CSCE 22104

Lab Report

Brent Marcus Orlina

ID: 011019116

Lab Section 001

Lab 7

Introduction

This lab's goal was to implement a register file and CPU component from a given template. The register file should be able to hold 16-bit data with a total of 16 registers, making the register addresses 4-bits in length. The CPU component should take in a 16-bit instruction, with the first four bits representing the opcode, the second four bits representing the register destination `rd`, and the third and fourth four bits representing the operand registers `rs` and `rt`. The ALU used in the CPU is from Lab 6, still supporting only four operations, meaning that the first two bits of the opcode is unused.

Approach

```
Entity RegFile is
    port(
        clk   : in std_logic;
        clear : in std_logic;

        write_addr : in std_logic_vector( 3 downto 0);
        write_data : in std_logic_vector(15 downto 0);
        load       : in std_logic;

        read_addr1 : in std_logic_vector(3 downto 0);
        read_addr2 : in std_logic_vector(3 downto 0);

        read_data1 : out std_logic_vector(15 downto 0);
        read_data2 : out std_logic_vector(15 downto 0)
    );
End RegFile;
```

Listing 1: The register file component's ports.

The register file was first implemented. Listing 1 shows the ports that the register file has. Since the CPU is synchronized, the register file needs to a clock, represented by the input port `clk`. The input port `clear` resets all the registers in the register file to 0, with the exception of register 1, which always holds 1. The input ports `write_addr`, `write_data`, `load` is for writing data into the register file. The input port `load` determines whether the data in `write_data` should be loaded into `write_addr`. Input ports `read_addr1` and `read_addr2` are the register addresses to be read, with the data being outputted to the output ports `read_data1` and `read_data2` respectively.

Listing 2 shows the implementation of the register file. The data is stored in a signal of an array of 16-bit vectors with a total of 16 registers. The process of writing to and clearing the register file is sensitive to the ports `clk`, `load`, and `clear`. The input port `clear` is low active asynchronous, meaning that the registers are resetted to zero independent of the clock if `clear` is inactive. The data in the port `write_data` is written to the register determined by the port `write_addr` if the clock is in a rising edge and if `load` is active as seen on line 12. On lines 15 and 16, it is ensured that registers 0 and 1 always hold the values 0 and 1 respectively, so that even if it is attempted to write to those registers, those registers will not be mutated. Finally, on lines registers 18 and 19, the output ports `read_data1` and `read_data2` are set to the values stored in the register addresses in `read_addr1` and `read_addr2` respectively. These are not in the process since they do not need to be synchronous to the clock nor do they depend on the `load` signal or `clear` signal.

```
architecture syn of RegisterFile is
    type ram_type is array (15 downto 0) of std_logic_vector(15 downto 0);
    signal REG_FILE : ram_type;
begin
    process(clk, load, clear)
    begin
        if (clear = '0') then
            REG_FILE(0)  <= x"0000";
            REG_FILE(1)  <= x"0001";
            ...
            REG_FILE(15) <= x"0000";
        elsif (clk'event and clk='1' and load='1') then
            REG_FILE(conv_integer(write_addr)) <= write_data;
        end if;
        REG_FILE(0) <= x"0000";
        REG_FILE(1) <= x"0001";
    end process;
    read_data1 <= REG_FILE(conv_integer(read_addr1));
    read_data2 <= REG_FILE(conv_integer(read_addr2));
end syn;
```

Listing 2: The register file component's architecture.

```
entity CPU is
    port(
        clk         : in std_logic;
        clear       : in std_logic;
        instruction : in std_logic_vector(15 downto 0)
     );
end CPU;
```

Listing 3: The CPU component's ports.

Listing 3 shows the ports for the CPU component. The input port `clk` is to synchronize the instructions that the CPU is fed. The input port `clear` is to reset the register file. Finally the input port `instruction` determines what the CPU does. The first four bits representing the opcode, the second four bits representing the register destination `rd`, and the third and fourth four bits representing the operand registers `rs` and `rt`.

```vhdl
architecture Behavioral of CPU is
    ...
    signal OP : std_logic_vector(3 downto 0);

    signal RS : std_logic_vector(3 downto 0);
    signal RT : std_logic_vector(3 downto 0);
    signal RD : std_logic_vector(3 downto 0);
    signal RSData : std_logic_vector(15 downto 0);
    signal RTData : std_logic_vector(15 downto 0);

    signal cout     : std_logic;
    signal Sout_ALU : std_logic_vector(15 downto 0);
begin
    ...
end Behavioral;

```

Listing 4: The signals that the CPU implementation uses.

Listing 4 shows the signals used in the implementation. The signals `OP`, `RS`, `RT`, and `RD` is used to separate the instruction to their representative sections. The signals `RSData` and `RTData` are used to connect the outputs of the reads from the register to the ALU. The signals `cout` and `Sout_ALU` are used for the outputs of the ALU, where `cout` is the final carry out and `Sout_ALU` is the result of the ALU, eventually writing back to the register file.

```vhdl
architecture Behavioral of CPU is
    ...
begin
    -- Instruction Fetch
    OP <= instruction(15 downto 12);
    RD <= instruction(11 downto 8 );
    RS <= instruction( 7 downto 4 );
    RT <= instruction( 3 downto 0 );

    -- Instruction Decode
    ...

    -- Execute
    ...
end Behavioral;
```

Listing 5: The instruction fetch phase of the CPU implementation.

Listing 5 shows the instruction fetch phase of the CPU implementation. This phase simply splits the instruction into its respective parts.

```vhdl
architecture Behavioral of CPU is
    ...
begin
    -- Instruction Fetch
    ...

    -- Instruction Decode
    CPU_RegisteRS_0: RegFile
    port map(
        clk   => clk,
        clear => clear,

        write_addr => RD,
        write_data => Sout_ALU,
        load       => '1',

        read_addr1 => RS,
        read_addr2 => RT,

        read_data1 => RSData,
        read_data2 => RTData
    );

    -- Execute
    ...

end Behavioral;
```

Listing 6: The instruction decode phase of the CPU implementation.

