

# **Altera's MAX+plus II and the UP 1 Educational Board**

*A User's Guide*

for

**Advanced Logic Design, CPE/EE 422/522**

**B. Earl Wells, Sin Ming Loo**

Department of Electrical and Computer Engineering

The University of Alabama in Huntsville

Huntsville, AL 35899

Version 1.1, August 21 2001

# Getting Started with Altera MAX+plus II and Altera's UP 1 Education Board - A User's Guide

B. Earl Wells

Sin Ming Loo

## Introduction

This manual is to be used as an introductory guide in the CPE/EE 422/522 Advanced Logic Design class, at UAH. It contains the basic information needed to rapidly prototype many digital designs on Altera Corporation's UP 1 Education Board [1] using the Altera MAX+plus II Computer Aided Design, CAD, tool [2]. The manual introduces the design process using a coordinated set of examples which will take the user through the most common steps necessary for design entry, functional simulation, logic synthesis, and the actual downloading of the design to configure Altera's UP 1 Education Board.

This guide is organized into five chapters which cover the following topics:

**Chapter 1:** General information is presented about the UP 1 Education Board [1] and the Altera MAX+plus II Design tools [2].

**Chapter 2:** A simple four bit binary counter design example is introduced and used to show the common steps associated with schematic capture design entry, functional simulation (including stimulus generation using the waveform editor), design implementation, and the UP-1 configuration using the Altera Flex 10K20 FPGA.

**Chapter 3:** A four bit binary to seven segment LED design example is introduced to illustrate the common steps associated with hardware description language design entry in VHDL [3,4], logic synthesis, functional simulation, and 10K20 based UP-1 implementation.

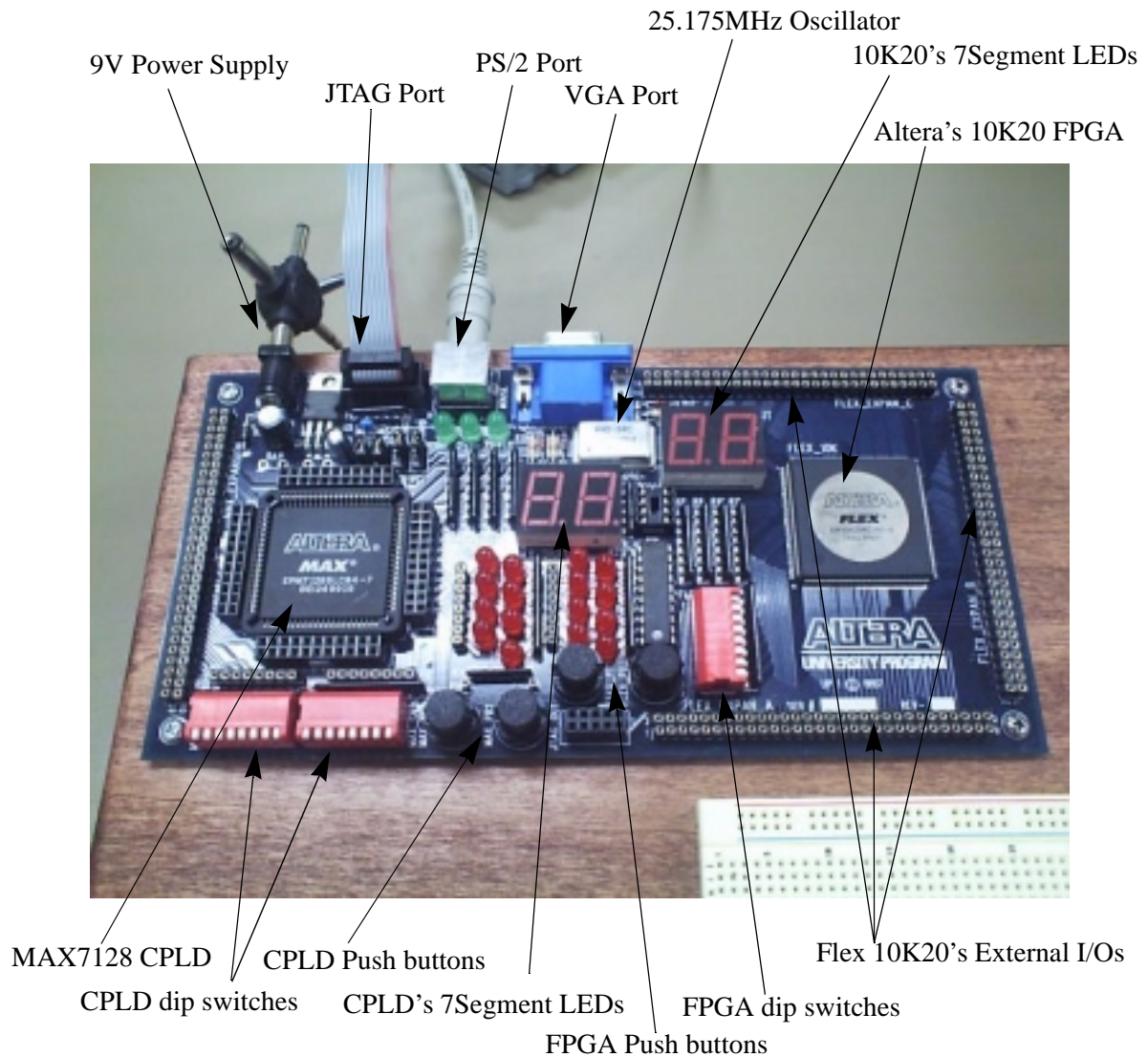
**Chapter 4:** The four bit binary counter and the binary to seven segment LED examples presented in Chapters 2 and 3 are combined to illustrate how designs can be created using hybrid schematic capture techniques and a hardware description language.

**Chapter 5:** References.

# Chapter 1: The UP 1 Education Board and Altera MAX+plus II

## 1.1 The UP 1 Education Board

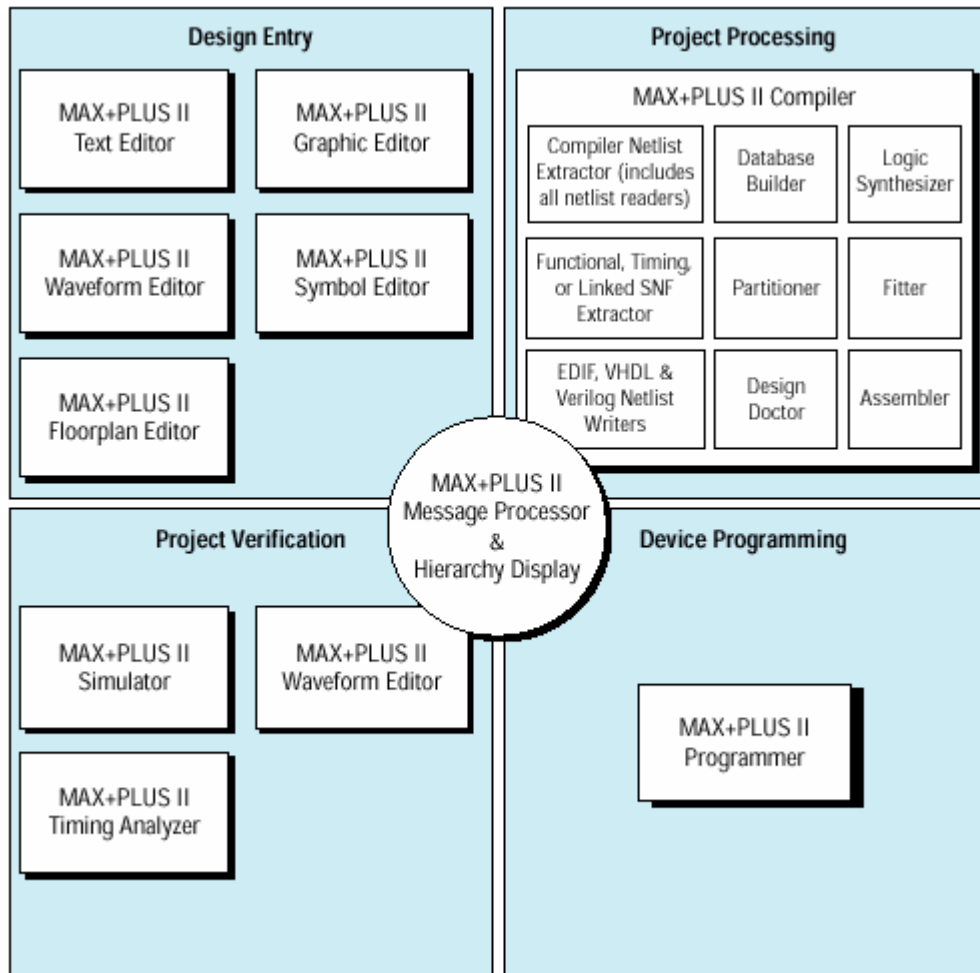
The UP 1 Education Board has many features that facilitate rapid prototyping of digital logic. This board is shown below in Figure 1.1. The UP 1 Education Board is a stand-alone experiment board that incorporates two programmable logic devices from Altera's MAX7000 and FLEX 10K line of Complex Programmable Logic Devices, CPLDs. The internal architecture of the MAX7128SLC84 is composed of PAL-like logic blocks which are interconnected together via a programmable interconnect. The FLEX 10K20RC240 is a SRAM look-up table device which is similar in function to a standard FPGA. Both devices can be programmed/configured without removing them from the system. For the purpose of this manual, the design examples will be configured to utilize the larger 10K20RC240 device but similar techniques can be applied to configure the MAX7128.



**Figure 1.1: Altera UP 1 Education Board with Flex 10K20 FPGA and MAX7128 CPLD.**

## 1.2 The Altera Max+Plus II Software

As mentioned previously, designs that are entered on the UP 1 Education Board require the use of special CAD software to configure the Flex10K20/MAX7128 CPLDs. For the designs in the Advance Logic Class students will use software that runs on standard PCs under Windows NT operating system. The general engineering design cycle which is supported by the Altera MAX+plus II CAD software is highlighted in Figure 1.2. It includes the design entry, project verification, project processing, and device programming, the function of which, will now be briefly described.



**Figure 1.2: The General Design Cycle supported by the Altera MAX+plus II Software.**

*Design Entry:* In this stage of the design cycle the design is specified in a form that is recognizable by the MAX+plus II design automation tools. Altera tools support design entry using schematic capture (graphic and symbols editor), hardware

description languages (VHDL, Verilog, and Altera Hardware Description Language -- AHDL), state diagram specification or a combination of these techniques.

*Project Processing:* Whenever a design is entered using a high-level language or state diagram specification, the design automation tool must synthesize the relatively abstract representation into its low-level logical representation. The Altera tools support this option for several HDL's (e.g. VHDL, Verilog, and AHDL). Whenever a design is entered via schematic capture a complete netlist must be generated and the components mapped to the targeted CPLD/FPGA constructs. This phase converts the design information into a form that can be used to verify by simulation or configure the targeted FPGA/CPLD device so that it will behave in the manner that is intended.

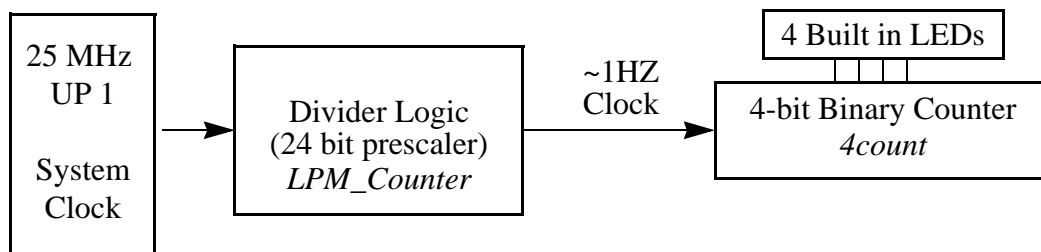
*Project Verification:* This phase of the design process allows for the logical correctness of the design to be validated before it is implemented. As design errors are exposed corrections will often be made by repeating the design entry portion of the design cycle. This phase of the design is used to verify that the resulting implementation meets timing and other constraints. This is very important for high speed designs.

*Device Programming (Device Configuring):* The resulting bit stream that was produced during the implementation phase to represent the design is downloaded directly (or indirectly through flash memory, etc.) to the targeted device. The design examples discussed in this manual will utilize the JTAG standard which is in turn supported through the use of a special programming ByteBlaster (TM) cable supplied by Altera which connects to the parallel port of a typical PC or unix workstation.

## Chapter 2: Schematic Capture Design Example

### Example

In this chapter, a simple four-bit binary counter will be used to illustrate the common steps needed to enter a digital design via schematic capture techniques. The design will be entered using the graphic editor, processed with the **MAX+plus II** software and verified using functional simulation. The designed will then be used to configure the Flex 10K20 portion of the UP 1 Educational Board. The binary counter design which will serve as a simple example is shown in Figure 2.1. In this design, the four outputs from the binary counter will each be connected (as shown later in Figure 2.10 -- using four jumpering wires) to built-in LEDs (the LEDs are pulled-up with 330 ohms resistors) on the UP 1. The counter will be clocked by a relatively slow clock (~1HZ) to allow the operation to be viewed on the UP 1 by the user. The clocking signal will originate from an external 25.175 MHz oscillator which will be directed through a 24 bit prescaler. For simply, both the 4-bit counter and 24-bit prescaler circuit will be implemented using Altera's counter primitives.



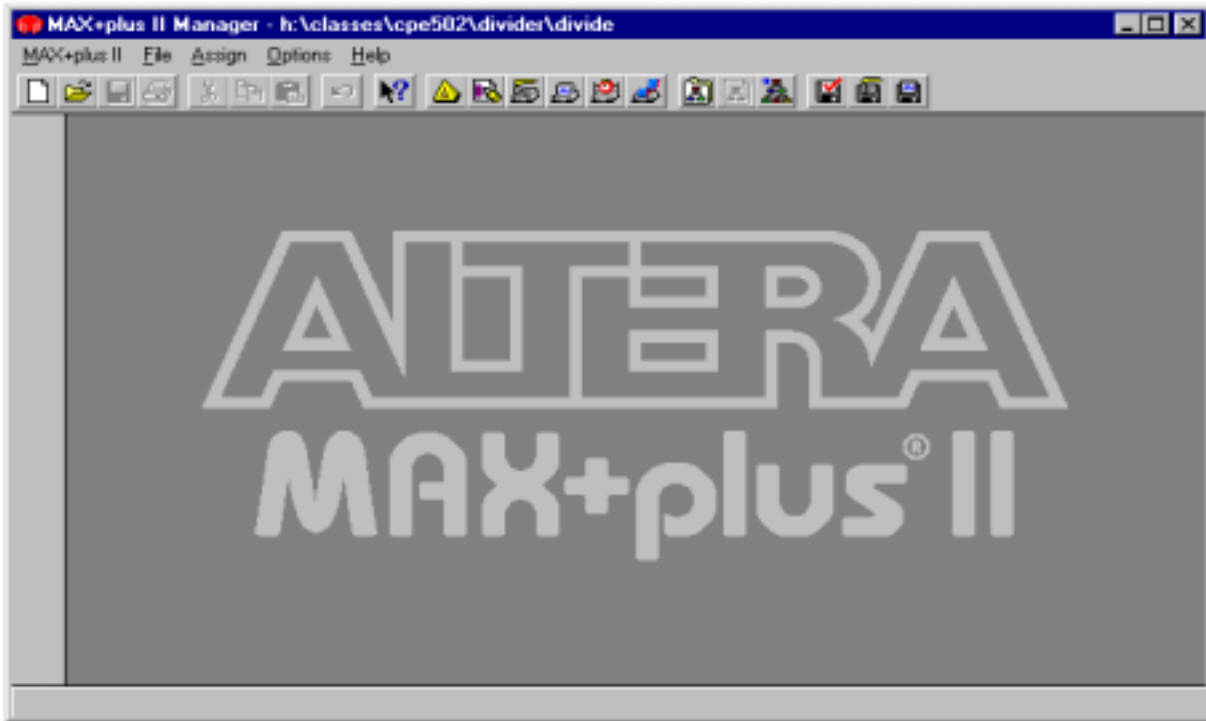
**Figure 2.1: Binary Counter Example**

### Design Entry

To begin a new logic circuit design, the first step is to create a working directory (i.e. folder) to hold the design files. This can be done using the normal utilities provided by NT. For this example you should create a folder called bcounter on the networked X: drive. Then start the **MAX+plus**

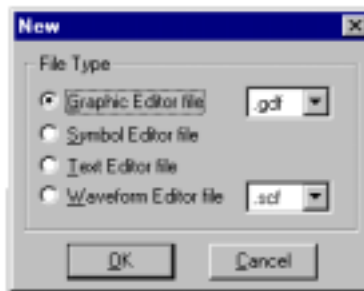
**II Manager** by double clicking on the **MAX+plus II** Icon,  from the Window's Desktop.

This window allows the user to access five pull-down windows: **MAX+plus II**, **File**, **Assign**, **Options**, and **Help** as shown in Figure 2.2.



**Figure 2.2: Altera MAX+plus II Manager**

To begin the design entry process for the binary counter design, first select the **New** option from the **File** pull-down menu. A window similar to the one shown in Figure 2.3 should appear.

