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*), de acordo com o diagrama representado na Figura 1. Neste caso o módulo *Control*, implementado em software, é a entidade consumidora.

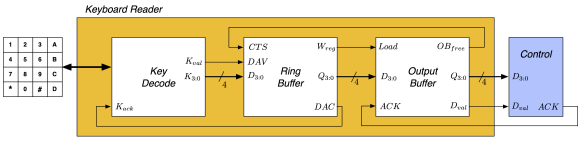
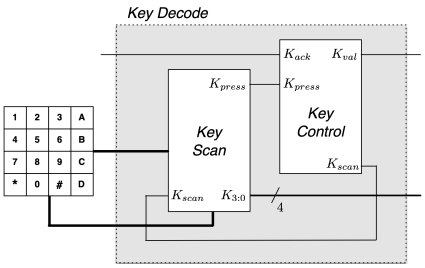


Figura 1: Diagrama de blocos do módulo *Keyboard Reader*



(a) Diagrama de blocos

(b) Diagrama temporal

Figura 2: Bloco *Key Decode*

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 *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.

O bloco *Key Scan* foi implementado de acordo com o diagrama de blocos representado 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 A.3.

Figura 3: Diagrama de blocos do módulo *Key Scan*

2 Key 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 in-

Figura 4: Máquina de estados do *Key Control*

formação, 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), 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 a estrutura do bloco *Ring Buffer*.

3 ...

4 Conclusões

A VHDL

A.1 KeyBoard Reader

A.2 KeyBoard Reader

```
component ficheiro1 is
port (
        CLK : in std_logic;
        RESET : in std_logic;
        SET : in std_logic;
        D : in std_logic;
        EN : in std_logic;
        Q : out std_logic);
end component;
```

A.3 Key Decode

```
component ficheiro2 is
port (
        CLK : in std_logic;
        RESET : in std_logic;
        SET : in std_logic;
        D : in std_logic;
        EN : in std_logic;
        Q : out std_logic);
end component;
```

A.4 Key Scan

A.5 ...

B *Kotlin*

B.1 HAL

Algoritmo 1: HAL - Hardware Abstract Layer

```
object HAL {
// Inicia o objeto
fun init()
{
}

// Retorna 'true' se o bit definido pela mask esta com o valor logico '1' no UsbPort
fun isBit(mask: Int): Boolean
{
}

// Retorna os valores dos bits representados por mask presentes no UsbPort
fun readBits(mask: Int): Int
{
}


// Escreve nos bits representados por mask os valores dos bits correspondentes em value
fun writeBits(mask: Int, value: Int)
{
}


// Coloca os bits representados por mask no valor logico '1'
fun setBits(mask: Int)
{
}


// Coloca os bits representados por mask no valor logico '0'
fun clrBits(mask: Int)
{
}

}
```

B.2 KBD

Algoritmo 2: KBD

```
// Ler teclas. Funcoes retornam '0'..'9','A'..'D','#','*' ou NONE.
object KBD {
const val NONE = 0;

// Inicia a classe
fun init()
{
}


// Retorna de imediato a tecla premida ou NONE se nao ha tecla premida.
fun getKey(): Char
{
}

// Retorna a tecla premida, caso ocorra antes do 'timeout' (em milissegundos),
// ou NONE caso contrario.
fun waitKey(timeout: Long): Char
{
}

}
```