**ALU – SEMESTER PROJECT**

Coordinator: *Dr. Eng. Hangan Lia-Anca*

Student: *Samuil Daniela Teodora*

Technical University of Cluj-Napoca, 2020

Contents

1. [INTRODUCTION](#introduction)
2. [BIBLIOGRAPHIC RESEARCH](#bibliographic_research)
3. [REQUIREMENTS & FEATURES](#requirements)
4. [DESIGN](#design)
5. [IMPLEMENTATION](#implementation)
6. [TESTS/EXPERIMENTS](#tests)
7. [CONCLUSION](#conclusion)
8. [BIBLIOGRAPHY](#bibliography)
9. **INTRODUCTION**

The Arithmetic Logic Unit is a pretty well-known component when it comes to hardware, also being quite simple when it comes to its implementation. For this project I will design an ALU that adds, subtracts two numbers represented in 2s complement, increments and decrements such a number, does simple Boolean operations like AND, OR, NOT, rotates a number to the left or to the right, negates a number and also does multiplication and division. For a better understanding of how such a component works, I will do further research using the sources I’m going to attach in the next chapter, trying also to underline the specific parts I’m going to use each research link for. Following the brief description of my project, I will attach a plan that I will try to follow in order to complete this project:

*Weeks 1-4*: Research the topic and gather useful links/books related to the subject

*Weeks 5-6*: Find the requirements for the project

*Weeks 7-10*: Start implementing using VHDL coding language

*Weeks 11-12*: Test the project and eventually solve any bugs

*Week 13:* Expose the conclusions and present the final project

1. **BIBLIOGRAPHIC RESEARCH**

*What is VHDL?*

“VHDL is a powerful language with numerous language constructs that are capable of describing very complex behavior. [..] VHDL Descriptions consist of primary design units and secondary design units. The primary design units are the Entity and the Package. The secondary design units are the Architecture and the Package Body. Secondary design units are always related to a primary design unit. Libraries are collections of primary and secondary design units. A typical design usually contains one or more libraries of design units. “[1]

*What is an ALU?*

“Arithmetic Logic Unit is the part of a computer that performs arithmetic operations on binary numbers. The inputs to an ALU are the data where we have to perform operations. They are called operands. They perform the necessary operation and the result is the output of the operation we have performed. [..] ALU performs Arithmetic and Logical Operations. Arithmetic Operations include Addition, Subtraction, Multiplication, and Division. Logical Operations include operations using AND, OR, and NOT. It does comparison of operations.” [2]

*How can numbers be represented in 2s complement?*

2s complement is the most know way of representing numbers when we talk about computers. The first bit of the representation is the sign bit, which will show if a number is positive (0) or if it is negative (1). If we want to convert from a positive to a negative representation of the same number, all we have to do is negate all the bits and add 1. Same approach goes for the transformation from a negative number to a positive one.

*How are addition and subtraction performed by an ALU for numbers in 2s complement?*

“There are three different cases possible when we add two binary numbers using 2's complement, which is as follows:

Case 1: Addition of the positive number with a negative number when the positive number has a greater magnitude. (Initially find the 2's complement of the given negative number. Sum up with the given positive number. If we get the end-around carry 1 then the number will be a positive number and the carry bit will be discarded and remaining bits are the final result.)

Case 2: Adding of the positive value with a negative value when the negative number has a higher magnitude. (Initially, add the positive value with the 2's complement value of the negative number. Here, no end-around carry is found. So, we take the 2's complement of the result to get the final result.)

Case 3: Addition of two negative numbers (In this case, first, find the 2's complement of both the negative numbers, and then we will add both these complement numbers. In this case, we will always get the end-around carry, which will be added to the LSB, and forgetting it in the final result; we will take the 2's complement of the result.)

These are the following steps to subtract two binary numbers using 2's complement:

In the first step, find the 2's complement of the subtrahend; Add the complement number with the minuend; If we get the carry by adding both the numbers, then we discard this carry and the result is positive else take 2's complement of the result which will be negative.” [3]

*How are multiplication and division performed by an ALU?*

“A simple algorithm for multiplication of signed integers:

– Convert to positive integer any of operands (if needed)

and remember original signs

– Perform multiplication of unsigned numbers using the

existing algorithm and hardware

– Negate product if original signs disagree” [4]

“Divide algorithm

Main ideas:

• Expand both divisor and dividend to twice their size

- Expanded divisor = divisor (half bits, MSB) zeroes (half bits,

LSB)

- Expanded dividend = zeroes (half bits, MSB) dividend (half

bits, LSB)

• At each step, determine if divisor is smaller than dividend

- Subtract the two, look at sign

- If >=0: dividend/divisor>=1, mark this in quotient as “1”

- If negative: divisor larger than dividend; mark this in quotient

as “0”

• Shift divisor right and quotient left to cover next power of two” [5]

1. **REQUIREMENTS & FEATURES**

*Project functionalities*

The Arithmetic Logic Unit I will implement will be able to do simple arithmetical operations like addition, subtraction, multiplication and division on numbers written in 2s complements, as well as logical ones: AND, OR, NOT, circular shift left and right. It will also increment or decrement a given number.

For each one of these functionalities there will be a specific algorithm behind, a simple one like the ones used for addition, subtraction (which is basically addition with the negated number) and the logical operation (the steps for addition and subtraction were briefly presented in the last chapter), as well as more complex ones like the one used for multiplication or division, which basically uses multiple additions/subtractions to deliver the wanted result. Because there are multiple ways to define such a complex algorithm, I will present the method I will try to implement for my project:

*Addition:*

As I presented in the last chapter, there are 3 possible ways of addition, namely: if one operand is positive and one negative with the positive one greater, if one operand is positive and one negative with the negative one greater, and the version where both have the same sign (plus the time when we have a basic addition, when both the numbers are positive). The overall steps needed to be taken were previously presented, but in addition to that I want to add the fact that I chose to use the Carry Lookahead Adder for implementing this operation.

*Multiplication:*

As shown in the laboratory, multiplication can be implemented using multiple Full Adders connected. The Wallace Tree method takes partial products and adds them, then propagates the result to the next stage where is will eventually be added with another result of partial products. This process is repeated until only two operands are left for the addition.

As we can see, in order to have a proper result, the component which needs to be implemented is a Full Adder, having as inputs two numbers and a carry-in and as outputs the result of the addition and a carry-out. The multiplication will be calculated as for the unsigned numbers, the sign being added at the end, depending on the signs of the two numbers to be multiplied.

*Division:*

The division operation is based on multiple subtractions and shifting. Again, because of the multiple different implementations we can use, I will attach a step-by-step of the algorithm I will use for my project (keeping in mind the division is expressed like: dividend=divisor \* quotient + remainder)

-subtract de divisor from the dividend and consider the result as the remainder

-if the remainder is smaller than 0, replace the actual remainder with the sum of the remainder and the divisor and then shift left the quotient and add as the most significant bit a 0

-if the remainder is bigger than 0, then shift left the quotient and add as the most significant bit a 1

-shift the divisor to the right

-repeat the process

As for the multiplication process, we can divide two numbers written in 2s complement as we do for unsigned ones, and only add the sign after, depending on the original signs of the two numbers to be divided.

The logical operations are going to be made bit by bit. For negation, we convert all the bits (0->1 or 1->0) and add 1 at the end. For the circular shifting, we shift all the bits excluding the sign bit, which is preserved.

1. **DESIGN**

As for the design of the project, I thought about using 8 big blocks which will be divided as follows:

-ADD/SUBTRACT

-MULTIPLY

-DIVIDE

-AND

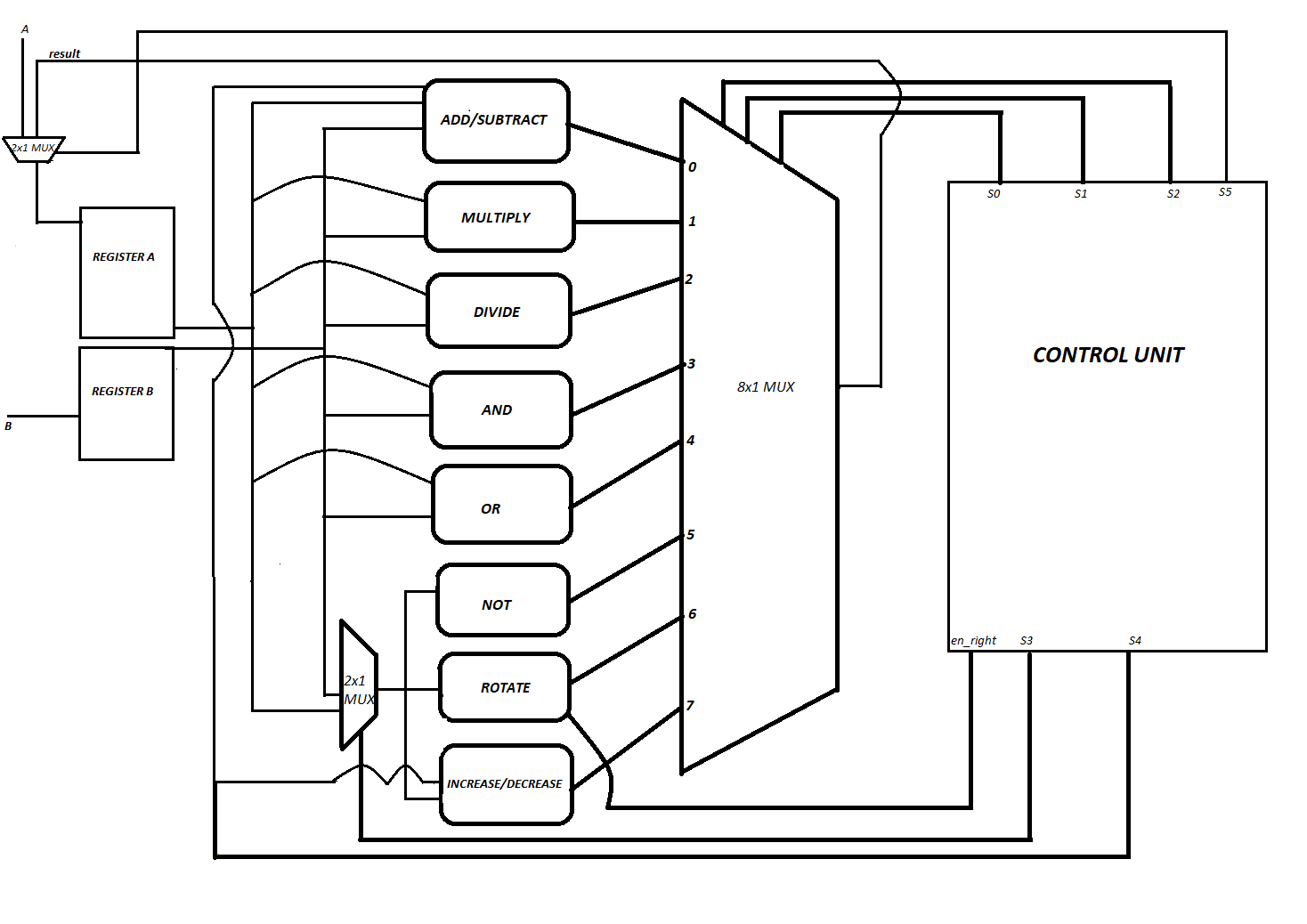
-OR

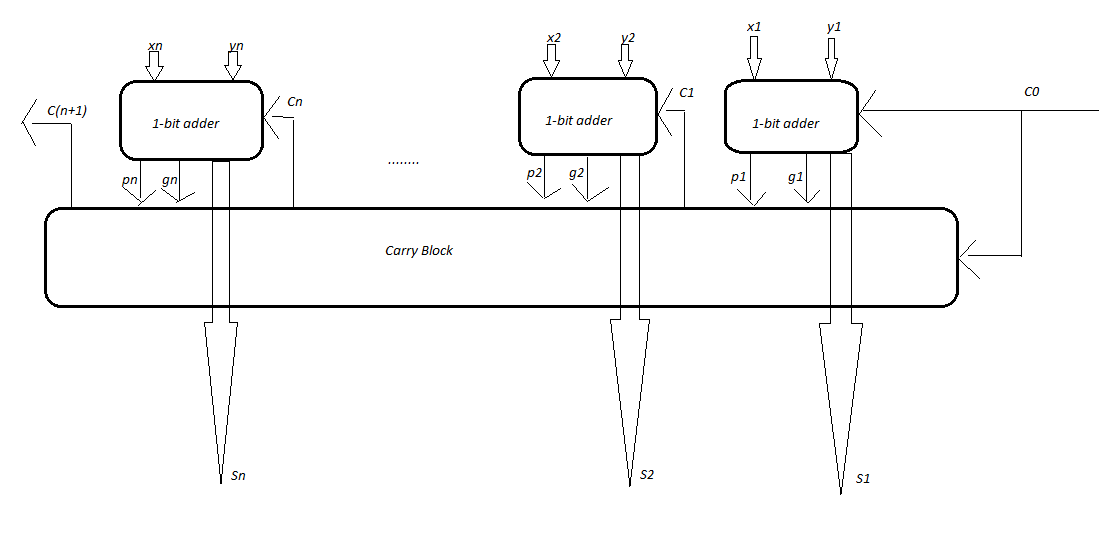
-NOT

-INCREASE/DECREASE

-ROTATE

Some of the blocks, like the logical operational ones or the addition/subtraction will be simpler, but some like the multiplication and division one will be quite complex (meaning that the block itself will contain more components, e.g. Full Adders for multiplication). In order to have a better view of how the components will communicate with one another, I will attach a picture representing the outlines of architecture of the project, noting that besides the blocks specified, there will also be four multiplexors present, one which has the purpose of choosing the operation that is to be made on the given operands, another one which has the purpose of choosing which one of the operands will be further given to the increment/decrement, not or rotate block, and a third and fourth one with the purpose of choosing between addition/subtraction, increase/decrease. Though, besides the two operands there will be 5 more select bits as inputs (3 for choosing the operation, 1 for choosing the right operand for the operations that require just one number as input and 1 for choosing between addition/subtraction and increasing/decreasing the number). Because the big block contains both add/subtract or increase/decrease, the third and fourth multiplexors are not shown in the picture, but they are contained in the ADD/SUBTRACT block, respectively INCREASE/DECREASE block, only the fifth select input being drawn.



