```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY myand IS
PORT ( A, B : IN STD_LOGIC;
         C : OUT STD_LOGIC
        );
END myand;
ARCHITECTURE dataflow OF myand IS
BEGIN
            C <= A and B;
END dataflow;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY fa IS
PORT ( A, B, Cin : IN STD_LOGIC;
         Cout, S : OUT STD_LOGIC
        );
END fa;
ARCHITECTURE dataflow OF fa IS
signal M: std_logic;
BEGIN
        M <= A xor B;
        S <= M xor Cin;
        Cout <= ( M and Cin ) or ( A and B );
END dataflow;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY rca4 IS
Port ( A, B: in std_logic_vector(3 downto 0);
       Cin: in std_logic;
       Cout: out std_logic;
       S: out std_logic_vector(3 downto 0));
END rca4;
ARCHITECTURE structural OF rca4 IS
component fulladder
PORT (A, B, Cin : IN STD_LOGIC;
         Cout, S : OUT STD_LOGIC
          );
end component;
signal CR: STD_LOGIC_VECTOR(3 downto 0);
BEGIN
U0: fulladder port map ( A => A(0), B => B(0),
                          Cin => Cin,
                         Cout=> CR(0), S => S(0)
                         );
U1: fulladder port map ( A => A(1), B => B(1),
                          Cin => CR(0),
                          Cout=> CR(1), S => S(1)
                         );
U2: fulladder port map ( A => A(2), B => B(2),
                          Cin => CR(1),
                         Cout=> CR(2), S => S(2)
                         );
U3: fulladder port map ( A => A(3), B => B(3),
                          Cin => CR(2),
                          Cout=> CR(3), S => S(3)
                          );
END structural;
```

```
--Unsigned 8x4-bit Multiplier
module multiplier(A, B, RES);
input [7:0] A;
input [3:0] B;
output [11:0] RES;
assign RES = A * B;
endmodule
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY myand_tb IS
END myand_tb;
ARCHITECTURE beh OF myand_tb IS
component myand
PORT( A, B : IN STD_LOGIC;
          C : OUT STD_LOGIC   );
end component;
signal TA, TB : STD_LOGIC;
signal TC: STD_LOGIC;
BEGIN
              uut: myand port map (
              A => TA,   B => TB,
              C => TC  );
              Process
              Begin
                TA <='0'; TB<='0';
                Wait for 10 ns;
                TA <='0'; TB<='1';
                Wait for 10 ns;
                TA <='1'; TB<='0';
                Wait for 10 ns;
                TA <='1'; TB<='1';
                Wait for 10 ns;
                 Wait;
              End process;
END beh;
```

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity dff is
Port ( d, clk: in std_logic;
        q: out std_logic );
End dff;
Architecture beh of dff is
begin
  process( clk )
  begin
    -- if (rising_edge (clk) ) then
      q <= d;
    end if;
  end process;
End beh;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY dff_tb IS
END dff_tb;
ARCHITECTURE beh OF dff_tb IS
component dff
PORT  (
          d, clk : IN STD_LOGIC;
          q : OUT STD_LOGIC
          );
end component;
signal d, clk, q : STD_LOGIC;
BEGIN
uut: dff port map ( d => d,
              clk => clk, q => q );
  Process
  Begin
    clk <= '0'; Wait for 10 ns;
    Clk <= '1'; Wait for 10 ns;
   End process;
   Process
   Begin
     d <='0'; Wait for 8 ns;
     d <='1'; Wait for 20 ns;
    d <='0'; Wait for 8 ns;
     Wait;
  End process;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY shiftreg IS
PORT ( Din, clk, clr : IN STD_LOGIC;
         Q : OUT STD_LOGIC_VECTOR(3 downto 0)
          );
END shiftreg;
ARCHITECTURE beh OF shiftreg IS
signal W: std_logic_vector(3 downto 0);
BEGIN
  process( clk, clr )
  begin
    if (clr = '1') then
      W <= "0000";
    elsif (rising_edge (clk)) then
      W(3) <= Din;
      W(2) <= W(3);
      W(1) <= W(2);
      W(0) <= W(1);
    end if;
  end process;
  Q <= W;
END beh;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY mux IS
PORT( A, B,C,D : IN STD_LOGIC;
         SEL: IN STD_LOGIC_VECTOR(1 downto 0);
          DOUT : OUT STD_LOGIC
        );
END mux;
ARCHITECTURE design OF mux IS
BEGIN
DOUT <= A WHEN SEL = "00" ELSE
           B WHEN SEL = "01" ELSE
          C WHEN SEL = "10" ELSE
          D;
END design;
```

```
/library ieee;
use ieee.std_logic_1164.all;
package MY_PACK is
 function PARITY (X : std_logic_vector)
          return std_logic;
end MY_PACK;
package body MY_PACK is
 function PARITY (X : std_logic_vector)
          return std_logic is
 variable TMP : std_logic;
 begin
   --TMP:= '0';
   TMP := X(0);
   --for J in X'range loop
   for J in 1 to X'high loop
      TMP := TMP xor X(J);
    end loop;        -- works for any size X
    return TMP;
  end PARITY;
end MY_PACK;
```

```
library ieee;
use ieee.std_logic_1164.all;
use work.MY_PACK.all;

entity PAR is
  port( db:  in std_logic_vector(7 downto 0);
        dw: in std_logic_vector(15 downto 0);
        pb, pw: out std_logic);
end PAR;

architecture ARCH1 of PAR is
begin
 pb <= PARITY(db);
 pw <= PARITY(dw);
end ARCH1;
```

Even Parity and Odd Parity Examples

even

```
ab_parity  (f_even)
00_0
01_1
10_1
11_0
f_even = a xor b
```

odd

```
ab_parity  (f_odd)
00_1
01_0
10_0
11_1
f_odd = not ( a xor b);
```

ADD/SUBTRACT
Modulo 2 uses XOR

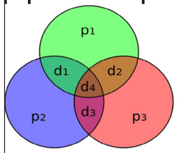
```
0 ± 0 = 0;
0 ± 1 = 1;
1 ± 0 = 1;
1 ± 1 = 0
```

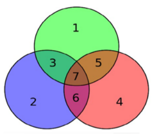
MULTIPLICATION

