

Laboratory Manual for
EC403 – CMOS VLSI Design

B. Tech.
SEM. IV (EC)



Department of Electronics & Communication
Faculty of Technology
Dharmsinh Desai University
Nadiad

TABLE OF CONTENTS

PART – 1 LABORATORY MANUAL

Sr No.	Title	Page No.
1.	Gate Level Modeling	1
2.	Data Flow Level Modeling.	3
3.	Combinational Circuit Using Conditional Operator	6
4.	(A) Behavioral Modeling	9
	(B) Implementation and Testing on Hardware Kit	16
5.	Blocking and Non Blocking Assignment	9
6.	Layout Generation & Design Rule Checking	21
7.	CMOS Inverter Characteristics	23
8.	Verification using Testbench	27
9.	Finite State Machines	30
10.	Projects	40

PART – 2 DATASHEETS

PART – 3 SESSIONAL QUESTION PAPERS

PART-4 APPENDIX

PART I

LABORATORY MANUAL

EXPERIMENT – 1

GATE LEVEL MODELING

OBJECTIVES:

- (i) To understand how to construct a Verilog description from the logic diagram of the circuit.
- (ii) Student must able to write a Verilog code for any combinational circuit using Gate Level Modeling

THEORY:

Verilog includes predefined modules that implement basic logic gates. These gates allow a circuit's structure to be described using gate instantiation statements of the form

gate_name [instance_name] (output_port,input_port);

The *gate_name* specifies the desired type of gate. It stands for one of the keywords *AND*, *NAND*, *OR*, *NOR*, *XOR*, *XNOR*. The instance name is any unique identifier. Each gate may have a different number of ports, with the output port listed first followed by a variable number of input ports.

Name Description Usage

AND	$f=(a.b)$	<code>and(f,,a,b)</code>
NAND	$f=(a.b)'$	<code>nand(f,a,b)</code>
OR	$f=(a+b)$	<code>or(f,a,b)</code>
NOR	$f=(a+b)'$	<code>nor(f,a,b)</code>
XOR	$f=(a \oplus b)$	<code>xor(f,a,b)</code>
XNOR	$f=(a \odot b)$	<code>xnor(f,a,b)</code>
NOT	$f=a'$	<code>not(f,a)</code>
BUF	$f=a$	<code>buf(f,a)</code>

SAMPLE PROGRAM:

```
module fulladder(c, a, b, sum, co);
input c, a, b;
output sum, co;
wire z1, z2, z3, z4;
and and1(z1,a,b);
and and2(z2,b,c);
and and3(z3,a,c);
```

```
or or1(co,z1,z2,z3);  
xor xor1(z4,x,y);  
xor xor2(sum,z4,c);  
endmodule
```

OBSERVATION:

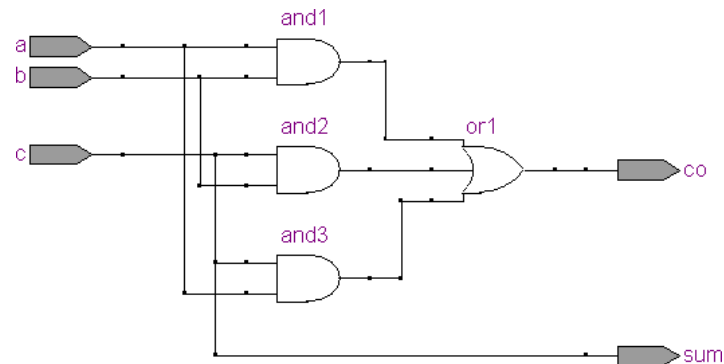


Fig.1.1 RTL View of Full-Adder using Gate-Level Modeling

CONCLUSION:

EXERCISES:

- (1) Design 2:1 Multiplexer and observe RTL viewer.
- (2) Design full adder using half adder

EXPERIMENT – 2

DATA FLOW LEVEL MODELING

OBJECTIVES:

- (i) Use data flow constructs to model practical digital circuits in Verilog.
- (ii) Describe the continuous assignment(assign) statement, restriction on the assign statement and the implicit continuous assignment statement
- (iii) Comparison between Gate level and data flow level modeling.
- (iv) Observe the difference between Functional and Time simulation.

THEORY:

For small circuits the gate level modeling approach works very well because the number of gates is limited. But in complex designs the number of gates is very large, so Verilog allows a circuit to be designed in terms of data flow between registers and how a design processes data rather than instantiation of individual gates.

The assignment is a most in dataflow modeling used to drive a value onto a net. the assignment statement starts with the keyword assign and the syntax of an assign statement is as follows.

assign net_assignment= (net_assignment, net_assignment);

The net_assignment can be any expression involving the operators as follow,

- 1. assign s = x^y^z;
- 2. assign Cout = (x&y) | (x & Cin) | (y & Cin);

Multiple assignments can be specified in one assign statement, using commas to separate the assignments.

- 3. assign Cout = (x&y) | (x & Cin) | (y & Cin),
s = x^y^z;
- a. An example of multibit assignment is
wire [1:3] A, B,C;
assign c=A & B;
This results in $c_1=a_1b_1$, $c_2=a_2b_2$ and $c_3=a_3b_3$.

Features:

- a. High level description,
- b. User friendly,
- c. Concise compare to Gate Level modeling
- d. Faster simulation speed (event driven),
- e. widely used for some common operations
- f. It contains expressions, operators and operands.

SAMPLE PROGRAM:

```
module full_adder (x, y, cin, cout, sum);  
input x;  
input y;  
input cin;  
output cout;
```

```

output sum;
assign cout=(x&y) | (x & cin) | (y & cin);
assign sum= x ^ y ^ cin;
endmodule

```

OBSERVATION:

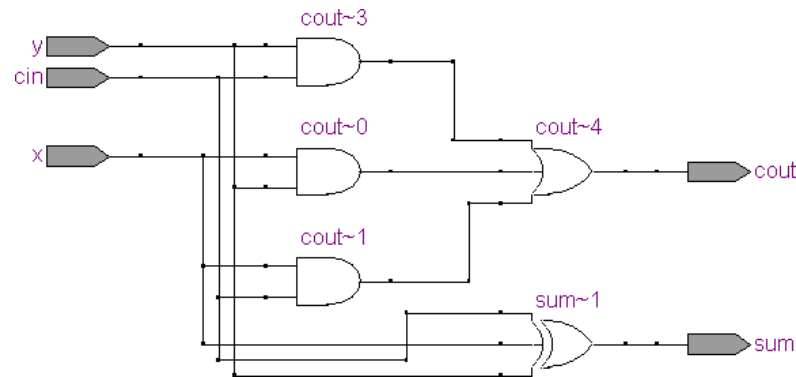


Fig.2.1 RTL View of Full-Adder using Data Flow Level Modeling

PROCEDURE: (to view simulation waveform)

- (1) After successful compilation we have to create a new vector waveform file as shown in fig. 2.2
- (2) Insert all the node or bus used in the design and design the input test signals pattern

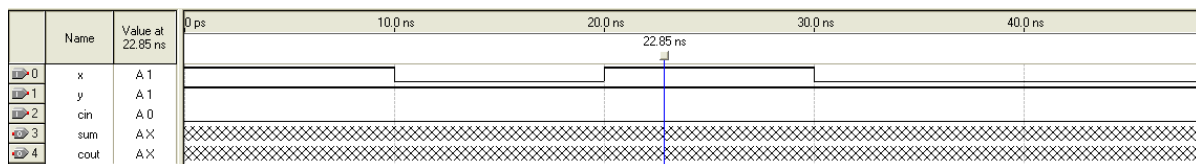


Fig.2.2 Vector Waveform File

- (3) Save the vector waveform file and choose the simulation mode as functional and generate the netlist the functional simulation netlist.
- (4) Start simulation and observe the simulation report(waveform) appears as shown in fig. 2.3

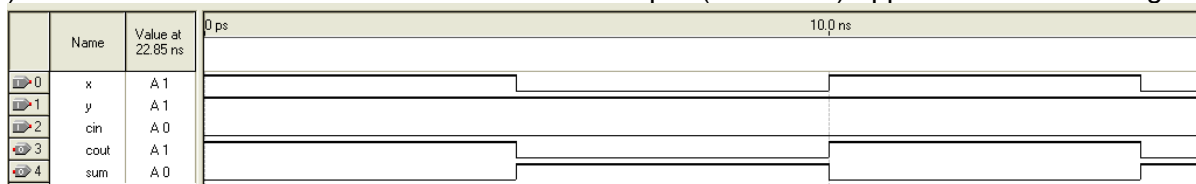


Fig. 2.3 Functional Simulation

- (5) Go again to simulator tool to set it as time mode and observe the simulation report after successful simulation
- (6) Observe the delay in the output which can be justified from the tpd file shown below. The tpd file is obtained from time analyzer option (compilation summary).

- (7) In particular cases we get glitch at output, it is the propagation time mismatch between two of the signals (x & Y to sum output). This mismatch is observed from tpd file as shown in Table 2.1

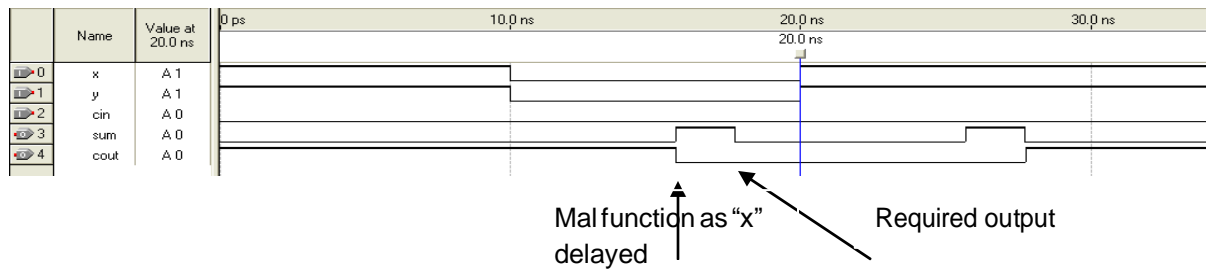


Fig. 2.4 Timing Simulation

Table 2.1 t_{pd} File Illustrates The Timing Between All The Ports.

tpd					
	Slack	Required P2P Time	Actual P2P Time	From	To
1	N/A	None	8.277 ns	cin	cout
2	N/A	None	8.260 ns	cin	sum
3	N/A	None	7.757 ns	x	cout
4	N/A	None	7.738 ns	x	sum
5	N/A	None	5.711 ns	y	cout
6	N/A	None	5.694 ns	y	sum

CONCLUSION:

EXERCISES:

- (1) Implement 4 to 1 multiplexer.
- (2) Implement 4 bit full adder using multi-bit assignment.

EXPERIMENT – 3

COMBINATIONAL CIRCUIT USING CONDITIONAL OPERATOR

OBJECTIVE:

- (i) To reduce the design complexity using conditional operator.

THEORY:

The conditional operator selects an expression for evaluation depending on the value of condition.

Simplified Syntax: `condition ? expression1 : expression2;`

If the condition is evaluated as false (or zero value) then expression2 is evaluated and used as a result of an entire expression. If condition is evaluated as true (or non-zero value) then expression1 is evaluated.

In case condition is evaluated as x or z value, then both expression1 and expression2 are evaluated, and the result is calculated bit by bit on the basis of table 3.1.

Table 3.1 Conditional Operator Execution

Exp1\exp2	0	1	x	z
0	0	X	x	x
1	x	1	x	x
x	x	X	x	x
z	x	x	x	x

If one of the expressions is of real type then the result of the whole expression should be 0 (zero). If expressions have different lengths, then length of an entire expression will be extended to the length of the longer expression. Trailing 0s will be added to the shorter expression.

