8 Designing Library Components

This chapter shows design of small components that we will use in the chapters that follow as library components. The purpose is to show various ways a design can be implemented and at the same time generate a reusable library. The library components will be individually tested and symbols will be created for them. In order to test these components, each will be created as a complete project and they will be individually tested. Sequential and combinational components will be designed, and for their design, the use of primitives, megafunctions and Verilog for description of functions will be illustrated.

8.1 Library Organization

Components that we are discussing in this chapter will be placed in a directory called *BookLibrary*. Each component will have its own complete project in this directory. This way, we will be able to test each component by simulation and / or by device programming. For testing our library components we use the MAX 7000S device. Obviously, these components can be used in a design using any programmable device as long as they fit in the device.

8.2 Switch Debouncing – Schematic Entry

This section describes hardware for creating clean filtered pulses from mechanical UP2 pushbuttons. In design of this hardware we will use schematic entry at the gate and functional levels.

Pushbuttons on the UP2 board are mechanical switches and are not debounced. This means that when you press a pushbutton, it makes several

contacts before it stabilizes. The result is that when you press a pushbutton that is **1** in the normal position, its output changes several times between logic **0** and **1** before it becomes **0**, and when you release it, it again switches several times between these logic values before it becomes **1**. Figure 8.1 shows a pushbutton contact bounce.

The problem described above causes no problems in combinational circuits if you give enough time for all changes to propagate before reading the switch's output. However, in sequential circuits with a fast clock, each of the bounces between **0** and **1** logic values may be regarded as an actual logic value. For example, for a counter with a fast clock for which a mechanical pushbutton is used as a count input, pressing the pushbutton may cause several counts.

Figure 8.1 Contact Bounce in a Pushbutton

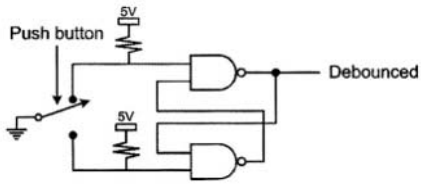

Figure 8.2 Debouncing a Single-Pole Double-Throw (SPDT) Switch

A Single-Pole Double-Throw (SPDT) mechanical switch such as that shown in Figure 8.2 can easily be debounced by an SR-latch also shown in this figure. However, UP2 pushbuttons are Single-Pole Single-Throw (SPST) and their only available terminals are those that connect to logic **1** or logic **0**, as shown in Figure 6.33.

Debouncing UP2 pushbuttons requires a slow clock to sample the switch output before and after it is pressed or released. The clock should be slow enough to bypass all the transitional changes that occur on its output terminal. This section shows generation of a switch debouncer and its necessary clock. For the design of the former part we use schematic entry at the gate level, and for the latter part we use schematic entry using Quartus II megafunctions.

8.2.1 Debouncer – Gate Level Entry

The *debouncer* project is created in the *BookLibrary*. We use schematic entry at the gate level for this design. The design is entered in Quartus II and is tested on the EPM7128S device of UP2.

The design, shown in Figure 8.3, has two inputs *Switch* and *SlowClock*. One of the flip-flops used here is triggered on the rising edge of *SlowClock* and the other is triggered on the falling edge of this clock. Since the output of this circuit is generated by ANDing the two flip-flop outputs, both flip-flops must see

logic **1** on their inputs before the output of the circuit becomes **1**. This means that the pushbutton connected to the *Switch* input of this circuit must stay high for the entire duration of the slow clock for the circuit output to become **1**.

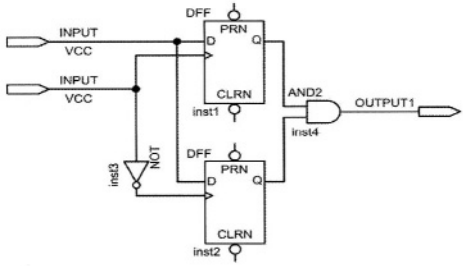

Figure 8.3 Schematic of the *Debouncer*

This design is entered in the Quartus II environment using its block editor. Flip-flops used here are part of the Quartus II library of primitives. These components are categorized in this library under *primitives/storage*. The specific flip-flop used is *dff*.

