CSCE 22104

Lab Report

Brent Marcus Orlina

ID: 011019116

Lab Section 001

Lab 6

Introduction

This lab's goal was to create a 1-bit ALU component and create a 16-bit ALU component, using 1-bit components. The ALU components should support four operations: addition, subtraction, logical-and, and logical-or, using the opcodes `00`, `01`, `10`, and `11`, respectively. The components should also be asynchronous, i.e. the component does not depend on a clock.

Approach

```
1  entity ALU1Bit is
2      port(
3          s : in std_logic_vector(1 downto 0);
4          a : in std_logic;
5          b : in std_logic;
6          cin  : in std_logic;
7          sout : out std_logic;
8          cout : out std_logic
9      );
10 end ALU1Bit;
```

Listing 1: The 1-bit ALU component's ports.

The 1-bit ALU component was first implemented. Listing 1 shows the ports that the ALU takes in. The input port `s` represents the opcode, determining wether the ALU should execute an addition, subtraction, logical-and, or logical-or operation. The input ports `a` and `b` are the operands. The input port `cin` is the carry-in from a previous ALU component, used for the addition and subtraction operations. It is unused when performing a logical-and or a logical-or operation.

The output port `sout` is the result of the operation that was executed. Output port `cout` is used for addition and subtraction operations, connected to the next ALU component so that it can correctly calculate the result. Although `cin`s and `cout`s are not used for the logical-and and logical-or operations, it will still be calculated regardless. It will not affect the result since the two logical operations do not use the `cin` input port.

```
architecture Datapath of ALU1Bit is
    signal inverse_b  : std_logic;
    signal sout_adder : std_logic;
    signal sout_and   : std_logic;
    signal sout_or    : std_logic;
begin
    inverse_b <= b xor s(0);

    sout_adder <= a xor inverse_b xor cin;
    sout_and   <= a and b;
    sout_or    <= a or  b;

    sout <= sout_adder when s(1) = '0' else
            sout_and   when s = "10" else
            sout_or    when s = "11";
    cout <= (a and inverse_b) or (cin and a) or (cin and inverse_b);
end Datapath;
```

Listing 2: The 1-bit ALU component's datapath implementation.

Listing 2 shows the implementation for each of the operations that the ALU supports. Firstly, the signals `sout_adder`, `sout_and`, and `sout_or`, correspond to the results of each operations that the ALU supports, with `sout_adder` corresponding to both the addition and subtraction operations. The signal `inverse_b` is used to support subtraction. The results of all of the operations are calculated with the input port `s` deciding which result to connect to the output port `sout` by using the operations' respective opcodes.

Notice that since the result of addition and subtraction operations both correspond to only one signal. This allows it so that only the 1^{st} bit is checked to be a 0 since both operations' opcodes have their 1^{st} as a 0. The motivation behind merging both operations into one signal is that binary subtraction in twos complement is equivalent to

$$A - B = A + (\widetilde{B} + 1) \tag{1}$$

therefore, the input port `b` must simply be inverted to support the subtraction operation. Since the addition and subtraction's opcodes are different in the 0^{th} bit where subtraction has a 1, `b` can simply be `XOR`ed by the 0^{th} bit of the opcode, which inverses `b` when the opcode signals for a subtraction operation. This potential inverse of `b` is connected to the `inverse_b` signal, used only by the addition and subtraction operations, so that it does not affect the other operations in which the 0^{th} bit of the opcode is a 1.

The addition of the extra 1 to $\widetilde{B}$ in 1 is not implemented in the 1-bit ALU since if the 16-bit ALU was implemented using sixteen 1-bit ALUs, it would add an extra 1 in each bit of the 16-bit ALU, producing the wrong result. The extra 1 will come from the `cin` of the first 1-bit ALU, again using the 0^{th} bit of the opcode, later shown in [REF to later figure here].