The conditional operator can be nested and its behavior is identical with the case statement behavior. Conditional operator can be used for tri-state buffer modeling. Conditional operator can be nested (its behavior is identical with the case statement behavior).

Example 1 `(a) ? 4'b110x : 4'b1000;`

If 'a' has a non-zero value then the result of this expression is 4'b110x. If 'a' is 0, then the result of this expression is 4'b1000. If 'a' is x value then the result is 4'b1x0x (this is due to the fact that the result must be calculated bit by bit on the basis of the Table 1).

Example 2 `assign data_out = (enable) ? data_reg : 8'bz;`

This shows modeling of tri-state buffers.

SAMPLE PROGRAM 1:

4 ×1 MUX using data flow level modeling

```
module mux_cond(out,i0,i1,i2,i3,s1,s0);  
output out;  
input i0,i1,i2,i3;  
input s1,s0;  
assign out = (~s1 & ~s0 & i0 ) | (~s1 & s0 & i1) | (s1 & ~s0 & i2) | (s1 & s0 & i3);  
endmodule
```

OBSERVATION:

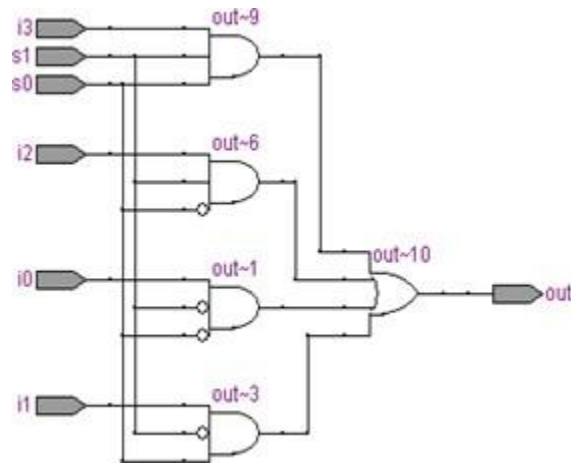


Fig. 3.1 RTL View of Full-Adder using Data Flow Level Modeling

Table 3.2 Propagation Delay Summary

mux_cond.v		Compilation Report - tpd				
Compilation Report		tpd				
Legal Notice		Slack	Required P2P Time	Actual P2P Time	From	To
Flow Summary		1	N/A	None	9.312 ns	i1
Flow Settings		2	N/A	None	8.789 ns	s1
Flow Non-Default Global Settin		3	N/A	None	8.161 ns	i2
Flow Elapsed Time		4	N/A	None	8.152 ns	i3
Flow OS Summary		5	N/A	None	8.079 ns	i0
Flow Log		6	N/A	None	5.655 ns	s0
Analysis & Synthesis						
Fitter						
Assembler						
Timing Analyzer						
Summary						
Settings						
Parallel Compilation						
tpd						
Messages						

SAMPLE PROGRAM 2:

4×1 MUX using conditional operator

```
module mux_cond(out,i0,i1,i2,i3,s1,s0);  
output out;  
input i0,i1,i2,i3;  
input s1,s0;  
assign out = (s1 & s0 & i3) | (s1 & ~s0 & i2) | (~s1 & s0 & i1) | (~s1 & ~s0 & i0);  
endmodule
```

```

input s1,s0;
assign out = s1 ? (s0? i3:i2) : (s0? i1:i0);
endmodule

```

OBSERVATION:

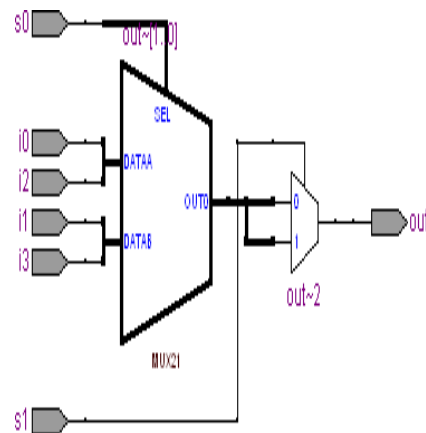


Fig. 3.2 RTL View of Full-Adder using Conditional Operator

Table 3.3 Propagation Delay Summary

Quartus II - c:/altera/91/quartus/mux_cond - mux_cond - [Compilation Report - tpd]						
Compilation Report - tpd						
	Slack	Required P2P Time	Actual P2P Time	From	To	
1	N/A	None	9.677 ns	s1	out	
2	N/A	None	8.777 ns	s0	out	
3	N/A	None	8.595 ns	i0	out	
4	N/A	None	8.455 ns	i3	out	
5	N/A	None	7.464 ns	i2	out	
6	N/A	None	6.648 ns	i1	out	

CONCLUSION:

EXERCISES:

- (1) Design 4:1 Multiplexer and 3 to 8 line Decoder using conditional operator and observe RTL viewer.
- (2) Understand each operator with their functionality (Refer page- 138 to 145 of Samir Palnitkar).

EXPERIMENT – 4 (A)

BEHAVIORAL MODELING

OBJECTIVES:

- (i) To model the sequential circuit logic using Behavioral modeling and brief awareness regarding synthesis operation.
- (ii) To understand the significance of structured procedures always and initial in Behavioral modeling.

THEORY:

Verilog provides designers the ability to describe design functionality in an algorithmic manner. In other words designer describes the Behavioral of the circuit. Thus Behavioral modeling represents the circuit at a very high level of abstraction. Verilog is rich at Behavioral constructs that provide the designer with a great amount of flexibility.

Verilog provides procedural statements also called sequential statements. Procedural statements are evaluated in the order in which they appear in the code. Verilog syntax requires that procedural statements be contained inside an always block. An always block is a construct that contains one or more procedural statements. It has the form as follow,

```
always @ (sensitivity_list)  
begin  
Procedural statements  
If-else statements.  
case statements  
end
```

When multiple statements are included in an always block, the begin and end keywords are needed. Otherwise keywords can be omitted. The sensitivity_list is a list of signals that directly affect the output results generated by the always block.

Example

```
always @ (x or y)  
begin  
S = x^y;  
C = x&y;  
end
```

Since the output variable s and c directly depends on x and y, the signals are included in the sensitivity list separated by keyword or.

Features

- a. Activity flows in Verilog is parallel rather than in sequence.
- b. Each always and initial statement represents a separate activity flow in Verilog.
- c. Each activity flow starts at simulation time 0.
- d. The statement always cannot be a nested.
- e. The always block has to be a variable of type reg or integer.

SAMPLE PROGRAM 1:

D Flip flop

```
module d_ff(clk,reset,d,out);  
input clk;  
input reset;  
input d;  
output out;  
reg out;  
always @(posedge clk or negedge reset)  
begin  
    if(~reset)  
        out<=0;  
    else  
        out <= d;  
    end  
end  
endmodule
```

OBSERVATION:

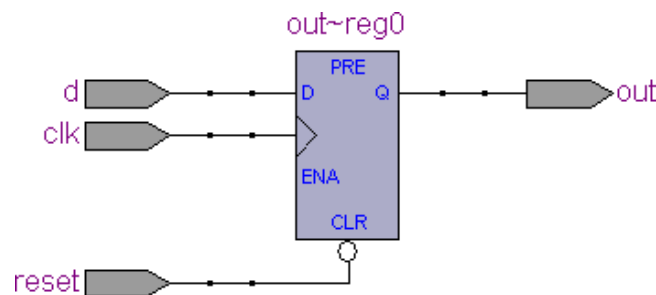


Fig.5.1 RTL View of D-FF using Behavioral Modeling

SAMPLE PROGRAM 2:

4 bit COUNTER

```
module counter(rst,count,clk);  
input clk,rst;  
output reg [3:0] count;  
always @(posedge clk)  
begin  
    if(rst)  
        count<=0;  
    else  
        count<=count+1;  
    end  
end  
endmodule
```

OBSERVATION:

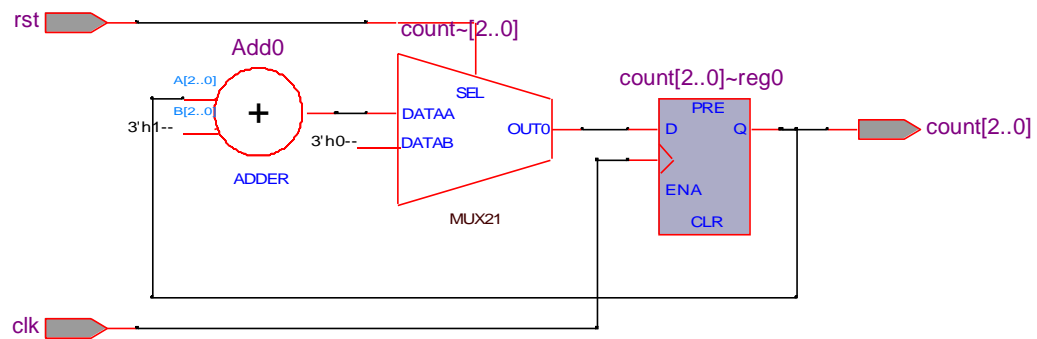


Fig. 5.2 RTL View of Counter using Behavioral Modeling

CONCLUSION:

EXERCISES:

- (1) Write a program to model JK flip flop.
- (2) Write a program to model 4 to 1 Multiplexer.

EXPERIMENT - 4 (B)

IMPLEMENTATION AND TESTING ON HARDWARE KIT

AIM: To implement, load and testing of digital circuits on hardware kit

OBJECTIVES:

1. Generate fully synthesizable code
2. Basic understandings of FPGA device families and main features to be looked for
3. To be familiar with basics and functionality of FPGA boards and kits
4. To be familiar with the all phases of design implementation on FPGA kits and possibilities of usage for different applications

THEORY:

Different FPGA device family features different number of pins, gate counts, speed, power consumption etc. One can choose by requirements of their design applications. Here the process is demonstrated for the Altera DE1 board for burning and testing of Verilog code.

PROCEDURE:

(1) Create a new project for counter design

Create a new project selecting device Cyclone II EP2C20F484C7. This device having features like 240 pin count and Speed grade 8 which is indicated in its nomenclature. Implement a Verilog code for counter and observe the simulation.

SAMPLE PROGRAM:

```
module cnt_kit (clk, reset, count);                // creating module  
  
input clk, reset;                                // IO declaration  
output [3:0] count;  
reg [3:0] count;  
  
always @ (posedge clk or posedge reset)           // sensitivity list  
begin  
if (reset == 1'b1)                                // asynchronous reset  
    count <= 0;  
else  
    count <= count + 1;  
end  
endmodule
```

(2) Create Block Diagram File for .v file

To design a block diagram file, the first step is designing a block symbol file for a current verilog hdl File. For that Go to File (menu)>New>Block Diagram as shown in **Fig. 9.1**. It will create a symbol file for current file. To update the same file at any point later one can select File (menu)>create/update> a symbol file for current file. It will show the successful message. Acknowledge it by click on OK.

