Debugging of VHDL Hardware Designs on Altera's DE2 Boards

This tutorial presents some basic debugging concepts that can be helpful in creating VHDL designs for implementation on Altera's DE2 boards. It shows how Quartus II tools can help in the debugging task.

The reader is expected to be familiar with the RTL Viewer and Signal Tap II Logic Analyzer tools included in Altera's Quartus II software. Also, a basic knowledge of simulation is needed.

Contents:

Designers of digital systems are innevitably faced with the task of debugging their imperfect initial designs. This task becomes more difficult as the system grows in complexity. The debugging process requires determination of possible flaws in the designed circuits. This tutorial introduces some basic debugging concepts that can help in designing of circuits specified in VHDL. The tutorial focuses on the debugging of digital hardware. A related tutorial, *Debugging of Application Programs on Altera's DE2 Boards*, deals with the debugging of software programs that are run by Altera's Nios II processor implemented on a DE2 board.

To clarify the concepts used in debugging, we will show how these concepts may be applied to an example circuit.

1 Example Circuit

To make the tutorial easy to follow, we use a relatively simple digital circuit. Our circuit is a tester that can be used to test the reaction time of a person to a visual stimulus. The user initiates the test by pressing and releasing a pushbutton key, KEY_1. After some delay (of at least four seconds) the circuit turns on a light. In response, the user presses another key, KEY_3, as quickly as possible, which results in turning the light off and displaying the ellapsed time in hundredths of a second on the 7-segment displays on the DE2 board.

A block diagram of the circuit is given in Figure 1. The circuit is designed in hierarchical manner, where a number of subcircuits are used to implement the simple tasks. This is a good design practice and the resulting circuit is easier to understand and debug.

Since the pushbutton keys on the DE2 board produce a logic value 0 when pressed, we have chosen to create inverted signals with more meaningful names as follows:

$$\begin{aligned} &reset = !KEY_0 \\ &request\_test = !KEY_1 \\ &stop\_test = !KEY_3 \end{aligned}$$

When the *request_test* signal goes to 1, the *run* signal is activated, which enables the counter circuit that generates the *start_test* signal after a delay of about four seconds. The *start_test* signal activates the *test_active* signal which causes the green light, $LEDG_0$, to be turned on. At this point the four-digit binary-coded-decimal (BCD) counter starts counting in one-hundredth of a second. It is enabled by a pulse (one clock cycle in duration) produced every one-hundredth of a second by the counter circuit called *hundredth*. The four BCD digits, *BCD*0 to *BCD*3, are decoded by the BCD-to-7-segment decoder circuits and displayed on the 7-segment displays, *HEX*0 to *HEX*3. The *clear* signal, which is generated when either *reset* or *stop_test* signal is activated by pressing the respective pushbutton, brings the control signals *run* and *test_active* to 0, which freezes the BCD count at its present value.

The VHDL code for the top-level module, which corresponds to the block diagram in Figure 1, is given in Figure 2. The entity is called the *reaction_tester*. It instantiates modules for each of the subcircuits shown in Figure 1. The statements are numbered for ease of reference in the discussion below. The subcircuits are specified as shown in Figures 3 to 13.

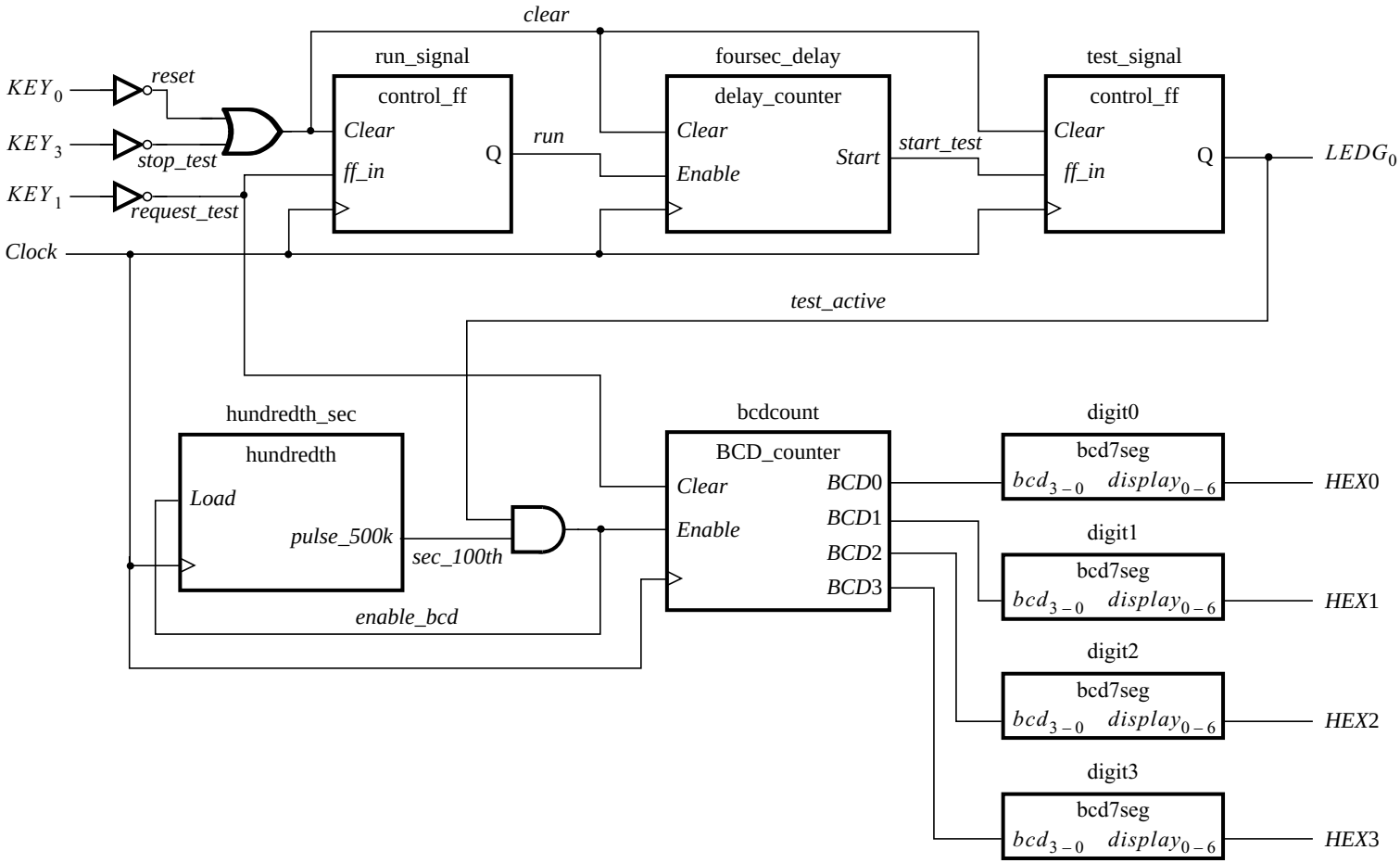


Figure 1. The reaction-tester circuit.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;
3   USE ieee.std_logic_unsigned.all ;
4   ENTITY reaction_tester IS
5       PORT ( CLOCK_50 : IN STD_LOGIC ;
6           KEY : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
7           HEX3, HEX2, HEX1, HEX0 : OUT STD_LOGIC_VECTOR(0 TO 6) ;
8           LEDG : OUT STD_LOGIC_VECTOR(0 DOWNTO 0) ) ;
9   END reaction_tester ;