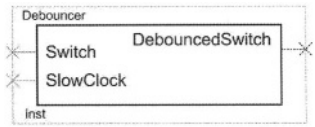

Figure 8.4 Default Symbol Created by Quartus II for *Debouncer*

After entering the schematic of this design, a symbol (shown in Figure 8.4) is created for it by using the Quartus II utility for generating default symbols. This part of the hardware for debouncing pushbuttons uses 3 of the 128 macrocells of the MAX 7000S device.

This part of our design can be simulated, but the real test of this circuit is using it with UP2 pushbuttons. This requires the use of a slow clock that will be created next.

8.2.2 Slow Clock – Using Megafunctions

The frequency of the UP2 clock is 25.175 MHz. Obviously this is too fast for filtering transitions in pushbuttons. Dividing this clock by 2^{21} produces a 12 Hz clock that will be more adequate for filtering slow mechanical transitions. We use a 21 bit counter for dividing the UP2 on-board clock. The Quartus II project for this purpose is called *Divider21* and is created in the *BookLibrary* directory. We demonstrate the use Altera megafunctions for the generation of this circuit.

The design shown in Figure 8.5 uses a 21-bit up-counter. The input is the fast clock and bit 20 of the counter output is the slow clock. The core of this counter is *Divider21c* that is made by configuring a Quartus II megafunction.

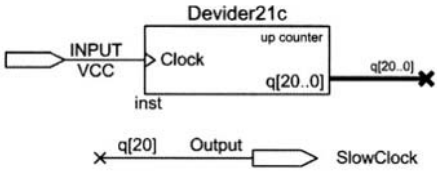

Figure 8.5 Slow Clock Generator

Megafunctions. Just like gates and flip-flops, megafunctions are part of the Quartus II library of components. Unlike gates and flip-flop primitives that are only available with predefined features and inputs, megafunctions are configurable. For example, an OR megafunction can be configured to become an array of n-input OR gates. A counter megafunction can be configured as an up- or down-counter with any number of bits with various forms of load, reset and preset control inputs.

In general, megafunctions are frequently used general purpose digital parts that can be customized according to specific applications. In a way, megafunctions replace the older 7400 series of parts that are available in many technologies for board level designs. The 7400 series packages cover a wide range of functions, but because they are actual physical parts, they only have a limited configurability. Altera megafunctions also cover a wide range of functions. They are described in a hardware description language and because of this, they are far more flexible than the 7400 series that are physical parts.

Megafunctions are available in five categories: *arithmetic, embedded_logic, gates, IO* and *storage*. The *arithmetic* megafunctions cover various forms of adders, counters, and other general purpose arithmetic functions. The *storage* category covers memories, registers, RAMs and ROMs.

Quartus II utility for configuring megafunctions is *MegaWizard Plug-In Manager*. When this utility is invoked, in a series of windows it asks users to specify and configure the megafunction that they have chosen. When done, it generates a schematic symbol for the configured part and generates an HDL design file that corresponds to the symbol. The symbol can be placed on the block editor and used with other configured megafunctions or primitives for completing a design.

To access a megafunction, go through the same process as for placing a primitive in your schematic. When the *Symbol* window appears, in the list of libraries select the standard Quartus II library (i.e., *\quartus\libraries*) and open the *megafunctions* folder in this library. In what follows we show how the *Divider21c* counter of Figure 8.5 is generated.

Frequency Divider. To enter the megafunction counter in the schematic of *Divider21* project, while the schematic entry window (*Block and Symbol Editors*) is open, select the *Symbol Tool* from the corresponding tool bar. This opens the *Symbol* window shown in Figure 8.6. Open this library and in the *arithmetic*

category, select *lpm_counter*. After clicking *OK,* a series of windows will appear that allow you to configure your counter.

