Lab 1

CS M152A

Aman Agarwal & Lowell Bander

April 23, 2015

Contents

1 FPGA Design Workflow

1.1 Design

The design stage of the FPGA development workflow is done outside of the the Xilinx ISE and its simulation tools, for it solely consists of the drawing of logical schematics so that the designer may understand the higher level functioning of the module to be constructed.

1.2 Implementation

The hierarchical file structure is determined based on which modules depend on others. At the very top-level we have a test bench file, `tb.v` which is directly above a unit under test file, `nexys3.v`. Below that we have a sequencer file and a uart file, `seq.v` and `uart_top.v`. The sequencer is composed of the register file, `seq_rf.v` and the ALU unit which has add and multiply modules. The Uart module represents the interface to the computer where we see the output. The Uart module is only needed for testing so it is not a part of the sequencer module.

1.3 Simulation

The simulation was done by loading the simulation files, choosing which signals to view/render, running the simulation for a short amount of time, and then examining the output to look for errors. Our simulation files were the test bench file and the model uart file, `tb.v` and `model_uart.v`. The signals we chose to view were: `uut_`, `inst_wd`, and `inst_vld`. After running the simulation, we looked at the `UART0` and noticed that it was exactly what we expected.

1.4 Logic Synthesis

To perform synthesis, we must import the synthesis file and then run synthesis using XST. The synthesis file contains the pin assignments and timing requirements. Our synthesis file is `nexys3.ucf`.

1.5 Technology Mapping and Cell Placement

Due to the lack of complexity in this project, we do not have to perform these steps.

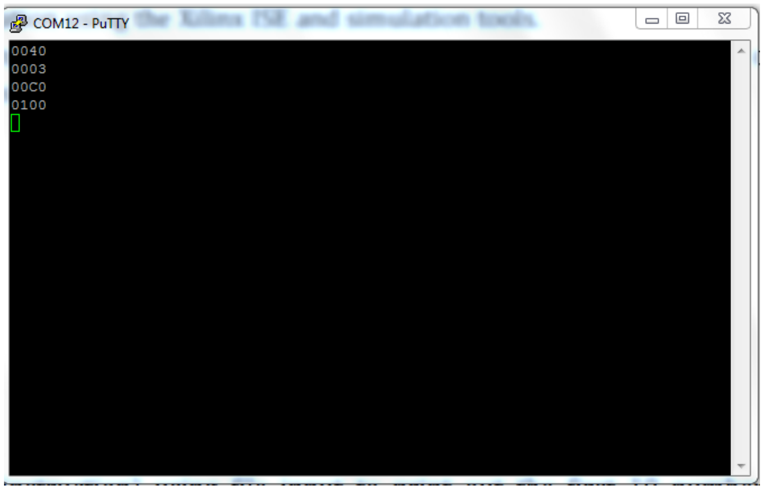
1.6 Route and Bitstream Generation

These two steps are performed by the "Implement Design" part of ISE. This process outputs a bitstream file, `nexys3.bit`. We must then use the bitstream file and program the device before we can run programs on it.

2 Example Program

Sequencer Instruction	Binary
`PUSH R0 0x4`	0000 0010
`PUSH R0 0x0`	0000 0000
`PUSH R1 0x3`	0001 0011
`MULT R0 R1 R2`	1000 0110
`ADD R2 R0 R3`	0110 0011
`PUSH R0 0x4`	0000 0100
`SEND R0`	1100 xxxx
`SEND R1`	1101 xxxx
`SEND R2`	1110 xxxx
`SEND R3`	1111 xxxx

Table 1: Sequencer instructions translated to binary. The four least significant bits are "don't care" as their value is not used for the `SEND` instruction.



```
0040
0003
00C0
0100
```

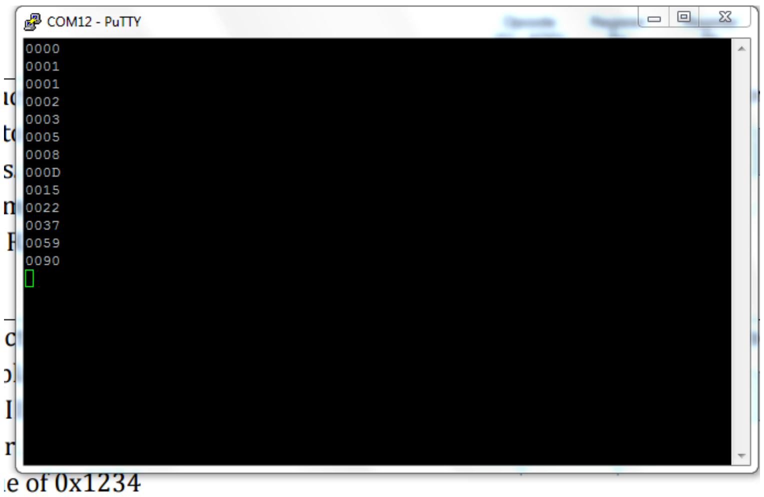
Figure 1: UART console output resulting from the executions in Table 1.

