Lab 5: Register Transfer Level (Design) – Verilog (Structural way)

Objectives:

- Practice writing Verilog code and testbench for Datapath components
- Practice writing Verilog code (structural way) to connect multiple modules to implement a digital system. Use the testbench to validate your design.

Start: Week 11 (Oct 30-Nov 3, 2023) – attend your scheduled lab session

Demo Due:

- Demo part (A) by the end of your lab session during the **week of Oct 30-Nov 3, 2023**
- Demo part (B) by the end of your lab session during the **week of Nov 6 - 10, 2023** (Friday lab session, Nov 11 is a holiday, each group - please sign up to attend different lab session on the signup sheet sent to your email)

Code submission:

Submit the files that you/your group create or modify on D2L Assignment Dropbox by 11.59 pm on Friday Nov 10, 2023

References: on D2L: Unit Verilog: Verilog_Structural, Verilog_for_Datapath_components, and exercises given in Datapath_example_practice_problems module

Lab Overview: you will implement a digital system for the following algorithm. I already design the Datapath and Controller of this RTL-based digital system. Your tasks are to write Verilog code for each component and to write Verilog code (structural way) to connect these components together to implement the digital system for the following algorithm.

Inputs: go (bit) (byte A[32] is used in the digital system and is implemented using 32x8 Register file)
Outputs: sum (12 bits), done (bit)

```
while(1) {
    while(!go);
    done = 0;
    for (i = 0; i < 32; i++)
    {
        temp1 = A[i];
        i++;
        temp2 = A[i];
        if(temp1 > temp2 )
            sum = sum + (temp1 - temp2);
        else
            sum = sum + (temp2 - temp1);
    }
    done = 1;
}
```

Note: You will learn in lecture that the first step is to draw the High-Level State Machine (HLSM) diagram from the given pseudo code or C-like code above. The HLSM diagram is given on the last page of this handout. Then, using RTL design process, we can design Datapath and Controller for the designed HLSM. As mentioned above, for this lab, I already design Datapath and Controller for you.

Part A) You will

1) Write Verilog code for each component using the given template (download lab5_template.zip on D2L).
2) Write testbench to test each component.

Part B) You will

1) Write Verilog code (structural way) to connect those components together to implement the digital system.
2) Use the given testbench to test your digital system.

The digital system for the given algorithm on page 1 is implemented on page 4. Observe that the following components are used:

- 12-bit register with clear (Clr) and load (Ld) for sum
- 8-bit register with clear (Clr) and load (Ld) for temp1
- 8-bit register with clear (Clr) and load (Ld) for temp2
- 6-bit register with clear (Clr) and load (Ld) for i
- 8-bit comparator to compare temp1 with temp2
- 6-bit comparator to compare i with 32
- 12-bit adder for sum
- 6-bit adder for i
- 8-bit subtractor
- Two 8-bit 2x1 muxes
- Controller
- 32x8 Register file (to implement the array A in the given algorithm). When reset is 1, Register file or A is initialized with

  A = {3 254 22 131 15 250 62 135
  27 246 102 139 39 242 142 143
  51 238 182 147 63 234 222 151
  75 230 6 155 87 226 46 159}

Part A: Get the code template from D2L (lab5_template.zip). Write Verilog code and testbench for each component.

1) **32x8 Register file** (RegisterFile_32_8.v) code is given – you can use it
2) **8-bit subtractor** (Subtractor_8bits.v) is given – you can use it
3) **(7 points) 8-bit 2x1 mux**. You will ***write this code*** using comments in the template. You will then ***write a testbench*** to test your 8-bit 2x1 mux. Use test cases for inputs in ***Figure 3 (page 6)*** in your testbench.
4) **(7 points) Comparator:** the given template is for 8-bit comparator. You have two options (**pick one**)

   Option 1)
   a) Write ***Verilog code*** for 8-bit comparator using the given template
   b) Create new .v file and write Verilog code for 6-bit comparator (the name of this module must be different from a))

   Option 2)
   Write ***Verilog code*** for 8-bit comparator. In part B, when you instantiate the module for 6-bit comparator, make sure that you perform port mapping correctly ({ } can be used to create a vector signal with larger size to be used with 8-bit module)

After you write the code, ***write a testbench*** to at least test your 8-bit comparator. Use test cases for inputs in ***Figure 4 (page 6)*** in your testbench.