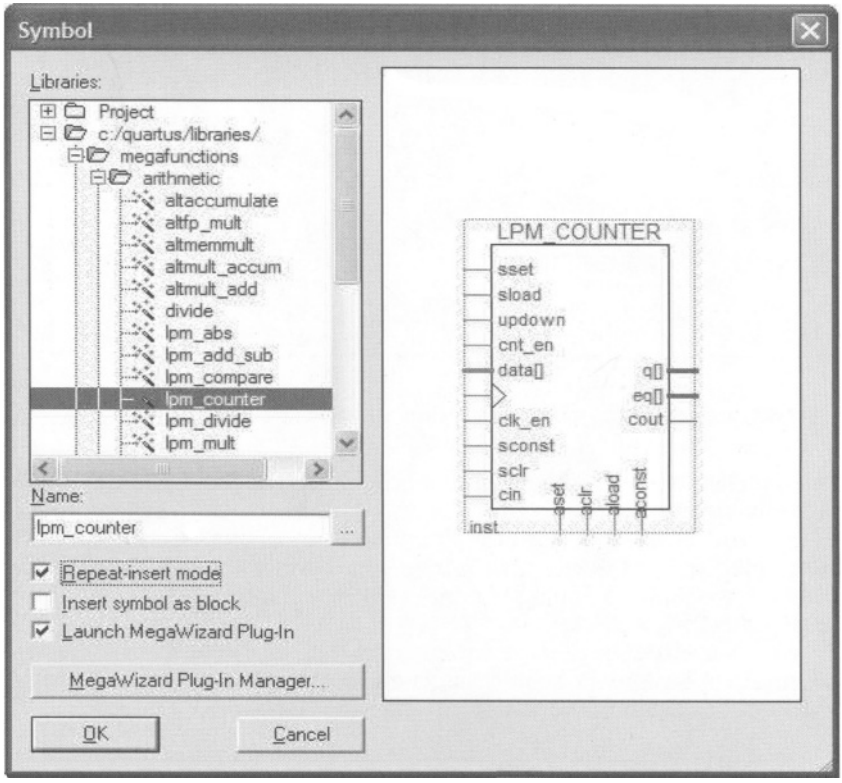

Figure 8.6 Megafunctions Library

In the first window, the HDL used for specifying this megafunction and the name you want to give it is defined. The language used can be any of the three choices shown. Use any language you are most comfortable with. For the name of the megafunction we use *Divider21c*. The next three windows allow you to specify count direction (up or down) number of bits, count sequence (binary or Modulus), enabling mechanism, and set or resetting mechanism (synchronous, asynchronous, etc). Figure 8.7 shows one of these three windows. The last window tells you the flies that are generated and added to your project when this mega function is generated. One of the files created is *Divider21c.bsf* that represents the symbol that corresponds to your configured counter. When configuration of a megafunction is complete, this symbol is placed on your schematic (see Figure 8.5).

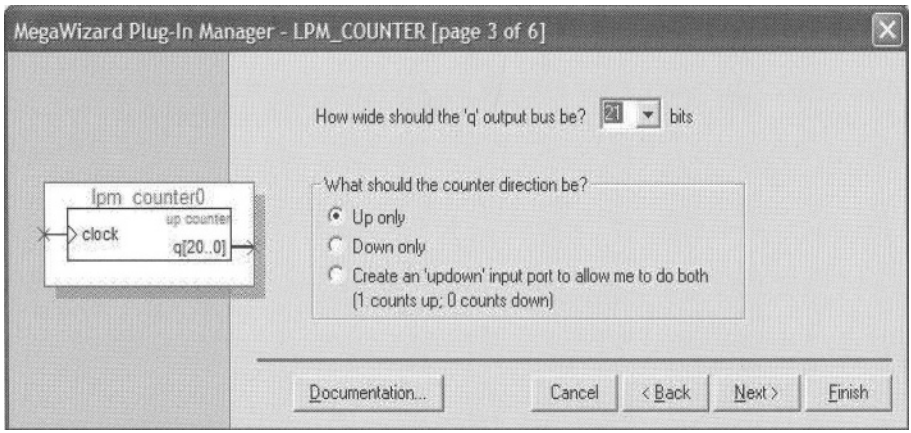

Figure 8.7 Counter Megafunction Configuration Window

After placing *Divider21c* on the schematic, the design of *Divider21* frequency divider will be complete by connecting the input *Clock* to the clock input of *Divider21c* and the *SlowClock* output to *q[20].* As shown in Figure 8.5, connection to *q[20]* port of *Divider21c* is done by connecting a bus to the *q* output of this component, assigning a name to it and using the indexed name for driving the *SlowClock* output.

This completes the design of the frequency divider. In order for this design to be usable in other designs, a symbol is generated for it. This symbol will be used in the complete design of the switch debouncing hardware.