10  ARCHITECTURE top_level OF reaction_tester IS
11      SIGNAL reset, request_test, stop_test, clear, sec_100th : STD_LOGIC ;
12      SIGNAL run, start_test, test_active, enable_bcd : STD_LOGIC ;
13      SIGNAL BCD3, BCD2, BCD1, BCD0 : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
14      COMPONENT control_ff
15          PORT ( Clock, ff_in, Clear : IN STD_LOGIC ;
16                  Q : BUFFER STD_LOGIC ) ;
17      END COMPONENT ;
18      COMPONENT hundredth
19          PORT ( Clock, Load : IN STD_LOGIC ;
20                  pulse_500k : OUT STD_LOGIC ) ;
21      END COMPONENT ;
22      COMPONENT delay_counter
23          PORT ( Clock, Clear, Enable : IN STD_LOGIC ;
24                  Start : OUT STD_LOGIC ) ;
25      END COMPONENT ;
26      COMPONENT BCD_counter
27          PORT ( Clock, Clear, Enable : IN STD_LOGIC ;
28                  BCD3, BCD2, BCD1, BCD0 : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
29      END COMPONENT ;
30      COMPONENT bcd7seg
31          PORT ( bcd : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
32                  display : OUT STD_LOGIC_VECTOR(0 TO 6) ) ;
33      END COMPONENT ;
34  BEGIN
35      reset <= NOT (KEY(0)) ;
36      request_test <= NOT (KEY(1)) ;
37      stop_test <= NOT (KEY(3)) ;
38      clear <= reset OR stop_test ;
39      enable_bcd <= test_active AND sec_100th ;
40      LEDG(0) <= test_active ;
41      run_signal: control_ff  PORT MAP(CLOCK_50, request_test, clear, run) ;
42      test_signal: control_ff  PORT MAP(CLOCK_50, start_test, clear, test_active) ;
43      hundredth_sec: hundredth  PORT MAP(CLOCK_50, enable_bcd, sec_100th) ;
44      foursec_delay: delay_counter  PORT MAP(CLOCK_50, clear, run, start_test) ;
45      bcdcount: BCD_counter  PORT MAP(CLOCK_50, request_test, enable_bcd, BCD3, BCD2, BCD1, BCD0) ;
46      digit3: bcd7seg  PORT MAP(BCD3, HEX3) ;
47      digit2: bcd7seg  PORT MAP(BCD2, HEX2) ;
48      digit1: bcd7seg  PORT MAP(BCD1, HEX1) ;
49      digit0: bcd7seg  PORT MAP(BCD0, HEX0) ;
50  END top_level ;
```

Figure 2. VHDL code for the top-level module of the example system.

The subcircuit in the *control_ff* modules is defined in Figure 3. It generates an output signal that goes high when the data input *ff_in* goes high, and then maintains this signal, even if *ff_in* becomes low again, until it is cleared by a signal on the *Clear* input. Note that while the inputs may be asynchronous (being connected to pushbutton switches), the operation is synchronized by the clock signal. A circuit generated from this code is displayed in Figure 4.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY control_ff IS
    PORT ( Clock, ff_in, Clear : IN STD_LOGIC ;
        Q : BUFFER STD_LOGIC ) ;
END control_ff ;

ARCHITECTURE control_circuit OF control_ff IS
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            IF Clear = '1' THEN
                Q <= '0' ;
            ELSE
                Q <= ff_in OR Q ;
            END IF ;
        END IF ;
    END PROCESS ;
END control_circuit ;
```

Figure 3. Code for the *control_ff* circuit.

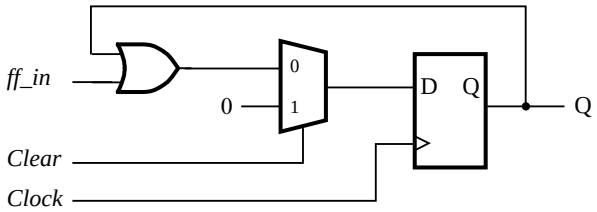


Figure 4. The *control_ff* circuit.

Figure 5 presents the code for the delay counter module, *delay_counter*. This is a 28-bit up-counter. Its most-significant bit, b_{27}, goes to 1 after about four seconds. It is used as the *start_test* signal. The code produces the circuit in Figure 6. Note that the counter is implemented as a 28-bit register (represented by the flip-flop symbol), and an adder which increments the contents of the register by 1.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY delay_counter IS
    PORT ( Clock, Clear, Enable : IN STD_LOGIC ;
           Start : OUT STD_LOGIC ) ;
END delay_counter ;

ARCHITECTURE delay_circuit OF delay_counter IS
    SIGNAL delay_count : STD_LOGIC_VECTOR(27 DOWNTO 0) ;
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            IF Clear = '1' THEN
                delay_count <= (OTHERS => '0') ;
            ELSIF Enable = '1' THEN
                delay_count <= delay_count + '1' ;
            END IF ;
        END IF ;
    END PROCESS ;
    Start <= delay_count(27) ;
END delay_circuit ;
```

Figure 5. Code for the *delay_counter* circuit.

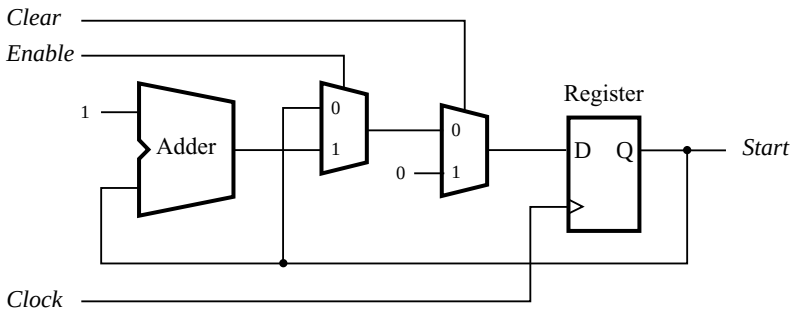


Figure 6. The *delay_counter* circuit.

Figure 7 gives the code that defines the *hundredth* subcircuit. This is a down-counter which generates an output pulse whenever the contents reach the value 0. To produce the interval of one hundredth of a second, the counter is repeatedly loaded with the value $(7A120)_{16}$ which corresponds to 500,000. The output of this circuit, *sec_100th*, allows the BCD counter to be incremented 100 times each second as long as the green light is on. The circuit is shown in Figure 8. This counter is implemented as a 20-bit register (represented by the flip-flop symbol), and an adder which decrements the contents of the register by 1.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY hundredth IS
    PORT ( Clock, Load : IN STD_LOGIC ;
           pulse_500k : OUT STD_LOGIC ) ;
END hundredth ;

ARCHITECTURE hundredth_circuit OF hundredth IS
    SIGNAL count_500k : STD_LOGIC_VECTOR(19 DOWNTO 0) ;
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            IF Load = '1' THEN
                count_500k <= X"7A120" ;
            ELSE
                count_500k <= count_500k − '1' ;
            END IF ;
        END IF ;
    END PROCESS ;
    pulse_500k <= '1' WHEN (count_500k = X"00000") ELSE '0' ;
END hundredth_circuit ;
```

Figure 7. Code for the *hundredth* circuit.

