Lab # 1

Introduction to VHDL

September, 2013
Introduction

This lab is an introduction to the VHDL language (scientific). We will learn the language syntax and we will stop at each word of the program explaining what it is and things related to it since these are the first programs. Also, we will illustrate the differences between structural and behavioral modeling in VHDL. We will build a half adder as an example. As an exercise you will be asked to do the full adder in the lab.

VHDL in more details

The general format of a VHDL program is built around the concept of BLOCKS which are the basic building units of a VHDL design. Within these design blocks a logic circuit of function can be easily described.

A VHDL design begins with an ENTITY block that describes the interface for the design. The interface defines the input and output logic signals of the circuit being designed. The ARCHITECTURE block describes the internal operation of the design. Within these blocks are numerous other functional blocks used to build the design elements of the logic circuit being created.

After the design is created, it can be simulated and synthesized to check its logical operation. SIMULATION is a bare bones type of test to see if the basic logic works according to design and concept.

Many software packages used for VHDL design also support schematic capture which takes a logic schematic or state diagram and translates it into VHDL code. This, in turn, makes the design process a lot easier. However, to fine tune any design, it helps to be familiar with the actual VHDL code.

REMEMBER: You are NOT writing software. You are DESCRIBING the functionality of the hardware you want.

When writing in C or other programming language you are allowed a lot of freedom by the compiler. But in this case you are physically creating blocks of digital circuits which are wired together and have to be implemented in a chip. A simple statement in C, like a division of two numbers, causes great problems to a VHDL compiler, and the hardware implementation is very complicated. Have this in mind when coding. Think that the compiler and synthesizer have to be able to layout and wire your design, and download it to a chip.

Coding style

Make sure you include all the appropriate libraries needed for your design. In some cases you will need a certain library for compiling, and a different one for synthesizing. Also, if you have created any packages, include the USE statement so that the program can find your package file. This is an example of how your program header should look:
-- The name of your program
-- Your name
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE (your own user defined packages)

• **Std_logic**

It is recommended to use the multi-valued logic system from the IEEE instead of the standard 'bit' data type. The new type is called 'std_logic' and is defined in the package 'std_logic_1164' which is placed in the library IEEE (i.e. it is included by the following statement: 'use IEEE.std_logic_1164.all'.

```plaintext
TYPE STD_ULOGIC IS (  
    'U', -- uninitialized (not connected)  
    'X', -- Forcing Unknown  
    '0', -- Forcing 0  
    '1', -- Forcing 1  
    'Z', -- High Impedance  
    'W', -- Weak Unknown  
    'L', -- Weak 0  
    'H', -- Weak 1  
    '-', -- don‘t care);
```

➤ **Behavior modeling (Processes)**

We can write a behavior architecture body of an entity which describes the function in an abstract way. Such an architecture body includes only process statements.

**Process contains:**

Conventional programming language constructs. A process is a sequentially executed block of code, which contains.

• Evaluating expressions.
• Conditional execution.
• Repeated execution.
• Subprogram calls.
• Variable assignments, e.g., \( x := y \), which, unlike signal assignment, take effect immediately.
• if-then-else and loop statements to control flow.
• Signal assignments to external signals.
Notes:
1. Signal assignment statements specify the new value and the time at which the signal is to acquire this value. The textual order of the concurrent signal assignment statements (CSAs) does NOT affect the results.
2. Processes contain sensitivity lists in which signals are listed, which determine when the process executes.
3. In reality, CSAs are also processes without the process, begin and end keywords.

Structure modeling

Structural model: A description of a system in terms of the interconnection of its components, rather than a description of what each component does. A structural model does NOT describe how output events are computed in response to input events.

A VHDL structural description must possess:
The ability to define the list of components.
The definition of a set of signals to be used to interconnect them.
The ability to uniquely label (distinguish between) multiple copies of the same component.

Components

blocks that already exist and are included into a higher level design, we need to know the entity declaration of the system we are calling, we "declare" a component using the keyword "component", we declare the component in the architecture which indicates we wish to use it. After the "begin" keyword, we can start adding components and connecting signals, we add components with a "Component Instantiation" syntax:

```vhdl
label : component-name port map (port => signal, ...);
```

All components are executed CONCURRENTLY - this mimics real hardware connection, this is different from traditional program execution (i.e., C/C++) which is executed sequentially because we are NOT writing code, we are describing hardware!!!

