

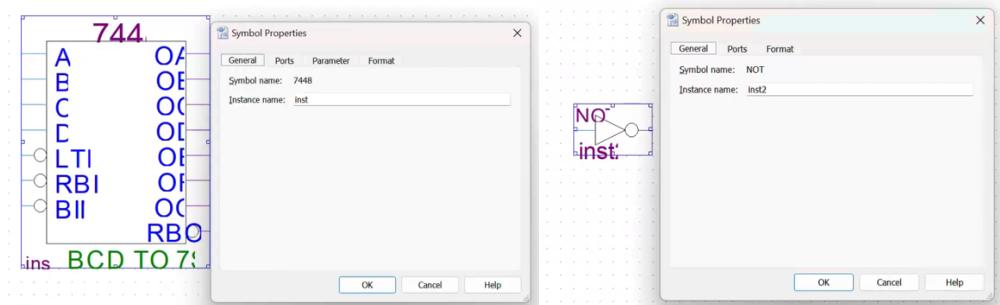
## **Altera Experiment**

### Contents

<b>1. Introduction .....</b>	<b>3</b>
<b>2. Schematic Capture .....</b>	<b>3</b>
2.1.Start the Altera software: .....	4
2.2.Using the Graphics Editor .....	6
2.3. Drawing Your Schematic .....	8
2.4.Compiling .....	9
2.5.Waveform Entry for Simulation .....	11
2.6.Simulation.....	15
2.7.Programming and Testing.....	16
<b>3. AHDL combinational design .....</b>	<b>21</b>
3.1.Boolean Equations .....	21
3.2.Logical Operators .....	21
3.3.Subdesign.....	22
3.4.Implementing Boolean Expressions & Equations .....	22
3.5.Declaring Nodes .....	23
3.6.If Then Statement Logic .....	23
3.7.Case Statement Logic .....	24
3.8.Creating Decoders .....	24
3.9.Decoder Implementation .....	25
<b>4. AHDL sequential design.....</b>	<b>31</b>
4.1.'D' Flip-Flop with Enable.....	31
4.2.Oscillator .....	31
4.3.Counter design in AHDL .....	31
4.4.Modified counter .....	33
<b>5. Reports.....</b>	<b>35</b>

## Important Tips:

1. Please make sure that the names of your project and folder are the same. Meanwhile, they must be the different names as the *tdf* files (AHDL coding file).
2. Please avoid naming your file starting with a number (except for the AHDL coding file, *tdf* file).
3. The instance name of each symbol should be different.

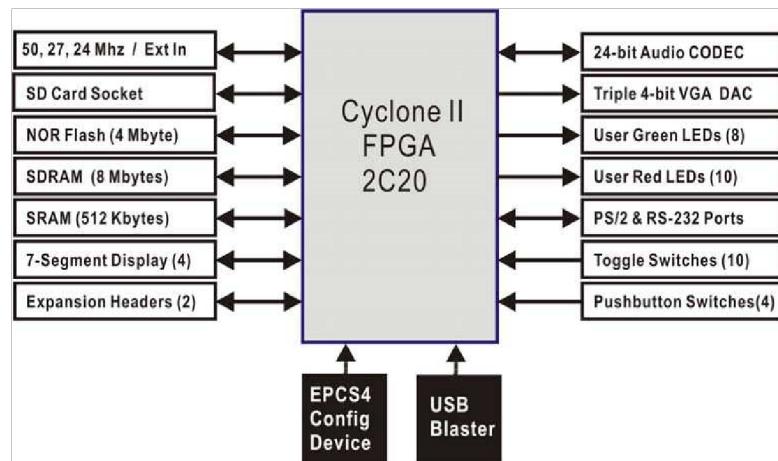


## **1. Introduction**

This experiment will take you through a tutorial on how to use the *Altera Quartus II* package and the *Altera University Programme DE1* board with a Cyclone II FPGA. The experiment is split into 3 parts:

1. The schematic capture elements of the package.
2. AHDL combinational design.
3. AHDL sequential design.

The block diagram of the DE1 board is as follows. To provide maximum flexibility for the user, all connections are made through the Cyclone II FPGA device.



To power-up the board perform the following steps:

1. Connect the provided USB cable from the host computer to the USB Blaster connector on the DE1 board. For communication between the host and the DE1 board, it is necessary to install the Altera USB Blaster driver software.
2. Connect the 7.5V adapter to the DE1 board.
3. Turn the RUN/PROG switch on the left edge of the DE1 board to RUN position.
4. Turn the power on by pressing the ON/OFF switch on the DE1 board.

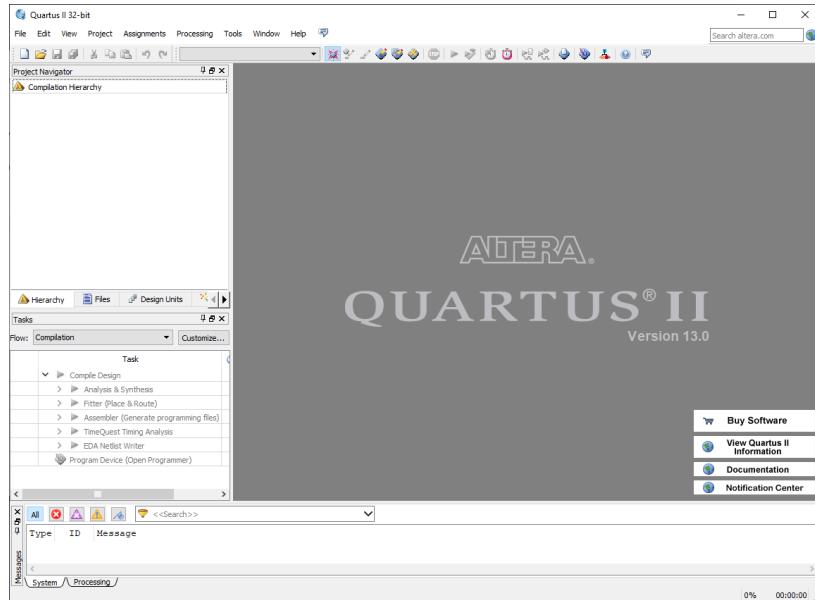
The DE1 board comes with a preloaded configuration bit stream to demonstrate some features of the board. At this point you should observe the all user LEDs are flashing and all 7-segment displays are cycling through the numbers 0 to F.

## **2. Schematic Capture**

Schematic capture is one of the most common methods for entering a design. It involves using a design tool to draw a schematic of the desired circuit. This is probably the easiest method of design capture to understand, but it is not necessarily the most efficient. The Quartus II software includes schematic symbols and models for the most common 74-series TTL IC's. This means that a designer can easily take a schematic or existing design of an existing circuit implemented using TTL parts and enter that schematic into the software. The entire circuit, which may contain dozens of TTL chips on a printed circuit board, can then be compiled into a single PLD package. Although very good for updating old projects, schematic capture is not usually the most efficient way of capturing the behaviour of new designs. For that purpose, a Hardware Description Language (HDL) is probably the more efficient. We will cover the Altera HDL (AHDL) in this experiment.

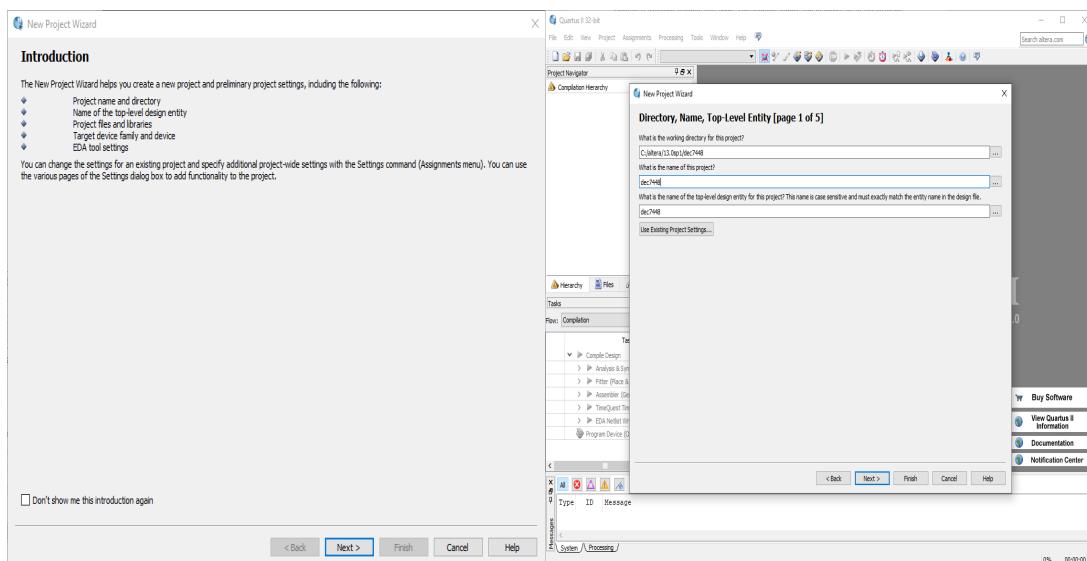
## **2.1.Start the Altera software:**

Select **Start | altera | Quartus II**, or select the Quartus II icon from the desktop. The Quartus II Manager window will open. This window allows you to access all of the different tools that you will be using from one place. The tool bar along the top is basically the same for all applications. Some icons will be greyed out, depending on what windows are in the foreground.

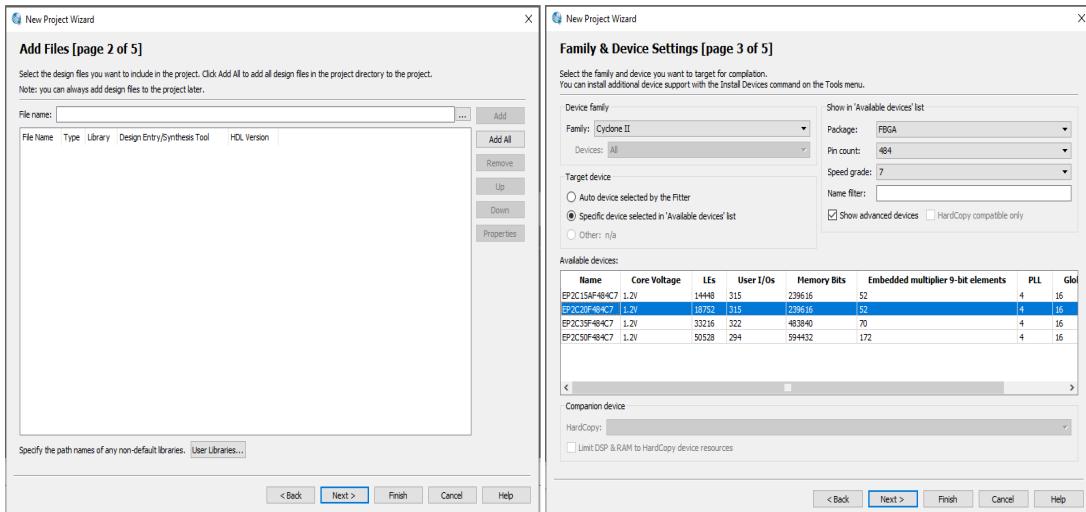


In “My Documents” create a new folder “DE1”.

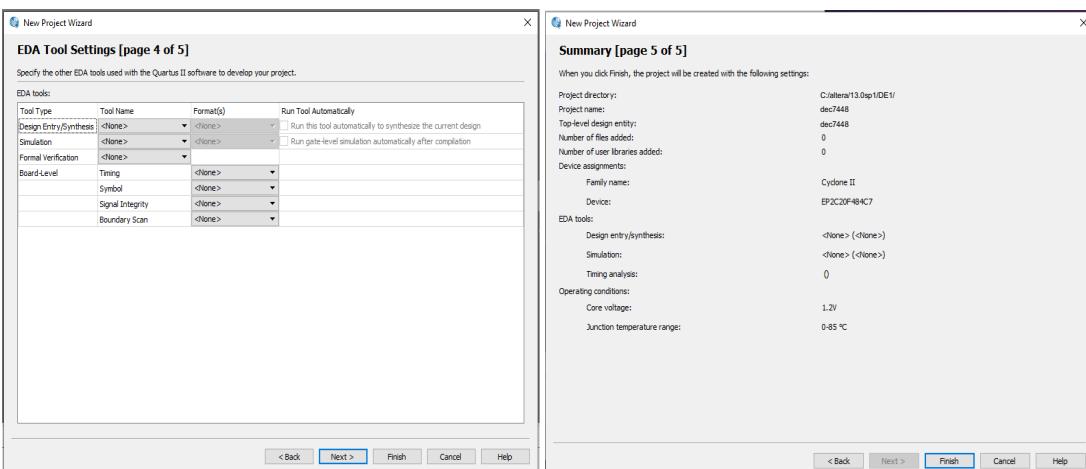
Select **File | New Project Wizard** to create a new project. Click **Next**. (Page 1of 5) In “My Documents”, select “DE1\dec7448” as the working directory which will be created afterwards. Type in “dec7448” as the name of the project. Click **Next**.



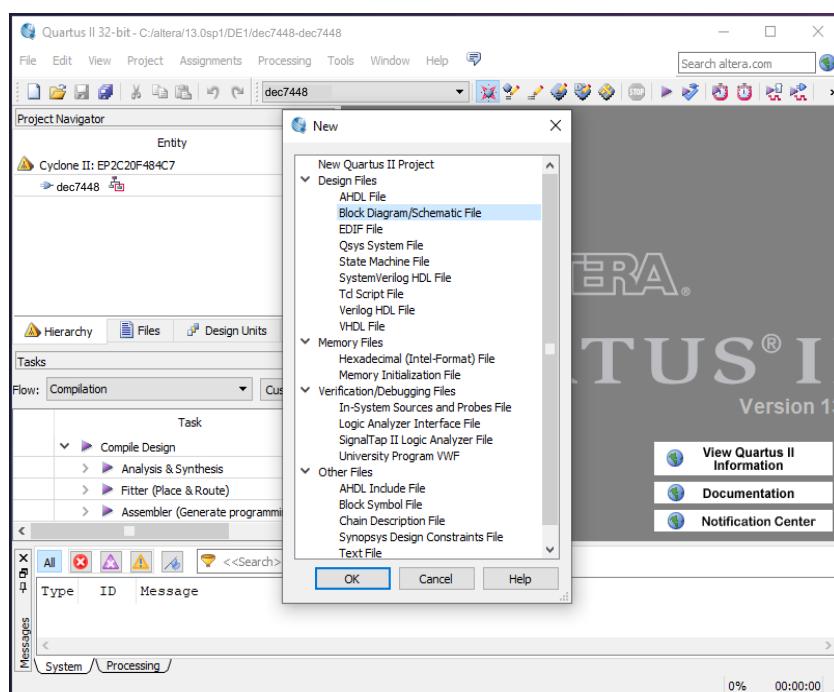
(Page 2of 5) Click **Next**. (Page 3of 5) Select Cyclone II family with the FBGA package, pin count 484, and speed grade 7. Select the EP2C20F484C7 device and click **Next**.