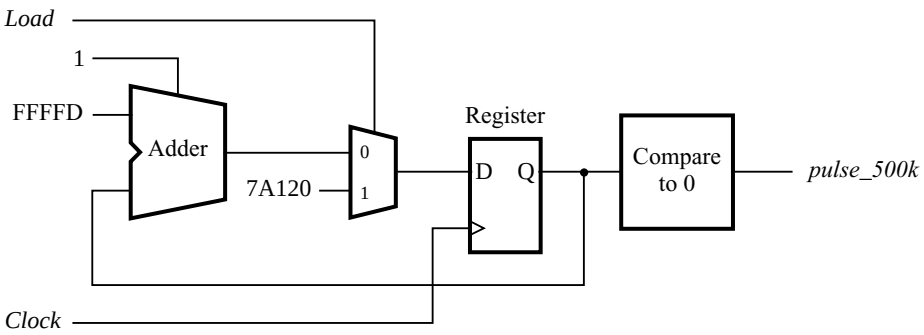


Figure 8. The *hundredth* circuit.

The BCD counter is specified by the code in Figure 9. The circuit for each of the four BCD digits is defined in the module *BCD_stage*. Four versions of this circuit are instantiated in the module *BCD_counter*. Figures 10 and 11 depict the circuits synthesized from the modules *BCD_counter* and *BCD_stage*, respectively.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;
ENTITY BCD_counter IS
    PORT ( Clock, Clear, Enable : IN STD_LOGIC ;
           BCD3, BCD2, BCD1, BCD0 : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END BCD_counter ;
ARCHITECTURE four_digits OF BCD_counter IS
    SIGNAL Carry : STD_LOGIC_VECTOR(4 DOWNTO 1) ;
    COMPONENT BCD_stage
        PORT ( Clock, Clear, Ecount : IN STD_LOGIC ;
               BCDq : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
               Value9 : OUT STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    stage0: BCD_stage  PORT MAP(Clock, Clear, Enable, BCD0, Carry(1)) ;
    stage1: BCD_stage  PORT MAP(Clock, Clear, (Carry(1) AND Enable), BCD1, Carry(2)) ;
    stage2: BCD_stage  PORT MAP(Clock, Clear, (Carry(2) AND Carry(1) AND Enable), BCD2, Carry(3)) ;
    stage3: BCD_stage  PORT MAP(Clock, Clear, (Carry(3) AND Carry(2) AND Carry(1) AND Enable), BCD3, Carry(4)) ;
END four_digits ;


LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;
ENTITY BCD_stage IS
    PORT ( Clock, Clear, Ecount : IN STD_LOGIC ;
           BCDq : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Value9 : OUT STD_LOGIC ) ;
END BCD_stage ;
ARCHITECTURE digit OF BCD_stage IS
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            IF Clear = '1' THEN   BCDq <= "0000" ;
            ELSIF Ecount = '1' THEN
                IF BCDq = "1001" THEN   BCDq <= "0000" ;
                ELSE   BCDq <= BCDq + '1' ;
                END IF ;
            END IF ;
        END IF ;
    END PROCESS ;
    PROCESS ( BCDq )
    BEGIN
        IF BCDq = "1001" THEN   Value9 <= '1' ;
        ELSE   Value9 <= '0' ;
        END IF ;
    END PROCESS ;
END digit ;
```

Figure 9. Code for the *BCD_counter* circuit.

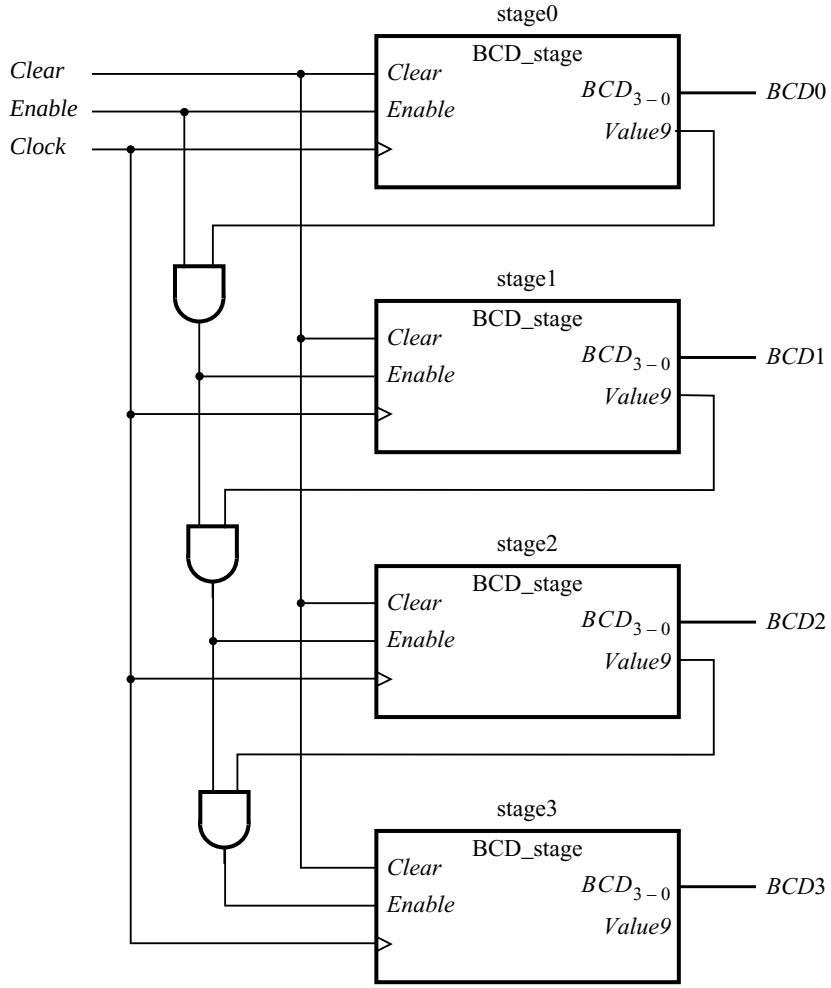


Figure 10. The *BCD_counter* circuit.

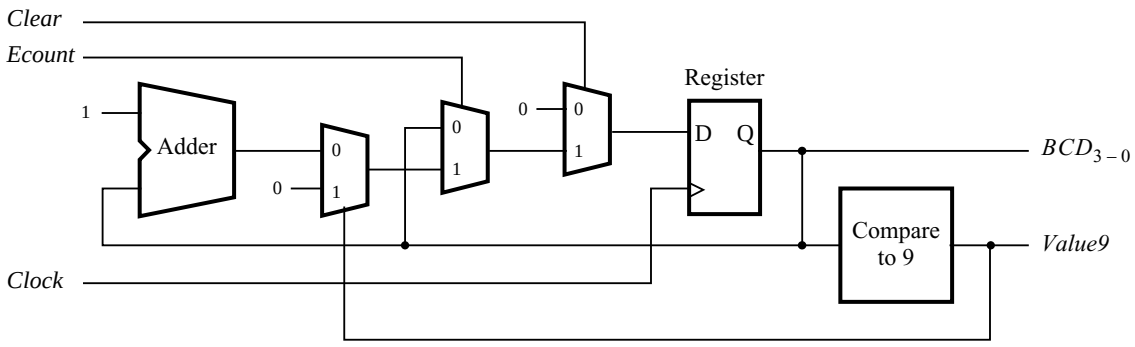
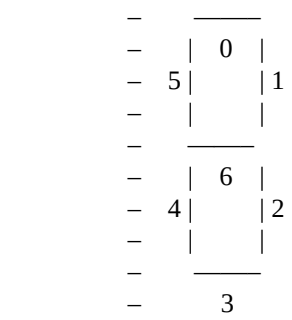


Figure 11. The *BCD_stage* circuit.

Each digit of the BCD counter is converted into a seven-bit pattern for display on a 7-segment display on the DE2 board. This is accomplished by using the circuit *bcd7seg*, which is specified by the code in Figure 12. The comment in the code shows the labeling of the segments that corresponds to the implementation on the DE2 board.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY bcd7seg IS
    PORT ( bcd : IN STD_LOGIC_VECTOR(3 downto 0) ;
           display : OUT STD_LOGIC_VECTOR(0 TO 6) ) ;
END bcd7seg ;
```




```
ARCHITECTURE seven_seg OF bcd7seg IS
BEGIN
    PROCESS ( bcd )
    BEGIN
        CASE bcd IS
            WHEN "0000" =>
                display <= "0000001" ;
            WHEN "0001" =>
                display <= "1001111" ;
            WHEN "0010" =>
                display <= "0010010" ;
            WHEN "0011" =>
                display <= "0000110" ;
            WHEN "0100" =>
                display <= "1001100" ;
            WHEN "0101" =>
                display <= "0100100" ;
            WHEN "0110" =>
                display <= "1100000" ;
            WHEN "0111" =>
                display <= "0001111" ;
            WHEN "1000" =>
            display <= "0000000" ;
            WHEN "1001" =>
                display <= "0001100" ;
            WHEN OTHERS =>
                display <= "1111111" ;
        END CASE ;
    END PROCESS ;
END seven_seg ;
```