```
   1 0 1 1
  x 0 1 0 1
   1 0 1 1
  0 0 0 0
 1 0 1 1
0 0 0 0
0 1 0 0 1 1 1
```

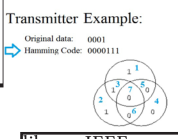
DIVISION

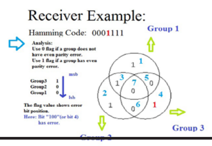
```
                1 0 0 0 1 rem. 1 0 1
1 0 0 1 1|1 0 0 1 0 0 1 1 0
          1 0 0 1 1
                1 0 1 1 0
                1 0 0 1 1
                    1 0 1

             1 rem. 1 0 1 0
1 1 0 0 1|1 0 0 1 1
          1 1 0 0 1
           1 0 1 0

             1 rem.1 0 1 0
1 0 0 1 1|1 1 0 0 1
          1 0 0 1 1
           1 0 1 0
```





Bit position of the data and parity bits

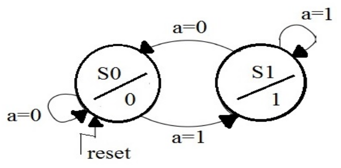



Receiver Example:
Hamming Code: 0001111
Analysis:
Use 0 flag if a group does not have even parity error.
Use 1 flag if a group has even parity error.
Group3 1
Group2 0
Group1 0
msb
lsb
The flag value shows error bit position.
Here: Bit "100"(or bit 4) has error.
Group 1
Group 2
Group 3

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity hamming is
 Port ( hamdin: in std_logic_vector(3 downto 0);
        hamout: out std_logic_vector(7 downto 1) );
end hamming;
architecture Behavioral of hamming is
signal p: std_logic_vector(4 downto 1);
begin
p(1) <= hamdin(3) xor hamdin(1) xor hamdin(0);
p(2) <= hamdin(3) xor hamdin(2) xor hamdin(0);
p(4) <= hamdin(3) xor hamdin(2) xor hamdin(1);
hamout <= p(1) & p(2) & hamdin(3) & p(4) & ham-
din(2) & hamdin(1) & hamdin(0);
end Behavioral;
```

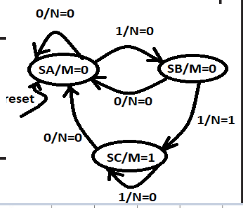
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY moorefsm IS
PORT ( reset, a, clk : IN STD_LOGIC;
           y: OUT STD_LOGIC
         );
END moorefsm;
ARCHITECTURE beh OF moorefsm IS
type state_type is (S0, S1);
signal cs, ns: state_type;
BEGIN
   Process(reset, clk)
   Begin
      If (reset = '1') then
         cs <= S0;
      elsif (rising_edge(clk)) then
         cs <= ns;
      end if;
   End process;
   Process(cs, a)
   Begin
      Case (cs) is
         When S0 => y <= ‘0’;
            If (a=’1’) then
               ns <= S1;
            else
               ns <= S0;
            end if;
         When S1 => y <= ‘1’;
            If (a=’1’) then
               ns <= S1;
            else
               ns <= S0;
            end if;
        When others=> y<= ‘0’; ns <= S0;
      end case;
   end process;
END beh;
```

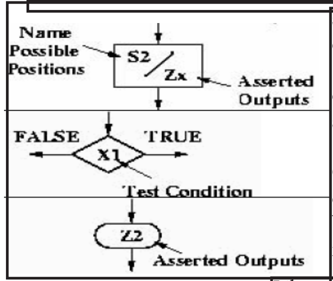
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY moorefsm_tb IS
END moorefsm_tb;
ARCHITECTURE beh OF moorefsm_tb IS
component moorefsm
PORT (
          reset, a, clk : IN STD_LOGIC;
          y: OUT STD_LOGIC
         );
end component;
signal reset, a, clk : STD_LOGIC;
signal y: STD_LOGIC;
BEGIN
   Uut: moorefsm port map ( reset => reset,
                              a => a, clk => clk,
                              y => y );
   reset <= '1', '0' after 5 ns, '0' after 100 ns;
   clk<= ‘0’, ‘1’ after 10 ns, ‘0’ after 20 ns ,
    0’ after 30 ns, ‘1’ after 40 ns,
   ‘0’ after 50 ns, ‘1’ after 60 ns,
   ‘0’ after 70 ns, ‘1’ after 80 ns,
    0’ after 90 ns, ‘1’ after 100 ns;
   a <= ‘0’, ‘1’ after 12 ns, ‘0’ after 56 ns;
END beh;
```



```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY mealyfsm IS PORT
   ( reset, a, clk : IN STD_LOGIC;
    y: OUT STD_LOGIC    );
END mealyfsm;
ARCHITECTURE beh OF mealyfsm IS
type  state_type is (SO,Sl);
signal   cs, ns: state_type;
BEGIN
  Process(reset, elk)
  Begin
    If(reset = '1') then
      cs <= SO;
    elsif (rising_edge(clk)) then
      cs <= ns;
    end if;
  End process;
  Process(cs, a)
  Begin
    Case (cs) is
      When S0 => If (a=’1’) then
                   ns <= S1; y <= ‘1’;
                 else
                   ns <= S0; y <= ‘0’;
                 end if;
      When S1 => If (a=’1’) then
                   ns <= S1; y <= ‘0’;
                 else
                   ns <= S0; y <= ‘1’;
                 end if;
      When others=> y<= ‘0’;  ns <= S0;

    end case;
  end process;
END beh;
```

```
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;
```

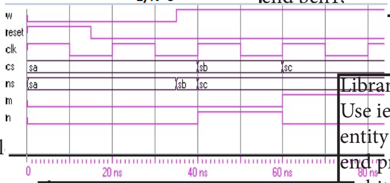




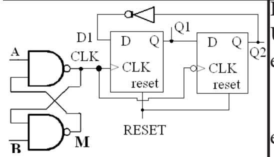
state box- with outputs

optional decision boxes

Optional conditional output box asynchronous/Mealy

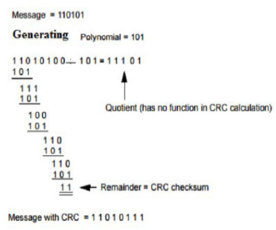


```
Library ieee; --tb for that circuit
Use ieee.std_logic_1164.all;
entity prob1_tb
end prob1_tb;
architecture beh of prob1_tb is
signal a, b, reset:   std_logic;
signal   q1, q2:    std_logic;
component prob1  (a, b, reset: in std_l
                q1, q2:  out  std_logic );
end component;
begin
uut: prob1 port map ( a=> a, b=>b,
          reset=>reset, q1=>q1, q2=>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;
```



```
Library ieee;
Use ieee.std_logic_1164.all;
entity prob1 is
   port (a, b, reset: in std_logic;
               q1, q2:  out  std_logic );
end prob1;
architecture circuit of prob1 is
ignal      clk,  m, d1, t1, t2: std_logic;
egin
   clk <= a  nand m;
   m  <=  b  nand clk;
   d1 <= not t2;
rocess ( clk , reset)
egin
   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 (falling_edge(clk)) then
        t2 <= t1;
   end if;
end process;
q1 <= t1;  q2 <= t2;
end circuit;
```

