**DSD(7.7.2024)**

**Weighted binary codes:** Weighted binary codes are those binary codes which obey the positional weight principle. Each position of the number represents a specific weight.

BCD: 1001, weight: 8421

## Non-Weighted Codes: In this type of binary codes, the positional weights are not assigned. The examples of non-weighted codes are *Excess-3* code, Example: 1000 and *Gray code.*

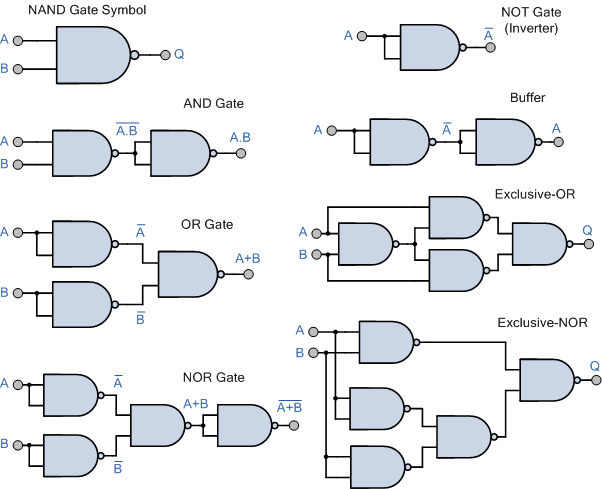
**Unite Distance Code:** Unit distance codes, also known as Gray codes, have the property that two successive values differ in only one bit position. This is useful in minimizing errors in digital systems.

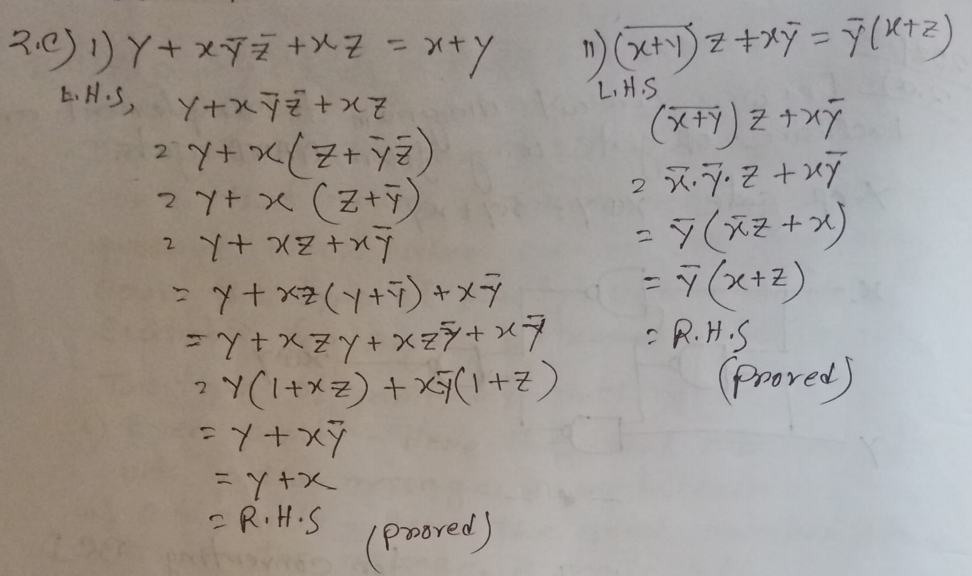
Gray code example: 111

**Parity** is a simple and effective method used for error detection in digital communication and storage systems. It involves adding an extra bit, called the parity bit, with the binary data to ensure that the total number of 1s in the data is either even or odd.

There are two types of parity: even parity and odd parity.

Parity can detect single-bit errors by checking if the total number of 1s in the received data, including the parity bit, matches the expected parity (even or odd). If it doesn't match, an error is detected.





