



## Using ModelSim to Simulate Logic Circuits in VHDL Designs

*For Quartus II 11.1*

### 1 Introduction

This tutorial is a basic introduction to ModelSim, a Mentor Graphics simulation tool for logic circuits. We show how to perform functional and timing simulations of logic circuits implemented by using Quartus II CAD software.

The reader is expected to have the basic knowledge of VHDL, and the Altera Quartus® II CAD software.

#### **Contents:**

- Introduction to simulation
- What is ModelSim?
- Functional simulation using ModelSim
- Timing simulation using ModelSim

## 2 Background

Designers of digital systems are inevitably faced with the task of testing their designs. Each design can be composed of many components, each of which has to be tested in isolation and then integrated into a design when it operates correctly.

To verify that a design operates correctly we use simulation, which is a process of testing the design by applying inputs to a circuit and observing its behavior. The output of a simulation is a set of waveforms that show how a circuit behaves based on a given sequence of inputs. The general flow of a simulation is shown in Figure 1.

There are two main types of simulation: functional and timing simulation. The functional simulation tests the logical operation of a circuit without accounting for delays in the circuit. Signals are propagated through the circuit using logic and wiring delays of zero. This simulation is fast and useful for checking the fundamental correctness of the designed circuit.

The second step of the simulation process is the timing simulation. It is a more complex type of simulation, where logic components and wires take some time to respond to input stimuli. In addition to testing the logical operation of the circuit, it shows the timing of signals in the circuit. This type of simulation is more realistic than the functional simulation; however, it takes longer to perform.

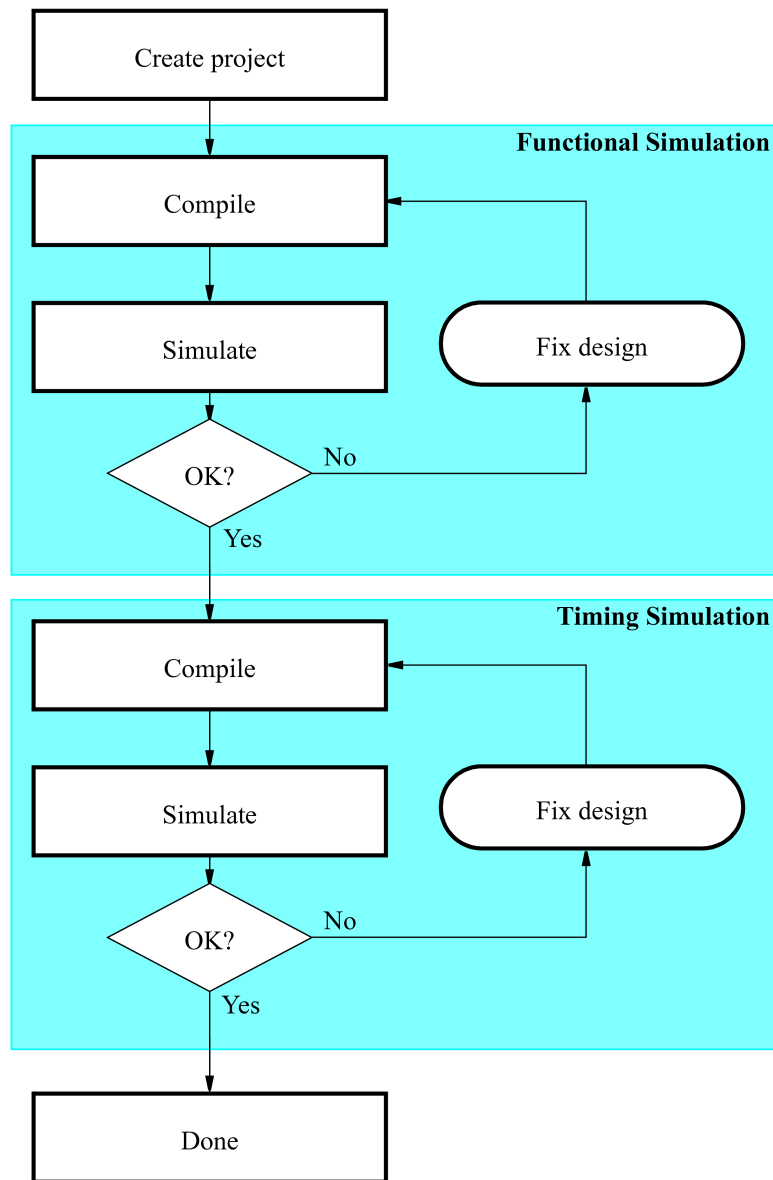


Figure 1. The simulation flow.

In this tutorial, we show how to simulate circuits using ModelSim. You need Quartus II CAD software and ModelSim software, or ModelSim-Altera software that comes with Quartus II, to work through the tutorial.

### 3 Example Design

Our example design is a serial adder. It takes 8-bit inputs *A* and *B* and adds them in a serial fashion when the *start* input is set to 1. The result of the operation is stored in a 9-bit *sum* register.

A block diagram of the circuit is shown in Figure 2. It consists of three shift registers, a full adder, a flip-flop to store carry-out signal from the full adder, and a finite state machine (FSM). The shift registers *A* and *B* are loaded with the values of *A* and *B*. After the *start* signal is set high, these registers are shifted right one bit at a time. At the same time the least-significant bits of *A* and *B* are added and the result is stored into the shift register *sum*. Once all bits of *A* and *B* have been added, the circuit stops and displays the *sum* until a new addition is requested.

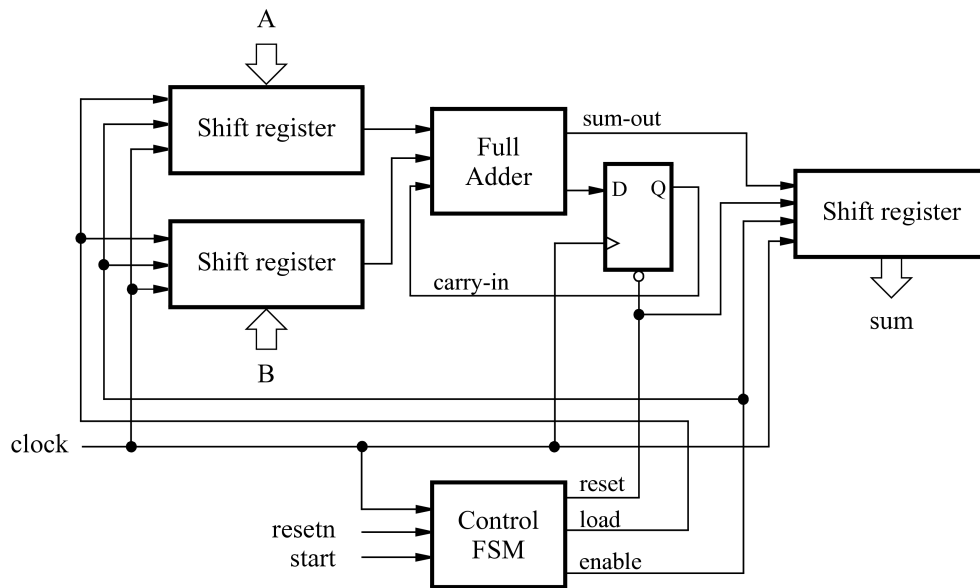


Figure 2. Block diagram of a serial-adder circuit.

The VHDL code for the top-level entity of this design is shown in Figure 3. It consists of the instances of the shift registers, an adder, and a finite state machine (FSM) to control this design. The component declarations for these instances on lines 32-63 have been omitted from Figure 3, as the declarations for the shift register and FSM will be shown in Figures 4 and 5.

```
1.  LIBRARY ieee;
2.  USE ieee.std_logic_1164.all;
3.
4.  -- This is an example of a serial adder.
5.  ENTITY serial IS
6.      PORT(
7.          A :IN STD_LOGIC_VECTOR(7 DOWNTO 0);
8.          B :IN STD_LOGIC_VECTOR(7 DOWNTO 0);
9.          start :IN STD_LOGIC;
10.         resetn :IN STD_LOGIC;
11.         clock :IN STD_LOGIC;
12.         sum :OUT STD_LOGIC_VECTOR(8 DOWNTO 0)
13.     );
14. END serial;
15.
16. ARCHITECTURE Behaviour OF serial IS
17.
18.     -- Registers
19.     SIGNAL A_reg :STD_LOGIC_VECTOR(7 DOWNTO 0);
20.     SIGNAL B_reg :STD_LOGIC_VECTOR(7 DOWNTO 0);
21.     SIGNAL sum_reg :STD_LOGIC_VECTOR(8 DOWNTO 0);
22.     SIGNAL cin :STD_LOGIC;
23.
24.     -- Wires
25.     SIGNAL reset :STD_LOGIC;
26.     SIGNAL enable :STD_LOGIC;
27.     SIGNAL load :STD_LOGIC;
28.     SIGNAL bit_sum :STD_LOGIC;
29.     SIGNAL bit_carry:STD_LOGIC;
30.
31.     -- Component declarations
32.-63. -- (...)
```

Figure 3. VHDL code for the top-level entity of the serial adder (Part *a*).

```

64. BEGIN
65.     PROCESS (clock)
66.     BEGIN
67.     IF clock'EVENT AND clock = '1' THEN
68.         IF (enable = '1') THEN
69.             IF (reset = '1') THEN
70.                 cin <= '0';
71.             ELSE
72.                 cin <= bit_carry;
73.             END IF;
74.         END IF;
75.     END IF;
76.     END PROCESS;
77.
78.     -- Component instantiations
79.     -- Control FSM
80.     my_control : FSM
81.     PORT MAP(start, clock, resetn, reset, enable, load);
82.
83.     -- Datapath
84.     reg_A :shift_reg
85.     PORT MAP(clock, '0', A, '0', enable, load, A_reg);
86.
87.     reg_B :shift_reg
88.     PORT MAP(clock, '0', B, '0', enable, load, B_reg);
89.
90.     -- a full adder
91.     bit_carry <= (A_reg(0) AND B_reg(0)) OR (A_reg(0) AND cin) OR (B_reg(0) AND cin);
92.     bit_sum <= A_reg(0) XOR B_reg(0) XOR cin;
93.
94.     reg_sum :shift_reg
95.     GENERIC MAP(9)
96.     PORT MAP(clock, reset, B"000000000", bit_sum, enable, '0', sum);
97.
98.     END Behaviour;

```

Figure 3. VHDL code for the top-level entity of the serial adder (Part *b*).

