ISEL
DEETC
Departamento de Engenharia Electrónica e de Telecomunicações e de Computadores

Licenciatura em Engenharia Informática e de Computadores
Ano letivo 2021/2022

Máquina de venda de bilhetes

DOCUMENTAÇÃO MÓDULO INTEGRATED OUTPUT SYSTEM

Docentes responsáveis:
Pedro Miguens — Manuel Carvalho
Diego Passos — Nuno Sebastião
Trabalho realizado por:
Francisco Engenheiro — Nº49428
João Perleques — Nº49498
Mariana Muñoz — Nº47076
Turmas 22D/23D

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.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.

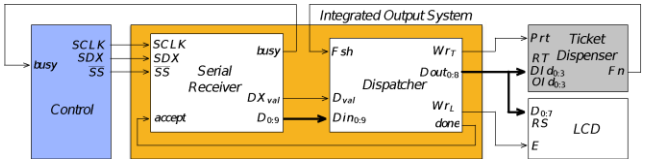

Figura 1 – Diagrama de blocos do *Integrated Output System*

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*.

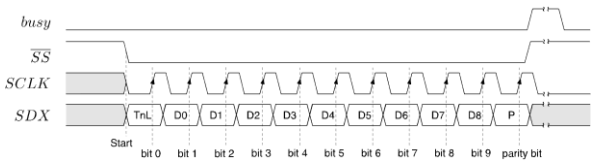

Figura 2 – Protocolo de comunicação com o módulo Integrated Output System

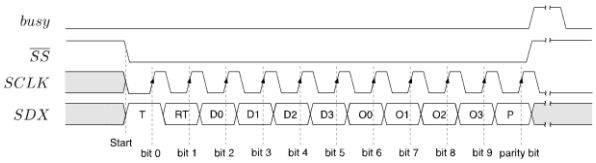

Figura 3 – Trama para o mecanismo de dispensa de bilhetes (Ticket Dispenser)

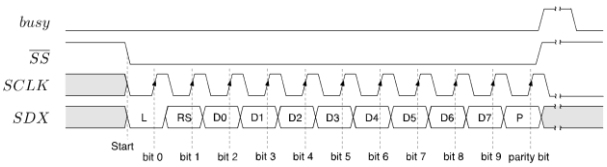

Figura 4 – Trama para o LCD

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 *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 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.

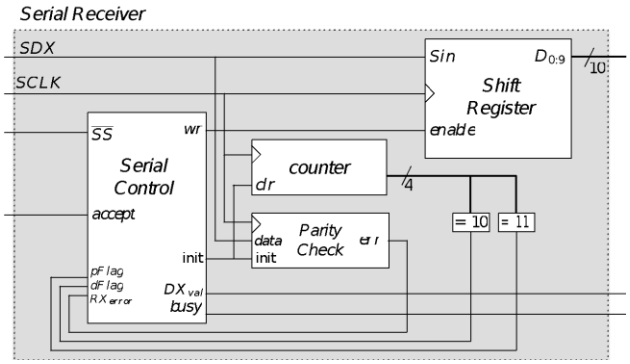

Figura 5 – Diagrama de blocos do bloco Serial Receiver

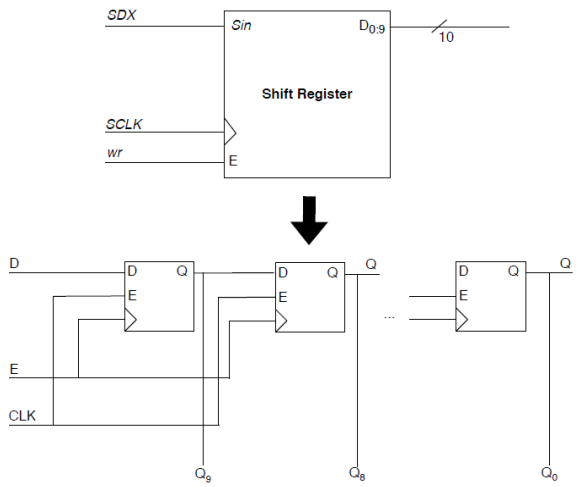

Figura 6 - Diagrama de blocos do bloco Shift-Register

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

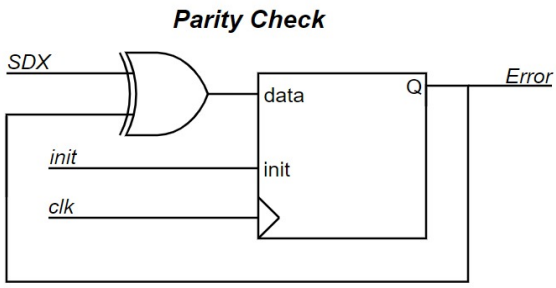

Figura 7 - Diagrama de blocos do bloco Parity Check

O bloco apresentado anteriormente, faz a acumulação do cálculo do erro, bit a bit, recorrendo à operação lógica *XOR* entre a *DATA* (bit da trama) e o *Q_LINK* (correspondente ao erro anterior) e que fica guardado no sinal *D_LINK*.