Next, I will also attach the hardware diagram for the adder that I’m going to use, specifically the Carry Lookahead Adder:

*User interaction*

The user may be anybody that wants to perform an arithmetical or logical operation on numbers written in 2s complement. The said person will give as input two numbers that will be operated on and 3 bits (S0S1S2) that will help them select what operation they want to perform:

-000: ADD/SUBTRACT

-001: MULTIPLY

-010: DIVIDE

-011: AND

-100: OR

-101: NOT

-110: ROTATE

-111: INCREASE/DECREASE

Then, if all the bits of the select are identical (all 0 or all 1) that means we need to use S4 to select if we want to add/increase or to subtract/decrease. For that the options for S4 will be:

-0: ADD/INCREASE

-1: SUBTRACT/DECREASE

Else, if one of the following selection codes were chosen (101,110 or 111) the fourth selection bit must be chosen. This will tell which one of the operands will be used for the operation. Though, S4 can express:

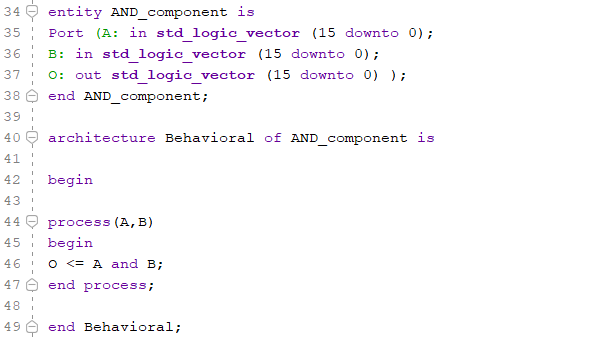
-0: first operand

-1: second operand

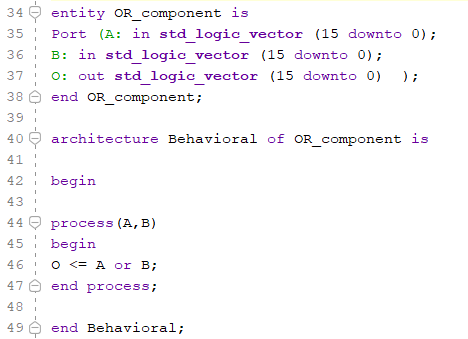
1. **IMPLEMENTATION**

Next, I will attach some screenshots of the implemented parts of the project, starting with the components presented in the design phase. I will begin with the implementation of the logical blocks (AND, OR, NOT, ROTATE), which are all just simple operations, except for the rotate one which I’m going to explain once I get to it.

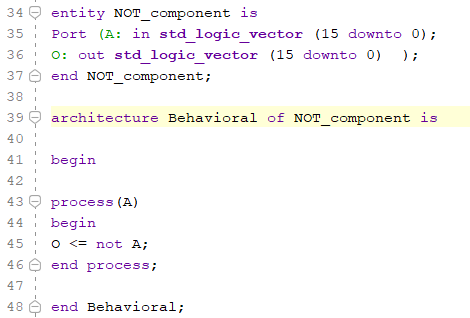
**AND\_component**, implementing the AND logical operation:



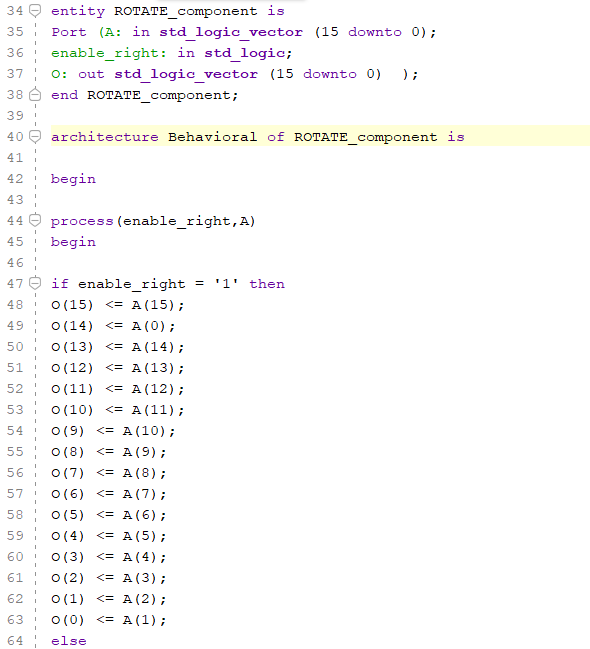
**OR\_component**, implementing the OR logical operation:

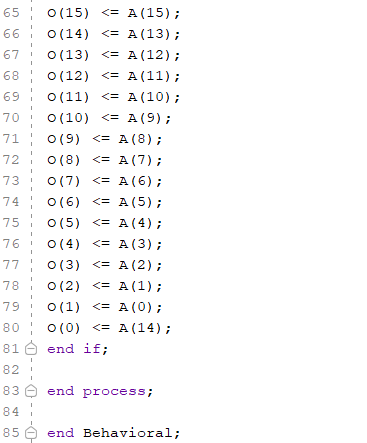


**NOT\_component**, implementing the NOT logical operation:



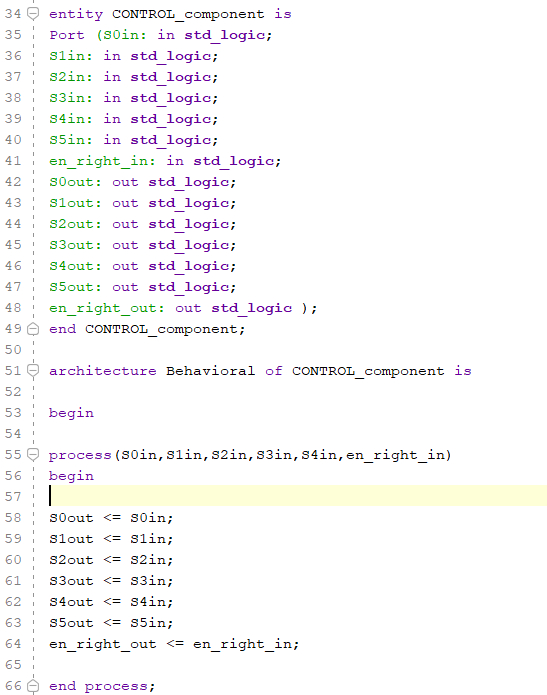
**ROTATE\_component**, implementing the logical left and right notation. Because of the representation in 2s complement, we need to first save the most significant bit, because the sign doesn’t need to be changed, and then shift the bits that are left to the left or to the right, like in a normal case. To choose the direction of the rotation, I will use a signal that I named “en\_right” (as seen in the design diagram as well), which has the following property: if it has the value 1, then the rotation is going to be a right one, else if it has the value 0, the bits will be circular shifted to the left.



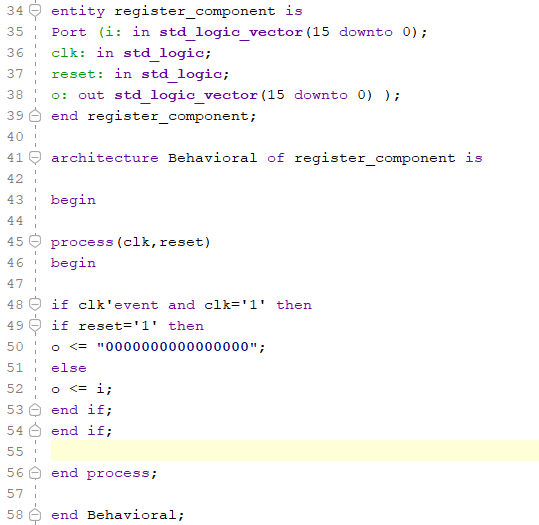


I also implemented the **MUX components** (8x1 and 2x1) that can be seen in the design diagram, but since the code is not that complicated and familiar, I will skip it and go to the part where I defined the control unit for the project and also the registers that will hold the values of the inputs, as well as the values of the output.

**CONTROL\_component**, which is a block that will get as input all the control signals we need (for MUX selection, for choosing if we want to add or to subtract, if we want to increase or decrease, the signal used to decide the direction of the rotation). The component’s functionality will not be a hard one, since it will only take the inputs and redirect them to the corresponding output, but the clear definition of all control signals in a single place is a good thing to have:

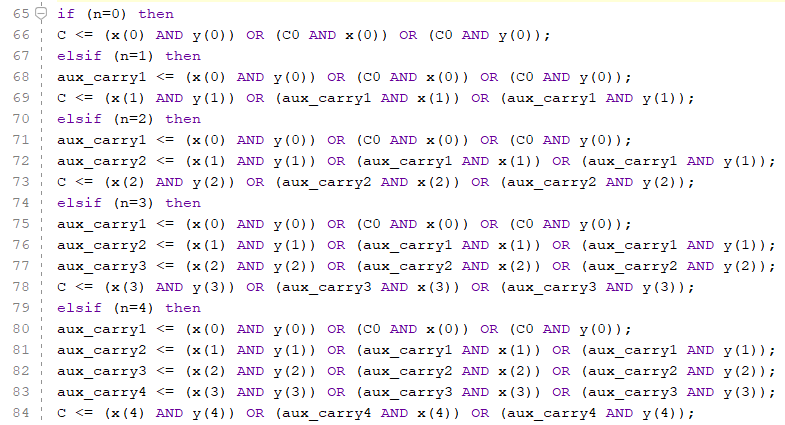


**Register\_component** is the implementation of the registers which will hold the values of inputs and output. It is based on a D flip-flop, working like one, but with the specification that the input of the first register will be the output of a MUX which will choose what to place in the register: either one of the inputs (if the selection signal S5 is 0), or the output of the ALU (if the selection signal S5 is 1). This ensures that the result will be kept in the same existing registers, not needing to add a new one just for that purpose:

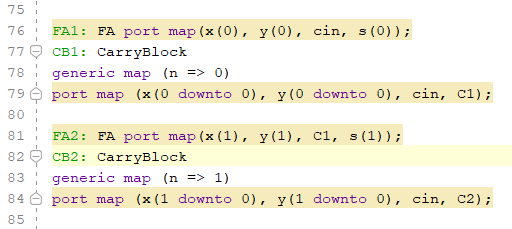


Next, getting to the more complicated components of the project, I will start with addition and subtraction components, then move on to multiplication and division, explaining each one as I go through the code implemented.