5. CRC Error Detection


Message = 110101
Generating Polynomial = 101
11010100 ÷ 101 = 11101
101
111
101
100
101
110
101
110
101
11
Quotient (has no function in CRC calculation)
Remainder = CRC checksum
Message with CRC = 11010111

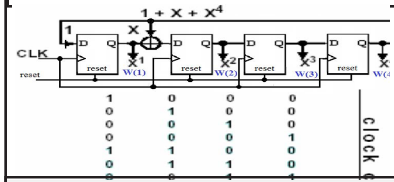
```
Use ieee.std_logic_1164.all;
entity prob2_fsm is
  port( w, Reset, Clk: in std_logic;
                  M, N: out std_logic );
End prob2_fsm;
Architecture beh of prob2_fsm is
type  st is ( SA, SB, SC );
signal  cs, ns:  st;
begin
 process ( w ,  cs )
 begin
  case (cs)
    when SA=>  if (w = ‘0’)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

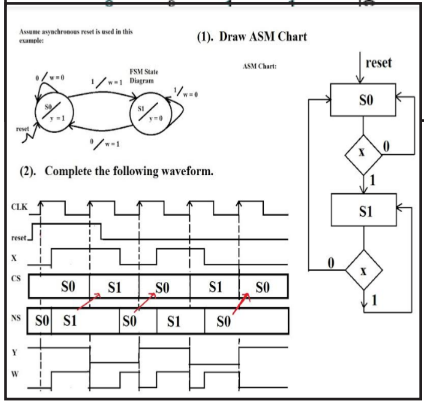
```
Library ieee;
Use ieee.std_logic_1164.all;
entity prob2_fsm_tb
end prob2_fsm_tb;
architecture beh of prob2_fsm_tb is
signal   w, Reset, Clk:  std_logic;
signal               M, N::    std_logic;
component prob2_fsm ( w, Reset,
                         Clk: in std_logic;
                    M, N: out std_logic );
end component;
begin
uut: prob2_fsm port map ( w,
                    Reset, Clk,  M, N);
reset <= ‘1’, ‘0’ after 15 ns;
w      <= ‘0’, ‘1’ after 35 ns;
process
begin
     Clk <= ‘1’; wait for 10 ns;
     Clk <= ‘0’; wait for 10 ns;
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;
```

```
LIBRARY IEEE; --LFSR
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY lfsr IS
PORT (  reset, clk: IN STD_LOGIC;
          Q : OUT STD_LOGIC_VECTOR(4 downto 1)
         );
END lfsr;
ARCHITECTURE beh OF lfsr IS
signal W: std_logic_vector(4 downto 1);
BEGIN
   process( clk, reset )
   begin
      if (reset='1') then
         W <= ( 1=>'1', others => '0' );
      elsif (rising_edge (clk)) then
         W <= W(3 downto 2) & ( W(1) xor W(4) ) & W(4);
      end if;
   end process;
   Q <= W;
END beh;
```

```
LIBRARY IEEE; --lfsr tb
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY lfsr_tb is
End lfsr_tb;
ARCHITECTURE beh OF lfsr_tb IS
Component lfsr
PORT (  reset, clk: IN STD_LOGIC;
          Q : OUT STD_LOGIC_VECTOR(4 downto 1)
         );
END Component;
signal reset, clk: std_logic;
signal Q: std_logic_vector(4 downto 1);
BEGIN
uut: lfsr port map( reset=>reset, clk=>clk, Q=>Q );
   Process
   Begin
      Clk <= '0';
      Wait for 10 ns;
      Clk <= '1';
      Wait for 10 ns;
   End process;
   Process
   Begin
      reset <='1';
      Wait for 6 ns;
      reset <='0';
      Wait for 40 ns;
      Wait;
   End process;
END beh;
```





```
library ieee; --Clock division
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity display_counter_vhd is
Port( clk: in std_logic;
        load: in std_logic;
        updown: in std_logic;
        din: in std_logic_vector (3 downto 0);
        cntout: out std_logic_vector (3 downto 0);
        led: out std_logic_vector (3 downto 0);
        clkout: out std_logic
       );
end display_counter_vhd;
architecture behavioral of display_counter_vhd is
signal cnt_div: std_logic_vector (9 downto 0);
signal cnt: std_logic_vector (3 downto 0);
signal clk2: std_logic;
begin
process(clk)
begin
if rising_edge (clk) then
  if(cnt_div = 999) then --(N-1), here N=1000
    cnt_div <= (others => '0');
    clk2 <= '1';
  elsif (cnt_div < 499) then --(N/2  - 1)
    cnt_div <= cnt_div + 1;
    clk2 <= '1';
  else
    cnt_div <= cnt_div + 1;
    clk2 <= '0';
  end if;
end if;
end process;
process (clk2, load, updown)
--process active on event of clk2,load, or updown
begin
    if(load = '1') then
       cnt <= din;
    elsif rising_edge (clk2) then
       if(updown = '1') then
          cnt <= cnt + 1;
       else
          cnt <= cnt - 1;
       end if;
    end if;
end process;

cnt_out <= cnt;  --output count
clkout <= clk2; --output clk
led <= cnt;  --led output value
end behavioral;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity mytt is
port( A, B: in std_logic_vector(5 downto 0);
        Y: out std_logic_vector(11 downto 0));
end mytt;
architecture arch_tt of mytt is
constant C: std_logic_vector(2 downto 0) := "100";
begin
process(A,B)
begin
   Y(2 downto 0) <= A(2 downto 0);
   Y(3) <= A(3) and B(3);
   Y(5 downto 4) <= (A(5)and B(5))&(A(4) or B(4));
   Y(8 downto 6) <= B(2 downto 0);
   Y(11 downto 9)<= C;
end process;
end arch_tt;
```

```
library ieee; --pos edg dff
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 (rising_edge(CLK)) then
   Q <= D;
end if;
end process;
end archi;
```