### Basic Difference Between Parallel Adder and Serial Adder

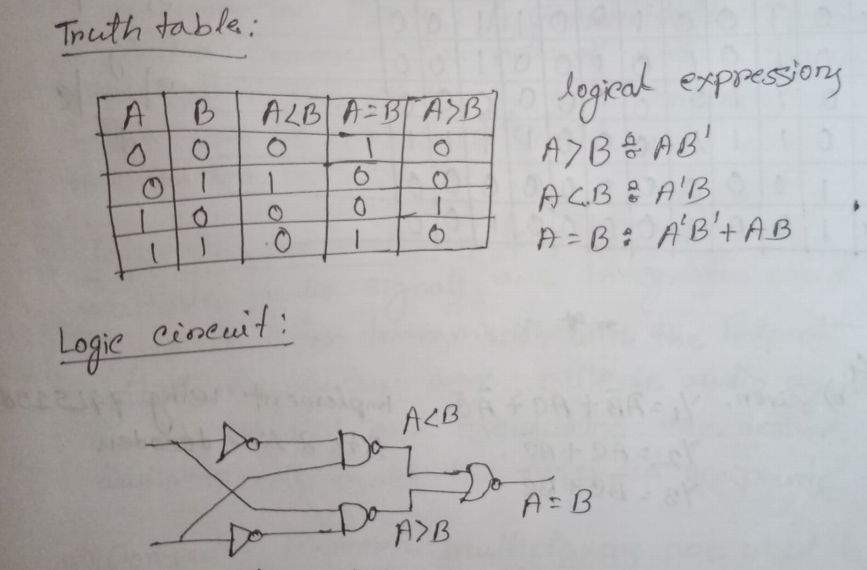
**Parallel Adder**:

* + **Operation**: Adds all bits simultaneously.
  + **Speed**: Fast.
  + **Complexity**: High (requires multiple adders).
  + **Hardware**: More components.

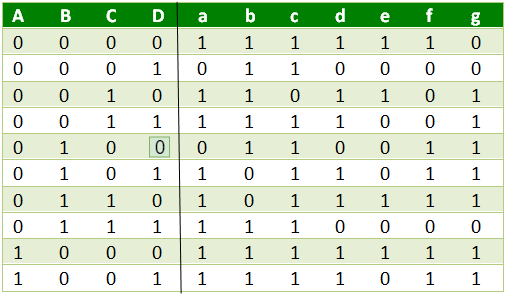
**Serial Adder**:

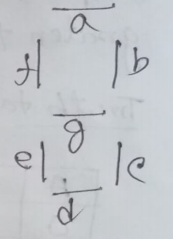
* + **Operation**: Adds bits sequentially, one at a time.
  + **Speed**: Slow.
  + **Complexity**: Low (requires fewer components).
  + **Hardware**: Fewer components.

A 1-bit comparator compares two single-bit binary values (A and B) and determines their relationship (A > B, A < B, or A = B). The logic circuit for a 1-bit comparator can be implemented using basic logic gates (AND, OR, NOT).



A BCD (Binary-Coded Decimal) to 7-segment decoder is a combinational logic circuit that converts a 4-bit BCD input into signals to drive a 7-segment display. Each segment in the display is labeled from 'a' to 'g', and the decoder activates the appropriate segments to display digits 0-9.





### Typical Uses of Multiplexer (MUX)

1. **Data Selection**: Selects one input from multiple sources.
2. **Signal Routing**: Combines multiple signals into one line.
3. **Resource Sharing**: Allows multiple inputs to share a single device.
4. **Data Compression**: Reduces the number of data paths.
5. **Control Systems**: Selects control signals based on input status.

### Typical Uses of Demultiplexer (DEMUX)

1. **Data Distribution**: Directs one input to multiple outputs.
2. **Signal Decoding**: Converts encoded data to a specific output.
3. **Memory Addressing**: Selects the correct memory location.
4. **Display Systems**: Drives multiple display segments.
5. **Test and Measurement Systems**: Routes test signals to multiple points.

### Combinational Logic Circuits

1. **Memory**: No memory elements; outputs depend only on current inputs.
2. **Timing**: No concept of time; outputs change immediately when inputs change.
3. **Components**: Built from basic logic gates (AND, OR, NOT).
4. **Examples**: Adders, subtractors, multiplexers, and decoders.

### Sequential Logic Circuits

1. **Memory**: Includes memory elements; outputs depend on current inputs and previous states.
2. **Timing**: Incorporates timing elements (clocks); outputs change at specific times based on clock signals.
3. **Components**: Built from flip-flops, latches, and basic logic gates.
4. **Examples**: Counters, registers, shift registers, and memory units.

### Synchronous Logic Circuits

1. **Clock Dependency**: Operate based on a clock signal, with changes occurring at discrete intervals (clock edges).
2. **Timing**: State changes synchronized with clock pulses, ensuring predictable behavior.
3. **Complexity**: Easier to design and debug due to well-defined timing.
4. **Examples**: Flip-flops, counters, synchronous registers.

