Half Adder.v

```
module halfadder(sout,cout,a,b);
output sout,cout;
input a,b;
assign sout=a^b;
assign cout=(a&b);
endmodule
```

Half Adder_TB.v

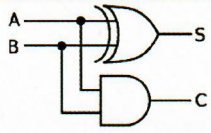
```
module halfadder_tb; //test batch
reg ta, tb;
wire tcout, tsum;
halfadder u1 (.a(ta), .b(tb), .cout(tcout), .sum(tsum)); //name
association
initial
begin
        ta = 0; tb = 0;
        #10; ta = 0; tb = 1;
        #10;
        ta = 1; tb = 0;
        #10;
        ta = 1; tb = 1;
        #10 $stop;
end
endmodule
```

Inputs		Outputs	
A	B	C_{out}	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Logic Equations:

$C_{out} = A \& B$

$Sum = A \oplus B$



Full Adder.v

```
module fulladder(sout,cout,a,b,cin);
output sout,cout;
input a,b,cin;
assign sout=(a^b^cin);
assign cout=((a&b)|(a&cin)|(b&cin));
endmodule
```

Full Adder_TB.v

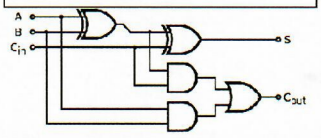
```
module fulladder_tb;
reg a, b, cin;
wire cout, sum;
fulladder u3(.a(a), .b(b), .cin(cin), .cout(cout), .sum(sum));
initial
begin
        a = 0; b = 0; cin = 0;
        #10 a = 0; b = 0; cin = 1;
        #10; a = 0; b = 1; cin = 0;
        #10; a = 0; b = 1; cin = 1;
        #10; a = 1; b = 0; cin = 0;
        #10; a = 1; b = 0; cin = 1;
        #10; a = 1; b = 1; cin = 0;
        #10; a = 1; b = 1; cin = 1;
        #10 $stop;
end
endmodule
```

Inputs			Outputs	
A	B	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Logic Equations:

$Sum = A \oplus B \oplus C_{in}$

$C_{out} = (A \oplus B)\, C_{in} + A \& B$



Multiply4bits.v

```
module multiply4bits(pro,a,b);
output [7:0]pro;
input [3:0]a;
input [3:0]b;
assign pro[0]=(a[0] & b[0]);
wire x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17;
halfadder halfadder1(pro[1],x1,(a[1] & b[0]),(a[0] & b[1]));
fulladder fulladder1(x2,x3,a[1]&b[1],(a[0]&b[2]),x1);
fulladder fulladder2(x4,x5,(a[1]&b[2]),(a[0]&b[3]),x3);
halfadder halfadder2(x6,x7,(a[1]&b[3]),x5);
halfadder halfadder3(pro[2],x15,x2,(a[2]&b[0]));
fulladder fulladder5(x14,x16,x4,(a[2]&b[1]),x15);
fulladder fulladder4(x13,x17,x6,(a[2]&b[2]),x16);
fulladder fulladder3(x9,x8,x7,(a[2]&b[3]),x17);
halfadder halfadder4(pro[3],x12,x14,(a[3]&b[0]));
fulladder fulladder8(pro[4],x11,x13,(a[3]&b[1]),x12);
fulladder fulladder7(pro[5],x10,x9,(a[3]&b[2]),x11);
fulladder fulladder6(pro[6],pro[7],x8,(a[3]&b[3]),x10);
endmodule
```

Multiply4bits_TB.v

```
module multiply4bits_tb();
reg [3:0]a;
reg [3:0]b;
wire [7:0]pro;
multiply4bits u1(.a(a), .b(b), .pro(pro));
initial
begin
        a = 4'b0000; b = 4'b0000;
        #10;
        a = 4'b0001; b = 4'b0001;
        #10;
        a = 4'b0010; b = 4'b0010;
        #10;
        a = 4'b0011; b = 4'b0011;
        #10;
        a = 4'b0100; b = 4'b0100;
        #10;
        a = 4'b0101; b = 4'b0101;
        #10;
        a = 4'b0110; b = 4'b0110;
        #10;
        a = 4'b0111; b = 4'b0111;
        #10;
        a = 4'b1000; b = 4'b1000;
        #10;
        a = 4'b1001; b = 4'b1001;
        #10;
        a = 4'b1010; b = 4'b1010;
        #10;
        a = 4'b1011; b = 4'b1011;
        #10;
        a = 4'b1100; b = 4'b1100;
        #10;
        a = 4'b1101; b = 4'b1101;
        #10;
        a = 4'b1110; b = 4'b1110;
        #10;
        a = 4'b1111; b = 4'b1111;
        #10; $stop;
end
endmodule
```

Inputs		Outputs
A[3:0]	B[3:0]	Pro[7:0]
0000	0000	00000000
0001	0001	00000001
0010	0010	00000100
0011	0011	00001001
0100	0100	00010000
0101	0101	00011001
0110	0110	00110001
0111	0111	00110001
1000	1000	01000000
1001	1001	01010001
1010	1010	01100100
1011	1011	01111001
1100	1100	10010000
1101	1101	10101001
1110	1110	11000100
1111	1111	11100001

Use Verilog to design a finite state machine to recognize the sequence 0110.

Analysis:

S_idle: does not detect anything

S1: detect 0

S2: detect 01

S3: detect 011

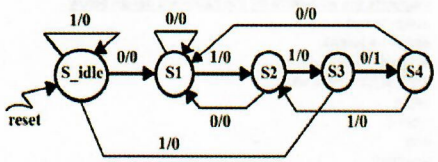
S4: detect 0110

State Table:

Current State	A	Next state	Y
S_idle	0	S1	0
	1	S_idle	0
S1	0	S1	0
	1	S2	0
S2	0	S1	0
	1	S3	0
S3	0	S4	1
	1	S_idle	0
S4	0	S1	0
	1	S2	0

Final Solution:

State Diagram:




```
module fsm_detector(reset, clk, a, y);
input reset, a, clk;
output y;
reg y;
parameter s_idle = 3'b000, s1=3'b001, s2=3'b010, s3=3'b011, s4=3'b100;
reg [2:0] cs, ns;
always@ (posedge clk or posedge reset)
begin
  if(reset) cs <= s_idle;
  else cs <= ns;
end
always@(cs or a)
begin
  case(cs)
    s_idle: if(a) ns = s_idle;
    else ns = s1;
    s1: if(a) ns = s2;
  else ns = s1;
    s2: if(a) ns = s3;
  else ns = s1;
    s3: if(~a) ns = s4;
  else ns = s_idle;
    s4: if(a) ns = s2;
  else ns = s1;
  default: ns = s_idle;
  endcase
end
always@(cs or a)
begin
  case(cs)
  s_idle: y = 0;
  s1: y=0;
  s2: y=0;
  s3:  if(~a) y=1;
       else y=0;
  s4:  y=0;
  default: y = 0;
  endcase
end
endmodule
```

