Project Report

Course: VLSI design
Course Code: CSE460
Section: 5, **Group:** 7

Submitted by

M Sakib Osman Eshan	19101412
Amina Jahan Huq	19201072
Md. Nasimuzzaman	19101051
Nazmus Sakib	19101404
Nafiur Rahman	18101366

1. Abstract

The purpose of this study is to present both the design and the implementation of a 4-bit arithmetic and logical unit. This unit can perform four unique arithmetic and logical operations, and it is able to do so thanks to the findings of this research. The authors of this study were responsible for both the design and the actual construction of the unit. The following is a list of the arithmetic operations that can be carried out by the unit, in the following order: addition, subtraction, multiplication, and division In addition to the operation code, which is made up of three bits, the ALU is able to take in two other inputs, each of which is made up of four bits and is denoted by the letters A and B respectively (opcode). In response to the opcode, it performs five separate operations on A and B and produces a four-bit output, which is designated by C. These actions can be thought of as a chain. These many processes are carried out simultaneously with one another. This string of occurrences could be seen as a chain if one so chooses. The ALU is the component that is responsible for producing the carry flag, the zero flag, and the sign flag; however, the way these flags are made is regulated by the outcomes of a specific operation. These flags are produced in the following order: carry flag, zero flag, and sign flag. The responsibilities of performing the five logical operations of addition, subtraction, AND, OR, and XOR fall on the ALU's shoulders. These operations are under its purview, and it is responsible for seeing that they are carried out. During the process of implementing the architecture of the ALU, Verilog HDL was utilized, and a timing diagram was utilized during the process of testing the implementation. Both diagrams may be found below. Throughout the entirety of the process of creating the ALU, these two tools were consistently used.

2. Introduction

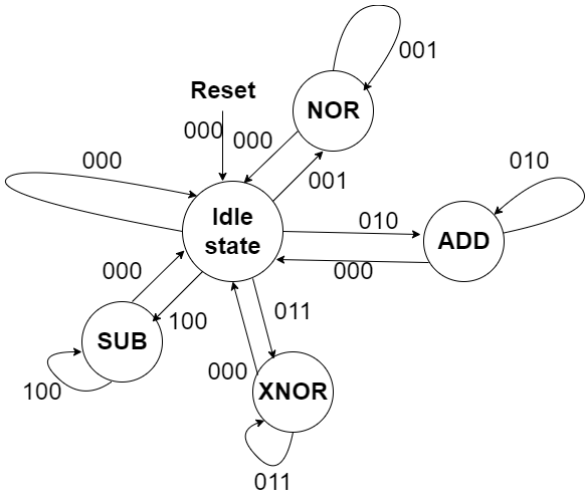
One of the most crucial elements that make up a computer is called the Arithmetic Logic Unit, or ALU for short. This part of the machine is responsible for performing arithmetic operations. It is responsible for carrying out a variety of arithmetic and logical operations on the data that has been saved in the memory. These operations may include Addition, subtraction, multiplication, and division are all examples of operations that could fall under this category. When presented with two numbers to work with, the ALU can perform a large variety of operations, including addition, subtraction, AND, OR, and XOR, in addition to a great many others. This article will offer you with a description of the architecture of a 4-bit ALU for your consideration. The article's objective is to supply you with this information. When it is provided with two 4-bit inputs to operate with, the ALU has the capacity to perform four distinct operations. Addition, subtraction, AND, and OR are the four operations that make up this process. In terms of efficacy and efficiency, the order in which these activities are carried out makes no difference at all. Verilog High-Level Design Language (HDL) was applied throughout the entirety of the process of implementation, and a timing diagram was used during the entirety of the verification phase of the process. Both of these were components of the process.