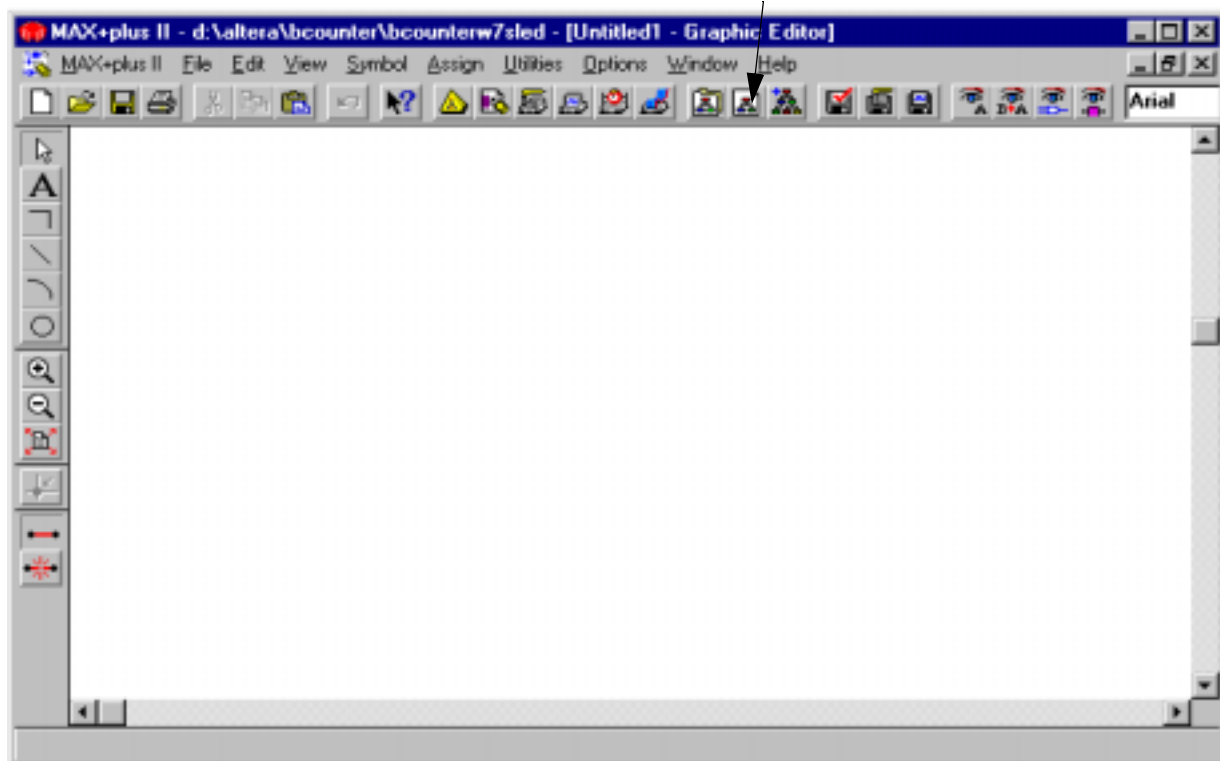


**Figure 2.3: Altera MAX+plus II Input Selection.**

From this window select the **Graphic Editor File (\*.gdf)** option and then click on the **OK** button. The graphic editor is the Max+plus II tool which supports design entry through schematic capture. A new **Graphic Editor** window will appear as shown in Figure 2.4. It will display the path name of the current working directory on the top left corner on the window. [Warning it should be noted that the default path that is displayed when one enters a new gdf file is always the path of the previous user on the same system!] For the binary counter example we will want to

give the gdf file the filename bcounter and save it under the current working directory (i.e.X:\bcounter).To do this, first select the **Save** option under the **File** pull-down menu. This will bring up the **SaveAs** dialog window. The filename bcounter should then been entered into the **File Name** field of this window and the **Automatic Extension** field should be set to .gdf.

#### Set Project to Current File shortcut button



**Figure 2.4: Graphic Editor Window**

The next step is to set up a *project* file to represent the current design. In Altera MAX+plus II, each logic circuit, or subcircuit, represents an entity that is called a *project*. Throughout the design process, the MAX+plus II software works on one *project* at a time and keeps all files generated for that project in a single directory. This can be accomplished by using the pull-down menu or by using the shortcut by two ways.

#### **Method 1: Pull-Down Window**

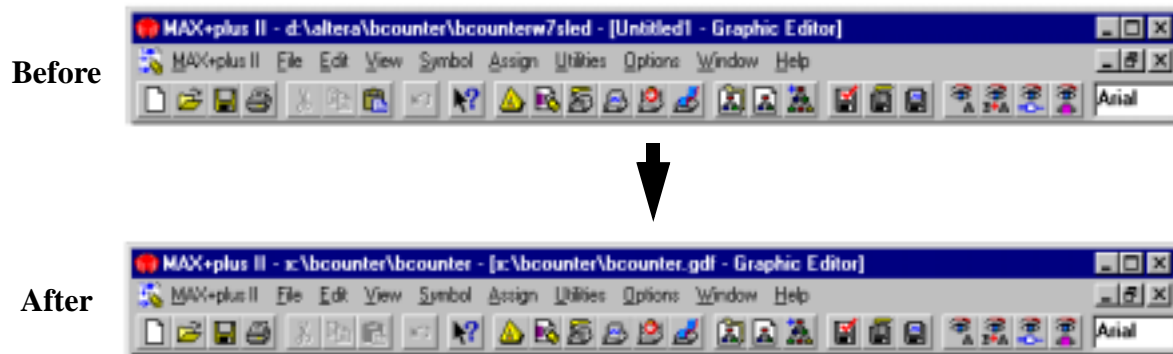
From the main **MAX+plus II** window, select the **Project** option from the **File** pull-down menu. Then go to and select the **Set Project to Current File** option.

#### **Method 2: Shortcut Button**

Click on the button as shown in Figure 2.4.



This should change the title bar of the MAX+plus II main window as shown in Figure 2.5.

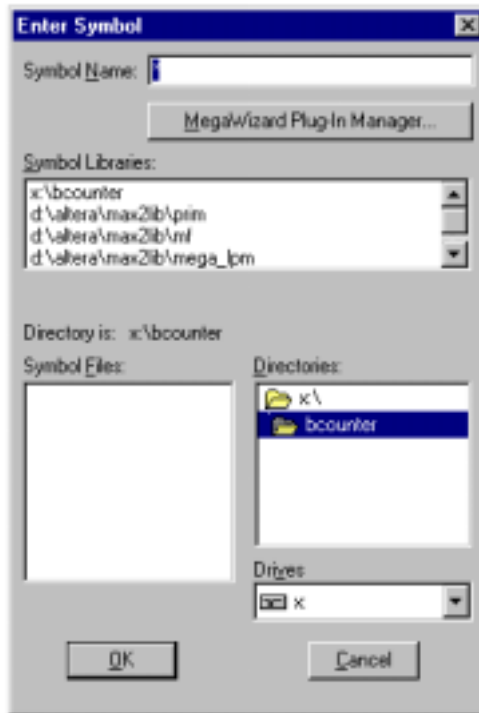


**Figure 2.5: MAX+plus II Window: Before and After a New file is added to the Project**

### Schematic Capture

Design entry using schematic capture techniques involves employing a CAD tool (such as the Altera MAX+plus II software) to enter a complete logic level schematic. This requires that the user employ the CAD tool to graphically enter the symbols that represent each of the components that make up the design and then use the tool to ‘wire’ the components to one another to form the complete schematic diagram.

The first step in the schematic capture design process is to enter each of the components which make up the design. This can be accomplished within the Altera MAX+plus II environment by double clicking the mouse with the cursor pointing on the white space of current gdf file. This will launch the **Symbols/Components Selection Window** as shown in Figure 2.6. This window contains the high-level macros and low-level primitive components which can be included in the design.



**Figure 2.6: Component Selection Window**

At the time of this writing the following set of component libraries were supported by the MAX+plus II software:

*prim* (primitive library) - A set of basic functional blocks used to design circuits with MAX+plus II software. The primitives include buffers, flip-flops, latches, input and output primitives, and logic primitives.

*mf* (megafunctions) - A set of complex or high-level building blocks (i.e. macros) that can be used together with gate and flip-flop primitives in MAX+plus II design files.

*mega\_lpm* (library of parameterized modules) - A technology-independent library of logic functions that are parameterized to achieve scalability and adaptability.

*EDIF* (Electronic Design Interchange Format) - vendor independent primitives useful when transporting a schematic capture based design from one CAD tool to another. This library includes such primitives as AND, NAND, OR, NOT, XOR2, DELAY, TRI, LATCHes, FLIP-FLOPs, NOR, XNOR2, FILTER, and RiseFall.

The binary counter example will require that the following components be extracted from the *prim* library. First, an **Input** symbol will be needed for the clock input and a set of four **Output** symbols will be needed for the four-bit binary counter output. A 4-bit binary counter (**4count**)

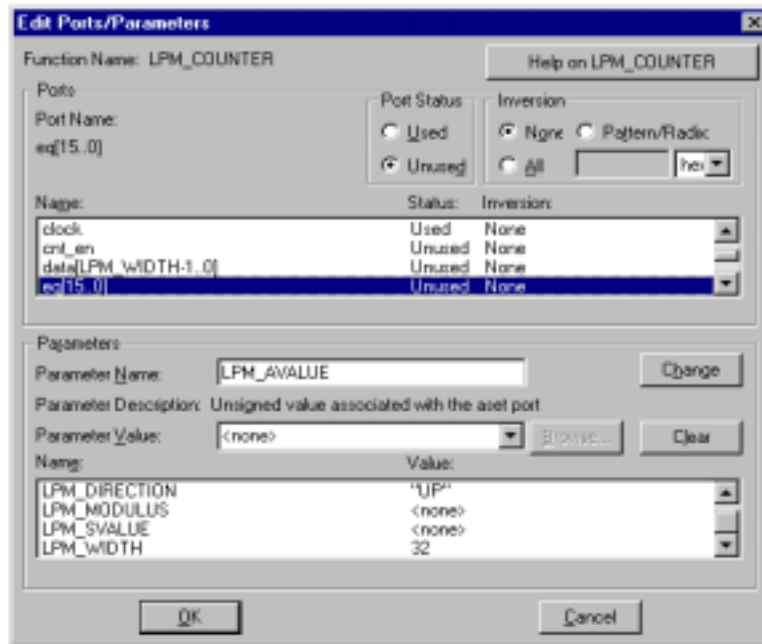
needs also to be extracted from the *mf* library to implement the binary counter portion of the design, and a *LPM\_Counter* needs to be extracted from *mega\_lpm* library to implement the prescaler. Finally, several instances of the components **Vcc** and **gnd** are needed to effectively tie various signals to logic high or logic low, respectively.

To extract these new components, first go to the **Symbols/Components Selection Window** and double click on the appropriate library, then go to the **Symbol Files:** section of the window and select the appropriate symbol from the list of symbols. Then click on the **OK** button. Place each symbol on the schematic at the desired location using the mouse.

The *LPM\_Counter* symbol is a parameterizable macro. When this component is selected as shown in Figure 2.7, another window will appear. This window shows the various ports and parameters that define the specific instance of the *LPM\_Counter*. For the case of the binary counter example, one should set all entries in the **Port's Name** window to the status of *unused* except the **Clk** and **q[LPM\_WIDTH-1..0]** ports. This is done by first clicking on each entry using the left mouse button and then go to **Port Status** area of the window and clicking on the **Unused** option.

Under the **Parameters** area of this window set the **LPM\_Direction** parameter to **UP** and the **PM\_WIDTH** parameter to 32. Then click on the **OK** button. One can always make changes to these parameters later by pressing the right mouse button when the cursor is placed on the desired

component from within the **Graphics Editor** and then selecting the **Edit Ports/Parameters** option.



**Figure 2.7: LPM\_Counter Ports and Parameters.**

When all the symbols/components have been extracted from the libraries one can always replicate these symbols by using Window’s built-in copy, cut, and paste utilities.

Components are wired together by moving the cursor over to a wire or bus connection point on the symbol (component). At this point in time the cursor will turn into a ‘+’ sign. The left mouse button is then pressed and held and the cursor is then moved over to the desired termination point and the mouse button is released. In this process other wires and busses can act as starting and termination points.

Individual wires (i.e. nodes) can be brought out from collections of nodes (i.e. busses) simply by giving the node the same **name** as the desired component of the bus. To name a node (wire) or a bus, simply double click on the wire or bus using the left mouse button and type in the node or bus name. Bus names usually have the following format:

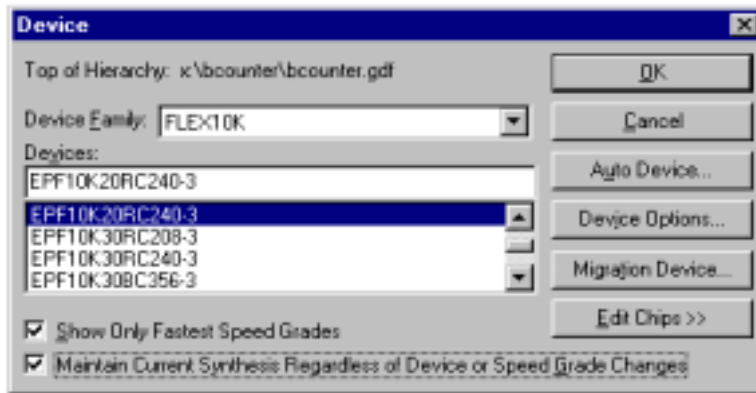
**bus name[most significant element number .. least significant element number]**

For example, in the binary counter design there is a simple bus named **q[31..0]**. This implies that there are 32 nodes contained within the bus that are individually named **q31, q30, ..., q1, q0**, respectively. To logically connect a given net to a bus, such as **q24** to the bus **q[31..0]**, as shown for the binary counter example in Figure 2.8, requires that the bus and wire be first created and named in the manner discussed above.

For the binary counter example one can complete the schematic by connecting the components as shown in Figure 2.8. Note that there are bubbles at the end the **QA:QD**, outputs of the *4count* symbols. These bubbles indicate that the signals are inverted. This is required because the discrete LEDs on UP 1 that are being driven in this example are all pulled-up with 300 ohms resistors which means a logic 0 is required to light them up. To set these bubbles on the *4count* symbol, first click on the *4count* symbol, and then click with the right mouse button to select the **Edit Ports/Parameters** option. Then select **QA**, **QB**, **QC**, and **QD** and choose the **Inversion to ALL** option.

The binary counter design has one global input, a clock signal, and four global outputs, the outputs of the 4 bit binary counter *4count* module. These global I/O nodes are interfaced to the outside world through the use of special **Input** and **Output** component symbols as shown in Figure 2.8. These symbols need to be labeled to identify their function for documentation and simulation purposes. This is done by moving the cursor onto the **Input** or **Output** symbol, right clicking with the mouse and selecting the **Edit Pin Name** option or by simply double clicking with the left mouse button on the existing I/O label (which defaults to the value “PIN\_NAME” when the drawing is being created for the first time). For the binary counter example, follow this

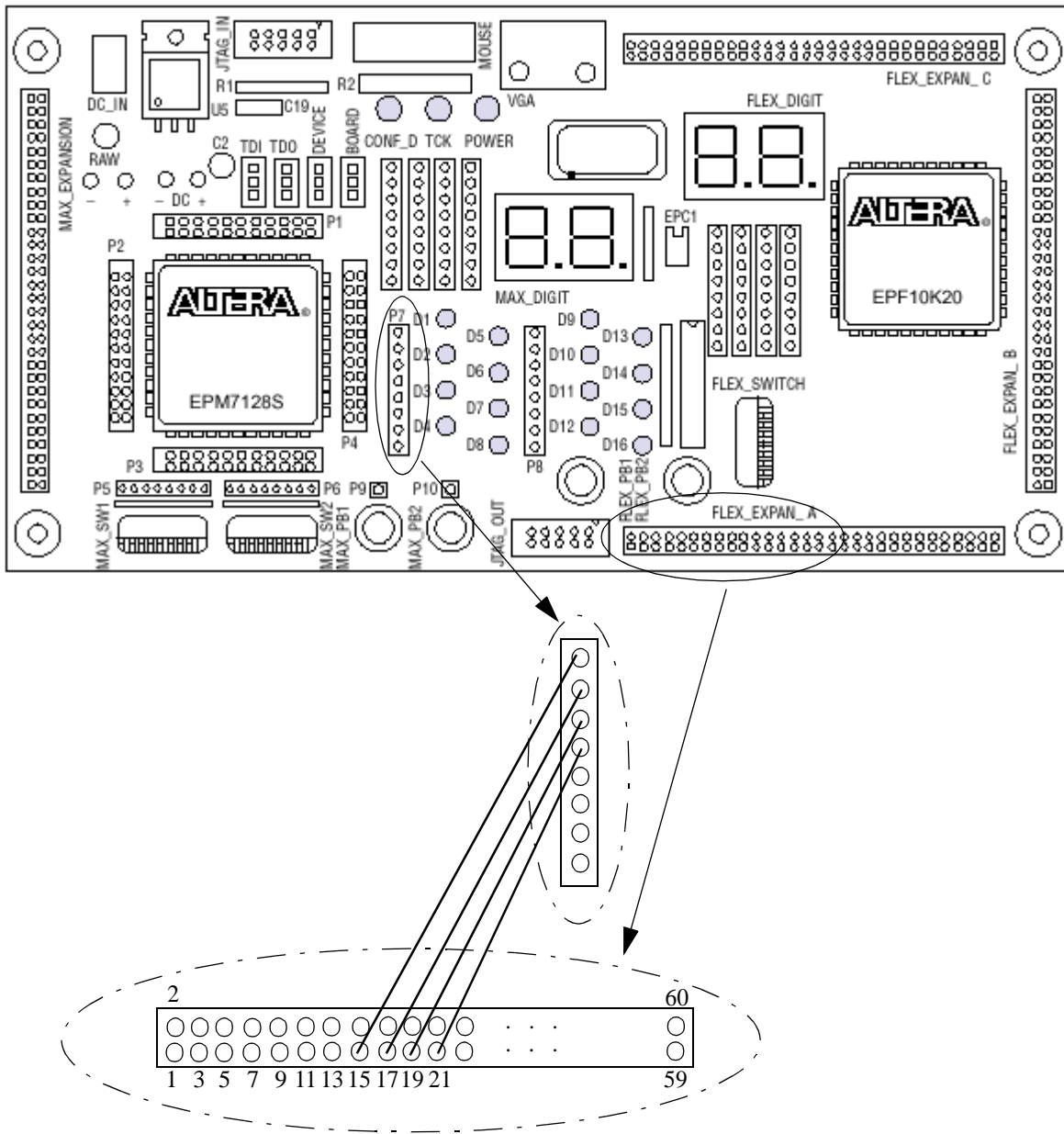




**Figure 2.9: Device Assignment Window**

### Constraints Entry

After the design is entered and saved and the programmable logic device has been assigned, it is the job of the MAX+plus II software to convert the user-entered schematic into a form that can be used to configure the targeted FPGA/CPLD device so that it will behave in the manner that is intended. This conversion process results in the mapping of the desired logical configuration into a form that can be implemented within the internal FPGA/CPLD architecture. It is at this point that users have the option of specifying certain constraints which the complex mapping algorithms (place and route) must adhere to when making the implementation files. There are a number of such constraints which can be specified, but the one that is of most importance is usually the assignment of which pins on the Altera FPGA/CPLD chip are going to be mapped to the input/output nets of the design. In the binary counter design, there is only one input pin that runs from the UP 1 Education Board clock circuit into the Flex 10K20 chip and there are four output pins that are to connect the Flex 10K20 chip to the discrete LEDs (via four external jumper wires connected in the manner shown in Figure 2.10).



