Lecture 2 – Combinational Logic Circuits

Reference: Roth/John Text: Chapter 2
Combinational logic

-- Behavior can be specified as concurrent signal assignments
-- These model concurrent operation of hardware elements

entity Gates is
  port (a, b, c: in  STD_LOGIC;
       d:       out STD_LOGIC);
end Gates;

architecture behavior of Gates is
  signal e: STD_LOGIC;
begin
  -- concurrent signal assignment statements
  e <= (a and b) xor (not c);  -- synthesize gate-level ckt
  d <= a nor b and (not e);    -- in target technology
end;
Example: SR latch (logic equations)

entity SRlatch is
  port (S,R: in std_logic; -- latch inputs
        Q,QB: out std_logic); -- latch outputs
end SRlatch;

architecture eqns of SRlatch is
  signal Qi,QBi: std_logic; -- internal signals
begin
  QBi <= S nor Qi;    -- Incorrect would be: QB <= S nor Q;
  Qi <= R nor QBi; -- Incorrect would be: Q <= R nor QB;
  Q <= Qi;       -- drive output Q with internal Qi
  QB <= QBi; -- drive output QB with internal QBi
end;
VHDL: Conditional signal assignment (form 1)

\[ z \leq m \text{ when } sel = '0' \text{ else } n; \]

\[ y \leq a \text{ when } (s="00") \text{ else } \\
\leq b \text{ when } (s="01") \text{ else } \\
\leq c \text{ when } (s="10") \text{ else } \\
\leq d; \]

Condition can be any Boolean expression

\[ y \leq a \text{ when } (F='1') \text{ and } (G='0') \ldots \]
Conditional signal assignment (form 2)

-- One signal (S in this case) selects the result
signal a, b, c, d, y : std_logic;
signal s: std_logic_vector (0 to 1);
begin
    with s select
    y <=   a when "00",
          b when "01",
          c when "10",
          d when "11";

--Alternative “default” *:
    d when others;

* “std_logic” values can be other than ‘0’ and ‘1’
32-bit-wide 4-to-1 multiplexer

signal a, b, c, d, y: std_logic_vector(0 to 31);
signal s: std_logic_vector(0 to 1);
begin

    with s select
        y <= a when "00",
            b when "01",
            c when "10",
            d when "11";

--y, a, b, c, d can be any type, if they match
32-bit-wide 4-to-1 multiplexer

-- Delays can be specified if desired

signal a, b, c, d, y: std_logic_vector (0 to 31);
signal s: std_logic_vector (0 to 1);

begin

with s select

    y <=
        a after 1 ns when "00",
        b after 2 ns when "01",
        c after 1 ns when "10",
        d when "11";

a->y delay is 1ns, b->y delay is 2ns, c->y delay is 1ns, d->y delay is δ
Verilog: 4-to-1 multiplexer

module mux (a, b, c, d, s, y);
    input a, b, c, d;
    input [1:0] s;
    output reg y;

always @(a or b or c or d or s)
begin
    case (s)
        2'b00   : y = a;
        2'b01   : y = b;
        2'b10   : y = c;
        default : y = d;
    endcase
end
endmodule
MUX using if-else statement

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);
        y : out std_logic);
end mux;

architecture arch_mux of mux is
begin
    process (a, b, c, d, s)
    begin
        if (s = "00") then
            y <= a;
        elsif (s = "01") then
            y <= b;
        elsif (s = "10") then
            y <= c;
        else
            y <= d;
        end if;
    end process;
end arch_mux;

