Introduction to Digital Logic Design Lab
EECS 31L

Lab #3 : Single Cycle Processor

Noah Mathew
Student ID #59622566

Date 2/20/2024

1 Objective

In this lab, we were to construct some of the modules of a RISC-V Single Cycle Processor. A Single Cycle Processor is responsible for the following: Supplying the instruction address to the instruction memory, then using register operands used by an instruction specified by fields of the particular instruction, which when the register operands have been fetched can then be operated on to compute: memory address, arithmetic result, or equality check. Within the diagram, we have gotten to explore and design components like the ALU. Within this Single Cycle Processor, if the operation is a load or store, the ALU result is used as an address to either store value from the registers or load a value from memory into the registers. Then, the result from either the ALU or memory is written back into the register file.

Some of the inputs include RegWrite, ALUSrc, ALUCC, MemRead, and MemWrite which are building blocks to developing the Single Cycle Processor. Some of these designs are to be implemented in this lab. See the diagram below:

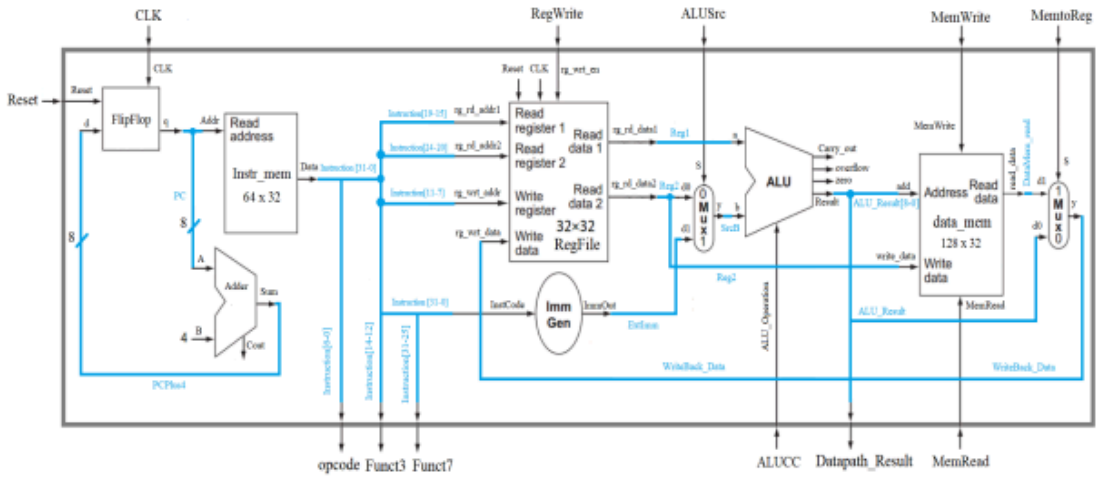


Figure 1: The building block of a single cycle processor.

2 Procedure

In this lab we are to develop the following components: the Flip Flop, the Instruction Memory, and the Register File.

The Flip Flop is where the function is to store a signal. It contains an input value d, a clock signal, a reset signal to determine when the output should be initialized, and an output value q that stores the value of the input d. Here is the design implementation in Verilog:

```
14      // Dependencies:
15      //
16      // Revision:
17      // Revision 0.01 - File Created
18      // Additional Comments:
19      //
20      //////////////////////////////////////////////////////////////////////////////////
21      `timescale 1ns / 1ps
22
23      module FlipFlop(clk, reset, d, q);
24      input [7:0] d;
25      input clk;
26      input reset;
27      output reg [7:0] q;
28
29      always @(posedge clk)
30          begin
31          if(reset == 1'b1)
32                  q <= 8'h00;
33          else
34                  q <= d;
35          end
36
37      endmodule
38
```

The Instruction Memory is implemented where the address line represents the input and there is an output signal that represents the instruction that is read from the memory. Here is the design implementation in Verilog:

