Digital Logic Families:

The basic logic gates discussed above were designed using discrete components like diodes, transistors and resistances etc. In the recent past, it has been possible to fabricate many hundreds of thousands of active and passive components on a small silicon chip. Such fabricated devices are known as integrated circuits (ICs).

The Integrated circuits are broadly classified in two categories namely Linear or analog ICs and digital ICs. The analog ICs mainly contain amplifiers, operational amplifiers, audio and power amplifiers etc. However, the digital ICs contain logic gates etc. The variety of logic gates are fabricated in digital ICs using various technologies.

The digital ICs may further be classified into following categories depending upon their level of integration:

(i) **Small Scale Integrated Circuits (SSI):** Twelve gates per IC are fabricated in SSI and total number of components per chip is less than 100.

(ii) **Medium Scale Integrated Circuits (MSI):** These ICs contain 12 to 100 gates per IC and total number of components per IC is 100 to 1000.

(iii) **Large Scale Integrated Circuits (LSI):** The large scale integrated circuits contain 100 to 1000 gates per IC and number of components is 1000 to 10000 per IC.

(iv) **Very Large Scale Integrated Circuits (LSI):** These ICs contain more than 1000 and less than 10000 gates per IC and total number of components per chip is 10000 to 100000.

(v) **Ultra Large Scale Integrated Circuits (LSI):** More than 10000 gates per IC are fabricated and total components are more than 100000 per chip

The logic families are classified into two categories depending upon the technologies used for fabrication.

1. Bipolar Logic Families
2. Uni-polar Logic Families

The bipolar logic families are mainly of two types.

a. **Saturated Logic Circuits:** In which the transistors are driven into saturation.

b. **Non-Saturated Logic:** In non-saturated transistor logic circuits, the transistors are avoided to go into saturation.

The Saturated logic circuits may further be classified into the following categories:

1. Resistor – Transistor Logic (RTL)
2. Direct Coupled Transistor Logic (DCTL)
3. Integrated Injection Logic (IIL or I^2L)
4. Diode – Transistor Logic (DTL)
5. High Threshold Logic (HTL)
6. Transistor – Transistor Logic (TTL)

The non-saturated logic families are:

1. Schottky Transistor – Transistor Logic (STTL)
2. Emitter Coupled Logic (ECL)

The Uni-polar logic families contains MOS FETs, these are:

1. NMOS or PMOS Logic
2. CMOS (Complementary MOS) logic

Before discussing the details of logic families mentioned above, it is necessary to explain the following characteristics related to them. These parameters will help in comparing the performances of the logic families.

(i) **Fan – in:** The maximum number of inputs that can be applied to a logic gate is known as Fan – in. Thus a three input AND has fan – in as three.

(ii) **Fan – out:** The fan –out of logic gate is the number of gates that can be driven by it. Thus, if a fan-out of a typical gate is 10, then it implies that this gate can drive 10 such gates.

(iii) **Propagation Delay Time:** The propagation delay time of a gate is defined as the time interval between the application of the inputs to a gate and appearance of the signal at the output of the gate.

In other words it is defined as the time interval between a change in input state and the resulting change in output state of the gate. This delay is a very small quantity; it is of the order of few nano second say 20 nsec ($20x10^{-9}$ sec) or 50 nsec ($50x10^{-9}$ sec). The propagation delay of the gate also specifies the speed of the logic gate.

The delay time is measured between 50% voltage levels of input and output waveforms. Figure 1 shows the input and output waveforms of an inverter. If t_{PHL} is the delay time when the output goes from low state (logic 0) to high state (logic 1) and t_{PLH} is the delay time when the output goes from high state (logic 1) to low

state (logic 0), the propagation delay time of the gate t_{pd} expressed as the average of the two delays as:

$$t_{pd} = \frac{t_{PHL} + t_{PLH}}{2}$$

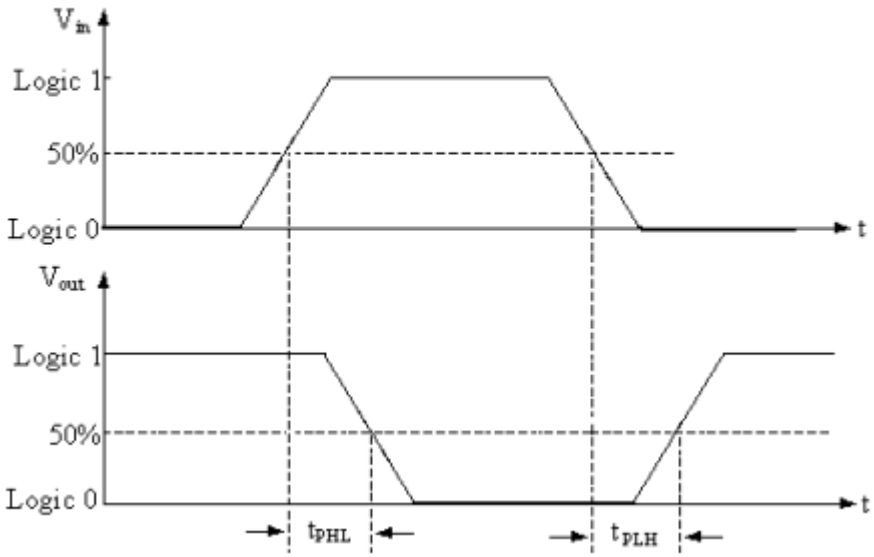


Fig 1.

(iv) **Power Dissipation:** It is defined as the amount of power that can be dissipated in an IC. It is calculated as the product of the d.c. voltage applied to an IC and the current drawn from the d.c. source. It is always desirable to have low power dissipation per gate. The normal working power per gate is required from few micro-watts to few milli-watts. The product of speed and power dissipation per gate is known as the figure of merit of the logic family. A low value of this product is desirable.

(v) **Operating Temperature:** The temperature range in which an IC functions properly is known as the operating temperature of the gate. It is specified by the manufacturer. The acceptable temperature range of the ICs is from 0 to $+70^0$C for commercial applications and this range is from -55^0C to 125^0C for military purposes.

(vi) **Noise Margin:** Spurious signals called noise are sometimes generated in the connecting leads of the logic circuits due to the stray electric and magnetic fields in the surroundings. This results the unpredictable operation of the logic circuit. The noise margin is sometimes called Noise- immunity. It is defined as the difference between the maximum permitted low input and the maximum guaranteed low

output, and that between the minimum permitted high input and the minimum guaranteed high output.

The idea of noise margin is illustrated in figure 2.

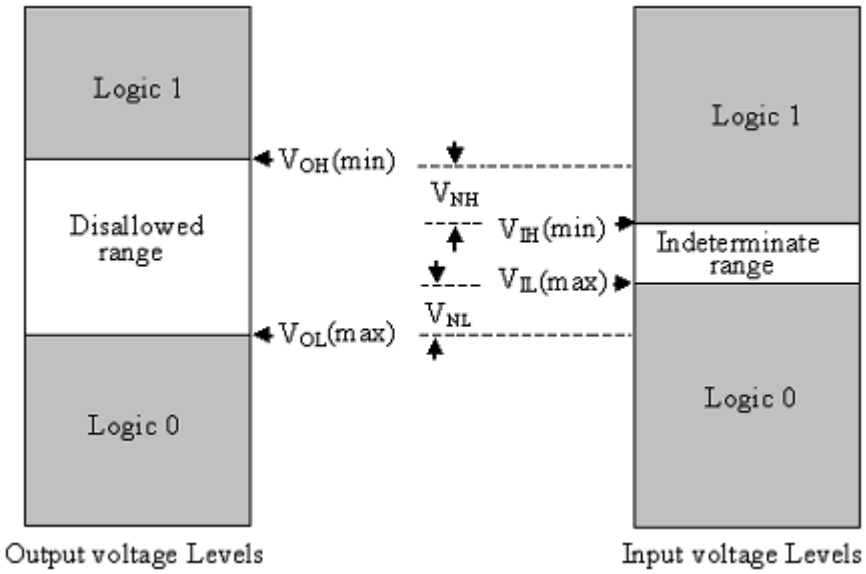


Fig 2.

Figure 2 shows that $V_{OH}(min)$ is the minimum high voltage for logic 1 and $V_{OL}(max)$ is the maximum low voltage for logic 0. The output should not occur in the disallowed range. Similarly, $V_{IH}(min)$ is the minimum high input voltage and $V_{IL}(max)$ is the maximum low input voltage and the voltage level between $V_{IH}(min)$ and $V_{IL}(max)$ is the indeterminate range and this voltage range should not be applied to the inputs of the logic gate.

As per definition of the noise margin, the noise margin for high state (V_{NH}) and the noise margin for low state (V_{NL}) are given by:

$$V_{NH} = V_{OH}(min) - V_{IH}(min)$$

$$V_{NL} = V_{OL}(max) - V_{IL}(max)$$

The large noise margin is always desirable.

When both the inputs A and B are at logic 0, the two transistors T_1 and T_2 will be in cutoff and no current flows through collector emitter circuit of the transistors. The output will, therefore, be high (logic 1). When either of the two inputs is at logic 1, the corresponding transistor will go into saturation and output will be $V_{CE,Sat}$ of the transistor (2.0 ≈ V). The output is said to be at logic 0. The output will also be low, if both the inputs are at logic 0, as both the transistors will saturate. It is concluded that it performs the operation of the NOR gate.

Though this is a simplest logic circuit yet it has become obsolete. RTL has the advantage that its power dissipation per gate is low. The disadvantages of this family are that it has low noise margin and its propagation delay is relatively larger.

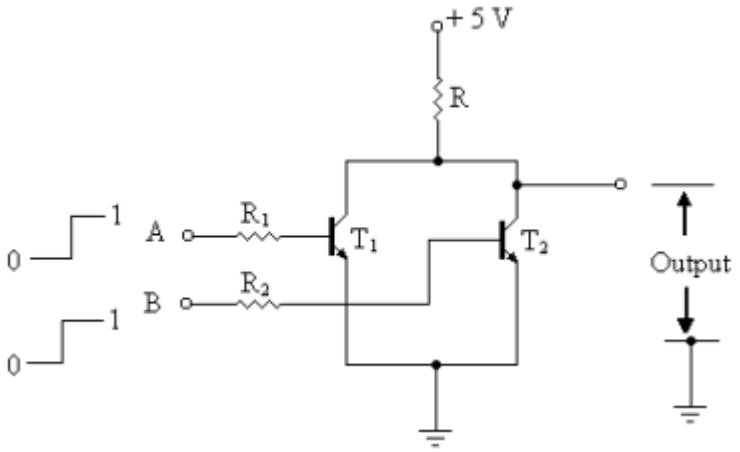


Fig. 3

Direct Coupled Transistor Logic (DCTL):

The direct coupled transistor logic circuit is similar to RTL, which obtained by omitting the base resistances in RTL. The DCTL circuit for two-input NOR gate is shown in figure 4. When one or both the inputs are high (logic1), the corresponding transistor or transistors will be conducting and the current flows through the resistance R gives the output low (logic 0).