module mux (a, b, c, d, s, y);
    input a, b, c, d;
    input [1:0] s;
    output reg y;
always @(a or b or c or d or s)
begin
    if (s == 2'b00)
        y = a;
    else if (s == 2'b01)
        y = b;
    else if (s == 2'b10)
        y = c;
    else
        y = d;
    end
endmodule
Conditional assignment can model the truth table of a switching function (without deriving logic equations)

```vhdl
signal S: std_logic_vector(1 downto 0);
begin
  S <= A & B;    -- S(1)=A, S(0)=B
  with S select
    Y <= '0' when "00",
        '1' when "01",
        '1' when "10",
        '0' when "11",
        'X' when others;
end;
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

& is the concatenate operator, merging scalars/vectors into larger vectors.
Example: full adder truth table

ADDin <= A & B & Cin;  --ADDin is a 3-bit vector
S    <= ADDout(0);    --Sum output (ADDout is a 2-bit vector)
Cout <= ADDout(1);    --Carry output

with ADDin select
   ADDout <= "00" when "000",
            "01" when "001",
            "01" when "010",
            "10" when "011",
            "01" when "100",
            "10" when "101",
            "10" when "110",
            "11" when "111",
            "XX" when others;

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Cout</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
library ieee; use ieee.std_logic_1164.all;

entity decode2_4 is
  port (A,B,EN: in std_logic;
        Y: out std_logic_vector(3 downto 0));
end decode2_4;

architecture behavior of decode2_4 is
  signal D: std_logic_vector(2 downto 0);
begin
  D <= EN & B & A; -- vector of the three inputs
  with D select
  Y <= "0001" when "100", --enabled, BA=00
       "0010" when "101", --enabled, BA=01
       "0100" when "110", --enabled, BA=10
       "1000" when "111", --enabled, BA=11
       "0000" when others; --disabled (EN = 0)
end;
Verilog: 3-to-8 Decoder

module decoder (Data, Code);
output [7: 0] Data;
input [2: 0] Code;
reg [7: 0] Data;

always @ (Code)
begin
  if (Code == 0) Data = 8'b00000001;
  else if (Code == 1) Data = 8'b00000010;
  else if (Code == 2) Data = 8'b00000100;
  else if (Code == 3) Data = 8'b00001000;
  else if (Code == 4) Data = 8'b00010000;
  else if (Code == 5) Data = 8'b00100000;
  else if (Code == 6) Data = 8'b01000000;
  else if (Code == 7) Data = 8'b10000000;
  else Data = 8'bx;
end
endmodule

/* Alternative description
always @ (Code)
case (Code)
  0 : Data = 8'b00000001;
  1 : Data = 8'b00000010;
  2 : Data = 8'b00000100;
  3 : Data = 8'b00001000;
  4 : Data = 8'b00010000;
  5 : Data = 8'b00100000;
  6 : Data = 8'b01000000;
  7 : Data = 8'b10000000;
  default: Data = 8'bx;
endcase */
VHDL: Structural model (no “behavior” specified)

architecture structure of full_add1 is
  component xor -- declare component to be used
      port (x,y: in std_logic;
           z: out std_logic);
  end component;
for all: xor use entity work.xor(eqns); -- if multiple arch’s in lib.
signal x1: std_logic;-- signal internal to this component
begin -- instantiate components with “map” of connections
  G1: xor port map (a, b, x1); -- instantiate 1st xor gate
  G2: xor port map (x1, cin, sum); -- instantiate 2nd xor gate
  …add circuit for carry output…
end;
Associating signals with formal ports

component AndGate
    port (Ain_1, Ain_2 : in std_logic; -- formal parameters
         Aout : out std_logic);
end component;

begin
    -- positional association of “actual” to “formal”
    A1:AndGate port map (X, Y, Z1);

    -- named association (usually improves readability)
    A2:AndGate port map (Ain_2=>Y, Aout=>Z2, Ain_1=>X);

    -- both (positional must begin from leftmost formal)
    A3:AndGate port map (X, Aout => Z3, Ain_2 => Y);
module Add_half_0_delay (sum, c_out, a, b);
    input a, b;
    output c_out, sum;
    xor (sum, a, b);
    and (c_out, a, b);
endmodule

module Add_full_0_delay (sum, c_out, a, b, c_in);
    input a, b, c_in;
    output c_out, sum;
    wire w1, w2, w3;
    Add_half_0_delay M1 (w1, w2, a, b);
    Add_half_0_delay M2 (sum, w3, c_in, w1);
    or (c_out, w2, w3);
endmodule
Verilog: Port Connection by Name

.formal_name(actual_name)

- Connect ports by name in modules that have several ports
- Regardless the position of this entry in the port list

Add_half_0_delay M1 ( .b (b),
    .c_out (w2),
    .a (a),
    .sum (w1) );
Design Hierarchy: 16-bit Ripple Carry Adder

Add_rca_16_0_delay

Add_rca_4_0_delay Add_rca_4_0_delay Add_rca_4_0_delay Add_rca_4_0_delay

Add_half_0_delay Add_half_0_delay

Add_full_0_delay Add_half_0_delay
### Verilog for 16-bit RCA

**module** Add_rca_4_0_delay (sum, c_out, a, b, c_in);

**output** [3: 0] sum;
**output** c_out;
**input** [3: 0] a, b;
**input** c_in;
**wire** c_in2, c_in3, c_in4;

Add_full_0_delay M1 (sum[0], c_in2, a[0], b[0], c_in);
Add_full_0_delay M2 (sum[1], c_in3, a[1], b[1], c_in2);
Add_full_0_delay M3 (sum[2], c_in4, a[2], b[2], c_in3);
Add_full_0_delay M4 (sum[3], c_out, a[3], b[3], c_in4);

endmodule

**module** Add_rca_16_0_delay (sum, c_out, a, b, c_in);

**output** [15:0] sum;
**output** c_out;
**input** [15:0] a, b;
**input** c_in;
**wire** c_in4, c_in8, c_in12, c_out;

Add_rca_4_0_delay M1 (.sum(sum[3:0]), .c_out(c_in4), .a(a[3:0]), .b(b[3:0]), .c_in(c_in));
Add_rca_4_0_delay M2 (sum[7:4], c_in8, a[7:4], b[7:4], c_in4);
Add_rca_4_0_delay M3 (sum[11:8], c_in12, a[11:8], b[11:8], c_in8);
Add_rca_4_0_delay M4 (sum[15:12], c_out, a[15:12], b[15:12], c_in12);

endmodule
Structural Model: 2-bit comparator

- Note the flexibility of n-input primitives
  - The same name but different numbers of inputs
- Compare two 2-bit binary words:
  \[
  A_{\text{lt}}_B = A_1' B_1 + A_1' A_0' B_0 + A_0' B_1 B_0 \\
  A_{\text{gt}}_B = A_1 B_1' + A_0 B_1' B_0' + A_1 A_0 B_0' \\
  A_{\text{eq}}_B = A_1' A_0' B_1' B_0' + A_1' A_0 B_1' B_0 \\
    + A_1 A_0 B_1 B_0 + A_1 A_0' B_1 B_0'
  \]

- Classical approach
  - use K-maps to reduce the logic and produce the schematic
- HDL approach using structural model
  - Connect primitives to describe the functionality implied by the schematic
  - Synthesis tool will automatically optimize the gate level design
- HDL approach using behavioral model
Structural Model: 2-bit comparator (Cont.)

- Schematic after minimization of K-maps
Verilog (Structural) Model:

