FPGA BASED SYSTEM DESIGN

Dr. Tayab Din Memon

tayabuddin.memon@faculty.muet.edu.pk

Lecture 9 & 10 : Combinational and Sequential Logic

Combinational vs Sequential Logic

- Combinational logic output depends upon the current input
  - Memory less system
- Sequential logic output needs memory because it depends upon the previous states
  - Storage elements connected in feedback loop with combinational logic

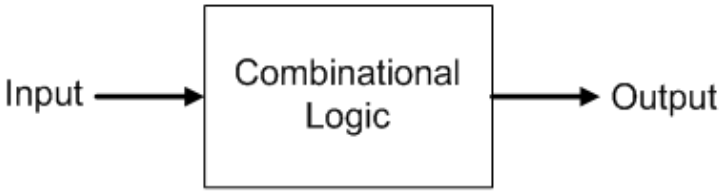


Figure (1)

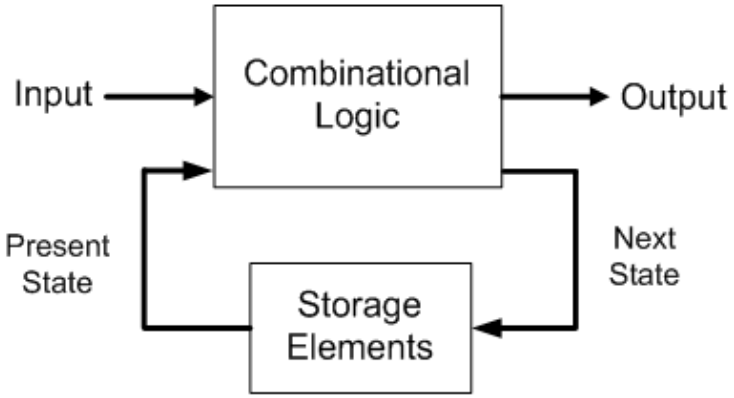


Figure (2)

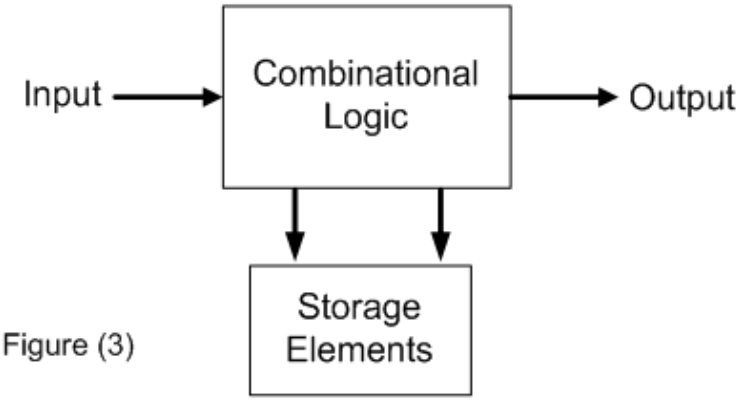
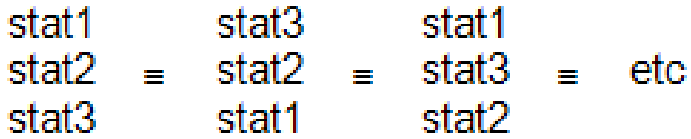


Figure (3)

Concurrent vs Sequential Code

- Only statements placed inside
  - the process
  - the procedure
  - or function are sequential
  - but VHDL code is inherently concurrent (parallel)
- In other words, concurrent are
  - Statements *outside of a process*
  - Processes are evaluated concurrently
- Concurrent code is also called *dataflow code*
- In general combinational logic circuits are build with concurrent code



Concurrent Statements

- Concurrent statements include:
  - Boolean equations
  - conditional assignments (when/else, with/select)
  - instantiation

Using Operators

Operator Type	Operators	Data Types
Logical	NOT, AND, NAND, OR, XOR, NOR, XNOR	BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR STD_ULOGIC, STD_ULOGIC_VECTOR
Arithmetic	+, -, *, /, ** MOD, REM, ABS	INTEGER, SIGNED, UNSIGNED
Comparison	=, /=, <, >, <=, >=	BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR STD_ULOGIC, STD_ULOGIC_VECTOR INTEGER, SIGNED, UNSIGNED
Shift	sll, srl, sla, sra, rol, ror	BIT_VECTOR
Concatenation	&, (,,,)	Same as for logical operators, plus signed and unsigned

- Easiest and basic way of creating concurrent code
- Complex circuits are easier to deal with sequential code comparatively, infact