**Figure 2.10: External wires connection from Flex 10K20 to LEDs**

For each programmable logic device that is supported by Altera there is a distinct pin number that is associated with each I/O pin on the chip. A cross reference can be created which specifies which pin number on the FPGA/CPLD is to be assigned to the logical I/O component pin name used in the design. If such information is not present, the Altera software will arbitrarily assign FPGA/CPLD pins to the schematic I/O nodes. (Note: there is a general rule -- the more



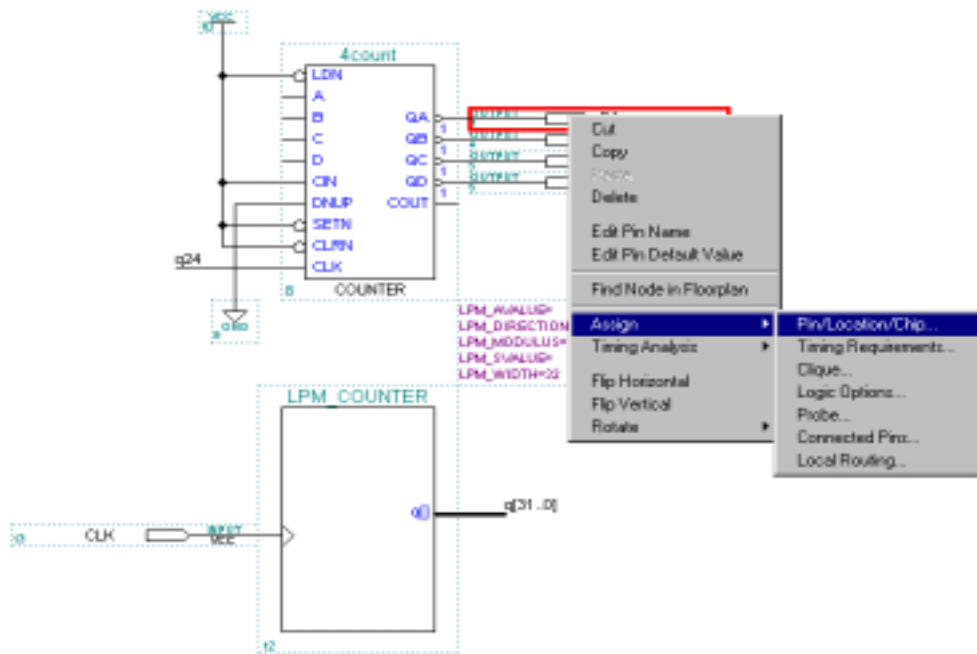
constraints entered by the user the more inefficient the implementation processes is in terms of processing time and complexity of the design that can be implemented in a given FPGA/CPLD. In tightly packed/high speed cases, it may be desirable to have the Altera software make its own I/O pin assignments and external PC board circuitry can then be designed to adhere to this placement.)

For the binary counter design we will lock all I/O component pin names to specific pin locations of the 10K20 device. The desired schematic node name to MAX+plus II pin number cross reference (i.e. Pin Locks) is shown in Table 2.1. (A full listing of pin numbers for the Flex 10K20 device and the UP 1 can be found in [1]).

**Table 2.1: Desired Pin Locking (Cross Reference) Configuration**

I/O Pin Description	Schematic I/O Pin Name	Flex 10K20 Pin Number	Cross Reference to UP 1
UP 1 25.175 MHZ Clock	CLK	91	Directly to 91
UP1 Discrete LED, D1	D1	45	Flex_Expan_A 15
UP1 Discrete LED, D2	D2	48	Flex_Expan_A 17
UP1 Discrete LED, D3	D3	50	Flex_Expan_A 19
UP1 Discrete LED, D4	D4	53	Flex_Expan_A 21

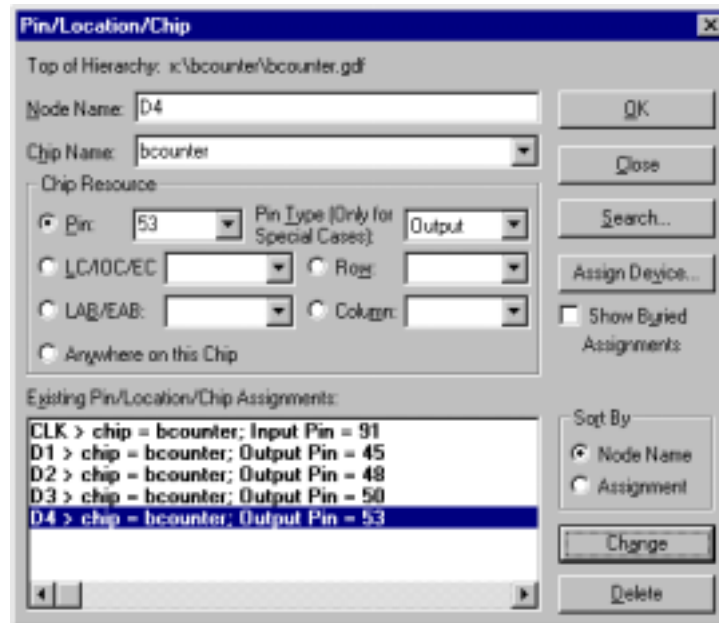
In the MAX+plus II environment, the assignment of specific 10K20 pin numbers to schematic I/O pin names occurs from within the graphic editor. First the schematic is entered and an **Input** or **Output** symbol is selected with the mouse using the right mouse button. Then the **Pin/Location/Chip..** option is selected from under the **Assign** menu as shown in Figure 2.11.



**Figure 2.11: Device assignment.**

This launches the **Pin/Location/Chip** window which is shown in Figure 2.12. The parameters that are needed to lock the pins are: **Node Name**, **Pin** (number), and **Pin Type** (input/output). (The **Chip Name** should automatically be set to the project name). The **Node Name:** field should initially contain the I/O pin name of the **Input** or **Output** component which was initially selected to bring up this window. To lock a given schematic pin name to a particular 10K20 pin number simply requires that the user enter the desired pin number in the **Pin:** field, set the **Pin Type:** attribute, and click on the **Add** button. (After this, the **Add** button will turn into a **Change** button to allow for further editing of the existing parameter.) This completes the pin locking for the selected **Input** or **Output** component in the schematic. To lock the remaining pins, the **Pin/Location/Chip** window can be **Closed** and the process repeated by following these steps after selecting a new I/O symbol. A better method, though is probably to remain in the **Pin/Location/Chip** window and repetitively enter new schematic I/O symbol names in the **Node Name:** field and then enter the associated 10K20 pin number in the **Pin:** field for each I/O symbol until all of the desired pins have been locked. Figure 2.12 shows the case where all five I/O pins for the binary counter example have been locked to specific 10K20 pin numbers which were defined in Table 2.1.

When all the desired pins are locked, one should exit the **Pin/Location/Chip** window by pressing the **OK** button.



**Figure 2.12: Pins assignment.**

## Design Processing (compilation)

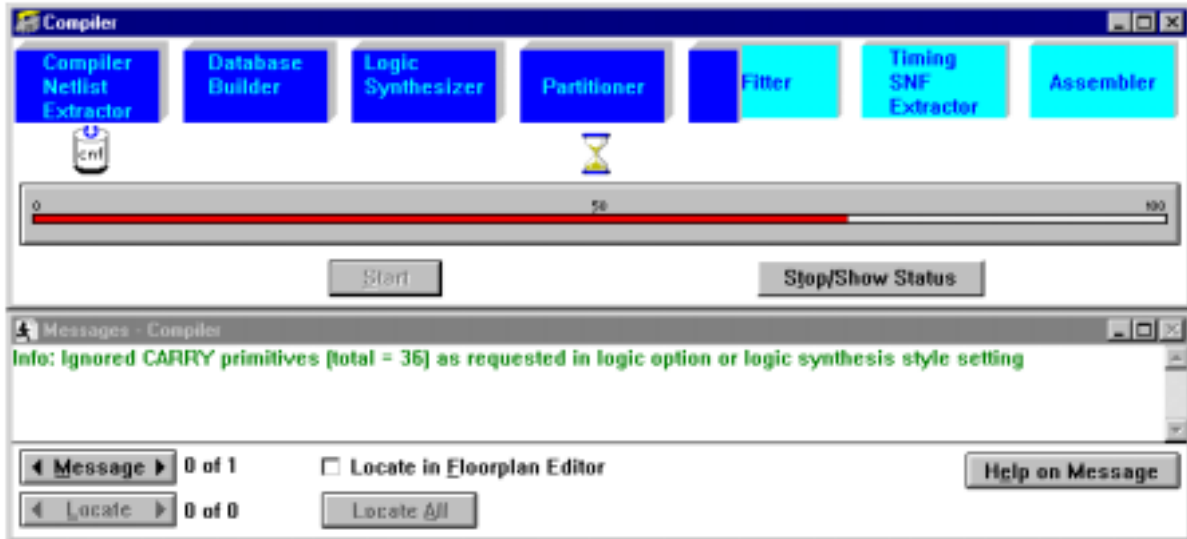
The next step in the design process is the project processing/compilation phase, in which the schematic file (\*.gdf) will be checked for design rule violations (wire connection faults, proper naming of pins, etc.), converted to a netlist and the bit file that is used to configure the FPGA/CPLD device will be created. This process can be activated as shown in Figure 2.13 by pressing the **Compilation Button** on the tool bar or by selecting the **Compiler** option from the **MAX+plus II** pull-down window from within the graphics editor.



**Figure 2.13: Compilation.**

If the **Compilation Button** is used, the compilation process will start automatically as shown in Figure 2.14. If the compiler is invoked using the pull-down window, then one must click on the

**Start** button on the **Compiler** window as shown in the figure. If there are errors or warning messages, double clicking on the particular message will provide additional information about the nature of the error.



**Figure 2.14: Compilation process.**

Errors in this process will most likely require modifications to the original design using the graphics editor. When the project processing/compilation process has been completed successfully, the message shown in Figure 2.15 should appear.



**Figure 2.15: Message.**

Click on the **OK** button to continue. The design is now ready to be simulated.

## **Design Verification (Functional Simulation)**

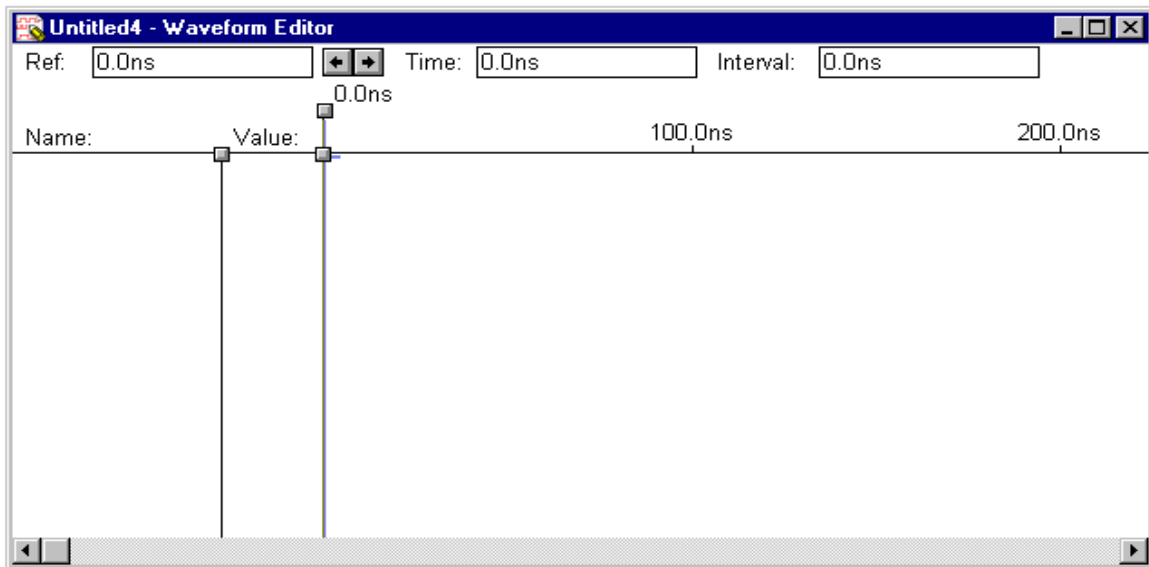
We are now at the point in the design cycle where the design has been entered and the pin locking constraints have been specified. The next most common step is the **Functional Simula-**

**tion** phase, in which one is able to check the logical correctness of a design before it is actually implemented on the targeted system (which will be in this case the Altera UP 1 board). Design errors found in this phase can then be corrected in an iterative manner by re-entering the design entry phase. While this phase can be bypassed with the behavior of the design being observed after the device is configured, this is not recommended since simulation often gives a much more detailed view of the operation of the design than is usually observable directly from the implementation. There are also some design errors that can be identified through simulation which might be destructive if the designed is actually implemented in hardware. Simulation is a powerful tool. The following is a brief overview of how simulation can be used to verify the functionality of the binary counter example. A much more detailed discussion of this material is presented in the Altera literature.

### **Selecting Observation Points**

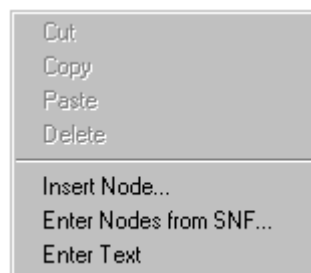
Before a functional simulation can be run, the user needs to select a set of logical nodes on the schematic which should be observed to verify the correct operation of the design. These nodes often include the global input/outputs of the design (which are usually assigned to specific device pin numbers) as well as certain internal nodes and component pins. To select the observation points from the Altera environment simply choose the **WaveForm Editor** option from the

**MAX+plus II** pull down menu. This will launch the **Waveform Editor** window which should appear as shown in Figure 2.16.



**Figure 2.16: Waveform Edi-**

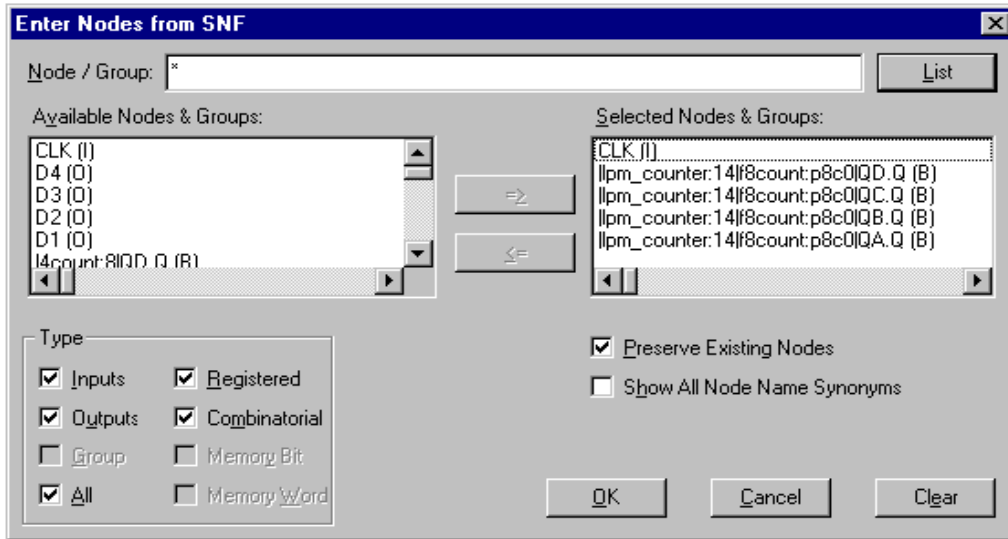
Then point the cursor to the white space under the time bar and press the right mouse button. This should bring up the window shown in Figure 2.17.



**Figure 2.17: Select Enter Nodes from SNF.**

From this window, select the **Enter Nodes from SNF ...** option which will launch the **Node Selection** window as shown in Figure 2.18. This window allows the user to choose the set of signals to be observed during the simulation (such as the global input and output nets/busses of the design as well as certain internal signals that are present). For the binary counter example, we will choose for demonstration purposes to observe the global **CLK** input and the set of internal

buss signals which drive the **q[3..0]** segment of the first bus (i.e. the least significant four bits of the prescaler).



**Figure 2.18: Nodes Selection Window**

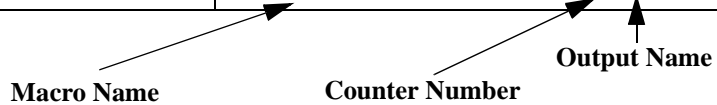
To select the signals which need to be observed, first select the **All** option under the **Type** area of the **Node Selection** window as shown in Figure 2.18, then click on the **List** button. A set of signals will appear under the **Available Nodes & Groups** section of the **Node Selection** window. These signals will include the five global I/O signals, **CLK**, **D1**, **D2**, **D3**, **D4** as well as a number of internal signals which correspond to the external I/O pins on the components (as well as to certain internal logical nodes which are used by Altera to implement the simulation model for the macros). For the binary counter example, we would like to observe the global **CLK** signal and the first four bits of the prescaler circuit. This is easily accomplished for the global clock signal by first single clicking on the **CLK** symbol which can be found under the **Available Nodes & Groups** section of the window and then clicking on the => button. The **CLK** signal should now appear under the **Selected Nodes & Groups** section of the window as shown in Figure 2.18.

Selection of the other signals to observe occurs in a similar manner but the signals to select are much less obvious. In the binary counter case, we want to select the lower order four bits of the **LPM\_Counter** component macro but this macro is represented within the Altera simulation software as four cascaded 8-bit counters of type **8count** which are labeled from 0 to 3, with 0 being the least significant 8-bit counter. The outputs of each 8-bit counter are labeled **QA** through **QH**, with **QA** being the least significant bit of the counter. We therefore want to select output bits

**QA** through **QD** of binary counter 0 in order to observe the four least significant bits of the prescaler. Table 2.2 shows the internal labeling of these four signals.