### Asynchronous Logic Circuits

1. **Clock Dependency**: Do not rely on a clock signal; changes occur immediately when inputs change.
2. **Timing**: State changes can occur at any time, leading to potential race conditions and timing hazards.
3. **Complexity**: More complex to design and debug due to unpredictable timing.
4. **Examples**: Asynchronous counters, latches, and certain types of control circuits.

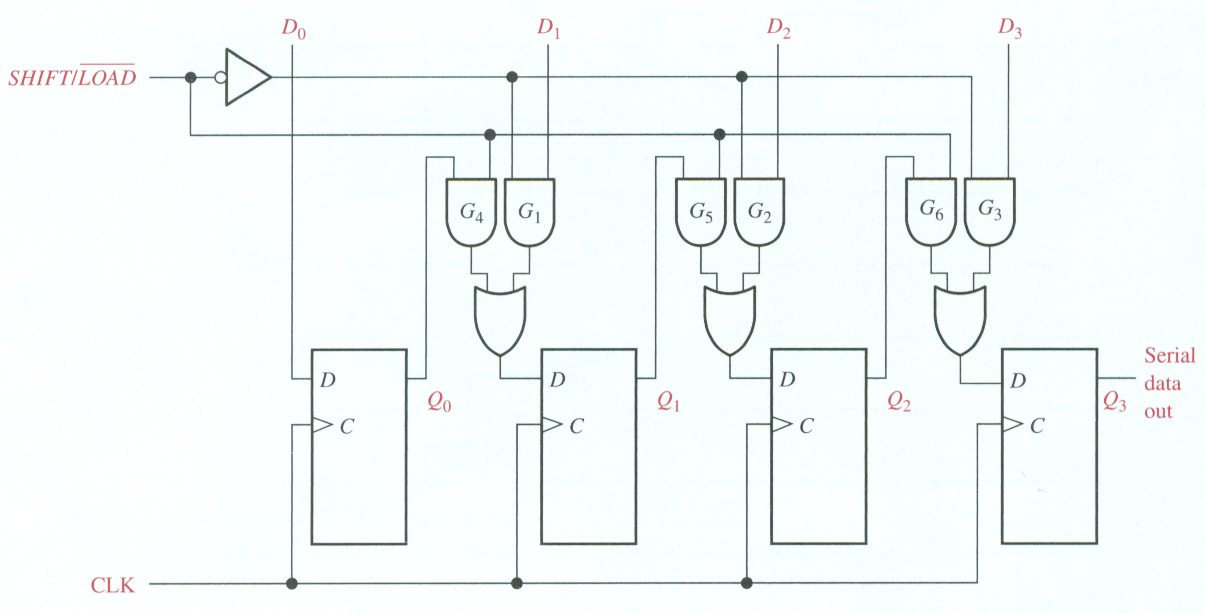
**Setup** Time (tS): The time interval before the active transition of the clock signal during which the synchronous input must be maintained.

**Hold** Time (tH): The time interval after the active transition of the clock signal during which the synchronous input must be maintained.

A register capable of shifting its binary information in one or both directions is called a shift register.

Parallel In/Serial Out Shift Registers

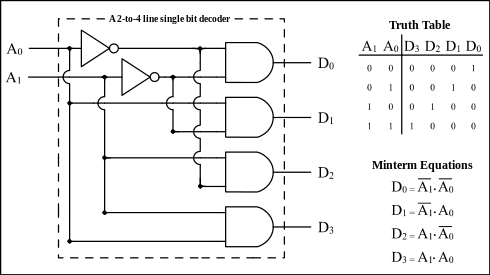
* The bits are entered simultaneously into their respective stages.
* The serial output appears bit by bit per clock pulse.
* To store 4 bits, we need 1 clock pulse
* To shift them out them, we need another 3 clock pulses.



