MODULE 3: Hardware Software Co design and Program Modelling: Fundamental issues in Hardware Software Co-design, Computational models in Embedded System Design.
Embedded Hardware Design and Development: Analog Electronic Components, Digital Electronic Components, VLSI & Integrated Circuit Design, Electronic Design Automation Tools.

HARDWARE SOFTWARE CO-DESIGN AND PROGRAM MODELING

Hardware Software Co-Design

In the traditional embedded system development approach, the hardware software partitioning is done at an early stage and engineers from the software group take care of the software architecture development and implementation, whereas engineers from the hardware group are responsible for building the hardware required for the product. There is less interaction between the two teams and the development happens either serially or in parallel. Once the hardware and software are ready, the integration is performed.

Fundamental Issues in Hardware Software Co-Design

The fundamental issues in hardware software co-design are:

1. Selecting the Model
2. Selecting the Architecture
3. Selecting the Language
4. Partitioning System Requirements into Hardware and Software

Selecting the Model: In hardware software co-design, models are used for capturing and describing the system characteristics. A model is a formal system consisting of objects and composition rules. It is hard to make a decision on which model should be followed in a particular system design. Most often designers switch between varieties of models from the requirements specification to the implementation aspect of the system design.

For example, at the specification stage, only the functionality of the system is in focus and not the implementation information. When the design moves to the implementation aspect, the information about the system component is revealed and the designer has to switch to a model capable of capturing the system's structure.

Selecting the Architecture: A model only captures the system characteristics and does not provide information on 'how the system can be manufactured?'. The architecture specifies how a system is going to implement in terms of the number and types of different components and the interconnection among them. Commonly used architectures are:

- Controller architecture: Implements the finite state machine model (FSM) using a state register and two combinational circuits.
- Datapath architecture: Best suited for implementing the data flow graph model where the output is generated as a result of a set of predefined computations on input data.
- Complex Instruction Set Computing (CISC) architecture: Uses an instruction set representing complex operations and can perform a large complex operation with a single instruction.
- Reduced Instruction Set Computing (RISC) architecture: Reuses instruction set representing simple operations and it requires the execution of multiple RISC instructions to perform a complex operation.
- Very Long Instruction Word Computing (VLIW) architecture: Implements multiple functional units (ALUs, multipliers, etc.) in the datapath.
- Single Instruction Multiple Data (SIMD) architecture: A single instruction is executed in parallel with the help of the Processing Element. The SIMD architecture forms the basis of reconfigurable processor.
- Multiple Instruction Multiple Data (MIMD) architecture: Executes different instructions at a given point of time. The MIMD architecture forms the basis of multiprocessor systems.

Selecting the Language: A programming language captures a 'Computational Model' and maps it into architecture. There is no hard and fast rule to specify which language should be used for capturing this model. A model can be captured using multiple programming languages like C, C++, C#, Java, etc. for software implementations and languages like VHDL, System C, Verilog, etc. for hardware implementations. On the other hand, a single language can be used for capturing a variety of models.

Certain languages are good in capturing certain computational model. For example, C++ is a good candidate for capturing an object oriented model. The only pre-requisite in selecting a programming language for capturing a model is that the language should capture the model easily.

Partitioning System Requirements into Hardware and Software: It may be possible to implement the system requirements in either hardware or software (firmware). It is a tough decision making task to figure out which one to opt. Various hardware software trade-offs are used for making a decision on the hardware-software partitioning.

Computational Models in Embedded Design

The commonly used computational models in embedded system design are:

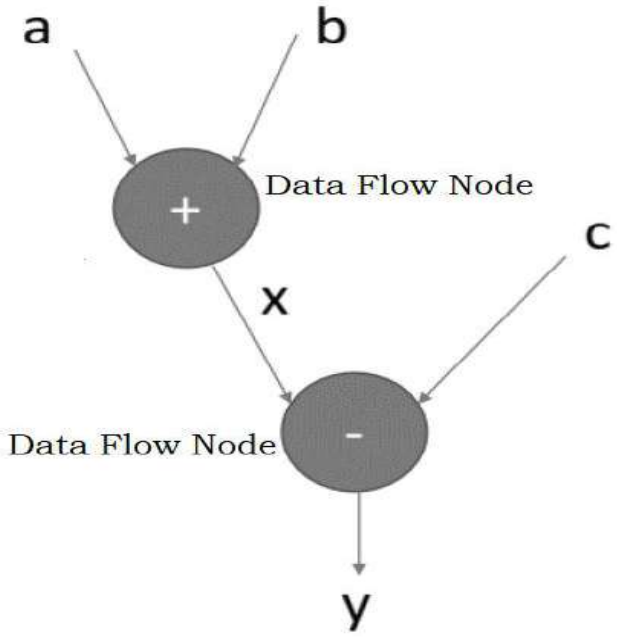
1. Data Flow Graph Model
2. Control Data Flow Graph Model
3. State Machine Model
4. Sequential Program Model
5. Concurrent/Communicating Process Model
6. Object-Oriented Model

Data Flow Graph Model

DFG model translates data processing requirements into data flow graph. It is a data driven model in which the program execution is determined by data. This model emphasizes on the data and operations on the data which transforms the input data to output data. Embedded applications which are computational intensive and data driven are modelled using the DFG model. DSP applications are typical examples for it.

DFG is a visual model in which the operation on the data (process) is represented using a block (circle) & data flow is represented using arrows. Inward arrow to the process (circle) represents input data & an outward arrow from the process (circle) represents output data in DFG notation.

Suppose one of the functions in an application contains the computational requirement $x = a + b$; and $y = x - c$. Figure illustrates the implementation.



A DFG model is said to be acyclic DFG (ADFG) if it doesn't contain multiple values for the input variable and multiple output values for a given set of input(s). Feedback inputs (Output is fed back to Input), events, etc. are examples for non-acyclic inputs. A DFG model translates the program as a single sequential process execution.

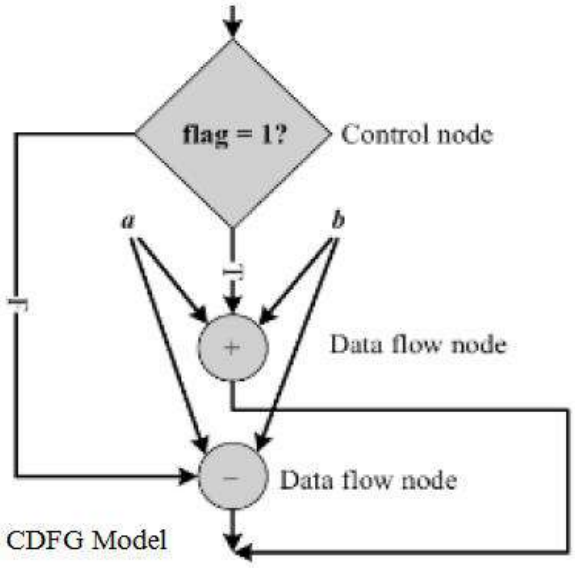
Control Data Flow Graph/Diagram (CDFG) Model

The DFG model is a data driven model in which the execution is controlled by data and it doesn't involve any control operations (conditionals). The Control DFG (CDFG) model is used for modelling applications involving conditional program execution. CDFG models contains both data operations and control operations. The CDFG uses Data Flow Graph (DFG) as element and conditional (constructs) as decision makers. CDFG contains both data flow nodes and decision nodes, whereas DFG contains only data flow nodes.

Consider the implementation of the CDFG for following requirement: *If flag* = 1, $x = a + b$; *else* $y = a - b$; The CDFG model for the same is given in the figure. The control node is represented by a 'Diamond' block which is the decision making element in a normal flow chart based design. Decision on which process is to be executed is determined by control node.

Real world example for modelling the embedded application using CDFG is capturing & saving of the image to a format set by the user in a digital camera.

The decision on, in which format the image is stored (formats like JPEG, TIFF, BMP, etc.) is controlled by the camera settings, configured by the user.



