Using the ModelSim-Intel FPGA Simulator with Verilog Testbenches

For Quartus® Prime 18.0

1 Introduction

This tutorial introduces the simulation of Verilog 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 not described here, but can be found 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 Verilog source-code file called *Addern.v* and a subfolder named *ModelSim*. The *Addern.v* file, shown in Figure 1, is the Verilog 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, *Sum* and *Cout*.

```
// A multi-bit adder
module Addern (Cin, X, Y, Sum, Cout);
    parameter n = 16;
    input Cin;
    input [n-1:0] X, Y;
    output [n-1:0] Sum;
    output Cout;

    assign {Cout, Sum} = X + Y + Cin;
endmodule
```

Figure 1. Verilog 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.v*, *testbench.tcl*, and *wave.do*.

The *testbench.v* file is a style of Verilog code known as a *testbench*. The purpose of a testbench is to *instantiate* a Verilog module 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). The first statement in the Verilog testbench, illustrated in Figure 2, is called a *timescale compiler directive*. Its first argument sets the *units* of simulation time to 1 nanosecond. The user can specify values for the inputs to the DUT in terms of these time units, as we will illustrate shortly. The second argument sets the *resolution* of the simulation to 1 picosecond. This parameter specifies the granularity of time for which ModelSim evaluates signal values during a simulation. We will use these parameters for all of our simulations in this tutorial.

Line 2 is the start of the testbench module, which has no inputs or outputs. In Lines 4 and 5 we declare the signals *Cin*, *X*, and *Y*, which will be used as the inputs to the DUT. Values will be assigned at various simulation times to these signals in the testbench code. Verilog syntax requires that these signals have the type *reg* as given in the figure. When a value is specified in the testbench code for a signal with the reg type, the signal maintains this value until

it is changed again in the testbench. Lines 7 and 8 in Figure 2 declare the signals *Sum* and *Cout*, which will be connected to the outputs of the DUT. These signals, whose values are determined by the behavior of the DUT during the simulation, have to be declared with the type *wire* as given in the code.

The *Addern* module, our design under test, is instantiated in Line 11 of the testbench. The *Addern* inputs and outputs are attached to the signals declared previously in Lines 4 to 8 of the testbench.

Lines 14 to 22 provide an *initial block*. It is is used to assign values to the reg signals that provide inputs to the DUT. The initial block starts executing at the beginning of simulation time, so that line 16 initializes the signals *X*, *Y*, and *Cin* to 0 at simulation time 0. Line 17 specifies that after 20 simulation time *units* the value of the input *Y* changes to 10. Since the unit of simulation time is set to 1 ns by the timescale directive in Line 1, this means that *Y* changes to the value 10 at 20 ns in simulation time. Line 18 specifies that after another 20 ns, meaning at 40 ns in simulation time, input *X* changes to 10. The rest of the initial block specifies various values for the adder inputs at 20 ns time increments.

```
1  `timescale 1ns / 1ps
2  module testbench ( );
3      // reg signals provide inputs to the design under test
4      reg Cin;
5      reg [15:0] X, Y;
6      // wire signals are used for outputs
7      wire [15:0] Sum;
8      wire Cout;
9
10     // instantiate the design under test
11     Addern U1 (Cin, X, Y, Sum, Cout);
12
13     // assign signal values at various simulation times
14     initial
15     begin
16         X <= 0; Y <= 0; Cin <= 0;
17         #20 X <= 0; Y <= 10; Cin <= 0;
18         #20 X <= 10; Y <= 10; Cin <= 0;
19         #20 X <= 10; Y <= 10; Cin <= 1;
20         #20 X <= 16'hFFF0; Y <= 16'hF; Cin <= 0;
21         #20 X <= 16'hFFF0; Y <= 16'hF; Cin <= 1;
22     end // initial
23 endmodule
```

Figure 2. The Verilog 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*.