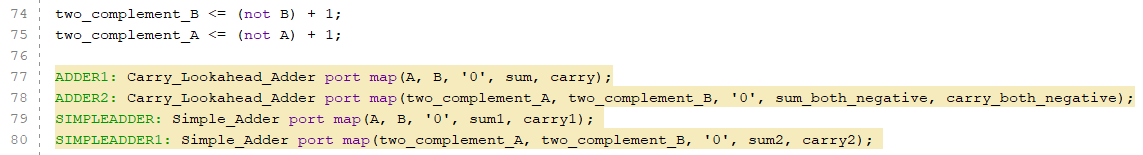
**ADD\_component**, which uses the Carry\_Lookahead\_Adder component (made of Full Adders and Carry\_Block components). First, I will insert some screenshots of the Carry Block design and the Carry Lookahead Adder, the first one calculating the carry bit depending on the bits of the number, while the second one does an addition of the two numbers. For the Carry Block, the sequence presented below repeats until we get to the 16th bit, just like the model:



As for the Carry Lookahead Adder, the sequence below is again repeated constantly until the 16th bit is reached, using both the full adder formula from FA component and the Carry Block component:



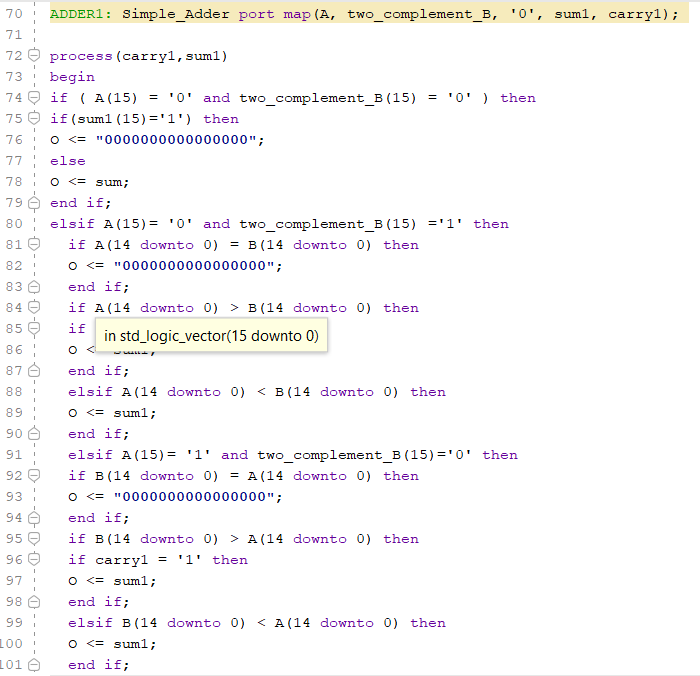
For the whole ADD component, we need to find the 2’s complement of each number, because they will be further needed, as well as some sums, each being used later, depending on the case in which the two numbers are part of. Also, as I proceded to simulate the addition, I found that the carry block had some malfunctions, as the final value of the auxiliary carrys would not be proper calculated when used to find the final carry (but they would calculate fine on their own), so that’s why, for the first 11 bits of the carry block I wrote the full expression without the auxiliary carrys, which worked out fine, but as I got to the last 5 bits, I couldn’t continue with the full expression approach because Vivado got really slow as the number increased exponentially and it would freeze every sequence I wrote and it wouldn’t get the simulation going, that’s why I also used a simple adder, for the cases where the carry for bits 15-11 are ‘1’, because the carry lookahead adder won’t work properly for them because of the bug in the carry block component:



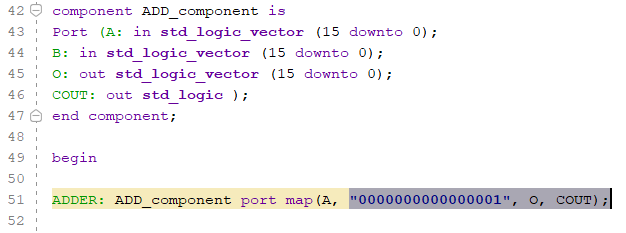
Then, as I mentioned, we need to see which category we are working with: both numbers positive, one positive greater than the negative, one positive smaller than the negative or both negative. As explained earlier in the second chapter, there is a different approach to get the sum depending on the categories, which I will not explain again, but which can be observed in the screenshot below:

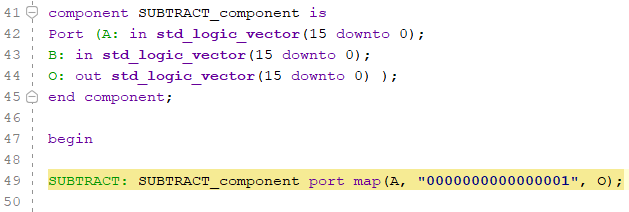


**SUBTRACT\_component** uses the Simple adder also, but first we need to find the two’s complement of the subtracted number (subtraction is just addition with the two’s complement of the number to be subtracted). The decision about the final result is being made just like in the case of a sum, only that now the second operand has to be the two’s complement.

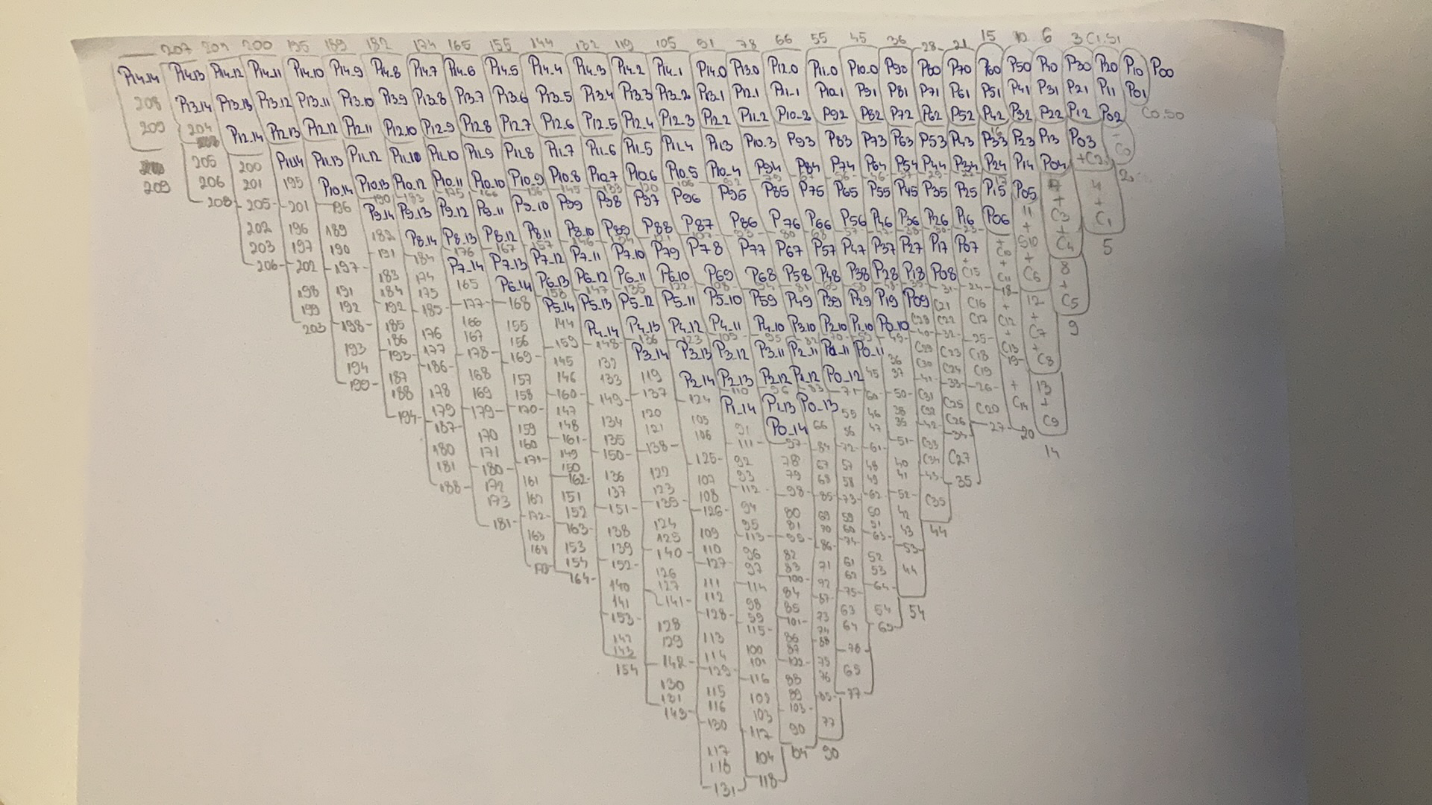


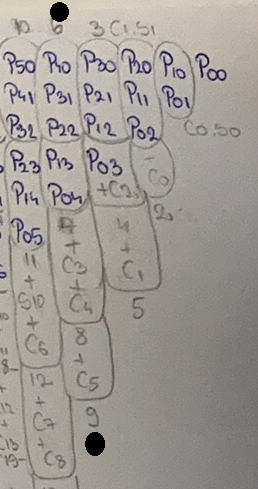
Now that the addition and subtraction are implemented, I will use these components to further implement the **INCREASE\_component** and the **DECREASE\_component**, which are really similar.



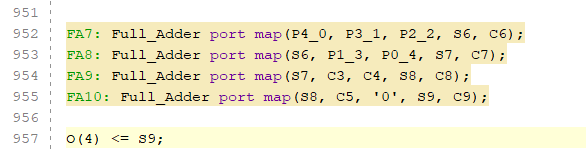


**MULTIPLICATION\_component** is implemented using the Wallace Tree algorithm for 2 numbers on 16 bits, for which I’m going to add the schematic I drew also, because the algorithm gets lengthy because of the number of bits involved. As we can observe in the following schematic, I drew rectangles for the parts that are going to be added together using the Full Adder, as well as the carry signals and partial sums, which I denoted with numbers from 1 to 209.

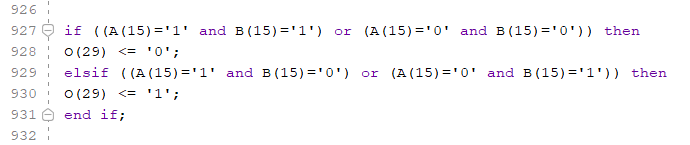




For example, to get the 4th bit of the result we need to deal with the column marked on the left with two black dots. That means that we have a total of 4 add processes, each with 3 operands (A, B and CIN), as seen from the rectangles too. The last three rectangles each have only two operands inside because they take the previous partial sum as the third. C3, C4 and C5 that are added are the carry signals propagated from the previous stage and they all need to be added for an accurate result. Further, I will add some code snippets, related to this example, because the code for the rest of the columns are similar, and the code for the Full Adder used is a simple one, already known. Also, something to be reminded of is the fact that each product (ex Px\_y) is resulted after the operation: A(x) AND B(y), where A and B are the two numbers to be multiplied.

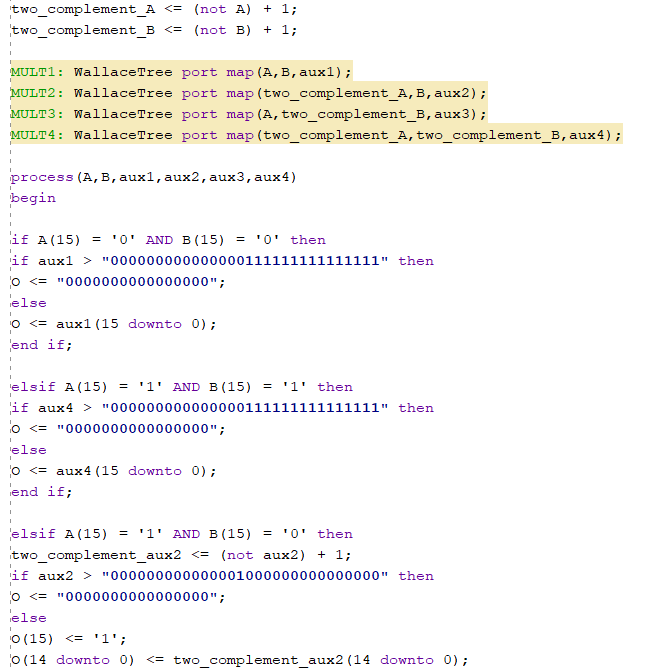


Since we work with signed numbers, something notable is that we only multiply 14 bits, because the first bits from each number are used only to determine the sign of the result, like shown in the following screenshot (if both are either positive or negative, the result will be positive, else the result will be negative):



Besides the Wallace tree implementation, there is the actual multiplier, which works in the following way: depending on which number is negative, we find its complement and use it in the Wallace tree implementation (ex: if both positive, use them as they are; if both negative, use the complements of both). Something to notice is that if one of the is negative (meaning that the result is going to be negative), we need to use the complement of the result of Wallace tree method when we give the final result, since all the result of Wallace tree algorithm will be positive numbers because we use only positive inputs.

Also, because our numbers are on 16 bits, we need to get a 0 if the values extend over the interval -215 , ( 215  - 1).

