CSCE 22104

Lab Report

Brent Marcus Orlina

ID: 011019116

Lab Section 001

Lab 11

Introduction

The goal of this lab was to integrate instruction memory, which is a memory component that holds the instructions that the CPU should execute. The previously provided memory component only held 16 bytes, or eight 16-bit instructions. However, the CPU supports fifteen instructions so the memory component was updated to hold up to 32 bytes, or sixteen 16-bit instructions. The instruction memory should also read from a text file called `instr.txt` instead of `in.txt`, which the previously integrated memory, the data memory, read in from.

Another component that is provided is the program counter register, `PC_REG`. The purpose of this component is to hold the address that the CPU should read from in the instruction memory. After each clock cycle, the program counter should increment by two to read the next instruction since each instruction is two bytes long. The program counter should also reset to 0 if the CPU has its input port `clear` as 0, since it is active low.

Approach

```
entity Memory is
   generic (
       INPUT : string := "instr.txt"; -- changed!
       OUTPUT : string := "out.txt"
   );
    port ( ... );
end entity Memory;
```

Listing 1: The memory component's new generic.

```
architecture beh of Memory is
    type mem_type is array (0  to  31) of std_logic_vector(7  downto 0); -- changed!
    ...
    signal mem : mem_type := init_mem;
begin
    ...
end architecture beh;
```

Listing 2: The memory component's new signal.

Listing 1 shows the generics that the memory component now uses. The `INPUT` generic of the memory component, the data that the component is intialized to, is now `instr.txt` by default. This means that even the data memory reads in `instr.txt`, filling the it with "random" junk data. This is done because there was trouble with changing the `INPUT` generic from the default by using

a generic map, thus the default had to be changed and its effect of filling the data memory with junk data has minimal impact.

Listing 2 shows the type `mem_type` changed from an array of 16 bytes to an array of 32 bytes. This is so that all of the instructions supported by the CPU can be tested since there are 15 total instructions with each instruction having a length of two bytes, thus all 15 instructions can fit into memory.

```
entity PC_REG is
    port(
        clk : in std_logic;
        reset : in std_logic;
        Input : in std_logic_vector(15 downto 0);
        Output : out std_logic_vector(15 downto 0)
    );
end PC_REG;
```

Listing 3: The program counter register's ports.

Listing 3 shows the ports of the provided program counter register component. Since the program counter increments after each clock cycle, it is synchronous and thus needs a `clk` input port. It also resets back to 0 if the CPU clears so it also takes an input port `reset`. The ports `Input` and `Output` are for the program counter, which feeds back to itself.

```
architecture Behavioral of PC_REG is
    signal Reg : std_logic_vector(15 downto 0) := x"0000";
    signal Delay : std_logic;
begin
    process(reset, clk)
    begin
        if (reset = '0') then
            Reg <= x"0000";
            Delay  <= '1';
        elsif (clk'event and clk='1') then
            if (Delay = '1') then
                Reg <= x"0000";
                Delay <= '0';
            else
                Reg <= Input + 2;
            end if;
        end if;
    end process;
    Output <= Reg;
end Behavioral;
```

Listing 4: The program counter register's implementation.

Listing 4 shows the implementation of the provided program counter register. It uses two signals `Reg` and `Delay`. The signal `Reg` is simply connected to the output port `Output`. A signal for this is needed since VHDL does not allow an output port to be set using itself.

Since the register is synchronous to the clock, the main process is sensitive to the input port `clk`. However, the reset is asynchronous, so the main process is also sensitive to the input port `reset`. Firstly, it checks if the reset signal is inactive to reset the register back to 0 since the CPU uses active low for its reset signal. If there is a signal to reset, it resets the register to 0 and activates the `Delay` signal.

