Laboratory Journal

of

VERY LARGE SCALE INTEGRATION

For completion of term work of 8^{th} *semester*

curriculum program

Bachelor of Technology

In

ELECTRONICS AND TELECOMMUNICATION ENGINEERING

DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATION

ENGINEERING

Dr. BABASAHEB AMBEDKAR TECHNOLOGICAL UNIVERSITY

Lonere-402 103, Tal. Mangaon, Dist. Raigad (MS)

INDIA

INDEX

EXPERIMENT NO:1

Aim: Design Entry and Simulation of Combinational Logic Circuits

Software: Xillinx

Procedure:

- Double click the project navigator and select the option File-New project**.**
- Give the project name.
- Select Verilog module.
- Type your Verilog coding.
- Check for syntax.
- Select the new source of test bench waveform.
- Choose behavioral simulation and simulate it by Xilinx ISE simulator.
- Verify the output.

Verilog coding:

Logic gates:

AND GATE:

```
module gl(a,b,c);

input a;

input b;

output c;

and(c,a,b);

end module
```

OR GATE:

```
module gl(a,b,c);

input a;

input b;

output c;

or(c,a,b);
```

```
end module
```

XOR GATE:

```
 module gl(a,b,c);
input a;
input b;
output c;
xor (c,a,b);
end module
```

NAND GATE:

```
module gl(a,b,c);
input a;
input b;
output c;
nand(c,a,b);
end module
```

NOR GATE:

```
 module gl(a,b,c);
input a;
input b;
output c
nor(c,a,b);
end module
```

HALF ADDER:

```
module half adder(a ,b ,c ,s);
input a;
input b;
output c;
output s;
xor(s,a,b);
and(c, ~a, b);
endmodule
```

HALF SUBTRACTOR:

```
module half sub(a ,b, c, s);
input a;
input b;
output c;
output s;
xor(s,a,b);
and(c,~a,b);
end module
```

ENCODER:

```
module Encd2to4(i0, i1, i2, i3, out0, out1);
input i0,i1, i2, i3;
output out0, out1;
reg out0,out1;
always@(i0,i1,i2,i3)
case({i0,i1,i2,i3})
```

```
4'b1000:{out0,out1}=2'b00;
4'b0100:{out0,out1}=2'b01;
4'b0010:{out0,out1}=2'b10;
4'b0001:{out0,out1}=2'b11;
default: $display("Invalid");
endcase
endmodule
```

DECODER:

```
// Module Name: Decd2to4
module Decd2to4(i0, i1, out0, out1, out2, out3);
input i0, i1;
output out0, out1, out2, out3;
reg out0,out1,out2,out3;
always@(i0,i1)
case({i0,i1})
2'b00:
{out0,out1,out2,out3}=4'b1000;
2'b01:
{out0,out1,out2,out3}=4'b0100;
2'b10:
 {out0,out1,out2,out3}=4'b0010;
2'b11:
{out0,out1,out2,out3}=4'b0001;
default:
```

```
$display("Invalid");
endcase
endmodule
```

MULTIPLEXER:

```
// Module Name: Mux4to1
module Mux4to1(i0, i1, i2, i3, s0, s1, out);
input i0, i1, i2, i3, s0, s1;
output out;
wire s1n,s0n;
wire y0,y1,y2,y3;
not (s1n,s1);
not (s0n,s0);
and (y0,i0,s1n,s0n);
and (y1,i1,s1n,s0);
and (y2,i2,s1,s0n);
and (y3,i3,s1,s0);
or (out,y0,y1,y2,y3);
endmodule
```

DEMULTIPLEXER:

```
// Module Name: Dux1to4
module Dux1to4(in, s0, s1, out0, out1, out2, out3);
input in, s0, s1;
output out0, out1, out2,out3;
```

```
wire s0n,s1n;
not(s0n,s0);
not(s1n,s1);
and (out0,in,s1n,s0n);
and (out1,in,s1n,s0);
and (out2,in,s1,s0n);
and (out3,in,s1,s0);
endmodule
```

8 BIT ADDER :

```
module adder(a,b, s,c);
input [7:0] a,b;
output [7:0] s,c;
assign {c,s} = a + b;
endmodule
```

MULTIPLIER:

```
module multi(a,b, c);
input [3:0] a,b;
output [7:0] c;
assign c = a * b;
endmodule
```

RESULT:

Thus the program for study of simulation using tools and the output also verified successfully.

