Lab 5

Pre-Knowledge

In order to complete this lab you will need an understanding what a register is.

Objective

In this lab, we will design a register file containing 16-bit registers. We will then hook up the register file to our 16-bit ALU (designed in Lab 4).

Lab Execution

Guidelines: For this lab, the online Blackboard submission will consist of

- A report following the lab report template which **includes your waveforms, and your testing results table** (See table at the end of this document). Your waveforms should be included in your report. Your waveforms should not contain unnecessary information on the screen. Video of your lab is required.

Tools: Xilinx ISE. If you have forgotten how to use ISE please review lab 1.

VHDL Programming instructions: For this lab your first goal is to create a register file and then connect the register file with the 16-bit ALU you constructed last week (Lab 4). To assist you, you have been provided the entity of the register as well as detailing how registers work and how you might code one.

Your entity should be as follows:

```
5  Entity RegisterFile is
6      port(
7          clk     :   in   std_logic;      -- positive edge triggered clock
8          clear_n :   in  std_logic;       -- asynchronous, active low reset
9
10         a_addr  :   in   std_logic_vector( 3 downto 0); -- register select for input a
11         a_data  :   in   std_logic_vector(15 downto 0); -- input data port
12         load    :   in   std_logic;                     -- load enable/enable signal for "loading"
13
14         b_addr: in   std_logic_vector( 3 downto 0); -- register select for output b
15         c_addr: in   std_logic_vector( 3 downto 0); -- register select for output c
16
17         b_data: out std_logic_vector(15 downto 0);  -- first output data port
18         c_data: out std_logic_vector(15 downto 0)   -- second output data port
19     );
20 End RegisterFile;
```

The registerfile you will create using the above entity will have the following properties.

1. The register file should have 16 registers, each should be 16 bits wide.
2. The register will have 7 inputs (clk, clear, load, a_addr, a_data, b_addr, and c_addr) and two outputs (b_data and c_data).
    a. Clk : the system clock
    b. Clear: sets all values in the register to 0. **It is active (when it's) low = active low**.
    c. Load: When the load signal is asserted, a rising edge on clk causes the data on a_data to be stored in the register identified by a_addr.
    d. a_addr – The address that the data in a_data will be stored at on a positive clk edge **if the load signal is asserted**
    e. a_data – the data that will be stored at the address referenced by a_addr on a positive clk edge **if the load signal is asserted**
    f. b_addr & c_addr : the address of the data that will be output at b_data and c_data.
        i. NOTE : THESE SIGNALS AND THEIR OUTPUTS ARE ASYNCHRONOUS AND INDEPENDENT, THEY DO NOT DEPEND ON CLK!!!!!!!!!!
    g. b_data & c_data: the data that will be output based on b_addr and c_addr.
        i. NOTE : THESE SIGNALS ARE ASYNCHRONOUS, THEY DO NOT DEPEND ON CLK!!!
3. As mentioned above, if load is asserted then the value in a_data will be placed in memory at the address specified by a_addr on a positive clock edge.
4. As mentioned above, the outputs b_data and c_data will output the data in memory at the address specified by b_addr and c_addr respectively. This happens instantaneously regardless of the clock (it is asynchronous).
5. **Register 0 always contains the value 0. Writing to register 0 is ignored.**
6. **Register 1 always contains the value 1. Writing to register 1 is ignored.**
7. When the clear signal is 0, all registers (other than register 1) are reset to 0. Note that this is asynchronous.

