Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

O módulo *Integrated Output System (IOS)* implementa a interface com o mecanismo de dispensa de bilhetes e com o *LCD*, fazendo a receção em série da informação enviada pelo módulo de controlo e entregando-a posteriormente ao destinatário, conforme representado na Figura 1.

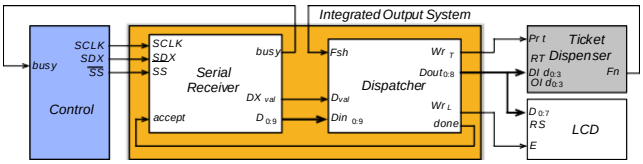


Figura 1 – Diagrama de blocos do *Integrated Output System*

O módulo *IOS* recebe, em série, uma mensagem constituída por 10 bits de informação e um bit de paridade. A comunicação com este módulo realiza-se segundo o protocolo ilustrado na Figura 2, em que o bit *TnL* identifica o destinatário da mensagem. Nas mensagens para o mecanismo de dispensa de bilhetes, ilustrado na Figura 3, o bit *RT* é o primeiro bit de informação e indica o tipo de bilhete (ida ou ida/volta), os seguintes 4 bits contêm o código de identificação da estação de destino, e os restantes 4 bits contêm o código de identificação da estação de origem. O último bit contém a informação de paridade par, utilizada para detetar erros de transmissão. As mensagens para o *LCD*, ilustrado na Figura 3, contêm para além do bit *TnL* e do bit paridade outros 9 bits de dados a entregar ao dispositivo: o bit *RS*, que é o primeiro bit de informação e indica se a mensagem é de controlo ou dados, e os restantes 8 bits que contêm os dados a entregar ao *LCD*.

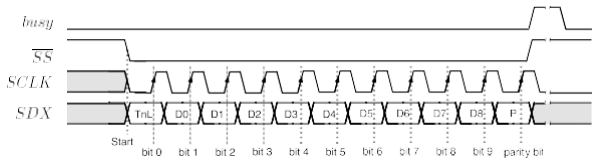


Figura 2 – Protocolo de comunicação com o módulo *Integrated Output System*

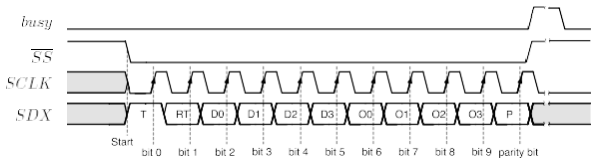


Figura 3 – Trama para o mecanismo de dispensa de bilhetes (*Ticket Dispenser)*

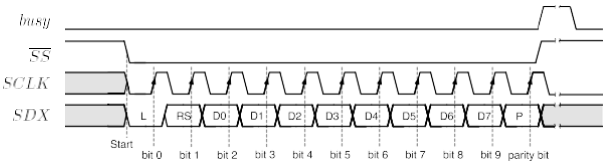


Figura 4 – Trama para o *LCD*

O emissor, realizado em *software*, quando pretende enviar uma trama para o módulo *IOS* aguarda que este esteja disponível para receção, ou seja, que o sinal *busy* esteja desativo. Em seguida, promove uma condição de início de trama (*Start*), que corresponde a uma transição descendente na linha $\overline{SS}$, mantendo-se esta no valor lógico zero até ao fim da transmissão. Após a condição de início, o módulo *IOS* armazena os bits de dados da trama nas transições ascendentes do sinal *SCLK*. O sinal *busy* é ativado, pelo módulo *IOS*, quando termina a receção de uma trama válida, ou seja, quando recebe a totalidade dos bits de dados e o bit de paridade correto. O sinal *busy* é desativado após o *Dispatcher* informar o *IOS* que já processou a trama.

1.1.1 *Serial Receiver*

O bloco *Serial Receiver* do módulo *IOS* é constituído por quatro blocos principais: *i*) um bloco de controlo; *ii*) um bloco de memória, implementado através de um registo de deslocamento; *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 5.

