B38DF Lab 1 – Introduction to Verilog

# Introduction

A custom digital logic design can be implemented in a programmable logic device. Schematic entry can be used, where logic gates were drawn and connected as required. While this method works, it is inefficient in terms of development time.

In industry, digital design is not done in this way – instead, a *hardware description language* is used. There are two industry-standard languages:

* VHDL
* Verilog

The choice of language is a matter of preference. This course shows students how to use the Verilog language. The syntax of VHDL is different, but essentially it does the same thing.

This labsheet guides users in the first stages of Verilog design.

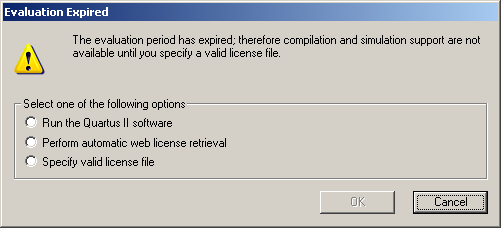
# Open the Quartus Software

Firstly, open the **Quartus II** programme. This is available at:

Start menu -> EPS Programs -> Quartus II -> Quartus II

The icon is: 

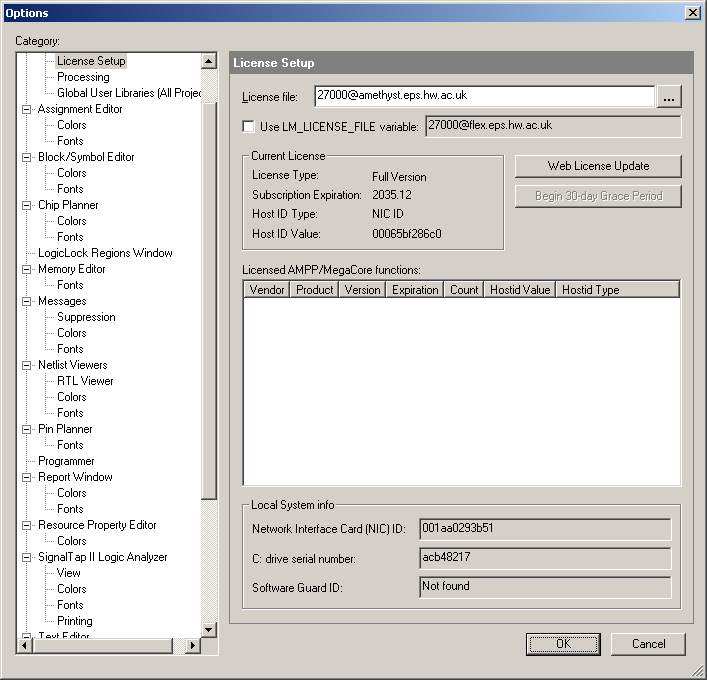
On a small number of machines, a dialogue box may appear at this point stating that an evaluation period has expired, as shown below. **Note that this should not happen on the majority of machines, and if this window does not appear, proceed now to the next section on ‘Starting a New Project’.**



If this does appear, select the option of ‘Specify valid licence file’ and click ‘OK’. When the next window appears, enter:

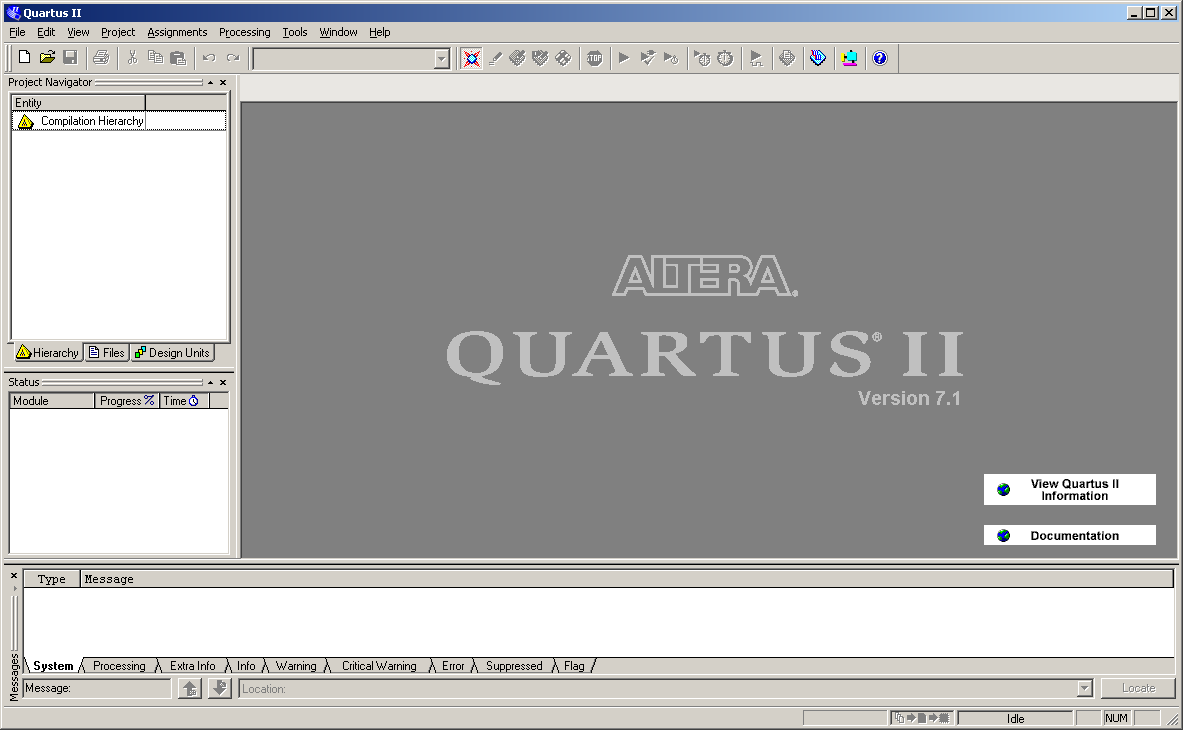
27000@amethyst.eps.hw.ac.uk

into the box at the top of this window, as shown below. Click OK.



