Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

O módulo Integrated Output System (IOS) implementado é constituído por dois blocos principais: i) o que valida (Serial Receiver); e ii) o bloco que entrega a informação validada ao dispenser e ao LCD (designado por Dispatcher), conforme ilustrado na Figura 1. Neste caso o módulo de controlo, implementado em software, é a entidade reguladora e o Dispenser e o LCD são as entidades consumidoras.

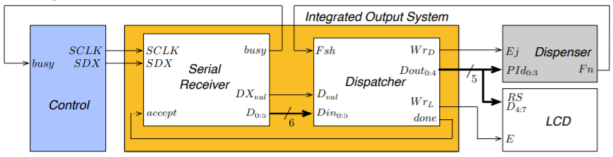
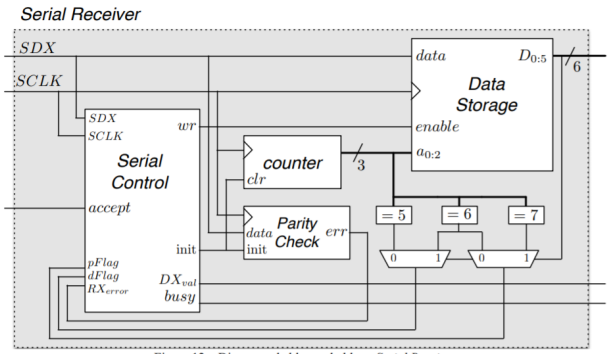


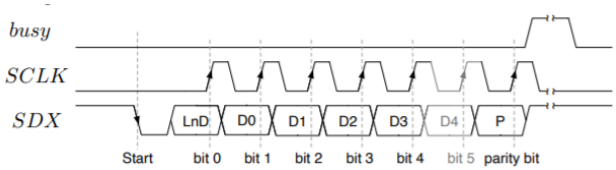
Figura 1 – Diagrama de blocos do módulo *Integrated Output System*

1 Serial Receiver

O bloco Serial Receiver é recetor de dados em série que contém um verificador da trama recebida do bloco de software control, por hardware, sendo constituído por quatro sub-blocos: i) um bloco de controlo (Serial Control); ii) o bloco de memória (designado Data Storage); iii) um contador de bits e iv) um bloco de validação da paridade (Parity check). O controlo de fluxo de saída do bloco Serial Receiver (para o módulo Dispatcher), define que o sinal DXval é ativado quando é feita a validação de uma trama recebida sendo também disponibilizado o código dessa trama no barramento D0:5. Apenas é iniciado um novo ciclo de verificação de uma nova trama quando o sinal busy se encontrar desativado e quando for promovida uma condição de início de trama O diagrama temporal do controlo de fluxo está representado na Figura 2b. O Serial Control encontrase dividido em dois ASM-Chart no qual o primeiro, apresentado na fig. 3, verifica se houve uma transição de sdx de 1 para 0 com o sclk a 0 provocando assim o a ativação do sinal start. O segundo ASM-chart tem como função a preparação dos blocos data storage, parity check e counter para a sua utilização, ativando também os sinais de busy e dxval.



a) Diagrama de blocos do *Serial Receiver*



b) Protocolo de comunicação do *Control* com o *IOS*

Figura 2 – Bloco *Serial Receiver.*

Primeiro *ASM-chart do Serial Control* de acordo com o apresentado Figura 3.

Iniciando o pelo primeiro estado (000), este não apresenta nenhum sinal ativo na sua saída mantendo-se neste estado até o sinal de *SDX* se encontrar ativo e o sinal de *SCLK* se encontrar desativo passando assim para o segundo estado. No segundo estado (001), inicia sem nada na sua saída e espera até o sinal de *SDX* se encontrar a 0 de modo a verificar o sinal de *SCLK* em que, caso este se encontre a 0 irá permitir a entrega do sinal de *start* no próximo estado e caso se encontre ativo (1) este irá voltar ao primeiro estado reiniciado o processo de ativação do *start*. Por fim no último estado este irá entregar na sua saída o valor de *start* ativo recomeçando o processo de *start*.

