The **Microelectronics Training Center**

The MTC is an initiative within the INVOMEC division
Industrialization & Training in Microelectronics

Lab-exercise

Lab 4:
Testbench ALU

Cluster: Cluster1
Module: Module2b

Target group: Students

Version: 1.1
Date: 13/12/06
Author: Osman Allam
Modified by: Geert Vanwijnsberghe
History : minimal change

This material was developed with support of the European Social Fund.
ESF: Prevent and combat unemployment by promoting employability, entrepreneurship, adaptability and equal opportunities between women and men, and by investment in people.
http://www.esf-agentschap.be

Introduction

A full testbench forces the inputs of the Unit Under Test (UUT) with simulation patterns and reads its outputs and compares them against a set of expected outputs.

In this module, you create a full testbench for the ALU you've designed in the previous module.

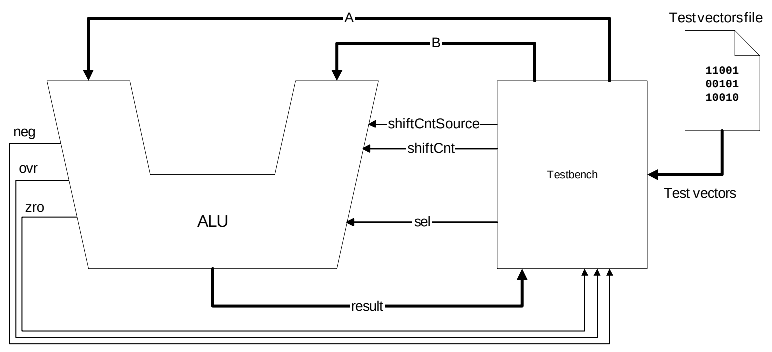


Figure 1: ALU + Testbench

Objectives

After completing this module, You should be able to:
- Read files using TEXTIO procedures
- Convert from one type to another in VHDL
- Verify the behavior of a relatively complex combinational logic block

Knowledge background

- Understanding of binary representation of numbers
- Understanding of arithmetic and logical operations on binary operands
- Basic knowledge of VHDL

Classification

- Level: 3
- Duration: 90 minutes

Input

A VHDL template for the testbench is provided along with some test vectors.

The lab

Reading from a file:

`STD.TEXTIO` and `IEEE.STD_LOGIC_TEXTIO` packages provide a means to enable reading and write formatted text files.
In order to read a file, you first OPEN a FILE of TEXT file type and specify the open kind to be READ_MODE. You also need to specify the file name.

```
file my_file : text open read_mode is "file_name.ext";
```

ⓘ Note: If you don't specify the file path, the simulator assumes that it exists in the current directory.

Reading information from a file is achieved by reading its contents line by line and extracting the information from each individual line. To read lines from the file, you need to declare a variable of type LINE. To extract information from the lines, you need to declare a (set of) variable(s) of the same type as the information you wish to extract.

Example
To read an integer followed by a bit from a line, you declare 3 variables: a line, an integer and a bit.

```
variable my_line    : line;
variable my_integer : integer;
variable my_bit     : bit;
…
readline (my_file, my_line); -- reading the line
read (my_line, my_integer);  -- extracting (reading) the integer
read (my_line, my_bit);      -- extracting (reading) the bit
```

Assertion statements

An assert statement checks if a specified condition is true and reports an error if it is not.

```
Assert <condition>
  report <message>
  severity <severity_level>;
```

The condition is a boolean expression.

The report message can be any expression of type String. In case the report message is omitted, the message "assertion violation" is used by default.

The severity level determines the behavior of the simulator when an assertion is violated. The default severity level is ERROR.

Assertion statements can be used in testbenches to report the result of comparing the UUT's outputs against the expected outputs.

Example: to verify the output "neg" of the ALU. Assume that you have already read the expected value from the test vectors file into the variable "neg_v". You can use an assertion statement like the one below:

```
Assert (neg = neg_v)
  Report “NEG doesn’t match expected value”
  Severity ERROR;
```

You can have more detailed messages printed. For instance:

```
NEG ‘1’ doesn’t match expected value ‘0’ at testvector 17
```

For printing such messages, you use the `‘image()` attribute to convert a `std_logic` or `integer` into a `string` literal. The `‘image()` attribute takes a prefix scalar type or subtype and a parameter of the specified type or subtype.

```
T’image(x)
```

The above example can be rewritten as

```
Variable vector : natural; -- number of current testvector
….
assert (neg = neg_v)
  report “NEG ” & std_logic’image (neg) & “doesn’t match
    expected value “ & std_logic’image (neg_v) & “at
    testvector “ & natural’image (vector)
  severity ERROR;
```

Note: using the `‘image()` attribute with composite data types is not possible. You need to apply the attribute to each element of the parameter, for instance in a loop for array types, provided that the elements are of scalar types or subtypes.

Full testbench

Create a full testbench for the ALU you have designed in the previous module. The testbench uses `STD.TEXTIO` and `IEEE.STD_LOGIC_TEXTIO` procedures to read test vectors (stimuli + expected outputs) from a text file.

