

**POLITECNICO  
DI TORINO**

## – Elettronica dei Sistemi Digitali – Lab#1

# Multiplexers, Light Emitting Diodes, Switches and Testbench

The purpose of this laboratory session is to learn how to connect simple input and output devices to an FPGA chip and implement a circuit that uses these devices for a specific target. We will use the switches *SW17-0* on the DE2 board as inputs and we will use light emitting diodes (LEDs) as outputs of our logic circuit.

### Contents:

1. Controlling the LEDs
2. 2-to-1 Multiplexer
3. 5-to-1 Multiplexer
4. Testing your circuit with a testbench

### Abbreviations and acronyms:

IC – Integrated Circuit

LED – Light Emitting Diode

MUX – Multiplexer

VHDL – Very high speed integrated circuits Hardware Description Language

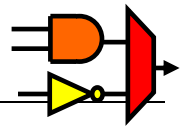
[VHDL cookbook: <http://www.onlinefreebooks.net/engineering-ebooks/electrical-engineering/the-vhdl-cookbook-pdf.html>]

**USE EXACTLY THE SAME I/O PINS SPECIFIED IN THIS DOCUMENT.  
YOU ARE NOT ALLOWED TO USE “IF”, “SELECT”, “WHEN” OR “CASE” STATEMENTS IN ALL THE  
EXERCISES.**

## 1 – Controlling the LEDs

The DE2 board provides 18 toggle switches, called *SW17-0* that can be used as circuit inputs, and 18 red lights, called *LEDR17-0*, that can be used to display output values. Figure 1 shows a simple VHDL entity that uses these switches and shows their states on the LEDs. As specified in the code, since there are 18 switches and 18 LEDs, it is convenient to represent them as arrays. Here we have used a single assignment statement for all 18 *LEDR* outputs, which is equivalent to the following individual assignments:

```
LEDR(17) <= SW(17);
LEDR(16) <= SW(16);
. . .
```



```
LEDR(0) <= SW(0);
```

The DE2 board has hardwired connections between the FPGA chip, the switches and the LEDs. To use  $SW_{17-0}$  and  $LEDR_{17-0}$  it is necessary to include in your Quartus II project the correct pin assignments which are given in the *DE2 User Manual*. For example, the manual specifies that  $SW_0$  is connected to the FPGA pin  $N25$  and  $LEDR_0$  is connected to pin  $AE23$ . A good way to make the required pin assignments is to import into the Quartus II software the file called *de2\_pin\_assignments.csv*, which is provided on the web site of the course. The procedure for making a pin assignment is described in the first lab session LAB#0...

It is important to realize that the pin assignments in the *DE2\_pin\_assignments.csv* file are useful only if the pin names given in the file are exactly the same as the port names used in your VHDL entity. The file uses the names  $SW[0] \dots SW[17]$  and  $LEDR[0] \dots LEDR[17]$  for the switches and lights. This is the reason why we used these names in Figure 1 (note that the Quartus II software uses [ ] square brackets for array elements, while the VHDL syntax uses ( ) round brackets).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- Simple module that connects
-- the SW switches to the LEDR lights

ENTITY part1 IS
    PORT ( SW : IN STD_LOGIC_VECTOR(17 DOWNT0 0);
          LEDR : OUT STD_LOGIC_VECTOR(17 DOWNT0 0)); -- red LEDs
END part1;

ARCHITECTURE Behavior OF part1 IS
BEGIN
    LEDR <= SW;
END Behavior;
```

Figure 1 - VHDL code that uses the switches and LEDs on the DE2 board.

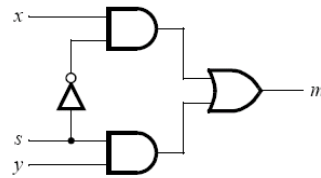
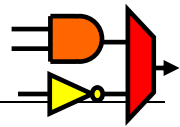
You need to do steps and implement a circuit corresponding to the code in Figure 1 on the DE2 board.

1. **Create a new Quartus II project** for your circuit. Select Cyclone II EP2C35F672C6 as the target chip, which is the FPGA chip on the Altera DE2 board.
2. **Create a VHDL entity** for the code in Figure 1 and include it in your project.
3. Include in your project the required **pin assignments** for the DE2 board, as discussed above.
4. **Compile** the project.
5. **Download** the compiled circuit in the FPGA chip.
6. **Test** the functionality of the circuit by toggling the switches and looking at the LEDs.

**YOU MUST DEMONSTRATE THAT YOU UNDERSTOOD THE CODE  
AND YOU TESTED IT ON THE FPGA. PLEASE REFER TO THE LAB#0 FILE FOR DETAILS ON  
PROGRAMMING THE FPGA**

## 2 - 2-to-1 Multiplexer

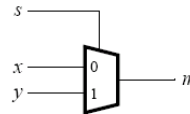
Figure 2a shows a sum-of-products circuit that implements a 2-to-1 *multiplexer* with a select input  $s$ . If  $s = 0$  the multiplexer's output  $m$  is equal to the input  $x$ , and if  $s = 1$  the output is equal to  $y$ . Part b of the figure gives a truth table of this MUX, and part c shows its circuit symbol.



a) Circuit

s	m
0	x
1	y

b) Truth table



c) Symbol

Figure 2 - A 2-to-1 multiplexer.

The multiplexer can be described with the following VHDL statement:

```
m <= (NOT (s) AND x) OR (s AND y);
```

You need to write a VHDL entity that includes eight assignment statements like the ones shown above to describe the circuit given in Figure 3a. This circuit has two eight-bit inputs,  $X$  and  $Y$ , and produces the eight-bit output  $M$ .

If  $s = 0$  then  $M = X$ , while if  $s = 1$  then  $M = Y$ . We refer to this circuit as an eight-bit wide 2-to-1 multiplexer. The circuit symbol is shown in Figure 3b. The signals  $X$ ,  $Y$  and  $M$  are depicted as eight-bit wires. You need to do the steps shown below.

1. **Create a new Quartus II project** for your circuit.
2. **Write and include your VHDL file** for the eight-bit wide 2-to-1 multiplexer in your project. Use switch  $SW_{17}$  on the DE2 board as the  $s$  input, switches  $SW_{7-0}$  as the  $X$  input and  $SW_{15-8}$  as the  $Y$  input. Connect the  $SW$  switches to the red lights  $LEDR$  and connect the output  $M$  to the green lights  $LEDG_{7-0}$ .
3. Include in your project the required **pin assignments** for the DE2 board. As discussed in Part I, these assignments ensure that the input ports of your VHDL code use the pins on the Cyclone II FPGA that are connected to the  $SW$  switches. Moreover ensure that the output ports of your VHDL code use the FPGA pins connected to the  $LEDR$  and  $LEDG$  lights.
4. **Compile** the project.
5. **Download** the compiled circuit into the FPGA chip.
6. **Test** the functionality of the eight-bit wide 2-to-1 multiplexer by toggling the switches and observing the LEDs.

1. **YOU NEED TO WRITE THE VHDL CODE OF THE 8 BIT 2-TO-1 MULTIPLEXER.**
2. **YOU NEED TO TEST THE MULTIPLEXER ON THE FPGA.**

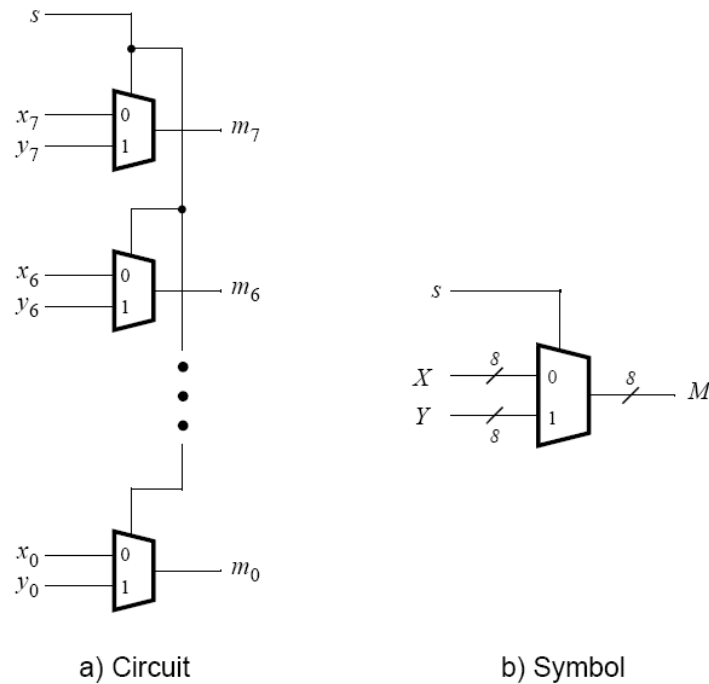
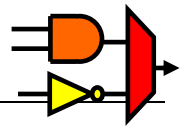
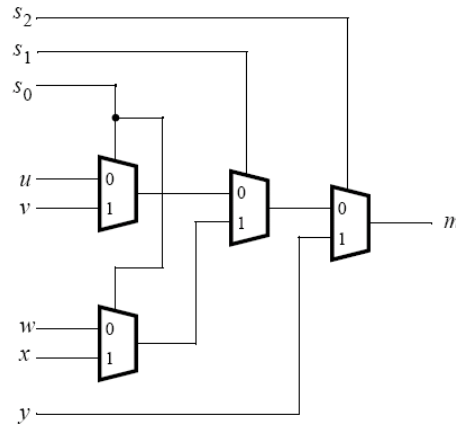
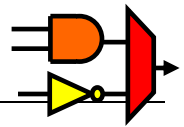


Figure 3 - An eight-bit wide 2-to-1 multiplexer.

### 3 – 5-to-1 Multiplexer

Figure 2 shows a 2-to-1 multiplexer that redirections the two inputs  $x$  and  $y$ . Now, consider a circuit in which the output  $m$  has to be selected from five inputs  $u$ ,  $v$ ,  $w$ ,  $x$ , and  $y$ . Part *a* of Figure 4 shows how we can make the required 5-to-1 multiplexer by using four 2-to-1 multiplexers. The circuit uses a 3-bit select input  $s_2 s_1 s_0$  and implements the truth table shown in Figure 4*b*. A circuit symbol for this multiplexer is given in part *c* of the figure.

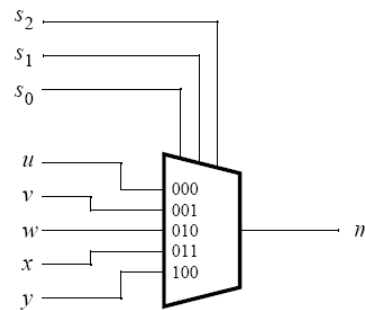
Figure 3 reminds you that an eight-bit wide 2-to-1 multiplexer can be built out of eight instances of a 2 to-1 multiplexer. Figure 5 applies this concept to define a three-bit wide 5-to-1 multiplexer. It contains three instances of the circuit in Figure 4*a*.



a) Circuit

$s_2$	$s_1$	$s_0$	$m$
0	0	0	$u$
0	0	1	$v$
0	1	0	$w$
0	1	1	$x$
1	0	0	$y$
1	0	1	$y$
1	1	0	$y$
1	1	1	$y$

b) Truth table



c) Symbol

Figure 4 - A 5-to-1 multiplexer.

You need to do the following steps and implement the 3-bit wide 5-to-1 multiplexer.

1. **Create a new Quartus II project** for your circuit.
2. **Create a VHDL entity** for the three-bit wide 5-to-1 multiplexer. Connect its select inputs to switches  $SW_{17-15}$ , and use the remaining 15 switches  $SW_{14-0}$  to provide the five 3-bit inputs  $U$  to  $Y$ . Connect the  $SW$  switches to the red lights  $LEDR$  and connect the output  $M$  to the green lights  $LEDG_{2-0}$ .
3. Include in your project the required **pin assignments** for the DE2 board. Compile the project.
4. **Download** the compiled circuit in the FPGA IC.
5. **Test** the functionality of the three-bit wide 5-to-1 multiplexer by toggling the switches and observing the LEDs. Ensure that each input from  $U$  to  $Y$  can be properly multiplexed on the output  $M$ .

**1. YOU NEED TO WRITE THE VHDL CODE OF THE 3 BIT 5-TO-1 MULTIPLEXER.**  
**2. YOU NEED TO TEST THE MULTIPLEXER ON THE FPGA.**

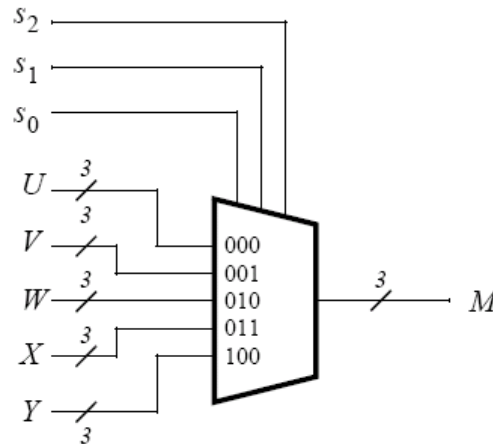
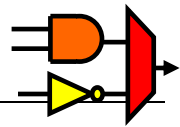


Figure 5 - A three-bit wide 5-to-1 multiplexer.

## 4 – Testing your circuit with a testbench

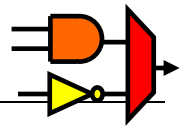
**READ THIS VERY CAREFULLY. TESTBENCH IS AN IMPORTANT TOPIC.**

In the next lab sessions you will be asked to test the VHDL units on the Altera DE2 board. In fact, this board and the FPGA installed on it represent a testing environment for your design. This is actually a physical (“hardware”) environment in which you can really test what is going on with your blocks and measure the output of your circuit. It is mainly used in production, where engineers need a fast plug-and-play platform to test a product in series. This custom collection of devices that support the target system is called *testbed*.

