CSCE 22104

Lab Report

Brent Marcus Orlina

ID: 011019116

Lab Section 001

Lab 8

Introduction

This lab's goal was to implement a control block and sign extension components to support two additional instructions `addi` and `subi`, with opcodes `0100` and `0101` respectively. These two new instructions uses the last four bits of the instruction as an immediate value instead of an address for a register.

The control block component should take in the first four bits of the instruction, representing the opcode of the instruction and output the two-bit opcode used in the ALU, and which source should be used in the second operand of the ALU, the data in register RT or the immediate value. The sign extension component should take in the last four bits of the instruction, representing as the immediate value, and extend it to 16 bits, taking into account its sign. The two components should then be able to be combined with the CPU previously created from Lab 7.

Approach

```
1  entity Control is
2      port(
3          op           : in std_logic_vector(3 downto 0);
4          ctrl_alu_op  : out std_logic_vector(1 downto 0);
5          ctrl_alu_src : out std_logic
6       );
7  end Control;
8
9  architecture Datapath of Control
10 begin
11     ctrl_alu_op <= op(1 downto 0);
12     ctrl_alu_src <= op(2);
13 end Datapath;
```

Listing 1: The control block component's implementation.

Listing 1 shows the control block component's implementation. It takes in the four-bit opcode through the input port `op` and outputs the ALU opcode and ALU source through the output ports `ctrl_alu_op` and `ctrl_alu_src` respectively.

In the architecture, the ALU opcode is simply the last two bits of the instruction opcode and the ALU source is the third bit of the instruction opcode. This works because the opcode for the immediate and non-immediate instructions of addition and subtraction only differ in the third bit. As a reminder, the opcodes for `add` and `sub` are `0000` and `0001` and the opcodes for `addi` and `subi`

are 0100 and 0101. This allows the control block to use the third bit to determine whether the content in register RT or the immediate value should be used as the source of data for the ALU. The ALU opcode remains the same whether the content of register RT or the immediate value.

```
1  entity ImmExt is
2      port(
3          imm : in std_logic_vector(3 downto 0);
4          ext : out std_logic_vector(15 downto 0)
5       );
6  end ImmExt;
7
8  architecture Datapath of ImmExt
9      signal extension : std_logic_vector(11 downto 0);
10 begin
11     extension <= x"111" when imm(3) = '1' else x"000";
12     ext <= extension & imm;
13 end Datapath;
```

Listing 2: The sign extension component's implementation.

Listing 2 shows the sign extension component's implementation. It takes in the four-bit immediate value through the input port imm and outputs the sign-extended through the output port ext.

In the architecture, it uses a 12-bit signal, extension, as a placeholder for what value should be concatenated to imm to extend it to 16 bits. For positive numbers, zeroes can simply be concatenated as the most significant bit to extend the number. For example the 4-bit positive number in two's complement, 0011, can be extended to 16 bits by simply concatenating twelve zeroes to the most significant bit resulting in 0000 0000 0000 0011 which keeps the value.

For negative numbers, a series of ones can be concatenated to the most significant bit to extend the number. For example, the 4-bit negative number in two's complement, 1001, -7 in decimal form, can be extended to 16 bits by simply concatenating twelve ones to the most significant bit, resulting in 1111 1111 1111 1001 which is still -7 in decimal form. In two's complement, if the most significant bit is a 1 then the number is a negative. This allows the sign-extension component to simply check whether the most significant bit is a 1 or a 0 to see if it must concatenate twelve ones or twelve zeros, as seen on line 11 of listing 2. On line 12, ext is set as the concatenation of the determined extension and the input immediate value, done through the & operator in VHDL.

```
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 RSData : std_logic_vector(15 downto 0);
    signal RTData : std_logic_vector(15 downto 0);

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

    signal Immediate : std_logic_vector(15 downto 0); -- new!
begin
    ...
end Behavioral;
```

Listing 3: The signals used in the CPU implementation.