```
library ieee; --n.edg dff asy rst
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 (falling_edge(CLK))
then
   Q <= D;
end if;
end process;
end archi;
```

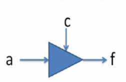
```
library ieee; --p.edg sync set
use ieee.std_logic_1164.all;
entity flop is
port( C, D, S : in std_logic;
        Q : out std_logic );
end flop;
architecture archi of flop is
begin
process (C)
begin
   if (C'event and C='1') then
      if (S='1') then
         Q <= '1';
      else
         Q <= D;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

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

```
library ieee; --4b p.edg clk, asy set, clk en
use ieee.std_logic_1164.all;
entity flop is
port( C, CE, PRE : in std_logic;
        D: in std_logic_vector(3 downto 0);
        Q : out std_logic_vector (3 downto 0)
      );
end flop;
architecture archi of flop is
begin
process (C, PRE)
begin
   if (PRE='1') then
      Q <= "1111";
   elsif (C'event and C='1')then
      if (CE='1') then
         Q <= D;
      end if;
   end if;
end process;
end archi;
```

```
library ieee; --tristate
use ieee.std_logic_1164.all;
entity three_st is
port( T, I : in std_logic;
        O : out std_logic
      );
end three_st;
architecture archi of three_st is
begin
process (I, T)
begin
   if (T='0') then
      O <= I;
   else
      O <= 'Z';
   end if;
end process;
end archi;
```

Tri-State Buffer



c	a	f
0	0	Z
0	1	Z
1	0	0
1	1	1

```vhdl
library ieee; --4b  up/down counter, asy. reset
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity counter is
port( C, CLR, UP_DOWN : 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, CLR)
begin
   if (CLR='1') then
      tmp <= "0000";
   elsif (C'event and C='1') then
      if (UP_DOWN='1') then
         tmp <= tmp + 1;
      else
         tmp <= tmp - 1;
      end if;
   end if;
end process;
Q <= tmp;
end archi;
```

```vhdl
library ieee; --8b Sft L Reg, P.Edg Clk, S.i/O
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity shift is
port( C, SI : 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)
begin
   if (C'event and C='1') then
      for i in 0 to 6 loop
         tmp(i+1) <= tmp(i);
      end loop;
      tmp(0) <= SI;
   end if;
end process;
SO <= tmp(7);
end archi;
```

```vhdl
library ieee; --unsigned 8b adder/subtractor
use ieee.std_l ogic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity addsub is
port( A, B : in std_logic_vector(7 downto 0);
        OPER : in std_logic;
        RES : out std_logic_vector(7 downto 0) );
end addsub;
architecture archi of addsub is
begin
   RES <= A + B when OPER='0'
   else A - B;
end archi;
```

```vhdl
library ieee; --Concatenation
use ieee.std_logic_1164.all;
entity mytt2 is
port( A, B: in std_logic_vector(2 downto 0);
Y: out std_logic_vector(14 downto 0));
end mytt2;
architecture arch_tt of mytt2 is
constant C: std_logic_vector(2 downto 0) := "100";
begin
Y <= A & B & C & C & "110";
end arch_tt;
```

```vhdl
library ieee; -- 8b Sft-L Reg, Pedg Clk, Asy Clr, S I/O
use ieee.std_logic_1164.all;
entity shift is
port( C, SI, CLR : 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, CLR)
begin
   if (CLR='1') then
      tmp <= (others => '0');
   elsif (C'event and C='1') then
      tmp <= tmp(6 downto 0) & SI;
   end if;
end process;
SO <= tmp(7);
end archi;
```

```vhdl
library ieee; --8b Sft L Reg, PEdg Clk, Asy Load, S I/O
use ieee.std_logic_1164.all;
entity shift is
port( C, SI, ALOAD : in std_logic;
        D : in std_logic_vector(7 downto 0);
        SO : out std_logic
       );
end shift;
architecture archi of shift is
signal tmp: std_logic_vector(7 downto 0);
begin
process (C, ALOAD, D)
begin
   if (ALOAD='1') then
      tmp <= D;
   elsif (C'event and C='1') then
      tmp <= tmp(6 downto 0) & SI;
   end if;
end process;
SO <= tmp(7);
end archi;
```

```vhdl
library ieee; --4-1 mux if statement
use ieee.std_logic_1164.all;
entity mux is
port ( a, b, c, d : in std_logic;
         s : in std_logic_vector (1 downto 0);
         o : out std_logic );
end mux;
architecture archi of mux is
begin
process (a, b, c, d, s)
begin
   if (s = "00") then
      o <= a;
   elsif (s = "01") then
      o <= b;
   elsif (s = "10") then
      o <= c;
   else
      o <= d;
   end if;
end process;
end archi;
```

```vhdl
library ieee; --comparator
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity compar is
port( A, B : in std_logic_vector(7 downto 0);
        CMP : out std_logic
);
end compar;
architecture archi of compar is
begin
   CMP <= ‘1’ when A >= B else ‘0’;
end archi;
```

```vhdl
library ieee; --decoder
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
   res <= “00000001” when sel = “000”
   else “00000010” when sel = “001”
   else “00000100” when sel = “010”
   else “00001000” when sel = “011”
   else “00010000” when sel = “100”
   else “00100000” when sel = “101”
   else “01000000” when sel = “110”
   else “10000000”;
end archi;
```

```vhdl
library ieee; --decoder
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;
```

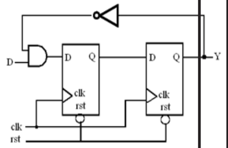
```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity mux is
port ( a, b, c, d : in std_logic;
         s : in std_logic_vector (1 downto 0);
         o : out std_logic
       );
end mux;
architecture archi of mux is
begin
process (a, b, c, d, s)
begin
   case s is
      when "00" => o <= a;
      when "01" => o <= b;
      when "10" => o <= c;
      when others => o <= d;
   end case;
end process;
end archi;
```

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity my_block is
port( I1 : in std_logic; I2 : in std_logic;
        O : out std_logic );
end my_block;
architecture arch1 of my_block is
. . .
end arch1;

library ieee;
use ieee.std_logic_1164.all;
entity top is
port( DI_1, DI_2, DI_3, DI_4 : in std_logic;
        DOUT1, DOUT2 : out std_logic
);
end top;
architecture top_arch of top is
component my_block
port ( I1 : in std_logic; I2 : in std_logic;
         O : out std_logic
        );
end component;
begin
inst1: my_block port map ( I1=>DI_1,
                  I2=>DI_2, O=>DOUT1
                   );
Inst2: my_block port map ( DI_3, DI_4,
DOUT2 );
end top;
```