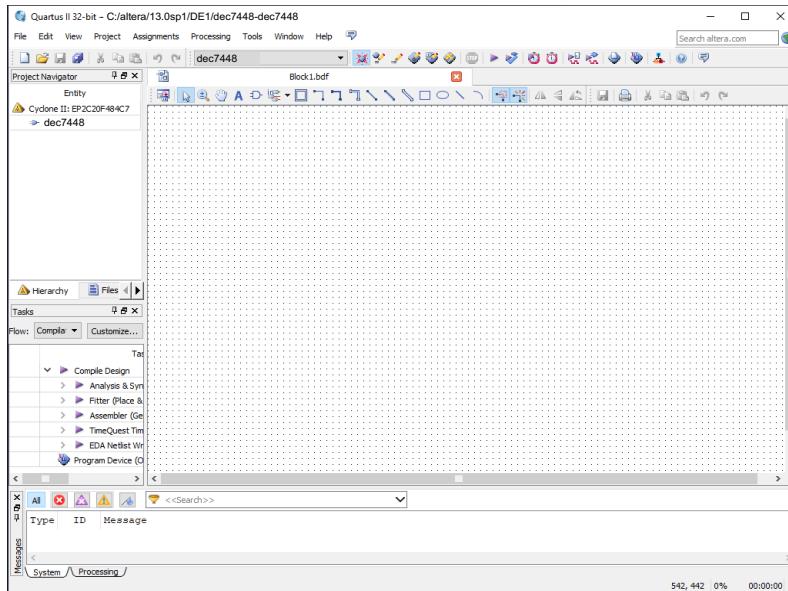
(Page 4 of 5) Click **Next**. (Page 5 of 5) Click **Finish**.



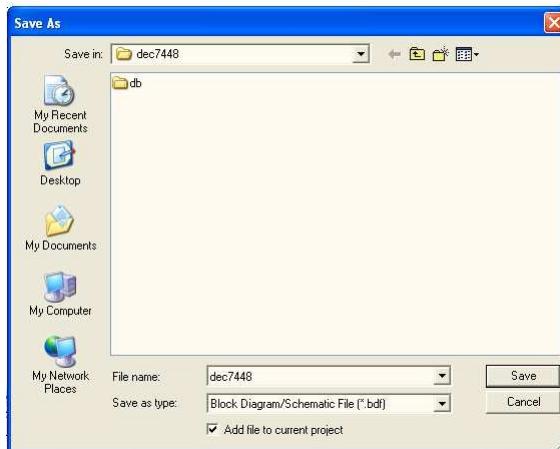
Select File | New... Select Block Diagram/Schematic File. Click **OK**.



The Schematic Editor window will open. Note the new toolbar for the Graphic Editor at the left of the screen.

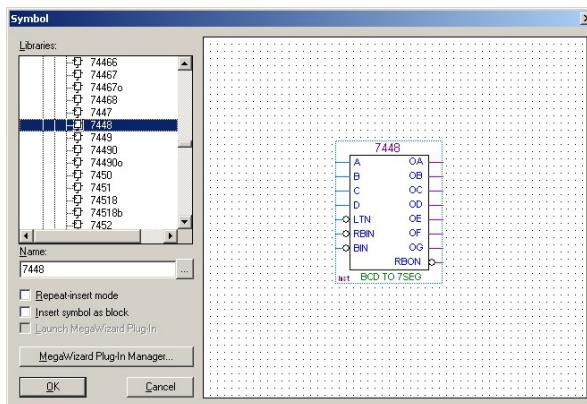


Select **File | Save As**. Select the “dec7448” subdirectory and enter **dec7448** in the *File Name* field. Make sure that the **Add file to current project** check box is selected. Click **Save**.

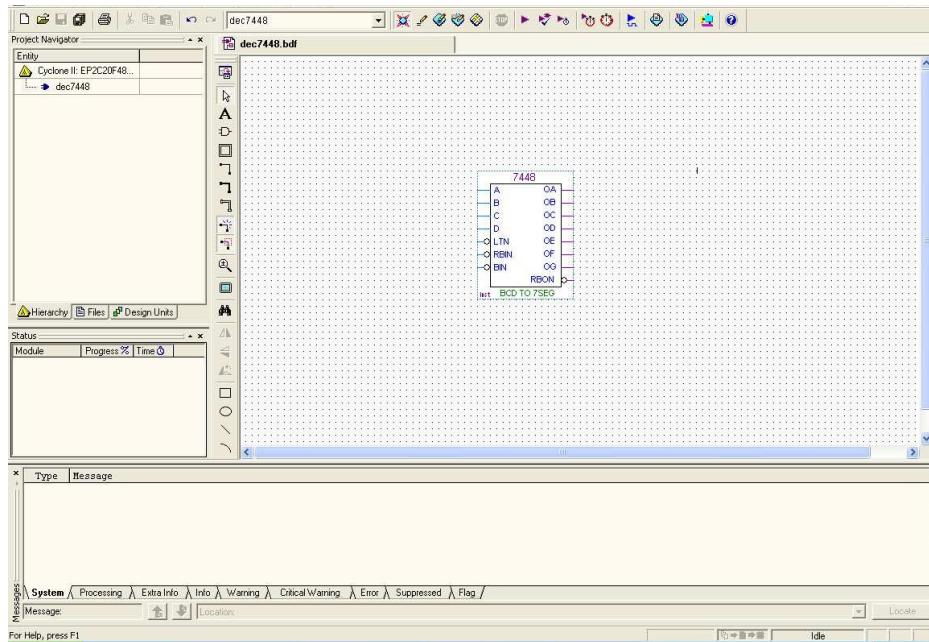


## **2.2.Using the Graphics Editor**

Double click anywhere in the empty Graphic Editor window. The *Symbol* dialog box will open. Expand **others\ maxplus2** library and then select **7448** in the **Libraries** box and click **OK**.

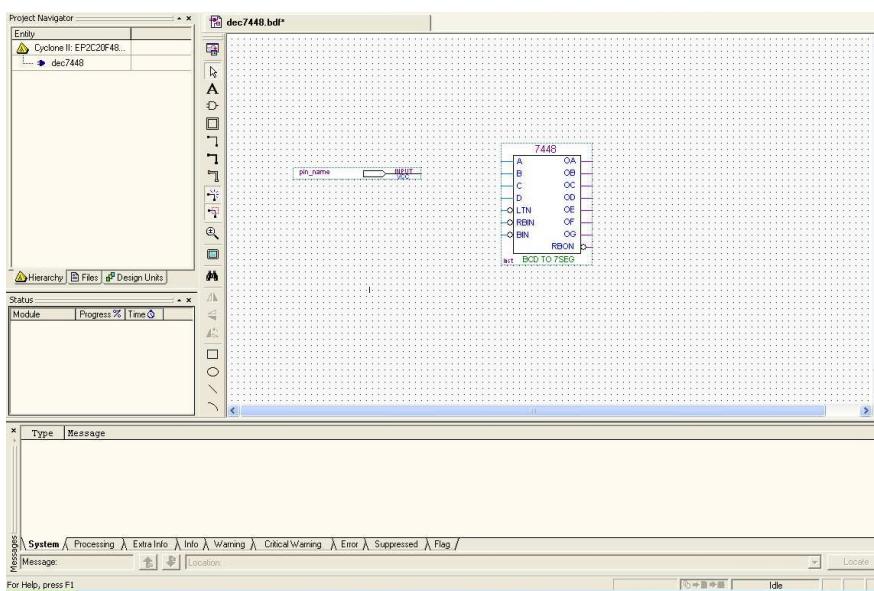


A 7448 symbol will appear in the editor screen.



The Graphics Editor has many built in symbols. It has primitives such as AND gates, NOR gates, etc. It also has *megafunctions*, which are combinations of gates that form higher level objects such as multipliers and dividers. Double click on any empty part of the Graphics Editor window to bring up the *Symbol* dialog box again. In the *Symbol Libraries* window expand the **Primitives** folder to see the different symbols available in the primitives library. Don't select any right now; you can look at the symbols in more detail later, if you want to. Now just click on Cancel and continue with the tutorial.

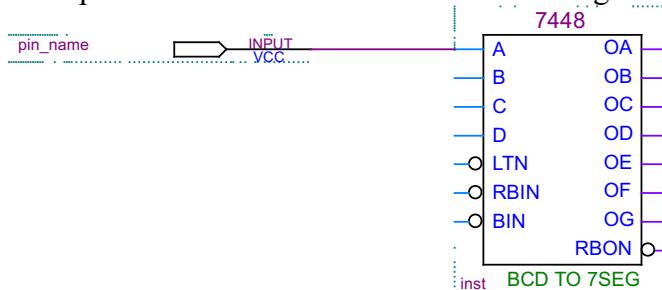
Since we just want to emulate a 7448, most of our work has already been done. Now we need to add inputs and outputs to the drawing. Double click inside the Graphic Editor window, anywhere to the left of the 7448 symbol. You should get the *Symbol* dialog box like before. Enter **input** for the symbol name. An input symbol will appear on the diagram at the point where you double-clicked. The input symbol represents an input to that layer of the design. If the design is the top layer then it represents a physical input pin on the PLD.



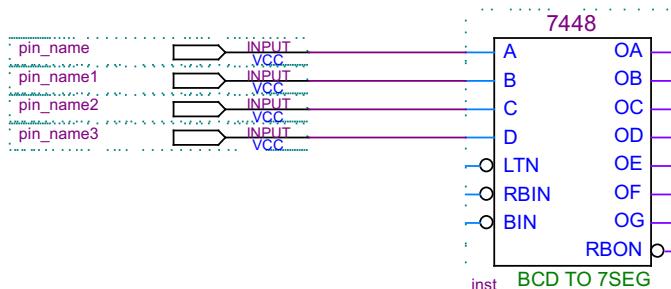
### **2.3. Drawing Your Schematic**

Now we need to connect the input pin of the PLD to one of the input of the 7448. Move the cursor to the right end of the horizontal line in the input pin symbol. The cursor will turn into a cross, indicating that you can attach a wire to that point:

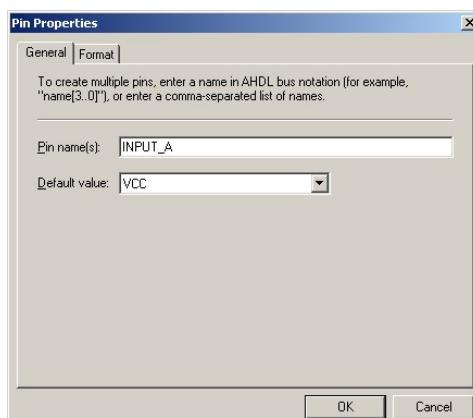
Now click on the right end of the input signal and hold the mouse button. Drag the cursor to the left end of the pin labelled A on the 7448 and release the button. You have now connected the PLD input pin to the input of the 7448. It should look something like this:



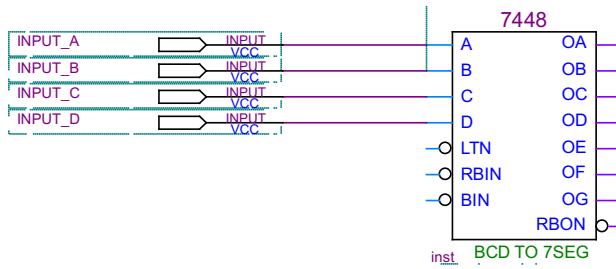
Click on the input pin symbol and drag it so that the wire between it and the pin on the 7448 is straight. The input pin should be selected. Select **Edit | Copy** to copy the pin. Click just below the first pin to select an insertion point and then select **Edit | Paste** to paste in the copy of the pin. (Alternatively, you can hold down the Ctrl key on the keyboard and click on and drag the pin. A copy will be created wherever you dragged to). Move it until it is just below the first pin and aligned with it horizontally. Connect the second pin to the B input. Repeat this to create a third and fourth pin. Connect these to inputs C and D. Your schematic should now look like this:



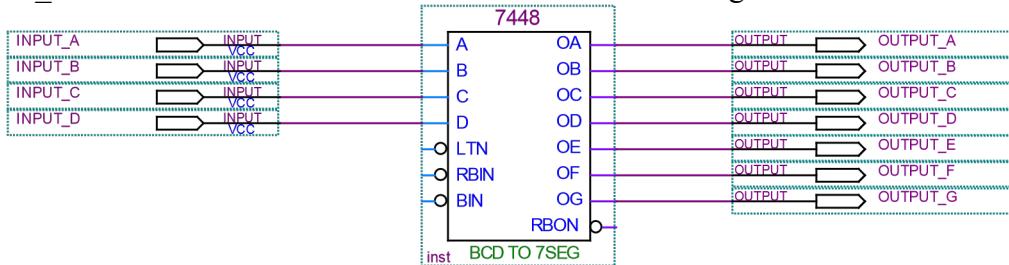
Although the pins on the 7448 are labelled correctly already, the PLD inputs are all labelled **PIN\_NAME**. Double click on the first pin and change the Pin Name to **INPUT\_A**.



Repeat this for the other three pins, naming them **INPUT\_B**, **INPUT\_C**, and **INPUT\_D**, as shown below.

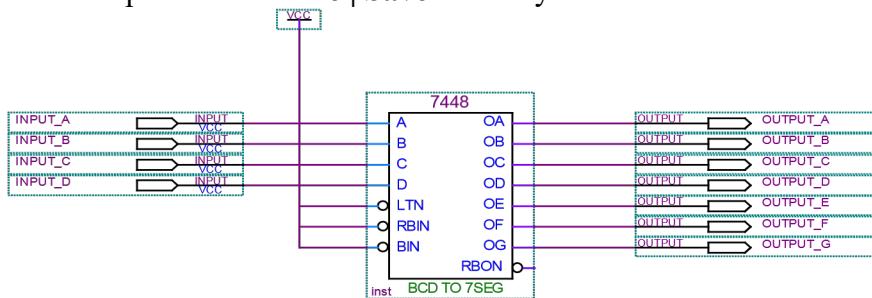


Now we need to add output pins. Repeat the procedure you used to get the first input pin, except use the symbol name *output*. Copy the pin to get a total of seven pins. Connect them to the seven outputs labelled OA through OG. Label the output pins as OUTPUT\_A through OUTPUT\_G. Your schematic should now look like the following:



We have three input pins that are not connected. They are labelled LTN, RBIN, and BIN on the 7448 symbol. We need to look at the data sheet for the 7448 to determine what these pins do and how we need to connect them. Look in the Annex for the 7448 data sheet. The labels in the data sheets are somewhat different than those on the symbol above, but a little reflection and the process of elimination makes it clear that the data sheet signal LT is the same as the LTN pin on the symbol, RBI is the same as RBIN, and BI/RBO is the same as BIN. Note that the signal names in the data book have a bar over them, indicating that the signal is asserted low. The bubble on the input pin of the signal means the same thing; these signals are active when the signal is low. Since we don't want to assert any of these signals, we must make sure that these signals remain high. The easiest way to do this is to connect them to our positive power supply voltage.

Add a new symbol VCC to the schematic above and to the left of the 7448 symbol. This symbol represents the positive supply voltage connected to the PLD. Connect the VCC symbol to the LTN, RBIN, and BIN pins on the 7448. Your schematic should like the one below. Note that when wires cross, they are NOT connected unless there is a black connection dot where the wires cross. So the Vcc symbol is NOT connected to any of the input pins A, B, C, and D. The schematic is now complete. Select **File | Save** to save your work.

