O módulo de interface com o *LCD* (*Serial LCD Interface*, *SLCDC*) implementa a receção em série da informação enviada pelo módulo de controlo, entregando-a posteriormente ao *LCD*, conforme representado na Figura 1.

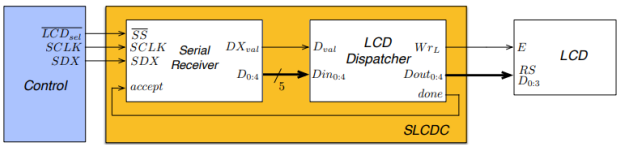

Figura 1 – Diagrama de blocos do *Serial LCD Controller*

O *SLCDC* recebe em série uma mensagem constituída por cinco bits de informação. A comunicação com o *SLCDC* realiza-se segundo o protocolo ilustrado na Figura 2, tendo como primeiro bit de informação, o bit RS que indica se a mensagem é de controlo ou dados, os restantes bits contêm os dados a transmitir ao *LCD*.

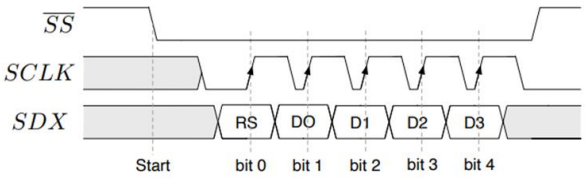

Figura 2 – Protocolo de comunicação com *Serial LCD Controller*

O emissor, realizado em software, quando pretende enviar uma trama para o *SLCDC* promove uma condição de início de trama (Start), que corresponde a uma transição descendente na linha de $\overline{LCD_{sel}}$. Após a condição de início, o *SLCDC* armazena os bits da trama nas transições ascendentes do sinal $SCLK$.

1 Serial Receiver

O bloco *Serial Receiver* do *SLCDC* é constituído por três blocos principais: i) um bloco de controlo; ii) um contador de bits recebidos; e iii) um bloco conversor série paralelo, designados respetivamente por *Serial Control*, *Counter*, e *Shift Register*. O *Serial Receiver* foi implementado com base no diagrama de blocos apresentado na Figura 3.

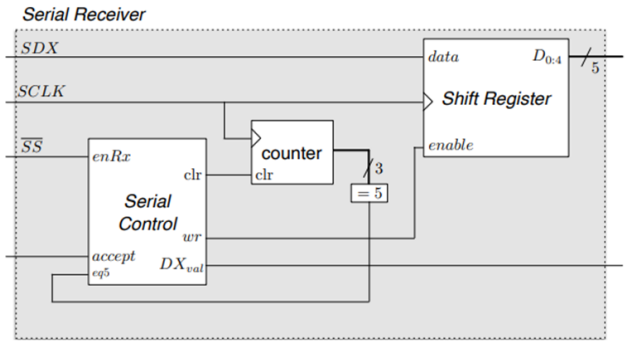

Figura 3 – Diagrama de blocos do Serial Receiver

O bloco *Shift Register* foi implementado de acordo com o diagrama de blocos representado na Figura 4.

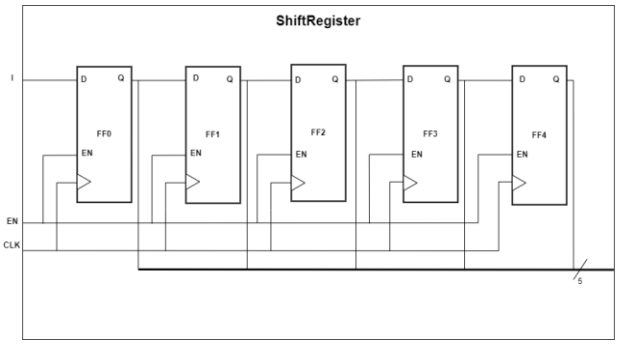

Figura 4 – Diagrama de blocos do Shift Register

Foram utilizados 5 *Flip-Flops* ligados em série. A entrada do primeiro *Flip-Flop* é a entrada *I* do circuito, e os restantes *Flip-Flop* têm como a entrada a saída do *Flip-Flop* anterior. As saídas dos cinco *Flip-Flops* representam o sinal de dados de cinco bits.

O bloco *Serial Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 5.
A descrição hardware do bloco *Serial Receiver* em *VHDL* encontra-se no Anexo A.

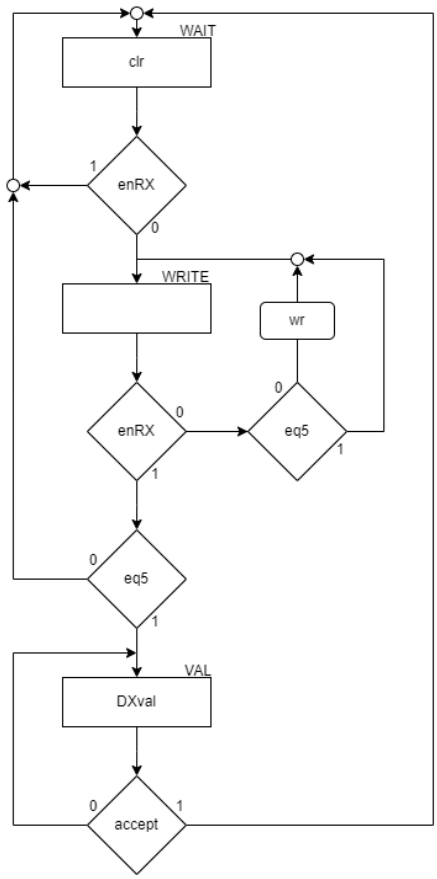

Figura 5 – Máquina de estados do bloco *Serial Control*

Inicialmente a máquina encontra-se no estado **WAIT**, em que se limpa o valor do contador e se espera para que o *LCD* seja selecionado. Quando é selecionado o serial receiver do *LCD* através de $enRX$, a máquina passa para o estado **WRITE** em que ativa o *Shift Register* através do sinal *wr* enquanto não são enviados os 5 bits de informação (Figura 2). Assim que são enviados os cinco bits de informação, a máquina desativa o sinal *wr* e espera para que o *LCD* deixe de ser selecionado através de *enRX* para sinalizar que tem dados a enviar ao *LCD Dispatcher*. Se forem enviados bits de informação insuficientes ou a mais, então a máquina de estados invalida a trama quando o *LCD* deixa de ser selecionado e volta para o estado **WAIT**. Caso seja enviada uma trama válida a máquina passa para o estado **VAL**. Neste estado, envia o sinal $DXval$ para o *LCD Dispatcher*, para indicar que tem dados a enviar. Quando o sinal $accept$ é enviado pelo *LCD Dispatcher* a indicar que a trama foi aceite, pode ser iniciado um novo ciclo de escrita e a máquina volta para o estado **WAIT**.

2 Dispatcher

O bloco *Serial Dispatcher* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 6.
A descrição hardware do bloco *Serial Dispatcher* em VHDL encontra-se no Anexo B.

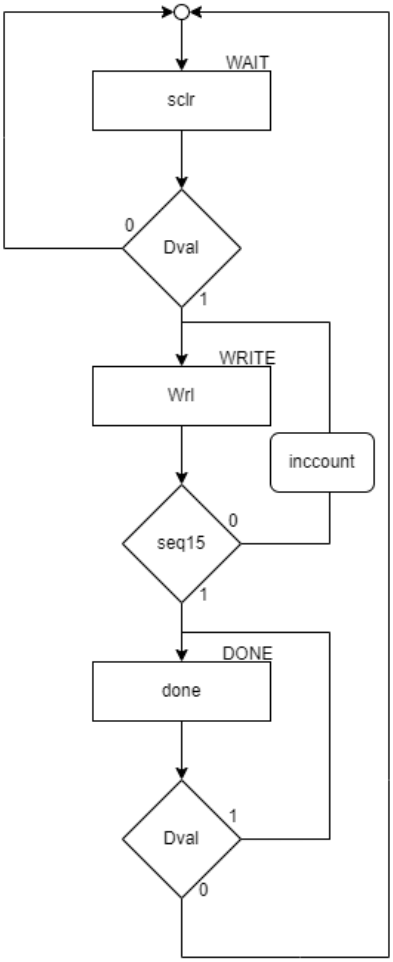

Figura 6 - Máquina de estados do bloco *LCD Dispatcher*

O bloco *Dispatcher* entrega a trama recebida pelo Serial Receiver ao LCD através da ativação do sinal WrL, após este ter recebido uma trama válida, indicado pela ativação do sinal $DXval$.
O *LCD* processa as tramas recebidas de acordo com os comandos definidos pelo fabricante, não sendo necessário esperar pela sua execução para libertar o canal de receção série. Assim, o *Dispatcher* pode sinalizar ao *Serial Receiver* que a trama foi processada, ativando o sinal $done$.
Esta implementação de máquina de estados usa um contador de 4 bits para obedecer às especificações do *LCD* (Figura 7).
No estado **WRITE**, em que o sinal de WrL está ativo, ou seja, está a ser enviado um sinal High de $Enable$ para o *LCD*, incrementa-se o contador até chegar a 15. Quando o contador chega a 15, a máquina de estados passa para o

estado **DONE** em que sinaliza ao *Serial Receiver* que a trama foi processada. Neste estado, a máquina espera até o *Serial Receiver* baixar o sinal *Dval*, e quando o faz, volta para o estado inicial **WAIT** em que se faz reset ao valor do contador. Como cada ciclo de clock na placa DE-10 Lite dura 20 ns (50 MHZ), para o contador chegar a 15 são necessários 300 ns, ou 15 ciclos de clock. Então, a máquina obedece às especificações do *LCD*, que requerem que o sinal de EN esteja ativo pelo menos 230 ns.

