EE126 Lab 1, Fall 2005

VHDL, Verilog, and the Xilinx environment Tutorial

Table of Contents

This tutorial is intended to familiarize you with the Xilinx environment and introduce the hardware description languages VHDL and Verilog. The tutorial will step you the implementation and simulations of a full-adder in both languages. Using this background you will implement a four-bit adder in both VHDL and Verilog. In the future, HDL labs can be done in either language.
You may want to refer to Appendix A to review the standard structures of VHDL and Verilog modules.

1. Example Project 1: Full Adder in VHDL

On starting Xilinx Project Navigator, you should be faced with a screen like this:

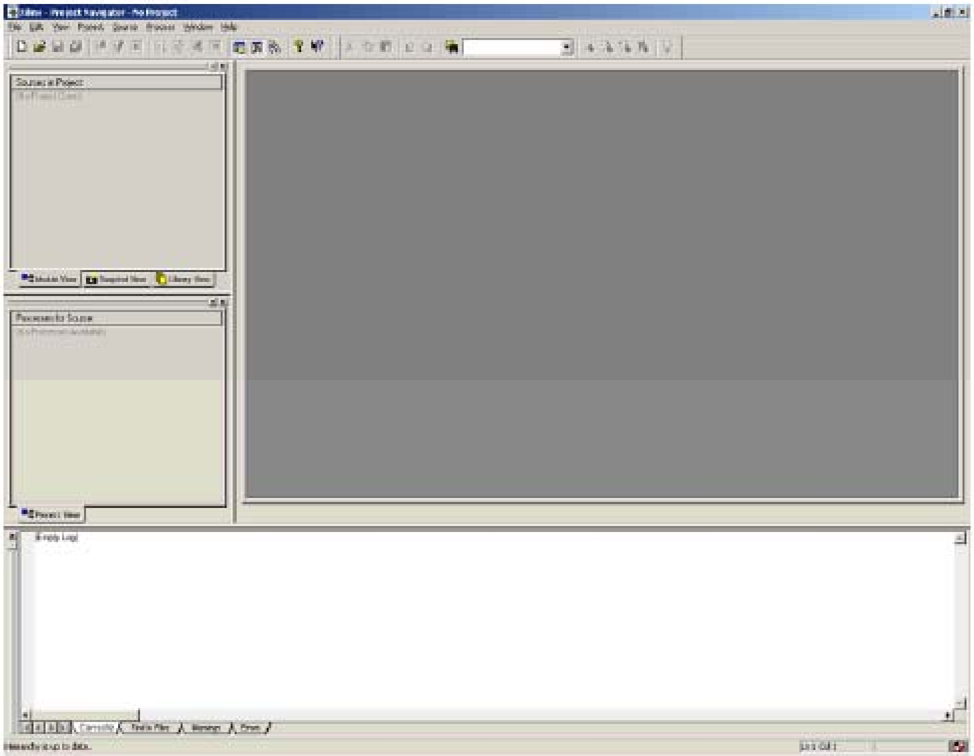

Figure 1. Xilinx Project Navigator

Go to "File -> New Project"

Type in the name of your project (let’s use lab1_*yourname*), choose a location, and specify the top-level module type as HDL (Hardware Description Language) as in Fig. 2.

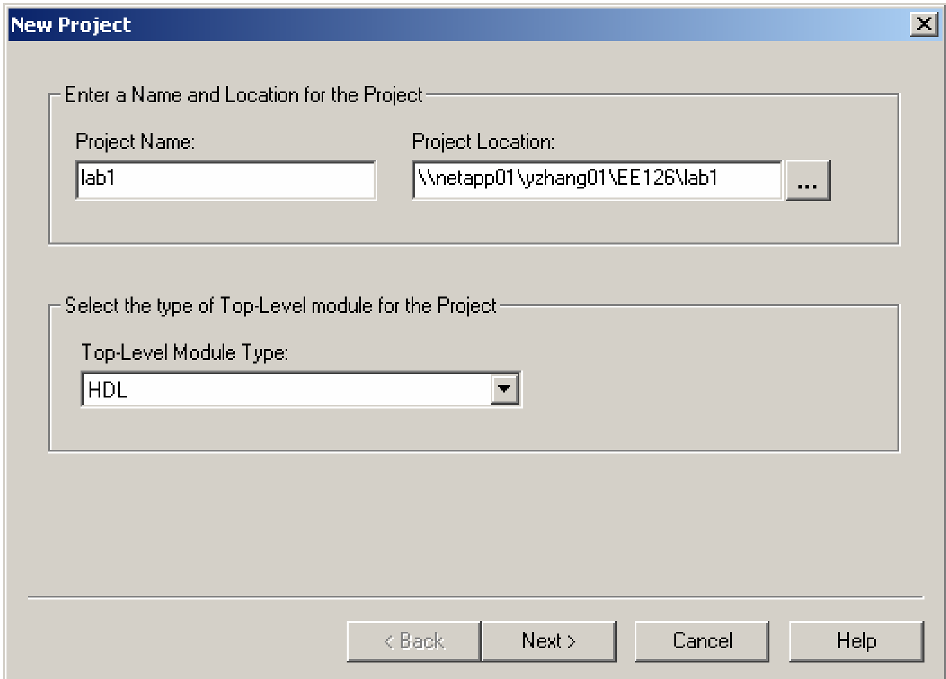

Figure. 2

Click Next. Enter the settings shown in Figure 3. These are the parameters for the Xilinx Spartan2E FPGA chip on our Digilab D2E boards.