Listing 3 shows the new signals used for the CPU implementation. The `ALUOP` signal is used to determine the operation that the ALU executes, addition, subtraction, and-operation, or the or-operation. The signal `ALUSource` is the control signal that determines which data should go into the second operand of the ALU. It is decided between the data from the `RT` register, in the `RTData` signal, or the sign-extended immediate value in the `Immediate` signal. The ALUInput is the signal used to put the result of the decision made by the `ALUSource`, connecting to the second operand of the ALU. Finally, the `Immediate` signal holds the value of the sign-extended immediate value of the instruction.

```
architecture Behavioral of CPU is
    -- Components + Signals
    ...
begin
    -- Instruction Fetch
    ...

    -- Instruction Decode

    -- new!
    ControlBlock : Control
    port map(
        op => OP,
        ctrl_alu_op => ALUOP,
        ctrl_alu_src => ALUSource
    );

    -- new!
    ImmediateExtension : ImmExt
    port map (
        imm => RT,
        ext => Immediate
    );

    CPU_Registers_0: RegFile port map( ... );

    ALUInput <= Immediate when ALUSource = '1' else RTData; -- new!

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

Listing 4: The instruction decode phase of the CPU implementation.

Listing 4 shows the instruction decode phase of the CPU implementation. This connects the `OP` signal to the control block, which outputs the results to the `ALUOP` and `ALUSource` signals.

This phase also connects the signal `RT` to the immediate extension component, outputting the sign-extended immediate value. Normally, the signal `RT`, which is the last four bits of the instruction, represents the address of the `RT` register. However, since the immediate value is also in the last four bits of the instruction for the `addi` and `subi` instructions, it also represents the immediate value in these instructions. Since the operands of the ALU are 16-bits long, the input immediate value needs to be sign extended.

Finally, the signal `ALUInput` is determined whether it uses the data in the `Immediate` signal or the `RTData` signal. Since the immediate instructions `addi` and `subi` both have a `1` in the third bit of their opcodes, which is what the `ctrl_alu_src` looks at in the control block, the ALU should

use the immediate value when `ALUSource` is active and `RTData` when it is inactive.

```
architecture Behavioral of CPU is
    -- Component + Signals
    ...
begin
    -- Instruction Fetch
    ...

    -- Instruction Decode
    ...

    -- Execute
    CPU_ALU_0: ALU16Bit
    port map(
        A => RSData,
        B => ALUInput, -- new!
        S => ALUOP,    -- new!
        Sout => ALUSout,
        Cout => cout
    );
end Behavioral;
```

Listing 5: The execution phase of the CPU implementation.

Finally, 5 shows the execution phase of the CPU implementation. The new `ALUInput` and `ALUOP` signals are connected to the ALU's `B` and `S` ports. The `ALUInput` signal is connected to account for the use of the sign-extended immediate value for the `addi` and `subi` instructions.

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

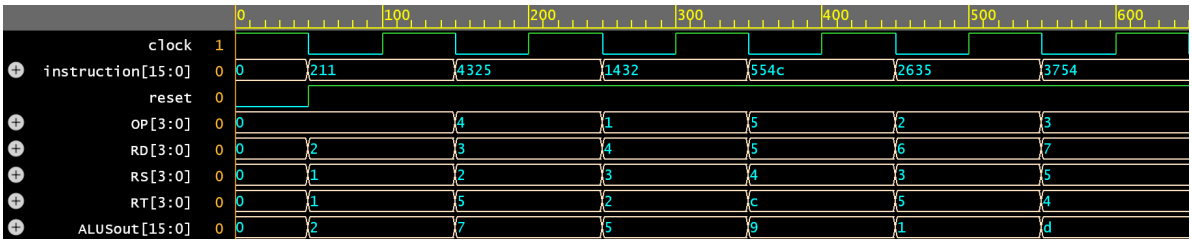

Figure 1: The waveform for the CPU component.

Instruction	op	rd	rs	rt	value (of rd)
`ADD  R2, R1, R1`	0x0	0x2	0x1	0x1	2
`ADDI R3, R2,  5`	0x4	0x3	0x2	0x5	7
`SUB  R4, R3, R2`	0x1	0x4	0x3	0x2	5
`SUBI R5, R4, -4`	0x5	0x5	0x4	0xC	9
`AND  R6, R3, R5`	0x2	0x6	0x3	0x5	1
`OR   R7, R5, R4`	0x3	0x7	0x5	0x4	13

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 teh fact that the ALU does not have any negative results. Table 1 shows the waveform results in table form.

Conclusions

Both the control block and signed-extension components worked correctly shown through the test-bench for the CPU. The knowledge learned from this lab was learning how to construct a control block component and a signed-extension component to support two instructions, `addi` and `subi` that uses an immediate value. The CPU was also rewired to support the use of these components and the two new instructions.