5) **(7 points) Adder:** the given template is for 12-bit adder. You have two options (**pick one**)
   Option 1)
   a) ***Write Verilog code*** for 12-bit adder using the given template
   b) Create new .v file and write Verilog code for 6-bit adder (the name of this module must be different from a))
   Option 2)
   Write Verilog code for N-bit adder. Use parameter N = 12. In part B, when you instantiate the module for 6-bit adder, use a proper way to reassign N.

After you write the code, ***write a testbench*** to at least test your 12-bit adder. Use test cases for inputs in ***Figure 5 (page 6)*** in your testbench.

6) **(7 points) Register**: the given template is for 12-bit register. You have three options (**pick one**)
   Option 1)
   a) ***Write Verilog code*** for 12-bit register using the given template
   b) Create new .v file and write Verilog code for 6-bit register (the name of this module must be different from a))
   c) Create new .v file and write Verilog code for 8-bit register (the name of this module must be different from a) and b))
   Option 2)
   Write Verilog code for N-bit register. Use parameter N = 12. In part B, when you instantiate the module for 6-bit register or 8-bit register, use a proper way to reassign N.
   Option 3)
   Write Verilog code for 12-bit register using the given template. In part B, when you instantiate the module for 6-bit register or 8-bit register, make sure that you perform port mapping correctly ({ } can be used to create a vector signal with larger size)

After you write the code, ***write a testbench*** to at least test your 12-bit register. Use test cases in ***Figure 6 (page 6)*** in your testbench.

7) **(12 points)** Write ***Verilog code (behavioral way)*** for **Controller**. See its state diagram in Figure 2 (page 5).
   In Controller, encode the state SA, SB, SC, …, SJ using 0,1,2, …, 9.
   Note: the way to write this code is exactly the same way that we use to describe a state diagram of sequential circuits (lab 3 and lab 4)

After you write the code, ***use the given Controller_tb.v*** file to test your Verilog code. If working, the outputs from your circuit should be exactly as shown in ***Figure 7 (page 6)*** in your testbench.

Part B (60 points):

a) Add .v files in part A into the Vivado project. Add lab5.v into the project and write (complete) lab5.v to connect components together as shown in Figure 1 (page 4).
b) Use the lab5_tb.v to test your digital system. Run the simulation waveform until you see the done signal equals to 1. If your system works correctly, when done = 1, sum should 2148. See the simulation waveform in Figure 8 (page 7).
   Note: you MUST add the following internal signals into your behavioral simulation waveform. 8-bit temp1, 8-bit temp2, temp1_gt_temp2, the 12-bit output from the 12-bit adder, 4-bit state (of Controller), 6-bit i, (Register file) 5-bit R_Addr, R_en, 8-bit R_Data.

   See the last page of this handout on how to add internal signals.

c) After your behavioral simulation waveform works correctly, run synthesis and generate post-synthesis functional simulation waveform.

Figure 1. **Note: all components using clock signal are connected to the same clock input signal. No wire is drawn in the system below (to make it easier to read).**

