Unidade Lógico Aritmética

Relatório do primeiro trabalho da prática da disciplina Sistemas Digitais - EEL480

Alunos	Clara Albino Pacheco, DRE: 122077676
	Thiago Marques de Oliveira, DRE: 122053397
	Matheus Soares Gonçalves, DRE: 122037773
Professor	João Baptista de Oliveira e Souza Filho
Turma	EL2
Horário	15:00 ás 17:00

Índice

1. Introdução

O relatório apresenta a descrição da implementação de uma Unidade Lógica-Aritmética em uma placa Xilinx Spartan. A ULA é implementada a partir de um código-fonte escrito em VHDL, linguagem de programação cujo principal objetivo é o desenvolvimento de circuitos integrados.

A unidade lógica-aritmética criada é um circuito integrado que permite a realização de oito operações, sendo essas soma, subtração, multiplicação, AND, NAND, OR, NOR e NOT. A ULA permite que, através da placa, o usuário insira dois vetores binários que serão utilizados para realizar as operações e um terceiro que seleciona a operação a ser executada. Posteriormente, o resultado é exibido na placa também como um vetor binário.

Os LEDs da placa são utilizados como uma forma de representar os vetores que entram e o resultado que sai da ULA. Inicialmente, os três LEDs mais à esquerda acendem, um de cada vez, para indicar o que o usuário está inserindo no circuito. O primeiro vetor de entrada das operações, que será representado como A, é inserido, seguido do segundo vetor de entrada, que será representado como B, seguido do vetor de seleção. Após a fase de inserção dos dados, o circuito passa para uma fase apenas de exibição, onde os quatro LEDs mais à esquerda acendem para indicar o que está sendo exibido e os quatro mais à direita exibem os vetores binários. Os três vetores escolhidos pelo usuário são mostrados, seguidos do resultado da operação selecionada. O retorno para a inserção de novos dados só pode ser feito através de um reset.

A introdução e posterior exposição dos vetores são feitas de forma sequencial, ou seja, são estabelecidas através de uma máquina de estados. A máquina de estados será o principal módulo de implementação da ULA e contará com um divisor de frequências associado a ela para geração de um sinal de clock de dois segundos. O desenvolvimento de tais entidades é abordado nas seções seguintes.

Para a realização dos códigos, foi importada a seguinte biblioteca:

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

Figura (1). Biblioteca importada

2. Descrição da Implementação da ULA

2.1 Máquina de estados

A unidade lógica-aritmética contém como seu principal módulo uma máquina de estados, implementada a partir da entidade *my_STATEMACHINE*. Todas as outras entidades se conectam e formam a lógica da ULA a partir dela.

A máquina de estados recebe como entrada um clock de 50MHz, *clk_50*, do tipo STD_LOGIC, um vetor de quatro bits inserido pelo usuário, *input*, do tipo STD_LOGIC_VECTOR e quatro entradas provenientes de botões da placa, *but_A*, *but_B*, *but_S* e *but_reset*, todos do tipo STD_LOGIC. O clock de 50MHz é passado como entrada do divisor de frequência, que o modifica de forma a retornar um clock de 2 segundos, associado ao signal *my_clk*. O vetor *input* pode assumir três categorias de valores diferentes, ele pode receber o valor de A, o valor de B ou a seleção inserida pelo usuário (S). A categoria assumida dependerá do estado atual da máquina. Por último, as entradas relacionadas aos botões são utilizadas para registrar os dados inseridos pelo usuário e para implementar a função reset. A entrada *but_A* registra o valor de A, *but_B*, o valor de B e *but_S*, o valor da seleção. O botão de reset, *but_reset*, faz com que a máquina de estados retorne ao seu estado inicial. É a única forma de fazer esse retorno.

A entidade também apresenta quatro saídas, o vetor *leds_direita*, do tipo STD_LOGIC_VECTOR, e quatro saídas do tipo STD_LOGIC, *LD7_A*, *LD6_B*, *LD5_S* e *LD4_OUT*. O vetor *leds_direita* pode assumir o valor de A, de B, da seleção S e do resultado das operações e é o responsável por acender os quatro LEDs mais à direita na placa. O valor assumido depende do estado atual da máquina. As outras saídas, por outro lado, são utilizadas para indicar qual valor está sendo mostrado pelo vetor *leds_direita* e são responsáveis pelos LEDs mais à esquerda na placa. Quando *LD7_A* estiver aceso, o valor de A será mostrado pelo vetor, quando *LD6_B* estiver aceso, o valor de B será exibido, quando *LD5_S*, a seleção do MUX será mostrada e, por fim, quando *LD4_OUT*, a resposta das operações aparecerá nos LEDs da placa.

Este módulo contém dois componentes instanciados dentro dele. Um que desempenha o comportamento de um MUX 8:1, chamado *my_MUX*, e um que trabalha como um divisor de frequências, o *my_FREQDIV*. O *my_MUX* se encarrega da realização das operações da ULA, enquanto o *my_FREQDIV* gera o clock com período de 2 segundos que é utilizado na máquina de estados.

```
entity my_STATEMACHINE is
    Port ( clk_50 : in  STD_LOGIC;
           input : in  STD_LOGIC_VECTOR (3 downto 0);
           but_A : in  STD_LOGIC;
           but_B : in  STD_LOGIC;
           but_S : in  STD_LOGIC;
           but_reset : in  STD_LOGIC;
           LD7_A : out  STD_LOGIC;
           LD6_B : out  STD_LOGIC;
           LD5_S : out  STD_LOGIC;
           LD4_OUT : out  STD_LOGIC;
           leds_direita : out  STD_LOGIC_VECTOR (3 downto 0));
end my_STATEMACHINE;
```

Figura (2). Entradas e saídas da entidade *my_STATEMACHINE*

```
architecture Behavioral of my_STATEMACHINE is

    component my_FREQDIV is
        Port ( CLOCK_50M : in  STD_LOGIC;
               my_CLOCK : out  STD_LOGIC);
    end component my_FREQDIV;

    component my_MUX is
        Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           S : in  STD_LOGIC_VECTOR (3 downto 0);
           Y : out  STD_LOGIC_VECTOR (3 downto 0) );
    end component my_MUX;
```

Figura (3). Componentes da entidade *my_STATEMACHINE*

A máquina de estados construída possui 35 estados, onde cada um desses estados representa uma determinada situação evidenciada pelos LEDs. Basicamente, as situações passíveis de representação se resumem em:

1. Inserção do valor de A
2. Inserção do valor de B
3. Inserção da seleção do MUX
4. Exposição do valor de A
5. Exposição do valor de B
6. Exposição da seleção

7. Exposição do resultado da operação

A implementação dos 35 estados é motivada pela necessidade de repetir o ciclo de exposição para cada uma das 8 possíveis operações da ULA. Cada operação conta com seu próprio ciclo de exposição dos dados.