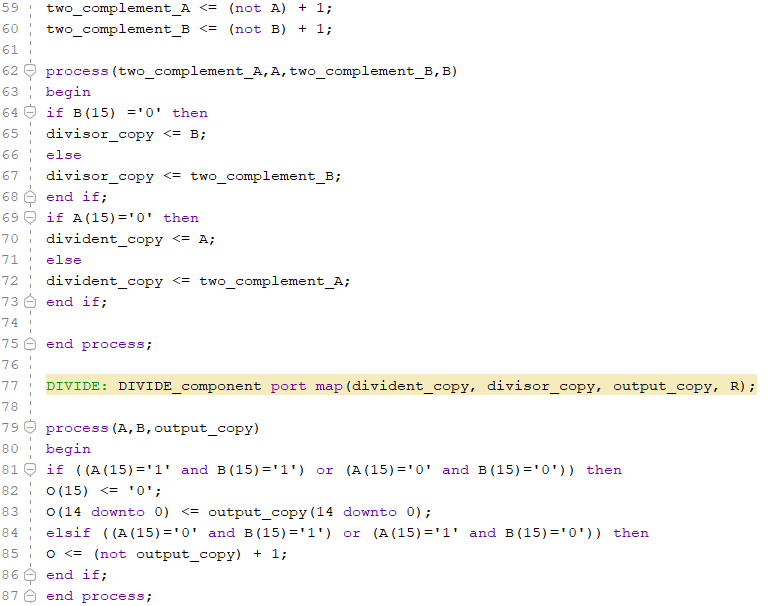


**DIVISION\_component** is implemented using repeated subtractions and shifting phases. I chose to implement the division of two signed numbers in the following way: I designed a division operation that works well with unsigned numbers, which I am going to attach below, and then because we work with numbers written in two’s complement, I decided to perform that operation of my original numbers (if they are both positive) or on their complements, depending on which is the negative one. Also, if one of them is negative, the result of the division for two positive numbers should be complemented, to get the final result. As I mentioned in the first chapters, the sign is going to be determined using both the 15 bit values from the inputs. Also, the case for division with 0 is also treated, in which case the result of both the output and the remainder will be denoted with high impedance.

The division operation working on two positive numbers:



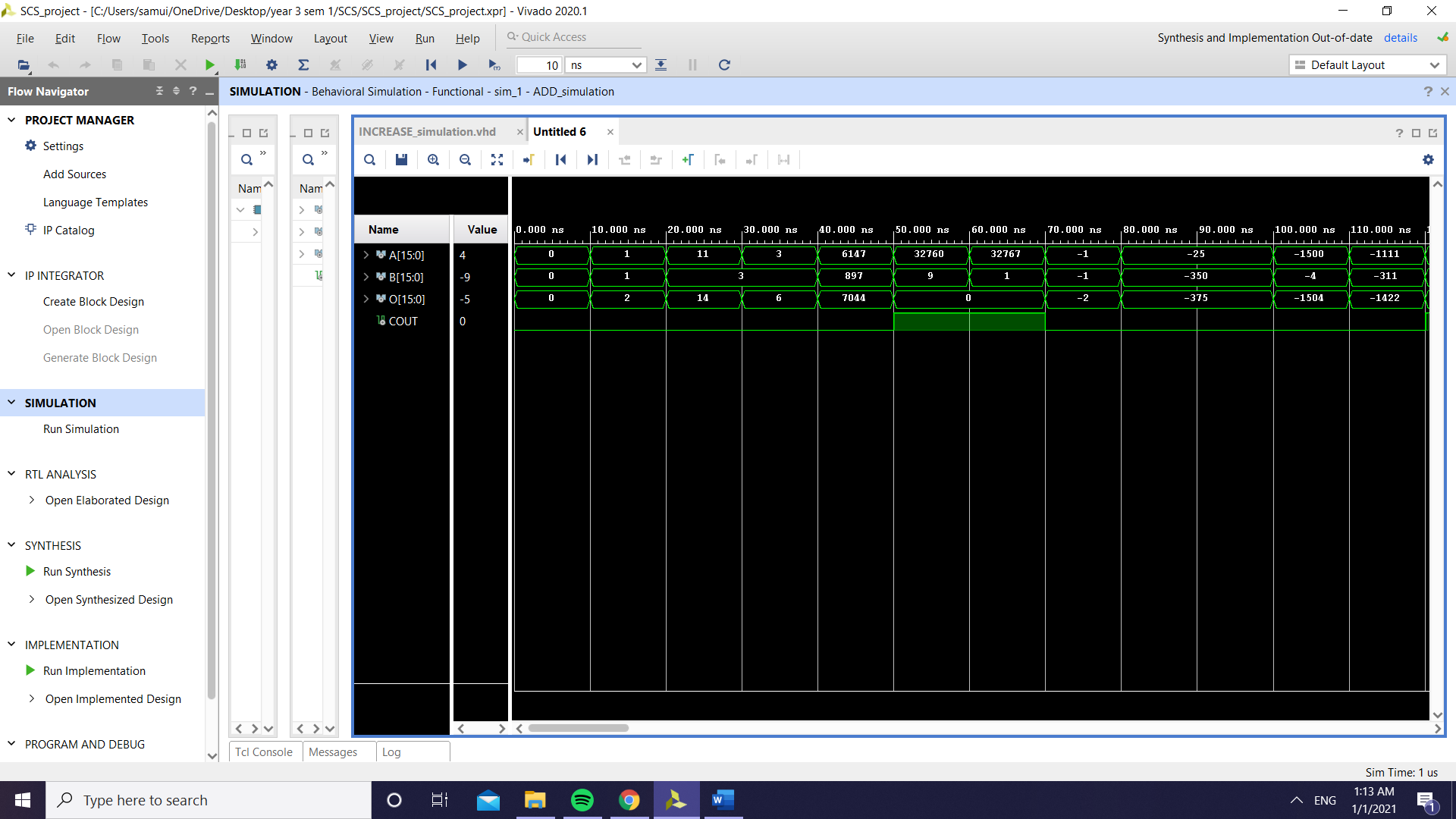
That component being used to define division for signed numbers:

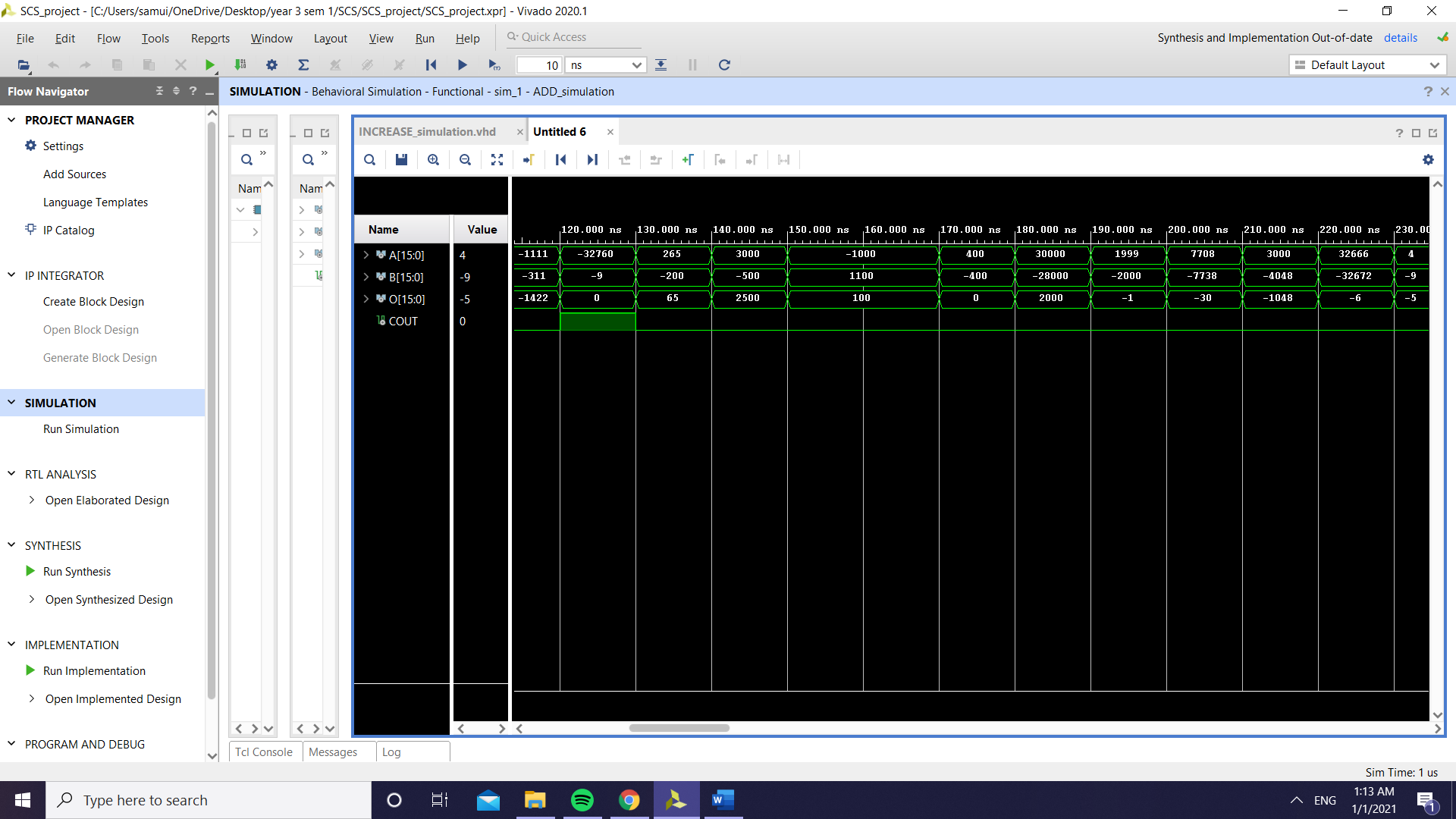


1. **TEST/EXPERIMENTS**

For addition:

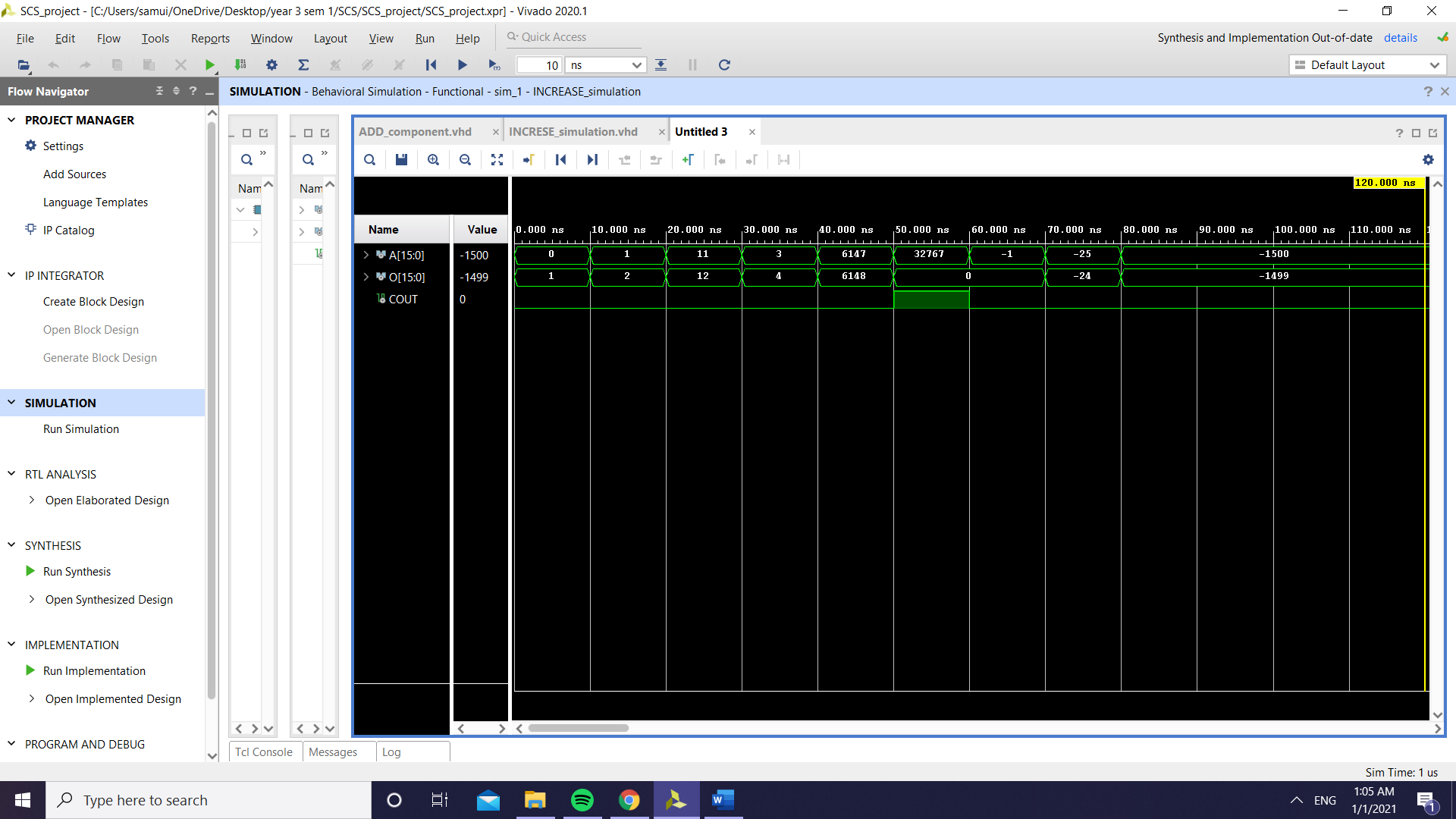
|  |  |  |
| --- | --- | --- |
| *Input* | *Expected output* | *Real output* |
| 3, 3 | O = 6, COUT = 0 | O = 6, COUT = 0 |
| 6147, 897 | O = 7044, COUT = 0 | O = 7044, COUT = 0 |
| 32760, 9 | O = 0, COUT = 1 | O = 0, COUT = 1 |
| -25, -350 | O = -375, COUT = 0 | O = -375, COUT = 0 |
| -1111, -311 | O = -1422, COUT = 0 | O = -1422, COUT = 0 |
| -32760, -9 | O = 0, COUT = 1 | O = 0, COUT = 1 |
| 265, -200 | O = 65, COUT = 0 | O = 65, COUT = 0 |
| 3000, -500 | O = 2500, COUT = 0 | O = 2500, COUT = 0 |
| 400, -400 | O = 0, COUT = 0 | O = 0, COUT = 0 |
| 1999, -2000 | O = -1, COUT = 0 | O = -1, COUT = 0 |
| 7708, -7738 | O = -30, COUT = 0 | O = -30, COUT = 0 |