If there is no signal to reset, it checks whether `clk` is in a rising edge since it is synchronous to the clock. During a rising edge, it checks whether there is a delay signal, through the signal `Delay`. This is so that in the clock cycle right after a reset signal is made, the register will output a 0. If there is no delay signal, the register would immediately increment the register in the following clock cycle of the delay signal, and thus the program counter wouldn't really start at 0. This can be seen when the `Delay` signal is `'1'`. It sets `Reg` to 0 and deactivates the delay signal so that the program counter can be incremented in the next clock cycle.

```
1  entity CPU is
2      port(
3          clk         : in std_logic;
4          clear       : in std_logic
5          -- instruction : in std_logic_vector(15 downto 0) -- removed!
6       );
7  end CPU;
```

Listing 5: The CPU component's ports.

Listing 5 shows the CPU-components ports. Since the instructions are now being fetched from memory, there is no reason for the CPU to have an input port that receives instructions and thus the `instruction` input port is removed.

```
architecture Behavioral of CPU is
    -- Components
    ...

    -- new!
    component PC_REG
    port(
        clk : in std_logic;
        reset : in std_logic;
        Input : in std_logic_vector(15 downto 0);
        Output : out std_logic_vector(15 downto 0)
    );
    end component;

    ...

    -- Signals
    signal PC : std_logic_vector(15 downto 0);          -- new!
    signal Instruction : std_logic_vector(15 downto 0); -- new!

    signal OP : std_logic_vector(3 downto 0);
    ...
begin
    ...
end Behavioral;
```

Listing 6: The other components and signals that the CPU component uses.

Listing 6 shows the components and signals that the CPU uses. The CPU of course uses the new program counter register component `PC_REG`. The new signal `PC` is used as the program counter, which is the address that the instruction memory reads from to output the current instruction to the new signal `Instruction`.

```
1  architecture Behavioral of CPU is
2      -- Components + Signals
3      ...
4  begin
5      -- Instruction Fetch
6      PCRegister : PC_REG              -- new!
7      port map (
8          clk    => clk,
9          reset  => clear,
10         Input  => PC,
11         Output => PC
12     );
13     InstructionMemory : Memory       -- new!
14     port map(
15         clk      => clk,
16         read_en  => '1',
17         write_en => '0',
18         addr => PC,
19         data_in  => x"0000",
20         data_out => Instruction,
21         mem_dump => '0'
22     );
23     OP <= Instruction(15 downto 12); -- new!
24     RD <= Instruction(11 downto  8); -- new!
25     RS <= Instruction( 7 downto  4); -- new!
26     RT <= Instruction( 3 downto  0); -- new!
27
28     -- Instruction Decode + Execute + Memory + Writeback
29     ...
30 end Behavioral;
```

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

Listing 7 shows the instruction fetch phase of the CPU. A `PC_REG` component is initialized. Notably, the signal `PC` is connected to both the `Input` and `Output` ports of the PC register component. This is because the `PC` signal is being incremented within the `PC_REG` component after a clock cycle and the result should be outputted to the `PC` signal again.

The `InstructionMemory` component is initialized. Notably, the ports `read_en` and `write_en` are constantly active and inactive respectively since the CPU instructions should only be read and never be modified. Of course this is not how it works in the real world. However in this simulation, there is no reason for the instructions to change. The port `addr` is mapped to `PC` since the CPU should read the next instruction in memory after every clock cycle. Since this memory component will never be written to, the `data_in` port is set to a constant of 0. The data read is outputted to the signal `Instruction`, which is split to its respective sections `OP`, `RD`, `RS`, and `RT` on lines 23 to 25.

Experimentation

The integration of the new instruction memory component was tested by writing a testbench for the CPU component that simply runs the clock for however long is needed to test all the given instructions. Along with the testbench, the `instr.txt` file is given which contains all of the given instructions to be tested, used to intialize the instruction and data memories.