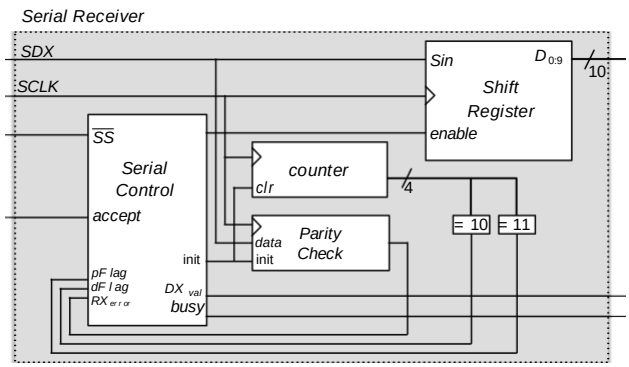
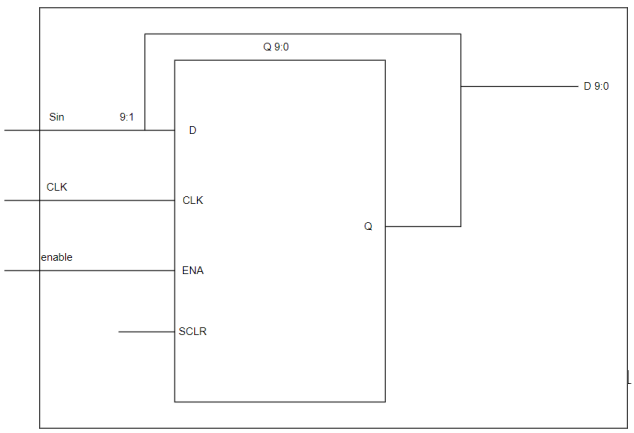


Figura 5 – Diagrama de blocos do bloco *Serial Receiver*



Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

Figura 6 - Diagrama de blocos do bloco *Shift-Register*

Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

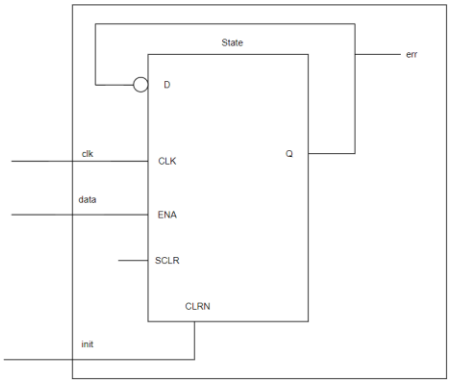


Figura 7 - Diagrama de blocos do bloco *Parity Check*

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

A nossa solução de ASM para o Serial Control do bloco Serial Receiver tem os seguintes estados:

Estado INIT: Ativamos o sinal Init para começar a contagem dos bits da trama com o nosso bloco counter.

Estado WR: Aqui ativamos o sinal WR para escrever a a trama no nosso bloco Shift Register e damos clocks para o counter ir contando até 10, a partir do qual recebe a flag dFlag que indica que os bits de informação já foram recebidos.

Estado READ_D_FLAG: Neste estado, estamos à espera da flag pFlag para verificar o (11º) bit de paridade, verificamos também se o sinal nSS continua desativo para sabermos que não paramos de receber a trama.

Estado BUSY: Neste estado ativamos o sinal Dxval para indicar que a trama é válida (caso não seja detetado nenhum erro) para o Dispatcher a ler e ativamos o sinal busy para indicarmos que a trama está a ser entregue ao Dispatcher, logo não podemos receber mais tramas enquanto o sinal accept for ativo.

A descrição hardware do bloco *Serial Receiver* em VHDL encontra-se no Anexo 3.

1.1.2 Dispatcher

O bloco *Dispatcher* é responsável pela entrega das tramas válidas recebidas pelo bloco *Serial Receiver* ao *Ticket Dispenser* e ao *LCD*, através da ativação do sinal Wr_T e Wr_L. A receção de uma nova trama válida é sinalizada pela ativação do sinal D_{val}.
O processamento das tramas recebidas pelo *IOS*, para o *Ticket Dispenser* ou para o *LCD*, deverá respeitar os comandos definidos pelo fabricante de cada periférico, devendo sinalizar o término da execução logo que seja possível ao *Serial Receiver*.

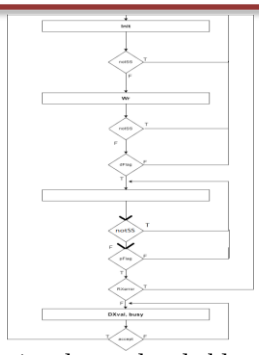


Figura8- Máquina de estados do bloco *Serial Control*

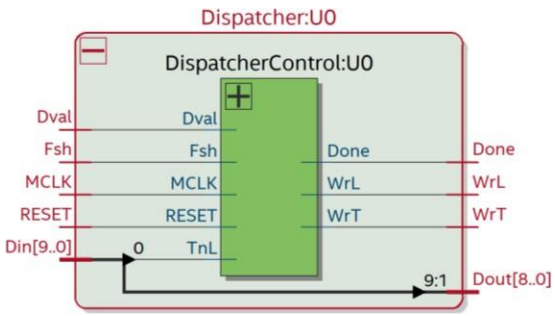


Figura 9 – diagrama do bloco Dispatcher

Dispatcher foi implementado de acordo com o diagrama de blocos representado na **Error! Reference source not found.**.

O nosso Dispatcher é constituido por uma máquina de estados e liga o bit 0 (TnL) a esta, retornando na saída os restantes 9 bits de informação para entregar ao respetivo destino identificado pelo TnL.

O nosso Dispatcher Control tem um ASM com os seguintes estados:

Estado INIT: Neste estado, não ativamos saídas, apenas esperamos pelo sinal Dval para sinalizar uma trama válida para receber e esperamos pelo bit TnL que indica o destino da trama.

Estado WrT: Neste estado enviamos a trama válida para o TicketDispenser pois o bit TnL está a 1, sinalizando que é esse o destino, por fim esperamos pelo sinal Finish para indicar que a trama foi entregue.

Estado WrL: Neste estado enviamos a trama válida para o LCD pois o bit TnL está a 0, sinalizando que é esse o destino.

Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

Estado DONE: Neste estado indicamos que já entregámos a trama ao destino respetivo e avançamos diretamente para o estado inicial.

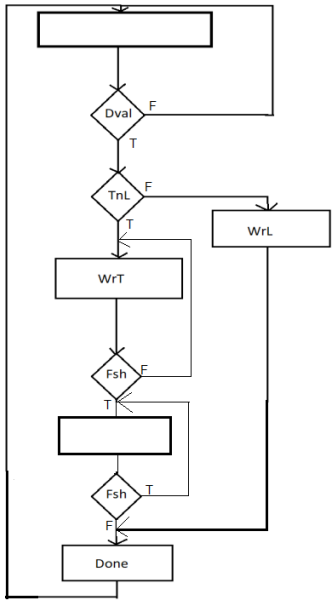


Figura 10 - Máquina de estados do bloco *Dispatcher Control*

No dispatcher, **alerámos, desde o trabalho anterior,** o ASM dele para que o sinal Finish pudesse voltar a 0 quando fosse ativo antes do Dispatcher começar a receber outra trama válida. Adicionámos também um CLKDIV para dividir o CLK da máquina de estados do Dispatcher para quando a trama fosse para o LCD, desse tempo para esta ser exposta no mesmo.

Com base nas descrições do bloco *Serial Receiver* e *Dispatcher* implementou-se o módulo *Integrated Output System*, a descrição hardware do bloco *Serial Receiver* em VHDL encontra-se no Anexo C.

1.1.3 Ticket Dispenser

O *Ticket Dispenser* recebe em 4 bits o código da estação de destino ($DId_{0:3}$), noutros 4 bits o código da estação de origem ($OId_{0:3}$) a imprimir no bilhete e ainda o bit *RT* que define o tipo de bilhete (ida ou ida/volta). O comando de impressão do bilhete com os códigos presentes em *DId* e *RT* é realizado pela ativação do sinal de impressão (*Prt*). Em resposta, o *Ticket Dispenser* ativa o sinal de término de execução (*Fn*) quando concluída a dispensa do bilhete. Os sinais *Fn* e *Prt* têm o comportamento descrito no diagrama temporal apresentado na Figura 8.

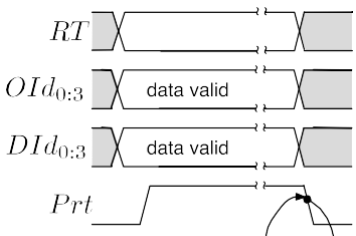


Figura 8- Diagrama temporal do mecanismo de dispensa de bilhetes

A descrição hardware do bloco *Ticket Dispenser* em VHDL encontra-se no Anexo D.

2 Interface com o *Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem *Kotlin* seguindo a arquitetura lógica apresentada na Figura 9.

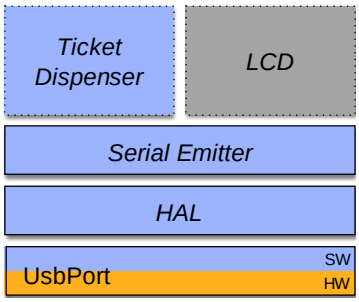


Figura 9 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

Os módulos de software *HAL*, *Serial Emitter* e *Ticket Dispenser* desenvolvidos são descritos nas secções 2.1., 2.2 e 0, e o código fonte desenvolvido nos Anexos E, F e G, respetivamente.

2.1 HAL

A classe HAL é responsável pela ligação entre o hardware físico e o software. Esta acessa ao sistema UsbPort por Software, lendo os dados de entrada e escrevendo os dados de saída. Esta classe dá-nos também funções básicas para desenvolver o controlo das portas do UsbPort.

2.2 SerialEmitter

A classe SerialEmitter é responsável pelo envio de tramas e geração do bit de paridade para o Serial Receiver. Para esta classe foi criado um método adicional (ParityCheck) que permite gerar o bit de paridade de acordo com a trama de bits de informação.

Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

2.3 TicketDispenser

A classe Ticket Dispenser é responsável pelo mecanismo da emissão de bilhete em si, ou seja, envia comandos com a informação necessária para imprimir e dispensar os bilhetes.

3 Conclusões

Até agora, concluimos que ao desenvolver o resto do projeto, temos de ter uma determinada atenção na interação entre software e hardware e respeitar as hierarquias de blocos. Devíamos também tirar proveito de teste feitos na placa visto que não conseguimos ter acesso a certas ocorrências em ambiente de simulação.

Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

A. Descrição VHDL do bloco *Serial Receiver*

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity SerialReceiver is
      Port (
            MCLK : in STD_LOGIC;
            RESET : in STD_LOGIC;
            SDX : in STD_LOGIC;
            SCLK : in STD_LOGIC;
            nSS : in STD_LOGIC;
            accept : in STD_LOGIC;
            busy : out STD_LOGIC;
            DXval : out STD_LOGIC;
            Dout : out STD_LOGIC_VECTOR(9 downTo 0)
      );
end SerialReceiver;

architecture Structural of SerialReceiver is

component SerialControl is
    Port ( RESET : in STD_LOGIC;
           MCLK : in  STD_LOGIC;
           nSS : in  STD_LOGIC;
           accept : in  STD_LOGIC;
           pFlag : in  STD_LOGIC;
           dFlag : in  STD_LOGIC;
           RXerror : in  STD_LOGIC;
           busy : out STD_LOGIC;
           wr : out  STD_LOGIC;
           init : out  STD_LOGIC;
           DXval : out  STD_LOGIC);
end component;
```

SerialReceiver 1

```vhdl
component counter_4bit is
    Port ( clk : in STD_LOGIC;
           clr : in STD_LOGIC;
           q : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component shift_register is
    Port ( Sin  : in STD_LOGIC;
           CLK  : in STD_LOGIC;
           enable  : in STD_LOGIC;
           D    : out STD_LOGIC_VECTOR(9 downto 0));
end component;

component ParityCheck is
    Port ( clk : in  STD_LOGIC;
           data : in  STD_LOGIC;
           init : in  STD_LOGIC;
           err : out  STD_LOGIC);
end component;

signal tenFlag : std_logic;
signal elevenFlag : std_logic;
signal dispatcher_wr : std_logic;
signal dispatcher_init : std_logic;
signal dispatcher_err : std_logic;
signal dispatcher_q : std_logic_vector(3 downTo 0);

begin


tenFlag <= dispatcher_q(3) and (not dispatcher_q(2)) and dispatcher_q(1) and (not dispatcher_q(0));
elevenFlag <= dispatcher_q(3) and (not dispatcher_q(2)) and dispatcher_q(1) and dispatcher_q(0);
```

SerialReceiver 2

```vhdl
U0:     SerialControl
        port map(RESET => RESET, MCLK => MCLK, nSS => nSS, accept => accept, pFlag => elevenFlag,
        dFlag => tenFlag, RXerror => dispatcher_err,
        wr => dispatcher_wr, init => dispatcher_init, DXval => DXval, busy => busy);


U1:     counter_4bit
        port map(clk => SCLK, clr => dispatcher_init, q => dispatcher_q);


U2:     ParityCheck
        port map(clk => SCLK, data => SDX, init => dispatcher_init, err => dispatcher_err);


U3:     shift_register
        port map(clk => SCLK, enable => dispatcher_wr, Sin => SDX, D => Dout);


end Structural;
```

SerialReceiver 3

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity SerialControl is
      Port (
         RESET : in STD_LOGIC;
         MCLK : in  STD_LOGIC;
         nSS : in  STD_LOGIC;
         accept : in  STD_LOGIC;
         pFlag : in  STD_LOGIC;
         dFlag : in  STD_LOGIC;
         RXerror : in  STD_LOGIC;
         busy : out STD_LOGIC;
         wr : out  STD_LOGIC;
         init : out  STD_LOGIC;
         DXval : out  STD_LOGIC
      );
end SerialControl;

architecture Behavioral of SerialControl is

      type  STATE_TYPE is (
            ESTADO_INIT,
            ESTADO_WR,
            ESTADO_READ_D_FLAG,
            ESTADO_BUSY
      );

      signal CurrentState, NextState : STATE_TYPE;

begin

      CurrentState <= ESTADO_INIT when RESET = '1' else NextState when rising_edge(MCLK);


      GenerateNextState: process(CurrentState, nSS, dFlag, pFlag, RXerror, accept)
      begin
```

SerialControl 1

```vhdl
CurrentState <= ESTADO_INIT when RESET = '1' else NextState when rising_edge(MCLK);


GenerateNextState: process(CurrentState, nSS, dFlag, pFlag, RXerror, accept)
begin

   case CurrentState is
   when ESTADO_INIT          => if(nSS = '0') then
                                   NextState <= ESTADO_WR;
                                else
                                   NextState <= ESTADO_INIT;
                                end if;

   when ESTADO_WR            => if(nSS = '1') then
                                   NextState <= ESTADO_INIT;
                                elsif(nSS = '0' and dFlag = '1') then
                                   NextState <= ESTADO_READ_D_FLAG;
                                else
                                   NextState <= ESTADO_WR;
                                end if;

   when ESTADO_READ_D_FLAG   => if(nSS = '1') then
                                   NextState <= ESTADO_INIT;
                                elsif(nSS = '0' and pFlag = '1' and RXerror = '0') then
                                   NextState <= ESTADO_BUSY;
                                elsif(nSS = '0' and pFlag = '0') then
                                   NextState <= ESTADO_READ_D_FLAG;
                                else
                                   NextState <= ESTADO_INIT;
                                end if;

   when ESTADO_BUSY          => if(accept = '1') then
                                   NextState <= ESTADO_INIT;
                                else
                                   NextState <= ESTADO_BUSY;
                                end if;

   end case;
```

SerialControl 2

Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

```
        end process;

    init <= '1' when (CurrentState = ESTADO_INIT) else '0';
    wr <= '1' when (CurrentState = ESTADO_WR) else '0';
    DXval <= '1' when (CurrentState = ESTADO_BUSY) else '0';
    busy <= '1' when (CurrentState = ESTADO_BUSY) else '0';

end Behavioral;
```

SerialControl 3

```
use IEEE.STD_LOGIC_1164.ALL;

entity shift_register is
   port(
      Sin    : in STD_LOGIC;
      CLK    : in STD_LOGIC;
      enable : in STD_LOGIC;
      D      : out STD_LOGIC_VECTOR(9 downto 0)
      );
end shift_register;

architecture Behavioral of shift_register is

signal q :  std_logic_vector(9 downto 0);

begin

   shiftBits: process(Sin,CLK)
   begin
         if(enable = '1' and rising_edge(CLK)) then
            q(0) <= q(1);
            q(1) <= q(2);
            q(2) <= q(3);
            q(3) <= q(4);
            q(4) <= q(5);
            q(5) <= q(6);
            q(6) <= q(7);
            q(7) <= q(8);
            q(8) <= q(9);
            q(9) <= Sin;
            end if;
   end process;

   D <= q;

end Behavioral;
```

Shift Register

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity counter_4Bit is
   Port ( clk : in STD_LOGIC;
          clr : in STD_LOGIC;
          q : out STD_LOGIC_VECTOR (3 downto 0));
end counter_4Bit;

architecture Structural of counter_4Bit is

   component CounterLogic_4Bit is
      PORT( operandA : IN std_logic_vector(3 downto 0);
            R : OUT std_logic_vector(3 downto 0));
   end component;

   component register_D_R is
      Generic ( WIDTH : positive := 1 );
      Port( CLK : in STD_LOGIC;
            RST : in STD_LOGIC;
            D : in STD_LOGIC_VECTOR(WIDTH-1 downto 0);
            Q : out STD_LOGIC_VECTOR(WIDTH-1 downto 0));
   end component;

   signal operandA, result : std_logic_vector(3 downto 0);

   begin

   U0: CounterLogic_4Bit
         PORT MAP(   operandA => operandA,
                     R => result
                     );

   U1: register_D_R
         Generic map (  WIDTH => 4 )
         Port map(
            CLK => clk,
            RST => clr,
            D => result,
            Q => operandA
         );

   q <= operandA;

end Structural;
```

Counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ParityCheck is
   Port (
      clk : in  STD_LOGIC;
      data : in  STD_LOGIC;
      init : in  STD_LOGIC;
      err : out  STD_LOGIC
   );

end ParityCheck;

architecture Behavioral of ParityCheck is

   signal estado: STD_LOGIC; -- guarda estado do parity check

begin

   check: process (init, clk)
   begin
      if (init = '1') then -- reset
         estado <= '0';
      elsif (rising_edge(clk)) then
         estado <= data xor estado;
      end if;
   end process;

   err <= estado; -- como queremos paridade par, enviamos "estado" para "err"

end Behavioral;
```

Parity Checker

```
library IEEE;
use  IEEE.STD_LOGIC_1164.ALL;

entity CounterLogic_4bit is
   Port ( operandA : in STD_LOGIC_VECTOR (3 downto 0);
          R : out STD_LOGIC_VECTOR (3 downto 0));
end CounterLogic_4bit;

architecture Structural of CounterLogic_4bit is

component adder_4bit is
   Port( A : in STD_LOGIC_VECTOR(3 downto 0);
         B : in STD_LOGIC_VECTOR(3 downto 0);
         Cin : in STD_LOGIC;
         S : out STD_LOGIC_VECTOR(3 downto 0);
         Cout : out STD_LOGIC);
end component;

begin

U0:   adder_4bit
      port map(A => operandA, B => "0001", Cin => '0', S => R, Cout => open);

end Structural;
```

Counter Logic

```
library IEEE;
use  IEEE.STD_LOGIC_1164.ALL;

entity register_D_R is
   generic(
      WIDTH : POSITIVE := 1
   );
   Port ( CLK : in STD_LogiC;
          D : in STD_Logic_VECTOR (WIDTH-1 downto 0);
          Q : out STD_Logic_VECTOR (WIDTH-1 downto 0);
          RST : in STD_LOGIC);
end register_D_R;

architecture Behavioral of register_D_R is

begin

   Q <= (others => '0') when RST = '1' else D when rising_edge(clk);

end Behavioral;
```

Register_D_R

Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

```
library IEEE;
use  IEEE.STD_LOGIC_1164.ALL;

entity adder_4bit is
    Port ( A : in STD_LogiC_VECTOR (3 downto 0);
           B : in STD_Logic_VECTOR (3 downto 0);
           Cin : in STD_Logic;
           S : out STD_Logic_VECTOR (3 downto 0);
           Cout : out STD_Logic);
end adder_4bit;

architecture Structural of adder_4bit is
component FA is
    Port ( A : in STD_Logic;
           B : in STD_Logic;
           Cin : in STD_Logic;
           S : out STD_Logic;
           Cout : out STD_Logic);
end component;

signal carry      : std_logic_vector(3 downto 1);
begin
U0:       FA
          port map(A => A(0), B => B(0), Cin => Cin, S => S(0), Cout => carry(1));

U1:       FA
          port map(A => A(1), B => B(1), Cin => carry(1), S => S(1), Cout => carry(2));

U2:       FA
          port map(A => A(2), B => B(2), Cin => carry(2), S=> S(2), Cout => carry(3));

U3:       FA
          port map(A => A(3), B => B(3), Cin => carry(3), S=> S(3), Cout => Cout);

end Structural;
```

adder_4bit

```
library IEEE;
use  IEEE.STD_LOGIC_1164.ALL;

entity FA is
    Port ( A : in STD_Logic;
           B : in STD_Logic;
           Cin : in STD_Logic;
           S : out STD_Logic;
           Cout : out STD_Logic);
end FA;

architecture Behavioral of FA is
    signal xor_ab   : std_logic;
    signal and_ab   : std_logic;
begin
    xor_ab    <= A xor B;
    and_Ab    <= A and B;

    S         <= xor_ab xor Cin;

    Cout      <= and_ab or ( Cin and xor_ab );

end Behavioral;
```

FA

Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

B. Descrição VHDL do bloco *Dispatcher*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Dispatcher is
    Port (
        RESET : in  STD_LOGIC;
        MCLK : in STD_LOGIC;
        Dval : in STD_LOGIC;
        Fsh : in STD_LOGIC;
        Din : in STD_LOGIC_VECTOR(9 downTo 0);
        Dout : out STD_LOGIC_VECTOR(8 downTo 0);
        WrT : out STD_LOGIC;
        WrL : out STD_LOGIC;
        Done : out STD_LOGIC
    );
end Dispatcher;

architecture Structural of Dispatcher is

component DispatcherControl is
    Port ( RESET : in STD_LOGIC;
           Fsh : in STD_LOGIC;
           MCLK : in  STD_LOGIC;
           Dval : in  STD_LOGIC;
           TnL : in  STD_LOGIC;
           WrT : out  STD_LOGIC;
           WrL : out  STD_LOGIC;
           Done : out STD_LOGIC
           );
end component;

begin
Dout(8) <= Din(9);
Dout(7) <= Din(8);
Dout(6) <= Din(7);
Dout(5) <= Din(6);
Dout(4) <= Din(5);
Dout(3) <= Din(4);
Dout(2) <= Din(3);
Dout(1) <= Din(2);
Dout(0) <= Din(1);

U0:     DispatcherControl
        port map(RESET => RESET, MCLK => MCLK, Fsh => Fsh, Dval => Dval, TnL => Din(0), WrT => WrT, WrL=> WrL,
        Done => Done);

end Structural;
```

Dispatcher

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DispatcherControl is
      Port (
         RESET : in STD_LOGIC;
         MCLK : in STD_LOGIC;
         Fsh : in STD_LOGIC;
         Dval : in  STD_LOGIC;
         TnL : in  STD_LOGIC;
         WrT : out  STD_LOGIC;
         WrL : out  STD_LOGIC;
         Done : out STD_LOGIC
      );
end DispatcherControl;

architecture Behavioral of DispatcherControl is

      type  STATE_TYPE is (
            ESTADO_INIT,
            ESTADO_WrT,
            ESTADO_WrL,
            ESTADO_DONE
      );

      signal CurrentState, NextState : STATE_TYPE;

begin

      CurrentState <= ESTADO_INIT when RESET = '1' else NextState when rising_edge(MCLK);


      GenerateNextState: process(CurrentState, Dval, TnL, Fsh)
      begin
     CurrentState <= ESTADO_INIT when RESET = '1' else NextState when rising_edge(MCLK);


     GenerateNextState: process(CurrentState, Dval, TnL, Fsh)
     begin

        case CurrentState is
        when ESTADO_INIT           => if(Dval = '1' and TnL = '1') then
                                         NextState <= ESTADO_WrT;
                                      elsif(Dval = '1' and TnL = '0') then
                                         NextState <= ESTADO_WrL;
                                      else
                                         NextState <= ESTADO_INIT;
                                      end if;

        when ESTADO_WrT            => if(Fsh = '1') then
                                         NextState <= ESTADO_DONE;
                                      else
                                         NextState <= ESTADO_WrT;
                                      end if;

        when ESTADO_WrL            => NextState <= ESTADO_DONE;

        when ESTADO_DONE           => NextState <= ESTADO_INIT;


        end case;
     end process;

   Done <= '1' when (CurrentState = ESTADO_DONE) else '0';
   WrT <= '1' when (CurrentState = ESTADO_WrT) else '0';
   WrL <= '1' when (CurrentState = ESTADO_WrL) else '0';

end Behavioral;
```

Dispatcher Control

Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

C. Descrição VHDL do bloco *Integrated Output System*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity IOS is
    Port (
        MCLK : in STD_LOGIC;
        RESET : in STD_LOGIC;
        SCLK : in STD_LOGIC;
        SDX: in STD_LOGIC;
        nSS : in STD_LOGIC;
        Fsh : in STD_LOGIC;
        WrT : out STD_LOGIC;
        WrL : out STD_LOGIC;
        busy : out STD_LOGIC;
        Dout : out STD_LOGIC_VECTOR(8 downTo 0)
    );

end IOS;

architecture Structural of IOS is

component Dispatcher is
    Port ( MCLK : in  STD_LOGIC;
        RESET : in STD_LOGIC;
        Dval : in STD_LOGIC;
        Fsh : in STD_LOGIC;
        Din : in STD_LOGIC_VECTOR(9 downTo 0);
        Dout : out STD_LOGIC_VECTOR(8 downTo 0);
        WrT : out STD_LOGIC;
        WrL : out STD_LOGIC;
        Done : out STD_LOGIC
            );
end component;

component SerialReceiver is
    Port (SDX : in STD_LOGIC;
          SCLK : in STD_LOGIC;
          nSS : in STD_LOGIC;
          MCLK : in STD_LOGIC;
          RESET : in STD_LOGIC;
          accept : in STD_LOGIC;
          busy : out STD_LOGIC;
          DXval : out STD_LOGIC;
          Dout : out STD_LOGIC_VECTOR(9 downTo 0)
          );
end component;

signal IOS_D : std_logic_vector(9 downTo 0);
signal IOS_DXval : std_logic;
signal IOS_Done : std_logic;

begin

U0:       Dispatcher
          port map(RESET => RESET, MCLK => MCLK, Dval => IOS_DXval, Fsh => Fsh, Din => IOS_D, Dout => Dout,
          WrT => WrT, WrL => WrL, Done => IOS_Done);

U1:       SerialReceiver
          port map(MCLK => MCLK, RESET => RESET, SDX => SDX, SCLK => SCLK, nSS => nSS, accept => IOS_DONE,
          busy => busy, DXval => IOS_DXval, Dout => IOS_D);

end Structural;
```

IOS

Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

D. Descrição VHDL do bloco *Ticket Dispenser*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity TicketDispenser is
    Port (
        FnINT : in STD_LOGIC;
        Prt : in STD_LOGIC;
        RT : in STD_LOGIC;
        DId : in STD_LOGIC_VECTOR(3 downTo 0);
        OId : in STD_LOGIC_VECTOR(3 downTo 0);
        RT_out : out STD_LOGIC_VECTOR(7 downTo 0);
        DId_out : out STD_LOGIC_VECTOR(7 downTo 0);
        OId_out : out STD_LOGIC_VECTOR(7 downTo 0);
        Fn : out STD_LOGIC;
        Print : out STD_LOGIC
    );

end TicketDispenser;

architecture Structural of TicketDispenser is

component decoderHex is
        Port(A: in std_logic_vector(3 downto 0);
        clear : in std_logic;
        HEX0 : out std_logic_vector(7 downto 0)
        );
end component;

signal RT_vector : std_logic_vector(3 downTo 0);
signal nPRT : std_logic;

begin

RT_vector(0) <= RT;
RT_vector(1) <= '0';
RT_vector(2) <= '0';
RT_vector(3) <= '0';
Print <= Prt;
nPrt <= not Prt;
Fn <= FnINT;

U0:   decoderHex
      port map(A => RT_vector, clear => (nPrt), HEX0 => RT_out);


U1:   decoderHex
      port map(A => OId, clear => (nPrt), HEX0 => OId_out);


U2:   decoderHex
      port map(A => DId, clear => (nPrt), HEX0 => DId_out);

end Structural;
```

Ticket Dispenser

Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

E. Código Kotlin - *HAL*

```kotlin
object HAL {
    var lastOutput = 0
    fun init() {
        UsbPort.write( val: 0)
    }

    fun isBit(mask: Int): Boolean {
    val value = UsbPort.read()
    val final = value and mask
    if (final != 0) return true else return false
    }

    fun readBits(mask: Int): Int {
        val value = UsbPort.read()
        val final = value and mask
        return final
    }

    fun writeBits(mask: Int, value: Int) {
        var finalOutput = 0
        finalOutput = (mask.inv() and lastOutput) or (value and mask)
        lastOutput = finalOutput
        UsbPort.write(finalOutput)
    }

    fun setBits(mask: Int) {
        var finalOutput = 0
        finalOutput = lastOutput or mask
        lastOutput = finalOutput
        UsbPort.write(finalOutput)
    }
    fun clrBits(mask: Int) {
        var finalOutput = 0
        finalOutput = mask.inv() and lastOutput
        lastOutput = finalOutput
        UsbPort.write(finalOutput)
    }
}
```

HAL

Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

F. Código Kotlin – *SerialEmitter*

```
bject SerialEmitter {

    enum class Destination {
        LCD, TICKET_DISPENSER
    }
    private var send_MASK = 0

    private val nSS_MASK = 1 shl 2

    private val SDX_MASK = 1 shl 1

    private val SCLK_MASK = 1

    private val BUSY_MASK = 1

    private var FRAME_LENGTH = 10

    fun init () {   //inicia classe e mete respetivos bits a estado inicial
        HAL.init()
        HAL.setBits(nSS_MASK)
    }

    fun send(addr: Destination, data: Int) {

        while (isBusy()) {
        }

        var mMask = 0

        if (addr == Destination.LCD) {
            mMask = 0 shl FRAME_LENGTH
        }   else if (addr == Destination.TICKET_DISPENSER) {
            mMask = 1 shl FRAME_LENGTH
        }

        send_MASK = (send_MASK or mMask) or data

        val parityMask = if (!parityCheck(send_MASK)) 1 shr FRAME_LENGTH else 0 shr FRAME_LE
        //se sendMask for impar o meu parity bit ira ser 1 para os 11 bits ficarem par
        send_MASK = send_MASK or parityMask

        HAL.clrBits(nSS_MASK)   //inicia o serial receiver para receber bits
        for (i in 0..FRAME_LENGTH) {    //mandar SDX respetivo para placa
            HAL.clrBits(SCLK_MASK)
            Thread.sleep( millis: 2000)
            if (send_MASK and (1 shl i) != 0) HAL.setBits(SDX_MASK) else HAL.clrBits(SDX_MASK
            HAL.setBits(SCLK_MASK)
            Thread.sleep( millis: 2000)
        }

        HAL.setBits(nSS_MASK)   //acabo transmissao para serial receiver
    }

    private fun parityCheck(data: Int): Boolean {
        var numOnes = 0
        var tempMASK = 1
        for (i in 0 until FRAME_LENGTH) {
            if (data and tempMASK != 0) numOnes ++
            tempMASK = tempMASK shl 1
        }
        return numOnes % 2 == 0
    }

    /** Retorna true se o canal série estiver ocupado  */
    fun isBusy(): Boolean {
        return HAL.isBit(BUSY_MASK)
```

SerialEmitter

Autores: Pedro Miguens Matutino (pedro.miguens@isel.pt)
Diego Passos (diego.passos@isel.pt)
Manuel Carvalho (manuel.carvalho@isel.pt)
Nuno Sebastião (nuno.sebastiao@isel.pt)

G. Código Kotlin – *TicketDispenser*

```kotlin
object TicketDispenser {
    // Inicia a classe, estabelecendo os valores iniciais.
    fun init() {
    SerialEmitter.init()
    }
    // Envia comando para imprimir e dispensar um bilhete
    fun print(destinyId: Int, originId: Int, roundTrip: Boolean) {
        val RT:Int
        if (roundTrip) RT = 1 else RT = 0
        var data = RT or destinyId.shl( bitCount: 1) or originId.shl( bitCount: 5) //meter bits na posição final para enviar
        SerialEmitter.send(SerialEmitter.Destination.TICKET_DISPENSER, data)
    }
}
```

Ticket Dispenser (Software)