CS 152A Lab 1: Sequencer

Gee Won (Jennifer) Jo, Andrew Lin, Daxuan (Albert) Shu
704171725, 204455444, 204853061
CS 152A Lab 5
April 27th, 2017

Introduction

In this lab, we will understand a small scale FPGA project and make modification of it, study the Verilog module and test bench, and learn the styles, formats, structures of codes and useful techniques, like clock dividers and debouncers.

Workshop 1

Clock Dividers

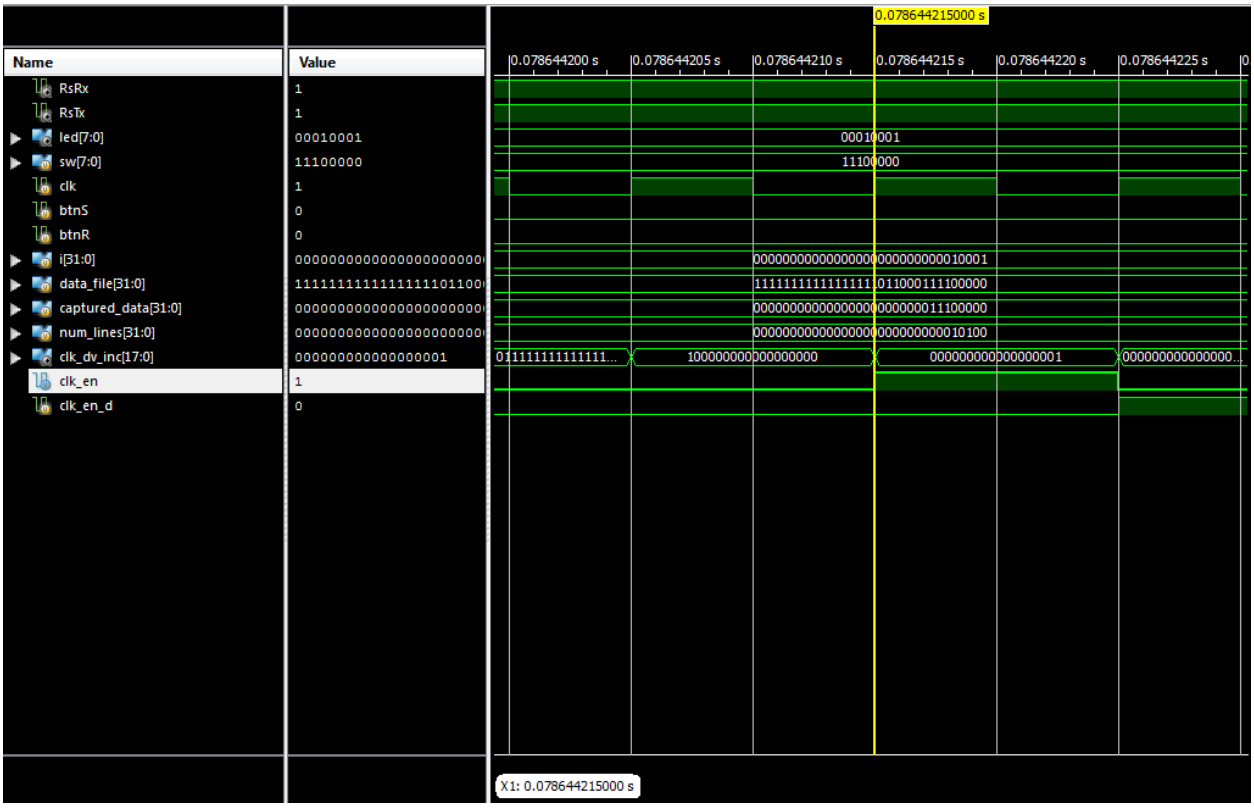
```
always @ (posedge clk)
  if (rst)
    begin
       clk_dv   <= 0;
       clk_en   <= 1'b0;
       clk_en_d <= 1'b0;
    end
  else
    begin
       clk_dv   <= clk_dv_inc[16:0];
       clk_en   <= clk_dv_inc[17];
       clk_en_d <= clk_en;
    end
```

clk_en gets changed every time the 18th bit of clk_dv_inc gets turned on.
This means it's changed every (2^17)+1 ticks.

