Comparator

Task:

- Write out Boolean equations for the outputs of a simple 1-bit comparator (Table 1). Use these equations to describe the comparator in VHDL.
- Write out the truth table and Boolean equations for a 2-bit comparator. Use these equations to describe the comparator in VHDL.
- Use "when .. else" VHDL statement to describe a 2-bit comparator.

Table 1: Comparator Truth Table

in1	in2	eq_o	gr_o	ls_o
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

Perform functional simulation to verify correctness of the received VHDL descriptions. Implement and test the designs on FPGA development board. Compare implementation results of both 2-bit comparators: in RTL Analysis and Synthesis schematics, Synthesis and Implementation reports, etc. How do they differ from each other?

Half Adder Example

A simple 1-bit half adder circuit has two inputs and two outputs (Figure 1).

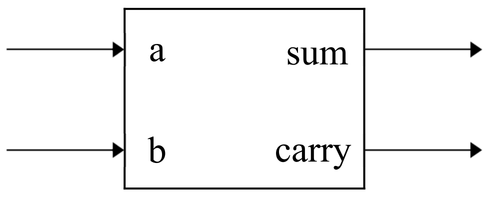


Figure 1: Half Adder Block Diagram

Table 2: Half Adder Truth Table

a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The functionality of the half adder can be specified with the truth table (Table 2). It provides enough information to write out the Boolean equations (1) and (2) for the outputs.

$$sum = (\neg a) \cdot b + a \cdot (\neg b) \quad (1) \qquad carry = a \cdot b \quad (2)$$

VHDL code, which is based on the above specification, is provided in Listing 1. Code consists of two main parts: *entity* and *architecture*. *Entity* represents the interface of the circuit, while *architecture* describes either internal structure or behavior.

Listing 1: VHDL Description of a Half Adder

```
entity half_adder is
        port (a, b: in std_logic;
              sum, carry: out std_logic);
end half_adder;

architecture half_adder_arch of half_adder is
        begin
        sum <= (not a and b) or (a and not b);
        carry <= a and b;
end  half_adder_arch;
```

Entity represents the external view of the half adder, just like in the block diagram (Figure 1). Note, that inputs and outputs are of *std_logic* type, as it closely resembles the values a signal may have in the actual circuit. There is no need to declare this type, as the skeleton code generated by Vivado does so by default.

Architecture body consists of two concurrent signal assignment statements. These assignments correspond to Boolean equations (1) and (2), and are used to compute the output. However, **and**, **or** and **not** keywords in these statements are VHDL logic operators, and do not represent actual gates. Physical implementation may be synthesized using different logic gates or logic elements after undergoing optimization.

Alternatively, combinational logic can be described in a more abstract way using general conditional VHDL statements. For example, consider a simple 2-bit two-to-one multiplexer (Figure 2). I has two 2-bit data inputs *a* and *b*, control input *sel* and data output *o*. When *sel* is equal to logic 1, data input *a* is propagated to the data output *o*. Similarly, when *sel* is equal to logic 0, data input *b* is propagated to the data output *o*. Such behavior can be described without devising neither the truth table, nor Boolean equations.

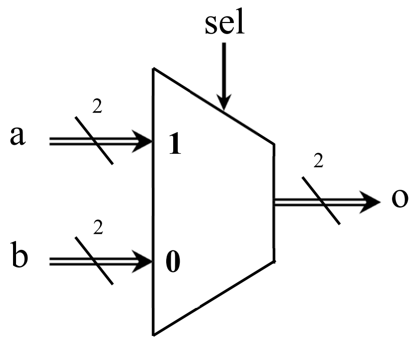


Figure 2: 2-bit 2-to-1 Multiplexer Block Diagram

VHDL code, which is based on the above specification, is provided in Listing 2. In order to describe a 2-bit two-to-one multiplexer in VHDL, "**when** .. **else**" conditional statement can be used. Data inputs/output *a*, *b* and *o* are now of *std_logic_vector* type (from 1 downto 0), since their size has been defined as being 2 bits wide. Note, that this description would generally invoke an actual multiplexer during synthesis.

