CSCE 22104

Lab Report

Brent Marcus Orlina

ID: 011019116

Lab Section 001

Lab 9

Introduction

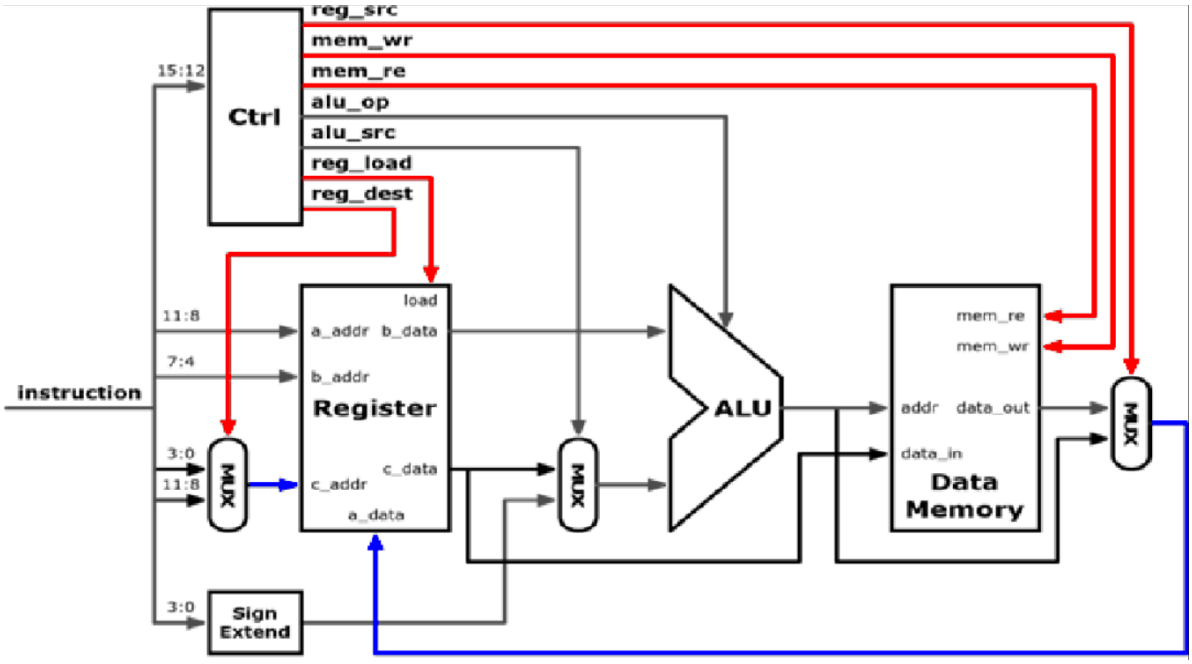

Figure 1: The circuit diagram of the CPU component to be implemented.

This lab's goal was to integrate a given memory component to the CPU previously built in lab 8 to support two additional instructions, load-word and store-word, or `lw` and `sw` for short respectively, with opcodes `1000` and `1100`. The CPU along with any other component within the CPU should follow the circuit diagram as shown in figure 1.

The two new instructions are an I-type instruction, meaning that it uses the last four bits as an immediate value. The load-word instruction, `lw`, reads an address in memory, and writes the content to the RD register provided by the instruction. The store-word instruction, `sw`, reads the content of the RD register and stores it to an address in memory. The address read in memory is calculated by the sum of the immediate value and the content of the RS register provided by the instruction.

Approach

```
entity Memory is
    ...
    port (
        clk : in std_logic;
        read_en : in std_logic;
        write_en : in std_logic;
        addr : in std_logic_vector(15 downto 0);
        data_in : in std_logic_vector(15 downto 0);
        data_out : out std_logic_vector(15 downto 0);
        mem_dump : in std_logic := '0'
    );
end entity Memory;
```

Listing 1: The memory component's ports.