Sometimes, it is not possible to verify the correctness of a design on a physical board for many reasons: Time, complexity, price, possible combinations of the inputs values and so on. Should you be asked to develop a single block of a complex system, i.e. an Arithmetic Logic Unit (ALU), a rapid and flexible way to test it is with some kind of virtual environment, in which you can generate stimuli that stay in the “software” domain, just as your VHDL blocks. You have already seen it when we introduced the Waveform Vector files. Graphically, one or a set of logic signals could be forced with a specific timing and verify if the output of the circuit is correct. Sometimes entering the timing and values specifications for each input is tedious and takes time. To avoid this, it is possible to write a *wrapper* to your design unit that, by using the same features of your development software and the same language, can generate all the inputs to your system required for the verification. In addition, it can collect also the output of your circuit and automatically check if some output is wrong, for example by comparing it to a file. This kind of environment is called *testbench*.

Normally, a testbench is written in the same language of the unit you are developing, in our case VHDL, but for other kinds of systems, for example mixed signal, it can also be written in VHDL-AMS, Spice, Verilog-A or even TCL. Imagine you have an 8-bit multiplexer 16-to-1: Testing the functionality of the circuit by using switches or buttons would take forever. Instead, by using a VHDL process that sequentially generates, nanosecond after nanosecond, all the possible combinations of the inputs, would save time and would take advantage of the speed of the simulation software. It can be used both for functional and timing simulations.

A testbench must not necessarily be synthesizable, because its aim is to virtually generate the input values to your system, while it should take advantage of all the advanced functions of the language. Generally a testbench is written as a VHDL entity with no input nor outputs, and it instantiates the components under test (also known as DUT, Device Under Test) in the architecture.



Unfortunately, the simulator embedded in the Quartus-II software cannot handle a testbench because a testbench is not synthesizable and generally don't have a physical connotation. The testbench was available also in the Quartus-II simulator but it was the Waveform Vector file that could set inputs only graphically by rising and lowering signals at specified times. Here it is required another tool, specifically designed to simulate VHDL, that is called Modelsim. This tool has been developed by Mentor Graphics, but there exists also a customized Altera version that we are going to use. The **Modelsim-Altera starter edition version 6.5b** (specific for Quartus-II 9.1 sp2) is available at the following web page: <https://www.altera.com/servlets/download3?swcode=WWW-SWD-MDS-ASE-65B-91SP2-PC>. You can download it in the same way you did for Quartus-II and install it to the same installation directory as an add-on, for example in `c:\altera\91` if you previously installed Quartus-II there.

For this functional simulation, you don't need any modifications to your code when you decide to create a testbench. You just have to instantiate your design from your top-level hierarchy and start writing a wrapper. Figure 6 shows the conceptual block scheme, an example of VHDL files hierarchy. As you can see, your design is instantiated unmodified while the testbench generates inputs to your design and collects outputs, thus it has no port declaration in its entity. Make sure you do not synthesize the testbench, because it has no physical counterpart!

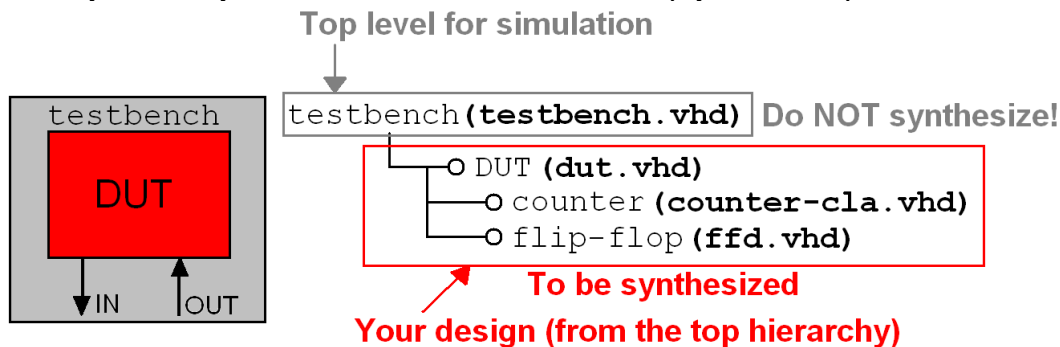


Figure 6 – Testbench hierarchy

Let us focus now on an example, a testbench that verifies the operation of the circuit designed in the previous exercise, the 3bit 5-to-1 multiplexer. For the sake of convenience, create another directory for example *es1tbsim* where you will copy the entity and architecture of your 3bit 5-to-1 multiplexer. If your design has more than one file, copy all the VHDL source of the required sub-unit in that directory. Make sure that for this version you name the multiplexer as *b35to1MUX* and have a port declaration like:

```
ENTITY b35to1MUX IS
PORT (SW: IN STD_LOGIC_VECTOR (17 DOWNTO 0);
        LEDR: OUT STD_LOGIC_VECTOR (17 DOWNTO 0);
        LEDG: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END b35to1MUX;
```

This is your DUT. Here all the inputs have been “packed” in the input SW declaration. Since the testbench instantiates the component and is specifically designed for this port map, it is necessary that this is exact. Next times, you will design the testbench and so you will decide the name and the port format of the DUT.

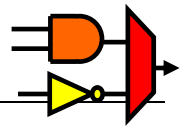
Create a new text file in the same directory with name *testbench.vhd* and cut and paste there the following code. The filename shouldn't necessarily have the same name of the entity it contains.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- "Empty" entity.
ENTITY testbench_es1 IS
END testbench_es1;

ARCHITECTURE Behavior OF testbench_es1 IS

-- DEVICE UNDER TEST
```



```

COMPONENT b35to1MUX
PORT (SW: IN STD_LOGIC_VECTOR (17 DOWNTO 0);
        LEDR: OUT STD_LOGIC_VECTOR (17 DOWNTO 0);
        LEDG: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END COMPONENT;

-- TESTBENCH SIGNALS (WRAPPING UP THE DUT)

SIGNAL U, V, W, X, Y, M: STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL S: STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL LEDS: STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL INPUTS: STD_LOGIC_VECTOR(17 DOWNTO 0);

BEGIN

-- These signals emulate
-- the switches SW14-0.
-- In this example they are FIXED.
U <= "010";
V <= "101";
W <= "111";
X <= "000";
Y <= "011";

-- This emulates selection signals
-- s0, s1 and s2.
-- Control signals are dynamically changed every 20 ns.
-- This PROCESS is kind of a "clock":
-- After the last "WAIT FOR" it loops at the beginning.

-- The instruction WAIT FOR <time> IS
-- ABSOLUTELY NOT SYNTHESIZABLE AND HAS NO
-- PHYSICAL COUNTERPART.
-- IT IS NOT A FLIP-FLOP OR A LOGIC ELEMENT!

PROCESS
BEGIN
    S <= "000";
    WAIT FOR 20 ns;
    S <= "001";
    WAIT FOR 20 ns;
    S <= "010";
    WAIT FOR 20 ns;
    S <= "011";
    WAIT FOR 20 ns;
    S <= "100";
    WAIT FOR 20 ns;
END PROCESS;

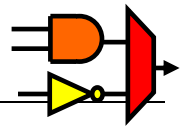
-- This is to align the inputs S...Y in the form allowed by the component.
INPUTS <= S & U & V & W & X & Y;

-- DEVICE UNDER TEST.
MUX_UT: b35to1MUX PORT MAP (INPUTS, LEDS, M);
-----

-- M is the output of the circuit.