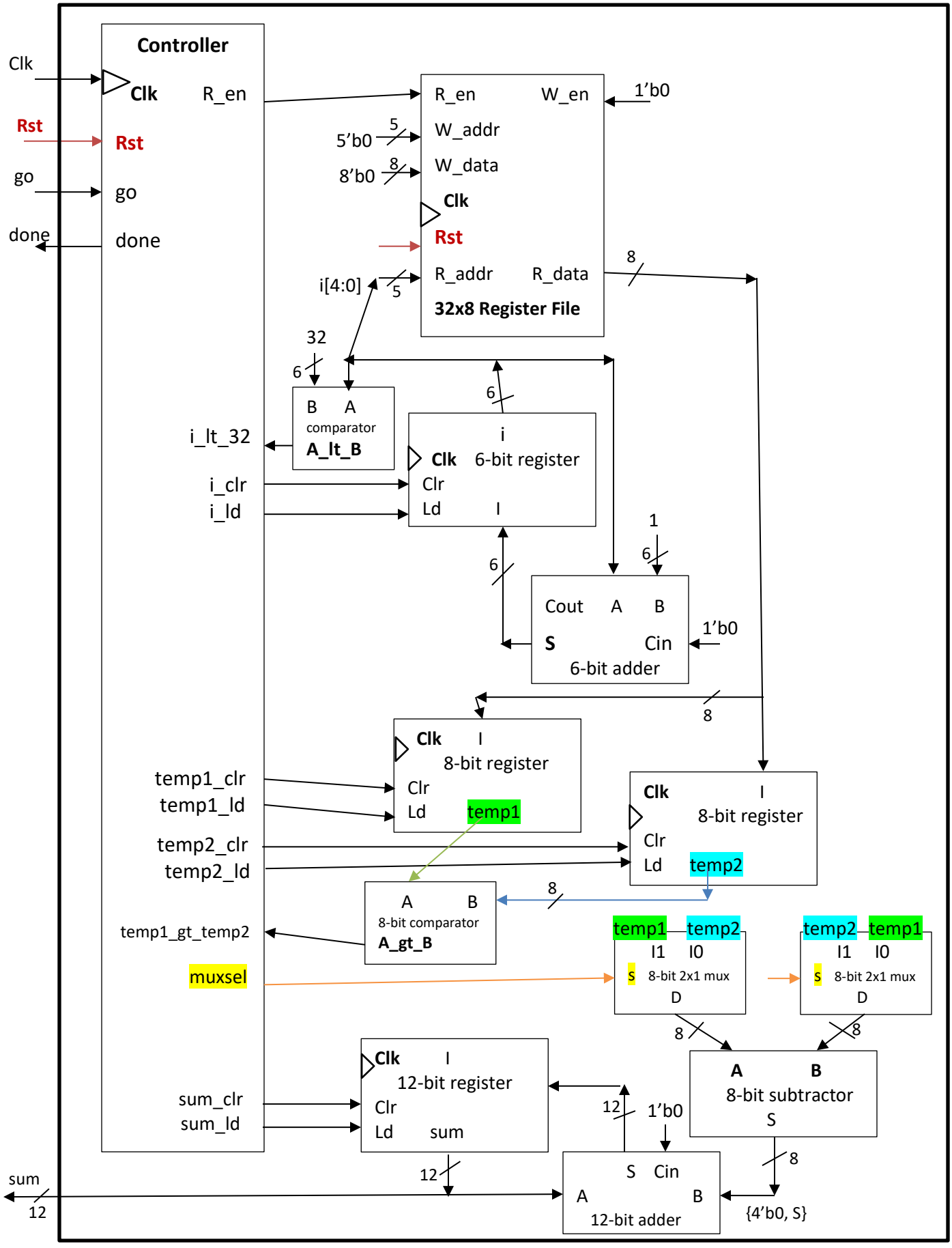


Figure 2: State diagram of Controller

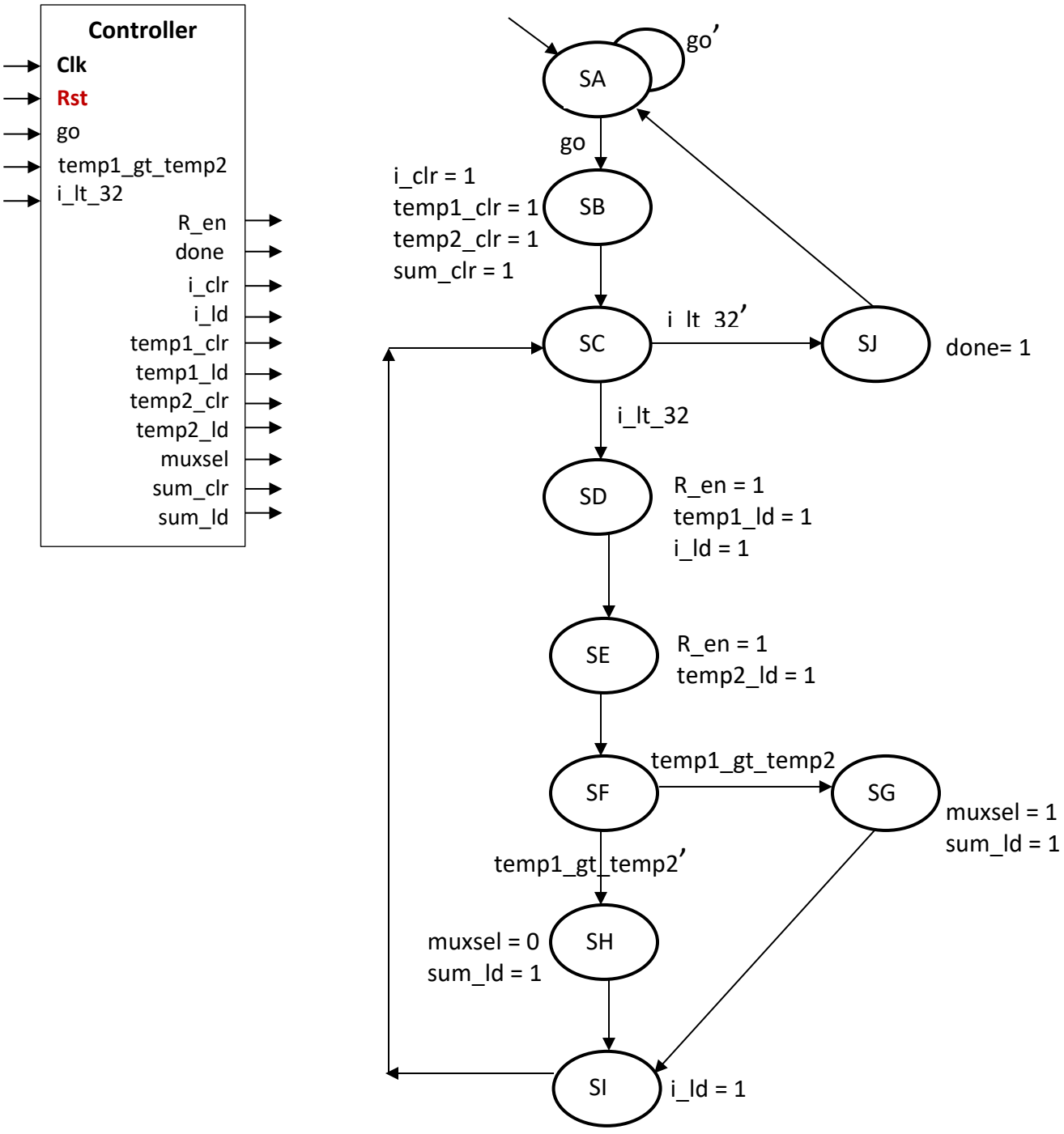


Figure 3: Simulation waveform for 8-bit 2x1 mux

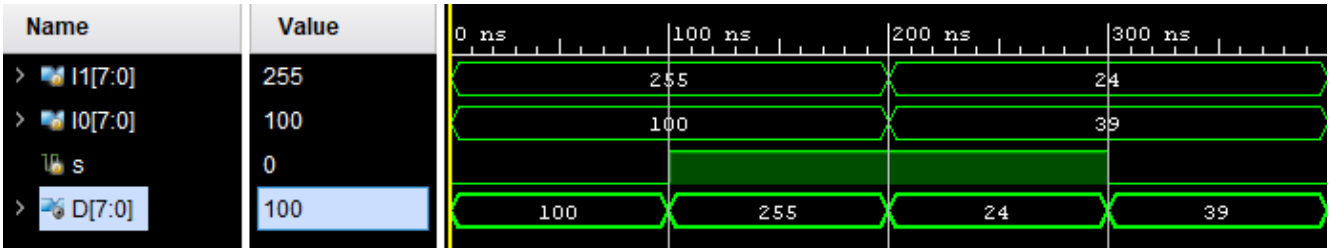