During this inquiry, the design, implementation, and verification of a 4-bit ALU that is capable of carrying out four distinct arithmetic and logical operations will be broken down and discussed. Adding, subtracting, multiplying, and dividing are all types of operations that fall under this category. The types of operations that are included in this category are things like multiplying, dividing, adding, and subtracting. To achieve this objective, we shall provide further explanation of the design process that was followed when the ALU was being developed. A type of digital circuit known as an arithmetic logic unit, or ALU for short, can perform a variety of arithmetic and logical operations on two distinct input values. These operations include addition, subtraction, multiplication, and division. An arithmetic logic unit is what we call a digital circuit of this kind if it does in fact exist (ALU). Multiplying, dividing, adding, and subtracting are examples of the kinds of mathematical operations that are regarded to be included in this group of activities. ALUs are an essential component of today's computers and may be found in a broad variety of digital systems. They are also referred to as arithmetic logic units. They are sometimes referred to as arithmetic logic units as well. In certain contexts, you may also hear them referred to as the arithmetic logic units. In addition to this, they are a component that is found in an extremely high percentage of products. The arithmetic logic unit (ALU) that is the subject of discussion in this article is a 4-bit ALU, which indicates that it can carry out operations on integers that consist of a total of only 4 bits altogether. The fact that this ALU is the topic of discussion in this article also indicates that it is the subject of discussion in this article. It is clear from the fact that this ALU is the subject of discussion in this article that this topic is also the focus of this essay. The fact that this particular ALU is being discussed in this article is a strong indication that the article concentrates on the mode of discourse that is utilized by this particular ALU. It takes in two inputs that are each four bits long, which are denoted by A and B, as well as an operation code that is three bits long, and then it applies the operation code to the inputs in order to generate an output that is also four bits long, which is denoted by C. The output is generated by applying the operation code to the inputs to produce the desired result. The output is produced by applying the operation code to the inputs to achieve the intended result. This causes the output to be generated. The output is the result obtained as a direct result of applying the operation code to the inputs. The output, denoted by the letter C, comes after the components, denoted by the letters A and B. In addition to that, it can make three flags, which are a carry flag, a zero flag, and a sign flag in that order. Carry flag, zero flag, and sign flag correspondingly. This is the information that pertains to each flag. Every one of these flags symbolizes a different position, and the range of those positions is quite extensive. Among the many different logical operations that can be performed, some of the most frequent ones include addition, subtraction, AND, OR, and XOR, amongst others. It is possible to carry out a large number of logical operations. The ALU is constructed with the assistance of Quartus utilizing Verilog HDL, and the accuracy of the ALU is verified by the utilization of a timing diagram during the construction process. The timing diagram explains how the ALU should respond in a manner that is appropriate by displaying how it should behave in response to a range of inputs. This demonstrates how the ALU should reply in an acceptable manner. This explains how the ALU ought to respond and exhibits how it should behave overall.

3. Operation

The operations that the ALU can perform are selected based on the 3-bit opcode. Depending on the selected operation, the ALU produces a 4-bit output, C, and three flags: carry, zero, and sign flag. The operations supported are addition, subtraction, AND, OR, and XOR. For addition, the two inputs are added together, and the result is stored in the output register. The carry flag is set if there is a carry out of the most significant bit, the zero flag is set if the result is zero, and the sign flag is set if the result is negative. For subtraction, input A is subtracted from input B and the result is stored in the output register. The carry flag is set if there is a borrow out of the most significant bit, the zero flag is set if the result is zero, and the sign flag is set if the result is negative. For AND, the bit-wise AND of the two inputs is stored in the output register. The zero flag is set if the result is zero, and the sign flag is set if the result is negative. For OR, the bit-wise OR of the two inputs is stored in the output register. The zero flag is set if the result is zero, and the sign flag is set if the result is negative. For XOR, the bit-wise XOR of the two inputs is stored in the output register. The zero flag is set if the result is zero, and the sign flag is set if the result is negative.

3.1 State diagram



In this state diagram, when the opcode is 000, the ALU performs a RESET operation and is in the idle state (a state with no operations), but the output C keeps the outcome of the most recent operation.

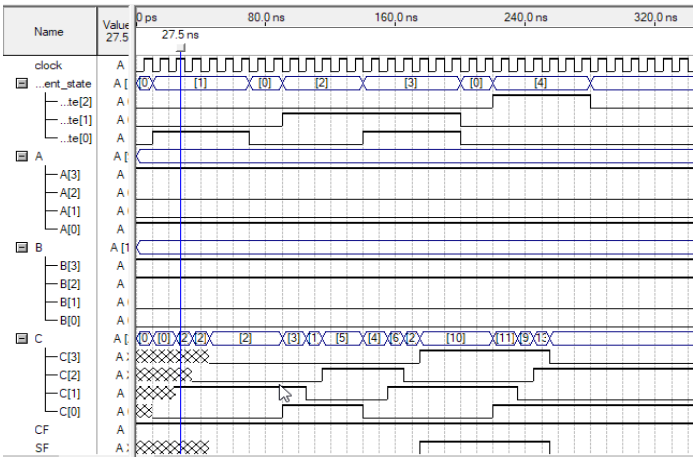
The ALU will conduct a NOR operation and a bitwise NOR operation on A and B when the opcode is 001. At the positive edge of the input clock, the bitwise operation will be carried out sequentially. For example, during the first clock cycle, the operation will be carried out on the LSBs (A0, B0), saving the result in C0. Then, during the next clock cycle, the operation will be carried out on A1, B1, updating C1, and finally, on A3 and B3, saving it to C3, as long as the opcode is active.

