TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

Structure of Computer Systems

Arithmetic Logic Unit (ALU) Implementation in VHDL

Student: Jurcan Diana-Maria

Group: 30432

Coordinating Teacher: Dr. Eng. Hangan Anca

Academic Year: 2022-2023

Institution: Technical University of Cluj-Napoca

Table of Contents

1. Introduction

1.1. Project Assignment:

Design and implement an ALU – Arithmetic Logic Unit in VHDL which performs the following operations:

- Addition in 2’s Complement
- Subtraction in 2’s Complement
- Incrementation and decrementation
- Logical bitwise operations AND, OR and NOT
- Logical operation of negation
- Left and right rotations

The ALU should utilize an accumulator register for the input and for the result of an operation and a supplementary circuit for multiplication and division.

2. Bibliographical Research

2.1. What is an Arithmetic Logic Unit?

An arithmetic logic unit is digital circuit, representing a fundamental building block of the central processing unit (CPU). It has the ability to perform all processes related to arithmetic and logic operations such as addition, subtraction, and shifting operations, including boolean comparisons (XOR, OR, AND, NOT operations). Most of the operations of a CPU are performed by one or more ALUs, which load data from input registers [1].

2.2. How are numbers represented in 2's Complement?

Two's Complement is a method of representing numbers derived from the binary representation, through which negative numbers, as well as positive numbers can be obtained. The most significant bit determines the sign of the number. When this bit is '1' the number is negative and when said bit is '0' the number is positive.

Positive numbers are represented as in the signed magnitude method, while negative numbers are represented through a series of steps. The number should be converted to a binary system and then, to One's Complement, which is done by inverting all the bits. Lastly, '1' is added to the result of the inversion.

Example: Let us use 5 bits registers. The representation of -5 and +5 will be as follows.

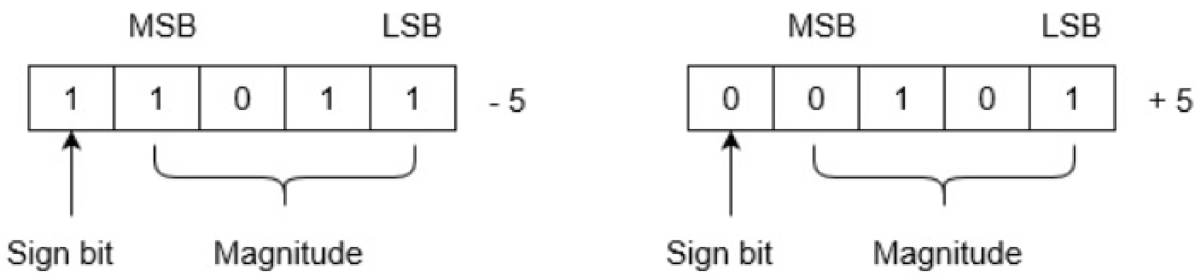


Figure 1. Representation of -5 and +5

+5 is represented as it is represented in sign magnitude method.
-5 is represented using the following steps:
Step 1: +5 = 0 0101
Step 2: Take 2's complement of 0 0101 and that is 1 1011. MSB is 1 which indicates that number is negative. MSB is always 1 in case of negative numbers [3].

2.3. How is addition performed in 2's Complement?

The addition works similarly to binary, but the carry bit denotes the sign of the result. We have three cases:

Case 1: Both numbers are positive.

The addition is performed the same way as the addition in binary.

Case 2: One number is negative and the other is positive.

- If the negative number has a higher magnitude: 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.
- If the positive number has a higher magnitude: 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.

Example: Add −8 to +3: [6]

```
   (+3)  0000 0011
 + (−8)  1111 1000
 ----------------------------
    (−5)  1111 1011
```

Case 3: Both numbers are negative.

Convert both numbers to 2's Complement and perform the addition. The carry is discarded, and the final result is the 2's Complement of the result [2].

Example: Add −5 to −2: [6]

```
  (−2)  1111 1110
+(−5)  1111 1011
---------------------------------------------
  (−7) 1 1111 1001 -> discard carry-out
```

Overflow Rule for addition: If 2 Two's Complement numbers are added, and they both have the same sign (both positive or both negative), then overflow occurs if and only if the result has the opposite sign. Overflow never occurs when adding operands with different signs [6]. Overflow occurs if:

- (+A) + (+B) = −C
- (−A) + (−B) = +C

Example: Subtract -6 from -7: [6]

```
  (−7)    1001
+(−6)    1010
------------------------
 (−13) 1 0011 = 3 -> Overflow (largest number is −8)
```

2.4. How is subtraction performed in 2’s Complement?

Find the 2’s Complement of the subtrahend. Add the 2’s complement of the minuend. If we have a final carry, then the result is positive, in its true form. Else if we have a final carry, then the result is negative and in its 2’s complement form [2].

Example: Subtract +5 from +8

```
  (+8) 0000 1000                       0000 1000
−(+5) 0000 0101 -> negate -> + 1111 1011
----------------------                -----------------
   (+3)                              1 0000 0011 -> discard carry-out
```

Overflow Rule for Subtraction: If 2 Two's Complement numbers are subtracted, and their signs are different, then overflow occurs if and only if the result has the same sign as the subtrahend [6]. Overflow occurs if

- (+A) − (−B) = −C
- (−A) − (+B) = +C

Example: Subtract −6 from +7: [6]

```
  (+7) 0111                      0111
−(−6) 1010 -> negate -> +0111
-----------------        ---------------------------
          13                     1101 = −8 + 5 = −3 -> Overflow
```

2.5. How is multiplication performed in 2’s Complement?

Take the complement of two of both operands. Sign extend both integers to twice as many bits [7]. Perform multiplication between the numbers. Then take the correct number of result bits from the least significant portion of the result. Negate product if original signs indicate so [3].

Example: Multiply 3 with -5: [7]

```
                    0000 0011 (3)
                  x 1111 1011 (-5)
-------------------------------------
                     00000011
                    00000011
                   00000000
                 00000011
                00000011
               00000011
              0000001
   + 00000011
-------------------------------------
                 1011110001
------------------------------------- take the least significant 8 bits 11110001 = -15
```

2.6. How is division performed in 2’s Complement?

For the 2’s complement division, the method is 2’s complement subtraction repeatedly. Take the 2’s complement of the divisor and add it to the dividend. In the following subtraction cycle, the quotient replaces the dividend. Keep subtracting until the quotient becomes too small or 0 (if it’s not 0, it is treated as a remainder) [4].

Example: Divide 19 by 6, result = 2, remainder = 1 [8]:

Firstly, 19 is represented in binary as 0001 0011 and 6 is represented as 0000 0110. Consequently, the two’s complement representation of −6 is 1111 1010.
Now if we add together these two numbers and discard the carry bit we get:

```
  00010011
+ 11111010
-----------------
   00001101
   q=0001
```

0000 0110 is smaller than 0000 1101 so we should try again:

```
  00001101
+11111010
-----------------
  00000111
  q=0010
```

Once again, 0000 0110 is smaller than 0000 0111, so we try again:

```
  00000111
+11111010
-----------------
  00000001
  q=0011
```

We can now see that 0000 0110 is bigger than 0000 0001 so we can stop. Consequently, as expected, we have a quotient of 0011 (3) and a remainder of 0001 (11).

2.7. How are logical operations on bitwise integers performed in VHDL?

Use:

- type conversions (automatic and manual)
- logical and arithmetic operators (and, or, not, +, -, *, /)
- comparison operators (<, >, =, /=, <=, >=)
- shift operators (>>, <<)
- packages: std_logic_1164, std_logic_arith [5].

3. Project plan and proposal

3.1. Project Plan

Week	Tasks
Week 1 (3.10.2022-9.10.2022) & Week 2 (10.10.2022-16.10.2022) – Project Meeting 1	→ Choosing theme of project → Getting acquainted with chosen theme
Week 3 (17.10.2022-23.10.2022) & Week 4 (24.10.2022-30.10.2022) – Project Meeting 2	→ Research of number representation in 2's complement & operations → Writing of project introduction and bibliographical research
Week 5 (31.10.2022-6.11.2022) & Week 6 (7.11.2022-13.11.2022) – Project Meeting 3	→ Project scheduling, weekly plan of tasks → Research algorithms used for the design of the project → Analysis of main parts of project
Week 7 (14.11.2022-20.11.2022) & Week 8 (21.11.2022-27.11.2022) – Project Meeting 4	→ Design of components needed → Implementation of arithmetical and logical operations (add, subtract, increment, decrement, OR, NOT, AND)
Week 9 (28.11.2022-4.12.2022) & Week 10 (5.12.2022-11.12.2022) – Project Meeting 5	→ Implementation of multiplication and division operations → Implementation of register unit and control unit
Week 11 (12.12.2022-18.12.2022) & Week 12 (19.12.2022-25.12.2022) – Project Meeting 6	→ Creating a test bench → Testing the project on the FPGA → Fixing potential bugs → Finalizing documentation
Week 13 (9.01.2022-15.01.2022) & Week 14 (16.01.2022-22.01.2022) – Project Meeting 7	→ Presentation of final project and documentation

3.2. Project Proposal and Analysis

3.2.1. Project Proposal

The project will perform arithmetical and logical operations on numbers represented on 16 bits. Beginning from simple components on 1-bit, the interconnection of 16 such components will generate the final result on 16 bits, which will be displayed for the user in hexadecimal using the 7-segment display on the Basys3 FPGA Board. The operands will be selected using the 16 slide-switches and pushing one of the buttons to load them. There will be two buttons for switching between operations (when choosing the operation, the name will be displayed on the 7-segment display and the user will have to press another button for the operation to be done).

