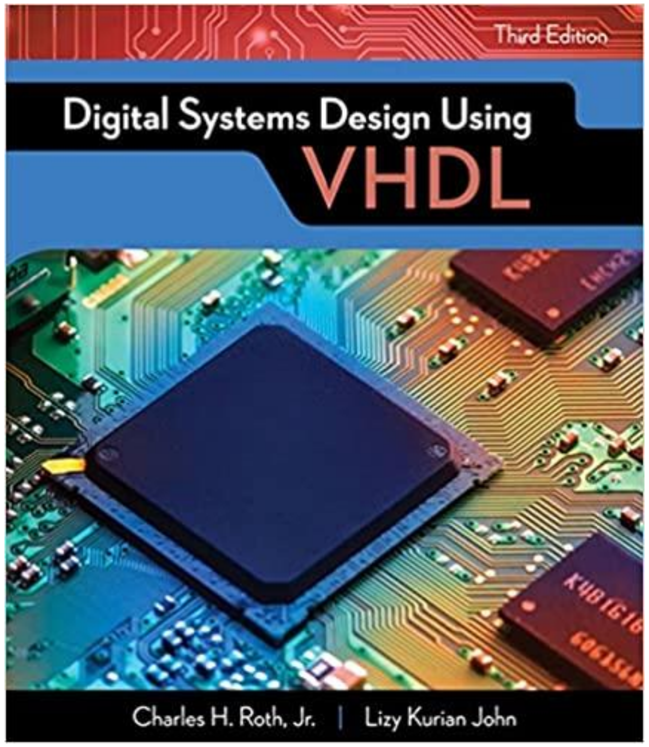
EE2000 Logic Circuit Design

Lecture 8– VHDL 2


Third Edition
Digital Systems Design Using
VHDL
Charles H. Roth, Jr. | Lizy Kurian John

What will you learn?

8.1 Use Component and instantiation

8.2 Understand the Conditional signal assignment

8.3 Understand the Selected signal assignment

8.4 Understand the Sequential statements

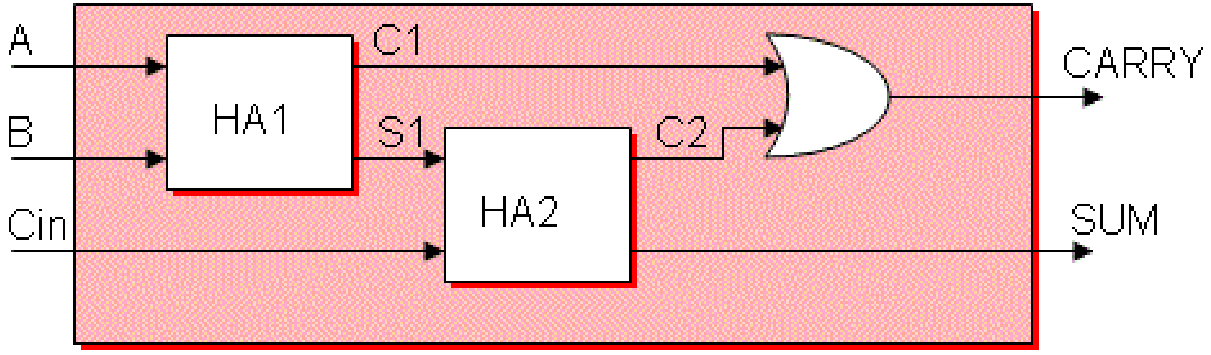
8.5 Understand various Decoder designs

- Using Boolean Operators
- Using Case statement
- Using IF statement
- With ENable Signal

8.6 Other examples – Encoder, MUX, DMUX, Flip-Flop

8.1 Components and Instantiation

- **Structural modeling**: Modular design of a complex project
- When designing a complex project, we can split it into two or more simple designs (sub-modules/sub-circuits/**components**)
- Example: A full adder (FA) contains of 2 half adders (HAs); Half adder can be modeled by a **component**



Structural Modeling

- **Structural modeling** or modular design allows us to pack low-level functionalities into modules
- Allows a designed module to be **reused** without the need to reinvent and re-test the same functions/modules every time
- To include a **component** into a **module**, we need to
  1. **declare** the component
  2. **instantiate** the component

  in **architecture**

Component Declaration

- An architecture may contain multiple components and they must be declared first

```
architecture [name] …
[signal]

      component XX           -- Component declaration
      …
      end component;

      component YY           -- Component declaration
      …
      end component;

begin
…          -- Component instantiation
end [name];
```

Half Adder

Create the sub-module of half adder first

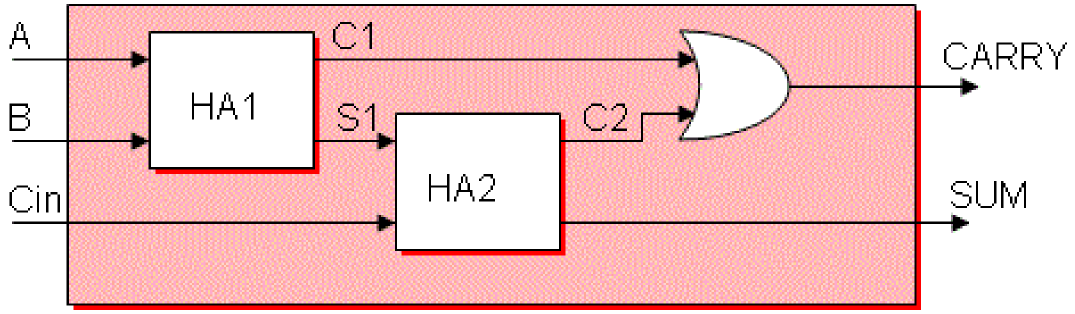
$$\mathrm{sum} = a \oplus b \quad \mathrm{carry} = a \cdot b = ab$$

Inputs		Outputs	
a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

```vhdl
--sub module(half adder) entity declaration
entity halfadder is
port (a : in  STD_LOGIC;
      b : in  STD_LOGIC;
      sum : out  STD_LOGIC;
      carry : out  STD_LOGIC
     );
end halfadder;

architecture Behavioral of halfadder is
begin
sum <= a xor b;
carry <= a and b;
end Behavioral;
```

Full Adder

- Create the component entity **halfadder**
- Create the module entity **fulladder**
- Determine the number of **components** (i.e. **2 halfadder** in this case) used in the design
- Define signals for inter-connections between **halfadder**
- Provide each component a different name
- Then instantiates the declared component



Full Adder

```
--top module(full adder) entity declaration
entity fulladder is
    port (a : in std_logic;
           b : in std_logic;
           cin : in std_logic;
           sum : out std_logic;
           carry : out std_logic
         );
end fulladder;

--top module architecture declaration
architecture behavior of fulladder is

   component halfadder
    port(
         a : in std_logic;
         b : in std_logic;
         sum : out std_logic;
         carry : out std_logic
        );
    end component;
```

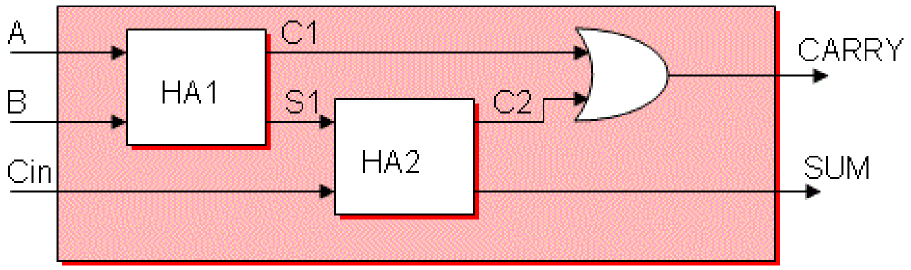
--sub-module(half adder) is declared as a component before the keyword "begin"

Component Instantiation

Differences between a **component** and an **entity** declaration:

- **Entity** declaration declares a circuit model containing one or multiple architectures
- **Component** declaration declares a **virtual circuit** template, which must be **instantiated** to take effect during the design
- **Instantiation** – To map the signals in the entity with the input/output of the component
- **Port map** is required for component **instantiation**

Full Adder



- Two HAs are needed
- Internal signals s1,c1,c2 are used to connect the two Has
- In HA, we define **port** (a:**in** STD_LOGIC; b:**in** STD_LOGIC; sum:**out** STD_LOGIC; carry:**out** STD_LOGIC);

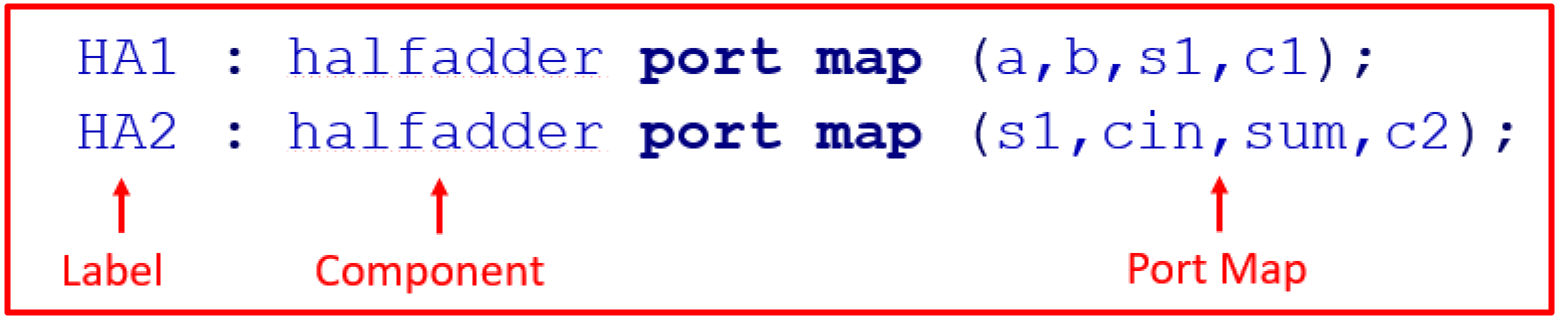
```
signal s1,c1,c2 : std_logic:='0’;  --declare internal signal

begin

--Provide a different name for each half adder.
--instantiate and do port map for the half adders.

HA1 : halfadder port map (a,b,s1,c1);
HA2 : halfadder port map (s1,cin,sum,c2);
carry <= c1 or c2;  --final carry calculation

end;
```

Component Instantiation

For creating connections between components and ports,

3 steps in VHDL instantiation:

- Label: identify a unique **instance** of component
- Component type: select a targeted declared **component**
- Port Map: Connect **component to signals**



```
HA1 : halfadder port map (a,b,s1,c1);
HA2 : halfadder port map (s1,cin,sum,c2);
```