**Table 2.2: Altera Internal Signal Cross Reference**

Description of Output Signal to be Observed	Altera Signal Name
Least Significant Bit of Prescaler, <b>q0</b>	llpm_counter:14lf8count:p8c0lQA.Q (B)
Next Least Significant Bit of Prescaler, <b>q1</b>	llpm_counter:14lf8count:p8c0lQB.Q (B)
Next Most Significant Bit of Prescaler, <b>q2</b>	llpm_counter:14lf8count:p8c0lQC.Q (B)
Most Significant Bit of Prescaler, <b>q3</b>	llpm_counter:14lf8count:p8c0lQD.Q (B)



To select these signals to observe, continue as before by first single clicking on the desired signal from under the **Available Nodes & Groups** section of the window and then clicking on the => button. When all four signals have been selected, they should appear as shown in Figure 2.18. When the process is complete, click on the **OK** button to return to the **Waveform Editor**.

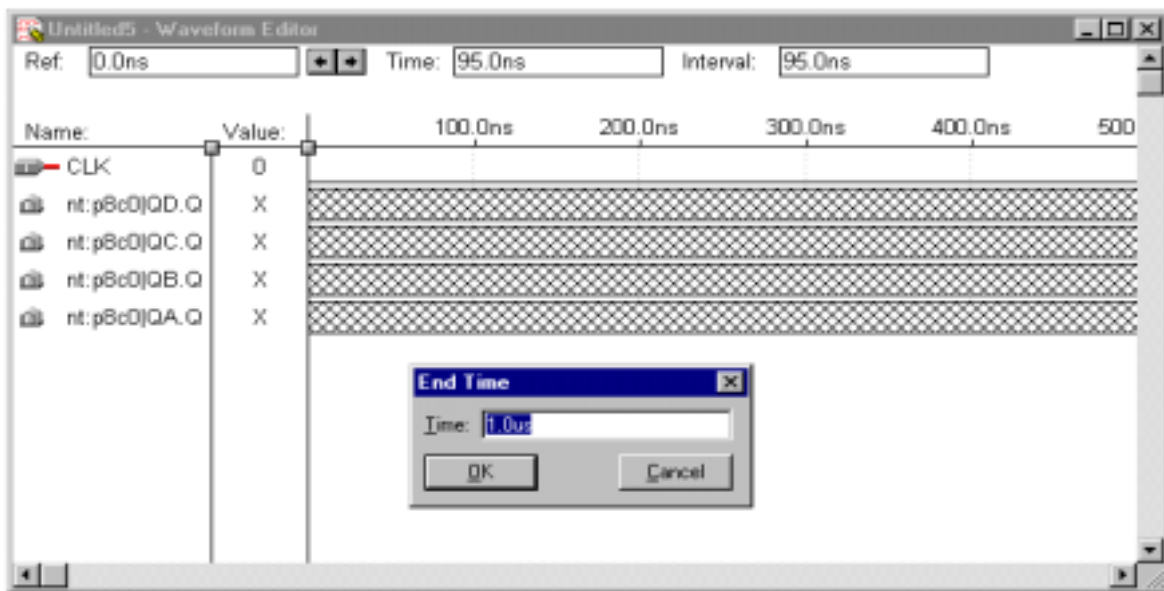
It should be noted that the four prescaler output signals are not really a good set of signals to observe for this design, but complete simulation runs which would exercise the main counter output lines, **D1-D4**, would take several hours to complete. The large amount of computer simulation time required to simulate sequential circuits is one of the major drawbacks of simulation. One could get an idea of how well the circuit works though by running the following simulation and then temporarily modifying the design to remove the prescaler.

### Stimulus Entry

Before a design can be simulated, the logical inputs to the design for each point in time need to be defined. This process is often called design stimulation. The Altera MAX+plus tool has a special editor to allow the user to enter this stimulus information in a graphical manner. Entering the appropriate stimulus patterns is important in the areas of design evaluation and test. To fully exercise a design, such stimulus patterns can often be very lengthy and complex. In the case of the binary counter example, though the only stimulus that is needed is a periodic clock. The following illustrates how to enter such stimulus to allow for subsequent functional logic simulation.



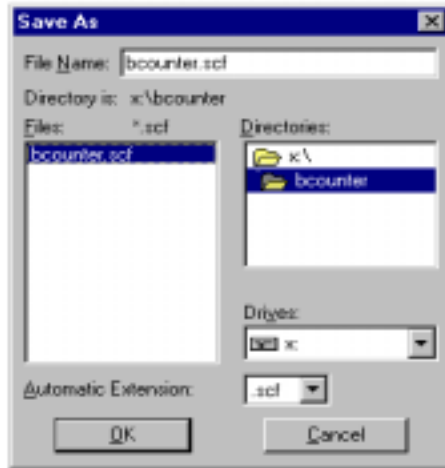
To enter new input stimulus after selecting the observations points as described in the previous section, one should make the **Waveform Editor** window the active window. This window should then show all the signals which were previously selected as observation points (see Figure 2.19 for the binary counter case). The Altera CAD tool requires that the simulation end time be entered before the stimulus is entered. To do this from the **Waveform Editor**, select the **End Time** option from the **File** pull-down menu. This should launch the **End Time** window which is shown in Figure 2.19. Then for the binary counter example, enter 1.0us in the dialog box and click on the **OK** button.



**Figure 2.19: Signals for simulation.**

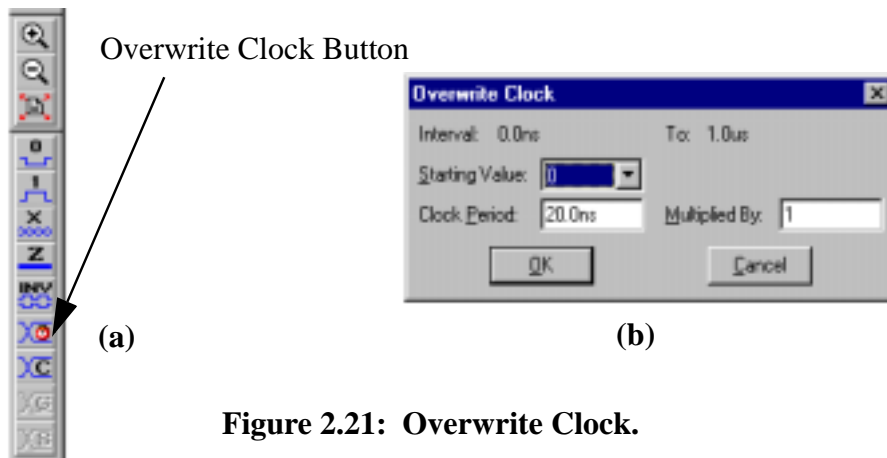
At this point the waveform information must be saved (without saving the waveform file the **MAX+plus II** CAD tool will not allow stimulus information to be entered). To do this, select the **Save As** option from the **File** pull down menu which is on the **MAX+plus II** tool bar. Save

the files as a *.scf* file. The file should have the same name as the *gdf* file but the extension will be *.scf* (see Figure 2.20 for the binary counter example).



**Figure 2.20: Saving Waveform file.**

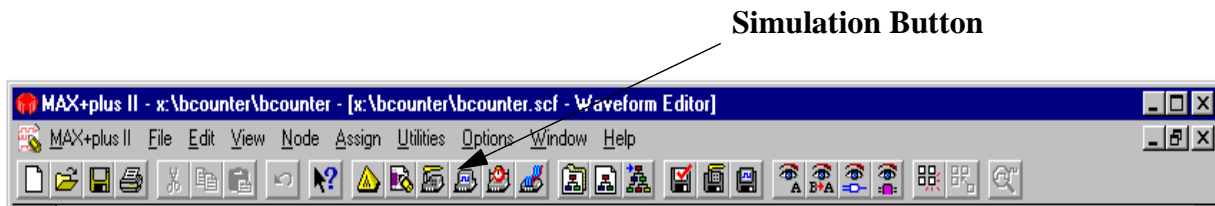
In the case of the binary counter example, we can now apply a periodic stimulus to the **CLK** signal. To do this, select the **CLK** signal from within the **Waveform Editor**, then click on the **Overwrite Clock** shortcut button which is on the vertical tool bar as shown in Figure 2.21(a). This will launch the **Overwrite Clock** window which is shown in Figure 2.21(b). This window will allow one to enter the starting logic value of the clock and its period. For the binary counter example, one should enter 20 ns in the **Clock Period** entry area of this window and then press the **OK** button. (If the **Clock Period** entry area is not activated, then click on the **cancel** button. Then make sure that the **Snap on Grid** option is not selected, This can be disabled using the **Option** pull down menu of the main **MAX+plus II** window.)



**Figure 2.21: Overwrite Clock.**

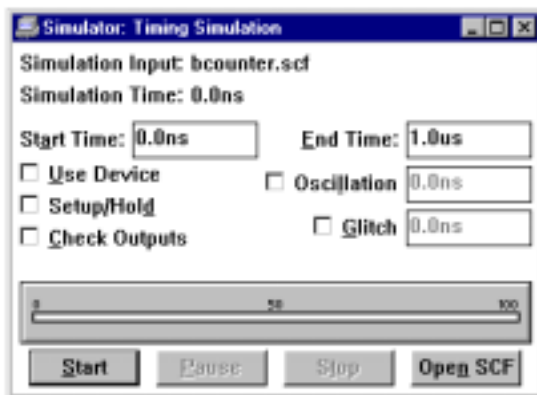
## Functional Simulation

To start the functional simulation simply click on the **Simulation Button** which is located on the horizontal tool bar of the main **MAX+plus II** window as shown in Figure 2.22.



**Figure 2.22: Start Timing Simulation.**

A **Simulator** status window will appear as shown Figure 2.23(a). Then click on the **Start** button to run the simulation. A message will appear which is similar to the one shown in Figure 2.23(b) after completion. Press the **OK** button to continue.



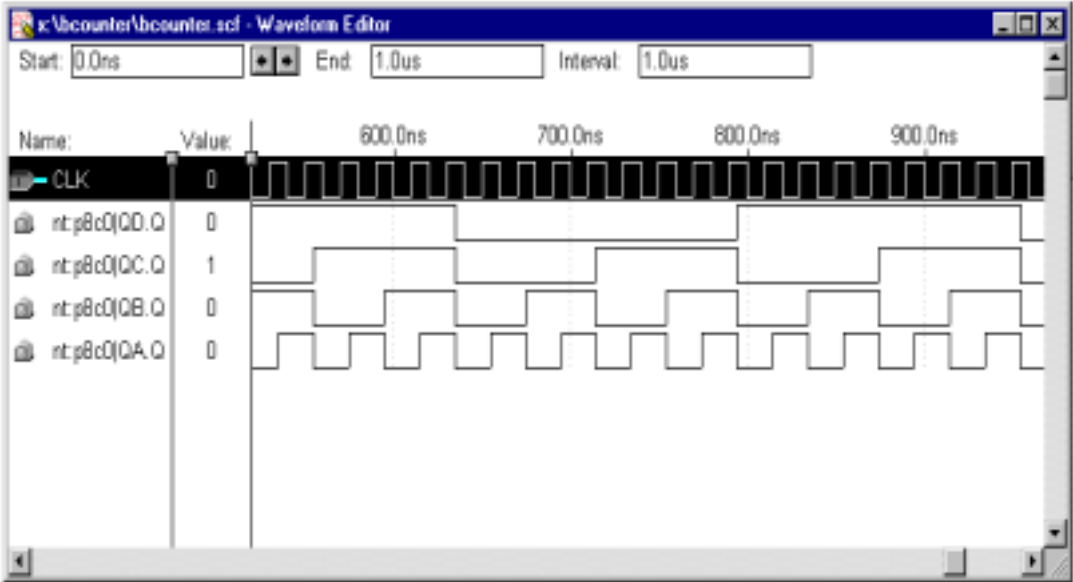
(a)



(b)

**Figure 2.23: Simulator.**

Figure 2.17 shows the waveforms that occur during the binary counter simulation. Notice that the lower order bits of the prescaler are counting in the binary sequence as expected.



**Figure 2.24: Simulation Results: Binary Counter Example**

## Device Programming (configuration)

After the design has been entered, processed, and verified, the next step is to actually download the design to configure the targeted hardware (in this case the 10K20 FPGA/CPLD that is on the Altera UP 1 Educational Board). This is done by using a special Altera ByteBlaster (TM) cable which is connected to the JTAG port of the UP 1 board and the parallel port of the host PC. It is required that certain jumpers be installed before the JTAG port can be used. Figure 2.25 shows the part placement diagram for the Altera UP 1 Educational Board. These jumpers must be set as shown in the figure in order to support the JTAG method of downloading configuration data into the 10K20.

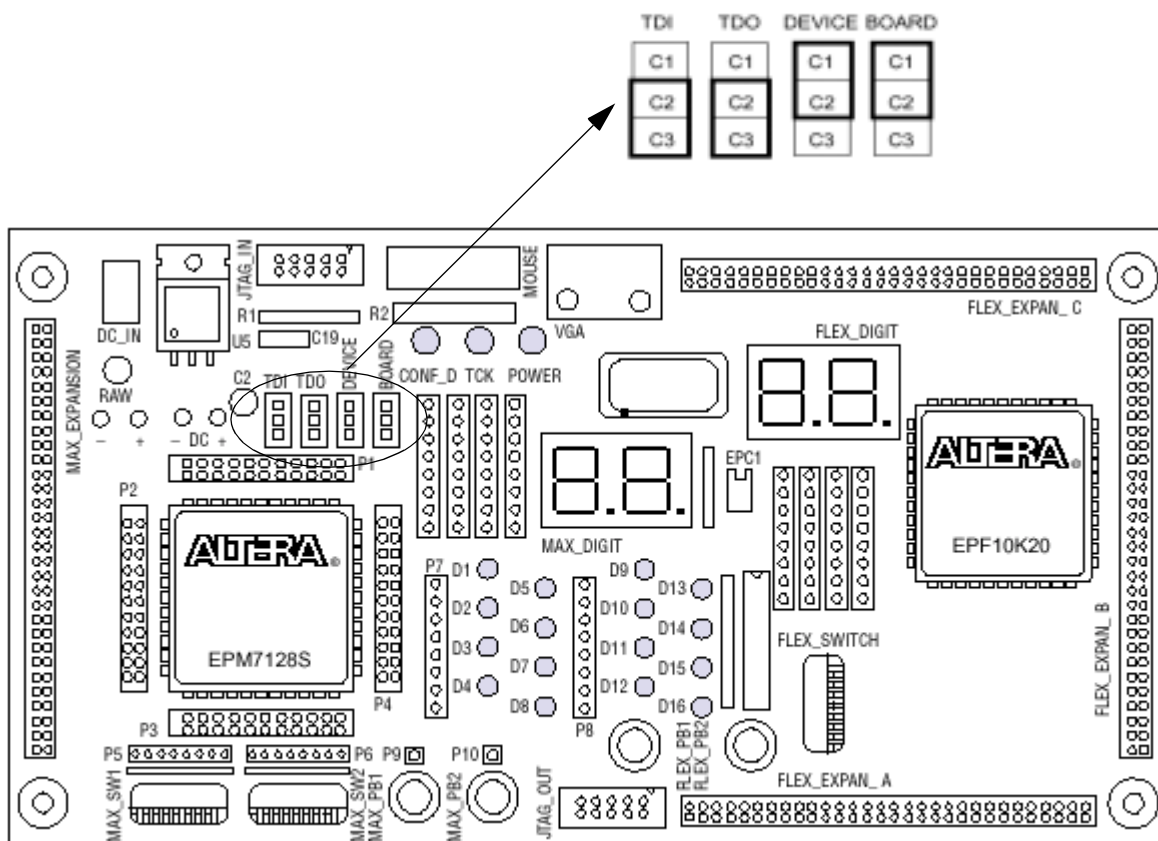


Figure 2.25: Jumper settings for configuring only the 10K20 device.

After the UP 1 board has been properly connected and jumpered as described above, the next step is to begin the download process. To do this, first click on the **Programming Button** which is on the horizontal **MAX+plus II** tool bar as shown in Figure 2.26.



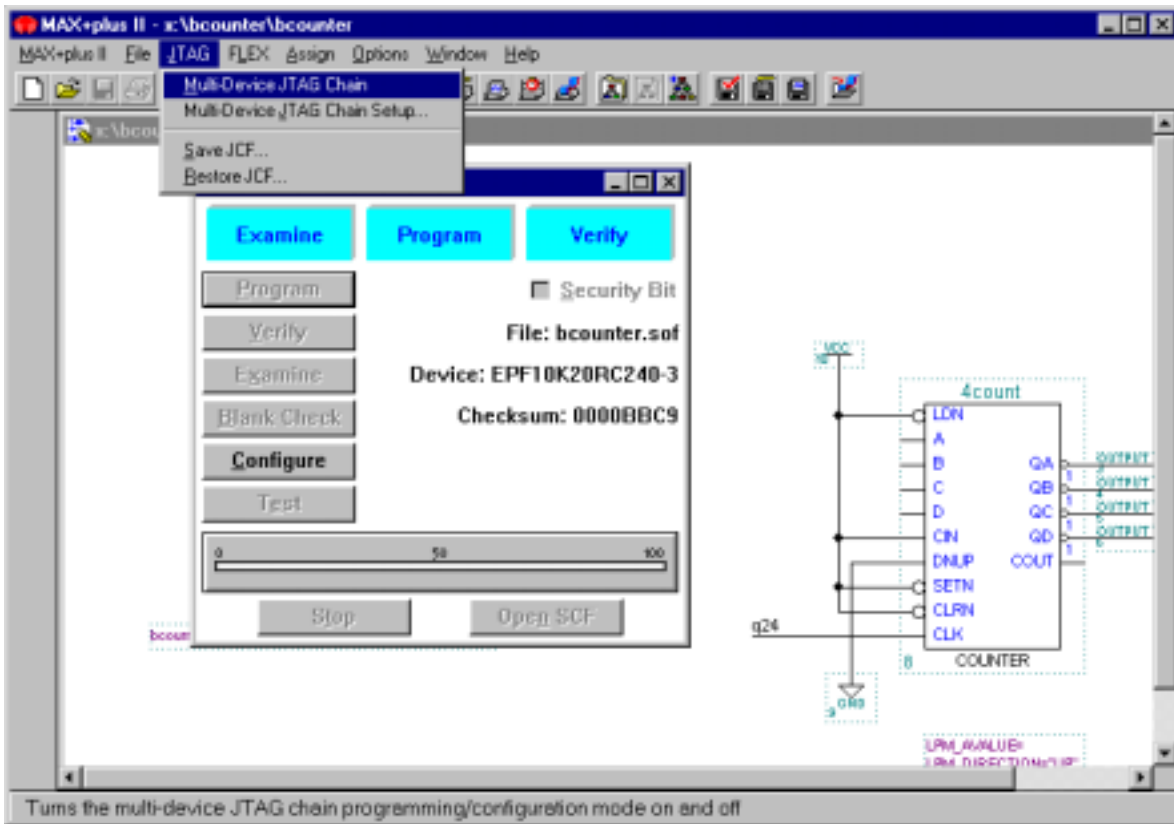
**Figure 2.26: Device Programming**

This will launch the **Programmer Window** as shown in Figure 2.27. To configure the port, double click on the **Multi-Device JTAG Chain** text, as in Figure 2.27.



