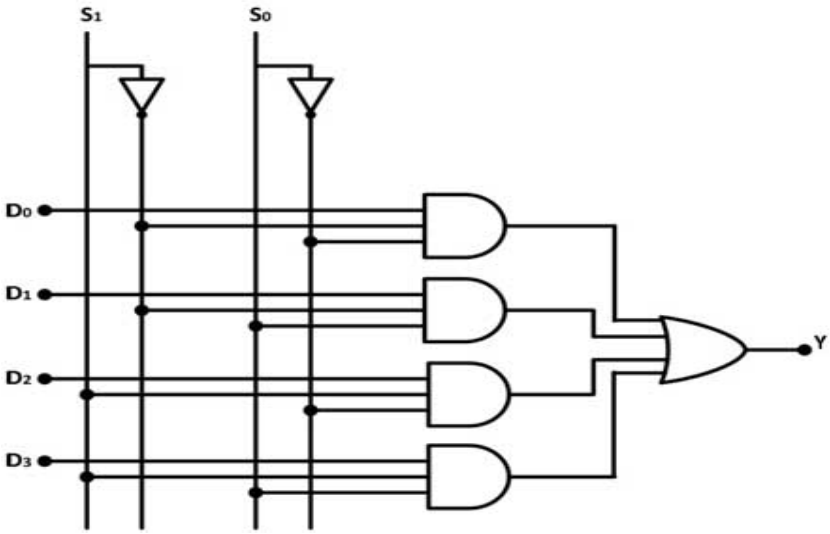


Figure 2.36 Circuit diagram of 4-to-1 multiplexer

STOP TO CONSIDER

- Multiplexers are also known as data selectors.
- A 4-input data lines multiplexer has two select lines.
- A 8-input data lines multiplexer has three select lines.
- A 2^n input data lines multiplexer has n select lines.
- The output depends on the input data and select lines bits.

2.9.9 Demultiplexer

The demultiplexer (DEMUX) basically reverses the multiplexing function. It takes digital information from one line and distributes it to a specified number of output lines. Therefore, the demultiplexer is also called a data distributor. As you will learn, the decoder can also be used as a demultiplexer.

Space for learners:

A 1-to-4-line demultiplexer (DEMUX) circuit is shown in Figure 2.37. The data-input line is connected to all AND gates. Only one gate is enabled at a time by the two data-select lines, and data on the data-input line passes via the selected gate to the associated data output line.

Space for learners:

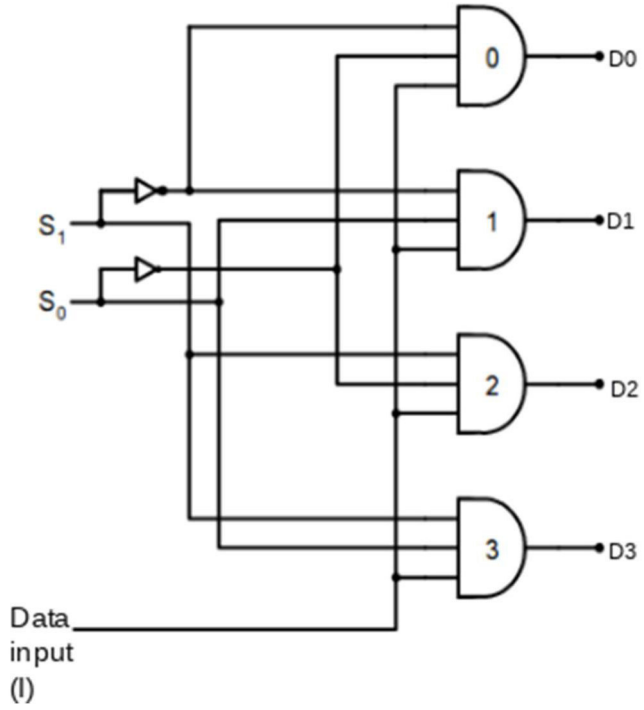


Figure 2.37 Circuit diagram of a 1-to-4 line demultiplexer

Table 2.11: 1 to 4 line demultiplexer truth table

Select code		**Outputs**			
S_1	**S_0**	**D_3**	**D_2**	**D_1**	**D_0**
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0

The algebraic expressions for the functions shown in Table 2.9 are:

$D_0 = IS'_1S'_0$

$D_1 = IS'_1 S_0$

$D_2 = IS_1S'_0$

$D_3 = I S_1 S_0$

CHECK YOUR PROGRESS-VI

19. Demultiplexer basically _____ the multiplexing function.
20. In demultiplexer only _____ gate is enabled at a time by the data-select lines.
21. Data on the data-input line passes via the selected gate to the associated data _____ line.

2.10 SUMMING UP

- AND-OR logic produces an output expression in SOP form
- AND-OR-Invert logic produces a complemented SOP form, which is actually a POS form.
- Combinational circuits are designed either using Boolean expression or truth tables.
- The operational symbol for exclusive-OR is $\oplus$. An exclusive-OR expression can be stated in two equivalent ways: $AB'+A'B=A\oplus B$
- To do an analysis of a logic circuit, start with the logic circuit, and develop the Boolean output expression or the truth table or both.
- Implementation of a logic circuit is the process in which you start with the Boolean output expressions or the truth table develop a logic circuit that produces the output function.
- Minimization of Boolean expressions should be tried before implementing a logic circuit.
- NAND and NOR gates are called universal logic gates.
- All NAND or NOR logic diagrams should be drawn using appropriate dual symbols so that bubble outputs are connected to bubble inputs and non-bubble outputs are connected to non-bubble inputs.
- When two negation indicators (bubbles) are connected, they effectively cancel each other.
- The basic logic functions are comparison, arithmetic, code conversion, decoding, encoding, data selection, storage, and counting.
- Addition, subtraction, multiplication, division, encoding, decoding, multiplexing, demultiplexing, etc. are the functionalities of the combinational logic circuits.

Space for learners:

- To perform the addition operation half-adder, full-adders, parallel adders are used.
- Half-adders can be combined to design the full-adders.
- Ripple carry and look-ahead carry are the examples of carries seen in the adder circuits.
- Comparator circuits are used to compare any two binary numbers and MSB are given more precedence while comparing two binary numbers.
- Subtractor circuits also have a similar design like the adder circuits.
- Encoder and decoder are the code converter circuits, they perform reverse operation with each other.
- Multiplexer and demultiplexer are data selector and data distributor, they also perform reverse operation with each other.

Space for learners:

2.11 KEY TERMS

- **SOP:** Sum of Product expressions
- **POS**: Product of Sum expressions
- **Half-adders:** add two binary numbers and produce sum and carry in the output.
- **Full-adders:** add two binary numbers with input carry and produce sum and carry in the output.
- **Half-subtractor**: subtract binary numbers and produce difference and borrow out.
- **Full-subtractor**: subtract two binary numbers with borrow in and produce difference and borrow out.
- **Comparators:** Compare two binary numbers
- **Decoders:** Detect the specific combination of bits in the input.
- **Encoders:** An encoder converts understandable alphabet to numbers into the coded forms.
- **Multiplexers:** Multiplexers are the data selectors. Multiplexers transmit data coming from different sources over a single line.
- **Demultiplexers:** Demultiplexers show the reverse operation of multiplexers. It takes digital data from a single line and distributes them in several lines.

2.12 ANSWERS TO CHECK YOUR PROGRESS

1. Product of Sum
2. Sum of Product
3. A′B+AB′
4. One (one 3-input AND gate)
5. Two
6. Two
7. Four
8. Three
9. False
10. False
11. True
12. True
13. True
14. True
15. True
16. True
17. 2^n
18. $I'_3I_2I_1I_0$
19. Reverses
20. One
21. Output

Space for learners:

2.13 POSSIBLE QUESTIONS

Short answers type questions

1. Determine the output (1or 0) of a 4-variable AND-OR-Invert circuit for each of the following input conditions:
    a. A=1, B=0, C=1, D=0
    b. A=1, B=1, C=0, D=1
    c. A=0, B=1, C=1, D=1
2. Draw the logic diagram for an exclusive-NOR circuit.
3. Determine the output of an exclusive-OR gate for each of the following input conditions:
    a. A=1, B=0
    b. A=1, B=1
    c. A=0, B=1
    d. A=0, B=0
4. Implement the following boolean expressions:
    a. Y = ABC+AB+AC
    b. Y = AB(C+DE)
5. Reduce *Question 4* to minimum SOP form.

