Using the ModelSim-Intel FPGA Simulator with VHDL Testbenches

For Quartus® Prime 18.0

1 Introduction

This tutorial introduces the simulation of VHDL code using the *ModelSim-Intel FPGA* simulator. We assume that you are using *ModelSim-Intel FPGA Starter Edition version 18.0*. This software can be downloaded and installed from the *Download Center for Intel FPGAs*. In this download center, you can select release *18.0* of the *Quartus Prime Lite Edition*, and then on the `Individual Files` tab choose to download and install the *ModelSim-Intel FPGA Starter Edition* software. We assume that you are using a computer that is running the Windows operating system. If you are using the Linux operating system then minor differences to the instructions would apply, such as using a / filesystem delimiter rather than the \ delimiter that is used with Windows.

Contents:

- Getting Started with ModelSim
- Simulating a Sequential Circuit
- Simulating a Circuit that Includes a Memory Module
- Setting up a ModelSim Simulation
- Using the ModelSim Graphical User Interface

Requirements:

- ModelSim-Intel FPGA Starter Edition software
- A computer running either Microsoft* Windows* (version 10 is recommended) or Linux (Ubuntu, or a similar Linux distribution). The computer would typically be either a desktop computer or laptop, and is used to run the ModelSim software.

Optional:

- Intel Quartus® Prime software
- A DE-series development and education board, such as the DE1-SoC board. These boards are described on Intel's FPGA University Program website, and are available from the manufacturer Terasic Technologies.

2 Getting Started

The ModelSim Simulator is a sophisticated and powerful tool that supports a variety of usage models. In this tutorial we focus on only one design flow: using the ModelSim software as a *stand-alone* program to perform *functional* simulations, with simulation inputs specified in a *testbench*, and with simulator commands provided via *script* files. Other possible design flows for using ModelSim include *invoking* it from within the Intel Quartus Prime software, performing *timing* simulations, and specifying simulation inputs by *drawing* waveforms in a graphical editor instead of using a testbench. These flows are described in other documentation that is available on the Internet.

To introduce the *ModelSim* software, we will first open an existing simulation example. The example is a multibit adder named *Addern*, and is included as part of the *design files* provided along with this tutorial. Copy the *Addern* files to a folder on your computer, such as *C:\ModelSim_Tutorial\Addern*. In the *Addern* folder there is a VHDL source-code file called *Addern.vhd* and a subfolder named *ModelSim*. The *Addern.vhd* file, shown in Figure 1, is the VHDL code that will be simulated in this part of the tutorial. We will specify signal values for the adder's inputs, *Cin*, *X*, and *Y*, and then the *ModelSim* simulator will generate corresponding values for the outputs, *S* and *Cout*.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

ENTITY Addern IS
   GENERIC (n : INTEGER := 16);
   PORT ( X, Y : IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
          Cin  : IN STD_LOGIC;
          S    : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0);
          Cout : OUT STD_LOGIC);
END Addern;

ARCHITECTURE Behaviour OF Addern IS
   SIGNAL Sum : STD_LOGIC_VECTOR(n DOWNTO 0);
BEGIN
   Sum <= ('0' & X) + ('0' & Y) + Cin;
   S <= Sum(n-1 DOWNTO 0);
   Cout <= Sum(16);
END Behaviour;
```

Figure 1. VHDL code for the multibit adder.

We will use three files, included in the *ModelSim* subfolder, to control the *ModelSim* simulator. The files are named *testbench.vht*, *testbench.tcl*, and *wave.do*.

The *testbench.vht* file is a style of VHDL code known as a *testbench*. The purpose of a testbench is to *instantiate* a VHDL entity that is to be simulated, and to specify values for its inputs at various simulation times. In this case the module to be simulated is our multibit adder, which we refer to as the *design under test* (DUT). Line 5 is the start of the testbench entity, which has no inputs or outputs. Line 9 declares the *Addern* component, which will be instantiated later in the testbench code. In Lines 16 to 18 we declare signals to drive the adder inputs *Cin*, *X*, and *Y*. Lines 19 and 20 declare signals to connect to the adder outputs *S* and *Cout*.

Lines 22 to 36 provide a process labeled `vectors` that is used to specify the values of the adder inputs. First, in Line 24 the adder inputs *X*, *Y*, and *Cin* are set to 0. The next line of code waits until 20 ns of simulation time has passed, and then line 26 changes the input *Y* to the 10. Following another 20 ns of waiting time, meaning at 40 ns in simulation time, line 28 changes the input *X* to 10. The rest of the process specifies various values for the adder inputs at 20 ns time increments. Finally, in Line 38 the *testbench* entity instantiates the *Addern* entity. Its inputs are driven by the testbench signal values specified in the `vectors` process.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.std_logic_signed.all;
4
5  ENTITY testbench IS
6  END testbench;
7
8  ARCHITECTURE Behavior OF testbench IS
9      COMPONENT Addern
10         PORT ( X, Y : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
11                Cin  : IN STD_LOGIC;
12                S    : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
13                Cout : OUT STD_LOGIC );
14     END COMPONENT;
15
16     SIGNAL Cin : STD_LOGIC;
17     SIGNAL X : STD_LOGIC_VECTOR(15 DOWNTO 0);
18     SIGNAL Y : STD_LOGIC_VECTOR(15 DOWNTO 0);
19     SIGNAL S : STD_LOGIC_VECTOR(15 DOWNTO 0);
20     SIGNAL Cout : STD_LOGIC;
21 BEGIN
22     vectors: PROCESS
23     BEGIN
24         X <= X"0000"; Y <= X"0000"; Cin <= '0';
25         WAIT FOR 20 ns;
26         Y <= X"000A"; Cin <= '0';
27         WAIT FOR 20 ns;
28         X <= X"000A"; Cin <= '0';
29         WAIT FOR 20 ns;
30         Cin <= '1';
31         WAIT FOR 20 ns;
32         X <= X"FFF0"; Y <= X"000F"; Cin <= '0';
33         WAIT FOR 20 ns;
34         Cin <= '1';
35         WAIT;
36     END PROCESS;
37
38     U1: Addern PORT MAP (X, Y, Cin, S, Cout);
39 END;
```

Figure 2. The VHDL testbench code.

Open the *ModelSim* software to reach the window shown in Figure 3. Click on the *Transcript* window at the bottom of the figure and then use the `cd` command to navigate to the ModelSim folder for the multibit adder. For example, in our case we would type `cd C:/ModelSim_Tutorial/Addern/ModelSim`. Note that ModelSim uses the / symbol to navigate between filesystem folders, even though the Windows operating system uses the \ symbol for this purpose. Next, we wish to run a series of simulator commands that are included in the script file *testbench.tcl*.