Example – I: Multiplexer

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

entity MUX_Test is
port (s,t,u,v,w,x: in std_logic;
      y: out std_logic);
  end MUX_Test;

  architecture dataflow of MUX_Test is
  begin
    y<= (s and not x and not w) OR
        (t and not x and w) OR
        (u and x and not w) OR
        (v and x and w);
      end dataflow;
```

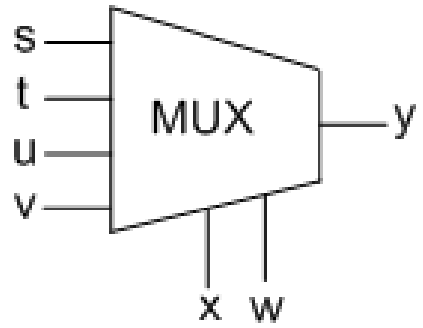


Fig: MUX Block

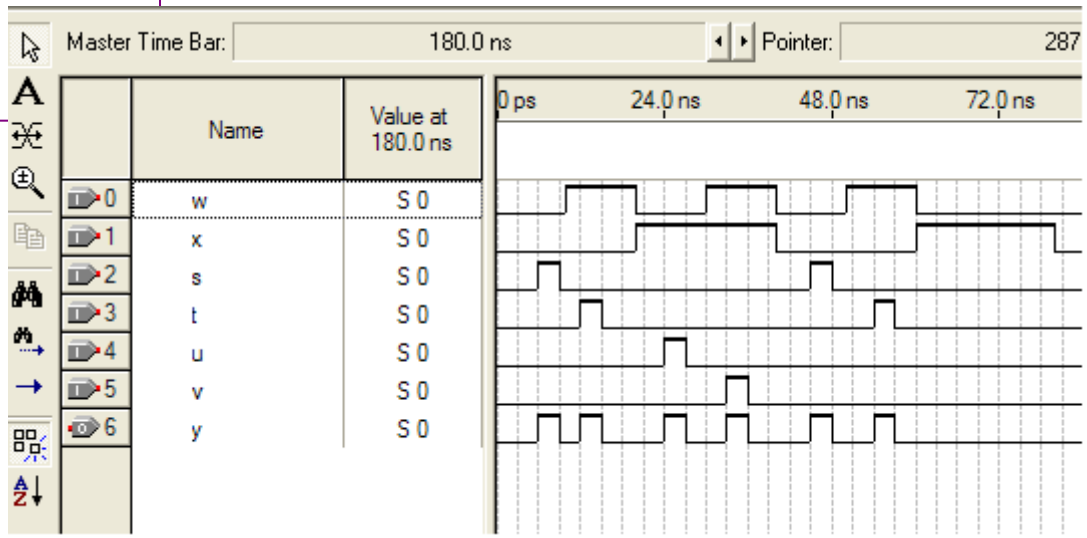


Fig: MUX

When Statement

When/ Else or with/ Select / When Syntax

```
--WHEN / ELSE STATEMENT

assignment WHEN condition ELSE
assignment WHEN condition ELSe
....;

-- WITH/ SELECT / WHEN STATEMENT

WITH identifier SELECT
assignment WHEN value,
assignment WHEN value,
....;
```

Example: When/ Else or with/ Select / When

```
-- WHEN/ ELSE

y <= "00" WHEN (a = '0' or re='1')
     else
     "01" WHEN  ena='1'
     else
     "10";

-- WTIH / SELECT / WHEN

WITH control SELECT
y <= "00" WHEN re
     "01" WHEN ena
     UNAFFECTED WHEN OTHERS;
```

```
WHEN value

WHEN value1 to value2

WHEN value1 | value2 | ...
```

Example – 2: Solution with WHEN/ELSE

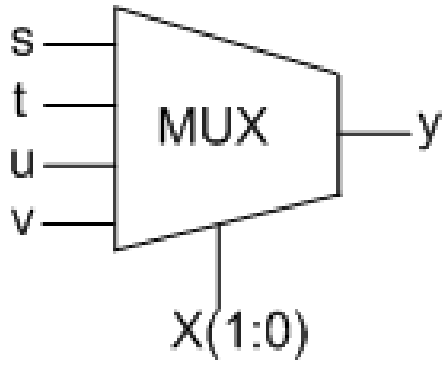


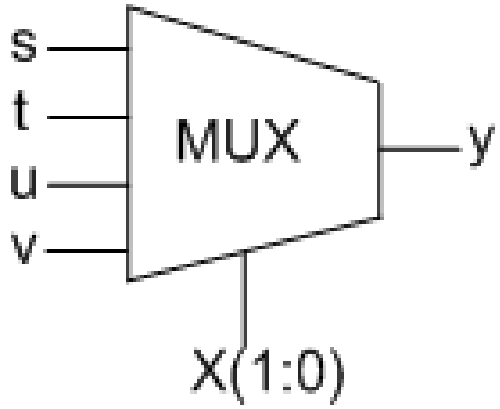
Fig: MUX Example - II