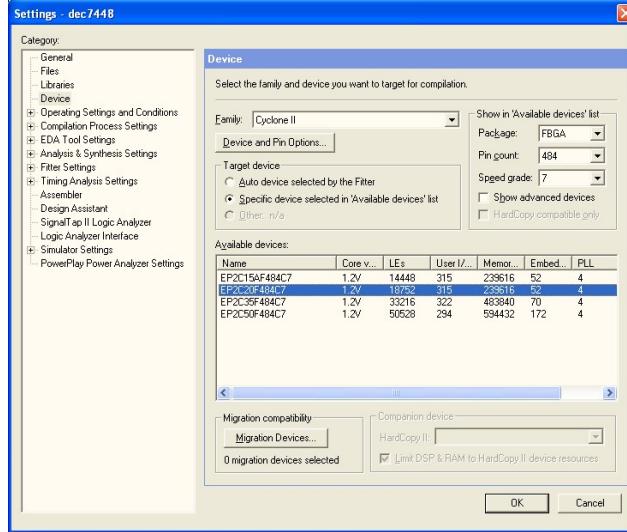


## 2.4. Compiling

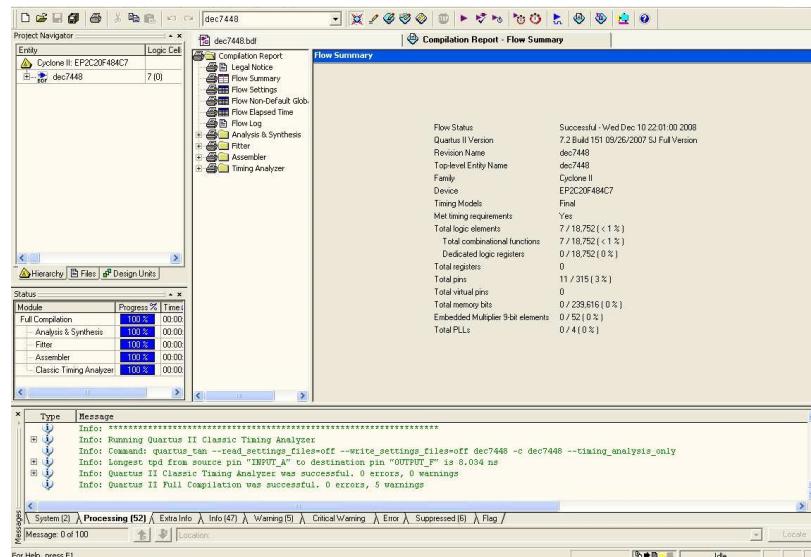
Now that the schematic is complete, we can compile it. Compiling is the process by which the schematic (or another design file) is translated into a form that determines how the PLD will be configured when it is programmed. This configuration is what makes the PLD behave in the way specified by the design file. This is analogous to way in which a program in C is compiled

from a form that is convenient for humans to use, the source code, into a form that the CPU can understand, the machine code. Just as a C compiler must know what kind of CPU the machine code is going to run on, the Altera Compiler must know what type of PLD it is compiling for. We have already specified the PLD in the project creation step.

Select **Assignments | Device...** You can change the device in this dialog box. Click **OK**.



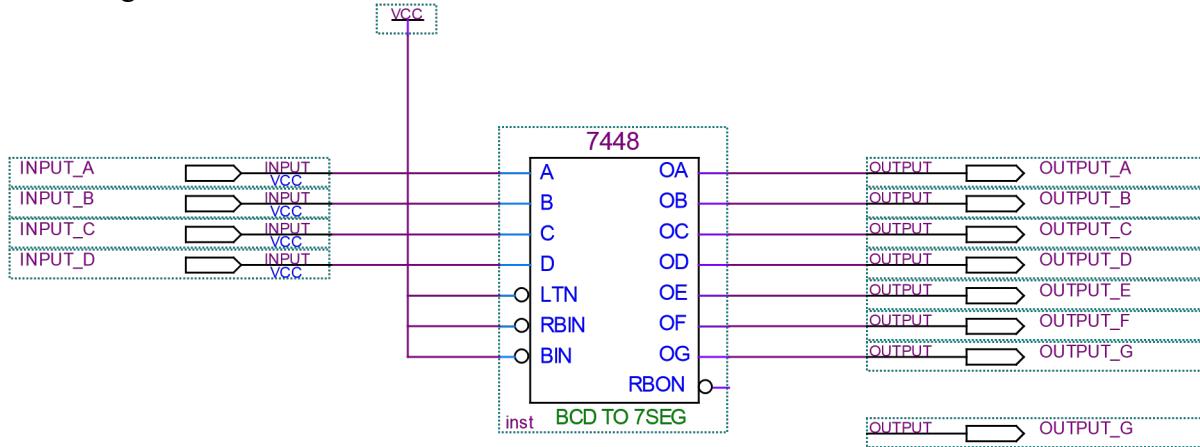
Start the Compiler by selecting **Start Compilation** from the **Processing** menu or click the **Start Compilation** button. The Compiler window should open as shown below.



If everything goes as it is supposed to, then you should see the green progress bars go from left to right, and the different grey task boxes turn dark blue as each is completed. Finally, you should get a dialog box stating that the compilation was successful.

If you did everything correctly, you won't see any errors, but in your actual projects, you will probably see lots of errors and warnings. An error is a problem so serious that the compilation can't continue. You will have to fix the error before you can compile. A warning won't usually stop the compilation, but you need to look at the warning message to see if it something you need to fix before proceeding. Some warnings can be ignored, while others will need to be fixed. Look at the help messages and use your own judgment and experience to determine which warnings you need to heed.

If you didn't have any errors, we will introduce one so that we can see what happens. Select **Window | 1 dec7448.bdf** to bring your schematic back to the foreground. Select the output pin **OUTPUT\_G** and copy it. Do not connect it to anything. Your schematic should now look something like this:



Select **File | Save** to save this schematic. Click **Start Compilation** again to compile again. This time, the Compiler will stop before it finishes compiling and a dialog box will open, showing that you have an error. Close the dialog box, and look at the *Messages* window: **Error: Illegal name "OUTPUT\_G" -- pin name already exists**

You can see that there is one error, referring to the fact that you have two pins with the same name.

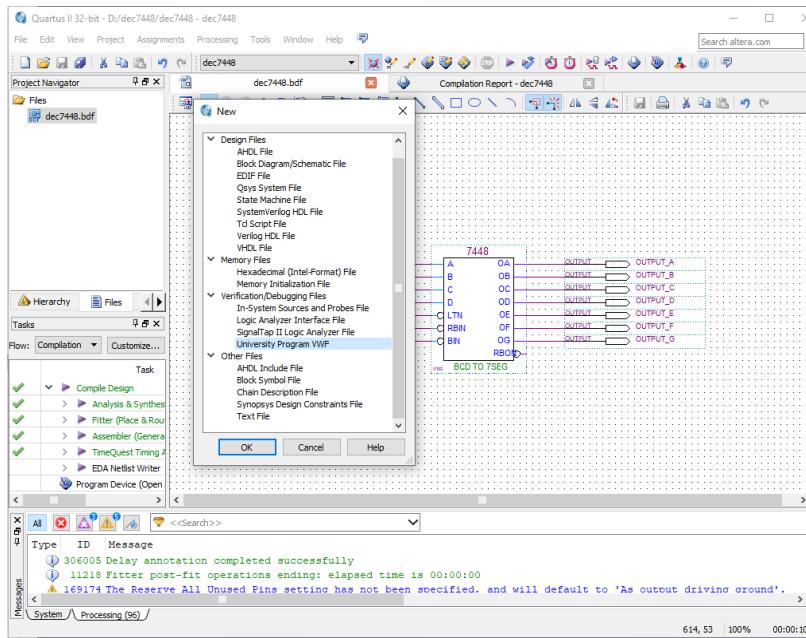
The error messages are generally fairly detailed and clear and you can usually determine what they mean just by reading them. If you need more information, double click on the error message text. You will see that the two outputs are selected to show the place of error. You can also use the **Help | Messages** to get more information on the error.

Delete the duplicated pin and save your schematic. Run the compilation again and make sure that it completes successfully.

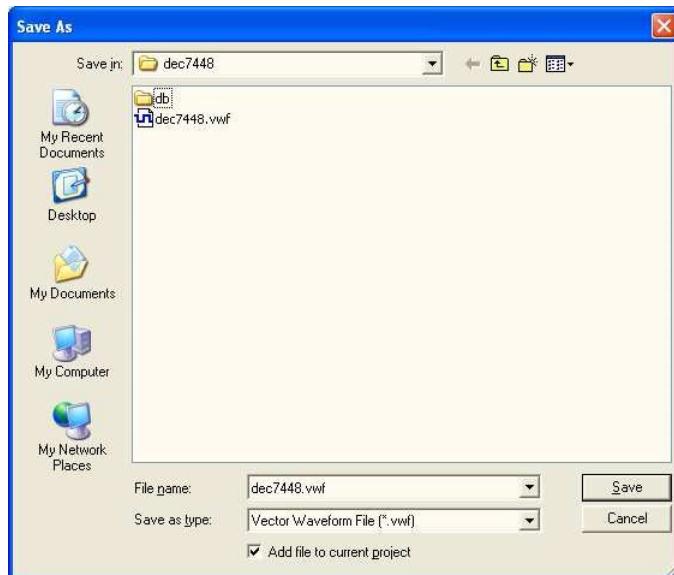
## **2.5. Waveform Entry for Simulation**

We have entered our design and it has compiled without errors. If this were a C program, the next step would likely be to run the program and see if it worked as we expected. We could, at this point, program our chip and try it out and see if it works. However, we are better off if we are as certain as we can be that our design will work properly, before we program the PLD. We will do this by using the Simulator.

Before we can use the Simulator, we need to set up a waveform file. The waveform file will hold the inputs that we want to apply to our design. We will then use the Simulator to apply these inputs to the design and see what the outputs will be. We will then check the outputs to make sure that they are what we expect. We will create the waveform file by first selecting the **New** item from the **File** menu. In the **New** window select **University Program VWF**, then click **OK**. The Waveform Editor opens, displaying an empty waveform file.



To save the file as **dec7448.vwf**, on the File menu, click **Save As**. The **Save As** dialog box appears. Accept the default file name and click **Save**. Make sure that the *Add file to current project* check box is selected.

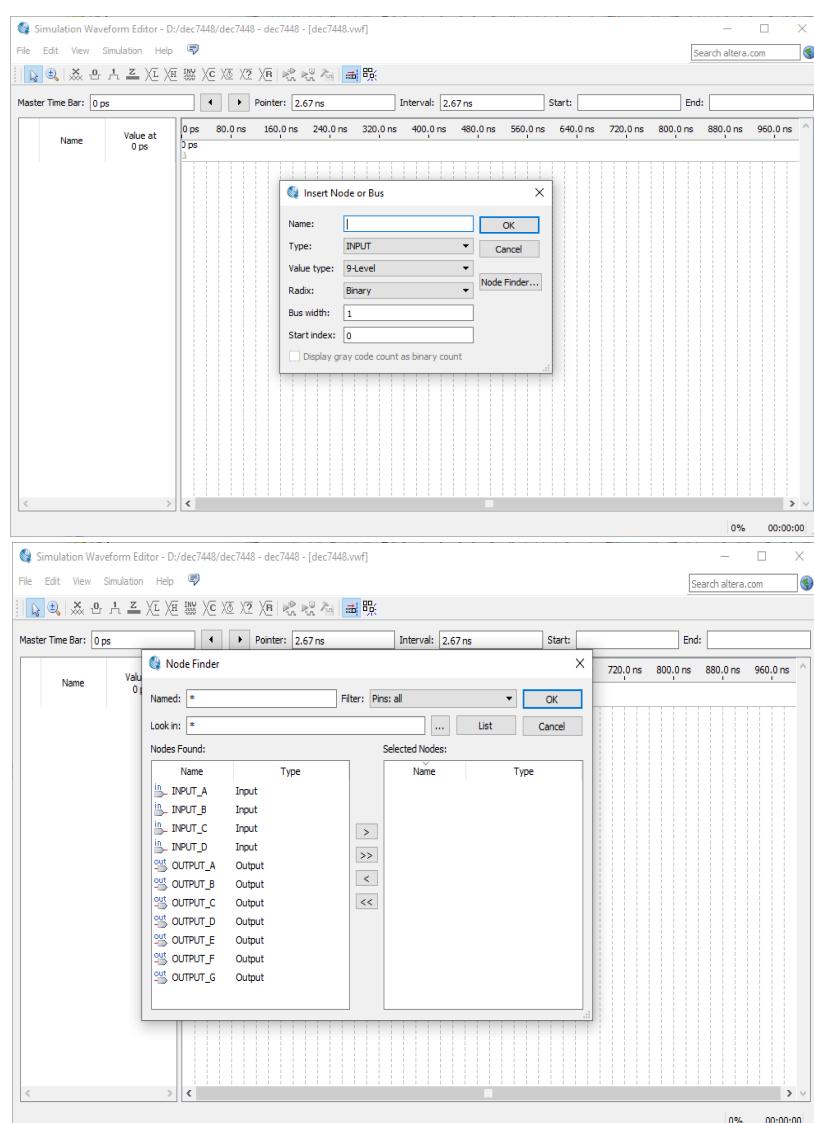


First we need to define what the inputs will be, and also define the outputs that we want to view after simulation. In our case, we want to use all of the inputs and view all of the outputs, but this might not always be the case.

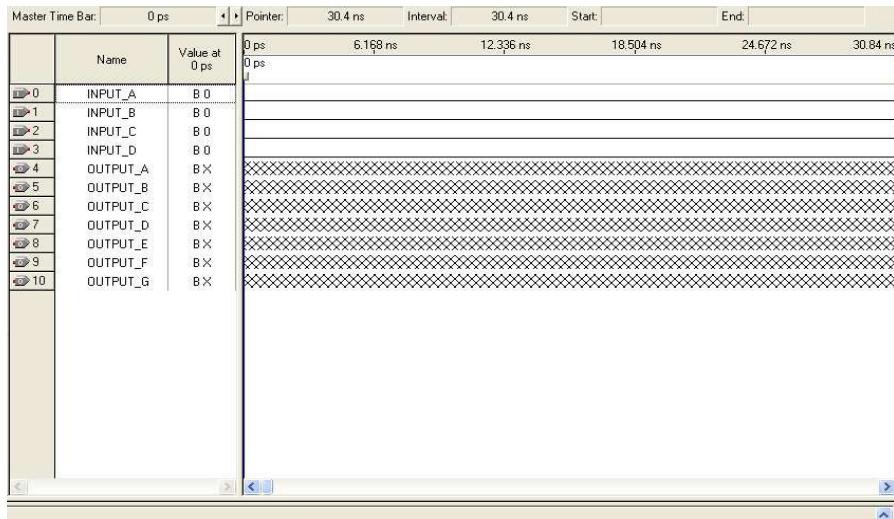
To add the input and output nodes, follow these steps:

1. To find the node names you want to add to the file, on the View menu, point to **Utility Windows** and click **Insert Node or Bus**, then click **Node Finder**. The Node Finder appears.
2. In the Node Finder, select **Pins: all** in the **Filter** list.
3. To find the nodes you want to add to the VWF, click **List**.

4. In the **Nodes Found** list, select all the pins and drag their icons into the **Name** column of the VWF. You can select multiple contiguous names with Shift+Click or select multiple non-contiguous names with Ctrl+Click.
5. Close the Node Finder.

