DIGITAL SYSTEM DESIGN USING VERILOG
VTU EXAMINATION 2025-2026
SOLUTIONS

CBCS SCHEME

USN | 1 | C | R | 2 | 4 | E | C | 1 | 3 | 7 |

BEC302

Third Semester B.E./B.Tech. Degree Examination, Dec.2025/Jan.2026
Digital System Design Using Verilog

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

			M	L	C
		Module – 1			
Q.1	a.	Define the following : i) Canonical SOP ii) Canonical POS iii) Essential Prime Implicant (EPI) iv) Prime Implicant (PI) v) Min term and Max term with a table of two input variable representation	10	L1	CO1
	b.	Using Quine Mccluskey method and PI reduction table, determine the minimal SOP expression for the following function. $Y = f(a, b, c, d) = \Sigma(1, 2, 3, 5, 9, 12, 14, 15) + \Sigma d(4, 8, 11)$	10	L3	CO1
		OR			
Q.2	a.	Explain the procedure to place SOP and POS equations into canonical form convert the following equations into canonical forms i) $T = f(a, b, c) = (a + b')(b' + c)$ ii) $G = f(w, x, y, z) = w'x + yz'$	10	L3	CO1
	b.	Solve the given functions using Kamangh Map i) $F = f(w, x, y, z) = \Sigma(0, 7, 8, 9, 10, 12) + \Sigma d(2, 5, 13)$ ii) $G = f(a, b, c, d) = \pi(0, 4, 5, 7, 8, 9, 11, 12, 13, 15)$ Also identify the prime implicants.	10	L3	CO1
		Module – 2			
Q.3	a.	Implement the function using multiplexers i) $f(x, y, z) = \Sigma m(0, 2, 3, 5)$ using 4 × 1 multiplexer ii) $f(w, x, y, z) = \Sigma m(0, 1, 5, 6, 7, 9, 12, 15)$ using 8 × 1 multiplexer.	10	L2	CO2
	b.	Explain the concept of carry look ahead adder with related equation and block diagram.	10	L2	CO2
		OR			
Q.4	a.	Using a 4-bit binary adder, design a logic to convert a decimal digit in 8421 code into a decimal adder.	10	L3	CO2
	b.	Implement the following functions using 3 : 8 decoder. i) $P = f(w, x, y, z) = \Sigma(1,4, 8, 13)$ $Q = g(w, x, y, z) = \Sigma(2, 7, 13, 14)$ ii) $A = f(x, y, z) = \pi(0, 1, 3, 5)$ $B = f(x, y, z) = \Sigma(1, 4, 5, 7)$	10	L3	CO2

		Module – 3			
Q.5	**a.**	Explain master slave JK flip-flop with the help of circuit diagram and timing diagrams.	**10**	**L2**	**CO3**
	b.	Design a synchronous Mod -6 counter with sequence 0-2-3-6-5-1 using JK flip-flop.	**10**	**L3**	**CO3**
		OR			
Q.6	**a.**	Design a 4-bit binary ripple up counter with logic diagram and counting sequence and explain its operation.	**10**	**L3**	**CO3**
	b.	With the help of logic diagram and counting sequence, explain. i) Ring counter ii) Mod – 8 twisted ring counter.	**10**	**L2**	**CO3**
		Module – 4			
Q.7	**a.**	What are the different data types in verilog. Explain with examples.	**10**	**L2**	**CO4**
	b.	i) Develop a verilog program to implement 2×1 multiplexer using conditional operator. Also write the truth table of 2×1 mux ii) Design a 2×1 multiplexer using dataflow verilog description draw a logic diagram and logic gate level circuit for it.	**4** **6**	**L3** **L3**	**CO4** **CO4**
		OR			
Q.8	**a.**	Discuss in detail different description styles in verilog.	**8**	**L2**	**CO4**
	b.	Let A = 5' b11011, B = 5' b 10101 C = 4' d3 Determine the output the following verilog statements. i) d = & A ii) e = ~^ 4' b1011 iii) f = ~ (A & (~B)) iv) g= A| |B v) b = 3** 2 vi) i = {2{A}}	**12**	**L3**	**CO4**
		Module – 5			
Q.9	**a.**	Design a 4-bit counter with synchronous hold using verilog. Also draw the simulation waveform.	**10**	**L3**	**CO5**
	b.	Explain the operation of positive edge triggered JK flip-flop by using a verilog code using case statement. Write truth table and timing diagram.	**10**	**L2**	**CO5**
		OR			
Q.10	**a.**	Explain the operation of half adder and implement using structural description in verilog.	**6**	**L2**	**CO5**
	b.	Write the verilog format of case statement and explain.	**4**	**L2**	**CO5**
	c.	Develop a verilog behavioral description for 3 bit binary up counter.	**6**	**L3**	**CO5**
	d.	Develop a verilog program for D latch using behavioral description style.	**4**	**L3**	**CO5**

* * * * *

MODULE 1

Q1. a) Define the following:

i) Canonical SOP

An expression where each product term (minterm) contains all variables either in complemented or uncomplemented form and outputs are ORed.

Example:

F(A,B)=A′B+AB′

ii) Canonical POS

An expression where each sum term (maxterm) contains all variables, and all such terms are ANDed.

Example:

F(A,B)=(A+B′)(A′+B)

iii) Essential Prime Implicant (EPI)

A prime implicant that covers at least one minterm not covered by any other PI.

iv) Prime Implicant (PI)

A product term that cannot be further combined with another term using Karnaugh map grouping.

v) Minterm and Maxterm (2-variable)

A	B	Minterm	Maxterm
0	0	m_0=A'B'	M_0=A+B
0	1	m_1=A'B	M_1=A+B'
1	0	m_2=AB'	M_2=A'+B
1	1	m_3=AB	M_3=A'+B'

1b) Quine Mccluskey Method:

$Y = f(a,b,c,d) = \Sigma(1,2,3,5,9,12,14,15) + \Sigma d(4,8,11)$

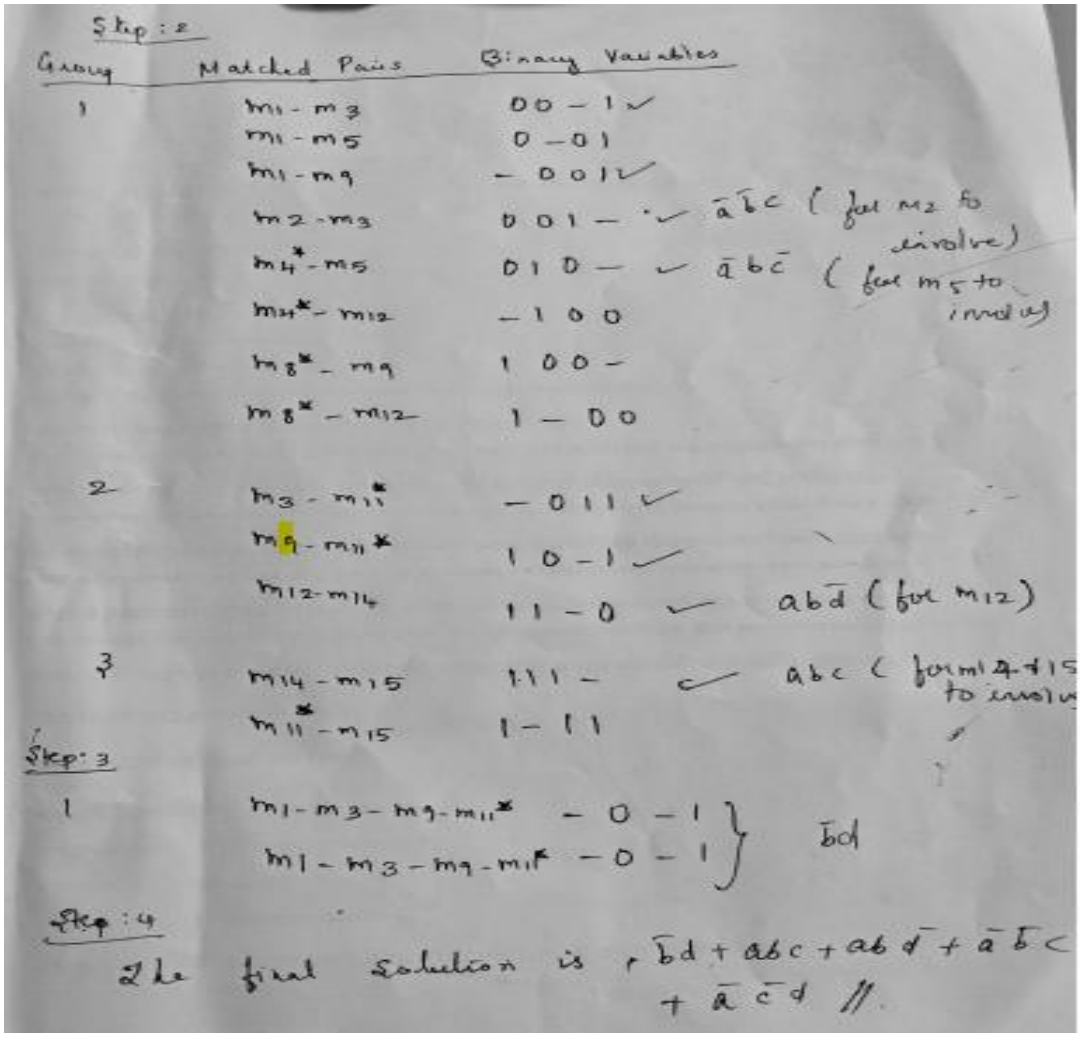
1 → 0001 ✓
2 → 0010 ✓
3 → 0011 ✓
5 → 0101 ✓
9 → 1001 ✓
12 → 1100 ✓
14 → 1110 ✓
15 → 1111 ✓

don't care terms

4 → 0100 ✓
8 → 1000 ✓
11 → 1011 ✓

Step :1

Group	Minterms	Binary Variable
1	m_1	0001 ✓
	m_2	0010 ✓
	m_4	0100* ✓
	m_8	1000* ✓
2	m_3	0011 ✓
	m_5	0101 ✓
	m_9	1001 ✓
	m_{12}	1100 ✓
3	m_{14}	1110* ✓
	m_{11}	1011* ✓
4	m_{15}	1111 ✓



