

O módulo *Keyboard Reader* é constituído por três blocos principais:

- o decodificador de teclado (*Key Decode*);
- o bloco de armazenamento (designado por *Ring Buffer*);
- o bloco de entrega ao consumidor (designado por *Output Buffer*). Neste caso o módulo *Control*, implementado em *software*, é a entidade consumidora.

O relatório presente refere-se aos módulos *Ring Buffer* e o módulo *Output Buffer* e a componente de Key control, parate do diagrama de blocos que se encontra plasmado na informação que se segue.

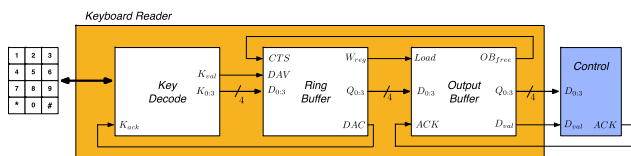


Figura 1

Diagrama de blocos do módulo *Keyboard Reader*

1 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 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. Caso a RAM esteja cheia, os dados novos são perdidos.

A implementação do *Ring Buffer* é baseada numa memória RAM (*Random Access Memory*). O endereço de escrita/leitura, selecionado por *putget*, 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 é apresentado o diagrama de blocos para a estrutura do bloco *Ring Buffer*, e na Figura 3 o detalhe do bloco *Memory Address Control*, cujo código VHDL é apresentado no anexo A..