```vhdl
library ieee; --priority encoder
use ieee.std_logic_1164.all;
entity priority is
port ( sel : in std_logic_vector (7 downto 0);
         code :out std_logic_vector (2 downto 0)
       );
end priority;
architecture archi of priority is
begin
code <= "000" when sel(0) = '1' else
          "001" when sel(1) = '1' else
          "010" when sel(2) = '1' else
          "011" when sel(3) = '1' else
          "100" when sel(4) = '1' else
          "101" when sel(5) = '1' else
          “110” when sel(6) = ‘1’ else
          “111” when sel(7) = ‘1’ else
          “000”;
end archi;
```

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity adder is
port( A, B : in std_logic_vector(7 downto 0);
        SUM : out std_logic_vector(7 downto 0);
        CO : out std_logic );
end adder;
architecture archi of adder is
signal tmp: std_logic_vector(8 downto 0);
begin
  tmp <= conv_std_logic_vector((conv_integer(A) +
                                        conv_integer(B)),9);
  SUM <= tmp(7 downto 0);
  CO <= tmp(8);
end archi;
-- Or: Res <= ("0" & A) + ("0" & B);
```

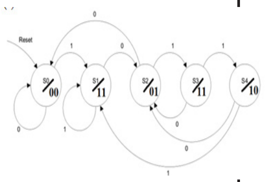
```
//pmidterm dff with not and
module prob2(clk,d,rst,y);
input clk,d,rst;
output y;
reg y,n;
wire m;
assign m=d&~y;
always @(posedge clk or negedge rst)
beigin
  if (~rst)
  begin
    n<=0; y<=0;
  end
  else
  begin
    n<=m; y<=n;
  end
end
endmodule
```



```
//Shift register (4 bit)
module dffb(clk,clr,load,sft,db,qb);
input clk, clr, load, sft;
input [3:0] db;
output [3:0] qb;
reg [3:0] qb;
always @(posedge clr or posedge clk)
begin
  if (clr) qb<=0;
  else if (load)
     qb<=db;
  else if (sft)
     qb<={1'b0; qb[3:1]};
     //qb[3]<=1'b0;
     //qb[2]<=qb[3];
     //qb[1]<=qb[2];
     //qb[0]<=qb[1];
end
endmodule
```

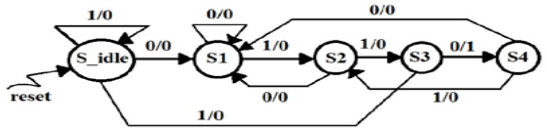
```
`timescale 1ns/1ps
module dffa_tb;
reg       clk, clr, load;
reg [3:0] da;
wire [3:0] qa;
dffa uut ( clk, clr, load, da, qa );
initial  clk = 0;
always   #10 clk = ~ clk;
initial
begin
   clr = 1;    da  = 4'b1011;
   #24  clr = 0;  load = 1;
   #24  load = 0;
   #60 $stop;
end
endmodule
```

```
module moorefsm(clk,rst,a,y); //moore fsm
input clk, rst,a;
output [1:0] y;
reg [1:0] y;
parameter s0=3'b000, s1=3'b0001, s2=3'b010,
s3=3'b011, s4=3'b100;
reg [2:0] cs, ns;
always @(posedge clk or posedge rst)
begin
  if (rst) 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    ns = s2;
      default: ns = s0;
  endcase
end
always@(cs )
begin
  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
```



```
//state machine to find 0110
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
```

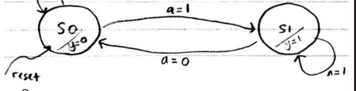
```
//8-bit Shift-Left Reg w/ PosEdge Clk,
//Async paralel Load, Serial In, Serial Out
module shift (C, ALOAD, SI, D, SO);
input C, SI, ALOAD;
input [7:0] D;
output SO;
reg [7:0] tmp;
always @(posedge C or posedge ALOAD)
begin
  if (ALOAD)
    tmp <= D;
  else
    tmp <= {tmp[6:0], SI};
end
assign SO = tmp[7];
endmodule
```



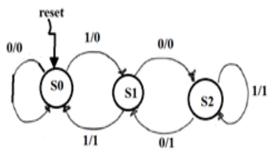
```
//Unsigned 8x4-bit Multiplier
module compar(A, B, RES);
input [7:0] A;
input [3:0] B;
output [11:0] RES;
assign RES = A * B;
```

<u>Dataflow</u>
```
assign y=a&b;
```
<u>behaviorial design</u>
```
always @(posedge clk)
begin
```

```
//Moore State Machine
module moore_fsm(rst, clk, a y);
input rst, clk, a;
output y;
reg y;
parameter s0=1'b0, s1=1'b1;
reg cs, ns;
 //state register block
always @(posedge rst or posedge clk)
begin
  fif (rst) cs<=s0;
  else cs<=ns;
end
//combinational next state block
always @(cs or a)
begin
  case (cs)
    s0: if (a) ns=s1;
         else ns=s0;
    s1: if (a) ns=s1;
         else ns=s0;
    default: ns=s0;
  endcase
end
//output combinational block
always @(cs)
begin
  case(cs)
    s0: y=0;
    s1: y=1;
    default y=0;
  endcase
end
endmodule
```



```
//finite state machine, 3 states
module fsm(clk,rst,a,y);
input clk,rst,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 rst)
begin
  if (rst) 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
```



Metastability

- The output of an edge-triggered flipflop has two valid states: high and low. To ensure reliable operation, designs must meet the timing req. The input to the flipflop must be stable for a min time before the clock edge (register setup time or t_{SU}) and a min time after the clock edge (register hold time or t_H). Specific values for t_{SU} and t_H are provided in each device family data sheet.
- In non-synchronous systems, if the asynch. input signals violate a flipflop's timing requirements, the output of the flipflop can become metastable. Metastable outputs oscillate/hover between high and low states for some time = system failure. Therefore, you must analyze the metastability characteristics of a device to determine the reliability of a non-synchronous design. In synchronous systems, the input signals always meet the flipflop's timing requirements; therefore, metastability does not occur.
- Violating a flipflop's setup or hold time can cause its output to become metastable. When a flipflop is in a metastable ("in between") state, the output hovers at a voltage level between high and low, causing the output transition to be delayed beyond the specified clock-to-output delay (t_{CO}). The additional time beyond t_{CO} that a metastable output takes to resolve to a stable state is called the settling time (t_{MET}). Not every transition that violates the setup or hold times results in a metastable output. Generally, flipflops will quickly return to a stable state
- When a flipflop's data input complies with minimum setup (t_{SU}) and hold (t_H) times, the output passes from one stable state to another (i.e., from high to low or low to high) without an additional delay.
- However, when a flipflop's data input violates the setup or hold time, the flipflop is marginally triggered, and the output may not immediately resolve to either of the two stable states within the specified time. This marginal triggering can cause the output to glitch or to remain temporarily at a metastable state between the high and low logic levels, taking longer to return to a stable state. Either condition increases the delay from the clock transition to a stable output.

