eCAD & VLSI
LABORATORY MANUAL
B.TECH
(IV YEAR – I SEM)
(2019-20)

Prepared by:

Mr CH Kiran Kumar, Assistant Professor
Mr M Anantha Guptha, Assistant Professor

Department of Electronics and Communication Engineering

MALLA REDDY COLLEGE
OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
Recognized under 2(f) and 12 (B) of UGC ACT 1956
Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – ‘A’ Grade - ISO 9001:2015 Certified
Maisammaguda, Dhulapally (Post Via. Kompally), Secunderabad – 500100, Telangana State, India
ELECTRONICS & COMMUNICATION ENGINEERING
ELECTRONICS & COMMUNICATION ENGINEERING

VISION

To evolve into a center of excellence in Engineering Technology through creative and innovative practices in teaching-learning, promoting academic achievement & research excellence to produce internationally accepted competitive and world class professionals.

MISSION

To provide high quality academic programmes, training activities, research facilities and opportunities supported by continuous industry institute interaction aimed at employability, entrepreneurship, leadership and research aptitude among students.

QUALITY POLICY

- Impart up-to-date knowledge to the students in Electronics & Communication area to make them quality engineers.
- Make the students experience the applications on quality equipment and tools.
- Provide systems, resources and training opportunities to achieve continuous improvement.
- Maintain global standards in education, training and services.
PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEO1: PROFESSIONALISM & CITIZENSHIP
To create and sustain a community of learning in which students acquire knowledge and learn to apply it professionally with due consideration for ethical, ecological and economic issues.

PEO2: TECHNICAL ACCOMPLISHMENTS
To provide knowledge based services to satisfy the needs of society and the industry by providing hands on experience in various technologies in core field.

PEO3: INVENTION, INNOVATION AND CREATIVITY
To make the students to design, experiment, analyze, interpret in the core field with the help of other multi disciplinary concepts wherever applicable.

PEO4: PROFESSIONAL DEVELOPMENT
To educate the students to disseminate research findings with good soft skills and become a successful entrepreneur.

PEO5: HUMAN RESOURCE DEVELOPMENT
To graduate the students in building national capabilities in technology, education and research.

PROGRAMME SPECIFIC OBJECTIVES (PSOs)

PSO1
To develop a student community who acquire knowledge by ethical learning and fulfill the societal and industry needs in various technologies of core field.

PSO2
To nurture the students in designing, analyzing and interpreting required in research and development with exposure in multi disciplinary technologies in order to mould them as successful industry ready engineers/entrepreneurs

PSO3
To empower students with all round capabilities who will be useful in making nation strong in technology, education and research domains.
PROGRAM OUTCOMES (POs)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design / development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication**: Communicate effectively on complex engineering activities.
with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multi-disciplinary environments.

12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.
LABORATORY RULES

General Rules of Conduct in Laboratories:

1. You are expected to arrive on time and not depart before the end of a laboratory.
2. You must not enter a lab unless you have permission from a technician or lecturer.
3. You are expected to comply with instructions, written or oral, that the laboratory instructor gives you during the laboratory session.
4. You should behave in an orderly fashion always in the lab.
5. You must not stand on the stools or benches in the laboratory.
6. Keep the workbench tidy and do not place coats and bags on the benches.
7. You must ensure that at the end of the laboratory session all equipment used is stored away where you found it.
8. You must put all rubbish such as paper outside in the corridor bins. Broken components should be returned to the lab technician for safe disposal.
9. You must not remove test equipment, test leads or power cables from any lab without permission.
10. Eating, smoking and drinking in the laboratories are forbidden.
11. The use of mobile phones during laboratory sessions is forbidden.
12. The use of email or messaging software for personal communications during laboratory sessions is forbidden.
13. Playing computer games in laboratories is forbidden.

Specific Safety Rules for Laboratories:

1. You must not damage or tamper with the equipment or leads.
2. You should inspect laboratory equipment for visible damage before using it. If there is a problem with a piece of equipment, report it to the technician or lecturer. DONOT return equipment to a storage area.
3. You should not work on circuits where the supply voltage exceeds 40 volts without very specific approval from your lab supervisor. If you need to work on such circuits, you should contact your supervisor for approval and instruction on how to do this safely before commencing the work.
4. Always use an appropriate stand for holding your soldering iron.
5. Turn off your soldering iron if it is unlikely to be used for more than 10 minutes.
7. Never touch a soldering iron element or bit unless the iron has been disconnected from the mains and has had adequate time to cool down.
8. Never strip insulation from a wire with your teeth or a knife, always use an appropriate wire stripping tool.
9. Shield wire with your hands when cutting it with a pliers to prevent bits of wire flying about the bench.
**CYCLE-I**

<table>
<thead>
<tr>
<th>S.NO.</th>
<th>EXPERIMENT NAME</th>
<th>PAGE NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Full Adder</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Design Of Ripple Carry ADDER</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Design Of Carry Save Adder</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Design Of Carry Select Adder</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>BCD Adder Realization</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>Design of a 4 to 1 multiplexer</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Array Multiplier Realization</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>Ripple Counters Realization -(Mod -10 &amp; Mod-12)</td>
<td>26</td>
</tr>
<tr>
<td>9</td>
<td>Ring Counter Realization</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>Pseudo Random Binary Sequence Generator</td>
<td>39</td>
</tr>
<tr>
<td>11</td>
<td>Design of Accumulator</td>
<td>42</td>
</tr>
</tbody>
</table>

**CYCLE-II**

<table>
<thead>
<tr>
<th>S.NO.</th>
<th>EXPERIMENT NAME</th>
<th>PAGE NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CMOS INVERTER</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>NAND Gate</td>
<td>49</td>
</tr>
<tr>
<td>3</td>
<td>NOR Gate</td>
<td>53</td>
</tr>
<tr>
<td>4</td>
<td>XOR Gate</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>CMOS 1-Bit Full Adder</td>
<td>60</td>
</tr>
<tr>
<td>6</td>
<td>Common Source Amplifier</td>
<td>63</td>
</tr>
<tr>
<td>7</td>
<td>Differential Amplifier</td>
<td>67</td>
</tr>
</tbody>
</table>
EXPERIMENT: 1
FULL ADDER

AIM:
Realize the full adder using Verilog.