6. Use NAND and NOR gates or combination of both to implement the following logic expressions:
    a. Y = A′B+CD+(A+B)′(ACD+(BE)′)
    b. Y = ABC′D′+DE′F+(AF)′
7. Find out the full adders input for the following set of outputs:
    a. Σ =0 ,C_{out}=0
    b. Σ =1 ,C_{out}=0
    c. Σ =1 ,C_{out}=1
8. Implement the expression Y = ((A′+B′+C′)DE)′ using NAND logic.
9. Implement the expression Y = ((A′B′C′)+(D+E))′ using NOR logic.
10. Develop a logic circuit that produces a 1 on its output only when all three inputs are 1s or when all three inputs are 0s.

Space for learners:

Long answers type questions:

1. Define combinational logic circuits. Explain the various components used to design the combinational logic circuits.
2. Design combinational logic circuit for the following expression
    Y = AB(C+DEF)+CE(A+B+F)
3. Develop the truth table for a certain 3-input logic circuit with the output expression
    Y = AB′C+A′BC+A′B′C′+ABC′+ABC
4. Develop truth for the following expressions and draw the circuits:
    a. A′B+ABC′+(AC)′+AB′C
    b. P′+QR′+ SR+PQ′R
5. From the following truth table draw the logic circuit in minimized form

Inputs			Output Y
A	B	C	
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1

1	1	0	0
1	1	1	1

6. Design a 6-bit parallel adder.
7. Design a 4-to-2 line encoder using logic gates.
8. Design a 8-to-1 line multiplexer using logic gates
9. Design a 4-to-16 line decoder using logic gates
10. Design a 1-to-8 line demultiplexer using logic gates
11. Design an adder-subtractor circuit.

2.14 REFERENCES AND SUGGESTED READINGS

1. Mano, M. Morris, *Digital Logic and Computer Design,* Pearson Education.
2. Mano, M. Morris, *Computer System Architecture,* Pearson Education.
3. Jain, R. P., *Modern Digital Electronics,* Mc Graw Hill India.

Space for learners:

UNIT-3: COMPUTER ARITHMETIC

Space for learners:

Unit Structure:

3.1INTRODUCTION

A separate section in central processing unit used to execute arithmetic operations is called arithmetic processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation. In digital computers data is manipulated by using arithmetic instructions. Data is manipulated to produce results necessary to give solution for the computation problems. The addition, subtraction, multiplication and division are the four basic arithmetic operations. We can derive some other operations by using these four operations.

3.2UNIT OBJECTIVES

This unit is an attempt to give an idea of multiplication and division of numbers in digital computer. After going through this unit you will be able to-

- understand the multiplication operation
- understand the division operation
- explain the floating-point arithmetic operation

Space for learners:

3.3MULTIPLICATION OF NUMBERS

Multiplication of two fixed point unsigned binary numbers can be done by a process of successive shift and add operations. But the multiplication of two fixed point signed binary numbers in 2's complement representation requires special consideration.

3.3.1 Multiplication of Unsigned Numbers

Multiplying unsigned numbers in binary is quite easy. We already know that with 4 bit numbers we can represent numbers from 0 to 15.

For Multiplication of binary numbers only we have to remember the number facts: 0*1=0 and 1*1=1 (this is the same as a logical "and").

Multiplication is different than addition. Multiplication of an *n* bit number by an *m* bit number results in an *n*+*m* bit number. Let's discuss with an example where n=m=4 and the result of multiplication is 8 bits:

Example 1:

Decimal	Binary
10 x 6 60	1010(Multiplicand) x 0110(Multiplier) 0000 1010 Partial Product 1010 +0000 0111100 (Product)

In this case of binary multiplication the result is 7 bit, which can be extended to 8 bits by adding a 0 at the left.

Example 2:

Decimal	Binary
7 x 6 42	0111 x0110 0000 0111 0111 +0000 0101010

3.3.2 Multiplication of Signed Numbers

For multiplying binary integers in signed 2's complement representation requires special consideration.

Example 3:

Decimal	Binary
7 x -6 -42	0111 x1010 (2's complement) 0000 0111 0000 +0111 1000110 (The result is incorrect) So, there is an error

Space for learners:

Solution: We must sign extend to the product bit width. The additional values out to the MSB position are called sign extension.

Decimal Number	3 bits	4 bits	8 bits	12 bits
6	110	0110	0000 0110	0000 0000 0110
-6	110	1110	1111 1110	1111 1111 1110
7	111	0111	0000 0111	0000 0000 0111
-7	111	1111	1111 1111	1111 1111 1111

As we know that multiplication of two 4 bit numbers results in8 bits. So for signed multiplication of two 4 bit numbers we must sign extent the numbers to the product bit width i.e, 8 bits.

Example 4:

Decimal	Binary
7 x -6 -42	0111 x 1010 After Sign extension 00000111 x 11111010 00000000 00000111 00000000 00000111 10000111 00000111 00000111 +00000111 |11010110 (2's complement of 42) Stop after 8 bits So the result is correct

Space for learners:

CHECK YOUR PROGRESS-I

1. Do the binary multiplication of (-7) and (-6)
2. Do the binary multiplication of (-7) and (-6)

Space for learners:

3.3.3 Hardware Implementation of Multiplication Operation

The multiplier and multiplicand are stored in two registers Q and M. A third register A is initially set to 0. A 1-bit register C is used to store the carry bit resulting from addition. Control logic reads the bit of the multiplier one at a time. The multiplicand is added to the register A if Q_0 is 1 and then stored the result back to register A with C bit is used to store carry. Then all the bits of CAQ are shifted one position right. No addition is performed if Q_0 is 0. The process is repeated for each bit of the multiplier. The resulting 2n bit product is the contained in QA register

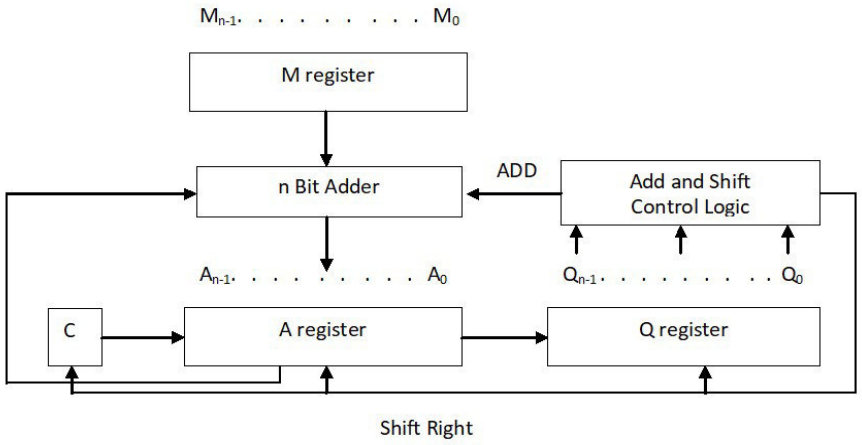
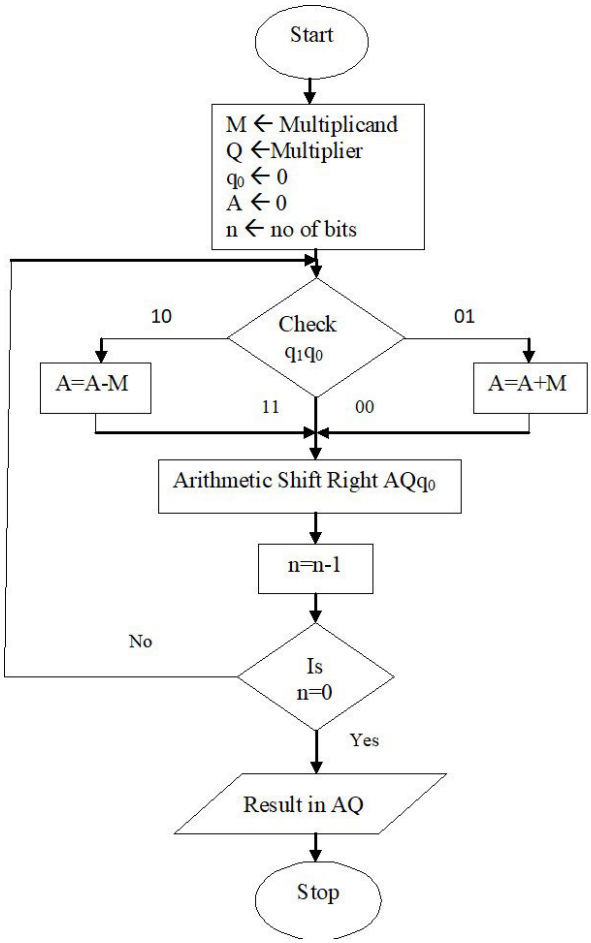


Figure 1: Hardware for Multiplication Operation

3.3.4Booth's Multiplication Algorithm

Booth's algorithm gives a procedure for multiplying binary integers in signed 2's complement representation.

