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.

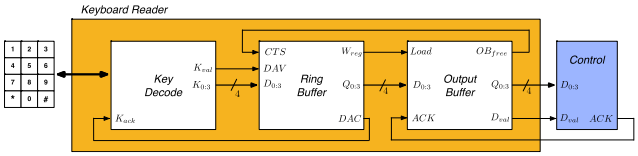
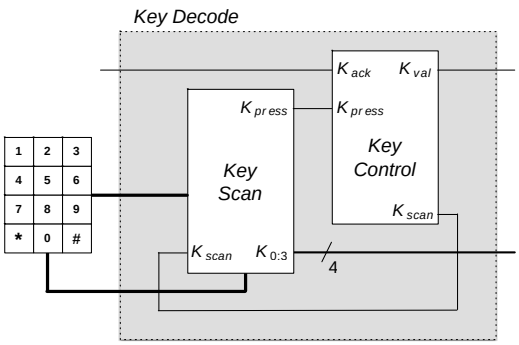


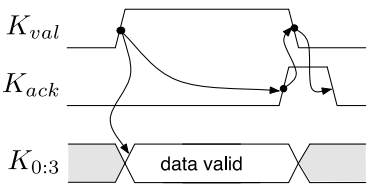
Figura 1 – Diagrama de blocos do módulo *Keyboard Reader*

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. Esta versão foi escolhida devido a ter sido a primeira versão a ser estudada, feita e testada com sucesso, pelo que para esta avaliação intercalar não faria sentido mudar. Na discussão final, iremos defender o porquê de usarmos esta versão.

O bloco *Key Control* foi implementado pela máquina de estados representada em *ASM-chart* na figura 4. Para este *ASM-chart*, a forma que ele foi concebido foi a seguinte: no primeiro estado (*STATE_SCANNING*) este estado irá estar a fazer o varrimento do teclado, ou seja, terá de ter *Kscan* ativo (esta ativo pois tem de deixar o Contador do *Key Scan* contar) e vai estar a espera que seja premida alguma tecla do teclado (*Kpress* estar ativo) para que esteja pronto para passar para o estado seguinte. No segundo estado (*STATE_SEARCHING*) temos *Kval* ativo (ativado pois é detetada a pressão de uma tecla) e esperamos que o sinal *Kack* seja ativo para que saibamos que a tecla que o utilizador pressionou está válida. Assim que *Kack* estiver ativo iremos passar para o terceiro estado *(STATE_WAITING*) em que consiste em saber qual tecla o utilizador pressionou e esperar que possamos voltar a ler mais uma tecla. Para isso iremos esperar que o *Kack* não esteja ativo e que *Kpress* também não esteja ativo, o que significa que a tecla já foi lida e que podemos voltar a ler outra.

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

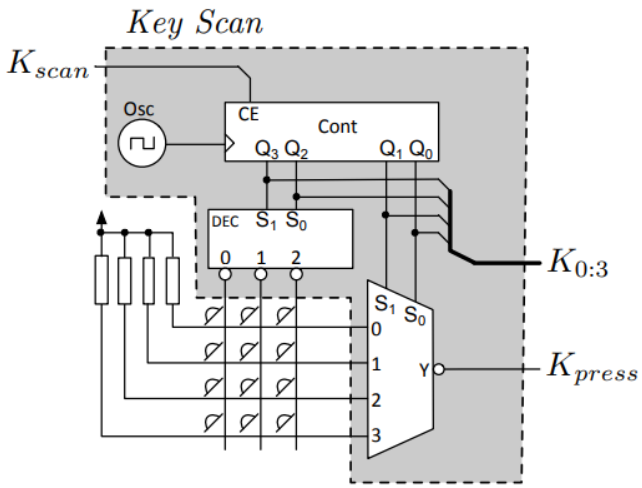


Figura 3 - Diagrama de blocos do bloco *Key Scan*

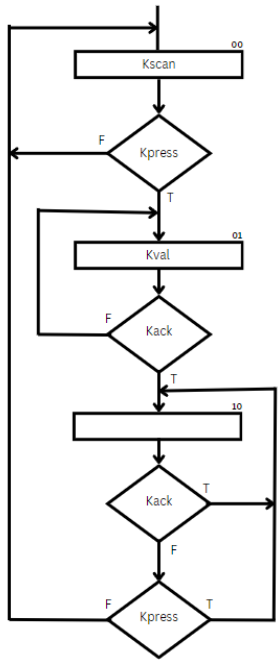


Figura 4 – Máquina de estados do bloco *Key Control*

2 Ring Buffer

O bloco *Ring Buffer* implementa 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 $D_{0:3}$ 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 $put\overline{get}$, 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 5 é apresentado o diagrama de blocos para a estrutura do bloco *Ring Buffer*.