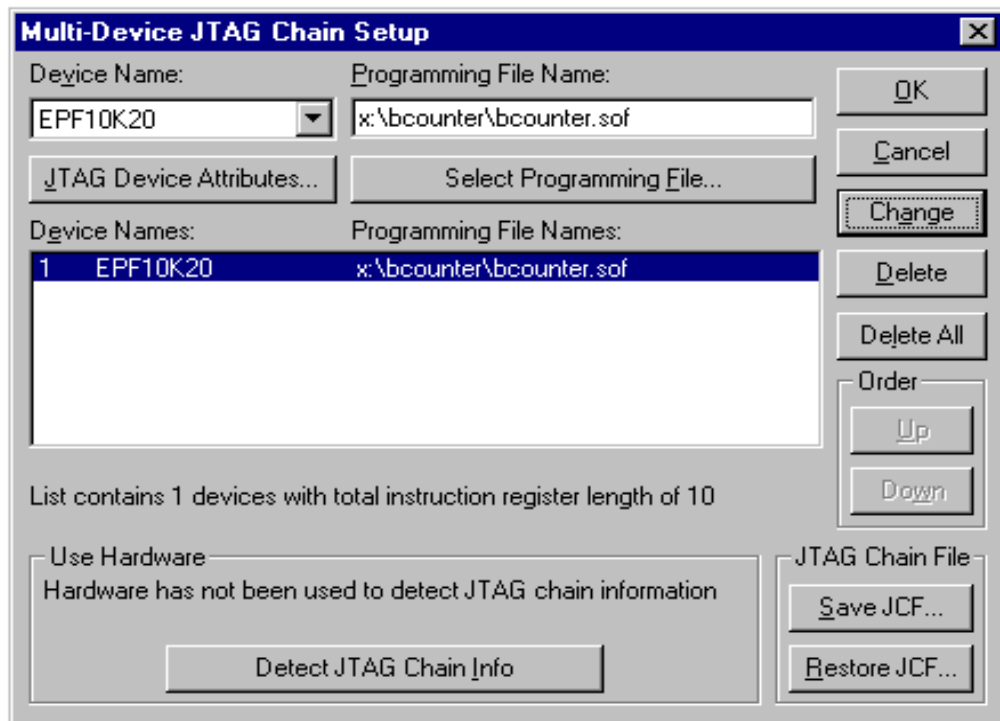
**Figure 2.27: Programmer Window**

[Note: if the **Multi-Device JTAG Chain** text is not present in this window (Figure 2.27) then the default device configuration options have probably been changed by another user. To correct this problem one should select the **Multi-Device JTAG Chain** option from the **JTAG** pull down window as shown in Figure 2.28. This should cause the **Programmer Window** to appear as shown in Figure 2.27 allowing the user to proceed in the manner to be described.]



**Figure 2.28: Enabling JTAG Configuration of External Devices**

After the **Multi-Device JTAG Chain** text has been selected on the **Program Window**, the the **Device Setup** window will appear as shown in Figure 2.29. If the file highlighted under **Programming File Names** area of the window is the correct file for download, one should click on the **OK** button. (Note: the items listed under the **Programming File Names** area default to the ones chosen by the previous user on the particular PC. If the correct item is not listed, then one must single click on the item under **Device Names: Programming File Names:** cross reference window, then click on the **Select Programming File** button, go to the appropriate project folder, select the *sof* file and then single click on the **Change** button. When this is done, the **OK** button must be pressed to continue.



**Figure 2.29: Device Setup**

To begin the download process, go back to the **Programmer** window as shown in Figure 2.27 and click on the **Configure** button. After a few seconds, the design should be configured. In the case of the four bit counter, one would now expect to see the four lower order UP 1 LEDs repetitively counting through the binary sequence.



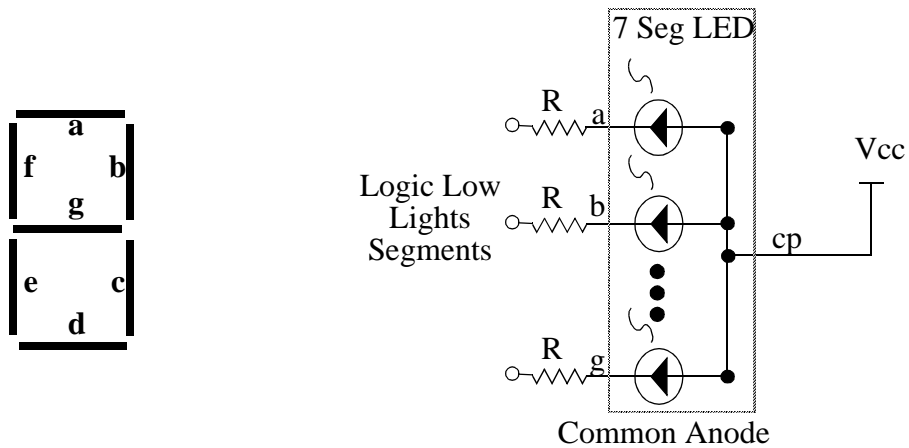
## Chapter 3: Hardware Description Language Design Example

### Example

In this chapter, a binary to seven segment display converter example will be used to illustrate how a design can be entered using VHDL [3,4], synthesized, simulated, and implemented on the UP 1 Educational Board. The purpose of this design will be to display the correct hexadecimal symbol on a seven-segment LED that corresponds to the four bit binary value that is placed on the input pins through a set of dip switches.

### Background

A single seven-segment indicator can be used to display the digits '0' through '9' and the hexadecimal symbols 'A' through 'F' (with symbols 'b', and 'd' being displayed in lower case) by lighting up the appropriate set of segments. For example, the number '8' can be displayed by illuminating all seven segments and the lower case letter 'b' can be displayed by illuminating the segments labeled c,d,e,f, and g for the seven segment display element that is shown in Figure 3.1.:



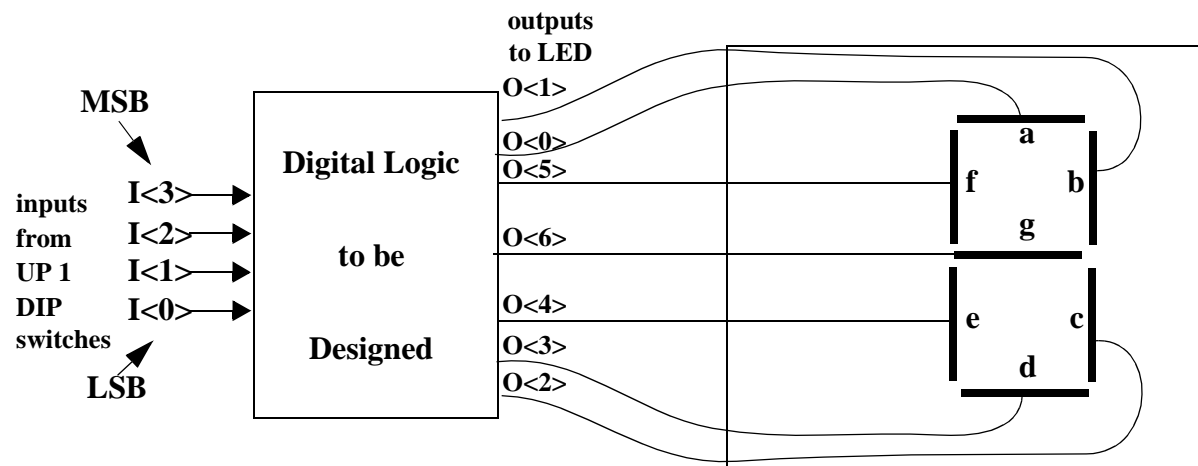
**Figure 3.1: Seven Segment Display Unit**

One common type of seven segment display unit utilizes Light Emitting Diodes, LEDs, as the display elements. In this arrangement, each segment that makes up the seven segment display unit is a separate light emitting diode (LED) that will light up when it is forward biased. Often commercially available seven-segment LED display units minimize the number of external pins needed by internally connecting together one node of each of the seven individual LEDs. In one arrangement, the *common anode*, the anodes of the diodes have a common connection point. If

this common point is connected to  $V_{cc}$  and a set of current limiting resistors are connected in series with the individual segments (or if a single current limiting resistor is connected between  $V_{cc}$  and the common point) then each segment of the display can be independently illuminated by placing a logic low on the corresponding segment lead (assuming the logic device is capable of sinking enough current).

The binary to seven segment display example assumes that an external common anode seven segment LED will be used as the targeted display element with the common anode point being connected to  $V_{cc}$ . Thus a logic low will be required to light up each segment. In this design, the left-most common anode LED that is associated with the Altera Flex 10K20 FPGA on the UP 1 board will be used as the display element (see Figure 1.1).

Figure 3.2 shows a block diagram of the display converter circuit that is to be designed. The display converter circuit is to contain the logic necessary to drive the seven segment display in a manner in which the hexadecimal symbol associated with the four bit input is displayed. Thus the symbol 0 would be displayed if all of the input bits were logic low, and the symbol 8 would be displayed if bit  $I<3>$  was high and the rest low. Table 3.1 shows the desired display configuration for each of the 16 possible input scenarios. In this design, the inputs come from the set of DIP switches on the UP 1 board that are connected to the Altera Flex 10K20 FPGA (see Figure 1.1 for the placement of the DIP switches on the UP 1 -- the set of DIP switches are labeled FLEX\_SWITCH).



**Figure 3.2: Display Converter Seven Segment Display System Overview**

**Table 3.1: Desired LED Display Configurations**

Inputs				Display Configuration	Inputs				Display Configuration
I3	I2	I1	I0		I3	I2	I1	I0	
0	0	0	0	0	1	0	0	0	8
0	0	0	1	1	1	0	0	1	9
0	0	1	0	2	1	0	1	0	A
0	0	1	1	3	1	0	1	1	b
0	1	0	0	4	1	1	0	0	C
0	1	0	1	5	1	1	0	1	d
0	1	1	0	6	1	1	1	0	E
0	1	1	1	7	1	1	1	1	F

### Design Entry

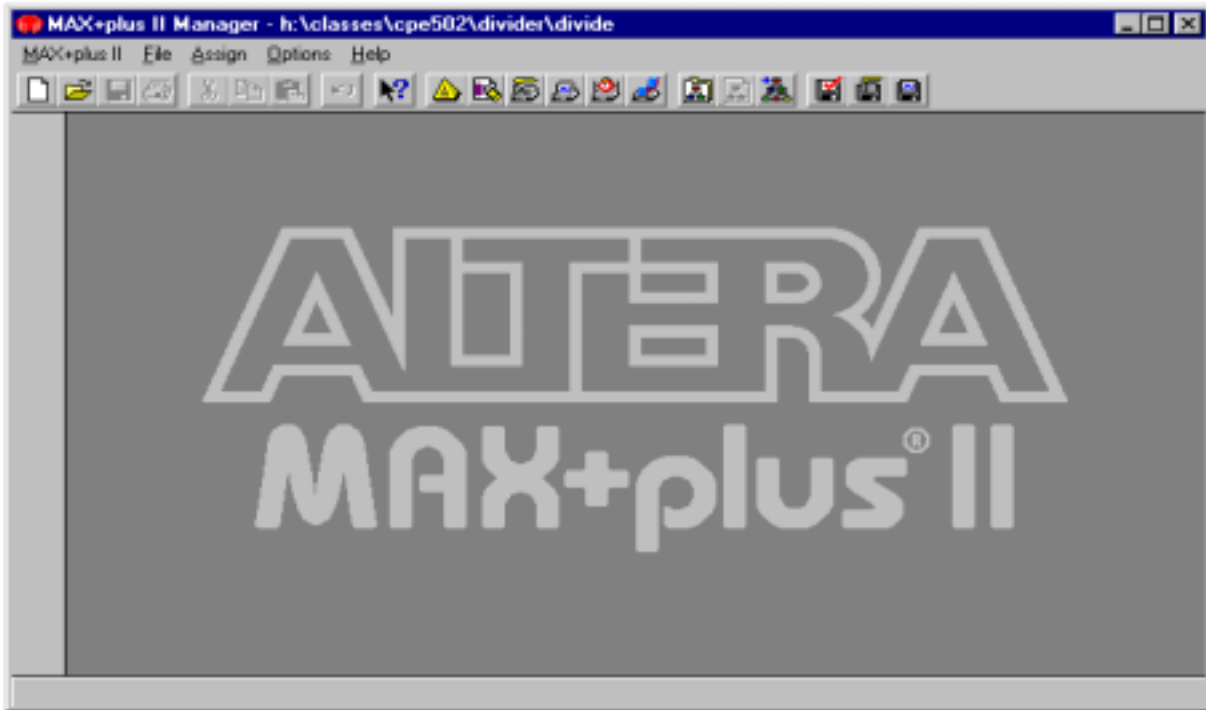
The focus of this chapter is to present a simple example of how a Hardware Description Language, HDL, can be used to implement a simple digital design. HDLs are textual representations that are used to model the structure and/or behavior of the system hardware. They are analogous in some ways to high-level software languages such as C, FORTRAN or C++ with the important distinction that HDLs have special constructs specifically designed to model the characteristics of digital hardware. The major difference in HDLs and high-level software languages are that HDL's can easily model the timing attributes and the highly concurrent aspects of digital hardware (i.e. in physical hardware many events often happen at the same time). In addition HDL's have the power to fully describe a logic system using both behavioral and structural design techniques.

We will now illustrate how a HDL can be used to implement the binary to hexadecimal converter design discussed previously. To do this, first invoke the **MAX+plus II Manager** win-

down in the same manner as was done previously, by double clicking **MAX+plus II** icon,

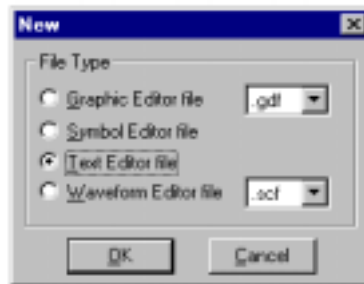


from the Window's Desktop. The main **MAX+plus II** window will appear as shown in Figure 3.3.



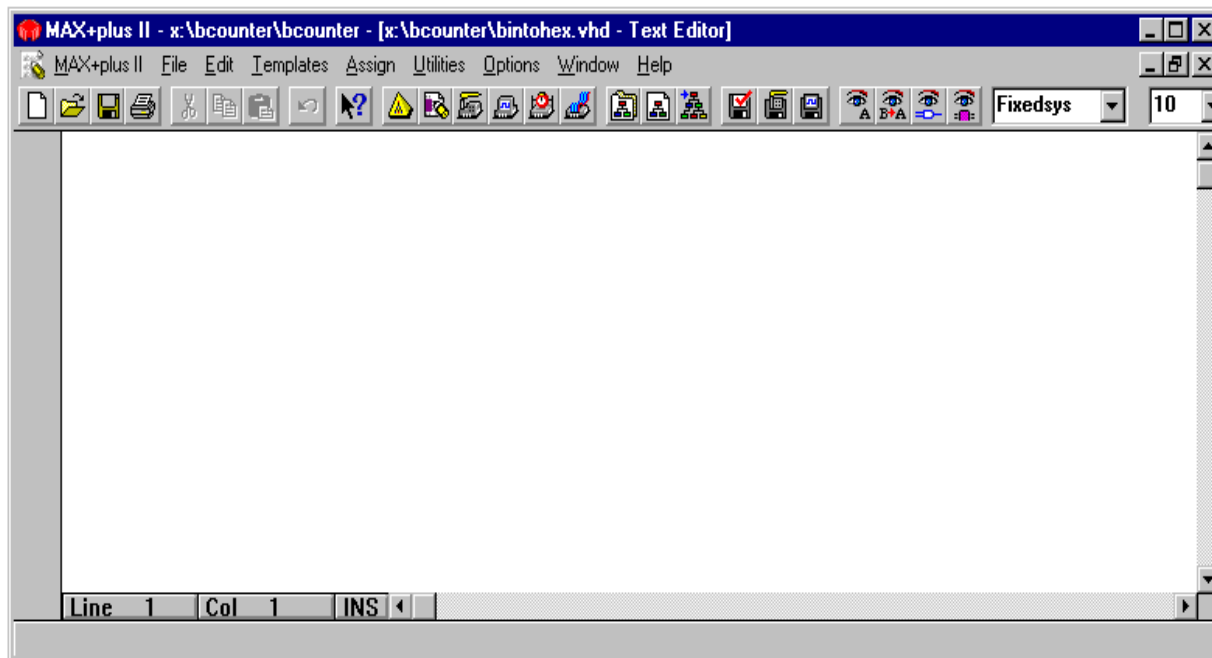
**Figure 3.3: Altera MAX+plus II Manager.**

Then select the **New** option from under the **File** pull down menu. The window shown in Figure 3.4 should then appear. Choose the **Text Editor File** option since we are implementing this design in a textual manner using an HDL. Then click on the **OK** button.