**Decoder**: A combinational circuit that converts binary information from n coded inputs to a maximum 2n coded outputs.

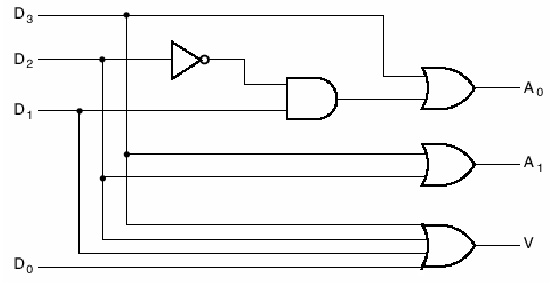
n-to-m decoder, m ≤ 2n

Examples: BCD-to-7-segment decoder, where n=4 and m=10



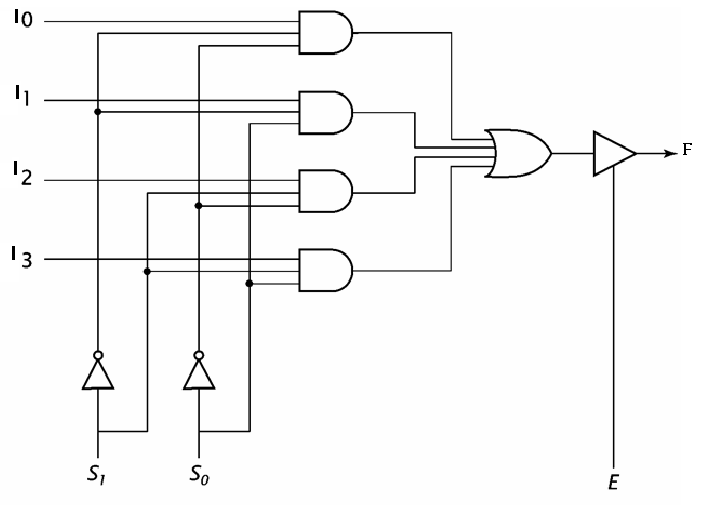
**Encoder**: An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2n input lines and n output lines.

**Priority encoder** in electronics is a digital circuit that takes multiple inputs and outputs the binary code of the highest-priority active input. If multiple inputs are active, it encodes the input with the highest priority.

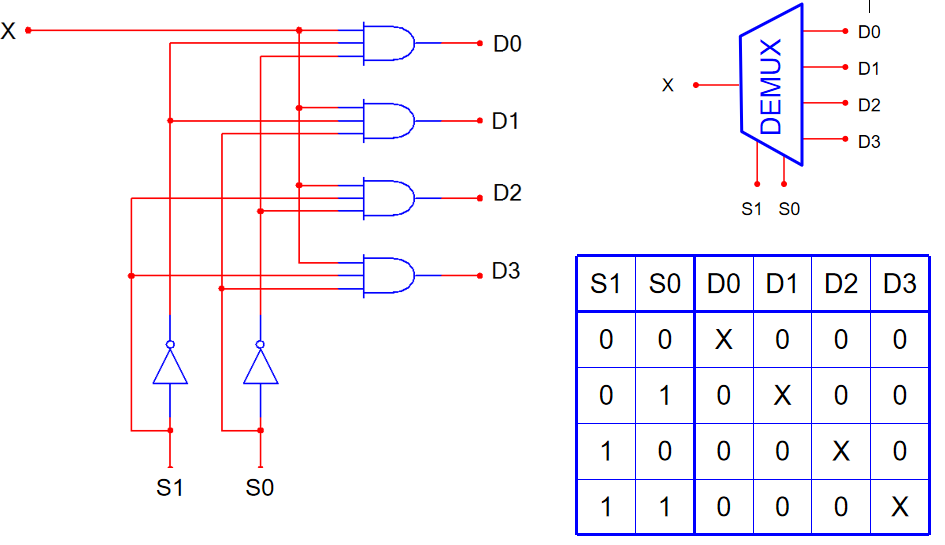


A **MUX** is a digital switch that has multiple inputs (sources), select lines determine which input is connected to the output and a single output (destination).

*4-to-1 Multiplexer*



A **DEMUX** is a digital switch with a single input (source) and a multiple outputs (destinations) and the select lines determine which output the input is connected to.



**Difference between Multiplexer (MUX) and Decoder:**

MUX:

- Uses n select lines to choose from 2ⁿ inputs.

- Has 1 output line.

- Outputs the data of the selected input.

- Used for data selection.

Decoder:

- Uses n inputs to activate one of 2ⁿ output lines.

- No select lines; output depends on the input combination.

- Converts binary input to a unique active output.

- Used for binary-to-decimal conversion and address decoding.

**Flip-Flop:**

* **Clocked Device**: Changes state only on clock edges (rising or falling).
* **Synchronous**: Operates based on a clock signal, making it predictable and synchronized.
* **Edge-Triggered**: State changes at specific points in time (on clock edges).
* **Applications**: Used in registers, counters, and state machines.

