Eddie Huang, Lucas Jenkins, Jason Less
CS M152A, Lab 3
TA: Hongxiang Gu

Lab 1 Report

Clock Dividers

Question 1:

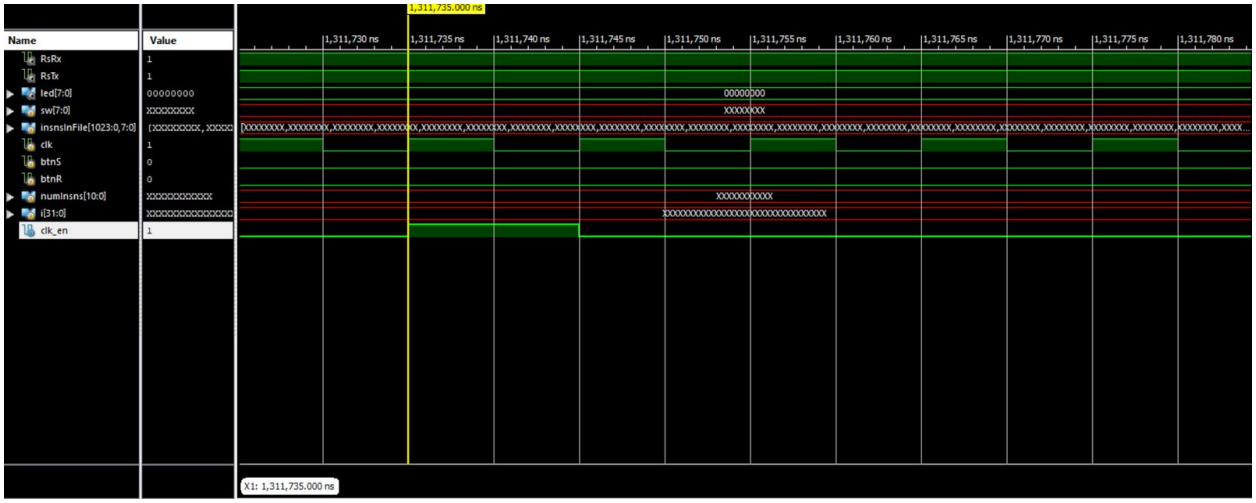

Figure 1: Positive edge of the clock signal for clk_en, occurs at time t1 = 1,311,735.000 ns.

