CALIFORNIA STATE UNIVERSITY LONG BEACH

CECS 341 - Computer Architecture and Organization

Lab Assignment 2

Design Lower Level Modules

Professor: *Maryam S. Hosseini*
Team members:
Nikki Valerie Benitez
Fiona Le
Thanh Nguyen
Marie Payad
Due date: *10/30/2020*

I. *Flip Flop:*

1. Introduction:

We installed D-FlipFlop, which stores the value of its data input signal in the internal memory. It has three inputs:

- The data value (called d)
- A clock signal (called clk) - this signal indicates when the Flip flop should read the value on d and store it.
- A reset signal (called reset) - reset signal indicates when the output should set back to the initialized value (usually 0).

This module produces one output called q - the value of the internal state.

2. Procedure:

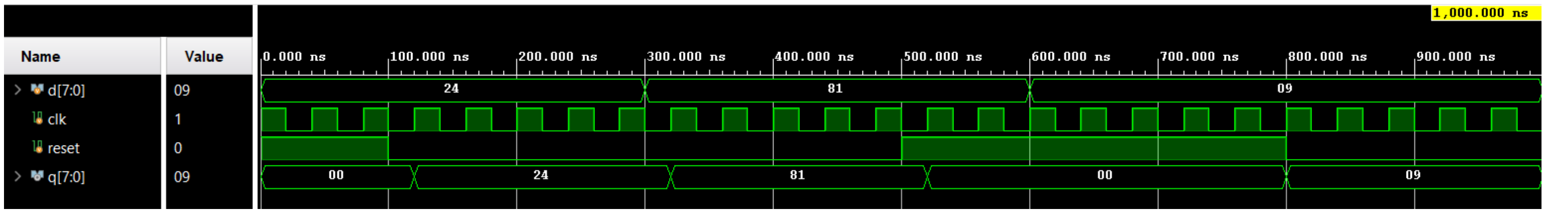
- ➢ We used @(posedge clk) to check every time that the clock input changes from deasserted to asserted (rising edge of the clock).
- ➢ Then, if reset is still on, q will be equal to 0.
- ➢ Otherwise, q will get the value of d.

3. Result:

- Testbench:

```
`timescale 1ns / 1ps

module tb_FF();
    // Inputs
    reg [7:0] d;
    reg clk;
    reg reset;

    // Output
    wire [7:0] q;

    // Instantiate the module want to test
    FlipFlop dut(clk, reset, d, q);

    // Generate Clock
    initial
        begin
        clk = 1;
        forever #20 clk = ~clk;
        end

    // Testcase
    initial
        begin
        // Case 1 when reset is set
        reset = 1;
        d = $random;
        #100;

        // Case 2 when reset is not set and d is not change
        reset = 0;
        #200;

        // Case 3 when reset is not set and d is change
        d = $random;
        #200;

        // Case 4 when reset is set again and d is not change
        reset = 1;
        #100;

        // Case 5 when reset is set and d is change
        d = $random;
        #200;

        // Case 6 when reset is not set again
        reset = 0;
        end
endmodule
```

- Waveform:



4. Appendix:

```
`timescale 1ns / 1ps

module FlipFlop(
    clk, reset, // 1 bit each
    d, // 8 bit
    q // 8 bit
    );

    // Define the input and output signals
    input clk;
    input reset;
    input [7:0] d;
    output reg[7:0] q;

    // Define the D Flip Flop modules' behaviour
    always @(posedge clk)
        begin
        if (reset == 1'b1)
            q <= 1'b0;
        else
            q <= d;
        end
endmodule
```

II. Instruction Memory:

1. Introduction:

The instruction memory has internal storage to store the instructions. This module has one input - the address line (called Addr, has 8 bits), and produces one output which is the instruction we read from the memory (called Data, has 32 bits).

2. Procedure:

- ➢ We defined a 64 x 32 2D array called ***memory*** to store 64 instructions each with 4 bytes (32 bits).
- ➢ Then stored some instructions (which are 32 bits data) in the internal storage.
- ➢ The new instruction was given the value of the memory at addr[7:2]. Because, each line of the instruction memory includes 8 bits and each instruction is 32 bits, we use bits [7:2] of the address line to point to a new instruction.