Figure 12. Code for the BCD-to-7-segment decoder circuit.

Figure 13 gives a circuit that may result from the code in Figure 12. Each segment of the 7-segment display is driven by a signal generated by a simple decoder from the four bits of a BCD digit. The figure shows only a part used to drive two of the segments, $display_0$ and $display_6$. Each decoder realizes the assignment indicated in Figure 12. Note that the segments of a 7-segment display are illuminated when a ground signal, logic 0, is applied to them. They are turned off when a high-voltage signal, logic 1, is applied.

While Figure 13 shows specific decoder circuits, it is important to keep in mind that the Quartus II compiler may synthesize different looking circuits but with the same functionality.

We will use our example circuit to illustrate the debugging process. To get the most out of this tutorial, create a new Quartus II project, compile the circuit, and follow the discussion by performing the various tasks on your design. All of the files involved in the design are provided with this tutorial.

Before starting the discussion of debugging, we will consider some Quartus II tools that make the debugging task easier.

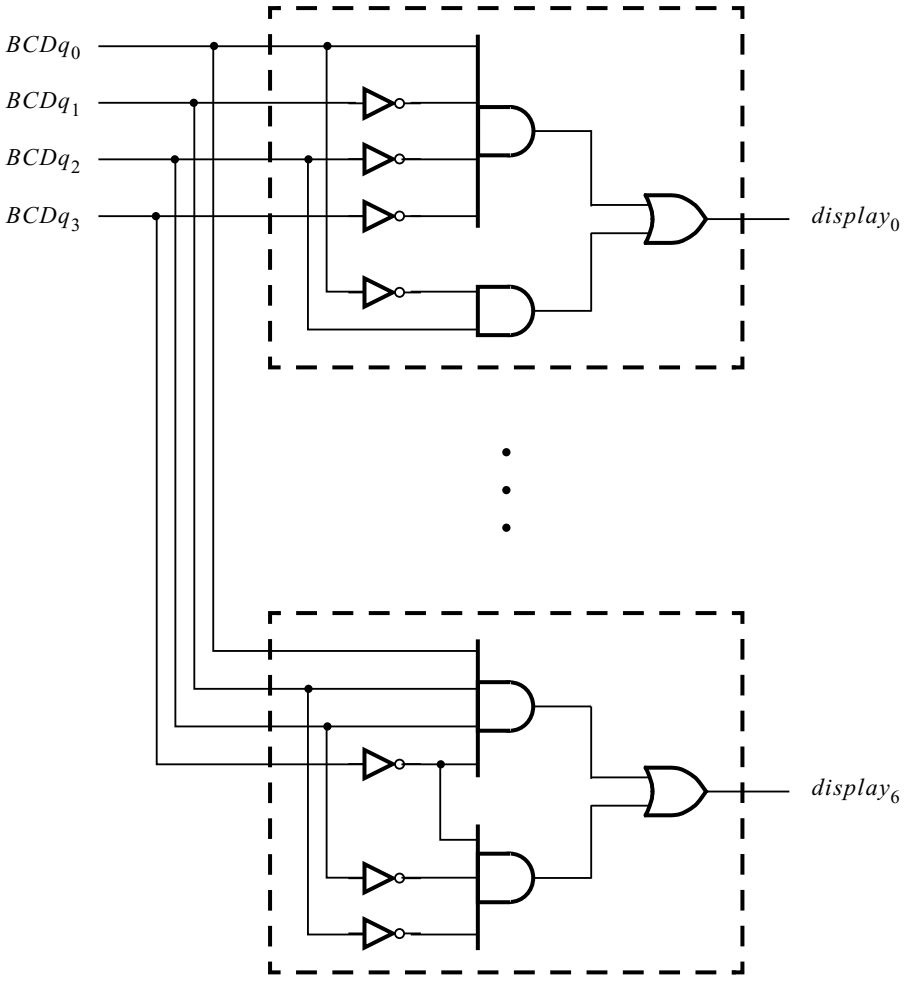


Figure 13. The *bcd7seg* circuit.

2 Quartus II Tools for Use in Debugging of Hardware Designs

The Quartus II software includes many tools that are useful for a variety of purposes. We will discuss three types of tools: *Netlist Viewers*, *SignalTap II Logic Analyzer*, and *Simulator*. While their use is broader, we will restrict

our discussion to their utility as debugging aids.

2.1 Netlist Viewers

The Netlist Viewers provide a graphical indication of a synthesized circuit. A *register transfer level (RTL)* view of a designed circuit, generated after the initial synthesis, can be seen by using the *RTL Viewer*. A view of the final implementation, obtained after *technology mapping*, is available through the *Technology Map Viewer*. If a designed circuit involves a finite state machine, a diagram of this FSM can be examined by means of the *State Machine Viewer*.

2.1.1 RTL Viewer

The RTL Viewer provides a block diagram view of a circuit, at the level of registers, flip-flops and functional blocks that constitute the design. The displayed image is the circuit generated after the analysis and initial synthesis steps. It is not necessary to wait for the rest of the compilation process to be completed, which includes placing and routing the designed circuit. Using the project with our example circuit, activate the initial synthesis process by clicking on the Start Analysis and Synthesis icon in the toolbar. Should this icon not be displayed in the toolbar, it can be found by selecting Processing > Compiler Tool. Upon performing the synthesis, select Tools > Netlist Viewers > RTL Viewer to reach the window depicted in Figure 14. This shows a portion of the designed circuit. The rest of the circuit can be examined by scrolling through this window. The view of the circuit can be enlarged or reduced by means of the Zoom Tool. The complete circuit is given in Figure 15.