```
Architecture dataflow of MUXAgain is
begin
y <= s WHEN X=0 ELSE
     T WHEN X=1 ELSE
     U WHEN X=2 ELSE
     V;
END dataflow;
```

If x is an integer i.e., x : in integer range 0 to 3;

```
--Solution - 1: with WHEN/ELSE
library ieee;
use ieee.std_logic_1164.all;

Entity MUXAgain is
port (s,t,u,v: in std_logic;
      x : in std_logic_vector(1 downto 0);
      y : out std_logic);
end MUXAgain;

Architecture dataflow of MUXAgain is
begin
y <= s WHEN X="00" ELSE
     t WHEN X="01" ELSE
     u WHEN X="10" ELSE
     v;
   END dataflow;
```

Solution – 2: with WITH/SELECT/WHEN

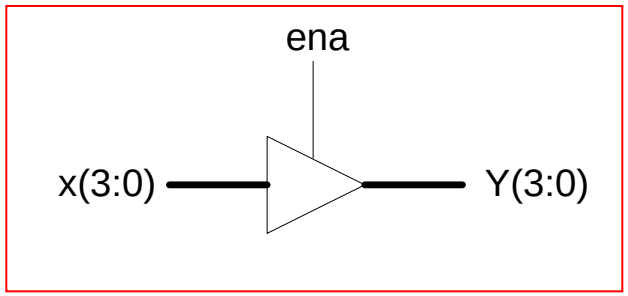



```
Architecture dataflow of MUXAgain is
begin
WITH X SELECT
y <= s WHEN 0,
     T WHEN 1,
     U WHEN 2,
     V WHEN 3;
END dataflow;
```

```
--Solution -2: with WITH/SELECT/WHEN

library ieee;
use ieee.std_logic_1164.all;

Entity MUXAgain is
port (s,t,u,v: in std_logic;
      x : in std_logic_vector(1 downto 0);
      y : out std_logic);
end MUXAgain;

Architecture dataflow of MUXAgain is
begin
WITH x SELECT
y <= s WHEN "00",
     t WHEN "01",
     u WHEN "10",
     v WHEN OTHERS;
END dataflow;
```

Example – 3: Tri-

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

Entity tri_state_use is
port (ena : in std_logic;
        x : in std_logic_vector (7 downto 0);
        y : out std_logic_vector (7 downto 0));
      end tri_state_use;
architecture dataflow of tri_state_use is
begin
y<= x when (ena ='1') else
    (others => 'Z');
  end dataflow;
```



A

Fig: Tri-State Buffer

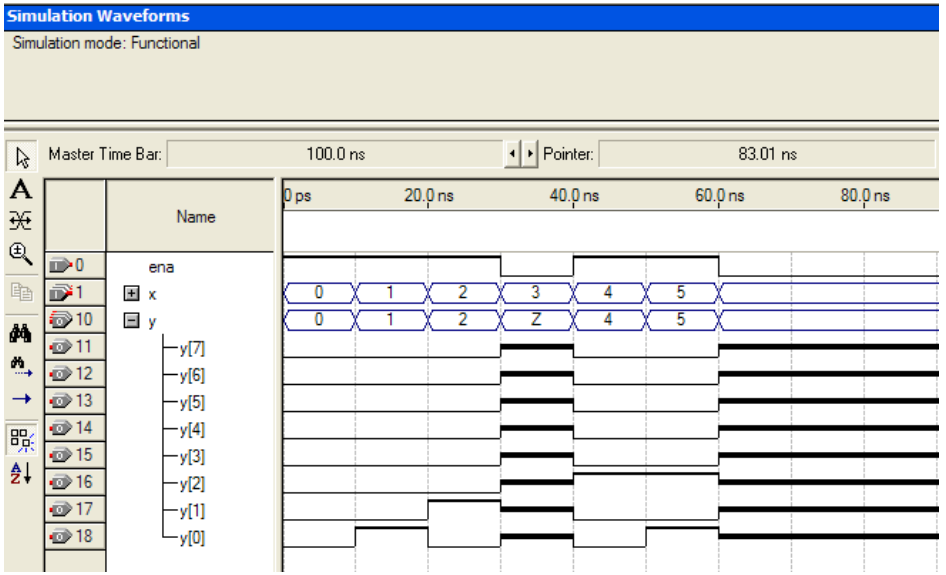


Fig: Vector Waveform

Encoder 8by3 (i.e., n=8, m=3) When-Else

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

Entity encoder8by3 is
port (x : in std_logic_vector (7 downto 0);
      y : out std_logic_vector (2 downto 0));
      end encoder8by3;

