

O módulo *Keyboard Reader* é constituído por três blocos principais: i) o decodificador 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.

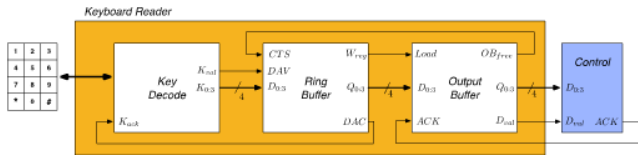
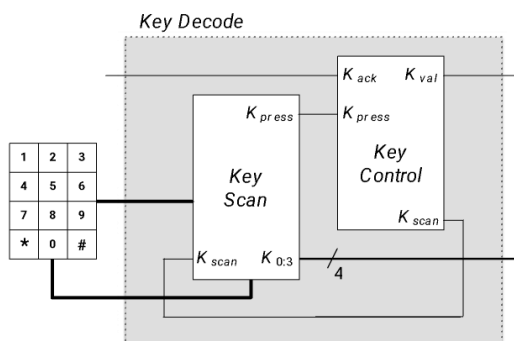


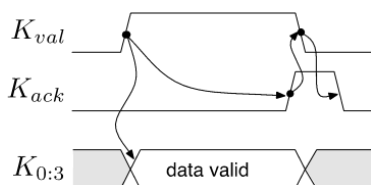
Figura 1 – Diagrama de blocos do módulo *Keyboard Reader*

1 Key Decode

O bloco *Key Decode* implementa um decodificador 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 detectada 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 do 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. Para implementar o bloco *Key Scan* optou-se por escolher a versão 1 presente no guião do trabalho. Após analisar as restantes hipóteses concluímos que a versão 3 é, das três, a mais económica em termos de CLK. Ainda assim, optámos pela versão 1 por uma questão de previsibilidade.

O bloco *Key Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 4. A máquina de estados do bloco *Key Control*, tem como primeiro estado o que denominámos por scanning. Neste primeiro estado é realizado o varrimento do teclado apresentando o sinal *Kscan* ativo. Este sinal só será desativado quando uma tecla for pressionada, ou seja, quando *Kpress* for “true” a máquina de estados passa para o segundo estado, caso contrário o varrimento do teclado continua. No segundo estado, pressing, por ter sido detectada a pressão de uma tecla, o sinal *Kval* é ativado. Neste estado verificámos também se *Kpress* ainda era *true* e, caso fosse, permanecíamos no estado pressing, caso contrário passávamos à verificação do valor do sinal *Kack*. Tendo o *Kpress* a *false* podíamos então verificar o sinal *Kack* que era necessário estar a “true” para podermos passar para o próximo estado, pois no caso de estar a *false* não podia ser iniciado um novo ciclo de varrimento do teclado. Verificando *Kpress* a *false* e *Kack* a “true” podíamos então passar ao terceiro estado. No terceiro estado, waiting, fez-se novamente uma verificação do sinal *Kack*. Caso estivesse a *false* podíamos então iniciar novo ciclo de varrimento do teclado. Optámos pela criação deste terceiro estado para estarmos de acordo com o diagrama temporal do controlo de fluxos da Figura 2b, no qual se verifica a transição do *Kack* para *false* após ter estado a “true”. O estado waiting serve para garantir a sincronização com os outros estados, isto significa que o bloco que envia a informação já sabe que o bloco que a processa terminou o processamento da tecla antiga e está, assim, disponível para processar de novo a informação relativa à nova tecla.