$$MTBF = \frac{e^{(C2 \times tMET)}}{C_1 \times f_{CLOCK} \times f_{DATA}}$$

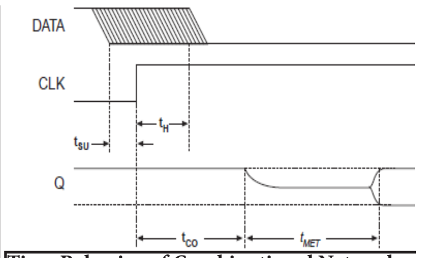
- The f_{CLOCK} parameter refers to the system clock frequency while the f_{DATA} parameter refers to the data transfer frequency. The t_{MET} parameter is the additional time allowed by the system for the flipflop to settle to a stable state. The constants C1 and C2 vary according to the process technology used to manufacture the device.
- The constants C1 and C2 are determined by plotting the natural log of MTBF versus tMET and performing a linear regression analysis on the data. The y- intercept and slope of the resulting line determine the values of C1 and C2. The formulas for the constants C1 and C2 are shown below

$$C2 = \frac{\Delta \ln(MTBF)}{\Delta t_{MET}} \quad , \quad C1 = \frac{e^{(C2 \times tMET)}}{MTFB \times f_{CLOCK} \times f_{DATA}}$$

- [Given settling time (t_{MET}).] The t_{MET} delay is the additional time required for the flipflop to resolve to a legal state, i.e. the difference between the minimum system clock period and the actual clock period. You can also use the metastability equation to determine the t_{MET} delay required for a given MTBF value.

$$t_{MET} = \frac{\ln(MTBF \times f_{CLOCK} \times f_{DATA} \times C1)}{C2}$$

- Reduce metastability in a system.:If an asynchronous signal is fed to several flipflops, the probability that a metastable event increaseses. Avoid metastability by using output of the synchronizing flipflop throughout the system rather than asynchr. signal.
- Avoid the negative effects of metastability by adding the t_{MET} calculated for a specific MTBF to the worst-case timing delay calculations, giving the output of the synchronizing flipflops time to settle. Faster devices provide faster t_{CO} and t_{SU} times, which provide additional time for the t_{MET} delay without sacrificing overall system speed.
- Metastability only affects flipflops used to synchronize data from asynch. systems. Good metastability characteristics: add a small t_{MET} delay to the t_{CO} delay to achieve a high MTBF value.



Time Behavior of Combinational Networks

Gate Delays

- When input to a logic gate is changed, the output will not change immediately.
- The switching elements within a gate take a finite time to react to a change (transition) in input.
- Such delay is called the propagation delay of the logic gate (t_p)
- The propagation delay for a 0 to 1 output change (t_{pLH}) may be different than the delay for a 1 to 0 change (t_{pHL}).
- gate delay: Δ time at inp to cause change at outp
- min delay: typical/nominal delay – max delay for both worst case and best case
- rise time: time for output to transition from low to high voltage
- fall time: time for output to transition from high to low voltage
- pulse width: time that an output stays high or stays low between changes

Effect of gate delays

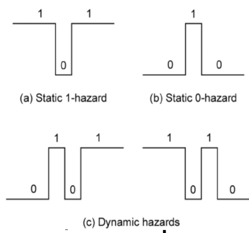
- The analysis of combinational circuits ignoring delays can predict only the steady-state behavior of a circuits. That is they predict a circuit's output as a function of its inputs under the assumption that the inputs have been stable for a long time, relative to the delays into the circuit's electronics.
- A circuit's output may produce a short pulse (often called a glitch) at a time when steady state analysis predicts that the output should not change.
- A Race condition is when a device's output depends on 2(+) nearly simultaneous events to occur, and where which signal arrives first will change the output of the circuit.

Glitch/Hazard

- A **glitch** is an unwanted pulse at the output of a combinational logic network – a momentary change in an output that should not have changed.
- A circuit with the potential for a glitch is a **hazard**.
- A circuit with hazard may or may not have a glitch depending on input patterns and the electric characteristics of the circuit.
- Hazards are potential unwanted transients that occur in the output when different paths from input to output have different propagation delays.
- The fundamental strategy for eliminating an hazard is to add redundant prime implicants (extra prime implicants won't change F, but can cause F to be asserted independently of the change to the input that cause the hazard).
- A properly designed two level AND-OR circuit based on a SoP expression has no static 0-hazards. A static 0-hazard would exist only if both a variable and its complement were connected to the same AND gate, which would be a nonsense (A*A'*X=0)
- A properly designed two level OR-AND circuit of a Product Of Sums expression has no static 1-hazards. A static 1-hazard would exist only if both a variable and its complement were connected to the same OR gate, which would be a nonsense (A+A'+X=1)

Dynamic Hazard

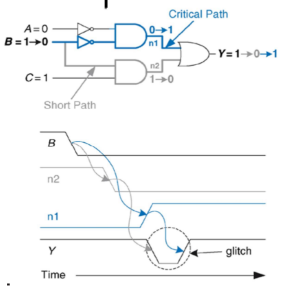
- If there are 3+ paths from an input or its complement to the output, the circuit has the potential for a dynamic hazard.
- 3+ paths only in a multi-level networks.
- If you need a hazard free network, it is best to use a 2-level network and use the techniques shown earlier to eliminate the static hazards.

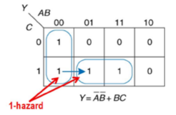


The output undergoes a momentary transition when it is expected to remain unchanged

The output changes multiple times as the result of a single input transition

Detection of Static 1-hazards



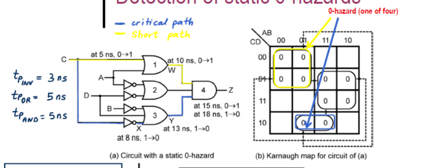


Basically because of gate delays for a moment when B changes is not true that $B+\bar{B}=1$

Temporary violation of complementary law.

