

Um (não muito curto...) auxiliar de memória para programar em
assembler

ou

Como fazer programas em assembler com um mínimo de sustos

ou ainda

os 10% de instruções do 80C188 que permitem fazer 90% dos
programas

Pedro Fonseca*

Outubro de 1998

1 Introdução (vamos por as coisas claras)

Alguns pontos sobre este texto e a programação em assembler:

1. Peço desculpa por começar com uma negativa, mas:

Antes de mais nada: este texto *não é* um manual de linguagem assembly para os processadores Intel (nem eu sou capaz de fazer tal coisa...) (Se alguém for, força!...) O que se pretende aqui é apresentar um conjunto de instruções tão pequeno quanto possível (e por isso, tão fácil de compreender quanto possível) e um conjunto de noções sobre o processador 80C188 (e por extensão, de todos os seus primos) que permitam fazer um leque de programas (outra vez) tão vasto quanto possível.

2. A linguagem assembler assusta da primeira vez que a encontramos, mas depois, vamo-nos habituando e ficamos a gostar cada vez mais dela. É assim uma linguagem feia (horrível!, o C ou o Pascal são coisas muito mais bonitas e elegantes), mas extremamente simpática e disponível. Deixa-nos fazer tudo o que quisermos, desde que ela (a linguagem) consiga. (Ao contrário do C e — muito menos! — do Pascal). Tão disponível que por vezes deixa-nos fazer coisas que são autênticos suicídios do microprocessador (o que é uma disponibilidade que nos obriga a tomar alguns cuidados...)

3. Para quê andar a mexer em assembler? Não é verdade que “toda a gente” programa em C ou C++? (Eu sempre gostei destas frases com o “toda a gente”...)

Talvez pouca gente trabalhe em assembler. (É verdade...) Mas muita gente usa o assembler (talvez mesmo sem o saber). E dentre aqueles que trabalham com programas que têm de ser rápidos, poucos são aqueles que, depois de fazerem um programa em C, não tiveram que, uma vez ou outra, ir olhar para o código gerado e tentar perceber porque é que uma coisa tão simples como copiar um array de um sítio para o outro demora às vezes uma eternidade (então se aparecer no programa qualquer coisa como

*Com a preciosa ajuda de José Luís Azevedo (na técnica) e Fernando Santos (na forma)

Nome	Tamanho		Nome	Tamanho
AX	16 bits	=	AH	8 bits
DX	16 bits		AL	8 bits
CX	16 bits		DH	8 bits
BX	16 bits		DL	8 bits
BP	16 bits		CH	8 bits
SI	16 bits		CL	8 bits
DI	16 bits		BH	8 bits
SP	16 bits		BL	8 bits
			BP	16 bits
			SI	16 bits
			DI	16 bits
			SP	16 bits

Figura 1: Variáveis principais do 80C188

Flag	Bit	Nome	Descrição
CF	0	Carry	“Carry” do último resultado
PF	2	Parity	Resultado tem nº par de bits
AF	4	Aux. Carry	“Carry” do bit 3 para 4
ZF	6	Zero	Resultado é zero
SF	7	Sign	Resultado é negativo
OF	11	Overflow	Resultado “transbordou”
TF	8	Trace	Execução passo-a-passo
IF	9	Interrupt	Autoriza interrupções
DF	10	Direction	Sentido do incremento (+/-) em instruções de string

Tabela 1: Flags do 80188

`struct Enorme *catastrofico`, o resultado é garantido! Experimentem com um compilador de C que gere assembler, e logo vêem.)

Bom, vamos então ao que interessa!...

2 Como olhar para um programa em assembler

Com descontração e sem preconceitos!

Obviamente, que não é a mesma coisa que Pascal ou C, mas há-de lá chegar. Podemos pensar no programa em assembly como um programa numa outra linguagem, mas em que as variáveis já estão definidas à partida, e não temos direito a criar variáveis novas (por acaso, até temos, mas já lá iremos). Assim, tudo o que quisermos fazer tem que ser com essas variáveis (Figura 1).