```





```
END Behavior;
```

Save the file. This is an example of a possible testbench. As you can see, it is a wrapper that contains one or more component instantiations. The entity has no inputs and no outputs and, in this case, the architecture places the component *b35to1MUX* (red) and generates the input signals (blue). In particular, inputs U,V,W,X and Y are fixed, while control signal *s* changes with time. The process with no sensitivity list emulates the sequence “000”, “001”, “010”, “011” and “100” with steps of 20 ns. The sequence is periodic since after the last `WAIT FOR`, the process loops at the beginning. The `WAIT FOR` statement can be very useful to generate a clock. For example, by defining a standard logic signal `clk` in the architecture you can emulate a 50% Duty Cycle clock by using the following code:

```
PROCESS
BEGIN
    CLK <= "0";
    WAIT FOR 1 ns;
    CLK <= "1";
    WAIT FOR 1 ns;
END PROCESS;
```

Here, the generated clock has 500MHz frequency. Keep it in mind for the next exercises.

By simulating the top level *testbench\_es1*, by selecting all the internal testbench signals *S*, *U*, *V*, *W*, *X*, *Y* and *M* it will be possible to run a complete simulation of *MUX\_UT* and check if output *M* corresponds to the correct input selection. It is possible to integrate Quartus-II and Modelsim-Altera, but here we will use Modelsim-Altera as a **standalone product**. Here follows the procedure to create a new project and to *functionally simulate* the top level VHDL unit, in our case the testbench. In the next classes you will learn advanced functions of this tool. Let us concentrate on the basics right now.

1. Open Modelsim-Altera by double clicking on its icon and press close on the pop-up window. You should obtain a window like in figure 7.

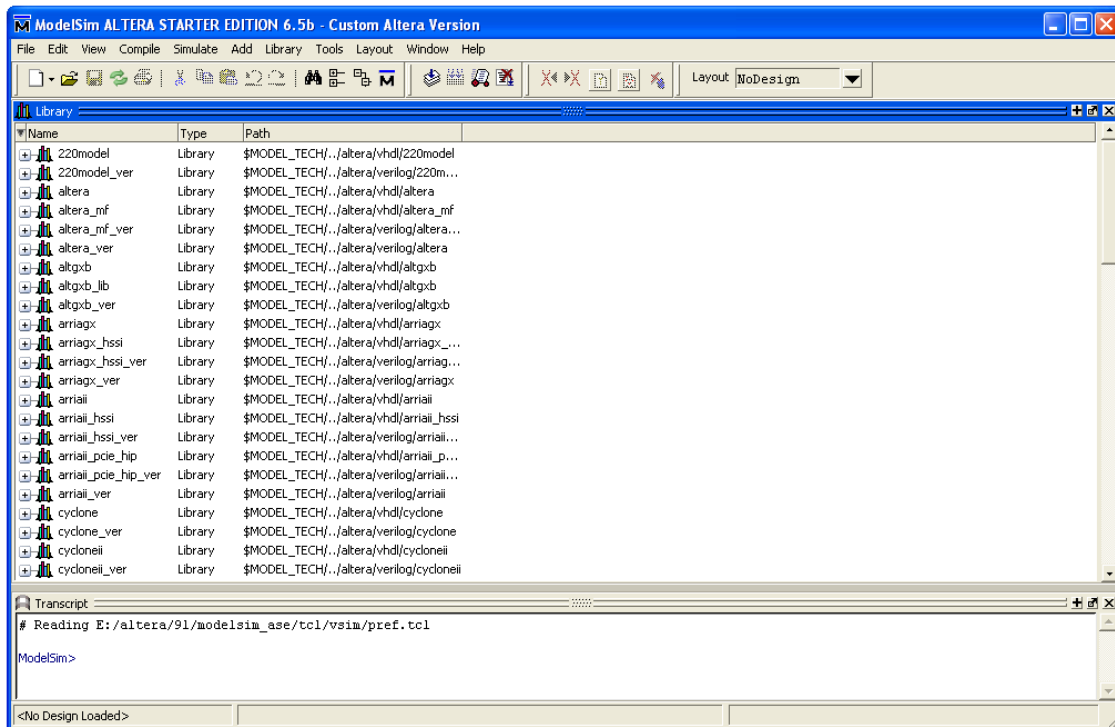
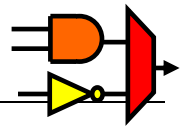