1.
clk_en 1

Period of Signal:
On at 0.077333495000 s
Off at 0.077333505000 s

clk_en 2



Period of Signal:
On at 0.078644215000 s
Off at 0.078644225000 s

P = Second_on - First_On = 0.078644215000 s- 0.077333495000 s = 0.00131072 s

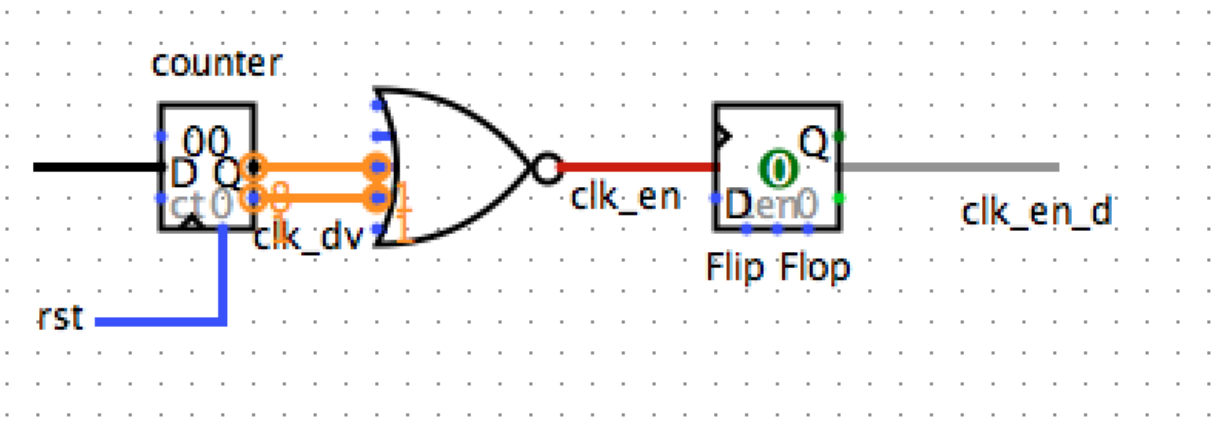
2.
Duty cycle of clk_en:

$$D = \frac{T}{P} \times 100\%$$

T = Off - On = 0.077333505000 s - 0.077333495000 s = $1 \times 10^{-8} s$
P = Second_on - First_On = 0.078644215000 s- 0.077333495000 s = 0.00131072 s
Thus, $D = \frac{T}{P} \times 100\%$
$= 1 \times 10^{-8} s$ / 0.00131072 s $\times 100\%$
$= 7.63 \times 10^{-6} \times 100\%$
$= 7.63 \times 10^{-4}$ %
The exact duty cycle is 1 / 2^17 $\times 100\%$ = 7.623$\times 10^{-4}$ %

3.
000000000000000001

4.



Debouncing

1.

For reference

```
80      // ============================================================================
81      // Instruction Stepping Control
82      // ============================================================================
83
84      always @ (posedge clk)
85        if (rst)
86          begin
87             inst_wd[7:0] <= 0;
88             step_d[2:0]  <= 0;
89          end
90        else if (clk_en)
91          begin
92             inst_wd[7:0] <= sw[7:0];
93             step_d[2:0]  <= {btnS, step_d[2:1]};
94          end
95
96      always @ (posedge clk)
97        if (rst)
98          inst_vld <= 1'b0;
99        else
100         inst_vld <= ~step_d[0] & step_d[1] & clk_en_d;
101
102     always @ (posedge clk)
103       if (rst)
104         inst_cnt <= 0;
105       else if (inst_vld)
106         inst_cnt <= inst_cnt + 1;
107
108     assign led[7:0] = inst_cnt[7:0];
```