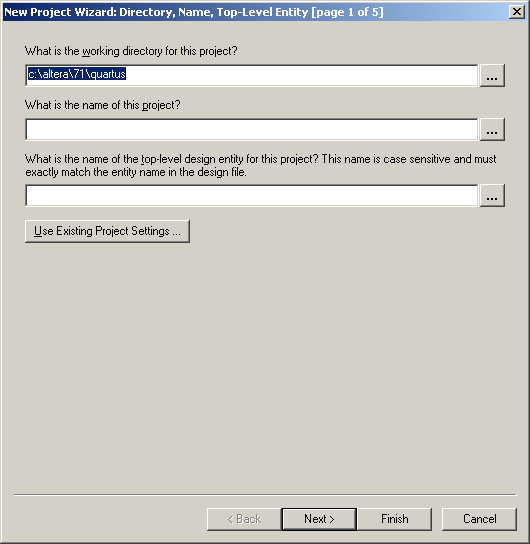
# Start New Project

The screen should now appear as below:



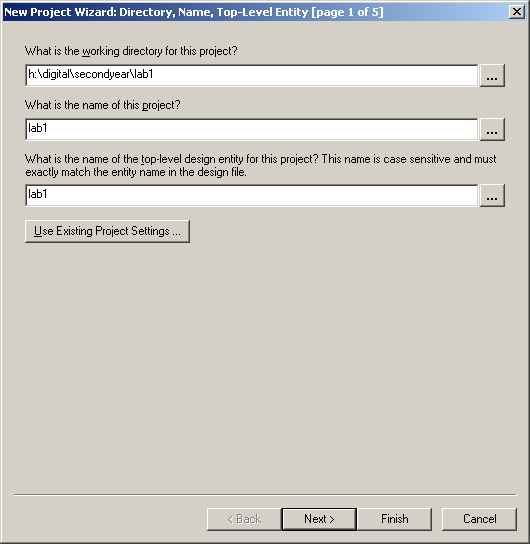
The first thing to do is to create a new project: File -> New Project Wizard

The window below appears:



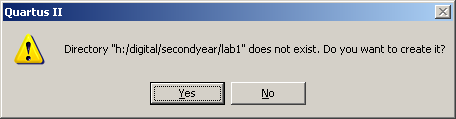
**IMPORTANT: It is not possible to save a project to the C or D drives. When Specifying the working directory, you must change this from the default to a location on your H:/ drive.**

Enter a name for the project, and the top part of the window should be similar to that below:



Click ‘Next’

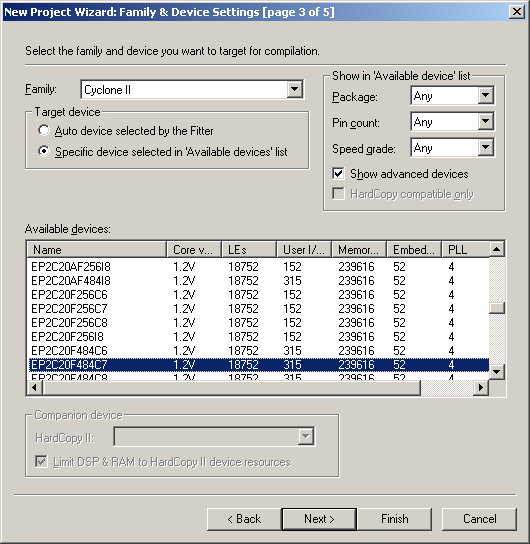
The following window appears if the directory does not already exist:



Click ‘Yes’ to create it.

At the next screen where you are prompted to add any files, simply click ‘Next’ to move on.

The following screen asks you to select the device that the design will be downloaded to. For this module, all designs will be downloaded to the DE1 board. The digital chip on this board is from the Cyclone II family, and the device is ‘**EP2C20F484C7**’. Set these as shown below:



Click ‘Next’

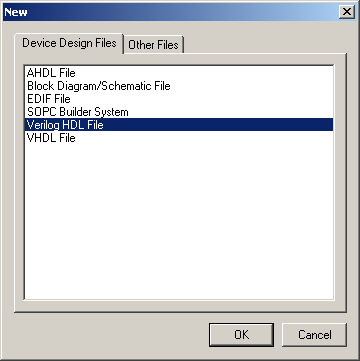
On the following screen, click ‘Next’, then at the last screen click ‘Finish’.

# Create New Verilog Source File

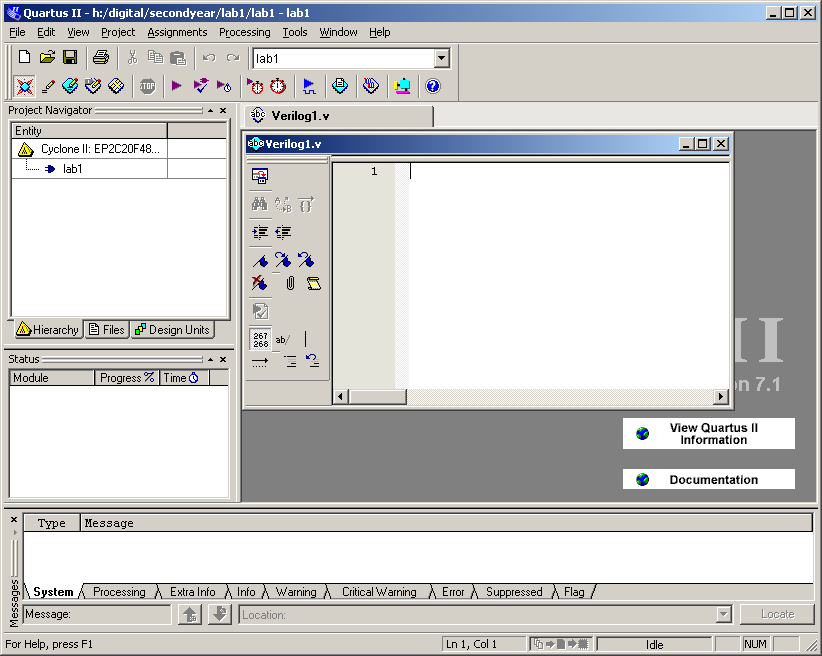
The project is now created, and now a file that will contain the digital design is needed.

Open a new file by selecting: File -> New

In the window that appears, select ‘Verilog HDL File’, as shown below. Click ‘OK’.



Within the main Quartus window, a new window appears, into which the Verilog code may be entered:



At present this file has the default name ‘Verilog1.v’. Before entering any text, save this file:

File -> Save

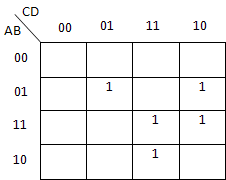
Accept the default name of ‘lab1’, and click ‘Save’. Code may now be entered as described in the following sections.

