SRI SHANMUGHA COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

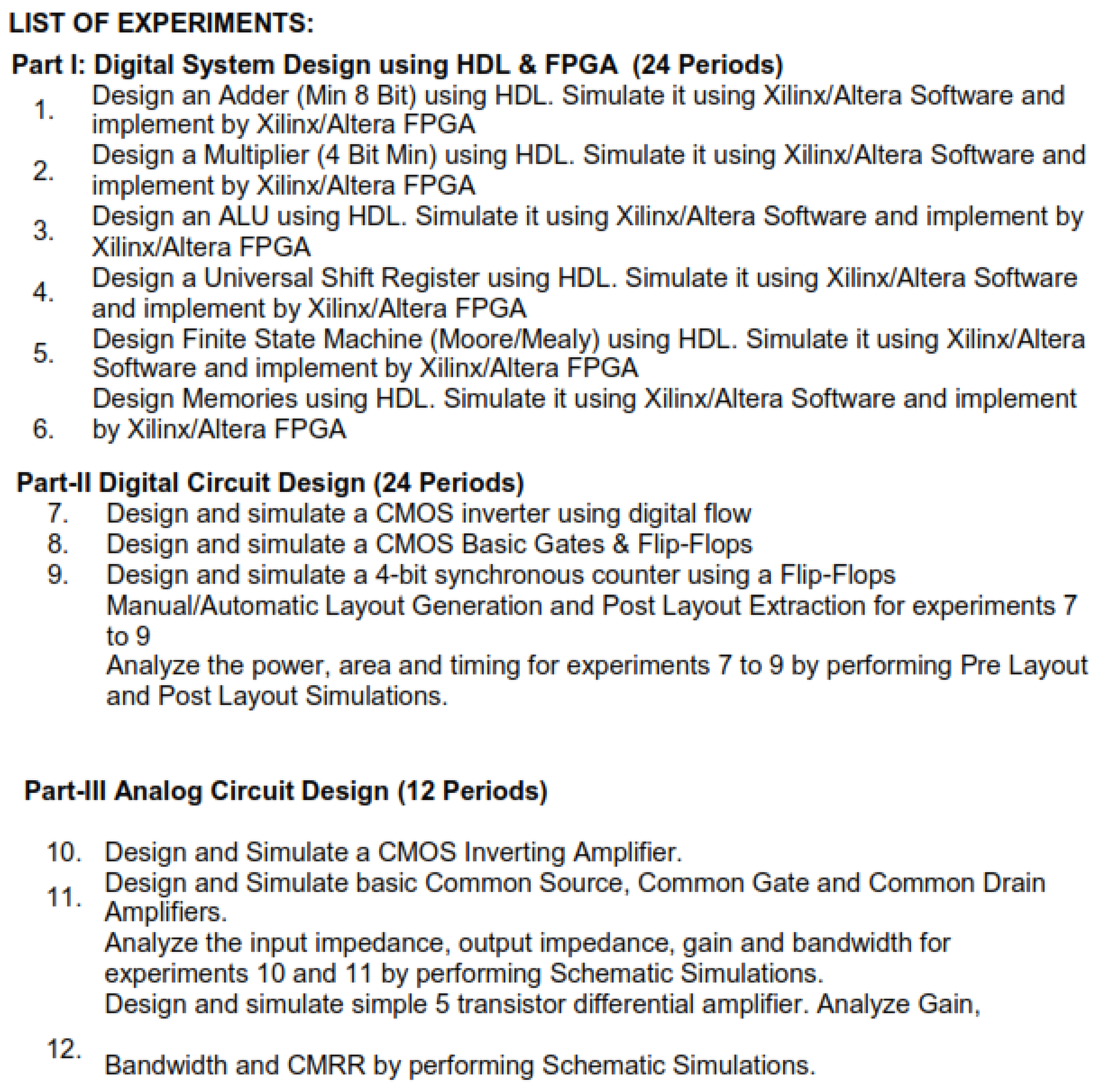
LAB MANUAL

EC8661-VLSI DESIGN LAB

REGULATION 2017

EX. NO	DATE	NAME OF THE EXPERIMENT	MARK	SIGNATURE
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				

EC8661 VLSI DESIGN LABORATORY L T P C 0 0 3 2



LIST OF EXPERIMENTS:

Part I: Digital System Design using HDL & FPGA (24 Periods)

1. Design an Adder (Min 8 Bit) using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA
2. Design a Multiplier (4 Bit Min) using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA
3. Design an ALU using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA
4. Design a Universal Shift Register using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA
5. Design Finite State Machine (Moore/Mealy) using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA
6. Design Memories using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA

Part-II Digital Circuit Design (24 Periods)

7. Design and simulate a CMOS inverter using digital flow
8. Design and simulate a CMOS Basic Gates & Flip-Flops
9. Design and simulate a 4-bit synchronous counter using a Flip-Flops
   Manual/Automatic Layout Generation and Post Layout Extraction for experiments 7 to 9
   Analyze the power, area and timing for experiments 7 to 9 by performing Pre Layout and Post Layout Simulations.

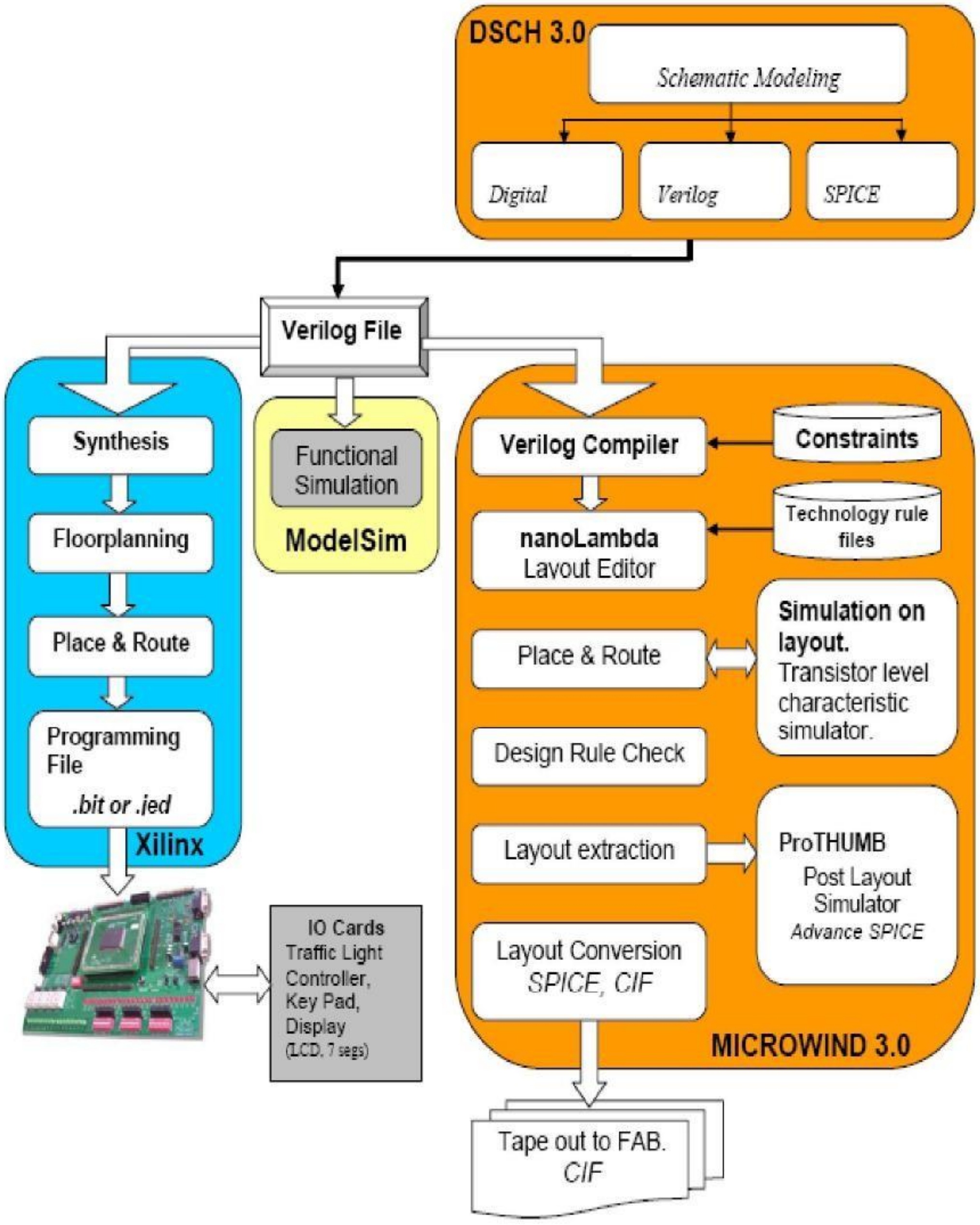
Part-III Analog Circuit Design (12 Periods)

10. Design and Simulate a CMOS Inverting Amplifier.
11. Design and Simulate basic Common Source, Common Gate and Common Drain Amplifiers.
    Analyze the input impedance, output impedance, gain and bandwidth for experiments 10 and 11 by performing Schematic Simulations.
12. Design and simulate simple 5 transistor differential amplifier. Analyze Gain, Bandwidth and CMRR by performing Schematic Simulations.

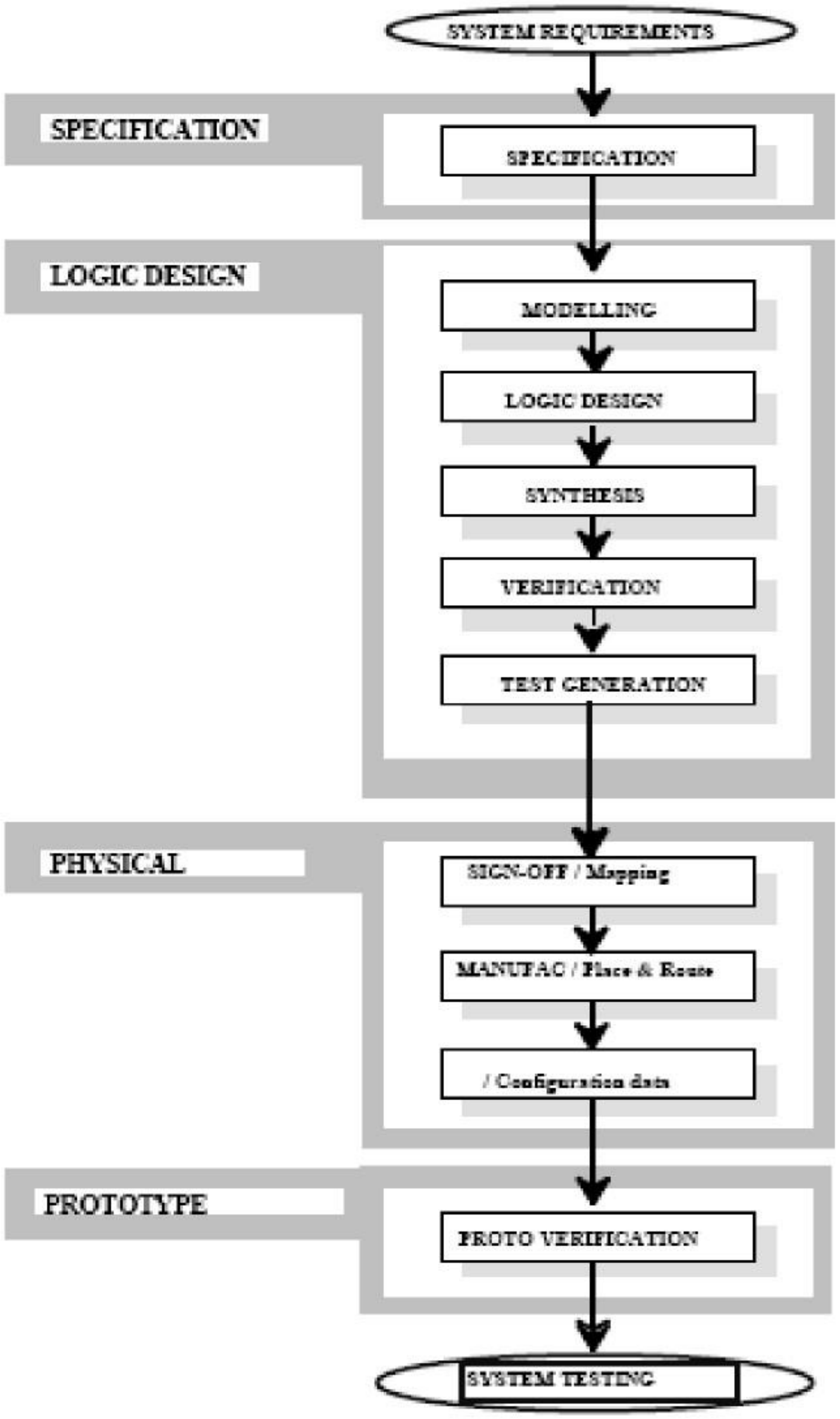
<u>CONTENT BEYOND SYLLABUS:</u>

1. Design and Simulate a Ripple Carry Adder
2. Design and Simulate a Multiplexer and De-Multiplexer.

VLSI DESIGN


DSCH 3.0
Schematic Modeling
Digital
Verilog
SPICE
Verilog File
Synthesis
Floorplanning
Place & Route
Programming File
.bit or .jed
Xilinx
Functional Simulation
ModelSim
Verilog Compiler
Constraints
nanoLambda Layout Editor
Technology rule files
Place & Route
Simulation on layout. Transistor level characteristic simulator.
Design Rule Check
Layout extraction
ProTHUMB Post Layout Simulator Advance SPICE
Layout Conversion SPICE, CIF
MICROWIND 3.0
IO Cards
Traffic Light Controller,
Key Pad,
Display
(LCD, 7 segs)
Tape out to FAB. CIF

ASIC DESIGN FLOW



EXP: Design of Logic gates

1.1 Introduction

The purpose of this experiment is to simulate the behavior of several of the basic logic gates and you will connect several logic gates together to create simple digital model.

1.2 Software tools Requirement

Equipments:

Computer with Modelsim Software

Specifications:

HP Computer P4 Processor - 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Softwares: Modelsim - 5.7c, Xilinx - 6.1i.

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -><filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

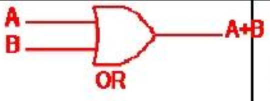
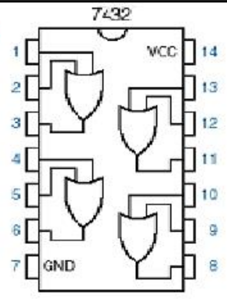
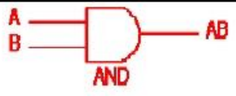
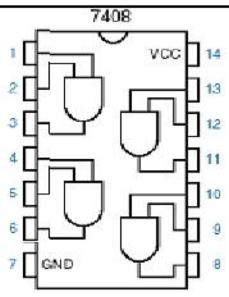
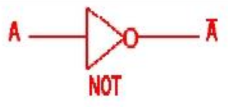
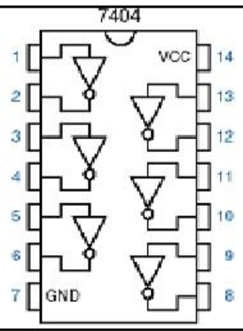
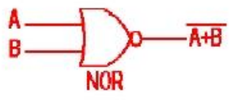
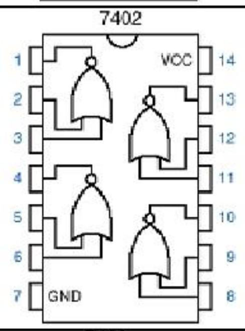
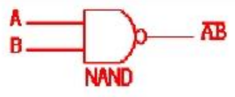
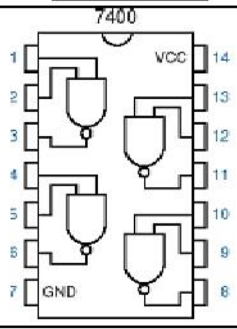
STEP 9: View -> Signals

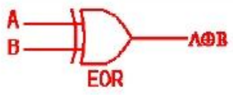
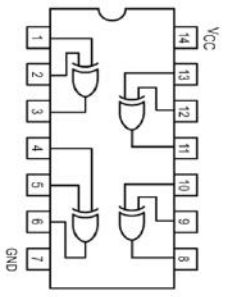
STEP 10: Select values -> Edit -> Force -> input values

STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again

1.3 Logic Gates and their Properties

Gate	Description	Truth Table			Logic Symbol	Pin Diagram
OR	The output is active high if any one of the input is in active high state, Mathematically, Q = A+B	A 0 0 1 1	B 0 1 0 1	Output Q 0 1 1 1		
AND	The output is active high only if both the inputs are in active high state, Mathematically, Q = A.B	A 0 0 1 1	B 0 1 0 1	Output Q 0 0 0 1		
NOT	In this gate the output is opposite to the input state, Mathematically, Q=A	A 0 1		Output Q 1 0		
NOR	The output is active high only if both the inputs are in active low state, Mathematically, Q = (A+B)'	A 0 0 1 1	B 0 1 0 1	Output Q 1 0 0 0		
NAND	The output is active high only if any one of the input is in active low state, Mathematically, Q = (A.B)'	A 0 0 1 1	B 0 1 0 1	Output Q 1 1 1 0		

XOR	The output is active high only if any one of the input is in active high state, Mathematically, Q = A.B' + B.A'	A 0 0 1 1	B 0 1 0 1	Output Q 0 1 1 0		7486 

1.4 Pre lab Questions

1. What is truth table?

2. Which gates are called universal gates?

3. A basic 2-input logic circuit has a HIGH on one input and a LOW on the other input, and the output is HIGH. What type of logic circuit is it?

4. A logic circuit requires HIGH on all its inputs to make the output HIGH. What type of logic circuit is it?

5. Develop the truth table for a 3-input AND gate and also determine the total number of possible combinations for a 4-input AND gate.

