O módulo *Serial LCD Controller (LCDC)* é constituído por dois blocos principais:

i) *Serial Receiver (LCDSR)*

ii) *LCD Dispatcher (LCDD)*

No presente relatório apresenta-se a solução dos dois blocos, mencionados, que implementam o processo de visualização dos dados relativos à realização do jogo, mostrando os valores associados às teclas e respetivas funções no *LCD*. Enquanto o bloco *LCDSR é* responsável pela recolha da informação, controladamente, em formato *serial*, bit a bit e, sempre que ocorram dados válidos, transmite/entrega cada trama de 9 bits (onde, no bit zero (0) se encontra o valor *RS* e nos oito (8) bits restantes de dados), no formato *paralell*, ao *LCDD*.

O Bloco *LCDD* é responsável pela comunicação, ao mostrador *LCD*.

A implementação destes dois blocos permite a independência quanto à escrita no bloco *LCDSR*, que ao despachar a informação para o bloco *LCDD*, pode voltar a receber nova série de dados, enquanto o *LCDD* resolve a apresentação da informação no *LCD*.

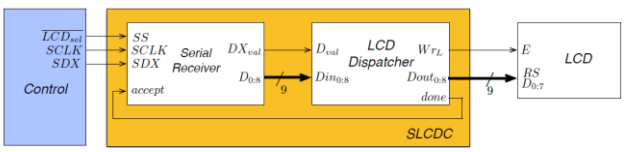

Figure 1

Diagrama de blocos do módulo Serial LCD Controller

1 Serial Receiver

O Bloco *LCDSR*, conforme Figure 1 é implementado com recurso a:

i) um bloco de controlo, designado por *Serial Control*, que efetua o controlo e validação dos dados recebidos no bloco e a sua transmissão para o bloco *CDSR*, conforme o diagrama de blocos representado na

ii) um bloco conversor série paralelo, designado por *Shift Register*, está implementado com recurso a um *Shift Register* que tem por encargo a transformação dos bits rececionados, conforme requerido, e sua transmissão ao bloco *LCDD*;

iii) um contador, designado por *Counter*, de 4 bits recebidos, cujo objetivo é o controlo e estabilidade dos dados no conversor série paralelo, ao receber o nono (9.º) bit, altera o valor da *dFlag* para um (1) e o sinal *Wr* passa a 1, e aguarda pela validação da paridade, para que seja possível passar os dados para o bloco *LCDSR*. Quando este contador recebe o décimo (10.º) bit, activa a *pFlag*, que passa a um (1), e ao ser analisado o sinal de *Rxerr*, caso esteja a um (1) ocorreu erro e é efetuado "reset", à informação contida neste bloco *LCDD*, que aguarda pela chegada de dados, mas caso seja zero (0), a informação ($D_{0..8}$ *e* DX_{val}) passa para o bloco *LCDSR* e aguarda pela conclusão do processo através da ativação do sinal *accept*. Fica assim disponível para processar mais dados;

iv) um bloco de validação de paridade, designado *Parity Check*, que é um contador de 1 bit, que quando recebe o décimo (10.º) bit, avalia se ocorreu erro de transmissão, com base no bit de paridade, no sinal data. Caso seja um (1) ocorreu um erro na transmissão, por alteração de um valor ímpar de bits, entre a origem e a sua receção e o processo é interrompido. Caso seja zero (0) a transmissão é processada.

vi) O bloco Serial Control, conforme Figure 2, seguiu-se o sugerido no enunciado do trabalho, por ser uma solução ajustada e exequível, cuja implementação de código VHDL está em anexo A.

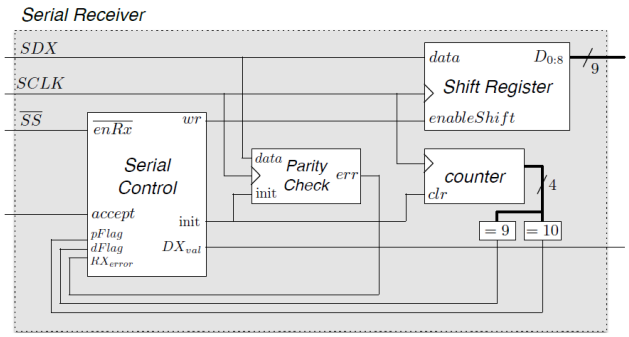

Figure 2

Diagrama de blocos do Bloco Serial Receiver

O bloco *Serial Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figure 3 e foi resultado do exercício de avaliação dos estados dos sinais de entrada que garantem o normal e independente funcionamento do módulo principal *Serial Receiver*, de alguma forma já explanados na explicação dos propósitos dos restantes 3 módulos adjacentes a este.

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

Para garantir a transmissão de dados em série o o módulo *Serial Control utiliza o protocolo que se plasma na figura.*

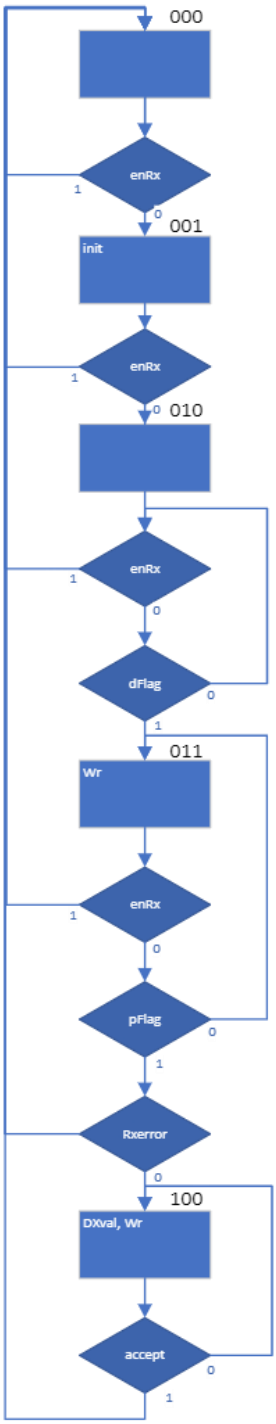

Figure 3

Máquina de estados do bloco Serial Control

2 *LCD Dispatcher*

O bloco *LCDD* que garante a passagem de informação para o LCD, e desta forma garante que o bloco LCDSR possa estar disponível para receber informação. O processo enquanto não sinal de Dval (=0) continua num ciclo sem fim, até que ocorre a ativação do sinal Dval (=1) e assim vai transmitir os sinais de E e de 9 bits, onde o primeiro é de RS (que indica se a trama é de controlo ou de dados) e 8 bits de informação. eguarda que é implementado pela ASM *ASM-chart* na Figure 3 e cuja descrição de hardware em VHDL está no anexo A

Atendendo a que a porta USB apenas garante a transferência de 8 bits simultaneamente, foi necessário adotar a implementação de hardware complementar para garantir a passagem de 10 bits em dois ciclos para o LCD e só depois proceder à sua visualização.

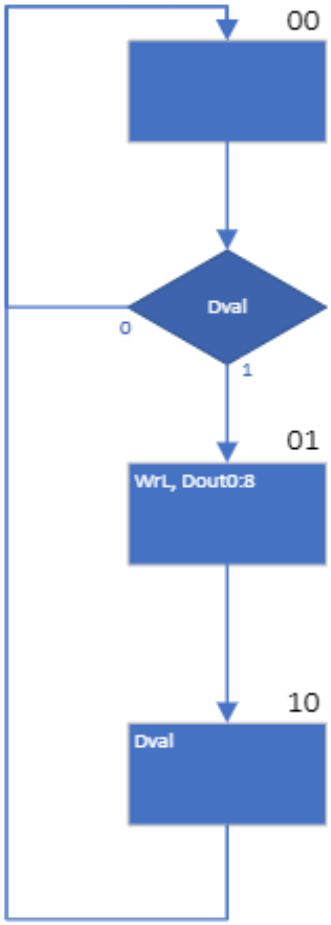

Figure 4

Máquina de estados do bloco Key Control

3 *Interface com o Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem Kotlin e seguindo a lógica apresentada no modelo da figura 14 do enunciado.

As classes *LCD* e *SerialEmitter* desenvolvidas são descritas em secções 3.1 e 3.2 e os códigos fonte desenvolvidos apresentam-se no anexo B.

3.1 Classe *LCD*

Sendo a interface entre hardware e software utiliza a classe *HAL*, que virtualiza o acesso ao

sistema *UsbPort*, e escreve no *LCD* usando a interface a 4 bits e é contituida pela função main, que genericamente após inicializar *HAL* e *LCD, e* para testes de validação das funções foram criados ciclos de mensagens.
Esta classe tem as seguintes funções:

i) private fun writeByteParallel(rs: Boolean, data: Int): Escreve um byte de comando/dados no LCD em paralelo;
ii) private fun writeByteSerial(rs: Boolean, data: Int): Escreve um byte de comando/dados no LCD em série;
iii) private fun writeByte(rs: Boolean, data: Int) Escreve um byte de comando/dados no LCD;
iv) fun writeCMD(data: Int): Escreve um comando no LCD;
v) private fun writeDATA(data: Int): Escreve um dado no LCD;
vi) fun init(): Envia a sequência de iniciação para comunicação a 4 bits. Com esta função é garantido o protocolo de funcionamento do LCD conforme protocolo definido pelo fabricante;
vii) fun write(c: Char): Escreve um caráter na posição corrente;
viii) fun write(text: String): Escreve uma string na posição corrente;
ix) fun clear(): Envia comando para limpar o ecrã e posicionar o cursor em (0,0)

3.2 Classe *SerialEmitter*

Esta classe, é usada conjuntamente por este módulo e pelo módulo Serial Score Controller, e recorre à classe HAL para passar informação para o LCD, na função main depois de inicializar, foram testados o envio de duas tramas de Score e comporta ainda as seguintes duas funções:

i) fun init(): Inicia a classe;
ii) fun send(addr: Destination, data: Int, size : Int): Envia uma trama para o SerialReceiver identificado o destino em addr,os bits de dados em ‘data’ e em size o número de bits a enviar;

4 Pins

Foram utilizada a configuração fornecida em SpaceInvaders, conforme anexo C.

5 Conclusões

Nesta fase do projeto é possível garantir apenas a visualização de informação no simulador, o que indica que a componente do software não apresenta erros graves, mas ocorre alguma incorreção no Hardware (VHDL), que estamos a analisar.

A. Descrição VHDL do bloco Serial Receiver

```
library ieee;
use ieee.std_logic_1164.all;

entity SerialReceiver is
port(
    rst: in std_logic;
    clk: in std_logic;
    SDX: in std_logic;
    SCLK: in std_logic;
    SS: in std_logic;
    accept: in std_logic;
    D: out std_logic_vector(8 downto 0);
    DXval: out std_logic
);
end SerialReceiver;

architecture structural of SerialReceiver is
component SerialControl is
port(
    rst: in std_logic;
    clk: in std_logic;
    enRx: in std_logic;
    accept: in std_logic;
    pFlag: in std_logic;
    dFlag: in std_logic;
    RXerror: in std_logic;
    wr: out std_logic;
    init: out std_logic;
    DXval: out std_logic
);
end component;

component ParityCheck is
port(
    data: in std_logic;
    init: in std_logic;
    clk: in std_logic;
    err: out std_logic
);
end component;

component Cont is
port(
    R: in std_logic;
    CE: in std_logic;
    Clk: in std_logic;
    Q: out std_logic_vector(3 downto 0)
    );
end component;
```

```
component ShiftRegister is
port(
    CLK: in std_logic;
    RST: in std_logic;
    data: in std_logic;
    enable: in std_logic;
    D: out std_logic_vector(8 downto 0)
    );
end component;

signal Serr: std_logic;
signal Swr: std_logic;
signal Sinit: std_logic;
signal Scount: std_logic_vector(3 downto 0);
signal Sdflag: std_logic;
signal Spflag: std_logic;

begin
SC: SerialControl port map(
    rst => rst,
    clk => clk,
    enRX => SS,
    accept => accept,
    pflag => Spflag,
    dflag => Sdflag,
    RXerror => Serr,
    wr => Swr,
    init => Sinit,
    DXval => DXval
);

Pcheck: ParityCheck port map(
    data => SDX,
    clk => SCLK,
    init => Sinit,
    err => Serr
);

ShRegister: ShiftRegister port map(
    data => SDX,
    CLK => SCLK,
    RST => rst,
    enable => Swr,
    D => D
);

Counter: Cont port map(
    R => Sinit,
    Clk => SCLK,
    CE => '1',
    Q => Scount
);

Spflag <= Scount(1) and Scount(3);
Sdflag <= Scount(0) and Scount(3);
end structural;
```

B. Descrição VHDL do bloco Serial Control

```
library ieee;
use ieee.std_logic_1164.all;

entity SerialControl is
port(
    rst: in std_logic;
    clk: in std_logic;
    enRx: in std_logic;
    accept: in std_logic;
    pFlag: in std_logic;
    dFlag: in std_logic;
    RXerror: in std_logic;
    wr: out std_logic;
    init: out std_logic;
    DXval: out std_logic
);
end SerialControl;

architecture behavioral of SerialControl is

type STATE_TYPE is (STATE_000, STATE_001, STATE_010, STATE_011, STATE_100);
signal CurrentState, NextState : STATE_TYPE;

begin

--Flip-Flop
CurrentState <= STATE_000 when rst ='1' else NextState when rising_edge(clk);

GenerateNextState:
process(CurrentState, enRX, dFlag, pFlag, RXerror, accept)
begin

case CurrentState is
    when STATE_000 => if (enRX = '1') then
                                                NextState <= STATE_000;
                                            else
                                                NextState <= STATE_001;
                                            end if;

    when STATE_001 => if (enRX = '1') then
                                                NextState <= STATE_000;
                                            else
                                                NextState <= STATE_010;
                                                end if;

    when STATE_010 => if (enRX = '1') then
                                                NextState <= STATE_000;
                                            elsif (dFlag = '0') then
                                                NextState <= STATE_010;
                                            else
                                                NextState <= STATE_011;
                                                end if;
    when STATE_011 => if (enRX = '1') then
                                                NextState <= STATE_000;
```

```
                                        elsif (pFlag = '0') then
                                               NextState <= STATE_011;
                                        elsif (RXerror = '1') then
                                               NextState <= STATE_000;
                                        else
                                               NextState <= STATE_100;
                                        end if;

    when STATE_100 => if (accept = '0') then
                                               NextState <= STATE_100;
                                        else
                                               NextState <= STATE_000;
                                        end if;

    end case;

end process;

init <= '1' when (CurrentState = STATE_001) else '0';
wr <= '1' when (CurrentState = STATE_011 or CurrentState = STATE_100) else '0';
DXval <= '1' when (CurrentState = STATE_100) else '0';

end behavioral;
```

C. Descrição VHDL do bloco Serial Control

```
library ieee;
use ieee.std_logic_1164.all;

entity LcdDispatcher is
port(
    rst: in std_logic;
    clk: in std_logic;
    Dval: in std_logic;
    Din: in std_logic_vector(8 downto 0);
    Wrl: out std_logic;
    Dout: out std_logic_vector(8 downto 0);
    done: out std_logic
);
end LcdDispatcher;

architecture behavioral of LcdDispatcher is

type STATE_TYPE is (STATE_00, STATE_01, STATE_10);
signal CurrentState, NextState : STATE_TYPE;

begin

--Flip-Flop
CurrentState <= STATE_00 when rst ='1' else NextState when rising_edge(clk);

GenerateNextState:
process(CurrentState, Dval)
    begin

    case CurrentState is
    when STATE_00 => if (Dval = '0') then
                                                NextState <= STATE_00;
                                            else
                                                NextState <= STATE_01;
                                            end if;
    when STATE_01 => NextState <= STATE_10;

    when STATE_10 => NextState <= STATE_00;

    end case;

end process;

Wrl <= '1' when (CurrentState = STATE_01) else '0';
Dout <= Din when (CurrentState = STATE_01);
done <= '1' when (CurrentState = STATE_10) else '0';

end behavioral;
```

A. 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 maskRS = 0x40
    private val maskE = 0x20
    private val maskClk = 0x10
    private val LCDsize = 9
    data class Pos(var line: Int = 0 , var col: Int = 0)
    var cursor = Pos()

    // 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) {
        var d: Int = 0
        val shiftedData = data.shl(1)
        d = if(rs){
```

```
            shiftedData.or(0x001)
        }else{
            shiftedData
        }
       SerialEmitter.send(SerialEmitter.Destination.LCD, d, LCDsize)
}

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

// Escreve um comando no LCD
private 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)
    cursor.col++
}

// 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) {
    if (line == -1){
        cursor.line = when (cursor.line){
```

```
                1 -> 0
                else -> 1
            }
            val l = cursor.line * 64
            val pos = (l + cursor.col).or(0x80)
            return writeCMD(pos)
        }
        val l = line * 64
        val pos = (l + column).or(0x80) //cmd DDRAM
        cursor.col = column

        writeCMD(pos)
    }

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

fun main() {
    HAL.init()
    SerialEmitter.init()
    LCD.init()

   // while (true);

/*
    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
            }

    }
```

B. Código Kotlin - SerialEmitter

```
object SerialEmitter { // Envia tramas para os diferentes módulos Serial Receiver.
    enum class Destination {LCD, SCORE}
    private val LCD_sel = 0x80
    private val SC_sel = 0x40
    private val SDX = 0x01

    private val SCLK = 0x10
    // Inicia a classe
    fun init(){
        HAL.setBits(SC_sel)
        HAL.setBits(LCD_sel)
        HAL.clrBits(SCLK)
    }
// Envia uma trama para o SerialReceiver identificado o destino em addr,os bits de dados em
    //‘data’ e em size o número de bits a enviar.
    fun send(addr: Destination, data: Int, size : Int){
        var parity = 0
        var sendData = data

        if(addr == Destination.LCD){
            HAL.clrBits(LCD_sel)
            var countBit = 0
            for (i in 0..<size){
                val bit = SDX.and(sendData)
                HAL.writeBits(SDX, sendData)

                if (sendData.and(SDX) == 1){
                    parity++
                }
                sendData = sendData.shr(1)

                HAL.setBits(SCLK)
                Thread.sleep(2)
                HAL.clrBits(SCLK)
                Thread.sleep(2)
                countBit++

            }
            val p = parity % 2
            Thread.sleep(1)
            HAL.writeBits(SDX, p)
            HAL.setBits(SCLK)
            Thread.sleep(2)
            HAL.clrBits(SCLK)
            Thread.sleep(2)

            HAL.setBits(LCD_sel)

        }
        else{
            HAL.clrBits(SC_sel)
            var countBit = 0
            for (i in 0..<size){
                //val bit = SDX.and(sendData)
```

```
                HAL.writeBits(SDX, sendData)

                if (sendData.and(SDX) == 1){
                    parity++
                }
                sendData = sendData.shr(1)

                HAL.setBits(SCLK)
                Thread.sleep(50)
                HAL.clrBits(SCLK)
                Thread.sleep(50)
                countBit++

            }
            HAL.writeBits(SDX, parity % 2)
            HAL.setBits(SCLK)
            Thread.sleep(250)
            HAL.clrBits(SCLK)
            Thread.sleep(250)

            HAL.setBits(SC_sel)
        }
    }
}

fun main(){
    HAL.init()
    SerialEmitter.init()
    SerialEmitter.send(SerialEmitter.Destination.SCORE, 0b1111011, 7)
    //Thread.sleep(5000)
    SerialEmitter.send(SerialEmitter.Destination.SCORE, 0b00000000, 7)

}
```

A. Atribuição de pinos do módulo SpaceInvaders

```
set_global_assignment -name BOARD "MAX 10 DE10 - Lite"
set_global_assignment -name DEVICE 10M50DAF484C6GES
set_global_assignment -name FAMILY "MAX 10"

set_location_assignment PIN_C10 -to rst
set_location_assignment PIN_P11 -to clk

set_location_assignment PIN_W8 -to RS
set_location_assignment PIN_V5 -to EOut


set_location_assignment PIN_W5 -to I[0]
set_location_assignment PIN_AA14 -to I[1]
set_location_assignment PIN_W12 -to I[2]
set_location_assignment PIN_AB12 -to I[3]
set_location_assignment PIN_AB11 -to O[0]
set_location_assignment PIN_AB10 -to O[1]
set_location_assignment PIN_AA9 -to O[2]
set_location_assignment PIN_AA8 -to O[3]
```