

DIGITAL LOGIC DESIGN

VHDL Coding for FPGAs

Unit 3

✓ *BEHAVIORAL DESCRIPTION*

- *Asynchronous processes (decoder, mux, encoder, etc): if-else, case, for-loop.*
- *Arithmetic expressions inside asynchronous processes.*

✓ BEHAVIORAL DESCRIPTION (OR SEQUENTIAL)

- In this design style, the circuit is described via a series of statements (also called sequential statements) that are executed one after other; here **the order is very important**. This feature is advantageous when it comes to implement **sequential circuits**. The sequential statements must be within a block of VHDL code called *'process'*.
- The sequential code suits the description of sequential circuits very well. However, we can also describe **combinatorial circuits** with sequential statements.
- Here we will use the sequential description style to implement combinatorial circuits. In this instance, the block of VHDL code (*'process'*) is called asynchronous process.

■ ASYNCHRONOUS PROCESSES

(Implementation of combinatorial circuits with sequential statements)

Below we show the syntax of a sequential description. Note that the *'process'* statement denotes the sequential block.

```
entity example is
  port ( ...
        ... );
end example;
```

```
architecture behav of example is
begin
```

```
  process (signal_1, signal_2, ...)
begin
  ...
  ...
  ...
end process;
end behav;
```

Beginning of process block →

Sequential Statements

Sensivity list
(all the signals used inside the *process*)

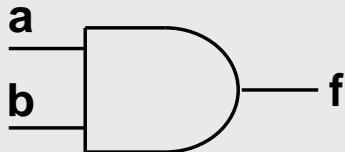
End of process block →

▪ SEQUENTIAL STATEMENTS:

▪ IF Statement: Simple Conditional

- **Example:** AND gate. The sensitivity list is made of 'a' and 'b'. We can use any other gate: OR, NOR, NAND, XOR, XNOR.
- It is a good coding practice to include all the signals used inside the process in the sensitivity list.
- Xilinx Synthesizer: DO NOT omit any signal in the sensitivity list, otherwise the Behavioral Simulation (iSIM) will be incorrect. This is usually not a problem for other Synthesizers.

```
library ieee;  
use ieee.std_logic_1164.all;  
  
entity my_and is  
  port ( a, b: in std_logic;  
         f: out std_logic);  
end my_and;
```



```
architecture behav of my_and is  
begin  
  process (a,b)  
  begin  
    if (a = '1') and (b = '1') then  
      f <= '1';  
    else  
      f <= '0';  
    end if;  
  end process;  
end behav;
```

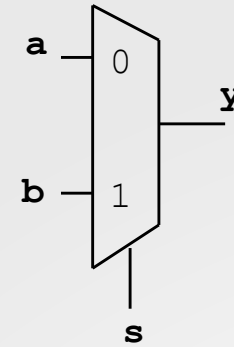
- **IF Statement:**
- **Example: 2-to-1 Multiplexor:**
Three different coding styles:

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

```
entity my_mux21 is
  port ( a, b, s: in std_logic;
        y: out std_logic);
end my_mux21;
```

```
architecture st of my_mux21 is
begin
  y <= (not(s) and a) or (s and b);
end st;
```

```
architecture st of my_mux21 is
begin
  with s select
    y <= a when '0',
        b when others;
end st;
```



$$y = \bar{s}a + sb$$

```
architecture st of my_mux21 is
begin
  process (a,b,s)
  begin
    if s = '0' then
      y <= a;
    else
      y <= b;
    end if;
  end process;
end st;
```

- **IF Statement:**
- **Example:** 4-to-1 Multiplexor

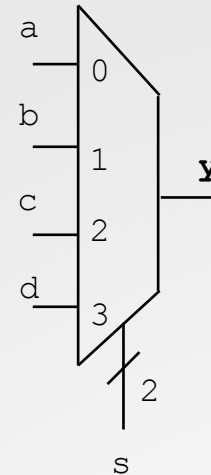
Two different styles:

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

```
entity my_mux41 is
  port ( a,b,c,d: in std_logic;
         s: in std_logic_vector (1 downto 0);
         y: out std_logic);
end my_mux41;
```

```
architecture st of my_mux41 is
begin
  with s select
    y <= a when "00",
        b when "01",
        c when "10",
        d when "11",
        '-' when others;
end st;
```