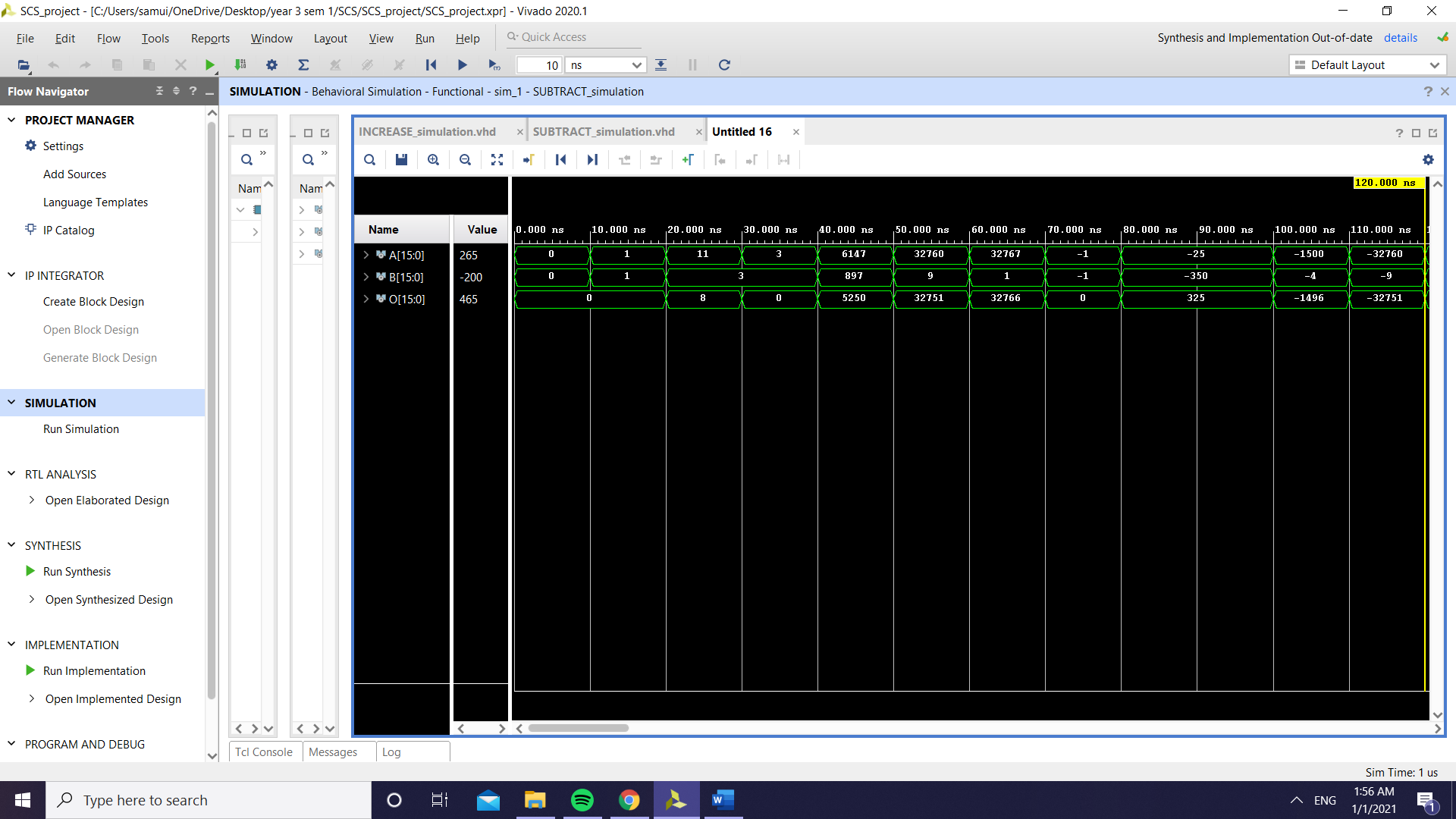
For increasing operation:

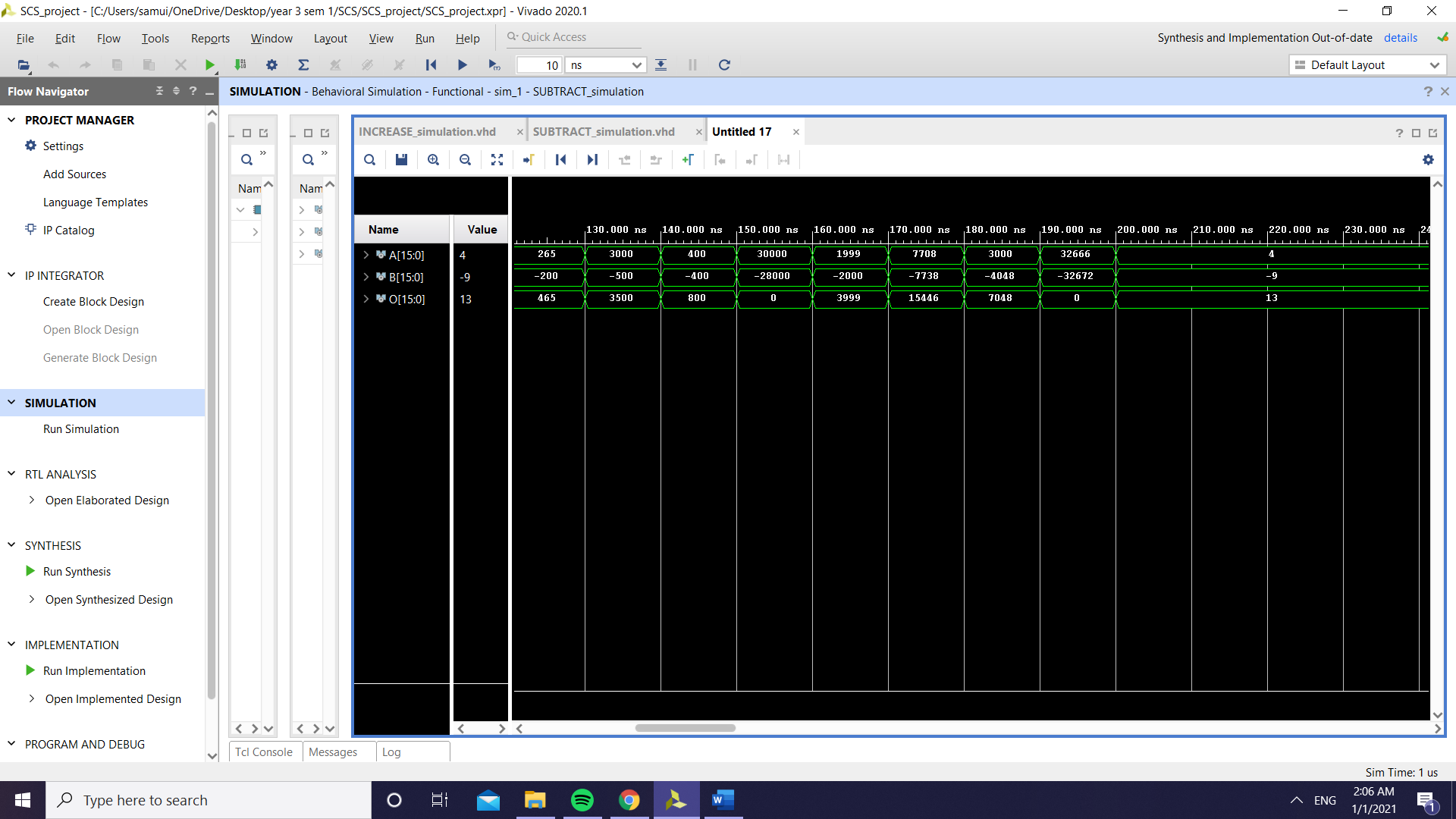
|  |  |  |
| --- | --- | --- |
| *Input* | *Expected output* | *Real output* |
| 0 | O = 1, COUT = 0 | O = 1, COUT = 0 |
| 11 | O = 12, COUT = 0 | O = 12, COUT = 0 |
| 32767 | O = 0, COUT = 1 | O = 0, COUT = 1 |
| -1 | O = 0, COUT = 0 | O = 0, COUT = 0 |
| -25 | O = -24, COUT = 0 | O = -24, COUT = 0 |



For subtraction (using the same values as for addition):

|  |  |  |
| --- | --- | --- |
| *Input* | *Expected output* | *Real output* |
| 3, 3 | O = 0 | O = 0 |
| 6147, 897 | O = 5250 | O = 5250 |
| 32760, 9 | O = 32751 | O = 32751 |
| -25, -350 | O = 325 | O = 325 |
| -32760, -9 | O = -32751 | O = -32751 |
| 265, -200 | O = 465 | O = 465 |
| 3000, -500 | O = 3500 | O = 3500 |
| 400, -400 | O = 800 | O = 800 |
| 1999, -2000 | O = 3999 | O = 3999 |
| 7708, -7738 | O = 15446 | O = 15446 |





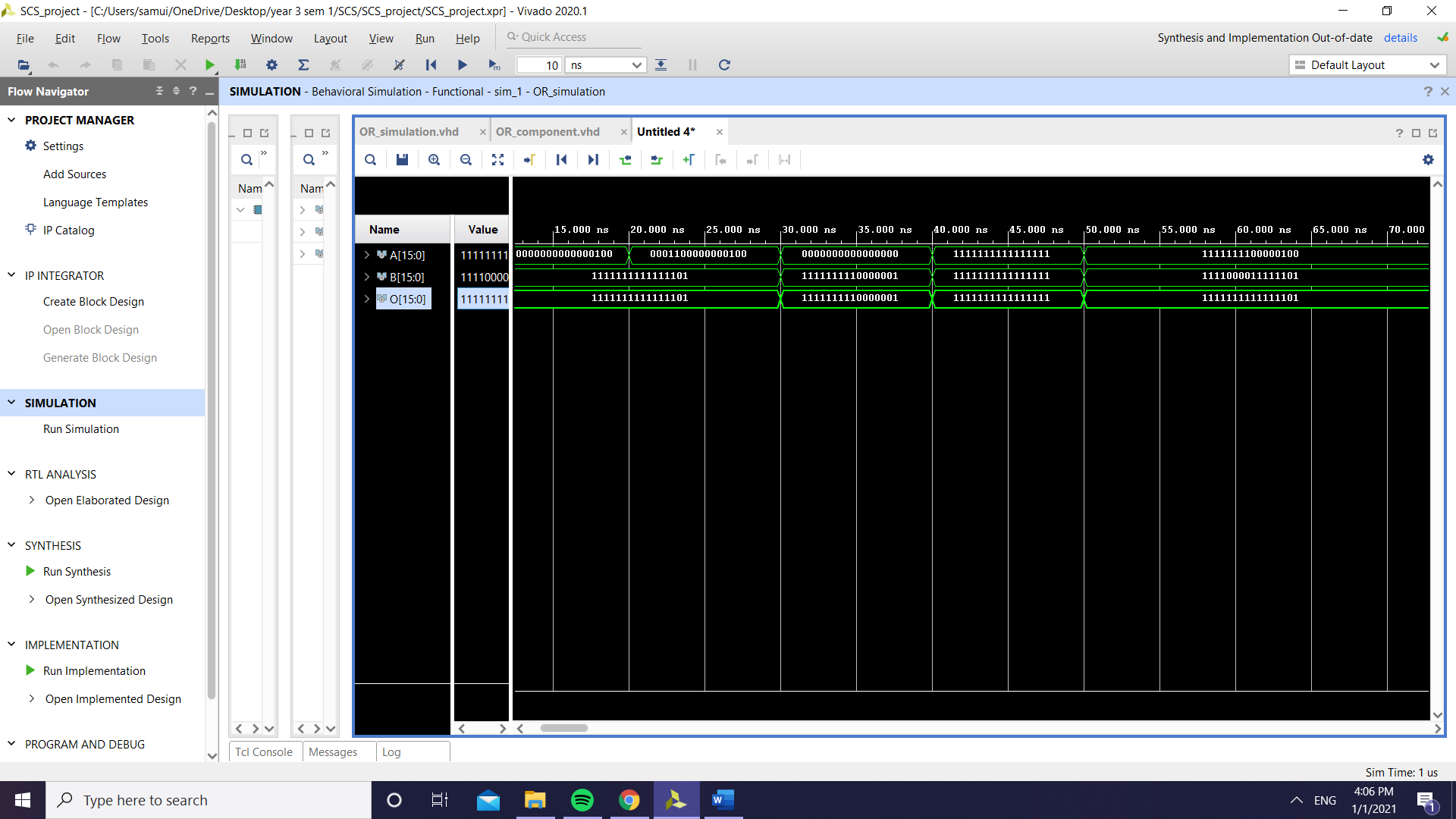
For decreasing operation (same examples used as for increasing operation):

|  |  |  |
| --- | --- | --- |
| *Input* | *Expected output* | *Real output* |
| 0 | O = -1 | O = -1 |
| 11 | O = 10 | O = 10 |
| 32767 | O = 32766 | O = 32766 |
| -1 | O = -2 | O = -2 |
| -25 | O = -26 | O = -26 |