Figure 4: Simulation waveform for 8-bit comparator

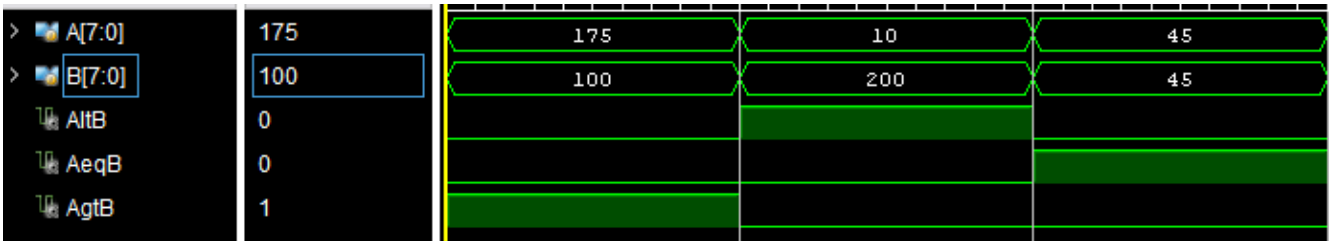


Figure 5: Simulation waveform for 12-bit adder

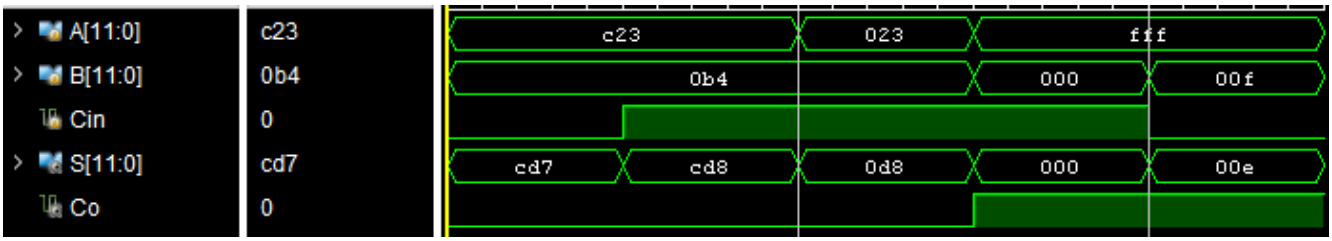


Figure 6: (behavioral) Simulation waveform for 12-bit register

