UNIVERSIDADE FEDERAL DO ABC
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

IMPLEMENTAÇÃO DE UMA CALCULADORA EM FPGA

GUSTAVO HENRIQUE BARRIONUEVO (11017011)
NILTON GOMES MARTINS JUNIOR (11029213)

Orientador: Prof. Dr. Rodrigo Moreira Bacurau

Santo André – SP
2019

GUSTAVO HENRIQUE BARRIONUEVO
NILTON GOMES MARTINS JUNIOR

IMPLEMENTAÇÃO DE UMA CALCULADORA EM FPGA

Relatório apresentado ao curso Bacharelado em Ciência da Computação da Universidade Federal do ABC como requisito parcial para aprovação na disciplina de Sistemas Digitais.

Orientador: Prof. Dr. Rodrigo Moreira Bacurau

Santo André – SP
2019

Sumário

1 Introdução

Este trabalho implementa uma calculadora aritmética simples em *VHDL*, com o projeto baseado sobre a placa de desenvolvimento *ALTERA Development and Education 1*, com o *FPGA Cyclone II EP2C20F484C7*.

Esta calculadora é capaz de realizar operações aritméticas fundamentais (soma, subtração, multiplicação e divisão). As operações ocorrem em sequência, então as precedências são definidas unicamente com base na ordem com que as operações e os operandos são definidos.

Os operandos são números inteiros, representados internamente na forma de complemento de dois.

A visualização dos operandos e do resultado é feita por meio de um conjunto de quatro *displays* de sete segmentos.

Esta calculadora recebe os valores de entrada e as operações, bem como quaisquer outros comandos, através de um teclado com interface *PS2* conectado diretamente à placa de desenvolvimento.

Além das operações aritméticas, também foi implementado o controle de brilho dos *displays* de sete segmentos, através de um controlador do tipo *PWM* (*Pulse Width Modulation*).

2 Manual de utilização

A calculadora foi implementada no *FPGA Cyclone II EP2C20F484C7*, de maneira que o uso da mesma fica limitado à dispositivos compatíveis com essa arquitetura. Este manual supõe que o leitor está utilizando o projeto na placa *ALTERA Development and Education 1* (Figura 1).

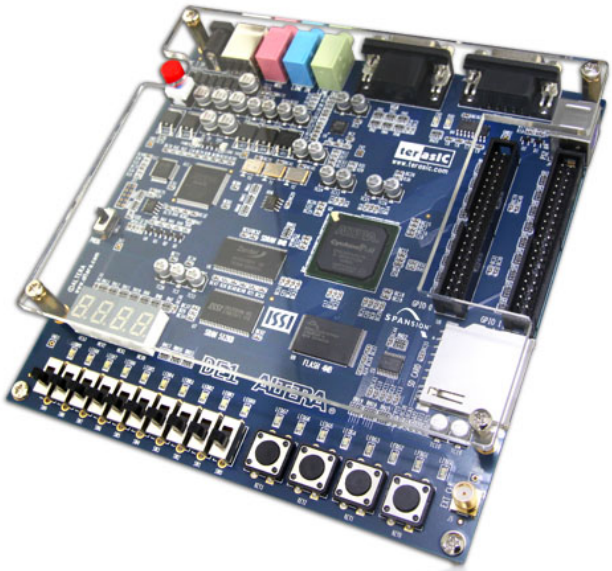

Figura 1: Placa de desenvolvimento *ALTERA Development and Education 1*.

2.1 Carregamento do programa e preparação do kit

O carregamento do arquivo de saída no formato `.sof` é feito através do *software ALTERA QUARTUS*. O procedimento a ser seguido para a realização do carregamento é descrito em detalhes por Rodrigo M. Bacurau [3].

Como a entrada de dados para as operações aritméticas é feita exclusivamente com o teclado, é necessário conectar um teclado com padrão *PS2* na entrada apropriada da placa de desenvolvimento.

2.2 Operações aritméticas com dois operandos

Com o código carregado na placa e o teclado devidamente conectado, o programa encontra-se no estado inicial, onde nenhum operando e nenhuma operação foram definidos, e o resultado padrão é 0.

A calculadora suporta entrada de números representáveis com até 13 *bits* em complemento de dois, isso é, podem ser inseridos números com módulo no intervalo de 0 à 4095. Os valores negativos são obtidos indiretamente, através de operações de subtração ou de multiplicação (caso o resultado atual seja negativo). O intervalo de valores que podem ser representados na calculadora é [-4096, 4095]. Não é possível inserir valores maiores do que esse (há um bloqueio de leitura). Se alguma operação for realizada de tal forma que o resultado não pertença ao intervalo, o resultado será *ERRO*.

Ao se digitar uma entrada válida com o teclado, o que pode ser feito tanto com o teclado numérico (*numpad*), quanto com as teclas numéricas comuns, o valor de entrada é exibido nos *displays* de sete segmentos, substituindo o resultado anterior.

Com a inserção de um operando finalizada, é necessário selecionar o operador, utilizando o teclado numérico. As opções são:

- $+$: adição;
- -: subtração;
- *: multiplicação;
- $/$: divisão (inteira).

Nesta etapa, se for selecionada alguma operação por engano, basta pressionar a tecla correspondente à operação desejada para que a operação seja sobrescrita, não sendo necessário reiniciar o cálculo com a inserção do primeiro operando.

Após a seleção da operação desejada, prossegue-se com a definição do segundo operando, que é feita da mesma maneira com que o primeiro foi definido, observando as mesmas restrições.

Uma vez que o operando tenha sido selecionado, basta apertar a tecla *ENTER* para que o resultado da operação seja exibido nos *displays*. Para números positivos dentro do intervalo válido de operação a visualização é convencional, com o algarismo mais significativo do lado esquerdo. Para números negativos a representação do sinal se dá de duas possíveis maneiras:

- Sinal representado num *display* e com LED vermelho: Caso o resultado negativo tenha módulo inferior à 1000, isto é, está contido no intervalo [-999, -1], o LED vermelho número 9 (LEDR9, localizado mais à esquerda da placa) será aceso e o sinal será exibido no *display* da esquerda do algarismo mais significativo. Por exemplo, o número -1 tem seu sinal representado no terceiro *display*, da esquerda para a direita, enquanto o número -100 tem o sinal no primeiro *display*. Se houverem *displays* à esquerda do sinal, os mesmos permanecerão desativados;
- Sinal representado apenas com LED vermelho: Caso o sinal não possa ser representado nos *displays*, quando o resultado tem módulo superior ou igual à 1000 e, portanto, demanda quatro algarismos, apenas o LED vermelho ficará aceso.

2.3 Operações aritméticas em sequência

Com um resultado salvo, é possível operar sobre o mesmo utilizando uma recorrência com a última operação e o último operando. Isto é realizado através de sucessivos pressionamentos da tecla *ENTER*. Por exemplo, supondo que utiliza-se o estado inicial, com resultado 0, e deseja-se incrementar este resultado em 2 unidades à cada pressionamento de *ENTER*, basta selecionar a operação de adição (tecla $+$), e o operando 2. Ao se apertar *ENTER* pela primeira vez, conforme uma operação convencional, o resultado exibido será 2 (uma vez que $0 + 2 = 2$). Ao se pressionar *ENTER* novamente, sem a necessidade de se selecionar uma operação ou outro operando, o valor 2 será novamente adicionado ao resultado atual, 2, e o resultado 4 será exibido ($2 + 2 = 4$). Isso pode ser realizado indefinidamente, respeitando a limitação de representação no intervalo [-4096, 4095].

As operações em sequência também pode ser feitas com outras operações e operandos. Após cada operação ser finalizada, se um resultado válido for obtido, este resultado é armazenado e utilizado como o primeiro operando da operação subsequente. Assim, basta selecionar uma nova operação e outro número e apertar *ENTER* novamente para reutilizar o resultado anterior num novo cálculo.

2.4 Limpando a memória da calculadora

Tanto o cancelamento de qualquer operação quanto a saída do estado de *ERRO* são feitos através da tecla *ESC*. Com isso todos os resultados são limpos da memória e a calculadora é zerada, retornando ao estado inicial.

2.5 Controle de brilho do display

O controle de brilho do *display* é feito através das oito primeiras chaves (*SW0-SW7*) . Com todas as chaves em nível baixo (posição inferior) se obtém o brilho máximo. Com os oito *bits* representados por cada chave, o brilho pode variar entre 256 níveis de intensidade. A chave com o *bit* mais significativo é a SW7.

O controle de brilho dos *displays* é feita independentemente da operação da calculadora. Desse modo, o controle de brilho com as chaves pode ser feito à qualquer momento sem nenhum impacto ao estado atual da operação aritmética corrente.

3 Organização do projeto

O projeto é estruturado em torno do arquivo principal `calculator.vhd`, que contém a descrição da máquina de estados que é o cerne deste trabalho.

O código é descrito no apêndice A.1.

A máquina de estados é descrita de forma geral pela Figura 2.

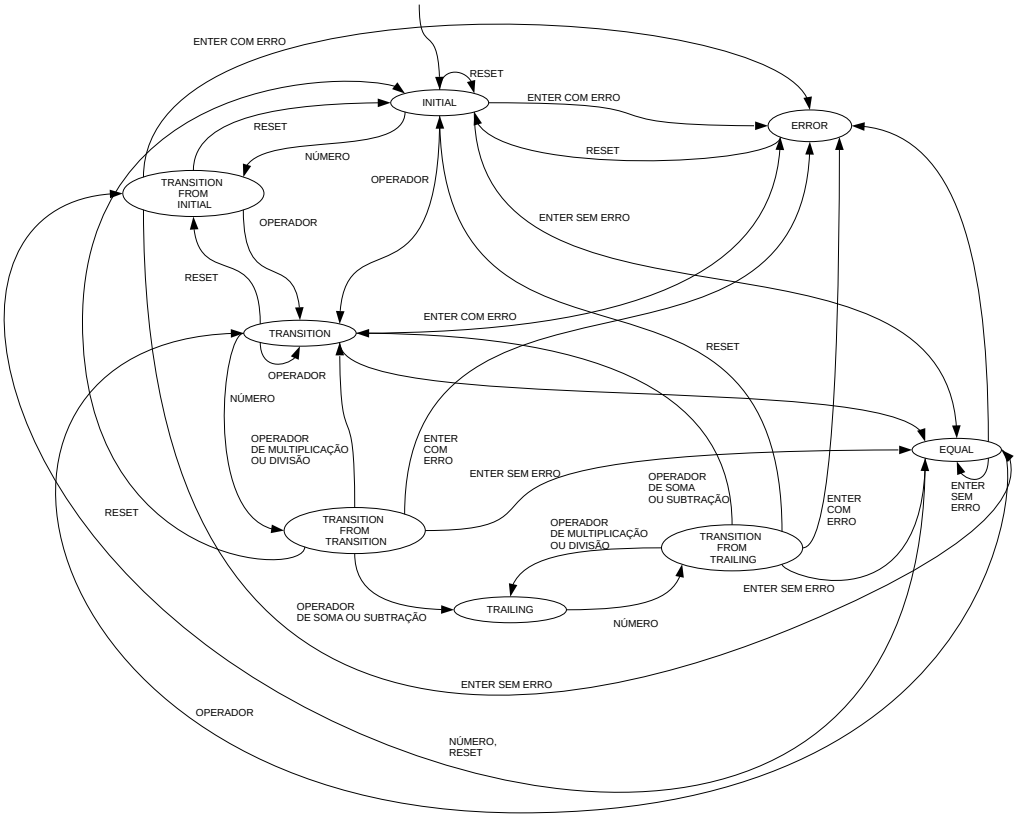

Figura 2: Descrição geral da máquina de estados.

Existem três operandos de interesse: o primeiro, o segundo e o subsequente, denominados no código como `fn`, `sn` e `tn`, respectivamente. Os dois primeiros definem as operações convencionais e o terceiro é utilizado como uma variável auxiliar para operações sequenciais.

Os estados são descritos por:

- *INITIAL*: Estado inicial, com resultado zero. Obtido no início da execução ou com o uso do *RESET*;
- *TRANSITION FROM INITIAL*: Estado que identifica a entrada do primeiro operando;
- *TRANSITION*: Estado que identifica que o primeiro operando foi definido e determina a entrada da operação;
- *TRANSITION FROM TRANSITION*: Identifica a entrada do segundo operando, após a definição da operação;

- *EQUAL*: Após a inserção do segundo operando, exibe o resultado da operação, caso o mesmo seja válido;
- *TRAILING*: Caso onde são feitas operações sequenciais, lê-se um operador para a próxima operação sobre o resultado atual;
- *TRANSITION FROM TRAILING* : Ainda no caso de operações sequenciais, lê o número subsequente, que se torna o segundo operando, enquanto que o primeiro operando é o resultado da operação anterior;
- *ERROR*: Estado de erro, quando algum resultado inválido é obtido (fora do intervalo pré-determinado ou divisão por zero).

Além da definição dos estados e transições, o arquivo principal também estrutura a interação entre as bibliotecas genéricas (`general`), de entrada e saída (`io`) e numérica (`numeric`). Os arquivos que constituem cada uma destas bibliotecas estão dentro do diretório `/lib/`.

3.1 Biblioteca genérica

A biblioteca genérica é composta pelos seguintes arquivos:

- `binary_to_bcd.vhd`: realiza a conversão de um valor binário de 13 *bits* (em complemento de 2) para o formato *BCD* com 16 *bits*.

  O código está descrito na seção A.4.1.

  Este código, implementado em outra aula prática da disciplina de Sistemas Digitais [5], é uma replicação do algoritmo de *Double-Dabble*, que codifica um número binário representado por uma quantidade arbitrária de *bits* na forma *BCD*. O algoritmo é um registrador de deslocamento que itera tantas vezes quantos forem o número de *bits* e, a cada iteração, adiciona o valor 3 à cada *nibble* se este representar um valor numérico superior à 4;
- `conv_7seg.vhd`: converte um valor binário de 4 *bits* referente à um dígito de 0 à 9, sinal negativo ou valor nulo, para *display* de sete segmentos.

  O código é descrito na seção A.4.2;
- `general.vhd`: define o pacote com os demais arquivos da biblioteca genérica;
- `pwm.vhd`: define o controlador *PWM* para o controle de brilho dos *displays* de sete segmentos.

  O código é descrito em A.4.3.

  O algoritmo consiste numa combinação de um contador e um comparador. O contador incrementa a contagem a cada pulso de *clock* e o comparador ativa a saída caso a contagem seja inferior ao valor do *duty-cycle*. O tempo que a saída permanece ativa em relação ao período da onda determina a potência luminosa dos *LEDs* de cada *display* de sete segmentos;
- `scancode_to_calc_input.vhd`: realiza a conversão de uma entrada do teclado *PS2* para um vetor de 5 *bits* representando um dígito, uma operação aritmética, o comando *ENTER* ou o comando *CLEAR*.

  O código é descrito na seção A.4.4.

3.2 Biblioteca de I/O

A biblioteca de entrada e saída é composta pelos seguintes arquivos:

- `io.vhd`: define o pacote com os demais arquivos da biblioteca de entrada e saída;
- `kbdex_ctrl.vhd`: define o controlador para o teclado *PS2*, permitindo a leitura do código referente à um conjunto de até três teclas, totalizando 48 *bits*;
- `ps2_iobase.vhd`: realiza a comunicação com o teclado através da porta *PS2*.

3.3 Biblioteca numérica

A biblioteca numérica contem um único arquivo, `numeric.vhd`, que define o pacote contendo todas as operações aritméticas, bem como os algoritmos que as implementam.

O código com o pacote é descrito em A.2

O pacote com as operações aritméticas contém as funções:

- `bit_adder_subtractor()`: Realiza as operações de soma e subtração com apenas um *bit*. É o algoritmo de um somador completo;
- `ripple_adder_subtractor()`: Realiza as operações de soma e subtração com vetores de *bits*. Este algoritmo implementa a estrutura de um somador completo de múltiplos *bits*, que é o encadeamento de somadores completos de um *bit* em cascata, onde o *carry-out* de um é alimentado ao *carry-in* do seguinte (mais significativo);
- `booth_multiplier()`: Implementa um multiplicador de Booth para vetores de *bits* representando números em complemento de dois. O algoritmo do multiplicador de Booth é baseado numa implementação descrita em aula [2];
- `divide()`: Utiliza o método de divisão do pacote `IEEE.NUMERIC_STD`.

4 Problemas identificados e não resolvidos e demais limitações de projeto

Não foram detectados *bugs* com solução pendente, mas listam-se as principais limitações do projeto:

- Há a limitação de representação com 13 *bits*, de modo que os valores inseridos e/ou resultados obtidos devem estar contidos no intervalo [-4096, 4095];
- Os valores podem ser representados com os quatro *displays*, mas o sinal negativo para números com quatro dígitos precisou ser representado exclusivamente através de um LED auxiliar;
- Todos os valores de entrada e resultados são inteiros, não foi implementado ponto fixo ou flutuante;
- Muito embora todas as operações aritméticas básicas tenham sido implementadas, a divisão é inteira, e o resto é desconsiderado, isto é, o resultado da divisão entre dois operandos $a, b \in \mathbb{Z}$ é dado por $a \div b = \lfloor a/b \rfloor$.

5 Autoria dos códigos utilizados

Para a estrutura da aplicação descrita na Seção 3, são feitas as considerações referentes à autoria dos códigos implementados.

São códigos originais aqueles contidos nos arquivos;

- `binary_to_bcd.vhd`;
- `conv_7seg.vhd`;
- `general.vhd`;
- `pwm.vhd`;
- `scancode_to_calc_input.vhd`;
- `io.vhd`;

O trecho de código do arquivo principal, `calculator.vhd`, que estrutura a máquina de estados em *VHDL* é baseado em uma implementação em *Javascript* por Robert Vunabadi [7]. O resto do código contido no arquivo implementa as bibliotecas númerica, de entrada e saída e outras generalidades. Esta parte do código é original.

Para os códigos da biblioteca númerica, o somador/subtrator completo de um *bit* e o somador/subtrator completo em cascata foram implementados originalmente em aulas práticas anteriores da disciplina [6].

Os códigos da biblioteca de entrada e saída referentes à comunicação com o teclado *PS2* foram elaborados por Thiago Borges Abdnur, do Instituto de Computação da Unicamp (Universidade Estadual de Campinas) [1]. No contexto de entrada e saída, apenas o código que realiza a conversão do *scancode* do teclado para um formato internamente interpretável é original (este código está contido na biblioteca genérica).

Referências

[1] Thiago Borges Abdnur. Keyboard controller. Disponível em: www.computer-engineering.org/ps2protocol/. Acesso em: 22 out. 2019.

[2] Rodrigo Moreira Bacurau. Pacotes, bibliotecas e algortimos de multiplicação. Disponível em: https://sites.google.com/site/rodrigobacurau/cursos-2019-2/sistemas-digitais, . Acesso em: 22 out. 2019.

[3] Rodrigo Moreira Bacurau. Introdução à programação em vhdl utilizando o software quartus ii e o kit cyclone ii. Disponível em: https://sites.google.com/site/rodrigobacurau/cursos-2019-2/sistemas-digitais, . Acesso em: 22 out. 2019.

[4] Rodrigo Moreira Bacurau. Uso de bibliotecas para comunicação com teclado ps2 e display vga. Disponível em: https://sites.google.com/site/rodrigobacurau/cursos-2019-2/sistemas-digitais, . Acesso em: 22 out. 2019.

[5] Rodrigo Moreira Bacurau. Modulação por largura de pulsos - pwm & conversão binário-bcd usando o algoritmo double-dabble. Disponível em: https://sites.google.com/site/rodrigobacurau/cursos-2019-2/sistemas-digitais, . Acesso em: 22 out. 2019.

[6] Rodrigo Moreira Bacurau. Projeto de circuitos combinacionais em vhdl: somador de 4 bits com exibição do resultado em display de 7 segmentos. Disponível em: https://sites.google.com/site/rodrigobacurau/cursos-2019-2/sistemas-digitais, . Acesso em: 22 out. 2019.

[7] Robert Vunabandi. Designing a calculator with fsm logic. Disponível em: https://medium.com/@rvunabandi/making-a-calculator-in-javascript-64193ea6a492. Acesso em: 22 out. 2019.

A Apêndice - Códigos desenvolvidos

Os códigos desenvolvidos e implementados são brevemente discutidos.

A.1 Código principal

O arquivo `calculator.vhd` contém a descrição da máquina de estados que define o funcionamento da calculadora. O código é exibido à seguir.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  LIBRARY lib;
5  USE lib.io.ALL;
6  USE lib.general.ALL;
7  USE lib.numeric.ALL;
8
9  ENTITY calculator IS
10 PORT
11 (
12 ------------------------        Clock Input                 ------------------------
13 clock_24 :          IN        STD_LOGIC_VECTOR (1 DOWNTO 0);        --        24 MHz
14 clock_27 :         IN        STD_LOGIC_VECTOR (1 DOWNTO 0);        --        27 MHz
15 clock_50 :
   ↪          IN        STD_LOGIC;                                          --        50
   ↪  MHz
16
17 ------------------------        Push Button                 ------------------------
18 key :        IN        STD_LOGIC_VECTOR (3 DOWNTO 0);        --        Pushbutton[3:0]
19
20 ------------------------        DPDT Switch                 ------------------------
21 sw         :        IN        STD_LOGIC_VECTOR (9 DOWNTO 0);        --        Toggle Switch[9:0]
22
23 ------------------------        7-SEG Display        ------------------------
24 hex0         :        OUT        STD_LOGIC_VECTOR (6 DOWNTO 0);        --        Seven Segment
   ↪  Digit 0
25 hex1         :        OUT        STD_LOGIC_VECTOR (6 DOWNTO 0);        --        Seven Segment
   ↪  Digit 1
26 hex2         :        OUT        STD_LOGIC_VECTOR (6 DOWNTO 0);        --        Seven Segment
   ↪  Digit 2
27 hex3         :        OUT        STD_LOGIC_VECTOR (6 DOWNTO 0);        --        Seven Segment
   ↪  Digit 3
28
29 ----------------------------        LED                 ----------------------------
30 ledg         :        OUT        STD_LOGIC_VECTOR (7 DOWNTO 0);        --        LED Green[7:0]
31 ledr         :        OUT        STD_LOGIC_VECTOR (9 DOWNTO 0);        --        LED Red[9:0]
32
33 ------------------------        PS2                 --------------------------------
34 ps2_dat        :        INOUT        STD_LOGIC;        --        PS2 Data
35 ps2_clk        :        INOUT        STD_LOGIC        --        PS2 Clock
36 );
37 END calculator;
```

```
38
39  ARCHITECTURE struct OF calculator IS
40
41  SIGNAL clockhz, resetn          : STD_LOGIC;
42  SIGNAL key0                                 : STD_LOGIC_VECTOR(15 DOWNTO 0);
43  SIGNAL lights, key_on         : STD_LOGIC_VECTOR(2 DOWNTO 0);
44
45  -- Sinal utilizado para armazenar os possíveis
46  -- valores de entrada do teclado PS2.
47  SIGNAL calc_input : STD_LOGIC_VECTOR(4 DOWNTO 0);
48
49  -- Sinal utilizado para armazenar o código BCD
50  -- que será utilizado para apresentar os valores
51  -- númericos da calculadora nos displays de 7 segmentos
52  SIGNAL bcd : STD_LOGIC_VECTOR(15 DOWNTO 0);
53
54  -- Sinal utilizado para armazenar o primeiro operando da
55  -- calculadora
56  SIGNAL first_number : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
57  -- Sinal utilizado para armazenar o segundo operando da
58  -- calculadora
59  SIGNAL second_number : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
60  -- Sinal utilizado para armazenar o operando de trailing da
61  -- calculadora
62  SIGNAL trailing_number : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
63
64  -- Sinal utilizado para armazenar o valor atual do display
65  SIGNAL display_number : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
66
67  -- Sinal utilizado para armazenar o primeiro operador da
68  -- FSM da calculadora
69  SIGNAL operation1 : STD_LOGIC_VECTOR(4 DOWNTO 0);
70
71  -- Sinal utilizado para armazenar o segundo operador da
72  -- FSM da calculadora
73  SIGNAL operation2 : STD_LOGIC_VECTOR(4 DOWNTO 0);
74
75  -- Definição dos 8 possíveis estados da máquina de
76  -- estados da calculadora.
77  TYPE state_type IS (
78  INITIAL,
79  TRANSITION_FROM_INITIAL,
80  TRANSITION,
81  TRANSITION_FROM_TRANSITION,
82  TRAILING,
83  TRANSITION_FROM_TRAILING,
84  EQUAL,
85  ERROR
86  );
87
88  -- Sinais utilizados para armazenar o estado atual (state)
89  -- e o próximo estado (next_state) da máquina de estados
90  -- da calculadora.
```

```
91  SIGNAL state, next_state : state_type;
92
93  SIGNAL pwm_out : STD_LOGIC;
94
95  SIGNAL duty : STD_LOGIC_VECTOR(7 DOWNTO 0);
96
97  -- Sinais utilizados para armazenar os valores
98  -- dos quatro displays de 7 segmentos
99  SIGNAL f0, f1, f2, f3 : STD_LOGIC_VECTOR(6 DOWNTO 0);
100
101 -- Sinal utilizado para indicar a ocorrência de algum
102 -- erro na ultima operação realizada pela calculadora.
103 SIGNAL error_result : STD_LOGIC;
104
105 BEGIN
106
107 resetn <= key(0);
108
109 -- Definição do duty utilizado pelo gerador PWM.
110 duty <= sw(7 DOWNTO 0);
111
112 -- Instanciação do controlador PWM.
113 pwm_controller: pwm PORT MAP(
114 clk => clock_50,
115 enable => '1',
116 rstn => '1',
117 duty => duty,
118 pwm_out => pwm_out
119 );
120
121 -- Responsável pela conversão dos scancodes do teclado para um formato
122 -- interno utilizado na calculadora.
123 conv_input: scancode_to_calc_input PORT MAP(
124 scancode => key0(7 DOWNTO 0),
125 calc_input => calc_input
126 );
127
128 -- Processo responsável pelo controle de brilho dos displays de 7 segmentos.
129 -- Também é responsável por mostrar a mensagem ERRO nos displays de 7 segmentos
130 -- caso algum erro ocorra.
131 pwm_dimmer: PROCESS(pwm_out)
132 BEGIN
133 IF (pwm_out = '1') THEN
134 hex0 <= (OTHERS => '1');
135 hex1 <= (OTHERS => '1');
136 hex2 <= (OTHERS => '1');
137 hex3 <= (OTHERS => '1');
138 ELSE
139 IF (error_result = '0') THEN
140 hex0 <= f0;
141 hex1 <= f1;
142 hex2 <= f2;
143 hex3 <= f3;
```

```
144 ELSE
145 hex0 <= "1000000"; -- E
146 hex1 <= "0001000"; -- R
147 hex2 <= "0001000"; -- R
148 hex3 <= "0000110"; -- O
149 END IF;
150 END IF;
151 END PROCESS;
152
153 -- Processo responsável pela mudança para o estado seguinte da FSM.
154 register_func: PROCESS(clock_24(0), resetn)
155 BEGIN
156 IF (resetn = '0') THEN
157 state <= INITIAL;
158 ELSIF (clock_24(0)'event and clock_24(0) = '1') THEN
159 state <= next_state;
160 END IF;
161 END PROCESS;
162
163 -- Processo responsável pela definição do próximo estado da FSM
164 -- de acordo com os valores de entrada do teclado e o estado atual.
165 next_state_func: PROCESS(calc_input, state)
166
167 -- Variáveis temporárias utilizadas para armazenar os valores do primeiro
168 -- operando (fn), segundo operando (sn) e operando de trailing (tn).
169 VARIABLE fn, sn, tn : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
170
171 -- Variáveis utilizadas para armazenar um indicador de possível erro
172 -- de acordo com os operandos e operadores possíveis.
173 VARIABLE error_result_f_op1_s, error_result_s_op2_t, error_result_f_op1_s_op2_t  : STD_LOGIC;
174
175 BEGIN
176 IF (key_on(0)'event and key_on(0) = '1') THEN
177
178 -- Inicializa os valores das variáveis temporárias com
179 -- os valores atuais dos sinais correspondentes.
180 fn := first_number;
181 sn := second_number;
182 tn := trailing_number;
183
184 CASE state IS
185
186 WHEN INITIAL =>
187 fn := ZERO;
188 operation1 <= SUM;
189 sn := ZERO;
190 operation2 <= SUM;
191 tn := ZERO;
192 error_result_f_op1_s := '0';
193 error_result_s_op2_t := '0';
194 error_result_f_op1_s_op2_t := '0';
195 error_result <= '0';
196
```

```
197  ------------------ lida com entrada numérica -----------------
198  IF (is_number(calc_input)) THEN
199  fn := ZERO;
200  fn := get_resulting_display(fn, calc_input(3 DOWNTO 0));
201  next_state <= TRANSITION_FROM_INITIAL;
202  ------------------------------------------------------------
203
204  --------------- lida com a tecla enter (igual) ---------------
205  ELSIF (calc_input = ENTER) THEN
206  evaluate(fn, sn, operation1, fn, error_result_f_op1_s);
207  IF (error_result_f_op1_s = '0') THEN
208  next_state <= EQUAL;
209  ELSE
210  error_result <= error_result_f_op1_s;
211  next_state <= ERROR;
212  END IF;
213  ------------------------------------------------------------
214
215  ---------------- lida com a tecla esc (reset) ----------------
216  ELSIF (calc_input = RES) THEN
217  next_state <= INITIAL;
218  ------------------------------------------------------------
219
220  ---------------- lida com entrada de operador ----------------
221  ELSIF (is_operation(calc_input)) THEN
222  -- pressionando qualquer tecla de operador (+,-,*/)
223  fn := sn;
224  operation1 <= calc_input;
225  next_state <= TRANSITION;
226  END IF;
227  ------------------------------------------------------------
228
229
230  WHEN TRANSITION_FROM_INITIAL =>
231
232  ------------------ lida com entrada numérica -----------------
233  IF (is_number(calc_input)) THEN
234  fn := get_resulting_display(fn, calc_input(3 DOWNTO 0));
235  ------------------------------------------------------------
236
237  --------------- lida com a tecla enter (igual) ---------------
238  ELSIF (calc_input = ENTER) THEN
239  evaluate(fn, sn, operation1, fn, error_result_f_op1_s);
240  IF (error_result_f_op1_s = '0') THEN
241  next_state <= EQUAL;
242  ELSE
243  error_result <= error_result_f_op1_s;
244  next_state <= ERROR;
245  END IF;
246  ------------------------------------------------------------
247
248  ---------------- lida com a tecla esc (reset) ----------------
249  ELSIF (calc_input = RES) THEN
```

```
250 IF (fn /= ZERO) THEN
251 fn := ZERO;
252 ELSE
253 next_state <= INITIAL;
254 END IF;
255 ------------------------------------------------------------
256
257 ---------------- lida com entrada de operador ----------------
258 ELSIF (is_operation(calc_input)) THEN
259 -- caso a entrada seja qualquer operação:
260 -- move para TRANSITION
261 sn := fn; -- to verify
262 operation1 <= calc_input;
263 next_state <= TRANSITION;
264 END IF;
265 ------------------------------------------------------------
266
267
268 WHEN TRANSITION =>
269
270 ------------------ lida com entrada numérica -----------------
271 IF (is_number(calc_input)) THEN
272 -- caso entrada seja qualquer número: move de volta para
273 -- TRANSITION_FROM_TRANSITION
274 sn := ZERO;
275 sn := get_resulting_display(sn, calc_input(3 DOWNTO 0));
276 next_state <= TRANSITION_FROM_TRANSITION;
277 ------------------------------------------------------------
278
279 --------------- lida com a tecla enter (igual) ---------------
280 ELSIF (calc_input = ENTER) THEN
281 evaluate(fn, sn, operation1, fn, error_result_f_op1_s);
282 IF (error_result_f_op1_s = '0') THEN
283 next_state <= EQUAL;
284 ELSE
285 error_result <= error_result_f_op1_s;
286 next_state <= ERROR;
287 END IF;
288 ------------------------------------------------------------
289
290 ---------------- lida com a tecla esc (reset) ----------------
291 ELSIF (calc_input = RES) THEN
292 fn := ZERO;
293 next_state <= TRANSITION_FROM_INITIAL;
294 ------------------------------------------------------------
295
296 ---------------- lida com entrada de operador ----------------
297 ELSIF (is_operation(calc_input)) THEN
298 -- caso a entrada seja qualquer operação:
299 -- move de volta para TRANSITION
300 operation1 <= calc_input;
301 next_state <= TRANSITION;
302 ------------------------------------------------------------
```

```
303 END IF;
304
305 WHEN TRANSITION_FROM_TRANSITION =>
306
307 ------------------ lida com entrada numérica -----------------
308 IF (is_number(calc_input)) THEN
309 sn := get_resulting_display(sn, calc_input(3 DOWNTO 0));
310 ------------------------------------------------------------
311
312 --------------- lida com a tecla enter (igual) ---------------
313 ELSIF (calc_input = ENTER) THEN
314 evaluate(fn, sn, operation1, fn, error_result_f_op1_s);
315 IF (error_result_f_op1_s = '0') THEN
316 next_state <= EQUAL;
317 ELSE
318 error_result <= error_result_f_op1_s;
319 next_state <= ERROR;
320 END IF;
321 ------------------------------------------------------------
322
323 ---------------- lida com a tecla esc (reset) ----------------
324 ELSIF (calc_input = RES) THEN
325 IF (sn /= ZERO) THEN
326 sn := ZERO;
327 ELSE
328 next_state <= INITIAL;
329 END IF;
330 ------------------------------------------------------------
331
332 ---------------- lida com entrada de operador ----------------
333 ELSIF (is_complex_operation(calc_input) AND is_simple_operation(operation1)) THEN
334 operation2 <= calc_input;
335 tn := sn;
336 next_state <= TRAILING;
337 ELSIF (is_simple_operation(calc_input) OR is_complex_operation(operation1)) THEN
338 evaluate(fn, sn, operation1, fn, error_result_f_op1_s);
339 IF (error_result_f_op1_s = '0') THEN
340 operation1 <= calc_input;
341 sn := fn;
342 next_state <= TRANSITION;
343 ELSE
344 error_result <= error_result_f_op1_s;
345 next_state <= ERROR;
346 END IF;
347 END IF;
348 ------------------------------------------------------------
349
350
351 WHEN TRAILING =>
352
353 ------------------ lida com entrada numérica -----------------
354 IF (is_number(calc_input)) THEN
355 tn := ZERO;
```

```
356  tn := get_resulting_display(tn, calc_input(3 DOWNTO 0));
357  next_state <= TRANSITION_FROM_TRAILING;
358  ------------------------------------------------------------
359
360  --------------- lida com a tecla enter (igual) ---------------
361  ELSIF (calc_input = ENTER) THEN
362
363  evaluate(sn, tn, operation2, sn, error_result_s_op2_t);
364  evaluate(fn, sn, operation1, fn, error_result_f_op1_s_op2_t);
365  IF (error_result_f_op1_s_op2_t = '0') THEN
366  next_state <= EQUAL;
367  ELSE
368  error_result <= error_result_f_op1_s_op2_t;
369  next_state <= ERROR;
370  END IF;
371  ------------------------------------------------------------
372
373  ---------------- lida com a tecla esc (reset) ----------------
374  ELSIF (calc_input = RES) THEN
375  tn := ZERO;
376  next_state <= TRANSITION_FROM_TRAILING;
377  ------------------------------------------------------------
378
379  ---------------- lida com entrada de operador ----------------
380  ELSIF (is_simple_operation(calc_input)) THEN
381  -- caso de operação simples (+,-): move de volta para TRANSITION
382  evaluate(sn, tn, operation2, sn, error_result_s_op2_t);
383  evaluate(fn, sn, operation1, fn, error_result_f_op1_s_op2_t);
384  IF (error_result_f_op1_s_op2_t = '0') THEN
385  sn := fn;
386  operation1 <= calc_input;
387  next_state <= TRANSITION;
388  ELSE
389  error_result <= error_result_f_op1_s_op2_t;
390  next_state <= ERROR;
391  END IF;
392  ELSIF (is_complex_operation(calc_input)) THEN
393  -- caso de operação complexa (*,/): permanece em TRAILING
394  operation2 <= calc_input;
395  END IF;
396  ------------------------------------------------------------
397
398
399  WHEN TRANSITION_FROM_TRAILING =>
400
401  ------------------ lida com entrada numérica -----------------
402  IF (is_number(calc_input)) THEN
403  tn := get_resulting_display(tn, calc_input(3 DOWNTO 0));
404  ------------------------------------------------------------
405
406  --------------- lida com a tecla enter (igual) ---------------
407  ELSIF (calc_input = ENTER) THEN
408  evaluate(sn, tn, operation2, sn, error_result_s_op2_t);
```

```
409 evaluate(fn, sn, operation1, fn, error_result_f_op1_s_op2_t);
410 IF (error_result_f_op1_s_op2_t = '0') THEN
411 next_state <= EQUAL;
412 ELSE
413 error_result <= error_result_f_op1_s_op2_t;
414 next_state <= ERROR;
415 END IF;
416 ------------------------------------------------------------
417
418 ---------------- lida com a tecla esc (reset) ----------------
419 ELSIF (calc_input = RES) THEN
420 IF (tn /= ZERO) THEN
421 tn := ZERO;
422 ELSE
423 next_state <= INITIAL;
424 END IF;
425 ------------------------------------------------------------
426
427 ---------------- lida com entrada de operador ----------------
428 ELSIF (is_simple_operation(calc_input)) THEN
429 evaluate(sn, tn, operation2, sn, error_result_s_op2_t);
430 evaluate(fn, sn, operation1, fn, error_result_f_op1_s_op2_t);
431 -- caso de operação simples (+,-): move de volta para TRANSITION
432 IF (error_result_f_op1_s_op2_t = '0') THEN
433 operation1 <= calc_input;
434 sn := fn;
435 next_state <= TRANSITION;
436 ELSE
437 error_result <= error_result_f_op1_s_op2_t;
438 next_state <= ERROR;
439 END IF;
440
441 ELSIF (is_complex_operation(calc_input)) THEN
442 -- caso de operação complexa (*,/): permanece em TRAILING
443 evaluate(sn, tn, operation2, sn, error_result_s_op2_t);
444 IF (error_result_s_op2_t = '0') THEN
445 tn := sn;
446 operation2 <= calc_input;
447 next_state <= TRAILING;
448 ELSE
449 error_result <= error_result_s_op2_t;
450 next_state <= ERROR;
451 END IF;
452
453 END IF;
454 ------------------------------------------------------------
455
456
457 WHEN EQUAL =>
458
459 ------------------ lida com entrada numérica -----------------
460 IF (is_number(calc_input)) THEN -- any number
461 fn := ZERO;
```

```
462  fn := get_resulting_display(fn, calc_input(3 DOWNTO 0));
463  next_state <= TRANSITION_FROM_INITIAL;
464  ------------------------------------------------------------
465
466  --------------- lida com a tecla enter (igual) ---------------
467  ELSIF (calc_input = ENTER) THEN
468  evaluate(fn, sn, operation1, fn, error_result_f_op1_s);
469  IF (error_result_f_op1_s = '0') THEN
470  next_state <= EQUAL;
471  ELSE
472  error_result <= error_result_f_op1_s;
473  next_state <= ERROR;
474  END IF;
475  ------------------------------------------------------------
476
477  ---------------- lida com a tecla esc (reset) ----------------
478  ELSIF (calc_input = RES) THEN
479  fn := ZERO;
480  next_state <= TRANSITION_FROM_INITIAL;
481  ------------------------------------------------------------
482
483  ---------------- lida com entrada de operador ----------------
484  ELSIF (is_operation(calc_input)) THEN -- any operation
485  operation1 <= calc_input;
486  sn := fn;
487  next_state <= TRANSITION;
488  END IF;
489  ------------------------------------------------------------
490
491
492  WHEN ERROR =>
493  ---------------- lida com a tecla esc (reset) ----------------
494  IF (calc_input = RES) THEN
495  next_state <= INITIAL;
496  error_result <= '0';
497  fn := (OTHERS => '0');
498  END IF;
499  ------------------------------------------------------------
500
501  END CASE;
502
503  -- Atualiza os valores dos sinais com suas respectivas variáveis.
504  first_number <= fn;
505  second_number <= sn;
506  trailing_number <= tn;
507
508  END IF;
509  END PROCESS;
510
511  -- Função que define o valor de saída da máquina de estados da calculadora
512  -- que será apresentado nos quatro displays de 7 segmentos de acordo com
513  -- o estado atual.
514  output_func: PROCESS(calc_input, state)
```

```
515 BEGIN
516 CASE state IS
517
518 WHEN INITIAL =>
519 display_number <= first_number;
520 ledr(7 DOWNTO 0) <= (OTHERS => '0');
521 ledr(7) <= '1';
522
523 WHEN TRANSITION_FROM_INITIAL =>
524 display_number <= first_number;
525 ledr(7 DOWNTO 0) <= (OTHERS => '0');
526 ledr(6) <= '1';
527
528 WHEN TRANSITION =>
529 display_number <= first_number;
530 ledr(7 DOWNTO 0) <= (OTHERS => '0');
531 ledr(5) <= '1';
532
533 WHEN TRANSITION_FROM_TRANSITION =>
534 display_number <= second_number;
535 ledr(7 DOWNTO 0) <= (OTHERS => '0');
536 ledr(4) <= '1';
537
538 WHEN TRAILING =>
539 display_number <= second_number;
540 ledr(7 DOWNTO 0) <= (OTHERS => '0');
541 ledr(3) <= '1';
542
543 WHEN TRANSITION_FROM_TRAILING =>
544 display_number <= trailing_number;
545 ledr(7 DOWNTO 0) <= (OTHERS => '0');
546 ledr(2) <= '1';
547
548 WHEN EQUAL =>
549 display_number <= first_number;
550 ledr(7 DOWNTO 0) <= (OTHERS => '0');
551 ledr(1) <= '1';
552
553 WHEN ERROR =>
554 ledr(7 DOWNTO 0) <= (OTHERS => '0');
555 ledr(0) <= '1';
556
557 END CASE;
558 END PROCESS;
559
560 -- Instanciação da entidade que faz a conversão de um número binário para
561 -- codificação binária decimal (BCD) utilizando o algoritmo double dabble.
562 bin2bcd: binary_to_bcd PORT MAP(
563 binary => display_number,
564 bcd_uni => bcd(3 DOWNTO 0),
565 bcd_ten => bcd(7 DOWNTO 4),
566 bcd_hun => bcd(11 DOWNTO 8),
567 bcd_tho => bcd(15 DOWNTO 12)
```

```
568 );
569
570 -- Instanciação da entidade responsável pela conversão de um código binário
571 -- para representação no primeiro display de 7 segmentos.
572 hexseg0: conv_7seg PORT MAP(
573 bcd(3 DOWNTO 0), f0
574 );
575
576 -- Instanciação da entidade responsável pela conversão de um código binário
577 -- para representação no segundo display de 7 segmentos.
578 hexseg1: conv_7seg PORT MAP(
579 bcd(7 DOWNTO 4), f1
580 );
581
582 -- Instanciação da entidade responsável pela conversão de um código binário
583 -- para representação no terceiro display de 7 segmentos.
584 hexseg2: conv_7seg PORT MAP(
585 bcd(11 DOWNTO 8), f2
586 );
587
588 -- Instanciação da entidade responsável pela conversão de um código binário
589 -- para representação no quarto display de 7 segmentos.
590 hexseg3: conv_7seg PORT MAP(
591 bcd(15 DOWNTO 12), f3
592 );
593
594 kbd_ctrl : kbdex_ctrl generic map(24000) PORT MAP(
595 ps2_dat, ps2_clk, clock_24(0), key(1), resetn, lights(1) & lights(2) & lights(0),
596 key_on, key_code(15 DOWNTO 0) => key0
597 );
598 -- Saída do LEDR9 indicando se o número nos displays de 7 segmentos é
599 -- negativo (ligado) ou não (desligado).
600 ledr(9) <= '1' WHEN display_number(N-1) = '1' ELSE
601 '0';
602 END struct;
```

A.2 Biblioteca numérica

O arquivo que descreve os algoritmos implementados para a realização das operações aritméticas, `/lib/numeric/numeric.vhd`, é exibido.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.std_logic_signed.ALL;
4  USE ieee.numeric_std.ALL;
5
6  -- Pacote que contém as constantes e funções numéricas utilizadas pela calculadora.
7  PACKAGE numeric IS
8
9  -- Define a quantidade máxima de bits para os números da calculadora.
10 CONSTANT N : INTEGER := 13;
```

```
11
12  -- Constantes utilizadas para a representação dos operadores e comandos
13  -- possíveis da calculadora.
14  CONSTANT SUM         : STD_LOGIC_VECTOR(4 DOWNTO 0) := "10001";
15  CONSTANT SUB         : STD_LOGIC_VECTOR(4 DOWNTO 0) := "10010";
16  CONSTANT MUL         : STD_LOGIC_VECTOR(4 DOWNTO 0) := "11001";
17  CONSTANT DIV         : STD_LOGIC_VECTOR(4 DOWNTO 0) := "11010";
18  CONSTANT ENTER : STD_LOGIC_VECTOR(4 DOWNTO 0) := "11101";
19  CONSTANT RES         : STD_LOGIC_VECTOR(4 DOWNTO 0) := "11100"; -- RESET
20
21  -- Constante utilizada para representar o valor zero de N bits em binário.
22  CONSTANT ZERO : STD_LOGIC_VECTOR(N-1 DOWNTO 0) := (OTHERS => '0');
23
24  -- Constantes com o valor máximo e mínimo operado pela calculadora.
25  CONSTANT MAX_VALUE : INTEGER :=         4095; -- 0111111111111
26  CONSTANT MIN_VALUE : INTEGER := -4096; -- 1000000000000
27
28  -- Função que define a operação de multiplicação utilizando função da biblioteca
29  -- numeric_std
30  FUNCTION divide(a : STD_LOGIC_VECTOR; b : STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR;
31
32  -- Função que define a operação de multiplicação utilizando o algoritmo de
33  -- Booth melhorado.
34  -- Entradas: 1. Vetor de 13 bits do tipo STD_LOGIC_VECTOR representando o
35  --              multiplicando.
36  --           2. Vetor de 13 bits do tipo STD_LOGIC_VECTOR representando o
37  --              multiplicador.
38  -- Saída: Vetor de N + N - 1 bits do tipo STD_LOGIC_VECTOR com o resultado da
39  -- operação de multiplicação.
40  FUNCTION booth_multiplier(x : STD_LOGIC_VECTOR(N-1 DOWNTO 0); y : STD_LOGIC_VECTOR(N-1 DOWNTO
    ↪  0)) RETURN STD_LOGIC_VECTOR;
41
42  -- Procedimento responsável pela operação numérica da calculadora.
43  -- Entradas: 1. Vetor de N bits do tipo STD_LOGIC_VECTOR representando primeiro
44  --              operando da operação.
45  --           2. Vetor de N bits do tipo STD_LOGIC_VECTOR representando segundo
46  --              operando da operação.
47  --           3. Vetor de 5 bits do tipo STD_LOGIC_VECTOR representando um
48  --              possível operador.
49  -- Saídas: 1. Vetor de N bits do tipo STD_LOGIC_VECTOR representando o resultado
50  --            da operação.
51  --         2. Um bit do tipo STD_LOGIC indicando se ocorreu algum tipo de erro
52  --            na operação.
53  PROCEDURE evaluate(VARIABLE a : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
54  VARIABLE b : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
55  CONSTANT op : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
56  VARIABLE result : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
57  VARIABLE error : OUT STD_LOGIC);
58
59  -- Procedimento responsável pelas operações de soma ou subtração.
60  -- Entradas: 1. Vetor de N bits do tipo STD_LOGIC_VECTOR representando primeiro
61  --              operando da operação.
62  --           2. Vetor de N bits do tipo STD_LOGIC_VECTOR representando segundo
```

```
63  --              operando da operação.
64  --           3. Um bit indicando o tipo de operação (1 = soma, 0 = subtração)
65  -- Saídas: 1. Vetor de N bits do tipo STD_LOGIC_VECTOR representando o resultado
66  --            da operação.
67  --                                2. Um bit do tipo STD_LOGIC indicando se ocorreu overflow na
    ↪  operação.
68  PROCEDURE ripple_adder_subtractor(VARIABLE a : IN STD_LOGIC_VECTOR;
69  VARIABLE b : IN STD_LOGIC_VECTOR;
70  CONSTANT add_sub : IN STD_LOGIC := '1';
71  VARIABLE y : OUT STD_LOGIC_VECTOR;
72  VARIABLE overflow : OUT STD_LOGIC);
73
74  -- Procedimento responsável pela operação de soma ou subtração de apenas um bit.
75  -- Entradas: 1. Um bit do tipo STD_LOGIC representando o primeiro operando.
76  --           2. Um bit do tipo STD_LOGIC representando o segundo operando.
77  --           3. Um bit do tipo STD_LOGIC representando o valor de entrada do
78  --              carry.
79  --           4. Um bit indicando o tipo de operação (1 = soma, 0 = subtração)
80  -- Saídas: 1. Um bit do tipo STD_LOGIC representando o resultado da operação.
81  --         2. Um bit do tipo STD_LOGIC representando o valor de saída do
82  --              carry.
83  PROCEDURE bit_adder_subtractor(CONSTANT a : IN STD_LOGIC;
84  CONSTANT b : IN STD_LOGIC;
85  CONSTANT cin : IN STD_LOGIC;
86  CONSTANT add_sub : IN STD_LOGIC;
87  VARIABLE y : OUT STD_LOGIC;
88  VARIABLE cout : OUT STD_LOGIC);
89
90  END numeric;
91
92  PACKAGE BODY numeric IS
93
94  PROCEDURE evaluate(VARIABLE a : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
95  VARIABLE b : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
96  CONSTANT op : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
97  VARIABLE result : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
98  VARIABLE error : OUT STD_LOGIC) IS
99
100 -- Variáveis temporárias utilizadas para armazenar os valores
101 -- de resultados de operações intermediária
102 VARIABLE localresult : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
103 VARIABLE tempresult : STD_LOGIC_VECTOR(N+N-1 DOWNTO 0);
104 -- Variável utilizada para indicar a ocorrência de overflow
105 VARIABLE overflow : STD_LOGIC;
106
107 BEGIN
108 -- Inicializa error como zero (sem ocorrência de erro)
109 error := '0';
110 -- Verifica se operação é de soma.
111 IF (op = SUM) THEN
112
113 ripple_adder_subtractor(a, b, '1', localresult, overflow);
114
```

```
115 IF (overflow = '1') THEN
116 error := '1';
117 END IF;
118 result := localresult;
119
120 -- Verifica se operação é de subtração.
121 ELSIF (op = SUB) THEN
122
123 ripple_adder_subtractor(a, b, '0', localresult, overflow);
124
125 IF (overflow = '1') THEN
126 error := '1';
127 END IF;
128 result := localresult;
129
130 -- Verifica se operação é de multiplicação.
131 ELSIF (op = MUL) THEN
132 tempresult := booth_multiplier(a, b);
133
134 IF (SIGNED(tempresult) > MAX_VALUE) THEN
135 error := '1';
136 ELSIF (SIGNED(tempresult) < MIN_VALUE) THEN
137 error := '1';
138 END IF;
139 result := STD_LOGIC_VECTOR(RESIZE(SIGNED(tempresult), result'LENGTH));
140
141 -- Verifica se operação é de divisão.
142 ELSIF (op = DIV) THEN
143 IF (b = ZERO) THEN
144 -- Retorna erro caso o divisor seja zero
145 error := '1';
146 ELSE
147 result := divide(a,b);
148 END IF;
149 ELSE
150 result := ZERO;
151 error := '1';
152 END IF;
153 END evaluate;
154
155 PROCEDURE bit_adder_subtractor(CONSTANT a : IN STD_LOGIC;
156 CONSTANT b : IN STD_LOGIC;
157 CONSTANT cin : IN STD_LOGIC;
158 CONSTANT add_sub : IN STD_LOGIC;
159 VARIABLE y : OUT STD_LOGIC;
160 VARIABLE cout : OUT STD_LOGIC) IS
161 VARIABLE b_sig : STD_LOGIC;
162
163 BEGIN
164 IF (add_sub = '0') THEN
165 b_sig := NOT b;
166 ELSE
167 b_sig := b;
```

```
168  END IF;
169
170  y := a XOR b_sig XOR cin;
171
172  cout := (a AND b_sig) OR
173  (a AND cin) OR
174  (b_sig AND cin);
175
176  END bit_adder_subtractor;
177
178  PROCEDURE ripple_adder_subtractor(VARIABLE a : IN STD_LOGIC_VECTOR;
179  VARIABLE b : IN STD_LOGIC_VECTOR;
180  CONSTANT add_sub : IN STD_LOGIC := '1';
181  VARIABLE y : OUT STD_LOGIC_VECTOR;
182  VARIABLE overflow : OUT STD_LOGIC) IS
183  VARIABLE carry : STD_LOGIC_VECTOR(a'RANGE);
184  VARIABLE temp_result : STD_LOGIC_VECTOR(a'RANGE);
185  BEGIN
186
187  bit_adder_subtractor(a(0), b(0), NOT add_sub, add_sub, temp_result(0), carry(0));
188
189  FOR i IN 1 TO a'LENGTH -1 LOOP
190  bit_adder_subtractor(a(i), b(i), carry(i-1), add_sub, temp_result(i), carry(i));
191  END LOOP;
192
193  overflow := carry(a'LENGTH-1) XOR carry(a'LENGTH-2);
194
195  y := temp_result;
196
197  END ripple_adder_subtractor;
198
199  FUNCTION booth_multiplier(x : STD_LOGIC_VECTOR(N-1 DOWNTO 0); y : STD_LOGIC_VECTOR(N-1 DOWNTO
     ↪  0)) RETURN STD_LOGIC_VECTOR IS
200
201  VARIABLE result : STD_LOGIC_VECTOR(N + N - 1 DOWNTO 0);
202  VARIABLE a, s, p : STD_LOGIC_VECTOR(N + N + 1 DOWNTO 0);
203  VARIABLE nx : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
204  VARIABLE overflow : STD_LOGIC;
205  CONSTANT z : STD_LOGIC_VECTOR(N-1 DOWNTO 0) := (OTHERS => '0');
206  BEGIN
207
208  a := (OTHERS => '0');
209  s := (OTHERS => '0');
210  P := (OTHERS => '0');
211
212  a(N+N DOWNTO N+1) := x;
213  a(N+N+1) := x(N-1);
214
215  nx := (NOT x) + 1;
216  s(N+N DOWNTO N+1) := nx;
217  s(N+N+1) := NOT(x(N-1));
218
219  -- Somente executa o algoritmo de Booth se o multiplicador e o multiplicando
```

```
220  -- são diferentes de zero.
221  IF (x /= z AND y /= z) THEN
222  p(N DOWNTO 1) := y;
223  FOR i IN 0 TO N-1 LOOP
224  IF (p(1 DOWNTO 0) = "01") THEN
225  ripple_adder_subtractor(p, a, '1', p, overflow);
226  ELSIF (p(1 DOWNTO 0) = "10") THEN
227  ripple_adder_subtractor(p, s, '1', p, overflow);
228  END IF;
229  p(N+N DOWNTO 0) := p(N+N+1 DOWNTO 1);
230  END LOOP;
231  END IF;
232
233  result := p(N+N DOWNTO 1);
234  RETURN result;
235  END booth_multiplier;
236
237  FUNCTION divide(a : STD_LOGIC_VECTOR; b : STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR IS
238  BEGIN
239  RETURN STD_LOGIC_VECTOR(SIGNED(a) / SIGNED(b));
240  END divide;
241
242  END numeric;
```

A.3 Biblioteca de I/O

O pacote de I/O, contido no arquivo `/lib/io/io.vhd`, é exibido.

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   USE ieee.std_logic_signed.ALL;
4   USE ieee.numeric_std.ALL;
5   USE lib.numeric.All;
6
7   LIBRARY lib;
8   USE lib.io.ALL;
9   -- Pacote com as funções e componentes relacionadas às operações de entrada e
10  -- saída da calculadora.
11  PACKAGE io IS
12
13  -- Constante utilizada para definir um valor de entrada que não
14  -- seja utilizado pela calculadora.
15  CONSTANT IGNORED : STD_LOGIC_VECTOR(4 DOWNTO 0) := "11111";
16
17  -- Função que verifica se o código de entrada representa um valor numérico.
18  -- Entrada: um vetor de 5 bits do tipo STD_LOGIC_VECTOR representando um
19  -- código de entrada da calculadora.
20  -- Saída: um valor do tipo BOOLEAN que representa se o valor de entrada é
21  -- numérico ou não.
22  FUNCTION is_number(calc_input : STD_LOGIC_VECTOR(4 DOWNTO 0)) RETURN BOOLEAN;
23
```

```
24  -- Função que verifica se o código de entrada representa um possível código de
25  -- operador (+, -, * ou /) da calculadora.
26  -- Entrada: um vetor de 5 bits do tipo STD_LOGIC_VECTOR representando um
27  -- código de entrada da calculadora.
28  -- Saída: um valor do tipo BOOLEAN que representa se o valor de entrada é
29  -- um operador ou não.
30  FUNCTION is_operation(calc_input : STD_LOGIC_VECTOR(4 DOWNTO 0)) RETURN BOOLEAN;
31
32  -- Função que verifica se o código de entrada representa um operador de soma ou
33  -- subtração.
34  -- Entrada: um vetor de 5 bits do tipo STD_LOGIC_VECTOR representando um
35  -- código de entrada da calculadora.
36  -- Saída: um valor do tipo BOOLEAN que representa se o valor de entrada é
37  -- é um operador simples (soma ou subtração) ou não.
38  FUNCTION is_simple_operation(calc_input : STD_LOGIC_VECTOR(4 DOWNTO 0)) RETURN BOOLEAN;
39
40  -- Função que verifica se o código de entrada representa um operador de
41  -- multiplicação ou divisão.
42  -- Entrada: um vetor de 5 bits do tipo STD_LOGIC_VECTOR representando um
43  -- código de entrada da calculadora.
44  -- Saída: um valor do tipo BOOLEAN que representa se o valor de entrada é
45  -- é um operador complexo (multiplicação ou divisão) ou não.
46  FUNCTION is_complex_operation(calc_input : STD_LOGIC_VECTOR(4 DOWNTO 0)) RETURN BOOLEAN;
47
48  -- Função que retorna um valor para o display deslocando o valor do display
49  -- atual para uma casa decimal a esquerda e adicionando um valor de entrada
50  -- númerico (de 0 a 9) na primeira casa decimal.
51  -- Entrada: 1. um vetor de N bits do tipo STD_LOGIC_VECTOR que representa o
52  --             valor numério atual mostrado nos displays de 7 segmentos.
53  --          2. um vetor de 4 bits do tipo STD_LOGIC_VECTOR representando um
54  --             valor numérico entre 0 e 9 que será adicionado a primeira casa
55  --             decimal do valor do display atual.
56  -- Saída: um vetor de N bits com o valor mostrado nos displays de 7 segmentos
57  -- com uma nova casa decimal.
58  FUNCTION get_resulting_display(display : STD_LOGIC_VECTOR(N-1 DOWNTO 0); input :
    ↪  STD_LOGIC_VECTOR(3 DOWNTO 0)) RETURN STD_LOGIC_VECTOR;
59
60  -- Componente que interpreta e transmite os pacotes com os comandos utilizados
61  -- pelo teclado.
62  -- Sinais bidirecionais:
63  --                ps2_data: corresponde aos bits transferidos serialmente para e da porta PS2.
64  --          ps2_clock: é o clock de funcionamento do controlador PS2.
65  -- Entradas:
66  --          clk: corresponde ao clock do sistema. Deve ter a mesma frequência atribuída
67  --          ao clkfreq, 24000 (kHz).
68  --          en: sinal de habilitação (ativo em nível baixo.
69  --          resetn: sinal de reinicialização (ativo em nível baixo).
70  --          lights: é um vetor que determina o estado dos LEDs do teclado: light(0) é o
71  --                  do scroll lock, light(1) é o do nunlock e o lights(2) o do capslock.
72  -- Saídas:
73  --          key_on: é um vetor que indica se temos ou não teclas pressionadas. O índice 0
74  --                  representa a primeira tecla pressionada, o 1 a segunda e o 2 a terceira.
75  --          Key_code: é o vetor que indica o código de leitura (scancode) das teclas
```

```
76  --                                  pressionadas. Os bits 15-0 representam a primeira tecla
    ↪  pressionada;
77  --                                  os bits 31-16 representam a segunda tecla pressionada; e os
    ↪  bits 47-32
78  --                                  representam a segunda tecla pressionada.
79  COMPONENT kbdex_ctrl
80  GENERIC(
81  clkfreq : INTEGER
82  );
83  PORT(
84  ps2_data                :        INOUT        STD_LOGIC;
85  ps2_clk                 :        INOUT        STD_LOGIC;
86  clk                     :        IN           STD_LOGIC;
87  en                      :        IN           STD_LOGIC;
88  resetn                  :        IN           STD_LOGIC;
89  lights                  :        IN           STD_LOGIC_VECTOR(2 DOWNTO 0); -- lights(Caps, Nun,
    ↪  Scroll)
90  key_on                  :        OUT          STD_LOGIC_VECTOR(2 DOWNTO 0);
91  key_code                :        OUT          STD_LOGIC_VECTOR(47 DOWNTO 0)
92  );
93  END COMPONENT;
94  END io;
95
96  PACKAGE BODY io IS
97  FUNCTION is_number(calc_input : STD_LOGIC_VECTOR(4 DOWNTO 0)) RETURN BOOLEAN IS
98  BEGIN
99  RETURN calc_input(4) = '0';
100 END is_number;
101
102 FUNCTION is_operation(calc_input : STD_LOGIC_VECTOR(4 DOWNTO 0)) RETURN BOOLEAN IS
103 BEGIN
104 RETURN calc_input(4) = '1' AND calc_input /= IGNORED;
105 END is_operation;
106
107 FUNCTION is_simple_operation(calc_input : STD_LOGIC_VECTOR(4 DOWNTO 0)) RETURN BOOLEAN IS
108 BEGIN
109 RETURN calc_input(4 DOWNTO 3) = "10";
110 END is_simple_operation;
111
112 FUNCTION is_complex_operation(calc_input : STD_LOGIC_VECTOR(4 DOWNTO 0)) RETURN BOOLEAN IS
113 BEGIN
114 RETURN calc_input(4 DOWNTO 3) = "11" AND calc_input /= IGNORED;
115 END is_complex_operation;
116
117 FUNCTION get_resulting_display(display : STD_LOGIC_VECTOR(N-1 DOWNTO 0); input :
    ↪  STD_LOGIC_VECTOR(3 DOWNTO 0)) RETURN STD_LOGIC_VECTOR IS
118 -- Variável temporária utilizada para armazenar o resultado da multiplicação
119 -- do valor atual do display multiplicado por 10.
120 VARIABLE lr : STD_LOGIC_VECTOR(N+N-1 DOWNTO 0);
121
122 -- Constante com o valor decimal 10 em binário.
123 CONSTANT TEN : STD_LOGIC_VECTOR(N-1 DOWNTO 0) := STD_LOGIC_VECTOR(TO_UNSIGNED(10,N));
124
```

```
125  -- Variável utilizada para armazenar se o resultado da soma excedeu o valor
126  -- máximo possível.
127  VARIABLE overflow : STD_LOGIC;
128
129  -- Variáveis temporárias utilizadas para armazenar os valores do número atual
130  -- do display multiplicado por 10 e o valor de entrada ambas com o tamanho de
131  -- N bits.
132  VARIABLE a, b : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
133
134  VARIABLE display_out : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
135  BEGIN
136  -- Multiplica o valor atual do display por 10 para deslocar o valor do display
137  -- para uma casa decimal a esquerda.
138  lr :=  booth_multiplier(TEN, display);
139
140  -- Transforma o valor do resultado anterior e o número de entrada para o
141  -- tamanho de N bits.
142  a := (OTHERS => '0');
143  a := lr(N-1 DOWNTO 0);
144  --a := STD_LOGIC_VECTOR(RESIZE(UNSIGNED(lr), N));
145
146  b := (OTHERS => '0');
147  b(3 DOWNTO 0) := input;
148  --b := STD_LOGIC_VECTOR(RESIZE(UNSIGNED(input), N));
149
150  -- Verifica se o valor do display deslocado uma casa decimal a esquerda
151  -- excede o valor máximo de 13 bits de um número binário de complemento de 2.
152  IF (lr <= MAX_VALUE) THEN
153
154  -- Adiciona o valor decimal de 0 a 9 a primeira casa decimal do número do
155  -- display deslocado uma casa decimal a esquerda.
156  ripple_adder_subtractor(a, b,        '1', display_out, overflow);
157
158  -- Verifica se o valor do display deslocado uma casa decimal a esquerda
159  -- e adicionado um dígito na primeira casa decimal excede o valor máximo
160  -- de 13 bits de um número binário de complemento de 2. Caso o valor
161  -- exceda o limite máximo é retornado o mesmo número binário de entrada.
162  IF (overflow = '1') THEN
163  RETURN display;
164  ELSE
165  RETURN display_out;
166  END IF;
167
168  ELSE
169  RETURN display;
170  END IF;
171
172  END get_resulting_display;
173
174  END io;
```

O pacote define as funções que realizam a leitura e interpretação da entrada do teclado *PS2*:

- `is_number()`: Verifica se a entrada é numérica, quando o *bit* mais significativo é 0;
- `is_operation()`: Verifica se a entrada é um operador, quando o *bit* mais significativo é 1;
- `is_simple_operation()`: Verifica se a entrada é um operador de soma ou subtração, quando os dois *bits* mais significativos da entrada são 10;
- `is_complex_operation()`: Verifica se a entrada é um operador de multiplicação ou divisão, quando os dois *bits* mais significativos da entrada são 11;
- `get_resulting_display()`: Função responsável por atualizar o *display*.

A.4 Bibliotecas genéricas

Os arquivos da pasta `/lib/general/` descrevem funções auxiliares para a exibição de valores em *display* de sete segmentos, conversão de números binários para o formato *BCD*, conversão da leitura serial do teclado para um código interpretável numericamente, e a definição do funcionamento do *PWM* para controle de brilho do *display*. Estes códigos são descritos individualmente à seguir.

A.4.1 Conversor binário para BCD

O código para conversão de binário para *BCD*, contido no arquivo `binary_to_bcd.vhd` é exibido.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  --USE ieee.std_logic_signed.ALL;
4  USE ieee.std_logic_unsigned.ALL;
5  USE ieee.numeric_std.ALL;
6
7  -- Conversão de um número binário para codificação binária decimal (BCD)
8  -- utilizando o algoritmo double dabble.
9  -- Entrada: um número binário de 13 bits do tipo STD_LOGIC_VECTOR no formato de
10 -- complemento de 2.
11 -- Saídas: quatro números binários de 4 bits do tipo STD_LOGIC_VECTOR.
12 -- Cada número corresponde a uma casa decimal distinta. O número de saída no
13 -- formato BCD é seu valor absoluto. Caso seja negativo e o número não utilize
14 -- todas as 4 casas decimais, o display de 7 segmentos da primeira casa decimal
15 -- mais a esquerda do número exibe o sinal negativo.
16 ENTITY binary_to_bcd IS
17 PORT (binary : IN STD_LOGIC_VECTOR(12 DOWNTO 0);
18 bcd_uni : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
19 bcd_ten : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
20 bcd_hun : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
21 bcd_tho : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
22 END binary_to_bcd;
23
24 ARCHITECTURE binary_to_bcd_arch OF binary_to_bcd IS
25 BEGIN
26
27 PROCESS(binary) IS
28
```

```
29  VARIABLE bcd : STD_LOGIC_VECTOR(15 DOWNTO 0);
30  VARIABLE abs_num : STD_LOGIC_VECTOR(12 DOWNTO 0);
31
32  -- Constantes definidas para representar o valor zero (ZERO), um display
33  -- apagado (BLANK) e o símbolo negativo (NEG_SIGN).
34  CONSTANT ZERO : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0000";
35  CONSTANT BLANK : STD_LOGIC_VECTOR(3 DOWNTO 0) := "1111";
36  CONSTANT NEG_SIGN : STD_LOGIC_VECTOR(3 DOWNTO 0) := "1110";
37
38  BEGIN
39  -- Se o número binário de entrada for negativo, obtem-se o valor absoluto
40  -- utilizando o operador NOT e somando mais um no resultado. Considera-se
41  -- aqui que o número binário esteja no formato de complemento de 2.
42  IF (binary(12) = '1') THEN
43  abs_num := STD_LOGIC_VECTOR(UNSIGNED((NOT binary) + 1));
44  ELSE
45  abs_num := binary;
46  END IF;
47
48  -- Inicializa bcd com zeros em todos os bits.
49  bcd := (OTHERS => '0');
50
51  -- Laço for utilizando o algoritmo double dabble.
52  FOR i IN 0 TO 12 LOOP
53
54  IF (bcd(3 DOWNTO 0) > 4) THEN
55  bcd(3 DOWNTO 0) := bcd(3 DOWNTO 0) + 3;
56  END IF;
57
58  IF (bcd(7 DOWNTO 4) > 4) THEN
59  bcd(7 DOWNTO 4) := bcd(7 DOWNTO 4) + 3;
60  END IF;
61
62  IF (bcd(11 DOWNTO 8) > 4) THEN
63  bcd(11 DOWNTO 8) := bcd(11 DOWNTO 8) + 3;
64  END IF;
65
66  -- Desloca para esquerda.
67  bcd := bcd(14 DOWNTO 0) & abs_num(12-i);
68
69  END LOOP;
70
71  -- Verifica se as últimas casas decimais do número são iguais a zero.
72  -- Caso isso ocorra, as casas decimais que não possuem valor são definidas
73  -- como BLANK, fazendo com que o display de 7 segmento não mostre valor algum.
74  IF (bcd(15 DOWNTO 12) = ZERO AND bcd(11 DOWNTO 8) = ZERO AND bcd(7 DOWNTO 4) = ZERO) THEN
75  bcd(15 DOWNTO 12) := BLANK;
76  bcd(11 DOWNTO 8) := BLANK;
77
78  IF (binary(12) = '1') THEN
79  bcd(7 DOWNTO 4) := NEG_SIGN;
80  ELSE
81  bcd(7 DOWNTO 4) := BLANK;
```

```
82  END IF;
83
84  ELSIF (bcd(15 DOWNTO 12) = ZERO AND bcd(11 DOWNTO 8) = ZERO AND bcd(7 DOWNTO 4) /= ZERO) THEN
85  bcd(15 DOWNTO 12) := BLANK;
86
87  IF (binary(12) = '1') THEN
88  bcd(11 DOWNTO 8) := NEG_SIGN;
89  ELSE
90  bcd(11 DOWNTO 8) := BLANK;
91  END IF;
92
93  ELSIF (bcd(15 DOWNTO 12) = ZERO AND bcd(11 DOWNTO 8) /= ZERO) THEN
94
95  IF (binary(12) = '1') THEN
96  bcd(15 DOWNTO 12) := NEG_SIGN;
97  ELSE
98  bcd(15 DOWNTO 12) := BLANK;
99  END IF;
100
101 END IF;
102
103 bcd_uni <= bcd(3 DOWNTO 0);
104 bcd_ten <= bcd(7 DOWNTO 4);
105 bcd_hun <= bcd(11 DOWNTO 8);
106 bcd_tho <= bcd(15 DOWNTO 12);
107
108 END PROCESS;
109
110 END binary_to_bcd_arch;
```

A.4.2 Conversor binário para display de sete segmentos

O conversor de binário para *display* de sete segmentos, descrito pelo arquivo `conv_7seg.vhd`, simplesmente codifica um algarismo, sinal negativo, ou valor nulo, de tal forma a representá-lo nos *dislplays* do *kit*. O código é exibido à seguir.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;
3
4   -- Entidade responsável pela conversão de um código binário para representação
5   -- no display de 7 segmentos. Os valores de bit "1110" e "1111" são reservados,
6   -- respectivamente, para a representação do símbolo negativo e quando o display
7   -- está apagado.
8   -- Entrada: um valor binário de 4 bits do tipos STD_LOGIC_VECTOR.
9   -- Saída: um vetor de 7 bits do tipo STD_LOGIC_VECTOR com as informações para um
10  -- display de 7 segmentos.
11  ENTITY conv_7seg IS
12  port( digit        : in STD_LOGIC_VECTOR (3 downto 0);
13  seg        : out STD_LOGIC_VECTOR (6 downto 0));
14  END conv_7seg;
15
```

```
16 ARCHITECTURE structural OF conv_7seg IS
17 BEGIN
18 WITH digit SELECT
19 seg <=  "1000000" WHEN "0000", -- 0
20         "1111001" WHEN "0001", -- 1
21         "0100100" WHEN "0010", -- 2
22         "0110000" WHEN "0011", -- 3
23         "0011001" WHEN "0100", -- 4
24         "0010010" WHEN "0101", -- 5
25         "0000010" WHEN "0110", -- 6
26         "1011000" WHEN "0111", -- 7
27         "0000000" WHEN "1000", -- 8
28         "0010000" WHEN "1001", -- 9
29         "0111111" WHEN "1110", -- NEG SIGN
30         "1111111" WHEN "1111", -- BLANK
31         "1111111" WHEN OTHERS;
32 END structural;
```

A.4.3 Modulador PWM

Assim como o *Double-Dabble*, o algoritmo para o modulador *PWM*, descrito no arquivo `pwm.vhd`, foi implementado em aula prática [5]. O código é exibido à seguir.

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  USE IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5  -- Entidade responsável pela geração do sinal PWM.
6  -- Entradas: 1. sinal de clock
7  --           2. sinal que indica se está o PWM habilitado.
8  --           3. sinal de reset ativo em nível baixo.
9  --           4. um vetor de 8 bits do tipo STD_LOGIC_VECTOR utilizado para o
10 --              calculo do duty-cycle
11 -- Saída: um bit do tipo STD_LOGIC que representa se o sinal PWM está ativo ou não.
12 ENTITY pwm IS
13 PORT (clk, enable, rstn : IN STD_LOGIC;
14 duty : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
15 pwm_out : OUT STD_LOGIC);
16 END pwm;
17
18 ARCHITECTURE pwm_arch OF pwm IS
19
20 SIGNAL count : STD_LOGIC_VECTOR(7 DOWNTO 0);
21
22 BEGIN
23 PROCESS(rstn, clk, enable) IS
24 BEGIN
25 IF (rstn = '0') THEN
26 count <= "00000000";
27 pwm_out <= '0';
28 ELSIF (enable = '1') THEN
```

```
IF (clk'EVENT AND clk = '1') THEN
count <= count + 1;
IF (count <= duty) THEN
pwm_out <= '1';
ELSIF (count = "11111111") THEN
count <= "00000000";
ELSE
pwm_out <= '0';
END IF;
END IF;
END IF;
END PROCESS;
END pwm_arch;
```

A.4.4 Conversor da entrada do teclado para formato numérico

O código do arquivo `scancode_to_calc_input.vhd` decodifica a entrada serial do teclado de forma a tornar esta entrada "legível" aos códigos que determinam o funcionamento da calculadora. Isso é feito através de uma correspondência direta, e utiliza a tabela de *scancodes* de uma das aulas práticas da disciplina [4]. O código é exibido à seguir.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

-- Entidade responsável pela conversão dos scancodes do teclado para um formato
-- interno utilizado pela calculadora.
-- Entrada: um vetor de 8 bits do tipo STD_LOGIC_VECTOR representando o valor
-- scancode do teclado.
-- Saída: um vetor de 4 bits do tipo STD_LOGIC_VECTOR representando os
-- números de 0 a 9, as teclas +, -, *, /, enter e esc. O valor "11111"
-- representa um valor qualquer que não seja utilizado pela calculadora.
ENTITY scancode_to_calc_input IS
PORT(scancode : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
calc_input : OUT STD_LOGIC_VECTOR(4 DOWNTO 0));
END scancode_to_calc_input;

ARCHITECTURE scancode_to_calc_input_arch OF scancode_to_calc_input IS
BEGIN
WITH scancode SELECT
calc_input <=   "00000" WHEN x"70" | x"45", -- "01110000", -- 0        70
                "00001" WHEN x"69" | x"16", -- "01101001", -- 1        69
                "00010" WHEN x"72" | x"1E", -- "01110010", -- 2        72
                "00011" WHEN x"7A" | x"26", -- "01111010", -- 3        7A
                "00100" WHEN x"6B" | x"25", -- "01101011", -- 4        6B
                "00101" WHEN x"73" | x"2E", -- "01110011", -- 5        73
                "00110" WHEN x"74" | x"36", -- "01110100", -- 6        74
                "00111" WHEN x"6C" | x"3D", -- "01101100", -- 7        6C
                "01000" WHEN x"75" | x"3E",  -- "01110101", -- 8        75
                "01001" WHEN x"7D" | x"46", -- "01111101", -- 9        7D
                "10001" WHEN x"79", -- "01111001", -- +        79
                "10010" WHEN x"7B", -- "01111011", -- -        7B
```

```
31                  "11001" WHEN x"7C", -- "01111100", -- *         7C
32                  "11010" WHEN x"4A", -- "01001010", -- /         4A
33                  "11101" WHEN x"5A", -- ENTER "01011010", -- En         5A
34                  "11100" WHEN x"76", -- ESC
35                  "11111" WHEN OTHERS; -- INVALID
36  END scancode_to_calc_input_arch;
```