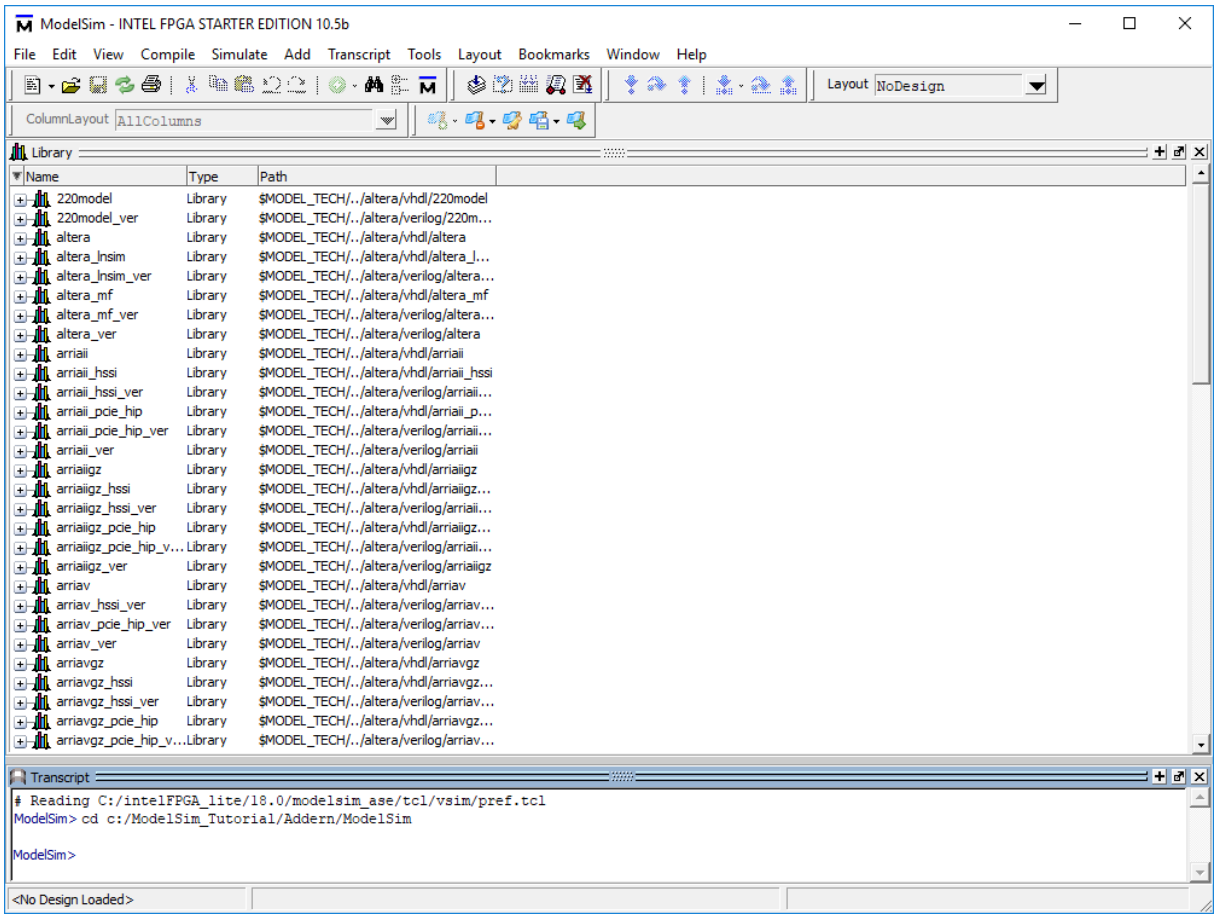

Figure 3. The *ModelSim* window.

Figure 4 shows the contents of the script *testbench.tcl*. First, the `quit` command is invoked to ensure that no simulation is already running. Then, in Line 4 the `vlib` command is executed to create a *work* design library; ModelSim stores compilation/simulation results in this working library. The VHDL compiler is invoked in Line 7 to compile the source code for the *Addern* entity, which is in the *parent* folder (*../*), and in Line 8 to compile *testbench.vht* in the current folder. The simulation is started by the `vsim` command in Line 9. It includes some simulation libraries for Intel FPGAs that may be needed by ModelSim. If the included libraries aren't required for the current design, then they will be ignored during the simulation. Line 10 in Figure 4 executes the command `do wave.do`. The *do* command is used to execute other ModelSim commands provided in a file. In this case the file *wave.do*, which will be described shortly, contains various commands that are used to configure the ModelSim waveform-display window. The final command in Figure 6 advances the simulation by a desired amount of time, which in this case is 120 ns.

To run the script, in the *Transcript* window type the command `do testbench.tcl`. ModelSim will execute the commands in this script and then update its graphical user interface to show the simulation results. The updated ModelSim window after running the *testbench.tcl* script is illustrated in Figure 5.

```
1  # stop any simulation that is currently running
2  quit -sim
3  # create the default "work" library
4  vlib work;
5
6  # compile the VHDL source code, and the testbench
7  vcom ../*.vhd
8  vcom *.vht
9  vsim work.testbench -Lf 220model -Lf altera_mf
10 do wave.do
11 run 120 ns
```

Figure 4. The *testbench.tcl* file.

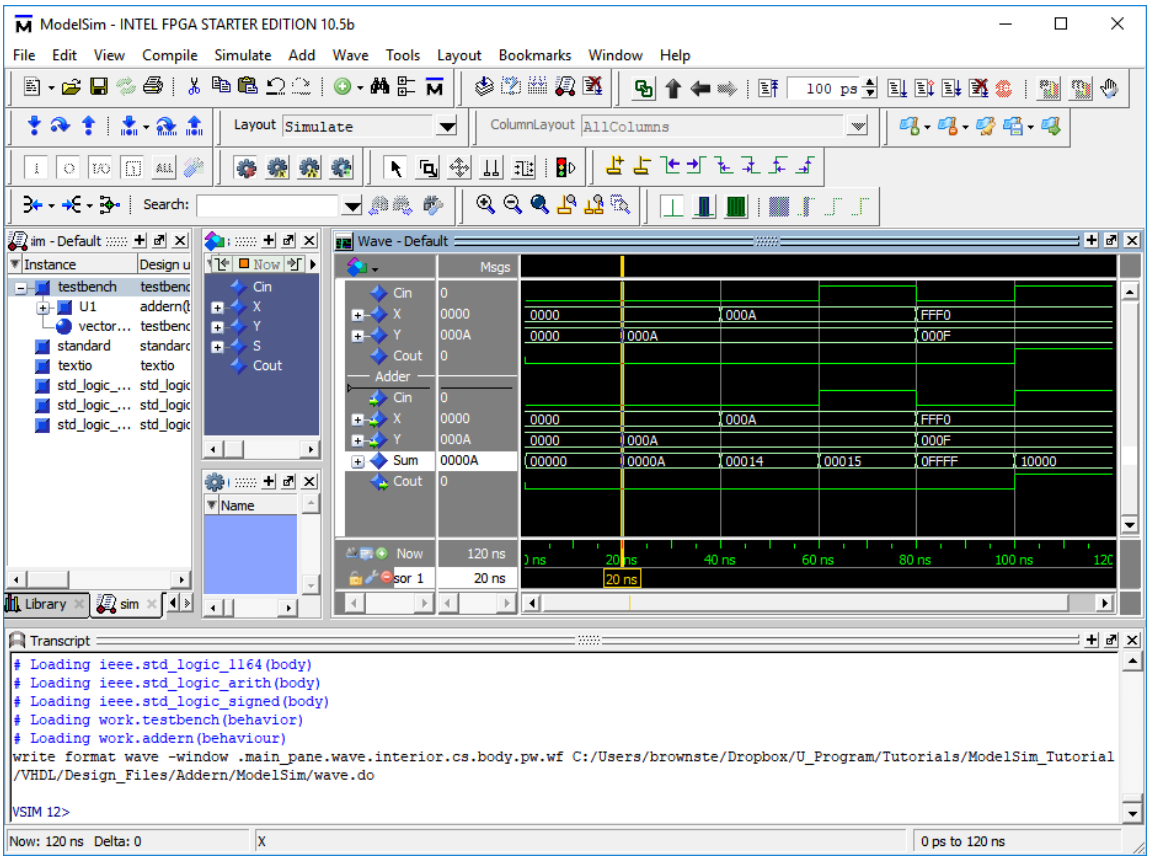

Figure 5. The updated *ModelSim* window.

The *wave.do* file used for this design example appears in Figure 6. It specifies in Lines 3 to 12 which signal waveforms should be displayed in the simulation results, and also includes a number of settings related to the display. To add or delete waveforms in the display you can manually edit the *wave.do* file using any text editor, or you can select which waveforms should be displayed by using the ModelSim graphical user interface. Referring to Figure 5, changes to the displayed waveforms can be selected by right-clicking in the waveform window. Waveforms can be added to the display by selecting a signal in the `Objects` window and then *dragging-and-dropping* that signal name into the `Wave` window. A more detailed discussion about commands available in the graphical user interface is provided in Appendix A.

Quit the ModelSim software to complete this part of the tutorial. To quit the program you can either select the `File` > `Quit` command, or type `exit` in the *Transcript* window, or just click on the `X` in the upper-right corner of the ModelSim window.

```
1  onerror {resume}
2  quietly WaveActivateNextPane {} 0
3  add wave -noupdate -label Cin /testbench/Cin
4  add wave -noupdate -label X -radix hexadecimal /testbench/X
5  add wave -noupdate -label Y -radix hexadecimal /testbench/Y
6  add wave -noupdate -label Cout /testbench/Cout
7  add wave -noupdate -divider Adder
8  add wave -noupdate -label Cin /testbench/U1/Cin
9  add wave -noupdate -label X -radix hexadecimal /testbench/U1/X
10 add wave -noupdate -label Y -radix hexadecimal /testbench/U1/Y
11 add wave -noupdate -label Sum -radix hexadecimal /testbench/U1/Sum
12 add wave -noupdate -label Cout /testbench/U1/Cout
13 TreeUpdate [SetDefaultTree]
14 WaveRestoreCursors {{Cursor 1} {20000 ps} 0}
15 quietly wave cursor active 1
16 configure wave -namecolwidth 73
17 configure wave -valuecolwidth 64
18 configure wave -justifyvalue left
19 configure wave -signalnamewidth 0
20 configure wave -snapdistance 10
21 configure wave -datasetprefix 0
22 configure wave -rowmargin 4
23 configure wave -childrowmargin 2
24 configure wave -gridoffset 0
25 configure wave -gridperiod 1
26 configure wave -griddelta 40
27 configure wave -timeline 0
28 configure wave -timelineunits ns
29 update
30 WaveRestoreZoom {0 ps} {120 ns}
```

Figure 6. The *wave.do* file.