State Machine Model

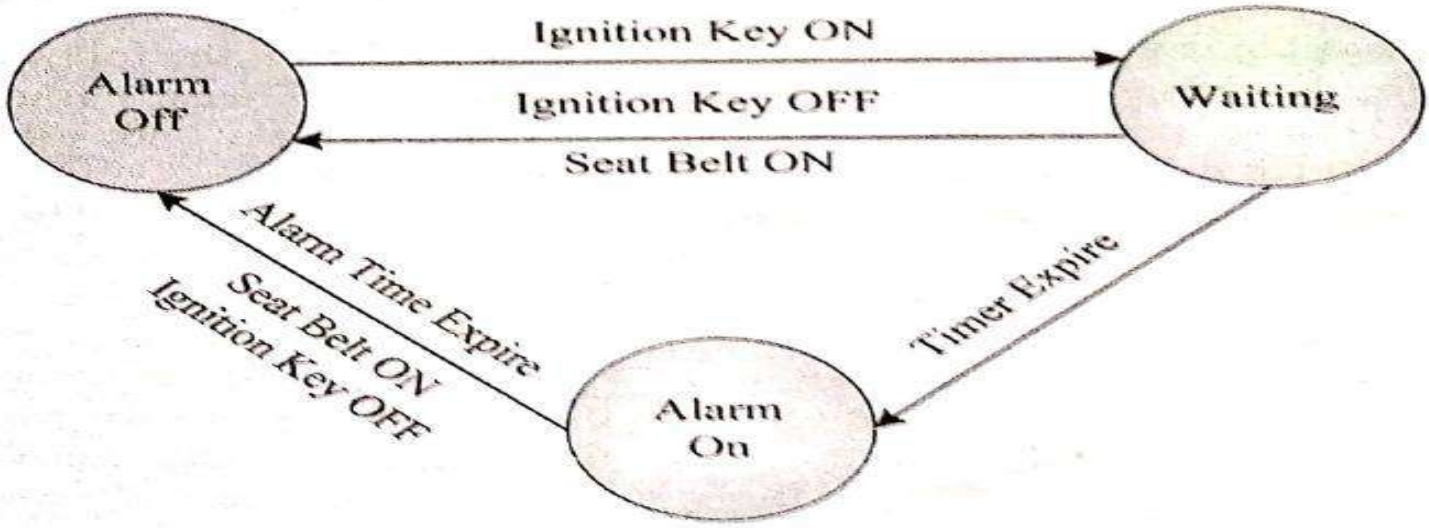
The State Machine model is used for modelling reactive or event-driven embedded systems whose processing behaviour is dependent on state transitions. Embedded systems used in the control and industrial applications are typical examples for event driven systems.

The State Machine model describes the system behaviour with 'States', 'Events', 'Actions' and 'Transitions'.

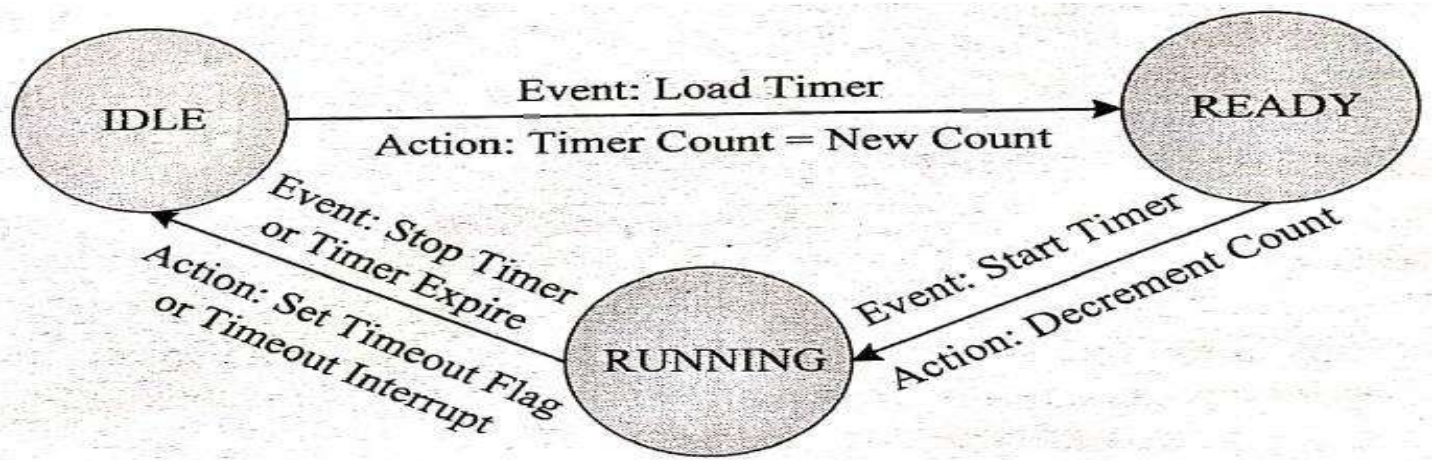
- State is a representation of a current situation.
- An event is an input to the state. The event acts as stimuli for state transition.
- Transition is the movement from one state to another.
- Action is an activity to be performed by the state machine.

Finite State Machine (FSM): A FSM model is one in which the number of states are finite. In other words the system is described using a finite number of possible states. For example, consider the design of an embedded system for driver/ passenger '*Seat Belt Warning*' in an automotive using the FSM model. The system requirements are captured as:

- When the vehicle ignition is turned on and the seat belt is not fastened within 10 seconds of ignition ON, the system generates an alarm signal for 5 seconds.
- The Alarm is turned off when the alarm time (5 seconds) expires or if the driver/ passenger fasten the belt or if the ignition switch is turned off, whichever happens first.
- Here the states are 'Alarm Off', 'Waiting' and 'Alarm On' and the events are 'Ignition Key ON', 'Ignition Key OFF', 'Timer Expire', 'Alarm Time Expire' and 'Seat Belt ON'.
- Using the FSM, the system requirements can be modelled as given in following Figure.



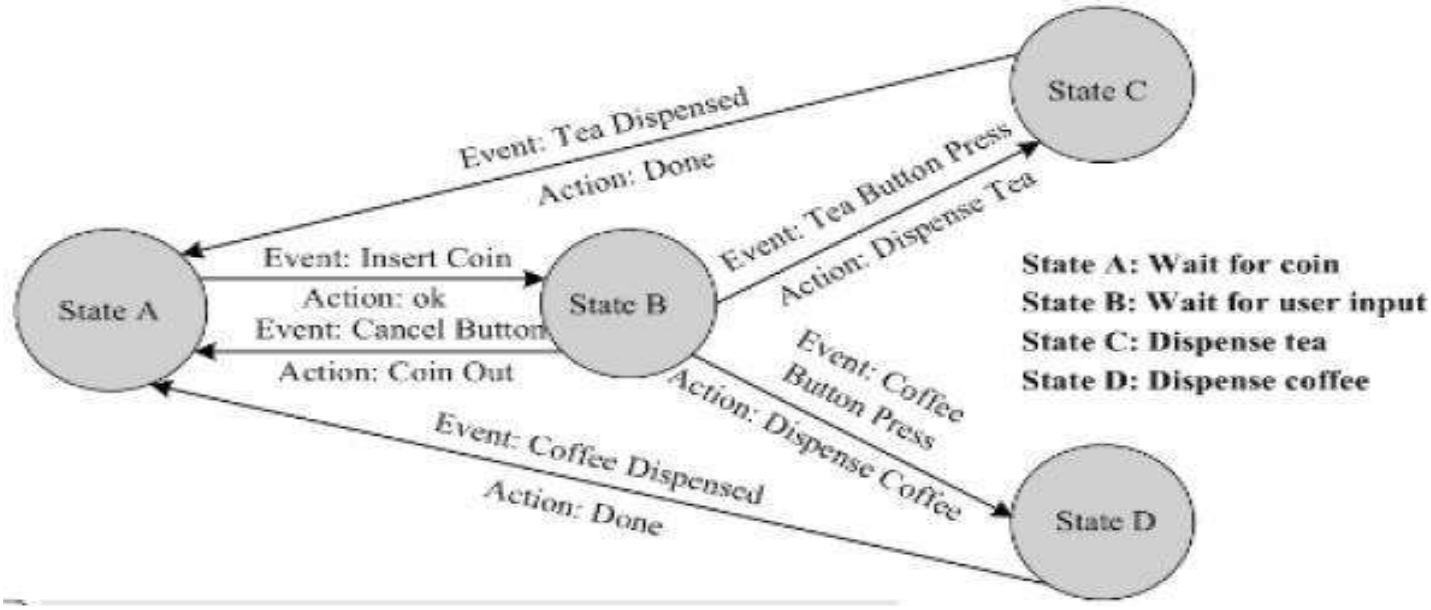
The wait state is implemented using a timer. The timer also has certain set of states and events for state transitions. Using the FSM model, the timer can be modelled as shown in the below figure.