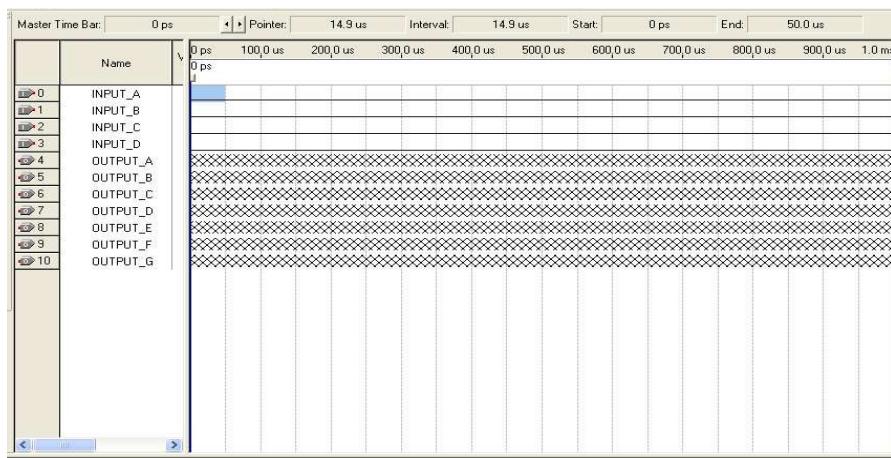


You will now see all of your inputs and outputs on the left side of the screen. There is a different symbol at the far left to identify the inputs and outputs. Next is the node name, to the right of the name is the value of that node at the time specified by the **time bar**. The cursor starts out at time 0. The inputs have value zero, by default, and the outputs have value X, meaning unknown, since we haven't run a simulation yet. To the right of the node names and values is the waveform display. The inputs all show a flat line at level zero, and the outputs show a cross-hatched pattern, indicating that they are unknown. Your window should like the one below.

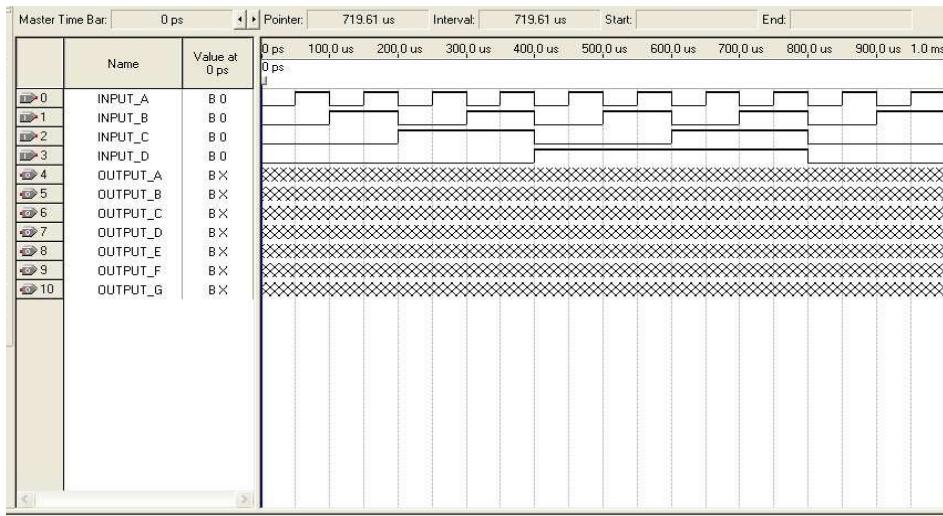


We need to change the scale of our waveform. Since we are not really concerned with timing or speed for this project, we will use a larger time scale. Select **Edit | End Time** and enter **100 us** in the box to set the end time to 100 us. The end time sets the duration of the simulation, so we will run our simulation over a time range of 0 to 100 us. Now we need to set a larger grid size so that it will be easier to draw our waveforms. Select **Edit | Grid Size** and enter **5us** in the box. Select **View | Fit In Window** to display the whole 100 us interval in the window.

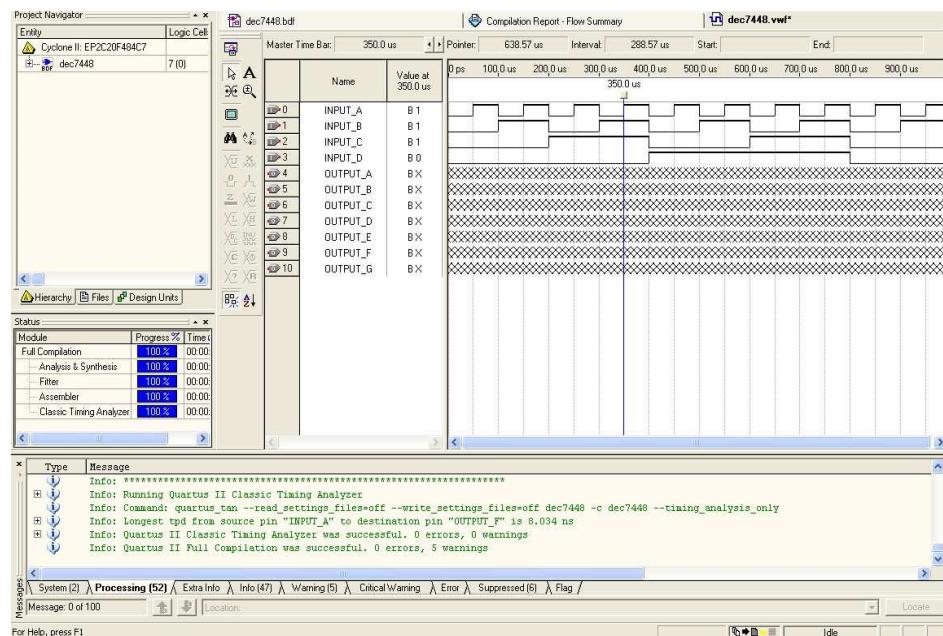
Now we need to define inputs. We would like to try out every possible combination of inputs and check the outputs for each set of inputs. Click the left mouse button on the signal for *INPUT\_A* at time 0 and drag until the time 5 us. You should have a selected region from 0 to 5 us as shown below. Look at the three boxes along the top of the window. *Start* and *End* show the start and end time of the selected region.



Once you have selected a region you change its value by clicking on the Waveform Editor Toolbar at the left of the main window. There are buttons to make the selected region a zero or a one, and also buttons to make the selection indeterminate (X), high impedance (Z), to invert the waveform, or to insert a clock or count sequence. You can do most of your work just using the zero and one buttons. Change the waveforms so that the input counts from '0000' to '1111' and back to '0000', changing every 50us. Remember that *INPUT\_A* is the Least Significant Bit (LSB) and *INPUT\_D* is the Most Significant Bit (MSB). When you are done, your waveforms should look like this:



Notice that when there is not a selected interval, the boxes at the top of window are different. The first box, *Ref* shows the location of the reference cursor. The second, *Time* shows the position of your cursor if it is in the waveform region of the window, and the third, *Interval* shows the difference between the first two. To move the Reference cursor, click the arrows to the right of the *Ref* box. Notice that the reference cursor jumps to the next transition and moves at 50us intervals. Move the cursor from 0 to 850us and verify that the inputs do count correctly from '0000' to '1111'.

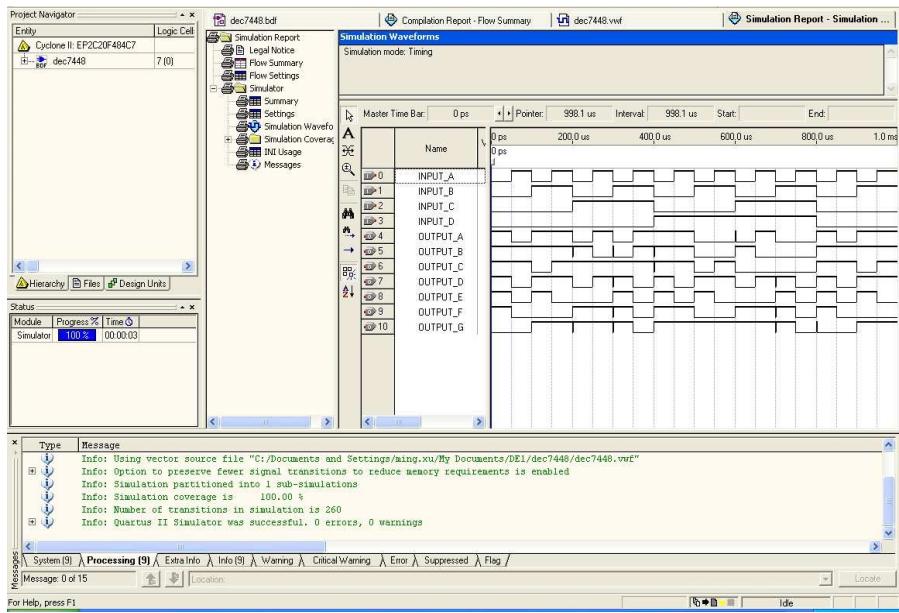


Our waveform file is now complete. Save it by selecting File | Save.

## 2.6. Simulation

Now that we have our waveform file entered, we can simulate the design and verify that it works correctly. Open the Simulator by selecting Processing | Start Simulation .

The simulation should finish quickly and you should get a dialog box showing that the simulation finished with no errors or warnings. The outputs should no longer be undefined; they should now have waveforms that represent what each output will be at each point in the simulation. Your Waveform Editor window should look like this:



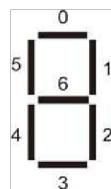
The reference cursor should start at 0. The inputs should be '0000' and the outputs should all be '1' except for *OUTPUT\_G*, which should be '0'. This means that all of the segments of the display will be on, except for segment G, which is the centre horizontal segment. Since this is the pattern to display a '0', we can see that the outputs are correct. Click the right arrow at the top of the window to move the reference cursor to the right. It should stop at 5 us. The inputs are now '0001', but the outputs have not changed. This might seem like an error until you realize that the simulator accurately simulates the propagation delays of the PLD. The inputs change at 5 us, but the outputs cannot change instantly and thus remain the same. Move the reference cursor to the right again. It should now stop at 5.0075 us. (If the time bar jumped to 10 us it means that it has only been snapped to grid.

## **2.7. Programming and Testing**

For the Altera University Programme DE1 board which you will be using in the laboratory, there are 10 toggle switches (sliders). These switches are not debounced and are intended for use as level-sensitive data inputs to a circuit. Each switch is connected directly to a pin on the cyclone II FPGA. When a switch is in the DOWN position (closest to the edge of the board) it provides a LOW logic level (0 volts) to the FPGA, and when the switch is in the UP position it provides a HIGH logic level (3.3 volts). In this experiment you can use the first four switches, SW[0] to SW[3], for the inputs INPUT\_A to INPUT\_D. The pin assignments for the ten toggle switches are shown as follows.

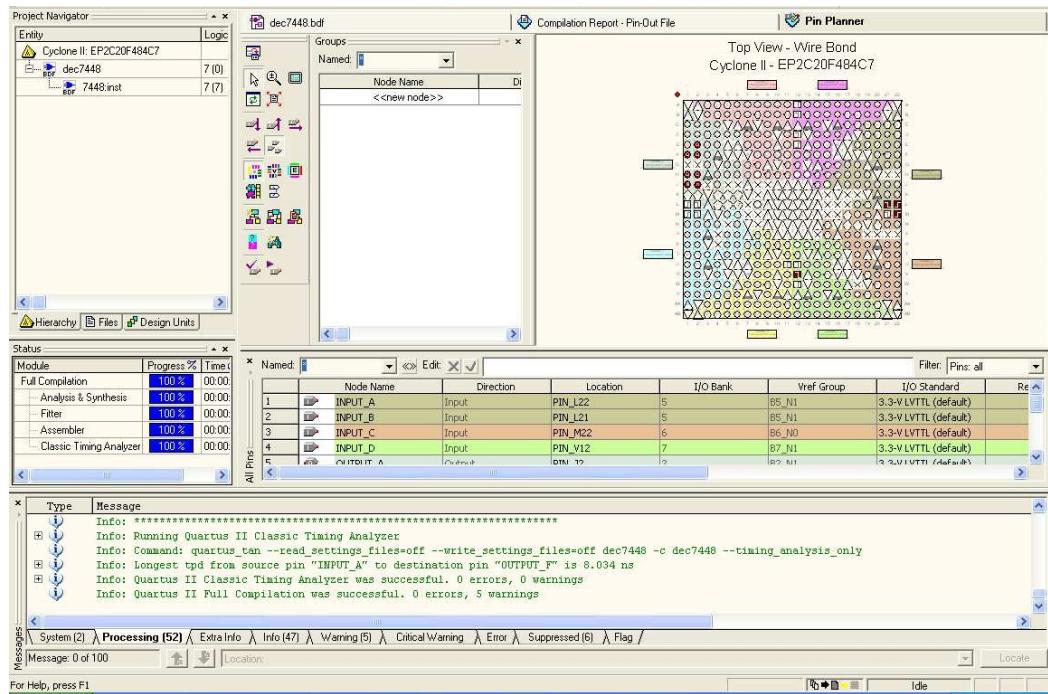
Signal Name	FPGA Pin No.	Description
SW[0]	PIN_L22	Toggle Switch[0]
SW[1]	PIN_L21	Toggle Switch[1]
SW[2]	PIN_M22	Toggle Switch[2]
SW[3]	PIN_V12	Toggle Switch[3]
SW[4]	PIN_W12	Toggle Switch[4]
SW[5]	PIN_U12	Toggle Switch[5]
SW[6]	PIN_U11	Toggle Switch[6]
SW[7]	PIN_M2	Toggle Switch[7]
SW[8]	PIN_M1	Toggle Switch[8]
SW[9]	PIN_L2	Toggle Switch[9]

On the Altera DE1 board, there are four 7-segment displays which are connected to pins on the Cyclone II FPGA. **Applying a LOW level to a segment causes it to light up**, and applying a HIGH logic level turns it off. The position and index of each segment in a 7segment display is as follows. Please note that the segments 0-6 correspond to the commonly used segments a-g, that is, 0 for a, 1 for b, 2 for c, 3 for d, 4 for e, 5 for f, and 6 for g. In this experiment, you can use Seven Segment Digit 0 (HEX0) for the output. The pin assignments for HEX0 and HEX1 are given as follows.