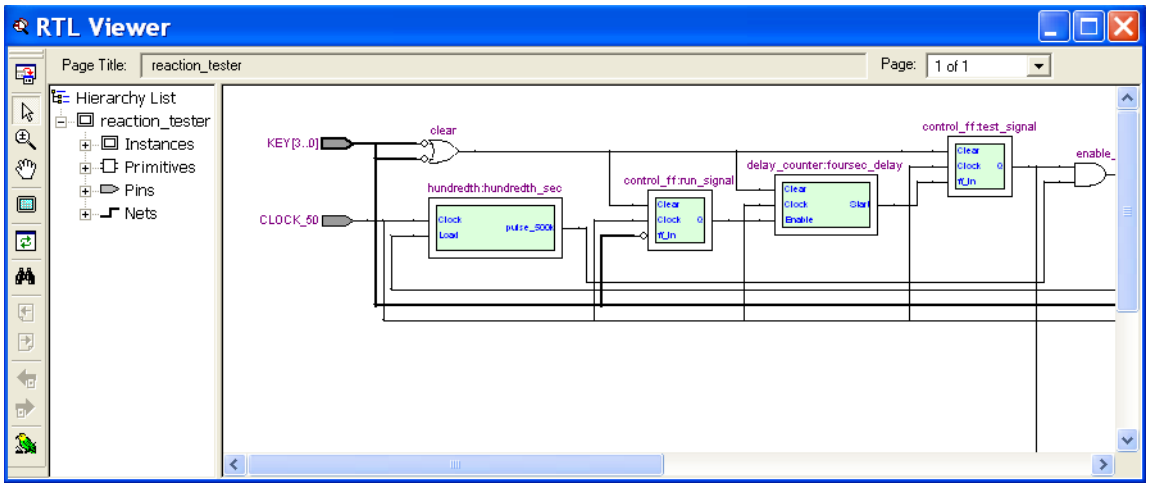


Figure 14. The RTL Viewer.

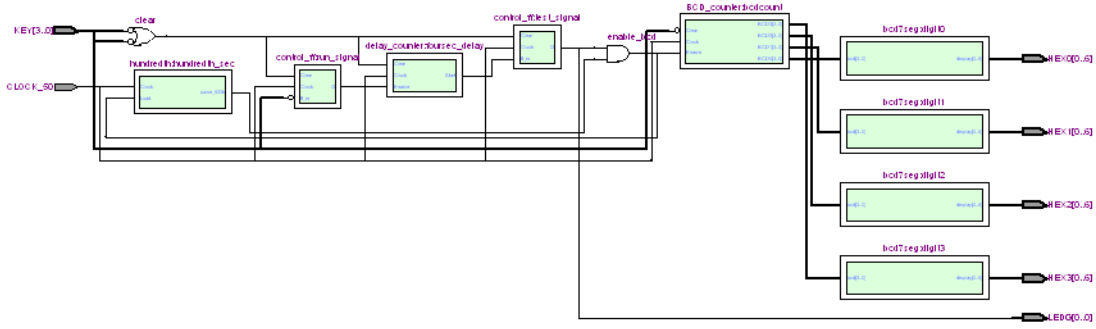


Figure 15. The complete RTL view of the *reaction-tester* circuit.

Double-clicking on any block of the displayed circuit will reveal a more detailed structure of this block. For example, doing this on the block labeled "control_ff:run_signal" produces the image in Figure 16. This is the circuit that we anticipated in Figure 4.

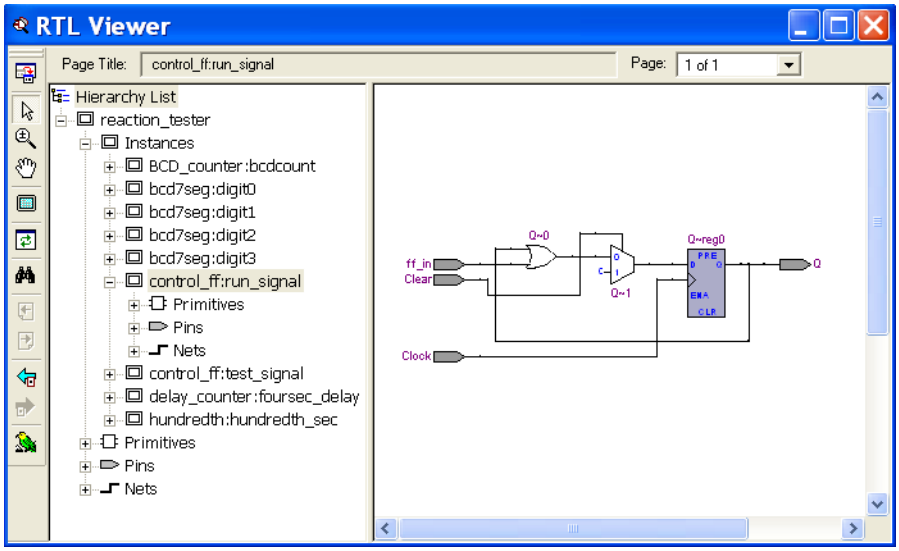


Figure 16. The RTL Viewer presentation of the *control_ff* circuit.

The RTL Viewer is a very useful debugging aid. It allows the designer to quickly see the structure of the circuit that is being designed. It shows the connections between functional blocks. Names of the signals on the connecting wires can be seen by hovering the mouse over the wires, which makes it easy to trace the signals. The displayed diagram includes only those circuit blocks that are driven by valid inputs and produce outputs that connect to other blocks or pins on the FPGA device. Thus, if an expected block is missing, it is very likely that the VHDL specification of the corresponding inputs or outputs is incorrect.

Since the RTL Viewer displays the circuit obtained after the initial synthesis (without needing to perform a complete compilation), it takes relatively little time to see the effect of any changes that are made in the design.

2.1.2 Technology Map Viewer

The Technology Map Viewer can be used to examine a circuit that was compiled. It displays not only the structure of the circuit, but it also indicates the details of logic cells that are used to implement the various parts of the circuit. It is activated by selecting Tools > Netlist Viewers > Technology Map Viewer. Figure 17 shows a portion of the displayed image for our example circuit. Double-clicking on a particular block displays the details of the block.

The displayed image indicates how the designed circuit is implemented in a specific technology. On the DE2 board this is the technology of the Cyclone II FPGA.

2.1.3 State Machine Viewer

The State Machine Viewer can be used to examine the implementation of FSMs that are a part of a designed circuit. It is accessed by selecting Tools > Netlist Viewers > State Machine Viewer.

The FSM implementation is depicted in both graphical and tabular forms. The encoding of states is also presented.

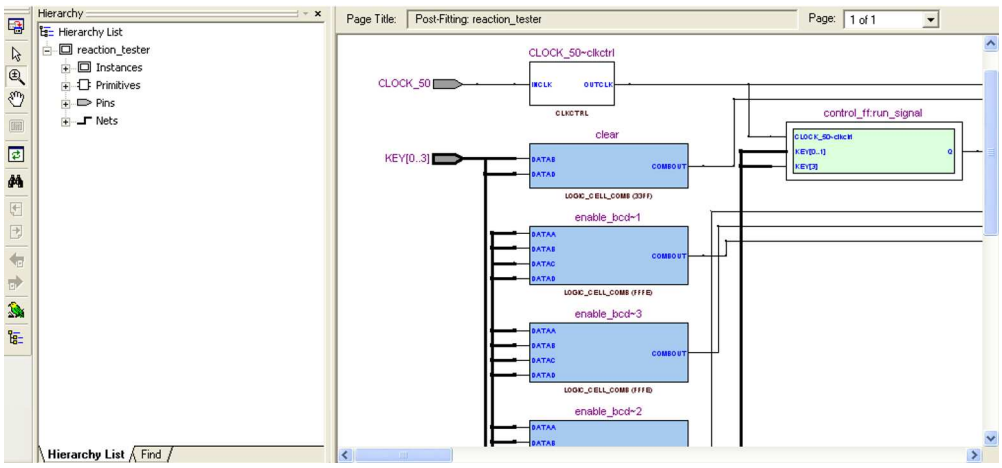


Figure 17. The Technology Map Viewer.