SOFTWARE REQUIRED:
QuestaSim Simulator

Theory:
A combinational circuit that performs the addition of three bits is called a half-adder. This circuit needs three binary inputs and produces two binary outputs. One of the input variables designates the augend and other designates the addend. Mostly, the third input represents the carry from the previous lower significant position. The output variables produce the sum and the carry.

The simplified Boolean functions of the two outputs can be obtained as below:
Sum \( S = x \oplus y \oplus z \)
Carry \( C = xy + xz + yz \)
Where \( x, y \) & \( z \) are the two input variables.

Program:

//Gate-level description of Full Adder using two Half Adder
module halfadder(s,co,x,y);
input x,y;
output s,co;
//Instatiate primitive gates
xor (s,x,y);
and (co,x,y);
endmodule

//Description of Full Adder
module fulladder(s,co,x,y,ci);
input x,y,ci;
output s,co;
wire s1,d1,d2; //Outputs of first XOR and AND gates
// Instantiate Half Adder
halfadder ha_1(s1,d1,x,y);
halfadder ha_2(s,d2,s1,ci);
or or_gate(co,d2,d1);
endmodule

//Stimulus for testing Full Adder module
simulation;
reg x,y,ci;
wire s,co;

//Instantiate Full Adder fulladder fa_test(s,co,x,y,ci);
Initial
begin
x=1'b0; y=1'b0; ci=1'b0;
#100 x=1'b0; y=1'b0; ci=1'b1; #100 x=1'b0; y=1'b0; ci=1'b0; #100 x=1'b0; y=1'b1; ci=1'b1; #100 x=1'b1; y=1'b0; ci=1'b0; #100 x=1'b1; y=1'b0; ci=1'b1; #100 x=1'b1; y=1'b1; ci=1'b0; #100 x=1'b1; y=1'b1; ci=1'b1;
end
endmodule

LOGIC DIAGRAM:

EXPECTED OUTPUT WAVEFORM:
SIMULATION OUTPUT WAVEFORM:

RESULT:

Thus the logic circuit for the Full adder is designed in Verilog HDL and the output is verified.
EXPERIMENT: 2
DESIGN OF RIPPLE CARRY ADDER USING VERILOG HDL

AIM:
To Design Ripple Carry Adder using Verilog HDL

SOFTWARE REQUIRED:
QuestaSim Simulator

THEORY:

The n-bit adder built from n one-bit full adders is known as ripple carry adder because of the carry is computed. The addition is not complete until \( n-1 \)th adder has computed its \( S_{n-1} \) output; that results depends upon \( c_i \) input, \( n \) and so on down the line, so the critical delay path goes from the 0-bit inputs up through \( c_i \)'s to the \( n-1 \) bit. (We can find the critical path through the n-bit adder without knowing the exact logic in the full adder because the delay through the n-bit adder without knowing the exact logic in the full adder because the delay through the n-bit carry chain is so much longer than the delay from \( a \) and \( b \) to \( s \)). The ripple-carry adder is area efficient and easy to design but it is when \( n \) is large. It can also be called as cascaded full adder.

The simplified Boolean functions of the two outputs can be obtained as below:

\[
\text{Sum } s_i = a_i \oplus b_i \oplus c_i \\
\text{Carry } c_{i+1} = a_i b_i + b_i c_i + a_i c_i \\
\text{Where } x, y \text{ & } z \text{ are the two input variables.}
\]

Procedure:

1) The ripple carry adder is designed.
2) The Verilog program source code for the circuit is written.
3) It is implemented in Model Sim and Simulated.
4) Signals are provided and Output Waveforms are viewed.

Circuit diagram:
RIPPLE CARRY ADDER USING VERILOG CODE:

```verilog
module ripplecarryadder(s,cout,a,b,cin);
output[7:0] s;
output cout;
input[7:0] a,b;
input cin;
wire c1,c2,c3,c4,c5,c6,c7;
fulladd fa0(s[0],c1,a[0],b[0],cin);
fulladd fa1(s[1],c2,a[1],b[1],c1);
fulladd fa2(s[2],c3,a[2],b[2],c2);
fulladd fa3(s[3],c4,a[3],b[3],c3);
fulladd fa4(s[4],c5,a[4],b[4],c4);
fulladd fa5(s[5],c6,a[5],b[5],c5);
fulladd fa6(s[6],c7,a[6],b[6],c6);
fulladd fa7(s[7],cout,a[7],b[7],c7);
endmodule

module fulladd(s,cout,a,b,cin);
output s,cout;
input a,b,cin;
wire s1,c1,c2;
xor(s1,a,b);
xor(s,s1,cin);
and(c1,a,b);
and(c2,s1,cin);
xor(cout,c2,c1);
endmodule
```

WAVEFORM OF RIPPLE CARRY ADDER:
TEST BENCH WAVE FORM OF RIPPLE CARRY ADDER:

RESULT:

Thus the logic circuit for the Ripple carry adder is designed in Verilog HDL and the output is verified.
EXPERIMENT: 3
DESIGN OF CARRY SAVE ADDER USING VERILOG HDL

AIM:
To design Carry Save Adder using Verilog HDL

SOFTWARE REQUIRED:
QuestaSim Simulator

THEORY:
Carry save adders are suitable when three or more operands are to be added, as in some multiplication schemes. In this adder a separate sum and carry bit is generated for partial results, except when the last operand is added. For example, when three numbers are added, the first two are added using a carry save adder. The partial result is two numbers corresponding to the sum and the carry. The last operand is added using a second carry save adder stage. The results become the sum and carry numbers. Thus a carry save adder reduces the number of operands by one for each adder stage. Finally the sum and carry are added using an adder with carry propagation- for example carry look ahead adder.