Architecture dataflow of encoder8by3 is
    begin
    y <= "000" WHEN X="00000001" ELSE
         "001" WHEN X="00000010" ELSE
         "010" WHEN X="00000100" ELSE
         "011" WHEN X="00001000" ELSE
         "100" WHEN X="00010000" ELSE
         "101" WHEN X="00100000" ELSE
         "110" WHEN X="01000000" ELSE
         "111" WHEN X="10000000" ELSE
    "ZZZ";
    END dataflow;
```

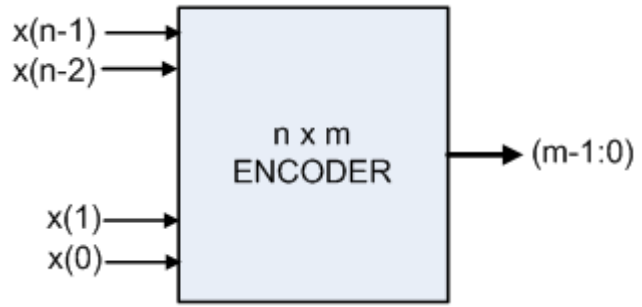


Fig: Encoder Block

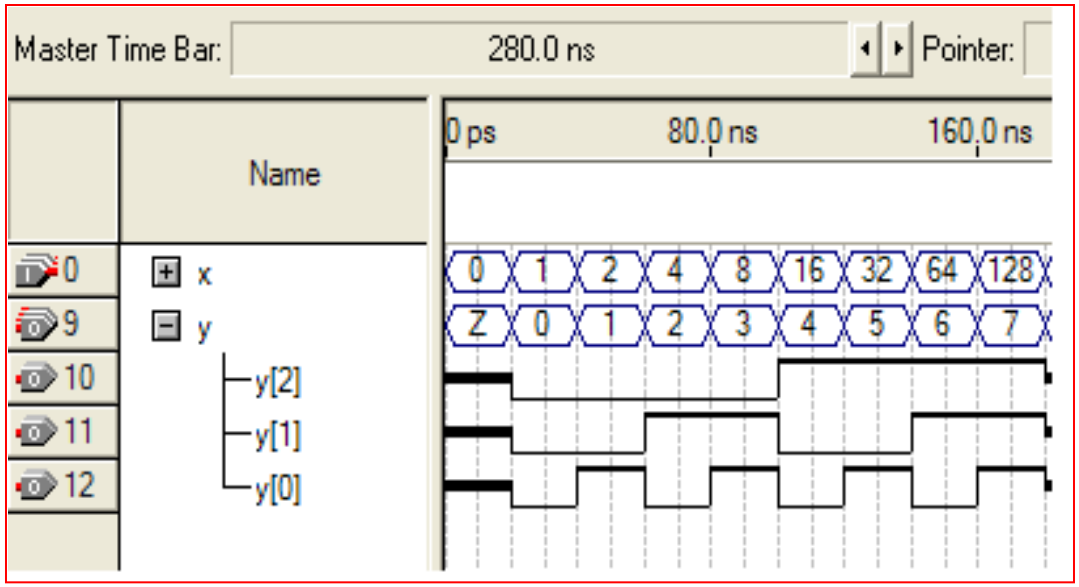


Fig: Simulation Results

Encoder 8by3 (i.e., n=8, m=3) With-Select-When

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

Entity encoder8by3_2 is
port (x : in std_logic_vector (7 downto 0);
      y : out std_logic_vector (2 downto 0));
      end encoder8by3_2;

Architecture dataflow of encoder8by3_2 is
    begin
    with x select
    y <= "000" WHEN "00000001",
         "001" WHEN "00000010",
         "010" WHEN "00000100",
         "011" WHEN "00001000",
         "100" WHEN "00010000",
         "101" WHEN "00100000",
         "110" WHEN "01000000",
         "111" WHEN "10000000",
    "ZZZ" WHEN OTHERS;
    END dataflow;
```

