Project report of Digital Circuit Design

Promotion 2024

RÉSUMÉ

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

Authors :

Briac Marchandise

Pablo Sanchez

Houda Zamani

Intervenants : Saber Charbel

Overview

Introduction

For this digital circuit project, we had to model a microcontroller using the VHDL language in a programmable architecture component. Our microcontroller is composed of three components: an arithmetic and logic unit (ALU) allowing to realize logical operations or combination, internal memories of calculation as well as memories of instruction.

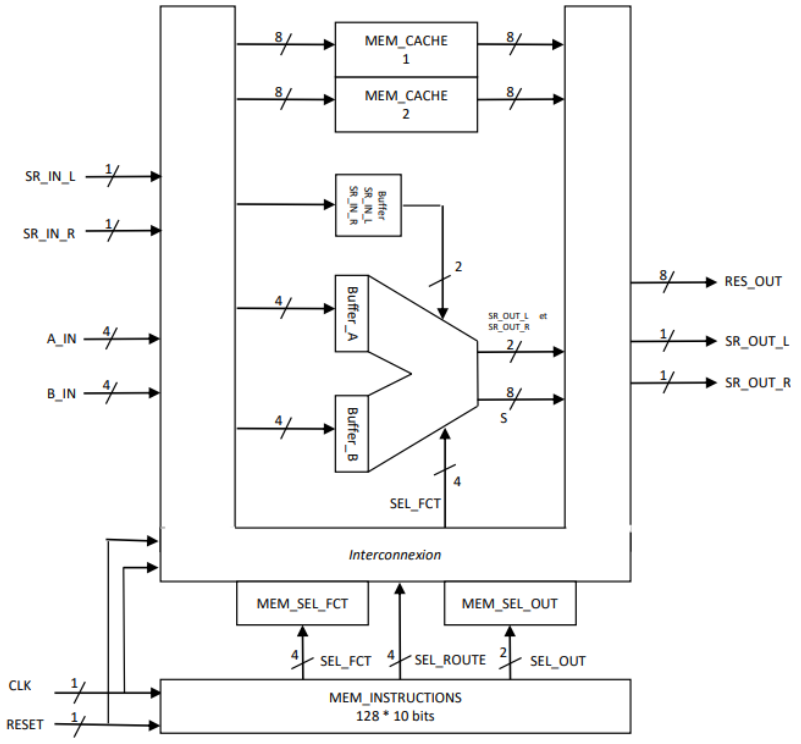
To do this, we will design and analyze a logic circuit (sequential and combinatorial) and a state graph, then perform simulations and syntheses on a programmable component. We will have to distinguish the use cases of a programmable component in relation to CPU/GPU/ASIC/MC and also configure an FPGA while knowing the different steps of an EDA tool. We will apply the essential rules of logic circuit design and try to optimize our programmable components as much as possible. The programming will be done with the VHDL description language.

You can find our entire code by clicking on the following link:
https://www.edaplayground.com/x/7G5N

Introduction to microcontroller

This microcontroller core operates on 4 bits coded in natural signed for the two inputs A_IN and B_IN and on 8 bits coded in natural signed binary for the outputs and the internal memories. We have two inputs for the incoming holds SR_IN_L and SR_IN_R which are stored in a buffer.

The operation instructions are stored in the MEM_INSTRUCTIONS memory as 10-bit sequences. The 4 MSBs designate the operation to be carried out by the ALU, the next 4 bits designate the operations to be carried out in the internal memory. According to this binary sequence, the microcontroller will determine where the different data will be stored. The 2 LSB allow to manage the outputs of the microcontroller. All these operations are carried out on the rising edge of the CLK clock.



Microcontroller to design

I. Functions carried out by the UAL

From basic operations, it is possible to realize all the logical functions and arithmetic operations with the help of automata (successions of instructions). The more "complex" functions are only a succession of simple functions with sometimes the memorization of intermediate results. The operations to be performed by our ALU are given by the sequence of four bits SEL_FCT stored in the memory MEM_SEL_FCT.

SEL_FCT being a sequence of 4 bits, this limits us to 16 operations that can be performed by our ALU. So we can perform a bit shift, basic arithmetic operations (multiplication, addition, subtraction) and logical operations (NOT, XOR, OR, AND).