Figure 2: Flow chart of Booth's Algorithm for multiplication of signed numbers



Example 5:

Decimal
- 6 (Multiplicand) x 3 (Multiplier) - 1 8 (Product)

Here, number of bits (n) required for this calculation is 4 bits (1 bit to represent the sign and 3 bits to represent the numbers). Since 6 can be represented using 3 bits and negative sign is represented using 1 bit.

Space for learners:

So, the size of registers M, Q, A(Accumulator) is 4 bits and register q_0 is 1 bit.

$M=(-6)_{10}$

= 2's complement of $(0110)_2$

$= (1010)_2$

$-M = (0110)_2$

$Q = (3)_{10} = (0110)_2 = Q\ (q_4q_3q_2\ q_1)$

Operations:

(i) If q_1q_0 bits are 1 0 then then do A = A – M = A + (-M)
(ii) If $q_1\ q_0$ bits are 0 1 then then do A = A + M
(iii) Otherwise Arithmetic Shift Right of (A Q q_0) is done.

Suppose (A Q q_0) = (0 1 1 0 0 0 1 1 0)

Then ASR will yield the result = (0 0 1 1 0 0 0 1 1). Here sign bit(MSB) is restored and all bits (including the sign bit) is shifted one position right.

TABLE 1: Multiplication of Example 5 using Booth's Algorithm

n	A	Q (q_4 q_3 q_2 q_1)	q_0	Action/Comment
4	0 0 0 0	0 0 1 1	0	Initialization
	0 1 1 0	0 0 1 1	0	A = A – M
3	0 0 1 1	0 0 0 1	1	ASR of (A Q q_0)
2	0 0 0 1	1 0 0 0	1	ASR of (A Q q_0)
	1 0 1 1	1 0 0 0	1	A = A + M
1	1 1 0 1	1 1 0 0	0	ASR of (A Q q_0)
0	1 1 1 0	1 1 1 0	0	ASR of (A Q q_0)

Result is content of AQ i.e, 1 1 1 0 1 1 1 0

Here, LSB (Sign bit) is 1 so the result is –ve.

Space for learners:

Therefore result: - (2's complement of 1 1 1 0 1 1 1 0) = - (0 0 0 0 1

SAQ

1. Do the binary multiplication of -7 and 3 using Booth's Algorithm.

3.4 DIVISION OPERATION

The division operation involves repetitive shifting and addition or subtraction.

First, the bits of the dividend are examined from left to right to search a number greater than or equal to the divisor. Until this event occurs, 0s are placed in the quotient from left to right. When such a number is found and the divisor divides the number, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a partial remainder. In the subsequent cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor. The divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.

CHECK YOUR PROGRESS-II

3. Divide 1001 by 11
4. Divide 111000 by 111

3.5 FLOATING-POINT ARITHMETIC OPERATIONS

Arithmetic operations on floating point numbers are addition, subtraction, multiplication and division.

A floating point number can be represented as m x r^e, where m is called mantissa, r is called radix and e is called exponent part. In computer memory two registers: mantisa and exponent is used to represent a floating point number.

Space for learners:

For example, the decimal number 423.75 can be represented in a register with m=42375 and e=3 and is interpreted to represent the floating point number

. 42375 x 10^3

3.5.1 Addition/Subtraction of two Floating point numbers:

Steps to add/subtract two floating point numbers are as discussed below:

(i) Alignment: Compare the magnitudes of two exponents and align the number with smaller magnitude of exponents

(ii) Addition/Subtraction: Addition or subtraction is done following the addition or subtraction rules.

(i) Normalize the result: If MSB of mantissa part of the product is 1, the product is already normalized. If it is 0 underflow occurs and the mantissa of the product is shifted left and decrement the exponent value. If overflow occurs, mantissa is shifted right and exponent is incremented

Example 6: Add 1.1010 x $2^4$and 1.101x 2^2

Solution:

Step (i),Here 1.101 x 2^2 is aligned to 0.01101 x 2^4.

Step (ii), Add the two numbers 1.1010 x 2^4 and 0.01101 x 2^4 to get 10.00001 x 2^4

Step (iii),Result = 10.00001 x 2^4. So, overflow in the result.

After normalization the result is 0.1000001 x 2^6

SAQ

1. Add 1.1100 x 2^4and1.100 x 2^2.

Space for learners:

3.5.2 Multiplication of two Floating point numbers:

(ii) **Add the exponents:** Exponents of two numbers are added to get the exponent of the product.

(iii) **Multiply the mantissas:** Multiplication of mantissas are done following multiplication rule.

(iv) **Normalize and round the result:** Overflow cannot occur during multiplication. If MSB of mantissa part of the product is 1, the product is already normalized. If it is 0, then the mantissa of the product is shifted left and decrement the exponent value.

Example 7: Multiply 1.000×2^{-2} and 1.010×2^{-1}

Solution:

Step (i), (-2) + (-1)=-3, this is the exponent value of the product.

Step (ii), Multiply the mantissas: 1.000 x (1.010) = 1.010000

Step (iii),Result = 1.010000×2^{-3}

Underflow in the result. So after normalization result is = 0.10000×2^{-2}

STOP TO CONSIDER

In floating point multiplication if either operand is equal to zero, the product is set to zero and operation is terminated. Procedure for arithmetic operations on floating point numbers is different than integers.

3.6 SUMMING UP

- Procedure to do multiplication of signed and unsigned number is different.
- Multiplication of two fixed point unsigned binary numbers can be done by a process of successive shift and add operations.

Space for learners:

- Multiplication of signed numbers can be done using Booth's Algorithm.
- The multiplier and multiplicand are stored in two registers Q and M.
- The division operation involves repetitive shifting and addition or subtraction.
- Arithmetic operations on floating point numbers is done in a different way.

3.7POSSIBLE QUESTIONS

1. Perform binary multiplication of -8 and -3 using sign extension method
2. Perform binary multiplication of 9 and -4 using Booth's Algorithm.
3. Write the steps of Booth's Algorithm.
4. Discuss the hardware implementation of multiplication operation.
5. Write the steps of division operation.
6. How the alignment and normalization is done in addition of two floating point numbers?

3.8REFERENCES AND SUGGESTED READINGS

1. Computer System Architecture, M. Morries Mano

Space for learners:

UNIT4: REGISTER TRANSFER LANGUAGE AND PROCESSOR LOGIC DESIGN

Space for learners:

Unit Structure:

4.1 INTRODUCTION

As you know, all the operations or instructions in a digital computer are carried out by a processor with the help of various other interconnected modules. The elementary operations are also called as micro-operations that are performed on the data stored on the processor registers. This unit contains the fundamentals of micro-operations and the language used to represent various micro-operations which is known as Register Transfer Language

(RTL), the concept of Datapaths and other significant parts of the Central Processing Unit (CPU) such as the Arithmetic and Logic Unit (ALU) where you will be able to understand the functioning of the Arithmetic Circuit and Logic Circuit and lastly in the Control Unit (CU) part, you will be able learn the design of CU and its functionalities and also about alternative designs of the CU - hardwired and micro-programmed control unit.

Space for learners:

4.2 UNIT OBJECTIVES

After completing this unit, you will be able to learn:

- The concepts Micro-operation and representation of Register and uses of Register Transfer Language (RTL)
- Concepts of one-bus, two-bus and three-bus organization
- Concepts related to ALU (basic design of Arithmetic Circuit and Logic Circuit)
- Concept of the Control Unit (Micro-programmed and Hardwired Control Unit)

4.3 REGISTER TRANSFER LANGUAGE

The internal hardware organization of a digital computer exhibits an interconnection of digital modules such as registers, decoders, arithmetic logic and control logic etc. The complete digital system is interconnected with data and control paths commonly known as bus. The elementary operations performed by the CPU on the data stored in one or more registers are termed as micro-operations. clear, count, load and shift are some examples of such micro-operations.