Figure 7 – Modelsim-Altera main window.



2. Create a new project by clicking on **File > New > Project...** The concept of project, that is a collection of files and customized resources, is the same as in Quartus-II. The project will include all the source VHDL files, the compilation outputs and, alternatively, some script to make the simulation and compilation process faster. Use project name *testbench\_es1* and specify the location of your project, not necessarily with the same name of the project. Leave *work* as library name: This library will be used by the tool to collect all the compilation outputs. If the directory *testbench\_es1* doesn't exist, create it and then press **OK** on the window that appears.

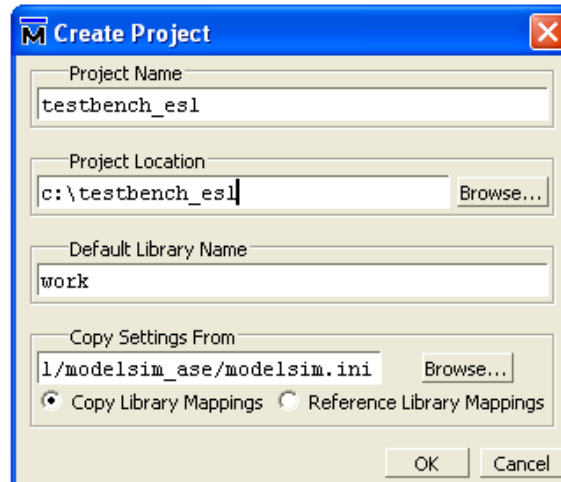


Figure 8 – New project dialog.

3. Add your existing files to your project. You can also select the creation of new VHDL files but here you are required to simulate pre-existing units, then press *Add Existing File* (Figure 9).

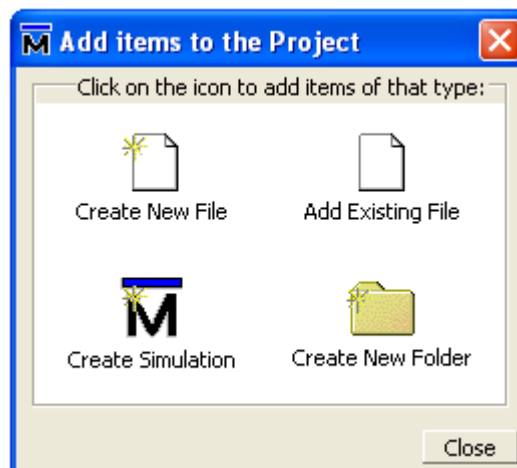


Figure 9 – Add items to the current project.

4. You should now obtain the window in Figure 10. Select all the VHDL files to be added in your project. To do it click on **Browse...** and by using the **CTRL** or the **SHIFT** key select all the files you need. Make sure you keep “reference from current location” selected: This way the created project has links to the VHDL files, and does not copy them in the project directory if absent. This avoids the duplication of the source files on the hard disk, and allows only a copy of each VHDL code. Files with the same name having different versions are dangerous! Press **OK** then.

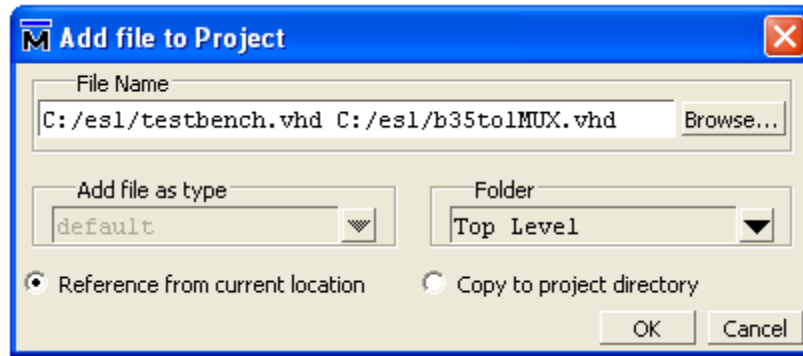
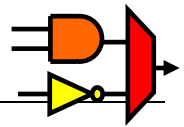


Figure 10 – Adding VHDL files.

