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*

# 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 *Kval* é ativado quando é detetada a pressão de uma tecla, sendo também disponibilizado o código dessa tecla no barramento *K0:3*. Apenas é iniciado um novo ciclo de varrimento ao teclado quando o sinal *Kack* 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. Existem 3 possibilidades de o bloco Key Scan ser implementado (versão I, II e III respetivamente). Nesta fase inicial do projeto, decidimos optar pela de mais fácil implementação e que também foi sugerida pelos professores (versão I). Temos como objetivo final implementar uma das outras versões, visto que têm melhores resultados, por exemplo, no caso da versão I temos um contador apenas que manipula 1 decoder e 1 mux, no entanto é melhor ter 2 contadores (como na versão II) devido ao facto de que teremos uma maior taxa de ativação quer do mux quer do decoder, permitindo assim uma maior rapidez de varredura e que sejam perdidas menos teclas.

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

A máquina de estados do Key Control foi implementada com base em 3 estados: **STATE\_DETETAR\_TECLA**, **STATE\_TECLA\_PREMIDA** e **STATE\_ESPERAR\_TECLA**, organizados por ordem temporal. No 1º estado, é ativado a saída **Kscan** de modo a sinalizar que o Key Controle está a detetar se alguma tecla é pressionada; se for pressionada, significa que o sinal **Kpress** tem como valor lógico ‘1’ e assim avançaríamos para o estado **STATE\_TECLA\_PREMIDA**; se tal não acontecer ativamos voltamos ao estado inicial e, como saída condicionada, ativamos também o **Kscan**, uma vez que queremos que o **contador** se mantenha ativo de forma a permitir a constante deteção de teclas. No 2º estado, ativamos a saída **Kval** sinalizando assim que uma tecla foi premida; se esta tecla já foi tratada/reconhecida, significa que o **Kack** se encontra com valor lógico ‘1’ e se de seguida essa mesma tecla já não está a ser pressionada (**Kpress** com valor lógico a ‘0’) avançamos para o último estado **STATE\_ESPERAR\_TECLA**; caso contrário permanecemos no 2º. No 3º e último estado apenas realizamos uma espera pelo sinal **Kack** que tome o valor lógico ‘0’, visto que o **bloco control** não é totalmente síncrono e pode haver casos em que ele ainda não tenha processado uma tecla, no entanto o utilizador já esteja a querer premir outra diferente; caso tome o valor lógico ‘0’, volta para o estado inicial **STATE\_DETETAR\_TECLA**.