Here is the list:

SEL_FCT[3]	SEL_FCT[2]	SEL_FCT[1]	SEL_FCT[0]	Significations
0	0	0	0	nop (no operation) S = 0 | SR_OUT_L = 0 et SR_OUT_R = 0
0	0	0	1	S = Déc. droite A sur 4 bits (avec SR_IN_L) | SR_IN_L pour le bit entrant et SR_OUT_R pour le bit sortant
0	0	1	0	S = Déc. gauche A sur 4 bits (avec SR_IN_R) | SR_IN_R pour le bit entrant et SR_OUT_L pour le bit sortant
0	0	1	1	S = Déc. droite B sur 4 bits (avec SR_IN_L) | SR_IN_L pour le bit entrant et SR_OUT_R pour le bit sortant
0	1	0	0	S = Déc. gauche B sur 4 bits (avec SR_IN_R) | SR_IN_R pour le bit entrant et SR_OUT_L pour le bit sortant
0	1	0	1	S = A * B multiplication binaire | SR_OUT_L = 0 et SR_OUT_R = 0
0	1	1	0	S = A + B addition binaire avec SR_IN_R comme retenue d'entrée | SR_OUT_L = 0 et SR_OUT_R = 0
0	1	1	1	S = A + B addition binaire sans retenue d'entrée | SR_OUT_L = 0 et SR_OUT_R = 0
1	0	0	0	S = A – B soustraction binaire | SR_OUT_L = 0 et SR_OUT_R = 0
1	0	0	1	S = A | SR_OUT_L = 0 et SR_OUT_R = 0
1	0	1	0	S = B | SR_OUT_L = 0 et SR_OUT_R = 0
1	0	1	1	S = not A | SR_OUT_L = 0 et SR_OUT_R = 0
1	1	0	0	S = not B | SR_OUT_L = 0 et SR_OUT_R = 0
1	1	0	1	S = A and B | SR_OUT_L = 0 et SR_OUT_R = 0
1	1	1	0	S = A or B | SR_OUT_L = 0 et SR_OUT_R = 0
1	1	1	1	S = A xor B | SR_OUT_L = 0 et SR_OUT_R = 0

So we will design our UAL using the VHDL description language. First of all, we will describe the UAL entity that we name UAL_Core in our Design file.

```
entity ALUCore is
    Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
           B : in STD_LOGIC_VECTOR (3 downto 0);
           SR_IN_L : in STD_LOGIC;
           SR_IN_R : in STD_LOGIC;
           S : out STD_LOGIC_VECTOR (7 downto 0);
           SR_OUT_L : out STD_LOGIC;
           SR_OUT_R : out STD_LOGIC;
           SEL_FCT : in STD_LOGIC_VECTOR (3 downto 0));
end ALUCore;
```

Inputs A and B are vectors of 4 bits each from Buffer_A and Buffer_B respectively. SR_IN_L and SR_IN_R are the incoming holds while SR_OUT_L and SR_OUT_R are the outgoing holds. S is the output on 8 bits, the result of the binary combination between A and B. SEL_FCT is the operation to be performed by the ALU.

After having described the external architecture of the ALU, we must describe its internal architecture. We will list the operations to be performed for the 16 values of SEL_FCT.

Let's take as a first example the operation corresponding to SEL_FCT = '0001' which shifts the bits from A to the right according to the value of the SR_IN_L carry. Bits 4 to 7 will take the value 0 while bit3 will take the value of the carry. The 3 LSBs will take the value of the three MSBs of A. SR_OUT_R will take the value of the LSB of A.

```
case SEL_FCT is
    when "0001" => -- S = Déc. droite A (avec SR_IN_L) | SR_IN_L pour le bit entrant et SR_OUT_R pour le bit sortant
        S(7 downto 4) <= (others => '0');
        S(3) <= SR_IN_L; S(2 downto 0) <= A(3 downto 1);
        SR_OUT_L <= '0'; SR_OUT_R <= A(0);
```

A second case, the one corresponding to the sequence SEL_FCT = '0110' which adds the bits A and B together with SR_IN_R as an incoming carry. To perform this operation, we create vectors My_A_Var and My_B_Var of eight bits long. The MSB (bit 4 to 7) take the value of the sign bit of A. We then add the incoming carry SR_IN_R to the addition of the two bits.

```
when "0110" => -- S = A + B binary addition with SR_IN_R as a carry-in
    My_A_Var(3 downto 0) := A; My_B_Var(3 downto 0) := B;
    My_A_Var(7 downto 4) := (others => A(3)); My_B_Var(7 downto 4) := (others => B(3));
    My_S_Var := My_A_Var + My_B_Var;
    My_S_Var := My_S_Var + ("0000000" & SR_IN_R);
    S <= My_S_Var;
    SR_OUT_L <= '0'; SR_OUT_R <= '0';
```

Finally, it is possible to perform logical operations by simply calling predefined logic gates. Here we have the example of the combination of A or B corresponding to the case where SEL_FCT has the combination 1110.

```
when "1110" => -- S = A or B | SR_OUT_L = 0 and SR_OUT_R = 0
    S(7 downto 4) <= (others => '0');
    S(3 downto 0) <= A or B; SR_OUT_L <= '0'; SR_OUT_R <= '0';
```

II. Buffers

The second step of this project is to create and take care of the internal memories of calculation. It will be a question of creating the buffer A and the buffer B allowing to store the data directly related to the heart of the ALU, i.e. to the arithmetic and logic sub-function. We name this document UALBuffer.