5. The main window of Modelsim should become like the one in figure 11.

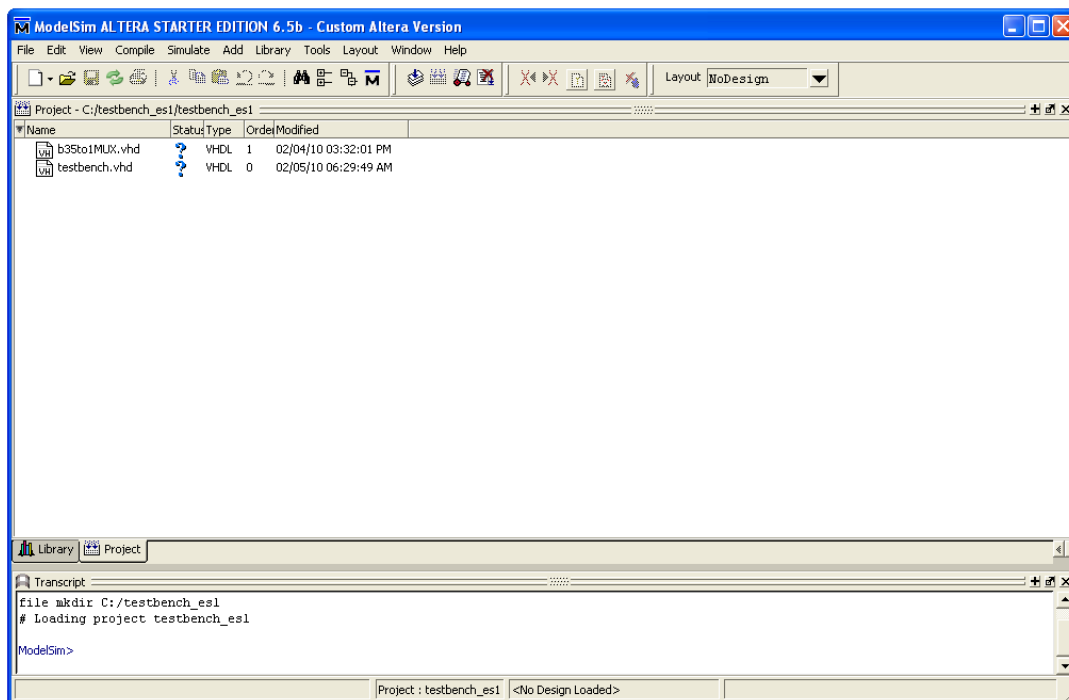
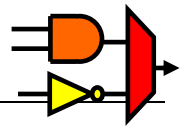



Figure 11 – Modelsim window with project files.

6. Now compile all. To do this, right click on one of the two VHDL files and select **Compile > Compile All**. On the transcript window a message in green should appear, saying that the compilation was successful. There should also be a mark sign for each design file like ✓. If not, ✗, double click on the errors in red on your transcript window and point to the error like you did for Quartus-II. Typically, you did a cut and paste error. Once the compilation has succeeded, Modelsim has mapped your VHDL units to an intermediate level language, which is used by the VHDL simulator. The compilation output files are placed in the library directory, which in our case is *work*.
7. Run the simulation. Make sure once again you added all the files to the project. If some unit is not added, compilation is run with any error but the simulation cannot be run since one or more modules are missing. To run a simulation you can select **Simulate > Start Simulation...** (Figure 12) or alternatively click on the *Library* tab, expand the first entry *work* and finally double click on the name of the testbench entity that is *testbench\_es1*. For this time leave default simulator resolution. After that the system should give the output in Figure 13.



8. Now that the design has been loaded, you need to select the waveforms being added for view. If the design is not too big, just add all the waveforms in the testbench. Type `add wave *` in the transcript window. The main Quartus-II window should become as in Figure 13.
9. Run the simulation for 1 $\mu$ s by typing `run 1000ns` on the transcript window. Click on the waveform and adjust the zoom with the buttons . You should obtain an output as in Figure 14. As you can see, the output of the multiplexer changes every 20ns since its control signal toggles every 20 ns. Inputs U...Y do not change. Output toggles according to the selection bit. Now you can verify if the circuit works correctly or not.

This is an example of simulation with a testbench. You can always create scripts containing all the commands you entered in the transcript window and recall them by typing `do <scriptfile>`. The system will batch execute the script. You should feel more comfortable running Moldesim simulations rather than using the integrated simulator in Quartus-II. Keep in mind that for the next lab you will be asked to use Modelsim for VHDL functional and post-synthesis simulation. Once you have learnt more on the VHDL language during this course you will be able to efficiently create custom testbench for any of your design units. For example, you will be able to read text files containing waveform information or save the output data of your circuit in a text file with a specified format.

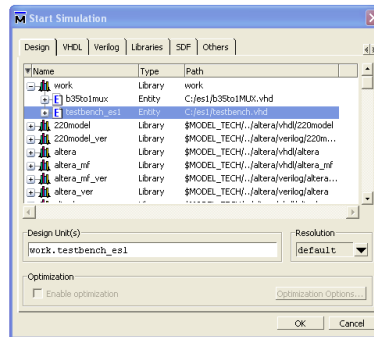


Figure 12 – Start Simulation window.

