1 Keyboard Reader

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.

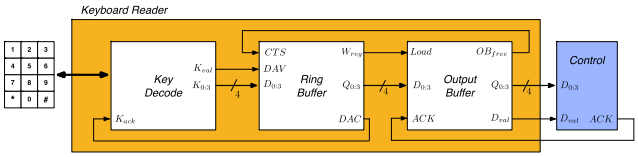
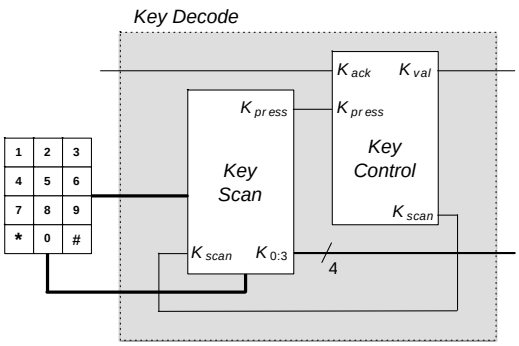


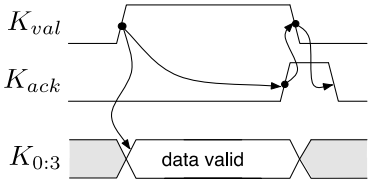
Figura 1 – Diagrama de blocos do módulo *Keyboard Reader*

2 Key Decode

O bloco *Key Decode* implementa um descodificador de um teclado matricial 4x4 por *hardware*, sendo constituído por três sub-blocos: *i)* um teclado matricial de 4x4; *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. E optámos por implementá-lo desta maneira devido à facilidade de implementação consoante os componentes já feitos em semestres anteriores.

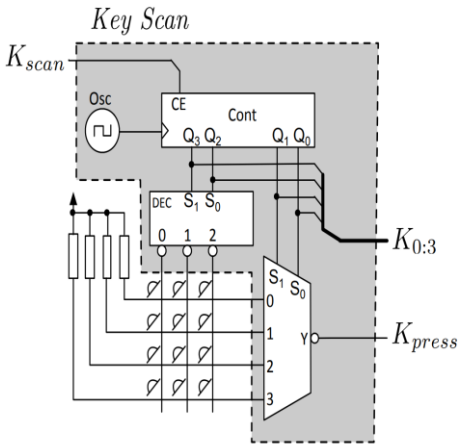


Figura 3 - Diagrama de blocos do bloco *Key Scan*

O bloco *Key Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 4. O mesmo deixo o sinal K_{scan} ativo enquanto aguarda por uma tecla ser premida, tendo o K_{press} a '0' lógico, quando o K_{press} passa a '1' lógico e o K_{ack} mantem-se a 0 a máquina muda de estado e deteta que tem um input premido enviando um sinal K_{val} a '1' lógico, depois a mesma espera o sinal K_{ack} seja recebido para voltar a ativar somente o K_{scan}. A máquina de estados apenas possibilita o envio de um "outro" K_{val} após os sinais K_{ack} e K_{press} voltem ao valor '0' lógico, fazendo com que a máquina volte ao estado inicial.

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

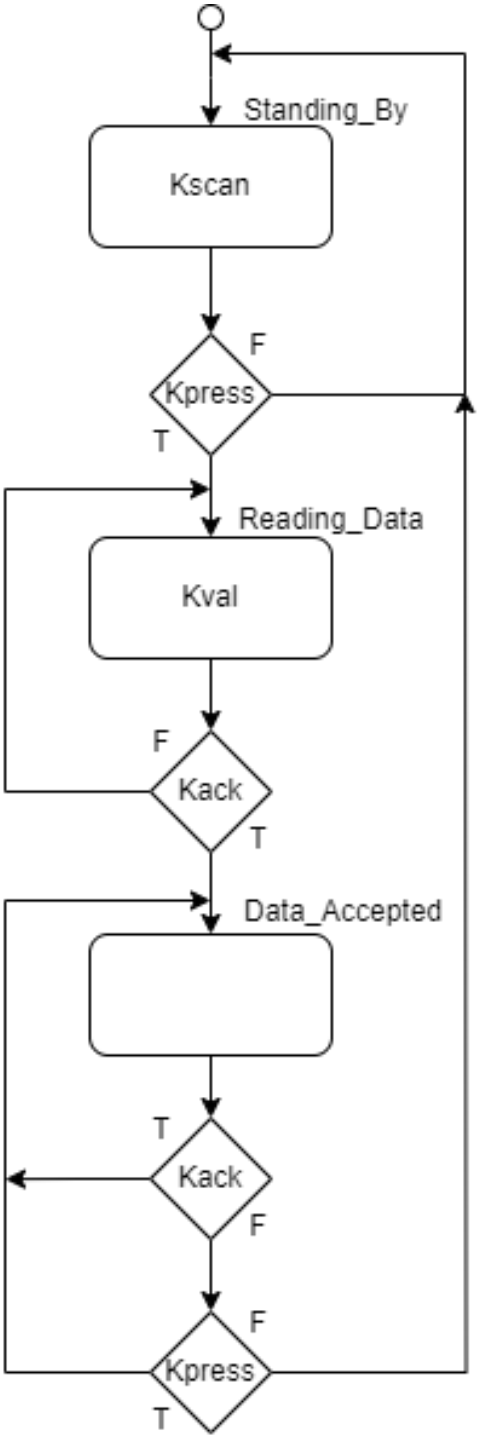


Figura 4 – Máquina de estados do bloco *Key Control*

Com base nas descrições do bloco *Key Decode* implementou-se parcialmente o módulo *Keyboard Reader* de acordo com o esquema elétrico representado no Anexo 0.