# First Verilog Design

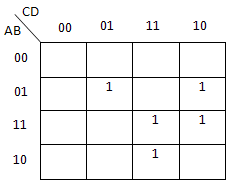
We wish to design a combinational circuit. This has four inputs (A,B,C,D) and one output (Z). The desired truth table is shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | C | D | Z |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

The conventional method is to plot a Karnaugh map based on the truth table, as illustrated below:



The next stage is to group the 1s:



B.C./D

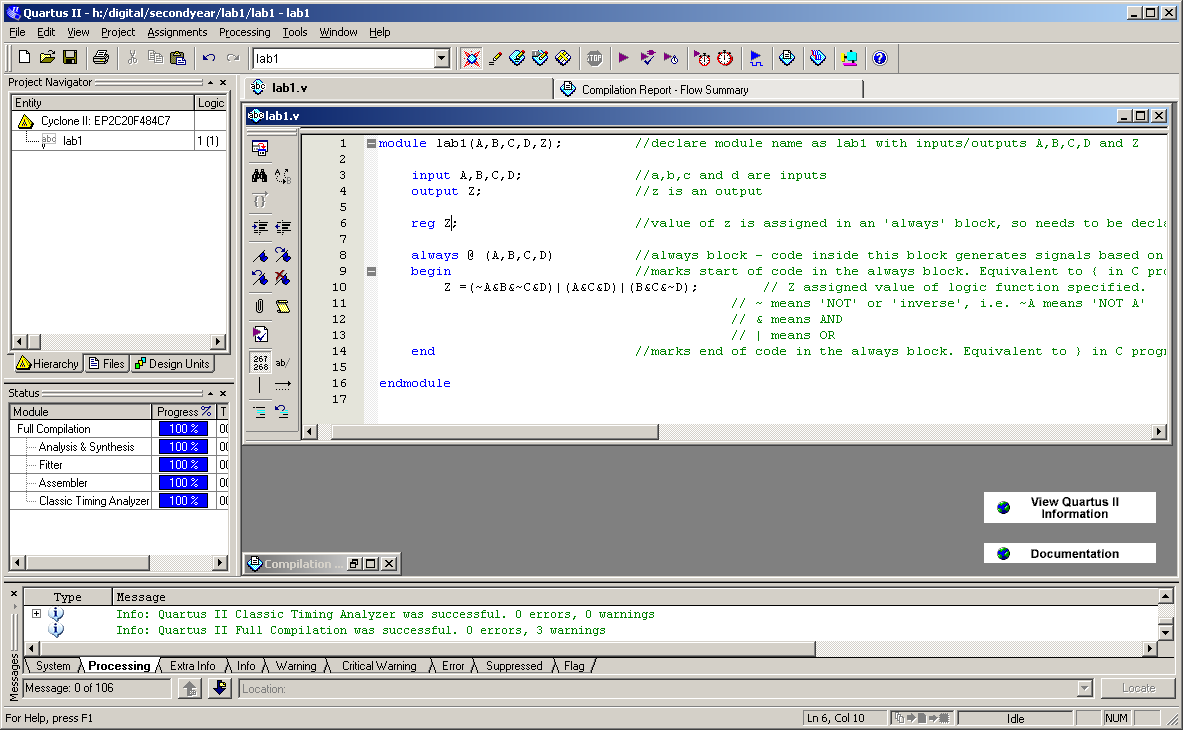
A.C.D

/A.B./C.D

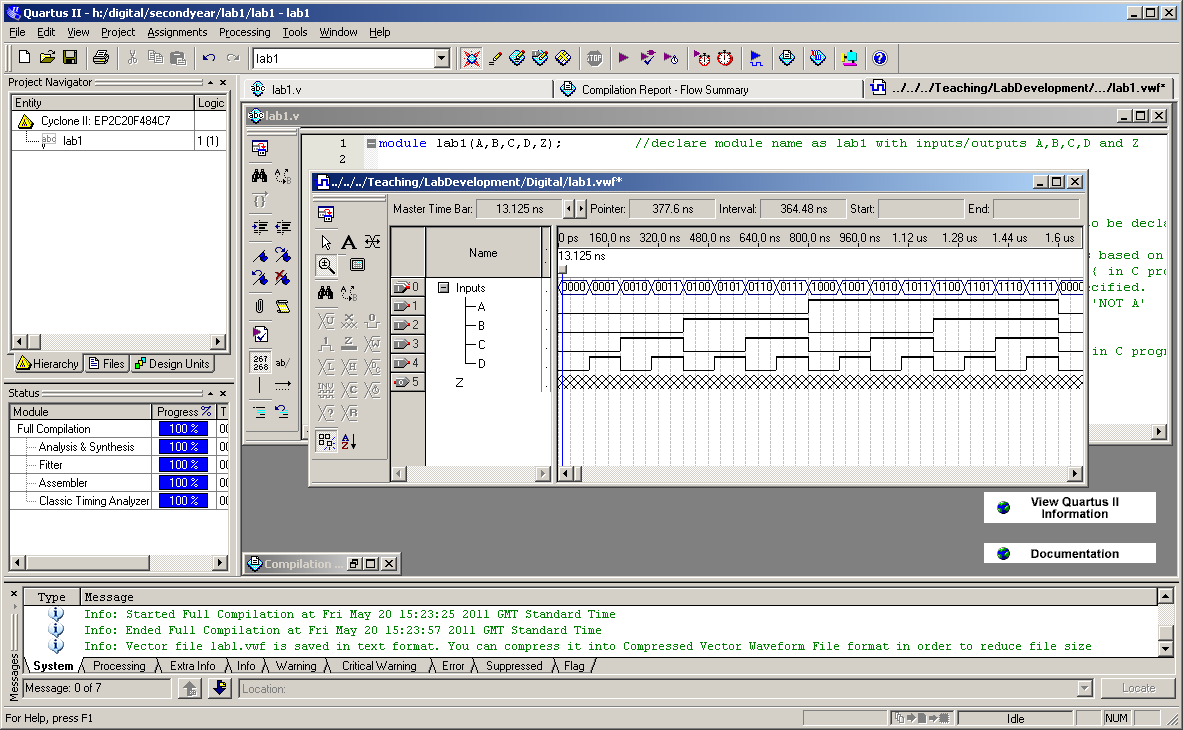
So logic expression is: Z=(/A.B./C.D)+(A.C.D)+(B.C./D)

Verify that this expression is correct.

Code has been pre-written to accomplish this. Open the text file ‘lab1\_part1.txt’, available from the course Vision page and appended at the end of these notes. **Open this file. Select all of the text, copy it, and paste it into the Verilog source file, lab1.v** which should now appear as follows:



Keywords appear in blue. Comments are indicated by typing ‘//’, and it can be seen that comments appear in green. Look over the code to see how it works, and check that it implements the correct logic function. Compile the design by clicking the start compilation button from the top bar:



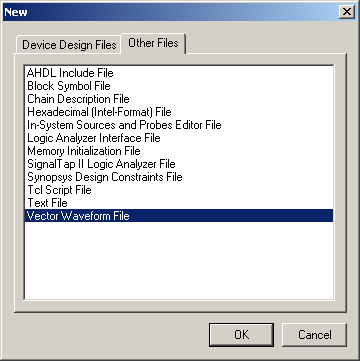
After some delay, a message should appear indicating that compilation is complete. Click ‘OK’.