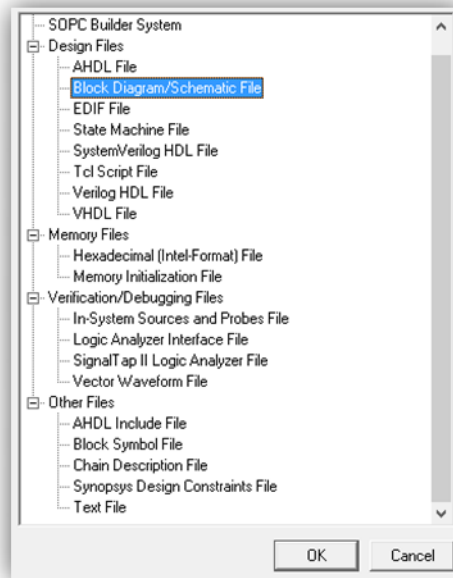


Fig. 9.1 Select block diagram file

(3) Now add the block of *cnt_kit*

For this file, double click anywhere in the file. So have new window symbol on Quartus II desktop. Expand the option project and select *cnt_kit* option. The display have window as shown in **Fig.9.2**. Click on OK to place this symbol in the Block Diagram File (.bdf).

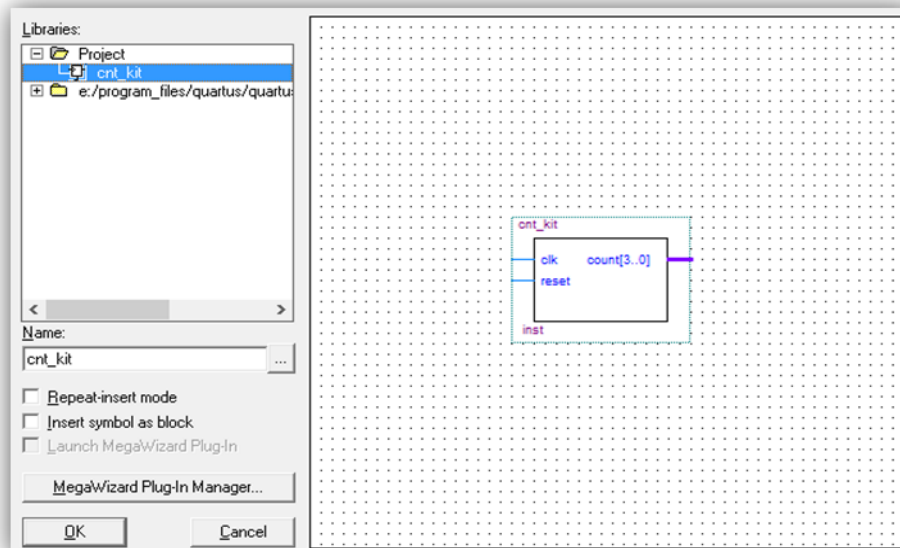


Fig. 9.2 Add symbol file

(4) Pin Assignment

Select the symbol Load and right click on it so you have number of options. Select Generate pins for symbol ports option as shown in **Fig. 9.3**.

Edit: <input checked="" type="checkbox"/> Yes								
	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reserved	Enabled
1	clk	PIN_R22	6	3.3-V LVTTTL	Row I/O	LVDS81p		Yes
2	count			3.3-V LVTTTL				Yes
3	count[0]	PIN_R20	6	3.3-V LVTTTL	Row I/O	VREFB6N0		Yes
4	count[1]	PIN_U19	6	3.3-V LVTTTL	Row I/O	LVDS89n		Yes
5	count[2]	PIN_Y19	6	3.3-V LVTTTL	Row I/O	LVDS90n		Yes
6	count[3]	PIN_T18	6	3.3-V LVTTTL	Row I/O	PLL4_OUTp		Yes
7	reset	PIN_R21	6	3.3-V LVTTTL	Row I/O	LVDS81n		Yes

Fig. 9.3 Pin Assignment

During the compilation above, the Quartus II Compiler was free to choose any pins on the selected FPGA to serve as inputs and outputs. **However, the DE1 board has hardwired connections between the FPGA pins and the other components on the board.** We will use two **PIN_R22** and **PIN_R21** for **clock and reset** respectively which assigns KEY[0] and KEY[1] switches from the board as inputs for the design. To observe output on LEDs available on the board labelled **LEDR[0] to LEDR[3]** to **PIN_R20, PIN_R19, PIN_U19 and PIN_Y19** for 4 bit output respectively, which is as shown in **Fig. 9.3**.

To assign these pin numbers follow the steps below.

- Select all input and output pins of a symbol *cnt_kit* in .bdf file.
- Right click on the selected pins and select Locate>Locate in Assign editor. The window by following these options is as shown in **Fig.9.3**.
- In Assignment editor window under editor tab select the location tab and give the pin number as mentioned above the port *clk*, *count* and *reset*. Click on save and close the Assignment Editor window.

(5) Setting .bdf as top level entity

Switch to block diagram file. Click on save and give the name *block1*. We want to compile the .bdf file so that select assignments (menu)>Settings. We have one new window as shown in **Fig. 9.4**.

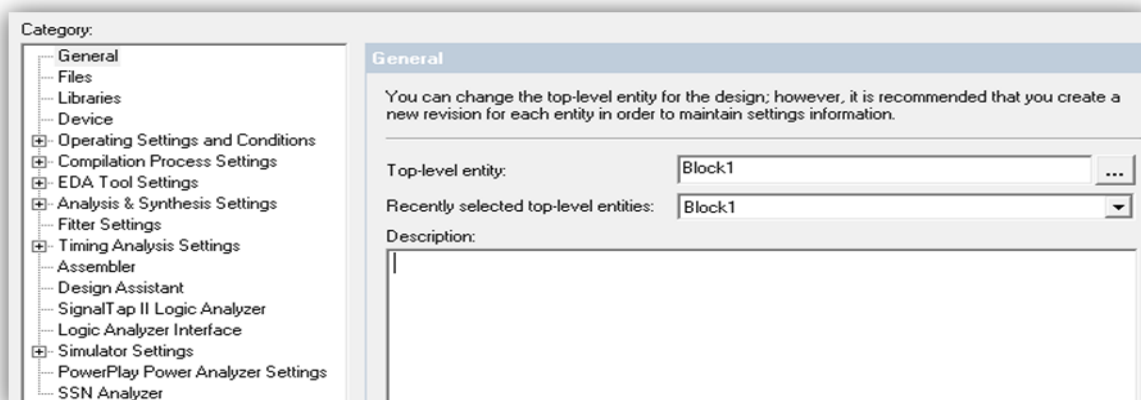


Fig. 9.4 Compiling .bdf

(6) Compile .bdf file

At present top level entity name is our module name (e.g. in our program it is *cnt_kit*). Replace the module name with block diagram file name (in our program it is *block1*). Click on **OK**. So can see that now top level entity name is *block1* under project navigator in Quartus II main window. Click on compilation icon and so that you have a window as shown in **Fig. 9.5** if compilation will successful.

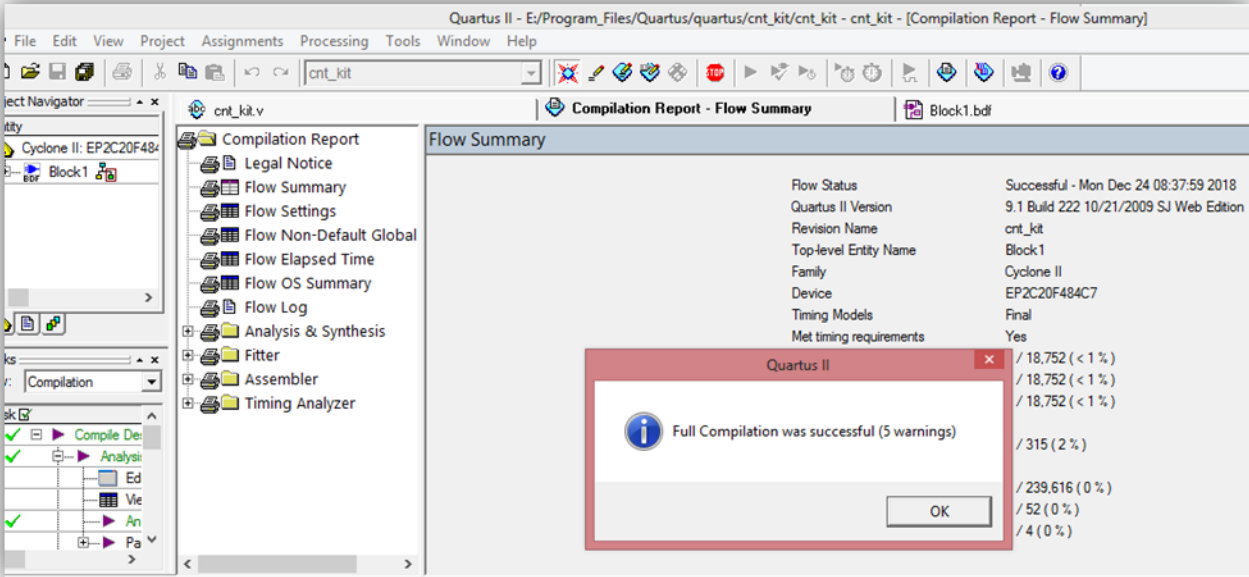


Fig. 9.5 Compilation

Acknowledge it by click on **OK**.

Now in *block1.bdf* file you can see as shown in **Fig. 9.6**, input and output ports with their assigned pin numbers.

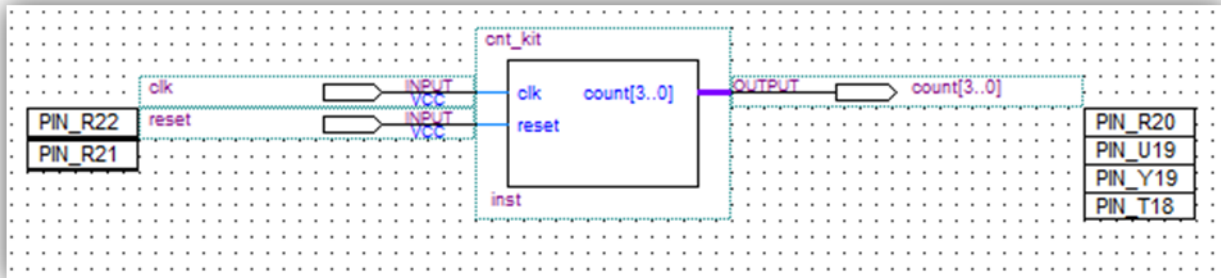


Fig. 9.6 Pins included in bdf

(7) To check the project on hardware

- One will have to observe SRAM Object File (.sof) in your project directory. This file will be generate during compilation in assembler step. For loading this file on hardware the steps below are to be followed.

Click on tools (menu) >Programmer and you have window as shown in **Fig. 9.7**.

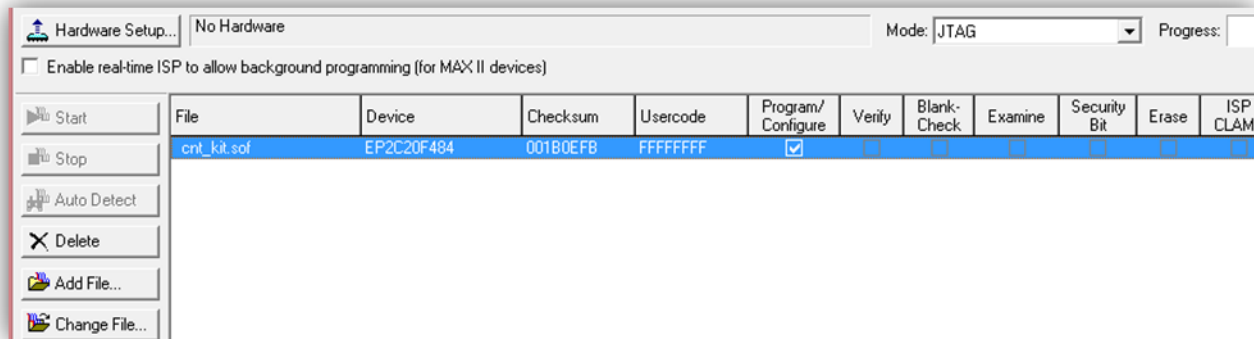


Fig. 9.7 Programmer

- (b) Observe that under hardware setup there is no hardware to load the program in DE1 kit. To select hardware click on hardware setup and you have a window as shown in **Fig. 9.8**
- (c) Click on Add hardware and you have a number of hardware supported by Quartus II. Select hardware that is available with you. We are using Byte Blaster II so we will select that hardware to load the .sof file to Altera DE1 kit.