Q2. a)Explain procedure to convert to canonical form & convert:

i). T=f(a,b,c)=(a+b′)(b+c)

Expand:

T =(ab+ac+b′b+b′c)

Since b′b=0:

=ab+ac+b′c

Convert to canonical SOP:

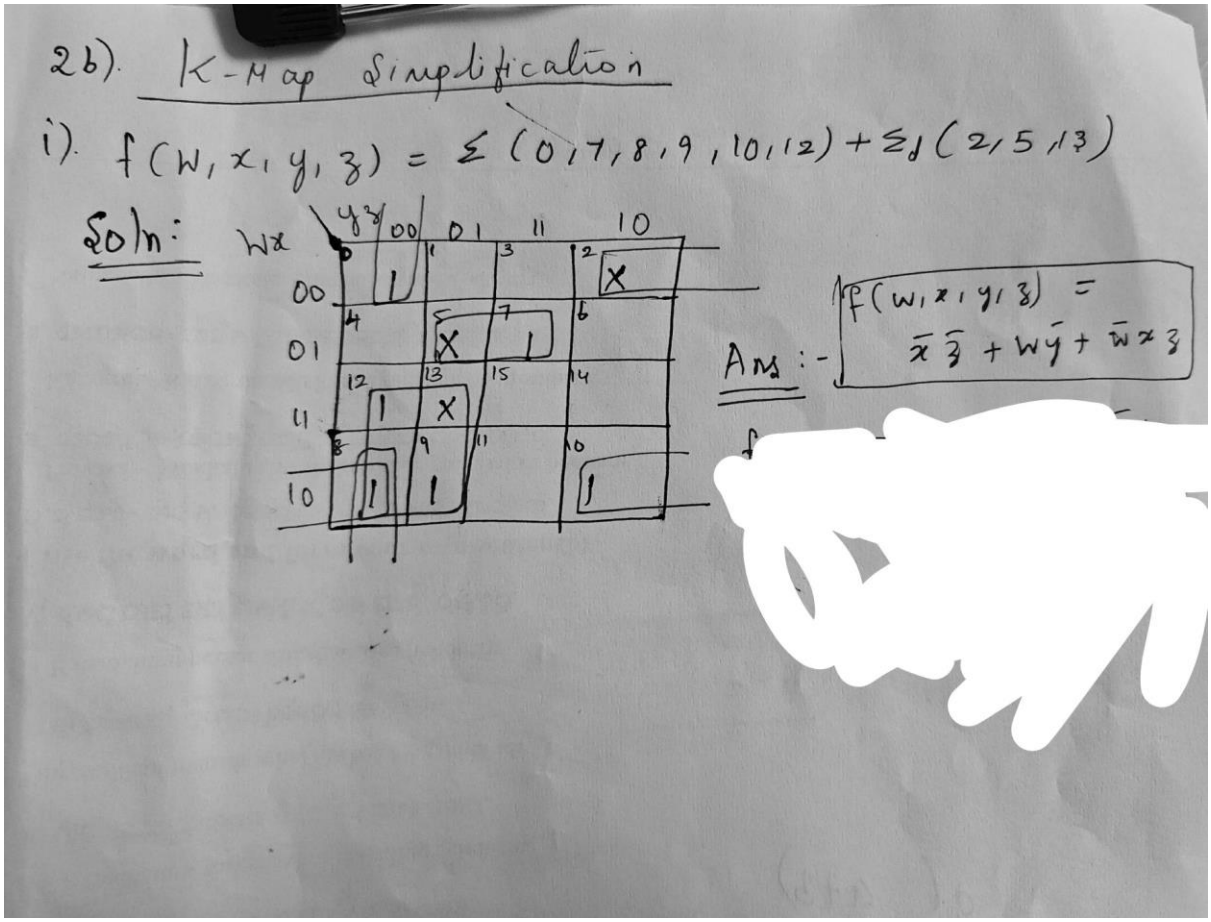
T= ab(c+c′)+ac(b+b′)+b′c(a+a′)

=abc+abc′+abc+ab′c+a′b′c

T=Σ(1,3,6,7)

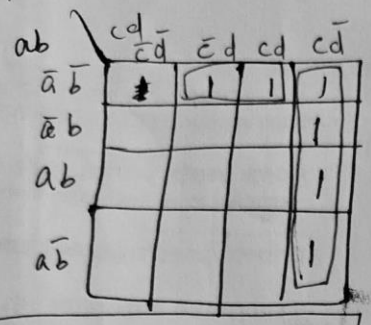
ii). G=f(w,x,y,z)=wx+yz′

Convert to canonical SOP:
wx(y+y′)(z+z′)+yz′(w+w′)(x+x′) =Σ(12,13,14,15,2,6,10,14)



2.b) ii) $f(a,b,c,d) \neq \Pi\ (0,4,5,7,8,9,11,12,13,15)$-

Complement to POS is $\Sigma\ (1,2,3,6,10,14)$

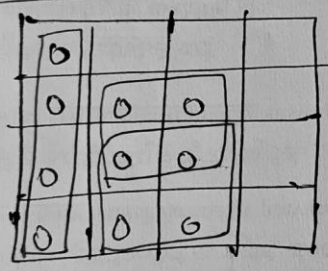


Sol^n :-

Ans = $c\bar{d} + \bar{a}\bar{b}d$ // ← SOP Ans, hence POS

hence ,

Ans: $(c+d)\cdot(\bar{a}+\bar{d})\cdot(\bar{b}+\bar{d})$ ← POS



← POS //

MODULE 2

Q3. a) f(x,y,z)=Σ(0,2,3,5) Use **4×1 MUX**, select lines = y,z

yz	Output
00	1
01	x
10	1
11	x'

Implementation using data inputs:
D_0=1, D_1=x, D_2=1, D_3=x'

To implement the function $f(x, y, z) = \sum(0, 2, 3, 5)$ using a 4×1 MUX with select lines y and z, the required data inputs are $I_0 = \bar{x}$, $I_1 = x$, $I_2 = \bar{x}$, and $I_3 = \bar{x}$.

→ Step 1: Create the Truth Table

Map the function outputs for all combinations of the variables x, y, z:

x	y	z	Minterm	f
0	0	0	m_0	1
0	0	1	m_1	0
0	1	0	m_2	1
0	1	1	m_3	1
1	0	0	m_4	0
1	0	1	m_5	1
1	1	0	m_6	0
1	1	1	m_7	0

➞ Step 2: Determine MUX Inputs

With y and z as select lines ($S_1 = y, S_0 = z$), we group the truth table by (y, z) pairs and determine the input relative to x:

- **For $yz = 00$ (I_0):** $f = 1$ when $x = 0$ and $f = 0$ when $x = 1$. Therefore, $I_0 = \bar{x}$.
- **For $yz = 01$ (I_1):** $f = 0$ when $x = 0$ and $f = 1$ when $x = 1$. Therefore, $I_1 = x$.
- **For $yz = 10$ (I_2):** $f = 1$ when $x = 0$ and $f = 0$ when $x = 1$. Therefore, $I_2 = \bar{x}$.
- **For $yz = 11$ (I_3):** $f = 1$ when $x = 0$ and $f = 0$ when $x = 1$. Therefore, $I_3 = \bar{x}$.

✓ Answer:

The function is implemented with a 4×1 MUX as follows:

- Select Lines: $S_1 = y, S_0 = z$
- Data Inputs: $I_0 = \bar{x}, I_1 = x, I_2 = \bar{x}, I_3 = \bar{x}$

i) f(w,x,y,z)=Σ(0,1,5,6,7,9,12,15)

Use **8×1 MUX**, select = x,y,z, Data inputs obtained by evaluating w.

To implement the function $f(w, x, y, z) = \Sigma(0, 1, 5, 6, 7, 9, 12, 15)$ using an 8×1 MUX with x, y, z as select lines, the data inputs are: $I_0 = \bar{w}$, $I_1 = 1$, $I_2 = 0$, $I_3 = 0$, $I_4 = w$, $I_5 = \bar{w}$, $I_6 = \bar{w}$, and $I_7 = 1$.

➞ Step 1: Map Variables and Select Lines

The function has four variables (w, x, y, z). Since an 8×1 MUX has $2^3 = 8$ inputs, we use three variables as **select lines** (S_2, S_1, S_0). Given $S_2 = x$, $S_1 = y$, and $S_0 = z$, the fourth variable w will be used to determine the logic level of the data inputs I_0 through I_7.