# Simulating the First Design

To test for correct functionality, it is necessary to simulate the design. To do this, follow the steps below.

Open a new simulator file: File -> New

At the top of the window that appears, select the ‘Other Files’ tab, and in the list that appears, select ‘Vector Waveform File’, as shown below.

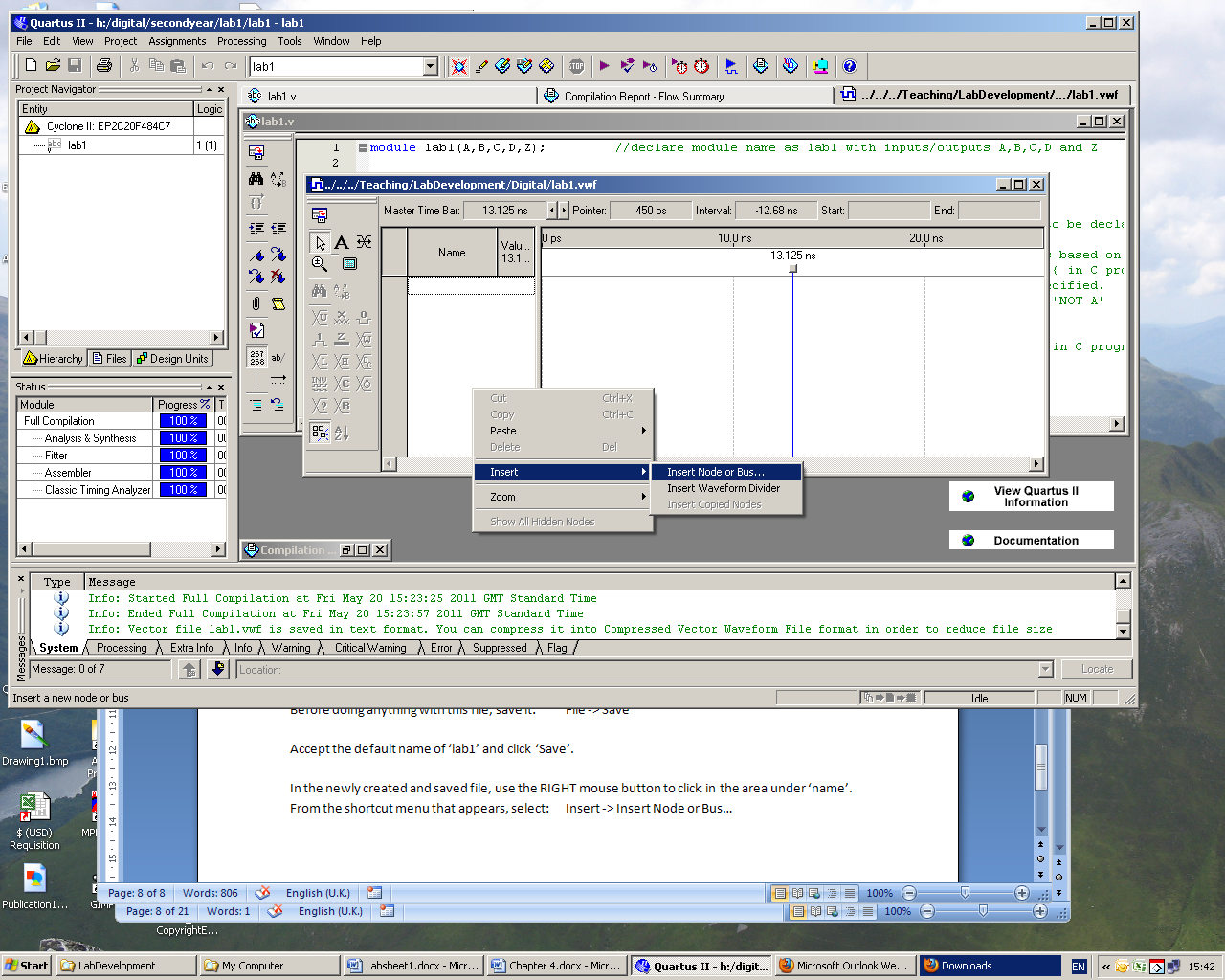


Before doing anything with this file, save it: File -> Save

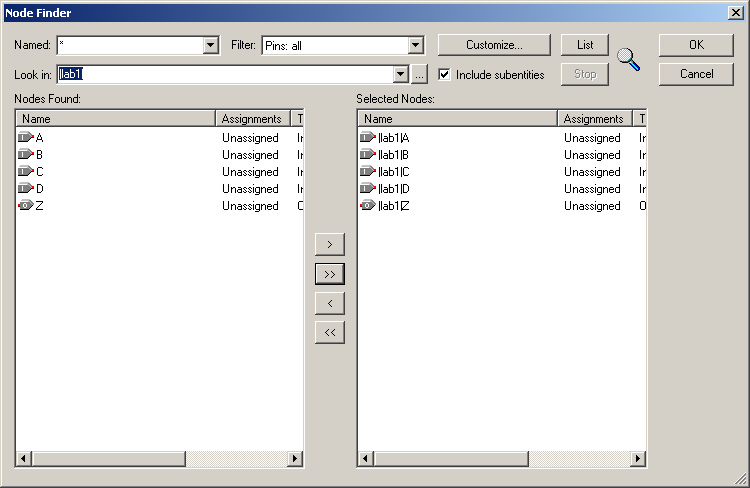
Accept the default name of ‘lab1’ and click ‘Save’.

In the newly created and saved file, use the RIGHT mouse button to click in the area on the left hand side of the window under ‘name’. From the shortcut menu that appears, select: Insert -> Insert Node or Bus…

This is shown below.

