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.

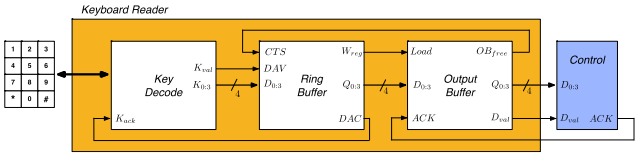
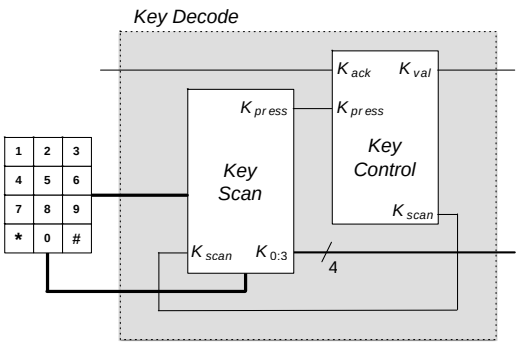


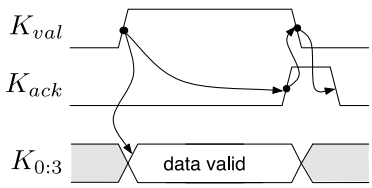
Figura 1 – Diagrama de blocos do módulo *Keyboard Reader*

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. Foi escolhida a versão 3, já que nesta versão a tecla premida é detectada mais rapidamente (o varrimento da linha demora apenas um ciclo de clock, devido ao registo do valor de saída do *PENC* e o varrimento da coluna é mais rápido em relação à primeira versão porque é feito através de um contador de 2 bits separadamente do varrimento das linhas) e são utilizados menos recursos (menos dois full adders já que é usado apenas um contador de dois bits).

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

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

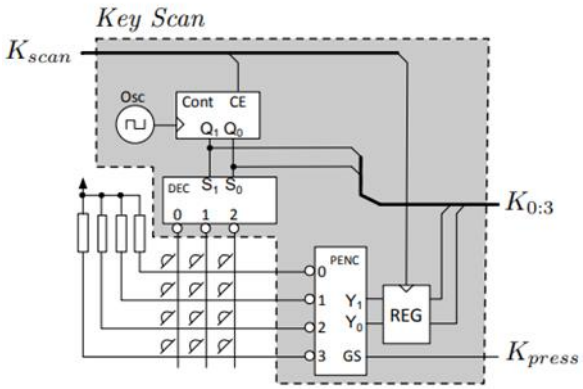


Figura 3 - Diagrama de blocos do bloco *Key Scan*

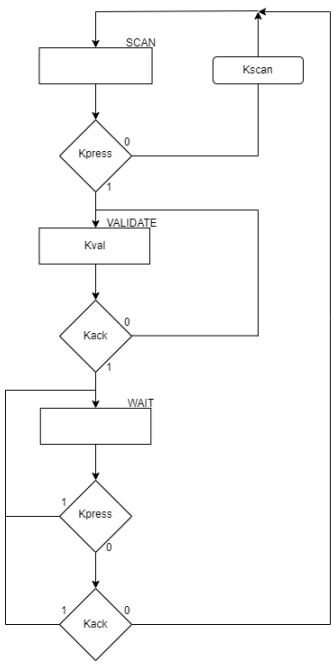


Figura 4 – Máquina de estados do bloco *Key Control*

Quando a máquina se encontra no estado inicial, **SCAN**, encontra-se a fazer o varrimento da tecla até ser detetada uma tecla. Assim que é detetada uma tecla premida, a máquina de estados passa para o estado **VALIDATE** em que envia o sinal $Kval$ para indicar ao *Ring Buffer* que tem uma tecla para ser lida. Depois de ser enviado o sinal de K_{ack} pelo *Ring Buffer*, a máquina passa para o estado **WAIT**, em que espera para que o *Ring Buffer* baixe o sinal K_{ack} e o utilizador deixe de premir a tecla. Assim que a tecla deixa de ser premida e o sinal de K_{ack} baixa, a máquina volta para o estado **SCAN** em que faz o varrimento da tecla.

Este módulo opera com a frequência da placa DE10-Lite da Intel(50 MHz), mas no bloco *Key Decode* foi adicionado um divisor de clock por 500000 que resulta em uma frequência de 100 Hz ou 10 ms por clock. Foi escolhida esta frequência para evitar a deteção de bouncing nas teclas e no sinal K_{press}, que causaria múltiplas leituras da mesma tecla.

