Universidade Federal do Rio de Janeiro
Escola Politécnica – Departamento de Eletrônica
Sistemas Digitais 1 – 2023.1

Calculadora BCD

Alunos: Martina Marques Jardim
DRE: 121078124
Gabriel Henrique Braga Lisboa
DRE: 120095995
Professor: João Baptista Souza Filho
Turma: EL1
Horário: Sexta-feira 13:00 – 15:00

Rio de Janeiro – 21 de julho de 2023

- **Introdução:**

O presente relatório tem como objetivo apresentar o desenvolvimento de uma calculadora de soma e multiplicação em BCD em FPGA, em uma placa Xilinx Spartan 3AN. Foi utilizada a linguagem de programação VHDL, a qual é paralela e foi criada especificamente para projetar circuitos integrados.

O objetivo do projeto é permitir que o usuário insira um número de 4 casas decimais, através de um teclado, e que consiga ver os algarismos que digitou em um *display*, bem como selecionar a operação que ele deseja através de switches, e ver o resultado da operação no mesmo *display.*

Assim, percebe-se que o projeto exige a presença de uma máquina de estados, de módulos que permitam a interação com o teclado e com o display, além de módulos aritméticos.

- **Projeto:**

1- Módulo State Machine:

Este é o módulo principal do projeto, o qual contém a máquina de estados e promove a interação entre o teclado e o *display.*

```
32  entity StateMachine is
33      Port  (clk, reset: in  std_logic;
34             ps2d, ps2c: in  std_logic;
35             botao : in std_logic;
36             SW: in std_logic;
37             LCD_DB1: out std_logic_vector (7 downto 0));
38
39             ---resultadoSoma: out std_logic_vector (15 downto 0);
40             ---resultadoMult: out std_logic_vector (31 downto 0));
41  end StateMachine;
42
43  architecture Behavioral of StateMachine is
44
45  component kb_code is
46     generic(W_SIZE: integer:=1);  -- 2^W_SIZE words in FIFO
47     port (
48        clk, reset: in  std_logic;
49        ps2d, ps2c: in  std_logic;
50        rd_key_code: in std_logic;
51        number_code: out std_logic_vector(7 downto 0);
52        kb_buf_empty: out std_logic
53     );
54  end component;
```

Fig. 1.1 – Início do código da máquina de estados

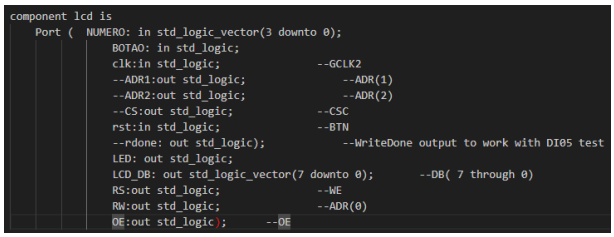


Fig. 1.2 – Componente do display

Como é possível ver no código acima, a definição do *top module* é feita com as entradas de *clock, reset, ps2d, ps2c, botao, SW, LCD_DB1.* A primeira entrada é utilizada em todo o projeto, a segunda no componente do *display, ps2d, ps2c, botao, LCD_DB1* no módulo do teclado e SW no módulo aritmético.

Os componentes *kb_code* e *lcd* se referem ao teclado e ao *display,* respectivamente.

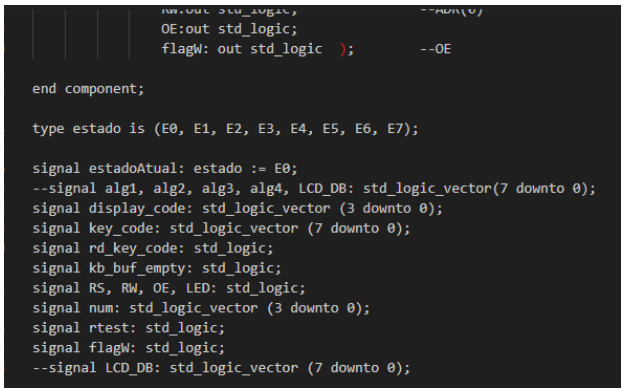


Fig. 1.3 – Implementação do tipo estado e dos sinais

Inicialmente, é definido o tipo estado. A criação desse tipo é necessária pois auxiliará na definição de cada estado e na consequente transição entre eles.

Abaixo dessa nova definição, foram definidos diversos sinais, que serão utilizados na máquina de estados e atuarão como as entradas e saídas de cada componente. A explicação de cada um será feita ao longo deste capítulo.