Temos assim 8 variáveis de 16 bits, AX, BX, CX, DX, BP, SI, DI e SP. Para além disso, cada uma das quatro primeiras (as que terminam em X) podem ser vistas também como duas de 8 bits: AH e AL, BH e BL, CH e CL e DH e DL. (H vem de “High”, a parte alta do registo, e L de “Low”).

Para além destes, há o registo IP (Instruction Pointer, que indica a próxima instrução a ser executada) e o registo STATUS WORD, que contém as flags (Tabela 1). TF, IF e DF são as flags de comando (são instruções do utilizador ao processador). As restantes são flags de “status” (a sua função é serem lidas para verificar a ocorrência do acontecimento associado).

Há ainda mais quatro registos de 16 bits, que têm a ver com o acesso à memória, por isso, vamos ver primeiro como é que isso se faz.

3 Memória

Vamos começar por uma frase-chave:

A memória do 80188 está dividida em segmentos e offsets.

(Fica sempre bem dizer isto.)

O que é isso dos segmentos e offsets? Para já, é apenas uma convenção! A memória que está ligada ao 80188 é uma fileira de bytes ($2^{20} = 1048576 = 1\text{MB}$ no máximo), todos em fila indiana e cada um com o seu número

(o seu endereço físico ou endereço efectivo), que vai de 0 a $FFFFF_h = 1048576_d$. O processador olha para essa memória como estando dividida em *segmentos*. Cada segmento são 64K bytes consecutivos ($= 2^{16}$ bytes), e começa num endereço múltiplo de 16 ($16_d = 10_h$). O segmento 0 começa no endereço 00000_h , o segmento 1 no endereço 00010_h , o segmento 2 em 00020_h , ... A posição de um determinado byte dentro de um segmento é o seu offset. Cada endereço na memória é dado por um par $SSSS : AAAA$, em que $SSSS$ representa o segmento e $AAAA$ o offset. Isso permite aceder a memórias com mais de 64KBytes (1MB para o caso do 80188, o que requer 20 bits de endereço) com processadores que têm registos só de 16 bits.

Como traduzir endereços dados pelo segmento e offset num endereço físico (efectivo)? É simples: pega-se no valor do segmento, acrescenta-se um zero, e soma-se-lhe o offset. Dito por números:

$$[SSSS : AAAA] = \frac{SSSS0_h + AAAA_h}{RRRRR_h}$$

Por exemplo, para o caso do endereço 100:50 (segmento=100, offset=50) vem:

$$\frac{1000_h + 50_h}{1050_h}$$

É fácil de ver que os segmentos se sobrepõem e que uma posição na memória pertence a vários segmentos ao mesmo tempo. Do mesmo modo, um só endereço efectivo pode ter várias representações na forma segmento:offset. Por exemplo, 200:50, 203:20 e 1A0:650 representam a mesma posição de memória em três segmentos diferentes. (Qual é? Conseguem-se arranjar mais representações diferentes para essa posição?)

3.1 Registos para aceder à memória

Sempre que acede à memória, o 80188 utiliza dois tipos de registos: um para determinar o segmento, e outro para determinar o offset. Os registos de segmento são especializados e só devem ser usados para essa função (armazenar segmentos). Há quatro desses registos: CS (Code Segment), DS (Data Segment), SS (Stack Segment) e ES (Extra Segment). Para já, vamos esquecer o ES, que só é utilizado nalgumas instruções. Como se pode ver pela tabela 2,

Registo	Segmento	Uso	End.
CS	Código	Ler instruções da memória	CS:IP
SS	Stack	Uso da stack (PUSH e POP)	SS:SP
DS	Dados	Leitura e escrita de dados	

Tabela 2: Registos de segmento mais importantes

o endereço na memória da instrução que vai ser lida é dado por $CS : IP$ (Code Segment e Instruction Pointer) e o endereço do cimo da stack por $SS : SP$ (Stack Segment e Stack Pointer). Normalmente, tudo o que seja acesso de dados à memória usa o segmento definido pelo registo DS. As excepções são:

- as operações com as instruções de string que utilizam o registo DI como ponteiro (neste caso, o endereço efectivo associado é $ES : DI$);
- as operações que utilizam o registo BP como ponteiro (em que o segmento associado é o SS).

4 Instruções

ou

Que farei com estes registos?

Carregar registos A utilização mais simples para um registo de um processador é para lá guardar um valor. Carregar um registo com um valor é feito com a instrução MOV:

“à la C”	assembly	Descrição
AX = 10;	MOV AX, 10	AX ← 10
AX = 0x10;	MOV AX, 10h	AX ← 16
AX = 0xFF;	MOV AX, 0FFh	AX ← 255
BX = CX;	MOV BX, CX	BX ← CX
BL = CL;	MOV BL, CL	BL ← CL
CX = *BX;	MOV CX, [BX]	CX ← [BX] CL ← [BX] ou CH ← [BX + 1]
CL = *BX;	MOV CL, [BX]	CL ← [BX]
BX = &val;	MOV BX, offset val	BX ← Endereço de val

As primeiras instruções carregam o registo AX com um valor constante. Nas duas seguintes, um registo é carregado com o valor de um outro registo. Podem-se usar registos de 16 bits (AX, BX, ...) ou de 8 bits (AL, AH, ...). O que não se pode é misturar registos de comprimento diferente. Reparem que na instrução para carregar o valor FF_h num registo: a representação deste valor é 0FFh (começa por zero; senão seria interpretado como a variável com o nome FFH).

Na instrução MOV CX, [BX], o registo CX é carregado com o que estiver na posição de memória apontada por BX. (Os parênteses rectos [...] significam “o conteúdo da posição de memória apontada por ...”) Como CX é um registo de 16 bits, há dois bytes transferidos da memória para CX. O byte menos significativo é guardado na posição de memória mais baixa. Na instrução seguinte, MOV CL, [BX], apenas um byte é transferido da memória para o registo (aquele que efectivamente está no endereço apontado por BX).

Finalmente, na última instrução, o registo BX é carregado com a posição de memória em que se encontra guardada a variável val.

Contas Transferir valores de um lado para o outro pode ser uma coisa que um computador faz muitas vezes (de facto, tantas que há uns senhores no Estados Unidos que ficaram ricos só a fazer computadores para transferir dados de um lado para o outro...¹), mas (e até nos computadores para transferir dados de um lado para o outro) de vez em quando é preciso fazer contas. Para não complicar, vamos supor que só faremos contas de somar e subtrair; isso é feito com as instruções ADD (somar) e SUB (subtrair).

“à la C”	assembly	Descrição
AX += 10;	ADD AX, 10	AX ← AX + 10
AX += BX;	ADD AX, BX	AX ← AX + BX
AL = BL+CL;	MOV AL, BL	
	ADD AL, CL	AL ← BL + CL
BX -= CX;	SUB BX, CX	BX ← BX - CX
SI++;	INC SI	SI ← SI + 1
DI-;	DEC DI	DI ← DI - 1

Notem que o processador implementa as operações de C += e -=, não as operações de + e -. Por isso, fazer coisas como $x = y + z$ obriga a mais do que uma instrução. A instruções de incremento e decremento (inc e dec) têm a vantagem de serem mais rápidas e mais compactas do que somar ou subtrair 1 com add ou sub.

Tomar decisões O passo seguinte é dar alguma “inteligência” (será??...) ao programa, ou seja, torná-lo capaz de tomar decisões e fazer algo mais do que executar as instruções pela ordem em que estão escritas. Isso é feito com as instruções de salto (*jump*), incondicional ou condicional.

As instruções de salto condicional seguem sempre a mesma estrutura:

1. há uma operação (geralmente aritmética, lógica ou de “shift”) que afecta as flags do processador;
2. a instrução de salto testa essas flags e decide, conforme o seu valor, se a execução do programa vai *saltar* para uma outra posição ou se continua na instrução seguinte.