**Figure 3.4: Altera MAX+plus II Input Selection.**

This will launch the **Text Editor** window which is shown in Figure 3.5. After the file is saved, the current path will appear on the title bar that is on the top of the window. [Warning it should be noted that the default path that is displayed when one enters the text editor in this way is always the path of the previous user on the same system!]



**Figure 3.5: Text Editor Window**

Since all three of the designs in this manual are related to one another, we will keep all design files under the same working directory. For this binary to hexadecimal converter example we will want to give the file the filename bintoeh.vhd and save it under this current working directory, i.e. **X:\bcounter**. To do this, select the **Save** option under the **File** pull-down menu. This will bring up the **SaveAs** dialog window. The filename bintoeh should then be entered into the **File Name** field of this window and the **Automatic Extension** field should be set to .vhd.<sup>1</sup>

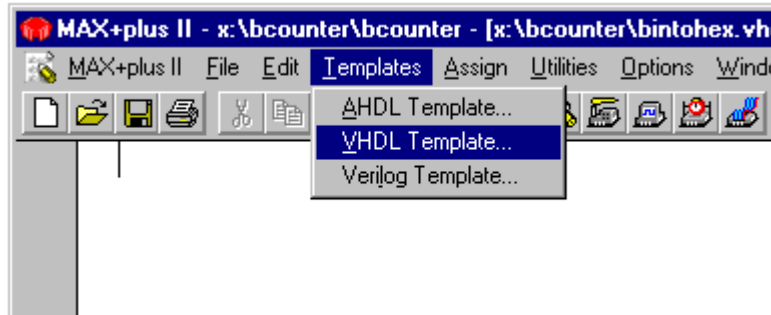
The next step is to set up a *project* file to represent the current design. To do this from the main **MAX+plus II** window, select the **Project** option from the **File** pull-down menu. Then go to and select the **Set Project to Current File** option or simply press the **Set Project to Current File** shortcut button (shown in Figure 2.4 of Chapter 2).

---

1. Note: Very High Speed Integrated Circuit Hardware Description Language (VHDL) is the hardware description language that we will be using to represent the hexadecimal converter design. Files written in VHDL should have the **vhd** extension to be correctly identified by the Altera CAD tool.

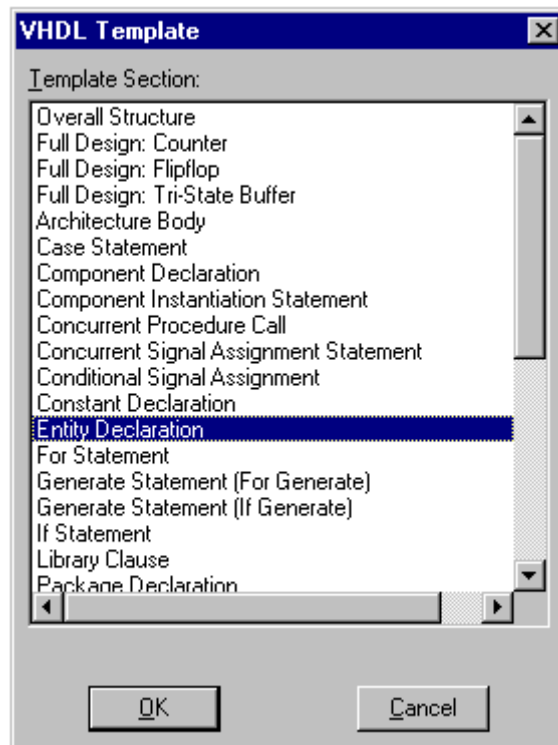
The VHDL file that represents the binary to hexadecimal converter design can now be directly entered using the text editor or the built-in VHDL template facility can be used to aid the user with setting up the VHDL model.

The VHDL template facility provides the necessary keywords usage with all the right structure. The VHDL template is activated by first placing the cursor at the desired point in the text file, then selecting the **VHDL Template** option which can be found under the **Templates** pull-down menu as shown in Figure 3.6.



**Figure 3.6: Accessing a VHDL Template.**

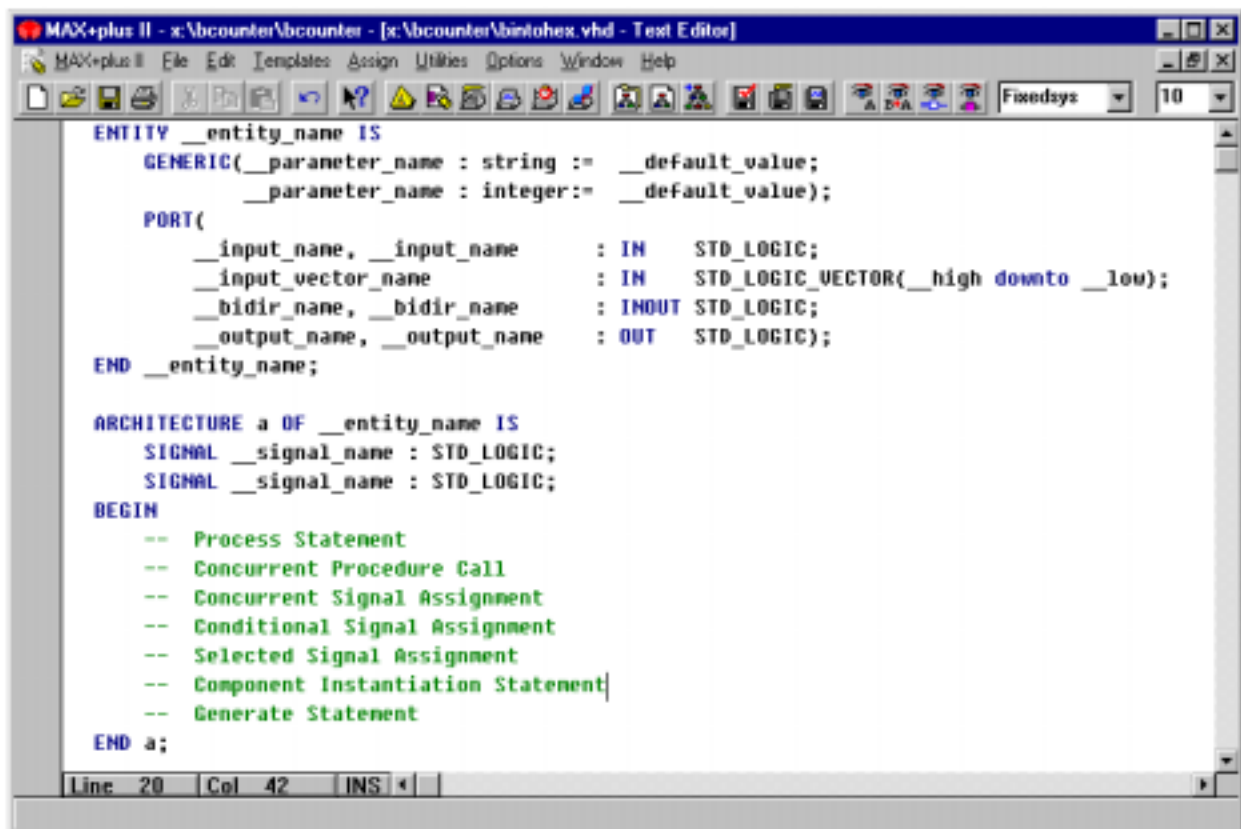
This launch the **VHDL Template Window** as shown in Figure 3.7.



**Figure 3.7: VHDL Template Selection Window**

The user then selects the desired language construct and the major components of the particular construct are automatically placed in the file at the point where the original cursor was when the template facility was selected.

For example, all VHDL files contain at least two main sections, the *Entity* and the *Architecture*. The *Entity* section describes how the design that is being modeled is to interface with the outside world and *Architecture* section describes how the design is to function (i.e. details of the implementation). The syntax for these two sections can be easily obtained by first placing the cursor the cursor at the top of the file, invoking the VHDL template feature from the Templates pull down menu and selecting the *Entity Declaration* option from the VHDL Template window. The *Architecture Body* option can be selected in a similar. The resulting two templates can act as a starting point for further VHDL model development. Figure 3.8 shows the basic skeleton model that was automatically generated in this manner for the binary to hexadecimal converter example.



```
MAX+plus II - x:\bcounter\bcounter - [x:\bcounter\bin2ohex.vhd - Text Editor]
MAX+plus II File Edit Templates Assign Utilities Options Window Help
Fixedsys 10
ENTITY __entity_name IS
    GENERIC(__parameter_name : string := __default_value;
            __parameter_name : integer:= __default_value);
    PORT(
        __input_name, __input_name      : IN   STD_LOGIC;
        __input_vector_name             : IN   STD_LOGIC_VECTOR(__high downto __low);
        __bidir_name, __bidir_name      : INOUT STD_LOGIC;
        __output_name, __output_name    : OUT  STD_LOGIC);
END __entity_name;

ARCHITECTURE a OF __entity_name IS
    SIGNAL __signal_name : STD_LOGIC;
    SIGNAL __signal_name : STD_LOGIC;
BEGIN
    -- Process Statement
    -- Concurrent Procedure Call
    -- Concurrent Signal Assignment
    -- Conditional Signal Assignment
    -- Selected Signal Assignment
    -- Component Instantiation Statement
    -- Generate Statement
END a;
Line 20 Col 42 INS
```

Figure 3.8: Skeleton VHDL Model for the Binary to Hexadecimal Converter Example

Figure 3.9 shows the final VHDL file after the basic template was modified.

```

-- VHDL Template File for binary to hex converter example
-- File: bintoehex.vhd
-- B. Earl Wells, Sin Ming Loo, August 2000, ECE, University of Alabama in Huntsville
library IEEE;
use IEEE.std_logic_1164.all;

entity bintoehex is
    port (I: in STD_LOGIC_VECTOR (3 downto 0);
          O: out STD_LOGIC_VECTOR (6 downto 0) );
end bintoehex;

architecture bintoehex_arch of bintoehex is
begin
-- with/select construct use to create a simple 7 output/4 input
-- truth table (with inputs on the right side as shown below).
with I select
--      Outputs 00<6> 0<5> 0<4> 0<3> 0<2> 0<1> 0<0>      Inputs  I<3> I<2> I<1> I<0>
--
--          gfedcba
    0 <=
        "1000000"      when "0000",
        "1111001"      when "0001",
        "0100100"      when "0010",
        "0110000"      when "0011",
        "0011001"      when "0100",
        "0010010"      when "0101",
        "0000010"      when "0110",
        "1111000"      when "0111",
        "0000000"      when "1000",
        "0011000"      when "1001",
        "0001000"      when "1010",
        "0000011"      when "1011",
        "1000110"      when "1100",
        "0100001"      when "1101",
        "0000110"      when "1110",
        "0001110"      when "1111",
        "-----"      when others;

end bintoehex_arch;

```

Figure 3.9: Final VHDL Model for bintoehex Example

The structure of the VHDL file can be described as follows. Double dashes, "--", are used to introduce comments and semicolons are used to terminate the statements. The file begins with the VHDL keyword *entity* followed by the user-defined name of the logic design (in this case *bintoehex*). In **MAX+plus II** software, this entity name (i.e. the entity name at the top of the hierar-



chy) must be the same as the name of the project. The entity name is then followed by the keywords **is port** that designates that a list of input/output ports is to follow. In this case, we need to define two bus signals. One is named *I* (for input) which is a four member input vector (direction specified by the **in** keyword),  $I(3) \text{ -- } I(0)$ , with  $I(3)$  acting as the most significant bit (this is due to the **downto** keyword). The other is a seven member output vector (direction specified by the **out** keyword) which is named *O*, which has seven members  $O(6) \text{ -- } O(0)$ . [Note: The `STD_LOGIC_VECTOR` defines the vector type. It is declared in the library `IEEE.std_logic_1164.all`. It is possible for the user to declare arbitrary data types in VHDL. The data types defined in this library are standardized allowing for increased portability among VHDL simulators and synthesizers.] The entity section ends with the **end** statement followed by the design name.

The desired behavior of the entity section is modeled in the architectural body section. This section begins with the keyword **architecture** which is followed by a user defined name for the architecture (in this case the *bintohex\_arch* is used) which is paired with the identity of the entity using the **of** keyword followed by the entity name (i.e. in this case *of bintohex*). The **begin** keyword is used to separate the *architecture declarative* part of the model from the *architecture statement* part. The architecture declarative part is used to make declarations for such items type, signals, and components. For the model presented in Figure 3.8, no declarations are needed to represent the design. The architecture statement part of the model is placed between the **begin** keyword and the **end** keyword. This is where the VHDL modeling statements are to be placed. VHDL is very rich in constructs that allow one to model digital logic using a wide range of styles.

A single **with... select...when** statement was added to the *architecture* section. (The template for this construct can also automatically be entered by choosing the **Selected Signal Assignment Statement** option from the **VHDL Templates** window in the manner previously described.) The **with... select...when** statement is analogous in many ways to a **case** statement in

the C programming language in that provides selective signal assignments. It has the following structure:

```
with input_signal select  
output_signal <= value_a when value_1,  
                value_b when value_2,  
                .  
                .  
                .  
                value_x when last value,  
                value_z when others;
```

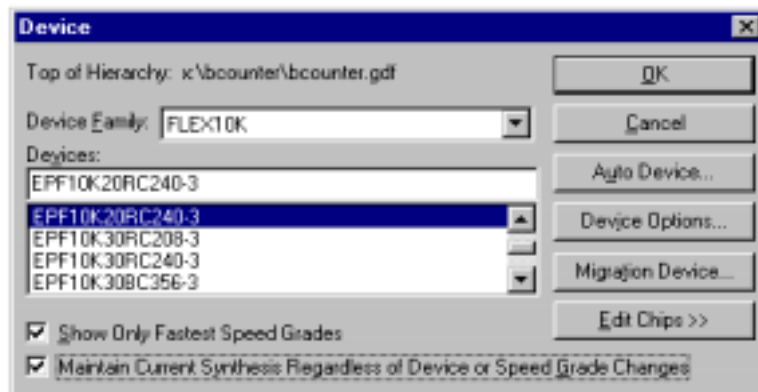
Here when the input signal equals *value\_1* then the *output\_signal* will be set equal to *value\_a*. When the input\_signal equals *value\_2* the *output\_signal* will be set to *value\_b* and so on. In this example, this construct is being used to implement the truth table that describes the desired binary to hexadecimal converter operation (but unlike most truth tables the inputs appear on the left side). The input\_signal and output\_signal are both vectors that are defined within the library package IEEE.std\_logic\_1164.all. In VHDL, the logic values that are support include '0' for logic low (forcing 0), '1' for logic high (forcing 1), and '-' for don't care.

After the VHDL model for the binary to hexadecimal converter has been entered it should be saved by selecting the **Save** option from the **Files** pull-down menu.

### **Device Assignment**

Since this is a new design with a new project name, the targeted device needs to be re-assigned so that the Altera MAX+plus II CAD tool will know the type of programmable logic device that will be used to implement the design. This is done in the same manner as previously described for the binary counter example in Chapter 2. The targeted device is again the Altera Flex 10K20RC240 which is present on the UP 1 Educational Board. To assign this device, from the **Graphic Editor**, window choose the **Assign** option which is under the **Device** pull down menu. This will launch the **Device Assignment** window which is shown in Figure 3.10. Then

select the FLEX10K option from the **Device Family:** portion of this window after which select the EPF10K20RC240-3 device from the **Devices:** section of the window and press the **OK** button.



**Figure 3.10: Device Selection.**

### Constraint Entry

In this binary to hexadecimal design example, it will be necessary to assign (lock) the four logical inputs of the binary to hexadecimal design to the set of DIP switches located on the UP 1 Educational Board and assign the seven outputs to drive the individual segments of one of the UP 1 common anode seven segment LED displays (specifically the left-most one which is connected to the 10K20 device as shown in Figure 1.1). To accomplish this mapping the desired VHDL signal name to MAX+plus II pin number cross reference (i.e. Pin Locks) will be as shown in Table 3.2. (A full listing of pin numbers for the Flex 10K20 device and the UP 1 can be found in [1])

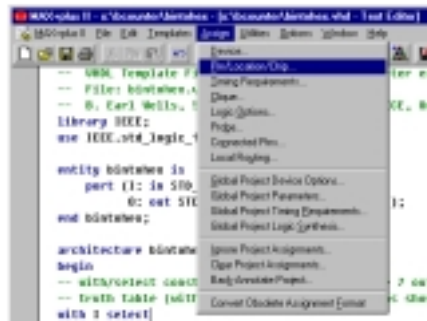
**Table 3.2: Desired Pin Locking (Cross Reference) Configuration**

I/O Pin Description	HDL Signal Name	Flex 10K20 Pin Number
FLEX_Switch #1 (low - ON, high - OPEN)	I0	41
FLEX_Switch #2 (low - ON, high - OPEN)	I1	40
FLEX_Switch #3 (low - ON, high - OPEN)	I2	39
FLEX_Switch #4 (low - ON, high - OPEN)	I3	38
Flex 10K20 Pin 6, (segment 'a' of LED)	O0	6
Flex 10K20 Pin 7, (segment 'b' of LED)	O1	7
Flex 10K20 Pin 8, (segment 'c' of LED)	O2	8

**Table 3.2: Desired Pin Locking (Cross Reference) Configuration**

I/O Pin Description	HDL Signal Name	Flex 10K20 Pin Number
Flex 10K20 Pin 9, (segment 'a' of LED)	O3	9
Flex 10K20 Pin 11, (segment 'e' of LED)	O4	11
Flex 10K20 Pin 12, (segment 'f' of LED)	O5	12
Flex 10K20 Pin 13, (segment 'g' of LED)	O6	13

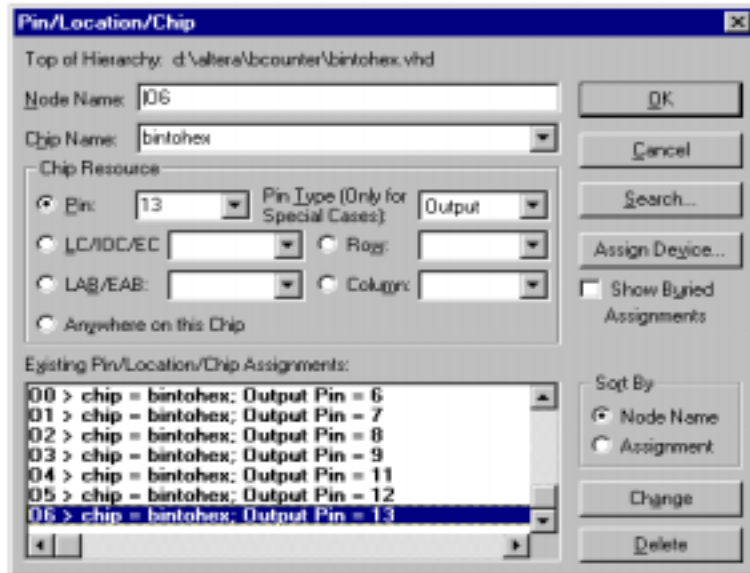
To perform this VHDL signal name to 10K20 pin number assignment, first select the **Pin/Location/Chip** option from under the **Assign** pull down menu as shown in Figure 3.11.



