Autores: João Magalhães (A48348@alunos.isel.pt)
Ricardo Henriques (A48322@alunos.isel.pt)

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.

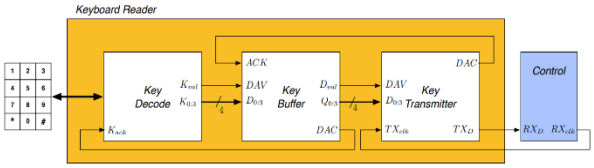
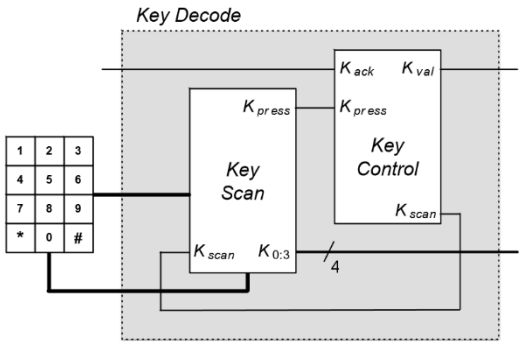


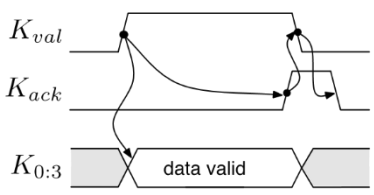
Figura 1 – Diagrama de blocos do módulo *Keyboard Reader*

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.



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. 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.

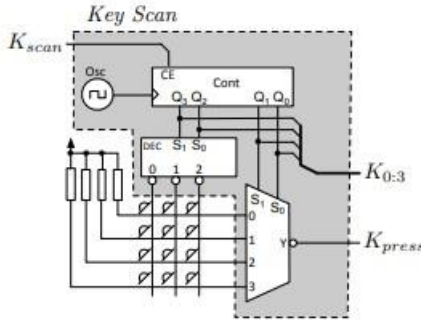


Figura 3 - Diagrama de blocos do bloco *Key Scan*

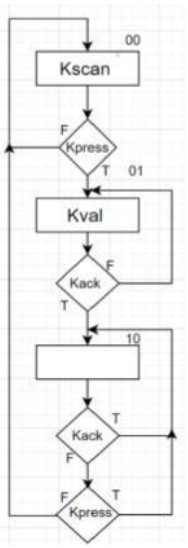


Figura 4 – Máquina de estados do bloco *Key Control*

Autores: João Magalhães (A48348@alunos.isel.pt)
Ricardo Henriques (A48322@alunos.isel.pt)

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.

2. 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 $D_{0:3}$ 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 *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.

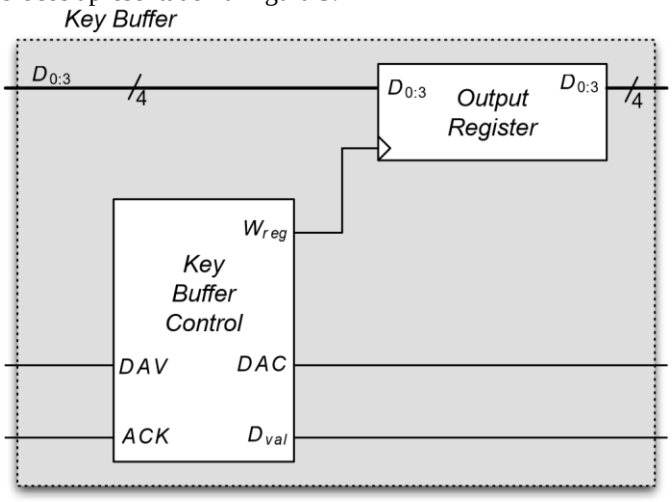


Figura 5 – Diagrama de blocos do *Key Buffer*

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 D_{val} 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 D_{val}, 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.

