Team Kublinh
Kubilay Agi 304784519
Baolinh Nguyen 104732121
Lab 1
January 31, 2018

Workshop 2

1. In the tb.v file, instructions are sent to the task tskRunInst where the sw variable is set to the corresponding instruction number to be run. This is used in the nexys3.v to send instructions to the sequencer. We know this because each of the tasks apart from tskRunInst load a certain bit pattern depending on the instruction and the registers that are being used into the input register (inst) for taskRunInst. tskRunInst then receives this instruction via the input register and then executes the instruction (puts it into sw).
2. In this process, to send the instruction, the user presses the center button. To reset all register values, the user can press the reset button, which in this case, is set to the right button. The user also sets all of the switches in order to indicate the instruction that they want to be executed. Additionally, the tasks that are called by the user in this process are tskRunADD, tskRunMULT, tskRunSEND, and tskRunPUSH. The rest is handled by the tskRunInst task and other verilog code.

Workshop 1

Clock Dividers:

1.

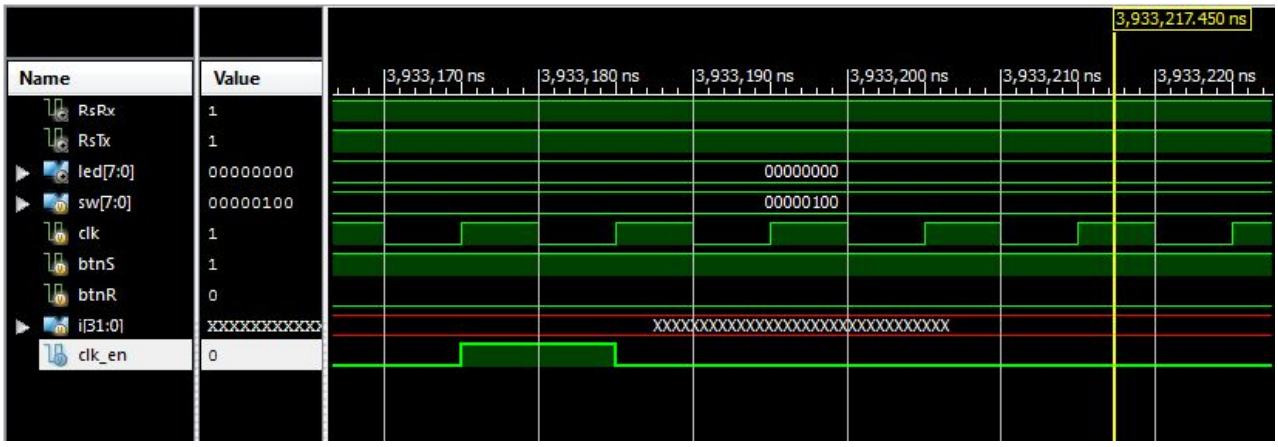

Figure 1: The above screenshot shows the clk_en signal changing from low to high to low again. This image was captured after zooming into the chart to see the period of the clk_en signal.