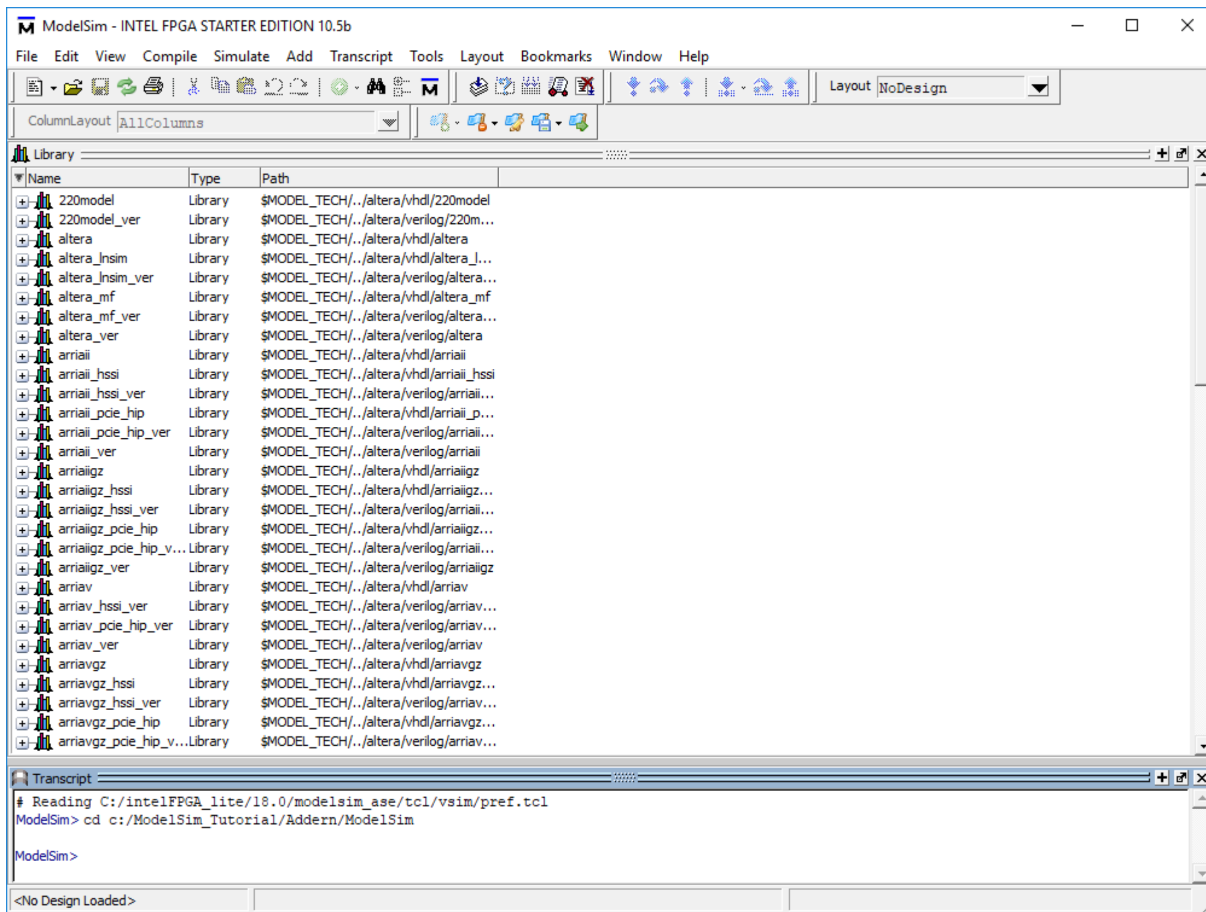


Figure 3. The *ModelSim* window.

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 Verilog compiler is invoked in Line 7 to compile the source code for the *Addern* module, which is in the *parent* folder (../), and in Line 9 to compile *testbench.v* in the current folder. The simulation is started by the `vsim` command in Line 11. 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 13 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 Verilog source code in the parent folder
7  vlog ../*.v
8  # compile the Verilog code of the testbench
9  vlog *.v
10 # start the Simulator, including some libraries
11 vsim work.testbench -Lf 220model -Lf altera_mf_ver -Lf
       verilog
12 # show waveforms specified in wave.do
13 do wave.do
14 # advance the simulation the desired amount of time
15 run 120 ns
```

Figure 4. The *testbench.tcl* file.

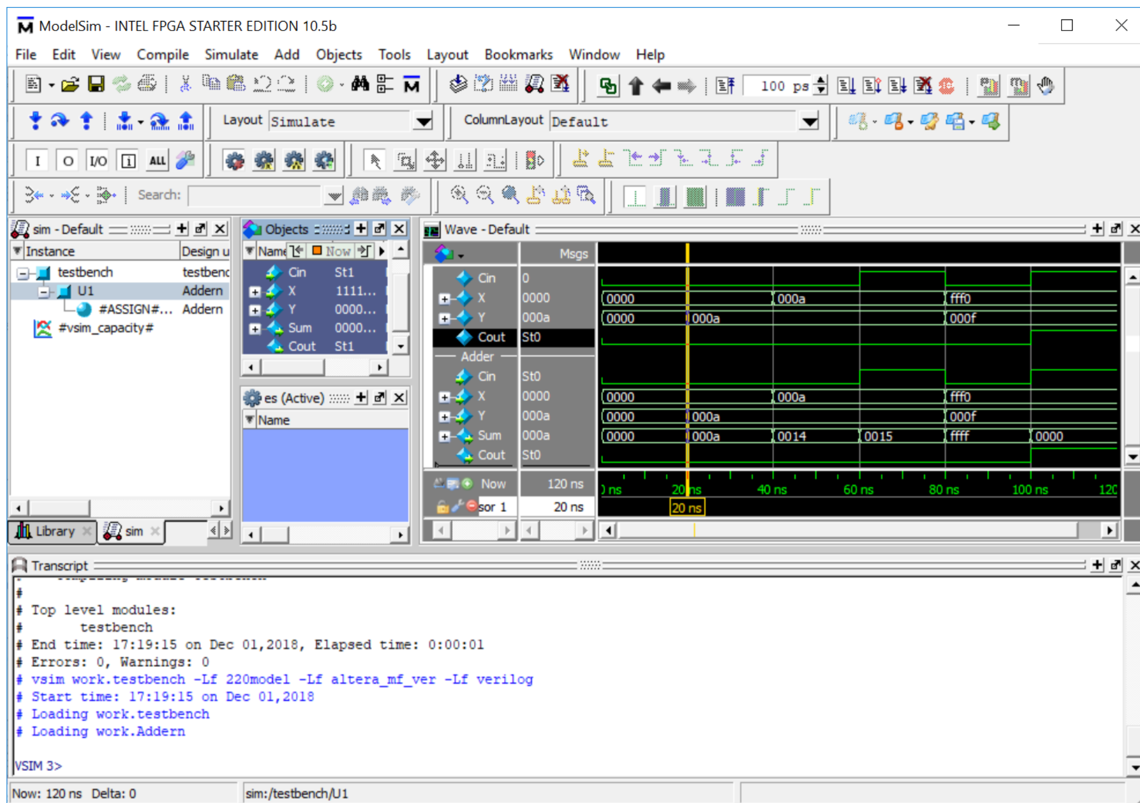


Figure 5. The updated *ModelSim* window.

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 Verilog source-code file called *Accumulate.v* and a subfolder named *ModelSim*. The *Accumulate.v* file, which provides the Verilog 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.

The *Accumulate* module in Figure 7 has ports *KEY*, *CLOCK_50*, *SW*, and *LEDR* because the module is intended to be implemented on a DE-series board that features an Intel FPGA, such as the *DE1-SoC* board. After simulating the Verilog 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.

```
module Accumulate (KEY, CLOCK_50, SW, LEDR);
    input [0:0] KEY;
    input CLOCK_50;
    input [9:0] SW;
    output [9:0] LEDR;

    wire Clock, Resetn, z;
    wire [4:0] X, Y;
    reg [9:0] Sum;
    reg [4:0] Count;

    assign Clock = CLOCK_50;
    assign Resetn = KEY[0];
    assign X = SW[4:0];
    assign Y = SW[9:5];

    always @(posedge Clock)
        if (Resetn  == 1'b0)     // synchronous clear of the accumulator
            Sum <= 0;
        else if (z == 1'b1)
            Sum <= Sum + X;

    always @(posedge Clock)
        if (Resetn  == 1'b0)     // synchronous load of the counter
            Count <= Y;
        else if (z == 1'b1)
            Count <= Count - 1'b1;