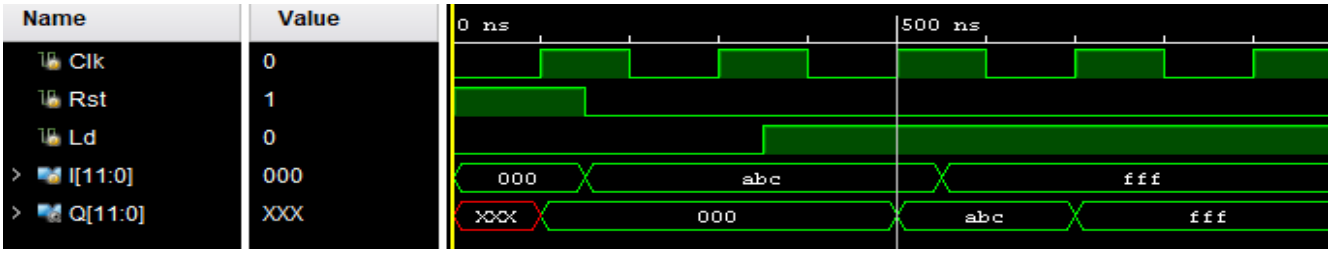


Figure 7: (behavioral) Simulation waveform for Controller. State is the internal signal added to the waveform below. The state SA, SB, SC, …, SJ are encoded using 0,1,2, …, 10, respectively.