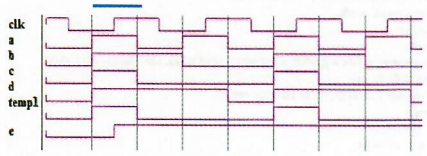
D-Flip-Flop

```
module dffa ( reset, clk, load, da, qa);
input   reset, clk, load;
input  [3:0]  da;
output [3:0]  qa;
reg    [3:0]  qa;
always@(posedge clk or posedge reset)
begin
    if (reset)
        qa <= 4'b0000;
    else if (load)
        qa <= da;
end
endmodule
`timescale 1ns / 1ps
module dffa_tb;
                // Inputs
                reg reset;
                reg clk;
                reg load;
                reg [3:0] da;
                // Outputs
                wire [3:0] qa;
                // Instantiate the Unit Under Test (UUT)
                dffa uut (
                                .reset(reset),
                                .clk(clk),
                                .load(load),
                                .da(da),
                                .qa(qa)
                );
                //continuous 20 sec clk
                always
                begin
                                clk = 1'b1;
                                #10;
                                clk = 1'b0;
                                #10;
                end
                initial begin
                                // Initialize Inputs
                                load = 1'b1;
                                da = 1'b1;
                                reset = 1'b0;
                                #20;
                                load = 1'b0;
                                da = 1'b0;
                                reset = 1'b1;
                                #20;
                                load = 1'b0;
                                da = 1'b0;
                                reset = 1'b0;
                                #20;
                                load = 1'b0;
                                da = 1'b1;
                                reset = 1'b1;
                                #20;
                                load = 1'b1;
                                da = 1'b1;
                                reset = 1'b0;
                                #20
                                // Wait 100 ns for global reset to finish
                                 $stop;
                                // Add stimulus here
                end
endmodule
```

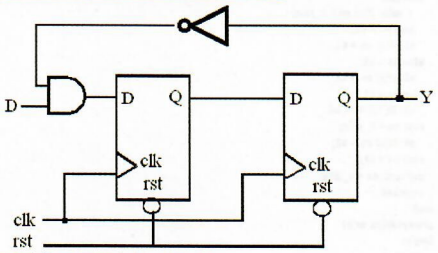
Complete the waveform for the hardware design below

```
assign temp1 = a & b;
assign c = temp1;
assign d = a|b;
always@(posedge clk)
begin
  c<=d;
end
```

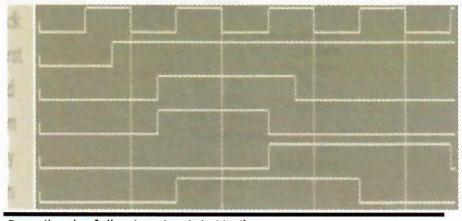
Solution:



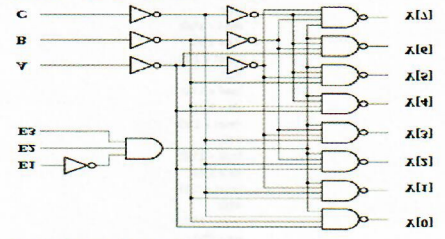
Write Verilog Program & TB for this circuit.



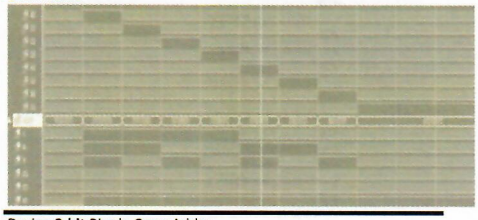

```
module program (clk, d, rst, y0;
input clk, d, rst;
output y;
reg y, n;
wire m;
assign m = d & (~y);
always@ (posedge clk or negedge rst)
begin
  if(~rst)
  begin
    n <= 0; y <= 0;
  end
  else
  begin
    n <= m; y <= n;
  end
end
endmodule
//////////////////////////////////////////////////////////////////
module program_tb;
reg clk, d, rst;
wire y;
program uut(clk, d, rst, y);
initial clk = 0;
always #10 clk = ~clk;
initial
begin
  rst = 0; d = 0;
  #16 rst = 1;
  #10 d = 1;
  #30 d = 0;
  #200 $stop;
end
endmodule
```



Describe the following circuit in Verilog.

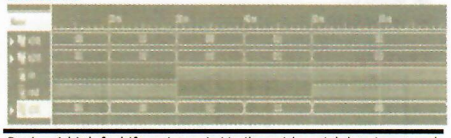



```
module decoder (e1, e2, e3, a, b, c, y);
input a, b, c, e1, e2, e3;
output [7:0] y;
wire m;
assign m = ~e1 & e2 & e3;
assign y[0] = ~((~c & ~b & ~a) & m);
assign y[1] = ~((~c & ~b & a) & m);
assign y[2] = ~((~c & b & ~a) & m);
assign y[3] = ~((~c & b & a) & m);
assign y[4] = ~((c & ~b & ~a) & m);
assign y[5] = ~((c & ~b & a) & m);
assign y[6] = ~((c & b & ~a) & m);
assign y[7] = ~((c & b & a) & m);
endmodule
//////////////////////////////////////////////////////////////////
//////////////
module decoder_tb;
reg e1, e2, e3, a, b, c;
wire [7:0] y;
decoder uut(e1, e2, e3, a, b, c, y);
initial
begin
  {e1, e2, e3} = 3'b111;
  {c, b, a} = 3,b111;
  #10 {e1, e2, e3} = 3'b011;
  {c, b, a} = 3'b000;
  #10 {c, b, a} = 3'b001;
  #10 {c, b, a} = 3'b010;
  #10 {c, b, a} = 3'b011;
  #10 {c, b, a} = 3'b100;
  #10 {c, b, a} = 3'b101;
  #10 {c, b, a} = 3'b110;
  #10 {c, b, a} = 3'b111;
  #50 $stop;
end
endmodule
```



Design **3-bit** Ripple Carry Adder

```
module fulladder(a, b, cin, cout, s);
input a, b, cin;
output cout, s;
assign s = a ^ b ^ cin;
assign cout = ((a^b)&cin) | (a&b);
endmodule
//////////////////////////////////////////////////////////////////
module ripplecarryadder3(a, b, cin, cout, s);
input [2:0] a, b;
input cin;
output [2:0] s;
output cout;
wire [1:0] m;
fulladder g1(.a(a[0]), .b(b[0]), .cin(cin), .cout(m[0]), .s(s[0]));
fulladder g2(.a(a[1]), .b(b[1]), .cin(m[0]), .cout(m[1]), .s(s[1]));
fulladder g3(a[2], b[2], m[1], cout, s[2]);
endmodule
//////////////////////////////////////////////////////////////////
module ripplecarryadder3_tb;
reg [2:0] a, b;
reg cin;
wire[2:0] s;
wire cout;
ripplecarryadder3 uut(a, b, cin, cout, s);
initial
begin
  a = 0; b = 0; c = 0;
  #10 a = 4; b = 5;
  #10 a = 6; b = 6;
  #10 a = 0; b = 2; cin = 1;
  #10 a = 3; b = 3;
  #10 a = 5; b = 2;
  #20 $stop;
end
endmodule
```

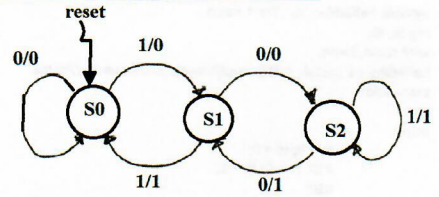


Design 4-bit left shift registers in Verilog with serial data input and serial data output.

```
module shift_L4(clk, din, dout);
input clk, din; output dout;
reg [3:0] q; assign dout = q[3];
always@(posedge clk)
begin
  //q <= { q[2:0], din };
  q[3] <= q[2];
  q[2]<= q[1];
  q[1] <= q[0];
  q[0] <= din;
end
endmodule
```