Listing 1 shows the memory component's ports. Since the memory is synchronous to the clock, it has an input port `clk`. The input ports `write_en` and `read_en` allows for the memory block to written to and read, respectively. The memory should not be written to and read simulatenously, which is ensured by the control block component. The input port `addr` is the address to be written to, by the data in input port `data_in`, or read from, outputted to the output port `data_out`. Finally, the input port `mem_dump` is used for debugging purposes, and is set inactive by default.

Since the memory component was given, the implementation details won't be shown.

```
entity Control is
    port(
        op : in std_logic_vector(3 downto 0);

        ctrl_alu_op  : out std_logic_vector(1 downto 0);
        ctrl_alu_src : out std_logic;

        ctrl_reg_src   : out std_logic; -- new!
        ctrl_reg_dst   : out std_logic; -- new!
        ctrl_reg_write : out std_logic; -- new!

        ctrl_mem_read  : out std_logic; -- new!
        ctrl_mem_write : out std_logic  -- new!
     );
end Control;
```

Listing 2: The control block component's ports.

Listing 2 shows the new output ports added to the control block component. The output port `ctrl_reg_src` controls whether the output of the ALU or the output of the memory read should

be written back to the register file. The output port `ctrl_reg_dst` controls whether the RT or RD register should be read for the second register read. The reason this is done is because the store-word instruction `sw` uses the RD register section of the instruction to determine what to store into the calculated memory address and thus must be read. This is valid since the RT section of the instruction is used as an immediate value. All other instructions only write to the RD register and does not require it to be read.

The output port `ctrl_reg_write` determines whether the register file should be written to. Previously, the register file was always written to. However, the new `sw` instruction does not write anything to the register file, only writing into the memory component. Similarly, output ports `ctrl_mem_read` and `ctrl_mem_write` control whether the memory component should be read from or be written to since not all instructions read from or write to the memory. For example, the `lw` instruction does not write to the memory, and thus must be ensured that `ctrl_mem_write` is inactive. However, the instruction does need to read from memory, and so it must be ensured that `ctrl_mem_read` is active.

```
architecture Datapath of Control is
begin
    ctrl_alu_op <= op(1 downto 0);
    ctrl_alu_src <= '1' when op "1000" else op(2);   -- new!

    ctrl_reg_src   <= '0' when op = "1000" else '1'; -- new!
    ctrl_reg_dst   <= '1' when op = "1100" else '0'; -- new!
    ctrl_reg_write <= '0' when op = "1100" else '1'; -- new!

    ctrl_mem_read  <= '1' when op = "1000" else '0'; -- new!
    ctrl_mem_write <= '1' when op = "1100" else '0'; -- new!
end Datapath;
```

Listing 3: The control block component's implementation.

Listing 3 shows the implementation for the new control block component. The output ports `ctrl_alu_op` remains the same since the opcodes for the new instructions, `lw` and `sw` since their opcodes `1000` and `1100` have a `00` in the last two bits, which correctly uses the ALU opcode for addition. However, `ctrl_alu_src` was changed to account for the fact that in the old implementation, `ctrl_alu_src` was simply determined by `op(2)`, the second bit of the opcode. This is incompatible with `lw`'s opcode since the opcode's second bit, index-0, is inactive, implying that the last four bits of the instruction is a register address instead of an immediate value. Therefore, a

special case has to be made for the `lw` instruction to use an immediate value.

The signal `ctrl_reg_src` is inactive for the `lw` instruction since the data that should be written to the register comes from the memory component, as shown in figure 1. Otherwise, all other instructions use the ALU's output to write to the register file, or that it does not write to the register file at all. The signal `ctrl_reg_dst` is active for the `sw` instruction since it reads from the RD register given in the instruction to provide the data to store in the memory, as shown in figure 1. Since instruction `sw` is the only instruction that does not write to the register file, `ctrl_reg_write` is inactive for `sw` and active for all other instructions.