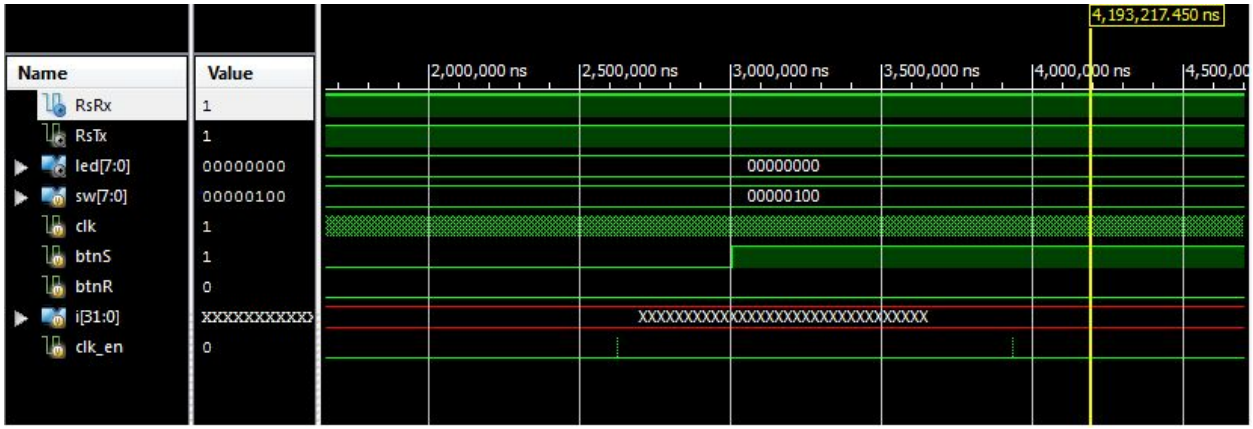

Figure 2: The above screenshot shows the clk_en signal (the very bottom plot), but this time zoomed out much further to give a sense of the brevity of the signal. From the plot, we observe that the two vertical dotted lines on the bottom line are the high-signals from clk_en.

Note the amount of time that is between the two peaks. To find the differences, we employed a feature of iSim which allows us to use the left and right arrow keys to find precisely the positive and negative edges of the high signals for clk_en. We used the positive edges from two consecutive high signals. From this, we can determine the period of the clk_en signal which we calculated to be:

Period = 3933175-2622455 ns = 1310720 ns.

2. To calculate the Duty Signal, we looked at the formula given to us: $D = \frac{T}{P} \times 100\%$. We calculated P previously to be 1310720 ns. Our observed value for T was 10ns. Thus, for D, we see that we have a duty cycle of 0.000763%. This number goes along with the comment in the source code for this project which states that these signals are used to implement the "763Hz timing signal for clock enable".
3. From the screenshot below, we observe that the clk_dv signal is 0 when clk_en is high. This information tells us that clk_en could be used as a form of terminal count signal for this module.

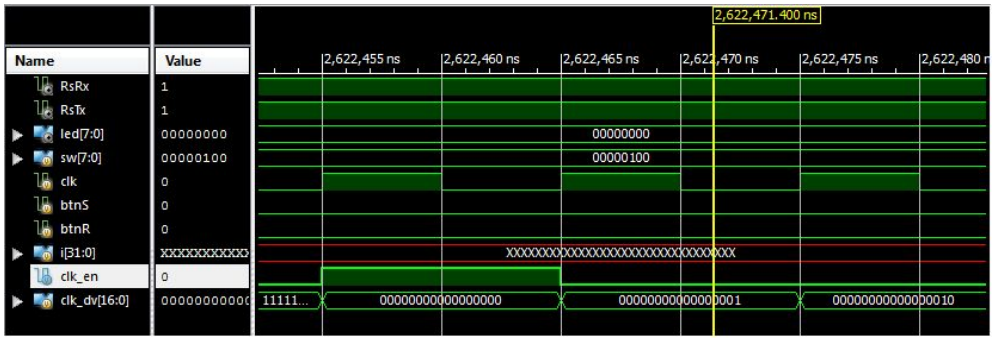

Figure 3

4. The following is the schematic for the implementation of the signals that were referenced in this section. One should note that there were limitations when drawing this diagram. The program that was used to create this schematic is Logism. Logism did not allow to split the wires in the way that they should be in this implementation. Therefore, we get the orange wire indicators which mean that the sizes do not match between the hardware input and the wire. However, the wires are labeled with the signals that they carry to indicate which signals go where.

