Universidade Federal do Rio de Janeiro
Departamento de Engenharia Eletrônica e de Computação
EEL480- Laboratório de Sistemas Digitais

Unidade-Lógico-Aritmética (ULA) de 4 Bits

Data: 10/05/2019

Turma: EL4

Nomes:

Iverton Darlan Rodrigues Nunes
DRE:
Ass.:_____

Paulo Henrique Bucco dos Santos Caetano
DRE: 116169635
Ass.:_____

1. Introdução

Este projeto tem por objetivo o desenvolvimento de uma Unidade-Lógico-Aritmética (ULA) de 4 Bits que realiza 8 operações. Ela deve conter um sistema de interface para o teste em uma placa de desenvolvimento. Determinamos essas 8 operações divididas em dois grupos, o Lógico, que atribui as funções AND, OR, NOT e XOR, e o Aritmético, que atribui as funções de soma (+), subtração (-), multiplicação (*) e incremento de 1 (+1).

Uma Unidade-Lógico-Aritmética (ULA) é parte fundamental de sistemas mais complexos, pois se trata de um circuito digital que realiza operações aritméticas e booleanas entre operandos de N bits. Possui uma forma de seleção que determina a operação a ser realizada, chamada de "chave seletora". A saída indica o resultado da operação e, também, pode ter N bits, dependendo do que foi realizado na ULA.

2. Descrição da Implementação

A ULA foi desenvolvida a partir de blocos menores que se relacionam dentro de um sistema gerenciador. O particionamento em módulos menores, em certo ponto de vista, torna mais fácil e simples a estruturação lógica da tarefa a ser realizada. Dito isso, podemos listar abaixo cada módulo e descrição do que foi realizado no projeto.

2.1 - Operações Aritméticas:

2.1.1 - Soma (+)

Definimos que a partir de um bloco somador de 1 Bit, ficaria mais fácil a realização da soma de 4 Bits, pois esse último bloco usaria repetidas vezes o bloco mais simplista.

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MySum1 is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           Cin : in  STD_LOGIC;
           S : out  STD_LOGIC;
           Cout : out  STD_LOGIC);
end MySum1;

architecture Behavioral of MySum1 is

begin
    S <= A xor B xor Cin;
    Cout <= (A and B) or (Cin and A) or (Cin and B);

end Behavioral;
```

Módulo MySum1 (Somador de 1 bit)

"S" é a saída do bloco que contém o resultado e ela recebe as entradas "A", "B" e "Cin" (carry de entrada). A saída é atribuída após a realização da operação lógica XOR, padronizada pela biblioteca do sistema IEEE, entre as entradas. "Cout" (carry de saída) também é uma saída do bloco, porém não carrega o resultado

da operação, apenas uma informação a respeito do que foi realizado. Comumente é chamada de bit de transporte e leva esse nome porque indica que houve uma extrapolação na operação de soma. No “Cout” as entradas se relacionam por meio de AND’s e OR’s, que também são padronizados pelo sistema IEEE.

A soma de 4 Bits dentro da ULA usará o bloco acima da seguinte forma:

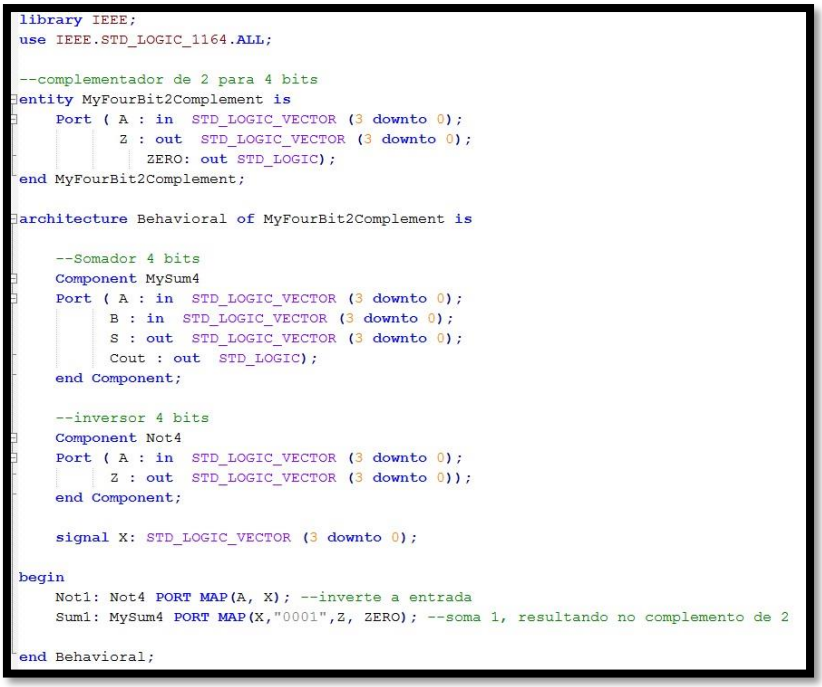
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MySum4 is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           S : out  STD_LOGIC_VECTOR (3 downto 0);
           Cout : out  STD_LOGIC);
end MySum4;

architecture Behavioral of MySum4 is

    component MySum1
        Port ( A : in STD_LOGIC;
        B : in STD_LOGIC;
        Cin : in STD_LOGIC;
        S : out STD_LOGIC;
        Cout : out STD_LOGIC);
    end component;

    signal X: STD_LOGIC_VECTOR(3 downto 0);
    signal Y: STD_LOGIC_VECTOR(2 downto 0);
    signal C: STD_LOGIC_VECTOR(2 downto 0);

begin
    Sum1: MySum1 PORT MAP(A(0),B(0),'0',S(0),X(0));
    Sum2: MySum1 PORT MAP(A(1),B(1),'0',Y(0),X(1));
    Sum3: MySum1 PORT MAP(A(2),B(2),'0',Y(1),X(2));
    Sum4: MySum1 PORT MAP(A(3),B(3),'0',Y(2),X(3));

    Sum5: MySum1 PORT MAP(X(0),Y(0),'0',S(1),C(0));
    Sum6: MySum1 PORT MAP(X(1),Y(1),C(0),S(2),C(1));
    Sum7: MySum1 PORT MAP(X(2),Y(2),C(1),S(3),C(2));
    Cout <= X(3) or C(2);

end Behavioral;
```

Módulo MySum4 (Somador de 4 bits)

Para cada bit de saída “S” (S0, S1, S2 e S3) que relaciona os bits de entrada de “A” (A0, A1, A2 e A3) e “B” (B0, B1, B2 e B3) e “C” (carry de entrada), a função “MySum1” (soma de 1 bit) é chamada. No módulo acima, é realizada a soma bit a bit das entradas A e B da ULA e é retornado o valor da soma em 4 bits de saída (S0, S1, S2 e S3). Neles está contido, em binário, o valor resultado da soma binária de dois números de 4 bits cada. Desta forma, o processo de soma aritmética das entradas da ULA se tornou mais simples e eficaz após a modularização.

2.1.2 - Subtração (-)

Na realização da operação aritmética de subtração, utilizamos do sistema Complemento a 2 para tornar a subtração uma soma de um número binário positivo com um número binário negativo representado em complemento de 2, afim de torná-la mais simples. Na mudança do segundo número para deixá-lo no sistema de complemento a 2, utilizamos o seguinte módulo conversor:

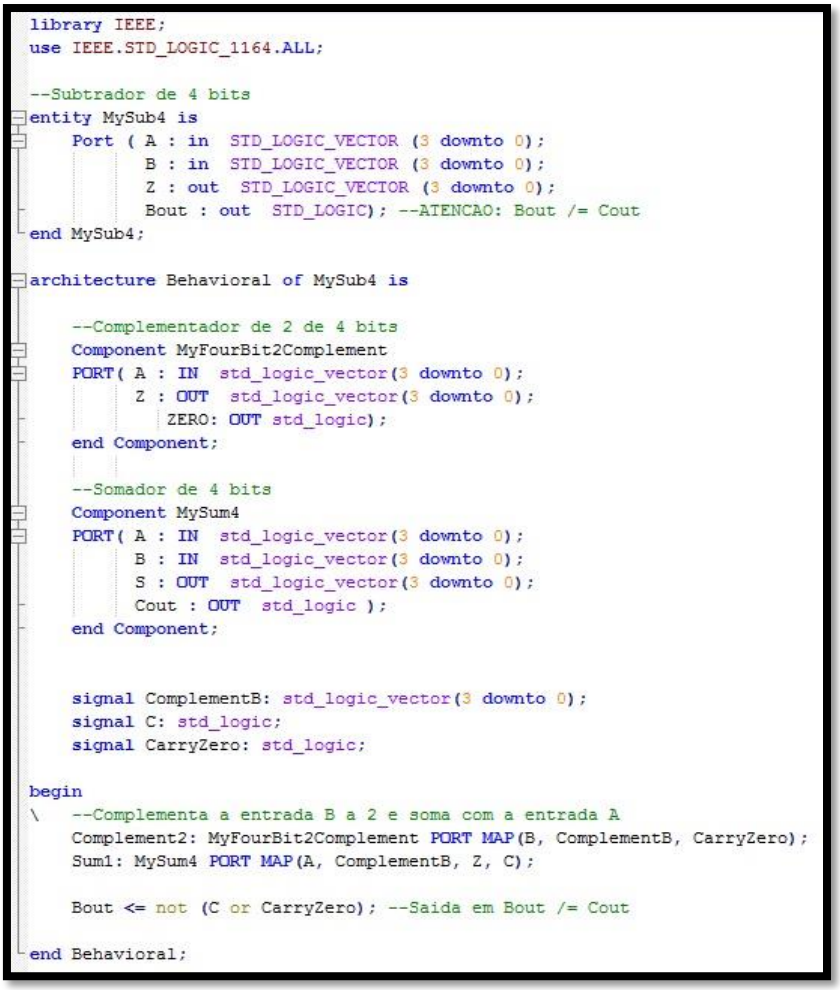


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--complementador de 2 para 4 bits
entity MyFourBit2Complement is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           Z : out  STD_LOGIC_VECTOR (3 downto 0);
           ZERO: out STD_LOGIC);
end MyFourBit2Complement;

architecture Behavioral of MyFourBit2Complement is

    --Somador 4 bits
    Component MySum4
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           S : out  STD_LOGIC_VECTOR (3 downto 0);
           Cout : out  STD_LOGIC);
    end Component;

    --inversor 4 bits
    Component Not4
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           Z : out  STD_LOGIC_VECTOR (3 downto 0));
    end Component;

    signal X: STD_LOGIC_VECTOR (3 downto 0);

begin
    Not1: Not4 PORT MAP(A, X); --inverte a entrada
    Sum1: MySum4 PORT MAP(X,"0001",Z, ZERO); --soma 1, resultando no complemento de 2

end Behavioral;
```

Módulo MyFourBit2Complement (Complementador a 2 de 4 bits)

No módulo acima da nossa ULA, utilizamos um módulo aritmético de soma de números com 4 bits (“MySum4”, em 2.1.1) e um módulo lógico inversor de números de 4 bits (“Not4”), que será mostrado na continuação deste relatório, para modificar o número desejado e deixá-lo em sistema de complemento a 2. O número em questão será sempre o composto dos bits de entradas B (B0, B1, B2 e B3) da nossa ULA, pois será ele que queremos deixar como um número negativo no sistema complemento de 2. Nesse sistema, invertemos as N bits entradas do número e depois somamos a ele o valor 1 (em N bits também). A saída desse módulo é o valor da entrada selecionada em complemento de 2 e, portanto, pronto para ser realizada a operação de subtração com a outra entrada não modificada.

Para realizarmos a subtração, agora que o segundo número de entrada está em sistema complemento a 2, deveremos apenas somar a entrada não modificada ao complemento a 2 da segunda entrada, obtendo assim a soma dos dois números de 4 bits. Utilizamos o seguinte módulo para isso:



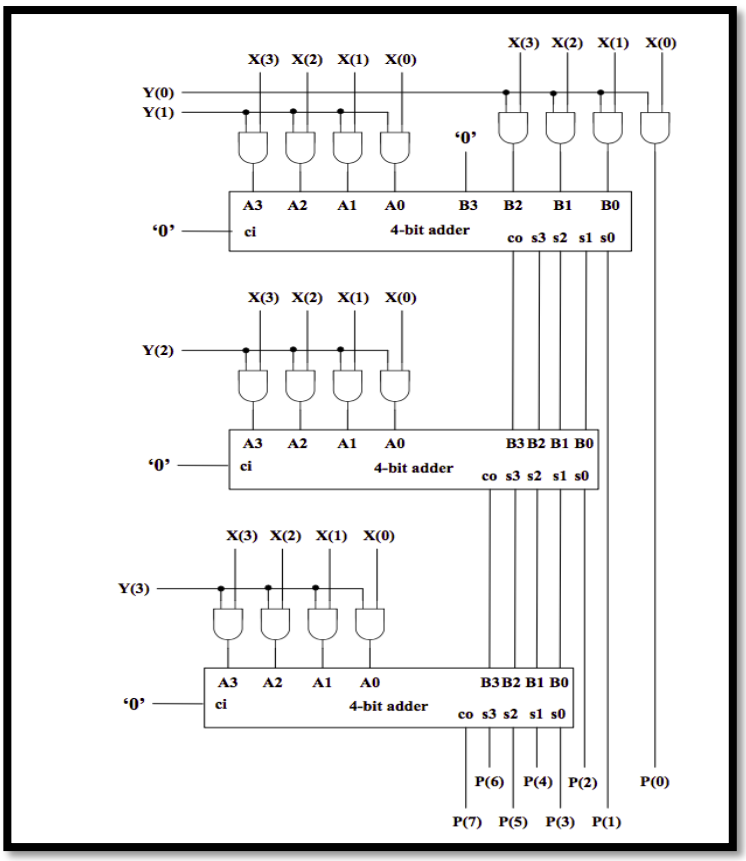
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--Subtrador de 4 bits
entity MySub4 is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Z : out  STD_LOGIC_VECTOR (3 downto 0);
           Bout : out  STD_LOGIC); --ATENCAO: Bout /= Cout
end MySub4;

architecture Behavioral of MySub4 is

    --Complementador de 2 de 4 bits
    Component MyFourBit2Complement
    PORT( A : IN  std_logic_vector(3 downto 0);
          Z : OUT  std_logic_vector(3 downto 0);
          ZERO: OUT std_logic);
    end Component;

    --Somador de 4 bits
    Component MySum4
    PORT( A : IN  std_logic_vector(3 downto 0);
          B : IN  std_logic_vector(3 downto 0);
          S : OUT  std_logic_vector(3 downto 0);
          Cout : OUT  std_logic );
    end Component;


    signal ComplementB: std_logic_vector(3 downto 0);
    signal C: std_logic;
    signal CarryZero: std_logic;

begin
    --Complementa a entrada B a 2 e soma com a entrada A
    Complement2: MyFourBit2Complement PORT MAP(B, ComplementB, CarryZero);
    Sum1: MySum4 PORT MAP(A, ComplementB, Z, C);

    Bout <= not (C or CarryZero); --Saida em Bout /= Cout

end Behavioral;
```

Módulo MySub4 (Subtrator de 4 bits)

No módulo acima da nossa ULA, utilizamos os módulos de soma de 4 bits (“MySum4”, em 2.1.1) e complementador a 2 de 4 bits (“MyFourBit2Complement”, em 2.1.2) para realizar a subtração de 2 números de 4 bits cada. Nele primeiro se complementa a 2 a entrada B e depois soma com a entrada A, dando o resultado “Sum1” e outra saída apenas operacional, que é o “Bout” (Borrow de saída). O “Bout” é atribuído a partir de operações NOT e OR, padronizadas pelo sistema IEEE, do “C” e “CarryZero”, que são os carry’s da soma e do complemento a 2.

2.1.3 - Multiplicação (*)

Na realização da operação aritmética de multiplicação da nossa ULA, utilizamos repetidas vezes o módulo somador de 4 bits (“MySum4”, em 2.1.1) afim de tornar mais simples e entendível a operação por completo. Baseamos o nosso módulo e em outro esquemático, mostrado a seguir:



Esquemático do multiplicador de 4 bits

A partir da ideia desse esquemático fizemos o módulo abaixo:

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

--multiplicador de 4 bits utilizando 3 somadores
entity MyFourBitMultiplier is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Z : out  STD_LOGIC_VECTOR (7 downto 0));
end MyFourBitMultiplier;

architecture Behavioral of MyFourBitMultiplier is

    --Somador 4 bits
    Component MySum4
        Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
               B : in  STD_LOGIC_VECTOR (3 downto 0);
               S : out  STD_LOGIC_VECTOR (3 downto 0);
               Cout : out  STD_LOGIC);
    end Component;

    --resultados dos produtos de cada bit do vetor B pelo vetor A
    signal Z1: STD_LOGIC_VECTOR (3 downto 0);
    signal Z2: STD_LOGIC_VECTOR (3 downto 0);
    signal Z3: STD_LOGIC_VECTOR (3 downto 0);
    signal Z4: STD_LOGIC_VECTOR (3 downto 0);

    --carry dos somadores
    signal C: STD_LOGIC_VECTOR (2 downto 0);
    --saida dos somadores
    signal S1: STD_LOGIC_VECTOR (3 downto 0);
    signal S2: STD_LOGIC_VECTOR (3 downto 0);
    signal S3: STD_LOGIC_VECTOR (3 downto 0);

begin
    Z1(0) <= A(0) and B(0);
    Z1(1) <= A(1) and B(0);
    Z1(2) <= A(2) and B(0);
    Z1(3) <= A(3) and B(0);

    Z2(0) <= A(0) and B(1);
    Z2(1) <= A(1) and B(1);
    Z2(2) <= A(2) and B(1);
    Z2(3) <= A(3) and B(1);
```

Módulo MyFourBitMultiplier (Multiplicador de 4 bits) - parte 1

```
    Z3(0) <= A(0) and B(2);
    Z3(1) <= A(1) and B(2);
    Z3(2) <= A(2) and B(2);
    Z3(3) <= A(3) and B(2);

    Z4(0) <= A(0) and B(3);
    Z4(1) <= A(1) and B(3);
    Z4(2) <= A(2) and B(3);
    Z4(3) <= A(3) and B(3);

    Sum1:
        MySum4 PORT MAP(
            A => Z2,
            B(3) => '0',
            B(2) => Z1(3),
            B(1) => Z1(2),
            B(0) => Z1(1),
            S => S1,
            Cout => C(0));

    Sum2:
        MySum4 PORT MAP(
            A => Z3,
            B(3) => C(0),
            B(2) => S1(3),
            B(1) => S1(2),
            B(0) => S1(1),
            S => S2,
            Cout => C(1));

    Sum3:
        MySum4 PORT MAP(
            A => Z4,
            B(3) => C(1),
            B(2) => S2(3),
            B(1) => S2(2),
            B(0) => S2(1),
            S => S3,
            Cout => C(2));
```

Módulo MyFourBitMultiplier (Multiplicador de 4 bits) - parte 2

```
    Z(0) <= Z1(0);
    Z(1) <= S1(0);
    Z(2) <= S2(0);
    Z(3) <= S3(0);
    Z(4) <= S3(1);
    Z(5) <= S3(2);
    Z(6) <= S3(3);
    Z(7) <= C(2);

end Behavioral;
```

Módulo MyFourBitMultiplier (Multiplicador de 4 bits) - parte 3

No módulo multiplicador da nossa ULA, usamos 3 módulos somadores de 4 bits (“MySum4”, em 2.1.1) e a operação lógica AND, padronizada pelo sistema IEEE. Como sabemos, a saída com o resultado da operação deve conter a soma do número de bits das duas entradas, portanto Z irá de 0 a 7. Cada saída Z (Z0, Z1, Z2, Z3, Z4, Z5, Z6 e Z7) recebe os valores atribuídos no corpo do código acima.

2.1.4 - Incremento de 1 (+1)

Na realização da operação aritmética de incrementação de 1 da nossa ULA, usamos o módulo de soma de 4 bits (“MySum4”, em 2.1.1) para incrementar “0001” aos 4 bits de entrada. Ele se resume a nada mais do que uma soma fixa da entrada de N bits com o vetor de N bits com valor unitário. O módulo está descrito abaixo:

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

--Incrementador de 1
entity MyPlus1 is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           Z : out  STD_LOGIC_VECTOR (3 downto 0);
           Cout : out  STD_LOGIC);
end MyPlus1;

architecture Behavioral of MyPlus1 is

    --Somador 4 bits
    Component MySum4
    Port( A : in  STD_LOGIC_VECTOR (3 downto 0);
          B : in  STD_LOGIC_VECTOR (3 downto 0);
          S : out  STD_LOGIC_VECTOR (3 downto 0);
          Cout : out  STD_LOGIC);
    end Component;

begin
    Soma: MySum4 PORT MAP(A, "0001", Z, Cout); --Soma 0001 ao vetor de entrada

end Behavioral;
```

Módulo MyPlus1 (Incrementador de 1 em 4 bits)

2.2 - Operações Lógicas:

2.2.1 - AND

Na realização operação lógica AND da nossa ULA, usamos a função lógica AND padronizada pelo sistema IEEE 4 vezes relacionando as entradas A (A0, A1, A2 e A3) e B (B0, B1, B2 e B3), respectivamente. As saídas Z (Z0, Z1, Z2 e Z3) são atribuídas com essas relações e formam o resultado da nossa operação. O módulo está descrito abaixo:

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MyAnd4 is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Z : out  STD_LOGIC_VECTOR (3 downto 0));
end MyAnd4;

architecture Behavioral of MyAnd4 is

begin
    Z(0) <= A(0) and B(0);
    Z(1) <= A(1) and B(1);
    Z(2) <= A(2) and B(2);
    Z(3) <= A(3) and B(3);

end Behavioral;
```

Módulo MyAnd4 (Operação lógica AND de 4 bits)

2.2.2 - OR

Na realização operação lógica OR da nossa ULA, usamos a função lógica OR padronizada pelo sistema IEEE 4 vezes relacionando as entradas A (A0, A1, A2 e A3) e B (B0, B1, B2 e B3), respectivamente. As saídas Z (Z0, Z1, Z2 e Z3) são atribuídas com essas relações e formam o resultado da nossa operação. O módulo está descrito abaixo:

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MyOr4 is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Z : out  STD_LOGIC_VECTOR (3 downto 0));
end MyOr4;

architecture Behavioral of MyOr4 is

begin
    Z(0) <= A(0) or B(0);
    Z(1) <= A(1) or B(1);
    Z(2) <= A(2) or B(2);
    Z(3) <= A(3) or B(3);


end Behavioral;
```

Módulo MyOr4 (Operação lógica OR de 4 bits)

2.2.3 - NOT

Na realização operação lógica NOT da nossa ULA, usamos a função lógica NOT padronizada pelo sistema IEEE uma vez, relacionando a entrada A (vetor de 4 bits) a saída Z (vetor de 4 bits). Esse último vetor forma o resultado da nossa operação inversora. O módulo está descrito abaixo:

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Not4 is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           Z : out  STD_LOGIC_VECTOR (3 downto 0));
end Not4;

architecture Behavioral of Not4 is

begin
    Z <= not A;

end Behavioral;
```

Módulo Not4 (Operação lógica NOT de 4 bits)

2.2.4 - XOR

Na realização operação lógica XOR da nossa ULA, usamos a função lógica XOR padronizada pelo sistema IEEE 4 vezes relacionando as entradas A (A0, A1, A2 e A3) e B (B0, B1, B2 e B3), respectivamente. As saídas Z (Z0, Z1, Z2 e Z3) são atribuídas com essas relações e formam o resultado da nossa operação. O módulo está descrito abaixo:

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MyXor4 is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Z : out  STD_LOGIC_VECTOR (3 downto 0));
end MyXor4;

architecture Behavioral of MyXor4 is

begin
    Z(0) <= A(0) xor B(0);
    Z(1) <= A(1) xor B(1);
    Z(2) <= A(2) xor B(2);
    Z(3) <= A(3) xor B(3);

end Behavioral;
```

Módulo MyXor4 (Operação lógica XOR de 4 bits)

2.3 - Unidade-Lógico-Aritmética:

A Unidade-Lógico-Aritmética funciona basicamente chamando os módulos lógicos ou aritméticos descritos acima, nas seções 2.1 e 2.2 do capítulo 2. Escolhemos um seletor para determinar qual operação o sistema da ULA deve realizar. Chamamos esse vetor seletor de “SeletorOperacao” e ele é um vetor de 3 bits, no qual podem ser descritas as 8 operações do sistema. As operações tem valores fixos nessa seleção, que são determinados conforme lista abaixo:

- “000” - AND de 4 bits (“MyAnd4”, em 2.2.1);
- “001” - OR de 4 bits (“MyOr4”, em 2.2.2);
- “010” - NOT de 4 bits (“Not4”, em 2.2.3);
- “011” - XOR de 4 bits (“MyXor4”, em 2.2.4);
- “100” - Soma de 4 bits (“MySum4”, em 2.1.1);
- “101” - Incrementador de 1 de 4 bits (“MyPlus1”, em 2.1.4);
- “110” - Subtração de 4 bits (“MySub4”, em 2.1.2);
- “111” - Multiplicador de 4 bits (“MyFourBitMultiplier”, em 2.1.3).

Após o módulo ter sido selecionado, a ULA atribui o resultado a saída “saidaZ”. Há, também, uma saída dependente da operação selecionada no sistema da ULA, chamada de “CarryBorrow”. Essa saída carrega o Carry ou Borrow da operação de soma (“MySum4”, em 2.1.1) ou subtração (“MySub4”, em 2.1.2), respectivamente. O módulo da Unidade-Lógico-Aritmética (ULA) está descrito abaixo:

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MyULA is
    Port ( entradaX : in  STD_LOGIC_VECTOR (3 downto 0);
           entradaY : in  STD_LOGIC_VECTOR (3 downto 0);
           SeletorOperacao : in  STD_LOGIC_VECTOR (2 downto 0);
           saidaZ : out  STD_LOGIC_VECTOR (3 downto 0);
           CarryBorrow : out  STD_LOGIC);
end MyULA;

architecture Behavioral of MyULA is

    Component MyAnd4
        Port( A : in  STD_LOGIC_VECTOR (3 downto 0);
              B : in  STD_LOGIC_VECTOR (3 downto 0);
              Z : out  STD_LOGIC_VECTOR (3 downto 0));
    end Component;

    Component MyOr4
        Port( A : in  STD_LOGIC_VECTOR (3 downto 0);
              B : in  STD_LOGIC_VECTOR (3 downto 0);
              Z : out  STD_LOGIC_VECTOR (3 downto 0));
    end Component;

    Component Not4
        Port( A : in  STD_LOGIC_VECTOR (3 downto 0);
              Z : out  STD_LOGIC_VECTOR (3 downto 0));
    end Component;
```

Módulo MyULA - parte 1

```
    Component MyXor4
        Port( A : in  STD_LOGIC_VECTOR (3 downto 0);
              B : in  STD_LOGIC_VECTOR (3 downto 0);
              Z : out  STD_LOGIC_VECTOR (3 downto 0));
    end Component;

    Component MySum4
        Port( A : in  STD_LOGIC_VECTOR (3 downto 0);
              B : in  STD_LOGIC_VECTOR (3 downto 0);
              S : out  STD_LOGIC_VECTOR (3 downto 0);
              Cout : out  STD_LOGIC);
    end Component;

    Component MySub4
        Port( A : in  STD_LOGIC_VECTOR (3 downto 0);
              B : in  STD_LOGIC_VECTOR (3 downto 0);
              Z : out  STD_LOGIC_VECTOR (3 downto 0);
              Bout : out  STD_LOGIC);
    end Component;

    Component MyFourBitMultiplier
        Port( A : in  STD_LOGIC_VECTOR (3 downto 0);
              B : in  STD_LOGIC_VECTOR (3 downto 0);
              Z : out  STD_LOGIC_VECTOR (7 downto 0));
    end Component;

    Component MyPlus1
        Port( A : in  STD_LOGIC_VECTOR (3 downto 0);
              Z : out  STD_LOGIC_VECTOR (3 downto 0);
              Cout : out  STD_LOGIC);
    end Component;

    --Operacoes lógicas
    signal saidaMyAnd4: STD_LOGIC_VECTOR (3 downto 0);
    signal saidaMyOr4: STD_LOGIC_VECTOR (3 downto 0);
    signal saidaMyNot4: STD_LOGIC_VECTOR (3 downto 0);
    signal saidaMyXor4: STD_LOGIC_VECTOR (3 downto 0);
```

Módulo MyULA - parte 2

```
    --Operacoes Aritméticas
    signal saidaMySum4: STD_LOGIC_VECTOR (3 downto 0);
    signal saidaMyPlus1: STD_LOGIC_VECTOR (3 downto 0);
    signal saidaMySub4: STD_LOGIC_VECTOR (3 downto 0);
    signal saidaMyMult4: STD_LOGIC_VECTOR (7 downto 0);
    signal CoutMySum4: STD_LOGIC;
    signal CoutMyPlus1: STD_LOGIC;
    signal BorrowMySub4: STD_LOGIC;

begin

    --Operacoes lógicas
    TMyAnd4: MyAnd4 PORT MAP(entradaX, entradaY, saidaMyAnd4);
    TMyOr4: MyOr4 PORT MAP(entradaX, entradaY, saidaMyOr4);
    TMyNot4: Not4 PORT MAP(entradaX, saidaMyNot4);
    TMyXor4: MyXor4 PORT MAP(entradaX, entradaY, saidaMyXor4);
    --Operacoes Aritméticas
    TMySum4: MySum4 PORT MAP(entradaX, entradaY, saidaMySum4, CoutMySum4);
    TMyPlus1: MyPlus1 PORT MAP(entradaX, saidaMyPlus1, CoutMyPlus1);
    TMySub4: MySub4 PORT MAP(entradaX, entradaY, saidaMySub4, BorrowMySub4);
    TMyMult4: MyFourBitMultiplier PORT MAP(entradaX, entradaY, saidaMyMult4);

    WITH SeletorOperacao select
        saidaZ <=   saidaMyAnd4 when "000",
                    saidaMyOr4 when "001",
                    saidaMyNot4 when "010",
                    saidaMyXor4 when "011",
                    saidaMySum4 when "100",
                    saidaMyPlus1 when "101",
                    saidaMySub4 when "110",
                    saidaMyMult4(3) & saidaMyMult4(2) & saidaMyMult4(1) & saidaMyMult4(0) when "111",
                    "0000" when others;

   WITH SeletorOperacao select
        CarryBorrow <=    CoutMySum4 when "100",
                          CoutMyPlus1 when "101",
                          BorrowMySub4 when "110",
                          '0' when others;

end Behavioral;
```

Módulo MyULA - parte 3

2.4 - Interface:

2.4.1 - Contador:

A interface da nossa Unidade-Lógico-Aritmética, precisou, primeiramente, de um módulo contador para que pudéssemos exibir os resultados na sequência determinada em projeto (entrada A, entrada B, resultado). Este módulo contador, descrito abaixo, utiliza o clock de 50 MHz da placa para converter em um pulso de 6 segundos, que será usado para mostrar os números e resultados na ordem e tempo determinados em projeto.

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

entity contador6sec is
port(  clk_50Mhz : in std_logic; --clock de 50MHz da placa FPGA
       ENABLE: in std_logic; --ativa(1) ou desativa bloco contador
       CLEAR: in std_logic; --zera a contagem em 1
       counter : out integer range 300000000 downto 0; --são necessarios 300000000 clocks de 50MHz para contar 6 segundos
       clk_6sec : out std_logic := '0' --apenas para fins de teste do programa, clock com periodo de 6 segundos;
      );
end contador6sec;

architecture Behavioral of contador6sec is

begin

process(clk_50Mhz, CLEAR)
    variable counterTemp: integer range 300000000 downto 0; --VHDL não se da bem com autoatribuições em um sinal de saida.

    begin
        if(CLEAR = '1') then --zera a contagem e zera o sinal de clock de saida (assincrono)
            counterTemp := 0;
            clk_6sec <= '0';
        elsif( rising_edge(clk_50Mhz) ) then --transição positiva do clock de entrada (sincrono)
            if(ENABLE = '0') then --zera a contagem e zera o sinal de clock de saida
                clk_6sec <= '0';
                counterTemp := 0;
            else
                if(counterTemp = 0) then
                    clk_6sec <= '1';
                elsif(counterTemp = 150000000) then
                    clk_6sec <= '0';
                end if;

                if(counterTemp >= 300000000) then --retorna a contagem para zero
                    counterTemp := 0;
                else
                    counterTemp := counterTemp + 1; --incrementa 1 na contagem
                end if;
            end if;
        end if;
        counter <= counterTemp; --atribui o sinal temporario ao sinal de saida.
end process;

end Behavioral;
```

Módulo contador6sec (contador 6 segundos)

2.4.2 - Interface:

O programa da interface possui como entradas, o clock de 50MHz da placa, um botão para reset, um botão para mudar de estado e 4 switches para escolher os vetores de entrada. E possui como saídas, 4 LEDs responsáveis por mostrar o vetor de 4 bits e mais 4 bits para mostrar qual dentre os 4 vetores de bits está sendo mostrado (entrada A, entrada B, seleção de Operação, Resultado).

Começamos declarando as componentes do contador e da ULA, após isso, declaramos os estados referentes à máquina de estados, state_A, state_B, state_Operação, state_Result. Declaramos então variáveis para guardar o estado atual e indice do estado. Também declaramos variáveis para guardar os valores da entrada A, entrada B, seleção de operação, Carry/Borrow da ULA, Saída da ULA, contagem do contador, guardados nas respectivas variáveis Xtemp, Ytemp, SeletorOperacao, CarryBorrow, SaidaZ, numeroContador.

Começamos o programa com um “port map” da Unidade Logico Aritmética e do contador de 6 segundos (contador6sec, em 2.4.1). Com o detalhe que usaremos o botaoEntrada, ou seja, o botão que muda de estado, como clear do contador, ou seja, após clicar nesse botão o contador é zerado, e a contagem recomeça.

Após entrar no processo da máquina de estado, iniciamos declarando o efeito do botão de reset, o circuito retorna para o estado A. Temos então, um bloco responsável por mudar de estado e gravar os valores de entrada A, entrada B, seleção de operação.

O bloco de código a seguir é responsável por mostrar os vetores entrada A, entrada B, resultado, sequencialmente com intervalos de 2 segundos. O bloco final é responsável por atualizar os LEDs da esquerda com o valor atual dos switches durante os estados, state_A, state_B, state_Operacao.

Ao clicar no botãoEntrada no state_A, a entrada A é gravada com os valores dos switches.

Ao clicar no botãoEntrada no state_B, a entrada B é gravada com os valores dos switches.

Ao clicar no botãoEntrada no state_Operacao, o vetor de operação é gravado com os valores dos switches.

Ao clicar no botãoEntrada no state_Result, é selecionada uma nova operação.

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


entity MyULAInterface is
    port (clk_50Mhz: in std_logic; --clock de 50MHz da placa
          rst: in std_logic; --botao de reset
          botaoEntrada: in std_logic; -- botão para mudar de estado e inserir as entradas
          entradaUla: in std_logic_vector (3 downto 0); -- switches para escolher as entradas
          ledSelecao : out std_logic_vector (3 downto 0) := "1000" ;  -- leds para sinalizar qual vetor está sendo mostrado em led_Show
          led_Show : out std_logic_vector (3 downto 0)); -- leds para mostrar os vetores de bits

end MyULAInterface;

architecture interface of MyULAInterface is

    --Contador de 6 segundos
    component contador6sec
    port(  clk_50Mhz : in std_logic;
           ENABLE: in std_logic;
           CLEAR: in std_logic;
          counter : out integer range 300000000 downto 0;
          clk_6sec : out std_logic := '0' --apenas para de teste do programa, clock com periodo de 6 segundos;
        );

    end component;

    --Unidade Logico Aritmetica
    component MyULA
    Port ( entradaX : in  STD_LOGIC_VECTOR (3 downto 0);
           entradaY : in  STD_LOGIC_VECTOR (3 downto 0);
           SeletorOperacao : in  STD_LOGIC_VECTOR (2 downto 0);
           saidaZ : out  STD_LOGIC_VECTOR (3 downto 0);
           CarryBorrow : out  STD_LOGIC);
    end component;

    signal clk_6sec: std_logic; --Sinal para guardar o valor do clock do contador
```

Módulo MyULAInterface (ULA interface) - parte 1

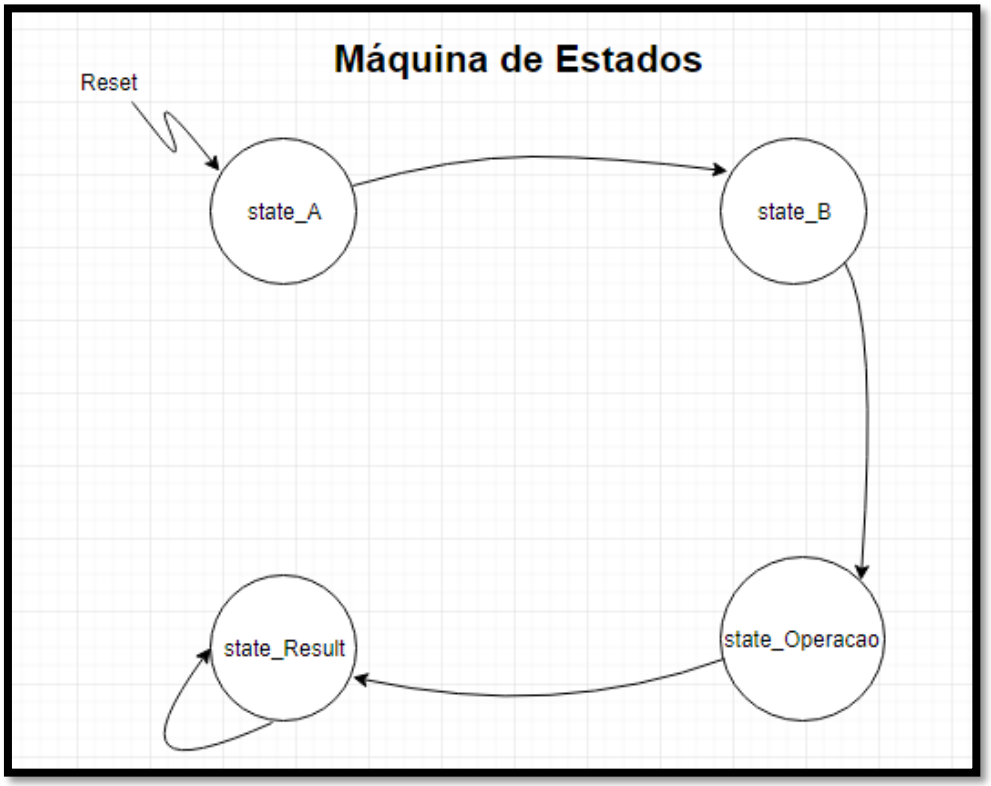
```
    type state is (state_A,state_B,state_Operacao,state_Result); --Estados da máquina de estados
    signal estado_Atual: state := state_A; --Estado Atual da máquina de estados
    signal indiceEstado: std_logic_vector (1 downto 0) := "10"; --Indice do estado da maquina de estados
    signal XTemp: STD_LOGIC_VECTOR (3 downto 0);  --Guarda o valor de X ao mudar de estado
    signal YTemp: STD_LOGIC_VECTOR (3 downto 0); --Guarda o valor de Y ao mudar de estado
    signal SeletorOperacao: STD_LOGIC_VECTOR (2 downto 0) := "000"; --Guarda a operacao selecionada ao mudar de estado
    signal CarryBorrow: STD_LOGIC; --Guarda o carry/Borrow da ULA
    signal SaidaZ : STD_LOGIC_VECTOR (3 downto 0); --Saida da ULA
    signal numeroContador: integer range 300000000 downto 0;    --Saida de contagem do contador

begin
    UnidadeLogicoAritmetica: myULA port map (XTemp, YTemp, SeletorOperacao, SaidaZ, CarryBorrow); --ULA
    contador6Segundos: contador6sec port map (clk_50Mhz, '1', botaoEntrada, numeroContador, clk_6sec); --Contador
    --Para o contador utilizamos botaoEntrada como clear, assim, ao pressionar o botaoEntrada, a contagem irá para 0

    maquinaDeEstado: process(botaoEntrada, rst, estado_Atual, numeroContador, entradaUla, XTemp, YTemp, SaidaZ)
    begin
        if rst = '1' then       --Reseta para o estado A
            XTemp <= "0000";
            YTemp <= "0000";
            estado_Atual <= state_A;
            indiceEstado <= "00";
            SeletorOperacao <= "000";
            led_Show <= entradaUla;
            ledSelecao <= "1000";
            indiceEstado <= "10";

        elsif(botaoEntrada'event and botaoEntrada = '1') then --Ao clicar no botaoEntrada, muda de estado
            if(numeroContador >= 5000000) then --Debouncing
                case estado_Atual is
                    when state_A => --Se estado é A, guarda valor de X, muda para estado B
                        XTemp <= entradaUla ;
                        ledSelecao <= "0100";
                        indiceEstado <= "01" ;
                        estado_Atual <= state_B;
                        led_Show <= entradaUla;
```

Módulo MyULAInterface (ULA interface) - parte 2

```
                when state_B => --Se estado é B, guarda valor de Y, muda para estado C
                    YTemp <= entradaUla;
                    ledSelecao <= "0010";
                    indiceEstado <= "00";
                    estado_Atual <= state_Operacao;
                    led_Show <= entradaUla;

                when state_Operacao => --Se estado é C, guarda qual é a operacao, muda para estado de resultado
                    seletorOperacao <= entradaUla(2) & entradaUla(1) & entradaUla(0);
                    ledSelecao <= "0001";
                    indiceEstado <= "11";
                    estado_Atual <= state_Result;
                    led_Show <= '0' & entradaUla(2) & entradaUla(1) & entradaUla(0);

                when state_Result => --Troca a operacao que esta sendo efetuada
                    seletorOperacao <= entradaUla(2) & entradaUla(1) & entradaUla(0);
            end case;
        end if;
    end if;

    if(estado_Atual = state_Result) then    --Mostra os resultados
        if numeroContador  < 100000000 then -- 2 segundos de contagem --Mostra o Valor de X até 2 sec
            led_Show <= XTemp;      --Entrada X
            ledSelecao <= "1000";
        elsif ((numeroContador >= 100000000) and (numeroContador < 200000000))  then --mostra o valor de Y entre 2 e 4 sec
            led_Show <= YTemp;    -- Entrada Y
            ledSelecao <= "0100";
        elsif numeroContador >= 200000000  then         -- Mostra o valor do resultado entre 4 e 6 sec
            led_Show <= SaidaZ;             -- Resultado
            ledSelecao <= "0001";
        end if;

    elsif(estado_Atual = state_A) then --Mostra o valor de X constantemente nos leds, mesmo que altere o switch
        led_Show <= entradaUla;
    elsif(estado_Atual = state_B) then --Mostra o valor de Y constantemente nos leds, mesmo que altere o switch
        led_Show <= entradaUla;
    elsif(estado_atual = state_Operacao) then --Mostra o indice da operacao constantemente nos leds, mesmo que altere o switch
        led_Show <= '0' & entradaUla(2) & entradaUla(1) & entradaUla(0);
    end if;
  end process maquinaDeEstado;
end interface;
```

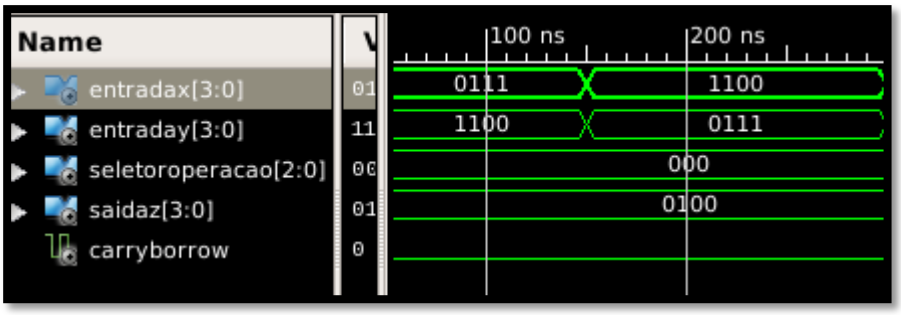
Módulo MyULAInterface (ULA interface) - parte 3

A seguir temos um diagrama de estados representando a máquina de estados implementada na interface.



3. Resultados

3.1 – AND



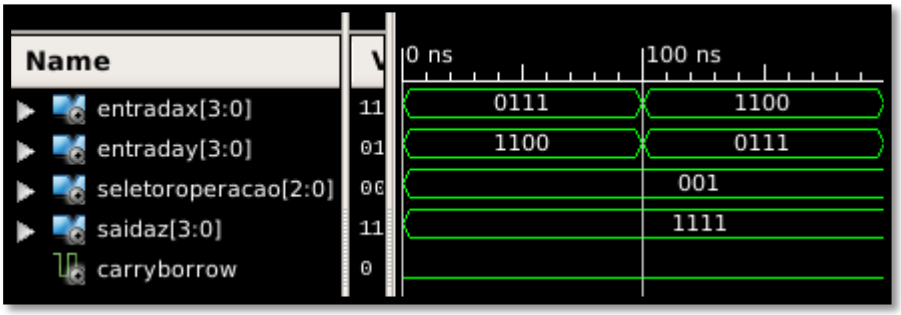
Na imagem começamos com A = 0111 e B=1100, após 100ns alternamos as entradas, A=1100 e B=0111. Vemos que carryborrow = 0, pois não é utilizado nessa operação. Observamos que alternar as entradas não altera a saída.

saídaz = 0111 and 1100 = 0100

saidaz = 1100 and 0111 = 0100

O resultado está correto.

3.2 – OR



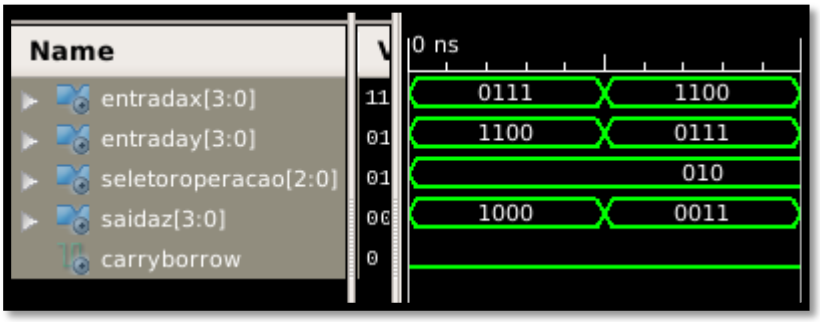
Na imagem começamos com A = 0111 e B=1100, após 100ns alternamos as entradas, A=1100 e B=0111. Vemos que carryborrow = 0, pois não é utilizado nessa operação. Observamos que alternar as entradas não altera a saída.

saídaz = 0111 or 1100 = 1111

saidaz = 1100 or 0111 = 1111

O resultado está correto.

3.3 – NOT



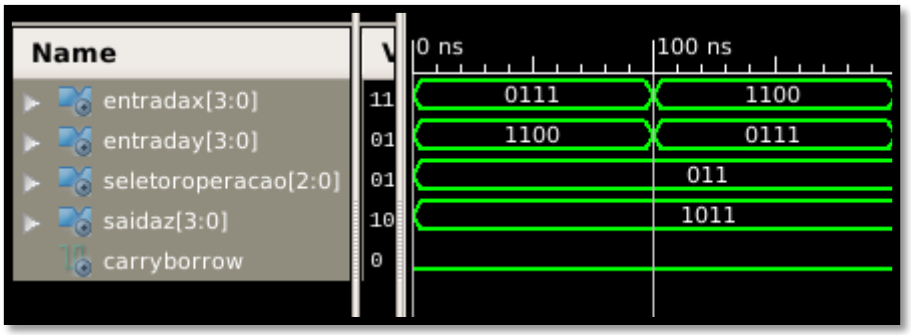
Na imagem começamos com A = 0111 e B=1100, após 100ns invertemos as entradas, A=1100 e B=0111. Vemos que carryborrow = 0, pois não é utilizado nessa operação. Apenas a entrada A é utilizada para este caso.

saídaz = not 0111 = 1000

saidaz = not 1100 = 0011

O resultado está correto.

3.4 – XOR



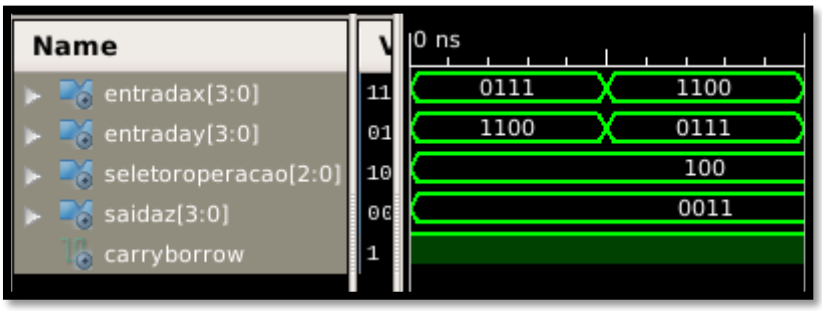
Na imagem começamos com A = 0111 e B=1100, após 100ns alternamos as entradas, A=1100 e B=0111. Vemos que carryborrow = 0, pois não é utilizado nessa operação. Observamos que alternar as entradas não altera a saída.

saídaz = 0111 and 1100 = 1011

saidaz = 1100 and 0111 = 1011

O resultado está correto.

3.5 – Soma (+)



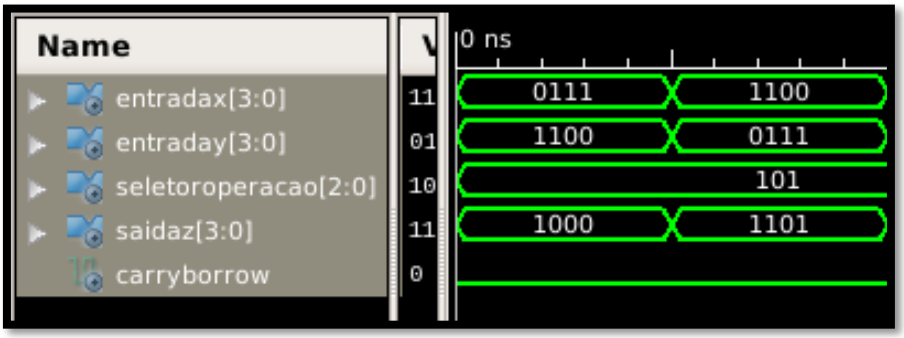
Na imagem começamos com A = 0111 e B=1100, após 100ns alternamos as entradas, A=1100 e B=0111. Observamos que alternar as entradas não altera a saída e que o carry é 1 nos dois casos.

saídaz = 0111 + 1100 = 0011 com carry = 1

saidaz = 1100 + 0111 = 0011 com carry = 1

O resultado está correto.

3.6 – Incremento de 1 (+1)



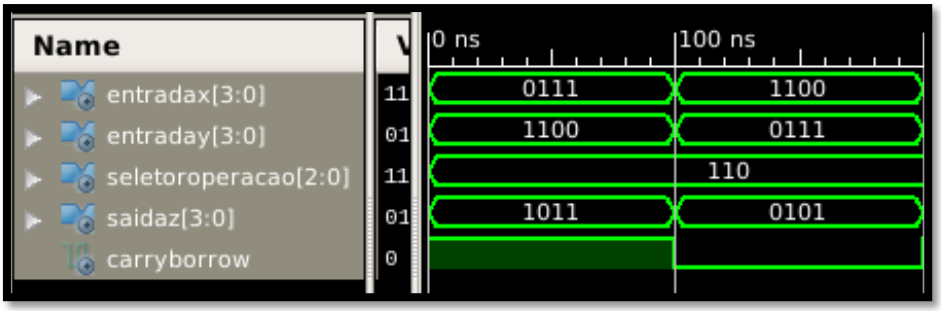
Na imagem começamos com A = 0111 e B=1100, após 100ns alternamos as entradas, A=1100 e B=0111. Apenas a entrada A é utilizada neste caso, observamos que mudar a entrada A altera a saída e que carryborrow se mantém em 0.

saídaz = incrementa1(0111) = 1000 com carry = 0

saidaz = incrementa1(1100) = 1101 com carry = 0

O resultado está correto.

3.7 – Subtrator (-)



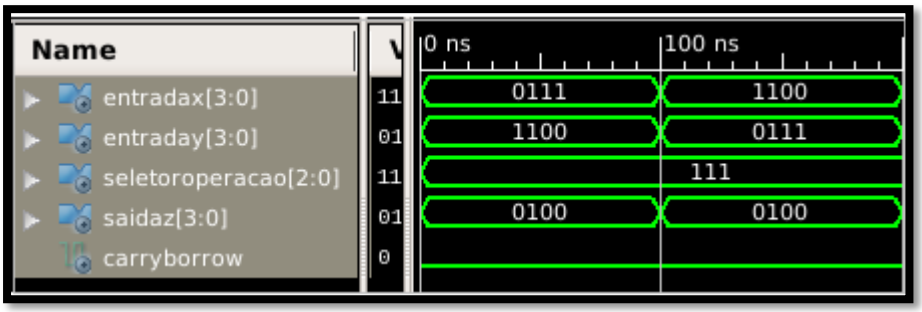
Na imagem começamos com A = 0111 e B=1100, após 100ns alternamos as entradas, A=1100 e B=0111. Observamos que alternar as entradas altera a saída. Observamos também que carry se mantém em 1 até a primeira metade e 0 na segunda meta.

saídaz = 0111 - 1100 = 1011 com borrow = 1

saidaz = 1100 - 0111 = 0101 com borrow = 0

O resultado está correto.

3.8 – Multiplicador (*)



Na imagem começamos com A = 0111 e B=1100, após 100ns invertemos as entradas, A=1100 e B=0111. Vemos que carryborrow = 0, pois não é utilizado nessa operação. Observamos que alternar as entradas não altera a saída. Apenas os últimos 4 bits são mostrados nessa operação.

saídaz = 0111 * 1100 = 01010100

saidaz = 1100 * 0111 = 01010100

O resultado está correto.

4. Conclusões

O projeto funciona conforme o esperado, no entanto foi observado um número excessivo de saídas não utilizadas. Isso ocorreu pelo fato de serem implementadas as funções imaginando-se o modelo real de cada módulo da ULA e do próprio sistema. Observamos em certo momento, as diferenças entre simulação e funcionamento real, em que foi possível notar o efeito de “bouncing” (trepidação) na placa de teste. Na simulação esse efeito não ocorre, pois trata-se de condições ideais de uso, sendo necessário uma pequena mudança no código do módulo da interface (capítulo 2.4) para corrigir este efeito, especificado como “Debouncing”.