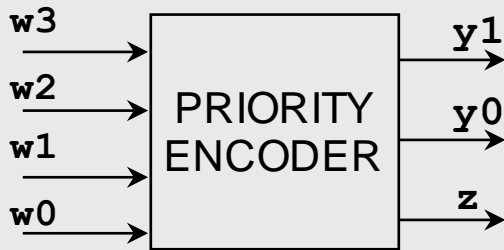
```
architecture st of my_mux41 is
begin
  process (a,b,c,d,s)
  begin
    if s = "00" then y <= a;
    elsif s = "01" then y <= b;
    elsif s = "10" then y <= c;
    else y <= d;
    end if;
  end process;
end st;
```



IF Statement

Example:

4-to-2 priority encoder



w ₃	w ₂	w ₁	w ₀	y ₁	y ₀	z
0	0	0	0	0	0	0
1	x	x	x	1	1	1
0	1	x	x	1	0	1
0	0	1	x	0	1	1
0	0	0	1	0	0	1

- The priority level is implicit by having w(3) in the first *'if'*, and w(2) in the second *'if'*, and so on.

```

library ieee;
use ieee.std_logic_1164.all;

entity my_prienc is
  port ( w: in std_logic_vector (3 downto 0);
         y: out std_logic_vector (1 downto 0);
         z: out std_logic);
end my_prienc;

architecture bhv of my_prienc is
begin
  process (w)
  begin
    if w(3) = '1' then y <= "11";
    elsif w(2) = '1' then y <= "10";
    elsif w(1) = '1' then y <= "01";
    else y <= "00";
    end if;
    if w = "0000" then
      z <= '0';
    else
      z <= '1';
    end if;
  end process;
end bhv;
  
```

▪ IF Statement

- **Example:** 4-to-2 priority encoder (another style)

- **Process:** Statements are '*executed*' (the way the synthesizer reads it) one after the other.
- The first statement assigns $y \leq "00"$. Then the value of 'y' changes ONLY if the conditions are met for the input 'w'.
- Note the order: w(1), w(2), w(3). This establishes a priority for w(3) (last statement to be executed).
- 'z' starts with '1', but if the condition is met, it is changed to '0'.

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

entity my_tprienc is
    port ( w: in std_logic_vector (3 downto 0);
          y: out std_logic_vector (1 downto 0);
          z: out std_logic);
end my_tprienc;

architecture bhv of my_tprienc is
begin
    process (w)
    begin
        y <= "00";
        if w(1) = '1' then y <= "01"; end if;
        if w(2) = '1' then y <= "10"; end if;
        if w(3) = '1' then y <= "11"; end if;

        z <= '1';
        if w = "0000" then z <= '0'; end if;
    end process;
end bhv;
```

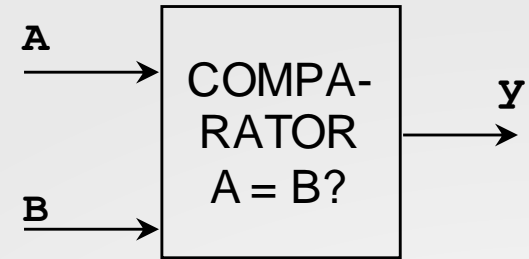
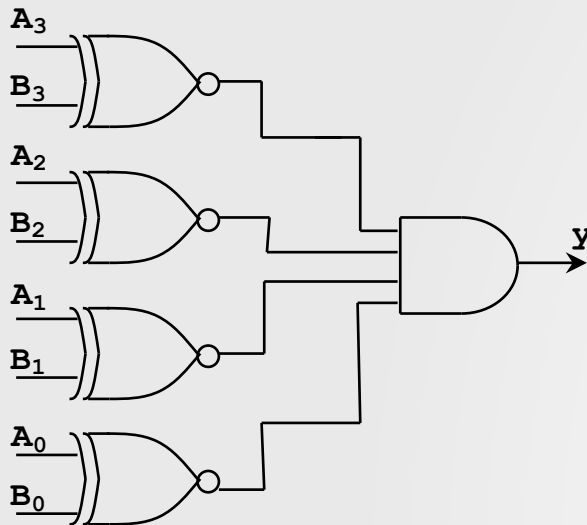

IF Statement:

Example: 4-bit comparator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all; -- unsigned #s
```

```
entity my_comp is
  port ( A,B: in std_logic_vector (3 downto 0);
        y: out std_logic);
end my_comp;
```

```
architecture struct of my_comp is
begin
  y <= '1' when A = B else '0';
end struct;
```



```
architecture behav of my_comp is
begin
  process (a,b)
  begin
    if (A = B) then
      y <= '1';
    else
      y <= '0';
    end if;
  end process;
end behav;
```