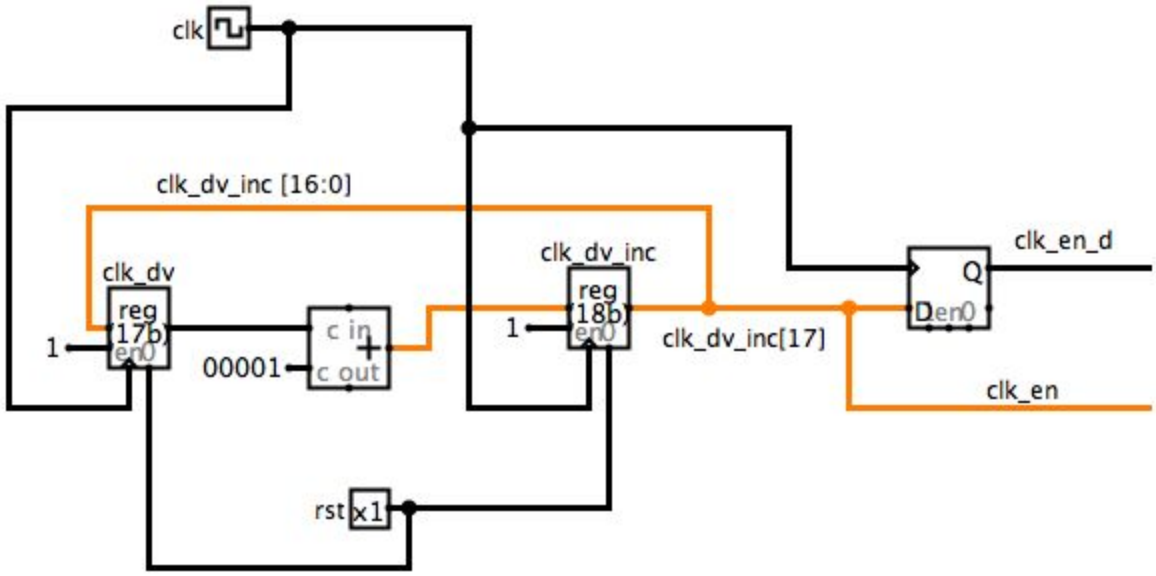

Figure 4

Debouncing:

1. The clk_en_d is offset from the clk_en signal by 1 clock cycle, ensuring that only a single press is registered once, as it should be. This prevents the ripple effect and effectively debounces the signal, preventing the signal from the button from being registered multiple times unknowingly. If we use clk_en in this statement, it may be triggered incorrectly. Thus, we use clk_en_d inside ~step_d[0] & step_d[1] & clk_en_d to delay the signal and ensure that it is properly debounced.

2. If change the value of clk_en to take on the value of clk_dv[16], then the duty cycle will be 50%. This is because when we count to a certain binary number, the bit that is one degree less significant than the most significant bit of the number we are counting to remains high for half the time. Essentially, to count to a certain number, we need to count to half of that number twice over. In the case of clk_dv, the 17th most significant bit will be 0 for the first time that we count to 2^16 and then it will be 1 for the next time we

count to 2^16. And after we have counted to 2^16 twice, we get 2^17, which is the number at which clk_dv_inc resets back to zero.

3. The above screenshot below shows the relationship between clk_en, step_d[1], step_d[0], btnS, clk_en_d and inst_vld.

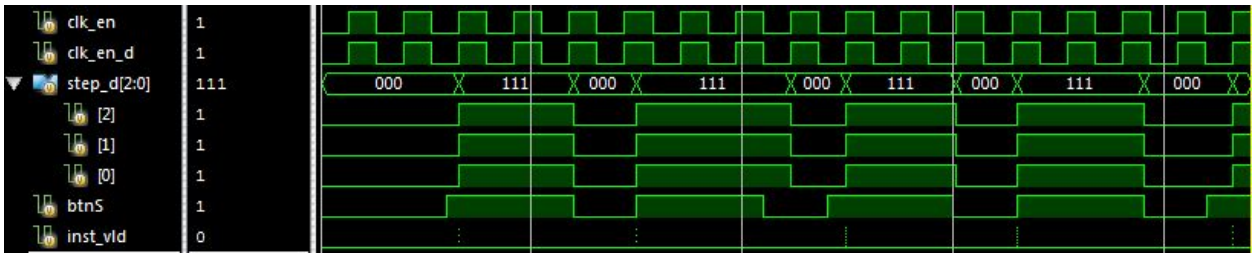

Figure 5

4.