Problem 1: Design an automatic tea/ coffee vending machine based on FSM model for the following requirement:

- The tea/coffee vending is initiated by user inserting a 5 rupee coin.
- After inserting the coin, the user can either select 'Coffee' or 'Tea' or press 'Cancel' to cancel the order and take back the coin.

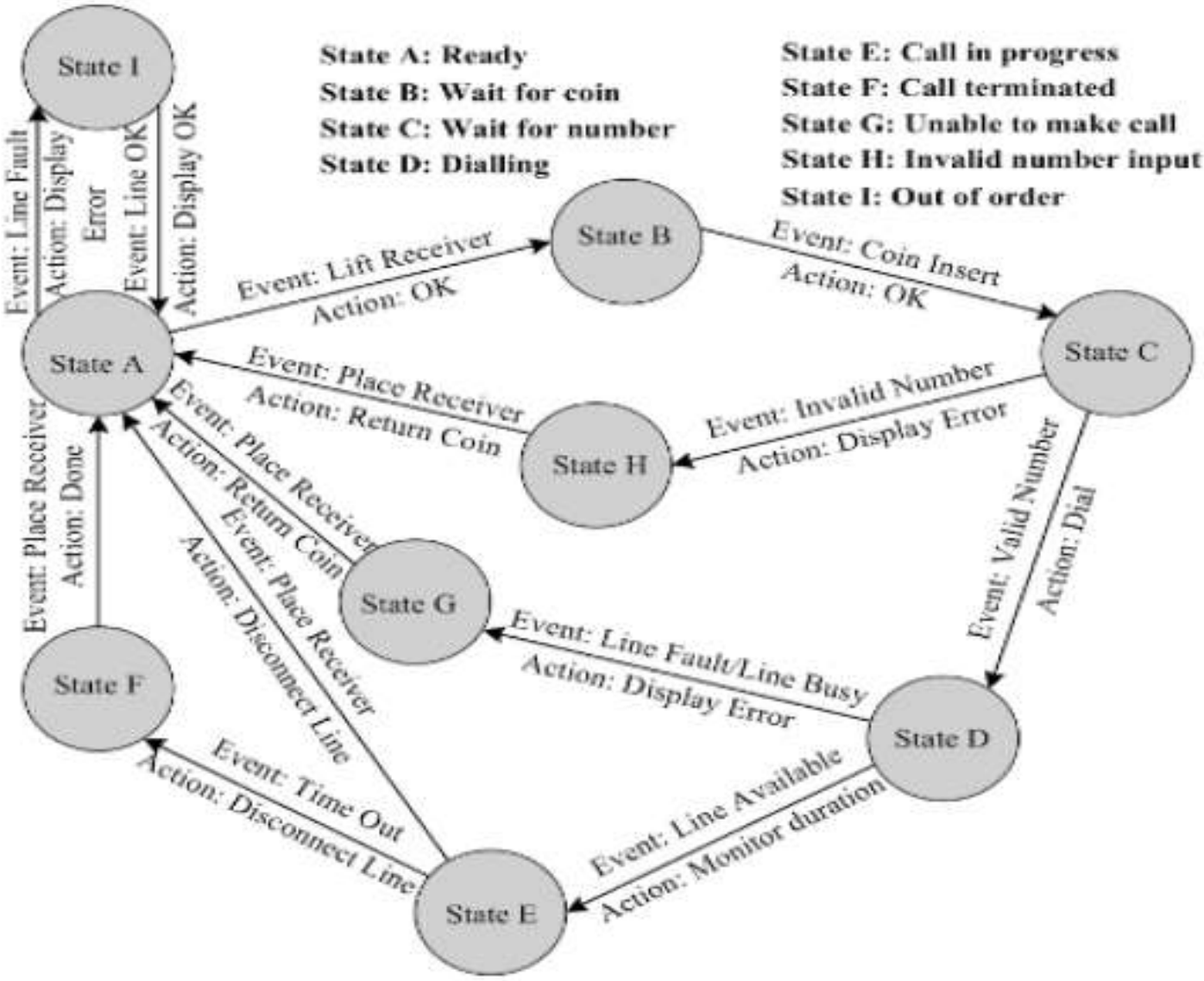
Solution: The FSM representation contains four states namely; 'Wait for coin' 'Wait for User Input', 'Dispense Tea' and 'Dispense Coffee'. The FSM representation for the above requirement is given in the below figure.



Problem 2: Design a coin operated public telephone unit based on FSM model for the following requirements.

- The calling process is initiated by lifting the receiver (off-hook) of the telephone unit.
- After lifting the phone the user needs to insert a 1 rupee coin to make the call.
- If the line is busy, the coin is returned on placing the receiver back on the hook (on-hook).
- If the line is through, the user is allowed to talk till 60 seconds and at the end of 45th second, prompt for inserting another 1 rupee coin for continuing the call is initiated.
- If the user doesn't insert another 1 rupee coin, the call is terminated on completing the 60 seconds time slot.
- The system is ready to accept new call request when the receiver is placed back on the hook (on-hook).
- The system goes to the 'Out of Order' state when there is a line fault.

Solution: The FSM representation for the above requirement is given in the below figure.



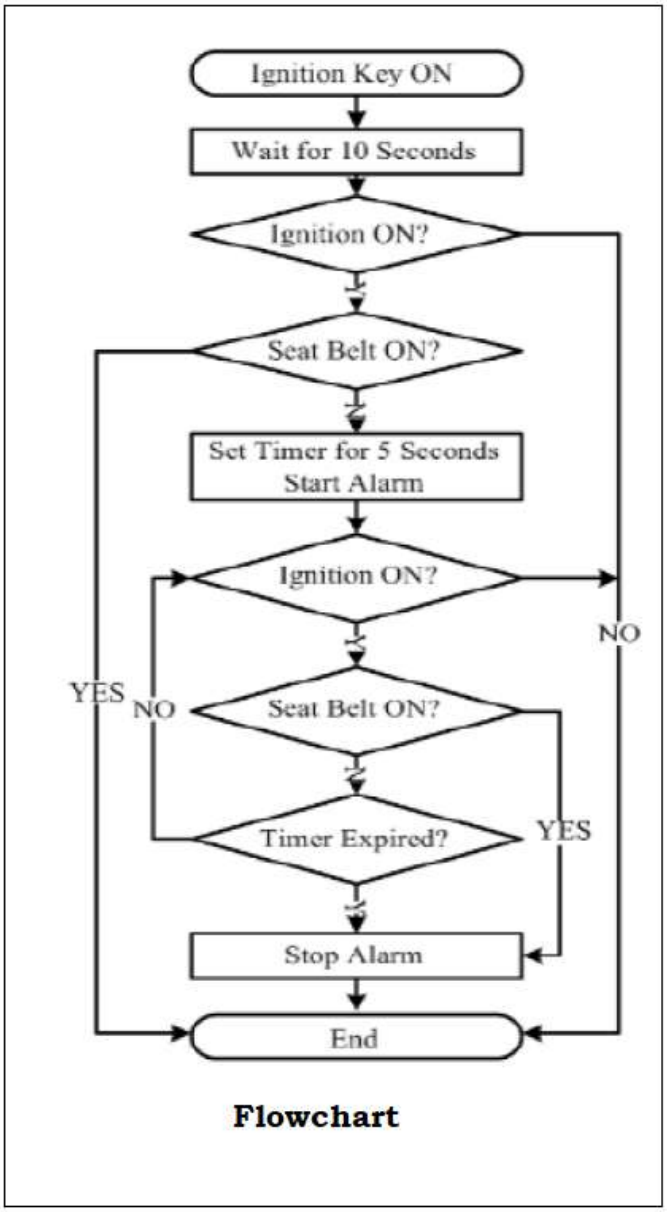
Sequential Program Model

In the Sequential Program Model, the functions or processing requirements are executed in sequence. It is same as the conventional procedural programming. Here the program instructions are iterated and executed conditionally and the data gets transformed through a series of operations. Finite State Machines (FSMs) and Flow Charts are used for modelling sequential program. The FSM approach represents the states, events, transitions and actions, whereas the Flow Chart models the execution flow.