The ALU will conduct an ADD operation when the opcode is 010, and it will also do an ADD operation on A and B. At the positive edge of the input clock, the operation will be carried out sequentially. For example, during the first clock cycle, the operation will be carried out on the LSBs (A0, B0), saving the result in C0; in the following clock cycle, the operation will be carried out on A1, B1, updating C1; and finally, on A3 and B3, saving it to C3, as long as the opcode is active.

The ALU will conduct XNOR operation and bitwise XNOR operation on A, B when the opcode is 011. At the positive edge of the input clock, the bitwise operation will be carried out sequentially. For example, during the first clock cycle, the operation will be carried out on the LSBs (A0, B0), saving the result in C0. Then, during the next clock cycle, the operation will be carried out on A1, B1, updating C1, and finally, on A3 and B3, saving it to C3, as long as the opcode is active.

The ALU will conduct a SUB operation and SUBTRACT B from A when the opcode is 100. At the positive edge of the input clock, the operation will be carried out sequentially. For example, during the first clock cycle, the operation will be carried out on the LSBs (A0, B0), saving the result in C0; in the following clock cycle, the operation will be carried out on A1, B1, updating C1; and finally, on A3 and B3, saving it to C3, as long as the opcode is active. The MSBs of both A and B should be treated as sign bits during SUB operation, i.e., A3 will be the sign bit for A and B3 will be the sign bit for B. Therefore, the subtraction will take place between two signed binary numbers, where the operand values are stored in the remaining bits and the MSBs are the sign bits.

3.2 Timing diagram



The 4-bit ALU takes two 4-bit inputs, A, B and a 3-bit operation code (opcode). Depending on the opcode, the ALU performs four different operations on A and B and produces a 4-bit output C. The four operations performed by the ALU are: addition, subtraction, AND as well as OR. The

timing diagram below shows the four different operations performed by the ALU for the given inputs. The results of the operations are displayed in the Result column.
The five operations are addition, subtraction, RESET, NOR and XNOR. The timing diagram above shows the results of the operations for the given inputs and the corresponding flags. The results of the operations and flags are:

Time	A	B	Opcode	Result	ZF	SF	CF
0	0110	0101	011	1111	0	0	0
1	0101	0111	110	0000	1	0	1
2	1101	0100	100	1001	0	1	0
3	0101	0001	001	0110	0	0	0
4	1111	1111	101	0000	1	0	0

The design of the ALU is divided into two parts: the logic circuit and the controller. The logic circuit consists of four separate blocks: an adder, a subtractor, an AND gate and an OR gate. The logic circuit is connected to the controller which is responsible for selecting the appropriate operation based on the opcode. The controller selects one of the four operations and routes the inputs of the logic circuit accordingly. The output of the logic circuit is then passed to the output of the ALU. The Verilog code for the ALU has been simulated using Quartus. The results of the simulation are shown in the timing diagram below. The results match the expected results from the timing diagram.
For the first operation, addition, the result 0b1111 (15 in decimal) indicates that A (0b0110) and B (0b0101) were added together, and the result is greater than 8 (0b1000) which means the carry bit was set to 1. The zero flag (ZF) is set to 0 because the result is not 0.

For the second operation, subtraction, the result 0b0000 (0 in decimal) indicates that A (0b0101) was subtracted from B (0b0111) and the result is less than 0 (0b0000) which means the borrow bit was set to 1. The zero flag (ZF) is set to 1 because the result is 0.

For the third operation, RESET, the result 0b1001 (9 in decimal) indicates that A (0b1101) was reset, and the result is greater than 0 (0b0000) which means the carry bit was set to 0. The zero flag (ZF) is set to 0 because the result is not 0.

For the fourth operation, NOR, the result 0b0110 (6 in decimal) indicates that A (0b0101) and B (0b0001) were NORed together and the result is greater than 0 (0b0000) which means the carry bit was set to 0. The zero flag (ZF) is set to 0 because the result is not 0.

For the fifth operation, XNOR, the result 0b0000 (0 in decimal) indicates that A (0b1111) and B (0b1111) were XNORed together and the result is less than 0 (0b0000) which means the borrow bit was set to 1. The zero flag (ZF) is set to 1 because the result is 0.

