O módulo *Serial LCD Controller* é constituído por ? blocos principais dos quais apenas são objeto deste relatório os seguintes:

i) 1
ii) 2
iii) *3*

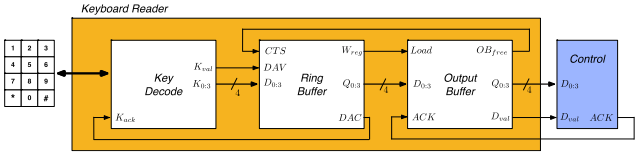


Figura 1 – Diagrama de blocos do módulo *Keyboard Reader*

1 ?

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;
iii) o bloco *Key Control*, que realiza o controlo do varrimento e o controlo de fluxo, conforme o diagrama de blocos representado na Figura 2. 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.

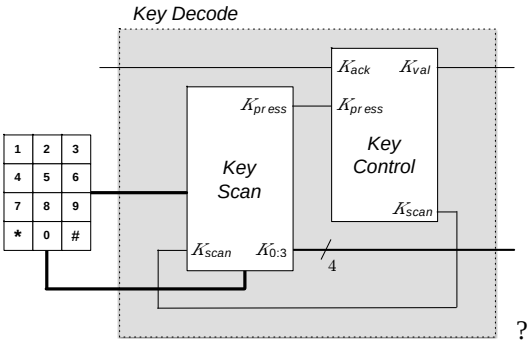
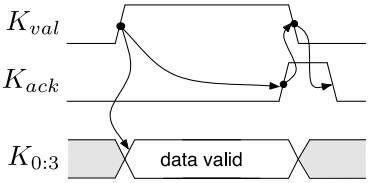


Figura 2 – Diagrama de blocos Key Decode



b) Diagrama temporal

Figura 2 – Bloco *Key Decode*

O bloco *Key Scan*, por agora implementado, integra-se no processo de evolução de implementação de código VHDL, por ser o que nos pareceu mais simples e, que se apresenta no diagrama de blocos na Figura 3. **[Adicionar a justificação da opção tomada.]**

O bloco *Key Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 4. **[Adicionar a descrição da solução apresentada.]**

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

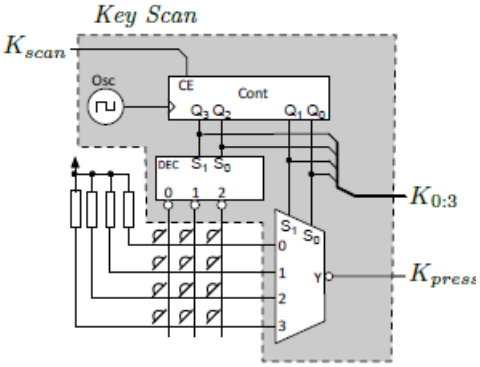


Figura 3 - Diagrama de blocos do bloco *Key Scan*

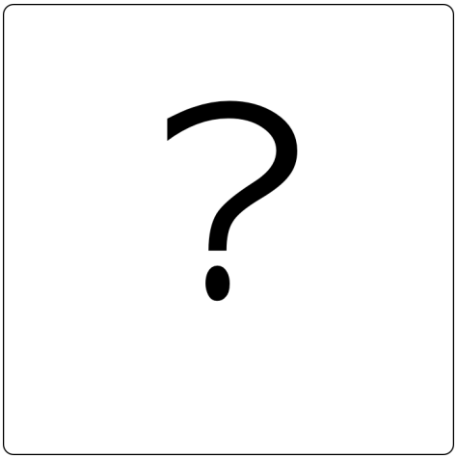


Figura 4 – Máquina de estados do bloco *Key Control*

Com base nas descrições do bloco *Key Decode* implementou-se parcialmente o módulo *Keyboard Reader* de acordo com o esquema elétrico representado no Anexo D. **[Justificar as opções tomadas, como por exemplo, o valor das resistências, as frequências de relógio, etc.]**

2 ?

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 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 5 é apresentado o diagrama de blocos para a estrutura do bloco *Ring Buffer*.

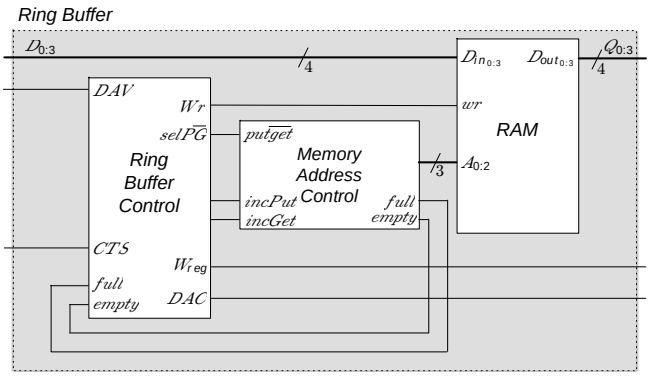


Figura 5 - Diagrama de blocos do bloco *Ring Buffer*

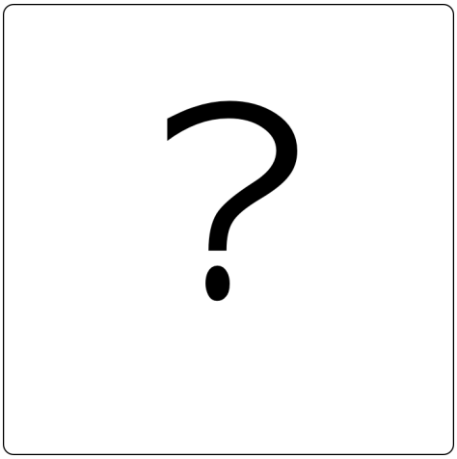


Figura 6 - Diagrama de blocos do ?

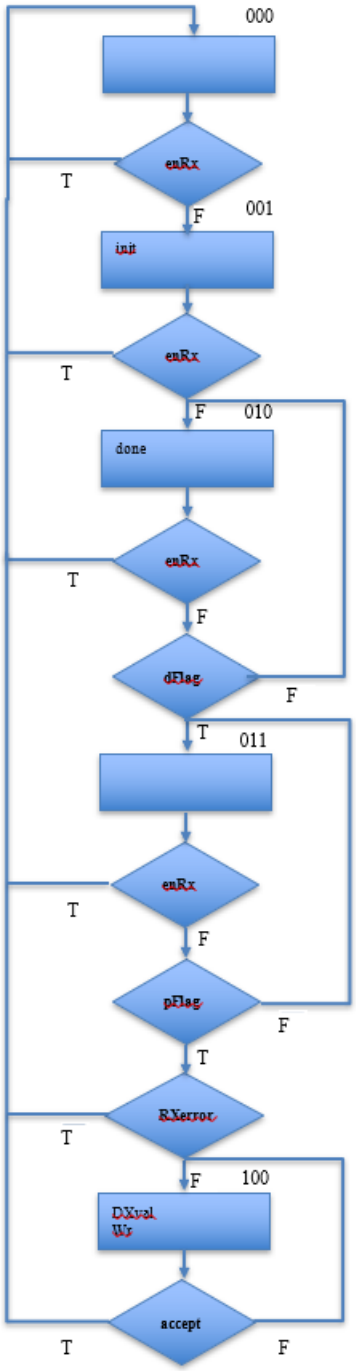


Figura 7 – Máquina de estados do bloco

3 ?

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*.

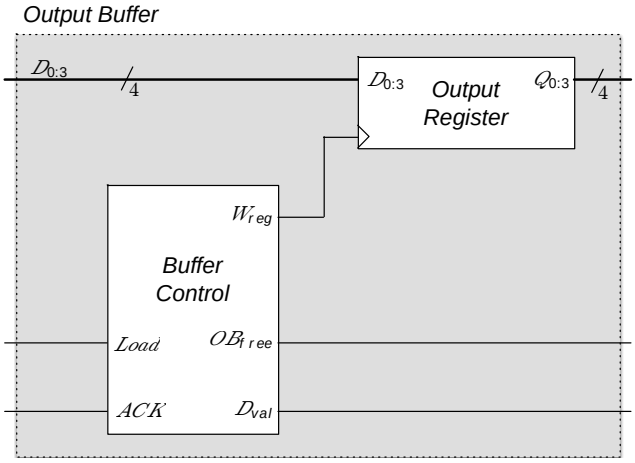


Figura 8 – Diagrama de blocos do *Output Buffer*

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. **[Adicionar a justificação da opção tomada.]**

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

?

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

Com base nas descrições do bloco *Key Decode* e do bloco *Key Buffer Control* implementou-se o módulo *Keyboard Reader* de acordo com o esquema elétrico representado no Anexo D. **[Justificar as opções tomadas, como por exemplo, as frequências de relógio, etc.]**

4 ?

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

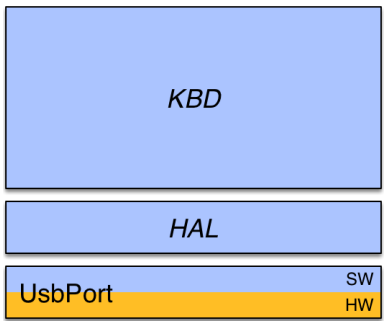


Figura 10 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

As classes *HAL* e *KBD* desenvolvidas são descritas nas secções 4.1. e **Error! Reference source not found.**, e o código fonte desenvolvido nos Anexos E e **Error! Reference source not found.**, respetivamente.

4.1 Classe *LCD*

[Descrever nesta secção de forma sucinta a classe, referindo se adicionaram métodos.]

5 Conclusões

[Escrever nesta secção as conclusões da implementação do modulo, incluindo os recursos utilizados, latência na de deteção de tecla, etc.]

A. Descrição VHDL do bloco Serial LCD Controller??

B. Descrição VHDL do bloco ?

C.

D. Atribuição de pinos do módulo ?

E. Código *Kotlin - LCD*

```
import HAL

object LCD { // Escreve no LCD usando a interface a 4 bits.
    // Dimensão do display.
    private const val LINES = 2
    private const val COLS = 16
    private val maskDlow = 0x0F
    private val maskDhigh = 0xF0
    private val maskRS = 0x40
    private val maskE = 0x20
    private val maskClk = 0x10
    private val parallelmask = 0x1F
    private class pos(val line: Int, val column: Int)

    // Escreve um byte de comando/dados no LCD em paralelo
    private fun writeByteParallel(rs: Boolean, data: Int) {
        if (rs) {
            HAL.setBits(maskRS)
        } else {
            HAL.clrBits(maskRS)
        }
        Thread.sleep(1)
        HAL.clrBits(maskClk)
        Thread.sleep(1)
        //write high
        var byte = data shr 4
        HAL.writeBits(0x0F, byte)

        HAL.setBits(maskClk)
        Thread.sleep(1)
        HAL.clrBits(maskClk)
        Thread.sleep(1)

        //write low
        byte = data.and(maskDlow)
        HAL.writeBits(0x0F, byte)

        HAL.setBits(maskClk)
        Thread.sleep(1)
        HAL.clrBits(maskClk)

        Thread.sleep(1)
        HAL.setBits(maskE)
        Thread.sleep(1)
        HAL.clrBits(maskE)
        Thread.sleep(1)

    }

    // Escreve um byte de comando/dados no LCD em série
    private fun writeByteSerial(rs: Boolean, data: Int) {
        TODO()
    }

    // Escreve um byte de comando/dados no LCD
    private fun writeByte(rs: Boolean, data: Int) {
        writeByteParallel(rs, data)
    }

    // Escreve um comando no LCD
     fun writeCMD(data: Int) {
        writeByte(false, data)
    }

    // Escreve um dado no LCD
    private fun writeDATA(data: Int) {
        writeByte(true, data)
    }

    // Envia a sequência de iniciação para comunicação a 4 bits.
    fun init() {
        Thread.sleep(16)
        writeCMD(0x30)
        Thread.sleep(5)
        writeCMD(0x30)
        Thread.sleep(2)

        writeCMD(0x30)
        Thread.sleep(1)
        writeCMD(0x38)
        writeCMD(0x08)
        writeCMD(0x01)
        writeCMD(0x06)
```

```
    writeCMD(0x0F)

}

// Escreve um caráter na posição corrente.
fun write(c: Char) {
    writeDATA(c.code)
}

// Escreve uma string na posição corrente.
fun write(text: String) {
    for(i in text)
        write(i)
}

// Envia comando para posicionar cursor (‘line’:0..LINES-1 , ‘column’:0..COLS-1)
fun cursor(line: Int, column: Int) {
    val l = line * 64
    val pos = (l + column).or(0x80) //cmd DDRAM
    writeCMD( pos )
}

// Envia comando para limpar o ecrã e posicionar o cursor em (0,0)
fun clear() {
    writeCMD(0x01)
    }
}

fun main() {
    HAL.init()
    LCD.init()
    KBD.init()
    /*

    LCD.write("Hello")
    LCD.cursor(1, 3)
    LCD.write("LIC")
    LCD.clear()
     */
  testKBD_LCD()







}

fun testKBD_LCD(){


    var c = 0
    while (true){

            val key = KBD.waitKey(1000)
            if (key != 0.toChar()){
                LCD.write(key)
                c++
            }
            if (c == 15){
                LCD.clear()
                c = 0
            }


    }

}
```