Various categories of micro-operations:

The most commonly used micro-operations in a digital computer are listed below-

- **Register transfer micro-operations:** The micro-operations that are used to transfer binary information among various registers.
- **Arithmetic micro-operations:** The micro-operations that are used to perform various arithmetic operations (add, subtract,

increment, decrement) on arithmetic data stored in the registers.

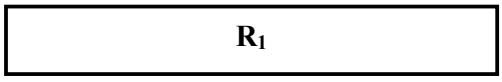
- **Logic micro-operations:** The micro-operations that are used to perform logical operations (AND, OR, NOT) on the data stored in the registers.
- **Shift micro-operations:** The micro-operations that are used to perform either left or right shift operations (logical, arithmetic, circular) on the data stored in the registers.

The **Register Transfer Language (RTL)** is the representation system used to describe the sequence of micro-operations in a symbolic form. The term register transfer refers to the transfer of binary information among the registers via a common path or bus.

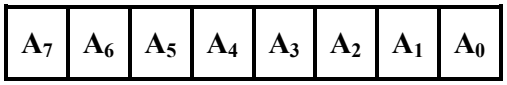
Space for learners:

4.3.1 Representation of Registers

In a digital computer system the registers are represented using upper case letters (and followed by a numeral sometimes). PC- Program Counter, IR- Instruction Register etc. are examples of special purpose registers and R_1, R_2, R_3… etc. are examples of general purpose registers. A register is represented by a rectangular box containing the name inside and the bit numbers can be marked at the top of the box starting from left to right as shown in figure 4.1 (a) & (b). Each bit of the data stored in the register can be represented as shown in the figure where each individual bit is assigned a letter along with a numeral subscript that indicates the position of the bit. Considering a 16-bit register with bits numbered from 0 to7 are termed as low byte (L) while the bits from 8 to 15 are termed as high byte (H) of the register as shown in figure 4.1 (c) & (d).



(a)



(b)

A_{15} A_0

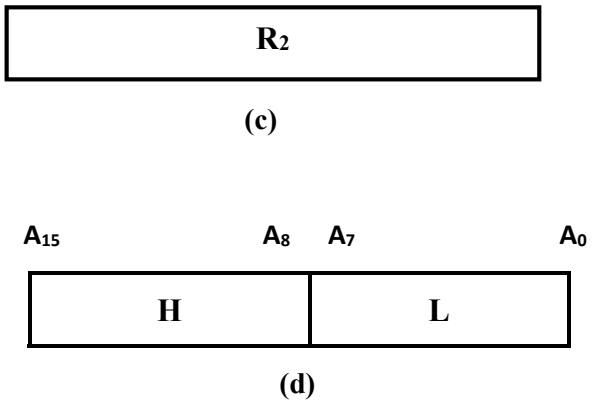


Fig4.1: Block diagram of Register

4.3.2 Register Transfer Representation

The transfer of contents from one register to another register can be shown with the help of replacement operator (←). For example, transfer of data from register R_1 to register R_2 can be symbolically expressed using the following statement in RTL.

$R_2 \leftarrow R_1$

When this statement gets executed contents of R_2 will be replaced by contents of R_1, but the contents of R_1 remains unchanged. The execution of this statement can be controlled by putting some control condition also. That means when the control condition satisfies then only the transfer takes place, otherwise not. This is shown in the following expression:

`If (P = 1)  then (`$R_2 \leftarrow R_1$`)`

Here, `P=1` is the control statement or control function which is a Boolean variable.

The statement can also be written as

`P:` $R_2 \leftarrow R_1$

The colon (:) separates the control condition from the rest.

The hardware implementation of the statement `P:` $R_2 \leftarrow R_1$ is shown below:

Space for learners:

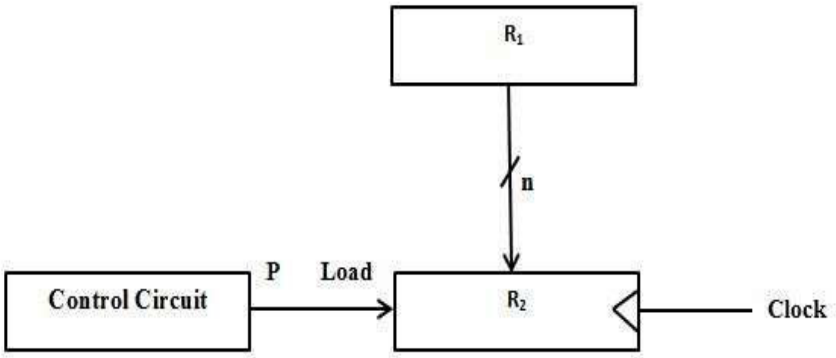


Fig4.2: Hardware implementation of $P: R_2 \leftarrow R_1$

4.3.3 RTL Representation of Memory Transfers

Data flow from memory to external environment is known as *memory read* operation while data from external environment is stored in memory is referred to as *memory write* operation. In RTL, the memory word is symbolized by the letter *M* followed by square brackets [ ] having the address of the memory word. For example, transferring a data word *M* from memory whose address is stored in Address Register (AR) to Data Register (DR) can be symbolized as:

`Read: DR ←M[AR]`

Similarly, the write operation can be symbolized as:

`Write: M[AR] ←` R_1

which means transfer of data from register R_1 to the memory word *M* whose address is stored in the Address Register (AR).

CHECK YOUR PROGRESS-I

1. The operations executed on data stored on registers are called _____.
   a) macro operations b) micro operations
   c) Byte Operations d) Bit Operations
2. Which of the following register transfer statements is correct?
   a) P, $R_1 \leftarrow R_2$ b) P : $R_1 \leftarrow R_2$: $R_3 \leftarrow R_4$
   c) P : $R_1 \leftarrow R_2$, $R_3 \leftarrow R_4$ d) None of the these
3. What does the following transfer statement indicate?
   $R_2 \leftarrow M[R_1]$
   a) Read a memory word at the address stored in R_1
   b) Read a memory word at the address stored in R_2
   c) Write a memory word at the address stored in R_1
   d) Write a memory word at the address stored in R_2

Space for learners:

4. *State TRUE or FALSE:*
   a) Considering a 16-bit register the bits numbered from 0 to7 are termed as low byte (L).
   b) Shift micro-operations are used to perform various logical (AND, OR, NOT) operations.
   c) A register transfer can't occur unless the specified control condition becomes true.

Space for learners:

4.4 DATAPATH

The Central Processing Unit of a digital computer can be divided into a data section and a control section. The data section, also called as data path, contains the registers and the Arithmetic Logic Unit (ALU) and the Buses. There are three types of buses- Address bus, Data bus and Control bus. The data path is accomplished for performing certain operations on data items stored in various memory units. The control section is basically comprised of the control unit, which issues various control signals to the data path. Internal data (which may be data, instructions or addresses) transfers are carried out via local buses. Externally, data transfer from registers to memory and Input-Output devices, often carried out by a system bus. The local bus organization to perform internal data transfer among registers and the ALU may be of different organizations like one-bus, two-bus, or three-bus organization.

4.4.1 One-Bus Datapath

In this organization, the CPU registers and the ALU use a single bus to transfer data. This bus organization is least expensive and simplest in design, but it restricts the amount of data transfer that can be done in the same clock cycle, which results in decrease of overall performance of the system. Figure 4.3 shows a one-bus data path organization comprising of a set of general-purpose registers, a memory address register (MAR), a memory data register (MDR), an instruction register (IR), a program counter (PC), and an ALU, all are interconnected via a single data path.