    assign z = | Count;
    assign LEDR = Sum;
endmodule
```

Figure 7. Verilog code for the accumulator.

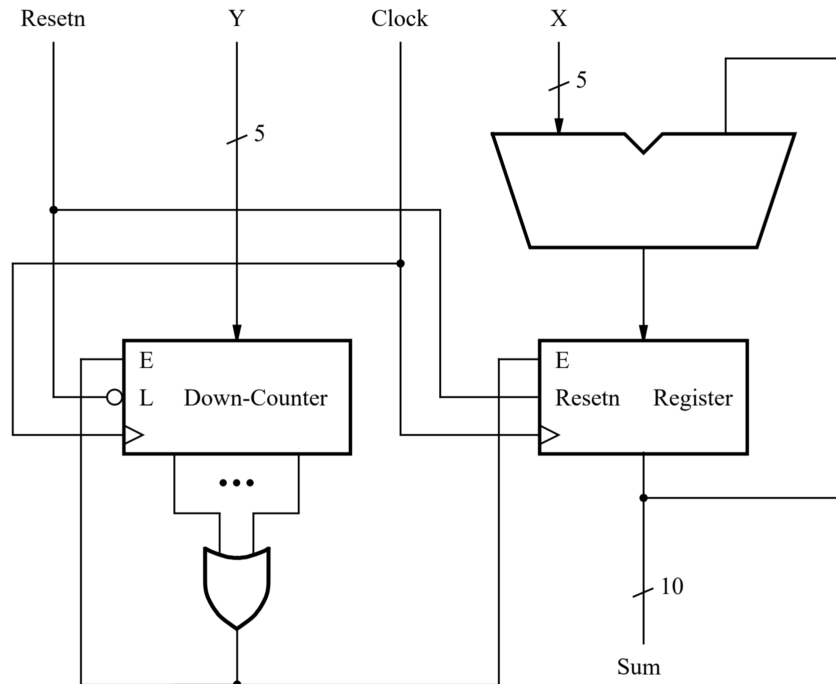


Figure 8. The accumulator circuit.

A *testbench.v* file for the accumulator design under test (DUT) is given in Figure 9. Three reg-type signals, *KEY*, *CLOCK_50*, and *SW* are declared to provide inputs to the DUT, as well as a wire-type signal *LEDR* for connecting to the DUT outputs. The *Accumulate* module is instantiated in Line 12.

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 by using an *initial* block in the testbench, but this method would be awkward. Instead, Lines 15 to 16 in Figure 9 show how a clock signal can be easily specified by using an *always* block. Unlike an initial block, in which the statements are executed just once in sequential order, the statements in an always block are executed repeatedly. Thus, Line 16 in the always block inverts the *CLOCK_50* signal every 10 ns in simulation time and creates a 50 MHz periodic waveform. The always block is executed *concurrently* by the Simulator along with the initial block in Lines 18 to 26. This code first initializes the *CLOCK_50* signal to 0 at the start of the simulation. This action is necessary because the default value of a reg-type signal is `X` (unknown value), which cannot be *inverted* in the always block. The initial block also 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.

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  `timescale 1ns / 1ps
2
3  module testbench ( );
4      // reg signals to provide inputs to the DUT
5      reg [0:0] KEY;
6      reg CLOCK_50;
7      reg [9:0] SW;
8      // wire signals to connect to outputs from the DUT
9      wire [9:0] LEDR;
10
11     // instantiate the design under test
12     Accumulate U1 (KEY, CLOCK_50, SW, LEDR);
13
14     // generate a 50 MHz periodic clock waveform
15     always
16         #10 CLOCK_50 <= ~CLOCK_50;
17
18     initial
19     begin
20         CLOCK_50 <= 1'b0;
21         KEY[0] <= 1'b0;
22         SW <= 0;
23         #20 SW[9:5] <= 10;
24         #20 KEY[0] <= 1'b1;
25             SW[4:0] <= 30;
26     end // initial
27 endmodule
```

Figure 9. The Verilog 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.

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 Verilog 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.v* that provides the Verilog code that we will simulate, and a subfolder named *ModelSim*.

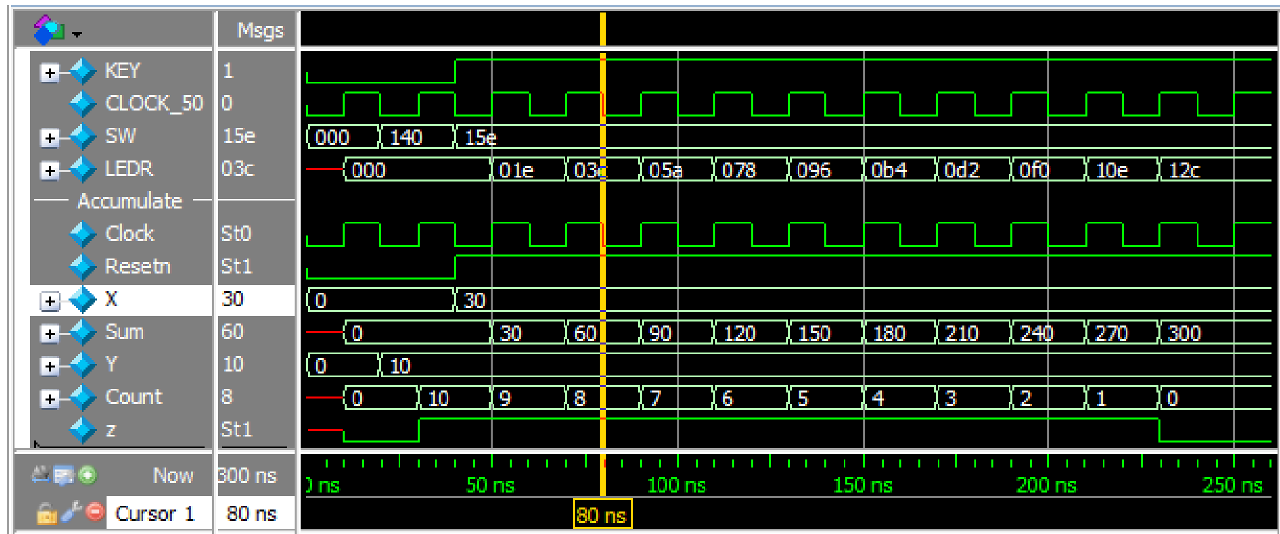


Figure 10. The simulation results for our sequential circuit.

Figure 11 shows the Verilog code for *Display.v*. Its ports are named *KEY*, *SW*, *HEX0*, and *LEDR* because the module is intended to be implemented on a DE-series board that features an Intel FPGA, such as the *DE1-SoC* board. After simulating the Verilog 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 module. 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.