8.2.3 A Debounced Switch – Using Completed Parts

Finally by putting together the hardware of Figure 8.3 with that of Figure 8.5 hardware for debouncing a switch is generated. For this hardware we generate the *Debounced* project in the *BookLibrary* directory, and in its schematic we use symbols for *Debouncer* (Figure 8.4) and *Divider21.*

For placing these symbols in the schematic of *Debounced* project, click on the *Symbol Tool* of the *Block and Symbol Editors* toolbar. When the *Symbol* window opens, in the *Libraries* hierarchy select *Project* (first item in Figure 8.6). This points to symbols created in the directory of our present project. Since the *Debouncer* and *Divider21* projects use the *BookLibrary* directory, their symbols are available in the *Project* directory.

When the *Project* hierarchy opens, select *Debouncer* and *Divider21* symbols and place them on the schematic of the *Debounced* project. The complete schematic diagram of this design is shown in Figure 8.8. The *SlowClock* signal feeds the input of the *Debouncer*; in addition it is pulled out as an output of the *Debounced* hardware. This way, the *SlowClock* output can be shared among multiple *Debouncer* components.

A symbol, shown in Figure 8.9, is created for this design so that it can be used in designs requiring a debounced pushbutton.

For testing our project, it is compiled and it is programmed into the MAX 7000S device of UP2. The *FastClock* input is put on pin number 83 that is the global lock of this chip. The *Switch* input is wired to a pushbutton, and the *CleanSwitch* and *SlowClock* outputs are put on two of the MAX LEDs. The functionality of this circuit is tested by pressing a pushbutton and observing its output. Note that if glitches transmit to the output, because of their short duration, we will not be able to see them on the output LED.

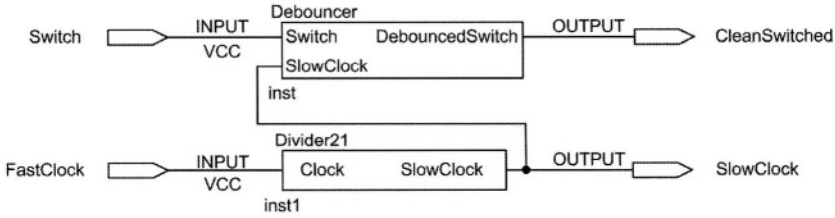

Figure 8.8 Schematic of the *Debounced* Project

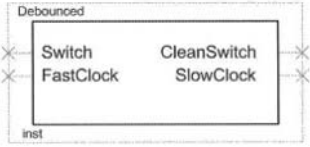

Figure 8.9 *Debounced* Symbol

The complete hardware of the *Debounced* component uses 24 of the 128 macrocells of the MAX 7000S device. Because of this high gate count, it is recommended that for multiple switches, only one *Debounced* is used and the rest use the *Debouncer* of Section 8.2.1 that only uses 3 macrocells.

8.3 Single Pulser – Gate Level

Often, start pulse for a sequential circuit must be only one clock pulse duration. A problem with using pushbuttons for this purpose is that operation of such a switch by human is usually very slow, and the best we can do is to generate pulses of several milliseconds by pushing a push button.

The *OnePulser* project of this section takes a clock and a long pulse as inputs and produces a single pulse of the duration of the clock period for every time the long pulse becomes **1**. The output pulse is synchronous with the clock. The long-pulse input connects to a debounced switch, and the clock input of this circuit connects to the main clock signal of the sequential circuit using the start pulse. The design is done using a block diagram using primitives shown in Figure 8.10.

This design is done by a 2-bit shift register. As shown in Figure 8.10, when *LongPulse* is **0** on the rising edge of the *ClockPulse* at time t_i and **1** on the rising

edge of *ClockPulse* at t_{i+1}, the *OnePulse* output becomes **1** after the edge at t_{i+1} and remain **1** until the next edge that a **1** is shifted into the shift-register. The output waveform of this circuit that is actually a **01** detector is shown in Figure 8.11.

