ECE 449
Register File & ALU

Register File and ALU

In this lab, you will write HDL code for a register file and an ALU. You may decide to design the two components different from those discussed in this document. However, the general structure of the components should be the same.

Register File

Figure 1, depicts the schematic for the register file. The register file has eight 16-bit registers: **r0**, **r1**, **r2, r3, r4, r5, r6, r7**. The register file can read two registers and write one register in a single cycle.

In a read operation, the register file gets the index of two registers through ***rd_index1*** and ***rd_index2***. Register file drives ***rd_data1*** and ***rd_data2*** with the content of registers corresponding to *rd_index1* and *rd_index2*. Depending on your design, a read operation may be asynchronous or synchronous. In this lab, however, we assume an asynchronous read operation for the register file.

For writing to a register, ***wr_en*** should be one. The register file writes the value received on ***wr_data*** to the register determined by ***wr_index***. The write operation is synchronous and the data is written on the falling edge of the clock (you may design a different writing scheme for the register file). All the other registers in the processor work on the rising edge. Having such a register file enables the processor to write data on the falling edge and read the new data during the same cycle.

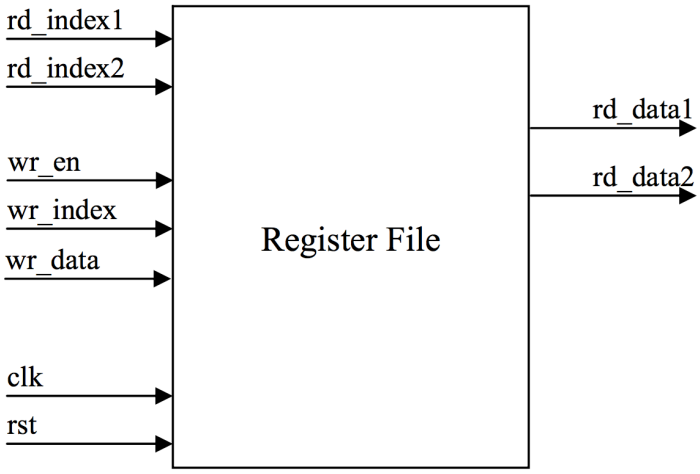

Figure 1. Register file.

Figure 2 shows the suggested VHDL code for the register file. Lines 22 to 35 describe the write operation. In VHDL, a **process** block models a sequential components. If ***rst*** is active, all internal registers are initialized to **0**. Otherwise, if the ***write_enable*** is **1**, one of the registers is selected by ***wr_index*** and is written with ***wr_data***.

Lines 38 to 47 describe read operation. *rd_index1* determines which register should drive *rd_data1*.