The VHDL code for the FSM is shown in Figure 4. The FSM is a 3-state Mealy finite state machine, where the first and the third state waits for the *start* input to be set to 1 or 0, respectively. The computation of the sum of *A* and *B* happens during the second state, called *WORK\_STATE*. The FSM completes computation when the counter reaches a value of 8, indicating that inputs *A* and *B* have been added. The state diagram for the FSM is shown in Figure 5.

```
1.  LIBRARY ieee;
2.  USE ieee.std_logic_1164.all;
3.  USE ieee.std_logic_unsigned.all;
4.  USE ieee.numeric_std.all;
5.
6.  ENTITY FSM IS
7.      GENERIC(
8.          WAIT_STATE :STD_LOGIC_VECTOR(1 DOWNTO 0) := B"00";
9.          WORK_STATE :STD_LOGIC_VECTOR(1 DOWNTO 0) := B"01";
10.         END_STATE :STD_LOGIC_VECTOR(1 DOWNTO 0) := B"11"
11.     );
12.     PORT(
13.         start :IN STD_LOGIC;
14.         clock :IN STD_LOGIC;
15.         resetn :IN STD_LOGIC;
16.         reset :BUFFER STD_LOGIC;
17.         enable :BUFFER STD_LOGIC;
18.         load :BUFFER STD_LOGIC
19.     );
20. END FSM;
21.
22. ARCHITECTURE Behaviour OF FSM IS
23.     SIGNAL current_state :STD_LOGIC_VECTOR(1 DOWNTO 0);
24.     SIGNAL next_state :STD_LOGIC_VECTOR(1 DOWNTO 0);
25.     SIGNAL counter :STD_LOGIC_VECTOR(3 DOWNTO 0);
26.
27. BEGIN
28.     -- next state logic
29.     PROCESS(current_state, start, next_state, counter)
30.     BEGIN
31.         CASE(current_state) IS
32.             WHEN WAIT_STATE =>
33.                 IF (start = '1') THEN
34.                     next_state <= WORK_STATE;
35.                 ELSE
36.                     next_state <= WAIT_STATE;
37.                 END IF;
```

Figure 4. VHDL code for the FSM to control the serial adder (Part a).

```

38.         WHEN WORK_STATE =>
39.             IF (counter = B"1000") THEN
40.                 next_state <= END_STATE;
41.             ELSE
42.                 next_state <= WORK_STATE;
43.             END IF;
44.         WHEN END_STATE =>
45.             IF (start = '0') THEN
46.                 next_state <= WAIT_STATE;
47.             ELSE
48.                 next_state <= END_STATE;
49.             END IF;
50.         WHEN OTHERS =>
51.             next_state <= '-' & '-'; -- don't care
52.     END CASE;
53. END PROCESS;
54.
55. -- state registers and a counter
56. PROCESS(clock, resetn)
57. BEGIN
58.     IF (resetn = '0') THEN
59.         current_state <= WAIT_STATE;
60.         counter <= (OTHERS => '0');
61.     ELSIF (clock'EVENT AND clock = '1') THEN
62.         current_state <= next_state;
63.         IF (current_state = WAIT_STATE) THEN
64.             counter <= (OTHERS => '0');
65.         ELSIF (current_state = WORK_STATE) THEN
66.             counter <= counter + '1';
67.         END IF;
68.     END IF;
69. END PROCESS;
70.
71. -- Outputs
72. reset <= '1' WHEN (current_state = WAIT_STATE) AND (start = '1') ELSE '0';
73. load <= '1' WHEN (current_state = WAIT_STATE) AND (start = '1') ELSE '0';
74. enable <= '1' WHEN (load = '1') OR (current_state = WORK_STATE) ELSE '0';
75. END Behaviour;

```

Figure 4. VHDL code for the FSM to control the serial adder (Part *b*).



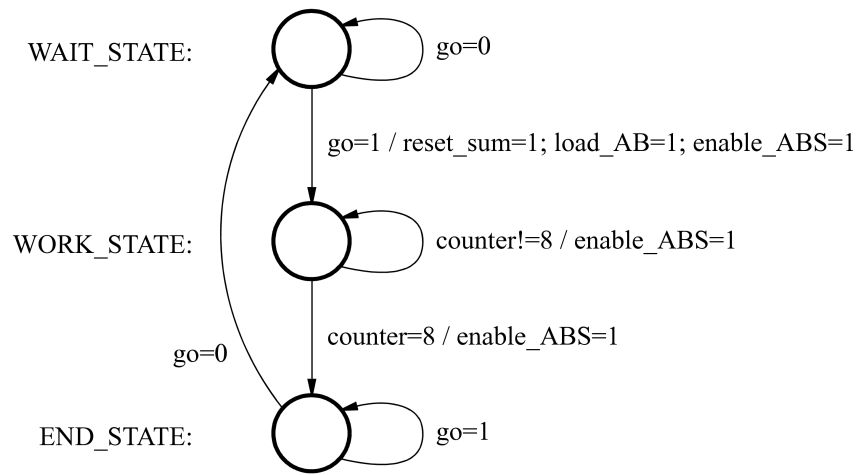


Figure 5. State diagram.

The VHDL code for the shift register is given in Figure 6. It consists of synchronous control signals to allow data to be loaded into the shift register, or reset to 0. When enable input is set to 1 and the data is not being loaded or reset, the contents of the shift register are moved one bit to the right (towards the least-significant bit).

```
1.  LIBRARY ieee;
2.  USE ieee.std_logic_1164.all;
3.
4.  ENTITY shift_reg IS
5.      GENERIC(
6.          n :INTEGER := 8
7.      );
8.  PORT(
9.      clock :IN STD_LOGIC;
10.     reset :IN STD_LOGIC;
11.     data :IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
12.     bit_in :IN STD_LOGIC;
13.     enable :IN STD_LOGIC;
14.     load :IN STD_LOGIC;
15.     q :BUFFER STD_LOGIC_VECTOR(n-1 DOWNTO 0)
16. );
17. END shift_reg;
18.
19. ARCHITECTURE Behaviour OF shift_reg IS
20. BEGIN
21.     PROCESS (clock)
22.     BEGIN
23.         IF clock'EVENT AND clock = '1' THEN
24.             IF (enable = '1') THEN
25.                 IF (reset = '1') THEN
26.                     q <= (OTHERS => '0');
27.                 ELSE
28.                     IF (load = '1') THEN
29.                         q <= data;
30.                     ELSE
31.                         q(n-2 DOWNTO 0) <= q(n-1 DOWNTO 1);
32.                         q(n-1) <= bit_in;
33.                     END IF;
34.                 END IF;
35.             END IF;
36.         END IF;
37.     END PROCESS;
38. END Behaviour;
```

Figure 6. VHDL code for the shift register.

The design is located in the *example/functional* and *example/timing* subdirectories provided with this tutorial. A Quartus II project for this design has been created as well.

In the following sections, we use the serial adder example to demonstrate how to perform simulation using ModelSim. We begin by describing a procedure to perform a functional simulation, and then discuss how to perform a timing simulation.

## 4 Functional Simulation with ModelSim

We begin this tutorial by showing how to perform a functional simulation of the example design. We start by opening the ModelSim program.

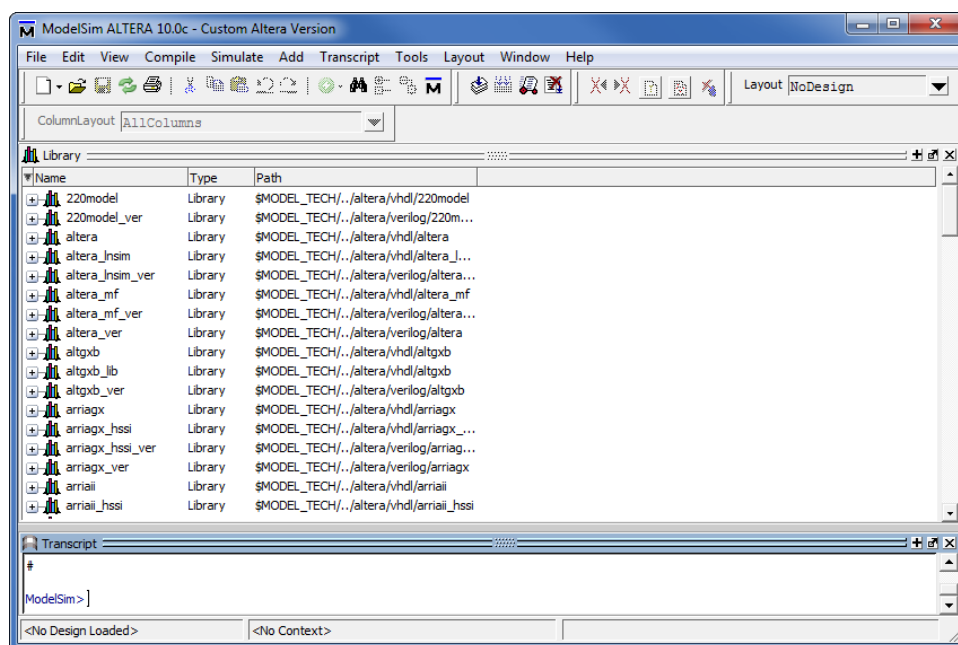


Figure 7. ModelSim window.