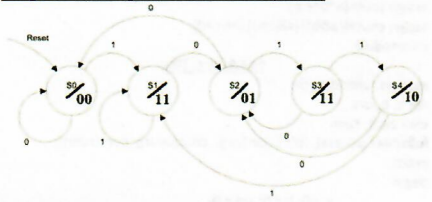
```
//Right shift//
module shift_R4(clk, din, dout);
input clk, din;
output dout;
reg [3:0] q;
assign dout = q[0];
always@(posedge clk)
begin
  q <= { din, q[3:1] };
end
endmodule
```

Design the following finite state machine (mealy)




```
module fsm(clk, reset, a, y);
input clk, reset, a;
output y;  reg y;
parameter s0 = 2'b00, s1=2'b01, s2=2'b10;
reg [1:0] cs, ns;
always@(posedge clk or posedge reset)
begin
  if(reset)  cs <= s0;
  else  cs <= ns;
end
always@(cs or a)
begin
  case(cs)  s0: if(a) ns = s1;
  else ns = s0; s1: if(~a) ns = s2;
  else ns = s0; s2: if(a) ns = s2;
  else ns = s1; default: ns = s0;
  endcase
end
always@(cs or a)
begin
  case(cs)  s0: y = 0;
  s1: if(a) y = 1; else y = 0;
  s2: y = 1;
  default: y = 0;
  endcase
end
endmodule
```

Design the following finite state machine (moore)

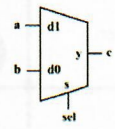



```
module fsm ( clk, reset, a, y );
input clk, reset, a;
output [1:0] y;
reg [1:0] y;
parameter s0 = 3'b000, s1=3'b001, s2=3'b010, s3=3'b011,
s4=3'b100; reg [2:0] cs, ns;
always@(posedge clk or negedge reset)
begin
  if(~reset) cs <= s0;
  else  cs <= ns;
end
always@(cs or a)
begin
  case(cs)  s0: if(a) ns = s1;
  else ns = s0; s1: if(~a) ns = s2;
  else ns = s0; s2: if(a) ns = s3;
  else ns = s0; s3: if(a) ns = s4;
  else ns = s2; s4: if(a) ns = s1;
  else default:
  endcase
end
always@(cs)
begin
  ns = s2; ns = s0;
  case(cs)
  s0: y = 2'b00;
  s1:  y = 2'b11;
  s2:  y = 2'b01;
  s3: y = 2'b11;
  s4: y = 2'b10;
  default: y = 2'b00;
  endcase
end
endmodule
```

```
process(a, b,
sel)
begin
 if (sel = '1')
then
  c <= a;
 else
  c <= b;
end if;
end process;
```

Draw the synthesized schematic for the VHDL code.

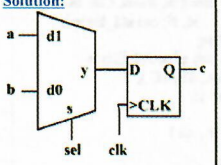
Solution:




```
process (clock )
begin
 if (rising_edge(clock))
then
   if(sel = '1') then
      c <= a;
   else
      c <= b;
   end if;
end if;
end process;
```

Draw the synthesized schematic for the VHDL code.

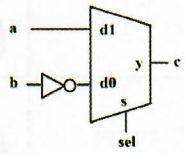
Solution:




```
process (a , b ,
sel)
begin
case (sel) is
when '0'=> c
<= ~b;
when '1' => c
<= a;
end case;
end process;
```

Draw the synthesized schematic for the VHDL code.

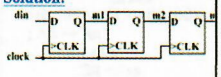
Solution:




```
process (clock )
begin
 if (rising_edge(clock))
then
 m1 <= din;
 m2 <= m1;
 m3 <= m2;
end if;
end process;
```

Draw the synthesized schematic for the VHDL code.

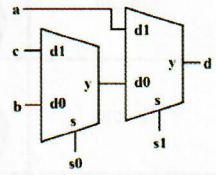
Solution:




```
process ((sl ,
s0 , a , b , c)
begin
if (sl ='1') then
  d <= a;
elsif (s0 = '0')
then
   d <= b;
else
  d <= c;
end if;
end process;
```

Draw the synthesized schematic for the VHDL code.

Solution:



Counter

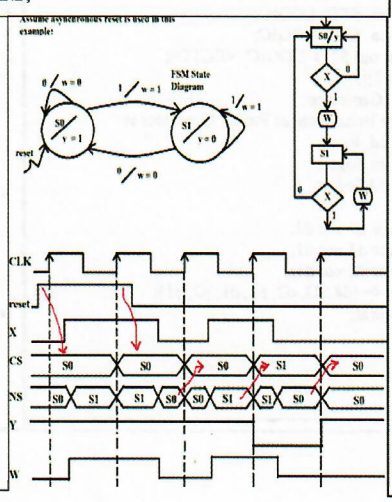
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity new_cnt is
 port(osc: in std_logic; -- 50MHz
  load, updown: in std_logic;
  sw: in std_logic_vector(3 downto 0);
  dout: out std_logic_vector(3 downto 0);
  clkref: out std_logic
 );
end new_cnt;
architecture cnt_beh of new_cnt is
signal  clk: std_logic;
signal  cnt10: std_logic_vector(3 downto 0);
signal  cnt : std_logic_vector(3
              downto 0);
begin
   clkref <= clk;
   process(osc)
   begin
    if( rising_edge(osc) ) then
      if (cnt10 = 9 ) then
         cnt10 <= ( others => '0' );
         clk  <= '1';
      elsif (cnt10 < 4) then
         cnt10 <= cnt10 + 1;
         clk  <= '1';
      else
         cnt10 <= cnt10 + 1;
         clk  <= '0';
      end if;
    end if;
   end process;
   process(clk, load, updown)
   begin
if( rising_edge(clk) ) then
      if(load = '1' ) then
         cnt <= sw;
      else
        if(updown = '1') then
          if( cnt = 9 ) then
            cnt <= "0000";
          else
            cnt <= cnt + 1;
          end if;
        else
          if( cnt = 0 ) then
            cnt <= "1001";
          else
            cnt <= cnt - 1;
          end if;
        end if;
      end if;
    end if;
   end process;
   dout <= cnt;
end cnt_beh;
```

Finite State Machine