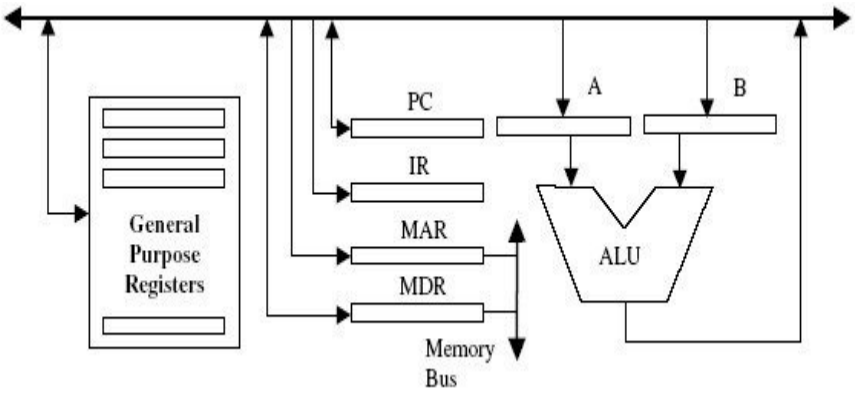
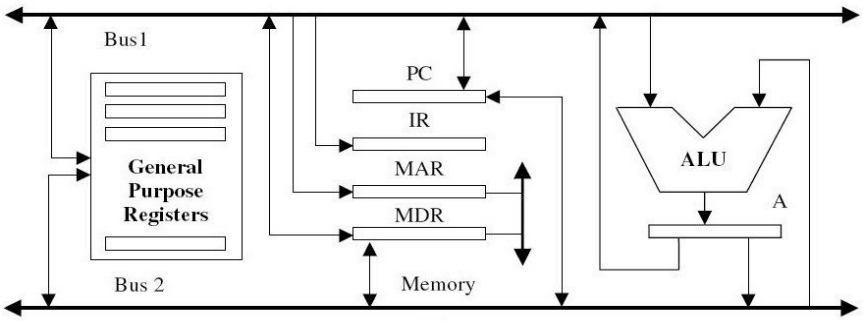


Fig4.3: One-bus data path

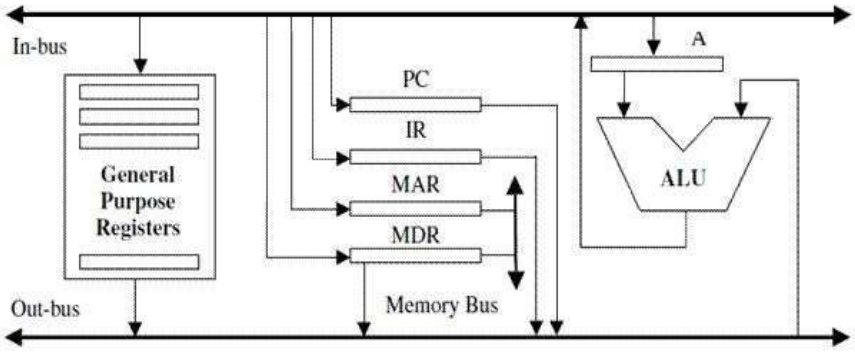
Space for learners:

4.4.2 Two-Bus Datapath

In two-bus organization, two buses are used which results a faster performance than the one-bus organization. In this case, the general-purpose registers are connected to both the buses. Data can be transferred from two different registers to the input point of the ALU at the same time. Therefore, an operation having two operands can fetch both operands in the same clock cycle. There may be a need of an additional buffer to hold the output of the ALU when the two buses remain busy in carrying the two operands. Figure (4.4-a) shows a two-bus organization. There may be another implementation of two bus organization where one of the buses is dedicatedly used for moving data into registers (in-bus), while the other bus is dedicatedly used for transferring data out of the registers (out-bus). For this purpose, the buffer register may be used additionally, as one of the ALU inputs, to hold one of the operands. The ALU output can be connected directly to the in-bus, which transfers the result to one of the registers. A two-bus organization with in-bus and out-bus is shown in Figure (4.4-b).



(a)



(b)

Fig4.4 :(a) Two bus data path(b) Two bus data path with in-bus and out-bus

4.4.3 Three-Bus Datapath

In case of three-bus organization, two buses may be used as source buses whereas the third bus is used as destination. The source buses are used to transfer data out from registers (out-bus), and the destination bus may be used to transfer data into a register (in-bus). Each of the two out-buses is connected to an ALU input point and the output of the ALU is connected directly to the in-bus. As we have more buses in this organization, more data can be transferred within a single clock cycle. However, increasing the number of buses also increases the complexity as well as cost of the hardware. Figure (4.5) shows the organization of a three-bus data path.

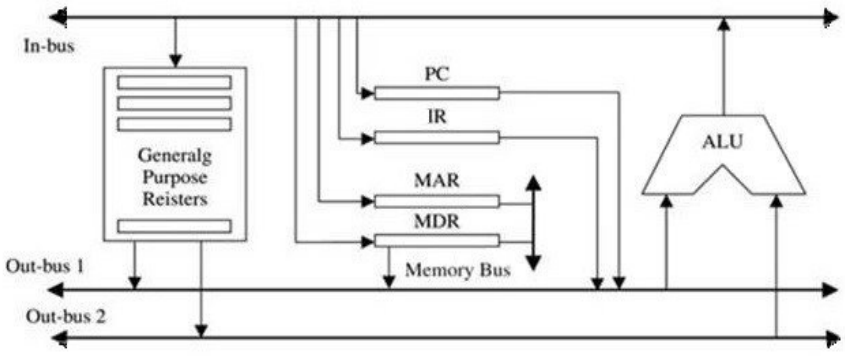


Fig 4.5 : Three-bus data path

Space for learners:

CHECK YOUR PROGRESS-II

5. The data section of the CPU is also known as _____.
6. In-bus, out-bus organization is related to _____.
   a) one-bus data path b) two-bus data path
   c) three-bus data path d) None of these.
7. In _____ all the General Purpose Registers (GPR), Special Purpose Registers (SPR) and the ALU are connected via a single data path.
   a) one-bus data path b) two-bus data path
   c) three-bus data path d) None of these.
8. Two-bus data path is more efficient than Three-bus data path. (*State TRUE or FALSE)*

Space for learners:

4.5 ALU DESIGN

The arithmetic and logic unit (ALU) is a combinational circuit in a digital computer which performs the following operations-

- *Arithmetic operations* such as add, subtract, increment and decrement etc.
- *Logic operations* such as AND, OR, XOR and compliment etc.
- *Bit Shifting operations* such as logical left and right shift used for multiplication purpose.

Therefore, we can say ALU is the combination of arithmetic unit, logic unit and shift unit all the three circuits together. It is usually a part of the central processing unit (CPU). Many CPUs have separate units for arithmetic operations (Arithmetic Unit-AU) and for logic operations (Logic Unit-LU).

4.5.1 Arithmetic Circuit

The 4-bit arithmetic circuit which is shown in Figure 4.6 is able to perform different basic arithmetic operations such as add, subtract, increment and decrement. It employs parallel full adders (FA) to perform these operations depending on the inputs. The select inputs S_0 and S_1 are used to provide different inputs to the multiplexers (MUX) present in the circuit in order to obtain different arithmetic operations as outputs.

The output of the arithmetic circuit is calculated from the following arithmetic expression-

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

Where A is the 4-bit number (A_0, A_1, A_2, A_3) to the x input (X_0, X_1, X_2, X_3) of the full adders, y is the 4-bit number (the outputs from the multiplexers) to the y inputs (Y_0, Y_1, Y_2, Y_3) to the full adders and C_{in} is the input carry which is either 0 or 1.

Depending on the values of S_0, $S_1$and C_{in}, the arithmetic circuit performs eight different microoperations as listed in the function table shown in Table 4.1.

Let's consider a few cases for better understanding the functioning of the arithmetic circuit.

CASE I: $S_1 = 0$ and $S_0 = 0$

In this situation, the input pins of multiplexers I_0(i.e. the bits of B) are chosen as the output and as a result B directly goes to they inputs of the full adders (FA). i.e. $y = B$. Now, if $C_{in} = 0$ then the output becomes $D = A + B$ and if $C_{in} = 1$ then $D = A + B + 1$. This is how add microoperation is performed.

CASE II: $S_1 = 0$ and $S_0 = 1$

In this situation, the input pins of multiplexers I_1 (i.e. the complimented bits of B) are chosen as the output and as a result$\overline{B}$ goes to the *y* inputs of the full adders (FA). i.e. $y = \overline{B}$. Now, if $C_{in} = 0$ then the output becomes $D = A+\overline{B}$which is equivalent to $D = A-B-1$ and if $C_{in} = 1$ then $D = A+\overline{B}+1$ which is equivalent to $D = A-B$. This is how subtract microoperation is performed.

CASE III: $S_1 = 1$ and $S_0 = 0$

In this situation, the input pins of multiplexers I_2 (connected to logic 0) are chosen as the output and as a result0 goes to the *y* inputs of the full adders (FA). i.e. $y = 0$. Now, if $C_{in} = 0$ then the output becomes $D = A+0$ i.e. $D = A$ which means transfer operation is done from A to Dand if $C_{in} = 1$ then $D = A+1$ which means increment operation is performed.