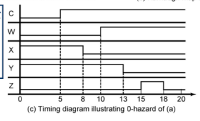
11

Detection of static 0-hazards



Basically because of gate delays for a moment when C changes is not true that $C \cdot \bar{C} = 0$

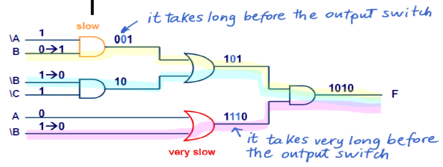
Temporary violation of complementary law



$F = (A+C) \cdot (\bar{A}+D) \cdot (\bar{B}+C+D)$

13

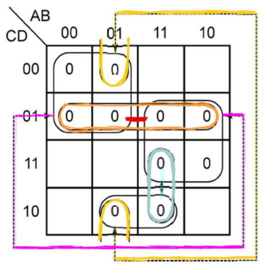
Dynamic hazard example


slow
it takes long before the output switch
\A 1
B 0→1
001
\B 1→0
\C 1
10
101
1010
F
A 0
\B 1→0
1110
very slow
it takes very long before the output switch

All gates are ideal (very fast) except a slow AND gate and a very slow OR gate

17

Removing the static hazard



The circuit has 4 potential sources of hazards that must be removed

$F = (A+C) \cdot (\bar{A}+\bar{D}) \cdot$
$\cdot (\bar{B}+\bar{C}+D) \cdot (C+\bar{D}) \cdot$
$\cdot (A+\bar{B}+D) \cdot$
$\cdot (\bar{A}+\bar{B}+\bar{C})$

14

Complex Programmable Logic Device

Field-Programmable Device (FPD)

- refers to any type of integrated circuit used for implementing digital hardware, where the chip can be configured by the end user to realize different designs.

Programmable Logic Devices (PLDs)

Programmable Logic Array (PLA)

- A relatively small FPD
- Contain two levels of logic
    - an AND-plane and an OR-plane, both planes are programmable

Programmable Array Logic (PAL)

- PLAs were introduced in the early 1970s by
- Philips, their main drawbacks were :
    - Expensive to manufacture
    - Somewhat poor speed-performance.
    - Due to the two levels of configurable logic
- To overcome these weaknesses, Programmable Array Logic (PAL) devices were developed.
- Relatively small FPD that has a programmable AND-plane followed by a fixed OR-plane

Complex Programmable Logic Devices (CPLDs)

- More Complex PLD that consists of an arrangement of multiple SPLD-like blocks on a single chip.

Antifuses

- Originally open-circuits and take on low resistance only when programmed.

Functionality of the following programmable components available on Xilinx FPGA:

- CLBs: configurable logic blocks which implement gates and flipflops
- IOBs: input/output blocks which provide off-chip connectivity
- BlockRAM: onchip static RAM storage
- Multiplier: hardwired multiplier cell
- DCM: digital clock manager for clock generation and distribution

What is the functionality of a dedicated recursive 'fish-bone' network used inside Xilinx FPGA?

- It is used to ensure that clock arrives every D Flip-flop at the same time.

Spartan 3

5 programmable elements in regular networks

1. CLB - (configurable logic block) implements gates and flip flops
    - 4 'Slices' per CLB, can work as logic (gates + flip-flop), distributed RAM, or shift reg
2. Input/ Output blocks - provide off-chip connectivity
3. Block ram - on-chip 18 kBit static RAM storage
4. Multiplier - Hardwired 18 x 18 multiplier cell
5. DCM - digital clock manager for clk generation and distribution

A 'gate-level netlist' is mapped on an FPGA by configuring these elements

- Choose CLB configuration (LUT, flip-flop, carry logic, ..)
- Choose interconnections in network

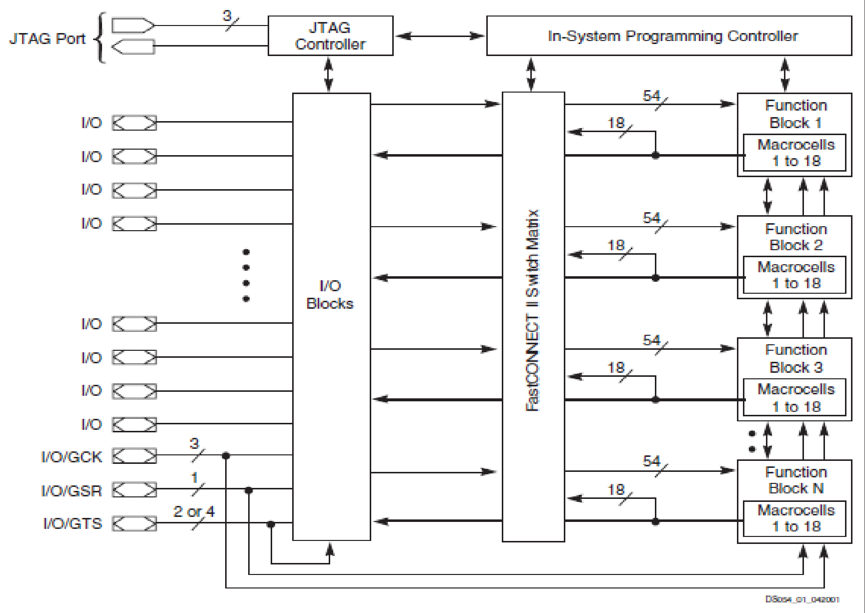


Figure 1: **XC9500XL Architecture**
Note: Function block outputs (indicated by the bold lines) drive the I/O blocks directly.

Each Function Block, as shown in Figure 2 is comprised of 18 independent macrocells, each capable of implementing a combinatorial or registered function. The FB also receives global clock, output enable, and set/reset signals. The FB generates 18 outputs that drive the FastCONNECT switch matrix. These 18 outputs and their corresponding output enable signals also drive the IOB. Logic within the FB is implemented using a sum-of-products representation. Fifty-four inputs provide 108 true and complement signals into the programmable AND-array to form 90 product terms. Any number of these product terms, up to the 90 available, can be allocated to each macrocell by the product term allocator.