O valor das resistências usado foi de 4,7KΩ nas entradas das linhas do teclado e de 330Ω nas entradas das colunas pois é o que é estipulado pela placa *DE-10 Lite Expansion Board*, sendo a das colunas menores do que as das linhas pois estas não recebem tensões diretas de 5V, a mesma passa por outros componentes do circuito podendo de tal modo ser dissipada enquanto que nas linhas a entrada é direta de 5V, servindo também para limitar valores de corrente no circuito, as frequências de relógio usadas foram de 33,97 Hz ou 0,0292 segundos por período de relógio.

3 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 *counters* de 3 bits, que contêm o endereço de escrita e leitura, designados por *putIndex* e *getIndex* respetivamente. O MAC suporta assim ações de *incPut* e *incGet*, que geram informação, demonstrando se a estrutura de dados está cheia (*Full*) ou se está vazia (*Empty*) Na figura 6 é apresentada a estrutura de blocos deste componente. 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*.

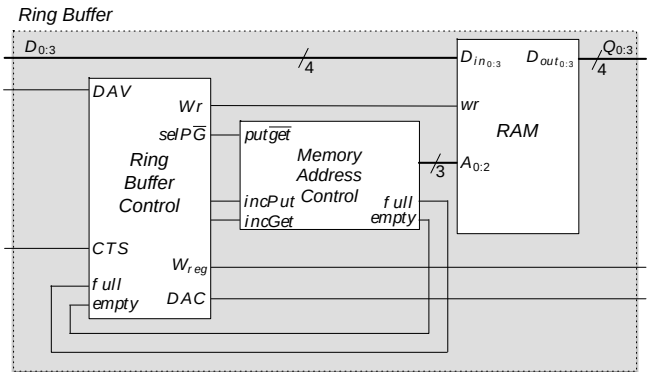


Figura 5 - Diagrama de blocos do bloco *Ring Buffer*

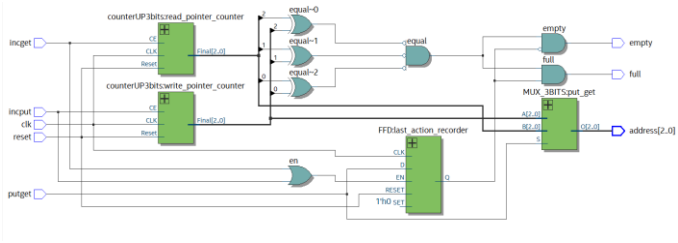


Figura 6 - Diagrama de blocos do bloco *Memory Address Control*

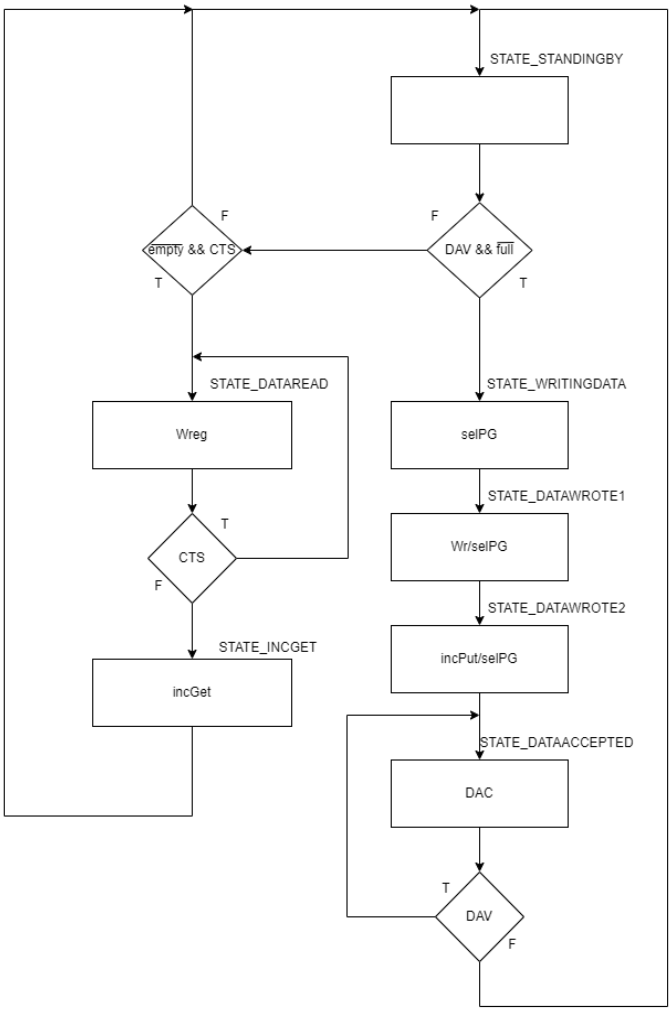


Figura 7 – Máquina de estados do bloco *Ring Buffer Control*

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*.