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

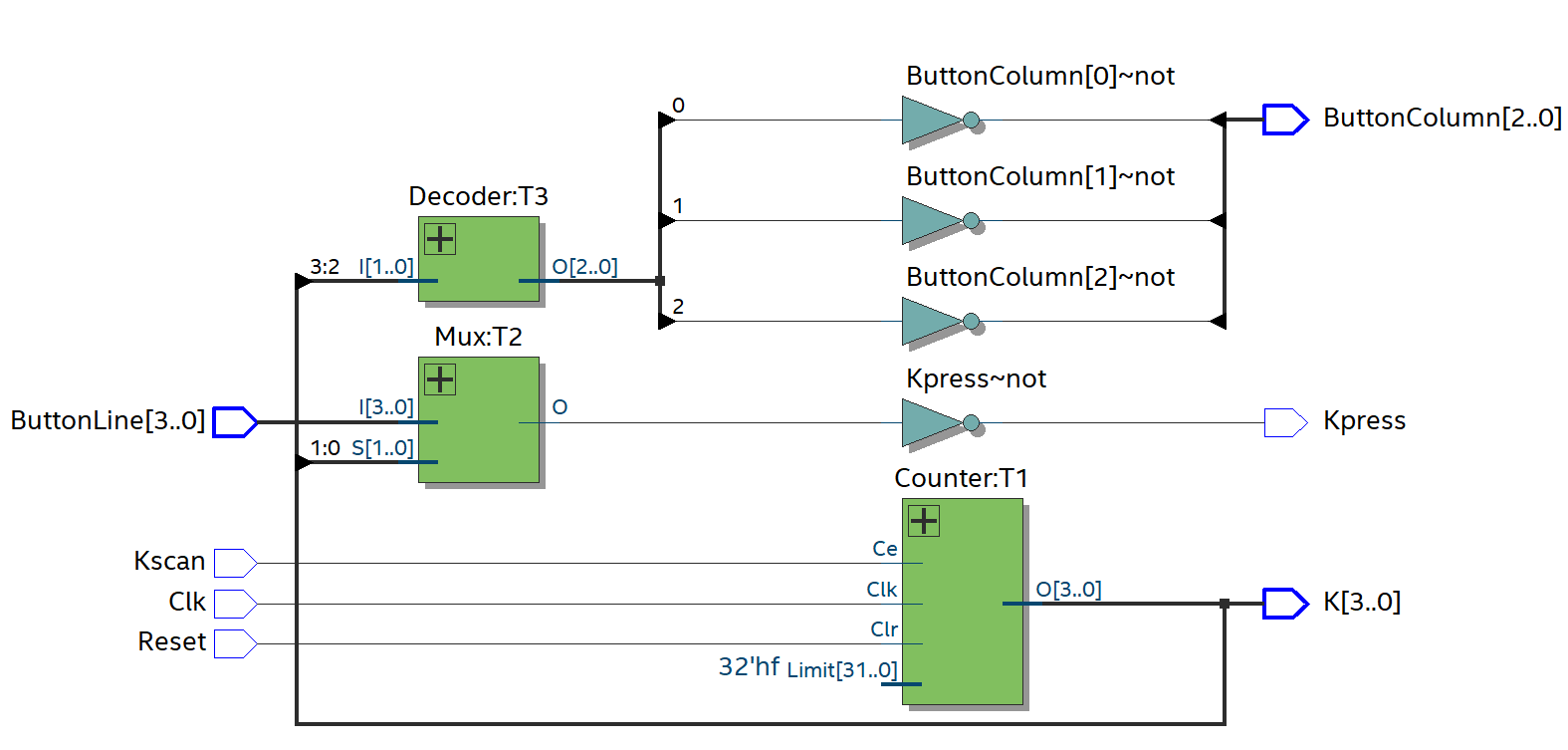


Figura 3 - Diagrama de blocosdo bloco *Key Scan*

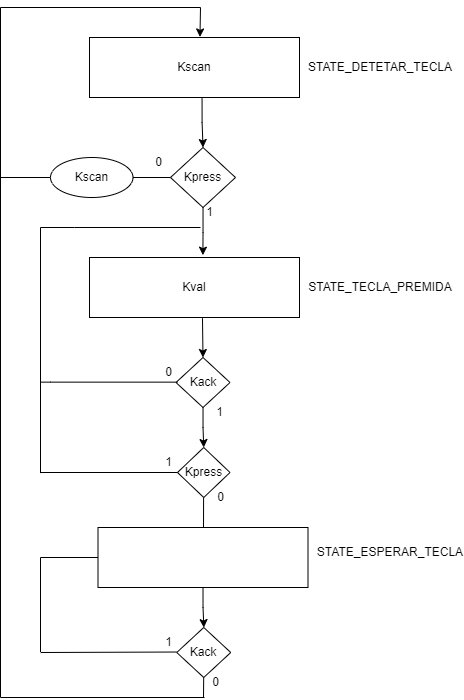


Figura 4 – Máquina de estados do bloco *Key Control*

# Ring Buffer

O bloco Ring Buffer a desenvolver deverá ser 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. A implementação do Ring Buffer deverá ser baseada numa memória RAM (Random Access Memory). O endereço de escrita/leitura, selecionado por deverá ser 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 6 é apresentado o diagrama de blocos para uma estrutura do bloco Ring Buffer.

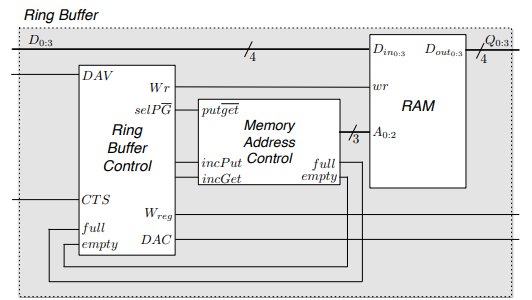


Figura 5 – Diagrama de blocos do bloco Ring Buffer

O bloco *Ring Buffer Control* foi implementado pela máquina de estados representada em *ASM-chart* na 6.

A máquina de estados do Ring Buffer Control implementada com base em 9 estados: **WAITING, ACTIVATE\_PUT, WRITE\_KEY, INC\_PUT, END\_WRITE, READ\_KEY** e **INC\_GET**. No 1º estado, **WAITING**, é verificado o valor lógico do sinal DAV que indica se o *KeyDecode* quer ou não realizar uma escrita de dados. Se este estiver ativo, iremos verificar o valor lógico do sinal *full* de modo a verificar se é possível realizar a escrita de dados e se este estiver de facto ativado significa que o buffer está cheio e assim será necessário que uma leitura seja realizada de modo a que este deixe de estar cheio. Se o caso não for esse é porque então é possível realizar a escrita de dados e assim iremos para o estado **ACTIVATE\_PUT** onde a saída selPnG é ativada de modo a indicar ao MAC (Memory Address Control) que se trata de uma operação de escrita. De seguida vamos para o estado **WRITE\_KEY** onde a saída *selPnG* continua ativa e onde a saída *Wr* vai ser ativada com o intuito de fazer com que a RAM escreva os dados recebidos no seu endereço respetivo (este fornecido pelo MAC). Após este estado vamos para o estado **INC\_PUT** onde a saída incPut vai ser ativada com o objetivo de incrementar o contador de escritas e consequentemente atualizar as saídas *empty* e *full* do MAC. Finalmente, vamos para o estado **END\_WRITE** onde o sinal DAC vai ser ativado, indicando ao *KeyDecode*  que pode receber uma nova key (tecla) e de seguida vai-se verificar o valor lógico do sinal *DAV*. Se este tiver o valor lógico ‘1’ significa que o sistema produtor ainda não se apercebeu que o sinal DAC está ativado, visto que este só mantem o sinal DAV ativado até que o sinal DAC seja ativado. Assim, logo que o sinal DAV deixe de estar ativo iremos de volta para o estado inicial, o estado **STATE\_WAITING**.No entanto, se o sinal *DAV* tiver o valor lógico de ‘0’ significa que se pretende fazer uma leitura de dados e assim iremos verificar o valor lógico do sinal *empty.* Se este estiver ativo significa que queremos realizar uma leitura mas não existem dados para serem lidos, logo ficamos no mesmo estado. No entanto, se este tiver o valor lógico ‘0’ significa que o buffer tem algo para ser lido e assim vamos verificar o valor lógico do sinal *CTS*, enquanto este estiver com o valor lógico ‘0’ vamos permanecer no mesmo estado visto que significa que o *Output Buffer* ainda não está pronto para receber os dados em questão. Quando o sinal *CTS* estiver ativo vamos para o estado **READ\_KEY** onde a saída *Wreg* vai ser ativada de modo a que os dados sejam registados pelo *Output Buffer* e onde se vai verificar novamente o valor do sinal *CTS* mas desta vez permaneceremos no mesmo estado enquanto este estiver ativo, visto que só quando este estiver inativo é que o *Output Buffer* já registou os dados recebidos .Quando o sinal *CTS* estiver com o valor lógico ‘0’ vamos para o estado **INC\_GET** ondea saída *incGet* vai ser ativada de modo a incrementar o contador de leituras no MAC (Memory Address Control) com o objetivo de atualizar as saídas *empty* e *full* do mesmo. Finalmente, iremos de volta para o estado **WAITING**.

