32-bit Arithmetic Logic Unit

Chețe Doru-Gabriel

Contents:

1. Introduction

1.1 Context

The goal of this project is to design, implement and test a 32-bit Arithmetic Logic Unit. This unit should be capable of performing addition, subtraction, multiplication, division, logical operations such as OR, AND or NOT, as well as rotations right or left. This ALU will function using an accumulator. So, this unit, besides being a calculator for integer operations, could be used within a larger architecture, such as the MIPS, since it can also perform logical operations.

1.2 Specifications

This unit will be described and simulated in the IDE provided by Vivado and then, using a Basys3 FPGA, it will be tested. For the arithmetic operations, the unit will use Two's Complement representation. For multiplication, the operands will need to be represented on 16 bits, since their result will need a maximum number of 32 bits to be represented. The other arithmetic operation will use 1 bit for their sign and the rest of the 31 bits for the representation of the number.

1.3 Objectives

Design and implement addition and subtraction on 32 bits using Two's Complement. Build the 32-bit Adder in a structural manner using a 1-bit Adder. Use an accumulator for performing these operations (the accumulator being one of the operands) and storing the

result. Find a way to perform multiplication and division, since these are more complicated operations that require further considerations and will be part of a separate supplementary unit. Implement the 3 required logical operations: OR, AND, NOT. Design and implement as part of the further functionality of the ALU rotations left and right. Use a control unit to control the operations performed in the whole Arithmetic Logic Unit.

2. Bibliographic study

The ALU will perform a range of operations, some of them logical and some of them arithmetic, so will tackle the two functionalities differently. Firstly, for the arithmetic operations, we'll begin with addition and subtraction on signed (Two's Complement) numbers.

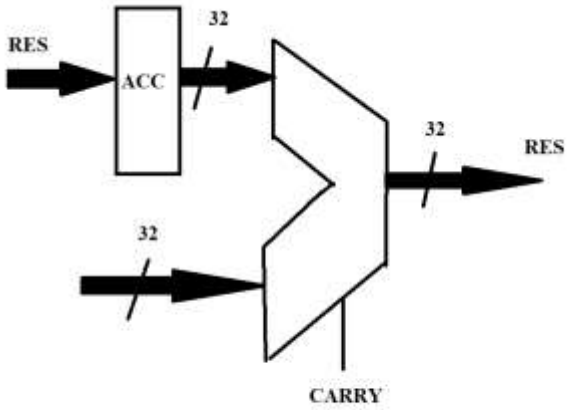


Figure 1 General structure of the unit: accumulator is an operand, as well as the destination of the result

Two's Complement is a way of representing binary numbers that will be used for the purpose of performing signed addition and subtraction. This representation is obtained by inverting all bits of the binary representation of a number and then adding 1. The range for this representation is -2^{n-1} to $2^{n-1}-1$ where n is the number of bits used for representing the

number. The leftmost bit is used to indicate the sign. The leftmost bit has a weight of -2^{n-1}, and all other bits make up a number that is to be added to this negative weight. Obviously, in the case of positive integers, the weight gets multiplied by 0 (+ sign) and therefore the number is represented essentially the same as unsigned integers but using 1 less bit. One example, represented on 4 bits:

-2^3	2^2	2^1	2^0	
1	0	1	1	
-8	+0	+2	+1	= -5

The general formula for the value w of an N-bit integer can be more formally defined as follows:

$$w = -a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i$$

Using Two's Complement we can perform addition and subtraction easily. Addition can be performed as in unsigned representations, but we must pay attention to overflow. For example, if we add two numbers and the result is out of the range of our representation there will be an overflow and the result will be incorrect. Example: 0111 (7 in decimal) added to 0001 (1 in decimal) will result in 1000 (-8). Attention must be paid to the ranges of the operands in order to avoid this.

Addition can be performed in Two's Complement just as it would be in an unsigned representation. We only have to take in consideration that the MSB will always be the sign and we want to perform operations that can be represented in the range provided by our number of bits.

Subtraction can be performed using the same unit as addition, since in Two's Complement, we can obtain the negative version of a positive number by flipping all the bits and then adding 1. If we want to perform A – B we can just perform A + NOT(B) + 1 instead. [1]

Multiplication can be performed simply with shift and add. For division, an algorithm that can be used is one that successively subtracts the divisor from the dividend, much like the multiplication works, taking into account of course, the possible remainder. [3]

Sign extension is a procedure by which we can accommodate any number represented in Two's Complement by simply copying the most significant bit into all the higher order bits. 1011 is -5 in Two's Complement. Performing sign extension to represent this on 8 bits would mean writing: 11111011.