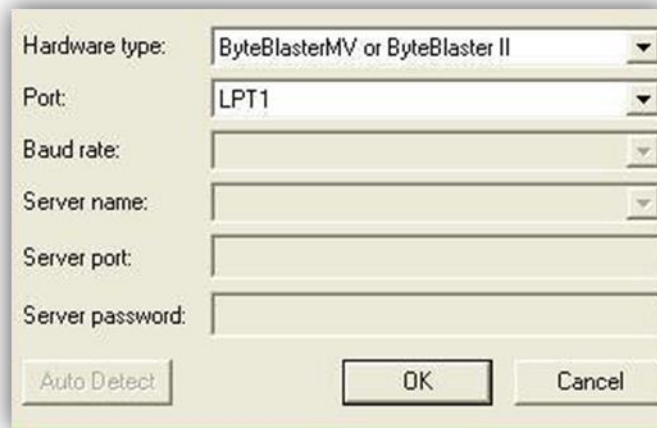


Fig. 9.8 Adding Hardware

- (d) Close this window. Click on program/configure as shown in **Fig. 9.7** and click on start. You can see the progress that program will load in a kit.

Having downloaded the configuration data into the FPGA device, you can now test this implemented circuit. *PINR_22* to give the clock input and verify the changes shown as an increment of count value and also reset the count variable. Verify that the circuit implementation matches with the simulation report for the verification. One can also use the internal clock for the clock signal.

CONCLUSION:

EXERCISES:

- (1) Design and test full adder on DE1 FPGA kit.
- (2) Design and test SISO shift register circuit.

EXPERIMENT – 5

BLOCKING AND NON BLOCKING ASSIGNMENT

OBJECTIVE:

- (i) To understand parallel and serial execution of digital circuits.

THEORY:

Procedural assignments update values of reg, integer, real or time variables. The value placed on a variable will remain unchanged until another procedural assignment updated the variable with a different value. These are unlike continuous assignments. The left hand side of procedural assignment can be one of the following.

- A reg, integer, real or time register variable or a memory element
- A bit select of these variables (e.g addr[0])
- A part select of these variables (e.g addr[31:16])
- A concatenation of any of the above

Blocking Assignment

Blocking assignment statements are executed in the order they are specified in a **sequential block**. A blocking assignment **will not block execution of statements** that follow in a parallel block. The “=” operator is used to specify blocking assignment.

Example:

```
always @(posedge clk)
begin
  Z=Y; Y=X; // shift register
  y=x; z=y; //parallel ff.
End
```

Non blocking Assignment

Non blocking assignment allows scheduling of assignments without blocking execution of the statements that follow in a sequential block. A “<=” operator is used to specify non blocking assignments. Note that this operator has the same symbol as a relational operator. The operator <= is interpreted as a relational operator in an expression and as an assignment operator in the context of a non blocking assignment.

Example

```
always @(posedge clk)
begin
  Z<=Y; Y<=X; // shift register
  y<=x; z<=y; //also a shift register
end
```

Application of Non Blocking assignment

Non blocking assignment is used as a method to model several concurrent data transfers that take place after a common event. Consider the following case where concurrent data transfer take place.

Example

```
always @(posedge clk)
begin
reg1<= #1 in1;
reg2<= @(negedge clock) in2 ^ in3;
reg3<= #1 reg1;
end
```

At each positive edge of clock, the following sequence takes place for the nonblocking assignments.

A read operation is performed on each right hand side variable, in1, in2, in3, and reg1, at the positive edge of clock. The right hand side expressions are evaluated and the result is stored internally in the simulator.

The write operation to the left hand side variables are scheduled to be executed at the time specified by the intra-assignment delay in each assignment, that is, schedule "write" to reg1 after q time unit, to reg2 at the next negative edge of clock, and to reg3 after 1 time unit.

The write operation is executed at the scheduled time steps. The order in which the write operation is executed is not important because the internally stored right hand side expression values are used to assign to the left hand side values. For example, note that reg3 is assigned the old value of reg1 that was stored after the read operation, even if the write operation wrote a new value to reg1 before the write operation to reg3 was executed.

SAMPLE PROGRAM 1:

Non-Blocking Assignment

```
module tis(in1,in2,in3,reg1,reg2,reg3,clock);
output reg1,reg2,reg3;
input clock,in1,in2,in3;
reg reg1,reg2,reg3;
always @(posedge clock)
begin
    reg1 <= #20 in1;
    reg2 <= @(negedge clock)in2^in3;
    reg3 <= reg1;
end
endmodule
```

OBSERVATION:

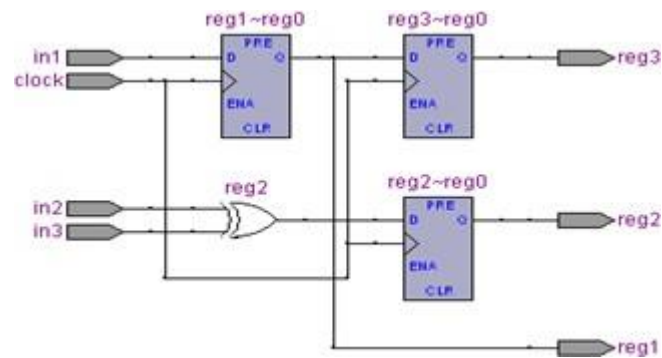


Fig. 4.1 RTL View of Non-Blocking Assignment

SAMPLE PROGRAM 2:

Blocking Assignment

```
module test(in1,in2,in3,reg1,reg2,reg3,clock);  
output reg1,reg2,reg3;  
input clock,in1,in2,in3;  
reg reg1,reg2,reg3;  
always @(posedge clock)  
begin  
    reg1 = #20 in1;  
    reg2 = @(negedge clock)in2^in3;  
    reg3 = reg1;  
end  
endmodule
```

OBSERVATION:

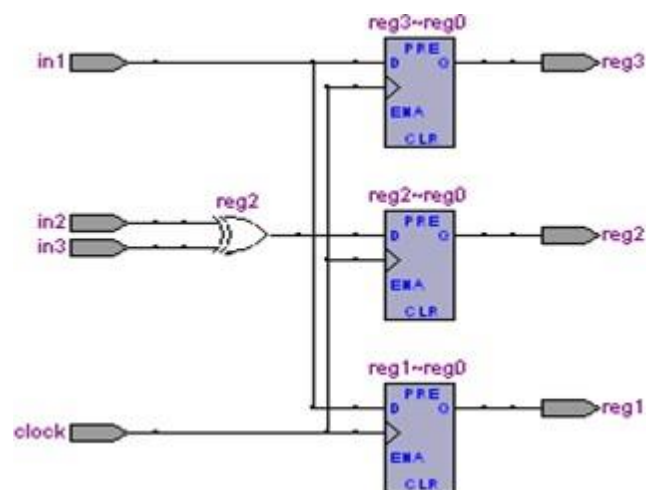


Fig. 4.2 RTL View of Blocking Assignment

Quartus-II simulator doesnot show delay assigned to each statement in blocking and non blocking assignment. To visualize effect of it, run the same verilog code on Modelsim Simulator and observe the effect of blocking and non-blocking assignment.

Note: Steps to use Modelsim is given in Appendix A

CONCLUSION:

EXERCISES:

- (2) Write a program of register that can store 2- bit serial value with blocking and nonblocking assignment. Observe the RTL view and derive conclusion.
- (3) Write a program to implement function $f_1 = a \mid b$ and $f_2 = f_1 \& c$, with the initial value $a=1$, $b=0$, $c=1$ and $f_1=1$. Observe the RTL view and derive conclusion.

EXPERIMENT – 6

LAYOUT GENERATION & DESIGN RULE CHECKING

OBJECTIVES:

- (i) To draw a layout of CMOS Inverter using Microwind Tool
- (ii) To use Design Rule Checker(DRC) in order to remove generated errors.

THEORY:

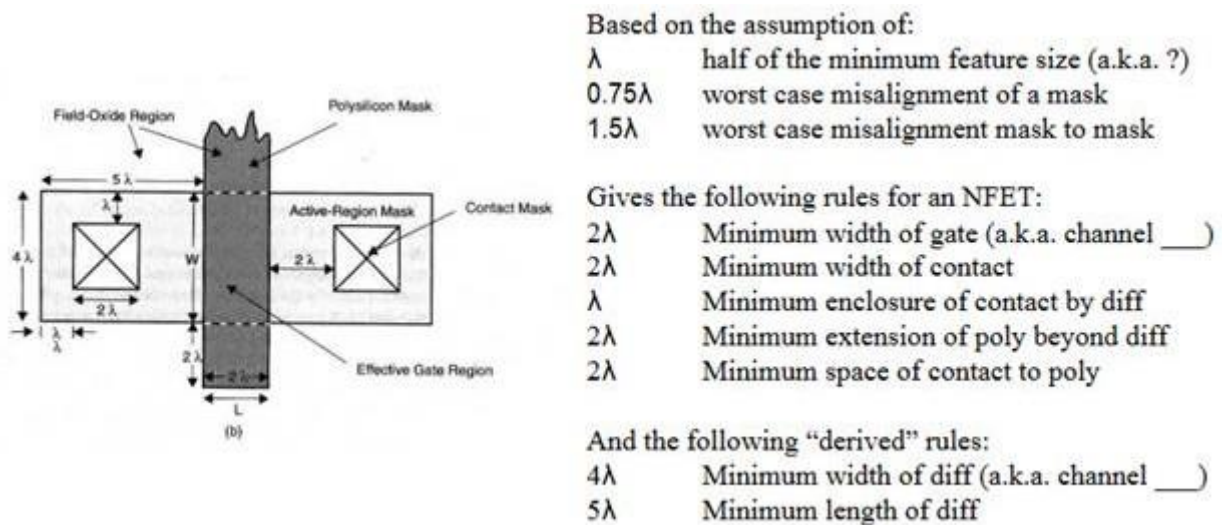


Fig. 6.1 Lambda Based Rules

PROCEDURE:

- (1) Open the file cmos06 which consists of design rules for the 0.06 micro Tech.
- (2) Open the microwind.exe and create the new file.
- (3) Observe the value of lambda 0.060 micron.
- (4) Create the n-well layout as per design rule.
- (5) Perform the design rule checking.
- (6) Create the V_{DD} (in n-well) and GND (outside n-well) power lines using metal1 layer. Perform the design rule checking.
- (7) Create the layout for the p+ and n+ diffusion as per design rule. Perform the design rule checking.
- (8) Create the layout for the poly (Gate) as per design rule. Perform the design rule checking.
- (9) Create the contact in p+ and n+ diffusion. Perform the design rule checking.
- (10) Create metal layer1 to connect contact of p+ diffusion to V_{DD} .
- (11) Create metal1 layer to connect contact of n+ diffusion to V_{SS} .
- (12) Create n+ diffusion in n-well. Create the contact. Fill up the metal to contact with metal1 layer and apply the V_{DD} .
- (13) Create p+ diffusion outside the n-well. Create the contact. Fill up the metal to contact with the metal2 layout and apply the V_{SS} .
- (14) Connect Metal1 and Metal2 outside the n-well using via.