Após as definições, inicia-se o processo da máquina de estados.

```
begin
    teclado: kb_code port map (clk, reset, ps2d, ps2c, rd_key_code, key_code, kb_buf_empty);

    process (clk, rtest, estadoAtual, rd_key_code, key_code, display_code)

        begin

            if (rising_edge(clk)) then

                case estadoAtual is

                    when E0 =>
                        if (kb_buf_empty = '1') then
                            rd_key_code <= '0';
                            rtest <= '0';
                        else
                            rd_key_code <= '1';
                            if (key_code = "00001101") then
                                estadoAtual <= E4;
                            else
                                display_code <= key_code (3 downto 0);
                                rtest <= '1';
                                estadoAtual <= E1;
                            end if;
                        end if;

                    when E1 =>
                        if (kb_buf_empty = '1') then
                            rd_key_code <= '0';
                            rtest <= '0';
                        else
```

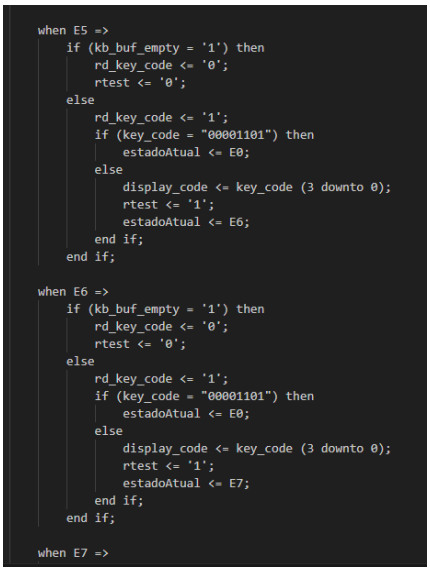
Fig. 1.4 – Início da máquina de estados

```
    when E1 =>
        if (kb_buf_empty = '1') then
            rd_key_code <= '0';
            rtest <= '0';
        else
            rd_key_code <= '1';
            if (key_code = "00001101") then
                estadoAtual <= E4;
            else
                display_code <= key_code (3 downto 0);
                rtest <= '1';
                estadoAtual <= E2;
            end if;
        end if;

    when E2 =>
        if (kb_buf_empty = '1') then
            rd_key_code <= '0';
            rtest <= '0';
        else
            rd_key_code <= '1';
            if (key_code = "00001101") then
                estadoAtual <= E4;
            else
                display_code <= key_code (3 downto 0);
                rtest <= '1';
                estadoAtual <= E3;
            end if;
        end if;

    when E3 =>
        if (kb_buf_empty = '1') then
```

Fig. 1.5 – Máquina de Estados

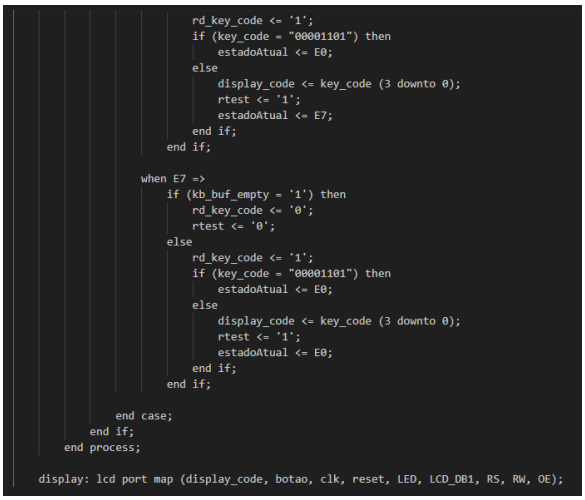
```
    when E3 =>
        if (kb_buf_empty = '1') then
            rd_key_code <= '0';
            rtest <= '0';
        else
            rd_key_code <= '1';
            if (key_code = "00001101") then
                estadoAtual <= E4;
            else
                display_code <= key_code (3 downto 0);
                rtest <= '1';
                estadoAtual <= E4;
            end if;
        end if;

    when E4 =>
        if (kb_buf_empty = '1') then
            rd_key_code <= '0';
            rtest <= '0';
        else
            rd_key_code <= '1';
            if (key_code = "00001101") then
                estadoAtual <= E0;
            else
                display_code <= key_code (3 downto 0);
                rtest <= '1';
                estadoAtual <= E5;
            end if;
        end if;
```

Fig. 1.6 – Máquina de Estados



```
when E5 =>
    if (kb_buf_empty = '1') then
        rd_key_code <= '0';
        rtest <= '0';
    else
        rd_key_code <= '1';
        if (key_code = "00001101") then
            estadoAtual <= E0;
        else
            display_code <= key_code (3 downto 0);
            rtest <= '1';
            estadoAtual <= E6;
        end if;
    end if;

when E6 =>
    if (kb_buf_empty = '1') then
        rd_key_code <= '0';
        rtest <= '0';
    else
        rd_key_code <= '1';
        if (key_code = "00001101") then
            estadoAtual <= E0;
        else
            display_code <= key_code (3 downto 0);
            rtest <= '1';
            estadoAtual <= E7;
        end if;
    end if;

when E7 =>
```

Fig. 1.7 – Máquina de Estados



```
                        rd_key_code <= '1';
                        if (key_code = "00001101") then
                            estadoAtual <= E0;
                        else
                            display_code <= key_code (3 downto 0);
                            rtest <= '1';
                            estadoAtual <= E7;
                        end if;
                    end if;

                when E7 =>
                    if (kb_buf_empty = '1') then
                        rd_key_code <= '0';
                        rtest <= '0';
                    else
                        rd_key_code <= '1';
                        if (key_code = "00001101") then
                            estadoAtual <= E0;
                        else
                            display_code <= key_code (3 downto 0);
                            rtest <= '1';
                            estadoAtual <= E0;
                        end if;
                    end if;

            end case;
        end if;
    end process;

display: lcd port map (display_code, botao, clk, reset, LED, LCD_DB1, RS, RW, OE);
```

Fig. 1.8 – Fim da Máquina de Estados

A primeira coisa a ser feita é o *port map* do teclado. É importante que essa declaração seja feita logo no início pois os algarismos utilizados nos procedimentos aritméticos serão recebidos por meio do teclado.

Conforme é visto na figura 2, o sinal estado é incialmente definido como *E0.* Este é o tipo do estado inicial da máquina. A mudança de estados está em uma estrutura *case...when,* a qual está dentro de uma condicional. A condicional verifica se o clock está em *rising_edge,* ou seja, se está indo para o valor lógico 1. Ao entrar na estrutura *case...when,* verifica-se qual é o valor armazenado no sinal *EstadoAtual.* Os possíveis valores de *EstadoAtual* referem-se aos estados da máquina.

Todos os blocos *When* verificam o sinal *kb_buf_empty,* o qual é uma saída do componente do teclado do tipo *std_logic.* Esse

sinal indica se o *buffer* está vazio ou não. Se esse sinal retornar 1, o buffer está vazio. A estrutura condicional dentro de cada *When* verifica o valor desse sinal: se for 1, ele impõe às variáveis *rd_key_code* e *rtest* o valor 0, o qual impossibilita a operação de leitura. Se for 0, significa que o buffer está preenchido e é feita uma verificação do valor que está no buffer. O buffer, nesta aplicação, é chamado de *key_code.* Se o valor que está no buffer for 00001101, a tecla que foi apertada no teclado foi o *enter,* o que indica que o usuário já terminou de digitar todos os algarismos que desejava. Assim, ele passa a *EstadoAtual* o nome do estado de transição entre os estados de *input,* sendo esses *E0* e *E4.* O motivo pelo qual a tecla *enter* é representada pelo valor 00001101 será explicitado mais a diante.

Caso o valor em *keycode* seja diferente de 00001101, esse valor será repassado ao sinal *display_code,* o qual é a entrada do *display.* Além disso, o valor do sinal *EstadoAtual* é modificado para o próximo estado.

Ao todo, são oito estados, um para cada número inserido, considerando que o usuário pode inserir dois números de quatro algarismos.

Ao fim da estrutura *Case...When,* é feita uma chamada para o componente do *display,* o qual recebe o valor que está armazenado em *display_code.*

Infelizmente, mesmo após exaustivas tentativas e diversas modificações, o modelo proposto acima não conseguiu implementar corretamente o que se propunha na placa FPGA. Foi possível compilá-lo corretamente, mas observou-se que o display e o teclado não interagiam.

2 – Módulo LCD:

Este módulo se refere ao display da placa. Nesse sentido, idealmente, ele recebe a saída do módulo do teclado.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity lcd is
    Port (  NUMERO: in std_logic_vector(3 downto 0);
            BOTAO: in std_logic;
            clk:in std_logic;               --GCLK2
            --ADR1:out std_logic;               --ADR(1)
            --ADR2:out std_logic;               --ADR(2)
            --CS:out std_logic;             --CSC
            rst:in std_logic;               --BTN
            --rdone: out std_logic);            --WriteDone output to work with DI05 test
            LED: out std_logic;
            LCD_DB: out std_logic_vector(7 downto 0);       --DB( 7 through 0)
            RS:out std_logic;               --WE
            RW:out std_logic;               --ADR(0)
            OE:out std_logic);              --OE
end lcd;
```

Fig. 2.1 – Entidade do módulo LCD

Como é possível ver, a entidade recebe três entradas: uma para o código ASCII (NUMERO), outra para o botão que irá permitir a exibição do caractere ASCII digitado, outra para *reset* (rst). Ele tem cinco saídas, das quais

LED é a responsável por acender o caractere no display. Mesmo após o estudo do código, ficou incerta qual a função das demais saídas.

No módulo, também se observou o seguinte:

```
type LCD_CMDS_T is array(23 downto 0) of std_logic_vector(9 downto 0);
signal LCD_CMDS : LCD_CMDS_T := ( 0 => "00"&X"3C",          --Function Set
                  1 => "00"&X"0C",          --Display ON, Cursor OFF, Blink OFF
                  2 => "00"&X"01",          --Clear Display
                  3 => "00"&X"02",          --return home

                  4 => "10"&X"30",          --0
                  5 => "10"&X"65",          --e
                  6 => "10"&X"6C",          --l
                  7 => "10"&X"6C",          --l
                  8 => "10"&X"6F",          --o
                  9 => "10"&X"20",          --space
                  10 => "10"&X"46",         --F
                  11 => "10"&X"72",         --r
                  12 => "10"&X"6F",         --o
                  13 => "10"&X"6D",         --m

                  14 => "10"&X"20",         --space

                  15 => "10"&X"44",         --D
                  16 => "10"&X"69",         --i
                  17 => "10"&X"67",         --g
                  18 => "10"&X"69",         --i
                  19 => "10"&X"6C",         --l
                  20 => "10"&X"65",         --e
                  21 => "10"&X"6E",         --n
                  22 => "10"&X"74",         --t
                  23 => "00"&X"02");        -- Shifts left


signal lcd_cmd_ptr : integer range 0 to LCD_CMDS'HIGH + 1 := 0;
signal TRAVA: std_logic:='1';
```

Fig 2.2 – LCD_CMDS e TRAVA

Compreende-se que os sinais LCD_CMDS representam comandos que são executados pelo Display, ou seja, se algum dos sinais desse tipo for chamado, o Display deve exibir o caractere referente a ele. Além disso, ao final da imagem, consta o sinal TRAVA, o qual é um sinal *standard logic* o qual controlará a saída de exibição do Display.

Como o Display deve ser a saída do módulo, houve a tentativa de implementá-lo como *top module* do projeto, definindo o teclado como componente e definindo a máquina de estados dentro do próprio código do LCD. Para fazer com que ele exiba o caractere digitado, tentou-se modificar sinais LCD_CMD, fazendo com que esses possam ser alterados na máquina de estados. Contudo, houve a recorrência de um erro que afirmava que o LCD_CMD estava associado a diversos *drivers.*

3 – Módulo Kb_code:

Este módulo se refere ao teclado. Ele possui quatro arquivos diferentes, nos quais estão definidos o comportamento dos componentes do teclado.

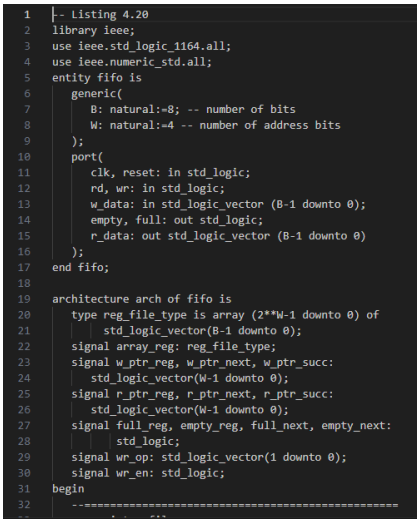
key2ascii.vhd

```
1   -- Listing 8.4
2   library ieee;
3   use ieee.std_logic_1164.all;
4   use ieee.numeric_std.all;
5
6   entity key2ascii is
7      port (
8         key_code: in std_logic_vector(7 downto 0);
9         ascii_code: out std_logic_vector(7 downto 0)
10     );
11  end key2ascii;
12
13  architecture arch of key2ascii is
14  begin
15     with key_code select
16        ascii_code <=
17
18           "00110000" when "01000101",  -- 0
19           "00110001" when "00010110",  -- 1
20           "00110010" when "00011110",  -- 2
21           "00110011" when "00100110",  -- 3
22           "00110100" when "00100101",  -- 4
23           "00110101" when "00101110",  -- 5
24           "00110110" when "00110110",  -- 6
25           "00110111" when "00111101",  -- 7
26           "00111000" when "00111110",  -- 8
27           "00111001" when "01000110",  -- 9
```

Fig. 3.1 – Key to Ascii

O código acima contém a conversão do código que vem das teclas do teclado para o equivalente na tabela Ascii.

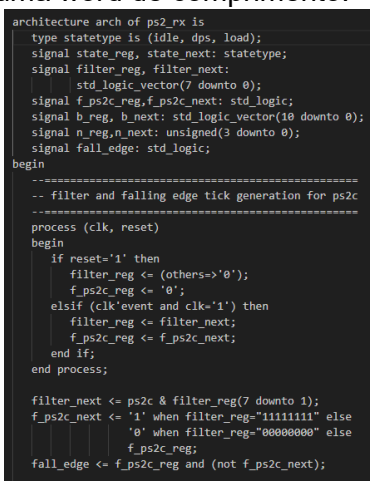
O resultado dessa conversão seria, posteriormente, convertido para BCD para ser utilizado nos módulos aritméticos.



```
-- Listing 4.20
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo is
   generic(
      B: natural:=8; -- number of bits
      W: natural:=4 -- number of address bits
   );
   port(
      clk, reset: in std_logic;
      rd, wr: in std_logic;
      w_data: in std_logic_vector (B-1 downto 0);
      empty, full: out std_logic;
      r_data: out std_logic_vector (B-1 downto 0)
   );
end fifo;

architecture arch of fifo is
   type reg_file_type is array (2**W-1 downto 0) of
        std_logic_vector(B-1 downto 0);
   signal array_reg: reg_file_type;
   signal w_ptr_reg, w_ptr_next, w_ptr_succ:
      std_logic_vector(W-1 downto 0);
   signal r_ptr_reg, r_ptr_next, r_ptr_succ:
      std_logic_vector(W-1 downto 0);
   signal full_reg, empty_reg, full_next, empty_next:
        std_logic;
   signal wr_op: std_logic_vector(1 downto 0);
   signal wr_en: std_logic;
begin
   -------------------------------------------------
```

Fig. 3.2 - FIFO

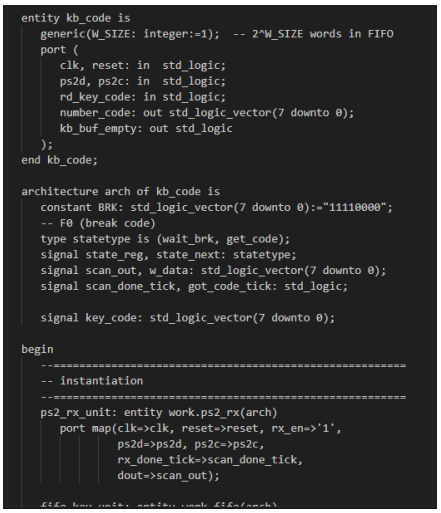
É possível ver na imagem acima que a estrutura adotada pelo teclado é a de uma FIFO, ou seja, que tudo que ela recebe é automaticamente enviado. Contudo, ela só consegue fazer isso com dados de até uma word de comprimento.



```
architecture arch of ps2_rx is
   type statetype is (idle, dps, load);
   signal state_reg, state_next: statetype;
   signal filter_reg, filter_next:
        std_logic_vector(7 downto 0);
   signal f_ps2c_reg,f_ps2c_next: std_logic;
   signal b_reg, b_next: std_logic_vector(10 downto 0);
   signal n_reg,n_next: unsigned(3 downto 0);
   signal fall_edge: std_logic;
begin
   -------------------------------------------------
   -- filter and falling edge tick generation for ps2c
   -------------------------------------------------
   process (clk, reset)
   begin
      if reset='1' then
         filter_reg <= (others=>'0');
         f_ps2c_reg <= '0';
      elsif (clk'event and clk='1') then
         filter_reg <= filter_next;
         f_ps2c_reg <= f_ps2c_next;
      end if;
   end process;

   filter_next <= ps2c & filter_reg(7 downto 1);
   f_ps2c_next <= '1' when filter_reg="11111111" else
                  '0' when filter_reg="00000000" else
                  f_ps2c_reg;
   fall_edge <= f_ps2c_reg and (not f_ps2c_next);
```

Fig 3.3 – Ps2_Rx

A imagem acima é um trecho do código de P2_Rx, que é responsável pela conexão do cabo do teclado.



```
entity kb_code is
   generic(W_SIZE: integer:=1);  -- 2^W_SIZE words in FIFO
   port (
      clk, reset: in  std_logic;
      ps2d, ps2c: in  std_logic;
      rd_key_code: in std_logic;
      number_code: out std_logic_vector(7 downto 0);
      kb_buf_empty: out std_logic
   );
end kb_code;

architecture arch of kb_code is
   constant BRK: std_logic_vector(7 downto 0):="11110000";
   -- F0 (break code)
   type statetype is (wait_brk, get_code);
   signal state_reg, state_next: statetype;
   signal scan_out, w_data: std_logic_vector(7 downto 0);
   signal scan_done_tick, got_code_tick: std_logic;

   signal key_code: std_logic_vector(7 downto 0);

begin
   --=============================================================
   -- instantiation
   --=============================================================
   ps2_rx_unit: entity work.ps2_rx(arch)
      port map(clk=>clk, reset=>reset, rx_en=>'1',
               ps2d=>ps2d, ps2c=>ps2c,
               rx_done_tick=>scan_done_tick,
               dout=>scan_out);
```

Fig. 4.2 – Teclado.vhd

A imagem acima é o código principal do componente do teclado. Nesse sentido, ele recebe as entradas e as informações dos demais módulos e as encaminha para a saída. Alguns sinais usados nesse código forma usados na máquina.

4 – Módulos Aritméticos:

4.1 – Soma2Alg:

```
33  entity Soma2Alg is
34      Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
35             B : in  STD_LOGIC_VECTOR (3 downto 0);
36               Cin: in STD_LOGIC;
37             S : out  STD_LOGIC_VECTOR (3 downto 0);
38             Cout : out  STD_LOGIC);
39  end Soma2Alg;
40
41  architecture Behavioral of Soma2Alg is
42
43  signal aux: unsigned (4 downto 0) := "00000";
44  signal resultado: unsigned (3 downto 0) := "0000";
45
46  begin
47
48      P1: process (A, B, Cin, aux, resultado)
49      begin
50          aux <= '0' & unsigned(A) + unsigned(B) + ("0000" & Cin);
51          if (aux > 9) then
52              -- se passar de 9, tem que somar 6 no resultado e o Cout vai pra 1.
53              Cout <= '1';
54              resultado <= resize((aux + "00110"), 4);
55          else
56              Cout <= '0';
57              resultado <= aux(3 downto 0);
58          end if;
59          S <= std_logic_vector (resultado);
60      end process;
61
62  end Behavioral;
63
```

Fig 4.1.1 – Código Soma2Alg

O código acima implementa a soma de dois algarismos em BCD. Para isso, ele recebe dois vetores do tipo *std_logic_vector.* Para realizar a operação, é definido um vetor auxiliar, o qual consiste na soma dos argumentos A e B (como tipo unsigned), concatenada com um zero, acrescida do vetor “0000” concatenado com o carry in no último algarismo.

Em seguida, avalia-se se o resultado na variável auxiliar é maior que 9. Se for, converte-o para BCD. Tal processo é feito adicionando-se 6 e ajustando o tamanho do vetor de novo para quatro bits. Em seguida, o número é passado para a saída. Se o

número não for maior que 9, o valor em *aux* é simplesmente passado para a saída, com seu bit mais significativo sendo retirado.

4.2 – Somador:

Somador.vhd

```
entity Somador is
    Port ( A : in  STD_LOGIC_VECTOR (15 downto 0);
           B : in  STD_LOGIC_VECTOR (15 downto 0);
           soma : out  STD_LOGIC_VECTOR (15 downto 0);
           Cout : out  STD_LOGIC);
end Somador;

architecture Behavioral of Somador is

component Soma2Alg 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 component;


signal c: std_logic_vector (2 downto 0);
signal vetor1, vetor2: std_logic_vector (15 downto 0);

begin

    somador1: Soma2Alg port map (A(3 downto 0), B(3 downto 0), '0', soma(3 downto 0), c(0));
    somador2: Soma2Alg port map (A(7 downto 4), B(7 downto 4), c(0), soma(7 downto 4), c(1));
    somador3: Soma2Alg port map (A(11 downto 8), B(11 downto 8), c(1), soma(11 downto 8), c(2));
    somador4: Soma2Alg port map (A(15 downto 12), B(15 downto 12), c(2), soma(15 downto 12), Cout);

end Behavioral;

```

Fig. 4.2.1 – Implementação do somador

A implementação do somador de dois algarismos BCD com quatro casas decimais consiste no uso consecutivo do módulo básico de soma de dois algarismos de 4 bits.

A chamada *somador1* realiza a soma dos números na casa decimal das unidades. O *carry out* dessa soma é utilizado como *carry in* da soma seguinte, a soma das dezenas. A lógica de se realizar a soma entre dois algarismos na mesma casa decimal e entre o *carry out* da soma da casa decimal anterior é aplicada nesse módulo.

4.3 – Multiplicador2Alg:

Mult2Alg.vhd

```
--library UNISIM;
--use UNISIM.VComponents.all;

entity Mult2Alg is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
             Cin: in STD_LOGIC;
           resultado : out std_logic_vector (7 downto 0);
             Cout: out STD_LOGIC);
end Mult2Alg;

architecture Behavioral of Mult2Alg is

signal produto, aux: unsigned (7 downto 0):="00000000";
signal dezena, unidade: unsigned (7 downto 0) := "00000000";

begin

           produto <= unsigned(A)*unsigned(B);

           dezena <= produto/10;

           unidade <= produto mod 10;

           aux(7 downto 4) <= dezena (3 downto 0);
           aux(3 downto 0) <= unidade (3 downto 0);

           resultado <= std_logic_vector (aux);

end Behavioral;
```

Fig. 4.3.1 – Implementação da multiplicação de dois algarismos

Este módulo define, incialmente, quatro sinais do tipo *unsigned,* um para o produto todo, outro para a dezena do produto, outro para a unidade do produto e outro auxiliar.

Inicialmente, o sinal produto recebe o produto entre o tipo *unsigned* dos argumentos. Em seguida, o sinal dezena recebe o quociente do valor salvo no sinal produto, quando esse é divido por 10. Esse valor será, obrigatoriamente, o número que estará na dezena do número. Após isso, o sinal unidade recebe o resto da divisão do valor salvo no sinal produto por 10. Este será o valor da unidade do número resultante.

Por fim, os quatro bits mais significativos do sinal auxiliar são definidos como sendo a dezena, e o quatro bits menos significativos do sinal auxiliar são definidos como sendo a unidade. Assim, é possível escrever o número do produto em BCD.

4.4 – MultiplicadorBCD:

```
entity MultiplicadorBCD is
    Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
           B : in  STD_LOGIC_VECTOR (7 downto 0);
           S : out  STD_LOGIC_VECTOR (15 downto 0);
             Cout: out STD_LOGIC );
end MultiplicadorBCD;

architecture Behavioral of MultiplicadorBCD is

component Mult2Alg is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
             Cin: in STD_LOGIC;
           resultado : out std_logic_vector (7 downto 0);
             Cout: out STD_LOGIC);
end component;

component Somador is
    Port ( A : in  STD_LOGIC_VECTOR (15 downto 0);
           B : in  STD_LOGIC_VECTOR (15 downto 0);
           soma : out  STD_LOGIC_VECTOR (15 downto 0);
           Cout : out  STD_LOGIC);
end component;


component Soma2Alg 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 component;
```

Fig. 4.4.1 – Componentes do módulo Multiplicador BCD

```
signal fator11, fator21, fator12, fator22: std_logic_vector (7 downto 0);

signal soma1, soma2: std_logic_vector (15 downto 0);

signal aux1, aux2: std_logic_vector (15 downto 0) := "0000000000000000";

signal c1, c2: std_logic;

begin

    mult11: Mult2Alg port map (A(3 downto 0), B(3 downto 0), '0', fator11, Cout);
    mult21: Mult2Alg port map (A(7 downto 4), B(3 downto 0), '0', fator21, Cout);

    aux1 (3 downto 0) <= fator11 (3 downto 0);
    s1: Soma2Alg port map (fator21 (3 downto 0), fator11 (7 downto 4), '0', aux1 (7 downto 4), c1);
    aux1 (11 downto 8) <= fator21 (7 downto 4);

    mult12: Mult2Alg port map (A(3 downto 0), B(7 downto 4), '0', fator12 (7 downto 0), Cout);
    mult22: Mult2Alg port map (A(7 downto 4), B(7 downto 4), '0', fator22 (7 downto 0), Cout);

    aux2 (7 downto 4) <= fator12 (3 downto 0);
    s2: Soma2Alg port map (fator22 (3 downto 0), fator12 (7 downto 4), '0', aux2 (11 downto 8), c2);
    aux2 (15 downto 12) <= fator22 (7 downto 4);

    somaFinal: Somador port map (aux1, aux2, S, Cout);