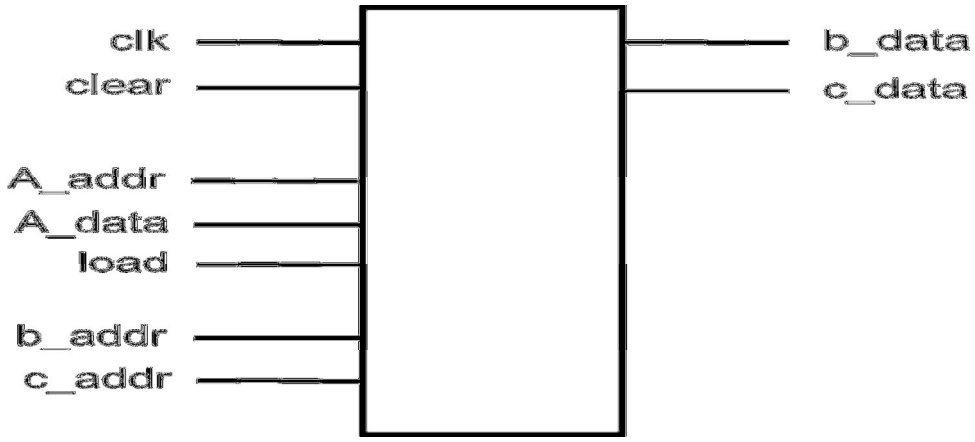
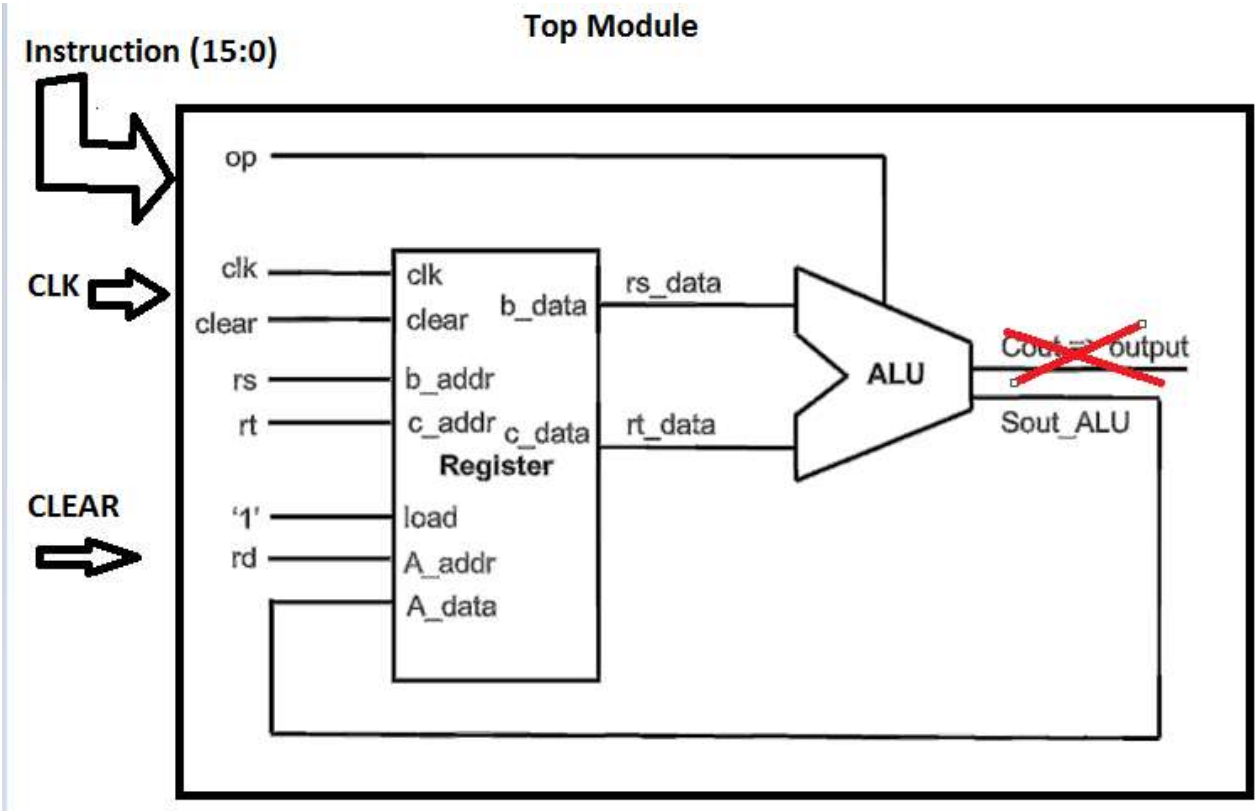


Figure 1: A diagram of your register file.

Once you have finished coding your register (**We will skip creating a testbench for the register file**) you will create a new module. Inside this module you will include both your register and the 16-bit ALU components. You will then connect these components as shown in the following diagram. Where does rs, rt, rd come from? **Please note we won't be using Cout coming from the 16-bit ALU. Feel free to remove it or tie it to a signal called output (leave it dangling).**



You will then test that your components work by creating a testbench that runs the following commands. Again, a testbench has already been provided in the ISE project. Please make sure in your waveform, that you show:

1. A hexadecimal radix for your instruction (I don't need to see the individual bits/expanded view)
2. Your registerFile is in signed decimal format for all of your registers in the register file. **You will need to manually add this after bringing up your waveform.**

Using your waveform, please fill in the blanks and turn this table along with your waveform in with your report.

Instruction	Op	rd	rs	rt	Value (rd)
ADD R2, R1, R1	0x0	0x2	0x1	0x1	2
ADD R3, R2, R1					
ADD R4, R3, R2					
SUB R5, R4, R3					
SUB R1, R5, R1					
SUB R0, R3, R0					
AND R6, R1, R1					
AND R7, R3, R0					
OR R8, R4, R1					

Review the instruction set architecture of the MIPS processor

Instruction	Description	Opcode bits 15:12	Rd bits 11:8	Rs bits 7:4	Rt bits 3:0
ADD Rd, Rs, Rt	Rd := Rs + Rt	0	0-15	0-15	0-15
ADDI Rd, Rs, Imm	Rd := Rs + SignExt(Imm)	4	0-15	0-15	Imm
SUB Rd, Rs, Rt	Rd := Rs - Rt	1	0-15	0-15	0-15
SUBI Rd, Rs, Imm	Rd := Rs - SignExt(Imm)	5	0-15	0-15	Imm
AND Rd, Rs, Rt	Rd := Rs and Rt	2	0-15	0-15	0-15
OR Rd, Rs, Rt	Rd := Rs or Rt	3	0-15	0-15	0-15
SLT Rd, Rs, Rt	if Rs < Rt then Rd := 1 else Rd := 0	7	0-15	0-15	0-15
LW Rd, off(Rs)	Rd := M[off + Rs]	8	0-15	0-15	offset
SW Rd, off(Rs)	M[off + Rs] := Rd	C	0-15	0-15	offset
BNE Rd, Rs, Imm	if Rd != Rs then pc := pc + 2 + addr [1]	9	0-15	0-15	offset
JMP Address	pc := JumpAddress [2]	B	12-bit offset		

On a branch, the offset value specifies the number of *words* relative to the next instruction to branch to, not the number of *bytes*.

On a jump, the target address is calculated as following:

The high-order three bits are the high-order three bits of the current PC + 2.

The next twelve bits are the 12-bit offset value from the JMP instruction.

The least significant bit is always 0.