The control unit will handle the more complex interface operations s.a. selection of desired operation, display of operation selection and display of result, while the command unit will send the proper signals and operating modes to an operational unit (used for handling arithmetical and logical operations) and a register unit (used for exchange and storage of data inside the ALU).

3.2.2. Project Analysis

During the analysis, the functionalities provided in the project assignment and proposal will be explored.

→ Addition operation

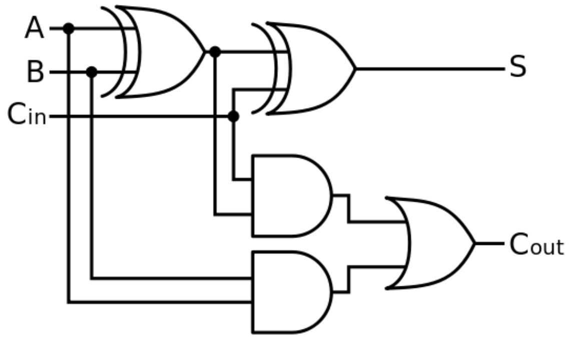


Figure 2. Full Adder on 1 bit

Firstly, the starting point is the implementation of a 1-bit adder. The equations that describe this operation are:

- $\text{Sum} = X \oplus Y \oplus \text{CarryIn}$
- $\text{CarryOut} = X \cdot Y + (X \oplus Y) \cdot \text{CarryIn}$

The addition on 16 bits will be implemented using Ripple Carry Addition. A ripple carry adder is a logic circuit in which the carry-out of each full adder is the carry in of the succeeding next most significant full adder.

The operations for obtaining the addition will become:

- $\text{Sum}_i = X_i \oplus Y_i \oplus \text{CarryIn}_i$
- $\text{CarryOut}_i = X_i \cdot Y_i + (X_i \oplus Y_i) \cdot \text{CarryIn}_{i+1}$

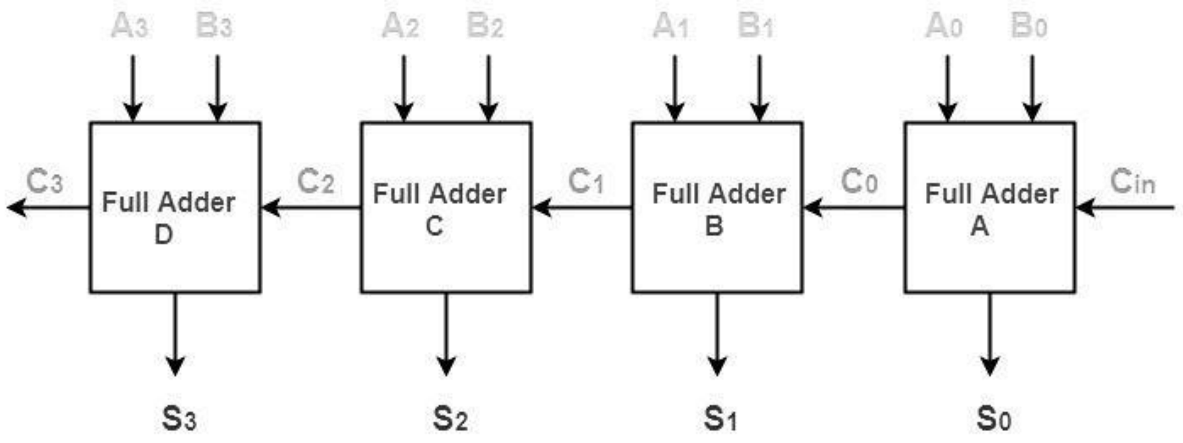


Figure 3. Ripple Carry Adder on n bits

Overflow can occur only if both the operands have the same sign or if the sign of the result differs from the sign of the operands. For that reason, overflow will be detected if the input and output carry of the most significant rank differ.

→ Subtraction operation

The subtraction will be done by negating all the bits of the second operand and by adding one to said negation. This operation ends up being a simple addition between the subtrahend and the opposite of the subtractor: $X - Y = X + [(\text{not})Y + 1]$.

→ Increment operation

Incrementation is done by increasing the value of the operand by 1 (X = X + 1). This operation is an addition for which the second operand is equal to 1, so the implementation will remain the same as for addition.

→ Decrement operation

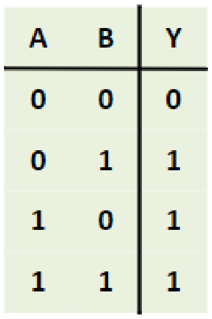
Similar to incrementation, decrementation is done by decreasing the value of the operand by 1 (X = X - 1). This operation is a subtraction for which the second operand is equal to 1, so the implementation will remain the same as for subtraction.

→ Negation

Negation operation returns the opposite of a number. In 2's complement, this is achieved by inverting all the bits and adding one to the result. The equation is: $X = \overline{X} + 1$.

→ Logical OR

Operation OR will generate a logical '1' when at least one of the operand bits has the value '1' logical.

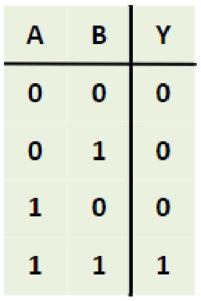


A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4. Logical OR Truth Table

→ Logical AND

Operation AND will generate the value logical '1' when both operands have the value '1' logical.

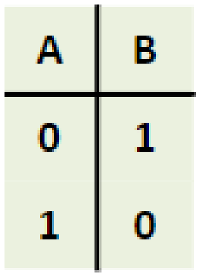


A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 5. Logical AND Truth Table

→ Logical NOT

Operation NOT will negate the value of the operand.



A	B
0	1
1	0

Figure 6. Logical NOT Truth Table

→ Rotate Left

The operation performs circular shifts to the left of the bits of the operand by one position. On the position of the least significant bit, the most significant one will be placed.

After the rotation, a 16-bit number represented initially as: $X = X_{31}X_{30}...X_1X_0$,
the number will become: $X = X_{30}X_{29}...X_0X_{31}$.

→ Rotate Right

The operation performs circular shifts to the right of the bits of the operand by one position. On the position of the most significant bit, the least significant one will be placed.

After the rotation, a 16-bit number represented initially as: $X = X_{31}X_{30}...X_1X_0$,
the number will become: $X = X_0X_{31}...X_2X_1$.

→ Multiplication

The multiplication operation will be the shift-and-add multiplication which is similar to the multiplication performed by paper and pencil. This method adds the multiplicand X to itself Y times, where Y denotes the multiplier. To multiply two numbers, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results.

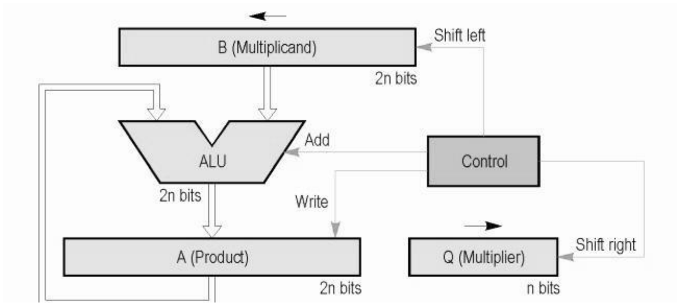


Figure 7. Shift and Add Multiplier Circuit

The 2n-bit product register (A) is initialized to 0. Since the basic algorithm shifts the multiplicand register (B) left one position each step to align the multiplicand with the sum being accumulated in the product register, we use a 2n-bit multiplicand register with the multiplicand placed in the right half of the register and with 0 in the left half.

The algorithm starts by loading the multiplicand into the B register, loading the multiplier into the Q register, and initializing the A register to 0. The counter N is initialized to n. The least significant bit of the multiplier register (Q_0) determines whether the multiplicand is added to the product register. The left shift of the multiplicand has the effect of shifting the intermediate products to the left, just as when multiplying by paper and pencil. The right shift of the multiplier prepares the next bit of the multiplier to examine in the following iteration.

→ Division

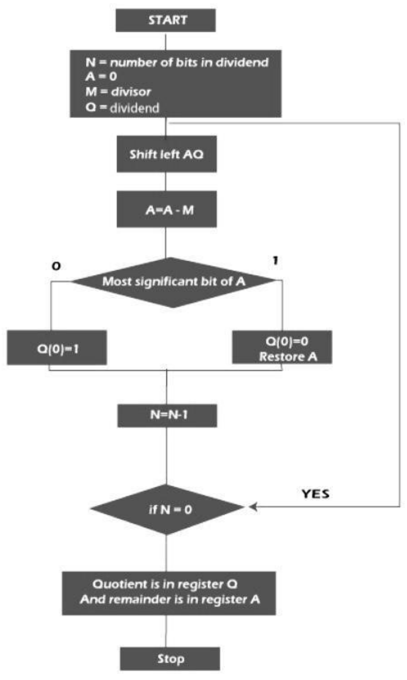

START
N = number of bits in dividend
A = 0
M = divisor
Q = dividend
Shift left AQ
A=A - M
0
1
Most significant bit of A
Q(0)=1
Q(0)=0
Restore A
N=N-1
YES
if N = 0
Quotient is in register Q
And remainder is in register A
Stop

Figure 8. Division Algorithm

The division operation will be the restoring division algorithm. In each step of the algorithm there is a shift of the divisor to the right by one position and a shift of the quotient to the left by one binary position, the algorithm ending when the divisor becomes less than the remainder of the division.