2.2 SignalTap II Logic Analyzer

A traditional Logic Analyzer is an instrument that can display waveforms that correspond to signals obtained by connecting probes to various points in an implemented circuit. For an FPGA device, it is only possible to gain such access to external pins. However, Quartus II software includes a software-implemented tool that acts as a virtual logic analyzer, which allows the user to examine signals that are going to occur anywhere within a circuit implemented in an FPGA chip. It is called the SignalTap II Logic Analyzer. Its use is described in the tutorial *SignalTap II with VHDL Designs*.

Figures 18 and 19 indicate how the analyzer may be used on our example circuit. We chose to look at several signals that are affected by the *start_test* signal going to 1. As seen in Figure 18, a positive edge of this signal is enabled as the trigger that causes the analyzer to take a snapshot of signal activity. Figure 19 shows the waveforms that occur at trigger time. Observe that the *test_active* signal goes to 1 in the next clock cycle (as expected). Also, observe that the contents of the *hundredth* counter, called *count_500k* in Figure 7, are decremented by 1 in each clock cycle.

It is important to know that the Quartus II Compiler will not necessarily preserve the exact names of signals in combinational logic as defined in a VHDL design file. Also, when the Node Finder is used to find signals that the designer wants to include in the Setup window of the SignalTap II Logic Analyzer, many signals that are not registered or found on the FPGA pins may not be listed. It is possible to force the listing of a particular signal under its original name by means of the "keep" option, which is invoked by defining the attribute *keep*. For example, we can ensure that the *run* and *enable_bcd* signals will be preserved and listed by inserting in the code in Figure 2 the following statements:

```
ATTRIBUTE  keep:  BOOLEAN ;
ATTRIBUTE  keep OF run:  SIGNAL IS true ;
ATTRIBUTE  keep OF enable_bcd:  SIGNAL IS true ;
```

Using the keep option may result in a slightly different circuit being synthesized. The circuit will have the same functionality as the intended circuit, but it may have a slightly different timing behavior. Therefore, upon successful completion of the debugging (or simulation) task, the modifications inserted to invoke the keep option should be removed.

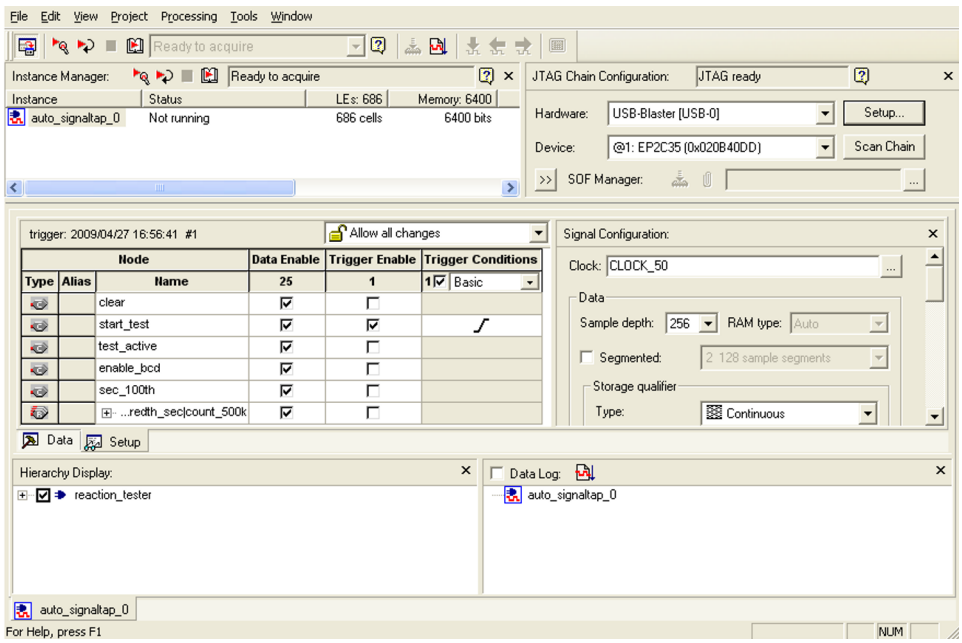


Figure 18. The Setup window of SignalTap II Logic Analyzer.

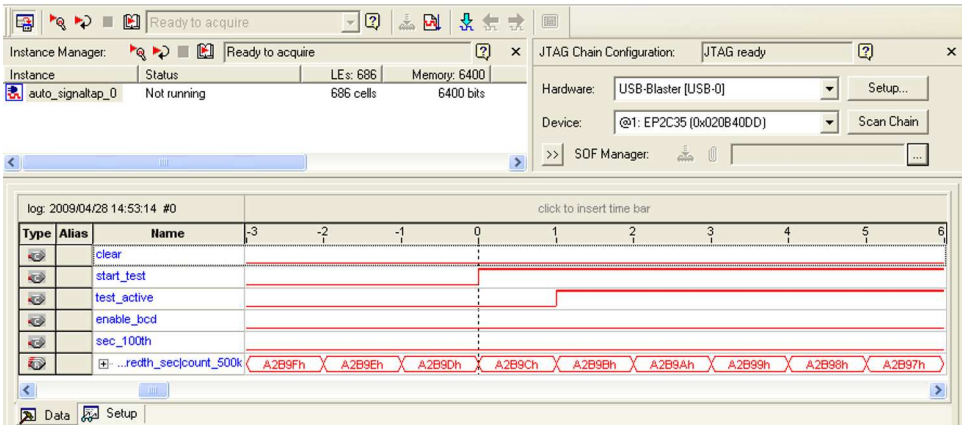


Figure 19. The Data window of SignalTap II Logic Analyzer.

2.3 Simulators

The tools discussed above are very useful in determining whether or not a given circuit appears to have the desired structure and functionality. To test the expected functional correctness and performance of the designed circuit it is useful to simulate the circuit. For example, a circuit that performs extensive arithmetic operations may appear to to be designed correctly in terms of the components it contains, but a small error in detail (which could be difficult to detect using either a netlist viewer or the logic analyzer) can cause wrong results to be produced when a particular operation is performed. Functional simulation provides an excellent vehicle for ascertaining that the circuit performs correctly as far as its functionality is concerned. It is also important to ensure that the timing behavior of the circuit meets the specification requirements, which can be determined by means of timing simulation.

A complete simulation of our example circuit would require a large number of clock cycles, making it difficult to produce an informative display. However, we can perform a meaningful simulation by scaling down some parameters. For example, let us reduce the *delay_counter* circuit to be a 4-bit counter so that the *start_test* signal will go to 1 after a delay of 8 clock cycles. Let us also reduce the *hundredth* counter to be a 3-bit counter into which the value 4 is loaded whenever the *Load* signal is active. A functional simulation of this scaled-down circuit is shown in Figure 20.

We applied the input signals that correspond to the pushbutton keys. The observed behavior of the simulated circuit is correct. The BCD counter evaluates correctly the number of *sec_100th* pulses that occur before the KEY_3 signal goes to 0 (which corresponds to pressing of the pushbutton). The BCD-to-7-segment decoders also correctly decode the BCD digits, which is easily verified by examining the displayed patterns.