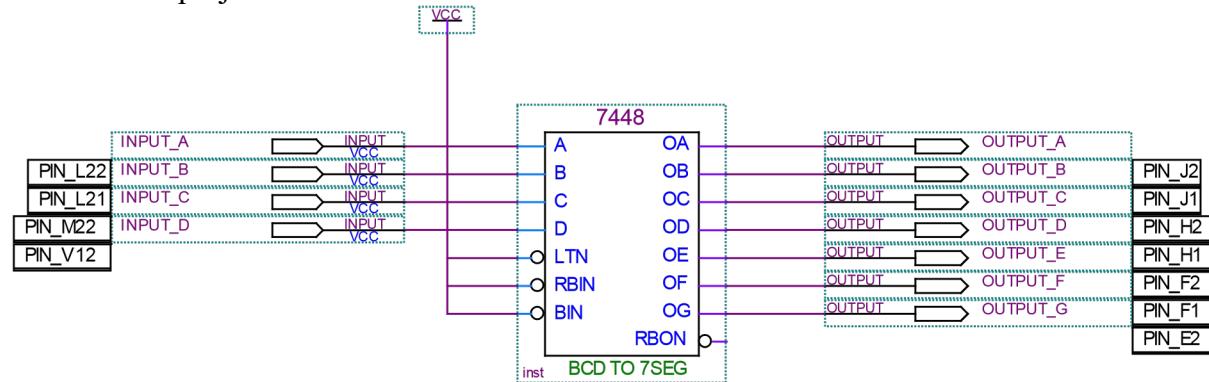


Signal Name	FPGA Pin No.	Description
HEX0[0]	PIN_J2	Seven Segment Digit 0[0]
HEX0[1]	PIN_J1	Seven Segment Digit 0[1]
HEX0[2]	PIN_H2	Seven Segment Digit 0[2]
HEX0[3]	PIN_H1	Seven Segment Digit 0[3]
HEX0[4]	PIN_F2	Seven Segment Digit 0[4]
HEX0[5]	PIN_F1	Seven Segment Digit 0[5]
HEX0[6]	PIN_E2	Seven Segment Digit 0[6]
HEX1[0]	PIN_E1	Seven Segment Digit 1[0]
HEX1[1]	PIN_H6	Seven Segment Digit 1[1]
HEX1[2]	PIN_H5	Seven Segment Digit 1[2]
HEX1[3]	PIN_H4	Seven Segment Digit 1[3]
HEX1[4]	PIN_G3	Seven Segment Digit 1[4]
HEX1[5]	PIN_D2	Seven Segment Digit 1[5]
HEX1[6]	PIN_D1	Seven Segment Digit 1[6]

Click on the **Assignments** menu and then click on the **Pins** to open the **Pin Planar** window. Assign pins to the inputs and outputs by double clicking on the location drop-down box and selecting the pins which are physically connected to the Seven Segment and the input toggle switches of the board.

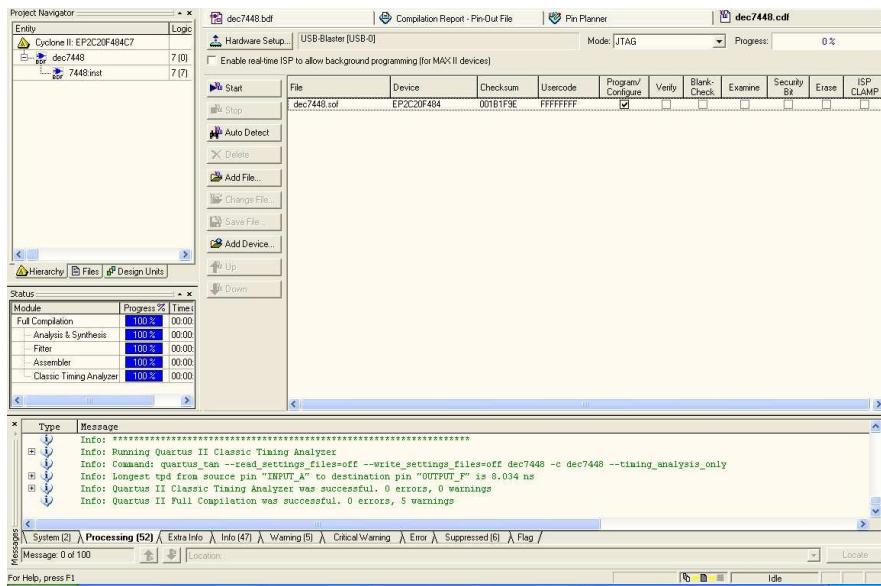


Look at your schematic. The pin assignments that you just made have been added as annotations on the schematic. You don't need to save your schematic file, since the pin assignments are stored in the project file.



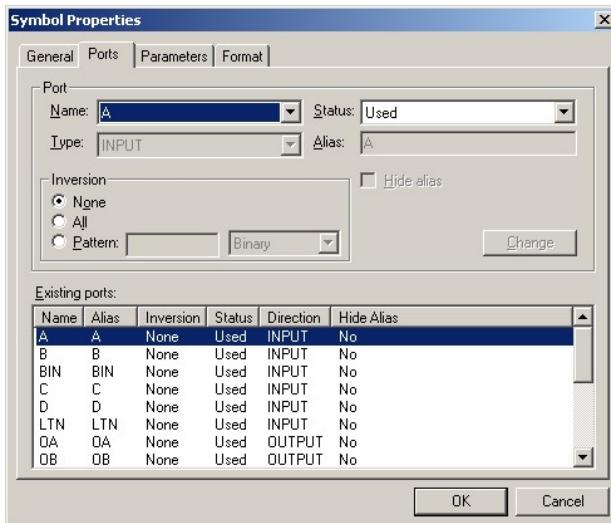
**We have made the pin assignments, but they won't take effect until we compile again.** Click on the **Start Compilation** button to recompile. When the compilation is done, open the Pin-Out file from the Fitter directory in the Compilation report window and look at the new report. Our new pin assignments have been implemented and we are now ready to program our PLD. As a general rule, you should re-simulate after changing pin assignments, but as pin assignments should only affect timing, and timing is not a concern, we won't check the simulation again here. If we were concerned with the speed and timing of our design, and you almost always will be in practice, we would most certainly have to re-simulate at this point.

Open the Programmer window by selecting **Tools | Programmer**. The programmer window should look like the one below. Note that the programming file for your project, *dec7448.pof* will be selected by default.

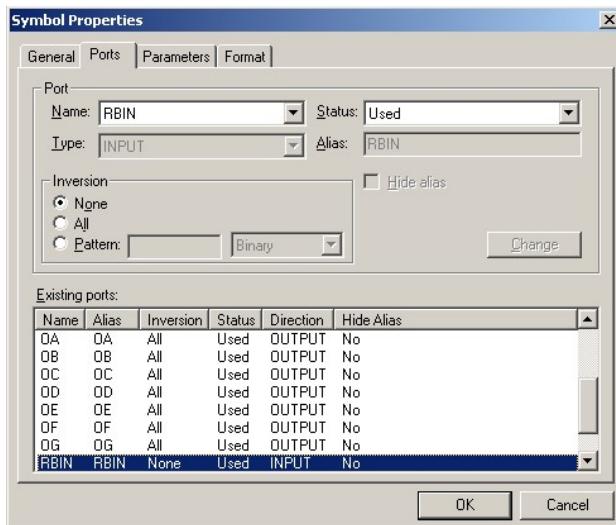


Make sure that the Program/Configure check box is selected. Now Switch on the DE1 Board. Click on the **Start** button in quartus II Programmer window. The device will be examined, programmed, the programming will be verified.

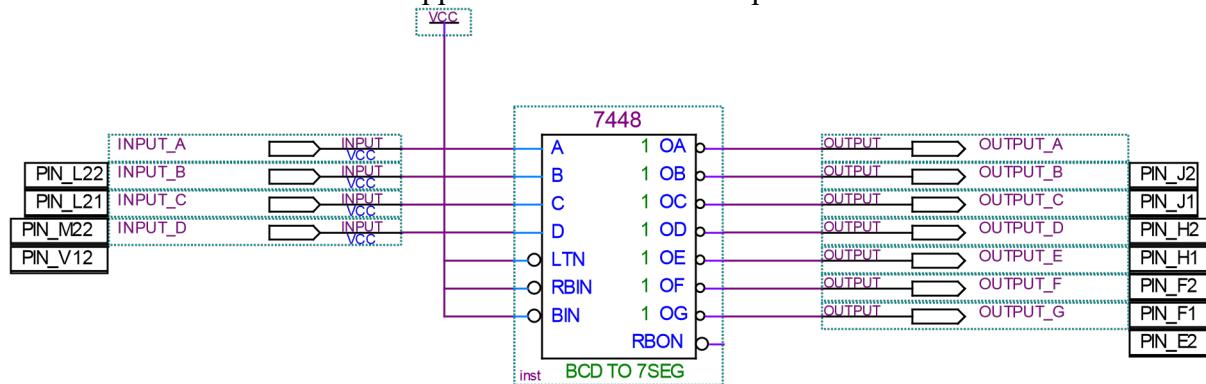
Now record the 7 segment LED display for each of the 16 possible combinations of inputs by changing the toggle switches. You will notice that when a LED segment should be switched on, it is actually switched off and vice versa. This is because the LEDs are wired up so that when the output is Logic ‘0’ the LED is illuminated and not when it is Logic ‘1’. Therefore we need to modify the design to take into account the required inversion. Whilst we could add invertors to the design, it is easy to change the polarity of outputs. Right click on the 7448 Symbol and select “Properties/Ports..”.



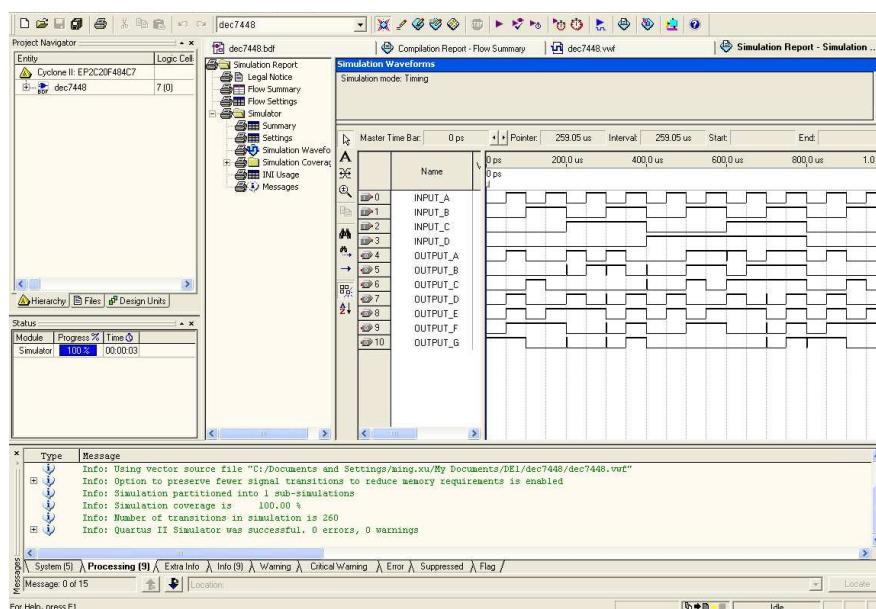
For the outputs OA to OG change the Inversion from “None” to “All” by highlighting each output and selecting All. Click on OK to return to the bdf file.



Notice how the little circle has appeared at each of the outputs to show the inversion.



Recompile your design and re-run the simulator. You should notice that the outputs have now been inverted. When all the inputs are “0” only OUTPUT\_G is off (high) all the other outputs are on (low).



Reprogram your device and again record the outputs against the 16 input combinations.

### **3. AHDL combinational design**

AHDL is Altera's own Hardware Description Language which is used for the text based entry of designs. Both combinational and sequential systems can be entered.

#### **3.1.Boolean Equations**

Boolean equations are used in the Logic Section of your AHDL TDF (text definition file) to represent the connection of nodes, the flow of inputs into and the flow of outputs from input and output pins, primitives, macrofunctions, and state machines.

The following example shows a complex Boolean equation:

```
a[] = ((c[] & -B"001101") + e[6..1]) # (p, q, r, s, t, v);
```

The left side of the equation can be a symbolic, port, or group name. You can use the NOT (!) operator to invert any item on the left. The right side of the equation consists of a Boolean expression.

The equals symbol (=) is used in Boolean equations to indicate that the result of the Boolean expression on the right side is the source of the symbolic node or group on the left side. The single equals symbol differs from the double equals symbol (==), which is used as a comparator.

In the example shown above, the Boolean expression on the right is evaluated according to Boolean equation priority rules:

1. The binary number B"001101" is negated (Two's complemented) and becomes B"110011". The unary minus (-) has first priority.
2. B"110011" is ANDed (&) with the group c[]. This expression has second priority because it is enclosed in parentheses.
3. The result of the group expression in step 2 is added (+) to the group e[6..1].
4. The result of the expression in step 3 is ORed (#) with the group (p, q, r, s, t, v). This expression has last priority.

The final result is assigned to the group a[].

For the sample equation shown above to be legal, the number of bits in the group on the left side of the equation must be evenly divisible by the number of bits in the group on the right side of the equation. The bits on the left side of the equation are mapped to the right side of the equation in order.

#### **3.2.Logical Operators**

The following logical operators can be used in Boolean expressions:

Operator:	Example:	Description:
-	-num	two's complement
!	!tob	one's complement (prefix inverter)
NOT	NOT tob	
&	bread & butter	AND
AND	bread AND butter	
! &	a[3..1] !& b[5..3]	AND inverter (NAND)
NAND	a[3..1] NAND b[5..3]	
#	trick # treat	OR

OR	trick OR treat	
!#	c[8..5] !# d[7..4]	OR inverter (NOR)
NOR	c[8..5] NOR d[7..4]	
\$	foo \$ bar	exclusive OR
XOR	foo XOR bar	
!\$	x2 !\$ x4	exclusive NOR
XNOR	x2 XNOR x4	

Each operator represents a two-input logic gate, except the NOT (!) operator, which is a prefix inverter on a single node. You can use either the name or the symbol to represent a logical operator.

### **3.3.Subdesign**

The Subdesign Section declares the input, output, and bidirectional ports of the TDF. The following example shows a Subdesign Section:

```
SUBDESIGN top
( foo, bar, clk1, clk2 : INPUT = VCC;  a0,
a1, a2, a3, a4 : OUTPUT;  b[7..0]    : BIDIR;
)
```

The Subdesign Section has the following characteristics:

The keyword SUBDESIGN is followed by the subdesign name. The subdesign name must be the same as the TDF filename. In this example, the subdesign name is top.

- The list of signals is enclosed in parentheses ( ).
- Signal names are represented by symbolic names such as foo, and are assigned a port type such as INPUT.
- Signal names are separated by commas (,), are followed by a colon (:) and a port type, and end with a semicolon (;).

The port type may be INPUT, OUTPUT, BIDIR, MACHINE INPUT, or MACHINE OUTPUT. In the example shown above, the foo, bar, clk1, and clk2 signals are inputs and a0, a1, a2, a3, and a4 are outputs. The bus b[7..0] is bidirectional.

You can optionally assign a default value of GND or VCC after the port type (otherwise, no default value is assumed). In the example shown above, VCC is the default value for the input signals unless they are assigned in a higher-level file (assignments in a higher-level file take precedence).

In a top-level design file, INPUT, OUTPUT, and BIDIR port types represent actual device pins. In a lower-level design file, all port types are the inputs and outputs of the file, but not of the project itself.

### **3.4.Implementing Boolean Expressions & Equations**

Boolean expressions are sets of nodes, numbers, constants, and other Boolean expressions, separated by operators and/or comparators, and optionally grouped with parentheses. A Boolean equation sets a node or group equal to the value of a Boolean expression.

The boole1.tdf file shown below shows two simple Boolean expressions that represent two logic gates.

```
SUBDESIGN boole1
(
    a0, a1, b : INPUT;
    out1, out2 : OUTPUT;
) BEGIN      out1 =
    a1 & !a0;      out2 =
    out1 # b; END;
```

In this sample file, the out1 output is driven by the logical AND of a1 and the inverse of a0, and the out2 output is driven by the logical OR of out1 and b. Since these equations are evaluated concurrently, their order in the file is not important.

### **3.5.Declaring Nodes**

A node, which is declared with a Node Declaration in the Variable Section, can be used to hold the value of an intermediate expression.

Node Declarations are especially useful when a Boolean expression is used repeatedly. The Boolean expression can be replaced with a descriptive node name, which is easier to read.