VERILOG Program

a) AND Gate

Structural Model	**Data Flow Model**	**BehaviouralModel**

moduleandstr(x,y,z);
 	
inputx,y;
 	
output z;
 	
and g1(z,x,y);
 	
endmodule
	moduleanddf(x,y,z);
	
	inputx,y;
	
	output z;
	
	assign z=(x&y);
	
	endmodule
		module andbeh(x,y,z);
		
		input x,y;
		
		output z;
		
		reg z;
		
		always @(x,y)
		
		z=x&y;
		
		endmodule

b) NAND Gate

Structural Model	Data Flow Model	BehaviouralModel

modulenandstr(x,y,z);
 	
inputx,y;
 	
output z;
 	
nand g1(z,x,y);
 	
endmodule
	modulenanddf(x,y,z);
	
	inputx,y;
	
	output z;
	
	assign z= !(x&y);
	
	endmodule
		module nandbeh(x,y,z);
		
		input x,y;
		
		output z;
		
		reg z;
		
		always @(x,y)
		
		z=!(x&y);
		
		endmodule

c) OR Gate

Structural Model	Data Flow Model	BehaviouralModel

module orstr(x,y,z);
 	
inputx,y;
 	
output z;
 	
or g1(z,x,y);
 	
endmodule
	module ordf(x,y,z);
	
	inputx,y;
	
	output z;
	
	assign z=(x|y);
	
	endmodule
		module orbeh(x,y,z);
		
		input x,y;
		
		output z;
		
		reg z;
		
		always @(x,y)
		
		z=x|y;
		
		endmodule

d) NOR Gate

Structural Model	Data Flow Model	BehaviouralModel

modulenorstr(x,y,z);
 	
inputx,y;
 	
output z;
 	
nor g1(z,x,y);
 	
endmodule
	modulenordf(x,y,z);
	
	inputx,y;
	
	output z;
	
	assign z= !(x|y);
	
	endmodule
		Modulenorbeh(x,y,z);
		
		input x,y;
		
		output z;
		
		reg z;
		
		always @(x,y)
		
		z=!(x|y);
		
		endmodule

e) XOR Gate

Structural Model	Data Flow Model	BehaviouralModel

module xorstr(x,y,z);
 	
inputx,y;
 	
output z;
 	
xor g1(z,x,y);
 	
endmodule
	module xordf(x,y,z);
	
	inputx,y;
	
	output z;
	
	assign z=(x^y);
	
	endmodule
		module xorbeh(x,y,z);
		
		input x,y;
		
		output z;
		
		reg z;
		
		always @(x,y)
		
		z=x^y;
		
		endmodule

f) XNOR Gate

Structural Model	Data Flow Model	BehaviouralModel

modulexnorstr(x,y,z);
 	
inputx,y;
 	
output z;
 	
xnor g1(z,x,y);
 	
endmodule
	modulexnordf(x,y,z);
	
	inputx,y;
	
	output z;
	
	assign z= !(x^y);
	
	endmodule
		module xnorbeh(x,y,z);
		
		input x,y;
		
		output z;
		
		reg z;
		
		always @(x,y)
		
		z=!(x^y);
		
		endmodule

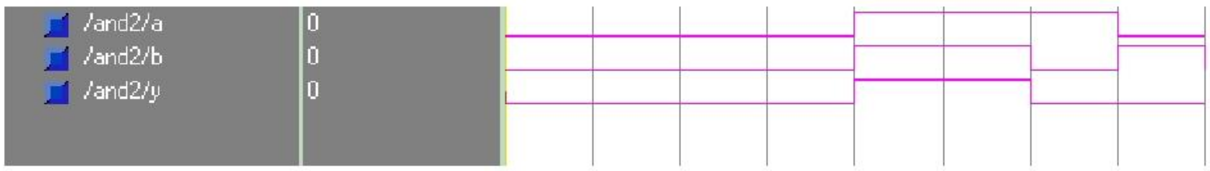
g) NOT Gate

Structural Model	Data Flow Model	BehaviouralModel

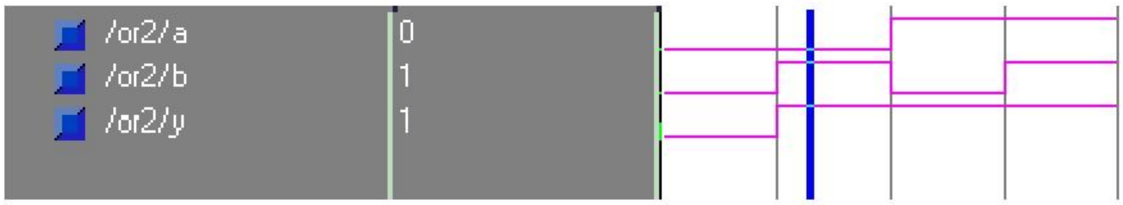
module notstr(x,z);
 	
input x;
 	
output z;
 	
not g1(z,x);
 	
endmodule
	module notdf(x,z);
	
	input x;
	
	output z;
	
	assign z= !x;
	
	endmodule
		module notbeh(x,z);
		
		input x;
		
		output z;
		
		reg z;
		
		always @(x)
		
		z=!x;
		
		endmodule

Out put waveforms

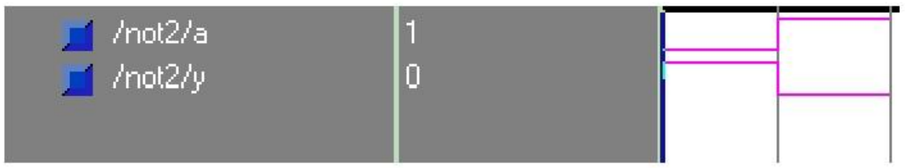
AND Gate:



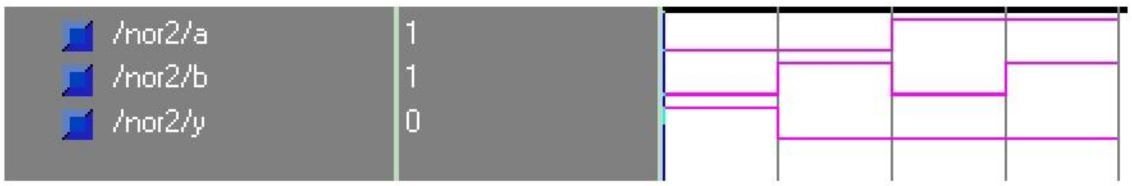
OR Gate:



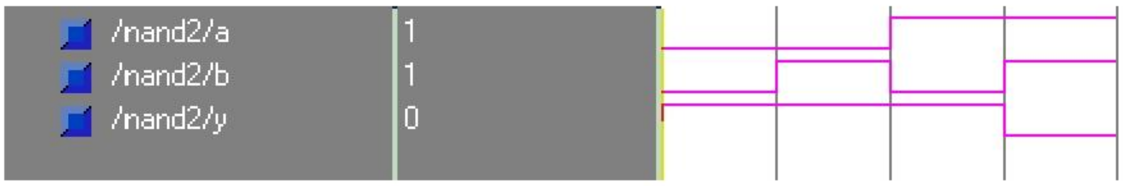
NOT Gate:



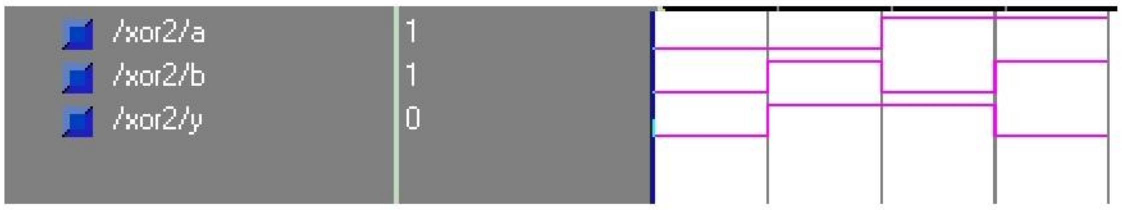
NOR Gate:



NAND Gate:



XOR Gate:



RESULT

Thus the logic gates were implemented using XILINX .

EXP: Design of Binary Adders

2.1 Introduction

The purpose of this experiment is to introduce the design of simple combinational circuits, in this case half adders, half subtractors, full adders and full subtractors.

2.2 Software tools Requirement

Equipments:

Computer with Modelsim Software

Specifications:

HP Computer P4 Processor - 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Softwares: Modelsim - 5.7c, Xilinx - 6.1i.

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -><filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

STEP 9: View -> Signals

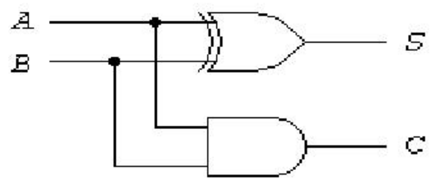
STEP 10: Select values -> Edit -> Force -> input values

STEP 11: Add -> Wave -> Selected signals -> Run

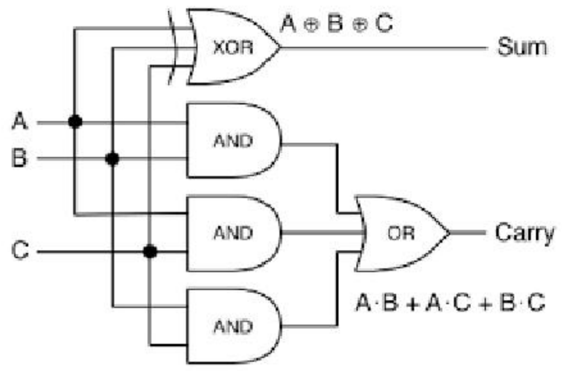
STEP 12: Change input values and run again

2.3 Logic Diagram

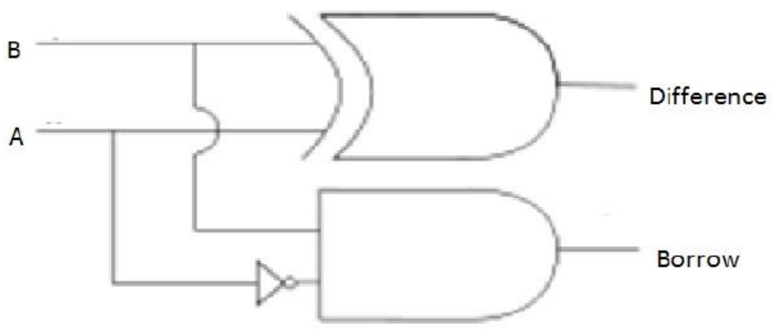
Half adder

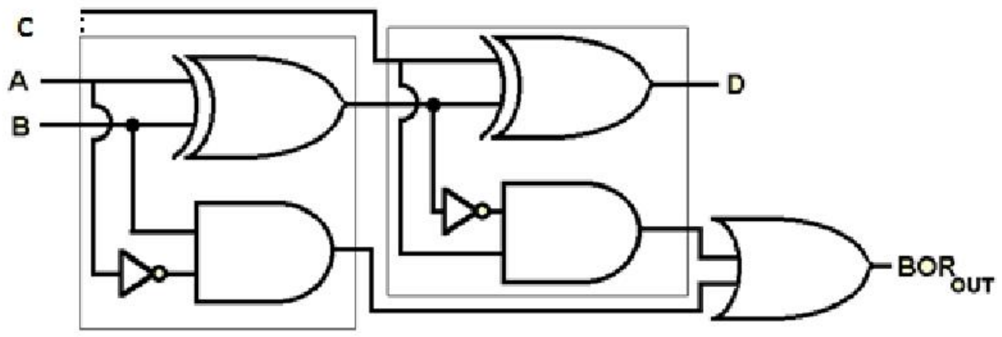


Full adder



Halfsubtractor





Full subtractor

2.4 Pre lab Questions

1. What is meant by combinational circuits?

2. Write the sum and carry expression for half and full adder.

3. Write the difference and borrow expression for half and full subtractor.

4. What is signal? How it is declared?

5. Design a one bit adder.

VERILOG Program

HALF ADDER:

Structural model	**Dataflow model**	**Behaviouralmodel**

modulehalfaddstr(sum,carry,a,b);
 	
outputsum,carry;
 	
inputa,b;
 	
xor(sum,a,b);
 	
and(carry,a,b);
 	
endmodule
	modulehalfadddf(sum,carry,a,b);
	
	outputsum,carry;
	
	inputa,b;
	
	assign sum = a ^ b;
	
	assign carry=a&b;
	
	endmodule
		modulehalfaddbeh(sum,carry,a,b);
		
		outputsum,carry;
		
		inputa,b;
		
		regsum,carry;
		
		always @(a,b);
		
		sum = a ^ b;
		
		carry=a&b;
		
		endmodule

FULL ADDER:

Structural model	Dataflow model	Behaviouralmodel

module fulladdstr(sum,carry,a,b,c);
 	
outputsum,carry;
 	
inputa,b,c;
 	
xor g1(sum,a,b,c);
 	
and g2(x,a,b);
 	
and g3(y,b,c);
 	
and g4(z,c,a);
 	
or g5(carry,x,z,y);
 	
endmodule
	modulefulladddf(sum,carry,a,b,c);
	
	outputsum,carry;
	
	inputa,b,c;
	
	assign sum = a ^ b^c;
	
	assign carry=(a&b) | (b&c) | (c&a);
	
	endmodule
		modulefulladdbeh(sum,carry,a,b,c);
		
		outputsum,carry;
		
		inputa,b,c;
		
		regsum,carry;
		
		always @ (a,b,c)
		
		sum = a ^ b^c;
		
		carry=(a&b) | (b&c) | (c&a);
		
		endmodule

HALF SUBTRACTOR:

Structural model	Dataflow Model	BehaviouralModel

modulehalfsubtstr(diff,borrow,a,b);
 	
outputdiff,borrow;
 	
inputa,b;
 	
xor(diff,a,b);
 	
and(borrow,~a,b);
 	
endmodule
	modulehalfsubtdf(diff,borrow,a,b);
	
	outputdiff,borrow;
	
	inputa,b;
	
	assign diff = a ^ b;
	
	assign borrow=(~a&b);
	
	endmodule
		modulehalfsubtbeh(diff,borrow,a,b);
		
		outputdiff,borrow;
		
		inputa,b;
		
		regdiff,borrow;
		
		always @(a,b)
		
		diff = a ^ b;
		
		borrow=(~a&b);
		
		endmodule

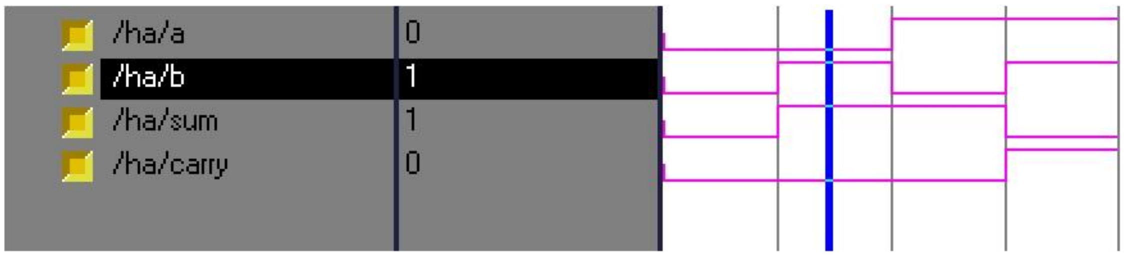
FULL SUBTRACTOR:

Structural model	Dataflow Model	BehaviouralModel

module fullsubtstr(diff,borrow,a,b,c);
 	
outputdiff,borrow;
 	
inputa,b,c;
 	
wire a0,q,r,s,t;
 	
not(a0,a);
 	
xor(x,a,b);
 	
xor(diff,x,c);
 	
and(y,a0,b);
 	
and(z,~x,c);
 	
or(borrow,y,z);
 	
endmodule
	modulefullsubtdf(diff,borrow,a,b,c);
	
	outputdiff,borrow;
	
	inputa,b,c;
	
	assign diff = a^b^c;
	
	assign borrow=(~a&b)|(~(a^b)&c);
	
	endmodule
		modulefullsubtbeh(diff,borrow,a,b,c);
		
		outputdiff,borrow;
		
		inputa,b,c;
		
		outputdiff,borrow;
		
		always@(a,b,)
		
		diff = a^b^c;
		
		borrow=(~a&b)|(~(a^b)&c);
		
		endmodule

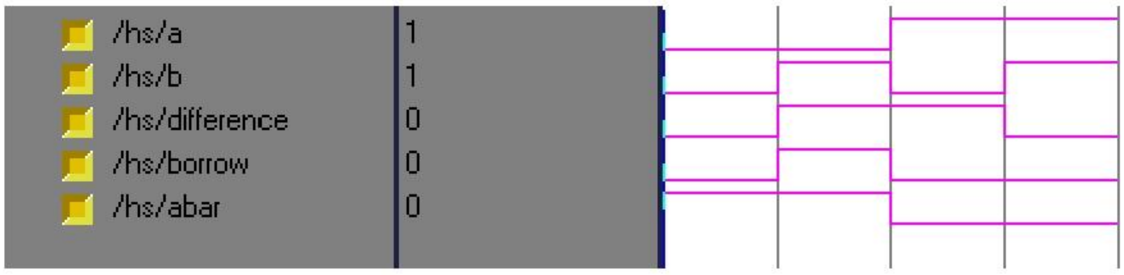
Output waveforms:

Half Adder:



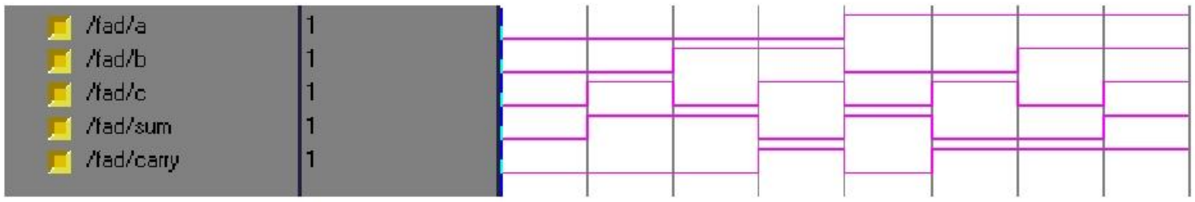
```
/ha/a      0
/ha/b      1
/ha/sum    1
/ha/carry  0
```

Half subtractor:

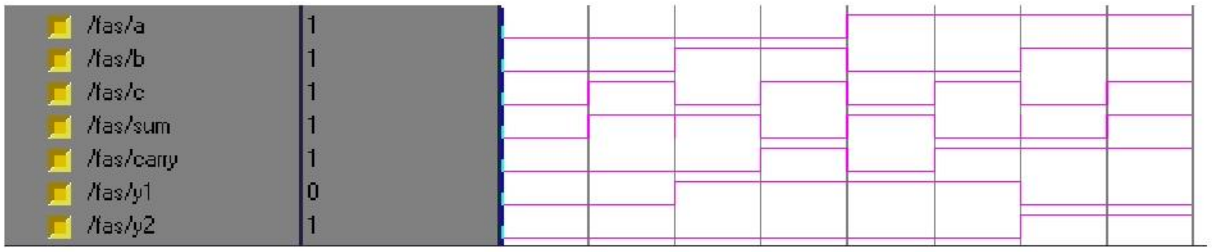


```
/hs/a           1
/hs/b           1
/hs/difference  0
/hs/borrow      0
/hs/abar        0
```

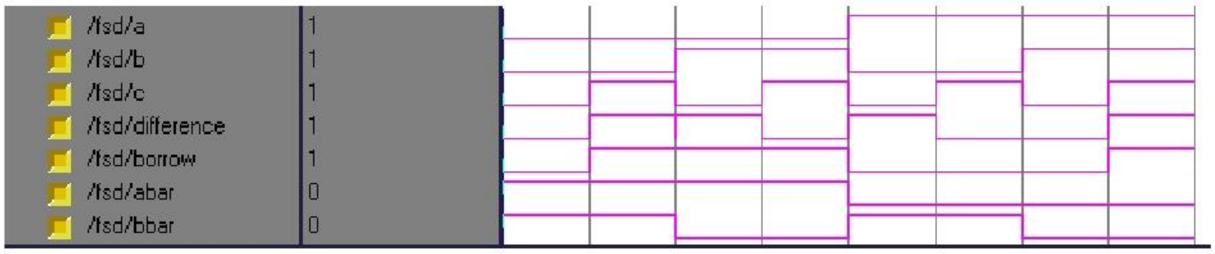
Full adder Dataflow modeling:



Full adder structural modeling:



Full Subtractor Dataflow modeling:



RESULT

Thus the Half-adder, Full adder, Four bit adder were implemented using Xilinx.

EXP: Design of Ripple carry,Carry select and Carry save Adders

3.1 Introduction

The purpose of this experiment is to introduce the design of Ripple carry,Carry select and Carry save Adders

3.2 Software tools Requirement

Equipments:

Computer with Modelsim Software

Specifications:

HP Computer P4 Processor - 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Softwares: Modelsim - 5.7c, Xilinx - 6.1i.

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -><filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

STEP 9: View -> Signals

STEP 10: Select values -> Edit -> Force -> input values

STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again

3.3 Ripple carry adder

VHDL Program

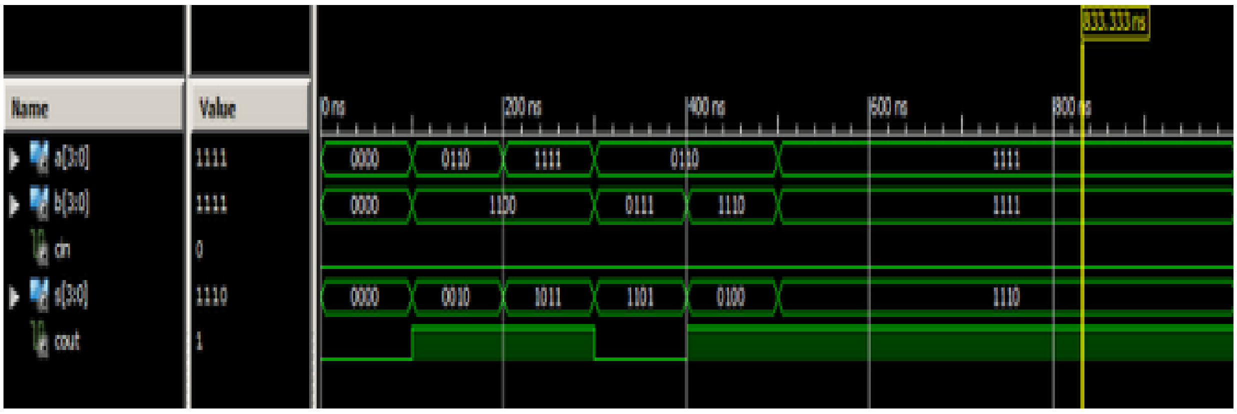
8-bit Ripple Carry Adder

Structural code for RCA	Test bench for RCA

module rippe_adder(X, Y, S, Co);
 input [3:0] X, Y;
 output [3:0] S;
 output Co;
wire w1, w2, w3;
fulladder u1(X[0], Y[0], 1'b0, S[0], w1);
 fulladder u2(X[1], Y[1], w1, S[1], w2);
 fulladder u3(X[2], Y[2], w2, S[2], w3);
 fulladder u4(X[3], Y[3], w3, S[3], Co);
endmodule
module fulladder(X, Y, Ci, S, Co);
  input X, Y, Ci;
  output S, Co;
  wire w1,w2,w3;
 	
  xor G1(w1, X, Y);
  xor G2(S, w1, Ci);
  and G3(w2, w1, Ci);
  and G4(w3, X, Y);
  or G5(Co, w2, w3);
endmodule
	ENTITY Tb_Ripple_Adder IS
	END Tb_Ripple_Adder;
	
	ARCHITECTURE behavior OF Tb_Ripple_Adder IS
	
	-- Component Declaration for the Unit Under Test (UUT)
	
	COMPONENT Ripple_Adder
	PORT(
	A : IN std_logic_vector(3 downto 0);
	B : IN std_logic_vector(3 downto 0);
	Cin : IN std_logic;
	S : OUT std_logic_vector(3 downto 0);
	Cout : OUT std_logic
	);
	END COMPONENT;
	
	--Inputs
	signal A : std_logic_vector(3 downto 0) := (others => '0');
	signal B : std_logic_vector(3 downto 0) := (others => '0');
	signal Cin : std_logic := '0';
	
	--Outputs
	signal S : std_logic_vector(3 downto 0);
	signal Cout : std_logic;
	
	BEGIN
	
	-- Instantiate the Unit Under Test (UUT)
	uut: Ripple_Adder PORT MAP (
	A => A,
	B => B,
	Cin => Cin,
	S => S,
	Cout => Cout
	);
	
	-- Stimulus process
	stim_proc: process
	begin
	-- hold reset state for 100 ns.
	wait for 100 ns;
	A <= "0110";
	B <= "1100";
	
	wait for 100 ns;

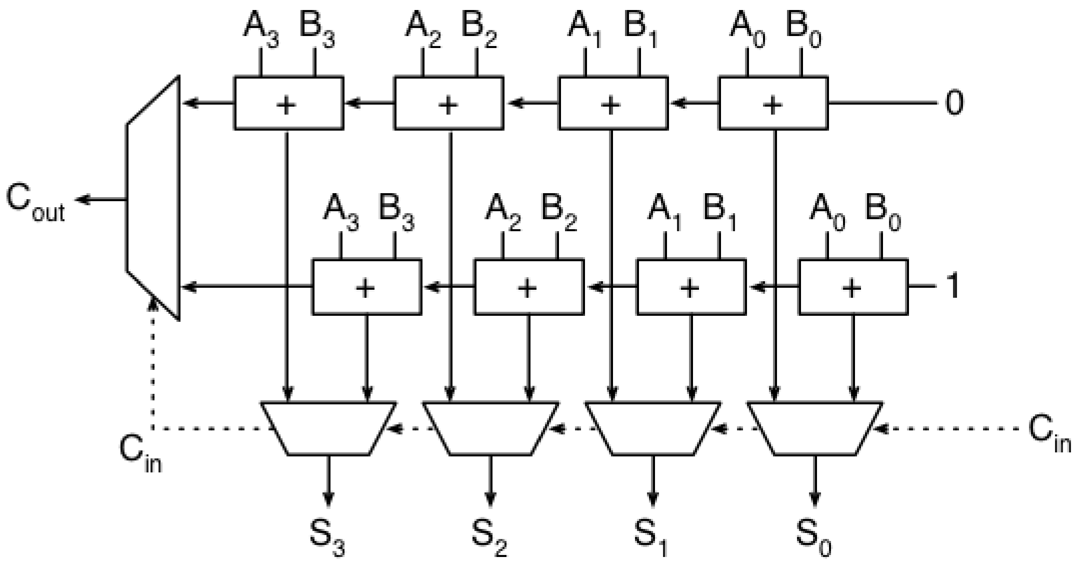
	A <= "1111";
	B <= "1100";
	
	wait for 100 ns;
	A <= "0110";
	B <= "0111";
	
	wait for 100 ns;
	A <= "0110";
	B <= "1110";
	
	wait for 100 ns;
	A <= "1111";
	B <= "1111";
	
	wait;
	
	end process;
	
	END;

Output Waveform



3.4 Carry select adder

Logic diagram



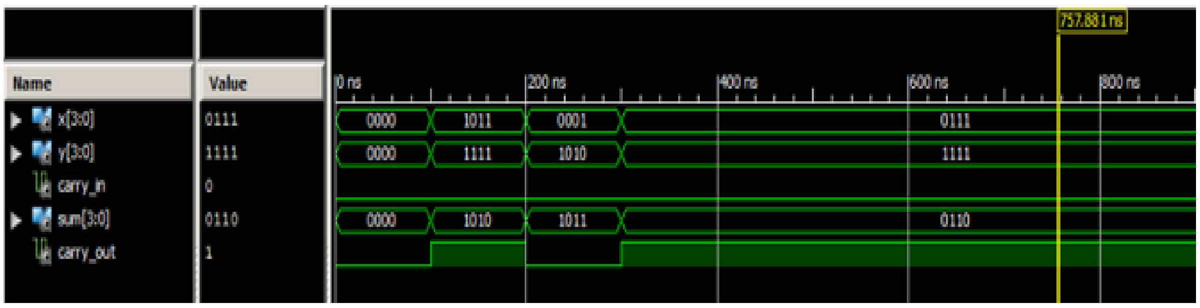
Verilog program

Structural code for CSA	Structural code for CSA

Module carry select (a,b,s,cin,cout);
Input [3:0]a,b;
Input cin;
Output cout;
Output [3:0]s;
Wire [3:0]s;
Wire Cout;
Wire
s1,c1,s2,c2,s3,c3,s4,c4,s11,c11,s22,c22,s33,c33,s44,c44;
fa x1(a[0],b[0],0,s1,c1);
fa x2(a[1],b[1],c1,s2,c2);
fa x3(a[2],b[2],c2,s2,c2);
fa x4(a[3],b[3],c3,s3,c3);
fa x5(a[0],b[0],1,s11,c11);
fa x6(a[1],b[1],c11,s22,c22);
fa x7(a[2],b[2],c22,s33,c33);
fa x8(a[3],b[3],c33,s44,c44);
mux x9(s1,s11,cin,s[0]);
mux x10(s2,s22,cin,s[1]);
mux x11(s3,33,cin,s[2]);
mux x12(s4,s44,cin,s[3]);
mux x12(c4,c44,cin,cout);
 	
endmodule
	`ENTITY Tb_carry_select_adder IS`
	`END Tb_carry_select_adder;`
	
	`ARCHITECTURE behavior OF`
	`Tb_carry_select_adder IS`
	
	`-- Component Declaration for the`
	`Unit Under Test (UUT)`
	
	`COMPONENT carry_select_adder`
	`PORT(`
	`X : IN std_logic_vector(3 downto`
	`0);`
	`Y : IN std_logic_vector(3 downto`
	`0);`
	`CARRY_IN : IN std_logic;`
	`SUM : OUT std_logic_vector(3`
	`downto 0);`
	`CARRY_OUT : OUT std_logic`
	`);`
	`END COMPONENT;`
	
	`--Inputs`
	`signal X : std_logic_vector(3`
	`downto 0) := (others => '0');`
	`signal Y : std_logic_vector(3`
	`downto 0) := (others => '0');`
	`signal CARRY_IN : std_logic :=`
	`'0';`
	
	`--Outputs`
	`signal SUM : std_logic_vector(3`
	`downto 0);`
	`signal CARRY_OUT : std_logic;`
	
	`BEGIN`

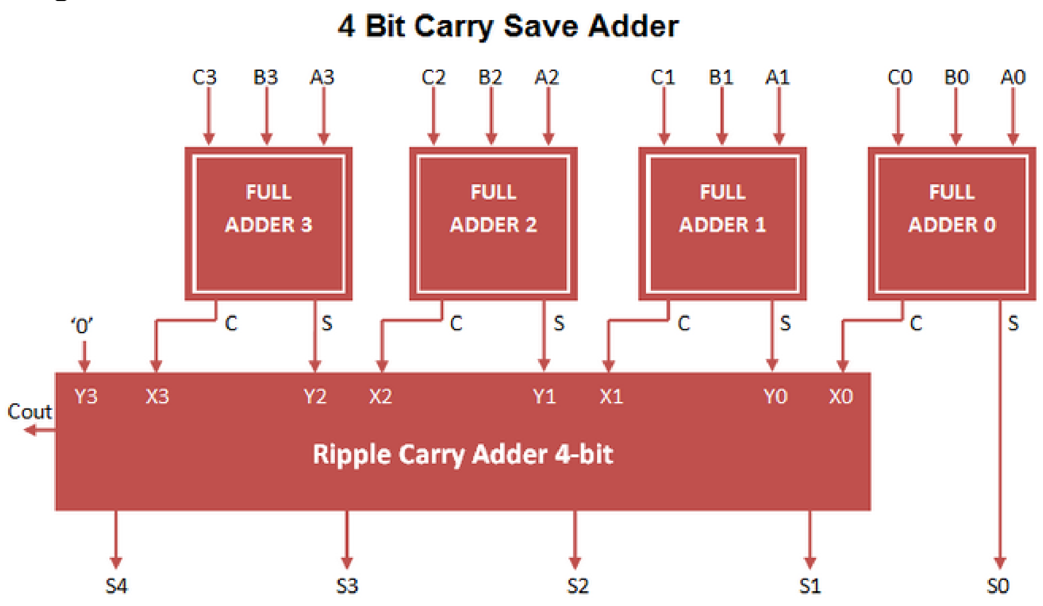
	-- Instantiate the Unit Under Test (UUT)
	uut: carry_select_adder PORT MAP (
	X => X,
	Y => Y,
	CARRY_IN => CARRY_IN,
	SUM => SUM,
	CARRY_OUT => CARRY_OUT
	);
	
	-- Stimulus process
	stim_proc: process
	begin
	-- hold reset state for 100 ns.
	wait for 100 ns;
	X <= "1011";
	Y <= "1111";
	
	wait for 100 ns;
	X <= "0001";
	Y <= "1010";
	
	wait for 100 ns;
	X <= "0111";
	Y <= "1111";
	wait;
	end process;
	
	END;

Output waveform of CSA



3.5 Carry save adder

Logic diagram



VHDL Program

Structural code for CSA

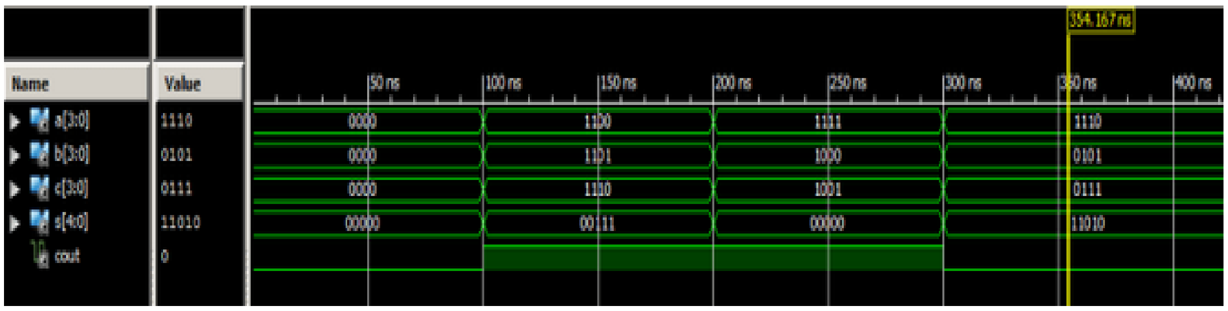
```vhdl
entity carry_save_adder is

Port ( A : in STD_LOGIC_VECTOR (3
downto 0);
B : in STD_LOGIC_VECTOR (3 downto 0);
C : in STD_LOGIC_VECTOR (3 downto 0);
S : OUT STD_LOGIC_VECTOR (4 downto 0);
Cout : OUT STD_LOGIC);
end carry_save_adder;

architecture Behavioral of
carry_save_adder is

component full_adder_vhdl_code
Port ( A : in STD_LOGIC;
B : in STD_LOGIC;
Cin : in STD_LOGIC;
S : out STD_LOGIC;
Cout : out STD_LOGIC);
end component;

-- Intermediate signal
signal X,Y: STD_LOGIC_VECTOR(3 downto
0);
signal C1,C2,C3: STD_LOGIC;

begin
-- Carry save adder block
FA1: full_adder_vhdl_code PORT
MAP(A(0),B(0),C(0),S(0),X(0));
FA2: full_adder_vhdl_code PORT
MAP(A(1),B(1),C(1),Y(0),X(1));
FA3: full_adder_vhdl_code PORT
MAP(A(2),B(2),C(2),Y(1),X(2));
FA4: full_adder_vhdl_code PORT
MAP(A(3),B(3),C(3),Y(2),X(3));

-- Ripple carry adder block
FA5: full_adder_vhdl_code PORT
MAP(X(0),Y(0),'0',S(1),C1);
FA6: full_adder_vhdl_code PORT
MAP(X(1),Y(1),C1,S(2),C2);
FA7: full_adder_vhdl_code PORT
MAP(X(2),Y(2),C2,S(3),C3);
FA8: full_adder_vhdl_code PORT
MAP(X(3),'0',C3,S(4),Cout);

end Behavioral;
```

Test bench for CSA

```vhdl
ENTITY Tb_carry_save IS
END Tb_carry_save;

ARCHITECTURE behavior OF
Tb_carry_save IS

-- Component Declaration for the
Unit Under Test (UUT)

COMPONENT carry_save_adder
PORT(
A : IN std_logic_vector(3 downto
0);
B : IN std_logic_vector(3 downto
0);
C : IN std_logic_vector(3 downto
0);
S : OUT std_logic_vector(4 downto
0);
Cout : OUT std_logic
);
END COMPONENT;

--Inputs
signal A : std_logic_vector(3
downto 0) := (others => '0');
signal B : std_logic_vector(3
downto 0) := (others => '0');
signal C : std_logic_vector(3
downto 0) := (others => '0');

--Outputs
signal S : std_logic_vector(4
downto 0);
signal Cout : std_logic;

BEGIN

-- Instantiate the Unit Under Test
(UUT)
uut: carry_save_adder PORT MAP (
A => A,
B => B,
C => C,
S => S,
Cout => Cout
);

-- Stimulus process
stim_proc: process
begin
-- hold reset state for 100 ns.
wait for 100 ns;
A <= "1100";
B <= "1101";
C <= "1110";

wait for 100 ns;
A <= "1111";
B <= "1000";
C <= "1001";
```

```
wait for 100 ns;
A <= "1110";
B <= "0101";
C <= "0111";

wait;
end process;

END;
```

Output waveform



RESULT

Thus the Ripple carry,Carry select and Carry save Adders were implemented using XILINX .