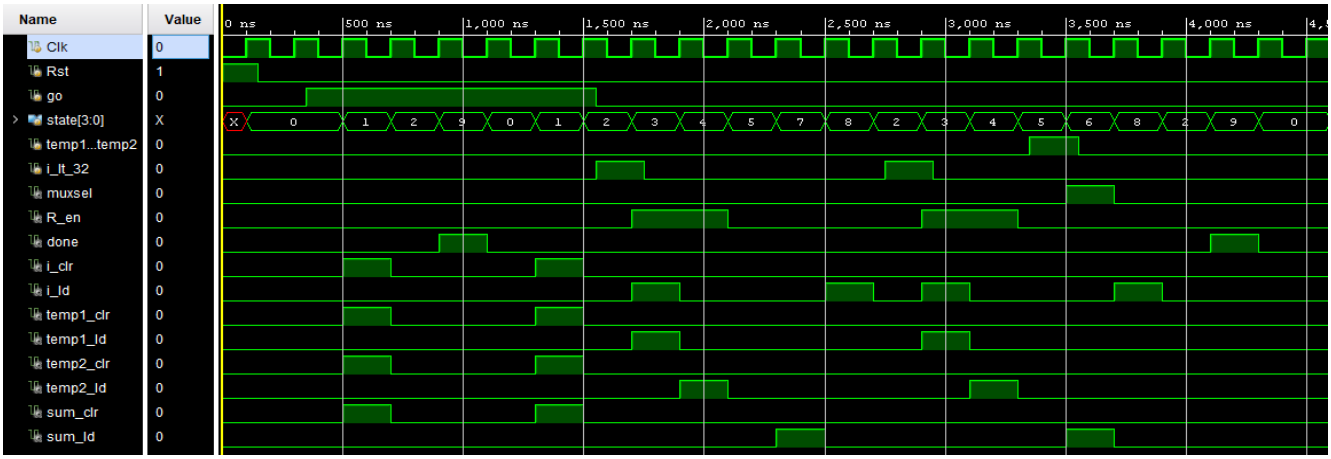
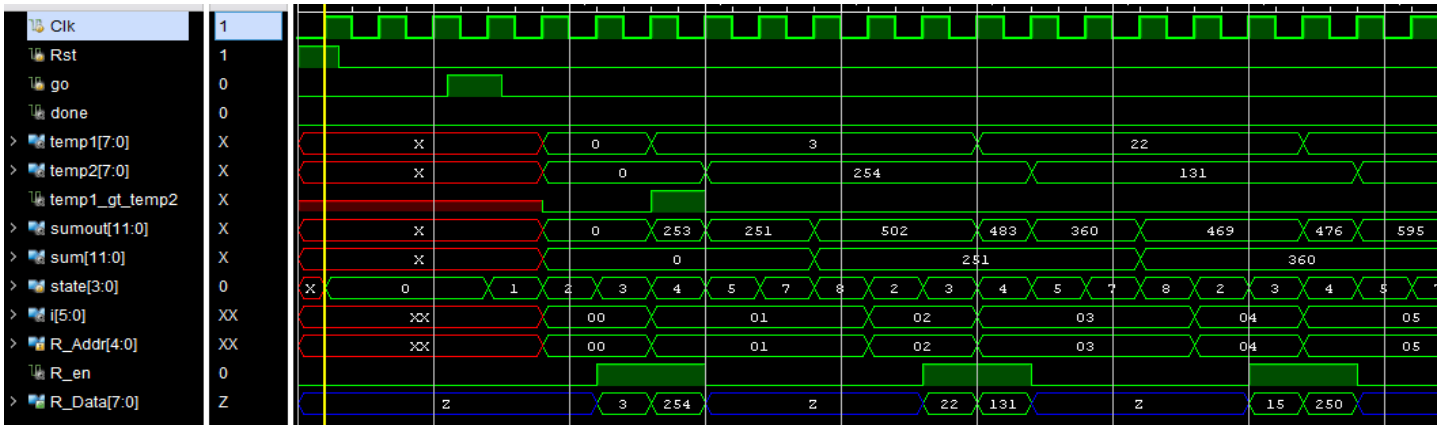