For the three logical operations, they will be performed *bitwise*:

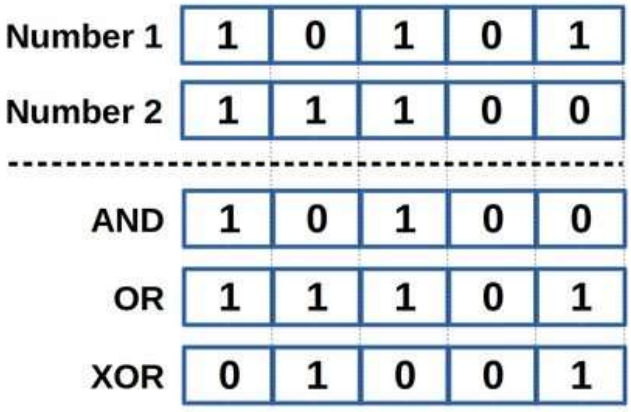


Figure 2 Bitwise logical operations [2]

For the *rotate* operations, we have to consider this functionality: bits to the left are fed into the right end of the register.

3. Project proposal and plan

For implementing this ALU, my proposal is to separate the operations by their necessities and try to group the operations that are related together. Addition and subtraction can, as previously discussed, be computed by the same 32-bit Full-Adder. Only the second operand is chosen as itself (B) in the case of addition and (-B) in the case of subtraction, by a control bit. Logical operations will also be grouped together and have common control. Rotation can be achieved by regrouping the order of the bits in the input, both for rotate-left and rotate-right, these will also have a control bit for choosing the type of rotation. Finally, multiplication and division are a separate unit. All these results will be joined together in a final multiplexer that chooses the final result according to another 2 control bits. The entire ALU will have an internal structure that is largely described by the diagram:

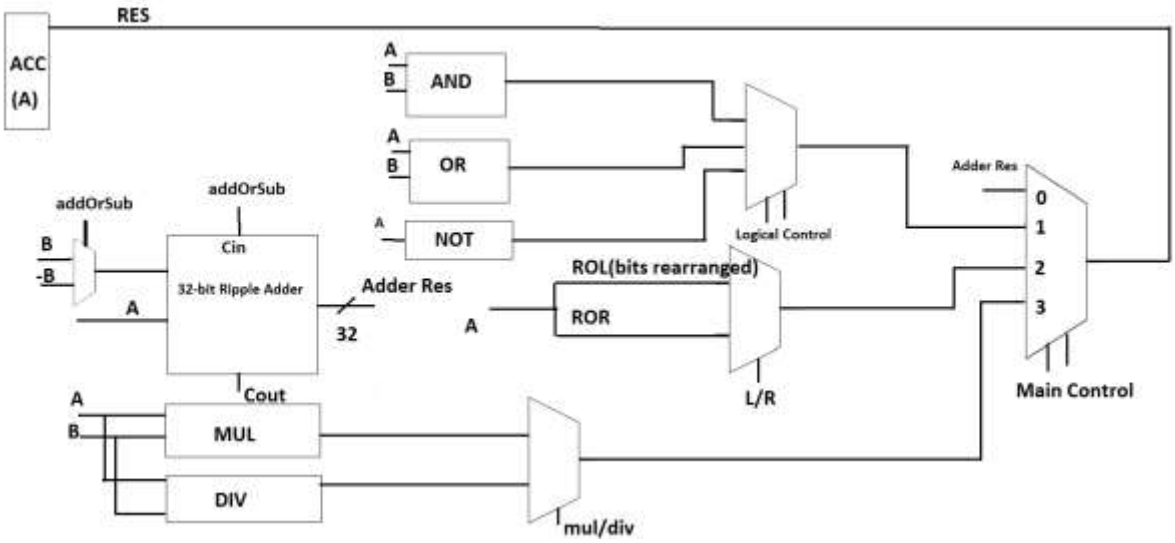


Figure 3 Internal Structure of the ALU

In the configuration in the diagram, there's a total of 6 control bits. The Main Control bits split the ALU into four: the Add/Subtract Unit (0), the Logical Unit (1), Rotation (2) and Multiplication/Division (3).