The ModelSim program window, shown in Figure 7, consists of three sections: the main menu at the top, a set of workspace tabs, and a command prompt at the bottom. The menu is used to access functions available in ModelSim. The workspace contains a list of entities and libraries of entities available to you, as well as details of the project you are working on. A new work area will appear on the right of the libraries of entities when needed to display waveforms and/or text files. Finally, the command prompt at the bottom shows feedback from the simulation tool and allows users to enter commands.

To perform simulation with ModelSim follow a basic flow shown in Figure 1. We begin by creating a project where all design files to be simulated are included. We compile the design and then run the simulation. Based on the results of the simulation, the design can be altered until it meets the desired specifications.

## 4.1 Creating a Project

To create a project in ModelSim, select File > New > Project.... A Create Project window shown in Figure 8 will appear.

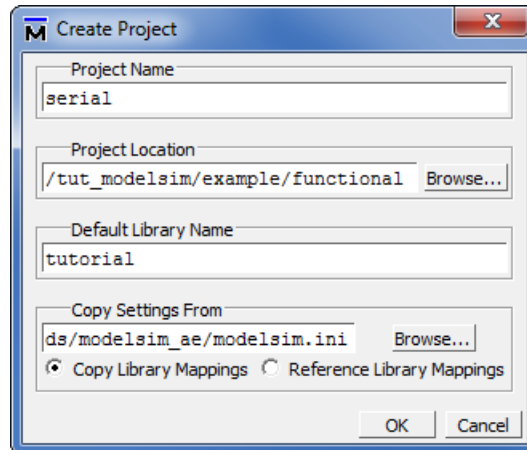


Figure 8. Creating a new project.

The create project window consists of several fields: project name, project location, default library name, and copy settings field. Project name is a user selected name and the location is the directory where the source files are located. For our example, we choose the project name to be *serial*, to match the top-level entity name of our example design, and the location of the project is the *example/functional* subdirectory.

The default library name field specifies a name by which ModelSim catalogues designs. For example, a set of files that describe the logical behaviour of components in an Altera Cyclone II device are stored in the *cycloneii* library. This allows the simulator to include a set of files in simulation as libraries rather than individual files, which is particularly useful for timing simulations where device-specific data is required. For the purpose of this tutorial, specify *tutorial* as the library name for your project.

The last field in the create project window is the copy settings field. This allows default settings to be copied from the initialization file and applied to your project. Now, click OK to proceed to add files to the project using the window shown in Figure 9.

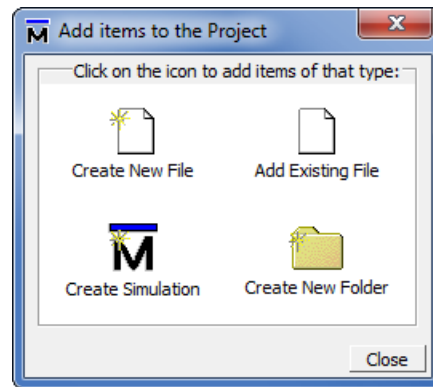


Figure 9. Add a file to project window.

The window in Figure 9 gives several options to add files to the project, including creating new files and directories, or adding existing files. Since the file for this tutorial exists, click **Add Existing File** and select *serial.vhd* file. Once the file is added to the project, it will appear in the Project tab on the left-hand side of the screen, as shown in Figure 10.

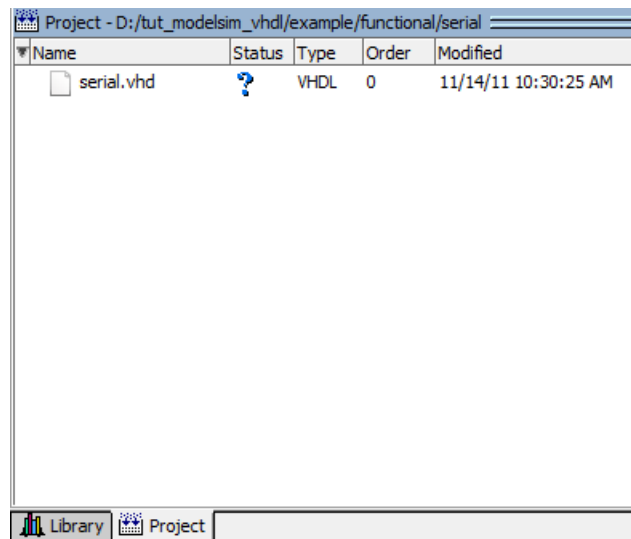


Figure 10. Workspace window after the project is created.

Now that all design files have been included in the project, click **Close** to close the window in Figure 9.

## 4.2 Compiling a Project

Once the project has been created, it is necessary to compile it. Compilation in ModelSim checks if the project files are correct and creates intermediate data that will be used during simulation. To perform compilation, select **Compile All** from the **Compile** menu. When the compilation is successful, a green check mark will appear to the right of the *serial.vhd* file in the Project tab.

## 4.3 Simulation

To begin a simulation of the design, the software needs to be put in simulation mode. To do this, select **Simulate > Start Simulation...**. The window in Figure 11 will appear.

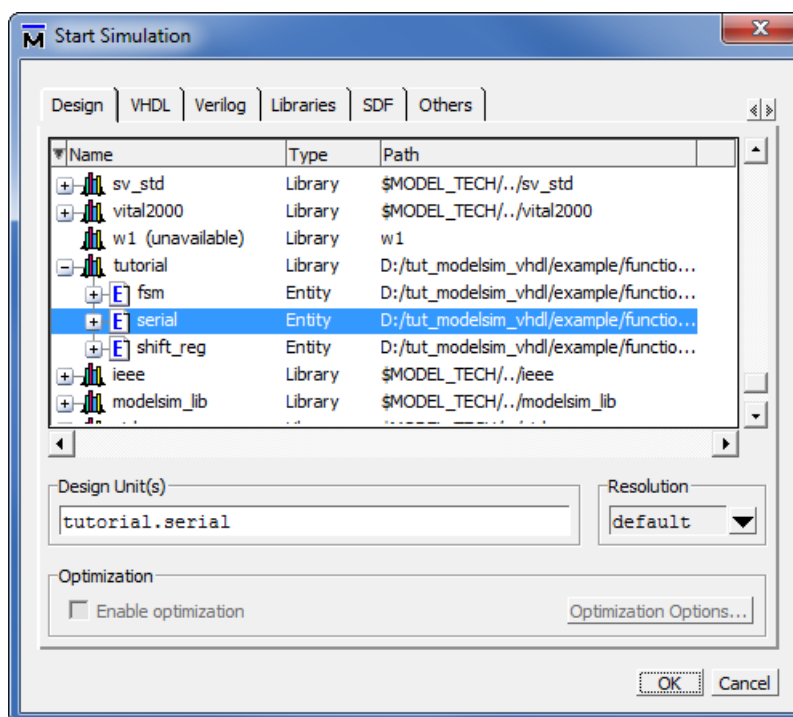


Figure 11. Start simulation mode in ModelSim.

The window to start simulation consists of many tabs. These include a **Design** tab that lists designs available for simulation, **VHDL** and **Verilog** tabs to specify language-specific options, a **Libraries** tab to include any additional libraries, and timing and other options in the remaining two tabs. For the purposes of the functional simulation, we only need to look at the **Design** tab.

In the **Design** tab you will see a list of libraries and entities you can simulate. In this tutorial, we want to simulate an entity called *serial*, described in *serial.vhd* file. To select this entity, scroll down and locate the *tutorial* library and click on the plus (+) sign. You will see three entities available for simulation: *fsm*, *serial*, and *shift\_reg*. Select the *serial* entity, as shown in Figure 11 and click **OK** to begin simulation.

When you click OK, ModelSim will begin loading the selected libraries and preparing to simulate the circuit. For the example in this tutorial, the preparation should complete quickly. Once ModelSim is ready to simulate your design, you will notice that several new tabs on the left-hand side of the screen and a new **Objects** window have appeared, as shown in Figure 12.

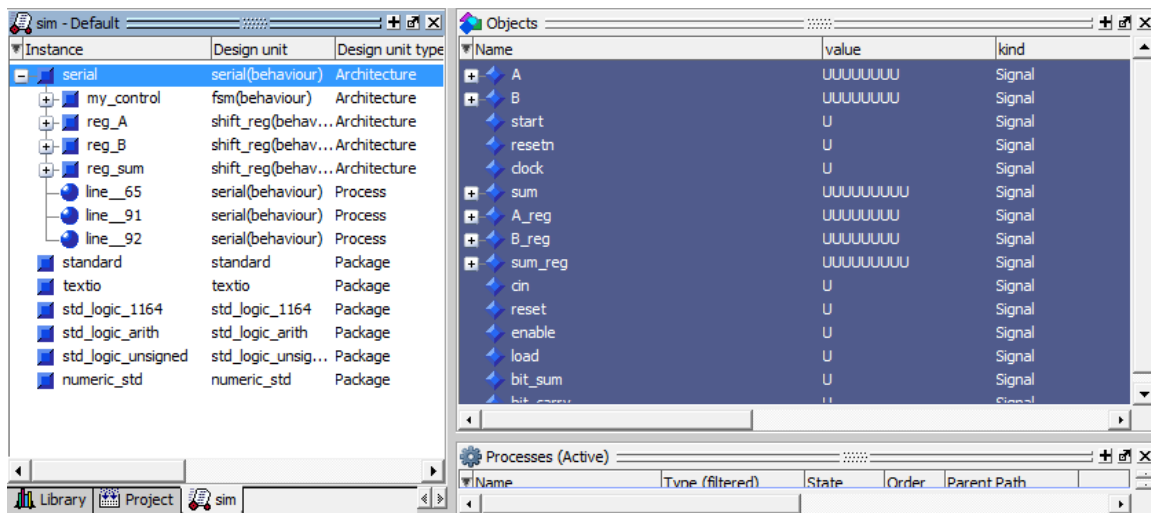


Figure 12. New displays in the simulation mode.

