

# EE-354

## Lab #1

### Combinational Logic Synthesis using GHDL

In this lab project we will be simulating combination logic circuits using GHDL. GHDL is a command-line software utility that allows us to compile and simulate VHDL modules.

VHDL—Very High-Level Description Language

GHDL—“GNU” implementation of VHDL

#### Installing GHDL (On Your Own Computer)

GHDL may be downloaded via one of the following two websites:

<http://ghdl.free.fr/site/pmwiki.php?n=Main.Download>

Both downloads and installation instructions are contained within the sites. In order to “view” the output of GHDL modules (and module simulations), we can use GTKWAVE:

<https://sourceforge.net/projects/gtkwave/>

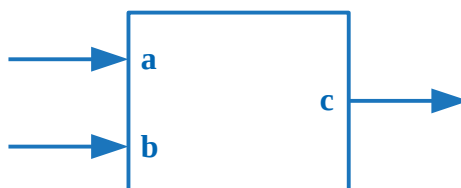
#### Implementing Elementary Boolean Functions using GHDL

GHDL is a command-line application: all operations are performed using the Console or “DOS” prompt. First, a VHDL file must be created. This file must have an extension “.vhd” and must be created using Notepad or Notepad++. (Microsoft Word or Wordpad can be used, but the file must be saved as a “text file.”) The following is an example Verilog file (called **a.vhd**):

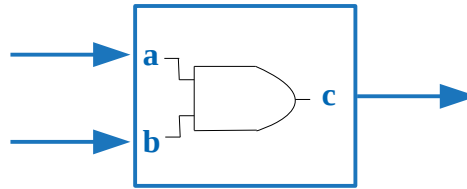
```
entity and1 is
  port(
    a : in bit;
    b : in bit;
    c : out bit
  );
end and1;

architecture behavior of and1 is
begin
  c <= a and b;
end;
```

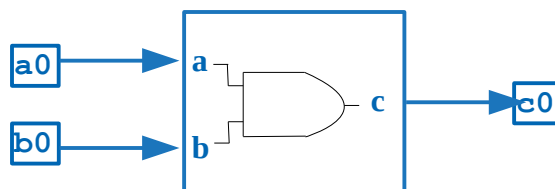
You can think of the “**port()**” as a description of a logic block:



Within the logic block is a Boolean function:



The (inner) Boolean function is defined with the statement `c <= a and b;`.  
In order to test this module, we need to create some inputs and outputs



The inputs `a0` and `b0` need to be set to (logic) values. we may construct a “test module”:

```
entity tand1 is
end tand1;

architecture a of tand1 is

    component and1
        port(
            a : in bit;
            b : in bit;
            c : out bit
        );
    end component;

    signal a0,b0 : bit;
    signal c0 : bit;

begin

    and10: and1 port map(a0, b0, c0);

    a0 <= '1'; b0 <= '1';

end;
```

This “test module” can be “compiled” and run using the commands:

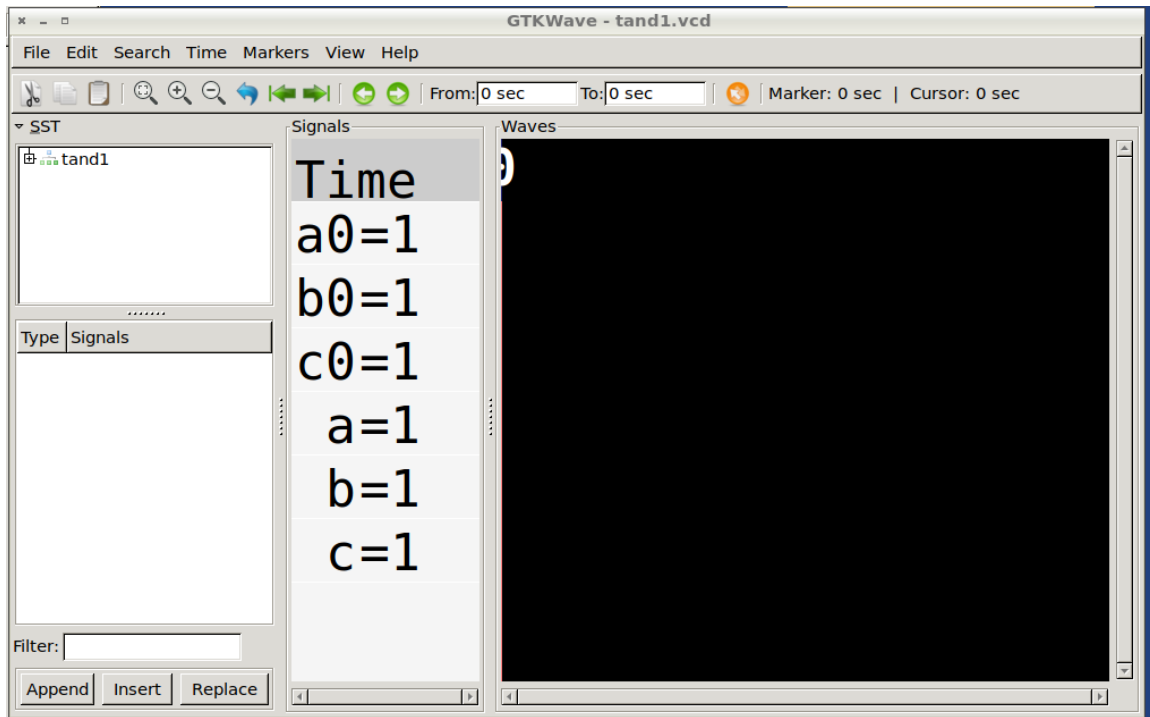
```
ghdl -a tand1.vhd
```

```
ghdl -r tand1 --vcd=tand1.vcd
```

No output should be generated. To “view” the result, we need to use gtkwave:

```
gtkwave tand1.vcd
```

You should see a diagram similar to **Figure 1**.



**Figure 1.** Display window for gtkwave. Note that the “Waves” pane is blank—later it will not be blank.

Important note: The only “hardware” component is **entity and1**. Ultimately, this module will be burned into an actual FPGA. The “test module” **entity tand1** is just for test—it is “software” without a corresponding physical component. (The “hardware” module will not be physical until it is burned into a physical FPGA chip.)

We can expedite the testing process using variables within a “process” in our “test” module:

```
entity tand1 is
end tand1;

architecture a of tand1 is

    signal a,b : bit;
    signal c : bit;

    component and1
        port(
            a : in bit;
            b : in bit;
            c : out bit
        );
    end component;

begin

    and10: and1 port map(a,b,c);

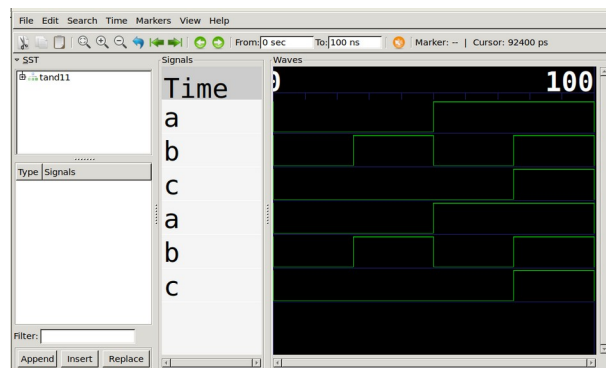
    process
    begin
        a <= '0';      b <= '0';      wait for 25 ns;
        a <= '0';      b <= '1';      wait for 25 ns;
        a <= '1';      b <= '0';      wait for 25 ns;
        a <= '1';      b <= '1';      wait for 25 ns;
    end process;

end;
```


This module is compiled as usual, but is run in the following manner:

```
ghdl -r tand1 --vcd=tand1.vcd --stop-time=100ns
```

We can now “view” the output using gtkwave (`gtkwave tand1.vcd`). The output should look like **Figure 2**.



**Figure 2.** Display window for gtkwave.



**Assignment:** Implement the following (three-input) Boolean functions into modules.

$$d = ab + c.$$

$$d = (a + b)c.$$

$$d = abc + ab + c.$$

In each case write test modules, and generate test outputs (and include them in your lab report). In some cases minimization may be performed. If so, find the minimized function(s) as well. (Show Karnaugh maps for minimized functions.)

