BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
CS F342
Lab Sheet – 5
Topic: Register File and Memory Implementation

Learning objectives:

1) **Designing a sample 32-bit register file for a MIPS processor**
2) **Creating an Instruction Memory and Data Memory for the MIPS processor**

Task 1: Register File

In the last lab session, we have designed the ALU and the Control Unit of the ALU. In this session, we will design the register file for MIPS processor. Remember that MIPS has 32-bit registers. Therefore, the register file should be 32-bit wide. The formal problem statement is as follows:

(1) Implement a register file using verilog HDL. The following circuit diagram explains reading and writing data to a register in the register file. Assume there are thirty-two 32-bit registers in a register file. Each register is made up of D flip-flop. Register file can read two registers at a time. Two register numbers and clock are the input to the read register module and the output will be the register data as shown in the diagram. Two multiplexers are used for selecting the register for the inputted register number. For writing to the register file, a 5:32 decoder has to be designed which takes register number as input and enables required register. Input to register write module will be register data, register number, write enable signal and clock.

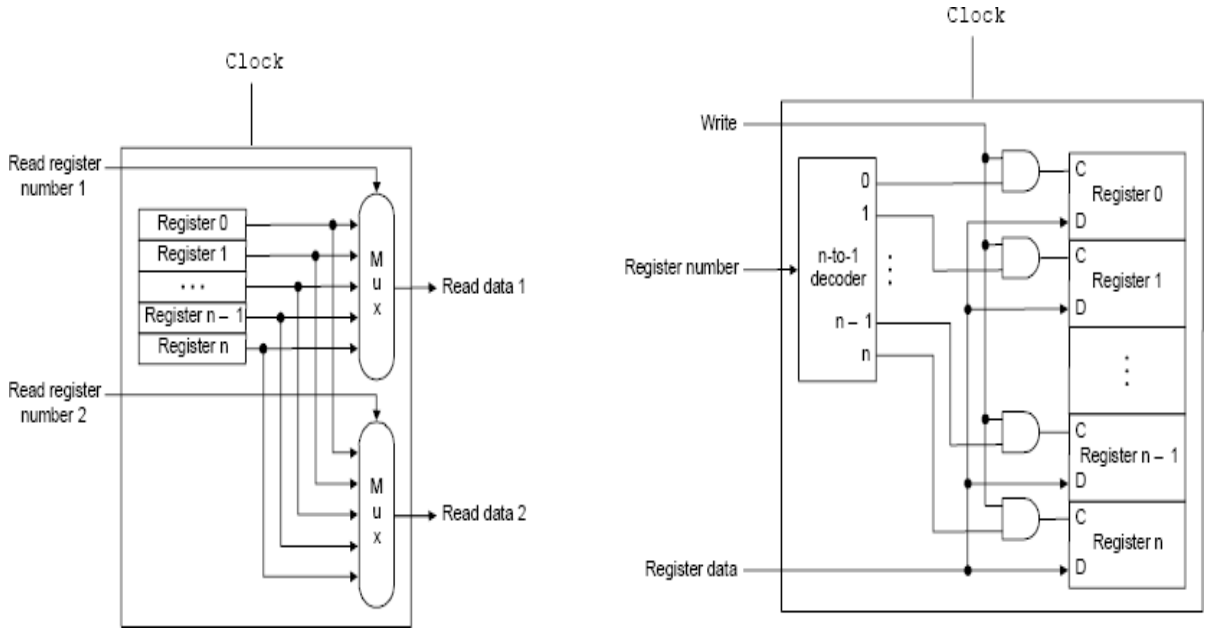

Fig1: Read and Write Operations of the Register File

The block diagram representation of the register file will look like as follows.

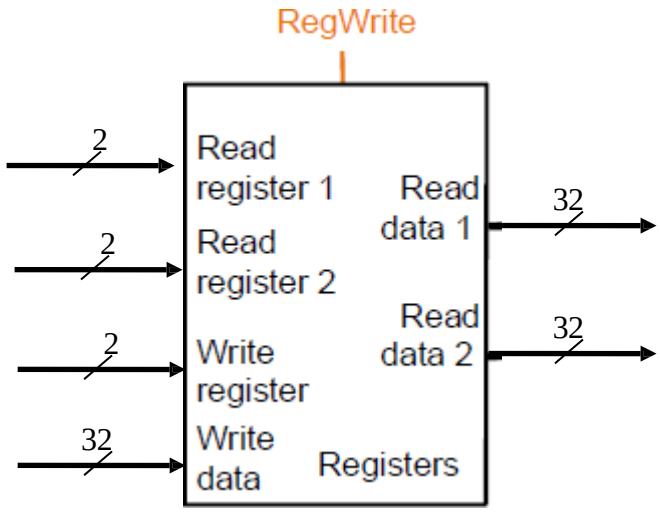

Fig2: Block diagram of the Register File

We will develop the components of the register file part-by-part in the following sub-tasks. In addition, it is recommended to test each of the components individually to ease the final integration into the register file.