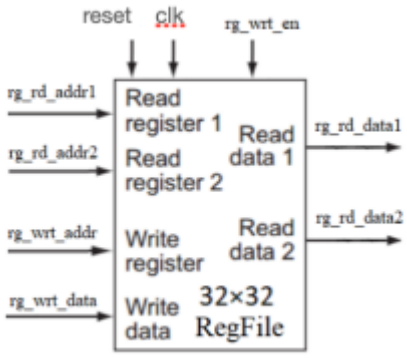
```
`timescale 1ns / 1ps

module InstMem(
    input [7:0] addr,
    output wire [31:0] instruction
    );


reg [31:0] memory [63:0];
```

```
30  initial begin
31  memory[0] = 32'h00007033; // and r0, r0, r0 32'h00000000
32  memory[1] = 32'h00100093; // addi r1, r0, 1 32'h00000001
33  memory[2] = 32'h00200113; // addi r2, r0, 2 32'h00000002
34  memory[3] = 32'h00308193; // addi r3, r1, 3 32'h00000004
35  memory[4] = 32'h00408213; // addi r4, r1, 4 32'h00000005
36  memory[5] = 32'h00510293; // addi r5, r2, 5 32'h00000007
37  memory[6] = 32'h00610313; // addi r6, r2, 6 32'h00000008
38  memory[7] = 32'h00718393; // addi r7, r3, 7 32'h0000000B
39  memory[8] = 32'h00208433; // add r8, r1, r2 32'h00000003
40  memory[9] = 32'h404404b3; // sub r9, r8, r4 32'hfffffffe
41  memory[10] = 32'h00317533; // and r10, r2, r3 32'h00000000
42  memory[11] = 32'h0041e5b3; // or r11, r3, r4 32'h00000005
43  memory[12] = 32'h0041a633; // if r3 is less than r4 then r12 = 1 32'h00000001
44  memory[13] = 32'h007346b3; // nor r13, r6, r7 32'hfffffff4
45  memory[14] = 32'h4d34f713; // andi r14, r9, "4D3" 32'h000004D2
46  memory[15] = 32'h8d35e793; // ori r15, r11, "8d3" 32'hfffff8d7
47  memory[16] = 32'h4d26a813; // if r13 is less than 32'h000004D2 then r16 = 1 32'h00000000
48  memory[17] = 32'h4d244893; // nori r17, r8, "4D2" 32'hfffffb2C
49
50  end
51
52  assign instruction = memory[addr[7:2]];
53  endmodule
```

The Register File is where there is a set of registers that can be read and written when given a register number that needs to be accessed. This register file has two read ports and one write port. Additionally, there is a reset and clock signal that operates asynchronous reset. Here is the register file diagram:



One of the inputs represents the register number, one represents the data to write, and a clock controls the writing to the register. Here is the design implementation in Verilog:

```
21      `timescale 1ns / 1ps
22
23      module RegFile(
24      clk, reset, rg_wrt_en,
25      rg_wrt_addr,
26      rg_rd_addr1,
27      rg_rd_addr2,
28      rg_wrt_data,
29      rg_rd_data1,
30      rg_rd_data2
31      );
32
33      input clk, reset, rg_wrt_en;
34      input [4:0] rg_wrt_addr;
35      input [4:0] rg_rd_addr1;
36      input [4:0] rg_rd_addr2;
37      input [31:0] rg_wrt_data;
38
39      output reg [31:0] rg_rd_data1;
40      output reg [31:0] rg_rd_data2;
41
42      reg [31:0] registers [31:0]; //represents our 32 elements (i.e. 32 registers), of our 32-bit registers
43      integer i;
```

```
44
45
46  always @(posedge clk or posedge reset)
47  begin
48   if(reset)
49   begin
50      for(i = 0; i < 32; i = i + 1)
51      begin
52       registers[i] <= 32'b00000000; //reset them all to 0
53      end
54   end
55   else if(rg_wrt_en)
56   begin
57   registers[rg_wrt_addr] <= rg_wrt_data; //we will write the data from input line rg_wrt_data to the register number rg_wrt_addr
58   end
59  end
60
61
62
63
64
65  endmodule
```