It, however, corresponds to high output voltage when both the inputs are at low. This logic is very simple and requires a few components but it has the disadvantage of low noise margin.

Transistor – Transistor Logic (TTL):

The TTL is the most popular amongst all logic families and is widely used IC technology.. It is the modified form of DTL. The propagation delay time is reduced in TTL by using multi-emitter transistor in place of diodes. Figure 8 shows the schematic diagram of a basic TTL positive logic NAND gate. It consists of a multi-emitter transistor T_1. A two emitter transistor is equivalent to two transistors with common base and common collector as shown in figure 9.

The operation of TTL NAND gate may be explained as follows:

When either of two inputs A or both the inputs are at logic 0, emitter base junction of the multi-emitter transistor will be in forward bias and base current is supplied by the resistor R1. The transistor T1 saturates and the voltage at the point will

be equal to VCE,Sat of the transistor (2.0 ≈ V). The transistor T2 will be in cutoff and output voltage will be high (logic 1).

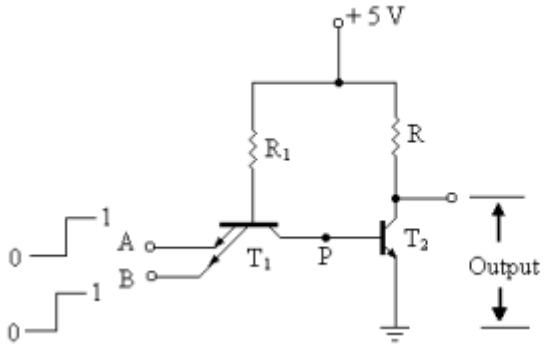


Fig. 8

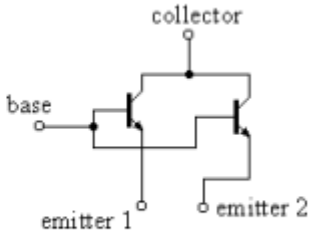


Fig. 9

When both the inputs are at logic 1 (+5 V), the emitter base junctions of transistor T_1 will be reverse biased and current will flow, through R_1 and through the forward biased base collector junction of transistor T_1 into the base of transistor T_2. In this mode the transistor is said to be operated in the inverted mode, as the collector of transistor T_1 operates as emitter and the emitter as collector. The voltage at the point P will be sufficient to drive the transistor T_2 into saturation, the output voltage will therefore, be equal to $V_{CE,Sat}$ (2.0 ≈ V) or logic 0.

The propagation delay time of this gate is smaller than that of DTL NAND gate, since when the transistor T_2 goes into cutoff region from saturation region, the transistor T_1 saturates and provides a low impedance path to ground. Thus the stored base charge of the transistor T_2 is quickly removed thereby reducing the propagation delay time.

The output resistance of the basic TTL circuit (fig. 8) is low when the transistor T_2 saturates or output is low (logic 0). However, the output resistance of this circuit is almost equal to the resistance R, when the transistor is in cutoff or output is high (logic 1). This will restrict the fan out of the gate. The reduction in resistor R would increase the power dissipation in R and in the gate. Also the reduction in the value of R would difficult to saturate the transistor T_2. To overcome this difficulty, TTL gate with totem pole arrangement is used.

TTL NAND Gate with Totem-pole Output:

Figure 10 shows the standard form of a TTL circuit with input NAND gate. The circuit works as follows:

When either the inputs or both the inputs are low (logic 0), the transistor T_2 goes into cutoff. The transistor T_4 will also be in cutoff, as the voltage drop across the resistor R_3 is nearly zero. Now the transistor T_3 conducts and works as emitter follower. The output voltage available at the emitter of this transistor will be equal to the collector voltage of the transistor T_2, which is high (logic1). The emitter follower, however, provides a low output resistance to the input of the driven gate.

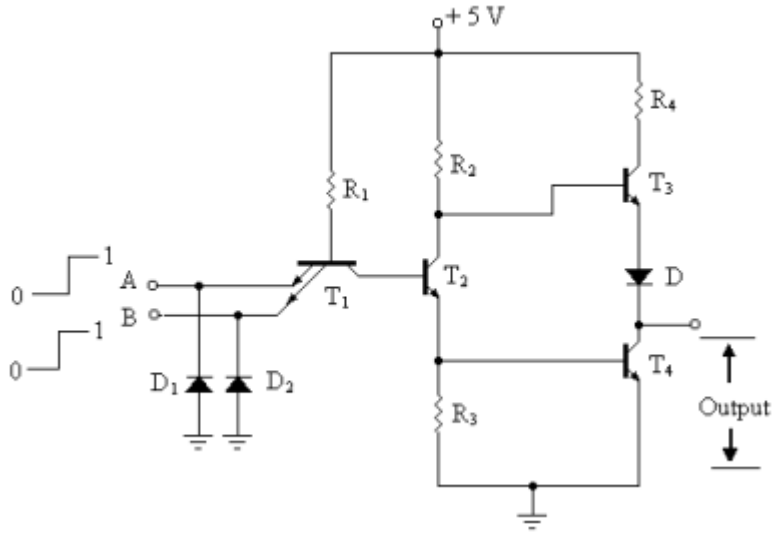


Fig. 10

When both the inputs are high (or at logic 1), transistor T_2 conducts and acts as an emitter follower. The potential across R_3 will be sufficient to drive the transistor T_4 into saturation. Because the transistor T_4 saturates, the output voltage will, therefore, be equal to $V_{CE,Sat}$ (2.0 ≈ V) or logic 0. Since this output is taken at the collector of the transistor T_4, which is in saturation, so it provides the low output impedance. The diode D prevents the transistor T_3 from being conducting when the transistor T_4 saturates. The potential across the emitter base junction of the transistor T_4 is approximately 0.8 V ($V_{BE,Sat}$) and collector emitter voltage of T_2 is 0.2 V ($V_{CE,Sat}$). This means a total of 1.0 V is applied to the base of transistor T_3. In the absence of the diode D, this voltage would be sufficient for the conduction of the transistor T3. The diode D, however, reduces the base emitter voltage of transistor T3 below 0.7 V, required voltage for the conduction of a transistor. Thus the diode D drives the transistor T3 into cutoff when T4 saturates.

Diodes D_1 and D_2 protect the transistor T_1 from being damaged when the negative spikes of the voltage appears at the inputs. When the negative spikes appear at the input terminals the diodes conducts and the spikes are grounded. The transistors T_3

and T_4 and the diode D form the totem pole output, which provides the low output impedance in every case. The TTL gates are faster having the propagation delay of about 15 nsec.

TTL Inverter:

Figure 11 shows a TTL circuit for an inverter. The operation principle of this is same as discussed for TTL NAND gate, with the difference that it has only one input. So when input A is at logic 0, output will be high (logic 1) and if input is high (logic 1), output will be low (logic 0). This circuit also has the totem-pole output.

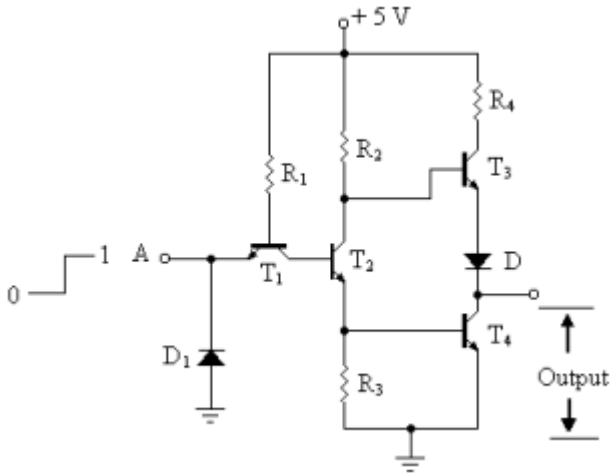


Fig. 11

TTL NOR Gate:

Figure 12 shows a TTL circuit for two input NOR gate. It consists of two input transistors T_1 and T_2 and two other transistors T_3 and T_4 connected in parallel which acts as a phase splitter. In addition the output is obtained using the totem pole circuit comprising transistors T_3, T_4 and diode D. The operation of this circuit may be explained as follows:

When both the inputs are low, the emitter base junctions of the input transistors will be in forward bias, no current will flow through the base of transistors T_3 and T_4. So these transistors will be in cutoff. The transistor T_5 will, therefore, conduct and T_6 will be in cutoff, producing a high (logic 1) output.

When input A is low and input B is high, the transistor T_3 is in cutoff and transistor T_4 saturates. The transistor T_6 will, therefore, conduct and T_5 will be in cutoff, producing a low (logic 0) output.

Similarly, when input A is high and input B is low, the transistor T_4 saturates and transistor T_3 goes in cutoff. The transistor T_6 will, therefore, conduct and T_5 will be in cutoff, producing a low (logic 0) output.

When both the inputs are high, the emitter base junctions of the input transistors will be in reverse bias, the current will flow through the base of transistors T_3 and T_4. So these transistors will saturate. The transistor T_6 will, therefore, conduct and T_5 will be in cutoff, producing a high (logic 0) output.

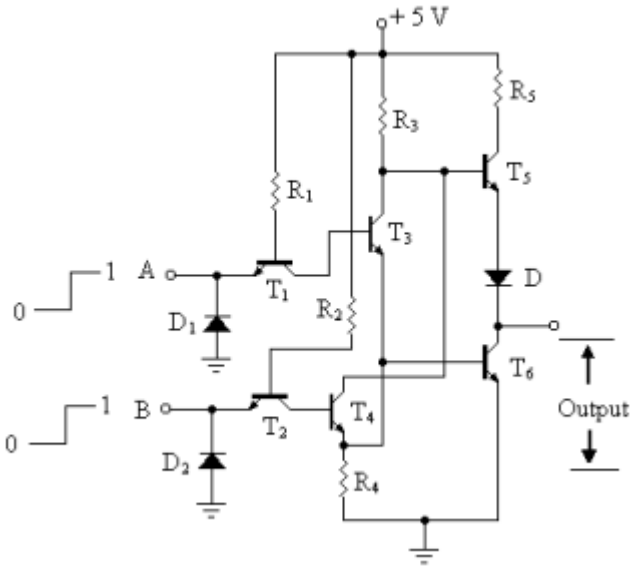


Fig. 12

TTL AND Gate :

Figure 13 shows a TTL two input AND gate. The AND operation is obtained by inserting an extra inversion circuit before the totem output of the TTL NAND gate. This converts the NAND gate to an AND gate. The extra inversion circuit comprises the transistors T_2 and T_3.