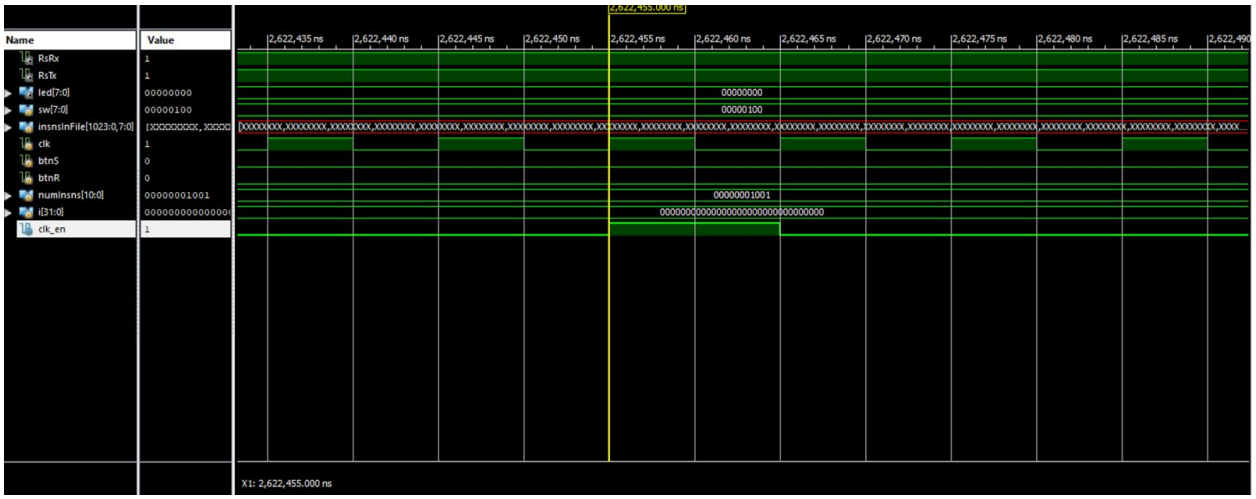

Figure 2: Positive edge of the clock signal that immediately followed the clock signal shown in Figure 1, occurs at time t2 = 2,622,455.000 ns.

Period: t2 - t1 = 1,310,720 ns

Question 2:
T = 1310720 ns
P = 10 ns
D = (T/P) * 100% = 0.00076%

Question 3:

The value of clk_dv during the clock cycle that clk_en is high is 16'b0 (a 16 bit binary 0). This is because clk_en is only high during the clock cycle that clk_dv_inc is 2^17 i.e. 17'b10000000000000000, and clk_dv gets the lower 16 bits of clk_dv_inc.

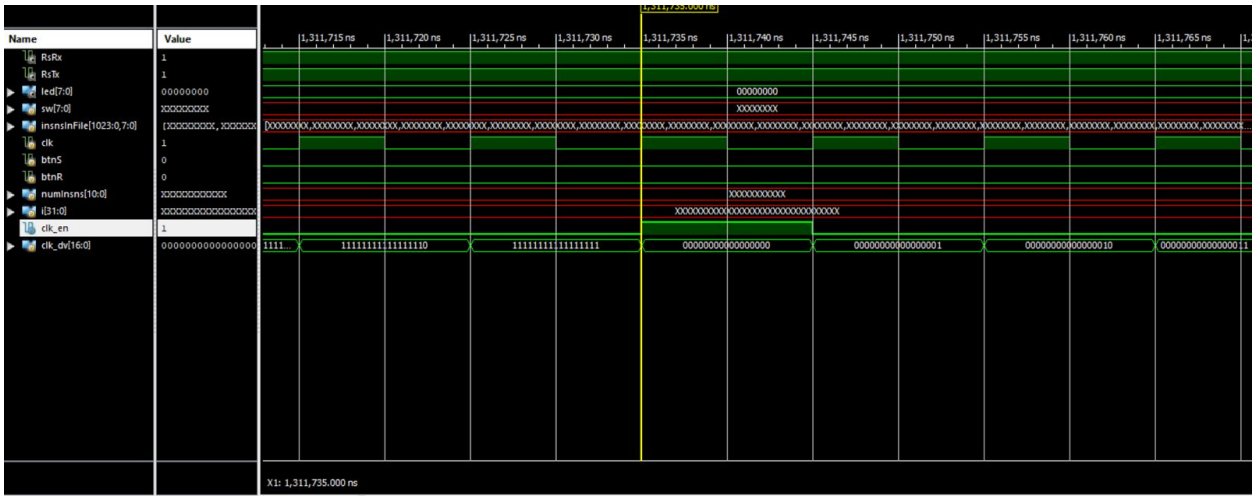

Figure 3: The value of clk_en is high when clk_dv is zero.

Question 4:

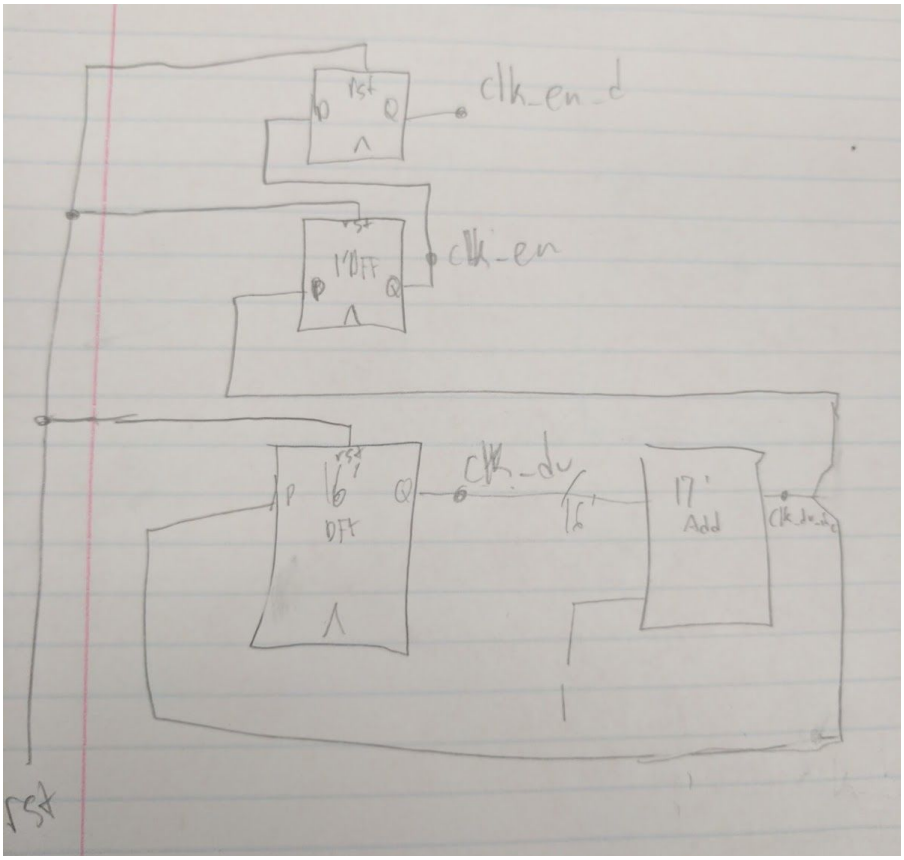

Figure 4: A simple schematic that shows clk_en_d, clk_en, and clk_dv translated from the Verilog code provided.

Debouncing

Question 1:
The purpose of clk_en_d signal in the expression ~step_d[0] & step_d[1] & clk_en_d is so that the inst_vld signal is only high for one clk period of every clk_en period i.e. so that instructions are only sent a maximum of once for every clk_en cycle if there is a positive edge of the btn signal. We don't use clk_en because if we did, the step_d register would update before we were able to make inst_vld high.

Question 2:
No, we cannot simply use the 16th bit of clk_dv for clk_en to make the duty cycle of clk_en 50% because the 16th bit of clk_dv is high for half of the current period of clk_en, thus the duty cycle of clk_en would be 50% instead of 1 / (2^{17}) of the time as it is now. This would invalidate all the instruction sending signals that depend on clk_en to only be high once every period of clk_dv.

Question 3:

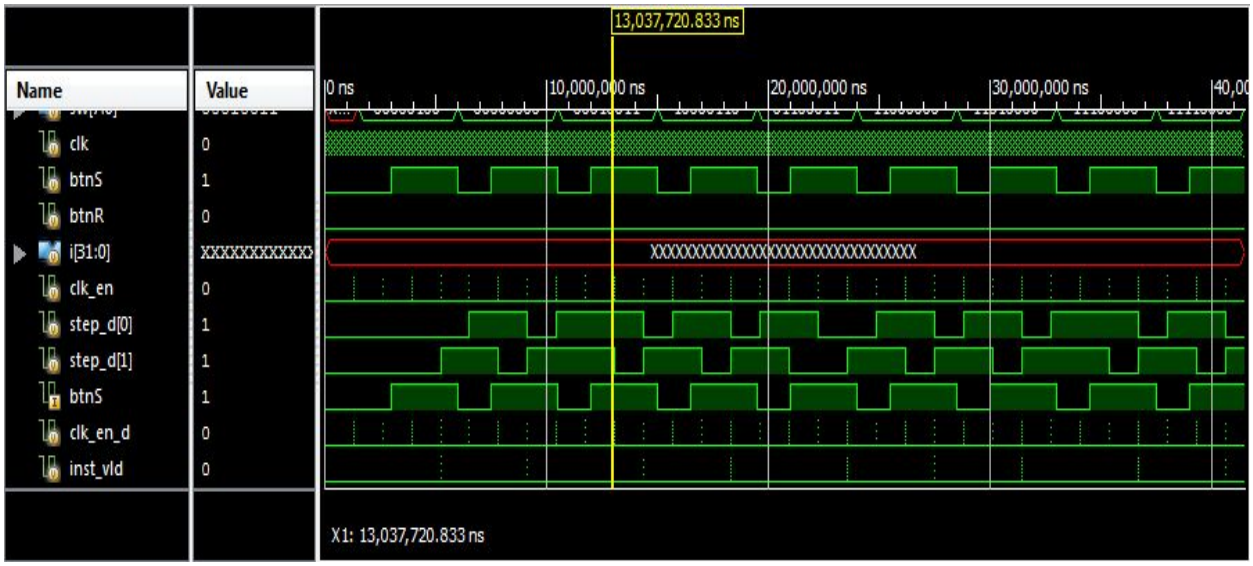

Figure 5: Waveform highlighting the signals: clk_en, step_d[1], step_d[0], btnS, clk_en_d, and inst_vld.

In this waveform, btnS is an asynchronous input signal. Step_d[1] and step_d[0] can only switch on every clk_en_d (which is clk_en delayed by 1 clock cycle), and the values of step_d[1] and step_d[0] are determined by the value of btnS 1 and 2 cycles of clk_en_d before, respectively. Inst_vld is only high when clk_en_d and step_d[1] are high AND step_d[0] is low. Essentially, step_d[1:0] are using clk_en_d as a less frequent clock signal for debouncing purposes.

Question 4:

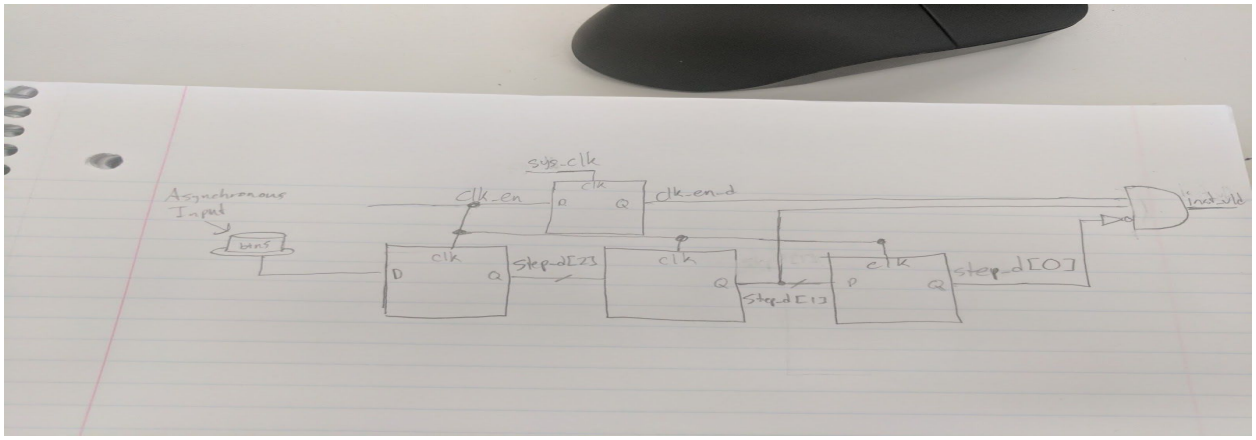

Figure 6: A simple schematic that shows the relationship between the signals in the waveform of Figure 6.

Register Files

Question 1:
rf[i_wsel] <= i_wdata;

The above line of code is where registers are given a non-zero value. As this code is located within an always block, which is set to trigger in response to a clock edge, this is is sequential logic. Additionally, the fact that we are assigning to a register means that the circuit stores state, i.e. it is a sequential circuit.