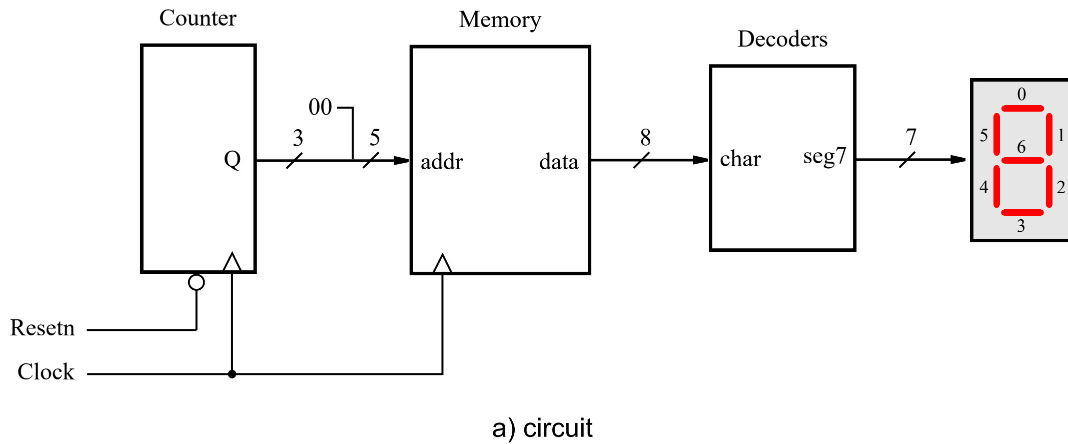
A *testbench.v* file for the *Display* design under test (DUT) is given in Figure 14. Two reg-type signals, *KEY* and *SW* are declared to provide inputs to the DUT, as well as two wire-type signals *HEX0* and *LEDR* for connecting to the DUT outputs. The DUT is instantiated in Line 9 of the testbench. It uses an always block to create a clock waveform with a 20 ns period on the *KEY* signal. An initial block is used to initialize the *KEY* clock waveform to 0, and to perform a synchronous reset of the counter with the SW signal.

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 5 to 8 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 10 to 12 the script checks if an “empty black box” file,

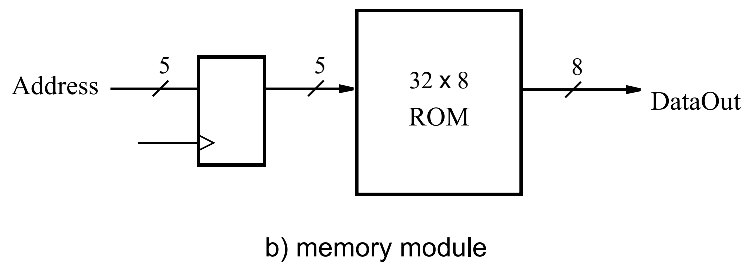
```
module Display (KEY, SW, HEX0, LEDR);
    input [0:0] KEY;
    input [0:0] SW;
    output reg [6:0] HEX0;
    output [9:0] LEDR;

    parameter A = 8'd65, b = 8'd98, C = 8'd67, d = 8'd100, E = 8'd69,
        F = 8'd70, g = 8'd103, h = 8'd104;
    wire Resetn, Clock;
    wire [2:0] Count;
    wire [7:0] char;

    assign Resetn = SW[0];
    assign Clock = KEY[0];

    count3 U1 (Resetn, Clock, Count);
    inst_mem U2 ({2'b0, Count}, Clock, char);
    assign LEDR = {2'b0, char};

    always @(*)
        case (char)
            A: HEX0 = 7'b0001000;
            b: HEX0 = 7'b0000011;
            C: HEX0 = 7'b1000110;
            d: HEX0 = 7'b0100001;
            E: HEX0 = 7'b0000110;
            F: HEX0 = 7'b0001110;
            g: HEX0 = 7'b0010000;
            h: HEX0 = 7'b0001011;
            default HEX0 = 7'b1111111;
        endcase
endmodule

module count3 (Resetn, Clock, Q);
    input Resetn, Clock;
    output reg [2:0] Q;

    always @ (posedge Clock, negedge Resetn)
        if (Resetn == 0)
            Q <= 3'b000;
        else
            Q <= Q + 1'b1;
endmodule
```

Figure 11. Verilog code for the display circuit.

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.



a) circuit



b) memory module

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.

```
1  `timescale 1ns / 1ps
2
3  module testbench ( );
4      reg [0:0] KEY;
5      reg [0:0] SW;
6      wire [6:0] HEX0;
7      wire [9:0] LEDR;
8
9      Display U1 (KEY, SW, HEX0, LEDR);
10
11     always
12     begin : Clock_Generator
13         #10 KEY <= ~KEY;
14     end
15
16     initial
17     begin
18         KEY <= 1'b0;
19         SW <= 1'h0;
20         #20 SW[0] <= 1'b1;
21     end // initial
22 endmodule
```

Figure 14. Testbench code for the memory example.

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 Verilog source-code to be simulated, make a subfolder named *ModelSim*. Copy into this subfolder the files *testbench.v*, *testbench.tcl*, and *wave.do* from one of the examples above. Then, modify *testbench.v* to instantiate your top-level Verilog module 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 *.v* 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.

```
1  # stop any simulation that is currently running
2  quit -sim
3
4  # if simulating with a MIF file, copy it. Assumes
       inst_mem.mif
5  if {[file exists ../inst_mem.mif]} {
6     file delete inst_mem.mif
7     file copy ../inst_mem.mif .
8  }
9  # if Quartus generated an "empty black box" file, delete it
10 if {[file exists ../inst_mem_bb.v]} {
11    file delete ../inst_mem_bb.v
12 }
13 # create the default "work" library
14 vlib work;
15
16 # compile the Verilog source code in the parent folder
17 vlog ../*.v
18 # compile the Verilog code of the testbench
19 vlog *.v
20 # start the Simulator, including some libraries
21 vsim work.testbench -Lf 220model -Lf altera_mf_ver -Lf
       verilog
22 # show waveforms specified in wave.do
23 do wave.do
24 # advance the simulation the desired amount of time
25 run 120 ns
```

Figure 15. The *testbench.tcl* file.