Each XC9500XL macrocell may be individually configured for a combinatorial or registered function. The macrocell and associated FB logic is shown in Figure 3. Five direct product terms from the AND-array are available for use as primary data inputs (to the OR and XOR gates) to implement combinatorial functions, or as control inputs including clock, clock enable, set/reset, and output enable. The product term allocator associated with each macrocell selects how the five direct terms are used. The macrocell register can be configured as a D-type or T-type flip-flop, or it may be bypassed for combinatorial operation. Each register supports both asynchronous set and reset operations. During power-up, all user registers are initialized to the user-defined preload state (default to 0 if unspecified).

```
//3-bit ripple carry adder hierachical design
module fa(a,b,c,cout,s);
input a,b,c;
output cout,s;
assign s=a^b^c;
assign cout=((a^b)&cin)|(a&b);
endmoudle
module rca3(a,b,c,cout,s);
input [2:0] a,b;
input c;
output [2:0] s
output cout;
wire [1:0] m;
fa g1(.a(a[0]), .b(b[0]), .c(c), .cout(m[0]), .s(s[0]));
fa g2(a(a[1]), .b(b[1]), .c(m[0]), .cout(m[1]), .s(s[1]));
fa g3(a[2], b[2], m[1], cout, s[2]);
endmodule
```

```
module rca3_tb;
reg [2:0] a,b;
reg c;
wire [2:0] s;
wire cout;
rca3 uut(a,b,c,cout,s);
initial
begin
 a=0; b=0; c=0;
 #10 a=4; b=5;
 #20 a=6; b=2; c=1;
 # 20 $stop;
end
endmodule
```

AND
a&b;
a·b

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

OR
a|b;
a+b

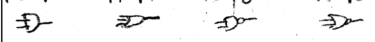
a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

NAND
a~&b;
$\overline{a.b}$

a	b	y
0	0	1
0	1	1
1	0	1
1	1	0

NOR
a~|b;
$\overline{a+b}$

a	b	y
0	0	1
0	1	0
1	0	0
1	1	0



XOR
a^b;
$a \oplus b$

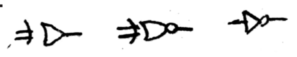
a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

XNOR
a~^b,
$a \odot b$

a	b	y
0	0	1
0	1	0
1	0	0
1	1	1

NOT
~a
$\overline{a}$

a	y
0	1
1	0



JTAG Boundary Scan

- 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

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

Boundary Scan Instructions

SAMPLE/PRELOAD
- This instruction causes the TDI and TDO to be connected to the Boundary Scan Register (BSR).
- However, the device is left in its normal functional mode. During this instruction, the BSR can be accessed by a data scan operation to take a sample of the functional data entering and leaving the device.
- The instruction is also used to preload test data into the BSR prior to loading an EXTEST instruction.

Purpose:
- 1. Get snapshot of normal chip output signals
- 2. Put data on boundary scan chain before next instruction

EXTEST
- This instruction causes the TDI and TDO to be
- connected to the Boundary Scan Register (BSR). The device's pin states are sampled with the 'capture DR' JTAG state and new values are shifted into the BSR with the 'shift DR' state; these values are then applied to the pins of the device using the 'update DR' state.
- DR: data register

EXTEST Instruction
- Purpose: Test off-chip circuits and board-level interconnections

INTEST
- This instruction causes the TDI and TDO lines to be connected to the Boundary Scan Register (BSR). While the EXTEST instruction allows the user to set and read pin states, the INTEST instruction relates to the core-logic signals of a device.

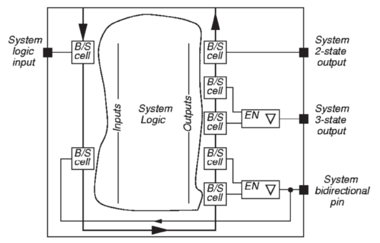
INTEST Instruction Purpose:
- 1. Shifts external test patterns onto component
- 2. External tester shifts component responses out

IDCODE
- Instruction Purpose: Connects the component device identification register serially between TDI and TDO
- In the Shift-DR TAP controller state
- Allows board-level test controller or external tester to read out component ID
- Required whenever a JEDEC identification register is included in the design

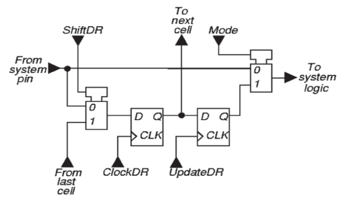
HIGHZ
- Instruction Purpose: Puts all component output pin signals into highimpedance state
- Control chip logic to avoid damage in this mode
- May have to reset component after HIGHZ runs
- Optional instruction

BYPASS Instruction
- Purpose: Bypasses scan chain with 1-bit register

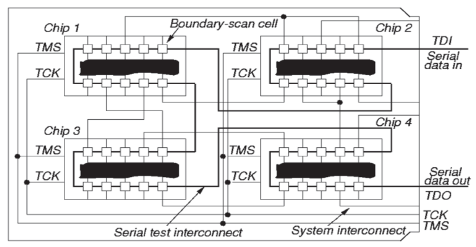
System View of Interconnect



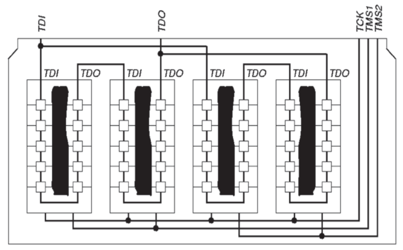
Elementary Boundary Scan Cell



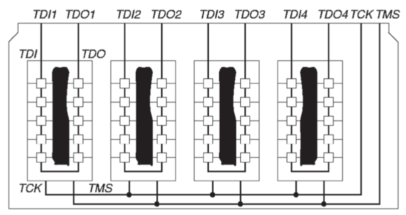
Serial Board / MCM Scan



Parallel Board / MCM Scan



Independent Path Board / MCM Scan

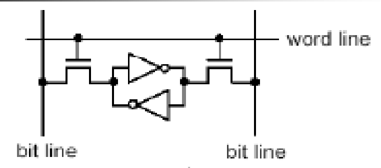


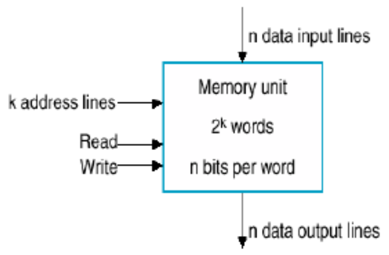
SRAM

Reading
- Steps
- Setup address lines
- Activate read line
- Data available after specified amount of time

Writing
- Sequence of steps
- Setup address lines
- Setup data lines
- Activate write line

SRAM Cell


word line
bit line
bit line


n data input lines
k address lines
Memory unit
2^k words
n bits per word
Read
Write
n data output lines