3 Fibonacci

For this section, we generated the first 10 Fibonacci numbers by writing the necessary sequencer instructions, translating them into machine instructions, and then loading them into the simulator using the `$fopen` and `$fscanf`.

Sequencer Instruction	Binary
`PUSH R0 0x0`	`0000 0000`
`PUSH R1 0x1`	`0001 0001`
`SEND R0`	`1100 xxxx`
`SEND R1`	`1101 xxxx`
`ADD R0 R1 R2`	`0100 0110`
`SEND R2`	`1110 xxxx`
`ADD R1 R2 R0`	`0101 1000`
`SEND R0`	`1100 xxxx`
`ADD R2 R0 R1`	`1010 0001`
`SEND R1`	`1101 xxxx`
`ADD R0 R1 R2`	`0100 0110`
`SEND R2`	`1110 xxxx`
`ADD R1 R2 R0`	`0101 1000`
`SEND R0`	`1100 xxxx`
`ADD R2 R0 R1`	`0110 0001`
`SEND R1`	`1101 xxxx`
`ADD R0 R1 R2`	`0100 0110`
`SEND R2`	`1110 xxxx`
`ADD R1 R2 R0`	`0101 1000`
`SEND R0`	`1100 xxxx`

Table 2: The sequencer instructions, and their corresponding machine translations, necessary to generate the first 10 Fibonacci numbers. When loaded into simulation, the "don't care" values were substituted for `0`, but it is important to note that we could have just as easily used `1`.



```
0000
0001
0001
0002
0003
0005
0008
000D
0015
0022
0037
0059
0090
```

Figure 2: UART console output resulting from the executions in Table 2.

4 Exercise 1

4.1 Clock Enable

1. The signal `clk_en` is a clock enable signal whose assertion is a prerequisite for the modification of the `inst_wd` and `step_d` signals, meaning that new instructions will only be executed by the sequencer module of this clock enable is asserted.

2. .

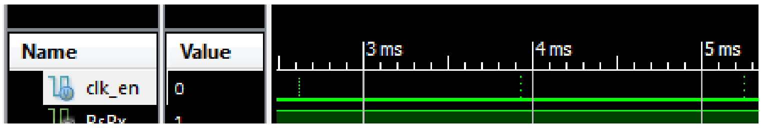


Figure 3: The period T can be calculated by taking half of the difference between the end of the second period and the beginning of the first: $T = 5.243895 - 2.622455 = 1.31072$ ms.

3. The value of the 16-bit `clk_dv` signal is 0 when `clk_en` is 1.

4. .

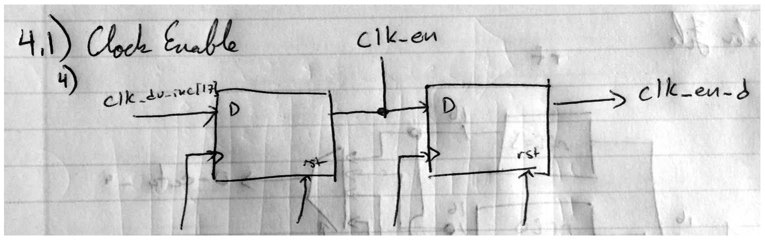


Figure 4: A simple schematic illustrating the logical relationship amongst clk_dv, clk_en, and clk_en_d signals.

4.2 Instruction Valid

1. The `inst_vld` signal is set if `btnS`, the middle button that executes an instruction, was pressed during the last clock cycle, but not the clock cycle before that.

2. The first simulation time interval during which the expression inst_vld = step_d[0] & step_d[1] & clk_en_d evaluates to 1 occurs at 5.243905 ms.

3. We use `clk_en_d` instead of `clk_en` so that instruction execution occurs a clock cycle later than it would with `clk_en`.

