Por: Frederico Pissarra

ASSEMBLY XII

A partir de agora veremos, resumidamente, como desenvolver funções/procedures em assembly no mesmo código PASCAL.

O TURBO PASCAL (a partir da versao 6.0) fornece algumas palavras-chave dedicadas à construção de rotinas assembly in-line (esse recurso é chamado de BASM nos manuais do TURBO PASCAL - BASM é a abreviação de Borland ASseMbler).

Antes de começarmos a ver o nosso primeiro código em assembly vale a pena ressaltar alguns cuidados em relação a codificação de rotinas assembly em TURBO PASCAL... As nossas rotinas devem:

- Preservar sempre o conteúdo dos registradores DS, BP e SP.
- Nunca modificar, diretamente, o conteúdo dos registradores CS, IP e SS.

O motivo dessas restrições é que os registradores BP, SP e SS são usados na obtenção dos valores passados como parametros à função/procedure e na localização das variaveis globais na memória. O registrador DS é usado por todo o código PASCAL e aponta sempre para o segmento de dados corrente (o qual não sabemos onde se encontra... deixe que o código PASCAL tome conta disso!).

Com relação ao conteúdo de CS e IP, não é uma boa prática (nem mesmo em códigos assembly puros) alterar o seus valores. Deixe que as instruções de salto e chamada de subrotinas façam isso por você!).

Os demais registradores podem ser alterados a vontade.

A função HexByte() abaixo é um exemplo de função totalmente escrita em assembly... Ela toma um valor de 8 bits e devolve uma string de 2 bytes contendo o valor hexadecimal desse parametro:

```
FUNCTION    HexByte(Data : Byte) : String; ASSEMBLER;
ASM
    LES     DI,@Result   { Aponta para o inicio da string. }

    MOV     AL,2         { Ajusta tamanho da string em 2.  }
    STOSB

    MOV     AL,Data      { Pega o dado a ser convertido.   }

    MOV     BL,AL        { Salva-o em BL.                  }
    SHR     AL,1         { Para manter compatibilidade com }
    SHR     AL,1         { os microprocessadores 8088/8086 }
    SHR     AL,1         { nao é prudente usar SHR AL,4.   }
    SHR     AL,1
    ADD     AL,'0'       { Soma com ASCII '0'.             }
    CMP     AL,'9'       { Maior que ASCII '9'?            }
    JBE     @NoAdd_1     { ... Nao é, então nao soma 7.    }
```

```
    ADD     AL,7          { ... É, então soma 7.            }
  @NoAdd_1:
    MOV     AH,AL         { Salva AL em AH.                 }

    MOV     AL,BL         { Pega o valor antigo de AL em BL.}
    AND     AL,1111B      { Zera os 4 bits superiores de AL.}
    ADD     AL,'0'        { Soma com ASCII '0'.             }
    CMP     AL,'9'        { Maior que ASCII '9'?            }
    JBE     @NoAdd_2      { ... Nao é, então nao soma 7.    }
    ADD     AL,7          { ... É, então soma 7.            }
  @NoAdd_2:

    XCHG    AH,AL         { Trocar AH com AL para gravar na }
    STOSW                 { ordem correta.                  }
  END;
```

A primeira linha é a declaração da função seguida da diretiva ASSEMBLER (informando que a função TODA foi escrita em assembly!). A seguir a palavra-chave ASM indica o inicio do bloco assembly até que END; marque o fim da função...

A primeira linha do código assembly é:

```
    LES     DI,@Result
```

Quando retornamos uma string numa função precisamos conhecer o endereço do inicio dessa string. A variável @Result contém um pointer que aponta para o inicio da string que será devolvida numa função. Esse endereço é sempre um endereço FAR (ou seja, no formato SEGMENTO:OFFSET).

A seguir inicializamos o tamanho da string em 2 caracteres:

```
    MOV     AL,2
    STOSB
```

Note que STOSB vai gravar o conteúdo de AL no endereço apontado por ES:DI, ou seja, o endereço apontado por @Result, e logo após DI é incrementado, apontando para a primeira posição valida da string.

O método que usei para gerar uma string hexadecimal é o seguinte:

- Pegamos o parametro 'Data' e colocamos em AL.
- Salva-se o conteúdo de AL em BL para que possamos obter os 4 bits menos significativos sem termos que ler 'Data' novamente!
- Com AL fazemos:
  - Desloca-se AL 4 posiçoes para a direita, colocando os 4 bits mais significativos nos 4 menos significativos e preenchendo os 4 mais significativos com 0B.
  (a)- Soma-se o valor do ASCII '0' a AL.
  (b)- Verifica-se se o resultado é maior que o ASCII '9'.
    - Se for, somamos 7.
  - Salvamos o conteúdo de AL em AH.
- Recuperamos o valor antigo de AL que estava em BL.
- Com AL fazemos:
  - Zeramos os 4 bits mais significativos para obtermos apenas os 4 menos significativos em AL.
  - Repetimos (a) e (b)

- Trocamos AL com AH e gravamos AX com STOSB

A primeira pergunta é: Porque somar 7 quando o resultado da soma com o ASCII '0' for maior que o ASCII '9'? A resposta pode ser vista no pedaço da tabela ASCII abaixo:

```
 +-------------------------------------------------------------+
 ¦         0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F       ¦
 ¦                           +-----------+                     ¦
 ¦                         E esses 7 bytes ?                   ¦
 +-------------------------------------------------------------+
```

Observe que depois do ASCII '9' segue o ASCII ':' ao invés do ASCII 'A', como é desejado... Entao, se o resultado da soma dos 4 bits menos signficativos (que varia de 0000B até 1111B - ou de 0 a 15) com o ASCII '0' for maior que o ASCII '9' precisamos compensar a existencia dos 7 caracteres indesejáveis!

Imagine que AL seja 0. Somando o ASCII '0' (que equivale ao número 30h) a AL obteriamos:

```
 +-------------------------------------------------------------+
 ¦  AL = 0010B = 2h                                            ¦
 ¦  AL = 2h + '0'                                              ¦
 ¦  AL = 2h + 30h                                              ¦
 ¦  AL = 32h = '2'                                             ¦
 +-------------------------------------------------------------+
```

Imagine agora que AL seja 1011B. Fazendo as mesmas contas obteriamos AL = 3Bh (que é a mesma coisa que o ASCII ';'. No entando, 3Bh é maior que o ASCII '9' (ou seja, 39h)... Entäo:

```
 +-------------------------------------------------------------+
 ¦  AL = ';' = 3Bh                                             ¦
 ¦  AL = 3Bh + 7h                                              ¦
 ¦  AL = 42h = 'B'                                             ¦
 +-------------------------------------------------------------+
```

A outra coisa que você poderia me perguntar é o porque eu usei a instrução XCHG AH,AL no final do código. A resposta é simples... Os microprocessadores da INTEL gravam words na memória da seguinte maneira:

```
 +-------------------------------------------------------------+
 ¦  Word = FAFBh                                               ¦
 ¦  Na memória: FBh FAh                                        ¦
 +-------------------------------------------------------------+
```

Näo importa se o seu computador seja um Pentium ou um XT... A memória é sempre dividida em BYTES. A CPU apenas "le" um conjunto maior de bytes de acordo com a quantidade de bits da sua CPU. Por exemplo, os microprocessadores 8086 e 80286 säo CPUs de 16 bits e por isso conseguem ler 2 bytes (8 bits + 8 bits = 16 bits) de uma só vez... As CPUs 386 e 486 säo de 32 bits e podem ler de uma só vez 4 bytes!

Esse conjunto de bytes que a CPU pode enxergar é sempre armazenado da forma contrária do que os olhos humanos leem... O byte menos significativo SEMPRE vem ANTES do mais significativo. No caso de um DOUBLEWORD (ou numero de 32 bits de tamanho) o formato é o mesmo... Exemplo:

```
 +-------------------------------------------------------------+
 ¦  Número = FAFBFCFDFEh                                       ¦
```

```
 ¦  Na memória: FE FD FB FA                                          ¦
 +-------------------------------------------------------------------+
```

Analizando a rotina HexByte() a gente ve que AH tem o byte mais significativo e AL o menos significativo. Como o menos significativo vem sempre antes do mais significativo fiz a troca de AH com AL para que o número HEXA seja armazenado de forma correta na memória (string). Um exemplo: Suponha que o você passe o valor 236 à função HexByte():

```
 +-------------------------------------------------------------------+
 ¦  Valor = 236 ou ECh                                               ¦
 ¦  Até antes de XCHG AH,AL:    AH = ASCII 'E'                       ¦
 ¦                              AL = ASCII 'C'                       ¦
 +-------------------------------------------------------------------+
```

Se não tivessemos a instrução XCHG AH,AL e simplesmente usassemos o STOSW (como está no código!) AH seria precedido de AL na memória (ou na string!), ficariamos com uma string 'CE'! Não me lembro se já falei que o L de AL significa LOW (ou menos significativo!) e H de AH significa HIGH (ou mais significativo), portanto AL e AH são, respectivamente, os bytes menos e mais significativos de AX!

Não se importe em coloca um RET ao fim da função, o TURBO PASCAL coloca isso sozinho...

Você deve estar se perguntando porque não fiz a rotina de forma tal que a troca de AH por AL não fosse necessária... Well... Fiz isso pra ilustrar a forma como os dados são gravados na memória! Retire XCHG AH,AL do código e veja o que acontece! Um outro bom exercício é tentar otimizar a rotina para que a troca não seja necessária...

E... para fechar a rotina, podemos aproveitar HexByte() para construir HexWord():

```
 +-------------------------------------------------------------------+
 ¦  Function HexWord(Data : Word) : String;                          ¦
 ¦  Var H, L : String;                                               ¦
 ¦  Begin                                                            ¦
 ¦      H := HexByte(HIGH(Data));                                    ¦
 ¦      L := HexByte(LOW(Data));                                     ¦
 ¦      HexWord := H + L;                                            ¦
 ¦  End;                                                             ¦
 +-------------------------------------------------------------------+
```

HexDoubleWord() eu deixo por sua conta :)

Aguardo as suas duvidas...