3 Simulating a Sequential Circuit

Another ModelSim example, called *Accumulate*, is included as part of the *design files* for this tutorial. Copy the *Accumulate* example to a folder on your computer, such as *C:\ModelSim_Tutorial\Accumulate*. In the *Accumulate* folder there is a VHDL source-code file called *Accumulate.vhd* and a subfolder named *ModelSim*. The *Accumulate.vhd* file, which provides the VHDL code that we will simulate, is shown in Figure 7. It represents the logic circuit illustrated in Figure 8, which includes an adder, register, and down-counter. The purpose of this circuit is to add together, or *accumulate*, values of the input *X* for each clock cycle until the counter reaches zero.

```
ENTITY Accumulate IS
   PORT ( KEY : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
          SW : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
          CLOCK_50 : IN STD_LOGIC;
          LEDR : OUT STD_LOGIC_VECTOR(9 DOWNTO 0));
END ENTITY;

ARCHITECTURE Behaviour OF Accumulate IS
   SIGNAL X, Y, Count : STD_LOGIC_VECTOR(4 DOWNTO 0);
   SIGNAL Clock, Resetn, z : STD_LOGIC;
   SIGNAL Sum : STD_LOGIC_VECTOR(9 DOWNTO 0);
BEGIN
   Clock <= CLOCK_50;
   X <= SW(4 DOWNTO 0);
   Y <= SW(9 DOWNTO 5);
   Resetn <= KEY(0);
   PROCESS (Clock, Resetn, z)    -- define the Sum register
   BEGIN
      IF Resetn = '0' THEN
         Sum <= "0000000000";
      ELSIF Clock'EVENT AND Clock = '1' AND z = '1' THEN
         Sum <= Sum + ("00000" & X);
      END IF;
   END PROCESS;

   PROCESS (Clock, Resetn, Y, z) -- define the down-counter
      BEGIN
         IF Resetn = '0' THEN
            Count <= Y;
         ELSIF Clock'EVENT AND Clock ='1' AND z = '1' THEN
            Count <= Count - "00001";
        END IF;
   END PROCESS;
   z <= Count(0) OR Count(1) OR Count(2) OR Count(3) OR Count(4);
   LEDR <= Sum(9 DOWNTO 0);
END Behaviour;
```

Figure 7. VHDL code for the accumulator.

The *Accumulate* entity in Figure 7 has ports *KEY*, *CLOCK_50*, *SW*, and *LEDR* because it is intended to be implemented on a DE-series board that features an Intel FPGA, such as the *DE1-SoC* board. After simulating the VHDL code to verify its correct operation, you may wish to compile it using the Quartus Prime CAD tools and then download and test the resulting circuit on a board.

A *testbench.vht* file for the accumulator design under test (DUT) is given in Figure 9. Three signals, *KEY*, *CLOCK_50*, and *SW* are declared to provide inputs to the DUT, as well as a signal *LEDR* for connecting to the DUT outputs. The *Accumulate* entity is instantiated in Line 17. It is useful to define a periodic signal that can be used as a clock input for the *Accumulate* sequential circuit. We could manually define some number of cycles for such a signal in a process, but this method would be awkward. Instead, Lines 19 to 25 in Figure 9 show how to make a periodic signal by specifying just one period. This process gets executed repeatedly, because it does not end with a sepatate `WAIT` statement. Thus, the *CLOCK_50* signal is inverted every 10 ns in simulation time to create a 50 MHz periodic waveform. This clock process is executed *concurrently* by the Simulator along with the process in Lines 27 to 36. This process sets $KEY_0 = 0$ and $SW = 0$ at the start of the simulation, which allows the *Sum* in the accumulator to be cleared. At 20 ns in simulation time SW_{9-5} is set to 10, so that this value can be loaded into the counter. Finally, at 40 ns in simulation time KEY_0 is set to 1 and SW_{4-0} is set to 30, so that this value can be *accumulated* for each clock cycle until the counter reaches 0.

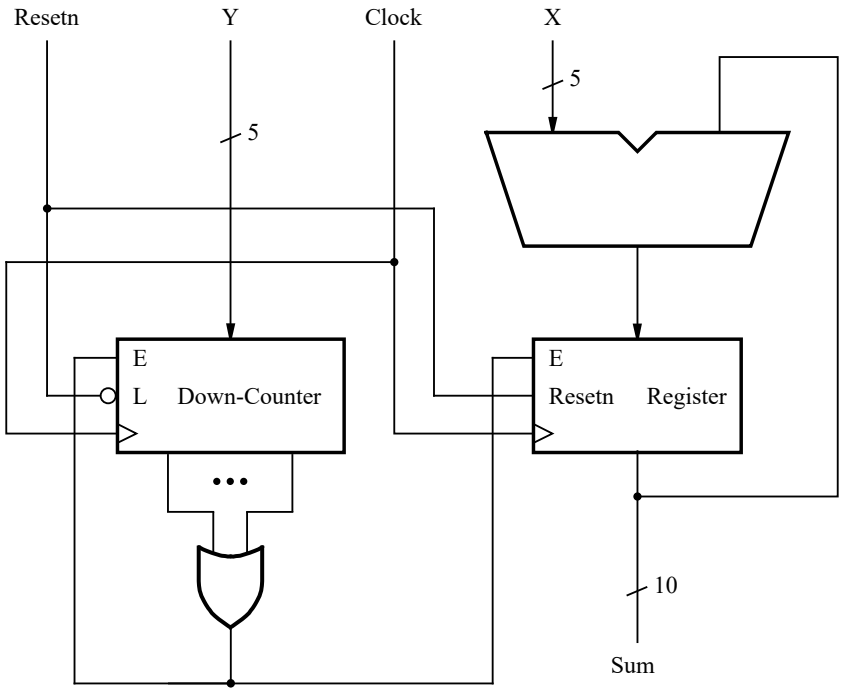

Figure 8. The accumulator circuit.

Reopen the *ModelSim* software to get to the window in Figure 3. Click on the *Transcript* window at the bottom of the figure and then use the `cd` command to navigate to the ModelSim folder for the accumulator. For example, in our case we would type `cd C:/ModelSim_Tutorial/Accumulate/ModelSim`. Then, in the *Transcript* window type the command `do testbench.tcl` as you did for the previous example. The *testbench.tcl* script for this example is identical to the one shown in Figure 4, except that the last line specifies `run 300 ns`.

```
1  ENTITY testbench IS
2  END testbench;
3
4  ARCHITECTURE Behavior OF testbench IS
5      COMPONENT Accumulate
6         PORT ( KEY : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
7                SW : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
8                CLOCK_50 : IN STD_LOGIC;
9                LEDR : OUT STD_LOGIC_VECTOR(9 DOWNTO 0));
10     END COMPONENT;
11
12     SIGNAL CLOCK_50 : STD_LOGIC;
13     SIGNAL KEY : STD_LOGIC_VECTOR(0 DOWNTO 0);
14     SIGNAL SW : STD_LOGIC_VECTOR(9 DOWNTO 0);
15     SIGNAL LEDR : STD_LOGIC_VECTOR(9 DOWNTO 0);
16 BEGIN
17 U1: Accumulate PORT MAP (KEY, SW, CLOCK_50, LEDR);
18
19     clock_process: PROCESS
20     BEGIN
21         CLOCK_50 <= '0';
22         WAIT FOR 10 ns;
23         CLOCK_50 <= '1';
24         WAIT FOR 10 ns;
25     END PROCESS;
26
27     vectors: PROCESS
28     BEGIN
29         KEY(0) <= '0'; SW <= "0000000000";
30         WAIT FOR 20 ns;
31         SW(9 DOWNTO 5) <= "01010";
32         WAIT FOR 20 ns;
33         SW(4 DOWNTO 0) <= "11110";
34         KEY(0) <= '1';
35         WAIT;
36     END PROCESS;
37 END;
```

Figure 9. The VHDL testbench code for the sequential circuit.

The simulation results for our sequential circuit, which display the waveforms selected in its *wave.do* file, appear in Figure 10. In this figure the *SW* and *LEDR* signals are displayed in hexadecimal, while *X*, *Sum*, *Y*, and *Count* are displayed as unsigned (decimal) values. The *Sum* is cleared by the clock edge at 10 ns, and the *Count* is initialized to 10 at 30 ns. Starting with the clock edge at 50 ns the value $X = 30$ is accumulated until the counter reaches 0.

Quit the ModelSim software to complete this part of the tutorial.