O bloco *Memory Address Control Buffer Control* foi implementado de acordo com o diagrama de blocos representado na figura 7.

O bloco *RAM* O bloco *Counter* foi implementado de acordo com o diagrama de blocos representado na figura 8.

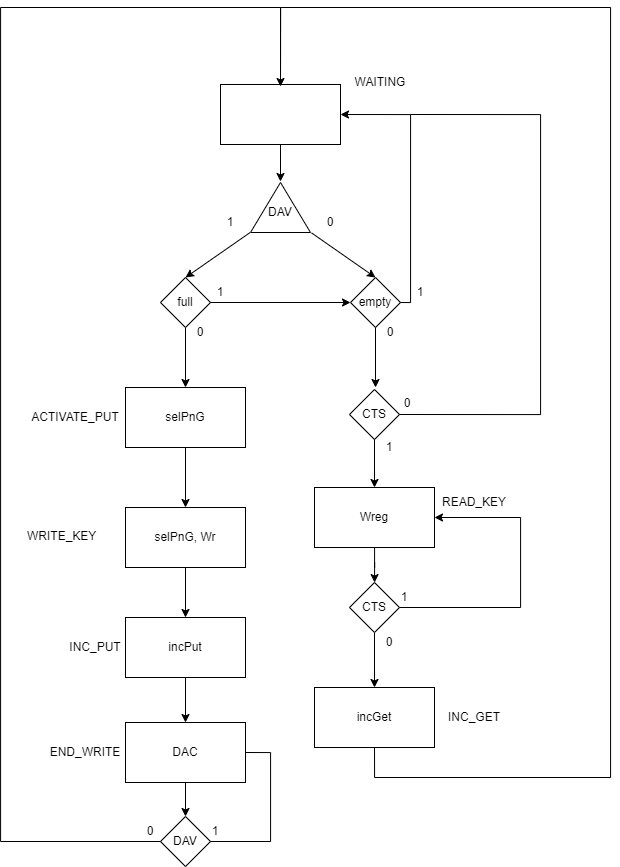


Figura 6 – Máquina de estados do bloco *Ring Buffer Control*

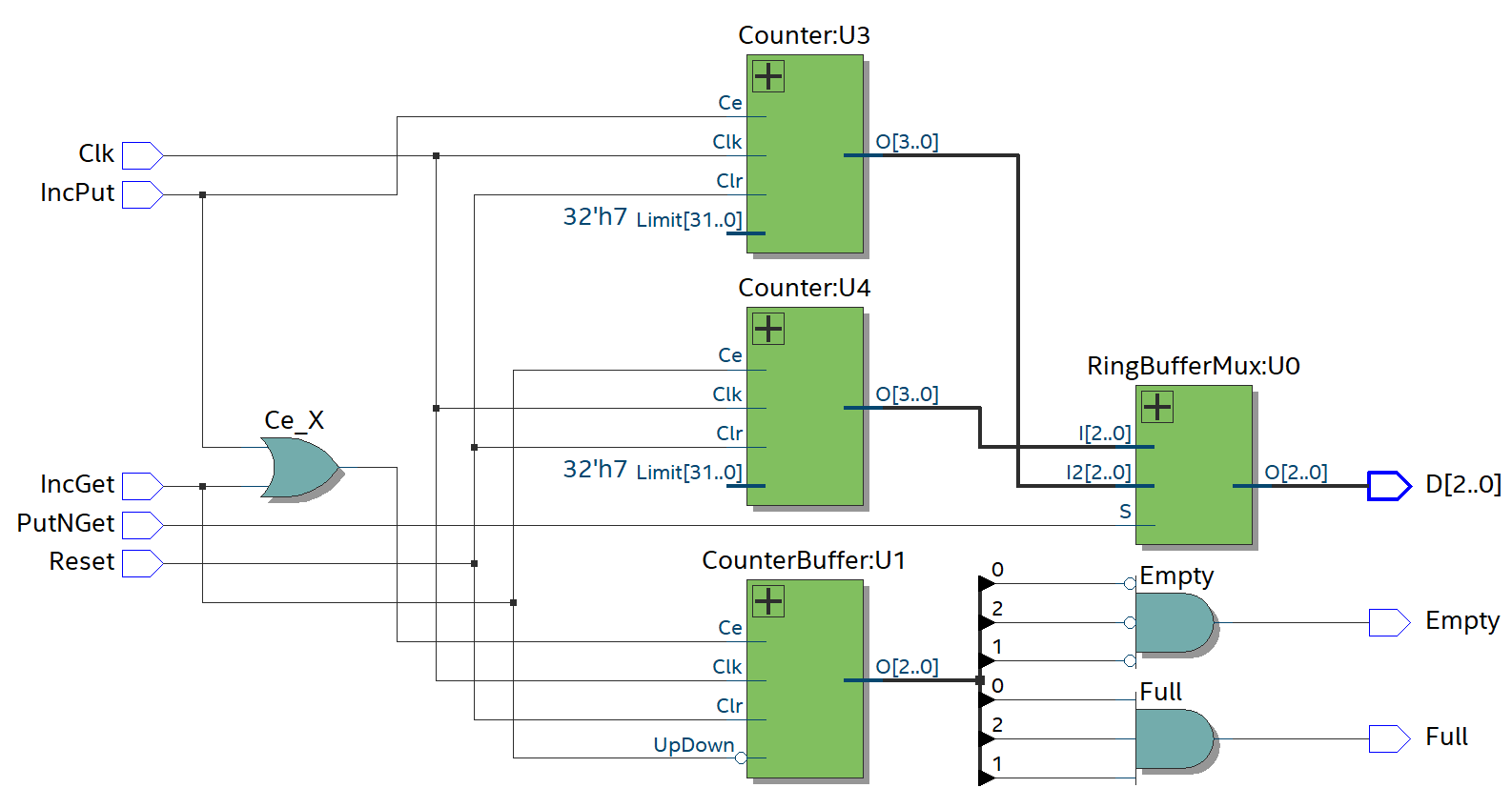
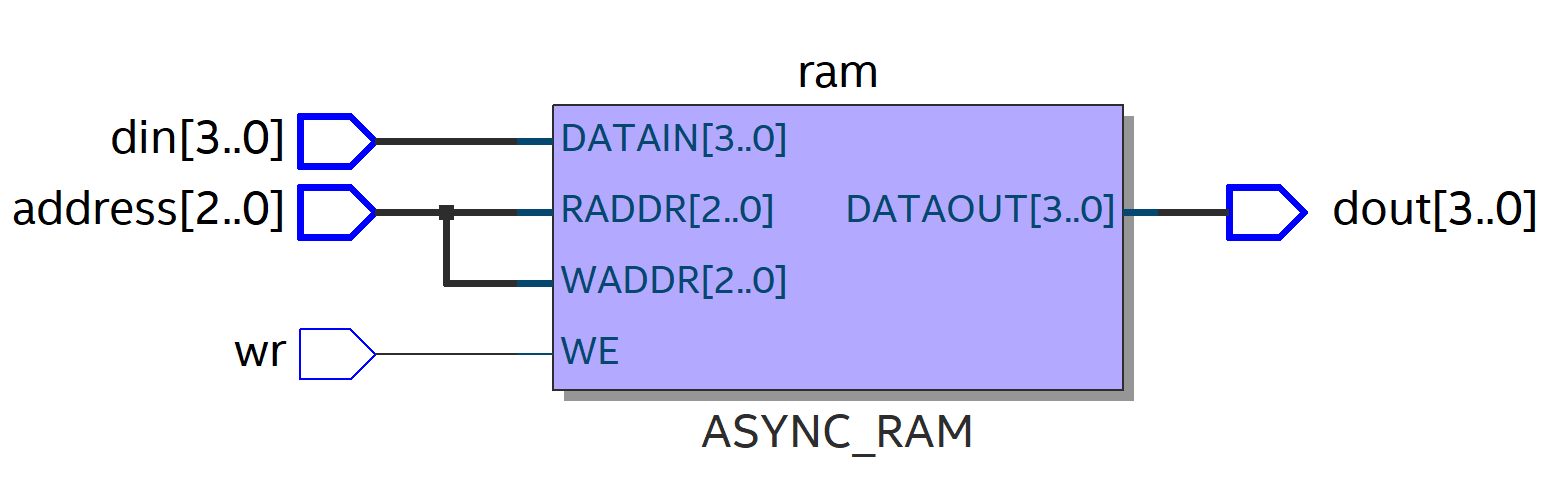
**

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

Figura 8 – Diagrama de blocos do bloco *RAM*

# 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.

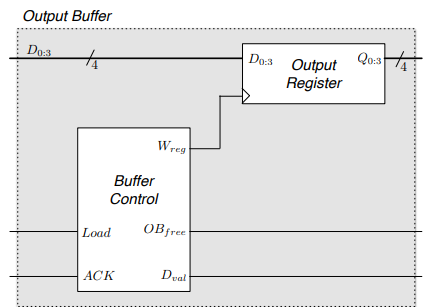


Figura 9 – Diagrama de blocos do bloco *Output Buffer*

Sempre que o bloco emissor Ring Buffer tenha dados disponíveis e o bloco de entrega Output Buffer esteja disponível (OBfree ativo), o Ring Buffer realiza uma leitura da memória e entrega os dados ao Output Buffer ativando o sinal Wreg. O Output Buffer indica que já registou os dados desativando o sinal OBfree.