Suponhamos que se quer codificar em assembly a seguinte função:

$$f(x) = \begin{cases} x - 10 & , x \leq 100 \\ 90 & , x > 100 \end{cases}$$

(Sim, é verdade! O assembly faz isso, nem é preciso o Matlab!...) Para tal, o valor de x é guardado em AL e o resultado, $f(x)$, é devolvido em BL. Uma possível implementação é:

¹Para os mais curiosos: a companhia chama-se CISCO.

Flag = 1?	Flag = 0?	Flag testada
JC	JNC	CF, Carry Flag
JZ	JNZ	ZF, Zero Flag
JS	JNS	SF, Sign Flag
JO	JNO	OF, Overflow Flag
JP	JNP	PF, Parity Flag

Tabela 3: Salto condicional de teste às flags

Flag	Código	Flags a 1
Carry	mov ax,0f000h	
	add ax,1000h	CF,PF,ZF
Parity	mov al,3	
	or al,al	PF
	inc al	(PF=0)
Aux. Carry	mov ax,0fh	
	add ax,3	AC,PF
Zero	xor ax,ax	ZF,PF
Sign	xor ax,ax	
	dec ax	SF,AC,PF
Overflow	mov ax,7000h	
	add ax,1000h	OF,SF,PF

Tabela 4: Exemplos de actuação sobre as flags

```

CMP    AL,100
JA     MAIOR
SUB    AL,10
MOV    BL,AL
JMP    FIM
MAIOR: MOV    BL,90
FIM:

```

; E segue em frente...

A instrução `CMP` equivale a uma subtracção, mas em que o resultado é perdido; apenas as flags são afectadas. Neste caso, subtrai 100 ao registo `AL` sem guardar o resultado em nenhum registo. `JA` é a instrução *Jump if Above*, que faz o salto se o resultado da comparação for “o primeiro é maior do que o segundo” (ou o primeiro está “above” o segundo, perdoem-me a confusão de linguas...). As comparações são sempre o primeiro argumento em relação ao segundo.

As instruções de jump condicionais podem-se dividir em dois grupos: as que testam directamente uma flag e as que testam uma condição (o que geralmente envolve uma expressão com mais do que uma flag). A tabela 3 descreve os saltos condicionais com teste às flags e a tabela 5 as instruções de salto com teste a uma condição. Neste caso, supõe-se que se seguem a uma instrução `CMP x,y`. Para melhor compreender o funcionamento das flags, a tabela 4 apresenta alguns exemplos de instruções e o seu efeito sobre as flags. A flag indicada na primeira coluna é aquela que o exemplo pretende demonstrar; inevitavelmente, vem associada a outras, que são representadas na terceira coluna.

As instruções de teste a uma condição estão divididas em duas colunas: sem sinal e com sinal. A diferença entre elas é que umas consideram que os valores a comparar são sempre positivos (sem sinal). Neste caso, um registo de 8 bits pode conter um valor entre 0 e 255. Nas instruções com sinal, os valores a comparar podem ser positivos ou negativos (em complemento para 2). Para as instruções com sinal, um registo de 8 bits pode conter um valor entre -128 e +127 (tabela 6)². Para os testes de igual ou diferente, como é óbvio, não faz diferença ser com sinal ou sem sinal.

Para arredar os bits Outro tipo de instruções são as instruções de *shift* e *rotate*. Estas instruções permitem deslocar os bits de um registo para a direita ou esquerda, um determinado número de vezes. O formato da instrução

²Para os militantes de C: esta é a mesma diferença que entre os valores `signed` e `unsigned`.

Sem sinal	Com sinal	Teste
JA	JG	$x > y?$
JAЕ	JGE	$x \geq y?$
JB	JL	$x < y?$
JBE	JLE	$x \leq y?$
JE	JE	$x = y?$
JNE	JNE	$x \neq y?$

Tabela 5: Salto condicional com teste a uma condição

Valor do registo	Quantidade representada	
	Com sinal	Sem sinal
00 _h	0	0
01 _h	1	1
	(...)	
7F _h	127	127
80 _h	-128	128
81 _h	-127	129
	(...)	
FF _h	-1	255

Tabela 6: Quantidade representada num registo de 8 bits

é:

`inst dest, count`

em que *inst* é uma das instruções deste grupo, *dest* é o registo que vai ser deslocado e *count* o número de “deslocações” (shifts). O valor *count* só pode ser 1 (SHL AL, 1) ou o registo CL (SHR AL, CL). Neste caso, o número de deslocamentos é dado pelo valor do registo CL. A tabela 7 descreve a acção de cada uma destas instruções.

5 Chamar rotinas

Vamos escrever a rotina `tolower` (ou, “à la C”³: `tolower(char *)`⁴). Esta rotina recebe um valor de 8 bits (um `char`) e, se o valor representar o código ASCII de uma das letras ‘A’ a ‘Z’, transforma-o na representação da mesma letra em minúsculas; caso contrário, não lhe faz nada. Em termos matemáticos (sim, sim!, a programação tem a ver com matemática!), aplica ao valor de entrada a seguinte função:

$$\text{tolower}(x) = \begin{cases} x + 'a' - 'A' & , \text{ se } 'A' \leq x \leq 'Z' \\ x & , \text{ outros } x \end{cases}$$

A versão em assembly, em que tanto o valor de entrada como de saída são passados pelo registo AL é apresentada na figura 2⁵.

Dois pontos a notar sobre funções:

1. a rotina começa com o seu nome (*label*) seguido de dois pontos;
2. a rotina termina por uma instrução RET.

No caso da rotina `tolower`, a chamada desta rotina para converter para letra minúscula o carácter apontado pelo registo BX seria:

```
mov  al, [bx]    ; Carregar al com o valor a converter
call tolower    ; chamar tolower
```

Depois da chamada a `tolower`, o registo AL teria o resultado da conversão.

³Para os não-militantes de C: se não perceberem, não liguem e sigam em frente...

⁴Outra para os militantes de C: ou `char tolower(char)?` Qual delas é a correcta?

⁵Ainda sem dar descanso aos militantes de C e voltando à pergunta anterior sobre a declaração (prototype) da rotina `tolower`: olhando para o código em assembly desta rotina, as declarações anteriores estão correctas?

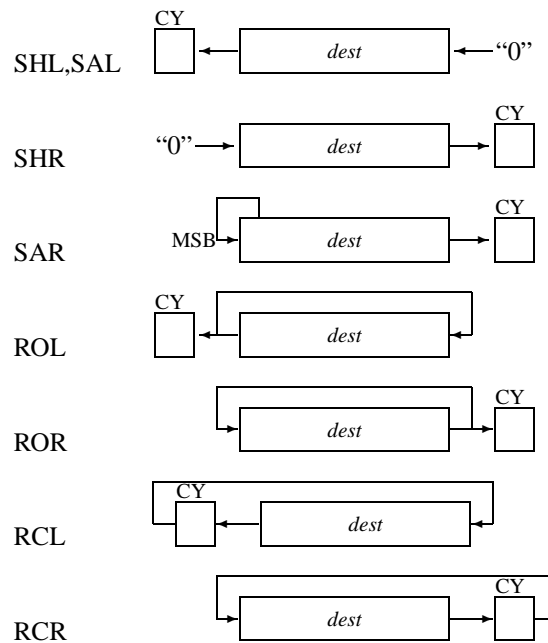


Tabela 7: Accção das instruções de deslocamento

Comentário sobre comentários e assuntos conexos... Os princípios da Engenharia de Programação (ou de Software, como lhe queiram chamar...) aplicam-se também ao assembly, nomeadamente no caso das rotinas. (Escrever em baixo nível não implica necessariamente programação de baixo nível, não sei se me faço entender...)

Uma rotina pode ser definida através de 3 coisas: a entrada, o processamento e a saída (ou em inglês **IPO**: Input, Processing, Output). Deve estar bem claro:

- a *entrada* da rotina, indicando quais são os valores passados para a rotina e como são passados;
- aquilo que a rotina faz, ou seja o *processamento*) que é feito aos valores de entrada; e
- a *saída* da rotina, isto é, quais os valores devolvidos.

O reaproveitamento de código deve também ser explorado (sob pena de ter que voltar a escrever aquilo que já se tinha feito). Para que isso seja possível é preciso que a descrição da rotina especifique bem o que faz e o seu interface (entrada/processamento/saída). Para além disso, devem também estar bem evidentes quais são os seus “efeitos secundários”: por exemplo, quais são os registos que são usados internamente e cujo valor é modificado⁶. A rotina *tolower* exemplifica como isso pode ser feito.

Passagem de argumentos As funções e rotinas são procedimentos (ou algoritmos) que operam sobre determinados valores (os argumentos de entrada) para devolver um ou mais valores (argumentos de saída). Ao escrever uma rotina, é necessário definir como é que essa rotina vai saber qual o valor a trabalhar.

Uma das hipóteses é, como está na rotina *tolower*: a passagem pelos registos do microprocessador. Neste caso, os valores dos argumentos de entrada são colocados nos devidos registos do processador, a rotina é chamada e o resultado estará também guardado num ou em vários registos no fim da chamada.

(Tópico avançado) Uma hipótese alternativa (que é usada, por exemplo, em linguagens como C ou Pascal) é a passagem dos argumentos pela *stack*. Neste caso, é feita a escrita dos valores de entrada na *stack* antes da chamada, e depois o resultado é recuperado da *stack*. Ao utilizar a passagem de argumentos pela *stack*, é preciso não esquecer que estamos a usar a mesma *stack* para onde vai o endereço de retorno da rotina. Quando há um `call`, o endereço da instrução para onde o programa deve voltar no final da rotina é guardado na *stack*. Quando a instrução `ret` é encontrada, o programa parte do princípio que “à boca da *stack*” está o endereço de retorno e salta para o endereço que ler da *stack* (o que quer que esse valor seja; ele só sabe que são dois bytes). Se entretanto a *stack* foi “escangalhada”, o resultado é a catástrofe... Há que ter o cuidado de garantir que dentro de cada rotina,

⁶Como corolário: uma rotina correctamente escrita não altera registos (ou posições de memória) que não sejam os usados para o interface. Se for necessário usar registos internamente (ou...), o seu valor é armazenado à entrada e reposto à saída da rotina. Em termos de C, equivale a evitar o uso de variáveis globais

```

; tolower
;
; Converte um caracter de maiúscula para minúscula.
; Se o valor a converter não for uma letra maiúscula,
; não o altera.
;
; Recebe:
;   AL - valor a converter
; Devolve:
;   AL - valor convertido
; Modifica:
;   Nenhum outro registo
; Afecta flags.
tolower:  cmp    al, 'A'
          jb     fim      ; se esta antes de A, salta fora
          cmp    al, 'Z'
          ja     fim      ; se esta' para alem de Z, tambem
          add    al, 'a'
          sub    al, 'A'
fim:      ret

```

Figura 2: Rotina tolower

o número de pushes e pops é o mesmo. Note-se a propósito que push e pop não são a única maneira de ler ou escrever na stack⁷.

6 Como o 80188 acede à memória

Ao falar das instruções do assembly, falámos brevemente da maneira do processador aceder à memória. Vamos agora ver com um pouco mais de pormenor quais as maneiras de que dispomos (sintacticamente...) para aceder à memória.

A tabela 8 apresenta as opções disponíveis para aceder a posições de memória com o exemplo da instrução MOV. Todos estes modos são aplicáveis por analogia às outras instruções (ADD, SUB, CMP, ...).

Assembly	Modo	Coment.
MOV AX, <i>k</i>	Immediate	
MOV AX, <i>r</i>	Register	
MOV AX, [<i>k</i>]	Direct	
MOV AX, [<i>r</i>]	Register indirect	$r \in \{BX, BP, SI, DI\}$
MOV AX, [<i>r_b</i> + <i>k</i>]	Based	$r_b \in \{BX, BP\}$
MOV AX, [<i>r_i</i> + <i>k</i>]	Indexed	$r_i \in \{SI, DI\}$
MOV AX, [<i>r_b</i> + <i>r_i</i> + <i>k</i>]	Based Indexed with Displacement	

k: constante, *r*: registo

Tabela 8: Modos de endereçamento

Os endereços passados nas instruções são offsets (são valores de 16 bits). Para calcular o endereço efectivo, é necessário somar-lhe o segmento (de acordo com as normas que já vimos!). O registo de segmento utilizado é normalmente o DS (ou seja, as posições de memória são offsets relativos ao segmento de dados). A excepção é o BP, em que o endereço é calculado usando o SS (Stack Segment, Tabela 9). Pode-se forçar a utilização de um outro segmento, indicando-o expressamente. Por exemplo

```
mov ax, ds:[bp]
```

⁷Note-se a propósito que não é inocente a existência de um registo (BP) que utiliza o SS (e não o DS) como segmento no acesso à memória

Registo	Endereço associado
BX, SI, DI	DS: <i>r</i>
BP	SS: <i>r</i>

Tabela 9: Segmentos associados aos registos de index e base.

força a utilização do registo DS com acessos usando o BP como ponteiro.

7 Gramática do assembly

Na figuras 3 e 4 apresentam-se dois “esqueletos” para programas em assembly, uma com segmentação e a outra colocando dados e código no mesmo segmento. O código na figura 4 contém também alguns exemplos de definição de constantes e variáveis em assembler, que estão explicadas na tabela 10 com o seu equivalente em C.

“à la C”	assembly
#define TRUE 1	TRUE EQU 1
char val=7;	val DB 7
int pval;	pval DW ?
char array[16];	array DB 16 DUP(?)
char *str="Ola!"	str DB "Ola!"

Tabela 10: Directivas de assembler: definição de constantes e variáveis

As directivas EQU, DB, DW e DUP são então utilizadas para definir contantes e variáveis.

EQU atribui um valor a uma label.

DB, DW Reserva espaço na memória, a que é atribuído uma label. DB reserva um byte, DW reserva dois bytes (uma “word”). O valor pode ser inicializado (escrevendo o valor a seguir a DB ou DW) ou não (com ?).

DUP A directiva n DUP reserva n blocos (bytes ou words). Pode ser inicializado (n DUP(val)) ou não (n DUP(?))

As labels para definição de valores são escritas sem dois pontos (':').

SEGMENT, ENDS Definem o começo e o fim de um segmento.

ASSUME Indica ao compilador que pode calcular todos os endereços (do programa ou dados) supondo que os registos CS e DS (code segment e data segment) estão a apontar para os segmentos indicados. Por exemplo, assume ds:data diz ao compilador que o DS vai estar a apontar para o segmento chamado “data” e que os offsets das variáveis são referidos a este segmento. No entanto, não dá nenhuma instrução para carregar o registo DS com o que quer que seja! É por isso que é necessário (de preferência logo no início do programa) carregar o registo DS com o valor do segmento referido. Isso é feito com as instruções mov ax, data e mov ds, ax. (A solução lógica seria mov ds, data mas infelizmente é inválida...)

END Indica o fim do programa.

ORG Define o endereço (offset no segmento respectivo) em que é colocada a linha de código que se segue (que tanto pode ser código como dados como ...)

Terminamos com a explicação das directivas BYTE PTR e WORD PTR. Estas directivas servem para resolver situações em que o compilador não consegue descobrir se deve trabalhar com valores de 8 bits (BYTE) ou 16 bits (WORD). Por exemplo, o que significa a instrução seguinte?

```
mov [bx], 0
```

“Escrever 0 na posição de memória apontada por BX” corresponde a qual das situações da figura 5. a) ou b)? Só pela instrução acima, o compilador não sabe se queremos escrever 1 ou 2 bytes com zero (8 ou 16 bits). Para isso são necessárias as directivas BYTE PTR e WORD PTR:

```

; esquelet.asm
;
; "Esqueleto" basico para um programa em assembler.
; Versao com um so' segmento
;
; Pedro Fonseca, Out 98
; Departamento de Electronica
; Universidade de Aveiro

code    segment
        assume cs:code,ds:code

prog:   jmp start                ; "salta por cima" da
                                   ; area de dados

;   Area de dados
;
;   Colocar aqui os DB e DW (constantes e variáveis)
;
;   Colocar definicao de rotinas
;

; *****
; *
; *   Programa
; *
; *****

start:  mov ax,code              ;Codigo do programa
        mov ds,ax              ;comeca aqui...

        ; Corpo do programa
        ; Corpo do programa
        ; Corpo do programa

        ; ... e termina aqui.
        int 10h                ; Terminacao do programa no kit

code    ends                    ; Fim do segment de codigo

        end prog                ; Fim do programa assembler

```

Figura 3: Estrutura de programa sem segmentação

```

; esquelet.asm
;
; "Esqueleto" basico para um programa em assembler.
; Versao com segmentos
;
; Pedro Fonseca, Out 98
; Departamento de Electronica
; Universidade de Aveiro

dados    segment
        ; Incluir aqui
        ;         - todas as definicoes de variaveis
        ;         - todos os DB e DW
        ;; Exemplos:
TRUE     EQU 1
FALSE    EQU 0
val      DB 7
pval     DW ?
array    DB 16 DUP(?)
zeros    DB 16 DUP(0)
dados    ends                ; Fim do segmento de dados

codigo   segment            ; Inicio do segmento de codigo
        assume cs:codigo,ds:dados

prog:    mov ax,dados        ; Inicializaçãõ do registo DS
        mov ds,ax

        ; Codigo do programa
        ; começa aqui.

        int 10h            ; Terminaço do programa no kit

codigo   ends                ; Fim do segment de codigo

end prog                ; Fim do programa assembler

```

Figura 4: Estrutura de um programa com segmentação

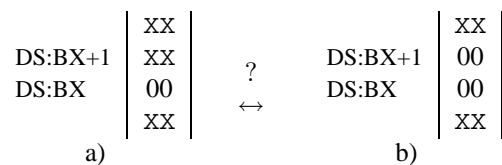


Figura 5: Escrita de 8 ou 16 bits

```
mov byte ptr [bx],0
```

indica expressamente que só queremos copiar um byte (a) na figura 5), e

```
mov word ptr [bx],0
```

indica que queremos copiar dois bytes (uma word) (b) na figura).

8 Uso do debug ou Vamos por a mão na massa

Como o 80188 faz parte da família x86, os programas em assembly do 80188 podem correr nos PCs, o que permite ensaiar os programas e resolver dúvidas de sintaxe da linguagem.

Para isso, é necessário compilar o programa, “linká-lo” e corrê-lo dentro do debug. Para um programa chamado mem.asm, é necessário:

```
C:\users\pf\Aulas\IP\aula1>masm mem;  
C:\users\pf\Aulas\IP\aula1>link mem;  
C:\users\pf\Aulas\IP\aula1>debug mem.exe  
-
```

Algumas observações:

1. No programa em assembly, é necessário alterar o interrupt de terminação do programa:

```
No kit:      Em DOS  
int 10h  mov ah,4ch  
          int 21h
```

2. O debug carrega o programa numa posição de memória livre e inicializa os registos CS e IP. Assim, basta fazer g (go) para correr o programa.

De resto, todos os comandos válidos para o CTERM são válidos também para o DEBUG (na verdade, o CTERM implementa um subconjunto dos comandos do DEBUG). O comando de help é ? (e não h, como no CTERM),

Agora, é só experimentar!... E, se não estiverem a usar DOS (que é um sofisticadíssimo sistema operativo que, por razões de segurança, não permite mais do que um processo ao mesmo tempo), não estejam a fazer nada de importante ao mesmo tempo que correm os programas em assembler...