

O m dulo *Serial LCD Controller*   constitu do por dois blocos principais: i) bloco que recebe a informa  o em s rie enviada pelo m dulo de controlo (*Serial Receiver*); ii) O bloco *Dispatcher*, que entrega a trama recebida pelo *Serial Receiver* ao LCD (designado por *LCD Dispatcher*). Neste caso o m dulo *Control*, implementado em *software*,   a entidade Emissora enquanto que o LCD   a entidade consumidora.

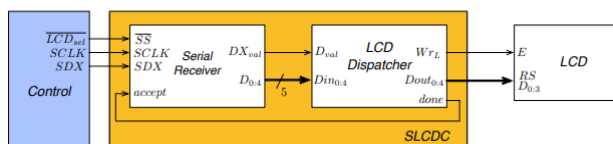
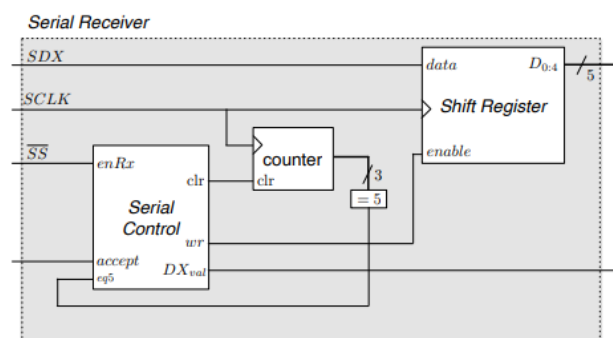


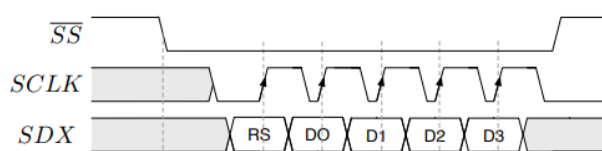
Figura 1 – Diagrama de blocos do m dulo Serial LCD Controller

1 Serial Receiver

O bloco *Serial Receiver* implementa um decodificador de um teclado matricial 4x3 atrav s de *hardware*, sendo constitu do por tr s sub-blocos: i) bloco respons vel por deslocar um bit para a esquerda (*Shift Register*); ii) o bloco *Counter*, respons vel por, a cada impulso clock, somar 1 ao valor anteriormente guardado; e iii) o bloco *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 sa da do bloco *Serial Receiver* (para o m dulo *LCD Dispatcher*), define que o sinal DX_{val}   ativado quando $enRx$ est  ativo e se todos os bits (5 bits) foram recebidos (a partir do eq5 estar com o valor 1), sendo tamb m disponibilizado o c digo dessa tecla no barramento $K 0:4$. Apenas   iniciado um novo ciclo de varrimento do teclado quando o sinal *accept* for ativado e posteriormente desativado. O diagrama temporal do controlo de fluxo est  representado na Figura 2b.



a) Diagrama de blocos



b) Diagrama temporal

Figura 2 – Comunica  o com o *Serial LCD Controller*

2 Serial Control

O bloco *Serial Control*, tal como o nome indica, controla a leitura da trama recebida pelo *Serial Receiver*. O bloco aguarda uma transi  o descendente do sinal not SS que est  dependente da comunica  o com o *Serial LCD Controller* e quando essa transi  o acontece, o bloco *Serial Control*, atrav s de liga  es com os blocos *counter* e *Shift Register*, l  bit a bit a trama recebida. No final, caso todos tenham sido lidos com sucesso, o bloco "avisa" o bloco *LCD Dispatcher* que foi transmitida uma trama v lida. Este processo foi explicado e implementado com base na seguinte m quina de estados:

A m quina de estados do bloco *Serial Control* tem como primeiro estado o estado *Clear*.

O sinal de sa da "clear" est  ativado para termos a certeza que o "counter" est  a zero.

Neste estado aguarda-se que o sinal $enRx$ passe para o valor l gico "false" para haver uma passagem para o estado seguinte, caso contr rio a m quina de estados mant m-se neste estado (ap s esta transi  o do sinal $enRx$ para "false"   que o *SLCDC* armazena os bits da trama nas transi  es ascendentes do sinal *SCLK*).

No segundo estado, *While*, sempre que a confirma  o do sinal $enRx$ seja "false" o sinal de sa da *wr* ativa (o que depois, tendo em conta o resto do *Serial Receiver*, vai "permitir ir trocando de bit que queremos ler") e mant m-se o segundo estado.

Caso o $enRx$ esteja "true", verifica-se o valor l gico do sinal $eq5$. Tendo o $enRx$ "true" e o $eq5$ "false" a m quina de estados volta ao primeiro estado pois significa que houve um erro que n o permitiu ler os 5 bits e como tal, o processo tem de ser reiniciado.

Se $enRx$ e $eq5$ estiverem "true", significa que a leitura de todos os bits da trama foi bem sucedida e podemos assim passar ao terceiro estado.

No nosso terceiro estado, *Value*, a sa da DX_{val} encontra-se a "true" pois todos os bits foram lidos com sucesso (sendo assim uma trama v lida) e aguarda-se neste estado que o sinal *accept* esteja a "true" para confirmar que a trama foi processada pelo bloco *Dispatcher* e para a m quina de estados poder avan ar para o quarto e  ltimo estado.

No  ltimo estado, *Wait*, a m quina de estados espera que o sinal de entrada encontre-se realmente a "false" e, caso isso aconte a, passa assim para o estado inicial. Optou-se pela cria  o deste estado para ter completa certeza que a m quina de estados encontrava-se realmente preparada para receber nova trama.

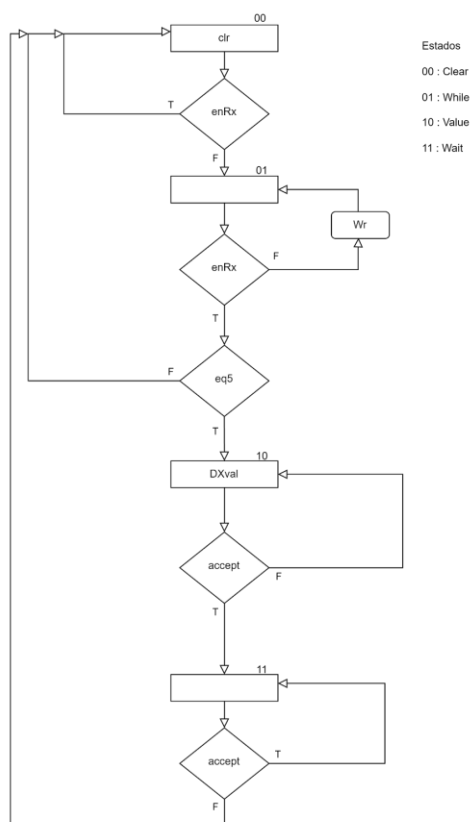


Figura 3 - M quina de estados Serial Control

3 Dispatcher

O bloco *Dispatcher*   respons vel pela entrega das tramas recebidas pelo bloco *Serial Receiver* ao LCD. Sabe-se que uma trama   recebida quando o sinal *Dval*   ativado.

O *Dispatcher*   iniciado com um estado vazio (00) (uma vez que n o existe output – neste caso,   necess rio esperar por uma condi o). O *Dval* quando fica com o valor l gico ‘1’ vamos para o estado waiting (01), caso contr rio permanece no estado inicial.

Entre o estado Waiting e o estado Done, existe um tempo de espera (clkdiv) para os dados serem processados, durante esse tempo o *Wrl* fica com o valor l gico ‘1’. Foi preciso recorrer   utiliza o de um clkdiv visto que h  um maior tempo para processar os dados.

Depois do processamento dos dados, a m quina de estados fica no estado DONE (10), em que enquanto o *Dval* n o fica com o valor l gico ‘0’ iremos permanecer neste estado em que o *done* fica com o valor l gico ‘1’, saindo apenas quando o *Dval* ficar com o valor l gico ‘0’

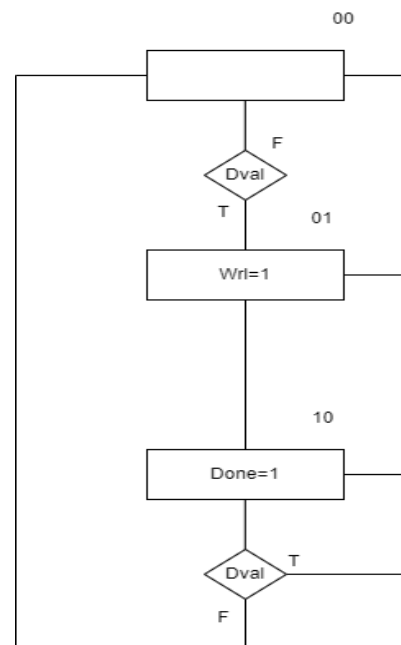


Figura 4 – M quina de estados Dispatcher

Com base nas descri es dos blocos *Serial Receiver* e do *LCD Dispatcher* implementou-se o m dulo *SLCDC* de acordo com o esquema el trico representado no anexo C.

Para o m dulo *SLCDC* foi usado a mesma frequ ncia de rel gio que se usou para o *Keyboard Reader*, e tamb m, foi utilizado um clock div para reduzir a frequ ncia de rel gio do m dulo *LCD Dispatcher*. A partir do clock div, dividiu-se a frequ ncia de rel gio do master clock (que neste caso   a frequ ncia da placa, 50MHz) por 1000 hz, para assim o *LCD Dispatcher* ter tempo suficiente para processar as tramas recebidas.

4 Interface com o Control

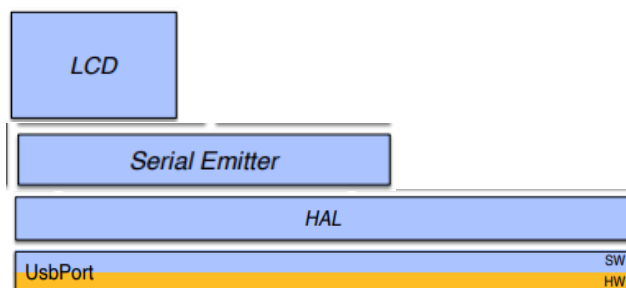


Figura 5 - interface LCD com Control

4.1 LCD

O LCD é a camada de software responsável por apresentar os dados no ecrã do LCD, e tem uma interface de 4 bits (nibble). Este modulo tem as funcoes `writeNibbleParallel` e `writeNibbleSerial`, responsáveis por escrever um nibble de dados ou comandos no LCD e é no `writeNibble()` onde se decide se a informação é transmitida em paralelo ou em série. Também é responsável pelos comandos de limpar o ecrã (`clear()`) e posicionar o cursor em uma posição desejada (`cursor()`), relativamente ao data usamos as funções `write` para escrever uma string, ou escrever um char.

5 Conclusão

Para concluir, podemos salientar que o módulo LCD Dispatcher podia ter sido otimizado no sentido em que, em vez de dividir a frequência de relógio, podia ser implementado um contador para aguardar um determinado número de `clks` e não ir de imediato do estado 01 para o estado 10. A frequência de relógio dividida no `clkdiv` podia ser um valor menor , mas não deu para ser testado na placa.

A. Descri o VHDL do bloco *Serial Receiver*

```
LIBRARY ieee;
USE ieee.STD_LOGIC_1164.ALL;

entity serial_receiver is port (
    SDX : in std_logic;
    SCLK : in std_logic;
    MCLK : in std_logic;
    reset : in std_logic;
    not_SS : in std_logic;
    accept : in std_logic;
    D : out std_logic_vector (4 downto 0);
    DXval : out std_logic;
    busy : out std_logic
);

end serial_receiver;

architecture arq of serial_receiver is

    component shift_register is port (
        data : in std_logic;
        clk : in std_logic;
        enable : in std_logic;
        D : out std_logic_vector (4 downto 0);
        reset : in std_logic
    );
    end component;

    component counter is port (
        reset : in std_logic;
        ce : in std_logic;
        clk : in std_logic;
        Q : out std_logic_vector (3 downto 0)
    );
    end component;

    component serial_control is port (
        enRx : in std_logic;
        accept : in std_logic;
        eq5 : in std_logic;
        clr : out std_logic;
        wr : out std_logic;
        DXval : out std_logic;
        reset : in std_logic;
        clk : in std_logic
    );
    end component;

    signal s_wr, s_clr, c5_s : std_logic;
    signal d_s : std_logic_vector (3 downto 0);

begin
    u_counter : counter port map (
        clk => SCLK,
        ce => '1',
        reset => s_clr,
        Q(3) => d_s(3),
        Q(2) => d_s(2),
        Q(1) => d_s(1),
        Q(0) => d_s(0)
    );

    u_shift_register: shift_register port map(
        data => SDX,
        clk => SCLK,
        enable => s_wr,
        D => D,
        reset => reset
    );

    u_serial_control: serial_control port map (
        enRx => not_SS,
        accept => accept,
        clr => s_clr,
        wr => s_wr,
        eq5 => c5_s,
        DXval => DXval,
        clk => MCLK,
        reset => reset
    );

    c5_s <= (d_s(0) and not d_s(1) and d_s(2));

end arq;
```

B. Descrição VHDL do bloco *Dispatcher*

```

LIBRARY ieee;
USE ieee.STD_LOGIC_1164.ALL;

entity dispatcher is port (
    dval : in std_logic;
    din : in std_logic_vector (4 downto 0);
    wr1 : out std_logic;
    dout : out std_logic_vector(4 downto 0);
    done : out std_logic;
    clk : in std_logic;
    reset : in std_logic
);
end dispatcher;

architecture arq of dispatcher is

    type STATE_TYPE is (STATE_INITIAL, STATE_WAITING, STATE_DONE);

    signal currentState, nextState: STATE_TYPE;

begin
    currentState <= STATE_INITIAL when reset = '1' else nextState when rising_edge(clk);

    generatenextState:
    process(currentState, dval)
    begin
        case currentState is
            when STATE_INITIAL => if (dval = '1') then
                                    nextState <= STATE_WAITING;
                                else
                                    nextState <= STATE_INITIAL;
                                end if;

            when STATE_WAITING => nextState <= STATE_DONE;

            when STATE_DONE => if (dval = '0') then
                                    nextState <= STATE_INITIAL;
                                else
                                    nextState <= STATE_DONE;
                                end if;

        end case;
    end process;

    -- generate outputs
    wr1 <= '1' when (currentState = STATE_WAITING) else '0';
    done <= '1' when (currentState = STATE_DONE) else '0';

    dout <= din;
end arq;

```

C. Descrição VHDL do bloco *SLCDC*

```

LIBRARY ieee;
USE ieee.STD_LOGIC_1164.ALL;

entity SLCDC is port (
    SDX : in std_logic;
    SCLK : in std_logic;
    MCLK : in std_logic;
    reset : in std_logic;
    not_SS : in std_logic;
    E : out std_logic;
    D : out std_logic_vector (3 downto 0);
    RS : out std_logic
);
end SLCDC;

architecture arq of SLCDC is

    component clkdiv is generic (div : natural := 25);
    port (
        clk_in : in std_logic;
        clk_out : out std_logic
    );
    end component;

    component serial_receiver is port (
        SDX : in std_logic;
        SCLK : in std_logic;
        MCLK : in std_logic;
        reset : in std_logic;
        not_SS : in std_logic;
        accept : in std_logic;
        D : out std_logic_vector (4 downto 0);
        DXval : out std_logic;
        busy : out std_logic
    );
    end component;

    component dispatcher is port (
        dval : in std_logic;
        din : in std_logic_vector (4 downto 0);
        wr1 : out std_logic;
        dout : out std_logic_vector (4 downto 0);
        done : out std_logic;
        clk : in std_logic;
        reset : in std_logic
    );
    end component;

    signal done_s, dval_s : std_logic;
    signal d_s : std_logic_vector (3 downto 0);
    signal din_s : std_logic_vector (4 downto 0);
    signal clk : std_logic;

begin
    u_serial_receiver: serial_receiver port map(
        SDX => SDX,
        SCLK => SCLK,
        MCLK => MCLK,
        reset => reset,
        not_SS => not_SS,
        accept => done_s,
        D => din_s,
        DXval => dval_s,
        busy => open
    );

    u_dispatcher: dispatcher port map (
        dval => dval_s,
        din => din_s,
        wr1 => E,
        dout(0) => RS,
        dout(4) => D(3),
        dout(3) => D(2),
        dout(2) => D(1),
        dout(1) => D(0),
        done => done_s,
        clk => clk,
        reset => reset
    );

    u_clkdiv: clkdiv generic map (1000) port map (
        clk_in => MCLK,
        clk_out => clk
    );
end arq;

```

D. Atribuição de pinos do SLCDC

set_location_assignment PIN_P11 -to Mclk

set_location_assignment PIN_W5 -to Lines[0]
set_location_assignment PIN_AA14 -to Lines[1]
set_location_assignment PIN_W12 -to Lines[2]
set_location_assignment PIN_AB12 -to Lines[3]
set_location_assignment PIN_AB11 -to Columns[0]
set_location_assignment PIN_AB10 -to Columns[1]
set_location_assignment PIN_AA9 -to Columns[2]

set_location_assignment PIN_A8 -to Kval
set_location_assignment PIN_A9 -to K[0]
set_location_assignment PIN_A10 -to K[1]
set_location_assignment PIN_B10 -to K[2]
set_location_assignment PIN_D13 -to K[3]

set_location_assignment PIN_C10 -to reset
set_location_assignment PIN_C11 -to Kack

set_location_assignment PIN_W8 -to rs
set_location_assignment PIN_V5 -to en

set_location_assignment PIN_W11 -to data[4]
set_location_assignment PIN_AA10 -to data[5]
set_location_assignment PIN_Y8 -to data[6]
set_location_assignment PIN_Y7 -to data[7]

E. Código Kotlin - LCD

```
import isel.leic.utils.Time
object LCD { // Escreve no LCD usando a interface a 4 bits.
    //private const val LINES = 2 // Dimensão do display.
    //private const val COLS = 16 // Dimensão do display.
    private const val LCD_RS = 0x20
    private const val LCD_ENABLE = 0x40 // 0b0100_0000
    private const val LCD_DATA = 0x1E
    private const val CLEAR_DISPLAY = 0x01
    private var state = false
    private const val CMD_DISPLAY_LENGTH = 0x28 //0b0000_1100
    private const val CMD_DISPLAY_ENTRY_MODE = 0x06 // 0b0000_0110
    private const val CMD_DISPLAY_OFF = 0x08 // 0b0000_1000
    private const val CMD_DISPLAY_ON = 0x0F // 0b0000_1111
    private const val CMD_DISPLAY_CLEAR = 0x01 //0b0000_0001

    // Escreve um nibble de comando/dados no LCD em paralelo
    private fun writeNibbleParallel(rs: Boolean, data: Int) {
        HAL.writeBits(LCD_DATA, data shl 1)
        if (rs) HAL.setBits(LCD_RS) else HAL.clrBits(LCD_RS)
        Time.sleep(1)
        HAL.setBits(LCD_ENABLE)
        Time.sleep(1)
        HAL.clrBits(LCD_ENABLE)
        Time.sleep(1)
    }

    // Escreve um nibble de comando/dados no LCD em série
    // rs: false -> comando; true -> dado
    private fun writeNibbleSerial(rs: Boolean, data: Int) {
        val rsToInt = if (rs) 1 else 0
        val newData = data.shl(1) or rsToInt
        SerialEmitter.send(SerialEmitter.Destination.LCD,newData)
    }

    // Escreve um nibble de comando/dados no LCD
    private fun writeNibble(rs: Boolean, data: Int) {
        if (state) writeNibbleParallel(rs,data) else writeNibbleSerial(rs,data)
    }

    // Escreve um byte de comando/dados no LCD
    private fun writeByte(rs: Boolean, data: Int) {
        writeNibble(rs,data shr 4)
        writeNibble(rs,data and 0x0F)
        Time.sleep(2)
    }

    // 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 sequ ncia de in cia o para comunica o a 4 bits.
fun init() {
    Time.sleep(15)
    writeNibble(false, 0x03)
    Time.sleep(5)
    writeNibble(false, 0x03)
    Time.sleep(1)
    writeNibble(false, 0x03)
    writeNibble(false, 0x02)
    writeCMD(CMD_DISPLAY_LENGTH)
    writeCMD(CMD_DISPLAY_OFF)
    writeCMD(CMD_DISPLAY_CLEAR)
    writeCMD(CMD_DISPLAY_ENTRY_MODE)
    writeCMD(CMD_DISPLAY_ON)
}

fun write(c: Char) {
    writeDATA(c.code)
}

fun write(text: String) {
    for (c in text) {
        write(c)
    }
}

// Envia comando para posicionar cursor ('line':0..LINES-1 , 'column':0..COLS-1)
fun cursor(lin: Int, col: Int) {
    writeCMD((lin * 0x40 + col) or 0x80) // colocar bit de maior peso a 1
}

// Envia comando para limpar o ecr  e posicionar o cursor em (0,0)
fun clear() {
    writeCMD(CLEAR_DISPLAY)
}

fun main() {
    HAL.init()
    LCD.init()
    LCD.cursor(0,0)
    LCD.write("hey word")
    Time.sleep(5000)
    LCD.clear()
    Time.sleep(200)
    while (true) {
        LCD.cursor(0,0)
        LCD.write("0123456789ABCDEF")
        LCD.cursor(1,0)
        LCD.write("0123456789ABCDEF")
    }
}
```