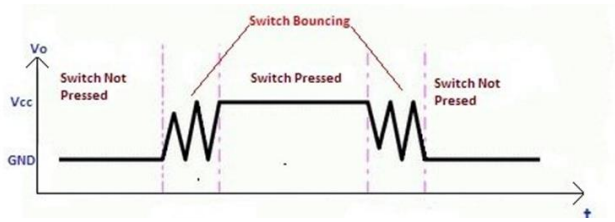


Figura 5 – Exemplo de bouncing

2 Ring Buffer

O bloco *Ring Buffer* desenvolvido 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* foi baseada numa memória RAM (Random Access Memory). O endereço de escrita/leitura, selecionado por putget, 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 *incPut* e *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 6 é apresentado o diagrama de blocos para uma estrutura do bloco *Ring Buffer*.

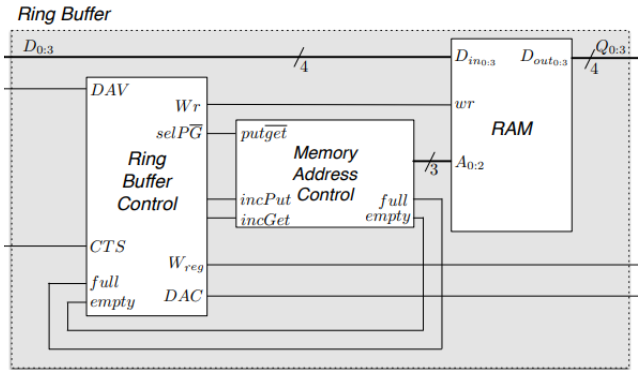


Figura 6 – Diagrama de blocos do bloco *Ring Buffer*.

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

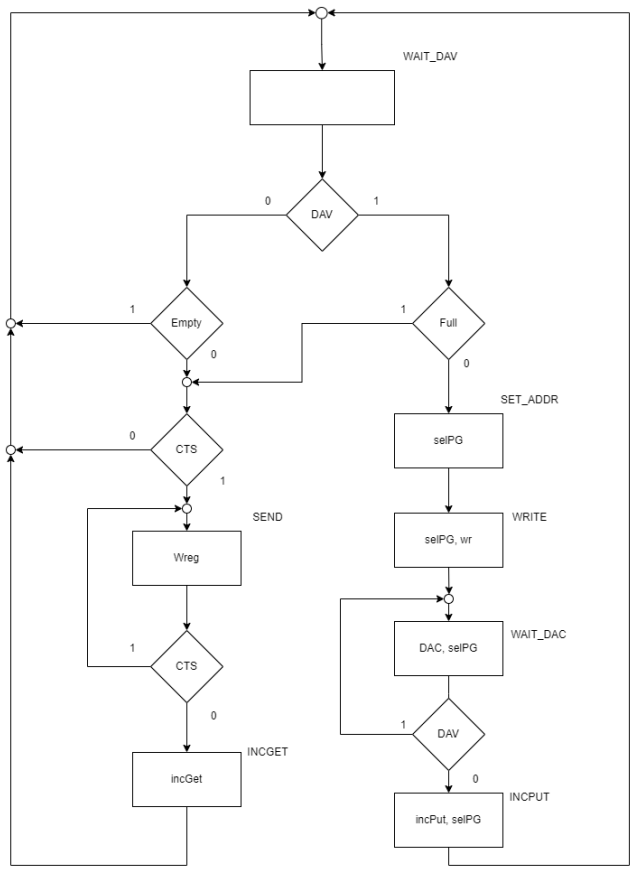


Figura 7 – Máquina de estados do bloco *Ring Buffer Control*.