Antes da máquina de estados começar a ser devidamente implementada, um novo tipo de dado é definido. Para possibilitar a representação das mudanças de estados da máquina foi definido o tipo *state*, que não recebe nenhum tipo de dado concreto existente na linguagem VHDL. Os dois *signals* definidos dentro desse tipo, *state_1* e *aux*, recebem somente a informação de estado atual e de próximo estado da máquina. Além desses, outros *signals* também são definidos. O *signal my_clk*, do tipo STD_LOGIC, recebe o clock de 2 segundos do divisor de frequências, enquanto *A*, *B* e *S* são *signals* do tipo STD_LOGIC_VECTOR que recebem, respectivamente, as três entradas A, B e seleção do usuário. Os últimos signals criados, *Y0*, …, *Y7*, do tipo STD_LOGIC_VECTOR, são utilizados na instanciação do MUX para cada uma das operações. Cada um desses vetores vai ser associado à resposta de uma operação.

```
type state is (state0, state1, state2, state3, state4, state5, state6, state7,
               state8, state9, state10, state11, state12, state13, state14,
               state15, state16, state17, state18, state19, state20, state21,
               state22, state23, state24, state25, state26, state27, state28,
               state29, state30, state31, state32, state33, state34);

signal state_1, aux : state := state0;
signal my_clk : STD_LOGIC;
signal A, B, S : STD_LOGIC_VECTOR (3 downto 0);
signal Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7 : STD_LOGIC_VECTOR (3 downto 0);
```

Figura (4). Type e signals da entidade *my_STATEMACHINE*

Uma vez que todos os dados estejam devidamente definidos, a implementação da máquina de estados é formalmente iniciada. O primeiro procedimento implementado é a instanciação dos componentes *my_FREQDIV* e *my_MUX* através da realização de alguns *port maps*. Primeiramente, o clock de 2 segundos é gerado e, após isso, cada uma das operações da ULA é conectada à máquina de estados por intermédio do multiplexador. As instâncias do multiplexador fornecem à máquina os resultados de cada uma das operações.

```
begin

    divisor_freq : my_FREQDIV port map(clk_50, my_clk);

    op_AND: my_MUX port map (A, B,"0000", Y0);
    op_NAND: my_MUX port map (A, B,"0001",Y1);
    op_OR: my_MUX port map (A, B,"0010", Y2);
    op_NOR: my_MUX port map (A, B,"0011", Y3);
    op_NOT: my_MUX port map (A, B,"0100", Y4);
    op_ADD: my_MUX port map (A, B,"0101", Y5);
    op_SUB: my_MUX port map (A, B,"0110", Y6);
    op_MULT: my_MUX port map (A, B,"0111", Y7);
```

Figura (5). Port maps da entidade *my_STATEMACHINE*

Em sequência, o clock com período de 2 segundos é implementado. A cada pulso de subida do clock, o estado armazenado em *aux* é passado para *state_1*, que representa o estado atual da máquina. Assim, a mudança de estado é realizada.

```
process (my_clk, aux)

begin
    if (rising_edge(my_clk)) then
        state_1 <= aux;
    end if;
end process;
```

Figura (6). Aplicação do clock da entidade *my_STATEMACHINE*

O segundo e último processo define quais ações são postas em prática em cada um dos estados. Inicialmente, o processo verifica o valor do reset, para determinar se o usuário deseja que a ULA retorne ao estado que permite o início da inserção dos valores de entrada, *state0*. Caso o *but_reset* esteja ativo, o valor *state0* é associado a *aux* e todos os LEDs da direita são apagados. O estado é devidamente atualizado no pulso de clock. Caso contrário, a máquina de estados permite que o usuário inicie a entrada de dados.

```
process (input, but_A, but_B, but_S, but_reset, state_1, aux,
            A, B, S, Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7)

begin

    if (but_reset = '1') then
        aux <= state0;
        leds_direita <= "0000";
```

Figura (7). Configuração do reset de *my_STATEMACHINE*

Uma vez dentro do *else*, através de uma estrutura *case … when*, há a verificação de qual é o estado atual da máquina e o desempenho das ações descritas nele. Os três primeiros estados são para coleta de dados, enquanto todos os outros são para exposição de dados.

No primeiro estado, *state0*, o LED mais a esquerda da placa acende, indicando ao usuário que ele deve inserir o valor de A. A máquina aguarda até que o usuário registre o valor, ativando o *but_A*, para associar a *aux* o próximo estado, *state1*. Enquanto *but_A* não é ativo, o valor de *aux* permanece *state0*. O *state1* funciona de forma semelhante, porém o LED que acende indica ao usuário que ele deve inserir o valor de B e a máquina aguarda a ativação do *but_B* para passar *state2* a *aux*.

O *state2* tem um funcionamento diferente, porque é nesse estado que a seleção do MUX é implementada. O funcionamento inicial dele é similar aos dois estados anteriores. O terceiro LED da esquerda para a direita irá acender, indicando que a seleção pode ser inserida na placa. Posteriormente, a máquina aguarda o registro do dado pela ativação do *but_S*. Entretanto, uma vez que o botão é ativado a máquina deve escolher um entre oito possíveis próximos estados. A escolha depende justamente da seleção introduzida na placa e cada um desses próximos estados representa o início da implementação de uma das operações da ULA.

Independentemente da escolha da operação, os próximos estados possuem uma estrutura genérica. O primeiro estado após a seleção tem como objetivo mostrar ao usuário o valor que foi inserido como A, a partir dos LEDs mais a direita, e indicar que é o vetor A que está sendo apresentado, acendendo o LED mais a esquerda da placa. Para isso, o valor de *A* é passado para a saída *leds_direita* e o valor ‘1’ é passado para *LD7_A*. Além disso, o próximo estado é associado a *aux*. O segundo estado posterior mostra o vetor B e indica que é ele que está sendo exibido, acendendo o segundo LED da esquerda para a direita da placa. O valor de

B é passado para a saída *leds_direita*, o valor '1' é passado para *LD6_B* e o próximo estado é associado a *aux*. O terceiro e o quarto estado seguintes a seleção funcionam na mesma lógica, entretanto os vetores expostos são, respectivamente, a seleção *S* e o resultado da operação (*Y0*, …, *Y7)*. Esse ciclo de estados ocorre para qualquer uma das operações da ULA.

```
else
    case state_1 is
        when state0 => --entro com A
            LD7_A <= '1';
            LD6_B <= '0';
            LD5_S <= '0';
            LD4_OUT <= '0';

            if (but_A = '1') then
                A <= input;
                aux <= state1;
            else
                aux <= state0;
            end if;

        when state1 => -- entro com B
            LD7_A <= '0';
            LD6_B <= '1';
            LD5_S <= '0';
            LD4_OUT <= '0';

            if (but_A = '1') then
                A <= input;
                aux <= state2;
            else
                aux <= state1;
            end if;
```

```
when state2 => --entro com a selecao
    LD7_A <= '0';
    LD6_B <= '0';
    LD5_S <= '1';
    LD4_OUT <= '0';
    if (but_S = '1') then
        S <= input;
        if (S = "0000") then
            aux <= state3;
        elsif (S = "0001") then
            aux <= state7;
        elsif (S = "0010") then
            aux <= state11;
        elsif (S = "0011") then
            aux <= state15;
        elsif (S = "0100") then
            aux <= state19;
        elsif (S = "0101") then
            aux <= state23;
        elsif (S = "0110") then
            aux <= state27;
        elsif (S = "0111") then
            aux <= state31;
        end if;
    else
        aux <= state2;
    end if;
```

Figura (8) e (9). Implementação de *state0*, *state1* e *state2* em *my_STATEMACHINE*

Após a máquina prosseguir para o estado correspondente a operação selecionada, ela parte para a operação associada a seleção seguinte e em diante. Caso o reset não seja acionado, a máquina roda por todas as operações e reinicia o ciclo em *state3*, estado que representa a primeira seleção do MUX, para a operação AND.

```
when state3 => -- and
    LD7_A <= '1';
    LD6_B <= '0';
    LD5_S <= '0';
    LD4_OUT <= '0';
    leds_direita <= A;
    aux <= state4;

when state4 =>
    LD7_A <= '0';
    LD6_B <= '1';
    LD5_S <= '0';
    LD4_OUT <= '0';
    leds_direita <= B;
    aux <= state5;
```

```
when state5 => --exibe a seleção do and
    LD7_A <= '0';
    LD6_B <= '0';
    LD5_S <= '1';
    LD4_OUT <= '0';
    leds_direita <= "0000";
    aux <= state6;

when state6 =>
    LD7_A <= '0';
    LD6_B <= '0';
    LD5_S <= '0';
    LD4_OUT <= '1';
    leds_direita <= Y0;
    aux <= state7;
```

Figura (10) e (11). Exemplo do ciclo de exposição aplicado a operação AND em *my_STATEMACHINE*

```
when state34 =>
    LD7_A <= '0';
    LD6_B <= '0';
    LD5_S <= '0';
    LD4_OUT <= '1';
    leds_direita <= Y7;
    aux <= state3;

when others =>
    null;
```

Figura (12). Representação da volta à primeira operação após o último estado. Representação de uma anulação diante de estados não existentes.