**Figure 3.11: Assigning Pins**

This launches the **Pin/Location/Chip** window which is shown in Figure 3.12. The parameters that are needed to lock the pins are the same as they were in the schematic capture case. The user should first enter the desired signal name that is declared in the entity section of the VHDL file into the **Node Name:** field. Then enter the desired pin number in the **Pin:** field, set the **Pin Type:** attribute, and click on the **Add** button. (After this, the **Add** button will turn into a **Change** button to allow for further editing of the existing parameter.) Then repeat this process for each input/output signal name that should be locked to a specific Altera pin number. Figure 3.12 shows

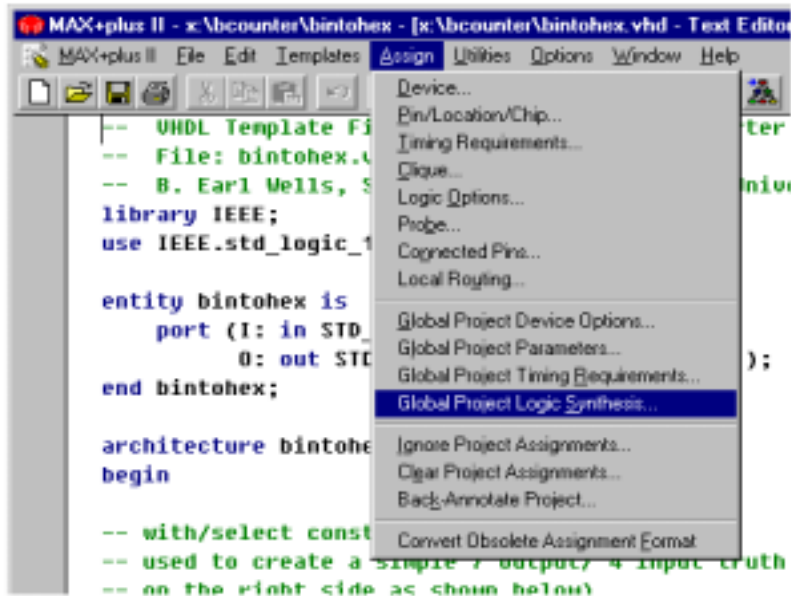
the situation where the seven output pins have been so locked. When all the desired pins are locked, one should exit the **Pin/Location/Chip** window by pressing the **OK** button.



**Figure 3.12: Pin/Location/Chip.**

## Project Processing

Before we compile/synthesize the design into a format that can configure the selected device it is a good idea to check the current status of the various synthesis parameters. To do this select the **Global Project Logic Synthesis** option from under the **Assign** pull-down menu as shown in Figure 3.13.



**Figure 3.13: Global Project Logic synthesis.**

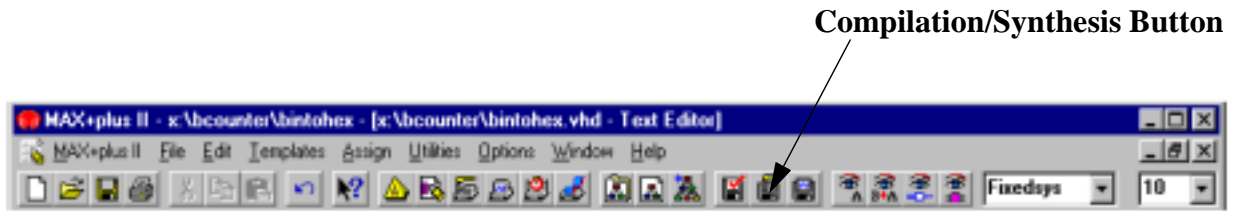
This will launch the **Global Project Logic Synthesis** window as shown in Figure 3.14.



**Figure 3.14: Global Project Logic Synthesis Window**

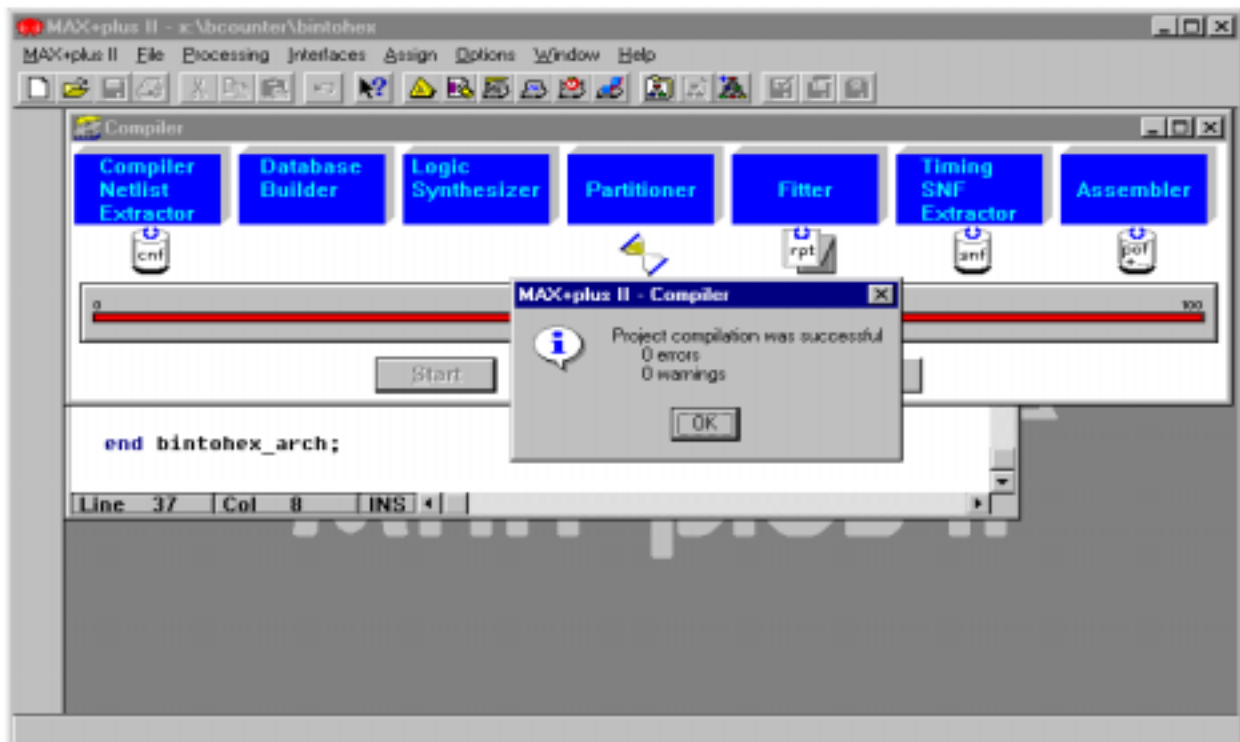
To prepare the Altera software for logic synthesis of the VHDL model to the 10K20 device on the UP 1 make sure that all parameters are set as shown in the figure.

After this is done the synthesis process can be activated as shown in Figure 3.15 by pressing the **Compilation/Synthesis Button** on the tool bar or by selecting the **Compiler** option from the **MAX+plus II** pull-down window from within the graphics editor.



**Figure 3.15: Compilation/Synthesis.**

As in the previous chapter, if the **Compilation Button** is used, the compilation process will start automatically as shown in Figure 3.16. If the compiler/synthesizer is invoked using the pull-down window, then one must click on the **Start** button on the **Compiler** window as shown in the figure.



**Figure 3.16: Compilation/Synthesis process.**

If there are syntax errors in the VHDL source code, then there will be a red error message in the text area of the window. Double clicking on the error message text will automatically bring up the text editor at the point in the file where the offending line of the source code was placed.

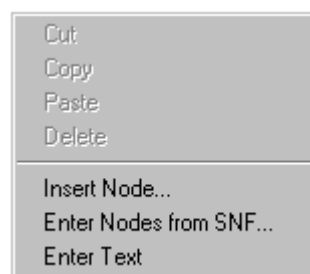
When the project processing/compilation process has been completed successfully, the design is ready for verification.

## Project Verification (Functional Simulation)

The next step is to perform a **Functional Simulation** to check the logical correctness of a design before it is actually implemented on the targeted system (which will be in this case the Altera UP 1 board). The following is a brief overview of how simulation can be used to verify the functionality of the binary to hexadecimal converter example. A much more detailed discussion of this material is presented in the Altera literature.

### Selecting Observation Points

Before a functional simulation can be run, the user needs to select a set of logical signals from the VHDL file which should be observed to verify the correct operation of the design. These nodes often include the *port* input/outputs of the design described in the highest level *entity* section of the VHDL file (which are usually assigned to specific device pin numbers) as well as certain internal signals. To select the observation points from the Altera environment simply choose the **WaveForm Editor** option from the **MAX+plus II** pull down menu. This will launch the **Waveform Editor** window. Then point the cursor to the white space under the time bar and press the right mouse button. This should bring up the window shown in Figure 3.17.

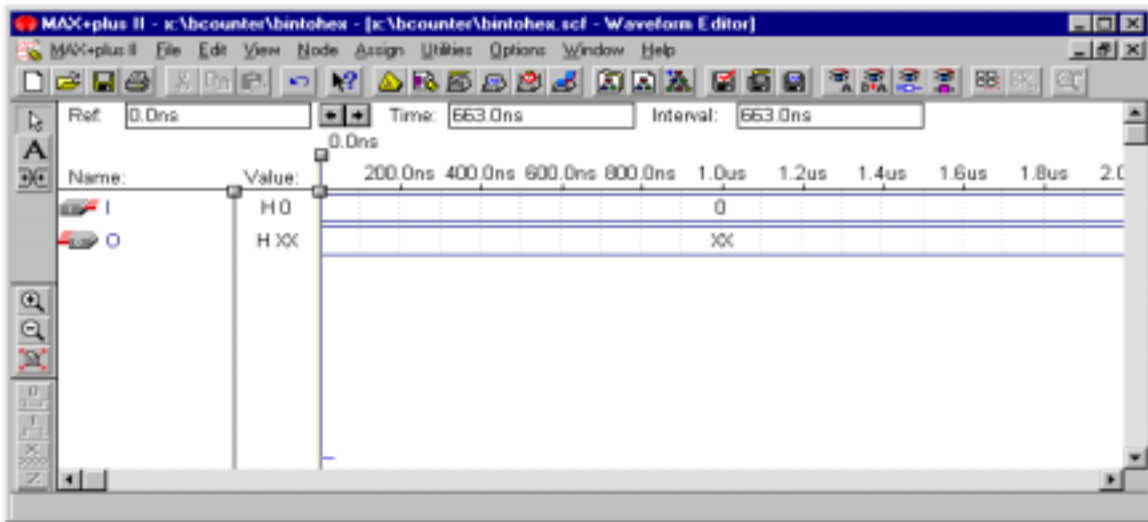


**Figure 3.17: Select Enter Nodes from SNF.**

From this window, select the **Enter Nodes from SNF ...** option which will launch the **Node Selection** window. This window allows the user to choose the set of signals to be observed during the simulation. For the binary to hexadecimal converter example, we will choose for demonstration purposes to observe the input signals **I(0) -- I(3)** and the output signals **O(0) -- O(6)** as defined in the entity section of the VHDL file. To select these signals for observation, select the **List** option under the **Type** area of the **Node Selection** window. These I/O signals will then



appear under the **Available Nodes & Groups** section of the **Node Selection** window. For the binary to hexadecimal converter example, we would like to observe all of these signals. This is done by highlighting all of the signals that are under the **Available Nodes & Groups** section of the window and then clicking on the => button. These signals will then appear **Selected Nodes & Groups** section of the window. When this is done simply press the **OK** button to continue. The signals should appear under the Name: area of the Waveform Editor window as shown in Figure 3.18).



**Figure 3.18: Waveform Editor (binary to hexadecimal converter example)**

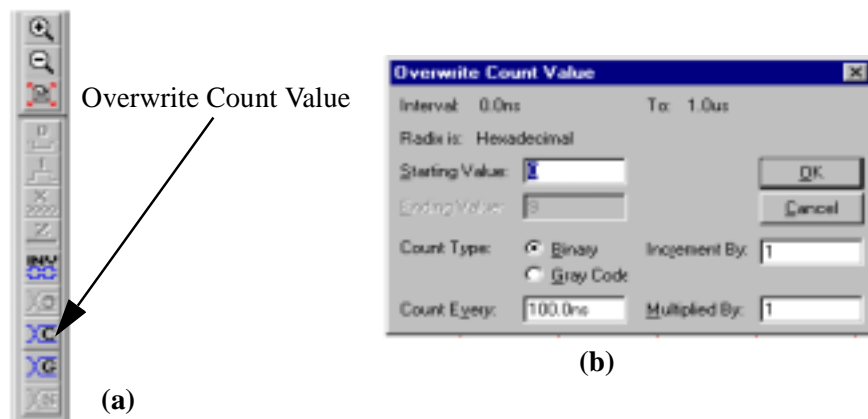
### Stimulus Entry

The next step is to enter to stimulus information. To enter new input stimulus after selecting the observations points as described in the previous section, one should make the **Waveform Editor** window the active window. This window should then show all the signals which were previously selected as observation points (see Figure 3.18 for the binary hexadecimal converter case). The Altera CAD tool requires that we enter the simulation end time before the stimulus is entered. To do this from the **Waveform Editor**, select the **End Time** option from the **File** pull-down menu. This should launch the **End Time** window. Then for the binary to hexadecimal converter example, enter 2.0us in the dialog box and click on the **OK** button.

Set **End Time** to 2 us. By selecting the **I** signal, click on the **Overwrite Count Value** button as shown in Figure 3.18(a). Figure 3.18(b) shows the parameters needed, now click on the **OK** button.

At this point the waveform information must be saved (without saving the waveform file the **MAX+plus II** CAD tool will not allow stimulus information to be entered). To do this, select the **Save As** option from the **File** pull down menu which is on the **MAX+plus II** tool bar. Save the files as a *.scf* file. The file should have the same name as the *vhd* file but the extension will be *.scf*.

In the case of the binary to hexadecimal converter example, we would like to apply all possible input values to the input lines **I0 -- I3** to exhaustively test the design. The easiest way to do this is to have the simulator use a ‘virtual’ counter to drive these inputs. To do this, select the **I** set of signals from within the **Waveform Editor**, then click on the **Overwrite Count Value** shortcut button which is on the vertical tool bar as shown in Figure 3.19(a). This will launch the **Overwrite Count Value** window which is shown in Figure 3.19(b). This window will allow one to enter the starting logic value of the count, the type of count performed and the period of the time between successive count values. For the binary to hexadecimal converter, we have chosen to start the count at 0, perform a standard binary count sequence and count every 100nS as shown in Figure 3.18. When all the data has been entered as shown in Figure 3.18 the window was exited by pressing the **OK** button.

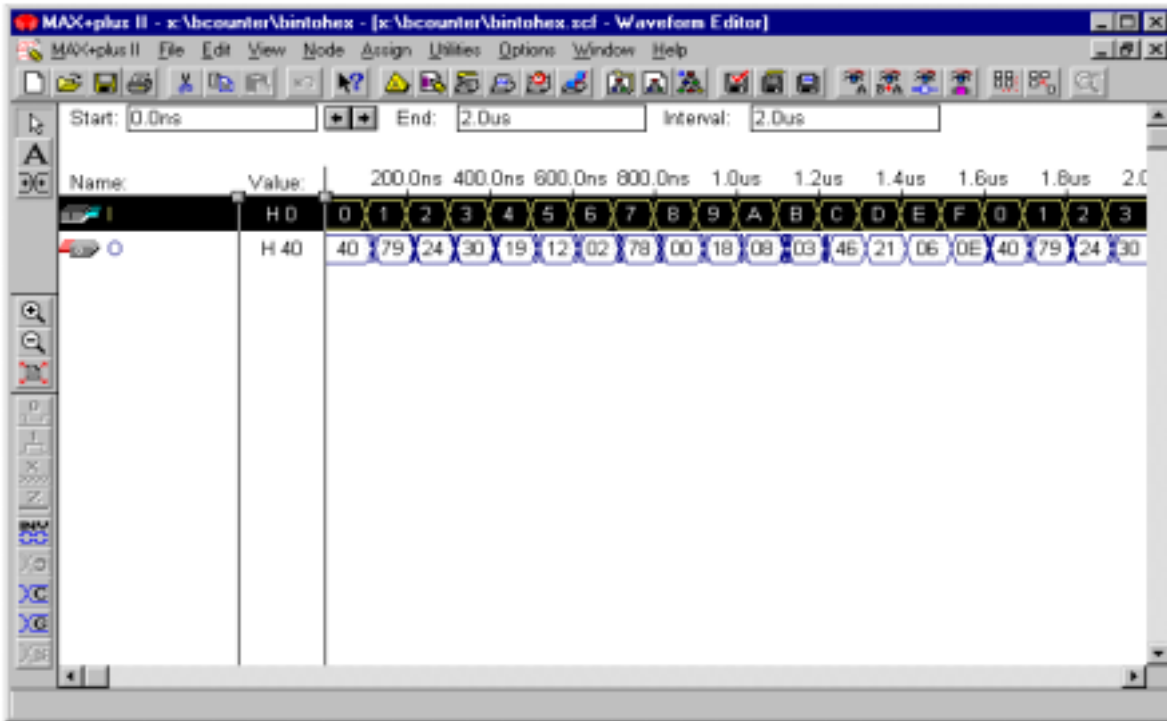


**Figure 3.19: bintohex's Waveform Editor.**

### Functional Simulation

To start the functional simulation simply click on the **Simulation Button** which is located on the horizontal tool bar of the main **MAX+plus II** window. A **Simulator** status window will appear. To run the simulation simply click on the **Start** button of this window. Upon successful

completion of the simulation process a status message should appear. Press the **OK** button to continue. Figure 3.19 shows the results of this simulation.