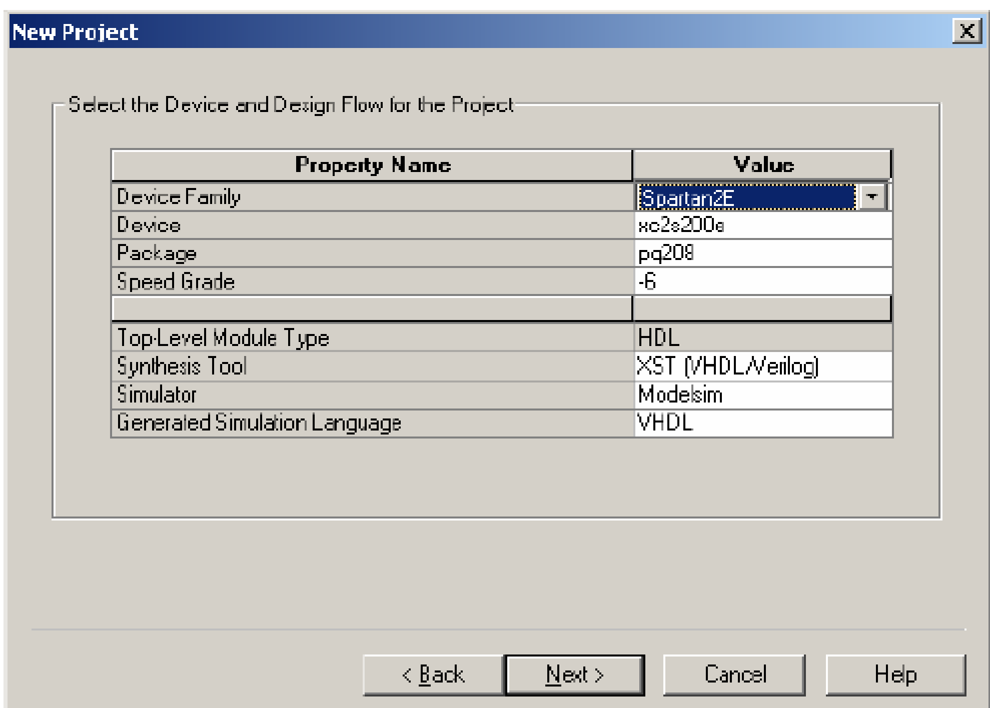

Figure. 3

Click Next > Next > Next > Finish to close the window.

Go to “Project -> New Source”

Select VHDL Module and type in the filename (here we use *fulladder*), as shown in Figure 4.

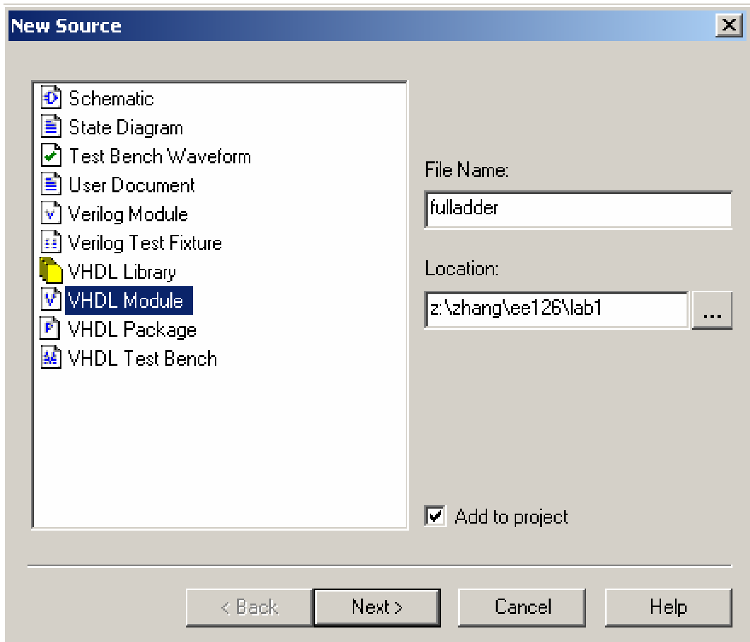

Figure. 4

We will build a 1-bit full-adder in this module. Let the tool generate the entity interface for us. Set the parameters as shown in Fig. 5.

