

# Computer Architecture

## CECS 440

### LAB 5

CECS Department  
California State University, Long Beach

Spring 2020

Due on Friday 04/10/2020 by 8 pm

Through this course, we want to design a RISC-V Single Cycle Processor. We have already designed some of the modules needed in a single cycle processor. Here, we extend our module set. In the next lab, we will learn how to design a high-level module to connect the lower-level modules we have developed.

## 1 Single-cycle Processor

The building block of a single cycle processor is shown as follows. The instruction execution starts by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or an equality check (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file.

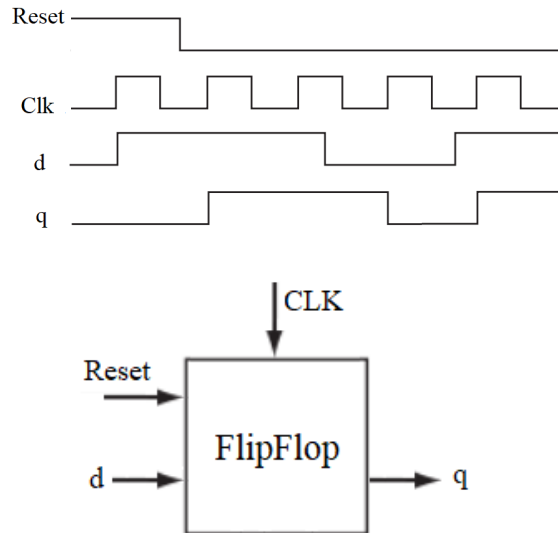
The blue lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

Some of the inputs (RegWrite, ALUSrc, ALUCC, MemRead, MemWrite, MemtoReg) are control signals which are derived by a module named "Control". The control unit is supposed to be designed later in Lab 7.



Here we want to design a Flip flop with an 8-bit d line and synchronous reset. Synchronous reset means reset is sampled with respect to the clock. In other words, when reset is enabled, it will not be effective till the next active clock edge (here rising edge).

Figure 3 : D Flip flop with a rising-edge trigger.



Use this code for the Entity part of the Flip flop.

Code 1: D Flip flop

```

1  'timescale 1ns / 1ps
2
3
4  // Module definition
5  module FlipFlop(
6      clk , reset ,
7      d ,
8      q
9  );
10
11  // Define the input and output signals
12
13  // Define the D Flip flop modules' behaviour
14
15  endmodule //FlipFlop

```

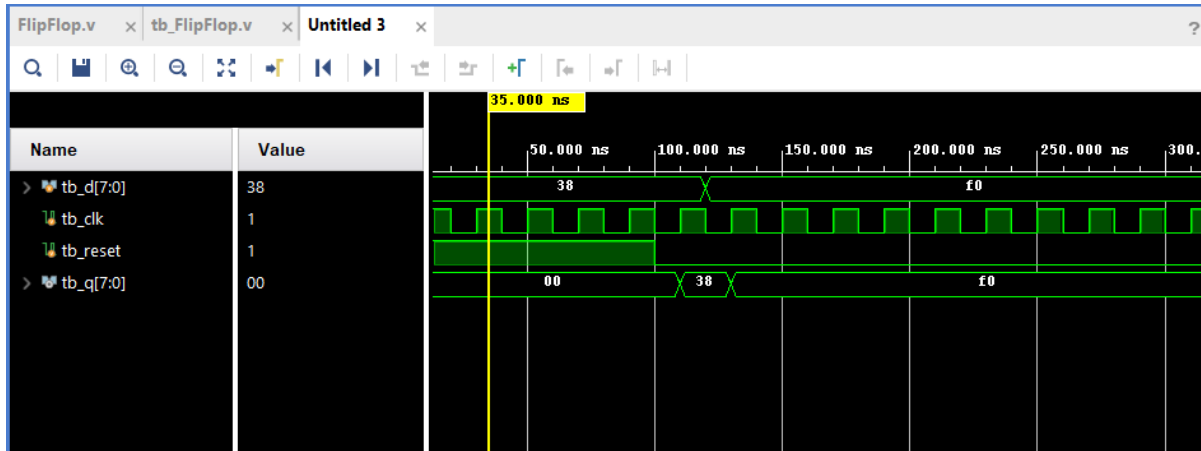
This Flip flop holds the program counter (PC), which is a register that holds the address of the current instruction. We will send this address to the instruction memory to read the current instruction.

**Test your module:** Complete the module for a D-FlipFlop and also write a test bench to test the following testing scenario:

- In the test bench define the clock signal with the duration of 20 ns.
- Activate the reset signal for the first 100 ns and the output signal have to be set to zero (at the rising edge of the clock) regardless of the input values.

- After 100 ns the reset signal is inactive and the output is reflecting the input values. Note that the output may change only at the rising edge of the clock.

Here is the simulation result for a similar testing scenario:



### 1.1.2 Adder

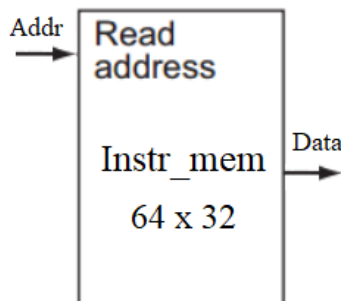
We need an adder to increment the PC with 4 to get the address of next instruction. This adder can be implemented with a simple statement in Verilog:

**PC\_Next = PC + 4;**

Using "+" operation in Verilog you are able to design this adder.

### 1.1.3 Instruction Memory

Instruction memory has one input which is the address line (called Addr, has 8 bits), and one output which is the instruction we read from the memory (called Data, has 32 bits). With the 8 bits address which comes from the PC, we can address  $2^8$  bytes (This memory is byte addressable). An instruction has 4 bytes (32 bits). So we can store  $2^8/4 = 64$  instructions in our instruction memory.



In the instruction memory, we need internal storage to store the instructions. We need to define a 2D array to store 64 instructions each with 4 bytes (32 bits). So, Define Instruction\_memory as a new **type** which is a 64 x 32 2D array and define signal **memory** from type Instruction\_memory. Use the following Verilog statement to define your instruction memory as a 2D array with the name of "memory":

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

Then we need to actually store some instructions (which are 32 bits data) in the signal that we defined as internal storage. Use the code below for the instruction memory.

Code 2: Instruction Memory

```

1  'timescale 1ns / 1ps
2
3
4  // Module definition
5  module InstMem(
6      input [7:0] addr,
7      output wire [31:0] instruction
8  );
9
10 // Define the input and output signals
11
12 // Define the Instruction memory modules' behaviour
13
14 endmodule //InstMem

```

You need to save the following instruction codes in your Instruction memory for further usage. The comments show the meaning of each instruction and the expected output.

```

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'hffffffe
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'hffffff4
memory[14] = 32'h4d34f713;    // andi r14, r9, "4D3"    32'h000004D2
memory[15] = 32'h8d35e793;    // ori r15, r11, "8d3"    32'hffff8d7
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'hffffb2C

```

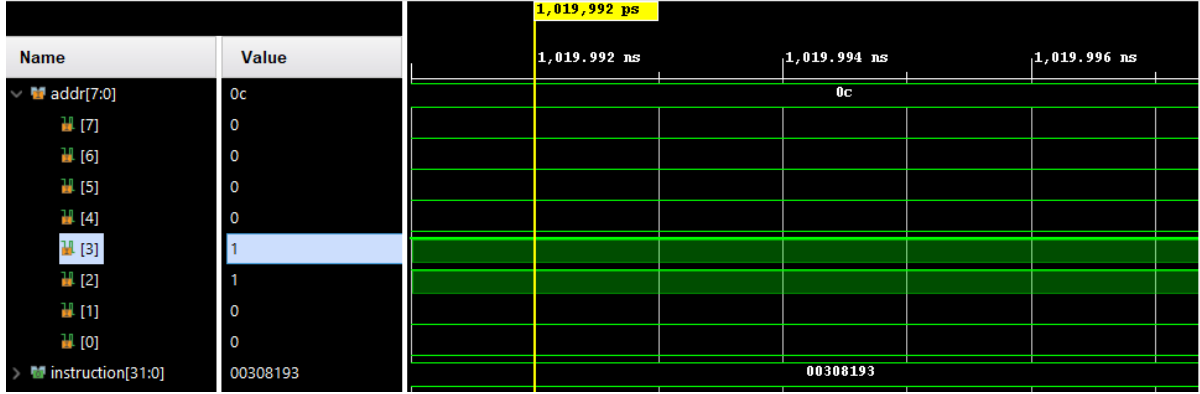
**Test your module:** you may try to use "Force Constant" method in Verilog to test this module. Try to set the addr signal to different numbers between 0-16 and see if the output is equal to the corresponding instruction code. Here are a few test cases:

- addr[7:2]: 16 (decimal) (row number) the expected output instruction: 32'h4d26a813
- addr[7:2]: 3 (decimal) (row number) the expected output instruction: 32'h00308193

**Note:**  $\text{addr}[7:0]$  gives the first byte number for each instruction and  $\text{addr}[7:2]$  gives the exact row number for each instruction in our memory array.

**Important note:** each line of the instruction memory includes 8 bits and each instruction is 32 bits. This means that you need to use bits  $[7:2]$  of the address line to point to a new instruction.

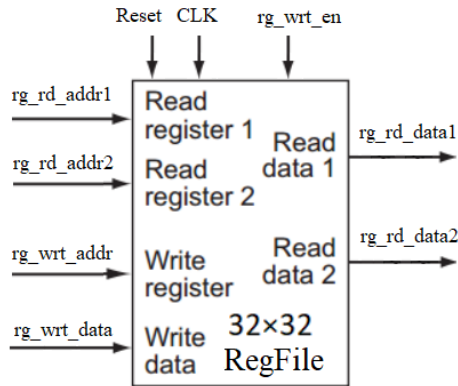
Here is the simulation result for the second test case:



#### 1.1.4 Register File

One structure that is central to our datapath is a register file. 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 only 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. This register file is shown in Figure 4.

Figure 4 : Register File.



We have two register read addresses ( $\text{rg\_rd\_addr1}$ ,  $\text{rg\_rd\_addr2}$ ). These two input signals have 5 bits. with 5 bits we can address  $2^5 = 32$  registers. To read from the register file regardless of the clock or reset we will send two registers through two output lines  $\text{rg\_rd\_data1}$ , and  $\text{rg\_rd\_data2}$  (these are 32 bits). To write to the register file if Reset input is zero and write enable signal ( $\text{rg\_wrt\_en}$ ) is '1' (on) then in the rising edge (posedge) of the clock we will write the data from input line  $\text{rg\_wrt\_data}$  to the register number  $\text{rg\_wrt\_addr}$ .

Unlike what we had for Flip flop, here we want to have an asynchronous reset. Means regardless of the

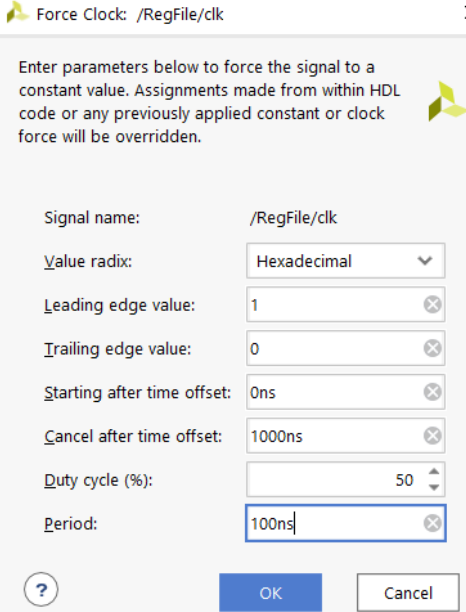
clk, whenever we have Reset signal we should reset the register file (set all registers to 32'h00000000). Here is the Entity of the register file. Same as instruction memory we need some internal storage for register file.

Code 3: Register File

```
1  'timescale 1ns / 1ps
2
3
4  // Module definition
5  module RegFile(
6      clk, reset, rg_wrt_en,
7      rg_wrt_addr,
8      rg_rd_addr1,
9      rg_rd_addr2,
10     rg_wrt_data,
11     rg_rd_data1,
12     rg_rd_data2
13 );
14
15 // Define the input and output signals
16
17 // Define the Register File modules' behaviour
18
19 endmodule //RegFile
```

**Test your module:** you may try to use "Force Constant" to test this module. Follow the following steps:

- **Set the Clock:** After running the simulation, right click on the clock signal and push "Force clock". The following window will pop up. Set the numbers as you see. We will have 10 clock cycles each with a duration of 100 ns.

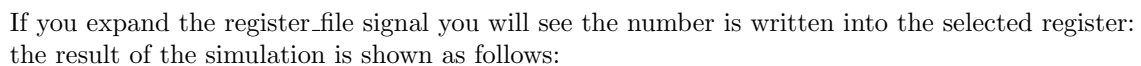


The image shows a "Force Clock" dialog box for the signal `/RegFile/clk`. The dialog contains the following fields and values:

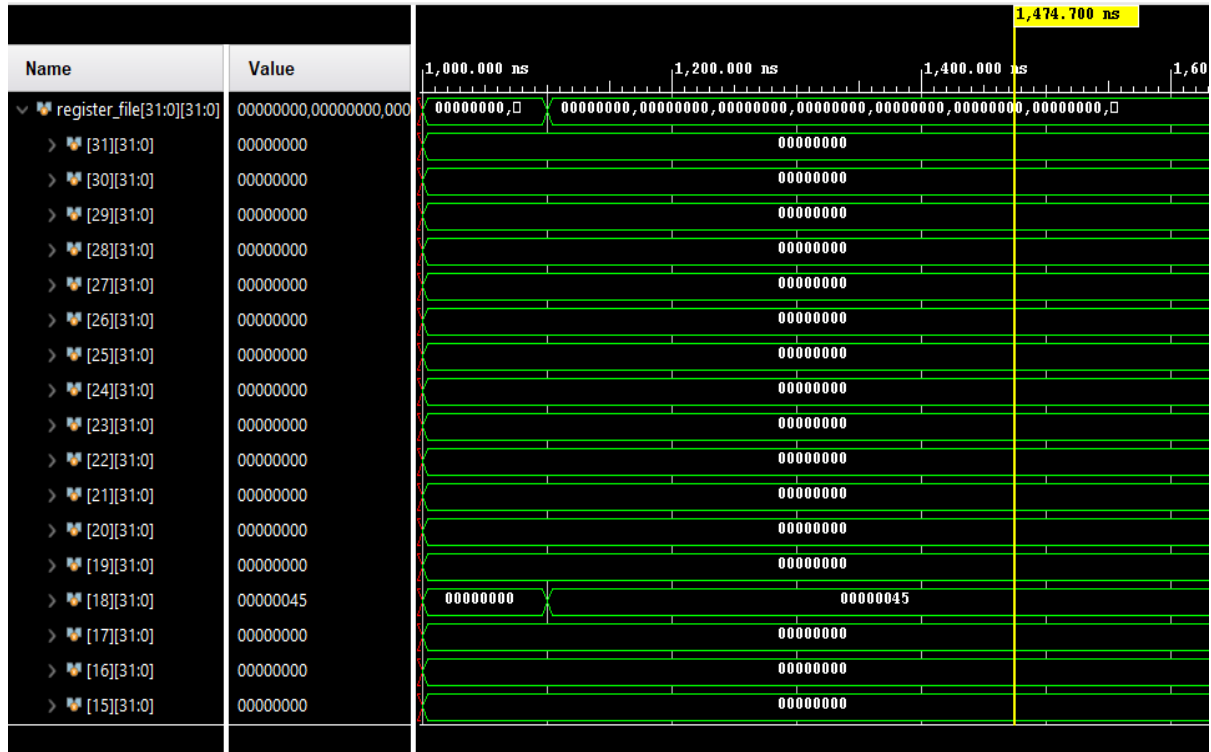
| Field                       | Value                     |
|-----------------------------|---------------------------|
| Signal name:                | <code>/RegFile/clk</code> |
| Value radix:                | Hexadecimal               |
| Leading edge value:         | 1                         |
| Trailing edge value:        | 0                         |
| Starting after time offset: | 0ns                       |
| Cancel after time offset:   | 1000ns                    |
| Duty cycle (%):             | 50                        |
| Period:                     | 100ns                     |

At the bottom of the dialog are three buttons: a help button (question mark icon), an "OK" button, and a "Cancel" button.

- the result of the simulation is shown as follows:



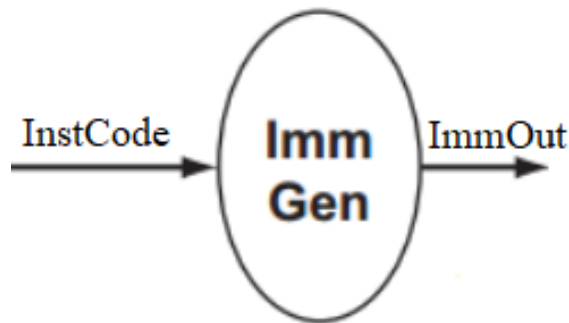




### 1.1.5 Immediate Generator

Immediate Generator (ImmGen) receives the 32-bits instruction, and based on the opcode decides which bits of the instruction should be interpreted as the 12-bit immediate. Then sign extends the 12-bits immediate into 32-bits. The opcode part of the instruction is bit (7 bits). For the current set of instructions the opcode could be "0010011" or "0110011". For those instructions with opcode "0110011" we don't need an immediate value and we can define the output to be 32'h00000000. For other instructions with opcode "0010011" the 12 bits immediate value is instruction (31 downto 20).

Sign extension means to copy the sign bit. As an example, if the 12-bits immediate are 12'h900 then the output of this module should be 32'hffff900. Or if the 12-bits immediate are 12'h700, then the output will be 32'h00000700. Below you see the block diagram. This module is designed for you as you can see the coding as follows:



Code 4: Immediate Generator

```

1  module ImmGen(
2      InstCode ,
3      ImmOut
4  );
5  input [31:0] InstCode;
6  output reg [31:0] ImmOut;
7
8  always @(InstCode)
9  begin
10 case (InstCode[6:0])
11
12     7'b0000011 :
13         ImmOut = {InstCode[31]? {20{1'b1}}:20'b0 , InstCode[31:20]};
14     7'b0010011 :
15         ImmOut = {InstCode[31]? {20{1'b1}}:20'b0 , InstCode[31:20]};
16     7'b0100011 :
17         ImmOut = {InstCode[31]? {20{1'b1}}:20'b0 , InstCode[31:25], InstCode[11:7]};
18     7'b0010111 :
19         ImmOut = {InstCode[31:12], 12'b0};
20 default
21         :
22         ImmOut = {32'b0};
23 endcase
24 end
25 endmodule

```

### 1.1.6 ALU

The ALU is the same ALU we designed in Lab4.

### 1.1.7 resultmux

The MUX on the output of the Data Memory will decide whether the writing data (to the register file) should come from the ALU (ALU\_Result) or come from the Data Memory (DataMem\_read). Use the following code for two MUX in your Datapath design.

Code 5: 2:1 MUX

```

1  // Module definition
2  module MUX21 (
3      D1 , D2 , S , Y
4  );
5
6  input S; // select line
7  input [31:0] D1;
8  input [31:0] D2;
9
10 output [31:0] Y;
11
12 assign Y = (!S & D1) | (S & D2);
13
14 endmodule // Mux21

```

## 2 Assignment Deliverables

Your submission should include the following items:

- A screenshot of your design code for each module. (FlipFlop.v, Instr\_mem.v, RegFile.v)
- A few lines explaining how you designed each module.
- An screenshot of the test cases you have tried for each module to show that module is working correctly.
- A report in pdf format. Your report should have all details for your design + screenshot of the wave. Each report should include group members if you are working as a group.

Note1: Upload your report to the **Beachboard** before deadline.

Note2: Use the code samples that are given in the lab description. **The Module part of your code should exactly look like the code sample otherwise you lose points.**