**Latch:**

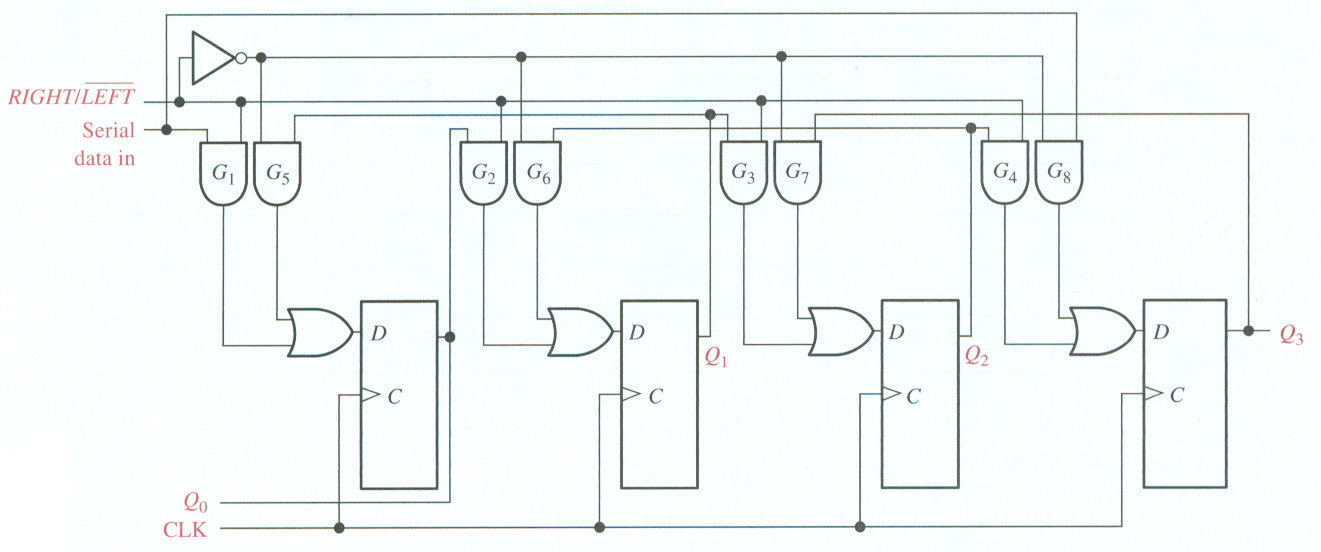
* **Level-Sensitive**: Changes state based on the level of the control signal (high or low).
* **Asynchronous**: Operates as long as the control signal is active, not tied to clock edges.
* **Level-Triggered**: State changes while the control signal is active.
* **Applications**: Used in temporary storage, memory elements, and simple data holding.

A **register** is a digital storage element that holds data consists of a group of flip-flops or latches connected together to store multiple bits of data.

**Applications**:

* Storage registers
* Frequency division
* Shift registers
* Counters

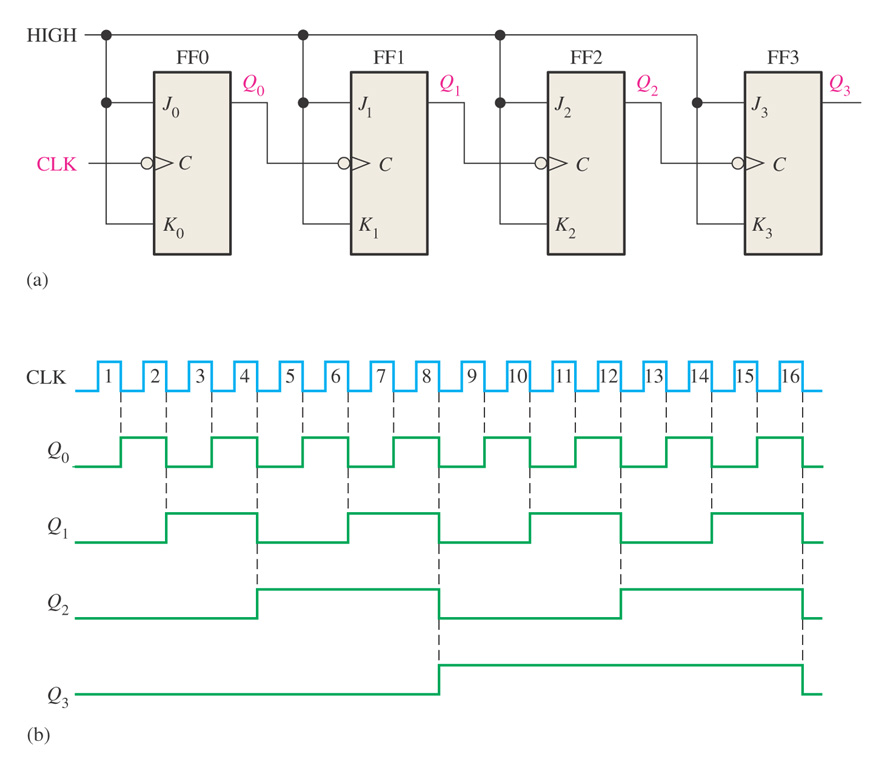
A **bidirectional** shift register is one in which the data can be shifted either left or right.