➞ Step 2: Construct the Implementation Table

We create a table to compare minterms when $w = 0$ and $w = 1$ for each combination of x, y, z. The minterm index is calculated as $8w + 4x + 2y + 1z$.

w	$I_0(000)$	$I_1(001)$	$I_2(010)$	$I_3(011)$	$I_4(100)$	$I_5(101)$	$I_6(110)$	$I_7($
$w = 0$	**0**	**1**	2	3	4	**5**	**6**	**7**
$w = 1$	8	**9**	10	11	**12**	13	14	**15**

Note: Bold numbers indicate minterms present in the given function
$\Sigma(0, 1, 5, 6, 7, 9, 12, 15)$.

➞ Step 3: Determine Data Input Logic

For each column, we evaluate the input based on which rows are circled (included in the function):

- If both $w = 0$ and $w = 1$ are included: $I_n = 1$
- If only $w = 0$ is included: $I_n = \bar{w}$
- If only $w = 1$ is included: $I_n = w$
- If neither is included: $I_n = 0$

1. I_0: Only 0 is present → $I_0 = \bar{w}$
2. I_1: Both $1, 9$ are present → $I_1 = 1$
3. I_2: Neither is present → $I_2 = 0$
4. I_3: Neither is present → $I_3 = 0$
5. I_4: Only 12 is present → $I_4 = w$
6. I_5: Only 5 is present → $I_5 = \bar{w}$
7. I_6: Only 6 is present → $I_6 = \bar{w}$
8. I_7: Both $7, 15$ are present → $I_7 = 1$

✓ Answer:

The 8×1 MUX implementation with select lines $S_2 = x, S_1 = y, S_0 = z$ requires the following data inputs:

$$I_0 = \bar{w}, I_1 = 1, I_2 = 0, I_3 = 0, I_4 = w, I_5 = \bar{w}, I_6 = \bar{w}, I_7 = 1$$

3.b Carry Looakahead Adder.

Carry Lookahead Adder

All the networks are subject to ripple effect, as the operands are applied in parallel. The ripple effect dictates the overall speed at which the network operates.

By considering the figure from binary, it is possible that a carry is generated in the least-significant-bit-position stage & owing to the operands, this carry must propagate through all the remaining stages to the highest-order-bit position stage.

For example, such a situation occurs when the two n-bit operands are 01...11 and 00...01, leads to a n-bit sum 10...00 is produced.

Two levels of logic are needed to generate the carry at the least-significant-bit position stage, two levels of logic are needed to propagate the carry through each of the next n-2 higher order stages, and two levels of logic are needed to form the sum or carry at highest order-bit position stage.

If each stage is assumed to require a unit time of propagation delay, then the maximum propagation delay of the ripple adder becomes 2n units of time.

The disadvantage is that, all signals must complete their propagations through a network, before new inputs are applied. This becomes a limiting factor in the adder's overall speed of operation.

To decrease the time required to perform addition, an effort must be made to speed up the propagation of the carries. One approach, is to reduce the number of logic levels in the path of the propagated carries. Adders designed with this consideration are called High-Speed Adders.

Equations sum and carry,

$$C_{i+1} = x_i y_i + x_i c_i + y_i c_i \longrightarrow (1)$$

$$S_i = c_i \oplus (x_i \oplus y_i) \longrightarrow (2)$$

are the outputs at the i^{th} stage of a binary adder.

The sum and carry outputs at a given stage are a function of the output carry from the previous stage, which in turn is a function of the output carry from another previous stage. This corresponds to undesirable ripple effect.

If the input carry at a given stage is expressed in terms of the operands variables ie, $x_0, x_1, \ldots x_{n-1}$ and $y_0, y_1, \ldots y_{n-1}$, then the ripple effect is eliminated & the overall speed of the adder is increased.

Let,

$$C_{i+1} = x_i y_i + x_i c_i + y_i c_i$$
$$= x_i y_i + (x_i + y_i) c_i$$

The first term in the last equation, $x_i y_i$ is called carry-generate function, since it corresponds to the formation of a carry at the ith stage.

The second term, $(x_i + y_i) c_i$ corresponds to a previously generated carry c_i, that must propagate past the ith stage to the next stage.

The $x_i + y_i$ part of this term is called carry-propagate function.

Letting the carry-generate function be denoted by Boolean variable "g_i" and the carry propagate function by "P_i"; i.e.,

$$g_i = x_i y_i \longrightarrow (3)$$

$$P_i = x_i + y_i \longrightarrow (4)$$

The output carry equation for the ith stage is given by

$$C_{i+1} = g_i + P_i C_i \longrightarrow (5)$$

The output carry at each of the stages can be written in terms of carry-generate functions, carry-propagate functions and the initial input carry C_0 as follows,

$$C_1 = g_0 + P_0 C_0 \longrightarrow (6)$$

$$\begin{aligned} C_2 &= g_1 + P_1 C_1 \\ &= g_1 + P_1 (g_0 + P_0 C_0) \\ &= g_1 + P_1 g_0 + P_0 P_1 C_0 \longrightarrow (7) \end{aligned}$$

$$\begin{aligned} C_3 &= g_2 + P_2 C_2 \\ &= g_2 + P_2 (g_1 + P_1 g_0 + P_1 P_0 C_0) \\ &= g_2 + P_2 g_1 + P_2 P_1 g_0 + P_2 P_1 P_0 C_0 \longrightarrow (8) \end{aligned}$$

$$C_4 = g_3 + P_3 C_3$$

$$= g_3 + P_3 (g_2 + P_2 g_1 + P_2 P_1 g_0 + P_2 P_1 P_0 C_0)$$

$$= g_3 + P_3 g_2 + P_3 P_2 g_1 + P_3 P_2 P_1 g_0 + P_3 P_2 P_1 P_0 C_0$$

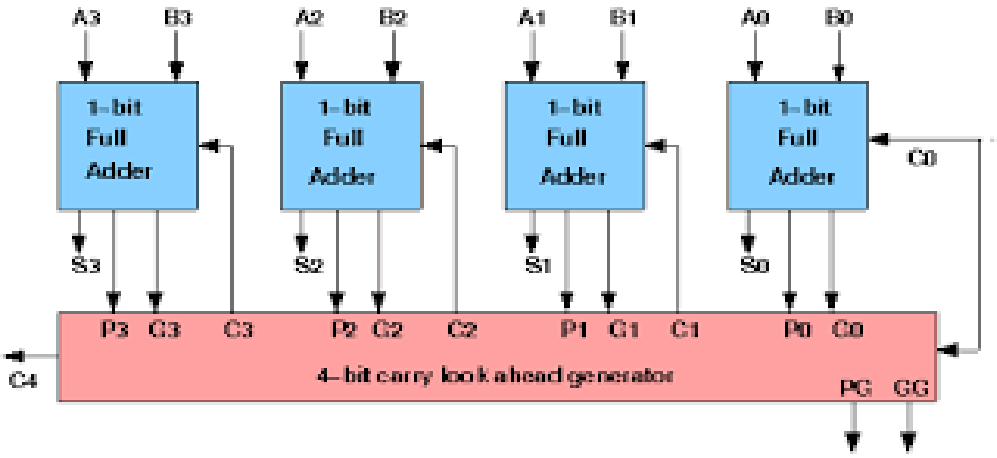
$$\vdots$$

$$C_{i+1} = g_i + P_i g_{i-1} + P_i P_{i-1} g_{i-2} + \cdots + P_i P_{i-1} \cdots P_1 g_0 + P_i P_{i-1} \cdots P_0 C_0 \longrightarrow (4)$$

Since each carry-generate function and carry-propagate function is itself only a function of the operand variables as indicated by eqn (3) & (4), the output carry and the input carry, at each stage can be expressed as a function of the operand variables & the initial input carry "C_0".

Since the output sum bit at any stage is also a function of the previous stage output carry as indicated by equation (2).

Thus, the parallel adders whose realizations are based on the above equations are called carry lookahead adders



Q4. a) Decimal digit in 8421 → BCD adder

Decimal Adder -

The digital systems are required to handle decimal numbers.

Owing to the availability and reliability of two-valued circuits, the decimal digits are represented by groups of binary digits.

When performing arithmetic in these digital systems, the system must be capable of accepting the operands in some binary coded form and producing results also in the same coding scheme. The general form of a single-decade decimal adder, i.e. an adder corresponding to a single-order digit position is given in figure.

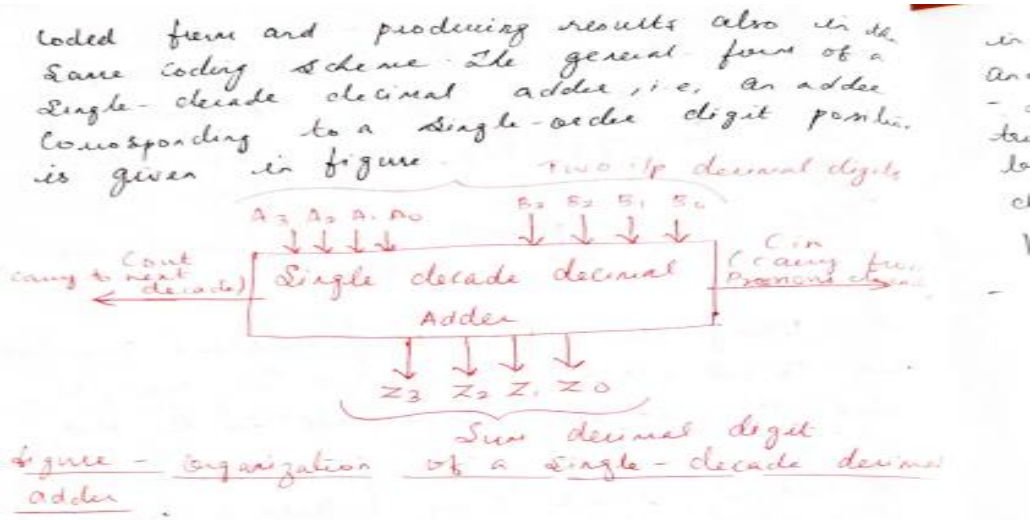


figure - Organization of a single-decade decimal adder

From this figure, it is assumed that a 4-bit code is used for each decimal digit. The two decimal digits serving as operands are denoted by $A_3A_2A_1A_0$ & $B_3B_2B_1B_0$.