▪ IF Statement:

▪ Example of 'bad design':

4-bits comparator, but the 'else' is omitted:

Warning!

If $a \neq b \rightarrow y = ?$

Since we did not specify what happens when $a \neq b$, the synthesizer assumes that we want to keep the last value of 'y'.

In the circuit, initially 'y' will be '0'. But:

If $a = b \rightarrow y = '1'$ forever. It is said that the output has an implicit memory since it 'remembers' the previous value of y. This results in a faulty comparator.

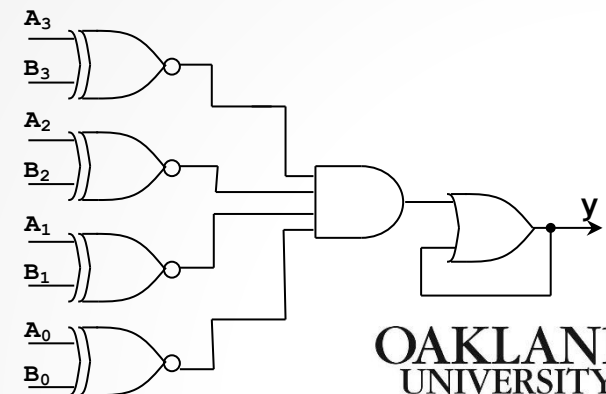
```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all; -- unsigned #s

entity my_comp is
  port ( A,B: in std_logic_vector (3 downto 0) ;
        y: out std_logic);
end my_comp;

architecture behav of my_comp is
begin
  process (a,b)
  begin
    if (A = B) then
      y <= '1';
    end if;
  end process;
end behav;
  
```

The synthesized circuit would look like this:






RULES FOR A GOOD COMBINATORIAL DESIGN USING PROCESSES

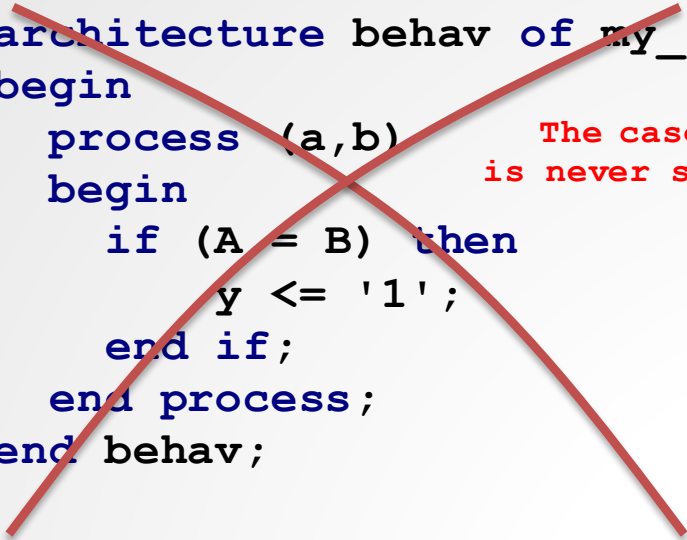
- Rule 1:** EVERY input signal that is used within the *process* must appear in the sensitivity list.
- Rule 2:** ALL the possible Input/Output combinations must be specified. Otherwise, we will find issues with implicit memory..

```
architecture behav of my_comp is
begin
  process (a,b)
  begin
    if (A = B) then
      y <= '1';
    else
      y <= '0';
    end if;
  end process;
end behav;
```



```
architecture behav of my_comp is
begin
  process (a,b)
  begin
    if (A = B) then
      y <= '1';
    end if;
  end process;
end behav;
```