Figure 8 Behavioral simulation of Digital System (Part B)

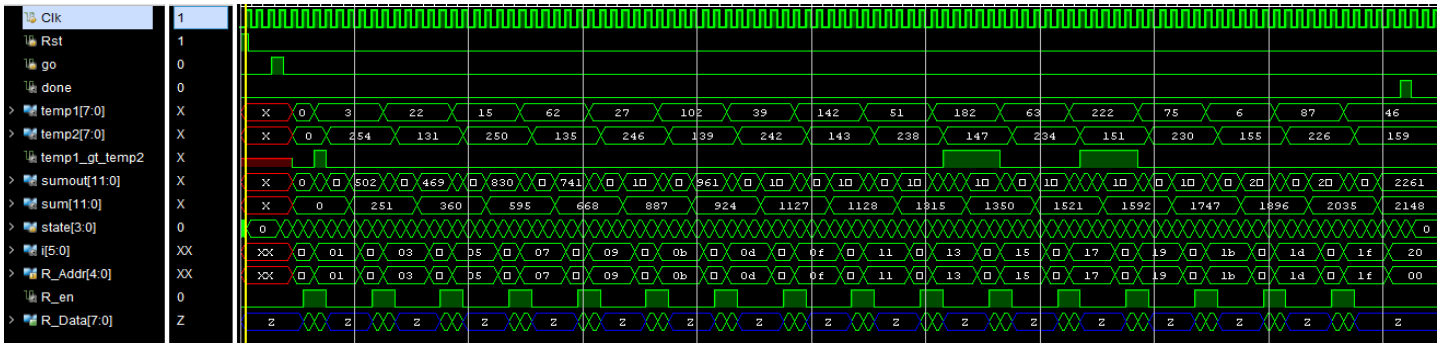


The internal signals are added to the above waveform: 8-bit temp1, 8-bit temp2, temp1_gt_temp2, sumout (12-bit output from the 12-bit adder), 4-bit state (of Controller), 6-bit i, (Register file) 5-bit R_Addr, R_en, 8-bit R_Data.

In controller, if you encode state SA, SB, SC, …, SJ using 0,1,2, …, 10, you will see, in your waveform, the state values exactly as shown in the given waveform above.

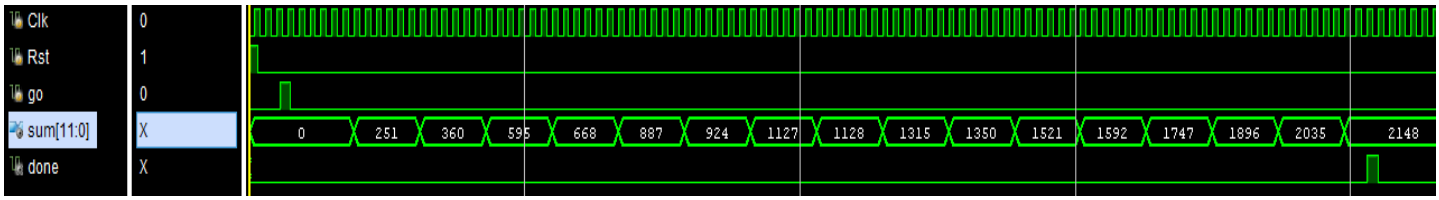
Observe that 1) sumout changes as the inputs to the "sum" adder change but the sum register only gets the value when sum_ld = 1 at the positive edge of clock signal. 2) when R_en is 1, content of register file at R_Addr (also i) is available on R_data.

Complete Behavioral simulation of Digital System (Part B)



When done = 1, sum should be 2148 for the given set of values in the given Register file.