Space for learners:

CASE IV: $S_1 = 1$ and $S_0 = 1$

In this situation, the input pins of multiplexers I_3 (connected to logic 1) are chosen as the output and as a result `1` goes to the all *y* inputs of the full adders (FA) and we know that if all bits of a number are 1 then it's equivalent to `-1`. So, y =`-1` here. Now, if $C_{in} = 0$ then the output becomes `D = A-1` which means decrement operation is done and if $C_{in} = 1$ then `D = A-1+1` i.e. `D = A` which means which means transfer operation is done from `A` to `D`.

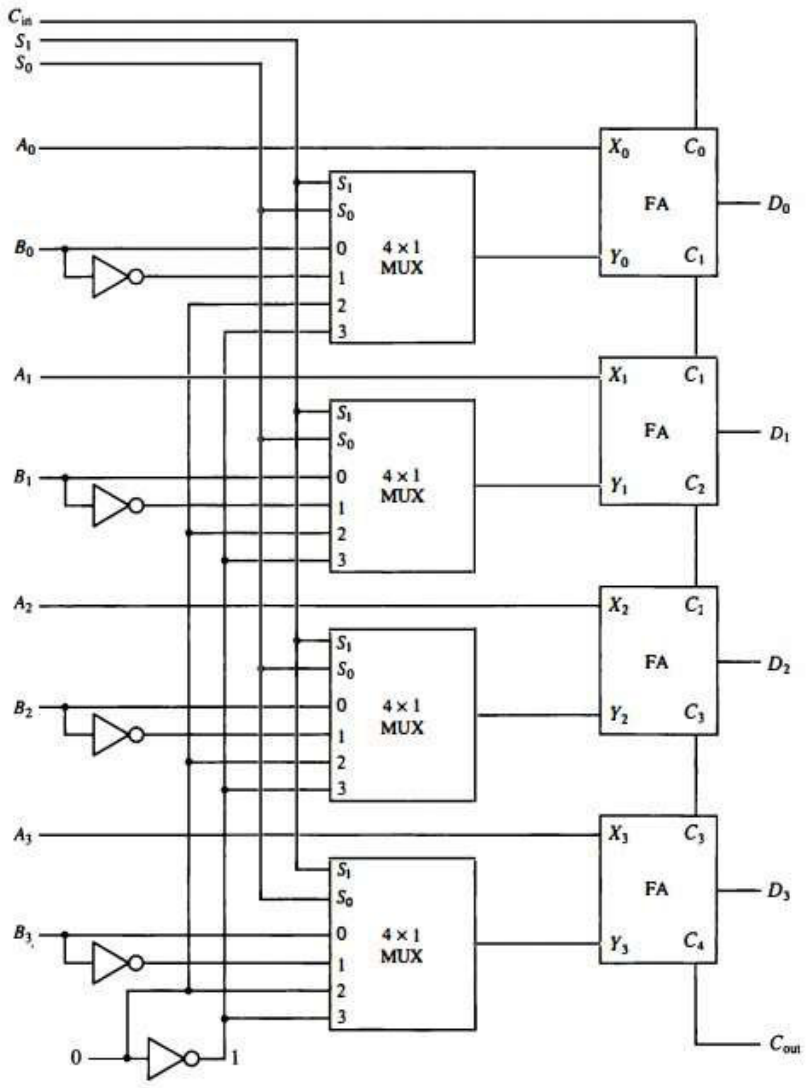


Fig 4.6 : A 4-bit Arithmetic Circuit

Space for learners:

Table 4.1 Function Table of Arithmetic Circuit

Inputs				Outputs	Microoperations
S_1	S_0	C_{in}	Y	$D = A + y + C_{in}$	
0	0	0	B	A + B	Add
0	0	1	B	A + B + 1	Add with Carry
0	1	0	$\overline{B}$	A + $\overline{B}$orA − B − 1	Subtract with Borrow
0	1	1	$\overline{B}$	A + $\overline{B}$ + 1 or A − B	Subtract
1	0	0	0	A	Transfer A
1	0	1	0	A + 1	Increment A
1	1	0	1	A − 1	Decrement A
1	1	1	1	A	Transfer A

4.5.2 Various Arithmetic Microoperations

Add, subtract, increment and decrement are the basic set of arithmetic microoperations which are described below-

- **Add**: To add the contents of two or more registers and store the resultant sum in either one of the registers or in a third register, this microoperation is used. For example, to add the contents of two registers R_1 and R_2 and store the result in a third register R_3, the microoperation can be symbolized as:

  $$R_3 = R_1 + R_2$$

- **Subtract:** When the contents of one register needs to be subtracted from another register and store the result in either one of the registers or in a third register, then this microoperation is used. The subtraction operation is implemented through complement and addition operation. For example, to subtract the contents of register R_2 from register $R_1$and store the result in a third register R_3, the microoperation can be symbolized as:

  $$R_3 = R_1 + \overline{R_2} + 1 \text{ [Equivalent to } R_3=R_1-R_2\text{]}$$

  Here, first we take the complement of R_2, add 1 to it and then the content of $R_1$is added to it. In other words, the 2's complement of R_2 is added with R_1 in order to carry out R_1-R_2 operation.

Space for learners:

- **Increment:** This type of microoperation is used to increase the contents of a register by 1. For example, to increase the contents of register R_1 by one the symbolic microoperation will be:

$$R_1 = R_1 + 1$$

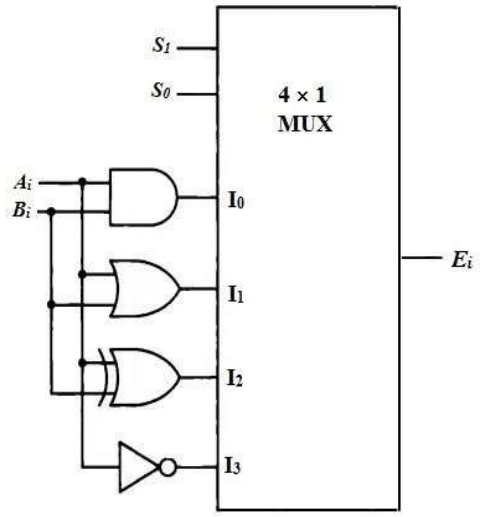
- **Decrement:** This type of microoperation is used to decrease the contents of a register by 1. For example, to decrease the contents of register R_1 by one the symbolic microoperation will be:

$$R_1 = R_1 - 1$$

Space for learners:

4.5.3 Logic Circuit

The basic logic circuit of the ALU performs various logic microoperations such as AND, OR, XOR and Compliment at bit-level.



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*

Fig 4.7: Single stage of Logic Circuit with Function Table

Fig 4.7 shows the hardware implementation for four common logic microoperations. The circuit is consisting of a 4×1 multiplexer with four inputs (I_0, I_1, I_2 and I_3) and two select pins (S_0 and S_1) to perform one of the four logic microoperations and direct it as the output E_i as shown in the function table.

Space for learners:

4.5.4 Some Applications of Logic Micro operations

The basic logic operations (AND, OR, NOT, XOR) can be applied to achieve various operations like set, clear, masking and inserting new bits etc. Let's discuss such common applications here-

- **Selective-set:** To set selected bits in register R_1 to 1 where there are corresponding 1's in register R_2. The 0'sare not considered. For example, before operation if R_1=0011 and R_2=0101then after selective-set operation the contents of R_1will be 0111. This operation is achieved by the OR logic micro-operation, for above example this will be symbolized as:$R_1 \leftarrow R_1 \vee R_2$

- **Selective-clear:** This operation clears those bits in register $R_1$to 0 where there are corresponding 1's in registerR_2. For example, before operation if R_1=0011 and R_2=0101then after selective-clear operation the contents of R_1will be 0010. This operation is achieved by the AND logic micro-operation with $R_1$and complement of R_2, for above example this will be symbolized as: $R_1 \leftarrow \overline{R_1 \wedge R_2}$

- **Selective-complement:** This operation complements those bits in register $R_1$where there are corresponding 1's in register R_2. For example, before operation if R_1=0011 and R_2=0101then after selective-complement operation the contents of R_1will be 0110. This operation is achieved by the XOR logic micro-operation with R_1andR_2, for above example this will be symbolized as: $R_1 \leftarrow R_1 \oplus R_2$

- **Mask:** This operation clears those bits to 0 in R_1 where there are corresponding 0's in R_2. For example, before operation if R_1=0011 and R_2=0101then after mask operation the contents of R_1will be 0001. This operation is achieved by the AND logic micro-operation with R_1andR_2, for above example this will be symbolized as: $R_1 \leftarrow R_1 \wedge R_2$

- **Clear:** To compare the contents of two registers and results in all 0's if the contents of both the registers are same. For example, before operation if R_1=0011 and R_2=0011then after clear operation the contents of R_1will be 0000. This

operation is achieved by the XOR logic micro operation with R_1andR_2, for above example this will be symbolized as: $R_1 \leftarrow R_1 \oplus R_2$.Thus, XOR operation can be implemented to determine whether two binary numbers are equal.