The boole2.tdf file, shown below, contains the same logic as boole1.tdf (shown in Implementing Boolean Expressions & Equations), but has only one output.

```
SUBDESIGN boole2
(
    a0, a1, b : INPUT;
    out : OUTPUT;
) VARIABLE
    a_equals_2 : NODE; BEGIN
    a_equals_2 = a1 & !a0;
    out = a_equals_2 # b; END;
```

This file declares the node a\_equals\_2 and assigns the value of the expression a1 & !a0 to it. Using nodes can save device resources when the node is used in several expressions.

### **3.6.If Then Statement Logic**

The priority.tdf file shown below shows a priority encoder that converts the level of the highest-priority active input into a value. It generates a 2-bit code that indicates the highestpriority input driven by VCC.

```
SUBDESIGN priority
(
    low, middle, high : INPUT;
    highest_level[1..0] : OUTPUT;
)
BEGIN
    IF high THEN
        highest_level[] = 3;
    ELSIF middle THEN
        highest_level[] = 2;      ELSIF
    low THEN          highest_level[]
    = 1;      ELSE
        highest_level[] = 0;
    END IF;
END;
```

In this example, the inputs high, middle, and low are evaluated to determine whether they are driven by VCC. The “If Then” statement activates the equations that follow the active IF or ELSE clause, e.g., if high is driven by VCC, highest\_level[] is 3.

If more than one input is driven by VCC, the If Then Statement evaluates the priority of the inputs, which is determined by the order of the IF and ELSIF clauses (the first clause has the highest priority). In priority.tdf, high has the highest priority, middle has the next highest priority, and low has the lowest priority. The “If Then” statement activates the equations that follow the highest-priority IF or ELSE clause that is true.

If none of the inputs are driven by VCC, the equations following the ELSE keyword are activated.

### **3.7.Case Statement Logic**

The decoder.tdf file shown below is a 2-bit-to-4-bit decoder. It converts two binary code inputs into a "one-hot" code.

```
SUBDESIGN decoder
(
    code[1..0] : INPUT;
    out[3..0]      : OUTPUT;
)
BEGIN
    CASE code[] IS
        WHEN 0 => out[] = B"0001";
        WHEN 1 => out[] = B"0010";
        WHEN 2 => out[] = B"0100";
        WHEN 3 => out[] = B"1000";
    END CASE;
END;
```

In this example, the input group code[1..0] has the value 0, 1, 2, or 3. The equation following the appropriate => symbol in the Case Statement is activated. For example, if code[] is 1, out1 is set to B"0010". Since the values of the expression are all different, only one WHEN clause can be active at one time.

### **3.8.Creating Decoders**

A decoder contains combinatorial logic that interprets input patterns and converts them to output values. In AHDL, you can use a Truth Table Statement or the lpm\_compare or lpm\_decode functions to create a decoder.

The 7segment.tdf file shown below is a decoder that specifies logic for patterns of lightemitting diodes (LEDs). The LEDs are illuminated in a seven-segment display to show the hexadecimal numbers 0 to 9 and the letters A to F.

```
% -a-
% f |   | b %
% -g-
% e |   | c %
% -d-
%
% 0 1 2 3 4 5 6 7 8 9 A b C d E F %
```

```

SUBDESIGN 7segment
(
    i[3..0]           : INPUT;
a, b, c, d, e, f, g : OUTPUT;
)
BEGIN
TABLE
    i[3..0] => a, b, c, d, e, f, g;
H"0"    => 1, 1, 1, 1, 1, 1, 0;
H"1"    => 0, 1, 1, 0, 0, 0, 0;
H"2"    => 1, 1, 0, 1, 1, 0, 1;
H"3"    => 1, 1, 1, 1, 0, 0, 1;
H"4"    => 0, 1, 1, 0, 0, 1, 1;
H"5"    => 1, 0, 1, 1, 0, 1, 1;
H"6"    => 1, 0, 1, 1, 1, 1, 1;
H"7"    => 1, 1, 1, 0, 0, 0, 0;
H"8"    => 1, 1, 1, 1, 1, 1, 1;
H"9"    => 1, 1, 1, 1, 0, 1, 1;
H"A"    => 1, 1, 1, 0, 1, 1, 1;
H"B"    => 0, 0, 1, 1, 1, 1, 1;
H"C"    => 1, 0, 0, 1, 1, 1, 0;
H"D"    => 0, 1, 1, 1, 1, 0, 1;
H"E"    => 1, 0, 0, 1, 1, 1, 1;
H"F"    => 1, 0, 0, 0, 1, 1, 1;
END TABLE;
END;

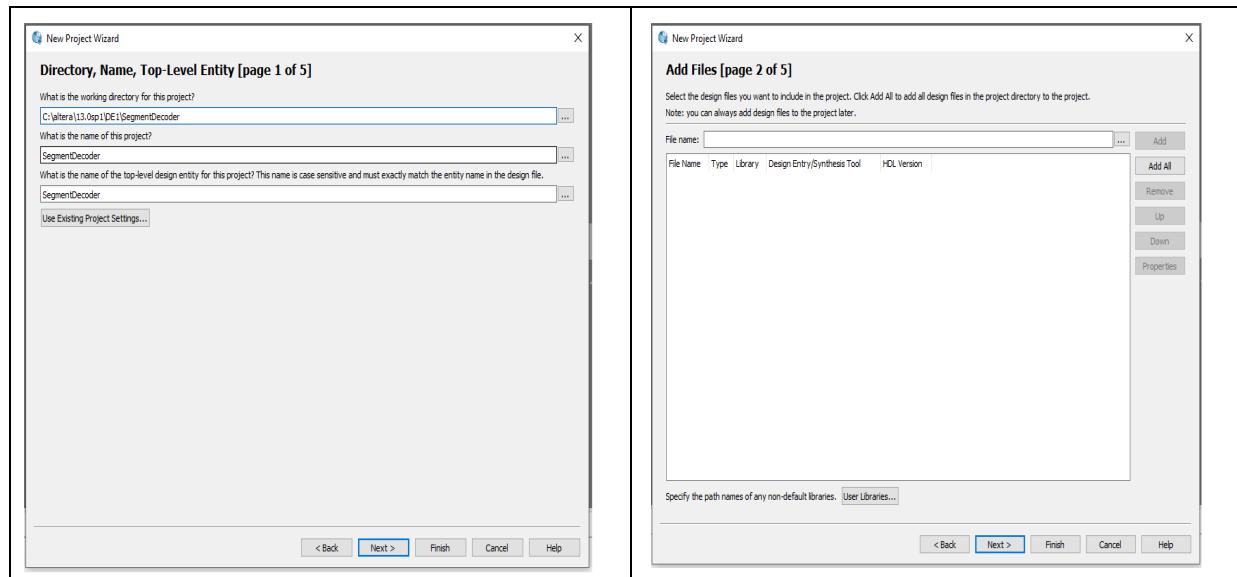
```

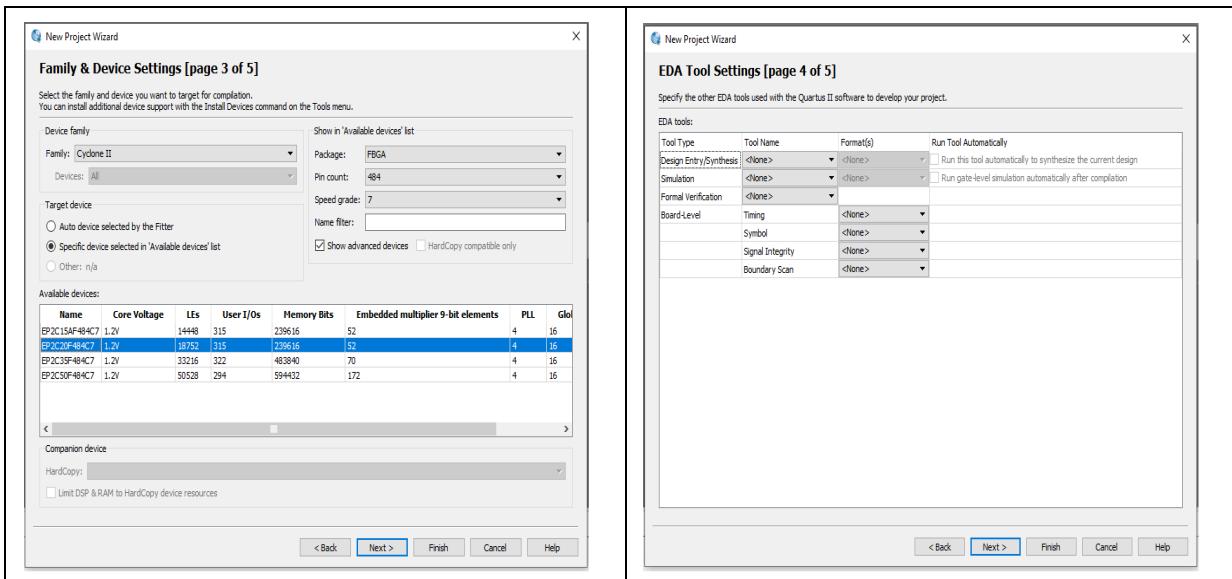
In this example, the output pattern for all 16 possible input patterns of  $i[3..0]$  is described in the Truth Table Statement.

### **3.9.Decoder Implementation**

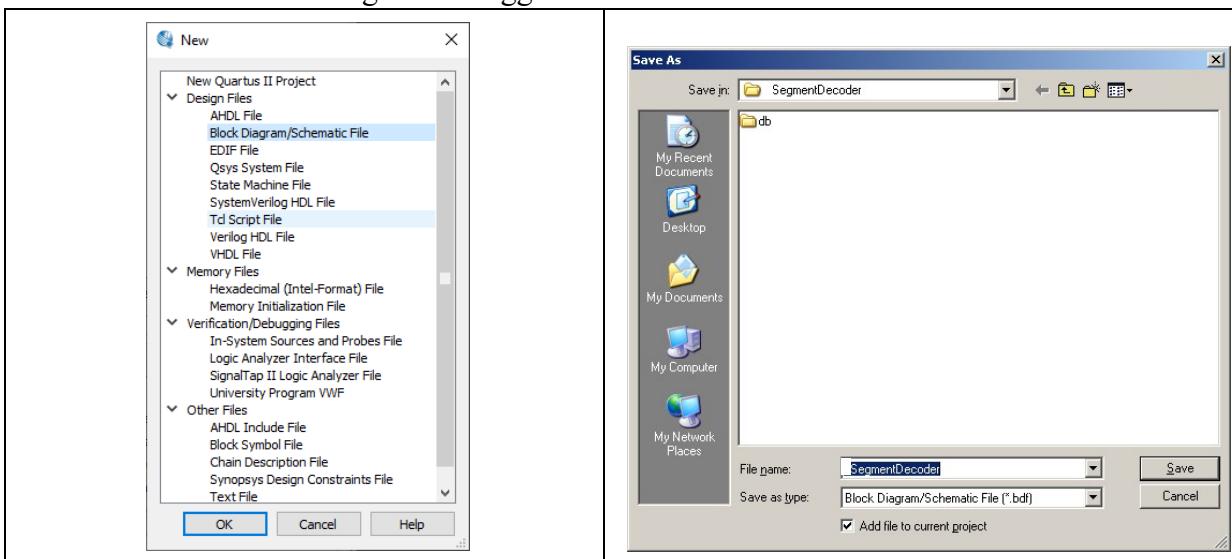
We will start by producing a 7 segment decoder for a hexadecimal number (0-15). As previously indicated the Quartus II package allows the hierarchical entry of files. What is easiest is to create a series of text based sub-designs and link them together using the graphical editor.

We will start by creating a new project called *SegmentDecoder* using **File | New Project Wizard**.





Then click on **File | New...** Select **Block Diagram/Schematic File**. A new block1.bdf file is created. Click on save and give the suggested name to the file. Click **Save**.



Now create an AHDL file by selecting the AHDL file type from the **File | New** menu item. Then enter the text for the decoder as shown in the next figure. The lines beginning and ending with the % sign are comment fields which are used to document designs. (Hint: use of the help system can save a lot of typing here. Do a search for help on “decoders AHDL” and then select “Creating Decoders (AHDL)”, you should then be able to cut and past the text into your tdf file)

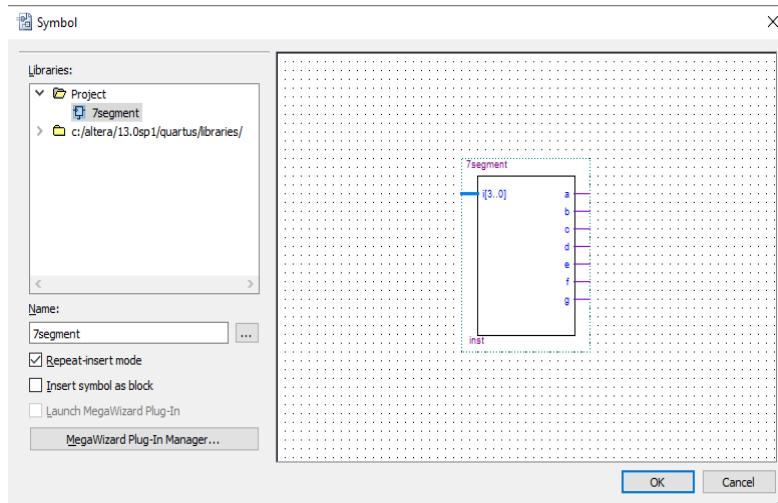
```

Quartus II 32-bit - C:/altera/13.0sp1/DE1/SegmentDecoder/SegmentDecoder - SegmentDecoder
File Edit View Project Assignments Processing Tools Window Help
Project Navigator Entity SegmentDecoder SegmentDecoder.bdf Compilation Report - SegmentDecoder.tdf
Cyclone II: EP2C20F484C7
SegmentDecoder
7segmentinst
1 SUBDESIGN 7segment
2   i[3..0] : INPUT;
3   [a, b, c, d, e, f, g : OUTPUT;
4 ]
5 BEGIN
6 TABLE
7   i[3..0] => a, b, c, d, e, f, g;
8   H"00" => 1, 1, 1, 1, 1, 1, 0;
9   H"01" => 0, 1, 1, 0, 0, 0, 0;
10  H"02" => 1, 1, 0, 1, 1, 0, 1;
11  H"03" => 1, 1, 1, 0, 0, 0, 1;
12  H"04" => 1, 1, 1, 1, 0, 0, 1;
13  H"05" => 1, 1, 1, 1, 1, 0, 1;
14  H"06" => 1, 1, 1, 1, 1, 1, 1;
15  H"07" => 1, 0, 1, 1, 1, 1, 1;
16  H"08" => 1, 1, 1, 0, 0, 0, 0;
17  H"09" => 1, 1, 1, 1, 1, 1, 1;
18  H"0A" => 1, 1, 1, 1, 0, 1, 1;
19  H"0B" => 1, 1, 1, 0, 1, 1, 1;
20  H"0C" => 0, 0, 1, 1, 1, 1, 1;
21  H"0D" => 0, 1, 1, 1, 1, 0, 1;
22  H"0E" => 1, 0, 0, 1, 1, 1, 1;
23  H"0F" => 1, 0, 0, 0, 1, 1, 1;
24  H"0G" => 0, 0, 1, 1, 1, 1, 1;
25 END TABLE;
26 END;

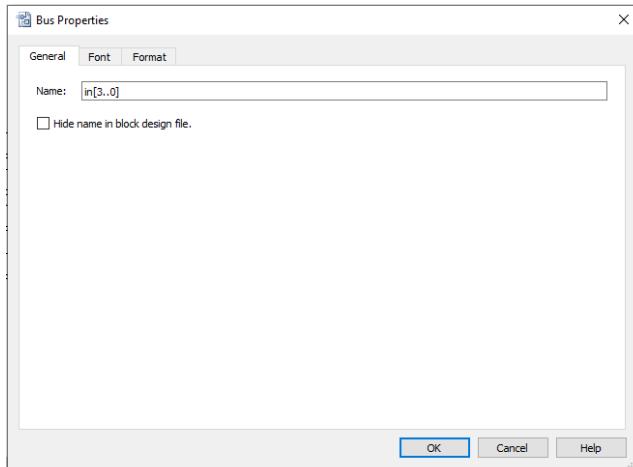
```