Signals must be of the **same data type** for the connecting pins

Component Instantiation

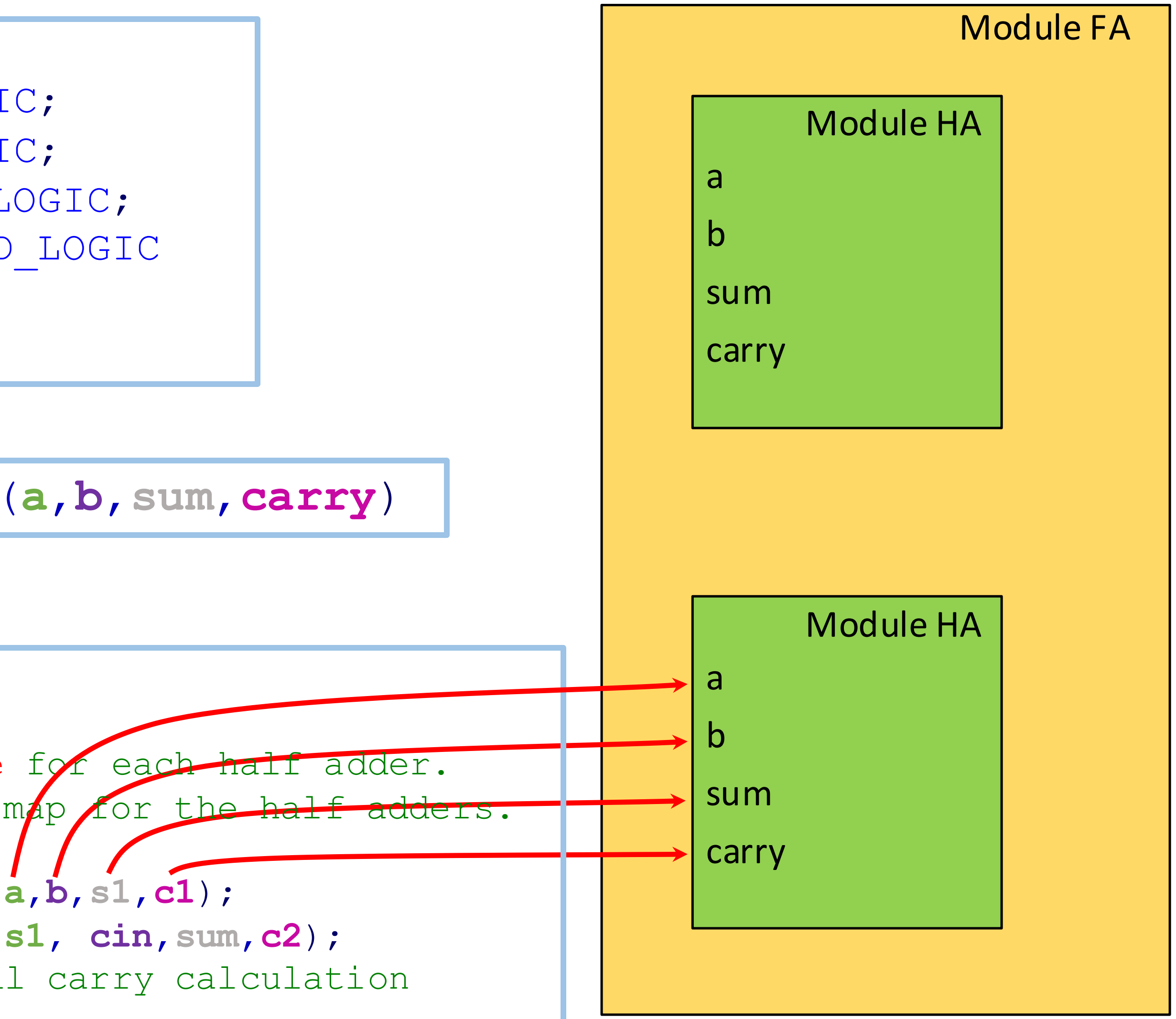
```
signal s1,c1,c2 : std_logic:='0';  --declare internal signal

begin

--Provide a different name for each half adder.
--instantiate and do port map for the half adders.

HA1 : halfadder port map (a,b,s1,c1);
HA2 : halfadder port map (
       a => s1,
       b => cin,
       sum => sum,
       carry => c2
);
carry <= c1 or c2;  --final carry calculation

end;
```

Port Map

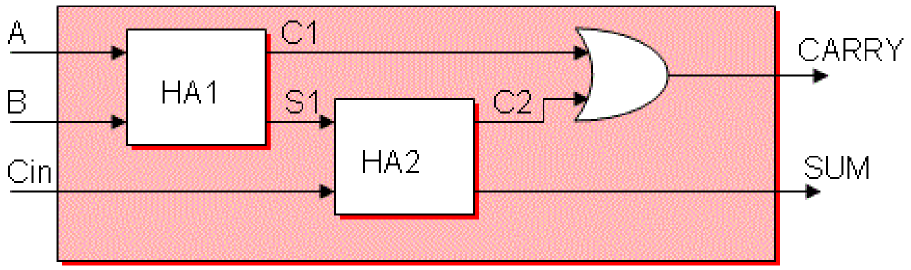
```
entity halfadder is
Port ( a : in  STD_LOGIC;
       b : in  STD_LOGIC;
       sum : out  STD_LOGIC;
       carry : out  STD_LOGIC
    );
end halfadder;
```

Port name of `Halfadder (a,b,sum,carry)`

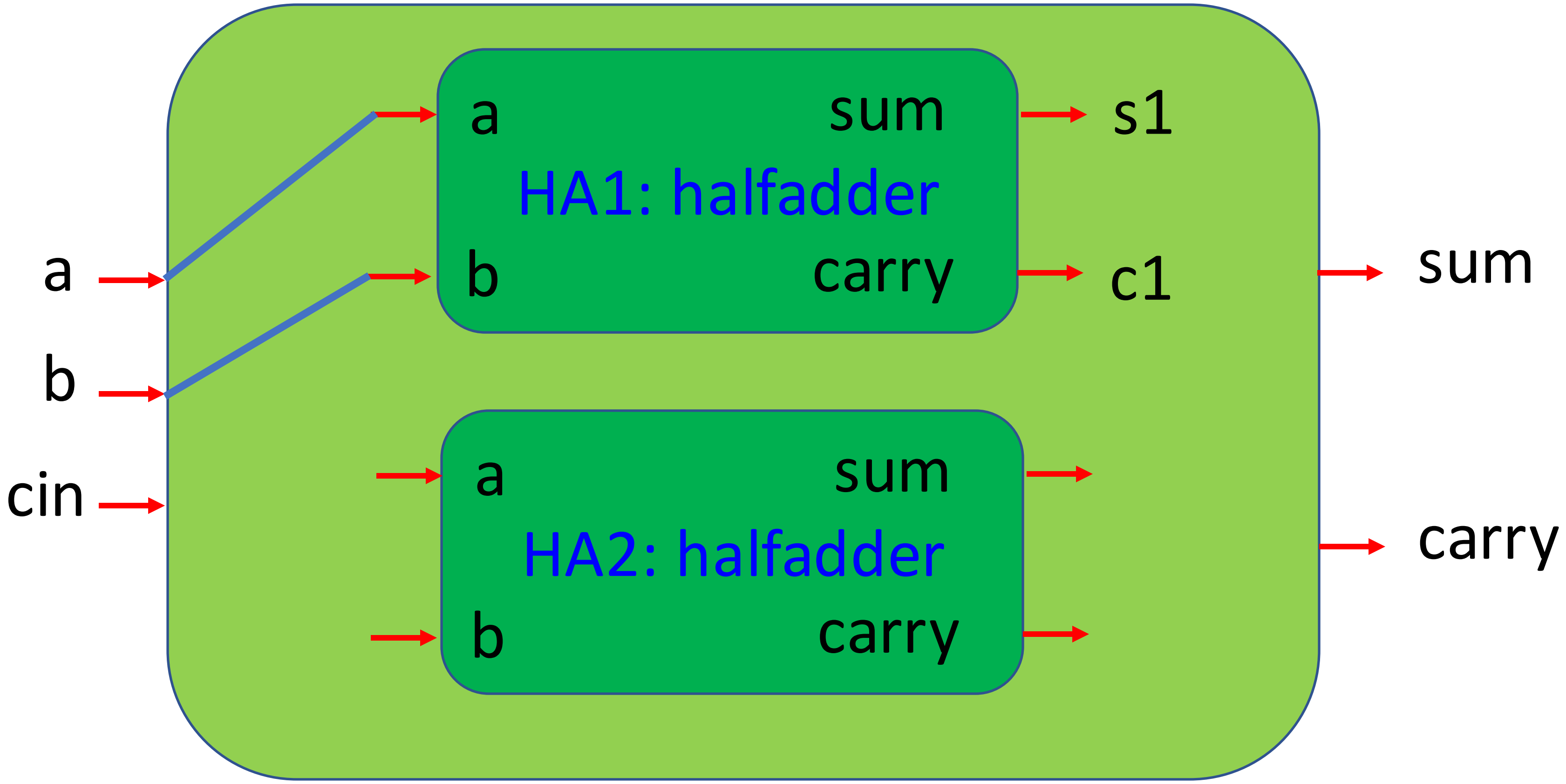
```
begin

--Provide a different name for each half adder.
--instantiate and do port map for the half adders.

HA1 : halfadder port map (a,b,s1,c1);
HA2 : halfadder port map (s1, cin,sum,c2);
carry <= c1 or c2;  --final carry calculation

end;
```