O bloco *Buffer Control* foi implementado pela máquina de estados representada em *ASM-chart* na 10.

A máquina de estados do *Buffer Control* foi implementada com base em 4 estados : **WAITING, RECEIVING, ACKNOWLEDGED, END,** organizados por ordem temporal. No 1º estado, é ativada a saída OBfree de modo a indicar ao *Ring Buffer* que está disponível para receber uma *key* (tecla) e de seguida verifica-se o valor do sinal *Load* para verificar se os dados fornecidos pelo  *Ring Buffer* são de facto os dados pretendidos. Se este sinal tiver o valor lógico ‘1’ significa que os dados que estão a ser recebidos são os dados pretendidos e assim iremos para o estado **RECEIVING** , se não for esse o caso não nos interessa os dados que estamos a receber. No estado **RECEIVING**  o sinal *Wreg* vai ser ativado com o objetivo de registar os dados recebidos sendo o valor do sinal *Load* novamente avaliado. Enquanto este tiver o valor lógico ‘1’ continuamos no mesmo estado pois significa que os dados que estão a ser recebidos ainda se tratam dos dados que devem ser registados. Quando o sinal *Load* deixar de estar ativo significa que já não se justifica estar a registar os dados recebidos e avançamos para o estado **ACKNOWLEDGED** onde o sinal Dval vai ficar ativo para dar a conhecer ao Software que os dados estão prontos para este os obter e onde de seguida se vai verificar o valor lógico do sinal *Ack* por parte do Software , visto que este fica ativo quando o Software já tiver recebido os dados fornecidos , e assim avançando para o estado **END** onde se irá verificar o valor do sinal *Ack* novamente, mas desta vez permaneceremos no mesmo estado enquanto o sinal *Ack* tiver o valor lógico ‘1’ de modo a garantir que o software já recebeu e processou os dados. Quando o sinal *Ack* deixar de estar ativo iremos de volta para o estado **WAITING**

O bloco *Output Register* foi implementado de acordo com o diagrama de blocos representado na figura 11.