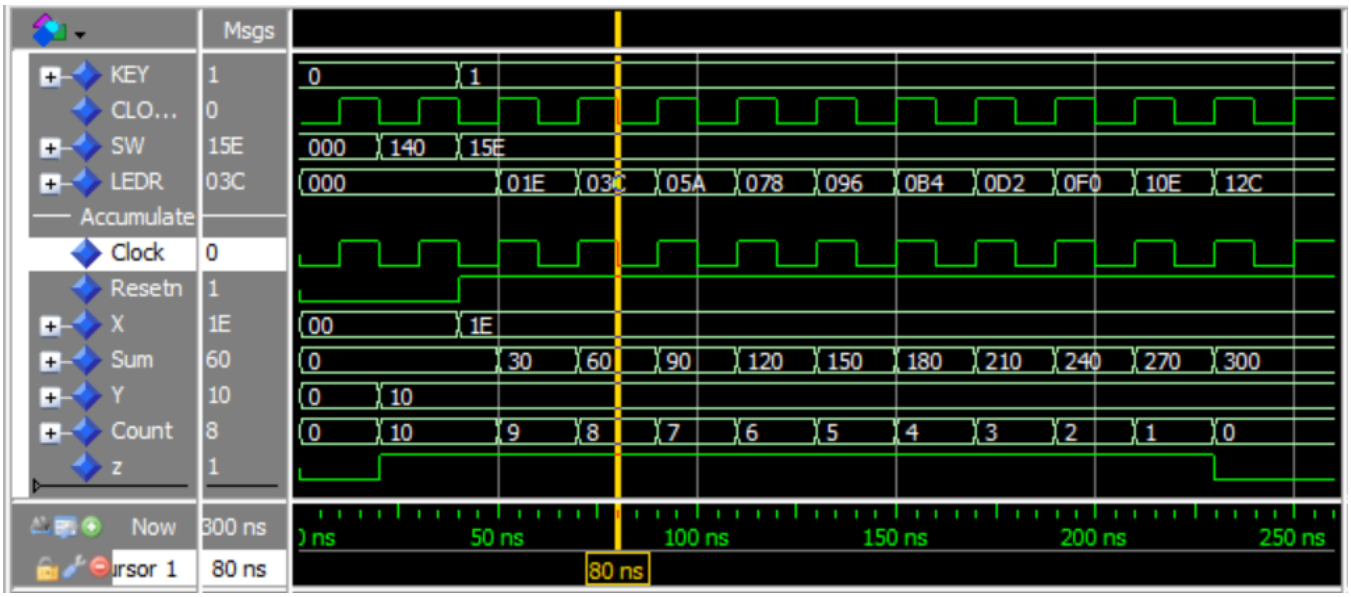

Figure 10. The simulation results for our sequential circuit.

4 Simulating a Circuit that Includes a Memory Module

The *design files* archive provided along with this tutorial includes a ModelSim example called *Display*. It shows how to instantiate a memory module in VHDL code, and how to initialize the stored contents of the memory in a ModelSim simulation. Copy the *Display* files to a folder on your computer, such as *C:\ModelSim_Tutorial\Display*. In the *Display* folder there is a file called *Display.vhd* that provides the VHDL code that we will simulate, and a subfolder named *ModelSim*.

Figure 11 shows the VHDL code for *Display.vhd*. Its ports are named *KEY*, *SW*, *HEX0*, and *LEDR* because the entity is intended to be implemented on a DE-series board that features an Intel FPGA, such as the *DE1-SoC* board. After simulating the VHDL code to verify its correct operation, you may wish to compile it using the Quartus Prime CAD tools and then download and test the resulting circuit on a board.

Figure 12*a* gives a logic circuit that corresponds to the code in Figure 11. The circuit contains a counter that is used to read the contents of successive addresses from a memory. This memory provides codes in ASCII format for some upper- and lower-case letters, which are provided as inputs to a decoder entity. The counter and memory module have a common clock signal, and the counter has a synchronous clear input. Each successive clock cycle advances the counter and reads a new ASCII code from the memory. Since the counter is three-bits wide, only the first eight locations in the memory are read (the upper two address bits on the memory are set to 00), and they provide the ASCII codes for letters A, b, C, d, E, F, g, and h. The decoder produces an appropriate bit-pattern to render each letter on a seven-segment display. The memory used in the logic circuit is depicted in part *b* of Figure 12. It is a 32 × 8 synchronous read-only memory (ROM), which has a register for holding address values. The memory is initialized with the contents of the file *inst_mem.mif*, which is illustrated in Figure 13. This file contains the ASCII codes for the eight letters displayed by the circuit.

```
ENTITY display IS
    PORT ( KEY : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
           SW : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
           HEX0 : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
           LEDR : OUT STD_LOGIC_VECTOR(9 DOWNTO 0) );
END ENTITY;

ARCHITECTURE Behavior OF display IS
    COMPONENT inst_mem
        PORT ( address    : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
               clock      : IN STD_LOGIC ;
               q          : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
    END COMPONENT;
    COMPONENT count3
        PORT ( Resetn, Clock    : IN   STD_LOGIC;
               Q                : OUT  STD_LOGIC_VECTOR(2 DOWNTO 0));
    END COMPONENT;

    CONSTANT A : STD_LOGIC_VECTOR(7 DOWNTO 0) := x"41";
    CONSTANT b : STD_LOGIC_VECTOR(7 DOWNTO 0) := x"62";
    CONSTANT C : STD_LOGIC_VECTOR(7 DOWNTO 0) := x"43";
    CONSTANT d : STD_LOGIC_VECTOR(7 DOWNTO 0) := x"64";
    CONSTANT E : STD_LOGIC_VECTOR(7 DOWNTO 0) := x"45";
    CONSTANT F : STD_LOGIC_VECTOR(7 DOWNTO 0) := x"46";
    CONSTANT g : STD_LOGIC_VECTOR(7 DOWNTO 0) := x"67";
    CONSTANT h : STD_LOGIC_VECTOR(7 DOWNTO 0) := x"68";
    SIGNAL Resetn, Clock : STD_LOGIC;
    SIGNAL Count : STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL Address : STD_LOGIC_VECTOR(4 DOWNTO 0);
    SIGNAL char : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
    Resetn <= SW(0);
    Clock <= KEY(0);

    U1: count3 PORT MAP (Resetn, Clock, Count);
    Address <= "00" & Count;
    U2: inst_mem PORT MAP (Address, Clock, char);
    LEDR <= "00" & char;

    HEX0 <= "0001000" WHEN char = A ELSE
            "0000011" WHEN char = b ELSE
            "1000110" WHEN char = C ELSE
            "0100001" WHEN char = d ELSE
            "0000110" WHEN char = E ELSE
            "0001110" WHEN char = F ELSE
            "0010000" WHEN char = g ELSE
            "0001011" WHEN char = h ELSE
            "1111111";
END Behavior;
```

Figure 11. VHDL code for the display circuit.

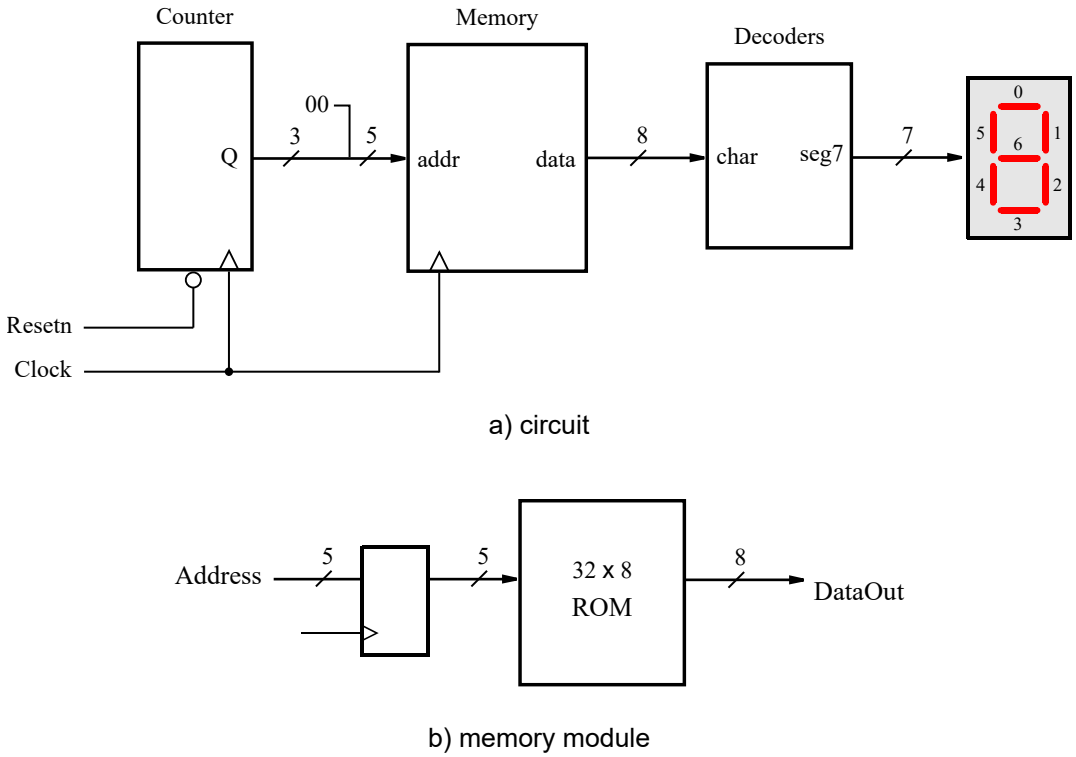

Figure 12. The circuit for the memory example.

```
DEPTH = 32;
WIDTH = 8;
ADDRESS_RADIX = HEX;
DATA_RADIX = DEC;
CONTENT
BEGIN
        00  : 65;        % A %
        01  : 98;        % b %
        02  : 67;        % C %
        03  : 100;       % d %
        04  : 69;        % E %
        05  : 70;        % F %
        06  : 103;       % g %
        07  : 104;       % h %
END;
```

Figure 13. The *inst_mem.mif* memory initialization file.