The execution of functions in a sequential program model for the 'Seat Belt Warning' system is illustrated below and the Sequential Program Model (flowchart) for the same is also given in below figure:

```
#define ON 1
#define OFF 0
#define YES 1
#define NO 0
void seat_belt_warn ()
{   wait_10sec ();
    if (check_ignition_key () == ON)
    {
        if (check_seat_belt () == OFF)
        {
            set_timer (5);
            start_alarm ();
            while ((check_seat_belt ()
            == OFF) &&
            (check_ignition_key ()
             == OFF) &&
            (timer_expire () == ON));
            stop_alarm ();
        }
    }
}
```

Execution Flow



Flowchart

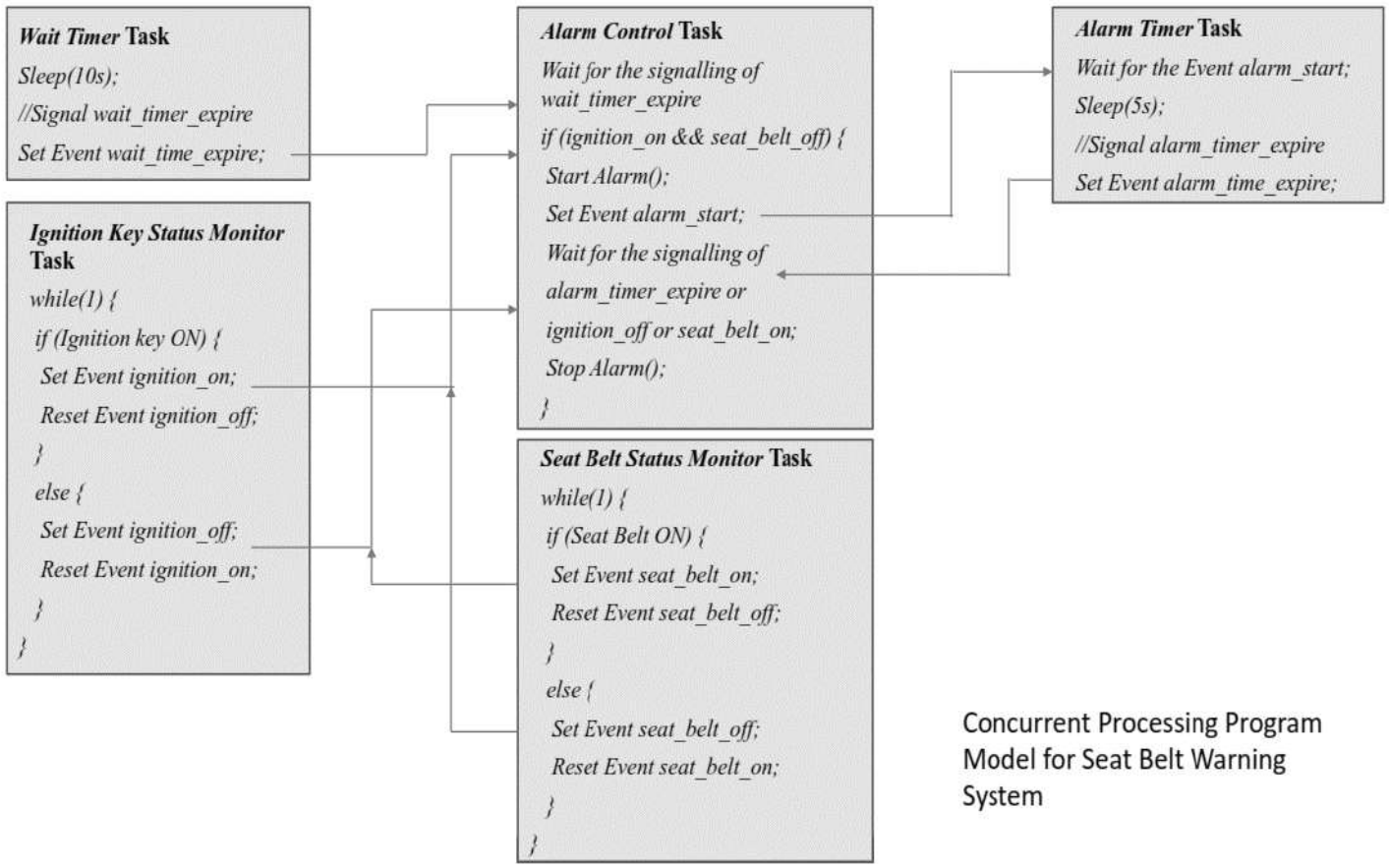
Concurrent/ Communicating Process Model

The concurrent or communicating process model models concurrently executing tasks/ processes. It is easier to implement certain requirements in concurrent processing model than the conventional sequential execution. Sequential execution leads to a single sequential execution of task and thereby leads to poor processor utilization, when the task involves I/O waiting, sleeping for specified duration etc. If the task is split into multiple subtasks, it is possible to tackle the CPU usage effectively, when the subtask under execution goes to a wait or sleep mode, by switching the task execution. However, concurrent processing model requires additional overheads in task scheduling, task synchronization and communication.

For example, consider the implementation of the 'Seat Belt Warning' system in concurrent processing model. We can split the tasks into:

1. Timer task for waiting 10 seconds (wait timer task)
2. Task for checking the ignition key status (ignition key status monitoring task)
3. Task for checking the seat belt status (seat belt status monitoring task)
4. Task for starting and stopping the alarm (alarm control task)
5. Alarm timer task for waiting 5 seconds (alarm timer task)

The tasks cannot be executed them randomly or sequentially. One way of implementing a concurrent model for the 'Seat Belt Warning' system is illustrated in the below figure:



Concurrent Processing Program Model for Seat Belt Warning System

Object-Oriented Model

The object-oriented model is an object based model for modelling system requirements. It disseminates a complex software requirement into simple well defined pieces called *objects*. Object-oriented model brings re-usability, maintainability and productivity in system design. In the object-oriented modelling, object is an entity used for representing or modelling a particular piece of the system. Each object is characterized by a set of unique behaviour and state.

A class is an abstract description of a set of objects and it can be considered as a '*blueprint*' of an object. A class represents the state of an object through member variables and object behaviour through member functions. The member variables and member functions of a class can be private, public or protected. Private member variables and functions are accessible only within the class, whereas public variables and functions are accessible within the class as well as outside the class. The protected variables and functions are protected from external access.

Analog Electronic Components

Resistors, capacitors, diodes, inductors, operational amplifiers (Opamp), transistors, etc. are the commonly used analog electronic components in embedded hardware design.

A resistor limits the current flowing through a circuit. Interfacing of LEDs, buzzer, etc. with the port pins of microcontroller through current limiting resistors is a typical example for the usage of resistors in embedded application. Capacitors and inductors are used in signal filtering and resonating circuits. Reset circuit implementation, matching circuits for RF designs, power supply decoupling, etc. are examples for the usage of capacitors in embedded hardware circuit. Electrolytic capacitors, ceramic capacitors, tantalum capacitors, etc. are the commonly used capacitors in embedded hardware design. Inductors are widely used for filtering the power supply from ripples and noise signals. Inductors with inductance value in the micro-henry (µH) range are commonly used in embedded applications for filter and matching circuit implementation.

P-N Junction diode, Schottky diode, Zener diode, etc. are the commonly used diodes in embedded hardware circuits. A Schottky diode is same as a P-N junction diode except that its forward voltage drop is very low when compared to ordinary P-N junction diode. A Zener diode acts as normal P-N junction diode when forward biased. It also permits current flow in the reverse direction, if the voltage is greater than the junction breakdown voltage. It is normally used for voltage clamping applications. Voltage rectification, freewheeling of current produced in inductive circuits, clamping of voltages to a desired level etc. are examples for the usage of diodes in embedded applications.