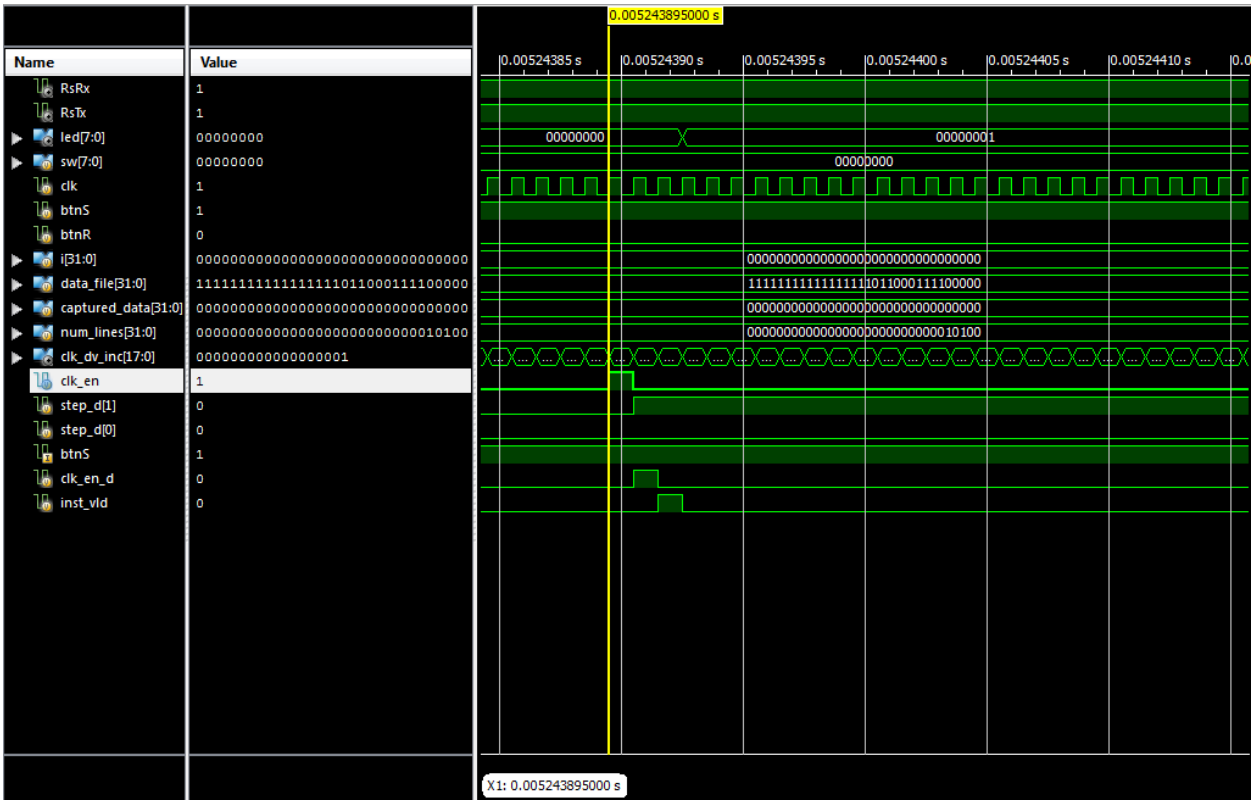
clk_en_d lags behind clk_en by one tick, as seen here:

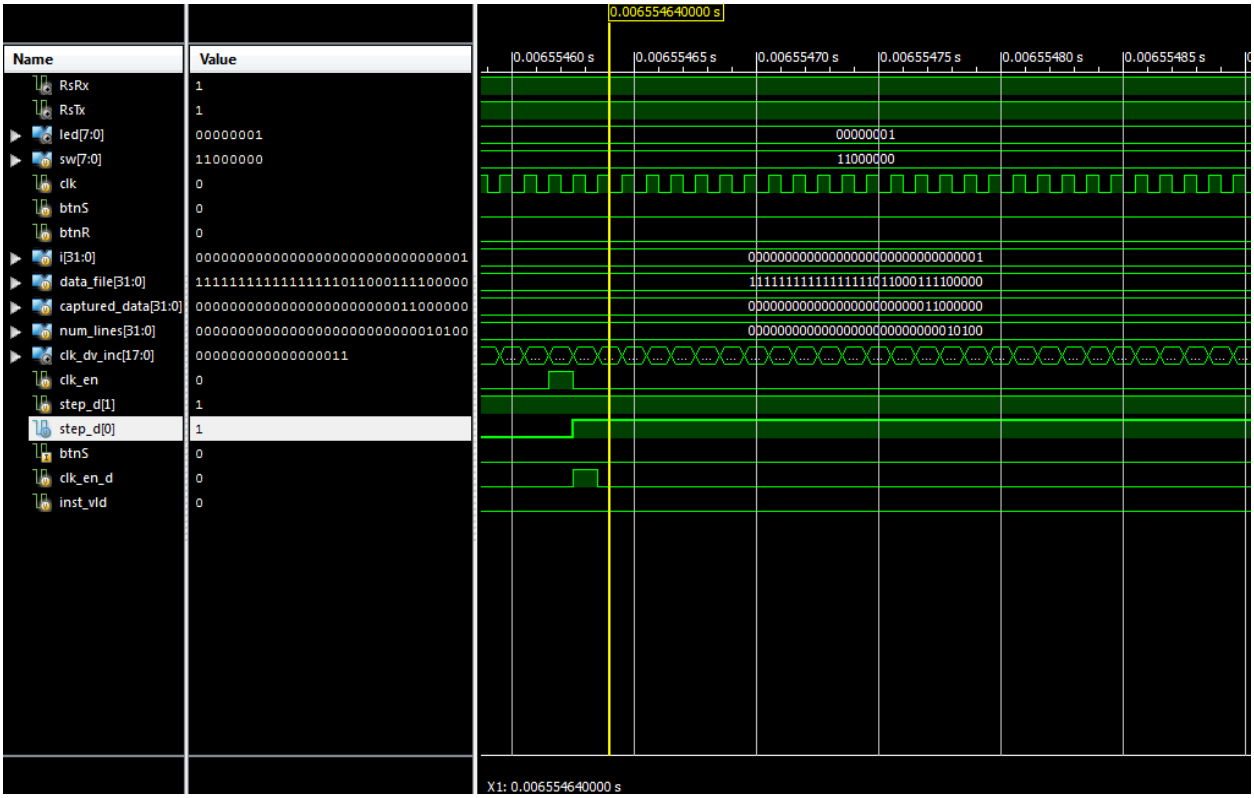
```
// ============================================================================
// 763Hz timing signal for clock enable
// ============================================================================

assign clk_dv_inc = clk_dv + 1;

always @ (posedge clk)
  if (rst)
    begin
       clk_dv   <= 0;
       clk_en   <= 1'b0;
       clk_en_d <= 1'b0;
    end
  else
    begin
       clk_dv   <= clk_dv_inc[16:0];
       clk_en   <= clk_dv_inc[17];
       clk_en_d <= clk_en;
    end
```

