O módulo *Keyboard Reader* implementado é constituído por dois blocos principais: *i*) o descodificador de teclado (*Key Decode*); e *ii*) o bloco de armazenamento (designado por *Key Buffer*) e iii) o bloco de transmissão e de entrega ao consumidor (*Key Transmitter)*, conforme ilustrado na Figura 1. Neste caso o módulo de controlo, implementado em *software*, é a entidade consumidora.

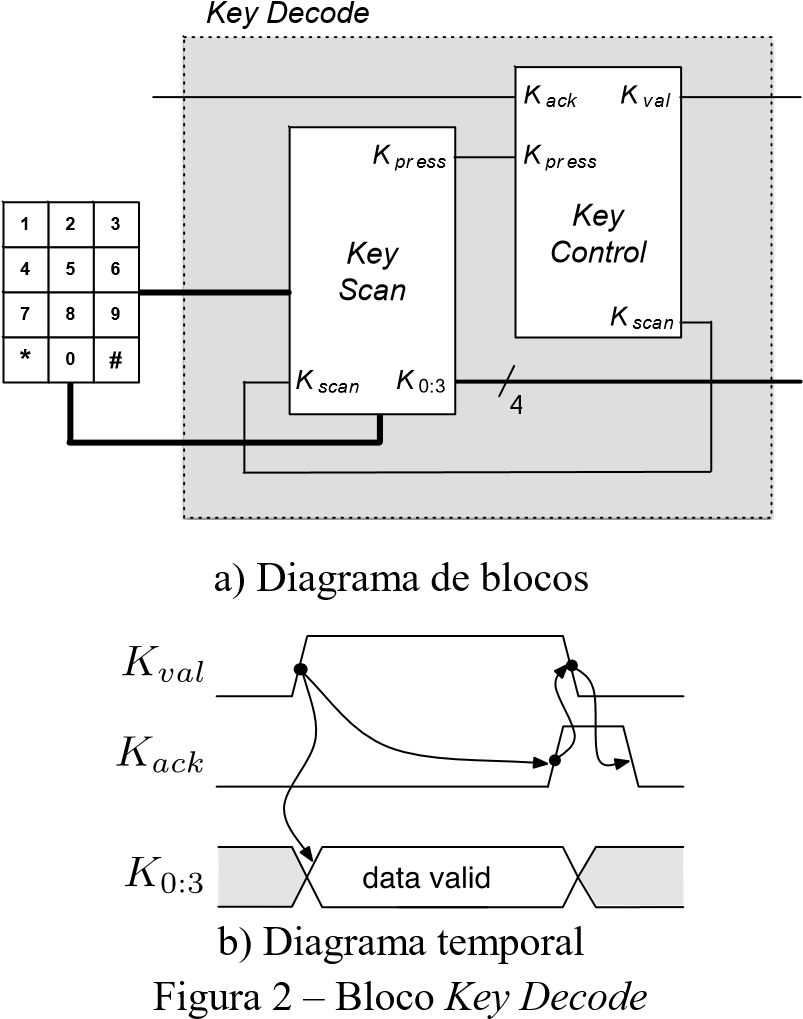
|  |  |
| --- | --- |
|  |  |

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.



O bloco *Key Scan* foi implementado de acordo com o diagrama de blocos representado na Figura 3. Optou-se pela versão 1 de implementação do bloco key scan, devido a ser de mais fácil compreensão e de implementação, na fase de projeto em que encontra.

O bloco *Key Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 4. No estado inicial, tendo o key decode “vazio”, o mesmo indica ao key scan pode receber uma tecla. Sendo esta tecla validada, se houver uma tecla premida (Kpress), O sistema só progride se for indicado que, o valor da tecla foi lido (Kack), só podendo assim, prosseguir para o início do control, não tendo quer Kack, nem tecla premida(Kpress), para ser possível haver um ajustamento de clocks.

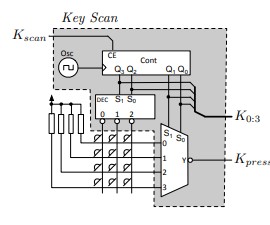
A descrição hardware do bloco *Key Decode* em CUPL encontra-se no Anexo A.