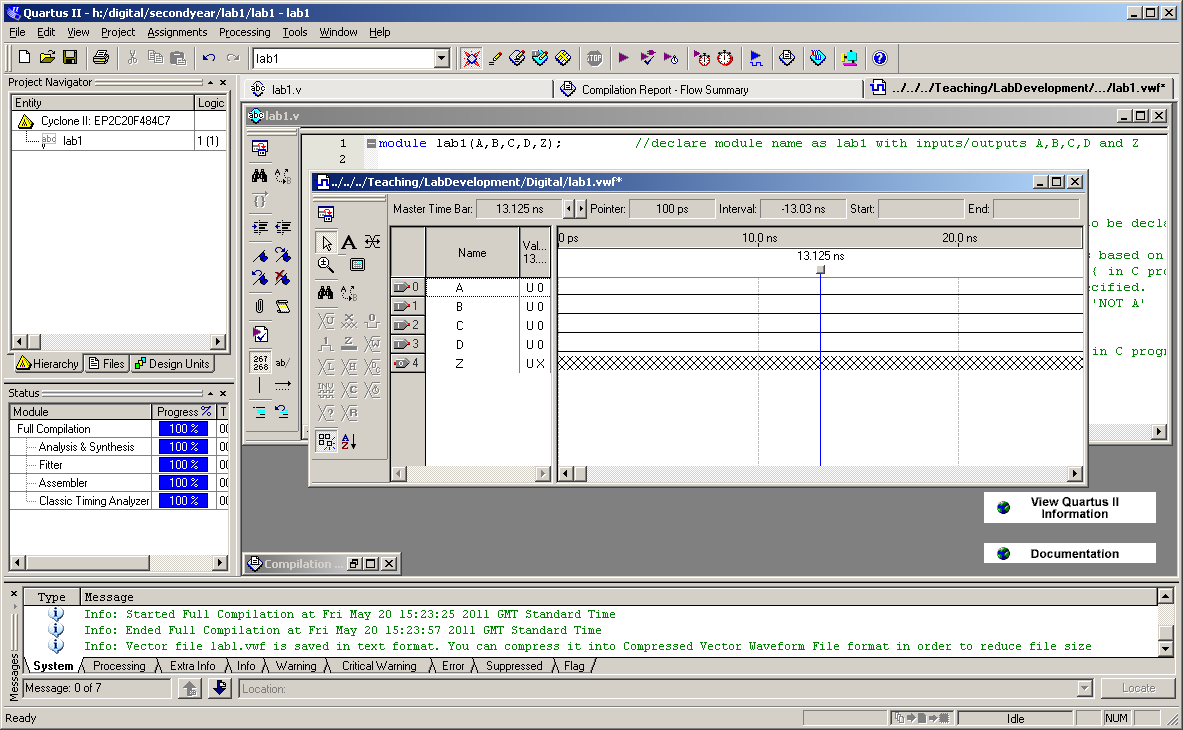


In the window that appears, click the ‘Node Finder’ button, then in the next window, click the ‘List’ button towards the top right corner. All signal names in the design should appear on the left hand side. In the middle of the window, click the ‘>>’ button, which should copy all of the signals over to the right-hand side, as shown below:



Click the ‘OK’ button.

Click ‘OK’ in the ‘Insert Node or Bus’ window that reappears, and the signals should appear in the waveform editor as shown below:



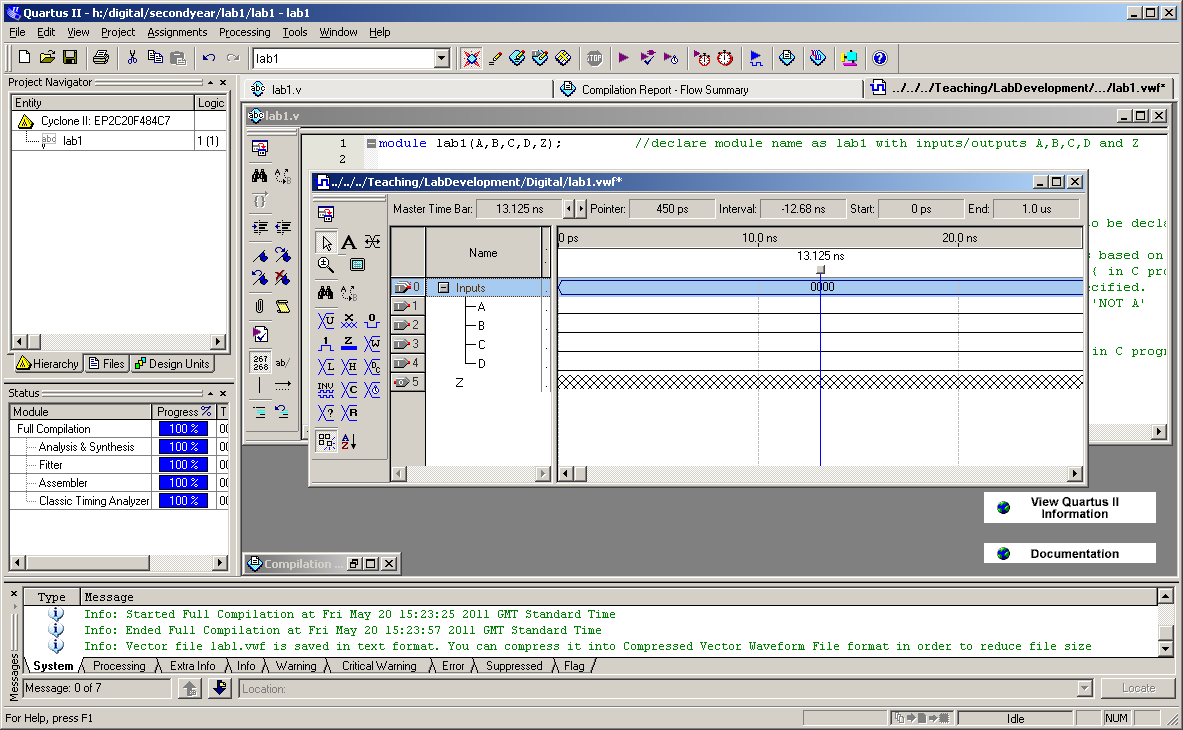
Change the end time to give a longer simulation run: Edit -> End Time…

Change the time to 100us, and click ‘OK’.

Select signals A, B, C and D by clicking each signal name while holding down the ‘ctrl’ key on the keyboard. Press the right mouse button to bring up the shortcuts menu, and select:

Grouping -> Group

Set the group name to be ‘Inputs’ and the radix to be ‘Binary’. Click ‘OK’.

The four signals should now appear as a single group called ‘Inputs’. Click on the ‘+’ sign beside the name ‘Inputs’, and the 4 signals within the group should be displayed. Click on ‘Inputs’ to highlight it. From the group of button on the far left hand side of the window, select the  button. This allows you to apply a count sequence.