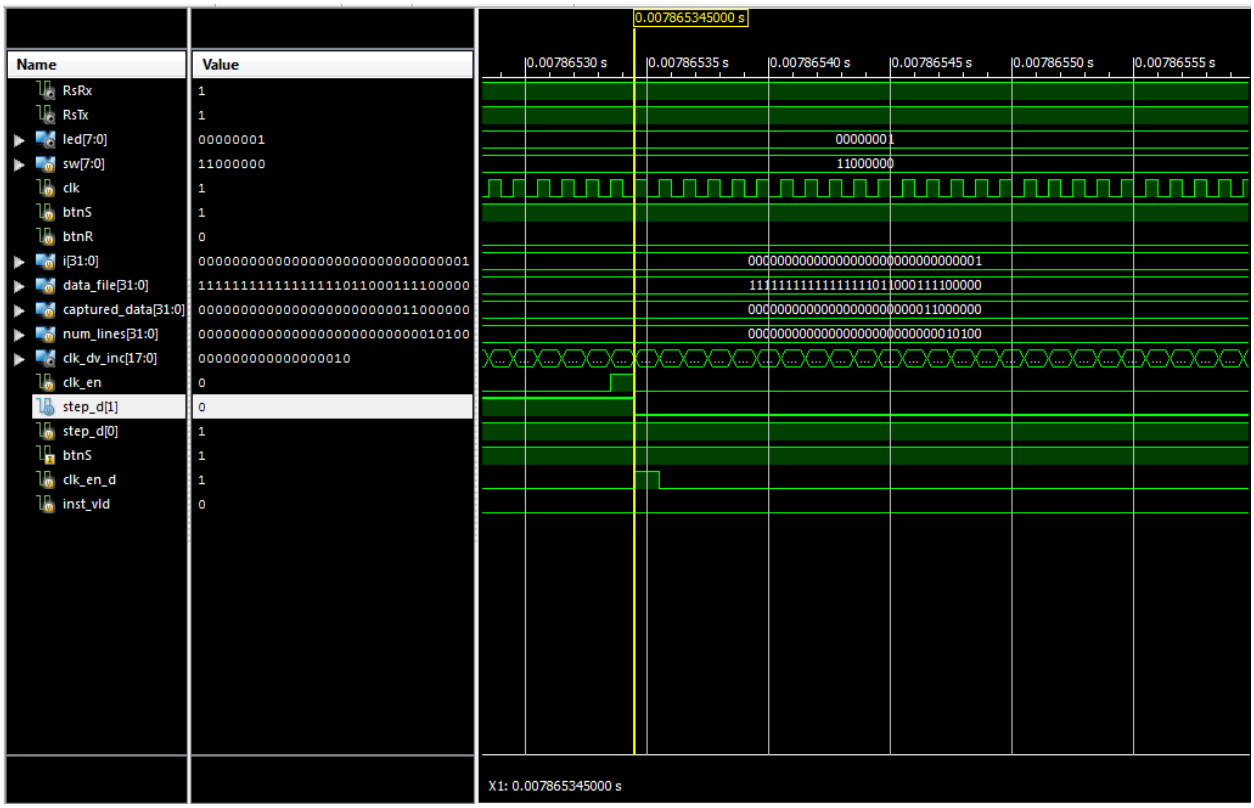
As can be seen in line 90-94, the values of step_d are updated to include the instruction btnS only when clk_en turns on. Then, we want inst_vld to be evaluated after the values of step_d are updated, but by this point, clk_en will have turned off already. Thus, we need clk_en_d, which lags behind clk_en by one tick, so that it will still be on when we evaluate inst_vld.