EXP: Design of Multiplexers and Demultiplexers

4.1 Introduction

The purpose of this experiment is to write and simulate a VERILOG program for Multiplexers and Demultiplexers.

4.2 Software tools Requirement:

Equipments:

Computer with Modelsim Software

Specifications:

HP Computer P4 Processor - 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Softwares: Modelsim - 5.7c, Xilinx - 6.1i.

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -><filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

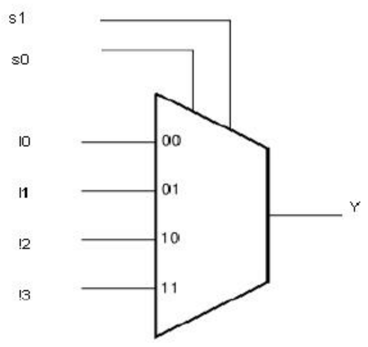
STEP 9: View -> Signals

STEP 10: Select values -> Edit -> Force -> input values

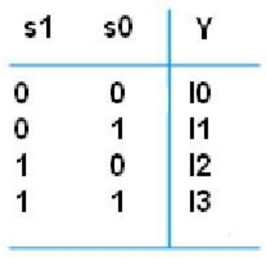
STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again

4.3 Logic Diagram

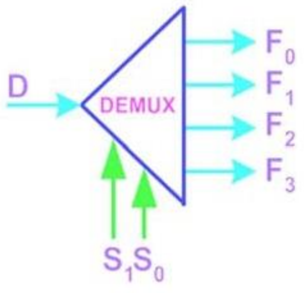


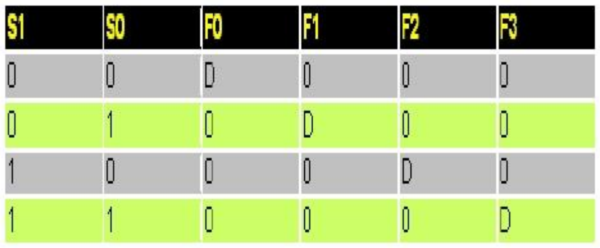
4:1 Multiplexer Block diagram



s1	s0	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Function Table



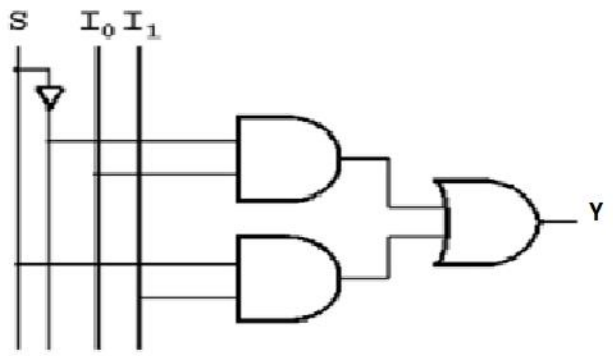


S1	S0	F0	F1	F2	F3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

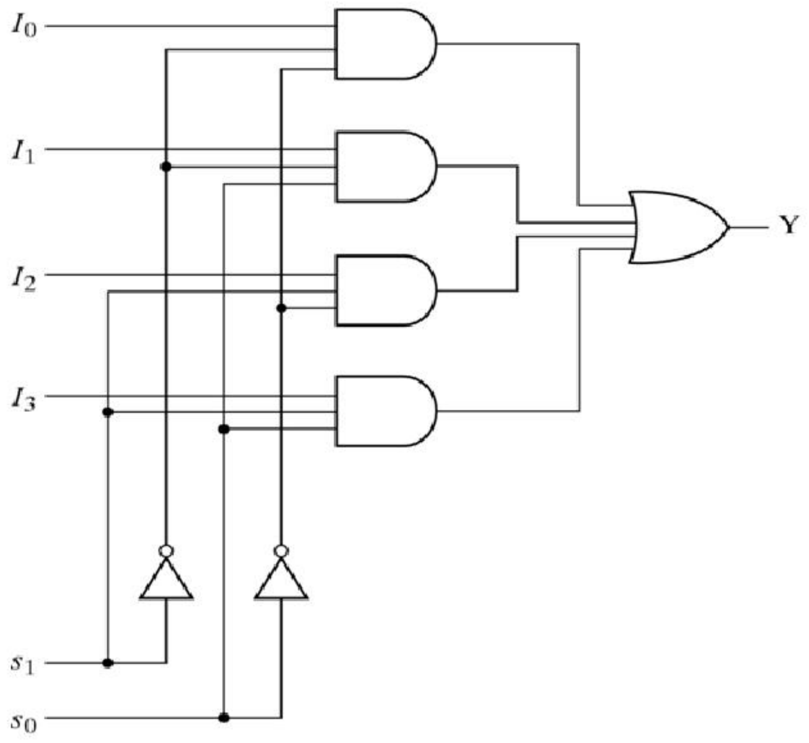
1:4 Demux Symbol

Function Table

Logic Diagram



2:1 Multiplexer



4:1 Multiplexer

4.4 Pre lab Questions

1. Define mux and demux. Write
2. their applications.
3. What is the relationship b/w input lines and select lines.
4. Design 4:1 mux and 1:4 demux.
5. Write brief notes on case statement.

VERILOG Program

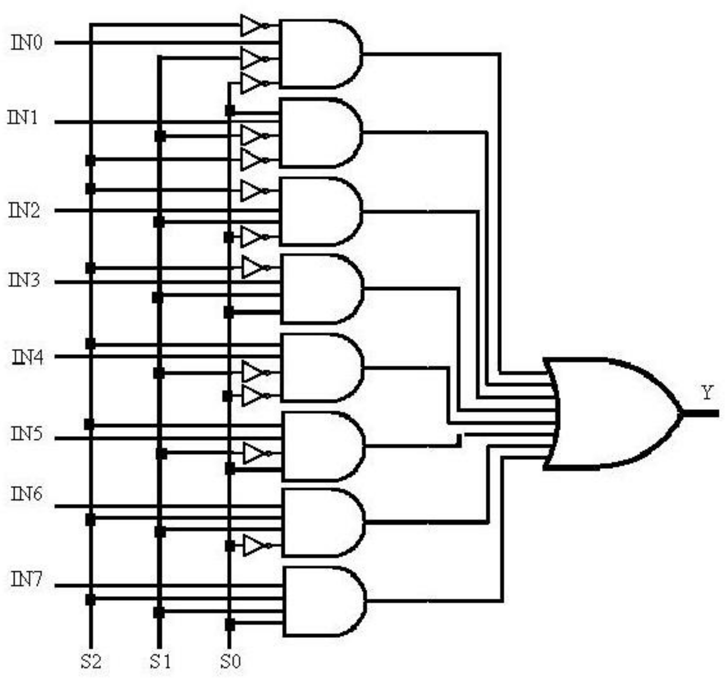
Multiplexers 2:1 MUX

Structural Model	**Dataflow Model**	**BehaviouralModel**
module mux21str(i0,i1,s,y);	module mux21df(i0,i1,s,y);	module mux21beh(i0,i1,s,y);
input i0,i1,s;	input i0,i1,s;	input i0,i1,s;
output y;	output y;	output y;
wire net1,net2,net3;	assign y =(i0&(~s))|(i1&s);	reg y;
not g1(net1,s);	endmodule	always@(i0,i1)
and g2(net2,i1,s);		begin
and g3(net3,i0,net1);		if(s==0) y=i1;
or g4(y,net3,net2);		if(s==1)y=i0;
endmodule		end
		endmodule

4:1 MUX

Structural Model	**Dataflow Model**	**BehaviouralModel**
module	module	module mux41beh(in,s,y);
mux41str(i0,i1,i2,i3,s0,s1,y);	mux41df(i0,i1,i2,i3,s0,s1,y);	output y ;
input i0,i1,i2,i3,s0,s1;	input i0,i1,i2,i3,s0,s1;	input [3:0] in ;
wire a,b,c,d;	output y;	input [1:0] s ;
output y;	assign	reg y;
and g1(a,i0,s0,s1);	y=((i0&(~(s0))&(~(s1)))|	always @ (in,s)
and g2(b,i1,(~s0),s1);	(i1&(~(s0))&s1)|	begin
and g3(c,i2,s0,(~s1));	|(i2&s0&(~(s1)))|	if (s[0]==0&s[1]==0)
and g4(d,i3,(~s0),(~s1));	(i3&s0&s1);	y = in[3];
or(y,a,b,c,d);	endmodule	else if (s[0]==0&s[1]==1)
endmodule		y = in[2];
		else if (s[0]l==1&s[1]==0)
		y = in[1];
		else
		y = in[0];
		end
		endmodule

Logic Diagram



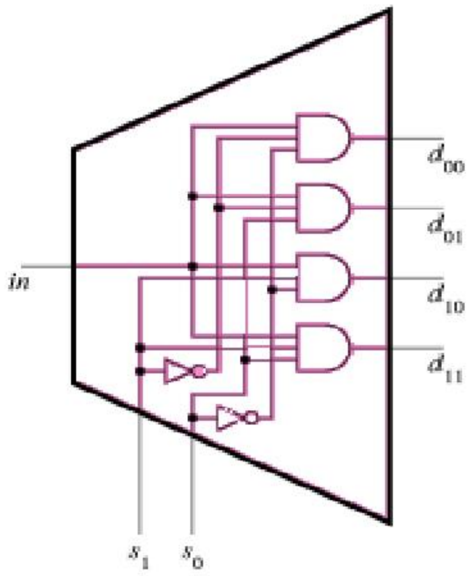
8:1 Multiplexer

VERILOG Program

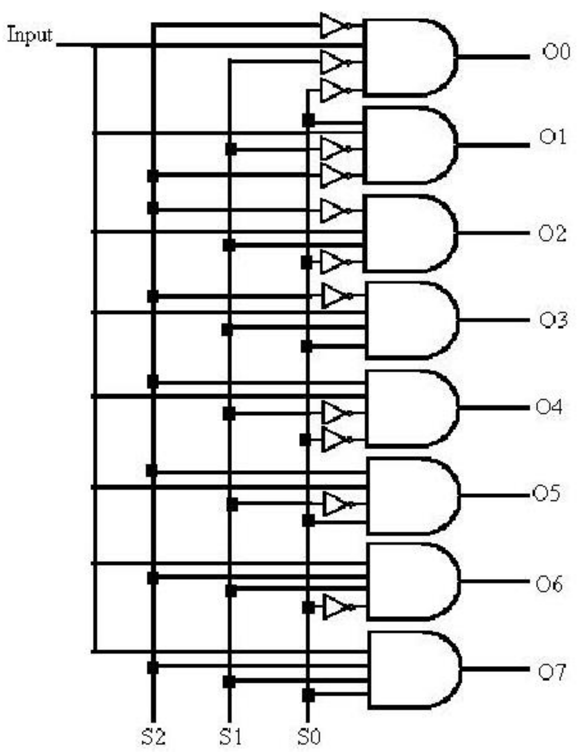
8:1 MUX

Structural Model	Dataflow Model	BehaviouralModel
modulemux81str(i0,i1,i2,i3,i4,i5,i6,i7,s0,s1,s2,y);	modulemux81df(y,i,s);	modulemux81beh(s,i0,i1,i2,i3,i4,i5,i6,i7,y);
input i0,i1,i2,i3,i4,i5,i6,i7,s0,s1,s2;	output y;	input [2:0] s;
wire a,b,c,d,e,f,g,h;	input [7:0] i;	input i0,i1,i2,i3,i4,i5,i6,i7;
output y;	input [2:0] s;	regy;
and g1(a,i7,s0,s1,s2);	wire se1;	always@(i0,i1,i2,i3,i4,i5,i6,i7,s) begin
and g2(b,i6,(~s0),s1,s2);	assign se1=(s[2]*4)|(s[1]*2)|(s[0]);	case(s) begin
and g3(c,i5,s0,(~s1),s2);		3'd0:MUX_OUT=i0;
		3'd1:MUX_OUT=i1;
and g4(d,i4,(~s0),(~s1),s2);	assign y=i[se1];	3'd2:MUX_OUT=i2;
and g5(e,i3,s0,s1,(~s2));	endmodule	3'd3:MUX_OUT=i3;
		3'd4:MUX_OUT=i4;
and g6(f,i2,(~s0),s1,(~s2));		3'd5:MUX_OUT=i5;
and g7(g,i1,s0,(~s1),(s2));		3'd6:MUX_OUT=i6;
and g8(h,i0,(~s0),(~s1),(~s2));		3'd7:MUX_OUT=i7;
		endcase
or(y,a,b,c,d,e,f,g,h);		end
endmodule		endmodule

Logic Diagram



1:4 Demultiplexer



1:8 Demultiplexer

VERILOG Program

1:4 DEMUX

Structural Model	Dataflow Model	BehaviouralModel

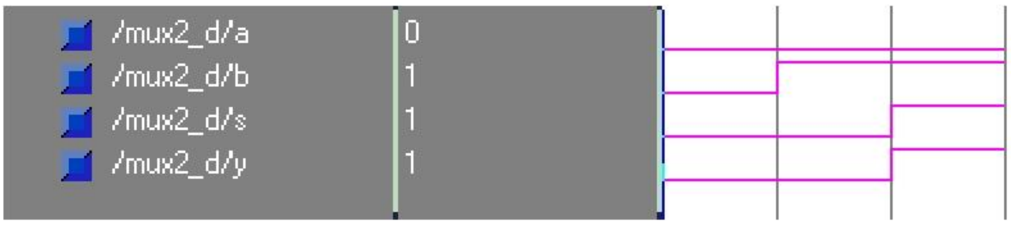
module
demux14str(in,d0,d1,d2,d3,s0,s1);
output d0,d1,d2,d3;
input in,s0,s1;
and g1(d0,in,s0,s1);
and g2(d1,in,(~s0),s1);
and g3(d2,in,s0,(~s1));
and g4(d3,in,(~s0),(~s1));
endmodule
	module demux14df(
	in,d0,d1,d2,d3,s0,s1);
	output d0,d1,d2,3;
	input in,s0,s1;
	assign s0 = in & (~s0) & (~s1);
	assign d1= in & (~s0) & s1;
	assign d2= in & s0 & (~s1);
	assign d3= in & s0 & s1;
	endmodule
		module demux14beh(
		din,sel,dout);
		output [3:0] dout ;
		reg [3:0] dout ;
		input din ;
		wire din ;
		input [1:0] sel ;
		wire [1:0] sel ;
		always @ (din or sel) begin
		case (sel)
		0 : dout = {din,3'b000};
		1 : dout = {1'b0,din,2'b00}; 2
		: dout = {2'b00,din,1'b0};
		default : dout = {3'b000,din};
		endcase
		end
		endmodule