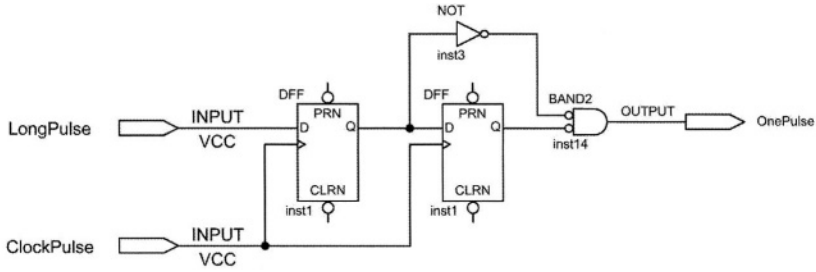

Figure 8.10 The *OnePulser* Circuit

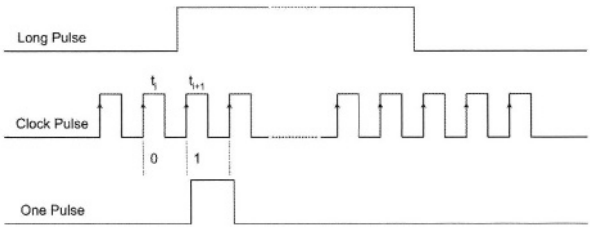

Figure 8.11 *OnePulser* Output Waveform

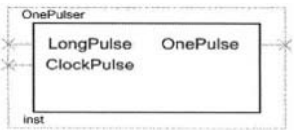

Figure 8.12 *OnePulser* Symbol

The *OnePulser* circuit is tested by instantiating a copy of the *Debounced* circuit (Figure 8.8) in its schematic and driving its inputs. The *CleanSwitch* and *SlowClock* outputs of *Debounced* connect to the *LongPulse* and *ClockPulse* inputs of *OnePulser,* respectively. The *Switch* and *FastClock* inputs of *Debounced* connect to a pushbutton and the system clock respectively, and the *OnePulse* output is displayed on an LED on the UP2 board. By pressing the input pushbutton, a short blink on the output LED indicates the correct operation of this circuit.

The *OnePulser* uses 3 of the 128 macrocells of the MAX device on UP2.

8.4 Debouncing Two Pushbuttons – Using Completed Parts

We generate a project and a design called *Pulser2* to debounce both MAX pushbuttons of the UP2 board. This circuit uses two copies on the *Debouncer* of Figure 8.4 and the *Divider21* of Figure 8.5. This design is shown in Figure 8.13.

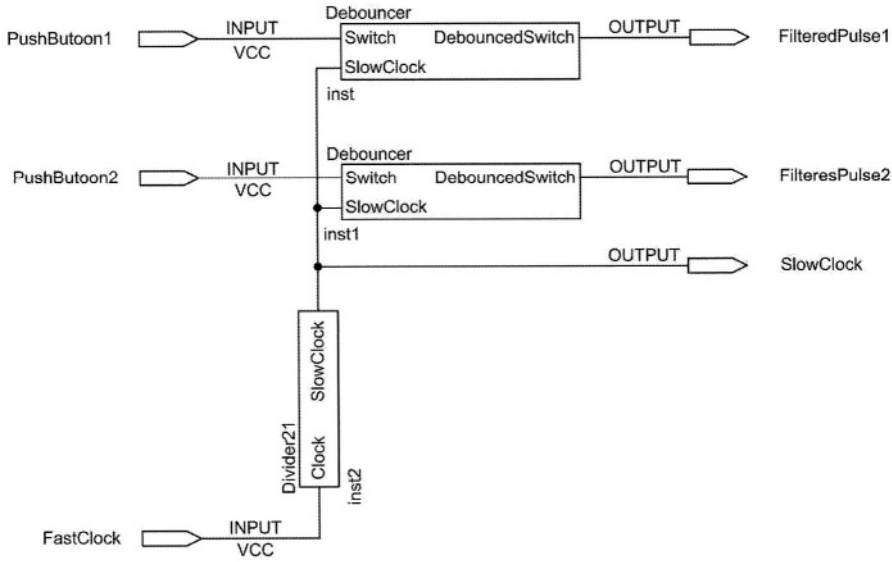

Figure 8.13 ***Pulser2*** **Schematic**

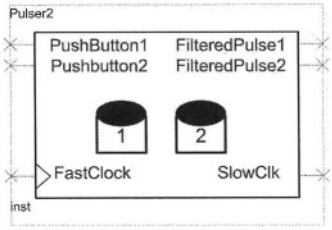

Figure 8.14 ***Pulser2*** **Symbol**

The *Pulser2* symbol is shown in Figure 8.14. For generation of this symbol we first use Quartus II symbol generation utility to generate a default symbol. Using the symbol editor tools (on the *Block and Symbol Editor* toolbar), the default symbol is edited to look as shown in this figure. Tools used from this toolbar are *Text, Rectangle, Oval,* and *Line* tools.