Após ser ligada a máquina, encontra-se no estado **WAIT_DAV** à espera de que sejam enviados dados. Quando existem dados para serem escritos e a memória não estiver cheia, a máquina procede à escrita de dados em memória. Estabelece o endereço de escrita no estado **SETADDR**, escreve na memória no estado **WRITE**, e espera para o bloco *Key Decode* baixar o sinal *DAV* no estado **WAIT_DAC**, após ativar o sinal *DAC*. Depois de escrever os dados, é incrementado o endereço de escrita e volta-se ao estado inicial. Se a memória estiver cheia com *DAV*, então a máquina de estados não escreve dados em memória e verifica se o *Output Buffer* está disponível para receber dados através do sinal CTS. Da mesma forma, se a memória não estiver vazia sem DAV, a máquina procede à verificação do sinal CTS, para verificar se podem ser enviados dados. Nestas condições, se CTS está ativo, a máquina procede ao envio de dados para o *Output Buffer* no estado **SEND**. Quando é completo o envio, é incrementado o endereço de leitura e volta-se ao estado inicial. Sempre que não existirem dados para enviar e a memória estiver vazia, ou se houverem dados em memória, mas não houver sinal CTS, ou se a memória estiver cheia, mas não houver sinal CTS, a máquina de estados espera no estado inicial **WAIT_DAV**.

O bloco *MAC* foi implementado de acordo com o diagrama de blocos representado na Figura 8.

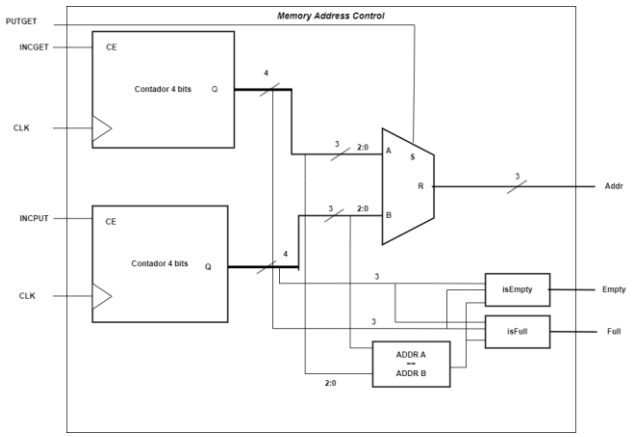


Figura 8 – Diagrama de blocos do bloco *Memory Address Control.*

Nesta implementação foram utilizados dois contadores de 4 bits, em que os 3 bits de menor peso representam os endereços *putIndex* e *getIndex*, e o bit de maior peso representa um sinal de controlo para verificação de ciclos. A lógica é a seguinte: Se os dois contadores tiverem o bit de controlo igual e os endereços forem iguais, então a memória encontra-se vazia. Se os dois contadores tiverem o bit de controlo diferente e os endereços forem iguais, então a memória encontra-se cheia. Como exemplo, quando o circuito é ligado inicialmente, os contadores encontram-se a 0000, e como os bits de controlo e endereço são iguais neste caso, a memória está vazia. Depois de 8 escritas, o contador de escrita encontra-se a 1000 e o de leitura a 0000. Neste caso, os bits de controlo são diferentes, e os de endereço são iguais, então a memória está cheia. Se forem lidos alguns valores até os contadores ficarem com os valores 1000-0011 então a memória não se encontra cheia nem vazia, porque apesar de os bits de controlo serem diferentes, os endereços não são iguais. A memória volta a ficar vazia quando o contador de leitura chegar ao valor 1000.

Para seleção de endereços, os 3 bits de endereço dos contadores entram num Mux 2:1 de 3 bits, e são selecionados dependendo do sinal putget.

Foram ligados aos sinais de EN dos contadores as entradas INCGET e INCPUT, que são saídas do bloco *Ring Buffer Control.*

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 OB_{free}. 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 OB_{free}. Na Figura 9, é apresentado o diagrama de blocos do *Output Buffer*.

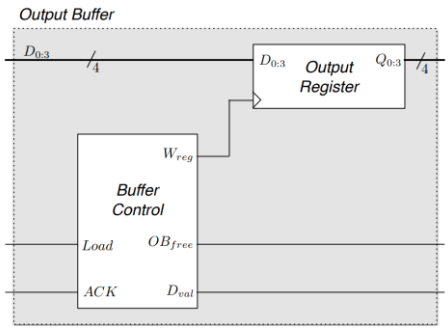


Figura 9 – Diagrama de blocos do bloco *Output Buffer*

Sempre que o bloco emissor *Ring Buffer* tenha dados disponíveis e o bloco de entrega *Output Buffer* esteja disponível (OB_{free} 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 OB_{free}.

O bloco *Output Buffer Control* foi implementado pela máquina de estados representada em ASM-chart na figura 10.

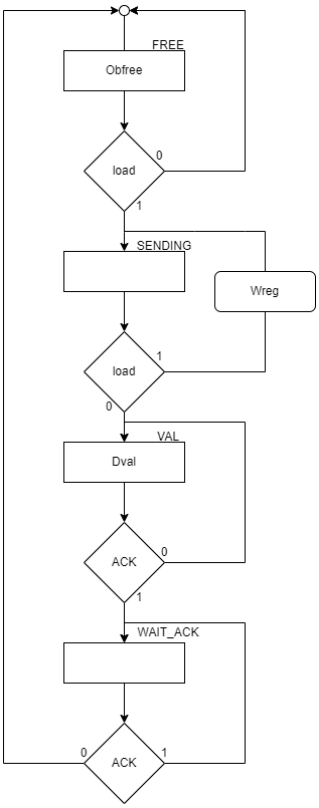


Figura 10 – Máquina de estados do bloco *Output Buffer*

Quando a máquina é ligada inicialmente no estado **FREE**, o *Output Buffer* encontra-se disponível para receber dados e envia para o *Ring Buffer* o sinal OB_{free}. Quando são enviados dados pela entidade emissora através da ativação do sinal *Load*, a máquina passa para o estado **SENDING**, em que ativa o sinal $Wreg$ para escrever os dados no registo. Quando *Load* baixar, passa para o estado **VAL** em que indica que tem dados para enviar ao Controlo de software Através da ativação do sinal $Dval$. Após ser ativado o sinal de ACK pelo software, a máquina passa para o último estado **WAIT_ACK**, em que espera para o software baixar o sinal ACK de forma a poder libertar o *Output Buffer*, para poder ler mais dados da memória.

Para implementar o bloco *Output Register*, foi utilizado um registo de 4 bits.

4 Interface com o *Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem *Kotlin* e seguindo a arquitetura lógica apresentada na Figura 11.

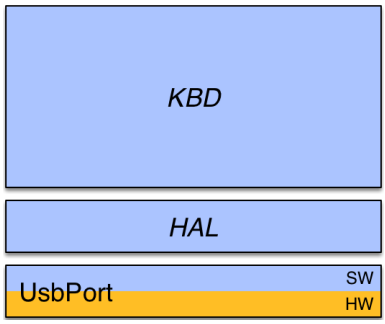


Figura 11 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

HAL e *KBD* desenvolvidos são descritos nas secções 4.1. e 4.2, e o código fonte desenvolvido nos Anexos E e F, respetivamente.

4.1 HAL

A classe HAL (*Hardware Abstraction Layer*) providencia ao software uma interface de interação com o hardware através de leitura e escrita de bits na *USB Port*.

Foi adicionada a variável val_write que indica o valor das saídas atuais da USB Port.

`init()`: Inicializa as saídas da USB Port com o valor 0.

`readbits(mask: Int)`: Lê o valor da USB port e faz um and "bit a bit" com `mask`. O resultado desta operação é de leitura dos valores da USB port nas posições em que os bits de mask são 1.

`isBit(mask: Int)`: Verifica se o bit na posição mask da USB Port está ativo ao chamar a função `readbits(mask)` e verificar se o resultado é maior que zero.

`setBits(mask: Int)`: Ativa os bits nas posições mask da USB Port ao realizar um or "bit a bit" do valor atual da USB Port com `mask` e escreve esse valor na USB Port.

`clrBits(mask : Int)` : Desativa os bits nas posições `mask` da USB Port ao inverter os bits `mask` e realizar um

and bit a bit entre o resultado dessa operação e o valor atual na USB Port.

`writeBits(mask: Int, value: Int)` Escreve os bits do valor `value` nas posições `mask` da USB Port. Para isto, inicialmente desativam-se os bits nas posições `mask`, ao realizar a mesma operação que em `clrBits`. Posteriormente, é realizado um or do resultado dessa operação com um and entre `value` e `mask`.

4.2 KBD

A classe KBD é responsável pela leitura de teclas inseridas pelo utilizador para o software.

Foram adicionadas algumas constantes. `NONE` representa um carácter nulo, que é útil para controlar o resto do software de acordo com o retorno das funções. `KBD`, `KVALMASK`, `KACKMASK`, `KCODEMASK`, representam as posições bit da USB Port que indicam os valores de $Keyval$, ACK , e do código da tecla, respetivamente. `CHECK_INTERVAL` é o atraso entre verificações de tecla premida na função `waitKey`.

Keycode representa o valor do código da tecla, *keyval* representa o estado do bit K_{val} da USB Port e kbdmatrix representa a matriz do teclado.

A função init inicializa *keyval* com o valor false e *keycode* com o valor `NONE`.

A função `getkey` retorna o carácter correspondente ao código keycode lido na USBPort e envia a ACK para o hardware. Para esta operação, inicialmente é lido da USB Port o valor de *keyval* através de `HAL.isbit(KVALMASK)`, se *keyval* é true, então procede-se à leitura do código da tecla através de `HAL.readbits(KCODEMASK)` e dá-se o sinal de *ACK* ao hardware através de `HAL.setbits(KACKMASK)`. Posteriormente espera-se para o circuito receber o sinal *ACK* e desativar o sinal *keyval*, e então desativa-se o sinal *ACK* (figura 2b). Caso *keyval* seja false inicialmente ou o código da tecla não se encontrar na matriz do teclado, então a função retorna carácter nulo.

A função `waitKey` espera que uma tecla seja premida, chamando repetidamente a função `getKey` até uma tecla ser premida ou até que o tempo ultrapasse o timeout predefinido. Quando uma tecla é premida a função retorna o caractere correspondente à mesma.
Neste módulo foi utilizada uma função global adicional `waitTimeMilli`, que espera um tempo definido em milissegundos.

5 Conclusões

Neste módulo foi implementado o *KeyboardReader* através dos blocos *Key Decode*, *Ring Buffer*, e *Output Buffer*.

O Key Decode descodifica um teclado matricial 4x3 e foi implementado através dos blocos *KeyScan* e *KeyControl*.

O Ring *Buffer* é responsável pelo armazenamento de até 8 palavras de 4 bits em memória, e foi implementado através dos blocos *Ring Buffer Control*, *Memory Address Control*, e *RAM*.

Finalmente, o bloco *Output Buffer* é responsável pela entrega de dados à entidade consumidora, que neste caso é o módulo *Control* de software, e foi implementado através dos blocos *Output Buffer Control* e *Output Register*.

Nesta implementação do bloco *KeyDecode* foi adicionado um divisor de clock por 500000, o que resulta numa frequência de 100 Hz (50MHz / 500000), ou um ciclo a cada 10 milissegundos. Tendo em conta que o varrimento da linha demora um ciclo (registo da saída do PENC no registo de 2 bits), e o varrimento da coluna demora no pior caso 3 ciclos e no melhor caso 0 ciclos, a latência de deteção da tecla é de 10-30 milissegundos.

Na implementação do bloco *Keyboard Reader*, em relação a recursos da placa DE-10 Lite utilizados, são utilizados 157 de 49,670 elementos lógicos (0,32%), 56 registos, e 15 de 360 pinos (4%).

A. Descrição VHDL do bloco *Key Decode*

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY KeyDecode is
PORT( RST: IN std_logic;
              DATA_IN : in std_logic_vector(3 downto 0);
              K : out std_logic_vector(3 downto 0);
              Kval: out std_logic;
              Osc: in std_logic;
              DecE: out std_logic_vector(2 downto 0);
              Kack: in std_logic;
              Skpress: out std_logic;
              Skscan: out std_logic
              );
end KeyDecode;

ARCHITECTURE logicKeyDecode OF KeyDecode is

component KeyScan is
PORT( RST : in std_logic;
              DATA_IN : in std_logic_vector(3 downto 0);
              Kscan : in STD_LOGIC;
              K : out std_logic_vector(3 downto 0);
              Kpress: out std_logic;
              Osc: in std_logic;
              DecE: out std_logic_vector(2 downto 0)
              );
end component;

component KeyControl is
       port (
              RST : in std_logic;
              clk : in std_logic;
              Kpress : in std_logic;
              Kscan : out std_logic;
              Kack : in std_logic;
              Kval : out std_logic
       );
end component;

component CLKDIV is
generic(div: natural := 50000000);
port ( clk_in: in std_logic;
              clk_out: out std_logic);
end component;


signal Sscan, press, sclk: std_logic;
Begin


              divisor: CLKDIV generic map(500000) port map(
                     clk_in => Osc,
                     clk_out => sclk
              );


              scan: KeyScan port map(
                     RST => RST,
                     DATA_IN => DATA_IN,
                     Kscan => Sscan,
                     K => K,
                     Osc => sclk,
                     DecE => DecE,
```

```
                Kpress => press
        );

        Controlo: KeyControl port map(
                RST => RST,
                clk => sclk,
                Kpress => press,
                Kscan => Sscan,
                Kack => Kack,
                Kval => Kval

        );

        Skpress <= press;
        Skscan <= Sscan;
end logicKeyDecode;
```

B. Descrição VHDL do bloco *Ring Buffer*

```
LIBRARY IEEE;
use IEEE.std_logic_1164.all;

entity RingBuffer is
   port(
            D           : in std_logic_vector(3 downto 0);
            clk   : in std_logic;
            RST   : in std_logic;
            DAV   : in std_logic;
            CTS   : in std_logic;
            DAC   : out std_logic;
            Wreg  : out std_logic;
            Q           : out std_logic_vector(3 downto 0)
   );
end RingBuffer;

architecture arq_buffer of RingBuffer is
signal sAddr : std_logic_vector(2 downto 0);
signal full, empty, sWr, putget, incget, incput : std_logic;


      component RingBufferControl is
            port (
                  clk   : in std_logic;
                  RST   : in std_logic;
                  DAV   : in std_logic;
                  CTS   : in std_logic;
                  full  : in std_logic;
                  empty : in std_logic;
                  Wr          : out std_logic;
                  selPG : out std_logic;
                  Wreg  : out std_logic;
                  DAC   : out std_logic;
                  incput: out std_logic;
                  incget: out std_logic
      );
      end component;

component MAC is
   port(
            clk   : in std_logic;
            RST   : in std_logic;
            Addr  : out std_logic_vector(2 downto 0);
            putget: in std_logic;
            incget: in std_logic;
            incput: in std_logic;
            full  : out std_logic;
            empty : out std_logic
   );
end component;

component RAM is
      generic(
            ADDRESS_WIDTH : natural := 3;
```

```
            DATA_WIDTH : natural := 4
      );
      port(
            address : in std_logic_vector(ADDRESS_WIDTH - 1 downto 0);
            wr: in std_logic;
            din: in std_logic_vector(DATA_WIDTH - 1 downto 0);
            dout: out std_logic_vector(DATA_WIDTH - 1 downto 0)
      );
end component;

begin
      FIFO_RAM : RAM port map(
            address     => saddr,
            wr          => sWr,
            din         => D,
            dout        => Q
      );

      MemoryAddressControl : MAC port map(
            clk         => clk,
            RST         => RST,
            Addr        => saddr,
            putget      => putget,
            incget      => incget,
            incput      => incput,
            full        => full,
            empty       => empty
      );

      Control : RingBufferControl port map(
            clk         => clk,
            RST         => RST,
            DAV         => DAV,
            CTS         => CTS,
            full        => full,
            empty       => empty,
            Wr          => sWR,
            selPG       => putget,
            Wreg        => Wreg,
            DAC         => DAC,
            incput      => incput,
            incget      => incget
      );


end architecture;
```

C. Descrição VHDL do bloco *Output Buffer*

```
LIBRARY IEEE;
use IEEE.std_logic_1164.all;

entity OutputBuffer is
   port(
            D     : in std_logic_vector(3 downto 0);
            clk : in std_logic;
            RST : in std_logic;
            Load: in std_logic;
            ACK : in std_logic;
            Obfree : out std_logic;
            Dval  : out std_logic;
            Q     : out std_logic_vector(3 downto 0)
   );
end OutputBuffer;

architecture arq_buffer of OutputBuffer is

signal wReg : std_logic;

component register4 is
PORT( I  : in std_logic_vector(3 downto 0);
            EN : in STD_LOGIC;
            RST: in std_logic;
            CLK: IN STD_LOGIC;
            Q     : out std_logic_vector(3 downto 0)
            );
end component;

component OutBufferControl is
      port (
            clk : in std_logic;
            RST : in std_logic;
            Load: in std_logic;
            ACK : in std_logic;
            Obfree: out std_logic;
            Dval: out std_logic;
            wreg: out std_logic
      );
end component;


begin
      OutputReg : register4 port map(
            I   => D,
            EN  => '1',
            CLK => wReg,
            RST => RST,
            Q => Q
      );

      BufferControl : OutBufferControl port map(
            clk  => clk,
            RST  => RST,
            Load => Load,
```