```verilog
module compare_2_str (A_gt_B, A_lt_B, A_eq_B, A0, A1, B0, B1);
  output   A_gt_B, A_lt_B, A_eq_B;
  input    A0, A1, B0, B1;
  // Note: w1, w2, ... are implicit wires
  nor      (A_gt_B, A_lt_B, A_eq_B);
  or       (A_lt_B, w1, w2, w3);
  and      (A_eq_B, w4, w5);
  and      (w1, w6, B1);       // 2-input AND
  and      (w2, w6, w7, B0);  // 3-input AND
  and      (w3, w7, B0, B1);  // Note: interchanging w7, B0 and B1 has no effect
  not      (w6, A1);
  not      (w7, A0);
  xnor     (w4, A1, B1);
  xnor     (w5, A0, B0);
endmodule
```
4-bit Comparator

- Using design hierarchy
  - 4-bit CMP constructed by 2-bit CMP
  - A strict inequality in the higher-order bit-pair determines the relative magnitudes of the 4-bit words
  - If higher-order bit-pair are equal, then the lower-order bit-pair determine the output of the 4-bit CMP
Verilog Model:

```verilog
module Comp_4_str (A_gt_B, A_lt_B, A_eq_B, A3, A2, A1, A0, B3, B2, B1, B0);
  output A_gt_B, A_lt_B, A_eq_B;
  input  A3, A2, A1, A0, B3, B2, B1, B0;
  wire   w1, w0;