8.5 Hexadecimal Display – Using Verilog

The next library component described in this chapter is a hexadecimal display driver. As with the other components, we will generate a design file and a symbol for this circuit. This circuit takes a 4-bit HEX input and generates Seven Segment Display (SSD) code that corresponds to the input data.

Alternative methods of design entry that exist for creating this design include gate-level schematic entry, table-driven ROM specification, use of megafunctions, use of standard 7400 parts (7400 parts are available in *Libraries\others\maxplus2*), and using an HDL like Verilog. We have chosen the latter method, since it is easy to describe, uses fewer device cells than some other methods, and is adaptable to both MAX and FLEX. The entry method used here starts from the block diagram editor of Quartus II, and enables the use of Verilog for describing a block used in the schematic diagram.

The project used for the display adapter is called *DisplayHEX* and is created in the *BookLibrary* directory. Once this project is defined, we use the *New Block Diagram/Schematic File* tool of the *Applications* toolbar to open a new schematic file. In the schematic, the *HexDecoder* block will be defined to include the Verilog code for the core of our project.

8.5.1 Block Specification

When a new schematic file opens, its corresponding toolbar (*Block and Symbol Editors*) becomes available, and the schematic window is initially blank. Select the *Block* tool from this toolbar to place a blank block on the schematic window. The size of this block is not important at this point and can be adjusted once more details are known about it. Figure 8.15 shows a block that needs to be configured. This becomes our *HexDecoder*.

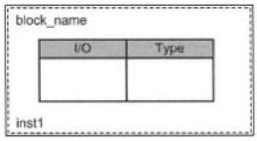

Figure 8.15 An Empty *Block*

8.5.2 Block Properties

The next step in defining a block for inclusion of a design description in Verilog is to specify its name and ports. This is done by specifying block properties. Right-click on the block of Figure 8.15, and in the pull-down menu that opens select *Block Properties*. In the *General* tab of the window of Figure 8.16 the block name (*HexDecoder*) is entered and in its *I/Os* tab its input and output ports are specified.

The 4-bit input of *HexDecoder* is *HEXin[3..0]* and its 7-bit output is *SSDout[6..0]*. The next step is entering the Verilog code of *HexDecoder*.