Transistors in embedded applications are used for either switching or amplification purpose. In switching application, the transistor is in either ON or OFF state. In amplification operation, the transistor is always in the ON state. The common emitter configuration of NPN transistor is widely used in switching and driving circuits in

embedded applications. Relay, buzzer and stepper motor driving circuits are examples for common emitter configuration based driver circuit implementation using transistor.

Digital Electronic Components

Digital electronics deal with digital or discrete signals. Microprocessors, Microcontrollers, and System on Chips (SoCs) work on digital principles. They interact with the rest of the world through digital I/O interfaces and process digital data. Embedded systems employ various digital electronic circuits for 'Glue logic' implementation. 'Glue logic' is the custom digital electronic circuitry required to achieve compatible interface between two different integrated circuit chips. Address decoders, latches, encoders/ decoders, etc. are examples for glue logic circuits. Transistor - Transistor Logic (TTL), Complementary Metal Oxide Semiconductor (CMOS) logic etc. are some of the standards describing the electrical characteristics of digital signals in a digital system.

Open Collector Configuration

Open collector is an I/O interface standard in digital system design. It facilitates the interfacing of IC output to other systems which operate at different voltage levels. In the open collector configuration, the output line from an IC circuit is connected to the base of an NPN transistor. The collector of the transistor is left unconnected and the emitter is internally connected to the ground signal of IC. Figure below illustrates an open collector output configuration.

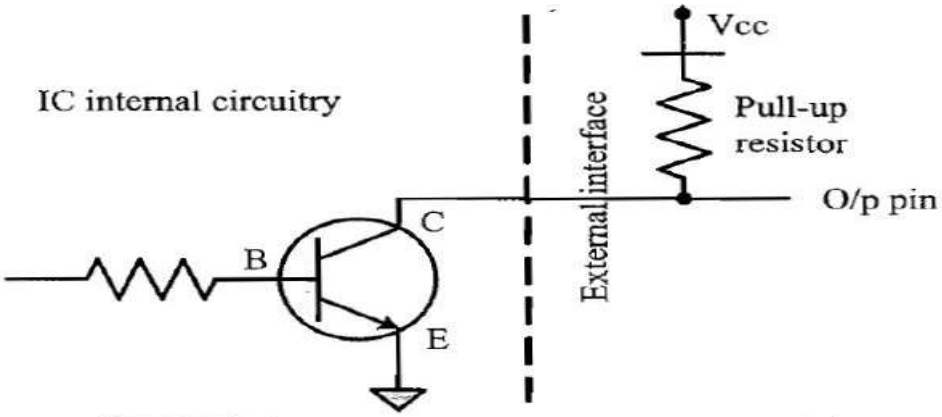


Fig. 8.1 **Open collector output configuration**

For the output pin to function properly, the output pin should be pulled, to the desired voltage for the output device, through a pull-up resistor. The output signal of the IC is fed to the base of an open collector transistor. When the base drive to the transistor is ON and the collector is in open state, the output pin floats. This state is known as 'high

impedance' state. If a pull-up resistor is connected to the output pin, when the base drive is ON, the output pin becomes at logic 0 (0V). With a pull-up resistor, if the base driver is 0, the output will be at logic high (Voltage = V).

The advantage of open collector output in embedded system design is listed below:

(1) It facilitates interfacing of devices, operating at a voltage different from IC, with the IC.

(2) An open collector configuration supports multi-drop connection.

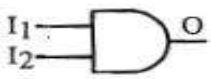
(3) It is easy to build Wired AND & Wired OR configuration using open collector output line.

Logic Gates

Logic gates are the building blocks of digital circuits. Logic gates control the flow of digital information by performing a logical operation of the input signals. Depending on the logical operation, the logic gates used in digital design are classified into-AND, OR, XOR, NOT, NAND, NOR and XNOR. The logical relationship between the output signal and the input signals for a logic gate is represented using a truth table. Figure 8.2 illustrates the truth table and symbolic representation of each logic gate.

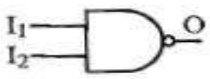
AND Gate –Truth Table

I_1	I_2	O
0	0	0
0	1	0
1	0	0
1	1	1



NAND Gate –Truth Table

I_1	I_2	O
0	0	1
0	1	1
1	0	1
1	1	0

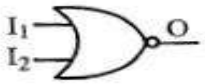


OR Gate –Truth Table

I_1	I_2	O
0	0	0
0	1	1
1	0	1
1	1	1

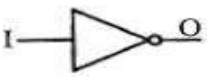
NOR Gate –Truth Table

I_1	I_2	O
0	0	1
0	1	0
1	0	0
1	-1	0



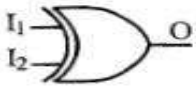
NOT Gate –Truth Table

I	O
0	1
1	0



XOR Gate –Truth Table

I_1	I_2	O
0	0	0
0	1	1
1	0	1
1	1	0



XNOR Gate –Truth Table

I_1	I_2	O
0	0	1
0	1	0
1	0	0
1	1	1

Fig. 8.2 Logic Gates Truth Table and Symbolic representation

Buffer

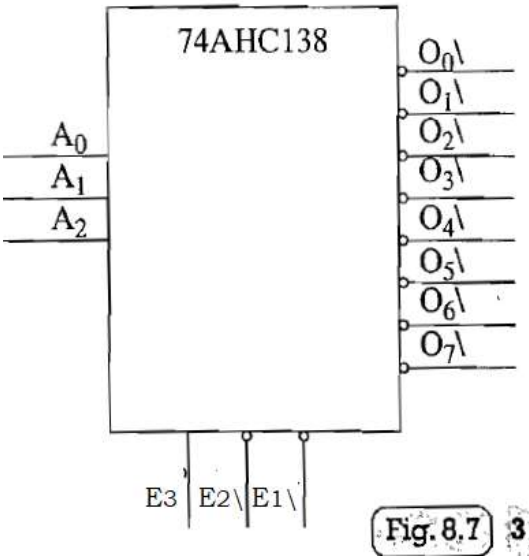
A buffer circuit is a logic circuit for amplifying the current or power. It increases the driving capability of a logic circuit. A tri-state buffer is a buffer with Output Enable control. When the Output Enable control is active, the tri-state buffer functions as a buffer. If the Output Enable is not active, the output of the buffer remains at high impedance state (Tri-stated). Tri- state buffers are commonly used as drivers for address bus and to select the required device among multiple devices connected to a shared data bus. 74LS244/74HC244 is an example of unidirectional octal buffer.

Latch

A latch is used for storing binary data. It contains an input data line, clock or gating control line for triggering the latching operation and an output line. Whenever a latch trigger happens, the data present on the input line is latched. The latched data is available on the output line of the latch until the next trigger. D flip flop is a typical example of a latch. In electronic circuits, latches are commonly used for latching data, which is available only for a short duration.

Decoder

A decoder is a logic circuit which generates all the possible combinations of the input signals. Decoders are named with their input line numbers and the possible combinations of the input as output. E.g.: 2 to 4 decoder, 3 to 8 decoder and 4 to 16 decoder. Decoders are mainly used for address decoding and chip select signal generation in electronic circuits and are available as integrated circuits. 74LS138/74AHC138 is an example for 3 to 8 decoder IC. Figure 8.7 illustrates the 74AHC138 decoder and the function table for it.



