Relatório

Patrick F. Braz*, Gabriel Gazola Milan†
Departamento de Engenharia Eletrônica e da Computação
Escola Politécnica
Universidade Federal do Rio de Janeiro
Rio de Janeiro, RJ
Email:patrickfbraz@poli.ufrj.br*, gabriel.gazola@poli.ufrj.br†
DRE:116105403*, 116034377†

***Resumo*—O objetivo deste presente relatório é apresentar as etapas de desenvolvimento de uma unidade lógica e aritmética para a disciplina de sistemas digitais.**

I. Introdução

A unidade lógica e aritmética (ULA) é um dispositivo digital capaz de realizar operações lógicas e aritméticas. É um circuito de importância fundamental em unidades centrais de processamento (UCP), até dos mais simples microprocessadores. Consiste numa grande ”calculadora eletrônica”a qual sua tecnologia já estava disponível antes mesmo dos primeiros computadores.
A tecnologia utilizada foi inicialmente relés e posteriormente válvulas. Com o aparecimento dos transistores, e depois dos circuitos integrados, os circuitos da unidade aritmética e lógica passaram a ser implementados com a tecnologia de semicondutores.
Muitas das ações dos computadores são executadas pela ULA. Esta recebe dados dos registradores, que são processados e os resultados da operação são armazenados nos registradores de saída. Outros mecanismos movem os dados entre esses registradores e a memória. Uma unidade de controle controla a ULA através de circuitos que dizem que operações a ULA deve realizar. As entradas para a ULA são os dados a serem operados e o código da unidade de controle indicando as operações para executar. As saídas são os resultados da computação.
Neste trabalho foi utilizado uma FPGA e a linguagem de descrição de hardwere (VHDL) para implementar uma unidade lógica e aritmética. A FPGA é um chip que suporta a implementação de circuitos lógicos relativamente grandes. Consiste de um grande arranjo de células lógicas ou blocos lógicos configuráveis contidos em um único circuito integrado. Cada célula contém capacidade computacional para implementar funções lógicas e realizar roteamento para comunicação entre elas. A primeiro FPGA disponível comercialmente foi desenvolvido pela empresa Xilinx Inc, em 1983.

A. Especificações do trabalho

Implementar e testar uma unidade lógica e aritmética capaz de realizar as seguintes operações:

- A ULA deve realizar 8 operações incluindo soma, subtração e multiplicação, além de operações lógicas básicas como AND, OR e XOR.
- Deve conter um sistema de interface de testes.
- Os valores de entrada devem ser inseridos pelas chaves da placa FPGA.

II. Implementação dos blocos mais básicos

Para implementar a ULA, a equipe decidiu separar o projeto em blocos básicos e depois implementar blocos mais complexos e um controlador capaz de ordenar os elementos para realizar a operação pedida. Os blocos de circuitos mais básicos são:

A. Somador de 1 bit

Para a futura implementação do somador, foi escolhido utilizar o projeto de circuitos expansíveis. Para tal, foi implementado o somador completo de 1 bit para depois organizar os blocos em cascata com a finalidade de somar entradas com maior número de bits.

```
32  entity ADDER_1BIT is
33      Port ( I0 : in  STD_LOGIC;
34             I1 : in  STD_LOGIC;
35             Cin : in  STD_LOGIC;
36             S : out  STD_LOGIC;
37             Cout : out  STD_LOGIC);
38  end ADDER_1BIT;
39
40  architecture Behavioral of ADDER_1BIT is
41
42  begin
43
44         S <= (I0 xor I1) xor Cin;
45         Cout <= ((I0 and I1) or (I0 and Cin)) or (I1 and Cin);
46
47  end Behavioral;
```

Figura 1. Código VHDL do somador completo.

B. Comparador de 1 bit

Da mesma forma que o somador, para a implementação futura do comparador foi utilizada o projeto de circuitos expansíveis. A seguir será apresentado a implementação do comparador de 1 bit.

```
entity COMPARADOR_1BIT is
    Port ( Gin : in  STD_LOGIC;
           Ein : in  STD_LOGIC;
           Sin : in  STD_LOGIC;
           A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           Gout : out  STD_LOGIC;
           Eout : out  STD_LOGIC;
           Sout : out  STD_LOGIC);
end COMPARADOR_1BIT;

architecture Behavioral of COMPARADOR_1BIT is

signal equal : STD_LOGIC;

begin

        equal <= A xnor B;

        process (equal, Gin, Ein, Sin, A, B)
        begin
                if (equal = '1') then
                                Gout <= Gin;
                                Eout <= Ein;
                                Sout <= Sin;
                else
                        Gout <= A and (Not B);
                        Eout <= '0';
                        Sout <= B and (Not A);
                end if;
        end process;

end Behavioral;
```

Figura 2. Código VHDL do comparador.

C. Complementador

O complementador será um bloco essencial na composição do subtrator. Dado a entrada de controle referente a subtração, a saída do complementador será encaminhada para a entrada do somador. Dessa maneira, o bloco somador irá efetuar a subtração.

```
entity COMPLEMENTADOR_4BITS is
    Port ( I : in  STD_LOGIC_VECTOR(3 downto 0) ;
           K : in  STD_LOGIC;
           Z : out  STD_LOGIC_VECTOR (3 downto 0));
end COMPLEMENTADOR_4BITS;

architecture Behavioral of COMPLEMENTADOR_4BITS is

begin

compl4: process (I, K)
begin
       if (K = '1') then
                Z <= not I;
       else
                Z <= I;
       end if;
end process compl4;

end Behavioral;
```

Figura 3. Código VHDL do complementador.

III. Implementação das operações da ULA

A ULA projetada será capaz de efetuar 8 operações:

- Complemento a 2 da primeira entrada
- Soma
- Multiplicação
- Subtração
- AND bit a bit
- OR bit a bit
- XOR bit a bit
- Comparação entre as entradas

A seguir serão mostradas as implementações das operações da ULA.

A. AND

Código que implementa a função AND bit-a-bit.

```
entity AND_4BIT is
    Port ( x: in STD_LOGIC_VECTOR (3 downto 0);
                        y: in STD_LOGIC_VECTOR (3 downto 0);
                        z: out STD_LOGIC_VECTOR (3 downto 0));
end AND_4BIT;

architecture Behavioral of AND_4BIT is

begin
  --> Pega cada entrada e tira os and's individuais
       process (x,y)
                begin
                  --z <= x and y;
                  z(0) <= x(0) and y(0);
                  z(1) <= x(1) and y(1);
                  z(2) <= x(2) and y(2);
                  z(3) <= x(3) and y(3);
                end process;
end Behavioral;
```

Figura 4. Código VHDL do AND.

B. OR

Código que implementa a função OR bit-a-bit.

```
entity OR_4BITS 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 OR_4BITS;

architecture Behavioral of OR_4BITS 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;
```

Figura 5. Código VHDL do OR.

C. XOR

Código que implementa a função XOR bit-a-bit.

```
24   entity XOR_4BITS is
25       Port ( A : in  STD_LOGIC_VECTOR(3 downto 0);
26              --B : in  STD_LOGIC_VECTOR (3 downto 0);
27              Z : out  STD_LOGIC_VECTOR(3 downto 0));
28   end XOR_4BITS;
29
30   architecture Behavioral of XOR_4BITS is
31
32   signal B : STD_LOGIC_VECTOR (3 downto 0);
33
34   begin
35
36         B <= "1010";
37
38         Z <= A xor B;
39
40   end Behavioral;
```

Figura 6. Código VHDL do XOR.

D. Subtrator

```
architecture Behavioral of SUBTRATOR is

component COMPLEMENTADOR_4BITS is
    Port ( I : in  STD_LOGIC_VECTOR(3 downto 0) ;
           K : in  STD_LOGIC;
           Z : out  STD_LOGIC_VECTOR (3 downto 0));
end component COMPLEMENTADOR_4BITS;
component FULL_ADDER_4BITS is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Cin : in  STD_LOGIC;
           Cout : out  STD_LOGIC;
           Cout2 : out  STD_LOGIC;
           Z : out  STD_LOGIC_VECTOR (3 downto 0));
end component FULL_ADDER_4BITS;

signal Bcomplementado: STD_LOGIC_VECTOR (3 downto 0);
signal cout, cout2: STD_LOGIC;

begin
Uc: COMPLEMENTADOR_4BITS port map(EntradaB, '1', Bcomplementado);
Uf: FULL_ADDER_4BITS port map(EntradaA,Bcomplementado, '1', cout, cout2, ResultadoSUBTRACAO);
end be behavioral;
```

Figura 7. Código VHDL do Subtrator.

E. Comparador

```
32   entity COMPARADOR_4BITS is
33       Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
34              B : in  STD_LOGIC_VECTOR (3 downto 0);
35              Gout : out  STD_LOGIC;
36              Eout : out  STD_LOGIC;
37              Sout : out  STD_LOGIC);
38   end COMPARADOR_4BITS;
39
40   architecture Behavioral of COMPARADOR_4BITS is
41
42   -- Incluindo comparador 1 bit
43   component COMPARADOR_1BIT
44         port (Gin, Ein, Sin, A, B: in STD_LOGIC;
45                     Gout, Eout, Sout: out STD_LOGIC);
46   end component;
47
48   -- Sinais dos comparadores intermediarios
49   signal outC1, outC2, outC3 : STD_LOGIC_VECTOR (2 downto 0);
50
51   begin
52
53         -- Comparando o LSB
54         U1: COMPARADOR_1BIT port map ('0', '1', '0', A(0), B(0), outC1(0), outC1(1), outC1(2));
55         U2: COMPARADOR_1BIT port map (outC1(0), outC1(1), outC1(2), A(1), B(1), outC2(0), outC2(1), outC2(2));
56         U3: COMPARADOR_1BIT port map (outC2(0), outC2(1), outC2(2), A(2), B(2), outC3(0), outC3(1), outC3(2));
57         U4: COMPARADOR_1BIT port map (outC3(0), outC3(1), outC3(2), A(3), B(3), Gout, Eout, Sout);
58
59   end Behavioral;
```

Figura 8. Código VHDL do Comparador.

F. Multiplicador

```vhdl
entity MULTIPLICADOR_4BITS 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 MULTIPLICADOR_4BITS;

architecture Behavioral of MULTIPLICADOR_4BITS is

-- 1-bit adder component
component ADDER_1BIT
      port (I0, I1, Cin : in STD_LOGIC;
                  S, Cout: out STD_LOGIC);
end component;

signal tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7, tmp8, tmp9 : STD_LOGIC;
signal CoutZ1, CoutZ2, Cout2, Cout4, Cout5 : STD_LOGIC;
signal sum2, sum4, sum5 : STD_LOGIC;

begin

      Z(0) <= A(0) and B(0);
      tmp1 <= A(1) and B(0);
      tmp2 <= A(0) and B(1);
      tmp3 <= A(2) and B(0);
      tmp4 <= A(1) and B(1);
      tmp5 <= A(0) and B(2);
      tmp6 <= A(3) and B(0);
      tmp7 <= A(2) and B(1);
      tmp8 <= A(1) and B(2);
      tmp9 <= A(0) and B(3);
      U1: ADDER_1BIT port map (tmp1, tmp2, '0', Z(1), CoutZ1);
      U2: ADDER_1BIT port map (tmp3, tmp4, CoutZ1, sum2, Cout2);
      U3: ADDER_1BIT port map (sum2, tmp5, Cout2, Z(2), CoutZ2);
      U4: ADDER_1BIT port map (tmp6, tmp7, CoutZ2, sum4, Cout4);
      U5: ADDER_1BIT port map (sum4, tmp8, Cout4, sum5, Cout5);
      U6: ADDER_1BIT port map (sum5, tmp9, Cout5, Z(3));

end Behavioral;
```

Figura 9. Código VHDL do Multiplicador.

G. Somador

```vhdl
entity FULL_ADDER_4BITS is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Cin : in  STD_LOGIC;
           S : out  STD_LOGIC_VECTOR (3 downto 0);
           Cout : out  STD_LOGIC);
end FULL_ADDER_4BITS;

architecture Behavioral of FULL_ADDER_4BITS is

-- 1-bit adder component
component ADDER_1BIT
      port (I0, I1, Cin : in STD_LOGIC;
                  S, Cout: out STD_LOGIC);
end component;

-- Signals
signal Cout0, Cout1, Cout2 : STD_LOGIC;

begin

      U0: ADDER_1BIT port map (A(0), B(0), Cin, S(0), Cout0);
      U1: ADDER_1BIT port map (A(1), B(1), Cout0, S(1), Cout1);
      U2: ADDER_1BIT port map (A(2), B(2), Cout1, S(2), Cout2);
      U3: ADDER_1BIT port map (A(3), B(3), Cout2, S(3), Cout);

end Behavioral;
```

Figura 10. Código VHDL do Somador.

IV. Implementação da interface com o usuário

A seguir será apresentada a implementação da interface com o usuário. Como o código é grande para ser apresentado em uma única imagem, a implementação será mostrada em quatro partes:

```vhdl
entity INTERFACE_USUARIO is
      port (
            clock, reset, buttonA, buttonB, buttonOp : in  STD_LOGIC;
            input : in  STD_LOGIC_VECTOR (3 downto 0);
            ledA, ledB, ledOp : out STD_LOGIC; -- Vou usar esse ledOp pa
            output : out STD_LOGIC_VECTOR (3 downto 0)
      );
end INTERFACE_USUARIO;

-- Arquitetura
architecture Behavioral of INTERFACE_USUARIO is

-- Componente do contador
component CONTADOR
      port (
            load : in  STD_LOGIC;
            clock: in STD_LOGIC;
            reset: in STD_LOGIC;
            data: in INTEGER RANGE 300000000 DOWNTO 0;
            output: out INTEGER RANGE 300000000 DOWNTO 0
      );
end component;

-- Componente da ULA
component ULA
      port (
            A : in  STD_LOGIC_VECTOR (3 downto 0);
            B : in  STD_LOGIC_VECTOR (3 downto 0);
            Operation : in  STD_LOGIC_VECTOR (3 downto 0);
            Z : out  STD_LOGIC_VECTOR (3 downto 0)
      );
end component;
```

Figura 11. Primeira parte do código da interface.

```vhdl
type stateType is (stateInput, stateOutput);

-- Signals
signal state : stateType;
signal inputsDone : STD_LOGIC_VECTOR (2 downto 0); -- 2 = LedA, 1 = Led
signal inputA, inputB, inputOp : STD_LOGIC_VECTOR (3 downto 0);
signal counterOutput : INTEGER RANGE 300000000 DOWNTO 0;
signal outputULA : STD_LOGIC_VECTOR (3 downto 0);

begin

      -- Chamando os componentes da ULA e do contador
      U0: ULA port map (inputA, inputB, inputOp, outputULA);
      U1: CONTADOR port map ('0', clock, reset, 0, counterOutput);

      -- Implementando a máquina de estados
      FSM: process (clock, reset)
      begin

            if (reset = '1') then
                  inputA <= "0000";
                  inputB <= "0000";
                  output <= "0000";
                  ledA <= '0';
                  ledB <= '0';
                  ledOp <= '0';
                  state <= stateInput;
                  inputsDone <= "000";

            elsif (clock'event and clock = '1') then
                  case state is

                        when stateInput =>
                              output <= input;
                              if (buttonA = '1') then
```

Figura 12. Segunda parte do código da interface.

```
            if (buttonA = '1') then
                    inputA <= input;
                    ledA <= '1';
                    inputsDone (2) <= '1';
            elsif (buttonB = '1') then
                    inputB <= input;
                    ledB <= '1';
                    inputsDone (1) <= '1';
            elsif (buttonOp = '1') then
                    inputOp <= input;
                    ledOp <= '1';
                    inputsDone (0) <= '1';
            elsif (inputsDone = "111") then
                    ledA <= '0';
                    ledB <= '0';
                    ledOp <= '0';
                    state <= stateOutput;
            else
                    state <= stateInput;
            end if;

        when stateOutput =>
            if (counterOutput < 100000000) then
                    ledA <= '1';
                    ledB <= '0';
                    ledOp <= '0';
                    output <= inputA;
            elsif ((counterOutput >= 100000000) and (counterOutput < 200000000)) then
                    ledA <= '0';
                    ledB <= '1';
                    ledOp <= '0';
                    output <= inputB;
            elsif (counterOutput >= 200000000) then
```

Figura 13. Terceira parte do código da interface.

```
124                                     elsif (counterOutput >= 200000000) then
125                                             ledA <= '0';
126                                             ledB <= '0';
127                                             ledOp <= '1';
128                                             output <= outputULA;
129                                     end if;
130                                     state <= stateOutput;
131
132                         end case;
133                 end if;
134         end process;
135
136 end Behavioral;
137
```

Figura 14. Quarta parte do código da interface.

Caso a visualização do código seja prejudicada, é possível acessá-lo pelo link do GitHub disponível no apêndice.

V. Componentes da interface

A. *Componente: ULA*

As imagens que serão mostradas a seguir não possuem a parte da implementação onde são citados todos os componentes necessários para a ULA. Como já foram mostrados as implementações de cada componenete de operação lógica e aritmética, parte do código foi retirado para que apenas o funcionamento da entidade fosse mostrado. Este componente coleta as entradas fornecidas à interface pelo usuário e efetua a operação escolhida, retornando para a interface o resultado da operação.

```
86  -- Comportamento da ULA
87  begin
88
89          -- Declarando os componentes
90          U1: COMPLEMENTADOR_4BITS port map (A, '1', Not_A); -- Para operação 1
91          U2: FULL_ADDER_4BITS port map (Not_A, "0000", '1', Compl_A); -- Para operação 1
92          U3: FULL_ADDER_4BITS port map (A, B, '0', A_plus_B); -- Para operação 2
93          U4: MULTIPLICADOR_4BITS port map (A, B, A_times_B); -- Para operação 3
94          U5: COMPLEMENTADOR_4BITS port map (B, '1', Not_B); -- Para operação 4
95          U6: FULL_ADDER_4BITS port map (A, Not_B, '1', A_minus_B); -- Para operação 4
96          U7: AND_4BIT port map (A, B, A_and_B); -- Para operação 5
97          U8: XOR_4BITS port map (A, B, A_xor_B); -- Para operação 6
98          U9: OR_4BITS port map (A, B, A_or_B); -- Para operação 7
99          U10: COMPARADOR_4BITS port map (A, B, A_compare_B(2), A_compare_B(1), A_compare_B(0)); -- Para opera
100         A_compare_B(3) <= '0'; -- Para operação 8
101
102         -- Dando as saídas baseadas na escolha da operação
103         process (Operation, Compl_A, A_plus_B, A_times_B, A_minus_B, A_and_B, A_xor_B, A_or_B, A_compare_B)
104         begin
105                 case Operation is
106                         when "0000" =>
107                                 Z <= Compl_A;
108                         when "1000" =>
109                                 Z <= Compl_A;
110                         when "0001" =>
111                                 Z <= A_plus_B;
112                         when "1001" =>
113                                 Z <= A_plus_B;
114                         when "0010" =>
115                                 Z <= A_times_B;
```

Figura 15. Primeira parte do código do componente ULA.

```
114                         when "0010" =>
115                                 Z <= A_times_B;
116                         when "1010" =>
117                                 Z <= A_times_B;
118                         when "0011" =>
119                                 Z <= A_minus_B;
120                         when "1011" =>
121                                 Z <= A_minus_B;
122                         when "0100" =>
123                                 Z <= A_and_B;
124                         when "1100" =>
125                                 Z <= A_and_B;
126                         when "0101" =>
127                                 Z <= A_xor_B;
128                         when "1101" =>
129                                 Z <= A_xor_B;
130                         when "0110" =>
131                                 Z <= A_or_B;
132                         when "1110" =>
133                                 Z <= A_or_B;
134                         when "0111" =>
135                                 Z <= A_compare_B;
136                         when "1111" =>
137                                 Z <= A_compare_B;
138                         when others =>
139                                 Z <= "0000";
140                 end case;
141         end process;
142
143 end Behavioral;
144
```

Figura 16. Segunda parte do código do componente ULA

Caso a visualização do código seja prejudicada, é possível acessá-lo pelo link do GitHub disponível no apêndice.

B. *Componente: Contador*

O contador esta sendo utilizado como um divisor de frequência para efetuar a contagem de dois segundos na amostragem das entradas, operação e resultados, assim como especificado no projeto.

```
24  entity CONTADOR is
25          Port (
26                  load : in  STD_LOGIC;
27                  clock: in STD_LOGIC;
28                  reset: in STD_LOGIC;
29                  data: in INTEGER RANGE 300000000 DOWNTO 0;
30                  output: out INTEGER RANGE 300000000 DOWNTO 0
31          );
32          -- Valores devido ao clock de 50MHz (6s = 300000000 clocks)
33  end CONTADOR;
34
35  architecture Behavioral of CONTADOR is
36
37  begin
38
39          count: process (clock, reset)
40          variable counting : INTEGER RANGE 300000000 DOWNTO 0;
41          begin
42                  if (reset = '1') then
43                          counting := 0;
44                  elsif (clock'event and clock = '1') then
45                          if (load = '1') then
46                                  counting := data;
47                          else
48                                  if (counting >= 300000000) then
49                                          counting := 0;
50                                  else
51                                          counting := counting + 1;
52                                  end if;
53                          end if;
54                  end if;
55                  output <= counting;
56          end process;
57
58  end Behavioral;
```

Figura 17. Implementação do contador.

VI. Resultados

Para executar as simulações foram atribuidos valores fixos para as entradas da ULA, exceto a entrada da operação. Essas entradas fixas foram denominadas A e B e os valores fixados são mostrados na figura a seguir:

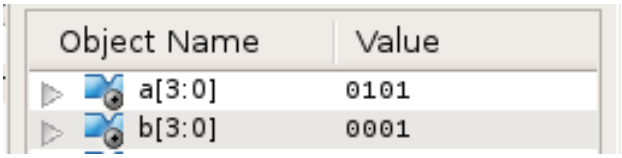

Figura 18. Valores fixados para as entradas para realizar as simulações

A. Operação 1: Complemento a 2 da entrada A

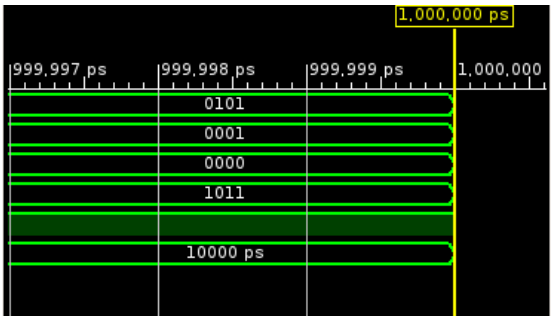

Figura 19. Resultado no tempo para o complemento a 2 da entrada A.

B. Operação 2: Soma das entradas

A saída não possui nenhuma informação sobre o 'Carry' por decisão de projeto.

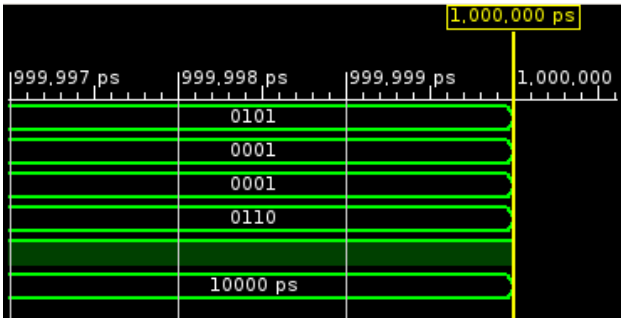

Figura 20. Resultado no tempo para a soma das entradas.

C. Operação 3: Multiplicação das entradas

Como no projeto a saída foi especificada como sendo um vetor de 4 bits, a saída será os 4 bits menos significativos oriundos da multiplicação.

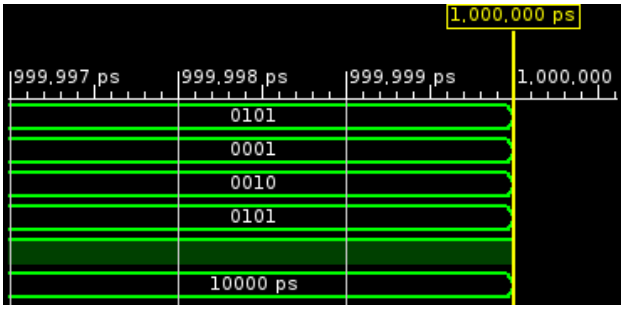

Figura 21. Resultado no tempo para a multiplicação das entradas.

D. Operação 4: Subtração das entradas

Por decisão de projeto, a saída não apresenta informações sobre o 'borrow'.

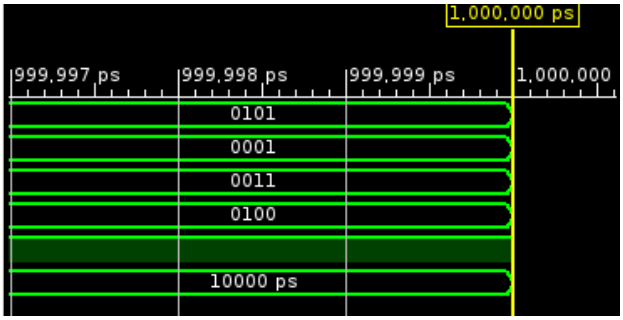

Figura 22. Resultado no tempo para a subtração das entradas.

E. Operação 5: A AND B

A operação é realizada bit-a-bit.