The signal `ctrl_mem_read` is only active for instruction `lw` instruction since `lw` is the only instruction that should read from the memory. Similarly, signal `ctrl_mem_write` is only active for `sw` since it is the only instruction that should write to the memory. This implementation ensures that the CPU cannot read and write to the memory simulatenously.

```
architecture Behavioral of CPU is
    -- Components
    ...
    -- Signals
    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 RegisterSource      : std_logic; -- new!
    signal RegisterDestination : std_logic; -- new!
    signal RegisterWrite       : std_logic; -- new!
    signal RSData              : std_logic_vector(15 downto 0);
    signal Register2Address    : std_logic_vector(3  downto 0); -- new!
    signal Register2Data       : std_logic_vector(15 downto 0); -- renamed!

    signal ALUSource : std_logic;
    signal ALUOP     : std_logic_vector(1 downto 0);
    signal ALUInput  : std_logic_vector(15 downto 0);
    signal ALUOutput : std_logic_vector(15 downto 0); -- renamed!
    signal cout      : std_logic;

    signal MemoryRead   : std_logic;                     -- new!
    signal MemoryWrite  : std_logic;                     -- new!
    signal MemoryOutput : std_logic_vector(15 downto 0); -- new!

    signal Immediate : std_logic_vector(15 downto 0);
    signal WriteBack : std_logic_vector(15 downto 0);    -- new!
begin
```

Listing 4: The new signals used in the CPU implementation.

Listing 4 show the new signals used in the CPU implementation to support the two new instructions. Signals `RegisterSource`, `RegisterDestination`, `RegisterWrite`, `MemoryRead`, and `MemoryWrite` are signals simply connected to the output ports of the control block component. Since it is not especially important, this is shown in listing 7 in the appendix. Signal `Register2Address` is the address of the second register chosen to be read between RD or RT, determined by the signal `RegisterDestination`. The data is outputted to the signal `Register2Data` which was formerly named `RTData` since the second register that was read was always register RT.

The signal `MemoryOutput` is the output of the memory component when it is read from. The signal `WriteBack` is the data to be sent back to the register file to either be written or ignored, chosen between `ALUOutput` (formerly named `ALUSout`) and `MemoryOutput`, and determined by `RegisterSource`.

```
architecture Behavioral of CPU is
    -- Components + Signals
    ...
begin
    -- Instruction Fetch
    ...
    -- Instruction Decode
    ...
    Register2Address <= RD when RegisterDestination = '1' else RT;
    CPU_Registers_0: RegFile
    port map(
        clk   => clk,
        clear => clear,

        a_addr => RD,
        a_data => WriteBack,
        load   => RegisterWrite,

        b_addr => RS,
        c_addr => Register2Address,

        b_data => RSData,
        c_data => Register2Data
    );
    ALUInput <= Immediate when ALUSource = '1' else Register2Data;
    -- Execute + Memory
    ...
    -- WriteBack
    WriteBack <= ALUOutput when RegisterSource = '1' else MemoryOutput;
end Behavioral;

```

Listing 5: The decode and writeback phase of the CPU implementation.

Listing 5 shows the decode adn writeback phase of the CPU implementation. The signal `Register2Address` is set to either be register RD or RT, determined by the `RegisterDestination` signal and is connected to the input port `c_addr`. The signal `RegisterWrite` is now connected to the input port `load` of the register file, where it previously was set to be constantly active. The signal `Regsiter2Data` is connected to the output port `c_data`. Then, the signal `ALUInput` is set to either be an immediate or the content of the register chosen by `Register2Address`, determined by the signal `ALUSource`. Finally, the signal `WriteBack` is the data to be written back to the register file, either the ALU result in the signal `ALUOutput` or the data read in the memory in the signal `MemoryOutput`, determined by the `RegisterSource` signal. The signal `ALUOutput` is calculated in the execute phase of the CPU, which is unchanged. Finally, the signal `MemoryOutput` is determined in the memory phase of the CPU as shown in listing 6.

```
architecture Behavioral of CPU is
    -- Components + Signals
    ...
begin
    -- Instruction Fetch + Instruction Decode + Execute
    ...
    -- Memory
    MemoryBlock : Memory
    port map(
        clk      => clk,
        read_en  => MemoryRead,
        write_en => MemoryWrite,

        addr => ALUOutput,
        data_in  => Register2Data,
        data_out => MemoryOutput,

        mem_dump => '0'
    );
    -- WriteBack
    ...
end Behavioral;