Input						Output							
A_2	A_1	A_0	E_1\	E_2\	E_3	O_0\	O_1\	O_2\	O_3\	O_4\	O_5\	O_6\	O_7\
0	0	0	0	0	1	0	1	1	1	1	1	1	1
0	0	1	0	0	1	1	0	1	1	1	1	1	1
0	1	0	0	0	1	1	1	0	1	1	1	1	1
0	1	1	0	0	1	1	1	1	0	1	1	1	1
1	0	0	0	0	1	1	1	1	1	0	1	1	1
1	0	1	0	0	1	1	1	1	1	1	0	1	1
1	1	0	0	0	1	1	1	1	1	1	1	0	1
1	1	1	0	0	1	1	1	1	1	1	1	1	0

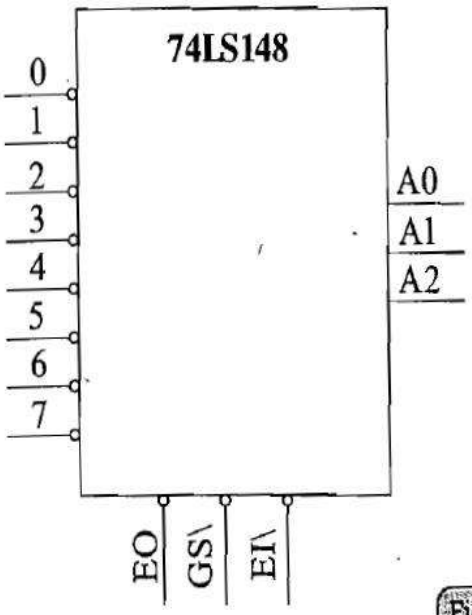
Fig. 8.7 3 to 8 Decoder IC and I/O signal states

The decoder output is enabled only when the "Output Enable' signal lines E1\, E2\ and E3 are at logic levels 0, 0 and 1 respectively. If the output enable signals are not at the required logic state, all the output lines are forced to the inactive (High) state. The output line corresponding to the input state is asserted "Low" when the 'Output Enable signal lines are at the required logic state.

Encoder

The encoder encodes the corresponding input state to a particular output format. The binary encoder encodes the input to the corresponding binary format. Encoders are named with their input line numbers and the encoder output format. E.g.: 4 to 2 encoder, 8 to 3 encoder and 16 to 4 encoder.

The 8 to 3 encoder contains 8 input signal lines and it is possible to generate a 3 bit binary output corresponding to the input. Encoders are mainly used for address decoding and chip select signal generation in electronic circuits and are available as integrated circuits. 74F148/74LS148 is an example of 8 to 3 encoder IC. Figure 8.8 illustrates the 74F148/74LS148 encoder and the function table for it.



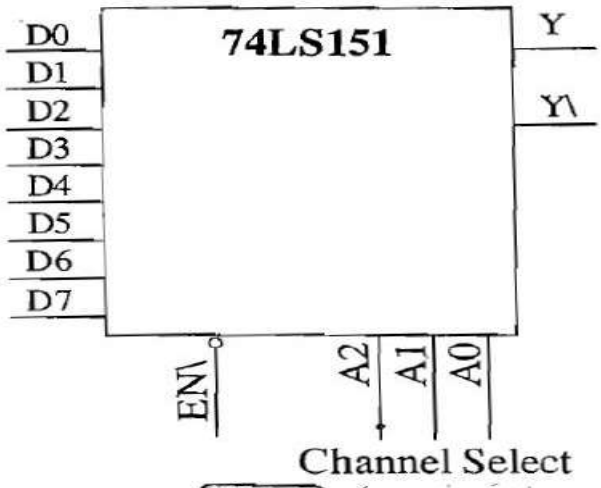
Input									Output				
7	6	5	4	3	2	1	0	E1	GS	E0	A2	A1	A0
1	1	1	1	1	1	1	0	0	0	1	0	0	0
1	1	1	1	1	1	0	1	0	0	1	0	0	1
1	1	1	1	1	0	1	1	0	0	1	0	1	0
1	1	1	1	0	1	1	1	0	0	1	0	1	1
1	1	1	0	1	1	1	1	0	0	1	1	0	0
1	1	0	1	1	1	1	1	0	0	1	1	0	1
1	0	1	1	1	1	1	1	0	0	1	1	1	0
0	1	1	1	1	1	1	1	0	0	1	1	1	1

Fig. 8.8 8 to 3 Encoder IC and I/O signal states

The encoder output is enabled only when the 'Enable Input (El)' signal line is at logic 0. The group signal (GS) is active-Low when any input is Low: this indicates when any input is active. The Enable Output (EO) is active-Low when all inputs are at logic 'High'. Encoding of key press in a keyboard is a typical example for an application requiring encoder. The encoder converts each key press to a binary code.

Multiplexer (MUX)

A multiplexer (MUX) can be considered as a digital switch which connects one input line from a set of input lines, to an output line at a given point of time. It contains multiple input lines and a single output line. The inputs of a MUX are said to be multiplexed. It is possible to connect one input with the output line at a time. The input line is selected through the MUX control lines. 745151 is an example for 8 to 1 multiplexer IC. Figure 8.9 illustrates the 745151 multiplexer and the function table for it.



Input				Output	
A2	A1	A0	EN\	Y	Y\
0	0	0	0	D0	D0\
0	0	1	0	D1	D1\
0	1	0	0	D2	D2\
0	1	1	0	D3	D3\
1	0	0	0	D4	D4\
1	0	1	0	D5	D5\
1	1	0	0	D6	D6\
1	1	1	0	D7	D7\
x	x	x	1	0	1

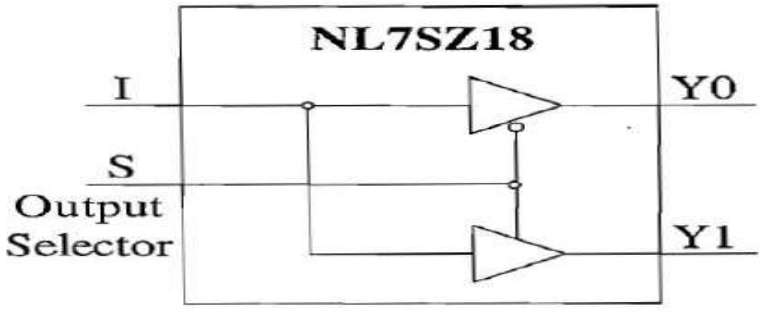
x: Don't care

Fig. 8.9 8 to 1 multiplexer IC and I/O signal states

The multiplexer is enabled only when the 'Enable signal (EN)' line is at logic 0. The input signal is switched to the output line through the channel select control lines A2, A1 and A0.

De-multiplexer (D-MUX)

A de-multiplexer performs the reverse operation of multiplexer. De-multiplexer switches the input signal to the selected output line among a number of output lines. The output line to which the input is to be switched is selected by the output selector control lines. The 1 to 2 de-multiplexer, NL7SZ18 is a typical example for 1 to 2 de-multiplexer IC. Figure 8.10 illustrates the NL7SZ18 de-multiplexer and the function table for it.



Input		Output	
S	I	Y0	Y1
0	0	0	Z
0	1	1	Z
1	0	Z	0
1	1	Z	1

Z : High Impedance

Fig. 8.10 1 to 2 De-multiplexer IC and I/O signal states

Combinational Circuits

In digital system design, a combinational circuit is a combination of the logic gates. The output of the combinational circuit, at a given point of time, is dependent only on the state of the inputs at the given point of time. Encoders, decoders, multiplexers, de-multiplexers, adder circuits, comparators, multiple input gates, etc. are examples of digital combinational circuits.

Half adder

A half adder is combinational circuit that adds two 1-bit binary numbers and produces the result sum & carry. After simplifying the truth table, the half adder circuit can be realised using an XOR and an AND gate. The circuit implementation is shown in Fig. 8.13.

Truth Table

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

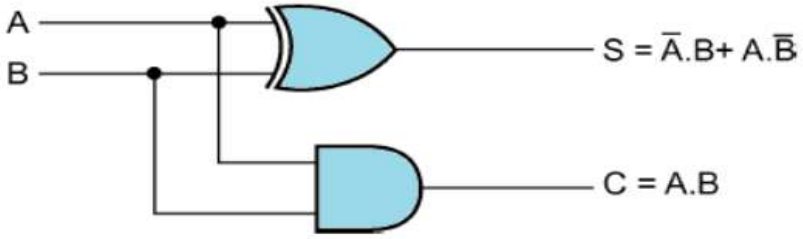


Fig. 8.13: Half Adder Circuit Implementation

Sequential Circuits

Digital logic circuit, whose output at any given point of time depends on both the present and past inputs, is known as sequential circuits. Hence, sequential circuits contain a memory element for holding the previous input states. In general, a sequential circuit can be visualised as a combinational circuit with memory elements as shown in fig.8.14.