```
library IEEE;
use IEEE.std_logic_1164.all;
entity fsm is port (
clk, reset, x1 : IN std_logic;
 outp : OUT std_logic
);
end entity;
 architecture beh1 of fsm is
type state_type is (s1,s2,s3,
                    s4);
signal state, next_state:
state_type;
begin
process1:
 process (clk, reset)
 begin
 if (reset ='1') then
state <= s1;
elsif (clk = '1' and clk'Event)
then
state <= next_state;
 end if;
end process process1;
process2 : process (state, x1)
begin
 case state is when
s1 => if x1='1' then next_state
<= s2;
 else next_state <= s3;
end if;
when s2 => next_state <= s4;
when s3 => next_state <= s4;
when s4 => next_state <= s1;
end case;
end process process2;
 process3 : process (state)
begin
case state is when
s1 => outp <= '1';
when s2 => outp <= '1';
when s3 => outp <= '0';
 when s4 => outp <= '0';
end case;
end process process3;
end beh1;
```

Test bench for Left shift Register

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY leftshiftregister_tb IS
END leftshfitregister_tb;
ARCHITECTURE behavior OF leftshiftregister_tb IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT leftshiftregister
 PORT(
       clk : IN  std_logic;
       rst : IN  std_logic;
     random_sequence : OUT  std_logic_vector(3 downto
0)
      );
  END COMPONENT;
  --Inputs
  signal clk : std_logic := '0';
  signal rst : std_logic := '0';
--Outputs
  signal random_sequence : std_logic_vector(3 downto 0);
  -- Clock period definitions
  constant clk_period : time := 65 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
 uut: leftshiftregister PORT MAP (
        clk => clk,
        rst => rst,
        random_sequence => random_sequence
      );
 -- Clock process definitions
 clk_process :process
  begin
 clk <= '0';
 wait for clk_period/2;
 clk <= '1';
 wait for clk_period/2;
  end process;
  -- Stimulus process
  stim_proc: process
  begin
    -- hold reset state for 100 ns.
rst <= '1';
    wait for clk_period/2;
rst <= '0';
    wait for clk_period/2;
    -- wait for clk_period*10;
    -- insert stimulus here
    wait;
  end process;

END;
```



Left Shift Register

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
entity leftshiftregister is
 Port ( clk : in  STD_LOGIC;
 rst : in  STD_LOGIC;
random_sequence: out
STD_LOGIC_VECTOR
 (3 downto 0));
end leftshiftregister;
architecture Behavioral of leftshiftregister
is
signal df1: std_logic;
signal df2: std_logic;
signal df3: std_logic;
signal df4: std_logic;
begin
process(clk, rst)
begin
if (rst = '1') then
df1 <= '1';
df2 <= '0';
df3 <= '1';
df4 <= '1';
elsif rising_edge (clk) then
df4 <= df3;
df3 <= df2;
df2 <= df1 xor df4;
df1 <= df4;
end if;
end process;
random_sequence <= (df4, df3, df2, df1);
end Behavioral;
```

Hamming code Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-- Uncomment the following library
declaration if using
-- arithmetic functions with Signed or
Unsigned values
--USE ieee.numeric_std.ALL;
ENTITY parity_generator_tb IS
END parity_generator_tb;
ARCHITECTURE behavior OF
parity_generator_tb IS
   -- Component Declaration for the Unit
Under Test (UUT)
   COMPONENT Parity_Generator
   PORT(
       d4 : IN  std_logic;
       d3 : IN  std_logic;
       d2 : IN  std_logic;
       d1 : IN  std_logic;
       Ham_code : OUT  std_logic_vector(6
downto 0)
      );
   END COMPONENT;
   --Inputs
                signal clk : std_logic := '0';
                signal d4 : std_logic := '1';
   signal d3 : std_logic := '0';
   signal d2 : std_logic := '1';
   signal d1 : std_logic := '1';
  --Outputs
   signal Ham_code : std_logic_vector(6
downto 0);
   -- No clocks detected in port list. Replace
<clock> below with
   -- appropriate port name
   constant clock_period : time := 10 ns;
BEGIN
 -- Instantiate the Unit Under Test (UUT)
   uut: Parity_Generator PORT MAP (
        d4 => d4,
        d3 => d3,
        d2 => d2,
        d1 => d1,
        Ham_code => Ham_code
      );
  -- Clock process definitions
   clock_process :process
   begin
clk <= '0';
wait for clock_period/2;
clk <= '1';
wait for clock_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
-- hold reset state for 100 ns.
 wait for 100 ns;

 wait for clock_period*10;
    -- insert stimulus here
    wait;
  end process;
END;
```

DECODER

```
library ieee; use
ieee.std_logic_1164.all; entity
dec is port (
sel: in std_logic_vector (2
downto 0); res: out
std_logic_vector (7 downto 0)
); end dec; architecture archi of
dec is begin with sel select res
<= "00000001" when "000",
"00000010" when "001",
"00000100" when "010",
"00001000" when "011",
"00010000" when "100",
"00100000" when "101",
"01000000" when "110",
"10000000" when others;
end archi;
```

8-bit Shift-Left Register with Positive-Edge Clock, Asynchronous Parallel Load, Serial In, and Parallel Out

```
library ieee; use
ieee.std_logic_1164.all; entity
shift is port(
C, SI, ALOAD : in std_logic;
D : in std_logic_vector(7
downto 0);
PO : out std_logic_vector(7
downto 0));
end shift;
architecture archi of shift is
signal tmp: std_logic_vector(7
downto 0);
begin process (C, ALOAD)
begin if (ALOAD='1') then tmp
<= D;
elsif (C'event and C='1') then
tmp <= tmp(6 downto 0) & SI;
end if;
end process;
PO <= tmp;
end archi;
```

8-bit Shift-Left Register with Negative-Edge Clock, Clock Enable, Serial In, and Serial Out

```
library ieee; use
ieee.std_logic_1164.all; entity
shift is port(
C, SI, CE : in std_logic;
SO : out std_logic
); end shift;
architecture archi of shift is
signal tmp: std_logic_vector(7
downto 0); begin process (C,
CE) begin
if (C'event and C='0') then if
(CE='1') then for i in 0 to 6 loop
tmp(i+1) <= tmp(i); end loop;
tmp(0) <= SI; end if; end if;
 end process;
```

Flip-flop with Positive-Edge Clock

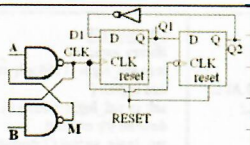
```
library ieee; use ieee.std_logic_1164.all;
entity flop is port(
CLK, D : in std_logic;
Q : out std_logic ); end flop;
architecture archi of flop is
 begin process (CLK)
begin
 if(CLK'event and CLK='1') then
 --if (rising_edge(CLK)) then
Q <= D; end if; end process; end archi;
```

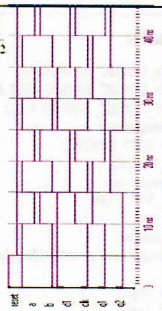
Flip-flop with Negative-Edge Clock and Asynchronous Clear

```
library ieee; use ieee.std_logic_1164.all;
entity flop is port(
C, D, CLR: in std_logic;
Q : out std_logic );
 end flop;
 architecture archi of flop is
 begin process (C, CLR)
 begin
 if (CLR = '1')then
 Q <= '0';
elsif (C'event and C='0')then
--if (falling_edge(CLK)) then
Q <= D; end if; end process; end archi;
```

Flip-flop with Positive-Edge Clock and Synchronous Set

```
library ieee; use ieee.std_logic_1164.all; entity
flop is port(
C, D, S : in std_logic; Q : out std_logic); end
flop;
architecture archi of flop is begin process (C,
S) begin
if (C'event and C='1') then if (S='1') then Q <=
'1'; else Q <= D; end if; end if; end process;
end archi;
```





Complete the following wavef, m

Testbench Solution:

```
Library ieee;
Use ieee.std_logic_1164.all;
entity prob1_tb
end prob1_tb;
architecture beh of prob1_tb is
component prob1 (a, b, reset: in std_logic;
                 q1, q2: out std_logic );
end component;
signal a, b, reset:  std_logic;
signal  q1, q2:   std_logic;
--signal  a:   std_logic  := '1';
begin
uut: prob1  port map(a, b, reset, q1, q2);
process
begin
   reset <= '1';
   wait for 5 ns;
   reset <= '0';
   wait for 100 ns;
   wait;
end process;
b <= not a;
 process
 begin
   a <= '1';
   wait for 10 ns;
   a <= '0';
   wait for 5 ns;
   a <= '1';
   wait for 5 ns;
   a <= '0';
   wait for 5 ns;
   a <= '1';
   wait for 5 ns;
   a <= '0';
   wait for 5 ns;
   a <= '1';
   wait for 5 ns;
   a <= '0';
   wait for 5 ns;
   wait;
end process;
end beh;
```

Complete VHDL Design:
You are not allowed to use hierarchical design method in this question.

