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.

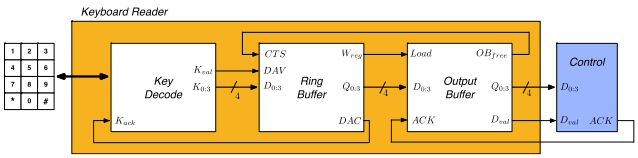
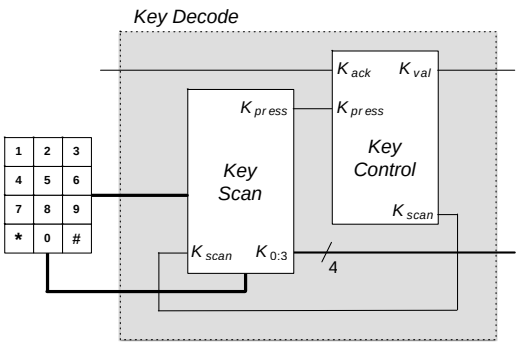


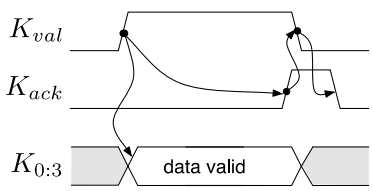

Figura 1 – Diagrama de blocos do módulo *Keyboard Reader*

# 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 *Ring 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. Escolhemos a versão I pois foi a que o professor recomendou e ao longo do desenvolvimento do trabalho consideramos ser uma versão que nos agradou.

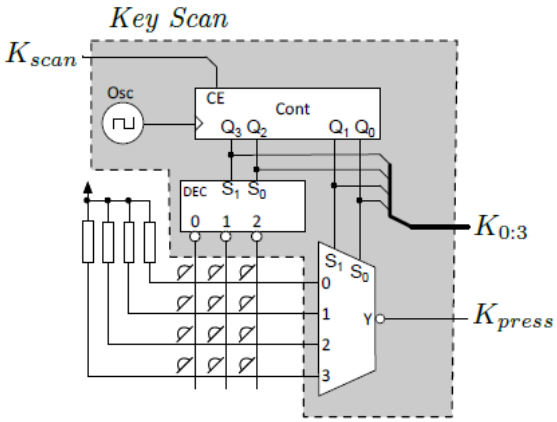



Figura 3 - Diagrama de blocos do bloco *Key Scan*

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

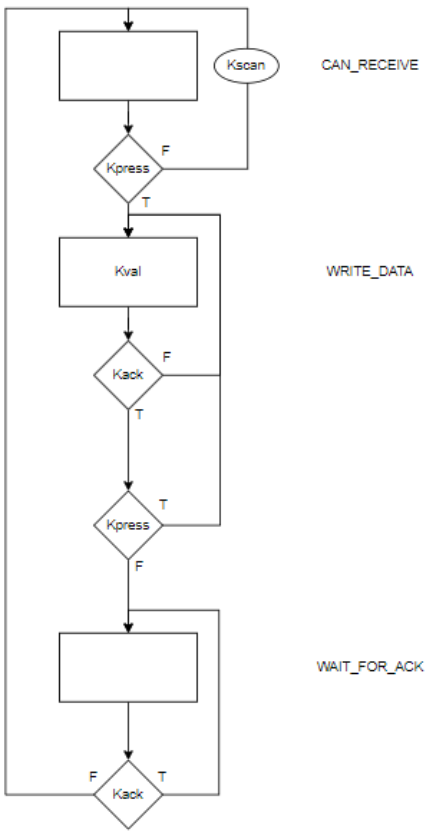


Figura 4 – Máquina de estados do bloco *Key Control*

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

# 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.

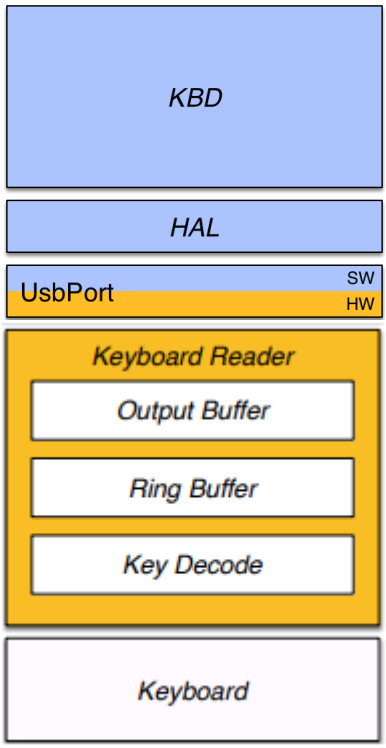



Figura 5 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

*HAL* e *KBD* desenvolvidos são descritos nas secções 2.1. e 2.2, e o código fonte desenvolvido nos Anexos O e P, respetivamente.

## 2.1 *HAL*

É a interação entre usbPort e o software.

Desenvolvemos 6 funções:

### *2.1.1 fun init()*

Inicia o usbPort a 0 e atualiza o lastState para 0.

### *2.1.2 fun readBits()*

Lê o valor de output nos bits correspondentes a máscara enquanto restantes bits retornam a 0.

### *2.1.3 fun isBits()*

Verifica o valor de um bit em específico.

### *2.1.4 fun setBits()*

Coloca valores correspondentes a máscara a 1 e deixa os restantes como estavam.

### *2.1.5 fun clrBits()*

Coloca valores correspondentes à máscara a 0 e deixa os restantes como estavam.

### *2.1.6 fun writeBits()*

Tem como função escrever valor recebido nos bits indicados pela máscara (restantes bits mantém-se).

A utilização da linha de código:

**lastState = lastState.or(mask)**

nas funções *setBits*, *clrBits* e *writeBits* serve para manter valor de *lastState* atualizado.

## 2.2 *KBD*

O desenvolvimento o software de KBD é composto por 3 funções:

### *2.2.1 fun init()*

Inicia escrita e leitura do usbPort chamando o *HAL.init* e faz o *clearBits(HAL.clrBits)* à máscara do *Kack* (para este começar a 0).

### *2.2.2 fun getKey()*

Lê o código da máscara que está no valor de cada tecla e transforma-o no devido caracter caso seja válida, é dado *ack* da tecla, espera tecla ser validada

e devolve o char correspondente á tecla que leu. Isto no caso de *Kval* estar ativo. Se *Kval* estiver inativo retorna NONE.

### *2.2.3 fun waitKey()*

Recebe período (timeout) e tem um ciclo que tenta ler teclas. Se ler uma tecla válida retorna essa tecla. Se não, continua à procura até o tempo de o ciclo exceder e retorna NONE.

## *2.3 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 $D_{0:3}$ 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*get, é definido pelo bloco *Memory Address Control* (MAC) composto por dois registos, que contêm o endereço de escrita e leitura, designados por *putIndex* e *getIndex* respetivamente. O MAC suporta assim ações de *incPut* e *incGet*, gerando informação se a estrutura de dados está cheia (*full*) ou se está vazia (e*mpty*). 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*.

