```
ì-----------------------------------------------©
¦ RBT   ¦   Curso de Assembly   ¦   Aula Nº 13 ¦
È-----------------------------------------------¥
```

Por: Frederico Pissarra

```
+---------------+
¦ ASSEMBLY XIII ¦
+---------------+
```

Algumas pessoas, depois de verem o código-exemplo do texto anterior, desenvolvido para ser compilado em TURBO PASCAL, me perguntaram: "E quanto ao C?!". Well... aqui vão algumas técnicas para codificação mixta em C...

Antes de começarmos a dar uma olhada nas técnicas, quero avisar que meu compilador preferido é o BORLAND C++ 3.1. Ele tem algumas características que não estão presentes do MicroSoft C++ 7.0 ou no MS Visual C++! Por exemplo, O MSC++ ou o MS-Visual C++ não tem "pseudo"-registradores (que ajudam um bocado na mixagem de código, evitando os "avisos" do compilador).

Mesmo com algumas diferenças, você poderá usar as técnicas aqui descritas... As regras podem ser usadas para qualquer compilador que não gere aplicações em modo protegido para o MS-DOS.

¦ Regras para a boa codigificação assembly em C

Assim como no TURBO PASCAL, devemos:

* Nunca alterar CS, DS, SS, SP, BP e IP.
* Podemos alterar com muito cuidado ES, SI e DI
* Podemos alterar sempre que quisermos AX, BX, CX, DX

O registrador DS sempre aponta para o segmento de dados do programa... Se a sua função assembly acessa alguma variável global, e você tiver alterado DS, a variável que você pretendia acessar não estará disponível! Os registradores SS, SP e BP são usados pela linguagem para empilhar e desempilhar os parametros e variáveis locais da função na pilha... altera-los pode causar problemas! O par de registradores CS:IP nao deve ser alterado porque indica a próxima posição da memória que contém uma instrução assembly que será executada... Em qualquer programa "normal" esses últimos dois registradores são deixados em paz.

No caso dos registradores ES, SI e DI, o compilador os usa na manipulação de pointers e quando precisa manter uma variável num registrador (quando se usa a palavra-reservada "register" na declaração de uma variável, por exemplo!). Dentro de uma função escrita puramente em assembly, SI e DI podem ser alterados a vontade porque o compilador trata de salva-las na pilha (via PUSH SI e PUSH DI) e, ao término da função, as restaura (via POP DI e POP SI). A melhor forma de se saber se podemos ou não usar um desses registradores em um código mixto é compilando o programa e gerando uma listagem assembly (no BORLAND C++ isso é feito usando-se a chave -S na linha de comando!)... faça a análize da função e veja se o uso desses registradores vai prejudicar alguma outra parte do código!

Se você não quer ter essa dor de cabeça, simplesmente salve-os antes de usar e restaure-os depois que os usou!

¦ Modelamento de memória:

O mais chato dos compiladores C/C++ para o MS-DOS é o modelamento de memória, coisa que näo existe no TURBO PASCAL! Digo "chato" porque esse recurso, QUE É MUITO UTIL, nos dá algumas dores de cabeça de vez em quando...

Os modelos COMPACT, LARGE e HUGE usam, por default, pointers do tipo FAR (segmento:offset). Os modelos TINY, SMALL e MEDIUM usam, por default, pointers do tipo NEAR (apenas offset, o segmento de dados é assumido!).

A "chatisse" está em criarmos códigos que compilem bem em qualquer modelo de memória. Felizmente isso é possível graças ao pre-processador:

```
#if defined(__TYNY__) || defined(__SMALL__) || defined(__MEDIUM__)
/* Processamento de pointers NEAR */
#else
/* Processamento dos mesmos pointers... mas, FAR! */
#endif
```

Concorda comigo que é meio chato ficar enchendo a listagem de diretivas do pré-processador?... C'est la vie!

¦ C + ASM

Os compiladores da BORLAND possuem a palavra reservada "asm". Ela diz ao compilador que o que a segue deve ser interpretado como uma instrução assembly. Os compiladores da MicroSoft possuem o "_asm" ou o "__asm". A BORLAND ainda tem uma diretiva para o pré-processador que é usada para indicar ao compilador que o código deve ser montado pelo TURBO ASSEMBLER ao invés do compilador C/C++:

```
¦ #pragma inline
```

Você pode usar isto ou então a chave -B da linha de comando do BCC... funciona da mesma forma! Você deve estar se perguntando porque usar o TURBO ASSEMBLER se o próprio compilador C/C++ pode compilar o código... Ahhhhh, por motivos de COMPATIBILIDADE! Se você pretende que o seu código seja compilável no TURBO C 2.0, por exemplo, deve incluir a diretiva acima!! Além do mais, o TASM faz uma checagem mais detalhada do código assembly do que o BCC...

Eis um exemplo de uma funçäozinha escrita em assembly:

```
int f(int X)
{
    asm mov     ax,X    /* AX = parametro X */
    asm add     ax,ax   /* AX = 2 * AX */
    return _AX;         /* retorna AX */
}
```

Aqui segue mais uma regra:

¦ Se a sua função pretende devolver um valor do tipo "char" ou

"unsigned char", coloque o valor no registrador AL e (nos compiladores da BORLAND) use "return _AL;"

¦ Se a sua função pretende devolver um valor do tipo "int" ou "unsigned int", coloque o valor no registrador AX e (também nos compiladores da BORLAND) use "return _AX;"

A última linha da função acima ("return _AX;") não é necessária, mas se não a colocarmos teremos um aviso do compilador, indicando que "a função precisa retornar um 'int'". Se você omitir a última linha (é o caso dos compiladores da MicroSoft que não tem pseudo-registradores) e não ligar pros avisos, a coisa funciona do mesmo jeito.

Agora você deve estar querendo saber como devolver os tipos "long", "double", "float", etc... O tipo "long" (bem como "unsigned long") é simples:

¦ Se a sua função pretende devolver um valor do tipo "long" ou "unsigned long", coloque os 16 bits mais significativos em DX e os 16 menos significativos em AX.

Não existe uma forma de devolvermos DX e AX ao mesmo tempo usando os pseudo-registradores da Borland, então prepare-se para um "aviso" do compilador...

Os demais tipos não são inteiros... são de ponto-flutuante, portanto, deixe que o compilador tome conta deles.

¦ Trabalhando com pointers e vetores:

Dê uma olhada na listagem abaixo:

```
¦   unsigned ArraySize(char *str)                                          ¦
¦   {                                                                      ¦
¦#if defined(__TYNY__) || defined(__SMALL__) || defined(__MEDIUM__)        ¦
¦       asm mov     si,str  /* STR = OFFSET apenas */                     ¦
¦#else                                                                     ¦
¦       asm push    ds                                                     ¦
¦       asm lds     si,str  /* STR = SEGMENTO:OFFSET */                   ¦
¦#endif                                                                    ¦
¦                                                                          ¦
¦       asm mov     cx,-1                                                  ¦
¦ContinuaProcurando:                                                      ¦
¦       asm inc     cx                                                     ¦
¦       asm lodsb                                                          ¦
¦       asm or      al,al                                                  ¦
¦       asm jnz     ContinuaProcurando                                     ¦
¦       asm mov     ax,cx                                                  ¦
¦                                                                          ¦
¦#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)       ¦
¦       asm pop     ds          /* Restaura DS */                          ¦
¦#endif                                                                    ¦
¦                                                                          ¦
¦       return _AX;                                                        ¦
¦   }                                                                      ¦
```

A rotina acima é equivalente a função strlen() de <string.h>.

Como disse antes, nos modelos COMPACT, LARGE e HUGE um pointer tem o formato SEGMENTO:OFFSET que é armazenado na memória em uma

grande variável de 32 bits (os 16 mais significativos säo o SEGMENTO e os 16 menos significativos säo o OFFSET). Nos modelos TINY, SMALL e MEDIUM apenas o OFFSET é fornecido no pointer (ele tem 16 bits neste caso), o SEGMENTO é o assumido em DS (não devemos alterá-lo, neste caso!).

Se você compilar essa listagem nos modelos COMPACT, LARGE ou HUGE o código coloca em DS:SI o pointer (lembre-se: pointer é só um outro nome para "endereço de memória!"). Senäo, precisamos apenas colocar em SI o OFFSET (DS já está certo!).

Ao sair da função, DS deve ser o mesmo de antes da função ser chamada... Portanto, nos modelos "LARGOS" (hehe) precisamos salvar DS ANTES de usá-lo e restaura-lo DEPOIS de usado! O compilador não faz isso automaticamente!

Não se preocupe com SI (neste caso!)... este sim, o compilador salva sozinho...

Um macete com o uso de vetores pode ser mostrado no seguinte código exemplo:

```
char a[3];
int b[3], c[3];
long d[3];

void init(void)
{
    int i;

    for (i = 0; i < 3; i++)
        a[i] = b[i] = c[i] = d[i] = 0;
}
```

O compilador gera a seguinte função equivalente em assembly:

```
void init(void)
{
    asm xor     si,si           /* SI = i */
    asm jmp     short @1@98
@1@50:
    asm mov     bx,si           /* BX = i */
    asm shl     bx,1
    asm shl     bx,1            /* BX = BX * 4 */
    asm xor     ax,ax
    asm mov     word ptr [d+bx+2],0 /* ?! */
    asm mov     word ptr [d+bx],ax

    asm mov     bx,si
    asm shl     bx,1
    asm mov     [c+bx],ax

    asm mov     bx,si       /* ?! */
    asm shl     bx,1        /* ?! */
    asm mov     [b+bx],ax

    asm mov     [a+si],al
    asm inc     si
@1@98:
    asm cmp     si,3
    asm jl      short @1@50
```

```
}
```

Quando poderiamos ter:

```
void init(void)
{
    asm xor     si,si           /* SI = i = 0 */
    asm jmp     short @1@98
@1@50:
    asm mov     bx,si           /* BX = i */
    asm shl     bx,1
    asm shl     bx,1            /* BX = BX * 4 */
    asm xor     ax,ax           /* AX = 0 */
    asm mov     word ptr [d+bx+2],ax /* modificado! */
    asm mov     word ptr [d+bx],ax

    asm shr     bx,1            /* BX = BX / 2 */
    asm mov     [c+bx],ax
    asm mov     [b+bx],ax

    asm mov     [a+si],al
    asm inc     si
@1@98:
    asm cmp     si,3
    asm jl      short @1@50
}
```

Note que economizamos 3 instruções em assembly e ainda aceleramos um tiquinho, retirando o movimento de um valor imediato para memória (o 0 de "mov word ptr [d+bx+2],0"), colocando em seu lugar o registrador AX, que foi zerado previamente.

Isso parece besteira neste código, e eu concordo... mas, e se tivessemos:

```
void init(void)
{
    for (i = 0; i < 32000; i++)
        a[i] = b[i] = c[i] = d[i] =
        e[i] = f[i] = g[i] = h[i] =
        I[i] = j[i] = k[i] = l[i] =
        m[i] = n[i] = o[i] = p[i] =
        r[i] = s[i] = t[i] = u[i] =
        v[i] = x[i] = y[i] = z[i] =
        /* ... mais um monte de membros de vetores... */
        = _XYZ[i] = 0;
}
```

A perda de eficiência e o ganho de tamanho do código seriam enormes por causa da quantidade de vezes que o loop é exeutado (32000) e por causa do numero de movimentos de valores imediatos para memória, "SHL"s e "MOV BX,SI" que teriamos! Conclusão: Em alguns casos é mais conveniente manipular VARIOS vetores com funções escritas em assembly...

EXEMPLO de codificação: ** O swap() aditivado :)

Alguns códigos em C que precisam trocar o conteúdo de uma variável pelo de outra usam o seguinte macro:

```
#define swap(a,b) { int t; t = a; a = b; b = t; }
```

Bem... a macro acima funciona perfeitamente bem, mas vamos dar uma olhada no código assembly gerado pelo compilador pro seguinte programinha usando o macro swap():

```
#define swap(a,b) { int t; t = a; a = b; b = t; }

int x = 1, y = 2;

void main(void)
{ swap(x,y); }
```

O código equivalente, após ser pre-processado, ficaria:

```
int x = 2, y = 1;
void main(void) {
    int t;

    asm mov ax,x
    asm mov t,ax
    asm mov ax,y
    asm mov x,ax
    asm mov ax,t
    asm mov y,ax
}
```

No máximo, o compilador usa o registrador SI ou DI como variavel 't'... Poderiamos fazer:

```
int x = 2, y = 1;
void main(void)
{
    asm mov     ax,x
    asm mov     bx,y
    asm xchg    ax,bx
    asm mov     x,ax
    asm mov     y,ax
}
```

Repare que eliminamos uma instrução em assembly, eliminando também um acesso à memória e uma variável local... Tá bom... pode me chamar de chato, mas eu ADORO diminuir o tamanho e aumentar a velocidade de meus programas usando esse tipo de artifício! :)