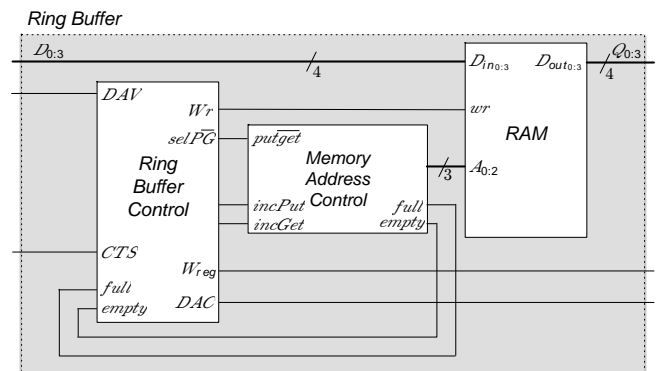


Figura 2

Diagrama de blocos do bloco *Ring Buffer*

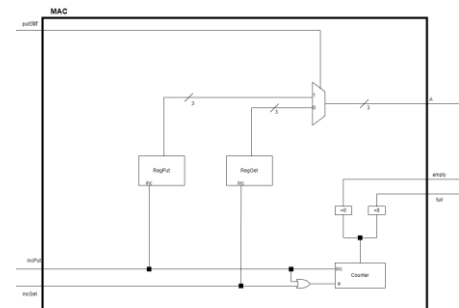


Figura 3

Diagrama de blocos da *Memory Address Control*

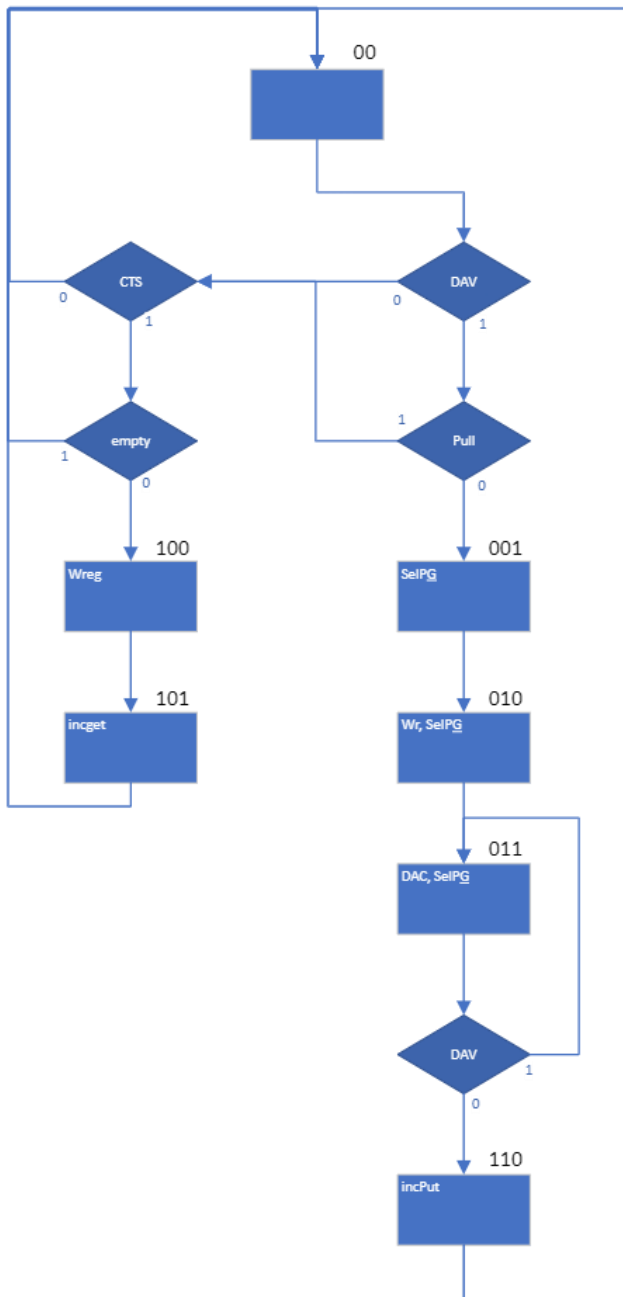


Figura 4

Máquina de estados do bloco *Ring Buffer*

2 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 , é apresentado o diagrama de blocos do *Output Buffer*.

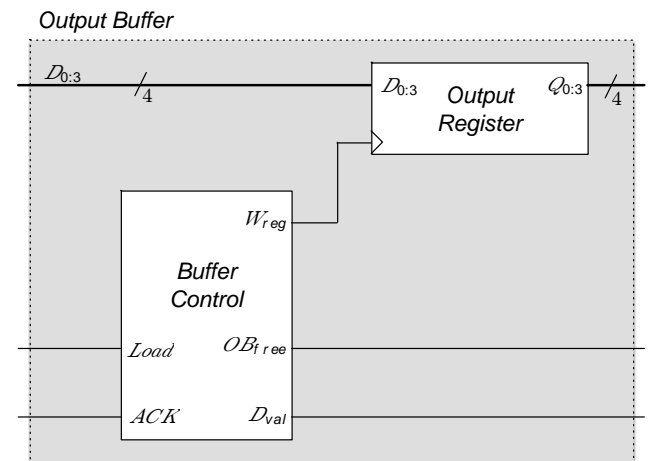


Figura 5

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 diagrama de blocos representado na Figura 5.

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

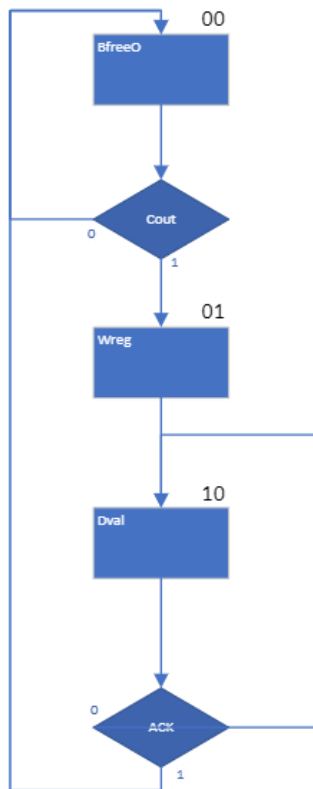


Figura 6

Máquina de estados do bloco *Buffer Control*

Com base nas descrições do bloco *Key Decode* e do bloco *Key Buffer Control* implementou-se o módulo *Keyboard Reader* o código VHDL.

3 Interface com o *Control*

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

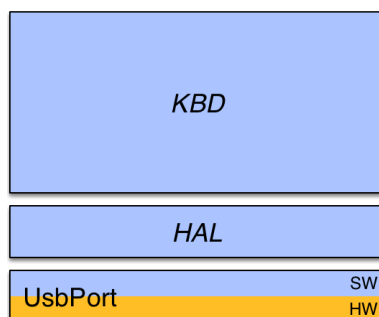


Figura 7

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

3.1 Classe *HAL*

A classe *Hal* é o object *HALVirtualiza* o acesso ao sistema *UsbPort*, cujo código kotlin está incluído no anexo B, e é constituída para além da função *main*, por:

- i) *fun init()*: Inicia a classe;
- ii) *fun isBit(mask: Int)*: Boolean: Retorna *true* se o bit tiver o valor lógico '1';
- iii) *fun readBits(mask: Int)*: Int: Retorna os valores dos bits representados por *mask* presentes no *UsbPort*;
- iv) *fun writeBits(mask: Int, value: Int)*: Escreve nos bits representados por *mask* os valores dos bits correspondentes em *value*;
- v) *fun setBits(mask: Int)*: Coloca os bits representados por *mask* no valor lógico '1'
- vi) *fun clrBits(mask: Int)*: Coloca os bits representados por *mask* no valor lógico '0'.

3.2 Classe *KBD*

A classe *keyboard* serve para apanhar as teclas lidas, e utiliza a classe *HAL*, o código kotlin é apresentado no anexo B, e é constituída para além da função *main* pelas seguintes funções:

- i) *fun init()*: para iniciar;
- ii) *fun getKey()*: Char: Retorna de imediato a tecla premida ou *NONE* se não há tecla premida;
- iii) *fun waitKey(timeout: Long)*: Char: Retorna a tecla premida, caso ocorra antes do 'timeout' (representado em milissegundos), ou *NONE* caso contrário;

3.3 Classe TUI

Esta classe apresenta-se como uma interface para apresentação/ gestão da informação, nos vários movimentos e localizações com que vai ser mostrada no LCD, neste momento apenas tem implementadas as seguintes funções:

- i) fun keyToLCD;
- ii) fun writeFromLeft(s: String);
- iii) fun writeFromRight(s: String);

Nota:

Atendendo a que será necessário recorrentemente disponibilizar funções que garantam o aparecimento em diversos movimentos e posicionamentos no LCD é intenção de criar uma função cursor que recebe (pos, rl, qtd), em que pos será a posição inicial do cursos, rl, em que 0 terá movimento para a esquerda e 1 movimento para a direita, e qtd qual o número de movimentos a executar, que poderá ser acedida por outras como inicio dum local qualquer, direita, esquerda centro, ou mesmo de forma aleatória, passando os respetivos parâmetros a esta função que se encarregar de efetuar os respetivas instruções.

4 conclusões

A implementação deste modulo, com base no descrito no relatório tem possibilidade para melhoria. No entanto, devido às limitações presentes podem ocorrer certos problemas, particularmente a possível perda de informação devido ao espaço limitado de RAM.

A.Descrição VHDL do Keyboard Reader

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

entity keyboardReader is
port(
    rst: in std_logic;
    clk: in std_logic;
    kack: in std_logic;
    I: in std_logic_vector(3 downto 0);
    O: out std_logic_vector(3 downto 0);
    dval: out std_logic;
    D: out std_logic_vector(3 downto 0)
);

end keyboardReader;

architecture structure of keyboardReader is
component keyDecode is
port(
    rst: in std_logic;
    clk: in std_logic;
    kack: in std_logic;
    I: in std_logic_vector(3 downto 0);
    O: out std_logic_vector(3 downto 0);
    kval: out std_logic;
    K: out std_logic_vector(3 downto 0)
);

end component;

begin

kdecode: keyDecode port map(
    rst => rst,
    clk => clk,
    kack => kack,
    I => I,
    O => O,
    kval => dval,
    K => D
);

end structure;
```

B. Descrição VHDL de Key Control

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

entity keyControl is
port(
    clk: in std_logic;
    rst: in std_logic;
    kpress: in std_logic;
    kack: in std_logic;
    kscan: out std_logic;
    kval: out std_logic
);

end keyControl;

architecture behavioral of keyControl is

    type STATE_TYPE is (STATE_00, STATE_01, STATE_10);
    signal CurrentState, NextState : STATE_TYPE;

begin

    --Flip-Flop
    CurrentState <= STATE_00 when rst = '1' else NextState when rising_edge(clk);

    GenerateNextState:
    process(CurrentState, kack, kpress)
    begin

        case CurrentState is
            when STATE_00 => if (kpress = '0') then
                                NextState <= STATE_00;
                            else
                                NextState <= STATE_01;
                            end if;

            when STATE_01 => if( kack = '0') then
                                NextState <= STATE_01;
                            else
                                NextState <= STATE_10;
                            end if;

            when STATE_10 => if(kpress = '1') then
                                NextState <= STATE_10;
                            else
                                NextState <= STATE_00;
                            end if;

        end case;

    end process;

end keyControl;
```

end process;

kscan <= '1' when (CurrentState = STATE_00) else '0';

kval <= '1' when (CurrentState = STATE_01) else '0';

end behavioral;

C. Descrição VHDL de Key Scan

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

entity keyScan is
port(
    Kscan: in std_logic;
    I: in std_logic_vector(3 downto 0);
    clk: in std_logic;
    Rst: in std_logic;
    K: out std_logic_vector(3 downto 0);
    Kpress: out std_logic;
    O: out std_logic_vector(3 downto 0)
);
end keyScan;

architecture structure of KeyScan is
component Multiplexer is
port(
    I: in std_logic_vector(3 downto 0);
    OP0, OP1: in std_logic;
    Y: out std_logic
);
end component;

component Decoder is
port(
    S0, S1: in std_logic;
    O: out std_logic_vector(3 downto 0)
);
end component;

component clkDIV is
port (
    clk_in: in std_logic;
    clk_out: out std_logic);
end component;

component Cont is
port
(
    R: in std_logic;
    CE: in std_logic;
    Clk: in std_logic;
    Q: out std_logic_vector(3 downto 0)
);
end component;

signal QtoDec: std_logic_vector(3 downto 0);
```



```
signal CDiv: std_logic;
```

```
begin
```

```
contador: Cont port map(
```

```
    CE => Kscan,
```

```
    R => Rst,
```

```
    Clk => CDiv,
```

```
    Q(3) => QtoDec(3),
```

```
    Q(2) => QtoDec(2),
```

```
    Q(1) => QtoDec(1),
```

```
    Q(0) => QtoDec(0)
```

```
);
```

```
dec: Decoder port map(
```

```
    S0 => QtoDec(3),
```

```
    S1 => QtoDec(2),
```

```
    O => O
```

```
);
```

```
mux: Multiplexer port map(
```

```
    I => I,
```

```
    OP0 => QtoDec(1),
```

```
    OP1 => QtoDec(0),
```

```
    Y => Kpress
```

```
);
```

```
clkD: clkDIV port map(
```

```
    clk_in => clk,
```

```
    clk_out => CDiv
```

```
);
```

```
K <= QtoDec;
```

```
end structure;
```

D. Descrição VHDL de Key Decode

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

entity keyDecode is
port(
    rst: in std_logic;
    clk: in std_logic;
    kack: in std_logic;
    I: in std_logic_vector(3 downto 0);
    O: out std_logic_vector(3 downto 0);
    kval: out std_logic;
    K: out std_logic_vector(3 downto 0)
);

end keyDecode;
architecture structure of keyDecode is
component keyScan is
port(
    Kscan: in std_logic;
    I: in std_logic_vector(3 downto 0);
    clk: in std_logic;
    Rst: in std_logic;
    K: out std_logic_vector(3 downto 0);
    Kpress: out std_logic;
    O: out std_logic_vector(3 downto 0)
);
end component;

component keyControl is
port(
    clk: in std_logic;
    rst: in std_logic;
    kpress: in std_logic;
    kack: in std_logic;
    kscan: out std_logic;
    kval: out std_logic
);
end component;

signal kp: std_logic;
signal ks: std_logic;
```

begin

kscan: keyScan port map(

Kscan => ks,

I => I,

clk => clk,

Rst => rst,

K => K,

Kpress => kp,

O => O

);

kctrl: keyControl port map(

clk => clk,

rst => rst,

kpress => kp,

kack => kack,

kscan => ks,

kval => kval

);

end structure;

E. Descrição VHDL de Memory Address Control (MAC)

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

entity MAC is
port(
    clk: in std_logic;
    rst: in std_logic;
    putGet: in std_logic;
    incPut: in std_logic;
    incGet: in std_logic;
    full: out std_logic;
    empty: out std_logic;
    Address: out std_logic_vector(2 downto 0)
);
end MAC;

architecture structure of MAC is
component UpDownCounter is
port(
    CLK: in std_logic;
    RST: in std_logic;
    Enable: in std_logic;
    IncDec: in std_logic;
    R: out std_logic_vector(3 downto 0)
);
end component;

component counter3 is
port(
    CLK: in std_logic;
    RST: in std_logic;
    CE: in std_logic;
    Q: out std_logic_vector(2 downto 0)
);
end component;

component Mux2bit is
port(
    X: in std_logic_vector(2 downto 0);
    Y: in std_logic_vector(2 downto 0);
    S: in std_logic;
    R: out std_logic_vector(2 downto 0)
);
end component;
```

```
signal s_putGet: std_logic;
signal s_UDCounter: std_logic_vector(3 downto 0);
signal s_put: std_logic_vector(2 downto 0);
signal s_get: std_logic_vector(2 downto 0);
signal s_enable: std_logic;

begin

putCount: counter3 port map(
    CLK => clk,
    RST => rst,
    CE => incPut,
    Q => s_put
);

getCount: counter3 port map(
    CLK => clk,
    RST => rst,
    CE => incGet,
    Q => s_get
);

mux: Mux2bit port map(
    X => s_put,
    Y => s_get,
    S => putGet,
    R => Address
);

UpDownCount: UpDownCounter port map(
    CLK => clk,
    RST => rst,
    Enable => s_enable,
    IncDec => incPut,
    R => s_UDCounter
);

s_enable <= (incPut or incGet);
full <= s_UDCounter(3);
empty <= not (s_UDCounter(0) or s_UDCounter(1) or s_UDCounter(2) or s_UDCounter(3));

end structure;
```

F. Descrição VHDL dos blocos *Ring buffer*

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

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

architecture structure of RingBuffer is
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;
    incPut: out std_logic;
    incGet: out std_logic;
    Wreg: out std_logic;
    DAC: out std_logic
);
end component;

component MAC is
port(
    clk: in std_logic;
    rst: in std_logic;
    putGet: in std_logic;
    incPut: in std_logic;
    incGet: in std_logic;
    full: out std_logic;
    empty: out std_logic;
    Address: out std_logic_vector(2 downto 0)
);
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;

```
signal s_wr: std_logic;  
signal s_selPG: std_logic;  
signal s_incPut: std_logic;  
signal s_incGet: std_logic;  
signal s_address: std_logic_vector(2 downto 0);  
signal s_full: std_logic;  
signal s_empty: std_logic;
```

begin

```
RingBufferCtrl: RingBufferControl port map (  
    clk => clk,  
    rst => rst,  
    DAV => DAV,  
    CTS => CTS,  
    full => s_full,  
    empty => s_empty,  
    wr => s_wr,  
    selPG => s_selPG,  
    incPut => s_incPut,  
    incGet => s_incGet,  
    Wreg => Wreg,  
    DAC => DAC  
);
```

```
MemoryAddressCtrl: MAC port map(  
    clk => clk,  
    rst => rst,  
    putGet => s_selPG,  
    incPut => s_incPut,  
    incGet => s_incGet,  
    full => s_full,  
    empty => s_empty,  
    Address => s_address  
);
```

```
R: RAM port map(  
    address => s_address,  
    wr => s_wr,  
    din => D,  
    dout => Q  
);
```

```
end structure;
```

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

```
entity 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;  
    incPut: out std_logic;  
    incGet: out std_logic;  
    Wreg: out std_logic;  
    DAC: 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_110);  
signal CurrentState, NextState : STATE_TYPE;
```

```
begin
```

```
--Flip-Flop
```

```
CurrentState <= STATE_000 when rst = '1' else NextState when rising_edge(clk);
```

```
GenerateNextState:
```

```
process(CurrentState, CTS, DAV, full, empty)  
begin
```



```

case CurrentState is
  when STATE_000 => if (DAV = '0' and CTS = '0') then
    NextState <= STATE_000;
  elsif (DAV = '0' and CTS = '1' and empty = '1') then
    NextState <= STATE_000;
  elsif (DAV = '0' and CTS = '1' and empty = '0') then
    NextState <= STATE_100;
  elsif (DAV = '1' and full = '0') then
    NextState <= STATE_001;
  elsif (DAV = '1' and full = '1' and CTS = '1' and
empty = '1') then
    NextState <= STATE_000;
  elsif (DAV = '1' and full = '1' and CTS = '1' and
empty = '0') then
    NextState <= STATE_100;
  end if;
  when STATE_100 => NextState <= STATE_101;
  when STATE_101 => NextState <= STATE_000;
  when STATE_001 => NextState <= STATE_010;
  when STATE_010 => NextState <= STATE_011;
  when STATE_011 => if (DAV = '1') then
    NextState <= STATE_011;
  else
    NextState <= STATE_110;
  end if;
  when STATE_110 => NextState <= STATE_000;

end case;

end process;

Wreg <= '1' when (CurrentState = STATE_100) else '0';
incGet <= '1' when (CurrentState = STATE_101) else '0';

SelPG <= '1' when (CurrentState = STATE_001 or CurrentState = STATE_010 or CurrentState =
STATE_011 ) else '0';
incPut <= '1' when (CurrentState = STATE_110) else '0';

wr <= '1' when (CurrentState = STATE_010) else '0';

DAC <= '1' when (CurrentState = STATE_011) else '0';

end behavioral;

```

G. Descrição VHDL do bloco *Output Buffer*

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

entity OutputBuffer is
port(
    clk: in std_logic;
    rst: std_logic;
    D: in std_logic_vector(3 downto 0);
    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 structure of OutputBuffer is

    component Registo is
    port(
        A: in std_logic_vector(3 downto 0);
        Clk: in std_logic;
        Reset: in std_logic;
        E: in std_logic;
        S: out std_logic_vector(3 downto 0)
    );

    end component;

    component OutputBufferControl is
    port(
        clk: in std_logic;
        rst: in std_logic;
        Load: in std_logic;
        ack: in std_logic;
        Wreg: out std_logic;
        OBfree: out std_logic;
        Dval: out std_logic
    );

    end component;

    signal s_Wreg: std_logic;
```

begin

OutputBufferCtrl: OutputBufferControl port map(

clk => clk,

rst => rst,

Load => Load,

ack => ack,

Wreg => s_Wreg,

OBfree => OBfree,

Dval => Dval

);

Reg: Registo port map(

A => D,

clk => s_Wreg,

Reset => rst,

E => '1',

S => Q

);

end structure;

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

entity OutputBufferControl is
port(
    clk: in std_logic;
    rst: in std_logic;
    Load: in std_logic;
    ack: in std_logic;
    Wreg: out std_logic;
    OBfree: out std_logic;
    Dval: out std_logic
);
end OutputBufferControl;

architecture behavioral of OutputBufferControl is

type STATE_TYPE is (STATE_00, STATE_01, STATE_10);
signal CurrentState, NextState : STATE_TYPE;

begin

--Flip-Flop
CurrentState <= STATE_00 when rst = '1' else NextState when rising_edge(clk);