- **Insert:** This operation is used to insert new group of bits in a register. To perform this operation, first the unwanted bits of the register are masked and then OR operation is performed with the desired value. For example, if R_1=00111100 and we want to insert 0110 in the rightmost four bits. For this, first we mask the rightmost four bits which is done by ANDing the contents of R_1 with the value 11110000. After this mask operation we get R_1=00110000. Now the contents of R1 are ORed with the desired value (00000110) and after this operation we get R1 = 00110110. Thus, the new bits (0110) are inserted at the rightmost four bits.

4.5.5 Shift Microoperations

The shift microoperations move the contents (bits) of a register either to the left or to the right. There are three types of shift microoperations: *arithmetic, logical* and *circular* shifts. Let's discuss them one by one here.

- **Arithmetic Shift**: It shifts a signed binary number either to left or right without changing the sign of the number. To understand the arithmetic shift operation, let's consider a n-bit signed binary number $b_{n-1}, b_{n-2}, \ldots\ldots, b_1, b_0$ where b_{n-1} denotes the sign bit and b_{n-2} denotes most significant bit (MSB) and b_0 denotes the least significant bit (LSB).

  The arithmetic shift operations can be symbolized as:

Space for learners:

$R_1 \leftarrow$ ashr R_1[1-bit arithmetic shift right R_1]

$R_1 \leftarrow$ ashl R_1[1-bit arithmetic shift left R_1]

In arithmetic shift right operation, as the sign bit must be kept unchanged; all the bits including the sign bit are shifted to the right. So, the rightmost bit is lost. The b_{n-1} remains the same, while $b_{n-2}$is replaced by b_{n-1}, b_{n-3} is replaced by b_{n-2}, and so on and at last b_0 is lost.

In arithmetic shift left operation, all the bits are shifted to the left and a 0 is inserted in the previous b_0 bit position. The original value of b_{n-1} is lost as it is replaced by b_{n-2};$b_{n-2}$is replaced by b_{n-3} and so on. After this operation if the value of b_{n-1} changes, then a sign reversal occurs which happens because of *overflow* which occurs if $b_{n-1} \neq b_{n-2}$ before the shift operation. The overflow condition can be checked by XORing the bit b_{n-1} with bit b_{n-2}. If the XOR operation results in 1 then there is overflow; otherwise not.

- **Logical Shift:** This micro-operation moves the contents (bits) of a register either to the left or to the right. After left or right shift the empty/lost(the leftmost or the rightmost) bit is replaced by a 0. Symbolically they are represented as:

  $R_1 \leftarrow$ shr R_1[1-bit shift right R_1]

  $R_1 \leftarrow$ shl R_1[1-bit shift left R_1]

- **Circular Shift:** This micro-operation is almost same as the logical shift, except there is no bit lost occurs here as the leftmost or rightmost bit which is shifted out at one end is circulated back to the other end. Symbolically they are represented as:

  $R_1 \leftarrow$ cir R_1[1-bit circular shift right R_1]

  $R_1 \leftarrow$ cil R_1[1-bit circular shift left R_1]

Space for learners:

CHECK YOUR PROGRESS-III

9.The full form of ALU is _____

10.The ALU gives the output of the operations and the output is stored in the _____.

a) Memory Devices
b) Registers
c) Flags
d) Output Unit

11. The content of an 8-bit register is initially is 10011100. The content of the register after an arithmetic right shift operation will be _____.

a) 11001110 b) 11001111

c) 11001101 d) 01001110

12. In arithmetic left shift, overflow occurs when _____.

a) $b_{n-1}=b_{n-2}$ b) $b_0=b_{n-1}$

c) $b_{n-1}\neq b_{n-2}$ d) $b_0 \neq b_n$

13. A digital computer performs which one of the following microoperations to subtract R2 from R1?

a) $R_3 = R_1+R_2+1$ b) $R_3 = R_1-R_2$

c) $R_3 = R_1+\overline{R2}+1$ d) $R_2= R_1-1$

14. *State TRUE or FALSE:*

a) The ALU performs logic operations only.

b) XOR operation can be used to check whether the contents of two registers are same.

c) In Circular Shift the bit in either end is circulated back to the other end.

Space for learners:

4.6 CONTROL UNIT

The main unit of the CPU is the control unit (CU) which generates control signals to the data path to direct the entire system operations. Data inside the CPU, memory unit and I/O devices are controlled by

these control signals issued by the control unit. The CU generally uses the control bus to carry the control signals to control the data flow between the CPU and other external units. The two basic tasks performed by the CU are:

- **Sequencing:** The CU is responsible for generating a proper sequence for the microoperations depending on the program currently being executed by the CPU.
- **Execution:** The CU causes the execution of micro-operations by generating control signals for opening and closing of gates to let the data pass through, while performing ALU operations.

Space for learners:

4.6.1 General Model of the CU

The Control Unit (CU) has inputs that empower it to identify the state of the system and outputs that empower it to control the function of the whole system. In addition to input and outputs it must include the logic required to perform the sequencing and execution which are the main functions of the CU. Fig 4.8 illustrates a general model of a control unit consisting of four major inputs: clock, Instruction Register (IR), flags, and the control signals from the control bus. The outputs from the CU are: control signals within CPU and control signals to control bus.

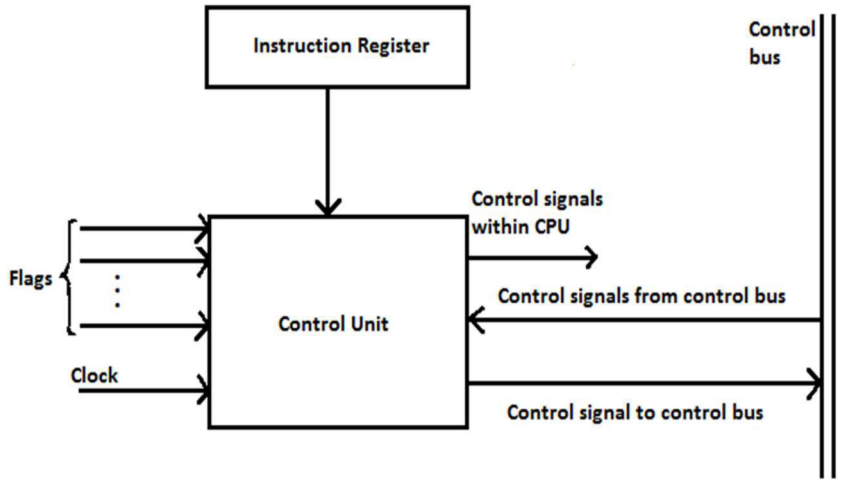


Fig 4.8: General Model of the CU

The inputs of the control unit are described below:

- **Clock:** The control unit uses a clock to keep track the time and sequence of microoperation execution. For each clock

pulse the control unit executes one microoperation or a set of concurrent microoperations which can be referred to as clock cycle time or processor cycle time.

- **Instruction Register (IR):** Microoperations that are fetched from the memory are stored in the IR. The opcode part of the instruction used for decoding the type of instruction to be executed.
- **Flags:** These are certain memory units capable of holding just 1 bit of information that are used to indicate the CU the current state of the processor and the results of recent ALU operations.
- **Control signals form control bus:** Interrupt signals, acknowledgement signals are such signals that are received by the CU from the control bus.

The outputs from the CU are described below:

- **Control signals within CPU:** Two types of control signals are generated by the CU within the CPU; one of these causes the register transfers and the other one is used to activate specific ALU functions.
- **Control signals to control bus:** Two types of control signals are generated by the CU to the control bus; one goes to the I/O modules and another goes to the memory.