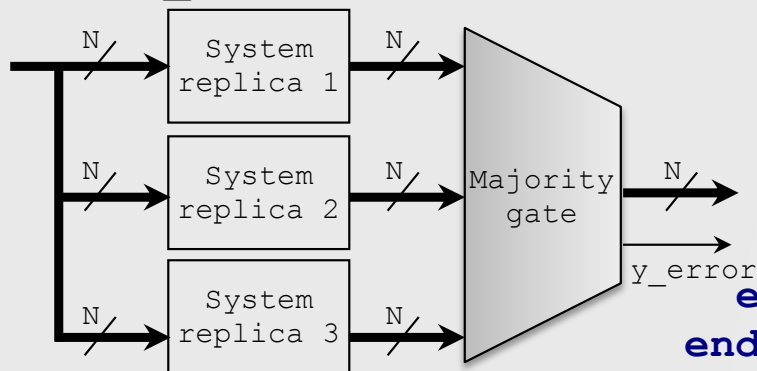
The case 'A ≠ B' is never specified



▪ IF Statement. Example: Majority gate

Triple Modular Redundancy:
 To improve reliability, a system is replicated three times. The 3 generated outputs go into a majority-voting system (majority gate) to produce a single output.

If at least two replicas produce identical outputs → the majority gate selects that output. If the three replicas produce different results, the majority gate asserts an error flag ($y_error = '1'$)



```

library ieee;
use ieee.std_logic_1164.all;

entity my_maj_gate is
  generic (N: INTEGER:= 8);
  port (A,B,C: in std_logic_vector(N-1 downto 0);
        f: out std_logic_vector(N-1 downto 0);
        y_err: out std_logic);
end my_maj_gate;

architecture bhv of my_maj_gate is
begin
  process (A,B,C)
  begin
    y_err <= '0';
    if (A = B) then f <= A; end if;
    if (A = C) then f <= A; end if;
    if (B = C) then f <= B; end if;
    if (A/=B) and (B/=C) and (A/=C) then
      f <= (others => '0');
      y_err <= '1';
    end if;
  end process;
end bhv;

```

➤ **my_maj_gate.zip**: my_maj_gate.vhd,
 tb_my_maj_gate.vhd

SEQUENTIAL STATEMENTS:

CASE statement

It is used in multi-decision cases when nested **IF's** become complex.

All possible choices must be included (see the keyword **when** for every choice of the 'selection signal')

Last case: We must use **when others** (even if all the 0/1s, as std_logic has 9 possible values). This also avoids outputs with implicit memory.

▪ **Example:** MUX 8-to-1 →

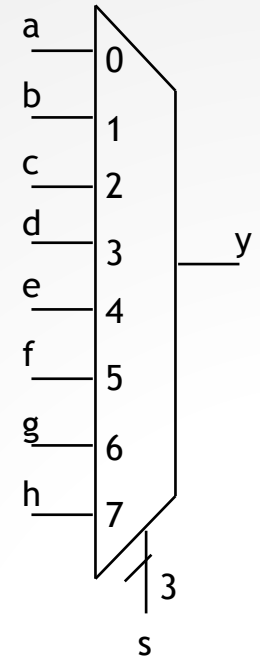
```

library ieee;
use ieee.std_logic_1164.all;

entity my_mux8to1 is
  port ( a,b,c,d,e,f,g,h: in std_logic;
        s: in std_logic_vector (2 downto 0);
        y: out std_logic);
end my_mux8to1;

architecture bhv of my_mux8to1 is
begin
  process (a,b,c,d,e,f,g,h,s)
  begin
    case s is
      when "000" => y <= a;
      when "001" => y <= b;
      when "010" => y <= c;
      when "011" => y <= d;
      when "100" => y <= e;
      when "101" => y <= f;
      when "110" => y <= g;
      when others => y <= h;
    end case;
  end process;
end bhv;

```



- **CASE Statement:**
- **Example:** MUX 7-to-1

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

- Note: $y \leq '-'$ (don't care). This allows the synthesizer to optimize the circuit.

```
entity my_mux7to1 is
  port ( a,b,c,d,e,f,g: in std_logic;
         s: in std_logic_vector (2 downto 0);
         y: out std_logic);
end my_mux7to1;
```