- (15) Create the contact on the gate terminal. Fill up the metal by metal1 Or metal2 layer. Perform the design rule checking. Apply clock input to gate terminal.
- (16) Provide metal layer1 between drain of p+ diffusion and n+ diffusion for shorting purpose. Perform the design rule checking.
- (17) Apply observable point s1 on short (metal layer). Perform the design rule checking.
- (18) Perform simulation and obtain all the characteristics (Voltage vs time, V and I vs time, Voltage vs Voltage)

OBSERVATION:

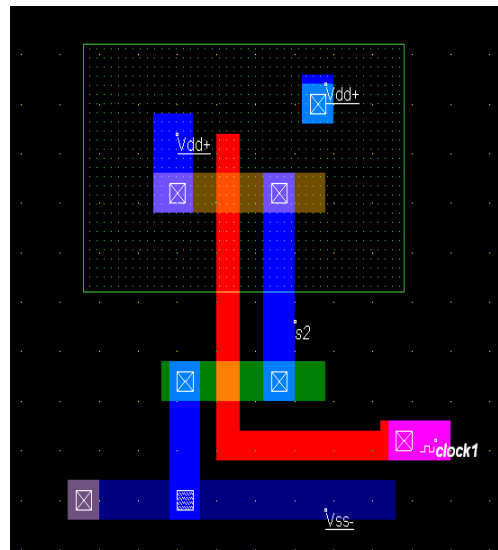


Fig. 6.2 Layout of CMOS Inverter

Observe transfer characteristic by applying DC input signal varying from 0 to 5 V.

EXERCISES:

- (1) Show different types of design rules violation with neat sketch.
- (2) Draw a layout for CMOS NAND gate and observe its transfer characteristic.

CONCLUSION:

EXPERIMENT – 7

CMOS INVERTER CHARACTERISTICS

OBJECTIVES:

- (i) To obtain DC and AC characteristics of CMOS Inverter.
- (ii) To understand the effect of Length(L) and width(W) of the channel of MOSFET on DC and AC characteristics.

THEORY:

The DC characteristics of the CMOS inverter shown in fig. 8.1 are portrayed in the voltage transfer characteristic(VTC), which is a plot of V_{out} as a function of V_{in} as shown in figure 8.2 . The midpoint voltage V_m is the point where the VTC intersects the unity gain line that is defined by $V_{out} = V_{in} = V_m$. From inverter analysis, we obtain the equation for V_m is

$$V_M = \frac{\beta_n \overline{V_{DD}} + (V_{DD} - V_{tp})}{1 + \beta_n / \beta_p} \quad (1)$$

This equation shows that V_M can be set by adjusting the ratio β_n / β_p , and $V_{tn} = V_{tp}$, then $V_M = V_{DD}/2$. Increasing this ratio decrease the inverter switching voltage. If the nFET and pFET are of equal size, then $\beta_n > \beta_p$ (as $k_n' > k_p'$), and $V_M < V_{DD}/2$.

Note: The mobility of holes in p-channels is about half that of electrons in n-channels, $\mu_p = \mu_n / 2$, which implies that we must adjust the width-length ratios to compensate:

$$k_n' = k_p' \rightarrow (W/L)_p = 2(W/L)_n$$

Fig.8.1 CMOS Inverter

Fig.8.2 Inverter Characteristic depends on β_n / β_p

The transient characteristics are obtained by analyzing the charge and discharge current flow paths through the transistors. By using the switch model as shown in fig. 8.3, the primary time constants are

$$\tau_n = R_n C_{out} = \frac{C_{out}}{\beta_n((V_{DD} - V_{thn}))} \quad (1)$$

$$\tau_p = R_p C_{out} = \frac{C_{out}}{\beta_p((V_{DD} - V_{thp}))} \quad (2)$$

If $k_n' > k_p'$, equal size transistors will give $t_{LH} > t_{LH}$. To obtain symmetrical switching, the pFET must have an aspect ratio of $(W/L)_p = k_n' > k_p' (W/L)_n$. This illustrates that which the ratio of β values sets the DC switching voltage V_M , the individual choices for β_n and β_p determine the transient switching times. In general, fast switching requires large transistors, illustrating the speed vs. area trade-off in CMOS design. The propagation delay time exhibits the same dependence.

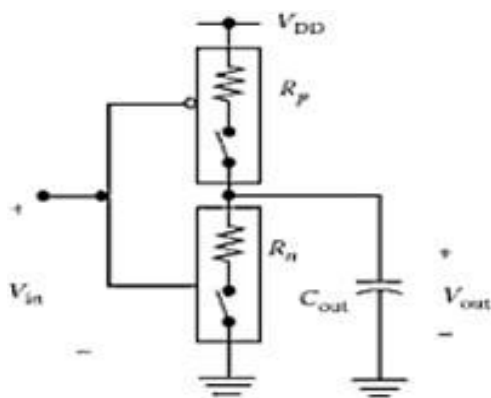


Fig. 8.3 Switch Model of an inverter

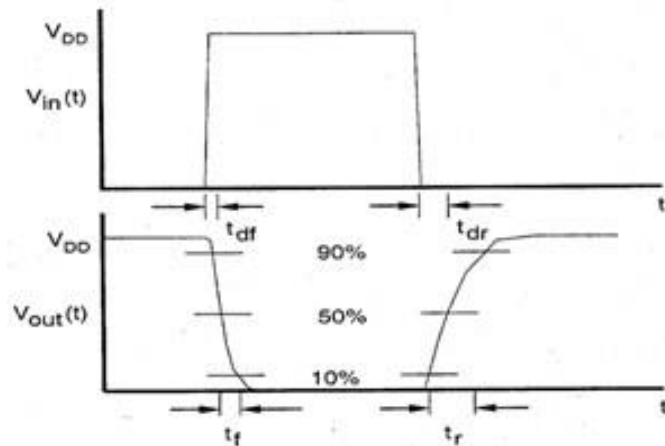


Fig. 8.4 Transient Characteristics of an inverter

SAMPLE PROGRAM:

*SPICE Deck for Simulating the Two Transistor CMOS Inverting Amplifier

M1 1 4 5 1 CMOSP L=3U W=95.0U

M2 5 4 0 0 CMOSN L=3u W=47.0U

C3 4 0 0.012000PF

* The following lines show a 50 pF load capacitor in parallel

* with a 10K load resistor.

C4 5 0 50.00000pF

*R4 5 0 10K

* IN 4

* CMOSN 0

* CMOSP 1

* GND0

* OUT 5

* Vdd 1

* These SCN-2.0um parameters taken from MOSIS

.MODEL CMOSN NMOS LEVEL=2 LD=0.250000U TOX=408.000001E-10

+ NSUB=6.264661E+15 VTO=0.77527 KP=5.518000E-05 GAMMA=0.5388

+ PHI=0.6 UO=652 UEXP=0.100942 UCRIT=93790.5

+ DELTA=1.000000E-06 VMAX=100000 XJ=0.250000U LAMBDA=2.752568E-03

+ NFS=2.06E+11 NEFF=1 NSS=1.000000E+10 TPG=1.000000

+RSH=31.020000 CGDO=3.173845E-10 CGSO=3.173845E-10 +CGBO=4.260832E-10

```

+ CJ=1.038500E-04 MJ=0.649379 CJSW=4.743300E-10 MJSW=0.326991 PB=0.800000
.MODEL CMOSF PMOS LEVEL=2 LD=0.213695U TOX=408.000001E-10
+ NSUB=5.574486E+15 VTO=-0.77048 KP=2.226000E-05 GAMMA=0.5083
+ PHI=0.6 UO=263.253 UEXP=0.169026 UCRIT=23491.2
+ DELTA=7.31456 VMAX=17079.4 XJ=0.250000U LAMBDA=1.427309E-02
+ NFS=2.77E+11 NEFF=1.001 NSS=1.000000E+10 TPG=-1.000000
+ RSH=88.940000 CGDO=2.712940E-10 CGSO=2.712940E-10 CGBO=3.651103E-10
+ CJ=2.375000E-04 MJ=0.532556 CJSW=2.707600E-10 MJSW=0.252466 PB=0.800000
Vdd 1 0 5.0
* To run the specific set of simulations, edit out the appropriate
* section's comment columns (*).
* DC TRANSFER CHARACTERISTIC
Vin 4 0
.dc Vin 0 5 0.1
.ic v(4)=0.01
*.plot dc v(4) v(5) (0,6)
* SWITCHING CHARACTERISTICS, tr and tf pulse response,
*Vin 4 0 pulse(0 5 0 0 0 0.000005 0.00001)
*.tran 0 .00001550
*.plot dc V(1) V(3)
.probe
.end

```

PROCEDURE:

- (1) Type above program on LtSPICE simulator and obtain DC characteristics.
- (2) Measure midpoint voltage V_m for a given width(W) of nFET and pFET.
- (3) Now change the width of nFET and pFET and measure V_m .
- (4) Repeat same task for other values of widths and measure V_m .
- (5) Repeat above step 2 and 3 to measure rise time and fall time for given input square wave.
- (6) Plot $V_{out} \rightarrow V_{in}$ for different β_n / β_p .
- (7) Plot $V_{out} \rightarrow$ time t for different values of β_n and β_p .
- (8) Derive your conclusion.

Observation

- a. Observe the change of V_m with respect to change in size of nFET and pFET.
- b. Observe the change of tr and tf with respect to change in size of nFET and pFET.

OBSERVATION TABLE:

DC Characteristics:

Width of nFET W_n	Width of pFET W_p	Midpoint Voltage V_m
159 nm	47 nm	
47 nm	47 nm	
47 nm	159 nm	

Switching Characteristics:

Width of nFET W_n	Width of pFET W_p	Rise Time t_r	Fall Time t_f

CONCLUSION:

EXERCISES:

- (1) Obtain switching characteristics of nFET Pass transistor and show that $\tau_r = 18 R_n C_{out}$ and $\tau_f = 2.84 R_n C_{out}$.
- (2) Obtain DC Characteristics of NAND and NOR gate and compare these characteristics with inverter DC characteristics.

EXPERIMENT – 8

Verification using Testbench

OBJECTIVES:

- (i) Create a testbench for the given digital circuit.
- (ii) Identify the bugs from the given design.

THEORY:

Verification is done before silicon development. It is done at time of product development for quality checking and bug fixing in design. This happens along with the development of the design and can start from the time the design architecture/micro architecture definition happens. The **main goal of verification is to ensure functional correctness of the design before the tape out**. However, with increasing design complexities, the scope of verification is also evolving to include much more than functionality. The verification process is considered very critical as part of design life cycle as any serious bugs in design not discovered before tape-out can lead to the need of newer stepping and increasing the overall cost of design process.