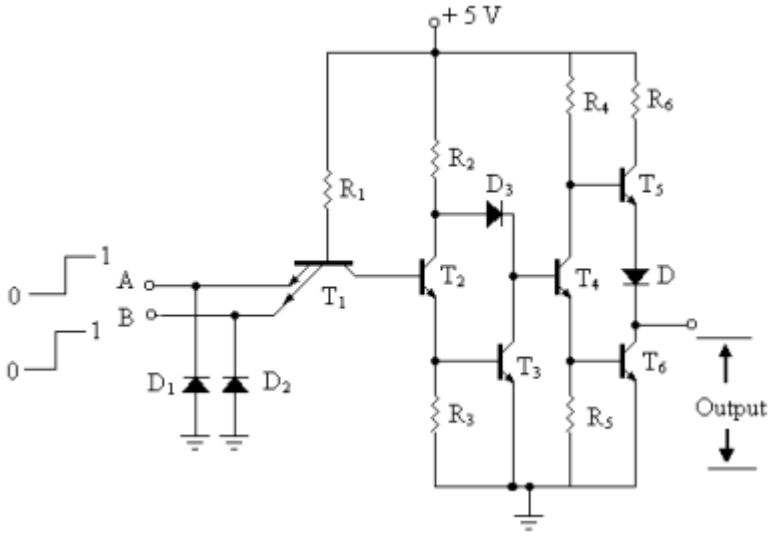


Fig. 13

TTL OR Gate :

The TTL OR gate is obtained by inserting a common emitter circuit before the totem pole output of the TTL NOR gate as shown in figure 14. The common emitter circuit provides an inversion, which converts the NOR gate to an OR gate. The transistor T_5 with associated components forms the common emitter circuit.

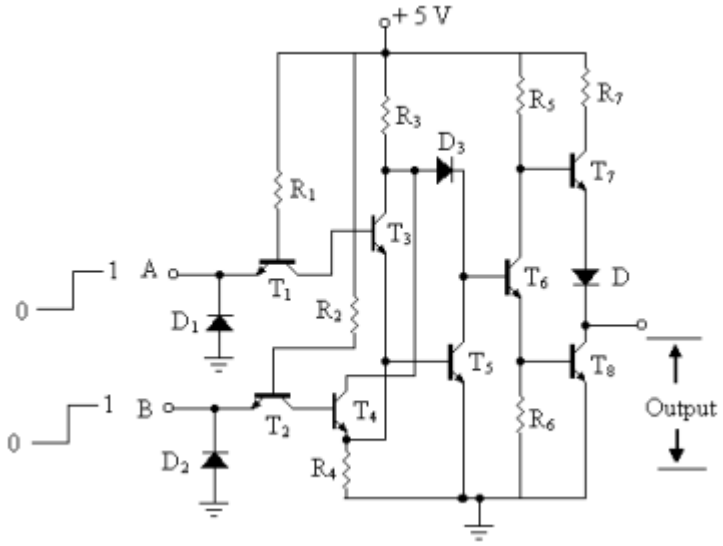


Fig. 14

Schottky Transistor – Transistor Logic (STTL):

In Schottky TTL circuits, the operation speed is much more larger than the high speed TTL circuits. The transistors used in TTL circuits take certain time when the transistors switch form saturation to cutoff. This limits the propagation delay of the gates. This delay can however, be reduced by replacing the transistors in TTL circuits by the Schottky transistors. The Schottky transistor is formed by connecting Schottky barrier diode between base and collector of a transistor as shown in figure 22. The Schottky barrier diode (SBD) has a forward drop of only 0.25 V, it therefore prevent the transistor from saturating fully. Figure 23 shows the circuit diagram of two-input Schottky TTL NAND gate. Notice the transistor T_4 is the ordinary transistor.

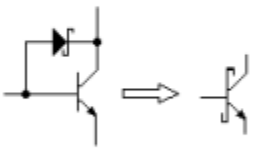


Fig. 22

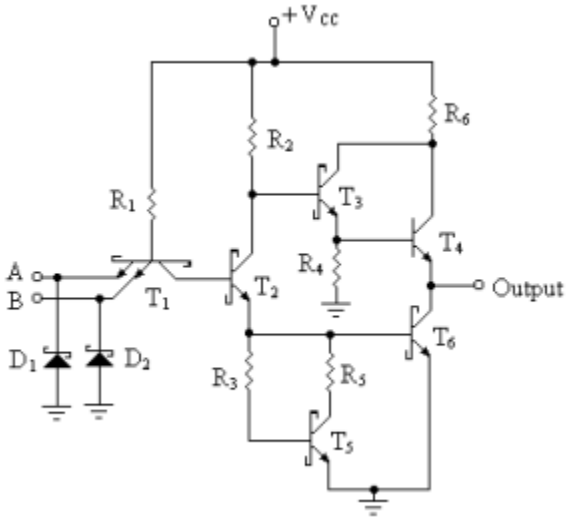


Fig. 23

The 54S/74S series is available for Schottky Transistor – Transistor Logic (STTL) gates. This series of logic family has less power consumption as compared to 54H/74H series and the speed is double to that of 54/74 series. Still low power

54LS/74LS series of Schottky TTL is available. This series is obtained by increasing the resistances used in 54S/74S series. This family of logic gates therefore has the same switching speed as that of standard TTL family (54/74), and the power dissipation is 1/5 of the 54/74 series.

Emitter Coupled Logic (ECL):

Emitter Coupled Logic (ECL) circuits fall in the category of non-saturated digital logic family i.e. the transistors in this family do not saturate. This eliminates the storage time delay, so the speed of operation of this family is increased. This logic family has the fastest speed and propagation delay time per gate is approximately 1 nsec.

Figure 24 shows the basic circuit of four-input ECL OR/NOR gate. The outputs provide both OR and NOR functions. The transistors T_1 through T_5 form the differential amplifier circuit, transistor T_6 forms the internal temperature and voltage compensation bias network and the transistors T_7 and T_8 gives the emitter follower outputs for OR and NOR functions. Logic levels for this family are negative, – 0.9 V is assumed for logic 1 and – 1.75 V for logic 0. The operation of this circuit may be explained as follows:

When all the inputs are at low (– 1.75 V), the transistors T_1 through T_4 are off, as emitter base junctions are reverse biased. The transistor T_5 is conducting not saturated. Due to the proper biasing of the transistor T_6, the base of transistor T_5 remains at – 1.29 V. Therefore its emitter is at – 2.09 V which is 0.8 V below the base voltage. The transistor T_5 therefore conducts. The differential voltage between base and emitter of the transistors T_1 through T_4 is about –0.34 V, so they are in cutoff. The emitter follower transistors T_7 and T_8 give the outputs – 1.75 V (logic 0) and – 0.9 V (logic1) respectively.

When any one or all the inputs are at – 0.9 V (logic1), in that condition the corresponding transistor or transistors will conduct. The voltage at the emitters of T_1 through T_5 therefore rises to – 2.09 V. Since the base of transistor T_5 is held constant at – 1.29 V due to the bias network, it goes into cutoff. The emitter follower transistors T_7 and T_8 give the outputs – 0.9 V (logic1) and – 1.75 V (logic 0) respectively. Symbolic representation of OR/NOR ECL gate is shown in figure 25.

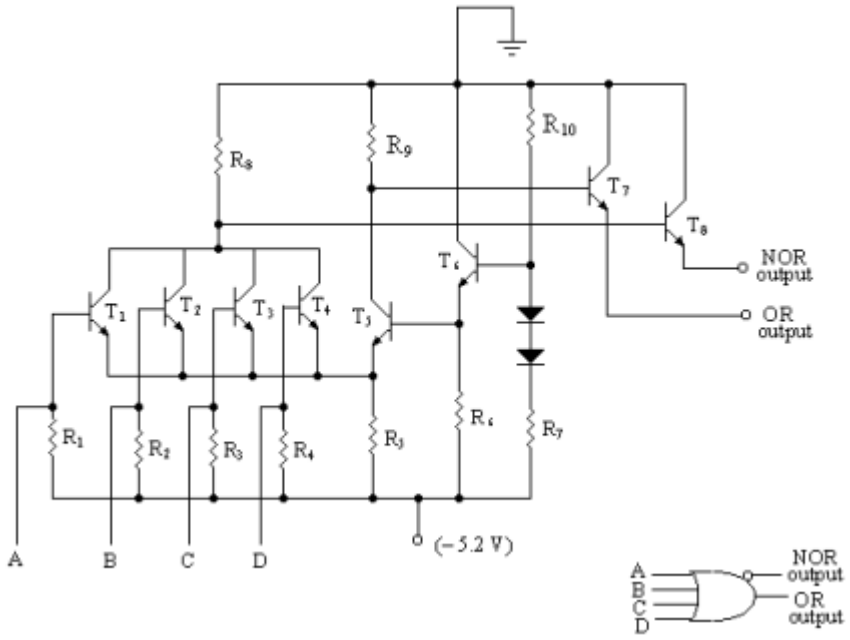


Fig. 24 Fig. 25

The wired logic can be formed by connecting together the outputs of two or more ECL gates as shown in figure 26. The external -wired connection of two NOR outputs produces a wired –OR function. The internal –wired connection of two OR outputs in some ECL ICs is used to produce a wired –AND logic.

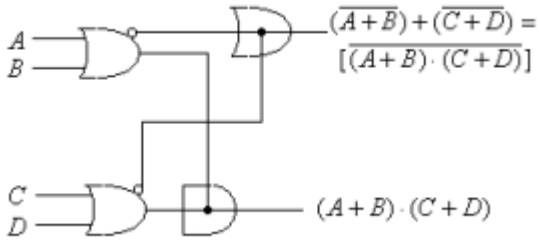


Fig. 26

MOS Logic:

The logic families discussed so far were based on bipolar transistor. Their comparisons were made with respect to certain parameters of the logic family. One more logic family based on the unipolar devices such as Metal Oxide Semiconductor field effect transistor (MOS FET) will now be discussed. The MOS logic family is the simplest to fabricate and occupies less space. It requires N channel MOS or P channel MOS field effect

transistors and no other components such as resistors, diodes etc. This logic family has the high packing density, low power dissipation and high fan-out.

The logic circuits may be designed using NMOS (enhancement type N channel MOS FET's) or PMOS (enhancement type P channel MOS FET's). From the operations of MOS FET's one can note following characteristics of MOS FET's. The NMOS conducts when gate is at a positive potential with respect to source and PMOS, however, conducts when gate is at a negative potential with respect to source. If the gate is at zero potential neither of the two MOS FET's will conduct.

MOS inverter:

Figure 27 shows the circuit diagram for NMOS inverter and figure 28 shows for PMOS inverter. The working operation of the circuits is same. The MOS FET T_1 in both the circuits work as resistor since T_1 is conducting as gate is connected to drain.