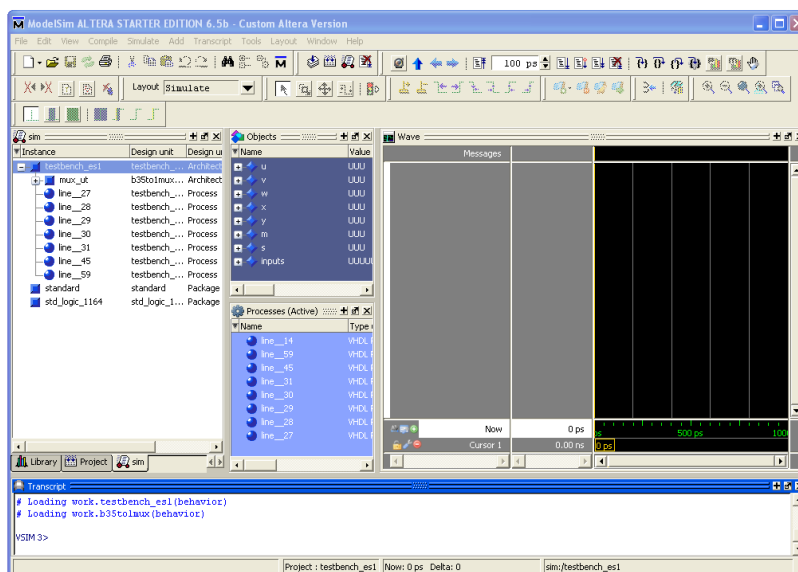


Figure 13 – Modelsim window after simulation load.

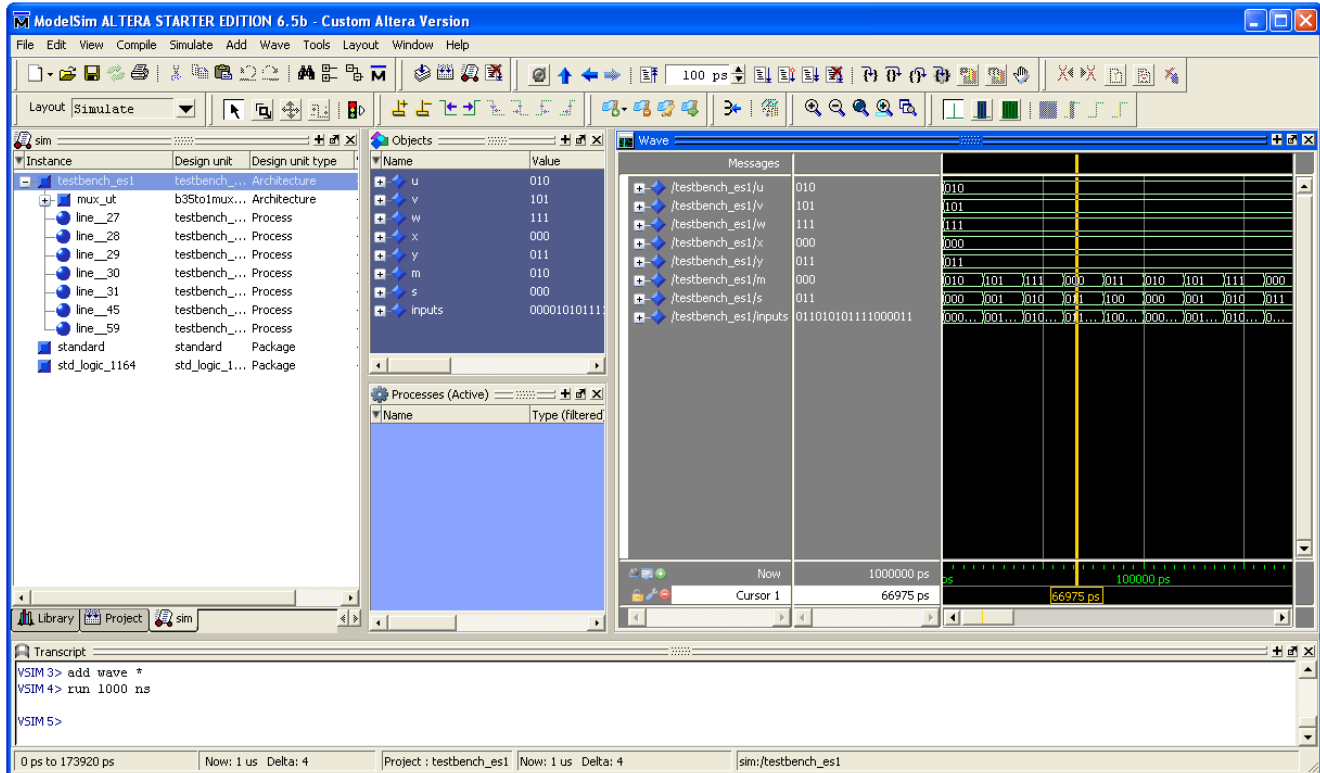
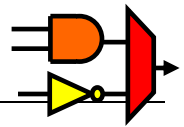


Figure 14 – Output and inputs after 1000 ns simulation.