Sub task 1.1: Designing the registers

You have already designed the D_FF using behavioral modeling in previous labs. You can use the same module to develop a 32-bit register. The D-FF is an active-low reset FF, i.e., it is reset when reset is "0".

The outline of these modules is as follows:

```
(1)  module d_ff(q,d,clk,reset);

(2)  module reg_32bit(q,d,clk,reset); // Register
```

Test the 32-bit Register using the following test bench.

```
module tb32reg;
reg [31:0] d;
reg clk,reset;
wire [31:0] q;
reg_32bit R(q,d,clk,reset);
always @(clk)
#5 clk<=~clk;
initial
begin
clk= 1'b1;
reset=1'b0;//reset the register
#20 reset=1'b1;
#20 d=32'hAFAFAFAF;
#200 $finish;
end
endmodule
```

Exercise: Design a 32-bit register module and test it with the above test bench.

Sub task 1.2: Designing the decoder and multiplexer

This is pretty simple and straight forward. The output of each of these registers i.e. the content of these registers is connected to each of the inputs of the multiplexer (mux). So the register whose data is to be read on a particular read port is selected using this mux. Thus it will be 32:1 mux as we have 32 registers. What will be the width of the input and output data lines of this mux? i.e. it is ____bit wide. What will be the width of the select lines of this mux?

Also during the write operation we specify the address of the register to be written i.e. "Write Register" in Fig2. This address is used to enable the writing to a register by using a decoder. So we use a 2:4 decoder.

The outline of these modules is given below.

```
(3)  module mux4_1(regData,q1,q2,q3,q4,reg_no);
(4)  module decoder2_4 (register,reg_no);
```

Exercise: Design 4:1 mux and 2:4 decoder.

Sub task 1.3: Integrating the sub components

Refer to Fig.1 for writing opeartion. As we stated before we have to enable writing of only one register depending on the address specified by "Write Register". So one way we implement this by anding the Register File clock, decoder output for a register and RegWrite Signal. The output of this and gate is provided as input to the clock of the corresponding register. The scheme looks as shown below:

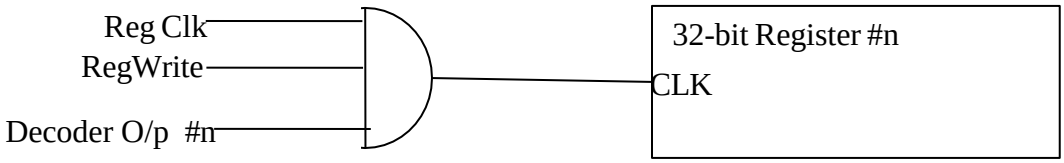

Fig. 3 Clock Gating

Therefore, for Regsiter #0, its CLK will be ANDing of Reg Clk, Reg Write and Decoder O/p #0. So only when register #0 is written, it will be provided with a Clock input. At all other times clock input of the register is 0. This scheme is formally called as Clock gating. This technique is simple and also saves power.

For this lab, you should NOT implement clock gating explicitly.

Clock gating at RTL level can:

- Cause **glitches** on the gated clock signal
- Break synchronous behavior across the datapath
- Create synthesis and timing issues in real hardware

Instead of modifying the clock signal, we will use a **clean, synchronous design**:

1. Use a single, ungated clock for all registers.
2. Use the decoder output and RegWrite signal as an **enable condition** inside the register write logic:

```
always @(posedge clk) begin
    if (!rst)
        ……….
    else if (RegWrite && decode_out[i])
        ……..
end
```

Now create a module for Register File that would instantiate the modules created in the previous tasks. The outline of this module is as follows.

```
module
RegFile(clk,reset,ReadReg1,ReadReg2,WriteData,WriteReg,RegWrite,ReadData1,ReadData2);
```

Now the question is that how will we test the functionality of this Register file? Since the number of registers is small, we will fill the registers with some data in the test bench first and do some read operations to check if it reads correctly.

Remember:

i. First reset the registers initially.
ii. In addition, a clock has to be "created".

Exercise: Integrate the component modules and write a suitable test bench to test the functionality of the Register file using a suitable test bench.

Task 2: Memory Modules

In this section we will create an instruction memory and data memory module to use with our MIPS processor implementation in the next lab.

Sub task 2.1: Instruction Memory

The **Instruction Memory** is a read-only memory that holds the program to be executed by the processor. The CPU fetches an instruction from this memory every clock cycle and the address of the instruction to be fetched is provided by the Program Counter (PC).

