UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Departamento de Engenharia Eletrônica e da Computação
Escola Politécnica
EEL 480 – Sistemas Digitais

UNIDADE LÓGICA ARITMÉTICA EM VHDL

Alunos: Bruno Dantas de Paiva, João Ricardo Magalhães e Guilherme Bergman

Professor: Luis Henrique Maciel Kosmalski Costa

Rio de Janeiro,
2019

I. Introdução:

A unidade lógica aritmética (ULA) é um dispositivo capaz de realizar operações lógicas e aritméticas após ser inseridos números de 4 bits (1 ou 2) e escolhida a sua operação.

Neste trabalho, foi utilizado uma FPGA (que consiste em um arranjo de células lógicas ou blocos lógicos configuráveis contidos em um único circuito integrado) e a linguagem de descrição de hardware (VHDL) para implementar tais funções.

Dentre as funções implementadas, estão elas: adição, subtração, complementa, incrementa um e xor, or, not, and (ambos bit a bit).

II. Implementação:

Para implementar a ULA, foi necessário a implementação separada de cada função a ser implementada, de modo a facilitar a visualização do código e também tornar mais simples o momento de encontrar erros de código ou falhas de projetos.

Além disso, foi necessário (como viés introdutório) criar um módulo somador de um bit, a fim de introduzir a linguagem aos membros do grupo, que será usado, futuramente, na confecção do módulo somador de 4 bits. Segue o código do somador de um bit:

```
entity somapain is
   Port ( cin : in  STD_LOGIC;
          x : in  STD_LOGIC;
          y : in  STD_LOGIC;
```

```
        cout : out  STD_LOGIC;
        saida : out  STD_LOGIC);
end somapain;

architecture Behavioral of somapain is
SIGNAL k1, k2, k3, k4, kfinal : STD_LOGIC;

begin
        k1 <= x and y;
        k2 <= x and cin;
        k3 <= y and cin;
        k4 <= k1 or k2;
        cout <= k3 xor k4;
        kfinal <= x XOR y;
        saida <= kfinal xor cin;
end Behavioral;
```

A- Somador de 4 bits:

O projeto consiste na utilização de 4 somadores de um bit cascateados entre si por meio de seu carry. Para isso, é necessário importar o módulo somador de 1 bit (previamente implementado) em um novo arquivo, a fim de utilizar todas suas entradas e saídas como pode ser observado no código abaixo:

```
entity somapain4 is
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
```

```
        y : in  STD_LOGIC_VECTOR (3 downto 0);
        cin : in  STD_LOGIC;
        cout : out  STD_LOGIC;
        saida : out  STD_LOGIC_VECTOR (3 downto 0);
        Flag_Zero : out STD_LOGIC;
        Flag_Sinal : out STD_LOGIC;
        Flag_Overflow : out STD_LOGIC;
                Flag_Cout : out STD_LOGIC);
end somapain4;

architecture Behavioral of somapain4 is

COMPONENT somapain
   PORT(
      cin : IN  std_logic;
      x : IN  std_logic;
      y : IN  std_logic;
      cout : OUT  std_logic;
      saida : OUT  std_logic
     );
end COMPONENT somapain;

signal prop: std_logic_vector(2 downto 0);
signal valor : std_logic_vector(3 downto 0);
```

```
signal c3 : std_logic;

signal c2 : std_logic;

begin


a0: somapain port map (cin, x(0), y(0), prop(0), valor(0));

a1: somapain port map (prop(0), x(1), y(1), prop(1), valor(1));

a2: somapain port map (prop(1), x(2), y(2), c2, valor(2));

a3: somapain port map (c2, x(3), y(3), c3, valor(3));

Flag_Zero <= not(valor(0) or valor(1) or valor(2) or valor(3));

Flag_Sinal <= valor(3);

Flag_Overflow <= c3 xor c2;

Flag_Cout <= c3;

cout <= c3;

saida <= valor;

end Behavioral;
```

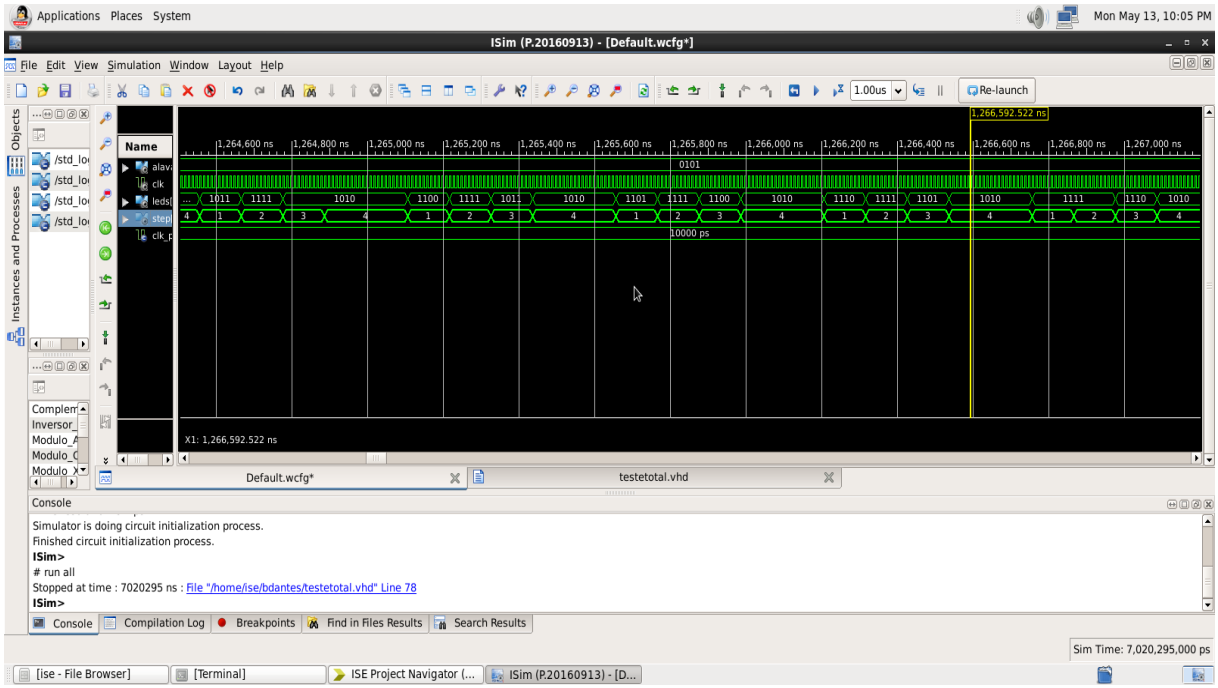

Figura 1: Simulação do Somador de 4 bits.

B- Incrementa um:

O projeto consiste em adicionar somar um bit ao final de um vetor de 4 bits inserido pelo operador da ULA.

Note que tal módulo necessitará da utilização do somador supracitado, devido o fato de que todos os vetores de entrada são de 4 bits. Além disso, tal módulo também será necessário em projetos posteriores. Segue o código a seguir:

```
entity incrementa1 is
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           saida : out  STD_LOGIC_VECTOR (3 downto 0);
           Flag_Zero : out STD_LOGIC;
           Flag_Sinal : out STD_LOGIC);
end incrementa1;

architecture Behavioral of incrementa1 is

COMPONENT somapain4
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           y : in  STD_LOGIC_VECTOR (3 downto 0);
           cin : in  STD_LOGIC;
           cout : out  STD_LOGIC;
           saida : out  STD_LOGIC_VECTOR (3 downto 0);
           Flag_Zero : out STD_LOGIC;
```

```
            Flag_Sinal : out STD_LOGIC;
            Flag_Overflow : out STD_LOGIC);
    end COMPONENT somapain4;

    SIGNAL resultado: STD_LOGIC_VECTOR(3 downto 0);
    SIGNAL um: STD_LOGIC_VECTOR(3 downto 0);
    SIGNAL complementado: STD_LOGIC_VECTOR(3 downto 0);
    SIGNAL cout: std_logic;
    SIGNAL Flag_Zero_somador : std_logic;
    SIGNAL Flag_Sinal_somador : std_logic;
    SIGNAL Flag_Overflow_somador : std_logic;

    begin

    um(0) <= '1';
    um(1) <= '0';
    um(2) <= '0';
    um(3) <= '0';
    label2: somapain4 port map (x, um, '0', cout, complementado);
    saida <= complementado;
    Flag_Zero <= not (complementado(0) or complementado(1) or
complementado(2) or complementado(3));
    Flag_Sinal <= complementado(3);
    end Behavioral;
```

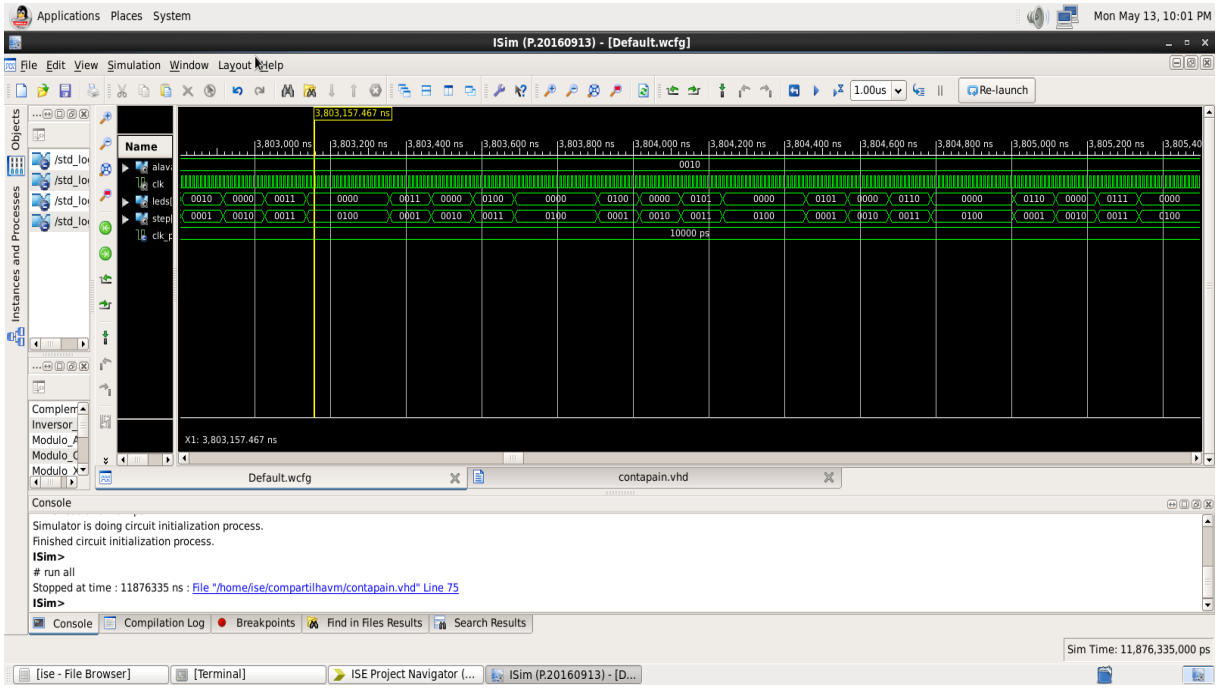

Figura 2: Simulação do incrementa um.

C- Complementa:

O projeto consiste em realizar a operação de complemento a 2 dado um vetor de 4 bits.

Para tal, é necessário fazer a importação dos códigos desenvolvidos acima (somador e incrementa um), deste modo será mais fácil desenvolver o desenvolvimento. Segue o código a seguir:

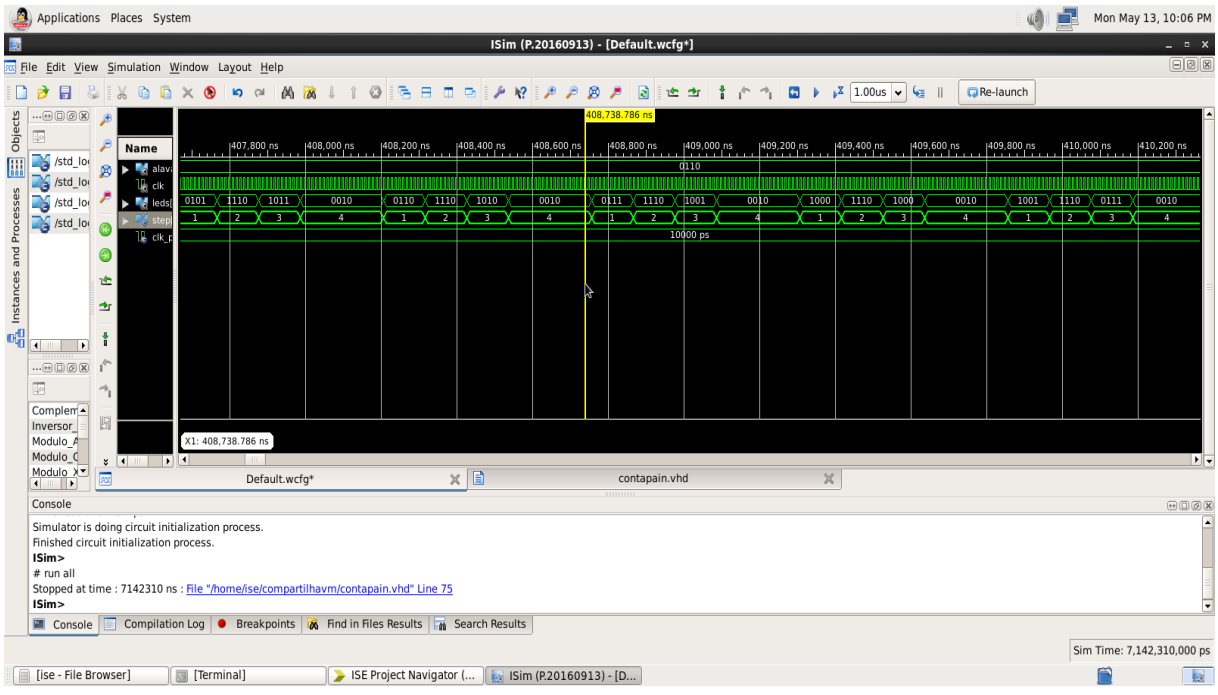

Figura 3: Simulação do Complementa.

D- Subtrator:

O projeto consiste em desenvolver o módulo subtrator que, dado 2 vetores de 4 bits A e B (respectivamente), retorna a A-B, com a indicação de carry / borrow e as respectivas flags.

Para o desenvolvimento de tal, foi necessário a utilização dos módulos previamente explicitados, como pode ser visto no código abaixo:

```
entity subtratain4 is
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           y : in  STD_LOGIC_VECTOR (3 downto 0);
           bin : in  STD_LOGIC;
           bout : out  STD_LOGIC;
          saida : out  STD_LOGIC_VECTOR (3 downto 0);
          Flag_Zero : out STD_LOGIC;
          Flag_Overflow : out STD_LOGIC;
          Flag_Sinal : out STD_LOGIC;
          Flag_Borrow : out STD_LOGIC
                        );
end subtratain4;

architecture Behavioral of subtratain4 is
```

```
COMPONENT somapain4
   PORT(
        cin : IN  std_logic;
        x : in  STD_LOGIC_VECTOR (3 downto 0);
        y : in  STD_LOGIC_VECTOR (3 downto 0);
        cout : out  std_logic;
        saida : out  STD_LOGIC_VECTOR (3 downto 0);
      Flag_Zero : out STD_LOGIC;
      Flag_Overflow : out STD_LOGIC;
      Flag_Sinal : out STD_LOGIC
        );

end COMPONENT somapain4;

COMPONENT Complementa
   PORT(
       x : in  STD_LOGIC_VECTOR (3 downto 0);
     saida : out  STD_LOGIC_VECTOR (3 downto 0);
     Flag_Zero : out STD_LOGIC;
     Flag_Sinal : out STD_LOGIC
       );

end COMPONENT Complementa;

SIGNAL ynvertido: STD_LOGIC_VECTOR (3 downto 0);
```

```
SIGNAL resultado: std_logic_vector (3 downto 0);
SIGNAL Flag_Zero_somador: std_logic;
SIGNAL Flag_Sinal_somador: std_logic;
SIGNAL Flag_Overflow_somador: std_logic;
signal armazena_borrow : std_logic;
signal borrow : std_logic;

begin

    a0:Complementa port map(y, ynvertido);
    a1:somapain4 port map (bin, x, ynvertido, borrow, resultado, Flag_Zero_somador, Flag_Overflow_somador,  Flag_Sinal_somador);
    saida <= resultado;
    Flag_Zero <= not(resultado(0) or resultado(1) or resultado(2) or resultado(3));
    Flag_Overflow <= Flag_Overflow_somador;
    armazena_borrow <= not borrow;
    bout<= armazena_borrow;
    Flag_Sinal <= resultado(3);
    Flag_Borrow <= armazena_borrow;
end Behavioral;
```

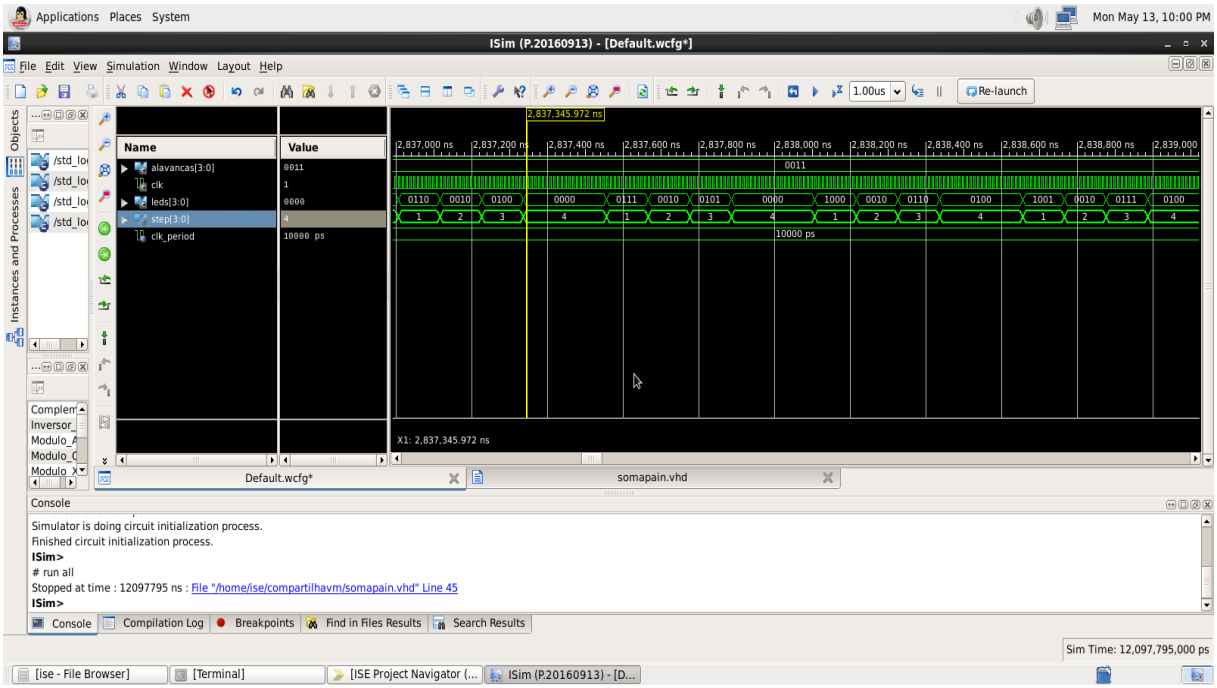

Figura 4: Simulação do subtrator.

E- And:

O projeto consiste em desenvolver um módulo and de 4 bits que executa a operação lógica and, bit a bit entre os 2 vetores de entrada. Segue o código abaixo:

```
entity Modulo_And is
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           y : in  STD_LOGIC_VECTOR (3 downto 0);
           saida : out  STD_LOGIC_VECTOR (3 downto 0);
          Flag_Zero : out STD_LOGIC;
          Flag_Sinal : out STD_LOGIC);
end Modulo_And;
```

```
architecture Behavioral of Modulo_And is

signal valor : std_logic_vector (3 downto 0);

begin
saida <= valor;
    Gen_1: For I IN 3 downto 0 generate
                    valor(I) <= x(I) and y(I);
    end generate;

    Flag_Zero <= not(valor(0) or valor(1) or valor(2) or valor(3));
    Flag_Sinal <= valor(3);
end Behavioral;
```

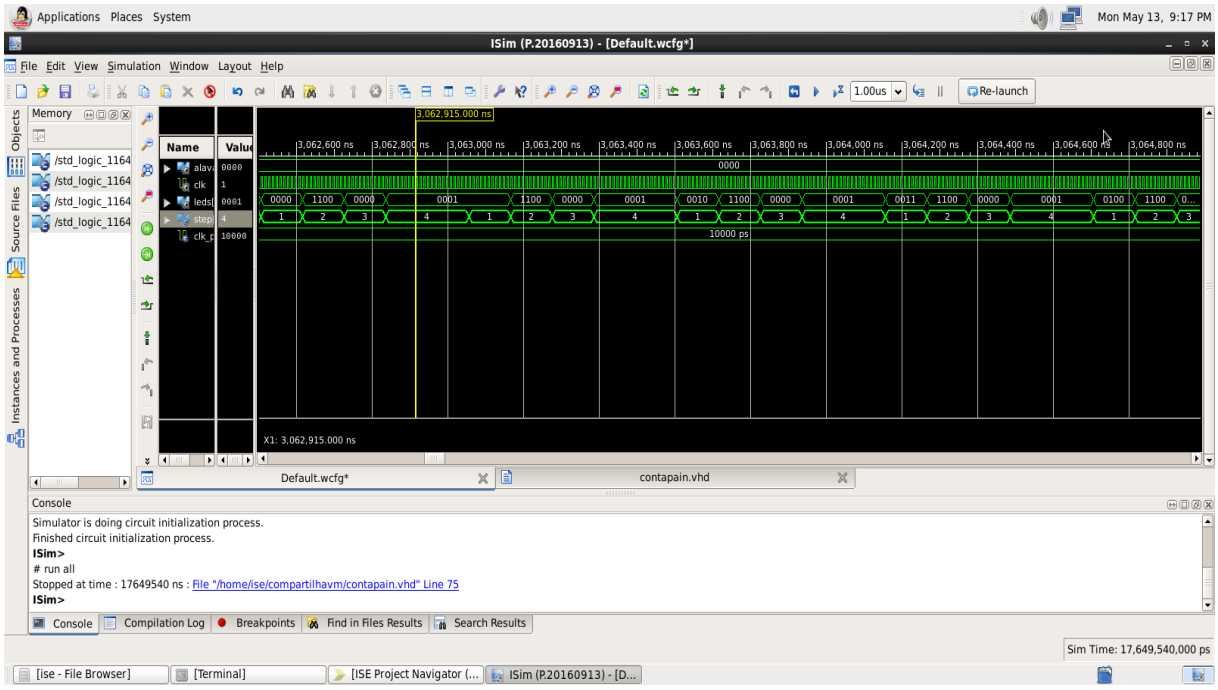

Figura 5: Simulação do and bit a bit.

F- Or:

O projeto consiste em desenvolver um módulo or de 4 bits que execute a operação lógica or, bit a bit entre 2 vetores de entrada. Segue o código abaixo:

```
entity Modulo_Or is
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           y : in  STD_LOGIC_VECTOR (3 downto 0);
           saida : out  STD_LOGIC_VECTOR (3 downto 0);
          Flag_Zero : out STD_LOGIC;
          Flag_Sinal : out STD_LOGIC);
end Modulo_Or;

architecture Behavioral of Modulo_Or is

signal valor : std_logic_vector(3 downto 0);

begin
saida <= valor;
      Gen_1: For I IN 3 downto 0 generate
                        valor(I) <= x(I) or y(I);
      end generate;

      Flag_Zero <= not(valor(0) or valor(1) or valor(2) or valor(3));
      Flag_Sinal <= valor(3);
```

```
end Behavioral;
```

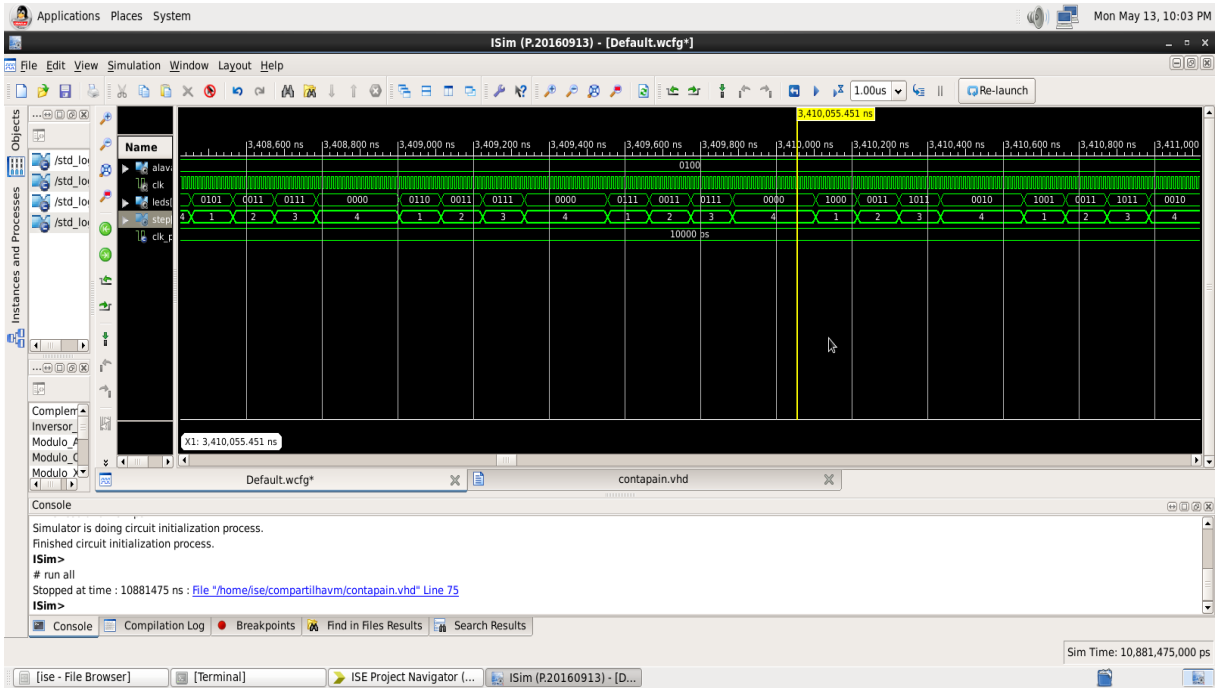

Figura 6: Simulação do Or bit a bit.

G- Xor:

O projeto consiste em desenvolver um módulo xor de 4 bits que execute a operação lógica xor, bit a bit entre 2 vetores. Segue o código abaixo:

```
entity Modulo_Xor is
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           y : in  STD_LOGIC_VECTOR (3 downto 0);
           saida : out  STD_LOGIC_VECTOR (3 downto 0);
          Flag_Zero : out STD_LOGIC;
```

```
        Flag_Sinal : out STD_LOGIC);
end Modulo_Xor;

architecture Behavioral of Modulo_Xor is

signal valor : std_logic_vector (3 downto 0);

begin
saida <= valor;
    Gen_1: For I IN 3 downto 0 generate
                    valor(I) <= x(I) xor y(I);
    end generate;

    Flag_Zero <= not(valor(0) or valor(1) or valor(2) or valor(3));
    Flag_Sinal <= valor(3);
end Behavioral;
```

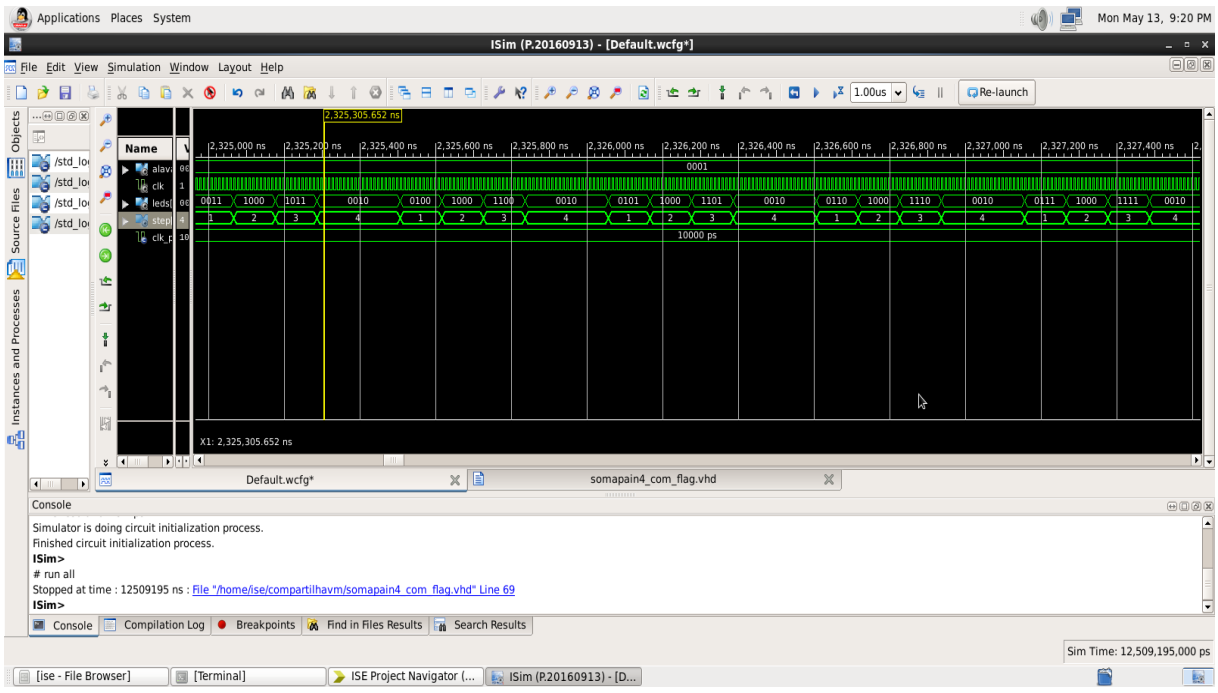

Figura 7: Simulação do xor bit a bit.

H- Inversor:

O projeto consiste em desenvolver um módulo not de 4 bits que execute a operação lógica not, bit a bit de um vetor. Segue o código abaixo:

```
entity Inversor is
   Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
          saida : out  STD_LOGIC_VECTOR (3 downto 0);
       Flag_Zero : out STD_LOGIC;
       Flag_Sinal : out STD_LOGIC);
end Inversor;

architecture Behavioral of Inversor is
```

```
SIGNAL valor : STD_LOGIC_VECTOR(3 downto 0);
begin
    Gen_1: For I IN 3 downto 0 generate
                    valor(I) <= not x(I);
    end generate;
    Flag_Zero <= not (valor(0) or valor(1) or valor(2) or valor(3));
    Flag_Sinal <= valor(3);
    saida <= valor;
end Behavioral;
```

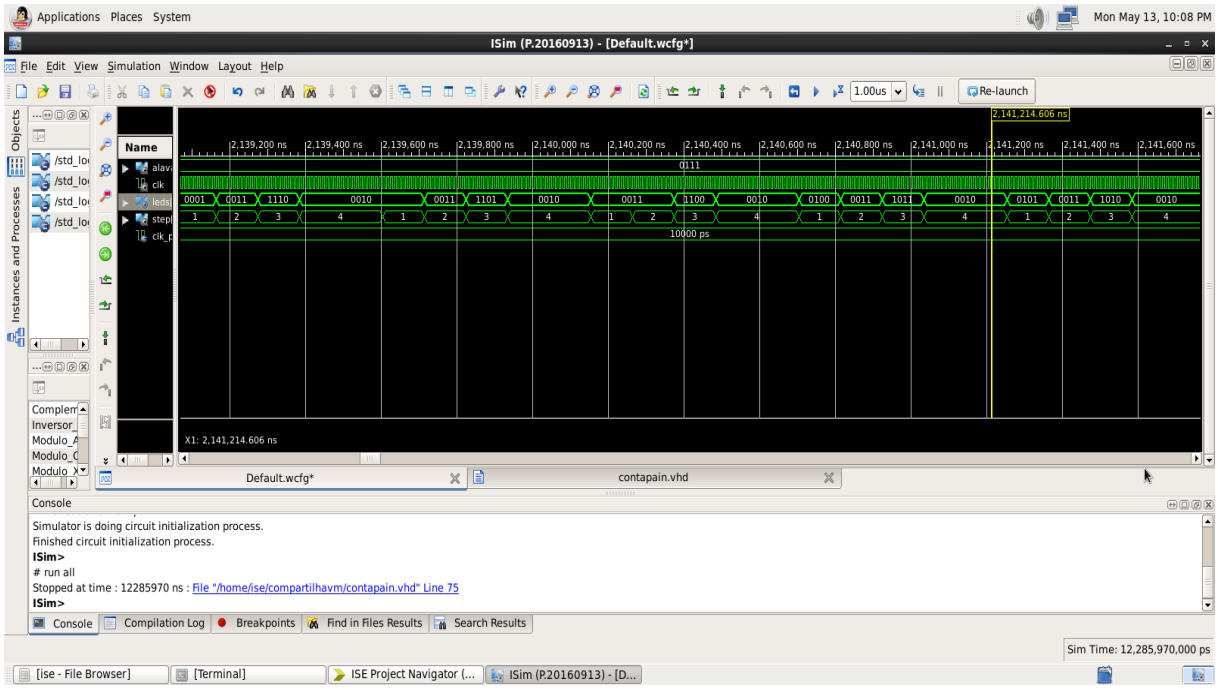

Figura 8: Simulação do inversor.

I- Contador:

O projeto consiste em desenvolver um contador de 28 bits para ser usado como divisor de frequência de 50 Mhz para 4 segundos, por decisão do grupo também foi escolhido fazer a interação do usuário

com a escolha da função a ser utilizada neste módulo também. Segue o código abaixo:

```
entity contapain is
    Port ( alavanca1 : in  STD_LOGIC;
           alavanca2 : in  STD_LOGIC;
           alavanca3 : in  STD_LOGIC;
           alavanca4 : in  STD_LOGIC;
           Led1 : out  STD_LOGIC;
           Led2 : out  STD_LOGIC;
           Led3 : out  STD_LOGIC;
           Led4 : out  STD_LOGIC;
                    op1 : out  STD_LOGIC;
           op2 : out  STD_LOGIC;
           op3 : out  STD_LOGIC;
           op4 : out  STD_LOGIC;
                    clk: in std_logic
                    );
end contapain;

architecture Behavioral of contapain is
component ULA
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Operacao : in  STD_LOGIC_VECTOR (3 downto 0);
           Z : out  STD_LOGIC_VECTOR (3 downto 0);
```

```
        Flag_Zero : out STD_LOGIC;
        Flag_Sinal : out STD_LOGIC;
        Flag_Overflow : out STD_LOGIC;
        Flag_Cout : out STD_LOGIC
        );
end component ULA;

signal inutil : STD_LOGIC_VECTOR (7 downto 0):= "00000000";
signal vetor1 : STD_LOGIC_VECTOR (3 downto 0):= "0000";
signal vetor2 : STD_LOGIC_VECTOR (3 downto 0):= "0000";
signal vetorgrande : UNSIGNED(7 downto 0):= "00000000";
signal result : STD_LOGIC_VECTOR (3 downto 0):= "0000";
signal fzero : STD_LOGIC := '0';
signal fsinal : STD_LOGIC:= '0';
signal fover : STD_LOGIC:= '0';
signal fcout : STD_LOGIC:= '0';
signal oparcione: STD_LOGIC_VECTOR (3 downto 0):= "0000";
signal carrie : STD_LOGIC:= '0';
begin
vetor1(0)<= vetorgrande(0);
vetor1(1)<= vetorgrande(1);
vetor1(2)<= vetorgrande(2);
vetor1(3)<= vetorgrande(3);
vetor2(0)<= vetorgrande(4);
vetor2(1)<= vetorgrande(5);
```

```
vetor2(2)<= vetorgrande(6);
vetor2(3)<= vetorgrande(7);
oparcione(0) <= alavanca1;
oparcione(1) <= alavanca2;
oparcione(2) <= alavanca3;
oparcione(3) <= alavanca4;
ulala: ULA port map (vetor1, vetor2, oparcione, result, fzero, fsinal, fover, fcout);
    process(clk)
        variable parte : INTEGER RANGE 15 DOWNTO 0:=0;
        variable redutor : INTEGER RANGE 200_000_000 DOWNTO 0:=0;
        begin
            if (clk'event and clk = '1') then
            if (redutor >= 10) then
                redutor := 0;
                if (parte = 0) then
                    Led1 <= vetor1(0);
                    Led2 <= vetor1(1);
                    Led3 <= vetor1(2);
                    Led4 <= vetor1(3);
                    op1 <= '1';
                    op2 <= '0';
                    op3 <= '0';
                    op4 <= '0';
```

```
elsif (parte = 1) then
        Led1 <= vetor2(0);
        Led2 <= vetor2(1);
        Led3 <= vetor2(2);
        Led4 <= vetor2(3);
        op1 <= '0';
        op2 <= '1';
        op3 <= '0';
        op4 <= '0';
elsif (parte = 2) then
        Led1 <= result(0);
        Led2 <= result(1);
        Led3 <= result(2);
        Led4 <= result(3);
        op1 <= '1';
        op2 <= '1';
        op3 <= '0';
        op4 <= '0';
elsif (parte = 3) then
        Led1 <= fzero;
        Led2 <= fsinal;
        Led3 <= fover;
        Led4 <= fcout;
        op1 <= '0';
        op2 <= '0';
```

```
                op3 <= '1';
                op4 <= '0';
            else
                vetorgrande<= vetorgrande + 1;
            end if;
            if (parte = 4) then
                parte := 0;
            else
                parte := parte + 1;
            end if;
        else
            redutor := redutor + 1;
        end if;
    end if;
  end process;
end Behavioral;
```

J- ULA:

Esta é a parte onde se encontra o projeto completo, com toda a interface do usuário pronta, o contador, subtrator e as operações lógicas unidas. Segue o código abaixo:

```
entity ULA is
  Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
      B : in  STD_LOGIC_VECTOR (3 downto 0);
      Operacao : in  STD_LOGIC_VECTOR (3 downto 0);
```

```
        Z : out  STD_LOGIC_VECTOR (3 downto 0);
      Flag_Zero : out STD_LOGIC;
      Flag_Sinal : out STD_LOGIC;
      Flag_Overflow : out STD_LOGIC;
      Flag_Cout : out STD_LOGIC
      );
end ULA;

architecture Behavioral of ULA is

COMPONENT somapain4
   Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
          y : in  STD_LOGIC_VECTOR (3 downto 0);
          cin : in  STD_LOGIC;
          cout : out  STD_LOGIC;
          saida : out  STD_LOGIC_VECTOR (3 downto 0);
      Flag_Zero : out STD_LOGIC;
      Flag_Sinal : out STD_LOGIC;
      Flag_Overflow : out STD_LOGIC;
            Flag_Cout : out STD_LOGIC
      );
end COMPONENT somapain4;

COMPONENT Complementa is
   Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
```

```
        saida : out  STD_LOGIC_VECTOR (3 downto 0);
      Flag_Zero : out STD_LOGIC;
      Flag_Sinal : out STD_LOGIC
    );
end COMPONENT Complementa;

COMPONENT Inversor is
   Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
        saida : out  STD_LOGIC_VECTOR (3 downto 0);
      Flag_Zero : out STD_LOGIC;
      Flag_Sinal : out STD_LOGIC);
end COMPONENT Inversor;

COMPONENT subtratain4 is
   Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
        y : in  STD_LOGIC_VECTOR (3 downto 0);
        bin : in  STD_LOGIC;
        bout : out  STD_LOGIC;
                    saida : out  STD_LOGIC_VECTOR (3 downto 0);
                    Flag_Zero : out STD_LOGIC;
                    Flag_Sinal : out STD_LOGIC;
                    Flag_Overflow : out STD_LOGIC;
                    Flag_Borrow : out STD_LOGIC);
end COMPONENT subtratain4;
```

```
COMPONENT Modulo_Xor is
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           y : in  STD_LOGIC_VECTOR (3 downto 0);
           saida : out  STD_LOGIC_VECTOR (3 downto 0);
          Flag_Zero : out STD_LOGIC;
          Flag_Sinal : out STD_LOGIC);
end COMPONENT Modulo_Xor;

COMPONENT Modulo_Or is
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           y : in  STD_LOGIC_VECTOR (3 downto 0);
           saida : out  STD_LOGIC_VECTOR (3 downto 0);
          Flag_Zero : out STD_LOGIC;
          Flag_Sinal : out STD_LOGIC);
end COMPONENT Modulo_Or;

COMPONENT Modulo_And is
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           y : in  STD_LOGIC_VECTOR (3 downto 0);
           saida : out  STD_LOGIC_VECTOR (3 downto 0);
          Flag_Zero : out STD_LOGIC;
          Flag_Sinal : out STD_LOGIC);
end COMPONENT Modulo_And;

COMPONENT incrementa1 is
```

```
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           saida : out  STD_LOGIC_VECTOR (3 downto 0);
           Flag_Zero : out STD_LOGIC;
           Flag_Sinal : out STD_LOGIC);
end COMPONENT incrementa1;

signal Not_A, Compl_A, A_mais_B, A_mais_1, A_menos_B, A_and_B, A_xor_B, A_or_B : STD_LOGIC_VECTOR (3 downto 0);
signal cout_somador, bout_subtrator : STD_LOGIC;
signal Flag_Zero_somador, Flag_Sinal_somador, Flag_Overflow_somador, Flag_Cout_somador, Flag_Zero_complementador, Flag_Sinal_complementador : STD_LOGIC;
signal Flag_Zero_inversor, Flag_Sinal_inversor, Flag_Zero_subtrator, Flag_Sinal_subtrator, Flag_Overflow_subtrator, Flag_Borrow_subtrator : STD_LOGIC;
signal Flag_Zero_xor, Flag_Sinal_xor, Flag_Zero_or, Flag_Sinal_or, Flag_Zero_and, Flag_Sinal_and, Flag_Zero_incrementa, Flag_Sinal_incrementa : STD_LOGIC;

begin

    -- Declarando os componentes
    U0: Inversor port map (A, Not_A, Flag_Zero_inversor, Flag_Sinal_inversor); -- Para operação 0
    U1: Complementa port map (A, Compl_A, Flag_Zero_complementador, Flag_Sinal_complementador); -- Para operação 1
```

```
U2: somapain4 port map (A, B, '0', cout_somador, A_mais_B, Flag_Zero_somador, Flag_Sinal_somador, Flag_Overflow_somador, Flag_Cout_somador); -- Para operação 2

U3: incrementa1 port map (A, A_mais_1, Flag_Zero_incrementa, Flag_Sinal_incrementa); -- Para operação 3

U4: subtratain4 port map (A, B, '0', bout_subtrator, A_menos_B, Flag_Zero_subtrator, Flag_Sinal_subtrator, Flag_Overflow_subtrator, Flag_Borrow_subtrator); -- Para operação 4

U5: Modulo_And port map (A, B, A_and_B, Flag_Zero_and, Flag_Sinal_and); -- Para operação 5

U6: Modulo_Xor port map (A, B, A_xor_B, Flag_Zero_xor, Flag_Sinal_xor); -- Para operação 6

U7: Modulo_Or port map (A, B, A_or_B, Flag_Zero_or, Flag_Sinal_or); -- Para operação 7


process (Operacao, Not_A, Compl_A, A_mais_B, A_mais_1, A_menos_B, A_and_B, A_xor_B, A_or_B)

begin

        case Operacao is

                when "0000" =>

                        Z <= A_and_B;

                        Flag_Zero <= Flag_Zero_and;

                        Flag_Sinal <= Flag_Sinal_and;

                        Flag_Overflow <= '0';

                        Flag_Cout <= '0';

                when "1000" =>

                        Z <= A_and_B;
```

```
            Flag_Zero <= Flag_Zero_and;
            Flag_Sinal <= Flag_Sinal_and;
            Flag_Overflow <= '0';
            Flag_Cout <= '0';
        when "0001" =>
            Z <= A_xor_B;
            Flag_Zero <= Flag_Zero_xor;
            Flag_Sinal <= Flag_Sinal_xor;
            Flag_Overflow <= '0';
            Flag_Cout <= '0';
        when "1001" =>
            Z <= A_xor_B;
            Flag_Zero <= Flag_Zero_xor;
            Flag_Sinal <= Flag_Sinal_xor;
            Flag_Overflow <= '0';
            Flag_Cout <= '0';
        when "0010" =>
            Z <= A_mais_1;
            Flag_Zero <= Flag_Zero_incrementa;
            Flag_Sinal <= Flag_Sinal_incrementa;
            Flag_Overflow <= '0';
            Flag_Cout <= '0';
        when "1010" =>
            Z <= A_mais_1;
            Flag_Zero <= Flag_Zero_incrementa;
```

```
            Flag_Sinal <= Flag_Sinal_incrementa;
            Flag_Overflow <= '0';
            Flag_Cout <= '0';
        when "0011" =>
            Z <= A_menos_B;
            Flag_Zero <= Flag_Zero_subtrator;
            Flag_Sinal <= Flag_Sinal_subtrator;
            Flag_Overflow <= Flag_Overflow_subtrator;
            Flag_Cout <= Flag_Borrow_subtrator;
        when "1011" =>
            Z <= A_menos_B;
            Flag_Zero <= Flag_Zero_subtrator;
            Flag_Sinal <= Flag_Sinal_subtrator;
            Flag_Overflow <= Flag_Overflow_subtrator;
            Flag_Cout <= Flag_Borrow_subtrator;
        when "0100" =>
            Z <= A_or_B;
            Flag_Zero <= Flag_Zero_or;
            Flag_Sinal <= Flag_Sinal_or;
            Flag_Overflow <= '0';
            Flag_Cout <= '0';
        when "1100" =>
            Z <= A_or_B;
            Flag_Zero <= Flag_Zero_or;
            Flag_Sinal <= Flag_Sinal_or;
```

```
            Flag_Overflow <= '0';
            Flag_Cout <= '0';
        when "0101" =>
            Z <= A_mais_B;
            Flag_Zero <= Flag_Zero_somador;
            Flag_Sinal <= Flag_Sinal_somador;
            Flag_Overflow <= Flag_Overflow_somador;
            Flag_Cout <= Flag_Cout_somador;
        when "1101" =>
            Z <= A_mais_B;
            Flag_Zero <= Flag_Zero_somador;
            Flag_Sinal <= Flag_Sinal_somador;
            Flag_Overflow <= Flag_Overflow_somador;
            Flag_Cout <= Flag_Cout_somador;
        when "0110" =>
            Z <= Compl_A;
            Flag_Zero <= Flag_Zero_or;
            Flag_Sinal <= Flag_Sinal_or;
            Flag_Overflow <= '0';
            Flag_Cout <= '0';
        when "1110" =>
            Z <= Compl_A;
            Flag_Zero <= Flag_Zero_complementador;
            Flag_Sinal <= Flag_Sinal_complementador;
            Flag_Overflow <= '0';
```

```
                Flag_Cout <= '0';
            when "0111" =>
                Z <= Not_A;
                Flag_Zero <= Flag_Zero_inversor;
                Flag_Sinal <= Flag_Sinal_inversor;
                Flag_Overflow <= '0';
                Flag_Cout <= '0';
            when "1111" =>
                Z <= Not_A;
                Flag_Zero <= Flag_Zero_inversor;
                Flag_Sinal <= Flag_Sinal_inversor;
                Flag_Overflow <= '0';
                Flag_Cout <= '0';
            when others =>
                z <= Not_A;
                Flag_Zero <= Flag_Zero_inversor;
                Flag_Sinal <= Flag_Sinal_inversor;
                Flag_Overflow <= '0';
                Flag_Cout <= '0';

        end case;
    end process;

end Behavioral;
```

III. Conclusão

Concluímos que o trabalho além da reafirmação de que é necessário dar muitas instruções para um computador entender algo supostamente básico para um humano, há também a percepção da quantidade de tópicos diferentes e úteis na disciplina de Sistemas Digitais.

IV. Apêndice

https://github.com/DantasB/DigitalSystem