A carry denoted by C_{in} appears as an input from the addition of the previous decade.

→ C_{in} & C_{out} are represented as 0 or 1.

→ The five output variables Z_3, Z_2, Z_1, Z_0 & C_{out} are Boolean functions of the rest

input variables $A_3, A_2, A_1, A_0, B_3, B_2, B_1, B_0$ and C_{in}.

- To design the single-decade adder, a truth table for the output functions can be considered in which the desired sum digit and o/p carry are given for each possible pair of input digits & input carry.

- Truth table has $2^9 = 512$ rows. However since each of the decimal digits has only 10 code groups & the carry from the previous decade is only 0 or 1, it follows the o/p variables have specified values for only 200 of 512 rows.

This extremely a large table, so an alternate approach should be considered

The 8421 weighted coding scheme is the most commonly occuring in digital systems, it is referred to as BCD for binary-coded decimal

When using BCD, A single-decade decimal adder can be constructed by performing conventional binary addition on the two

binary-coded operands & then applying a corrective procedure

- The code groups for the two decimal digits are added using 4-bit binary adder to produce intermediate results of $K P_3 P_2 P_1 P_0$.

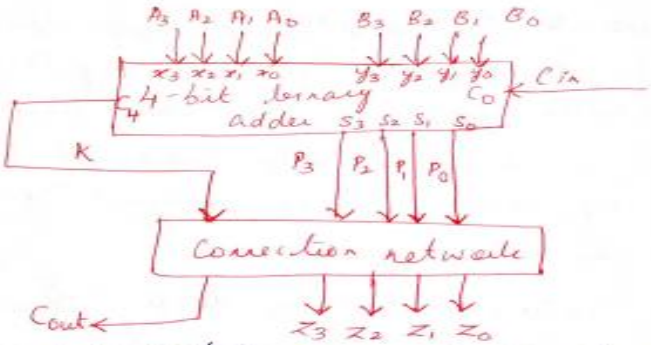


Figure - Organization of a single-decade BCD adder

These results are modified, inorder to obtain appropriate O/p carry + code group for the sum digit, ie Cout $Z_3 Z_2 Z_1 Z_0$

Since, each operand digit has a decimal value from 0 to 9, when a carry from a previous digit position is at most 1, the decimal sum at each digit position must be in the range from 0 to 19.

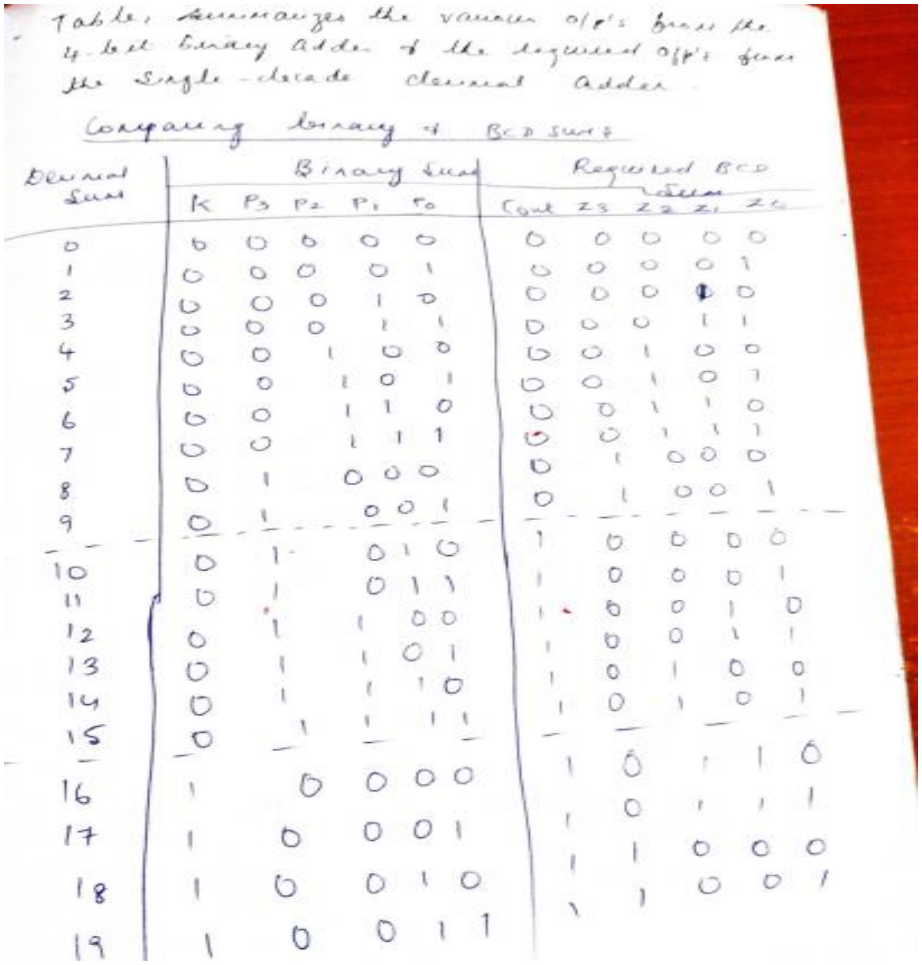


Table summarizes the various o/p's from the 4-bit binary adder & the required o/p's from the single-decade decimal adder

Comparing binary & BCD sums

Decimal Sum	Binary Sum					Required BCD Sum				
	K	P_3	P_2	P_1	P_0	Cout	Z_3	Z_2	Z_1	Z_0
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
2	0	0	0	1	0	0	0	0	1	0
3	0	0	0	1	1	0	0	0	1	1
4	0	0	1	0	0	0	0	1	0	0
5	0	0	1	0	1	0	0	1	0	1
6	0	0	1	1	0	0	0	1	1	0
7	0	0	1	1	1	0	0	1	1	1
8	0	1	0	0	0	0	1	0	0	0
9	0	1	0	0	1	0	1	0	0	1
10	0	1	0	1	0	1	0	0	0	0
11	0	1	0	1	1	1	0	0	0	1
12	0	1	1	0	0	1	0	0	1	0
13	0	1	1	0	1	1	0	0	1	1
14	0	1	1	1	0	1	0	1	0	0
15	0	1	1	1	1	1	0	1	0	1
16	1	0	0	0	0	1	0	1	1	0
17	1	0	0	0	1	1	0	1	1	1
18	1	0	0	1	0	1	1	0	0	0
19	1	0	0	1	1	1	1	0	0	1

Q4. b) i) P=Σ(1,4,8,13),Q=Σ(2,7,13,14) , Using **3:8 decoder**, connect OR gates to respective outputs.

To implement the functions P and Q using a **3:8 decoder**, we first observe that the minterms (such as 8, 13, and 14) require 4 input variables (A, B, C, D). A single 3:8 decoder only handles 3 variables (minterms 0–7). Therefore, we must use **two 3:8 decoders** configured as a **4:16 decoder** to cover the required range.

➞ Step 1: Construct a 4:16 Decoder

To decode 4 variables (A, B, C, D where A is the MSB), connect the lower three bits (B, C, D) to the address inputs of both 3:8 decoders. Use the MSB (A) to enable the decoders:

1. Connect $\bar{A}$ to the Enable pin of the **first decoder** (handles minterms 0–7).
2. Connect A to the Enable pin of the **second decoder** (handles minterms 8–15).

➞ Step 2: Map Minterms to Decoder Outputs

Identify the specific output pins for each function based on the binary representation of the minterms:

- **Decoder 1 (A=0):** Outputs Y_0 through Y_7.
- **Decoder 2 (A=1):** Outputs Y_8 through Y_{15} (internally mapped as pins 0–7 of the second decoder).

➞ Step 3: Connect OR Gates

Combine the required minterm outputs using two separate OR gates:

1. **For $P = \Sigma(1, 4, 8, 13)$:**
   Connect outputs Y_1 and Y_4 from Decoder 1, and Y_8 and Y_{13} from Decoder 2 to the inputs of a **4-input OR gate**.
2. **For $Q = \Sigma(2, 7, 13, 14)$:**
   Connect outputs Y_2 and Y_7 from Decoder 1, and Y_{13} and Y_{14} from Decoder 2 to the inputs of another **4-input OR gate**.

✓✓ Answer:

The logic implementation is:

$$P = Y_1 + Y_4 + Y_8 + Y_{13}$$

$$Q = Y_2 + Y_7 + Y_{13} + Y_{14}$$

Where Y_n represents the n^{th} output of the combined decoder circuit. For P, the OR gate is connected to pins **1, 4, 8, and 13**. For Q, the OR gate is connected to pins **2, 7, 13, and 14**.

ii) A=π(0,1,3,5)⇒Σ(2,4,6,7) B=Σ(1,4,5,7) , use decoder and OR corresponding minterms.

To implement the functions A and B, use a **3-to-8 line decoder** where the outputs for A are the OR sum of minterms $\{2, 4, 6, 7\}$ and the outputs for B are the OR sum of minterms $\{1, 4, 5, 7\}$.