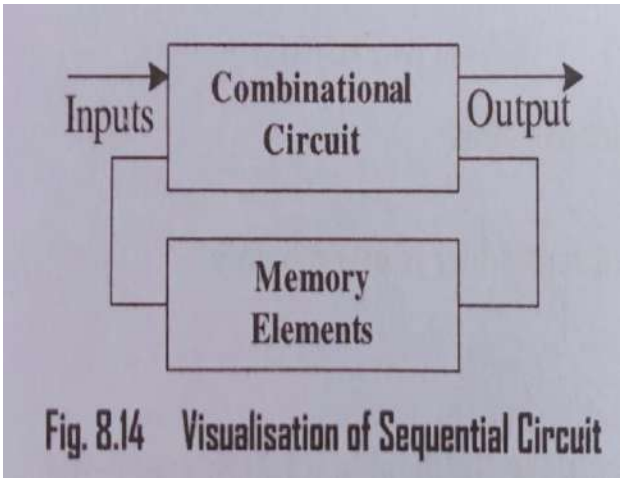


Fig. 8.14 Visualisation of Sequential Circuit

Flip-flops act as the basic building blocks of sequential circuits. Sequential circuits are of two types: synchronous (clocked) sequential circuits and asynchronous sequential circuits. The operation of a synchronous sequential circuit is synchronised to a clock signal, whereas an asynchronous sequential circuit does not require a clock for operation. Register, synchronous counters, etc. are examples of synchronous serial circuits, whereas ripple or asynchronous counter is an example for asynchronous sequential circuits.

Set Reset (S-R) flip-flop

The logic circuit and the I/O states for an S-R flip-flop are given in Fig. 8.15. The S-R flip-flop is built using 2 NOR gates. The output of each NOR gate is fed back as input to the other NOR gate. This ensures that if the output of one NOR gate is at logic 1, the output of the other NOR gate will be at logic 0.

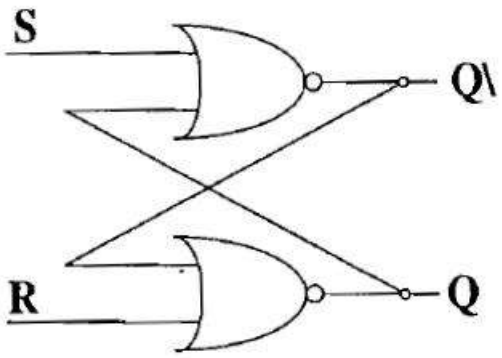
The S-R flip-flop works in the following way:

(1) If the Set input (S) is at logic high and Reset input (R) is at logic low, the output remains at logic high regardless of the previous output state.

(2) If the Set input (S) is at logic low and Reset input (R) is at logic high, the output remains at logic low regardless of the previous output state.

(3) If both the Set input (S) and Reset input (R) are at logic low, the output remains at the previous logic state.

(4) The condition Set input (S) = Reset input (R) = Logic high (1) will lead to race condition and the state of the circuit becomes undefined or indeterminate (x).



Input		Output	
S	R	Previous State	Present State
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	x
1	1	1	x

Fig. 8.15: SR flip flop and truth table

A clock signal can be used for triggering the state change of flip-flops. The clock signal can be either level triggered or edge triggered. For level triggered flip-flops, the output responds to any changes in input signal, if the clock signal is active (ie., if the clock signal is at logic 1 for 'HIGH' level triggered and at logic 0 for 'LOW' level triggered clock signal). For edge triggered flip-flops, the output state changes only when a clock trigger happens regardless of the changes in the input signal state. The clock trigger signal can be either a positive edge (0 to 1 transition) or a negative edge (1 to 0 transition). Figure 8.16 illustrates the implementation of an edge triggered S-R flip-flop.

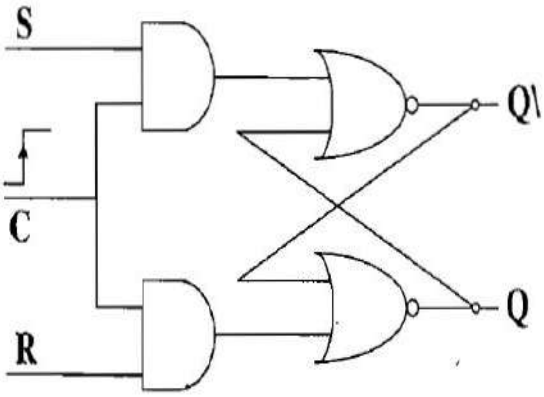
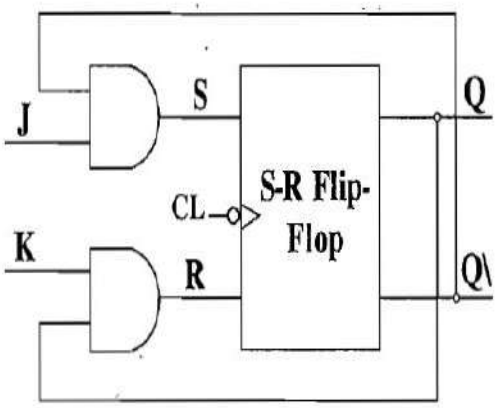


Fig. 8.16 Clocked S-R Flip-Flop



Input		Output	
J	K	Previous State	Present State
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Fig. 8.17: JK flip-flop implementation & truth table

The input condition S=R=1 is undefined in an S-R flip-flop. The J-K flip-flop augments the behaviour of SR flip-flop by interpreting the input state S=R=1 as a toggle command. The logic circuit and I/O states for JK flip-flop are given in fig.8.17.

The J-K flip-flop operates in the following way:

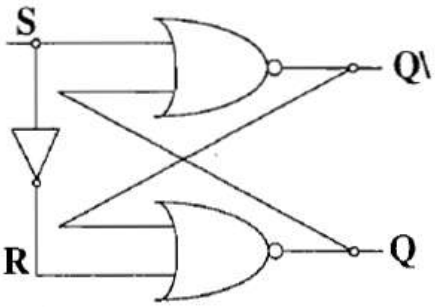
(1) When J=1 and K=0, the output remains in the set state.

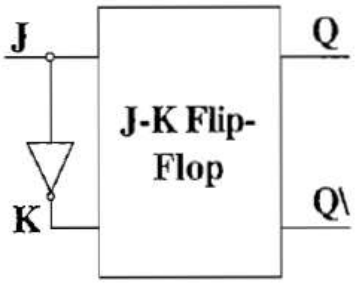
(2) When J=0 and K=1, the output remains in the reset state.

(3) When J=K=0, the output remains at the previous logic state.

4) When J=1 and K=1, the output toggles its state.

A D-type (Delay) flip-flop is formed by connecting a NOT gate in between the S and R inputs of an S-R flip-flop or by connecting a NOT gate between the J and K inputs of a J-K flip-flop. Figure 8.18 illustrates a D-type flip-flop and its I/O states.

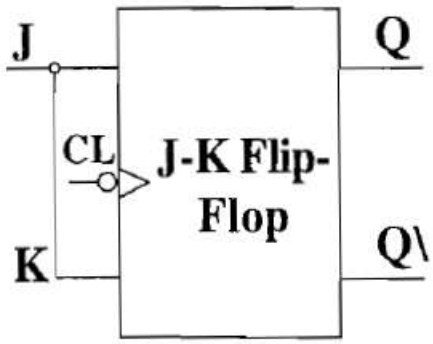




Input	Output
D	Q
0	0
1	1

Fig. 8.18 **D-Type Flip-Flop – Circuit and Truth Table**

This flip-flop is known with the so-called name 'Delay' flip-flop for the following reason-the input to the flip-flop appears at the output at the end of the clock pulse (for falling edge triggering). A Toggle flip-flop or T flip-flop is formed by combining the J and K inputs of a J-K flip-flop. Figure 8.19 illustrates a T flip-flop and its I/O states.