O valor lógico em *D_LINK* entra no flip-flop do tipo *Edge Triggered*, e na próxima transição ascendente do relógio *SCLK*, passa para *Q_LINK*, sendo esta a saída. Deste modo, atualiza assim o último erro calculado. A flag *ERR* é, portanto, o valor que *Q_LINK* tiver no momento, representando o erro com um ciclo de relógio de atraso.

O bloco *Serial Control* do bloco *Serial Receiver* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 8.

Este bloco foi idealizado para representar 4 estados (inicialmente): um estado zero (***STATE_INIT***) que ativa a saída *init*, responsável por reiniciar a contagem presente no *Counter* e por reiniciar o cálculo do erro no *Parity Check*; um primeiro estado (***STATE_WRITE***) que ativa a saída *wr*, sendo esta responsável pela ativação do enable do *Shift-Register*, possibilitando assim que os bits em *SDX* sejam colocados no barramento de dados *D*, 10 bits correspondentes à trama sem o parity bit, que o *COUNTER* sinaliza o fim dessa contagem pela ativação da *dFlag*. Ainda neste estado se for detetado *nSS* com o valor lógico '1', a trama é descartada. O segundo estado (***STATE_OFF***) é responsável pela desativação do *wr* (de forma a não colocar o *parity bit* na saída *D*), e permanece neste estado até que o Control sinalize que já enviou a trama completa, se neste ponto tiver sido contabilizado 11 bits (indicado pela ativação da *pFlag*) e não existir erro (indicado pela flag RXerror) é passado para o estado 3, no entanto se tal não acontecer significa que *Control* enviou uma trama com mais ou menos bits do que os que estão presentes no protocolo de comunicação e por isso a trama é imediatamente descartada, o mesmo acontece quando um erro é detetado; O terceiro estado (***STATE_END***) *é* responsável pela ativação das saídas *Dxval* (para sinalizar ao *Dispatcher* a presença de uma trama válida) e *busy* (que indica ao *Control* que o *Dispatcher* está a processar ainda a trama enviada), a flag *accept* fica ativa para indicar ao *Serial Receiver* que o *Dispatcher* já processou a trama anterior, que consequentemente, no próximo estado irá desativar o sinal *busy*, indicando ao *Control* que o *Serial Receiver* está pronto para receber mais uma trama.

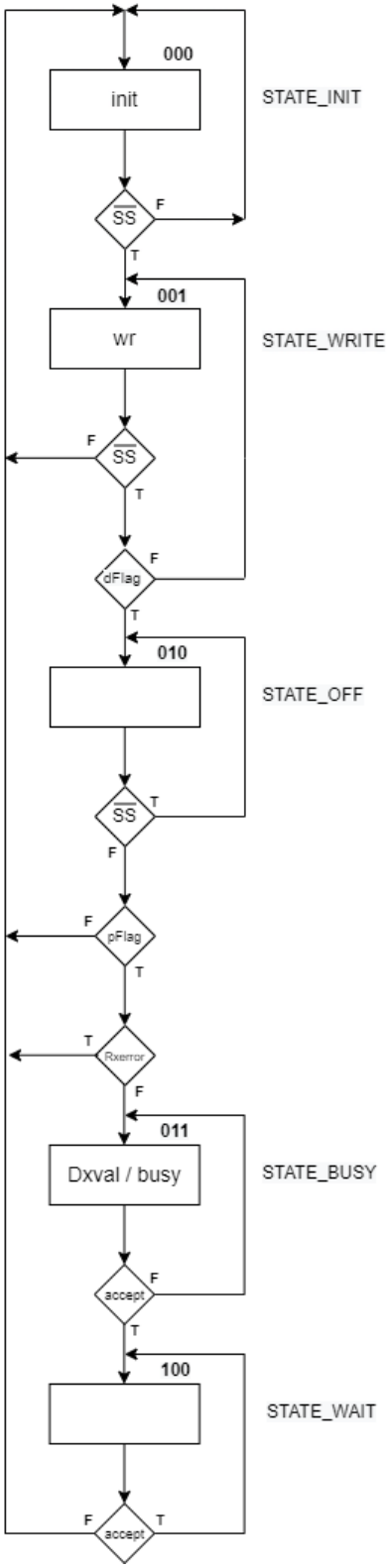

Figura 8 – Máquina de estados do bloco Serial Control

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

No entanto, a lógica atual só funciona se os *MCLKs* respetivos dos módulos *Dispatcher* e *Serial Receiver* apresentarem a mesma frequência. Para conseguir continuar a garantir a sincronização entre os módulos, noutras situações, foi utilizada a técnica de *full interlock*, e, portanto, foi adicionado um quarto estado (**STATE_WAIT**), que, tirando a falta de um melhor nome, é responsável por desativar o sinal *busy*/*Dxval*, permitindo à máquina de estados do *Dispatcher* evoluir de estado e consequentemente desativar a flag *accept/done*, o que por usa vez permite à máquina de estados do *Serial Receiver* de voltar ao estado inicial, garantido assim a sincronização dos dois módulos, mesmo com frequências diferentes. Se tal não fosse tido em conta, o módulo IOS poderia ficar bloqueado, consequentemente bloqueando o resto do sistema.

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

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*.

O bloco *Dispatcher* foi implementado de acordo com o diagrama de blocos representado na Figura 9.

No processo de planeamento da construção dos blocos para o bloco *Dispatcher* entendeu-se que, a construção do bloco *Dispatcher Control* e utilização de um contador auxiliar de 4 bits, era suficiente para conseguir representar as funcionalidades deste bloco.

O bloco *Dispatcher Control* foi implementado pela máquina de estados representada em *ASM-chart* na Figura 10.

O bloco foi idealizado para representar 6 estados: um estado zero (***STATE_RECEIVE***), de modo a ficar dependente da ativação da entrada *Dval* (que representa a sinalização por parte do *Serial Receiver* que este contém uma trama válida para enviar); um primeiro (***STATE_WRITE_TD)*** e um segundo (***STATE_RST_COUNTER)*** estados, que, usando a entrada *Tnl (*representada na trama pelo *LSB)*, faz o encaminhamento recorrendo ao seu valor lógico 1 e 0, para um destes estados, respetivamente; um terceiro estado (***STATE_WRITE_LCD***) que ativa o *enable* do *LCD* até que o contador interno chegue ao valor 15 (15x20ns=300ns), ativando a flag *cFlag*, visto que é preciso garantir, no mínimo, 230 ns de duração da ativação do *enable*, para que seja possível aceder ao *LCD* com sucesso; um quarto estado (***STATE_TICKET_EXIT***) para avaliar quando o bilhete é retirado pelo cliente (desativação do sinal *Fsh*); um quinto estado (***STATE_DONE***), que é responsável pela ativação do sinal *done*, de forma a sinalizar o *Serial Receiver* que o *Dispatcher* já processou a trama, e que só evolui deste estado quando o sinal *Dval* é desativo, ou seja, quando o sinal busy e Dxval são desativados pelo *Serial Receiver*, permitindo assim o *full interlock* entre os 2 módulos, como foi explicado no ponto 1.

De referir que o *address setup* time de, no mínimo, 40ns, já é garantido pelo tempo que as tramas demoram a ser enviadas do software para o hardware, e por isso não foram feitas mais alterações aos blocos apresentados.

A descrição hardware do bloco *Dispatcher* em VHDL encontra-se no Anexo B.

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.

Na escolha da frequência de relógios para os diferentes blocos que compõem o IOS, chegou-se à conclusão de que o relógio utilizado para o bloco *Serial Control*, denominado por *MCLK* e mapeado no relógio da placa *MAX 10 LITE* de 50 MHz, seria o mesmo para o *Dispatcher Control*, com a mesma fase, de forma a existir um sincronismo entre a mudança de estados de ambas as máquinas de estados, diminuindo assim o atraso na propagação de informação de um bloco para o outro. Seguindo este raciocínio, o relógio controlado pelo *Control*, o *SCLK*, teria de ter uma frequência menor que o *MCLK*, para garantir que as máquinas de estados dos blocos referidos anteriormente têm tempo suficiente para atualizar o seu estado antes das alterações nos seus módulos provocados pelo *SCLK*, no entanto como foi referido em diversos pontos anteriormente, a forma como estão construídos ambos os módulos, o facto de terem a mesma frequência não é imperativo.

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

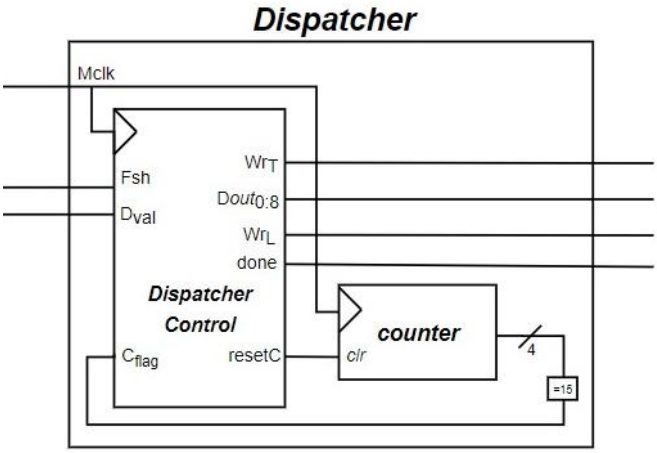

Figura 9 - Diagrama de blocos do bloco Dispatcher

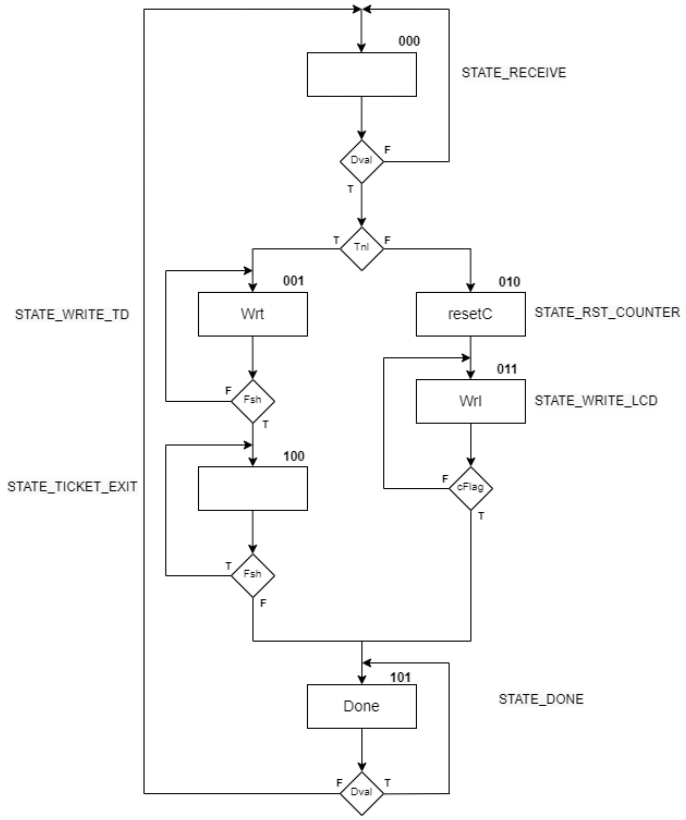

Figura 10 - Máquina de estados do bloco Dispatcher Control

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 11.

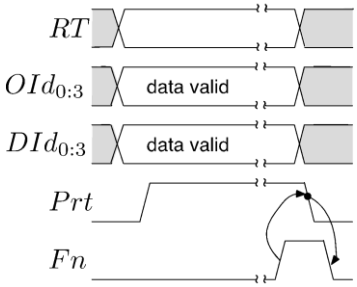

Figura 11- Diagrama temporal do mecanismo de dispensa de bilhetes

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

4 Interface com o *Control*

Implementou-se o módulo *Control* em *software*, recorrendo a linguagem *Kotlin* seguindo a arquitetura lógica apresentada na Figura 12.

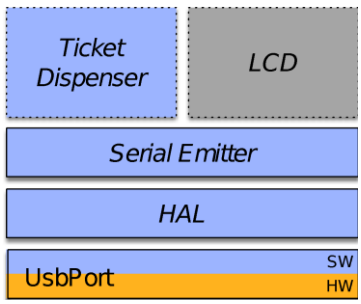

Figura 12 – Diagrama lógico do módulo *Control* de interface com o módulo *Keyboard Reader*

Os módulos de software *HAL*, *Serial Emitter* e *Ticket Dispenser* desenvolvidos são descritos nas secções 2.1, 2.2 e 2.3, e o código fonte desenvolvido nos Anexos E, F e G, respetivamente.

4.1 *HAL*

Esta classe foi construída como objeto, visto que, não é pretendido criar mais do que uma instância para a sua utilização. Deste modo, deve existir apenas uma que comunicará diretamente com o módulo *USBPort* e as funções que este representa.
Este objeto, recorrendo aos seus métodos, é responsável, usando uma máscara que representa os bits que se quer alterar, pela leitura de 1 ou mais bits em simultâneo que estejam presentes no *inputport* do *UsbPort*, e da escrita de 1 ou mais bits em

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

simultâneo para o *outputport* do *UsbPort*, escolhendo entre colocar todos os bits representados no valor lógico 0 ou 1, ou colocar especificamente uns a 0 e outros a 1.
Em relação aos métodos já existentes no *HAL* provenientes do enunciado, não foram adicionados mais.

4.2 SerialEmitter

Esta classe também foi construída como objeto, para que não existam várias instâncias do *SerialEmitter* a mandar tramas para o *Serial Receiver*, mas apenas uma.
Este objeto é responsável, pelo envio de tramas para o *Serial Receiver*, para esse efeito avalia o sinal busy. Se este estiver no valor lógico 0, coloca o SS no valor lógico 0. Antes do envio, a trama é construída de acordo com o Protocolo de comunicação representado na Figura *2*. Para esse objetivo, além de operações lógicas necessárias para a construção da trama, foi acrescentado uma função para calcular o *parity bit*, recorrendo a um algoritmo genérico, uma função de extensão para avaliar o último bit e uma função, para dar um ciclo ao relógio *SCLK* sempre que é chamada.

4.3 *TicketDispenser*

Esta classe também foi construída como objeto, para que não existam várias instâncias do *TicketDispenser* a mandar tramas para o *SerialEmitter*, mas apenas uma.
Este objeto é responsável, pelo envio das informações necessárias, entre as quais: a estação de destino e de origem, e o tipo de bilhete, para o SerialEmitter. De modo que, este possa construir a restante trama. Em relação aos métodos já existentes no *TicketDispenser* provenientes do enunciado, não foram adicionados mais.

5 Conclusões

A introdução dos testes para cada módulo criado ajudou, a incutir o conceito de modularização, portanto, mais tarde se for preciso alterar algum módulo, não será preciso alterar todos os que foram criados, entretanto.
No entanto, na fase de planeamento e desenvolvimento dos diferentes blocos que constituem o IOS, ocorreu alguns percalços, devido à falta de conhecimento e perceção da melhor forma de chegar ao resultado pretendido sem duvidar do que foi feito até ao momento. Recorremos por diversas vezes, a correções e ajustes em módulos que se encontravam previamente validados. Contudo, o grupo tem noção que essa é uma capacidade a adquirir na unidade curricular que este projeto se integra e que por isso continua a trabalhar para que a mesma seja apreendida.
De referir que, o bloco que verificamos mais dificuldade no desenvolvimento foi o *Serial Receiver*, devido a constantes alterações noutros blocos que o compõem, nomeadamente o *Parity Check* e o *Shift-Register*.
Este relatório permitiu que ficasse claro a importância e a necessidade de descrever e documentar a informação pertinente, no momento de desenvolvimento de módulos em software e hardware. De forma que, outros possam perceber de forma sucinta e clara, e replicar com sucesso o que foi feito.
Por fim, é importante referir que um projeto como este, proporcionou curiosidade nos elementos do grupo na aprendizagem de novos conceitos relacionados com as temáticas lecionadas, preparando-nos para o que esperar no mercado de trabalho e consequentemente vontade de integrar outros projetos futuramente.

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

A. Descrição VHDL do bloco *Serial Receiver*

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

entity SERIAL_RECEIVER is

    port( SDX, MCLK, SCLK, nSS, accept, RESET: in STD_LOGIC;
          D: out STD_LOGIC_VECTOR(9 downto 0);
          DXval, busy: out STD_LOGIC );

end SERIAL_RECEIVER;

architecture arq_SERIAL_RECEIVER of SERIAL_RECEIVER is

component SHIFTREGISTER

    port( Sin, SCLK, EN, RESET: in STD_LOGIC;
          D: out STD_LOGIC_VECTOR(9 downto 0) );

end component;

component COUNTER4BITS

    port( CLR, SCLK, EN: in STD_LOGIC;
          O: out STD_LOGIC_VECTOR(3 downto 0) );

end component;

component SERIAL_CONTROL

    port( nSS, accept, dFlag, pFlag, RXerror, RESET, MCLK: in STD_LOGIC;
          wr, init, DXval, busy: out STD_LOGIC );

end component;

component PARITY_CHECK

    port( DATA, SCLK, INIT: in STD_LOGIC;
          ERR: out STD_LOGIC);

end component;

signal COUNTER_LINK: STD_LOGIC_VECTOR(3 downto 0);
signal dFlag_LINK, pFlag_LINK, INIT_LINK, ERR_LINK, WR_LINK: STD_LOGIC;

begin

    M0: SHIFTREGISTER port map (
        Sin => SDX,
        SCLK => SCLK,
        EN => WR_LINK,
        RESET => RESET,
        D => D );

    M1: COUNTER4BITS port map (
        CLR => INIT_LINK,
        SCLK => SCLK,
        EN => '1',
```

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

```
        O => COUNTER_LINK );

    M2: PARITY_CHECK port map (
        DATA => SDX,
        SCLK => SCLK,
        INIT => INIT_LINK,
        ERR => ERR_LINK );

    M3: SERIAL_CONTROL port map (
        nSS => nSS,
        RESET => RESET,
        accept => accept,
        dFlag => dFlag_LINK,
        pFlag => pFlag_LINK,
        RXerror => ERR_LINK,
        MCLK => MCLK,
        wr => WR_LINK,
        init => INIT_LINK,
        DXval => DXval,
        busy => busy );

    -- Active flag when counter equals to 10 (1010)
    dFlag_LINK <= COUNTER_LINK(3) and not COUNTER_LINK(2) and COUNTER_LINK(1) and
not COUNTER_LINK(0);


    -- Active flag when counter equals to 11 (1011)
    pFlag_LINK <= COUNTER_LINK(3) and not COUNTER_LINK(2) and COUNTER_LINK(1) and
COUNTER_LINK(0);


end arq_SERIAL_RECEIVER;
```

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

B. Descrição VHDL do bloco *Dispatcher*

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

entity DISPATCHER is

    port( MCLK, Fsh, Dval, RESET: in STD_LOGIC;
          Din: in STD_LOGIC_VECTOR(9 downto 0);
          Wrt, Wrl, done: out STD_LOGIC;
          Dout: out STD_LOGIC_VECTOR(8 downto 0) );

end DISPATCHER;

architecture arq_DISPATCHER of DISPATCHER is

component DISPATCHER_CONTROL

    port( MCLK, Fsh, Dval, RESET, cFlag: in STD_LOGIC;
          Din: in STD_LOGIC_VECTOR(9 downto 0);
          Wrt, Wrl, done, resetC: out STD_LOGIC;
          Dout: out STD_LOGIC_VECTOR(8 downto 0) );

end component;

component COUNTER4BITS

    port( CLR, SCLK, EN: in STD_LOGIC;
          O: out STD_LOGIC_VECTOR(3 downto 0) );

end component;

signal COUNTER_LINK: STD_LOGIC_VECTOR(3 downto 0);
signal resetC_LINK, cFlag_LINK: STD_LOGIC;

begin

    M0: DISPATCHER_CONTROL port map (
        MCLK => MCLK,
        Fsh => Fsh,
        Dval => Dval,
        RESET => RESET,
        cFlag => cFlag_LINK,
        Din => Din,
        Wrt => Wrt,
        Wrl => Wrl,
        done => done,
        resetC => resetC_LINK,
        Dout => Dout );

    M1: COUNTER4BITS port map (
        CLR => resetC_LINK,
        SCLK => MCLK,
        EN => '1',
        O => COUNTER_LINK );

    -- Active flag when counter equals to 15 (1111)
    cFlag_LINK <= COUNTER_LINK(3) and COUNTER_LINK(2) and  COUNTER_LINK(1) and
COUNTER_LINK(0);
```

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

```
end arq_DISPATCHER;
```

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

C. Descrição VHDL do bloco *Integrated Output System*

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

entity IOS is

    port( MCLK, RESET, SCLK, SDX, nSS, Fsh: in STD_LOGIC;
          Dout: out STD_LOGIC_VECTOR(8 downto 0);
          busy, Wrl, Wrt: out STD_LOGIC );

end IOS;

architecture arq_IOS of IOS is

component SERIAL_RECEIVER

    port( SDX, MCLK, SCLK, nSS, accept, RESET: in STD_LOGIC;
          D: out STD_LOGIC_VECTOR(9 downto 0);
          DXval, busy: out STD_LOGIC );

end component;

component DISPATCHER

    port( MCLK, Fsh, Dval, RESET: in STD_LOGIC;
          Din: in STD_LOGIC_VECTOR(9 downto 0);
          Wrt, Wrl, done: out STD_LOGIC;
          Dout: out STD_LOGIC_VECTOR(8 downto 0) );

end component;

signal DXval_LINK, done_LINK: STD_LOGIC;
signal D_LINK: STD_LOGIC_VECTOR(9 downto 0);

begin

    SLR: SERIAL_RECEIVER port map (
        SDX => SDX,
        MCLK => MCLK,
        SCLK => SCLK,
        nSS => nSS,
        accept => done_LINK,
        RESET => RESET,
        D =>  D_LINK,
        DXval => DXval_LINK,
        busy => busy );

    DPTR: DISPATCHER port map (
        MCLK => MCLK,
        Fsh => Fsh,
        Dval => DXval_LINK,
        RESET => RESET,
        Din => D_LINK,
        Wrt => Wrt,
        Wrl => Wrl,
        done => done_LINK,
        Dout => Dout );
```

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

```
end arq_IOS;
```

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

D. Descrição VHDL do bloco *Ticket Dispenser*

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.all;

entity TICKET_DISPENSER is

    port ( Prt, CollectTicket: in STD_LOGIC;
           Dout: in STD_LOGIC_VECTOR(8 downto 0);
           Fn: out STD_LOGIC;
           HEX0, HEX1, HEX2, HEX3, HEX4, HEX5: out STD_LOGIC_VECTOR(7 downto 0) );

end TICKET_DISPENSER;

architecture arq_TICKET_DISPENSER of TICKET_DISPENSER is

component DECODERHEX

    port ( A: in STD_LOGIC_VECTOR(3 downto 0);
           ewr: in STD_LOGIC_VECTOR(7 downto 0);
           clear: in STD_LOGIC;
           HEX0: out STD_LOGIC_VECTOR(7 downto 0) );

end component;

signal RT: STD_LOGIC;
signal RTtoBDC, O, D: STD_LOGIC_VECTOR(3 downto 0);

begin

    -- RT bit indicates either is a one-way ticket (1) or a two-way ticket (0)
    RT <= Dout(0) when (Prt = '1') else '0';
    -- Destination Station Code
    D <= Dout(4 downto 1) when (Prt = '1') else "0000";
    -- Origin Station Code
    O <= Dout(8 downto 5) when (Prt = '1') else "0000";

    -- Activate Fn when client retrieves the printed ticket
    Fn <= CollectTicket;

    -- Convert RT bit to BDC
    RTtoBDC <= "0000" when RT = '0' else "0001";

    -- Using the six 7 segment displays available on the DE10-Lite Board
    -- produce a valid array of numbers and letters which the client can understand
    -- Valid format example: [b0o1d4]
    -- Means: "bilhete de ida e volta, da estação de origem com o código (0001) para
a estação de destino com o código (0100)"
    -- Implementation on the six 7 displays segment, facing the board, starting from
the right to the left:
    U0: DECODERHEX port map (
        A => D,
        ewr => "11111111",
        clear => '0',
        HEX0 => HEX0 );

    U1: DECODERHEX port map (
        -- Always "d"
        A => "1101",
        ewr => "11111111",
```

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

```
        clear => '0',
        HEX0 => HEX1 );

    U2: DECODERHEX port map (
        A => O,
        ewr => "11111111",
        clear => '0',
        HEX0 => HEX2 );

    U3: DECODERHEX port map (
        A => "0000",
        -- A value assignment is redundant here, since the value in ewr will
overwrite it, because its different from "11111111"
        -- Always "o"
        ewr => "10100011",
        clear => '0',
        HEX0 => HEX3 );

    U4: DECODERHEX port map (
        -- RT bit in BDC
        A => RTtoBDC,
        ewr => "11111111",
        clear => '0',
        HEX0 => HEX4 );

    U5: DECODERHEX port map (
        -- Always "b"
        A => "1011",
        ewr => "11111111",
        clear => '0',
        HEX0 => HEX5 );

end arq_TICKET_DISPENSER;
```

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

E. Código Kotlin – *HAL*

```kotlin
package ticketMachine

import isel.leic.UsbPort

// Virtualizes the access for the UsbPort system
object HAL {
    // Creates a mutable variable to store the value of the last output written
    private var lastOutput = 0
    // Initializes this class, which means to have a predefied output when
UsbPort.write is executed
    fun init() {
        // Writes lastOutput value on the output of the board
        UsbPort.write(lastOutput)
    }
    // Returns true when a bit we want to evaluate is set to logical '1'
    fun isBit(mask: Int): Boolean {
        // Reads the current input on the board
        val currentInput = UsbPort.read()
        // Performs an AND logic operation, bit by bit, with the current input and
the received mask
        // If it returns the mask (true), means that the bit we want to evaluate is
indeed set to logical '1'
        // otherwise returns false, indicating that bit is set to logical '0'
        return currentInput.and(mask) == mask
    }
    // Returns the values of the bits represented by the mask in the UsbPort
    fun readBits(mask: Int): Int {
        // Reads the current input on the board
        val currentInput = UsbPort.read()
        // Returns the result of an AND logic operation, bit by bit, between the
currentInput and the mask
        return currentInput.and(mask)
    }
    // Writes in the bits represented by the mask the given value
    fun writeBits(mask: Int, value: Int) {
        // Inverts mask bits
        val invertedMask: Int = mask.inv()
        // This secondMask variable will have the bits we want to keep and the bits
we want to
        // change set to logical '0'
        val secondMask: Int = lastOutput.and(invertedMask)
        // For the output, we want to merge the bits represented by the value with
the bits in secondMask
        lastOutput = secondMask.or(value)
        UsbPort.write(lastOutput)
    }
    // Sets bits represented by the mask as logical '1'
    fun setBits(mask: Int) {
        // Writes on the output of the board the result of an OR logic operation,
bit by bit,
        // between the lastOutput and the mask
        lastOutput = lastOutput.or(mask)
        UsbPort.write(lastOutput)
    }
```

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

```
    // Sets bits represented by the mask as logical '0'
    fun clrBits(mask: Int) {
        // Inverts mask bits
        val invertedMask: Int = mask.inv()
        // Writes on the output of the board the result of an AND logic operation,
bit by bit,
        // between the lastOutput and the invertedMask
        lastOutput = lastOutput.and(invertedMask)
        UsbPort.write(lastOutput)
    }
}
```

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

F. Código Kotlin – *SerialEmitter*

```
package ticketMachine

// Send frames for the different modules of the Serial Receiver
object SerialEmitter {
    // Constant values for the masks used in the inputport of UsbPort
    private const val BUSY_MASK = 0b00000001 // inputPort(0) of UsbPort
    // Constant values for the masks used in the outputport of UsbPort
    private const val SDX_MASK  = 0b00000001 // outputPort(0) of UsbPort
    private const val SCLK_MASK = 0b00000010 // outputPort(1) of UsbPort
    private const val SS_MASK   = 0b00000100 // outputPort(2) of UsbPort
    // This enum class will help declare for which module we want to send the frames
    enum class Destination { LCD, TICKET_DISPENSER }
    // Initializes a default mask to always target the first bit (bit 0)
    private const val LSB_MASK = 0b00000001
    // Initialize a flag to indicate when printing on console is required
    var EN_PRINT: Boolean = false
    // Initializes this class
    fun init() {
        // Set SDX value with logical '0'
        HAL.clrBits(SDX_MASK)
        // Set SCLK value with logical '0'
        HAL.clrBits(SCLK_MASK)
        // Set SS value with logical '1'
        HAL.setBits(SS_MASK)
        // Disables this function to print on console
        EN_PRINT = false
    }
    // Sends a frame to SerialReceiver identifying the destination with "addr" and
the
    // bits containing the data with "data", which has a total of 9 bits
    fun send(addr: Destination, data: Int) {
        // Waits for Busy signal to be disabled
        while (isBusy());
        // Set SS value with logical '0'
        HAL.clrBits(SS_MASK)
        // Build frame with 10 bits while making Destination bit LSB
        val din = (data.shl(1)).or(addr.ordinal)
        // Find parity bit
        val parityBit = findParityBit(din)
        // Full frame with MSB as the parity Bit
        var frame: Int = (parityBit.shl(10)).or(din)
        // With the full frame built, start a for loop to send each bit in every
ascending transition of SCLK
        for (i in 10 downTo 0) {
            // Evaluate if a frame has at least one bit set to logical '1'
            if (frame >= 1) {
                // Use checkLSB extension function to evaluate if the LSB of the
frame
                // is set to either '1' (true) or '0' (false)
                if (frame.checkLSB()) {
                    // Set SDX value with logical '1'
                    HAL.setBits(SDX_MASK)
                    if (EN_PRINT) println("SDX: 1")
                } else {
```

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

```
                    // Set SDX value with logical '0'
                    HAL.clrBits(SDX_MASK)
                    if (EN_PRINT) println("SDX: 0")
                }
                // Shift frame 1 bit to the right, this way discarding the LSB
                frame = frame.shr(1)
            } else {
                // Set SDX value with logical '0'
                HAL.clrBits(SDX_MASK)
                if (EN_PRINT) println("SDX: 0")
            }
            // Enables 1 clock cycle for SCLK
            enableSCLKCycle()
        }
        // Set SS value with logical '1'
        HAL.setBits(SS_MASK)
    }
    // Enables 1 clock cycle for SCLK
    private fun enableSCLKCycle() {
        // Set SCLK value with logical '1'
        HAL.setBits(SCLK_MASK)
        // Set SCLK value with logical '0'
        HAL.clrBits(SCLK_MASK)
    }
    // Returns true if busy, hardware output, is set to logical '1'
    fun isBusy(): Boolean = HAL.isBit(BUSY_MASK)
    // All the bits set to logical '1' in din have to be checked in order to assert
the
    // parity bit (MSB of the frame) with either 0 or 1, this way preserving the
agreed even parity.
    // This algorithm, which is generic, will find the parity bit while
consecutively compressing the initial number
    // with unassigned right shifts which will become smaller and smaller until the
end of din is reached
    // At that point, the parity bit can be found at the LSB
    // Time complexity: O(log(n)), whereas n is the number of bits of din
    private fun findParityBit(din: Int): Int {
        var d: Int = din
        // d = d xor (d.ushr(32))
        d = d xor (d.ushr(16))
        d = d xor (d.ushr(8))
        d = d xor (d.ushr(4))
        d = d xor (d.ushr(2))
        d = d xor (d.ushr(1))
        return (d and 1)
    }
    // Extension function to check if the LSB bit is either 1 (true) or 0 (false)
    fun Int.checkLSB(): Boolean {
        return this.and(LSB_MASK) == 1
    }
}
```

Autores: Francisco Engenheiro (a49428@alunos.isel.pt)
João Perleques (a49498@alunos.isel.pt)
Mariana Muñoz(a47076@alunos.isel.pt)

G. Código Kotlin - *TicketDispenser*

```
package ticketMachine

// Controls the state of the mechanism for ticket printing
object TicketDispenser {
    // Initializes this class
    fun init() {
        // Since the hardware of the TicketDispenser doesn't need prior activation
to enable its use,
        // this function is redundant, but it was kept in order to be coherent with
the current
        // steps used in this application for OOP
    }
    // Sends a command to print and dispense a ticket
    fun print(destinyId: Int, originId: Int, roundTrip: Boolean) {
        // Starts a mutable variable that will hold the entire frame while it's
being built
        // Step1: Shift Origin station code 4 bits to the left, in order to allocate
space for the next instruction
        var data = originId.shl(4)
        // Step2: Perform a OR logic operation between the current frame and the
Destiny station code,
        // to be able to add it to the frame
        data = data.or(destinyId)
        // Step3: Shift current frame 1 bit to the left, in order to allocate space
for the next instruction
        data = data.shl(1)
        // Step4: Perform an OR logic operation between the current frame and the
roundTrip bit,
        // to be able to add it to the frame
        // The boolean roundTrip identifies if a ticket is: a two-way ticket (true)
or a one-way ticket (false)
        if (!roundTrip) {
            data = data.or(1)
        }
        // Send data to the Serial Emitter
        SerialEmitter.send(addr = SerialEmitter.Destination.TICKET_DISPENSER, data =
data)
    }
}
```