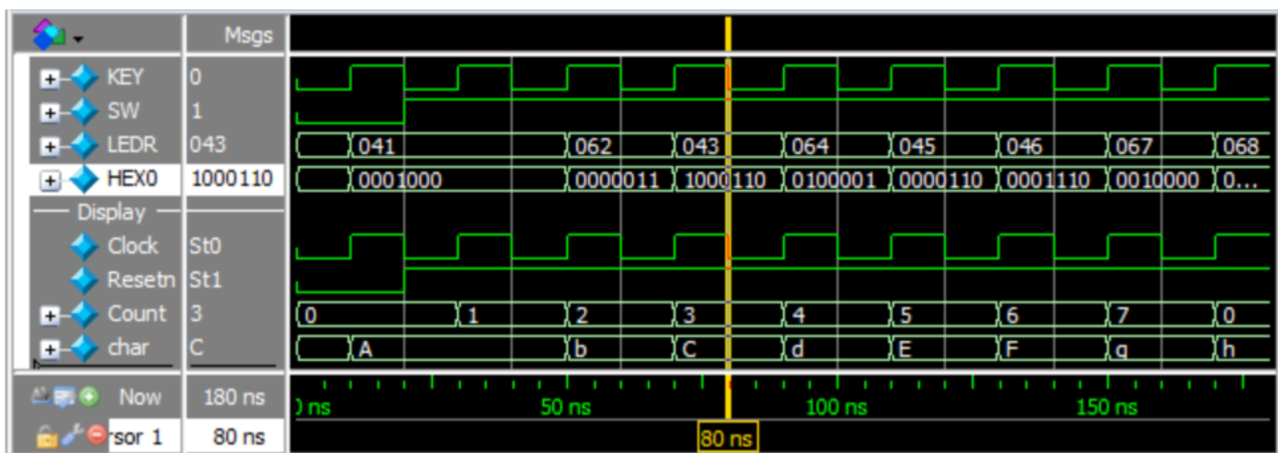


Figure 16. The simulation results for our memory example.

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.

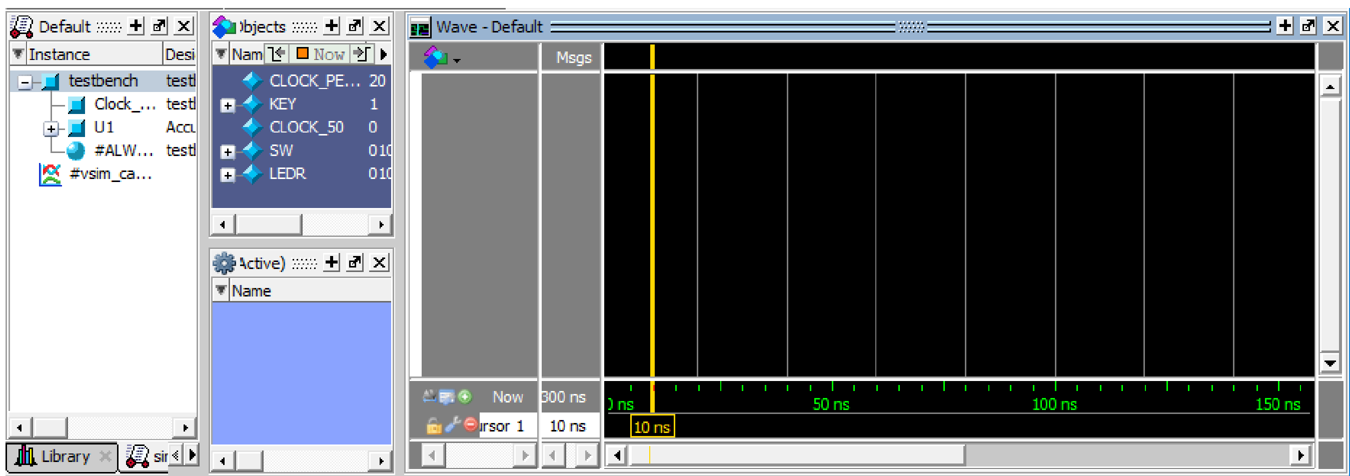


Figure 17. The *ModelSim* waveform display.

In Figure 18 we have selected a waveform, as follows. First, we clicked on the `testbench` module in the upper left part of the display. As a result of this action the signals that exist in the selected module 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.

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` module 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` module to the `Wave` window. But first we can add a *divider*, as a visual aid that separates the `testbench` signals and the `Accumulate` module 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.

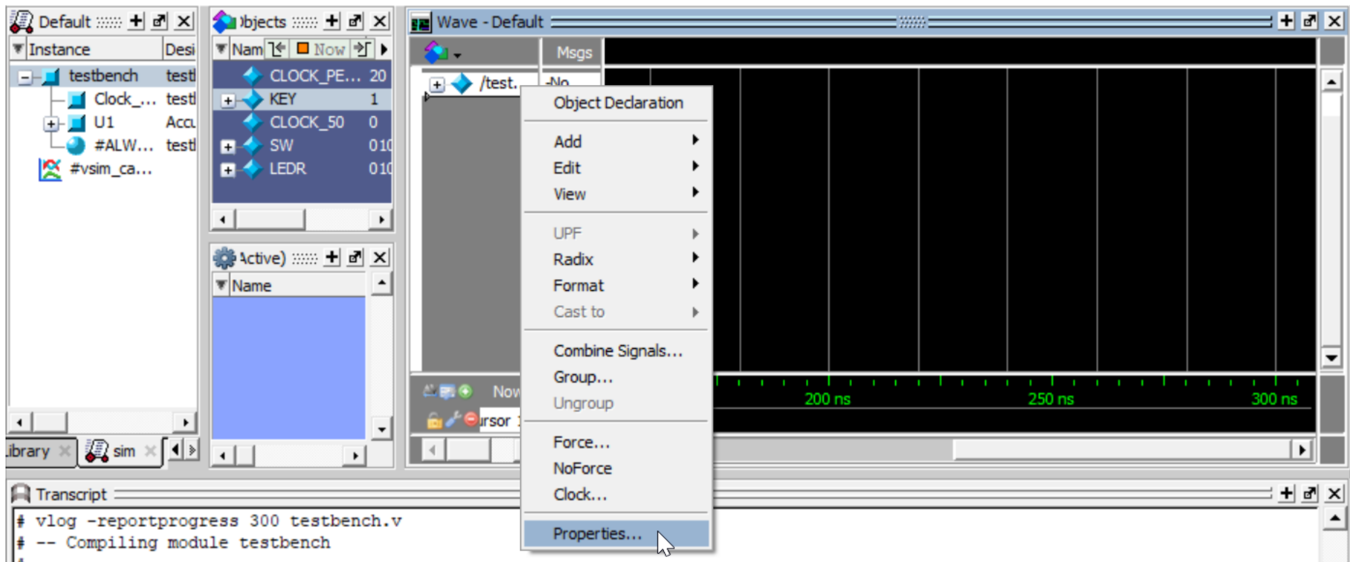


Figure 18. Adding a waveform from the testbench module.