GenerateNextState:
process(CurrentState, Load, ack)
begin

case CurrentState is
    when STATE_00 => if(Load = '0') then
                                NextState <= STATE_00;
                        else
                                NextState <= STATE_01;
                        end if;

    when STATE_01 => NextState <= STATE_10;

    when STATE_10 => if (ack = '0') then
                                NextState <= STATE_10;
                        else
                                NextState <= STATE_00;
                        end if;

    end case;

end process;

end process;
```

OBfree <= '1' when (CurrentState = STATE_00) else '0';

Wreg <= '1' when (CurrentState = STATE_01) else '0';

Dval <= '1' when (CurrentState = STATE_10) else '0';

end behavioral;

H. Código Kotlin - HAL

```
import isel.leic.UsbPort

object HAL { // Virtualiza o acesso ao sistema UsbPort

    var prev_state: Int = 0
    // Inicia a classe
    fun init() {
        prev_state = 0
        UsbPort.write(prev_state)
    }
    // Retorna true se o bit tiver o valor lógico '1'
    fun isBit(mask: Int): Boolean {
        var value = UsbPort.read()
        value = value and mask
        if ( value == mask) {
            return true
        }
        return false
    }

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

    // Escreve nos bits representados por mask os valores dos bits correspondentes em value
    fun writeBits(mask: Int, value: Int) {
        prev_state = (prev_state and mask.inv()) or (mask and value)
        UsbPort.write(prev_state)
    }

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

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

fun main() {
    HAL.init()
}
```

```
while(true){  
    HAL.setBits(0xCC)  
    Thread.sleep(2000)  
    HAL.writeBits(0x66, 0x33)  
    Thread.sleep(2000)  
    HAL.setBits(0x3C)  
    Thread.sleep(2000)  
    HAL.clrBits(0x99)  
    Thread.sleep(2000)  
  
}  
  
}
```

I. Código Kotlin – KBD

```
import isel.leic.utils.Time
import java.time.LocalDateTime
import java.time.LocalTime

object KBD { // Ler teclas. Métodos retornam '0'..'9','#','*' ou NONE.
    const val NONE = 0;
    private const val keyboardMask = 0x0F
    private const val kvalMask = 0x10
    private const val kackMask = 0x80

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

    // Retorna de imediato a tecla premida ou NONE se não há tecla premida.
    fun getKey(): Char {

        if(HAL.isBit(kvalMask)){
            val key = HAL.readBits(keyboardMask)

            val c = when(key){
                0x00 -> '1'
                0x01 -> '4'
                0x02 -> '7'
                0x03 -> '*'
                0x04 -> '2'
                0x05 -> '5'
                0x06 -> '8'
                0x07 -> '0'
                0x08 -> '3'
                0x09 -> '6'
                0x0A -> '9'
                0x0B -> '#'
                else -> NONE.toChar()
            }
            if (c != NONE.toChar()){

                if(HAL.isBit(kvalMask)){
                    HAL.setBits(kackMask)
                    while(HAL.isBit(kvalMask));
                    HAL.clrBits(kackMask)
                }
                return c
            }
        }

        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 prevTime = Time.getTimeInMillis()

    while (true){

        val key = getKey()

        if(key != NONE.toChar()){
            return key
        }

        val currTime = Time.getTimeInMillis()
        if((currTime - prevTime) > timeout){
            return NONE.toChar()
        }
    }
}

fun main(){
    HAL.init()
    KBD.init()
    /*
    while (true){
        Thread.sleep(1000)
        println(KBD.getKey())
    }

    */
    testKBDHardware()
}

fun testKBDHardware(){

    while (true){
        val key = KBD.waitKey(3000)
        println(key)
    }
}
```