GENERATE Statement

- It is another concurrent statement. It allows a section of code to be repeated a number of times, thus creating several instances of the same assignment.

```
label: FOR identifier IN range GENERATE
(concurrent assignments)
ENG GENERATE;
```

- An irregular form of GENERATE statement is IF/GENERATE, syntax given below:

```
label: FOR identifier IN range GENERATE
  ......
  label2: IF condition GENERATE
    (concurrent assignments)
  end GENERATE;
  ....
end GENERATE;
```

How to use GENERATE Statement?

```
signal a: bit_vector (7 downto 0);
signal b: bit_vector (15 downto 0);
signal c: bit_vector (7 downto 0);

.....

D: FOR i in a'range generate
  c(i) <= a(i) AND b(i+8);
end generate;
```

```
g: FOR i IN 0 to input generate
   (concurrent statement)
end generate;
```

Example: input as variable

GENERATE Syntax – I

```
right: FOR i in 0 to 7 generate
  outcome(i) <= '1' when (a(i) and b(i)) ='1' else '0';
end generate;
```

Example: outcome as single driven

```
wrong: FOR i in 0 to 7 generate
  outcome <= "11111111" when (a(i) and b(i))='1' else "00000000";
end generate;
```

Example: outcome as multiple driven

GENERATE Shifter Example

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

entity GENERATE_as_Shifter is
port (input: in std_logic_vector (3 downto 0);
      sel: in integer range 0 to 4;
      output: out std_logic_vector (7 downto 0));
     end GENERATE_as_Shifter;
architecture dataflow of GENERATE_as_Shifter is

subtype vector is std_logic_vector (7 downto 0);
type matrix is array (4 downto 0) of vector;
signal rowarray: matrix;
begin

rowarray(0) <= "0000" & input;
d: For i in 1 to 4 generate
    rowarray(i) <= rowarray(i-1) (6 downto 0) & '0';
    end generate;
output <= rowarray(sel);
end dataflow;
```

Rowarray (0): 0 0 0 0 1 1 1 1

Rowarray (1): 0 0 0 1 1 1 1 0

Rowarray (2): 0 0 1 1 1 1 0 0

Rowarray (3): 0 1 1 1 1 0 0 0

Rowarray (4): 1 1 1 1 0 0 0 0

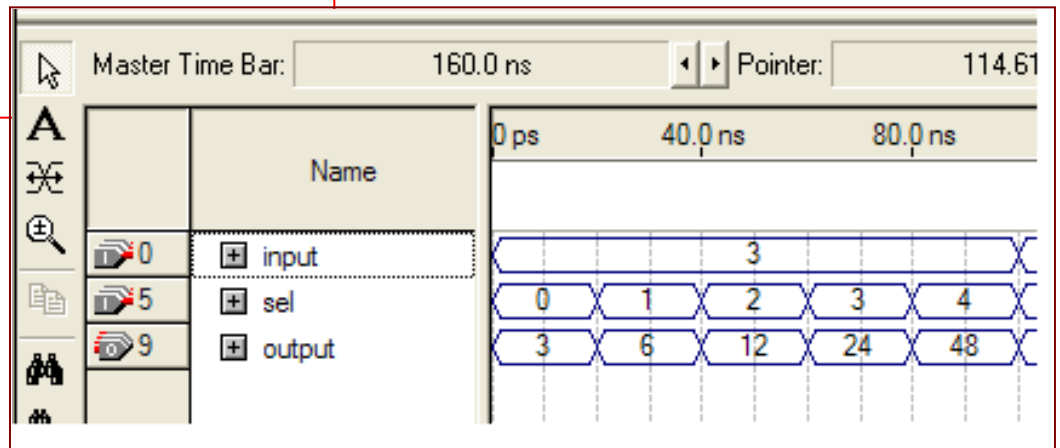


Fig: VHDL Code and Vector Waveform Output

Home Work

PART-II: SEQUENTIAL CODE

Sequential Code

- VHDL is inherently concurrent
- IF, WAIT, CASE, and LOOP are executed inside the PROCESSES, FUNCTIONS, and PROCEDURES that are sequentially processed.
- Variable is not global, should be declared inside the process
- Signal can be used globally

Sequential statements: The Process

- A VHDL construct used for grouping sequential statements
- Statements are processed sequentially during simulation
- Can be either *active* or *inactive* during simulation
- A Process typically has a SENSITIVITY LIST except when WAIT is used

PROCESS (*sensitivity list*)
-- optional *variable declarations*
BEGIN
sequential statements
END PROCESS ;

The Process Sensitivity List

- A Process is invoked when one or more of the signals within the sensitivity list change, e.g.