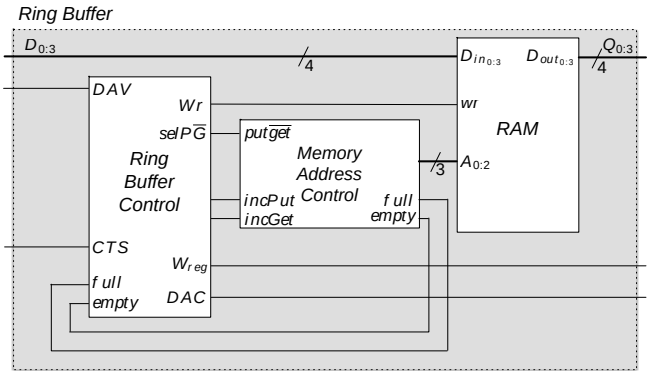


Figura 5 - Diagrama de blocos do bloco *Ring Buffer*

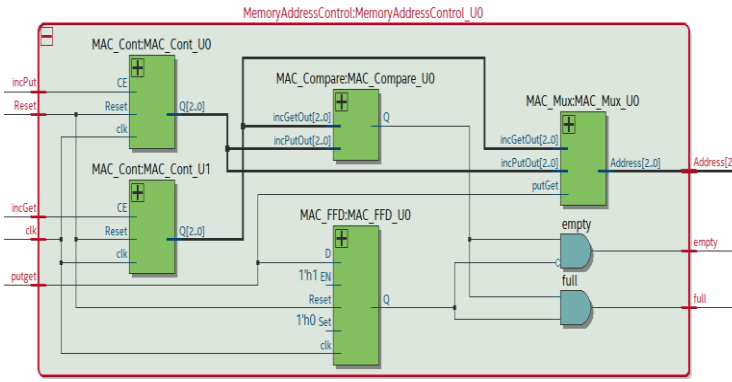


Figura 6 - Diagrama de blocos do bloco *MAC*

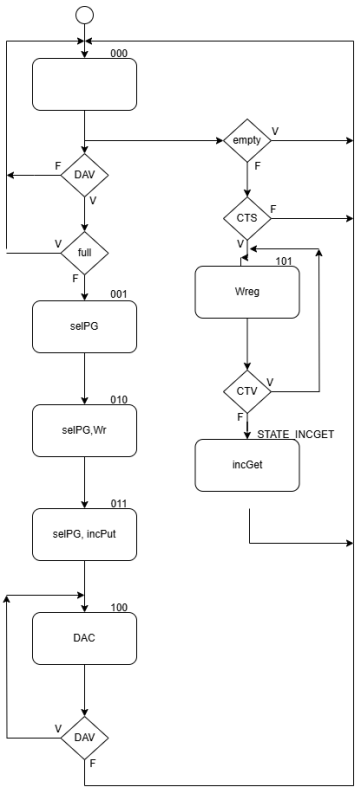


Figura 7 – Máquina de estados do bloco *Ring Buffer Control*

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 D_{val} 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 D_{val} e sinalizar que está novamente disponível para entregar dados ao sistema consumidor, ativando o sinal OB_{free}. Na figura 8, é apresentado o diagrama de blocos do *Output Buffer*.

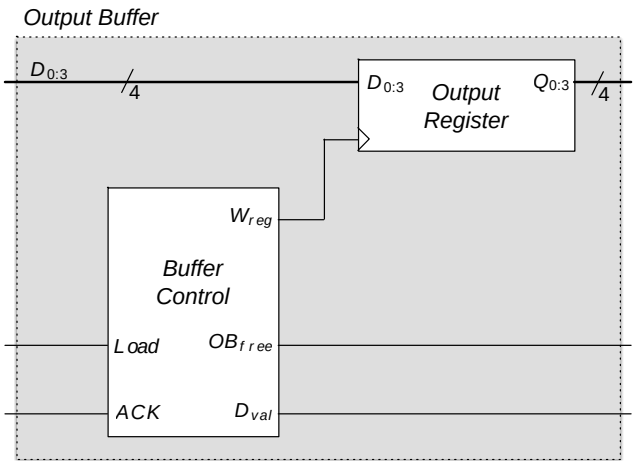


Figura 8 – Diagrama de blocos do *Output Buffer*

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 W_{reg}. O *Output Buffer* indica que já registou os dados desativando o sinal OB_{free}.
O bloco *Buffer Control* foi implementado de acordo com o ASM-chart representado na Figura 9. Para este componente, iremos ter como sinais de entrada o *Load* e o *ACK*. O *Load* tem como função indicar se temos dados para escrever e o *ACK* indica-nos se os dados já foram escritos ou não. Também teremos como saídas o *OBfree*, que indica para registar dados e *Dval*, que envia os dados para serem escritos. Esta máquina de estados tem de estar *FI* (*FullInterlock)* com a máquina do *RingBuffer Control*, para que elas estejam interligadas entre si e para que haja sintonia.

A descrição hardware do bloco *Buffer Control* em *VHDL* encontra-se no Anexo J.

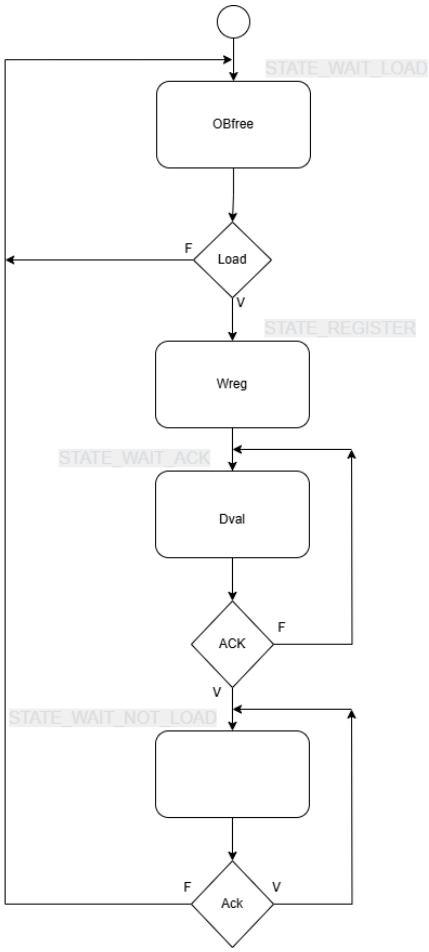


Figura 9 - Máquina de estados do bloco *Buffer Control*

Com base nas descrições do bloco *Key Decode*, *Ring Buffer* e do bloco *Key Buffer Control* implementou-se o módulo *Keyboard Reader* de acordo com o a descrição *VHDL* apresentada no Anexo A. Para este componente, tivemos de fazer a concatenação dos blocos *Key Decode*, *Ring Buffer* e *Output Buffer* e tivemos de ter em atenção cada máquina de estados para que cada uma pudesse estar *FI* (*FullInterlock*) entre si.

4 Interface com o *Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem Java e seguindo a arquitetura lógica apresentada na Figura 10.

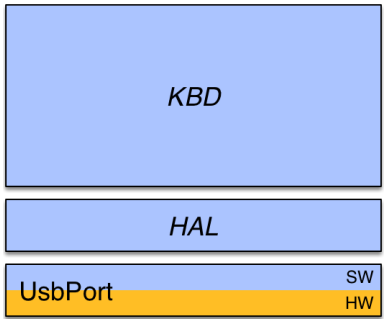


Figura 10 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

As classes *HAL* e *KBD* desenvolvidas são descritas nas secções 4.1 e 4.2, e o código fonte desenvolvido nos Anexos L e M, respetivamente.

4.1 Classe *HAL*

Este Módulo tem como objetivo escrever e ler os bits no *UsbPort* para isso desenvolvemos as seguintes funções.