For implementation purposes, each of the 4 can be implemented separately and tested once finished. The plan for describing the ALU in VHDL is as follows:

1. 32-bit Full Adder used for signed addition/subtraction and testing.

2. 3 logical operations performed on 32 bits and testing.

3. Rotations left and right on operand A and testing.

4. Multiplication and Division Unit and testing for each of them.

4. Analysis

4.1 Multiplication

Multiplying in binary is similar to multiplying in decimal, however we need to perform a lot of shifting, since we have more digits. Using Shift-and-Add multiplication, a multiplication by N, for example, means multiplying each of the individual N bits of the multiplier with the multiplicand, and then shifting the multiplicand left by 1 bit to account for the correct weight of the digit that we are multiplying it by (2nd LSB of the multiplier has weight 2^1 so we shift left by 1). Each intermediate result is stored in an accumulator which is both the destination and also one of the two operands of the adder that will add all the partial results of the operation. After N shifts to the left and sums, the final product will be ready in the accumulator.

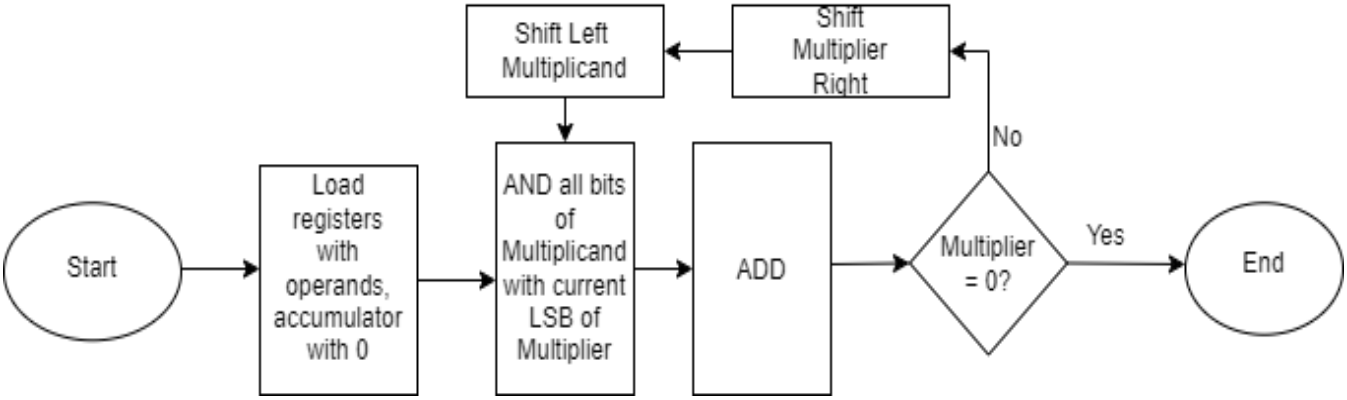


Figure 4 Flowchart for the multiplication algorithm

To achieve the implementation of this algorithm, 3 control bits are needed to coordinate the shifting of the operands and for adding the partial results. These are: Add, Shift Left (Multiplicand), Shift Right (Multiplier). If the operands are represented on 32 bits then the result will be on 2*32 = 64 bits and the Multiplicand register will also need to be a 64-bit register.

4.2 Division

The division operation is the only operation performed by the ALU which will produce 2 outputs: the Quotient (Q) and the Remainder (R). The algorithm for division, similarly with the algorithm for multiplication, subtracts from the partial Remainder (R), only if the divisor still "fits" into the partial Remainder, i.e. only if the divisor (Y) is: Y <= R which results in a quotient digit of 1, otherwise 0. Again, in order to achieve this we need some control signals: 1 for the Quotient, 1 for the Divisor, 1 for Add/Subtract and 1 for Write in the partial remainder register.