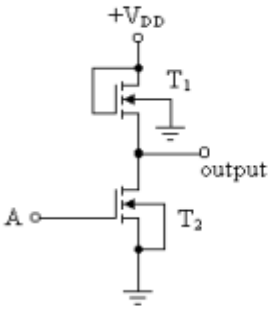


Fig. 26

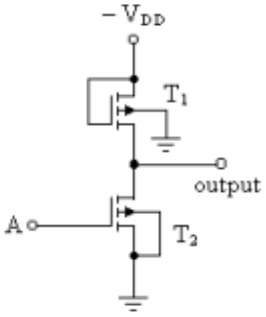


Fig. 27

In figure 26 when input A is at logic 0 (ground potential), the MOS FET T_2 will be OFF giving the high voltage at the output. So the output is at logic 1. If on the other hand input A is at logic 1 (VDD potential), the MOS FET T_2 will be ON and output will be at logic 0. This verifies the operation of inverter. The operation of PMOS will be discussed in the similar fashion with the only difference that it works for negative logic.

MOS NOR gate:

Figure 28 shows the circuit diagram of NMOS positive logic three-input NOR gate and figure 29 for PMOS negative logic three-input NOR gate. In NMOS NOR gate (ref. fig. 28), when all the three inputs are at logic 0 (ground potential), MOS FET's T_2 through T_4 will be off giving the high output (logic 1). If all the three inputs or any (one or two) of the three inputs are at logic 1, the corresponding MOS FET or MOS FETs will conduct giving low output (logic 0). This verifies the operation of positive logic NOR

gate. The working operation of PMOS NOR gate may be explained in the similar which works for negative logic. The MOS FET T_1 acts as a resistor in both the circuits.

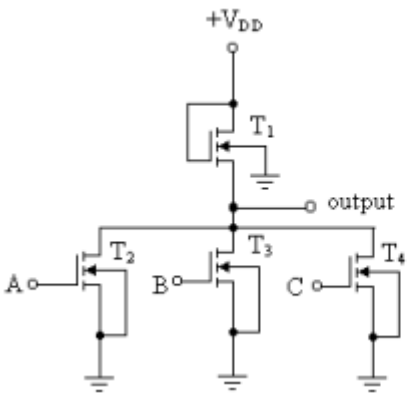


Fig. 28

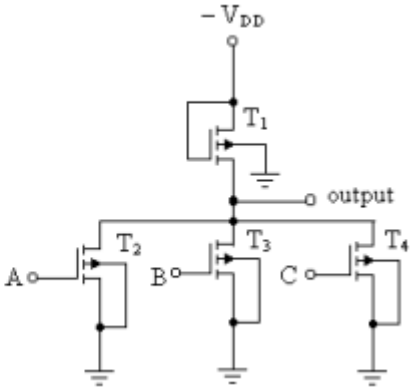
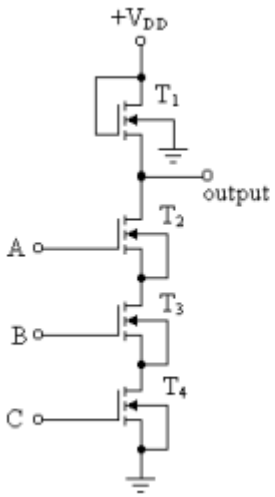


Fig. 29

MOS NAND gate:

Three input NAND gate with NMOS and PMOS transistors are shown in figure 30 and 31 respectively. The NMOS NAND gate works with positive logic and PMOS NAND gate work with negative logic. The working of NMOS NAND gate is explained as follows (ref. Fig 28).

The NMOS FET's T_2 through T_4 will conduct when all the three inputs are at logic 1 (+ VDD), giving the output low (logic 0). When either of the three inputs or any (one or two) of the inputs is at logic 0 (ground potential), the corresponding MOS FET or MOS FET's will be off giving the high output (logic1). This verifies the operation of positive logic NAND gate, Similarly, one can explain the operation of PMOS NAND gate which work with negative logic.



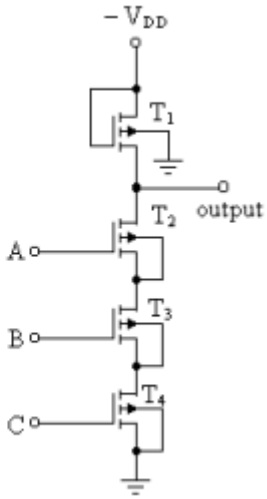


Fig. 30 Fig. 31

Complementary MOS (CMOS) Logic:

The complementary metal oxide semiconductor (CMOS) logic family contains both enhancement type P-channel and N-channel MOS FET's arranged in a complementary connection. The power consumption of CMOS logic family is very less as neither of P-channel or N-channel MOS FET's conducts simultaneously when no signal is applied to the input terminals of the logic.

Thus only the leakage current flows between the terminals of the supply. The CMOS gate can be operated on wide range of supply voltage between 3 V to 15 V. It has good noise margin better than TTL devices. Fan-out of this is much larger. The speed of the CMOS logic is comparable with that of TTL circuits but larger than Schottky TTL circuits.

CMOS Inverter:

Figure 32 shows the circuit diagram of CMOS inverter which consist of a PMOS transistor T_1 and an NMOS transistor T_2 which are connected in complementary mode. The drains of both the transistors are connected together, through which the output is taken. The source terminal of PMOS transistor T_1 is connected to the positive supply, where as the source of the NMOS transistor T_2 is grounded.

When the input A is grounded (logic 0), the gate of PMOS transistor T_1 is at the negative potential with respect to its source, so it is ON. The gate of NMOS transistor T_2 is at ground potential, so it is off. The output is, therefore, high (+VDD), logic 1.

If on the other hand input A is high (logic 1), the gate of PMOS transistor T_1 is at zero potential with respect to its source, so it is off. The gate of NMOS transistor T_2 is at the positive potential with respect to ground, so it is ON. The output is, therefore, low logic 0.

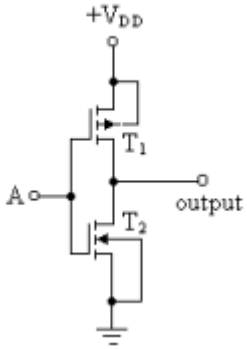


Fig. 32

CMOS NAND Gate:

The circuit diagram of CMOS NAND gate is shown in figure 33. The two PMOS transistors T_1 and T_2 are connected in parallel with the sources connected together and two NMOS transistors T_3 and T_4 are connected in series.

When both the inputs are at logic 0 (grounded), the gates of T_1 and T_2 are at negative potentials with respect to their sources; the gates of T_3 and T_4 are at zero potential. So both PMOS transistors (T_1 and T_2) are ON and NMOS transistors T_3 and T_4 are off. The output will, therefore, be high (logic 1).

When input A is at logic 0 (grounded) and input B is at logic 1, the gate of T_1 is at negative potential with respect to its source and the gate of T_2 will be zero; the gates of T_4 and T_3 are at zero potential and VDD potential respectively. So T_1 and T_3 are ON and T_2 and T_4 are off. The output will, therefore, be high (logic 1).

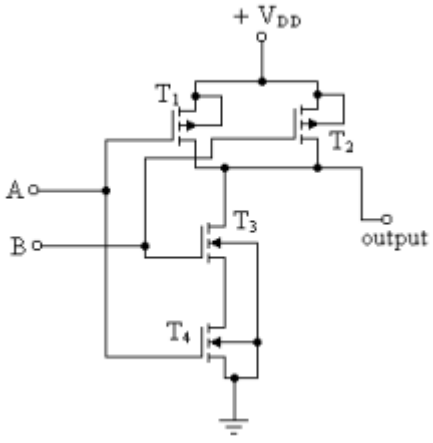


Fig. 33

When input A is at logic 1 and input B is at logic 0 (grounded), T_1 and T_3 will be off and T_2 and T_4 will be ON. The output will, therefore, be high (logic 1). When both the inputs are at logic 1 (+VDD), the gates of T_1 and T_2 are at zero potential; the gates of T_3 and T_4 are at negative potentials with respect to their sources. So both

PMOS transistors (T_1 and T_2) are off and NMOS transistors T_3 and T_4 are ON. The output will, therefore, be grounded (logic0).

CMOS NOR Gate: :

The circuit diagram of CMOS NOR gate is given in figure 34. The two PMOS transistors T1 and T2 are connected in series and two NMOS transistors T3 and T4 are connected in parallel.

When both the inputs are at logic 0 (grounded), the gate of T1 and T2 are at negative potentials; the gates of T3 and T4 are at zero potential. So both PMOS transistors (T1 and T2) are ON and NMOS transistors T3 and T4 are off. The output will, therefore, be high (logic 1).

When input A is at logic 0 (grounded) and input B is at logic 1, the gate of T1 is at negative potential with respect to its source and the gate of T2 will be zero; the gates of T3 and T4 are at zero potential and VDD potential respectively. So T1 and T3 are ON and T2 and T4 are off. The output will, therefore, be low (logic 0).

When input A is at logic 1 and input B is at logic 0 (grounded), T1 and T3 will be off and T2 and T4 will be ON. The output will be low (logic 0).

When both the inputs are at logic 1 (+VDD), the gates of T1 and T2 are at zero potential; the gates of T3 and T4 are at VDD potential. So both PMOS transistors (T1 and T2) are off and NMOS transistors T3 and T4 are ON. The output will, therefore, be grounded (logic0).

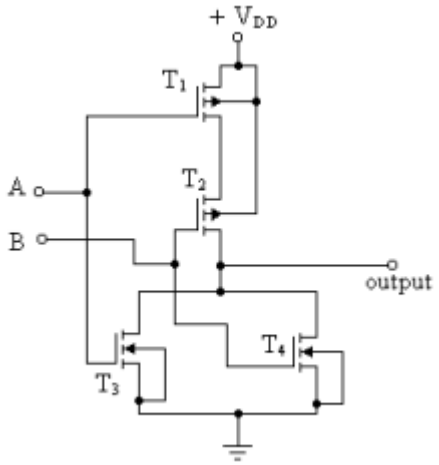


Fig. 34

Comparison of Logic Families:

The comparison of important logic families are given in table 1 in respect of logic parameters.

Table 1

Logic Parameters	RTL	DTL	TTL	ECL	MOS	CMOS
Basic gates with +ve logic	NOR	NAND	NAND	OR/NOR	NAND	NAND/NOR
Maximum fan-in	5	10	8	5	8	8
Fan-out	5	8	10	25	20	>50
Power dissipation / gate (in mW)	12	10	10	50	1	0.01 static at 1 MHz
Propagation delay per gate (nsec)	20	30	12	4	400	70
Noise immunity	Nominal	Good	Very Good	Good	Nominal	Very good
Number of functions	High	Fairly high	Very high	High	Fair	Good
Clock rate, MHz	5	12	15	300	2	5

Following is the list of general TTL gates available in the form of 54/74 series SSI.