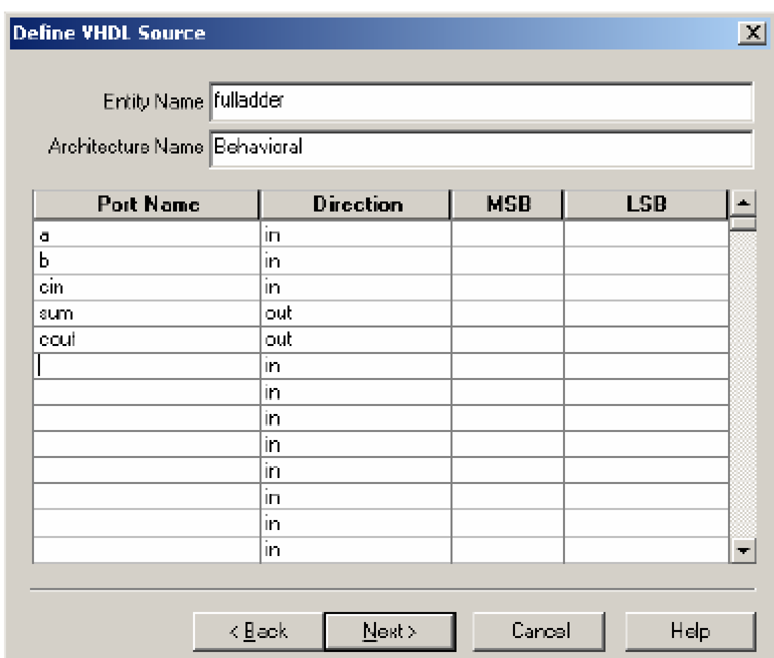

Figure. 5

After clicking “Next” and “Finish”, you should see a piece of code generated for you:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity fulladder is
    Port ( a : in std_logic;
           b : in std_logic;
           cin : in std_logic;
           sum : out std_logic;
           cout : out std_logic);
end fulladder;

architecture Behavioral of fulladder is

begin


end Behavioral;
```

Figure. 6

Complete the description of full-adder by adding two lines describing how **cout** and **sum** are generated, as in Figure 7:

```
architecture Behavioral of fulladder is
   signal s1,s2,s3: std_ulogic;
   constant gate_delay: Time := 100 ps;