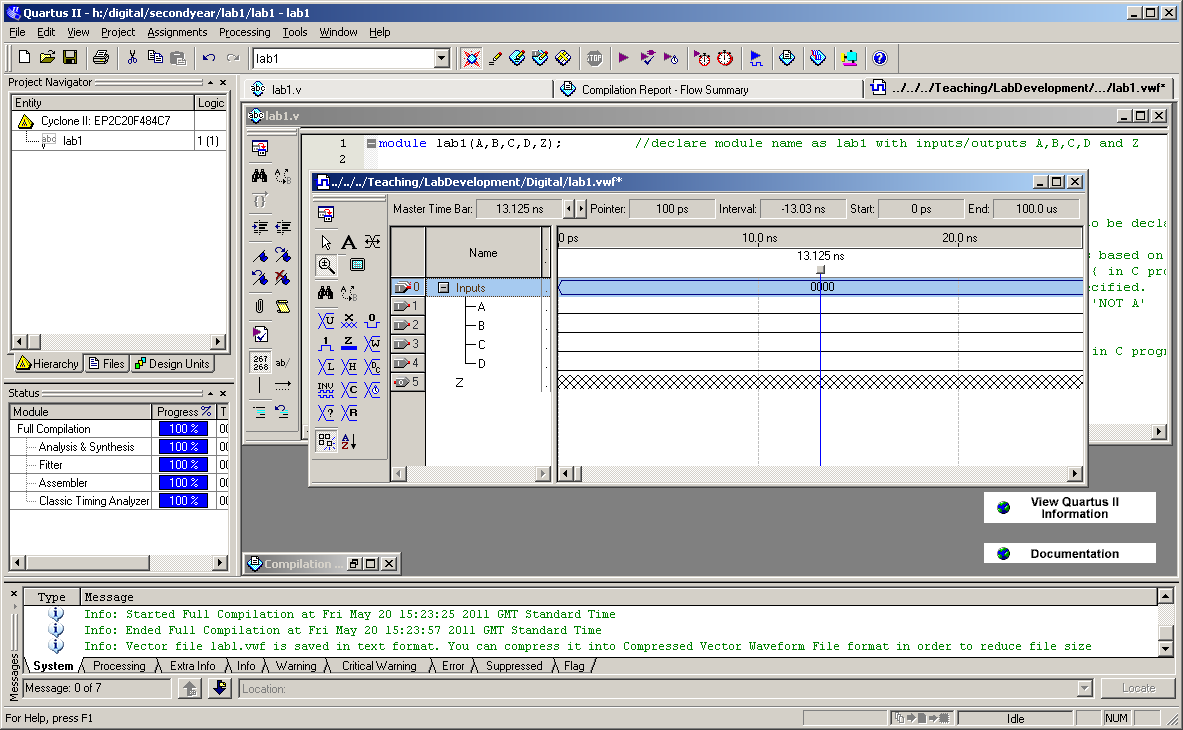
In the box that appears, select:

Radix: Binary

Start Value: 0000

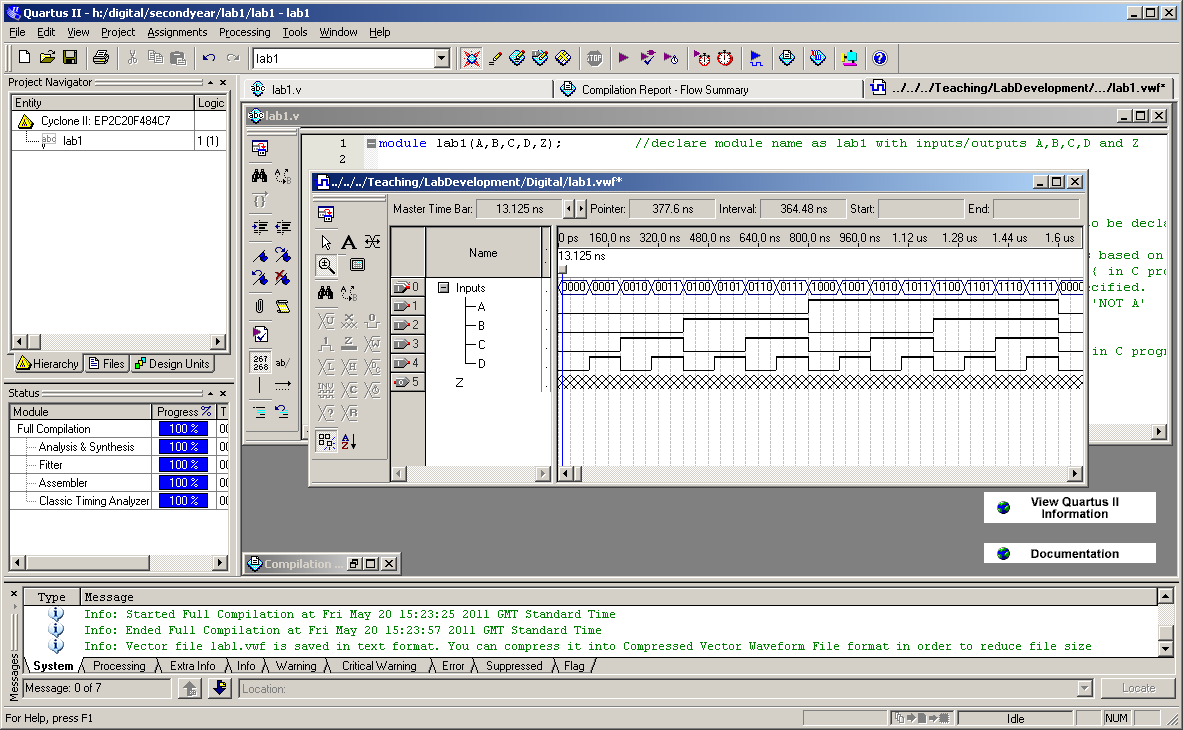
Increment by: 1

Click the ‘Timing’ tab. At the bottom of the window, change the value to count every 100ns. Click ‘OK’.