```
ARCHITECTURE archlist OF list IS
 BEGIN
      nand0: PROCESS (a,b)
      BEGIN
            c <= NOT (a AND b);
      END PROCESS nand;
 END archlist;
```

if either a or b changes in any way, the process is invoked

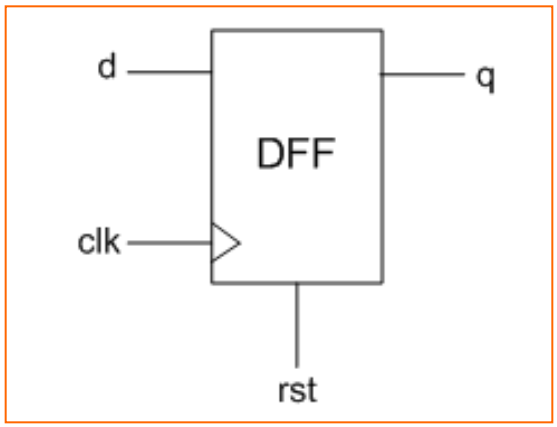
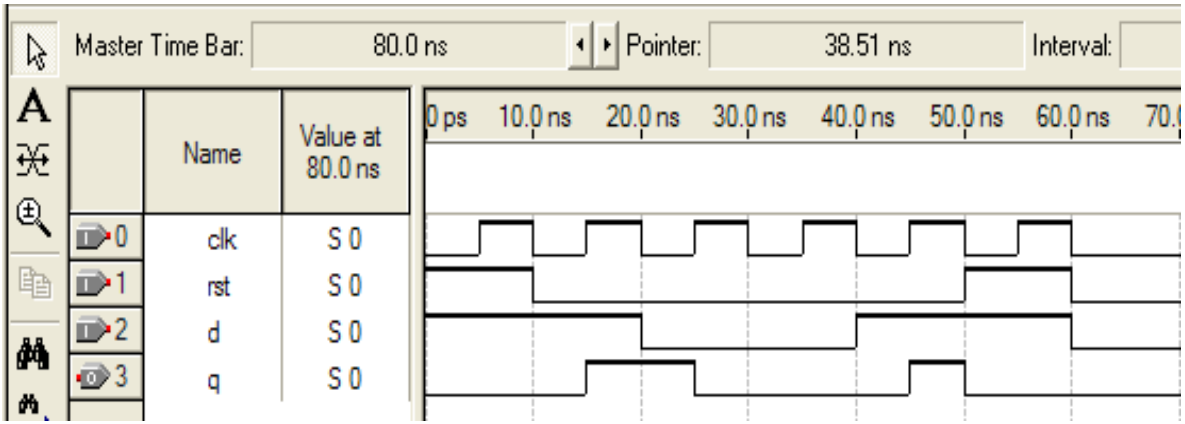


Fig: D Flip Flop Symbol

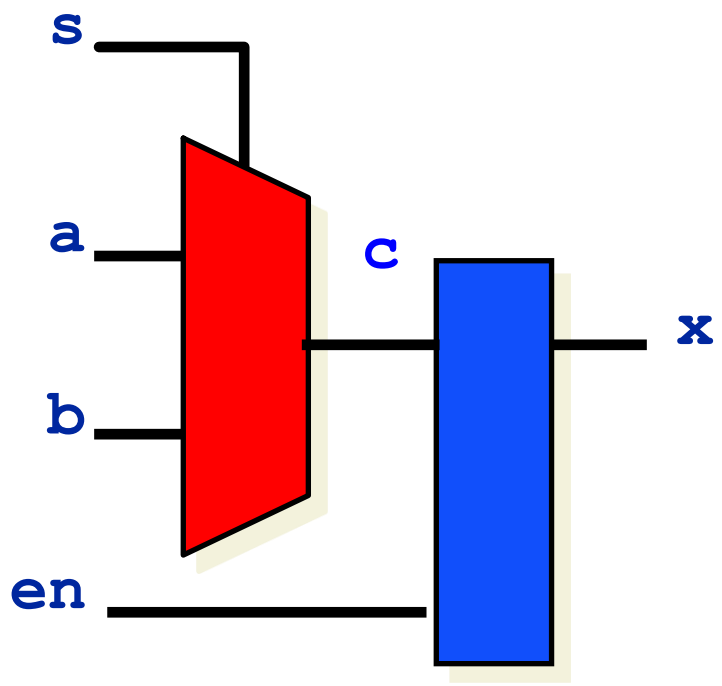
```
library ieee;
use ieee.std_logic_1164.all;