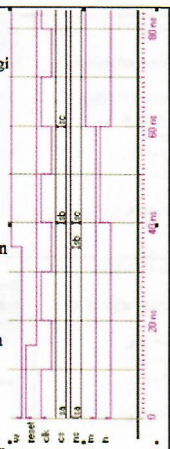
Solution:

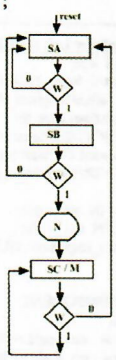
```
Library ieee;
Use ieee.std_logic_1164.all;
entity prob1  (
a, b, reset: in std_logic;
q1, q2:  out  std_logic );
end prob1;
architecture circuit of prob1 is
signal     clk,  m, d1, t1, t2:
std_logic;
begin
   clk <= a  nand m;
   m  <=  b  nand clk;
   d1 <= not t2;
process ( clk , reset)
begin
   if (reset= '1') then
      t1 <= '0';
   elsif (rising_edge(clk)) then
      t1 <= d1;
   end if;
end process;
process (  clk , reset)
begin
   if (reset= '1') then
      t2 <= '0';
   elsif (rising_edge(clk)) then
      t2 <= t1;
   end if;
end process;
q1 <= t1;
q2 <= t2;
end circuit;
```

Clock Division

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
entity clkdiv is
 Port ( clk :   in  STD_LOGIC;
 rst : in  STD_LOGIC;
 clk2 :  out  STD_LOGIC);
end clkdiv;
architecture Behavioral of clkdiv is
signal cnt_div:  std_logic_vector (26 downto 0);
--signal cnt_div:  std_logic_vector (3 downto 0);
begin
process (clk, rst)
begin
if (rst = '1') then
cnt_div <= ( others=>'0' );
clk2 <= '0';
   elsif rising_edge (clk) then
if (cnt_div = 49999999) then
      cnt_div  <= ( others=>'0' );
      clk2    <= '1';
 elsif (cnt_div < 24999999) then
cnt_div  <= cnt_div + 1;
      clk2    <= '1';
     else
      cnt_div  <= cnt_div + 1;
      clk2    <= '0';
     end if;
   end if;
end process;
end Behavioral;
```

```
Library ieee;
Use ieee.std_logic_1164.all;
entity prob2_fsm ( w, Reset, Clk: in std_logi
        M, N: out std_logic );
End prob2_fsm;
Architecture beh of prob2_fsm is
type  st is ( SA, SB, SC );
signal  cs, ns:  st;
begin
   process ( w ,  cs )
   begin
   case (cs)
            when SA=>  if (w = '0') then
                          ns <= SA ;
                          else
                     ns  <= SB ;
                    end if;
            when SB=>  if(w = '0') then
                          ns  <= SA ;
                       else
                     ns <= SC ;
                    end if;

            when SC=>  if (w = '0') then
                          ns <= SA ;
                       else
                     ns <= SC ;
                    end if;
            when others=> ns <= SA;
   end case;
   end process;
process (  Clk ,   Reset)
begin
   if( Reset = '1') then
            cs <= SA ;
   elsif (rising_edge(Clk)) then
            cs <= ns ;
   end if;
 end process;
  M <= '1' when (cs=SC)  else  '0';
  N <= w when (cs=SB)  else '0';
end beh;
```





CRC Error Detection

Message 110101

Generating Polynomial = 101

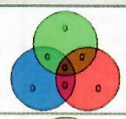
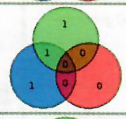
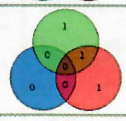
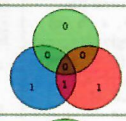
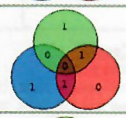
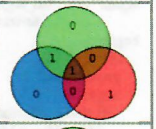
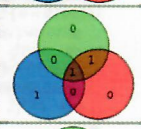
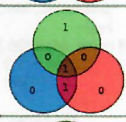
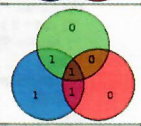
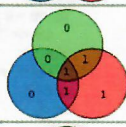
```
          111011
101 ) 110101 00
      101
       111
       101
         100
         101
           011
           000
             110
             101
               110
               101
                 11   <- Remainder = CRC checksum
```

111011 — Quotient (has no function in CRC calculation)

Message with CRC = 11010111

Hamming Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
entity Parity_Generator is
   Port ( d4 : in  STD_LOGIC;
          d3 : in  STD_LOGIC;
          d2 : in  STD_LOGIC;
          d1 : in  STD_LOGIC;
Ham_code: out STD_LOGIC_VECTOR
 (6 downto 0));
end Parity_Generator;
architecture Behavioral of Parity_Generator is
signal p1: std_logic;
signal p2: std_logic;
signal p3: std_logic;
begin
p3 <= d4 xor d3 xor d2;
p2 <= d4 xor d3 xor d1;
p1 <= d4 xor d2 xor d1;
Ham_code <= (d4, d3, d2, p3, d1, p2, p1);
end Behavioral;
```

Data D1,D2,D3,D4	Hamming(7,4) Transmitted P1,P2,D1,P3,D2,D3,D4	Diagram
0000	0000000	
1000	1110000	
0100	1001100	
1100	0111100	
0010	0101010	
1010	1011010	
0110	1100110	
1110	0010110	
0001	1101001	
1001	0011001	
0101	0100101	
1101	1010101	
0011	1000011	
1011	0110011	
0111	0001111	
1111	1111111	

1. [28 points] Fill out the blanks shown below.