- If, however, we had used **when others => y <= g;** The synthesizer would have assigned the value 'g' for the cases "110" and "111" (a slightly less optimal circuit).

```
architecture bhv of my_mux7to1 is
begin
  process (a,b,c,d,e,f,g,s)
  begin
    case s is
      when "000" => y <= a;
      when "001" => y <= b;
      when "010" => y <= c;
      when "011" => y <= d;
      when "100" => y <= e;
      when "101" => y <= f;
      when "110" => y <= g;
      when "111" => y <= g;
      when others => y <= '-';
    end case;
  end process;
end bhv;
```

■ CASE Statement:

■ Example:

Binary to gray decoder

- It could also be described using the 'with-select' statement (no *process*)

$b_2b_1b_0$	$g_2g_1g_0$
0 0 0	0 0 0
0 0 1	0 0 1
0 1 0	0 1 1
0 1 1	0 1 0
1 0 0	1 1 0
1 0 1	1 1 1
1 1 0	1 0 1
1 1 1	1 0 0

```

library ieee;
use ieee.std_logic_1164.all;

entity my_gray2bin is
  port ( B: in std_logic_vector(2 downto 0);
        G: in std_logic_vector(2 downto 0));
end my_gray2bin;

architecture bhv of my_gray2bin is
begin
  process (B)
  begin
    case B is
      when "000" => G <= "000";
      when "001" => G <= "001";
      when "010" => G <= "011";
      when "011" => G <= "010";
      when "100" => G <= "110";
      when "101" => G <= "111";
      when "110" => G <= "101";
      when others => G <= "100";
    end case;
  end process;
end bhv;

```


■ CASE statement

■ Example:

7-segment decoder.

```

library ieee;
use ieee.std_logic_1164.all;

entity my_7segdec is
  port ( bcd: in std_logic_vector(3 downto 0);
        leds: out std_logic_vector(6 downto 0));
end my_7segdec;

architecture bhv of my_7segdec is
begin
  process (bcd)
  begin
    case bcd is
      when "0000" => leds <= "1111110";
      when "0001" => leds <= "0110000";
      when "0010" => leds <= "1101101";
      when "0011" => leds <= "1111001";
      when "0100" => leds <= "0110011";
      when "0101" => leds <= "1011011";
      when "0110" => leds <= "1011111";
      when "0111" => leds <= "1110000";
      when "1000" => leds <= "1111111";
      when "1001" => leds <= "1111011";
      when others => leds <= "-----";
    end case;
  end process;
end bhv;

```

■ We use the don't care value ('-') to optimize the circuit, since we only expect inputs from "0000" to "1111".

■ Note that the CASE statement avoids the output with implicit memory, since the **when others** clause makes sure that the remaining cases are assigned.

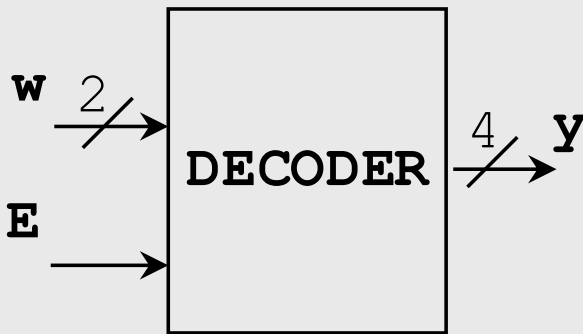
▪ CASE Statement:

▪ Example:

2-to-4 decoder with enable.

▪ Note how we combine IF with CASE for this decoder with enable.

▪ The **else** cannot be omitted, otherwise the output will have implicit memory (it will be a LATCH)



Example: 2-to-4 decoder (3 styles):

- **mydec2to4.zip:**
- mydec2to4.vhd,
- tb_mydec2to4.vhd,
- mydec2to4.ucf

```

library ieee;
use ieee.std_logic_1164.all;

entity my_dec2to4 is
  port ( w: in std_logic_vector(1 downto 0);
        y: out std_logic_vector(3 downto 0);
        E: in std_logic);
end my_dec2to4;

architecture bhv of my_dec2to4 is
begin
  process (w,E)
  begin
    if E = '1' then
      case w is
        when "00" => y <= "0001";
        when "01" => y <= "0010";
        when "10" => y <= "0100";
        when others => y <= "1000";
      end case;
    else y <= "0000";
    end if;
  end process;
end bhv;
  