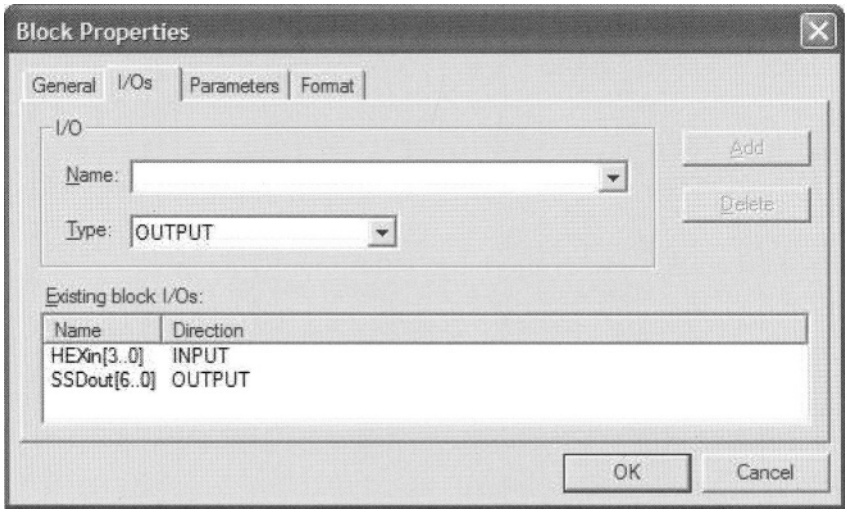

Figure 8.16 Block Properties

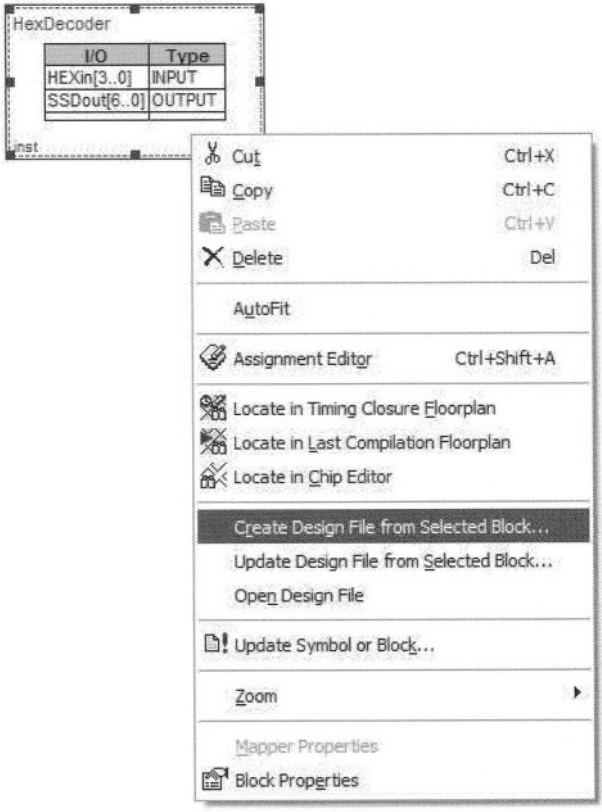

Figure 8.17 Creating a Verilog HDL Design File

8.5.3 Block Verilog Code

With the block properties defined above, Quartus II can generate a Verilog template file for entering the Verilog code of *HexDecoder*. In the schematic of *DisplayHEX,* right-click on the block symbol of *HexDecoder* and in the pull-down menu that opens (Figure 8.17) select *Create Design File from Selected Block* .... This generates a Verilog template that contains declarations and I/O ports of the *HexDecoder* **module.**

The complete Verilog code of *HexDecoder* is shown in Figure 8.18. The *HexDecoder.v* file in the *BookLibrary* contains this code. What is shown here in bold is the part of the code that we have entered for the description of our design. The rest of the code has been generated automatically by Quartus II. The description of *HexDecoder* is now complete.

```
// Generated by Quartus II Version 3.0 (Build Build 199 06/26/2003)
// Created on Thu Mar 11 02:53:38 2004

//  Module Declaration
module  HexDecoder
(
      // {{ALTERA_ARGS_BEGIN}} DO NOT REMOVE THIS LINE!
      HEXin, SSDout
      // {{ALTERA_ARGS_END}} DO NOT REMOVE THIS LINE!
);
// Port  Declaration

      // {{ALTERA_IO_BEGIN}} DO NOT REMOVE THIS LINE!
      input [3:0]  HEXin;
      output [6:0]  SSDout;
      // {{ALTERA_IO_END}} DO NOT REMOVE THIS LINE!

  assign  SSDout =
      HEXin == 4'b0000 ? 7'b0000001 :
      HEXin == 4'b0001 ? 7'b1001111 :
      HEXin == 4'b0010 ? 7'b0010010 :
      HEXin == 4'b0011 ? 7'b0000110 :
      HEXin == 4'b0100 ? 7'b1001100 :
      HEXin == 4'b0101 ? 7'b0100100 :
      HEXin == 4'b0110 ? 7'b0100000 :
      HEXin == 4'b0111 ? 7'b0001111 :
      HEXin == 4'b1000 ? 7'b0000000 :
      HEXin == 4'b1001 ? 7'b0000100 :
      HEXin == 4'b1010 ? 7'b0001000 :
      HEXin == 4'b1011 ? 7'b1100000 :
      HEXin == 4'b1100 ? 7'b0110001 :
      HEXin == 4'b1101 ? 7'b1000010 :
      HEXin == 4'b1110 ? 7'b0110000 :
      HEXin == 4'b1111 ? 7'b0111000 :
                  7'b1111111 ;
endmodule
```

Figure 8.18 ***HexDecoder*** **Verilog Code**

8.5.4 Connections to Block Ports

When a block is defined, it can be used like any symbol in a design file. The *HexDecoder* block in the *DisplayHEX* design file is completely defined and the next step is to wire its ports to other components of this design. Place an input pin symbol and an output pin symbol in *DisplayHEX* schematic. Assign *HEX[3..0]* for the input pin name and *SSD[6..0]* for the output pin. This makes the input a 4-bit bus and the output of *DisplayHEX* a 7-bit bus.

