O módulo Serial LCD Controller é constituído por dois blocos principais: *i*) o recetor da trama envida pelo Controlo (*Serial Receiver*); *ii*) o bloco de entrega ao LCD (designado por Dispatcher). Neste caso o módulo *Control*, implementado em *software*, é a entidade que envia a trama.

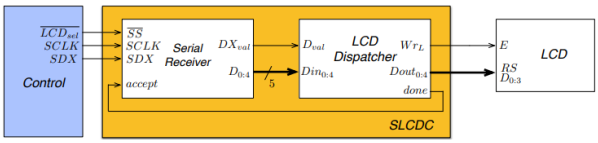
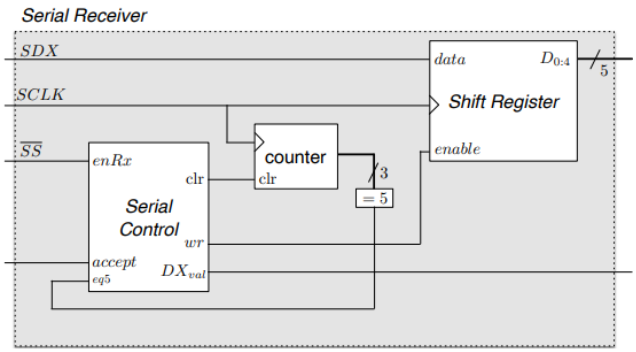


Figura 1 – Diagrama de blocos do módulo *Keyboard Reader*

1 Serial Receiver

O bloco *Serial Receiver* tem como função converter o input de em série, num output de 5 bits (em paralelo), de forma que o próprio LCD (Hardware) consiga receber os dados transmitidos pelo controlo (Software em *kotlin*). Este bloco cumpre a sua funcionalidade da forma que passo a explicar. O Sinal *SDX* é o bit de transmissão dos dados, o *SCLK* é o *clock* controlado pelo *Software*, o not SS é o sinal de *enable* que diz se o controlo está a transferir dados.
Os Blocos que podemos ver no diagrama da Figura 2 são: um Shift *Register*, que tem a função de ir guardando os bits que vão chegando do sinal SDX; um *counter* de 3 bits, que tem a função de guardar em que que bit da trama recebida é que o circuito vai; um bloco "=5" ou *equals five*, que tem como função fazer output de '1', caso o contador chegue ao valor binário 5 e, finalmente, um Controlo (Serial *Control*), que tem como função, como o próprio nome indica, controlar estes componentes dividindo as "tarefas" em estados.



O bloco *Serial Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 2.

O Serial *Control* define três estados: o estado *Receiving*, *Waiting* e o *Accepted*. Este começa no estado *Receiving*, em que not SS está a '1' (está desactivado, pois é um sinal "*active low*") e, portanto, está à espera de que alguma trama seja enviada. Assim que o sinal not SS é posto a '0' (fica activo), o controlo passa para o estado *Waiting*, em que o controlo liga o *enable* do Shift *Register* e desliga o *clr* do *counter*, permitindo que este conte. Neste estado o Serial *Receiver* está a receber os bits da trama. Quando o contador chega ao valor binário de 5, o módulo *equals five* gera um '1' no output, fazendo o controlo mudar de estado, visto que significa que os 5 bits da trama já foram recebidos. O novo estado é o estado *Accepted*, em que o *enable* do Shift *register* não é ligado, mas o sinal *DXval* é posto a '1'. Como o output do Shift *register* está sempre conectado à saída "K" do bloco em análise, a forma que temos de avisar o próximo componente de que a saída é valida é explicitando-o noutra saída, que é o caso do *DXval*, que tem como função avisar o LCD *Dispatcher* que os 5 bits que está a receber são válidos. Assim que o próximo componente tiver lido o output "K", este envia um sinal ao Serial *Receiver* a dizer que já o fez. Este bit é o sinal *accepted* que, estando o controlo no estado *Accepted*, quando o bit homónimo é posto a '1', o controlo percebe que os dados foram lidos, e passa, de novo, para o primeiro estado, *Receiving*, onde repetirá todo este processo.

A descrição hardware do bloco *Serial Receiver* em VHDL encontra-se no Anexo 0A.

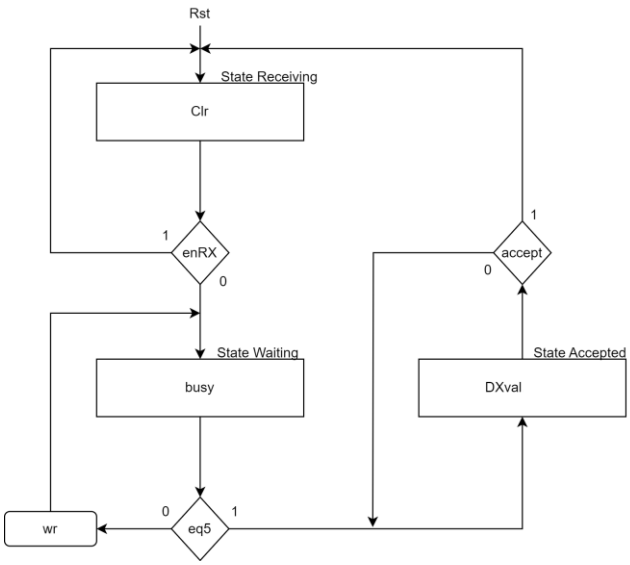


Figura 2 – Máquina de estados do bloco *Serial Control*

2 LCD Dispatcher

Implementou-se o módulo *Dispatcher* em *hardware*, recorrendo a linguagem *VHDL* e seguindo a arquitetura lógica apresentada na Figura 3.

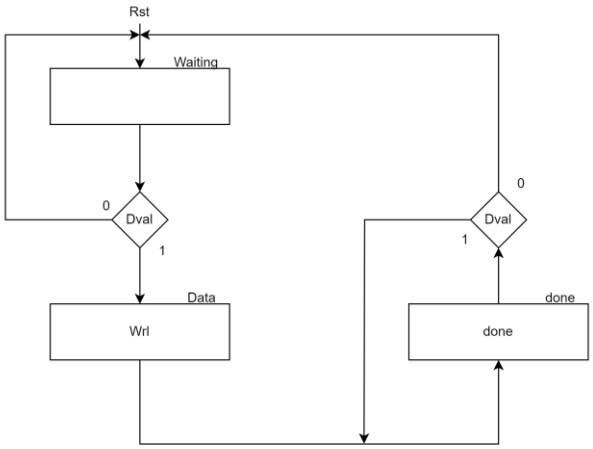


Figura 3 – ASM-*Chart* do módulo *Dispatcher*.

O *Dispatcher* possui a logica de uma maquina de estados, que em um momento inicial permanece no estado *Waiting* até que receba o sinal *Dval* vindo do modulo Serial *Receiver*, e quando recebe o mesmo, passa para o estado Data que garante um ciclo de *clock* com o estado *Wrl* a 1 (oque possibilita a leitura da trama pelo LCD) e finaliza com o estado *Done* que faz a leitura do sinal *Dval* e caso esteja a 0, passa novamente ao estado *Waiting* até que o Serial *Receiver* termine a leitura da trama enviada pelo controlo.

3 Interface com o Control

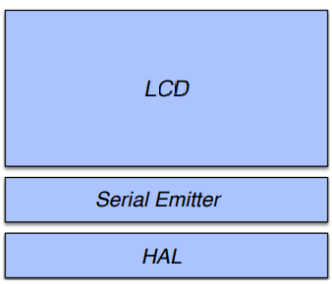


Figura 6 – Diagrama lógico do módulo Control de interface com o módulo Serial LCD Controller

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

3.1 Serial Emitter

O *Serial Emitter* é um objeto em *Kotlin* que atua por cima do HAL (*Hardware Abstraction Layer*) e é uma camada que foi implementada tendo em vista o limite de bits que podem ser utilizados na saída *UsbPort* e os diferentes destinos que a mensagem possui, sendo capaz de diferenciar os dados encaminhados para o SLSDC e para o *Door Controller*, além de também reduzir o número de bits necessários para transmitir os sinais de controlo e a trama de 5 bits. Enviando os sinais de controlo do *door controller*, *SLCDC* e *clock* em saídas especificas e então a trama de 5 bits em serie por somente uma saída. Totalizando o uso de 4 bits para enviar a trama.

3.2 LCD

O LCD é responsável por enviar todas as mensagens lidas pelo LCD, sejam dados ou comandos para a inicialização do mesmo. O um modulo funciona diretamente sobre o *Serial Emitter*, sendo assim todas as tramas são enviadas em série.

4 Conclusões

O bloco Serial Receiver tem como função converter o input em série num output de 5 bits (em paralelo), de forma que o próprio LCD possa receber os dados transmitidos pelo controlo implementado em software. Já o bloco Dispatcher é implementado em hardware e possui a lógica de uma máquina de estados que recebe o sinal Dval do Serial Receiver e passa pelos estados Data e Wrl antes de terminar no estado Done. O módulo Control, implementado em software atua como interface com o Serial LCD Controller e utiliza o objeto Serial Emitter para diferenciar os dados encaminhados para o SLSDC.

A. Descrição VHDL do bloco Serial Receiver

```
library IEEE;
use IEEE.std_logic_1164.all;

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

architecture logic of SerialReceiver is

      component SerialControl is
            port (
                  clk: in std_logic;
                  enRx: in std_logic;
                  accept: in std_logic;
                  clr: out std_logic;
                  wr: out std_logic;
                  rst: in std_logic;
                  DXval: out std_logic;
                  eq5: in std_logic
            );
      end component;
      component ContadorUpDown3 is
            port (
      Clk : in std_logic; -- Clock
      Rst : in std_logic; -- Reset
      Q : out std_logic_vector(2 downto 0) -- output
            );
      end component;
      component shiftReg5 is
            port(
                  Clk: in std_logic;
                  I: in std_logic;
                  En: in std_logic;
                  rst: in std_logic;
                  O: out std_logic_Vector(4 downto 0)
            );
```

```
        end component;

    signal seq5in, swr, sclr: std_logic;
        signal seq5out: std_logic_vector(2 downto 0);

begin

    UCounter: ContadorUpDown3
            port map(
                Clk => SCLK, -- Clock
                Rst => sclr, -- Reset
                Q => seq5out
            );

        UshiftReg5: shiftReg5
            port map(
                Clk => SCLK,
                I => SDX,
                En => swr,
                rst => reset,
                O => D
            );
        USerialControl: SerialControl
            port map(
                clk => MCLK,
                enRx => not_SS,
                accept => accept,
                clr => sclr,
                wr => swr,
                rst => reset,
                DXval => DXval,
                eq5 => seq5in
            );
        seq5in <= (seq5out(0) and not seq5out(1) and seq5out(2));


end logic;
```

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);
                  Dout: out std_logic_vector(4 downto 0);
                  Wrl: out std_logic;
                  rst: in std_logic;
                  clk: in std_logic;
                  done: out std_logic
   );
end Dispatcher;

architecture behavioral of Dispatcher is
      type STATE_TYPE is (STATE_DATA, STATE_WAITING, STATE_DONE);

      signal CurrentState, NextState: STATE_TYPE;
      signal sWrl : std_logic;

      begin
--          Flip-Flop's
            CurrentState <= STATE_WAITING when rst = '1' else Nextstate when rising_edge(clk);

--          Generate Next State
            GenerateNextState:
                  process (CurrentState, Dval, sWrl)
                        begin
                              case CurrentState is
                                    when STATE_WAITING  => if (Dval = '1') then

                                                                NextState <= STATE_DATA;

                                                          else

                                                                NextState <= STATE_WAITING;
```

```
                                        end if;
                    when STATE_DATA     =>  if (sWrl = '1') then
                                                NextState <= STATE_DONE;
                                            else
                                                NextState <= STATE_DATA;
                                            end if;
                    when STATE_DONE     =>  if (Dval = '0') then
                                                NextState <= STATE_WAITING;
                                            else
                                                NextState <= STATE_DONE;
                                            end if;
                end case;
            end process;

--          Generate Outputs
            sWrl <= '1' when (CurrentState = STATE_DATA) else '0';
            done <= '1' when (CurrentState = STATE_DONE) else '0';
            Wrl <= sWrl;
            Dout <= Din;

end architecture;
```

C. Atribuição de pinos do módulo *LCD Controller*

```
#============================================================
# 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_location_assignment PIN_P11 -to Clk

#============================================================
# SW
#============================================================
set_location_assignment PIN_C10 -to SCLK
set_location_assignment PIN_C11 -to not_SS
set_location_assignment PIN_D12 -to SDX
set_location_assignment PIN_F15 -to rst

#============================================================
# LED
#============================================================
set_location_assignment PIN_A8 -to LedD[0]
set_location_assignment PIN_A9 -to LedD[1]
set_location_assignment PIN_A10 -to LedD[2]
set_location_assignment PIN_B10 -to LedD[3]
set_location_assignment PIN_D13 -to LedD[4]
set_location_assignment PIN_A11 -to LedEn

#============================================================
# GPIO, GPIO connect to GPIO Default
#============================================================

set_location_assignment PIN_W8 -to D[4]
set_location_assignment PIN_V5 -to En
set_location_assignment PIN_W11 -to D[3]
set_location_assignment PIN_AA10 -to D[2]
set_location_assignment PIN_Y8 -to D[1]
set_location_assignment PIN_Y7 -to D[0]
```

D. Código *Kotlin - LCD*

```
const val NIBBLE = 0x0F
const val DATA = 0x01
const val CLEAR = 0x01
const val FUNC_SET = 0x03
const val FUNC_SET_2 = 0x2C
const val FUNC_SET_3 = 0x06
const val DISPLAY_OFF = 0x08
const val DISPLAY_CLEAR = 0x01
const val FUN_SET_4 = 0x02
const val ON_DISPLAY = 0x0F
const val CURSOR_MASK = 0x80

object LCD {
    const val LINES = 2
    const val COLS = 16
//    private fun writeNibbleParallel(rs: Boolean, data: Int){
//        HAL.writeBits(NIBBLE, data)
//        if (rs) HAL.setBits(COMMAND) else HAL.clrBits(COMMAND)
//        HAL.setBits(ENABLE)
//        HAL.clrBits(ENABLE)
//    }
    private fun writeNibbleSerial(rs: Boolean, data: Int){
        val msg = if (!rs) (NIBBLE and data) shl (1) else DATA or
((NIBBLE and data) shl(1))
        SerialEmitter.send(Destination.LCD, msg)
        Thread.sleep(1)
    }
    private fun writeBytes(rs: Boolean, data: Int){
        writeNibbleSerial(rs,data shr (4))
        writeNibbleSerial(rs, data)
    }
    private fun writeCMD(data: Int) =
        writeBytes(false, data)
    private fun writeData(data: Int) =
        writeBytes(true, data)
    fun init(){
        repeat(3){
            writeNibbleSerial(false, FUNC_SET)
            isel.leic.utils.Time.sleep(5)
        }
        writeNibbleSerial(false, FUN_SET_4)
        writeCMD(FUNC_SET_2)
        writeCMD(DISPLAY_OFF)
        writeCMD(DISPLAY_CLEAR)
        writeCMD(FUNC_SET_3)
        writeCMD(ON_DISPLAY)
    }
    fun write(c: Char) =
```

```
        writeData(c.code)
    fun write(text: String) =
        text.forEach(::write)
    fun cursor(line:Int, column: Int) =
        writeCMD(CURSOR_MASK or column+(line*0x40))
    fun clear() =
        writeCMD(CLEAR)
}
```

E. Código *Kotlin - SerialEmitter*

```
const val SDC_ENABLE = 0x02
const val LCD_ENABLE = 0x01
const val SERIAL_CLK = 0x10
const val SIZE = 0x05
const val SERIAL_NIBBLE = 0x08
const val BUSY = 0x40
const val SERIAL_NIBBLE_MASK = 0x01
const val SERIAL_NIBBLE_SHIFT = 0x03
enum class Destination { LCD, DOOR }

object SerialEmitter {
    fun init(){
        HAL.setBits(SDC_ENABLE or LCD_ENABLE)
        HAL.clrBits(SERIAL_CLK)
    }
    fun send(addr: Destination, data: Int){
        when(addr){
            Destination.LCD -> HAL.clrBits(LCD_ENABLE)
            Destination.DOOR -> HAL.clrBits(SDC_ENABLE)
        }
        repeat(SIZE){
            val msg = ((data shr(it)) and SERIAL_NIBBLE_MASK) shl
(SERIAL_NIBBLE_SHIFT)
            HAL.writeBits(SERIAL_NIBBLE, msg)
            HAL.setBits(SERIAL_CLK)
            HAL.clrBits(SERIAL_CLK)
        }
        when(addr) {
            Destination.LCD -> HAL.setBits(LCD_ENABLE)
            Destination.DOOR -> HAL.setBits(SDC_ENABLE)
        }
    }
    fun isBusy(): Boolean = HAL.readBits(BUSY) != 0
}
```