➞ Step 1: Identify Decoder Size and Inputs

The functions involve minterms ranging from 0 to 7. This requires a **3-to-8 line decoder** ($2^3 = 8$ outputs). Let the three input variables be X, Y, and Z, representing the binary weight of the minterms. The decoder will generate eight outputs, D_0 through D_7, where each output D_i corresponds to minterm m_i.

➞ Step 2: Determine Minterms for Each Function

A Boolean function is implemented by ORing the decoder outputs corresponding to its minterms.

- **For Function A:** The problem provides the maxterm list $\Pi(0, 1, 3, 5)$. The minterm list consists of the remaining indices in the 3-variable truth table:
  $A(X, Y, Z) = \sum(2, 4, 6, 7)$
- **For Function B:** The minterm list is explicitly given:
  $B(X, Y, Z) = \sum(1, 4, 5, 7)$

➞ Step 3: Connect Decoder Outputs to OR Gates

To complete the circuit, connect the specific decoder outputs to the inputs of two separate OR gates:

1. **OR Gate 1 (Function A):** Connect outputs D_2, D_4, D_6, and D_7.
2. **OR Gate 2 (Function B):** Connect outputs D_1, D_4, D_5, and D_7.

✓ Answer:

The logical expressions for the implementation using the decoder outputs D_i are:

$$A = D_2 + D_4 + D_6 + D_7$$

$$B = D_1 + D_4 + D_5 + D_7$$

The implementation requires one **3-to-8 line decoder** and two **4-input OR gates**.

MODULE 3

Q5. a) Master-Slave JK Flip-Flop

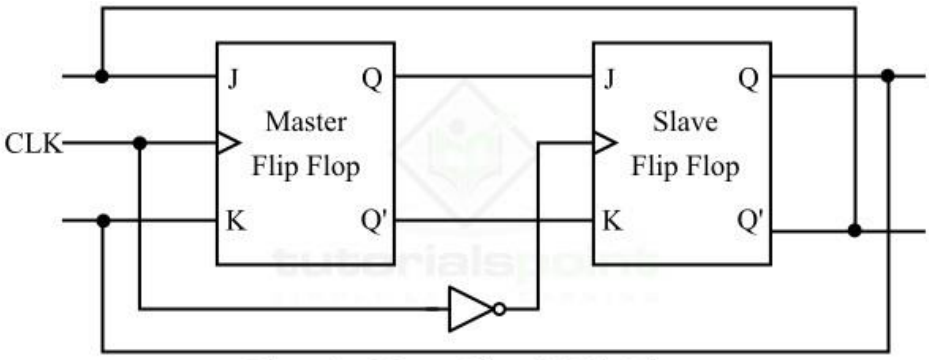


Figure 1 - Master-Slave JK Flip Flop

When J = K = 1 and the clock stays high, a JK flip-flop output toggles continuously, causing the race around condition. Using short clock pulses led to the Master-Slave JK Flip-Flop to ensure stable operation.

The Master-Slave JK Flip-Flop consists of two JK flip-flops in series, with one as the master and the other as the slave. The master output feeds the slave input, and the slave output is fed back to the master. An inverter provides the inverted clock to the slave, so only one flip-flop is active at a time. This ensures stable and predictable output.

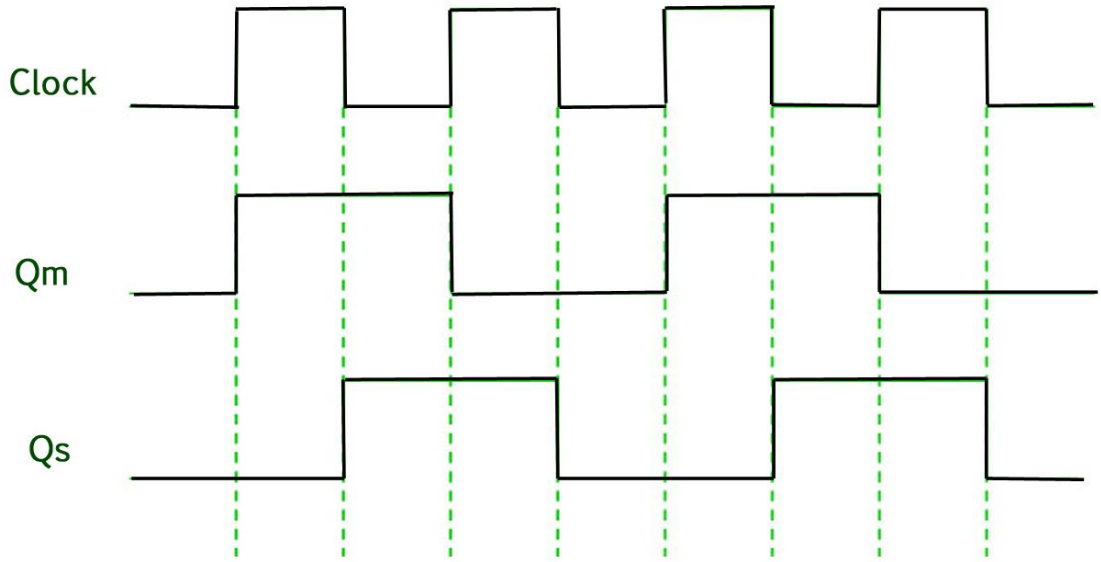
- The Master-Slave JK Flip-Flop is made of two JK flip-flops in series.
- One flip-flop acts as the master, the other as the slave.
- Master output → Slave inputs, and slave output → Master inputs.
- An inverter provides the inverted clock pulse to the slave.
    1. If CP = 1 for master, CP = 0 for slave.
    2. If CP = 0 for master, CP = 1 for slave

1. When the clock pulse goes to 1, the slave is isolated; J and K inputs may affect the state of the system. The slave flip-flop is isolated until the CP goes to 0. When the CP goes back to 0, information is passed from the master flip-flop to the slave and output is obtained.
2. Firstly the master flip flop is positive level triggered and the slave flip flop is negative level triggered, so the master responds before the slave.
3. If J=0 and K=1, the high Q' output of the master goes to the K input of the slave and the clock forces the slave to reset, thus the slave copies the master.
4. If J=1 and K=0, the high Q output of the master goes to the J input of the slave and the Negative transition of the clock sets the slave, copying the master.
5. If J=1 and K=1, it toggles on the positive transition of the clock and thus the slave toggles on the negative transition of the clock.
6. If J=0 and K=0, the flip flop is disabled and Q remains unchanged.

The Master-Slave JK Flip-Flop is a memory element widely used in digital systems. If you want to dive deeper into digital logic and master the flip-flop mechanisms, the GATE CS Self-Paced

Course offers detailed explanations and examples to help you understand this important concept.

Timing Diagram of a Master Slave flip flop -



1. When the Clock pulse is high the output of master is high and remains high till the clock is low because the state is stored.
2. Now the output of master becomes low when the clock pulse becomes high again and remains low until the clock becomes high again.
3. Thus toggling takes place for a clock cycle.
4. When the clock pulse is high, the master is operational but not the slave thus the output of the slave remains low till the clock remains high.
5. When the clock is low, the slave becomes operational and remains high until the clock again becomes low.
6. Toggling takes place during the whole process since the output is changing once in a cycle.

This makes the Master-Slave J-K flip flop a Synchronous device as it only passes data with the timing of the clock signal

Q5. b) Synchronous Mod-6 Counter (0→2→3→6→5→1→0)

State table:

PS	NS
000	010
010	011
011	110
110	101
101	001
001	000

Derive JK inputs using excitation table and K-maps → final logic equations.

A synchronous MOD-6 counter with the sequence 0→2→3→6→5→1→0 requires three flip-flops ($2^2 < 6 \leq 2^3$), with all clock inputs connected simultaneously. The design uses JK flip-flops, where the next state is determined by the current state (e.g., $000 \rightarrow 010 \rightarrow 011 \rightarrow 110 \rightarrow 101 \rightarrow 001 \rightarrow 000$), using K-maps to derive input expressions for J_i, K_i.

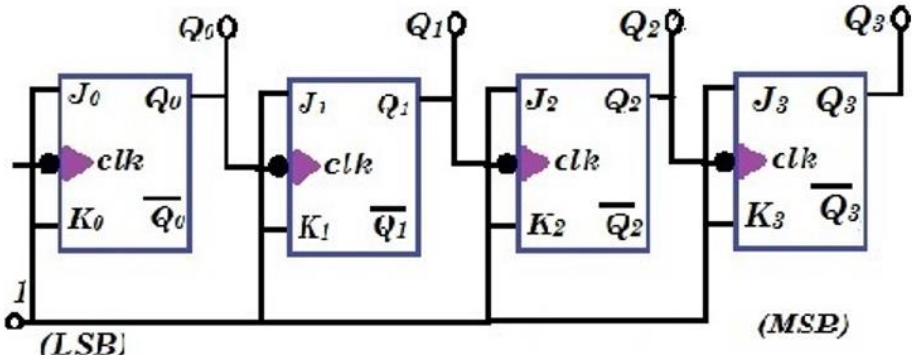
Design and Logic Steps

- **Flip-Flops Needed:** 3 (let's denote them Q_2, Q_1, Q_0).
- **State Mapping:** 0 (000), 1 (001), 2 (010), 3 (011), 5 (101), 6 (110).
- **Sequence:** $000 \rightarrow 010 \rightarrow 011 \rightarrow 110 \rightarrow 101 \rightarrow 001 \rightarrow 000$.
- **Logic Derivation:** Based on the desired sequence, the inputs J and K for each flip-flop are derived using the excitation table (e.g., $J_2 = Q_1Q_0$, $K_2 = Q_0$, etc.).
- **Circuit Operation:** A common clock drives all flip-flops, ensuring they change states synchronously based on the combinational logic feeding their inputs.
- **Unused States:** The states 100 (4) and 111 (7) are treated as "don't care" (X) conditions in the K-maps to minimize logic.