A key new tab on the left-hand side is the **sim** tab. It contains a hierarchical display of design units in your circuit in a form of a table. The columns of the table include the instance name, design unit and design unit type names. The rows of the table take a form of an expandable tree. The tree is rooted in the top-level entity called *serial*. Each entity instance has a plus (+) sign next to its name to indicate it can be expanded to allow users to examine the contents of that entity instance.

Expanding the top-level entity in this view gives a list of components and/or constructs within it. For example, in Figure 12 the top-level entity *serial* is shown to contain an instance of the FSM entity, called *my\_control*, three instances of a *shift\_reg* entity, two assignment statements, and a process block. Double-clicking on any of the constructs will cause ModelSim to open a source file and locate the given construct within it. Double-clicking on an entity/component will open a source file and point to the description of the entity in the source file.

In addition to showing entities and/or constructs, the **sim** tab can be used to locate signals for simulation. Notice that when the *serial* entity is highlighted, a list of signals (inputs, outputs, and local wires) is shown in the **Objects** window. The signals are displayed as a table with four columns: name, value, kind, and mode. The name of a signal may be preceded by a plus (+) sign to indicate that it is a bus. The top-level entity comprises signals *A*, *B*, *resetrn*, *start*, and *clock* as inputs, a *sum* output and a number of internal signals.

We can also locate signals inside of components in the design. To do this, highlight a component whose signals you wish to see in the **Objects** window. For example, to see the signals in the *my\_control* instance of the FSM entity, highlight the *my\_control* instance in the **sim** tab. This will give a list of signals inside of the instance as shown in Figure 13.

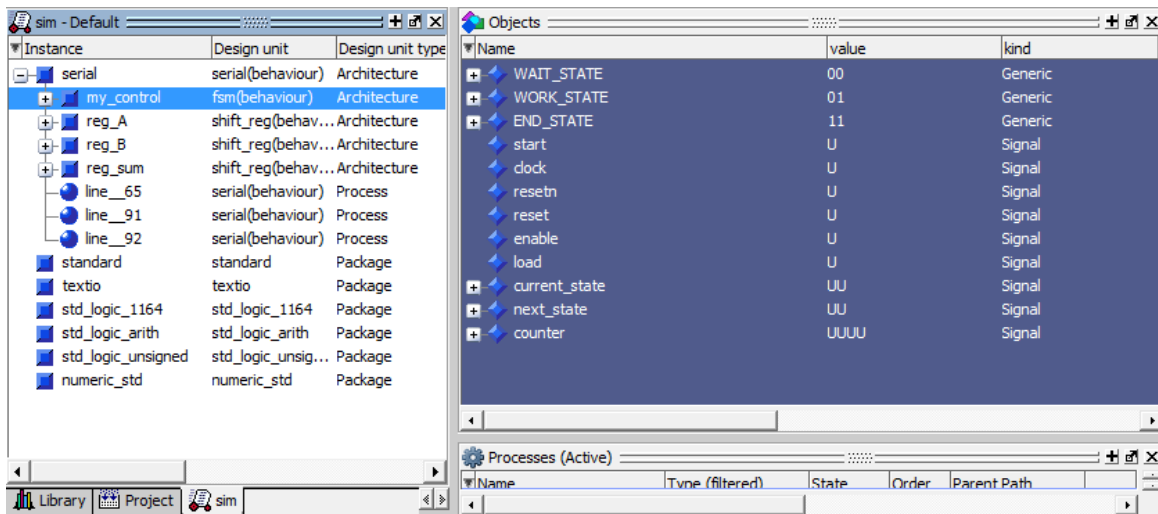


Figure 13. Expanded *my\_control* instance.

Using the *sim* tab and the *Objects* window we can select signals for simulation. To add a signal to simulation, right-click on the signal name in the *Objects* window and select *Add > To Wave > Selected Signals* from the pop-up menu. Using this method, add signals *A*, *B*, *resetn*, *start*, *clock*, *sum*, and *current\_state* to the simulation. When you do so, a waveform window will appear in the work area. Once you have added these signals to the simulation, press the *Undock* button in the top-right corner of the waveform window to make it a separate window, as shown in Figure 14.



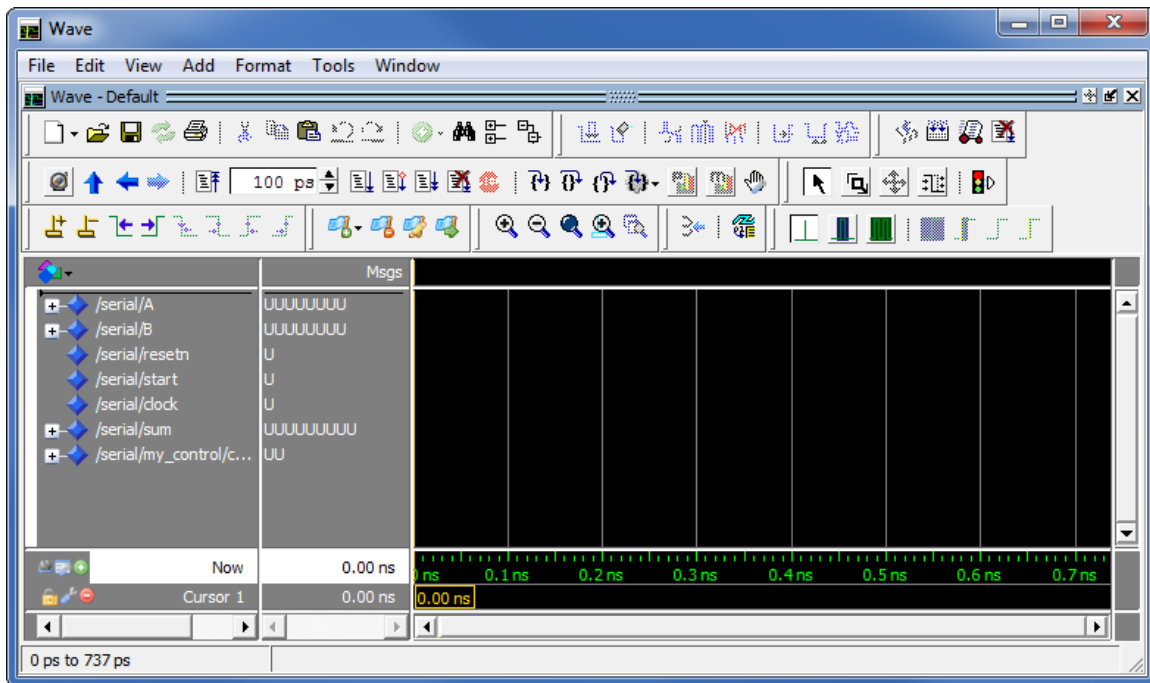


Figure 14. A simulation window.

Now that we set up a set of signals to observe we can begin simulating the circuit. There are two ways to run a simulation in ModelSim: manually or by using scripts. A manual simulation allows users to apply inputs and advance the simulation time to see the results of the simulation in a step-by-step fashion. A scripted simulation allows the user to create a script where the sequence of input stimuli are defined in a file. ModelSim can read the file and apply input stimuli to appropriate signals and then run the simulation from beginning to end, displaying results only when the simulation is completed. In this tutorial, we perform the simulation manually.

In this simulation, we use a clock with a 100 ps period. At every negative edge of the clock we assign new values to circuit inputs to see how the circuit behaves. To set the clock period, right-click on the *clock* signal and select Clock... from the pop-up menu. In the window that appears, set the clock period to 100 ps and the first edge to be the falling edge, as shown in Figure 15. Then click OK.

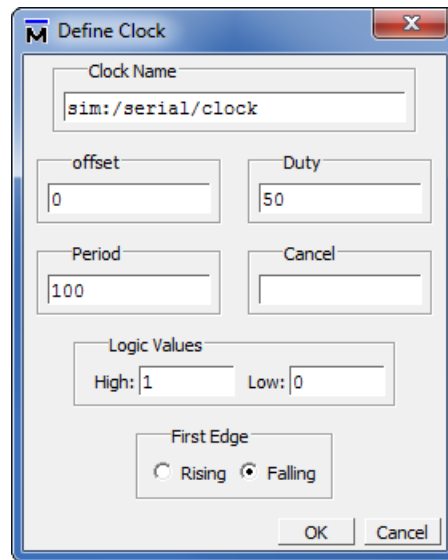


Figure 15. Set the clock period.

We begin the simulation by resetting the circuit. To reset the circuit, set the *resetn* signal low by right-clicking on it and selecting the **Force...** option from the pop-up menu. In the window that appears, set **Value** to 0 and click **OK**. In a similar manner, set the value of the *start* signal to 0. Now that the initial values for some of the signals are set, we can perform the first step of the simulation. To do this, locate the toolbar buttons shown in Figure 16.

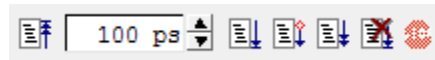


Figure 16. Simulation control buttons on the toolbar.

The toolbar buttons shown in Figure 16 are used to step through the simulation. The left-most button is the restart button, which causes the simulation window to be cleared and the simulation to be restarted. The text field, shown with a 100 ps string inside it, defines the amount of time that the simulation should run for when the **Run** button (to the right of the text field) is pressed. The remaining three buttons, **Continue**, **Run -All** and **Break**, can be used to resume, start and interrupt a simulation, respectively. We will not need them in this tutorial.