**Figure 3.20: binto hex's simulation results.**

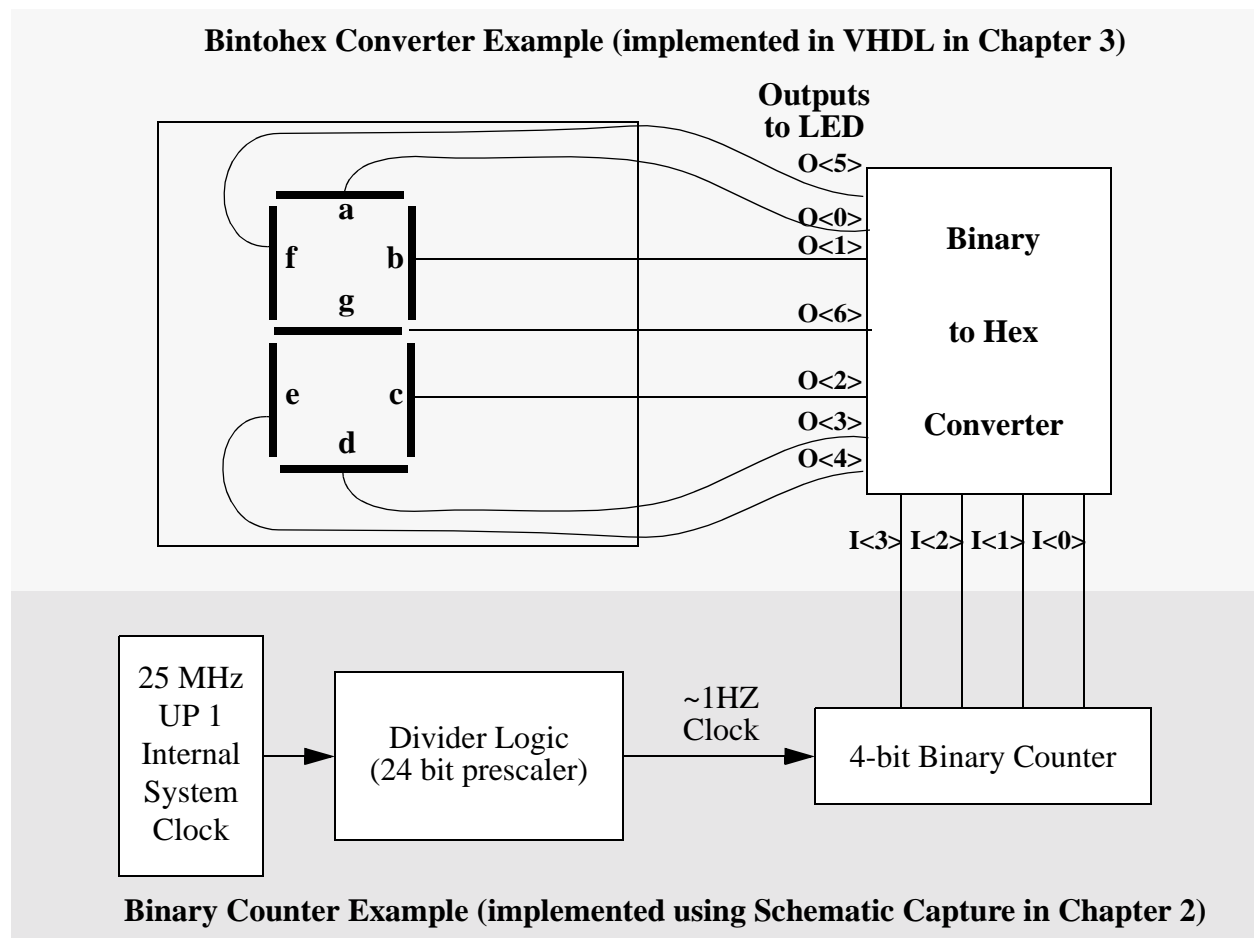
## Device Programming

This phase is identical to that described in Chapter 2. Please refer to this material to configure the Flex 10K20 for the binary to hexadecimal converter design. When the design has been configured the seven-segment LEDs display should display the hexadecimal symbol which represents the current state of the four FLEX\_Switch Dip switches.

## Chapter 4: Combined Schematic Capture/HDL Design Example

### Example

In this chapter, the two previous design examples from Chapter 2 (the binary counter) and Chapter 3 (the binary to seven segment hexadecimal converter) are combined to form a complete hexadecimal counter design. The final design will result in a new hexadecimal symbol being displayed on the built-in seven segment LED after each master clock pulse of ~1HZ. The hexadecimal counter will be constructed by connecting the outputs of the binary counter design directly to the inputs to the binary to seven segment hexadecimal converter as shown in Figure 4.1. As in the binary counter example of Chapter 2, the clocking signal will originate from an external 25.175 MHz oscillator which will be directed through a 24 bit prescaler network. To minimize the design effort and illustrate the mechanics of hybrid design methodology, this design is to be entered using both schematic capture and HDL methodology and constructs.



**Figure 4.1: Hexadecimal Counter Example**

## Hybrid Design Entry Techniques

This section presents a basic methodology which allows for hybrid hierarchical designs to be entered using a combination of schematic capture and VHDL. The base methodology is demonstrated using the hexadecimal counter example which was described in the previous section of this chapter. The material in this chapter assumes that the reader is already familiar with the basic design entry, design processing, design verification, and design implementation techniques discussed in Chapters 2, and 3 of this manual.

The hexadecimal counter example will be created by combining the major components of the binary counter example from Chapter 2, and the binary to hexadecimal converter from Chapter 3. The manner in which this can be done will now be described.

It is assumed that the directory **X:\bcounter** will again be used to store all of the files associated with the design. The first step is to make a copy of the binary counter design. To do this one should **Open** the schematic diagram bcounter.gdf created in Chapter 2. Then **Save** it to a separate filename such as bcounterw7seg.gdf so as not to destroy the original design.

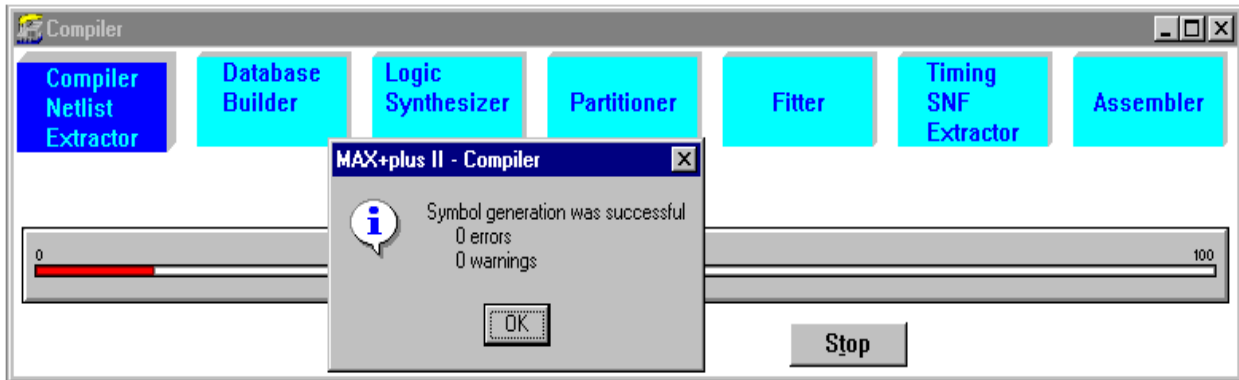
The next step is to set up a *project* file to represent this design. To do this from the main **MAX+plus II** window, select the **Project** option from the **File** pull-down menu. Then go to and select the **Set Project to Current File** option or simply press the **Set Project to Current File** shortcut button (shown in Figure 2.4 of Chapter 2).

This binary counter design needs to be modified before it can be used to drive the binary to hexadecimal converter element in the manner shown in Figure 4.1. This is because the original binary counter example's four global outputs ports, **D0 - D3**, have all been inverted to light up individual LED's which all require a logic 0 is to light them up. The hexadecimal converter design of Chapter 3 was designed to display the hexadecimal equivalent of the true four-bit binary value associated with its input, so if the hexadecimal counter is to count in sequence then either the binary counter or the binary to hexadecimal converter portion of the design needs to be appropriately modified. In this chapter it is assumed that the binary counter part of the design will be changed by removing the bit inversion at the outputs of the design.

In the original binary counter example the inversion on the output pins was accomplished, not by adding external inverters in the diagram but rather by setting certain parameters associated with the **QA:QD** outputs of the *4count* symbol. Removing this inversion, requires that one click on the *4count* symbol, and then click with the right mouse button to select the **Edit Ports/Param-**



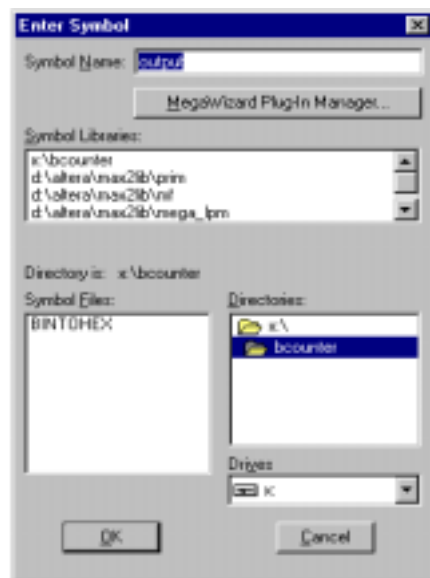
symbol for the binary to hexadecimal converter which allows it to be placed in the schematic in the same manner as any other component. To do this first close the VHDL file and return to the bcounterw7seg.gdf file under the **Graphic Editor**.



**Figure 4.3: Symbol Compilation Window**

The new symbol that has been created will be given the same name as the name of the project file (which is also the same name as the name of the top-level *Entity* section of the VHDL file). It will appear as one of the symbols present in the **Symbol Selection** window which can be accessed from within the **Graphics Editor**.

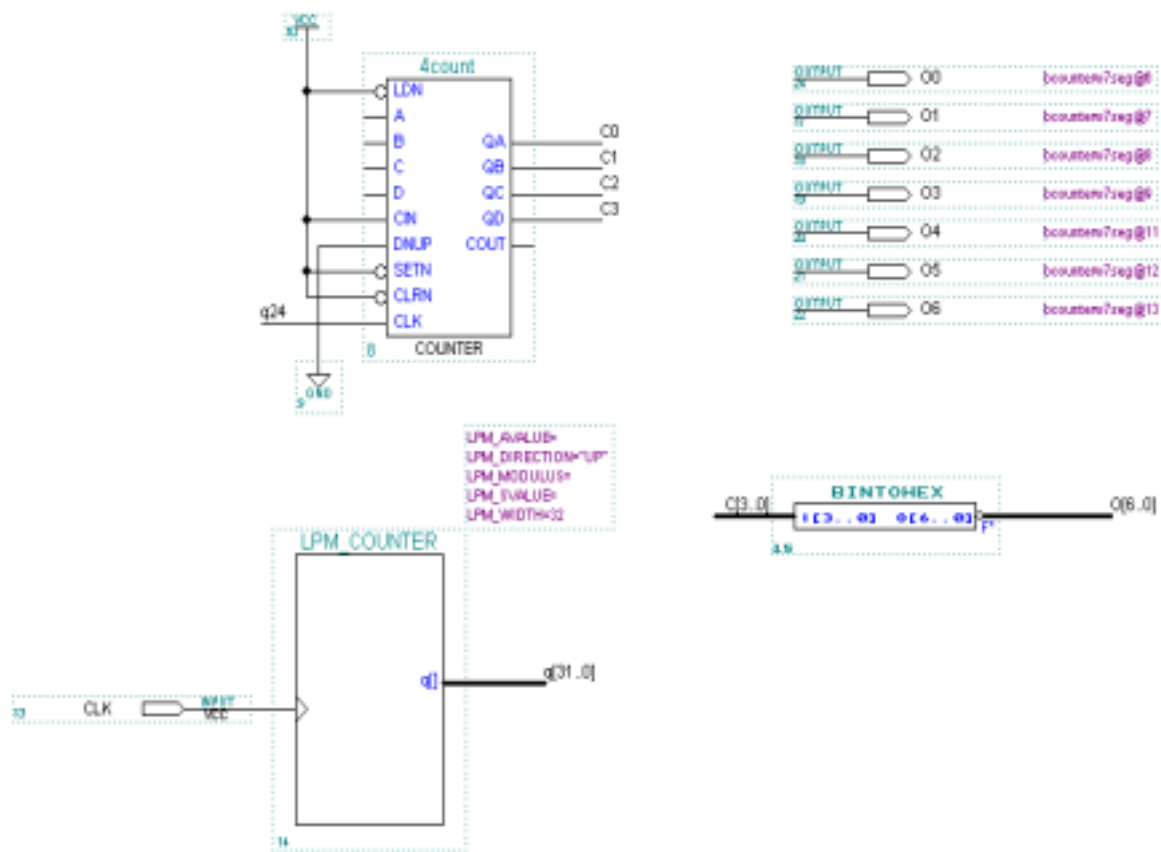
To enter the binary to hexadecimal component symbol into the bcounterw7seg.gdf file first double click on the white space background to bring up the **Symbol Selection** window. The symbol name **BINTOHEX** should appear under **Symbol Files** area of the window as shown in Figure 4.4.



**Figure 4.4: Symbol Selection Window**

To add the binary to hexadecimal converter into the design one only needs to click on **BINTOHEX** from under the **Symbol Files** area of the window and click on the **OK** button. The symbol is then placed in the normal manner.

To complete the hexadecimal counter design then only requires the addition of the seven instances of the **Output** symbols (named **O0** -- **O6**) to provide the global outputs from the design to drive the seven segment LEDs and the proper placement/naming of the wires used to connect the binary counter to the binary to hexadecimal converter and the wires used to connect the hexadecimal converter to the global outputs. The final schematic diagram for this design is shown in Figure 4.5.



**Figure 4.5: Final Hexadecimal Counter Design**

### Constraint Entry

As in the previous cases, all I/O symbols in the schematic must be assigned (locked) to specific Altera device pin numbers to allow the design to function correctly on the UP 1 board.

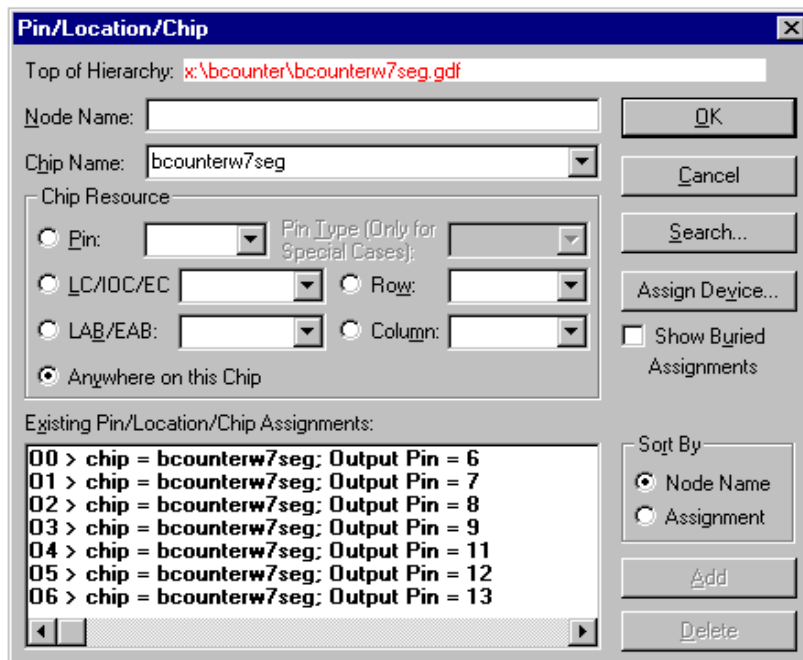


For the hexadecimal design example this means that the global clock **Input** symbol and the seven **Output** symbols must all be locked to specific pin locations as shown in Table 4.1.

**Table 4.1: Desired Pin Locking (Cross Reference) Configuration**

I/O Pin Description	Schematic I/O Pin Name	Flex 10K20 Pin Number
UP 1 25.175 MHZ Clock	CLK	91
Flex 10K20 Pin 6, (segment 'a' of LED)	O0	6
Flex 10K20 Pin 7, (segment 'b' of LED)	O1	7
Flex 10K20 Pin 8, (segment 'c' of LED)	O2	8
Flex 10K20 Pin 9, (segment 'a' of LED)	O3	9
Flex 10K20 Pin 11, (segment 'e' of LED)	O4	11
Flex 10K20 Pin 12, (segment 'f' of LED)	O5	12
Flex 10K20 Pin 13, (segment 'g' of LED)	O6	13

To implement this pin locking assignment one should utilize the **Pin/Location/Chip Window** (see Figure 4.6) and follow the procedures outlined in Chapter 2. When this is done correctly the pin locking information will appear next to each I/O symbol as shown in Figure 4.5.



**Figure 4.6: Lock Pins.**

## **Project Verification**

Both parts of the hexadecimal counter design have been verified separately in Chapters 2 and 3. Verification of the correctness of the complete design can be made in the manner described in these chapters.

## **Programming (Configuration) Phase**

This phase is identical to that described in Chapter 2. Please refer to this material to configure the FLEX 10K20. When the design has been configured the external LED display should successively and repetitively display the hexadecimal pattern '0' to 'F' in sequence (lowest to highest with wrap around) with out requiring any interaction from the user.

## Chapter 5: References

- [1] *Altera University Program Design Laboratory Package*, Altera Corporation, November 1992.
- [2] *MAX+plus II Programmable Logic Development System: Getting Started*, Altera Corporation.
- [3] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, McGraw Hill, Boston, 2000.
- [4] *VHDL Help*, The Altera MAX+plus II Help Menu.