ISEL
ADEETC
Área Departamental de Engenharia Electrónica e de Telecomunicações e de Computadores

Licenciatura em Engenharia Informática e de Computadores

Jogo Invasores Espaciais
(*Space Invaders Game*)

Projeto de
Laboratório de Informática e Computadores
2016 / 2017 verão

Índice

1 Visão geral

1.1 Sumário dos conteúdos do relatório

Este relatório foi elaborado no âmbito da unidade curricular de Laboratório de Informática e Computadores e acompanha a implementação do jogo Invasores Espaciais (*Space Invaders Game*) utilizando um PC e periféricos para interação com o jogador.

Durante a realização do trabalho, identificaram-se objetivos (1) metodológicos: estruturação; modularidade e generalidade; e (2) práticos: utilização da linguagem descrição de hardware VHDL, implementação de interface full interlock na comunicação de sinais entre dispositivos e entre módulos do mesmo componente. No **Ponto 1.2** faz-se a descrição das regras e a composição do sistema de jogo. Partindo dos objetivos identificados, o **Capítulo 2** apresenta uma arquitetura para o sistema que implementa o jogo e uma análise dos módulos que a compõe. Acompanha-se esta análise com referências aos métodos e recursos utilizados. No **Capítulo 3** discutem-se os resultados obtidos, explicitando-se conclusões relevadas pelo trabalho desenvolvido, apontando-se propostas de desenvolvimento futuro. Em anexo encontra-se listado em dois apêndices (Apêndice A e Apêndice B) o código VHDL e JAVA que foi usado com comentários que facilitam a sua compreensão. Na programação JAVA, foi seguida a convenção de escrita do nome de métodos em camelCase, e do nome de constantes em MAIÚSCULAS.

A estrutura deste documento, assim como a descrição dos módulos que compõem a arquitetura do sistema implementado, seguem de perto o enunciado proposto. As ocasiões em que as recomendações não foram exatamente observadas são oportunamente indicadas e justificadas.

Em cada ponto do documento inclui-se, em texto cinzento, os elementos fornecidos pelo enunciado (arquitetura proposta) que esclarecem sobre a composição modular do sistema e a comunicação entre os módulos. O texto a negro corresponde ao trabalho de projeto concretizado.

1.2 Descrição

Para representar os invasores espaciais, são usados números entre 0 e 9. O jogador controla os disparos da nave espacial, mirando sobre o primeiro invasor da fila. Este é eliminado se, no momento do disparo, o número da mira e do invasor coincidirem. O jogo termina quando a nave for atingida pelos invasores espaciais. Para se iniciar um jogo, é necessário um crédito, obtido pela introdução de moedas. Só são aceites pelo sistema moedas de um euro, que correspondem a dois créditos.

O sistema de jogo é constituído por: um teclado de 12 teclas; um moedeiro (*Coin Acceptor*); um mostrador Liquid Cristal Display (LCD) de duas linhas com 16 caracteres; um gerador de sons (Sound Generator) e uma chave de manutenção designada por M, para colocação do sistema em modo de Manutenção. O diagrama de blocos do jogo Invasores Espaciais é apresentado na Figura 1:

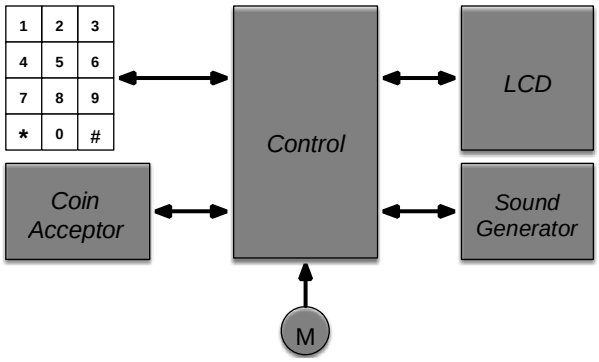


Figura 1 – Diagrama de blocos do jogo Invasores Espaciais (*Space Invaders Game*)

O sistema de jogo inclui as seguintes funcionalidades:

Em modo de jogo: depois de introduzidos créditos, e de ser pressionada a tecla de início de jogo, iniciar um novo jogo; em modo de espera de início de novo jogo, uma a uma, mostrar as melhores pontuações por ordem decrescente.

Em modo de manutenção: consultar os contadores de jogos e créditos, iniciá-los ou realizar um jogo de teste, sem créditos e sem que a pontuação do jogo seja guardada na lista de pontuações; desligar o sistema, mediante confirmação, armazenando persistentemente em dois ficheiros de texto, a lista de pontuações e os contadores de jogos e moedas.

2 Descrição da Arquitectura

Arquitetura proposta no enunciado

O sistema foi implementado numa solução híbrida de *hardware* e *software*, como apresentado no diagrama de blocos da Figura 2. A arquitetura proposta é constituída por cinco módulos principais: i) um leitor de teclado, designado por *Keyboard Reader*; ii) um módulo de interface com o LCD, designado por *Serial LCD Controller* (SLCDC); iii) um módulo de interface com o gerador de sons (*Sound Generator*), designado por Serial *Sound Controller* (SSC); iv) um moedeiro, designado por *Coin Acceptor*; e v) um módulo de controlo, designado por *Control*. Os módulos i), ii) e iii) foram implementados em hardware, o moedeiro foi simulado, enquanto o módulo de controlo foi implementado em software a executar num PC usando linguagem JAVA.

O módulo *Keyboard Reader* é responsável pela descodificação do teclado matricial de 12 teclas, determinando qual a tecla pressionada e disponibilizando o seu código, com quatro bits, ao módulo Control. Caso este não esteja disponível para o receber imediatamente, o código da tecla é armazenado até ao limite de dois códigos.

O módulo Control processa os dados e envia a informação a apresentar no LCD através do módulo SLCDC. O gerador de sons é atuado pelo módulo Control, através do módulo SSC. Por razões de ordem física, e por forma a minimizar o número de fios de interligação, a comunicação entre o módulo Control e os módulos SLCDC e SSC é realizada através de um protocolo série.

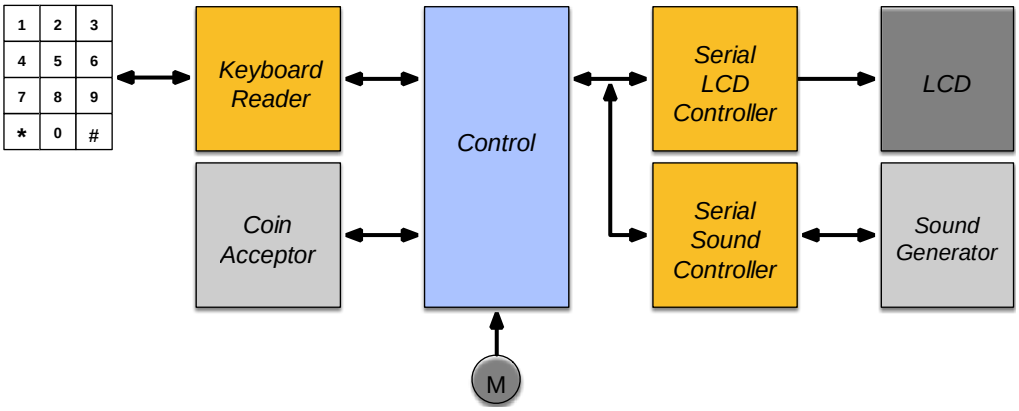


Figura 2 – Arquitetura do sistema que implementa o jogo Invasores Espaciais (*Space Invaders Game*)

Montagem realizada em Laboratório

Para a montagem do sistema foi utilizada a placa de desenvolvimento µLICχ, que inclui um CPLD (*Complex Programmable Logic Device*) Xilinx XC95144XL com 144 macro-células. A placa de desenvolvimento fornece ligações para interface com um PC, através de um conetor *USB Type-B* (J5 na figura abaixo), que também fornece alimentação à placa quando ligada a um *slot* USB de um PC na extremidade oposta [1].

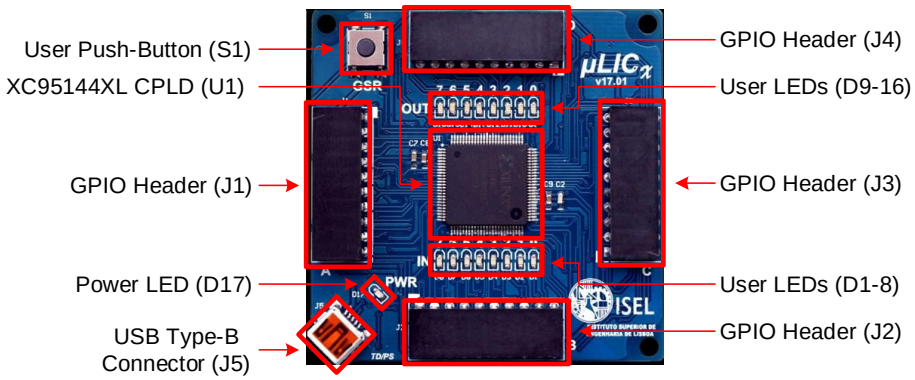


Figura 3 – Visão geral das características da placa de desenvolvimento µLICχ conforme datasheet

A placa tem 4 *headers GPIO* (*J1-4*), cada um com 18 pins de I/O.

Na comunicação com o módulo de controlo, os LED D1-D8 são sinais de saída e as entradas são mostradas nos LED 9-16. O módulo de controlo utiliza a biblioteca UbsPort para leitura e escrita na comunicação com a placa de desenvolvimento µLICχ.

Dispõe de oscilador a 1MHz, configurável, por meio de divisores de frequência para 1000, 500,250 ou 125 MHz. Para esta implementação foi usado o oscilador a 1MHz, como *MasterClock*. Entendeu-se que, a este nível de complexidade, era possível desenhar internamente os módulos de forma a assegurar o sincronismo. Também se constatou em laboratório, que a integridade de sinais a dado momento não é comprometida por uma discrepância de volume de sinais a receber, tratar e comunicar por um módulo face a outros.

Foi ligado a J2 o teclado *µLIC revision 17.1*.

Os interruptores da placa de teste ATB foram usados para simular o moedeiro e a chave da manutenção; e os seus portos de saída com LED foram usados para simular o componente *Sound Generator*. Ambos foram ligados a pinos do *Header* J1.

Em *breadboard* foi montado um LCD da família MC1602C, de 2 linhas por 16 colunas, 16 pinos, com controlador HD44780-U. Este teclado oferece a possibilidade de gerar e armazenar caracteres e a seleção da posição de escrita no LCD é feita por definição de endereço da DDRAM.

A CPLD foi programada usando a linguagem de descrição de *hardware VHSIC* (*Very High Speed Integrated Circuits*), também designada neste documento por VHDL. Para informação sobre pinout da montagem, consultar ficheiro de constrições em apêndice. A figura 4 mostra o sistema em funcionamento:

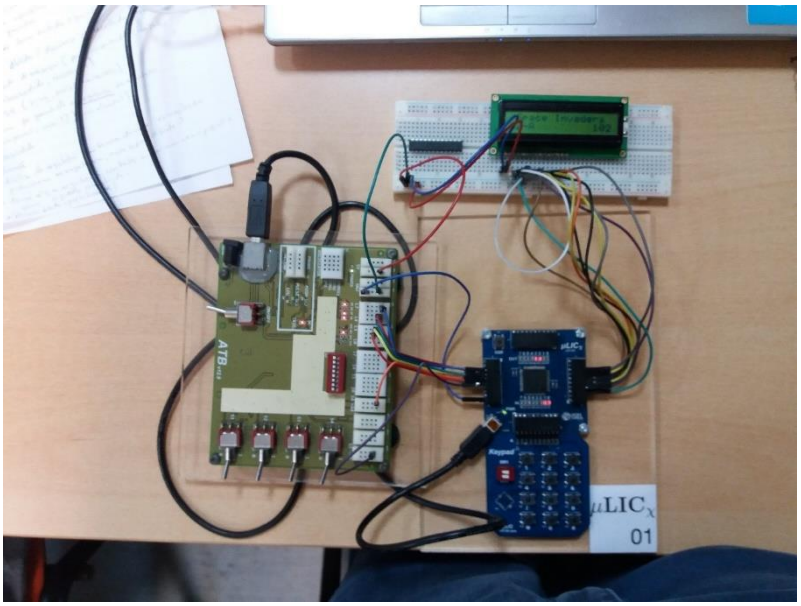


Figura 4 – aspeto do sistema em funcionamento

2.1 Keyboard Reader

O módulo *Keyboard Reader* é constituído por dois blocos principais: i) o descodificador de teclado (*Key Decode*); e ii) o bloco de armazenamento e de entrega ao consumidor (designado por *Key Buffer*), conforme ilustrado na Figura 5. Neste caso, o módulo de controlo, implementado em software, é a entidade consumidora.

A necessidade de implementar o modulo Keyboard Reader, prendesse com a dimensão de barramento disponível, isto é, pela dimensão do teclado são necessários comunicar 12 bits de informação (um para cada tecla), seria necessário que o modulo Control, pudesse receber 12 bit em paralelo, desta forma o modulo Control apenas necessita de processar 5 bits de entrada e 1 de sinal (Ack)

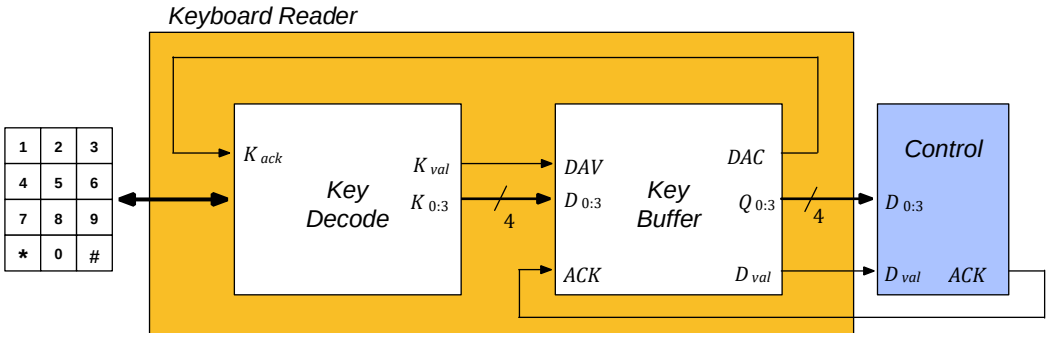
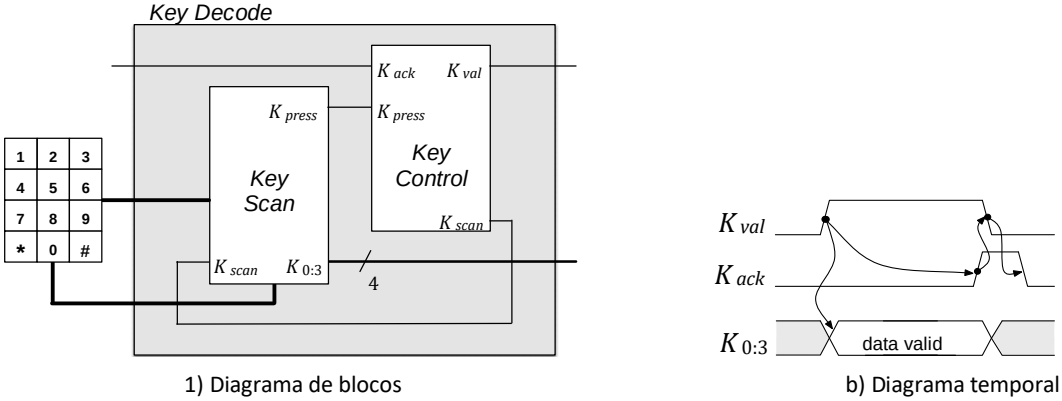


Figura 5 - Diagrama de blocos do módulo *Keyboard Reader*

2.1.1 Key Decode

O bloco *Key Decode* implementa um descodificador de um teclado matricial 4x3 por hardware, sendo constituído por três sub-blocos: i) um teclado matricial de 4x3; ii) o bloco *Key Scan*, responsável pelo varrimento do teclado; e iii) o bloco *Key Control*, que realiza o controlo do varrimento e o controlo de fluxo, conforme o diagrama de blocos representado na Figura 6a. O controlo de fluxo de saída do bloco *Key Decode* (para o módulo *Key Buffer*), define que o sinal Kval é ativado quando é detetada a pressão de uma tecla, sendo também disponibilizando o código dessa tecla no barramento K0:3. Apenas é iniciado um novo ciclo de varrimento ao teclado quando o sinal Kack for ativado e a tecla premida for libertada. O diagrama temporal do controlo de fluxo está representado na *Figura 6b*



1) Diagrama de blocos

b) Diagrama temporal

Figura 6 – Bloco Key Decode

2.1.1.1 Key Scan

Foram estudados dois métodos de varrimento do teclado para determinação de pressão sobre uma tecla a *que correspondem dois diagramas de blocos, que se apresentam na Figura 7.*

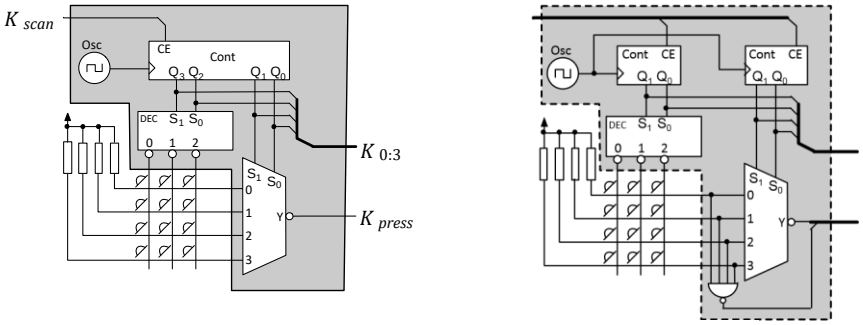


Figura 7 – Diagrama de blocos do bloco Key Scan

Versão I *Versão II*

A malha do teclado é uma rede *pull-up*. A pressão sobre uma tecla causa um dos sinais de entrada do *MUX* a tomar o valor lógico 0. O contador de 4 bit tem domínio [0,15]
A figura 8 faz corresponder o valor de saída do contador com o valor de entrada no *MUX*, abstraindo a malha numa tabela. Tome-se o caso inicial em que o valor de saída do contador é 12. Outros casos semelhantes poderiam ter sido usados.
Se no momento inicial for pressionado o botão na 3ª coluna e 4ª linha do teclado, a latência do sinal *Kpress* (*active low*) será de 15 impulsos de *clock*, sem contar com o tempo de comutação do hardware do circuito combinatório,

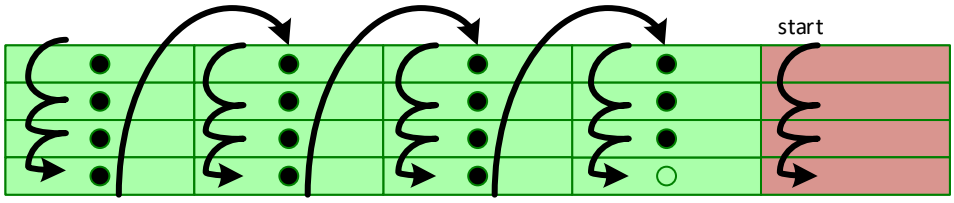


Figura 8 – caso de estudo do varrimento efetuado pela versão I do bloco KeyScan

Tal acontece, pois a versão I compromete-se com um varrimento linha a linha para cada coluna da malha. Para além disso, para cada contagem de 0 a 15, há quatro impulsos de relógio que não correspondem ao varrimento de qualquer coluna da malha.

Na versão II, a pesquisa por linhas e colunas processa-se independentemente, por meio de dois contadores, controlados por CE diferentes. Sempre que é detetada pressão sobre uma tecla numa coluna/linha, o contador correspondente para. Observa-se, no máximo, um impulso de relógio correspondente a uma coluna inexistente na malha. No pior caso, a latência do sinal *Kpress* será de 5 impulsos do oscilador, sem contar com o tempo de comutação do hardware do circuito combinatório. Por apresentar menor latência, isto é, maior eficiência na pesquisa, a versão II foi preferida à versão I.

2.1.1.2 Key Control

O bloco Key Control é implementado à custa de uma máquina de estados, com o ASM apresentado na figura 9.

O sinal Kack é ativo quando um valor de 4 bits correspondente a uma tecla foi armazenada. Kpress é um sinal de 2 bits que indica que foi detetada pressão numa coluna (bit de peso 1) ou numa linha (bit de peso 0). Kval é ativo quando a pesquisa termina, tendo sido encontrada a tecla premida, e só é permitido o começo de uma nova pesquisa, assim que ocorrer transição ascendente de Kack.

2.1.2Key Buffer

O bloco *Key Buffer* a desenvolver corresponderá a uma estrutura de armazenamento de dados, com capacidade para armazenar uma palavra de quatro bits. A escrita de dados no bloco *Key Buffer*, cujo diagrama de blocos é apresentado na Figura 6, inicia-se com a ativação do sinal *DAV* (*Data Available*) pelo sistema produtor, neste caso pelo bloco *Key Decode*, indicando que tem dados para serem armazenados. Logo que tenha disponibilidade para armazenar informação, o bloco *Key Buffer* regista os dados $D_{0:3}$ em memória. Concluída a escrita em memória, ativa o sinal *DAC* (*Data Accepted*) para informar o sistema produtor que os dados foram aceites. O sistema produtor mantém o sinal *DAV* ativo até que o sinal *DAC* seja ativado. O bloco *Key Buffer* só desativa o sinal *DAC* após o sinal *DAV* ter sido desativado.
A implementação do bloco *key Buffer* deverá ser baseada numa máquina de controlo (*Key Buffer Control*) e num registo (*Output Register*).

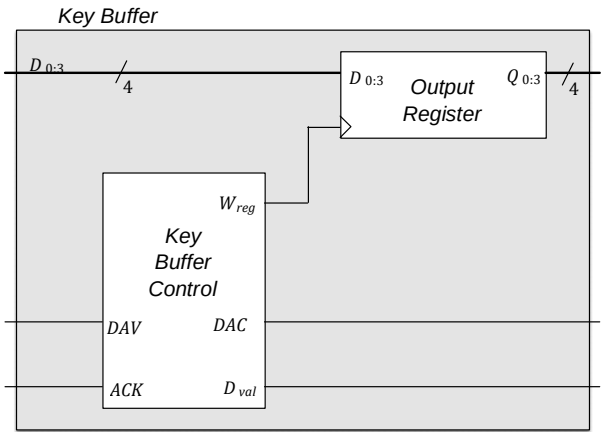


Figura 10 – Diagrama de blocos do bloco Key Buffer

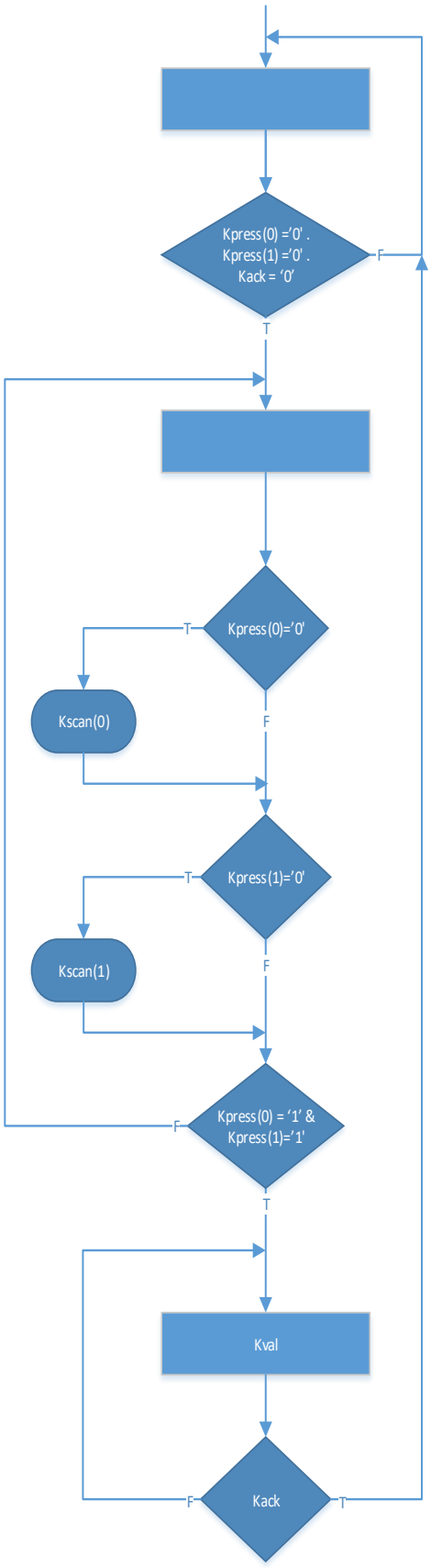


Figura 9 – Diagrama ASM do módulo Key Control

2.1.2.1 Key Buffer Control

O bloco *Key Buffer Control* foi implementado de acordo com o diagrama ASM na figura 12.

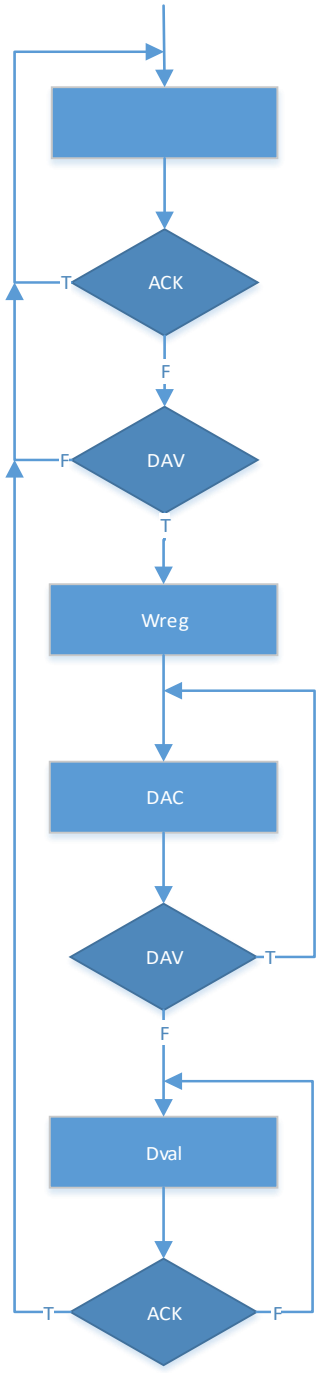
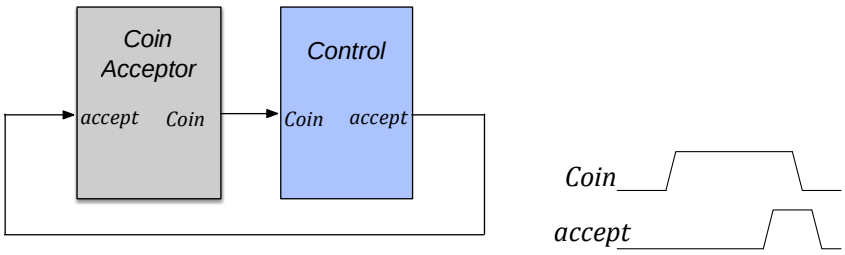


Figura 12 – Diagrama ASM do bloco *Key Buffer Control*

2.2Coin Acceptor

O módulo *Coin Acceptor* implementa a interface com o moedeiro, sinalizando ao módulo *Control* que o moedeiro recebeu uma moeda através da ativação do sinal *Coin*. A entidade consumidora informa o *Coin Acceptor* que já contabilizou a moeda ativando o sinal *accept*, conforme apresentado no diagrama temporal da figura 11.



a) Diagrama de blocos b) Diagrama temporal

Figura 11 Módulo Coin Acceptor

2.3 Serial LCD Controller

O módulo *Serial LCD Controller* (*SLCDC*) implementa a interface com o *LCD*, fazendo a receção em série da informação enviada pelo módulo de controlo e entregando-a posteriormente ao *LCD*, conforme representado na Figura 13.

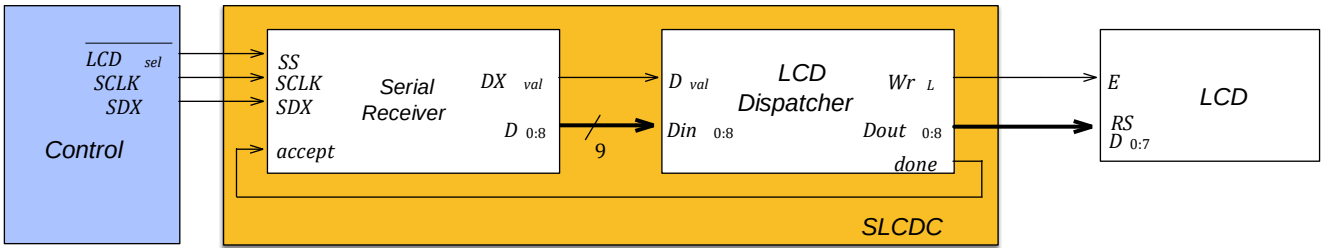


Figura 13 – Diagrama de blocos do módulo *Serial LCD Controller*

O módulo *SLCDC* recebe em série uma mensagem constituída por nove bits de informação e um bit de paridade. A comunicação com o este módulo realiza-se segundo o protocolo ilustrado na Figura 14, em que o bit *RS* é o primeiro bit de informação e indica se a mensagem é de controlo ou dados. Os seguintes 8 bits contêm os dados a entregar ao *LCD*. O último bit contém a informação de paridade par, utilizada para detetar erros de transmissão.

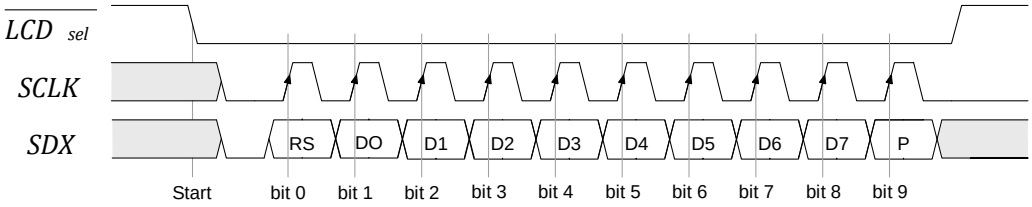


Figura 14 - Protocolo de comunicação com o módulo *Serial LCD Controller*

O emissor, *software*, quando realizado em pretende enviar uma trama para o módulo S*LCDC* promove uma condição de início de trama (*Start*), que corresponde a uma transição descendente na linha $LCD_{\$\%\&}$. Após a condição de início, o módulo S*LCDC* armazena os bits de dados da trama nas transições ascendentes do sinal *SCLK*.

2.3.1Serial Receiver

O bloco *Serial Receiver* do módulo S*LCDC* é constituído por quatro blocos principais: *i*) um bloco de controlo; *ii*) um bloco conversor série paralelo; *iii*) um contador de bits recebidos; e *iv*) um bloco de validação de paridade, designados por *Serial Control, Shift Register*, *Counter* e *Parity Check* respetivamente. O bloco *Serial Receiver* deverá ser implementado com base no diagrama de blocos apresentado na Figura 15.

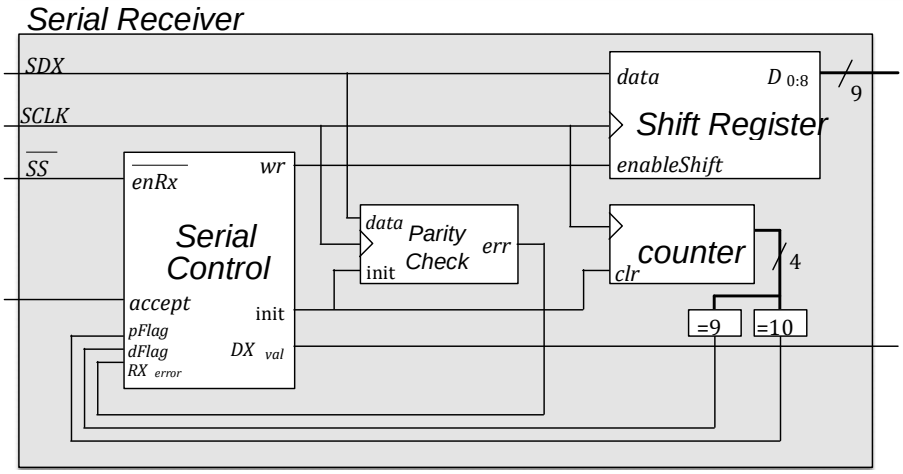


Figura 15 - Diagrama de blocos do bloco *Serial Receiver*

2.3.1.1 Serial Control

O bloco Serial Control é responsável por controlar o funcionamento do Serial Receiver.

Tem as seguintes entradas de informação:

- enRx – que recebe indicação de que a informação recebida no Serial Receiver pelo SS lhe é destinada, ativando ou não o funcionamento do bloco.
- accept – que recebe indicação de que é possível enviar nova informação, permitindo ativar DXval, isto é, depois de ativar DXval, apenas se pode ativar novamente depois de ter recebido indicação pelo accept que que a informação foi tratada (fully interlocked).
- dFlag – foram lidados 9 bit.
- pFlag – que recebe indicação do bloco counter de que a informação recebida no Serial Receiver por SDX corresponde ao ultimo bit de dados.
- RXerror – que recebe indicação do bloco Parity Check de que o Bit de paridade não corresponde á paridade dos dados recebidos.

E as seguintes saídas de informação:

- wr – envia indicação ao bloco Shift Register de que a informação para efetuar o shift dos dados que contem com a entrada adicional do nove bit proveniente de SDX.
- Init – enviar indicação para iniciar os blocos Parity Check e counter
- DXval – envia indicação para o exterior do Serial Receiver de que este tem um conjunto de dados validos, que se encontram disponíveis na saída D do bloco Shift Register

O bloco Serial Control, foi implementado com uma maquina de estados representada pelo ASM figura 16.

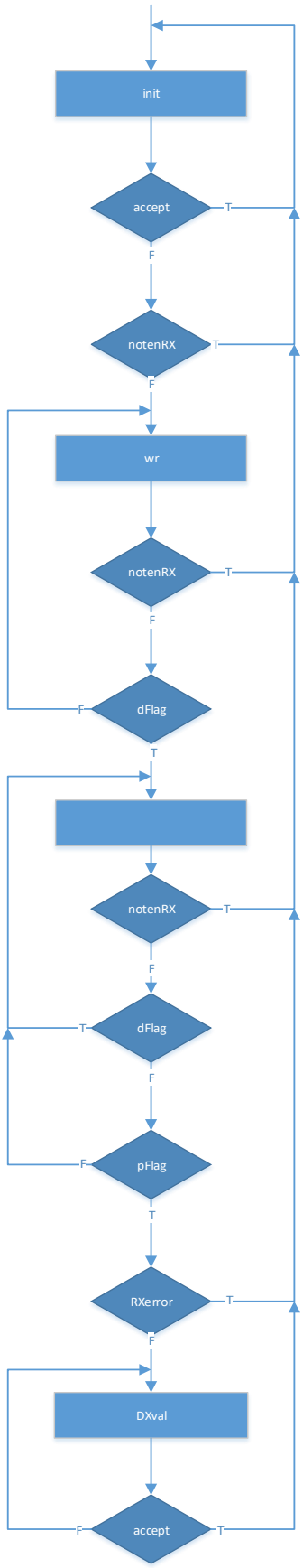


Figura 16 – ASM do bloco *Serial Control*

2.3.1.2 Shift Register

O bloco Shift Register é responsável transformar os dados recebido em serie no bloco Serial Receiver em dados disponíveis em paralelo, foi implementado com recurso a flip-flops do tipo D em sequencia, de acordo com diagrama de blocos da figura 17.
Tem as seguintes entradas de informação:

- data – entrada bit a bit dos dados para efetuar Shift, proveniente do Serial Receiver por SDX
- enableShift – recebe indicação do bloco Serial Control para efetuar o Shift dos dados.

E as seguintes saídas de informação:

- D0..8 – Saida dos dados

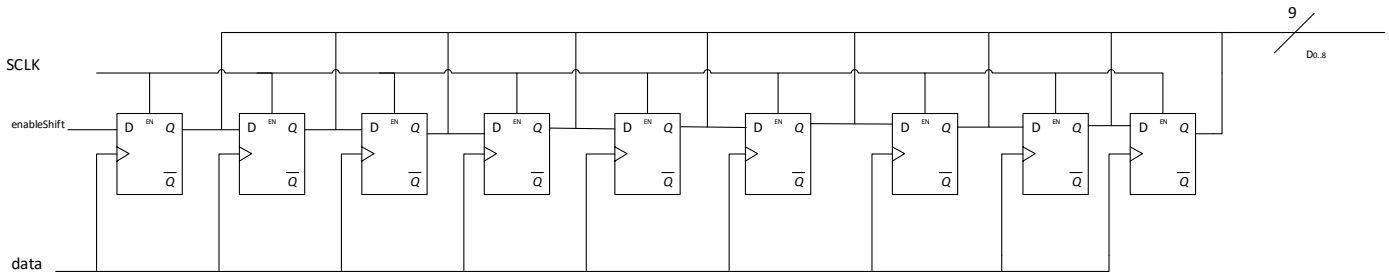


Figura 17 – Diagrama do bloco Shift Register

2.3.1.3 Parity Check

O bloco Parity Check é responsável pelo calculo da paridade dos dados recebidos, pelo método de contagem do numero de bit de dados recebidos a "1" tendo paridade quando esse numero por impar.
Na pratica é uma versão muito resumida da operação adição onde soma o valor de cada bit, mas apenas nos interessa saber o valor do primeiro bit do resultado, pois se for 0 temos um numero par de bits a "1" e se for 0 temos um numero impar de bits a "1", como não necessitamos do resto da informação da operação adição e como para a adição de dois bit se pode efetuar através da operação xor, o bloco Parity Check é implementado com um flip-flop do tipo D, que regista o resultado da operação xor entre o bit recebido e valor do registo anterior, de acordo com esquema da figura 18.
Tem as seguintes entradas de informação:

- data – entrada bit a bit dos dados para calculo da paridade
- init – recebe indicação do bloco Serial Control para fazer um init (iniciar o somador / iniciar o flip-flop)

E as seguintes saídas de informação:

- err – envia indicação ao bloco Serial Control da existência de erro no calculo da paridade

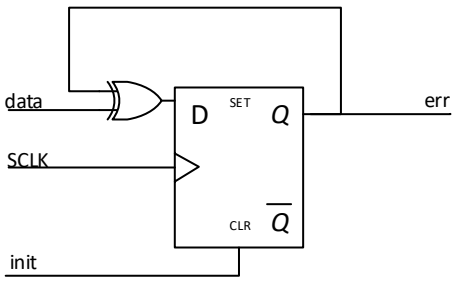


Figura 18 - Diagrama Parity Check

2.3.1.4 counter

O bloco counter é um simples contador de 4 bits e com o apoio de 2 comparadores ambos de 4 bits envia indicação ao bloco Serial Control para dFlag de que o bit recebido é um bit de dados (compara com 9) e para pFlag de que o bit recebido é o bit de paridade (compara com 10).
O contador de 4 bits foi implementado com um somador de 4 bits e quatro registos que ao ritmo de clock adiciona 1 e efetua clear a pedido do bloco Serial Control.

Os comparadores foram implementados de acordo com o esquema da figura 19

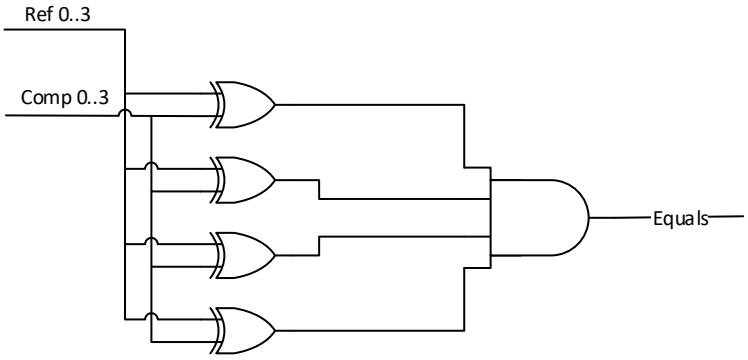


Figura 19 - Diagrama comparador

2.3.2 LCD Dispatcher

O bloco *LCD Dispatcher* é responsável pela entrega das tramas válidas recebidas pelo bloco *Serial Receiver* ao *LCD*, através da ativação do sinal Wr_L. A receção de uma trama válida é sinalizada pela ativação do sinal D_{val}.
O processamento das tramas recebidas pelo *LCD* respeita os comandos definidos pelo fabricante, não sendo necessário esperar pela sua execução para libertar o canal de receção série. Assim, o bloco *LCD Dispatcher* pode ativar, prontamente, o sinal *done* para notificar o bloco *Serial Receiver* que a trama já foi processada (fully interlocked).

O bloco *LCD Dispatcher*, foi implementado com uma maquina de estados representada pelo ASM figura 20.

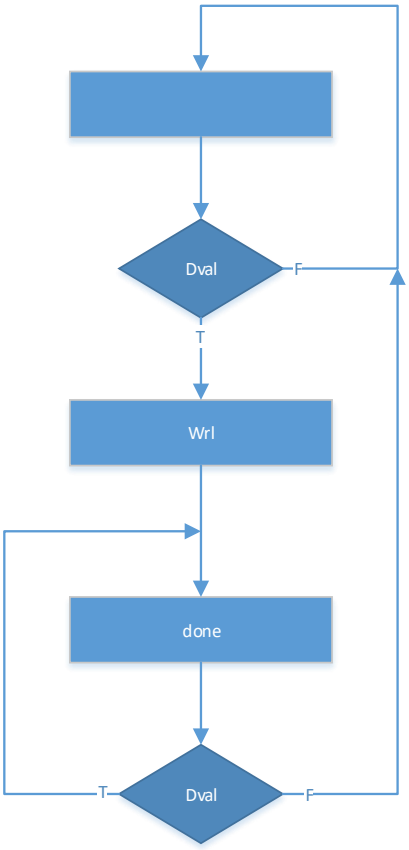


Figura 20 – ASM do LCD Dispatcher

2.4 Serial Sound Controller

O módulo *Serial Sound Controller* (*SSC*) implementa a interface com o gerador de sons, realizando a receção em série da informação enviada pelo módulo de controlo e entregando-a posteriormente ao gerador de sons, conforme representado na Figura 21.

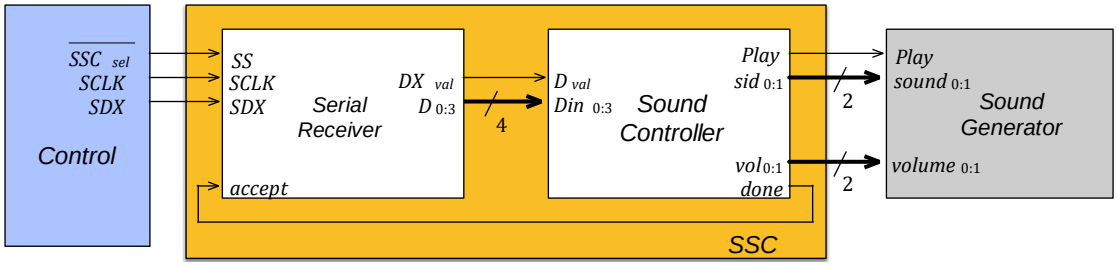


Figura 21 - Diagrama de blocos do módulo Serial Sound Generator Controller

O módulo *SSC* recebe em série uma mensagem composta por quatro bits de informação, segundo o protocolo de comunicação ilustrado na Figura 12. Os dois primeiros bits de informação, indicam o comando a realizar no gerador de sons, segundo a Tabela 1. Os restantes dois bits identificam o som a reproduzir ou o valor do volume. Tal como acontece com o *SLCDC*, o canal de receção série pode ser libertado após a receção da trama recebida pelo *Sound Generator*, não sendo necessário esperar pela sua execução do comando correspondente. Assim, o bloco *Sound Controller* pode ativar, prontamente, o sinal *done* para informar o bloco *Serial Receiver* que a trama já foi processada.

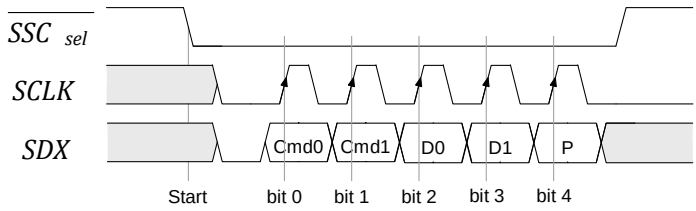


Figura 22 - Diagrama de blocos do módulo Serial Sound Generator Controller

Cmd *1 0*	*Data* *1 0*	*Function*
0 0	* *	*stop*
0 1	* *	*Play*
1 0	S1 S0	*Set sound*
1 1	V1 V0	*Set volume*

Tabela 1 – Comandos do módulo Sound Generator

2.4.1 Serial Receiver

O bloco *Serial Receiver* do módulo *SSC* deve ser implementado adotando, com as devidas adaptações, uma arquitetura similar à do bloco *Serial Receiver* do módulo *SLCDC*, neste caso adaptando a estrutura para a receção de quatro bits de informação em vez de nove bits.

2.4.2 Sound Controller

Após a receção de uma trama válida (proveniente do bloco *Serial Receiver*), o bloco *Sound Controller*, deverá proceder à atuação do comando recebido sobre o gerador de sons. Salienta-se que para reproduzir um som, o gerador de sons necessita de ter presente nas entradas *volume* e *sound* o volume e o identificador do som, respetivamente. O bloco *Sound Controller* funciona em *fully interlocked* com o bloco *Serial Receiver*.

O bloco *Sound Controller*, foi implementado de acordo com o diagrama de blocos figura 23.

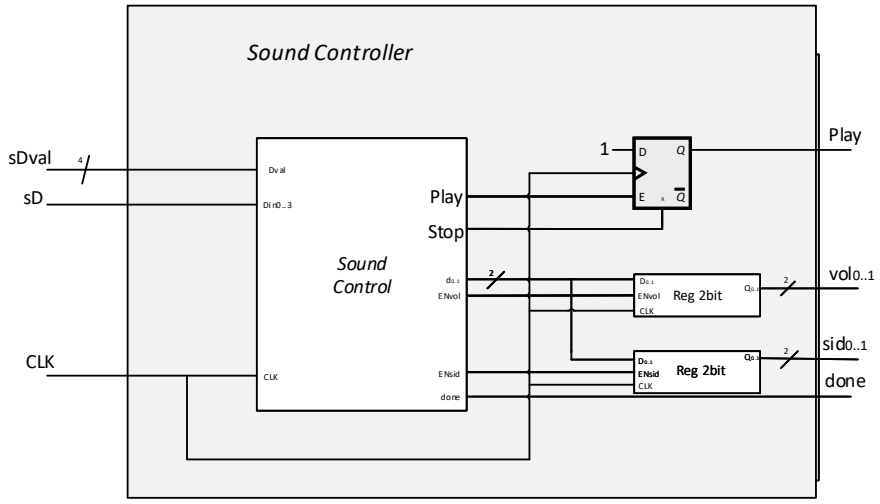


Figura 23 – Diagrama de blocos do Sound Controller

O bloco *Sound Control* é responsável por descodificar os comandos do modulo *Sound Generator*.

As saídas de *volume* e *sound* são controladas com registos de 2 bits cada, em que o enable de cada registo é dado pelo bloco Sound Control, conforme a 1ª coluna da tabela 1.

A saída play é controlada com um registo simples em que o enable recebe informação da saída play do bloco *Sound Control* (comando play) e regista o valor "1" e o reset recebe informação da saída stop do bloco *Sound Control*.

2.4.2.1 Sound Control

O bloco *Sound control* foi implementado com uma máquina de estados representada pelo ASM figura 24, tendo em consideração as saídas de volume e sound de acordo com a codificação da Tabela 1 Comandos do módulo *Sound Generator*

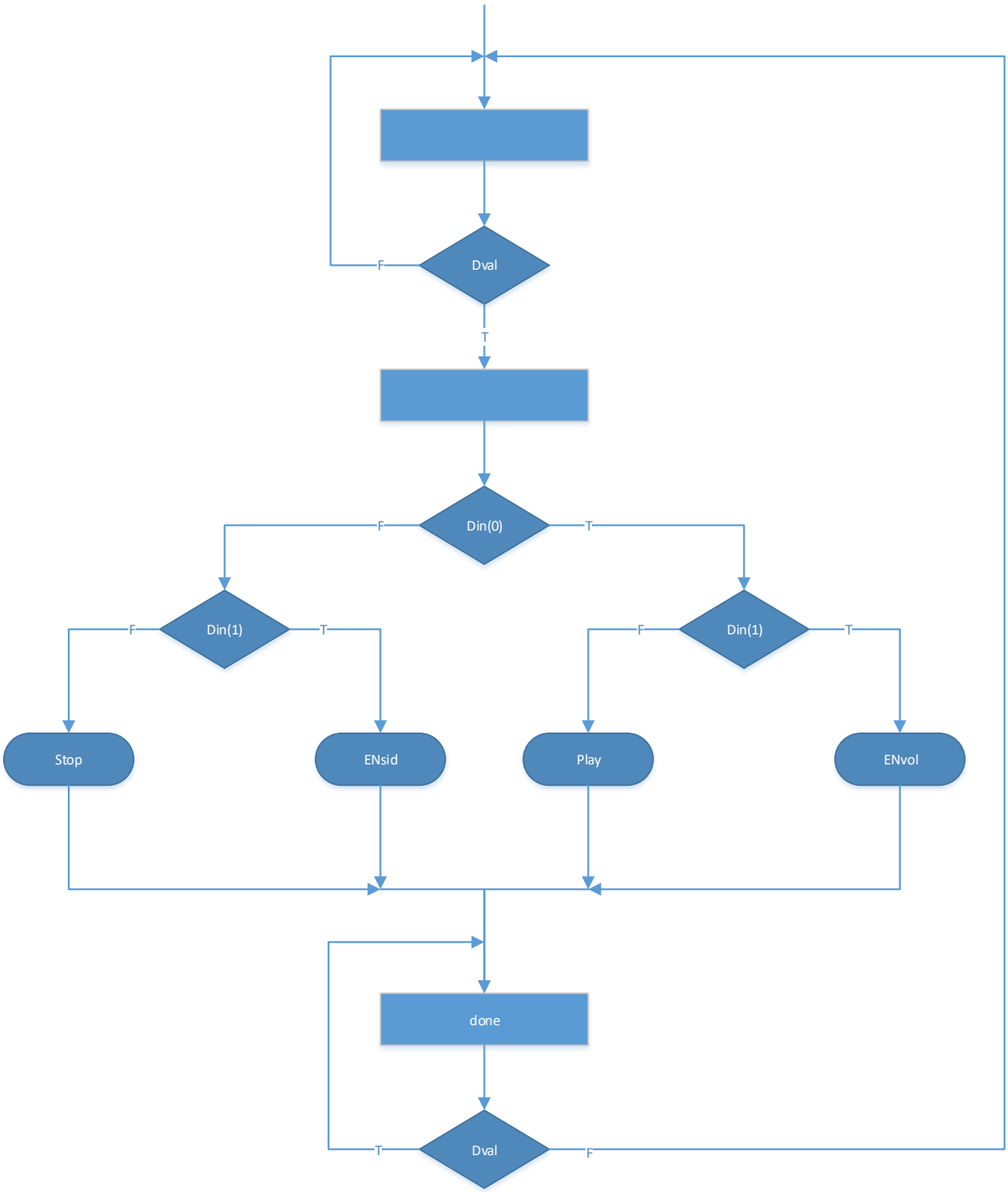


Figura 24 – ADM do bloco Sound Control

2.5 Control

2.5.1 Classe HAL

Metodo **void init():**
Este método inicia a variável estática “lastval” a zero, publicando esse valor nos LEDs 9-19 da µLICχ.

Metodo **int readBits(int mask):**
Método que faz a operação lógica "and" do valor que recebe como argumento e o valor que se encontra no UsbPort e retorna este valor. Este método serve para ir buscar informação (um bit específico) que se encontra nos LEDs de saída.

Metodo **boolean isBit(int mask):**
Vai avaliar se o bit tiver o valor lógico '1' e retorna o resultado

Metodo **void setBits(int mask):**
Atualiza o estado da variável global lastval com o valor recebido como argumento (coloca no valor '1') e envia para UsbPort.

Metodo **void clearBits(int mask):**
Este método faz o inverso do setBits(int mask) e coloca os bits recebitos em "mask" a 0.

Metodo **void writeBits(int mask, int value):**
Método que escreve nos bits desejados(representados por "mask")o valor de value.

Metodo **int in():** e **void out(int val):**
Estes métodos servem apenas para testar se a aplicação está em modo de simulação ou não. Se estiver os bits do UsbPort têm que ser.

Metodo **boolean isMaintenance():**
Método que vai testar se o bit que indica se a aplicação se encontra em manutenção está ativo utilizando a máscara 0x40(0b0100 0000 o segundo bit mais à esquerda do conjunto de 8 bits de informação enviada pelo hardware) e retorna o valor do mesmo.

2.5.2 Classe KBD

Metodo **void init():**
A única operação que faz sentido no método init desta classe será a avaliação se a aplicação está a correr em modo simulação ou não para poder instanciar a variável keyboard com o array correto. Em modo hardware as teclas são avaliadas por coluna-linha. Em modo simulação a avaliação é feita por linha-coluna.

Metodo **char getKey():**
Método que retorna a tecla premida, se hardware não acusa tecla premida este método retorna logo "NONE" ou seja 0. Caso contrario, lê os bits da tecla premida e vai buscar a tecla certa ao index do array, de seguida é enviado um Keyacknowledge. Após isto é fundamental recorrer a uma operação de limpeza ao bit do acknowledge para permitir à máquina de estados avançar para o estado seguinte avaliação.

Metodo **char waitKey(long timeout):**
Este método é semelhante ao getKey() apenas com uma adaptação. Espera um tempo mandado como argumento para que uma tecla seja premida. Caso uma tecla não seja premida dentro desse espaço de tempo o método retorna "NONE".

2.5.3 Classe SerialEmitter

Método **void init():**
O método init desta classe apenas envia para o serial receiver as mascaras “Destination Masks” (0x8,0x4) através do método HAL.setBitscorrespondente a binário a: 1000 e 0100 respetivamente. Estes comandos server para ter a certeza que tanto para o Serial LCD como para o Serial Sound estão desligados e que estão no estado de espera (uma vez que são ativos de forma complementar).

Método **void send(Destination addr, int size, int data):**
Este método envia bit a bit a informação a enviar ao SerialReceiver. No final do ciclo é essencial enviar o bit de paridade para o hardware avaliar se recebeu o número de bits necessários para executar a operação.

Método **void clock():**
Método responsável por dar o clock necessário ao shift register presente no bloco do Serial receiver. Este clock permite o shift dos bits que entram neste bloco até formar um conjunto de 10 bits (incluíndo o bit de paridade).

Método **void sendData(int data):**
Este método chama o writeBits da class HAL para meter o valor do argumento "data" na posição correspondente à mascara "SDX_MASK"(0x1).

2.5.4 Classe CoinAcceptor

Esta classe vai tratar quando ocorre uma inserção de uma moeda.

Método **boolean checkForInsertedCoin():**
único método da classe. Apenas vai confirmar se foi inserida alguma moeda. Se foi inserida uma moeda chama o método da classe HAL setBits() recebendo como parâmetro o valor 0x80(bit que indica o mesmo ao hardware), de seguida chama-se o mesmo método para repor o valor a zero deste bit. Caso não seja inserida nenhuma moeda este método retornará "false".

2.5.5 Classe M

Esta classe vai tratar quando ocorre um pedido de manutenção. Fazem parte desta classe os métodos:

Método **boolean checkIsInMaintenance():**
Vai testar se o bit que dá a informação do modo da manutenção está ativo e retorna um valor booleano consoante se o bit tiver o valor lógico '1'(true) ou '0'(false).

2.5.6 Classe LCD

Método **void init():**

Nesta classe o método init é uma sequencia de comandos default para inicializar o LCD e ainda chama o método CreateSpecialChars(). Há que ter em atenção à Busy Flag(BF), tem que ser mantida no estado Busy Flag(BF = '1') até que esta sequência de comandos termine.

Metodo **void createSpecialChars():**
Este método apenas chama o método SendSpecialChar (LCD_Special_Char schar) chamado com os vários enumerados criados na class LCD_Special_Char como argumento.

Metodo **void sendSpecialChar(LCD_Special_Char schar):**
Este método recebe um Enum da classe LCD_Special_Char e chama o método writeDATA() e envia cada posição do array (byte a bye). Isto vai resultar na gravação na memória do LCD dos Sprites necessários para a aplicação ter uma apresentação gráfica o mais semelhante possível ao desejado.

Método **void writeDATA(int data):**
Este método recebe os dados que se pretende mandar ao LCD como argumento. O funcionamento deste é apenas chamar o método writeByte (boolean rs, int data) com a booleana a true, necessário para indicar ao hardware que se trata de um dado e não de um comando (colocação do bit "RS" a 1) e o inteiro correspondente os dados a enviar.

Método **void writeCMD(int data):**este método é muito semelhante ao writeDATA (int data). A única diferença é a booleana mandada como argumento ao método writeByte (boolean rs, int data) ser a false. Isto vai indicar ao hardware que se trata de um comando (colocação do bit "RS" a 0).

Método **void writeByte(boolean rs, int data):**
Este método prepara os bits a enviar de forma a permitir a inserção do bit de rs. Faz uma operação de shift ao argumento data e adiciona rs(ou '1' ou '0') resultante da tomada de decisão anterior. Após os bits estarem preparados é enviado para o SerialEmitter com o LCD como destino.

Método **void write(char c)** e **write(String txt) :**
Estes dois métodos têm o mesmo comportamento com uma pequena diferença. Ambos chamam a função writeDATA (int data). Enquanto o primeiro escreve um caracter o segundo escreve uma string.

Método **void setCursorPosition(int lin, int col):**
Metodo que envia um comando ao LCD para mover o cursor. A condição deste método serve para decidir em que linha é. Se for na linha 0 faz-se o setAddress com o envio de 0x80 caso contrario ativa-se o bit à direita com a soma de 0x40. Depois soma-se a coluna respetiva e chama-se a função para mandar o comando.

Método **void setCursorBlink(boolean blink):**
Este método envia o comando necessário se for pretendido por o cursor a acender e apagar(0xF, Flag D = '1'(display ligado), Flag C= '1'(cursor ligado), B= '1'(ligar "blink")) ou não(0xC Flag D = '1',Flag C= '0'(cursor desligado), B= '0'("blink" desligado))).

2.5.7 Classe SoundGenerator

Método **void init():**
Chama a função responsável por definir um volume, a função por iniciar a música e inicia a variável "lastsound" que é responsável pela identificação do som que está a tocar ou tocou anteriormente.

Método **void send(int command, int data)**
Este método é responsável por preparar os dados a enviar ao SerialEmiter com a a variavel do SerialEmitter. Destination.SSC que indica que é um comando respectivo ao SerialSoundControler.

Método **void setVolume(int volume)**
A única responsabilidade é chamar o método send (int command, int data) com a mascara "CMD_SETVOL_MASK"(0b0011) e com o volume pretendido.

Método **void stop()**
Repõe o valor da variável lastSOUND e chama o método send com a mascara respetiva para desligar o som(0b0000).

Método **void play(SOUND sound)**
Responsável por mandar dois comandos ao hardware. Um para ligar o som e o outro para definir o som para reproduzir.

Método **void playIfDiferent(SOUND sound)**
Testa se o som enviado como parâmetro é o que está a ser reproduzido. Senão for, este método força a paragem do som que está a ser produzido e manda o comando necessário (chama o método play(SOUND sound)) para reproduzir o som pretendido.

2.5.8 Classe TUI

A classe TUI vai ser a nossa camada de abstração entre o jogo propriamente dito e as três classes de software, explicadas anteriormente, que vão controlar as partes de hardware: Serial Sound Controler, Serial LCD Controler e o KeyboardReader. Estas classes são: LCD, KBD e o SerialEmitter. A existência desta classe é fundamental uma vez que se for necessária a implementação de outra aplicação, poder-se-á reaproveitar tudo o que foi feito.

Método **void init():**
Este método vai ser responsável por iniciar todos os módulos com os quais tem contacto direto. Vai chamar:HAL.init(), SerialEmitter.init(), KBD.init(), LCD.init(). Por fim vai posicionar o cursor de LCD na posição Coluna 0 e Linha 0 para o cursor ficar posicionado pronto a escrever e vai chamar o método SetCursorBlink (false) que vai impedir que o cursor de LCD esteja a piscar.

Método **String getString(int size, int linStart, int colStart):**
Vai ser responsável pela instaciação do array "sarry". Este método tem varias tomadas de decisão, consoante a tecla premida vai instaciando o array "sarry" no index "pos". Esta variável "pos" é incrementada ou decrementada se o utilizador clicar na tecla '6' ou '4' respetivamente. O caracter a escrever no index "pos" é incrementado ou decrementado se o usuário clicar na tecla '2' ou na tecla '8' respetivamente. Caso o utilizador clique na tecla'*' o valor 0 é colocado no index respetivo e a variável que indexa é decrementada. No ultimo caso, o utilizador clicar na tecla '5' o ciclo é terminado e o método retorna o array. Para além deste processo instanciar o array e chamar o método writeSetFinalPos(char c, int linFinal,int colFinal)que vai na realidade, escrever no LCD , este método também retorna a String que foi inserida pelo utilizador, sendo esta a única maneira de ter acesso ao que foi escrito pela ultima vez.

Método **void writeSetFinalPos(char c, int linStart, int colStart):**

Desliga o "blink",escreve no LCD o caracter mandado como parâmetro e posiciona o cursor na posição correta. Quando estas operações forem terminadas volta a ligar o "blink".

Método **void setCursorBlink(boolean cursor):**
Chama o método de LCD explicado mais acima setCursorBlink(boolean cursor).

Método **void sclearScreen():**
Vai escrever uma String vazia, do tamanho em cada linha

Método **void writeAtPosition(char a, int line, int column):**
Escreve o caracter a na posição dada por line e column.

Método **void writeAtPosition(String a, int line, int column):**
Escreve a String "a" na posição dada por "line" e "column".

Método **void createSpecialChars()**
Cria no LCD os catareres especiais.

Esta classe vai comunicar com as outras classes através de métodos que a única função é chamar os métodos estáticos das outras classes de forma a ter contacto com as mesmas. São os casos de **char waitKey(long timeout)** e **char waitKey(long timeout)**.

2.5.9 Classe TUI_Game

Esta classe contem métodos de chamada ao TUI utilizados pela classe Game

2.5.10 Classe TUI_HighScores

Esta classe é uma especialização de TUI para utilização pela classe HighScores

2.5.11 Classe TUI_M

Esta classe contem métodos de chamada ao TUI utilizados pela classe M

2.5.12 Classe TUI_SpaceInvaders

Esta classe contem métodos de chamada ao TUI utilizados pela classe SpaceInvaders

2.5.13 Enumerador TUI_Special_CharTUI

Este enumerado contem os dados de construção dos caracteres especiais

2.5.14 Classe FileAccess

Método **ArrayList<String> load(String fileName, int initialCapacity)**
Este método vai ao ficheiro cujo nome é recebido como parâmetro percorre-o, preenche um ArrayList de Strings e retorna o mesmo.

Método **void save(String fileName, ArrayList<String> SL)**
Grava no ficheiro cujo nome é recebido como argumento as Strings presentes no ArrayList, também ele recebido como argumento.

Caso exista algum erro de execução na execução destes dois métodos a exceção é tratada de forma a que seja imprimido na linha de comandos a razão da mesma.

2.5.15 Classe Statistics

Método **void init():**
Apenas executa o método load() da classe.

Método **void clear():**
Limpa o número de jogos e o numero de moedas da estatística.

Método **void load():**
Este método executa o método clear() fundamental no caso do ficheiro não conter informação necessária para instanciar as variáveis "coins" e "games". Após a execução do método clear() chama o método da class FileAccess load(String fileName, int initialCapacity) para ir buscar os valores presentes no ficheiro e instancia as variáveis "games" e "coins".

Método **void save():**
Apenas executa o método do FileAccess save(String fileName, ArrayList<String> SL) explicado acima.

Método **void addGame():**
Incrementa a variável "games" e chama o método save da própria classe.

Método **void addCoins():**
Incrementa a variável "coins" e chama o método save da própria classe.

2.5.16 Classe Scores

Método **void init():**
Apenas executa o método load() da classe.

Método **void load():**
Limpa o conteúdo do ArrayList "highScores", e preenche-o com o conteúdo presente no ficheiro que contem as pontuações. Após este preenchimento chama a função sortScores().

Método **void save():**
Preenche um novo ArrayList e salva as entradas num ficheiro chamando o método da classe FileAccess save().

Método **void sortScores():**
Ordena as pontuações por ordem decrescente.

Método **boolean isNewScoreHighScore(int score):**
Utilizado apenas para testar se a nova pontuação é maior que a pontuação mais baixa.

Método **void addScore(String name, int score):**

Se o tamanho limite do ArrayList for atingido procede-se à remoção do último elemento da lista, caso contrário, insere na lista, ordena e salva.

Método **void addScore(int score):**
Testa se é nova pontuação mais alta, se não for retorna. Se for, limpa a linha 0 do LCD, escreve a pontuação no LCD. Faz-se a validação do tamanho da String para se por alguma razão, num caso exepcional a String for enviada vazia, não insere.

2.5.17 Classe Score

Método **void getName():**
Retorna o nome do objecto.

Método **void getScore():**
Retorna a pontuação do objecto.

Método **String toString():**
Redefinição do método "toString" da classe mãe Object. É redefinido de forma a que a String retornada seja "score;name".

Método **String fromText(String txt):**
Após a confirmação que o String recebida como parâmetro não está vazia, procede à divisão utilizando ";" como fator divisório e construi um novo objeto "Score".

2.5.18 Classe SpaceInvaders

Método **void init():**
Inicia a classe TUI, SoundGenerator, Statistics, Scores e instancia a variável "SCORES" com o valor 0.

Método **void startMenu():**
Método entra num ciclo infinito e dentro deste ciclo vai testar quatro condições:

Se é para entrar em modo de manutenção:
Executa o método da classe M onMaintenanceMode() explicada mais acima.

Se foi inserida uma moeda:
Insere na classe Statistics dois créditos(cada moeda dá direito a 2 créditos), adiciona este valor à variável "CREDITS" e volta a imprimir o titulo do menu inicial da aplicação.

Se não foi premida nenhuma tecla, ou se a tecla premida foi diferente de asterisco("STARTGAME):
Percorre os HighScores existentes e imprime-os enquanto houver "next" no iterador. Caso o iterador não tenha "next" a o contador é reiniciado e imprime novamente os HighScores do inicio. É importante frisar que os Scores são ordenados de forma decrescente logo, o primeiro Score a aparecer é o mais alto existente no ficheiro. A diferença de tempo entre as amostragens de pontuações é de um segundo porque foi entendido que seria o tempo necessário para o utilizador conseguir visualizar e não ser tempo demais de forma a quebrar o "mementum".

Se a tecla premida foi "STARTGAME" e o número de Créditos é maior que 0:
É decrementado o valor da variável "CREDITS", adiciona um jogo para as estatísticas e inicia um jogo.

2.5.19 Classe Game

Esta classe vai ser aquela que é considerada a principal do jogo uma vez que vai conter toda a lógica do jogo propriamente dito.

Método **char generateGame():**
Na parte inicial deste método são chamados os métodos da classe GameView que são responsáveis por imprimir o estado inicial da aplicação quando entra no modo de jogo:
printMissileToShot(char missile), printHumanShip(), PrintAliens(char[] alienTrain), printScoreText(), printScore(int score), printLevelText(), printLevel(int level).
Após as impressões o método entra num ciclo em que a condição de paragem é ó ultimo caracter chegar à nave. Dentro deste ciclo vai testar se:

É para avaliar uma explosão:
Executa o método evaluateExplosion() e afeta a variável invalidate com o valor true(esta variável dá a informação se é necessário atualizar a vista)

É para inserir um novo alien:
Executa o método evaluateAlien() e afeta a variável invalidate com o valor true.

É para actualizar a vista:
Executa o método da classe GameView PrintAliens(char[] aliensTrain).

Após estas três condições executa o método evaluateLevel()que é explicado em baixo.
Antes de voltar a correr o ciclo testa se houve um disparo e se é necessário repor o som do jogo.

Quando ocorre gameover, executa o método da classeGenerator playIfDiferent(Sound sound) para ocorrer uma transição da música do jogo para a música correspondente ao fim de jogo. Chama também o método da class GameView PrintGameOver(), pára a execução do Sound generator e retorna o score produzido pelo user para que se possa adicionar aos Scores.

Método **void evaluateKey(char key):**
Este método apenas avalia a tecla que foi premida pelo utilizador:
Se foi SHOT_KEY(*):
Executa o método evaluateShoot() explicado mais baixo.
Se foi QUIT_KEY(*):
Força que o último index do array alienTrain seja maior que 10(condição de paragem da execução do jogo).

Se não foi nenhuma das anteriores:
Chama o método updateMissileToShot(char key) explicado mais abaixo

Método **void evaluateShoot():**
Método chama o método playIfDiferent(SOUND sount) para gerar o som correspondente ao míssil disparado, depois entra num ciclo que a condição de paragem é o campo length do array de aliens. Dentro deste ciclo é chamada a função da classe GameView, printMissileShoted(int col)(explicado na secção da

classe GameView) até o index onde o i se encontra for correspondente ao index do primeiro "alien", se o "missile" disparado pelo utilizador for igual ao primeiro "alien", procede-se à animação de explosão e atualiza-se o score e o método sai do ciclo sai do ciclo.

Método **void updateMissileToShot(char key):**
Atualiza a vista do míssil a disparar chamando o método da class GameView printMissileToShot(char missile) enviando como parâmetro a variável local "missileToShot" que foi a tecla premida pelo utilizador.

Método **void evaluateLevel():**
Testa se a pontuação atual justifica a subida de nível, se for o caso é incrementada a velocidas, o nível, a variável scoreToLevelUp(que indica se é altura para incrementar a dificuldade) e actualiza a vista do nível.

Método **char generateAlienNumber():**
Utiliza as variáveis "upper_alien_number" e "lower_alien_number" como limite superior e inferior respetivamente para calcular o número random. Após este calculo é somado o código ASCII base(48). Pois os números nesta tabela começam no numero 48, sendo este o 0.

Método **void addAlienToArray(char[] alienTrain):**
Vai inserir no array de "aliens já existente" o novo número gerado. Este método não só insere o novo alien gerado como ainda é responsável pela movimentação dos "aliens".
Foi implementado com um ciclo que iguala o "char" existente num index ao char correspondente ao char do index seguinte. O último index é preenchido com a chamada ao método generateAlienNumber() explicado acima.

Método **void evaluateAlien():**
Vai testar se é tempo de indexar outro alien no array. Em caso afirmativo retorna true, caso contrário, retorna false.

3 Conclusões

Os módulos desenvolvidos foram testados em laboratório, e usando o simulador fornecido. Na programação da CPLD foram usados 36 pinos de I/O (11 de entrada e 25 de saída). A taxa de utilização das macrocélulas foi 37%, tendo sido usados 33% dos registos e 46% dos pinos disponíveis.

CPLD Fitter Summary

SUMMARY

Design Name	SpaceInvadersWrapper
Fitting Status	Successful
Software Version	P.20131013
Device Used	XC95144XL-10-TQ100
Date	7- 2-2017, 8:03PM

RESOURCES SUMMARY

Macrocells Used	Pterms Used	Registers Used	Pins Used	Function Block Inputs Used
53/144 (37%)	135/720 (19%)	47/144 (33%)	37/81 (46%)	82/432 (19%)

PIN RESOURCES

Signal Type	Required	Mapped	Pin Type	Used	Total
Input	11	11	I/O	36	74
Output	25	25	GCK/IO	1	3
Bidirectional	0	0	GTS/IO	0	4
GCK	1	1	GSR/IO	0	1
GTS	0	0			
GSR	0	0			

GLOBAL RESOURCES

Signal mapped onto global clock net (GCK1)	CLK

POWER DATA

Macrocells in high performance mode (MCHP)	53
Macrocells in low power mode (MCLP)	0
Total macrocells used (MC)	53

Os resultados obtidos foram satisfatórios. As limitações encontradas na jogabilidade derivam dos constrangimentos impostos pelo número de linhas e colunas do LCD-MC1602C e do *overhead* do protocolo de comunicação USB.

"O que teríamos mudado fôssemos começar o projeto hoje?"
- Mais planeamento, menos urgência em testar e obter resultados.
- No desenvolvimento hardware, teriamos feito uso de módulos de *testbenching*, por forma a enveredar por outro modo de *debugging,* consolidando esses conhecimentos e dando outro tipo de suporte aos resultados obtidos.

Futuros desenvolvimentos
O jogador iniciar com três vidas, reforçando assim a dependência entre duração do jogo e a habilidade do jogador.
Substituir o LCD utilizado por um ecrã com mais linhas.
Com alguma facilidade, o jogo dos invasores espaciais poderia ser substituído por um jogo de congruências matemáticas, substituindo a classe TUI e suas especializações, bem como SpaceInvaders e Game.

4 Referências

[1] *μLICχ* Reference Manual, p.1-2
Página da disciplina no Thoth, em particular as ligações encontradas no separador "Laboratório"

5 Apêndice A – Listagem de código VHDL

6 Apêndice B – Listagem de código JAVA

JAVAFile - D:\LIC\Project\SpaceInvaders\src\CoinAcceptor.java

```
 1 class CoinAcceptor {
 2
 3     public static void main(String[] args) {
 4         HAL.init();
 5         int c=0;
 6         while (true){
 7             if ( checkForInsertedCoin()) {
 8                 c++;
 9                 System.out.println("Coin accepted " + c);
10             }
11         }
12     }
13
14
15     private final static int COIN_MASK=0x80;
16     private final static int COIN_ACCEPT_MASK=0x40;
17
18     public static boolean checkForInsertedCoin(){
19         if(HAL.isBit(COIN_MASK)) {
20             HAL.setBits(COIN_ACCEPT_MASK);
21             HAL.clrBits(COIN_ACCEPT_MASK);
22             return true;
23         }
24         return false;
25     }
26 }
27
```