To run a simulation for 100 ps, set the value in the text field to 100 ps and press the **Run** button. After the simulation run for 100 ps completes, you will see the state of the circuit as shown in Figure 17. You can change the time scale of your waveform by going to **View > Zoom > Zoom Range....** Change the end time to 1200 ps and press **OK**.

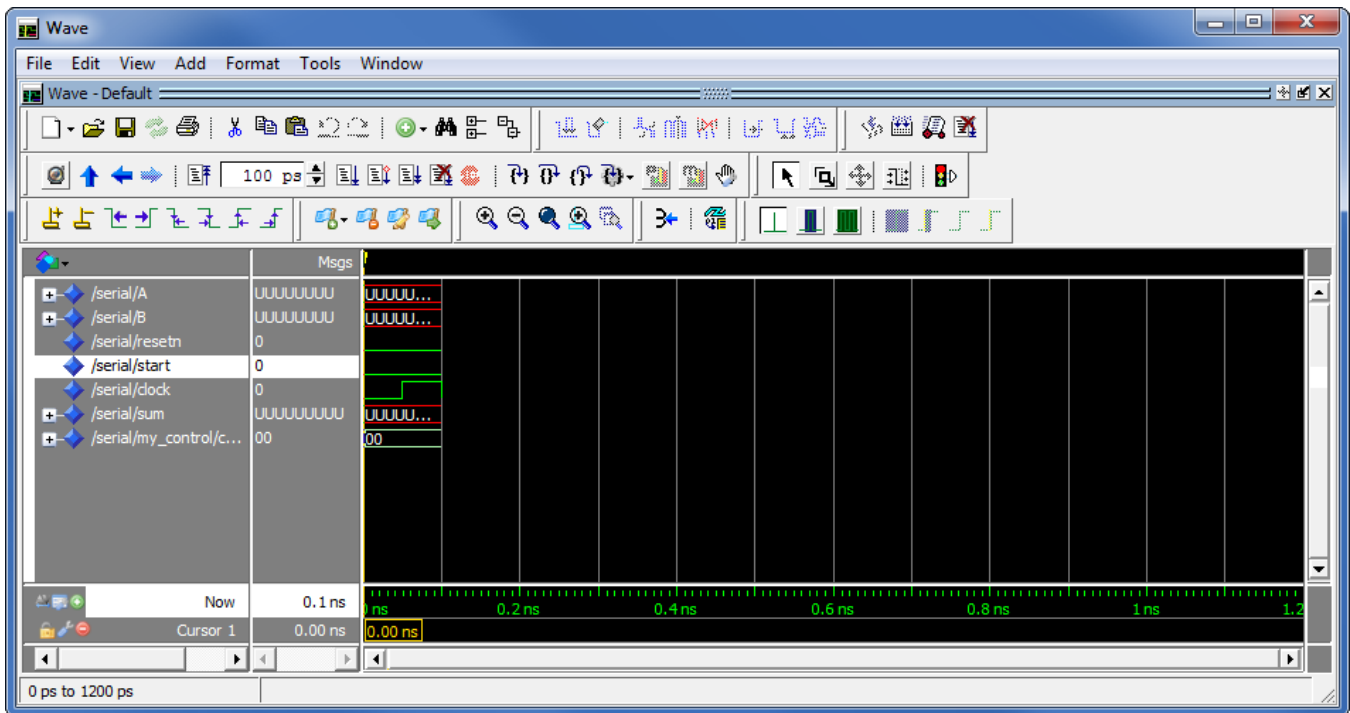


Figure 17. Simulation results after 100 ps.

In the figure, each signal has a logic state. The first two signals, *A* and *B*, are assigned an unknown value represented by *U*s. This value indicates these signals are not driven to any logic state. The *start* and *resetn* signals are at a logic 0 value thereby resetting the circuit. The *clock* signal toggles state every 50 ps, starting with a falling edge at time 0, a rising edge at time 50 ps and another falling edge at 100 ps.

Now that the circuit is reset, we can begin testing to see if it operates correctly for desired inputs. To test the serial adder we will add numbers 143 and 57, which should result in a sum of 200. We can set *A* and *B* to 143 and 57, respectively, using decimal notation. To specify a value for *A* in decimal, right-click on it, and choose *Force...* from the pop-up menu. Then, in the *Value* field put 10#143. The 10# prefix indicates that the value that follows is specified in decimal. Similarly, set the *Value* field of *B* to 10#57.

To see the decimal, rather than binary, values of buses in the waveform window we need to change the *Radix* of *A* and *B* to *unsigned*. To change the radix of these signals, highlight them in the simulation window and select *Format > Radix > Unsigned*, as shown in Figure 18. Change the radix of the *sum* signal to *unsigned* as well.

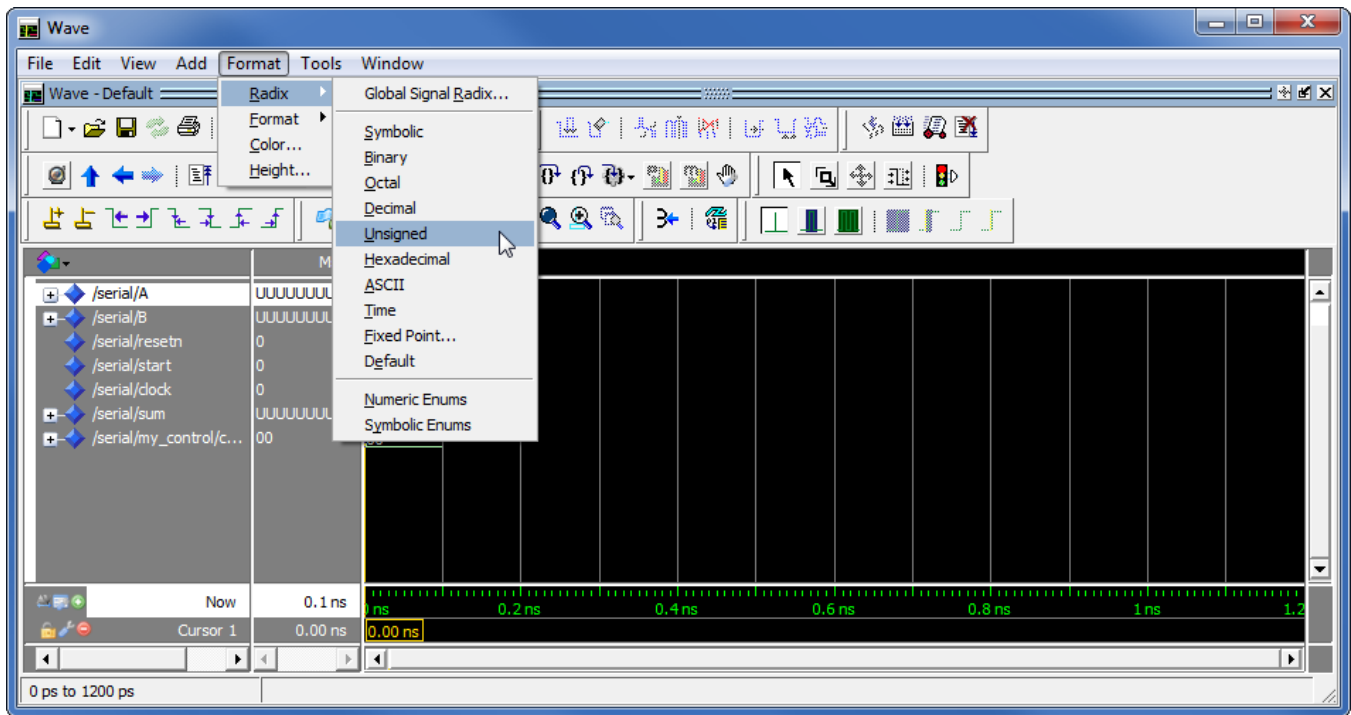


Figure 18. Changing the radix of *A*, *B* and *sum* signals.

Now that inputs *A* and *B* are specified, set *resetn* to 1 to stop the circuit from resetting. Then set *start* to 1 to begin serial addition, and press the RUN button to run the simulation for another 100 ps. The output should be as illustrated in Figure 19. Notice that the values of inputs *A* and *B* are shown in decimal as is the *sum*. The circuit also recognized a *start* signal and moved to state 01 to begin computing the sum of the two inputs.