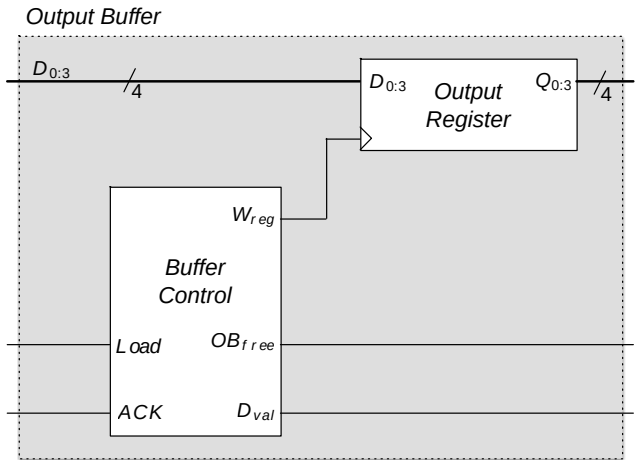


Figura 8 – Diagrama de blocos do *Output Buffer*

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 diagrama de blocos representado na Figura 9. O mesmo encontra-se num estado em que aguarda informação até começar a receber (bit *Load* a '1' lógico) sinalizando uma leitura de dados(ativando D_{val}), quando o mesmo acaba de ler, se o bit *ACK* estiver a valor lógico '0' volta a tentar ler, se o bit for a '1' lógico avança de estado em que aguarda que o *ACK* volte a ser "0" para voltar ao ciclo inicial.

A descrição hardware do bloco *Buffer Control* em VHDL encontra-se no ficheiro "Buffer Control.vhd".

4 Output Buffer

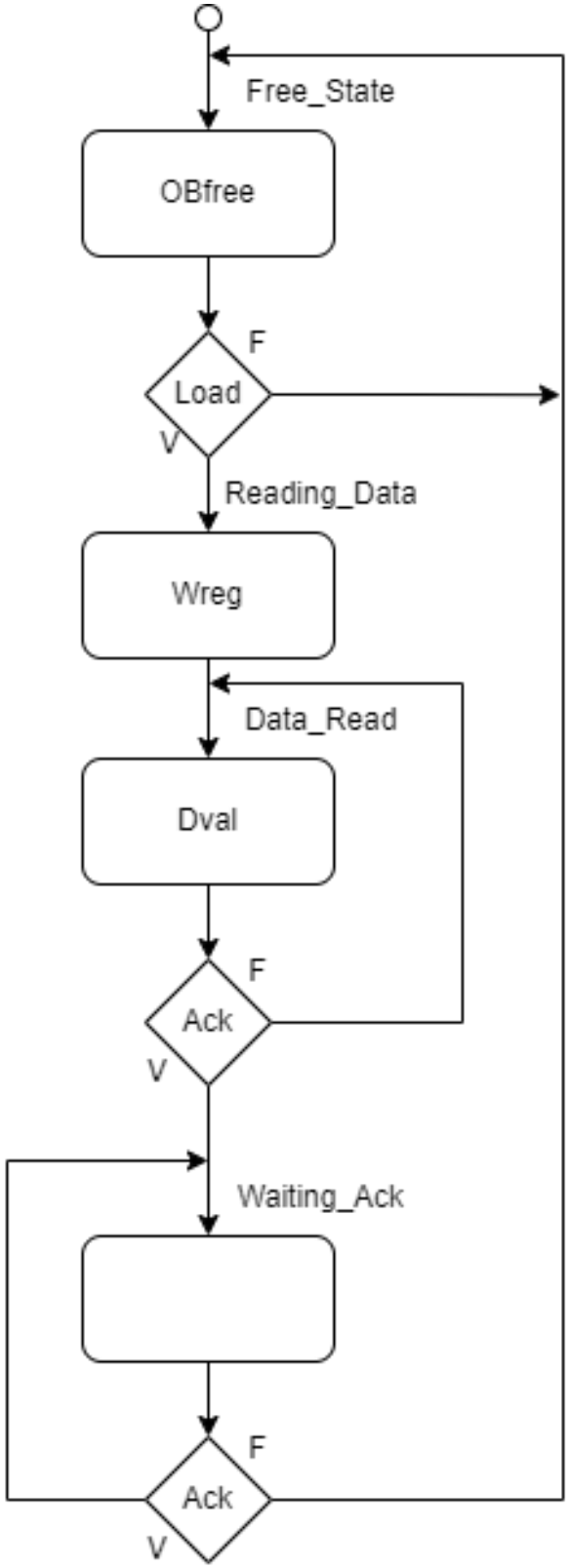


Figura 9 - Máquina de estados do bloco *Buffer Control*

5 Interface com o *Control*

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

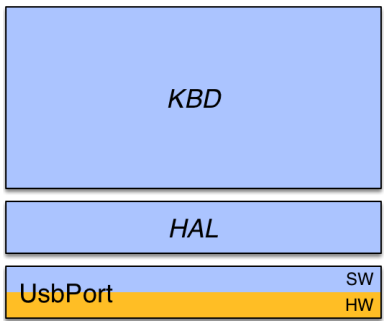


Figura 10 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

As classes *HAL* e *KBD* desenvolvidas são descritas nas secções 5.1. e 5.2, e o código fonte desenvolvido nos Anexos 0 e 0, respetivamente.