begin
  s1 <= (a xor b) after gate_delay;
  s2 <= (cin and s1) after gate_delay;
  s3 <= (a and b) after gate_delay;
  sum <= (s1 xor cin) after gate_delay;
  cout <= (s2 or s3) after gate_delay;
end Behavioral;
```

Figure. 7

-- Constant can be used to declare a constant of a particular type. In this case, Time.
-- The functional relation between the input and output signals is described by the architecture body.
-- Only one architecture body should be bound to an entity, although many architecture bodies can be defined.

Save your code and then double click “Synthesize” in the "Processes for current source" window.

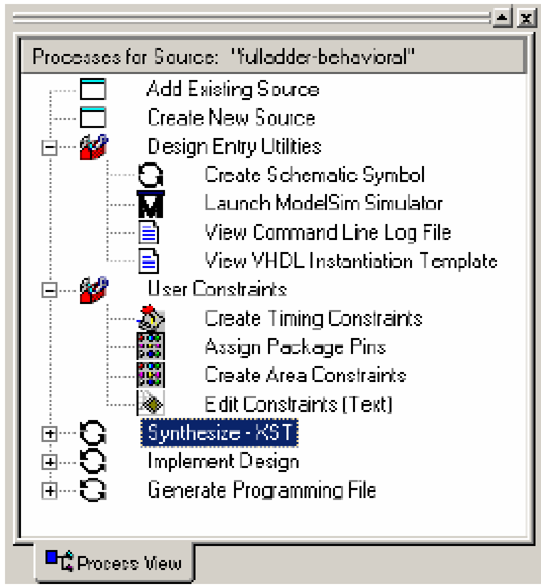

Figure. 8

A green check will appear next to "Synthesize – XST" if the software is able to synthesize your code. If you see a red cross, you need to double check the syntax of your code and synthesize again.

Simulation with ModelSim

Before implementing our design into hardware, we want to simulate the behavior of our code and see if the function is correct logically. A software package called ModelSim has been installed for this purpose. Expand "Design Entry Utilities" and double click on "Launch ModelSim Simulator".

