CENG 450: Lab 2 Register File & ALU

Register File and ALU

In this lab, you write HDL code for a register file and an ALU. You may decide to design the two components different from those discussed in this lab manual. 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 four registers: R0, R1, R2, and R3. 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.

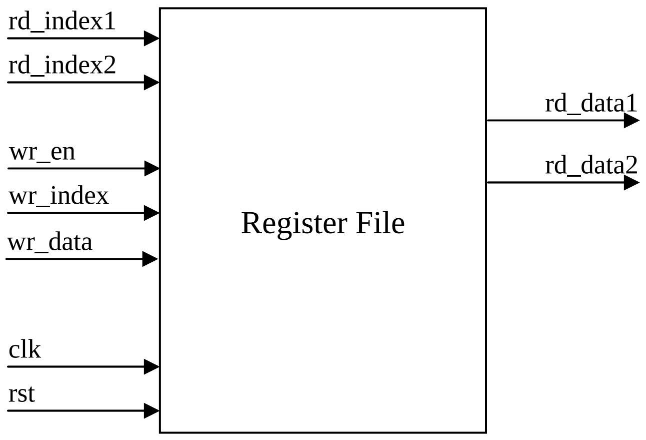

Figure 1. Register file.

Figure 2 shows the suggested Verilog code for the register file. Lines 26 to 41 describe the write operation. In Verilog, an always 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 45 to 47 describe read operation. In Verilog, an assignment statement models a combinatorial component. *rd_index1* determines which register should drive *rd_data1*.

You should complete always block for write operation (line 38) and assignment statement (line 50) for read operation.

```
module register_file (  rst, clk,

                        //read signals
                        rd_index1, rd_index2, rd_data1, rd_data2,

                        //write signals
                        wr_enable, wr_index, wr_data);


input           rst, clk;

//read signals
input   [1:0]   rd_index1;
input   [1:0]   rd_index2;
output  [7:0]   rd_data1;
output  [7:0]   rd_data2;

//write signals
input           wr_enable;
input   [1:0]   wr_index;
input   [7:0]   wr_data;

//internals signals
reg     [7:0]   reg_file[3:0];

//write operation
always @(negedge clk)
        if(rst)
                begin
                reg_file[0] = 8'h00;
                reg_file[1] = 8'h00;
                reg_file[2] = 8'h00;
                reg_file[3] = 8'h00;
                end
        else if(wr_enable)
                begin
                        case(wr_index)
                        2'b00: reg_file[0] = wr_data;
                        //fill this part
                        default:;
                        endcase
                end


//read operation
assign rd_data1 =       (rd_index1 == 2'b00) ? reg_file[0] :
                        (rd_index1 == 2'b01) ? reg_file[1] :
                        (rd_index1 == 2'b10) ? reg_file[2] : reg_file[3];


assign rd_data2 =       //fill this part

endmodule
```

Figure 2. Verilog code for register file.

Figure 3 shows the test bench for the register file. The test bench writes four values to registers R0~R3 and then read those registers. Figure 4 and 5 show register file and test bench in VHDL. Run Modelsim and simulate either Verilog or VHDL version of register file. 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*.

```
module test_register_file;

reg             rst, clk;

//read signals
reg     [1:0]   rd_index1;
reg     [1:0]   rd_index2;
wire    [7:0]   rd_data1;
wire    [7:0]   rd_data2;

//write signals
reg             wr_enable;
reg     [1:0]   wr_index;
reg     [7:0]   wr_data;

register_file u0 (rst, clk,

                        //read signals
                        rd_index1, rd_index2, rd_data1, rd_data2,

                        //write signals
                        wr_enable, wr_index, wr_data);

initial
begin
rst = 1;
clk = 0;
@(negedge clk);
@(negedge clk);
@(negedge clk);
@(negedge clk);
rst = 0;
end

always #50 clk = ~clk;

initial
begin
rd_index1 = 0;
rd_index2 = 0;

wr_enable = 0;
wr_index = 0;
wr_data = 0;
@(negedge rst);
```

```
@(posedge clk);
@(posedge clk);
@(posedge clk);

//writing data into register file
wr_enable = 1;
wr_index = 0;
wr_data = 8'h2a;
@(posedge clk);
wr_index = 1;
wr_data = 8'h37;
@(posedge clk);
wr_index = 2;
wr_data = 8'h8b;
@(posedge clk);
wr_index = 3;
wr_data = 8'hfd;

//stop writing data into register file
@(posedge clk);
wr_enable = 0;

@(posedge clk);
//reading data from register file
rd_index1 = 0;
rd_index2 = 1;
@(posedge clk);
rd_index1 = 2;
rd_index2 = 3;
end

endmodule
```

Figure 3. Test bench for register file in Verilog.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity register_file is
        port(rst : in std_logic;
           clk: in std_logic;

           --read signals
           rd_index1: in std_logic_vector(1 downto 0);
           rd_index2: in std_logic_vector(1 downto 0);
           rd_data1: out std_logic_vector(7 downto 0);
           rd_data2: out std_logic_vector(7 downto 0);