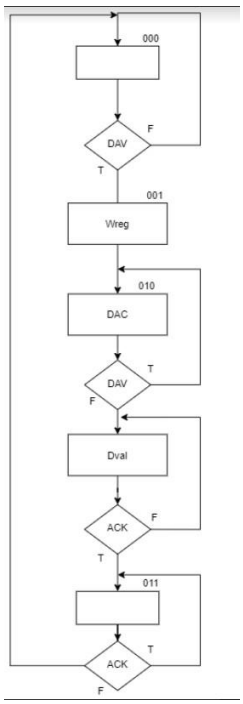


Figura 6 - Máquina de estados do bloco *Key Buffer Control*

Com base nas descrições do bloco *Key Decode* e do bloco *Key Buffer Control* implementou-se o módulo

Autores: João Magalhães (A48348@alunos.isel.pt)
Ricardo Henriques (A48322@alunos.isel.pt)

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.

3. Key Trasmitter

O bloco do *Key Transmitter* corresponde a uma estrutura de transmissão em série, tendo o mesmo a capacidade de armazenar uma palavra de 4 *bits*, também é este bloco que é o responsável pela interação com o sistema consumidor.

Ao registar a palavra de 4 *bits*, avisa o *Control* que o mesmo tem data que pode ser enviada e conforme a entidade consumidora gerir a mesma ir tratar dos clocks e assim receber a palavara *bit* a *bit*.

Contudo a receção da mesma provem da sinalização previa do *bit* '0' e logo após o *bit* '1', sabendo então a entidade consumidora que os 4 *bits* seguintes serem os bits que formam a palavra, proveniente do *bit* do menor peso ao maior, dando então depois o valor de *bit* '0' para indicação do término da mesma. Com isto é possível criar uma máquina de estados possível para o envio dos bits, sendo o *TXclk* o *counter* da máquina, como mostrado na figura 7 e na figura 8 é possível verificar o *ASM-chart* do bloco no seu todo fornecendo num passo inicial o registro da palavra e depois a ativação da máquina para a entidade consumidora começar a receber a palavra.

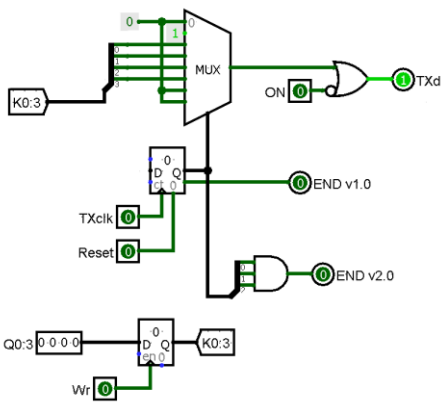


Figura 7- Esquema em *logisim* do bloco Key Transmitter

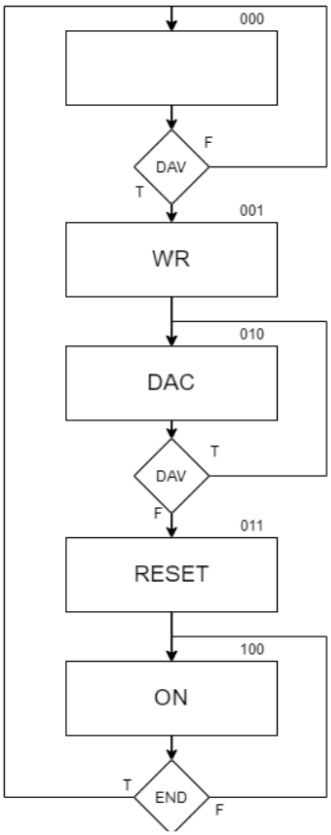


Figura 8- *ASM chart* do bloco K*ey Transmitter*

4. Interface com o *Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem *Kotlin* e seguindo a arquitetura lógica apresentada na Figura 9.

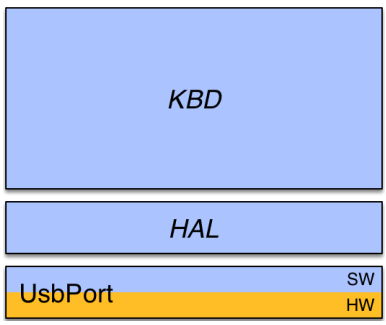


Figura 9 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

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

Autores: João Magalhães (A48348@alunos.isel.pt)
Ricardo Henriques (A48322@alunos.isel.pt)

4.1. 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.

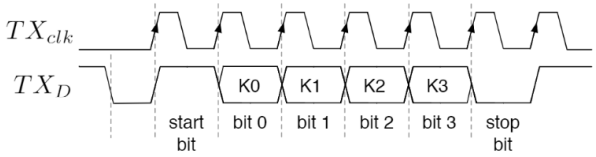
4.2. Classe *KBD*

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*.

4.3 Classe *Key Receiver*

Na classe *Key Receiver*, foi implementado um transmissor em serie, onde os *bits* que constituem são enviados em serie em vez de paralelo. Para esta implementação, foi aplicado uma função, que cumpre o protocolo proposto, na figura 10.

O *TXclk* é controlado pela própria função, para ir juntando os *bits* da palavra sequencialmente, usou se uma soma sequencial de potência de 2, onde o expoente vai diminuindo ao fim de cada *bit* enviado. Para utilizar esta função vai-se a constante *SERIAL_INTERFACE* e coloca-se o seu valor como *true*. A descodificação do conteúdo da tecla é feita pela função *getKeySerial* que procura no *array* do teclado, a tecla respondente ao conjunto dos bits.



5. 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.

Autores: João Magalhães (A48348@alunos.isel.pt)
Ricardo Henriques (A48322@alunos.isel.pt)

A. Descrição *CUPL* do bloco *Key Decode*

```
/* Start Here */
PIN 1 = MCLK;
PIN 2 = CLK;
PIN [3..6] = [KO0..3];
PIN 7 = ACK;
PIN [14..16] =[KI0..2];
PIN [17..20] = [Q0..3];
PIN 23 = Dval;
PINNODE [26,28,30,29] = [C1,C0,C2,C3];
PINNODE [32..34] = [T0..2];
PINNODE [31,27] = [D0,D1];



/* Count */
[C0..3].SP='b'0;
[C0..3].AR='b'0;
[C0..3].ck=!MCLK;
CE=Kscan;
R = !(!C0&!C1&C2&C3);
C0.d=(C0$CE)&R;
C1.d=((C0&CE)$C1)&R;
C2.d=((C0&CE&C1)$C2)&R;
C3.d=((C0&CE&C1&C2)$C3)&R;

[K0..3]= [C0..3];

/*  Decoder */
KI0 = !(!C2&!C3);
KI1 = !(C2&!C3);
KI2 = !(!C2&C3);

/* KPressed */
A0 = !C0&!C1;
A1 = C0&!C1;
A2 = !C0&C1;
A3 = C0&C1;

KPressed = !((A0&KO0)#(A1&KO1)#(A2&KO2)#(A3&KO3));

/* Key Control */

[D0..1].SP='b'0;
[D0..1].AR='b'0;
[D0..1].ck=MCLK;

Kack=DAC;

Sequence [D0..1]{
    present 0
        out Kscan;
            if KPressed next 1;
            default next 0;

    present 1
        out Kval;
            if Kack next 2;
            default next 1;

    present 2
            if !Kack&!KPressed next 0;
            default next 2;
}
```

Autores: João Magalhães (A48348@alunos.isel.pt)
Ricardo Henriques (A48322@alunos.isel.pt)

B. Descrição *CUPL* do bloco *Key Buffer*

```
/* Control */

[T0..2].SP='b'0;
[T0..2].AR='b'0;
[T0..2].ck=CLK;

DAV = Kval;

Sequence [T0..2]{
    present 0
            if DAV next 1;
            default next 0;

    present 1
        out Wreg;
            default next 2;

    present 2
        out DAC;
            if !DAV next 3;
            default next 2;

    present 3
        out Dval;
            if ACK next 4;
            default next 3;

    present 4
            if !ACK next 0;
            default next 4;
}
/* Register */

[Q0..3].SP='b'0;
[Q0..3].AR='b'0;
[Q0..3].ck=Wreg;

[Q0..3].d = [K0..3];
```

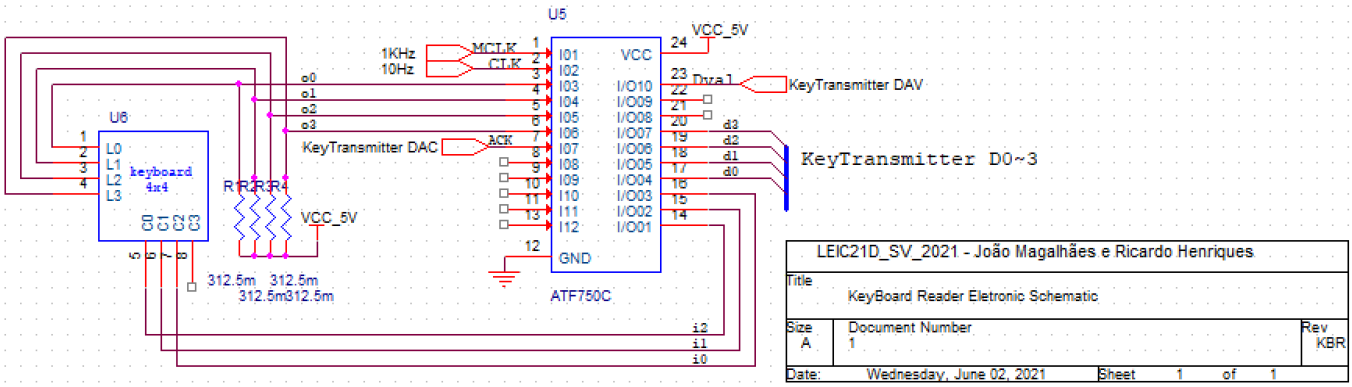
Autores: João Magalhães (A48348@alunos.isel.pt)
Ricardo Henriques (A48322@alunos.isel.pt)

C. Descrição *CUPL* do bloco *Key Transmitter*

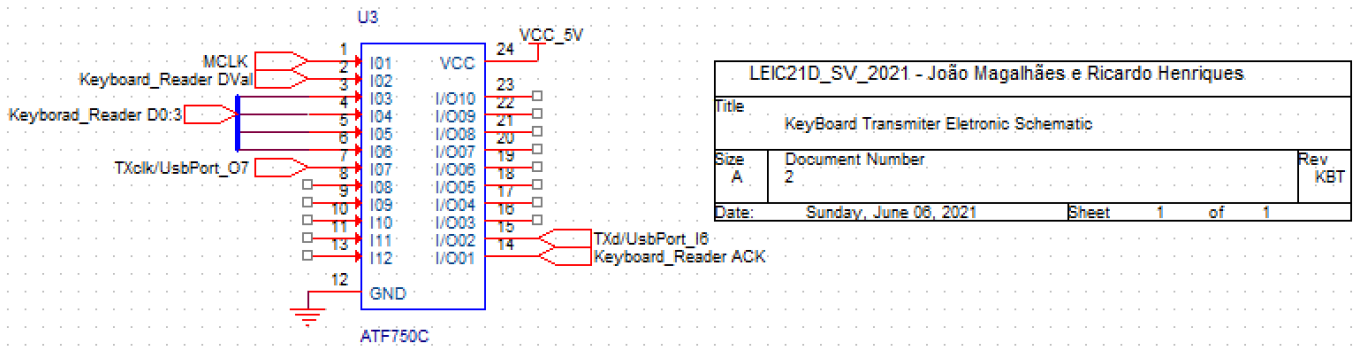
```
/* Start Here */
PIN 1 = MCLK;
PIN 2 = DAV;
PIN[3..6]=[D0..3];
PIN 7 = TXclk;
PIN 14 = DAC;
PIN 15 = TXd;
PIN [16..18]=[T0..2];
PINNODE [20..23]=[K0..3];
PINNODE [27..29]=[Q0..2];

[T0..2].SP='b'0;
[T0..2].AR='b'0;
[T0..2].ck=MCLK;

Sequence[T0..2]{


    Present 0
        if DAV next 1;
        default next 0;

    Present 1
        out Wr;
        default next 2;

    Present 2
        out DAC;
        if !DAV next 3;
        default next 2;

    Present 3
        out Reset;
        default next 4;

    Present 4
        out ON;
        if END next 0;
        default next 4;
}

/* Counter */
[Q0..2].SP='b'0;
[Q0..2].AR=Reset;
[Q0..2].ck=TXclk;

CE = T2 # T1&T0;
Q0.t=CE;
Q1.t=Q0&CE;
Q2.t=Q0&Q1&CE;
END = Q0&Q1&Q2;

/* Register */

[K0..3].SP='b'0;
[K0..3].AR='b'0;
[K0..3].ck=Wr;

[K0..3].d=[D0..3];

/* MUX */

C1= !Q2&!Q1&Q0;
C2= !Q2&Q1&!Q0;
C3= !Q2&Q1&Q0;
C4= Q2&!Q1&!Q0;
C5= Q2&!Q1&Q0;
TXd= !ON # (C1 # C2&K0 # C3&K1 # C4&K2 # C5&K3);
```

Autores: João Magalhães (A48348@alunos.isel.pt)
Ricardo Henriques (A48322@alunos.isel.pt)

D. Esquema elétrico do módulo *Keyboard Decoder* e *Keyboard Buffer*



E. Esquema elétrico do módulo *Keyboard Trasmitter*



Autores: João Magalhães (A48348@alunos.isel.pt)
Ricardo Henriques (A48322@alunos.isel.pt)

F. 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())
   }
}
```

Autores: João Magalhães (A48348@alunos.isel.pt)
Ricardo Henriques (A48322@alunos.isel.pt)

G. Código *Kotlin* da classe *KBD*

```
 import isel.leic.utils.Time

object KBD {
    const val NONE = 0
    private const val ACK_MASK = 0x80
    private const val DVAL_MASK = 0x10 //0x80 -> 0x10 is for simulation purposes
    private const val KEY_VALUE = 0x0F
    private const val SERIAL_INTERFACE = false
    private val KEYBOARD= charArrayOf('1', '4', '7','*','2','5','8','0','3','6','9','#')

    fun init() {
        HAL.clrBits(ACK_MASK)
    }
    fun getKey():Char{
        if(SERIAL_INTERFACE) return getKeySerial()
        else return getKeyParallel()
    }

    private fun getKeySerial():Char{
        val x =  KeyReceiver.rcv()
        return KEYBOARD[x]
    }

    private fun getKeyParallel():Char {
        var x:Char = NONE.toChar()
        if (HAL.isBit(DVAL_MASK)) {
            x=KEYBOARD[HAL.readBits(KEY_VALUE)]
            HAL.setBits(ACK_MASK)
            while (HAL.isBit(DVAL_MASK)){} /*Waiting for Dval to be 0*/
            HAL.clrBits(ACK_MASK)
            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()
    }
}

fun main(){
    HAL.init()
    KBD.init()
    while (true) {
        print(KBD.waitKey(50))
        Time.sleep(50)
    }
}
```

Autores: João Magalhães (A48348@alunos.isel.pt)
Ricardo Henriques (A48322@alunos.isel.pt)

H. Código *Kotlin* da classe *Key Receiver*

```
import isel.leic.utils.Time
import kotlin.math.pow

object KeyReceiver {

    private const val TX_CLK = 0x40
    private const val TXD = 0x40
    private const val NUMB_ITERATION = 6
    private val KEY_ITERATION = (1..4)

    /**
    TXclk -> Output 6
    TXd-> Input 6
     **/

    fun init(){
        HAL.clrBits(TX_CLK)
    }

    fun rcv():Int {
        var count = 0
        var s = -1.0
        if(!HAL.isBit(TXD)) {
            s=0.0
            while (count <= NUMB_ITERATION) {
                HAL.setBits(TX_CLK)
                Time.sleep(5)
                HAL.clrBits(TX_CLK)
                Time.sleep(5)
                val x = HAL.readBits(TXD)
                if (count in KEY_ITERATION) {
                    if (x > 0) s += ((2.0).pow(count -1))   /* Recreation on just one number
of the key value */
                }
                count++
            }
        }
        return s.toInt()               /* If s = -1 the higher code will understand like
incorrect value */
    }
}

fun main(){
    KeyReceiver.init()
    while (true){
        println(KeyReceiver.rcv())
        Time.sleep(250)
    }
}
```