PROCEDURE:
1) The carry save adder is designed.
2) The Verilog program source code for the circuit is written.
3) It is implemented in Model Sim and Simulated.
4) Signals are provided and Output Waveforms are viewed.

CARRY SAVE ADDER USING VERILOG:

Module
carrysaveadder(d,a,b,e);
output [4:0]d;
input e;
input [3:0]a,b;
wire s1,s2,s3,c0,c1,c2,c3,c4,c5,c6,c7;
fulladder a1(d[0],c7,a[0],b[0],e);
fulladder a2(s3,c6,a[1],b[1],e);
fulladder a3(s2,c5,a[2],b[2],e);
fulladder a4(s1,c4,a[3],b[3],e);
fulladder a5(d[1],c3,c7,s3,e);
fulladder a6(d[2],c2,c6,c3,s2);
fulladder a7(d[3],c1,c5,s1,c2);
fulladder a8(d[4],c0,c4,c1,e);
endmodule
module
fulladder(s,c, x,y,z);
output s,c;
input x,y,z;
xor (s,x,y,z);
assign c = ((x & y )|(y & z)|( z & x)) ;
endmodule

LOGIC DIAGRAM:
WAVEFORM CARRY SAVE ADDER:

```
<table>
<thead>
<tr>
<th>Now: 1000 ns</th>
<th>0</th>
<th>200</th>
<th>400</th>
<th>600</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

TEST BENCH WAVEFORM CARRY-SAVE ADDER:

RESULT:
Thus the logic circuit for the carry save adder is designed in Verilog HDL and the output is verified.
EXPERIMENT: 4
DESIGN OF CARRY SELECT ADDER USING VERILOG HDL

AIM:
To design a Carry Select Adder using Verilog HDL

SOFTWARE REQUIRED:
QuestaSim Simulator

THEORY
Carry-select adders use multiple narrow adders to create fast wide adders. A carry-select adder provides two separate adders for the upper words, one for each possibility. A MUX is then used to select the valid result. Consider an 8-bit adder that is split into two 4-bit groups. The lower-order bits and are fed into the 4_bit adder l to produce the sum bits and a carry-out bit. the higher order bits and are used as input to one 4_bit adder and and \( \overline{Y_1} \overline{Y_0} Y_4 Y_8 \) are used as input of the another 4_bit adder. Adder U0 calculates the sum with a carry-in of \( C_3=0 \). while U1 does the same only it has a carry-in value of \( C_3=1 \). both sets of results are used as inputs to an array of 2:1 MUXes . the carry bit from the adder L is used as the MUX select signal. If \( =0 \) then the results \( U0 \) are sent to the output, while a value of \( =1 \) selects the results of U1 for \( S_{11} S_{10} S_9 S_8 \). The carry-out bit is also selected by the MUX array.

PROCEDURE:

1) The carry-select adder circuit is designed and the Boolean function is found out.
2) The Verilog Module Source for the circuit is written.
3) It is implemented in Model Sim and Simulated.
4) Signals are provided and Output Waveforms are viewed.
CARRY-SELECT ADDER USING VERILOG:

```verilog
module project2(s, m, x, y, z);
   output [0:3] s;
   output [1:5] m;
   input [0:11] x;
   input [0:11] y;
   input z;
   wire c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, s4, s5, s6, s7, s8, s9, s10, s11;

   fulladder f1(s[0], c0, x[0], y[0], z);
   fulladder f2(s[1], c1, x[1], y[1], c0);
   fulladder f3(s[2], c2, x[2], y[2], c1);
   fulladder f4(s[3], c3, x[3], y[3], c2);
   fulladder f5(s[4], c4, x[4], y[4], c3);
   fulladder f6(s[5], c5, x[5], y[5], c4);
   fulladder f7(s[6], c6, x[6], y[6], c5);
   fulladder f8(s[7], c7, x[7], y[7], c6);
   fulladder f9(s[8], c8, x[8], y[8], ~c3);
   fulladder f10(s[9], c9, x[9], y[9], c8);
   fulladder f11(s[10], c10, x[10], y[10], c9);
   fulladder f12(s[11], c11, x[11], y[11], c10);
   muxer mu1(m[1], s4, s8, c3);
   muxer mu2(m[2], s4, s8, c3);
   muxer mu3(m[3], s6, s10, c3);
   muxer mu4(m[4], s7, s11, c3);
   muxer mu5(m[5], c7, c11, c3);
endmodule

module fulladder (s, c, x, y, z);
   output s, c;
   input x, y, z;
   xor (s, x, y, z);
   assign c = ((x & y) | (y & z) | (z & x));
endmodule

module muxer (m, s1, s2, c);
   output m;
   input s1, s2, c;
   wire f, g, h; not (f, c);
   and (g, s1, c); and (h, s2, f);
   or (m, g, h);
endmodule
```
LOGIC DIAGRAM:

WAVEFORM OF CARRY-SELECT ADDER:
TEST BENCH WAVEFORM OF CARRY-SELECT ADDER:

RESULT:
Thus the logic circuit for the carry select adder is designed in Verilog HDL and the output is verified.
EXPERIMENT 5
BCD ADDER REALIZATION IN VERILOG HDL

AIM: To design a BCD adder circuit using Verilog HDL

SOFTWARE REQUIRED:
QuestaSim Simulator

THEORY:
A BCD adder is the circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. The input digit does not exceed 9, the output sum cannot greater than 9+9+1=19, the 1 in the sum being an input carry. Suppose we apply two decimal digits, together with the input carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output carry is equal to zero, nothing is added in the binary sum. When it is equal to one, binary 0110 is added to binary sum through the bottom 4-bit binary adder. Output generated from bottom binary adder can be ignored.

The output carry can be expressed in Boolean function

$$k = c_4 + s_3s_2 + s_3s_1$$

PROCEDURE:
1) The BCD adder circuit is designed and the Boolean function is found out.
2) The VHDL program source code for the circuit is written.
3) It is implemented in Model Sim and Simulated.
4) Signals are provided and Output Waveforms are viewed.

BCD ADDER USING VERILOG:

```vhdl
module bcdadder(s,k,a,b,c,d,e);
output [4:7] s;
inout k;
input [0:3]a,b;
input c,d,e;
wire c1,c2,c3,c4,s0,s1,s2,s3,e1,e2,e3,e4;
fulladder f1(s0,c1,a[0],b[0],c);
fulladder
```
f2(s1,c2,a[1],b[1],c1);
fulladder
f3(s2,c3,a[2],b[2],c2);
fulladder
f4(s3,c4,a[3],b[3],c3);
assign k=((s3 & s2) | (s3 & s1) | c4);
fulladder f5(s[4],e1,s0,d,e);
fulladder f6(s[5],e2,s1,k,e1);
fulladder f7(s[6],e3,s2,k,e2);
fulladder f8(s[7],e4,s3,d,e3);
endmodule
module fulladder(s,ca,a,b,c);
output s,ca;
input a,b,c; xor(s,a,b,c);
assign ca=((a & b)|(b & c) | (c & a));
endmodule

LOGIC DIAGRAM:
WAVEFORM:

<table>
<thead>
<tr>
<th>\text{load}</th>
<th>1001</th>
<th>110</th>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{load}</td>
<td>0110</td>
<td>1010</td>
<td>1001</td>
</tr>
<tr>
<td>\text{load}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{load}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{load}</td>
<td>0110</td>
<td>1100</td>
<td>1001</td>
</tr>
</tbody>
</table>

SIMULATION OUTPUT:
**SIMULATION OUTPUT OF BCD ADDER**

**RESULT:** The BCD adder circuit using Verilog HDL has been simulated.
EXPERIMENT: 6
DESIGN OF MULTIPLEXERS

AIM: Design a 4 to 1 Multiplexer circuit in Verilog.

SOFTWARE REQUIRED:
QuestaSim Simulator

THEORY:
A digital multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. Multiplexing means transmitting a large number of information units over a smaller number of channels or lines. The selection of a particular input line is controlled by a set of selection lines. Normally, there are $2^n$ input lines and $n$ selection lines whose bit combinations determine which input is selected. A multiplexer is also called a data selector, since it selects one of many inputs and steers the binary information to the output lines. Multiplexer ICs may have an enable input to control the operation of the unit. When the enable input is in a given binary state (the disable state), the outputs are disabled, and when it is in the other state (the enable state), the circuit functions as normal multiplexer. The enable input (sometimes called strobe) can be used to expand two or more multiplexer ICs to digital multiplexers with a larger number of inputs. The size of the multiplexer is specified by the number $2^n$ of its input lines and the single output line. In general, a $2^n$ – to – 1 line multiplexer is constructed from an $n$ – to – $2^n$ decoder by adding to it $2^n$ input lines, one to each AND gate. The outputs of the AND gates are applied to a single OR gate to provide the 1 – line output.

PROCEDURE:
1) The multiplexer circuit is designed and the Boolean function is found out.
2) The Verilog Module Source for the circuit is written.
3) It is implemented in Model Sim and Simulated.
4) Signals are provided and Output Waveforms are viewed.
TRUTH TABLE:

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>s[1]</td>
<td>s[0]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

LOGIC DIAGRAM 4 TO 1 MULTIPLEXER:

MULTIPLEXER USING VERILOG CODE:

```verbatim
module multiplexer(y,d,s);
output y;
input [3:0] d;
input [1:0] s;
wire a,b,c,e,f,g,h,i;
//Instantiate Primitive gates
not (a,s[1]);
```
not (b, s[0]);
and (c, d[0], b, a);
and (e, d[1], s[0], a);
and (f, d[2], b, s[1]);
and (g, d[3], s[0], s[1]);
or (h, c, e);
or (i, f, g);
or (y, h, i);
endmodule

//Stimulus for testing 4 to 1 Multiplexer

module simulation;
reg [3:0] d;
reg [1:0] s;
wire y;
//Instantiate the 4 to 1 Multiplexer
multiplexer mux_t(y, d, s);
initial begin
s=2'b00; d[0]=1'b1; d[1]= 1'b0; d[2]= 1'b0; d[3]= 1'b0; #100
s=2'b00; d[0]= 1'b0; d[1]= 1'b1; d[2]= 1'b1; d[3]= 1'b1; #100
s=2'b01; d[0]= 1'b0; d[1]= 1'b1; d[2]= 1'b0; d[3]= 1'b0; #100
s=2'b01; d[0]= 1'b1; d[1]= 1'b0; d[2]= 1'b1; d[3]= 1'b1; #100
s=2'b10; d[0]= 1'b0; d[1]= 1'b0; d[2]= 1'b1; d[3]= 1'b0; #100
s=2'b10; d[0]= 1'b1; d[1]= 1'b1; d[2]= 1'b0; d[3]= 1'b1; #100
s=2'b11; d[0]= 1'b0; d[1]= 1'b0; d[2]= 1'b0; d[3]= 1'b1; #100
s=2'b11; d[0]= 1'b1; d[1]= 1'b1; d[2]= 1'b1; d[3]= 1'b0;
end
endmodule
WAVEFORM OF MULTIPLEXERS:

RESULT: Thus the multiplexer is designed in Verilog HDL and the output is verified.
EXPERIMENT: 7
ARRAY MULTIPLIER REALIZATION IN VERILOG HDL

AIM: To design an array multiplier circuit for 4 inputs and 8 outputs using VHDL.

SOFTWARE REQUIRED: QuestaSim Simulator

THEORY:

Binary multiplication can be accomplished by several approaches. The approach presented here is realized entirely with combinational circuits. Such a circuit is called an array multiplier. The term array is used to describe the multiplier because the multiplier is organized as an array structure. Each row, called a partial product, is formed by a bit-by-bit multiplication of each operand. For example, a partial product is formed when each bit of operand ‘a’ is multiplied by b0, resulting in a3b0, a2b0, a1b0, a0b0. The binary multiplication table is identical to the AND truth table. Each product bit \( o(x) \), is formed by adding partial product columns. The product equations, including the carry-in \( c(x) \), from column \( c(x-1) \), are (the plus sign indicates addition not OR). Each product term, \( p(x) \), is formed by AND gates and collection of product terms needed for the multiplier. By adding appropriate \( p \) term outputs, the multiplier output equations are realized, as shown in figure.

4X 4 ARRAY MULTIPLIER:

\[
\begin{array}{cccc}
\text{a3} & \text{a2} & \text{a1} & \text{a0} \\
\text{b3} & \text{b2} & \text{b1} & \text{b0} \\
\text{a3b0} & \text{a2b0} & \text{a1b0} & \text{a0b0} \\
\text{a3b1} & \text{a2b1} & \text{a1b1} & \text{a0b1} \\
\text{a3b2} & \text{a2b2} & \text{a1b2} & \text{a0b2} \\
\text{a3b3} & \text{a2b3} & \text{a1b3} & \text{a0b3} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
o7 & o6 & o5 & o4 & o3 & o2 & o1 \\
\end{array}
\]

\[
\begin{align*}
a0b0 &= p0 & a1b2 &= p8 \\
a1b0 &= p1 & a0b3 &= p9 \\
a0b1 &= p2 & a3b1 &= p10 \\
a2b0 &= p3 & a2b2 &= p11 \\
a1b1 &= p4 & a1b3 &= p12
\end{align*}
\]
a0b2 = p5  a3b2 = p13
a3b0 = p6  a2b3 = p14
a2b1 = p7  a3b3 = p15

PROGRAM:
module mmmm(m,a,b);
input [3:0]a;
input [3:0]b;
output [7:0]m;
wire [15:0]p;
wire [12:1]s;
wire [12:1]c;
and(p[0],a[0],b[0]);
and(p[1],a[1],b[0]);
and(p[2],a[0],b[1]);
and(p[3],a[2],b[0]);
and(p[4],a[1],b[1]);
and(p[5],a[0],b[2]);
and(p[6],a[3],b[0]);
and(p[7],a[2],b[1]);
and(p[8],a[1],b[2]);
and(p[9],a[0],b[3]);
and(p[10],a[3],b[1]);
and(p[11],a[2],b[2]);
and(p[12],a[1],b[3]);
and(p[13],a[3],b[2]);
and(p[14],a[2],b[3]);
and(p[15],a[3],b[3]);
half ha1(s[1],c[1],p[1],p[2]);
half ha2(s[2],c[2],p[4],p[3]);
half ha3(s[3],c[3],p[7],p[6]);
full fa4(s[4],c[4],p[11],p[10],c[3]);
full fa5(s[5],c[5],p[14],p[13],c[4]);
full fa6(s[6],c[6],p[5],s[2],c[1]);
full fa7(s[7],c[7],p[8],s[3],c[2]);
full fa8(s[8],c[8],p[12],s[4],c[7]);
full fa9(s[9],c[9],p[9],s[7],c[6]);
half ha10(s[10],c[10],s[8],c[9]);
full fa11(s[11],c[11],s[5],c[8],c[10]);
full fa12(s[12],c[12],p[15],s[5],c[11]);
buf(m[0],p[0]);
buf(m[1],s[1]);
buf(m[2],s[6]);
buf(m[3],s[9]);
buf(m[4],s[10]);
buf(m[5],s[11]);
buf(m[6],s[12]);
buf(m[7],c[12]);
endmodule