The Buffer_A, Buffer_B memories allow to store the data directly linked to the heart of the ALU, i.e. to the arithmetic and logic sub-function. They are loaded (activated on the rising edge of the clk input) according to the values of the SEL_ROUTE input.

The MEM_SR_IN_L and MEM_SR_IN_R memories are used to store the values of the input holds. They are systematically stored at each rising edge of the CLK clock.

```
entity UALBuffers is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           Buf_A_in : in STD_LOGIC_VECTOR (3 downto 0);
           Buf_B_in : in STD_LOGIC_VECTOR (3 downto 0);
           Mem_1_In : in STD_LOGIC_VECTOR (7 downto 0);
           Mem_2_In : in STD_LOGIC_VECTOR (7 downto 0);
           Buf_SR_IN_L_in : in STD_LOGIC;
           Buf_SR_IN_R_in : in STD_LOGIC;
           CE_Buf_A : in STD_LOGIC; -- Sert a autorisé l'utilisation du buffer A
           CE_Buf_B : in STD_LOGIC;
           CE_Mem_1 : in STD_LOGIC;
           CE_Mem_2 : in STD_LOGIC
           Buf_A_out : out STD_LOGIC_VECTOR (3 downto 0);
           Buf_B_out : out STD_LOGIC_VECTOR (3 downto 0);
           Mem_1_out : out STD_LOGIC_VECTOR (7 downto 0);
           Mem_2_out : out STD_LOGIC_VECTOR (7 downto 0);
           Buf_SR_IN_L_out : out STD_LOGIC;
           Buf_SR_IN_R_out : out STD_LOGIC);
end UALBuffers;
```

First of all we have to write the variables representing our buffers. We have to create a buffer for each part of our microcontroller.

We have as inputs Buf_A_in, Buf_B_in, Buf_SR_IN_L_in, Buf_SR_IN_R_in, Mem_1_In and Mem_2_In where the sequences of bits to be stored enter.

Buf_A_out, Buf_B_out, Buf_SR_IN_L_out, Buf_SR_IN_R_out, Mem_1_out and Mem_2_out are the outputs that store the binary data ready to be restored when the clock allows it.

CE_Buf_A, CE_Buf_B_ CE_Mem_1 and CE_Mem_2 are one-bit variables that, depending on their value, will indicate in which memory the information should be stored.

```
begin
    BufAProc : process (reset, clk, CE_Buf_A)
    begin
        if reset = '1' then
            Buf_A_out <= (others => '0');
             elsif (rising_edge(clk) and CE_Buf_A = '1') then -- rising edge indique l'evenement "front montant" de la clk
            Buf_A_out <= Buf_A_in;
        end if;
    end process;

    BufBProc : process (reset, clk, CE_Buf_B)
    begin
        if reset = '1' then
            Buf_B_out <= (others => '0');
        elsif (rising_edge(clk) and CE_Buf_B = '1') then
            Buf_B_out <= Buf_B_in;
        end if;
    end process;
```

Above, we have the architecture of Buffer_A and Buffer_B which will store the binary sequences coming from A_IN and B_IN. These processes take as parameters the variables reset, clk and CE_Buf_A or CE_Buf_B.

We note the presence of the reset parameter which, when its value is equal to 1, resets all four bits of the variable Buf_A_Out to 0. As we can see, the storage of the variable Buf_A_in "in" the variable Buf_A_out is carried out on the rising edge of the clock.

III. Interconnections and data routing

The third step of the project will be to take care of what will activate our previous operations. In order to know when and how our buffers will be loaded, we need to create functions that take care of the SEL_ROUTE values that will allow us to load the buffers on the rising edge of the CLK input. SEL_Route will be equal to the 5 to 2 of the ten bit instruction sequence.

Here is the list of operations to be performed according to the different binary sequences of Sel_Route. This variable being coded on 4 bits, there are thus 16 possible storage operations.