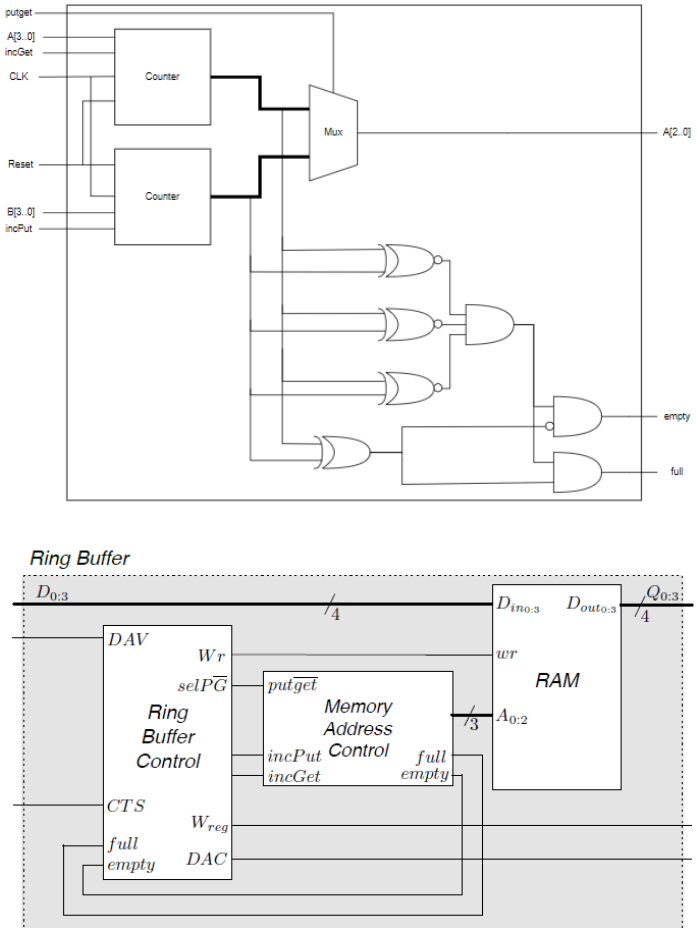



Figura 6 - Diagrama de blocos do bloco *Ring Buffer*

Figura 7 – Diagrama de blocos do bloco *Memory Address Control*

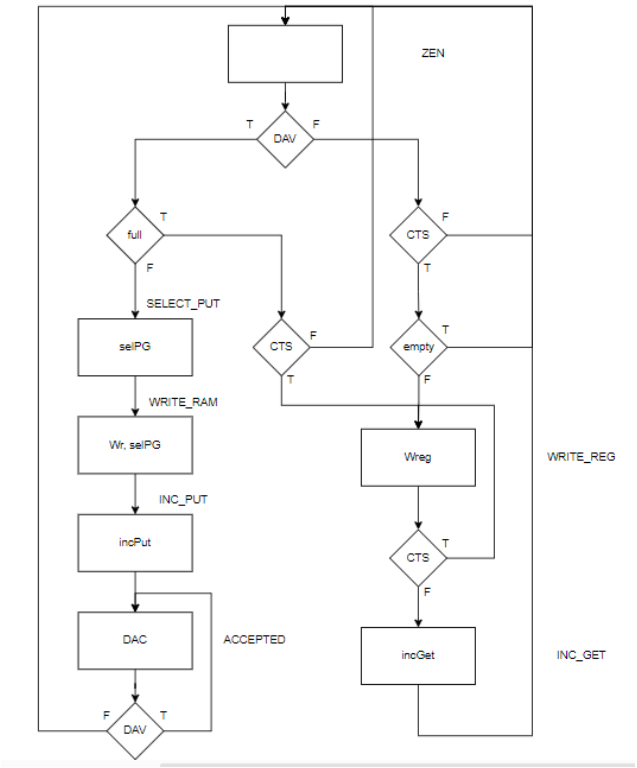



Figura 8 – Máquina de estados do *Ring Buffer Control*

## 2.4 Output Buffer

O bloco *Output Buffer* do *Keyboard Reader* é responsável pela interação com o sistema consumidor, neste caso o módulo
*Control*.
O *Output Buffer* indica que está disponível para armazenar dados através do sinal *OBfree*. Assim, nesta situação o sistema produtor pode ativar o sinal *Load* para registar os dados.
O *Control* quando pretende ler dados do *Output Buffer*, aguarda que o sinal *Dval* fique ativo, recolhe os dados e pulsa o sinal *ACK* indicando que estes já foram consumidos.
O *Output Buffer*, logo que o sinal *ACK* pulse, deve invalidar os dados baixando o sinal *Dval* e sinalizar que está novamente disponível para entregar dados ao sistema consumidor, ativando o sinal *OBfree*. Na Figura 9, é apresentado o diagrama de blocos do *Output Buffer*.

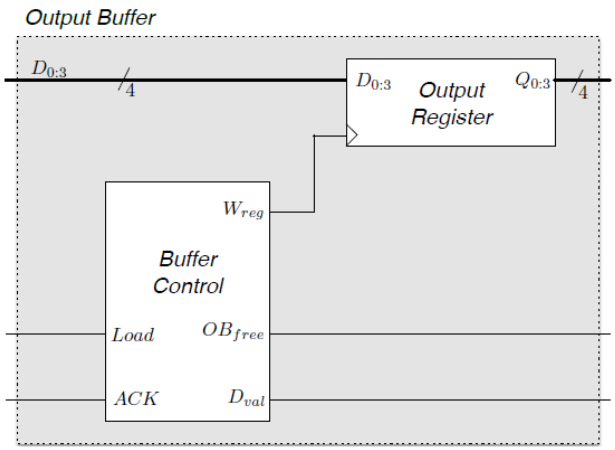



Figura 9 - Diagrama de blocos do bloco *Output Buffer*

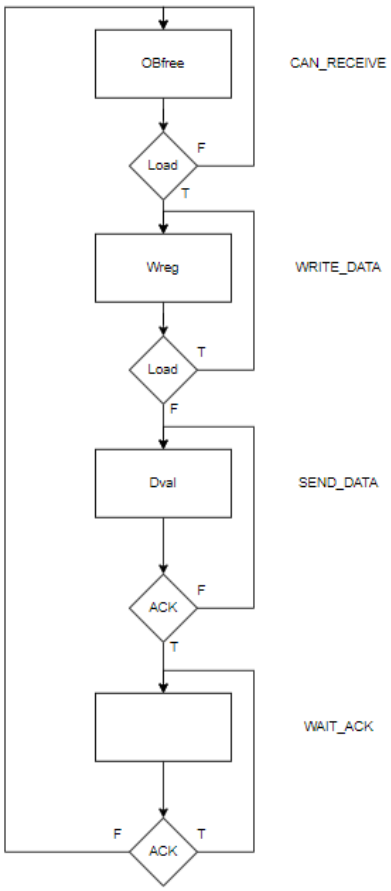



Figura 10 – Máquina de estados do *Buffer Control*

# 3 Conclusões

Por o clock da placa ser 50MHz, estava demasiado rápido para o propósito desejado, pelo que utilizámos o CLK_div com div = 1000, para que fosse mais lento e atingisse o objetivo.

# A. Descrição VHDL do bloco Key Scan

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

entity KeyScan is
port(
KScan, CLK, Reset: in std_logic;
I : in std_logic_vector(3 downto 0);
Kpress : out std_logic;
O : out std_logic_vector(2 downto 0);
K : out std_logic_vector(3 downto 0));
end KeyScan;

architecture arc_Keyscan of KeyScan is

component Counter
port(
PL, CE, CLK, Reset: in std_logic;
Data_in: in std_logic_vector(3 downto 0);
TC: out std_logic;
Q: out std_logic_vector(3 downto 0));
end component;

component Decoder
port(
S: in std_logic_vector(1 downto 0);
O: out std_logic_vector(2 downto 0));
end component;

component MUX4x1
port(
S: in std_logic_vector(1 downto 0);
I: in std_logic_vector(3 downto 0);
Y: out std_logic);
end component;

signal kp : std_logic;
signal col : std_logic_vector(2 downto 0);
signal qcount : std_logic_vector(3 downto 0);

begin

count: Counter port map(
Reset => Reset,
PL => '0',
Data_in => "0000",
CE => KScan,
CLK => CLK,
Q => qcount);
```

```
dec: Decoder port map(
S(1) => qcount(3),
S(0) => qcount(2),
O => col);

mux: MUX4x1 port map(
S(1) => qcount(1),
S(0) => qcount(0),
I => I,
Y => kp);

O <= col;
Kpress <= not kp;
K <= qcount;

end arc_Keyscan;
```

## B. Descrição VHDL do bloco *Key Control*

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

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

architecture arcKC of KeyControl is

type STATE_TYPE is (WAIT_FOR_PRESS, VALIDATE, WAIT_FOR_AFK);

signal CurrentState, NextState: STATE_TYPE;

begin

CurrentState <= WAIT_FOR_PRESS when Reset = '1' else NextState when rising_edge(CLK);

GenerateNextState:
process (CurrentState, Kpress, Kack)
        begin
                case CurrentState is
                        when WAIT_FOR_PRESS =>
                                if (Kpress = '1') then NextState <= VALIDATE;
                                else NextState <= WAIT_FOR_PRESS;
                                end if;
                        when VALIDATE =>
                                if (Kack = '1' and Kpress = '0') then NextState <= WAIT_FOR_AFK;
                                else NextState <= VALIDATE;
                                end if;
                        when WAIT_FOR_AFK =>
                                if (Kack = '0') then NextState <= WAIT_FOR_PRESS;
                                else NextState <= WAIT_FOR_AFK;
                                end if;
                end case;
end process;

Kval <= '1' when ( CurrentState = VALIDATE) else '0';
Kscan <= '1' when ( CurrentState = WAIT_FOR_PRESS and Kpress = '0') else '0';
```

```
end arcKC;
```