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

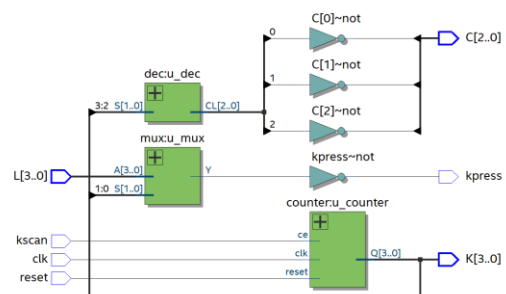


Figura 3 - Diagrama de blocos do bloco *Key Scan*

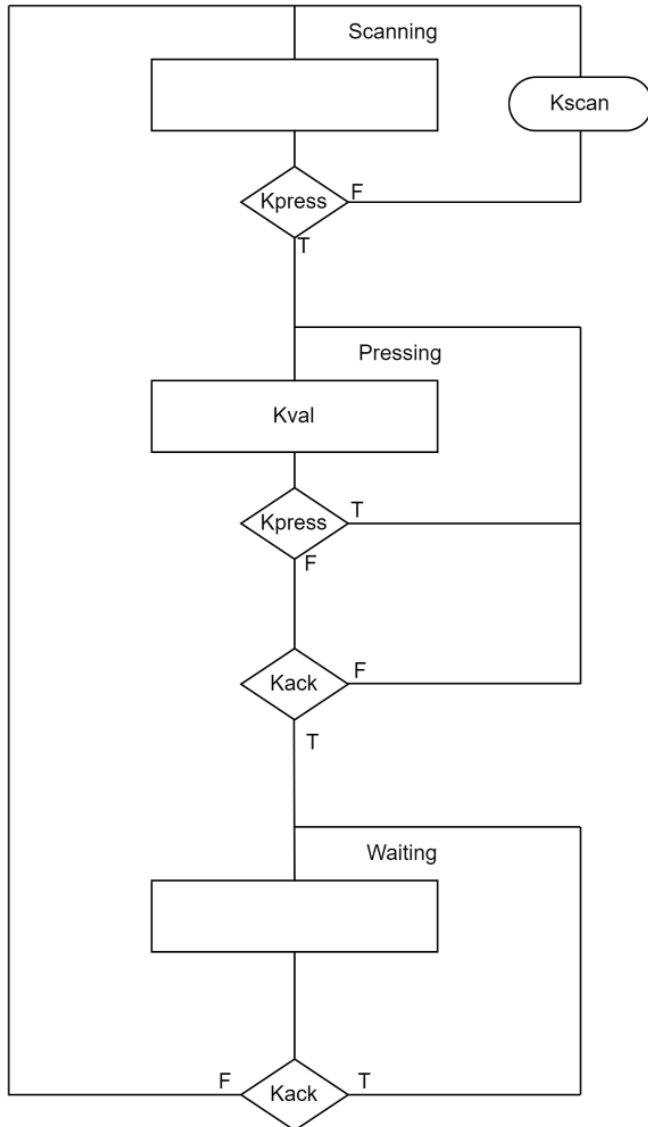


Figura 4 – Máquina de estados do bloco *Key Control*

Com base nas descrições do bloco *Key Decode* implementou-se parcialmente o módulo *Keyboard Reader* de acordo com o esquema elétrico representado no Anexo C. Para o módulo *Key Decode* foi necessário adicionar um *clkdiv* para reduzir a frequência de relógio, devido ao **bouncing**. O **bouncing** acontece quando é premida uma tecla no teclado e a tecla é detectada como premida inúmeras vezes, sendo que na realidade só foi premida uma vez. Por este motivo houve a necessidade de dividir a frequência de relógio proveniente do exterior por 10000 Hz (que neste caso é a frequência da placa, 50MHz). Com esta divisão de frequência o *Key Decode* é lento o suficiente para não detectar que uma tecla foi premida várias vezes, detectando apenas o primeiro pressionar da mesma.

2 Interface com o Control

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem *Kotlin* e seguindo a arquitetura lógica apresentada na Figura 5.

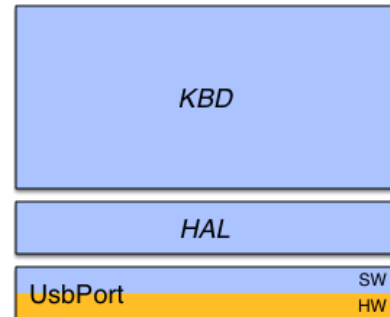


Figura 5 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

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

2.1 HAL

O *HAL* é a camada de software mais próxima do hardware, estabelecendo assim, contacto direto com o *UsbPort*. Como consequência, as funções criadas nesta camada vão ser utilizadas em outros módulos de software. O *HAL* tem como função captar os dados proveniente do *UsbPort* cria-se as funções: *isBit* em que verifica se um bit está ativo ou não, *readBits* que retorna o valor presente no *UsbPort*, *setBits* que coloca no parâmetro *mask* no *UsbPort*, *clrBits* responsável por apagar o valor colocado no *mask* e, por fim, a função *writeBits* que escreve o valor pretendido entre o valor colocado na máscara.

2.2 KBD

O *KBD* tem como objetivo obter o código das teclas que são recebidas como input (*UsbPort*) através do *HAL* (em software). É importante reforçar que o *KBD* interage diretamente com o *HAL* e, por isso, foi preciso obter os bits correspondentes ao código da tecla. Para todo este processo funcionar de forma correta, é preciso primeiro verificar se existe bit (*isBit*) – tecla premida. De seguida é necessário ler os bits correspondentes a essa tecla, guardá-los e ativar o *ACK*. Enquanto a tecla estiver premida o programa fica parado, caso contrário (após a libertação da tecla) o *ACK* é desativado apagando todos os bits e voltando ao valor lógico '0'.

3 Ring Buffer

O bloco *Ring Buffer* é uma estrutura de dados para armazenamento de teclas com disciplina FIFO (*first in first out*), com capacidade de armazenar até oito palavras de 4 bits. Este, é constituído por três sub-blocos: i) o bloco *Ring Buffer Control*; ii) o bloco *Memory Address Control*; e iii) o bloco *RAM*, conforme o diagrama de blocos representado na Figura 6. 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).

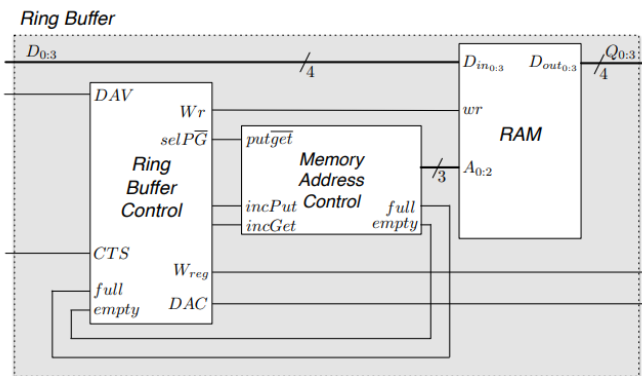


Figura 6 - Diagrama de blocos do bloco *Ring Buffer*

3.1 Ring Buffer Control

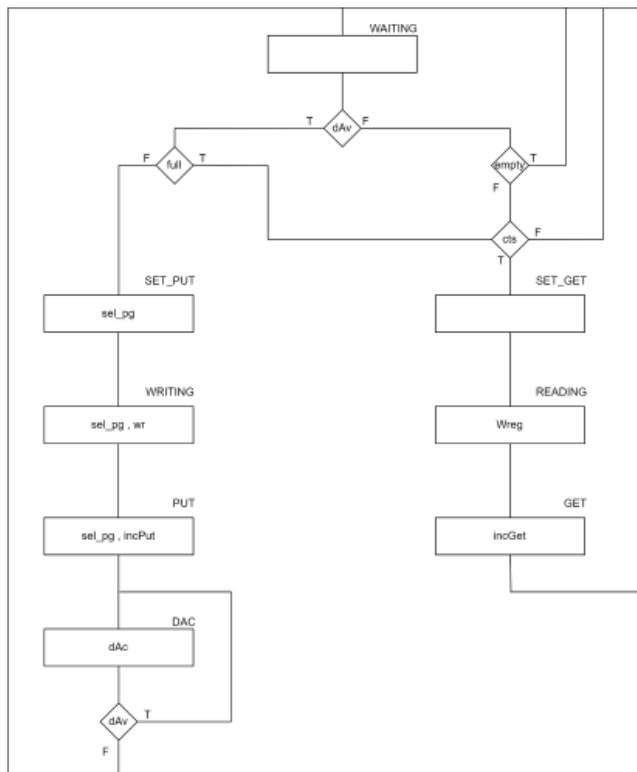


Figura 7 - Máquina de estados do bloco *Ring Buffer Control*

O bloco *Ring Buffer Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 7. A máquina de estados do bloco *Ring Buffer*, tem como primeiro estado o estado *WAITING*. No estado inicial, é realizada a verificação do sinal *dAv*. Se *dAv* estiver ativo, significa que houve uma tecla pressionada e por isso é necessário verificar se o código da tecla pode ser guardado. Para isso é preciso verificar o sinal *full*. Caso apresente o valor lógico 0 a tecla pode então ser guardada, indo assim para o estado *SET_PUT*. Caso contrário, iremos verificar o sinal *cts*. Caso esteja com o valor lógico 1 entramos no estado *SET_GET*, caso contrário voltamos ao estado inicial. Durante o estado *SET_PUT* a saída *sel_pg* está ativa para indicar ao módulo *MAC* que é necessário fazer a instrução *incPut*. De seguida é aplicado um clock, que nos obriga a ir para o estado *WRITING*. Neste estado, a saída *wr* ativa para habilitar a escrita na *RAM*, desta forma, o sinal *sel_pg* permanece ativo para garantir que o comando *incPut* seja executado. Ao fim de mais um clock a máquina de estados encontra-se no estado *PUT*, onde a saída *incPut* é ativada para incrementar o endereço do *idxPut* e é necessário que o *sel_pg* continue ativo, pois caso este esteja desativado, é escolhido no *MUX* o *idxGet* (o que não é pretendido). De seguida encontra-se o estado *DAC* onde é ativado o sinal *dAc* significando que a escrita em memória foi concluída. Este sinal só é desativado depois de o *dAv* ser desativado, voltando assim para o estado inicial. Caso o sinal *dAv* esteja com o valor lógico '0' significa que não há nenhuma tecla a ser pressionada, ou seja, pretende-se ler uma tecla da *RAM*. Desta forma, será verificado o valor do sinal *empty*. Caso este tenha o valor lógico '1' retornamos ao estado inicial, caso esteja a '0' é verificado de imediato o valor do sinal *cts*. O sinal *cts*, ao ter valor lógico '0', obriga a retornar ao estado inicial, caso contrário avança-se para o estado *SET_GET*, que é um estado vazio, cujo objetivo é escolher o *get* do *sel_pg*. Posteriormente, no estado *READING*, o sinal *Wreg* está ativo para indicar à *RAM* que pretende ler uma tecla. Para finalizar, no estado *GET* é ativado o sinal *incGet* para incrementar o valor do *idxGet*. No próximo clock, a máquina de estados volta ao estado inicial.

3.2 Memory Address Control

O bloco *MAC - Memory Address Control*, é composto por dois registos, que contêm o endereço de escrita e leitura, *idxPut* e *idxGet* respetivamente. Desta forma, vai ser possível suportar ações de *incPut* e *incGet* de forma a saber se a estrutura de dados está *full* ou *empty*. Para a construção deste bloco foi necessário recorrer à implementação de vários sub-blocos: i) os blocos contador *idxPut* - cada vez que o *incPut* é ativado ele vai incrementar o endereço onde a tecla vai ser guardada, e contador *idxGet* - cada vez que o *incGet* é ativado ele vai ler e incrementar o endereço da tecla; ii) o bloco *mux* - tem como objetivo selecionar leitura ou escrita, e iii) o bloco *counter* - tem como objetivo informar se as 8 teclas já foram obtidas ou não de forma a saber se está *full* ou *empty*.

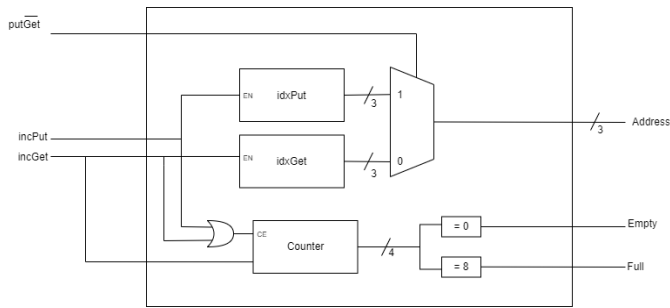


Figura 8 - Diagrama de blocos do *Memory Address Control*

4 Output Buffer

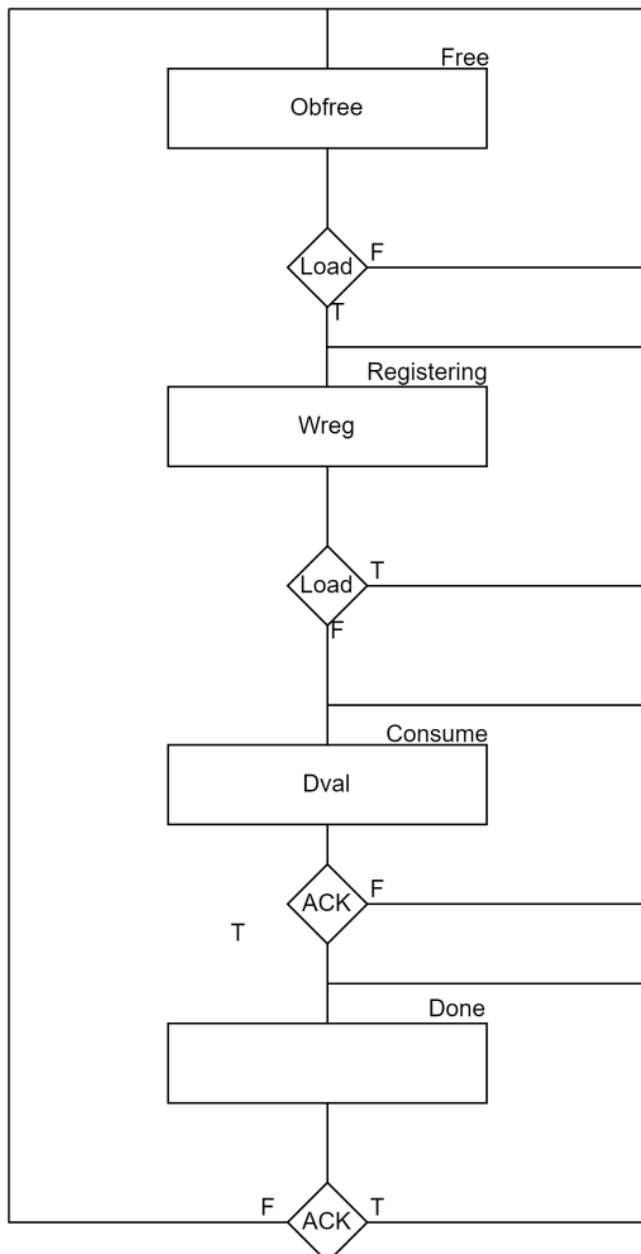


Figura 9 - Máquina de estados do Buffer Control

O bloco *Output Buffer* do *Keyboard Reader* é responsável pela interação com o módulo *Control* (sistema consumidor).

É composto por um módulo *Buffer Control* que controla o processo de passagem de dados e um bloco *Output Register* que regista os dados.

Quando o *Ring Buffer* tenha dados disponíveis e o bloco *Output Buffer* esteja disponível o *Ring Buffer* realiza uma leitura de memória e entrega os dados ao *Output Buffer*. Nesse momento inicia-se o processo que é explicado a seguir:

Para construirmos este processo, recorreu-se a uma máquina de estados para caracterizar o módulo *Buffer Control*. Primeiramente a saída *Obfree* está ativa no primeiro estado (*Free*), esta saída fica ativa para indicar que está disponível para armazenar dados e mantém-se neste estado até que o sistema produtor ative o sinal *Load* para registar os dados. Quando *Load* passa para valor lógico 1, passamos para o segundo estado (*Registering*), momento em que é feito o registo dos dados. No terceiro estado (*Consume*), a saída *Dval* fica ativa para "avisar" o *Control* que pode ler os dados pretendidos do *Output Buffer*. Depois do *Control* recolher os dados o sinal *ACK* ativa para indicar que os dados já foram consumidos. No quarto e último estado (*Done*), o sinal *Dval* passa para valor lógico 0 e faz-se uma "sincronização" para o *ACK* voltar a 0 e assim poder-se voltar ao estado inicial e sinalizar que está novamente disponível para entregar dados ao sistema consumidor.

5 Conclusões

Concluindo, o módulo *Keyboard Reader* é capaz de detetar e guardar em memória a linha e a coluna da respectiva tecla pressionada, facultando quatro bits de código. A implementação escolhida tem no pior caso 15 ciclos de clock como latência da tecla. Para implementar o módulo *Keyboard Reader* necessitamos de um *UsbPort*, um teclado matricial de 4x3 e uma máquina capaz de executar a parte de software.

A. Descrição VHDL do bloco *Key Decode*

```
LIBRARY ieee;
USE ieee.STD_LOGIC_1164.ALL;

entity key_decode is
  port (
    Mclk : in std_logic;
    reset : in std_logic;
    Kack : in std_logic;
    Lines : in std_logic_vector (3 downto 0);    -- linhas
    Columns : out std_logic_vector (2 downto 0); -- colunas
    K : out std_logic_vector (3 downto 0);
    Kval : out std_logic
  );
end key_decode;

architecture arq of key_decode is
  component key_scan is port (
    kscan : in std_logic;
    clk : in std_logic;
    reset : in std_logic;
    L : in std_logic_vector (3 downto 0);    -- linhas
    K : out std_logic_vector (3 downto 0);
    C : out std_logic_vector (2 downto 0);    -- colunas
    kpress : out std_logic
  );
end component;

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

  component clkdiv generic (div: NATURAL := 25000000);
  port (
    clk_in : in std_logic;
    clk_out : out std_logic
  );
end component;

  signal clks, kscan_s, kpress_s : std_logic;

begin
  u_key_scan: key_scan port map (
    kscan => kscan_s,
    clk => clks,
    reset => reset,
    L => Lines,
    K => K,
    kpress => kpress_s,
    C => Columns
  );

  u_key_control: key_control port map (
    clk => clks,
    Kack => Kack,
    Kpress => kpress_s,
    reset => reset,
    Kval => kval,
    Kscan => kscan_s
  );

  u_clk : clkdiv generic map (10000) port map (
    clk_in => Mclk,
    clk_out => clks
  );

end arq;
```

A. 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);
    DAV : in std_logic;
    CTS : in std_logic;
    Q : out std_logic_vector (3 downto 0);
    Wreg : out std_logic;
    DAC : out std_logic;
    clk : in std_logic;
    reset : in std_logic
);
end ringbuffer;

architecture arq of ringbuffer is

    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;

    component mac is port (
        put_notGet : in std_logic;
        clk : in std_logic;
        reset : in std_logic;
        incPut : in std_logic;
        incGet : in std_logic;
        address : out std_logic_vector (2 downto 0);
        empty : out std_logic;
        full : out std_logic
    );
    end component;

    component ring_controller is port (
        dAv : in std_logic;
        empty : in std_logic;
        full : in std_logic;
        cts : in std_logic;
        clk : in std_logic;
        reset : in std_logic;
        wr : out std_logic;
        sel_pg : out std_logic;
        incPut : out std_logic;
        incGet : out std_logic;
        wReg : out std_logic;
        dAc : out std_logic
    );
    end component;
```

```
signal wr_s, sel_pg_s, incPut_s, incGet_s : std_logic;
signal empty_s, full_s: std_logic;
signal address_s: std_logic_vector (2 downto 0);

begin
u_ram : RAM port map (
    din => D,
    wr => wr_s,
    address => address_s,
    dout => Q
);

u_mac: mac port map(
    put_notget => sel_pg_s,
    incPut => incPut_s,
    incGet => incGet_s,
    full => full_s,
    empty => empty_s,
    clk => clk,
    reset => reset,
    address => address_s
);

bu_ring_controller: ring_controller port map (
    dAv => DAV,
    CTS => CTS,
    full => full_s,
    empty => empty_s,
    DAC => DAC,
    wreg => wreg,
    incPut => incPut_s,
    incGet => incGet_s,
    sel_pg => sel_pg_s,
    wr => wr_s,
    reset => reset,
    clk => clk
);

end arq;
```

B. Descrição VHDL do bloco *Output Buffer*


```
LIBRARY ieee;
USE ieee.STD_LOGIC_1164.ALL;