Input	Output
T	Q_{n+1}
0	Q_n
1	Q_n\

Q_{n+1} : Present Output

Q_n : Previous Output

Fig. 8.19 T (Toggle) Flip-Flop – Circuit and Truth Table

VLSI and Integrated Circuit Design

An Integrated Circuit (IC) is a miniaturised form of an electronic circuit containing transistors and other passive electronic components. In an IC, the circuits, required for building the functionality are built on the surface of a thin silicon wafer. Depending on the number of integrated components, the degree of integration within an integrated circuit (IC) is known as:

Small-Scale Integration (SSI): Integrates one or two logic gate(s) per IC. E.g. LS7400

Medium-Scale Integration (MSI): Integrates up to 100 logic gates in an IC. The decade Counter 7490 is an example for MSI device.

Large-Scale Integration (LSI): Integrates more than 1000 logic gates in an IC.

Very Large-Scale Integration (VLSI): Integrates millions of logic gates in an IC. Pentium processor is an example of a VLSI Device.

Depending on the type of circuits integrated in the IC, the IC design is categorised as:

Digital Design: Deals with the design of integrated circuits handling digital signals and data. The I/O requirement for such an IC is always digital. Microprocessor/microcontroller, memory, etc. are examples of digital design.

Analog Design: Deals with the design of integrated circuits handling analog signals and data. Analog design gives more emphasis to the physics aspects of semiconductor devices such as gain, power dissipation, resistance, etc. RF IC design, Op-Amp design, voltage regulator IC design, etc. are examples for Analog IC Design.

Mixed Signal Design: In today's world, most of the applications require the co-existence of digital signals and analog signals for functioning. Mixed signal design involves design of ICs, which handle both digital and analog signals as well as data. Design of an analog-to-digital converter is an example for mixed signal.

VHDL for VLSI Design

Very High Speed Integrated Circuit HDL or VHDL is a hardware description language used in VLSI design. VHDL is a technology independent description, which enables creation of designs targeted for a chosen technology (like CPLD, FPGA, etc.) using synthesis tools. VHDL can be used for describing the functionality and behaviour of the system [Behavioural representation], or describing the actual gate and register levels of the system [Register Transfer Level (RTL) representation]. VHDL supports concurrent, sequential, hierarchical and timing modelling. The concurrent modelling describes the activities happening in parallel, whereas sequential modelling describes the activities in a serial fashion one after another. Hierarchical model describes the structural aspects, and timing model models the timing requirements of the design.

The basic structure of a VHDL design consists of an entity, architecture and signals. The entity declaration defines the name of the function being modelled and its interface ports to the outside world, their direction and type. The basic syntax for an entity declaration is given below:

Entity name-of-entity **is**

Port (list of interface ports and their types);

Entity item declarations;

Begin

Statements

End entity name-of-entity

The architecture describes the internal structure and behaviour of the model. Architecture can define a circuit structure using individually connected modules or model its behaviour using suitable statements. The basic syntax of architecture body is given below:

Architecture name-of-architecture **of** name-of-entity **is**

Begin

Statements;

End architecture name-of-architecture;

VHDL also all compiled modules are stored in library. Packages are also stored in a library. A package may contain functions, types, components, etc. For using components from a particular library, first the library and packages must be specified in the VHDL code as

Library name-of-library;

Use name-of-library.name-of-package.**ALL**;

If packages or libraries are to be used, they must be defined before entity declaration. E.g.

Library ieee;

Use ieee.std_logic_1164.**ALL**;

E.g. The VHDL description of the D flip-flop is given below:

Library ieee;

Use ieee.std_logic_1164.**all**;

Entity DFF **is**

Port (D, CLK: in bit; Q: out bit);

End DFF;

Architecture DFF-ARCH **of** DFF **is**

Begin

Process (D, CLK)

Begin

If CLK='1' **and** CLK' event **then**

Q <= D;

End if;

End Process;

End DFF_ARCH;

HDL based VLSI Design Process

The HDL model of the circuit is compiled and loaded in an HDL simulator. The simulator enables the designer to apply the input stimuli to the design and observe whether the output response is as expected. If there are any functional errors, the HDL code is corrected and the design re-simulated, till the design meets the functional specifications. The design is now ready for synthesis.

A synthesiser tool compiles the source code to an optimised technology dependent circuit at the gate level. The inputs to a synthesiser are the HDL code, a technology library, and constraints. The constraints are in terms of the area and speed requirements of the circuit to be realised in the target technology. This process is done in two phases:

Compilation: The HDL is translated to a generic netlist.

Optimisation: The generic netlist is mapped to a target technology (ASIC/FPGA) satisfying the requirements of area and speed.

The next phase of implementation is physical layout (Place and Route). In the case of an FPGA implementation, during the placement stage the various instances in the netlist are mapped and their relative position inside the target FPGA resources is fixed. During the routing phase, the interconnection between the various placed instances is done as per the netlist to realise the circuit. Once the layout is complete, the final step is timing simulation. The design is now re-simulated with the actual component and interconnect delays to verify the functionality. Figure 8.25 illustrates the various steps involved in a HDL based design.

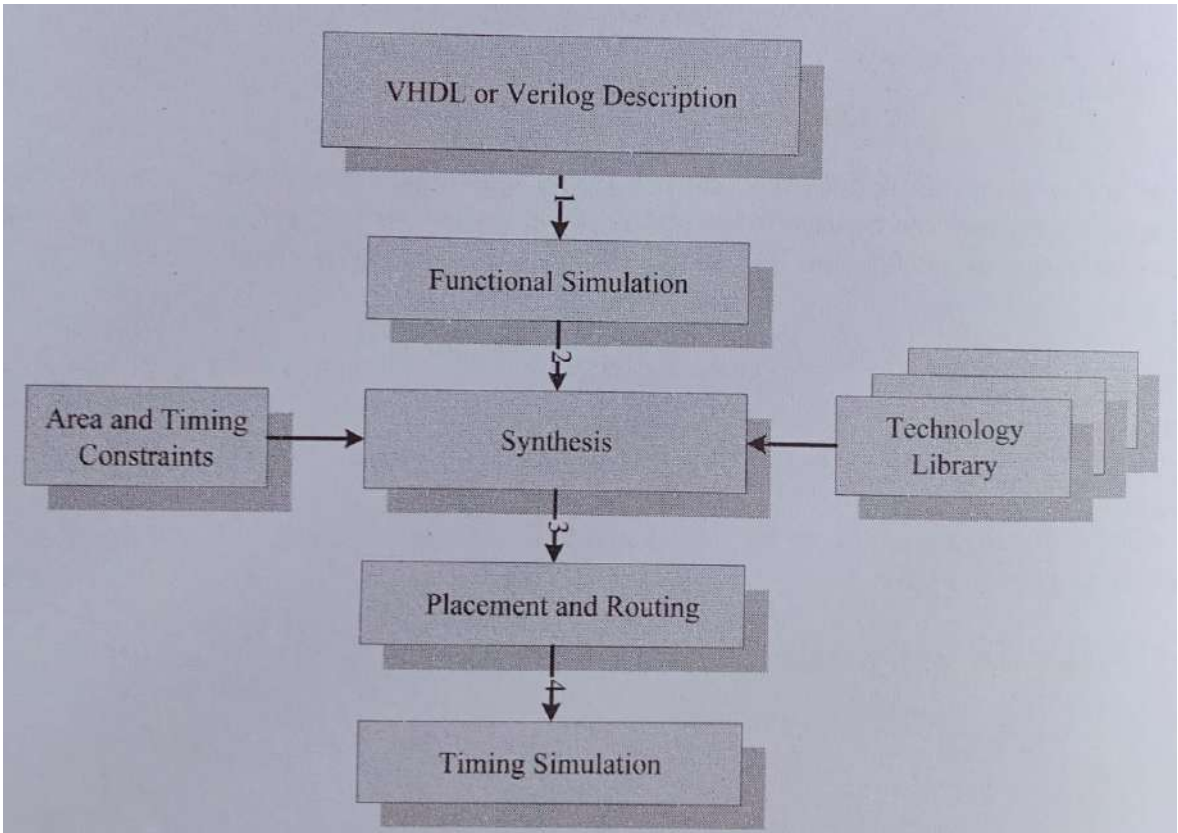


Fig. 8.25 The HDL based VLSI Design Process