```

FOR-LOOP statement

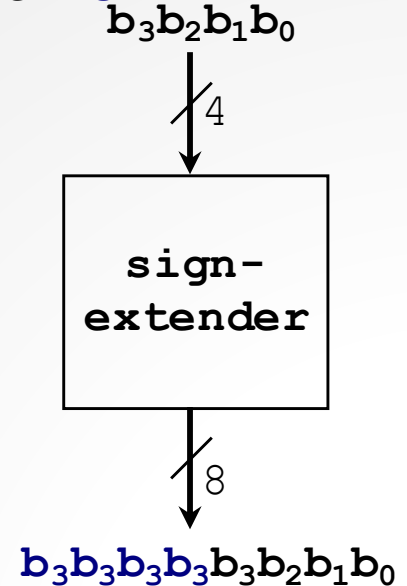
- Very useful for sequential circuit description. But, it can also be used to describe some combinatorial circuits.

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

```
entity my_signext is
  port ( A: in std_logic_vector(3 downto 0);
        y: out std_logic_vector(7 downto 0) );
end my_signext;
```

- Example:** Sign-extension (from 4 bits to 8 bits)

```
architecture bhv of my_signext is
begin
  process (A)
  begin
    y(3 downto 0) <= A;
    for i in 7 downto 4 loop
      y(i) <= A(3);
    end loop;
  end process;
end bhv;
```



FOR-LOOP statement

- Example:** Ones/zeros detector: It detects whether the input contains only 0's or only 1's.

- Input length: Parameter 'N'.
- This is a rare instance where using *process* for combinational circuits is the most efficient description.

- Variable** inside a process: it helps to describe this circuit. Depending on the implementation, a 'variable' could be a wire.

```

library ieee;
use ieee.std_logic_1164.all;

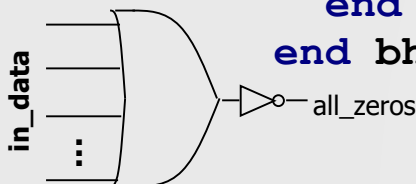
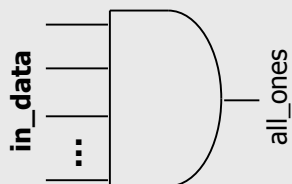
entity zeros_ones_det is
  generic (N: INTEGER:= 8);
  port (in_data: in std_logic_vector(N-1 downto 0);
        all_zeros, all_ones: out std_logic);
end zeros_ones_det;

architecture bhv of zeros_ones_det is
begin
  process(in_data)
    variable result_and, result_or: std_logic;
  begin
    result_and:= '1'; result_or:= '0';
    for i in in_data'range loop
      result_and:= result_and and in_data(i);
      result_or:= result_or or in_data(i);
    end loop;
    all_zeros <= not(result_or);
    all_ones <= result_and;
  end process;
end bhv;

```

zeros_ones_detector.zip:

zeros_ones_detector.vhd,
 tb_zeros_ones_detector.vhd,
 zeros_ones_detector.ucf



■ ARITHMETIC EXPRESSIONS

- We can use the operators $+$, $-$, and $*$ inside behavioral processes. We can also use the comparison statements ($>$, $<$, $=$, \neq , \geq , \leq).
- **Example:** Absolute value of $A-B$. A, B : treated as unsigned integers.

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

- Input length:
Parameter N .
- Signed numbers:
As the result requires an extra bit, you must first sign-extend the operands A and B .

```
entity my_uabs_diff is
  generic (N: INTEGER:= 4);
  port ( A,B: in std_logic_vector(N-1 downto 0);
        R: out std_logic_vector(N-1 downto 0));
end my_uabs_diff;
```

```
architecture bhv of my_uabs_diff is
begin
  process (A,B)
  begin
    if A >= B then
      R <= A - B;
    else
      R <= B - A;
    end if;
  end process;
end bhv;
```

➤ my_uabs_diff.zip:
my_uabs_diff.vhd,
tb_my_uabs_diff.vhd