```

Listing 6: The memory phase of the CPU implementation.

Listing 6 shows the memory phase of the CPU implementation. Its implementation is straightforward, simply connecting the clock and the signals from the control block component, `MemoryRead` and `MemoryWrite`. The address, in the input port `addr`, comes from the output of the ALU, coming from the signal `ALUOutput`. The data that is written to the memory for the `sw` instruction comes

from the signal `Register2Data`, which is connected to the output port `c_data` of the register file. The data read at the given address is outputted to the signal `MemoryOutput`, which is used in the `WriteBack` signal as shown in 5. The debug input port `mem_dump` is set to be constantly inactive. With the memory component implemented, the CPU is completed, supporting the two new instructions, `lw` and `sw`.

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

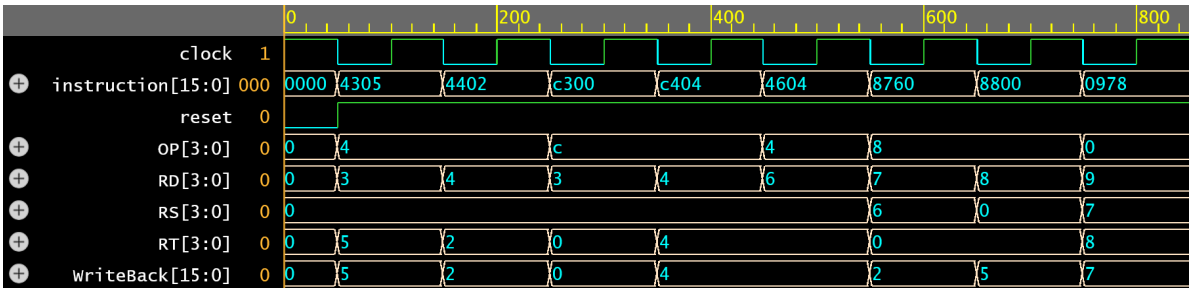

Figure 2: The waveform for the CPU component.

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
`SW R3, 0(R0)`	0xC	0x3	0x0	0x0	0
`SW R4, 4(R0)`	0xC	0x4	0x0	0x4	4
`ADDI R6, R0, 4`	0x6	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

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

The CPU component works as expected. Figure 2 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.

Conclusions

The memory component was integrated into the CPU correctly, shown through the testbench for the CPU. The knowledge learned from this lab was learning how to integrate a memory component into the CPU, along with supporting two new instructions using the memory component, correctly. This was done by identifying what new signals are needed to allow other components to communicate with the memory component, as well as ensuring that the right data goes to the right components under the right instructions.

Appendix

```
architecture Behavioral of CPU is
    -- Components + Signals
    ...
begin
    ...
    -- Instruction Decode
    ControlBlock : Control
    port map(
        op => OP,

        ctrl_alu_op  => ALUOP,
        ctrl_alu_src => ALUSource,

        ctrl_reg_src   => RegisterSource,
        ctrl_reg_dst   => RegisterDestination,
        ctrl_reg_write => RegisterWrite,

        ctrl_mem_read  => MemoryRead,
        ctrl_mem_write => MemoryWrite
    );
    ...
end Behavioral;
```

Listing 7: The CPU signals connected to the control block component.