Now we will learn some steps of restoring division algorithm, which is described as follows:

→ The corresponding value will be initialized to the registers, i.e., register A will contain value 0, register M will contain Divisor, register Q will contain Dividend, and N is used to specify the number of bits in dividend.
→ In this step, register A and register Q will be treated as a single unit, and the value of both the registers will be shifted left.
→ After that, the value of register M will be subtracted from register A. The result of subtraction will be stored in register A.
→ Now, check the most significant bit of register A. If this bit of register A is 0, then the least significant bit of register Q will be set with a value 1. If the most significant bit of A is 1, then the least significant bit of register Q will be set to with value 0, and restore the value of A that means it will restore the value of register A before subtraction with M.
→ After that, the value of N will be decremented. Here n is used as a counter.
→ Now, if the value of N is 0, we will break the loop. Otherwise, we have to again go to the second step.
→ This is the last step. In this step, the quotient is contained in the register Q, and the remainder is contained in register A.

4. Design

4.1. Internal structure

4.1.1. Structure Description

To achieve the functionalities required by the task at hand, the following module structure of the system was considered for implementation:

→ The Operational Unit (OP) will contain the implementation of the following arithmetic and logical operations in two large groups and smaller subgroups:
- Basic operations:
  - Arithmetic:
    - Addition (ADD)
    - Subtraction (SUB)
    - Incrementation (INC)
    - Decrementation (DEC)
  - Logical and Positional:
    - AND, OR, NOT
    - Negation (NEG).
    - Left rotation (LR)
    - Right rotation (RR)
- Complex operations:
  - Multiplication (MUL)
  - Division (DIV)

4.2. Internal encoding:

The Arithmetic Logic Unit's design is built on a few elements required to finish implementing the necessary operations from the original task.

ALU design requires the use of 16-bit registers, 1-bit full adders, and multiplexers. There will be two operands in the arithmetic logic unit. Additionally, there will be two additional distinct registers for the outputs. Because the result of the multiplication method will be on 32 bits, I require two 16-bit registers to store the result, hence I need two registers for the outputs. The second 16-bit register will be used to store the results of the remaining operations because the first 16-bit register will be idle and the result will only be on 16 bits.

The components from which the final ALU will be built will be depicted for each operation in this project. I'll need a Control Unit component, which is basically a 16:1 multiplexer, to determine which operation will be executed. The operations will be chosen based on a code that represents the multiplexer selection. Because there are 12 such operations, they must be encoded on 4 bits so that each instruction has a unique code.

4.2.1. OP unit encoding:

To practically refer to the operations, they will have the following encoding, which will enable the user to tactfully select which operation to perform (12 operations, thus 4 bits are used):

- ADD = 0000
- SUB = 0001
- INC = 0010
- DEC = 0011
- AND = 0100
- OR = 0101
- NOT = 0110
- NEG = 1111
- LR = 1101
- RR = 1011
- MUL = 1000
- DIV = 1001

4.3. Design Schematic

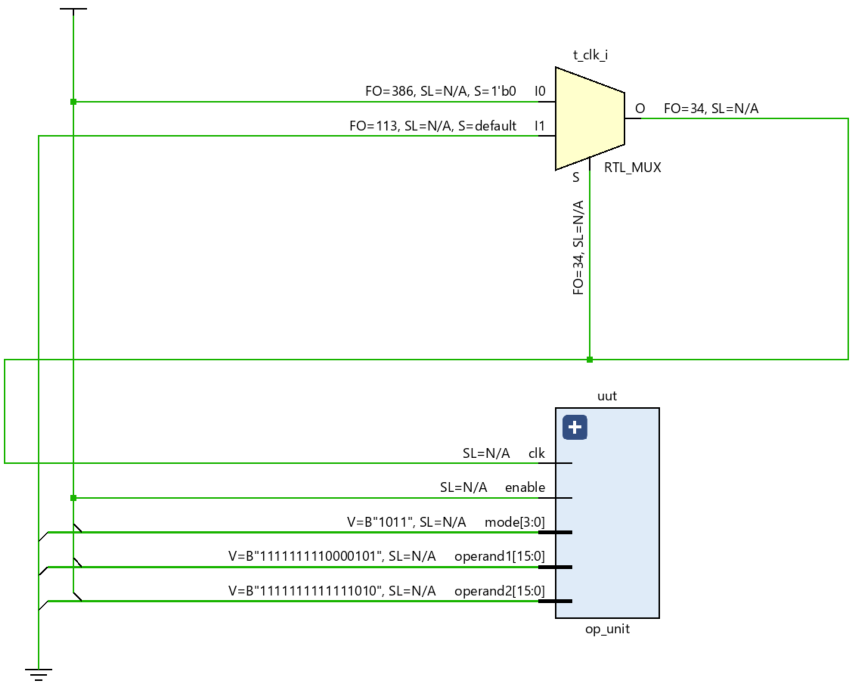

t_clk_i
FO=386, SL=N/A, S=1'b0 I0
FO=113, SL=N/A, S=default I1
O FO=34, SL=N/A
RTL_MUX
S
FO=34, SL=N/A
uut
SL=N/A clk
SL=N/A enable
V=B"1011", SL=N/A mode[3:0]
V=B"1111111110000101", SL=N/A operand1[15:0]
V=B"1111111111111010", SL=N/A operand2[15:0]
op_unit

Figure 9. RTL schematic

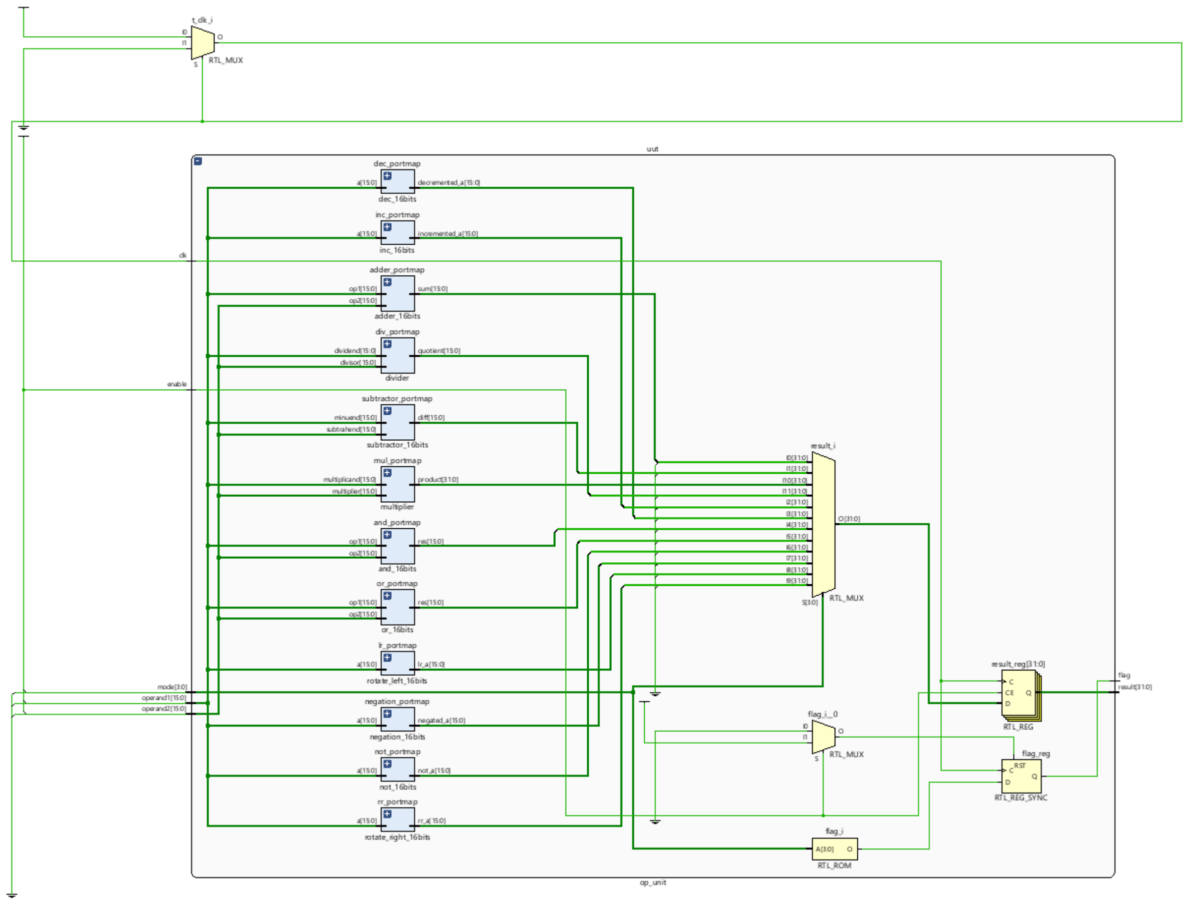


Figure 10. Elaborated Schematic

5. Implementation

This chapter is closely related to the Design one, but it makes the project more palpable. During this section I will explain all the components of this project and the logic behind them, as well as present the code that was written in VHDL.