  Comp_2_str M1 (A_gt_B_M1, A_lt_B_M1, A_eq_B_M1, A3, A2, B3, B2);
  Comp_2_str M0 (A_gt_B_M0, A_lt_B_M0, A_eq_B_M0, A1, A0, B1, B0);

  or   (A_gt_B, A_gt_B_M1, w1);
  and  (w1, A_eq_B_M1, A_gt_B_M0);
  and  (A_eq_B, A_eq_B_M1, A_eq_B_M0);
  or   (A_lt_B, A_lt_B_M1, w0);
  and  (w0, A_eq_B_M1, A_lt_B_M0);
endmodule
```
module encoder (Code, Data);
  output [2: 0] Code;
  input [7: 0] Data;
  reg [2: 0] Code;

  always @(Data)
  begin
    if (Data == 8'b00000001) Code = 0;
    else if (Data == 8'b00000010) Code = 1;
    else if (Data == 8'b00000100) Code = 2;
    else if (Data == 8'b00001000) Code = 3;
    else if (Data == 8'b00010000) Code = 4;
    else if (Data == 8'b00100000) Code = 5;
    else if (Data == 8'b01000000) Code = 6;
    else if (Data == 8'b10000000) Code = 7;
    else Code = 3'bx;
  end
endmodule
/* Alternative description is given below

always @ (Data)
    case (Data)
        8'b00000001 : Code = 0;
        8'b00000010 : Code = 1;
        8'b00000100 : Code = 2;
        8'b00001000 : Code = 3;
        8'b00010000 : Code = 4;
        8'b00100000 : Code = 5;
        8'b01000000 : Code = 6;
        8'b10000000 : Code = 7;
        default     : Code = 3'bx;
    endcase
*/
Priority Encoder

- If input code has multiple bits asserted for an encoder
  - Then priority rule is required to form an output bit pattern
  - Called priority encoder

Example:

<table>
<thead>
<tr>
<th>Data_In [3:0]</th>
<th>Data_out [1:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>- - - 1</td>
<td>00</td>
</tr>
<tr>
<td>- - 1 -</td>
<td>01</td>
</tr>
<tr>
<td>- 1 - -</td>
<td>10</td>
</tr>
<tr>
<td>1 - - -</td>
<td>11</td>
</tr>
</tbody>
</table>

Note: "-" denotes a don't care condition.
module priority (Code, valid_data, Data);
    output [2: 0] Code;
    output valid_data;
    input [7: 0] Data;
    reg [2: 0] Code;

    assign valid_data = |Data;     // "reduction or" operator

always @ (Data)
    begin
        if (Data[7]) Code = 7;
        else if (Data[6]) Code = 6;
        else if (Data[5]) Code = 5;
        else if (Data[4]) Code = 4;
        else if (Data[3]) Code = 3;
        else if (Data[2]) Code = 2;
        else if (Data[1]) Code = 1;
        else if (Data[0]) Code = 0;
        else Code = 3'bx;
    end
endmodule
### Boolean Algebra - Review

<table>
<thead>
<tr>
<th>Theorem</th>
<th>SOP Form</th>
<th>POS Form</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logical Adjacency</strong></td>
<td>$ab + ab' = a$</td>
<td>$(a + b)(a + b') = a$</td>
</tr>
<tr>
<td><strong>Absorption</strong></td>
<td>$a + ab = a$</td>
<td>$a(a + b) = a$</td>
</tr>
<tr>
<td>or:</td>
<td>$ab' + b = a + b$</td>
<td>$(a + b')b = ab$</td>
</tr>
<tr>
<td></td>
<td>$a + a'b = a + b$</td>
<td>$(a' + b)a = ab$</td>
</tr>
<tr>
<td><strong>Multiplication and</strong></td>
<td>$(a + b)(a' + c) = ac + a'b$</td>
<td>$ab + a'c = (a + c)(a' + b)$</td>
</tr>
<tr>
<td><strong>Factoring</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Consensus</strong></td>
<td>$ab + bc + a'c = ab + a'c$</td>
<td>$(a + b)(b + c)(a' + c) = (a + b)(a' + c)$</td>
</tr>
</tbody>
</table>

#### Diagrams

- **Logical Adjacency**
  - $ab'$ and $ab$ overlap,
  - Covered by $ab$ and $a'b$.

- **Consensus**
  - $ab$ and $bc$,
  - $bc$ is redundant,
  - Covered by $a'$ and $c$.

The consensus term, $bc$, is redundant.
• \( ab + bc + a'c = ab + a'c \)

Proof: 
\[
ab + bc + a'c = ab+(a+a')bc+a'c \\
= ab+abc+a'bc+a'c \\
= ab(1+c) + a'c(b+1) \\
= ab + a'c
\]
## Boolean Algebra - Review

### Exclusive-Or Laws

| Combinations with 0, 1 | a \oplus 0 = a  
|-----------------------|-------------------------  
| a \oplus 1 = a'        | a \oplus a = 0  
| a \oplus a' = 1       | a \oplus b = b \oplus a  
|                        | (a \oplus b) \oplus c = a \oplus (b \oplus c) = a \oplus b \oplus c  
| Commutative            |                        | a (b \oplus c) = ab \oplus ac  
| Associative            |                        | (a \oplus b)' = a \oplus b' = a' \oplus b = ab + a'b'  
| Distributive           |                        |                      


An implicant which does not imply any other implicant of the function is called a prime implicant.

