1 Serial LCD Controller

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

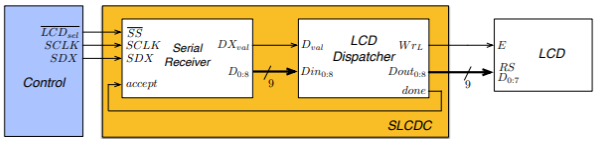


Figura 1 – Diagrama de blocos do módulo *SLCDC*

O SLCDC recebe uma mensagem constituída por nove (9) bits de informação e um (1) bit de paridade. A comunicação com este módulo realiza-se segundo o protocolo ilustrado na Figura 2, em que o bit RS é o primeiro bit de informação e indica se a mensagem é de controlo ou dados. Os seguintes oito (8) bits contêm os dados a entregar ao LCD. O último bit contém a informação de paridade par, utilizada para detetar erros de transmissão.

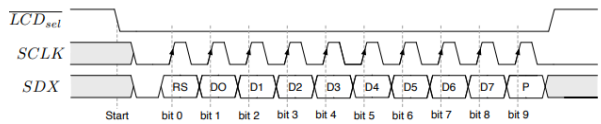


Figura 2 – Protocolo de comunicação com o módulo Serial LCD Controller

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

1.1 Serial Receiver

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

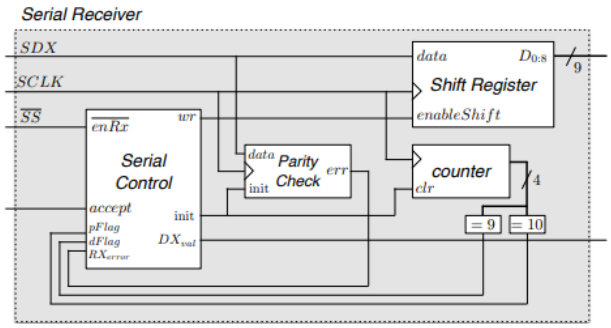


Figura 3 - Diagrama de blocos do bloco *Serial Receiver*

Na implementação deste bloco foram usados: um contador de 4 bits (*counter*), que realiza a contagem do número de bits que já foram recebidos, após quando receber um sinal de *clear* voltará a ter o valor lógico de saída “0000”; um controlador (*Serial_Control*), que será o responsável de verificar em que estado se encontra o *Serial Receiver* e fazer a passagem para outros estados, o diagrama do mesmo encontra-se representado na Figura 4; e um *Shifter* que aplica um *logical shift right* bit a bit (*Shift Register*), que irá guardar os valores dos bits até recebermos 9 bits no total pela lógica da máquina de estados.