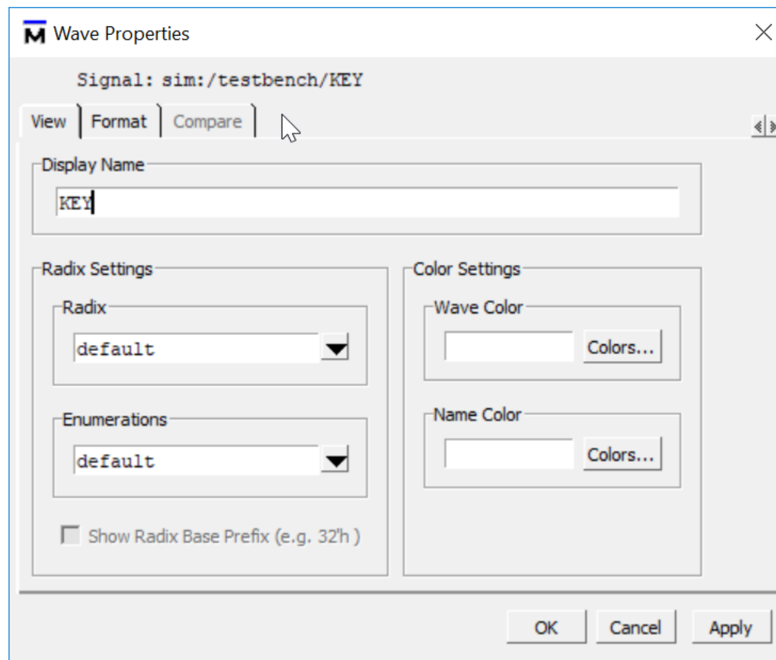


Figure 19. Specifying a display name for a waveform.

To add signals from the `Accumulate` module 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 module 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

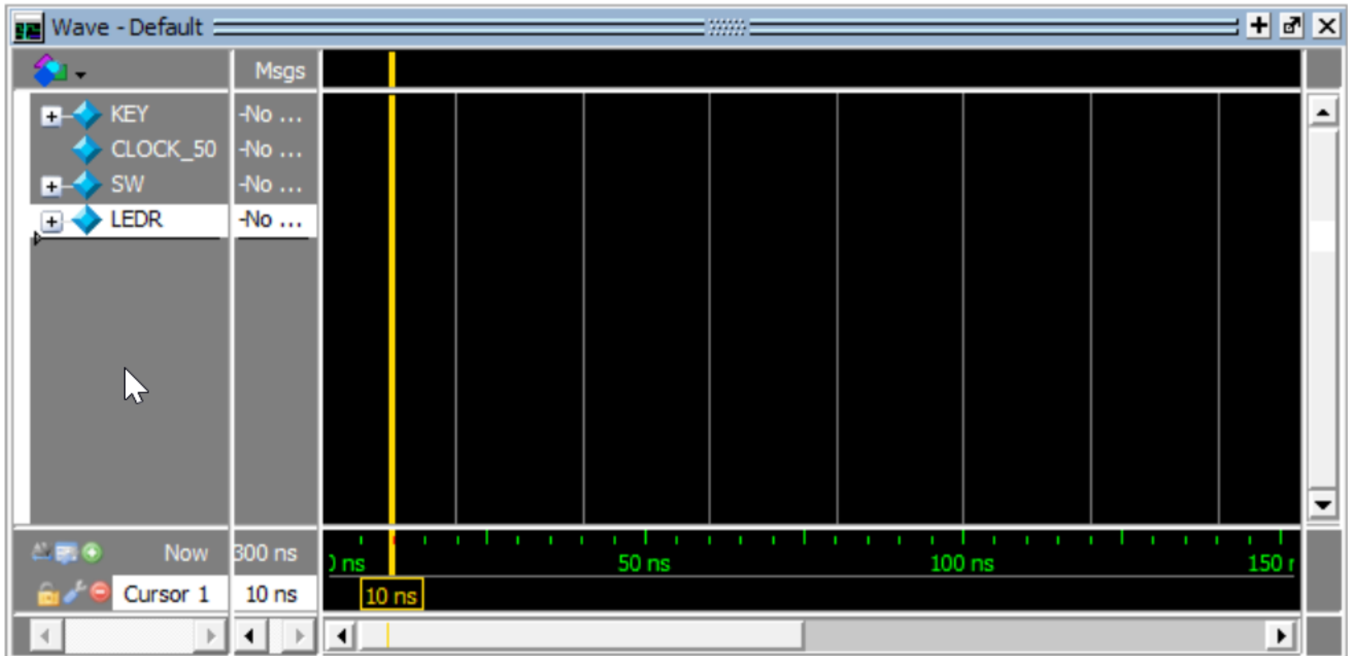


Figure 20. The waveform display after adding more *testbench* signals.