- A prime implicant is a cube that is not properly contained in some other cube of the function.
- A prime implicant cannot be combined with another implicant to eliminate a literal or to be eliminated from the expression by absorption.
- An implicant that implies another implicant is said to be "covered" by it; the set of its vertices is a subset of the vertices of the implicant that covers it. The covering implicant, having fewer literals, has more vertices.
Prime Implicant Example - Review

$f(a, b, c) = ab'c' + abc' + abc + a'b'c$

$f(a, b, c) = ac' + ab + a'b'c$

Implicants

Prime Implicants
**Essential Prime Implicant** - Review

- **Essential prime implicant**: A prime implicant that is not covered by any set of other implicants is an essential prime implicant.

- **Example**: \( f(a, b, c) = a'bc + abc + ab'c' + abc' \)
  - Prime Implicants: \( \{ac', ab, bc\} \)
  - Essential Prime Implicants: \( \{ac', bc\} \)
  - SOP Expression: \( f(a, b, c) = ac' + ab + bc \)
  - Minimal SOP Expression: \( f(a, b, c) = ac' + bc \)
Karnaugh Maps (SOP Form)

- Karnaugh maps reveal logical adjacencies and opportunities for eliminating a literal from two or more cubes.
- K-maps facilitate finding the largest possible cubes that cover all 1s without redundancy.
- Requires manual effort.
Don't-Care - Review

- Don't cares represent situations where an input cannot occur, or the output does not matter.
  - Use (i.e. cover) don't cares when they lead to an improved representation.
Example: K-Map and DC - Review

- Suppose a function is asserted when the BCD representation of a 4-variable input is 0, 3, 6 or 9.
  - $f(a, b, c, d) = a'b'c'd' + a'b'cd + abc'd' + abc'd + abcd + abcd' + ab'c'd + ab'cd$ has 32 literals.
  - Without don't-cares (16 literals):
    - $f(a, b, c, d) = a'b'c'd' + a'b'cd + a'bcd' + ab'c'd$
  - With don't cares (12 literals):
    - $f(a, b, c, d) = a'b'c'd' + b'cd + bcd' + ad$

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Glitches and Static Hazards

- The output of a combinational circuit may make a transition even though the patterns applied at its inputs do not imply a change. These unwanted switching transients are called "glitches."

- Glitches are a consequence of the circuit structure and the application of patterns that cause the glitch to occur. A circuit in which a glitch may occur under the application of appropriate inputs signals is said to have a hazard.
A **static 1-hazard** occurs if an output has an initial value of 1, and an input pattern that does not imply an output transition causes the output to change to 0 and then return to 1.

A **static 0-hazard** occurs if an output has an initial value of 0, and an input pattern that does not imply an output transition causes the output to change to 1 and then return to 0.
Cause and Elimination of Static Hazard

- Static hazards are caused by differential propagation delays on reconvergent fanout paths.
- A "minimal" realization of a circuit might not be hazard-free.
- Static hazards can be eliminated by introducing redundant cubes in the cover of the output expression (the added cubes are called a hazard cover).
Consider \( F = AC + BC' \)

- Initial inputs: \( A = 1, \ B = 1, \ C = 1 \) and \( F = 1 \)
- New inputs: \( A = 1, \ B = 1, \ C = 0 \) and \( F = 1 \)

In a physical realization of the circuit (i.e. non-zero propagation delays), the path to \( F1 \) will be longer than the path to \( F0 \), causing a change in \( C \) to reach \( F1 \) later than it reaches \( F0 \).

Consequently, when \( C \) changes from 1 to 0, the output undergoes a momentary transition to 0 and returns to 1.
Example: Elimination of Static-1 Hazard (cont.)

- The presence of a static hazard is apparent in the Karnaugh map of the output.
  - AC de-asserts before BC' asserts
    - In this example, the hazard occurs because the cube AC is initially asserted, while BC' is not. The switched input causes AC to de-assert before BC' can assert.

- **Hazard Removal**: A hazard can be removed by covering the adjacent prime implicants by a redundant cube (AB, a 'hazard cover") to eliminate the dependency on C (the boundary between the cubes is now covered).
Example: Elimination of Static-1 Hazard (cont.)