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

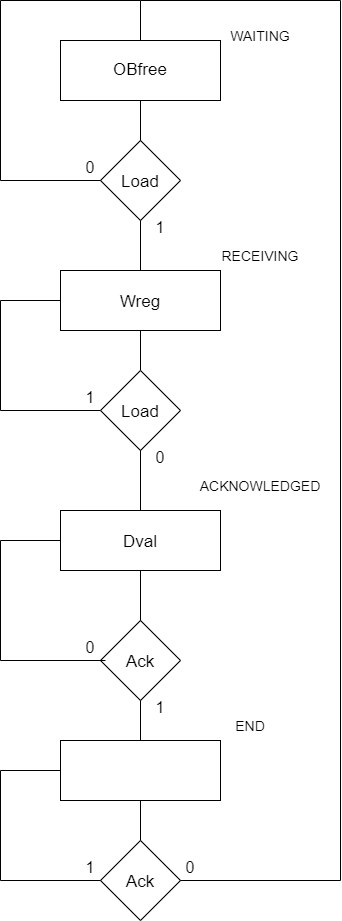


Figura 10 – Máquina de estados do bloco *Buffer Control*

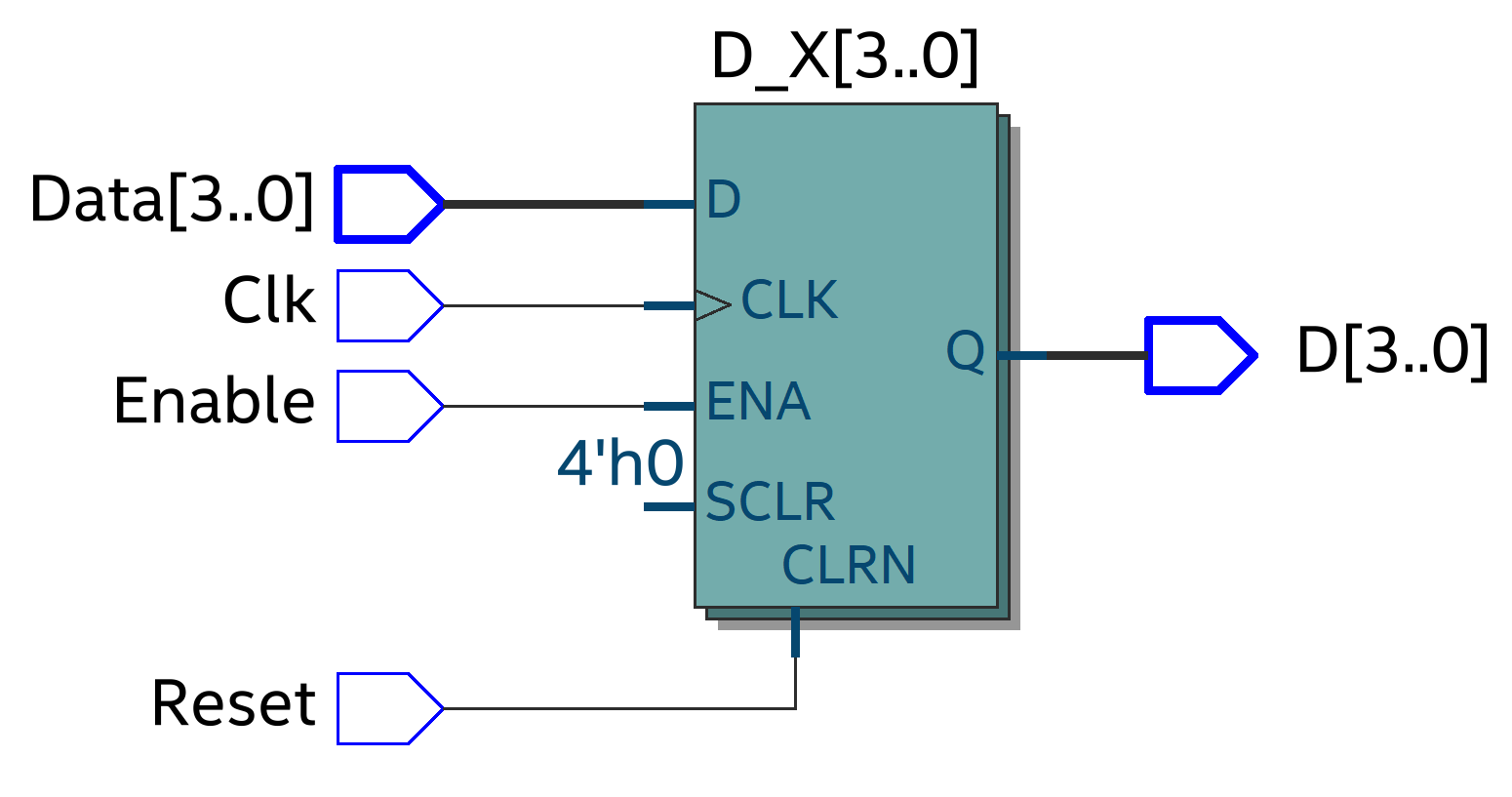
**

Figura 11 – Diagrama de blocos do bloco *Output Register*

Com base nas descrições do bloco *Key Decode, Ring Buffer e Output Buffer* implementou-se o módulo *Keyboard Reader* de acordo com o esquema elétrico representado no Anexo C. Para este módulo, é usado um único sinal de Clk o qual é mapeado para a placa DE10-Lite com uma frequência de 50MHz.

# Interface com o *Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem *Kotlin* e seguindo a arquitetura lógica apresentada na Figura 12.



Figura 12 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

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

## *HAL*

O objeto HAL, tem como objetivo servir de ‘ponto de ligação’ entre o UsbPort e o Hardware, ou seja, permitir o funcionamento entre os diversos blocos de código implementados com o Hardware e o software desenvolvido em kotllin. Para isso foram desenvolvidas várias funções como a **isBit**, **readBits**, **writeBits**, **setBits** e **clearBits** que manipulam os bits de entrada e saída do UsbPort. Para se ter noção da última escrita nos bits de saída foi também adicionada uma variável **lastWriting**.

## *KBD*

O objeto KBD ou Keyboard Decode serve para dados 4 bits do UsbPort, e “traduzi-los” para uma tecla, que na nossa implementação, estes 4 bits representam a posição num array e assim conseguimos saber qual tecla está a ser pressionada. Para atingir este objetivo foram criadas 2 funções, **getKey** e **waitKey** e uma vez que este objeto não interage diretamente com o UsbPort, ele usa as funções fornecidas pelo HAL para poder recolher os bits e tratá-los.

# Conclusões

Neste módulo, foi necessário implementar um descodificador do teclado (KeyDecoder) responsável por receber os códigos de cada tecla proveniente do teclado e convertê-los de acordo com a sua posição no mesmo. Para além do KeyDecoder, também foi necessário implementar um bloco de armazenamento (Ring Buffer) e um bloco de entrega ao consumidor (Output Buffer). O bloco de armazenamento é responsável por armazenar teclas com disciplina FIFO (*First In First Out*) e com capacidade de armazenar até oito palavras de quatro bits. Por fim, o bloco de entrega ao consumidor é responsável pela interação com o sistema consumidor, neste caso o módulo Control ao qual entrega os códigos que são depois processados de acordo com a funcionalidade pretendida, dando uso ao bloco de Software.

Para uma correta implementação destes módulo e dos seus respetivos submódulos (KeyScan e KeyControl, Ring Buffer, Memory Address Control, Buffer Control e Output Register) foram realizados testes tanto no simulador do Quartus como na própria placa DE10-Lite, dando uso ao bloco de Software.

Para certos módulos, foram utilizadas diferentes abordagens ao VHDL, como por exemplo no módulo Counter onde abdicamos de vários submódulos de hardware (Adders4Bits, FFD4Bits) e optamos por uma abordagem mais simples, utilizando apenas um contador implementado com programação lógica.

Um dos problemas encontrados foi o Bouncing das teclas que é um problema que ocorre quando uma tecla é pressionada e libertada, fazendo com que o sinal de entrada oscile entre 0 e 1 e o sistema interprete que a tecla foi pressionada várias vezes.

2. Descrição VHDL do bloco *Key Decode*

library ieee;

use ieee.std\_logic\_1164.all;

entity KeyDecode is

port

(

-- Input ports

Kack : in std\_logic;

Clk : in std\_logic;

Reset : in std\_logic;

ButtonLine : in std\_logic\_vector(3 downto 0);

-- Output ports

Kval : out std\_logic;

K : out std\_logic\_vector(3 downto 0);

ButtonColumn : out std\_logic\_vector(2 downto 0)

);

end KeyDecode;

architecture structural of KeyDecode is

component KeyScan is

port

(

-- Input ports

Kscan : in std\_logic;

Clk : in std\_logic;

ButtonLine : in std\_logic\_vector(3 downto 0);

-- Output ports

K : out std\_logic\_vector(3 downto 0);

Kpress : out std\_logic;

ButtonColumn : out std\_logic\_vector(2 downto 0)

);

end component;

component KeyControl is

port

(

-- Input ports

Kpress : in std\_logic;

Kack : in std\_logic;

Clk : in std\_logic;

Reset : in std\_logic;

-- Output ports

Kscan : out std\_logic;

Kval : out std\_logic

);

end component;

signal Kscan\_X, Kpress\_X : std\_logic;

begin

F1: KeyScan port map(Kscan => Kscan\_X, Clk => Clk, ButtonLine => ButtonLine,

K => K, Kpress => Kpress\_X, ButtonColumn => ButtonColumn);

F2: KeyControl port map(Kpress => Kpress\_X, Kack => Kack, Clk => Clk, Reset => Reset,

Kscan => Kscan\_X, Kval => Kval);

end structural;

A. Descrição VHDL do bloco *Ring Buffer*

library ieee;

use ieee.std\_logic\_1164.all;

entity RingBuffer is

port

(

-- Input ports

Clk : in std\_logic;

Reset : in std\_logic;

D : in std\_logic\_vector(3 downto 0);

DAV : in std\_logic;

CTS : in std\_logic;

-- Output ports

Q : out std\_logic\_vector(3 downto 0);

Wreg : out std\_logic;

DAC : out std\_logic

);

end RingBuffer;

architecture structural of RingBuffer is

component RingBufferControl is

port

(

-- Input ports

Clk : in std\_logic;

Reset : in std\_logic;

DAV : in std\_logic;

CTS : in std\_logic;

Full : in std\_logic;

Empty : in std\_logic;

-- Output ports

Wr : out std\_logic;

Wreg : out std\_logic;

selPnG : out std\_logic;

incPut : out std\_logic;

incGet : out std\_logic;

DAC : out std\_logic

);

end component;

component MemoryAddressControl is

port

(

-- Input ports

Clk : in std\_logic;

PutNGet : in std\_logic;

IncPut : in std\_logic;

IncGet : in std\_logic;

Reset : in std\_logic;

-- Output ports

D : out std\_logic\_vector(2 downto 0);

Full : out std\_logic;

Empty : out std\_logic

);

end component;

component RAM is

generic

(

ADDRESS\_WIDTH : natural := 3;

DATA\_WIDTH : natural := 4

);

port

(

-- Input ports

address : in std\_logic\_vector(ADDRESS\_WIDTH - 1 downto 0);

wr : in std\_logic;

din : in std\_logic\_vector(DATA\_WIDTH - 1 downto 0);

-- Output ports

dout : out std\_logic\_vector(DATA\_WIDTH - 1 downto 0)

);

end component;

signal Full\_X, Empty\_X, Wr\_X, selPnG\_X, incPut\_X, incGet\_X : std\_logic;

signal Address\_X : std\_logic\_vector(2 downto 0);

begin

U0: RingBufferControl port map (Clk => Clk, Reset => Reset, DAV => DAV, CTS => CTS, Full => Full\_X, Empty => Empty\_X,

Wr => Wr\_X, Wreg => Wreg, selPnG => selPnG\_X, incPut => incPut\_X, incGet => incGet\_X, DAC => DAC);

U1: MemoryAddressControl port map (Clk => Clk, Reset => Reset, PutnGet => selPnG\_X, IncPut => incPut\_X, IncGet => incGet\_X,

D => Address\_X, Full => Full\_X, Empty => Empty\_X);

U2: RAM port map (address => Address\_X, wr => Wr\_X, din => D,

dout => Q);

end structural;

1. Descrição VHDL do bloco *Output Buffer*

library ieee;

use ieee.std\_logic\_1164.all;

entity OutputBuffer is

port

(

-- Input ports

Clk : in std\_logic;

Load : in std\_logic;

Ack : in std\_logic;

Reset : in std\_logic;