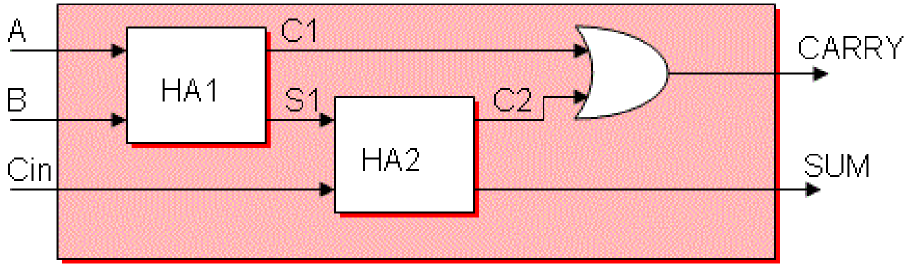
Half-Adder 1



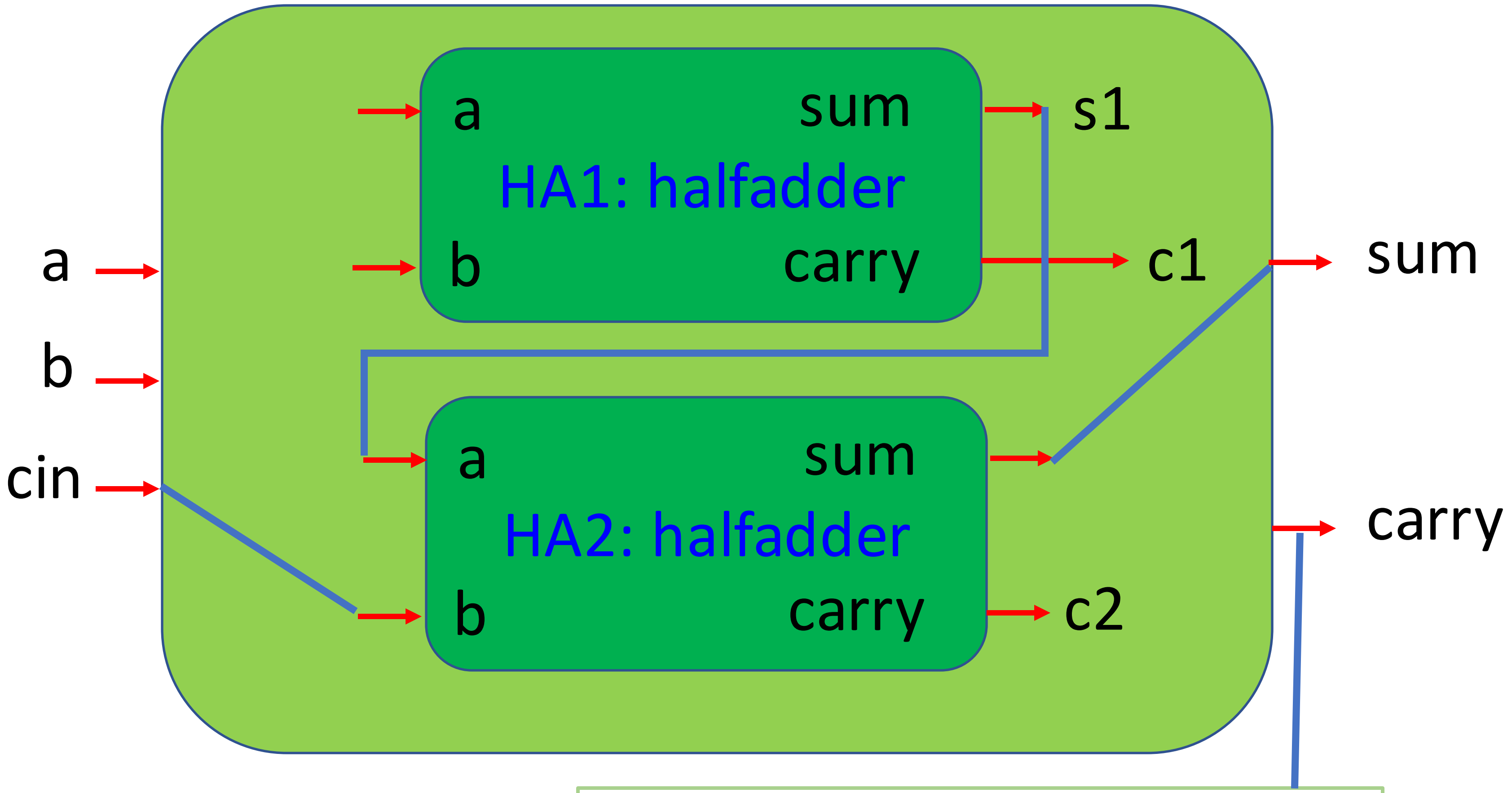
```
HA1 : halfadder port map (a,b,s1,c1);
```



Half-Adder 2



```
HA2 : halfadder port map (s1,cin,sum,c2);
```



```
carry <= c1 or c2;
```

8.2 Conditional Signal Assignment

```
signal_name <= expression1 when condition1 else
               expression2 when condition2 else
              [expressionN];
```

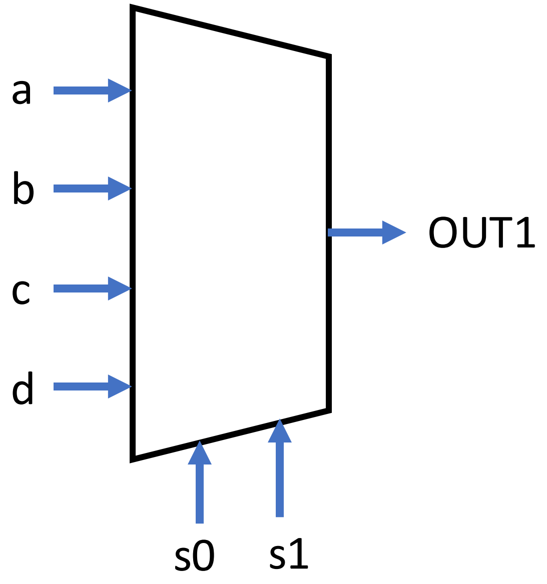
- Concurrent statement
- Conditions are evaluated successively until a true condition is found
- Conditions could be based on different signals

```
d <= a when b = '1' else
     c when d = '0' else
      e;
```

This means

1. When b = ‘1’ then d = a or else
2. When d = ‘0’ then d = c else
3. d = e

Example 4-to-1 MUX



```
OUT1 <= a when (s1 = '0' and s0 = '0') else
         b when (s1 = '0' and s0 = '1') else
        c when (s1 = '1' and s0 = '0') else
        d;
```

8.3 Selected Signal Assignment

```
with expression_s select
 signal_s <= expression1 [after delay-time] when choice1,
             expression2 [after delay-time] when choice2,
                …
             expression_n [after delay-time] when others];
```

- Concurrent statement
- Each line ends with ‘,’ and the last line with ‘;’
- “when others” is used to handle the default case, and also the don’t care cases
- No priority and based on a single signal

Examples

```
with d select
   Y <= '0' when "000",
      '1' when "001",
      '1' when "010",
      '0' when "011",
      '1' when "100",
      '0' when "101",
      '1' when "110",
      '1' when "111",
     NULL when others;
```

This means

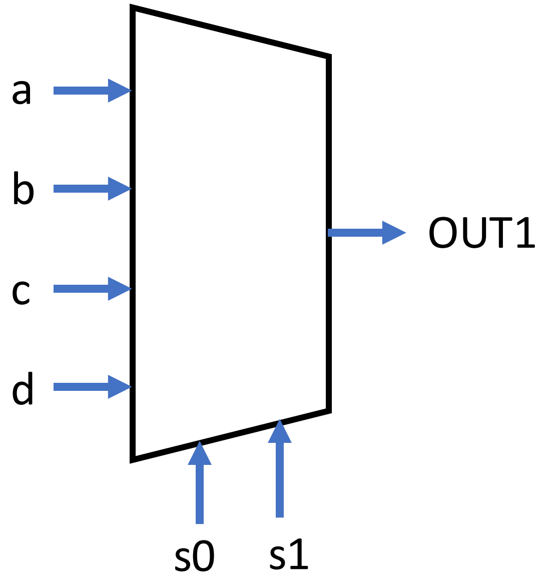
1. When d = ‘000’ then Y = ‘0’ or else
2. When d = ‘001’ then Y = ‘1’ or else
3. When d = ‘010’ then Y = ‘1’ or else

……

9. For other cases (don’t care), then Y = NULL

Example 4-to-1 MUX

```
signal sel: integer;

sel <= 0 when (s1 = '0' and
s0 = '0') else
         1 when (s1 = '0' and
s0 = '1') else
        2 when (s1 = '1' and
s0 = '0') else
        3;

with sel select
    OUT1 <= a when 0,
            b when 1,
            c when 2,
            d when others;
```



This means

1. When sel = '0' (00) then OUT1 = a or else
2. When sel = '1' (01) then OUT1 = b or else
3. When sel = '2' (10) then OUT1 = c or else
4. OUT1 = d

Conditional

```
d <= a when b = '1' else
     c when d = '0' else
      e;
```

- Can be based on different signals
- Evaluated successively until a true condition is found

Selected

```
with d select
 Y <= '0' when "000",
    '0' when "011",
    '0' when "101",
    '1' when others;
```

- Based on a single signal only
- Only one condition is TRUE

Question: Which one will you use for the priority encoder?