1:8 DEMUX

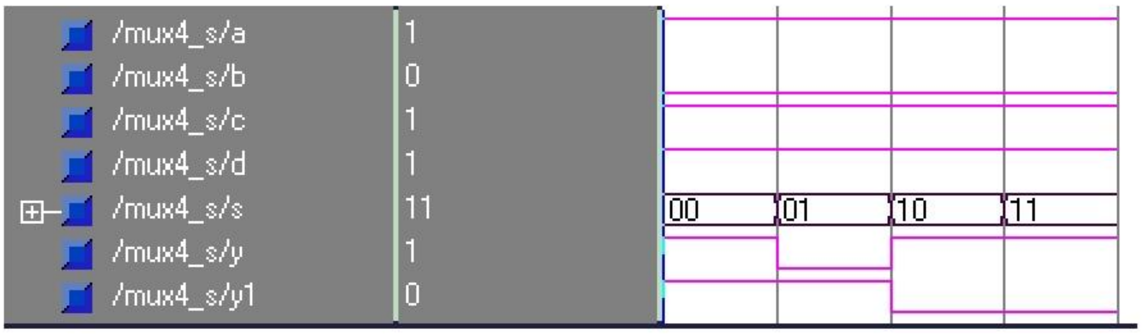
Structural Model	**Dataflow Model**	**BehaviouralModel**
module	module	module demux18beh(i,
demux18str(in,s0,s1,s2,d0,d1,d2	demux18df(in,s0,s1,s2,i0,d1,d2,d3,d4,d5	sel, y);
,d3,d4,d5,d6,d7);	,d6,d7);	
		input i;
input in,s0,s1,s2;	input in,s0,s1,s2;	
		input [2:0] sel;
output d0,d1,d2,d3,d4,d5,d6,d7;	output d0,d1,d2,d3,d4,d5,d6,d7;	
		output [7 :0] y ;
and g1(d0,in,s0,s1,s2);	assign d0 = in & s0 & s1 & s2;	
		reg [7:0] y;
and g2(d1,in,(~s0),s1,s2);	assign d1 = in & (~s0) & s1 & s2;	
		always@(i,sel)
and g3(d2,in,s0,(~s1),s2);	assign d2 = in & s0 & (~s1) & s2;	
		begin
and g4(d3,in,(~s0),(~s1),s2);	assign d3 = in & (~s0) &(~s1) & s2;	
		y=8'd0;
and g5(d4,in,s0,s1,(~s2));	assign d4 = in & s0 & s1 & (~s2);	
		case(sel)
and g6(d5,in,(~s0),s1,(~s2));	assign d5 = in & (~s0) & s1 & (~s2);	
		3'd0:y[0]=i;
and g7(d6,in,s0,(~s1),(~s2));	assign d6 = in & s0 & (~s1) & (~s2);	
		3'd1:y[1]=i;
and g8(d7,in,(~s0),(~s1),(~s2));	assign d7 = in & (~s0) & (~s1) & (~s2);	
		3'd2:y[2]=i;
endmodule	endmodule	
		3'd3:y[3]=i;
		3'd4:y[4]=i;
		3'd5:y[5]=i;
		3'd6:y[6]=i;
		default:y[7]=i;
		endcase
		end
		endmodule

Output Wave forms:

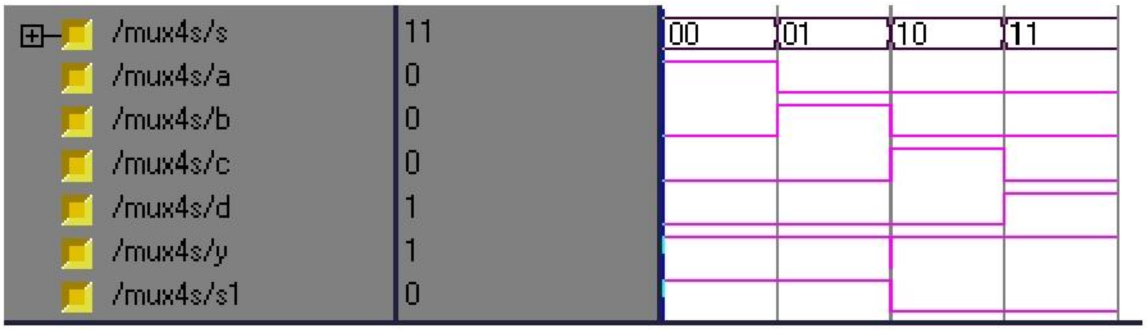
2 X 1 MUX:



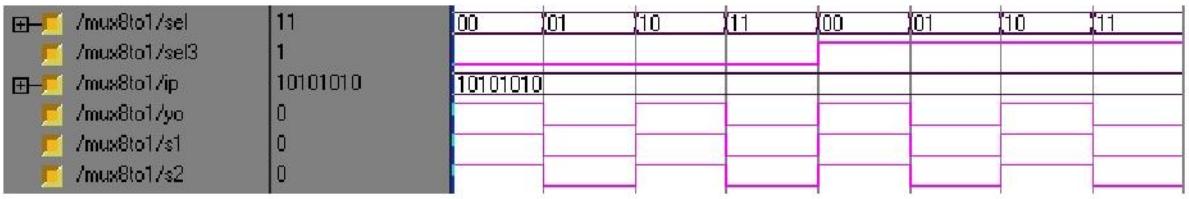
4 X 1 MUX **DATAFLOW MODELING**



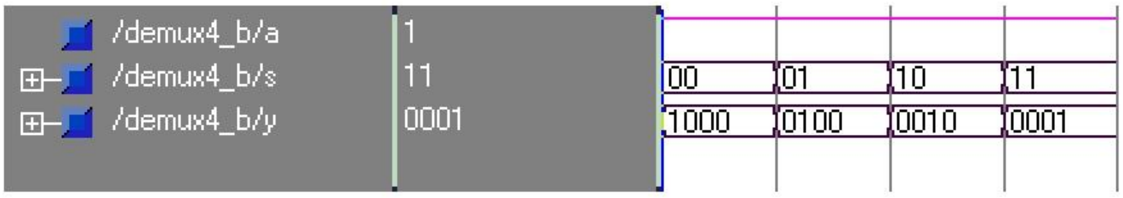
4 X 1 MUX **STRUCTURAL MODELING**



MUX 8:1 Using 4:1 & 2:1



1 TO 4 DEMUX BEHAVIOURAL MODELING



8 bit multiplier

VHDL Program

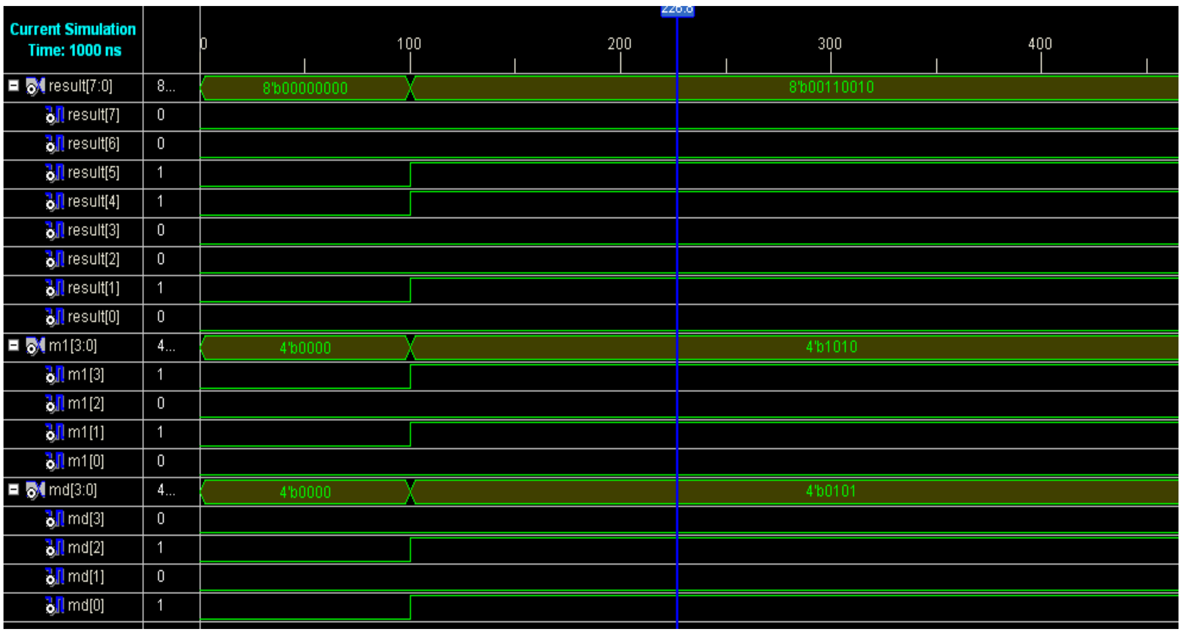
Structural code for multiplier

```
module multipliermod(a, b, out);
input [4:0] a;
input [4:0] b;
output [9:0] out;
assign out=(a*b);
endmodule
```

<u>TEST BENCH</u>

```
module multipliert_b;
reg [4:0] a;
reg [4:0] b;
wire [9:0] out;
multipliermod uut (.a(a),.b(b),.out(out) );
initial begin
#10 a=4’b1000;b=4’b0010;
#10 a=4’b0010;b=4’b0010;
#10 a=4’b0100;b=4’b0100;
#10 a=4’b1000;b=4’b0001;
#10$stop;
end
endmodule
```

Output waveform



RESULT

Thus the multiplexers, demultiplexers and 8 bit Multiplier were simulated using Xilinx.

EXP 5: Design of Flip Flops

5.1 Introduction

The purpose of this experiment is to introduce you to the basics of flip-flops. In this lab, you will test the behavior of several flip-flops and you will connect several logic gates together to create simple sequential circuits.

5.2 Software tools Requirement

Equipments:

Computer with Modelsim Software

Specifications:

HP Computer P4 Processor - 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Softwares: Modelsim - 5.7c, Xilinx - 6.1i.

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -><filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

STEP 9: View -> Signals

STEP 10: Select values -> Edit -> Force -> input values

STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again

5.3 Flip-Flops Logic diagram and their properties

Flip-flops are synchronous bitable devices. The term synchronous means the output changes state only when the clock input is triggered. That is, changes in the output occur in synchronization with the clock.

A flip-flop circuit has two outputs, one for the normal value and one for the complement value of the stored bit. Since memory elements in sequential circuits are usually flip-flops, it is worth summarizing the behavior of various flip-flop types before proceeding further.

All flip-flops can be divided into four basic types: SR, JK, D and T. They differ in the number of inputs and in the response invoked by different value of input signals. The four types of flip-flops are defined in the Table 5.1. Each of these flip-flops can be uniquely described by its graphical symbol, its characteristic table, its characteristic equation or excitation table. All flip-flops have output signals Q and Q'.

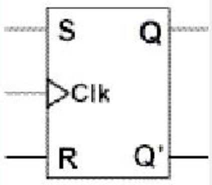
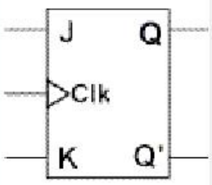
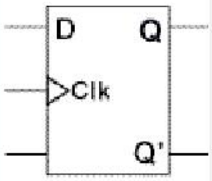
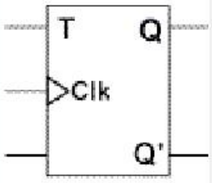
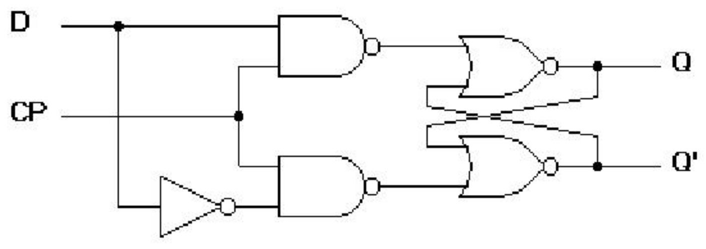
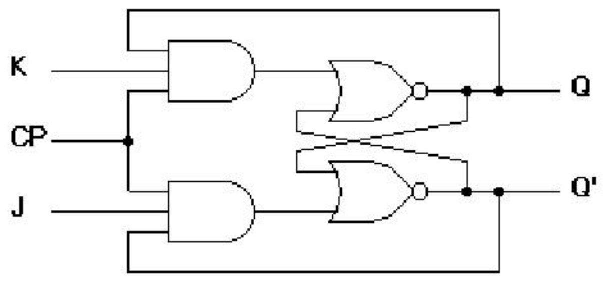
Flip-Flop Name	Flip-Flop Symbol	Characteristic Table	Characteristic Equation	Excitation Table
SR		S | R | Q(next) 0 | 0 | Q 0 | 1 | 0 1 | 0 | 1 1 | 1 | ?	Q(next) = S + R'Q SR = 0	Q Q(next) S | | R 0 | 0 | 0 | X 0 | 1 | 1 | 0 1 | 0 | 0 | 1 1 | 1 | X0 |
JK		J | K | Q(next) 0 | 0 | Q 0 | 1 | 0 1 | 0 | 1 | 1 | Q'	Q(next) = JQ' + K'Q	Q Q(next) J | | K 0 | 0 | 0 | X 0 | 1 | 1 | X 1 | 0 | X1 | 1 1 | 1 | X0 |
D		D | Q(next) 0 | 0 1 | 1	Q(next) = D	Q | Q(next) D | 0 | 0 | 0 0 | 1 | 1 1 | 0 | 0 1 | 1 | 1
T		T | Q(next) 0 | Q 1 | Q'	Q(next) = TQ' + T'Q	Q | Q(next) T | 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 0

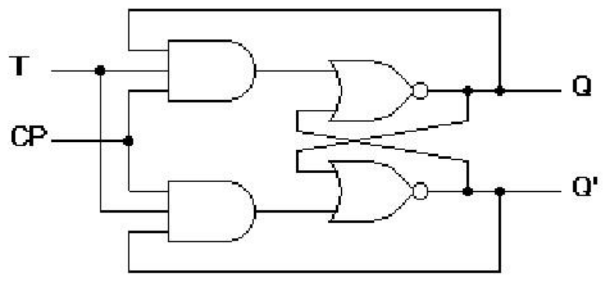
Table 5.3 Flip-flops and their properties



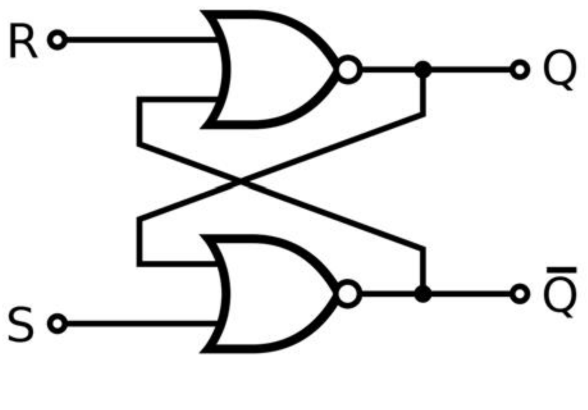
D- Flip Flop



JK Flip Flop



T Flip Flop



T Flip Flop

5.4 Pre-lab Questions

1. Describe the main difference between a gated S-R latch and an edge-triggered S-R flip-flop.

2. How does a JK flip-flop differ from an SR flip-flop in its basic operation?

3. Describe the basic difference between pulse-triggered and edge-triggered flip-flops.

4. What is use of characteristic and excitation table?

5. What are synchronous and asynchronous circuits?

6. How many flip flops due you require storing the data 1101?

Verilog prohram

S-R Flip Flop

Behavioral Modelling	**Structural Modelling**	**Dataflow Modelling**

modulesr_df (s, r, q, q_n);
input s, r;
output q, q_n;
assignq_n = ~(s | q);
assign q = ~(r | q_n);
endmodule
	module sr_st(s,r,q,q_n);
	input s, r;
	output q, q_n;
	or g1(q_n,~s,~q);
	or g2(q,~r,~q_n);
	endmodule
		module sr_beh(s,r,q,q_n);
		input s, r;
		output q, q_n;
		regq, q_n;
		always@(s,r)
		begin
		q,n = ~(s|q);
		assign q = ~(r | q_n);
		endmodule

T Flip Flop

Behavioral Modelling	**Structural Modelling**	**Dataflow Modelling**

module t_beh(q,q1,t,c);
output q,q1;
inputt,c;
reg q,q1;
initial
begin
q=1'b1;
q1=1'b0;
end
always @ (c)
begin
if(c)
begin
	module t_st(q,q1,t,c);
	output q,q1;
	input t,c;
	wire w1,w2;
	assign w1=t&c&q;
	assign w2=t&c&q1;
	assign q=~(w1|q1);
	assign q1=~(w2|q);
	endmodule
		module t_df(q,q,1,t,c);
		output q,q1;
		input t,c;
		and g1(w1,t,c,q);
		and g2(w2,t,c,q1);
		nor g3(q,w1,q1);
		nor g4(q1,w2,q);
		endmodule

if (t==1'b0) begin
q=q; q1=q1; end
else begin q=~q;
q1=~q1; end
end
end
end module		

D flip flop

Behavioral Modelling	**Structural Modelling**	**Dataflow Modelling**

Module dff_async_reset(data, clk, reset ,q);
input data, clk, reset ;
output q;
reg q;
always @ (posedgeclk or negedge reset)
if (~reset) begin
    q <= 1'b0;
end
else begin
   q <= data;
end
	module dff_df(d,c,q,q1);
	input d,c;
	output q,q1;
	assign w1=d&c;
	assign w2=~d&c;
	q=~(w1|q1);
	q1=~(w2|q);
	endmodule
		module dff_df(d,c,q,q1);
		input d,c;
		output q,q1;
		and g1(w1,d,c);
		and g2(w2,~d,c);
		nor g3(q,w1,q1);
		nor g4(q1,w2,q);
		endmodule

JK flip flop

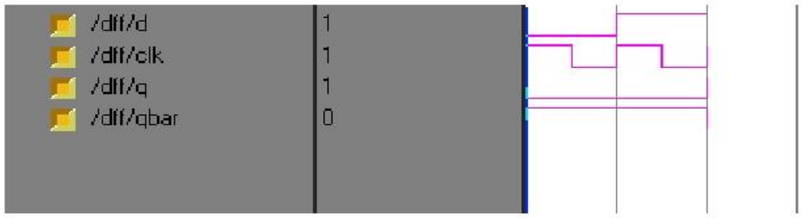
Behavioral Modelling	**Structural Modelling**	**Dataflow Modelling**

module jk(q,q1,j,k,c);
output q,q1;
input j,k,c;
reg q,q1;
initial begin q=1'b0; q1=1'b1; end
always @ (posedge c)
  begin
            case({j,k})
                          {1'b0,1'b0}:begin
q=q; q1=q1; end
                          {1'b0,1'b1}: begin
q=1'b0; q1=1'b1; end
                          {1'b1,1'b0}:begin
q=1'b1; q1=1'b0; end
                          {1'b1,1'b1}: begin
q=~q; q1=~q1; end
            endcase
	module jkflip_df (j,k,q,qn);
	input j,k,q;
	output qn;
	wire w1,w2;
	assign w1=~q;
	assign w2=~k;
	assign qn=(j & w1 | w2 & q);
	endmodule
		module jkflip_st(j,k,q,qn);
		input j,k,q;
		output qn;
		and g1(w1,j,~q);
		and g2(w2,~k,q);
		or g3(qn,w1,w2);
		endmodule

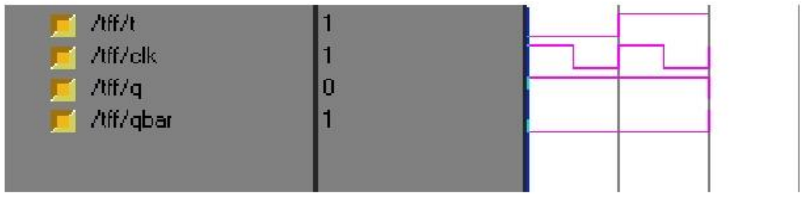
end		

Output Waveforms

D flip flop



T flip flop



RESULT

Thus the VHDL code for flip-flop was implemented and simulated using Xilinx.

