UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
ESCOLA POLITÉCNICA
LABORATÓRIO DE SISTEMAS DIGITAIS
2019.1

UNIDADE LÓGICA ARITMÉTICA (ULA)

ANA LÚCIA CANTO
VITOR LIMA
RENAN NERI

Índice

1. INTRODUÇÃO

A divisão do projeto de sistemas digitais em módulos é uma abordagem que facilita a sua execução pela possibilidade de divisão de uma tarefa grande em menores. Ter um módulo que já implementa diversas funções úteis fornece flexibilidade, uma vez que as tarefas ficam centralizadas, e agiliza a concepção do circuito, podendo até reduzir seu tamanho. É com interesse nesta flexibilidade que este projeto almeja a criação de uma ULA (Unidade Lógica e Aritmética) que implemente um conjunto de 8 operações sobre vetores de entrada de 4 bits.

As operações que a ULA realiza são: Soma, subtração, *OR*, *AND*, *XOR*, *NOT OR*, *NOT AND* e *NOT XOR*. A ULA tem como entrada um vetor de 3 bits para seleção da operação que será realizada, 1 vetor de 4 bits para entrada do operando A e 1 vetor de 4 bits para entrada do operando B. Como saída temos 1 vetor de 4 bits que representa o resultado da operação e um vetor de 4 bits que representam as *flags* da operação.

Referente a entrada de seleção, temos que: “000”refere-se a soma, “001”refere-se a subtração, “010”refere-se a operação OR, “011”refere-se a operação AND, “100”refere-se a operação XOR, “101”refere-se a operação NOT OR, “110”refere-se a operação NOT AND e “111”refere-se a NOT XOR.

A saída de 4 bits referente ao flags da ULA segue a ordem: *carry out,* zero, *overflow* e negativo.

A ULA será implementada numa FPGA utilizando o *Kit Xilinx Spartan3*. Para a implementação na FPGA, precisamos desenvolver como suporte um modulo que troca as entradas de seleção através de um contador, já que não temos entradas suficientes e gostariam de ver as respostas rapidamente, atuando parecido como um *FOR* que vai incrementando de 1 em 1. Além disso, precisamos de um divisor de *clock* para que a resposta se tornasse visível como pelo menos 1 segundo para ser mostrada.

2 - DESENVOLVIMENTO DO PROJETO

2.1 - Unidade Lógica Aritmética (ULA)

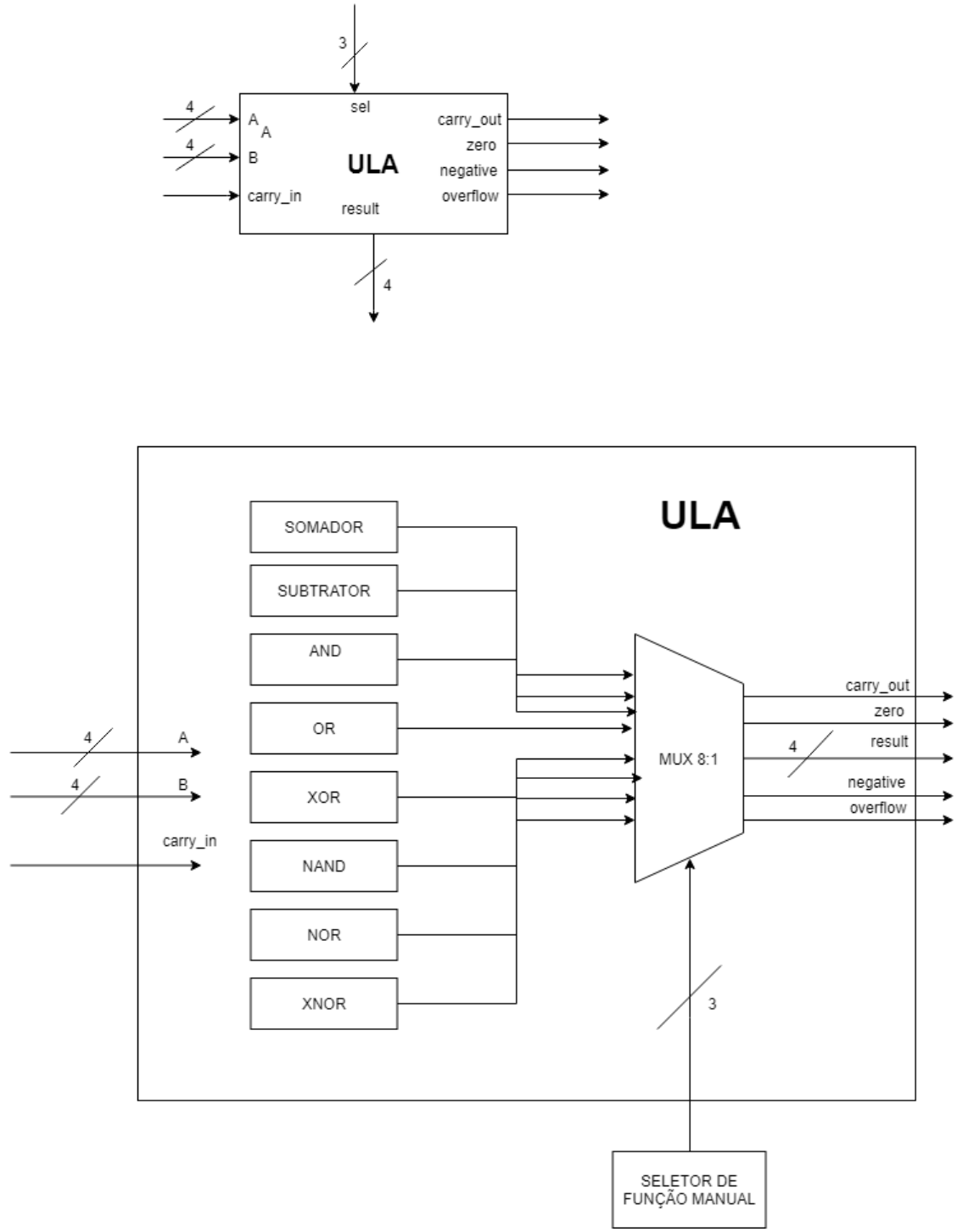


Figura 1- Unidade Lógica Aritmética

Para o desenvolvimento da unidade Lógica Aritmética, foram desenvolvidos, primeiramente, módulos de soma e subtração separados para as operações aritméticas. Serão demonstrados a seguir o desenvolvimento destes:

2.1.1 Operações Aritméticas

2.1.1.1 - Somadores

Para a implementação do somador, foi desenvolvido primeiramente um somador de 1 bit, com 2 entradas binárias e uma entrada para o *carry in,* 1 saída para o resultado, 1 para o flag *overflow* e outra saída para o *carry out.* Segue abaixo o código referente a essa implementação.

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

-- Definindo um somador para 1 bit

entity somador_1_bit is
   Port ( entrada1 : in  STD_LOGIC;
          entrada2 : in  STD_LOGIC;
          carry_in : in  STD_LOGIC;
          saida1 : out  STD_LOGIC;
          carry_out : out  STD_LOGIC);
end somador_1_bit;

architecture Behavioral of somador_1_bit is SIGNAL x1,x2,x3,x4,y1: STD_LOGIC;
BEGIN
            x1<=entrada1 AND entrada2;
            x2<=entrada1 AND carry_in;
            x3<=entrada2 AND carry_in;
            x4<= x1 OR x2;
            carry_out<=x3 OR x4;
            y1<= entrada1 XOR entrada2;
            saida1<= y1 XOR carry_in;
END Behavioral;
```

Código I - Somador 1 bit

Para o desenvolvimento do somador módulo 4, então, foi utilizado o componente do somador módulo 1, expandindo-o para se adaptar aos 4 bits que entrarão na ULA. Desse modo, as entradas são as mesmas, apenas com a modificação das entradas para operação, já que, agora as entradas do somador serão dois vetores de 4 bits. Segue abaixo a implementação:

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

--Implementação do somador 4 bits através do componente somador 1 bit:

entity somador_4_bits is
   Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
          b : in  STD_LOGIC_VECTOR (3 downto 0);
          carry_in : in  STD_LOGIC;
          carry_out : out  STD_LOGIC;
          sum : out  STD_LOGIC_VECTOR (3 downto 0);
                              overflow: out  STD_LOGIC);
end somador_4_bits;

architecture Behavioral of somador_4_bits is

-- Importando o componente do somador 1 bit:

component somador_1_bit
   Port ( entrada1 : in  STD_LOGIC;
          entrada2 : in  STD_LOGIC;
          carry_in : in  STD_LOGIC;
          saida1 : out  STD_LOGIC;
          carry_out : out  STD_LOGIC);
end component somador_1_bit;

signal c0, c1, c2, c3: std_logic;

-- Implementação através do somador 1 bit:

begin
            sum0 : somador_1_bit PORT MAP( a(0), b(0), carry_in, sum(0), c0 );
            sum1 : somador_1_bit PORT MAP( a(1), b(1), c0, sum(1), c1 );
            sum2 : somador_1_bit PORT MAP( a(2), b(2), c1, sum(2), c2 );
            sum3 : somador_1_bit PORT MAP( a(3), b(3), c2, sum(3), c3 );

            carry_out <= c3;

            overflow <= c3 XOR c2;

end Behavioral;
```

Código II - Somador 4 bits

2.1.1 – Subtrator

Uma maneira simples de subtrair números binários é através do complemento a 2. Ou seja, dado que queremos diminuir B de A, podemos complementar o subtraendo (B) e somar ao número (A) que obteremos o minuendo da operação. Portanto, foi utilizado como componente o somador, e, para complementar o subtraendo(B) apenas o invertemos e colocamos "1" no *carry in* do somador. Portanto, esse subtrator é composto por 2 entradas binárias , 1 saída para o resultado, 1 para o flag *overflow* e outra saída para o *carry out.* Segue a implementação:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- Implementação do subtrator através do complemento a 2 e o componente do somador 4
bits:

entity subtrator_4_bits is
   Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
          b : in  STD_LOGIC_VECTOR (3 downto 0);
          sub : out  STD_LOGIC_VECTOR (3 downto 0);
                                carry_out: out std_logic;
                                overflow: out  STD_LOGIC);
end subtrator_4_bits;

architecture Behavioral of subtrator_4_bits is
-- Importando componente do somador:

component somador_4_bits
   Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
          b : in  STD_LOGIC_VECTOR (3 downto 0);
          carry_in : in  STD_LOGIC;  -- Nesse caso o carry_in será 1 para ajudar no complemento
2.
          carry_out : out  STD_LOGIC;
          sum : out  STD_LOGIC_VECTOR (3 downto 0);
                                overflow: out  STD_LOGIC);
end component somador_4_bits;

-- Implementação da subtração através do somador, note que o subtraendo é invertido:
begin
b2 <= NOT b;
rsub: somador_4_bits PORT MAP( a, b2, '1' , carry_out, sub, overflow );

end Behavioral;
```

Código III - Subtrator 4 bits

2.1.2 – Operações Lógicas:

Como as operações logicas implementadas foram *AND, XOR, OR* e o inverso destas. A implementação se deu de maneira prática no próprio módulo da ULA. Descrevendo-as de maneira direta e invertendo-as. A lógica será mostrada mais adiante no "Código IV".

2.1.3 – Multiplexador e entradas de seleção

Para determinar qual operação a ULA irá realizar, é necessário que haja uma seleção. Como são 8 operações, essa seleção será dada por um vetor de 3 bits, o qual cada número representará uma operação. Para auxiliar na entrada dessa seleção portanto, foi desenvolvido um multiplexador com 8 entradas e 1 saída. A implementação estará mais adiante no "código IV".

2.1.4 – Saídas de flags

Os resultados das saídas dos flags também foram feitos de maneira bem direta. Para o flag *zero*, foi feito um *OR* negado de todos os bits que compõem o resultado. Para o flag *negative*, foi colocado o bit mais significativo, ou seja, se o bit mais significativo for 1, o resultado é negativo, caso contrário, positivo. O flag *overflow* foi definido no somador como o *XOR* do *carry out* do bit mais significativo com o *carry* out do bit anterior ao mais significativo. O *carry out*, dessa forma, será ativado quando a operação extrapolar o limite de 4 bits e for necessário mais um algoritmo para representar o resultado.

```
-- Projeto da ULA de 4 bits com as operações  aritméticas soma e subtração e as operações
lógicas XOR, AND, OR, XNOR, NAND e NOR.

entity ULA_4_bits is
   Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
        b : in  STD_LOGIC_VECTOR (3 downto 0);
                              carry_in : in  STD_LOGIC;
                              sel : in  STD_LOGIC_VECTOR (2 downto 0);
                              result : out STD_LOGIC_VECTOR (3 downto 0);
                              carry_out : out  STD_LOGIC;
                              zero : out  STD_LOGIC;
                              overflow : out  STD_LOGIC;
                              negative : out  STD_LOGIC);
end ULA_4_bits;
```

```
architecture Behavioral of ULA_4_bits is

 -- Importando o módulo somador de 4 bits:
component somador_4_bits
   Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
          b : in  STD_LOGIC_VECTOR (3 downto 0);
          carry_in : in  STD_LOGIC;
          carry_out : out  STD_LOGIC;
          sum : out  STD_LOGIC_VECTOR (3 downto 0);
         overflow: out  STD_LOGIC);
end component somador_4_bits;

--Importando o módulo subtrator de 4 bits.
component subtrator_4_bits
   Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
            b : in  STD_LOGIC_VECTOR (3 downto 0);
           carry_out: out std_logic;
            sub : out  STD_LOGIC_VECTOR (3 downto 0);
             overflow: out  STD_LOGIC);
end component subtrator_4_bits;

-- Definindo signals(fios) para passagens de valores:
signal saidaUla: STD_LOGIC_VECTOR (3 downto 0);
signal op_or: STD_LOGIC_VECTOR (3 downto 0);
signal op_and: STD_LOGIC_VECTOR (3 downto 0);
signal op_xor: STD_LOGIC_VECTOR (3 downto 0);
signal op_soma: STD_LOGIC_VECTOR (3 downto 0);
signal op_sub: STD_LOGIC_VECTOR (3 downto 0);
signal flag_zero: STD_LOGIC;
signal flag_overflow_soma: STD_LOGIC;
signal flag_overflow_sub: STD_LOGIC;
signal flag_negative: STD_LOGIC;
signal carry_soma: STD_LOGIC;
signal carry_sub: STD_LOGIC;

-- Definindo as operações lógicas e aritméticas importadas.
begin
           op_or <= a or b;  --OR
           op_and <= a and b; --AND
           op_xor <= a xor b; --XOR
           lab_soma: somador_4_bits port map ( a, b, carry_in, carry_soma, op_soma,
           flag_overflow_soma ); --SOMA
           lab_sub: subtrator_4_bits port map ( a, b, carry_sub, op_sub, flag_overflow_sub ); --
SUBTRAÇÃO

           overflow <= flag_overflow_soma when( sel = "000" ) else
                       flag_overflow_sub when( sel = "001" ) else
                                           '0';
```

```
-- Implementando um multiplexador para a seleção da operação realizada pela ULA.

        saidaUla <= op_soma when( sel = "000" ) else
                            op_sub when( sel = "001" ) else
                                        op_or when( sel = "010" )else
                                    op_and when( sel = "011" )else
                                    op_xor when( sel = "100" )else
                                    not(op_or) when( sel = "101" )else
                                    not(op_and) when( sel = "110" )else
                                    not(op_xor);

-- Definindo saídas de flags

        carry_out <= carry_soma when ( sel = "0000" ) else
                                        carry_sub when ( sel = "0001" ) else
                                        '0';

        result <= saidaUla;

        flag_zero <= not(a(0) or a(1) or a(2) or a(3));
        zero <= flag_zero;

        flag_negative <= a(3);
        negative <= flag_negative;

        end Behavioral;
```

Código IV - Unidade Lógica Aritmética

2.2 - Gerador de Números e tempo de representação dos resultados

A implementação do gerador de números se deu através da criação de um contador de 10 bits. Desses 2 bits, os 2 primeiros bits da saída foram para o tempo de representação dos resultados, ou seja, no tempo “00” aparecerá vetor de entrada A na placa, no tempo “01” aparecerá o vetor de entrada B, no tempo “10” o vetor de resultado e no tempo “11” o vetor de *flags*. E os vetores A e B foram gerados pelos 8 bits de saída restantes. O código referente poderá ser visto mais adiante no “Código VII”.

2.2.1 – Contador 10 bits

Este contador, então, foi implementado com as entradas de *clock* e *reset* e um vetor de saída da contagem. Segue adiante a implementação:

Figura 2 - Contador 10 bits

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

entity contador_10_bits is
   Port (
      clock,reset: IN STD_LOGIC;
      count: out STD_LOGIC_VECTOR( 9 DOWNTO 0 )
   );
end contador_10_bits;

architecture Behavioral of contador_10_bits is
CONSTANT nb: INTEGER := 9;

BEGIN

   upcount: PROCESS( clock )
      VARIABLE contagem: UNSIGNED (nb DOWNTO 0):="0000000000";
   BEGIN
      IF( clock'event AND clock= '1' ) THEN
         IF reset = '1' THEN
            contagem := "0000000000";
         ELSE
            contagem := contagem + 1;
         END IF; -- IF reset = '1
         count <= std_logic_vector(contagem);
      END IF; -- IF( clock'event AND clock = '1' )

   END PROCESS upcount;

END Behavioral;
```

Código V - Contador 10 bits

2.3 Gerador de Frequências

Para que o *clock* se desse no tempo exato de 1 segundo, foi necessária a construção de um divisor de *clock* como mostrado abaixo:

Figura 3 - Divisor de clock

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

entity clock_divider is
   Port (
      clock_in,reset: IN STD_LOGIC;
                     clock_out: OUT STD_LOGIC
   );
end clock_divider;

architecture Behavioral of clock_divider is

BEGIN
   PROCESS(clock_in,reset)
      VARIABLE contagem: INTEGER:= 0;
                     VARIABLE temp : STD_LOGIC:= '0';
   BEGIN
         IF reset = '1' THEN
           contagem:=0;
             temp := '0';
         ELSIF RISING_EDGE(clock_in) THEN
            contagem := contagem + 1;
         IF contagem=50000000 then
           temp:= NOT temp;
           contagem := 0;
         END IF; -- IF reset = '1
      END IF; -- IF( clock'event AND clock = '1' )
                     clock_out <= temp;
   END PROCESS;
END Behavioral;
```

Código VI – Clock DIvider

2.4 Interface

Para a integração de todas as partes descritas anteriormente, então, foi desenvolvida uma interface com a ULA, o gerador de frequências e o gerador de números. Esta possuirá a entrada de seleção da operação, que será manual, na própria FPGA, a entrada de *clock*, *rese*t e a saída de um vetor de 4 bits, ora A, ora, B, ora o resultado e por fim o vetor de *flags*.

Segue a implementação dessa situação:

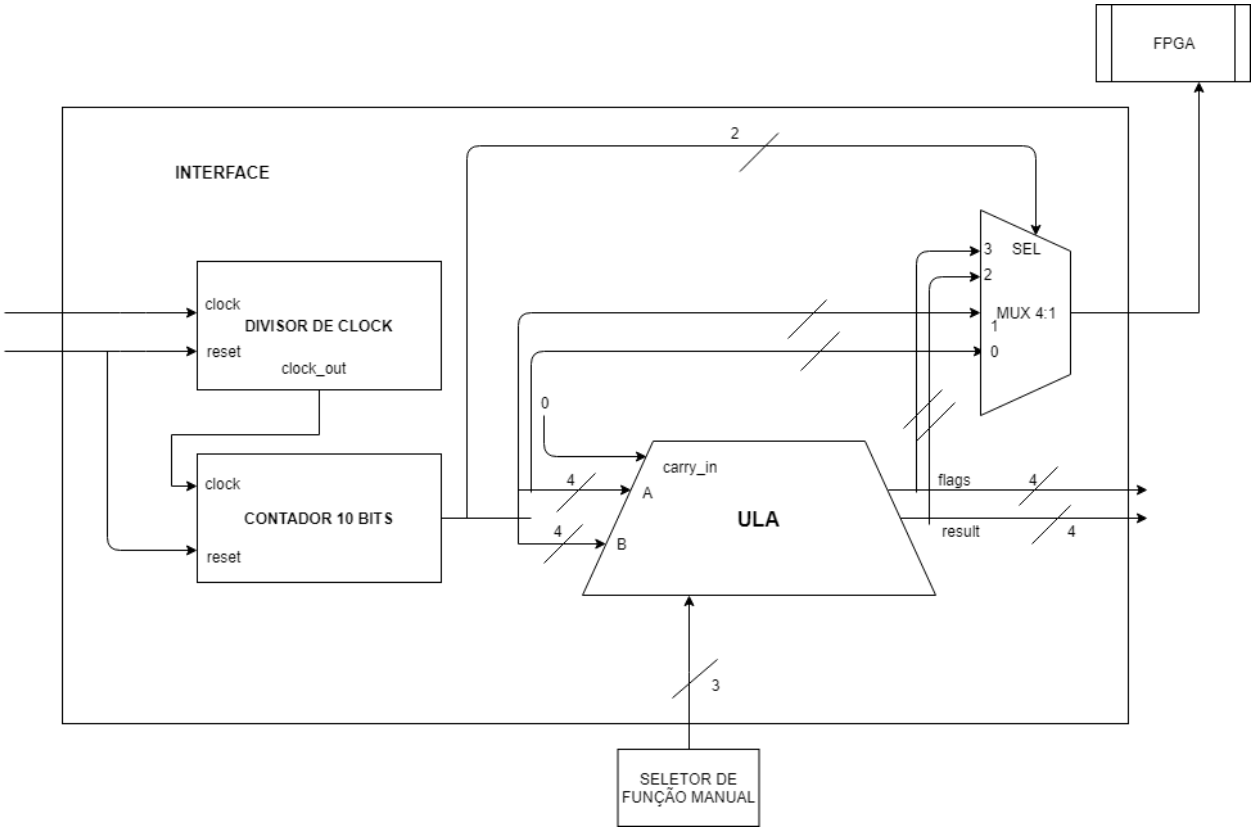


Figura 4 - Interface

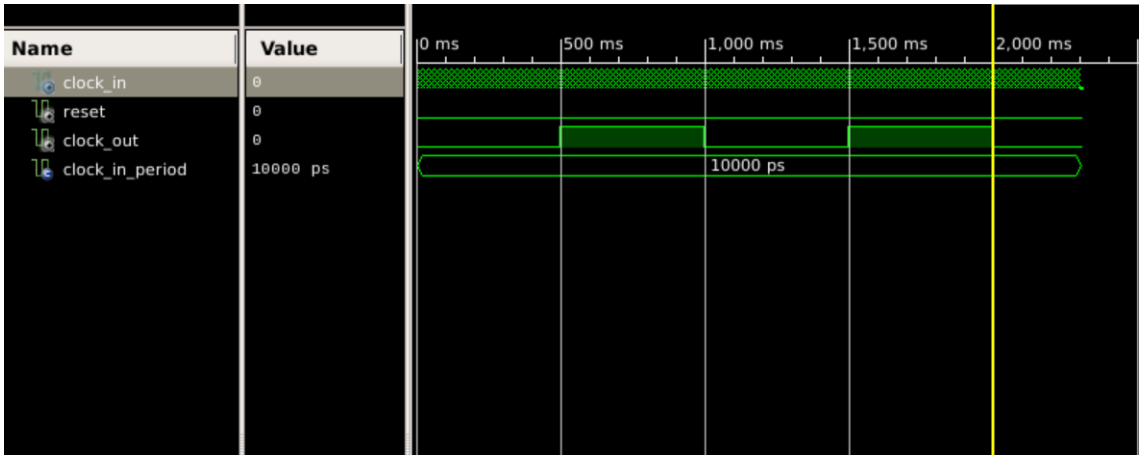
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity interface is
        Port ( sel : IN STD_LOGIC_VECTOR (2 downto 0);
                        clock : IN STD_LOGIC;
                        reset: IN STD_LOGIC;
                        -- FLAGS
                        saida: out STD_LOGIC_VECTOR (3 DOWNTO 0)
                       );
end interface;

architecture Behavioral of interface is

component ULA_4_bits is
   Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
       b : in  STD_LOGIC_VECTOR (3 downto 0);
                        carry_in : in  STD_LOGIC;
                        sel : in  STD_LOGIC_VECTOR (2 downto 0);
                        result : out STD_LOGIC_VECTOR (3 downto 0);
                        carry_out : out  STD_LOGIC;
                        zero : out  STD_LOGIC;
                        overflow : out  STD_LOGIC;
                        negative : out  STD_LOGIC);
end component ULA_4_bits;

component contador_10_bits is
   Port (
      clock,reset: IN STD_LOGIC;
      count: out STD_LOGIC_VECTOR( 9 DOWNTO 0 )
   );
end component contador_10_bits;

component clock_divider is
   Port (
      clock_in,reset: IN STD_LOGIC;
                  clock_out: OUT STD_LOGIC
   );
end component clock_divider;

SIGNAL result : STD_LOGIC_VECTOR (3 downto 0);
SIGNAL clock_b : STD_LOGIC;
SIGNAL a : STD_LOGIC_VECTOR( 3 DOWNTO 0 );
SIGNAL b : STD_LOGIC_VECTOR( 3 DOWNTO 0 );
SIGNAL timer : STD_LOGIC_VECTOR( 1 DOWNTO 0 );
SIGNAL flags : STD_LOGIC_VECTOR (3 DOWNTO 0);
SIGNAL carry_out : STD_LOGIC;
SIGNAL zero : STD_LOGIC;
SIGNAL overflow : STD_LOGIC;
SIGNAL negative : STD_LOGIC;
SIGNAL count: STD_LOGIC_VECTOR( 9 DOWNTO 0 );
```

```
-- Dividindo as saídas do contador de 10 bits para os 2 primeiros bits serem o tempo e os outros 8 serem os vetores A e B que entrarão na ULA.

timer(0) <= count(0);
timer(1) <= count(1);
b(0) <= count(2);
b(1) <= count(3);
b(2) <= count(4);
b(3) <= count(5);
a(0) <= count(6);
a(1) <= count(7);
a(2) <= count(8);
a(3) <= count(9);

flags(0) <= carry_out;
flags(1) <= overflow;
flags(2) <= zero;
flags(3) <= negative;


ULA : ULA_4_bits PORT MAP (a,b,'0',sel,result,carry_out,zero,overflow,negative);
DIVISOR: clock_divider PORT MAP (clock,reset,clock_b);
GERADOR: contador_10_bits PORT MAP (clock_b,reset,count);

-- Colocando para aparecer nos leds, primeiro o vetor A, depois o vetor B, posteriormente o resultado seguido dos flags.

saida <= a when( timer = "00" ) else
                b when( timer = "01" ) else
                result when( timer = "10" )else
                        flags when( timer = "11" )else
                "0000";

end Behavioral;
```

Código VII - Interface

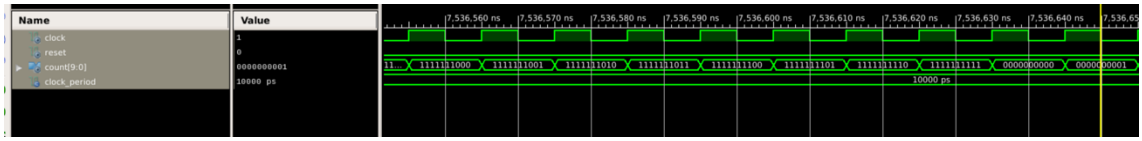
3 - Testes e simulações do projeto

3.1 Divisor de clock



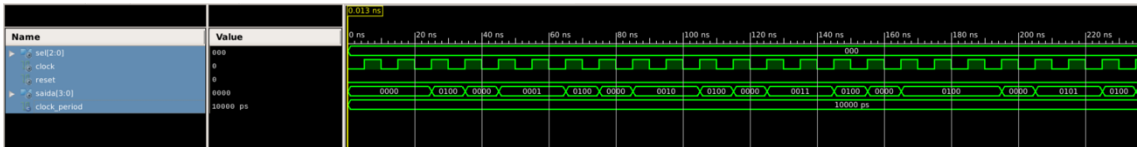
Simulação do divisor de *clock*, em 2.000 ms (2 segundos) o modulo completa dois pulsos de *clock* na saída.

3.2 Gerador de números



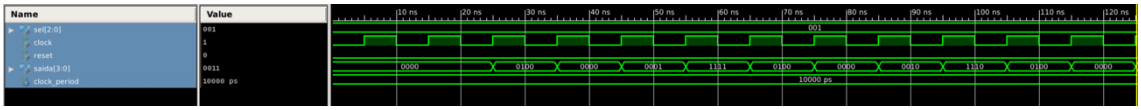
Simulação do gerador de números, que consiste em um contador de 10bits, a simulação mostrar o modulo indo até o final da contagem e retornando a zero.

3.3 Interface – Somador



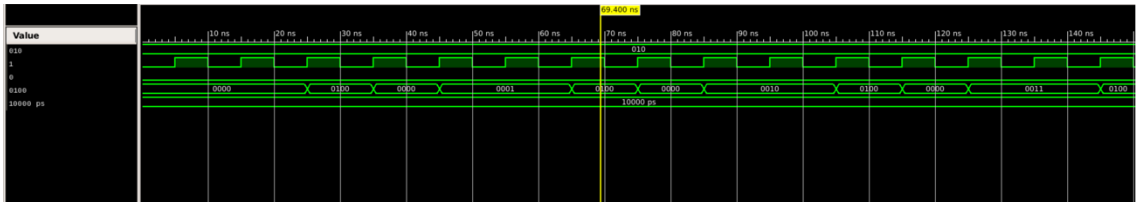
Simulação da interface com entrada de seleção “000”que é a configuração da soma. Depois do quarto *clock,* vimos a soma de “0000”com “0001”, resultando em “0001”com flags “0100”, sendo respectivamente a entrada A, a entrada B, a saída e as flags.

3.4 Interface Subtrator



Simulação da interface com entrada de seleção “001”que é a configuração da subtração. Depois do quarto *clock*, vimos a subtração de “0000”com “0001”, resultando em “1111”com *flags* “0100”, sendo respectivamente a entrada A, a entrada B, a saída e as *flags*.

3.5 Interface OR



Simulação da interface com entrada de seleção “010” que é a configuração da operação OR. Depois do quarto *clock*, vimos o OR de “0000” com “0001”, resultando em “0001”com *flags* “0100”, sendo respectivamente a entrada A, a entrada B, a saída e as *flags*.

4 – Conclusão

Diante do exposto nesse trabalho, podemos observar que a elaboração, foi feita de maneira progressiva, ou seja, o projeto foi dividido em módulos, de forma que começamos construindo módulos menores, já conhecidos pela equipe e posteriormente, a interligação destes módulos possibilitou a construção do projeto.

Ao desenvolver os módulos, foram feitos testes para cada um destes, afim de garantir a integralidade destes. E, por fim, o teste foi realizado na própria FPGA.

Portanto, a elaboração da Unidade Lógica Aritmética se deu de maneira satisfatória. A maior dificuldade encontrada foi o entendimento da linguagem e seu funcionamento, algo até então, não visto pelos autores do projeto.

Bibliografia

Figuras : Feitas no software Draw.io https://www.draw.io/ acesso em: 18/05/2019.

Transparências: https://www.gta.ufrj.br/ensino/EEL480/Introducao-VHDL.pdf