Select the zoom tool by clicking its button on the left hand side of the simulator window: 

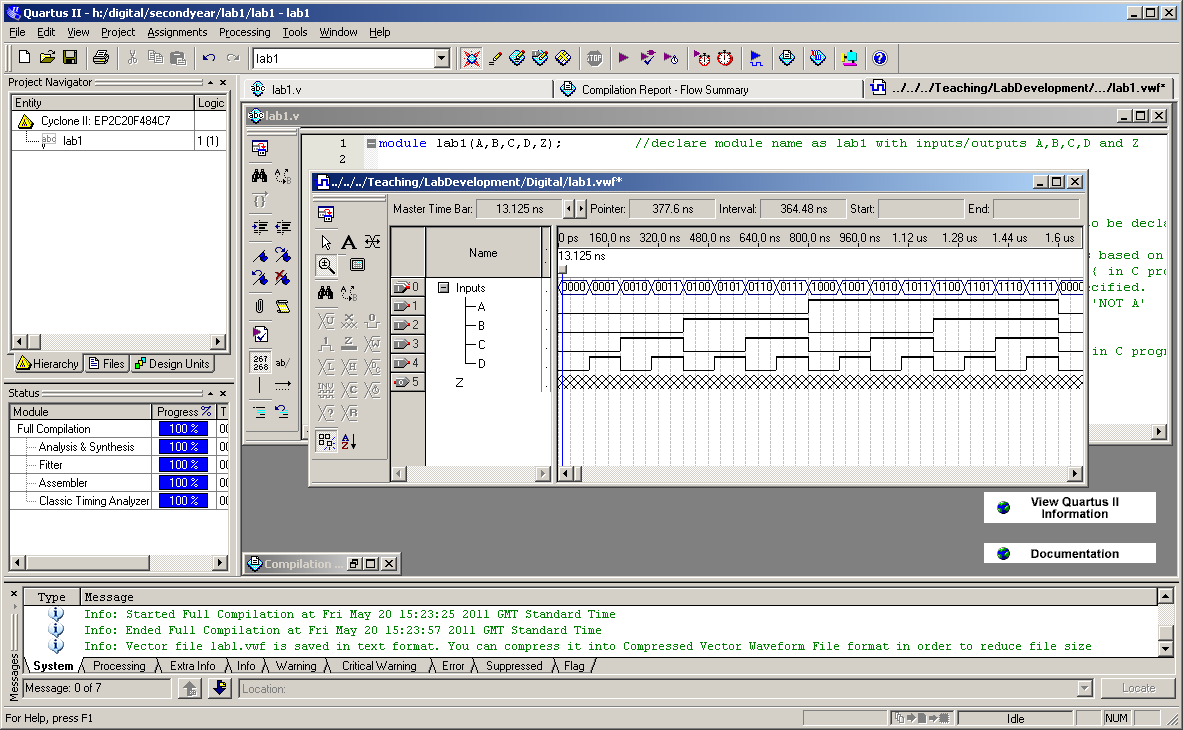
With the zoom tool selected, left-clicking in the main waveform area zooms in, and right-clicking zooms out.

Right-click a few times in the area showing the waveform to zoom out to a suitable scale, where waveforms can be seen:

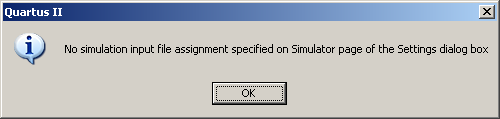


It can be seen that this has set up all possible combinations of inputs from 0000 to 1111. Z is as yet undefined, because this is an output. The purpose of simulating is so that input signals can be set up as desired, the simulation can be run, and the outputs found upon viewing the simulation result.

Before running the simulator, save the waveform file: File -> Save

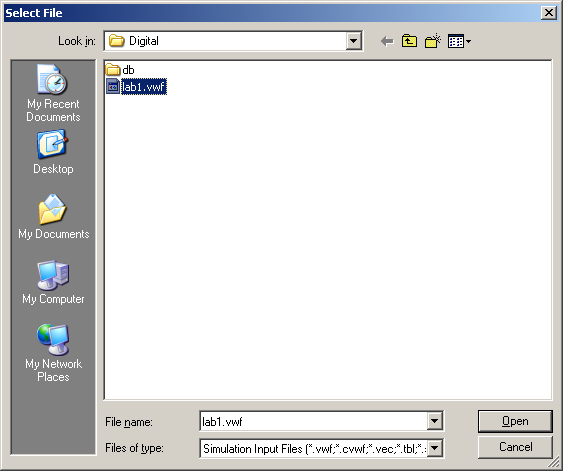
Then click the ‘Run simulation’ button from the top bar: 

The simulation should run, but in some cases an error message appears, as shown below:



If this message does not appear, and the simulator runs normally, proceed to the next section on ‘Checking the Simulation Result’. If the above window does appear, click ‘OK’ , then go to: Processing -> Simulator Tool

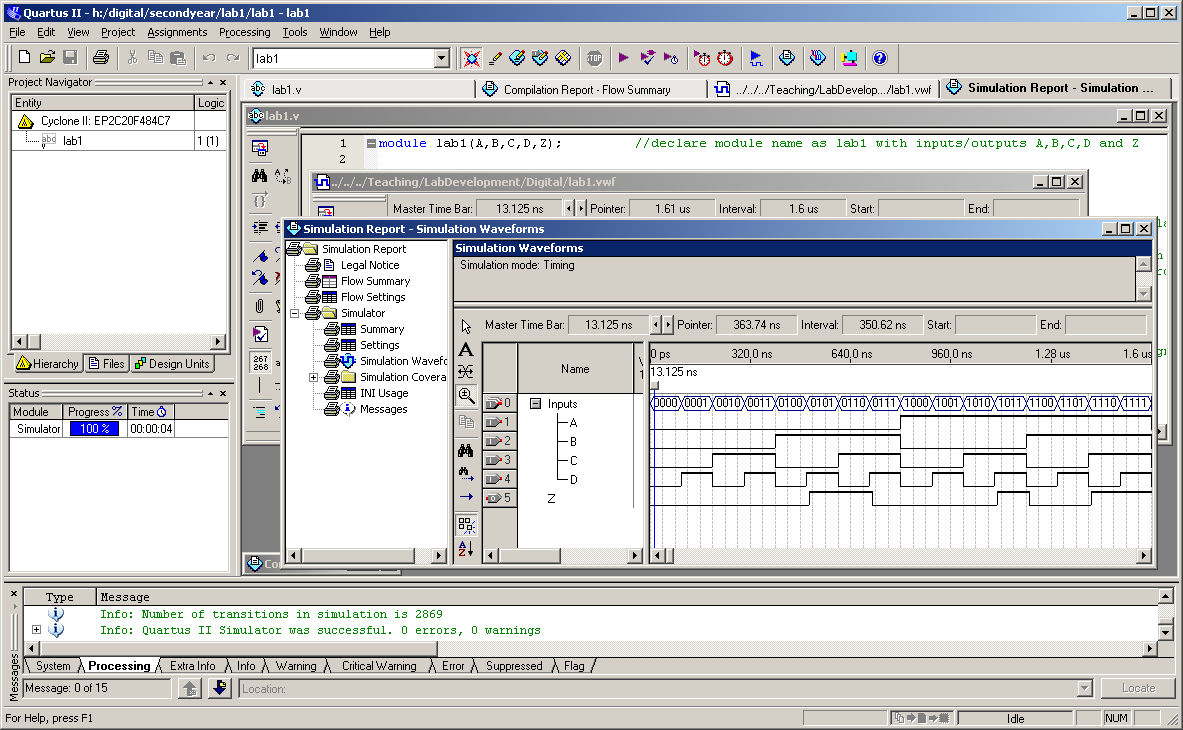
Click the ‘…’ button next to the ‘Simulation Input’ box, and find the directory where the project is saved. Select the appropriate simulation file, as shown below, and click ‘Open’.