54/7400	Quad two-input NAND gate
54/7402	Quad two-input NOR gate
54/7403	Quad two-input NAND gate with open collector
54/7404	Hex inverter
54/7405	Hex inverter with open collector
54/7406	Hex inverter buffer
54/7407	Hex buffer
54/7410	Triple three-input NAND gate
54/7411	Triple three-input AND gate
54/7420	Dual four-input NAND gate
54/7430	Single eight-input NAND gate
54/7440	Dual eight-input NAND buffer

Following is the list of general CMOS gates available in the form of CD 40 series SSI

4000	Dual three-input NOR gates plus one inverter
4001	Quad two-input NOR gate
4002	Dual four-input NOR gate
4011	Quad four-input NAND gate
4012	Dual four-input NAND gate
4023	Triple three-input NAND gate

14.4 INTRODUCTION TO VERILOG

Brief History of Verilog

The Verilog Hardware Description Language (HDL) came into existence in 1984. It was a proprietary language from Gateway Design Automation. Cadence Design Systems acquired Gateway Design Automation in 1989 and released the Verilog HDL to the public domain. Next, Open Verilog International (OVI) was formed to promote usage of the Verilog. OVI prepared the Verilog Reference Manuals and requested to the IEEE to standardise the frame work of the language. The IEEE formed a standards working group in 1993 to create the standard and in December 1995 Verilog HDL officially became IEEE 1364-1995 standard (**Verilog 1.0**). After publishing this standard, the 1364 Working Group started collecting feedback from the Verilog HDL users all over the world. A significantly revised version based on this feedback was published in 2001 as IEEE 1364-2001 (**Verilog 2.0**) standard. Further development continued in the Verilog community to identify outstanding issues with the language as well as ideas for possible enhancements. This resulted into Verilog-2001 plus an extensive set of high-level abstraction extensions called as a new standard, SystemVerilog (**SystemVerilog 3.x**). However, additional issues were identified which could possibly have led to incompatibilities between Verilog 1364 and SystemVerilog. To mitigate this situation, the IEEE P1364 Working Group was established. This subcommittee of the SystemVerilog P1800 Working Group ensured the consistency by releasing one more Verilog standard in 2005 as IEEE 1364-2005 standard. This standard corrects and clarifies features ambiguously described in the 1995 and 2001 editions.

14.4.1 Basics of Verilog

The intention of any HDL is to describe hardware design and we've seen that any logical circuit can have four possible input values — logic high (1), logic low (0), don't care (x) and tristate (z). Therefore,

Verilog supports four values, viz. 0, 1, x, and z. It is a case sensitive language and all the reserved keywords, given in Table 14.1, must be written in lowercase.

Table 14.1 *Some of the Reserved Keywords in Verilog HDL*

and	always	assign	attribute
begin	buf	bufif0	bufif1
case	cmos	deassign	default
defparam	disable	else	endattribute
end	endcase	endfunction	endprimitive
endmodule	endtable	endtask	event
for	force	forever	fork
function	highz0	highz1	if
initial	inout	input	integer
join	large	medium	module
nand	negedge	nor	not
notif0	notif1	nmos	or
output	parameter	pmos	posedge
primitive	pulldown	pullup	pull0
pull1	rcmos	reg	release
repeat	rnmos	rpmos	rtran
rtranif0	rtranif1	scalared	small
specify	specparam	strong0	strong1
supply0	supply1	table	task
tran	tranif0	tranif1	time
tri	triand	trior	trireg
tri0	tri1	vectored	wait
wand	weak0	weak1	while

One can insert single line comment anywhere in the code using // which ends at the end of the line. A block of comments starts with /* and it is terminated with */. Any single line comment (//) within such block comment has no meaning. User can provide a unique name to an object using an *identifier* so that it can be referenced. Such an identifier can be any sequence of letters, digits, dollar signs ($), and underscore characters (_). But the identifier should not begin with a digit or $ character. Below are some examples of identifiers:

```
cnt    //valid identifier
dout_bus     //valid identifier
8_bit_bus    //invalid identifier
```

The purpose of any hardware chip to process some input signal and produce some output. Usually, the problem statement provides us the relationship between the input and output signals and as a design engineer one has to identify in what way one has to lay the logic components to achieve the expected relationship. Since the internals of the chip are unknown in the initial stage let us call it as a black box. In the previous chapters, we have studied that a signal flow in a chip can be either unidirectional (only input, only output) or bidirectional (input-output), as shown in the Fig. 14.3. It means the circuit laid inside the black box communicates with the external environment through this interface. This entails us that the IO pins of any chip are nothing but the communication ports of that particular black box. Verilog reserves input, output, and inout keywords to declare such IO ports and hence the signal directions. By default, all input and output terminals are single bit and called as a scalar port. Sometimes we need to declare a signal bus which is called as a vector port which are declared as shown below.

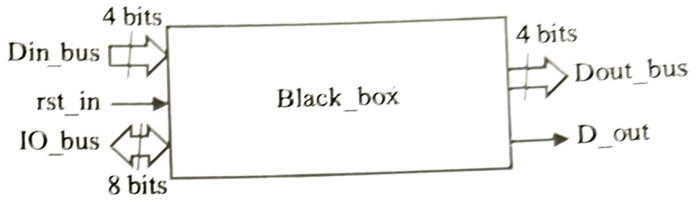


Fig. 14.3 Black Box Design

```
input rst_in;                  //input port (scalar) declaration
input [4:0] Din_bus;           //4 bits input port (vector) declaration
inout [7:0] IO_bus;            //8 bits IO port (vector) declaration;
                                 bidirectional bus declaration
output [3:0] Dout_bus;         //4 bits output port (vector) declaration
output D_out;                  //output port (scalar) declaration
```

14.4.2 Design Abstraction Levels

Once the interface with this black box is clear, we can focus on the internal details and its design paradigm. Four levels of abstraction are provided in Verilog for the logic design purpose and designer are free to use it as per requirement. So, one can either use any one abstraction level or mix and match all the abstraction levels to attain the design functionality. Below are the four abstraction levels.

Behavioural or algorithmic level: It is the highest level of circuit abstraction which is provided in Verilog. The relationship between input and output, i.e., behaviour of a black box is coded as per the algorithm using procedural constructs. So, the hardware design code looks like C program and the inner details of the design are unclear.

Dataflow level: Here, the designer has an idea how data is flowing in between the registers and how the data is being processed. Usually, designer has a rough or block level idea about the hardware to be implemented to attain the functionality. But complete gate level circuit is unclear and the hardware implementation job is left on the synthesis tools. Most of the time the combination of behaviour- and dataflow- modelling known as Register Transfer Level (RTL) modelling is used to realise the black box design by coding the data flow through the logic circuit.

Gate level: It is also known as structural design. The designer has clear idea about the logic gates used and their pin-to-pin connectivity. Entire functionality of a black box is described using logic gates and there is no ambiguity about the connections to be made. But, if the gate count of a circuit is more than

hundreds of gates, keeping track of the connectivity is very difficult and time-consuming work which is contradictory to the objectives of using HDL. Further, if there is any change is design specifications one need to rework on the entire design.

Switch level: It is the lowest level of abstraction which is supported by Verilog. It aims at the library cell development. It is possible to implement entire design using switches, i.e., transistors as every gate is made up of switches, refer to the concepts covered in Chapter 4. Similar to gate level modelling, it is difficult to modify such a circuit to accommodate small changes in the design specifications. It is very difficult to migrate the design with change in technology node again and again. Further, today's designs comprise of millions of gates and laying so many transistors without any issues is over ambitious task.

14.4.3 Verilog Constructs and Semantics

There are several constructs for each of these abstraction level design modelling. They are categorised into different types viz., synthesisable constructs or non-synthesisable constructs, as shown in Fig. 14.4. The aim of any

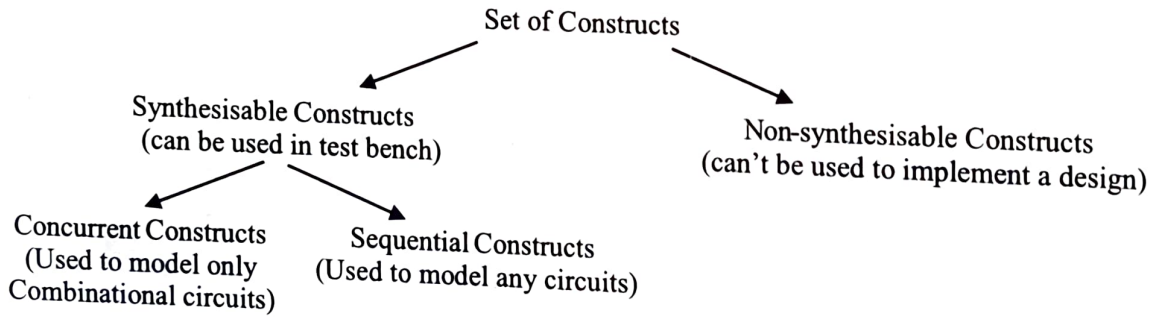


Fig. 14.4 Classification of the Constructs in any HDL

HDLs is to implement some design on a silicon chip and if a construct supports this aim, it is called as a synthesisable construct. If all constructs are unable to support this aim, what is the fun of having non-synthesisable constructs? The idea is simple. Once you design a circuit, you need to simulate it and check whether the design performs as intended or not. In this case, your intension is not mapping anything on the silicon chip rather to test the designed black box. You are interested in creating certain test vectors. It means, you would like to pass-on pattern of some 1s and 0s with certain time delay. It is a bad idea to study or use another language just for testing purpose. Hence, HDLs have some non-synthesisable constructs too. One can create a circuit testing environment using these constructs which is also known as test bench writing. Further, synthesisable constructs are grouped in two categories concurrent and sequential statements. Preferably concurrent statements are used while designing a combinational logic circuit. But, one can use sequential statements to design a combinational or a sequential logic circuit.

Note: Test bench can have both synthesisable as well as non-synthesisable constructs. But your design file should have only synthesisable constructs as you wish to map the designed circuit on a silicon chip.