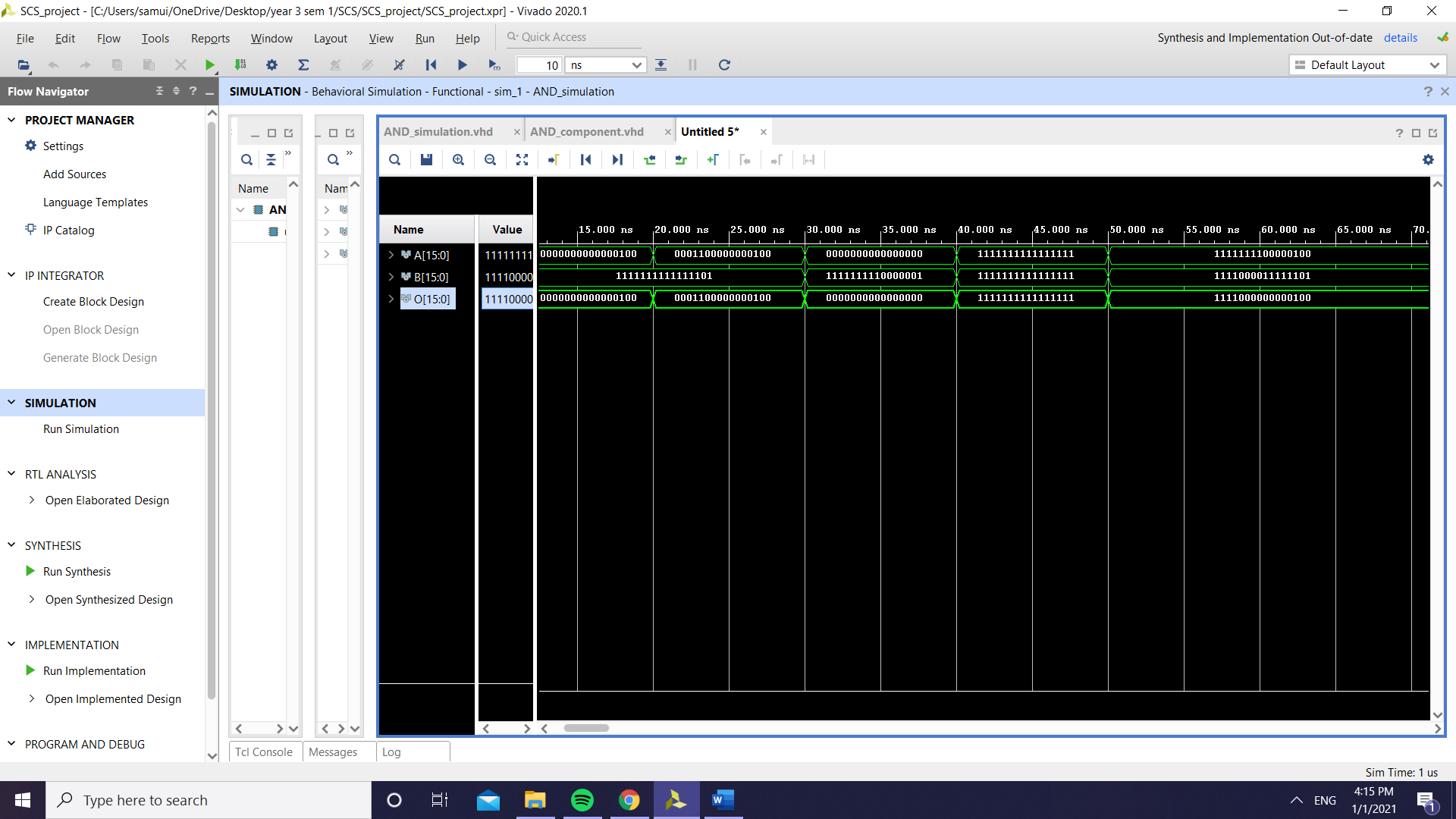
For OR operation:

|  |  |  |
| --- | --- | --- |
| *Input* | *Expected output* | *Real output* |
| 0000000000000100, 1111111111111101 | O = 1111111111111101 | O = 1111111111111101 |
| 0001100000000100, 1111111111111101 | O = 1111111111111101 | O = 1111111111111101 |
| 0000000000000000, 1111111110000001 | O = 1111111110000001 | O = 1111111110000001 |
| 1111111111111111, 1111111111111111 | O = 1111111111111111 | O = 1111111111111111 |
| 1111111100000100, 1111000011111101 | O = 1111111111111101 | O = 1111111111111101 |



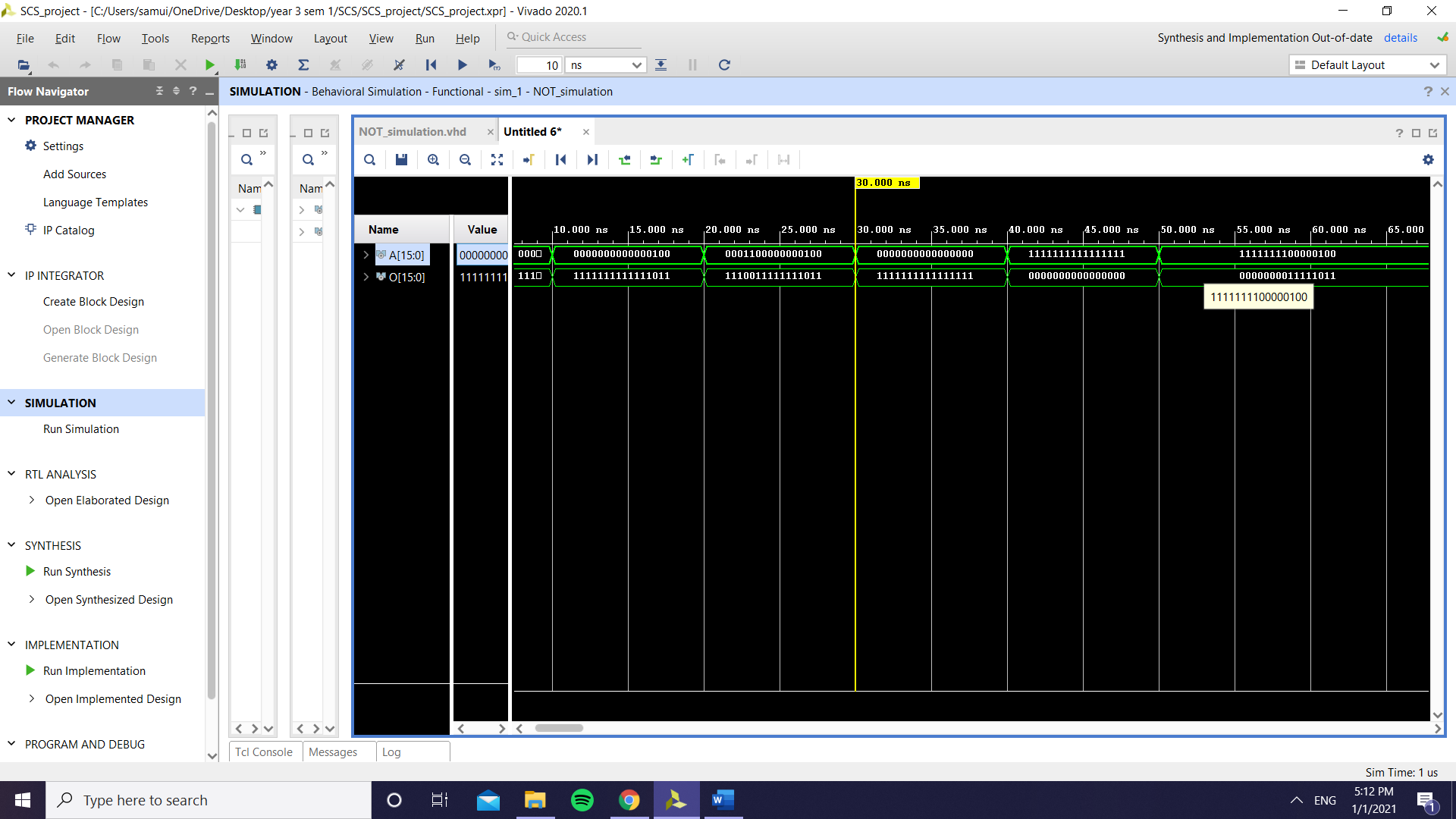
For AND operation (same operands used as for OR):

|  |  |  |
| --- | --- | --- |
| *Input* | *Expected output* | *Real output* |
| 0000000000000100, 1111111111111101 | O = 0000000000000100 | O = 0000000000000100 |
| 0001100000000100, 1111111111111101 | O = 0001100000000100 | O = 0001100000000100 |
| 0000000000000000, 1111111110000001 | O = 0000000000000000 | O = 0000000000000000 |
| 1111111111111111, 1111111111111111 | O = 1111111111111111 | O = 1111111111111111 |
| 1111111100000100, 1111000011111101 | O = 1111000000000100 | O = 1111000000000100 |



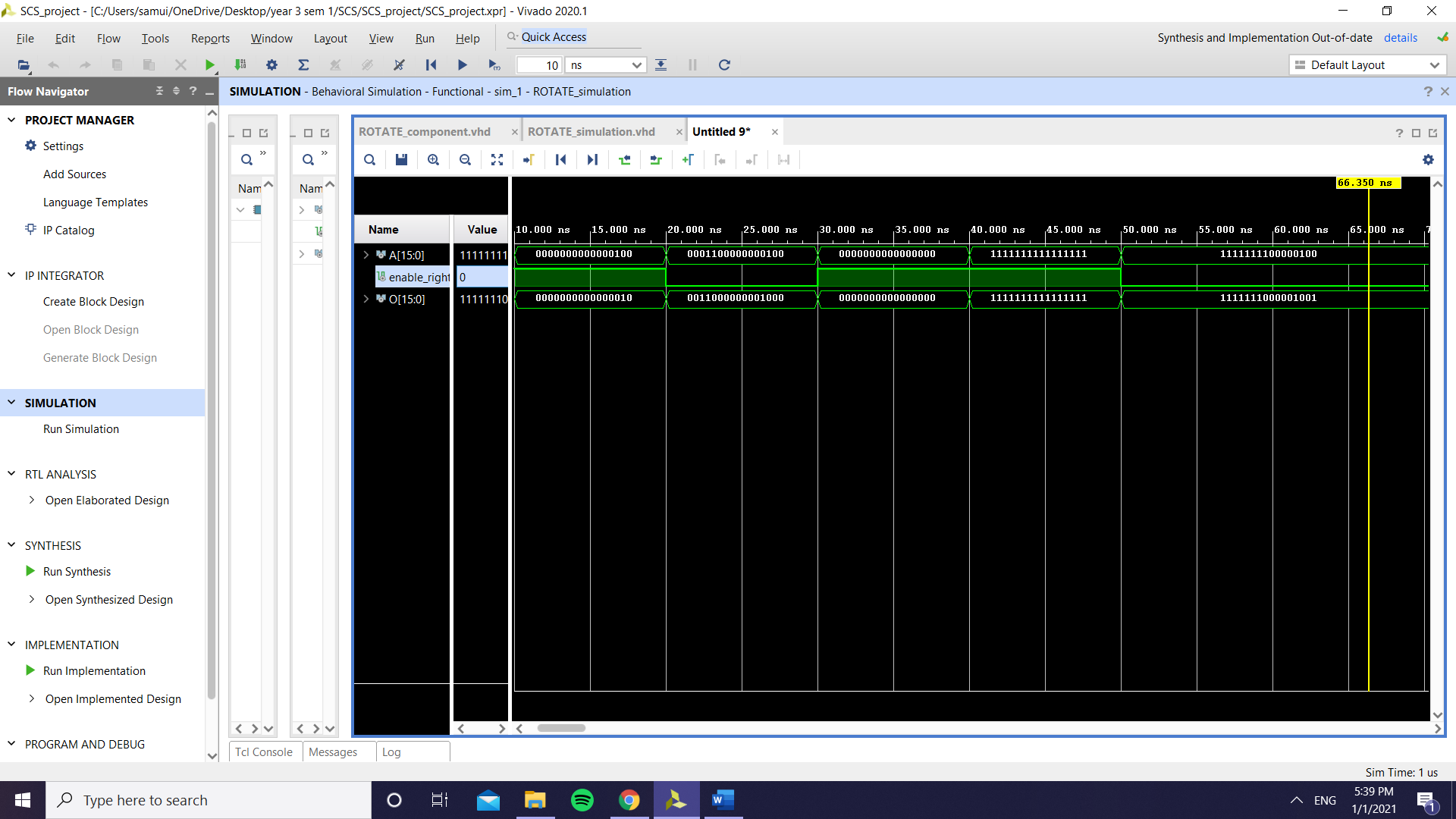
For NOT operation:

|  |  |  |
| --- | --- | --- |
| *Input* | *Expected output* | *Real output* |
| 0000000000000100 | O = 1111111111111011 | O = 1111111111111011 |
| 0001100000000100 | O = 1110011111111011 | O = 1110011111111011 |
| 0000000000000000 | O = 1111111111111111 | O = 1111111111111111 |
| 1111111111111111 | O = 0000000000000000 | O = 0000000000000000 |
| 1111111100000100 | O = 0000000011111011 | O = 0000000011111011 |