2. No. The duty cycle is calculated as $D = \frac{T}{P} \times 100\%$T is just the time of one tick (since clk_en is only high for one tick). If we change the code to clk_en <= clk_dv[16], this will make the clock only count to 2^16 ticks, which is half of the original 2^17. However, this halves the *period*, not the duty cycle nor signal high time.
Thus, changing it to clk_en <= clk_dv[16] will actually *double* the duty cycle instead.

3. Waveform captures that clearly show the timing relationship between clk_en, step_d[1], step_d[0], btnS, clk_en_d, and inst_vld

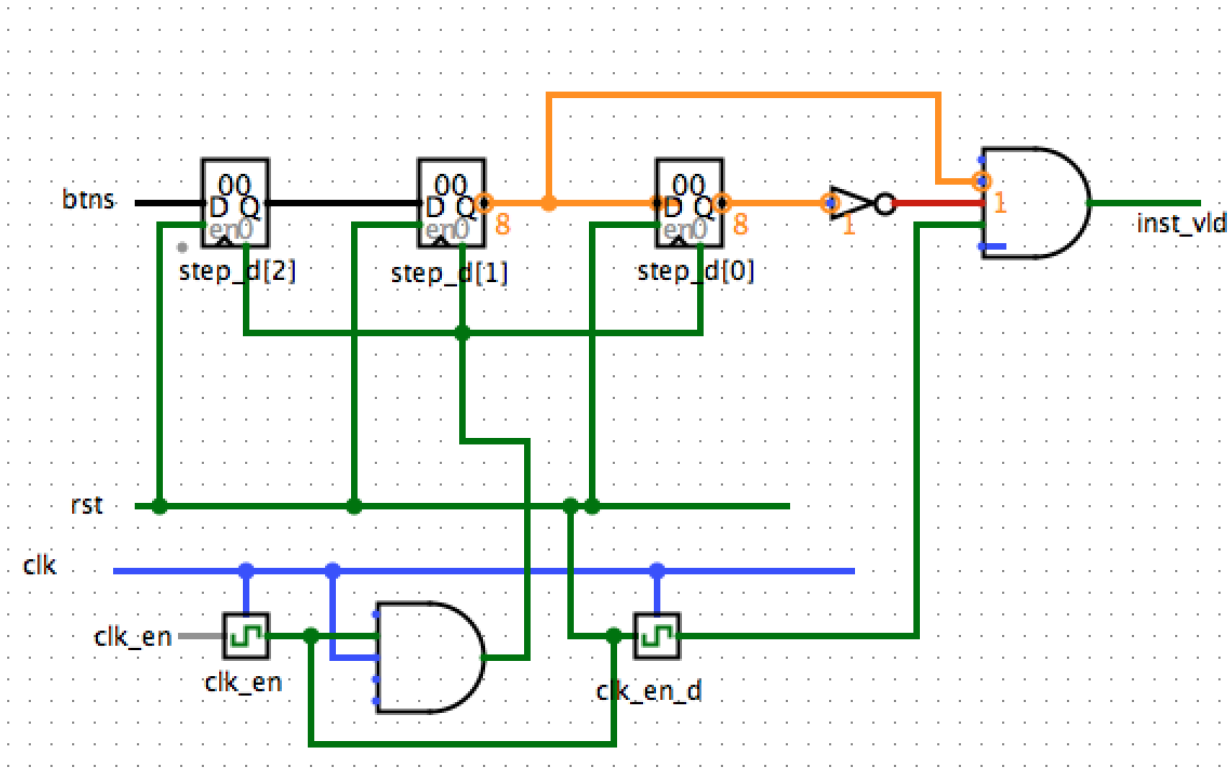

0.005243895000 s
Name
Value
RsRx
1
RsTx
1
led[7:0]
00000000
sw[7:0]
00000000
clk
1
btnS
1
btnR
0
i[31:0]
00000000000000000000000000000000
data_file[31:0]
11111111111111111011000111100000
captured_data[31:0]
00000000000000000000000000000000
num_lines[31:0]
00000000000000000000000000010100
clk_dv_inc[17:0]
00000000000000001
clk_en
1
step_d[1]
0
step_d[0]
0
btnS
1
clk_en_d
0
inst_vld
0
0.00524385 s
0.00524390 s
0.00524395 s
0.00524400 s
0.00524405 s
0.00524410 s
00000000
00000001
00000000
X1: 0.005243895000 s