This specific sequence 0→2→3→6→5→1 is a custom, non-binary sequence often designed using specific mapping techniques to achieve the desired state transitions.

Q. 6A). 4-BIT RIPPLE COUNTER:

A 4-bit Binary Ripple Counter is a specialized digital counting mechanism comprising a chain of flip-flops, often of the J-K or D type. These flip-flops are arranged such that the output of one serves as the clock signal for its successor. In a typical 4-bit layout, four flip-flops enable the counter to cycle through binary numbers from 0000 to 1111 or 0 to 15 in decimal terms. Each flip-flop changes its state during its operation in response to an incoming clock pulse. Because every flip-flop is clocked by the output of its predecessor (except the first, which is triggered by an external clock), changes propagate—or "ripple"—from the least significant bit (LSB) to the most significant bit (MSB), which is the reason behind the term "ripple" counter.



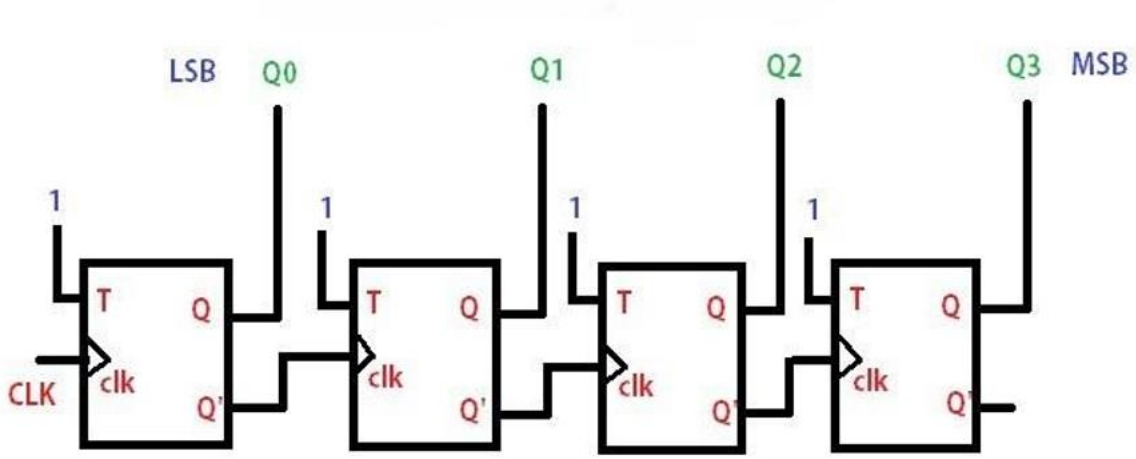
4-bit Binary Up/Down Counter:

A 4-bit Binary Up/Down Counter is a specialized digital tallying device designed to increase or decrease its numerical value. This is typically executed through an array of linked flip-flops, most commonly of the J-K variety. In its 4-bit form, the counter incorporates four such flip-flops, representing binary values from 0000 up to 1111, corresponding to the decimal range of 0 to 15. This counter's distinct feature is its bidirectional counting capability, governed by external inputs commonly marked as 'UP' and 'DOWN.' With the 'UP' input activated, the counter progresses from 0000 to 1111. On the other hand, when the 'DOWN' input is enabled and the 'UP' is not, it counts in reverse from 1111 back to 0000. This dual-mode operation renders it highly adaptable for scenarios demanding forward and reverse counting.

4-bit Binary Up Counter:

A 4-bit Binary Up Counter is a distinct digital counter engineered to increase its numerical value. It's typically built from an assembly of four linked flip-flops, commonly of either J-K or D variety. In its 4-bit configuration, the counter can represent binary figures spanning from 0000 to 1111, which equates to a decimal range of 0 through 15. Functioning on a 'UP' input directive, the counter escalates its count sequentially from 0000 to 1111. With its

exclusive orientation towards upward progression, it is optimally designed for use cases necessitating one-way, incremental tallying.



4-bit Binary Down Counter

A 4-bit Binary Down Counter is a specialized digital counting device designed explicitly to decrease its numeric count. Generally constructed using a series of four linked flip-flops, often of either the J-K or D type, this 4-bit counter can represent binary values ranging from 1111 down to 0000. This translates to a decimal span from 15 to 0. Activated by a 'DOWN' input, the counter descends through its sequence from 1111 to 0000. Its sole orientation towards decrementing makes it exceptionally suitable for scenarios that demand one-way, decremental tallying.

Q6. b) i) Ring Counter:

A ring counter is a typical application of the Shift register. The ring counter is almost the same as the shift counter. The only change is that the output of the last flip-flop is connected to the input of the first flip-flop in the case of the ring counter but in the case of the shift register it is taken as output. Except for this, all the other things are the same.

Ring Counter

It is a type of digital counter used in circuits, typically built with flip-flops. It works by shifting a 1 through a series of flip-flops, one at a time, in a loop. It’s called a ring counter because the 1 loops back to the start once it reaches the end, forming a cycle.

No. of states in Ring counter = No. of flip-flop used

Key Characteristics of a Ring Counter

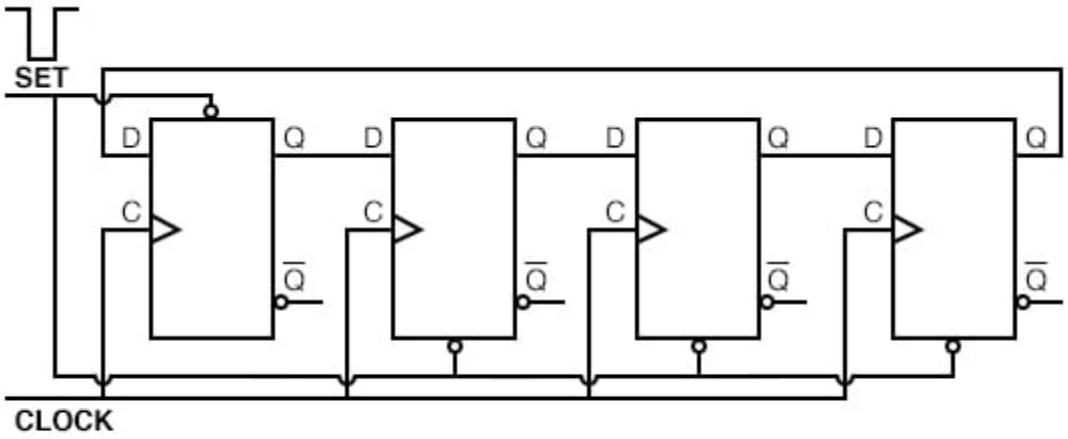
- **Single bit '1'**: Only one bit in the counter is set to 1 at any given time.
- **Cyclic Behavior**: The pattern of the bits follows a cyclic behavior and repeats after a fixed number of steps.
- **No Reset**: The ring counter doesn't automatically reset to zero. Instead, it starts at a defined state and then continues cycling.

Working Of Ring Counter

- A Ring Counter is typically built using flip-flops (D, T, or JK flip-flops).
- It consists of n flip-flops, where n is the number of states in the counter. The number of flip-flops determines the number of unique states the counter will cycle through.
- One bit in the counter is set to 1, and the rest of the bits are set to 0. In each clock cycle, the 1 bit shifts through the flip-flops, and the pattern repeats.
- The output of the last flip-flop is fed back into the first flip-flop, forming a loop, hence the name Ring Counter.

Example of a 4-bit Ring Counter

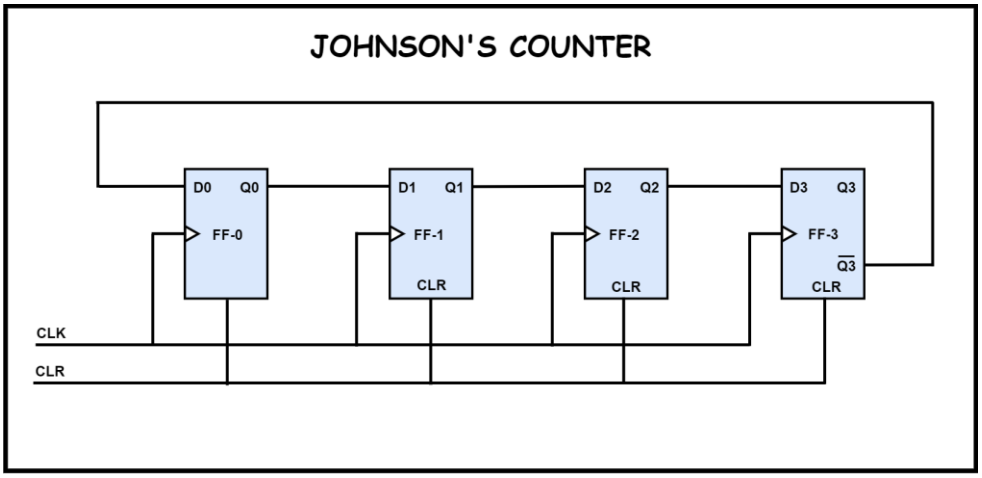
Let's consider a 4-bit Ring Counter: For 4-bit:
1000 → 0100 → 0010 → 0001 → 1000



Set one stage, clear three stages

ii) Mod-8 Twisted Ring (Johnson Counter)

Sequence:
0000 → 1000 → 1100 → 1110 → 1111 → 0111 → 0011 → 0001 → 0000



Module 4

7a) What are the different data typesin Verilog. Explain with examples

1.6.2.1 Nets

Nets are declared by the predefined word `wire`. Nets have values that change continuously by the circuits that are driving them. Verilog supports four values for nets, as shown in Table 1.13.