A *testbench.vht* file for the *Display* design under test (DUT) is given in Figure 14. Two signals, *KEY* and *SW* are declared to provide inputs to the DUT, as well as two signals *HEX0* and *LEDR* for connecting to the DUT outputs. The DUT is instantiated in Line 17 of the testbench. It uses a process to create a clock waveform with a 20 ns period on the *KEY* signal. Another process is used to perform a synchronous reset of the counter with the SW signal.

```
1  ENTITY testbench IS
2  END testbench;
3
4  ARCHITECTURE Behavior OF testbench IS
5      COMPONENT display
6          PORT ( KEY   : IN   STD_LOGIC_VECTOR(0 DOWNTO 0);
7                 SW    : IN   STD_LOGIC_VECTOR(0 DOWNTO 0);
8                 HEX0  : OUT  STD_LOGIC_VECTOR(6 DOWNTO 0);
9                 LEDR  : OUT  STD_LOGIC_VECTOR(9 DOWNTO 0));
10     END COMPONENT;
11
12     SIGNAL KEY : STD_LOGIC_VECTOR(0 DOWNTO 0);
13     SIGNAL SW : STD_LOGIC_VECTOR(0 DOWNTO 0);
14     SIGNAL HEX0 : STD_LOGIC_VECTOR(6 DOWNTO 0);
15     SIGNAL LEDR : STD_LOGIC_VECTOR(9 DOWNTO 0);
16 BEGIN
17     U1: display PORT MAP (KEY, SW, HEX0, LEDR);
18
19     clock_process: PROCESS
20     BEGIN
21         KEY(0) <= '0';
22         WAIT FOR 10 ns;
23         KEY(0) <= '1';
24         WAIT FOR 10 ns;
25     END PROCESS;
26
27     vectors: PROCESS
28     BEGIN
29         SW(0) <= '0';        -- Resetn = 0
30         WAIT FOR 20 ns;
31         SW(0) <= '1';        -- Resetn = 1
32         WAIT;
33     END PROCESS;
34 END;
```

Figure 14. Testbench code for the memory example.

Figure 15 shows the contents of the script *testbench.tcl* for this example. It has the same structure as the file shown in Figure 4, with two exceptions. First, in Lines 4 to 7 the script checks whether there exists an *inst_mem.mif* memory initialization file in the parent folder; if so, it copies this file to the ModelSim folder so that the memory will be properly initialized during simulation. Second, in Lines 9 to 11 the script checks if an "empty black box" file, which can optionally be created by the Quartus software, exists in the parent directory. If so, the script deletes this file, because it would cause an error during simulation.

```
quit -sim

# if simulating with a MIF file, copy it. Assumes inst_mem.mif
if {[file exists ../inst_mem.mif]} {
   file delete inst_mem.mif
   file copy ../inst_mem.mif .
}
# if Quartus generated an "empty black box" file, delete it
if {[file exists ../inst_mem_bb.vhd]} {
   file delete ../inst_mem_bb.vhd
}
# create the default "work" library
vlib work;

# compile the VHDL source code in the parent folder
vcom ../*.vhd
# compile the VHDL code of the testbench
vcom *.vht
# start the Simulator, including some libraries
vsim work.testbench -Lf 220model -Lf altera_mf
# show waveforms specified in wave.do
do wave.do
run 180 ns
```

Figure 15. The *testbench.tcl* file.

Reopen the *ModelSim* software to get to the window in Figure 3. In the *Transcript* window use the `cd` command to navigate to the ModelSim folder for the this part of the tutorial. For example, in our case we would type `cd C:/ModelSim_Tutorial/Display/ModelSim`. In the *Transcript* window type the command `do testbench.tcl` to run the script in Figure 15. The simulation results for our circuit, which display the waveforms selected in its *wave.do* file, appear in Figure 16.It shows in the *char* waveform, displayed using the "radix" ASCII, the values read from each address in memory. These values are also shown in hexadecimal in the *LEDR* waveform, and the decoder outputs are shown in binary in the *HEX0* waveform.

5 Setting up a ModelSim Simulation

The files described above can be used as a starting point for setting up your own ModelSim simulation, as follows. In the folder that contains your VHDL source-code to be simulated, make a subfolder named *ModelSim*. Copy into this subfolder the files *testbench.vht*, *testbench.tcl*, and *wave.do* from one of the examples above. Then, modify *testbench.vht* to instantiate your top-level VHDL entity and create whatever waveforms are needed. You can use an identical *testbench.tcl* script as shown above, except that you might want to specify a different amount of simulation time for the `run` command. Since the *.vht* and *.tcl* files are ASCII text files, you can edit them with any text editor of your choosing (or the text editor provided within ModelSim). Finally, modify the *wave.do* file to choose the waveforms that should be displayed. You can change the *wave.do* file manually by editing it with a text editor, or you can make use of the commands available in the ModelSim graphical user interface, as discussed in Appendix A.

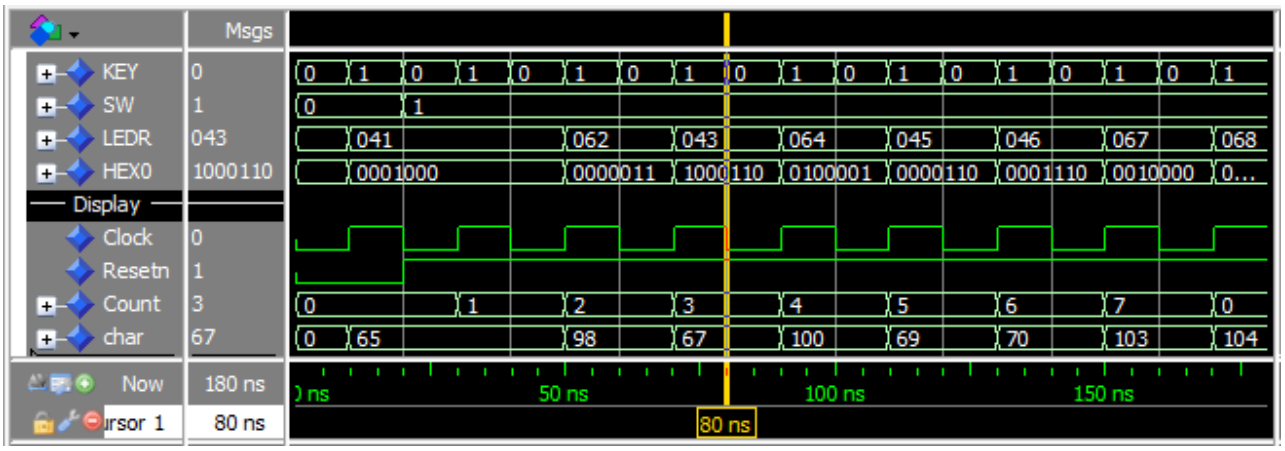

Figure 16. The simulation results for our memory example.

Appendix A: Using the ModelSim Graphical User Interface

This appendix illustrates some of the features available in the ModelSim graphical user interface for displaying waveforms. We will show how to add waveforms to the ModelSim window, and how to change the properties of a waveform, such as its displayed name and number radix.

As an example we will show how waveforms can be added to the ModelSim display for the *accumulator* circuit from the previous section. Figure 17 displays the ModelSim window for this circuit before any waveforms have been selected.

