Procedimentos em ASSEMBLY

I. Introdução e Objectivos

Um ciclo é a repetição da mesma sequência de acções até que determinada condição se verifique. Por vezes necessitamos de repetir a mesma sequência de acções não no mesmo ponto, mas sim em diferentes pontos de um programa. Para evitar escrever a mesma sequência de instruções de cada vez que necessitamos dela podemos escrevê-la como se fosse um "sub-programa" separado do programa principal e, nesse caso, sempre que precisarmos da sequência de instruções nele contidas, invocamo-lo. Estes "sub-programas" designam-se habitualmente por procedimentos. O objectivo desta ficha consiste em ficar a perceber como se implementam programas com procedimentos usando a linguagem Assembly do 8086.

II. Escrever e invocar procedimentos

Para delimitar o conjunto de instruções que constituem um procedimento utilizam-se as directivas **PROC** e **ENDP**. Se, em conjunto com cada uma destas directivas usarmos um nome (*label*), podemos invocar o procedimento usando esse *label*. A directiva **PROC** indica o inicio do procedimento e a directiva **ENDP** o fim.

Suponhamos que pretendíamos implementar um procedimento para calcular a média entre dois bytes. As fronteiras do procedimento poderiam ser estabelecidas da seguinte forma:

MEDIA **PROC**
<conjunto de instruções que constituem o procedimento MEDIA>
MEDIA **ENDP**

Sempre que precisamos de executar a sequência de instruções contida num procedimento usamos a instrução **CALL**. É esta instrução que indica ao processador qual o endereço da primeira instrução do procedimento. A instrução **RET** deve aparecer (normalmente no final do procedimento) quando se pretende indicar ao processador que deve voltar ao programa "chamante" e executar a instrução que está imediatamente a seguir á instrução **CALL**.

↳ **CALL**
A instrução CALL do 8086 tem o seguinte formato:
CALL < destino >

Quando executada, esta instrução, executa duas operações. Primeiro armazena o endereço da instrução que está imediatamente a seguir à instrução **CALL**. Este endereço é chamado o *endereço de retorno* porque é o endereço para o qual a execução volta depois do procedimento ser executado. Se a instrução **CALL** invocar um procedimento dentro do mesmo segmento de código então diz-se que estamos perante um call do tipo *near* ou *call intra-segmento*. Nesse caso só é necessário armazenar o conteúdo do registo IP, uma vez que o valor do registo CS se mantém inalterado. Se a instrução **CALL** invocar um procedimento noutro segmento de código, *call extra-segmento*, então diz-se que estamos perante um call do tipo *far* e é necessário armazenar o conteúdo do registo CS e o do registo

IP. A segunda operação da instrução CALL consiste em alterar o conteúdo do registo IP e, por vezes, também o do registo CS em função do < destino >. O novo par CS e IP deverá apontar para a primeira instrução do procedimento.

↳ **RET**

A instrução RET do 8086 não tem operandos.

Como já referido, quando é executada a instrução CALL, o endereço de retorno é armazenado. A instrução RET, quando executada, vai buscar esse valor e carrega-o de novo no registo IP.

III. A *stack* do 8086

Na secção anterior vimos que a instrução **CALL** necessitava de um local para armazenar o endereço de retorno, para que no fim da execução do procedimento a instrução RET o pudesse ir buscar. Esse local é uma zona de memória designada por *stack*. Além de servir para armazenar o endereço de retorno a *stack* tem também outras funções, nomeadamente:

- Armazenar o conteúdo de registos que queremos preservar
- Passar parâmetros para/de procedimentos

O 8086 trata a *stack* como um segmento de memória, ou seja, no máximo pode ter 64 KB.

Existem dois registos específicos para trabalhar com a *stack*: o registo SS (*Stack Segment*) que à semelhança dos registos DS e CS deverá conter os 16 bits mais significativos do endereço de inicio da *stack* e o SP (*Stack Pointer*) que contém o deslocamento em relação ao SS da última palavra escrita na *stack*. A posição ocupada pela última palavra escrita na *stack*, designa-se habitualmente por topo da *stack* e é para esta posição que aponta o registo SP.

O SP é automaticamente decrementado de dois valores antes de alguma palavra ser escrita na *stack*, por isso este registo deve ser iniciado com o tamanho da *stack* e não com zero.

a) Guardar o endereço de retorno

Já foi dito que o endereço de retorno era armazenado na *stack* pela instrução **CALL** e de lá retirado pela instrução **RET**.

Quando a instrução CALL é executada para efectuar um *call* do tipo *near* o SP é automaticamente decrementado de dois valores e o conteúdo do registo IP é copiado para a nova posição apontada por SP. Quando o processador encontra a instrução RET no fim do procedimento, copia o conteúdo da posição de memória apontada por SP para o registo IP e incrementa o SP de dois valores.

Por exemplo, assumindo que o valor do SS é 7000H e o valor de SP 0050H, então o endereço físico do topo da *stack* é 70050H. Depois de um *call* do tipo *near* o SP ficaria com o valor 004EH e o endereço físico do topo da *stack* passaria a ser 7004EH. Quando a instrução RET for executada, o valor armazenado na *stack* (a palavra apontada por SP) voltava a ser colocado no registo IP e o registo SP depois de incrementado voltava ao valor 0050H.

b) Gravar o conteúdo dos registos

É por vezes necessário usar registos dentro do procedimento que também estão a ser usados cá fora no programa “chamante”. Para que o uso de registos dentro dos procedimentos não interfira com a sua utilização cá fora pode-se usar a *stack* para guardar os seus valores originais. A ideia consiste em escrever na *stack* os valores dos registos que são usados no procedimento, antes de este começar a usá-los e consequentemente a alterá-los, e no fim do procedimento voltar a repor tudo como estava (**salvaguarda do contexto**).

Para isso usam-se as instruções PUSH e POP.

↳ **PUSH**

A instrução PUSH do 8086 tem o seguinte formato:
PUSH <fonte>

Quando executada, esta instrução, realiza duas operações. Primeiro decrementa o SP de dois valores e depois copia para a posição da *stack* apontada por SP o conteúdo de <fonte >. A fonte pode ser um registo de 16 bits ou uma palavra em memória.

↳ **POP**

A instrução POP do 8086 tem o seguinte formato:
POP <destino>

Quando executada, esta instrução, realiza duas operações. Primeiro copia para o conteúdo de <destino> a palavra em *stack* apontada pelo registo SP e depois incrementa SP de dois valores. O destino pode ser um registo de 16 bits ou uma palavra em memória.

Para repor o valor dos registos salvaguardados, uma vez que a *stack* funciona utilizando a disciplina de serviço *last-in-first-out*, os POPs têm que ser efectuados pela ordem inversa pela qual foram efectuados os PUSHs.

IV. Passagem de argumentos de/para procedimentos

Muitos procedimentos actuam sobre dados ou endereços e devolvem resultados ao programa "chamante". Aos dados e endereços que passamos para/de procedimentos chamam-se parâmetros.
Existem várias formas possíveis de passar parâmetros para e de procedimentos. São elas:

- Através dos registos;
- Através de posições de memória;
- Através de apontadores para posições de memória contidos em registos;
- Através da própria *stack.*

a) Passagem de parâmetros em registos

Neste caso é necessário escolher quais os registos que vão ser usados para transportar os parâmetros para o procedimento e quais os que vão ser usados para devolver os resultados ao programa "chamante".

Tomando como exemplo o procedimento que calcula a média entre dois bytes, vamos necessitar de dois registos de 8 bits (por exemplo o AL e o AH) para passar os bytes para o procedimento e um registo de 8 bits (por exemplo o AL) para devolver o resultado (média) ao programa "chamante".

✍ *Implemente um procedimento que calcula a média entre dois bytes e escreva um programa que o invoca. Utilize registos para passagem de parâmetros e para retornar o resultado.*

1) Passagem de parâmetros em posições de memória

É também possível que um procedimento aceda a variáveis em memória através do seu próprio nome. Embora se use por vezes este método, não é o mais aconselhável uma vez que torna os procedimentos pouco flexíveis e versáteis.

✍ *Altere o programa anterior para que o procedimento vá buscar os dados directamente à memória e guarde também directamente em memória o resultado.*

2) Passagem de parâmetros através de apontadores contidos em registos

Para ultrapassar as limitações do método anterior usa-se muitas vezes a passagem de parâmetros através de apontadores para posições de memória, apontadores esses contidos em registos. Para isso carregam-se os registos com os endereços das posições de memória

que contêm as variáveis que queremos manipular dentro dos procedimentos, por exemplo usando a instrução **LEA.** Este método é mais versátil do que o anterior pois permite que se passem para o procedimento apontadores para quaisquer posições de memória.

✍ *Altere o programa anterior de forma a que a passagem de parâmetros seja feita através de apontadores contidos em registos.*

3) Passagem de parâmetros utilizando a *stack*

Para usar este método é necessário que algures no programa principal, antes de se invocar o procedimento, se utilize a instrução PUSH para escrever na *stack* os vários parâmetros. Dentro do procedimento as instruções vão ler os argumentos à *stack* de acordo com as necessidades. O mesmo se passa para retornar os resultados ao programa "chamante", mas desta vez ao contrário. Neste caso são as instruções do procedimento que escrevem os resultados na *stack* e o programa principal que os vai lá buscar.

Para usar este método é necessário fazer um mapa da *stack*, para que facilmente se saiba onde é que estão os vários parâmetros e também para onde é que está a apontar, em cada instante, o registo SP.

✍ *Altere o programa anterior de forma a que a passagem de parâmetros seja feita utilizando a stack*