

Laboratório de Introdução à Arquitetura de Computadores

IST - Taguspark

2017/2018

Rotinas

Guião 5

30 de outubro a 3 de novembro 2017

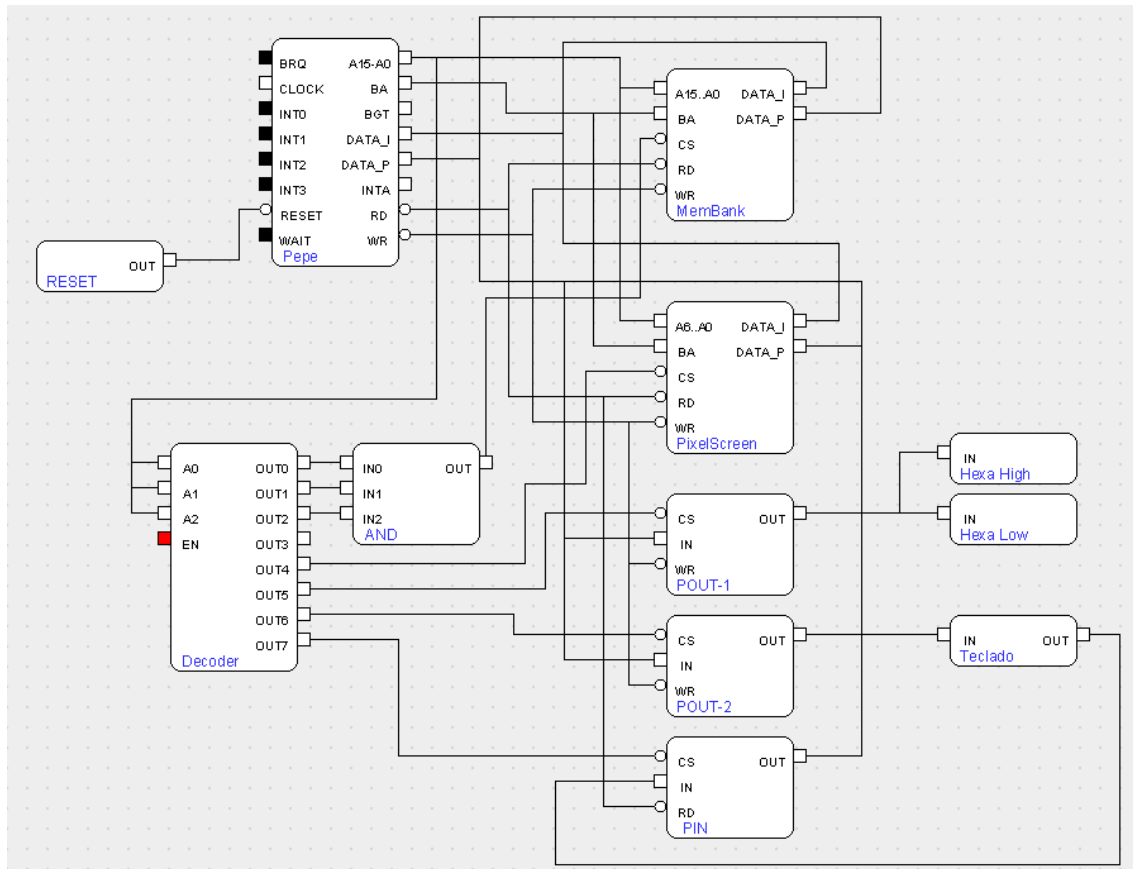
(Semana 7)

1 – Objetivos

Com este trabalho pretende-se que os alunos pratiquem programação em linguagem *assembly*, usando rotinas.

2 – O circuito de simulação

Use o circuito contido no ficheiro **lab5.cmod**. Este circuito é semelhante ao já usado no Guião 4, mas tem mais um periférico (PixelScreen, um ecrã gráfico de 32 x 32 pixels). Este circuito constitui mais um passo em direção ao circuito do projeto.



Para este guião são relevantes os seguintes módulos:

- Matriz de pixels (PixelScreen) – ecrã de 32 x 32 pixels. As coordenadas de um dado pixel no ecrã são dadas em linha e coluna, em que ambas podem variar entre 0 e 31. No entanto, o PixelScreen não passa de uma memória de 128 bytes (4 bytes em cada linha, 32 linhas). Assim, as coordenadas em linha e coluna têm de ser convertidas para um endereço de byte (em que o pixel se localiza) e para o peso do bit (7 a 0), dentro desse byte, que corresponde a esse pixel. Atenção: Em cada byte no ecrã, o pixel mais à esquerda tem o peso 7 (e não 0, como seria de esperar, atendendo a que a coluna cresce da esquerda para a direita);
- Dois displays de 7 segmentos, High (ligado aos bits 7-4 do periférico POUT-1) e Low (ligado aos bits 3-0 do periférico POUT-1);
- Teclado, de 4 x 4 botões, com 4 bits (linhas) ligados ao periférico POUT-2 e 4 bits (colunas) ligados ao periférico PIN (sempre nos bits 3-0). A deteção de qual botão está carregado é feita por varrimento.

O painel de controlo do PixelScreen, displays e teclado deverá ser aberto em execução (modo Simulação).

O mapa de endereços (em que os dispositivos podem ser acedidos pelo PEPE) é o seguinte:

Dispositivo	Endereços
RAM (MemBank)	0000H a 5FFFH
PixelScreen (ecrã)	8000H a 807FH
POUT-1 (porto de saída de 8 bits)	0A000H
POUT-2 (porto de saída de 8 bits)	0C000H
PIN (porto de entrada de 8 bits)	0E000H

3 – Execução do trabalho de laboratório



3.1 – Invocação de rotinas

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

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

- A declaração do stack na zona de dados e a etiqueta no fim dessa área;
- A inicialização do SP, no início do código, com essa etiqueta (o stack 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.

Execute o programa passo a passo, com o botão STEP () , sem carregar no botão de execução () . 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.

Este exemplo é o suficiente para ilustrar o funcionamento das rotinas.


Termine a execução do programa, carregando no botão  do PEPE.

3.2 – 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 são 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 estragar.



Para resolver este problema, a regra geral é que uma rotina não deve estragar o valor de nenhum registo. Mas a rotina precisa de usar registos para poder trabalhar, e com isso destrói os seus valores.

A solução é a rotina, no início, guardar no stack (com PUSH) os valores dos registos que destrói e no fim repor todos esses registos a partir do stack (com POP).

Carregue e compile () agora o 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 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 STEP () , sem carregar no botão de execução () . Verifique que os POPs repõem os valores guardados pelos PUSHes e que PUSHes, POPs, CALLs e RETs funcionam bem em conjunto (todos usam o stack e alteram o SP).


Vá observando a evolução das instruções e dos registos (nomeadamente o SSP – o Stack Pointer – e o R1, R2 e R3). O funcionamento do programa é igual à versão anterior.

No entanto, note que o SSP 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.


Termine a execução do programa, carregando no botão  do PEPE.

3.3 – Rotinas que chamam rotinas

Carregue e compile () agora o programa assembly **lab5-variante.asm**. Este programa mantém o comportamento dos anteriores, mas inclui uma rotina (hexa_display) que recebe um byte e escreve-o 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 passo a passo, com o botão STEP (), sem carregar no botão de execução (). Vá observando a evolução das instruções e dos registos (em particular, o SSP – o Stack Pointer). Note que a pilha mantém o estado da rotina hexa_display quando esta invoca outra. O SSP 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  do PEPE.

3.4 – Rotina para manipular o ecrã

Faça uma rotina que permita ligar/desligar o pixel na posição (linha, coluna) do ecrã (tem de determinar o byte, e dentro dele qual o bit, onde esse pixel está, a partir da linha e da coluna, em que ambos podem variar entre 0 e 31). Sugestões:

- O endereço do byte (dentro do ecrã, 0 a 128 bytes) onde o pixel está é dado por $\text{linha} * 4 + \text{coluna} / 8$, divisão inteira, considerando que o pixel (0, 0) está no canto superior esquerdo;
- O peso do bit (correspondente ao pixel) dentro desse byte pode ser obtido pelo resto da divisão inteira $\text{coluna} / 8$;
- Para afetar apenas o bit N (0 a 7) de um dado byte, use uma tabela de 8 máscaras (cada uma delas só com um bit a 1) e use o número N para indexar essa tabela, obtendo a máscara com o bit a 1 na posição N. Uma vez que em cada byte o pixel mais à esquerda tem o peso mais alto (7) mas a coordenada de coluna mais baixa, a ordem das máscaras na tabela deve estar invertida (80H deve ser a primeira).

NOTAS:

- A unidade mínima de escrita do ecrã é de um byte. Para alterar apenas um pixel (um bit) deve ler um byte do ecrã, alterá-lo (pondo a 1 o bit que for para ligar e a 0 se for para desligar) e voltar a escrevê-lo no mesmo endereço;
- Recomendam-se os seguintes parâmetros para esta rotina: linha, coluna e um valor 0 ou, consoante se quer ligar ou desligar o pixel.

4 – Objetivos a cumprir para o Projeto

Na semana seguinte à deste guião, e em termos do projeto, devem ser mostradas ao docente as seguintes rotinas, a funcionar (em boa parte, aproveitando o trabalho do guião de laboratório 4):

- Uma rotina que escreva um dado valor nos displays (um byte em hexadecimal), valor esse passado como parâmetro num registo;
- Uma rotina que faz um varrimento completo ao teclado e retorne o valor da tecla carregada (0 a FH) ou outro valor especial (FFH, por exemplo) se nenhuma tecla tiver sido carregada. Sugestão: faça outra rotina que testa uma dada linha do teclado, e outra rotina que invoca essa para cada uma das linhas. Se detetar algo, terá depois de converter a informação da linha e da coluna respetiva num valor de 0 a FH. Se varrer tudo e não detetar nada, retorne o tal valor especial;
- Uma rotina que invoca em ciclo a rotina de varrimento do teclado e, caso uma dada tecla seja carregada (e só esta), incrementa um valor mostrado no display, usando a primeira rotina. Esta rotina deve ser chamada em ciclo infinito pelo programa principal;
- Um programa que desenha no ecrã um padrão constituído por pixels alternadamente a 0 e 1 (com dois ciclos, um dentro do outro, que iteram pelas colunas e linhas todas, usando a rotina da secção 3.4 para cada pixel):