## C. Descrição VHDL do bloco *CLK Div*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;

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

architecture bhv of CLKDIV is

signal count: integer:=1;
signal tmp : std_logic := '0';

begin

process(clk_in)
begin

        if(clk_in'event and clk_in='1') then
                count <=count+1;
                if (count = div/2) then
                        tmp <= NOT tmp;
                        count <= 1;
                end if;
        end if;
end process;

clk_out <= tmp;

End bhv;
```

## D. Descrição VHDL do bloco *Key Decode*

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

entity KeyDecode is
port(
Kack , CLK, Reset: in std_logic;
I : in std_logic_vector(3 downto 0);
Kval : out std_logic;
O : out std_logic_vector(2 downto 0);
K : out std_logic_vector(3 downto 0));
end KeyDecode;
```

```
architecture arc_kd of KeyDecode is
component CLKDIV
generic(div: natural := 50000000);
port ( clk_in: in std_logic;
                    clk_out: out std_logic);
end component;

component KeyScan
port(
KScan, CLK, Reset: in std_logic;
I : in std_logic_vector(3 downto 0);
Kpress : out std_logic;
O : out std_logic_vector(2 downto 0);
K : out std_logic_vector(3 downto 0));
end component;

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

signal kp, ks, clockm : std_logic;

begin

clock: clkDIV generic map(1000)
port map(
clk_in => CLK,
clk_out => clockm);

scan : KeyScan port map(
KScan => ks,
CLK => clockm,
Reset => Reset,
I => I,
Kpress => kp,
O => O,
K => K);

control : KeyControl port map(
Kpress => kp,
Kack => Kack,
CLK => clockm,
Reset => Reset,
Kval => Kval,
KScan => ks);

end arc_kd;
```

# E. Descrição VHDL do sub-bloco full add

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

entity full_add is
Port(A, B, Cin: in std_logic;
Cout, S: out std_logic);
end full_add;

architecture arc_fa of full_add is
begin
S <= A xor B xor Cin;
Cout <= (A and B) or (A and Cin) or (B and Cin);
end arc_fa;
```

# F. Descrição VHDL do sub-bloco adder

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

entity adder is
Port(A, B :in std_logic_vector(3 downto 0);
Ci: in std_logic;
Co: out std_logic;
S: out std_logic_vector(3 downto 0));
end adder;


architecture arc_adder of adder is
component full_add
Port(A, B, Cin: in std_logic;
Cout, S: out std_logic);
end component;

signal c1, c2, c3: std_logic;

begin

fa1: full_add port map(
A => A(0),
B => B(0),
Cin => Ci,
S => S(0),
Cout => c1);

fa2: full_add port map(
A => A(1),
B => B(1),
Cin => C1,
S => S(1),
Cout => c2);

fa3: full_add port map(
A => A(2),
B => B(2),
Cin => C2,
S => S(2),
```

```
Cout => c3);

fa4: full_add port map(
A => A(3),
B => B(3),
Cin => C3,
S => S(3),
Cout => Co);

end arc_adder;
```

# G. Descrição VHDL do sub-bloco Counter

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

-- CC -> contador crescente!
entity Counter is
port(
PL, CE, CLK, Reset: in std_logic;
Data_in: in std_logic_vector(3 downto 0);
TC: out std_logic;
Q: out std_logic_vector(3 downto 0));
end Counter;

architecture arc_cc of Counter is
component adder
Port(A, B :in std_logic_vector(3 downto 0);
Ci: in std_logic;
Co: out std_logic;
S: out std_logic_vector(3 downto 0));
end component;

component MUX2x1
port(A, B: in std_logic_vector(3 downto 0);
S: in std_logic;
Y: out std_logic_vector(3 downto 0));
end component;

component Registry
port(
D:in std_logic_vector(3 downto 0);
CLK, E, Reset: in std_logic;
Q: out std_logic_vector(3 downto 0));
end component;

component Terminal_Count
port(
Q : in std_logic_vector(3 downto 0);
TC: out std_logic);
end component;

signal outadder, outmux, outreg: std_logic_vector(3 downto 0);

begin

ad: adder port map(
A => outreg,
```

```
B => "0000",
Ci => CE,
S => outadder);

mux: MUX2x1 port map(
A => Data_in,
B => outadder,
S => PL,
Y => outmux);

reg: Registry port map(
Reset => Reset,
E => '1',
D => outmux,
CLK => CLK,
Q => outreg);

Q <= outreg;

utc: Terminal_Count port map(
Q => outreg,
TC => TC);

end arc_cc;
```

# H. Descrição VHDL do sub-bloco MUX 2x1

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

entity MUX2x1 is
port(A, B: in std_logic_vector(3 downto 0);
S: in std_logic;
Y: out std_logic_vector(3 downto 0));
end MUX2x1;

architecture arc_mux of MUX2x1 is
begin
Y(0)<=(S and A(0)) or (not S and B(0));
Y(1)<=(S and A(1)) or (not S and B(1));
Y(2)<=(S and A(2)) or (not S and B(2));
Y(3)<=(S and A(3)) or (not S and B(3));
end arc_mux;
```

# I. Descrição VHDL do sub-bloco Memory Address Control

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

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

architecture arc of MemoryAddressControl is
```

```
component Counter
port(
PL, CE, CLK, Reset: in std_logic;
Data_in: in std_logic_vector(3 downto 0);
TC: out std_logic;
Q: out std_logic_vector(3 downto 0));
end component;

component Mux2x1
port(A, B: in std_logic_vector(3 downto 0);
S: in std_logic;
Y: out std_logic_vector(3 downto 0));
end component;

signal putIndex, getIndex, muxout : std_logic_vector(3 downto 0);
signal r, state_decider, equals : std_logic;

begin

--r <= Reset;
-- contador do indice no qual o put se encontra
incrementPut: Counter port map(
PL => '0',--r,
CE => incPut,
CLK => CLK,
Reset => Reset,
Data_in => "0000",
Q => putIndex);

-- contador do indice no qual o get se encontra
incrementGet: Counter port map(
PL => '0',--r,
CE => incGet,
CLK => CLK,
Reset => Reset,
Data_in => "0000",
Q => getIndex);

-- decide dependededendo do sinal putget qual indice indicar como output
idxDecider : Mux2x1 port map(
A => putIndex,
B => getIndex,
S => putget,
Y => muxout);

A <= muxout(2 downto 0);

state_decider <= putIndex(3) xor getIndex(3);
equals <= (putIndex(2) xnor getIndex(2)) and (putIndex(1) xnor getIndex(1)) and (putIndex(0) xnor getIndex(0));
--caso os indices put e get sejam iguais, o quarto bit indica se a ram está vazia ou cheia,
--pela lógica de que se forem iguais quer dizer que está vazio e se forem diferentes, quer dizer
--que está cheio
empty <= not state_decider and equals;
full <= state_decider and equals;
end arc;
```

# J. Descrição VHDL do sub-bloco Ring Buffer Control

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

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

architecture arc of RingBufferControl is

type STATE_TYPE is (ZEN, SELECT_PUT, WRITE_RAM, INC_PUT, ACCEPTED, WRITE_REG, INC_GET);

signal CurrentState, NextState: STATE_TYPE;

begin

CurrentState <= ZEN when Reset = '1' else NextState when rising_edge(CLK);

GenerateNextState:
process(CurrentState,DAV, full, empty, CTS)
	begin
		case CurrentState is
			when ZEN =>
				if (DAV = '1' and full = '0') then NextState <= SELECT_PUT;
				elsif(DAV = '1' and full = '1' and CTS = '1') then NextState <= WRITE_REG;
				elsif(DAV = '0' and CTS = '1' and empty = '0') then NExtState <= WRITE_REG;
				else NextState <= ZEN;
				end if;
			when SELECT_PUT => NextState <= WRITE_RAM;
			when WRITE_RAM => NextState <= INC_PUT;
			when INC_PUT => NextState <= ACCEPTED;
			when ACCEPTED =>
				if (DAV = '1') then NextState <= ACCEPTED;
				else NextState <= ZEN;
				end if;
			when WRITE_REG =>
				if (CTS = '1') then NextState <= WRITE_REG;
				else NextState <= INC_GET;
				end if;
			when INC_GET => NextState <= ZEN;
		end case;
end process;

selPG <= '1' when (CurrentState = SELECT_PUT or CurrentState = WRITE_RAM) else '0';
Wr <= '1' when (CurrentState = WRITE_RAM) else '0';
incPut <= '1' when (CurrentState = INC_PUT) else '0';
DAC <= '1' when (CurrentState = ACCEPTED) else '0';
Wreg <= '1' when (CurrentState = WRITE_REG) else '0';
incGet <= '1' when (CurrentState = INC_GET) else '0';
end arc;
```