Sample Program1: 4-bit up counter

DUT:

```
module verification( input clk, input reset, output[3:0] counter);
reg [3:0] counter_up;

always @(posedge clk or posedge reset)
begin
    if(reset)
        counter_up <= 4'd0;
    else
        counter_up <= counter_up + 4'd1;
end

assign counter = counter_up;

endmodule
```

Testbench:

```
module upcounter_testbench();
reg clk, reset;
wire [3:0] counter;

verification dut(clk, reset, counter);
initial
begin
    clk = 1'b0;
    forever #5 clk = ~clk;
end
```

```

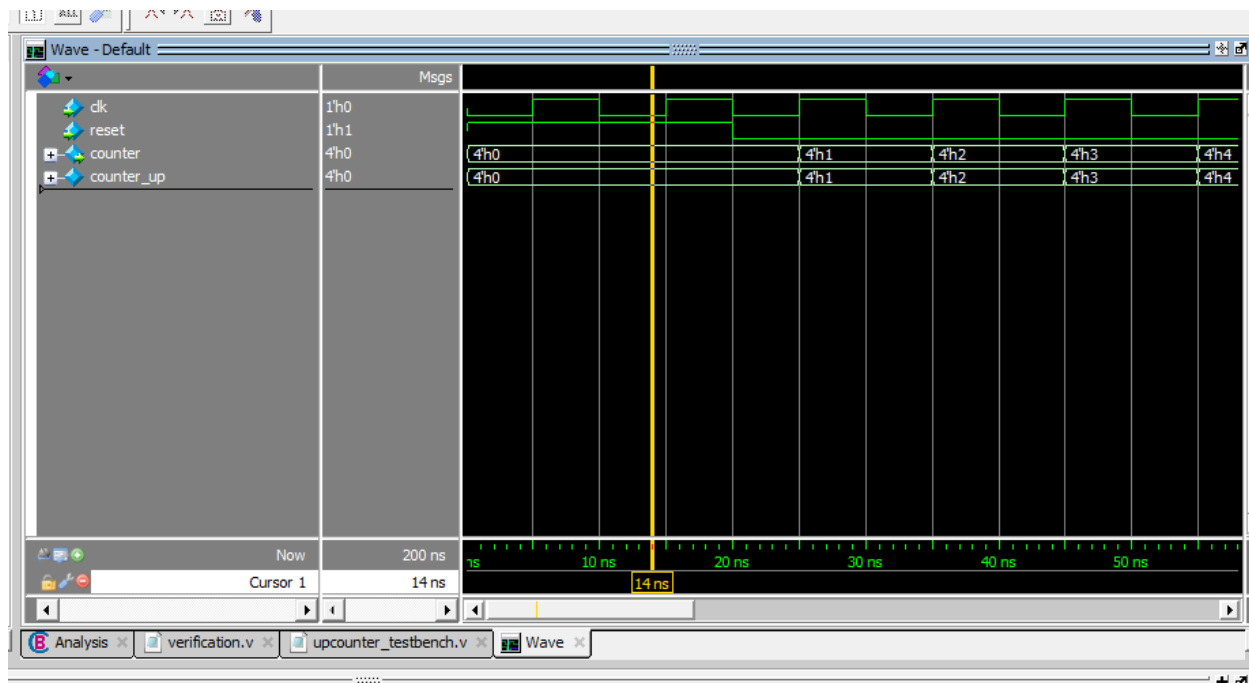
end

initial
begin
    reset = 1'b1;
    #20;
    reset = 1'b0;
    #500;
    $finish;
end

endmodule

```

Simulation Result:



Sample Program2: Full adder

DUT:

```

module fa (a, b, c, sum, carry);
input a;
input b;;
input c;
output sum;
output carry;

assign sum=a ^ b ^c;
assign carry=(a&b) | (b &c) | (c &a);
endmodule

```


Testbench:

```
module fulladdt_b;
reg a;
reg b;
reg c;
wire sum;
wire carry;
fa dut ( .a(a), .b(b),.c(c),.sum(sum),.carry(carry) );

initial begin
#10 a=1'b0; b=1'b0; c=1'b0;
#10 a=1'b0; b=1'b0; c=1'b1;
#10 a=1'b0; b=1'b1; c=1'b0;
#10 a=1'b0; b=1'b1; c=1'b1;
#10 a=1'b1; b=1'b0; c=1'b0;
#10 a=1'b1; b=1'b0; c=1'b1;
#10 a=1'b1; b=1'b1; c=1'b0;
#10 a=1'b1; b=1'b1; c=1'b1;
#10$stop;
end
endmodule
```

EXPERIMENT – 9

FINITE STATE MACHINE

OBJECTIVES:

- (i) To implement functionality of any digital circuit with finite state machine using Verilog .
- (ii) To study coding style for Moore and Mealy state machine.

THEORY:

Basically a FSM consists of combinational, sequential and output logic. **Combinational logic is used to decide the next state of the FSM, sequential logic is used to store the current state of the FSM. The output logic is a mixture of both combinational and sequential logic as shown in the Fig. below.**

Note: When modeling finite state machines, it is recommended to separate the sequential current-state logic from the combinational next-state and output logic.

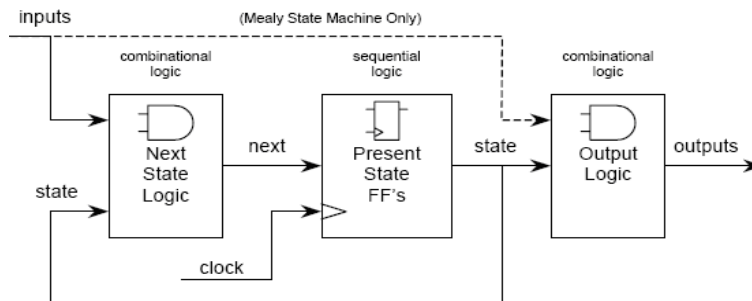


Fig.10.1 FSM Block Diagram

Types of State Machines

There are many ways to code these state machines, but before we get into the coding styles, let's first understand the basics a bit. There are two types of state machines:

Mealy State Machine

Its output depends on current state and current inputs. In the above picture, the blue dotted line makes the circuit a mealy state machine.

Moore State Machine :

Its output depends on current state only. In the above picture, when dotted line is removed the circuit becomes a Moore state machine. Depending on the need, we can choose the type of state machine. In general, or you can say most of the time, we end up using Mealy FSM.

One thing that need to be kept in mind when coding FSM is that combinational logic and sequence logic should be in two different always blocks. In the above two figures, next state logic is always the combinational logic. State Registers and Output logic are sequential logic. It is very important that any asynchronous signal to the next state logic be synchronized before being fed to the FSM. Always try to keep FSM in a separate Verilog file.

Important coding style notes:

- Parameters are used to define state encodings instead of the Verilog ``define` macro definition construct[3]. After parameter definitions are created, the parameters are used throughout the rest of the design, not the state encodings. This means that if an engineer wants to experiment with different state encodings, only the parameter values need to be modified while the rest of the Verilog code remains unchanged.
- Declarations are made for *state* and *next* (next state) after the parameter assignments.
- The sequential always block is coded using nonblocking assignments.
- The combinational always block sensitivity list is sensitive to changes on the *state* variable and all of the inputs referenced in the combinational always block.
- Assignments within the combinational always block are made using Verilog blocking assignments.
- The combinational always block has a default *next* state assignment at the top of the always block
- Default output assignments are made before coding the *case* statement (this eliminates latches and reduces the amount of code required to code the rest of the outputs in the *case* statement and highlights in the *case* statement exactly in which states the individual output(s) change).
- In the states where the output assignment is not the default value assigned at the top of the always block, the output assignment is only made once for each state.
- There is an if-statement, an else-if-statement or an else statement for each transition arc in the FSM state diagram. The number of transition arcs between states in the FSM state diagram should equal the number of if-else-type statements in the combinational always block.
- For ease of scanning and debug, all of the *next* assignments have been placed in a single column, as opposed to finding next assignments following the contour of the RTL code.

SAMPLE PROGRAM 1:

To detect sequence 11

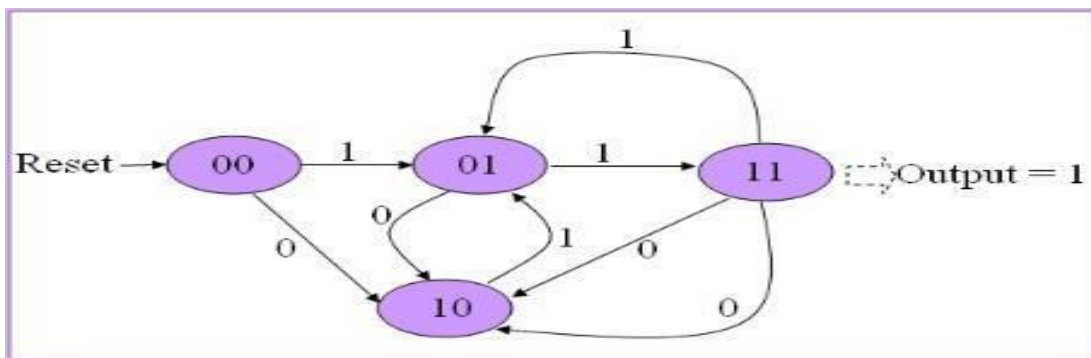


Fig. 10.2 Moore State Machine for Sequence Detector 11

```
module fsm( clk, rst, inp, outp);
    input clk, rst, inp;
    output outp;
    reg [1:0] state;
    reg outp;
```

```

always @( posedge clk, posedge rst )
begin
if( rst )
    state <= 2'b00;
else
begin
    case( state )
    2'b00:
        begin
            if( inp ) state <= 2'b01;
            else state <= 2'b10;
        end
    2'b01:
        begin
            if( inp ) state <= 2'b11;
            else state <= 2'b10;
        end
    2'b10:
        begin
            if( inp ) state <= 2'b01;
            else state <= 2'b11;
        end
    2'b11:
        begin
            if( inp ) state <= 2'b01;
            else state <= 2'b10;
        end
    endcase
end
end
always @(posedge clk, posedge rst)
begin
    if( rst )
    outp <= 0;
    else if( state == 2'b11 )
        outp <= 1;
    else outp <= 0;
end
endmodule

```

SAMPLE PROGRAM 2:

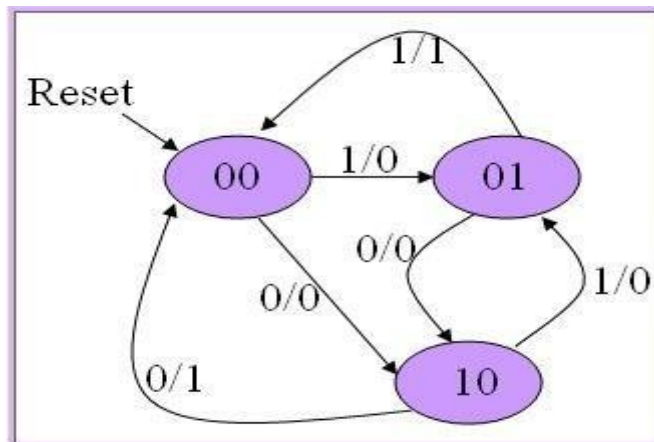


Fig.10.3 Mealy State Machine for sequence detector 11

```

module mealy( clk, rst, inp, outp);
  input clk, rst, inp; output outp;
  reg [1:0] state;
  reg outp;
  always @( posedge clk, posedge rst ) begin
    if( rst ) begin
      state <= 2'b00;
      outp <= 0;
    end
    else begin
      case( state )
        2'b00: begin
          if( inp ) begin
            state <= 2'b01;
            outp <= 0;
          end
          else begin
            state <= 2'b10;
            outp <= 0;
          end
        end
        2'b01: begin
          if( inp ) begin
            state <= 2'b00;
            outp <= 1;
          end
          else begin
            state <= 2'b10;
            outp <= 0;
          end
        end
        2'b10: begin
          if( inp ) begin
            state <= 2'b00;
            outp <= 0;
          end
          else begin
            state <= 2'b01;
            outp <= 0;
          end
        end
      endcase
    end
  end
end

```

```

        state <= 2'b01;
        outp <= 0;
    end
    else begin
        state <= 2'b00;
        outp <= 1;
    end
end
default: begin
    state <= 2'b00;
    outp <= 0;
end
endcase
end
endmodule

```

CONCLUSION:

EXERCISES:

- (1) Design a FSM for following sequence detector.