The simulation indicates that our circuit produces a slightly inaccurate result. Before the test starts, the *hundredth* counter runs in a counting span of 8 clock cycles, as shown by the *sec_100th* signal in Figure 20. During the test this span becomes 4 cycles, because this is the value repeatedly loaded into the counter. However, at the very beginning of the test the counter may contain the value 7, which means that it will take 8 clock cycles before the BCD counter starts counting. This means that our tester circuit may be wrong by 1/100th of a second. If we could not tolerate this inaccuracy, we would have to modify the circuit.

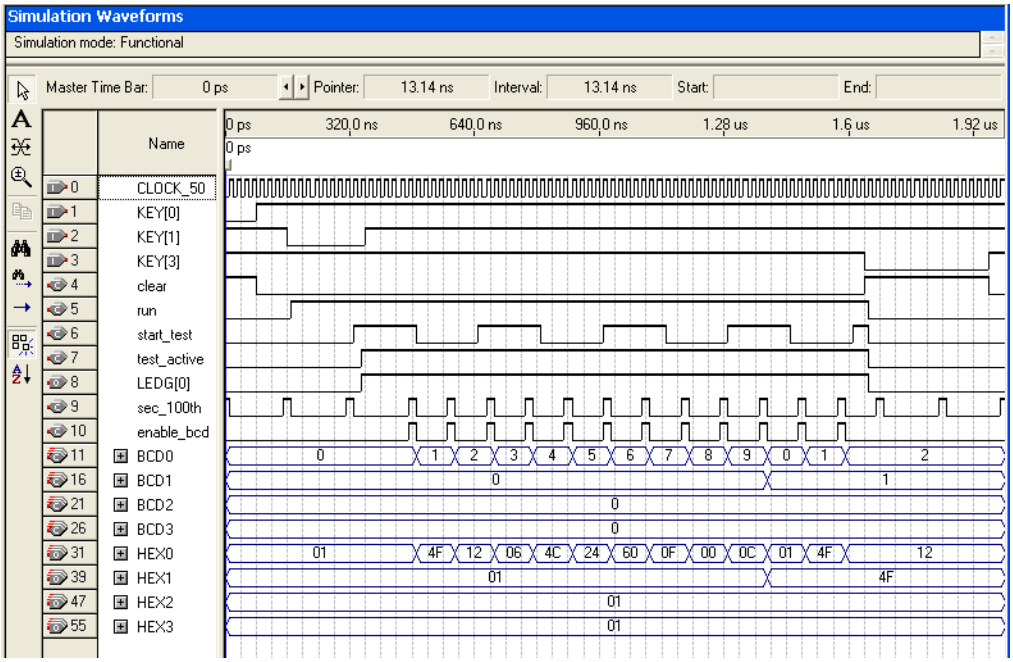


Figure 20. The result of functional simulation.

Quartus II software includes the simulation tools. They are described in the tutorial *Quartus II Simulation with VHDL Designs*. We encourage the user to use the ModelSim simulator, particularly when large circuits are involved.

3 Debugging Concepts

Debugging of complex logic circuits can be difficult. The task is made easier if one uses an organized approach with the aid of debugging tools. The debugging task involves:

- Observing that there is a problem
- Identifying the source of the problem
- Determining the design changes that have to be made

- Changing and re-implementing the designed circuit
- Testing the corrected design

3.1 Observing a Problem

Often it is easy to see that there is a problem because the designed circuit does not match the designer's expectations in an obvious way. For example, a graphical image of the circuit displayed by the RTL Viewer may indicate that there are missing logic blocks and/or connections.

Consider an error where line 39 in Figure 2 reads

enable_bcd <= test_active;

Compiling the design and testing the resulting circuit would show that the circuit simply does not work. Examining the designed circuit with the RTL Viewer gives the image in Figure 21. It is apparent that there is no output from the block *hundredth_sec*. The reason is that the Compiler recognized that the signal *sec_100th* is not used as an input anywhere in the rest of the circuit, hence it omitted this signal. Making this observation the designer would quickly discover the error in line 39.

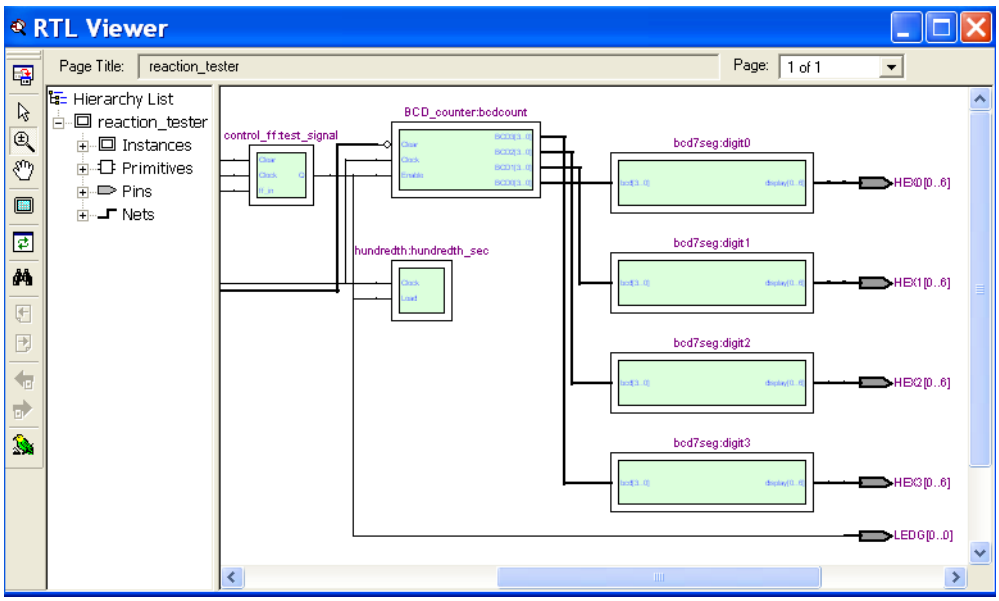


Figure 21. The erroneous circuit displayed by the RTL Viewer.

As another example, suppose that the designer assumes erroneously that the elements of a VHDL vector that refers to the segments of a 7-segment display are labeled as going from 6 to 0, which would mean that line 7 in Figure 2 would read

HEX3, HEX2, HEX1, HEX0 : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);

Compiling the design would result in a circuit that seems to respond properly to pressing of the input keys, but generates a strange-looking output on the 7-segment displays. Observing this behavior, the designer may suspect that there is something wrong with the display of BCD digits. A possible test is to see if the BCD counter generates a plausible result, which is easily accomplished by using the SignalTap II Logic Analyzer. Figure 22 shows an image that we obtained by triggering on the *clear* signal's rising edge (going from 0 to 1), which happens in response to KEY_3 being pressed causing the timer to stop counting. The logic analyzer display indicates that the BCD value should be 0023. However, the 7-segment displays depict the two least-significant digits as 5E and the two most-significant digits as upside-down letters AA. The latter fact provides an immediate clue because

the difference between the inverted A and the expected 0 is in segments labeled 0 and 6 in Figure 12, which are reversed. This should lead to a quick detection of the error that we created.

Note also that Figure 22 indicates that the control signals appear to work correctly. One clock cycle after the active edge of the *clear* signal, the *start_test* and *test_active* signals go to 0. The *sec_100th* and *enable_bcd* signals are both equal to 0, because they are equal to 1 only during one clock cycle in a 1/100 second period.