EXP : Design of Counters

6.1 Introduction

The purpose of this experiment is to introduce the design of Synchronous Counters, asynchronous,ring,johnson up/down Counter.

6.2 Software tools Requirement

Equipments:

Computer with Modelsim Software

Specifications:

HP Computer P4 Processor - 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Softwares: Modelsim - 5.7c, Xilinx - 6.1i.

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -><filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

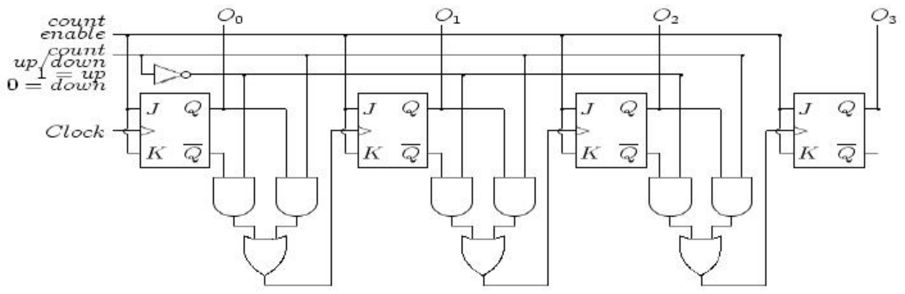
STEP 9: View -> Signals

STEP 10: Select values -> Edit -> Force -> input values

STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again

6.3 Logic Diagram



Updown Counter

Verilog program

Up down counter

Verilog code

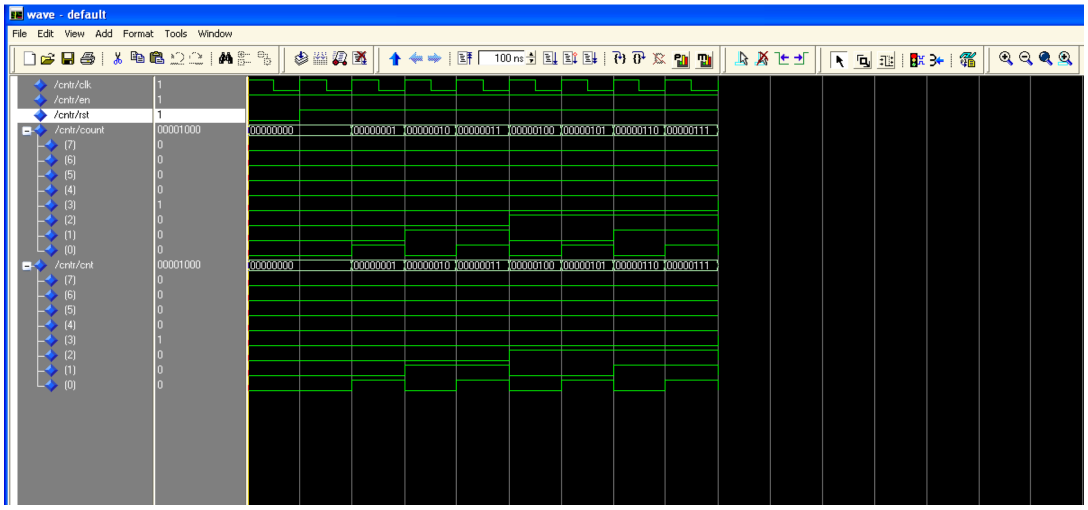
```
moduleupdown(out,clk,reset,updown);
output [3:0]out;
inputclk,reset,updown;
reg [3:0]out;
always @(posedgeclk)
if(reset) begin
out<= 4'b0;
end else if(updown) begin
out<=out+1;
end else begin
out<=out-1;
end
endmodule
```

Asynchronous counter & Synchronous Counters

VHDL code	VHDL code

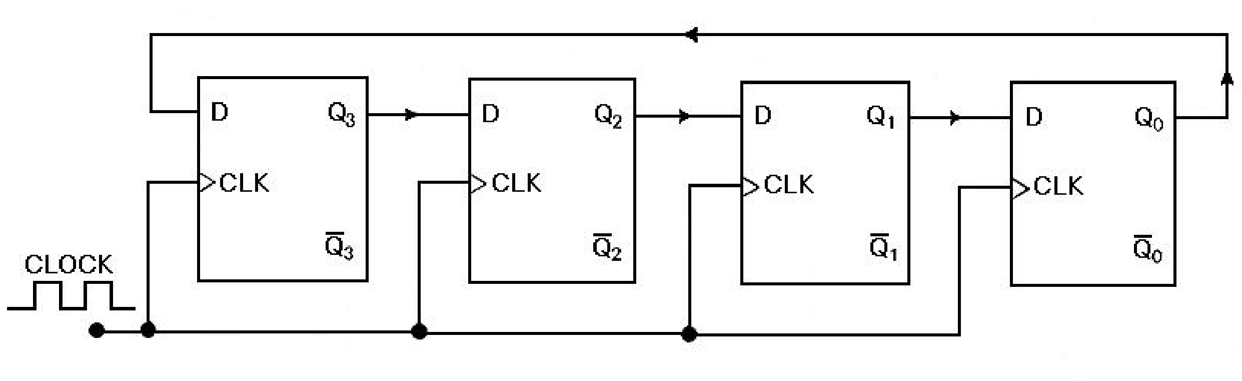
entity counter_8 is
port(clk,en,rst:in std_logic;
count:out std_logic_vector(7 downto 0));
end counter_8;
architecture beh of counter_8 is
signal cnt:std_logic_vector(7 downto 0);
begin
process(clk,en,rst)
begin
if(rst='0')then
cnt<=(others=>'0');
elsif(clk'event and clk='1')then
if(en='1')then
cnt<=cnt+'1';end if;end if;
end process;count<=cnt; end beh;
	entity counter is
	  port(C, S : in std_logic;
	        Q : out std_logic_vector(3 downto 0));
	end counter;
	architecture archi of counter is
	  signal tmp: std_logic_vector(3 downto 0);
	  begin
	    process (C)
	      begin if (C'event and C='1') then
	  if (S='1') then
	  tmp <= "1111"; else
	  tmp <= tmp - 1; end if; end if;
	   end process; Q <= tmp;
	end archi;

Output Waveform

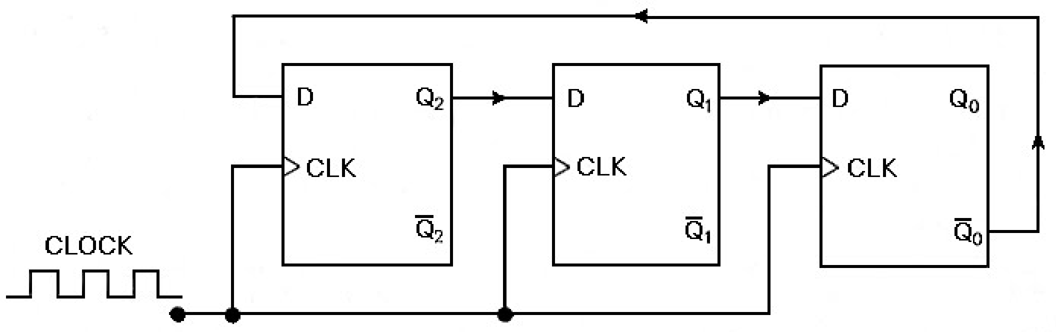


Ring counter & Johnson counter

Logic diagram



Ring counter



Johnson counter

VHDL program

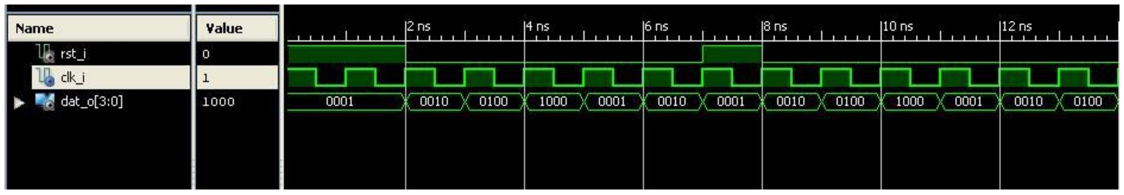
Ring counter	**Johnson counter**

entity ring_counter is
'0');
begin
 	
DAT_O <= temp;
process(CLK_I)
begin
    if(rising_edge(CLK_I)) then
        if (RST_I = '1') then
            temp <= (0=> '1', others => '0');
	entity johnson_counter is
	port (
	        DAT_O : out unsigned(3 downto 0);
	        RST_I : in std_logic;
	        CLK_I : in std_logic
	        );
	end johnson_counter;
	architecture Behavioral of johnson_counter is
	signal temp : unsigned(3 downto 0):=(others =>
	'0');

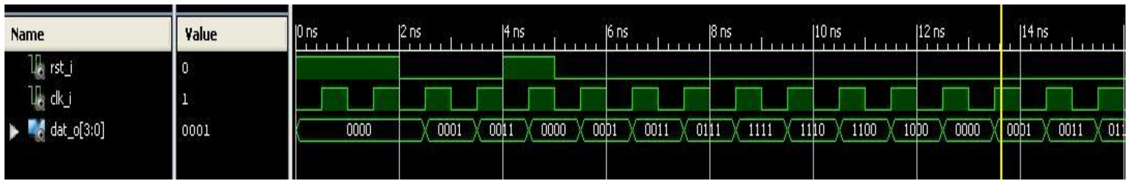
```
        else
            temp(1) <= temp(0);
            temp(2) <= temp(1);
            temp(3) <= temp(2);
            temp(0) <= temp(3);
        end if;
    end if;
end process;
end Behavioral;
```

```
begin
DAT_O <= temp;
process(CLK_I)
begin
    if( rising_edge(CLK_I) ) then
        if (RST_I = '1') then
            temp <= (others => '0');
        else
            temp(1) <= temp(0);
            temp(2) <= temp(1);
            temp(3) <= temp(2);
            temp(0) <= not temp(3);
        end if;
    end if;
end process;

end Behavioral;
```

Output waveform



Ring counter



Johnson counter

RESULT

Thus the Counter VHDL code were implemented and simulated by using Xilinx project navigator

EXP: Design of State machines

1.1 Introduction

The purpose of this experiment is to simulate the behavior of Moore and Mealy model

1.2 Software tools Requirement

Equipments:

Computer with Modelsim Software

Specifications:

HP Computer P4 Processor - 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Softwares: Modelsim - 5.7c, Xilinx - 8.1i.

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -><filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

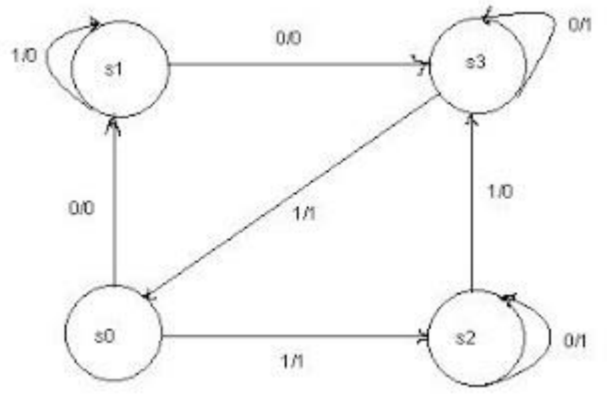
STEP 9: View -> Signals

STEP 10: Select values -> Edit -> Force -> input values

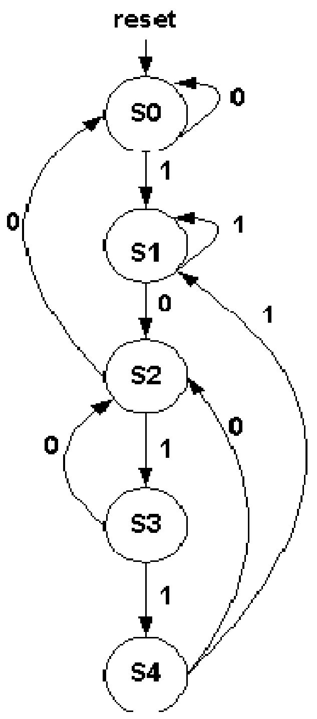
STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again

Mealy model



Moore model



State Transition Diagram

Program

VHDL Moore model	**Verilog Mealy model**

entity mealy is
port (clk : in std_logic;
      reset : in std_logic;
      input : in std_logic;
      output : out std_logic
  );
end mealy;
 	
architecture behavioral of mealy is
type state_type is (s0,s1,s2,s3); --type of state
	module seq_dect
	(
	    input clk, data_in, reset,
	    output reg data_out
	);
	    // Declare state register
	    reg [2:0]state;
	    // Declare states
	    parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3, S4 = 4;

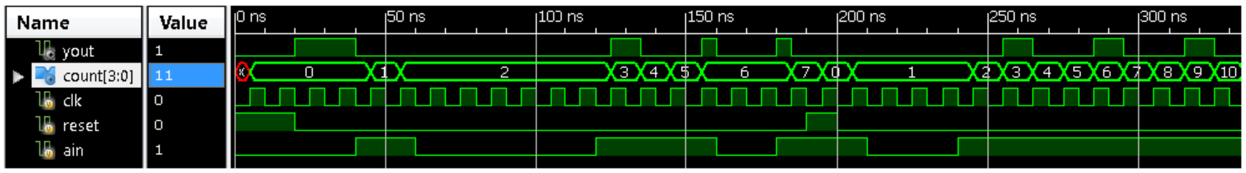
```
machine.
signal current_s,next_s: state_type;  --current
and next state declaration.

begin

process (clk,reset)
begin
 if (reset='1') then
  current_s <= s0;  --default state on reset.
elsif (rising_edge(clk)) then
  current_s <= next_s;   --state change.
end if;
end process;
--state machine process.
process (current_s,input)
begin
  case current_s is
     when s0 =>        --when current state is "s0"
     if(input ='0') then
      output <= '0';
      next_s <= s1;
    else
      output <= '1';
      next_s <= s2;
     end if;
     when s1 =>;        --when current state is
"s1"
    if(input ='0') then
      output <= '0';
      next_s <= s3;
    else
      output <= '0';
      next_s <= s1;
    end if;

    when s2 =>       --when current state is "s2"
    if(input ='0') then
      output <= '1';
      next_s <= s2;
    else
      output <= '0';
      next_s <= s3;
    end if;

  when s3 =>         --when current state is "s3"
    if(input ='0') then
      output <= '1';
      next_s <= s3;
```

```

    // Determine the next state
    always @ (posedge clk or posedge reset)
begin
        if (reset)
            state <= S0;
        else
            case (state)
                S0:
                    if (data_in)
                        state <= S1;
                    else
                        state <= S0;
                S1:
                    if (data_in)
                        state <= S1;
                    else
                        state <= S2;
                S2:
                    if (data_in)
                        state <= S3;
                    else
                        state <= S2;
                S3:
                    if (data_in)
                        state <= S4;
                    else
                        state <= S2;
                S4:
                    if (data_in)
                        state <= S1;
                    else
                        state <= S2;
            endcase // case (state)
    end // always @ (posedge clk or posedge
reset)
    // Output depends only on the state
    always @ (state) begin
        case (state)
            S0:
                data_out = 1'b0;
            S1:
                data_out = 1'b1;
            S2:
                data_out = 1'b0;
            S3:
                data_out = 1'b1;
            S4:
                data_out = 1'b1;
    else
      output <= '1';
      next_s <= s0;
    end if;
  end case;
end process;

end behavioral;
```

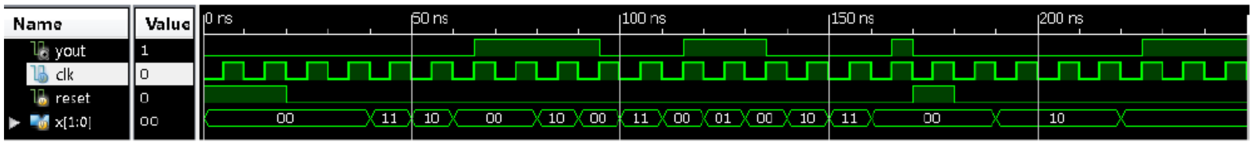
```
            default:
                data_out = 1'b0;
        endcase // case (state)
    end // always @ (state)

endmodule // moore_mac
```

Output waveform

Mealy model



Moore model



RESULT

Thus the Moore and Mealy state machines code were implemented and simulated by using Xilinx project navigator

EXP: Design of CMOS inverter

8.1 Introduction

To perform the functional verification of the CMOS Inverter through schematic entry

8.2 Software tools Requirement

S-Edit using cadance Tool

HP Computer P4 Processor - 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Algorithm

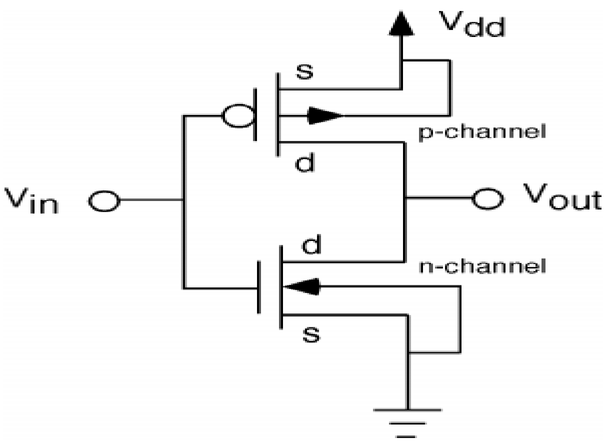
STEP 1: Draw the schematic of CMOS Inverter using S-edit.