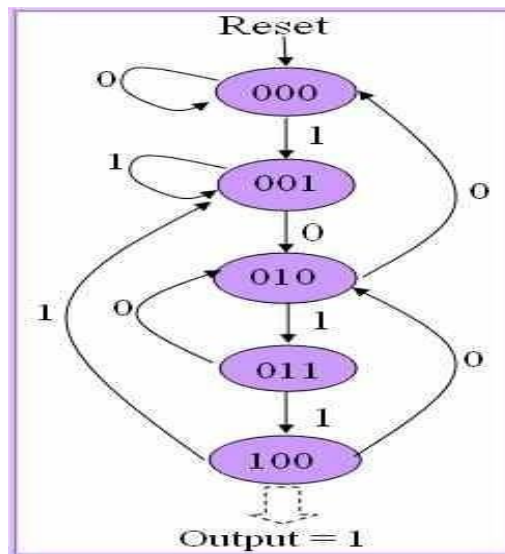


Fig. 10.4 Exercise 1

(2) Design a FSM for the following sequence detector.

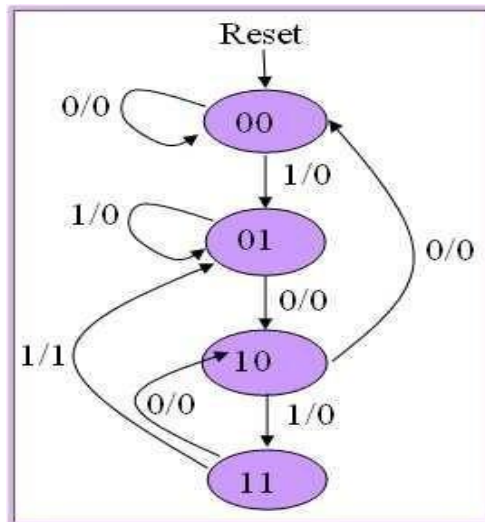


Fig. 10.5 Exercise 2

EXPERIMENT – 10

PROJECTS

OBJECTIVE:

- (i) To increase the coding efficiency in Verilog by giving some complex program.

PROJECT 1

Cold Drink Vending Machine

The machine accepts Re1 , Re 2 or Rs 5 coins. One coin at a time. Only after a total of Rs. 5 is reached the machine will release the cold-drink bottle and then return back to wait for next transaction.

PROJECT 2

Traffic Light Controller

Traffic lights are installed on an intersection of a busy highway and a local farm road. Detectors are installed on the intersection that cause signal *Car* to be asserted high in the presence of a car on the farm road approaching the intersection with the highway. In the initial state, the highway lights must be Green (HG) and the farm road light must be Red (FR). This state must remain for at least three clock cycles before any change of lights can occurs. When a car is detected on the farm road approaching the intersection, the highway lights should cycle from Green (HG) through Yellow (HY) to Red (HR), and the farm road light should subsequently turn Green (FG). Cycling through Yellow (HY) lights should take one clock cycle. The farm road lights are to remain Green (FG) only while the detector's signal *Car* remains high, but no longer than for two clock cycles. The farm road lights are then to cycle through Yellow (FY) to Red (FR), at which point the highway lights should turn Green (HG). Cycling through farm road Yellow (FY) also takes one clock cycle. Upon return to the initial state (HG, FR) the highway lights are not to be interrupted again for at least three clock cycles.

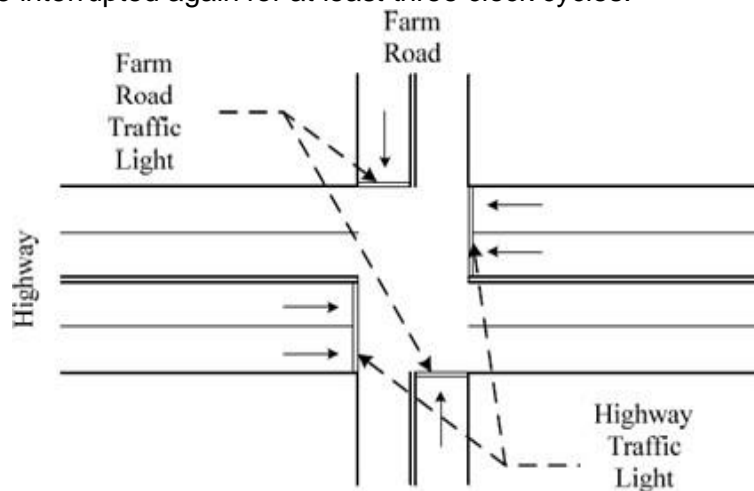


Fig.11.1: Highway & Farm Road Crossing

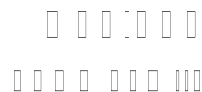
PROJECT 3

Lift Controller

In lift controller there is two state of lift condition. One is “idle” and second is “running”. One control signal is indicate whether lift is going in up direction or down direction. When the lift is in “idle” condition for more than 5 cycle it will come to ground floor. When someone call the lift from higher(lower) stories and in between someone call the lift from lower(higher) stories, lift will stop to lower(higher) stories if it is not passed and after that it will start to go towards first call.

PART II

DATASHEETS



□ □ □ □ □ □ □ □

[illegible][illegible]

<p> 000000 00 00000000 0 000000 0 00000000 </p>					
00000000	000000	000000	000000	000000	000000
000	000000	000000	000000	000000	000000
00000000 0000000000 00000000	00	00	00	00	00
00000000 00000000	000000	000000	000000	000000	000000
000000	0	0	0	0	0
00000000 00000000 00000000	0000	0000	0000	0000	0000

[illegible]

PART III

SESSIONAL QUESTION PAPERS



DHARMSINH DESAI UNIVERSITY, NADIAD
FACULTY OF TECHNOLOGY
SECOND SESSIONAL
SUBJECT: (EC-612) CMOS VLSI Design

Examination	: B.TECH (EC/IC)- Semester - VI	Seat No.	:
Date	: 15/02/2014	Day	: Saturday
Time	: 12.45 to 2 p.m.	Max. Marks	: 36

INSTRUCTIONS:

1. Figures to the right indicate maximum marks for that question.
2. The symbols used carry their usual meanings.
3. Assume suitable data, if required & mention them clearly.
4. Draw neat sketches wherever necessary.

- Q.1** Do as directed. [12]
- (a) What would be Fall time for four NMOS transistors connected in series? [02]
 - (b) Layout of MOSFET is given in fig. 1., Identify which design rules are not followed and why? [02]
 - (c) Prove that small size transistors are faster and having higher power dissipation. [02]
 - (d) Design a driver chain to drive $CL = 40$ pf if the initial stage has $C_{in} = 50$ fF. Use ideal scaling to determine the number of stages and the relative sizes. [02]
 - (e) Explain the terms (i) unit load and (ii) total compensation. [02]
 - (f) An 8-input CMOS NOR gate is to be designed to drive a load of 200 fF but is limited to an input capacitance of 20 fF. The options are logical cascade using either (i) NOR4 - NAND2 - INV or (ii) INV - NAND4 - NOR2. Which is best option? Justify. [02]

- Q.2** Attempt Any TWO of the following questions. [12]
- (a) Consider the logic cascade shown in **Fig.2**. If the i/p at A is switched from 1 to 0, Estimate the total delay after that changes in the i/p is reflected in output. Assume minimum size (1X) symmetrical layout ($r = 2.5$) for all logic blocks in the path. Assume nFET channel resistance with minimum dimension i.e. $R_n = 400 \Omega$ and $C_{FET} = 0$; [6]
 - (b) Consider the logic cascade shown in **Fig.2**. Use logical Effort to find the relative size of each stage needed to minimize delay through chain. [6]
 - (c) (i) Design CMOS logic gate for the logic function $Z = (A + BC + D)'$. Find out Eulerian path and give best layout (stick diagram) [6]
(ii) A region silicon doped p-type with a concentration of $5 \times 10^{15} \text{ cm}^{-3}$. If carrier mobilities are $\mu_n = 1400$ & $\mu_p = 500 \text{ cm}^2/\text{V-sec}$. What is the resistance of the region if it is 200 nm long and has a cross sectional area of $5 \mu\text{m}^2$?

OR

- Q.2** (a) A CMOS NAND2 is designed using identical nFETs with a value of $\beta_n = 2\beta_p$, the pFETs are the same size. The power supply is chosen to be $V_{DD} = 5\text{V}$, and the device threshold voltages are given as $V_{Tn} = 0.6$ and $V_{Tp} = -0.7\text{V}$. Find the midpoint voltage V_M for the case of simultaneous switching. What would be the midpoint voltage for an inverter made with the same β specification? [6]
- (b) An OAI function of the form $f = [(a+b)(b+c)d]'$ is built using series-parallel CMOS structuring. Design the circuit and find the transistor sizes needed to equalize the path resistances in both the nFET and pFET chain. An inverter with $\beta_n = 2\beta_p$ is used as a sizing reference. Expand the function into AOI form, and then apply the same sizing philosophy. Which design requires the smallest total transistor area? [6]

OR

- Q.3** (a) What is Out function in terms of A and B for the circuit of the layout shown in **fig. 3**. $V_{DD} = 3\text{V}$, $C_{jp} = C_{jn} = 1\text{fF}/\mu\text{m}^2$, $C_{ox} = 3\text{fF}/\mu\text{m}^2$, $\mu_n = 500\text{cm}^2/\text{V-sec}$, $\mu_p = 200\text{cm}^2/\text{V-sec}$, $V_{tn} = 0.6$, $V_{tp} = -0.7$, $L = 0.6\mu\text{m}$ for all transistors. Ignore the effects of lateral diffusion. What width, W_p , is required for a best-case rise time of 80psec? Assume $W_n = 1.8\mu\text{m}$ and $C_L = 5\text{fF}$. [6]
- (b) Draw a layout for $W/L = 10/1$ using unit size transistor of 2/1. [3]
 - (c) Draw a RC switch model for CMOS inverter having $R_p = 822\Omega$, $R_n = 427\Omega$ and $C_{out} = 150\text{fF}$. [3]
What would be shape of output waveform for $V_{in} = 1\sin 31000000000t$.