5.1 HAL

A classe HAL contém 6 funções: init que inicia a classe, limpando o porto de saída; isBit que retorna true se o bit ou bits referentes à máscara definida como parâmetro de entrada tiverem o valor lógico '1'; readBits que retorna os valores dos bits representados por "mask" presentes no porto de entrada do *UsbPort*; writeBits que escreve nos bits representados por "*mask*", presentes no porto de saída do *UsbPort*, os valores dos bits correspondentes em value; setBits coloca os bits representados por "mask" no valor lógico '1'; clrBits que coloca os bits representados por "mask" no valor lógico '0'. Nenhum método exterior foi implementado.

5.2 KBD

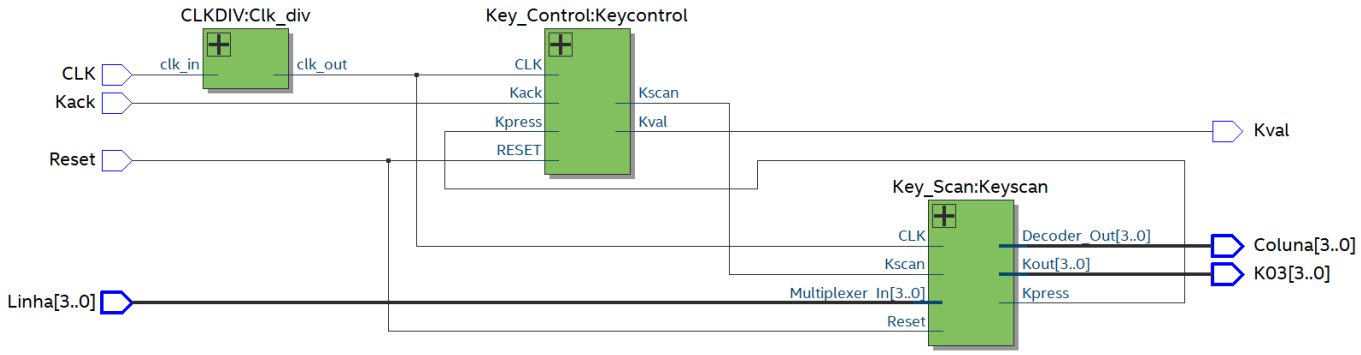
A classe KBD contém 3 funções: getKey que retorna de imediato a tecla premida ou NONE se não for detetada nenhuma tecla; waitKey que retorna a tecla premida, caso ocorra antes do "timeout" (representado em milissegundos), ou NONE caso contrário; init que inicia a classe garantindo que o sinal de *acknowledge* não esteja a ser enviado para o *Hardware*. Nenhum método exterior foi implementado.

6 Conclusões

Após a implementação destes dois módulos, HAL e KBD, entendemos a necessidade da cooperação de ambos: HAL, que permite verificar o porte de "input" e escrever ou apagar no porte de "output". Enquanto a funcionalidade do KBD é de obter a tecla premida utilizando o HAL. Desta forma conseguimos criar a função getKey para verificar o código binário ou hexadecimal que estiver nos portes. Usamos então esse código, convertemos para números naturais ou símbolos para o teclado entender que tecla é suposto acionar.

Para a implementação deste módulo foi necessário o Quartus, para a leitura do nosso código VHDL, como representação do Hardware do módulo e o IntelliJ IDEA de modo a compilar o código em Kotlin, o software do módulo, e o *USBPort* da *Intel*, para fazer a ligação entre estas duas componentes (*Hardware* e *Software*). Para a deteção da tecla premida, tivemos de adicionar alguma latência de modo ao varrimento do clock na placa conseguir passar por todas as teclas, no caso de carregarmos logo após o mesmo ter avançado e de tal modo tiver de passar pelas restantes teclas antes de voltar à tecla premida.

Descrição VHDL do bloco *Key Decode*



Anexo FA: Descrição do bloco Key_Decode

Entidade Key_Decode que define a interface do módulo

```
entity Key_Decode is
    Port (
        CLK : in std_logic; -- Entrada do clock principal
        Linha : in std_logic_vector(3 downto 0); -- Entrada das linhas do teclado
        Kack : in std_logic; -- Entrada do reconhecimento da tecla primida
        Reset : in std_logic; -- Entrada do reset
        Kval : out std_logic; -- Saída do sinal de tecla válida
        Coluna : out std_logic_vector(3 downto 0); -- Saída das colunas do teclado
        K03 : out std_logic_vector(3 downto 0) -- Saída do código da tecla pressionada
    );
end Key_Decode;

architecture arch_keydecode of Key_Decode is

    -- Declaração do componente Key_Scan
    component Key_Scan
        port(
            Kscan : in std_logic; -- Entrada do sinal que avisa que se pode enviar outra tecla
            Multiplexer_In : in std_logic_vector(3 downto 0); -- Entrada das linhas do teclado
            CLK : in std_logic; -- Entrada do clock
            Reset : in std_logic; -- Entrada do reset
            Decoder_Out : out std_logic_vector(3 downto 0); -- Saída do decodificador (colunas do teclado)
            Kout : out std_logic_vector(3 downto 0); -- Saída do código da tecla pressionada
            Kpress : out std_logic -- Saída do sinal de tecla pressionada
        );
```

```
    end component;

    -- Declaração do componente Key_Control
    component Key_Control
        port(
            clk : in std_logic; -- Entrada do clock
            reset : in std_logic; -- Entrada do reset
            kack : in std_logic; -- Entrada do reconhecimento da tecla primida
            kpress : in std_logic; -- Entrada do sinal de tecla pressionada
            kval : out std_logic; -- Saída do sinal de tecla válida
            kscan : out std_logic -- Saída do sinal que avisa que se pode enviar outra tecla
        );
    end component;

    -- Declaração do componente CLKDIV
    component CLKDIV
    generic(div: natural := 50000000);
        port(
            clk_in: in std_logic; -- Entrada do clock div
            clk_out: out std_logic -- Saída do clock div
        );
    end component;



    -- Declaração dos sinais internos
    signal s_kpress, s_kscan, CLK_Divider, not_clk_divider : std_logic;

begin

    -- Inversão do sinal do clock dividido
    not_clk_divider <= not CLK_Divider;

    -- Instanciação do componente Key_Scan
    Keyscan: Key_Scan port map(
        CLK => CLK_Divider, -- Entrada do clock dividido
        Reset => Reset, -- Entrada do sinal de reset
        Kscan => s_kscan, -- Entrada do sinal que avisa que se pode enviar outra tecla
        Multiplexer_In => Linha, -- Entrada das linhas do teclado
```

```
        Decoder_Out => Coluna, -- Saída das colunas do teclado
        Kout => K03, -- Saída do código da tecla pressionada
        Kpress => s_kpress -- Saída do sinal de tecla pressionada
    );

    -- Instanciação do componente Key_Control
    Keycontrol: Key_Control port map(
        clk => not_clk_divider, -- Entrada do clock invertido
        reset => Reset, -- Entrada do sinal de reset
        kscan => s_kscan, -- Saída do sinal de varredura do teclado
        kpress => s_kpress, -- Entrada do sinal de tecla pressionada
        kack => Kack, -- Entrada do reconhecimento da tecla primida
        kval => Kval -- Saída do sinal de tecla válida
    );

    -- Instanciação do componente CLKDIV
    Clk_div : CLKDIV generic map (500000)
    port map(
        clk_in => CLK, -- Entrada do clock principal
        clk_out => CLK_Divider -- Saída do clock dividido
    );

end arch_keydecode;
```

Descrição VHDL do bloco Ring Buffer

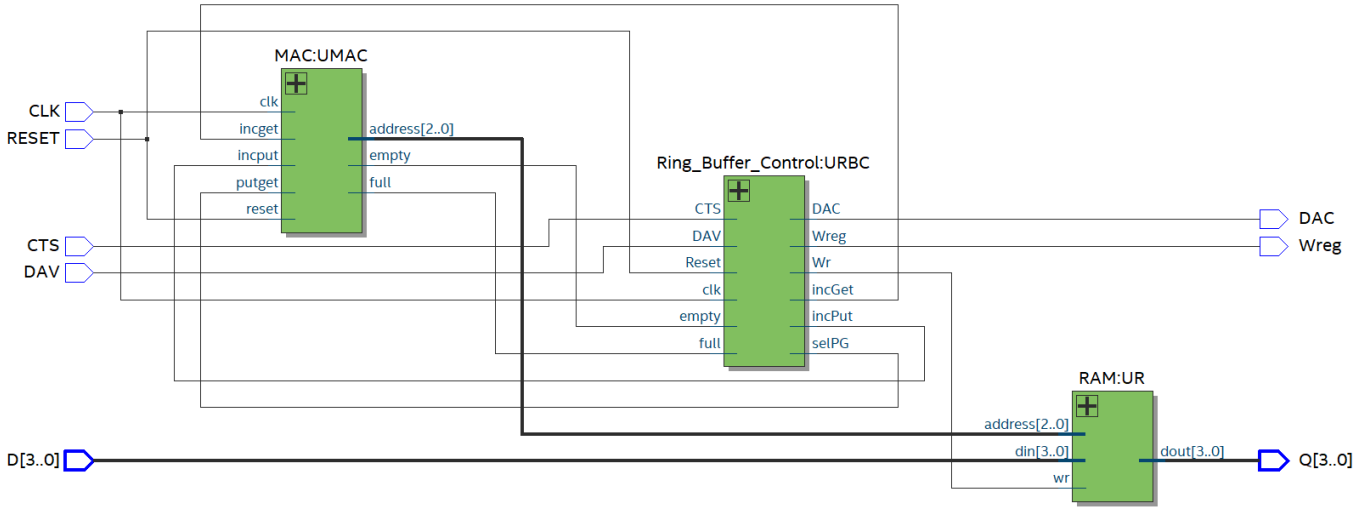


Figura 12: Diagrama de blocos do Ring Buffer

```
entity Ring_Buffer is
    Port (
        CLK : in std_logic;
        CTS : in std_logic;
        D : in std_logic_vector(3 downto 0);
        DAV : in std_logic;
        RESET : in std_logic;
        Wreg : out std_logic;
        Q : out std_logic_vector(3 downto 0);
        DAC : out std_logic
    );
end Ring_Buffer;

architecture Ring_Buffer_arch of Ring_Buffer is

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

component MAC
    Port (
        clk : in std_logic;
```

```
        reset : in std_logic;
        putget : in std_logic;
        incget : in std_logic;
        incput : in std_logic;
        full : out std_logic;
        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;

-- Input Signals
SIGNAL full_s, empty_s : std_logic;

-- Output Signals
SIGNAL Wr_s, selPG_s, incPut_s, incGet_s : std_logic;
SIGNAL address_s : std_logic_vector(2 downto 0);

    begin

    URBC : Ring_Buffer_Control port map(
        clk => CLK,
        Reset => RESET,
        CTS => CTS,
      DAV => DAV,
        full => full_s,
```

```
        empty => empty_s,
        Wreg => Wreg,
        Wr => Wr_s,
        selPG => selPG_s,
        incPut => incPut_s,
        incGet => incGet_s,
        DAC => DAC
    );

    UMAC : MAC port map(
        clk => CLK,
      reset => RESET,
      putget => selPG_s,
        incget => incGet_s,
        incput => incPut_s,
        full => full_s,
        empty => empty_s,
      address => address_s
    );

    UR : RAM port map(

        address => address_s,
        wr => Wr_s,
        din => D,
        dout => Q

    );

end Ring_Buffer_arch;
```

Descrição VHDL do bloco *Output Buffer*

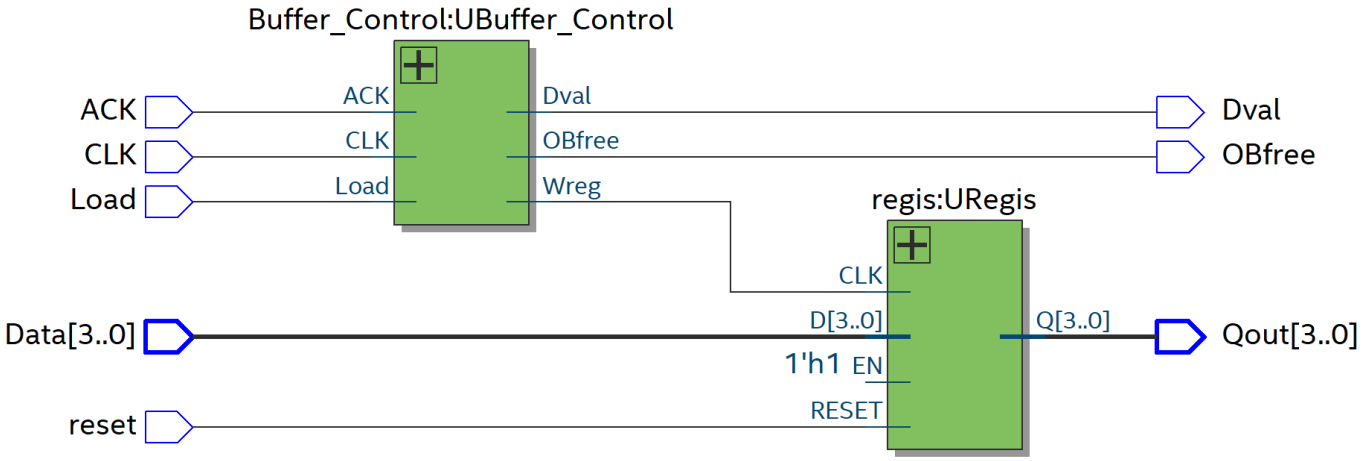


Figura 13: Diagrama de blocos do Output Buffer

```
entity Buffer_Control is
   port(
        CLK : in std_logic;
        ACK : in std_logic;
        Load : in std_logic;
        Wreg : out std_logic;
        OBfree : out std_logic;
        Dval : out std_logic
   );
end entity;

architecture Buffer_Control_arch of Buffer_Control is

   type state is (Free_State, Reading_Data, Data_Read, Waiting_Ack);


   signal current_state, Next_State : state;
   begin

   current_state <= Next_State when rising_edge(clk);


   process(CLK, Load, ACK)
   begin

        case current_state is
        when Free_State =>  if(Load = '1') then
                        Next_State <= Reading_Data;
                        end if;

        when Reading_Data => Next_State <= Data_Read;

        when Data_Read =>   if(Ack = '1') then
                        Next_State <= Waiting_Ack;
                                else Next_State <= Data_Read;
                        end if;

        when Waiting_Ack => if(ACK = '0') then
                        Next_State <= Free_State;
                                else Next_State <= waiting_ACK;
                        end if;
```

```
        end case;
    end process;

    Wreg <= '1' when (current_state = Reading_Data) else '0';
    OBfree <= '1' when (current_state = Free_State) else '0';
    Dval <= '1' when (current_state = DATA_READ) else '0'; --pode haver aqui um or com o waiting_ack state

end Buffer_Control_arch;
```