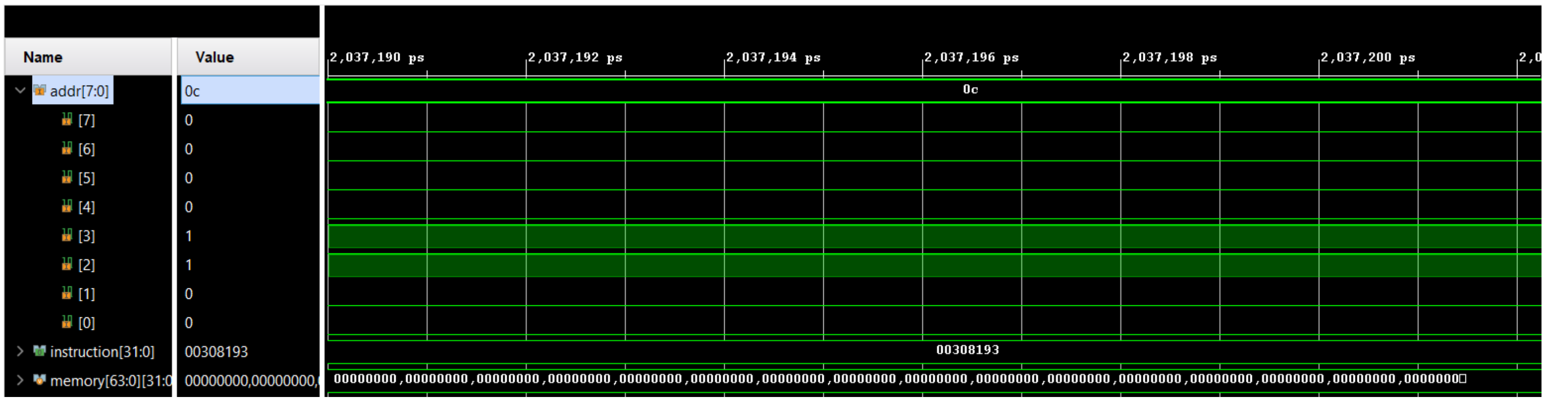
3. Result:

- Testbench:

```
`timescale 1ns / 1ps

module tb_InstMem();
    // Inputs
    reg [7:0] addr;

    // Output
    wire [31:0] instruction;

    // Instantiate the module want to test
    InstMem dut(addr, instruction);

    // Testcase
    initial
        begin
        //Case 1
        addr = 32'd12;
        #200;

        //Case 2
        addr = 32'd64;
        #200;

        //Case 3
        addr = 32'd32;
        #200;

        //Case 4
        addr = 32'd40;
        #200;

        //Case 5
        addr = 32'd52;
        end
endmodule
```

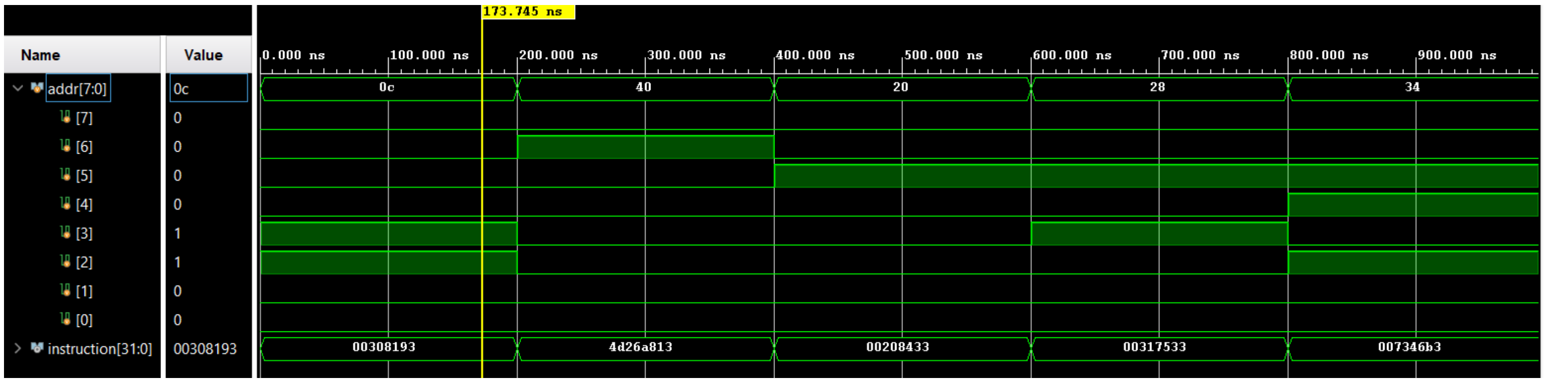
- Waveform:

- Using force constant:



Name	Value
addr[7:0]	0c
[7]	0
[6]	0
[5]	0
[4]	0
[3]	1
[2]	1
[1]	0
[0]	0
instruction[31:0]	00308193
memory[63:0][31:0]	00000000,00000000,