module half(s,co,x,y);
input x,y;
output s,co;

// Instantiate primitive gates
xor (s,x,y);
and (co,x,y);
endmodule

// Description of Full Adder
module full(s,co,x,y,ci);
input x,y,ci;
output s,co;
wire s1,d1,d2; // Outputs of first XOR and AND gates
// Instantiate Half Adder
half ha_1(s1,d1,x,y);
half ha_2(s,d2,s1,ci);
or or_gate(co,d2,d1);
endmodule

LOGIC DIAGRAM:
WAVE FORM:

RESULT:
Thus an array multiplier circuit for 4 inputs and 8 outputs using VHDL is designed and the output is verified.
EXPERIMENT: 8

RIPPLE COUNTER REALIZATION IN VERILOG HDL

AIM: To realize an asynchronous ripple counter in Verilog

SOFTWARE REQUIRED:
QuestaSim Simulator

THEORY:

In a ripple counter, the flip-flop output transition serves as a source for triggering other flip-flops. In other words, the Clock Pulse inputs of all flip-flops (except the first) are triggered not by the incoming pulses, but rather by the transition that occurs in other flip-flops. A binary ripple counter consists of a series connection of complementing flip-flops (JK or T type), with the output of each flip-flop connected to the Clock Pulse input of the next higher-order flip-flop. The flip-flop holding the LSB receives the incoming count pulses. All J and K inputs are equal to 1. The small circle in the Clock Pulse /Count Pulse indicates that the flip-flop complements during a negative-going transition or when the output to which it is connected goes from 1 to 0. The flip-flops change one at a time in rapid succession, and the signal propagates through the counter in a ripple fashion. A binary counter with reverse count is called a binary down-counter. In binary down-counter, the binary count is decremented by 1 with every input count pulse.

PROCEDURE:

1) The 4 bit asynchronous ripple counter circuit is designed.
2) The Verilog Module Source for the circuit is written.
3) It is implemented in Model Sim and Simulated.
4) Signals are provided and Output Waveforms are viewed.

//Structural description of Ripple Counter

module ripplecounter(A0,A1,A2,A3,COUNT,RESET);
output A0,A1,A2,A3;
input COUNT,RESET;
//Instantiate Flip-Flop
FF F0(A0,COUNT,RESET);
FF F1(A1,A0,RESET);
FF F2(A2,A1,RESET);
FF F3(A3,A2,RESET);
edendmodule

//Description of Flip-Flop
module
FF(Q,CLK,RESET);
output Q;
input CLK,RESET; reg Q;
always @(negedge CLK or negedge RESET) if(~RESET)
Q=1'b0; else
Q=(~Q); endmodule

//Stimulus for testing Ripple Counter

module simulation;
reg COUNT;
reg RESET;
wire A0,A1,A2,A3;

//Instantiate Ripple Counter
ripplecounter rc_t(A0,A1,A2,A3,COUNT,RESET);
always
#5 COUNT=~COUNT;
Initial begin
COUNT=1'b0;
RESET=1'b0; #10 RESET=1'b1; end
endmodule

LOGIC DIAGRAM 4-BIT RIPPLE COUNTER:
TRUTH TABLE:

<table>
<thead>
<tr>
<th>COUNT</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

WAVEFORM OF RIPPLE COUNTER:
TESTBENCHWAVEFORM OF RIPPLE COUNTER:

<table>
<thead>
<tr>
<th>End Time: 1000 ns</th>
<th>100</th>
<th>300</th>
<th>500</th>
<th>700</th>
<th>900</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RESET</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LOGIC DIAGRAM MOD-10 RIPPLE COUNTER:
TRUTH TABLE:

<table>
<thead>
<tr>
<th>COUNT</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

//Structural description of MOD10 Counter module
MOD10(A0,A1,A2,A3,COUNT);
output A0,A1,A2,A3;
input COUNT;
wire RESET;
//Instantiate Flip-Flop
FF F0(A0,COUNT,RESET);
FF F1(A1,A0,RESET);
FF F2(A2,A1,RESET);
FF F3(A3,A2,RESET);
//Instantiate Primitive gate nand (RESET,A1,A3);
endmodule

//Description of Flip-Flop module FF(Q,CLK,RESET);
output Q;
input CLK,RESET;
reg Q=1'b0;
always @(negedge CLK or negedge RESET)
if(~RESET)
  Q=1'b0;
else
  Q=(~Q);
endmodule
//Stimulus for testing MOD10 Counter module simulation;
reg COUNT;
wire A0,A1,A2,A3;

//Instantiate MOD10 Counter
MOD10 MOD10_TEST(A0,A1,A2,A3,COUNT);
Always #10 COUNT=~COUNT;
begin
  COUNT=1'b0;
end
Endmodule

WAVEFORM OF MOD 10 COUNTERS:
TESTBENCH WAVEFORM OF MOD 10:

<table>
<thead>
<tr>
<th>End Time: 1000 ns</th>
<th>100</th>
<th>300</th>
<th>500</th>
<th>700</th>