Test Benches

We need to stimulate our designs in order to test their functionality. Stimulus in a real system is from an external source, not from our design. We need a method to test our designs that is not part of the design itself; this is called a "Test Bench". Test Benches are VHDL entity/architectures with the following:
We instantiate the design to be tested using components, we call these instantiations "Unit Under Test" (UUT) or "Device Under Test". The entity has no ports, we create a stimulus generator within the architecture, and we can use reporting features to monitor the expected outputs.
Test Benches are for Verification, not for Synthesis!!! This allows us to use constructs that we ordinarily wouldn't put in a design because they are not synthesizable.
Example: half adder

A half adder is a logic circuit that performs one-digit addition. It has two inputs (the bits to be summed) and two outputs (the sum bit and the carry bit). An example of a Boolean half adder is this circuit in figure (1):

![Half Adder Diagram]

The Entity for the half adder:

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
entity half_adder is
  port (a : in std_logic;
        b : in std_logic;
        sum : out std_logic;
        cout : out std_logic);
end half_adder;
```

The Behavior Model for the half adder:

```vhdl
architecture basic_beh of half_adder is
begin
  process (a,b)
  begin
    if (a = '1' and b = '1') then
      sum <= '0';
      cout <= '1';
    elsif (a = '1' and b = '0') then
      sum <= '1';
      cout <= '0';
    elsif (a = '0' and b = '1') then
      sum <= '1';
      cout <= '0';
    elsif (a = '0' and b = '0') then
      sum <= '0';
      cout <= '0';
    else
      ...
  end process;
end basic_beh;
```
The basic Structural Model for the half adder:

```vhdl
architecture basic_struct of half_adder is
begin
sum <= a xor b;
cout <= a and b;
end basic_struct;
```

The Structural Model for the half adder (Using component instantiations):

- **The AND gate:**

  ```vhdl
  library IEEE;
  use IEEE.std_logic_1164.all;
  entity myAnd is
  port (a : in std_logic;
         b : in std_logic;
         o : out std_logic);
  end myAnd;
  
  architecture basic_and of myAnd is
  begin
  o <= a and b;
  end basic_and;
  ```

- **The XOR gate:**

  ```vhdl
  library IEEE;
  use IEEE.std_logic_1164.all;
  
  entity myXOR is
  port (a : in std_logic;
         b : in std_logic;
         o : out std_logic);
  end myXOR;
  
  architecture basic_XOR of myXOR is
  begin
  o <= a and b;
  end basic_XOR;
  ```
- The architecture of half_adder:

```vhdl
architecture halfadder_struct of half_adder is
begin
  and1: entity work.myAnd(basic_and) port map (a,b,c);
  xor1: entity work.myXOR(basic_XOR) port map (a,b,s);
end halfadder_struct;
```

The Structural Model for the half adder (Using component instantiations another way):

```vhdl
architecture halfadder_struct2 of half_adder is
component myAnd is
  port (a : in std_logic;
        b : in std_logic;
        o : out std_logic);
end component;

component myXOR is
  port (a : in std_logic;
        b : in std_logic;
        o : out std_logic);
end component;

begin
  and1: myAnd port map (a,b,c);
  xor1: myXOR port map (a,b,s);
end halfadder_struct2;
```

The Test bench:

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

entity half_adder_tb is
end half_adder_tb;

architecture TB_ARCHITECTURE of half_adder_tb is
  -- Stimulus signals - signals mapped to the input
  -- and inout ports of tested entity
  signal a : std_logic;
  signal b : std_logic;
  -- Observed signals - signals mapped to the output
  -- ports of tested entity
  signal sum : std_logic;
  signal cout : std_logic;
```
begin
  -- Design Under Test port map
  DUT : entity work.half_adder(basic_struct)
    port map (
      a => a,
      b => b,
      sum => sum,
      cout => cout
    );
  
  -- your stimulus here ...

stim : process is
begin
  a <= '0'; b <= '0'; wait for 20 ns;
  a <= '0'; b <= '1'; wait for 20 ns;
  a <= '1'; b <= '0'; wait for 20 ns;
  a <= '1'; b <= '1'; wait for 20 ns;
  wait;
end process stim;
end TB_ARCHITECTURE;

➢ Lab. Exercises

• L. Exercise:
  Write a complete VHDL behavior and structural description of a full adder, test the design using a suitable testbench (apply the truth table), and show the waveform.
H. Exercise:

1. Construct a 2-bit full adder using component instantiations. Show some random input combinations in your test bench.

2. Write a complete VHDL behavior and structural description of two-to-one multiplexer, test the design using a suitable testbench.

3. Construct a quadruple two-to-one multiplexer (A, B and M are 4 bits) using component instantiations, test the design using a suitable testbench.