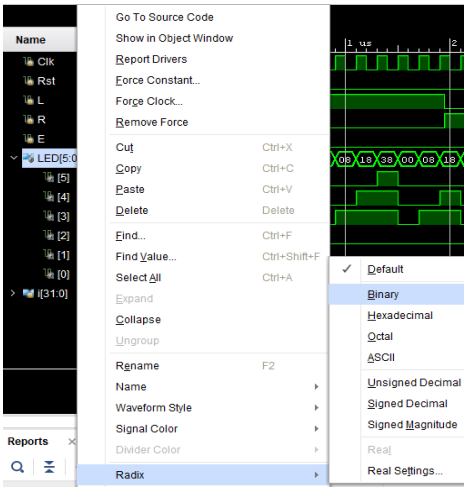
The **post-synthesis** waveform will show only done and sum. If correct, it should be exactly as shown below – it is also the same waveform that you get in the behavioral simulation.



Additional information

a) To change the base number (Radix) for the vector signal

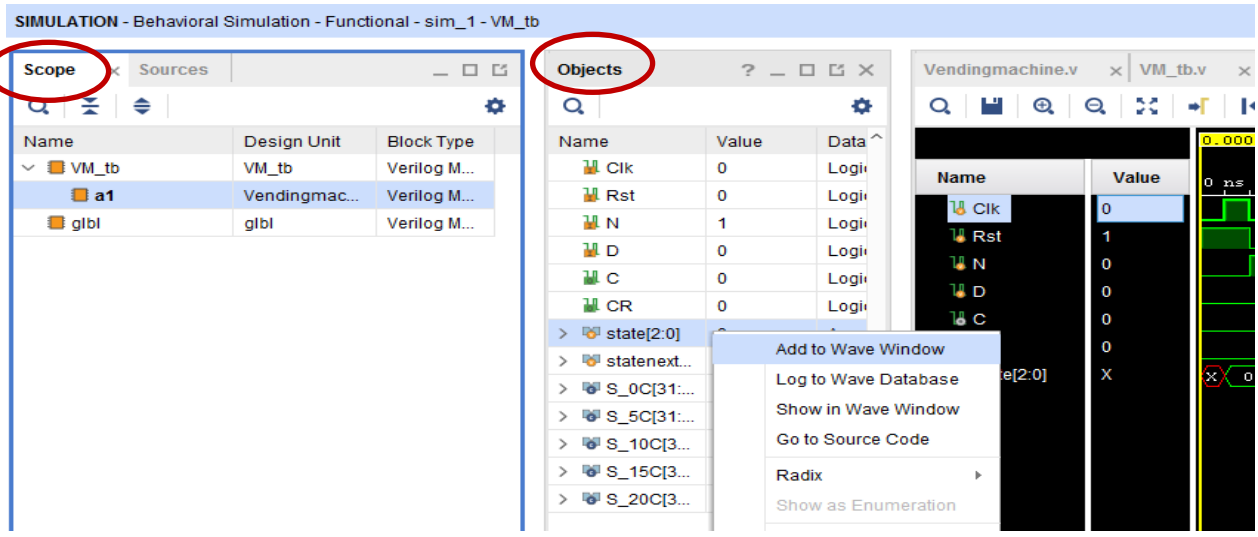
Right-click on the vector signal, choose Radix, then choose the base-number that you want to use. In this lab, we will use unsigned decimal since Number is used to represent number of cents that a user enters.



b) To add internal signal (such as state) to the waveform in *Behavioral Simulation*

On "**Scope**" window, click > to expand so that the module that you want to add the signal is shown

On "**Objects**" window, **Right-click** on the signal that you want to add to the waveform and choose Add to Wave Window.



HLSM diagram

Inputs: go (bit)
Outputs: sum (12 bits), done (bit)
Local storage: sum(12 bits), i (6 bits), temp1 (8 bits), temp2 (8 bits)

```
while(1) {
    while(!go);
    done = 0;
     for (i = 0; i < 32; i++){
          temp1 = A[i];
          i++;
          temp2 = A[i];
          if(temp1 > temp2 )
               sum = sum + (temp1 - temp2);
          else
               sum = sum + (temp2 - temp1);
    }
    done = 1;
}
```