Digital Circuit Design

Design, simulation, and synthesis of a microcontroller using VHDL in a programmable architecture component.

Contents

Project Content

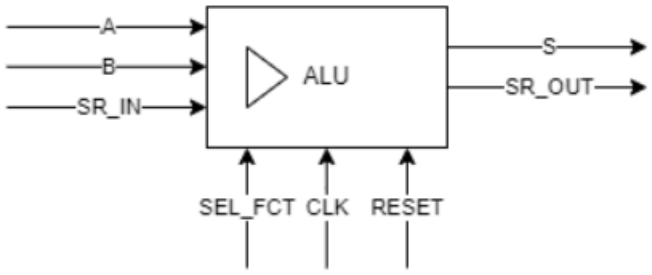
0. Introduction and how to run

The goal of this project is to design, simulate and synthesize a microcontroller using VHDL in a programmable architecture component. The microcontroller will be able to perform arithmetic and logic operations on 4 bits numbers. We decided to split the work into smaller parts : we first designed the ALU, then the buffers, the cache memory and then linked the instructions block to the ALU. Then, our main thought process was to link every little circuit together to create the microcontroller. So, we have little parts that we will explain in the next sections. To keep things easy to understand, we decided to make every component depend on the clock. That means that the calculation might be longer than expected, but it is easier to understand and to simulate.

If you want to try our code, you can modify the instructions.txt file with the RES_OUT .txt file that you want to test and check for the results. You can also modify the A, B and SR_IN variables from the RES_OUT files.

1. ALU

The ALU is a combinational circuit that performs arithmetic and logic operations. Its objective is to do calculations using A, B, Serial inputs and an operation Selector. The result is stored in the S output and a Serial Output. Here is a schematic of the ALU:



We decided that we combined SR_L and SR_R into a single std_logic_vector to make the code more readable. We access them using SR_IN(0) for the right input and SR_IN(1) for the left input.

The code consists of a case statement that checks the value of the operation selector. Depending on the value, the ALU will perform a different operation. The output S has a size of 8

bits, this is because when doing the multiplication, we need to store the result of the multiplication of two 4 bits numbers, which is at maximum is on 8 bits. Let's take a look at a code example:

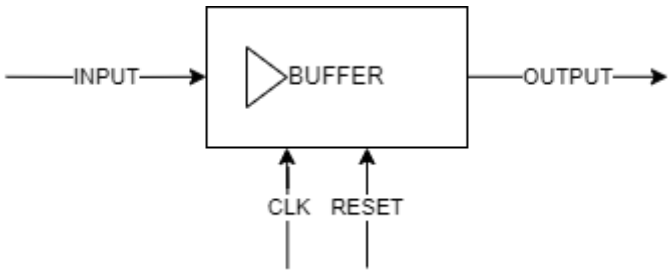
```
when "1011" => -- S = B left shift
  S <= "0000" & B(2 downto 0) & SR_IN(0);
  SR_OUT <= B(3) & '0';
```

This is a left shift of the variable B. So, the output S is composed of the 3 first bits of B and the right serial input SR_IN(0). Then, the SR_OUT is the value that has been popped, which is the 4th bit of B.

The ALU has a clock and a reset input, the clock is used to deliver the output on the falling edge of the clock. The reset is used to reset the ALU to its initial state asynchronously.

2. Buffer

The buffer is a sequential circuit that stores the inputs and outputs of the ALU. It has a clock input and a reset input. Here is a schematic of the buffer:



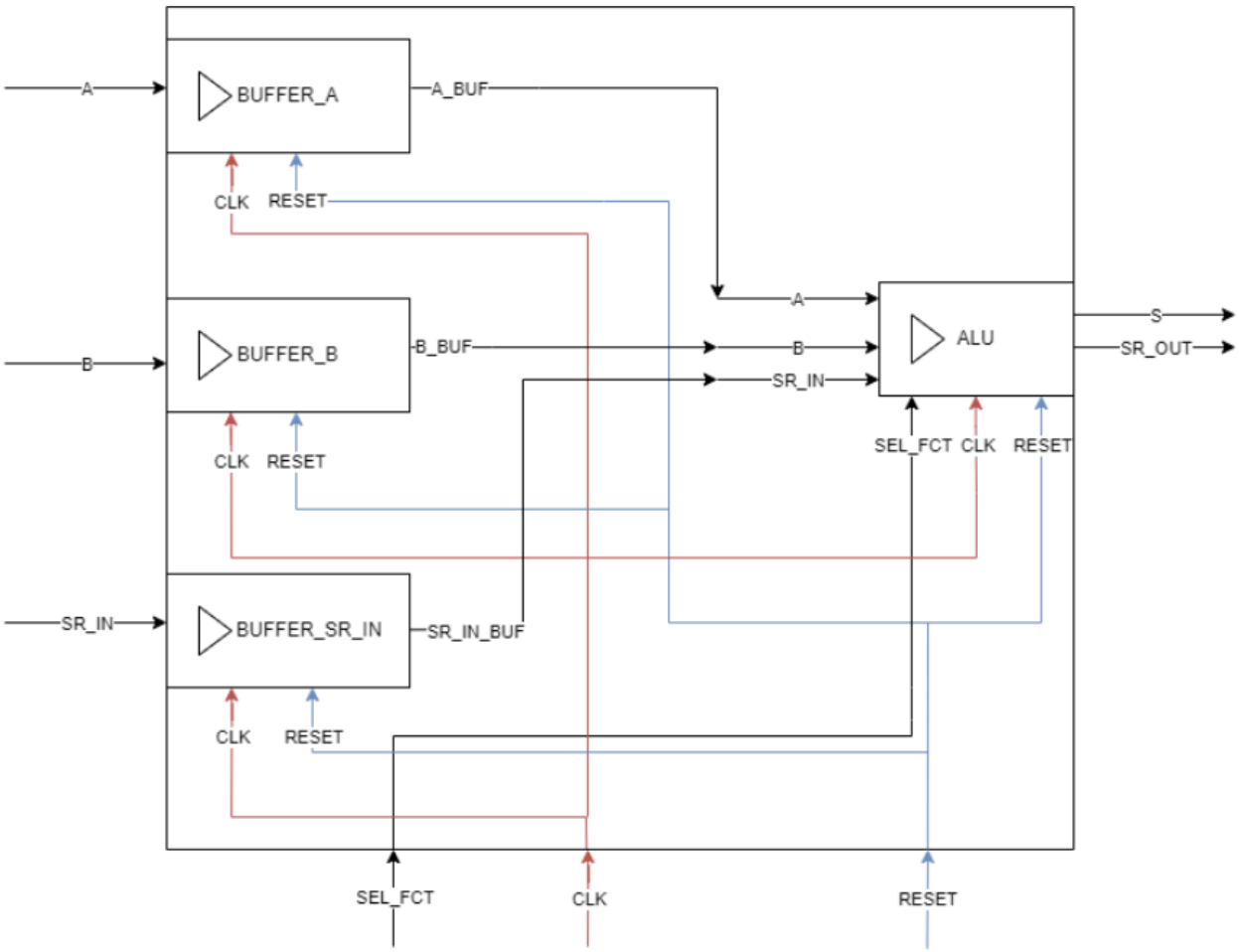
The clock is used to deliver the output on the falling edge of the clock. That means that every time the clock falls, the output will be updated with the input. The reset is used to reset the ALU to its initial state asynchronously. Here is its architecture:

```
architecture buffer_arch of buffer_memory is
begin
  process (clk, reset)
  begin
    if reset = '1' then
      OUTPUT <= (others => '0');
    elsif falling_edge(clk) then
      OUTPUT <= INPUT;
    end if;
  end process;
end architecture buffer_arch;
```

We have 3 buffers, one for the A input, one for the B input and one for the SR_IN input. The size of the register is 4.

3. Buffered ALU

Our next step was to build the ALU with the buffers. We decided to use buffers for SR_IN, A and B. So, we took the output of the buffers and connected it to the input of the ALU. This gives us the following diagram:

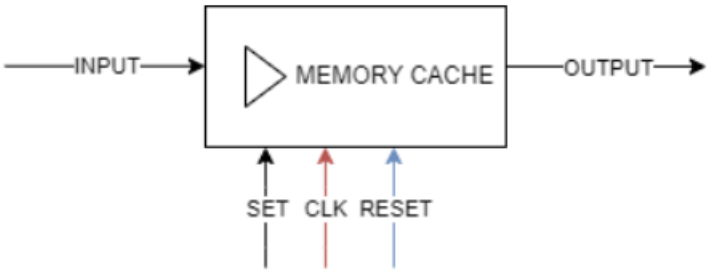


As you can see, they are connected using signals A_BUF, B_BUF and SR_IN_BUF. The outputs S and SR_OUT of the ALU are directly connected to the output of the buffered_alu, they are the same.

The code only consists of the link between the buffers and the ALU.

4. Cache Memory

Now that we have the ALU and the buffers, we can start working on the cache memory. The cache memory is a sequential circuit that stores the instructions and the data. It has a clock input, a SET input and a reset input. The main objective is to store the results of the ALU in the cache memory to use them later in the calculation, or to store the data that we want to use in the ALU.



Here is a schematic of the cache memory:

And here is its architecture:

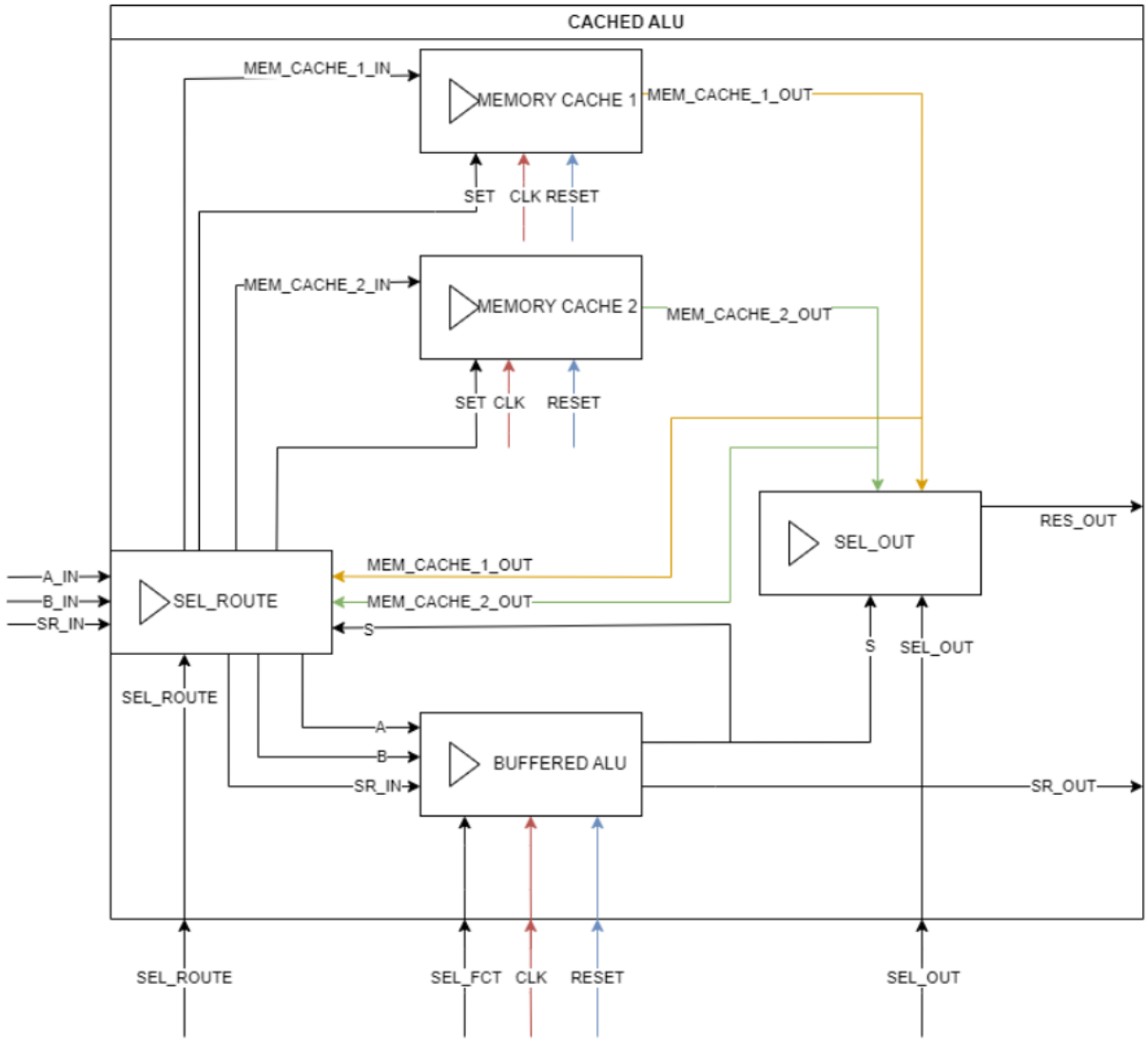
```
architecture rtl of memory_cache is
begin

  process (CLK, reset)
  begin
    if reset = '1' then
      OUTPUT <= (others => '0');
    elsif (falling_edge(CLK) and SET = '1') then
      OUTPUT <= INPUT;
    end if;
  end process;
end architecture rtl;
```

As you can see, the cache memory is very similar to the buffer. The only difference is that the cache memory has a SET input that is used to indicate that we want to store the input in the cache memory.

5. Cached ALU

Using the previous buffered ALU and the cached memory, we can now link them together to create the cached ALU. It is a much more complex circuit than the previous ones, but it is still

very similar to the previous ones. It contains the SEL_ROUTE and SEL_OUT process, which are used to route the inputs and outputs of the ALU to the right place and to select the right output of the ALU. You can find the graph below. Please note that SEL_ROUTE and SEL_OUT are not buffered here, but in the instructions component, described later. Here, their purpose is to route the values to the corresponding location and to correctly select the output when needed.



So, we are first linking the buffered_alu and the two cached memory, and we are creating two processes. Here is the first one, dependent on SEL_ROUTE:

```
process (SEL_ROUTE)
begin
```

```
  MEM_CACHE_1_SET <= '0';
  MEM_CACHE_2_SET <= '0';
  case SEL_ROUTE is
   when "0000" => -- Buffer_A <= A_IN
    ALU_A <= A_IN;
   when "0001" => -- Buffer_B <= B_IN
    ALU_B <= B_IN;
   when "0010" => -- Buffer_A <= MEM_CACHE_1 (4 LSB)
    ALU_A <= MEM_CACHE_1_OUT(3 downto 0);
   when "0011" => -- Buffer_A <= MEM_CACHE_1 (4 MSB)
    ALU_A <= MEM_CACHE_1_OUT(7 downto 4);
   when "0100" => -- Buffer_B <= MEM_CACHE_1 (4 LSB)
    ALU_B <= MEM_CACHE_1_OUT(3 downto 0);
   when "0101" => -- Buffer_B <= MEM_CACHE_1 (4 MSB)
    ALU_B <= MEM_CACHE_1_OUT(7 downto 4);
   when "0110" => -- MEM_CACHE_1 <= S
    MEM_CACHE_1_SET <= '1';
...
```

This process relies on SEL_ROUTE. We decided that when SEL_ROUTE is changing, we are changing the routing following its value. Here you can see code snippets, which routes the inputs and outputs as expected.

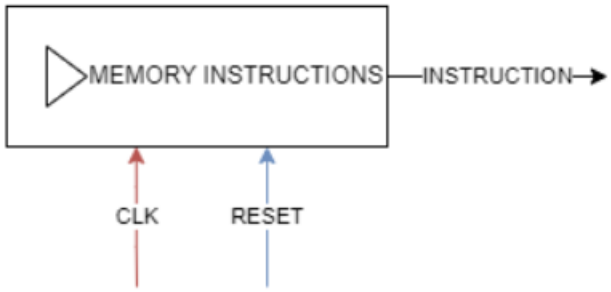
Now, for the SEL_OUT part:

```
 process (CLK)
 begin
  if rising_edge(CLK) then
   case SEL_OUT is
    when "00" => -- No output
     RES_OUT <= (others => '0');
    when "01" => -- RES_OUT <= MEM_CACHE_1
     RES_OUT <= MEM_CACHE_1_OUT;
    when "10" => -- RES_OUT <= MEM_CACHE_2
     RES_OUT <= MEM_CACHE_2_OUT;
    when others => -- RES_OUT <= S
     RES_OUT <= ALU_S;
   end case;
  end if;
 end process;
```

Here, we are using the SEL_OUT to select the right output. We are using the rising edge of the clock to update the output. This is because we want to update the output on the rising edge of the clock, and not on the falling edge. For the duration of the calculation, SEL_OUT will stay at

"00" in order to not output anything. Then, when the calculation is done, we will change SEL_OUT to "11" to output the result of the ALU.

6. Memory Instructions



This part is an interesting one. First, here is the diagram of the memory instructions:

For convenience purposes, we decided that all our instructions will not be contained within a .vhd file, but rather a .txt one. We will then read it and execute the queries. So, we created a file called "instructions.txt" that contains the instructions that we want to execute. Here is an example of the file:

0000000000
0000000100
1100000011

This file doesn't look like much, but it is a list of instructions. Each instruction is 10 bits long, and it is composed of the 4 MSB for the SEL_FCT, the 2 LSB for the SEL_OUT and the 4 other ones for SEL_ROUTE.

So, in the example above, we have 3 instructions. The first one is "0000000000", which means that we want to select the route A_IN > Buffer_A.

Then, we have "0000000100", which corresponds to selecting the route B_IN > Buffer_B.

Finally, we have "1100000011", which means that we want to execute the function "1100", which is the multiplication of A and B, and we want to output the result in the RES_OUT with the "11" instruction.

This is the code example for RES_OUT_1, we will see the other files in detail later.

Here is the code for the memory instructions:

```
 type my_array is array (0 to 127) of std_logic_vector(9 downto 0); -- 128 instructions
 signal instructions : my_array := (others => (others => '0')); -- initialize all to 0
 signal isStartup : boolean := true;
 signal instructions_size : integer := 0; -- size of instructions array
```

First we define a new type called my_array which will contain the instructions that we will read from the file. It has a max size of 128 instructions of 10 bits.

Then, we are instanciating the instructions of type my_array.

We are also defining a boolean that will be used to read the file only once.

Finally we will have the total instructions_size in order to loop them in the main program.

The first part of the program consists of reading the file “instructions.txt”:

```
process (isStartup)
 file F : text open read_mode is "instructions.txt";
 variable L : line;
 variable i : integer := 0; -- index for instructions array
 variable instruction_line : bit_vector(9 downto 0);
begin
 if (isStartup = true) then
  while not endfile(F) loop
   readline(F, L);
   read(L, instruction_line);
   instructions(i) <= to_stdlogicvector(instruction_line);
   i := i + 1;
  end loop;
  instructions_size <= i;
  file_close(F);
 end if;
 isStartup <= false;
end process;
```

So. We are first reading the file instructions.txt. While we are not at the end, we are reading the next line, converting it to a bit_vector and then converting it to a std_logic_vector. This new instruction is stored in the my_array instructions, and the index is incremented. Finally, we are closing the file and setting isStartup to false, so that we don’t read the file again.

Then, here is the other fundamental part of the program, which is the execution of the instructions:

```
process (clk, reset)
 variable i : integer := 0; -- index for instructions array

begin
```

```
 if (reset = '1') then
  i := 0;
 elsif (clk'event and clk = '1') then
  if (i < instructions_size) then
   instruction <= instructions(i);
   i := i + 1;
  end if;
 end if;
end process;
```

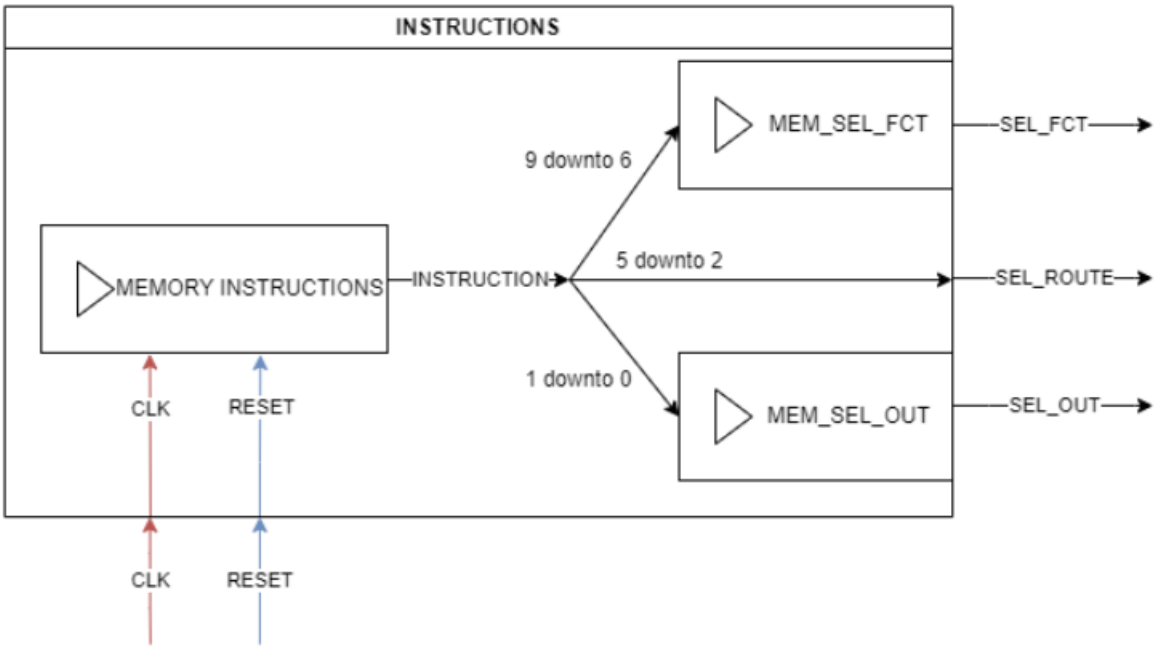
Here, each time the clock is rising, we are reading the next instruction. If reset is set to 0, we are keeping the counter to '0' in order to restart the program. We are reading the instructions while the index is smaller than the instructions_size. This is to avoid reading an instruction that doesn't exist. That way, we can execute the instructions that we want that are stored inside a file apart.

7. Instructions

The instructions contains the memory_instructions and two buffers for SEL_FCT and SEL_OUT. There is no logic in this file, it is just a way to link the different parts of the program. We are splitting the instruction in 3 parts, left part for SEL_FCT, right part for SEL_OUT, and the last part for SEL_ROUTE. Here is the code:

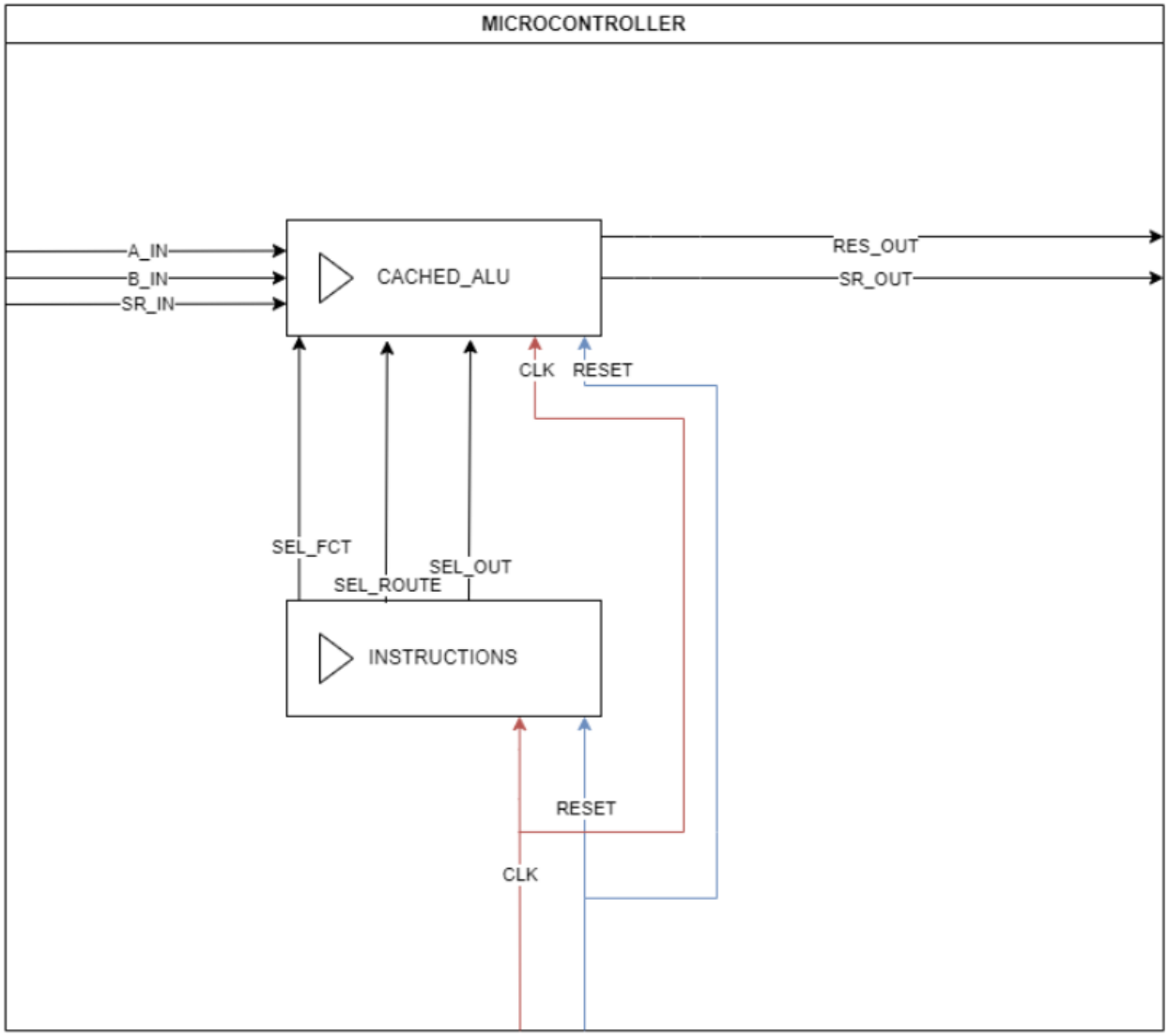
```
 signal sel_fct : std_logic_vector(3 downto 0);
 signal sel_out : std_logic_vector(1 downto 0);
 signal sel_route : std_logic_vector(3 downto 0);
 signal instruction : std_logic_vector(9 downto 0);
```

The graph is the following:



8. Microcontroller

The microcontroller is the main file of the program. It contains the different parts of the program and the clock. It is the file that we will use to simulate the program. It contains as input A, B, the SR_IN, clock and reset. It also contains the output RES_OUT and SR_OUT. Here is the final scheme of the program:



9. RES_OUT

The final tests are done with RES_OUT. We are required to do 3 calculations, which we will explain in detail below. Each time that we are doing a new result, we need to change the file “instructions.txt” in order to execute the new instructions. So, we copy the content of the

res_out file that we want to execute and paste it in the instructions.txt file. Then, we run the simulation and we get the result.

9.1. RES_OUT_1

The first calculation is A * B. We are using the following instructions:

0000000000
0000000100
1100000011

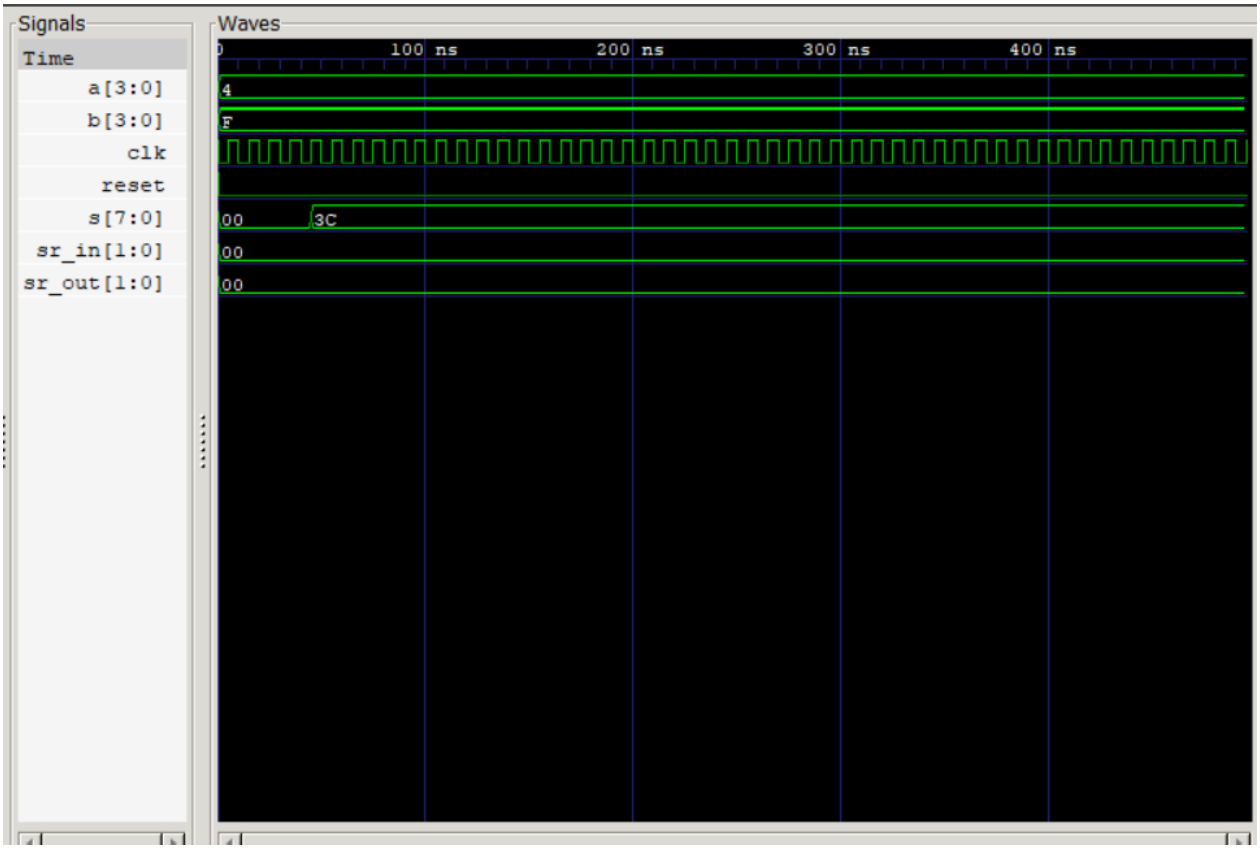
The instructions will be:

1. Routing A

2. Routing B

3. Calculating A * B (SEL_FCT = “1100”) and outputting the result (SEL_OUT = “11”)

With A = “0100” (4) and B = “1111” (15), we should have the result “0011 1100” (60 in decimal or 3C in hexadecimal). Here is what we get using GTKWave:



9.2. RES_OUT_2

The second calculation is (A add. B) xnor A. So, this is a bit more involving. Our plan was to first add A and B, then route the result back to B. We then xor the result with A, reroute the result back to B and finally not the result. So we have the following instructions:

0000000000
0000000100
1110000000
1110000000
0000111000
0111111000
0111111000
0000110000
0011110000
0011110000
0011110011

We are:

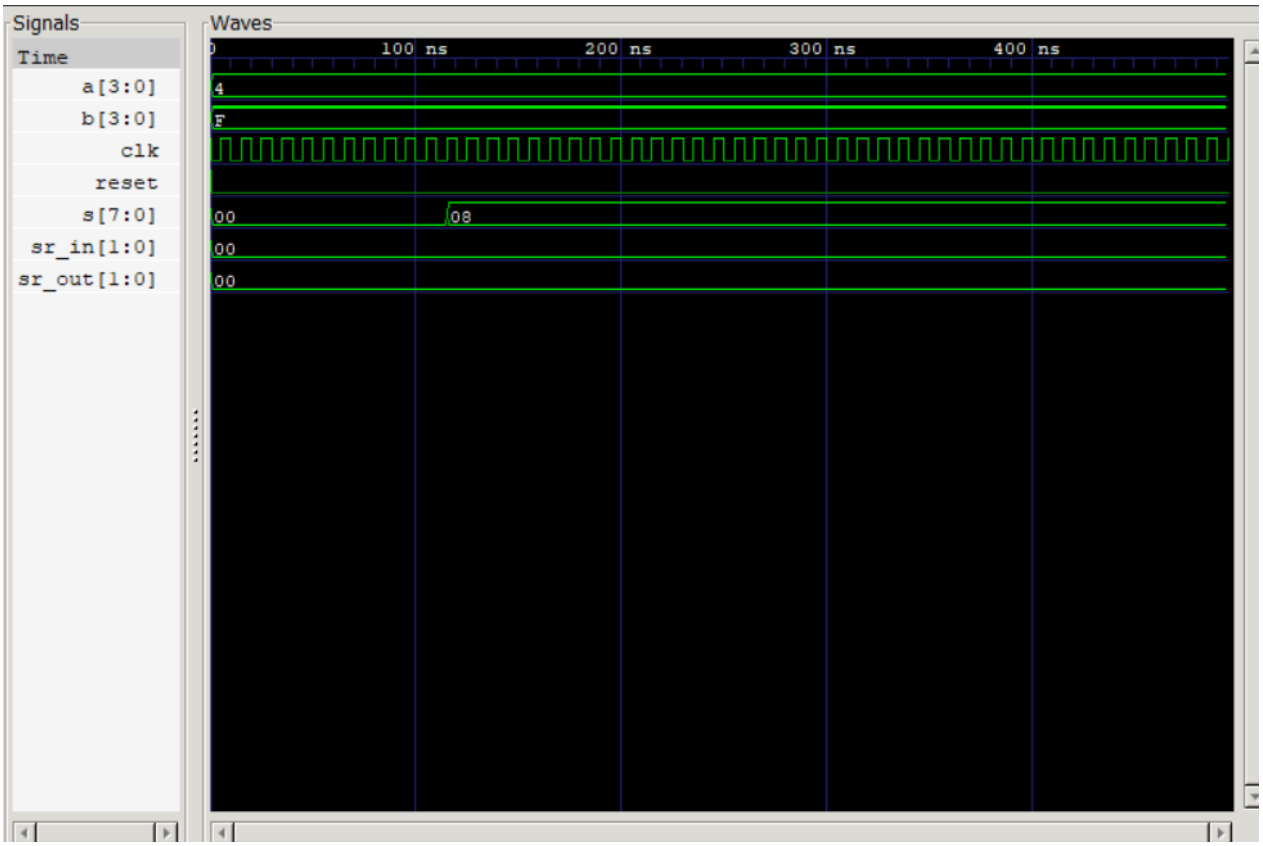
1. Routing A
2. Routing B
3. Calculating A + B (SEL_FCT = “1110”)
4. Waiting for the result
5. Routing the result S to B (SEL_ROUTE = “1110”)
6. Calculating A xor B with the new B (SEL_FCT = “0111”)
7. Waiting for the result
8. Routing the result S to A (SEL_ROUTE = “1100”)
9. Calculating not A (SEL_FCT = “0011”)
10. Waiting for the result
11. Outputting the result

With the same values for A and B, here are the steps:

1. A + B = 4 + 15 = 19 = 0011
2. (A + B) xor A = 0011 xor 0100 = 0111 = 7

3. (A + B) xnor A = not(0111) = 1000 = 8

And here is the result:



9.3. RES_OUT_3

The third calculation is (A0 and B1) or (A1 and B0). This one is much more complicated that the previous ones, since we also only want the last digit. So we are using the following instructions:

0000000000
0000000100
1010000000
1010000000
0000111000
0101000000
0101000000
0000011000
0000000000
0000000100
1000000000

1000000000
0101110000
0101110000
0000111000
0000001000
0110001000
0110001000
0000110000
1001110000
1001101100
1001110000
1001101100
1001110000
1001101100
1000110000
1000101100
1000110000
1000101100
1000110000
1000101111

We are:

1. Routing A
2. Routing B
3. Right shift B (SEL_FCT = “1010”)
4. Waiting for the result
5. Routing the result S to B (SEL_ROUTE = “1110”)
6. Calculating A and B (SEL_FCT = “0101”)
7. Waiting for the result
8. Storing the result in MEM_CACHE_1 (SEL_ROUTE = “0110”)
9. Routing A
10. Routing B
11. Right shift A (SEL_FCT = “1000”)
12. Waiting for the result
13. Routing the result S to A (SEL_ROUTE = “1100”)
14. Calculating A and B (SEL_FCT = “0101”)
15. Waiting for the result
16. Routing the result S to B (SEL_ROUTE = “1110”)
17. Routing MEM_CACHE_1 to A (SEL_ROUTE = “0110”)
18. Calculating A or B (SEL_FCT = “0110”)
19. We want the last digit, so we are going to shift the result 3 times to the left (SEL_FCT = “1001”)
20. And then 3 times to the right (SEL_FCT = “1001”)

21. Finally we are going to output the result (SEL_OUT = “11”)

Since we have SR_IN = “00” throughout the whole process, when shifting all bits in S will be 0 except for the last one. So, we can ignore the other bits and only focus on the last one.

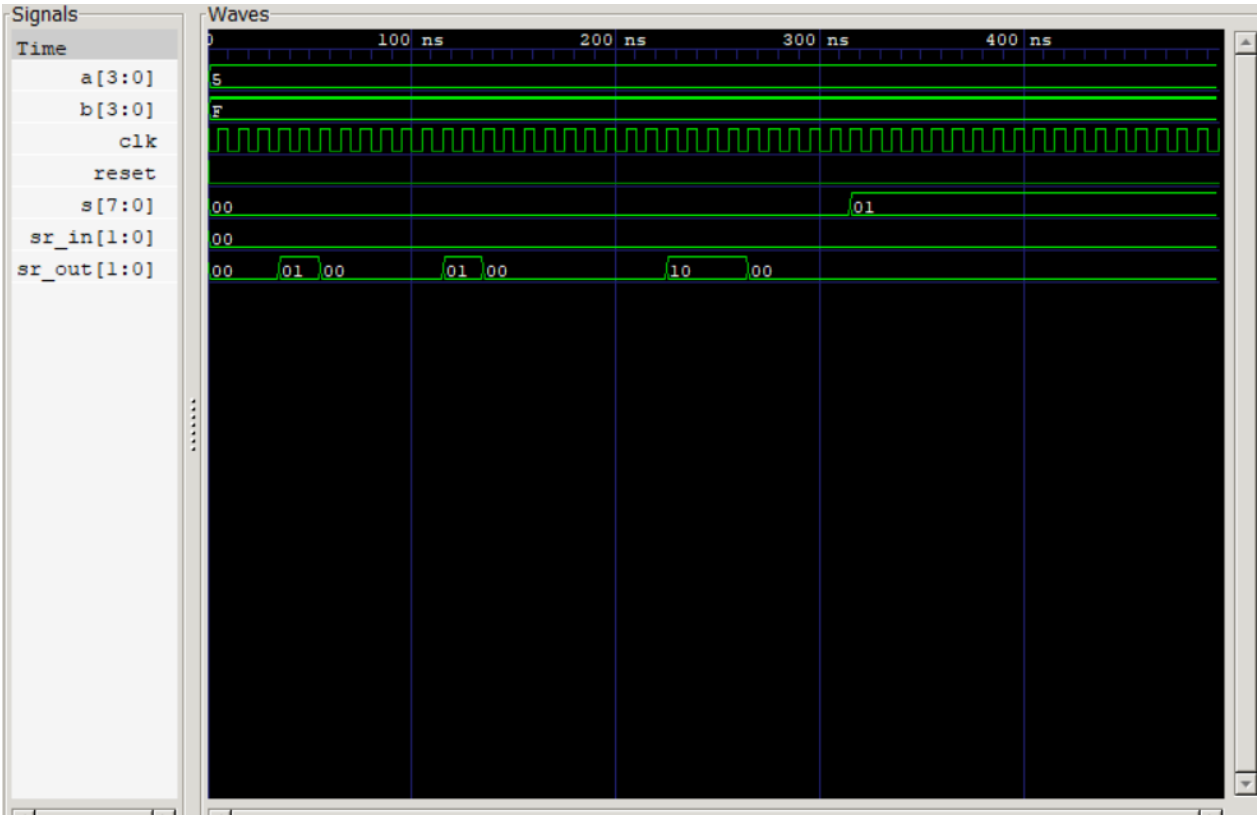
With A = “0101” (5) and B = “1111” (15), we should have:

1. A0 and B1 = 1 and 1 = 1

2. A1 and B0 = 0 and 1 = 0

3. (A0 and B1) or (A1 and B0) = 1 or 0 = 1

So, the result should be “0000 0001”. Here is the result:



Simulation

At first, we wanted to use the EDA Playground website provided by the teacher. However, with the lags and the lack of feedback from code errors, we decided to use the command line. We used the GHDL compiler and the GTKWave viewer. We coded using VSCode and the VHDL LS extension.

Here are the commands that we used to simulate the ALU (for example, we will use the ALU_testbench.vhd file):

```
ghdl -a *.vhd
ghdl -e alu_tb
ghdl -r alu_tb --vcd=alu.vcd --stop-time=1000ms
gtkwave alu.vcd
```

Difficulties encountered.

1. std_logic_vector

We are asked to use the std_logic_vector type to represent the inputs and outputs of the ALU. This means that every time that we do an operation, we need to add numbers of the same size. We had a lot of trouble with this, because we didn't know how to add "0000" bits to the result of the operation. We discovered that we can use the resize() function as such.

```
s <= std_logic_vector(resize(unsigned(a), s'length));
```

However since we will always have the same amount of bits, we thought it was better to use the "0000" & … method, which concatenates the left bits with the right ones.

```
s <= "0000" & a;
```

2. GHDL

We used the same free compiler as EDA Playground. However, what is good with the website is that we don't need to enter any command. So, it was harder to use it. We sometimes forgot that we needed to specify the entity testbench name rather than the file name. Also, since the files needed to be in the same folder for the compiler to see them, the project architecture is a bit different. We need to create a folder for each file and then put the files inside. It is a bit more complicated than the EDA Playground website.

3. Global behavior

A lot of time working on the project has been used to understand what was asked for and how we are going to implement it. It was a hard work, since interconnecting what we already had before needed us to think already about what do we want at the end. Splitting the work into

different little modules was a good idea, but it was hard to understand how to connect them all together.

4. RES_OUT

The final steps of the project were also very hard. To do the required calculations, we needed to perform operations using the ALU. So, we needed to find tricks to, for example, only output the LSB. All these tricks were very hard to find, since doing the calculation by hand was easy but transposing it to the ALU was a hard task. Also, since we followed a strict regulation concerning the clock, that means that we always need to wait for the ALU to process a calculation. That means that each time we are changing SEL_FCT, we need to wait for an additional clock cycle for the result to be output.

Conclusion

We have successfully implemented the ALU. We have also implemented the 3 different RES_OUT asked. We included all test files, test benches and all .vcd files for visualizing the results of the testbench.

Overall, this project helped us understand the VHDL thought process and the coding process better. It was our first time working with a descriptive language and as such, we had a lot of trouble understanding how to code. However, we are now more comfortable with it and we are able to code more easily.