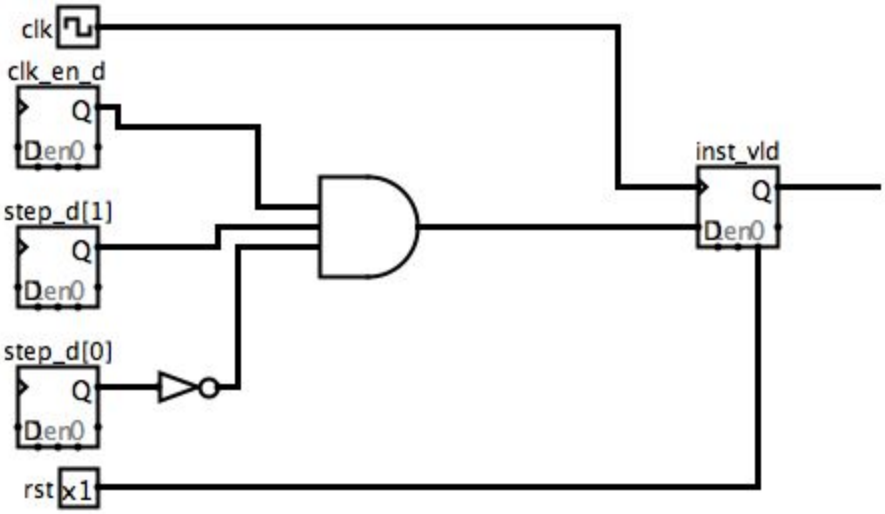

Figure 6

The screenshot above illustrates the circuit for the signals clk_en_d, step_d, rst, and clk. The combinational logic shown above mirrors the source code for this section of the program. These signals come together and are stored in a D flip-flop as a way of indicating whether the instruction that we wish to run is valid.

Register File

1. This line uses sequential logic, shown by the non-blocking assignment (<=) and the always loop that is triggered by posedge clk.

Figure 7:

```
26      always @ (posedge clk)
27        if (rst)
28          begin
29             for (i=0;i<seq_num_regs;i=i+1)
30               rf[i] <= 0;
31          end
32        else if (i_wstb)
33          rf[i_wsel] <= i_wdata;
```

2. This segment of code uses combinational logic, as shown in the assign statements. For this logic, we would need to use memory to act as registers and a way of selecting which register to write to. In order to implement this design, I would use D flip-flops to act as the registers for each rf[i] and to select which one to write to, I would either use demultiplexers or binary decoders. However, with the decoder, we would need to have a counter as well, which would require more hardware and is therefore more expensive in terms of what hardware is needed and in delay time. This means that we should probably use demultiplexers. The select signals in Figure 8 would be the select inputs for multiplexers. In this way, we can assign the output data values with the appropriate values from the registers.

Figure 8:

```
35      assign o_data_a = rf[i_sel_a];
36      assign o_data_b = rf[i_sel_b];
```