entity outputbuffer is port (
    Load    : in std_logic;
    ACK      : in std_logic;
    D        : in std_logic_vector (3 downto 0);
    Q        : out std_logic_vector (3 downto 0);
    OBfree   : out std_logic;
    Dval     : out std_logic;
    clk      : in std_logic;
    reset    : in std_logic
);

end outputbuffer;

architecture arq of outputbuffer is

    component buffercontroller is port (
        Load    : in std_logic;
        ACK      : in std_logic;
        Wreg     : out std_logic;
        OBfree   : out std_logic;
        Dval     : out std_logic;
        clk      : in std_logic;
        reset    : in std_logic
    );
    end component;

    component reg is port (
        F : in std_logic_vector(3 downto 0);
        CE, reset : in std_logic;
        clk : in std_logic;
        Q : out std_logic_vector(3 downto 0)
    );
    end component;

    signal s_Wreg : std_logic;

begin

    u_BufferControl: buffercontroller port map (
        Load => Load,
        ACK => ACK,
        Wreg => s_Wreg,
        OBfree => OBfree,
        Dval => Dval,
        clk => clk,
        reset => reset
    );

    u_OutputRegister: reg port map (
        F => D,
        Q => Q,
        clk => clk,
        CE => s_Wreg,
        reset => reset
    );

end arq;
```


C. Atribui o de pinos do m dulo *Keyboard Reader*

```
set_location_assignment PIN_P11 -to Mclk
```

```
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_A8 -to Kval  
set_location_assignment PIN_A9 -to K[0]  
set_location_assignment PIN_A10 -to K[1]  
set_location_assignment PIN_B10 -to K[2]  
set_location_assignment PIN_D13 -to K[3]
```

```
set_location_assignment PIN_C10 -to reset  
set_location_assignment PIN_C11 -to Kack
```

D. Código Kotlin - HAL

```
import isel.leic.UsbPort

object HAL { // Virtualiza o acesso ao sistema UsbPort// o hall os sabe os sinais ativos
    // Inicia a classe
    var state = 0

    fun init() {
        UsbPort.write(state)
    }
    // Retorna true se o bit tiver o valor lógico '1'
    fun isBit(mask: Int): Boolean = mask and UsbPort.read() == mask

    // 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
    fun writeBits(mask: Int, value: Int) {
        val valueUnderMask = mask and value
        val outMaskOn = mask.inv() and state

        val result = valueUnderMask or outMaskOn
        UsbPort.write(result)
        state = result
    }

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

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

fun main() {
    while(true) {
        val x = HAL.isBit(0x09)
        if (x) println("hey")
    }
}
```

E. Código Kotlin - KBD

```
import isel.leic.utils.Time

object KBD { // Ler teclas. Métodos retornam '0'..'9', '#', '*' ou NONE.
    const val DVAL_MASK = 0X01
    const val KACK_MASK=0X01
    const val LEDS_MASK=0x1E
    const val NONE = 0.toChar()
    private val values= charArrayOf('1','4', '7', '*', '2','5','8','0','3','6','9','#', NONE,
    NONE, NONE, NONE)

    // Inicia a classe
    fun init() {
        HAL.init()
        HAL.clrBits(KACK_MASK)
    }

    // Retorna de imediato a tecla premida ou NONE se não há tecla premida.
    fun getKey(): Char { // working
        if (HAL.isBit(DVAL_MASK)) {
            val value = HAL.readBits(LEDS_MASK).shr(1)

            HAL.setBits(KACK_MASK) // set bit
            while (HAL.isBit(DVAL_MASK)) {
            }
            HAL.clrBits(KACK_MASK)
            return values[value]
        }
        return NONE
    }

    // Retorna a tecla premida, caso ocorra antes do 'timeout' (representado em milissegundos), ou
    NONE caso contrário.
    fun waitKey(timeout: Long): Char {
        val StartTime = Time.getTimeInMillis()
        while (Time.getTimeInMillis() - StartTime < timeout) {
            val pressedKey = getKey()
            if (pressedKey != NONE) return pressedKey
        }
        return NONE
    }
}

fun main(){
    while (true){
        val k= KBD.waitKey(10000)
        println(k)
    }
}
```