end Behavioral;
```

Fig. 4.4.2 – Implementação do MultiplicadorBCD

Este módulo utiliza todos os outros componentes aritméticos. Ele realiza a multiplicação somente com números de até dois algarismos.

Inicialmente, são definidos diversos sinais auxiliares. Os sinais *fator* captam o resultado individual de cada multiplicação realizada. A exemplo, a multiplicação entre a dezena do multiplicador e a unidade do multiplicando é armazenada pelo fator12. Os sinais *aux1* e *aux2* recebem, respectivamente, o resultado da multiplicação do multiplicando pela unidade do multiplicador e o resultado da multiplicação do multiplicando pela dezena do multiplicador.

O módulo *MultAlg2* é chamado quatro vezes nessa implementação. A primeira para fazer a multiplicação das unidades, a segunda para fazer a multiplicação da unidade do multiplicador pela dezena do multiplicando, a terceira para realizar a multiplicação da unidade do multiplicando pela dezena do multiplicador e a última para realizar a multiplicação entre as dezenas.

Os quatro bits menos significativos do vetor auxiliar *aux1* recebe o resultado da multiplicação entre as unidades. Do oitavo ao quinto bit de *aux1*, está o resultado da soma do fator21 (3 downto 0) com o fator11 (7 downto 4). No décimo segundo ao nono bit, está o fator21 (7 downto 4).

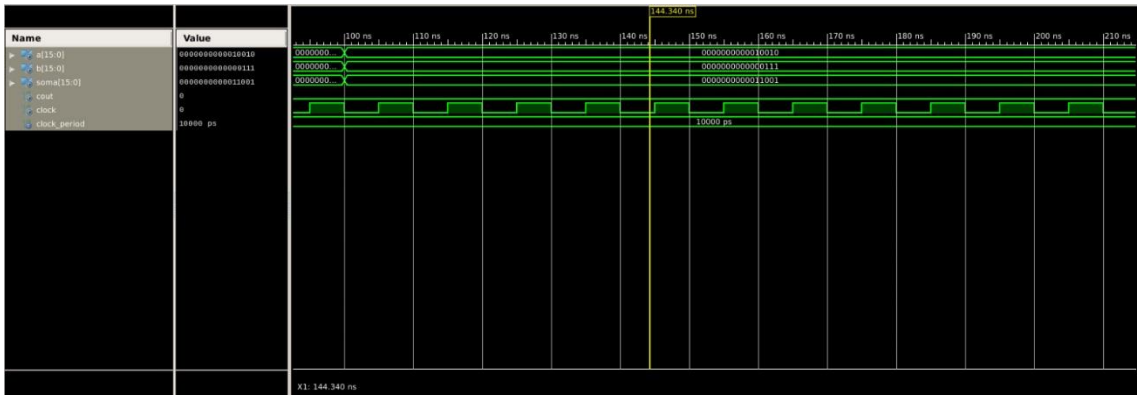
Para definir o *aux2,* é implementada a mesma lógica, utilizando os sinais fator12 e fator22. Além disso, o *aux2* tem seus quatro bits menos significativo como sendo igual a zero, para simular uma operação de *shift.*

Por fim, é chamado o módulo de somador para somar os dois auxiliares. O resultado dessa soma é o resultado da multiplicação.

5 – Resultados:

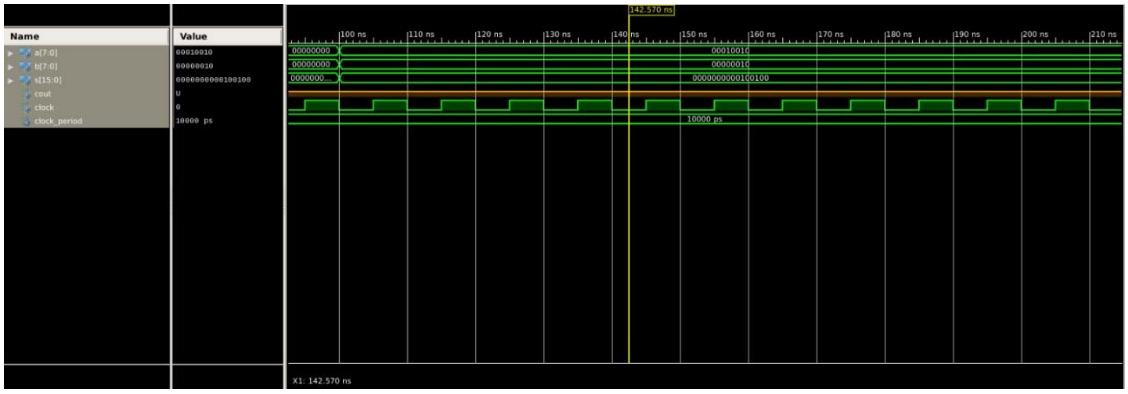
Aqui serão apresentadas as simulações dos módulos de soma e multiplicação do projeto. O clock que aparece nessas simulações foi inserido apenas para possibilitar a simulação, não tendo nenhuma influência no resultado final.

5.1 – Somador:



Na simulação do módulo Somador, foram utilizados os valores de entrada (em BCD): A = “0000000000010010” e B = “0000000000000111”, o que corresponde aos valores em decimal 12 e 7, respectivamente. Observa-se que o resultado mostra o valor “0000000000011001”, que representa o número 19 em BCD. Portanto, percebe-se o funcionamento correto do somador BCD de 4 algarismos.

5.2 – Multiplicador:



Já na simulação do módulo Multiplicador, foram utilizados os valores de entrada (em BCD): A = “00010010” e B = “00000010”, correspondendo aos valores em decimal 12 e 2, respectivamente. O resultado mostrado na saída S é “0000000000100100”, o que corresponde a representação em BCD do número 24. A simulação mostra, portanto, a lógica correta por trás do módulo implementado para o multiplicador.

6 – Conclusões:

No projeto apresentado, conseguimos implementar uma calculadora de 4 algarismos BCD, com módulos de soma e multiplicação. Porém, foram encontradas dificuldades para implementar a interface do teclado e do display junto com a calculadora, apesar do correto funcionamento dos códigos do teclado e do LCD quando implementados separadamente. Para tentar resolver esse problema, foi tentado aliar esses módulos

como componentes de uma máquina de estados, semelhante à implementada no projeto da Unidade Lógico Aritmética, para que os algarismos dos números BCD que seriam utilizados nas operações da calculadora fossem inseridos de forma sequencial pelo teclado, que logo em seguida seriam mostrados no display. Porém, a implementação da máquina de estados também falhou.