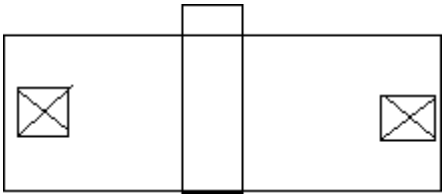


Fig. 1 Q. 1(b)

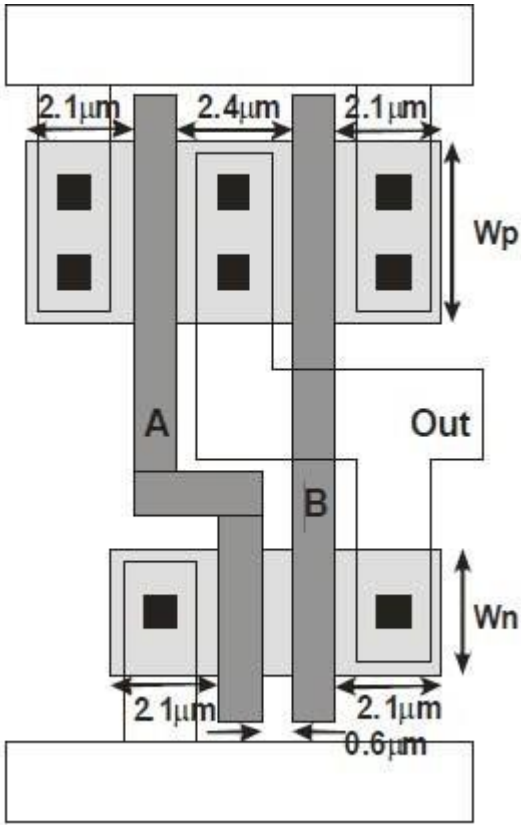
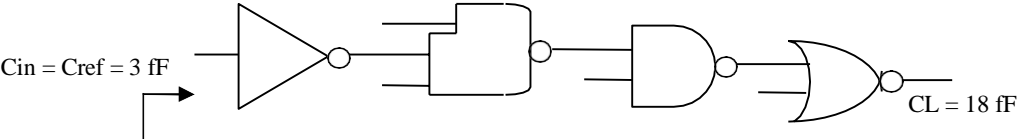


Fig. 3 OR Q. 3(a)



Assume

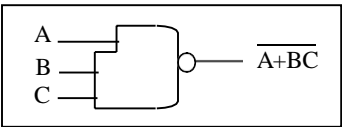


Fig. 2 Q. 2(a) and Q. 2(b)

APPENDIX

FLOOR PLANNING & OPTIMIZATION

OBJECTIVES:

- (i) To study an impact on critical path delay by floor planning and placement of logic cell (s).
- (ii) To relocate macro cells for delay optimization using ECHO fitting on chip planner.

THEORY:

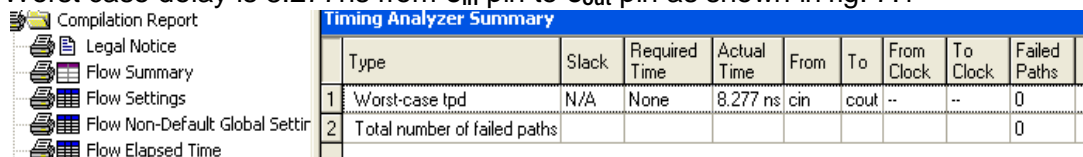
In VLSI, more than thousands of transistors are integrated into a single chip in order to fabricate an IC. It has two design phases. There are logical design and physical design. Here, physical design is the process of determining the physical location of active devices and interconnecting them inside the boundary of the VLSI chip. The purpose of VLSI physical design is to fix an abstract circuit description such as net-list, into silicon, creating a detailed geometric layout of the IC. In physical design, floor-planning determines the topology of the layout i.e. the relative positions of the blocks on the chip based on the interconnection requirements of the circuit and estimates for area.

Primary objectives of floor-planning problem are the optimized relative location and minimizing the dead space. The floor-plan design problem is by simultaneously considering the interconnection information as well as the area and shape information. This approach starts with an initial floor-plan and iteratively improves solutions by taking both interconnect and shape information into account until the convergence is reached or the runtime exceeds. Here, the optimization like area, wire-length, power and temperature optimization are achieved using any one of the meta-heuristic techniques.

PROCEDURE:

- (1) Compile the code given below and refer compilation report

```
module cri(x, y, cin, cout, sum);  
  input x;  
  input y;  
  input cin;  
  output cout;  
  output sum;  
  assign cout=(x&y) | (x & cin) | (y & cin);  
  assign sum= x ^ y ^cin;  
endmodule
```
- (2) Observe worst case delay report by clicking to Timing Analyzer and then summary tab. Worst case delay is 8.277ns from **c_{in}** pin to **c_{out}** pin as shown in fig. 7.1



The screenshot shows the 'Timing Analyzer Summary' window. On the left is a tree view with items: Compilation Report, Legal Notice, Flow Summary, Flow Settings, Flow Non-Default Global Settings, and Flow Elapsed Time. The main area displays a table with the following data:

Type	Slack	Required Time	Actual Time	From	To	From Clock	To Clock	Failed Paths
1 Worst-case tpd	N/A	None	8.277 ns	cin	cout	--	--	0
2 Total number of failed paths								0

Fig.7.1 Worst Case Delay Report

- (3) Open Chip Planner using tools menu (shown in Fig. 7.2). Now locate logic elements and generate fan in and fan-out connections. It shows connection of different i/o pin to logic elements and logic elements to other logic elements, so gives basically idea about netlist. Net list of a circuit tells about how connection has been made by compiler(Quartus).

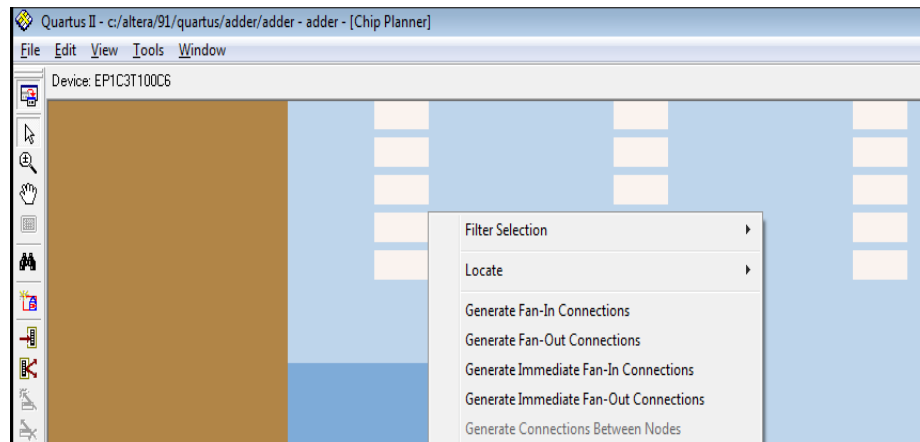


Fig.7.2 Chip Planner

- (4) Now we can see connections and path delay associated to those connections as shown in fig. 7.3. Our aim is to reduce path delay. Max frequency of any circuit depends on worst case path delay. To improve performance in terms of speed, we have to reduce worst case delay. We will put logic elements together in such manner so we can reduce that delay by doing manual placement of logic elements. This is true for a small circuit, but it is difficult to handle manual placement for larger logic elements.

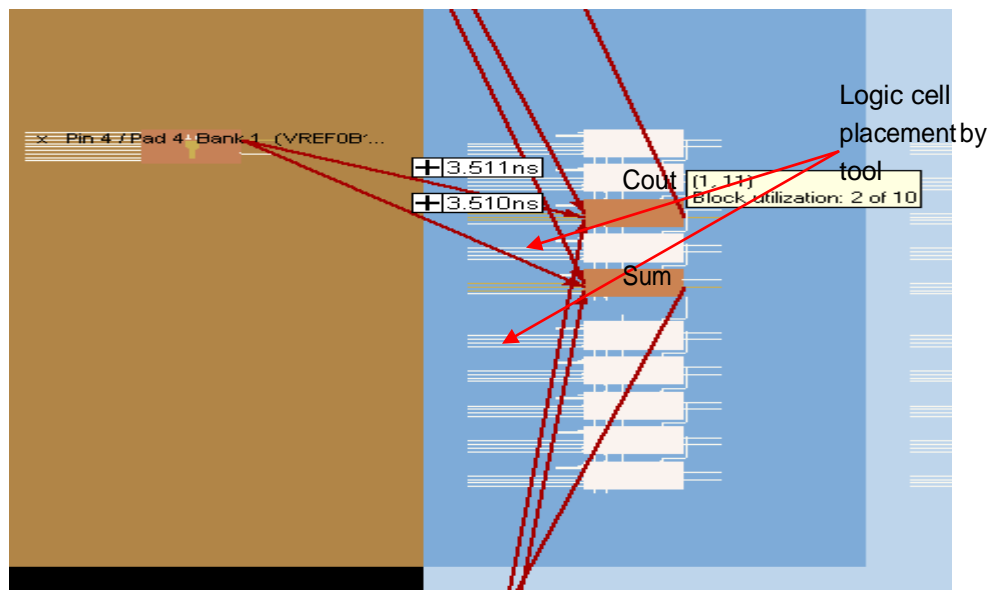


Fig.7.3 Floor Planning and Placement done Automatically

Type	Slack	Required Time	Actual Time	From Cell	To Cell	From Clock	To Clock	Failed Paths
1 Worst-case tpd	N/A	None	8.249 ns/cin	sum	0
2 Total number of failed paths	0

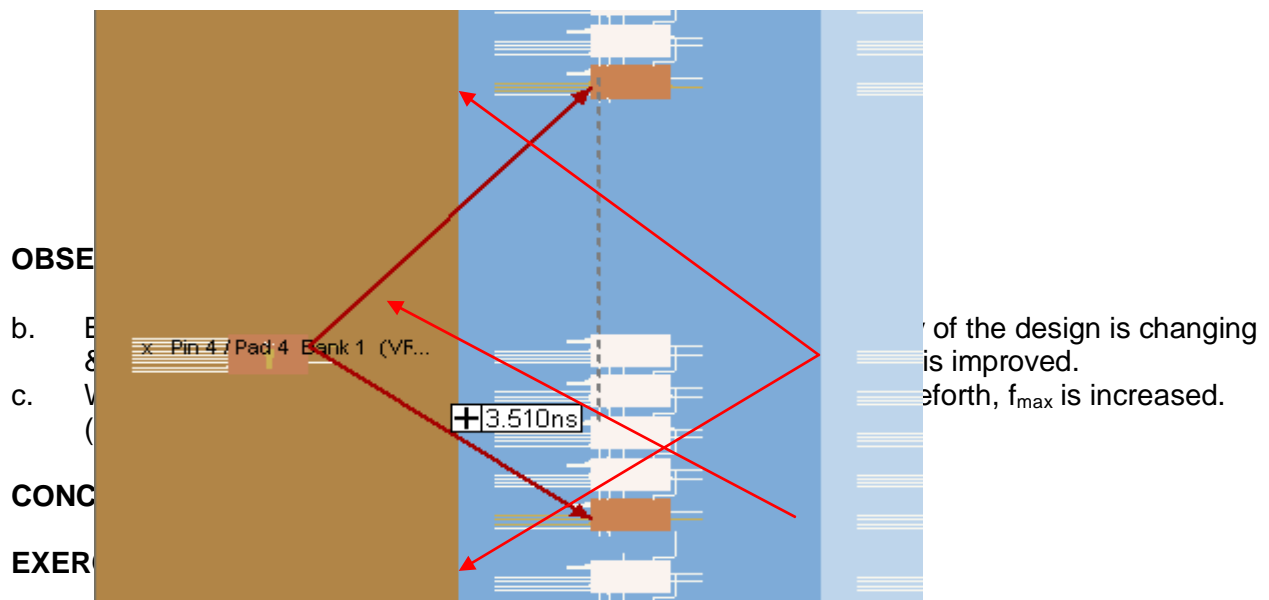
Logic cell placement manually (Cout cell shifted above)

Logic cell shifted (compare with the above figure)

Fig.7.4 Floor Planning and Placement done Manually

- (5) To change critical path by manual placement:
 Select logic elements, which contain connections between c_{in} and c_{out} pin drop near to c_{out} pin. (shown in fig. 7.4).
 Select new located logic elements and write click to locate logic elements in resource editor.
 Select check and save all net lists changes in edit menu.
- (6) Open Timequest Timing Analyzer and observe datasheet report. You will find less worst case path compare to previous floor plan.

Table 7.1 Time-quest Timing Analyzer



- (1) Repeat same steps for 4-bit Adder and observe effect on delay.