0.006554640000 s
Name
Value
RsRx
1
RsTx
1
led[7:0]
00000001
sw[7:0]
11000000
clk
0
btnS
0
btnR
0
i[31:0]
00000000000000000000000000000001
data_file[31:0]
11111111111111111011000111100000
captured_data[31:0]
00000000000000000000000011000000
num_lines[31:0]
00000000000000000000000000010100
clk_dv_inc[17:0]
00000000000000011
clk_en
0
step_d[1]
1
step_d[0]
1
btnS
0
clk_en_d
0
inst_vld
0
0.00655460 s
0.00655465 s
0.00655470 s
0.00655475 s
0.00655480 s
0.00655485 s
00000001
11000000
X1: 0.006554640000 s


0.007865345000 s
Name
Value
RsRx
1
RsTx
1
led[7:0]
00000001
sw[7:0]
11000000
clk
1
btnS
1
btnR
0
i[31:0]
00000000000000000000000000000001
data_file[31:0]
11111111111111111011000111100000
captured_data[31:0]
00000000000000000000000011000000
num_lines[31:0]
00000000000000000000000000010100
clk_dv_inc[17:0]
000000000000000010
clk_en
0
step_d[1]
0
step_d[0]
1
btnS
1
clk_en_d
1
inst_vld
0
0.00786530 s
0.00786535 s
0.00786540 s
0.00786545 s
0.00786550 s
0.00786555 s
X1: 0.007865345000 s


0.009176000500 s
Name
Value
RsRx
1
RsTx
1
led[7:0]
00000001
sw[7:0]
11000000
clk
0
btnS
1
btnR
0
i[31:0]
00000000000000000000000000000001
data_file[31:0]
11111111111111111011000111100000
captured_data[31:0]
00000000000000000000000011000000
num_lines[31:0]
00000000000000000000000000010100
clk_dv_inc[17:0]
011111111111111011
clk_en
0
step_d[1]
0
step_d[0]
1
btnS
1
clk_en_d
0
inst_vld
0
0.00917595 s
0.00917600 s
0.00917605 s
0.00917610 s
0.00917615 s
0.00917620 s
00000010
X1: 0.009176000500 s

4.


btns
00
D Q
en
step_d[2]
00
D Q
en
8
step_d[1]
00
D Q
en
8
step_d[0]
1
1
inst_vld
rst
clk
clk_en
clk_en
clk_en_d