You should complete **process** block for write operation (line 31) and assignment statement (line 48) for read operation.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4
5   entity register_file is
6   port(rst : in std_logic; clk: in std_logic;
7   --read signals
8   rd_index1, rd_index2 : in std_logic_vector(2 downto 0);
9   rd_data1, rd_data2: out std_logic_vector(15 downto 0);
10  --write signals
11  wr_index: in std_logic_vector(2 downto 0);
12  wr_data: in std_logic_vector(15 downto 0);
13  wr_enable: in std_logic);
14  end register_file;
15
16  architecture behavioural of register_file is
17
18  type reg_array is array (integer range 0 to 7) of std_logic_vector(15 downto 0);
19  --internals signals
20  signal reg_file : reg_array; begin
21  --write operation
22  process(clk)
23  begin
24    if(clk='0' and clk'event) then if(rst='1') then
25       for i in 0 to 7 loop
26          reg_file(i)<= (others => '0');
27       end loop;
28    elsif(wr_enable='1') then
29       case wr_index(2 downto 0) is
30       when "000" => reg_file(0) <= wr_data;
31       --fill this part
32       when others => NULL; end case;
33     end if;
34     end if;
35  end process;
36
37  --read operation
38  rd_data1 <=
39  reg_file(0) when(rd_index1="000") else
40  reg_file(1) when(rd_index1="001") else
41  reg_file(2) when(rd_index1="010") else
42  reg_file(3) when(rd_index1="011") else
43  reg_file(4) when(rd_index1="100") else
44  reg_file(5) when(rd_index1="101") else
45  reg_file(6) when(rd_index1="110") else reg_file(7);
46
47  rd_data2 <=
48  --fill this part
49
50  end behavioural;
```

Figure 2. VHDL code for register file.

Figure 3 shows the test bench for the register file. The test bench writes five values to five registers and then reads those registers. Simulate the VHDL version of the register file.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use work.all;
5
6  entity test_alu is end test_alu;
7
8  architecture behavioural of test_alu is
9  component register_file port(rst : in std_logic; clk: in std_logic;
10 rd_index1, rd_index2: in std_logic_vector(2 downto 0);
11 rd_data1, rd_data2: out std_logic_vector(15 downto 0);
12 wr_index: in std_logic_vector(2 downto 0);
13 wr_data: in std_logic_vector(15 downto 0); wr_enable: in std_logic);
14 end component;
15 signal rst, clk, wr_enable : std_logic;
16 signal rd_index1, rd_index2, wr_index : std_logic_vector(2 downto 0);
17 signal rd_data1, rd_data2, wr_data : std_logic_vector(15 downto 0);
18 begin
19 u0:register_file port map(rst, clk, rd_index1, rd_index2, rd_data1, rd_data2, wr_index, wr_data,
20 wr_enable);
21 process begin
22 clk <= ‘0'; wait for 10 us;
23 clk<=‘1'; wait for 10 us;
24 end process;
25 process  begin
26 rst <= '1'; rd_index1 <= "000"; rd_index2 <= "000"; wr_enable <= '0'; wr_index <= "000";
27 wr_data <= X"0000";
28 wait until (clk='0' and clk'event); wait until (clk='1' and clk'event); wait until (clk='1' and clk'event);
29 rst <= '0';
30 wait until (clk='1' and clk'event); wr_enable <= ‘1'; wr_data <= X"200a";
31 wait until (clk='1' and clk'event); wr_index <= “001"; wr_data <= X"0037";
32 wait until (clk='1' and clk'event); wr_index <= “010"; wr_data <= X"8b00";
33 wait until (clk='1' and clk'event); wr_index <= “101"; wr_data <= X"f00d";
34 wait until (clk='1' and clk'event); wr_index <= “110"; wr_data <= X"00fd";
35 wait until (clk='1' and clk'event); wr_index <= “111"; wr_data <= X"fd00";
36 wait until (clk='1' and clk'event); wr_enable <= '0';
37 wait until (clk='1' and clk'event); rd_index2 <= "001";
38 wait until (clk='1' and clk'event); rd_index1 <= "010";
39 wait until (clk='1' and clk'event); rd_index2 <= "101";
40 wait until (clk='1' and clk'event); rd_index1 <= "110";
41 rd_index2 <= "111"; wait;
42 end process;
43 end behavioural;
```

Figure 3. Test bench for register file in VHDL.

First, run ***Behavioural Simulation*** to check the functionality of your code. Then run ***Post-Route Simulation*** to assure that your code is synthesisable. In *Post-Route Simulation*, the outputs of register file do not change immediately when inputs vary. This is due to delay of FPGA components (configuration logic blocks and interconnects). You may need to change the test bench to reduce clock frequency in *Post-Route Simulation*.

ALU

Figure 6 depicts the ALU. The ALU receives two numbers (*in1* and *in2)* and generates *result* based on *alu_mode*. Table I shows the ALU operation for different *alu_mode* values.

Please refer to (**Table I.** A-format instructions) in the project description (Instruction Set) for details. You may need to extend the ALU modes and add new arithmetic operations to the ALU depending on your design. Write an HDL code (VHDL or Verilog) for the ALU and then write a test bench to verify your code. In your test bench, you should test the ALU for all possible values of *alu_mode*. Make sure that you run both behavioural and post-route simulations in ISE. A special TEST instruction determines whether a specific register's contents are zero, positive or negative. Please note that there is no functionality that stores or restores the contents of the zero (Z) or negative (N) flags explicitly.

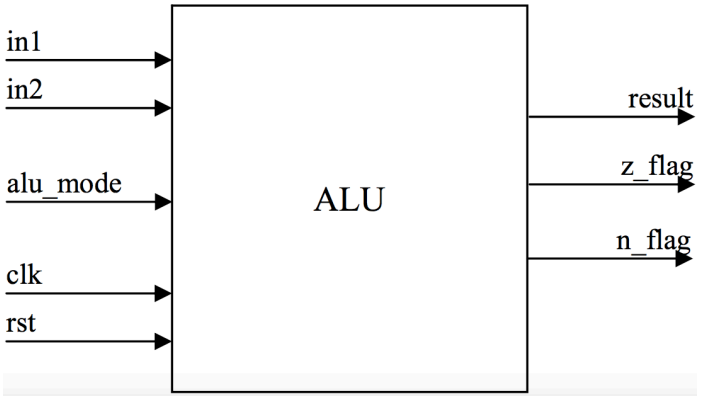

Figure 6. ALU.

Table I. ALU mode of operations

alu_mode	ALU operation
0	Nop
1	Add
2	SUB - Subtract
3	MUL- Multiply
4	NAND
5	SHL - Shift Left Logical
6	SHR - Shift Right Logical
7	TEST