5.1. Operational Unit:

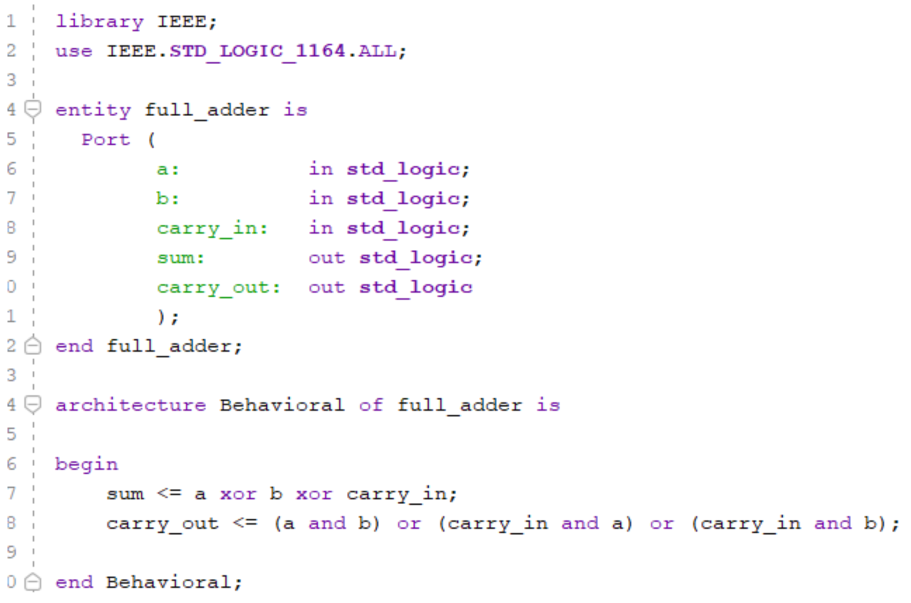
The Operational unit is the module that handles the arithmetical and logical operations on the provided operands. It contains the logic necessary for recognizing the user chosen operation and relaying it to the output of the unit.

The ports for the operational unit:

- enable: enables the unit.
- clk: clock signal for synchronization.
- operand1: the first operand for the operations.
- operand2: the second operand for the operations.
- mode: the code for the desired operation.
- result: the result of the chosen operation on the provided operands.

5.1.1. Addition

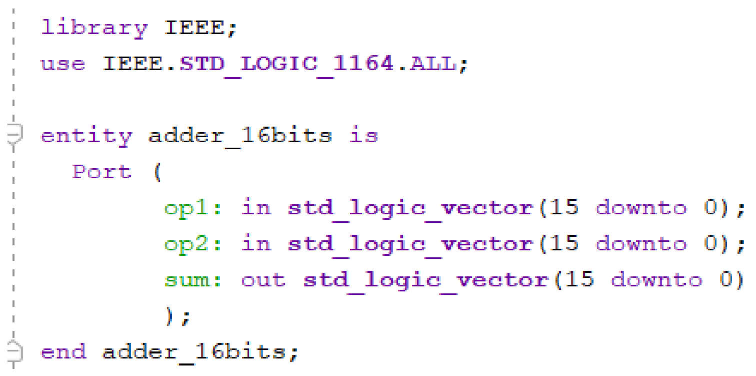
In order to implement the component for the 16-bit addition, we will use a full adder on 1 bit. The code for it is:



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder is
  Port (
        a:          in std_logic;
        b:          in std_logic;
        carry_in:   in std_logic;
        sum:        out std_logic;
        carry_out:  out std_logic
        );
end full_adder;

architecture Behavioral of full_adder is

begin
    sum <= a xor b xor carry_in;
    carry_out <= (a and b) or (carry_in and a) or (carry_in and b);

end Behavioral;
```

Figure 11. Full adder on 1 bit

The full adder takes in two single-bit inputs (a and b), a carry-in input, and outputs a sum and a carry-out. The 16-bit ripple adder will have this entity:



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity adder_16bits is
  Port (
        op1: in std_logic_vector(15 downto 0);
        op2: in std_logic_vector(15 downto 0);
        sum: out std_logic_vector(15 downto 0)
        );
end adder_16bits;
```

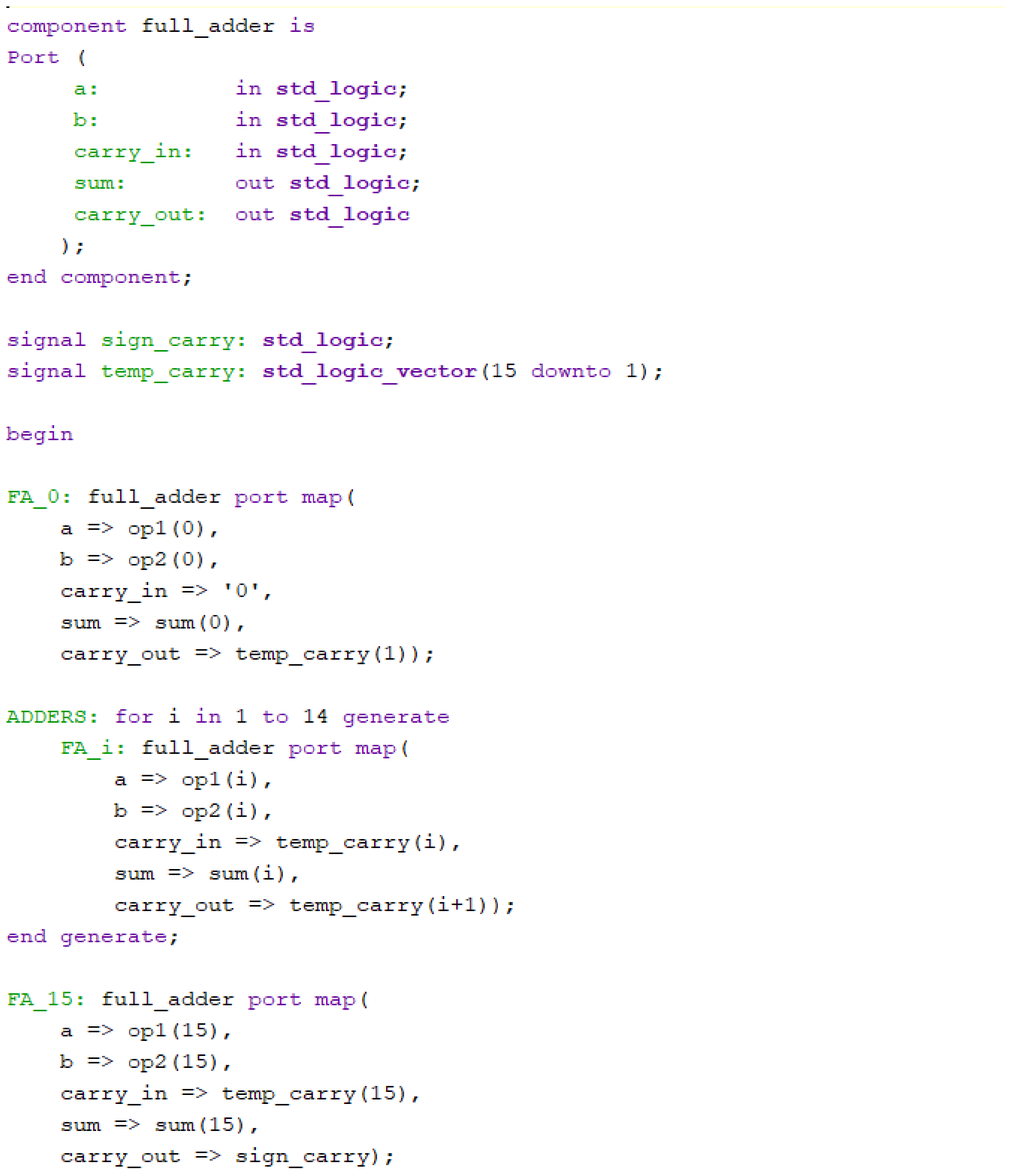
Figure 12. Entity of 16-bit adder

The entity "adder_16bits" has three ports: two 16-bit input vectors (op1 and op2), and a 16-bit output vector (sum). The architecture "Behavioral" of the entity describes the design of the adder using the full adder component.

The architecture uses a generate loop (ADDERS) to instantiate 15 full adders, with the first full adder instantiated separately before the loop (FA_0). Each full adder instance takes in one bit from each input vector (op1 and op2) as well as the carry-in from the previous full adder. The sum output of each full adder is connected to the corresponding bit of the output vector (sum), and the carry-out is connected to the carry-in of the next full adder.

The final full adder (FA_15) takes in the last bit of each input vector and the carry-in from the previous full adder, and outputs the last bit of the sum and a signal (sign_carry) representing the final carry-out.

This design allows the adder to perform a 16-bit addition and generate the sum and carry-out.