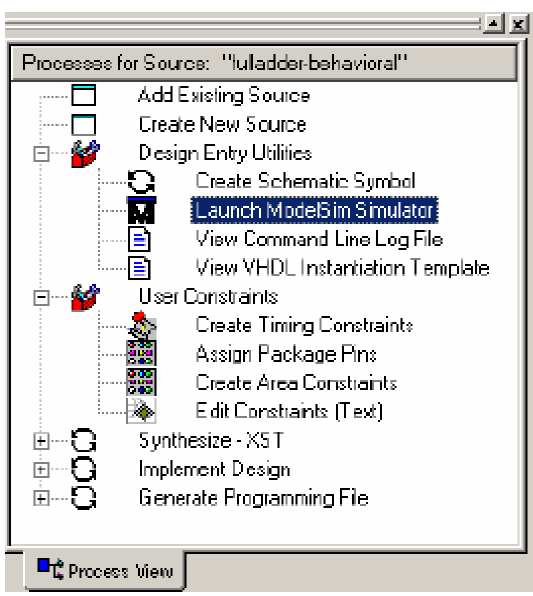

Figure. 9

A windows of ModelSim will show up.

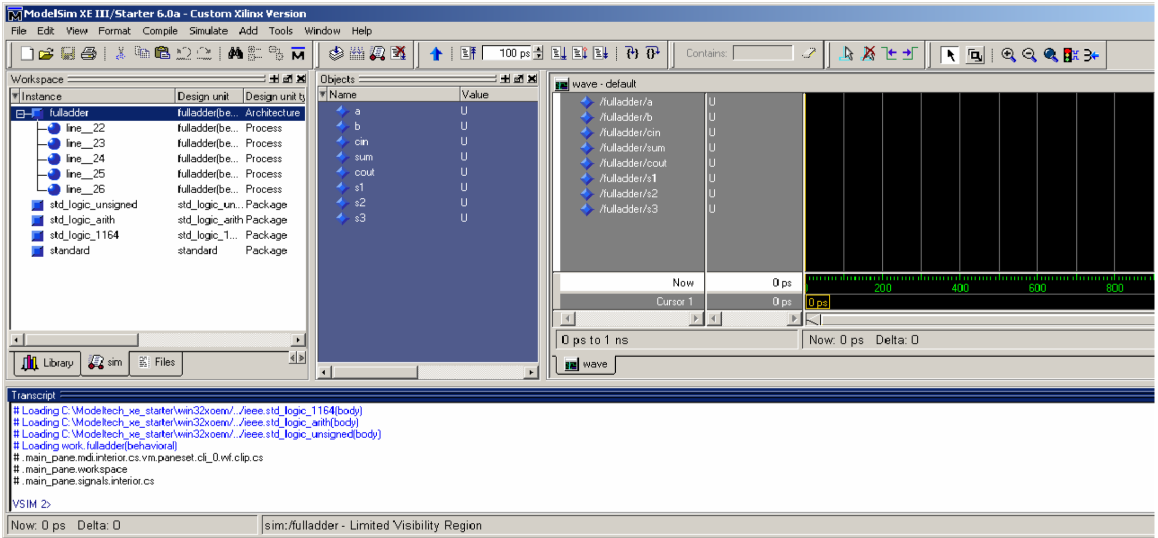

Figure. 10

Select signal "a" by clicking it. Go to "Edit" -> "Clock" and "Define Clock" window will show up:

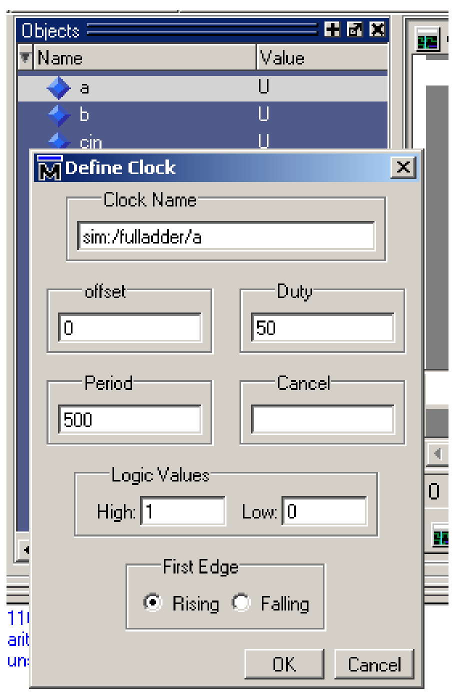

Figure. 11

We will assign a clock signal to "a" with a "period" of 500ps and a 50% duty cycle. We will apply clock signals with different frequencies to the inputs of the full adder such that all possible input combinations are tested.

Apply clock signals to inputs “b” and “cin.” Set the period of **“b” to be “1000”** and **“cin” to be “2000”** as shown in Figure 12

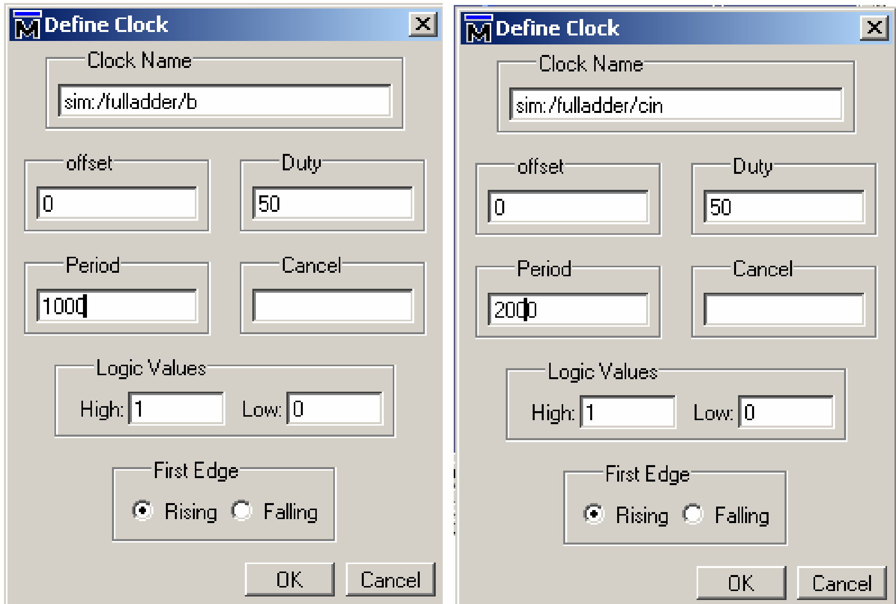

Figure.12

Now, go to the left side of ModelSim window,

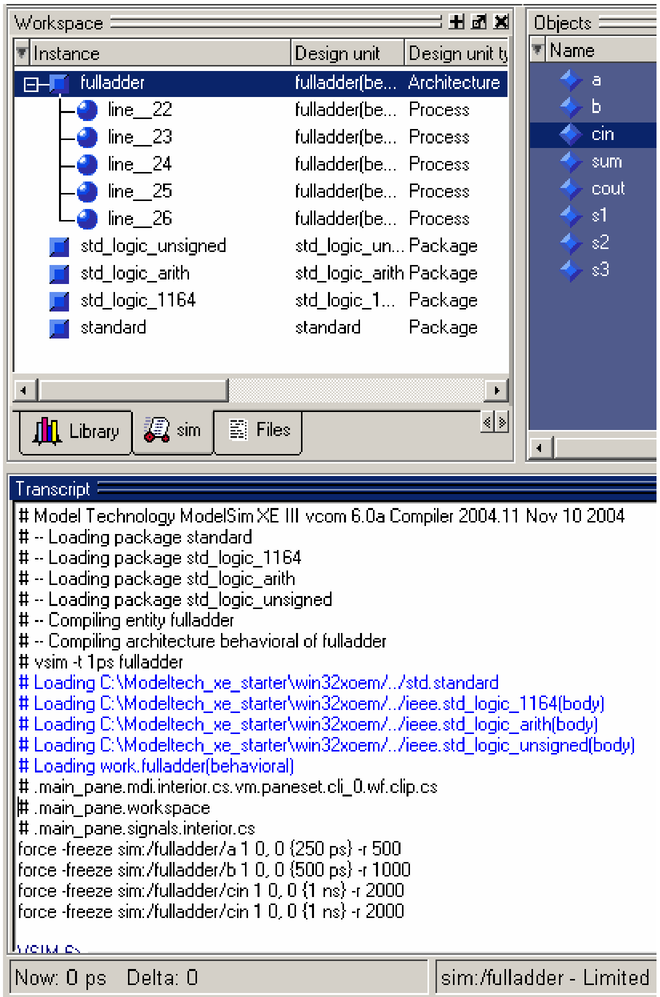

Figure. 13

The three “force-freeze” commands you see here are the command-line versions of the signal assignments we just made to “a”, “b” and “cin.”