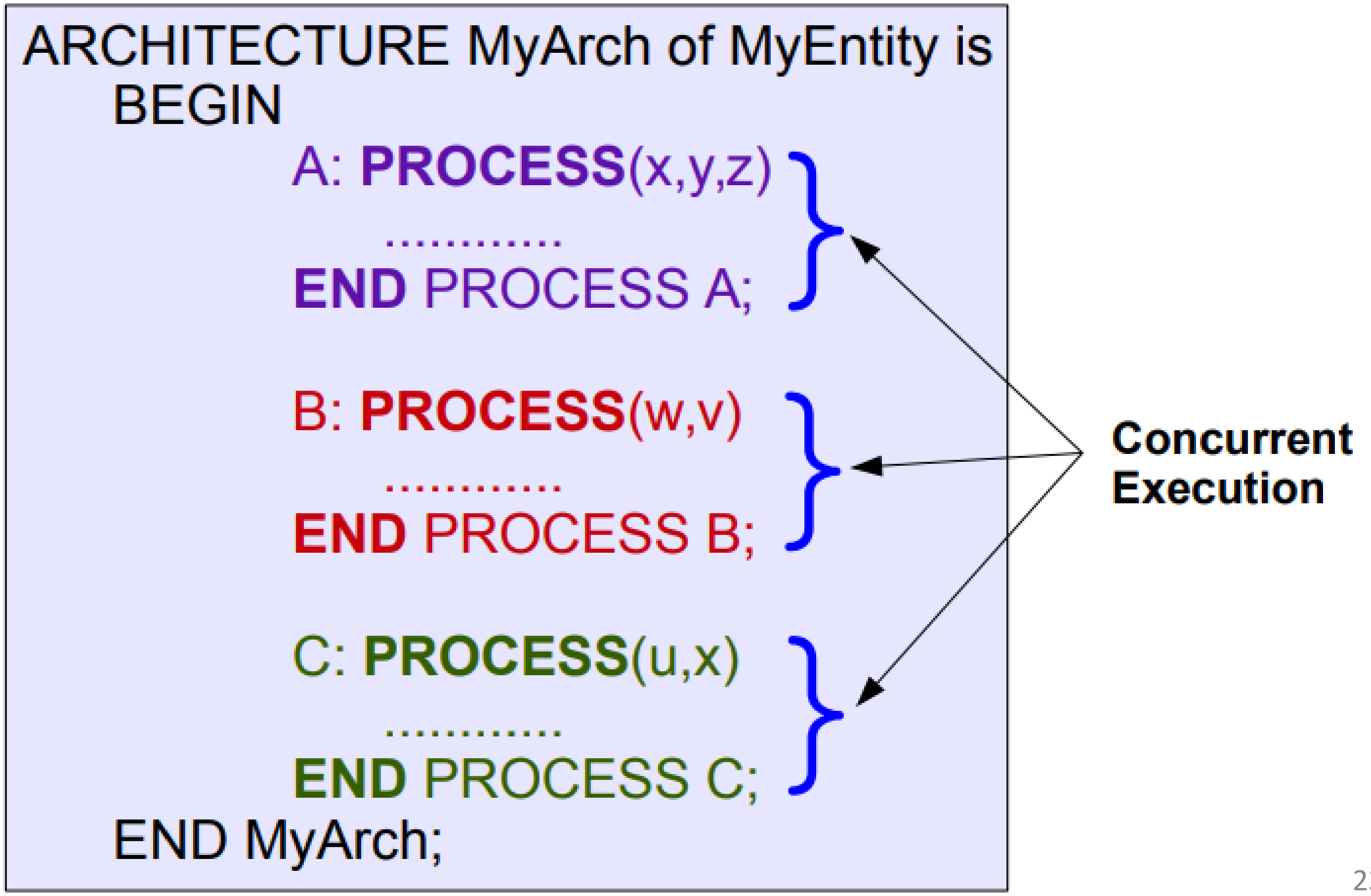
8.4 Sequential Statements

- **Process statement** is used to enclose sequential statements that are executed in order
- **Sequential statements** are used in processes to specify how signals are assigned
- After all the sequential statements in the process are executed, the signals are assigned to their new values

```
[name:] process [(sensitivity_list)]
begin
   sequential statements
end process;
```

Multiple Process Statements

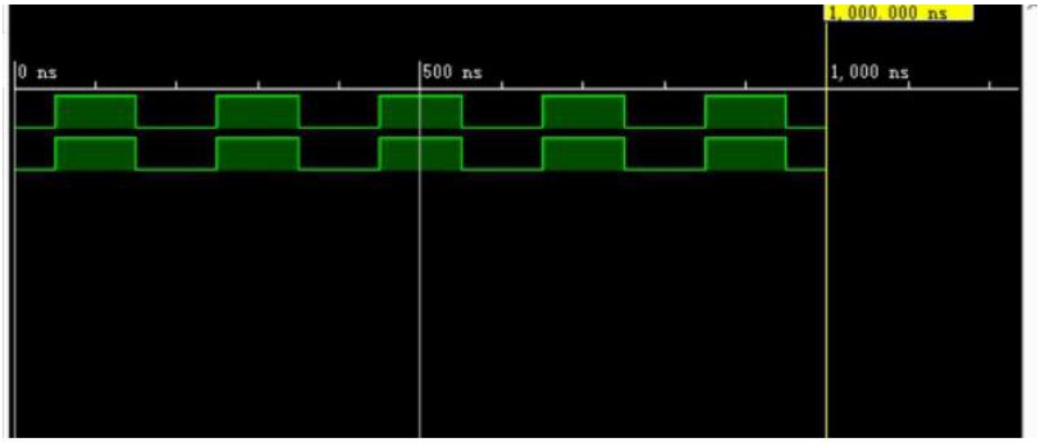
Process statement is concurrent statement and can have more than one Process statements in an architecture



```
ARCHITECTURE MyArch of MyEntity is
    BEGIN
            A: PROCESS(x,y,z)
                ............
            END PROCESS A;

            B: PROCESS(w,v)
                ............
            END PROCESS B;

            C: PROCESS(u,x)
                ............
            END PROCESS C;
    END MyArch;
```

Recap from Lab Session 1

```
simgen: process        -- no sensitivity list
begin
   sw <= '0';
   wait for 50 ns;
   sw <= '1';
   wait for 100 ns;
   sw <= '0';
   wait for 50 ns
end process;
```



- **Without** the sensitivity list, the process will be run **continuously**
- **With** the sensitivity list, the process will be executed once when a **new event (change value)** occurs on any of the signals in the list

Sensitivity List

```
proc1: process (a, b, c)
begin
   x <= a and b and c;
end process;
```

When either a, b or c changes from ‘1’ to ‘0’ or vice versa, the process will run one time to update the value of x

```
proc1: process
begin
   x <= a and b and c;
   wait on a, b, c;
end process;
```

Wait Statement Forms

```
wait for 50 ns;

wait on a, b, c;

wait until signal = value;
wait until clk = ‘1’;
```

The process will pause until clk changes to ‘1’

WHILE Loop Statement

```
[label:] while condition loop
      sequential statements
end loop [label];
```

- Conditional loop statement
- Condition is tested before the execution of the loop
- Terminate when the condition tested becomes false

```
while i < 10 loop
    while j < 20 loop
      ...
        j <= j + 1;
    end loop;
    i <= i + 1;
end loop;
```

FOR Loop Statement

```
[label:] for counter in range loop
      sequential statements
end loop [label];
```

- For the repeated execution of a sequence of statements a fixed number of times
- An iteration counter and a range are specified
- After an iteration, the counter is assigned the next value from the range
- Ascending order use **to**; descending order use **downto**

FOR Loop Statement

Compute the squares of integer values between 1 and 10 and stores them into the i_square array

```
for i in 1 to 10 loop
    i_square (i) <= i * i;
end loop;
```

i starts with 1, after computing i_square (1)

i becomes 2 (i = 2).....

Until i = 10.

FOR Loop Statement

```
entity match_bits is
 port (a, b: in bit_vector (7 downto 0);
       matches: out bit_vector (7 downto 0));
end match_bits

architecture functional of match_bits is
begin
   process (a, b)
   begin
      for i in 7 downto 0 loop
         matches (i) <= not (a(i) xor b(i));
      end loop;
end process;
end functional;
```

what is the function?

FOR Loop Statement

- A set of **1-bit comparators** to compare the bits of the same order of vectors **a** and **b**
- Result is stored into the **matches** vector, which will contain '1' wherever the bits of the two vectors match and '0' otherwise

```
entity match_bits is
 port (a, b: in bit_vector (7 downto 0);
        matches: out bit_vector (7 downto 0));
end match_bits

architecture functional of match_bits is
begin
   process (a, b)
   begin
      for i in 7 downto 0 loop
         matches (i) <= not (a(i) xor b(i));
      end loop;
end process;
end functional;
```

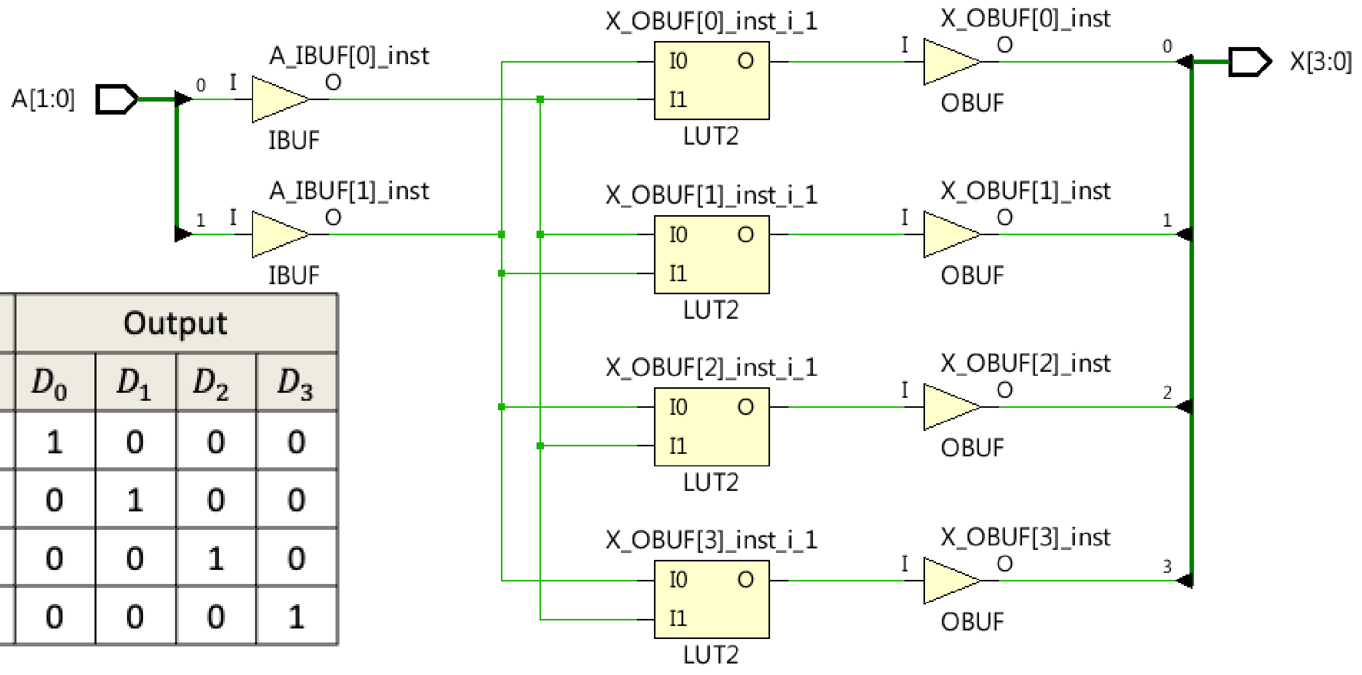
NEXT Statement

```
next [label:] [when condition];
```

The execution of the current iteration is skipped and the control is passed to the beginning of the loop statement

```
for i in 1 to 10 loop
   next when v(i) = '0';
   count <= count + 1;
end loop;
```

8.5 Decoder Designs (2-to-4 decoder)



Input		Output			
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

- Input A (2 bits)
- Output X (4 bits)

Using Boolean Operators

Input		Output			
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity decoder is
port(
    A : in STD_LOGIC_VECTOR(1 downto 0);
    X : out STD_LOGIC_VECTOR(3 downto 0)
);
end decoder;

architecture Structral of decoder is
begin
    X(0) <= not A(0) and not A(1);
    X(1) <=  A(0) and not A(1);
    X(2) <= not A(0) and A(1);
    X(3) <= A(0) and A(1);
end Structral;
```

Using CASE statement (Sequential)

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity decoder is
port(
    A : in STD_LOGIC_VECTOR(1 downto 0);
    X : out STD_LOGIC_VECTOR(3 downto 0)
);
end decoder;

architecture Behavioral of decoder is
begin

    process(a)
    begin
        case A is
            when "00" => X <= "0001";
            when "01" => X <= "0010";
            when "10" => X <= "0100";
            when "11" => X <= "1000";
        end case;
    end process;
end Behavioral;
```

```
case expression is
    when option1 =>
statement;
    when option2 =>
statement;
    ...
    [when others =>
statement;]
end case;
```