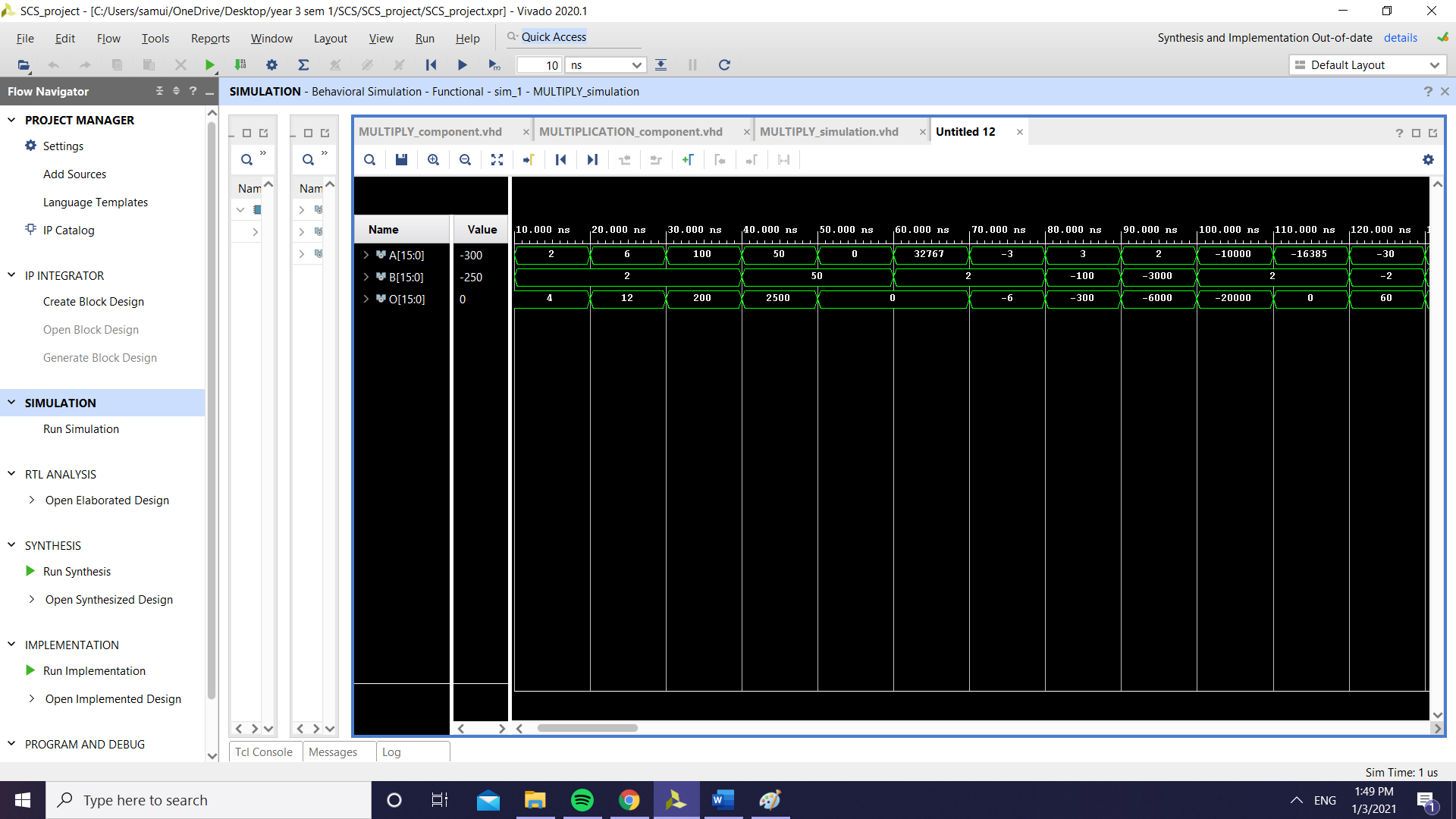
For ROTATE operation:

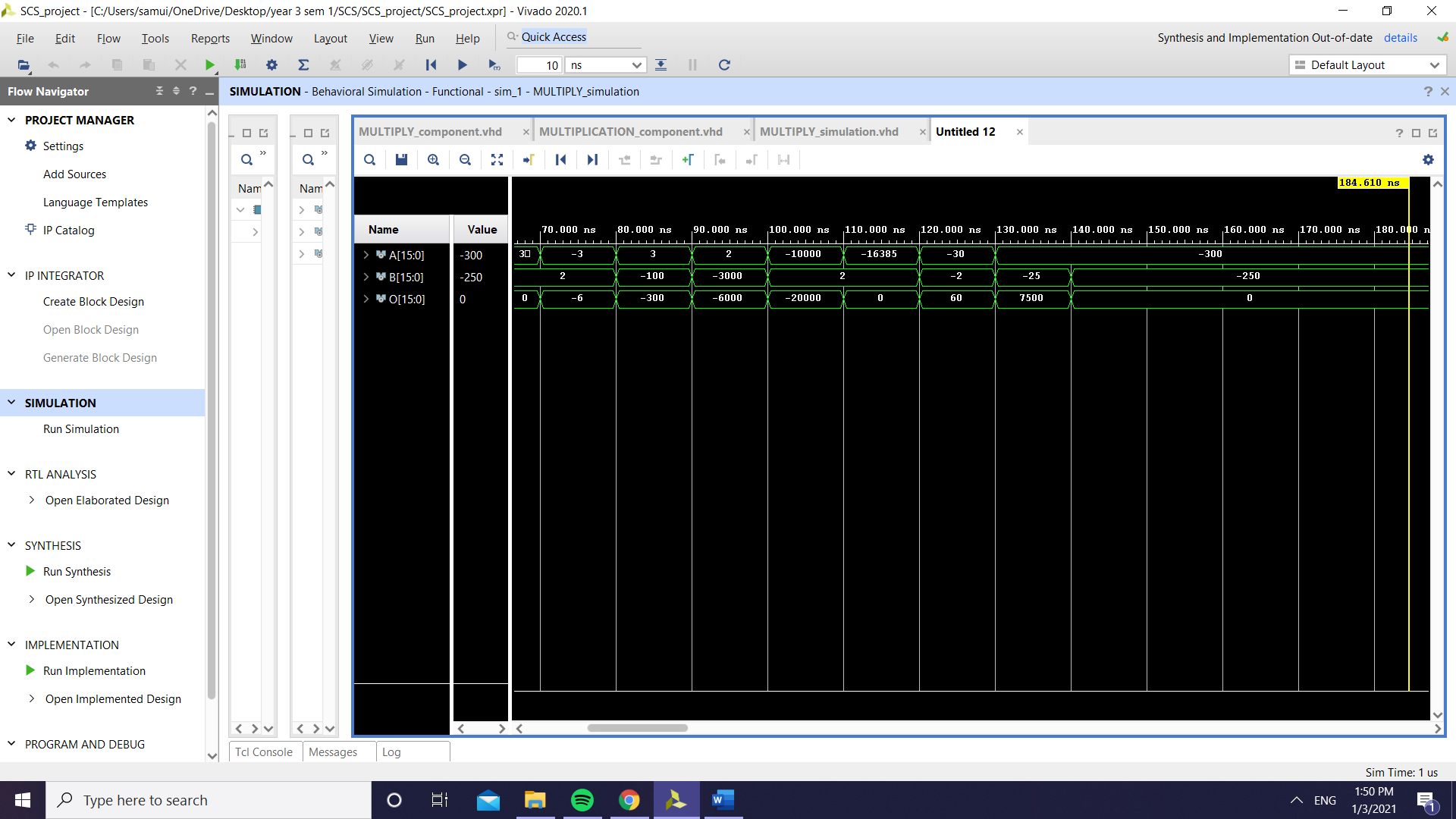
|  |  |  |
| --- | --- | --- |
| *Input* | *Expected output* | *Real output* |
| 0000000000000100, enable right = 1 | O = 0000000000000010 | O = 0000000000000010 |
| 0001100000000100, enable right = 0 | O = 0011000000001000 | O = 0011000000001000 |
| 0000000000000000,  enable right = 1 | O = 0000000000000000 | O = 0000000000000000 |
| 1111111111111111, enable right = 1 | O = 1111111111111111 | O = 1111111111111111 |
| 1111111100000100, enable right = 0 | O = 1111111000001001 | O = 1111111000001001 |



For MULTIPLICATION:

|  |  |  |
| --- | --- | --- |
| *Input* | *Expected output* | *Real output* |
| 2, 2 | O = 4 | O = 4 |
| 100, 2 | O = 200 | O = 200 |
| 50, 50 | O = 2500 | O = 2500 |
| 0, 50 | O = 0 | O = 0 |
| 32762, 2 | O = 0 | O = 0 |
| 3, -100 | O = -300 | O = -300 |
| 2, -3000 | O = -6000 | O = -6000 |
| 2, -16385 | O = 0 | O = 0 |
| -300, -25 | O = 7500 | O = 7500 |
| -300, -250 | O = 0 | O = 0 |

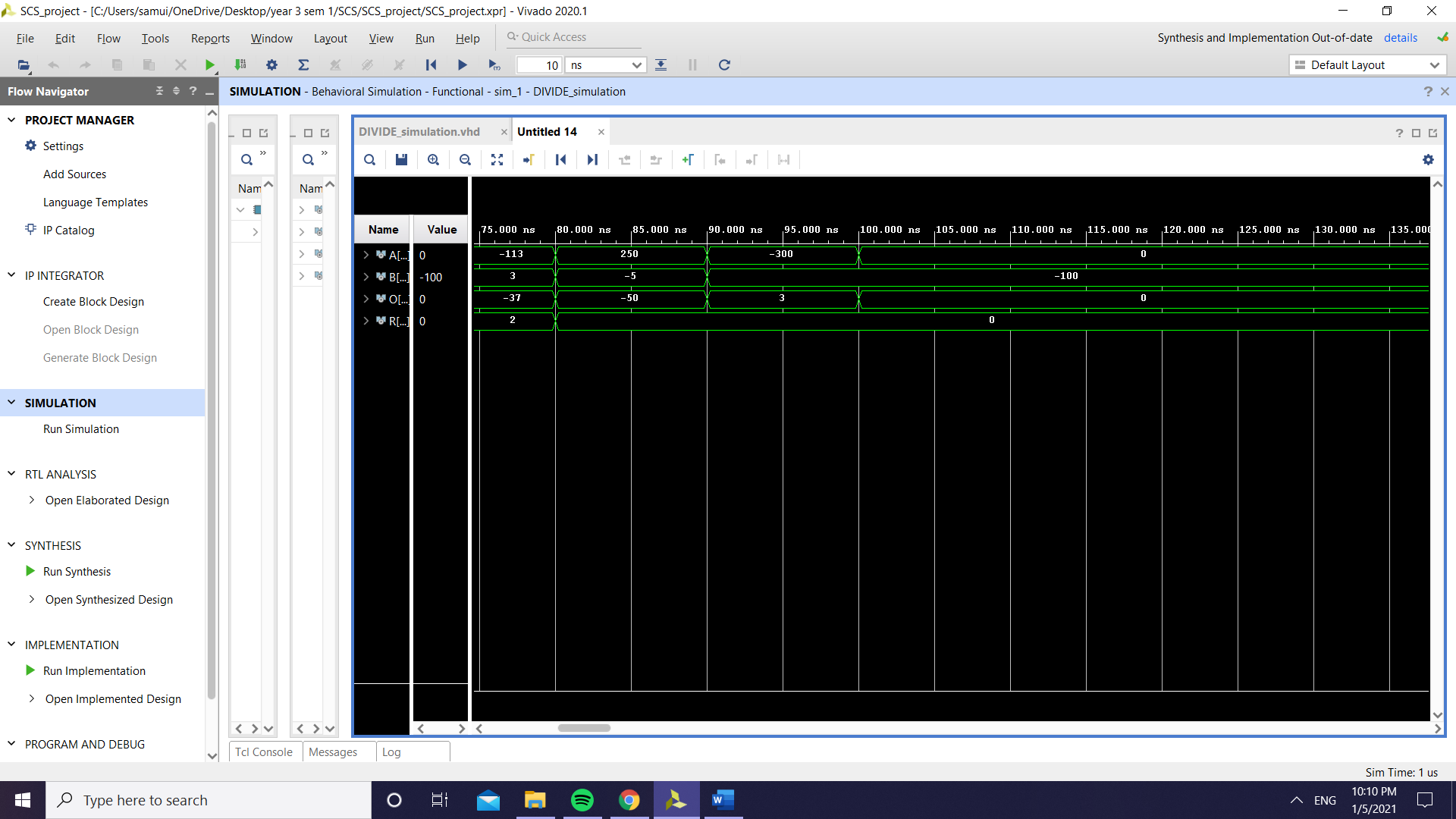




For DIVISION:

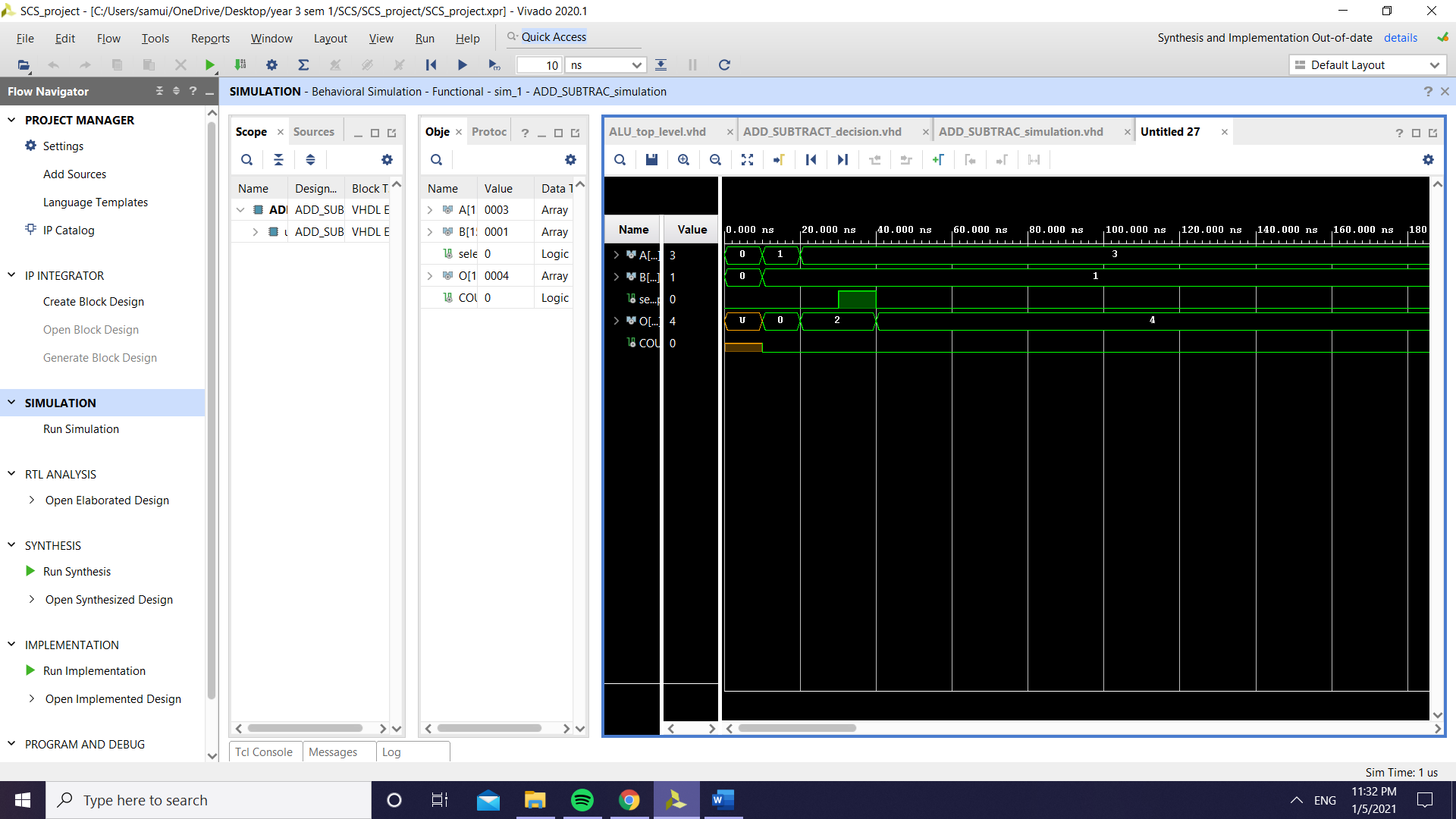
|  |  |  |
| --- | --- | --- |
| *Input* | *Expected output* | *Real output* |
| 2, 0 | O = Z, R = Z | O = Z, R = Z |
| 16000, 100 | O = 160, R =0 | O = 160, R =0 |
| 21, 4 | O = 5, R = 1 | O = 5, R = 1 |
| -16, 4 | O = -4, R = 0 | O = -4, R = 0 |
| -15000, 3 | O = -5000, R = 0 | O = -5000, R = 0 |
| -113, 3 | O = -37, R = 2 | O = -37, R = 2 |
| 250, -5 | O = -50, R = 0 | O = -50, R = 0 |
| -300, -100 | O = -3, R = 0 | O = -3, R = 0 |
| 0, -100 | O = 0, R = 0 | O = 0, R=0 |





1. **CONCLUSION**

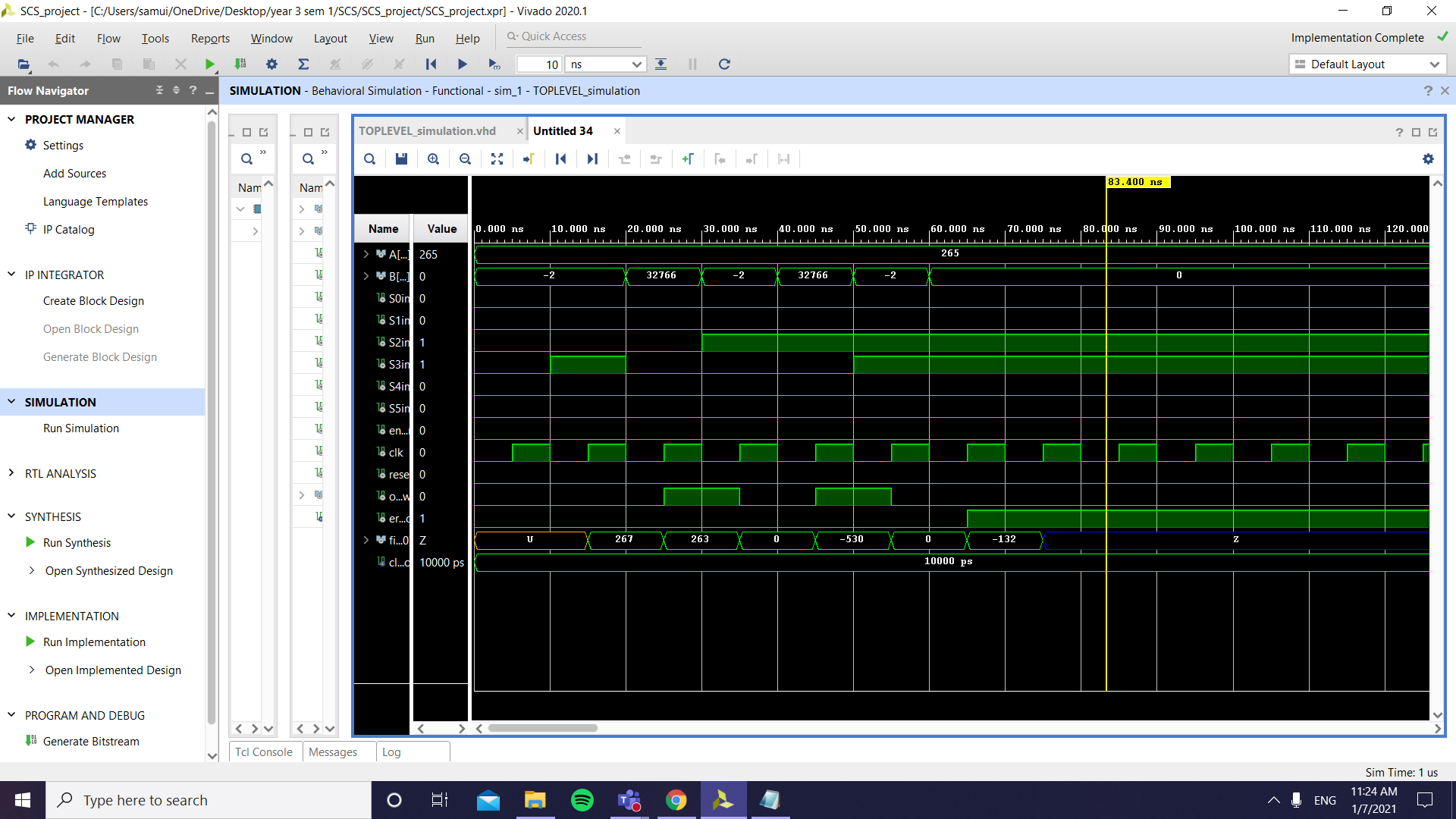
As a remark, for the final implementation I chose not to put addition/subtraction in the same block (same goes for increase/decrease) because of a delay seen when using them, which I couldn’t really figure out how to fix.



Because I thought maybe it would influence the flow of the final simulation, I chose to use separate blocks for the mentioned operations, so S4 in the block diagram would be used for the MUX also, since now it will be a 16 to 1 mux rather than an 8 to 1 (because I will have 10 components since I can’t group them anymore).

Also, something that I didn’t include in the final implementation is storing the result in the same register where the first operand is saved (register A on the block design), because when I wanted to store the result of the ALU there, the values would be damaged and I couldn’t figure out the reason why. Instead, I used a register to store the final result, which is different from the registers where the inputs are stored.

As for the final simulation, there is a delay because of the register used, which again I was not able to solve, but overall, the simulation shows that the program is working as intended to:



Something to note is that the operations are not all seen on the screen, since, for some reason, when the number of operations increased even more than the ones seen, some of the results will be skipped and not shown on the screen, that’s why I chose to put just a few of them in the screenshot(addition, subtraction, addition with overflow, multiplication, multiplication with overflow, division, division with error), and I’m going to also attach to the final project a .txt file which contains all the other tests (and, or, rotate left and right, increase, increase with overflow, decrease) which I’m also going to show at the lab, to proper present my project.

As future improvements, the problem with storing the input and result in the same register depending on an enable signal would be a good idea, as well as maybe finding a way to work with the carry lookahead adder for all of the cases in addition (my carry lookahead adder has some problems with the most significant 5 bits, because if I would replace them with the full expression for calculating them, Vivado would break down).

1. **BIBLIOGRAPHY**

[1] Douglas L. Perry, “*VHDL: Programming by example*” [Online]. Available: <http://vhdl-manual.narod.ru/books/programming_by_example.pdf>

[2] “*Arithmetic Logic Unit*” [Online]. Available: <https://teachcomputerscience.com/arithmetic-logic-unit/>

[3] “*Addition and Subtraction using 2's complement*” [Online]. Available:<https://www.javatpoint.com/addition-and-subtraction-using-2s-complement-in-digital-electronics>

[4] Gojko Babić , “*Arithmetic / Logic Unit – ALU Design*” [Online]. Available: <https://web.cse.ohio-state.edu/~crawfis.3/cse675-02/Slides/CSE675_05_ALUDesign.pdf>

[5] Ann Gordon-Ross , “*ALU Design : Division and Floating Point*” [Online]. Available: <http://www.ann.ece.ufl.edu/courses/eel4713_14fal/slides/Lec8-division.pdf>

[6] Dr. Warren Toomey, and slightly modified by Dr. Mirela Damian , “*Logic Gates - Building an ALU*” [Online]. Available: <http://www.csc.villanova.edu/~mdamian/Past/csc2400fa13/assign/ALU.html>

[7] Edward L. Bosworth , “*Design of the ALU: Adder, Logic, and the Control Unit*” [Online].Available: <http://www.edwardbosworth.com/CPSC2105/Lectures/Slides_06/Chapter_05/ALU_Design.htm>

[8] Dr. Eng. Sebestyen-Pal Gheorghe, Structure of Computer Systems, Course 3 slides, “*The Arithmetical and Logical Unit*”