For this lab, we use a **32-word deep memory**, where each word is 32 bits (1 instruction).
The PC is **word-aligned**. This means:

- Only bits [6:2] of the PC are used to index the memory (5 bits → 32 entries).
- Each memory entry corresponds to an instruction.

We use a **behavioral model** with a reg [31:0] memory [31:0]; array in Verilog.
The program is preloaded using an initial block (you will hardcode your instructions here).

Here is a sample instruction memory module, with a program to find the sum of the first 10 natural numbers.

```
module instr_mem(input [31:0] addr, output reg [31:0] instr);
   reg [31:0] memory [31:0];

   initial begin
      // sum = 0
      memory[0] = 32'b100011_00000_01000_0000_0000_0000_0000; // lw $t0, 0($zero)  ($t0 = 0)
      // i = 1
      memory[1] = 32'b100011_00000_01001_0000_0000_0000_0001; // lw $t1, 1($zero)  ($t1 = 1)
      // limit = 10
      memory[2] = 32'b100011_00000_01100_0000_0000_0000_1011; // lw $t4, 11($zero)  ($t4 = 10)
      // loop: $t3 = $t1 - $t4
      memory[3] = 32'b000000_01001_01100_01011_00000_100010; // sub $t3, $t1, $t4
      // if $t3 == 0, branch to end (offset = 6, to memory[10])
      memory[4] = 32'b000100_01011_00000_0000_0000_0000_0110; // beq $t3, $zero, 6
      // sum += i
      memory[5] = 32'b000000_01000_01001_01000_00000_100000; // add $t0, $t0, $t1
      // i++: $t1 = $t1 + 1
      memory[6] = 32'b100011_00000_01010_0000_0000_0000_0001; // lw $t2, 1($zero) ($t2 = 1)
      memory[7] = 32'b000000_01001_01010_01001_00000_100000; // add $t1, $t1, $t2
      // jump to loop (address = 3)
      memory[8] = 32'b000010_00000_00000_00000_00000_000011; // j 3
      // NOP (padding)
      memory[9] = 32'b000000_00000_00000_00000_00000_000000; // NOP
      // end: NOP
      memory[10] = 32'b000000_00000_00000_00000_00000_000000; // NOP
      memory[11] = 32'b000000_00000_00000_00000_00000_000000;
      memory[12] = 32'b000000_00000_00000_00000_00000_000000;
      memory[13] = 32'b000000_00000_00000_00000_00000_000000;
      memory[14] = 32'b000000_00000_00000_00000_00000_000000;
      memory[15] = 32'b000000_00000_00000_00000_00000_000000;
      memory[16] = 32'b000000_00000_00000_00000_00000_000000;
      memory[17] = 32'b000000_00000_00000_00000_00000_000000;
      memory[18] = 32'b000000_00000_00000_00000_00000_000000;
      memory[19] = 32'b000000_00000_00000_00000_00000_000000;
      memory[20] = 32'b000000_00000_00000_00000_00000_000000;
      memory[21] = 32'b000000_00000_00000_00000_00000_000000;
      memory[22] = 32'b000000_00000_00000_00000_00000_000000;
      memory[23] = 32'b000000_00000_00000_00000_00000_000000;
      memory[24] = 32'b000000_00000_00000_00000_00000_000000;
      memory[25] = 32'b000000_00000_00000_00000_00000_000000;
      memory[26] = 32'b000000_00000_00000_00000_00000_000000;
      memory[27] = 32'b000000_00000_00000_00000_00000_000000;
      memory[28] = 32'b000000_00000_00000_00000_00000_000000;
      memory[29] = 32'b000000_00000_00000_00000_00000_000000;
      memory[30] = 32'b000000_00000_00000_00000_00000_000000;
      memory[31] = 32'b000000_00000_00000_00000_00000_000000;
   end

   always @(*) begin
      instr = memory[addr[6:2]]; // word-aligned fetch
   end

endmodule
```

Note that we are expecting the lw instruction to load the calculated address to the destination register itself, which we will implement using the data memory module.

Sub task 2.2: Data Memory

The **Data Memory** is used to store and retrieve data during program execution. It is active during the lw and sw instructions.
For simplicity, we are going to implement a very basic *read-only* data memory that just returns the address as the output data. This will help us verify the datapath functionality in later labs.

```
module data_memory(input [31:0] addr, output reg [31:0] data_out);
    always @(*) begin
    data_out = addr;
    end
endmodule
```

Take Home Exercise: Try to implement a real data memory module, where values can be written and stored as well. The current implementation doesn't support the sw operation.
Remember to add clock, MemWrite, MemRead and WriteData ports to the data memory.