Two main data types 'wire' and 'register' are associated with these statements. One can use 'wire' or 'reg' keywords to declare these data types. All tools insert 1-bit wide wire for every signal and initialise the nets with a tristate value, i.e., 'z' except the 'trireg' net. When concurrent statements are used to implement a design functionality the LHS side signals must be declared as 'wire.' This declaration is incapable of holding the signal value and hence, a continuous net driver is required. This satisfies the requirement of the combinational logic gates wherein instantaneous changes at the input ports must be sensed to produce the output signal. On the contrary, 'reg' is not a default data type and it must be declared explicitly. Its default value is 'x' and it is capable of holding the signal value until the previous value is not overwritten. A register declared in this way is other than the register used in a hardware design. This keyword creates a temporary memory location in the tool wherein the current signal value is stored. Since the signal value is written in the temporary memory location, one can change its value by replacing the previous data. This is the actual requirement of the sequential circuits which change the circuit output based on either the active clock edge or asynchronous signals used in the circuit design. Therefore, sequential statements are used to implement a sequential circuit design functionality with the LHS side signals declared as a 'reg.'

Four types of number systems —binary, octal, decimal, and hexadecimal— are used to represent the sized numbers. Number base format (i.e., radix) is case insensitive so one can use b or B to represent a binary number. Similarly, o, d, and h can be used to octal, decimal, and hexadecimal numbers, respectively. Following construct is used for this purpose:

<size>'<radix><number>.

The **<radix>** field represents which number system is used to define the value of the number. The **<value>** field represents the value of the number and it must contain digits which are permissible in the given radix system. The **<size>** field indicates how many bits (i.e., data width) are used to represent the value of the number. The data width must be sufficient enough to represent the given number. These concepts are explained in following examples.

```
6'b01_1011          //is a 6-bit binary number. Use of _ is permitted. It
                      improves readability.
8'b1001_xxzz        //is a 8-bit binary number whose lower nibble has don't
                      care and tristate bits
5'D 23              //is a 5-bit decimal number
12'hx               //is a 12-bit unknown number
16'hz               //is a 16-bit high-impedance number
659                 //is a unsized decimal number which is a default number
                      format.
'h 837FF            //is a unsized hexadecimal number
'o7460              //is a unsized octal number
4af                 // is an illegal declaration because number format is
                       missing
```

All Verilog design files start and end with keyword module and endmodule, respectively. These files are saved with a .v extension and their nesting is not permitted.

14.5 INTRODUCTION TO VHDL

VHDL is a Hardware Description Language used for modelling digital systems made of interconnection of components. The complexity of the digital system being modelled may vary from that of a simple gate to a complete electronic circuit/system. VHDL is an acronym for VHSIC Hardware Description Language and VHSIC is an acronym for Very High Speed Integrated Circuits.

VHDL is an industry standard language used to describe hardware from the abstract to the concrete level and has become the universal communication medium of design and for specifiying input and output from various design tools, such as simulation tools, synthesis tools, placement and routing tools, etc. from vendors of CAD work stations, ASICs, CPLDs, FPGAs, etc.

VHDL contains elements which can be used to describe the *behaviour* (*concurrent* or *sequential*) or structure, with or without timing, of any digital system irrespective of the complexity of the sytem. Since it is the most widely used hardware description language, therefore, it is a commonly used method of documenting circuits by the designers, thus making it possible to understand circuits designed by other designers.

For computer aided design of digital systems, VHDL is a very convenient and commonly used HDL for design entry. It supports hierarchical modelling of the system and top-down and bottom-up methodologies of design. Models written can be verified using a VHDL simulator.

When a digital circuit is to be designed, it is required to be described in VHDL. For this we must specify an *entity* and an *architecture* at the top-level, and also specify an entity and architecture for each of the component modules that are part of the circuit. Each entity declaration includes a list of interface signals and their types that can be used to connect it to other modules or to the outside world. The behaviour of the circuit and each of its components is described in the architecture declaration. The behaviour may be described in structural form, i.e., interconnection of components, or as a set of concurrent or sequential statements.

Figure 14.10(a) illustrates the basic concept of VHDL. It has one entity declaration and an architecture. In general, a digital system or circuit may be composed of a number of sub-systems or components.

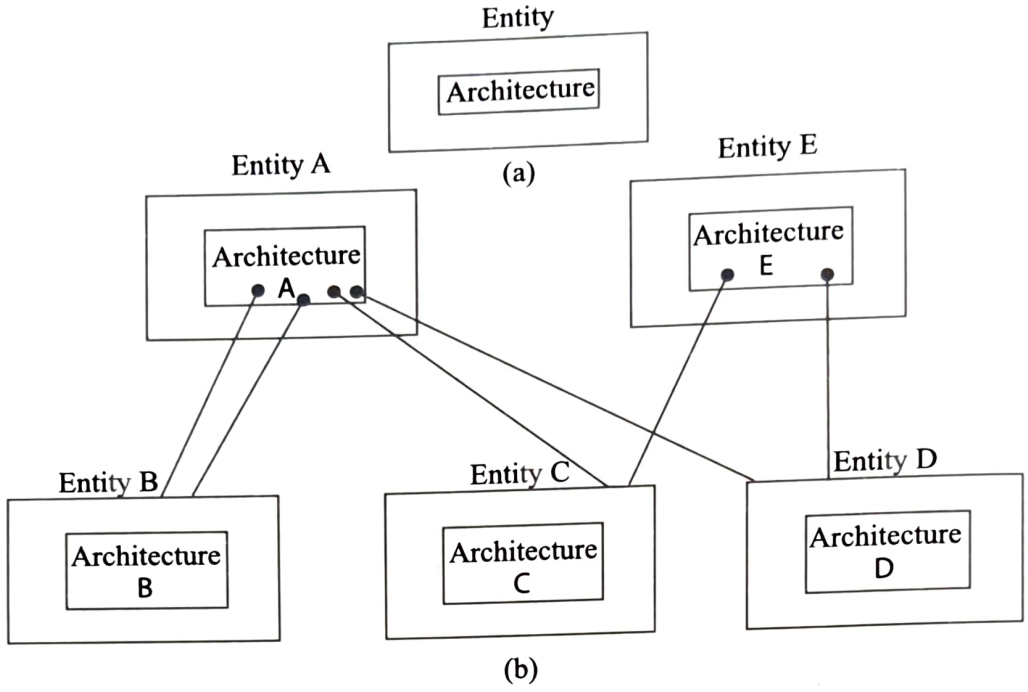


Fig. 14.10 (a) Basic Concept of VHDL (b) Use of Lower-Level Entities by Higher-Level Architectures

Every sub-system or component can be described by its VHDL, i.e., each sub-system or component h its own entity declaration and architecture. The architecture of the digital system or circuit can ma use of the entities of its sub-systems or components. This makes hierarchical system design possib Figure 14.10(b) shows a top-level architecture *A* making use of entities *B*, *C*, and *D*. The entity *B* is us two times. In general, a top-level architecture can make multi use of lower-level entity. An entity c also be used by any other architectures. In Fig. 14.10(b), the entities *C* and *D* are used by some oth architecture *E* also. The architectural details of the lower-level entities are not visible to the high level architectures.

A VHDL source code file has two main sections: an entity and an architecture. Similar to any oth programming language, VHDL has some rules and characteristics. Some of these are given below:

- VHDL is strongly-typed language. It has some predefined types. Every signal, variable, and consta in a VHDL program must match with the allowed type in the program. The type specifies the or range of values that the object can take on. There is also a set of operators (boolean and integ such as **AND**, **OR**, **NAND**, **NOR**, **XOR**, **XNOR**, **NOT**, addition, subtraction, multiplication, division, et
- There are a number of reserved words (or keywords) which have specific meanings and these can n be used for identifiers or as any other name. These are given in Appendix-B1.
- VHDL is a free-form language. Carriage returns, blank lines, and additional blank spaces may included between words for clarity without any ill effects.
- Long statements can continue over more than one line.
- A line starting with two adjacent hyphens (– –) is treated as comment.
- Concluding semicolon (;) is syntactically required. Its absence will cause error message during t compilation of the code.
- TIME is a predefined type, any of the following time units are available in VHDL:

fs – femtosecond
ps – picosecond
μs – microsecond
ms – millisecond
s – second
min – minute
hr – hour

- ns is built in VHDL. It need not be specified.
- VHDL allows arrays to be indexed in either direction (ascending or descending) because both conventions are prevalent in hardware.
- VHDL is a data flow language, unlike procedural computing languages such as *C* and assembly code, which run sequentially (one instruction at a time).
- A basic identifier in VHDL is composed of a sequence of one or more alphanumeric characters and underscore (_) character. The first character must be an alphabet, and the last character must not be an underscore. Two consecutive underscores are not allowed.
- VHDL is case insensitive, i.e., upper and lower case letters are considered identical.
- Boolean operators **AND**, **OR**, **NOT**, **NAND**, **NOR**, **XOR**, and **XNOR** are built in VHDL.
- A *system library* IEEE library is provided which stores packages for standard logic gates and other commonly used functions. A *user library* can be created by the user.
- Special notation symbols and syntax provided in VHDL are given in Appendix-B2.

14.5.1 Entity

The term *design entity* or just *entity* in VHDL refers to any digital device that possesses some form of intercommunication characteristic. It is a hardware abstraction of an actual digital hardware device. The device to be modelled (entity) may be a single gate, a central processsing unit (CPU), board containing discrete components, or even a complete digital system.

An entity may be decomposed into its constituent lower level entity, or subcomponents or alternatively it may be treated as a building block to construct a high level entity. This means an entity X can be used as a component of another entity Z, or an entity W can be decomposed into its lower level entities P, Q, R, . . ., etc. For example, an **EX-OR** gate shown in Fig. 14.11(a) is an entity which can be decomposed into its lower level enitites **AND**, **OR**, and **NOT** gates as shown in Fig. 14.11(b), or conversely an **EX-OR** entity can be considered as composed of **AND**, **OR**, and **NOT** entities.

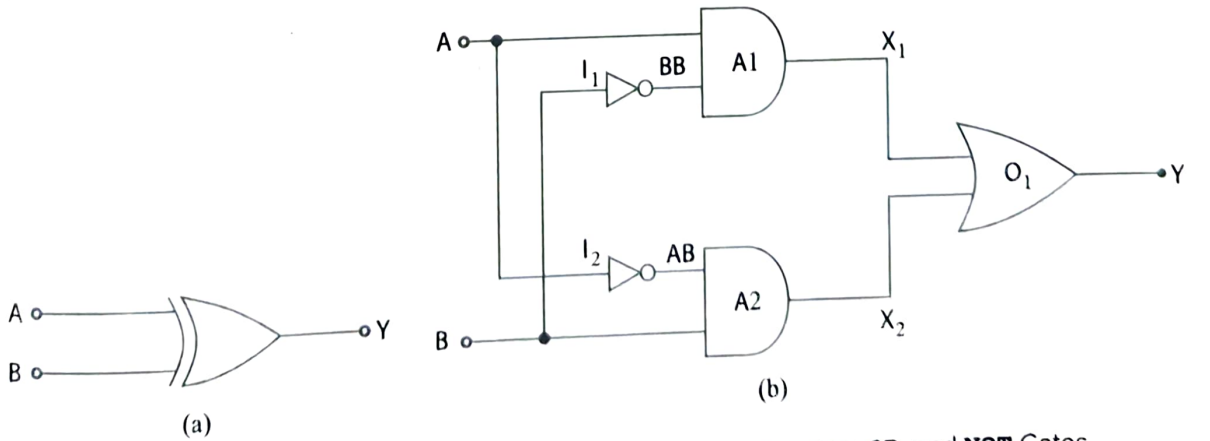


Fig. 14.11 (a) An **EX-OR** Gate (b) An **EX-OR** Function using **AND**, **OR**, and **NOT** Gates

In VHDL, all designs are created using entities. A design must have atleast one entity. An entity is the most basic building block in a digital design and since an entity can be decomposed into its lower level entities or can be treated as a building block for higher level entity, therefore, VHDL can support both top-down and bottom-up design methodologies. This methodology is referred to as *hierarchical design* and it is very useful in dealing with the complex circuits.