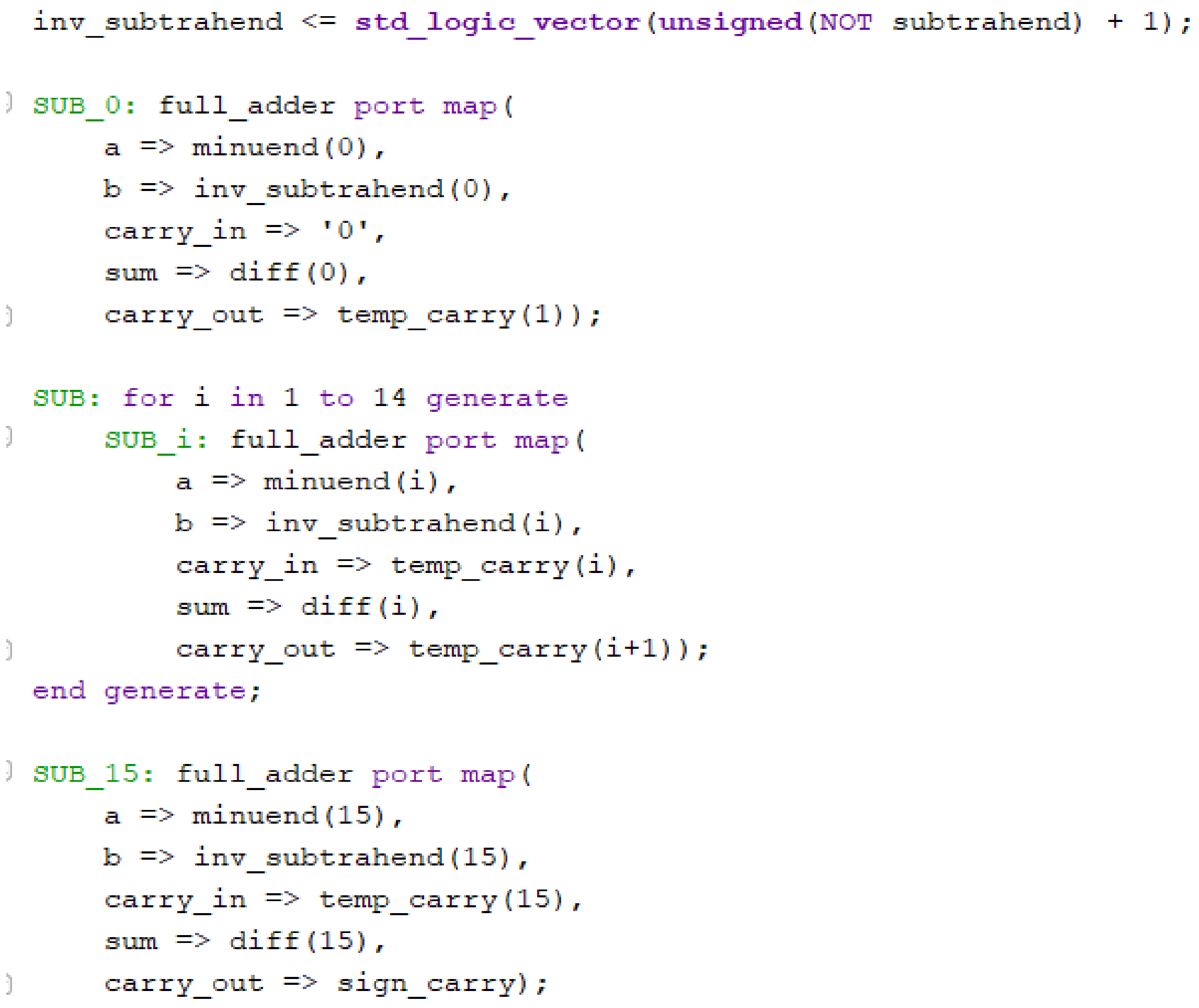


```
component full_adder is
Port (
     a:           in std_logic;
     b:           in std_logic;
     carry_in:    in std_logic;
     sum:         out std_logic;
     carry_out:   out std_logic
    );
end component;

signal sign_carry: std_logic;
signal temp_carry: std_logic_vector(15 downto 1);

begin

FA_0: full_adder port map(
    a => op1(0),
    b => op2(0),
    carry_in => '0',
    sum => sum(0),
    carry_out => temp_carry(1));

ADDERS: for i in 1 to 14 generate
    FA_i: full_adder port map(
        a => op1(i),
        b => op2(i),
        carry_in => temp_carry(i),
        sum => sum(i),
        carry_out => temp_carry(i+1));
end generate;

FA_15: full_adder port map(
    a => op1(15),
    b => op2(15),
    carry_in => temp_carry(15),
    sum => sum(15),
    carry_out => sign_carry);
```

Figure 13. Architecture of 16-bit adder

5.1.2. Subtraction

The subtraction is implemented using the 1-bit full adder as well. The only changed thing is the signal inv_subtrahend, which is assigned the result of the bitwise NOT operation on the subtrahend signal, followed by adding 1. This is because in 2's complement, the negative of a number can be obtained by inverting all its bits and adding 1.



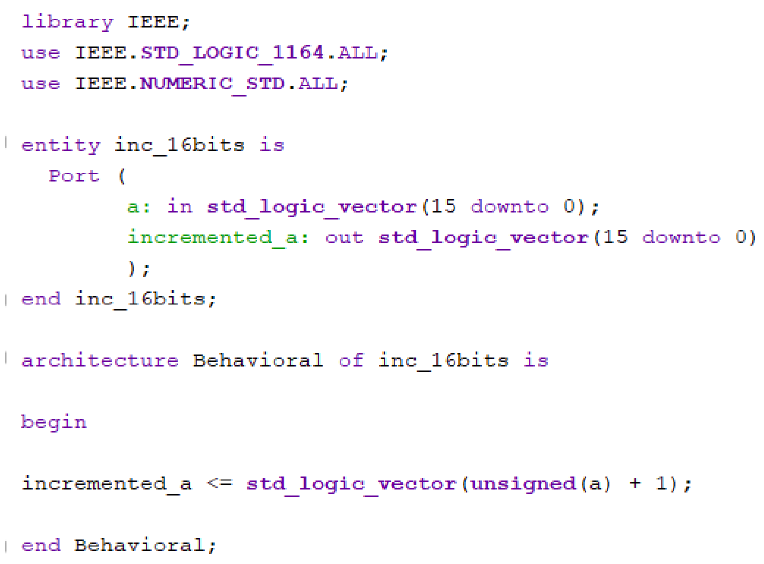
```
inv_subtrahend <= std_logic_vector(unsigned(NOT subtrahend) + 1);

SUB_0: full_adder port map(
    a => minuend(0),
    b => inv_subtrahend(0),
    carry_in => '0',
    sum => diff(0),
    carry_out => temp_carry(1));

SUB: for i in 1 to 14 generate
    SUB_i: full_adder port map(
        a => minuend(i),
        b => inv_subtrahend(i),
        carry_in => temp_carry(i),
        sum => diff(i),
        carry_out => temp_carry(i+1));
end generate;

SUB_15: full_adder port map(
    a => minuend(15),
    b => inv_subtrahend(15),
    carry_in => temp_carry(15),
    sum => diff(15),
    carry_out => sign_carry);
```

Figure 14. Architecture of 16 bit subtractor

The logic will remain the same for the architecture as used before for the 16-bit adder.

5.1.3. Increment

The incrementer has two ports, a and incremented_a. The input a is a std_logic_vector of size 16 bits and the output incremented_a is also a std_logic_vector of size 16 bits. It then increments the input a by using the std_logic_vector function, which converts the input a to an unsigned integer, adds 1 to the result, and then converts it back to a std_logic_vector. This is how the increment operation is performed on the 2's complement number.

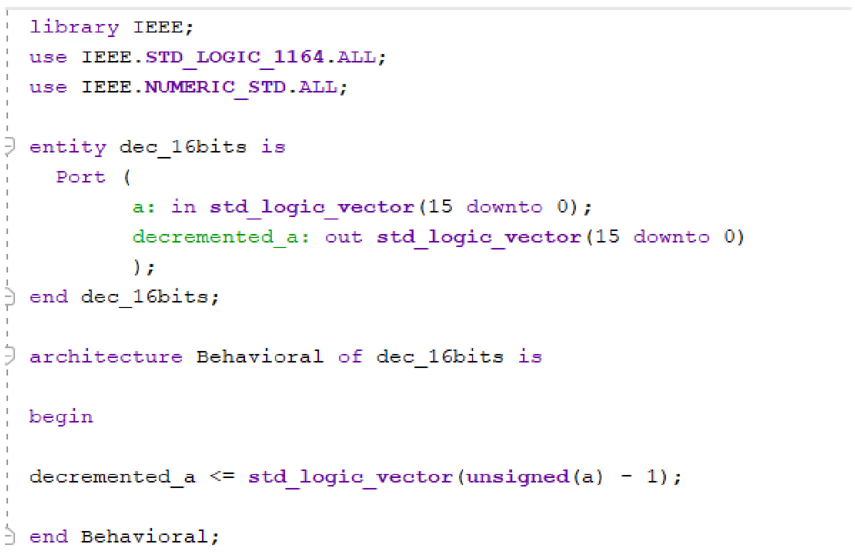


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity inc_16bits is
  Port (
        a: in std_logic_vector(15 downto 0);
        incremented_a: out std_logic_vector(15 downto 0)
        );
end inc_16bits;

architecture Behavioral of inc_16bits is

begin

incremented_a <= std_logic_vector(unsigned(a) + 1);

end Behavioral;
```

Figure 15. Increment on 16 bits

5.1.4. Decrement

The decrementer has two ports, a and decremented_a. The input a is a std_logic_vector of size 16 bits and the output decremented_a is also a std_logic_vector of size 16 bits. It then decrement the input a by using the std_logic_vector function, which converts the input a to an unsigned integer, subtracts 1 from the result, and then converts it back to a std_logic_vector. This is how the decrement operation is performed on the 2's complement number.

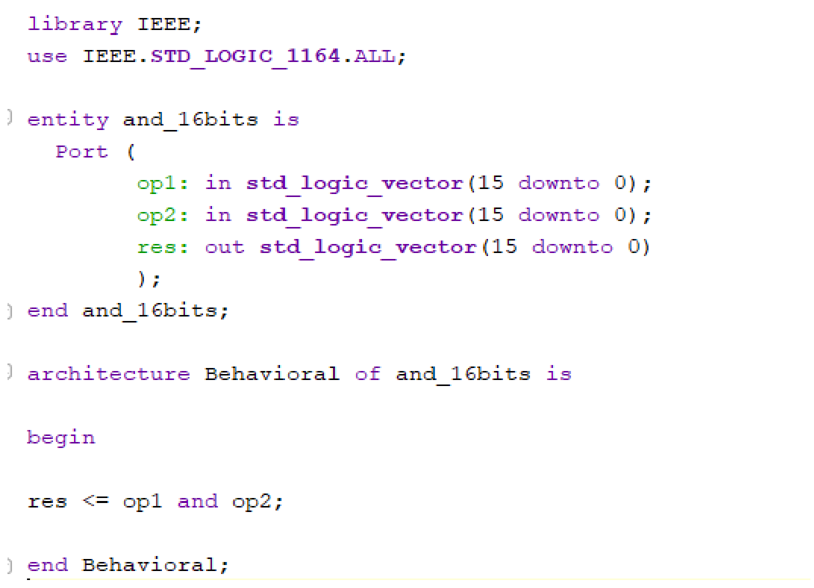


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity dec_16bits is
  Port (
        a: in std_logic_vector(15 downto 0);
        decremented_a: out std_logic_vector(15 downto 0)
        );
end dec_16bits;

architecture Behavioral of dec_16bits is

begin

decremented_a <= std_logic_vector(unsigned(a) - 1);

end Behavioral;
```