Segundo ASM-chart do Serial Control de acordo com a Figura 4.

Começando pelo primeiro estado este encontra-se com todos os sinais a 0 estando à espera de receber o sinal start o qual quando recebido provoca a passagem para o estado seguinte, enquanto este não estiver a 1 irá sempre voltar ao estado inicial (000).

Passando para o segundo estado (001) este é um estado de transição que apresenta na sua saída o sinal de *init* ativo e passando para o terceiro estado.

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

No terceiro estado (010) este entrega o sinal de *wr* ativo que caso o *start* se encontre ativo irá voltar ao segundo estado iniciando o *init* mais uma vez, caso este se encontre a falso irá fazer a verificação se a *dFlag* se encontra ativa, caso esta se mostre ativa significa que os bits de dados já foram todos recebidos passando então para o estado seguinte, caso este esteja a 0 significa que ainda falta receber dados voltando assim ao inicio deste estado de modo a deixar o *wr* ativo permitindo a receção de mais informação recomeçando por isso a este processo até *start* ou d*Flag* se ativarem.

No quarto estado este verifica mais uma vez o estado de *start* e como no estado anterior caso esteja ativo irá voltar ao estado dois para reiniciar o processo, no caso de não estar ativo irá verificar a *pFlag* de modo a saber se recebeu o bit de paridade que caso *pFlag* esteja a 1 irá avançar para o estado seguinte e caso esteja a 0 irá voltar ao início do estado reiniciando o processo até existir uma alteração no sinal de *start* ou no *pFlag*.

No quinto estado este inicia a 0 verificando o valor do sinal *RXerror*, caso este esteja a 1 significa que o bit de paridade está incorreto voltando por isso para o estado 000 reiniciando a preparação dos sub-blocos e a entrega do DXval e do busy, caso esteja a 0 irá passar para o útltimo estado que irá entregar o DXval e o busy na saída e volta para o primeiro estado de modo a recomeçar o processo

A descrição hardware do bloco *Serial Control* em CUPL encontra-se no Anexo AA.

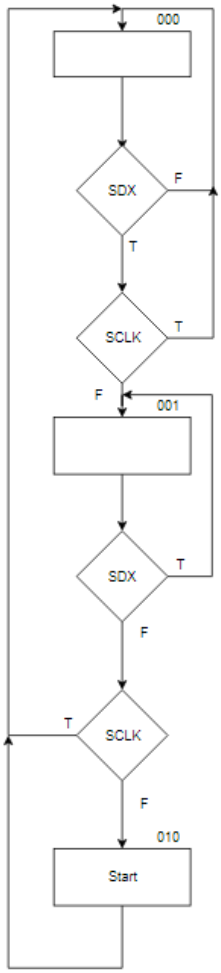


Figura 3 – Primeira máquina de estados do bloco *Serial Control*

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

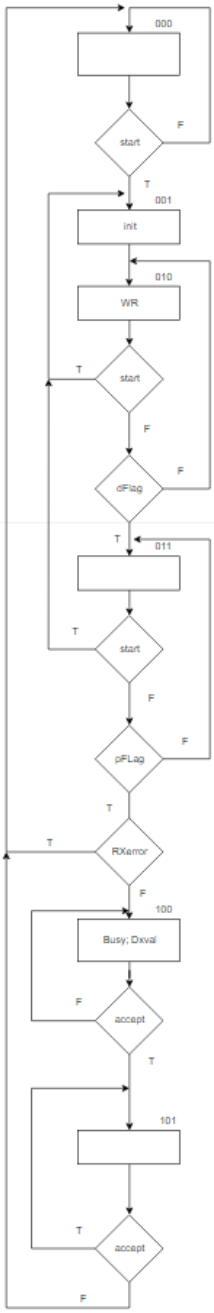


Figura 4 – Segunda máquina de estados do bloco *Serial Control.*

Existe uma segunda PAL na qual está contida o Data Storage, o counter e o parity Check de modo a completar o circuito do Serial Receiver.
Com base nas descrições do bloco *Serial Receiver* implementou-se parcialmente o módulo *IOS* de acordo com o esquema elétrico representado no Anexo D.

2 Dispatcher

O módulo *Dispatcher* implementa uma estrutura de entrega de dados, o qual recebe 6 bits de entrada e dependendo da saída assinalada envia os bits correspondentes. A receção de dados no *Dispatcher* inicia-se com a ativação do sinal *Dval* pelo sistema produtor, neste caso pelo *Serial Receiver*, enviando assim os dados para o seu destino e a ativação do sinal. Logo que receba o sinal de ativação (*Dval*), o *Dispatcher* envia os dados $Dout_{0:4}$ para o destino dependendo das flags que estejam ativas (*Wrd* para o *dispenser* e o *Wrl para o LCD).*

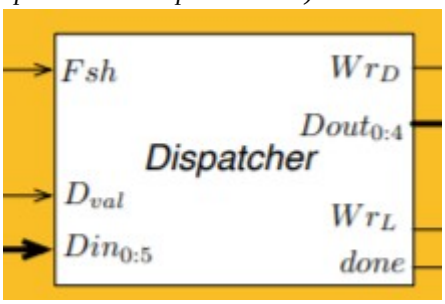


Figura 5 – Diagrama de blocos do *Key Buffer*

O bloco *dispatcher* é também responsável pelo envio do sinal que permite o *Serial Receiver* receber um novo trama neste caso o sinal *done*. O sinal *done* irá ser o sinal que ativará a entrada *accept* deixando esta a 1 de modo a reiniciar o processo de aceitação e verificação da trama.

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

A implementação do *Dispatcher* deverá ser baseada no diagrama de blocos da Figura 5 e no *ASM-chart* da figura 6.

Começando pelo primeiro estado (000) este não apresenta nenhum sinal ativo estando à espera de receber a informação do *Dval*, enquanto não a receber irá se manter neste estado ficando à espera de que este seja ativo e quando tal acontecer irá verificar o sinal de Din0 caso este seja 1 irá passar para o estado 001(*enable LCD*) e cas0 Din0 seja 0 irá passar para o estado 010(*enable Dispatcher*).

No caso de passar para o estado 001este irá começar com o sinal *WrL* ativo dando assim *enable* ao *LCD* de modo a este receber a trama de informação e passando para o estado 100.

No caso de passar para o estado 010 este irá iniciar com o sinal de *WrD* ativo e irá esperar a ativação do sinal de *Fsh* de modo a passar para o estado seguinte (001).

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

No estado 011 este irá esperar que o sinal de *Fsh* fique desativado (0) de modo a passar para o estado seguinte (100), enquanto o sinal Fsh se mantiver ativo este irá se manter neste estado.

Chegando ao estado final (100) este ativo o sinal de *done* de modo a informar que a entrega da trama foi executada com sucesso e mantendo-se neste estado até Dval se desativar o qual fará com que se regresse ao estado inicial de 000 reiniciando o circuito.

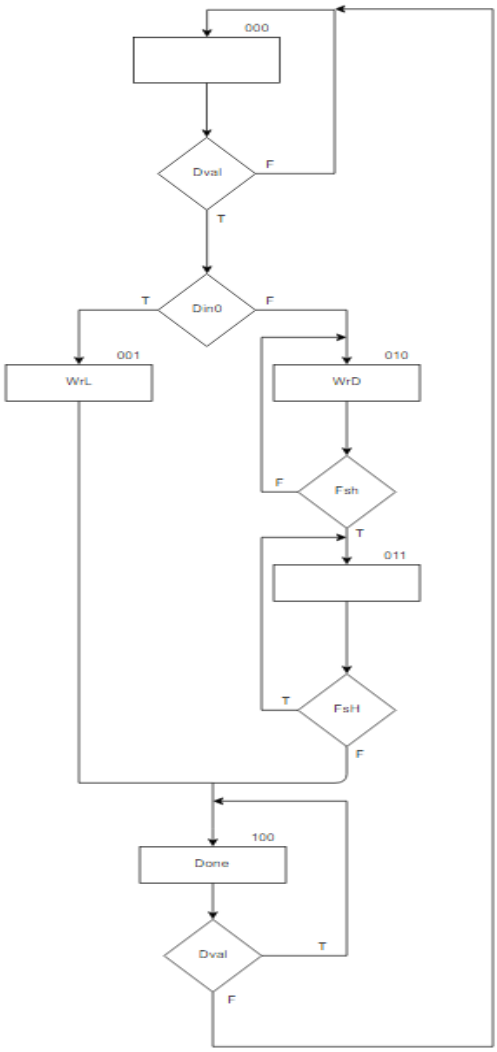


Figura 6 - Máquina de estados do bloco *Dispatcher*

Com base nas descrições do bloco *Serial Receiver* e do bloco *Dispatcher* implementou-se o módulo *Integrated Output System (IOS)* de acordo com o esquema elétrico representado no Anexo D.

3 Interface com o *Control*

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

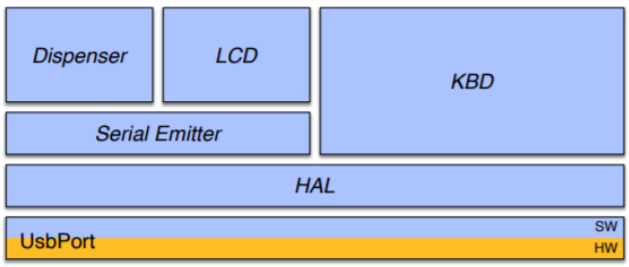


Figura 7 – Diagrama lógico do módulo *Control* de interface com o módulo *IOS*

Os módulos de software *Serial Emitter*, Dispenser e *LCD* desenvolvidos são descritos nas secções Error: Reference source not found. e Error: Reference source not found, e o código fonte desenvolvido nos Anexos G e L, respetivamente.

3.1 *Serial Emitter*

Esta classe tem a função de enviar uma trama para o *Serial Receiver* e de verificar o estado do sinal de *busy* de modo a permitir o começo do envio de uma nova trama.

Esta classe é composta por 3 funções principais.

A primeira função é a função *init* que permite iniciar a classe.

A segunda função é a função *send* e tem como objetivo o envio de uma trama para o *SerialReceiver* identificado o destino em *addr(adress)* e os bits de dados em *'data'*.

Por último tem a função *isBusy* a qual retorna true caso o canal esteja ocupado não permitindo o início de um novo ciclo de validação e entrega de trama.

Sendo que esta classe irá controlar o funcionamento do *IOS* para tal foram criadas 3 funções principais. Estas funções utilizam as funções criadas na classe HAL. De modo a iniciar esta implementação críamos constantes que definem os valores de entrada e saída dos sinais necessários.

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

```
private const val BUSY = 0x40
private const val SDX = 0x01
private const val SCLK = 0x02
private var BITS = 4
```

Figura 9- definição das constante dos bits de entrada e saída em hexadecimal.

3.1.1 Função init

Esta primeira função tem como objetivo iniciar o código.
De acordo com o que observámos anteriormente de modo a iniciar o processo do *IOS* é necessário ter o SDX ativo e o SCLK a desativo sendo por isso aplicada a função *setBits* no SDX e a função *clrBits* no SCLK.

```
// Inicia a classe
fun init() {
    HAL.setBits(SDX)
    HAL.clrBits(SCLK)
}
```

Figura 10 – Função init

3.1.2 Função send

Esta função é responsável por enviar a trama para o Serial Receiver com a Data e com a informação do destino para o qual é suposto a informação ir (seja este o LCD ou o Dispenser). Esta função irá começar por escolher para qual dos destinos irá enviar a trama, seguidamente procederá ao envio de cada bit do de maior peso para o de menor realizando de seguida um parity check. No caso de ser para o LCD vai ser enviado ainda um bit adicional (RS), que se econtra na posicao de maior peso de data, mas e mandando em primeiro lugar.

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

    while(isBusy());
    var p = 0
    HAL.clrBits(SDX)
    if(Destination.LCD == addr) {
        HAL.setBits(SDX)
        p++
        BITS = 5
    }
    if(Destination.DISPENSER == addr) {
        BITS = 4
    }
    for(i in 0 until BITS){
        val bit = (data shr i) and 0x1
        p += bit
        clockSCLK(bit)
    }
    val parityCheck = p.inv() and 0x1
    clockSCLK(parityCheck)
    clockSCLK(1)
}

//muda o valor do bit na posicao SDX
private fun changeSDXBit (int: Int){
    return when(int){
        1->{
            HAL.setBits(SDX)
        }
        else->{
            HAL.clrBits(SDX)
        }
    }
}
```

Figura 11- função *send*

3.1.3 Função isBusy

Esta função tem como objetivo retornar o valor de busy de modo a ser possível verificar se este está ativo ou não.

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

Figura 12- função Busy

3.1.4 Funções clockSCLK e changeSDXBit

A função *clockSCLK* tem como função indo variar o valor de SCLK de modo a este se comportar como um *clock*.
No caso da função *changeSDXBit* esta muda o sinal do *SDX* de acordo com o valor que este necessita de apresentar de modo a representar a trama correta.

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

```
private fun clockSCLK(bit:Int) {
    HAL.setBits(SCLK)
    changeSDXBit(bit)
    HAL.clrBits(SCLK)
}

 private fun changeSDXBit (int: Int){
     return when(int){
         1->{
             HAL.setBits(SDX)
         }
         else->{
             HAL.clrBits(SDX)
         }
     }
```

Figura 13- funções *clockSCLK* e *chageSDXBit*

3.2 Dispenser

A classe *dispenser* é composta por três funções.

A primeira função é a *fun init()* que tem como função iniciar a classe de modo a esta estar pronta para ser usada.

A segunda função é a *fun dispense* a qual chama *o Serial Emitter* e envia para o *dispenser* o local onde o produto está de modo a este ser entregue.

```
fun init(){}
fun dispense(productId: Int){
    SerialEmitter.send(SerialEmitter.Destination.DISPENSER,productId)
}
```

Figura 14 -funções do dispenser.

4 Conclusões

Com a conclusão deste módulo do IOS, verificamos que este é de maior complexidade que os outros pois é necessário manter sempre a trama ordenada de modo a entregar a informação correta para os restantes componentes. Observando os ASM-*charts* verificamos que estes necessitam no mínimo de 3 ou 6 *clocks* para funcionar mas serão sempre necessários muitos mais *clocks* pois são necessários *clocks* para verificar se certos sinais estão ativos ou não.

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

A. Descrição CUPL do bloco *Serial Receiver(PRIMEIRA PAL)*

```
/* Start Here */

/* Input */

PIN 1 = Osc;
PIN 2 = SCLK;
PIN 3 = SDX;
PIN 5 = Fsh;
PIN 6 = Qd0;
PIN 9 = pFlag;
PIN 10 = dFlag;
PIN 11 = RXerror;

/* Output */

PIN 14 = WrL;
PIN 15 = WrD;
PIN 20 = wr;
PIN 21 = init;
PIN 23 = busy;
PIN [16..18] = [Qss0..2];

PINNODE [25..26] = [Qs0..1];
PINNODE [28..30] = [Qds0..2];

/* Serial Control */
/* Frist Module */

[Qs0..1].AR = 'b'0;
[Qs0..1].SP = 'b'0;
[Qs0..1].CKMUX = Osc;

sequence [Qs0..1]{
    present 0
        if SDX & !SCLK next 1;
        default next 0;
    present 1
        if SCLK next 0;
        if !SDX & !SCLK next 2;
        default next 1;
    present 2
        out start;
        default next 0;
}

/* Second Module */

[Qss0..2].AR = 'b'0;
[Qss0..2].SP = 'b'0;
[Qss0..2].CK = !Osc;

sequence [Qss0..2]{
    present 0
        if start next 1;
        if !start next 0;
    present 1
        out init;
        default next 2;
    present 2
        out wr;
        if start next 1;
        if !start & !dFlag next 2;
        if !start & dFlag next 3;
    present 3
        if start next 1;
        if !start & !pFlag next 3;
        if !start & pFlag next 4;
    present 4
        if RXerror next 0;
        if !RXerror next 5;
    present 5
        out busy, DXval;
        if accept next 0;
        if !accept next 5;
}

/* Dispatcher */

[Qds0..2].AR = 'b'0;
[Qds0..2].SP = 'b'0;
[Qds0..2].CKMUX = Osc;

sequence [Qds0..2]{
    present 0
        if !DXval next 0;
        if DXval & Qd0 next 1; /* LCD */
        if DXval & !Qd0 next 2; /* Dispense */
    present 1 /* LCD */
        out WrL;
        default next 4;
    present 2 /* Dispense */
        out WrD;
        if Fsh next 3;
        default next 2;
    present 3
        if Fsh next 3;
        default next 4;
    present 4
        out accept;
        if DXval next 4;
        default next 0;
}
```

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

B. Descrição CUPL do bloco *Dispatcher/SerialReceiverBlocks(SEGUNDA PAL)*

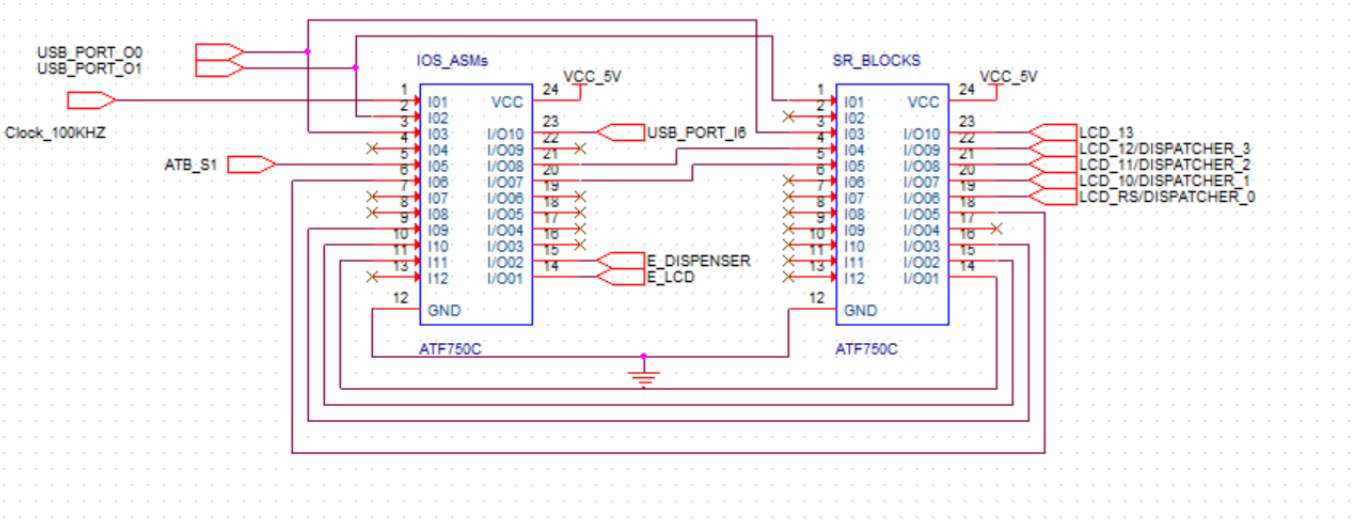
```
/* Start Here */

/* Input */

PIN 1 = SCLK;
PIN 3 = SDX;
PIN 4 = init;
PIN 5 = wr;

/* Output */

PIN 14 = RXerror;
PIN 15 = dFlag;
PIN 16 = pFlag;
PIN [18..23] = [Qd0..5];

PINNODE [30..32] = [Qc0..2];
PINNODE 33 = Qp0;


/* Counter */

[Qc0..2].AR = init;
[Qc0..2].SP = 'b'0;
[Qc0..2].CKMUX = SCLK;

C0 = ('b'1 $ Qc0);
C1 = (Qc0 $ Qc1);
C2 = ((Qc0 & Qc1) $ Qc2);

[Qc0..2].D = [C0..2];

/* Decoder */

Enable = wr;

DEC0 = (!Qc0 & !Qc1 & !Qc2) & Enable;
DEC1 = (Qc0 & !Qc1 & !Qc2) & Enable;
DEC2 = (!Qc0 & Qc1 & !Qc2) & Enable;
DEC3 = (Qc0 & Qc1 & !Qc2) & Enable;
DEC4 = (!Qc0 & !Qc1 & Qc2) & Enable;
DEC5 = (Qc0 & !Qc1 & Qc2) & Enable;

/* Flip-Flops */

[Qd0..5].AR = 'b'0;
[Qd0..5].SP = 'b'0;
[Qd0..5].CKMUX = SCLK;

[Qd0..5].D = [DEC0..5] & SDX # ![DEC0..5] & [Qd0..5];

/* Parity Check */

Qp0.AR = init;
Qp0.SP = 'b'0;
Qp0.CKMUX = SCLK;

Qp0.D = Qp0 $ SDX;
RXerror = !Qp0;

/* Flags */

dFlag = (Qc0 & !Qc1 & Qc2) & !Qd0 # (!Qc0 & Qc1 & Qc2) & Qd0;

pFlag = (!Qc0 & Qc1 & Qc2) & !Qd0 # (Qc0 & Qc1 & Qc2) & Qd0;
```

C.

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

D. Esquema elétrico do módulo *Integrated Output System*

E.



F.

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

G. Código *Kotlin – Serial Emitter*

H.

```
import isel.leic.utils.Time

object SerialEmitter {  // Envia tramas para os diferentes módulos Serial Receiver.

    private const val BUSY = 0x40
    private const val SDX = 0x01
    private const val SCLK = 0x02
    private var BITS = 4

    enum class Destination { DISPENSER, LCD }

    // Inicia a classe
    fun init() {
        HAL.setBits(SDX)
        HAL.clrBits(SCLK)
    }
```

I.

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

J.

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

    while(isBusy());
    var p = 0
    HAL.clrBits(SDX)
    if(Destination.LCD == addr) {
        HAL.setBits(SDX)
        p++
        BITS = 5
    }
    if(Destination.DISPENSER == addr) {
        BITS = 4
    }
    for(i in 0 until BITS){
        val bit = (data shr i) and 0x1
        p += bit
        clockSCLK(bit)
    }
    val parityCheck = p.inv() and 0x1
    clockSCLK(parityCheck)
    clockSCLK(1)
}

//muda o valor do bit na posicao SDX
private fun changeSDXBit (int: Int){
    return when(int){
        1->{
            HAL.setBits(SDX)
        }
        else->{
            HAL.clrBits(SDX)
        }
    }
}
```

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

```
        private fun changeSDXBit (int: Int){
            return when(int){
                1->{
                    HAL.setBits(SDX)
                }
                else->{
                    HAL.clrBits(SDX)
                }
            }
        }

        // Retorna true se o canal série estiver ocupado
        private fun isBusy(): Boolean {
            return HAL.isBit(BUSY)
        }

        private fun clockSCLK(bit:Int) {
            HAL.setBits(SCLK)
            changeSDXBit(bit)
            HAL.clrBits(SCLK)
        }
    }
```

K.

Autores:
Bernardo Serra 47539
Pedro Raposo 48316
Rafael Costa 48315

L. Código *Kotlin -Dispenser*

M.

N.

```
object Dispenser {
    fun init(){}
    fun dispense(productId: Int){
        SerialEmitter.send(SerialEmitter.Destination.DISPENSER,productId)
    }
}
```