**Counter** is a register that goes through a predetermined sequence of states upon the application of clock pulses.

**Asynchronous counters:**

1. A counter that follows the binary number sequence is called a binary counter.
2. Only the first flip-flop is clocked by an external clock. All subsequent flip-flops are clocked by the output of the preceding flip-flop.
3. Asynchronous counters are **slower** than synchronous counters because of the delay in the transmission of the pulses from flip-flop to flip-flop.
4. Asynchronous counters are also called **ripple-counters** because of the way the clock pulse ripples it way through the flip-flops.



4-bit asynchronous binary counter and its timing diagram for one cycle.

**Synchronous counters:**

1. All flip-flops are clocked simultaneously by an external clock.
2. Synchronous counters are faster than asynchronous counters because of the simultaneous clocking.
3. Synchronous counters are an example of state machine design because they have a set of states and a set of transition rules for moving between those states after each clocked event.

|  |  |
| --- | --- |
| **Asynchronous Counter** | **Synchronous Counter** |
| Clock input is applied to LSB FF. The output of first FF is connected as clock to next FF. | Clock input is common to all FF. |
| All Flip-Flops are toggle FF. | Any FF can be used. |
| Speed depends on no. of FF used for n bit . | Speed is independent of no. of FF used. |
| No extra Logic Gates are required.  Cost is less. | Logic Gates are required based on design. Cost is more. |

**Different Between latch and Flip Flop:**

· **Basic Definition**:

* · **Latch**: A latch is a level-sensitive storage element that changes its output immediately when its input changes, as long as the enable signal is active.
* **Flip-Flop**: A flip-flop is an edge-triggered storage element that changes its output only at the rising or falling edge of the clock signal.

· **Triggering**:

* · **Latch**: Level-triggered (changes state as long as the control/enable signal is active).
* **Flip-Flop**: Edge-triggered (changes state only on a clock edge, either rising or falling).

· **Timing Control**:

* · **Latch**: Asynchronous; output can change immediately when the enable signal is active.
* **Flip-Flop**: Synchronous; output changes only at a specific clock edge.

· **Common Use**:

* · **Latch**: Used when immediate changes in output are needed (for simple data storage).
* **Flip-Flop**: Used for synchronized operations, like in counters, shift registers, and memory elements.

**HDL**

HDL is designed for circuit verification and simulation, for timing analysis, for test analysis and for logic synthesis.