Testvector files are formatted text files. Each line is composed of a set of fields. Each field is either a stimulus or an expected value of the output. The fields are arranged in the same order throughout the file and are separated by white spaces. Below is a snapshot of how your testvector file should look like.

```
001  0    0    00000 00000 0    0    0    0    0    1
002  0    0    00000 00000 0    0    1    0    0    1
003  30   150  00000 00000 0    180  1    0    0    0
004  -30  150  00000 00000 0    120  1    0    0    0
005  -150 30   00000 00000 0    -120 1    1    0    0
006  30   -150 00000 00000 0    -120 1    1    0    0
007  150  -30  00000 00000 0    120  1    0    0    0
```

You may want to add comments and headers in the testvector file. Your VHDL code must be able to recognize and effectively ignore them.

ⓘ Hint: You can use your favourite spread sheet software to write the testvectors and then export them to a text file.

A full testbench for a combinational logic block can be accommodated in a single process as shown in the figure below.

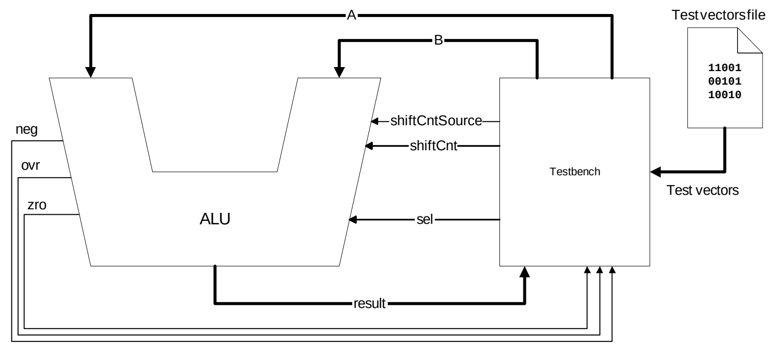


Figure 2: ALU + testbench

For readability, it is recommended to use the port names of the UUT for the signals that connect it to the testbench. The names of the variables into which you read the testvector should also be derived from the signal names by appending for instance "`_v`" to the signal name.

Example:
One of the ALU ports is "result". You declare the signal "`result`" to connect that port to the testbench. The variable "`result_v`" reads the expected value of the result.

The function `ENDFILE` returns TRUE when the end of the file is reached.

```
Function endfile (file F: FT) return boolean;
```

Your testbench loops as long as the end of the file has not been reached. Inside the loop, you read a line, extract the testvectors, apply stimuli and compare the outputs after some delay. The delay_time delay is optional for simulating behavioral models but it is necessary when simulating synthesized models because the actual circuits exhibit propagation delays. The delay offset_time is the time gap between the application of each testvector.

The wait statement at the end of the process causes it to wait forever. This is called "event starvation" and it stops the simulator.

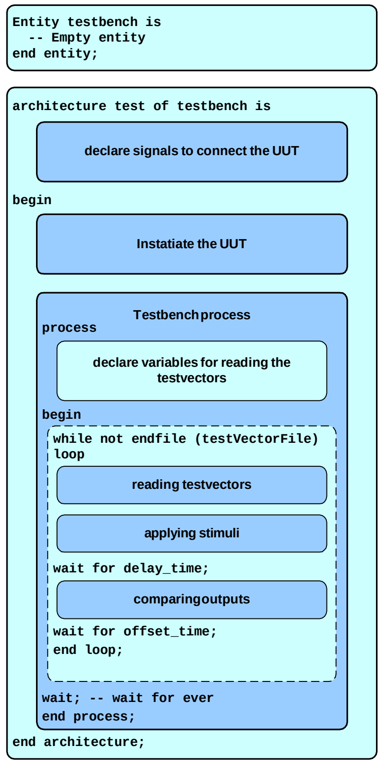
You can use the template provided in the file `cpu_tb.vhd` as guideline for your testbench.

To run the testbench, do the following:

1. Compile the packages micro_pk and micro_comp_pk.
2. Compile the ALU design.
3. Compile the ALU testbench (after completing it).
4. Load the ALU testbench in ModelSim.
5. Run the testbench to the end using the command `run –all` or pressing the button in ModelSim toolbar.

Example:

```
vlib work
# Compile
vcom micro_pk.vhd
vcom micro_comp_pk.vhd
vcom alu.vhd
vcom tb_alu.vhd
# Load
vsim tb_alu
# Run
run -all
```



```
Entity testbench is
  -- Empty entity
end entity;

architecture test of testbench is
    declare signals to connect the UUT
begin
    Instatiate the UUT

    Testbench process
    process
        declare variables for reading the testvectors
    begin
        while not endfile (testVectorFile)
        loop
            reading testvectors
            applying stimuli
        wait for delay_time;
            comparing outputs
        wait for offset_time;
        end loop;
    wait; -- wait for ever
    end process;
end architecture;
```