EXPERIMENT NO:2

Aim: To implement 4 bit parallel adder, subtractor using structural style. Simulate design and implement in Xilinx.

Theory:

1. Half adder:
   Half adder is a circuit of addition. We arbitrarily assign symbol x &y to the two i/p and for sum and for carry output. From truth table, we see that carry o/p is '0' unless both input are 1. The 's' output represents least significant bit. The carry o/p is 1 bit, both i/p are'1'.
2. Full adder:
   The full adder is combinational circuit that forms arithmetic sum of 3 i/p bits. It consist of 3i/p and 3 o/p. Two of its variable denoted x and y represents two significant bits to be added. The third bit to be added represents carry from lower significant bit position.
   The two output are represented by symbol 's' for sum and carry. The i/p and o/p logical relation of full adder circuit may be expressed as two Boolean function. Full adder can also be implemented with two half adder or one or gate.
3. Four bit adder/ subtractor:
   Four bit full adder- sutractor uses 4 full adder. The input carry is given to first full adder as Cin and for remaining the i/p as x with first carry bit, so when m=1, it acts as subtractor,when m=9,acts as adder.

Procedure:

1. Write verilog code for each module.
2. Implement 4 bit full adder-subtractor using full adder.
3. Check syntax and RTL schematic.
4. Check test bench waveform.
5. Observe simulation result.

Conclusion:

Hence, we implemented circuit in Xilinx using verilog module and simulated by creating test bench waveform and observed result.

Program :

Program for half adder using dataflow style

```
module half adder(a,b,s,c) ;
input a;
input b;
output s;
output c;
assign s=a^b;
assign c=a&b;
endmodule
```

Full adder using structural style

```
Module full adder(x,y,cin,sout,cout);
input x;
input y;
input cin;
output sout;
output cout;
wire w1,w2,w3;
half adder u1(x,y,w1,w2) ;
half adder u2(w1,cin,sout,w3);
or u3(cout,w2,w3);
endmodule
```

PARALLEL ADDER SUBSTRACTOR USING STRUCTURAL STYLE

```
module  addersub (p ,q ,r ,m ,n);

input [3:0] p, q;

input r;

output [3:0] m;

output n;

wire d1,d2,d3;

wire [3:0] b;

fulladder_1 u1(p[0],b[0],r, m[0],d1);

fulladder_1 u2(p[1],b[1],d1,m[1],d2);

fulladder_1 u3(p[2],b[2],d2,m[2],d3);

fulladder_1 u4(p[3],b[3],d3,m[3],n);

xor  u5(b[0],q[0],r);

xor  u6(b[1],q[1],r);

xor  u7(b[2],q[2],r);

xor  u8(b[3],q[3],r);

endmodule
```

EXPERIMENT NO:3

Aim:

1. Write a Verilog code for 4-bit ripple carry adder.
2. Simulate Verilog program for 4-bit carry look ahead adder.
3. Compare also utilization & speed of both adder.

Theory:

An 'n' bit binary adder can be constructed by using connecting a full adder is connected to carry o/p of first adder is connected to carry i/p of next full adder. The carry i/p has to simple through all stages before final sum & carry is produced.

$$\text{A} = A_3\ A_2\ A_1\ A_0$$

$$\frac{\text{B} = B_3\ B_2\ B_1\ B_0}{C_{out}\ S_3\ S_2\ S_1\ S_0}$$

Sum of 2 bits at given bit position depends on carry generated by addition of previous 2 bits. Thus, the sum of most significant bit is available only when carry signal has ripple through significant stage. The carry generated at first stages to most significant stage. The carry generated at first stage acts as i/p carry for next stage, corresponding carry generates another carry bit is obtained o/p.

Carry look ahead adder.