Parameter	Symbol	Min (1)	Typ (1)	Max (1)	Unit
Enable Cycle Time	t_c	500	-	-	ns
Enable Pulse Width (High)	t_w	230	-	-	ns
Enable Rise/Fall Time	t_r,t_f	-	-	20	ns
Address Setup Time	t_{as}	40	-	-	ns
Address Hold Time	t_{ah}	10	-	-	ns
Data Setup Time	t_{ds}	80	-		ns
Data Hold Time	t_h	10	-	-	ns

Figura 7 – Especificações de ciclo de escrita do *LCD*

3 Interface com o Control

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem *Kotlin* e seguindo a arquitetura lógica apresentada na Figura 8.

Figura 8 – Diagrama lógico do módulo *Control* de interface com o módulo *SLCDC*

LCD e *Serial Emitter* desenvolvidos são descritos nas secções 3.1 e 3.2, e o código fonte desenvolvido nos Anexos D e E, respetivamente.

3.1 *LCD*

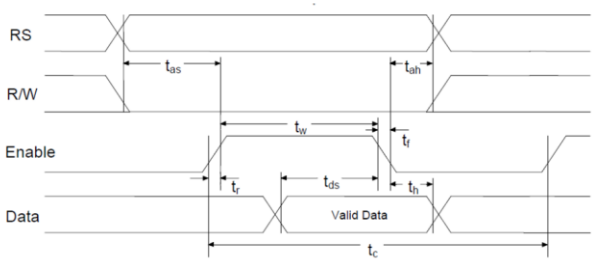

Figura 9 – Diagrama temporal de ciclo de escrita no *LCD*

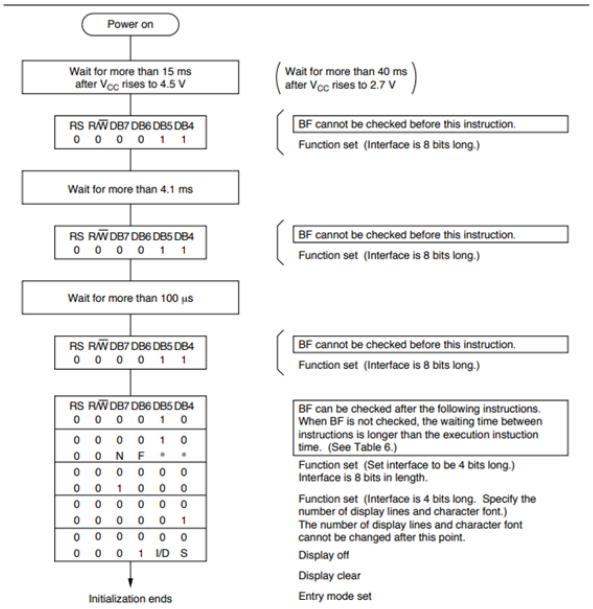

Figura 10 – Sequência de inicialização do *LCD*

Instruction	Code RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time (max) (when f_{cp} or f_{OSC} is 270 kHz)
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 µs
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 µs
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.	37 µs
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).	37 µs
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 µs
Set DDRAM address	0	0	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.	37 µs
Read busy flag & address	0	1	BF	AC	AC	AC	AC	AC	AC	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.	0 µs
Write data to CG or DDRAM	1	0	Write data								Writes data into DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*
Read data from CG or DDRAM	1	1	Read data								Reads data from DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*

I/D = 1: Increment
I/D = 0: Decrement
S = 1: Accompanies display shift
S/C = 1: Display shift
S/C = 0: Cursor move
R/L = 1: Shift to the right
R/L = 0: Shift to the left
DL = 1: 8 bits, DL = 0: 4 bits
N = 1: 2 lines, N = 0: 1 line
F = 1: 5 × 10 dots, F = 0: 5 × 8 dots
BF = 1: Internally operating
BF = 0: Instructions acceptable

DDRAM: Display data RAM
CGRAM: Character generator RAM
ACG: CGRAM address
ADD: DDRAM address (corresponds to cursor address)
AC: Address counter used for both DD and CGRAM addresses

Execution time changes when frequency changes
Example: When f_{cp} or f_{OSC} is 250 kHz,
37 µs × $\frac{270}{250}$ = 40 µs

Note: — indicates no effect.
* After execution of the CGRAM/DDRAM data write or read instruction, the RAM address counter is incremented or decremented by 1. The RAM address counter is updated after the busy flag turns off. In Figure 10, t_{ADD} is the time elapsed after the busy flag turns off until the address counter is updated.

Figura 11 – Lista de comandos do *LCD*

A classe *LCD* é responsável por controlar o ecrã *LCD* em Hardware, através de comunicação serial com interface de 4 bits.

Foram adicionadas as constantes `LINES`, `COLS`, `RSBITMASK`, `ENMASK`, `SERIALMASK`, `DATAMASK`, `PULSEDELAY`, `RISEDELAY`, `INITDELAY`, `CMDDELAY`, `DATA_FRAME` E `CMD_FRAME`.

As constantes *`LINES`* e *`COLS`* representam as linhas e colunas do *LCD*, as constantes *`MASK`* representam as posições dos respetivos sinais na *USB Port*, as constantes `DELAY` representam os tempos de atraso necessários para o funcionamento correto do *LCD* e as constantes `FRAME` representam as tramas de 5 bits a enviar para o *LCD*.

Também foi utilizada a função global adicional `waitTimeNano`, que espera um tempo definido em nanossegundos

Foi adicionada a função `Pulse()`, que realiza um pulso de *enable* com tempos de espera `PULSEDELAY` (500 nanosegundos) (ver tc na Figura 9) com a utilização de `HAL.setBits()`, e `HAL.clrBits()`.

A função `WriteNibbleParallel(rs: Boolean, data: Int)` ativa ou desativa o bit rs do LCD dependendo do valor rs passado como parâmetro (false para comando, true para dados). Após um tempo de espera de 100 nanosegundos (RISEDELAY), envia os dados para o *LCD* e chama a função `Pulse()`, que escreve os dados no *LCD*.

A função `WriteNibbleSerial(rs: Boolean, data: Int)` prepara os 5 bits de dados a enviar através do Serial Emitter e envia-os através de `SerialEmitter.send()`. A trama de cinco bits está representada na Figura 2.

A função `WriteNibble(rs : Boolean, data: Int)` escreve um nibble em modo paralelo ou serial, dependendo do valor do bit na posição `SERIALMASK` da *USB Port*. Como apenas vão ser escritos bits em modo serial através do *SLCDC*, esta função apenas envia bits em modo serial e não existe bit atribuído na USB Port para Serial.

A função `writeByte(rs : Boolean, data: Int)` escreve dois nibbles, chamando a função `writeNibble` duas vezes. Inicialmente escreve o valor alto de data (bits de data deslocados quatro vezes para a direita) e posteriormente o valor baixo de data (data and 00001111).

A função `writeCMD(data: Int)` escreve um comando no *LCD*, sendo que chama a função `writeByte` com o parâmetro `rs` False, e espera o tempo `CMDDELAY`(2 milissegundos).

A função `writeDATA(data: Int)` escreve dados no *LCD*, sendo que chama a função `writeByte` com o parâmetro `rs` True, e depois espera `CMDDELAY` (2 milissegundos).

A função `init()` realiza a sequência de inicialização do *LCD* representada na Figura 10, ao utilizar as funções

`writeNibble` e `writeCMD`, com os tempos de espera especificados.

A função `write(c: Char)` escreve no *LCD* o código do caráter passado como parâmetro ao chamar a função `writeDATA`.

A função `write(text: String)` escreve no *LCD* uma String ao chamar a função `write` anterior para cada caráter na String.

A função `cursor(line: Int, column: Int)` Posiciona o *LCD* ao chamar a função `writeCMD` com o comando **Set DDRAM address** do *LCD*.

A função `clear()` Limpa o *LCD* ao chamar a função `writeCMD` com o comando **Clear Display** do *LCD*

3.2 Serial Emitter

A classe *Serial Emitter* é responsável por envio de dados para o *LCD* ou o controlador da porta em modo serial.

Foram adicionadas as constantes `DELAY`, `SDXMASK`, `SCLKMASK` e `BUSYMASK`.

As constantes `MASK` representam as posições dos respetivos sinais na *USB Port*, e a constante `DELAY` representa o tempo de atraso entre pulsos de `SCLK`.
Também foi utilizada a função global adicional `waitTimeNano`, que espera um tempo definido em nanossegundos.

Foi adicionada a função `clkPulse()`, que realiza um pulso de `SCLK` com tempos de espera `DELAY`(100 nanosegundos) ao chamar `HAL.setBits()`, e `HAL.clrBits()`.

A função `init()` inicia a classe colocando o sinal `SCLK` a 0, e desselecionando os *Serial Receivers* da porta e do *LCD*.

A função `send(addr: Destination, data: Int)` envia os dados serialmente. Inicialmente seleciona um dos *Serial Controllers* através de `HAL.writeBits()`, e espera um tempo `DELAY`(100 nanosegundos) para o hardware preparar a escrita. Após o tempo `DELAY` são escritos os bits da trama de dados um a um através de `HAL.writeBits()` de acordo com a Figura 2, sendo que para cada um deles é chamada a função `clkPulse()`.

A função `isbusy()` verifica se o canal série está ocupado, utilizando a função `HAL.isBit()` para verificar o mesmo.

4 Conclusões

Neste módulo foi implementado o módulo *Serial LCD Controller* através dos blocos *Serial Receiver* e *LCD Dispatcher*, que implementa a receção em série da informação enviada pelo módulo de controlo, entregando-a posteriormente ao *LCD*.

Os blocos *Serial Control* e *LCD Dispatcher* operam com a frequência da placa DE-10 Lite da Intel (50 MHz), ou um ciclo a cada 20 nanosegundos.

A máquina de estados do *LCD Dispatcher* irá gastar 20 nanosegundos no primeiro estado com *wrL* desativado assumindo que *Dval* está ativo, 320 nanosegundos no segundo estado com *wrL* ativado, e no último estado o tempo para o *Serial Receiver* baixar *Dval* com *done* ativado e *wrL* desativado.

Para enviar 5 bits de informação, O *Serial Receiver* demora 20 ns depois de ser selecionado o *Serial Controller* do *LCD* a passar para o segundo estado. Neste segundo estado, o circuito demora o tempo do software a enviar os 5 bits de informação através de ciclos de *SCLK* e ativar o sinal *SS*. No terceiro estado, a máquina demora o tempo do *LCD Dispatcher* enviar os dados e ativar o sinal *Done* (340 nanosegundos) mais 20 nanosegundos quando *Done* é ativado.

Em termos de recursos da placa DE-10 Lite, na implementação do *Serial LCD Control* foram utilizados 23/49670 elementos lógicos (0,05%), 16 registos, e 11/360 pinos (3%).

A. Descrição VHDL do bloco *Serial Receiver*

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY SerialReceiver is
PORT( busy: out std_logic;
            SDX : in STD_LOGIC;
            SCLK : in STD_LOGIC;
            SS : in std_logic;
            accept : IN STD_LOGIC;
            clk: IN std_logic;
            D : out std_logic_vector(4 downto 0);
            DXval : out STD_LOGIC;
            RST: in std_logic
            );
end SerialReceiver;

ARCHITECTURE logic OF SerialReceiver is



component SerialControl is
      port (
            RST : in std_logic;
            clk : in std_logic;
            enRX : in std_logic;
            accept : in std_logic;
            eq5 : in std_logic;
            clr : out std_logic;
            wr : out std_logic;
            DXval : out std_logic
      );
end component;



component counterUp3 is
PORT(
            RST : in STD_LOGIC;
            CE : in std_logic;
            CLK : IN STD_LOGIC;
            Q : out std_logic_vector(2 downto 0)
            );
end component;

component Shiftregister is
PORT( I : in std_logic;
            EN : in STD_LOGIC;
            RST : in std_logic;
            CLK : IN STD_LOGIC;
            DATA : out std_logic_vector(4 downto 0)
            );
end component;

signal sclr, swr, seq5, sval: std_logic;
signal scounter : std_logic_vector ( 2 downto 0);
```

```
Begin

Control : SerialControl port map(
      RST => RST,
      clk => clk,
      enRX => SS,
      accept => accept,
      eq5 => seq5,
      clr => sclr,
      wr => swr,
      DXval => sval
);

Counter: counterUp3 port map(
      RST => sclr,
      CE => '1',
      CLK => SCLK,
      Q => scounter
);


SRegister : Shiftregister port map(
      I => SDX,
      EN => swr,
      RST => RST,
      CLK => SCLK,
      DATA => D
);

busy <= sval;
DXval <= sval;
seq5 <= scounter(0) and not scounter(1) and scounter(2) and not sclk;
end logic;
```

B. Descrição VHDL do bloco *LCD Dispatcher*

```
LIBRARY IEEE;
use IEEE.std_logic_1164.all;

entity LCD_Dispatcher is
      port (
            clk : in std_logic;
            RST : in std_logic;
            Dval : in std_logic;
            Din : in std_logic_vector ( 4 downto 0);
            WrL : out std_logic;
            Dout : out std_logic_vector (4 downto 0);
            done : out std_logic
      );
end LCD_Dispatcher;



architecture logic of LCD_Dispatcher is

component counterUp4 is
PORT(
            RST : in STD_LOGIC;
            CE : in std_logic;
            CLK : IN STD_LOGIC;
            Q : out std_logic_vector(3 downto 0)
            );
end component;


type STATE_TYPE is (STATE_WRITE, STATE_DONE, STATE_WAIT);

signal currentstate, nextstate: STATE_TYPE;

signal inccount: std_logic;
signal seq15 : std_logic;
signal scount: std_logic_vector(3 downto 0);
signal sclr: std_logic;


begin

      counter: counterUp4 port map(
            clk => clk,
            CE => inccount,
            RST => sclr,
            Q => scount
            );

      currentstate <= STATE_WAIT when RST = '1' else nextstate when
rising_edge(clk);

      GenerateNextState:
      process(currentstate, Dval, seq15)
            Begin
            case currentstate is
                  when STATE_WAIT    =>  if(Dval = '1') then
```

```
                                        nextstate <= STATE_WRITE;
                                    else
                                        nextstate <= STATE_WAIT;
                                    end if;

                when STATE_WRITE  =>  if(seq15 = '0') then
                                        nextstate <= STATE_WRITE;
                                    else
                                        nextstate <= STATE_DONE;
                                    end if;


                WHEN STATE_DONE  =>   if(Dval = '1') then
                                      nextstate <= STATE_DONE;
                                      else
                                        nextstate <= STATE_WAIT;
                                      end if;


            end case;
        end process;
        sclr <= '1' when currentstate = STATE_WAIT else '0';
        seq15 <= scount(3) and scount(2) and scount(1) and scount(0);
        inccount <= '1' when currentstate = STATE_WRITE and seq15 = '0' else '0';
        done <= '1' when currentstate = STATE_DONE else '0';
        WrL <= '1' when currentstate = STATE_WRITE  else '0';
        Dout <= Din;

end logic;
```

C. Atribuição de pinos do módulo *SLCDC*

```
#============================================================
# Altera DE10-Lite board settings
#============================================================
set_global_assignment -name FAMILY "MAX 10 FPGA"
set_global_assignment -name DEVICE 10M50DAF484C6GES
set_global_assignment -name TOP_LEVEL_ENTITY "DE10_Lite"
set_global_assignment -name DEVICE_FILTER_PACKAGE FBGA
set_global_assignment -name SDC_FILE DE10_Lite.sdc
set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE IMAGE WITH ERAM"

#============================================================
# CLOCK
#============================================================
#set_instance_assignment -name IO_STANDARD "3.3-V LVTTL" -to CLOCK_50
#set_instance_assignment -name IO_STANDARD "3.3-V LVTTL" -to CLOCK2_50
#set_instance_assignment -name IO_STANDARD "3.3-V LVTTL" -to CLOCK_ADC_10
set_location_assignment PIN_P11 -to clk
#set_location_assignment PIN_N14 -to CLOCK2_50
#set_location_assignment PIN_N5 -to CLOCK_ADC_10

#============================================================
# SW
#============================================================
set_location_assignment PIN_C10 -to LCDsel
set_location_assignment PIN_C11 -to SCLK
set_location_assignment PIN_D12 -to SDX
#set_location_assignment PIN_C12 -to SW[3]
#set_location_assignment PIN_A12 -to SW[4]
#set_location_assignment PIN_B12 -to SW[5]
#set_location_assignment PIN_A13 -to SW[6]
#set_location_assignment PIN_A14 -to SW[7]
#set_location_assignment PIN_B14 -to SW[8]
set_location_assignment PIN_F15 -to RST

#============================================================
# Keyboard
#============================================================
#set_location_assignment PIN_W5     -to KEYPAD_LIN[0]
#set_location_assignment PIN_AA14   -to KEYPAD_LIN[1]
#set_location_assignment PIN_W12    -to KEYPAD_LIN[2]
#set_location_assignment PIN_AB12   -to KEYPAD_LIN[3]
#set_location_assignment PIN_AB11   -to KEYPAD_COL[0]
#set_location_assignment PIN_AB10   -to KEYPAD_COL[1]
#set_location_assignment PIN_AA9    -to KEYPAD_COL[2]
#set_location_assignment PIN_AA8    -to KEYPAD_COL[3]

#============================================================
# LCD
#============================================================
set_location_assignment PIN_W8      -to D[0]
set_location_assignment PIN_V5      -to   E
#set_location_assignment PIN_AA15   -to Data_LCD[0]
#set_location_assignment PIN_W13    -to Data_LCD[1]
#set_location_assignment PIN_AB13   -to Data_LCD[2]
#set_location_assignment PIN_Y11    -to Data_LCD[3]
set_location_assignment PIN_W11     -to D[1]
```

```
set_location_assignment PIN_AA10    -to D[2]
set_location_assignment PIN_Y8      -to D[3]
set_location_assignment PIN_Y7      -to D[4]
#============================================================
# End of pin and io_standard assignments
#============================================================
```

D. Código Kotlin – *LCD*

```
/**
 * 22/5/2023
 *
 *
 * Manages the LCD for the [AccessControlSystem], using 4-bit interface.
 * @property LINES the number of lines on the LCD
 * @property COLS the number of columns on the LCD
 * @author Bernardo Pereira
 * @author AntÃ³nio Paulino
 * @see SerialEmitter
 * @see HAL
 */

object LCD {

    const val LINES = 2

    const val COLS = 16

    /**
     * The bit mask of the RS output on the USB output port
     */
    private const val RS_BIT_MASK = 0b00010000

    /**
     * The bit mask of the EN output on the USB output port
     */
    private const val EN_MASK = 0b00100000

    /**
     * The bit mask of the Data input on the USB input port
     */
    private const val DATA_MASK = 0b00001111

    /**
     * The delay in nanoseconds between EN pulses to the LCD
     */
    private const val PULSE_DELAY = 500


    /**
     * The delay in nanoseconds for the rise time of the bit signals.
     */
    private const val RISE_DELAY = 100


    /**
     * The delay between LCD initialization commands
     */
    private const val INIT_DELAY = 30

    /**
     * The delay in milliseconds between LCD commands.
     */
    private const val CMD_DELAY = 2
```

```
    /**
     * A nibble with the RS bit set to 0, to send a command to the LCD
     */
    private const val CMD_FRAME = 0b00000

    /**
     * A nibble with the RS bit set to 1, to send data to the LCD
     */
    private const val DATA_FRAME = 0b00001


    /**
     * Sends an EN pulse to the LCD
     */
    private fun pulse() {
        waitTimeNano(PULSE_DELAY)
        HAL.setBits(EN_MASK)
        waitTimeNano(PULSE_DELAY)
        HAL.clrBits(EN_MASK)
        waitTimeNano(PULSE_DELAY)
    }

    /**
     * Sends a command or data nibble (4 bits) to the LCD in parallel mode.
     * @param rs Defines whether to send a command (false) or data (true)
     * @param data Data to send
     */
    private fun writeNibbleParallel(rs: Boolean, data: Int) {
        if (!rs) {
            HAL.clrBits(RS_BIT_MASK)
            waitTimeNano(RISE_DELAY)
        } else {
            HAL.setBits(RS_BIT_MASK)
            waitTimeNano(RISE_DELAY)
        }
        HAL.writeBits(DATA_MASK, data)
        pulse()
    }

    /**
     * Writes a command or data nibble (4 bits) to the LCD in serial mode.
     * @param rs Defines whether to send a command (false) or data (true)
     * @param data Data to send
     */
    private fun writeNibbleSerial(rs: Boolean, data: Int) {
        val datasend = if (rs) DATA_FRAME or data.shl(1) else CMD_FRAME or
data.shl(1)
        SerialEmitter.send(SerialEmitter.Destination.LCD, datasend)
    }


    /**
     * Writes a command or data nibble (4 bits) to the LCD.
     * @param rs Defines whether to send a command (false) or data (true)
     * @param data Data to send
     */
    private fun writeNibble(rs: Boolean, data: Int) {
```

```
        // if (HAL.isBit(SERIALMASK)) {
        //     writeNibbleSerial(rs, data)h
        // } else {
        //    writeNibbleParallel(rs, data)
        // }
        writeNibbleSerial(rs, data)
    }


    /**
     * Writes a command or data byte (8 bits) to the LCD by calling
[writeNibble] two times. One for the 4 high bits
     * and another for the 4 low bits.
     *
     * The 4 High bits must be sent before the 4 Low bits according to the LCD
specifications.
     *
     * @param rs Defines whether to send a command (false) or data (true)
     * @param data Data to send
     */
    private fun writeByte(rs: Boolean, data: Int) {
        val dataHigh = data.shr(4) // 4 high bits
        val dataLow = data.and(0b00001111) // 4 low bits
        writeNibble(rs, dataHigh)
        writeNibble(rs, dataLow)
    }


    /**
     * Writes a command to the LCD by calling [writeByte] with the RS flag set
to false.
     * @param data Command data to send
     */
    private fun writeCMD(data: Int) {
        writeByte(false, data)
        waitTimeMilli(CMD_DELAY)
    }


    /**
     * Writes data to the LCD by calling [writeByte] with the RS flag set to
true.
     * @param data Data to be sent
     */
    private fun writeDATA(data: Int) {
        writeByte(true, data)
    }


    /**
     * Initializes the LCD for 4 bit communication mode by following the
specified LCD initialization sequence.
     *
     * Uses the Function set, Display Control, and Entry Mode set commands.
     *
     * Calls the [writeNibble] and [writeCMD] functions.
     */
```

```
    fun init() {
        waitTimeMilli(INIT_DELAY)
        writeNibble(false, 0b00000011)
        waitTimeMilli(INIT_DELAY)
        writeNibble(false, 0b00000011)
        waitTimeMilli(INIT_DELAY)
        writeNibble(false, 0b00000011)
        waitTimeMilli(INIT_DELAY)
        writeNibble(false, 0b00000010)

        writeCMD(0b00101000) // function set: 4-bit mode, 2 lines, 5x8 dots
        writeCMD(0b00001000) // display control : display off, cursor off, blink
off
        writeCMD(0b00000001) // clear
        writeCMD(0b00000110) // entry mode set: increment cursor, no display
shift


        writeCMD(0b00001100) // display control: display on, cursor off, blink
off
    }


    /**
     * Writes a character to the LCD.
     */
    fun write(c: Char) = writeDATA(c.code)

    /**
     * Writes a string to the LCD.
     */
    fun write(text: String) {
        for (char in text) {
            write(char)
        }
    }

    /**
     * Positions the cursor on the LCD by using the Write data to CG or DDRAM
command.
     *
     * The 7th bit indicates the line position, and the 4 low bits indicate the
column position.
     *
     * @param line the line position for the cursor, 1 for the first line, 2 for
the second line
     * @param column the column position for the cursor, 1 for the first column,
16 for the last column
     */
    fun cursor(line: Int, column: Int) = writeCMD(0b10000000 or (((line - 1) shl
(6)) + column - 1))

    /**
     * Clears the LCD by using the Clear display command.
     *
     */
    fun clear() = writeCMD(0b00000001)
```

```
}
```

E. Código *Kotlin – SerialEmitter*

```
import SerialEmitter.Destination

fun main() {
    HAL.init()
    SerialEmitter.send(Destination.LCD, 0x15)
}


/**
 * 22/5/2023
 *
 * Serial emitter for sending data to the door controller or the LCD of the
[AccessControlSystem]
 * @property Destination Serial emitter destinations.
 * @author Bernardo Pereira
 * @author AntÃ³nio Paulino
 * @see LCD
 * @see DoorMechanism
 * @see HAL
 */
object SerialEmitter {

    enum class Destination(val mask: Int) { LCD(0b00000010), DOOR(0b00000100) }

    /**
     * The wait time in nanoseconds between sending bit signals to the hardware
     */
    private const val DELAY = 1000

    /**
     * The bit mask of the SDX output to the USB Port
     */
    private const val SDXMASK = 0b00000001

    /**
     * The bit mask of the SCLK output to the USB Port
     *
     */
    private const val SCLKMASK = 0b10000000

    /**
     * The bit mask of the BUSY input on the USB Port. Represents if the serial
receiver is busy.
     *
     */
    private const val BUSYMASK = 0b00100000

    /**
     * The size in bits of frames sent through the serial emitter.
     */

    private const val FRAME_SIZE = 5
```

```
    /**
     * Initializes the Door Mechanism Control and LCD Serial Receivers by
deselecting them. Sets [isbusy] to false.
     */
    fun init() {

        HAL.setBits(Destination.LCD.mask)
        HAL.setBits(Destination.DOOR.mask)
        HAL.clrBits(SCLKMASK)

        waitTimeNano(DELAY)
    }

    /**
     * Sends an SCLK pulse to the hardware through the USB Port output port
     */
    private fun clkPulse() {
        waitTimeNano(DELAY)

        HAL.setBits(SCLKMASK)
        waitTimeNano(DELAY)

        HAL.clrBits(SCLKMASK)
        waitTimeNano(DELAY)
    }

    /**
     * Sends data to the destination Serial Receiver according to the serial
communication protocol for the [AccessControlSystem]
     * @param addr The destination
     * @param data The data to send
     */
    fun send(addr: Destination, data: Int) {


        HAL.clrBits(addr.mask) //Select the destination Serial Receiver
        waitTimeNano(DELAY)

        for (i in 0 until FRAME_SIZE) { //Write frame to LCD
            val sdx = (1.shl(i) and data).shr(i)
            HAL.writeBits(SDXMASK, sdx)
            clkPulse()
        }

        HAL.setBits(addr.mask) //Deselect the destination Serial Receiver

        if (addr == Destination.DOOR) {
            while (!DoorMechanism.finished()) {
                waitTimeMilli(DELAY / 10)
            }
        }

        waitTimeNano(DELAY)
    }

    /**
     * Checks if the Door Serial Receiver is busy
```

```
         * @return true if the Door Serial Receiver is busy, false otherwise
         */
        fun isBusy(): Boolean = HAL.isBit(BUSYMASK)


}
```