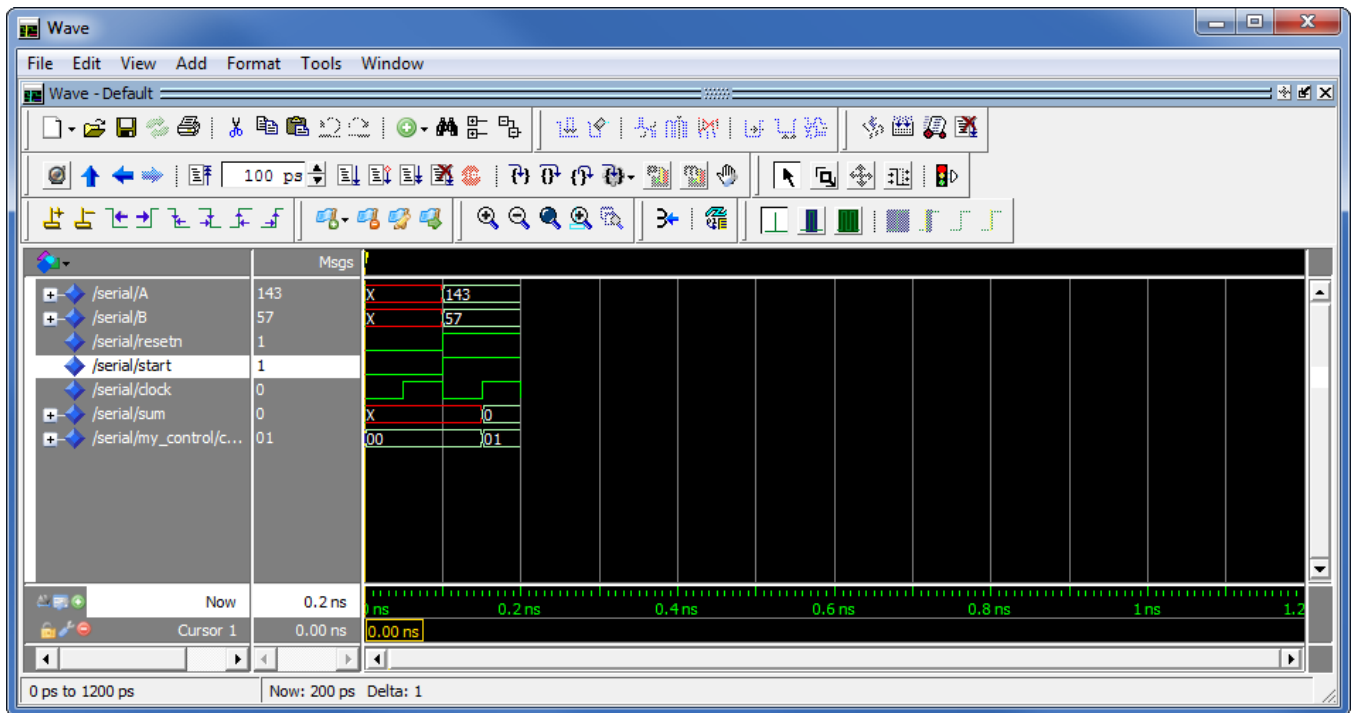


Figure 19. Simulation results after 200 ps.

To complete the operation, the circuit will require 9 clock cycles. To fast forward the simulation to see the result, specify 900 ps in the text field next to the run button, and press the run button. This brings the simulation to time 1100 ps, at which point a result of summation is shown on the *sum* signal, as illustrated in Figure 20.

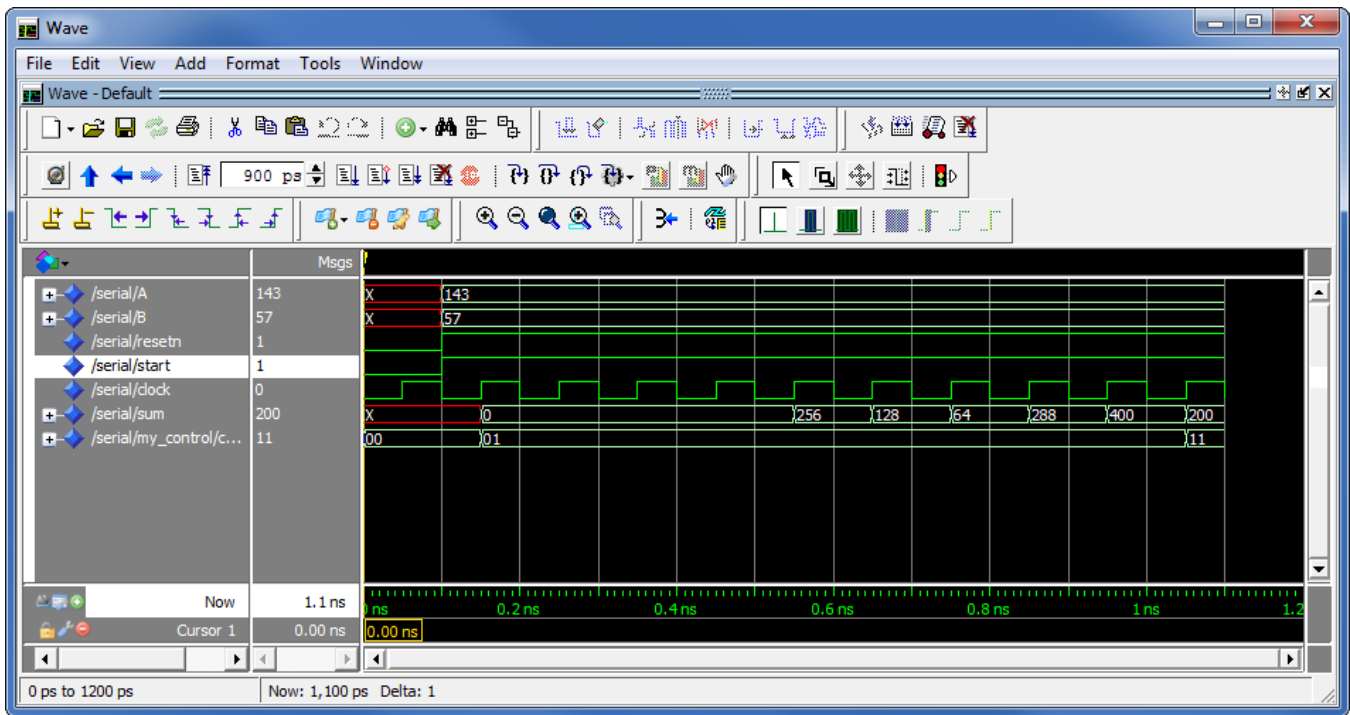


Figure 20. Simulation results after 1100 ps.

We can see that the result is correct and the finite state machine controlling the serial adder entered state 11, in which it awaits the *start* signal to become 0. Once we set the *start* signal to 0 and advance the simulation by 100 ps, the circuit will enter state 00 and await a new set of inputs for addition. The simulation result after 1200 ps is shown in Figure 21.

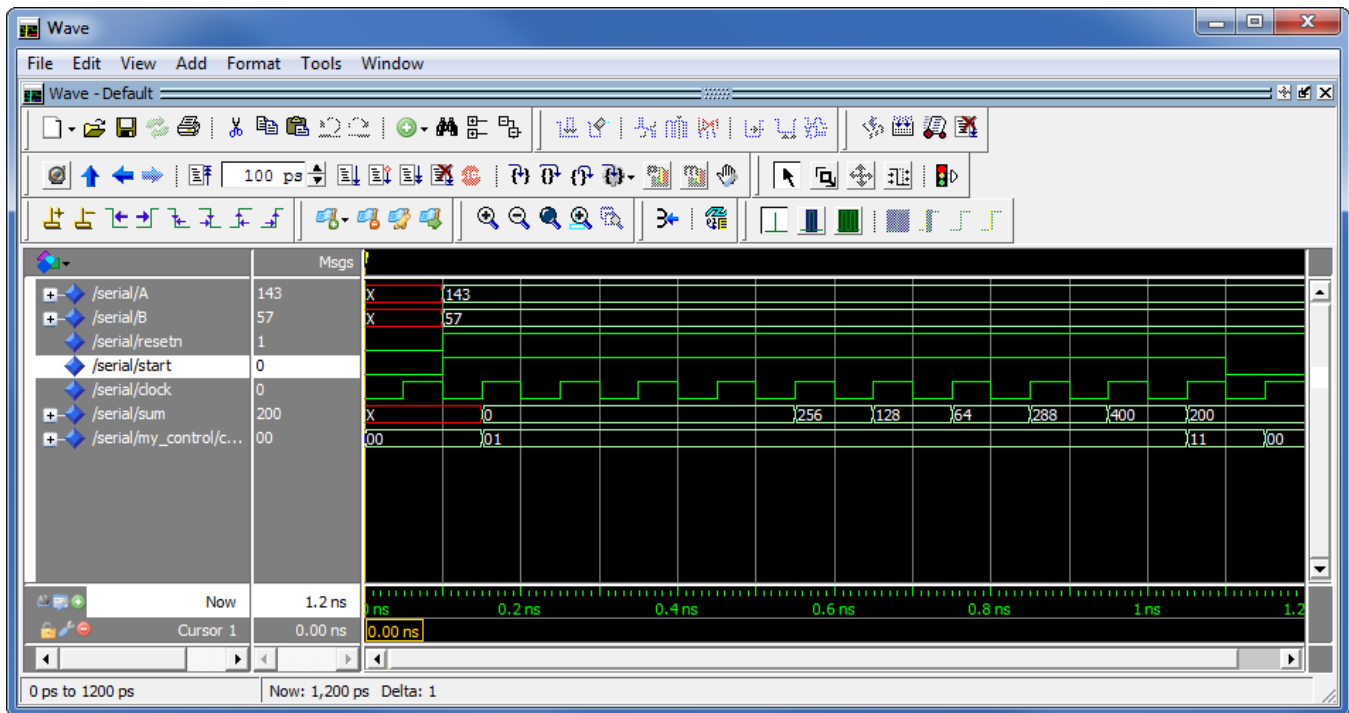


Figure 21. Simulation results after 1200 ps.

At this point, we can begin the simulation for a new set of inputs as needed, repeating the steps described above. We can also restart the simulation by pressing the restart button to begin again from time 0.

By using the functional simulation we have shown that the *serial.vhd* file contains an accurate VHDL description of a serial adder. However, this simulation did not verify if the circuit implemented on an FPGA is correct. This is because we did not use a synthesized, placed and routed circuit as input to the simulator. The correctness of the implementation, including timing constraints can be verified using timing simulation.

## 5 Timing Simulation with ModelSim

Timing simulation is an enhanced simulation, where the logical functionality of a design is tested in the presence of delays. Any change in logic state of a wire will take as much time as it would on a real device. This forces the inputs to the simulation be realistic not only in terms of input values and the sequence of inputs, but also the time when the inputs are applied to the circuit.

For example, in the previous section we simulated the sample design and used a clock period of 100 ps. This clock period is shorter than the minimum clock period for this design, and hence the timing simulation would fail to produce the correct result. To obtain the correct result, we have to account for delays when running the simulation and use a clock frequency for which the circuit operates correctly.

