TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

Structure of Computer Systems

ALU: Arithmetic Logic Unit

Doia Bogdan-Mihai

30431

Contents

1. Introduction

1.1 Context

The primary goal of this project is to develop an Arithmetic Logic Unit (ALU) capable of performing various mathematical operations on X-bit numbers. These operations encompass addition, subtraction, incrementing, decrementing, logic operations, negation, left and right rotation, as well as multiplication and division.

The designed ALU is intended to swiftly and accurately compute the outcomes of these operations when applied to X-bit numbers. It can be used as a standalone device or integrated into processors, such as the MIPS processor, to make use of its computational capabilities.

1.2 Specifications

The device will be simulated in the IDE provided by Vivado and then programmed into a Basys3 board. It should be able to complete all of the operations specified before, with addition and subtraction being done on the 2’s complement representation of the numbers, and then it should be able to show the results in an easy to read manner for the users.

1.3 Objectives

The project should meet several key objectives. These include devising a mechanism to input numbers for computation by the ALU, which will also include the ability to represent binary numbers using the 2's complement representation through an encoder. Additionally, it should have a control unit for the ALU and a separate component dedicated to handling multiplication and division operations. Moreover, it will incorporate the necessary logic components to facilitate the display of results on the 7-segment display of the Basys3 board.

2. Bibliographic study

2.1 Working with 2’s complement

In order to perform addition and subtraction in 2’s complement, what needs to be first understood is the difference between normal binary and 2’s complement.

In normal binary representation, each bit holds a positive or negative power of 2, and the leftmost bit (most significant bit) is the sign bit, determining whether the number is positive (0) or negative (1) in a signed integer representation. On the other hand, two's complement is a method for representing signed integers that simplifies arithmetic operations. In two's complement, negative numbers are represented by taking the one's complement (flipping the bits) of the positive number and then adding 1. This technique eliminates the need for a separate sign bit and ensures that addition and subtraction operations can be performed using standard binary addition.

Converting a binary number to its two's complement involves finding the one's complement (flipping the bits) and then adding 1 to the result. This method provides an efficient way to represent both positive and negative integers in a fixed number of bits.

2.2 Multiplication and division

In binary arithmetic, multiplication involves systematically multiplying corresponding bits of the multiplier and multiplicand, shifting left, and summing the partial products. For instance, multiplying 1101 by 1010 entails computing partial products and shifting them appropriately, resulting in the binary product 1000110. On the other hand, binary division follows a long division approach, subtracting the divisor from the dividend, noting the quotient bit, and iteratively shifting and bringing down the next bit. For example, dividing 110110 by 101 yields a quotient of 1, and the process continues until all bits are processed, leaving no remainder. These binary multiplication and division methods, though akin to their decimal counterparts, cater to the binary nature of computers, facilitating efficient computation in the digital realm.

2.3 Logic operations

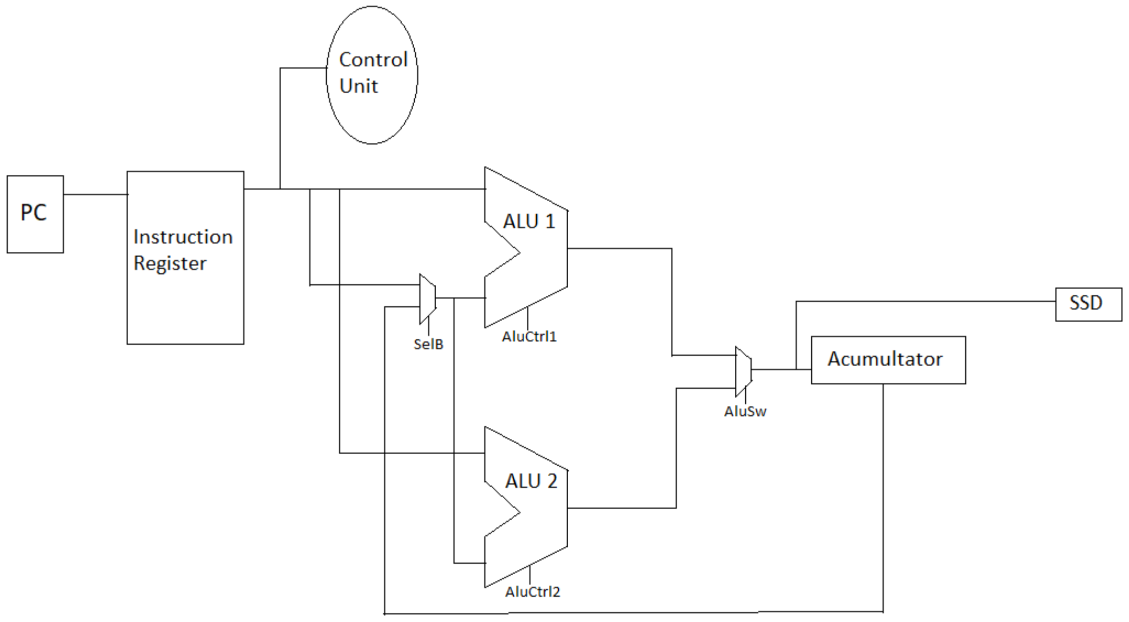
In binary logic operations, AND involves comparing corresponding bits of two binary numbers, producing a result where a bit is set to 1 only if both input bits are 1. NOR, in contrast, performs a logical OR and then negates the result, yielding 1 only if both input bits are 0. The NOT operation simply flips the bits, changing 0s to 1s and vice versa. Shifting operations, like shift left and shift right, involve moving the bits in a binary number. Shifting left effectively multiplies the binary number by 2, while shifting right divides it by 2. To shift left, append zeros on the right, and to shift right, discard the rightmost bits. For each left shift, the binary number's value increases by a factor of 2, similar to multiplication, and for each right shift, the value decreases, resembling division by 2.

3. Analysis

By analysing the project's requirements we can draw some conclusions: it must have 2 separate arithmetic logic units, one performing the 2's complement operations and the logical operations, and another one performing multiplication and division; a separate component must be designed in order to store the result of the previous computation; necessary adaptations and addition logic must be thought of in order to accommodate the use of the Basys3 board and possible hazards that may appear due to hardware limitations, such as the button registering more than one impulse on press and the seven segment display of the board being able to represent at most 4 characters at once.

4. Design

In order to realize the project's tasks, I have designed the following data-path and components:



As seen above, several components were needed to be designed in order to achieve the task. The program counter, the accumulator and the seven segment display are synchronous components, while all the others are asynchronous. The program counter and the accumulator can also be reset.

Program Counter

The program counter has the role of incrementing the index of the instruction register, effectively moving on to the next instruction present in the memory.

It receives input by pressing the button on the basys3 board, the signal first being passed through the mono pulse generator, and outputs the index of the instructions.

Instruction Register

The Instruction Register is where the instructions are stored and where they are fetched from and sent to the other components. It essentially is a ROM memory which receives the index as input from the program counter and return the value found in memory at the index address.

Main Control Unit

The Main Control Unit acts as the "brain" of the program, effectively decoding the instructions and controlling the rest of the components and their operations by assigning values to the control signals according to the instruction format.

Each instruction consists of 21 bits that are split into parts as follows: first 4 bits are the opcode which indicates the operation to be performed, next 8 bits are the 1st number, next 8 bits the 2nd number and finally the last bit indicates whether the actual 2nd number should be used, or the previous result in the program's execution.

Each operation has a unique operation code:

- 0000 ADD
- 0001 SUB
- 0010 INC
- 0011 DEC
- 0100 AND
- 0101 OR
- 0110 NOT
- 0111 SLL

- 1000 SRL
- 1001 MUL
- 1010 DIV

There are 4 control signals SelB, which selects the 2nd number to be used based on the last bit of the instruction, AluCtrl1 and AluCtrl2 which are based on the opcode and AluSw which indicates which of the 2 ALUs has done the operation.

Therefore, their values are as follows:

SelB – the value of the last bit in the instruction (0- use B, 1- use previous result).

AluCtrl1 – opcode for all operations other than multiply and divide, else 1111.

AluCtrl2 – opcode for multiply and divie, else 1111.

AluSw – 1 for multiply and divide, 0 for all the other operations.

ALU1

The first arithmetic logic unit takes care of addition and subtraction in 2's complement and the logic operations.

Despite being relatively the same operation when it comes to 2's complement, I have decided to implement both addition and subtraction as standalone operations for clarity and ease of working.

This component is controlled by the control unit using the AluCtrl1 signal and receives 2 inputs of 8 bits and return the result of the selected operation as a number represented on 16 bits.

ALU2

The second arithmetic logic unit takes care of multiplication and division.

An important part of this component was handling the classic division by 0 problem, in which case the result it returns will be 0.

This component is controlled by the control unit using the AluCtrl2 signal. Taking as input 2 numbers of 8 bits each, it return the result of the operation on 16 bits.

Acumulator

As per the project's task, an extra component that stores the value of the previous result to be later used as input value if decided so by the instruction and the control signals was implemented. It is a simple register that has an input and an output both on 16 bits.

Seven segment display

The SSD is a classic component used to be able to display the values of the results on the basys3 board. It has as input the value to be displayed and as output the anode and cathode.

Additional logic

To be able to control the flow of the program and of its' data several multiplexers were used to select different values based on the control unit's signals.

5. Implementation

The implementation of the components was done in behavioural style, the top level having all the components integrated using port maps. Bellow is the code for the more important parts of the program.

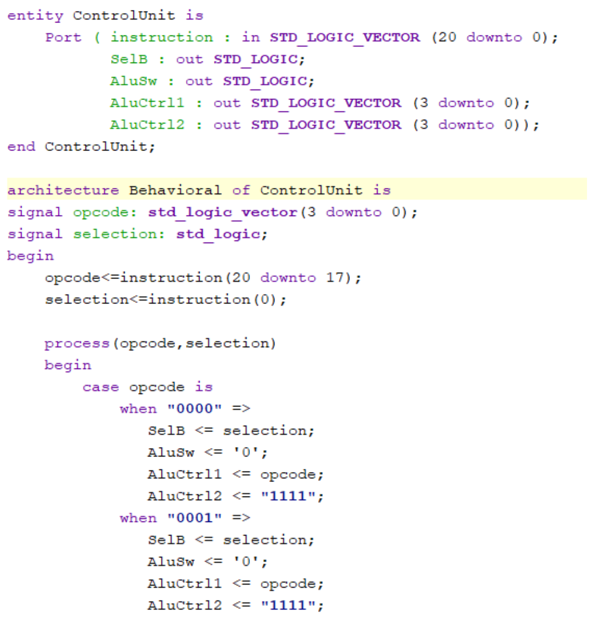
Instruction Register

```vhdl
entity InstructionRegister is
    Port ( adr : in STD_LOGIC_VECTOR (3 downto 0);
           d : out STD_LOGIC_VECTOR (20 downto 0));
end InstructionRegister;

architecture Behavioral of InstructionRegister is
type memorie is array (0 to 30) of std_logic_vector(20 downto 0);
signal stocare:memorie :=(0=>B"0000_00000101_11111111_0",
                          1=>B"0000_00000001_00000000_1",
                          2=>B"0001_00000100_00000110_0",
                          3=>B"0010_00010010_00000000_0",
                          4=>B"0011_00100000_00000000_0",
                          5=>B"0100_00000111_00000101_0",
                          6=>B"0101_00001011_00000000_1",
                          7=>B"0110_11001000_00000000_0",
                          8=>B"0111_00001011_00000000_0",
                          9=>B"1000_00101000_00000000_0",
                          10=>B"1001_00000011_00000101_0",
                          11=>B"1010_00001110_00000111_0",
                          others=>B"0000_00000000_00000000_0");


begin
d<=stocare(conv_integer(adr));
end Behavioral;
```

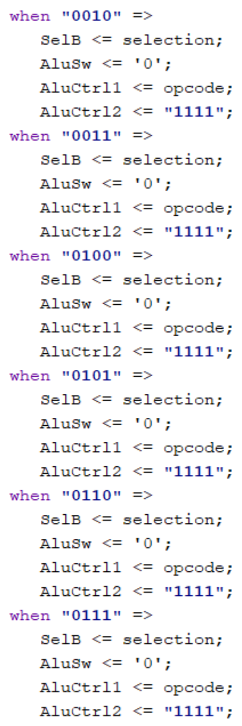
Main Control Unit



```vhdl
entity ControlUnit is
    Port ( instruction : in STD_LOGIC_VECTOR (20 downto 0);
           SelB : out STD_LOGIC;
           AluSw : out STD_LOGIC;
           AluCtrl1 : out STD_LOGIC_VECTOR (3 downto 0);
           AluCtrl2 : out STD_LOGIC_VECTOR (3 downto 0));
end ControlUnit;

architecture Behavioral of ControlUnit is
signal opcode: std_logic_vector(3 downto 0);
signal selection: std_logic;
begin
    opcode<=instruction(20 downto 17);
    selection<=instruction(0);

    process(opcode,selection)
    begin
        case opcode is
            when "0000" =>
               SelB <= selection;
               AluSw <= '0';
               AluCtrl1 <= opcode;
               AluCtrl2 <= "1111";
            when "0001" =>
               SelB <= selection;
               AluSw <= '0';
               AluCtrl1 <= opcode;
               AluCtrl2 <= "1111";
```



```vhdl
            when "0010" =>
               SelB <= selection;
               AluSw <= '0';
               AluCtrl1 <= opcode;
               AluCtrl2 <= "1111";
            when "0011" =>
               SelB <= selection;
               AluSw <= '0';
               AluCtrl1 <= opcode;
               AluCtrl2 <= "1111";
            when "0100" =>
               SelB <= selection;
               AluSw <= '0';
               AluCtrl1 <= opcode;
               AluCtrl2 <= "1111";
            when "0101" =>
               SelB <= selection;
               AluSw <= '0';
               AluCtrl1 <= opcode;
               AluCtrl2 <= "1111";
            when "0110" =>
               SelB <= selection;
               AluSw <= '0';
               AluCtrl1 <= opcode;
               AluCtrl2 <= "1111";
            when "0111" =>
               SelB <= selection;
               AluSw <= '0';
               AluCtrl1 <= opcode;
               AluCtrl2 <= "1111";
            when "1000" =>
               SelB <= selection;
               AluSw <= '0';
               AluCtrl1 <= opcode;
               AluCtrl2 <= "1111";
            when "1001" =>
               SelB <= selection;
               AluSw <= '1';
               AluCtrl1 <= "1111";
               AluCtrl2 <= opcode;
            when "1010" =>
               SelB <= selection;
               AluSw <= '1';
               AluCtrl1 <= "1111";
               AluCtrl2 <= opcode;
            when others =>
               SelB <= selection;
               AluSw <= '0';
               AluCtrl1 <= "1111";
               AluCtrl2 <= "1111";
         end case;
    end process;

end Behavioral;
```

ALU1

```
entity ALU1 is
    Port (
        A, B : in  std_logic_vector(7 downto 0);
        AluCtrl1 : in  std_logic_vector(3 downto 0);
        result : out std_logic_vector(15 downto 0)
    );
end ALU1;

architecture Behavioral of ALU1 is
begin
    process(A, B, AluCtrl1)
        variable temp : std_logic_vector(7 downto 0);
    begin
        case AluCtrl1 is
            when "0000" =>  -- Add in 2s complement
                temp := std_logic_vector(signed(A) + signed(B));
                result <= "00000000" & temp(7 downto 0);
            when "0001" =>  -- Subtract in 2s complement
                temp := std_logic_vector(signed(A) - signed(B));
                result <= "00000000" & temp(7 downto 0);

            when "0010" =>  -- Increment
                temp := std_logic_vector(signed(A) + 1);
                result <= "00000000" & temp(7 downto 0);
            when "0011" =>  -- Decrement
                temp := std_logic_vector(signed(A) - 1);
                result <= "00000000" & temp(7 downto 0);
            when "0100" =>  -- AND logical
                result <= "00000000" & (A and B);
            when "0101" =>  -- OR logical
                result <= "00000000" & (A or B);
            when "0110" =>  -- NOT
                result <= "00000000" & (not A);
            when "0111" =>  -- Shift Left
                result <= "0000000" & A(7 downto 0) & "0";
            when "1000" =>  -- Shift Right
                result <= "000000000" & A(7 downto 1);
            when others =>
                result <= (others => '0');
        end case;
    end process;
end Behavioral;
```

ALU2

```
entity ALU2 is
    Port (
        A, B : in  std_logic_vector(7 downto 0);
        AluCtrl2 : in  std_logic_vector(3 downto 0);
        result : out std_logic_vector(15 downto 0)
    );
end ALU2;

architecture Behavioral of ALU2 is
    signal temp : integer range 0 to 255;
    signal resultaux: std_logic_vector(7 downto 0);
begin
    process(A, B, AluCtrl2)
    begin
        case AluCtrl2 is
            when "1001" =>  -- Mul
                temp <= to_integer(unsigned(A)) * to_integer(unsigned(B));
                resultaux <= std_logic_vector(to_unsigned(temp, 8));
            when "1010" =>  -- Div
                if B /= x"00" then  --division by zero
                    temp <= to_integer(unsigned(A)) / to_integer(unsigned(B));
                    resultaux <= std_logic_vector(to_unsigned(temp, 8));
                else
                    resultaux <= (others => '0');
                end if;
            when others =>
                resultaux <= (others => '0');
        end case;
        result <= "00000000" & resultaux;
    end process;
end Behavioral;
```

Top level output

```
process(count,Acum)
begin
    case sw(1 downto 0) is
        when "00"=> SSDOutput <= AluRes;
        when "01"=> SSDOutput <= Acum;
        when "10"=> SSDOutput <= "00000000" & BInput;
        when others=> SSDOutput <= "000000000000" & count;
    end case;
    if AluRes(1)='1' then
        led(0)<='1';
    end if;
end process;
```

This process in the top level entity helps output different values from the program on the ssd. Also here it is signaled by turning a led on whether the result displayed is a negative number in 2's complement.

6. Testing and validation

The instructions used to test the program were the following operations encoded according to the instruction convention presented previously. Those instructions can be seen in the instruction register's code above. Below is the operation and the expected result, all the numbers represented in decimal.

- 5 + -1 = 4
- 1 + 4(previous result) = 5
- 4 – 6 = -2
- 18 increment = 19
- 32 decrement = 31
- 7 and 5 = 5
- 11 or 5(previous result) = 15
- not 200 = 55
- shift left logical 11 = 22
- shift right logical 40 = 20
- 3 * 5 = 15
- 14 : 7 = 2

The program was tested using the Basys3 board like so:

- pressing the increment button and observing the change on the seven segment display.

- switching between the output modes present in the code right above this section using the board's switches.

- resetting the program counter by pressing the reset button.

- going through all the operations and seeing if after 15 instructions the program resets back to the first one.

- checking if the negative result returned by the 3rd instruction lights up the led, indicating the negative representation in 2's complement.

All the instructions were tested on the board and the expected behaviour for each of the steps above was observed. All the instructions also returned the correct and expected results, which indicates the correctness of the program.

7. Conclusions

The goal of the project was to design 2 arithmetic logic units that perform addition and subtraction in 2's complement, logical operations and multiplication and division respectively, a separate component that stores the result of the previous operation and all the other necessary logic to ensure the proper flow of the program and to aid visualising the results on the Basys3 board.

What I found most difficult about this project was making sure the interactions between the numbers in 2's complement and the ones in normal binary caused by the possibility of reusing previous results did not affect the program in a weird way, and the implementation of that accumulator in on itself presented some problems because of the clock signal and the synchronization with the rest of the components.

I have practiced my knowledge from this semester and also previous academic years, this being the first project for which I think the design from scratch, not just writing code based on existing data-paths and instructions and I enjoyed this part of it a lot.

8. Bibliography

https://users.utcluj.ro/~negrum/index.php/computer-architecture/

https://digilent.com/reference/programmable-logic/basys-3/start

https://www.exploringbinary.com/twos-complement-converter/

Octavian Creţ, Lucia Văcariu – *Probleme de proiectare logică a sistemelor numerice. Logic Design Problems for Digital Systems*. Editura UTPres, Cluj-Napoca, ROMÂNIA, 2008,