Close the Simulator Tool dialogue box. Click the run simulation button again, and the simulator should run this time.

# Checking the Simulation Result

Upon completion of simulation, the window should be similar to that below. It may be necessary to zoom in (left click) or out (right click) with the zoom tool selected to see the waveform correctly.



It can be seen that in the first section of the waveform, inputs A,B, C and D are all 0. At this time, the output Z is also 0. The first line of the truth table for Z can then be filled in as a 0. **Now complete the remainder of the truth table below, based on the other sections of the waveform.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | C | D | Z |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |  |
| 0 | 0 | 1 | 0 |  |
| 0 | 0 | 1 | 1 |  |
| 0 | 1 | 0 | 0 |  |
| 0 | 1 | 0 | 1 |  |
| 0 | 1 | 1 | 0 |  |
| 0 | 1 | 1 | 1 |  |
| 1 | 0 | 0 | 0 |  |
| 1 | 0 | 0 | 1 |  |
| 1 | 0 | 1 | 0 |  |
| 1 | 0 | 1 | 1 |  |
| 1 | 1 | 0 | 0 |  |
| 1 | 1 | 0 | 1 |  |
| 1 | 1 | 1 | 0 |  |
| 1 | 1 | 1 | 1 |  |

**Does the actual truth table found from simulation match the intended truth table on page 5?**

# Modifying the Logic Function

The new logic function to be implemented has the following truth table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| c | B | C | D | Z |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Using the same procedure as before, **derive the correct logic expression. Your working and the logic expression derived should be shown below.**

**Now modify the code in the Verilog source file lab1.v to implement the modified function.** Save it, then compile. Run the simulator again. From the simulation result, complete the truth table obtained from simulation:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | C | D | Z |
| 0 | 0 | 0 | 0 |  |
| 0 | 0 | 0 | 1 |  |
| 0 | 0 | 1 | 0 |  |
| 0 | 0 | 1 | 1 |  |
| 0 | 1 | 0 | 0 |  |
| 0 | 1 | 0 | 1 |  |
| 0 | 1 | 1 | 0 |  |
| 0 | 1 | 1 | 1 |  |
| 1 | 0 | 0 | 0 |  |
| 1 | 0 | 0 | 1 |  |
| 1 | 0 | 1 | 0 |  |
| 1 | 0 | 1 | 1 |  |
| 1 | 1 | 0 | 0 |  |
| 1 | 1 | 0 | 1 |  |
| 1 | 1 | 1 | 0 |  |
| 1 | 1 | 1 | 1 |  |

**Does this match what was intended?**

1. Behavioural Design

The previous work used structural design – design where the logical expressions were worked out, and these implemented. Verilog also allows a higher level design method to be used, and this is behavioural design.

When designing a digital circuit behaviourally, it is only necessary to specify how the circuit should respond under certain circumstances.

Example: a system has one input (A) and 3 outputs (X,Y and Z). When A is 0, the user wants X to be 0, Y to be 1 and Z to be 0. When A is 1, X should be 1, Y should be 0 and Z should be 1.

Of course we could quite simply work out the required logic expressions, but it is not necessary to do so. This can be implemented simply, using the following code:

module lab1a (A,X,Y,Z);

if (A==0)

begin

X=0;

Y=1;

Z=0;

end

else

begin

X=1;

Y=0;

Z=1;

end

**Start a new project and create a new Verilog design file, lab1a.v** as in sections 3 and 4. Open the text file ‘lab1\_part2.txt’ included at the end of these notes or on Vision. If from Vision , copy all of the text into the newly created Verilog file lab1a.v

Save the Verilog design file and compile. Open a new simulator file, and add all signals, as in section 6. Set the input as necessary, then run the simulator. Are the results as intended?

# Modified Behavioural Design

Modify the code such that it will accept TWO inputs, A and B, and give FOUR outputs, W, X, Y and Z. A behavioural design should be used, and the outputs should be as follows, for each combination of inputs:

If ((A==0) & (B==0)) - all outputs low

If ((A==0) & (B==1)) - W and X low, Y and Z high

If ((A==1) & (B==0)) - W and X high, Y and Z low

If ((A==1) & (B==1)) - W and Z high, X and Y low

Use the following template:

if ((condition1)&(condition2))

begin

…

…

end

else if ((…………..)&(…………))

begin

…

…

end

else if …………………………………..

Save and compile. If compilation is not successful, errors will be reported. Resolve any errors, and compile again. Upon successful compilation, run the simulator, and check that the outputs are as intended for each combination of inputs.

# Report

**Submit a full report for Lab1 based on the report template given.**

**Prewritten Lab1\_part1.txt text file to be copied into the Verilog source file and saved as lab1.v**

**--------------------------------------------------------------------------------------------------------------------------------------------------------**

module lab1(A,B,C,D,Z); //declare module name as lab1

//with inputs/outputs A,B,C,D and Z

input A,B,C,D; //a,b,c and d are inputs

output Z; //z is an output

reg Z; //value of z is assigned in an 'always' block,

// so needs to be declared as data type 'reg

always @ (A,B,C,D) //always block - code inside this block

// generates signals based on (A,B,C,D)

begin //marks start of code in the always block.

// Equivalent to { in C programming

Z=(~A & B & ~C & D) | (A & C & D) | (B & C & ~D); // Z assigned value of logic function specified.

// ~ means 'NOT' or 'inverse', i.e. ~A means 'NOT A'

// & means AND

// | means OR

end //marks end of code in the always block.

//Equivalent to } in C programming

endmodule

**Prewritten Lab1\_part2.txt text file to be copied into the Verilog source file and saved as lab1a.v**

**--------------------------------------------------------------------------------------------------------------------------------------------------------**

module lab1a (A,X,Y,Z);

input A;

output X,Y,Z;

reg X,Y,Z;

always @ (A)

begin //start of always block

if (A==0)

begin

X=0;

Y=1;

Z=0;

end

else

begin

X=1;

Y=0;

Z=1;

end

end

endmodule