When A is “00” X <= “0001”

Input		Output			
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Using IF statement (Sequential)

```
entity decoder is
port(
    A : in STD_LOGIC_VECTOR(1 downto 0);
    X : out STD_LOGIC_VECTOR(3 downto 0)
);
end decoder;

architecture Behavioral of decoder is
begin
    process(a)
    begin
     if (A="00") then
        X <= "0001";
     elsif (A="01") then
        X <= "0010";
     elsif (A="10") then
        X <= "0100";
     else
        X <= "1000";
     end if;
    end process;
end Behavioral;
```

```
if condition then
    statement;
[elseif condition then
   statement;]
    ...
[else statement;]
end if;
```

If A is “00” X <= “0001”

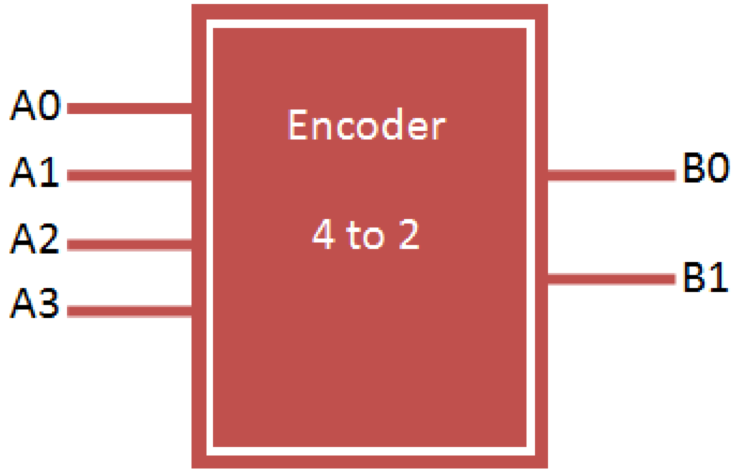
Input		Output			
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

With Enable Input

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decode_2to4_top is
    Port ( A  : in  STD_LOGIC_VECTOR (1 downto 0);     -- 2-bit input
           X  : out STD_LOGIC_VECTOR (3 downto 0);     -- 4-bit output
           EN : in  STD_LOGIC);                         -- enable input
end decode_2to4_top;

architecture Behavioral of decode_2to4_top is
begin
  process (A, EN)
  begin
      X <= "1111";          -- default output value
      if (EN = '1') then    -- active high enable pin
          case A is
              when "00" => X(0) <= '0';
              when "01" => X(1) <= '0';
              when "10" => X(2) <= '0';
              when "11" => X(3) <= '0';
              when others => X <= "1111";
          end case;
      end if;
  end process;
end Behavioral;
```

Active Low

8.6 Other Examples



```
entity encoder is
    port(
        a : in STD_LOGIC_VECTOR(3 downto 0);
        b : out STD_LOGIC_VECTOR(1 downto 0)
    );
end encoder;

architecture Behavioral of encoder is
begin
    process(a)
    begin
        case a is
            when "1000" => b <= "00";
            when "0100" => b <= "01";
            when "0010" => b <= "10";
            when "0001" => b <= "11";
            when others => b <= "ZZ";
        end case;
    end process;
end Behavioral;
```

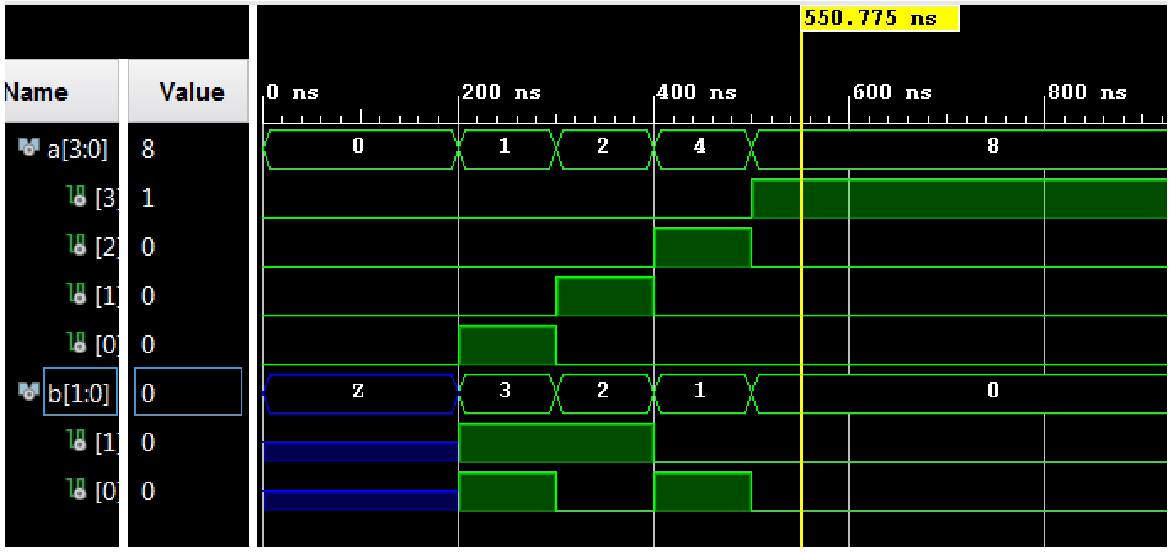
Input				Output	
A_3	A_2	A_1	A_0	B_1	B_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Simulation

Input				Output	
A_3	A_2	A_1	A_0	B_1	B_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

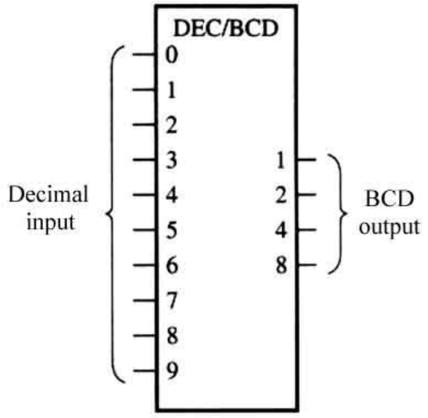
```vhdl
entity encoder is
    port(
        a : in STD_LOGIC_VECTOR(3 downto 0);
        b : out STD_LOGIC_VECTOR(1 downto 0)
    );
end encoder;

architecture Behavioral of encoder is
begin
    process(a)
    begin
        case a is
            when "1000" => b <= "00";
            when "0100" => b <= "01";
            when "0010" => b <= "10";
            when "0001" => b <= "11";
            when others => b <= "ZZ";
        end case;
    end process;
 end Behavioral;
```

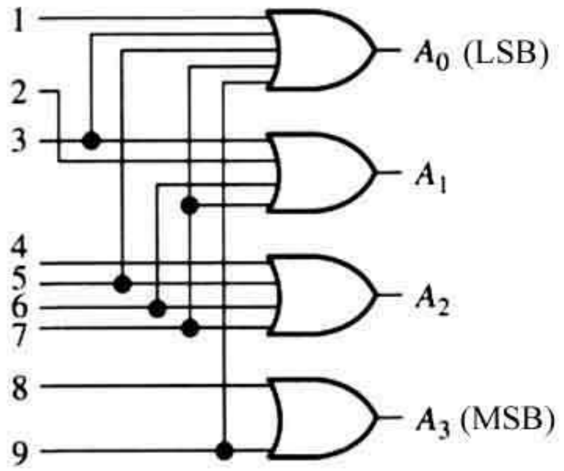
```vhdl
ENTITY tb_encoder IS
END tb_encoder;

ARCHITECTURE behavior OF tb_encoder IS
COMPONENT encoder
    PORT(
        a : IN std_logic_vector(3 downto 0);
        b : OUT std_logic_vector(1 downto 0)
    );
END COMPONENT;
signal a : std_logic_vector(3 downto 0) := (others => '0');
signal b : std_logic_vector(1 downto 0);
BEGIN
uut: encoder PORT MAP (a => a, b => b);
stim_proc: process
begin
    -- hold reset state for 100 ns.
    wait for 100 ns;
    a <= "0000";
    wait for 100 ns;
    a <= "0001";
    wait for 100 ns;
    a <= "0010";
    wait for 100 ns;
    a <= "0100";
    wait for 100 ns;
    a <= "1000";
    wait;
end process;
END;
```



Recap (Decimal-to-BCD Encoder)

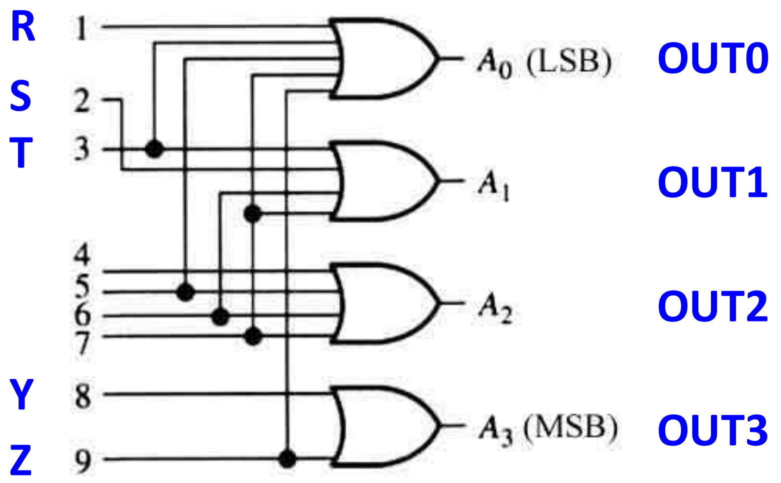
Inputs										Outputs			
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	A_3	A_2	A_1	A_0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	1	1	0
0	0	0	0	0	0	0	1	0	0	0	1	1	1
0	0	0	0	0	0	0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1	0	0	1



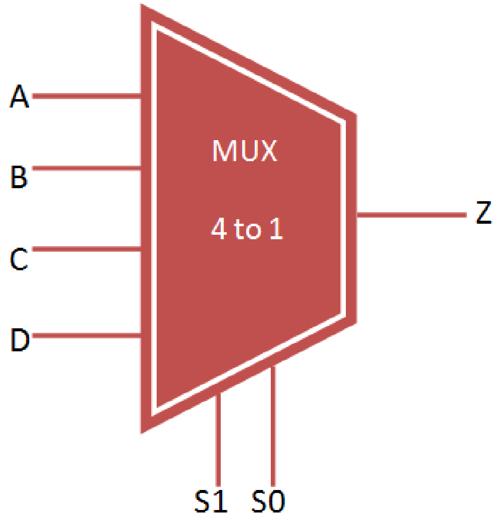


Encoder (Decimal-to-BCD Encoder)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ENC3 is
    Port ( Q : in std_logic;
           R : in std_logic;
           S : in std_logic;
           T : in std_logic;
           U : in std_logic;
           V : in std_logic;
           W : in std_logic;
           X : in std_logic;
           Y : in std_logic;
           Z : in std_logic;
           OUT0 : out std_logic;
           OUT1 : out std_logic;
           OUT2 : out std_logic;
           OUT3 : out std_logic);
end ENC3;
architecture Behavioral of ENC3 is
begin
            process (Q,R,S,T,U,V,W,X,Y,Z)
            begin

                        Enter your codes here

            end process;
end Behavioral;
```