- Hazard covers require extra hardware.
- Example: For the hazard-free cover:
  
  \[ F = AC + BC' + AB \]
Static-0 Hazard Elimination

- To eliminate a static 0-hazard:
  - Method #1: Cover the adjacent 0s in the corresponding POS expression.
  - Method #2:
    - First eliminate the static 1-hazards.
    - Then form complement function and consider whether the implicants of the 0s of the expression, that is free of static 1-hazards, also cover all adjacent 0s of the original function. If they do not, then a static 0-hazard exists.
    - Adds redundant prime implicant to the complement of the static 1 hazard-free expression in POS form as needed.
Dynamic Hazards (Multiple glitches)

- A circuit has a dynamic hazard if an input transition is supposed to cause a single transition in an output, but causes two or more transitions before reached its expected value.
- Dynamic hazards are a consequence of multiple static hazards caused by multiply reconvergent paths in a multilevel circuit.
Dynamic Hazards Elimination

- Dynamic hazards are not easy to eliminate.
- Elimination of all static hazards eliminates dynamic hazards.

Approach:
- Transform a multilevel circuit into a two-level circuit, and
- Eliminate all of the static hazards.
Summary for Hazard

- Hazard might not be significant in a synchronous sequential circuit if the clock period can be extended
  - Implies a slower design
- Static hazard can be eliminated by introducing redundant cubes in 2-level logic (SOP form)
  - Hazard cover
- Elimination of static hazard in a multiple-level logic
  - Flatten the design to 2-level first
- Dynamic hazard elimination
  - Transfer the design into a 2-level form
  - Detect and eliminate all static hazards
Latch v.s. Flip-Flop

- **Latch**: level sensitive (the enable signal is the clock)
  - The output of a transparent latch changes in response to the data input while the latch is enabled. Changes at the input are visible at the output.

- **Flip-flop**: edge sensitive
  - The value of data stored depends on the data that is present at the data input(s) when the clock makes a transition at its active (rising or falling) edge.

![Latch and Flip-Flop Diagrams](image-url)
Latch v.s. Flip-Flop (cont.)

- Latch often called transparent latch
  - Changes at the input are visible at the output while it is enabled
- Flip-flop consists of two data latches in a master-slave configuration
  - Samples the input during the active cycle of the clock applied to the master stage. The input is propagated to the output during the slave cycle of the clock.
- Master-slave implementation of negative edge-triggered D-FF:
  - The output of the master stage must settle before the enabling edge of the slave stage.
  - The master stage is enabled on the inactive edge of the clock, and the slave stage is enabled on the active edge.
Example: D flip-flop Primitives

FDCE: D Flip-Flop with Clock Enable and Asynchronous Clear

<table>
<thead>
<tr>
<th>CLR</th>
<th>CE</th>
<th>D</th>
<th>C</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>No Change</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>D</td>
<td>↑</td>
<td>D</td>
</tr>
</tbody>
</table>

https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/7series_hdl.pdf
Verilog Instantiation Template

// FDCE: Single Data Rate D Flip-Flop with Asynchronous Clear and Clock Enable (posedge clk).

// 7 Series


FDCE #(
  .INIT(1'b0) // Initial value of register (1'b0 or 1'b1)
) FDCE_inst (
  .Q(Q), // 1-bit Data output
  .C(C), // 1-bit Clock input
  .CE(CE), // 1-bit Clock enable input
  .CLR(CLK), // 1-bit Asynchronous clear input
  .D(D) // 1-bit Data input
);

// End of FDCE_inst instantiation
Library UNISIM;
use UNISIM.vcomponents.all;

-- FDCE: Single Data Rate D Flip-Flop with Asynchronous Clear and Clock Enable (posedge clk).
-- 7 Series
-- Xilinx HDL Libraries Guide, version 14.1

FDCE_inst : FDCE generic map (
INIT => '0') -- Initial value of register ('0' or '1') port map (
Q => Q, -- Data output
C => C, -- Clock input
CE => CE, -- Clock enable input
CLR => CLR, -- Asynchronous clear input D => D -- Data input );
-- End of FDCE_inst instantiation
4-bit Register (Structural Model)
VHDL Code for Register