4. .

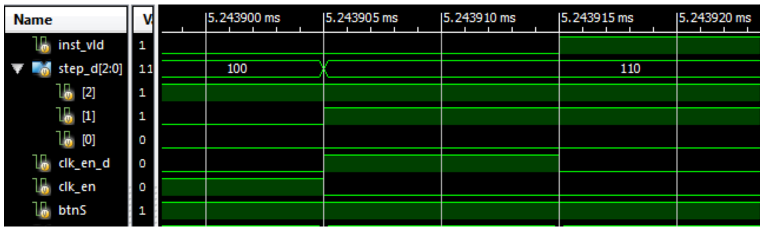


Figure 5: A waveform capture that clearly shows the timing relationship between `clk_en`, `step_d[1]`, `step_d[0]`, `btnS`, `clk_en_d`, and `inst_vld`.

5. .

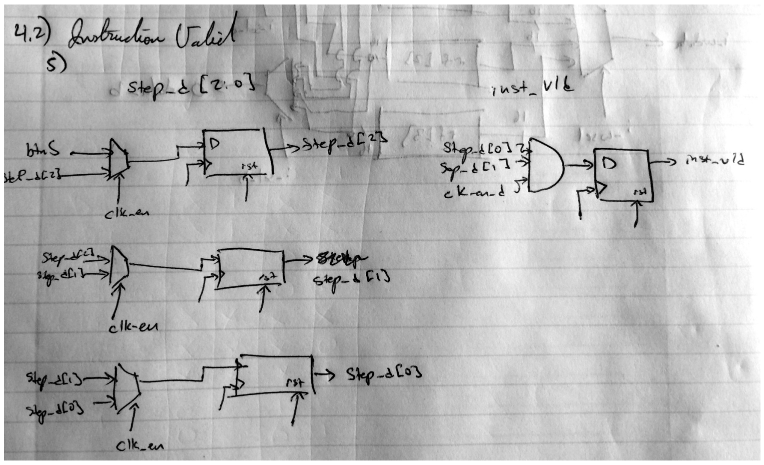


Figure 6: A simple schematic illustrating the logical relationship amongst clk_en, step_d[1], step_d[0], btnS, clk_en_d, and inst_vld.

4.3 Register File

1. The register file, takes as inputs, 2 read signals and 1 write signal, along with a write address and write data. The write signal tells us whether or not we will write to the address specified by `i_wsel` with the data in `i_wdata`. The two read signals specify where we are reading from.

2. The register is written to in sequential code. We know this because it is located in an `always` block.

3. The read code is combinatorial logic which we know because it is *not* located in a `always` block. This would be implemented using a multiplexor

that uses `i_sel_X`, a (in our case) 2-bit value, to determine which register we are reading out of.

4. .

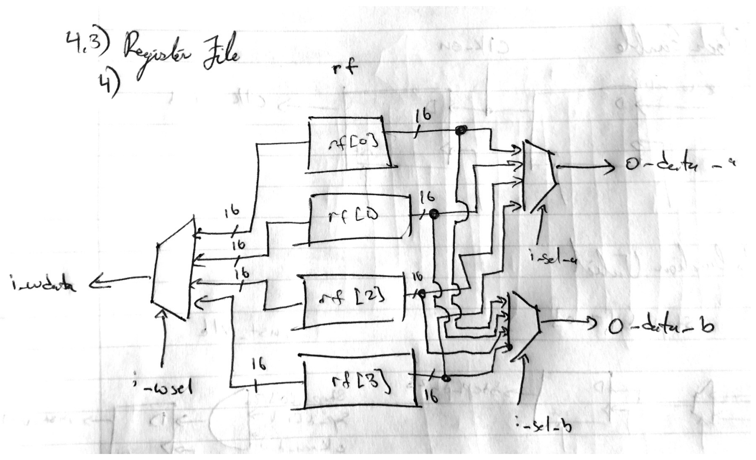
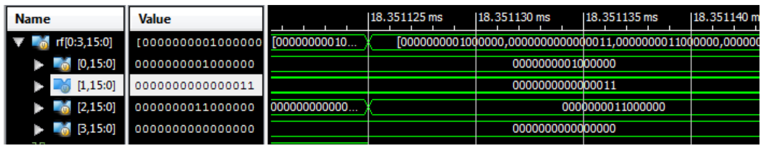


Figure 7: A circuit diagram of the register file block.

5. .