Carry look ahead adder are designed to overcome latency introduced by ripple effect of carry bit. The CLA also is based on origin of carry bit in equation.

$$C_{i+1} = a_i b_i + c_i(a_i \wedge b_i)$$

For the case that gives C_{i+1}=1. Since, either term may leave this o/p. It's $a_i b_i$=1 then a_{i+1} for generate term since i/p are viewed as generating carry out bit. If Q_i=1 then we must have a_i=b_i=1. The second term represents the class where an o/p carry C_i=1 may be propagate term.

$$P_i = a_i \wedge b_i$$

Sum bits given by,

$$S_i = P_i \wedge C_i \quad \text{, for every i}$$

Area utilization & speed of both address. If area of utilization of both 4-bit ripple carry adder & 4-bit carry look ahead adder are same but speed is different. For 4-bit carry look ahead adder speed is 9.606 ns & for ripple carry adder is 9.65 ns. The overall delay depends on the characteristics of full adder.

Conclusion:

Thus, we have studied & simulated program for 4-bit ripple carry adder & 4-bit carry look ahead adder.

Program for Ripple Carry adder

```
Module ripple (a , b ,cin ,s, cout);
Input [3:0]a, b;
Input cin;
Output [3:0]s;
Output cout ;
Wire [2:0]c;
 u1(a[0],b[0],cin[0],s[0],c[0]);
 u1(a[1],b[1],cin[0],s[1],c[1]);
 u1(a[2],b[2],cin[1],s[2],c[2]);
 u1(a[3],b[3],cin[2],s[3],cout);
endmodule
```

Program for carry look ahead adder

```
module clga (a ,b ,cin , s, cout);
input [3:0]a, b;
```

```
input cin;
output [3:0]s;
output cout;
wire[3:0]g;
wire[3:0]p;
assign g[0]=a[0]&b[0];
assign g[1]=a[1]&b[1];
assign g[2]=a[2]&b[2];
assign g[3]=a[3]&b[3];
assign p[0]=a[0]^b[0];
assign p[1]=a[1]^b[1];
assign p[2]=a[2]^b[2];
assign p[3]=a[3]^b[3];
assign c[0]=g[0]|(p[0]&cin);
assign c[1]=g[1]|(p[1]&g[0])|(p[1]&p[0]&cin);
assign c[2]=g[2]|(p[2]&g[1])|(p[1]&p[2]&g[0])|(p[0]&p[1]&p[2]&cin);
assign
cout=g[3]|(p[3]&g[2])|(p[2]&p[3]&g[1])|(p[1]&p[2]&p[3]&g[0])|(p[0]&p[1]&p[2]&p[3]&cin);
assign s[0]=a[0]^b[0]^cin;
assign s[1]=a[1]^b[1]^c[0];
assign s[2]=a[2]^b[2]^c[1];
assign s[3]=a[3]^b[3]^c[2];
endmodule
```

EXPERIMENT NO:4

Aim:

To design 1 bit ALU using with (1)2:1 MUX (2)4:1 MUX (3 full adder

Theory:

2:1 MUX :-

A 2:1 MUX has two input and one select line. Output depends on the state of select line. Table 1 shows output depending upon select line. Figure 1 shows the schematic diagram of 2:1 MUX. Output waveform shows the result .

Full Adder:

One bit full adder has 3 inputs and 2 output sum and carry.Table2 shows result of sum and carry depending upon input a, b, c .figure 2 shows schematic diagram of full adder

4:1 MUX

Using 2:1 MUX, 4:1 MUX designed.4:1 MUX has 4 inputs and 2 select lines. Depending upon status of select line, input is forwarded to output.Table 3 shows truth table of 4:1 MUX. Figure 3 shows schematic of4:1MUX,using 2:1MUX

One-bit Arithmetic and logic unit

Now with all three explained above, we have to implement 1-bit ALU.Table 4 shows operation performed by ALU depending on select line

Figure 4 shows schematic.

Procedure:

1. Write Verilog code for 2:1MUX
2. Write code for full adder using 2:1MUX
3. Write code for 4:1 MUX using 2:1MUX
4. Finally using all these components write code for 1-bit ALU
5. View schematic diagram
6. Observe test bench waveform

Conclusion:

Thus we have studied 1-bit ALU using full adder,2:1MUX and 4:1MUX in Verilog module and observed the result.