<th>900</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

LOGIC DIAGRAM MOD-12 RIPPLE COUNTER:
TRUTH TABLE:

<table>
<thead>
<tr>
<th>COUNT</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

//Structural description of MOD12 Counter

Module
MOD12(A0,A1,A2,A3,COUNT);
output A0,A1,A2,A3;
input COUNT;
wire RESET;

//Instantiate Flip-Flop
FF F0(A0,COUNT,RESET);
FF F1(A1,A0,RESET);
FF F2(A2,A1,RESET);
FF F3(A3,A2,RESET);

//Instantiate Primitive gates
nand (RESET,A2,A3);

endmodule

//Description of Flip-Flop
module FF(Q,CLK,RESET);
output Q;
input CLK,RESET;
reg Q=1’b0;
always @(negedge CLK or negedge RESET) if(~RESET)
Q=1’b0;
else
Q=(~Q);
endmodule

//Stimulus for testing MOD12 Counter

module simulation;
reg COUNT;
wire A0,A1,A2,A3;
// Instanitate MOD12 Counter
MOD12 MOD12_TEST(A0,A1,A2,A3,COUNT);
always
#10 COUNT=~COUNT;
initial
begin
COUNT=1'b0;
end
endmodule

WAVEFORM OF MOD 12 COUNTERS:
TESTBENCH WAVEFORM OF MOD 12 COUNTERS:

RESULT:

Thus the ripple counter is designed in Verilog HDL and the output is verified.
EXPERIMENT: 9
RING COUNTER REALIZATION IN VERILOG HDL

AIM: To realize a ring counter in Verilog and VHDL.

SOFTWARE REQUIRED:
QuestaSim Simulator

THEORY:
A ring counter is a circular shift register with only one flip-flop being set at any particular time; all others are cleared. The single bit is shifted from one flip-flop to the other to produce the sequence of timing signals.

PROCEDURE:
1) The 4 bit ring counter circuit is designed.
2) The Verilog Module Source for the circuit is written.
3) It is implemented in Model Sim and Simulated.
4) Signals are provided and Output Waveforms are viewed.

BINARY RING COUNTER DESIGN IN VERILOG

module my_ringcntvlog (q,clk,reset);
output [0 : 3]q;
input clk,reset;
reg [0 : 3] q;
always @ (negedge clk or reset)
begin
  if (~reset)
    q = 4'b 1000;
  else if (reset)
    begin
      q[0] <= q[3];
      q[1] <= q[0];
      q[2] <= q[1];
      q[3] <= q[2];
    end
end
endmodule
LOGIC DIAGRAM:

![Logic Diagram]

TRUTH TABLE:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>clk</td>
<td>qa</td>
</tr>
<tr>
<td>reset</td>
<td>qb</td>
</tr>
<tr>
<td></td>
<td>qc</td>
</tr>
</tbody>
</table>

WAVEFORM OF RING COUNTER:

![Waveform Diagram]

TEST BENCH WAVEFORM OF RING COUNTER:

![Test Bench Waveform Diagram]
RESULT:

Thus the ring counter is designed in Verilog HDL and the output is verified.
EXPERIMENT: 10

PSEUDO RANDOM BINARY SEQUENCE GENERATOR

AIM: Realize the Pseudo random binary sequence generator in Verilog HDL

SOFTWARE REQUIRED:
QuestaSim Simulator

THEORY:
Random numbers for polynomial equations are generated by using the shift register circuit. The random number generator is nothing but the Linear Feedback Shift Register (LFSR). The shift registers are very helpful and versatile modules that facilitate the design of many sequential circuits whose design may otherwise appear very complex. In its simplest form, a shift register consists of a series of flip-flops having identical interconnection between two adjacent flip-flops. Two such registers are shift right registers and the shift left registers. In the shift right register, the bits stored in the flip-flops shift to the right when shift pulse is active. Like that, for a shift left register, the bits stored in the flip-flops shift left when shift pulse is active. In the shift registers, specific patterns are shifted through the register. There are applications where instead of specific patterns, random patterns are more important. Shift registers can also be built to generate such patterns, which are pseudorandom in nature. Called Linear Feedback Shift Registers (LFSR's), these are very useful for encoding and decoding the error control codes. LFSRs used as a generators of pseudorandom sequences have proved externally useful in the area of testing of VLSI chips.

CIRCUIT DIAGRAM:
```verilog
module psrno(y, clk);
output y;
input clk;
wire [1:0] q;
wire a;
dff df1(q[0], a, clk);
dff df2(q[1], q[0], clk);
dff df3(y, q[1], clk);
xor x1(a, y, q[1]);
endmodule