Multiplexer



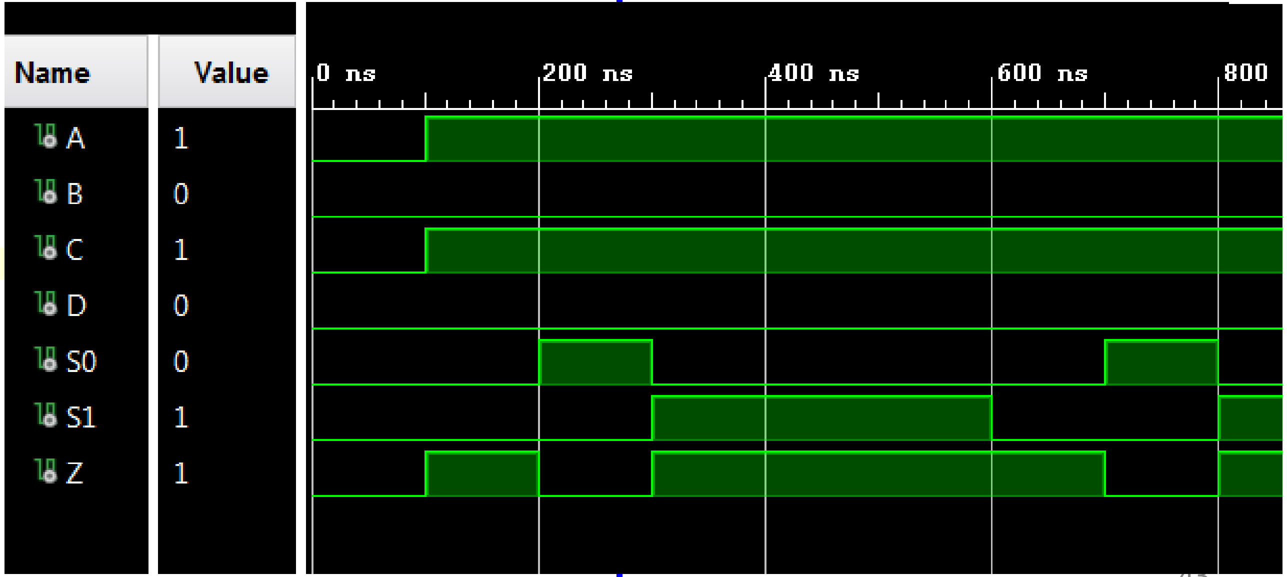
Input		output
S1	S0	Z
0	0	A
0	1	B
1	0	C
1	1	D

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mux_4to1 is
 port(
     A,B,C,D : in STD_LOGIC;
     S0,S1: in STD_LOGIC;
     Z: out STD_LOGIC
  );
end mux_4to1;

architecture bhv of mux_4to1 is
begin
    process (A,B,C,D,S0,S1) is
    begin
      if (S0 ='0' and S1 = '0') then
          Z <= A;
      elsif (S0 ='1' and S1 = '0') then
          Z <= B;
      elsif (S0 ='0' and S1 = '1') then
          Z <= C;
      else
          Z <= D;
      end if;
    end process;
end bhv;
```

Simulation

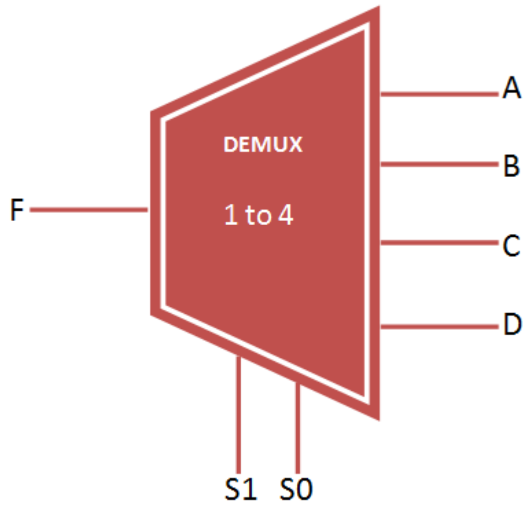
```
ENTITY tb_mux IS
END tb_mux;
ARCHITECTURE behavior OF tb_mux IS
   COMPONENT mux_4to1
   PORT(
         A : IN  std_logic;
         B : IN  std_logic;
         C : IN  std_logic;
         D : IN  std_logic;
         S0 : IN  std_logic;
         S1 : IN  std_logic;
         Z : OUT  std_logic
        );
   END COMPONENT;
   signal A, B, C, D, S0, S1 : std_logic := '0';
   signal Z : std_logic;
BEGIN
   uut: mux_4to1 PORT MAP (A => A,B => B,C => C,D => D,S0 => S0,S1 => S1,Z => Z);
   stim_proc: process
   begin
    -- hold reset state for 100 ns.
    wait for 100 ns;
    A <= '1';
    B <= '0';
    C <= '1';
    D <= '0';
    S0 <= '0'; S1 <= '0';
    wait for 100 ns;
    S0 <= '1'; S1 <= '0';
    wait for 100 ns;
    S0 <= '0'; S1 <= '1';
    wait for 100 ns;
    S0 <= '0'; S1 <= '1';
    wait for 100 ns;
    end process;

END;
```

Input		output
S1	S0	Z
0	0	A
0	1	B
1	0	C
1	1	D

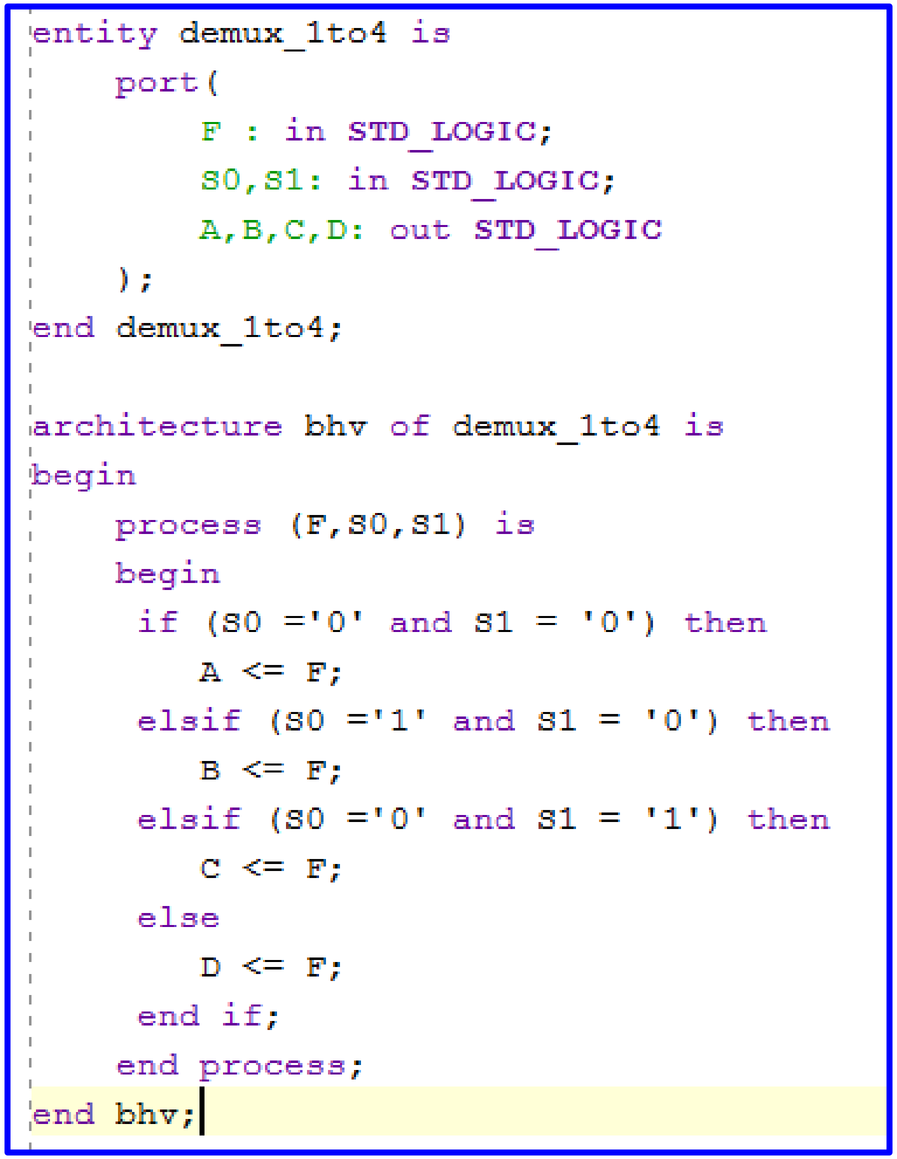


Name	Value
A	1
B	0
C	1
D	0
S0	0
S1	1
Z	1

Demultiplexer

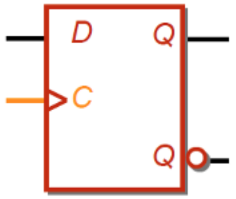


Input	Selection line		Output			
F	S1	S0	D	C	B	A
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
0	X	X	0	0	0	0



```
entity demux_1to4 is
    port(
        F : in STD_LOGIC;
        S0,S1: in STD_LOGIC;
        A,B,C,D: out STD_LOGIC
    );
end demux_1to4;

architecture bhv of demux_1to4 is
begin
    process (F,S0,S1) is
    begin
     if (S0 ='0' and S1 = '0') then
        A <= F;
     elsif (S0 ='1' and S1 = '0') then
        B <= F;
     elsif (S0 ='0' and S1 = '1') then
        C <= F;
     else
        D <= F;
     end if;
    end process;
end bhv;
```

D-Flip Flop



Inputs		Outputs		
D	***Clk***	**Next *Q***	**Next *Q'***	**State**
x	↓,0,1	*Q*	*Q'*	Hold
x	↑	*D*	*D'*	Set / Reset

Inputs		Outputs		
D	***Clk***	**Next *Q***	**Next *Q'***	**State**
x	↑,0,1	*Q*	*Q'*	Hold
x	↓	*D*	*D'*	Set / Reset

Rising-edge triggered

```
process(clk)
begin
  if rising_edge(clk)
then
    q <= d;
  end if;
end process;
```

Falling-edge triggered