Figure 16. Decrement on 16 bits

5.1.5. Logical AND

The architecture contains a single line of code "res <= op1 and op2", which assigns the bitwise AND of the inputs (op1 and op2) to the output (res). This code is not specific to 2's complement numbers, it is just a simple bit-wise logical AND operation on two 16-bit inputs and it doesn't matter whether the inputs are 2's complement numbers or not.

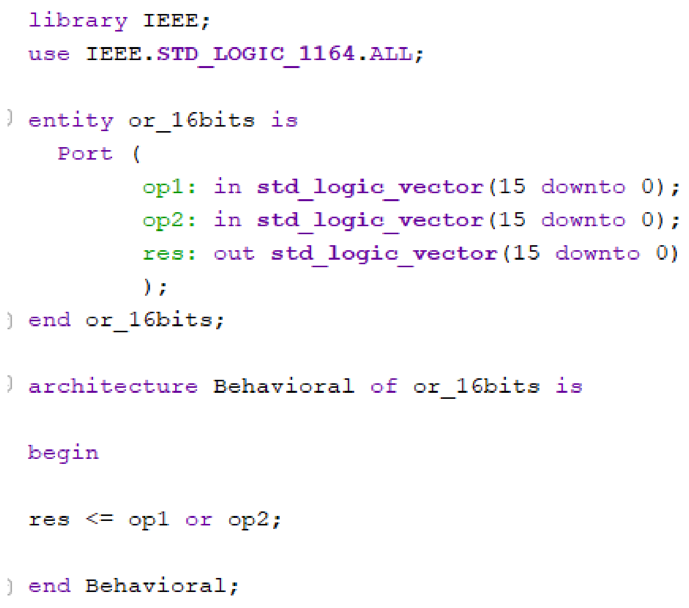


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity and_16bits is
  Port (
        op1: in std_logic_vector(15 downto 0);
        op2: in std_logic_vector(15 downto 0);
        res: out std_logic_vector(15 downto 0)
        );
end and_16bits;

architecture Behavioral of and_16bits is

begin

res <= op1 and op2;

end Behavioral;
```

Figure 17. Logical AND on 16 bits

5.1.6. Logical OR

The architecture contains a single line of code "res <= op1 or op2", which assigns the bit-wise OR of the inputs (op1 and op2) to the output (res). Same as before, this code is not specific to 2's complement numbers, it is just a simple bit-wise logical OR operation on two 16-bit inputs and it doesn't matter whether the inputs are 2's complement numbers or not.

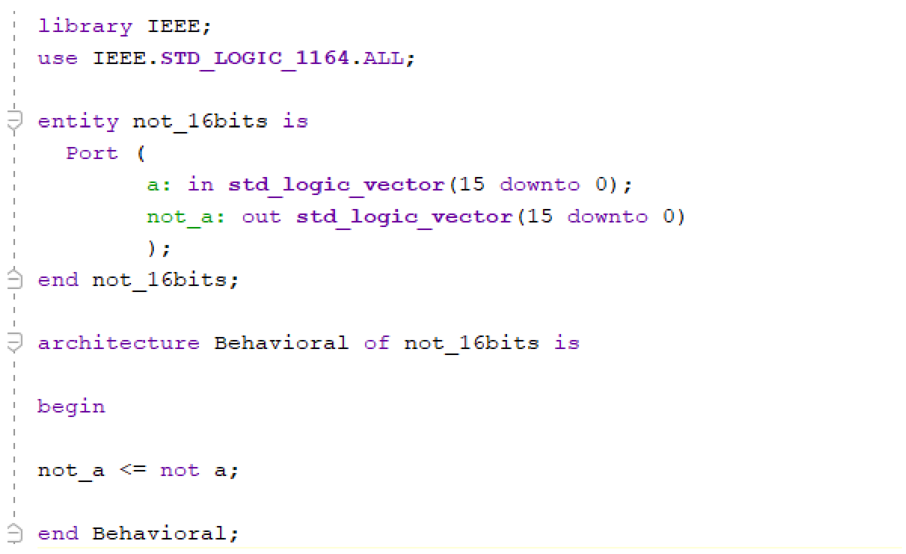


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity or_16bits is
  Port (
        op1: in std_logic_vector(15 downto 0);
        op2: in std_logic_vector(15 downto 0);
        res: out std_logic_vector(15 downto 0)
        );
end or_16bits;

architecture Behavioral of or_16bits is

begin

res <= op1 or op2;

end Behavioral;
```

Figure 18. Logical OR on 16 bits

5.1.7. Logical NOT

The architecture contains a single line of code "not_a <= not a", which assigns the bit-wise NOT of the input (a) to the output (not_a). The bit-wise NOT operation flips all the bits of the input, 0 becomes 1 and vice versa, regardless of representation of the input.

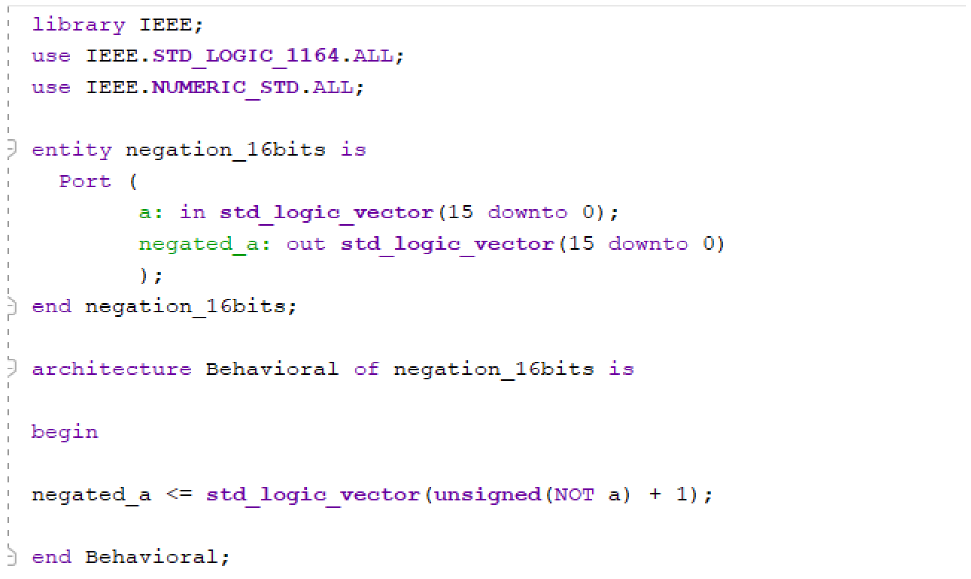


```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity not_16bits is
  Port (
        a: in std_logic_vector(15 downto 0);
        not_a: out std_logic_vector(15 downto 0)
        );
end not_16bits;

architecture Behavioral of not_16bits is

begin

not_a <= not a;

end Behavioral;
```

Figure 19. Logical NOT on 16 bits

5.1.8. Negation

The architecture contains a single line of code "negated_a <= std_logic_vector(unsigned(NOT a) + 1)", which assigns the negation of the input (a) to the output (negated_a) in two's complement representation. The line of code first performs a bit-wise NOT operation on the input (a) using the NOT function, then it converts the result to an unsigned integer using the unsigned function, then it adds 1 to the result using the + operator and finally it converts the result back to a std_logic_vector using the std_logic_vector function. In two's complement representation, negation of a number is achieved by inverting all the bits of the number and adding 1 to it. This is the operation that is being performed by the code.