Listing 2: VHDL Description of a 2-bit 2-to-1 Multiplexer

```
entity mux is
        port (a, b: in std_logic_vector(1 downto 0);
              sel: in std_logic;
              o: out std_logic_vector(1 downto 0));
end mux;

architecture mux_arch of mux is
begin
  o <= a when (sel = '1') else b;
end  mux_arch;
```

Functional Simulation

In order to test the functionality of the half adder, it should be simulated with a test bench. Test bench is a VHDL code, which applies stimulus to design entity during simulation. An example of the stimulus for "00" input combination is presented in Listing 3. First of all, inputs are assigned a test value. After a 20 ns wait period, the outputs are compared to the expected values using **assert** statement (both *sum* and *carry* outputs should be equal to '0'). An error message is reported if they do not match. The type of message is specified using **severity** statement. In Listing 3 the severity level is set to *error* (the default setting in case

severity statement is omitted). Other possibilities include *note*, *warning* and *failure* (in case of *failure* the simulation will stop immediately). The exact message to be displayed can be set using **report** statement.

Listing 3: Stimulus for "00" Input Combination

```
a <= '0'; b <= '0';
wait for 20 ns;
assert (sum = '0') and (carry = '0') report "test failed for 00" severity error;
```

Create a new source file and choose *Add or create simulation sources* option. Leave *I/O Port Definitions* in the *Define Module* window empty since testbench does not have any ports. The created testbench source file can be found in the *Sources* window under *Simulation Sources* category in the *Project Manager* flow section layout.

Double-click the testbench file to open it with the *Text Editor*. The skeleton code generated by Vivado does not contain any necessary declarations and instantiations, so they should be added manually. Listing 4 provides a skeleton testbench architecture description for the half adder design.

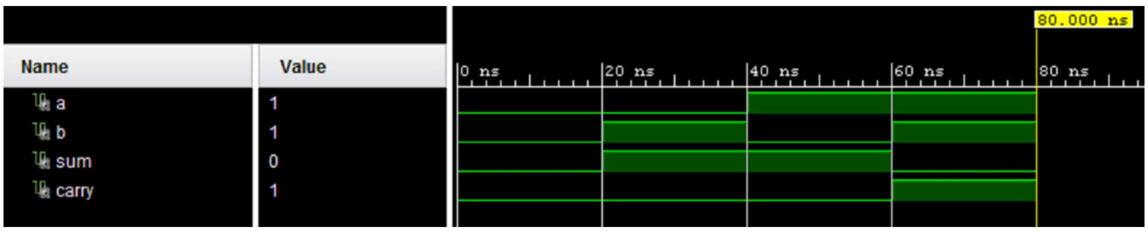
The declarative part of the architecture consists of the unit under test (UUT) component declaration (half adder design) and the declaration of signals that will be mapped to the inputs/outputs of the UUT. Note, that inputs and outputs are declared separately, since inputs are provided with an initial value. Initialization is not needed for the signals that represent outputs since their value depend directly on the value of signals that represent inputs.

The architecture body consists of the UUT component instantiation and the stimuli process. The component instantiation maps ports of the UUT component (on the left) to the signals that are declared in the testbench architecture (to the right). The stimuli process should be altered by adding the code to test all possible input combinations. Note, that the **wait** statement is placed at the end of the process in order to stop the repeating generation of the stimuli (since process loops back to the beginning when **end** statement is reached).

Listing 4: Skeleton Architecture Description for Half Adder Testbench

```
architecture Behavioral of half_adder_tb is

component half_adder is
Port ( a : in STD_LOGIC;
        b : in STD_LOGIC;
        sum : out STD_LOGIC;
        carry : out STD_LOGIC);
end component half_adder;

signal a, b : std_logic := '0';      -- signals for inputs
signal sum, carry : std_logic;   -- signals for outputs

begin

UUT : half_adder
Port map ( a => a,
           b => b,
           sum => sum,
           carry => carry );

stimuli : process
begin
-- insert stimuli here
wait;
end process;

end Behavioral;
```

When testbench is ready, set it as the top simulation source if this hasn't been done automatically by the tool (or in order to set a different simulation source file as top). Select *Run Simulation* option in the *Flow Navigator* under *Simulation* flow section, click *Run Behavioral Simulation*. The simulation result should be similar to the one presented in Figure 5. The simulation result fully corresponds to the truth table of half adder (Table 2).



Name	Value
a	1
b	1
sum	0
carry	1

Figure 5: Half Adder Simulation Waveform