Figura 3 - Diagrama de blocosdo bloco *Key Scan*

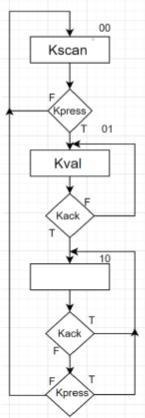


Figura 4 – Máquina de estados do bloco *Key Control*

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. Após visualizar as datasheets de cada elemento, que compõe esta parte do trabalho, chegou-se a conclusão que a corrente aceitável, para todos os blocos, sem haver um excesso de corrente, em algum destes elementos, foi escolhida para a corrente 16mA, e como a tensão será fornecida é de 5V, aplicando a lei de Ohm, chegou-se ao valor 0.313kΩ. O valor da frequência de relógio foi limitada aos clocks que estão disponíveis na ATB. Escolheu-se 1KHz, devido a necessidade do key decode ser mais rápido que o key buffer, mas que não fosse um clock tão alto que ocorria o fenómeno de Bounce.

# Key Buffer

O módulo *Key* *Buffer* implementa uma estrutura de armazenamento de dados, com capacidade de uma palavra de quatro bits. A escrita de dados no *Key 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 *Key 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 *Key Buffer* só desativa *DAC* depois de *DAV* ter sido desativado.

A implementação do *key Buffer* deverá ser baseada numa máquina de controlo (*Key Buffer Control*) e num registo externo (*Output Register*), conforme o diagrama de blocos apresentado na Figura 5.

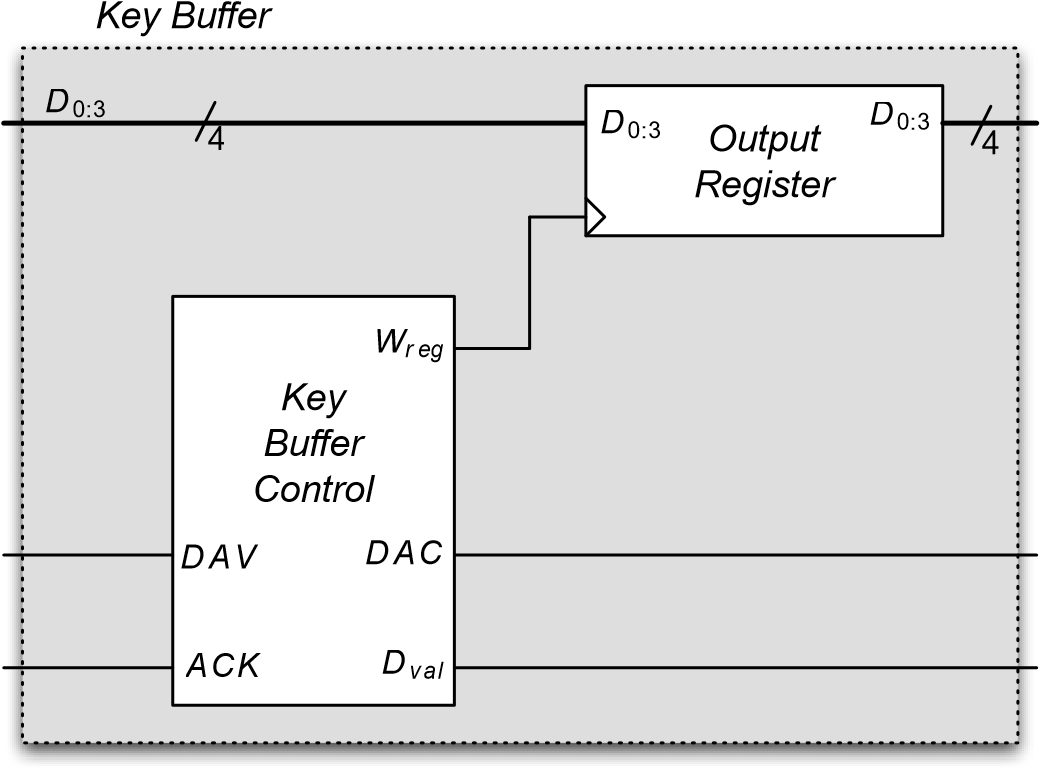


Figura 5 – Diagrama de blocos do *Key Buffer*

O bloco *Key Buffer Control* do *Key Buffer* é também responsável pela interação com o sistema consumidor, neste caso o módulo *Control*. O *Control* quando pretende ler dados do *Key Buffer*, aguarda que o sinal *Dval* fique ativo, recolhe os dados e ativa o sinal *ACK* indicando que estes já foram consumidos.

O *Key Buffer Control*, logo que o sinal *ACK* fique ativo, deve invalidar os dados baixando o sinal *Dval*, só deverá voltar a armazenar uma nova palavra depois do *Control* ter desativado o sinal *ACK*.

O bloco *Key Buffer Control* foi implementado de acordo com o diagrama de blocos representado na Figura 6. Para haver a inicialização do circuito, é necessário verificar se existe data valida (DAV). Sendo esta depois registada através de uma ascensão de clock (Wreg), prosseguindo com a indicação que a data foi aceite (DAC). Enquanto não houver indicação do bloco anterior, que a variável da data valida está com o valor ‘0’, mantem-se no mesmo estado, em caso contrario, indica que a data foi validada (Dval), ficando após, a aguardar indicação do bloco posterior, que a data foi aceite (ACK). E só com descensão de ACK, é que o sistema regressa ao seu estado inicial.

A descrição hardware do bloco *Key Buffer Control* em CUPL encontra-se no Anexo C.

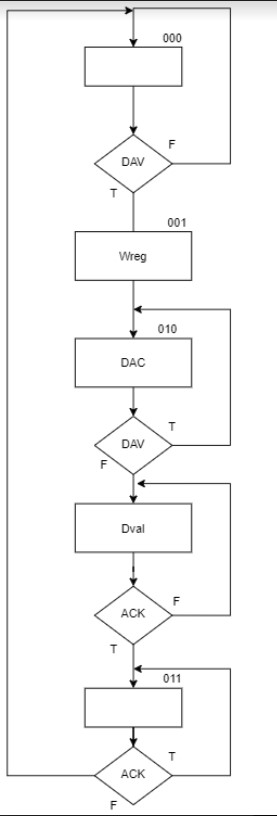


Figura 6 - Máquina de estados do bloco *Key Buffer*

## *Control*

Com base nas descrições do bloco *Key Decode* e do bloco *Key Buffer Control* implementou-se o módulo *Keyboard Buffer e Decoder* de acordo com o esquema elétrico representado no Anexo C. Para as frequências de relógio do key buffer, foi tomado em atenção, que as mesmas necessitavam de ser menores, do que as do *key decode*. Tal como o *key decode*, estava-se limitado as frequências da *ATB*, por isso escolheu-se 10Hz. Logo, o fenómeno de *bounce* não irá acontecer.

# *Key Trasmitter*

# Interface com o *Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem *Kotlin* e seguindo a arquitetura lógica apresentada na Figura 7.

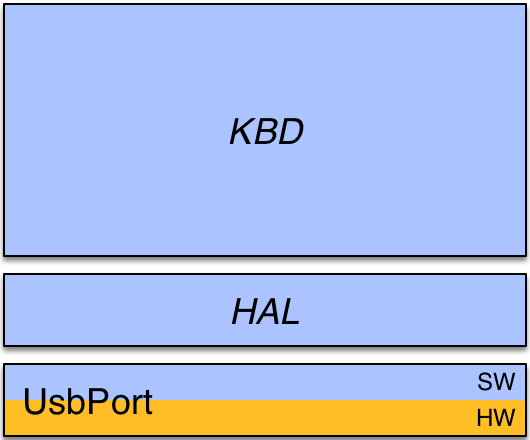


Figura 7 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

As classes *HAL* e *KBD* desenvolvidas são descritas nas secções 3.1. e 3.2, e o código fonte desenvolvido nos Anexos E e F, respetivamente.

## Classe *HAL*

A *HAL* (*Hardware Abstraction Layer*) é composta por 5 funções; a função inicializadora (*init*), garante nos que ao ser iniciado o código as saídas do *UsbPort* são neutras. A função *isBit*, através da variável *mask*, transmite-nos um valor booleano dependente do valor do *bit*, quando este valor é ‘1’, é nos retornado o valor ‘*true’*.

A função *readBits*, tendo igualmente os mesmos parâmetros de entrada, retorna o valor da máscara sem mexer nos outros valores presentes a entrada. Temos depois presente, duas funções uma que coloca os bits pertencentes a máscara a neutro (*clrBits*), e outra que os coloca com o valor positivo (*setBits*). Como último elemento da classe, temos o *writeBits* que nos possibilita substituir o valor da máscara por um pretendido, tendo sido assim, abordado como uma junção da função *setBits* e *clrBits*, sendo depois o valor pretendido colocado no lugar da máscara, através duma estância superior.

## Classe *KBD*

*KBD* (*KeyBoard Decoder*) a função inicializador desta classe, serve para nos colocar o valor de *ACK* neutro. A função para obter o valor da *key*, é obtida através duma primeira verificação com ajuda da classe *HAL*, para verificar o valor lógico de *Dval*, se o mesmo for positivo é feita uma descodificação da máscara recebida para obtenção do carácter pretendido; tratando também da variável *ACK*, para a correspondência do *ASM Chart* do *control* do *key buffer*, estando encarregue de colocar a variável *ACK* positiva e depois neutra. É criada uma função que permite a temporização do recebimento duma tecla através de um valor dado, sendo esta a função *waitKey*.

## Classe *Key Receiver*

# Conclusões

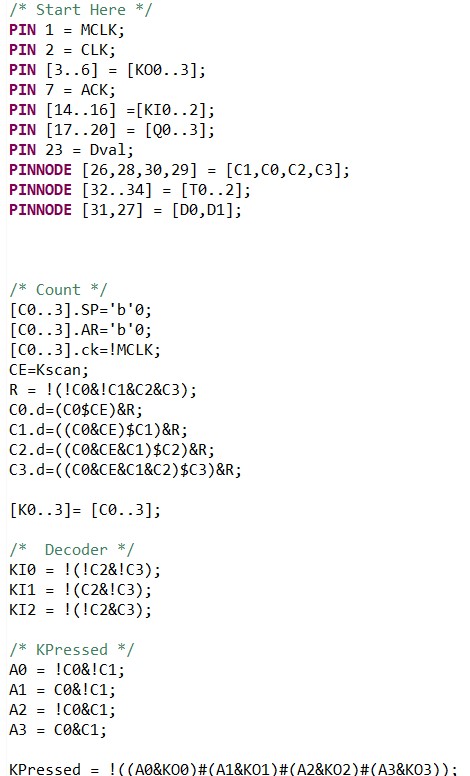
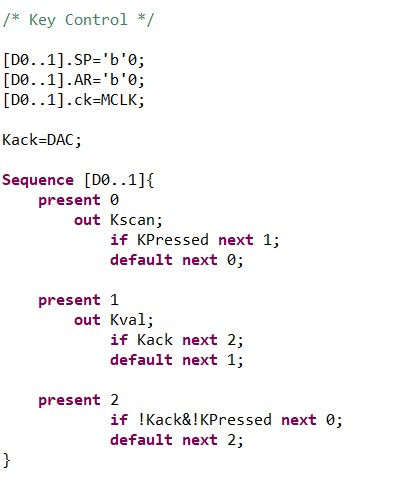
O módulo *KeyBoard Reader* é desenvolvido a partir de duas componentes hardware e software. Para realização desde módulos, temos como materiais para alem da *breadboard* e os fios, é necessário do teclado matricial 4x3, foi aplicada numa só *PAL* o *Keyboard Reader*, a utilização de um *UsbPort* que nos permite ter uma ligação, da componente hardware com a de software, uma *ATB* para se fornecer tensão no circuito, proveniente do pc, e controlar os *clock*, e um pc com o código correspondente a componente de software.

Este modulo, permite nos assim então, que tenha em registo os valores indicados no *keyboard* e os mesmos serem validados e descodificados, pelo código de software obtido; sendo todo este circuito realizado com algumas, instâncias para benefício do circuito, sendo estas por exemplo, a diferenciação de *clocks*, o caso das resistências e no caso de *software* a utilização de um *while* para espera de mudança de variável.

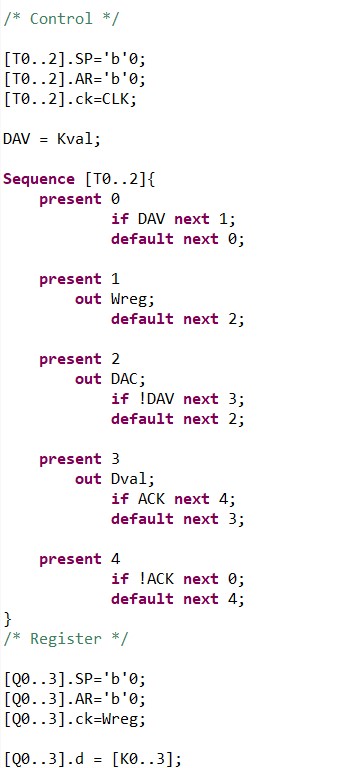
Para calcular, a latência de verificação da tecla premida até a mesma ser validada, foi contabilizado 17 *clocks* no *MCLK* e consequentemente, 51 para o *CLK*. Para se verificar estes dados, considerou-se o pior caso ser a escolha da última tecla de uma coluna, começando a contabilizar os *clocks* no momento, em que o mesmo avança para a coluna seguinte, sendo a tecla desejada a última desde o início da contagem.

Com *o key transmitter*, a entidade consumidora consegue ter a noção da existência de dados para a recolha e a mesma vai recebendo *bit* a *bit* a informação e indicando sobre o módulo que está pronto para a receção de outro *bit*. Todo este módulo em conjunto permite, alem de criar uma memória de breves teclas, permite ao mesmo corre em sincronia e dependendo de si mesmo, não prejudicando nem criando conflito no mesmo.

# Descrição *CUPL* do bloco *Key Decode*



# Descrição *CUPL* do bloco *Key Buffer*



# Esquema elétrico do módulo *Keyboard Decoder* e *Keyboard Buffer*

# Esquema elétrico do módulo *Keyboard Trasmitter*

# Código *Kotlin* da classe *HAL*

**object HAL {**

**var out = 0**

**fun init() {**

**UsbPort.out(out.inv())**

**} fun isBit(mask: Int): Boolean ...**

**return readBits(mask)>0**

**} fun readBits(mask: Int): Int{ val x = UsbPort.`in`().inv() return x.and(mask)**

**} fun writeBits(mask: Int, value: Int){ out = mask.inv().and(out) out = value.or(out)**

**UsbPort.out(out.inv())**

**}**

**fun setBits(mask: Int){ out = mask.or(out)**

**UsbPor.out(out.inv())**

**} fun clrBits(mask: Int){ out = mask.inv()and(out)**

**UsbPor.out(out.inv())**

**}**

**}**

# Código *Kotlin* da classe *KBD*

**object KBD {**

**const val NONE = 0;**

**fun init() {**

**HAL.clr(0x80)**

**}**

**fun getKey(): Char{ var x = NONE.toChar() if(HAL.isBit(0x80){**

**x = when(HAL.readBits(0x0F){**

**0x00 ->’1’**

**0x01 ->’4’**

**0x02 ->’7’**

**0x03 ->’\*’**

**0x04 ->’2’**

**0x05 ->’5’**

**0x06 ->’8’**

**0x07 ->’0’**

**0x08 ->’3’**

**0x09 ->’6’**

**0x0a ->’9’ 0x0b ->’#’**

**else-> NONE.toChar()**

**}**

**HAL.setBits(0x80) while(HAL.isBit(0x80)){} HAL.clrBits(0x01) return x**

**} return x**

**}**

**fun waitKey(timeout: Long): Char{ val temp = Time.getTimeInMillis() + timeout do{**

**val x = getKey()**

**if(x != NONE.toChar())**

**return x**

**} while(Time.getTimeInMillis()<= temp) return NONE.toChar()**

**}**

**}**

# Código *Kotlin* da classe *Key Receiver*