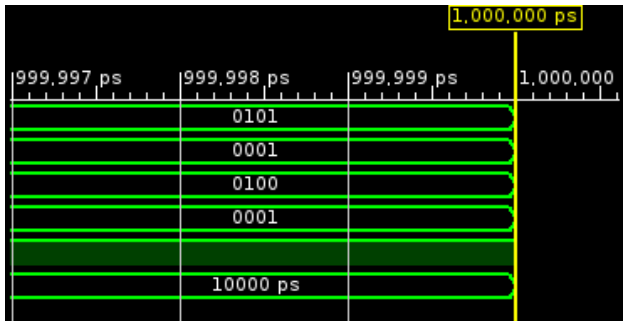

Figura 23. Resultado no tempo para a aplicação da função AND entre as entradas.

F. Operação 6: A XOR B

A operação é realizada bit-a-bit.

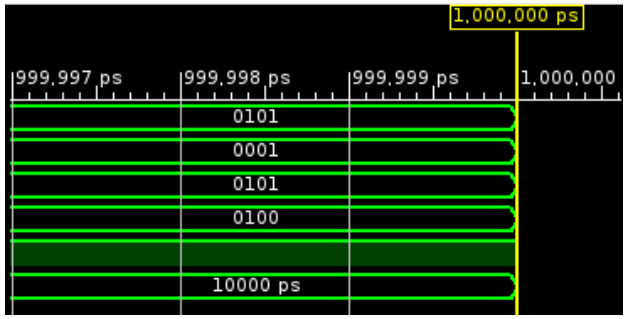

Figura 24. Resultado no tempo para a aplicação da função XOR entre as entradas.

G. Operação 7: A OR B

A operação é realizada bit-a-bit.

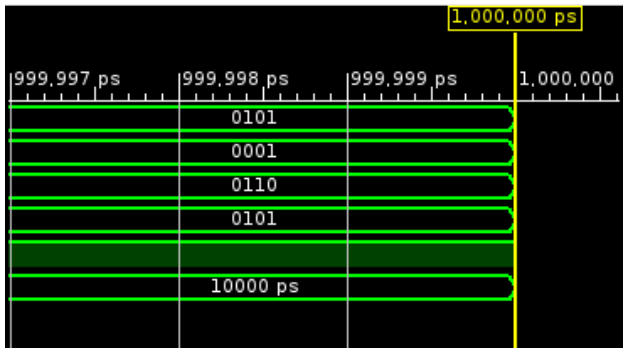

Figura 25. Resultado no tempo para a aplicação da função OR entre as entradas.

H. Operação 8: Comparação entre A e B

Os resultados dessa operação devem ser interpretados da seguinte maneira: O bit menos significativo, à direita, indica que a entrada A é menor que a entrada B. O segundo bit, à esquerda, indica que A é igual a B. O terceiro bit, à esquerda, indica que A é maior que B. O bit mais significativo, à esquerda, sempre estará em nivel baixo.

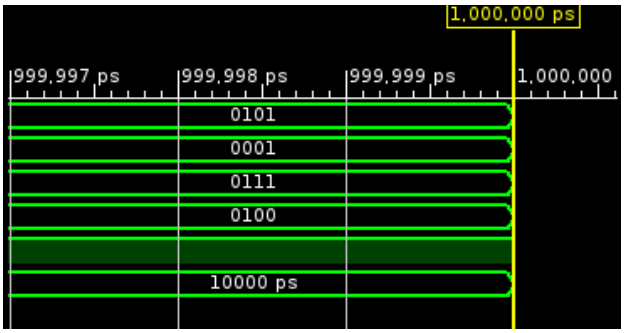

Figura 26. Resultado no tempo para a comparação das entradas.

VII. Conclusão

Ao longo do projeto a equipe foi capaz de notar a praticidade de trabalhar com dispositivos programáveis. A flexibilidade do dispositivo, com o auxílio da linguagem de descrição de hardware, possibilitou a equipe projetar desde componentes básicos como o somador, até maquinas de estado mais robustas. Desse modo, foi possível colocar em prática tanto técnicas de projeto, como conteúdo teórico da disciplina de sistemas digitais. Os resultados obtidos pela simulação e os testes no laboratório comprovaram o bom funcionamento da ULA projetada.

VIII. Apêndice

A. Link para acesso aos códigos no github

https://github.com/gabriel-milan/digitalsystems-01