- Using testbench:



Name	Value
addr[7:0]	0c
[7]	0
[6]	0
[5]	0
[4]	0
[3]	1
[2]	1
[1]	0
[0]	0
instruction[31:0]	00308193

4. Appendix:

```
`timescale 1ns / 1ps
```

```
// Module definition
module InstMem(
    addr,
    instruction
    );

    // Define the input and output signals
    input [7:0] addr;
    output wire [31:0] instruction;

    // Define a 2D array to store 64 instructions each with 4 bytes (32
bits)
    reg [31:0] memory [63:0];
    integer i;

    // Define the Instruction memory modules' behaviour
    initial
        begin
        for (i = 18; i < 64; i = i + 1)
            begin
            memory[i] = 0;
            end
       //$readmemh ("memfile.dat", memory); //stored in hexadecimal
format
        memory[0] = 32'h00007033; // and r0, r0, r0 32'h00000000
        memory[1] = 32'h00100093; // addi r1, r0, 1 32'h00000001
        memory[2] = 32'h00200113; // addi r2, r0, 2 32'h00000002
        memory[3] = 32'h00308193; // addi r3, r1, 3 32'h00000004
        memory[4] = 32'h00408213; // addi r4, r1, 4 32'h00000005
        memory[5] = 32'h00510293; // addi r5, r2, 5 32'h00000007
        memory[6] = 32'h00610313; // addi r6, r2, 6 32'h00000008
        memory[7] = 32'h00718393; // addi r7, r3, 7 32'h0000000B
        memory[8] = 32'h00208433; // add r8, r1, r2 32'h00000003
        memory[9] = 32'h404404b3; // sub r9, r8, r4 32'hfffffffe
        memory[10] = 32'h00317533; // and r10, r2, r3 32'h00000000
        memory[11] = 32'h0041e5b3; // or r11, r3, r4 32'h00000005
        memory[12] = 32'h0041a633; // if r3 is less than r4 then r12 =
1 32'h00000001
        memory[13] = 32'h007346b3; // nor r13, r6, r7 32'hfffffff4
        memory[14] = 32'h4d34f713; // andi r14, r9, "4D3" 32'h000004D2
        memory[15] = 32'h8d35e793; // ori r15, r11, "8d3" 32'hfffff8d7
        memory[16] = 32'h4d26a813; // if r13 is less than 32'h000004D2
then r16 = 1 32'h00000001
        memory[17] = 32'h4d244893; // nori r17, r8, "4D2" 32'hfffffb2C
        end

    assign instruction = memory[addr >> 2]; // word aligned
```

```
endmodule
```

III. Register File:

1. Introduction:

A register file consists of a set of registers that can be read and written by supplying a register number to be accessed.

For reading from a register file we need the register number. For writing to a register we need three inputs: a register number, the data to write, and a clock that controls the writing into the register. Our register file has two read ports and one write port.

For this module we need:

- 7 inputs:
  - A clock signal (called clk)
  - A reset signal (called reset)
  - A writing signal (called rg_wrt_en)
  - 3 register numbers (2 for reading, called rg_rd_addr1 and rg_rd_addr2, and 1 for writing called rg_wrt_addr) have 5 bits for each
  - A writing data (called rg_wrt_data, has 32 bits)
- 2 outputs: 2 datas we read from register file (called rg_rd_data1 and rg_rd_data2, each has 32 bits)

2. Procedure:

- Create 32 registers. Each register has 32 bits.
- For reading: To read from the register file regardless of the clock or reset, we assign two registers through two output lines rg_rd_data1 and rg_rd_data2.
- For writing: To write to the register file, if reset input is zero, write enable signal (rg_wrt_en) is on, and in the rising edge (posedge) of the clock => write the data from input line rg_wrt_data to the register number rg_wrt_addr. Notice: The register 0 is always equal to 0.
- Whenever we have a reset signal we should reset the register file, set all registers to 32'h00000000.