Init() - inicia o modulo e permite começar a leitura e escrita.

isBit(mask: Int) - retorna true se o bit tiver o valor lógico '1'.

readBits(mask: Int) - lê os bits no usbport usando a máscara como "filtro".

writeBits(mask: Int, value: Int) - escreve no usbport o value usando a mask.

setBits(mask: Int) - coloca os bits do usbport a '1' de acordo com a máscara.

clrBits(mask: Int) - coloca os bits do usbport a '0' de acordo com a máscara.

4.2 Classe *KBD*

Este Módulo tem como objetivo ler as teclas premidas na placa, para isso desenvolvemos as seguintes funções.

init() – limpamos o ubsport para fazer a leitura dos valores.

getKey() – retorna a tecla premida ou none.

waitKey(timeout: Long) – retorna a Tecla premida se ocorrer antes do timeout(milissegundos), se não, retorna none.

5 Conclusões

O módulo Keyboard Reader é composto por três blocos principais: Key Decode, Ring Buffer e Output Buffer.

1. *Key Decode*: Descodifica as entradas de um teclado matricial 4x3, composto por *Key Scan* (varrimento do teclado), *Key Control* (controle do fluxo de varrimento e sinais) e o teclado matricial. Utiliza sinais como *Kval* e *Kack* para garantir a leitura correta e sincronizada das teclas.
2. *Ring Buffer*: Implementa um *buffer FIFO* que armazena até oito palavras de quatro *bits*, garantindo a ordem correta das teclas pressionadas. Comunica com o *Key Decode* através dos sinais *DAV* e *DAC* e controla a leitura e escrita usando os registos *putIndex* e *getIndex* monitorados pelo *MAC* (*Memory Address Control*).
3. *Output Buffer*: Facilita a interação com o módulo *Control* em *software*, gerindo a disponibilidade e validação de dados com sinais *OBfree* e *Dval*, permitindo uma leitura eficiente das teclas pelo *Control*.

A implementação em *VHDL* assegura a interligação e sincronização dos blocos, enquanto a comunicação com o Control é feita via software em Java (kotlin). A classe *HAL* lida com a leitura e escrita na porta *USB* e a classe *KBD* facilita a leitura das teclas.

Finalizando, o módulo *Keyboard Reader* foi projetado para garantir uma leitura precisa e eficiente das teclas, armazenando-as corretamente e disponibilizando-as ao consumidor de forma sincronizada, essencial para aplicações que requerem precisão na captura de entradas de teclado.

A. Descrição *VHDL* do bloco *Keyboard Reader*

```
library ieee;
use ieee.std_logic_1164.all;

entity KeyboardReader is
port(
Line : in std_logic_vector(3 downto 0);
Reset, clk, ACK: in std_logic;
Dval : out std_logic;
Col : out std_logic_vector(2 downto 0);
Q : out std_logic_vector(3 downto 0)
);
end KeyboardReader;

architecture behavioral of KeyboardReader is

component KeyDecode is
port(
Reset: in std_logic;
clk: in std_logic;
Line: in std_logic_vector(3 downto 0);
Kack: in std_logic;
Col: out std_logic_vector(2 downto 0);
Kval: out std_logic;
K: out std_logic_vector(3 downto 0)
);
end component;

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

component OutputBuffer is
port(
D : in std_logic_vector(3 downto 0);
Load, Reset, ACK, clk: in std_logic;
Q : out std_logic_vector(3 downto 0);
OBfree, Dval : out std_logic
);
end component;

signal term0, term1, term3, term5 : std_logic;
signal term2, term4 : std_logic_vector(3 downto 0);

begin

KeyDecode_U0: KeyDecode port map(
Reset => Reset,
clk => clk,
Line => Line,
Kack => term0,
Col => Col,
Kval => term1,
K => term2
```

```
);

RingBuffer_U0: RingBuffer port map(
DAV => term1,
CTS => term3,
clk => clk,
Reset => Reset,
D => term2,
Q => term4,
Wreg => term5,
DAC => term0
);

OutputBuffer_U0: OutputBuffer port map(
D => term4,
Load => term5,
Reset => Reset,
ACK => ACK,
clk => clk,
Q => Q,
OBfree => term3,
Dval => Dval
);

end behavioral;
```

B. Descricao VHDL do bloco KeyDecode

```
library ieee;
use ieee.std_logic_1164.all;

entity KeyDecode is
port(
        Reset: in std_logic;
        CLK: in std_logic;
        Linhas: in std_logic_vector(3 downto 0);
        Kack: in std_logic;
        Col: out std_logic_vector(2 downto 0);
        Kval: out std_logic;
        K: out std_logic_vector(3 downto 0)
);
end KeyDecode;

architecture behavioral of KeyDecode is

component KeyScan is
port(
        Kscan: in std_logic;
        Linhas: in std_logic_vector(3 downto 0);
        clk: in std_logic;
        Reset: in std_logic;
        K: out std_logic_vector(3 downto 0);
        Kpress: out std_logic;
        Col: out std_logic_vector(2 downto 0)
);
end component;

component CLKdiv is
generic(div: natural := 50000000);
port ( clk_in: in std_logic;
```

```
                clk_out: out std_logic);
end component;

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

signal term0, term1, not_clk, virtualclk: std_logic;

begin

not_clk <= not virtualclk;

U0 : clkDIV generic map (10000) port map(
clk_in => clk,
clk_out => virtualclk
);

KeyScan_Unit0: KeyScan port map(
        Kscan => term0,
        Linhas => Linhas,
        CLK => virtualCLK,
        Reset => Reset,
        K => K,
        Kpress => term1,
        Col => Col
);

KeyControl_Unit0: KeyControl port map(
        Reset => Reset,
        CLK => not_clk,
        Kack => Kack,
        Kpress => term1,
        Kval => Kval,
        Kscan => term0
);


end behavioral;
```

C. Descrição VHDL do bloco KeyControl

```
library ieee;
use ieee.std_logic_1164.all;

entity KeyControl is
port(
        Reset: in std_logic;
        CLK: in std_logic;
        Kack: in std_logic;
        Kpress: in std_logic;
        Kval: out std_logic;
        Kscan: out std_logic
);
```

```
end KeyControl;

architecture behavioral of KeyControl is

type STATE_TYPE is (STATE_SCANNING, STATE_SEARCHING, STATE_WAITING);

signal CurrentState, NextState : STATE_TYPE;

begin

-- Flip-Flop's
CurrentState <= STATE_SCANNING when Reset = '1' else NextState when rising_edge(clk);

-- Generate Next State
GenerateNextState:
process (CurrentState, Kack, Kpress)
	begin
		case CurrentState is
			when STATE_SCANNING   => if (Kpress = '1') then
							NextState <= STATE_SEARCHING;
						else NextState <= STATE_SCANNING;
					end if;

			when STATE_SEARCHING    => if (Kack = '1') then
							NextState <= STATE_WAITING;
						else NextState <= STATE_SEARCHING;
						end if;

			when STATE_WAITING    => if(Kack = '0' and Kpress = '0') then
						NextState <= STATE_SCANNING;
							else NextState <= STATE_WAITING;
						end if;

	end case;
end process;

-- Generate Outputs
Kscan <= '1' when (CurrentState = STATE_SCANNING) else '0';

Kval <= '1' when (CurrentState = STATE_SEARCHING) else '0';


end behavioral;
```

D. Descrição VHDL do bloco KeyScan

```
library ieee;
use ieee.std_logic_1164.all;

entity KeyScan is
port(
	Kscan: in std_logic;
	Linhas: in std_logic_vector(3 downto 0);
	clk: in std_logic;
	Reset: in std_logic;
	K: out std_logic_vector(3 downto 0);
	Kpress: out std_logic;
	Col: out std_logic_vector(2 downto 0)
);
end KeyScan;
```

```
architecture behavioral of KeyScan is

component KeyScan_Cont is
port(
	CE, Reset, clk: in std_logic;
	Q: out std_logic_vector(3 downto 0)
);
end component;

component KeyScan_Decoder is
port(
	S: in std_logic_vector(1 downto 0);
	Col: out std_logic_vector(2 downto 0)
);
end component;

component KeyScan_Mux is
port(
	Linhas: in std_logic_vector(3 downto 0);
	S: in std_logic_vector(1 downto 0);
	Y: out std_logic
);
end component;

signal term: std_logic_vector(3 downto 0);

begin

KeyScan_Cont_U0: KeyScan_Cont port map(
	CE => Kscan,
	Reset => Reset,
	clk => clk,
	Q => term
);
KeyScan_Decoder_U0: KeyScan_Decoder port map(
	S(0) => term(2),
	S(1) => term(3),
	Col => Col
);
KeyScan_Mux_U0: KeyScan_Mux port map(
	Linhas => Linhas,
	S(0) => term(0),
	S(1) => term(1),
	Y => Kpress
);

K <= term;

end behavioral;
```

E. Descrição VHDL do bloco Ring Buffer

```
library ieee;
use ieee.std_logic_1164.all;

entity RingBuffer is
port(
	DAV, CTS, clk, Reset: in std_logic;
	D: in std_logic_vector(3 downto 0);
	Q: out std_logic_vector(3 downto 0);
```

```
        Wreg, DAC: out std_logic
);
end RingBuffer;

architecture behavioral of RingBuffer is

component RingBufferControl port(
        Reset, clk, DAV, CTS, full, empty: in std_logic;
        Wr, selPG, Wreg, DAC, incPut, incGet: out std_logic
);
end component;

component MemoryAddressControl port(
        putget, incPut, incGet, Reset, clk: in std_logic;
        full, empty: out std_logic;
        Address: out std_logic_vector(2 downto 0)
);
end component;

component RAM
        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;

signal term0, term1, term2, term3, term4, term5: std_logic;
signal term6: std_logic_vector(2 downto 0);
begin

RingBufferControl_U0: RingBufferControl port map(

        Reset => Reset,
        clk => clk,
        DAV => DAV,
        CTS => CTS,
        full => term0,
        empty => term1,
        Wr => term2,
        selPG => term3,
        Wreg => Wreg,
        DAC => DAC,
        incPut => term4,
        incGet => term5
);

MemoryAddressControl_U0: MemoryAddressControl port map(
        putget => term3,
        incPut => term4,
        incGet => term5,
        Reset => Reset,
        clk => clk,
        full => term0,
```

```
        empty => term1,
        Address => term6
);

RAM_U0: RAM port map(
        address => term6,
        wr => term2,
        din => D,
        dout => Q
);

end behavioral;
```

F. Descrição VHDL do bloco Ring Buffer Control

```
library ieee;
use ieee.std_logic_1164.all;

entity RingBufferControl is
port(
        Reset, clk, DAV, CTS, full, empty: in std_logic;
        Wr, selPG, Wreg, DAC, incPut, incGet: out std_logic
);
end RingBufferControl;

architecture behavioral of RingBufferControl is

type STATE_TYPE is (STATE_000, STATE_001, STATE_010, STATE_011, STATE_100, STATE_101,
STATE_INCGET);

signal CurrentState, NextState : STATE_TYPE;

begin

-- Flip-Flop's
CurrentState <= STATE_000 when Reset = '1' else NextState when rising_edge(clk);

-- Generate Next State
GenerateNextState:
process (CurrentState, DAV, CTS, full, empty)
        begin
                case CurrentState is
                        when STATE_000 => if (DAV = '1' and full = '0') then NextState <= STATE_001;
                                                                elsif (empty = '0' and CTS = '1') then
NextState <= STATE_101;
                                                                        else NextState <=
STATE_000;
                                                                                end if;

                        when STATE_001 => NextState <= STATE_010;

                        when STATE_010 => NextState <= STATE_011;

                        when STATE_011 => NextState <= STATE_100;

                        when STATE_100 => if (DAV = '0') then NextState <= STATE_000;
                                                                else NextState <= STATE_100;
                                                                        end if;
```

```
                when STATE_101 => if (CTS = '0') then NextState <= STATE_INCGET;
                                                      else NextState <= STATE_101;
                                                           end if;

                when STATE_INCGET => NextState <= STATE_000;
        end case;
end process;

Wr <= '1' when (CurrentState = STATE_010) else '0';
selPG <= '1' when (CurrentState = STATE_001 or CurrentState = STATE_010 or CurrentState = STATE_011) else '0';
Wreg <= '1' when (CurrentState = STATE_101) else '0';
DAC <= '1' when (CurrentState = STATE_100) else '0';
incPut <= '1' when (CurrentState = STATE_011) else '0';
incGet <= '1' when (CurrentState = STATE_INCGET) else '0';

end behavioral;
```

G. Descrição VHDL do bloco Memmory Address Control

```
library ieee;
use ieee.std_logic_1164.all;

entity MemoryAddressControl is
port(
        putget, incPut, incGet, Reset, clk: in std_logic;
        full, empty: out std_logic;
        Address: out std_logic_vector(2 downto 0)
);
end MemoryAddressControl;


architecture behavioral of MemoryAddressControl is

component MAC_Cont port(
        CE, Reset, clk: in std_logic;
        Q: out std_logic_vector(2 downto 0)
);
end component;

component MAC_Mux port(
        incPutOut, incGetOut: in std_logic_vector(2 downto 0);
        putGet: in std_logic;
        Address: out std_logic_vector(2 downto 0)
);
end component;

component MAC_Compare port(
        incPutOut, incGetOut: in std_logic_vector(2 downto 0);
        Q: out std_logic
);
end component;

component MAC_FFD port(
        clk : in std_logic;
        Reset : in STD_LOGIC;
        Set : in std_logic;
        D : IN STD_LOGIC;
        EN : IN STD_LOGIC;
        Q : out std_logic
);
```

```
end component;


signal term0, term1: std_logic_vector(2 downto 0);
signal term2, term3: std_logic;

begin

MAC_Cont_U0: MAC_Cont port map(
        CE => incPut,
        Reset => Reset,
        clk => clk,
        Q => term0
);

MAC_Cont_U1: MAC_Cont port map(
        CE => incGet,
        Reset => Reset,
        clk => clk,
        Q => term1
);

MAC_Mux_U0: MAC_Mux port map(
        incPutOut => term0,
        incGetOut=> term1,
        putGet => putget,
        Address => Address
);

MAC_Compare_U0: MAC_Compare port map(
        incPutOut => term0,
        incGetOut => term1,
        Q => term2
);

MAC_FFD_U0: MAC_FFD port map(
        CLK => CLK,
        RESET => Reset,
        SET => '0',
        D => putget,
        EN => '1',
        Q => term3
);

full <= term2 and term3;
empty <= term2 and not term3;

end behavioral;
```

H. Descrição VHDL do bloco RAM

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-- Project     : DE10-Lite
-- Affiliations: DEETC, ISEL - IPL
-- Funding     : -
--------------------------------------------------------------------------------
-- File        : RAM.vhd
-- Author(s)   : Pedro Miguens Matutino
-- Date        : 2023/01/03
```

```
----------------------------------------------------------------------------
-- Copyright (c) 2023 Pedro Miguens Matutino
----------------------------------------------------------------------------
-- Description :
-- .
----------------------------------------------------------------------------
----------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity 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 RAM;

architecture behavioral of RAM is

type RAM_TYPE is array (0 to 2**ADDRESS_WIDTH - 1) of std_logic_vector(DATA_WIDTH - 1 downto 0);
signal ram : RAM_TYPE;

begin

process(address, wr)
begin
dout <= ram(conv_integer(unsigned(address)));

if (wr='1') then
ram(conv_integer(unsigned(address))) <= din;
end if;
end process;

end behavioral;
```

I. 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);
Load, Reset, ACK, clk: in std_logic;
Q : out std_logic_vector(3 downto 0);
OBfree, Dval : out std_logic
);
end OutputBuffer;

architecture behavioral of OutputBuffer is
```

```
component OutputRegister is
port(
D : in std_logic_vector(3 downto 0);
clk : in std_logic;
E : in std_logic;
Reset : in std_logic;
Q : out std_logic_vector(3 downto 0)
);
end component;

component BufferControl is
port(
Reset, clk, Load, ACK: in std_logic;
OBfree, Wreg, Dval: out std_logic
);
end component;

signal term0: std_logic;

begin

BufferControl_U0: BufferControl port map(
Reset => Reset,
clk => clk,
Load => Load,
ACK => ACK,
OBfree => OBfree,
Wreg => term0,
Dval => Dval
);

OutputRegister_U0: OutputRegister port map(
D => D,
clk => term0,
E => '1',
Reset => Reset,
Q => Q
);

end behavioral;
```

J. Descrição VHDL do bloco Buffer Control

```
library ieee;
use ieee.std_logic_1164.all;

entity BufferControl is
port(
        Reset, clk, Load, ACK: in std_logic;
        OBfree, Wreg, Dval: out std_logic
);
end BufferControl;

architecture behavioral of BufferControl is

type STATE_TYPE is (STATE_WAIT_LOAD, STATE_REGISTER, STATE_WAIT_ACK, STATE_WAIT_NOT_LOAD);

signal CurrentState, NextState: STATE_TYPE;
```

```
begin

-- Flip-Flop's
CurrentState <= STATE_WAIT_LOAD when Reset = '1' else NextState when rising_edge(clk);

-- Generate Next State
GenerateNextState:
process (CurrentState, Load, ACK)
	begin
		case CurrentState is
			when STATE_WAIT_LOAD => if(Load = '1') then NextState <= STATE_REGISTER;
				else NextState <= STATE_WAIT_LOAD;
					end if;

			when STATE_REGISTER => NextState <= STATE_WAIT_ACK;


			when STATE_WAIT_ACK => if(ACK = '1') then NextState <= STATE_WAIT_LOAD;
				else NextState <= STATE_WAIT_ACK;
					end if;

			when STATE_WAIT_NOT_LOAD => if(ACK = '0') then NextState <= STATE_WAIT_LOAD;
				else NextState <= STATE_WAIT_NOT_LOAD;
					end if;

		end case;
	end process;

OBfree <= '1' when (CurrentState = STATE_WAIT_LOAD) else '0';
Dval <= '1' when (CurrentState = STATE_WAIT_ACK) else '0';
Wreg <= '1' when (CurrentState = STATE_REGISTER) else '0';

end behavioral;
```

K. Descrição VHDL do bloco Output Register

```
library ieee;
use ieee.std_logic_1164.all;

entity OutputRegister is
port(
D : in std_logic_vector(3 downto 0);
clk : in std_logic;
E : in std_logic;
Reset : in std_logic;
Q : out std_logic_vector(3 downto 0)
);
end OutputRegister;

architecture behavioral of OutputRegister is

component OutputRegister_FFD port(
clk : in std_logic;
Reset : in std_logic;
Set : in std_logic;
D : in std_logic;
EN : in std_logic;
Q : out std_logic
);
```

```
end component;

begin

FFD_U0: OutputRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
D => D(3),
EN => E,
Q => Q(3)
);

FFD_U1: OutputRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
D => D(2),
EN => E,
Q => Q(2)
);

FFD_U2: OutputRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
D => D(1),
EN => E,
Q => Q(1)
);

FFD_U3: OutputRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
D => D(0),
EN => E,
Q => Q(0)
);

end behavioral;
```

L. Código *Kotlin - HAL*

```
object HAL {   // Virtualiza o acesso ao sistema UsbPort
   var Lastoutput = 0

   // Inicia a classe
       fun init() {
           Lastoutput = 0
           UsbPort.write(Lastoutput)

       }

   // Retorna true se o bit tiver o valor lógico ‘1’
```

```
    fun isBit(mask: Int): Boolean = UsbPort.read().and(mask) != 0

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

    // Escreve nos bits representados por mask o valor de value
        fun writeBits(mask: Int, value: Int){

            Lastoutput = (Lastoutput and mask.inv()) or (value and mask)

            UsbPort.write(Lastoutput)
        }

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

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

M. Código *Kotlin - KBD*

```
import isel.leic.utils.Time

object KBD { // Ler teclas. Métodos retornam ‘0’..’9’,’#’,’*’ ou NONE.
    const val NONE = 0
    const val KVAL_MASK = 16
    const val K_DATA_MASK = 15
    const val KACK_MASK = 128

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

    //Retorna de imediato a tecla premida ou NONE se não há tecla premida.
    fun getKey(): Char {
        val Kval = HAL.isBit(KVAL_MASK)
        if (Kval) {
            val value = HAL.readBits(K_DATA_MASK)
            val button = when (value) {
                0 -> '1'
                1 -> '4'
                2 -> '7'
                3 -> '*'

                4 -> '2'
                5 -> '5'
                6 -> '8'
                7 -> '0'
```

```
                8 -> '3'
                9 -> '6'
                10 -> '9'
                11 -> '#'
                else -> NONE.toChar()
            }

            HAL.setBits(KACK_MASK) //Kack está a TRUE

            while (HAL.isBit(KVAL_MASK));

            HAL.clrBits(KACK_MASK) // Kack está a false

            return button

        }
        return NONE.toChar()
    }

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