Descrição VHDL do bloco Keyboard Reader

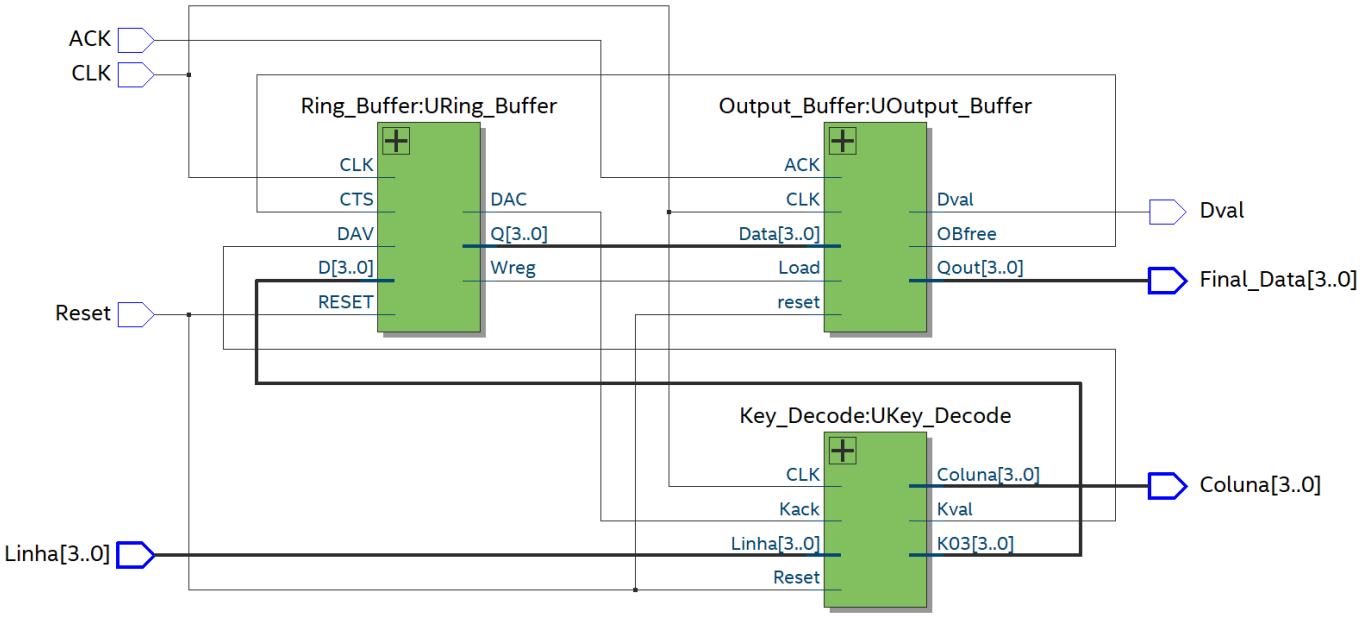


Figura 14: Diagrama de blocos da entidade de topo KeyBoard Reader

```
-- Entidade KeyboardReader que define a interface do módulo
entity KeyboardReader is
    Port (
        CLK : in std_logic; -- Clock do hardware
        Linha : in std_logic_vector(3 downto 0); -- Linhas que saem do teclado
        ACK : in std_logic; -- Sinal de reconhecimento da tecla primida
        Reset : in std_logic; -- Sinal de reset
        Dval : out std_logic; -- Sinal de dados válidos
        Coluna : out std_logic_vector(3 downto 0); -- Colunas do teclado
        Final_Data: out std_logic_vector(3 downto 0) -- Dados finais
    );
end KeyboardReader;

architecture KeyboardReader_arch of KeyboardReader is

    -- Declaração do componente Key_Decode
    component Key_Decode
        Port (
            CLK : in std_logic; -- Clock
            Linha : in std_logic_vector(3 downto 0); -- Linhas do teclado
            Kack : in std_logic; -- Sinal de reconhecimento da tecla primida
            Reset : in std_logic; -- Sinal de reset
            Kval : out std_logic; -- Sinal de tecla válida
```

```
        Coluna : out std_logic_vector(3 downto 0); -- Colunas do teclado
        K03 : out std_logic_vector(3 downto 0) -- Código da tecla pressionada
    );
end component;

-- Declaração do componente Ring_Buffer
component Ring_Buffer
    Port (
        CLK : in std_logic; -- Clock
        CTS : in std_logic; -- Clear to Send (pronto para enviar)
        D : in std_logic_vector(3 downto 0); -- Dados de entrada
        DAV : in std_logic; -- Dados disponíveis
        RESET : in std_logic; -- Reset
        Wreg : out std_logic; -- Sinal de registro de escrita
        Q : out std_logic_vector(3 downto 0); -- Dados de saída
        DAC : out std_logic -- Dados aceitos
    );
end component;

-- Declaração do componente Output_Buffer
component Output_Buffer
    Port (
        CLK : in std_logic; -- Clock
        reset : in std_logic; -- Reset
        Data : in std_logic_vector(3 downto 0); -- Dados de entrada
        Load : in std_logic; -- Sinal de carga
        ACK : in std_logic; -- Sinal de reconhecimento da tecla primida
        Qout : out std_logic_vector(3 downto 0); -- Dados de saída
        OBfree : out std_logic; -- Indicação de buffer livre
        Dval : out std_logic -- Sinal de dados válidos
    );
end component;

-- Declaração de sinais internos
signal Kack, Kval : std_logic;
signal K03 : std_logic_vector(3 downto 0);
signal CTS_S, Wreg_S : std_logic;
signal Ring_Buffer_Out_S : std_logic_vector(3 downto 0);
```

```
begin

    -- Instanciação do componente Key_Decode
    UKey_Decode : Key_Decode port map(
        CLK => CLK,
        Linha => Linha,
        Kack => Kack,
        Reset => Reset,
        Kval => Kval,
        Coluna => Coluna,
        K03 => K03 --
    );

    -- Instanciação do componente Ring_Buffer
    URing_Buffer : Ring_Buffer port map(
        CLK => CLK,
        CTS => CTS_S,
        RESET => Reset,
        D => K03,
        DAV => Kval,
        Wreg => Wreg_S,
        Q => Ring_Buffer_Out_S,
        DAC => Kack
    );

    -- Instanciação do componente Output_Buffer
    UOutput_Buffer : Output_Buffer port map(
        CLK => CLK,
        reset => Reset,
        Data => Ring_Buffer_Out_S,
        Load => Wreg_S,
        ACK => ACK,
        Qout => Final_Data,
        OBfree => CTS_S,
        Dval => Dval
    );
end KeyboardReader_arch;
```

Atribuição de pinos do módulo *Keyboard Reader*

PIN_W5 - Linha [0]
PIN_AA14 - Linha [1]
PIN_W12 - Linha [2]
PIN_AB12 - Linha [3]
PIN_AB11 - Coluna [0]
PIN_AB10 - Coluna [1]
PIN_AA9 - Coluna [2]
PIN_AA8 - Coluna [3]
PIN_N14 - CLK
PIN_C10 - Kack
PIN_C11 - Reset
PIN_A8 - K03 [0]
PIN_A9 - K03 [1]
PIN_A10 - K03 [2]
PIN_B10 - K03 [3]
PIN_B11 - Kval

Código *Kotlin - HAL*

```
object HAL { // Virtualiza o acesso ao sistema UsbPort

    var Output = 0


    // Inicia a classe
    fun init() {
        clrBits(0xFF)
    }


    // Retorna true se o bit tiver o valor lógico ‘1’
    fun isBit(mask: Int): Boolean {
        return (readBits(mask) == mask)
    }
    // Retorna os valores dos bits representados por mask presentes no UsbPort
    fun readBits(mask: Int): Int {
        return UsbPort.read().and(mask)
    }
    // Escreve nos bits representados por mask os valores dos bits correspondentes em value
    fun writeBits(mask: Int, value: Int) {
        Output = (value.and(mask)).or(Output.and(mask.inv()))
            UsbPort.write(Output)
    }

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

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

Código *Kotlin - KBD*

```
object KBD {
    const val KVAL_MASK = 0x10
    const val KACK_MASK = 10000000
    const val DATA_MASK = 0x0F
    const val NONE = ' '
// Inicia a classe
    fun init() {
        HAL.clrBits(KACK_MASK) //Garantir que o Kack é 0 para não termos key's a serem
acknowledged na inicialização
    }

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

        if (HAL.isBit(KVAL_MASK)) {
            val Read_Input = HAL.readBits(DATA_MASK) //data
            var Key = when (Read_Input) {
                0b0000 -> '1'
                0b0100 -> '2'
                0b1000 -> '3'
                0b0001 -> '4'
                0b0101 -> '5'
                0b1001 -> '6'
                0b0010 -> '7'
                0b0110 -> '8'
                0b1010 -> '9'
                0b0111 -> '0'
                0b1011 -> '#'
                0b0011 -> '*'
                else -> NONE
            }
            HAL.setBits(KACK_MASK) //Se houve input, temos de dar ack
            while(HAL.isBit(KVAL_MASK)); //Esperamos que o Kval vá abaixo
            HAL.clrBits(KACK_MASK)

            return Key
        }
        return NONE
    }

    // Retorna a tecla premida, caso ocorra antes do ‘timeout’ (representado em milissegundos), ou
NONE caso contrário.
    fun waitKey(timeout: Long): Char {
        val init = System.currentTimeMillis()


        while (System.currentTimeMillis() < init + timeout) {
            val Key = getKey()
            if(Key != NONE)
                return Key
        }
        return NONE
    }
}
```