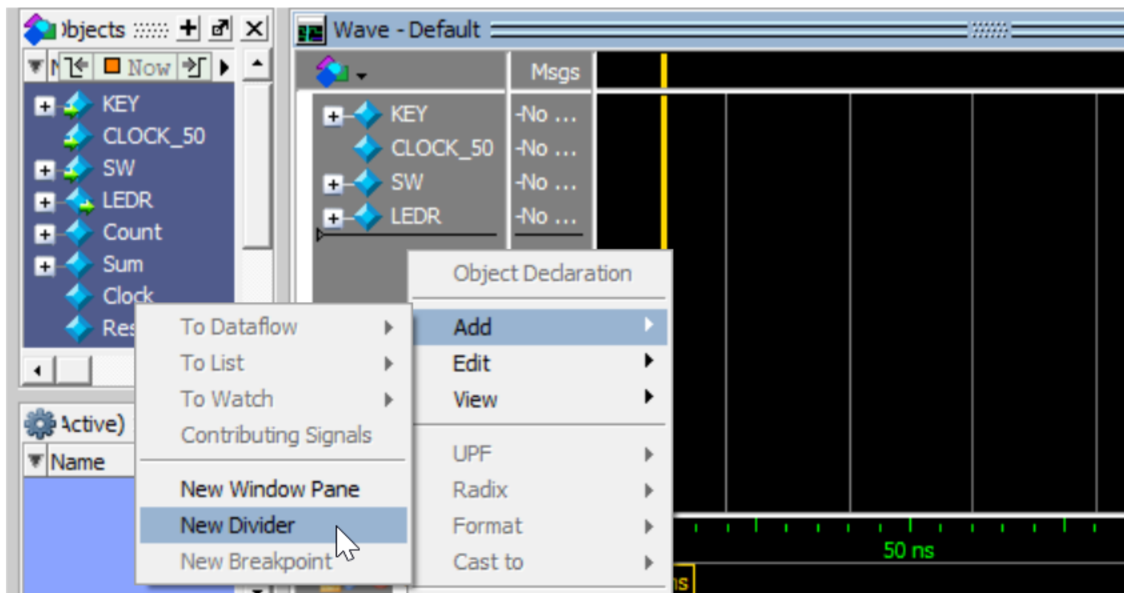


Figure 21. Adding a divider to the waveform display.

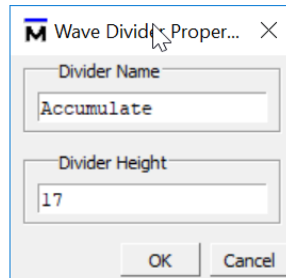


Figure 22. Assigning a name to the divider.

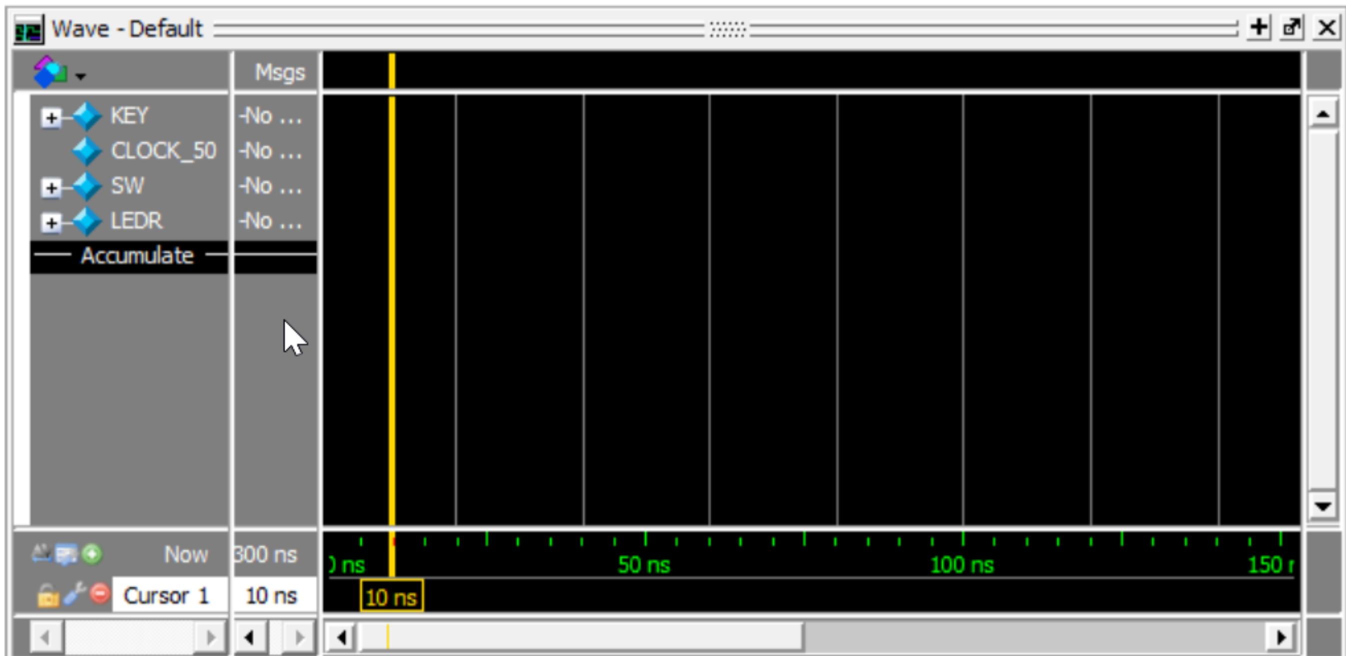


Figure 23. The waveform display after adding the `Accumulate` divider.

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!

The command `File > Save Format` opens the dialogue shown in Figure 25. After clicking `OK` and then over-writing 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.

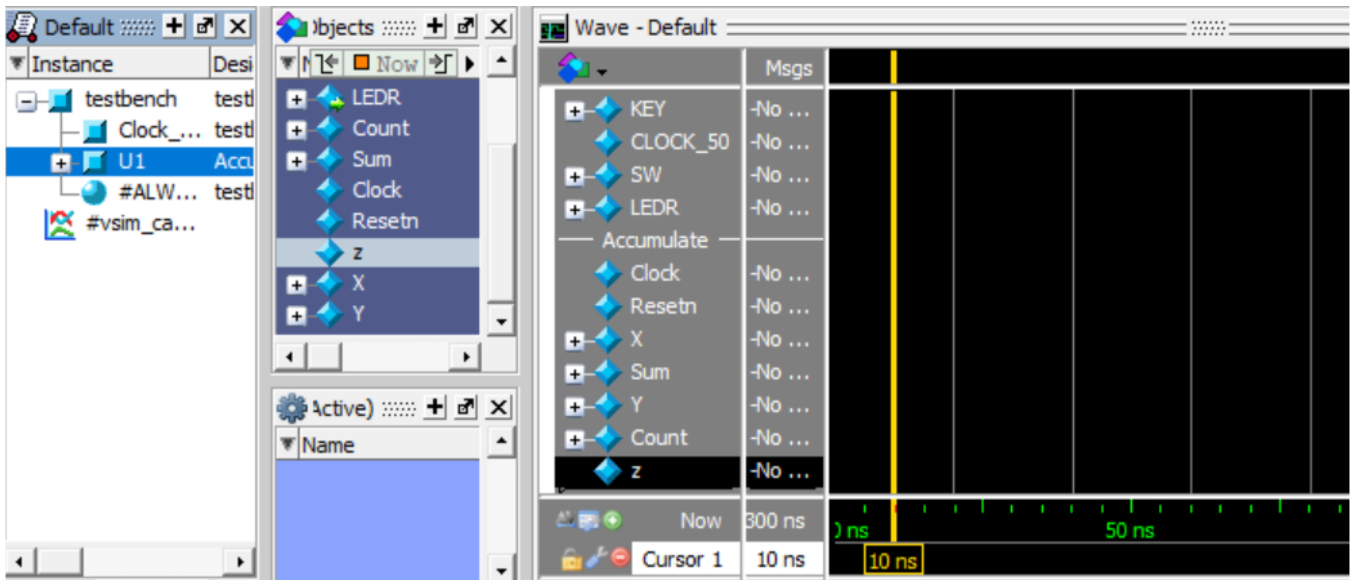


Figure 24. The waveform display after adding signals from the accumulate module.