By default, the simulation will step in increments of 100ps – you can change this step time in the field shown in Figure 14.

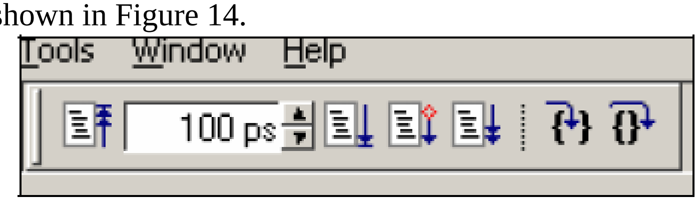

Figure. 14

Run the simulation for 2000ps to cover all input combinations, by clicking the “run” button (to the right of the step time field) four times. Waveforms should appear in the wave window as in Figure 15.

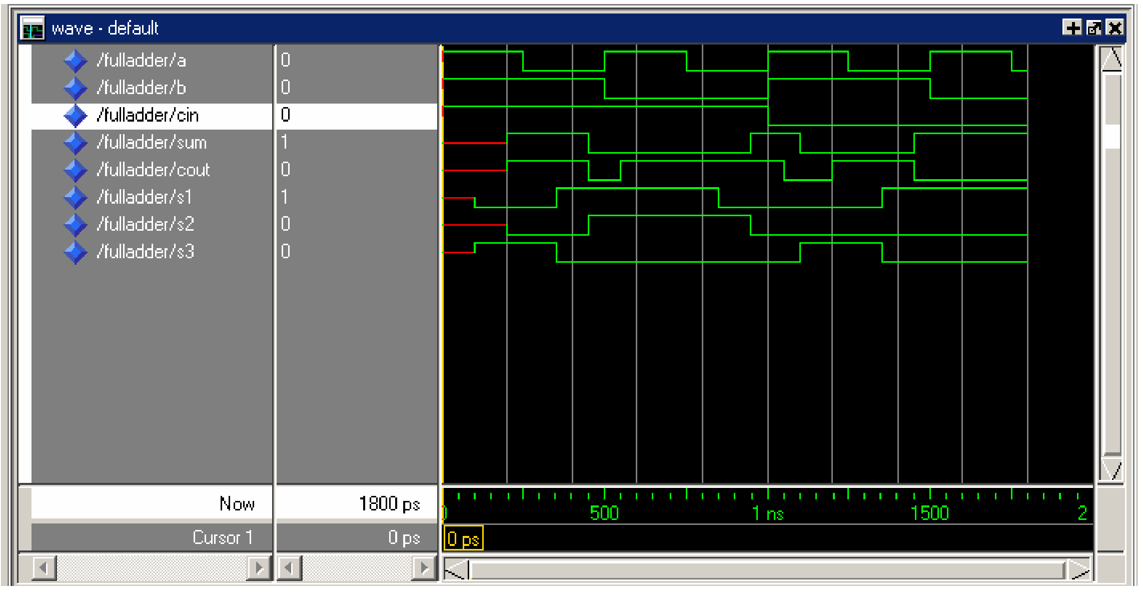

Figure. 15

Now you can verify the correctness of output values of “sum” and “cout” depending on different input values of “a”, ”b”, ”cin”. To end the simulation, close the main window of ModelSim.

2. Example Project 2: Full Adder in Verilog

In this example we will repeat the design and simulation of a full adder, now using Verilog. Open a new project with the same settings as example project 1 (with a new name though).

Click Project > New Source. Name the source "fulladder" again, but this time highlight "Verilog Module."

Define the IO of module, with "a," "b," and "cin" as inputs and "sum" and "cout" as outputs. Click Next > Finish.

The following piece of code should be generated automatically:

```
module fulladder(a,b,cin,sum,cout);
    input a;
    input b;
    input cin;
    output sum;
    output cout;



endmodule
```

Figure 16.

In Verilog, a module's inputs and outputs are listed at least twice – once in the IO list following the module name, and again inside the module where they are assigned a direction.

Verilog module outputs need to be registered. That is to say, the result of a logical expression cannot be sent directly to an output pin, but must first be buffered by a register. This is accomplished by declaring a register with the same name as the signal. Since "sum" and "cout" are output pins, add registers as shown in Figure 17.

```
module fulladder(a,b,cin,sum,cout);
    input a;
    input b;
    input cin;
    output sum;
    output cout;

    reg sum;
    reg cout;
```

Figure 17.

Finally we need to add the logical expressions used to generate values for "sum" and "cout."

Refer to Table 1 for the Verilog syntax of common logical operators.

Operator	Verilog Syntax
AND	&&
OR	||
XOR	^^
NOT	!

Table 1.

The final fulladder module should look as in Figure 18.

```
module fulladder(a,b,cin,sum,cout);
    input a;
    input b;
    input cin;
    output sum;
    output cout;

    reg sum;
    reg cout;

always @(a or b or cin)
  begin
    sum <= a ^^ b ^^ cin;
    cout <= (a && b) || (a && cin) || (b && cin);
  end
endmodule
```

Figure 18

Note that the expressions for "sum" and "cout" are placed in an **always** block. An **always** block is executed any time one of the signals in the sensitivity list ("a or b or cin" in this case) changes. This tells the synthesizer to update the "sum" and "cout" registers only when an input changes.

The procedure for synthesizing and simulating the fulladder module is the same as in the VHDL section. Repeat this procedure and verify that the ModelSim waveforms are correct.

3. Lab 1 Assignment

In both VHDL and Verilog, use the full-adder modules created in the above tutorials to implement four-bit adder modules with the architecture shown in figure 19. To do this, create a new source in the project where you designed the fulladder. You will have to declare multi-bit signals and instantiate the fulladder modules in this new source. Connect the "cout" pin of each full-adder to the "cin" pin of the next.

Refer to Appendix A for module instantiation format, multi-bit signal declarations etc.

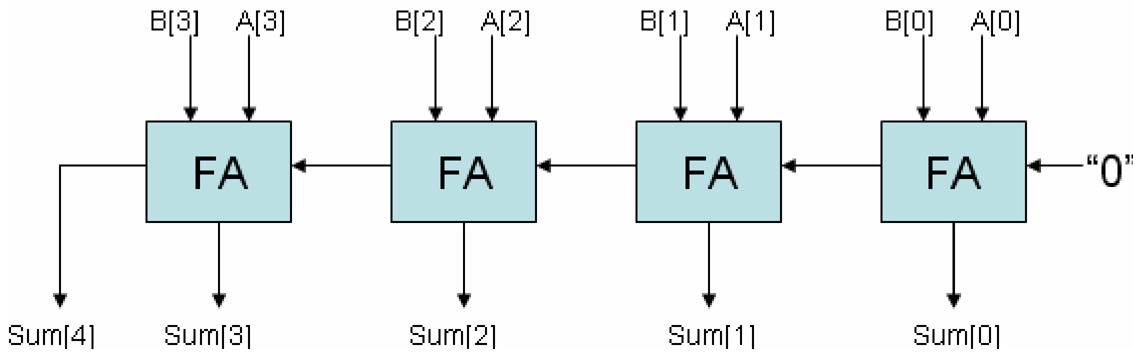

Figure 19

Once the four-bit adder is able to synthesize, run ModelSim to test your design. Step the simulation with several different input combinations and verify the adder's functionality. Record results and/or take some screenshots.

4. Programming the FPGA

If you would like to see your design implemented in an FPGA please follow the instructions in Appendix B.

5. Lab Report Guidelines

Please write up a report on the HDL implementation and simulation of the four-bit adders created in this lab. The lab report should at least include a purpose, procedure, results, and conclusion. Please include all HDL in an appendix.

Appendix A: VHDL and Verilog Standard Formats

Standard Structure of a VHDL Design

```
entity entity_name is
        Port(signal0 : in std_logic;
                signal1 : out std_logic;
                …
                signaln : out std_logic_vector (3 downto 0));
end entity_name;

architecture Behavioral of entity_name is

-- component declarations
component comp_name is
        Port(a : in std_logic;
                …);
end component;

-- signal declarations
signal wire0, wire1 : std_logic;

-- main block
begin

-- behavioral and/or structural code here.

-- module instantiation
instance_name: comp_name
        port map(signal0, signal1, …);

-- logical operations
signal3 <= (signal4 and signal5) xor signal8;

end
```

<u>Standard Structure of a Verilog Design</u>

```
module module_name(signal0,
                   signal1,
                   … ,
                   signaln);

    // module signals
    input signal0;
    output [15:0] signal1;
    …
    output signaln;

    // internal registers
    reg register0;
    reg signal1;

    // internal signals
    wire wire0;
    wire wire1;

// behavioral and/or structural code here.

// module instantiation
module_name1 instance_name1 (signal0, signal1);

// logical operations
always @ (signal4 or signal5 or signal8)
begin
    signal3 <= (signal4 && signal5) ^^ signal8;
end