```
process(clk)
begin
  if falling_edge(clk)
then
    q <= d;
  end if;
end process;
```

D-Flip Flop with RESET

Edge condition comes before reset

Syn active “high” reset
Rising edge FF

```
process(clk)
begin
  if rising_edge(clk)
then
    if sreset = '1' then
      q <= '0';
    else
      q <= d;
    end if;
  end if;
end process;
```

Syn active “low” reset
Falling edge FF

```
process(clk)
begin
  if falling_edge(clk)
then
    if sreset = ‘0’ then
      q <= '0';
    else
      q <= d;
    end if;
  end if;
end process;
```

D-Flip Flop with RESET

Reset condition comes before edge

Asyn active “high” reset

Rising edge FF

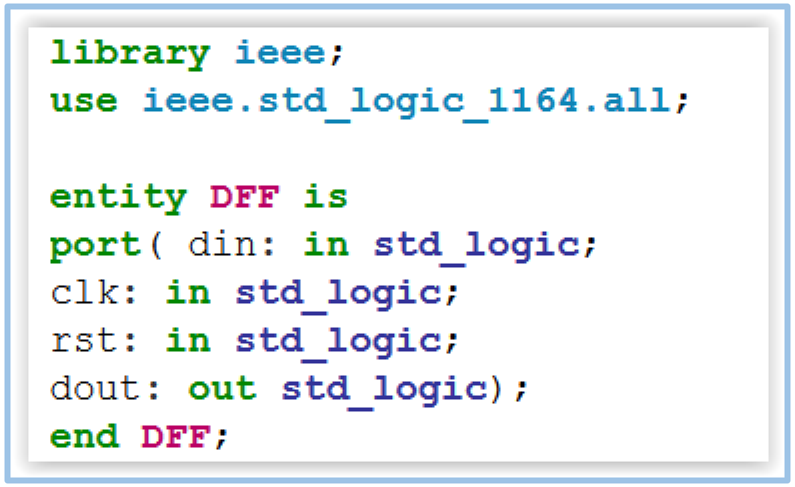
```
process(clk, areset)
begin
  if areset = '1' then
    q <= '0';
  elsif rising_edge(clk) then
    q <= d;
  end if;
end process;
```

Asyn active “low” reset

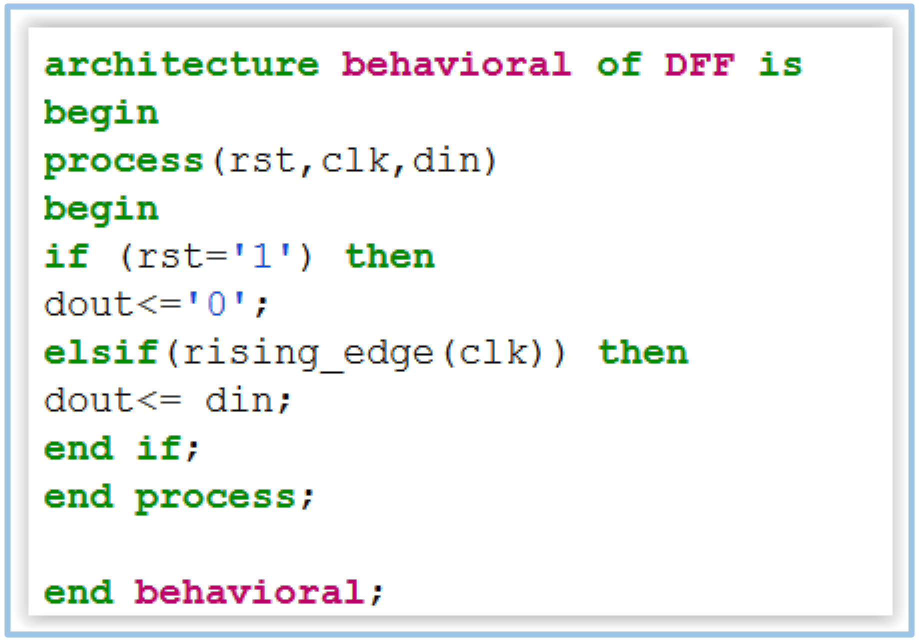
Falling edge FF

```
process(clk, areset)
begin
  if areset = ‘0' then
    q <= '0';
  elsif falling_edge(clk) then
    q <= d;
  end if;
end process;
```

D-Flip Flop with RESET



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

entity DFF is
port( din: in std_logic;
clk: in std_logic;
rst: in std_logic;
dout: out std_logic);
end DFF;
```



```vhdl
architecture behavioral of DFF is
begin
process(rst,clk,din)
begin
if (rst='1') then
dout<='0';
elsif(rising_edge(clk)) then
dout<= din;
end if;
end process;

end behavioral;
```

D-Flip Flop with SET and RESET

Asyn **set** active "high" , **reset** active "low"

Rising edge FF

```
process(clk, areset, aset)
begin
   if aset = '1' then
      q <= '1';
   elsif areset = '0' then
      q <= '0';
   elsif rising_edge(clk)
then
      q <= d;
   end if;
end process;
```

Note: **set** has higher priority than **reset**

T-Flip Flop with RESET

Rising edge FF

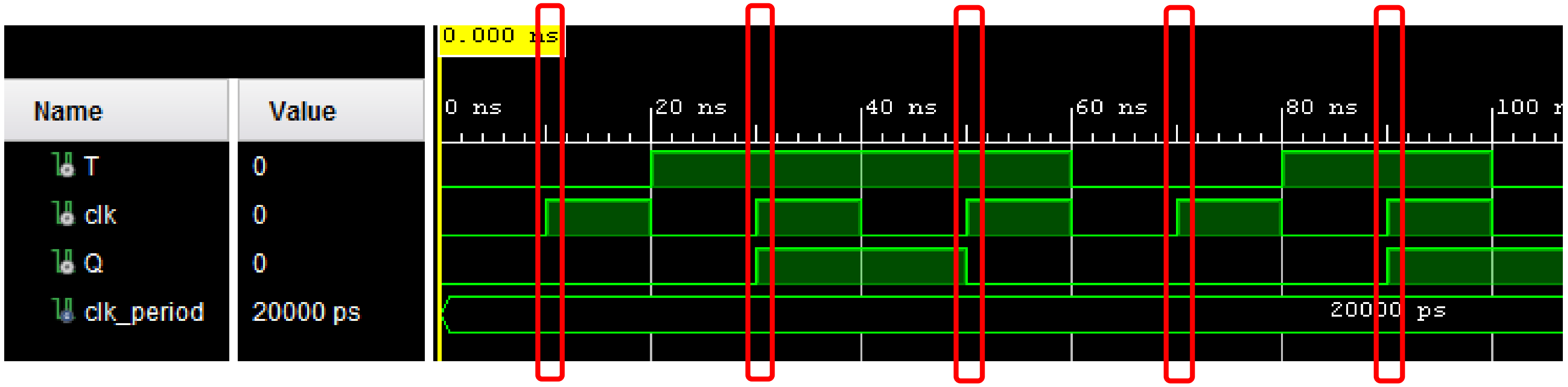
Asyn Active High Reset

C	T	Q_{t+1}	$\overline{Q_{t+1}}$	State
0	x	Q_t	$\overline{Q_t}$	Hold
1	0	Q_t	$\overline{Q_t}$	Hold
1	1	$\overline{Q_t}$	Q_t	Toggle

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

entity TFF is
port(   T: in std_logic;
        clk: in std_logic;
       areset: in std_logic;
        Q: out std_logic;
end TFF;
```

```
architecture behavioral of TFF is
signal tmp : std_logic;
begin
process(clk, areset)
begin








end process;
end behavioral;
```

Simulation



Name	Value
T	0
clk	0
Q	0
clk_period	20000 ps

Lab Session 3

Objectives

- To practice **designing combinational logic circuit**
- To reuse the materials and knowledge from previous labs
- To learn how to perform **type conversion** in VHDL Code
- To apply **bit slicing** and **bit concatenation** technique in VHDL
- To learn how to use a **logic analyzer** to trace the digital signals

Lab Session 3

- An arithmetic logic unit (ALU) is to perform arithmetic and bitwise operation on integer binary numbers
- In this experiment, you will implement a simple 3- bit ALU
- Given two 3-bit operands **A**, **B** and the 2-bit operator **op,** the result is shown in the table

op	**Result**	**carry**
00	A **ADD** B	It depends
01	A **AND** B	0
10	A **XOR** B	0
11	A **<< 1**	0

100 + 011 = 111

100 AND 111 = 100

100 XOR 111 = 011

Left shift A 011 -> 110

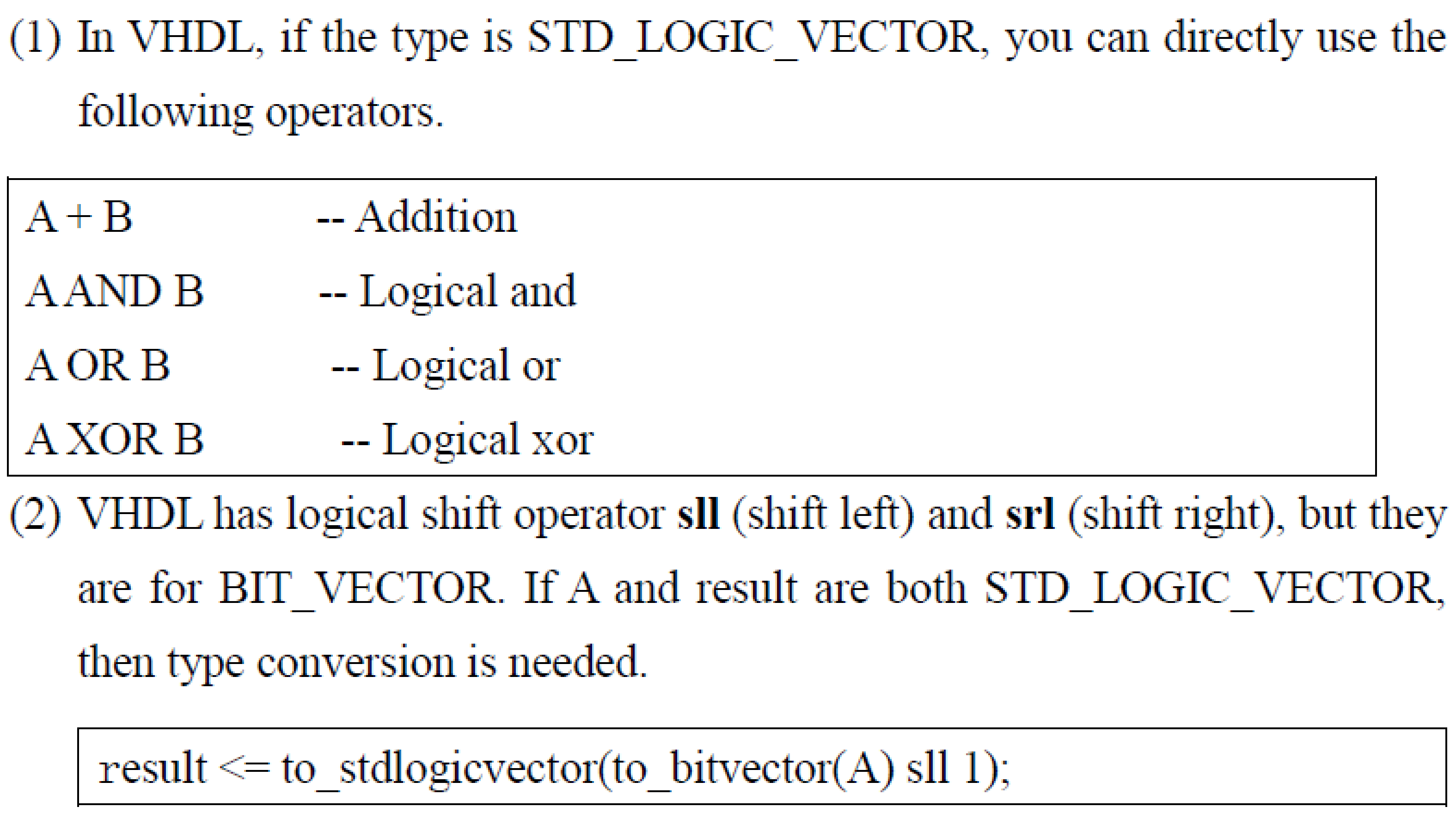
VHDL Codes

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

Entity ALU is
   Port (
      A: in STD_LOGIC_VECTOR (2 DOWNTO 0);
      B: in STD_LOGIC_VECTOR (2 DOWNTO 0);
      op: in STD_LOGIC_VECTOR (1 DOWNTO 0);
      result: out STD_LOGIC_VECTOR (2 DOWNTO 0);
      carry: out STD_LOGIC;

      --dup_result and dup_carry are used for observation using Logic Analyzer
      dup_result: out STD_LOGIC_VECTOR (2 DOWNTO 0);
      dup_carry: out STD_LOGIC
   );
end ALU;
```