           --write signals
           wr_index: in std_logic_vector(1 downto 0);
           wr_data: in std_logic_vector(7 downto 0);
           wr_enable: in std_logic);
end register_file;
```

```
architecture behavioural of register_file is


type reg_array is array  (integer range 0 to 3) of std_logic_vector(7 downto 0);
--internals signals
signal reg_file : reg_array;

begin

--write operation
process(clk)
        begin
                if(clk='0' and clk'event) then
                        if(rst='1') then
                                for i in 0 to 3 loop
                                        reg_file(i)<= (others => '0');
                                end loop;
                        elsif(wr_enable='1')then
                                case wr_index(1 downto 0) is
                                        when "00" => reg_file(0) <= wr_data;
                                     --fill this part
                                     when others => NULL;
                                end case;
                        end if;
                end if;
        end process;

--read operation
rd_data1 <=     reg_file(0) when(rd_index1="00") else
                reg_file(1) when(rd_index1="01") else
                reg_file(2) when(rd_index1="10") else reg_file(3);


rd_data2 <=     --fill this part

end behavioural;
```

Figure 4. VHDL code for register file.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use  work.all;

entity test_alu is
end test_alu;

architecture behavioural of test_alu is

component register_file
        port(rst : in std_logic;
           clk: in std_logic;
           rd_index1: in std_logic_vector(1 downto 0);
```

```
		rd_index2: in std_logic_vector(1 downto 0);
		rd_data1: out std_logic_vector(7 downto 0);
		rd_data2: out std_logic_vector(7 downto 0);

		wr_index: in std_logic_vector(1 downto 0);
		wr_data: in std_logic_vector(7 downto 0);
		wr_enable: in std_logic);
end component;

signal rst : std_logic;
signal clk : std_logic;
signal rd_index1 : std_logic_vector(1 downto 0);
signal rd_index2 : std_logic_vector(1 downto 0);
signal rd_data1 : std_logic_vector(7 downto 0);
signal rd_data2 : std_logic_vector(7 downto 0);

signal wr_index : std_logic_vector(1 downto 0);
signal wr_data : std_logic_vector(7 downto 0);
signal wr_enable : std_logic;

begin

u0:register_file port map(rst, clk, rd_index1, rd_index2, rd_data1, rd_data2, wr_index, wr_data,
wr_enable);

process
begin
clk <= '0';
wait for 10 us;
clk<='1';
wait for 10 us;
end process;

process
begin
rst <= '1';
rd_index1 <= "00";
rd_index2 <= "00";
wr_enable <= '0';
wr_index <= "00";
wr_data <= "00000000";

wait until (clk='0' and clk'event);
wait until (clk='1' and clk'event);
wait until (clk='1' and clk'event);

rst <= '0';

wait until (clk='1' and clk'event);
wr_enable <= '1';
wr_data <= X"2a";

wait until (clk='1' and clk'event);
wr_index <= "01";
wr_data <= X"37";
```

```
wait until (clk='1' and clk'event);
wr_index <= "10";
wr_data <= X"8b";

wait until (clk='1' and clk'event);
wr_index <= "11";
wr_data <= X"fd";


wait until (clk='1' and clk'event);
wr_enable <= '0';


wait until (clk='1' and clk'event);
rd_index2 <= "01";

wait until (clk='1' and clk'event);
rd_index1 <= "10";
rd_index2 <= "11";
wait;
end process;

end behavioural;
```

Figure 5. Test bench for register file in VHDL.

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. ALU generates two flags: *z_flag* and *n_flag*. The *z_flag* is affected by all arithmetic instruction, but the *n_flag* changes only by *ADD*, *SUB*, and *AND* instructions. Please refer to table IV in the project description section for details. Note that *result* is asynchronous and the two flags are synchronous outputs. You may need to extend the ALU modes and add new arithmetic operations to the ALU depending on your design.
Write an HDL code (Verilog or VHDL) 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.

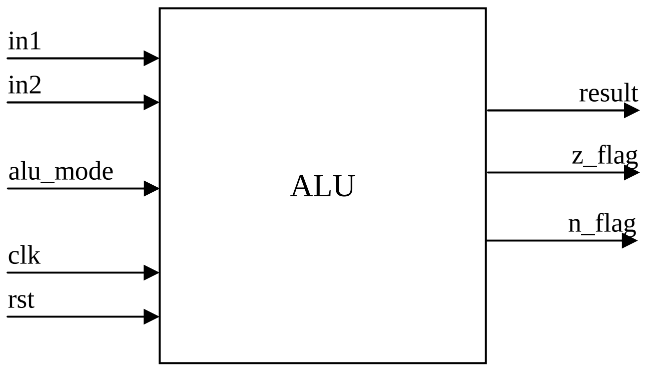

Figure 6. ALU.

Table I. ALU mode of operations

alu_mode	ALU operation
4	add
5	subtract
6	nand
7	shift left logical
0	shift right logical