```vhdl
library ieee;
Library UNISIM;
use UNISIM.vcomponents.all;
use ieee.std_logic_1164.all;

entity Register4 is
    Port ( D: in STD_LOGIC_VECTOR (3 downto 0);
        Clk, CLR, CE: in STD_LOGIC;
        Q : out STD_LOGIC_VECTOR (3 downto 0));
end Register4;

architecture structure of Register4 is

    begin
        F0: FDCE generic map (INIT=>'0') port map(Q=>Q(0), C=>Clk, CLR=>CLR, CE=>CE, D=>D(0));
        F1: FDCE generic map (INIT=>'0') port map(Q=>Q(1), C=>Clk, CLR=>CLR, CE=>CE, D=>D(1));
        F2: FDCE generic map (INIT=>'0') port map(Q=>Q(2), C=>Clk, CLR=>CLR, CE=>CE, D=>D(2));
        F3: FDCE generic map (INIT=>'0') port map(Q=>Q(3), C=>Clk, CLR=>CLR, CE=>CE, D=>D(3));
    end structure;
```

54
VHDL “Process” Construct

(Processes will be covered in more detail in “sequential circuit modeling”)

[label:] process (sensitivity list)
  declarations
  begin
    sequential statements
  end process;

- Process statements are executed in sequence
- Process statements are executed once at start of simulation
- Process halts at “end” until an event occurs on a signal in the “sensitivity list”
- Allows conventional programming language methods to describe circuit behavior
Modeling combinational logic as a process

-- All signals referenced in process must be in the sensitivity list.
entity And_Good is
  port (a, b: in std_logic; c: out std_logic);
end And_Good;

architecture Synthesis_Good of And_Good is
begin
  process (a,b)  -- gate sensitive to events on signals a and/or b
  begin
    c <= a and b;  -- c updated (after delay on a or b “events”
  end process;
  -- This process is equivalent to the simple signal assignment:
  -- c <= a and b;
end;
Bad example of combinational logic

-- This example produces unexpected results.
entity And_Bad is
  port (a, b: in std_logic; c: out std_logic);
end And_Bad;
architecture Synthesis_Bad of And_Bad is
begin
  process (a) -- sensitivity list should be (a, b)
  begin
    c <= a and b; -- will not react to changes in b
  end process;
end Synthesis_Bad;
-- synthesis may generate a flip flop, triggered by signal a
Verilog always@

always@(sensitivity list)
begin
…statements
end

- Non-blocking assignments (<=)
  - Non-blocking assignments happen parallelly.

- Blocking assignments (=)
  - Blocking assignments happen sequentially.