Save the file using **File | Save** in DE1 directory with the filename “7segement.tdf” and make sure that the check box for **Add file for Current project** is selected

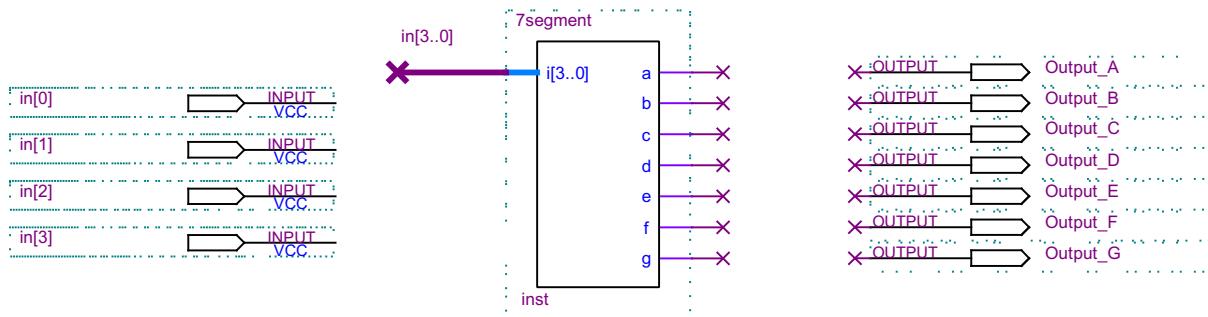
To create a graphical symbol from this file to include in the schematic select **File | Create/Update | Create Symbol Files For Current File**. Next we need to insert the created symbol into the top level schematic file. Right click on the window of the top-level **bdf** file and select **Insert | Symbol**.



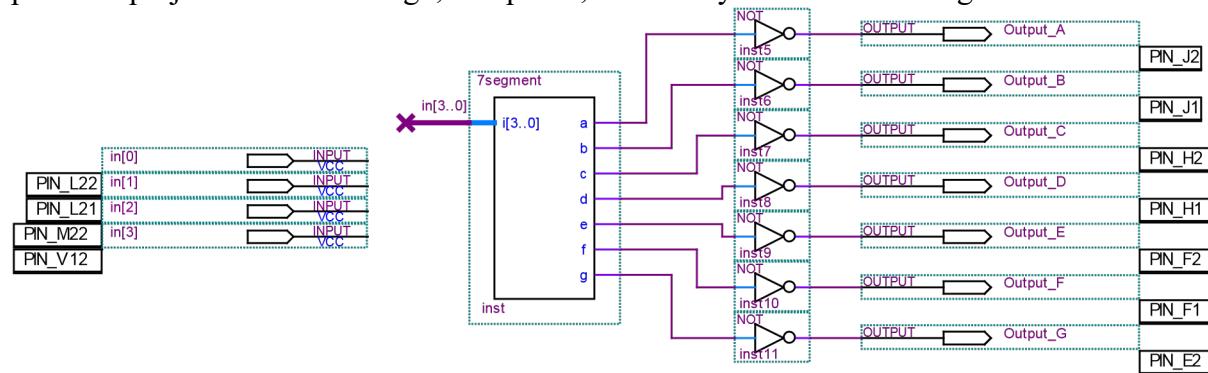
Notice how the inputs to the 7segment module are shown as a single thick line and labelled  $i[3..0]$ . This represents a “bus” of four connections. As we can’t directly connect single wires to buses we need to label the bus and the inputs using common names. To do this, extend the bus line from the 7segment module to the left by pressing the left mouse button and dragging. Then right click on the line and select **Properties**. Enter Node/Bus Name.



Type in “in[3..0]” which indicates a group of four signals labelled in[3] down to in[0]. Then insert input and output pins as you did for the first project but this time give the names of in[3] ... in[0] to the inputs.

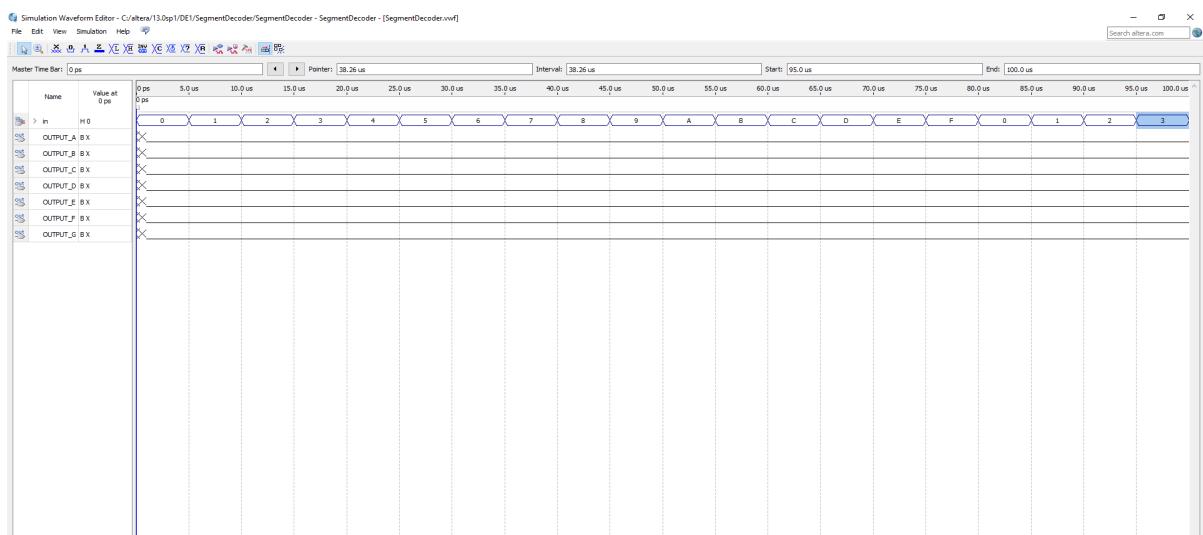
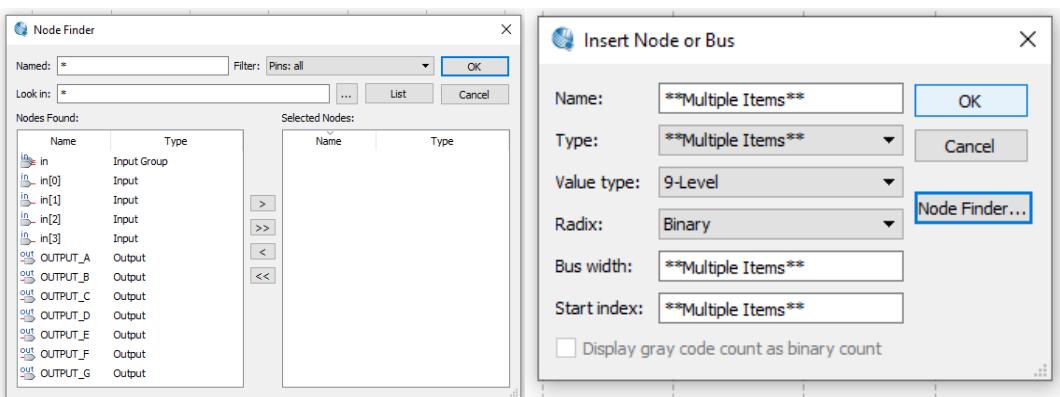
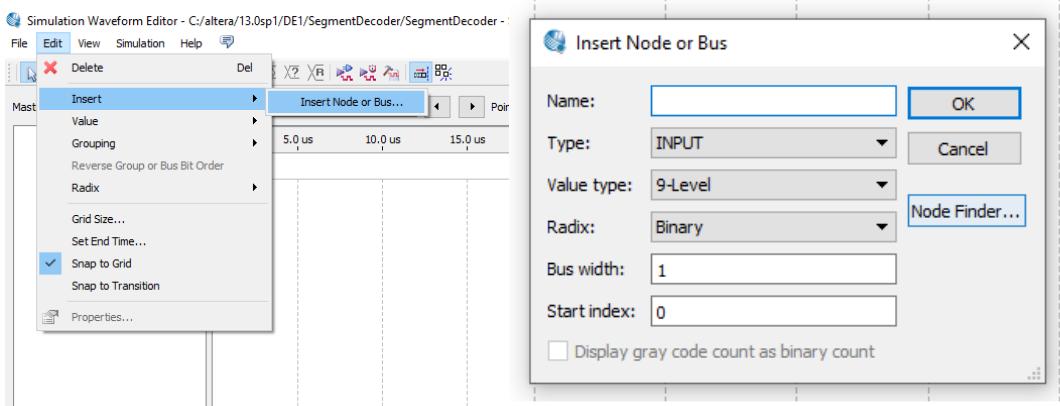


Finally you need to invert the polarity of the output signals by adding seven inverters to the design. Click on the Symbol button in the toolbar and type *not* in the name drop-down list. Now the design is complete. Next we want to assign the pin numbers exactly as you did for the previous project. Save the design, compile it, and finally simulate the design.



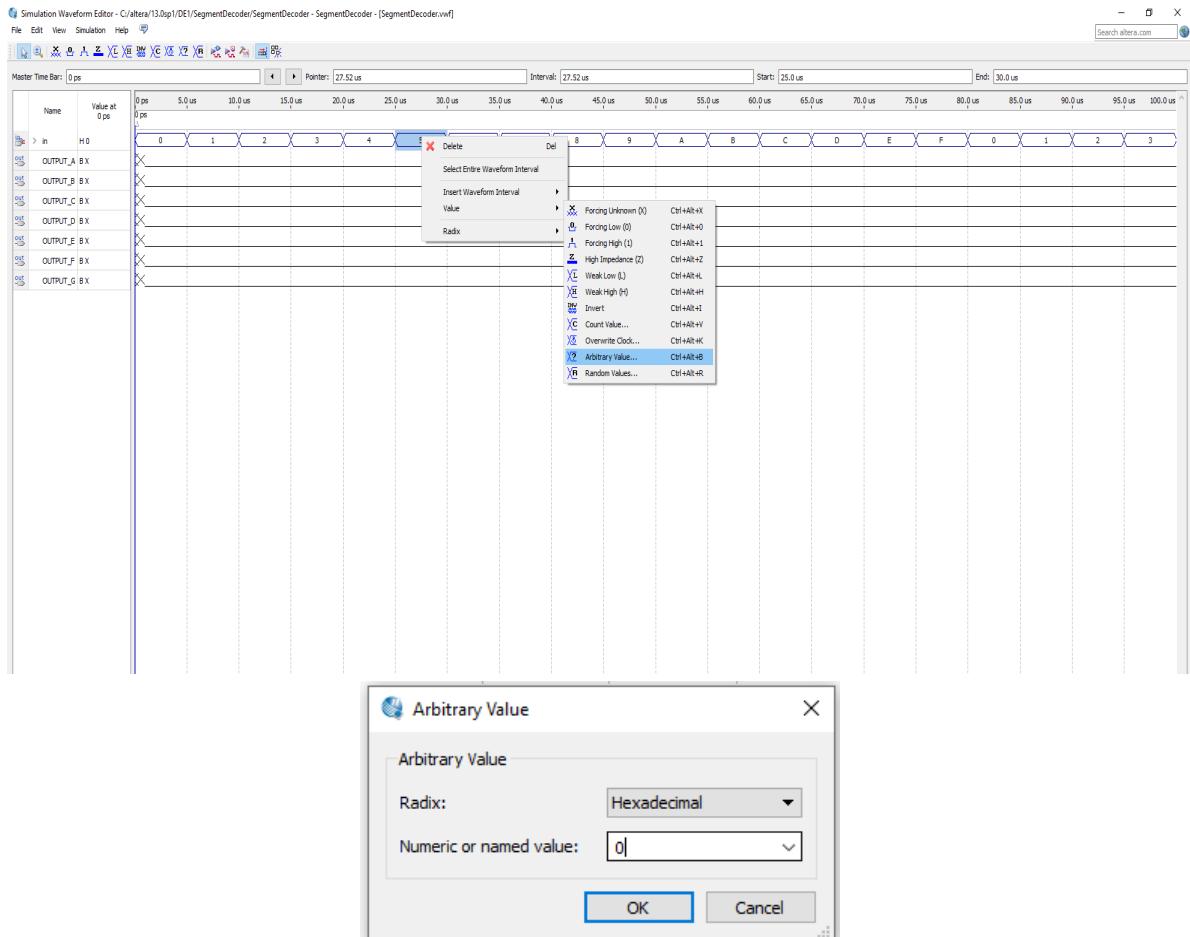
Create a new *Vector Waveform File* by clicking on the **File | New | Other Files | Vector Waveform**. Save your file by selecting **File | Save...** and accept the default name *SegmentDecoder.vwf*.

Open the **Node Finder** window by select **View | Utility Windows | Node Finder...**. Click on the List button to list all the nodes. Drag and drop the **in** Group node and the output nodes to the **Name** area of the Waveform Editor.



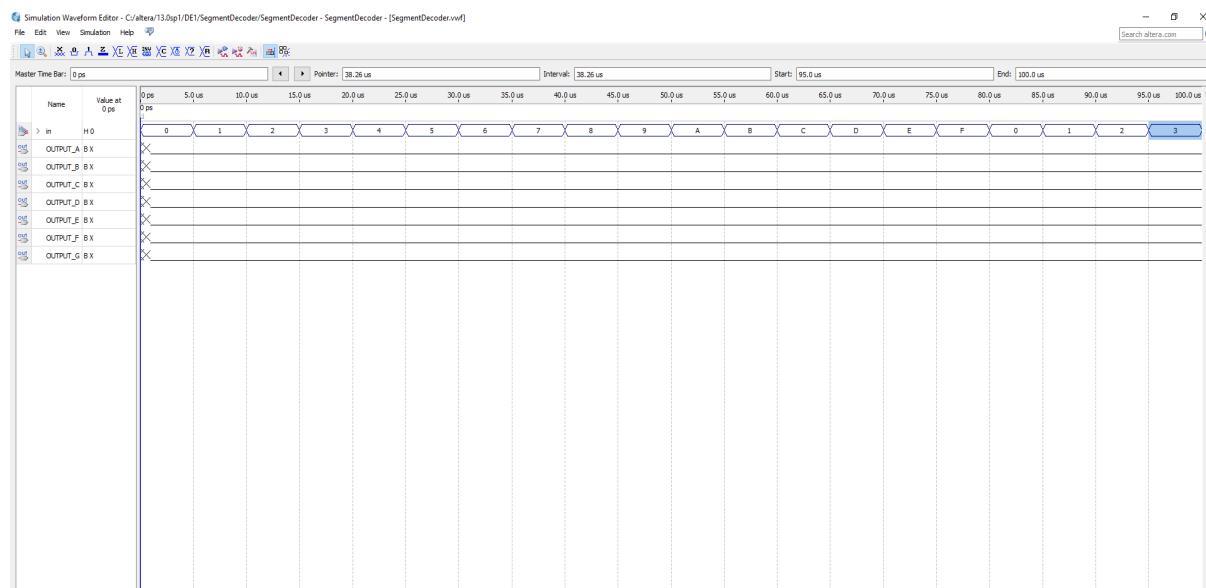
Note that the inputs are grouped together with a default base of “Hexadecimal”. When using buses it is often easier to only display the bus value rather than the individual voltage levels. We again need to change the scale of our waveform. Since we are not really concerned with timing or speed for this project, we will use a larger time scale. Select **Edit | End Time** and enter **1ms** in the box to set the end time to **100 us**. Select **Edit | Grid Size** and enter **5us** in the box. Select **View | Fit In Window** to display the whole 100 us interval in the window.