Examples of net types are as follows:

```
wire sum;
wire S1 = 1'b0;
```

TABLE 1.13 Verilog Net Values

Value	Definition
0	Logic 0 (false)
1	Logic 1 (true)
X	Unknown
Z	High impedance

1.6.2.2 Register

Register, in contrast to nets, stores values until they are updated. Register, as its name suggests, represents data-storage elements. Register is declared by the predefined word `reg`. Verilog supports four values for register, as shown in Table 1.14.

An example of register is:

```
reg Sum_total;
```

The above statement declares a register by the name `Sum_total`.

TABLE 1.14 Verilog Register Values

Value	Definition
0	Logic 0 (false)
1	Logic 1 (true)
X	Unknown
Z	High impedance

1.6.2.3 Vectors

Vectors are multiple bits. A register or a net can be declared as a vector. Vectors are declared by brackets `[ ]`. Examples of vectors are:

```
wire [3:0] a = 4'b1010;
reg [7:0] total = 8'd12;
```

1.6.2.4 Integers

Integers are declared by the predefined word `integer`. An example of integer declaration is:

```
integer no_bits;
```

The above statement declares `no_bits` as an integer.

1.6.2.4 Real

Real (floating-point) numbers are declared with the predefined word `real`. Examples of real values are 2.4, 56.3, and 5e12. The value 5e12 is equal to 5×10^{12}. The following statement declares the register weight as real:

```
real weight;
```

1.6.2.5 Parameter

Parameter represents a global constant. It is declared by the predefined word `parameter`. The following is an example of implementing parameters:

```
module compr_genr (X, Y, xgty, xlty, xeqy);
parameter N = 3;
input [N:0] X, Y;
output xgty, xlty, xeqy;
wire [N:0] sum, Yb;
```

To change the size of the inputs `x` and `y`, the size of the nets `sum`, and the size of net `Yb` to eight bits, the value of N is changed to seven as:

```
parameter N = 7
```

1.6.2.6 Arrays

Verilog, in contrast to VHDL, does not have a predefined word for array. Registers and integers can be written as arrays. Consider the following statements:

```
parameter N = 4;
parameter M = 3;
reg signed [M:0] carry [0:N];
```

The above statements declare an array by the name `carry`. The array `carry` has five elements, and each element is four bits. The four bits are in

7b i) Develop a Verilog program to implement 2x1 multiplexer using conditional operator. Also write truth table for 2x1 multiplexer

Solution

Assign Y = Conditional-expression ? true-expression : false-expression

```
module Mux2x1_conditional(input A,B,SEL,Gbar, output Y );
    assign Y = (Gbar) ? 1'b0 : (SEL & B ) | (~ SEL & A);
endmodule
```

TABLE 2.4 Truth Table for a 2x1 Multiplexer

Input		Output
SEL	**Gbar**	**Y**
X	H	L
L	L	A
H	L	B

iii) design a 2x1 multiplexer using data flow Verilog description draw a logic diagram and logic gate circuit for it

Solution :

EXAMPLE 2.3A **2x1 MULTIPLEXER WITH ACTIVE LOW ENABLE**

TABLE 2.4 Truth Table for a 2x1 Multiplexer

Input		Output
SEL	Gbar	Y
X	H	L
L	L	A
H	L	B

If the enable (Gbar) is high (1), the output is low (0) regardless of the input. When Gbar is low (0), the output is A if SEL is low (0), or the output is B if SEL is high (1). From Table 2.4, the Boolean function of the output Y is:

Y = (S1 and A and *SEL*) or (S1 and B and SEL); S1 is the invert of Gbar

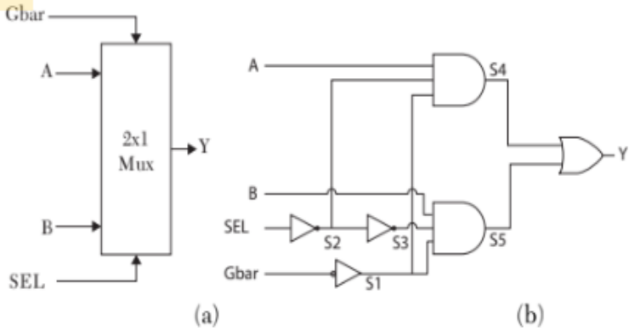


FIGURE 2.9 2x1 Multiplexer. a) Logic symbol. b) Logic diagram.

Verilog Description

```
module mux2x1 (A, B, SEL, Gbar, Y);
input A, B, SEL, Gbar;
output Y;
wire S1, S2, S3, S4, S5;
time dly = 7;
/* Assume 7 time units delay for all and, or, not operation
   delay here is expressedin simulation screen units. */

    assign # dly Y = S4 | S5; //st1
    assign #dly S4 = A & S2 & S1; //st2
    assign #dly S5 = B & S3 & S1; //st3
    assign #dly S2 = ~ SEL; //st4
    assign #dly S3 = ~ S2; //st5
    assign #dly S1 = ~ Gbar; //st6
endmodule
```

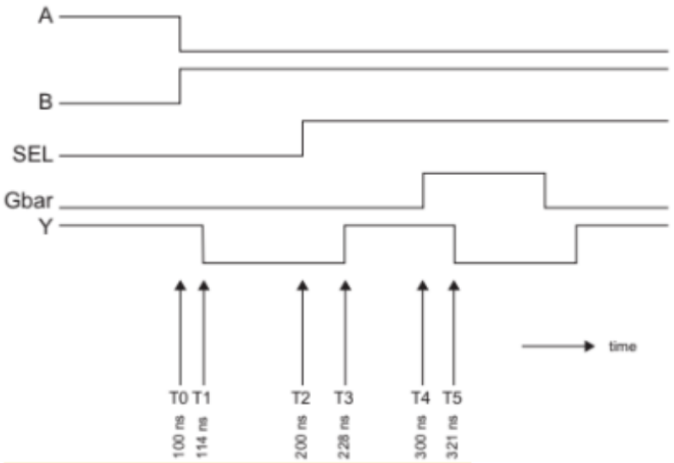


FIGURE 2.10 Simulation waveform for a 2x1 multiplexer.

8a) Discuss in detail different description styles in verilog

1.2.2 Structure of the Verilog Module

Verilog module has declaration and body. In the declaration, the name, inputs, and outputs of the module are entered. The body shows the relationship between the inputs and the outputs. Listing 1.2 shows a Verilog description of a half adder based on the Boolean function of the outputs.

Listing 1.2 Example of a Verilog Module

```
module Half_adder(a,b,S,C);
     input a,b;
     output S, C;
     // Blank lines are allowed

     assign S = a ^ b; // statement 1
     assign C= a & b; // statement 2
endmodule
```

module module_name (list_of_ports);

input/output declarations

local net declarations

Parallel statements

endmodule

1.3.1 Data Flow Description

Data flow describes how the system's signals flow from the inputs to the outputs. Usually, the description is done by writing the Boolean function of the outputs. The data-flow statements are concurrent; their execution is

Controlled by events

Listing 1.2 Example of a Verilog Module

```
module Half_adder(a,b,S,C);
     input a,b;
     output S, C;
     // Blank lines are allowed

     assign S = a ^ b; // statement 1
     assign C= a & b; // statement 2
endmodule
```

1.3.2 Behavioral Description

A behavioral description models the system as to how the outputs behave with the inputs; usually, a flowchart is used to show this behavior. In

Verilog Description

```
module Half_adder(a,b,S,C);
      input a,b;
      output S, C;
      reg S,C;
      // Blank lines are allowed
      always @ (a,b)
      begin
      if (a != b)
      S = 1'b1;
      else
      S = 1'b0;
      end
endmodule
```

1.3.3 Structural Description

Structural description models the system as components c

Verilog Description

```
module Half_adder1(a,b,S,C);
    input a, b;
    output S,C;
    and a1(C,a,b);
    //The above statement is AND gate
    xor x1(S,a,b);
    //The above statement is EXCLUSIVE-OR ga
endmodule
```

b. Let A = 5' b11011, B = 5' b 10101 C = 4' d3
Determine the output the following verilog statements.
i) d = & A ii) e = ~^ 4' b1011 iii) f = ~ (A & (~B))
iv) g= A| |B v) b = 3** 2 vi) i = {2{A}}

i) d= &A **(Reduction AND)**
Reduction AND means AND all bits of A:
A = 1 1 0 1 1
1 & 1 & 0 & 1 & 1 = 0

ii) **ii) e = A ~^ 4'b1011 (Bitwise XNOR)**

First match bit widths.
4'b1011 → zero-extended to 5 bits:

4'b1011 → 5'b01011

Now perform bitwise XNOR:

```
   11011
~^ 01011
  ---------
   01111
```

e = 5'b01111

iii) f = ~(A & (~B))

Step 1: Invert B

B = 10101
~B = 01010

Step 2: AND with A

```
   11011
&  01010
```

01010

Step 3: Invert result

~01010 = 10101

✅ **f = 5'b10101**

iv) g = A || B (Logical OR)

Logical OR checks if operands are non-zero.

- A ≠ 0
- B ≠ 0

So result is 1.