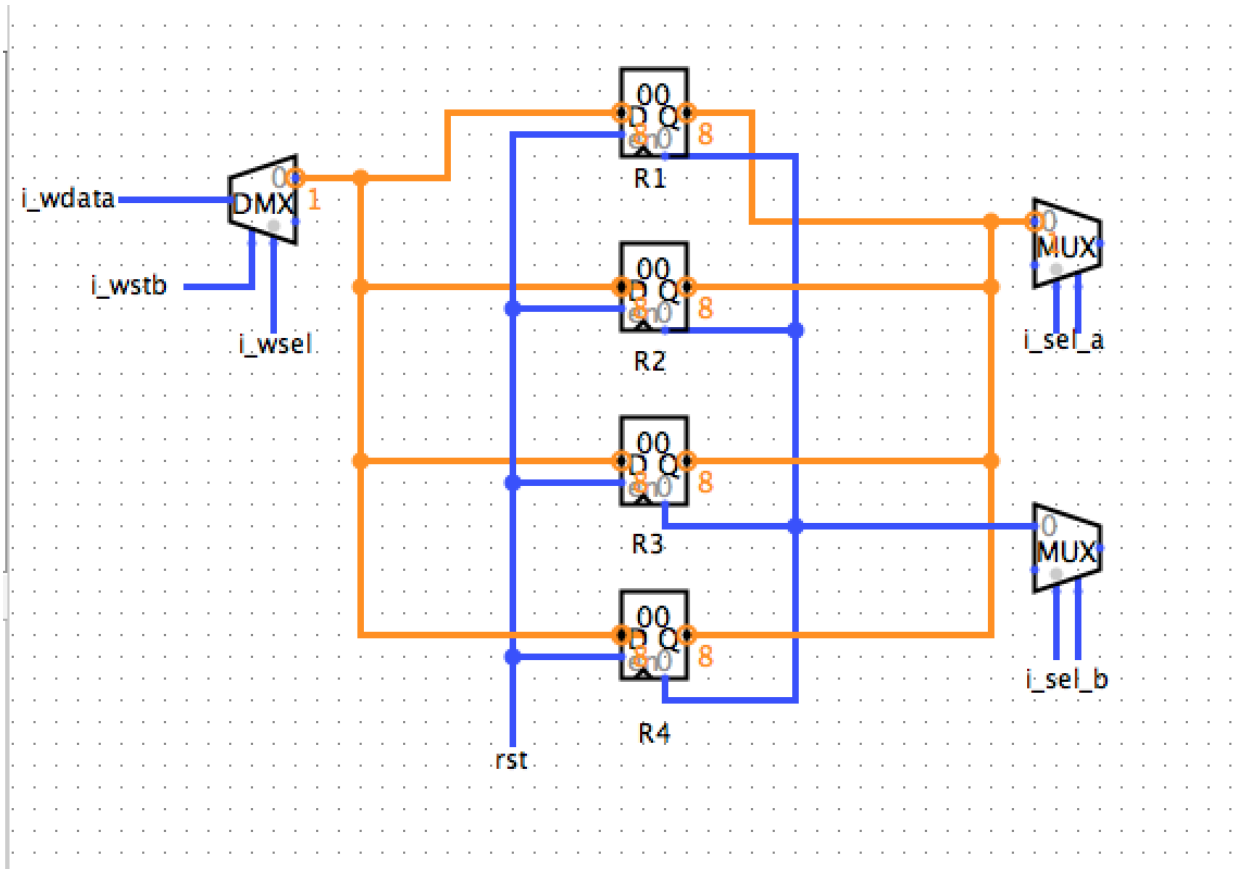
Register File

```
 8  `include "seq_definitions.v"
 9
10    output [alu_width-1:0]    o_data_a;
11    output [alu_width-1:0]    o_data_b;
12
13    input [seq_rn_width-1:0] i_sel_a;
14    input [seq_rn_width-1:0] i_sel_b;
15
16    input                     i_wstb;
17    input [alu_width-1:0]     i_wdata;
18    input [seq_rn_width-1:0] i_wsel;
19
20    input                     clk;
21    input                     rst;
22
23    reg [alu_width-1:0]       rf [0:seq_num_regs-1];
24    integer                   i;
25
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;
34
35    assign o_data_a = rf[i_sel_a];
36    assign o_data_b = rf[i_sel_b];
37
38  endmodule // seq_rf
```

1. Register is written a non-zero value: Line 33
This is part of **sequential** logic, since the register is only written when i_wstb is true. Thus,on every iteration, this will depend on previously run code.