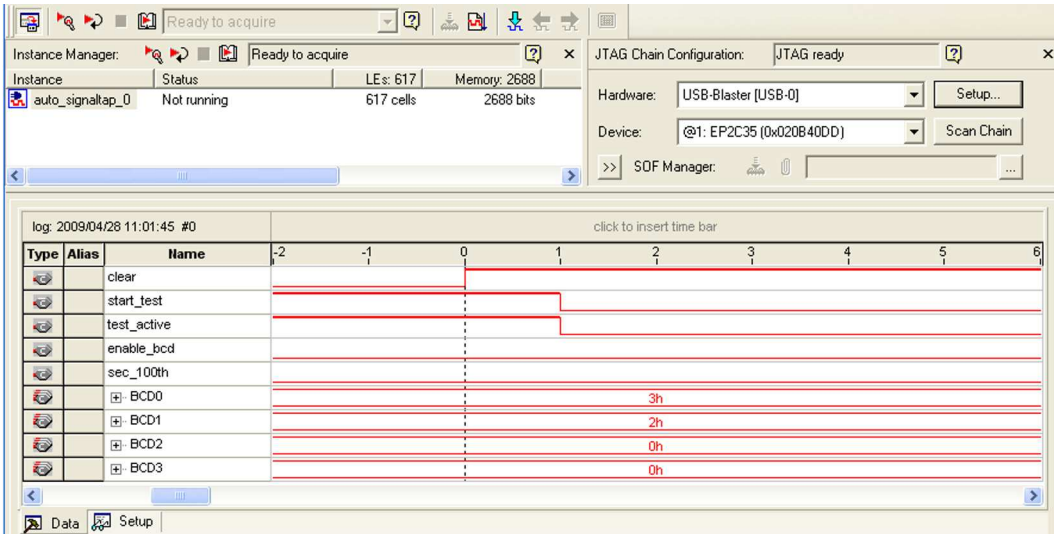


Figure 22. Using the SignalTap II Logic Analyzer to observe the output of the BCD counter.

A complex circuit may be difficult to debug. The circuit implementation may appear to contain all necessary components, it may appear to function properly, but the results it produces do not exhibit the expected behavior. In such cases, the first task is to identify the source of the problem.

3.2 Identifying the Problem

Designer's intuition (which improves greatly with experience) may suggest some tests that could be tried. Otherwise, it is necessary to adopt an organized procedure. A golden rule is to first test small portions of the circuit, which should be easy to do if the circuit is designed in modular fashion. This is referred to as the divide-and-conquer approach.

Our example circuit is constructed in modular fashion. Each module in Figure 1 can be tested separately by using the SignalTap II Logic Analyzer. It is also useful to compile, simulate and test each module on its own, before it is included in the bigger circuit.

It may be helpful to functionally exercise only a portion of a designed circuit, which can show whether or not this portion is working correctly. For example, we can test the BCD counter and the 7-segment displays by isolating this part from the rest of the circuit and providing separately-controlled inputs to this subcircuit. One way of doing this is to use a manual clock instead of the system clock, which would allow us to see the changes (on the 7-segment displays) that take place during the counting process. To accomplish this, we can change line 45 in Figure 2 to read

```
bcdcount: BCD_counter  PORT MAP(KEY(2), request_test, '1', BCD3, BCD2, BCD1, BCD0);
```

Now, KEY_2 is used as a manual clock and the counter is enabled at all times (by connecting 1 to the enable input). Then, pressing KEY_2 repeatedly will step the counter in the upward direction which should be observable on the displays. Note that the BCD counter can be cleared by pressing KEY_1, but only when an active clock signal edge arrives (as a result of pressing KEY_2) because the *BCD_counter* module uses synchronous clear.

4 Sources of Errors in VHDL Designs

The Quartus II Compiler can detect many errors in VHDL files that specify a given circuit. Typical errors include incorrect syntax, undeclared inputs or outputs, improper use of variables and incorrect sizes of vectors. The compiler stops compilation and displays an error message. Such errors are usually easy to find and correct. It is much more difficult to find errors in a circuit that appears to be correctly specified but the specification does not result in a circuit that the designer hoped to achieve. In this section we will consider some typical errors of this type.

Some common errors in VHDL designs are:

- Inadvertent creation of latches
- Omission of signals
- Not assigning a value to a wire
- Assigning a value to a wire more than once
- Incorrect specification of PORT MAP signals
- Wrong definition of a signal vector
- Incorrectly specified FSM (e.g. wrong or invalid next state)
- Incorrect timing where the output signal of a given circuit is off by one clock cycle
- Careless use of clocks

Inadvertent latches are created by the Compiler if the designer fails to specify the action needed for all cases in constructs where a certain number of cases are expected to be specified in what is supposed to be a combinational circuit (e.g. in IF-ELSE and CASE statements).

If the designer fails to use some signals in a VHDL design file, the Compiler will ignore these signals completely and may even omit the circuitry associated with these signals.

Incorrect definitions of signal vectors lead to problems, as illustrated in section 3.1.

Errors in the specification of an FSM may lead to a variety of undesirable consequences. They can cause wrong functional behavior by reaching wrong states, as well as wrong timing behavior by producing incorrect output signals. A common error results in an output signal that is off by one clock cycle.

It is particularly important to use clocks carefully. For example, a slower clock may be derived by using a counter to divide down the main system clock. Timing problems may arise when signals generated in a circuit controlled by one clock are used as inputs to a circuit controlled by a different clock. Whenever possible, all flip-flops should be driven by the same clock. For instance, if a given counter has to be incremented/decremented at a rate that is slower than the system clock rate, it is best to drive the counter with the system clock and use a slower changing *enable* signal to make the counter count at a slower rate. We used this approach in our example circuit to control the BCD counter.

5 Errors Due to Wrong Interpretation of DE2 Board Characteristics

Inadequate understanding of the DE2 board can lead to design errors. Typical examples include:

- Wrong pin assignment
- Wrong interpretation of the polarity of pushbutton keys and toggle switches
- Timing issues when accessing various chips on the board, such as the SDRAM memory

If pins are not assigned correctly, the circuit will not exhibit the desired behavior. This may be easy to detect when obviously observable input and output signals are involved. If the designer specifies a wrong assignment for a pushbutton key, then pressing this key will probably have no effect. If the connection to a 7-segment display is not made at all, the display will show the pattern 8. This means that all seven segments are driven by a logic 0

signal, because a segment lights up when connected to ground voltage. The Quartus II Compiler causes all unused pins to be driven to ground by default. (Of course, this default choice can be changed (in the Quartus II project) by specifying a different option for the unused pins.) The easiest way of ensuring that the pins are correctly assigned for the DE2 board is to import the pin-assignment file *DE2_pin_assignments.csv*.

Pushbutton switches produce logic 0 when pressed. Toggle switches generate logic 1 when in the up position (towards the middle of the board).

If the design involves access to the SDRAM chip, it is necessary to adhere to strict timing requirements, as explained in the tutorial *Using the SDRAM Memory on Altera's DE2 Board with VHDL Design*.

6 Design Procedure

It is prudent to follow a design procedure that tends to minimize the number of design errors and simplifies the debugging task. Here are some suggestions that are likely to help:

- Design the circuit in a modular, hierarchical manner.
- Use well-understood and commonly-used constructs to define circuits.
- Test each module, by simulating it, before it is incorporated into the larger circuit.
- Define and test portions of the final circuit by connecting two or more modules.
- Construct the complete circuit and test it through simulation. Both functional and timing simulation should be done.
- Download the compiled circuit into the FPGA on the DE2 board and test it.

It is prudent to write VHDL code in a style that allows one to easily visualize the circuit specified by the code. It is also useful to make the code easily understandable for other people.