D : in std\_logic\_vector(3 downto 0);

-- Output ports

Q : out std\_logic\_vector(3 downto 0);

Dval : out std\_logic;

OBfree : out std\_logic

);

end OutputBuffer;

architecture structural of OutputBuffer is

component BufferControl is

port

(

-- Input ports

Clk : in std\_logic;

Load : in std\_logic;

Ack : in std\_logic;

Reset : in std\_logic;

-- Output ports

Wreg : out std\_logic;

OBfree : out std\_logic;

Dval : out std\_logic

);

end component;

component OutputRegister is

port

(

-- Input ports

Clk : in std\_logic;

Enable : in std\_logic;

Reset : in std\_logic;

Data : in std\_logic\_vector(3 downto 0);

-- Output ports

D : out std\_logic\_vector(3 downto 0)

);

end component;

signal Wreg\_X : std\_logic;

signal O\_X : std\_logic\_vector(3 downto 0);

begin

U0: BufferControl port map (Load => Load, Ack => Ack, OBfree => OBfree, Dval => Dval, Wreg => Wreg\_X, Clk => Clk, Reset => Reset);

U1: OutputRegister port map (Clk => Wreg\_X, Reset => Reset, Data => D, Enable => '1',

D => Q);

end structural;

1. Atribuição de pinos do módulo *Keyboard Reader*

set\_location\_assignment PIN\_W5 -to ButtonLine[0]

set\_location\_assignment PIN\_AA14 -to ButtonLine[1]

set\_location\_assignment PIN\_W12 -to ButtonLine[2]

set\_location\_assignment PIN\_AB12 -to ButtonLine[3]

set\_location\_assignment PIN\_AB11 -to ButtonColumn[0]

set\_location\_assignment PIN\_AB10 -to ButtonColumn[1]

set\_location\_assignment PIN\_AA9 -to ButtonColumn[2]

set\_location\_assignment PIN\_P11 -to Clk

set\_location\_assignment PIN\_A8 -to Q[0]

set\_location\_assignment PIN\_A9 -to Q[1]

set\_location\_assignment PIN\_A10 -to Q[2]

set\_location\_assignment PIN\_B10 -to Q[3]

set\_location\_assignment PIN\_B11 -to Dval

set\_location\_assignment PIN\_C10 -to ACK

set\_location\_assignment PIN\_F15 -to Reset

1. Código *Kotlin* - *HAL*

import isel.leic.UsbPort

// Virtualiza o acesso ao sistema UsbPort

object HAL {

private var lastWriting = 0

// Inicia a classe

fun init() {

UsbPort.write(lastWriting)

}

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

fun isBit(mask: Int): Boolean {

val temp = mask and UsbPort.read()

return mask == temp

}

// Retorna os valores dos bits representados por mask presentes no UsbPort

fun readBits(mask: Int): Int = mask and UsbPort.read()

// Escreve nos bits representados por mask o valor de value

fun writeBits(mask: Int, value: Int) {

val a = mask and value

val b = mask.inv() and lastWriting

val c = a or b

UsbPort.write(c)

lastWriting = c

}

// Coloca os bits representados por mask no valor lógico ‘1’

fun setBits(mask: Int) {

writeBits(mask,0xFF)

}

// Coloca os bits representados por mask no valor lógico ‘0’

fun clearBits(mask:Int) {

writeBits(mask,0x00)

}

}

fun main() {

val mask = 0b00001111

HAL.init()

HAL.setBits(mask)

Thread.sleep(2000)

HAL.clearBits(mask)

Thread.sleep(2000)

println(HAL.isBit(mask))

Thread.sleep(2000)

HAL.writeBits(mask, 6)

Thread.sleep(2000)

// Change the value of the input port bits

val currentBits = HAL.readBits(mask)

println(currentBits)

}

1. Código *Kotlin* - *KBD*

import isel.leic.utils.Time

object KBD { // Ler teclas. Métodos retornam ‘0’..’9’,’#’,’\*’ ou NONE.

private const val NONE = 0

private val arrayTeclas = arrayListOf('1', '4', '7', '\*', '2', '5', '8', '0', '3', '6', '9', '#')

// Inicia a classe

fun init() {

HAL.init()

HAL.clearBits(ACK\_MASK)

}

// Retorna de imediato a tecla premida ou NONE se não há tecla premida.

fun getKey(): Char {

if (!HAL.isBit(DVAL\_MASK)) {

return NONE.toChar()

}

val key = HAL.readBits(Q3\_\_0\_MASK)

while (HAL.isBit(DVAL\_MASK)) { HAL.setBits(ACK\_MASK) }

HAL.clearBits(ACK\_MASK)

return arrayTeclas[key]

}

// Retorna a tecla premida, caso ocorra antes do ‘timeout’ (representado em milissegundos), ou NONE caso contrário.

fun waitKey(timeout: Long): Char {

var key = NONE.toChar()

val endTime = Time.getTimeInMillis() + timeout

while (endTime >= Time.getTimeInMillis()) {

key = getKey()

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

}

return key

}

}

fun main() {

KBD.init()

while (true) {

val key = KBD.waitKey(5000)

if (key != 0.toChar()) {

println("Key pressed: $key")

}

}

}