```
            ACK  => ACK,
            Obfree => Obfree,
            Dval => Dval,
            wreg => wReg
      );

end architecture;
```

D. Atribuição de pinos do módulo *Keyboard Reader*

```
# Altera DE10-Lite board settings
#============================================================
set_global_assignment -name FAMILY "MAX 10 FPGA"
set_global_assignment -name DEVICE 10M50DAF484C6GES
set_global_assignment -name TOP_LEVEL_ENTITY "DE10_Lite"
set_global_assignment -name DEVICE_FILTER_PACKAGE FBGA
set_global_assignment -name SDC_FILE DE10_Lite.sdc
set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE IMAGE WITH ERAM"
#============================================================
# CLOCK
#============================================================
#set_instance_assignment -name IO_STANDARD "3.3-V LVTTL" -to CLOCK_50
#set_instance_assignment -name IO_STANDARD "3.3-V LVTTL" -to CLOCK2_50
#set_instance_assignment -name IO_STANDARD "3.3-V LVTTL" -to CLOCK_ADC_10
set_location_assignment PIN_P11 -to Osc
#set_location_assignment PIN_N14 -to CLOCK2_50
#set_location_assignment PIN_N5 -to CLOCK_ADC_10
#============================================================
# SW
#============================================================
set_location_assignment PIN_C10 -to     ACK
#set_location_assignment PIN_C11 -to SW[1]
#set_location_assignment PIN_D12 -to SW[2]
#set_location_assignment PIN_C12 -to SW[3]
#set_location_assignment PIN_A12 -to SW[4]
#set_location_assignment PIN_B12 -to SW[5]
#set_location_assignment PIN_A13 -to SW[6]
#set_location_assignment PIN_A14 -to SW[7]
#set_location_assignment PIN_B14 -to SW[8]
#set_location_assignment PIN_F15 -to SW[9]
#============================================================
# LED
#============================================================
set_location_assignment PIN_A8 -to Q[0]
set_location_assignment PIN_A9 -to Q[1]
set_location_assignment PIN_A10 -to Q[2]
set_location_assignment PIN_B10 -to Q[3]
#set_location_assignment PIN_D13 -to LEDR[4]
#set_location_assignment PIN_C13 -to LEDR[5]
set_location_assignment PIN_E14 -to Dval
#set_location_assignment PIN_D14 -to LEDR[7]
#set_location_assignment PIN_A11 -to LEDR[8]
#set_location_assignment PIN_B11 -to LEDR[9]
set_location_assignment PIN_W5     -to  Lines[0]
set_location_assignment PIN_AA14   -to  Lines[1]
set_location_assignment PIN_W12    -to  Lines[2]
set_location_assignment PIN_AB12   -to  Lines[3]
set_location_assignment PIN_AB11   -to  Columns[0]
set_location_assignment PIN_AB10   -to  Columns[1]
set_location_assignment PIN_AA9    -to  Columns[2]
#set_location_assignment PIN_AA8    -to KEYPAD_COL[3]
#============================================================
# End of pin and io_standard assignments
#============================================================
```

E. Código *Kotlin - HAL*

```
import isel.leic.UsbPort

/**
 * 22/5/2023
 *
 * Hardware Abstraction Layer (HAL) for the [AccessControlSystem].
 *
 * Provides an interface to interact with the USB Port for reading and writing
bits.
 *
 * @property val_write stores the current value written to the USB port
 *
 * @author Bernardo Pereira
 * @author AntÃ³nio Paulino
 */

object HAL {
    var val_write = 0

    /**
     * Initializes the USB Output Port with the value 0
     */
    fun init() {
        val_write = 0
        UsbPort.write(val_write)
    }


    /**
     * Reads the bits from the USB port and applies [mask] to get the requested
bits
     * @param mask The bit mask used to extract the requested bits.
     * @return The result of applying the mask to the bits read from the USB
port.
     */
    fun readBits(mask: Int): Int = UsbPort.read().and(mask)

    /**
     * Checks if the specified bit by [mask] in the value read from the USB port
is set or not.
     * @param mask The bit mask representing the bit.
     * @return true if the bit is set, false if the bit is not set.
     */
    fun isBit(mask: Int): Boolean = readBits(mask) > 0


    /**
     * Sets the specified bits by [mask] and writes them to the USB port
     * @param mask The bit mask representing the bits to be set.
     *
     */
    fun setBits(mask: Int) {
        val_write = (mask.or(val_write))
        UsbPort.write(val_write)
    }
```

```
    /**
     * Clears the specified bits by [mask] and writes them to the USB port
     * @param mask The bit mask representing the bits to be cleared.
     *
     */
    fun clrBits(mask: Int) {
        val_write = mask.inv().and(val_write)
        UsbPort.write(val_write)
    }

    /**
     * Writes the specified bits by [mask] with [value] to the USB port
     * @param mask The bit mask representing the bits to be written
     * @param value The value to write on the specified bits
     */
    fun writeBits(mask: Int, value: Int) {
        val_write = mask.inv().and(val_write).or(value.and(mask))
        UsbPort.write(val_write)
    }
}
```

F. Código *Kotlin - KBD*

```
fun main() {
    while (true) {
        val key = KBD.getKey()
        if (key != KBD.NONE.toChar()) print(key)
    }
}

/**
 * 22/5/2023
 *
 * Manages the keyboard input for the [AccessControlSystem].
 *
 * @property NONE A constant representing no key pressed. Returned if there was
no key pressed
 * in the [getKey] and [waitKey] functions.
 *
 * @author Bernardo Pereira
 * @author António Paulino
 * @see HAL
 */
object KBD {
    const val NONE = 0

    /**
     * The bit mask for the key valid flag on the Usb input Port
     */
    private const val KVALMASK = 0b00010000

    /**
     * The bit mask for acknowledging the key input on the USB output port.
     */
    private const val KACKMASK = 0b01000000

    /**
     * The bit mask for the key code on the USB input port.
     */
    private const val KCODEMASK = 0b00001111

    /**
     * Stores the current key value.
     */
    private var keycode = NONE

    /**
     * Represents if there is a valid key to be read.
     */
    private var keyval = false

    /**
     * List of characters representing the keyboard matrix.
     */
    private val kbdmatrix = listOf('1', '4', '7', '*', '2', '5', '8', '0', '3',
'6', '9', '#')

    /**
```

```
     * Initializes the keyboard by setting the key valid flag to false and the
key code to NONE.
     */
    fun init() {
        keyval = false
        keycode = NONE
    }

    /**
     * Reads the current key from the keyboard.
     *
     * @return The character corresponding to the pressed key, or [NONE] if no
key is pressed.
     */
    fun getKey(): Char {

        keyval = HAL.isBit(KVALMASK) // Check if a key is pressed


        if (keyval) {

            keycode = HAL.readBits(KCODEMASK) // Read the key

            val key = if (keycode in kbdmatrix.indices) kbdmatrix[keycode] else
NONE.toChar()

            HAL.setBits(KACKMASK) // Send the ack signal flag

            while (keyval) {
                keyval = HAL.isBit(KVALMASK) // Wait for the hardware to lower
the key valid flag
            }

            HAL.clrBits(KACKMASK)   // Clear the ack signal flag

            return key

        } else return NONE.toChar()
    }

    /**
     * Waits for a key input until the timeout is reached.
     *
     * This function sleeps for 50 ms after every key press check if there is no
key pressed.
     * Since the system will spend most of its time waiting for key presses by
the user,
     * this allows the system to use less resources.
     *
     * @param timeout The timeout period in milliseconds.
     * @return The character corresponding to the pressed key, or [NONE] if no
key is pressed within the timeout.
     */
    fun waitKey(timeout: Long): Char {

        val finaltime = System.currentTimeMillis() + timeout
```

```
        var key = getKey()

        while (System.currentTimeMillis() < finaltime && key == NONE.toChar()) {
            key = getKey()
            if (key == NONE.toChar()) waitTimeMilli(CHECK_INTERVAL) //wait if
there was no key pressed

        }

        return key
    }
}
```

G. Código Kotlin – Wait Time

```
import isel.leic.utils.Time


/**
 * 22/5/2023
 *
 * Sleep time for functions that
 * do continuous checks. For example, checking if the door is open/closed, or
checking
 * if there was a key press.
 *
 * Having a sleep time between checks for these functions allows the system to
use less resources.
 *
 *
 */
const val CHECK_INTERVAL = 50

/**
 * Waits a set number of nanoseconds
 * @param time the number of nanoseconds to wait
 */
fun waitTimeNano(time: Int) {
    val endtime = System.nanoTime() + time
    while (System.nanoTime() <= endtime);
}


/**
 * Waits a set number of milliseconds.
 *
 * Used for sleep times and less precise wait times.
 * @param time the number of milliseconds to wait
 */
fun waitTimeMilli(time: Int) {
    Time.sleep(time.toLong())
}
```