entity FFD is
port (d,clk,rst: in std_logic;
        q: out std_logic);
end FFD;

architecture behav of FFD is

begin
FlipFlo: process (clk,rst)
begin
    if (rst='1') then
            q<='0';
    elsif (clk'event and clk='1') then
        q<=d;
    end if;
end process;
end behav;
```

Fig: DFF Vector Waveform



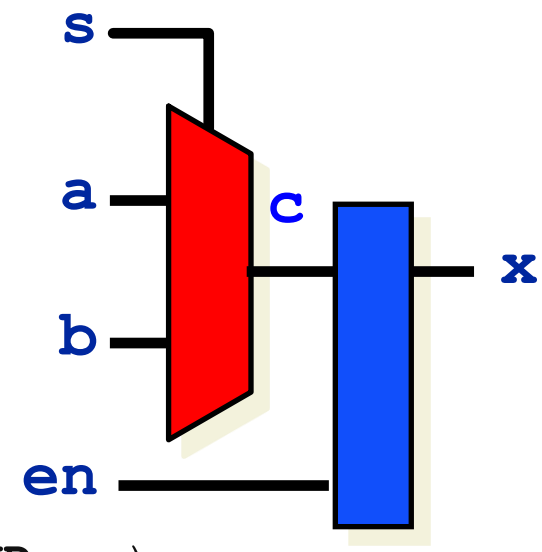
Signal Assignment in Processes

```
LIBRARY ieee;
USE
ieee.std_logic_1164.ALL;
ENTITY mux2ltch IS PORT (
     a, b: IN std_logic;
     s, en: IN std_logic;
     x: BUFFER std_logic);
END mux2ltch;
```



Signal Assignment in Processes: Incorrect Solution

```
ARCHITECTURE archmux2ltch OF mux2ltch IS
      SIGNAL c: std_logic;
BEGIN
      mux: PROCESS (a,b,s,en)
      BEGIN
            IF s = '0' THEN c <= a;
            ELSE c <= b;
            END IF;
            x <= (x AND (NOT en)) OR (c AND en);
      END PROCESS mux;  -- c is updated here!
END archmux2ltch;
```

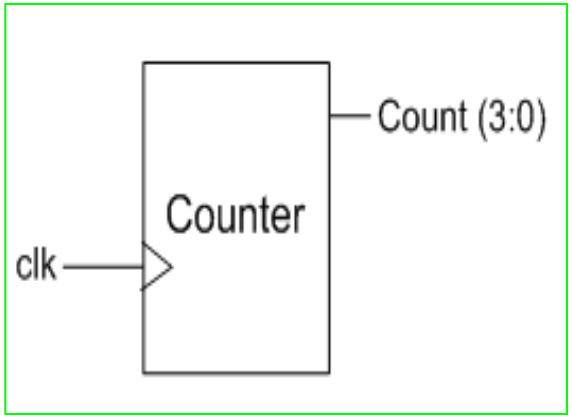


PROCESS: A correct solution

```
ARCHITECTURE archmux2ltch OF mux2ltch IS
     SIGNAL c: std_logic;
BEGIN
     mux: PROCESS (a, b, s)
     BEGIN
          IF s = '0' THEN c <= a;
          ELSE c <= b;
          END IF;
     END PROCESS mux;  -- c is updated here!
     x <= (x AND (NOT en)) OR (c AND en);
END archmux2ltch;
```

IF Statement

```
IF condition THEN assignment;
ELSIF condition THEN assignment;
   .....
ELSE assignments;
END IF;
```



Counter Example Code