**Behavioral description**

**RTL description (HDL)**

**Functional verification and testing**

**Logic synthesis/timing verification**

**Gate-Level Netlist**

**Logic verification and testing**

**Floor planning Automatic place and Route**

**Physical layout**

**Layout verification**

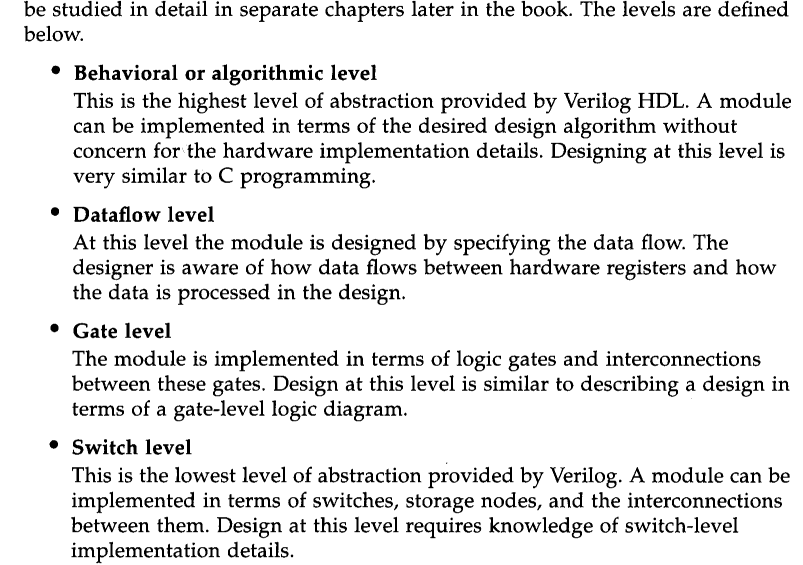
**Implementation**

**Design specification**

HDL Design Flow

The process of creating a object from module template is called **instantiation**.

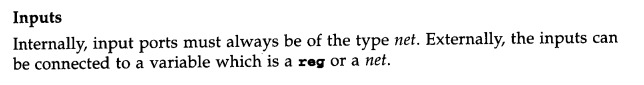
**Different Data Level Abstraction:**

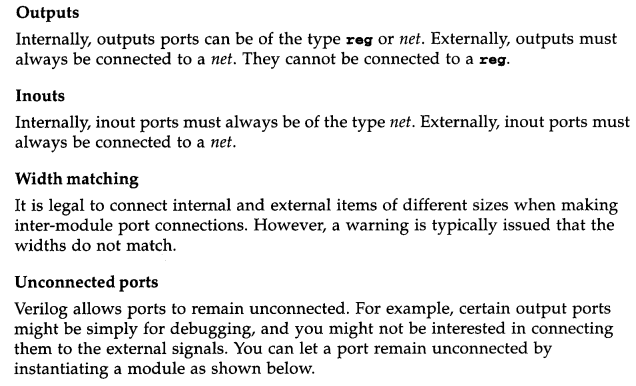


Describes system functionally

interconnection of components

**Port Connection Rules:**





Data Type in HDL

Net Types:

Represents connections between hardware elements (e.g., wire in Verilog).

Used for combinational logic and continuous assignments.

Register Types:

Represents storage elements (e.g., reg in Verilog).

Used to hold data and maintain state in sequential logic.

Vector Types:

Collections of bits (e.g., wire [3:0] for a 4-bit vector).

Used to represent multi-bit signals or data buses.

**HDL Description:**

· **Behavioral**:

* · Describes functionality (what it does).
* Uses constructs like if-else.

· **Structural**:

* · Describes interconnections of components.
* Instantiates modules and connects them.

· **Dataflow**:

* · Describes data movement through the system.
* Uses continuous assignments (e.g., assign).

An **excitation table** shows the required inputs to transition a flip-flop from its **current state** to a **next state**. It helps determine the inputs needed to achieve a specific state transition in flip-flops.

### ****Excitation Tables of Flip-Flops****:

**SR Flip-Flop**:

| **Current State (Q)** | **Next State (Q+)** | **S** | **R** |
| --- | --- | --- | --- |
| 0 | 0 | 0 | X |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | X | 0 |

**T Flip-Flop**:

| **Current State (Q)** | **Next State (Q+)** | **T** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**JK Flip-Flop**:

| **Current State (Q)** | **Next State (Q+)** | **J** | **K** |
| --- | --- | --- | --- |
| 0 | 0 | 0 | X |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | X | 0 |

**D Flip-Flop**:

| **Current State (Q)** | **Next State (Q+)** | **D** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

module array\_multiplier\_2x2 (A, B, P);

input [1:0] A;

input [1:0] B;

output [3:0] P;

// Partial products

wire p0, p1, p2, p3;

// AND gates for partial products

assign p0 = A[0] & B[0]; // A0 \* B0

assign p1 = A[0] & B[1]; // A0 \* B1

assign p2 = A[1] & B[0]; // A1 \* B0

assign p3 = A[1] & B[1]; // A1 \* B1

// Summing the partial products

assign P[0] = p0; // LSB

assign P[1] = p1 ^ p2; // Sum of A0\*B1 and A1\*B0

assign P[2] = p3 ^ (p1 & p2); // Sum with carry

assign P[3] = p3 & (p1 | p2); // Carry to MSB

endmodule