[1]. Given the following VHDL statement,
rst <= '0', '1' after 100 ns, '0' after 200ns;

rst is ___ at time zero, ___ at time 50 ns, ___ at time 100 ns;
___ at time 150 ns, ___ at time 200 ns, ___ at time 300 ns;

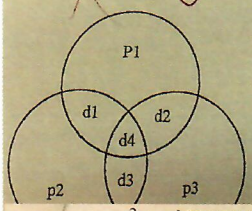
[2]. Each of the D2, D1, D0 signals is std_logic type,
signal D2, D1, D0: std_logic;
signal D3: std_logic_vector(2 downto 1) ;

Given the following codes:

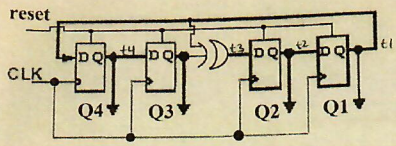
D2 <= '0'; D1 <= '1'; D0 <= '1';
D3 <= (D2 xor D1) & (D1 nor D0);

What is binary value for D3? D3 ___

[3]. (7,4) hamming code construction diagram is shown below.
If d4 d3 d2 d1 = "1100", the constructed 7-bit hamming code is:
Bit 7: ___ Bit 6: ___ Bit 5: ___ Bit 4: ___ Bit 3: ___ Bit 2: ___ Bit 1: ___



2. [32 points]
Design the following circuit in VHDL. This circuit is required to be loaded with value Q4 Q3 Q2 Q1 = "1100" if the asynchronous reset control signal is logic '1'.




```
library ieee;
use ieee.std_logic_1164.all;
entity flop is port(
  clk, reset : in std_logic;
  Q4,Q3,Q2,Q1 : out std_logic);
end flop;
architecture archi of flop is
signal t1,t2,t3,t4 : std_logic
process (clk,reset)
begin
  if (reset='1') then
    Q4<='1';
    Q3<='1';
    Q2<='0';
    Q1<='0';
  elsif (rising_edge(clk)) then
    t1<=t2;
    t2<=t3 xor t1;
    t3<=t4;
    t4<=t1;
  end if;
end process;
  Q4<=t4;
  Q3<=t3;
  Q2<=t2;
  Q1<=t1;
end archi;
```

(2). Design VHDL testbench for the above circuit.

```
library ieee;
use ieee.std_logic_1164.all
entity flop_tb
end flop_tb;
architecture beh of flop_tb is
component flop (
  clk, D, reset : in std_logic;
  Q4,Q3,Q2,Q1 : out std_logic);
end component;
signal clk, D, reset: std_logic;
signal Q4,Q3,Q2,Q1 : std_logic;
begin
uut : flop port map (clk, D, reset, Q4,Q3,Q2,Q1);
process
begin
  reset <='1';
  wait 5ns;
  reset <='0';
  wait 5ns;
  wait;
end process;
end beh;
```

missing clk

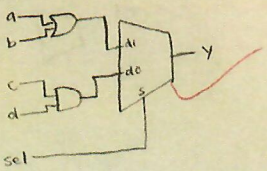
(3). The above circuits are preloaded with value
Q4Q3Q2Q1 = "1100".
What are the output values **after two clock cycles and reset signal is logic '0'**?

Your answer: Q4 = ___, Q3 = ___, Q2= ___, Q1= ___.

3. [16 points] Draw the synthesized schematic for the VHDL code.

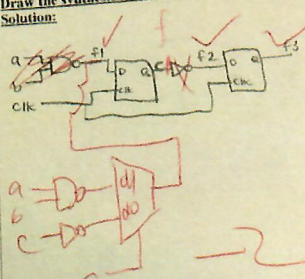
```
signal a, b, c, d, sel: std_logic;
...
...

process (a , b , c, d, sel)
begin
case (sel) is
when '0' => y <= a or b;
when '1' => y <= c and d;
end case;
end process;
```

Draw the synthesized schematic for the VHDL code.
Solution:




```
signal a, b, c: std_logic;
signal f1, f2, f3;
...
...

f1 <= a nand b;

process (clock )
begin
 if (rising_edge(clock)) then
   if( s = '1') then
      f2 <= f1;
   else
      f2 <= not c;
   end if;
    f3 <= f2;
 end if;
end process;
```

Draw the synthesized schematic for the VHDL code.
Solution:



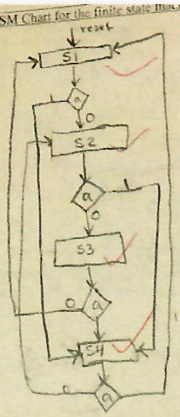
4. [24 points] Draw ASM Chart for the finite state machine and complete the waveform.

```
Library ieee;
Use ieee.std_logic_1164.all;
entity asm_chart (
   a, reset, clk: in std_logic;
       y: out std_logic );
end asm_chart;
```

```
architecture beh of asm_chart is
type  st is ( S1, S2, S3, S4 );
signal  cs, ns:  st;
begin

    process ( a ,  cs )
    begin
    case (cs) is
    when S1=>  if (a = '0') then
                    ns <= S2 ;
                    y  <= '0';
                else
                    ns  <= S4 ;
                    y  <= '1';
                end if;
    when S2=>  if(a = '0') then
                    ns  <= S4 ;
                    y  <= '1';
                else
                    ns <= S3 ;
                    y  <= '0';
                end if;
    when S3=>  if (a = '0') then
                    ns <= S1 ;
                    y  <= '0';
                else
                    ns <= S2 ;
                    y  <= '0';
                end if;
    when S4=>  if (a = '0') then
                    ns <= S1 ;
                    y  <= '1';
                else
                    ns <= S2 ;
                    y  <= '0';
                end if;
    when others=> ns <= S1;
                y  <= '0';
    end case;
    end process;

process (  clk ,   reset)
begin
    if( reset = '1') then
          cs <= S1 ;
    elsif (rising_edge(clk)) then
          cs <= ns ;
    end if;
end process;

end beh;
```

1. Draw ASM Chart for the finite state machine.

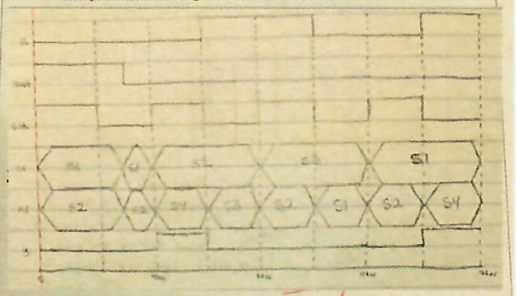
Your Solution:



Missing

-3

2. Complete the following waveform for signals cs, ns, y.

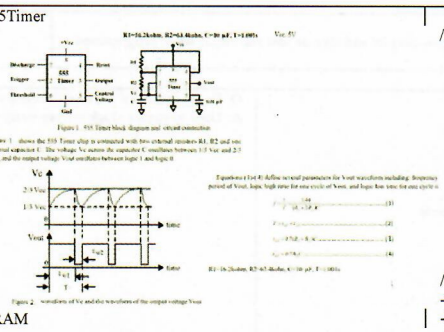


-1

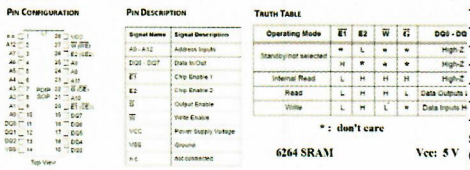
//BlockRAM Memory

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity ram is
port (
     address :in   std_logic_vector ( 3 downto 0);
     data    :inout std_logic_vector ( 7 downto 0);
     cs      :in   std_logic;
     we      :in   std_logic;
     oe      :in   std_logic );
end ram;

architecture beh_ram of ram is
   type memory is array (0 to 15)of std_logic_vector (7 downto 0);
   signal mem : memory ;
begin
  -- Memory Write Block
    process (address, data, cs, we)
    begin
      if (cs = '1' and we = '1') then
         mem(conv_integer(address)) <= data;
      end if;
    end process;
  -- Memory Read Block
   process (address, cs, we, oe, mem)
   begin
      if (cs = '1' and we = '0' and oe = '1')  then
          data <= mem(conv_integer(address));
      else
          data <=  (others => 'Z' );
      end if;
   end process;