SEL_ROUTE[3]	SEL_ROUTE[2]	SEL_ROUTE[1]	SEL_ROUTE[0]	Meanings
0	0	0	0	Input A_IN storage in Buffer_A
0	0	0	1	Input B_IN storage in Buffer_B
0	0	1	0	S storage in Buffer_A (4 LSB bits)
0	0	1	1	S storage in Buffer_A (4 MSB bits)
0	1	0	0	S storage in Buffer_B (4 LSB bits)
0	1	0	1	S storage in Buffer_B (4 MSB bits)
0	1	1	0	S storage in MEM_CACHE_1
0	1	1	1	S storage in MEM_CACHE_2
1	0	0	0	MEM_CACHE_1 storage in Buffer_A (4 LSB bits)
1	0	0	1	MEM_CACHE_1 storage in Buffer_A (4 MSB bits)
1	0	1	0	MEM_CACHE_1 storage in Buffer_B (4 LSB bits)
1	0	1	1	MEM_CACHE_1 storage in Buffer_B (4 MSB bits)
1	1	0	0	MEM_CACHE_2 storage in Buffer_A (4 LSB bits)
1	1	0	1	MEM_CACHE_2 storage in Buffer_A (4 MSB bits)
1	1	1	0	MEM_CACHE_2 storage in Buffer_B (4 LSB bits)
1	1	1	1	MEM_CACHE_2 storage in Buffer_B (4 MSB bits)

Here is the external view of the entity UAL_SEL_Route.

```
entity UALSELROUTE is
    Port (
            SEL_ROUTE : in STD_LOGIC_VECTOR (3 downto 0);
            RES_OUT : in STD_LOGIC_VECTOR (7 downto 0);
            A_IN : in STD_LOGIC_VECTOR (3 downto 0);
            B_IN : in STD_LOGIC_VECTOR (3 downto 0);

            Buf_A_out : in STD_LOGIC_VECTOR (3 downto 0);
            Buf_B_out : in STD_LOGIC_VECTOR (3 downto 0);

            Mem_1_out : in STD_LOGIC_VECTOR (7 downto 0);
            Mem_2_out : in STD_LOGIC_VECTOR (7 downto 0);

            Buf_A_in : out STD_LOGIC_VECTOR (3 downto 0);
            Buf_B_in : out STD_LOGIC_VECTOR (3 downto 0);
            Mem_1_In : out STD_LOGIC_VECTOR (7 downto 0);
            Mem_2_In : out STD_LOGIC_VECTOR (7 downto 0);

            CE_Buf_A : out STD_LOGIC;
            CE_Buf_B : out STD_LOGIC;
            CE_Mem_1 : out STD_LOGIC;
            CE_Mem_2 : out STD_LOGIC
    );
end UALSELROUTE;
```

- SEL_ROUTE : variable on 4 bits allowing to choose the operation which will be realized
- RES_OUT : 8 bits variable containing the output results - A_IN and B_IN : 4 bits input variables
- Buf_A_in, Buf_B_in, Buf_SR_IN_L_in, Buf_SR_IN_R_in, Mem_1_In and Mem_2_In are the input variables allowing to keep in memory our initial values.
- Buf_A_out, Buf_B_out, Buf_SR_IN_L_out, Buf_SR_IN_R_out, Mem_1_out and Mem_2_out are the output variables allowing to load the buffers when the clock allows it.
- CE_Buf_A, CE_Buf_B_ CE_Mem_1 and CE_Mem_2 are variables that will give us the authorizations

Here is the MySelRouteproc process which lists the 16 different actions to manage the internal memory of the microcontroller.

```
MySelRouteProc : process (SEL_ROUTE, S, A, B, Buf_A_out, Buf_B_out, Mem_1_out, Mem_2_out)
begin
    case SEL_ROUTE is
        when "0000" => -- Stockage de l'entrée A_IN dans Buffer_A
            CE_Buf_A <= '1'; CE_Buf_B <= '0'; CE_Mem_1 <= '0'; CE_Mem_2 <= '0';
            Buf_A_in <= A; Buf_B_in <= (others => '0'); Mem_1_In <= (others => '0'); Mem_2_In <= (others => '0');

        when "0001" => -- Stockage de l'entrée B_IN dans Buffer_B
            CE_Buf_A <= '0'; CE_Buf_B <= '1'; CE_Mem_1 <= '0'; CE_Mem_2 <= '0';
            Buf_A_in <= (others => '0'); Buf_B_in <= B; Mem_1_In <= (others => '0'); Mem_2_In <= (others => '0');

        when "0010" => -- Stockage de S dans Buffer_A (4 bits de poids faibles)
            CE_Buf_A <= '1'; CE_Buf_B <= '0'; CE_Mem_1 <= '0'; CE_Mem_2 <= '0';
            Buf_A_in <= S(3 downto 0); Buf_B_in <= (others => '0'); Mem_1_In <= (others => '0'); Mem_2_In <= (others => '0');


when "0111" => -- Stockage de S dans MEM_CACHE_2
    CE_Buf_A <= '0'; CE_Buf_B <= '0'; CE_Mem_1 <= '0'; CE_Mem_2 <= '1';
    Buf_A_in <= (others => '0'); Buf_B_in <= (others => '0'); Mem_1_In <= (others => '0'); Mem_2_In <= S;
```

If we take for example when SEL_ROUTE has the value "0000", our table tells us that the microcontroller must directly store the incoming value A in the Buffer_A. The program will therefore first set the value of our CE_Buf_A to 1 in order to authorize the use of Buffer_A. All the other buffer control variables remain at 0.

When SEL_Route takes the instruction sequence '0001', it is a similar storage operation but this time it is the incoming value of B that is stored in buffer B.

When we have the instruction sequence '0010' we always initialize CE_Buf_A to 1 and the others to 0 and then we store in Buffer_A the 4 LSBs that we obtain at the output S of the ALU.

Last example, when we have the instruction '0111' at the input of SEL_Route, the binary sequence at the output of ALU S is stored in cache number 2. We can see that the variable CE_Mem_2 is initialized to 1 and that the 8 bits of S are stored in Mem_2_in.

Next, we make the entity SEL_OUT, which allows us to manage the output of the results of the processing of the binary sequences entered in the microcontroller as a result of the operations performed. Here, SEL_OUT is defined in terms of the last two 2 bits of the 10-bit instruction sequence. The value of these two bits is first stored in the memory Mem_Sel_Out. Depending on the values taken by the variable, we will choose which values should be taken by the 8 bits of RES_OUT. To program these logical operations, we base ourselves on the table below:

SEL_OUT[1]	SEL_OUT[0]	Significations
0	0	Aucune sortie : RES_OUT = 0
0	1	RES_OUT = MEM_CACHE_1
1	0	RES_OUT = MEM_CACHE_2
1	1	RES_OUT = S

As we can see, when SEL_OUT is equal to '00', there is no output. Otherwise, we take either the 8-bit sequences stored in one of the two cache memories or directly the value at the output of UAL.

We therefore define the UALSELOUT entity as follows

```
entity UALSELOUT is
    Port (
        SEL_OUT : in STD_LOGIC_VECTOR (1 downto 0);
        S : in STD_LOGIC_VECTOR (7 downto 0);
        Mem_1_out : in STD_LOGIC_VECTOR (7 downto 0);
        Mem_2_out : in STD_LOGIC_VECTOR (7 downto 0);
        Res_out : out STD_LOGIC_VECTOR (7 downto 0)
    );
end UALSELOUT;
```

Then we obtain the architecture corresponding to the following table:

```
architecture UALSELOUT_Arch of UALSELOUT is

begin

    MySelRouteProc : process (SEL_OUT, S, Mem_1_out, Mem_2_out)
    begin
        case SEL_OUT is
            when "00" => -- Aucune sortie : RES_OUT = 0
                Res_out <= (others => '0');
            when "01" => -- RES_OUT = MEM_CACHE_1
                Res_out <= Mem_1_out;
            when "10" => -- RES_OUT = MEM_CACHE_2
               Res_out <= Mem_2_out;
            when others => -- RES_OUT = S
                Res_out <= S;
        end case;
    end process;

end UALSELOUT_Arch;
```

IV. Internal computing memory

Then, the fourth step of the project is to design the internal memory units where the operations to be performed by the microcontroller are recorded. First of all, we have MEM_SEL_FCT which allows to store the arithmetic or logic function to be performed. This synchronous memory is loaded at each rising edge of the clock. It is used to store the four operational bits intended for the ALU. Then we have the MEM_SEL_OUT memory which allows to store the 2 input bits of the SEL_OUT entity. This one is also loaded on the rising edge of the clock. We design the external view and architecture of these components in a file named CMDBuffers.

```
entity CMDBuffers is
Port ( clk : in STD_LOGIC;
       reset : in STD_LOGIC;
       Buf_SEL_FCT_in : in STD_LOGIC_VECTOR (3 downto 0);
       Buf_SEL_OUT_In : in STD_LOGIC_VECTOR (1 downto 0);
       Buf_SEL_FCT_out : out STD_LOGIC_VECTOR (3 downto 0);
       Buf_SEL_OUT_out : out STD_LOGIC_VECTOR (1 downto 0)
     );
end CMDBuffers;
```

Voici donc l'architecture du CMDBuffers :

```
architecture CMDBuffers_Arch of CMDBuffers is
begin

    Buf_SEL_FCT_Proc : process (reset, clk)
    begin
        if reset = '1' then
            Buf_SEL_FCT_out <= (others => '0');
        elsif (rising_edge(clk)) then
            Buf_SEL_FCT_out <= Buf_SEL_FCT_in;
        end if;
    end process;

    Buf_SEL_OUT_Proc : process (reset, clk)
    begin
        if reset = '1' then
            Buf_SEL_OUT_out <= (others => '0');
        elsif (rising_edge(clk)) then
            Buf_SEL_OUT_out <= Buf_SEL_OUT_In;
        end if;
    end process;

end CMDBuffers_Arch;
```

Here we have two synchronous processes. At each rising edge of the clock, the output of the buffer takes the value of the input. When the clock is in its low state, then the output variable is reset to '0000'. We note the presence of the reset variable in each process which, when its value is 1, resets SEL_OUT_MEM and SEL_FCT_MEM.

V. Instruction memory

The fifth step of our project is to realize the instruction memory. The MEM_INSTRUCTIONS memory allows to store the successive instructions to be carried out. Its pointer is systematically indented at each rising edge of the CLK clock. This memory must contain 128 instruction sequences of 10 bits. We will enter locally the binary sequences in order to test our microcontroller. We have to implement three specific functions: the multiplication of A by B, then (A + B) XNOR A and (A0 and B1) or (A1 and B0).

We will therefore first write in a VHDL file the sequences to be realized.

```
entity INSTRMemory is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           INSTR_in : in STD_LOGIC_VECTOR (9 downto 0);
           INSTR_out : out STD_LOGIC_VECTOR (9 downto 0);
           INSTR_addr : in STD_LOGIC_VECTOR (6 downto 0);
           INSTR_CE : in STD_LOGIC;
           R1W0 : in STD_LOGIC);
end INSTRMemory;

architecture Behavioral of INSTRMemory is

    Type data_memory IS ARRAY (0 to 127) of std_logic_vector (9 downto 0);
    SIGNAL INSTR_memory : data_memory;
```

As we can see above, the INSTRMemory entity is defined as follows. An input variable and an output variable with a length of 10 bits to store the instruction for the microcontroller. The input INSTR_addr corresponds to the number of the instruction to realize. This one is coded on 7 bits because there are 128 instructions, thus 128 addresses.

In the architecture, we define an array named data_memory of length 128 gathering the whole of the instructions.

```
begin

    MyReadWriteProc : process (clk, INSTR_CE, R1W0, reset)
    begin
        if (reset = '1') then
            for i in 0 to 127 loop
                INSTR_memory(i) <= "0000000000";
                INSTR_memory(i) <= INSTR_memory(i);
            end loop;
        elsif falling_edge(clk) and INSTR_CE = '1' then
            if (R1W0 = '1') then
                INSTR_out <= INSTR_memory(to_integer(unsigned(INSTR_addr)));
            else
                INSTR_memory(to_integer(unsigned(INSTR_addr))) <= INSTR_in;
                INSTR_out <= INSTR_in;
            end if;
        end if;
    end process;
```

We use this process which allows to read at each new rising edge of the clock the next instruction.

```
INSTR_out <= INSTR_memory(to_integer(unsigned(INSTR_addr))) when falling_edge(clk) and INSTR_CE = '1';

-- Default Value - A mult. B
INSTR_memory(0) <= "0000000000"; --| no op            | A -> Buf A    | 0     |
INSTR_memory(1) <= "0101000100"; --| A * B           | B -> Buf B    | 0     |
INSTR_memory(2) <= "0000011000"; --| no op            | S -> Mem 1    | 0     |
INSTR_memory(3) <= "0000000001"; --| no op            | A -> Buf A    | Mem 1 |
INSTR_memory(4) <= "0000000100";
INSTR_memory(5) <= "0000000101";
INSTR_memory(6) <= "0000000110";
INSTR_memory(7) <= "0000000111";
INSTR_memory(8) <= "0000001000";
INSTR_memory(9) <= "0000001001";
```

Here is our series of instructions in order to multiply A by B. Instruction 0 does not perform any operation in the ALU but stores the value of A_IN in Buffer A. It is during instruction 1 that we indicate the operation to be performed by the ALU. In the same instruction sequence, the value of SEL_Route that the value of B_IN must be stored in Buffer_B. The 4 MSBs corresponding to SEL_FCT = 0101 indicate that this is a multiplication. The result of the operation will be stored in buffer_B as indicated by the sequence of SEL_ROUTE equal to 0001.

Then, in instruction number 2, we indicate that the result in output S of the ALU must be stored in cache 1 (SEL_ROUTE = 0110). Finally, instruction number 3 allows the output of the final result from the microcontroller: RES_OUT takes the value of MEM_CACHE_1

```
-- Default Value - (A add. B) xnor A
INSTR_memory(32) <= "0000000000"; --| no op            | A -> Buf A    | 0     |
INSTR_memory(33) <= "0110011100"; --| A add B         | B -> Buf B    | 0     |
INSTR_memory(34) <= "1111010000"; --| A xor S         | S -> Buf B    | 0     |
INSTR_memory(35) <= "1100010000"; --| not B (xnor)    | S -> Buf B    | 0     |
INSTR_memory(36) <= "0000011000"; --| no op            | S -> Mem 1    | 0     |
INSTR_memory(37) <= "0000000001"; --| no op            | A -> Buf A    | Mem 1 |
INSTR_memory(38) <= "0000100110";
INSTR_memory(39) <= "0000100111";
INSTR_memory(40) <= "0000101000";
INSTR_memory(41) <= "0000101001";
INSTR_memory(42) <= "0000101010";
INSTR_memory(43) <= "0000101011";
```

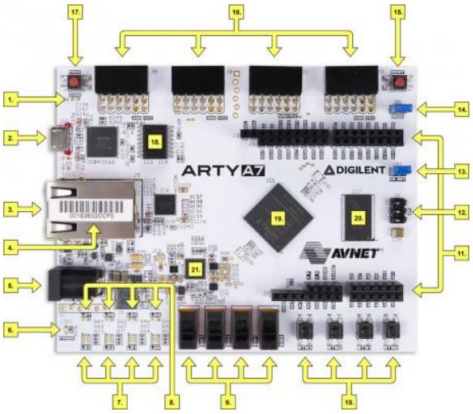
In order to perform the (A+B) XNOR A, we must have the above sequence. First, we load the A_IN into buffer A. At instruction 33, we load B_IN into buffer B and add A and B (SEL_FCT = 0110). Then, as requested in the instruction, we store the 4 LSB of S in buffer B (SEL_ROUTE = 0100) and then we carry out A XOR S (S being contained in buffer B). Then, the same memory operation as before, we store the 4 LSBs in buffer B and we perform the NOT operation (SEL_FCT = 1100). As before for the multiplication, we store the result at the output of UAL S in cache memory 1 and then at instruction 37, we indicate that the result of its RES_OUT operations takes the value stored in MEM_CACHE_1 (SEL_OUT = 01).

```
-- Default Value - (A0.B1 + A1.B0)
INSTR_memory(64) <= "0000000000"; --| no op              | A -> Buf A     | 0      |
INSTR_memory(65) <= "0001000100"; --| déc droite A      | B -> Buf B     | 0      |
INSTR_memory(66) <= "1101001000"; --| A1 and B0         | S -> Buf A     | 0      |
INSTR_memory(67) <= "0011011000"; --| déc droite B      | S -> Mem 1     | 0      |
INSTR_memory(68) <= "0000010000"; --| no op             | S -> Buf B     | 0      |
INSTR_memory(69) <= "1101000000"; --| A0 and B1         | A -> Buf A     | 0      |
INSTR_memory(70) <= "0000010000"; --| no op             | S -> Buf B     | 0      |
INSTR_memory(71) <= "1110000100"; --| Buf A or Buf B    | Mem1 -> Buf A  | 0      |
INSTR_memory(72) <= "0000011000"; --| no op             | S -> Mem 1     | 0      |
INSTR_memory(73) <= "0000000001"; --| no op             | A -> Buf A     | Mem1   |
INSTR_memory(74) <= "0001001010";
INSTR_memory(75) <= "0001001011";
INSTR_memory(76) <= "0001001100";
INSTR_memory(77) <= "0001001101";
```

Our last operation must give as output RES_OUT_3 = (A0 and B1) or (A1 and B0) (RES_OUT_3 on the least significant bit). First, we load A1 into the buffer in buffer_A. Then, at the same time as we load B0 into buffer_B, we perform a right shift of the bits of A in order to manage the sign. Then, we carry out A1 AND B0 then we store the output value in the buffer A. Then we perform the right shift of B and store the result in cache 1. Then we perform the operation A0 and B1 (A0 has been loaded in the parallel buffer) and the result is stored in buffer B. After having fetched the result of our first operation stored in cache 1 (A1 and B0), we perform the OR operation between the results of these two operations. Once the result is stored in cache 1, we return it (RES_OUT = 01).

After having programmed the correct sequences of instructions, we can move on to the test stage. The goal is to program the Xilinx Artix-35T FPGA board provided by the teacher and program it so that the LEDs return the results of our operations.

The values of A and B will be equal during our test.



Callout	Description	Callout	Description	Callout	Description
1	FPGA programming DONE LED	8	User RGB LEDs	15	chipKIT processor reset
2	Shared USB JTAG / UART port	9	User slide switches	16	Pmod connectors
3	Ethernet connector	10	User push buttons	17	FPGA programming reset button
4	MAC address sticker	11	Arduino/chipKIT shield connectors	18	SPI flash memory
5	Power jack for optional external supply	12	Arduino/chipKIT shield SPI connector	19	Artix FPGA
6	Power good LED	13	chipKIT processor reset jumper	20	Micron DDR3 memory
7	User LEDs	14	FPGA programming mode	21	Dialog Semiconductor DA9062 power supply

In the Top Level file we will initialize the clock to 100 MHz so that it activates and deactivates. And we are going to enter values for the LEDS in order to have different combinations and check the proper functioning of the microcontroller.

```
led <= My_Res_Out (7 downto 4);
led0_r <= My_Res_Out (0); led0_b <= '0';
led1_r <= My_Res_Out (1); led1_g <= '0'; led1_b <= '0';
led2_r <= My_Res_Out (2); led2_g <= '0'; led2_b <= '0';
led3_r <= My_Res_Out (3); led3_g <= '0'; led3_b <= '0';
```

Here is the general entity of our MCU.

```
entity MCU_PRJ_2021_TopLevel is
    Port (
        CLK100MHZ : in STD_LOGIC;
        sw : in STD_LOGIC_VECTOR(3 downto 0);
        btn : in STD_LOGIC_VECTOR(3 downto 0);
        led : out STD_LOGIC_VECTOR(3 downto 0);
        led0_r : out STD_LOGIC; led0_g : out STD_LOGIC; led0_b : out STD_LOGIC;

        led1_r : out STD_LOGIC; led1_g : out STD_LOGIC; led1_b : out STD_LOGIC;
        led2_r : out STD_LOGIC; led2_g : out STD_LOGIC; led2_b : out STD_LOGIC;

        led3_r : out STD_LOGIC; led3_g : out STD_LOGIC; led3_b : out STD_LOGIC
    );
end MCU_PRJ_2021_TopLevel;
```

In its architecture, we are going to implement all the components that we have defined before, that is to say the CMDBuffers, the UALCore, the Instruction memory, the UALBuffers, the UALSEL_Out, the UAL_SEL_Route .

Here is the general process of the MCU. Here, we only see the idle state.

```
MyAlgoProc : process (btn(3 downto 0), CLK100MHZ)
begin
    if(btn(0) = '1') then
        MyCounter1 <= (others => '0'); led0_g <= '0';
        FSM_Main <= s_Idle;
    elsif rising_edge(CLK100MHZ) then
        case FSM_Main is
            when s_Idle =>
                if(btn(3) = '1') then
                    MyCounter1 <= "1000000"; FSM_Main <= s_Funct_3; led0_g <= '0';
                elsif (btn(2) = '1') then
                    MyCounter1 <= "0100000"; FSM_Main <= s_Funct_2; led0_g <= '0';
                elsif (btn(1) = '1') then
                    MyCounter1 <= (others => '0'); FSM_Main <= s_Funct_1; led0_g <= '0';
                else
                    MyCounter1 <= (others => '0'); FSM_Main <= s_Idle; led0_g <= '0';
                end if;
```

We were able to test our ALU by adding and multiplying bits.

Unfortunately an implementation error did not allow us to test the general operation of our MCU when we returned from the project classes.

Conclusion

Unfortunately, we were unable to perform the MCU simulation to test the validity of our code.

In addition, due to some issues with the integration of the assembly on the ARTY development board, we were unable to complete this part of the project. We would have liked to get to the end of this project.

Although the majority of the codes necessary to carry out this project had previously been studied in class, we experienced some difficulties in linking them to each other. Thus, this project is a necessary investment, which will be beneficial to us to succeed in our partial future for this module.

However, this project was very rewarding for us in understanding the design of a digital circuit.

We were able to deepen our knowledge of VHDL, a hardware description language for the simulation and synthesis of digital systems.

This is the first project within the Efrei which seems really concrete to us and which allowed us to manipulate electronic components with a concrete result. In addition, building a microcontroller, a component at the base of modern computing, allows us to really understand how our everyday electronic devices work.