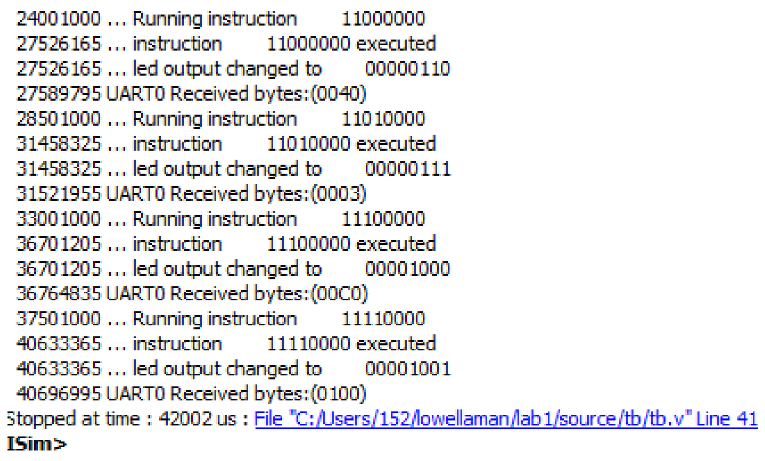
```
Name            Value
rf[0:3,15:0]    [0000000001000000
  [0,15:0]      0000000001000000
  [1,15:0]      0000000000000011
  [2,15:0]      0000000011000000
  [3,15:0]      0000000000000000
```

```
|18.351125 ms  |18.351130 ms  |18.351135 ms  |18.351140 m
[0000000010...  [0000000001000000,0000000000000011,0000000011000000,000000
                0000000001000000
                0000000000000011
000000000000...  0000000011000000
                0000000000000000
```

Figure 8: A waveform that shows the first time register 3 is written with a non-zero value.

5 Exercise 2

5.1 Nicer UART Output



```
24001000 ... Running instruction       11000000
27526165 ... instruction       11000000 executed
27526165 ... led output changed to       00000110
27589795 UART0 Received bytes:(0040)
28501000 ... Running instruction       11010000
31458325 ... instruction       11010000 executed
31458325 ... led output changed to       00000111
31521955 UART0 Received bytes:(0003)
33001000 ... Running instruction       11100000
36701205 ... instruction       11100000 executed
36701205 ... led output changed to       00001000
36764835 UART0 Received bytes:(00C0)
37501000 ... Running instruction       11110000
40633365 ... instruction       11110000 executed
40633365 ... led output changed to       00001001
40696995 UART0 Received bytes:(0100)
Stopped at time : 42002 us : File "C:/Users/152/lowellaman/lab1/source/tb/tb.v" Line 41
ISim>
```

Figure 9: Output of the modified testbench, where per-byte output is suppressed.

To accomplish this, we replaced the single `$display` line in the original file with the following lines of code, which buffer the output of the program before flushing them all to the display.

```
buff[47:0] <= {buff[39:0], rxData[7:0]};
buff_cnt <= buff_cnt + 1;

if (buff_cnt[2] & buff_cnt[0])
begin
    buff_cnt <= 0;
end

if (buff_cnt[2] & buff_cnt[0])
begin
    $display ("%d %s Received bytes:(%s%s%s%s)", $stime, name,
            buff[39:32], buff[31:24], buff[23:16], buff[15:8]);
end
```

5.2 An Easier Way to Load Sequencer Program

1. Line 56 instantiates a nexys3 module named uut_. This is where tb.v sends instructions to the UUT. The instructions are stored in the 8 bit

register sw.

```
nexys3 uut_ (/*AUTOINST*/
             // Outputs
             .RsTx                    (RsTx),
             .led                     (led[7:0]),
             // Inputs
             .RsRx                    (RsRx),
             .sw                      (sw[7:0]),
             .btnS                    (btnS),
             .btnR                    (btnR),
             .clk                     (clk));
```

2. The user tasks called are: tskRunPUSH, tskRunSEND, tskRunADD, tskRunMULT which each call the user task tskRunInst.

To accomplish loading instructions from a file, we used the following code.

```
$readmemb("seq.code", inst_mem);

length = inst_mem[0];

$display("File has %0d lines", length);

for(index = 1; index < length + 1; index = index + 1) begin
     tskRunInst(inst_mem[index]);
end
```

5.3 Fibonacci Numbers

The code was fed in through a file. The instructions are the same as those listed under the Fibonacci section.