```
library ieee;
use ieee.std_logic_1164.all;
entity counte_if is
port (clk   : in std_logic;
      count : out integer range 0 to 9);
end counte_if;
architecture behav of counte_if is
begin
countr: process (clk)
variable temp: integer range 0 to 10;
begin
    if (clk'event and clk='1') then
            temp:=temp+1;
    if (temp=10) then temp:=0;
            end if;
        end if;
    count<=temp;
end process;
end behav;
```

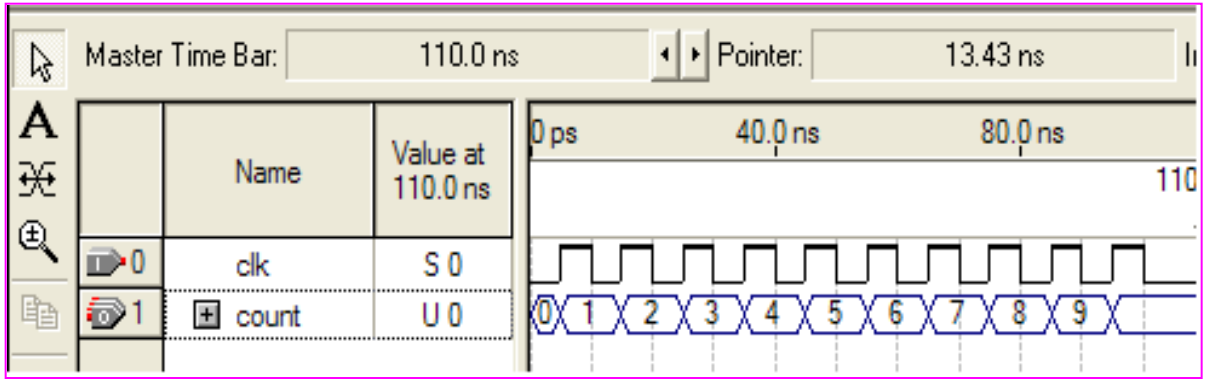


Fig: Counter Vector Waveform

WAIT

```
-- WAIT Statement Syntax
Wait until signal_condition;
Wait on signal [signal1, signal2, ...];
wait for time;
```

- No sensitivity list required
- WAIT UNTIL accepts one signal, WAIT ON accepts multiple, Wait For is only for simulation purpose
- Do yourself – develop DFF code using wait on instead of IF only, and simulate it in Quartus-II, verify the functionality and observe the area-performance differences .

```
-- WAIT Untill Sample Code

waitS: Process -- no sensitiviy list
begin
    wait untill (clk'event and clk='1');
    if (reset='1') then
      otpt<="00000000";
    elsif (clk'event and clk='1') then
      otpt<= input;
    end if;
  end process;


--Wait On Sample Code
waitS: Process
begin
  wait on clk, reset;
  if (reset = '1') then
    otpt<="00000000";
  elsif (clk'event and clk='1') then
    otpt<=input;
  end if;
end process;
```

CASE

```
--CASE Syntax

CASE identifier is
WHEN value => assignments;
WHEn value => assignments;
  ..
END case;
```

```
--CASE sample Code
CASE sel is
when "00" => x <=a; y<=b;
When "01" => x <=b; y<=c;
when others
```

- CASE statement has resemblance with WHEN
- Unlike WHEN, CASE allows multiple assignments

DFF with CASE Statement

```
Entity DFF_case is
port (d,clk,rst: in std_logic;
      q: out std_logic);
    end DFF_case;
architecture behav of DFF_case is
begin
process (clk,rst)
begin
case rst is
    when '1' => q <= '0';
      when '0' =>
        if (clk'event and clk='1') then
          q<=d;
        end if;
      when others=> NULL;
    end case;
  end process;
end behav;
```

LOOP: FOR and WHILE

```
--FOR LOOP Syntax
label: FOR identifier In range Loop
   (sequential statements)
end Loop [lable1];

-- WHILE LOOP Syntax
label: WHILE condition LOOP
   (sequential statements)
end Loop [lable1];

--EXIT Syntax
label: EXIT [label] [WHEN condition]

-- NEXT Syntax
label: NEXT [loop_label] [WHEN condition];
```

Sample Example Codes

```
--Example for FOR/LOOP

FOR i in 7 downto 0 LOOP
  a(i)<= enable AND t(i+2);
  r(0,i) <= t(i);
end Loop;
```

```
--Example code EXIT
d: for i in sample'range loop
  case sample (i) is
    when '0' => count:=count+1;
    when others => exit;
  end case;
end loop;
```

```
--Example code WHILE

While (i<5) loop
  wait untill clock'event and clock='1';
  (other statements)
end loop;
```

```
--Example Code NEXT
g: for i in 7 downto 0 loop
  next when i=skip;
  (.....)
end loop;
```

CASE Versus IF and WHEN

- CASE and IF allows selection of one sequence of statements for execution form a number of alternative sequences
- CASE vs WHEN
  - CASE is sequential, while WHEN is concurrent
  - CASE can only be used inside the process, FUNCTIONS, or PROCEDURES while WHEN outside is reverse
  - All permutations can be tested by both
  - WHEN can have any number of assignments per test, while CASE is limited to only one
  - NULL is the keyword for no-action in CASE, Unaffected is used in WHEN for no-action (shown previously in examples)

END OF THE LECTURE

Lecture 9 & 10