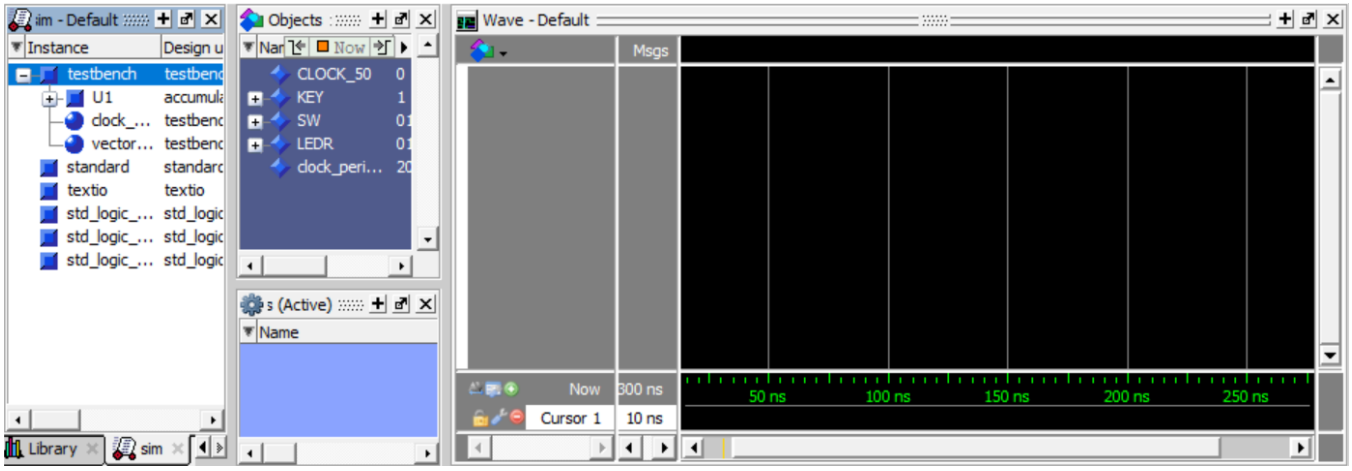

Figure 17. The *ModelSim* waveform display.

In Figure 18 we have selected a waveform, as follows. First, we clicked on the `testbench` entity in the upper left part of the display. As a result of this action the signals that exist in the selected entity are listed in the `Objects` pane (the area with the dark blue background). In this list we then used the left mouse button to *drag-and-drop* the

KEY signal name from the Objects list into the Wave window. Then, as illustrated in the figure, we right-clicked on the name of the signal in the Wave display, which is /testbench/KEY, and then clicked on Properties to open the window in Figure 19.

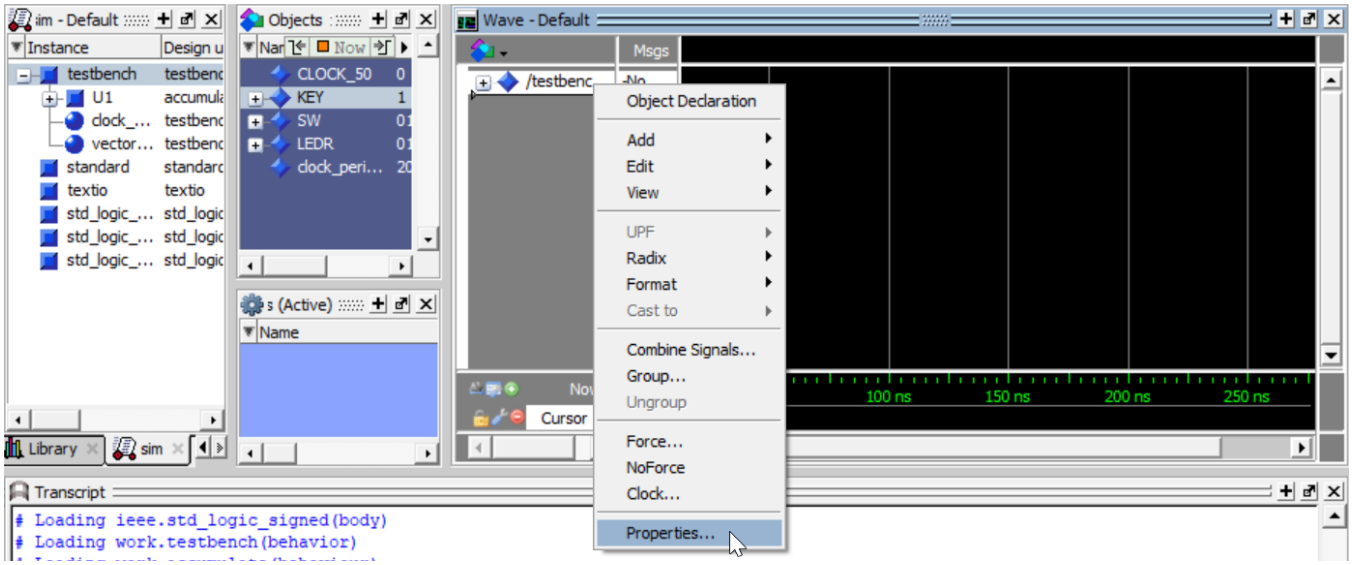

Figure 18. Adding a waveform from the testbench entity.

In Figure 19 we assigned the name KEY to the waveform, clicked Apply and then closed this dialogue. We then used the same drag-and-drop mechanism to add the signals CLOCK_50, SW, and LEDR from the testbench entity to the Wave window, and set convenient display names for these waveforms. The updated Wave window is shown in Figure 20.

Next, we wish to add signals from the Accumulate entity to the Wave window. But first we can add a *divider*, as a visual aid that separates the testbench signals and the Accumulate entity signals. A divider can be added by right-clicking on the Wave window, as indicated in the Figure 21, clicking on Add in the pop-up menu, and then selecting New Divider to open the window in Figure 22.

We assigned Accumulate as the divider name in Figure 22, and then closed this dialogue. The Wave window now appears as shown in Figure 23.

To add signals from the Accumulate entity to the Wave window we need to click on the U1 instance name of the Accumulate model, as indicated on the left-hand side of Figure 24. The signals available in this entity are then listed in the Objects pane. To obtain the display in the figure, we used the drag-and-drop mechanism, and the Wave Properties dialogue, described previously, to add the Clock, Resetn, X, Sum, Y, Count, and z signals to the Wave window.

We now wish to perform a simulation of our testbench so that waveforms will be generated and shown for our selected signals. However, it is *critical* to first *save* the selections that have been made in the Wave display to the *wave.do* file. If you run a simulation *without* first performing a save to the *wave.do* file, then all changes made to the Wave window will be discarded and lost!

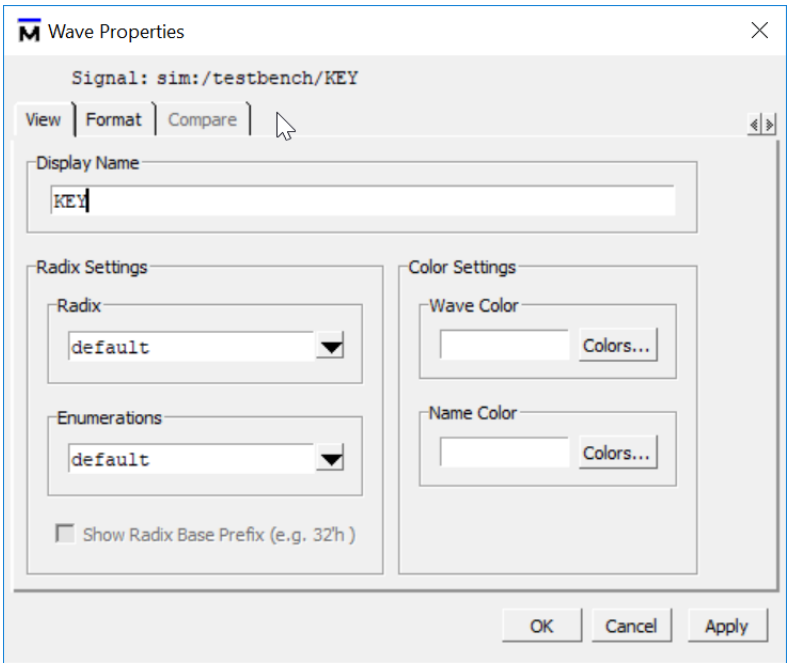

Figure 19. Specifying a display name for a waveform.

