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.

Uma imagem com diagrama, esquemático

Descrição gerada automaticamente

Figura 3 - Diagrama de blocosdo bloco *Key Scan*

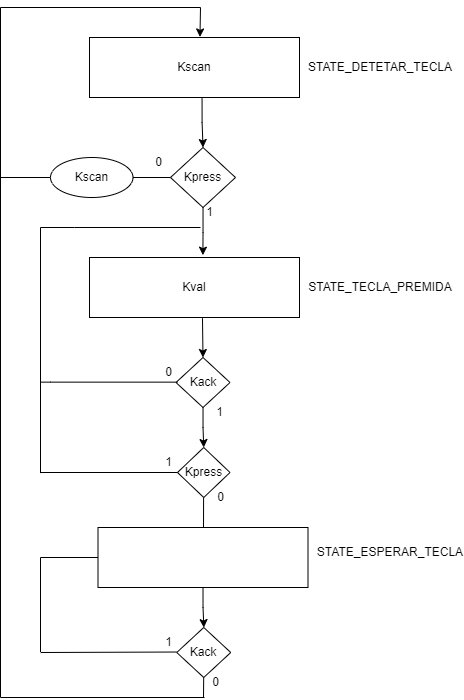


Figura 4 – Máquina de estados do bloco *Key Control*

Com base nas descrições do bloco *Key Decode* implementou-se parcialmente o módulo *Keyboard Reader* de acordo com o esquema elétrico representado no Anexo 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 8.



Figura 8 – 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, numa fase mais inicial, 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. Por fim, estes códigos eram entregues à entidade consumidora (Control), onde eram processados de acordo com a funcionalidade pretendida.

Para uma correta implementação deste módulo e dos seus respetivos submódulos (KeyScan e KeyControl) 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;

Descrição VHDL do bloco *Ring Buffer*

1. Descrição VHDL do bloco *Output Buffer*
2. 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 K[0]

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

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

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

set\_location\_assignment PIN\_B11 -to Kval

set\_location\_assignment PIN\_C10 -to Kack

set\_location\_assignment PIN\_C11 -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(D3\_\_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")

}

}

**}**