A declaração *when others* no código garante que nada acontece na placa caso a máquina de estados seja direcionada para um estado que não um dos idealizados. Uma possibilidade, por exemplo, seria a entrada de uma seleção não contemplada pelo MUX.

Segue o código completo da máquina de estados: my_STATEMACHINE

2.2 Divisor de frequências

O divisor de frequências tem como objetivo a geração de um sinal de clock com um período de 2 segundos a partir do clock padrão da placa utilizada, de frequência 50MHz (período 0.02µs) . A implementação é feita a partir da entidade *my_FREQDIV*.

A entidade tem apenas uma entrada, *CLOCK_50*, e uma saída, *my_CLOCK*, as duas do tipo STD_LOGIC. A entrada tem associado a ela um clock de 50MHz e a saída recebe, no final do código, o novo clock de 2 segundos.

```
entity my_FREQDIV is
    Port ( CLOCK_50M : in  STD_LOGIC;
           my_CLOCK : out  STD_LOGIC);
end my_FREQDIV;
```

Figura (13). Entrada e saída da entidade *my_FREQDIV*

Antes da implementação da geração do novo clock, dois signals são criados, *saida*, do tipo STD_LOGIC, que recolhe o valor do novo clock e o passa para *my_CLOCK*, e *counter*, do tipo INTEGER, que vai de 0 até 49999999. Seu valor máximo é o valor de oscilações que o sinal *CLOCK_50* apresenta em 1 segundo.

Após essas definições, um processo é iniciado. Inicialmente, há uma verificação se o *CLOCK_50* está ou não subindo. Caso esteja, duas possibilidades são apresentadas:

1 - Se o *counter* já tiver chegado ao valor máximo, 49999999, então *saida* recebe o seu próprio valor invertido por uma porta NOT e *counter* é zerado.

2 - Se o *counter* ainda não estiver em seu valor máximo, ele é incrementado.

Com essa condicional em cima do *counter*, o código garante que o *CLOCK_50* suba 50 milhões de vezes, ou seja, garante uma passagem de tempo de $(50M \times 0.02\mu) = 1$ segundo antes da inversão do *signal saida*. Essa condicional é repetida continuamente com as mudanças em *CLOCK_50*, possibilitando a construção de um sinal, *saída*, que permanece 1 segundo em '0' e 1 segundo em '1'. O *signal saida* será atribuído a *my_CLOCK* como o clock de 2 segundos.

```
architecture Behavioral of my_FREQDIV is

    signal saida: STD_LOGIC;
    signal counter: INTEGER range 0 to 49999999 := 0;

begin
    CLK: process (CLOCK_50M) begin

        if (rising_edge(CLOCK_50M)) then
            if (counter = 49999999) then
                saida <= not (saida);
                counter <= 0;
            else
                counter <= counter + 1;
            end if;
        end if;
    end process;

    my_CLOCK <= saida;

end Behavioral;
```

Figura (14). *Signals* e criação do clock de 2 segundos em *my_FREQDIV*

2.3 Multiplexador

Com o objetivo de permitir que o usuário escolha a operação a ser realizada, foi implementado um multiplexador 8:1 (seletor), que tem por entrada os vetores A, B e S, de 4 bits. A e B são os argumentos das operações, enquanto S é a codificação da operação desejada.

```
entity my_MUX is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           S : in  STD_LOGIC_VECTOR (3 downto 0);
           Y : out  STD_LOGIC_VECTOR (3 downto 0) );
end my_MUX;
```

Figura (15). Entradas e saídas da entidade *my_MUX*

Para multiplexar, cada operação recebeu um código de 0000 à 0111, entre os quais o usuário escolhe para selecionar a operação que deseja realizar. Tais entradas são declaradas na entidade *my_MUX*. Segue a correspondência entre seleção e operação:

- 0000 - AND
- 0001 - NAND
- 0010 - OR
- 0011 - NOR
- 0100 - NOT
- 0101 - SOMA
- 0110 - SUBTRAÇÃO
- 0111 - MULTIPLICAÇÃO

O MSB da seleção deve ser zero. Caso contrário, não haverá operação correspondente e a máquina de estados aplicará a condição *when others => null*. Nada será apresentado na placa.

Na *architecture*, cada uma das operações foi declarada através de um *component* e, posteriormente, foram instanciadas com o *port map.* Todas as operações são calculadas pelo módulo, mas a exibição na interface dependerá da seleção. Os resultados são armazenados nos vetores *d0, d1, d2, d3, d4, d5, d6 e d7,* que correspondem aos resultados das operações *AND, NAND, OR, NOR, NOT, SOMA, SUBTRAÇÃO* e *MULTIPLICAÇÃO*, respectivamente.

A lógica de seleção em si, é feita usando a estrutura condicional if-elsif, onde a condição de verificação é o vetor *S*. A saída selecionada é atribuída, dentro do escopo de cada if, à variável *Y*, que é o vetor que será, de fato, exibido na interface. A estrutura de verificação foi criada dentro de um *process,* a fim de só ser verificada e alterar a variável *Y* se houver alteração no seletor, e consequentemente nos vetores que guardam os resultados parciais (*d0 - d7*).

```
architecture Behavioral of my_MUX is

    component my_AND is
        Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
               B : in  STD_LOGIC_VECTOR (3 downto 0);
               Y : out  STD_LOGIC_VECTOR (3 downto 0) );
    end component my_AND;

    component my_NAND is
        Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
               B : in  STD_LOGIC_VECTOR (3 downto 0);
               W : out  STD_LOGIC_VECTOR (3 downto 0) );
    end component my_NAND;

    component my_OR is
        Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
               B : in  STD_LOGIC_VECTOR (3 downto 0);
               X : out  STD_LOGIC_VECTOR (3 downto 0) );
    end component my_OR;

    component my_NOR is
        Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
                B : in  STD_LOGIC_VECTOR (3 downto 0);
               V : out  STD_LOGIC_VECTOR (3 downto 0) );
    end component my_NOR;

    component my_NOT is
        Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
               Z : out  STD_LOGIC_VECTOR (3 downto 0) );
    end component my_NOT;
```

```
component my_FOURBITSUM is
    Port ( A, B : in STD_LOGIC_VECTOR (3 downto 0);
              C_out : out STD_LOGIC;
           sum : out STD_LOGIC_VECTOR (3 downto 0) );
end component my_FOURBITSUM;

component my_FOURBITSUBTRACTOR is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           DIFF : out  STD_LOGIC_VECTOR (3 downto 0) );
end component my_FOURBITSUBTRACTOR;

component my_FOURBITMULTIPLIER is
  Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Y : out  STD_LOGIC_VECTOR (3 downto 0) );
end component my_FOURBITMULTIPLIER;
```

Figuras (16) e (17). *Components* da entidade *my_MUX*

```
    signal d7, d6, d5, d4, d3, d2, d1, d0: STD_LOGIC_VECTOR (3 downto 0);
    signal carry_out : STD_LOGIC;

begin

    OP_AND : my_AND port map(A, B, d0);
    OP_NAND : my_NAND port map(A, B, d1);
    OP_OR : my_OR port map(A, B, d2);
    OP_NOR : my_NOR port map(A, B, d3);
    OP_NOT : my_NOT port map(A, d4);
    OP_ADD : my_FOURBITSUM port map(A, B, carry_out, d5);
    OP_SUB : my_FOURBITSUBTRACTOR port map(A, B, d6);
    OP_MULT : my_FOURBITMULTIPLIER port map(A, B, d7);
```

Figuras (18). *Signals* e *port maps* da entidade *my_MUX*

```
processo_1 : process(S, d0, d1, d2, d3, d4, d5, d6, d7)

    begin

        if (S = "0000") then --and
            Y <= d0;

        elsif (S = "0001") then --nand
            Y <= d1;

        elsif (S = "0010") then --or
            Y <= d2;

        elsif (S = "0011") then --nor
            Y <= d3;

        elsif (S = "0100") then --not
            Y <= d4;

        elsif (S = "0101") then --adder
            Y <= d5;

        elsif (S = "0110") then --subtractor
            Y <= d6;

        elsif (S = "0111") then --multiplier
            Y <= d7;

        end if;
end process processo_1;
```

Figuras (19). Condicional *if-elsif* da entidade *my_MUX*

2.4 AND

Para a operação AND, foi criada a entidade *my_AND* que recebe dois vetores de 4 bits como entrada, *A* e *B*, e produz um vetor de 4 bits como saída *Y*.

Além disso, foi implementada uma arquitetura denominada *Behavioral* para a entidade *my_AND*. Nesta parte do código, foi utilizado o operador lógico AND entre os vetores de entrada *A* e *B*, com o resultado sendo atribuído ao vetor de saída Y. Neste caso, cada bit do vetor de saída corresponde ao resultado da operação aplicada bit a bit entre *A* e *B*.

```
entity my_AND is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Y : out  STD_LOGIC_VECTOR (3 downto 0));
end my_AND;

architecture Behavioral of my_AND is

begin

 Y <= A AND B;

end Behavioral;
```

Figura (20). my_AND

2.5 NAND

Para a operação NAND, foi criada a entidade *my_NAND* que recebe dois vetores de 4 bits como entrada, *A* e *B*, e produz um vetor de 4 bits como saída *W*.

Além disso, foi implementada uma arquitetura denominada *Behavioral* para a entidade *my_NAND*. Nesta parte do código, foi utilizado o operador lógico NAND entre os vetores de entrada *A* e *B*, com o resultado sendo atribuído ao vetor de saída W. Neste caso, cada bit do vetor de saída corresponde ao resultado da operação aplicada bit a bit entre *A* e *B*.

```
entity my_NAND is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           W : out  STD_LOGIC_VECTOR (3 downto 0));
end my_NAND;

architecture Behavioral of my_NAND is

begin

    W <= A NAND B;

end Behavioral;
```

Figura (21). my_NAND

2.6 OR

Para a operação OR, foi criada a entidade *my_OR* que recebe dois vetores de 4 bits como entrada, *A* e *B*, e produz um vetor de 4 bits como saída *X*.

Além disso, foi implementada uma arquitetura denominada *Behavioral* para a entidade *my_OR*. Nesta parte do código, foi utilizado o operador lógico OR entre os vetores de entrada *A* e *B*, com o resultado sendo atribuído ao vetor de saída *X*. Neste caso, cada bit do vetor de saída corresponde ao resultado da operação aplicada bit a bit entre *A* e *B*.

```
entity my_OR is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           X : out  STD_LOGIC_VECTOR (3 downto 0));
end my_OR;

architecture Behavioral of my_OR is

begin

    X <= A OR B;

end Behavioral;
```

Figura (22). *my_OR*

2.7 NOR

Para a operação NOR, foi criada a entidade *my_NOR* que recebe dois vetores de 4 bits como entrada, *A* e *B*, e produz um vetor de 4 bits como saída, *V*.

Além disso, foi implementada uma arquitetura denominada *Behavioral* para a entidade *my_NOR*. Nesta parte do código, foi utilizado o operador lógico NOR entre os vetores de entrada *A* e *B*, com o resultado sendo atribuído ao vetor de saída *V*. Neste caso, cada bit do vetor de saída corresponde ao resultado da operação aplicada bit a bit entre *A* e *B*.

```
entity my_NOR is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           V : out  STD_LOGIC_VECTOR (3 downto 0));
end my_NOR;

architecture Behavioral of my_NOR is

begin

    V <= A NOR B;

end Behavioral;
```

Figura (23). *my_NOR*

2.8 NOT

Para a operação NOT, foi criada a entidade *my_NOT* que recebe um vetor de 4 bits como entrada, *A*, e produz um vetor de 4 bits como saída, *Z*.

Além disso, foi implementada uma arquitetura denominada *Behavioral* para a entidade *my_NOT*. Nesta parte do código, foi utilizado o operador lógico NOT sobre o vetor de entrada *A*, com o resultado sendo atribuído ao vetor de saída *Z*. Neste caso, cada bit do vetor de saída corresponde ao resultado da operação aplicada bit a bit a *A*.

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

architecture Behavioral of my_NOT is

begin

    Z <= NOT A;

end Behavioral;
```

Figura (24). *my_NOT*

2.9 Somador

Para a operação de soma, foi implementado um somador de 4 bits utilizando quatro somadores completos de 1 bit. Para a representação do somador de 4 bits, foi criada a

entidade *my_FOURBITSUM*, que utiliza quatro instâncias de um somador completo de 1 bit para realizar a soma bit a bit (considerando os carries).

O somador completo realiza a soma de três bits, as duas entradas da operação mais o carry-in, produzindo tanto o resultado da soma, quanto um bit de carry-out. A entidade *my_FULLADDER* foi criada para o representar.

A entidade *my_FULLADDER* tem como entradas *A*, *B* e *C*, do tipo STD_LOGIC, onde *A* e *B* são os valores de entrada da soma e *C* é o bit que representa o carry-in. Já para as saídas, temos o *CARRY*, que indica se houve carry-out na soma atual, e *SUM*, que indica o resultado da soma. As duas são do tipo STD_LOGIC.

```
entity my_FULLADDER is
    Port ( A, B, C : in  STD_LOGIC;
           CARRY, SUM : out  STD_LOGIC);
end my_FULLADDER;
```

Figura (25). Entradas e saídas da entidade *my_FULLADDER*

Além disso, foi implementada uma arquitetura denominada *Behavioral* para a entidade *my_FULLADDER*. Nesta parte do código, foram realizadas as operações lógicas que formam o somador de 1 bit, com os resultados sendo atribuídos às saídas *SUM* e *CARRY*. Os *signals XOR_1*, *AND_1* e *AND_2* foram criados para facilitar a visualização das operações.

A entidade *my_FOURBITSUM* tem como entradas dois vetores de 4 bits, *A* e *B*, um vetor de saída de 4 bits, *sum*, que representa o resultado da soma, e uma saída de 1 bit, *C_out*, que representa o último carry gerado pela soma. Todos são do tipo STD_LOGIC_VECTOR.

Foi implementada uma arquitetura denominada *Behavioral* para a entidade *my_FOURBITSUM*. Nesta parte do código, foi criado um *signal* denominado *c*, um vetor de 4 bits para armazenar os carries gerados durante a soma, e foi declarado o componente *my_FULLADDER*, que representa um somador completo de 1 bit. Posteriormente, 4 instâncias do somador completo foram criadas (*ADD0*, *ADD1*, *ADD2* e *ADD3*), através do *port map*, onde cada uma delas representa uma chamada ao componente *my_FULLADDER* para realizar a soma bit a bit dos vetores de entrada.

```
architecture Behavioral of my_FULLADDER is

    signal XOR_1, AND_1, AND_2: STD_LOGIC;

begin

    XOR_1 <= A XOR B;
    AND_1 <= A AND B;
    AND_2 <= C AND XOR_1;
    SUM <= XOR_1 XOR C;
    CARRY <= AND_1 OR AND_2;
```

Figura (26). Operações realizadas e saídas atribuídas a elas para a montagem da soma de 1 bit

```
entity my_FOURBITSUM is
    Port ( A, B : in STD_LOGIC_VECTOR (3 downto 0);
           C_out : out STD_LOGIC;
           sum : out STD_LOGIC_VECTOR (3 downto 0));
end my_FOURBITSUM;
```

Figura (27). Entradas e saídas da entidade *my_FOURBITSUM*

- Explicação das instâncias:
    - ADD0: Soma dos bits menos significativos de *A* e *B*. Para o carry in, como não ocorreu nenhuma soma anteriormente, foi atribuído o valor 0. O resultado da soma foi atribuído ao bit menos significativo do vetor *sum*, e o carry out da soma foi atribuído ao bit menos significativo do vetor *c*.
    - ADD1: Soma dos segundos bits menos significativos de *A* e *B*. Para o carry in, foi atribuído o valor do carry out da primeira soma. O resultado da soma foi atribuído ao segundo bit menos significativo do vetor *sum*, e o carry out da soma foi atribuído ao segundo bit menos significativo do vetor *c*.
    - ADD2: Soma dos segundos bits mais significativos de *A* e *B*. Para o carry in, foi atribuído o valor do carry out da segunda soma. O resultado da soma foi atribuído ao segundo bit mais significativo do vetor *sum*, e o carry out da soma foi atribuído ao segundo bit mais significativo do vetor *c*.
    - ADD3: Soma dos bits mais significativos de *A* e *B*. Para o carry in, foi atribuído o valor do carry out da terceira soma. O resultado da soma foi

atribuído ao bit mais significativo do vetor *sum*, e o carry out da soma foi atribuído ao bit mais significativo do vetor *c*.

- Observações:
  - No fim, o último carry gerado pela soma (o bit mais significativo de *c*) é atribuído à saída *C_out* para utilização no código do multiplicador.
  - Não foi abordado um meio para indicar na placa se houve carry out para o somador de 4 bits.

```
architecture Behavioral of my_FOURBITSUM is

    signal c: STD_LOGIC_VECTOR (3 downto 0);

    component my_FULLADDER
        port(A, B, C: in STD_LOGIC;
                sum, carry: out STD_LOGIC);
    end component;

    begin

        ADD0: my_FULLADDER port map (A(0), B(0), '0', sum(0), c(0));
        ADD1: my_FULLADDER port map (A(1), B(1), c(0), sum(1), c(1));
        ADD2: my_FULLADDER port map (A(2), B(2), c(1), sum(2), c(2));
        ADD3: my_FULLADDER port map (A(3), B(3), c(2), sum(3), c(3));

        C_out <= c(3);

end Behavioral;
```

Figura (28). *Signal* c, componente *my_FULLADDER* e lógica das instâncias da entidade *my_FOURBITSUM*

2.10 Subtrator

Para a operação de subtração, foi implementado um subtrator de 4 bits utilizando quatro subtratores completos de 1 bit. Para a representação do subtrator de 4 bits, foi criada a entidade *my_FOURBITSUBTRACTOR*, que utiliza quatro instâncias de um subtrator completo de 1 bit para realizar a subtração bit a bit.

O subtrator completo realiza a subtração de três bits, as duas entradas da operação e o borrow in, produzindo tanto o resultado da subtração, quanto um bit de borrow out. A entidade *my_SUBTRACTOR1BIT* foi criada para o representar.

A entidade *my_SUBTRACTOR1BIT* tem como entradas *A*, *B* e *B_in*, do tipo STD_LOGIC, onde *A* e *B* são os valores de entrada da subtração e *B_in* é o borrow in. Já para as saídas, temos o *B_out*, que representa o borrow out, e *Difference*, que indica o resultado da subtração, com as duas também sendo do tipo STD_LOGIC.

```
entity my_SUBTRACTOR1BIT is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           B_in : in  STD_LOGIC;
           Difference : out  STD_LOGIC;
           B_out : out  STD_LOGIC);
end my_SUBTRACTOR1BIT;
```

Figura (29). Entradas e saídas da entidade *my_SUBTRACTOR1BIT*

Além disso, foi implementada uma arquitetura denominada *Behavioral* para a entidade *my_SUBTRACTOR1BIT*. Nesta parte do código, foram realizadas as operações lógicas que formam o subtrator de 1 bit, com os resultados sendo atribuídos às saídas *Difference* e *B_out*. Também foram criados os *signals XOR_1*, *AND_1* e *AND_2* para a visualização das operações.

A entidade *my_FOURBITSUBTRACTOR* tem como entradas dois vetores de 4 bits, *A* e *B*, e um vetor de saída de 4 bits, *DIFF*, que representa o resultado da subtração. Todos são do tipo STD_LOGIC_VECTOR.

```
architecture Behavioral of my_SUBTRACTOR1BIT is

    signal XOR_1, AND_1, AND_2 : STD_LOGIC;

begin

    XOR_1 <= A XOR B;
    Difference <= XOR_1 XOR B_in;
    AND_1 <= not A and B;
    AND_2 <= not XOR_1 and B_in;
    B_out <= AND_1 or AND_2;

end Behavioral;
```

Figura (30). Operações realizadas e saídas atribuídas a elas para a montagem da subtração de 1 bit

```
entity my_FOURBITSUBTRACTOR is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           DIFF : out  STD_LOGIC_VECTOR (3 downto 0));
end my_FOURBITSUBTRACTOR;
```

Figura (31). Entradas e saídas da entidade *my_FOURBITSUBTRACTOR*

Além disso, foi implementada uma arquitetura denominada *Behavioral* para a entidade *my_FOURBITSUBTRACTOR*. Nesta parte do código, foi criado um *signal* denominado *borrow*, um vetor de 4 bits para armazenar os borrows gerados durante a subtração e foi declarado o componente *my_SUBTRACTOR1BIT*, que representa um subtrator completo. Posteriormente, 4 instâncias do somador completo foram criadas (*SUB0*, *SUB1*, *SUB2* e *SUB3*), através do *port map*, onde cada uma delas representa uma chamada ao componente *my_SUBTRACTOR1BIT* para realizar a subtração bit a bit dos vetores de entrada.

- Explicação das instâncias:
    - SUB0: Utiliza o bit menos significativo das entradas *A* e *B* para realizar a primeira operação de subtração. Para o borrow in, como não ocorreu nenhuma subtração anteriormente, foi atribuído o valor 0. O resultado da subtração foi atribuído ao bit menos significativo do vetor *DIFF*, e o borrow out da subtração foi atribuído ao bit menos significativo do vetor *borrow* gerado pelo *signal*.
    - SUB1: Utiliza o segundo bit menos significativo das entradas *A* e *B* para realizar a segunda operação de subtração. Para o borrow in, foi atribuído o valor do borrow out da subtração anterior, neste caso, da primeira subtração. O resultado da subtração foi atribuído ao segundo bit menos significativo do vetor *DIFF*, e o borrow out da soma foi atribuído ao segundo bit menos significativo do vetor *borrow* gerado pelo *signal*.
    - SUB2: Utiliza o segundo bit mais significativo das entradas *A* e *B* para realizar a terceira operação de subtração. Para o borrow in, foi atribuído o valor do borrow out da subtração anterior, neste caso, da segunda subtração. O resultado da subtração foi atribuído ao segundo bit mais significativo do vetor

DIFF, e o borrow out da subtração foi atribuído ao segundo bit mais significativo do vetor *borrow* gerado pelo *signal*.

- SUB3: Utiliza o bit mais significativo das entradas *A* e *B* para realizar a quarta e última operação de subtração. Para o borrow in, foi atribuído o valor do borrow out da subtração anterior, neste caso, da terceira subtração. O resultado da subtração foi atribuído ao bit mais significativo do vetor *DIFF*, e o borrow out da soma foi atribuído ao bit mais significativo do vetor *borrow* gerado pelo *signal*.

- Observações:
  - Pela forma na qual a estrutura e o encadeamento do código foi feita, o resultado do subtrator de 4 bits já está em complemento de 2.
  - Não foi implementado o bit de sinal para quando a resposta for negativa (*A* < *B*), pois haveria casos no qual 5 bits seriam necessários para a representação, e o projeto foi criado apenas para até 4 bits.

```
architecture Behavioral of my_FOURBITSUBTRACTOR is

    component my_SUBTRACTOR1BIT is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           B_in : in  STD_LOGIC;
           Difference : out  STD_LOGIC;
           B_out : out  STD_LOGIC);
   end component my_SUBTRACTOR1BIT;

    signal borrow : STD_LOGIC_VECTOR (3 downto 0);

begin
        SUB0: my_SUBTRACTOR1BIT port map (A(0), B(0), '0', DIFF(0), borrow(0));
        SUB1: my_SUBTRACTOR1BIT port map (A(1), B(1), borrow(0), DIFF(1), borrow(1));
        SUB2: my_SUBTRACTOR1BIT port map (A(2), B(2), borrow(1), DIFF(2), borrow(2));
        SUB3: my_SUBTRACTOR1BIT port map (A(3), B(3), borrow(2), DIFF(3), borrow(3));

end Behavioral;
```

Figura (32). *Signal borrow*, componente *my_SUBTRACTOR1BIT* e lógica das instâncias da entidade *my_FOURBITSUBTRACTOR*

2.11 Multiplicador

A lógica da operação de multiplicação pode ser generalizada para multiplicação de dois vetores de n bits cada da seguinte forma, sendo A e B os fatores da multiplicação, respectivamente: O bit B_n multiplica o vetor A bit a bit e é adicionado à uma soma, sofrendo deslocamento de n bits para esquerda. O resultado da multiplicação é o resultado dessa soma, que tem n termos deslocados. Construindo a tabela verdade da multiplicação vemos que é semelhante à da operação AND. O circuito que implementa tal lógica é mostrado na Figura (32).

Na lógica de implementação o circuito foi dividido da seguinte maneira:

1. 4 blocos de AND entre dois vetores de 4 bits, sendo o primeiro sempre o vetor A, e o segundo um vetor onde todos os bits são iguais, e são B_n , $n \in [0, 1, 2, 3]$, no bloco n em questão.
2. 3 módulos de somador de 4 bits, que geram 3 carry out

A entidade *my_FOUTBITMULITPLIER* tem como entradas A, B, do tipo STD_LOGIC_VECTOR, vetores de 4 bits. A saída Y, representa os 4 últimos bits do resultado da multiplicação entre A e B, por limitação dos leds de saída.

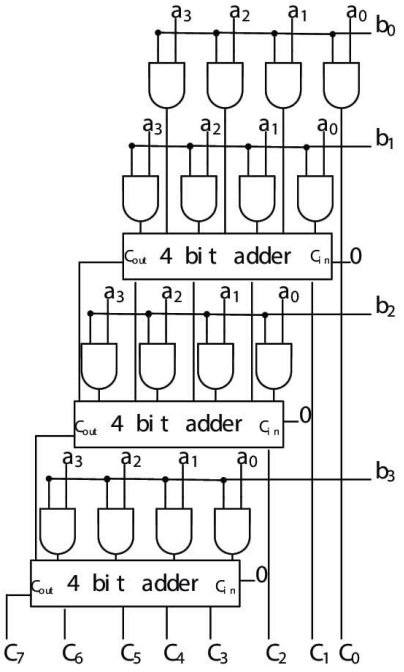

Figura (33). *Circuito que implementa a multiplicação de dois vetores de 4 bits.*

```
entity my_FOURBITMULTIPLIER is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Y : out  STD_LOGIC_VECTOR (3 downto 0));
end my_FOURBITMULTIPLIER;
```

Figura (34). Entradas e saídas da entidade *my_FOURBITMULTIPLIER*

Além disso, foi implementada uma arquitetura denominada *Behavioral* para a entidade *my_FOUTBITMULITPLIER*. Para implementar o circuito usamos o módulo do somador de 4 bits, já explicado anteriormente. Os *signals* criados representam as saídas de cada bloco. Em especial, o *signal CARRY* é um vetor por conveniência. Os *signals input_adder_2* e *input_adder_3*, que representam as entradas do segundo e do terceiro somador, respectivamente, precisaram ser criados por causa do deslocamento da operação.

As operações de *AND* foram realizadas bit a bit e atribuídas às suas respectivas posições em um dos vetores AND_n, bem como as somas, com seus carries. Para os blocos de somadores completos, usamos a funcionalidade *port map* para usar o componente

```
architecture Behavioral of my_FOURBITMULTIPLIER is

    component my_FOURBITSUM is
    Port ( A, B : in STD_LOGIC_VECTOR (3 downto 0);
           C_out : out STD_LOGIC;
           sum : out STD_LOGIC_VECTOR (3 downto 0));
    end component my_FOURBITSUM;

    signal AND_1: STD_LOGIC_VECTOR (3 downto 0);
    signal AND_2: STD_LOGIC_VECTOR (3 downto 0);
    signal AND_3: STD_LOGIC_VECTOR (3 downto 0);
    signal AND_4: STD_LOGIC_VECTOR (3 downto 0);

    signal CARRY : STD_LOGIC_VECTOR (2 downto 0);

    signal SOMA_1: STD_LOGIC_VECTOR (3 downto 0);
    signal SOMA_2: STD_LOGIC_VECTOR (3 downto 0);
    signal SOMA_3: STD_LOGIC_VECTOR (3 downto 0);

    signal input_adder_2: STD_LOGIC_VECTOR (3 downto 0);
    signal input_adder_3: STD_LOGIC_VECTOR (3 downto 0);
```

Figura (35). *Component* e *signals* da entidade *my_FOUTBITMULTIPLIER*

```
begin

    AND_1(0) <= A(1) AND B(0);
    AND_1(1) <= A(2) AND B(0);
    AND_1(2) <= A(3) AND B(0);
    AND_1(3) <= '0';

    AND_2(0) <= A(0) AND B(1);
    AND_2(1) <= A(1) AND B(1);
    AND_2(2) <= A(2) AND B(1);
    AND_2(3) <= A(3) AND B(1);

    AND_3(0) <= A(0) AND B(2);
    AND_3(1) <= A(1) AND B(2);
    AND_3(2) <= A(2) AND B(2);
    AND_3(3) <= A(3) AND B(2);
```

```
AND_4(0) <= A(0) AND B(3);
AND_4(1) <= A(1) AND B(3);
AND_4(2) <= A(2) AND B(3);
AND_4(3) <= A(3) AND B(3);

input_adder_2(0) <= SOMA_1(1);
input_adder_2(1) <= SOMA_1(2);
input_adder_2(2) <= SOMA_1(3);
input_adder_2(3) <= CARRY(0);

input_adder_3(0) <= SOMA_2(1);
input_adder_3(1) <= SOMA_2(2);
input_adder_3(2) <= SOMA_2(3);
input_adder_3(3) <= CARRY(1);
```

Figuras (36) e (37). Lógica combinacional da entidade *my_FOURBITMULTIPLIER*

```
soma_aux1: my_FOURBITSUM port map (AND_1, AND_2, CARRY(0), SOMA_1);
soma_aux2: my_FOURBITSUM port map (input_adder_2, AND_3, CARRY(1), SOMA_2);
soma_aux3: my_FOURBITSUM port map (input_adder_3, AND_4, CARRY(2), SOMA_3);

Y(0) <= A(0) AND B(0);
Y(1) <= SOMA_1(0);
Y(2) <= SOMA_2(0);
Y(3) <= SOMA_3(0);
```

Figura (38). *Port maps* da entidade *my_FOURBITMULTIPLIER*

3. Resultados

Esse tópico apresenta os resultados das simulações feitas por um VHDL Testbench. O Testbench foi feito a partir do módulo do MUX da ULA, com os valores “0110” e “0011” atribuídos aos vetores de entrada A e B, respectivamente.

Pode-se visualizar nas imagens apresentadas abaixo a existência de três vetores, a[3:0], b[3:0] e s[3:0]. Os vetores a[3:0] e b[3:0] representam as entradas A e B das operações, enquanto o vetor s[3:0] representa a seleção do MUX. O vetor y[3:0] retrata o resultado da operação em vigência.

3.1 AND

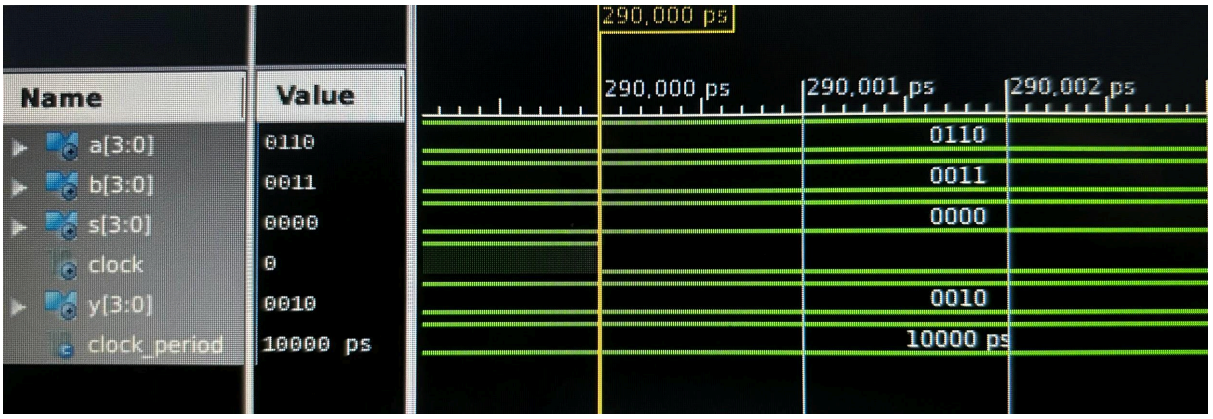

Figura (39). TESTBENCH do AND

Seleção = “0000”

Operação = A AND B = “0110” AND “0011” = “0010”

O resultado é o esperado.

3.2 NAND

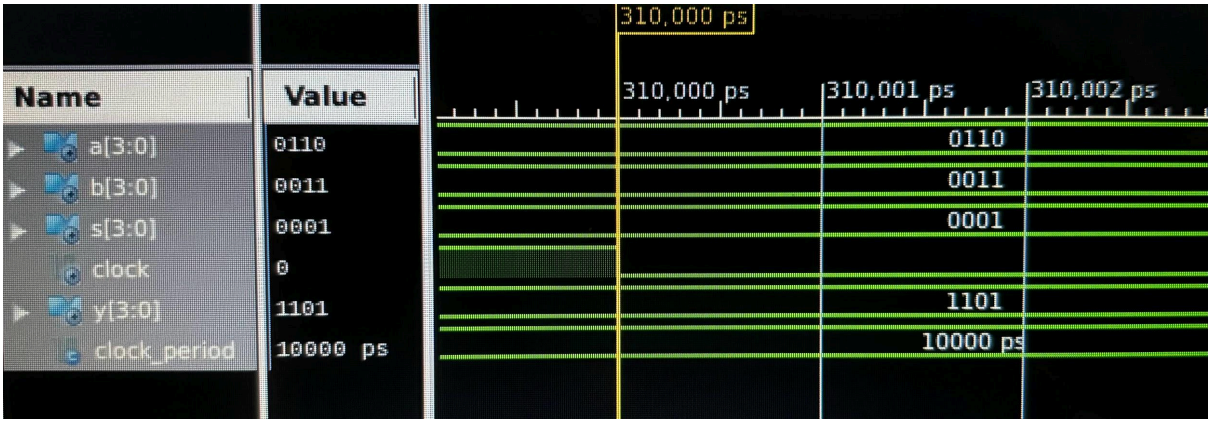

Figura (40). TESTBENCH do NAND

Seleção = “0001”

Operação = A NAND B = “0110” NAND “0011” = “1101”

O resultado é o esperado.

3.3 OR

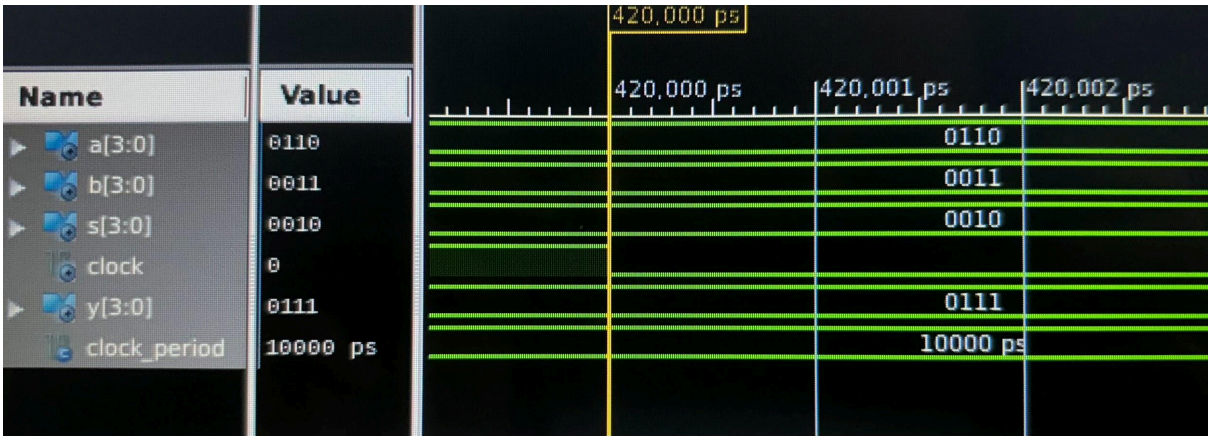

Figura (41). TESTBENCH do OR

Seleção = “0010”

Operação = A OR B = “0110” OR “0011” = “0111”

O resultado é o esperado.

3.4 NOR

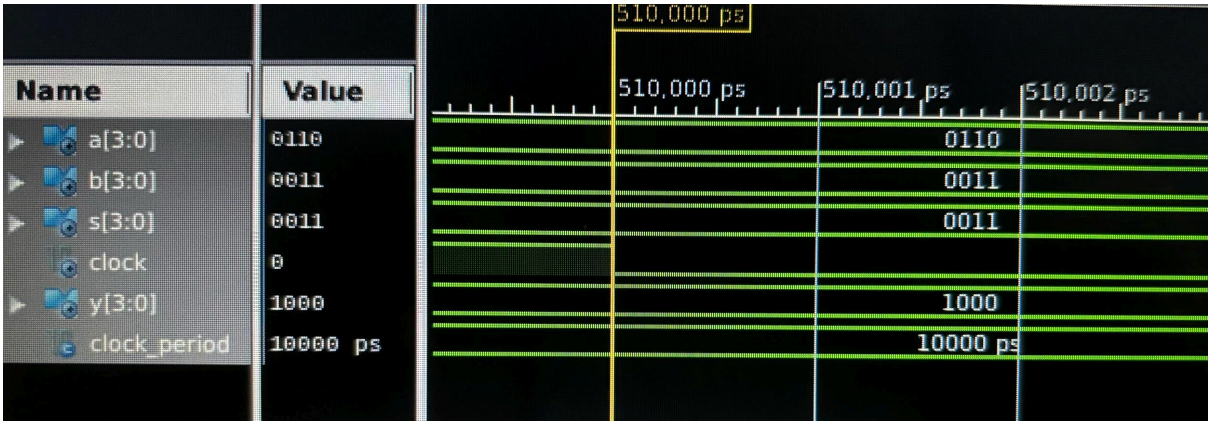

Figura (42). TESTBENCH do NOR

Seleção = “0011”

Operação = A NOR B = “0110” NOR “0011” = “1000”

O resultado é o esperado.

3.5 NOT

Figura (43). TESTBENCH do NOT

Seleção = “0100”

Operação = NOT(A) = NOT “0110” = “1001”

O resultado é o esperado.

O valor de B não é utilizado.

3.6 Somador

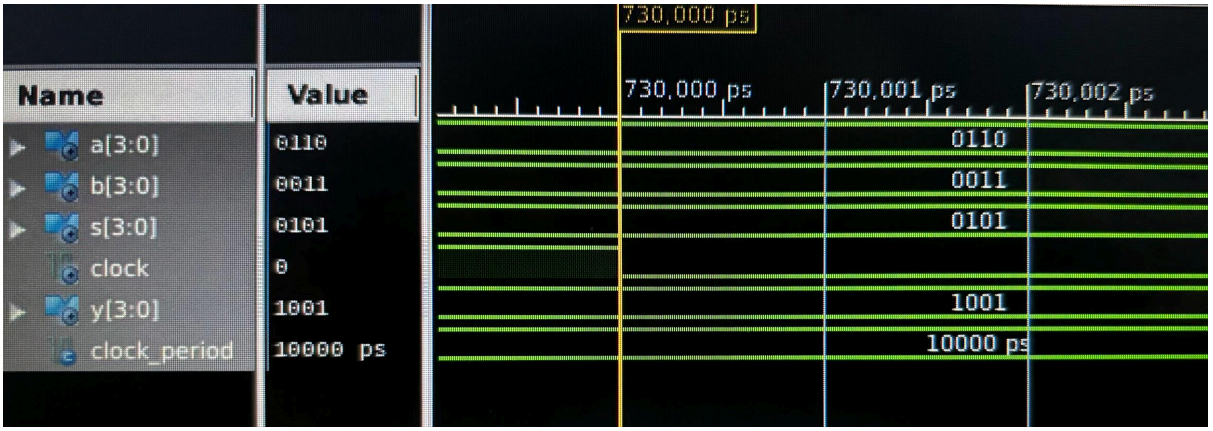

Figura (44). TESTBENCH do Somador

Seleção = “0101”

Operação = A + B = “0110” + “0011” = “1001”

O resultado é o esperado.

3.7 Subtrator

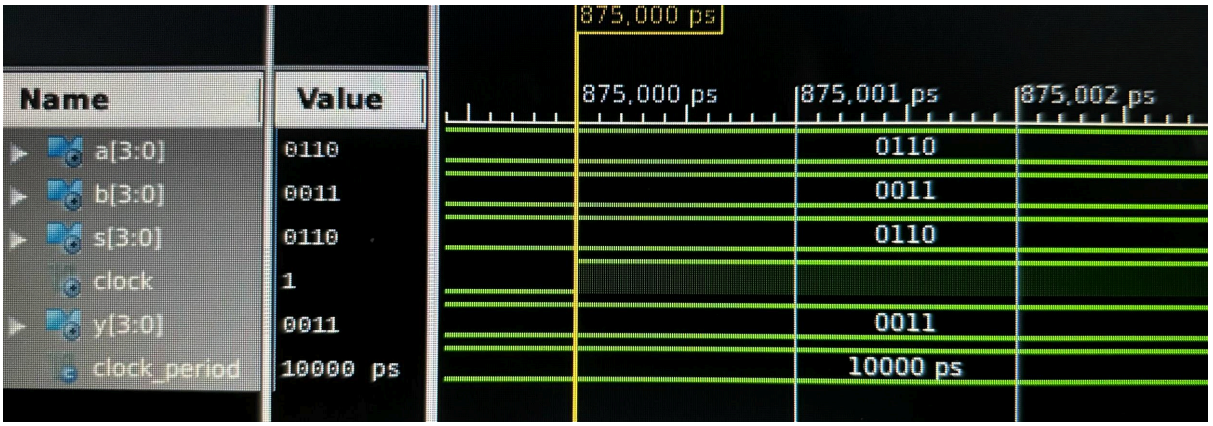

Figura (45). TESTBENCH do Subtrator

Seleção = “0110”

Operação = A - B = “0110” - “0011” = “0011”

O resultado é o esperado.

3.8 Multiplicador

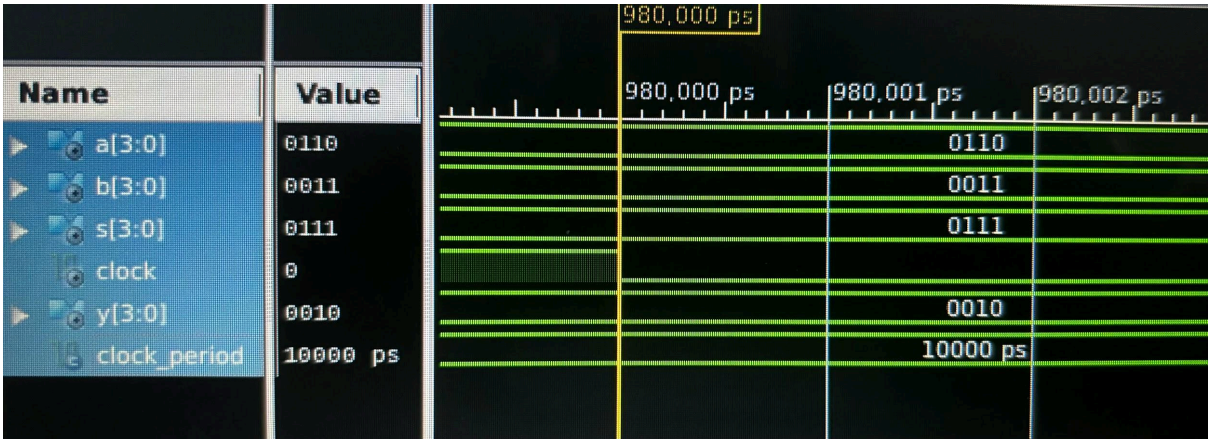

Figura (46). TESTBENCH do Multiplicador

Seleção = “0111”

Operação = A × B = “0110” × “0011” = “0010”

O resultado é o esperado.

4. Conclusão

O código desenvolvido foi implementado de forma bem sucedida tanto na placa Xilinx Spartan, quanto em simulação computacional no software Xilinx ISE. O uso de uma linguagem de descrição de hardware, o VHDL, para o desenvolvimento dos componentes da ULA conferiu maior simplicidade ao processo. Além disso, a utilização do software Xilinx ISE para a sintetização do código e, consequentemente, do circuito, ofereceu uma forma rápida e eficiente de visualização e correção dos erros na geração do circuito do projeto.