Results & Discussion

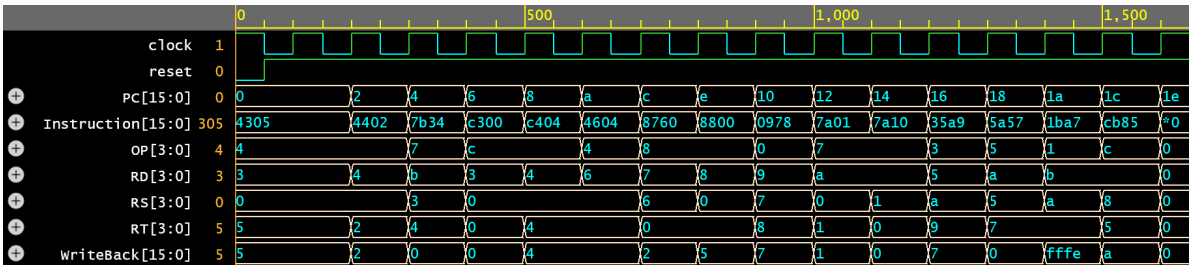

Figure 1: The waveform for the CPU component with a radix of hex.

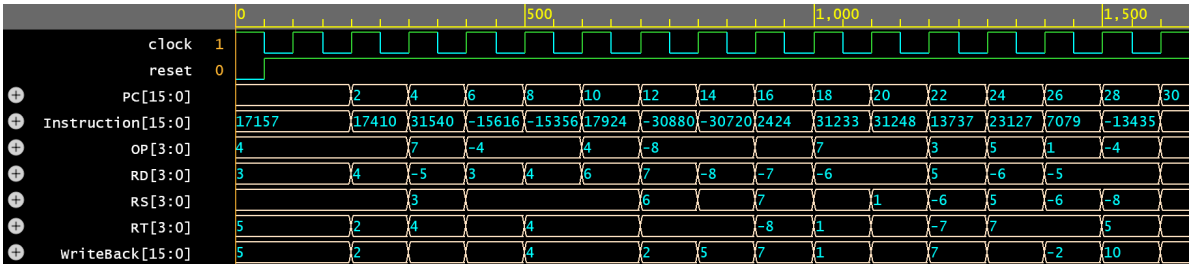

Figure 2: The waveform for the CPU component with a radix of signed decimal.

The CPU component works with the integration of the instruction memory works as expected. Figure 1 shows the waveform of the CPU component in hex radix to easily show the instruction in the current clock cycle. Figure 2 shows the waveform of the CPU component in signed decimal radix to easily show the values of the `WriteBack` signal since it contains a negative number. Signals that are blank are zeroes. Table 1 shows the waveform results in table form.

Instruction	op	rd	rs	rt	value (of rd)
ADDI R3, R0, 5	0x4	0x3	0x0	0x5	5
ADDI R4, R0, 2	0x4	0x4	0x0	0x2	2
SLT R11, R3, R4	0x7	0xB	0x3	0x4	0
SW R3, 0(R0)	0xC	0x3	0x0	0x0	0
SW R4, 4(R0)	0xC	0x4	0x0	0x4	4
ADDI R6, R0, 4	0x4	0x6	0x0	0x4	4
LW R7, 0(R6)	0x8	0x7	0x6	0x0	2
LW R8, 0(R0)	0x8	0x8	0x0	0x0	5
ADD R9, R7, R8	0x0	0x9	0x7	0x8	7
SLT R10, R0, R1	0x7	0xA	0x0	0x1	1
SLT R10, R1, R0	0x7	0xA	0x1	0x0	0
OR R5, R10, R9	0x3	0x5	0xA	0x9	7
SUBI R10, R5, 7	0x5	0xA	0x5	0x7	0
SUB R11, R10, R7	0x1	0xB	0xA	0x7	-2
SW R11, 5(R8)	0xC	0xB	0x8	0x5	10

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

Conclusions

The integration of the instruction memory was implemented into the CPU correctly, shown through the testbench and waveforms for the CPU. The knowledge learned from this lab was learning how to integrate instruction memory into the CPU by using a program counter and a register to store the current program counter value.