module dff(q, d, clk);
output q;
input d, clk;
reg q = 0;
always @(posedge clk)
begin
q = d;
end
endmodule

WAVEFORM OF PRBS:
```

![Waveform of PRBS](image-url)
TESTBENCHWAVEFORM OF PRBS:

<table>
<thead>
<tr>
<th>End Time: 1000 ns</th>
<th>100</th>
<th>300</th>
<th>500</th>
<th>700</th>
<th>900</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RESULT:

Thus the pseudo random binary generator is designed in Verilog HDL and the output is verified.
EXPERIMENT: 11

DESIGN OF ACCUMULATOR

AIM:
Realize the accumulator in Verilog HDL

SOFTWARE REQUIRED:
QuestaSim Simulator

THEORY:
An accumulator differs from a counter in the nature of the operands of the add and subtract operation:
• In a counter, the destination and first operand is a signal or variable and the other operand is a constant equal to 1: A <= A + 1.
• In an accumulator, the destination and first operand is a signal or variable, and the second operand is either:
  ♦ A signal or variable: A <= A + B
  ♦ A constant not equal to 1: A <= A + Constant

An inferred accumulator can be up, down or updown. For an updown accumulator, the accumulated data may differ between the up and down mode:

... if updown = '1' then a <= a + b;
else
    a <= a - c;

Program:

module accum (C, CLR, D, Q);
input C, CLR;
input [3:0] D;
output [3:0] Q;
reg [3:0] tmp;
always @(posedge C or posedge CLR) begin
    if (CLR)
        tmp = 4'b0000;
    else
        tmp = tmp + D;
end
endmodule
end
assign Q = tmp;
endmodule

LOGIC DIAGRAM

CIRCUIT DIAGRAM:

WAVE FORM OF ACCUMULATOR:

<table>
<thead>
<tr>
<th>Now: 1000 ns</th>
<th>0</th>
<th>200</th>
<th>400</th>
<th>600</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>4th</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PERIOD</td>
<td>3..</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DUTY_CYCLE</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OFFSET</td>
<td>3..</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLR</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4th</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TESTBENCH WAVEFORM OF ACCUMULATOR:

RESULT:
Thus the logic circuit for the Accumulator is designed in Verilog HDL and the output is verified.
CYCLE -II

EXPERIMENT: I

DESIGN AND IMPLEMENTATION OF AN INVERTER

AIM: To design and Implementation of an Inverter

TOOLS: Mentor Graphics: Pyxis Schematic, Pyxis Layout, Eldo, Ezwave, Calibre

THEORY:

The inverter is universally accepted as the most basic logic gate doing a Boolean operation on a single input variable. Fig.1 depicts the symbol, truth table and a general structure of a CMOS inverter. As shown, the simple structure consists of a combination of an pMOS transistor at the top and a nMOS transistor at the bottom. CMOS is also sometimes referred to as complementary-symmetry metal–oxide–semiconductor. The words "complementary-symmetry" refer to the fact that the typical digital design style with CMOS uses complementary and symmetrical pairs of p-type and n-type metal oxide semiconductor field effect transistors (MOSFETs) for logic functions. Two important characteristics of CMOS devices are high noise immunity and low static power consumption. Significant power is only drawn while the transistors in the CMOS device are switching between on and off states. Consequently, CMOS devices do not produce as much waste heat as other forms of logic, for example transistor-transistor logic (TTL) or NMOS logic, which uses all n-channel devices without p-channel devices.
**Schematic Capture:**

![Schematic Diagram]

**Procedure:**

1. Connect the Circuit as shown in the circuit diagram using Pyxis Schematic tool
2. Enter into Simulation mode.
3. Setup the Analysis and library.
4. Setup the required analysis.
5. Probe the required Voltages
6. Run the simulation.
7. Observe the waveforms in EZ wave.
9. Perform Routing using IRoute
10. Perform DRC, LVS, PEX.
Schematic Symbol:

Simulation Output: Input Vs Output

Transient and DC Characteristics:
Layout of the Inverter:
EXPERIMENT: II

NAND GATE

AIM: To create a library and build a schematic of a NAND GATE, to create a symbol for the Inverter, To build an Inverter Test circuit using your Inverter, To set up and run simulations on the Inverter Test design.

EDA Tool: Mentor Graphics

Schematic Diagram
PROCEDURE:

1. Connect the Circuit as shown in the circuit diagram using Pyxis schematic.
2. Create a simulation schematic for simulation.
3. Add necessary nets in outputs to view waveforms.
4. Run the Simulation and observe results in EZwave.
5. Draw the Layout for the circuit using Pyxis Layout.
6. Run the physical verification (DRC, LVS, PEX) using Calibre tool.
7. Run the post layout simulation by adding the .dsf file generated in PEX.
8. Observe the post layout results.

Symbol Creation:
Building the NAND Test Design

Creating a layout view of NAND gate
Simulation Output:
EXPERIMENT NO: III

NOR GATE

AIM: To design and simulate the CMOS NOR gate

TOOLS: Mentor Graphics: Pyxis Schematic, Pyxis Layout, Eldo, Ezwave, Calibre

CIRCUIT DIAGRAM:
SIMULATION CIRCUIT:

PROCEDURE:

1. Connect the Circuit as shown in the circuit diagram using Pyxis schematic.
2. Create a simulation schematic for simulation.
3. Add necessary nets in outputs to view waveforms.
4. Run the Simulation and observe results in EZwave.
5. Draw the Layout for the circuit using Pyxis Layout.
6. Run the physical verification (DRC, LVS, PEX) using Calibre tool.
7. Run the post layout simulation by adding the .dsf file generated in PEX.
8. Observe the post layout results.
Simulation Output:

![Simulation Output Image]

Layout:

![Layout Image]
EXPERIMENT NO: IV

XOR GATE

**AIM:** To create a library and build a schematic of an XOR gate, to create a symbol for the XOR, to build an inverter test circuit using your XOR, to set up and run simulations on the XOR_test design.

**EDA TOOLS:** pyxis schematic, pyxis layout, eldo, ezwave, calibre

**PROCEDURE:**
1. Connect the Circuit as shown in the circuit diagram using Pyxis schematic.
2. Create a simulation schematic for simulation.
3. Add necessary nets in outputs to view waveforms.
4. Run the Simulation and observe results in EZwave.
5. Draw the Layout for the circuit using Pyxis Layout.
7. Run the physical verification (DRC, LVS, PEX) using Calibre tool.
8. Run the post layout simulation by adding the .ds pf file generated in PEX.
9. Observe the post layout results.

Symbol Creation:

Building the XOR Gate Test Design
**PROCEDURE:**

1. Connect the Circuit as shown in the circuit diagram using Pyxis schematic.
2. Create a simulation schematic for simulation.
3. Add necessary nets in outputs to view waveforms.
4. Run the Simulation and observe results in EZwave.
5. Draw the Layout for the circuit using Pyxis Layout.
6. Run the physical verification (DRC, LVS, PEX) using Calibre tool.
7. Run the post layout simulation by adding the .dsfp file generated in PEX.
8. Observe the post layout results

**Simulation output:**

![Simulation output graph]
Layout:
EXPERIMENT NO: V

CMOS 1-BIT FULL ADDER

AIM: To design and simulate the CMOS 1 Bit Full Adder.

TOOLS: Pyxis Schematic, Pyxis Layout, Eldo, Ezwave, Calibre.

Schematic Diagram:

PROEDURE:

1. Connect the Circuit as shown in the circuit diagram using Pyxis schematic.
2. Create a simulation schematic for simulation.
3. Add necessary nets in outputs to view waveforms.
4. Run the Simulation and observe results in EZwave.
5. Draw the Layout for the circuit using Pyxis Layout.
6. Run the physical verification (DRC, LVS, PEX) using Calibre tool .
7. Run the post layout simulation by adding the .dsfp file generated in PEX.
8. Observe the post layout results
Testing the Full Adder:

Layout:
Simulation Output:
EXPERIMENT NO: VI

COMMON SOURCE AMPLIFIER

AIM: To design and simulate the Common Source Amplifier.

TOOLS: Pyxis Schematic, Pyxis Layout, Eldo, Ezwave, Calibre.

Circuit Diagram:
**Simulation Circuit:**

![Circuit Diagram]

**PROCEDURE:**

1. Connect the Circuit as shown in the circuit diagram using Pyxis schematic.
2. Create a simulation schematic for simulation.
3. Add necessary nets in outputs to view waveforms.
4. Run the Simulation and observe results in EZwave.
5. Draw the Layout for the circuit using Pyxis Layout.
6. Run the physical verification (DRC, LVS, PEX) using Calibre tool.
7. Run the post layout simulation by adding the .dsfp file generated in PEX.
8. Observe the post layout results.
AC Analysis:

Transient Analysis Result:
Layout:
EXPERIMENT NO VII

DIFFERENTIAL AMPLIFIER

AIM: To design and simulate the Differential Amplifier.

TOOLS: Pyxis Schematic, Pyxis Layout, Eldo, Ezwave, Calibre.

CIRCUIT DIAGRAM:
**Simulation Circuit:**

![Diagram of the simulation circuit]

**PROCEDURE:**

1. Connect the Circuit as shown in the circuit diagram using Pyxis schematic.
2. Create a simulation schematic for simulation.
3. Add necessary nets in outputs to view waveforms.
4. Run the Simulation and observe results in EZwave.
5. Draw the Layout for the circuit using Pyxis Layout.
6. Run the physical verification (DRC, LVS, PEX) using Calibre tool.
7. Run the post layout simulation by adding the .dspf file generated in PEX.
8. Observe the post layout results.
RESULTS:

AC Analysis result:

Transient Analysis Result: