ISEL
ADEETC
Área Departamental de Engenharia Electrónica e de Telecomunicações e de Computadores

Licenciatura em Engenharia Informática e de Computadores

Jogo da Roleta

Keyboard Reader
(Roulette Game)

A46378 Berto Barata
A44787 Gonçalo Garcia
A44776 João Gomes

Professores:
Pedro Miguens Matutino (pedro.miguens@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

Projeto de
Laboratório de Informática e Computadores
2020 / 2021 inverno

22 de outubro de 2020

Índice

Introdução

O projeto semestral desta unidade curricular consiste no desenvolvimento de um jogo da roleta. A roleta compreende números entre 0 e 9 e um jogador faz apostas premindo um teclado de acordo com os números em que quer apostar. O jogador possui créditos, obtidos através da introdução de moedas de um euro num moedeiro. Cada moeda corresponde a dois créditos. Por cada aposta é retirado um crédito ao saldo do jogador, sendo possível apostar mais do que um crédito no mesmo número.

O sistema que implementa o jogo será constituído por um computador (módulo de controlo), um teclado de doze teclas, um moedeiro, um mostrador LCD de duas linhas com dezasseis caracteres cada, um mostrador da roleta e uma chave de manutenção (para colocar o sistema em modo de manutenção).
Nesta primeira fase do projeto, pretende-se o desenvolvimento do módulo responsável pela descodificação do teclado, designado por Keyboard Reader, que determina qual a tecla pressionada e disponibiliza o seu código, e da parte do módulo de controlo, designado por Control, que interage com o Keyboard Reader.

O módulo Keyboard Reader é constituído por dois blocos principais: o descodificador do teclado, designado por Key Decode, e o bloco de armazenamento e entrega do código da tecla premida, designado por Key Buffer. A entidade consumidora, à qual o código da tecla em questão é entregue, é o módulo de controlo, designado por Control, implementado em software.), conforme ilustrado na Figura 1. Neste caso o módulo de controlo, implementado em software, é a entidade consumidora.

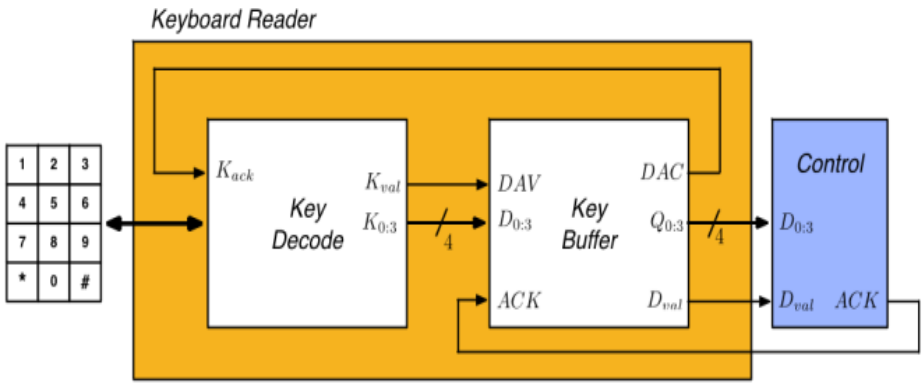
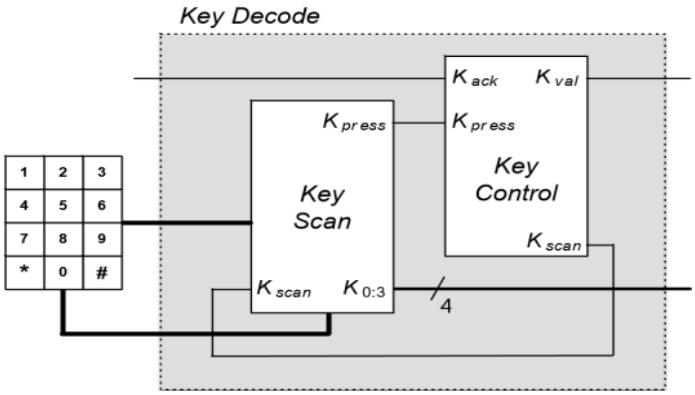


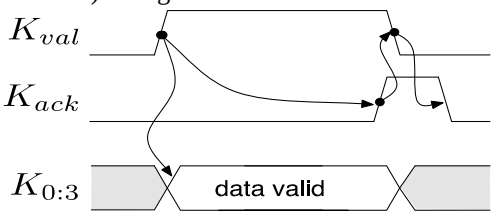
Figura 1 – Diagrama de blocos do módulo Keyboard Reader

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

Como ponto de partida, analisaram-se as três versões do sub-bloco Key Scan apresentadas no enunciado do trabalho, comparando-as de modo a concluir qual delas implementar.

Em termos de hardware, todas contam com um oscilador e com um descodificador 2 x 3 com as saídas negadas. No entanto, apresentam algumas diferenças:

- A versão I é produzida recorrendo a um contador crescente e síncrono de quatro bits e um multiplexador 4 x 1 com a saída negada.
- A versão II é produzida recorrendo a dois contadores crescentes e síncronos de dois bits cada, um multiplexador 4 x 1 com a saída negada e uma porta NAND.
- A versão III é produzida recorrendo a um contador crescente e síncrono de dois bits, um codificador prioritário 4 x 2 com as entradas negadas e um registo de dois bits.

Como estamos a trabalhar com lógica programável, sendo todo os módulos funcionais implementados através de uma PAL ATF750C, nenhuma das versões é mais vantajosa em termos de hardware:

Comparando a primeira com a segunda, ao usarmos dois contadores em vez de um recorremos a menos portas lógicas, no entanto esta diferença não é relevante. As portas extra que usamos já lá se encontram ao nosso dispor e não vão ser necessárias para mais nada.

Na terceira versão, os dois flip-flops que poupamos ao usar um só contador são necessários para construir o registo que recebe as duas saídas do codificador e, apesar deste ser logicamente mais simples que o multiplexador, a diferença também não é relevante. Em termos de tempo necessário para rastrear o teclado, a três versões já apresentam diferenças significativas:

Na versão I, o rastreamento consiste em passar por cada uma das três colunas do teclado, verificando em cada uma delas as quatro linhas, individualmente. No melhor caso, não é necessário nenhum clock para determinar a tecla a ser premida: é a que está a ser verificada, não sendo necessárias mais verificações. No pior caso, são necessários onze clocks: a tecla premida é a última a ser verificada, ou seja, cada uma das doze teclas tem de ser verificada individualmente.

Na versão II, o rastreamento consiste em passar por cada uma das três colunas do teclado ao mesmo tempo que se passa por cada uma das quatro linhas, o que é possível devido ao uso de dois contadores. Por outras palavras, a tecla verificada encontra-se sempre numa coluna e numa linha diferentes da que foi verificada anteriormente, o que não acontece na versão I. A porta NAND utilizada permite detetar se na coluna a ser verificada se encontra uma tecla premida, situação em que o contador que interage com o descodificador é parado, passando a ser rastreadas apenas as linhas do teclado. No melhor caso, não é necessário nenhum clock para determinar a tecla a ser premida: é a que está a ser verificada, não sendo 2/8 necessárias mais verificações. No pior caso, são necessários cinco clocks: a tecla premida encontra-se na última coluna a ser verificada, assim como na última linha a ser verificada dessa mesma coluna.

Na versão III, ao recorrer a um codificador prioritário é possível determinar de imediato a linha da tecla a ser premida, sendo apenas necessário percorrer cada uma das colunas do teclado. No entanto, uma ação extra é necessária: a escrita do código de dois bits que identifica a linha no registo que recebe as saídas do codificador. Assim sendo, no melhor caso é necessário um clock para determinar a tecla a ser premida: encontra-se na coluna que está a ser verificada, não sendo necessárias mais verificações, apenas escrever no registo de dois bits. No pior caso, são necessários três clocks: a tecla premida encontra-se na última coluna a ser verificada, sendo ainda necessário escrever no registo. Optou-se por implementar a terceira versão do sub-bloco Key Scan: em termos de hardware não há nenhuma vantagem, no entanto, recorrendo a esta arquitetura o tempo de rastreamento do teclado no pior caso é menor. Descreveram-se em CUPL o descodificador 2 x 3 com as saídas negadas, o contador crescente e síncrono de dois bits, o codificador prioritário 4 x 2 com as entradas negadas e o registo de dois bits (através flip-flops do tipo D). O contador conta apenas até dois (e não até três, a capacidade máxima de um contador de dois bits), uma vez que o teclado tem somente três linhas (zero, um e dois).

O codificador dá prioridade aos bits de maior peso, não que haja qualquer vantagem sobre dar prioridade aos de menor peso, apenas se implementou assim. O sinal GS (Group Select) produzido pelo codificador, que está ativo quando pelo menos uma das suas entradas está ativa, corresponde a um sinal designado por Kpress, que informa que uma tecla foi premida. O barramento K0:3 toma como bits de menor peso as saídas do registo de dois bits e como bits de maior peso as saídas do contador. O bloco *Key Scan* foi implementado de acordo com o diagrama de blocos representado na Figura 3.

O sub-bloco Key Control, que controla o varrimento do teclado e o fluxo de saída do bloco Key Decode, é uma máquina de estados. Quando uma tecla é pressionada, ativa um sinal designado por Kval, sendo disponibilizado o código dessa mesma tecla no barramento K0:3. Só é iniciado um novo ciclo de varrimento quando um sinal designado por Kack for ativado e a tecla premida libertada. O varrimento é efetuado em função de um sinal designado por Kscan, que, de modo a implementar a versão III do sub-bloco Key Scan, se divide em dois, Kscan0 e Kscan1. Kscan0 é responsável pelo contador, permitindo-o ou não contar, e Kscan1 é o clock do registo de dois bits. O bloco *Key Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 4.

A descrição hardware do bloco Key Decode em CUPL/VHDL encontra-se no Anexo 00.

Esta máquina de estados foi descrita em CUPL. Uma observação importante a fazer é que há um estado dedicado unicamente à escrita de dados no registo: só depois de serem escritos é que se pode sinalizar com certeza que foram registados, não se podendo lançar esse sinal ao mesmo tempo que se faz a escrita.

Com base nas descrições do bloco Key Decode implementou-se parcialmente o módulo Keyboard Reader de acordo com o esquema elétrico representado no Anexo 0.

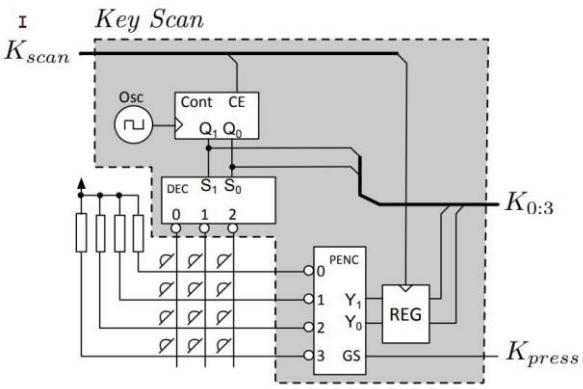


Figura 3 - Diagrama de blocos do bloco *Key Scan*

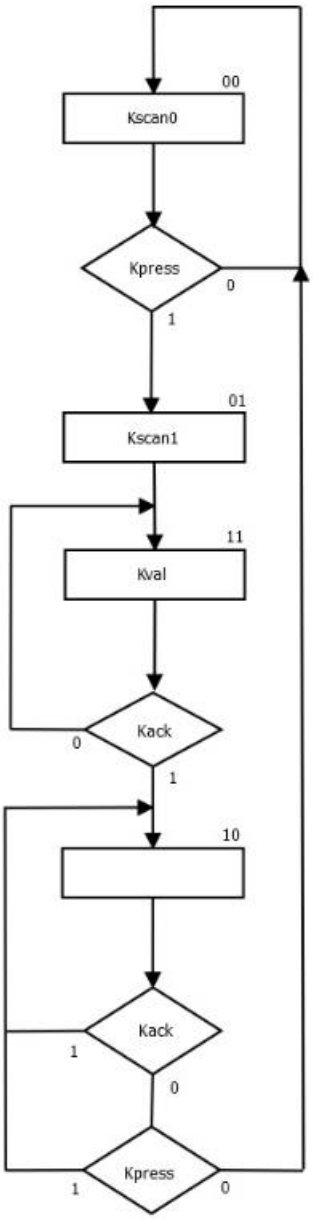


Figura 4 – Máquina de estados do bloco *Key Control*

2 Key Buffer

O bloco Key Buffer consiste numa estrutura de armazenamento de dados com capacidade para quatro bits. A escrita neste bloco inicia-se com a ativação de um sinal designado por DAV (Data Available) por parte do sistema produtor, o bloco Key Decode, indicando que tem dados, D0:3, para serem armazenados. Registados os dados em memória, um sinal designado por DAC (Data Accepted) é ativado pelo bloco Key Buffer, informando o sistema produtor que a informação foi aceite. O sistema produtor mantém o sinal DAV ativo até que o sinal DAC seja ativado e o bloco Key Buffer só desativa o sinal DAC após o sinal DAV ser desativado. O bloco Key Buffer é constituído por um sub-bloco designado por Key Buffer Control, que é uma máquina de estados, e por um registo de quatro bits, designado por Output Register. O sub-bloco Key Buffer Control interage também com o sistema consumidor, o módulo Control, que aguarda que um sinal designado por Dval fique ativo para ler os dados do bloco Key Buffer, recolhendo-os e ativando um sinal designado por ACK que indica que foram consumidos. Logo que este sinal fique ativo, o sub-bloco Key Buffer Control invalida os dados armazenados no registo de saída baixando o sinal Dval. Para que novos dados possam ser armazenados é necessário que o módulo de controlo desative o sinal ACK. 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.

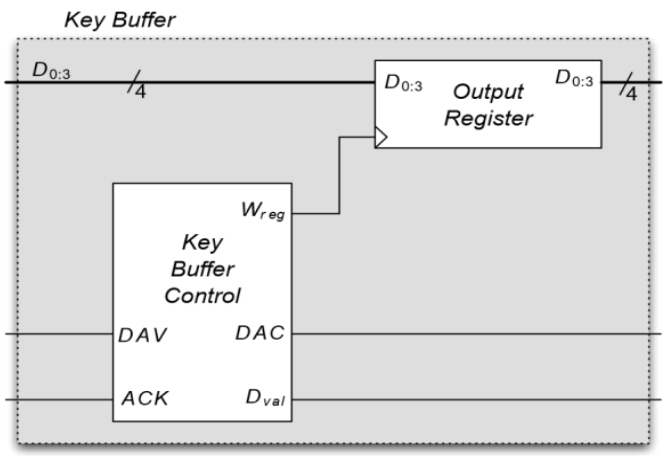


Figura 5 – Diagrama de blocos do *Key Buffer*

Esta máquina de estados, assim como o registo de quatro bits (através de flip-flops do tipo D), foram descritos em CUPL. Tal como na primeira máquina, há um estado dedicado unicamente à escrita de dados no registo. Foi aplicada codificação Gray aos estados de ambas as máquinas, de modo a evitar possíveis (mas pouco prováveis) glitches causados por jitter. O bloco *Key Buffer Control* foi implementado de acordo com o diagrama de blocos representado na Figura 6.

A descrição hardware do bloco *Key Buffer Control* em CUPL/VHDL encontra-se no Anexo A.1.

Com todos os blocos e sub-blocos do módulo Keyboard Reader desenvolvidos, juntaram-se todas as descrições de hardware num só ficheiro para implementar numa PAL. O sinal recebido no pin 1 da PAL, designado por Clk, é utilizado como clock para o contador e para a máquina de estados Key Buffer Control. O clock da máquina Key Control encontra-se em oposição de fase com Clk (!Clk). 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 A.1. .

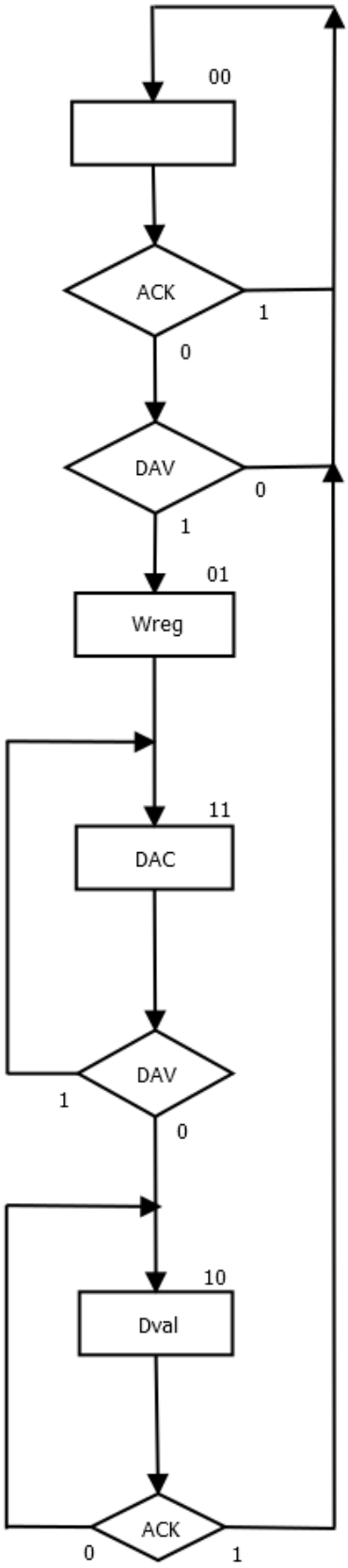


Figura 6 - Máquina de estados do bloco *Key Buffer Control*

3 Interface com o Control

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem Java e seguindo a arquitetura lógica apresentada na Figura 7. Nesta primeira fase do trabalho, realizaram-se apenas duas das classes apresentadas no enunciado: HAL (Hardware Abstraction Layer) e KBD (Keyboard).

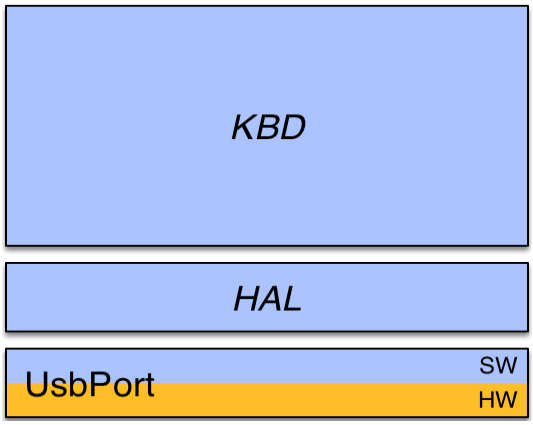


Figura 7 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

As classes *HAL* e *KBD* desenvolvidas são descritas nas secções 3.1. e 3.2, e o código fonte desenvolvido nos Anexos A.3. e A.4. , respetivamente.

3.1 Classe HAL

A classe HAL virtualiza o acesso ao sistema UsbPort, que possibilita a interação com o hardware. Nesta classe utiliza-se o UsbPort, para possibilitar a utilização do InputPort (de receção) e OutputPort (de envio). Ambos estes portos têm 8 bits e são negados, ou seja, quando se recebe qualquer dado/valor pelo InputPort temos de os negar para que a veracidade deles esteja correta assim como quando o enviamos para o OutputPort temos que o negar para que este também saia corretamente.
A partir da utilização da biblioteca UsbPort facultada pelo professor, conseguiu-se o uso das funções UsbPort.in e UsbPort.out que permitem receber valores do porto de entrada e enviar valores pelo porto de saída, respetivamente.

3.2 Classe KBD

A classe KBD é responsável pela leitura do teclado, onde é atribuído um valor binário a cade tecla, fazendo assim corresponder cada uma destas ao array de caracteres. Este método recorre também aos métodos da classe HAL.

4 Conclusões

Os objetivos solicitados foram alcançados. Percorrendo todos os passos enumerados neste relatório, desenvolvemos os conhecimentos previamente adquiridos sobre lógica e sistemas digitais, descrevendo hardware e produzindo máquinas de estados e vários módulos funcionais (contadores, descodificadores, codificadores prioritários...).
Aprendemos ainda a elaborar esquemas elétricos e melhorámos as nossas capacidades de programação em Java.
Algumas etapas foram mais trabalhosas que outras, nomeadamente a análise e compreensão das diferentes versões do sub-bloco Key Scan e o desenvolvimento das duas máquinas de estados, que, tal como muitas outras partes deste trabalho, passaram por várias versões e processos de simplificação.

A.1. Descrição *CUPL* do bloco *Key Decode* e *Key Buffer*

```
Name        KeyboardReader;
PartNo      ;
Date        04/11/20;
Revision    ;
Designer    GG;
Company     ;
Assembly    ;
Location    ;
Device      V750C;

/* Input Pins */

PIN 1 = Clk;                /* The clock signal for the counter and for the ASMs. */
PIN [2..5] = [PEncIn0..3];  /* Priority encoder. */
PIN 6 = ACK;

/* Output Pins */

PIN [23..21] = ![DecOut0..2];   /* Decoder. */
PIN [20..17] = [Q0..3];         /* Flip-flops for the output register. */
PIN 16 = Dval;

/* Pin Nodes */

PINNODE [34..33] = [CQ0..1];    /* Flip-flops for the counter. */
PINNODE [32..31] = [PEQ0..1];   /* Flip-flops for the priority encoder's register. */
PINNODE [30..29] = [KCQ0..1];   /* Flip-flops for the Key Control ASM. */
PINNODE [28..27] = [KBQ0..1];   /* Flip-flops for the Key Buffer Control ASM. */

/* Up Counter (2-Bit, Synchronous, Counts From 0 To 2) */

[CQ0..1].SP = 'b'0; /* Not used. */
[CQ0..1].AR = 'b'0; /* Not used. */
[CQ0..1].CKMUX = Clk;

Two = CQ1 & !CQ0;

CQ0.D = (CQ0 $ Kscan0) & !(Two & Kscan0);
CQ1.D = (CQ1 $ Kscan0 & CQ0) & !(Two & Kscan0);

/* Decoder (2 x 3, Active-Low Outputs) */

DecIn0 = CQ0;
DecIn1 = CQ1;

DecOut0 = !DecIn1 & !DecIn0;
DecOut1 = !DecIn1 & DecIn0;
DecOut2 = DecIn1 & !DecIn0;
```

```
/* Priority Encoder (4 x 2, Active-Low Inputs) */
PEncOut0 = !PEncIn1 & PEncIn2 # !PEncIn3;
PEncOut1 = !PEncIn2 # !PEncIn3;
PEncGS = ![PEncIn0..3]:#; /* Group select - active if at least one input is active. */
Kpress = PEncGS;

/* Priority Encoder's Register (2-Bit) */

[PEQ0..1].SP = 'b'0; /* Not used. */
[PEQ0..1].AR = 'b'0; /* Not used. */
[PEQ0..1].CK = Kscan1;

[PEQ0..1].D = [PEncOut0..1];

/* Output Register (4-Bit) */

[Q0..3].SP = 'b'0; /* Not used. */
[Q0..3].AR = 'b'0; /* Not used. */
[Q0..3].CK = Wreg;

[Q0..1].D = [PEQ0..1];
[Q2..3].D = [CQ0..1];

/* Key Control ASM */

[KCQ0..1].SP = 'b'0; /* Not used. */
[KCQ0..1].AR = 'b'0; /* Not used. */
[KCQ0..1].CK = !Clk; /* In phase opposition with Clk. */

Kack = DAC;
/* Gray encoding. */
$DEFINE KEY_SCAN_COUNTER                'b'00
$DEFINE KEY_SCAN_REGISTER               'b'01
$DEFINE KEY_VALID                       'b'11
$DEFINE WAIT_NOT_KACK_AND_NOT_KPRESS    'b'10

SEQUENCE [KCQ1..0] {
    PRESENT KEY_SCAN_COUNTER
        OUT Kscan0;
        IF Kpress NEXT KEY_SCAN_REGISTER;
        DEFAULT NEXT KEY_SCAN_COUNTER;
    PRESENT KEY_SCAN_REGISTER
        OUT Kscan1;
        NEXT KEY_VALID;
    PRESENT KEY_VALID
        OUT Kval;
        IF Kack NEXT WAIT_NOT_KACK_AND_NOT_KPRESS;
        DEFAULT NEXT KEY_VALID;
    PRESENT WAIT_NOT_KACK_AND_NOT_KPRESS
        IF !Kack & !Kpress NEXT KEY_SCAN_COUNTER;
        DEFAULT NEXT WAIT_NOT_KACK_AND_NOT_KPRESS;
}
```

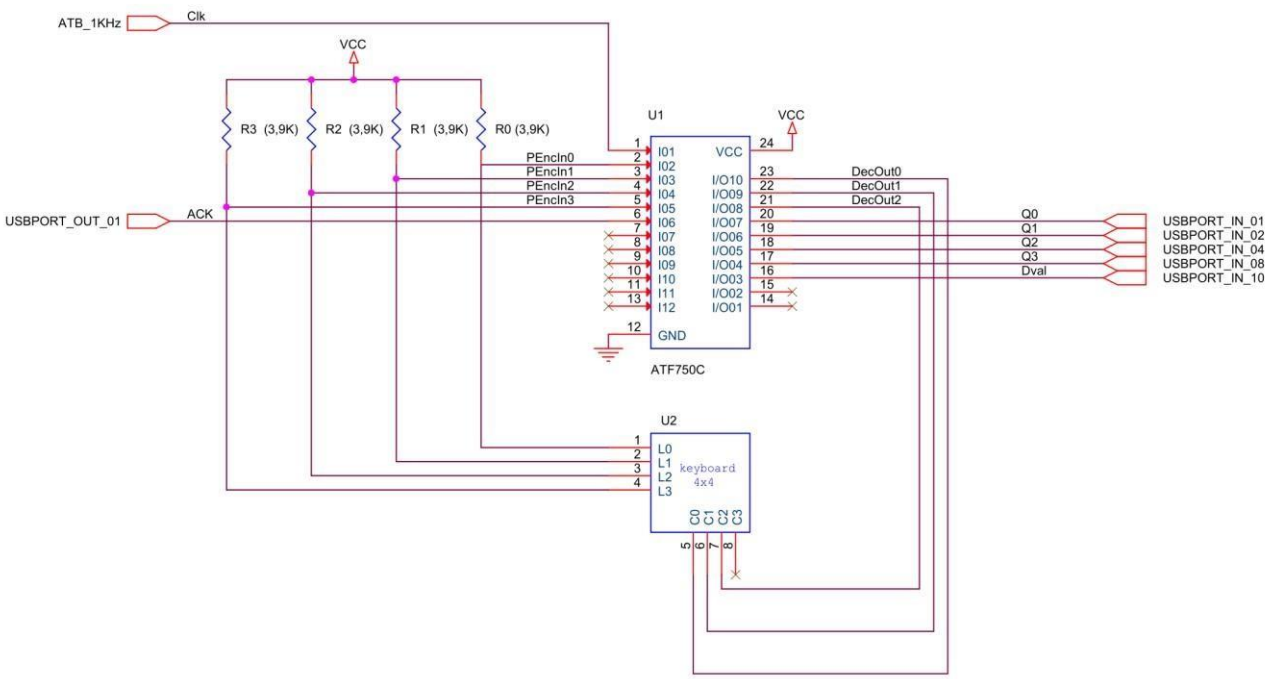
```
/* Key Buffer Control ASM */
[KBQ0..1].SP = 'b'0; /* Not used. */
[KBQ0..1].AR = 'b'0; /* Not used. */
[KBQ0..1].CKMUX = Clk;
DAV = Kval;

/* Gray encoding. */


$DEFINE WAIT_NOT_ACK_AND_DAV    'b'00
$DEFINE WRITE_REGISTER          'b'01
$DEFINE DATA_ACCEPTED           'b'11
$DEFINE DATA_VALID              'b'10

SEQUENCE [KBQ1..0] {
    PRESENT WAIT_NOT_ACK_AND_DAV
        IF !ACK & DAV NEXT WRITE_REGISTER;
        DEFAULT NEXT WAIT_NOT_ACK_AND_DAV;
    PRESENT WRITE_REGISTER
        OUT Wreg;
        NEXT DATA_ACCEPTED;
    PRESENT DATA_ACCEPTED
        OUT DAC;
        IF !DAV NEXT DATA_VALID;
        DEFAULT NEXT DATA_ACCEPTED;
    PRESENT DATA_VALID
        OUT Dval;
        IF ACK NEXT WAIT_NOT_ACK_AND_DAV;
        DEFAULT NEXT DATA_VALID;
}
```

A.2. Esquema elétrico do módulo *Keyboard Reader*



A.3.Código Java da classe *HAL*

```
// Hardware Abstraction Layer.
public class HAL {     private static int outputPort; // The value
on the output port.
    public static void main(String[] args) {
init();
    }
    public static void init() {
outputPort = 0;
out(outputPort);
    }
    // Checks, on the input port, if a specific bit is set.          return 0 !=
readBits(mask);
    }
    // Reads, from the input port, the bits specified by a mask.
public static int readBits(int mask) {          return in() &
mask;
    }
    // Writes, to the output port, on the bits specified by a mask (while keeping
the others), a value.
    public static void writeBits(int mask, int value) {
outputPort &= ~mask;          outputPort |= value &
mask;          setBits(outputPort);
clrBits(~outputPort);
    }
    // Sets, on the output port, the bits specified by a mask (and keeps the
others).
    public static void setBits(int mask) {
outputPort |= mask;          out(outputPort);
    }
    // Resets, on the output port, the bits specified by a mask (and keeps the
others).
    public static void clrBits(int mask) {
outputPort &= ~mask;
out(outputPort);
    }
    private static int in() {          return ~UsbPort.in(); //
UsbPort.in() reads from the input port.
    }
    private static void out(int value) {
```

A.4. Código Java da classe *KBD*

```
 import isel.leic.utils.Time;
// Keyboard.
public class KBD {
    public static final char NONE = 0;
    private static final char[] KEYBOARD = {
            '1', '4', '7',
            '*', '2', '5',
            '8', '0', '3',
            '6', '9', '#'
    };
    private static final int OUT_REGISTER_MASK_IN_PORT = 0x0F;
    private static final int DVAL_MASK_IN_PORT = 0x10; // Data Valid.
    private static final int ACK_MASK_OUT_PORT = 0x80; // Acknowledge.
    public static void main(String[] args) {
        HAL.init();
        init();
    }
    public static void init() {
        HAL.clrBits(ACK_MASK_OUT_PORT);
    }
    // Gets the key being pressed. Returns NONE if no key is being pressed.
    public static char getKey() {
        char key = HAL.isBit(DVAL_MASK_IN_PORT) ?
KEYBOARD[HAL.readBits(OUT_REGISTER_MASK_IN_PORT)] : NONE;
        if (NONE == key) {
            return NONE;
        }
        HAL.setBits(ACK_MASK_OUT_PORT);
        while (HAL.isBit(DVAL_MASK_IN_PORT));
        HAL.clrBits(ACK_MASK_OUT_PORT);
        return key;
    }
    // Waits for timeout milliseconds for a key press. Returns the pressed key or
NONE if no key was pressed during the specified  // time span.
    public static char waitKey(long timeout) {
        timeout += Time.getTimeInMillis();
        char key;
        while (Time.getTimeInMillis() <= timeout) {
            if (NONE != (key = getKey())) {
                return key;
            }        }        return NONE;     }
}
```