VHDL Codes

```
Architecture Behavioral of ALU is
SIGNAL result_t: STD_LOGIC_VECTOR (2 DOWNTO 0);
SIGNAL carry_t: STD_LOGIC;

--You can add your declaration here.
Begin
    dup_result <= result_t;
    dup_carry <= carry_t;
    result <= result_t;
    carry <= carry_t;

    --Write your own code to implement this ALU.
end Behavioral;
```

Type Conversion



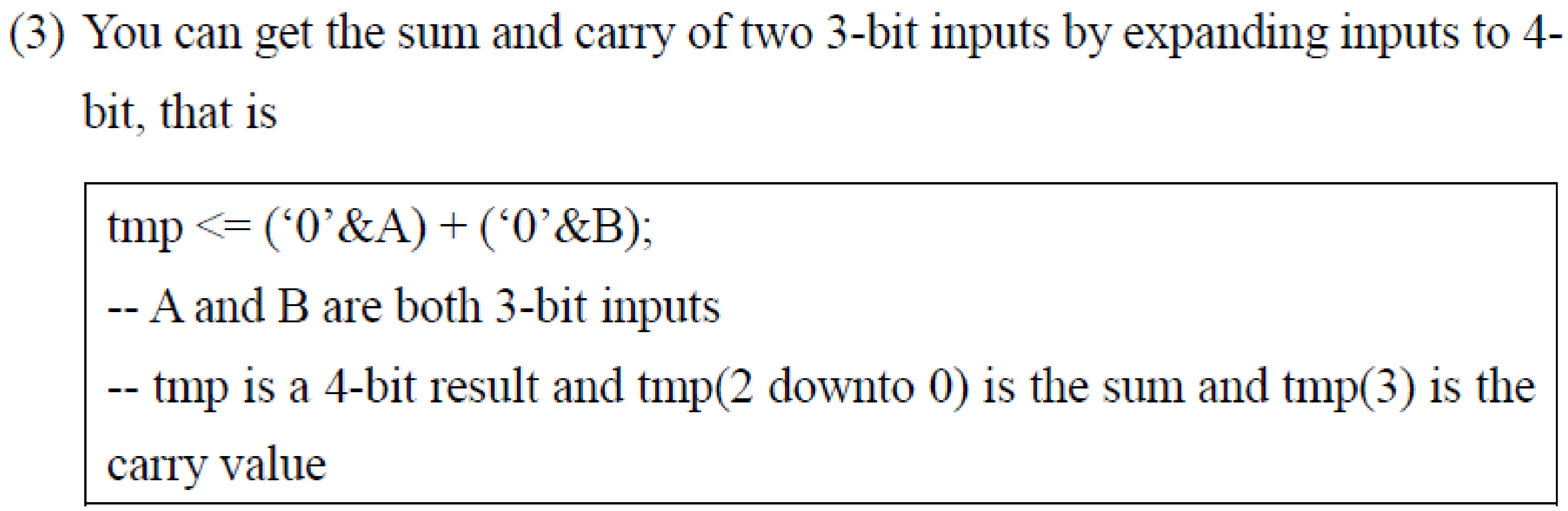
(1) In VHDL, if the type is STD_LOGIC_VECTOR, you can directly use the following operators.

```
A + B          -- Addition
A AND B        -- Logical and
A OR B         -- Logical or
A XOR B        -- Logical xor
```

(2) VHDL has logical shift operator **sll** (shift left) and **srl** (shift right), but they are for BIT_VECTOR. If A and result are both STD_LOGIC_VECTOR, then type conversion is needed.

```
result <= to_stdlogicvector(to_bitvector(A) sll 1);
```

Expand to 4 Bits



(3) You can get the sum and carry of two 3-bit inputs by expanding inputs to 4-bit, that is

```
tmp <= (‘0’&A) + (‘0’&B);
-- A and B are both 3-bit inputs
-- tmp is a 4-bit result and tmp(2 downto 0) is the sum and tmp(3) is the
carry value
```

A: 100
B: 100
Result: 1000

A: 0100
B: 0100
Result: 1000

Sum output: “000”
Carry out: “1”

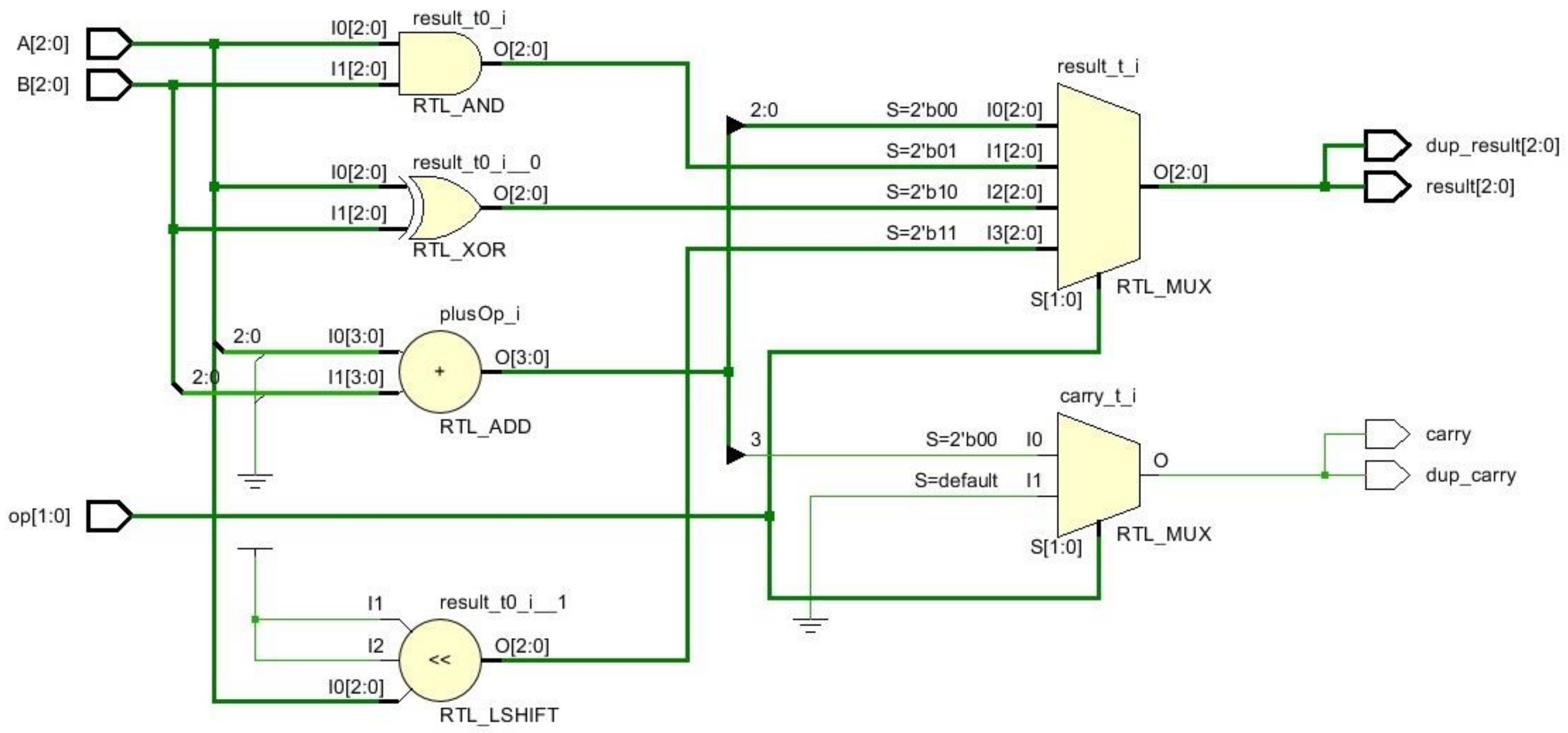
Selected Signal Assignment

op	**Result**	**carry**
00	A **ADD** B	It depends
01	A **AND** B	0
10	A **XOR** B	0
11	A **<< 1**	0

(4) Now you can learn how to use the **WHEN** statement. Here is a simple example for you. It is a simple decoder from SIGNAL sel to SIGNAL result_out. Note the use of ‘,’, and ‘;’ carefully.

```
WITH sel SELECT
      result_out <= “0001” WHEN “00”,
                    “0010” WHEN “01”,
                    “0100” WHEN “10”,
                    “1000” WHEN “11”,
                    “ZZZ” WHEN others;
```

RTL Analysis



As the design is different from one another, the schematic may not be the same

Logic Analyzer