The term **ENTITY** is a key-word in VHDL and is a construct used in VHDL code. The keyword **ENTITY** signifies the start of an entity statement. Each entity is uniquely declared describing its external communication links to the outside world. It specifies the number of ports, the directions of the ports, and the type of ports. Some more information such as timing can be put into the entity.

An entity is assigned a name and the corresponding construct declares its input and output signals, known as *ports*. A port is indentified by the keyword **PORT**. Each port has an associated *mode* that specifies whether it is an input port (keyword **IN**), an output port (keyword **OUT**) or a bidirectional port (keyword **INOUT**) of the entity. Since each port represents a signal, hence, it has an associated *type*.

The syntax of an entity declaration is:

ENTITY *entity-name* **IS**
PORT (List of input, output port names and their types);
END *entity-name*;

The words: **ENTITY**, **IS**, **PORT**, **IN**, **OUT**, and **END** used above are the *reserved* words or *keywords* in VHDL. These words have specific meanings for the VHDL compiler, and they can not be used for any other purpose.

An *entity-name* is composed of a string of one or more alphanumeric characters and the underscore (_) character. The first character of a name must be an alphabet (upper-or-lower case) and the last character must not be an underscore. Also a name can not have two successive underscores and it must not be a VHDL reserved word. VHDL is not case sensitive, which means upper and lower case letters can be freely used, for example, SET_CK_HIGH, Select_Signal, ROM_address are all valid names. Entry name should preferrably be assigned as a word meaningfully related to the function performed by the entity so as to make it convenient for the designer for indentification.

The list of input, output port names and their types describes the input and output signals of the entity and their types. For example, for the circuit of Fig. 14.11(a), it will appear as

```
PORT   (A, B : IN BIT;
          Y : OUT BIT);
```

A and *B* are the input ports (or signals), and *Y* is an output port (or signal). The signal names are user-selected identifiers and are separated by comma. The input and output signals are of the type BIT which can assume only one of the two binary digits 0 or 1.

The direction of the port can be input (**IN**), output (**OUT**), or bidirectional (**INOUT**) and is referred to as *mode*.

The reserved word **END** signifies the end of the **ENTITY** declaration. The symbols colon(:) and semicolon(;) are used as separator and terminator, respectively.

From the entry declaration illustrated above, it is clear that an entity declaration describes only the input and output signals of the entity. It does not give any details about the behaviour of the entity or the way the circuit components are connected, i.e., the structure of the circuit. In fact the entity declaration does not and cannot model a device's behaviour. It is merely an outer shell without any functional core.

Example 14.21

Write entity declaration constructs in VHDL for the **AND**, **OR**, **NOT**, and **EX-OR** gates shown in Fig. 14.12.

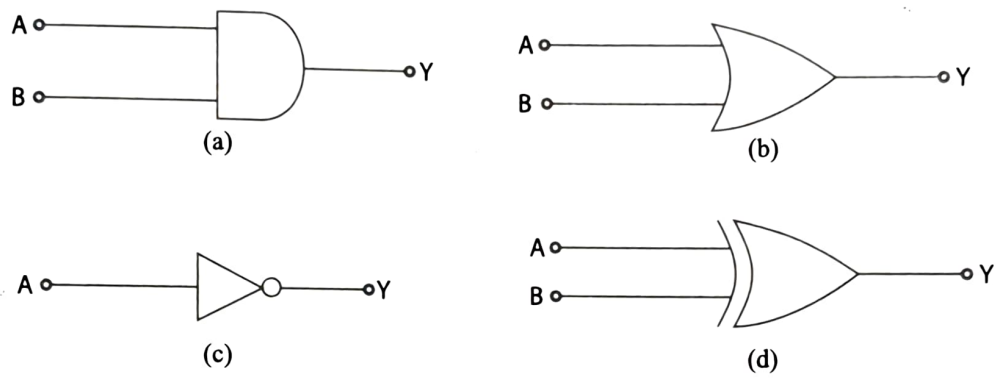


Fig. 14.12 (a) **AND** (b) **OR** (c) **NOT** (d) **EX-OR** Gates

Solution

(a) For **AND** gate

```
ENTITY and2 IS
PORT (A, B: IN BIT;
Y: OUT BIT);
END and2;
```

Here, the name of the entity is chosen as *and2* (a 2-input **AND** gate), A and B are the input ports, and Y is the output port. The type of signal is BIT.

(b) For **OR** gate

```
ENTITY or2 IS
PORT (A, B: IN BIT;
Y: OUT BIT);
END or2;
```

(c) For **NOT** gate

```
ENTITY not IS
PORT (A: IN BIT; Y: OUT BIT);
END not;
```

(d) For **EX-OR** gate

```
ENTITY xor2 IS
PORT (A, B: IN BIT; Y: OUT BIT);
END xor2;
```

Example 14.22

Write entity construct for the **EX-OR** circuit of Fig. 14.11(b).

Solution

Let the name of the entity be Circuit_Fig. It has two input ports A and B and one output port Y. The entity declaration for this circuit will be

ENTITY *Circuit_Fig* **IS**
PORT (*A*, *B*: **IN BIT**; *Y*: **OUT BIT**);
END *Circuit_Fig*;

From this entity declaration, we observe that although this circuit consists of **AND**, **OR**, and **NOT** gates, the circuit itself is an entity and the entity declaration gives no information about the structure or behaviour of the circuit.

Example 14.23

Write entity construct for the R-S **FLIP-FLOP** circuit shown in Fig. 14.13.

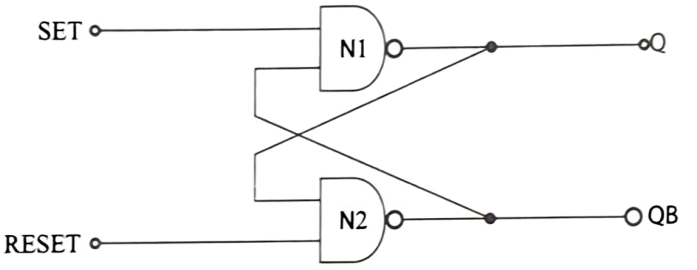


Fig. 14.13 R-S **FLIP-FLOP**

Solution

Let the name of the entity be rsff. It has two input ports SET and RESET and two bidirectional ports *Q* and *QB*. The entity construct will be

ENTITY *rsff* **IS**
PORT (SET, RESET: **IN BIT**;
Q, *QB*: **INOUT BIT**);
END *rsff*;

14.5.2 Architecture

A VHDL design entity is modelled using an entity declaration and its atleast one architecture body. The entity declaration describes the external view of the entity, whereas the architecture body contains the internal description of the entity. The internal description can be specified in the following ways:

- A set of interconnected components that represent the structure of the entity.
- A set of concurrent statements that represent the behaviour of the entity.
- A set of sequential statements that represent the behaviour of the entity.

Each of the above methods of representation can be specified in a different architecture body or mixed within a single architecture body. An architecture body is composed of two parts: *declarative part* and *statement part*. The syntax of architecture body is:

ARCHITECTURE *architecture-name* **OF** *entity-name* **IS**
[declarative part]
BEGIN

[statement part]
END *architecture-name*

An architecture must be given a name consisting of a text string which should be assigned by a designer in a way meaningful to the design.

The words **ARCHITECTURE**, **OF**, **BEGIN** are keywords in VHDL.

The declarative part appears before the keyword **BEGIN**. It can be used to declare signals, user-defined types, constants, components, function, and procedure definitions. The signals and other declarations in an architecture are local to that architecture only. Declarations common to multiple entities can be made in a separate 'package' used by all entities. The 'package' will be discussed later.

The statement part of the architecture is contained between the keywords **BEGIN** and **END**. All the statements are *concurrent* statements, which means, these are executed concurrently (simultaneously) and not sequentially as in the case of any programming language.

Structural Modelling

In structural modelling, an entity is described as a set of interconnected components in the architecture body. Let us consider the **EX-OR** gate shown in Fig. 14.11(b) and its entity declaration given in part (d) of Example 14.21. This circuit has three types of components, two 2-input **AND** gates, one 2-input **OR** gate, and two **NOT** gates. Its architecture body is given below:

```
ARCHITECTURE XOR_STRUCTURE OF xor2 IS
COMPONENT and2                          - - Entity and2 has X, Y inputs
PORT (X, Y: IN BIT; Z: OUT BIT);        - - and Z output.
END COMPONENT;
COMPONENT or2                           - - Entity or2 has P, Q inputs
PORT (P, Q: IN BIT; R: OUT BIT);        - - and R output.
END COMPONENT;
COMPONENT not                           - - Entity not has I input
PORT (I: IN BIT; J: OUT BIT);           - - and J output.
END COMPONENT;
SIGNAL AB, BB, X1, X2: BIT;
BEGIN
I1: not PORT MAP (B, BB);
I2: not PORT MAP (A, AB);
A1: and2 PORT MAP (A, BB, X1);
A2: and2 PORT MAP (AB, B, X2);
O1: or2 PORT MAP (X1, X2, Y);
END XOR_STRUCTURE;
```

In this, the architecture body has been named as *XOR_STRUCTURE* and it is associated with the entity declaration with the name *xor2* and therefore it inherits the list of interface ports from that

entity declaration. For each type of component, *component declaration* is required. The component declarations are shown in the declarative part of the architecture body. The architecture body also contains a signal declaration that declares four signals *AB*, *BB*, X_1, and X_2, of type **BIT**. These signals which represent wires, are used to connect the various components that form the **EX-OR** circuit. The scope of these signals is restricted to the architecture body and are not visible outside the architecture body. These signals are referred to as *local signals* or *buried nodes*, and are neither inputs nor outputs of the entity of the architecture. The component declaration lists its name, mode and type of each of its ports.