Question 2:

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

The above lines of code are where the selected register values are assigned to the given outputs. As it is not reliant on any clock signal, this code is achieved using combinational logic. We would implement this logic using a multiplexer for each of o_data_a and o_data_b, as the output to a given signal depends on the select signal.

Question 5:

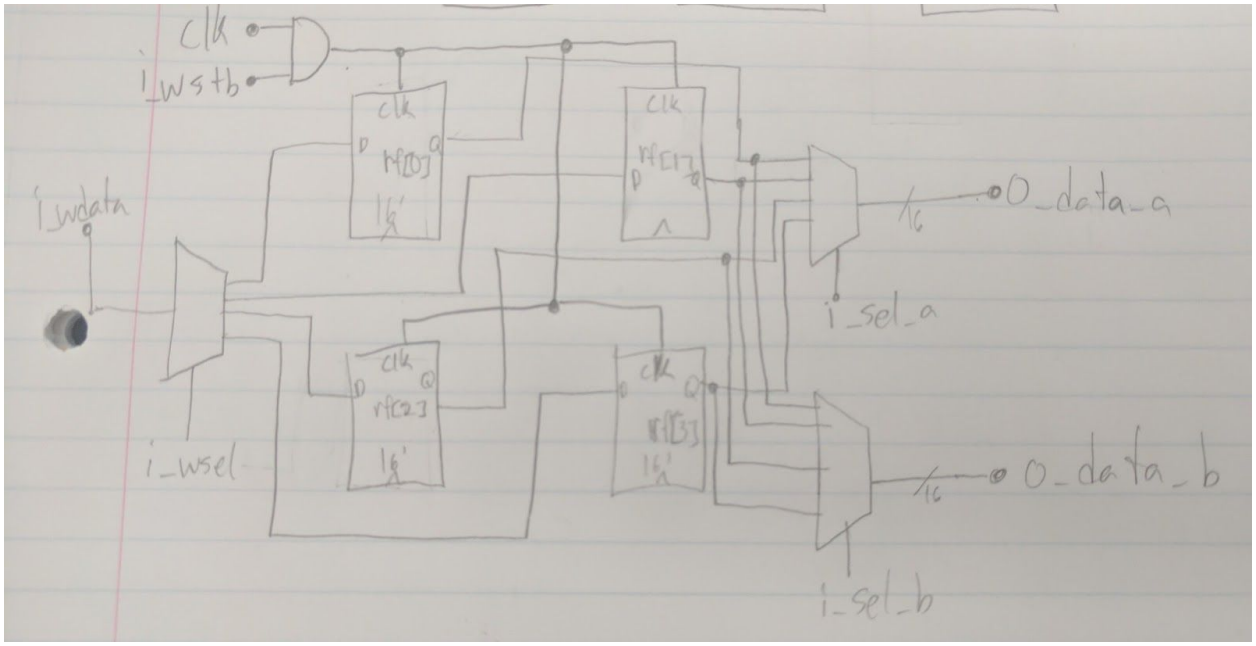

Figure 7: A simple schematic of the circuit of the register file block.

Question 3:

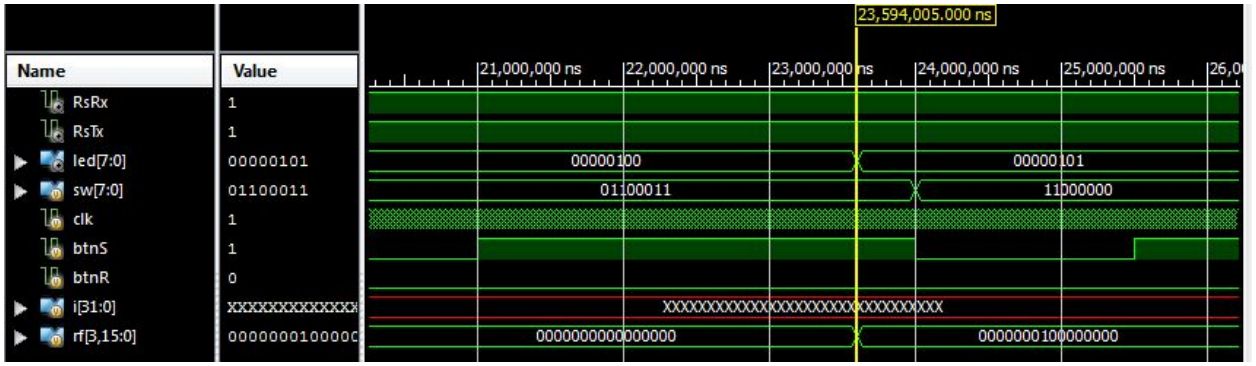

Figure 8: Waveform that illustrates the first time that register 3 has a nonzero value.

Workshop 2:

Nicer UART Output:

The original version of model_uart.v announced every byte that was received by the model. For this portion of workshop 2, the goal was to get the output to announce one line at a time of bytes, and then go the a new line once a carriage-return character (‘\r’) was received. This was implemented by creating a new register variable that was 4 bytes long, and would only print out the bytes received to the screen once a newline character (ASCII value 10) was received. Otherwise, the byte received would be concatenated to the larger register, but only if the carriage return character was not sent over. Both the generic, and the modified outputs are shown on the next page:

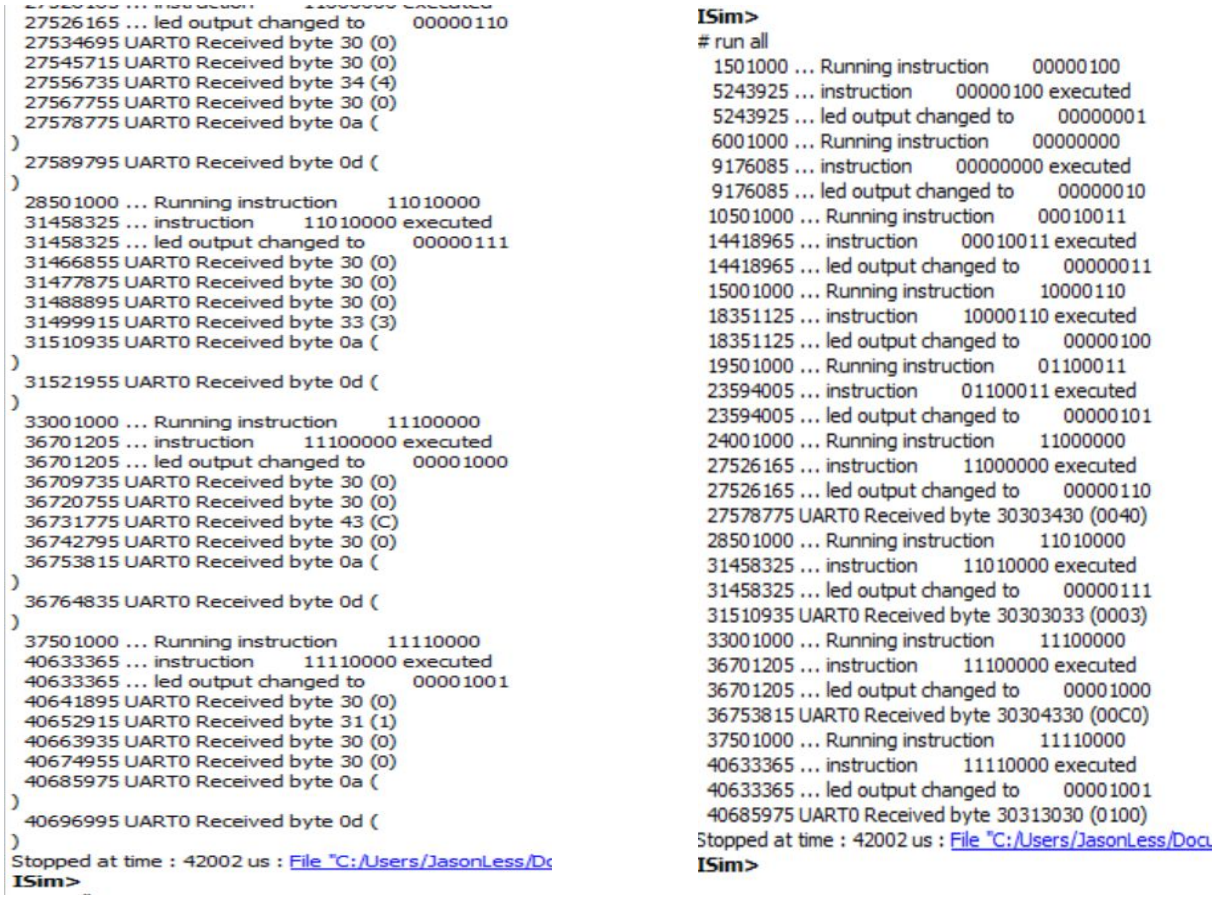

Figure 9: (Left: Depicts the generic (i.e. 1 byte per line) output, Right: Depicts the modified (bytes until '\r') output)

<u>An Easier Way to Load Sequencer Program:</u>

The original implementation of the program has a set of instructions (tasks) that are hard-coded into the test bench files. For this part of the workshop, the goal was to make use of simple file I/O to avoid hard-coding the tasks, and instead read a list of instructions from a given file. This could be achieved via the built-in Verilog system task $readmemb, or the combination of $fopen and $fscanf. We chose to implement this part using the $readmemb system task, and a simple for-loop to process each instruction of the specified file using the provided tskRunInst function. This method achieved the desired byte output representation as depicted in Figure 9.

Fibonacci Numbers:

The last part of this workshop involved using the methods created in the previous section (i.e. read in from a file) to read a sequence of instructions to print out the first 10 numbers of the Fibonacci series. This was done by storing initial conditions of the series in each of the four registers, and then continuously adding registers together to get the next value in the Fibonacci series. In addition, the first line of the file (called "fib.code") was the number of instructions represented in binary. Below is the binary representation of the instructions to execute the given task, as well as a small snippet of the output of the series:

```
1   10100
2   00000000
3   00010001
4   01010110
5   01011011
6   11000000
7   11010000
8   11010000
9   11100000
10  11110000
11  01101100
12  01001101
13  01000110
14  01011011
15  11000000
16  11010000
17  11100000
18  11110000
19  01101100
20  11000000
```

```
62915605 ... led output changed to      00001110
62968215 UART0 Received byte 30303035 (0005)
64501000 ... Running instruction      11010000
68158485 ... instruction      11010000 executed
68158485 ... led output changed to      00001111
68211095 UART0 Received byte 30303038 (0008)
69001000 ... Running instruction      11100000
72090645 ... instruction      11100000 executed
72090645 ... led output changed to      00010000
72143255 UART0 Received byte 30303044 (000D)
73501000 ... Running instruction      11110000
77333525 ... instruction      11110000 executed
77333525 ... led output changed to      00010001
77386135 UART0 Received byte 30303135 (0015)
78001000 ... Running instruction      01101100
81265685 ... instruction      01101100 executed
81265685 ... led output changed to      00010010
82501000 ... Running instruction      11000000
86508565 ... instruction      11000000 executed
86508565 ... led output changed to      00010011
86561175 UART0 Received byte 30303232 (0022)
87001000 ... Running instruction      xxxxxxxx
90440725 ... instruction      xxxxxxxx executed
90440725 ... led output changed to      00010100
Stopped at time : 91502 us : File "C:/Users/JasonLess/D
```

Figure 10: (Left: The instructions of "fib.code", Right: Byte output of the values 5, 8, 13, 21, and 34 in the Fibonacci series)