3. The screenshot below shows the first time that register 3, rf[3] is written with a non-zero value.

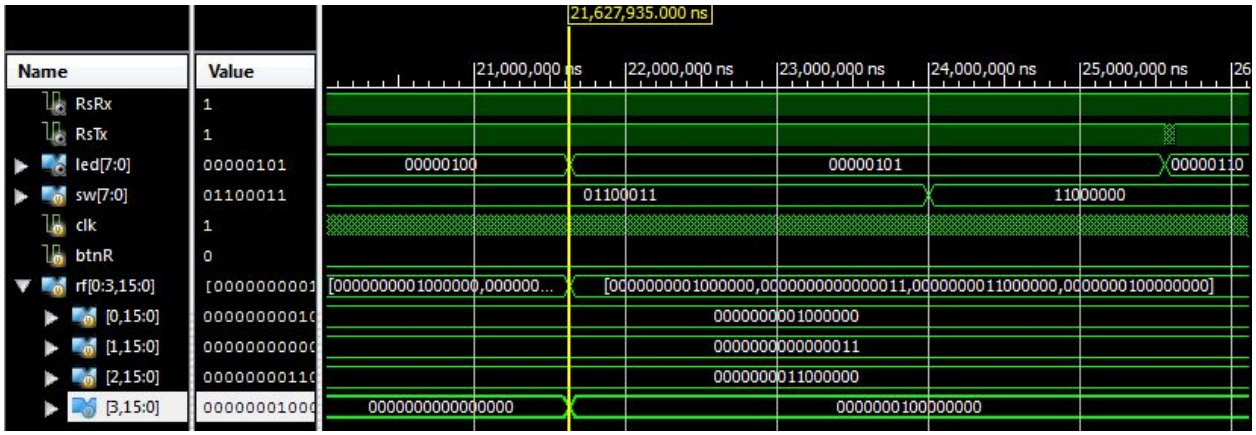

Figure 9

4. Schematic for seq_rf registers: Here we implement the logic that we came up with in Question #2 of this section.

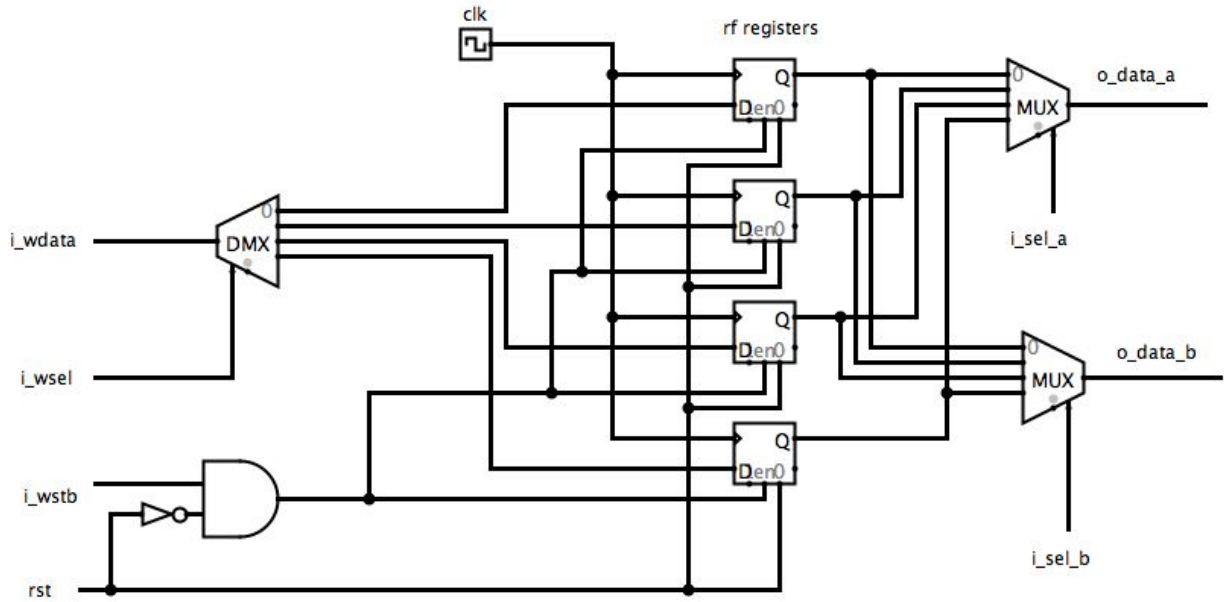

Figure 10