 end beh_ram;
```

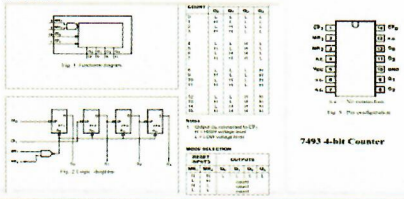
//555Timer



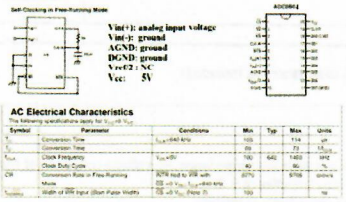
//SRAM



//Counter

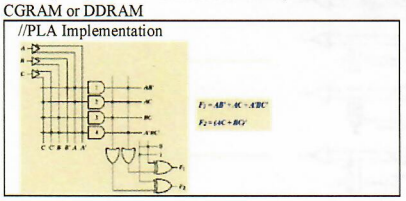


//ADC pins



//LCD Character Display
-Uses 4-bit interface
-Data = 8bit
-Upper nibble transferred first, followed by lower nibble
-RW/RS must be set 40ns before asserting
-RW/RS must be hold 10ns after de-asserting
-1 microsec between transfer of upper and lower nibble
-40microsec time between transfer of two 8-bit data
-Consist of: DDRAM, CGROM, CGRAM
-Commands: clear display, return cursor home, entry mode set, display on/off, cursor display shift, function set, set CGRAM address, Set DDRAM address, Read busy flag and address, write data to CGRAM or DDRAM, Read data from CGRAM or DDRAM

//PLA Implementation



//Logic Gates

//Traffic Light Controller

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.ALL;
use ieee.std_logic_unsigned.ALL;

entity tlc is
   Port ( clk, rst: in std_logic;
          R,G,Y: out std_logic;
          cnt2: out std_logic_vector(3 downto 0));
end tlc;

architecture Behavior of tlc is
       signal cnt: std_logic_vector(3 downto 0):= "0000";
       signal state: std_logic_vector(2 downto 0):= "000";
begin
       process(clk, rst)
      begin
           --Reset State
          if(rst = '1') then
                state <= "000";
                R <= '1'; G <= '0'; Y <= '0';
                cnt <= "0000";
           elsif (clk'event and clk = '1') then
                case state is
                --Red Light, State
                when "000" =>
                --Count=8, Next State = S1
                      if(cnt = "1000") then
                            cnt <= "0000";
                           state <= "001";
                --Count<8, State=0
                      else
                           cnt <= cnt + 1;
                           R <= '1'; G <= '0'; Y <= '0';
                      end if;

                --Yellow Light
                 when "001" =>
                 --Count=12, Next State = S2
                      if(cnt="1100") then
                            cnt <= "0000";
                            state <= "010";
                 --Count<12, State=1
                      else
                            cnt <= cnt + 1;
                            R <= '0'; G <= '0'; Y <= '1';
                       end if;

                --Green Light
                when "010" =>
                --Count=15, Next State = S0
                      if(cnt="1111") then
                            cnt <= "0000";
                            state <= "000";

                 --Count<15, State = S2
                      else
                            cnt <= cnt + 1;
                            R <= '0'; G <= '1'; Y <= '0';
                       end if;

                when others =>
                            state <= "000";
                            cnt <= "0000";
                end case;
          end if;
      end process;
```

//ASM Charts
ASM diagram:
-Advantages: larger state diagrams, proper state diagram conditions automatically satisfied, easily converted to other forms
-Logic signal high = asserted, low = deasserted

//Tap Control Signals
-Test Access Port(TAP) includes these signals:
Test Clock Input(TCK) –Clock for test logic
- Can run at different rate from system clock
Test Mode Select(TMS) –Switches system from functional to test mode
Test Data Input(TDI) – Accepts serial test data and instructions –used to shift in vectors or one of many test instructions
Test Data Output(TDO) –Serially shifts out test results captured in boundary scan chain (or device ID or other internal registers)
Test Reset (TRST) – Optional asynchronous TAP controller reset

//JTAG
-Can operate at chip, PCB, and system levels

//Spartan3E FPGA
XC3S500E = 500K system gates, 10,476 logic cells
-5 elements:
1. CLB: implement gates and flip flops
2. I/O Blocks: provide off-chip connectivity
3. Block RAM: on-chip 18-Kbit Static RAM storage
4. Multiplier: Hardwired 18x18 multiplier cell
5. DCM: Digital clock manager for clock generation and Distribution

//Complex Programmable Logic Device(CPLD)
-Field Programmable Device(FPD): refers to any type of IC used for implementing digital hardware, where chip can be configured by user for different designs.

-Programmable Logic Array(PLA)
-small FPD, two levels of logic (AND,OR)
-PLAs introduced 1970s by Philips
-main drawbacks:
Expensive to manufacture
Poor speed-performance
Both due to two levels of configurable logic

-Programmable Array Logci(PAL)
-small FPD, AND followed by fixed OR

-Generic Array Logic(GAL)
-invented by lattice semiconductor
-improvement on PAL
-able to take place of many PAL devices
-Primary benefit: erasable and reprogrammable

-CPLDs: more complex PLD that consists of multiple SPLD-like blocks on single chip

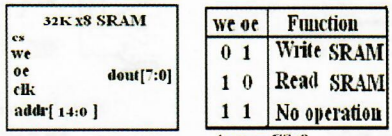
-PLDs: manufactured in "blank"/"erased" form.

-Erasable Programmable Read Only Memory(EPROM):
-Retain data when power off, non-volatile
- Array of floating-gate transistors individually programmed by electronic device that supplies higher voltages than those normally used in digital circuits.
-Erased by exposing to strong ultraviolet light from mercury-vapor Source
-Easily recognizable by transparent fused quartz window in top of package, through which silicon chip visible, which permits exposure to UV light during erasing.
-Electrically Erasable Read Only Memory (EEPROM)

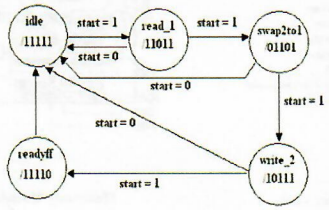
//Spartan3E Specs

Device	System Gates	Equivalent Logic Cells	CLB Array (One CLB = Four Slices) Rows	Columns	Total CLBs	Total Slices	Distributed RAM bits[1]	Block RAM bits[1]	Dedicated Multipliers	DCMs	Maximum User I/O	Maximum Differential I/O Pairs
XC3S100E	100K	2,160	22	16	240	960	15K	72K	4	2	108	40
XC3S250E	250K	5,508	34	26	612	2,448	38K	216K	12	4	172	68
XC3S500E	500K	10,476	46	34	1,164	4,656	73K	360K	20	4	232	92
XC3S1200E	1200K	19,512	60	46	2,168	8,672	136K	504K	28	8	304	124
XC3S1600E	1600K	33,192	76	58	3,688	14,752	231K	648K	36	8	376	156