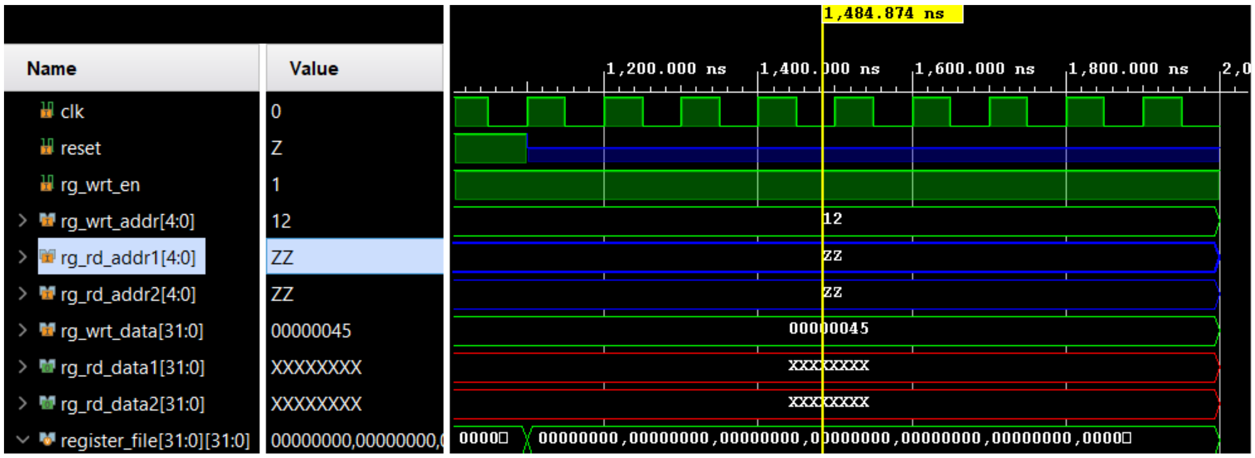
3. Result:

- Testbench:

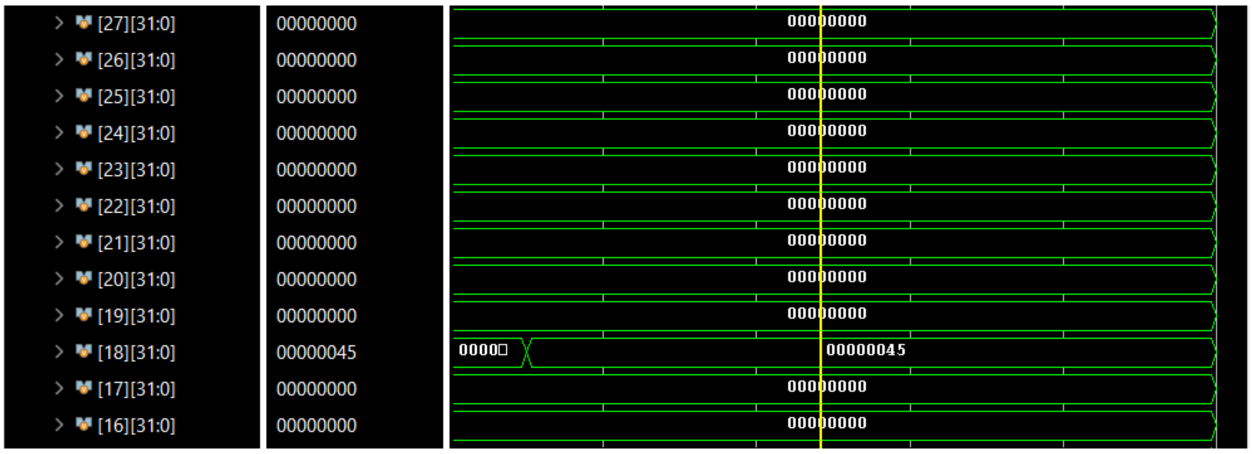
```
`timescale 1ns / 1ps

module tb_RegFile();
    // Inputs
    reg clk;
    reg reset;
    reg rg_wrt_en;
    reg [4:0] rg_wrt_addr;
    reg [4:0] rg_rd_addr1;
    reg [4:0] rg_rd_addr2;
    reg [31:0] rg_wrt_data;

    // Output
    wire [31:0] rg_rd_data1;
    wire [31:0] rg_rd_data2;

    // Instantiate the module want to test
    RegFile dut(clk, reset, rg_wrt_en, rg_wrt_addr, rg_rd_addr1,
rg_rd_addr2, rg_wrt_data, rg_rd_data1, rg_rd_data2);

    // Generate Clock
    initial
        begin
        clk = 1;
        forever #20 clk = ~clk;
        end

    // Test case for writing
    initial
        begin
        // Case 1 when reset is on and write flag is set
        reset = 1;
        rg_wrt_en = 1;
        rg_wrt_addr = 5'b10010;
        rg_wrt_data = 32'h45;
        #100;

        // Case 2 when reset is off and write flag is set
        reset = 0;
        #100

        rg_wrt_addr = 5'b00101;
        rg_wrt_data = 32'h89;
        #100;

        // Case 3 when reset is off and write flag is not set
        rg_wrt_en = 0;
        #100;

        // Case 4 test asynchronous reset
        #10;
        reset = 1;
        #100;

        // Case 5 test writing into register[0]
        reset = 0;
        rg_wrt_en = 1;
        rg_wrt_addr = 5'b00000;
        rg_wrt_data = 32'h32;
        #100;

        // Case 6 test reading after reset
        rg_wrt_addr = 5'b10010;
        rg_wrt_data = 32'h7b;
        end

    // Test case for reading
    initial
        begin
        rg_rd_addr1 = 5'b10010;
        rg_rd_addr2 = 5'b00101;
        #600;

        rg_rd_addr2 = 5'b00000;
        end
endmodule
```

- Waveform:

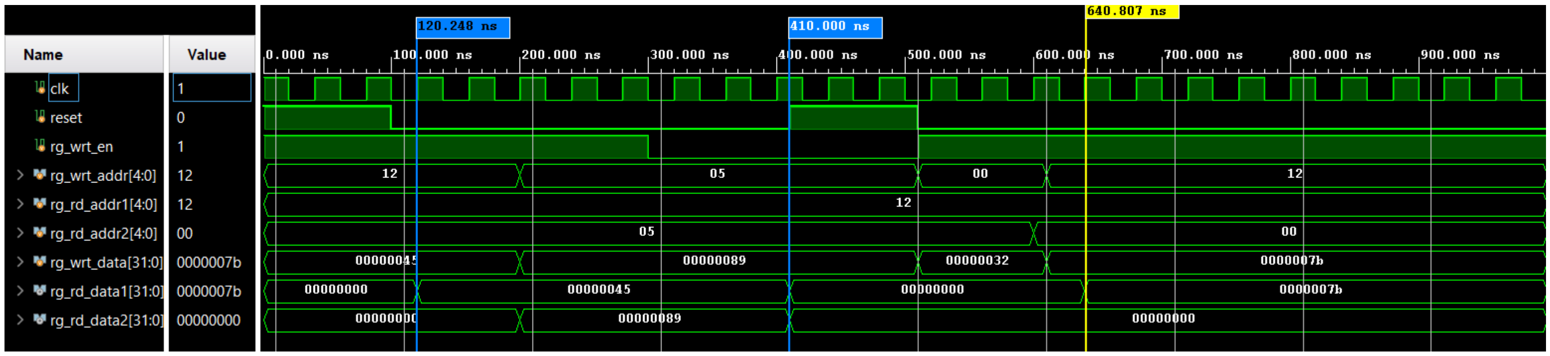
- Using force clock and force constant:



```
Name                      Value
clk                       0
reset                     Z
rg_wrt_en                 1
rg_wrt_addr[4:0]          12
rg_rd_addr1[4:0]          ZZ
rg_rd_addr2[4:0]          ZZ
rg_wrt_data[31:0]         00000045
rg_rd_data1[31:0]         XXXXXXXX
rg_rd_data2[31:0]         XXXXXXXX
register_file[31:0][31:0] 00000000,00000000,(
```



- Using testbench:



4. Appendix:

```
`timescale 1ns / 1ps

module RegFile(
    clk, reset, rg_wrt_en, // each 1 bit
    rg_wrt_addr, // 5 bits
    rg_rd_addr1, // 5 bits
    rg_rd_addr2, // 5 bits
    rg_wrt_data, // 32 bits
    rg_rd_data1, // 32 bits
    rg_rd_data2 // 32 bits
    );

    // Define the input and output signals
    input clk, reset, rg_wrt_en;
    input [4:0] rg_wrt_addr;
    input [4:0] rg_rd_addr1;
    input [4:0] rg_rd_addr2;
    input [31:0] rg_wrt_data;
    output [31:0] rg_rd_data1;
    output [31:0] rg_rd_data2;
    // Define the Register File modules ' behaviour
    // 32x32 bit array of registers
    reg [31:0] register_file [31:0];

    integer i;

    // writes are enabled on positive edge clock cycles
    always @ (posedge clk || reset)
        begin
        if (reset == 1'b1)
            begin
            for (i = 0; i < 32; i = i + 1)
                register_file[i] = 0;
            end
        else
            begin
            // if write is set
            if (rg_wrt_en == 1'b1 && rg_wrt_addr)
                register_file[rg_wrt_addr] = rg_wrt_data;
            end
        end

    // read
    assign rg_rd_data1 = register_file[rg_rd_addr1];
    assign rg_rd_data2 = register_file[rg_rd_addr2];

endmodule
```