## Implementing More Complex Descriptions using GHDL

One of the most powerful tools in any programming (or hardware description) language is **arrays**. Arrays allow you to treat multiple variables (or signals) as one variable (or signal). The fundamental array building block in VHDL is a “**bit\_vector**.” For example, suppose we wished to put the two input signals for an and gate into a single bit\_vector; here is the syntax:

```
signal x : bit_vector (0 to 1);
```

We can then construct the output using

```
y <= x(0) and x(1);
```

The entire module is as follows:

```
entity and2 is
  port(
    x : in bit_vector (0 to 1);
    y : out bit
  );
end and2;

architecture behavior of and2 is
begin
  y <= x(0) and x(1);
end;
```

The test module is **Figure 3**

```
entity tand2 is
end tand2;

architecture a of tand2 is

  component and2
    port(
      x : in bit_vector (0 to 1);
      y : out bit
    );
  end component;

  signal x0 : bit_vector (0 to 1);
  signal y0 : bit;

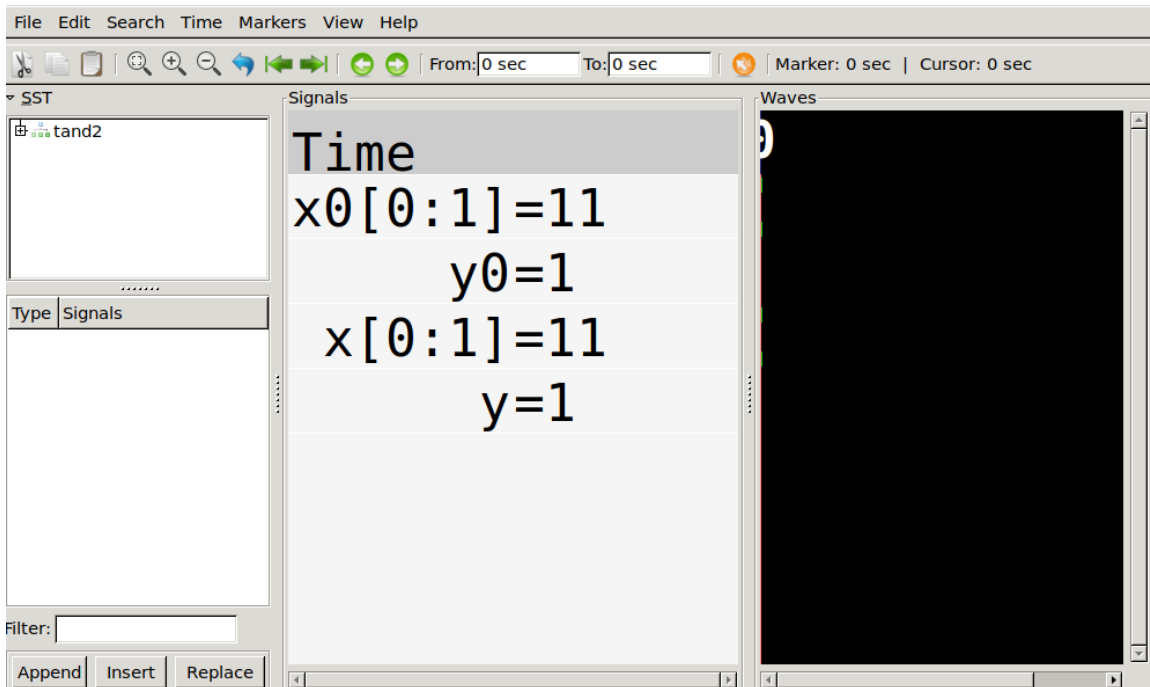
begin

  and20: and2 port map(x0, y0);

  --x0(0) <= '1'; x0(1) <= '1';
  x0 <= "11";

end;
```

After compiling and running this module, we can “view” the output signals (using gtkwave) and get an image such as that shown in **Figure 3**.



**Figure 3.** GTKWave output for **tand2.vhd**. Note that the arrays are shown with square brackets **[ ]** rather than parentheses **()**.

**Assignment:** Write modules that perform four-bit additions and a four-bit subtractions using four-bit **bit\_vectors**. Write a test module with a “process” that will test the addition/subtraction of the following pairs of numbers: **0001±0001, 0111±0111, 1000±1000, 1111±1111**. (Verify that the results are the correct arithmetic answers.)