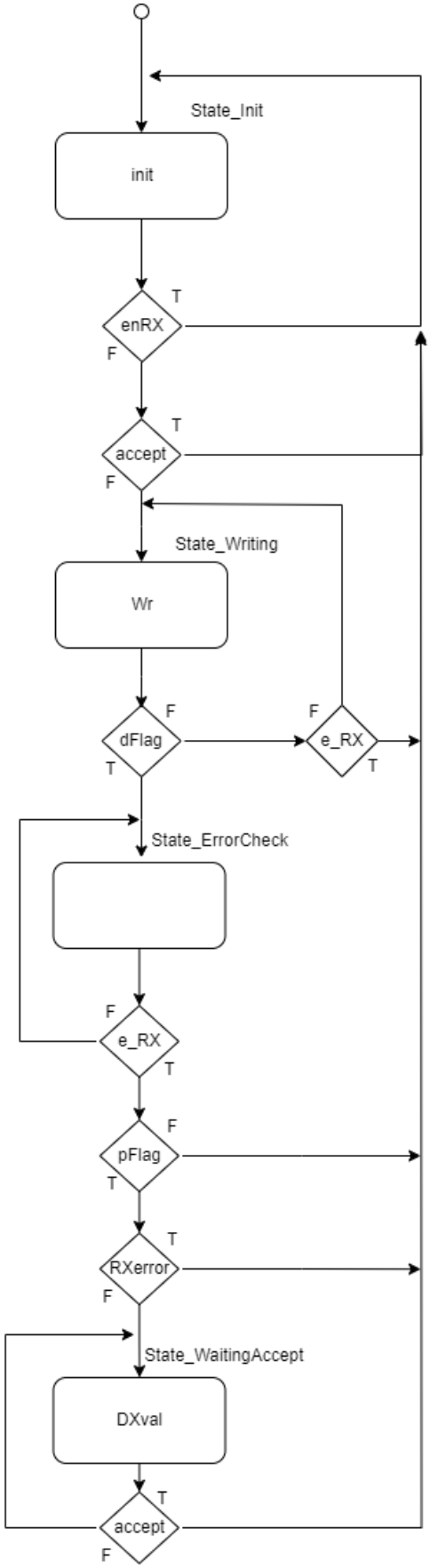


Figura 4 – Máquina de estados do bloco *Serial Control*

1.2 LCD Dispatcher

O bloco *LCD Dispatcher* é responsável pela entrega das tramas válidas recebidas pelo bloco *Serial Receiver* ao LCD, através da ativação do sinal *WrL* por pelo menos 230ns. A receção de uma trama válida é sinalizada pela ativação do sinal D_{val}. O processamento das tramas recebidas pelo LCD respeita os comandos definidos pelo fabricante, não sendo necessário esperar pela sua execução para libertar o canal de receção série. Assim, o bloco *LCD Dispatcher* pode ativar, prontamente, o sinal *done* para notificar o bloco *Serial Receiver* que a trama já foi processada.

A máquina de estados inicialmente encontra-se no estado *STATE_WAITING* onde não ativa nenhum bit de saída até esta receber o valor lógico '1' no D_{val}. Caso não receba este valor, ficará presa no estado referido. O próximo estado será o *STATE_ENABLE* onde permite a escrita de dados e ativa o sinal *WrL*, após alguns ciclos de relógio passa para o estado *STATE_DONE*. Neste estado é ativado o sinal *Done* para indicar que já acabou de efetuar a escrita. Quando o sinal D_{val} voltar a ter o valor lógico de '0' volta para o estado *STATE_WAITING* caso contrário fica neste estado.

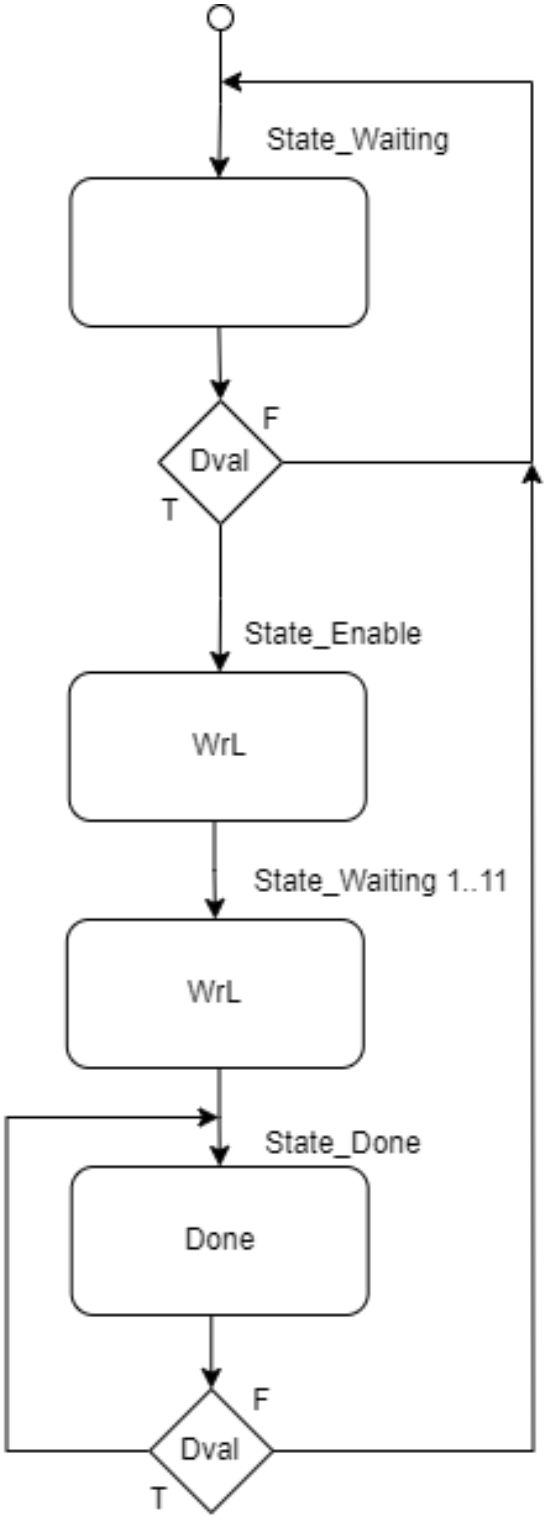


Figura 5 – Máquina de estados do LCD_Dispatcher

2 Interface com o *Control*

Implementou-se o módulo *Control* em *Software*, recorrendo a linguagem *Kotlin* e seguindo a arquitetura lógica apresentada na Figura 6.

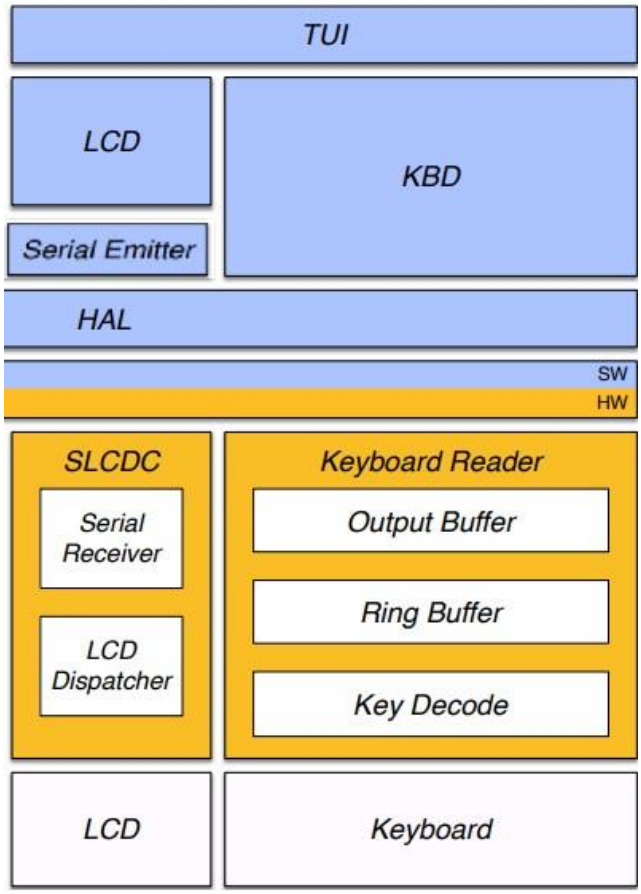


Figura 6 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader* e *SLCDC*

As classes *LCD* e *Serial Emitter* desenvolvidas são descritas nas secções 2.1 e .2, e o código fonte desenvolvido nos anexos “LCD.kt” e “Serial Emitter.kt”, respetivamente.

2.1 LCD

A função init() escreve a sequência de inicialização para comunicação. Mediante uma série de comandos e dados e inicializa o LCD para poder receber dados de escrita. Esta sequência foi seguida pela DataSheet do fabricante do LCD.

A função writeData(data: Int) recebe um parâmetro data que será escrita no LCD como dado, já a writeCMD(data: Int) envia um comando. A função writeByteSerial é responsável por enviar os dados, para o LCD, em série.

A função writeByte(rs: Boolean, data: Int) recebe dois valores; um "rs" que indicará se a mensagem é de controlo ou de dados e uma data equivalente a um byte

A função cursor(line: Int, column: Int) que envia um comando de modo a posicionar o cursor na respetiva linha (0 ou 1).

A função clear() que envia um comando de modo a apagar os valores escritos no LCD

A função writeChar(c: Char) em que se escreve recorremos à função writeData para escrever um *char* no LCD.

A função clear() envia um comando para limpar o ecrã e posicionar o cursor em (0,0). A função OFF desliga o LCD consoante um booleano de entrada como parâmetro. A função createCostumChar define um caracter especial na *CGRAM* do LCD, consoante um *array* de *bytes*, esta função é depois chamada para a escrita da nave e para a escrita dos *Invaders* nas funções drawShip(line: Int, column: Int) e drawInvader(line: Int, column: Int)

2.2 Serial Emitter

Este módulo tem como objetivo enviar tramas para os diferentes módulos Serial Receiver. Este módulo é utilizado para chegar aos mesmos objetivos que se chegaram no módulo LCD para o modo de envio de dados em paralelo.

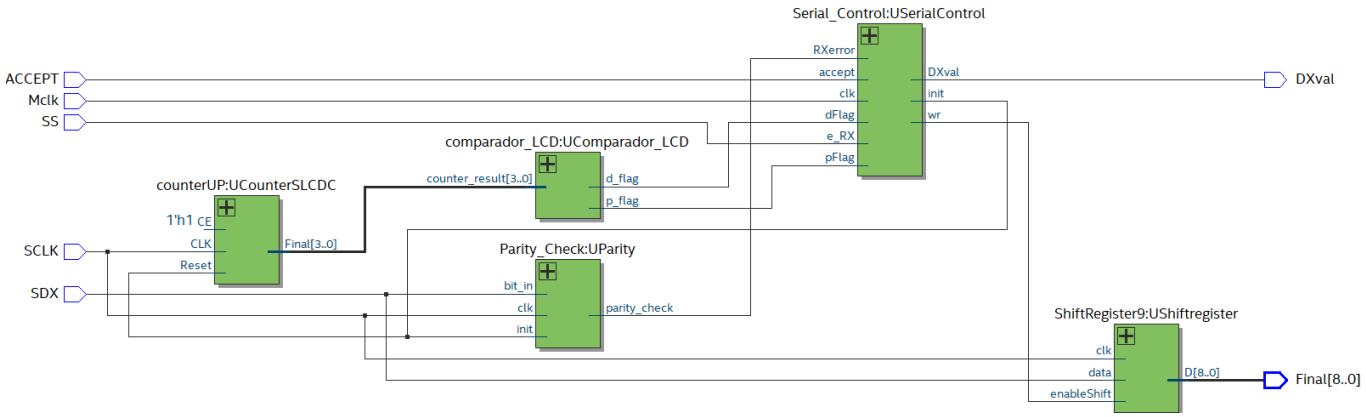
A função init() inicializa este módulo, inicializando o HAL, colocando o bit correspondente ao *SS* a '1' lógico e apaga os bits correspondentes ao *SCLK* e ao *SDX*.

A função send(addr: Destination, data: Int) envia uma trama para o Serial Receiver identificado o destino em *addr* e os bits de dados em 'data' de forma que o *Serial Receiver* possa receber os mesmos.

3 Conclusões

Para a realização deste módulo foram utilizados o Intellij para o *Software* e o *Quartus Prime Lite* para o *Hardware*. Também temos a apontar, que esta implementação, por ser feita em camadas, permitiu um desenvolvimento rápido e independente das várias partes do trabalho e preparou o mesmo para a segunda parte deste. Tivemos de colocar *delays* nas funções que manipulam bits, nomeadamente as que fazem parte do processo de envio de bits como por exemplo a SerialEmitter.send() ou então a função de inicialização do LCD que entre todos os comandos tem de ter um *delay* pois o *Hardware* não é tão rápido como o *Software*.

Descrição VHDL do bloco *Serial Receiver*



Anexo FA: Descrição do bloco Serial Receiver

```
entity Serial_Receiver_SLCDC is
    port(
        SDX: in std_logic; -- Entrada de dado em série
        Mclk: in std_logic; -- Entrada do Clock Secundário
        SCLK: in std_logic; -- Entrada do Clock Principal
        SS: in std_logic; -- Entrada do bit de seleção
        ACCEPT: in std_logic; -- Entrada do bit do sinal de aceitação
        DXval: out std_logic; -- Saída do bit de validação de dados
        Final: out std_logic_vector(8 downto 0) -- Saída dos dados
    );
end Serial_Receiver_SLCDC;

architecture Serial_Receiver_arch of Serial_Receiver_SLCDC is

    -- Declaração dos componentes utilizados no projeto

    component counterUP
        port(
            CE: in std_logic;
            Reset: in std_logic;
            CLK: in std_logic;
            Final: out std_logic_vector(3 downto 0)
        );
    end component;

    component Serial_Control
        port(
```

```
        clk: in std_logic;
        e_RX: in std_logic;
        accept: in std_logic;
        dFlag: in std_logic;
        pFlag: in std_logic;
        RXerror: in std_logic;
        wr: out std_logic;
        init: out std_logic;
        DXval: out std_logic
    );
end component;

component ShiftRegister9
    port (
        data: in std_logic;
        clk: in std_logic;
        enableShift: in std_logic;
        D: out std_logic_vector(8 downto 0)
    );
end component;

component Parity_Check
    port (
        clk: in std_logic;
        init: in std_logic;
        bit_in: in std_logic;
        parity_check: out std_logic
    );
end component;

component comparador_LCD
    port (
        counter_result: in std_logic_vector(3 downto 0);
        d_flag: out std_logic;
        p_flag: out std_logic
    );
end component;
```

```
    -- Sinais internos
    signal dFLAG_s, pFLAG_s, RXerror_s, wr_s, init_s: std_logic;
    signal CounterResult: std_logic_vector(3 downto 0);

begin

    -- Instância do componente comparador_LCD
    UComparador_LCD: comparador_LCD
        port map(
            counter_result => CounterResult,
            d_flag => dFLAG_s,
            p_flag => pFLAG_s
        );

    -- Instância do componente Serial_Control
    USerialControl: Serial_Control
        port map(
            e_RX => SS,
            accept => ACCEPT,
            clk => Mclk,
            dFlag => dFLAG_s,
            pFlag => pFLAG_s,
            RXerror => RXerror_s,
            wr => wr_s,
            init => init_s,
            DXval => DXval
        );

    -- Instância do componente Parity_Check
    UParity: Parity_Check
        port map(
            clk => SCLK,
            init => init_s,
            bit_in => SDX,
            parity_check => RXerror_s
        );

    -- Instância do componente counterUP
```

```
    UCounterSLCDC: counterUP
        port map(
            CE => '1',
            Reset => init_s,
            CLK => SCLK,
            Final => CounterResult
        );

    -- Instância do componente ShiftRegister9
    UShiftregister: ShiftRegister9
        port map(
            data => SDX,
            clk => SCLK,
            enableShift => wr_s,
            D => Final
        );
end Serial_Receiver_arch;
```

Descrição VHDL do bloco LCD Dispatcher

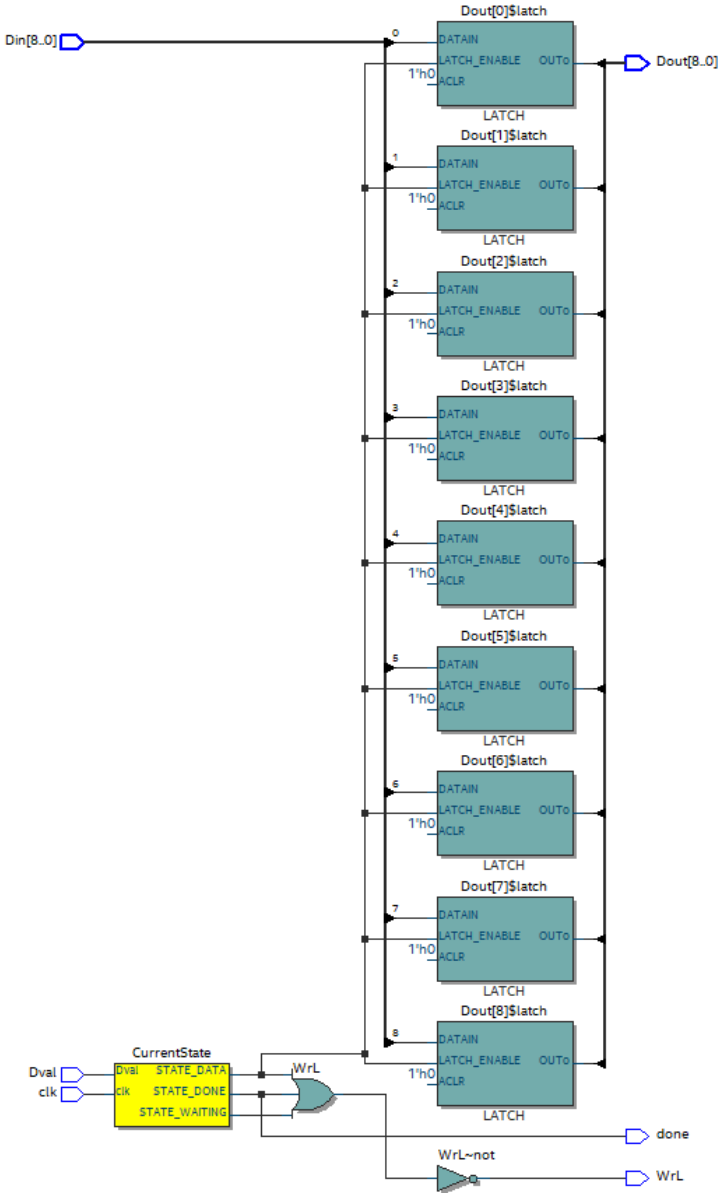


Figura 8: Diagrama de blocos do LCD_Dispatcher

```
entity LCD_Dispatcher is
   port(
      Din: in std_logic_vector(8 downto 0);
      Dval: in std_logic;
      clk: in std_logic;
      WrL: out std_logic;
      done: out std_logic;
      Dout: out std_logic_vector(8 downto 0)
   );
end LCD_Dispatcher;

architecture behavioral of LCD_Dispatcher is

   type STATE_TYPE is (STATE_WAITING,STATE_WAITING1, STATE_WAITING2, STATE_WAITING3,
STATE_WAITING4, STATE_WAITING5, STATE_WAITING6, STATE_WAITING7, STATE_WAITING8,
STATE_WAITING9, STATE_WAITING10, STATE_WAITING11, STATE_ENABLE, STATE_DONE);

   signal CurrentState, NextState : STATE_TYPE;

begin

CurrentState <= NextState when rising_edge (clk);

   process (CurrentState, Dval, Din)
   begin
      case CurrentState is
         when STATE_WAITING =>
            if Dval = '1' then
               NextState <= STATE_Enable;
            else
               NextState <= STATE_WAITING;
            end if;


         when STATE_ENABLE => NextState <= STATE_WAITING1;

          when STATE_WAITING1 =>
            NextState <= STATE_WAITING2;

             when STATE_WAITING2 =>
            NextState <= STATE_WAITING3;

             when STATE_WAITING3 =>
            NextState <= STATE_WAITING4;

          when STATE_WAITING4 =>
            NextState <= STATE_WAITING5;

             when STATE_WAITING5 =>
            NextState <= STATE_WAITING6;

         when STATE_WAITING6 =>
            NextState <= STATE_WAITING7;

                when STATE_WAITING7 =>
            NextState <= STATE_WAITING8;

                when STATE_WAITING8 =>
            NextState <= STATE_WAITING9;
```

```
                when STATE_WAITING9 =>
            NextState <= STATE_WAITING10;

                when STATE_WAITING10 =>
            NextState <= STATE_WAITING11;

                when STATE_WAITING11 =>
            NextState <= State_Done;


        when STATE_DONE =>
            if Dval = '0' then
                NextState <= STATE_WAITING;
            else
                NextState <= STATE_DONE;
            end if;
    end case;
end process;

Dout <= Din;
WrL <= '0' when (CurrentState = STATE_WAITING or CurrentState = STATE_DONE) else '1';
done <= '1' when (CurrentState = STATE_DONE) else '0';

end behavioral;
```

Descrição VHDL do bloco *SLCDC*

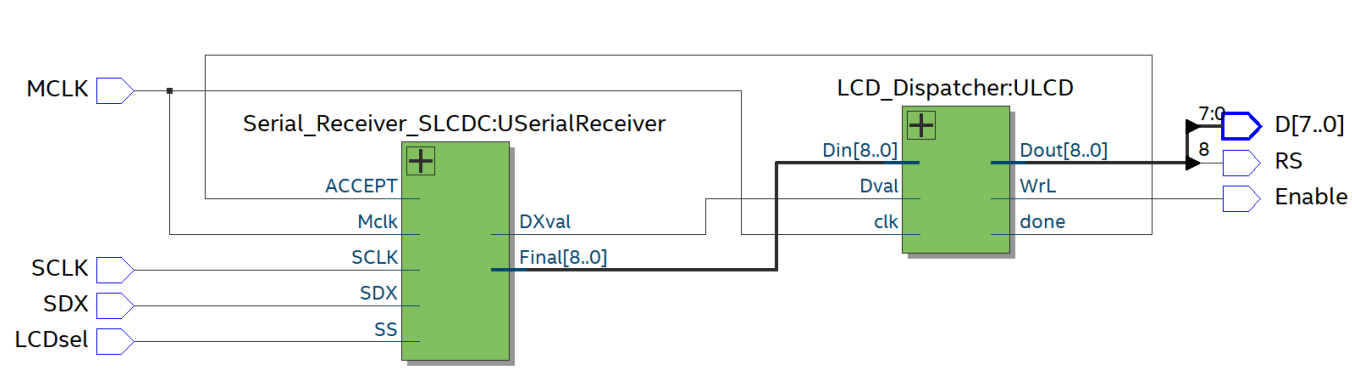


Figura 9: Diagrama de blocos do SLCDC

```
-- Entidade SLCDC que define a interface do componente SLCDC
entity SLCDC is
    port (
        SDX : in std_logic; -- Entrada de dados serial
        MCLK : in std_logic; -- Entrada do Clock secundário
        SCLK : in std_logic; -- Entrada do Clock Principal
        LCDsel : in std_logic; -- Seleção do LCD
        Enable : out std_logic; -- Saída do bit enable
        RS : out std_logic; -- Saída do bit RS (bit 8 dos dados)
        D : out std_logic_vector(7 downto 0) -- Saída dos Dados (bits 7 a 0 dos dados)
    );
end SLCDC;

architecture SLCDC_arch of SLCDC is

    -- Componente Serial_Receiver_SLCDC
    component Serial_Receiver_SLCDC
    port(
        SDX: in std_logic;
        Mclk : in std_logic;
        SCLK: in std_logic;
        SS: in std_logic;
        ACCEPT: in std_logic;
        DXval: out std_logic;
        Final: out std_logic_vector(8 downto 0)
        );
    end component;
```

```
    -- Componente LCD_Dispatcher
    component LCD_Dispatcher
    port(
        Din: in std_logic_vector(8 downto 0);
        Dval: in std_logic;
        clk: in std_logic;
        WrL: out std_logic;
        done: out std_logic;
        Dout: out std_logic_vector(8 downto 0)
        );
    end component;
    -- Sinais internos para conectar componentes
    signal Done_s : std_logic;
    signal DXval_s : std_logic;
    signal Final_s : std_logic_vector(8 downto 0);
begin
    -- Instância do componente Serial_Receiver_SLCDC
    USerialReceiver : Serial_Receiver_SLCDC port map (
        SDX => SDX,
        SCLK => SCLK,
        Mclk => MCLK,
        SS => LCDsel,
        ACCEPT => Done_s,
        DXval => DXval_s,
        Final => Final_s
    );
    -- Instância do componente LCD_Dispatcher
    ULCD : LCD_Dispatcher port map (
        Dval => DXval_s,
        CLK => MCLK,
        Din => Final_s,
        Wrl => Enable,
        Done => Done_s,
        Dout(8) => RS,
        Dout(7 downto 0) => D
    );
end SLCDC_arch;
```

Atribuição de pinos do módulo *SLCDC*

O módulo SLCDC tem interação tanto com o controlo como com o hardware, sendo assim a sua atribuição de pinos na placa "FPGA DE10-Lite" é somente dos sinais:

PIN_W8 -to LCD_RS
PIN_V5 -to LCD_EN
PIN_AA15 -to LCD_OUT [0]
PIN_W13 -to LCD_OUT[1]
PIN_AB13 -to LCD_OUT[2]
PIN_Y11 -to LCD_OUT[3]
PIN_W11 -to LCD_OUT[4]
PIN_AA10 -to LCD_OUT[5]
PIN_Y8 -to LCD_OUT[6]
PIN_Y7 -to LCD_OUT[7]

Código *Kotlin - LCD*

```
object LCD {
    private const val CLK_REG_MASK = 0x10 // É o bit (0001 0000)
    private const val E_MASK = 0x20 // É o bit 5(0010 0000)
    private const val RS_MASK = 0x40 // É o bit 6 (0100 0000)
    private const val ON_MASK = 0x0C // São os bits 2 e 3(1100)
    private const val OFF_MASK = 0x08 // É o bit 3 (1000)

    private val nave = byteArrayOf( //pixel art
        0b11110,
        0b11000,
        0b11100,
        0b11111,
        0b11100,
        0b11000,
        0b11110,
        0b00000
    )
    private val invader = byteArrayOf( //pixel art
        0b11111,
        0b11111,
        0b10101,
        0b11111,
        0b11111,
        0b10001,
        0b10001,
        0b00000
    )

    // Escreve um byte de comando/dados no LCD em paralelo
    private fun writeByteParallel(rs: Boolean, data: Int) {
        if (rs) {
            HAL.setBits(RS_MASK)
        } else {
            HAL.clrBits(RS_MASK)
        }
        val shift_right = data.shr(4)

        HAL.writeBits(15, shift_right)
        HAL.setBits(CLK_REG_MASK)
        HAL.clrBits(CLK_REG_MASK)

        HAL.writeBits(15, data)
        HAL.setBits(CLK_REG_MASK)
        HAL.clrBits(CLK_REG_MASK)
        HAL.setBits(E_MASK)
        HAL.clrBits(E_MASK)
    }

    // Escreve um byte de comando/dados no LCD em série
    private fun writeByteSerial(rs: Boolean, data: Int) {
        var not_data = data
        if (rs) {
            not_data = data.shl(1) + 1
        } else {
            not_data = data.shl(1)
        }

        SerialEmitter.send(SerialEmitter.Destination.LCD, not_data, 10)
    }

    // 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)
    }

    // Escreve um caráter na posição corrente.
    fun writeChar(c: Char) {
        writeDATA(c.code)
    }

    // Escreve uma string na posição corrente.
    fun writeString(text: String) {
        for (i in text) {
            writeChar(i)
        }
    }

    // Envia comando para posicionar cursor (‘line’:0..LINES-1 , ‘column’:0..COLS-1)
    fun cursor(line: Int, column: Int) {
        val writer = 128
        writeCMD((line * 0x40) + column + writer)
    }

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

    // Função usada para definir um caracter especial e guarda-lo na CGRAM
    fun createCustomChar(location: Int, charmap: ByteArray) {
        writeCMD(0x40 or (location shl 3)) // Definir o endereço da CGRAM (3 shifts para multiplica-lo
por 8)

        for (i in charmap.indices) {
            writeDATA(charmap[i].toInt()) //envio de dados sobre o caracter especial
        }
    }

    // Desliga o LCD quando off = true
    fun off(off:Boolean){
        if (off){
            writeCMD(OFF_MASK)
        }
        else
            writeCMD(ON_MASK)
    }

    // Desenha a nave no LCD
    fun drawShip(line: Int, column: Int) {

        createCustomChar(0, nave)
        cursor(line, column) //Aplica o cursor na linha e coluna de input
        writeDATA(0) // Display do caracter guardado na CGram na posição 0
    }
    fun drawInvader(line:Int, column: Int){

        createCustomChar(1, invader)
```

```
        cursor(line, column) //Aplica o cursor na linha e coluna de input
        writeDATA(1) // Display do caracter guardado na CGram na posição 1
    }

    fun init() {
        Thread.sleep(20)

        writeCMD(48)
        Thread.sleep(5)

        writeCMD(48)
        Thread.sleep(1)

        writeCMD(48)
        writeCMD(56)
        writeCMD(8)
        writeCMD(1)
        writeCMD(6)
        writeCMD(15)
    }
}
```

Código *Kotlin – Serial Emitter*

```
object SerialEmitter {
    private const val LCD_MASK = 0x01 //É o bit 0 (0001)
    private const val SSC_MASK = 0x02 //É o bit 1 (0010)
    private const val SDX_MASK = 0x08 //É o bit 3 (1000)
    private const val CLK_REG_MASK = 0x10 //É o bit 4 (0001 0000)
    private const val CLEAR_MASK = 0x1F // É os bits todos antes do 32(5) ou seja (0001 1111)

    // Envia tramas para os diferentes módulos Serial Receiver.
    enum class Destination { LCD, SCORE }

    // Inicia a classe
    fun init() {
        HAL.clrBits(CLEAR_MASK)
        HAL.setBits(LCD_MASK)
        HAL.setBits(SSC_MASK)
    }

    //  if LCD size = 10
    //  if Score size = 8

    // 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.
    // Vai primeiro o cmd e depois a data
    fun send(addr: Destination, data: Int, size: Int) {
        var dataShifted = data
        var ParityBit = 0
        var counter = 0

        if (addr == Destination.LCD) {
            HAL.clrBits(LCD_MASK) //Entra negado no hardware
        } else if (addr == Destination.SCORE) {
            HAL.clrBits(SSC_MASK) //Entra negado no hardware
        }


        dataShifted = data.shl(3) //Mover a data 3 vezes pois enviamos pelo bit do SDX
        for (i in 0 until size - 1) {
            HAL.clrBits(CLK_REG_MASK) //Efetuar o clock
            HAL.writeBits(SDX_MASK, dataShifted) //Escrever o valor de dataShifted que está no
valor de SDX_MASK
            if (dataShifted.shr(3) % 2 == 1) {
                counter++
            }
            dataShifted = dataShifted.shr(1) //Mover a data um bit para escrevermos o bit seguinte
            HAL.setBits(CLK_REG_MASK) //Finalizar o clock
        }

        if (counter % 2 == 1) {
            ParityBit = 1 //Se o resto de divisão do counter
        }
        HAL.clrBits(CLK_REG_MASK) //Iniciar o clock
        HAL.writeBits(SDX_MASK, ParityBit.shl(3)) //Escrever o bit de paridade e o shift é para
estar em SDX
        HAL.setBits(CLK_REG_MASK) //Finalizar o clock
```

```
        HAL.setBits(LCD_MASK) //Colocar o bit do LCD
        HAL.setBits(SSC_MASK) //Colocar o bit do SSC
    }
}
```