endmodule
```

Appendix B: Programming the Spartan2E FPGA

Ultimately HDL modules are implemented in hardware such as ASICs (Application-Specific Integrated Circuits) or FPGAs (Field-Programmable Gate Arrays). The Xilinx software is able to create a programming file from a synthesized HDL module, which can be downloaded into a Xilinx FPGA.

The following is a brief example of FPGA programming, using the fulladder module we created in VHDL. We will program a Spartan2E FPGA on a DIO1 programming/testing board.
In order to generate a programming file, we need to assign physical pins to each input/output port we declared in VHDL. We will use switches **SW1, SW2, and SW3** on DIO1 as inputs **“a”, “b”, “cin”**, respectively. LEDs **LD1** and **LD2** on DIO1 will be used as outputs **“sum”** and **“cout”**. The following table provides the mapping of the pins from DIO1 to FPGA.

VHDL Signal	DIO1 Net	Required FPGA Pin Assignment
a	SW1	P126
b	SW2	P129
cin	SW3	P133
sum	LD1	P154
cout	LD2	P161

Table B.1

To assign physical pins to our VHDL signals in Xilinx ISE, go to “Project” -> “New Source”. Click “Implementation Constraints File” and type in the file name (here we use “*fulladder_constraint*”)
Click Next.
Click Next
Expand “User Constraints” in the Processes window and then double click “Assign Package Pins”.
Xilinx PACE should be started. Referring to Table B.1, type in the FPGA pin location as shown in Figure B.1.

	I/O Name	I/O Direction	Loc	Bank	I/O Std.
	sum	Output	P154	BANK2	
	cout	Output	P161	BANK1	
	cin	Input	P133	BANK2	
	b	Input	P129	BANK3	
	a	Input	P126	BANK3	

Figure. B1

Save and then close PACE.

Now it’s time to implement our design into a more detail level. Double click

Make sure fulladder.vhd is selected in the Sources window, and double click “Implement Design” in the Processes window.

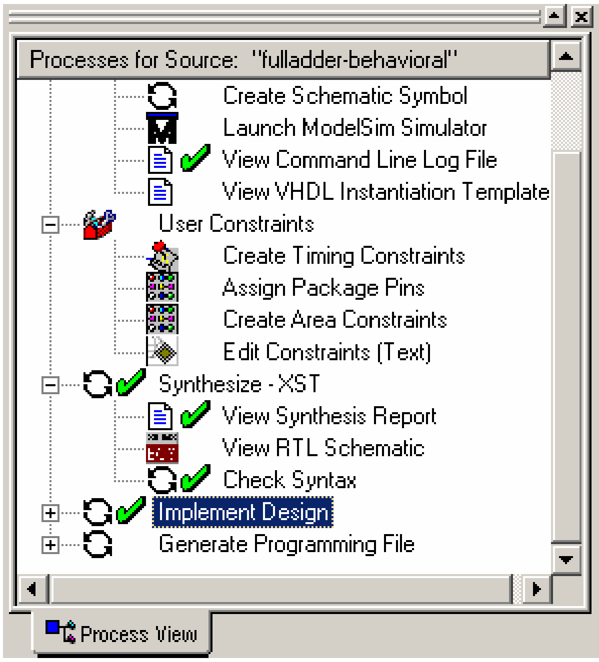

Figure B.2

If everything goes right, you should see a green check on “Implement Design”

Now that we have a design implementation that specifies pin assignments we can generate a programming file. Double click “Generate Programming File”

After this, there should be a file “*fulladder.bit*” created in your lab directory.
At this point, make sure the power cord of D2E board is plugged and the parallel cable is connected to your PC parallel port. Also, SW1 of D2E should be switched to **JTAG.**
In the Processes window, expand “Generate Programming File” and double click on “Configure Device.”
The iMPACT window will show up:
Select Configure Devices and click Next.
Select Boundary-Scan Mode and click Next.
Select “Automatically connect to cable and identify…” and click Finish.

Your board and chip should be automatically identified. The program will then ask you to select a configuration file. Select “fulladder.bit” we just created and click Open.
If a message pops up asking “A BIT file describing…Are you sure you want to do this?” click Yes.
Skip the warning message, if any shows up.
Right click on the Xilinx device in iMPACT window and select “Program”:
Click OK to download the bit stream into the FPGA
Once the FPGA has been programmed, test the functionality by changing the switches and observing LEDs **LD1** and **LD2**.

Appendix C: Troubleshooting the Xilinx Software

Most of the labs will involve implementing a digital design using a hardware description language – Verilog or VHDL – and then simulating the design for testing. We will use *Xilinx ISE Project* Navigator for HDL coding and synthesis, and *ModelSim* for simulation.

All the computers in Halligan 120 have *Xilinx ISE* and *ModelSim* installed.

<u>Xilinx ISE Project Navigator</u>

To run Project Navigator click Start > Programs > Xilinx ISE 6 > Project Navigator

<u>ModelSim</u>

ModelSim will usually be run from the *Project Navigator*. *ModelSim* does require a separate license. Before you can run *ModelSim* (if you have never run it before) you will need to run the Licensing Wizard:

1. Click Start > Programs > ModelSim XE II > Licensing Wizard
2. Click “Continue”
3. Enter the location of the license file as
C:\Modeltech_xe_starter\win32xoem\license.dat
Click “OK”
4. Wizard should ask if it can add an environment variable – click “Yes”
5. Environment variable is added – click “OK”
6. Rerun the Licensing Wizard.
7. Click “Continue”
8. Verify that the license file location is still
C:\Modeltech_xe_starter\win32xoem\license.dat
Click “OK”
9. A notice should pop up saying “A perpetual license was found” – click “OK”
10. Click “Close.” You should now be able to run ModelSim.

Note: If you would like to have the software at home, you can download both *Xilinx ISE* and *ModelSim* for free from the Xilinx website (www.xilinx.com).

1. Click on “Products & Services”
2. Under “Design Resources” click on “ISE Design Tools”
3. Click on “ISE WebPACK”
4. Click on “Register”
5. Click “Create an Account” and follow the instructions on obtaining a username and password. Requires confirmation emails etc.
6. Repeat steps 1 to 3, now click on “Download.” This should take you to a page where you can download both “Complete ISE WebPACK Software” (for *Project Navigator*) and “Complete MXE Simulator” (for *ModelSim*).

7. Install both packages.
8. You will need to obtain a license to run *ModelSim*. To do so, click
Start > Programs > Modelsim XE II > Submit License Request
Follow online instructions.