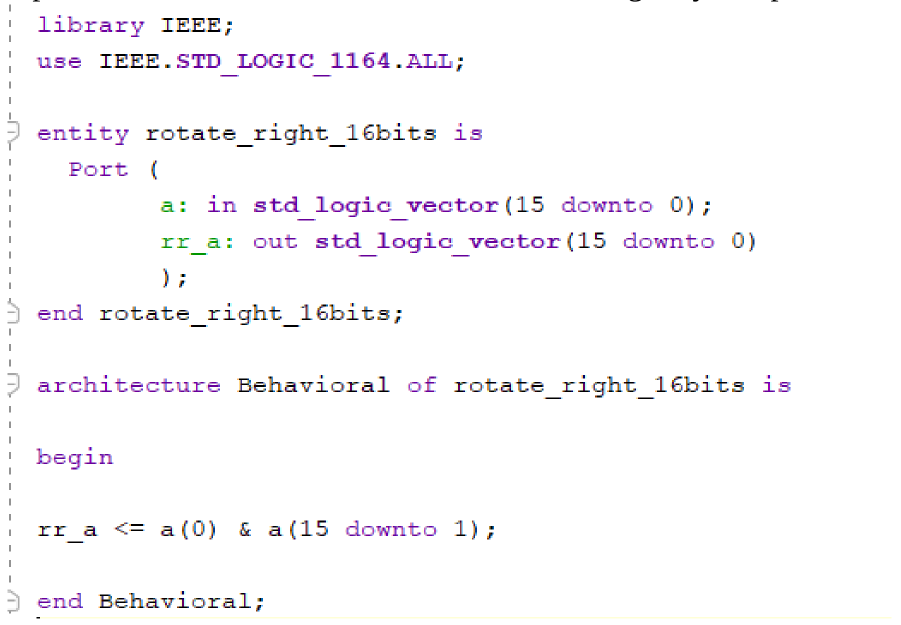


```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity negation_16bits is
  Port (
        a: in std_logic_vector(15 downto 0);
        negated_a: out std_logic_vector(15 downto 0)
        );
end negation_16bits;

architecture Behavioral of negation_16bits is

begin

negated_a <= std_logic_vector(unsigned(NOT a) + 1);

end Behavioral;
```

Figure 20. Negation on 16 bits

5.1.9. Rotate right

In the architecture Behavioral, the assignment rr_a <= a(0) & a(15 downto 1) concatenates the first bit of the input "a" with the rest of the bits in "a" shifted right by one position.

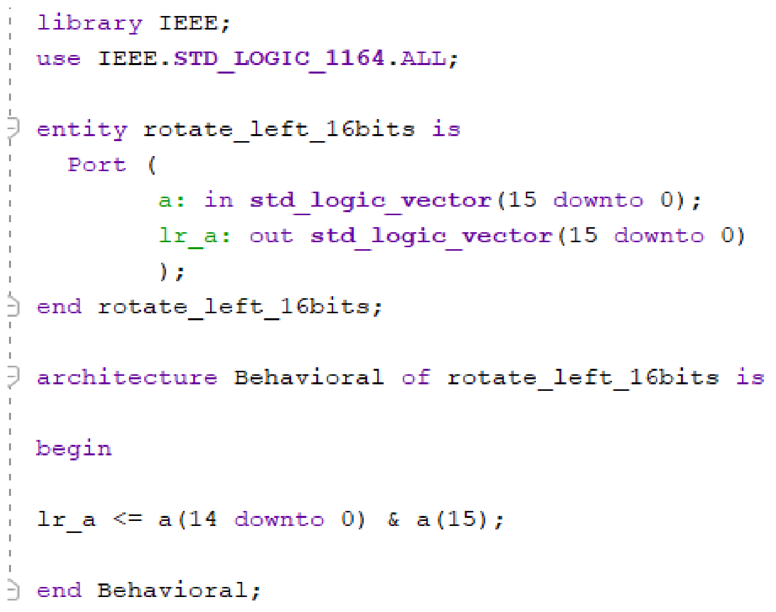


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity rotate_right_16bits is
  Port (
        a: in std_logic_vector(15 downto 0);
        rr_a: out std_logic_vector(15 downto 0)
        );
end rotate_right_16bits;

architecture Behavioral of rotate_right_16bits is

begin

rr_a <= a(0) & a(15 downto 1);

end Behavioral;
```

Figure 21. Rotate right on 16 bits

5.1.10. Rotate left

The architecture contains a single line of code "lr_a <= a(14 downto 0) & a(15)", which assigns the left rotated version of the input (a) to the output (lr_a). The line of code takes the 15 least significant bits of the input (a(14 downto 0)) and concatenates them with the most significant bit of the input (a(15)) using the & operator. This operation shifts all the bits of the input one position to the left, and the leftmost bit that falls off is wrapped around to become the rightmost bit of the output. In two's complement representation, a left rotate operation preserves the sign bit and changes the magnitude of the number.



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity rotate_left_16bits is
  Port (
        a: in std_logic_vector(15 downto 0);
        lr_a: out std_logic_vector(15 downto 0)
        );
end rotate_left_16bits;

architecture Behavioral of rotate_left_16bits is

begin

lr_a <= a(14 downto 0) & a(15);

end Behavioral;
```

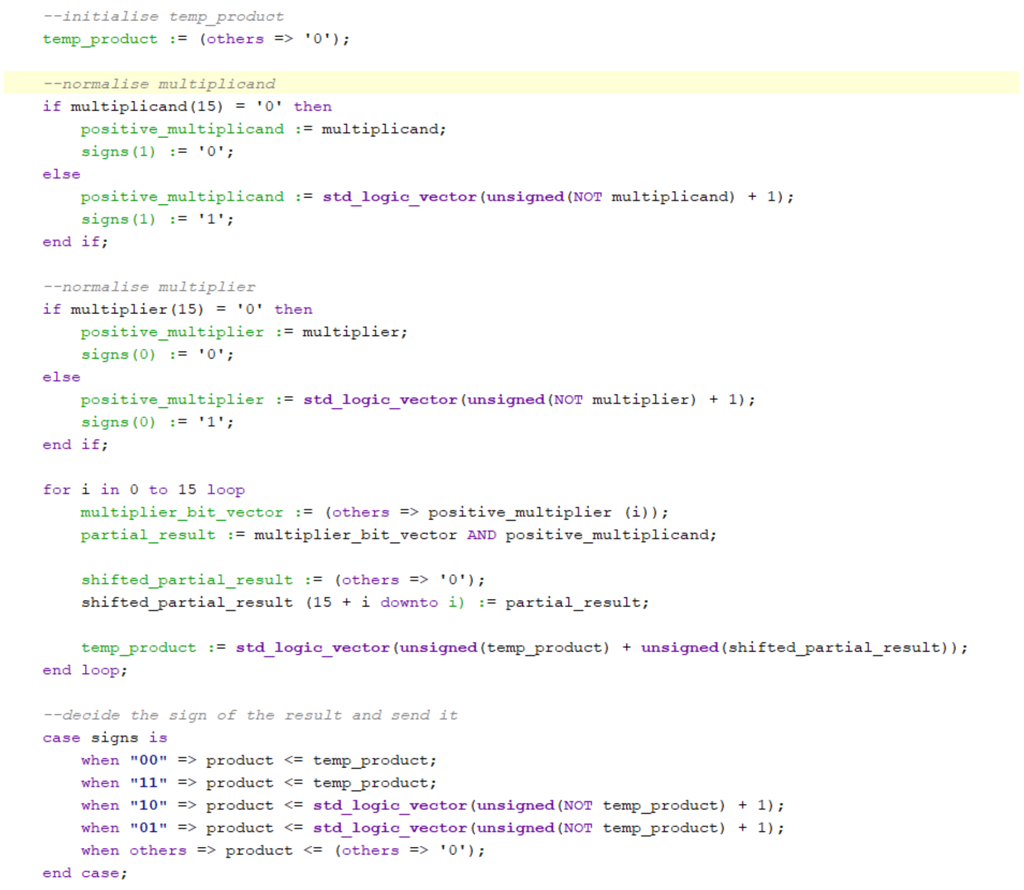
Figure 22. Rotate left on 16 bits

5.1.11. Multiplication

The multiplier uses a modified version of the traditional “long multiplication” algorithm to perform the multiplication. The algorithm works as follows:

- The “multiplicand” and “multiplier” are first “normalized” by converting them to their positive counterparts if they are negative. This is done by taking the two’s complement of the input if it’s negative, and setting a flag to indicate that the input was originally negative.
- The multiplier is then looped through bit by bit, from least significant bit to most significant bit. For each bit of the multiplier, the corresponding partial product is calculated by ANDing the bit with the multiplicand.
- The partial product is then shifted left by the appropriate number of bits, based on the position of the current bit of the multiplier.
- The shifted partial product is then added to the running total of the product, stored in the “temp_product” variable.
- Once all the bits of the multiplier have been processed, the “temp_product” variable contains the final product.
- The final step is to determine the sign of the result based on the signs of the original “multiplicand” and “multiplier”. If either of the inputs are negative, the result will be negative. The sign of the result is then calculated by taking the two’s complement of the “temp_product” if necessary, and then the result is output as the “product”.

The code is the following:



```
--initialise temp_product
temp_product := (others => '0');

--normalise multiplicand
if multiplicand(15) = '0' then
    positive_multiplicand := multiplicand;
    signs(1) := '0';
else
    positive_multiplicand := std_logic_vector(unsigned(NOT multiplicand) + 1);
    signs(1) := '1';
end if;

--normalise multiplier
if multiplier(15) = '0' then
    positive_multiplier := multiplier;
    signs(0) := '0';
else
    positive_multiplier := std_logic_vector(unsigned(NOT multiplier) + 1);
    signs(0) := '1';
end if;

for i in 0 to 15 loop
    multiplier_bit_vector := (others => positive_multiplier (i));
    partial_result := multiplier_bit_vector AND positive_multiplicand;

    shifted_partial_result := (others => '0');
    shifted_partial_result (15 + i downto i) := partial_result;

    temp_product := std_logic_vector(unsigned(temp_product) + unsigned(shifted_partial_result));
end loop;

--decide the sign of the result and send it
case signs is
    when "00" => product <= temp_product;
    when "11" => product <= temp_product;
    when "10" => product <= std_logic_vector(unsigned(NOT temp_product) + 1);
    when "01" => product <= std_logic_vector(unsigned(NOT temp_product) + 1);
    when others => product <= (others => '0');
end case;
```

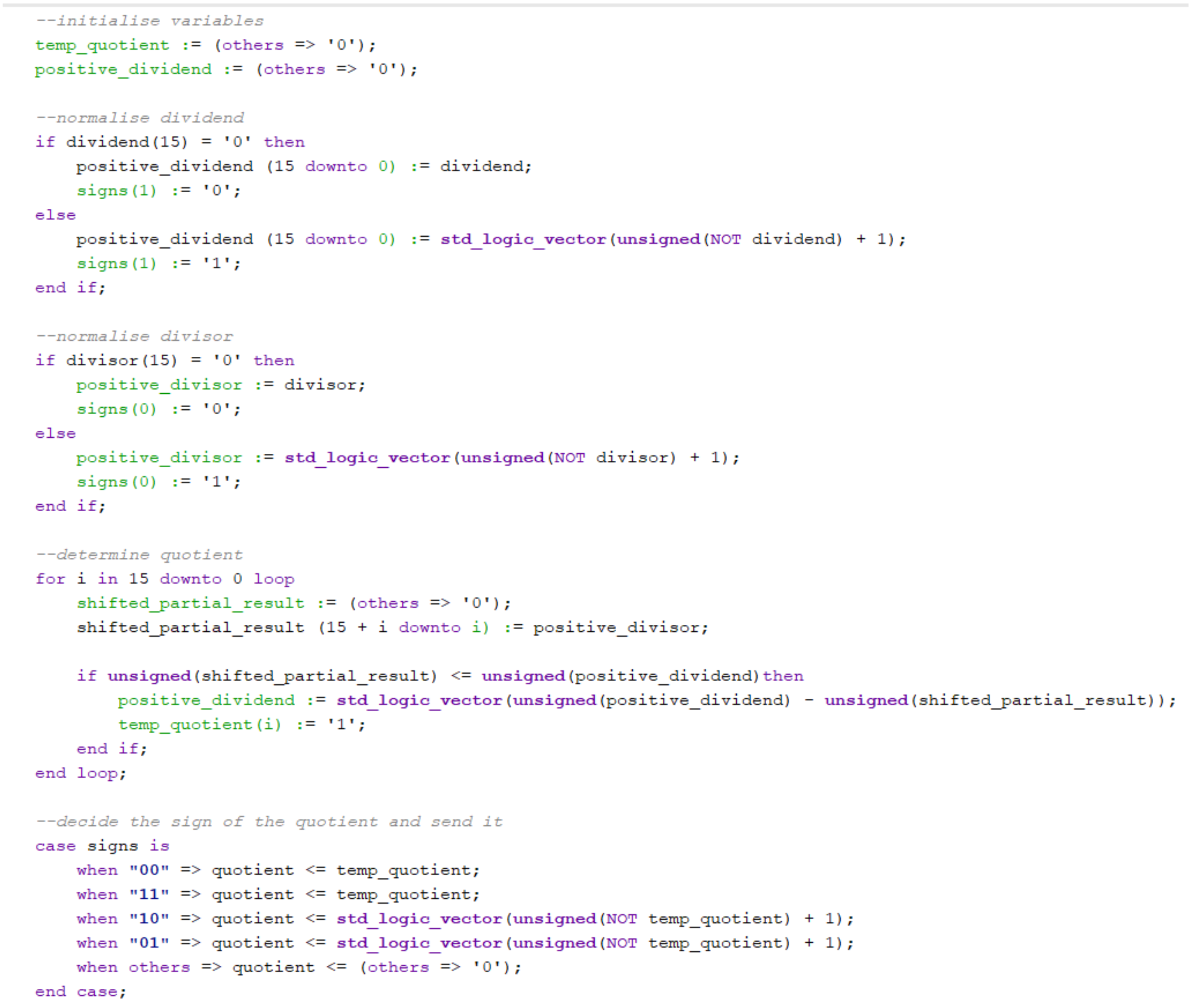
Figure 23. Architecture of multiplier

5.1.12. Division

The algorithm used for the divider is called “restoring division”, also known as “shift and subtract”. It works by repeatedly shifting the divisor left and subtracting it from the dividend until the dividend becomes smaller than the divisor. The number of times the divisor is subtracted is the quotient, with ant remainder discarded. This is how it works:

- Normalize the “dividend” and “divisor” by converting them to their positive equivalents if they are negative, and stores the signs in the “signs” variable.
- Repeatedly shift the “divisor” left and subtracts it from the “dividend”, storing the result back in “dividend”. The number of times the “divisor” is subtracted is recorded in ”temp_quotient”.
- Determine the sign of the “quotient” based on the signs of the “dividend” and “divisor” and returns the appropriate value.

The code is the following:



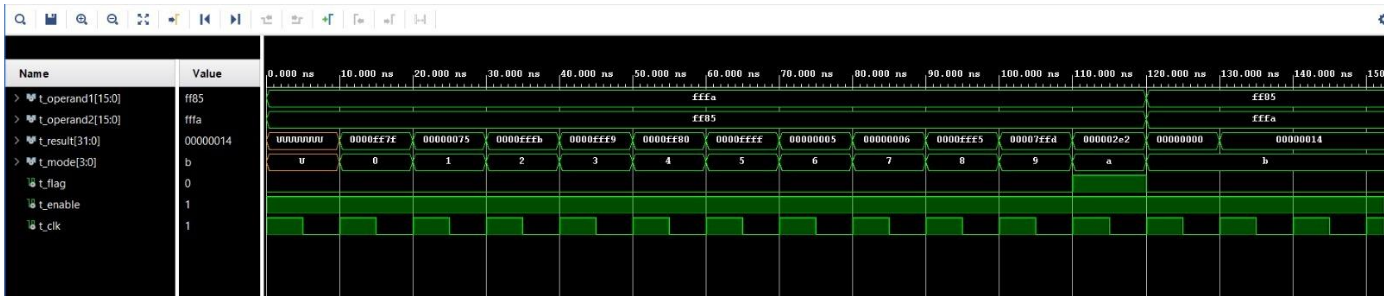
```
--initialise variables
temp_quotient := (others => '0');
positive_dividend := (others => '0');

--normalise dividend
if dividend(15) = '0' then
    positive_dividend (15 downto 0) := dividend;
    signs(1) := '0';
else
    positive_dividend (15 downto 0) := std_logic_vector(unsigned(NOT dividend) + 1);
    signs(1) := '1';
end if;

--normalise divisor
if divisor(15) = '0' then
    positive_divisor := divisor;
    signs(0) := '0';
else
    positive_divisor := std_logic_vector(unsigned(NOT divisor) + 1);
    signs(0) := '1';
end if;

--determine quotient
for i in 15 downto 0 loop
    shifted_partial_result := (others => '0');
    shifted_partial_result (15 + i downto i) := positive_divisor;

    if unsigned(shifted_partial_result) <= unsigned(positive_dividend)then
        positive_dividend := std_logic_vector(unsigned(positive_dividend) - unsigned(shifted_partial_result));
        temp_quotient(i) := '1';
    end if;
end loop;

--decide the sign of the quotient and send it
case signs is
    when "00" => quotient <= temp_quotient;
    when "11" => quotient <= temp_quotient;
    when "10" => quotient <= std_logic_vector(unsigned(NOT temp_quotient) + 1);
    when "01" => quotient <= std_logic_vector(unsigned(NOT temp_quotient) + 1);
    when others => quotient <= (others => '0');
end case;
```

Figure 24. Architecture of divider

6. Tests

The testing of all major components that were implemented was done using Vivado 2020.2 and its simulation environment. A testbench is a component that contains only the component to be tested, named “unit under test” or UUT for short. A testbench has no ports and no utility besides evaluating the behavior of the tested component. As such, internal signals are used to drive the ports of the UUT and visualize the UUT’s reaction.

The testbench for the ALU is as follows with the corresponding encodings for each operation:



7. Conclusion

The goal of this project was to design and build an Arithmetic Logic Unit with 16-bit inputs. There are numerous methods for carrying out the required operations, but I chose the one that I understood and knew how to carry out. In my opinion, the most difficult task was implementing the multiplication and division operations.

Furthermore, the implementation could be done with user interaction in mind by putting the code on the Basys3 FPGA board as initially intended.

References

[1] Functions of the arithmetic logic unit, Computer Science Wiki, Functions of the arithmetic logic unit (ALU) - Computer Science Wiki

[2] JavaTPoint, Addition and Subtraction using Two's complement, Addition and Subtraction using 2's Complement in Digital Electronics - Javatpoint

[3] Gojko Babic, Arithmetic / Logic Unit – ALU Design, Microsoft PowerPoint - CSE675_05_ALUDesign.ppt (ohio-state.edu)

[4] Ann Gordon-Ross, ALU Design: Division and Floating Point, Lec8-division.ppt (ufl.edu)

[5] Jim Lewis, VHDL Math Tricks of the Trade, vhdl_math_tricks_1.pdf (ufl.edu)

[6] Ian Harries, Arithmetic Operations on Binary Numbers, https://www.doc.ic.ac.uk/~eedwards/compsys/arithmetic/index.html

[7] Karen Miller, Two's Complement Multiplication, 2006, https://pages.cs.wisc.edu/~markhill/cs354/Fall2008/beyond354/int.mult.html

[8] Paul Mason, Division using Two's Complement, May 19, 2018, https://paulmason.me/blog/2018-05-19-dividing-binary-numbers-part-2/

[9] GateVidyalay, Ripple Carry Adder, https://www.gatevidyalay.com/tag/16-bit-ripple-carry-adder/