

Laboratório de Introdução à Arquitetura de Computadores

IST - LEIC

2021/2022

Rotinas

Guião 5

23 a 27 de maio de 2022

(Segundo guião da semana 3)

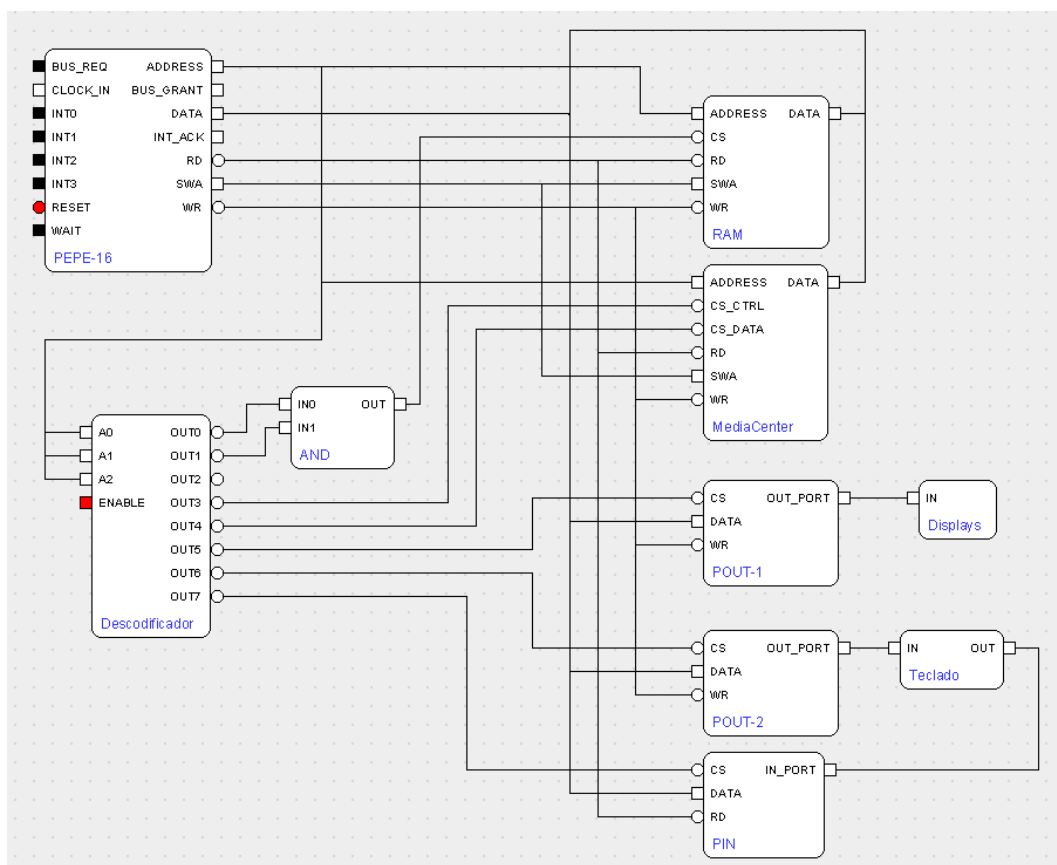
1 – Objetivos

Com este trabalho pretende-se que os alunos pratiquem programação em linguagem *assembly* do PEPE usando rotinas, para processamento e interação com periféricos.

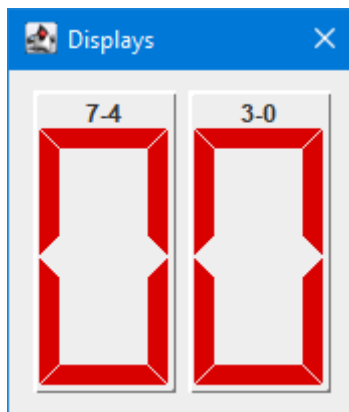
2 – Rotinas em linguagem assembly

2.1 – O circuito de simulação

Clique em **Load** (📁) e carregue o circuito contido no ficheiro **lab5.cir**. Este circuito é idêntico ao já usado no Guião 4 e será novamente usado para interação com o módulo MediaCenter e os displays, mas agora com rotinas.



O periférico POUT-1 é de 8 bits e está ligado a dois displays de 7 segmentos, um ligado aos bits 7-4 (designado High) e o outro ligado aos bits 3-0 (Low).



Para escrever um valor nos dois displays, basta executar uma instrução **MOVB** no endereço **A000H**, em que o periférico POUT-1 está disponível, tal como no guiões anteriores:

Dispositivo	Endereços
RAM	0000H a 3FFFH
MediaCenter (acesso aos comandos)	6000H a 6063H (ver secção 2.3)
MediaCenter (acesso à sua memória)	8000H a 8FFFH
POUT-1 (periférico de saída de 8 bits)	0A000H
POUT-2 (periférico de saída de 8 bits)	0C000H
PIN (periférico de entrada de 8 bits)	0E000H

2.2 – Invocação de rotinas

Passe para “Simulation” e faça clique no PEPE e nos displays para abrir os respetivos painéis de simulação.

Carregue o programa *assembly lab5-rotinas.asm*, com **Load source** (📁) ou *drag & drop*. Este programa contém duas rotinas, uma para escrever um valor entre 0 e FH no display Low e outra correspondente para o display High.

Observe o programa e tente perceber o seu funcionamento. Note:

- A declaração do *stack* (pilha) na zona de dados e a etiqueta no fim dessa área. Para tal usa-se a diretiva **STACK**, idêntica à diretiva **TABLE** (reserva espaço de memória), mas com a vantagem de proteção (se houver um erro de tal forma que uma operação que envolva a pilha tente aceder a um endereço fora da pilha, o PEPE dá um erro);
- A inicialização do *stack pointer* (**SP**), no início do código, com essa etiqueta (a pilha usa-se a partir do fim, para trás). Esta inicialização é obrigatória;
- As instruções **CALL**, seguidas da etiqueta do endereço de uma rotina, para a invocar;

- As instruções **RET**, no fim de cada rotina, para regressar à instrução seguinte ao **CALL** que invocou essa rotina;
- A inicialização do registo que serve de parâmetro para passar a cada rotina.

A figura seguinte mostra este programa carregado no PEPE, na View “Code only”. Tem menos informação do que a View “Source only”, ilustrada na figura seguinte, mas em compensação veem-se mais instruções simultaneamente. Use a View que lhe der mais jeito, podendo seleccionar qualquer delas a qualquer altura.

Note a zona dos dados, a partir do endereço 1000H, e o espaço ocupado pela pilha (pode ter de esticar um pouco a janela para ver tudo).

The screenshot shows the PEPE-16 simulation tool interface. The main window displays assembly code in the "Code only" view. The code starts at address 0000 H and ends at 1000 H. The code includes instructions for moving data, calling subroutines, and returning. The stack is initialized at 1000 H. The main registers (R0-R11, SP, BTE, SSP, USP) are shown on the right, all set to 0000 H. The flags and pending interrupts section shows various status bits. The program symbols table at the bottom lists labels and their addresses.

Program

Addr	Label	Mnemonic	Arguments
0000 H	início:	MOV	SP, SP_inicial
0004 H		MOV	R0, 7
0006 H		CALL	hexa_low
0008 H		MOV	R0, 3
000A H		CALL	hexa_high
000C H		MOV	R0, 0AH
000E H		CALL	hexa_low
0010 H		MOV	R0, 0FH
0012 H		CALL	hexa_high
0014 H	fim:	JMP	fim
0016 H	hexa_low:	MOV	R1, NIBBLE_3_0
0018 H		AND	R0, R1
001A H		MOV	R2, imagem_hexa
001E H		MOVB	R3, [R2]
0020 H		MOV	R1, NIBBLE_7_4
0024 H		AND	R3, R1
0026 H		OR	R3, R0
0028 H		MOVB	[R2], R3
002A H		MOV	R1, DISPLAYS
002E H		MOVB	[R1], R3
0030 H		RET	
0032 H	hexa_high:	MOV	R1, NIBBLE_3_0
0034 H		AND	R0, R1
0036 H		SHL	R0, 4
0038 H		MOV	R2, imagem_hexa
003C H		MOVB	R3, [R2]
003E H		MOV	R1, NIBBLE_3_0
0040 H		AND	R3, R1
0042 H		OR	R3, R0
0044 H		MOVB	[R2], R3
0046 H		MOV	R1, DISPLAYS
004A H		MOVB	[R1], R3
004C H		RET	
		PLACE	1000H
1000 H	pilha:	STACK	100H
1200 H	SP_inicial:		
	imagem_hexa:	BYTE	0

Main registers

Register	Value	Level
PC	0000 H	System
R0	0000 H	
R1	0000 H	
R2	0000 H	
R3	0000 H	
R4	0000 H	
R5	0000 H	
R6	0000 H	
R7	0000 H	
R8	0000 H	
R9	0000 H	
R10	0000 H	
R11	0000 H	
SP	0000 H	
RE	0000 H	
BTE	0000 H	
TEMP	0000 H	
SSP	0000 H	
USP	0000 H	

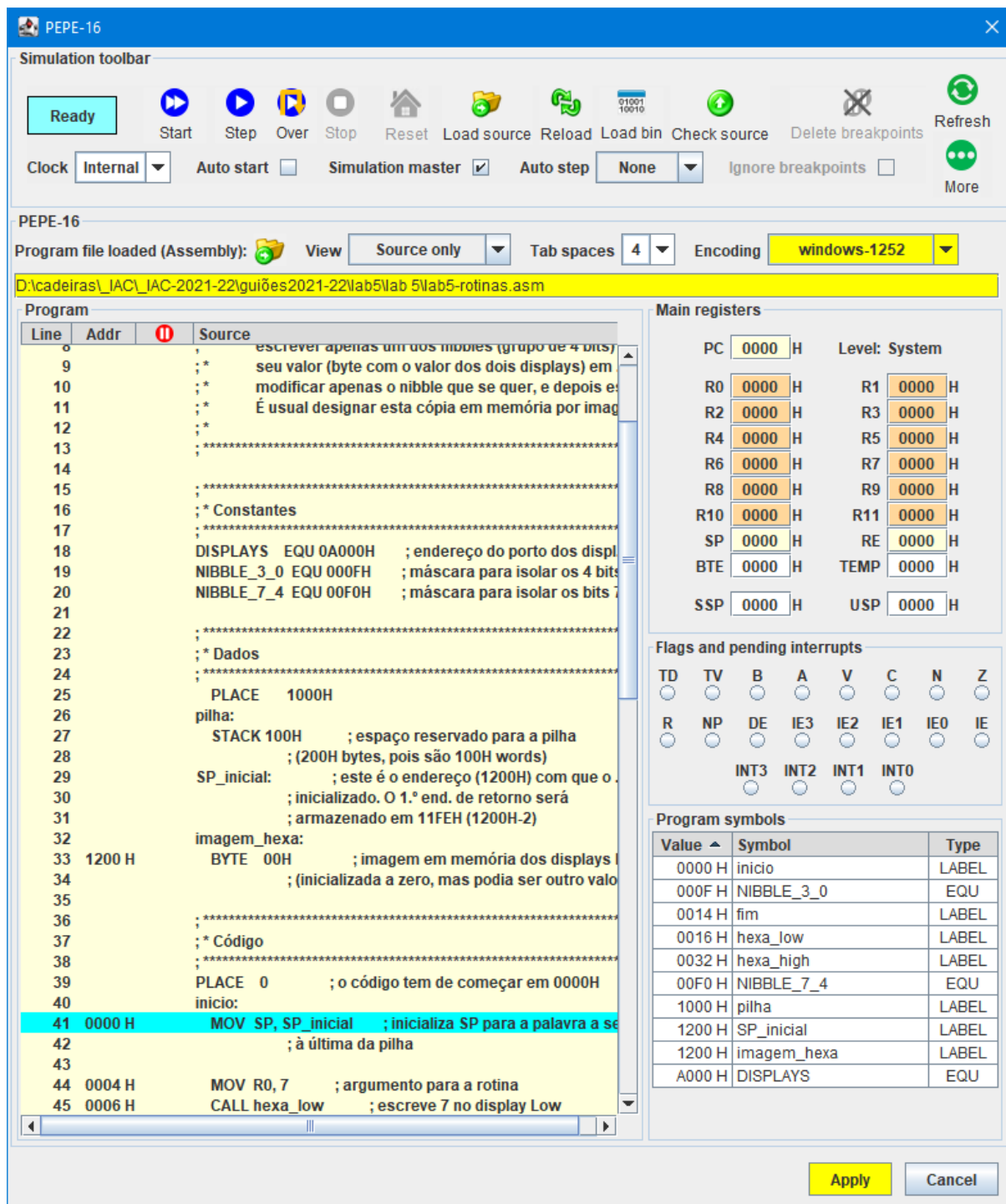
Flags and pending interrupts

Flag	Status
TD	0
TV	0
B	0
A	0
V	0
C	0
N	0
Z	0
R	0
NP	0
DE	0
IE3	0
IE2	0
IE1	0
IE0	0
IE	0
INT3	0
INT2	0
INT1	0
INT0	0

Program symbols

Value	Symbol	Type
0000 H	início	LABEL
000F H	NIBBLE_3_0	EQU
0014 H	fim	LABEL
0016 H	hexa_low	LABEL
0032 H	hexa_high	LABEL
00F0 H	NIBBLE_7_4	EQU
1000 H	pilha	LABEL
1200 H	SP_inicial	LABEL
1200 H	imagem_hexa	LABEL
A000 H	DISPLAYS	EQU

Apply **Cancel**



Execute o programa passo a passo, com o botão **Step** (🔍), SEM carregar em **Start** (▶️).
Verifique que o **CALL** transfere o controlo para a rotina e que o **RET** faz regressar à instrução a seguir ao **CALL**.



Verifique também a evolução do **SP** (que na janela do PEPE é o **SSP**). Um **CALL** reduz o **SP** de 2 unidades e um **RET** incrementa-o de 2 unidades.


Vá observando a evolução das instruções (nomeadamente a chamada e retorno das rotinas) e os displays, até chegar à última instrução do programa. Este exemplo é o suficiente para ilustrar o funcionamento das rotinas.


Termine a execução do programa, carregando no botão **Stop** (⏏) do PEPE.






2.3 – Execução passo a passo com *Step Over*

Faça *reset* ao PEPE, com clique no botão **Reset** () , para repor o programa na situação inicial.

Execute novamente o programa em passo a passo, mas desta vez usando o botão **Over** () em vez do botão **Step** () e tendo particular atenção às instruções **CALL**.

Note que, ao fazer *step-over* num **CALL**, o processador passa “por cima” do **CALL** e faz pausa na instrução com o endereço seguinte, e não na primeira instrução da rotina invocada, que é o que aconteceria com o botão **Step** () . Isto significa que a rotina foi invocada e executada à velocidade máxima, como se de uma só instrução se tratasse.

O botão **Over** () é muito útil para seguir um programa sem ter de entrar dentro de rotinas, interpolando os detalhes desnecessários.


Tal como o botão **Step** () , o botão **Over** () também funciona em auto-step, ficando com um aspeto diferente () . Tirando as instruções **CALL**, os botões **Step** () e **Over** () , bem como as suas versões auto-step, são equivalentes e funcionam de igual modo nas restantes instruções.

2.4 – Preservação de registos nas rotinas

Um **CALL** esconde as instruções da rotina invocada. O problema é que esta pode alterar valores de registos, sem que o código onde está o **CALL** se aperceba. Desta forma, os valores desses registos podem ser uns antes do **CALL** e outros logo a seguir, após o retorno. Como uma rotina pode ser invocada de vários sítios, a situação fica descontrolada porque já não se sabe que registos são ou não importantes e cujos valores a rotina não deve mesmo alterar.



Para resolver este problema, a regra geral é que uma rotina não deve alterar o valor de nenhum registo (exceto os de resultado). Mas a rotina precisa de usar registos para poder trabalhar, e com isso destrói os valores anteriores destes registos!

A solução é a rotina, no início, guardar na pilha (com **PUSH**) os valores dos registos que vai alterar e no fim repor todos esses registos a partir da pilha (com **POP**).

Faça agora **Load source** () ou *drag & drop* do programa *assembly lab5-registos.asm*. Este programa é igual ao anterior, com a exceção de que os registos usados pelas rotinas (**R1**, **R2** e **R3**) são preservados na pilha pelas próprias rotinas, com instruções **PUSH** e **POP**.

Estes registos são inicializados com valores para que se possa observar que após o retorno de uma rotina esses valores são repostos.

Note que a ordem dos **POPs** é inversa da dos **PUSHs**, pois a pilha funciona num regime LIFO (*Last In, First Out*).

Execute o programa do ficheiro passo a passo, com o botão o botão **Step** () , SEM carregar no botão **Start** () . Verifique que os **POPs** repõem os valores guardados pelos **PUSHs** e que **PUSHs**, **POPs**, **CALLs** e **RETs** funcionam bem em conjunto (todos usam a pilha e alteram o **SP** – o *Stack Pointer*).

Vá observando a evolução das instruções e dos registos (nomeadamente o **SP**, **R1**, **R2** e **R3**). A funcionalidade do programa é igual à versão anterior.

Note que o **SP** desce agora mais, por causa dos **PUSHs**, mas no fim de cada rotina volta ao valor inicial. Cada **PUSH** reduz o **SP** de 2 unidades e cada **POP** incrementa-o de 2 unidades.

Como boa prática de programação, cada rotina deve preservar os valores de todos os registos que altera (mas apenas desses, e não de todos os registos!).

Termine a execução do programa, carregando no botão **Stop** (■) do PEPE.

2.5 – Rotinas que chamam rotinas

Faça agora **Load source** (📁) ou *drag & drop* do programa *assembly lab5-variante.asm*. Este programa mantém a funcionalidade dos anteriores, mas inclui uma rotina (**hexa_display**) que recebe um byte e o escreve nos dois displays, chamando as outras duas rotinas (claro que o poderia fazer escrevendo simplesmente o byte no periférico...).

Note o uso das instruções de deslocamento para eliminar os bits que devem ser irrelevantes e para colocar os bits certos no lugar certo.

Execute o programa do ficheiro passo a passo, com o botão o botão **Step** (▶), SEM carregar no botão **Start** (▶). Vá observando a evolução das instruções e dos registos (em particular, o **SP** – o *Stack Pointer*). Note que a pilha mantém o estado da rotina **hexa_display** quando esta invoca outra. O **SP** desce ainda mais baixo que no caso anterior (porque agora há uma rotina que invoca outras), mas no fim do programa volta ao valor inicial.

Termine a execução do programa, carregando no botão **Stop** (■) do PEPE.

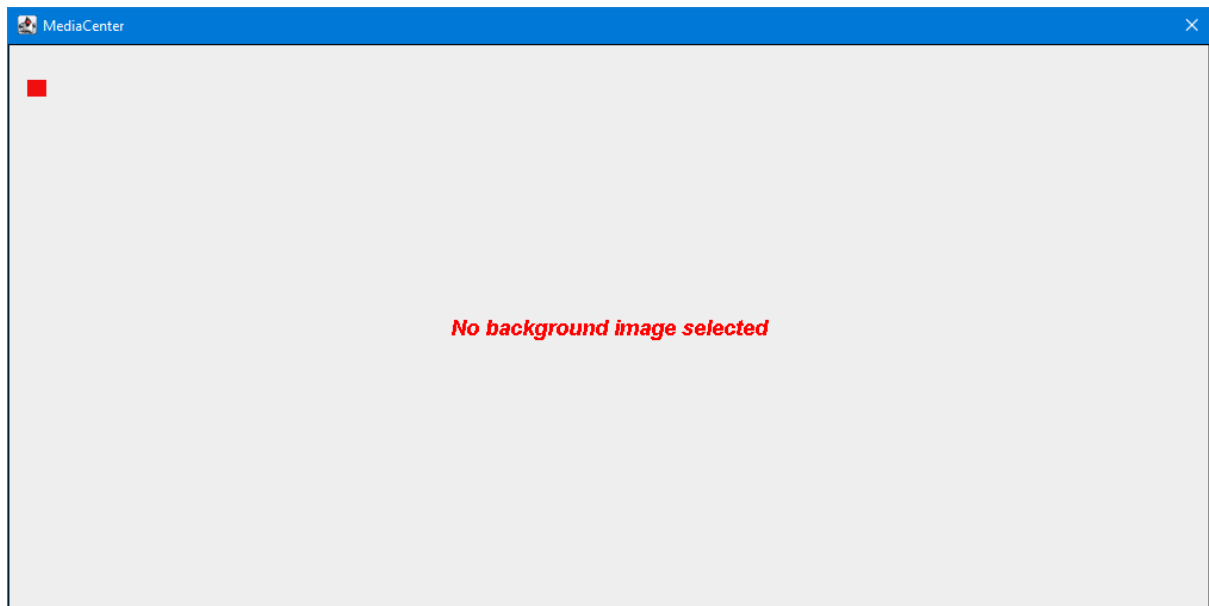
Qualquer programa, seja em *assembly* ou numa linguagem de alto nível como Python, consiste num conjunto de rotinas (em alto nível, chamam-se funções ou métodos). Existe uma que é principal e que invoca outras, que por sua vez invocam outras, até chegar a rotinas que já não invocam mais nenhuma e que apenas retornam, quando terminam. A pilha mantém a história de invocação, para que cada rotina possa retornar para aquela que a invocou.

3 – Rotinas para controlar o módulo MediaCenter

Recuperamos agora o módulo MediaCenter, já usado no guião de laboratório 4, mas agora controlado por meio de rotinas.

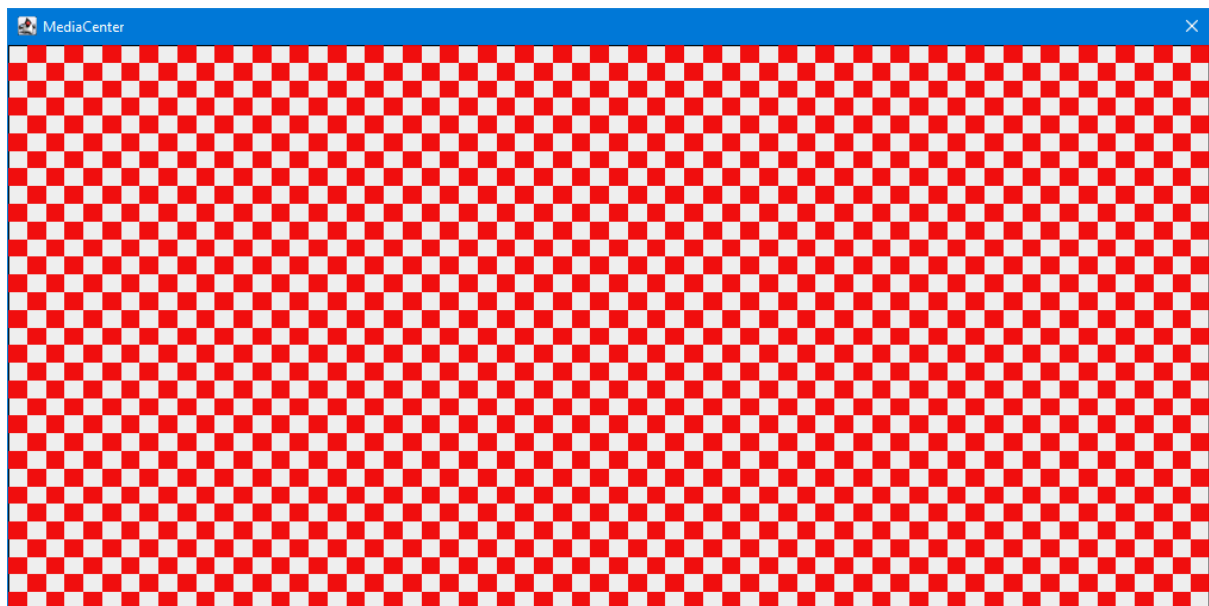
Para exemplificar o uso de comandos com o MediaCenter, edite e observe o programa contido no ficheiro **lab5-comandos.asm**. Contém uma rotina que recebe a linha, coluna e a cor do pixel e implementa a funcionalidade do programa **lab4-comandos.asm**, mas agora com rotinas.

Passe o simulador para “Simulation” e carregue o ficheiro **lab5-comandos.asm** no PEPE (com o botão **Load source**, 📁, ou por *drag & drop*). Execute-o, com o botão **Step** (▶) ou o botão **Over** (⏮) e verifique que o pixel da linha 2, coluna 1, se liga e depois se apaga. A mesma rotina (**escreve_pixel**) pode ser usada para ligar ou apagar o pixel.



Carregue agora o programa contido no ficheiro **lab5-comandos-xadrez.asm**.

Este programa usa a rotina do programa anterior para escrever um pixel numa dada linha e coluna para implementar uma rotina que escreve uma linha de pixels, que por sua vez é usada para escrever todas as linhas do ecrã. De cada vez que escreve um pixel ou uma linha troca o valor do pixel (entre desligado e vermelho), e o resultado, quando o executa com o botão **Start** (▶), é um padrão em xadrez, tal como se pode ver na figura seguinte.

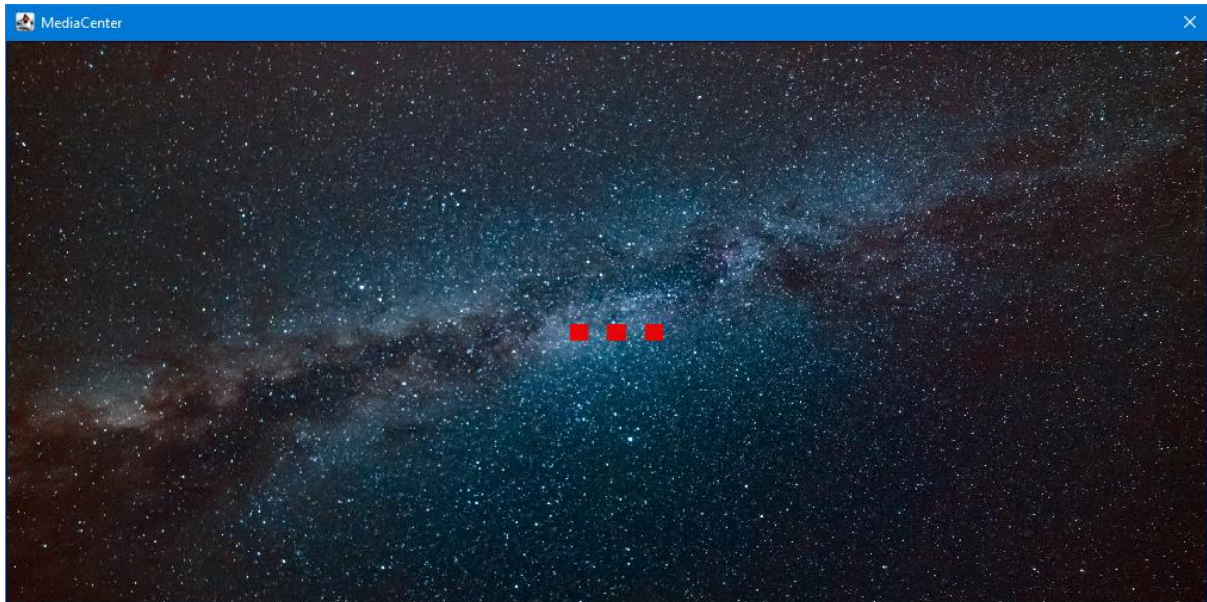


Este programa implementa a mesma funcionalidade que o programa **lab4-comandos-xadrez.asm**, mas agora com rotinas. Também ilustra a chamada de rotinas de mais baixo nível por outras de mais alto nível.

Note que com a estruturação em rotinas perde-se desempenho (há **CALLs** e **RETs**, **PUSHes** e **POP**s) mas ganha-se em clareza, pois fica mais explícito o que o programa faz. Como o recurso mais crítico é o programador, estruturar em rotinas vale a pena.

Termine a execução do programa, carregando no botão **Stop** (●) do PEPE.

Carregue agora e execute o programa contido no ficheiro **lab5-boneco.asm**. Este programa implementa a mesma funcionalidade do que o programa do ficheiro **lab4-boneco.asm**, isto é, desenhar um boneco num cenário de fundo, mas agora com uma rotina **desenha_boneco**, que vai percorrendo a tabela que define o boneco para saber a cor de cada pixel e invocando a rotina **escreve_pixel** para desenhar cada um dos pixels. O resultado obtido é o mesmo:



Carregue agora e execute o programa contido no ficheiro **lab5-move-boneco.asm**. Este programa implementa a mesma funcionalidade do que o programa do ficheiro **lab4-move-boneco.asm**, isto é, movimentar um boneco num cenário de fundo de um lado para o outro, mas agora com uma estruturação numa série de rotinas que tornam o programa mais claro sobre o que faz.

As rotinas têm como utilidade básica:

- Poderem ser invocadas várias vezes, em sítios diferentes do programa, usando a mesma funcionalidade (eventualmente com argumentos) sem se ter de copiar o seu código várias vezes;
- Mesmo que só sejam invocadas uma vez, o seu encapsulamento dentro de uma rotina torna o código mais simples de perceber, não só porque onde são chamadas só aparece o **CALL** (em vez de todas as suas instruções) mas também porque o seu cabeçalho permite explicar a sua funcionalidade.

É importante cada rotina ter o seu próprio cabeçalho, não necessariamente como indicado nos exemplos, mas de qualquer forma devendo permitir tornar claro que se trata de uma rotina, indicando o que ela faz e quais os seus argumentos (de entrada e de saída).

Os registos alterados devem ser **PUSHed** no início e **POPed** no fim, exceto os registos que constituam valores de saída, como por exemplo o R7 na rotina **testa_limites** do programa no ficheiro **lab5-move-boneco.asm**.

É importante cada rotina ter apenas um ponto de saída, onde são feitos os **POPs**. Se houver outros pontos intermédios onde se queira retornar da rotina, deve-se saltar para o único ponto de saída da rotina. O label **sai_testa_limites** na rotina **testa_limites** ilustra este aspeto. Desta forma, garante-se que cada rotina só tem um conjunto de **PUSHes** e de **POPs**.

Carregue agora e execute o programa contido no ficheiro **lab5-move-boneco-teclado.asm**. Este programa implementa a funcionalidade de mover o boneco, mas agora sob controlo do teclado em vez de se movimentar de forma autónoma.

A tecla C fá-lo movimentar para a esquerda e a D para a direita. A rotina **teclado** lê apenas a última linha, seguindo o exemplo do guião de laboratório 3. No projeto, terá de adaptar esta rotina para varrer as linhas todas do teclado e detetar qualquer tecla.

Neste exemplo, o boneco move-se enquanto se estiver a carregar numa tecla, pois logo a seguir a um movimento segue-se para nova chamada à rotina **teclado**, e portanto enquanto a tecla estiver carregada vão-se efetuando os movimentos.

A rotina **testa_limites** está algo diferente, pois este é um dos comportamentos que o projeto requer. Se o boneco chegar a um dos limites, não inverte o sentido, apenas não se movimenta mais.

No entanto, também é importante (vai precisar para o projeto) contemplar situações em que o premir de uma tecla só tenha um efeito, e que seja preciso deixar de carregar em alguma tecla e carregar de novo para repetir o efeito.

Carregue agora e execute o programa contido no ficheiro **lab5-move-boneco-teclado-toque.asm**. Este programa é semelhante ao anterior, com duas diferenças:

- O boneco só anda um pixel de cada vez que se carrega numa tecla (C ou D). De cada vez que se carrega, anda mais um pixel (mas só um, não movimento em contínuo). Isto consegue-se facilmente introduzindo um segundo ciclo de leitura de teclado, mas em que agora se espera até não haver nenhuma tecla carregada (e nessa altura passa-se ao ciclo em que se espera que alguma tecla seja carregada);
- O carregar numa tecla produz um efeito sonoro (não soaria bem no exemplo anterior, pois o som seria reproduzido múltiplas vezes antes sequer de ter tempo de terminar).

4 – Considerações em relação ao Projeto

O módulo MediaCenter é fundamental para o projeto, com imagens para os cenários de fundo, sons, vídeos e bonecos desenhados nos ecrãs de pixels.

Imagens, sons e vídeo são ficheiros que basta, essencialmente, seleccionar/reproduzir.

Os bonecos nos ecrãs de pixels, no entanto, têm de ser desenhados, pixel a pixel, de acordo com formas e cores previamente definidas, em posições que podem ir variando, quando os bonecos se movimentam no ecrã.

Para gerir o desenhar e movimentar destes bonecos, ficam as seguintes sugestões:

- Cada boneco tem em cada instante uma dada posição no ecrã (linha, coluna), que deve memorizar;
- Mover um boneco é apagá-lo na sua posição corrente, alterar essa posição e voltar a desenhá-lo na nova posição;
- A posição de um boneco é na realidade a posição de um dos seus pixels, que é tomado como referência (por exemplo, o canto superior esquerdo). Sabendo essa posição, os restantes pixels são desenhados relativamente a ela (na coluna a seguir, na linha de

baixo, etc.). Mudar a posição de um boneco é mudar a posição do seu pixel de referência;

- Não desenhe um boneco “à mão”, com comandos sucessivos, com mudança manual da posição dos vários pixels do boneco!
- Descreva os vários pixels de um boneco por meio de uma tabela, que inclua a largura, altura e cores de cada um dos pixels. Programe uma rotina genérica que receba o endereço desta tabela, bem como a linha e a coluna do pixel de referência. A rotina deve então percorrer a tabela e ir desenhando o boneco, pixel a pixel. Esta rotina serve para desenhar qualquer boneco cuja descrição siga este formato;
- Note que as variáveis que memorizam a linha e a coluna do pixel de referência não devem, em regra, estar nesta tabela, pois o valor da posição pode variar, ao contrário da forma e cores. Também pode haver várias instâncias do mesmo boneco (descritas pela mesma tabela, mas em posições diferentes);
- Nos exemplos anteriores, foram usados apenas registros para conter as variáveis do programa. Quando este começa a ter alguma dimensão, tal não é prático, e as variáveis que mantêm o estado do programa (por exemplo, a posição do boneco) devem estar definidas em memória (com WORD e BYTE).