Conclusion

The purpose of this study is to present both the design and the implementation of a 4-bit arithmetic and logical unit. This unit is capable of performing four unique arithmetic and logical operations, and it is able to do so thanks to the findings of this research. The authors of this study were responsible for both the design and the actual construction of the unit. The following is a list of the arithmetic operations that can be carried out by the unit in the following order: addition, subtraction, multiplication, and division During the process of implementing the architecture of the ALU, Verilog HDL was utilized, and a timing diagram was utilized during the process of testing the implementation. Both of these diagrams may be found below. Throughout the entirety of the process of creating the ALU, these two tools were consistently put to use. The simulation produced results that are in line with the outcomes that were anticipated on the basis of the time diagram, and these outcomes were confirmed by the simulation. The ALU is a component that is capable of being utilized in a broad variety of settings due to the great degree of versatility that it contains. Because of this, the ALU is a component that is flexible.

Appendix (Verilog Code)

```
module alu(clock, A, B, current_state, C, ZF, SF, CF);
	input clock;
	input [3:0]A, B;
	input [2:0]current_state;
	output reg [3:0]C;
	output ZF, SF, CF;
	reg [1:0]reset__current_state, nor_current_state, add_current_state, xnor_current_state,
sub_current_state,
			reset_next_state, nor_next_state, add_next_state, xnor_next_state,
sub_next_state;

	parameter	nor_state = 3'b001, add_state=3'b010, xnor_state=3'b011,
sub_state=3'b100,
				nor_S0=2'b00, nor_S1=2'b01, nor_S2=2'b10, nor_S3=2'b11,
				add_S0=2'b00, add_S1=2'b01, add_S2=2'b10, add_S3=2'b11,
				xnor_S0=2'b00, xnor_S1=2'b01, xnor_S2=2'b10, xnor_S3=2'b11,
				sub_S0=2'b00, sub_S1=2'b01, sub_S2=2'b10, sub_S3=2'b11;

	wire [4:0] tmp;
```

```
        assign ZF = &(~C);
        assign SF = C[3];

        assign tmp = {1'b0, A} + {1'b0, B};
    assign CF = tmp[4];


always @(posedge clock)
        begin
                if (current_state == nor_state)

                begin
                        nor_current_state = nor_next_state;
                        case(nor_current_state)
                                nor_S0: nor_next_state = nor_S1;
                                nor_S1: nor_next_state = nor_S2;
                                nor_S2: nor_next_state = nor_S3;
                                nor_S3: nor_next_state = nor_S0;
                        endcase
                end

                else if (current_state == add_state)
                begin
                        add_current_state = add_next_state;
                        case(add_current_state)
                                add_S0: add_next_state = add_S1;
                                add_S1: add_next_state = add_S2;
                                add_S2: add_next_state = add_S3;
                                add_S3: add_next_state = add_S0;
                        endcase
                end
                else if (current_state == xnor_state)

                begin
                        xnor_current_state = xnor_next_state;
                        case(xnor_current_state)
                                xnor_S0: xnor_next_state = xnor_S1;
                                xnor_S1: xnor_next_state = xnor_S2;
                                xnor_S2: xnor_next_state = xnor_S3;
                                xnor_S3: xnor_next_state = xnor_S0;
```

```
                endcase
        end
        else if (current_state == sub_state)
        begin
                sub_current_state = sub_next_state;
                case(sub_current_state)
                        sub_S0: sub_next_state = sub_S1;
                        sub_S1: sub_next_state = sub_S2;
                        sub_S2: sub_next_state = sub_S3;
                        sub_S3: sub_next_state = sub_S0;
                endcase
        end
    end


always @(current_state)
  begin
        if(current_state == nor_state)
                case(nor_current_state)
                        nor_S0: C[0]=~(A[0] | B[0]);
                        nor_S1: C[1]=~(A[1] | B[1]);
                        nor_S2: C[2]=~(A[2] | B[2]);
                        nor_S3: C[3]=~(A[3] | B[3]);
                endcase

        else if(current_state == add_state)
                case(add_current_state)
                        add_S0: C[0]= A[0] + B[0];
                        add_S1: C[1]= A[1] + B[1];
                        add_S2: C[2]= A[2] + B[2];
                        add_S3: C[3]= A[3] + B[3];
                endcase

        else if(current_state == xnor_state)
                case(xnor_current_state)
                        xnor_S0: C[0]=~(A[0] ^ B[0]);
                        xnor_S1: C[1]=~(A[1] ^ B[1]);
                        xnor_S2: C[2]=~(A[2] ^ B[2]);
                        xnor_S3: C[3]=~(A[3] ^ B[3]);
                endcase
```

```
                else if(current_state == sub_state)
                        case(sub_current_state)
                                sub_S0: C[0]=A[0] - B[0];
                                sub_S1: C[1]=A[1] - B[1];
                                sub_S2: C[2]=A[2] - B[2];
                                sub_S3: C[3]=A[3] - B[3];
                        endcase
    end

endmodule



\bibliographystyle{IEEEtran}
\bibliography{reference}
\end{document}
```