STEP 2: Perform Transient Analysis of the CMOS Inverter.

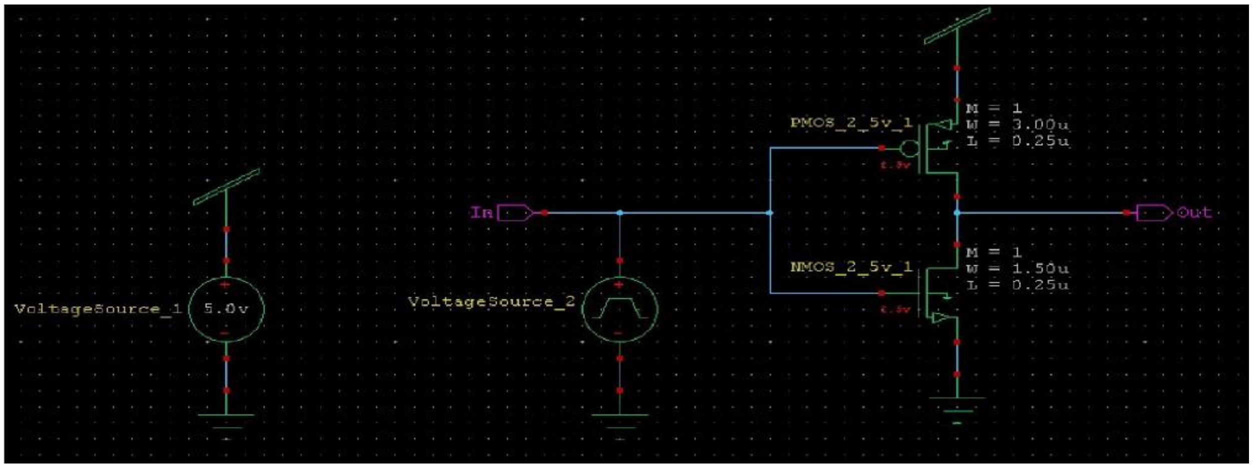
STEP 3: Obtain the output waveform from W-edit

STEP 4: Obtain the spice code using T-edit.

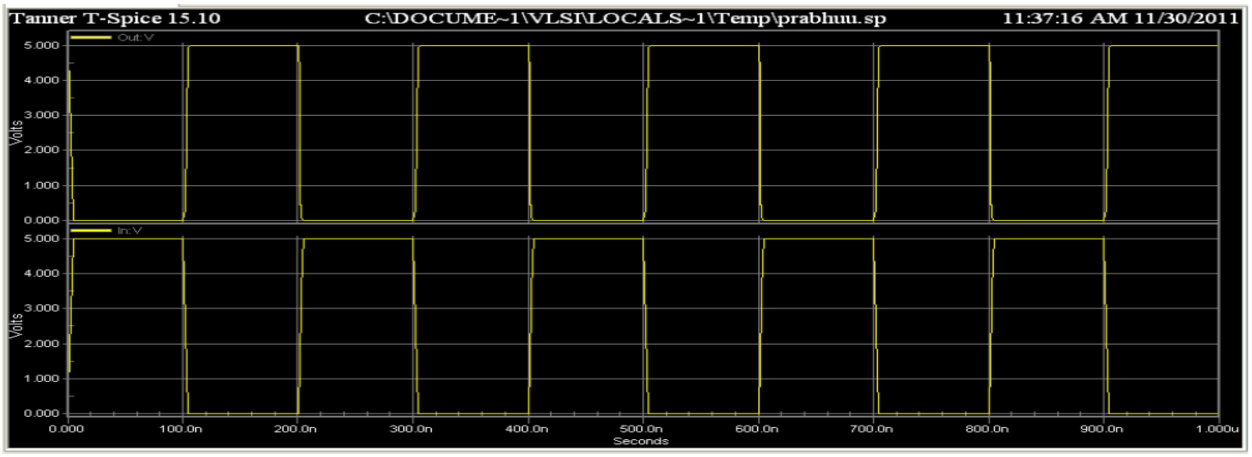
Circuit diagram of CMOS inverter



CIRCUIT USING TANNER



SIMULATED WAVEFORM:



RESULT

Thus the functional verification of the CMOS Inverter through schematic entry.and the output also verified successfully.

EXP: Design of Differential Amplifier

8.1 Introduction

To calculate the gain, bandwidth and CMRR of a differential amplifier through schematic entry.

8.2 Software tools Requirement

S-Edit using cadance Tool

HP Computer P4 Processor - 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Algorithm

STEP 1: Draw the schematic of differential amplifier using S-edit and generate the symbol.

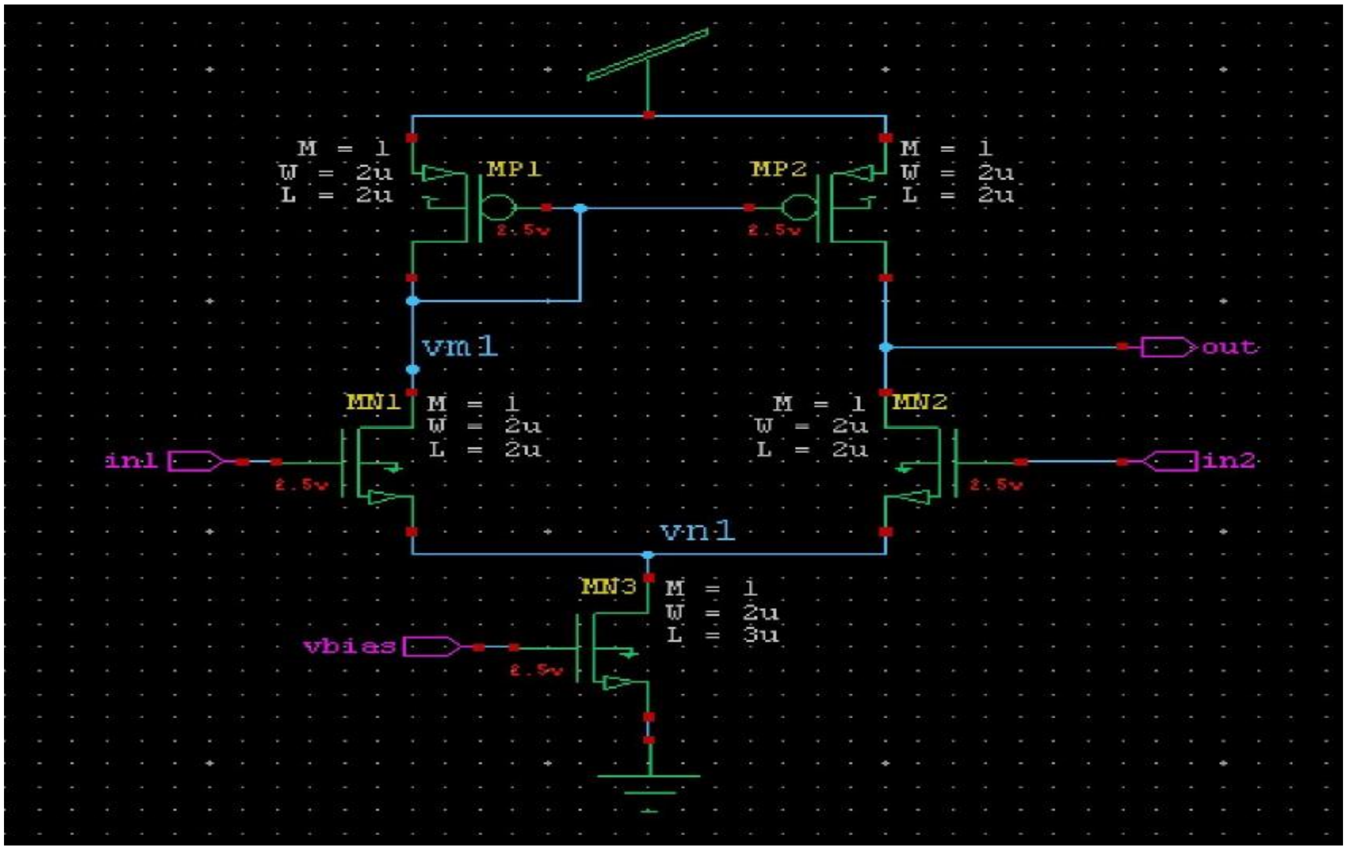
STEP 2: Draw the schematic of differential amplifier circuit using the generated symbol.

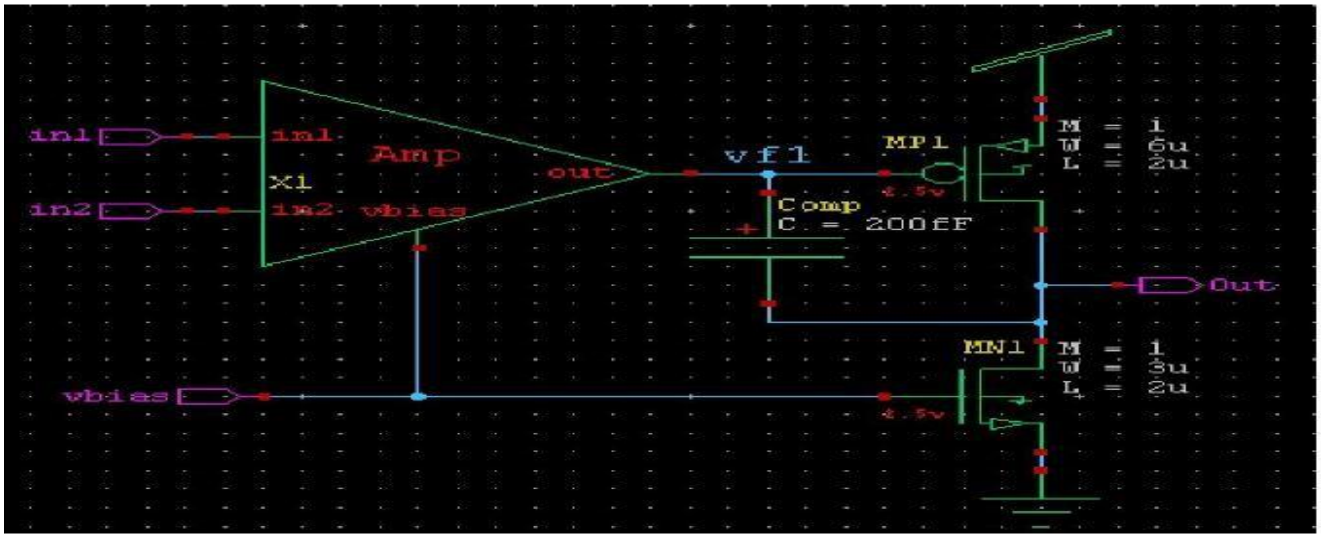
STEP 3: Perform AC Analysis of the differential amplifier.

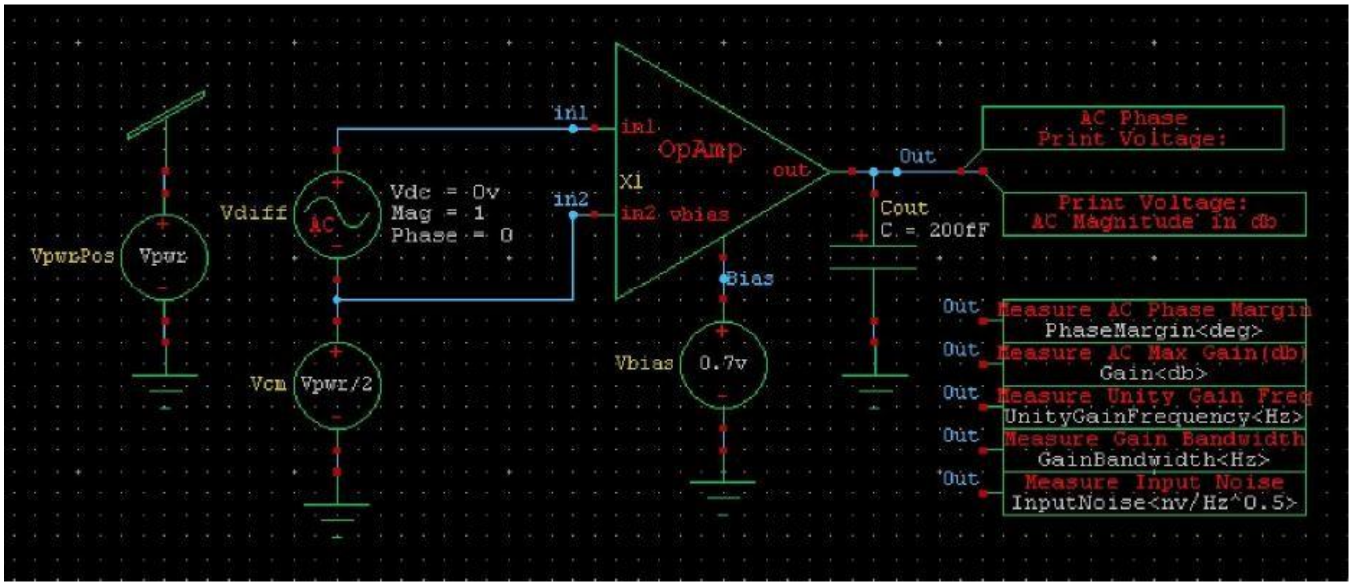
STEP 4: Obtain the frequency response from W-edit.

STEP 5: Obtain the spice code using T-edit.

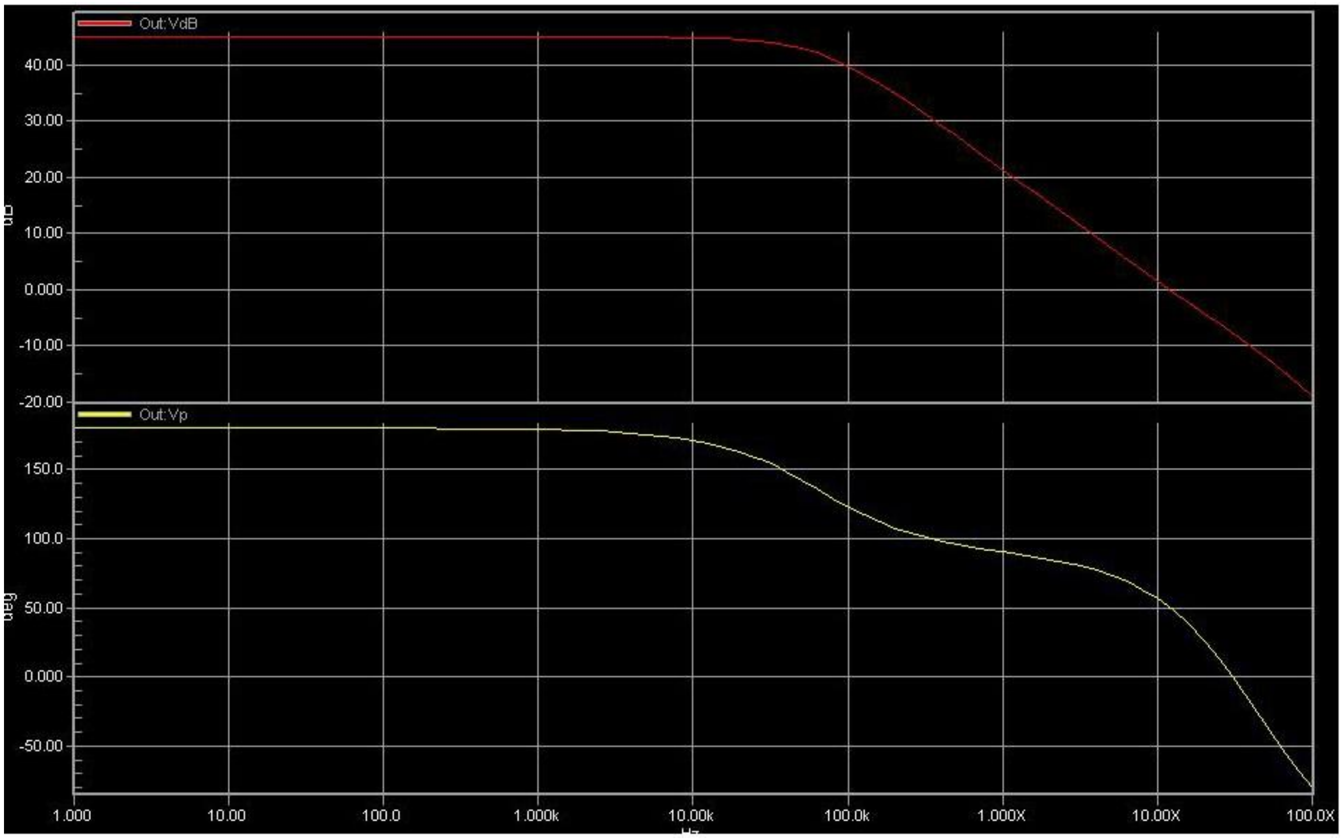
SCHEMATIC DIAGRAM:







VpwrPos
Vpwr
Vdiff
AC
Vdc = 0v
Mag = 1
Phase = 0
Vcm
Vpwr/2
in1
in2
OpAmp
X1
out
vbias
Bias
Vbias
0.7v
Out
Cout
C = 200fF
AC Phase
Print Voltage:
Print Voltage:
AC Magnitude in db
Measure AC Phase Margin
PhaseMargin<deg>
Measure AC Max Gain(db)
Gain<db>
Measure Unity Gain Freq
UnityGainFrequency<Hz>
Measure Gain Bandwidth
GainBandwidth<Hz>
Measure Input Noise
InputNoise<nv/Hz^0.5>



Out:VdB
Out:Vp
db
deg
Hz
40.00
30.00
20.00
10.00
0.000
-10.00
-20.00
150.0
100.0
50.00
0.000
-50.00
1.000
10.00
100.0
1.000k
10.00k
100.0k
1.000X
10.00X
100.0X

RESULT

Thus the functional verification of the **Differential Amplifier** through schematic entry and the output also verified successfully.