Use the *Orthogonal Bus* tool to make connections from *HEX[3..0]* and *SSD[6..0]*pins to the sides of the *HexDecoder* block. I/O mapper symbols will be placed on the block boundaries where bus connections touch the *HexDecoder* block (see Figure 8.19).

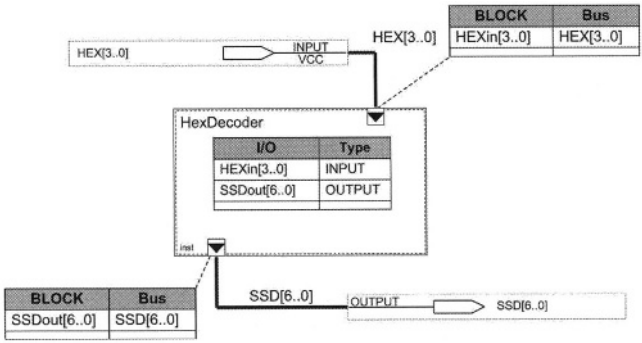

Figure 8.19 Connections of the *DisplayHEX* to *HexDecoder* Block

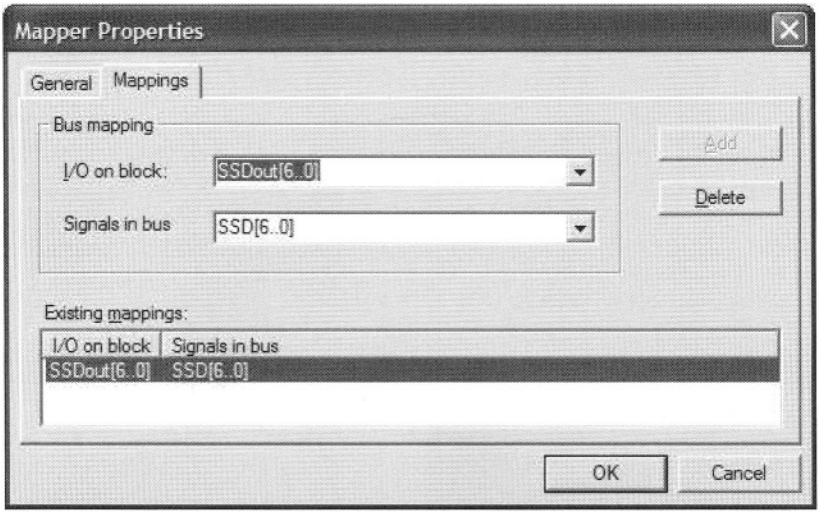

Figure 8.20 Block Port Mapper

The mappers shown in Figure 8.19 must be configured to map outside busses to the internal ports of *HexDecoder*. To configure a mapper, right-click on the mapper on the boundary of *HexDecoder* to bring up the *Mapper Properties* window, shown in Figure 8.20. In the *Mappings* tab of this window select an I/O on block and make a mapping between that and a signal in bus. Figure 8.20 shows a mapping made between *SSDout[6..0]* of *HexDecoder* and *SSD[6..0]* bus of *DisplayHEX*.

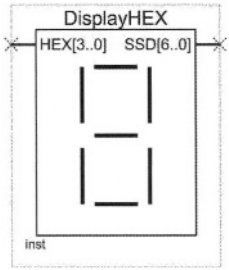

Figure 8.21 Symbol for *DisplayHEX*

8.5.5 Completing *DisplayHEX*

The schematic shown in Figure 8.19 is the complete design of *DisplayHEX* and is compiled in the *BookLibrary*. For it to be accessible by other designs, a symbol is generated that is shown in Figure 8.21. This design uses 5 of the 128 macrocells of the MAX device of UP2 board.

8.6 Summary

This chapter presented various ways designs can be generated in Quartus II. At the same time we presented several utility hardware structures. The structures presented are put in a library to be accessible by designs of the following chapters. On the use of Quartus II, this chapter showed definition and usage of megafunctions, defining and using HDL blocks, using existing components in a design, and editing and customizing component symbols. On the organizational side, this chapter showed how a library of parts could be generated and tested. Finally, from digital design point of view, this chapter showed small, but useful, parts that many designs can use.