The addition operation remains correct as the input port `b` won't be inverted and there won't be an extra addition of one since the 0^{th} bit of the opcode will be 0. The other two operations, logical-and and logical-or, are trivial by simply performing an `AND` or an `OR` between input ports `a` and `b`. The `cout` output port is also somewhat trivial, implemented similarly to a normal Full Adder. However, it uses the signal `inverse_b` to support subtraction.

```
entity ALU16Bit is
    port(
        s : in std_logic_vector(1 downto 0);
        a : in std_logic_vector(15 downto 0);
        b : in std_logic_vector(15 downto 0);
        sout : out std_logic_vector(15 downto 0);
        cout : out std_logic
    );
end ALU16Bit;

```

Listing 3: The 16-bit ALU component's ports.

Now that the 1-bit ALU has been implemented, the 16-bit ALU can now be constructed using sixteen 1-bit ALU components. Listing 3 shows the 16-bit ALU component's ports. Notice that the input ports `a` and `b` and the output port `sout` are now `std_logic_vectors` with 16 bits each, since the data will be 16 bits long. The input port `s` is unchanged as it still corresponds to the opcode. The output port `cout` also reamins unchanged as it will be the `cout` of the last 1-bit ALU component.

```
architecture Datapath of ALU16Bit is
    component ALU1Bit
        port(
            s : in std_logic_vector(1 downto 0);
            a : in std_logic;
            b : in std_logic;
            cin  : in std_logic;
            sout : out std_logic;
            cout : out std_logic
        );
     end component;
     signal carry : std_logic_vector(14 downto 0);
begin
    bit0 : ALU1Bit port map(s, a(0),  b(0),  s(0),      sout(0),  carry(0) );
    bit1 : ALU1Bit port map(s, a(1),  b(1),  carry(0),  sout(1),  carry(1) );
    bit2 : ALU1Bit port map(s, a(2),  b(2),  carry(1),  sout(2),  carry(2) );
    bit3 : ALU1Bit port map(s, a(3),  b(3),  carry(2),  sout(3),  carry(3) );
    bit4 : ALU1Bit port map(s, a(4),  b(4),  carry(3),  sout(4),  carry(4) );
    bit5 : ALU1Bit port map(s, a(5),  b(5),  carry(4),  sout(5),  carry(5) );
    bit6 : ALU1Bit port map(s, a(6),  b(6),  carry(5),  sout(6),  carry(6) );
    bit7 : ALU1Bit port map(s, a(7),  b(7),  carry(6),  sout(7),  carry(7) );
    bit8 : ALU1Bit port map(s, a(8),  b(8),  carry(7),  sout(8),  carry(8) );
    bit9 : ALU1Bit port map(s, a(9),  b(9),  carry(8),  sout(9),  carry(9) );
    bitA : ALU1Bit port map(s, a(10), b(10), carry(9),  sout(10), carry(10));
    bitB : ALU1Bit port map(s, a(11), b(11), carry(10), sout(11), carry(11));
    bitC : ALU1Bit port map(s, a(12), b(12), carry(11), sout(12), carry(12));
    bitD : ALU1Bit port map(s, a(13), b(13), carry(12), sout(13), carry(13));
    bitE : ALU1Bit port map(s, a(14), b(14), carry(13), sout(14), carry(14));
    bitF : ALU1Bit port map(s, a(15), b(15), carry(14), sout(15), cout     );
end Datapath;

```

Listing 4: The 16-bit ALU component's datapath implementation.

Listing 4 shows the implementation of the 16-bit ALU component, using a sixteen 1-bit ALU components. The opcode, in the input port `s`, is connected to all of the 1-bit ALU components since each 1-bit ALU component needs to know the operation to execute. Each bit of the input ports `a` and `b` as well as the output port `sout` are connected to the corresponding 1-bit ALU component.

To make the addition and subtraction operations work correctly, a 1-bit ALU component's `cout` output port must be connected to the next 1-bit ALU component's `cin` input port, acting as the carry. The last 1-bit ALU component's `cout` output port is connected to the 16-bit ALU component's `cout`. The first 1-bit ALU component's `cin` is connected to the 0^{th} bit of the opcode. This is because if the operation to be executed is subtraction, which has the opcode of `01`, must have an extra 1 added as shown in equation 1. If the operation is addition, the opcode would be `00`, thus not adding an extra 1, correctly calculating the result. The other two operations do not

use the `cin` and `cout` ports, correctly calculating the results bit by bit.

Experimentation

Both components were individually tested using a testbench. The 1-bit ALU component was tested by going through all of the operations, each with all possible combinations of the input ports `a`, `b` and `cin`. The 16-bit ALU component was tested with a given list of inputs and manually verifying that the outputs are correct.

Results & Discussion

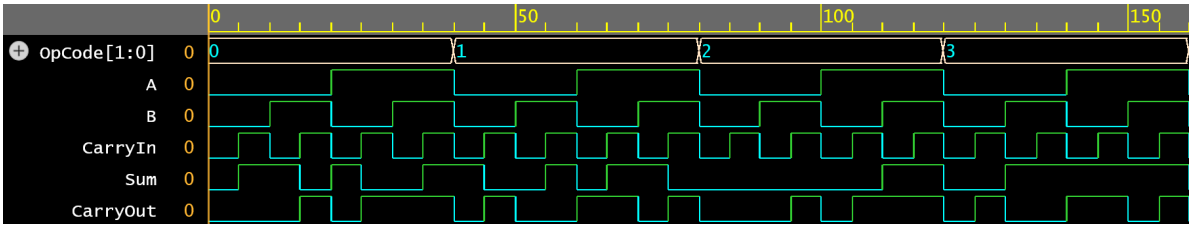

Figure 1: The waveform for the 1-bit ALU component.

The 1-bit ALU component works as expected. Figure 1 shows the waveform of the 1-bit ALU component, correctly outputting the results of each operation. For the subtraction operation, it is noteworthy that the output is unintuitive, especially shown when the signals `A`, `B`, and `CarryIn` are all zeroes while the output shows the signal `Sum` to be active. This is because `B` is being inverted and then being "added" to zeroes, resulting in `Sum` to be active. Though the implementation results in this peculiarity, it is required for the 16-bit ALU to work.

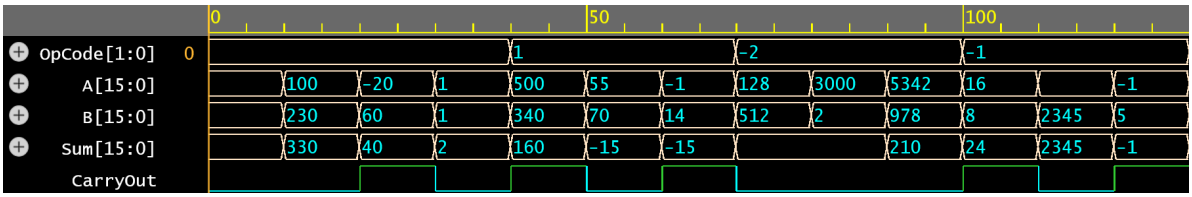

Figure 2: The waveform for the 16-bit ALU component. Signals that display a blank are zeroes.

The 16-bit ALU component works as expected. Figure 2 shows the waveform of the 16-bit ALU component, correctly outputting the results of each operation. The waveform also displays the signal `OpCode` in signed decimal, with the signals `-2` and `-1` corresponding to the opcodes `10` and `11` respectively. Table 1 shows the waveform results in table form.

S1	S0	A	B	Sout	Cout
0	0	100	230	330	0
0	0	-20	60	40	1
0	0	1	1	2	0
0	1	500	340	160	1
0	1	55	70	-15	0
0	1	-1	14	-15	1
1	0	128	512	0	0
1	0	3000	2	0	0
1	0	5342	978	210	0
1	1	16	8	24	1
1	1	0	2345	2345	0
1	1	-1	5	-1	1

Table 1: Results of the 16-bit ALU component testbench in table form.

Conclusions

Both the 1-bit ALU and 16-bit ALU components worked correctly and displayed the expected behavior during testing. The knowledge learned from this lab was learning how to construct 1-bit ALU component that handled multiple operations determined by an opcode. Learning how subtraction works in binary was also a part of the lab, as well as learning how to construct a 16-bit ALU component using out of 1-bit ALU components.