EXP: CMOS LOGIC GATES

Aim:

To Synthesize CMOS logic gates using Tanner

Description:

- **Logic NAND** $f(V_1, V_2) = (V_1 V_2)' = V_1' + V_2'$:

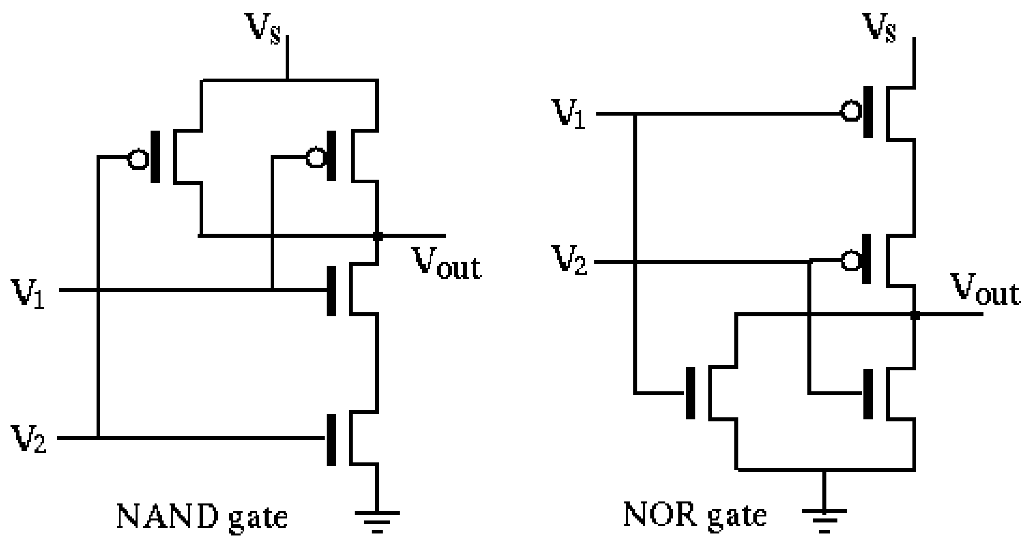
  The pull-down function is $f' = V_1\ V_2$, the pull-up function is $(V_1\ V_2)' = V_1' + V_2'$, The output function $V_{out} = f(V_1, V_2) = V_1' + V_2' = (V_1 V_2)'$ is the same as the pull-up function, a negation of AND, or NAND.

- **Logic NOR** $f(V_1 + V_2) = (V_1 + V_2)' = V_1' V_2'$:

  The pull-down function is $f' = V_1 + V_2$, the pull-up function is $(V_1 + V_2)' = V_1'\ V_2'$, The output is the same as the pull-up function $V_{out} = (V_1 + V_2)'$, negation of OR, or NOR.

		AND	OR	NAND	NOR
V_1	V_2	$V_1 V_2$	$V_1 + V_2$	$(V_1\ V_2)'$	$(V_1 + V_2)'$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

Diagram:



Result:

EXP: DESIGN OF ALU

Aim:

To Design an ALU and simulate using Xilinx.

PROGRAM:

```
module alu(
           input [7:0] A,B,  // ALU 8-bit Inputs
           input [3:0] ALU_Sel,// ALU Selection
           output [7:0] ALU_Out, // ALU 8-bit Output
           output CarryOut // Carry Out Flag
    );
    reg [7:0] ALU_Result;
    wire [8:0] tmp;
    assign ALU_Out = ALU_Result; // ALU out
    assign tmp = {1'b0,A} + {1'b0,B};
    assign CarryOut = tmp[8]; // Carryout flag
    always @(*)
    begin
        case(ALU_Sel)
        4'b0000: // Addition
           ALU_Result = A + B ;
        4'b0001: // Subtraction
           ALU_Result = A - B ;
        4'b0010: // Multiplication
           ALU_Result = A * B;
        4'b0011: // Division
           ALU_Result = A/B;
        4'b0100: // Logical shift left
           ALU_Result = A<<1;
         4'b0101: // Logical shift right
           ALU_Result = A>>1;
         4'b0110: // Rotate left
           ALU_Result = {A[6:0],A[7]};
         4'b0111: // Rotate right
```

```
           ALU_Result = {A[0],A[7:1]};
          4'b1000: //  Logical and
           ALU_Result = A & B;
          4'b1001: //  Logical or
           ALU_Result = A | B;
          4'b1010: //  Logical xor
           ALU_Result = A ^ B;
          4'b1011: //  Logical nor
           ALU_Result = ~(A | B);
          4'b1100: // Logical nand
           ALU_Result = ~(A & B);
          4'b1101: // Logical xnor
           ALU_Result = ~(A ^ B);
          4'b1110: // Greater comparison
           ALU_Result = (A>B)?8'd1:8'd0 ;
          4'b1111: // Equal comparison
            ALU_Result = (A==B)?8'd1:8'd0 ;
          default: ALU_Result = A + B ;
        endcase
    end

endmodule
```

RESULT:

EXP: DESIGN OF UNIVERSAL SHIFT REGISTER

Aim:

To Design a universal shift register and simulate using Xilinx.

DESCRIPTION:

Universal shift register is capable of converting input data to **parallel** or **serial** which also does shifting of data bidirectional, unidirectional (SISO , SIPO , PISO , PIPO) and also parallel load this is called as **Universal shift register** .

Shift register are used as: Data storage device , Delay element , communication lines , digital electronic devices (Temporary data storage , data transfer , data manipulation , counters) , etc .

Function table

Mode Control		Register operation
S1	S0	
0	0	No change
0	1	Shift left
1	0	Shift right
1	1	Parallel load

Block diagram of universal shift register(USR) :

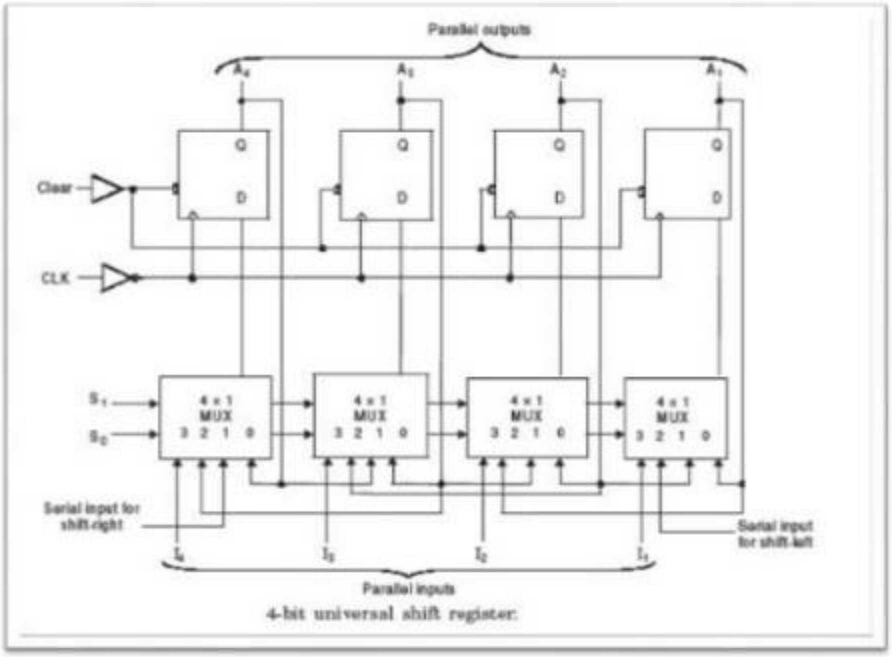

Parallel outputs
A4
A3
A2
A1
Q
D
Clear
CLK
S1
S0
4 x 1 MUX
3 2 1 0
Serial input for shift-right
Serial input for shift-left
I4
I3
I2
I1
Parallel inputs
4-bit universal shift register.

Figure block diagram of USR.

PROGRAM:

```
module Universal_shift_reg (data_out, msb_out, lsb_out, data_in,

msb_in, lasb_in, s1, s0, clk, rst);

output [3:0] data_out;          // Hold
output      msb_out, lsb_out;   // Serial shift from msb
input  [3:0] data_in;           // Serial shift from lsb
input       msb_in, lsb_in;     // Parallel load
input       s1, s0, clk, rst;
reg         data_out;

assign msb_out= data_out[3];
assign lsb_out= data-out[0];

always @ (posedge clk)
```

```
begin
 if (rst) data_out<=0;
 else case ({s1, s0})
 0 : data_out <= data_out;
 1 : data_out <= {msb_in, data_out[3:1]};
 2 : data_out <= {data_out[2:0], lsb_in};
 3 : data_out <= data_in;
 endcase
end
endmodule
```

RESULT:

EXP: DESIGN OF MEMORY

Aim:

To Design a memory and simulate using Xilinx.

DESCRIPTION:

Verilog Code for 16x4 Memory

Sr. No.	Name of the Pin	Direction	Width	Description
1	Address	Input	4	Input address
2	Ip	input	4	Input data to memory
3	Rd_wr	Input	1	Control signal 1=read from memory 0=write in to memory
4	Clk	Input	1	Clock input
5	op	Output	4	Output read from memory

PROGRAM:

```
module memory_16x4(op,ip,rd_wr,clk,address);
   output reg [3:0] op;
   input [3:0]  ip;
   input [3:0] address;
   input  rd_wr,clk;
   reg [3:0]  memory[0:15];



   always @(posedge clk)
     begin
       if (rd_wr)
         op=memory[address];
       else
         begin
           memory[address]=ip;
       end
     end
endmodule // memory_16x4
```

RESULT:

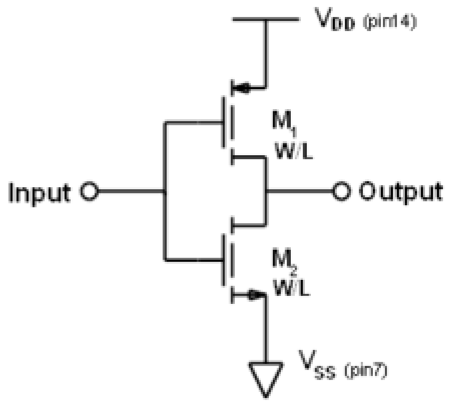
EXP: **DESIGN OF CMOS INVERTING AMPLIFIER**

Aim:

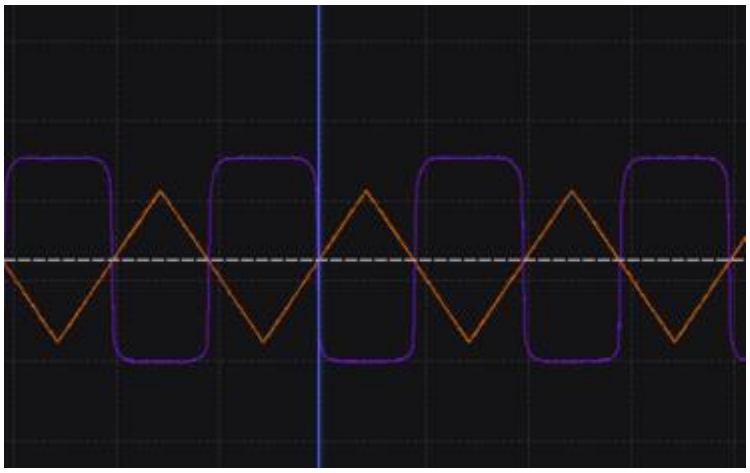
To Design and simulate a CMOS inverting amplifier.

DESCRIPTION:

A CMOS inverter can also be viewed as a high gain amplifier. It consists of one PMOS device, M_1 and one NMOS device M_2. Generally the CMOS fabrication process is designed such that the threshold voltage, V_{TH}, of the NMOS and PMOS devices are roughly equal i.e. complementary. The designer of the inverter then adjusts the width to length ratio, W/L, of the NMOS and PMOS devices such that their respective transconductance is also equal.



CMOS Inverting amplifier



RESULT:

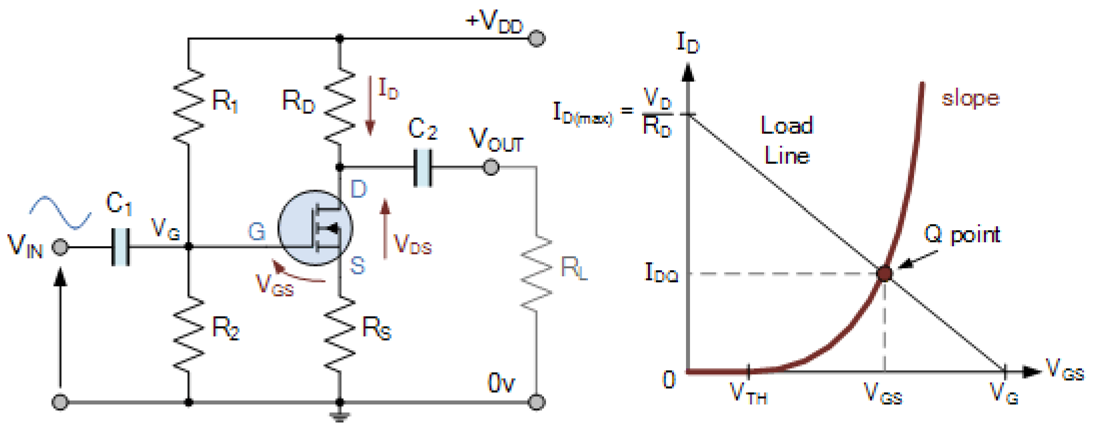
EXP: DESIGN OF COMMON SOURCE, COMMON GATE AND COMMON DRAIN AMPLIFIER

Aim:

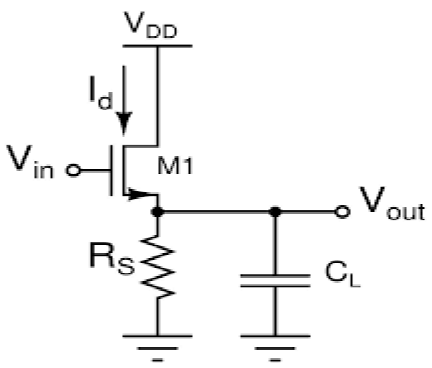
To Design and simulate a CS,CG and CD amplifier.

DESCRIPTION:

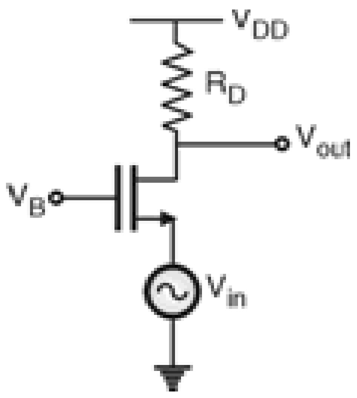
COMMON SOURCE AMPLIFIER:

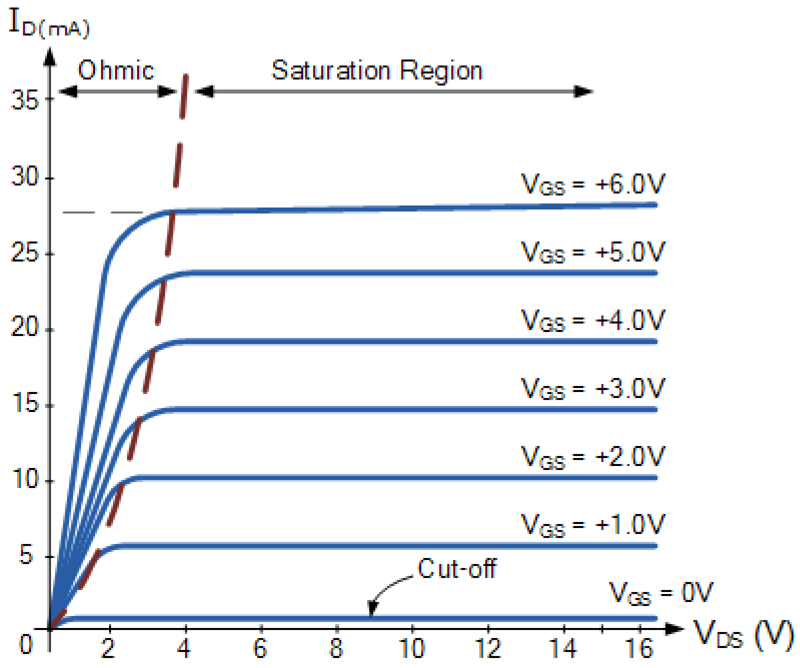


COMMON DRAIN AMPLIFIER:



COMMON GATE AMPLIFIER





RESULT: