O módulo *Keyboard Reader* é constituído por três blocos principais: *i*) o descodificador de teclado (*Key Decode*); *ii*) o bloco de armazenamento (designado por *Ring Buffer*); e *iii*) o bloco de entrega ao consumidor (designado por *Output Buffer*). Neste caso o módulo *Control*, implementado em *software*, é a entidade consumidora.

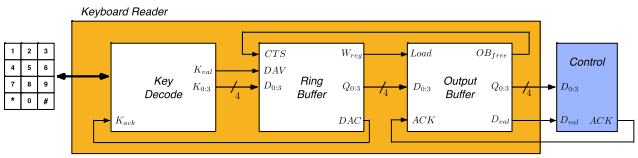
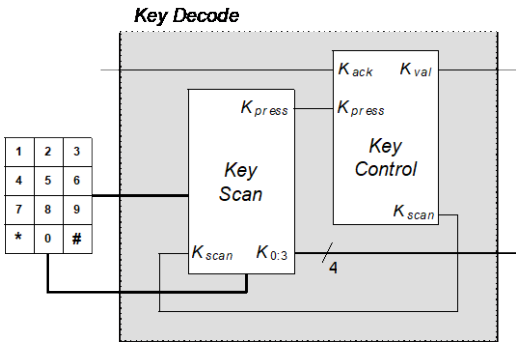


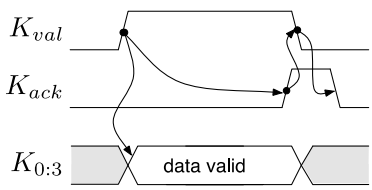
Figura 1 – Diagrama de blocos do módulo *Keyboard Reader*

1 Key Decode

O bloco *Key Decode* implementa um descodificador de um teclado matricial 4x3 por *hardware*, sendo constituído por três sub-blocos: *i)* um teclado matricial de 4x3; *ii)* o bloco *Key Scan,* responsável pelo varrimento do teclado; e *iii)* o bloco *Key Control*, que realiza o controlo do varrimento e o controlo de fluxo, conforme o diagrama de blocos representado na Figura 2a. O controlo de fluxo de saída do bloco *Key Decode* (para o módulo *Key Buffer*), define que o sinal K_{val} é ativado quando é detetada a pressão de uma tecla, sendo também disponibilizado o código dessa tecla no barramento $K_{0:3}$. Apenas é iniciado um novo ciclo de varrimento ao teclado quando o sinal K_{ack} for ativado e a tecla premida for libertada. O diagrama temporal do controlo de fluxo está representado na Figura 2b.



a) Diagrama de blocos



b) Diagrama temporal

Figura 2 – Bloco *Key Decode*

O bloco *Key Scan* foi implementado de acordo com o diagrama de blocos representado na Figura 3. Na implementação do bloco *Key Scan* optámos pelo diagrama de blocos *versão 1*, uma vez que na *versão 2* o *Kpress* fica ativo, se uma tecla for premida, independentemente da linha que esteja a ser observada.

Na versão 1, de acordo com a figura 3 este é constituído por um decoder (2 por 3), um mutiplexer (4 por 1) e um contador de 4 bits que é responsável pela seleção das saídas do decoder e das entradas do mutiplexer.

Às entradas do mutiplexer estão ligadas as linhas do teclado de forma a ativar o *Kpress* quando uma tecla na linha selecionada seja premida. Nas saídas do decoder encontram-se as colunas do teclado, de forma a selecionar uma linha. Quando uma tecla é premida a entrada enable do contador recebe o valor lógico 0, para que seja possível fazer a leitura do código da tecla premida, sendo os bits de maior peso a coluna da tecla foi premida e os bits de menor peso a linha da tecla premida.

O bloco *Key Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 4.

No estado 00 o Kscan está ativo de forma a fazer o varrimento do teclado até que uma tecla seja premida, sendo essa avaliação feita através do sinal Kpress. Quando uma tecla é premida entramos no estado 01 e o sinal Kval fica ativo indicando que foi pressionada uma tecla válida, e mantém-se neste estado até que o sinal Kack fique ativo indicando que a leitura do código da tecla está completa e foi aceite. Quando Kack ficar ativo saímos do estado e entramos no estado 10, a aguardar que o sinal Kack e o sinal Kpress estejam desativados de forma que apenas seja iniciado um novo varrimento do teclado quando o sinal Kack esteja desativado e a tecla premida for libertada.

A descrição hardware do bloco *Key Decode* em VHDL encontra-se no Anexo A.

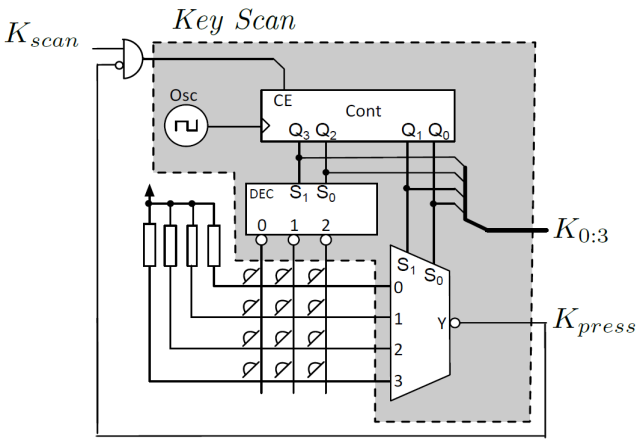


Figura 3 - Diagrama de blocos do bloco *Key Scan*

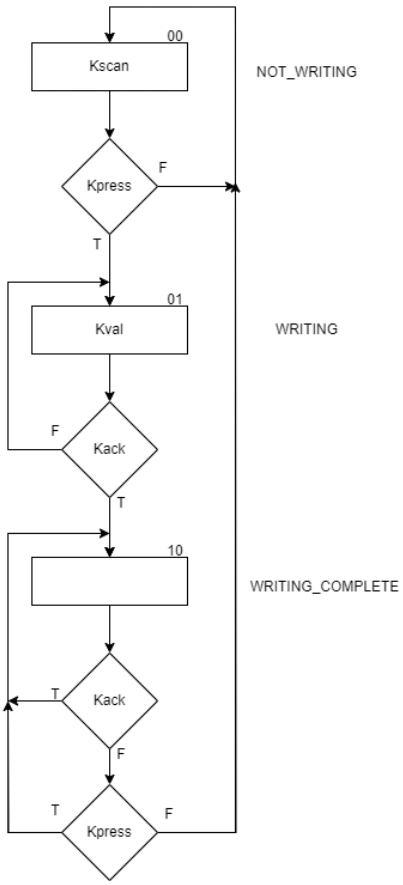


Figura 4 – Máquina de estados do bloco *Key Control*

Com base nas descrições do bloco *Key Decode* implementou-se parcialmente o módulo *Keyboard Reader* de acordo com o esquema elétrico representado no Anexo D.

2 Ring Buffer

O bloco RingBuffer é uma estrutura de dados para armazenamento de teclas com disciplina FIFO (First In First Out), com capacidade de armazenar até oito palavras de quatro bits. A escrita de dados no Ring Buffer inicia-se com a ativação do sinal DAV (Data Available) pelo sistema produtor, neste caso pelo Key Decode, indicando que tem dados para serem armazenados. Logo que tenha disponibilidade para armazenar informação, o Ring Buffer escreve os dados D0:3 em memória. Concluída a escrita em memória ativa o sinal DAC (Data Accepted) para informar o sistema produtor que os dados foram aceites. O sistema produtor mantém o sinal DAV ativo até que DAC seja ativado. O Ring Buffer só desativa DAC depois de DAV ter sido desativado .A implementação do Ring Buffer é baseada numa memória RAM (Random Access Memory). O endereço de escrita/leitura, selecionado por $put\overline{get}$ é definido pelo bloco Memory Address Control (MAC) composto por dois registos, que contêm o endereço de escrita e leitura, designados por putIndex e getIndex respetivamente. O MAC suporta assim ações de incPute incGet, gerando informação se a estrutura de dados está cheia (Full) ou se está vazia (Empty). O bloco Ring Buffer procede à entrega de dados à entidade consumidora, sempre que esta indique que está disponível para receber, através do sinal Clear To Send(CTS). Na Figura 5 é apresentado o diagrama de blocos para uma estrutura do bloco Ring Buffer.

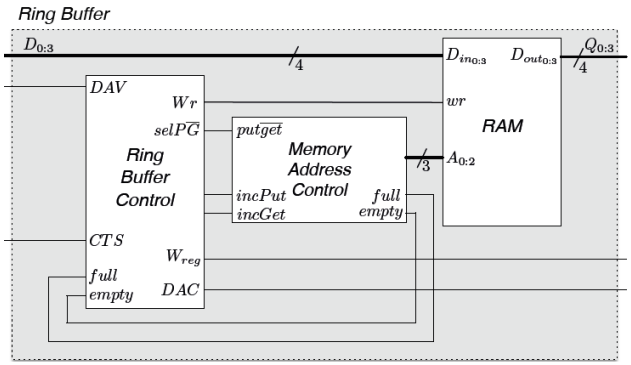


Figura 5 Diagrama de blocos do bloco Ring Buffer

No desenvolvimento do bloco Ring Buffer começou-se por desenvolver o bloco Memory Address Control de acordo com as informações fornecidas. Desta forma foi necessário recorrer a 2 contadores de 3 bits, uma vez que a RAM presente no diagrama de blocos tem 8 endereços disponíveis, para incrementar os endereços de leitura e escrita designados por putIndex e getIndex respetivamente.

Para gerar a informação dos sinais Full e Empty foi usado um contador cresecente/decrescente que opera quando o MAC recebe os sinais incput ou incGet, a contagem crescente ou decrescente é determinada pelo sinal incPut sendo crescente quando este tem o valor lógico '1' e decrescente quando tem o valor lógico '0', de forma a acompanhar o número de endereços ocupados com informação que não foi enviada para leitura.

Os endereços de leitura e escrita são fornecidos à RAM por um mutiplexer 2 por 1, onde a seleção da entrada recebe o sinal $put\overline{get}$, quando este tem o valor lógico '0' é enviado o idxGet e quando este apresenta o valor lógico '1' é enviado o idxPut.

Em baixo encontra-se o diagrama de blocos do bloco MAC.

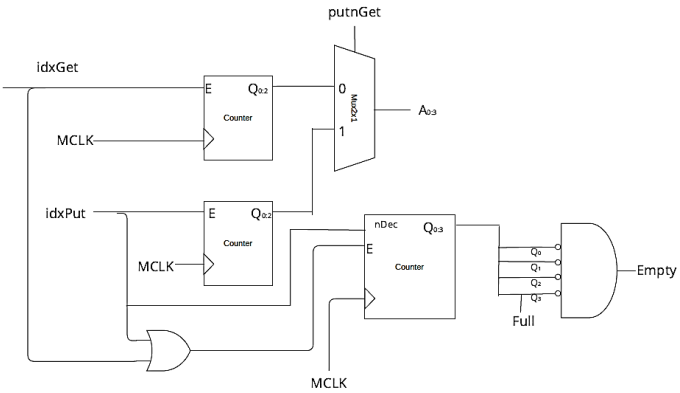


Figura 6 Diagrama de blocos do bloco MAC

O bloco *Ring Buffer Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 7.

No estado 00 aguarda-se uma instrução de escrita ou leitura, no caso da instrução recebida seja de escrita, com a ativação do sinal DAV, é verificando antes de realizar a escrita na RAM, se esta tem algum endereço disponível para armazenar dados. Se os endereços da RAM estiverem todos ocupados, entramos no estado 10 onde é realizada uma espera pela ativação do sinal CTS, de forma a realizar uma leitura da RAM, libertando espaço de forma a concluir a instrução de leitura. Quando o sinal CTS estiver ativo, ocorre a passagem para o estado 101, onde é ativado o sinal *Wreg* de forma a registar os dados lidos. Depois de enviar os dados para o bloco Output Buffer no estado 111 o sinal incGet é ativado de forma a incrementar o idxGet e verifica-se se estava a ser executada uma instrução de escrita, de forma a concluir a mesma. A instrução de escrita é realizada ativando o sinal selPG de forma a fornecer o endereço á RAM, no estado seguinte é ativado o sinal Wr de forma a escrever os dados recebidos na RAM, depois da escrita é incrementado o idxPut com a ativação do sinal incPut e a seguir passamos para o estado 110 onde o sinal DAC é ativado de forma a informar o bloco Key Decode que os dados recebidos foram aceites.

No caso de o sinal full não esteja ativo é realizada a instrução de escrita, sem realizar uma instrução de leitura.

Quando sinal DAV não estiver ativo, mas o sinal CTS tenha o valor lógico 1, verifica-se se a RAM está vazia através do sinal empty. Se a RAM não tem dados para leitura mantêm-se no estado 00, caso contrário ocorre a passagem para o estado 101 e é realizada a operação de leitura tal como descrita anteriormente.

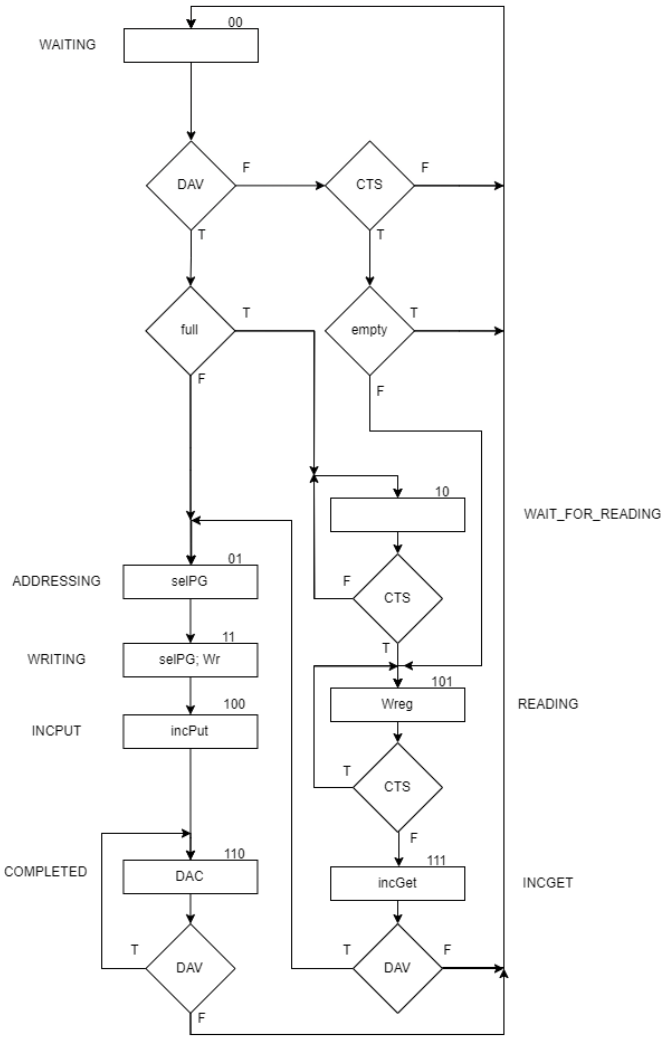


Figura 7 Máquina de estados do bloco Ring Buffer Control

A descrição hardware do bloco *Ring Buffer* em VHDL encontra-se no Anexo B.

3 Output Buffer

O bloco Output Buffer do Keyboard Reader é responsável pela interação com o sistema consumidor, neste caso o módulo Control. O Output Buffer indica que está disponível para armazenar dados através do sinal OBfree. Assim,nesta situação o sistema produtor pode ativar o sinal Load para registar os dados. O Control quando pretende ler dados do Output Buffer, aguarda que o sinal Dval fique ativo, recolhe os dados e pulsa o sinal ACK indicando que estes já foram consumidos. O Output Buffer, logo que o sinal ACK pulse, deve invalidar os dados baixando o sinal Dval e sinalizar que está novamente disponível para entregar dados ao sistema consumidor, ativando o sinal OBfree. Na Figura 8, é apresentado o diagrama de blocos do Output Buffer.

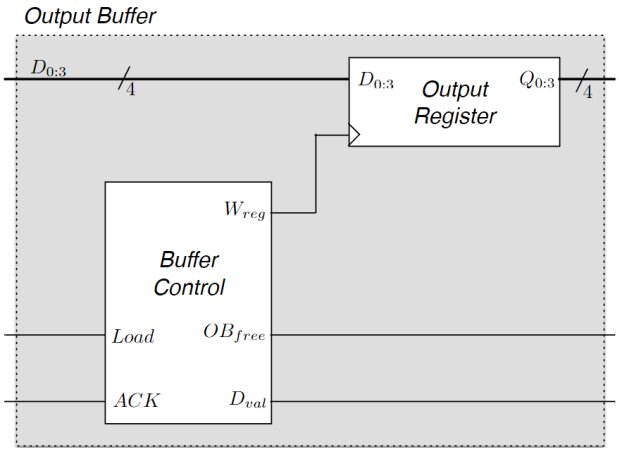


Figura 8 Diagrama de blocos do bloco Output Buffer

Sempre que o bloco emissor Ring Buffer tenha dados disponíveis e o bloco de entrega Output Buffer esteja disponível (OBfree ativo), o Ring Buffer realiza uma leitura da memória e entrega os dados ao Output Buffer ativando o sinal Wreg. O Output Buffer indica que já registou os dados desativando o sinal OBfree.

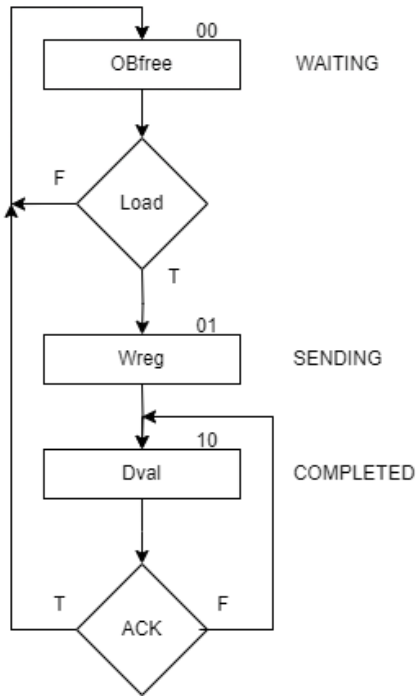


Figura 9 Máquina de estados do bloco Buffer Control

O bloco *Buffer Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 8.

A máquina de estados do bloco Buffer Control inicialmente encontra-se no estado 00, com o sinal OBfree indicando que está disponível para receber dados, quando o sinal Load tiver o valor lógico '1' (indicando que o bloco Ring Buffer tem dados para entregar) ocorre a passagem para o estado 01 onde o sinal Wreg é ativado de forma a realizar a escrita no bloco Output Register e a seguir ocorre a passagem para o estado 10 onde o sinal Dval é ativado (informando o Control de que tem dados disponíveis para recolha) e mantém-se ativo até que o sinal ACK tenha o valor lógico '1', indicando que os dados foram consumidos pelo Control. Por fim invalida os dados baixando o sinal Dval e sinalizar que está novamente disponível para entregar dados ao sistema consumidor, ativando o sinal OBfree.

A descrição hardware do bloco *Output Buffer* em VHDL encontra-se no Anexo C.

4 Interface com o *Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem *Kotlin* e seguindo a arquitetura lógica apresentada na Figura 8.

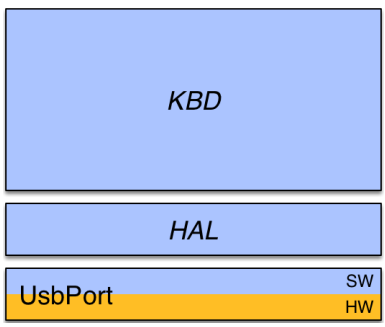


Figura 10 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

HAL e *KBD* desenvolvidos são descritos nas secções 3.1. e 3.2, e o código fonte desenvolvido nos Anexos C e D, respetivamente.

4.1 *HAL*

A classe HAL é responsável por comunicar com o bloco UsbPort, fazendo a leitura e a escrita no bloco UsbPort.
De forma a evitar efetuar a leitura do bloco UsbPort repetidamente foi criada uma variável global (written) dentro da classe para guardar o último valor escrito.
Nesta classe temos a função init que inicia a classe, a função isBit() que retorna true se o bit passado na máscara tiver o valor lógico 1. A função readBits() efetua uma leitura tal como a função isBit() mas para um conjunto de bits.
A função writeBits() escreve um valor num conjunto de bits, a função setBits() escreve nos bits da máscara o valor lógico 1. E a função clrBits() coloca o valor lógico 0 no bit indicado na máscara.

4.2 *KBD*

Esta classe utiliza a classe HAL de forma a obter do bloco UsbPort um código de uma tecla e retornar o char equivalente ao código lido.
Nesta classe temos uma lista com os Char´s correspondentes aos códigos possíveis, temos a função getKey() que retorna o char correspondente ao código da tecla premida imediatamente, quando o sinal Dval do bloco Output Buffer estiver ativo e ativa o sinal ACK do bloco Control informando que os dados foram recebidos . E por fim temos a função waitKey() que retorna o char correspondente ao código da tecla premida ao fim de um intervalo de tempo ou NONE se não for premida nenhuma tecla.

5 Conclusões

O objetivo deste módulo é descodificar a tecla pressionada e enviar o respetivo código para o Control. Para isso, utilizamos funções previamente desenvolvidas para leitura de portas, que convertem o código recebido em caracter de forma a descodificar a tecla premida. Para além das funções dos objetos KBD e HAL, foi necessário realizar um contador que efetuasse a soma com o resultado presente nas suas saídas, por essa razão foi implementado um register que coloca o valor da saída do contador numa das entradas do mesmo, tendo em conta que em uma das entradas é introduzido o valor 1. Em relação ao software, este foi testado no simulador e aparenta ter um funcionamento correto, e os módulos Key Decode, Ring Buffer e Output Buffer também foram testados, primeiro no simulador e depois na placa onde mostraram um funcionamento correto.

A. Descrição VHDL do bloco *Key Decode*

Key Scan

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY KEY_SCAN IS
   PORT( Kscan: in std_logic;
      Clr: in std_logic;
      Clk: in std_logic;
      Kpress : out std_logic;
      KEYPAD_LIN : in std_logic_vector(3 downto 0);
      KEYPAD_CODE: out std_logic_vector(3 downto 0);
      KEYPAD_COL : out std_logic_vector(3 downto 0)

      );
END KEY_SCAN;

ARCHITECTURE arq_KEY_SCAN OF KEY_SCAN IS

component DECODER2_3
   PORT( S: in STD_LOGIC_VECTOR(1 downto 0);
         A,B,C,D:  out STD_LOGIC
      );
end component;

component MUX4_1
   PORT( A,B,C,D: in STD_LOGIC;
         S: in STD_LOGIC_VECTOR(1 downto 0);
         Y:  out STD_LOGIC
      );
end component;

component COUNTER
   PORT(  CLK : in std_logic;
      E : in std_logic;
      clr: in std_logic;
      R : out std_logic_vector (3 downto 0)
      );
end component;

Signal Coluna, Linha: Std_Logic_Vector (1 downto 0);

Signal A , B : Std_Logic;

BEGIN

Kpress <=  B;
A <= Kscan and (not B); -- 1 and not 0 => 1 , 1 and 0

KEYPAD_CODE(0) <= Linha(0);
KEYPAD_CODE(1) <= Linha(1);
KEYPAD_CODE(2) <= Coluna(0);
KEYPAD_CODE(3) <= Coluna(1);

UDECODER: DECODER2_3 port map(
   S(0) => Coluna(0),
   S(1) => Coluna(1),
   A => KEYPAD_COL(0),
   B => KEYPAD_COL(1),
   C => KEYPAD_COL(2),
   D => KEYPAD_COL(3)
   );

UMUX4_1: MUX4_1 port map(
   S(0) => Linha(0),
   S(1) => Linha(1),
   A => KEYPAD_LIN(0),
   B => KEYPAD_LIN(1),
   C => KEYPAD_LIN(2),
   D => KEYPAD_LIN(3),
   Y => B);

UCOUNTER: COUNTER port map(
   Clk => Clk,
   E => A,
   clr => Clr,
   R(0) => Linha(0),
   R(1) => Linha(1),
   R(2) => Coluna(0),
   R(3) => Coluna(1));

end arq_KEY_SCAN;
```

Key Control

```
1   LIBRARY IEEE;
2   USE IEEE.std_logic_1164.all;
3
4   ENTITY KEY_CONTROL IS
5      PORT(
6         CLK : in std_logic;
7         Kack : in std_logic;
8         Kpress: in std_logic;
9         Reset: in std_logic;
10        Kval : out std_logic;
11        Kscan : out std_logic
12        );
13  END KEY_CONTROL;
14
15  architecture behavioral of KEY_CONTROL is
16
17  type STATE_TYPE is (STATE_NOT_WRITING, STATE_WRITING, STATE_WRITING_COMPLETE);
18
19  signal CurrentState, NextState : STATE_TYPE;
20
21  begin
22
23  -- Flip-Flop`s
24  CurrentState <= STATE_NOT_WRITING when Reset = '1' else NextState when rising_edge(CLK);
25
26  --Generate Next State
27  GenerateNextState:
28  process(CurrentState, Kack, Kpress)
29     begin
30        case CurrentState is
31
32           when STATE_NOT_WRITING     => if (Kpress = '1') then NextState <= STATE_WRITING;
33                                         else
34                                            NextState <= STATE_NOT_WRITING;
35                                         end if;
36
37           when STATE_WRITING         => if (Kack = '1') then NextState <=
    STATE_WRITING_COMPLETE;
38                                         else
39                                               NextState <= STATE_WRITING;
40                                         end if;
41           when STATE_WRITING_COMPLETE=> if (Kack = '0' and kpress = '0') then NextState <=
    STATE_NOT_WRITING;
42                                         else
43                                              NextState <= STATE_WRITING_COMPLETE;
44                                         end if;
45
46
47     end case;
48  end process;
49
50  --Generate outputs
51
52  Kscan <= '1' when (CurrentState = STATE_NOT_WRITING) else '0';
53
54  Kval <= '1' when (CurrentState = STATE_WRITING) else '0';
55
56  end behavioral;
```

Key Decode

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity Key_Decode is
5   port( Kack, CLK, Reset : IN STD_LOGIC;
6         Kval : OUT STD_LOGIC;
7         KEYPAD_LIN : in std_logic_vector(3 downto 0);
8         KEYPAD_CODE: out std_logic_vector(3 downto 0);
9         KEYPAD_COL : out std_logic_vector(3 downto 0)
10
11  );
12  end;
13
14  architecture arq_KeyDecode of Key_Decode is
15
16  component KEY_SCAN
17  port( Kscan, Clk, Clr : IN STD_LOGIC;
18        Kpress : OUT STD_LOGIC;
19        KEYPAD_LIN : in std_logic_vector(3 downto 0);
20        KEYPAD_CODE: out std_logic_vector(3 downto 0);
21        KEYPAD_COL : out std_logic_vector(3 downto 0)
22
23
24  );
25  end component;
26
27
28  component KEY_CONTROL
29  port( Kack,Kpress,CLK,Reset : IN STD_LOGIC;
30        Kval, Kscan : OUT STD_LOGIC
31  );
32  end component;
33
34
35
36
37  signal KscanSignal,KpressSignal : STD_LOGIC;
38
39  begin
40
41
42
43  uKeyScan : KEY_SCAN port map(
44                Kscan => KscanSignal,
45                Kpress => KpressSignal,
46                Clr => Reset,
47                Clk => CLK,
48                KEYPAD_LIN => KEYPAD_LIN,
49                KEYPAD_CODE => KEYPAD_CODE,
50                KEYPAD_COL =>  KEYPAD_COL
51
52  );
53
54  uKeyControL : KEY_CONTROL port map(
55                Reset => Reset,
56                Kack => Kack,
57                Kpress => KpressSignal,
58                Clk => CLK,
59                Kval => Kval,
60                Kscan => KscanSignal
61  );
62
63
64  end arq_KeyDecode;
```

B. Descrição VHDL do bloco *Ring Buffer*

Ring Buffer Control

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity RingBufferControl is
5   port( DAV,clk,reset, CTS, full, empty : IN STD_LOGIC;
6         Wreg, Wr, selPnG, DAC, incPut, incGet : OUT STD_LOGIC
7   );
8   end RingBufferControl;
9
10  architecture arq_RingBufferControl of RingBufferControl is
11
12  type STATE_TYPE is (WAITING, WAIT_FOR_READING, READING, ADDRESSING, WRITING, INCRPUT,
    INCRGET, COMPLETED);
13
14  signal CURRENT_STATE, NEXT_STATE : STATE_TYPE;
15
16  begin
17
18  --    Registo Current State
19
20  CURRENT_STATE <= WAITING when (reset = '1') else NEXT_STATE when rising_edge(clk);
21
22  --    Máquina de Estados
23
24  GenerateNextState:
25  process (CURRENT_STATE, DAV, CTS, full, empty)
26     begin
27        case CURRENT_STATE is
28           when WAITING         => if(DAV = '0' and CTS = '1' and empty = '0') then
29                                      NEXT_STATE     <= READING;
30                                   elsif(DAV = '1' and empty = '1') then
31                                      NEXT_STATE     <= ADDRESSING;
32                                   elsif(DAV = '1' and full = '1') then
33                                      NEXT_STATE     <= WAIT_FOR_READING;
34                                   elsif(DAV = '1' and full = '0') then
35                                      NEXT_STATE     <= ADDRESSING;
36                                   else
37                                      NEXT_STATE     <= WAITING;
38                                   end if;

39           when ADDRESSING      => NEXT_STATE        <= WRITING;
40
41           when WRITING         => NEXT_STATE        <= INCRPUT;
42           when INCRPUT            => NEXT_STATE        <= COMPLETED;
43
44           when WAIT_FOR_READING=>if(CTS = '1') then
45                                      NEXT_STATE     <= READING;
46                                   else
47                                      NEXT_STATE     <= WAIT_FOR_READING;
48                                   end if;
49
50           when READING         =>if(CTS = '0') then
51                                      NEXT_STATE     <= INCRGET;
52                                   else
53                                      NEXT_STATE     <= READING;
54                                   end if;
55           when INCRGET          =>if(DAV = '1') then
56                                        NEXT_STATE     <= ADDRESSING;
57                                     else
58                                        NEXT_STATE     <= WAITING;
59                                     end if;

60           when COMPLETED => if(DAV = '0') then
61                                        NEXT_STATE     <= WAITING;
62                                     else
63                                        NEXT_STATE     <= COMPLETED;
64                                     end if;
65
66                 end case;
67  end process;
```

```
68
69    --    Outputs
70
71
72    incGet   <= '1' when (CURRENT_STATE = INCRGET)     else '0';
73    selPnG<= '1' when (CURRENT_STATE = ADDRESSING  OR CURRENT_STATE = WRITING )   else '0';
74    Wreq  <= '1' when (CURRENT_STATE = READING)   else '0';
75    Wr <= '1' when (CURRENT_STATE = WRITING)   else '0';
76    incPut <= '1' when (CURRENT_STATE = INCRPUT ) else '0';
77    DAC <= '1' when (CURRENT_STATE = COMPLETED) else '0';
78
79
80    end arq_RingBufferControl;
```

Memory Address Control

Contador de 3 bits

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity Counter_3bit is
5       Port ( CLK : in  STD_LOGIC;
6              RST : in  STD_LOGIC;
7              ndecInc : in  STD_LOGIC;
8              en : in  STD_LOGIC;
9              Count : out  STD_LOGIC_VECTOR (3 downto 0));
10  end Counter_3bit;
11
12  architecture Behavioral of Counter_3bit is
13
14      COMPONENT CounterLogic_3bit
15      PORT(
16         en : IN std_logic;
17         ndecInc : IN std_logic;
18         operandA : IN std_logic_vector(3 downto 0);
19         R : OUT std_logic_vector(3 downto 0)
20         );
21      END COMPONENT;
22
23      COMPONENT register_D_R
24      GENERIC (
25         WIDTH : POSITIVE := 1
26      );
27      PORT(
28         CLK : IN std_logic;
29         RST : IN std_logic;
30         D : IN std_logic_vector(WIDTH-1 downto 0);
31         Q : OUT std_logic_vector(WIDTH-1 downto 0)
32         );
33      END COMPONENT;
34
35      signal operandA, result : std_logic_vector(3 downto 0);
36
37  begin
38
39      Inst_CounterLogic_3bit: CounterLogic_3bit PORT MAP(
40         en => en,
41         ndecInc => ndecInc,
42         operandA => operandA,
43         R => result
44      );
45
46      Inst_register_D_R: register_D_R GENERIC MAP(
47         WIDTH => 4
48      )
```

```
49      PORT MAP(
50         CLK => CLK,
51         RST => RST,
52         D => result,
53         Q => operandA
54      );
55
56      Count <= operandA;
57
58   end Behavioral;
59
60
```

Contador crescente/decrescente 4 bits

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity Counter_4bit is
5      Port ( CLK : in  STD_LOGIC;
6             RST : in  STD_LOGIC;
7             ndecInc : in  STD_LOGIC;
8             en : in  STD_LOGIC;
9             Count : out  STD_LOGIC_VECTOR (3 downto 0));
10 end Counter_4bit;
11
12 architecture Behavioral of Counter_4bit is
13
14    COMPONENT CounterLogic_4bit
15    PORT(
16       en : IN std_logic;
17       ndecInc : IN std_logic;
18       operandA : IN std_logic_vector(3 downto 0);
19       R : OUT std_logic_vector(3 downto 0)
20       );
21    END COMPONENT;
22
23    COMPONENT register_D_R
24    GENERIC (
25       WIDTH : POSITIVE := 1
26    );
27    PORT(
28       CLK : IN std_logic;
29       RST : IN std_logic;
30       D : IN std_logic_vector(WIDTH-1 downto 0);
31       Q : OUT std_logic_vector(WIDTH-1 downto 0)
32       );
33    END COMPONENT;
34
```

```
35    signal operandA, result : std_logic_vector(3 downto 0);
36
37 begin
38
39    Inst_CounterLogic_4bit: CounterLogic_4bit PORT MAP(
40       en => en,
41       ndecInc => ndecInc,
42       operandA => operandA,
43       R => result
44    );
45
46    Inst_register_D_R: register_D_R GENERIC MAP(
47       WIDTH => 4
48    )
49    PORT MAP(
50       CLK => CLK,
51       RST => RST,
52       D => result,
53       Q => operandA
54    );
55
56    Count <= operandA;
57
58 end Behavioral;
59
```

Register_D_R

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity register_D_R is
5       Generic (WIDTH : POSITIVE := 1);
6       Port (CLK : in  STD_LOGIC;
7             RST : in STD_LOGIC;
8             D : in  STD_LOGIC_VECTOR(WIDTH-1 downto 0);
9             Q : out  STD_LOGIC_VECTOR(WIDTH-1 downto 0));
10  end register_D_R;
11
12  architecture Behavioral of register_D_R is
13
14  begin
15
16     Q <= (others => '0') when RST = '1' else D when rising_edge(clk);
17
18  end Behavioral;
19
```

Multiplexer 2x1

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity MUX2_1_3bits is
5       Port ( I0 : in  STD_LOGIC_VECTOR (2 downto 0);
6              I1 : in  STD_LOGIC_VECTOR (2 downto 0);
7              sel : in  STD_LOGIC;
8              Y : out  STD_LOGIC_VECTOR (2 downto 0));
9   end MUX2_1_3bits;
10
11  architecture Behavioral of MUX2_1_3bits is
12
13  begin
14
15     Y <= I0 when sel='0' else I1;
16
17  end Behavioral;
18
```

CounterLogic_ 4bit

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity CounterLogic_4bit is
5       Port ( en : in  STD_LOGIC;
6              ndecInc : in  STD_LOGIC;
7              operandA : in  STD_LOGIC_VECTOR (3 downto 0);
8              R : out  STD_LOGIC_VECTOR (3 downto 0));
9   end CounterLogic_4bit;
10
11  architecture Structural of CounterLogic_4bit is
12
13     COMPONENT adder4bit
14     PORT(
15        A : IN std_logic_vector(3 downto 0);
16        B : IN std_logic_vector(3 downto 0);
17        CI : IN std_logic;
18        R : OUT std_logic_vector(3 downto 0);
19        CO : OUT std_logic
20        );
21     END COMPONENT;
22
23     COMPONENT MUX2_1
24     PORT(
25        I0 : IN std_logic_vector(3 downto 0);
26        I1 : IN std_logic_vector(3 downto 0);
27        sel : IN std_logic;
28        Y : OUT std_logic_vector(3 downto 0)
29        );
30     END COMPONENT;
31
```

```
32      signal operandB : STD_LOGIC_VECTOR(3 downto 0) ;
33      signal increment : STD_LOGIC_VECTOR(3 downto 0) ;
34
35  begin
36
37      U2_adder4bit: adder4bit PORT MAP(
38          A => operandA,
39          B => operandB,
40          CI => '0',
41          R => R,
42          CO => open
43      );
44
45      U0_MUX2_1: MUX2_1 PORT MAP(
46          I0 => "1111",
47          I1 => "0001",
48          sel => ndecInc,
49          Y => increment
50      );
51
52      U1_MUX2_1: MUX2_1 PORT MAP(
53          I0 => "0000",
54          I1 => increment,
55          sel => en,
56          Y => operandB
57      );
58
59  end Structural;
60
```

MAC

```
1   LIBRARY IEEE;
2   USE IEEE.std_logic_1164.all;
3
4   ENTITY MAC IS
5      PORT( MCLK,reset : IN STD_LOGIC;
6            putnget, incPut, incGet: IN STD_LOGIC;
7            full,empty : OUT STD_LOGIC;
8            Address : OUT STD_LOGIC_VECTOR(2 downto 0)
9      );
10  END MAC;
11
12
13  ARCHITECTURE arq_MAC OF MAC IS
14
15  component Counter_4bit is
16      Port ( CLK : in  STD_LOGIC;
17             RST : in  STD_LOGIC;
18             ndecInc : in  STD_LOGIC;
19             en : in  STD_LOGIC;
20             Count : out  STD_LOGIC_VECTOR (3 downto 0));
21  end component;
22
```

```
23    component COUNTER_3 IS
24       PORT(
25          CLK : in std_logic;
26          E : in std_logic;
27          CLR: in std_logic;
28          R : out std_logic_vector (2 downto 0) :=(others => '0')
29          );
30
31    end component;
32
33
34    component MUX2_1_3bits is
35        Port ( I0 : in  STD_LOGIC_VECTOR (2 downto 0);
36               I1 : in  STD_LOGIC_VECTOR (2 downto 0);
37               sel : in  STD_LOGIC;
38               Y : out  STD_LOGIC_VECTOR (2 downto 0));
39    end component;
40
41
42    signal idxGetSignal, idxPutSignal : std_logic_vector(2 downto 0);
43    signal membersSignal : std_logic_vector(3 downto 0);
44    signal enableSignal :STD_LOGIC;
45
46
47    begin
48
49    enableSignal <= incPut OR incGet;
50
51    Ucounter_4 : Counter_4bit port map (
52          CLK => MCLK,
53          RST => reset,
54          ndecInc =>  incPut, --0 dec 1 inc
55          en => enableSignal,
56          Count => membersSignal
57    );
58
59    Ucounter_3_putIdx : COUNTER_3 port map (
60          CLK => MCLK,
61          E => incPut,
62          CLR => reset,
63          R => idxPutSignal
64    );
65
66
67    Ucounter_3_getIdx : COUNTER_3 port map(
68          CLK => MCLK,
69          E => incGet,
70          CLR => reset,
71          R => idxGetSignal
72    );
73
74
75    UMux2_1 :  MUX2_1_3bits port map (
76                I0 => idxGetSignal,
77                I1 => idxPutSignal,
```

```
78                  sel => putnget,
79                  Y => Address
80      );
81
82      full <= membersSignal(3);
83      empty <= not membersSignal(3) AND not membersSignal(2) AND not membersSignal(1) AND not
        membersSignal(0);
84
85      end arq_MAC;
86
87
88
89
90
```

C. Descrição VHDL do bloco *Output Buffer*

Output Buffer Control

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity OutputBufferControl is
5   port( clk,reset, Load, ACK : IN STD_LOGIC;
6         Wreg, OBfree, Dval : OUT STD_LOGIC
7   );
8   end OutputBufferControl;
9
10  architecture arq_OutputBufferControl of OutputBufferControl is
11
12  type STATE_TYPE is (WAITING, SENDING, COMPLETED);
13
14  signal CURRENT_STATE, NEXT_STATE : STATE_TYPE;
15
16  begin
17
18  --    Registo Current State
19
20  CURRENT_STATE <= WAITING when (reset = '1') else NEXT_STATE when rising_edge(clk);
21
22  --    Máquina de Estados
23
24  GenerateNextState:
25  process (CURRENT_STATE, Load, ACK )
26     begin
27        case CURRENT_STATE is
28           when WAITING   => if(Load = '1') then
29                                NEXT_STATE     <= SENDING;
30                             else
31                                NEXT_STATE     <= WAITING;
32                             end if;
33
34           when SENDING   =>    NEXT_STATE     <= COMPLETED;
35
36           when COMPLETED => if(ACK = '1') then
37                                NEXT_STATE     <= WAITING;
38                             else
39                                NEXT_STATE     <= COMPLETED;
40                             end if;
41
42                 end case;
43  end process;
44
45  --    Outputs
46
47
48  OBfree <= '1' when (CURRENT_STATE = WAITING)     else '0';
49  Dval <= '1' when (CURRENT_STATE = COMPLETED)     else '0';
50  Wreg <= '1' when (CURRENT_STATE = SENDING)     else '0';
51
52
53
54  end arq_OutputBufferControl;
```

Registor

```
1   LIBRARY IEEE;
2   USE IEEE.std_logic_1164.all;
3
4   ENTITY REGISTOR IS
5      PORT( R : in std_logic_vector(3 downto 0);
6         CLR : in std_logic;
7         CL : in std_logic;
8         E : in std_logic;
9         TC : out std_logic;
10        F : out std_logic_vector (3 downto 0)
11        );
12  END REGISTOR;
13
14
15  ARCHITECTURE arq_REGISTOR OF REGISTOR IS
16
17  COMPONENT FFD
18     PORT( CLK : in std_logic;
19        RESET : in STD_LOGIC;
20        SET : in std_logic;
21        D : IN STD_LOGIC;
22        EN : IN STD_LOGIC;
23        Q : out std_logic
24        );
25  END COMPONENT;
26
27  SIGNAL Res: std_logic_vector(3 downto 0);
28
29
30  BEGIN
31
32  F <= Res;
33
34  TC <= not Res(0) and not Res(1) and not Res(2) and not Res(3);
35
36  U0FFD : FFD port map(
37     CLK => CL,
38     RESET => CLR,
39     SET => '0',
40     D => R(0),
41     EN => E,
42     Q => Res(0)
43     );
44
45  U1FFD : FFD port map(
46     CLK => CL,
47     RESET => CLR,
48     SET => '0',
49     D => R(1),
50     EN => E,
51     Q => Res(1)
52     );
53
54  U2FFD : FFD port map(
55     CLK => CL,
56     RESET => CLR,
57     SET => '0',
```

```
58       D => R(2),
59       EN => E,
60       Q => Res(2)
61       );
62
63    U3FFD : FFD port map(
64       CLK => CL,
65       RESET => CLR,
66       SET => '0',
67       D => R(3),
68       EN => E,
69       Q => Res(3)
70       );
71
72    END arq_REGISTOR;
```

Output Buffer

```
 1    LIBRARY IEEE;
 2    USE IEEE.std_logic_1164.all;
 3
 4    ENTITY Output_Buffer IS
 5       PORT( MCLK,reset : IN STD_LOGIC;
 6             Load, ACK: IN STD_LOGIC;
 7             Dval, OBfree : OUT STD_LOGIC;
 8             Din : IN STD_LOGIC_VECTOR(3 downto 0);
 9             Q : OUT STD_LOGIC_VECTOR(3 downto 0)
10       );
11    END Output_Buffer;
12
13
14    ARCHITECTURE arq_Output_Buffer OF Output_Buffer IS
15
16
17
18    COMPONENT OutputBufferControl is
19    port( clk,reset, Load, ACK : IN STD_LOGIC;
20          Wreg, OBfree, Dval : OUT STD_LOGIC
21    );
22    END COMPONENT;
23
24
25    COMPONENT REGISTOR IS
26       PORT( R : in std_logic_vector(3 downto 0);
27          CLR : in std_logic;
28          CL : in std_logic;
29          E : in std_logic;
30          TC : out std_logic;
31          F : out std_logic_vector (3 downto 0)
32          );
33    END COMPONENT;
34
35    signal  WregSignal :STD_LOGIC;
36
37    begin
38
```

```
39  UOutputBufferControl : OutputBufferControl port map(
40     clk      => MCLK,
41     reset    => reset,
42     Load     => Load,
43     ACK      => ACK,
44     Wreg     => WregSignal,
45     OBfree   => OBfree,
46     Dval     => Dval
47  );
48
49  UREGISTOR : REGISTOR port map(
50           R    => Din,
51         CLR    => reset,
52         CL     => MCLK,
53         E      => WregSignal,
54         F      => Q
55  );
56
57  end arq_Output_Buffer;
```

Keyboard Reader

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity Keyboard_Reader is
5   port( CLK, Reset, ACK : IN STD_LOGIC;
6         Dval : OUT STD_LOGIC;
7         KEYPAD_LIN : in std_logic_vector(3 downto 0);
8         KEYPAD_COL : out std_logic_vector(3 downto 0);
9         Q : out std_logic_vector(3 downto 0)
10
11  );
12  end Keyboard_Reader;
13
14  architecture arq_Keyboard_Reader of Keyboard_Reader is
15
16
17  component Ring_Buffer IS
18     PORT( MCLK,reset : IN STD_LOGIC;
19           DAV, CTS: IN STD_LOGIC;
20           DAC, Wreg : OUT STD_LOGIC;
21           Din : IN STD_LOGIC_VECTOR(3 downto 0);
22           Q : OUT STD_LOGIC_VECTOR(3 downto 0)
23     );
24  end component;
25
26
27  component Output_Buffer IS
28     PORT( MCLK,reset : IN STD_LOGIC;
29           Load, ACK: IN STD_LOGIC;
30           Dval, OBfree : OUT STD_LOGIC;
31           Din : IN STD_LOGIC_VECTOR(3 downto 0);
32           Q : OUT STD_LOGIC_VECTOR(3 downto 0)
33     );
34  end component;
35
36  component Key_Decode is
37  port( Kack,CLK,Reset : IN STD_LOGIC;
38        Kval : OUT STD_LOGIC;
39        KEYPAD_LIN : in std_logic_vector(3 downto 0);
40        KEYPAD_CODE: out std_logic_vector(3 downto 0);
41        KEYPAD_COL : out std_logic_vector(3 downto 0)
42
43  );
44  end component;
45
46
47  --component CLKDIV is
48  --generic(div: natural := 50000000);
49  --port ( clk_in: in std_logic;
50     --  clk_out: out std_logic);
51  --end component;
52
53
54  signal WregSignal, Clk_signal, OBfreeSignal,DAVSignal , DACSignal : STD_LOGIC;
55  signal codeSignal,QSignal : STD_LOGIC_VECTOR(3 downto 0);
56
57  begin
58
59
60  --uCLKDIV : CLKDIV generic map(2)  port map(
61     -- clk_in => CLK,
62        --clk_out => Clk_signal
63  --);
64
```

```
78                  MCLK => CLK,
79                  reset => reset,
80                  Load => WregSignal,
81                  ACK => ACK,
82                  Dval => Dval,
83                  OBfree => OBfreeSignal,
84                  Din => QSignal,
85                  Q => Q
86  );
87
88  uKey_Decode : Key_Decode port map(
89        Reset => reset,
90        CLK => CLK,
91        Kack => DACSignal,
92        Kval => DAVSignal,
93        KEYPAD_LIN => KEYPAD_LIN,
94        KEYPAD_CODE => codeSignal,
95        KEYPAD_COL  => KEYPAD_COL
96
97  );
65  uRing_Buffer : Ring_Buffer port map(
66                  MCLK => CLK,
67                  reset => reset,
68                  DAV => DAVSignal,
69                  CTS => OBfreeSignal,
70                  DAC => DACSignal,
71                  Wreg => WregSignal,
72                  Din => codeSignal,
73                  Q => QSignal
74  );
75
76
77  uOutput_Buffer : Output_Buffer port map(
```

D. Descrição VHDL do bloco KeyboardReaderUSBPORT

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity Keyboard_ReaderUSBPORT is
5   port( CLK, Reset : IN STD_LOGIC;
6         KEYPAD_LIN : in std_logic_vector(3 downto 0);
7         KEYPAD_COL : out std_logic_vector(3 downto 0)
8
9   ) ;
10  end Keyboard_ReaderUSBPORT;
11
12  architecture arq_Keyboard_Reader of Keyboard_ReaderUSBPORT is
13
14  COMPONENT UsbPort IS
15        PORT
16        (
17        inputPort:  IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
18        outputPort :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0)
19        );
20  END COMPONENT;
21
22
23  component Keyboard_Reader is
24  port( CLK, Reset, ACK : IN STD_LOGIC;
25        Dval : OUT STD_LOGIC;
26        KEYPAD_LIN : in std_logic_vector(3 downto 0);
27        KEYPAD_COL : out std_logic_vector(3 downto 0);
28        Q : out std_logic_vector(3 downto 0)
29
30  );
31  end component;
32
33
34  signal WregSignal, Clk_signal, OBfreeSignal,DAVSignal , DACSignal : STD_LOGIC;
35  signal codeSignal,QSignal : STD_LOGIC_VECTOR(3 downto 0);
36  signal UsbPortInputSignal, UsbPortOutputSignal : STD_LOGIC_VECTOR (7 downto 0);
37
38  begin
39
40
41  uUsbPort: UsbPort PORT MAP(
42            inputPort   => UsbPortInputSignal,
43            outputPort  => UsbPortOutputSignal
44  );
45
46
47
48  uKeyboardReader : Keyboard_Reader port map(
49     CLK => CLK,
50     Reset => reset,
51     ACK => UsbPortOutputSignal(7), --ver no KBD
52     Dval => UsbPortInputSignal(4), --ver no KBD
53     KEYPAD_LIN => KEYPAD_LIN,
54     KEYPAD_COL => KEYPAD_COL,
55     Q => UsbPortInputSignal(3 downto 0) --ver no KBD
56  );
57
58
59  end arq_Keyboard_Reader;
```

E. Atribuição de pinos do módulo *Keyboard_ReaderUSBPORT*

```
#============================================================
# CLOCK
#============================================================

set_location_assignment PIN_P11 -to CLK
#============================================================
#SWITCHES
#============================================================
set_location_assignment PIN_C10 -to Reset

#============================================================

set_location_assignment PIN_W5 -to KEYPAD_LIN[0]
set_location_assignment PIN_AA14 -to KEYPAD_LIN[1]
set_location_assignment PIN_W12 -to KEYPAD_LIN[2]
set_location_assignment PIN_AB12 -to KEYPAD_LIN[3]
set_location_assignment PIN_AB11 -to KEYPAD_COL[0]
set_location_assignment PIN_AB10 -to KEYPAD_COL[1]
set_location_assignment PIN_AA9 -to KEYPAD_COL[2]
set_location_assignment PIN_AA8 -to KEYPAD_COL[3]
#============================================================
# End of pin and io_standard assignments
#============================================================
```

F. Código *Kotlin - HAL*

```
object HAL {
           var written = 0b0000_0000
         private var ACTIVE = false
           fun init() { // Inicia a classe
               if(!ACTIVE) {
                   UsbPort.write(written)
                   ACTIVE=true
               }
           }

       // Retorna true se o bit tiver o valor lógico ‘1’
       fun isBit(mask: Int): Boolean = (mask and UsbPort.read()) != 0

       // Retorna os valores dos bits representados por mask presentes no UsbPort
       fun readBits(mask: Int): Int = mask and UsbPort.read()

       // Escreve nos bits representados por mask o valor de value
       /**
       * value -> 0000_1001.
       * mask -> 0000_1111.
       * lastWritten -> 1111_0111.
       * new lastWritten -> 1111_1001.
       * 1º: (value and mask) -> 0000_1001 sets the bits in value to be written to the ones in the
       mask.
       * 2º: (lastWritten and mask.inv()) -> 1111_0000 sets the bits in lastWritten that are not
       in the mask,this operation
       *      sets to 0 all the bits in lastWritten that are not in the mask, preparing it to
       receive the updated value.
       * 3º: (value and mask) or (lastWritten and mask.inv()) -> 0000_1001 or 1111_0000 ->
       1111_1001 sets the bits in
       *      lastWritten that are in the mask to the corresponding bits in value and keeps the
       bits that are not in the mask unchanged.
       */

       fun writeBits(mask: Int, value: Int) {
              written = (value and mask) or (written and mask.inv())
                UsbPort.write(written)
       }

       // Coloca os bits representados por mask no valor lógico ‘1’
       fun setBits(mask: Int) {
           written = (written or mask)
           UsbPort.write(written)
       }

       // Coloca os bits representados por mask no valor lógico ‘0’
       fun clrBits(mask: Int) {
           written = written and mask.inv()
           UsbPort.write(written)
       }

}
```

G. Código *Kotlin - KBD*

```
//
/**
 * Mapeamento das teclas do teclado matricial 4x4:
 *          Key  Column 1    Key Column 2   Key Column 3
 *Row 1     1    0x00        2   0x04       3   0x08
 *Row 2     4    0x01        5   0x05       6   0x09
 *Row 3     7    0x02        8   0x06       9   0x0A
 *Row 4     *    0x03        0   0x07       #   0x0B
 **/

//Read keys. Methods return '0'..' 9','#','*' or NONE.
object KBD {
    const val NONE = 0.toChar()
    const val DATA = 0x0F      //UsbPort.I0..3
    const val MASK_DVAL = 0x10 //UsbPort.I4
    const val MASK_ACK = 0x80 //UsbPort.O7
    private val KEYS : List<Char> = listOf('1', '4', '7', '*', '2', '5', '8', '0', '3', '6', '9',
'#')
                                        // 0    1    2    3    4    5    6    7    8    9    10
11
    // Starts the class.
    fun init() {
        HAL.init()
    }
     // Returns the pressed or NONE key immediately if there is no key pressed.
    fun getKey(): Char {
        if (HAL.isBit(MASK_DVAL)) {
            val a = HAL.readBits(DATA)
            return if (a in 0..11) {
                HAL.setBits(MASK_ACK)
                HAL.clrBits(MASK_ACK)
                KEYS[a]

            } else NONE
        }
         return NONE
    }
    // Returns when the key is pressed or NONE after millisecond timeout has elapsed.
    fun waitKey(timeout: Long): Char {
        var time = timeout
        while (time > 0) {
            val serial = getKey()
            if (serial != NONE) return serial
            Thread.sleep(1)
            time--
        }
        return NONE
    }

}
```