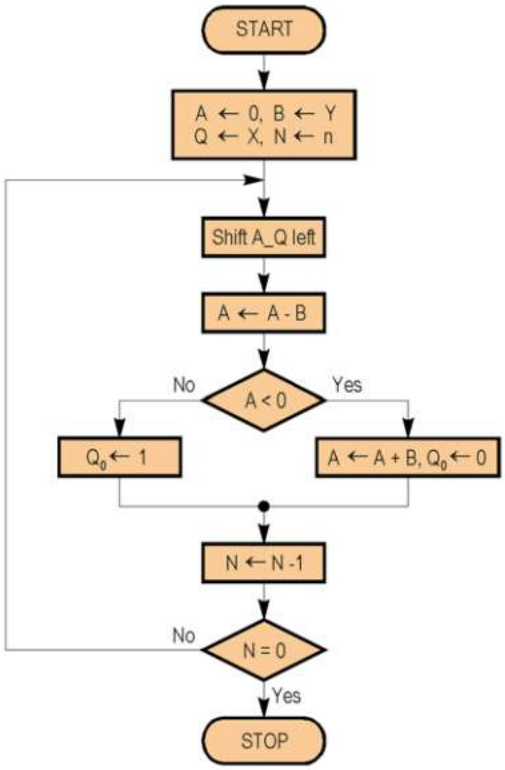


Figure 5 Flowchart for Division [5]

5. Design

As proposed in the project plan, the design of the entire ALU follows the separate functionalities of the unit separately, combining the hardware and reusing it for some functions in order to obtain the final design. For designing the 32-bit Ripple Adder, which will be at the basis of the addition and subtraction functionalities, we can first consider a 1-bit Full-Adder:

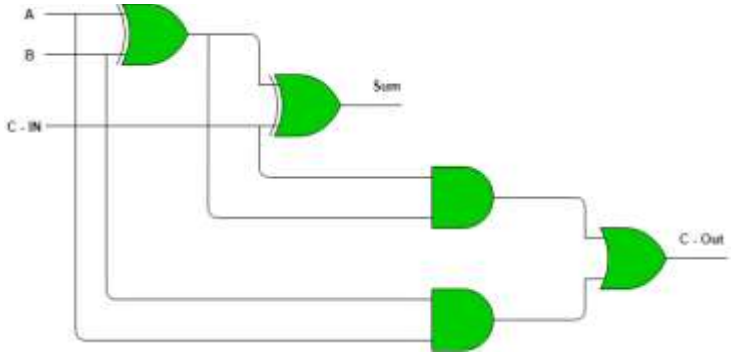


Figure 6 Internal Logic of a Full Adder [4]

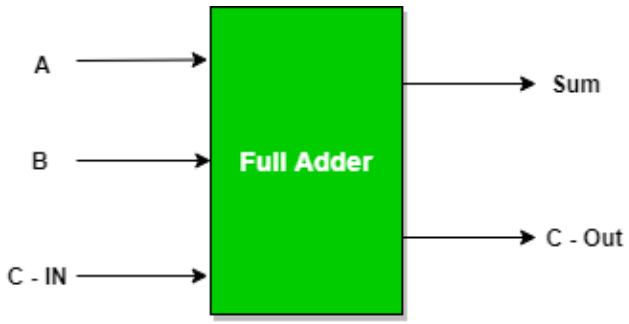


Figure 7 Full Adder [4]

Using this basic block we can build a 32-bit Ripple Carry Adder by "cascading" 32 of these 1-bit Full Adders and use the same logic for the subtraction functionality as previously described, by setting the second operand to its negative counterpart using additional control signals.

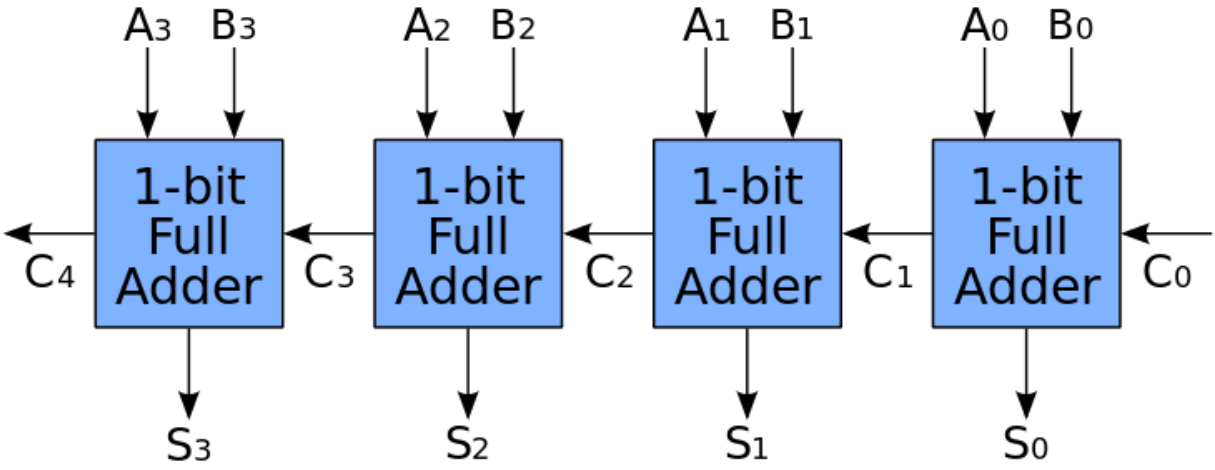


Figure 8 Cascading of 4 Full Adders.

The entire Add/Subtract unit can be thus controlled using the addOrSub control signal. When it is set ('1'), it will send the negated version of B to the input of the Ripple Carry Adder and set its Carry In to '1', therefore performing subtraction in Two's Complement Representation.

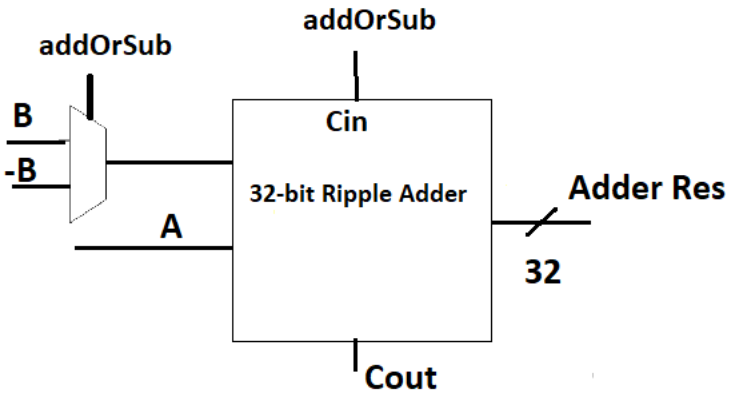


Figure 9 Add/Subtract Unit

Next, for designing the rotation functionalities, we'll separate them in a another block where the bits of A get shifted (rotated) correspondingly. We want to shift in both directions, so using a control signal "left_OR_right" we can do so:

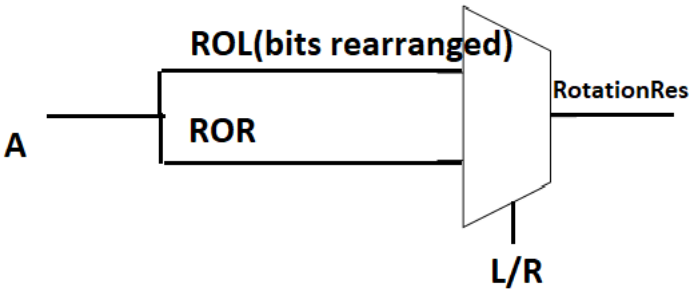


Figure 10 Unit for rotation operations

For the Logical Unit, we need a design for three operations(more can be added in this design): ADD, OR, NOT. Again, using a 2-bit control signal we can easily get a design:

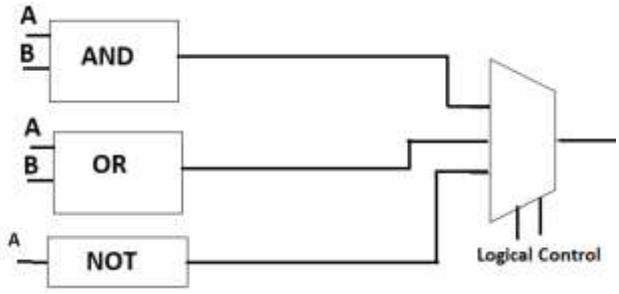
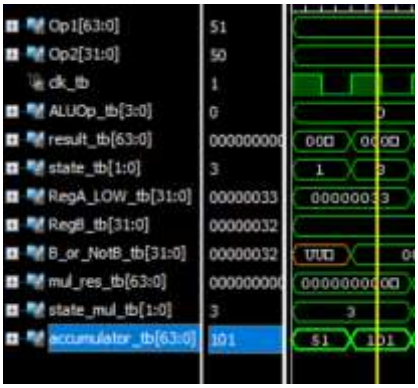


Figure 11 Unit for logical operations

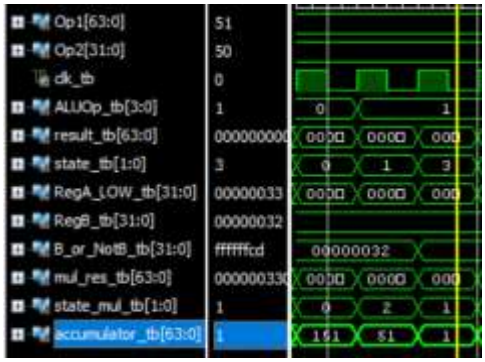
6. Testing and validation

For testing purposes, multiple combinations of inputs for each of the operations were tested and here is the complete results of a testbench that tests each of the operations one by one. We expect after each complete computation of a result to get an updated value in the accumulator.

First, simple addition: (ALUOp = 0)

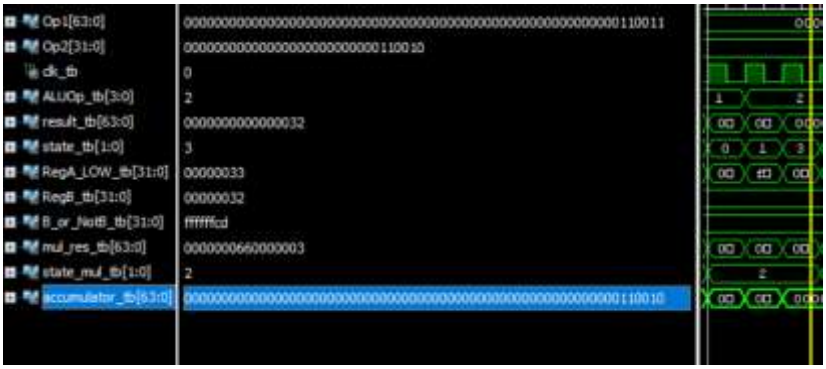


Then, subtraction: (ALUOp = 1)



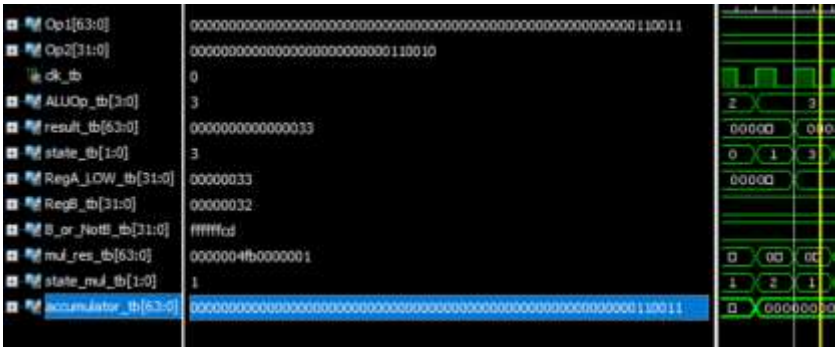
Logical AND: (ALUOp = 2)

0011 0011 AND 0011 0010 = 0011 0010



Logical OR: (ALUOp = 3)

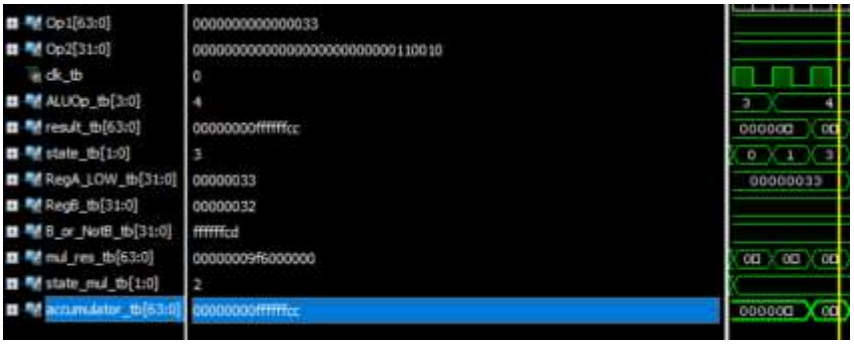
0011 0011 OR 0011 0010 = 0011 0011



Logical NOT: (ALUOp = 4)

Op1 = HEX 00000033

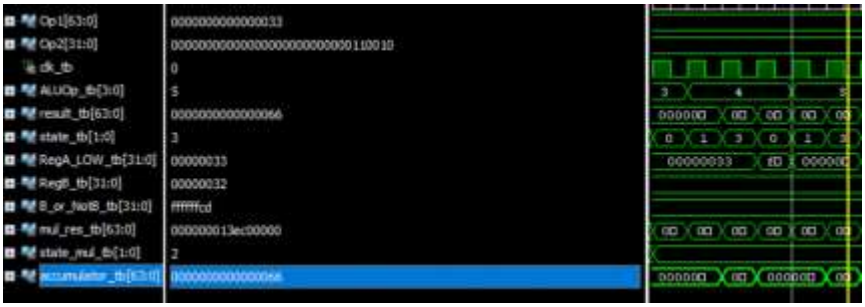
NOT (HEX 00000033) = FFFFFFCC



Rotate Left: (ALUOp = 5)

Op1 = 0011 0011

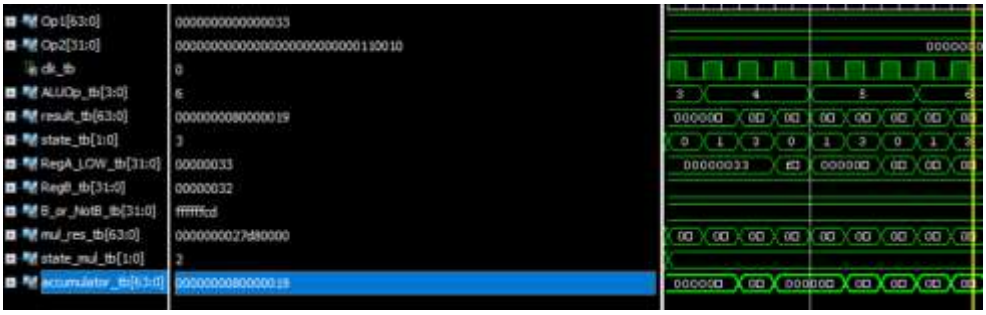
ROL (0011 0011) = 0110 0110



Rotate Right: (ALUOp = 6)

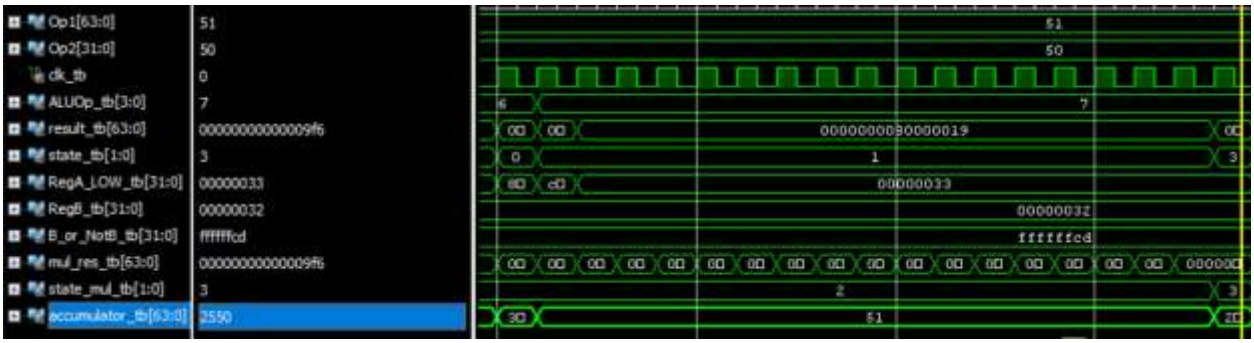
Op1 = 0000 0000 0000 0000 0000 0000 0011 0011

ROR (Op1) = 1000 0000 0000 0000 0000 0000 0001 1001 = HEX 80000019



Multiplication: (ALUOp = 7)
Op1 = 51, Op2 = 50

Op1 x Op2 = 2550



7. Conclusions

As a conclusion, the ALU manages to function properly using the accumulator. Because of the functionality of the accumulator register and because of the complexity of the mul/div unit there was the need for Finite State Machines, however after testing it is clear that they are functioning together well and different types of operations can be run one after the other.

8. Bibliography

[1] David Money Harris, Sarah L. Harris, in Digital Design and Computer Architecture (Second Edition), 2013, 5.2.2

[2] Philip Maus, “Bitwise calculator” [Online]. Available:

https://www.omnicalculator.com/math/bitwise

[3] Design of ALU Components, Laboratory work no. 4

[4] GeeksForGeeks, “Full Adder in Digital Logic” [Online]. Available:

https://www.geeksforgeeks.org/full-adder-in-digital-logic/

[5] Z. Baruch, “Structure of Computer Systems”, Fig. 3.21.