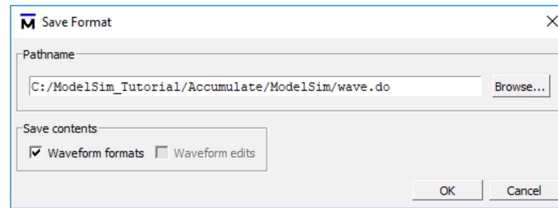


Figure 25. The `Save Format` dialogue.

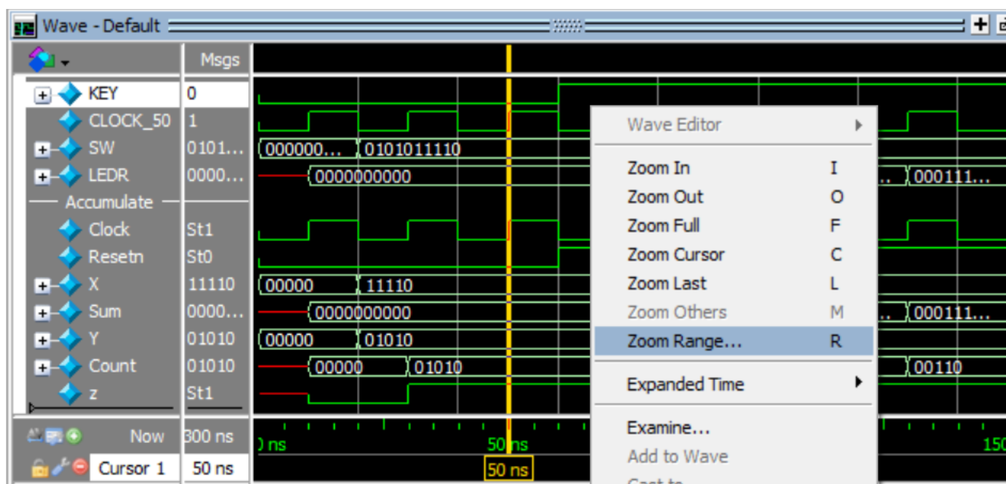


Figure 26. The display after running the simulation; changing the zoom range.

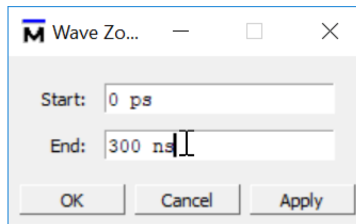


Figure 27. Inputting the zoom range.

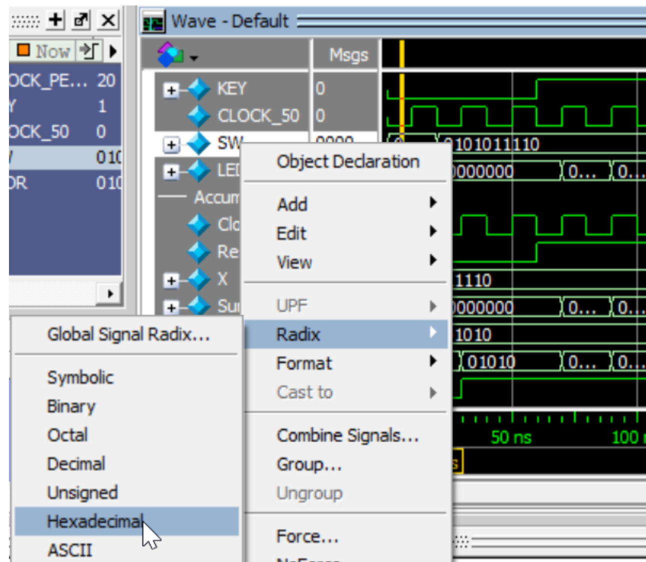


Figure 28. Setting the radix for a waveform.

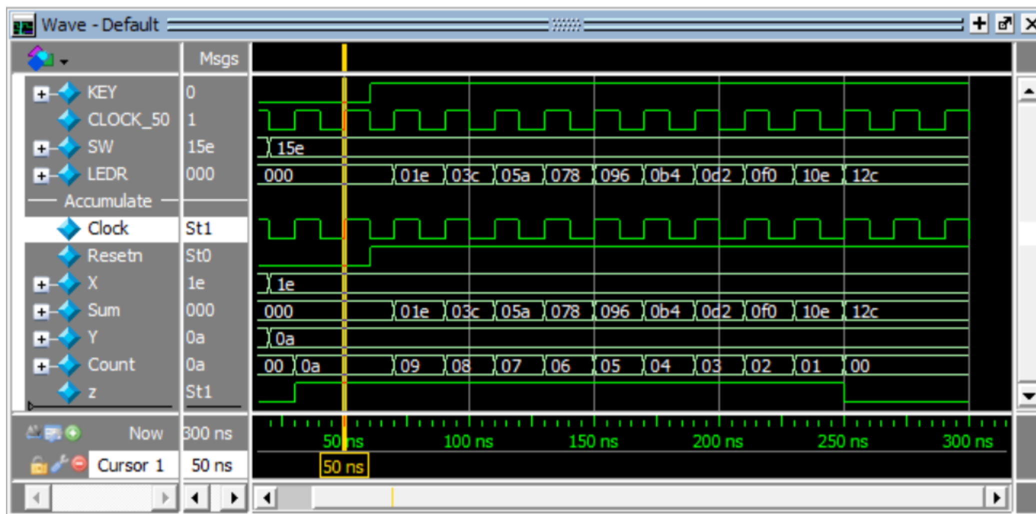


Figure 29. The final waveform display.

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.

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.