For Altera FPGA-based designs the delay information is available after the design is synthesized, placed and routed, and is generated by Quartus II CAD software. The project for this part of the tutorial has been created for you in the *example/timing* subdirectory.

## 5.1 Setting up a Quartus II Project for Timing Simulation with ModelSim

To perform timing simulation we need to set up Quartus II software to generate the necessary delay information for ModelSim by setting up EDA Tools for simulation in the Quartus II project. To set up EDA Tools for simulation, open the Quartus II project in *example/timing* subdirectory, and select **Assignment > Settings....** A window shown in Figure 22 will appear.

The window in the figure consists of a list on the left-hand side to select the settings category and a window area on the right-hand side that displays the settings for a given category. Select **Simulation** from the *EDA Tool Settings* category to see the screen shown on the right-hand side of Figure 22.

The right-hand side of the figure contains the tool name at the top, EDA Netlist Writer settings in the middle, and NativeLink settings at the bottom. The tool name is a drop-down list containing the names of simulation tools for which Quartus II can produce a netlist with timing information automatically. This list contains many well-known simulation tools, including ModelSim. From the drop-down list select **ModelSim-Altera**.

Once a simulation tool is selected, *EDA Netlist Writer settings* become available. These settings configure Quartus II to produce input for the simulation tool. Quartus II will use these parameters to describe an implemented design using a given HDL language, and annotate it with delay information obtained after compilation. The settings we can define are the HDL language, simulation time scale that defines time step size for the simulator to use, the location where the writer saves design and delay information, and others. Set these settings to match those shown in Figure 22 and click **OK**.

With the EDA Tools Settings specified, we can proceed to compile the project in Quartus II. The compilation process synthesizes, places, and routes the design, and performs timing analysis. Then it stores the compilation result in the simulation directory for ModelSim to use. Take a moment to examine the files generated for simulation using a text editor. The two main files are *serial.vo*, and *serial\_v.sdo*.

The *serial.vo* file is a Verilog file for the design. The file looks close to the original VHDL file, except that the design now contains a wide array of modules with a *cycloneii\_* prefix. These entities describe resources on an Altera Cyclone II FPGA, on which the design was implemented using lookup tables, flip-flops, wires and I/O ports. The list of delays for each module instance in the design is described in the *serial\_v.sdo* file.



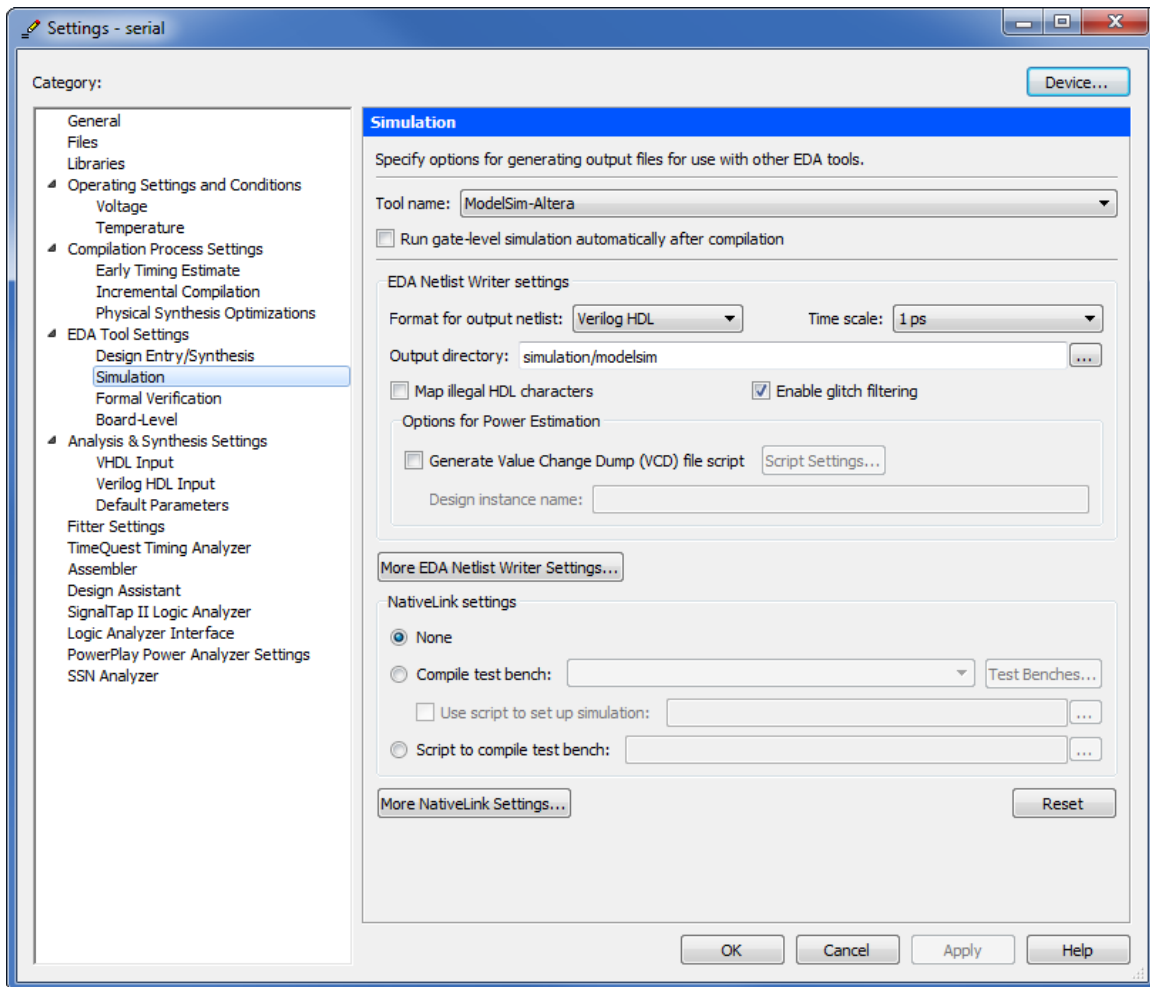


Figure 22. Quartus II EDA simulation tool settings.

## 5.2 Running a Timing Simulation

To simulate the design using timing simulation we must create a ModelSim project. The steps are the same as in the previous section; however, the project is located in the *example/timing/simulation/modelsim* subdirectory, and the source file is *serial.vo*. We do not need to include the *serial\_v.sdo* file in the project, because a reference to it is included in the *serial.vo* file. Once you added the source file to the project, compile it by selecting **Compile > Compile All**.

The next step in the simulation procedure is to place the ModelSim software in simulation mode. In the previous section, we did this by selecting **Simulate > Start Simulation...**, and specifying the project name. To run a timing simulation there is an additional step required to include the Altera Cyclone II device library in the simulation. This library contains information about the logical operation of entities with the *cycloneii\_* prefix. To include the Cyclone II library in the project, select **Simulate > Start Simulation...** and select the **Libraries** tab as shown in Figure 23.

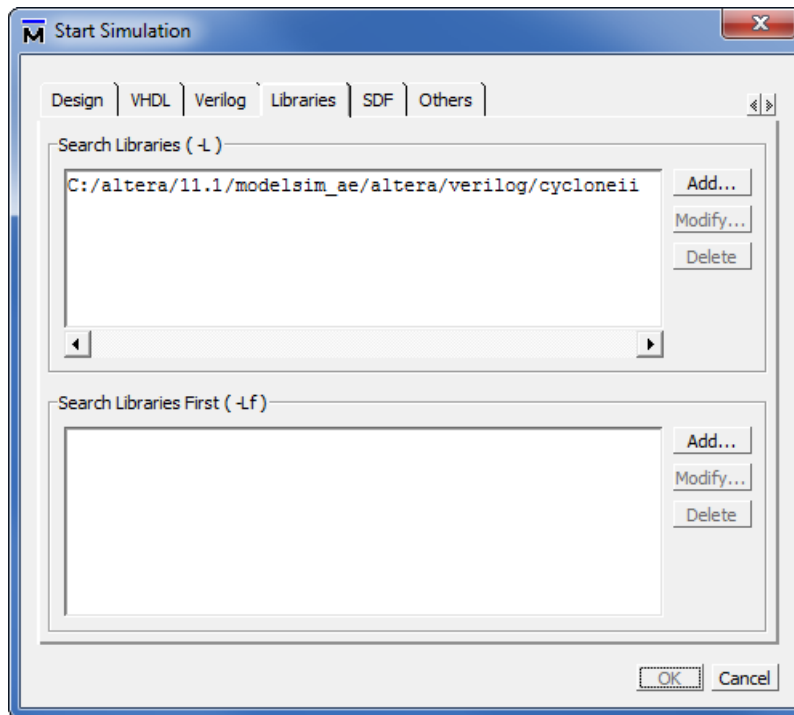


Figure 23. Including Altera Cyclone II library in ModelSim project.

The Altera Cyclone II library is located in the *altera/verilog/cycloneii* directory in the ModelSim-Altera software. To add this library to your project, include the *altera/verilog/cycloneii* directory using the **Add...** button. Then, click on the **Design** tab, select your project for simulation (**tutorial > serial**), and click **OK**.

When the ModelSim software enters simulation mode, you will see a significant difference in the contents of the workspace tabs on the left-hand side of the window as compared to when you ran the functional simulation. In particular, notice the **sim** tab and the **Objects** window shown in Figure 24. The list of entities in the **sim** tab is larger, and the objects window contains more signals. This is due to the fact that the design is constructed using components on an FPGA and is more detailed in comparison to an abstract description we used in the previous section of the tutorial.