For each component, *component instantiation* statement is required in the statement part. This requires port map, i.e., input and output ports to be specified. A component instantiation statement is a concurrent statement, therefore, these statements can appear in any order. The structural modelling describes only interconnection of components, without specifying behaviour of the components or the entity they collectively represent.

The component instantiation statements include a label, such as *I*1, *I*2, *A*1, *A*2 . . ., etc.; component name, such as and2, or2, not . . ., etc.; and association between the actual signals that are visible in the architecture body and the ports of the component being instantiated. The mapping of ports specifies the association between the ports of a component with the ports in the architecture body, which may consist of external input, output ports and internal ports declared through signal statement. For example, *I*1 is a **NOT** gate with port *B* as input and *BB* signal as output, similarly *I*2, *A*1, *A*2, and *O*1 are instantiated. The signals in the **PORT MAP** list are in the same order in which they appear in entity definition of the component.

Example 14.24

Write architecture body of R-S **FLIP-FLOP** shown in Fig. 14.13 using structural modelling. Assume entity *nand*2 with *A*, *B* inputs and *Y* output.

Solution

It consists of two 2-input **NAND** gates. Let us use the name of the architecture also rsff which is same as the entity name. It is allowed for the architecture body to have same name as entity. The architecture body will be

ARCHITECTURE rsff **OF** *rsff* **IS**
COMPONENT *nand2*
PORT (*A*, *B*: **IN BIT**; *Y*: **OUT BIT**);
END COMPONENT;
BEGIN
N1: *nand2*
PORT MAP (SET, *QB*, *Q*);
N2: *nand2*
PORT MAP (RESET, *Q*, *QB*);
END *rsff*;

In this example, there are no signals to be declared.

Data Flow Modelling

In this modelling, the flow of data through the entity is expressed using concurrent signal assignment statements. As the name implies, the statements contained in the model assign values to signals. These statements execute concurrently, i.e., in parallel, not serially, as in the case of a programming language. The structure of the entity is not explicitly specified in this modelling, but it can be implicitly deduced.

A signal assignment statement is of the form:

$$A <= B; \tag{14.1}$$

It means A gets the value of B, i.e., the current value of signal B is assigned to signal A. This statement is executed whenever signal B changes value. Signal B is in the sensitivity list of this statement. A signal assignment statement is executed whenever a signal in its sensitivity list changes value. A transaction is generated when a signal assignment statement is executed. The target signal may or may not change following the execution of the signal assignment statement. If a new value of target signal is generated, then an event is scheduled for the target signal.

Time delay can also be introduced in the signal assignment statements as given below:

$$A <= B \textbf{ AFTER } 10 \text{ ns}; \tag{14.2}$$

Here, signal B is in the sensitivity list of this signal assignment and according to this statement signal A gets the value of signal B after a lapse of 10 ns. **AFTER** is a VHDL keyword. The data flow modelling uses simple assignment statements involving logic or arithmetic expressions.

Example 14.25

Write the architecture body of each of the entities of Example 14.21 using concurrent signal assignments, i.e., the data flow model. Assume 10 ns delay.

Solution

(a) For **AND** gate

```
ARCHITECTURE df_ and2 OF and2 IS
BEGIN
Y <= A AND B AFTER 10ns;
END df_and2;
```

(b) For **OR** gate

```
ARCHITECTURE df_or2 OF or2 IS
BEGIN
Y <= A OR B  AFTER  10ns;
END df_or2;
```

(c) For **NOT** gate

```
ARCHITECTURE INV OF not IS
BEGIN
Y <= NOT A AFTER 10ns;
END INV;
```

(d) For **EX-OR** gate

```
ARCHITECTURE  df_xor OF xor2 IS
BEGIN
Y <= (A AND (NOT B)) OR (B AND (NOT A));
END df_xor2;
```

Since VHDL does not assume any precedence of logic operators, therefore, parentheses must be used in the expression to avoid compile-time error for the expression.

Example 14.26

Write architecture body of *R-S* **FLIP-FLOP** shown in Fig. 14.13 using data flow model. Assume 5 ns delay time.

Solution

```
ARCHITECTURE df_rsff OF rsff IS
BEGIN
Q <= NOT (QB AND SET) AFTER 5 ns;
QB <= NOT (Q AND RESET) AFTER 5 ns;
END df_rsff;
```

In Examples 14.25 and 14.26, **AND**, **OR**, **NOT**, and **XOR** have been used which are in-built boolean operators in VHDL.

There are two other types of concurrent signal assignment statements. These are:

- Conditional signal assignment, and
- Selected signal assignment.

Conditional Signal Assignment

A conditional signal assignment allows a signal to be one of several values based on certain conditions to be satisfied. It uses keywords, **WHEN** and **ELSE**, boolean operators such as **AND**, **OR**, **NOT**, and **XOR**, and relational operators = (equality), /= (inequality), > (greater than), >= (greater than or equal to), and < = (less than or equal).

Example 14.27

For a 2 : 1 multiplexer shown in Fig. 14.14, write entity declaration and conditional signal assignment data flow architecture body.

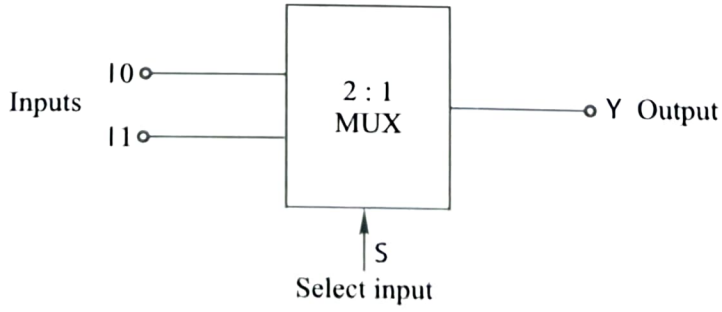


Fig. 14.14 A 2 : 1 Multiplexer

Solution

We shall make use of IEEE library which includes STD_LOGIC type. STD_LOGIC type is a standard data type for representation of logic signals in VHDL.

```
LIBRARY ieee;
USE ieee. std_logic_1164.all;
ENTTY MUX_FIG 13_7 IS
   PORT   (I0, I1, S: IN STD_LOGIC;
                Y : OUT STD_LOGIC);
END MUX_FIG 13_7;
ARCHIRECTURE DFA OF MUX_FIG 13_7 IS
BEGIN
   Y <= I0 WHEN S = '0' ELSE I1;
END DFA;
```

The conditional signal assignment specifies that Y is assigned the value of $I0$ where $S = 0$, or else Y is assigned the value of $I1$.

Selected Signal Assignment

A selected signal assignment allows a signal to be assigned one of several values, based on a selection criterion. The selected signal assignment begins with the keyword **WITH**, specifies the selection criterion and then **SELECT** keyword.

Example 14.28

Repeat Example 14.27 with selected signal assignment data flow architecture body.

Solution

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY MUX_FIG 13_7 IS
   PORT    (I0, I1, S: IN STD_LOGIC;
                Y : OUT STD_LOGIC);
END MUX_FIG 13_7;
ARCHITECTURE DFA OF MUX_FIG 13_7 IS
BEGIN
  WITH S SELECT
  Y <=  I0 WHEN '0',
        I1 WHEN OTHERS;
END DFA;
```

Here, the Keyword **OTHERS** is used for taking into consideration all the other possible values of other than 0. In STD_LOGIC data type, S can take on values 0, 1, Z (high-impedance state), and —(don't care).

Behavioural Modelling

In this type of modelling, the behaviour of an entity is expressed using statements which are execu sequentially similar to that of a high-level programming language. A process statement is the m mechanism used to model the behaviour of an entity. The functionality of an entity is described in algorithmic representation in the process statement.

The process statement starts with keyword **PROCESS** and ends with the keyword **END PROCESS**. the statements between the keywords **PROCESS** and **END PROCESS** are considered part of the pro

statement. A VHDL **PROCESS** statement can be used anywhere alongwith concurrent statements. Its execution is in parallel with other concurrent statements and other processes.

Statements in a **PROCESS** are sequentially, evaluated. Any assignments made to the signals inside the process are not visible outside the process until all of the statements in the process have been evaluated. In case there are multiple assignments to the same signal, only the last one will be visible outside. Neither conditional nor selected signal assignments are allowed within a process.

The process statement consists of three parts: senitivity list, declarative part, and statement part.

Sensitivity List

A process statement is always active and executes at all times if not suspended. Following **PROCESS** keyword, the sensitivity list (signals) in parentheses is specified. The sensitivity list includes all input signals that are used inside the **PROCESS**. For example, for the 2 : 1 multiplexer of Fig. 14.14, the sensitivity list will have *I*0, *I*1, and *S* signals. The process is activated when an event occurs on any one of these signals. When the program flow reaches the last sequential statement, the process becomes suspended until another event occurs on a signal that it is sensitive to. Regardless of the events on the sensitivity list signals, processes are executed once at the beginning of the simulation run.

Declarative Part

The declarative part is used to declare local variables or constants that can be used only inside the process. i.e., such objects are visible only to the process within which they are declared. Signals and constants declared in the declarative part of an architecture that encloses a process statement can be used inside a process. Such signals are the only means of communication between different processes.

Initialisation of objects declared in a process is done only once at the beginning of a simulation run. The initialisations in a subprogram are performed each time the subprogram is called.

The process declarative part consists of the area between the sensitivity list and the keyword **BEGIN**.

Statement Part

The statement part of the process consists of the area between the keywords **BEGIN** and **END PROCESS**. All the statements are sequential and are executed one after the other in a sequential order.

The statement part of a process is always active. When the program flow reaches the last sequential statement of this part, the execution returns to the first statement in the statement part and continues.

Example 14.29

Write architecture body of *R-S* **FLIP-FLOP** shown in Fig. 14.13 using behavioural model. Assume 10 ns delay time.

Solution

ARCHITECTURE *behave_rsff* **OF** *rsff* **IS**
BEGIN
PROCESS (SET, RESET)
BEGIN
IF SET = '1' **AND** RESET = '0' **THEN**
Q <= '0' **AFTER** 10 ns;

```
QB <= '1' AFTER 10 ns;
ELSIF SET = '0' AND RESET = '1' THEN
Q <= '1' AFTER 10 ns;
QB <= '0' AFTER 10 ns;
ELSIF SET = '0' AND RESET = '0' THEN
Q <= '1' AFTER 10 ns;
QB <= '1' AFTER 10 ns;
END IF;
END PROCESS;
END behave_rsff;
```

Let us examine the execution of this architecture when SET changes to a '0' and RESET remain '1'. Since, SET is in the sensitivity list of the process statement, the process is invoked and each state in the process is executed sequentially. The first statement is an IF statement. This statement yi negative result because SET = '0' and RESET = '1', therefore, the following two signal assignmen not executed. Then the next check is performed which succeeds and therefore, the next two assignments are executed.