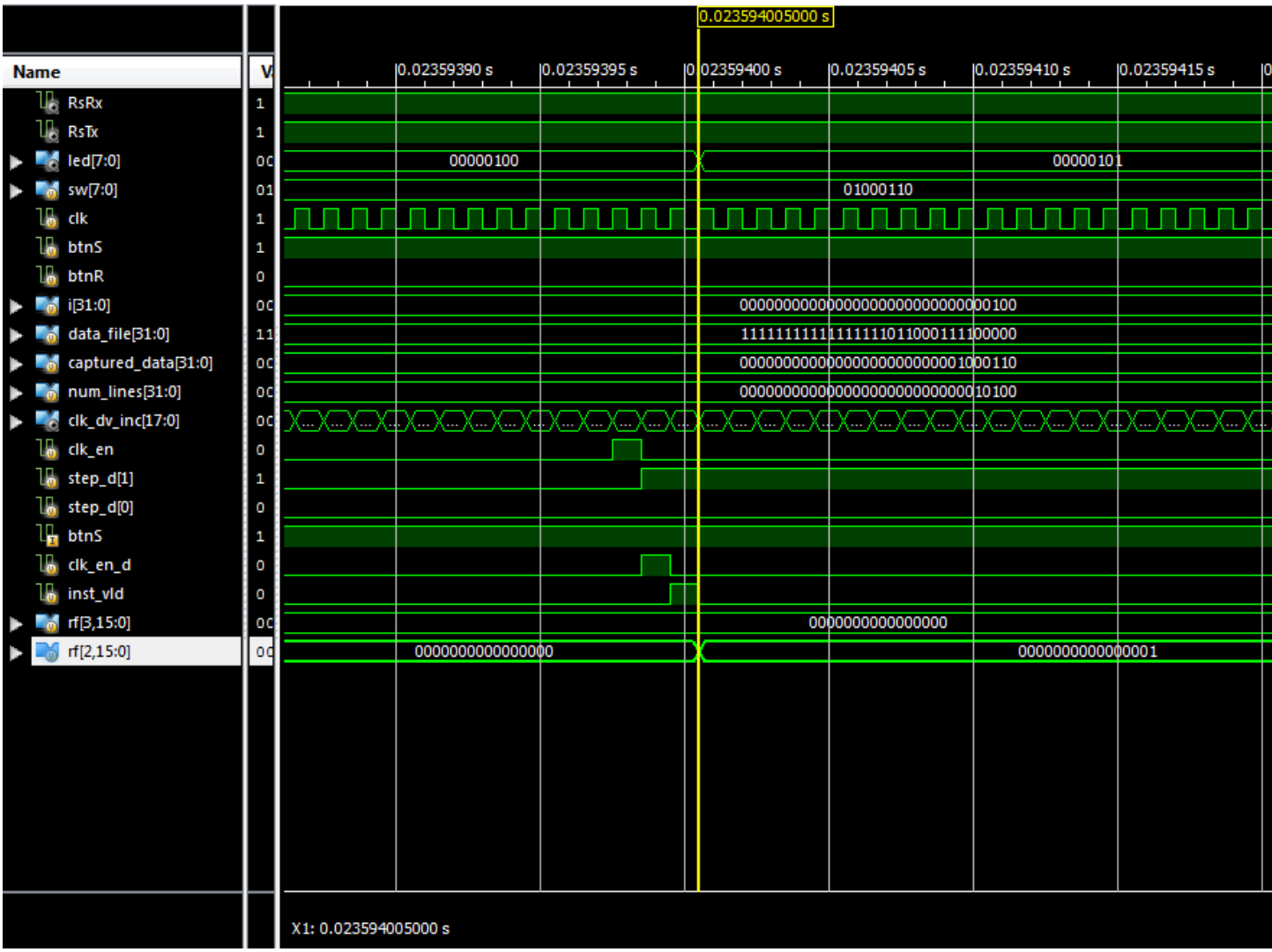
2. Register values are read out from the register on line 35.
i_sel_a is set by input, so this does not depend on what other code is run. Thus, this is **combinatorial** logic.

3.



4. Waveform of the first time register 3 is written with a non-zero value

Note: We renamed our registers from 3, 2, 1 to 2, 1, 0. Thus “register 3” is actually “register 2” for us. So please refer to rf[2, 15:0] below.



Workshop 2

Nicer UART Output

```
reg [7:0] rxData;
reg [63:0] tempRxData = 0;
event     evBit;
event     evByte;
event     evTxBit;
event     evTxByte;
reg       TX;
```

```
initial
  begin
    TX = 1'b1;
  end

  always @ (negedge RX)
  begin
    rxData[7:0] = 8'h0;

    #(0.5*bittime);
    repeat (8)
    begin
      #bittime ->evBit;
      //rxData[7:0] = {rxData[6:0],RX};
      rxData[7:0] = {RX,rxData[7:1]};
    end
    ->evByte;

    tempRxData = tempRxData<<8;
    tempRxData[7:0] = rxData;

    if(rxData == "\r")
    begin
      $display ("Output is: %s", tempRxData);
      tempRxData = 0;
  end
end
```

An Easier Way to Load Sequencer Program

Existing sequencer testbench

```
tskRunPUSH(0,4);
tskRunPUSH(0,0);
tskRunPUSH(1,3);
tskRunMULT(0,1,2);
tskRunADD(2,0,3);
tskRunSEND(0);
tskRunSEND(1);
tskRunSEND(2);
tskRunSEND(3);
```

Change the static set of instructions

```
data_file = $fopen("seq.code", "r");
if (data_file == 0)
begin
 $display("data_file handle was NULL");
 $finish;
end


$fscanf(data_file, "%b\n", captured_data);
num_lines = captured_data;

if (captured_data > 1023)
begin
  num_lines = 1023;
end

for( i = 0; i<num_lines; i=i+1)
begin
    $fscanf(data_file, "%b\n", captured_data);
    tskRunInst(captured_data);
end
```

Fibonacci Numbers

First 10 numbers of the Fibonacci series:
(fibonacci.code)
10100
00000000
11000000
00010001
11010000
01000110
11100000
01011000
11000000
01001001
11010000
01000110
11100000
01011000
11000000
01001001
11010000
01000110
11100000
01011000
11000000