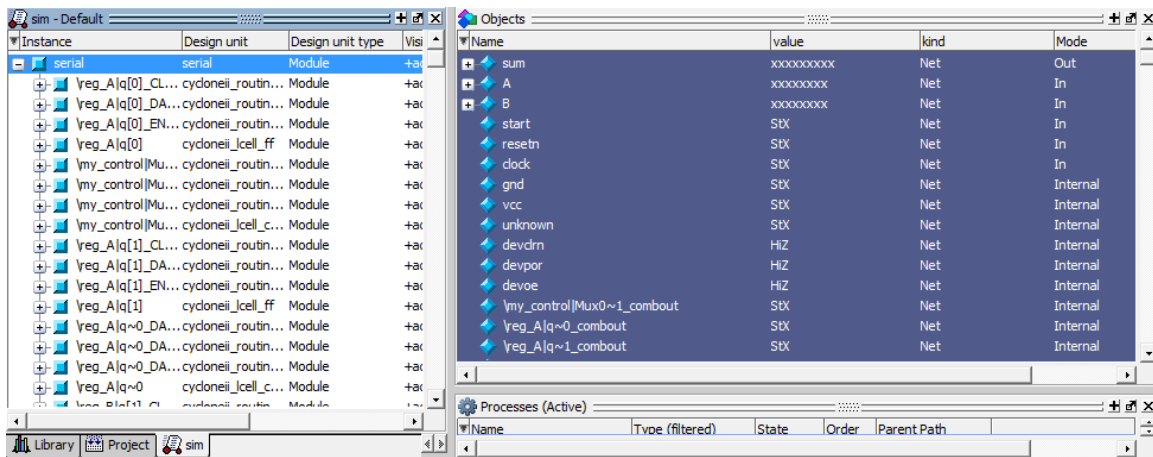


Figure 24. Workspace tabs and Objects window for timing simulation.

We simulate the circuit by creating a waveform that includes signals *sum*, *A*, *B*, *start*, and *resetn* as before. In addition, we include the *clock*, *reg\_sum[q]*, *reg\_A[q]*, and *reg\_B[q]* signals from the Objects window. Signals *reg\_A[q]* and *reg\_B[q]* are registers that store *A* and *B* at the positive edge of the clock. The *reg\_sum[q]* signal is a register that stores the resulting sum.

Begin the simulation by resetting the circuit. To do this, set *start* and *resetn* signals to 0. Also, set the *clock* input to have a period of 20 ns, whose first edge is a falling edge. To run the simulation, set the simulation step to 20 ns and press the Run button. The simulation result is shown in Figure 25.

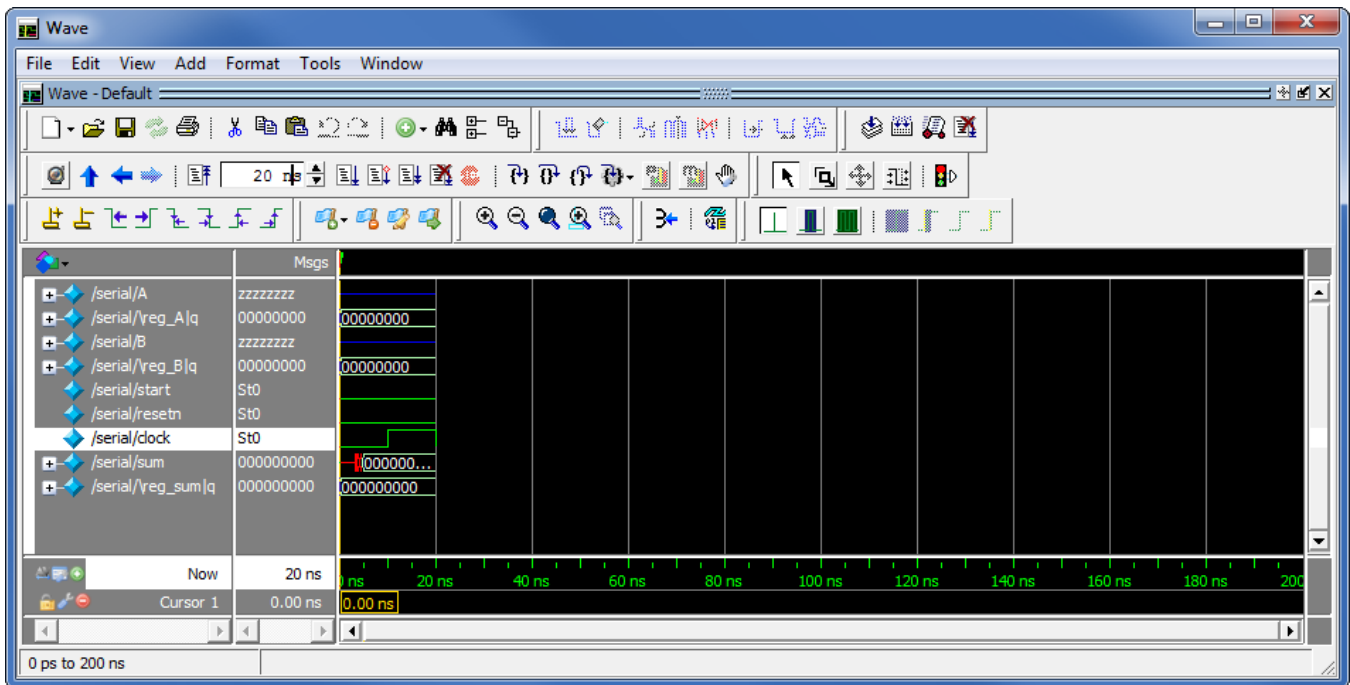


Figure 25. Timing Simulation after 20 ns.

To proceed with the simulation deassert the *resetn* signal by setting it to 1, and apply data to inputs *A* and *B*. Set them to 143 and 57, and assign a value of 1 to the *start* input as described in the Functional Simulation section of the tutorial. Then run the simulation for a period of 20 ns, by pressing the Run button. The simulation result is shown in Figure 26. Remember to change the radix of *A*, *B*, *sum*, and their corresponding registers to *unsigned*.

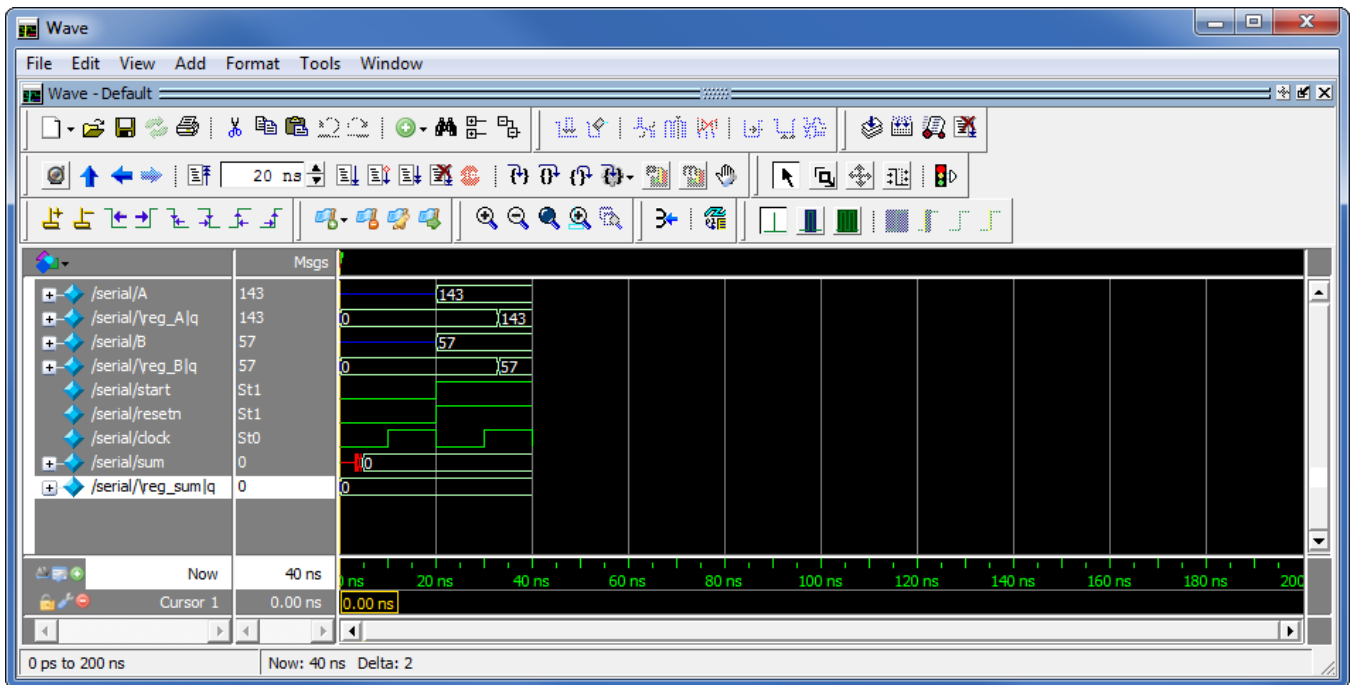


Figure 26. Timing Simulation after 40 ns.

In Figure 26 the data is stored in registers *reg\_A|q* and *reg\_B|q* at the positive edge of the clock. Notice that the simulation indicated that the data in those registers changes some time after the positive edge of the clock, unlike in the functional simulation. The reason for this difference are the delays in the circuit. We can use the zoom buttons to see this more clearly.

When we zoom in on the time when registers *reg\_A|q* and *reg\_B|q* change value, we see the register values change as shown in Figure 27. In the figure, register *reg\_B|q* stabilizes on a value of 57 at time 32918 ps. This is 2918 ps after the positive edge of the clock appeared at the *clock* input. Part of the difference in times between the clock edge and the change of data in register *reg\_B|q* comes from the fact that the clock signal must travel from the input pin on the FPGA device to the registers. The other part of the time is due to the clock-to-Q time of the register, which is the time it takes for a register to change output after it sees a positive edge on the clock input.



Copyright ©1991-2011 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.