Next we need to modify the inputs  $i[3..0]$  to represent the 16 combinations of inputs. Highlight the  $i[3..0]$  inputs between 5 and 10 us by using the left mouse button and dragging the cursor. Then right click to select the “Value” and “Arbitrary values” dropdown menus.

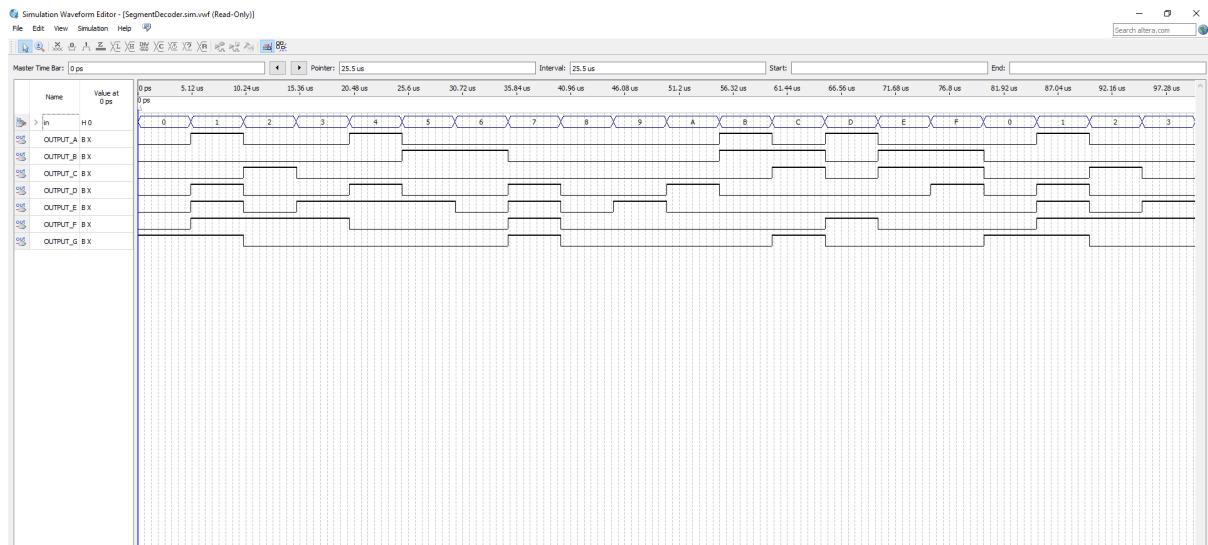


Set the Radix to Hexadecimal and the value to 1 and select OK.

This process now needs repeating for the other group values. Note that the value is in hexadecimal so you will enter ‘A’ rather than ‘10’ etc. Your simulation file should now look like this.



Start the simulator by clicking on the **Processing** menu and selecting the **Start Simulation**. Your waveform file should now look like this with the output values having been determined by the simulator.



Notice that for an output value of “0” only the G LED is disabled (high) whilst all the other LEDs are on (low).

Next program the PLD by selecting the **Programmer** from the **Tools** menu. Once the device has been programmed record the LED outputs for the 16 combinations of toggle switch inputs.

## 4. AHDL sequential design

For any sequential system some form of memory is required. Although “JK” flip flops are available in AHDL these are generally simulated within the device using “D” type flip-flops.

### 4.1. ‘D’ Flip-Flop with Enable

The AHDL definition of the D type flip flop is:

FUNCTION DFFE (D, CLK, CLRN, PRN, ENA)

RETURNS (Q);

Inputs

CLRN	PRN	ENA	D	CLK		Output
L	H	X	X	X		Q
H	L	X	X	X		L
L	L	X	X	X		H
H	H	L	X	X		Illegal
H	H	H	L	↑		Qo*
H	H	H	H	↑		L
H	H	X	X	L		H
						Qo*

\* Qo = level of Q before Clock pulse All flipflops are

positive-edge-triggered.

When the ENA (Clock Enable) input is high, the flipflop passes a signal from D to Q on the next rising clock edge. When the ENA input is low, the state of Q is maintained, regardless of the D input.

### 4.2. Oscillator

All synchronous devices need a clock. The DE1 board has a 27 MHz crystal oscillator which is connected to pin PIN\_D12 and PIN\_E12 of the Cyclone II FPGA.

### 4.3. Counter design in AHDL

Perhaps the simplest sequential system to implement is a simple counter.

```
SUBDESIGN dec_count
(
```

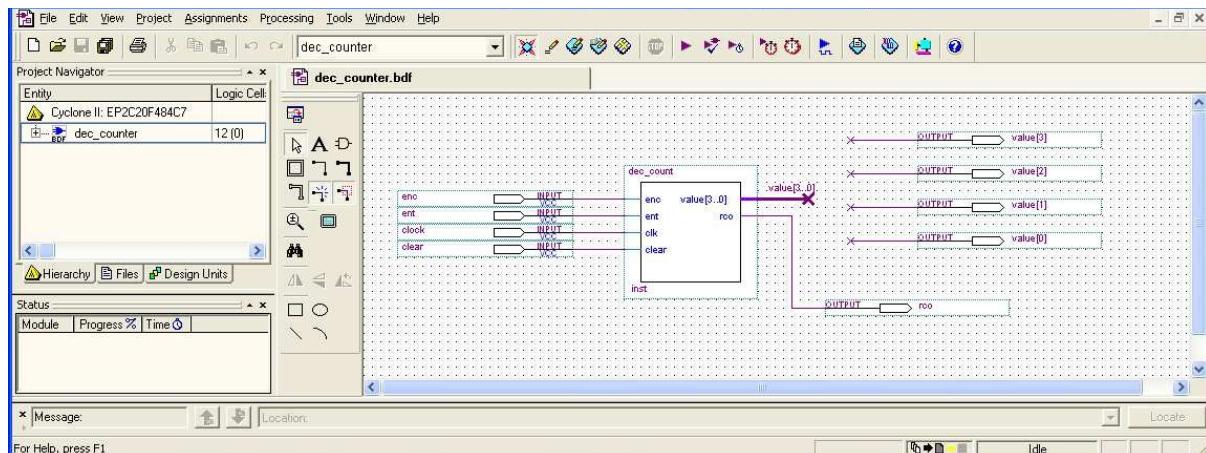
```

    enc, ent, clk      : INPUT;
clear      : INPUT;
value[3..0] : OUTPUT;           rco
: OUTPUT;
)
VARIABLE
    count[3..0] : DFF;  BEGIN
    count[].clk = clk;   value[]
= count[];  IF (clear) THEN
    count[].d = 0;
ELSIF (enc & ent & (count[].q != 9)) THEN
    count[].q + 1;
ELSIF (enc & ent & (count[].q == 9)) THEN
    count[].d = 0;
ELSE
    count[].d = count[].q;  END
IF;
    rco = ((count[].q == 9) & ent);
END;

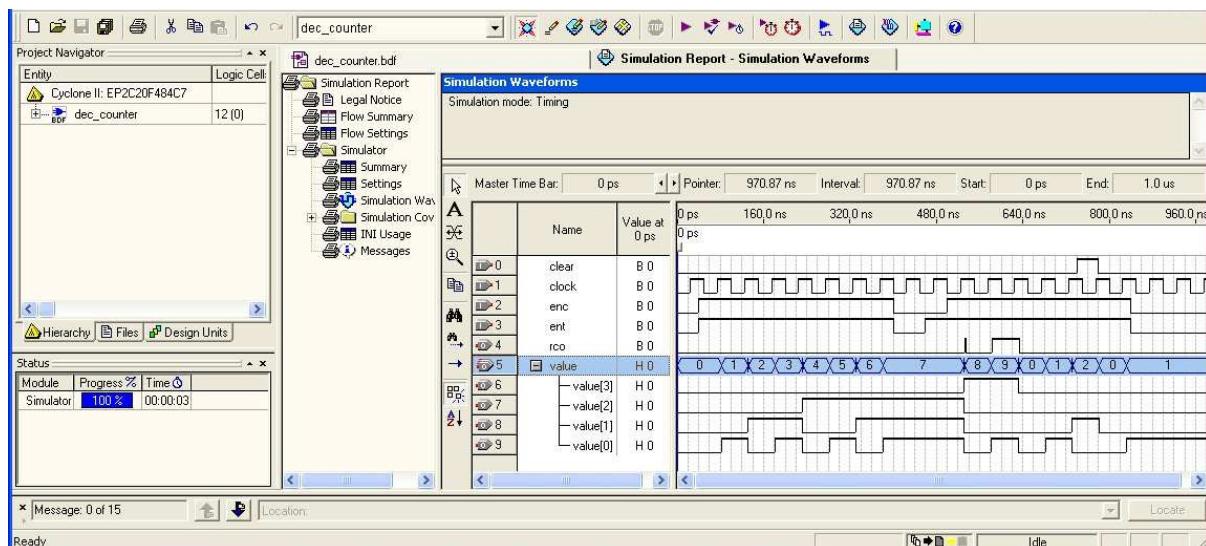
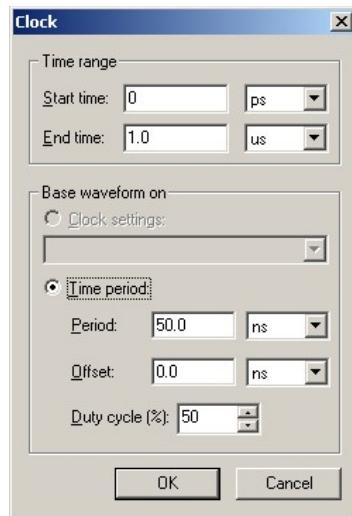
```

In the dec\_count subdesign a 4 bit counter is defined. There are four inputs consisting of the clock, two enables (ent & enc) and a clear. The outputs are the four count bits and a Ripple Carry out (rco). In the variable section 4 D type flip-flops are declared local to the subdesign. The comments in the file explain the operation.

Create a new project called dec\_counter. Enter this design in a new AHDL file called dec\_count.tdf and “Create Symbol File” to allow you to enter the symbol in a bdf file Create a BDF file called dec\_counter.bdf similar to:



Now compile and simulate the design and print out the resulting simulation file. For creating the clock waveform right-click on the clock and then select **Value | Clock**. In the dialog box set the period and duty cycle of the clock.



#### 4.4. Modified counter

If you use the 27MHz clock as the clock input and connect the counter to the input of the 7Segment decoder, you will find that all the segments of the display appear permanently on. This is what you would expect with a clock frequency of 27 MHz. If we want the counter to increment approximately every second we will need to enable the counter for one clock period after every 27 million clock pulses. We can construct another counter to do this and then use the output to drive one of the enables. (We don't use the output of the counter to drive the clock as this would make the design "Asynchronous").

```
SUBDESIGN sec_cnt
(
    clk      : INPUT;      second
    : OUTPUT;
)

VARIABLE
    count [25..0]      : DFF;

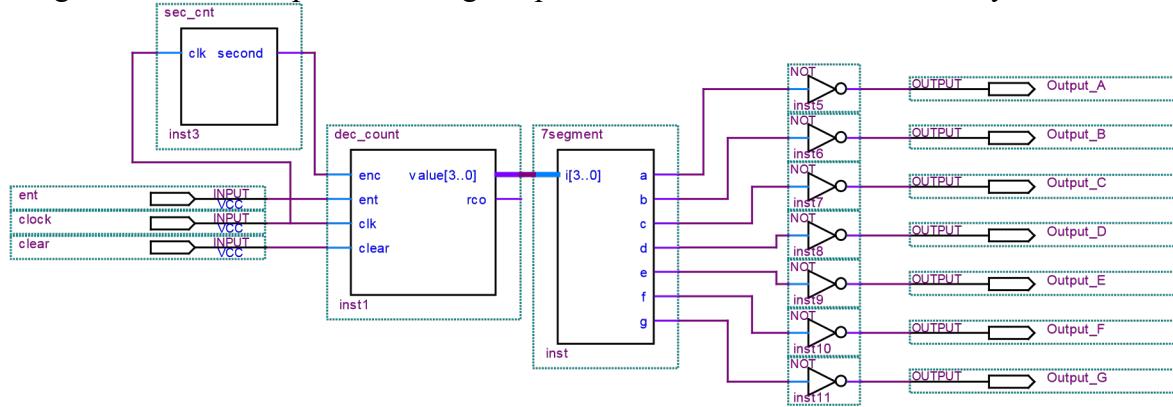
BEGIN
    count [].clk      = clk;
```

```

IF ((count[].q == 27000000)) THEN
    count[].d = 0;
second = VCC;      ELSE
    count[].d = count[].q + 1;
second = GND;
END IF;
END;

```

Create a TDF file called sec\_cnt.tdf and enter the code to produce a pulse every second. Then integrate this with the previous design to produce a circuit which counts every second.



If you try and simulate the design you will find that it takes a very long time as the counter will have to count to 27000000 before incrementing the dec\_count. Standard practice is to modify the count value for simulation purposes, i.e. change the count value from 27000000 to a more manageable value like 5. In programming and testing the design, ent is connected with PIN\_L22, clear is connected with PIN\_L21, and clock is connected with PIN\_D12. 1) Modify your design to make it a divide-by-12 counter and run it on the Altera DE1 board.

2) Now try cascading two divide-by-12 counters together and display the resulting values on both segments of the dual seven segment display (It will count up every 1 second).

## **5. Reports**

You should submit printouts of the designs you generate along with copies of any simulation waveforms. If the design functioned correctly indicate how it was tested. If the design did not operate as expected then indicate possible reasons.

The specific sections which require print outs are:

1. The design in Section 2.4
2. The simulation waveform in Section 2.6
3. The modified design in Section 2.7
4. The re-simulation in Section 2.7
5. The ***7segment.tdf*** of Section 3.9
6. The ***hexdec.bdf*** of Section 3.9
7. The simulation of ***hexdec*** in Section 3.9
8. The ***dec\_count*** in Section 4.3
9. The ***bdf*** file in Section 4.3
10. The Simulation file in Section 4.3
11. The ***sec\_cnt*** file in Section 4.4
12. The ***bdf*** file in Section 4.4
13. The Simulation waveform of Section 4.4
14. The ***bdf*** file of the divide by 12 and display on two Seven Segments.