Code for 1 bit ALU

```
module  ALU1 (y, cout, a, b, cin ,s);
output cout, y;
input a, b, cin;
input [2:0] s;
wire t1,t2,t3,t4,t5,t6,t7,t8;
not (t1,b);
nand (t4, a, b);
nor (t5 ,a, b);
xor (t6 , a, b);
or (t7 ,a ,b);
supply0 gnd;
supply1 vdd;
mux421 p0(t2, gnd, vdd, b,t1,s[0],s[2]);
FA1 p1(a, t2,cin,t3,cout);
mux421 p2(t8,t4,t5,t6,t7,s[0],s[1]);
mux221 p3(y,t3,t8,s[2]);
endmodule
```

CODE FOR 4:1 MUX

```
Module mux 421(O,a0,a1,a2,a3,s0,s1);
Output O;
Input a0,a1,a2,a3;
Input s0,s1;
Assign O=~s0&~s1&a0|~s0&s1&a1|s0&~s1&a2|s0&s1&a3;
```

```
Endmodule
```

CODE FOR 2:1 MUX

```
Module mux 221(y,a0,a1,s);
Output y;
Input a0,a1,s;
Assign y=((~s)&a0)|(s&a1));
endmodule
```

CODE FOR FULL ADDER

```
Module fa1(a,b,c,sum,cy);
Input a,b,c;
Output sum,cy;
Assign sum=a^b^c;
Assign cy=a &b | b&c| a&c ;
endmodule
```

EXPERIMENT NO:5

Aim:

To implement the sequence detector using Melay Machine.

Theory:

A sequence detector is a sequential circuit which gives output one at the end of specific input logic sequence.

Melay Machine is defined as sequential network whose output is a function of both the present state and the input of network.

The state diagram for the Melay Machine has output associated with the transition between states, as shown in the diagram.

In a Melay Machine, it may be possible to represent both combination, using the same state and to compute the single bit directly from the input. Hence, less states are required.

The state diagram for sequence can be designed as shown in the figure alongside.

Melay model is useful for application where faster response is needed.

The state diagram is drawn as to generate output with overlapping sequence.

Procedure:

1. Define a Verilog module to detect the sequence.
2. Define input and output.
3. Using if statement specific the present & next state depending upon the input, as in the state diagram.
4. Depending upon the present state, specify the value of input as in state diagram.
5. View the schematic diagram.
6. Observe the test bench waveform.

Conclusion:

Thus, we have designed and implemented sequence detector using Melay Machine and developed a Verilog code.

PROGRAM:

```
%mealy 1101
Module zzzzzz1101(x,clk,rst,y);
Input x,clk,rst;
Output y;
        Localparam[1:0]a=0,b=1,c=2,d=3;
        Reg[1:0]state;
        always@(posedge clk)
        begin
        if(rst)
        state<=a;
        else case(state)
a:begin
if(x==0)
state<=a;
else state<=b;
end
b:begin
if(x==0)
state<=a;
else state<=c;
end
```

```
c:begin
if(x==0)
state<=d;
else state<=c;
end
d:begin
if(x==0)
state<=a;
else state<=b;
end
endcase
end
assign y=(state==d&&x==1)?1:0;
endmodule
```

EXPERIMENT NO:6

Aim:

Write code and description of sequence detector 0101 using Moore machine

Theory:

A sequence detector is a sequence machine which gives output at the end of specific input logic sequence. A Moore machine output depend on state .A Moore machine tends to use Moore number of state as compared to mealy machine.

Procedure:

1. Define Verilog model.
2. Define input and output.
3. Define output terminal as register.
4. Define 5 3-bit variables to denote present state and next state.
5. Assign unique binary values to different values.
6. Using 'if' sequence, specify the next state depending upon input as in state diagram.
7. Depending on present state specify value of output as in state diagram.
8. Create test bench waveform.
9. Simulate

Conclusion:

Thus we have designed and implemented 0101 sequence detector using moore machine in Verilog HDL and verified output for same.

PROGRAM:

```
%moore 1010

Module pattern 1101(x,clk,rst,y);

Input x,clk,rst;

Output y;

    Localparam[2:0]a=0,b=1,c=2,d=3,e=4;

    Reg[2:0]state;

    always@(posedge clk)
```

```
        begin
        if(rst)
        state<=a;
        else case(state)
a:begin
if(x==0)
state<=a;
else state<=b;
end
b:begin
if(x==0)
state<=c;
else state<=b;
end
c:begin
if(x==0)
state<=a;
else state<=d;
end
d:begin
if(x==0)
state<=e;
else state<=b;
end
```

```
e:begin
if(x==0)
state<=a;
else state<=d;
end
endcase
end
assign y=(state==d&&x==0)?1:0;
endmodule
```