Listing 6 shows the instruction decode phase of the CPU implementation. This connects the respective `clk` and `clear` signals to the register file as well as the register signals to the register file. The register signals, `RS` and `RT` is as the read addresses of register file, with the register contents going to the signals `RSData` and `RTData`. The `RD` signal is used as the write address with the output of the ALU, the `Sout_ALU` signal, being written to that address. The input port `load` of the register file is always active since all of the instructions supported by the CPU implementation currently always writes back to the register file. Since this section also shows writing to the register, it can also be considered to be the write-back phase of the CPU implementation.

```vhdl
architecture Behavioral of CPU is
    ...
begin
    -- Instruction Fetch
    ...

    -- Instruction Decode
    ...

    -- Execute
    CPU_ALU_0: ALU16Bit
    port map(
        a => RSData,
        b => RTData,
        s => OP(1 downto 0),
        sout => Sout_ALU,
        cout => cout
    );
end Behavioral;
```

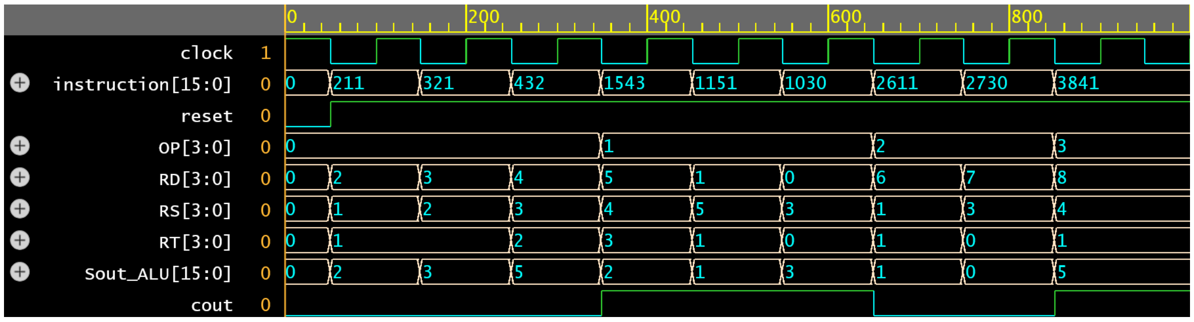
Listing 7: The instruction execution phase of the CPU implementation.

Listing 7 shows the execution phase. It connects the data read from the register file, in the signals `RSData` and `RTData`, to the 16-bit ALU implemented from a previous lab. It also notable connects the only the two lower bits of the opcode. This is because the CPU still only supports four instructions, thus the ALU only needs two bits with the two higher bits of the opcode being unused. The two higher bits will be used to implement future instructions. Finally, the result of the ALU is connected to `Sout_ALU`, which is connected back to the register file for writing, and the carry out of the ALU is connected to `cout` so that it can be observed.

Experimentation

The components were tested by writing a testbench for the CPU component. The testbench written tested for given instructions for the CPU and were manually verified to be correct.

Results & Discussion



0 200 400 600 800
clock 1
instruction[15:0] 0 | 0 211 321 432 1543 1151 1030 2611 2730 3841
reset 0
OP[3:0] 0 | 0 1 2 3
RD[3:0] 0 | 0 2 3 4 5 1 0 6 7 8
RS[3:0] 0 | 0 1 2 3 4 5 3 1 3 4
RT[3:0] 0 | 0 1 2 3 1 0 1 0 1
Sout_ALU[15:0] 0 | 0 2 3 5 2 1 3 1 0 5
cout 0

Figure 1: The waveform for the CPU component.

Instruction	op	rd	rs	rt	value (of rd)
`ADD R2, R1, R1`	0x0	0x2	0x2	0x1	2
`ADD R3, R2, R1`	0x0	0x3	0x2	0x1	3
`ADD R4, R3, R2`	0x0	0x4	0x3	0x2	5
`SUB R5, R4, R3`	0x1	0x5	0x4	0x3	2
`SUB R1, R5, R1`	0x1	0x1	0x5	0x1	1
`SUB R0, R3, R0`	0x1	0x0	0x3	0x0	0
`AND R6, R1, R1`	0x2	0x6	0x1	0x1	1
`AND R7, R3, R0`	0x2	0x7	0x3	0x0	0
`OR  R8, R4, R1`	0x3	0x8	0x4	0x1	5

Table 1: Results of the CPU component testbench in table form.

The CPU component works as expected. Figure 1 shows the waveform of the CPU component, correctly outputting the results of each operation. The waveform is in hex radix to easily show the instruction in the current clock cycle and the fact that the ALU does not have any negative results. Table 1 shows the waveform results in table form. It is also noteworthy that during the sixth instruction, instruction `0x1030`, that the "value (of rd)", with `rd` as register 0, is filled out as a `0` on table but displays as `3` on the waveform. This is because the waveform uses the output signal `Sout_ALU` of the CPU to display what will be written in the register specified by `RD`. However, in the register file implementation, it is known that no matter what is written to register 0, it will always be reverted back to 0. Similarly on the previous instruction, register 1 is being written to which would be reverted back to 1. However, it is by coincidence that the result of the ALU is also a 1, following that the "value of rd" in the waveform shown in figure 1 matches "value of rd" in table 1.

Conclusions

Both the CPU and Register File component worked correctly and the CPU component displayed the expected behavior during testing. the knowledge learned from this lab was leanring how to construct a Register File component and a CPU component.