//Write/Read SRAM



Outputs: we_1,we_2, oe_1, oe_2, swapdone



Swap two SRAM contents at address location 15'h66.

```
.....

assign address = 15'h 66;

always@(posedge swap_sw or negedge swapdone)
  begin
   if (swapdone==0) start <= 1'b0;
   else start <= 1'b1;
  end
 always@(start or c_state)
  begin

      if (start = = 0)  n_state<=idle;
      else
      begin
      case (c_state)
        idle:       n_state <= read_1;
        read_1:     n_state <= swap2to1;
        swap2to1:   n_state <= write_2;
        write_2:    n_state <= readyff;
        readyff:    n_state <= idle;
        default:    n_state <= idle;
      endcase
      end
      end


    always@(c_state)
     begin
     case (c_state)
     idle: begin we_1<=1'b1;we_2<=1'b1;oe_1<=1'b1;oe_2<=1'b1;swapdone<=1'b1;end
     read_1: begin we_1<=1'b1;we_2<=1'b1;oe_1<=1'b0;oe_2<=1'b1;swapdone<=1'b1;end
     swap2to1: begin
     we_1<=1'b0;we_2<=1'b1;oe_1<=1'b1;oe_2<=1'b0;swapdone<=1'b1;end
     write_2: begin we_1<=1'b1;we_2<=1'b0;oe_1<=1'b1;oe_2<=1'b1;swapdone<=1'b1;end
     readyff: begin we_1<=1'b1;we_2<=1'b1;oe_1<=1'b1;oe_2<=1'b1;swapdone<=1'b0;end
     default: begin we_1<=1'b1;we_2<=1'b1;oe_1<=1'b1;oe_2<=1'b1;swapdone<=1'b1;end
     endcase
     end
     always@(negedge clk)
     begin
      if (c_state= =read_1)
       templatch <= temp;
     end
      assign temp = (c_state= =read_1)? databus: 8'b00000000;
      assign databus = ((c_state= =read_1) | (c_state= =swap2to1) | (c_state= =idle)
                        | (c_state= =readyff) )? 8'bZZZZZZZZ: templatch;
     endmodule
```

cnt2 <= cnt; end Behavior;	-Antifuses: open-circuits and take on low resistance when programmed	

//Transmission Lines

-@Steady state: (transmission line impedance not included)

$$Vload = Vsrc * \frac{Zload}{Zsrc + Zload}$$

-Load Reflection coefficient is:

0 when load impedance = transmission line impedance, no voltage reflection at load node

-1 when load is short circuit, load impedance = 0.

1 when load is open circuit, load impedance is infinity.

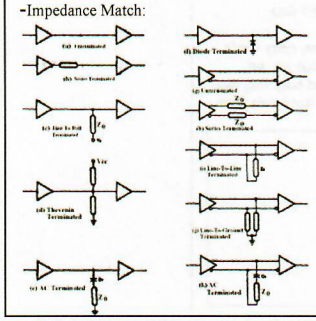
-Reflection Coefficient: $p = \frac{Zt-Z0}{Zt+Z0}$

-Transmission Coefficient: $t = \frac{2Zt}{Zt+Z0}$

-Voltage Delivered to Load $p_B = \frac{Rt-Z0}{Rt+Z0}$

-Voltage reflected back to Source $p_A = \frac{Rs-Z0}{Rs+Z0}$

-Impedance Match:



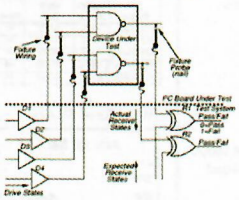
Q: Functionality of dedicated recursive 'fish-bone' network used inside Spartan3E FPGA?

A: Used to ensure clock arrives everywhere at same time.

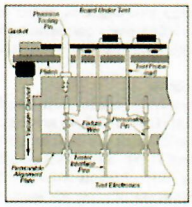
Motivation for Standard

- *Bed-of-nails printed circuit board* tester gone
  - We put components on both sides of PCB & replaced DIPs with flat packs to reduce inductance
    - Nails would hit components
  - Reduced spacing between PCB wires
    - Nails would short the wires
  - PCB Tester must be replaced with built-in test delivery system -- JTAG does that
  - Need standard System Test Port and Bus
  - Integrate components from different vendors
    - Test bus identical for various components
    - One chip has test hardware for other chips

Bed-of-Nails Tester Concept



Bed-of-Nails Tester



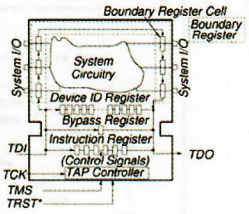
Purpose of Standard

- Lets test instructions and test data be serially fed into a *component-under-test* (CUT)
  - Allows reading out of test results
  - Allows *RUNBIST* command as an instruction
    - Too many shifts to shift in external tests
- JTAG can operate at chip, PCB, & system levels
- Allows control of tri-state signals during testing
- Lets other chips collect responses from CUT
- Lets system interconnect be tested separately from components
- Lets components be tested separately from wires

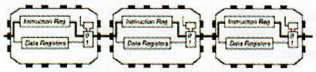
Purpose of Standard

- Lets test instructions and test data be serially fed into a *component-under-test* (CUT)
  - Allows reading out of test results
  - Allows *RUNBIST* command as an instruction
    - Too many shifts to shift in external tests
- JTAG can operate at chip, PCB, & system levels
- Allows control of tri-state signals during testing
- Lets other chips collect responses from CUT
- Lets system interconnect be tested separately from components
- Lets components be tested separately from wires

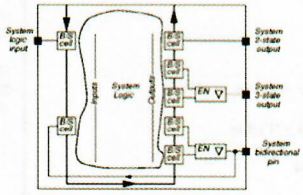
System Test Logic



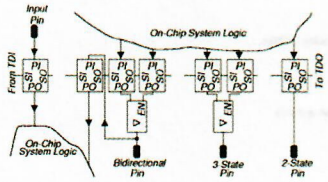
Instruction Register Loading with JTAG



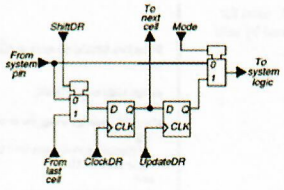
System View of Interconnect



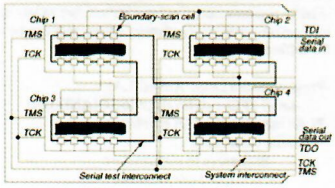
Boundary Scan Chain View



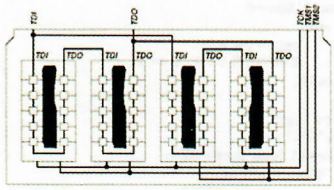
Elementary Boundary Scan Cell



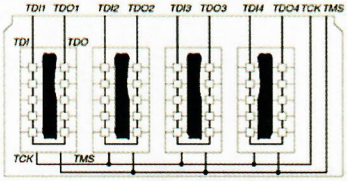
Serial Board / MCM Scan



Parallel Board / MCM Scan



Independent Path Board / MCM Scan



Tap Controller Signals

- *Test Access Port* (TAP) includes these signals:
  - *Test Clock Input* (*TCK*) -- Clock for test logic
    - Can run at different rate from system clock
  - *Test Mode Select* (*TMS*) -- Switches system from functional to test mode
  - *Test Data Input* (*TDI*) -- Accepts serial test data and instructions -- used to shift in vectors or one of many test instructions
  - *Test Data Output* (*TDO*) -- Serially shifts out test results captured in boundary scan chain (or device ID or other internal registers)
  - *Test Reset* (*TRST*) -- *Optional* asynchronous TAP controller reset