# K. 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);
Wreg, DAC : out std_logic;
Q : out std_logic_vector(3 downto 0));
end RingBuffer;

architecture arc of RingBuffer is
component MemoryAddressControl
port(
putget, incPut, incGet, CLK, Reset : in std_logic;
full, empty : out std_logic;
A : 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;

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

signal putget, incp, incg, f, e, wram : std_logic;
signal idx : std_logic_vector(2 downto 0);

begin

control : RingBufferControl port map(
Reset => Reset,
CLK => CLK,
DAV => DAV,
CTS => CTS,
full => f,
empty => e,
Wr => wram,
selPG => putget,
Wreg => Wreg,
DAC => DAC,
incPut => incp,
incGet => incg);

mac : MemoryAddressControl port map(
putget => putget,
incPut => incp,
incGet => incg,
```

```
CLK => CLK,
Reset => Reset,
full => f,
empty => e,
A => idx);

memory : RAM port map(
address => idx(2 downto 0),
wr => wram,
din => D,
dout => Q);
end arc;
```

## *L.* Descrição VHDL do sub-*bloco Buffer Control*

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

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

architecture arc of BufferControl is

type STATE_TYPE is (CAN_RECEIVE, WRITE_DATA, SEND_DATA, WAIT_ACK);

signal CurrentState, NextState: STATE_TYPE;

begin

CurrentState <= CAN_RECEIVE when Reset = '1' else NextState when rising_edge(CLK);

GenerateNextState:
process(Load, ACK)
        begin
                case CurrentState is
                        when CAN_RECEIVE =>
                                if (Load = '1') then NextState <= WRITE_DATA;
                                else NextState <= CAN_RECEIVE;
                                end if;
                        when WRITE_DATA =>
                                if (Load = '0') then NextState <= SEND_DATA;
                                else NextState <= WRITE_DATA;
                                end if;
                        when SEND_DATA =>
                                if (ACK = '0') then NextState <= SEND_DATA;
                                else NextState <= WAIT_ACK;
                                end if;
                        when WAIT_ACK =>
                                if(ACK = '1') then NextState <= WAIT_ACK;
                                else NextState <= CAN_RECEIVE;
                                end if;
                end case;
end processS;
```

```
OBfree <= '1' when (CurrentState = CAN_RECEIVE) else '0';
Dval <= '1' when (CurrentState = SEND_DATA) else '0';
Wreg <= '1' when (CurrentState = WRITE_DATA) else '0';
end arc;
```

## M.Descrição VHDL do sub-*bloco Output Buffer*

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

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

architecture arc of OutputBuffer is
component BufferControl
port(
CLK, Reset, Load, ACK : in std_logic;
Wreg, OBfree, Dval : out std_logic);
end component;

component Registry
port(
E, CLK, Reset : in std_logic;
D : in std_logic_vector(3 downto 0);
Q : out std_logic_vector(3 downto 0));
end component;

signal w : std_logic;

begin

control : BufferControl port map(
CLK => CLK,
Reset => Reset,
Load => Load,
ACK => ACK,
Wreg => w,
OBfree => OBfree,
Dval => Dval);

outreg : Registry port map(
E => w,
CLK => CLK,
Reset => Reset,
D => D,
Q => Q);
end arc;
```

## N. Descrição VHDL do bloco *Keyboard Reader*

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

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

architecture arcKC of KeyControl is

type STATE_TYPE is (WAIT_FOR_PRESS, VALIDATE, WAIT_FOR_AFK);

signal CurrentState, NextState: STATE_TYPE;

begin

CurrentState <= WAIT_FOR_PRESS when Reset = '1' else NextState when rising_edge(CLK);

GenerateNextState:
process (CurrentState, Kpress, Kack)
	begin
		case CurrentState is
			when WAIT_FOR_PRESS =>
				if (Kpress = '1') then NextState <= VALIDATE;
				else NextState <= WAIT_FOR_PRESS;
				end if;
			when VALIDATE =>
				if (Kack = '1' and Kpress = '0') then NextState <= WAIT_FOR_AFK;
				else NextState <= VALIDATE;
				end if;
			when WAIT_FOR_AFK =>
				if (Kack = '0') then NextState <= WAIT_FOR_PRESS;
				else NextState <= WAIT_FOR_AFK;
				end if;
		end case;
end process;

Kval <= '1' when ( CurrentState = VALIDATE) else '0';
Kscan <= '1' when ( CurrentState = WAIT_FOR_PRESS and Kpress = '0') else '0';

end arcKC;
```

# O. Código *Kotlin – HAL*

```
import isel.leic.UsbPort

object HAL {
    private const val OFF = 0x00
    private const val INIT_STATE = 0x00
    private var lastState = INIT_STATE
    private var initialize = false

    // inicia o usbport a 0 e atualiza o lastState para 0
    fun init() {
        if (!initialize) {
            UsbPort.write(INIT_STATE)
            lastState = INIT_STATE
```

```
            initialize = true
        }
    }

    // le o(s) bit(s) indicado(s) na máscara
    fun readBits(mask: Int) = UsbPort.read().and(mask)

    // verifica se o bit indicado na máscara está on ou off
    fun isBit(mask: Int) = UsbPort.read().and(mask) != OFF

    // coloca o(s) bit(s) indicado(s9 na máscara a on
    fun setBits(mask: Int) {
        lastState = lastState.or(mask)
        UsbPort.write(lastState)
    }

    // coloca o(s) bit(s) indicado(s9 na máscara a off
    fun clrBits(mask: Int) {
        lastState = lastState.and(mask.inv())
        UsbPort.write(lastState)
    }

    // coloca o valor indicado em value no(s) bit(s) indicado(s) na máscara
    fun writeBits(mask: Int, value: Int) {
        lastState = value.or(lastState.and(mask.inv()))
        UsbPort.write(lastState)
    }
}
```

# P. Código *Kotlin - KBD*

```
object KBD {
    private const val Kval = 0x01
    private const val Kack = 0x01
    private const val K = 0x1E
    const val waitTime = 2000L
    private const val sleepTime = 10L
    private const val CODE1 = 0x00
    private const val CODE2 = 0x04
    private const val CODE3 = 0x08
    private const val CODE4 = 0x01
    private const val CODE5 = 0x05
    private const val CODE6 = 0x09
    private const val CODE7 = 0x02
    private const val CODE8 = 0x06
    private const val CODE9 = 0x0A
    private const val CODE0 = 0x07
    private const val CODEEXT = 0x03
    private const val CODEHASH = 0x0B
```

```
    // inicia o HAL e coloca o bit correspondente a K Ack a 0
    fun init() {
        HAL.init()
        HAL.clrBits(Kack)
    }

    const val NONE = 0.toChar()

    // le o que é escrito no teclado e devolve o que leu, caso não consiga ler nada devolve NONE
    private fun getKey(): Int {
        if (HAL.isBit(Kval)) {
            val c = when (HAL.readBits(K).shr(1)) {
                CODE1 -> '1'.code
                CODE2 -> '2'.code
                CODE3 -> '3'.code
                CODE4 -> '4'.code
                CODE5 -> '5'.code
                CODE6 -> '6'.code
                CODE7 -> '7'.code
                CODE8 -> '8'.code
                CODE9 -> '9'.code
                CODEEXT -> '*'.code
                CODE0 -> '0'.code
                CODEHASH -> '#'.code
                else -> NONE.code
            }
            HAL.setBits(Kack)
            while (HAL.isBit(Kval)) {
                Thread.sleep(sleepTime)
            }
            HAL.clrBits(Kack)
            return c
        }
        return NONE.code
    }

    // chama o getKey() até passar o tempo de timeout ou até ler uma tecla
    fun waitKey(timeout: Long): Int {
        val timeInit = System.currentTimeMillis()
        while (true) {
            val time = System.currentTimeMillis()
            val c = getKey()
            if (c != NONE.code) return c
            if (time - timeInit >= timeout) return NONE.code
        }
    }
}
```