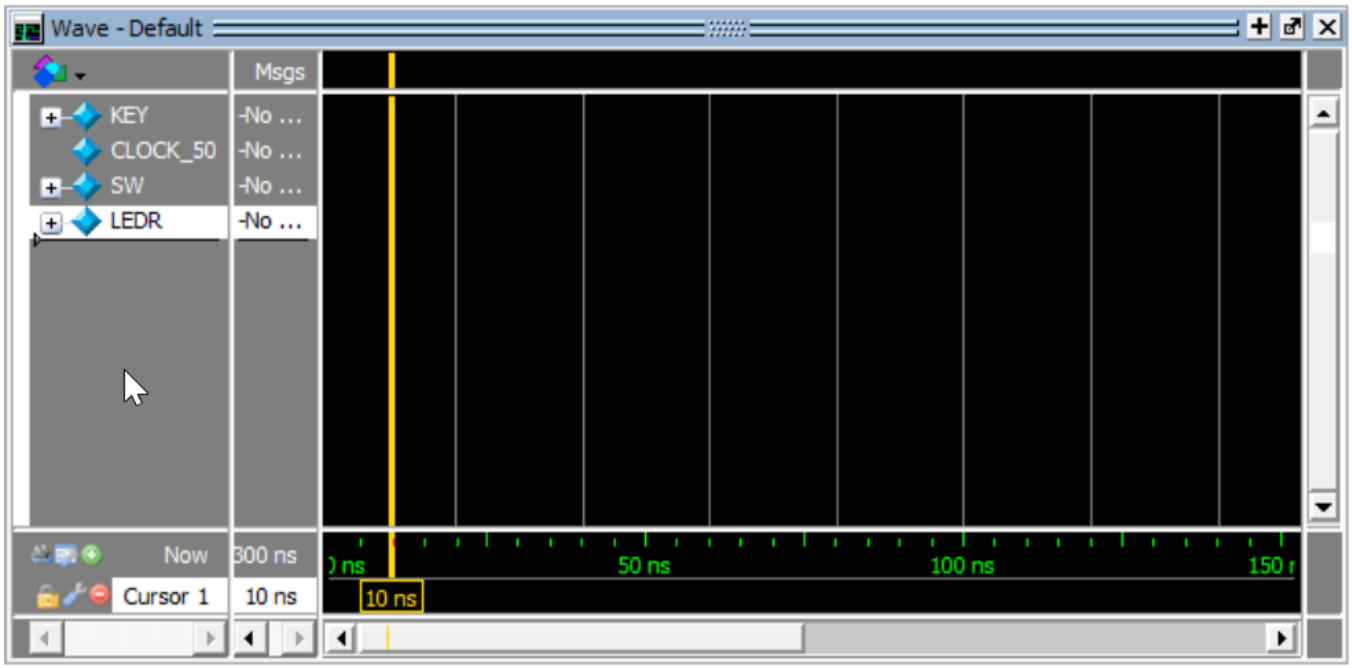

Figure 20. The waveform display after adding more *testbench* signals.

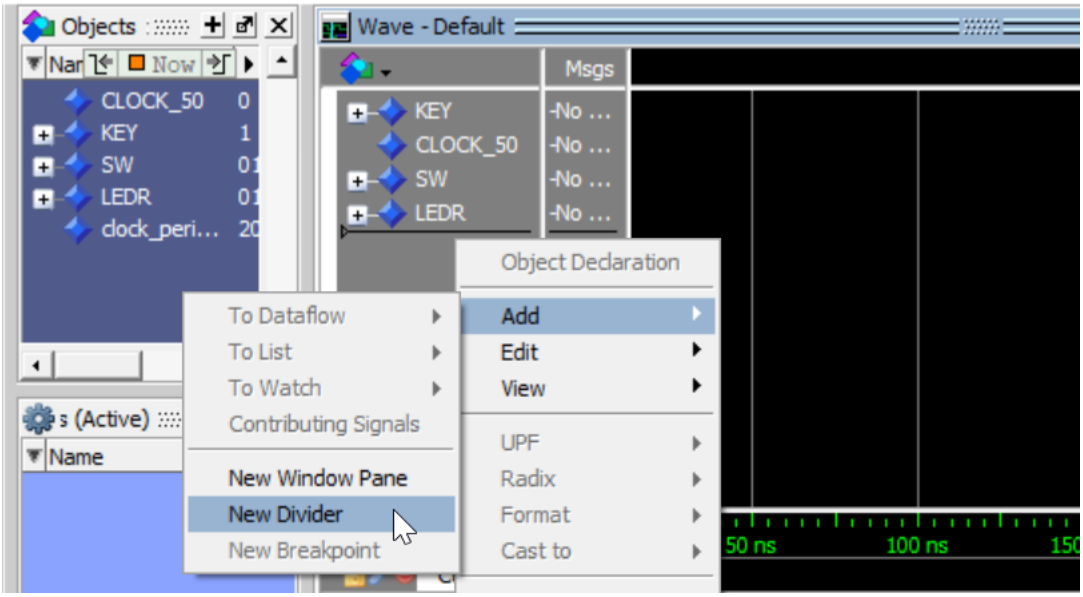

Figure 21. Adding a divider to the waveform display.

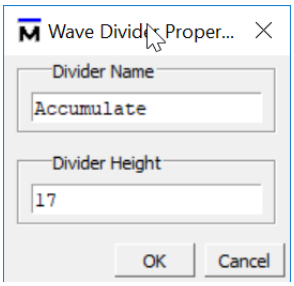

Figure 22. Assigning a name to the divider.

The command `File > Save Format` opens the dialogue shown in Figure 25. After clicking `OK` and then overwriting the *wave.do* file, the testbench simulation can be executed by typing the command `do testbench.tcl`. The resulting waveform display is illustrated in Figure 26. In this figure we right-clicked on the `Wave` window and selected `Zoom Range` to open the dialogue in Figure 27. As indicated in the figure, we select a time range from 0 to 300 ns for the Wave display.

To make it easier to see the values of signals in the `Wave` window, you can select radices other than binary, which is the default. For example, in Figure 28 we right-clicked on the `SW` signal, clicked on `Radix`, and then selected `Hexadecimal`. After setting the radix to hexadecimal for several additional signals, the final `Wave` display appears as illustrated in Figure 29. As mentioned earlier, changes to the waveforms have to be saved by using the `File > Save Format` command.

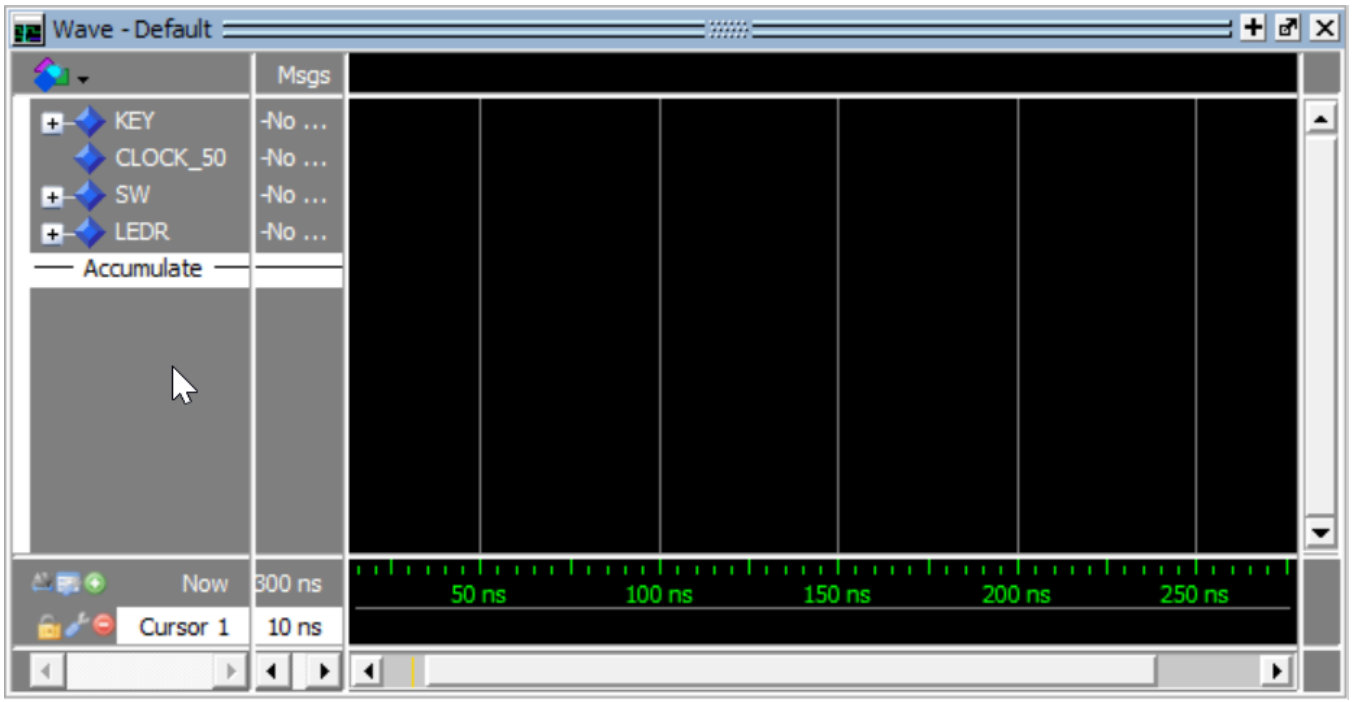

Figure 23. The waveform display after adding the `Accumulate` divider.

