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

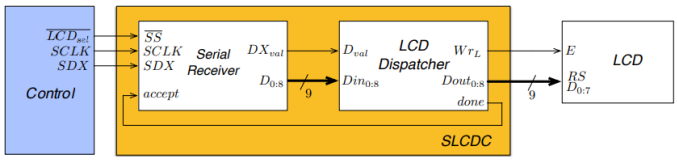

Figura 1 – Diagrama de blocos do módulo *Serial LCD Controller*

O módulo *SLCDC* recebe em série 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.

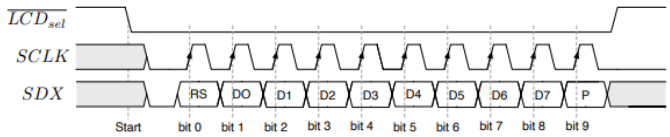

Figura 2 – Protocolo de comunicação com o módulo *Serial LCD Controller*

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 $\overline{LCD_{set}}$. Após a condição de início, o módulo *SLCDC* armazena os bits de dados da trama nas transições ascendentes do sinal *SCLK*.

A descrição *VHDL* do bloco *Serial LCD Controller* encontra-se no Anexo A.

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.

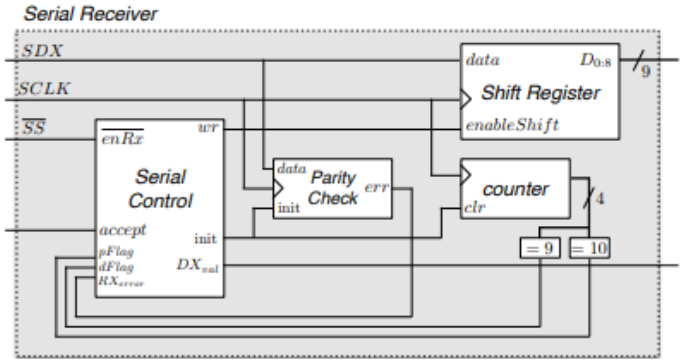

Figura 3 – Diagrama de blocos do bloco *Serial Receiver*

O bloco *Serial Receiver* foi implementado de acordo com o diagrama de blocos representados na figura 3. Este bloco tem como sinais de entrada *SDX*, *SCLK* e *enRX* e como saída tem *DXval* e *D0:8*. A descrição *VHDL* do bloco *Serial Receiver* encontra-se no Anexo B.

Para uma abordagem inicial, o nosso *Serial Control* ainda não terá o sinal *enRX* no modo *active low*, devido a ter sido mais pratico para nós. Futuramente iremos alterá-lo para que corresponda com o pedido. Na figura 4 poemos observar o *ASM-chart* do *Serial Control*.

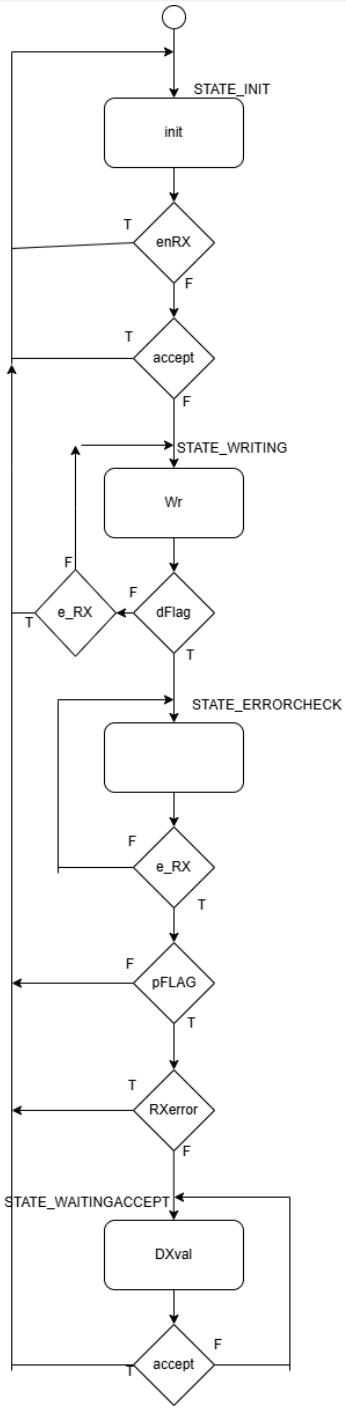

Figura 4 – *ASM-chart* do bloco *Serial Control*

No primeiro estado do *Serial Control* (*STATE_INIT*) ele ira dar clear ao controlo (*init* = '1') e ira verificar se enRX esta ativo ou não. Uma vez que *enRX* esteja ativo ele ira também verificar o sinal accept. Como accept tem por agora que estar a false pois ainda não passamos a trama toda ele ira ser "0" e iremos para o estado seguinte. No segundo estado iremos dar enable do wr, ou seja, iremos começar a escrever a trama que o SDX ira enviar. Para isso iremos estar num ciclo até que dFlag esteja a '1'. Assim que *dFlag* estiver ativo iremos para o estado seguinte em que iremos verificar o sinal pFlag, em que enquanto não passarmos o bit de paridade, ele estará sempre nesse estado. Assim que recebermos o bit de paridade, iremos para o estado seguinte em que iremos ativar DxVal e iremos fazer uma verificação do sinal accept. Assim que accept estiver ativo iremos voltar para o estado inicial. A descrição hardware em *VHDL* do bloco *Serial Control* encontra-se no Anexo C.

O bloco *Parity Check* tem como objetivo validar a paridade da trama enviada. Esta ira dar *err* (sinal de erro) sempre que a soma dos bits enviados por *SDX* mais o bit de paridade seja ímpar. A descrição *VHDL* do bloco *Serial Receiver* encontra-se no Anexo D.

O bloco Shift Register tem como objetivo receber a trama enviada por SDX em serie (bit a bit). A descrição hardware do bloco Serial Receiver em VHDL encontra-se no Anexo E.

O bloco *Counter* tem como objetivo incrementar sempre que se está a escrever no *Shift Register*. Ele terá como entradas o *SCLK*, proveniente do *Control*, e o *clr*, proveniente do Serial *Control*, que terá a função de limpar o contador. Neste contador, iremos avaliar as saídas utilizando dois comparadores. Um irá comparar a saída do *counter* com o valor 9 e o outro irá comparar com o valor 10. A descrição em hardware dos blocos *Counter*, *Compare9bits* e *Compare10bits* em *VHDL* encontra-se no Anexo F, G e H.

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*. A receção de uma trama válida é sinalizada pela ativação do sinal *Dval*. 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.

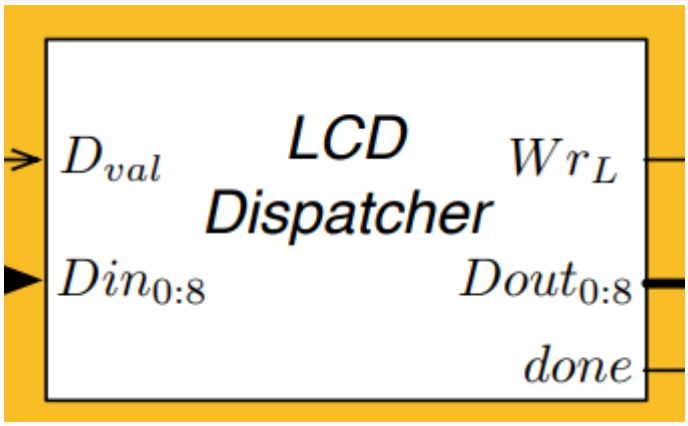

Figura 5 – Bloco *LCD Dispatcher*

A máquina de estados do bloco LCD Dispatcher foi feita foi da seguinte forma: primeiramente ele terá de detetar *Dval*, depois ele ira ler ou registar *Din*, colocar *Dout* na saída, fazer *clock* do sinal *WrL*, ativar done, esperar que *Dval* esteja a 0 e tirar *done* ao verificar *Dval*. Na figura 6 podemos observar o *ASM-chart* do *LCD Dispatcher*.

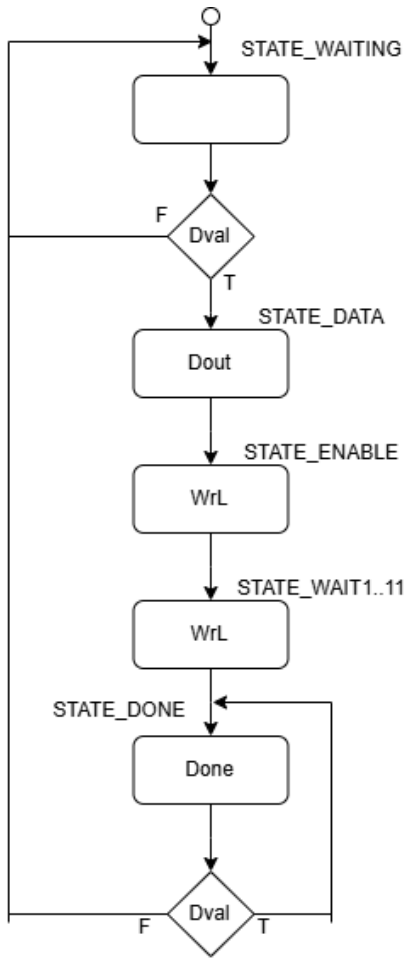

Figura 6 - *ASM-chart* do bloco *LCD Dispatcher*

3 Interface com o *Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem Java e seguindo a arquitetura lógica apresentada na Figura 7.

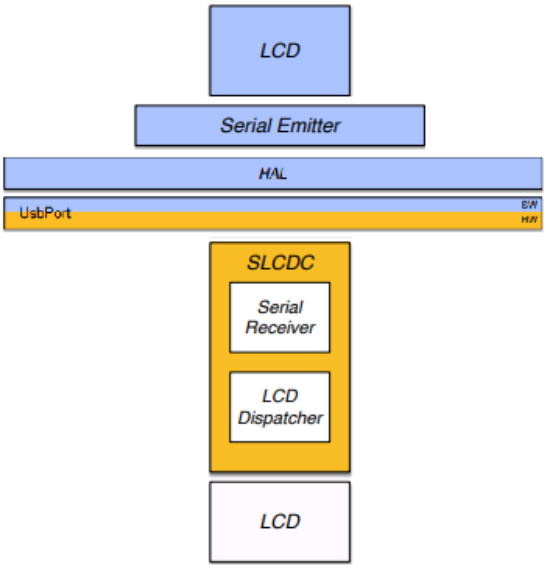

Figura 7– Diagrama lógico do módulo *Control* de interface com o módulo *Serial LCD Controller*

As classes LCD *e Serial Emiter* desenvolvidas são descritas nas secções 3.1 e 3.2, e o código fonte desenvolvido nos Anexos I e II, respetivamente.

3.1 Classe LCD

Este Módulo tem como objetivo escrever no LCD usando a interface a 4 bits.

init() – Inicializa o LCD

writeByteParallel(rs: Boolean, data: Int) – Escreve, em paralelo, data no LCD usando o *bit rs* para escolher o modo *data* ou *command*.

writeByteSerial(rs: Boolean, data: Int) – Escreve, em serie, data no LCD usando o *bit rs* para escolher o modo *data* ou *command*.

writeByte(rs: Boolean, data: Int) – Usa write ByteSerial.

writeCMD(data: Int) – Escreve um comando.

writeDATA(data: Int) – Escreve data.

write() – Escreve um Char ou uma String.

cursor(line: Int, column: Int) – Coloca o cursor na posição, (line, column).
clear() – Limpa o LCD.

3.2 Classe Serial Emitter

Este Módulo tem como objetivo enviar tramas para os diferentes módulos Serial Receiver.
Init() – Iniciar o usbport para escrever.
send(addr: Destination, data: Int, size : Int) - enviar 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 Conclusões

O módulo *Serial LCD Controller (SLCDC)* foi desenvolvido para facilitar a comunicação eficiente com um display *LCD* em serie.
O SLCDC recebe mensagens em serie compostas por nove bits de dados e um bit de paridade, identificando o tipo de mensagem (controlo ou dados) através do bit RS. O bit de paridade assegura a integridade da comunicação.
O *Serial Receiver* tem quatro componentes principais:
- *Serial Control*: Gerencia a receção de dados.
- *Shift Register*: Converte dados seriais para formato paralelo.
- *Counter*: Conta os bits recebidos.
- *Parity Check*: Verifica a paridade dos dados.

O *Serial Control* inicializa e verifica o sinal de receção (*enRX*), processando tramas completas e válidas. A verificação de paridade garante a integridade dos dados.
O *LCD Dispatcher* entrega dados validados ao *LCD*. O sinal *Dval* indica tramas válidas, e o *dispatcher* processa comandos rapidamente, mantendo o canal de receção livre.
Implementada em Java (Kotlin), a interface de controlo tem duas classes principais:
- *LCD*: Gerência a escrita no *LCD* em modo paralelo e serial.
- *Serial Emitter*: Envia tramas ao *Serial Receiver*, gerindo a comunicação via *USB*.

O *SLCDC* oferece uma solução robusta para comunicação serial com displays *LCD*, garantindo desempenho confiável através de técnicas de verificação de integridade e gestão eficiente de estados e sinais. Futuras melhorias incluirão a adaptação do sinal *enRX* para modo *active low*.

A. Descrição *VHDL* do bloco *Serial LCD Controller*

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

entity SerialLCDController is
port(
SDX, SCLK, SS, Reset, clk: in std_logic;
Dout: out std_logic_vector(8 downto 0);
WrL: out std_logic
);
end SerialLCDController;

architecture behavioral of SerialLCDController is

component SerialReceiver is
port(
SDX, SCLK, SS, accept, clk, Reset: in std_logic;
D: out std_logic_vector(8 downto 0);
DXval: out std_logic
);
```

```
end component;

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



signal term0: std_logic_vector(8 downto 0);
signal term1, term2: std_logic;


begin

SerialReceiver_U0: SerialReceiver port map(
SDX => SDX,
SCLK => SCLK,
SS => SS,
accept => term2,
Reset => Reset,
CLK => CLK,
D => term0,
DXval => term1
);


LCDDispatcher_U0: LCD_Dispatcher port map(
Din => term0,
Dval => term1,
clk => clk,
Reset => Reset,
WrL => WrL,
done => term2,
Dout => Dout
);


end behavioral;
```

B. Descrição VHDL do bloco Serial Receiver

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

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

```
architecture behavioral of SerialReceiver is

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

component Counter is
port(
Clear, clk: in std_logic;
Q: out std_logic_vector(3 downto 0)
);
end component;

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

component SerialControl is
port(
Reset, clk, enRX, accept, dFlag, pFlag, RXerror: in std_logic;
wr, init, DXval: out std_logic
);
end component;

component Compare9 is
port(
Data: in std_logic_vector(3 downto 0);
R: out std_logic
);
end component;

component Compare10 is
port(
Data: in std_logic_vector(3 downto 0);
R: out std_logic
);
end Component;


signal term0, term1, term2, term3, term4 : std_logic;
signal term5 : std_logic_vector(3 downto 0);

begin

SerialControl_U0: SerialControl port map(
Reset => Reset,
clk => clk,
```

```
enRX => SS,
accept => accept,
dFlag => term0,
pFlag => term1,
RXerror => term2,
wr => term3,
init => term4,
DXval => DXval
);

ParityCheck_U0: ParityCheck port map(
Data => SDX,
Init => term4,
clk => SCLK,
err => term2
);

Counter_U0: Counter port map(
Clear => term4,
clk => SCLK,
Q => term5
);

Compare9_U0: Compare9 port map(
Data => term5,
R => term0
);


Compare10_U0: Compare10 port map(
Data => term5,
R => term1
);

ShiftRegister_U0: ShiftRegister port map(
Data => SDX,
clk => SCLK,
enableShift => term3,
Reset => Reset,
D => D
);


end behavioral;
```

C. Descrição VHDL do bloco Serial Control

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

entity SerialControl is
port(
Reset, clk, enRX, accept, dFlag, pFlag, RXerror: in std_logic;
wr, init, DXval: out std_logic
);
end SerialControl;
```

```
architecture behavioral of SerialControl is

type STATE_TYPE is (STATE_INIT, STATE_WRITING, STATE_ERRORCHECK, STATE_WAITINGACCEPT);

signal CurrentState, NextState : STATE_TYPE;

begin

-- Flip-Flop's
CurrentState <= STATE_INIT when Reset = '1' else NextState when rising_edge(clk);

--Generate Next State
GenerateNextState:
process(CurrentState, enRX, accept, dFlag, pFlag, RXerror)

begin

case CurrentState is

when STATE_INIT => if(enRX = '1' and accept = '0') then
NextState <= STATE_WRITING;
else NextState <= STATE_INIT;
end if;

when STATE_WRITING => if(dFlag = '1') then
NextState <= STATE_ERRORCHECK;
else NextState <= STATE_WRITING;
end if;

when STATE_ERRORCHECK => if(pFlag = '1') then
if(RXerror = '0') then
NextState <= STATE_WAITINGACCEPT;
else NextState <= STATE_INIT;
end if;
else NextState <= STATE_ERRORCHECK;
end if;

when STATE_WAITINGACCEPT => if(accept = '1') then
NextState <= STATE_INIT;
else NextState <= STATE_WAITINGACCEPT;
end if;
end case;
end process;

wr <= '1' when (CurrentState = STATE_WRITING) else '0';
init <= '1' when (CurrentState = STATE_INIT) else '0';
DXval <= '1' when (CurrentState = STATE_WAITINGACCEPT) else '0';

end behavioral;
```

D. Descrição VHDL do bloco Parity Check

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

entity ParityCheck is
port(
```

```
data, init, clk: in std_logic;
err: out std_logic
);
end ParityCheck;

architecture behavioral of ParityCheck is

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

begin

ParityCheckCounter_U0: ParityCheck_Counter port map(
init => init,
data => data,
clk => clk,
err => err
);

end behavioral;
```

E. Descrição VHDL do bloco Shift Register

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

entity ShiftRegister is
port(
data : in std_logic;
clk : in std_logic;
enableShift : in std_logic;
Reset : in std_logic;
D : out std_logic_vector(8 downto 0)
);
end ShiftRegister;

architecture behavioral of ShiftRegister is

component ShiftRegister_FFD port(
clk : in std_logic;
Reset : in STD_LOGIC;
Set : in std_logic;
D : IN STD_LOGIC;
EN : IN STD_LOGIC;
Q : out std_logic
);
end component;

signal term0, term1, term2, term3, term4, term5, term6, term7, term8: std_logic;

begin
```

```
FFD0: ShiftRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
D => data,
EN => enableShift,
Q => term0
);

FFD1: ShiftRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
D => term0,
EN => enableShift,
Q => term1
);

FFD2: ShiftRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
D => term1,
EN => enableShift,
Q => term2
);

FFD3: ShiftRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
D => term2,
EN => enableShift,
Q => term3
);

FFD4: ShiftRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
D => term3,
EN => enableShift,
Q => term4
);

FFD5: ShiftRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
D => term4,
EN => enableShift,
Q => term5
);

FFD6: ShiftRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
```

```
D => term5,
EN => enableShift,
Q => term6
);

FFD7: ShiftRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
D => term6,
EN => enableShift,
Q => term7
);

FFD8: ShiftRegister_FFD port map(
clk => clk,
Reset => Reset,
Set => '0',
D => term7,
EN => enableShift,
Q => term8
);

D(0) <= term8;
D(1) <= term7;
D(2) <= term6;
D(3) <= term5;
D(4) <= term4;
D(5) <= term3;
D(6) <= term2;
D(7) <= term1;
D(8) <= term0;


end behavioral;
```

F. Descrição VHDL do bloco Counter

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

entity Counter is
port(
clear, clk: in std_logic;
Q: out std_logic_vector(3 downto 0)
);
end Counter;

architecture behavioral of Counter is

component Counter_Adder port(
A : in std_logic_vector(3 downto 0);
B : in std_logic_vector(3 downto 0);
CYi : in std_logic;
S : out std_logic_vector(3 downto 0)
);
end component;
```

```
component Counter_Reg port(
D : in std_logic_vector(3 downto 0);
CLK : in std_logic;
E : in std_logic;
Reset : in std_logic;
Q : out std_logic_vector(3 downto 0)
);
end component;

signal term0, term1 : std_logic_vector(3 downto 0);

begin

Counter_Adder_U0: Counter_Adder port map(
A => term0,
B(3) => '0',
B(2) => '0',
B(1) => '0',
B(0) => '1',
CYi => '0',
S => term1
);


Counter_Reg_U0: Counter_Reg port map(
D => term1,
clk => clk,
E => '1',
Reset => clear,
Q => term0
);

Q <= term0;

end behavioral;
```

G. Descrição VHDL do bloco Compare9

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

entity Compare9 is
port(
Data: in std_logic_vector(3 downto 0);
R: out std_logic
);
end Compare9;

architecture behavioral of Compare9 is

begin

R <= Data(3) and not Data(2) and not Data(1) and Data(0);

end behavioral;
```

H. Descrição VHDL do bloco Compare10

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

entity Compare10 is
port(
Data: in std_logic_vector(3 downto 0);
R: out std_logic
);
end Compare10;

architecture behavioral of Compare10 is

begin

R <= Data(3) and not Data(2) and Data(1) and not Data(0);

end behavioral;
```

I. Código Kotlin – LCD

```
import isel.leic.utils.Time
import kotlin.math.ln

object LCD {
    private const val LINES = 2
    private const val COLS = 16
    private const val rs_bit = 16
    private const val E_bit = 32
    private const val clk_reg_bit = 64

    fun init() {
        Time.sleep(15)

        writeCMD(48)
        Time.sleep(5)

        writeCMD(48)
        Time.sleep(1)

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

    }


    private fun writeByteParallel(rs: Boolean, data: Int){
        if(rs){
            HAL.setBits(rs_bit)
        }
        else{
            HAL.clrBits(rs_bit)
```

```
    }

    val shift_right = data.shr(4)

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



    HAL.writeBits(15, data)
    HAL.setBits(clk_reg_bit)
    HAL.clrBits(clk_reg_bit)

    HAL.setBits(E_bit)
    HAL.clrBits(E_bit)

}

private fun writeByteSerial(rs: Boolean, data: Int) {
    if(rs){
        HAL.setBits(rs_bit)
    }
    else{
        HAL.clrBits(rs_bit)
    }
    SerialEmitter.send(SerialEmitter.Destination.LCD, data, 9)
    Time.sleep(1)
}

private fun writeByte(rs: Boolean, data: Int){
    writeByteSerial(rs, data)
}

private fun writeCMD(data: Int){
    writeByte(false, data)
}

private fun writeDATA(data: Int){
    writeByte(true, data)
}

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

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


fun cursor(line: Int, column: Int){
    val write_data = 128
   writeCMD((line*0x40)+column+write_data)
}

fun clear(){
```

```
        writeCMD(1)
    }
}
```

II. Código Kotlin – Serial Emiter

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

    // Inicia a classe
    fun init(){
        val mask_clr = 6
        HAL.setBits(mask_clr)
    }


// 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){
        val mask_data = 1
        var counter = 0
        val mask_LCD = 2
        val mask_SCORE = 4
        val mask_clock = 8
        var data_temp = data

        if(addr == Destination.LCD)
            HAL.clrBits(mask_LCD)
        else if(addr == Destination.SCORE)
            HAL.clrBits(mask_SCORE)

        for(i in 0..size-1){
            HAL.clrBits(mask_clock)
            if (data_temp%2 ==0){
                HAL.writeBits(mask_data, 0)
                data_temp = data_temp.shr(1)
            }
            else {
                HAL.writeBits(mask_data, 1)
                data_temp = data_temp.shr(1)
                counter++
            }
            HAL.setBits(mask_clock)
        }

        HAL.clrBits(mask_clock)
        if (counter%2==0){
            HAL.writeBits(mask_data, 0)
        }else{
            HAL.writeBits(mask_data, 1)
        }
        HAL.setBits(mask_clock)
        HAL.setBits(mask_LCD)
        HAL.setBits(mask_SCORE)
    }
}
```