The control unit of a digital computer can be implemented in two alternate ways: *hardwired* and *microprogrammed* implementation. In hardwired implementation the control unit is comprised of logic gates, decoders, flip-flops and other control signal generating digital circuits etc. In microprogrammed implementation, the control unit is comprised of a *control memory* where the control information is stored, which is programmed in such a way to initiate proper sequence of microoperations as required. Let's discuss both the implementations one by one.

4.6.2 Hardwired Control Unit

If the control signals are generated by the hardware using conventional logic design then control unit is said to be hardwired controlled. Fig 4.9 depicts the block diagram of a hardwired control unit consisting of a sequence counter (SC) and a number of logic

Space for learners:

circuits which may include decoders, flip-flops and other control logic gates.

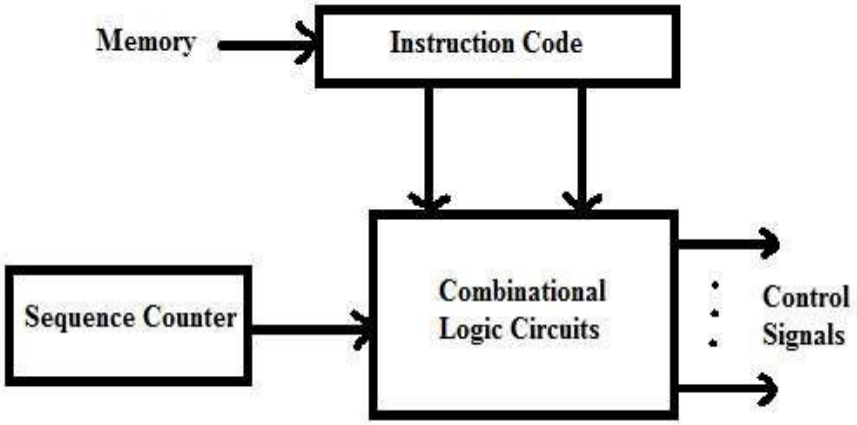


Fig 4.9 : Hardwired Control Unit

The hardwired organization is very complicated if we have a large control unit. In this organization, if the design has to be modified or changed then it requires changes in the wiring among various components. Thus the modification of all the combinational circuit may be very difficult.

Advantages:

- It works fast because of the use of combinational circuits to generate control signals.
- It can be optimized to operate in fast mode.
- It is faster than microprogrammed control unit.

Disadvantages:

- Hardwired control unit is expensive.
- If the design has to be modified or changed then it demands changes in the wiring among various components.
- The design becomes complex if the number of control points in the CPU is large.

4.6.3 Microprogrammed Control Unit

A microprogrammed control unit's design is based on the concepts of microprogramming. Unlike the hardwired CU where the control signals are generated via combinational circuits, here control signals are generated using a sequence of microinstructions that specify the internal control signals for executing the microoperations. *Fig 4.10* depicts the organization of a micro programmed control unit.

Space for learners:

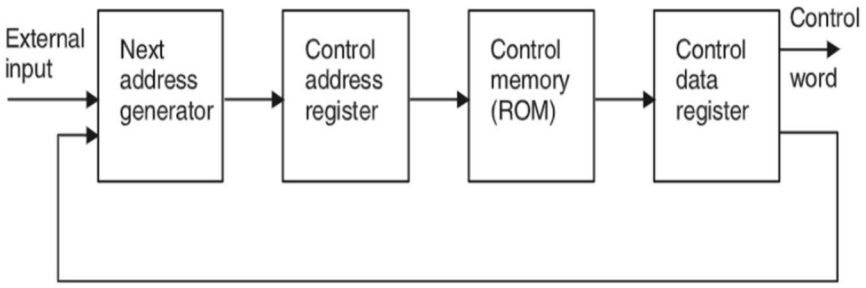


Fig 4.10 :Microprogrammed Control Unit

The microprogrammed control unit comprises of four components, which are described below:

- **Control Address Register (CAR):**The CAR specifies the address of the microinstruction that is read from the control memory. It can also be termed as the microprogram counter ($\boldsymbol{\mu PC}$).
- **Control Data Register:** Depending upon the address specified in the CAR the control data register holds the microinstruction fetched from the control memory. It can also be termed as microinstruction register ($\boldsymbol{\mu IR}$)
- **Control Memory:** In addition to the main memory, a micro programmed CU has a separate memory called the control memory to hold the microinstructions and fixed microprograms that cannot be changed by a general user. The control memory can be read only memory (ROM) as changes in the microprograms are not required once the CU is under operation.
- **Next-address generator:** After having executed all the microoperations generated by a microinstruction, it is required to find the address of the next microinstruction. The next-address generator is responsible for computing the address of the next microinstruction to be executed. This is why it can be termed as microprogram sequencer.

Advantages:

- The design of a microprogrammed control unit is less complex.
- It is cheaper as number of hardware units is lesser and less error prone to implement.
- It can efficiently compute complex functions such as floating-point arithmetic etc.

Space for learners:

- It offers more flexibility to modification or change; as the modification can be brought just by changing the micro-program residing in the control memory to specify a different control sequence.

Disadvantages:

- It is slower than the hardwired control unit that means it requires more time to execute an instruction.
- In case of limited hardware resources it costs more than the hardwired control unit.
- For smaller CPU, the design duration of microprogrammed control unit is more than the hardwired control unit.

Space for learners:

CHECK YOUR PROGRESS-IV

15. Control Memory is associated with _____.

a) Hardwired CU b) Microprogrammed CU

c) Both a) &b) d) None of these

16.Which one is **not** a function of a Control Unit?

a) Control Signal b) Execution
c) Sequencing d) Programming

17.Which one of the followings is also known as Microprogram Counter?

a) Address Register b) Program Counter (PC)

c) Control Address Register d) Data Register (DR)

18. *State TRUE or FALSE:*

a) Control Data Register is also known as Microinstruction Register (μIR).

b)Microprogrammed CU is faster than Hardwired CU.

c) Control Signals are carried by Control Bus.

4.7SUMMING UP

- In a digital computer, the elementary operations are also termed as micro-operations which are performed on the data stored on the processor registers.

- The language used to specify the sequence of microoperations is known as Register Transfer Language.
- Various types of microoperations are register transfer microoperations, arithmetic microoperations, logical microoperations and shift microoperations.
- The data section of the CPU is data path. There are three types of Buses: address bus, data bus, control bus. One-bus, two-bus and three-bus are the various types of data path organization.
- The arithmetic logic unit (ALU) performs arithmetic, logical and bit-shifting operations using various circuits. Arithmetic circuit for various arithmetic operations and logic circuit for logical operations. The shifting operation can be done with the help of Arithmetic Logic Shift Unit.
- Another major part of the CPU which is the control unit (CU) responsible for generating control and timing signals to maintain the proper sequence of microoperation executions.
- The CU can be differentiated based on its design approach as hardwired CU and micro programmed CU.

Space for learners:

4.8 ANSWERS TO CHECK YOUR PROGRESS

1. (a), 2 (c), 3 (a), 4.a True, 4.b False, 4.c True, 5. Datapath, 6. (b), 7. (a), 8. False, 9.Arithmetic Logic Unit, 10. (b), 11. (a), 12. (c), 13. (c), 14.a False, 14.b True, 14.c True, 15. (b), 16. (d), 17. (c), 18.a True, 18.b False, 18.c True.

4.9 POSSIBLE QUESTIONS

Short answer type questions:

- What do you mean by RTL? Explain.
- What is control function of an RTL?
- How memory transfers are represented by RTL?
- What is data path? What are its types?
- What are the operations performed by the ALU?

- How subtraction operation is performed by the arithmetic circuit?
- How overflow occurs in arithmetic shift left operation?
- What operations can be performed by the logic circuit of the ALU?
- What are the functions performed by the CU?
- What are various types of CU available?
- What are the components of a hardwired CU?
- What are the components of a microprogrammed CU?

Long answer type questions:

- Explain the arithmetic circuit with its function table.
- Explain the logic circuit with its function table.
- Explain the general model of the Control Unit? What are the various types of CU available?
- What is hardwired CU? Discuss its advantages and disadvantages.
- What is microprogrammed CU? Discuss its advantages and disadvantages.
- List out various differences between Hardwired and microprogrammed Control Unit.

Space for learners:

4.10 REFERENCES AND SUGGESTED READINGS

- M. Morris Mano, *Computer System Architecture,* Pearson Education, Latest edition.
- Express Learning Series- *Computer Organization and Architecture,* ITL Education Solutions Limited.