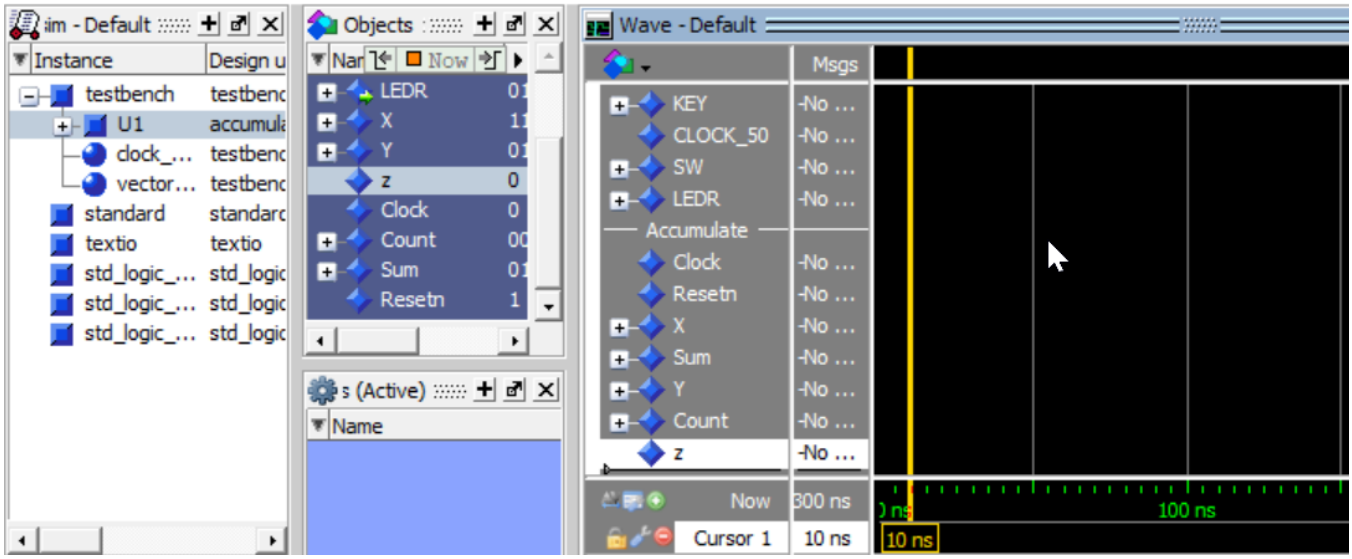

Figure 24. The waveform display after adding signals from the accumulate entity.

This tutorial has described only a subset of the commands that are provided in the ModelSim graphical user interface. Although a discussion of other available commands is beyond the scope of this tutorial, a number of more detailed ModelSim tutorials can be found by searching for them on the Internet.

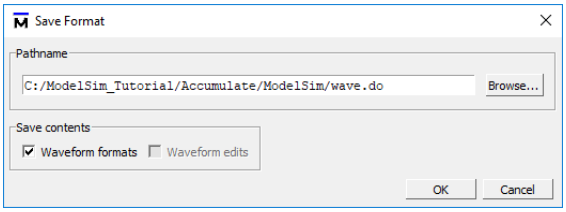

Figure 25. The `Save Format` dialogue.

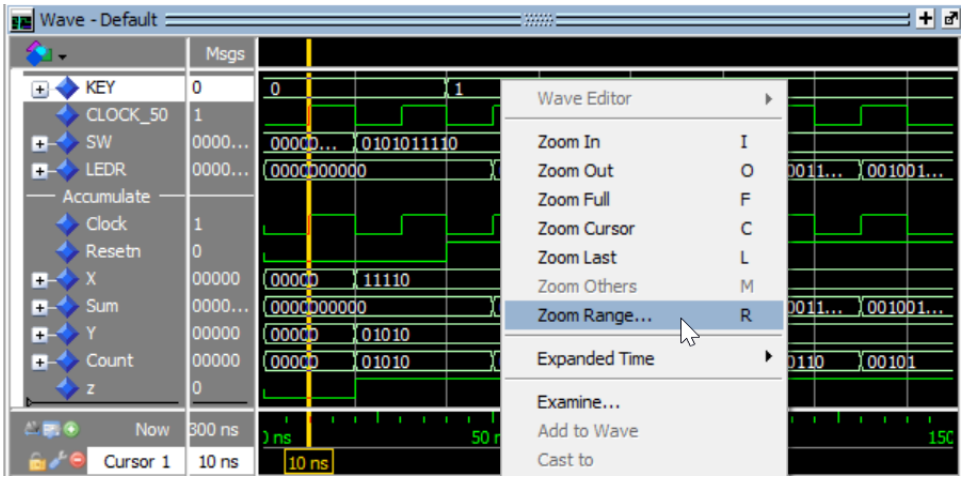

Figure 26. The display after running the simulation; changing the zoom range.

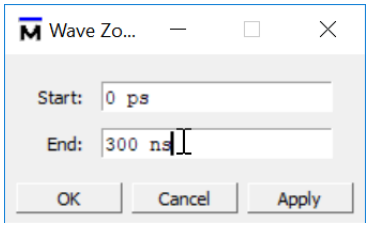

Figure 27. Inputting the zoom range.

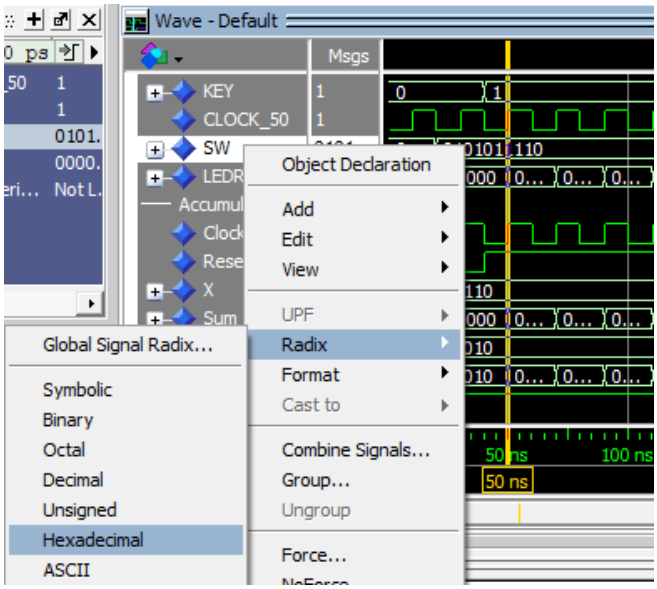

Figure 28. Setting the radix for a waveform.

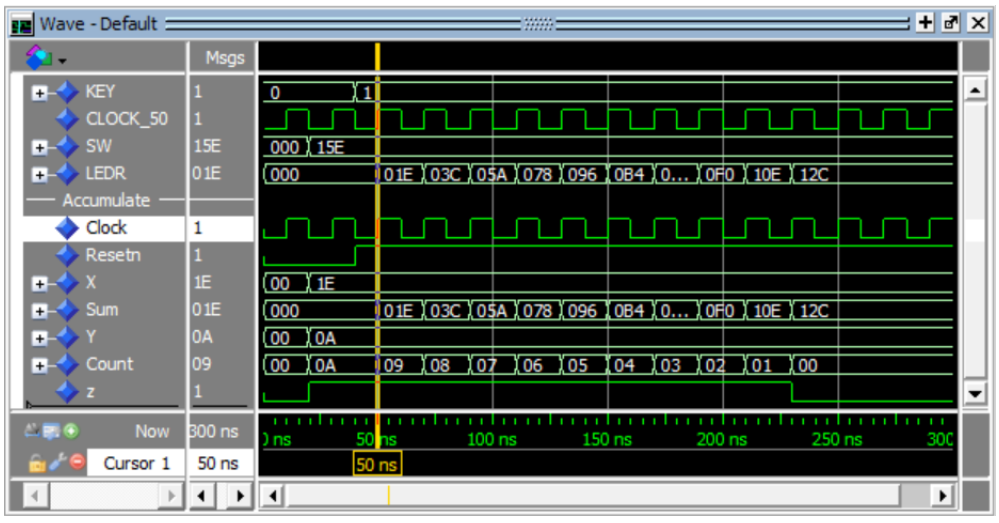

Figure 29. The final waveform display.