✅ **g = 1'b1**

---

v) b = 3 ** 2 (Exponentiation)

$3^2 = 9$

✅ **b = 9**

---

vi) i = {2{A}} (Replication Operator)

Repeat A two times:

A = 11011

i = 11011 11011

✅ **i = 10'b1101111011**

✅ **Final Answers**

Expression

d

e

f

g

b

i

Module – 5

9a) Design a 4 bit counter with synchronous hold using verilog. Also draw the simulation waveform.

Expression

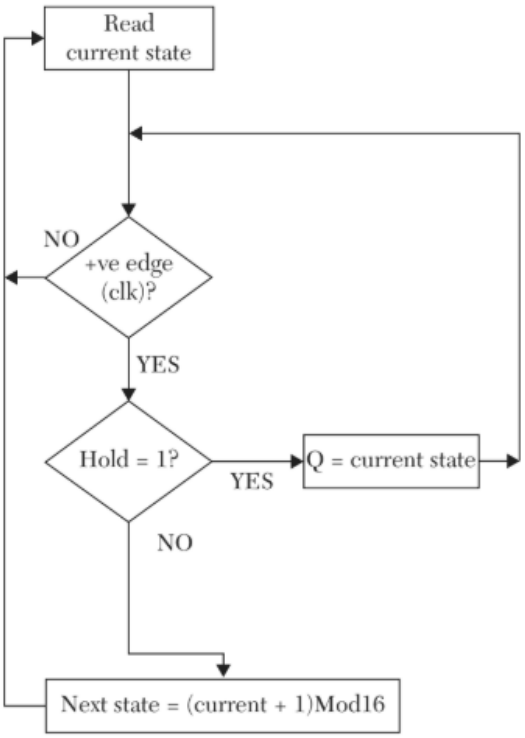


FIGURE 3.16 Flowchart of a four-bit counter with active high hold.

```
module CT_HOLD (clk, hold, q);
input clk, hold;
output [3:0] q;
reg [3:0] q;
integer i, result;
initial
begin
q = 4'b0000; //initialize the count to 0
end
always @ (posedge clk)
begin
result = 0;

//change binary to integer
```

9b)

b.	Explain the operation of positive edge triggered JK flip-flop by using a verilog code using case statement. Write truth table and timing diagram.

TABLE 3.2 Excitation Table of a Positive Edge-Triggered JK Flip-Flop

J	K	clk	q (next state)
0	0		No change (hold), next = current
1	0		1
0	1		0
1	1		Toggle (next state) = invert of (current state)
x	x	no +ve edge	No change (hold), next = current

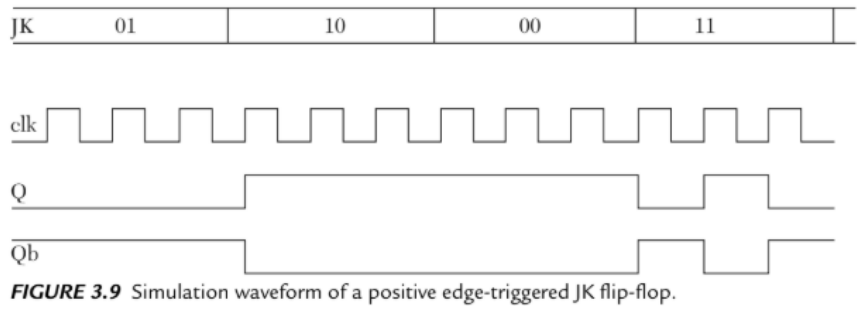


FIGURE 3.9 Simulation waveform of a positive edge-triggered JK flip-flop.

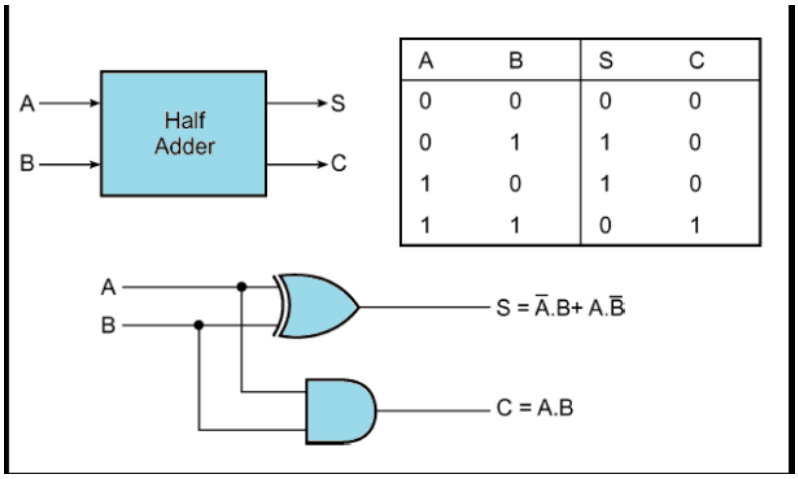
Verilog Description

```
module JK_FF (JK, clk, q, qb);
input [1:0] JK;
input clk;
output q, qb;
reg q, qb;
always @ (posedge clk)
begin
    case (JK)
    2'd0 : q = q;
    2'd1 : q = 0;
    2'd2 : q = 1;
    2'd3 : q =~ q;
    endcase
 qb =~ q;
 end

 endmodule
```

10a)

a.	Explain the operation of half adder and implement using structural description in verilog.



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

Verilog Description

```
module Half_adder1(a,b,S,C);
    input a, b;
    output S,C;
    and a1(C,a,b);
    //The above statement is AND gate
    xor x1(S,a,b);
    //The above statement is EXCLUSIVE-OR gate
endmodule
```

b.	Write the verilog format of case statement and explain.

Verilog Case Format

```
case (control-expression)
test value1 : begin statements1; end
test value2 : begin statements2; end
test value3 : begin statements3; end
default : begin default statements end
endcase
```

If, for example, `test value1` is true (i.e., it is equal to the value of the control expression), `statements1` is executed. The `case` statement must include all possible conditions (values) of the `control-expression`. The statement `when others` (VHDL) `or default` (Verilog) can be used to guarantee that all conditions are covered. The `case` resembles `IF` except the correct condition in `case` is determined directly, not serially as in `IF` statements. The `begin` and `end` are not needed in Verilog if only a single

statement is specified for a certain test value. The `case` statement can be used to describe data listed into tables.

In Example 3.7, the control is `sel`. If `sel` = 00, then `temp = I1`, if `sel` = 01, then `temp = I2`, if `sel` = 10, then `temp = I3`, if `sel` = 11 (others or default), then `temp = I4`. All four test values have the same priority; it means that if `sel` = 10, for example, then the third (VHDL) statement (`temp := I3`) is executed directly without checking the first and second expressions (00 and 01).

Verilog

```
case sel
2'b00 : temp = I1;
2'b01 : temp = I2;
2'b10 : temp = I3;
default : temp = I4;
endcase
```

c.	Develop a verilog behavioral description for 3 bit binary up counter.

// 3-bit Binary Up Counter - Behavioral Model

```
module up_counter_3bit (
   input wire clk,      // Clock input
   input wire reset_n,  // Active-low asynchronous reset
   output reg [2:0] q   // 3-bit counter output
);
 // Asynchronous reset, count increments on rising clock edge
   always @(posedge clk or negedge reset_n) begin
      if (!reset_n)
         q <= 3'b000;         // Reset counter to 0
      else
         q <= q + 1'b1;       // Increment counter
   end

endmodule
```

Testbench
Verilog

```
`timescale 1ns/1ps
module tb_up_counter_3bit;
   reg clk;
   reg reset_n;
   wire [2:0] q;

   // Instantiate the counter
   up_counter_3bit uut (
      .clk(clk),
      .reset_n(reset_n),
      .q(q)
   );

   // Clock generation: 10ns period
   initial begin
      clk = 0;
      forever #5 clk = ~clk;
   end

   // Test sequence
   initial begin
      // Initialize
      reset_n = 0;
      #12;             // Hold reset for some time
      reset_n = 1;      // Release reset
```

d.	Develop a verilog program for D latch using behavioral description style.

```
// D Latch with Enable and Asynchronous Reset
module d_latch (
   input  wire D,      // Data input
   input  wire EN,     // Enable signal
   input  wire RSTn,   // Active-low asynchronous reset
   output reg  Q       // Output
);
```

```
    // Behavioral description
    always @ (D or EN or RSTn) begin
        if (!RSTn)        // Reset active (low)
            Q <= 1'b0;
        else if (EN)      // Latch is transparent when EN=1
            Q <= D;
        // else: Q holds its previous value
    end

endmodule
```

Testbench for D Latch

Verilog

Copy code

```
`timescale 1ns/1ps
module tb_d_latch;
    reg D;
    reg EN;
    reg RSTn;
    wire Q;
    // Instantiate the D latch
    d_latch uut (
        .D(D),
        .EN(EN),
        .RSTn(RSTn),
        .Q(Q)
    );

    initial begin
        // Initialize signals
        D = 0; EN = 0; RSTn = 0;

        // Apply reset
        #5 RSTn = 1;  // Release reset

        // Test sequence
        #5 D = 1; EN = 1;  // Latch should capture D=1
        #5 EN = 0; D = 0;  // Output should hold previous value
        #5 EN = 1;        // Latch should capture D=0
        #5 D = 1;         // Latch should capture D=1
        #5 RSTn = 0;      // Reset output to 0
```

```
    #5 RSTn = 1; EN = 1; D = 1; // Capture again after reset

    #10 $finish;
end
```