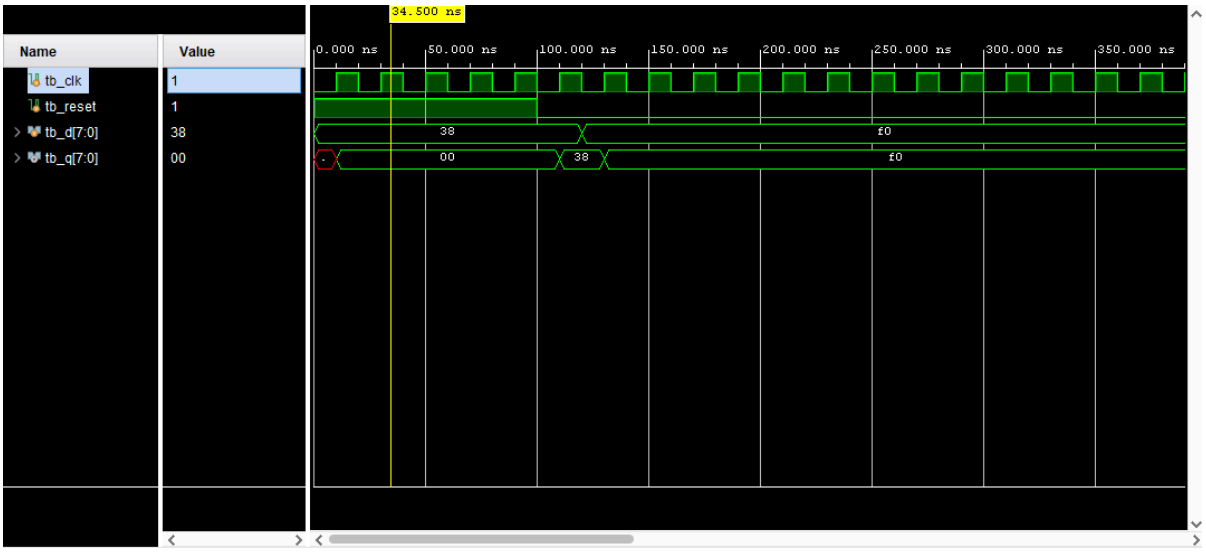
3 Simulation Results

Here talk about the expected result. Explain your test cases and how you design them to cover all combinations of inputs. Put the screenshot of your simulation here. You may put one image that contains all test cases or several images (if they are not clear in one image). Also, define the signals in the screenshots and explain how and when changes in the inputs make changes in the output. If there is something different from what you expect, explain why.

Flip Flop:

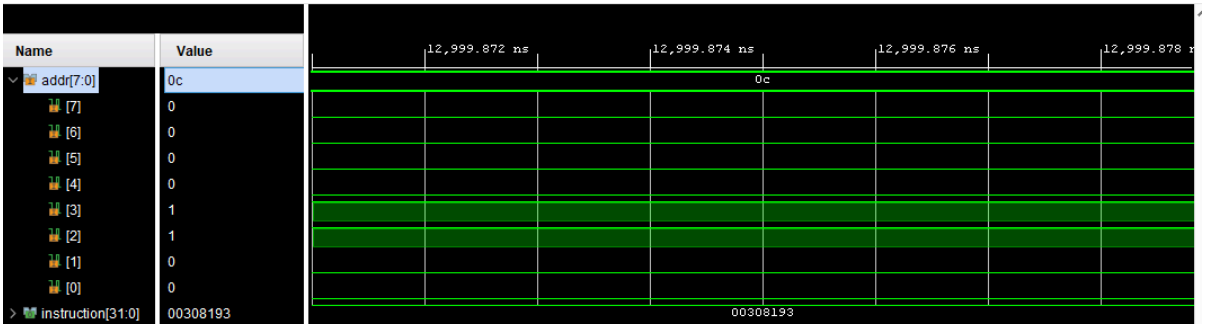
In the Flip Flop design, I defined the clk, and synchronous reset, an 8-bit input d, and output

q (8-bit). As you can see in the waveform below, the clock input changes from the rising edge, and then q stores the d input value and repeats until the next rising edge of the clock. In this module, I defined my testbench with the following: a clock signal with a clock period of 20ns, I reset the signal for 100ns, and the output q will be 0 regardless of the input of d, then after 100ns the reset will be set to 0 (i.e. inactive), and the output should reflect the following seen in the waveform.



Instruction Memory:

In the instruction memory, we have the 8-bit addr, along with the output 32-bit instruction which handles storing the instructions. As seen in the design implementation (i.e. the code segment in the procedure), given the instruction codes of memory, we can implement and store those in the instruction memory. In the waveform below, I tested my module using the Force Constant. If I have given the value to the addr signal a value of 3 the instruction output returns a hexadecimal value of 00308193 (32-bit) which returns as expected as seen in the waveform below:



Register File:

The register file is defined with the inputs of clk, reset, rg_wrt_en (enable signal), rg_rd addr1 (read input), rg_rd_addr2, rg_wrt_addr (write input register), and rg_wrt_data (write input data). The outputs are defined as rg_rd_data1 and rg_rd_data2, both are 32-bits. Unlike the Flip Flop, the Register File has an asynchronous reset, so the Register file will reset whenever there is a reset signal is on. In the waveform below, Force Constant and Force Clock were used. For the Force Constant, the reset signal is on (i.e. set to 1) for 100ns, the enable signal (i.e. rg_wrt_en) is on (set to one) to write into a register, the rg_wrt_addr represents what register number we choose to write into in this case 10010 (i.e. 18 in decimal), and setting rg_wrt_data to 45 – I chose this to model the expected output in the waveform seen in the Lab instructions. For the Force Clock, the clock was set with a leading edge of 1, a trailing edge of 0, a starting offset of 0ns, a cancel after time offset of 1000ns, and a period of 100ns. The period of 100ns and a duration of 1000ns gives us 10 clock cycles each with a clock period of 100ns. As seen in the waveform below, the simulation results are as expected: