⊛ References

مراجع :

1) Peter Abel, IBM PC ASSEMBLER LANGUAGE AND PROGRAMMING, Prentice Hall

آخرین version

2) 80386/80286 Assembly language Programming William H. Murray, III and Chris H. Pappas, Osborne McGraw-Hill

3) Assembly language Programming for the Intel 80xxx Family, William B. Giles, Maxwell Macmillan International Editions.

4) PC intern, Micheal Tischer, B. Jennrich, Abacus

5) The 8086 Book includes the 8088, Russell Rector, George Alery, Osborne, McGraw Hill.

۸۲/۱۲/۲۴

* کلاً فضاهای مختلف داده‌ها و دستورالعمل اول ارائه شوند

* مقدمه‌ای بر درس:

واحدهای مختلف در یک ریزپردازنده بصورت چکیده عبارتند از Control Unit (CU) یا واحد کنترل کننده و Arithmatic and logic Unit (ALU). واحد کنترل کننده خود شامل بخشهای مختلفی می‌تواند باشد برای مثال در یک ریزپردازنده 8086، آن را می‌توان به دو قسمت عمده تقسیم نمود: Execution Unit (EU) و Bus Interface Unit (BIU) که اسامی هرکدام بیانگر کار هر یک می‌باشد. واحد EU وظیفهٔ اجرا دستورالعملها را در یک ریزپردازندهٔ 8086 بعهده دارد. BIU وظیفهٔ تهیهٔ دستورالعمل و داده‌ها را برای EU و کنترل ورود و خروج اطلاعات را بر روی data Bus و Address Bus عهده دار می‌باشد. در حقیقت هر واحد کنترل در ریزپردازنده‌ها وظیفهٔ برداشت و اجرای دستورالعملها را بعهده دارد (fetch & Execute). تمام فلسفهٔ کار یک ریزپردازنده را در همین fetch & Execute می‌توان خلاصه نمود. حال که با این مفاهیم آشنائی حاصل گشت نیاز به برداشتن و اجرای دستورالعمل بخوبی احساس می‌شود. این خود نیاز به ساختاری جهت پیاده‌سازی نگاشت حافظه را نشان می‌دهد. شکل 1-1 این ساختار را نمایش می‌دهد.

* فضای آدرس دهی در یک ریزپردازنده:

فضای آدرس دهی یک ریزپردازنده عموماً به دو بخش کاملاً مجزا IO (ورودی و خروجی) و Memory (حافظه) تقسیم می‌گردد. هر یک از این دو قسمت توسط ریزپردازنده قابل دسترسی بوده و می‌توان از آنها خواند و یا بر روی آنها نوشت (در صورت قابل نوشتن بودن). البته دستورالعملهای متفاوتی جهت کار با فضای IO و حافظه مورد نیاز است که هر یک در CPU تعریف شده‌اند. برای اجرای هر دستورالعمل باید ابتدا آن را از حافظه خواند، در نتیجه CPU تنها قادر است که محتویات حافظه را اجرا کند. در این بین معمولاً چندین سلول حافظه نیز در داخل هر ریزپردازنده‌ای قرار داده شده است تا از آنها جهت تسریع عملیات و محاسبات استفاده گردد. کلاً نگاشت هر دو فضاهای حافظه و IO خطی می‌باشند که از صفر آغاز گردیده و تا انتهای فضای آدرس دهی ادامه می‌یابند. دسترسی به هر یک از این دو فضا توسط گروهی متفاوت از دستورالعملها انجام می‌شود. به عنوان مثال برای دسترسی به فضای IO معمولاً از دستورات IN و OUT استفاده می‌شود. برای دسترسی

به فضای حافظه نیز از دستورات مخصوص به آن که می‌تواند بصورت MOV و یا LD (Load) باشد می‌توان استفاده نمود.

* آدرس در ریزپردازنده های INTEL مدل 80X86

در این نوع ریز پردازنده ها علاوه بر ثباتی (Register) بنام PC که وظیفهٔ آن دنبال نمودن اجرا برنامه می‌باشد (شمارندهٔ برنامه و یا دستورالعمل) چهار ثبات دیگر هم در ساخت آدرس ها در سیستم مؤثر می‌باشند. این ثبات ها عبارتند از:

CS (Code Segment) قسمت کد

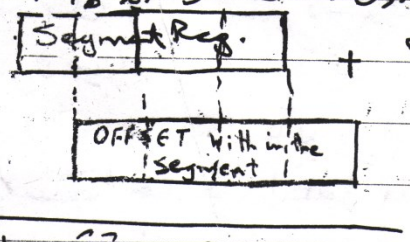
SS (Stack Segment) قسمت پشته (Stack)

DS (Data Segment) قسمت داده

ES (Extra Segment) قسمت اضافی

که به این ترتیب حافظه اصلی سیستم (مانند RAM و ROM) به قسمتهای مجزا از هم تقسیم می‌شوند (Segmentation). مقادیر موجود در این ثبات ها در هر لحظه قابل تغییر می‌باشند. سایز یا اندازهٔ فضای داخل هر یک از این قسمتها در 8086 64Kbytes می‌باشد. یعنی یک فضای آدرس 16 بیتی.

* محاسبهٔ آدرس حقیقی با استفاده از ثبات قسمت کد (CS) و ثبات PC همانطور که در (شکل ۲-۱) مشاهده می‌شود آدرس حقیقی در ریزپردازنده های INTEL

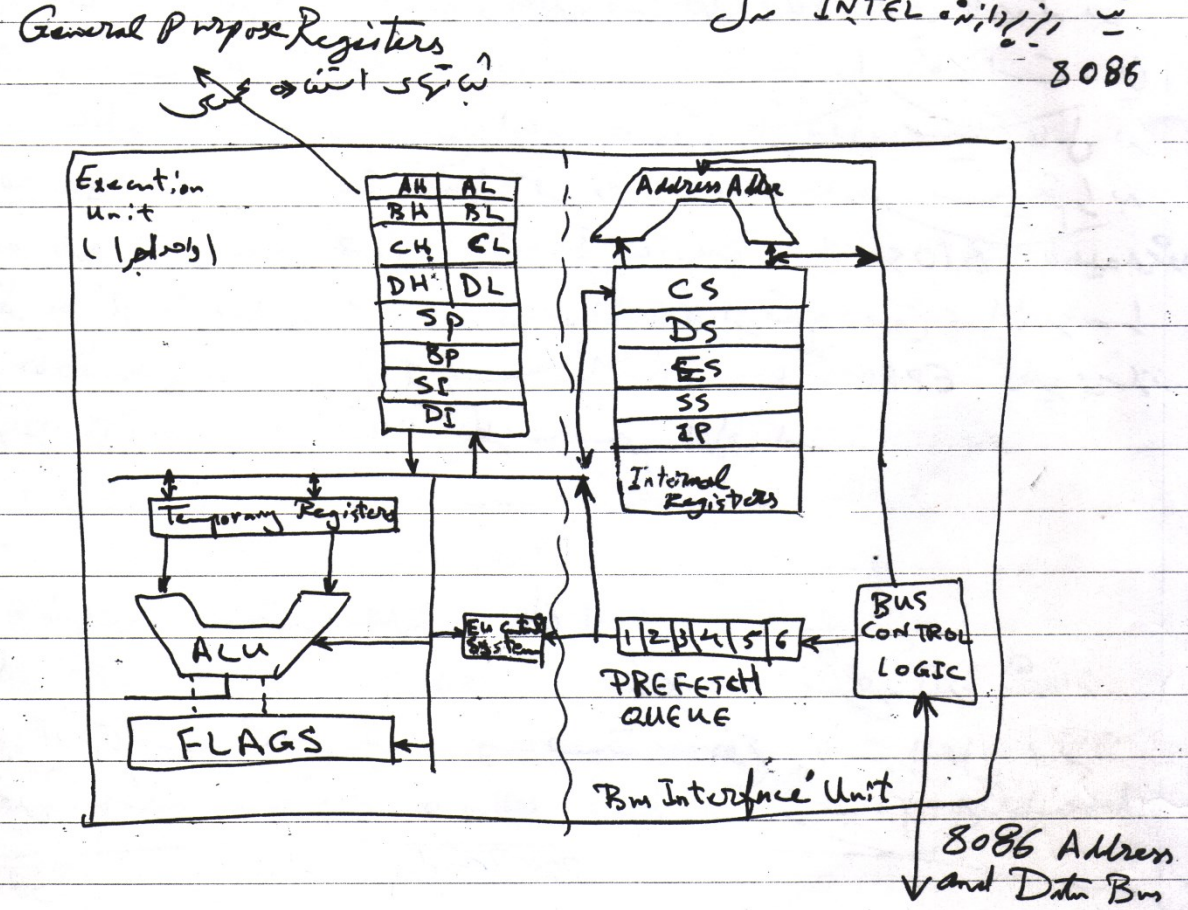


(شکل ۲-۱)

طرز محاسبهٔ آدرس حقیقی بکمک آدرس نسبی در هر قسمت و مقدار رجیستر قسمت.

مدل 80X86 با حرکت دادن ثبات قسمت به سمت چپ به اندازهٔ 4 بیت و جمع کردن آن با آدرس نسبی آن محل در قسمت حاصل می‌گردد. بکمک این روش هر مکان در فضای حافظه یک آدرس 16 بیتی خواهد داشت. اما آدرس واقعی 20 بیتی خواهد بود. این امر باعث می‌شود که آدرسهای متفاوتی برای یک خانهٔ حافظه وجود داشته باشند (مفهوم سایه

شکل ۱-۱ : ساختار داخلی
یک ریزپردازنده INTEL مدل
8086



فضای آدرس در یک ریزپردازنده:

در یک ریزپردازنده فضای آدرس بصورت خطی می‌باشد. یعنی از آدرس 0000H شروع شده و تا آدرس FFFFH (برای یک سیستم 16 بیتی) بطور پیوسته و متوالی ادامه خواهد داشت. شکل 3-1

شکل 3-1 : فضای آدرس در یک ریزپردازنده با ۱۶ بیت آدرس.

در یک مدار بر مبنای ریز پردازنده، پس از هر بار ریست شدن CPU بطور خودکار از آدرس خاصی شروع به برداشتن و اجرای دستورالعمل می‌نماید. اولین کاری که در این وضعیت صورت می‌گیرد آماده سازی سیستم برای کار است. این خود شامل آماده سازی سخت افزارهای جانبی و بار کردن سیستم عامل در حافظه می‌باشد. این برنامهٔ ابتدائی که وظیفهٔ آماده سازی سیستم را به عهده دارد با نام Basic Input Output System یا BIOS نامیده می‌شود. نظر به نیاز به پایداری این برنامه در حافظه حتی بعد از قطع برق، این برنامه را در حافظه‌های پایدار از نوع ROM و یا EPROM ذخیره می‌سازند. وجود و اجرای این برنامه برای آغاز بکار سیستم الزامی است.

*) قالب‌های دستورالعمل در زبان اسمبلی

بطور کلی قالب‌های دستورالعمل در شکل ٤-١ شرح داده شده‌اند. ولی این قالب ثابت نبوده و بسته به نوع دستورالعمل می‌تواند متغیر باشد. در نهایت این دستورالعمل توسط برنامه‌ای به نام Assembler

Operand = عملوند

شکل ٤-١

OP-code	Operand 1	Operand 2

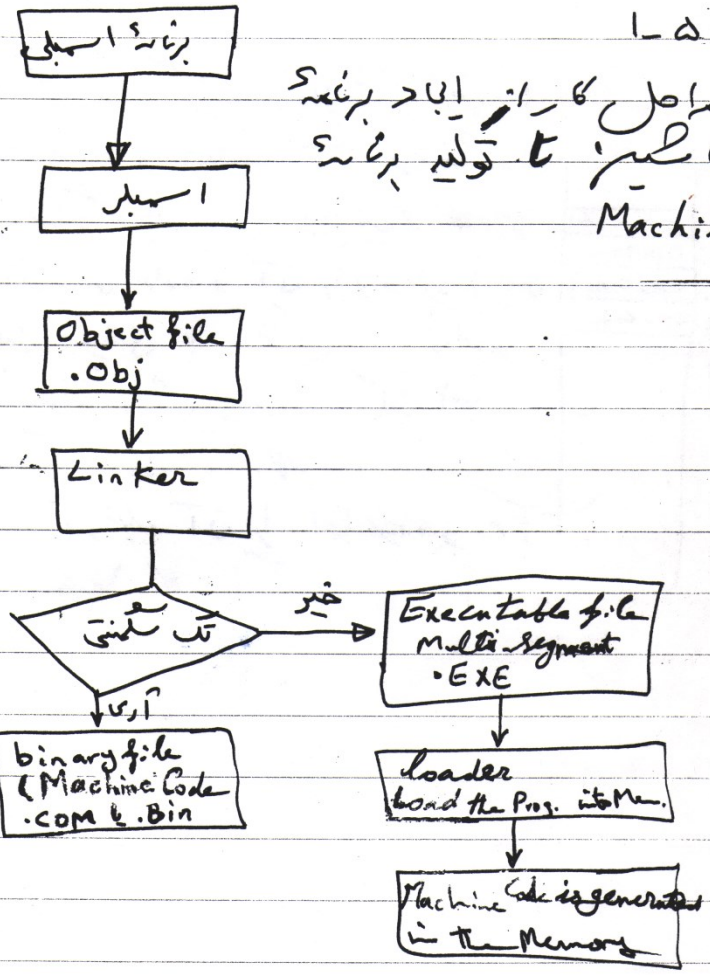
OP_Code : Operational Code که عملیاتی دستورالعمل است که از روی آن CPU تعداد Operand های مورد نظر، قالب دستورالعمل و وظیفه دستورالعمل را مشخص می‌نماید.

Operand : پارامترهای دستورالعمل هستند. معمولاً یکی برای برداشتن و دیگری محل قرار دادن است. از هر دو می‌توان در انجام عملیات استفاده نمود. یعنی از Operand ها بطور مجازی در دستورالعمل موجود می‌باشند.

Read Only Memory (ROM) -١

Erasable Programmable Read only Memory (EPROM) -٢

به کدهای عددی تبدیل می‌شود که به آن کد ماشین یا Machine Code می‌گویند. حال این کدهای ماشین هستند که برای ریزپردازنده قابل درک و قابل اجرا می‌باشند. مراحل کار تا تولید کد ماشین بصورت زیر می‌باشد (شکل ۵-۱)



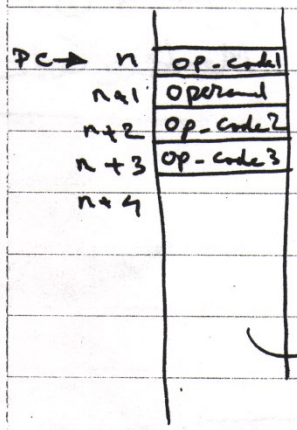
شکل ۵-۱
مراحل کار از ایجاد برنامهٔ زبان ماشین تا تولید برنامهٔ Machine Code

Loader قسمتی از سیستم عامل می‌باشد که وظیفهٔ تخصیص حافظه و قرار دادن قسمتهای مختلف از برنامه (Segments) را در قسمتهای مختلف حافظه به عهده دارد. در این حین بسته به سگمنتهایی که مورد استفاده قرار گرفته‌اند مقادیر آدرسهای برنامهٔ اصلی تغییر داده می‌شوند.

Linker : وظیفه این برنامه، استفاده از فایلهای حاصل از برنامه اسمبلر (فرمت obj) و متصل نمودن آنها به یکدیگر است، چه از لحاظ فیزیکی و چه از لحاظ آدرسها و اسامی بکار رفته در فایلهای obj مختلف. حاصل کار یک فایل قابل اجرا توسط سیستم عامل می باشد.

* نحوه اجرای یک دستورالعمل در یک پردازشگر فرضی:

ابتدا PC مساوی n است و OP_Code1 برداشته شده و واحد کنترل متوجه می شود که این دستورالعمل یک operand دارد پس PC را یکی افزایش داده و آن را می خواند و سپس دستورالعمل را اجرا می کند. سپس به PC یکی اضافه می کند و دوباره دستورالعمل را خوانده و آن را تفسیر می کند اینجا CU متوجه می شود که این OP_Code، دارای Operand نیست و در نتیجه آن را اجرا نموده و PC را اضافه می کند و همین کار را تکرار می نماید.



قالبهای متفاوت داده ها

Bit کوچکترین واحد حافظه است، ویژه ها یا بیت [0 1]

Nibble هر چهار بیت یک Nibble گویند [0 15]

Byte ~ Byte ~ ~ هر هشت [0 255]

Word ~ Word ~ ~ هر شانزده [0 65535]

Double Word هر دو Word را یک Double Word گویند

Quad Word ~ ~ ~ ~ چهار Quad Word گویند

Ten Bytes شامل ده بایت بوده و بزرگترین اندازه عددی برای 80286 باشد

انواع داده ها

Character معادل عددی یک حرف در جدول ASCII است که معادل یک بایت باشد (بدون علامت)

نمایش عددی

Binary Integer Numbers اعداد صحیح مبنای دو

* اعداد صحیح بر دو گونه اند، علامت دار و بدون علامت

Unsigned Integer بسته به طول بیتی آن فاصلهٔ عدد متفاوتی را توانند پوشانند. برای یک عدد n بیتی: $[0 (2^n-1)]$

Signed Integer اعداد صحیح علامت دار بر دو نوع هستند

2's Complement مکمل دو

این اعداد در یک سیستم مکمل دو کار می کنند. تصویر آنها می تواند منفی، صفر و یا مثبت باشد. وجود یا نبود پرارزشترین بیت نشانهٔ منفی بودن عدد است

Sign_Magnitude نمایش داده با مقدار علامت دار

در این روش پرارزشترین بیت را بیت نشان دهندهٔ علامت در نظر می گیرند. یک بودن این بیت بیانگر منفی بودن عدد است

Binary Coded Decimal (BCD)

در این نوع نمایش اعداد بصورت ده دهی نوشته می شوند که هر عدد شامل ۴ بیت می شود. نام دیگر این نوع نمایش Packed-Decimal باشد

معمولاً ۱۸ رقم و یک بایت علامت دارد.

Floating Point Number — اعداد با ممیز شناور (اعشاری)

بر سه نوع است:

- Short Real — 31 | S | Biased exponent (23) | Significand (0) — $[1.2\times10^{-38}\ \ 3.4\times10^{38}]$
- Long Real — 63 | S | Biased Exponent (52) | Significand (0) — $[2.3\times10^{-308}\ \ 1.7\times10^{308}]$
- Temporary Real — 79 | S | Biased Exponent (64) | Significand (0)

در هنگام استفاده در سیستم دو فرم اول به Temporary Real تبدیل می‌شوند.

Short real	127	Biased Exponent شده تعدیل
Long real	1023	
Temporary	16383	

قالبهای استور العمل (ثابت و متغیر)

<OP_code> <Operand1> <Operand2>

تکنیکهای آدرس دهی

1. Immediate addressing

در این تکنیک یکی از Operand ها مقدار عددی می‌باشد و operand دیگر یک رجیستر.

```
MOV AH, 04
```

2. Register addressing

در این تکنیک هر دو operand رجیستر می‌باشند. بعنوان نمونه

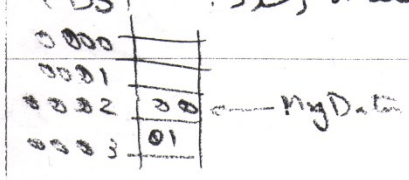
```
MOV DS, AX
MOV DL, AL
```

3. Direct Addressing

در این حالت یکی از operand ها آدرسی از حافظه می‌باشد. بعنوان مثال:

```
MOV AX, MYDATA
```

در این حالت محدوده از یکی از رجیسترهای قسمت Segment Registers هم استفاده می‌شود.




```
MOV AX, DS:0002
MOV AX, DS:MYDATA
```

4. Register Indirect Addressing

در این حالت آدرس دهی مقدار مورد نظر توسط آدرسی که در رجیسترهای SI ، DI ، BX و در بعضی مواقع BP به آن اشاره می‌شود را مورد استفاده قرار می‌دهد. بعنوان مثال: MOV AX,[BX]

این روش برای آدرس دهی خانه‌های آرایه قابل بکارگیری است.

DS:BX, DS:SI, DS:DI, SS:BP

5. Direct Indexed Addressing

در این روش آدرس دهی، عامل آدرس با اضافه کردن یک جابجایی به رجیسترهای اندیس‌دار (SI یا DI) در داخل قسمت انتهایی (شمنت انتهایی) انجام می‌شود. این روش معمولی‌ترین راه دسترسی به خانه‌های یک آرایه می‌باشد. در این حالت جابجایی آدرس پایه آرایه و DI ، SI در نقش اینديس برای آن خواهند بود. بعنوان مثال: MOV AX, ARRAY [SI]

DS:SI, DS:DI

6. Base Relative Addressing

در این روش آدرس مؤثر از اضافه کردن جابجایی (مقدار ثابت) به رجیسترهای BX و یا BP در شمنت مورد نظر حاصل می‌شود. بعنوان مثال، MOV AL, [BX]+4

DS:BX or SS:BX

7. Base Indexed Addressing

در این حالت آدرس مؤثر بعد از اضافه کردن تغییر مکان (عدد ثابت) به Base Reg (BX) و Index Reg. (SI, DI) در یک شمنت مشخص حاصل می‌شود. بعنوان مثال آرایه ELEMENT را در نظر بگیرید که فیلدهای مختلفی دارد. هر فیلد را می‌توان بصورت زیر مشخص نمود.

MOV AX, ELEMENT [BX][DI]

٣*

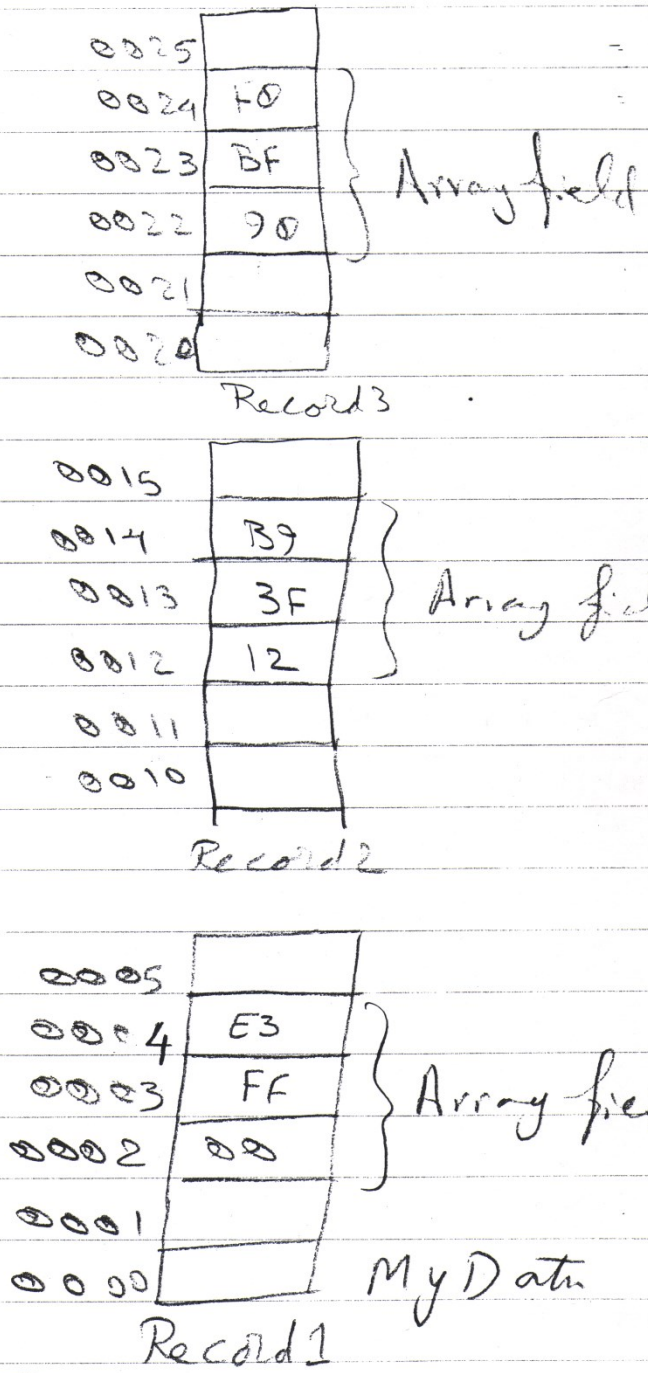
Base Indexed Addressing Mode مثال :

```
MOV AX, ELEMENT [BX] [DI]
```

BX [0000]

DI [0020]

ELEMENT EQU 3



AX [1111000010111111]

F0 BF

* برنامه نویسی بزبان اسمبلی 80x86

شکل کلی هر خط از برنامهٔ اسمبلی به فرم کلی زیر می باشد.

LABEL	MNEMONIC	OPERANDS	COMMENT
START (:)	MOV	AX, DS	(;) Move DS to AX

در این مثال یک دستورالعمل با دو Operand نمایش داده شده است، بعضی از دستورالعملها فاقد Operand هستند.

البته فاصله بین قسمتها در داخل یک خط می تواند متغیر (به دلخواه) باشد ولی حداقل یک خانه فاصله بین MNEMONIC و OPERANDS لازم است.

* دستورالعملها می توانند بدون Operand، با یک و دو یا بیشتر Operand باشند.

— در برخی دستورالعملها که دارای تنها یک Operand می باشند، Operand می تواند یک ثبات A یا 12 بیتی یا یک رجوع به حافظه (آدرس حافظه) باشد و نمی تواند یک داده فوری (immediate) باشد.

— دستورالعملهایی که دارای دو Operand می باشند، هر دو Operand باید یک سایز داشته باشند.

— در یک دستورالعمل با دو Operand یکی از آنها باید یک ثبات باشد مگر اینکه Operand منبع از نوع فوری (immediate) بوده که در این صورت Operand مقصد می تواند ثبات یا آدرس حافظه باشد.

* فرم کلی دستورالعملها با دو Operand بصورت زیر است.

فرم کلی: inst Destination, Source

```
inst Reg, Reg
inst Reg, Mem
inst Mem, Reg
inst Reg, Immed
inst Mem, Immed
```

— در مورد ثباتهای قسمت (Segment Registers) ترکیبات خاصی در شکل دستورالعملها وجود دارد که به قرار زیر می باشند.

- یک ثبات قسمت تنها می‌تواند در دستورالعملهای MOV ، PUSH و POP مورد خواندن و نوشتن قرار گیرد
- نمی‌توان یک مقدار فوری (immediate Value) را به داخل یک ثبات قسمت انتقال داد.
- انتقال از یک ثبات قسمت به ثبات قسمت دیگر ممنوع می‌باشد.
- ثبات CS نمی‌تواند مقصد برای دستورالعملهای MOV و POP باشد.

انواع دستورالعملها در 8086 :

* دستورالعمل MOV : بایتی و کلمه‌ای

```
MOV AL, BL
MOV AX, BX
```

* دستورالعملهای بیتی

BSF/BSR (Scan bit forward/reverse)

```
BSF  des_Reg, Src_Reg
BSR  des_Reg, Src_Reg
```

{ if Src_Reg = 0 then ZF = 1
{ if Src_Reg ≠ 0 then ZF = 0 , des_Reg = Position of 1 in Src_Reg

BT/BTS/BTR/BTC reg/mem, selector

Bit test/set/reset/Complement

عمل مورد نظر را بر روی بیت انتخابی توسط selector انجام می‌دهد.

* دستورالعملهای انشعاب و کنترل حلقه

دستورالعملهای انشعاب بر دو نوع هستند: شرطی و غیرشرطی

- انشعاب غیرشرطی (Unconditional Jump)

در این نوع انشعاب که تنها دستور JMP انجام می‌گیرد شرط

جهت انجام عمل پرش به آدرس مقصد وجود ندارد. محل انتقال اجرا برنامه به محل جدید و یا به فرمولی محاسبه، آدرس مورد نظر برای پرش به دو صورت انجام می‌شود.

- در نوع اول آدرس کمک یک تغییر مکان هشت بیتی (+127 تا -128) محاسبه می‌شود. در این حالت تغییر مکان تنها می‌تواند در داخل سگمنت کد انجام شود. برای مشخص کردن این نوع پرش یا در محل Label (برای زیر برنامه‌ها) از کلمه‌ی NEAR استفاده می‌نماییم و یا در بدنه دستورالعمل از دستور نمادین SHORT برای اینکار بهره می‌بریم. JMP SHORT A10

- در نوع دوم آدرس پرش کمک یک آدرس 16 بیتی محاسبه و اجرا به آن منتقل می‌گردد. در این حالت آدرس هدف می‌تواند خارج از قسمت فعلی کد باشد.

• انشعاب شرطی (Conditional Jump)

در این نوع انشعاب، جهت انتقال مسیر اجرا به محل جدید، شرطی خاص باید برقرار باشد. پرش در این نوع دستورالعمل‌ها از نوع SHORT می‌باشند. این دستورالعمل‌ها وجود حالت‌های مختلف در عملیات برای دربردارنده تنوع دستوری وسیعی دارد. که از انواع آن می‌توان به موارد زیر اشاره کرد:

JE jump on equal
JZ jump on zero — تقریباً شبیه هستند — Zf=1
JNZ jump on not zero → Zf=0
JG jump on greater

مثال آورد. این شرطی کمک بیتهای متناظر برای آنها در ثبات پرچمها (۱) که گاهی ثبات وضعیت (۲) هم نامیده می‌شود شناسائی می‌شوند. این ثبات در بخش‌های بعدی معرفی خواهد شد و مکانیزم یک از پرچمها و معنی آنها شرح داده خواهند شد.

علاوه بر دستورالعمل‌های انشعاب شرطی، جهت تسهیل پیاده‌سازی حلقه‌های تکرار در زبان اسمبلی، دستورالعمل خاصی نیز برای اینکار در نظر گرفته شده است.

۱- Flags Register
۲- Status Register

* دستورالعمل LOOP :

این دستورالعمل بطور ضمنی از ثبات CX بعنوان شمارنده کاهنده خود استفاده می‌نماید و در چهار فرم قابل استفاده می‌باشد. این فرم‌ها دو به دو متناظرند

(loop if equal) LOOPE (یا LOOPZ) ZF=1 و CX≠0 آگر Operand ، پرش به آدرس ، CX ← CX-1

(loop if not equal) LOOPNE (یا LOOPNZ) ZF=0, CX≠0 آگر Operand پرش به آدرس ، CX ← CX-1

LOOP CX≠0 آگر Operand پرش به آدرس ، CX ← CX-1

طرز استفاده از این دستورالعمل در ساختار زیر نشان داده شده است

```
MOV AX, 0          ; clear AX
MOV CX, 10         ; Load the Counter Reg. with the Max count
Iterate:
        INC AX
        LOOP Iterate
                   ; AX = 10 at the end of the loop
```

* ثبات‌های کنترل و وضعیت در ریزپردازنده‌های 80x86

ثبات وضعیت[1] یا ثبات پرچمها[2] در یک ریزپردازنده 80x86 جهت تعیین وضعیت CPU پس از انجام هر دستورالعمل مورد استفاده قرار می‌گیرد. محل هر کدام از پرچمها وضعیت و وظیفه آنها در زیر نشان داده شده است.

Bit 15													Bit 0	
	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	Flags
															Instruction Pointer
										ET	TS	EM	MP	PE	Machine Status Word

(۸ و ۱۶ بیتی شل جمع و تفریق)

Carry Flag (CF) : در حین عملیات محاسباتی کلا برای حالت دو برابر (شل ۱۶ بیتی) استفاده می‌شود. در عین حال در عملیات شیفت و دوران هم مورد استفاده قرار می‌گیرد. در صورت وجود دو برابر (یا) پرچم یک می‌شود و در غیر اینصورت صفر می‌ماند.

1) Status Register

2) Flags Register

odd parity bit

Parity Flag (PF) : مصرف اصلی آن در هنگام انتقال اطلاعات می‌باشد
هنگامی که هشت بیت پایین‌تر در عملیات داده‌ای تعداد زوجی از یک‌ها داشته باشد
این بیت یک می‌شود و تعدادی فرد از این بیت‌ها باعث می‌شود که این بیت صفر گردد.

Auxiliary Carry Flag (AF) : برای عملیات محاسباتی در BCD بکار می‌رود.
این بیت نشان می‌دهد که آیا ده برابری بر روی کم ارزش‌ترین رقم BCD
آن عدد رخ داده است یا نه؟

Zero Flag (ZF) : وقتی که نتیجه عملیات صفر می‌شود این بیت
یک می‌گردد.

Sign Flag (SF) : وقتی که نتیجه منفی باشد یک می‌شود و
برای نتیجه مثبت صفر می‌گردد.

Overflow Flag (OF) : بیانگر آن است که آیا عملیات منجر به
تولید overflow در بالاترین بیت نتیجه شده است یا خیر.
Overflow وقتی ایجاد می‌شود که یک carry in به آخرین بیت نتیجه
اضافه شود ولی carry out از آن خارج نگردد و یا carry out از
آخرین بیت (پرارزش‌ترین بیت) خارج گردد بدون آنکه carry in
به آن وارد شده باشد.

Trap Flag (TF) : وقتی که TF را یک کنیم ریزپردازنده را در حالت
تک قدم قرار می‌دهد و عیب‌یابی یک برنامه را امکان‌پذیر می‌سازد.

Interrupt Enable flag (IF) : وقتی IF را یک کنیم امکان می‌دهیم که وقفه‌های
خارجی رخ دهند و با صفر شدن آن وقفه‌های خارجی غیرفعال می‌شوند.

Direction Flag (DF) : جهت عملیات رشته‌ای (String) را
تعیین می‌کند. با صفر کردن DF، SI یا DI بطور خودکار به سمت
جلو افزایش می‌یابند. با یک کردن DF، SI یا DI بطور خودکار
به سمت عقب کاهش می‌یابند.

پرچمهای NT و IOPL پرچمهای دیگری هستند که برای حالت حفاظت شده در CPU بکار برده می‌شوند.

Output Privilege Level flag (IOPL) : برای این بکار می‌رود که تضمین کند که یک دستورالعمل تنها عملیاتی را انجام دهد که اجازه انجام آنها را دارد.

Nested Task flag (NT) : نشان می‌دهد که اجرای کار فعلی در داخل قالب کار دیگری انجام می‌شود. اگر NT را یک کنیم یعنی کار فعلی یک ارتباط درست با کار قبلی دارد.

---

* دستورالعملهای منطقی

این نوع دستورالعملها اعمال منطقی در جبر بول در سیستم اعداد دوتائی را انجام می‌دهند. دستورالعملهای این گروه با یک operand و یا دو operand می‌توانند باشند. انواع این دستورالعملها در زیر به عنوان مثال آورده شده‌اند.

NOT <operand>

OR <destination>, <Source>

AND <destination>, <Source>

XOR <destination>, <Source>

---

* دستورالعملهای کار با بیتها :

این نوع دستورالعملها که پس از معرفی 80386 وارد مجموعه دستورالعملها پردازشگرهای Intel شده‌اند، قادرند که تک تک بیتها operand را آدرس دهی کنند.

از جمله انواع این نوع دستورالعملها نوعی است که قادر می‌باشد که بیتهای موجود در operand را در پی 0 یا 1 جستجو نماید. چند نمونه از این نوع دستورالعملها در زیر شرح داده شده‌اند.

BSF/BSR (Bit Scan forward/Reverse) <des.>, <Source>

بیتهای موجود در source را در پی مقدار یک منطقی جستجو می‌کند و در اولین برخورد با 1، شماره بیت آن محل را گزارش می‌کند.

TEST <destination>, <Source> (logical Compare)

دو operand را بیت به بیت با هم AND می‌کند و اثر آن فقط بر روی پرچمها نمایان می‌شود (SF, ZF, PF)

BT / BTS / BTR / BTC (Bit Test / Set / Reset / Complement) <operand>, Selector

در این گروه از دستورالعملها، دستورالعملی نیز وجود دارد که بیش از یک عمل ساده بر روی بیتها انجام می‌دهند. بعنوان مثال در دستور IBTS که به فرم کلی

IBTS base, offset, length, source

IBTS reg/mem, Ax, CL, reg (Insert Bit String)

وجود دارد، بدون اعمال هیچ تغییری در بیتهای منبع، تعدادی از بیتهای منبع را به مجموعهٔ بیتهای مقصد در محل مورد نظر و بطول معین اضافه می‌کند. این دستورالعمل قادر است که بیتهای کمتر از شش بیت یک ثبات را داخل ثبات یا مکان حافظهٔ دیگری کپی کند. دستورالعمل IBTS دارای چهار operand می‌باشد، اولی آدرس پایهٔ رشتهٔ بیتی است (مقصد)، فاصلهٔ بیتی شروع زیر رشته که باید اضافه شود، طول آن زیر رشته بیتی و ثباتی که بیتها باید از آن برداشته شوند. نوع دیگر XBTS است که برای جدا سازی تعدادی از بیتها بکار می‌رود. (Extract bit string)

Discontinued after 80386

* دستورالعملهای حسابی

این نوع دستورالعملها که برای انجام محاسبات ریاضی مورد استفاده قرار می‌گیرند عموماً شامل چهار عمل اصلی می‌شوند. تعدادی از این دستورالعملها در زیر مثال زده شده‌اند.

INC <operand> مقدار operand را یکی افزایش می‌دهد

DEC <operand> " " " کاهش "

ADD <destination>, <Source> دو عدد را با هم جمع می‌کند

ADC <destination>, <Source> دو عدد را با در نظر گرفتن بیت Carry با هم جمع می‌کند.

SUB <destination>, <Source> destination ← destination − Source

SBB <destination>, <Source> destination ← destination − Source − CF

NEG <destination> Two's Complement negation

MUL <source> در نوع بایتی source در ثبات AL ضرب شده و نتیجه در AX ذخیره می‌شود

Unsigned Multiplication در نوع کلمه‌ای source در ثبات AX ضرب شده و نتیجه در DX:AX ذخیره می‌شود.

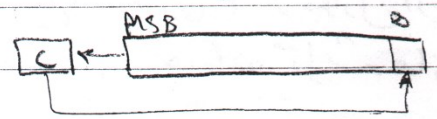
DIV <source> (unsigned division) در نوع بایتی AX/Source می‌شود و خارج قسمت در AL و باقیمانده در AH قرار می‌گیرد

در نوع کلمه‌ای DX:AX/Source می‌شود و خارج قسمت در AX و باقیمانده در DX قرار می‌گیرد

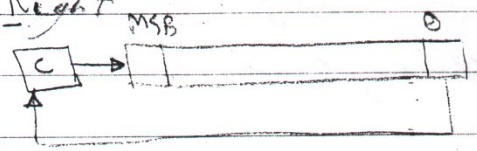
زبان اسمبلی

RCL/RCR/ROL/ROR <destination, {CL / Count} {Rotate / Shift} دستورات

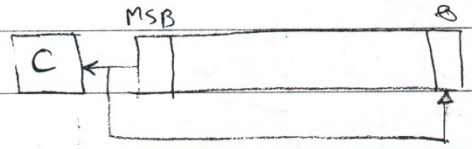
RCL Rotate through carry left



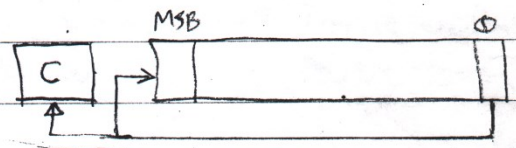
RCR Rotate through Carry to Right



ROL Rotate Left



ROR Rotate Right

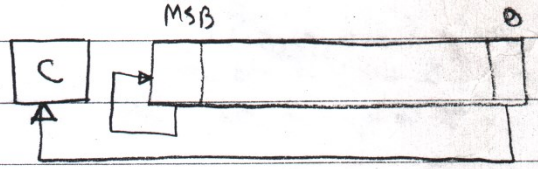


SAL/SAR/SHL/SHR <destination>, {CL / Count}

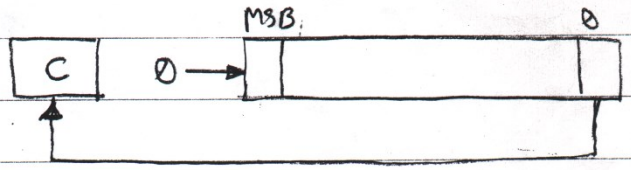
SAL Shift Arithmatic Left

SHL Shift Left

SAR Shift Arithmatic Right



SHR Shift Right



RCL AH, 1 immediate Address جدید

RCR Memory-byte, Rep_Val

ROL AX, CL

SHL Memory-byte, Rep_Val

SAR DX, CL

SAR AH, 1

SHL Table[BX][DI], CL

روش استخراج کد ماشین از زبان اسمبلی

جهت انجام دستی کار اسمبلر و تولید کد زبان ماشین (کد رقمی) از جداول ارائه شده توسط سازندهٔ تراشهٔ ریزپردازنده استفاده می‌شود. در خانواده تراشه‌های 80x86 کد عملیاتی (Operation Code) یک یا چند بایتی بوده و سپس جهت تعیین مُد آدرس‌دهی از یک بایت مد آدرس‌دهی استفاده می‌گردد.

Addressing Mode Byte

mod		reg			r/m		
x	x	y	y	y	z	z	z

xx — از این دو بایت جهت تفکیک آدرس‌دهی حافظه و رجیستر استفاده می‌شود. در حالت آدرس‌دهی حافظه هم تعیین می‌کند که آدرس‌دهی نسبی تا قطعه است.

00 آدرس‌دهی حافظه r/m حالت آن را تعیین می‌کند. بایت تغییر مکان وجود ندارد.

01 آدرس‌دهی حافظه r/m حالت را تعیین می‌کند. بایت تغییر مکان علامت‌دار در ادامه می‌آید (+127 -128)

mod reg r/m	disp byte

10 آدرس‌دهی حافظه r/m حالت را تعیین می‌کند و دو بایت تغییر مکان قطعی در ادامه می‌آید (بدون علامت)

mod reg r/m	disp low	disp high

11 آدرس‌دهی ثبات که بیت w مشخص می‌کند آیا ثبات 8 یا 16 بیتی است (w بیت صفر Op-Code است.)

از روی جدول ثباتهای مورد استفاده را انتخاب می‌کند

از روی جدول مُدهای آدرس‌دهی مختلف را تعیین می‌کند

مثال *

ADD DX, 0F0FH

81 C2 0F 0F

زمان تاشیر

CMP <destination>, <Source> (Compare Two operands)

destination - Source

CMC (complement Carry flag) CF ← $\overline{CF}$

CLC (clear carry flag) CF ← 0

* دستورالعملهای رشته‌های کاراکتری

این دستورالعملها مخصوص انجام عملیات بر روی رشته‌های کاراکتری می‌باشد و بطور مثال جهت کپی کردن تعداد بایت از یک محل به محل دیگر و یا جستجو جهت یافتن بایتی در یک رشته از کاراکترها می‌توان از این دستورالعملها استفاده نمود. از متداولترین دستورالعملها مورد استفاده در برنامه نویسی‌ها دستور MOV می‌باشد.

MOV <destination>, <Source> (Move a byte or a word)

حال نوع کاراکتر آن یک رشته را بر روی رشته‌ای دیگر کپی می‌کند.

MOVS / MOVSB / MOVSW (Move byte or word string)

ES:[DI] DS:[SI]

MOVS <destination-string>, <Source-string>

MOVSB (No operands)

MOVSW (No Operands)

+ اگر پرچم DF (Direction flag) صفر باشد مقدار رجیسترهای SI و DI اضافه می‌شود و اگر یک باشد کم می‌شود.

- دستورات مربوط به جستجو در یک رشته کاراکتری

SCAS / SCASB / SCASW (Compare string data)

مقدار AL (برای بایتی) و AX (برای کلمه) را با ES:[DI] مقایسه می‌نماید.

جهت حرکت با روش بالا و بکمک DF تعیین می‌گردد.

از دستورات اخیر

LODS/LODSB/LODSW (load byte or word string) ← DS:[SI]

STOS/STOSB/STOSW (Store byte or word string) → ES:[DI]

CMPS/CMPSB/CMPSW AL, AX

update SI, DI

REPE/REPZ : Repeat if Equal
REPNE, REPNZ: Repeat if Not Equal

* دستور تکرار در مجموعه دستورات مربوط به رشته‌های کاراکتری

این دستور بصورت مکملی بر دستورات رشته‌ها می‌باشد.

REP/REPZ/REPE/REPNE/REPNZ (Repeat String Operation)

این دستورالعمل تا زمانیکه CX مساوی صفر نشده است دستورات رشته‌ای (STOS, LODS, MOVS) را تکرار می‌کند. برای دستورات (CMPS و SCAS) علاوه بر شرط CX≠0، برای REPZ و یا REPE در صورت ZF=0 یا برابر نبودن حلقه متوقف می‌شود و برای REPNZ یا REPNE در صورت ZF=1 یا برابر بودن حلقه متوقف می‌شود

مثال:

```
LEA  DI, String1
LEA  SI, String2
REPE CMPSB
```

---

* دستورات دیگر:

XCHG <dest>, <source> (reg, mem/reg) — dest ↔ source (جابجایی / swap)

NOP No Operation → XCHG reg : Exchange reg with Accumul. (تبادل خاص)

PUSH/PUSHF/PUSHA SP←SP-2, [SP]←value

POP/POPF/POPA value←[SP], SP←SP+2

LEA (Load Effective Address)

STD (set direction flag (DF)), CLD (Clear direction flag)

XLAT AL ← خروجی ، AL=index ، DS:BX ← محل جدول

LAHF (load AH from flags), SAHF (Store AH in flags)

---

* استفاده از روال:

برای استفاده از روال باید حتماً stack آماده داشته باشیم.

* Call (call a procedure)

توسط این دستورالعمل یک روال یا زیربرنامه را می‌توان اجرا نمود.

* RET (Return from a Procedure)

این دستور جهت بازگشت از یک روال به محل برنامهٔ صدا زننده بکار می‌رود.

Call ها بر دو نوع هستند NEAR که باید در یک قسمت (Segment) از حافظه واقع باشند. FAR که می‌تواند در قسمت (Segment) متفاوتی از حافظه قرار داشته باشد. اجرای دستورالعمل Call باعث می‌شود که آدرس دستورالعمل بعد بر روی stack قرار داده شود و اجرای برنامه به آدرس مورد نظر

انتقال یابد. در RET برعکس این عمل انجام می‌شود و آدرسی که باید به آن انتقال یابد از stack بیرون کشیده شده و اجرای برنامه از آن آدرس دنبال می‌شود.

Call:

SP ← SP - 2
[SP] ← CS
SP ← SP - 2
[SP] ← IP

RET:

IP ← [SP]
SP ← SP + 2
CS ← [SP]
SP ← SP + 2

* روال‌های بازگشتی Recursive Procedures

می‌توان در داخل یک روال باز هم همان روال را صدا زد که در این صورت روال خودش را دوباره اجرا می‌کند. این کار باعث می‌شود که حجم زیادی از stack اشغال شده و یا پرشدن آن سیستم از کار بیفتد. به همین خاطر تعداد دفعات صدا زدن محدود می‌باشد. در ضمن این کار خود موجب کندی در اجرای برنامه نیز می‌شود. توجه داشته باشید که همه روال‌های بازگشتی قابل پیاده‌سازی با حلقه نیز می‌باشند که در بالا سریعتر است. باید توجه داشت که اولاً بازگشت از روال فراموش نشود! ثانیاً باید شرط یا شمارنده‌ای در روال تعداد این صدا زدن‌ها را کنترل نماید.

```
Proc1 Proc   NEAR
        CMP  CL, 00
        JC   go-out
        dec  CL
        call Proc1
go-out: ret
Proc1 ENDP
```

```
Main-Proc Proc NEAR
        Mov CL, 5
        Call Proc1
        ret
Main-Proc ENDP
```

* روش انتقال آرگومان به یک روال

انتقال آرگومان به یک روال سه روش وجود دارد.

۱- بعد ثبات: در این روش آرگومان در یک ثبات قرار داده می‌شود و سپس روال صدا زده می‌شود. پاسخ نیز به همین صورت قابل بازگشت می‌باشد.

۲- بلوک حافظه : در این روش مقدار آرگومان در آدرسی خاص از حافظه کپی می‌شود و پس از اجرای روال، روال آن را برداشته و مورد استفاده قرار می‌دهد. در هنگام بازگشت نیز به همین صورت می‌توان پاسخ را برگردانید.

۳- بلوک Stack : در این روش پیش از صدا زدن روال، آرگومان‌ها را در بخشی stack قرار می‌دهیم. سپس روال صدا زده شده بلوک شامل BP این آرگومان‌ها را مورد استفاده قرار می‌دهد. مثال

```
MainProc PROC NEAR
        Push AX
        Push Bx
        Call Proc1
        Pop Bx
        Pop AX
        RET
MainProc ENDP
```

```
Proc1 PROC NEAR
        push BP
        MOV BP, SP
        Push Cx
        Push Dx
        mov CX, [BP+6]
        mov DX, [BP+4]
        Do something
        mov [BP+6], CX
        mov [BP+4], DX
        POP DX
        POP CX
        POP BP
        ret
Proc1 ENDP
```

SP		
SP-1	AH	← BP+6
SP-2	AL	
SP-3	BH	← BP+4
SP-4	BL	
SP-5	IP(H)	
SP-6	IP(L)	
SP-7	BP(H)	
SP-8	BP(L)	← BP←SP
SP-9		

SP 0100		
0FF	AH	
0FE	AL	
0FD	BH	
0FC	BL	
0FB	IP(H)	
0FA	IP(L)	
0F9	BP(H)	
0F8	BP(L)	← BP←SP
0F7		

Push AX SP←SP-1
Ax←[SP]

POP AX Ax←[SP]
SP←SP+1

* فضای آدرس دهی IO

در بسیاری از ریزپردازنده‌ها فضایی جدا از فضای حافظه جهت آدرس دهی پورتهای ورودی و خروجی در سیستم در نظر گرفته شده است. این فضا بلوک پایه‌ای از پایه‌های ریزپردازنده به نام IO/M از فضای حافظه جدا می‌شود. دستورالعملهایی که با آن می‌توان داده‌ها حافظه در فضای IO نوشت و یا از آنها خواند با دستورات مربوط به حافظه متفاوت می‌باشند. مهمترین آنها دستورهای IN و OUT می‌باشند که بترتیب جهت خواندن از و نوشتن بر روی پورتهای IO از آنها استفاده می‌شود. بعضی از انواع این دستورالعمل نیز وجود دارند که قابلیت خواندن و

OUT port, accumulator
IN accumulator, port

Accumulator : AL یا AX
Port : Immediate یا DX

نوشتن رشته‌ها از/به پورت‌های ورودی و خروجی را دارا هستند

OUTS / OUTSB / OUTSW (Output string to port)

INS / INSB / INSW (Input from port to string)

آدرس پورت در DX قرار می‌گیرد و خانه‌های حافظه با ES:[DI] مشخص می‌شود. بعد از هر بار اجرا DI بطور خودکار افزایش (DF=0) و یا کاهش (DF=1) می‌یابد.

دستورات IO

{IN acc., Port
{IN acc., DX

{IN AX, 06H
{IN AL, 06H

{IN AX, DX
{IN AL, DX

همان برای دستورات IO

{OUT Port, acc
{OUT DX, acc

{OUT 05H, AX
{OUT 05H, AL

{OUT DX, AX
{OUT DX, AL

---

* دستور INT

این دستور در اجرای برنامه وقفه ایجاد نموده و کنترل برنامه را به یکی از ۲۵۶ (بردار) آدرس وقفه انتقال می‌دهد. اجرای دستور INT α بگونهٔ زیر می‌باشد.

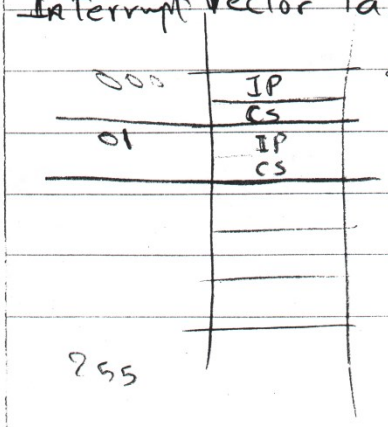
۱) SP ← SP-2 ، [SP] ← flags ، IF ← 0 ، TF ← 0

۲) SP ← SP-2 ، [SP] ← CS ، CS ← IVT[4×α+2]

۳) SP ← SP-2 ، [SP] ← IP ، IP ← IVT[4×α]

دستور: IRET

Interrupt Vector Table (IVT)



Interrupt Vector Table

Interrupt Service Routine

خانهٔ وقفهٔ بعد از اجرای دستور INT اجرای

شمارهٔ وقفه × 4 = IP برای ISR

* دستور مجازی Pseudo - Operation

این دستورالعملها جهت استفاده بوسیلهٔ اسمبلر بوده و توسط آنها برنامه نویس دستوراتی را به برنامهٔ اسمبلر ابلاغ می نماید. این دستورات موجب تولید هیچ کد ماشینی نمی شوند.

* دستورات مجازی جهت لیست گیری Listing Pseudo Operations

* PAGE توسط این دستورالعمل مجازی ابعاد صفحه‌های لیست برنامه در خروجی برنامهٔ اسمبلر تعیین می گردد. بطور مثال PAGE 60,132 یعنی 60 خط در هر صفحه و 132 کاراکتر در هر خط

PAGE <linesperpage>, <char. per line>

* TITLE توسط این دستورالعمل مجازی عنوان برنامه را در ابتدای هر صفحه معین می نماید.

TITLE <Text>

مثال: TITLE This is the title of the Program

* SEGMENT

نام برنامه‌های زبان اسمبلی حداقل از یک قسمت یا سگمنت تشکیل شده‌اند. یک سگمنت بصورت زیر تعریف می شود.

Name	Operation	Operand
name	SEGMENT	[options]
	⋮	
name	ENDS	

ENDS پایان سگمنت را نشان می دهد و هم نام خود دستور SEGMENT می باشد.

Options برای یک سگمنت عبارت است از سه نوع زیر که هر سه همزمان هم قابل استفاده می‌باشند:

* Alignment type : این ورودی محدوده شروع سگمنت را نشان می‌دهد در صورت استفاده از PARA این محدوده یک پاراگراف یا آدرسی بخش پذیر بر ۱۶ می‌باشد یعنی nnnnH . در صورت خالی بودن این قسمت، اسمبلر بصورت معمول PARA را برخواهد گزید.

* Combine type : این ورودی مشخص کننده آن است که این سگمنت را در زمان link کردن می‌توان با بقیه سگمنت‌ها ترکیب نمود یا نه. انواع این سگمنتها عبارتند از STACK, COMMON, PUBLIC, AT, MEMORY

بعنوان مثال یک سگمنت STACK را می‌توان بصورت زیر تعریف نمود:

```
name SEGMENT PARA STACK
{
name ENDS
```

در مواقعی که برنامه‌ها بصورت جداگانه نوشته شده و اسمبل شده باشند می‌توان از PUBLIC, COMMON, MEMORY استفاده نمود که در هنگام link کردن آنها این تعاریف اعمال خواهند شد در غیر اینصورت می‌توان از ذکر آنها صرفنظر نمود.

* Class type : این مقدار در داخل دو آپاستروف جهت گروه‌بندی سگمنتهای مرتبط باهم بکار می‌رود که در هنگام عملیات link کردن مورد استفاده قرار می‌گیرد. مثال:

```
name SEGMENT PARA STACK 'stack'
{
name SEGMENT
```

* دستور مجازی PROC

قسمت کد برنامه در حال اجرا را در خود جای می دهد. به همین خاطر این سگمنت حاوی روالها یا procedure هائی می باشد که بکمک دستور مجازی PROC تعریف می شوند. بعنوان مثال یک سگمنت با تنها یک PROC بصورت زیر نوشته می شود:

```
Segname  SEGMENT  PARA
Procname PROC     FAR
         |
         RET
Procname ENDP
Segname  ENDS
```

دستور مجازی ENDP پایان روال را نشان می دهد.

* دستور مجازی ASSUME

این دستور جهت نسبت دادن ثباتهای قسمت متفاوت (ES, DS, SS, CS) به سگمنت های تعریف شده در برنامه زبان اسمبلی بکار می رود. بعنوان مثال:

Operation	Operand
ASSUME	SS: stackname, DS: datasegname, CS: codesegname

در صورتیکه از یکی از ثباتهای قسمت استفاده نشود مقدار آن را می توان بصورت زیر حذف کرد: مثلاً برای ES می توان نوشت ES:NOTHING

* دستور العمل مجازی END

این دستور العمل مجازی بصورت زیر می باشد

Operation	Operand
END	[procname]

در برنامه هائی که تنها شامل یک روال هستند operand نام آن روال

که اغلب هم از نوع FAR است. در صورتیکه چند روال موجود باشد، نام روالی در operand قرار داده می‌شود که روال اصلی است. این END انتهای تمام برنامه را در آن فایل زبان اسمبلی مشخص می‌کند.

Program Segment Prefix (PSP) : پیشوند قسمت برنامه

در ابتدای هر برنامه اجرائی یک ناحیه به اندازه ۲۵۶ بایت (100 Hex) وجود دارد که به آن پیشوند قسمت برنامه (PSP) گویند. برنامهٔ بارکنندهٔ سیستم عامل (DOS loader) از ثبات قسمت DS جهت تعیین نقطهٔ شروع PSP استفاده می‌نماید. برنامهٔ اجرائی باید این آدرس را بکمک مشخص کردن آن بر روی پشته ذخیره سازد. در انتها دستور RET از این آدرس جهت بازگشت به DOS استفاده می‌نماید.

سیستم نیازمند آن است که مقدار بعدی بر روی پشته یک آدرس صفر (بعد دقیق از یک قسمت صفر zero offset) باشد. جهت انجام این کار بکمک دستور SUB AX را از AX کم می‌کنیم تا نتیجه صفر شود. آنگاه AX را بر روی پشته می‌فشاریم. از آن جهت که برنامهٔ بارکننده (loader) از ثبات قسمت DS برای منظور نگاری استفاده می‌کند نیاز است که آن را بر روی پشته فشار دهیم و سپس مقدار جدید را بکمک دستور MOV جایگزین آن سازیم.

در انتها دستور RET با استفاده از آدرسی که در ابتدای برنامه بر روی پشته فشرده شده است (PUSH DS) ادامهٔ اجرا را به سیستم عامل بازمی‌گرداند. یک روش خروج معمول دیگر مورد استفاده قرار گرفته استفاده از INT 20H می‌باشد.

```
CODESG  SEGMENT PARA 'CODE'
BEGIN   PROC FAR
        ASSUME CS:CODESG,DS:DATASG,SS:STACKSG
        PUSH DS             ;Store DS on stack
        SUB AX,AX           ;Set AX to zero
        PUSH AX             ;Store zero on stack
        MOV AX,DATASG       ;Store address of
        MOV DS,AX           ;DATASG in DS
        |
        |
        RET                 ;Return to DOS
BEGIN   ENDP
CODESG  ENDS
        END BEGIN           ;End of program
```

نمونه کامل از ساختار یک برنامه EXE.

```
        PAGE 60,132
TITLE EXAM1 (EXE) Example code structure for .EXE files
;-----------------------------------------------------
STACKSG SEGMENT PARA STACK 'Stack'
        DB      12 DUP ('STACKSEG')  ;we could say DB 12 Dup(?)
STACKSG ENDS
;-----------------------------------------------------
DATASG  SEGMENT PARA 'Data'
Data1   DB 10DUP(?)
Param1  DW ?
DATASG  ENDS
;-----------------------------------------------------
CODESG  SEGMENT PARA 'Code'
BEGIN   PROC    FAR
        ASSUME SS:STACKSG, CS:CODESG, DS:DATASG, ES:NOTHING
        PUSH DS                 ;Push DS onto the stack
        SUB AX,AX               ;Push zero offset onto the stack
        PUSH AX
        |
        |   } Do something
        |
        RET
BEGIN   ENDP
CODESG  ENDS
        END BEGIN               ;End of Program
```

* توجه داشته باشید که در انتهای برنامه شبیه دستور (دستور مجازی) END با نام BEGIN در مقابل آن پایان برنامه را نشان می‌دهد و در عین حال گویای آن است که شروع برنامه از نقطه BEGIN می‌باشد.

* ساختار برنامهٔ COM.

در این ساختار قسمت پشته تعریف نمی‌شود و به همین خاطر اسمبلر در هنگام تولید فایل obj. با پیغام اخطار این موضوع را گوشزد می‌کند که فقط جهت اطلاع می‌باشد. در حالت عادی پس از اجرای دستور link فایل EXE. حاصل می‌شود که آن را بکمک برنامهٔ EXE2BIN به فایل COM. تبدیل می‌کنیم. ولی در اسمبلرها و لینکرهای جدید این قسمت از بین رفته و خود آن برنامه‌ها مستقیماً فایل COM. را تولید می‌کنند (منظور TASM می‌باشد)

در ساختار برنامهٔ COM. تمام برنامه در یک سگمنت خلاصه می‌شود یعنی DS، CS و ES همه همان CS هستند. پشته توسط سیستم عامل آماده سازی می‌گردد که این کار بطور خودکار انجام می‌شود. از آنجا که DS همان CS است و هر برنامه‌ای PSP دارد در نتیجه ۲۵۶ بایت اول CS در برنامه‌های COM. همان PSP برنامه خواهند بود و کد اصلی باید از بعد از این قسمت آغاز گردد. جهت همین موضوع از دستور مجازی ORG 100H در ابتدای برنامه استفاده می‌شود. برای تعریف داده‌ها و متغیرها هم در ابتدای کد برنامه قسمتی به این کار تخصیص داده می‌شود و با یک دستور پرش اجرای برنامه از روی آن به جلوتر منتقل می‌شود.

* پشتهٔ تولید شده توسط سیستم عامل برای برنامه‌های COM.

در صورتیکه 64K برای برنامه کافی باشد DOS ثبات SP را به آدرسی در انتهای سگمنت (FFFEH) اشاره می‌دهد (بالای پشته) و در صورتیکه فضای کافی وجود نداشته باشد، DOS یک پشته در آدرسهای انتهایی حافظه برای برنامه ایجاد می‌کند. در انتهای هر دو این دو حالت DOS یک کلمه (دو بایت) صفر را بر روی پشته فشار می‌دهد.

یک مثال از ساختار فایل COM :

```
        PAGE 60,132
TITLE   EXAMCOM1 Com Program to Move and add
CODESG  SEGMENT PARA 'Code'
        ASSUME CS:CODESG,DS:CODESG,SS:CODESG,ES:CODESG
        ORG 100H                ;Start at the end of PSP
BEGIN:  JMP Main                ;Jump to pass the data
;-----------------------------------------------------
FLDA    DW      250             ;Data Definitions
FLDB    DW      125
FLDC    DW      ?
;-----------------------------------------------------
Main    PROC    NEAR
        MOV AX,FLDA             ;Move 0250H to AX
        ADD AX,FLDB             ;Add 0125H to AX
        MOV FLDC,AX             ;Store Sum in FLDC
        RET                     ;Return to DOS
Main    ENDP
CODESG  ENDS
        END BEGIN
```

* در محیط debug کردن ابتدا برنامه برای برنامه‌های EXE در CS:0000H می‌باشد ولی برای برنامه‌های COM در CS:0100H می‌باشد.

مثال: تمام کاراکترها را از کوچک به بزرگ تبدیل کنید.

```
TITLE CASE (COM) Change lowercase to uppercase
CODESG SEGMENT PARA 'CODE'
        ASSUME CS:CODESG,DS:CODESG,SS:CODESG
        ORG 100H
BEGIN:  JMP MAIN
TITLEX  DB 'Change to uppercase letters'
;-----------------------------------------
MAIN    PROC NEAR
        LEA BX,TITLEX          ; 1st char to change
        MOV CX,27D             ; No. chars to change
B20:
        MOV AH,[BX]            ; Char from TITLEX
        CMP AH,61H             ; Is it
        JB B30                 ; lower (jump if below)
        CMP AH,7AH             ;  case
        JA B30                 ;  letter? (jump if above)
        AND AH,11011111B       ; Yes - Convert
        MOV [BX],AH            ; Restore in TITLEX
B30:
        INC BX                 ; Set for next char
        LOOP B20               ; Loop 31 times
        RET
MAIN    ENDP
CODESG  ENDS
        END BEGIN
```

مثال: استفاده از دستورات رشته‌های کاراکتری

```
                Page 60,132
TITLE   STRING (EXE) Tests of string operations
;-----------------------------------------------
STACKSG SEGMENT PARA STACK 'Stack'
        DW      32 DUP(?)
STACKSG ENDS
;-----------------------------------------------
DATASG  SEGMENT PARA 'Data'
NAME1   DB      'Assemblers'            ;Data items
NAME2   DB      10 DUP(' ')
NAME3   DB      10 DUP(' ')
DATASG  ENDS
;-----------------------------------------------
CODESG  SEGMENT PARA 'Code'
BEGIN   PROC    FAR                     ;Main Procedure
        ASSUME CS:CODESG,DS:DATASG,SS:STACKSG,ES:EXTRASG
        PUSH DS
        SUB AX,AX
        PUSH AX
        MOV AX,DATASG
        MOV DS,AX
        MOV ES,AX
        CALL C10MVSB                    ;MVSB Subroutine
        CALL D10MVSW                    ;MVSW Subroutine
        CALL E10LODS                    ;LODS Subroutine
        CALL F10STOS                    ;STOS Subroutine
        CALL G10CMPS                    ;CMPS Subroutine
        CALL H10SCAS                    ;SCAS Subroutine
        RET
BEGIN   ENDP
;       Use of MOVSB:
;       -------------
C10MVSB PROC    NEAR
        CLD
        LEA     SI,NAME1
        LEA     DI,NAME2
        MOV     CX,10                   ;Move 10 bytes,
        REP     MOVSB                   ;NAME1 to NAME2
        RET
C10MVSB ENDP
;       Use of MOVSW:
;       -------------
D10MVSW PROC    NEAR
        CLD
        LEA     SI,NAME2
        LEA     DI,NAME3
        MOV     CX,05                   ;Move 5 words
        REP     MOVSW                   ;NAME2 to NAME3
        RET
D10MVSW ENDP
;       Use of LODSW:
;       -------------
E10LODS PROC    NEAR
        CLD
        LEA     SI,NAME1                ;Load 1st word of NAME1
        LODSW                           ;  into AX reg.
        RET
E10LODS ENDP
```

```
;                    Use of STOSW:
;
F10STOS     PROC    NEAR
            CLD
            LEA     DI,NAME3
            MOV     CX,05
            MOV     AX,2020H        ;Move blanks
            REP     STOSW           ; to NAME3
            RET
F10STOS     ENDP
;            Use of CMPSB:
;
G10CMPS     PROC    NEAR
            CLD
            MOV     CX,10
            LEA     SI,NAME1
            LEA     DI,NAME2
            REPE    CMPSB           ; Compare NAME1:NAME2
            JNE     G20             ;Not equal?
            MOV     BH,01
G20:        MOV     CX,10
            LEA     SI,NAME2
            LEA     DI,NAME3
            REPE    CMPSB           ;Compare NAME2:NAME3
            JE      G30             ; If equal, exit
            MOV     BL,02
G30:        RET
G10CMPS     ENDP
;           Use of SCASB:
;
H10SCAS     PROC    NEAR
            CLD
            MOV     CX,10
            LEA     DI,NAME1
            MOV     AL,'m'          ;Scan NAME1
            REPNE   SCASB           ; for 'm'
            JNE     H20             ; if not found, exit
            MOV     AH,03
H20:        RET
H10SCAS     ENDP

CODESG      ENDS
            END     BEGIN
```

ASCII ضرب دو عدد

```
TITLE ASCMUL (COM) Multiply ASCII Numbers
CODESG SEGMENT PARA 'Code'
        ASSUME CS:CODESG,DS:CODESG,SS:CODESG, ES: CODESG
        ORG 100H
BEGIN: JMP MAIN
;-----------------------------------------------
MULTCND DB  '3783'          ;Data items
MULTPLR DB  '5'
PRODUCT DB  5DUP(0)
;-----------------------------------------------
MAIN    PROC  NEAR
        MOV CX,04           ; initialize 4 loops
        LEA SI,MULTCND+3
        LEA DI,PRODUCT+4
        AND MULTPLR,0FH     ;clear ASCII 3
A20:
        MOV AL,[SI]         ;load ASCII char (or LODSB)
        AND AL,0FH          ;clear ASCII 3
        MUL MULTPLR         ;Multiply
        AAM                 ;ASCII Adjust for Multiplication
        ADD AL,[DI]         ;Add to
        AAA                 ;  stored  ;ASCII Adjust for Addition
        MOV [DI],AL         ;  product
        DEC DI
        MOV [DI],AH         ; Store product carry
        DEC SI
        LOOP A20            ;loop 4 times
        RET
MAIN ENDP
CODESG ENDS
      END BEGIN
```

* محتویات Program Segment Prefix (PSP)

Address	Data
PSP:00H	ماشین کد دستور INT20H (CP/M)
PSP:02H	اندازهٔ باقیماندهٔ حافظه برحسب پاراگراف (XXXX0H)
PSP:05H	آدرس هاهنگ کننده توابع DOS (CP/M)
PSP:0AH	آدرس برنامهٔ مربوط به وقفه INT22H
PSP:0EH	آدرس برنامهٔ مربوط به وقفه INT23H (C+Cntl) Int.
PSP:12H	آدرس برنامهٔ مربوط به وقفه INT24H (Critical Int.)
PSP:2CH	آدرس سگمنت محیط (Environment arguments)
PSP:50H	کد ماشین برای INT21H و Retf
PSP:5CH	اولین FCB باز نشده
PSP:6CH	دومین FCB باز نشده
PSP:80H	تعداد کاراکترها برای محیط پارامترهای بدون ساختار
PSP:81H	محیط پارامترهای بدون ساختار از اینجا تا فاصلهٔ 0FFH ادامه می‌یابد به عنوان بافر برای محیط انتقالی دیسک هم از آن استفاده می‌شود (DTA)

* مثال: اجرای برنامهٔ CALCIT.EXE بدون پارامتر یعنی CALCIT [return] در هنگام ایجاد PSP برای اجرای برنامه، DOS در آدرس 80H عدد 000DH را قرار می‌دهد. این کار برای آن انجام می‌شود که چون کاراکتری بعد از اسم برنامه نیامده (بغیر از CR) پس طول رشته کاراکتری متقابل اسم برنامه صفر بایت می‌باشد و بایت بعدی همان کاراکتر CR (Carriage return) است (0DH)

* مثال: اجرای برنامه با پارامتر در مقابل آن: COLORK BY

در این حالت در آدرس PSP:80H بایتهای زیر قرار می‌گیرند:

PSP:80H : 03 20 42 59 0D

طول کاراکترها — K B Y — CR

ASCII

* مثال : دستوری با پارامترهای نام فایل:

DEL B:CALCIT.OBJ [CR]

PSP : 5C : 02 43 41 4C 43 49 54 20 20 4F 42 4A

FCBA نام درایو B C A L C I T O B J

جهت پیگیری ۸ بایت نام فایل

PSP : 80 : 0D 20 42 3A 43 41 4C 43 49 54 2E 4F 42 4A 0D

طول رشته کاراکتری B : C A L C I T . O B J (CR)

حال اگر در این مثال دستور بصورت زیر بود:

XCOPY A:FILEA, B:FILEB

قسمت مربوط به فایل اول (FILEA) در FCB1 (5CH) و فایل دوم (FILEB) در FCB2 (6CH) قرار می‌گیرد

---

* نکته : در مورد Environment (PSP:2C)

این قسمت شامل آدرس segment Environment برنامه می‌باشد.
این محیط از تعدادی رشته‌های کاراکتری منتهی به صفر (ASCIIZ) تشکیل شده‌اند و در انتهای محیط دو بایت صفر علامت پایان را نشان می‌دهد.
از محتویات این قسمت می‌توان به COMMAND LINE برنامه و آدرس اجرای برنامه اشاره نمود.

⊗ هم پردازشگرهای رقمی The Numerical Coprocessors

به هم پردازنده‌های 8087/80287/80387 معمولاً بعنوان FPU (floating point unit).
این هم پردازنده‌ها هشت ثبات داده دارند که به آنها R0-R7 گویند ولی این ثباتها هیچگاه در دستورالعملهای زبان اسمبلی شرکت نمی‌کنند. عملکرد این سیستم بصورت پشته‌ای می‌باشد سر پشته را st یا st(0) می‌نامند. در ابتدا st، ثبات R7 می‌باشد ولی پس به R6 و R5 و ... تبدیل می‌شود. طرز کار به این ترتیب است که ابتدا operand ها را بر روی پشته قرار می‌دهیم و پس از اجرای دستورالعمل هم بر روی پشته باقی می‌ماند. یعنی S(0)-S(7) قرار دادن یک پارامتر جدید بر روی پشته‌ای پُر و برداشتن پارامتری از پشته خالی بترتیب باعث خطای stack overflow، stack underflow می‌شود.

دستورالعملهای هم پردازشگرها می‌توانند با دو، یک یا بدون operand باشند.

```
inst Reg,Reg
inst Mem
inst
```

در فرم اول هر Reg ای باید یک st(i) باشد و حداقل یکی از آنها باید st=st(0) باشد. اولین Reg مقصد و دومین Reg منبع نامیده می‌شوند.

در فرم دوم Mem یک متغیر حافظه می‌باشد. این متغیر منبع می‌باشد و مقصد بطور ضمنی st می‌باشد (st(0)). نشانگر پشته بدون تغییر باقی می‌ماند.

در نوع سوم که معمولاً به آن بعنوان فرمت ماشین پشته کلاسیک اطلاق می‌شود operand های ضمنی st بعنوان منبع و st(1) بعنوان مقصد مورد استفاده قرار می‌گیرند. در این فرم علاوه بر تغییر دادن st(1) ثبات پشته را بیرون می‌اندازد و در نتیجه st جدید مقصد خواهد بود. البته در ایجاد وضعیت ثبات پشته تغییر می‌یابد.

علاوه بر هشت ثبات بالا پنج ثبات دیگر هم وجود دارند که عبارتند از:

Status Word (کلمهٔ وضعیت)، Control Word (کلمهٔ کنترلی)،
Tag Word (کلمهٔ نشانه)،
Exception Pointers (اشاره‌گرهای استثنا) { اشاره‌گر دستورالعمل (Instruction Pointer)
اشاره‌گر داده (Data Pointer)

همزمانی (Synchronization)

نامگذاری هم پردازنده به دلیل خوبی می باشد چون که ریز پردازنده و هم پردازنده بطور همزمان دستورالعملها را اجرا می کنند. این رفتار می تواند به میزان زیادی به راندمان کار برنامه های هم پردازنده کمک کند ولی در عین حال خطراتی را نیز در پی دارد.

در نسل های اولیه پردازنده ها مثل "8087" ریز پردازنده منتظر پایان اجرا دستورالعمل هم پردازنده نمی شد دستور Fwait چنین وقفه ای را ایجاد می کند ولی در ریز پردازنده های نسل های بعدی (80287/80387) ریز پردازنده بطور خودکار منتظر هم پردازشگر می شود تا دستور را بپایان برساند. هنگامی که دستورات هم پردازشگر مقادیری را در حافظه می نویسند در صورتیکه در فاصله کوتاهی از اجرا دستورالعمل هم پردازشگر، پردازشگر هم بخواهد از آن خانهٔ حافظه استفاده کند برنامه نویس باید اطمینان حاصل کند که دستورالعمل FPU بپایان رسیده باشد. جهت این کار برنامه نویس یا باید از fwait استفاده کند و یا از دستوراتی از FPU که حافظه را پیش از امکان دسترسی CPU به آن نقطه از حافظه تغییر می یابد.

FIGURE 17.10 SOLVING QUADRATIC EQUATIONS

```
/* quad1.c: driver for the quadratic equation solving procedure */

#include <stdio.h>

#define     EQUAL      0
#define     REAL       1
#define     COMPLEX   -1

extern int solveQuad(double, double, double, double*);

void main(void)
{
    double a, b, c;
    double Roots[2];
    int ch;

    do
    {
        printf("\n\nInput coefficients a, b, and c of a quadratic:  \n");
        scanf("%lf %lf %lf", &a, &b, &c);
        switch(solveQuad(a,b,c,Roots))
        {
            case EQUAL:

                printf("\n\nThe equation has a repeated real root %lf.\n",
                    Roots[0]);
                break;

            case REAL:

                printf("\nThe roots are %lf and %lf.", Roots[0], Roots[1]);
                break;

            case COMPLEX:

                printf("\nThe roots are %lf + i%lf and %lf - i%lf.",
                     Roots[0], Roots[1], Roots[0], Roots[1]);
        }
        printf("\n\nDo another? (y or n)  ");
        scanf("%d",&ch);
    }
    while ((ch = getchar()) == 'y' || ch == 'Y');
}
```

```
TITLE  Quad2.asm: the FPU procedure that solves quadratics

COMMENT # Expects the coefficients a, b, and c to be pushed onto the stack
          as doubles, in that order, followed by the address aryAdr of an
        # array Ary of two doubles, used as indicated below.

;values returned by solveQuad in the ax register:
EQUAL       EQU      0    ;equal root case: Ary[0] holds the root
REAL        EQU      1    ;unequal real roots: Ary[0] and Ary[1] hold them
```

```
COMPLEX     EQU     -1      ;complex roots: Ary[0] and Ary[1] hold real and
                            ;  imaginary parts


.MODEL      SMALL, C
.CODE

solveQuad   PROC    USES bx, A:QWORD, B:QWORD, C:QWORD, aryAdr:WORD
                    LOCAL statusWord:WORD

    mov     bx, aryAdr      ;point bx to answer location
    fld1                    ;push 1.0
    fadd    st, st          ;double it
    fld     st              ;duplicate it
    fadd    st(1), st       ;double it again, so that st contains 4
    fld     A               ;push A
    fmul                    ;replace st by 2*A
    fld1
    fdivr                   ;st = 1/(2*A)
    fld     st              ;duplicate st
    fld     B               ;push B
    fchs                    ;st = -B
    fmul    st(1),st        ;st(1) = -B/(2*A)
    fmul    st, st          ;st = B*B
    fld     A               ;push A
    fmul    st, st(4)       ;st = 4*A
    fld     C               ;push C
    fmul                    ;st = 4*A*C
    fsub                    ;st = B*B-4*A*C
    ftst
    fstsw   statusWord      ;to check the result of the ftst
    mov     ax, statusWord
    sahf
    jb      complexCase     ;since the roots will be complex
    ja      realCase        ;there will be two real, unequal roots
    mov     ax, EQUAL       ;set return value for real, equal roots
    fstp    st              ;discard B*B-4*A*C, so that st = -B/2*A
    fstp    QWORD PTR [bx]  ;store st
    fstp    st              ;pop the FPU stack
    jmp     SHORT Done

complexCase:

    mov     ax, COMPLEX     ;set return value for complex roots
    fchs                    ;because st is negative
    fsqrt                   ;st = sqrt(4*A*C - B*B)
    fmulp   st(2), st       ;st = sqrt(4*A*C - B*B)/(2*A)
    fstp    QWORD PTR [bx]  ;store the imaginary part
    fstp    QWORD PTR [bx+8];store the real part
    jmp     SHORT Done

realCase:


    mov     ax, REAL        ;set return value for unequal real roots
    fsqrt                   ;st = sqrt(B*B - 4*A*C)
    fmul    st, st(2)       ;st = sqrt(B*B - 4*A*C)/(2*A)
    fld     st              ;duplicate it
```

Figure 17.10 cont.

```
67      fadd    st,st(2)          ;compute first root
68      fstp    QWORD PTR [bx]    ;  and store it
69      fsubp   st(1), st         ;compute second root
70      fstp    QWORD PTR [bx+8];and store it
71      fstp    st                ;pop the FPU stack
72
73  Done:
74
75      fstp    st                ;finish clearing the FPU stack
76
77      ret
78
79  solveQuad   ENDP
80
81      END
```

Our second example is an implementation of *Newton's Method* for finding a root of an equation. Given a function f(x), Newton's Method attempts to find a number r such that f(r) = 0, by making a series of approximations. The initial approximation X_0 is supplied by the user and, for each $n \geq 1$, new approximations are defined by the formula

$$X_n = X_{n-1} - f(X_n)/f'(X_n)$$

where f'(x) means the derivative of f(x). Using only the most basic ideas from calculus, it is easy to verify that this formula should generate a sequence of "reasonable" approximations to a root r, provided the starting X_0 is not too far from r.

Figure 17.12 contains the code for Newton's Method. It is written in our standard library module form, and in fact has been added to USER.LIB. Here are some comments on NEWTON.ASM:

Lines 63-64. NEWTON.ASM expects that the caller will have pushed two items onto the run-time stack: a near pointer to the derivative f'(x) followed by a near pointer to f(x). This is very easy to implement in assembly language. Lines such as

```
mov     ax, OFFSET funcName
push    ax
```

have the effect of pushing a near pointer to the procedure named *funcName* onto the stack, while an instruction like line 64 of the present program

```
call    [bp+4]
```

executes an "indirect near call" to the procedure whose offset is passed as **[bp+4]** on the stack. **MASM** resolves the anonymous reference by using **WORD PTR**, i.e., as a **NEAR** call, since this is the default when the **SMALL** model is used.

FIGURE 17.11 DIAGRAMMING THE FPU STACK IN QUAD2.ASM

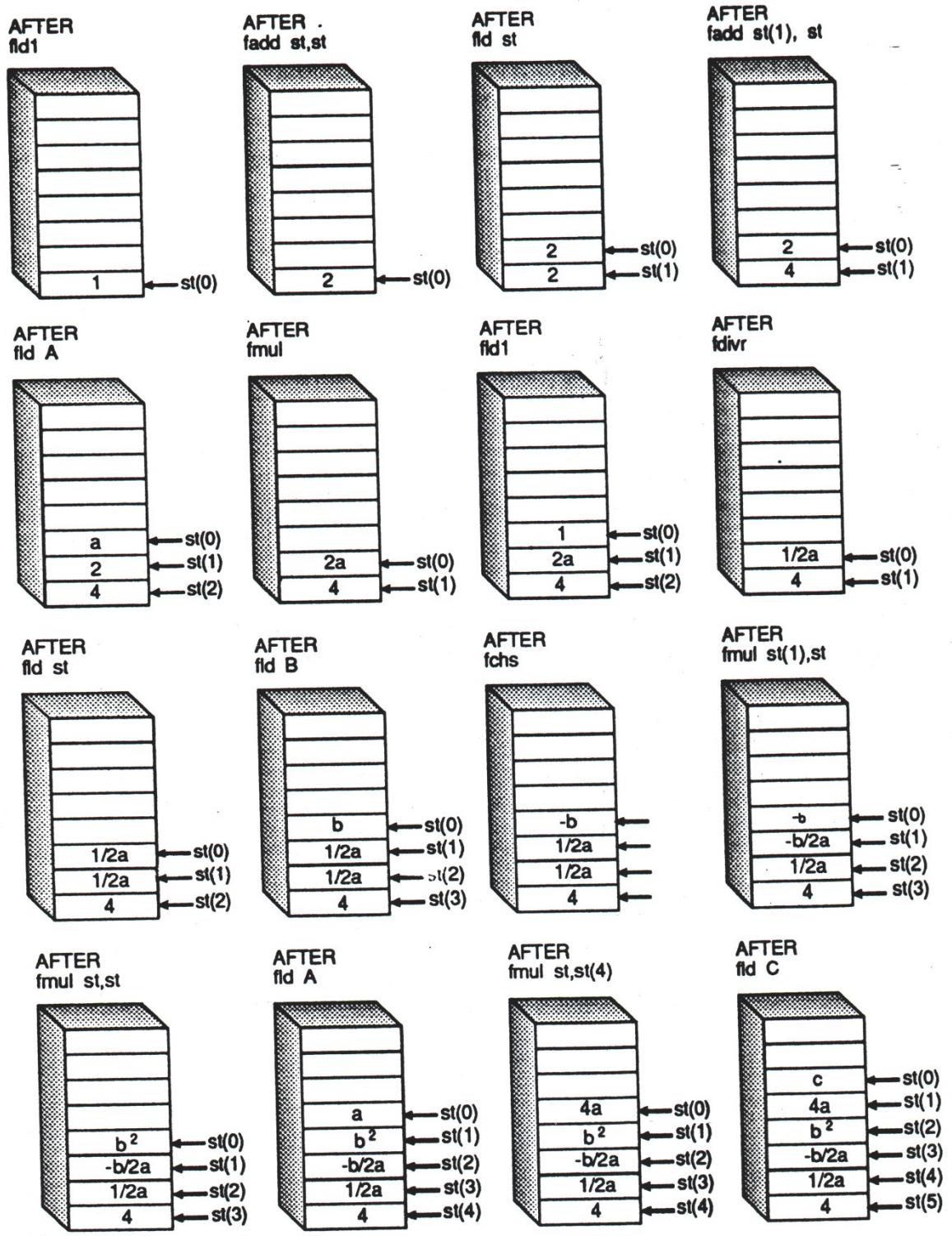


Figure 16.4 cont.

```
_TEXT   SEGMENT   PARA   PUBLIC   'CODE'

    ASSUME cs:_TEXT, ds:NOTHING, es:NOTHING

;Device Driver Header

    deviceHeader    DD      -1
                    DW      ATTRIBUTE
                    DW      Strategy
                    DW      Interrupt
    devNameOrNo     DB      8 DUP(?)

;Jump table: contains spaces for all commands that can be supported by
;different kinds of device drivers. No single driver will support all the
;commands. The comments indicate the kinds of device to which each command
;applies, and the minimal DOS version with which it can be used

    commandTable    DW      Initialize      ;BOTH      -- 2.0
                    DW      mediaCheck      ;BLOCKDEV  -- 2.0
                    DW      buildBPB        ;BLOCKDEV  -- 2.0
                    DW      IOCTLRead       ;BOTH      -- 2.0
                    DW      Read            ;BOTH      -- 2.0
                    DW      nonDestRead     ;CHARDEV   -- 2.0
                    DW      inputStatus     ;CHARDEV   -- 2.0
                    DW      flushInputBuf   ;CHARDEV   -- 2.0
                    DW      Write           ;BOTH      -- 2.0
                    DW      writeVerify     ;BOTH      -- 2.0
                    DW      outputStatus    ;CHARDEV   -- 2.0
                    DW      flushOutputBuf  ;CHARDEV   -- 2.0
                    DW      IOCTLWrite      ;BOTH      -- 2.0
                    DW      Open            ;BOTH      -- 3.0
                    DW      Close           ;BOTH      -- 3.0
                    DW      removMedia      ;BLOCKDEV  -- 3.0
                    DW      outputUntilBusy ;CHARDEV   -- 3.0
                    DW      numberNotUsed
                    DW      numberNotUsed
                    DW      genIOCTLReq     ;BOTH      -- 3.2
                    DW      numberNotUsed
                    DW      numberNotUsed
                    DW      numberNotUsed
                    DW      getLogicalDevice;BLOCKDEV  -- 3.2
                    DW      setLogicalDevice;BLOCKDEV  -- 3.2

    reqHdrAddr      LABEL   DWORD
    reqHdrOff       DW      ?
    reqHdrSeg       DW      ?

;-----------------------------------------------------------

Strategy    PROC    FAR

    mov     reqHdrOff, bx       ;initialize es:bx to point to
    mov     reqHdrSeg, es       ;  the request header
    ret
```

```
112   Strategy    ENDP
113
114   ;-----------------------------------------------------------
115
116   Interrupt   PROC    FAR
117
118       pushRegs ax,bx,cx,dx,si,di,ds,es,bp
119
120       les     di, reqHdrAddr        ;point es:di to the request header
121
122       mov     bl, reqHdr[02h]       ;command code
123       xor     bh, bh
124       cmp     bx, MAXCOMMAND        ;is command code in correct range?
125       jle     goodCommand
126       call    numberNotUsed
127
128   goodCommand:
129
130       shl     bx, 1                 ;index into table of words
131       call    commandTable[bx]
132
133       les     di, reqHdrAddr        ;in case called function changed es or di
134       or      ax, DONE              ;signal completion of command
135       mov    reqHdr[03h], ax        ;record status word in request header
136
137       popRegs
138       ret
139
140   Interrupt ENDP
141
142   ;-----------------------------------------------------------
143
144   numberNotUsed   PROC    NEAR
145
146       mov     ax, ERROR OR UNKNOWNCOMMAND     ;signal error
147       ret
148
149   numberNotUsed   ENDP
150
151   ;===========================================================
152
153   ;DEVICE SPECIFIC CODE COMES HERE
```

Lines 32-53. When the driver has completed a task, it must return a *status code* to MS-DOS. Normally, the DONE bit is set to signal completion and, if an error occurred, the ERROR bit is set, in which case the low 8 bits will also contain an error code, as listed on lines 40-53.

Line 59. The directive

```
ASSUME ds:NOTHING, es:NOTHING
```

deactivates any ASSUMEs that may be in effect for ds and es. In the present context, it has no effect since there are no active ASSUMEs. We have included them only as documentation, to emphasize the fact that we definitely do not want the ASSUME statements

```
ASSUME ds:_TEXT, es:_TEXT
```

to occur in our versions of device drivers. At a number of points in the program, the segment registers are pointed to different regions of memory. Recall from Chapter 6 that if several segment registers have **ASSUME**s associating them with the same segment, then **MASM** gives precedence to **ds** and **es** over **cs** when referencing a named variable in that segment. This can easily lead to addressing errors. In one of the exercises, the reader is asked to rebuild our first example, LASR, incorporating these extra **ASSUME**s, and then to explain the outcome.

Lines 63-67. Every device driver has a *driver header*, which must occur first thing — at offset zero — in the code segment. The first entry in the header is a doubleword, which should be initialized to −1 (= 0FFFFFFFFH). When the driver is installed, **DOS** will overwrite this entry with a pointer to the next device in the linked list (unless, of course, this is the last device in the list). The next item is the *Attribute* word, which is initialized to a symbolic-constant value ATTRIBUTE, to be supplied in the second module of the device driver. Next come near pointers (offsets only) to the driver's Strategy and Interrupt Routines. Finally, there is an 8-byte storage area which, in the case of a character device, should be initialized to the device name (padded on the right with spaces), and, in the case of a block device, should be initialized to the number of units the device will support.

Lines 74-98. This jump table contains all possible commands supported by any device driver running under **DOS**. Notice that the later commands only work with the more recent versions of **DOS** and so should not be implemented in device drivers intended for general use. The comments in the table also indicate whether each command is meant to be used in a character driver or a device driver or both. The commands will be discussed in the next section.

Lines 100-102. We are setting up storage for the address of the Request Header, which is passed to the driver by **DOS**.

Lines 106-112. The Strategy Routine. The only operation required of this routine is to store the address of the Request Header, which is passed to it in **es:bx**.

Lines 116-140. The Interrupt Routine first points **es:di** to the Request Header, retrieving the address stored there by the Strategy Routine. **DOS** has passed, as the byte at address *reqHdr[02H]*, the number of the device driver command to be called. The Interrupt Routine uses this number to index into the jump table, and makes the correct call on line 131. The driver is expected to return a completion code in **ax**. (Possibly an error number in the low 8 bits and bit 15 set to indicate an error, for example.) The Interrupt Routine then ors this completion code with DONE, returning the answer to *reqHdr[03H]*, as required by **DOS**.

Line 153. At this point, the device-specific part of the driver code begins. This consists mostly of the code for the particular driver commands supported by the driver. Later in the chapter, we will give concrete examples.

Exercise 16-6. Write a utility that generates a listing of the device drivers for any machine, formatted like Figure 16.1.

Suggestion: The hardest thing is to find NUL, the start of the device driver chain. In all versions of DOS, this is installed somewhere after 0006:0000 and so the simplest approach is to search through memory, upward from this starting address, looking for the string "NUL." After this driver is found, it is easy to follow the chain of pointers, ending when the device with pointer FFFF:FFFF is reached.

SECTION 5 THE DEVICE DRIVER COMMANDS

Table 16.4 contains a list of all the commands currently supported by installable device drivers. Most of these commands have been available since installable drivers were introduced with Version 2.0 of MS-DOS, but four new commands were introduced with Version 3.0 and three further commands with Version 3.2. The last column of Table 16.4 shows the minimum DOS version with which each command is usable. Of course, it is nearly always desirable to write drivers that will run under DOS versions earlier than 3.0 and so it is common to ignore the later commands or, at least, to check the DOS version at run time and use the later commands only when it is legal to do so. In our discussion, we will concentrate on the main features of the most basic of the commands, referring the reader to [3] or [24] for additional information.

Some of the device driver commands are intended for character devices only, some for block devices only, and some for any kind of device. This information about the commands is contained in the "DEVICE TYPE" column of Table 16.4.

The general usage of the commands is similar: each receives its inputs at certain offsets in the Request Header, and is expected to return its results at prescribed offsets in the Request Header. For the most part, the locations in the Request Header used by the various commands are arbitrary, and must be looked up for each command. However, some of the usage is constant throughout the collection of commands. As examples, during input the number of the command being called is always found as the byte at offset 02H, while each command is expected to record the "return status" as the word at offset 03H.

We will always access the inputs and return values as indicated in the last section: when any command is sent to the driver, **es:di** points to the Request Header, and so introducing an **EQUATE** of the form

```
reqHdr   EQU   es:di
```

allows the data item at offset k in the Request Header to be referred to as *reqHdr[k]*.

We will now discuss the 20 device commands. In some cases, it is possible to discuss several of these commands at once, and in other cases they must be handled singly. Here are the commands:

COMMAND 00H: DRIVER INITIALIZATION (ALL)

INPUTS:

reqHdr[00H] (BYTE)	Request header length
reqHdr[02H] (BYTE)	Command code (=00H)

TABLE 16.4 DEVICE DRIVER COMMANDS

NUMBER	NAME	DEVICE TYPE	DOS VERSION
00h	Initialize	All	2.0
01h	Media Check	Block	2.0
02h	Build BPB	Block	2.0
03h	I/O Control Read	All	2.0
04h	Read	All	2.0
05h	Non-destructive Read	Character	2.0
06h	Input Status	Character	2.0
07h	Flush Input Buffers	Character	2.0
08h	Write	All	2.0
09h	Write With Verify	All	2.0
0ah	Output Status	Character	2.0
0bh	Flush Output Buffers	Character	2.0
0ch	I/O Control Write	All	2.0
0dh	Open	All	3.0
0eh	Close	All	3.0
0fh	Removable Media	Block	3.0
10h	Output Until Busy	Character	3.0
13h	Generic IOCTL Request	All	3.0
17h	Get Logical Device	Block	3.2
18h	Set Logical Device	Block	3.2

reqHdr[12H] (DWORD)	Pointer to first byte after the '=' on the CONFIG.SYS line for this driver

RETURNS:

reqHdr[03H] (WORD)	Return status
reqHdr[0DH (BYTE)	Number of units supported (block devices only)
reqHdr[0EH] (DWORD)	Break address, i.e., address of first free memory above this driver
reqHdr[12H] (DWORD)	BPB pointer array (block devices only)

This command is unique in that it is only called once, when the device driver is installed during the boot process. Hence, it need not remain resident in memory, and, consequently, we will put the code for the *Initialization* command last in our source file and arrange that the resident portion of the driver ends where this code begins.

It is possible to pass parameters to a device driver on its "invocation line." For example, the line in the CONFIG.SYS file that installs a ramdisk program might have the form

```
DEVICE=RAMDISK.SYS /128
```

where the 128 indicates that the disk should have size 128K. It would then be a duty of the *Initialization* command to parse this invocation line, find the size, 128, and then to proceed with whatever setup was required to initialize a ramdisk of this size. To parse the invocation line, the driver code must be able to find it, and this is the reason for the input in *reqHdr[012H]*. It contains a pointer to the region of memory where the invocation line is

نوشتن MACRO

در زبان اسمبلی در بسیاری از مواقع ابتدائی بودن دستورالعملها و نیاز به تکرار آنها و یا گروهی از آنها مشکلاتی را برای برنامه نویس بوجود می آورد. MACRO ها از ابزارهایی هستند که بکمک آنها می توان زبان برنامه نویسی مورد علاقه خود را با دستورالعملهای مورد نظر خود برنامه نویس که همان MACROها هستند نوشت. MACRO ها قابلیتهای زیر را دارا می باشند

* برای ساده کردن و کاهش اندازه کد قابل استفاده است.
* برای ساده کردن یک برنامه اسمبلی و خوانا تر نمودن آن قابل استفاده است
* برای کاهش خطاها ایجاد شده توسط تکرار بعضی خطوط در برنامه.

کدهای تکراری

* فرم کلی یک MACRO به شکل زیر می باشد.

```
<macroname>  MACRO  <argument>
        :
        :
        ENDM
```

بعنوان مثال برای ابتدای یک برنامه EXE می توان کد زیر را بصورت ماکرو تهیه نمود تا این کار را انجام دهد.

```
INIT1   MACRO   CSEG,DSEG,SSEG
        ASSUME  CS:CSEG,DS:DSEG,SS:SSEG,ES:DSEG
        PUSH DS
        SUB AX,AX
        PUSH AX
        MOV AX,DSEG
        MOV DS,AX
        MOV ES,AX
        ENDM
```

صدا زدن: INIT1 CSG,DSG,SSG

برای نمایش لیست داخل ماکروها و یا عدم نمایش آن بصورت زیر عمل می شود:

- .LALL (List all) همه محتویات ماکرو در لیست نشان داده می شود
- .SALL (Suppress all) محتویات ماکرو در لیست برنامه نشان داده نمی شود
- .XALL تنها آن قسمتی از ماکروها در لیست برنامه نشان داده می شوند که از آنها کد OB تولید می شود.

* در هنگام لیست کردن مقابل خطوط لیست شده ماکرو علامت (+) قرار می گیرد.

دستور مجازی LOCAL

در داخل که برنامه گاهی مواقع نیاز به آن است که label ها منحصر بفرد نوشته شود. بخصوص در مورد ماکروها چون در ماکرو نویسی مسائل است که ماکرو بدفعات در برنامه تکرار گردد، اگر label ای در آن بکار گرفته شده باشد در هنگام تکرار ماکرو موجب اشکال خواهد شد. از دستور مجازی LOCAL در این موقع استفاده می گردد تا هنگام تکرار ماکرو در طول کد برنامه این label ها بطور خودکار بصورت منحصر بفردی تبدیل شوند. مثلاً ??0001 و ??0002

مثال:

```
LOCAL dummy-1, dummy-2, ----
```

* اتصال Concatination (&)

از این علامت '&' جهت اتصال حروف یا سمبل استفاده می شود. بعنوان مثال برای دستور MOVS جهت انتخاب نوع کلمه 'W' یا بایتی 'B' آن می توان از ماکرو زیر استفاده نمود.

```
MOVE MACRO TAG
     REP MOVS&TAG
     ENDM
```

حال اگر نوع بایتی آن را بخواهیم MOVE B می نویسیم و برای نوع کلمه آن MOVE W.

تکرار در ماکروها

* دستور تکرار REPT

این دستور به شکل زیر نوشته می شود:

```
REPT expression
```

این عبارت تا رسیدن به دستور ENDM کل عبارات نوشته شده در ادامه را به تعداد (expression) بار تکرار می نماید.

بعنوان مثال در ماکرو زیر پس از اجرا بصورت زیر تبدیل می شود:

```
N=0          DB1
REPT 5       DB2
N=N+1        DB3
DB N         DB4
ENDM         DB5
```

* تکرار IRP : Indefinite Repeat

این دستور باعث می شود که بلوک عبارات نوشته شده بین این خط و ENDM برای همهٔ آرگومانها تکرار شوند.

فرم کلی آن IRP dummy, <arguments>

مثال

```
IRP N, <3,9,17,25,28>
DB N
ENDM
```

اجرا:

```
DB 3
DB 9
DB 17
DB 25
DB 28
```

* تکرار کاراکتر IRPC (Indefinite Repeat character)

این دستور باعث می شود که بلوک عبارات نوشته شده بین این خط و ENDM برای همهٔ کاراکترهای رشتهٔ کاراکتر تکرار گردد.

مثال:

```
IRPC N,345678
DW N
ENDM
```

اجرا:

```
DW 3
DW 4
DW 5
DW 6
DW 7
DW 8
```

دستور العمل مجازی EXITM

اجرای این دستور العمل باعث خروج از ماکرو می شود قبل اینکه به ENDM رسیده باشیم. در ماکروهای شرطی کاربرد فراوان دارد. در ماکروهای تکرار هم قابل استفاده می باشد.

* دستورالعملهای مجازی شرطی

فرم کلی آن بصورت زیر می‌باشد:

IF** (شرط)
⋮
ELSE (در صورت نیاز)
⋮
ENDIF (پایان IF)

} بلوک شرطی

اشکال مختلف دستورالعملهای مجازی شرطی:

IF expression — درصورت (مخالف صفر بودن) درست بودن عبارت قسمت IF اجرا می‌شود

IFE expression — درصورت (صفر بودن) غلط بودن عبارت قسمت IF اجرا می‌شود

IF1 (no expression) — اگر اسمبلر در حال اجرا Pass1 باشد قسمت IF اجرا می‌شود

IF2 (no expression) — اگر اسمبلر در حال اجرا Pass2 باشد قسمت IF اجرا می‌شود

IFDEF symbol — اگر سمبل ذکر شده در برنامه و یا بلوک دستور EXTRN تعریف شده باشد IF اجرا می‌شود

IFNDEF symbol — اگر سمبل ذکر شده در برنامه و یا بلوک دستور EXTRN تعریف نشده باشد IF اجرا می‌شود

IFB <argument> — اگر آرگومان خالی باشد (Blank) آنگاه IF اجرا می‌شود

IFNB <argument> — اگر آرگومان خالی نباشد (Non Blank) آنگاه IF اجرا می‌شود

IFIDN <arg-1>, <arg-2> — اگر آرگومان اول عین آرگومان دوم باشد آنگاه IF اجرا می‌شود.

IFDIF <arg-1>, <arg-2> — اگر آرگومان اول با آرگومان دوم متفاوت باشد آنگاه IF اجرا می‌شود

* نکته: علامت '&' علاوه بر نقش اتصال دهندگی آن به عبارات دیگر کاربرد دیگری هم دارد. در عبارت زیر بدون علامت '&' این عبارت اجرا نخواهد شد.

```
MOVIF   MACRO TAG
        IFIDN <&TAG>, <B>
```

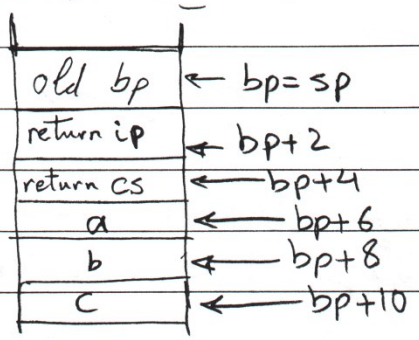
در این مثال <TAG> یعنی خود سمبل TAG ولی <&TAG> یعنی محتویات آن.

* ارتباط با زبان‌های سطح بالا

بطور کلی انتقال پارامترها بر دو نوع است گروهی که پارامترها را از راست به چپ بر روی پشته فشار می‌دهد و گروهی هم برعکس.

* زبان C از جمله گروهی است که از سمت راست پارامترها را بر روی پشته فشار می‌دهد.

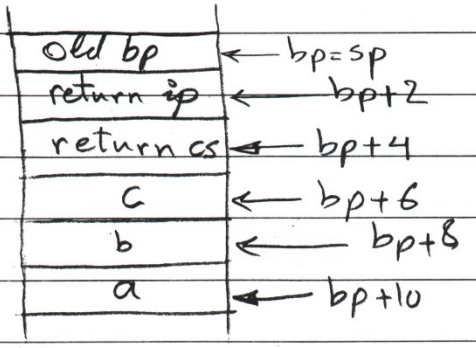
* شکل پشته پس از یک CALL توسط یک فشاردهنده از سمت راست.



call proc1 (a,b,c)

* زبانهای PASCAL، FORTRAN، BASIC از جمله زبانهایی هستند که پارامترها را از چپ به راست بر روی پشته فشار می‌دهند.

* شکل پشته پس از یک FAR CALL توسط یک فشاردهنده از سمت چپ.



proc1 (a,b,c)

* در زبانهایی مثل PASCAL، FORTRAN و BASIC زیر برنامه صدا زده شده وظیفه خالی کردن پشته را برعهده دارد. به همین خاطر در هنگام بازگشت و خروج از زیربرنامه از دستور ret 6 (در مثال بالا) استفاده می‌شود.

* در زبان C همانطور که روش فشار دادن پارامترها بر روی پشته برعکس است روش تخلیه پشته برعکس روش قبل می‌باشد. خالی کردن پشته در اینجا زبان به عهده برنامه صدا زننده است. در این زبان در هنگام بازگشت از زیربرنامه فقط ret استفاده می‌شود.

علامت @ برای استفاده از کلمات رزرو شده در برنامه اسمبلی بکار می‌رود.
مثلاً @Push

علامت % برای متغیرهای عبارت بکار می‌رود. مثلاً %OUT string

علامتهای < > برای تعریف عبارت Text یا string بکار می‌رود.

علامت ! علامت کاراکترها می‌باشد مثلاً "q!" به اسمبلر دستور می‌دهد که از "q" نه به عنوان نشانه Comment برای آن خط بلکه بعنوان خود کاراکتر "q" استفاده کند.

علامت ;; برای Comment (بجای q در زبان اسمبلی) بکار می‌رود.

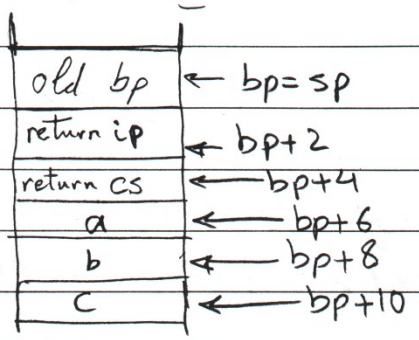
* در هنگام برنامه نویسی می‌توان همچنان که از library ها از زیربرنامه‌ها می‌توان استفاده نمود در عمل حال می‌توان از library ها از ماکروها استفاده کرد.

* ارتباط با زبان‌های سطح بالا

بطور کلی انتقال پارامترها بر دو نوع است گروهی که پارامترها را از راست به چپ بر روی پشته فشار می‌دهد و گروهی هم برعکس.

* زبان C از جمله گروهی است که از سمت راست پارامترها را بر روی پشته فشار می‌دهد.

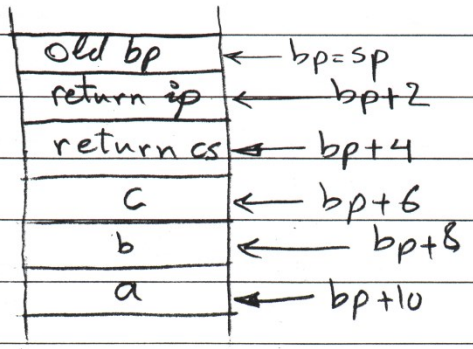
(*) شکل پشته پس از یک CALL توسط یک فشاردهنده از سمت راست.



call proc1 (a,b,c)

(*) زبانهای PASCAL، FORTRAN، BASIC از جمله زبانهایی هستند که پارامترها را از چپ به راست بر روی پشته فشار می‌دهند.

(*) شکل پشته پس از یک FAR CALL توسط یک فشاردهنده از سمت چپ.



proc1 (a,b,c)

* در زبانهایی مثل PASCAL، FORTRAN و BASIC زیر برنامه صدا زده شده وظیفه خالی کردن پشته را برعهده دارد. به همین خاطر در هنگام بازگشت و خروج از زیربرنامه از دستور ret 6 (در مثال بالا) استفاده می‌شود.

* در زبان C همانطور که روش فشار دادن پارامترها بر روی پشته برعکس است روش تخلیه پشته برعکس روش قبل می‌باشد. خالی کردن پشته در اینجا زبان به عهده برنامه صدا زننده است. در این زبان در هنگام بازگشت از زیربرنامه فقط ret استفاده می‌شود.

دل برای مقادیر بزرگتر offset آن در پشته توسط برنامه صدا زننده پوشش می‌شود

شکل پشته با مقدار بازگشتی

Return_Value Offset (rvo)

old bp	← bp=Sp
return ip	← bp+2
return cs	← bp+4
rvo	← bp+6
c	← bp+8
b	← bp+10
a	← bp+12

در زبان C اگر طول rvo بین ۱ تا ۲ بایت باشد در AX و اگر سه یا چهار بایت باشد در DX:AX قرار می‌گیرد. در صورتیکه طولانی‌تر باشد اگر آدرس آن در dx : Ax قرار داده می‌شود. مقادیر برگشتی از چهار بایت در Pascal segment offset

شکل اول شکل C مانند Basic, Fortran می‌باشند

* در هنگام استفاده از برنامه‌های زبانهای سطح بالا همراه با زبان اسمبلی باید به نکات زیر توجه نمود:

۱- از دستورالعملهای مجازی EXTERN (جهت برنامهٔ صدا زننده) و PUBLIC (جهت برنامهٔ صدا زده شده) استفاده کرد تا linker بتواند این دو را به هم وصل کند.

۲- چون زبان C به حروف کوچک یا بزرگ حساس است باید در هر دو سوی صدا زننده و صدا شده این موضوع را در نام زیربرنامهٔ صدا زده شده در نظر گرفت. در عین حال باید توجه داشت که اگر مثلاً نام زیر برنامه در زبان C 'Sub1' است نام آن در زبان اسمبلی 'Sub1' باید باشد (چه هنگامی که از C زبان اسمبلی را صدا می‌زنیم یا برعکس.

* دستور MODEL.

بفرم کلی

.MODEL Mdl, Lang

می‌باشد و کمک این دستور مدل کد را اسمبلر ایجاد می‌کند و هنگام پاس کردن پارامترها و آرگومانها با توجه به زبان گفته شده در ایجاد دستورالعمل آرگومانها بطور مناسبی پاس می‌شوند.

BASIC: تمام آرگومانها را با آدرس دهی نزدیک پاس می‌کند، یعنی تنها قسمت فاصله (offset) آدرس را پاس می‌کند.

C: مقادیر عددی را با مقدار و آرایه‌ها را با آدرس پاس می‌کند. در مدل SMALL آدرس‌ها از نوع نزدیک NEAR می‌باشند.

FORTRAN: تمام آرگومانها را با آدرس دهی دور پاس می‌کند. یعنی هم offset و هم segment آدرس را پاس می‌کند.

PASCAL: آرگومان‌ها را با مقدار پاس می‌کند مگر آنکه آنها VAR یا CONST تعریف شوند که در اینصورت آنها با آدرس دهی نزدیک پاس می‌شوند. یا اگر آنها با VARS یا CONSTS که در اینصورت آنها با آدرس دهی دور پاس می‌شود.

* وقفه‌ها در سیستم 80X86

وقفه‌ها در سیستم 80X86 بر سه نوع می‌باشند.

۱) وقفه‌ها نرم‌افزاری

این نوع وقفه‌ها توسط دستور INT xx فعال می‌شوند. وقفه‌های نرم‌افزاری بسیار شبیه به یک زیر برنامه می‌باشند (با این تفاوت که FLAGS هم بر روی پشته فشرده می‌شوند) که پارامترهای آن بلافاصله به او پاس داده می‌شوند. از انواع معروف این وقفه‌ها INT21H، DOS و وقفه‌های BIOS هستند.

TF ← ۰
IF ← ۰

۲) وقفه‌های سخت‌افزاری

این نوع وقفه‌ها توسط وقایع خارجی بر ریزپردازنده فعال می‌شوند و نه بوسیله INT یا هر دستورالعمل دیگری. این نوع وقفه‌ها بر دو نوع هستند.

الف) وقفه‌های قابل چشم پوشی Maskable Interrupts

این نوع وقفه‌ها با دستور DI قابل چشم پوشی و EI قابل فعال شدن هستند. وقوع وقفه باعث می‌شود تا یکی از بردارهای وقفه در جدول وقفه فعال شود. این خود مشابه یک دستور INT xx می‌باشد با این تفاوت که فعال شدن آن توسط یک عنصر خارجی انجام می‌شود و قابل چشم پوشی می‌باشد.

ب) وقفه‌های غیرقابل چشم پوشی Non-Maskable Interrupts

این وقفه همانند وقفهٔ قبل با واقعهٔ خارجی فعال می‌شود ولی غیرقابل چشم پوشی است. در صورت فعال شدن این وقفه INT02H اجرا خواهد شد.

۳) استثناها Exceptions

این نوع وقفه‌ها هم از نوع وقفه‌های سخت‌افزاری می‌باشند با این تفاوت که منبع درخواست وقفه در داخل ریزپردازنده می‌باشد. از جملهٔ این وقفه‌ها

INT 00H divide-overflow int. (تقسیم بر صفر یا تقسیم منجر به overflow)

INT01H Single-step int. (در debugger ها برای اجرای تک به تک دستورالعمل مورد استفاده قرار می‌دهد)

INT 03H the break point trap این وقفه در debugger ها بکار برده می‌شود. هرگاه نیاز باشد که اجرا بر روی دستورالعمل یا خطا خاصی متوقف شود آن دستورالعمل را با INT 03H جایگزین می‌کنیم. این دستورالعمل جایگزین شده را در داخل روتین وقفه اجرا می‌کنیم و یا در هنگام بازگشت آن را بجای اول قبلی نموده و سپس همان خطا را اجرا می‌کنیم.

* fault (اشکال) یک استثناء است که از شرایطی ایجاد می‌شود که احتمالاً توسط برنامه سرویس دهنده به وقفه قابل تعمیر می‌باشد. اگر چنین تعمیری ممکن و مورد نظر باشد برنامهٔ وقفه می‌تواند اجرای دوبارهٔ برنامهٔ متوقف شده در محل دستورالعملی که موجب خطا گشته است را از سر گیرد.

* Trap (تله) یک استثناء است که هنگام جلوگیری از تکرار یا اجرا دستورالعملهای ناخواسته این استثناء فعال می‌شود. پس از اجرای آن و پایان آن دستورالعمل نباید تکرار شود و در عوض باید به دستورالعمل بعدی رجوع شود.

* Abort یک استثناء است که با طبیعت بسیار جدی آن موجب می‌شود که آغاز بکار دوباره برنامه غیرعملی باشد.

انواع فرمت‌ها برای رشته‌های کاراکتری

۱) نوع (ASCII Z) ASCII Zero

در این نوع فرمت یک بایت صفر انتهای رشته کاراکتری را نشان می‌دهد.
مثلاً در رشته کاراکتری "Hello" :St[6] مقدار قرار گرفته در
حافظه بصورت زیر خواهد بود : ST DB "Hello",00

۲- نوع String

در این نوع فرمت که در ابتدا از Turbo Pascal بکار گرفته شد طول
رشته در اولین بایت (یا Word) نوشته می‌شود. انتخاب طول
بایتی یا Word برای ذخیره طول رشته حرفی باعث می‌شود تا طول آن
بترتیب حداکثر ۲۵۵ کاراکتر یا ۶۵۵۳۵ کاراکتر باشد.

انواع فرمت‌ها برای رشته‌های کاراکتری

۱) نوع ASCII Zero (ASCIIZ)

در این نوع فرمت یک بایت صفر انتهای رشته کاراکتری را نشان می‌دهد. مثلاً در رشته کاراکتری St[6] = "Hello" مقدار قرار گرفته در حافظه بصورت زیر خواهد بود:

ST DB "Hello", 00

۲- نوع String

در این نوع فرمت که در ابتدا از Turbo Pascal بکار گرفته شد طول رشته در اولین بایت (یا Word) نوشته می‌شود. انتخاب طول بایتی یا Word برای ذخیره طول رشته حرفی باعث می‌شود تا طول آن بترتیب حداکثر ۲۵۵ کاراکتر یا ۶۵۵۳۵ کاراکتر باشد.

TABLE 15.1 EXCEPTIONS AND HARDWARE INTERRUPTS

INTERRUPT NUMBER	INTERRUPT NAME
00H	Divide by zero
01H	Single step
02H	Non-maskable interrupt
03H	Breakpoint trap
04H	Overflow trap
05H	Print screen
06H	Unused
07H	Unused
08H	Timer tick (IRQ0)
09H	Keyboard (IRQ1)
0AH	Reserved (IRQ2)
0BH	COM2 (IRQ3)
0CH	COM1 (IRQ4)
0DH	Fixed disk (IRQ5)
0EH	Floppy disk (IRQ6)
0FH	Printer (IRQ7)

Int 03H, the *breakpoint trap*, is also used by debuggers. Whereas the other interrupts are represented by two bytes of machine code, **int 03H** translates into a single machine code byte, namely 0CCH. At any location in the code where a breakpoint has been requested, the debugger replaces the existing byte of code with 0CCH. When the microprocessor encounters this special byte, it activates **int 03H**. Again, the default handler for **int 03H** consists only of an **iret**, and so must be replaced with a handler designed to take appropriate action.

Int 04H is called the *overflow interrupt*. It is executed whenever the overflow flag is set *and* an **into** instruction is encountered. This allows *trapping* instances of overflow. By placing an **into** instruction at any location in the code, the programmer can arrange to have control transfer to **int 04H** if overflow occurs at that point. Once again, the default handler consists of an **iret** instruction. Of course, the programmer should decide on the required overflow response in a given program, and replace the standard **int 04H** handler with a new one delivering this result. In Section 3, we will show how to replace an interrupt handler, using **int 04H** as an example.

The hardware interrupts are of two types, the non-maskable interrupt and the maskable interrupts.

The *non-maskable interrupt* (*NMI*), which is **int 02H**, occurs when a particular pin (number 17) on the **8088** goes high. It is used to signal some hardware catastrophe, and the default response is to halt the machine. In the standard **IBM-XT** architecture, **int 02H** is initiated by a memory parity-check error, although it is sometimes put to additional uses by the vendors of peripheral cards. In a few paragraphs, we will explain why this interrupt is called non-maskable.

All other hardware interrupts in the **IBM-XT** are referred to as *maskable*. The maskable interrupts are those numbered 08H-0FH in Table 15.1, and their names indicate their uses.

The maskable interrupts are administered by the **8259A** *programmable interrupt controller* (*PIC*) *chip*. Many aspects of the PIC behavior can be controlled by the program-

dire consequences. However, Microsoft strongly discourages the direct overwriting of interrupt vectors because it may not work under future versions of **MS-DOS**, and certainly would not work under **OS/2**. One difficulty stems from the fact that more advanced operating systems allow *multi-tasking*. That is, they allow several programs to execute simultaneously on the same machine. If, under those circumstances, a program changed an interrupt vector without notifying the operating system, then another executing program would get a surprise if it tried to use that interrupt! Of course, if the operating system has been notified of the modification of the interrupt vector, it can make the appropriate adjustment when it is "switching tasks."

The end result is that more sophisticated operating systems will not allow a program to "poke" new values into the interrupt vectors. Rather, some mechanism is provided by which the program can request the operating system itself to make such changes. **MS-DOS** has such a mechanism, and we will always use it to change interrupt vectors.

Specifically, **function 25H** of **int 21H** is used to request the replacement of an interrupt vector. As inputs, it requires that the interrupt number be passed in **al** and that **ds:dx** point to the new handler code. The following macro can be used to call **function 25H**. It has only one interesting feature. Programs that change interrupts are frequently in .COM format, and so (since, for the most part, **ds = cs** in these programs) **ds** will probably already contain the segment address of the new handler. In such cases, it is unnecessary to pass the *Segment* argument, and the macro responds correctly when the *Segment* parameter is blank.

```
;install new interrupt vector
;if Segment parameter not passed, it is assumed to be ds
setInt   MACRO  Number, Offset, Segment

    pushRegs  ax, dx
    mov   dx, Offset
    IFNB <Segment>
        push  ds                ;to preserve it
        mov   ax, Segment
        mov   ds, ax
    ENDIF
    mov   al, Number
    mov   ah, 25h               ;set interrupt vector
    int   21h
    IFNB <Segment>
        pop ds                  ;to restore it
    ENDIF
    popRegs

ENDM
```

The old interrupt vector must be stored for later restoration. **Function 35H** of **int 21H** can be used to determine the value of the interrupt vector before it is changed. It requires that the interrupt number be input in **al** and it returns the segment and offset of the interrupt vector in **es and bx**, respectively. The following macro implements this function, taking as parameters the interrupt number and one-word storage locations for the offset and segment parts of the vector.

```
;store old interrupt vector - destroys ax, bx, es
getInt MACRO  Number, Offset, Segment

    mov    al, Number
    mov    ah, 35h              ;get interrupt vector
    int    21h
    mov    Offset, bx
    mov    Segment, es

ENDM
```

We are finally ready for our example, which is shown in Figure 15.1. This program demonstrates the use of the *getInt* and *setInt* macros to replace and restore the handler for **int 04H**, the divide-overflow interrupt.

We have chosen to write the program in .COM format, but there was no compelling reason to do this. However, several of the later programs in this chapter are memory-resident, and it is traditional to write memory-resident programs in .COM format, since this tends to minimize program size.

NEW04H.ASM uses the **GROUP** directive and the format introduced in Section 2 of Chapter 12. Recall the advantages of this approach: when **MASM** reads the source file, the data occur first; consequently, phase errors are avoided without the use of explicit segment overrides. Lines 7-11 ensure that, when the program is linked, code precedes data, a requirement for the program to be convertible to .COM format.

For the most part, the mechanics of the program are straightforward:

Lines 33-37. We store the old interrupt vector and replace it with the address of the substitute handler, the procedure *New04H*. Since **ds** = **cs**, the third argument was not needed in *setInt*.

Lines 39-46. A few trivial tests of the functioning of the new handler.

Line 49. Restoring the original handler.

Lines 56-75. The code for the replacement handler.

Lines 60-62. These lines show the inelegance of the *inCh* macro. A single character is expected from STDIN (presumably 'y' or 'n'), but this character arrives followed by a CR. In cooked mode, the CR is converted to a {CR, LF} pair and so three calls to *inCh* are required to clear the input buffer.

Line 69. The procedure must end with an **iret** rather than a **ret**. After all, this handler is called, not by a **call**, but by an **int**. Consequently, the flags register has been pushed onto the stack and needs the **iret** to remove it. (Of course, just as an **int** can be fabricated from a **pushf** and a **FAR call**, an **iret** can be constructed from a **retf** and a **popf** or, if the flag values are not of interest, by a **retf 2**.)

Lines 72-73. In this case no **iret** is needed — we are just transferring control back to the operating system.

Figure 15.2 cont.

```
169
170     ;get original int 05h vector
171     getInt 05h, orig05hOffset, orig05hSegment
172
173     ;replace with new int 05h vector
174     setInt 05h, <OFFSET New05h>, cs
175
176     ;get original int 1Ch vector
177     getInt 1ch, orig1CHOffset, orig1CHSegment
178
179     ;replace with new int 1ch vector
180     setInt 1ch, <OFFSET New1ch>, cs
181
182     ;get original int 28h vector
183     getInt 28h, orig28hOffset, orig28hSegment
184
185     ;replace with new int 28h vector
186     setInt 28h, <OFFSET New28h>, cs
187
188     mov     ah, 34h      ;find address of DOS ACTIVE BYTE
189     int     21h
190     mov     inDOSSegment, es          ;save it
191     mov     inDOSOffset, bx
192
193     ;free environment space
194
195     pop     es                        ;segment address of environment
196     mov     es, es:[2ch]              ;free memory block
197     mov     ah, 49h
198     int     21h
199
200                                       ;installation notice
201     outAsc Notice
202
203     mov     dx, _TEXT                 ;dx = starting paragraph of code
204     sub     dx, _DATA                 ;dx = number of paragraphs in _DATA
205     add     dx, 10h                   ;extra paragraphs for the PSP
206     add     dx, ((Start - New05h) + 15)/16 ;dx = total paragraphs to remain
207                                       ;  resident
208     mov     ax, 3100h                 ;terminate and stay resident
209     int     21h
210
211 _TEXT       ENDS
212
213 ;=========================================================
214
215 Stack  SEGMENT  STACK 'STACK'
216
217         DW 100 DUP (?)
218
219 Stack ENDS
220
221 ;=========================================================
222
223       END Start
```

FIGURE 15.2 A NEW HANDLER FOR INT 05H

```
TITLE  saveScr.asm: changes Shift PrtScr so that it writes to a disk file
;uses int 28h and int 1Ch as "hooks"
;=============================================================
INCLUDE console.mac
INCLUDE file.mac
;=============================================================
_DATA SEGMENT WORD PUBLIC 'DATA'
;=============================================================
    orig05hVector
    orig05hOffset       LABEL     DWORD
    orig05hSegment      DW        ?
                        DW        ?
    orig1CHVector
    orig1CHOffset       LABEL     DWORD
    orig1CHSegment      DW        ?
                        DW        ?
    orig28hVector
    orig28hOffset       LABEL     DWORD
    orig28hSegment      DW        ?
                        DW        ?
    inDOSAdr
    inDOSOffset         LABEL     DWORD
    inDOSSegment        DW        ?          ;address of DOS ACTIVE BYTE
                        DW        ?
    Request
                        DW        0
    Handle1
    dataFile            DW        ?          ;non-zero if Shift PrtScr
    scrBuf              DB        'Data', 0  ; request pending
    bufLen              DB        2050 dup(?) ;storage for file handle
                        DW        2050       ;file name
    Notice      DB  CR, LF, 'NEW PRT_SCR RESIDENT MODULE INSTALLED'
                DB  CR, LF, 0                ;temporary storage for screen
_DATA   ENDS
;=============================================================
_TEXT SEGMENT WORD PUBLIC 'CODE'
;=============================================================
        ASSUME cs:_TEXT, ds:_DATA
;-------------------------------------------------------------
New05h PROC FAR
;-------------------------------------------------------------
    pushRegs  ax, ds
    mov     ax, _DATA                   ;preserve interrupted ds
    mov     ds, ax                      ;so that resident data
    mov     Request, 1                  ;  can be addressed
                                        ;notification of request
```

Figure 15.2 cont.

```
    popRegs                         ;restore interrupted ds
    iret

New05h ENDP

;-----------------------------------------------------------

New1Ch  PROC FAR                    ;usr timer interrupt

    pushRegs ax, bx, ds, es
    mov     ax, _DATA
    mov     ds, ax
    pushf
    cli
    call    orig1CHVector

    sti                             ;turn on interrupts
    cmp     Request, 0              ;if no request
    je      Done1                   ;  pending -- exit
    mov     ax, inDOSSegment        ;else get DOS ACTIVE BYTE
    mov     es, ax
    mov     bx, inDOSOffset
    cmp     BYTE PTR es:[bx], 0
    jne     Done1                   ;if DOS is busy -- wait until next time
    call    saveScreen              ;  if not, get screen now
    mov     Request, 0              ;request has been dealt with

Done1:

    popRegs
    iret

New1Ch ENDP

;-----------------------------------------------------------

New28h  PROC FAR                    ;if here, DOS is safe

    pushRegs ax, ds

    mov     ax, _DATA
    mov     ds, ax
    pushf
    cli
    call    orig28hVector           ;chain to old one, in case another
                                    ;  application is also using it
    cmp     Request, 0              ;if no request pending, exit
    je     Done2
    call    saveScreen              ;else get the screen
    mov     Request, 0              ;  and signal that request fulfilled

Done2:

    popRegs
    iret
```

```
New28h  ENDP

;-----------------------------------------------------------------

saveScreen PROC NEAR

    pushRegs ax, bx, dx, di, si, ds, es
    mov     ax, _DATA
    mov     es, ax
    push    es

    mov     ax, 0b800h              ;0b000h for mono screen
    mov     ds, ax
    xor     si, si                  ;ds:si points to video buffer
    mov     di, OFFSET scrBuf       ;es:di points to the scrBuf

    mov     cx, 25                  ;number of lines to move

nextLine:

    mov     dx, 80                  ;number of characters per line

nextChar:

    lodsw                           ;load ASCII code and attribute byte
    stosb                           ;store only the ASCII code
    dec     dx
    jnz     NextChar
    mov     ax, (LF SHL 8) OR CR    ;terminate each line with (CR,LF) pair
    stosw

    loop    nextLine

    pop     ds
    Open    dataFile 2 Handle1      ;open for appending if it exists
    jnc     Exists
    Create dataFile 0 Handle1       ;else create

Exists:

    Lseek   Handle1 2 0 0
    Write   Handle1 bufLen scrBuf
    Close   Handle1
    mov     Request, 0              ;signal that task is complete

    popRegs
    ret

saveScreen ENDP

;-----------------------------------------------------------------

Start:

    mov     ax, _DATA
    mov     ds, ax

    push    es                      ;save segment of PSP
```

to call the device driver code, which translates the request into a form appropriate for the given device interface.

Under this new scheme, every device is totally integrated into the system and inherits all the perquisites of the existing devices: its name or letter designation can be used in any context where an existing device can be used. To illustrate, when we install our device driver RAMDISK later in this chapter, and assuming it is assigned the drive designation **F:** (as it was in our case), then the following commands are legal:

```
copy fileName f:
dir f:
chkdsk f:
```

They do the expected things.

The basic idea behind installable device drivers is that the operating system neither knows nor cares about the hardware details of input or output devices. The only requirement is that each I/O device have a unique name or other unique designation. When **DOS** receives a request from an application program to make use of an I/O device (perhaps to perform a read or a write), it formats this request in a special way (using a data packet called a *request header*) and transmits it to the corresponding device driver for execution. If it is a write, for example, the contents of the request header are (almost) independent of whether the data are written to PRN or to drive C:. Of course, a different device driver is called in each case.

It is therefore only a slight exaggeration to say that, to **MS-DOS**, a printer looks exactly like a disk drive! We will now discuss the nature of this "slight exaggeration."

DOS recognizes two kinds of device drivers, called *character device drivers* and *block device drivers*, and these names suggest the distinction being made. The first kind of driver is associated with devices that generally perform I/O one character at a time: the keyboard, the screen, and the printer are examples. The second kind is used for devices that read or write in larger blocks: disk drives, magnetic tapes, and CD ROMs are examples.

Of course, large blocks of data are sometimes sent to the screen and single characters are sent to a disk file, and so the preceding characterizations are not terribly precise. When we look at the device driver commands in detail in the later sections of this chapter, we will see that the real distinction is based on exactly which commands we wish to have the driver execute and what background services we desire from **DOS**. For example, **DOS** writes to a block device in blocks of size at least one "sector." While the size of a sector could be as small as one byte, it is frequently taken to be 512 bytes, which shows that block devices are generally used when large data transfers are anticipated.

The code for a device driver is written essentially in .COM format. The only difference is that there will be no **PSP** and so the code should begin at the start of the code segment — the *ORG 100* used in .COM files must be omitted.

As the reader probably knows, installable devices are installed at boot time by placing invocation lines in the CONFIG.SYS file. Device drivers are generally given the extension .SYS (although this is not required). To install the driver named DRIVER1.SYS, the CONFIG.SYS file should contain the line

```
DEVICE=DRIVER1.SYS
```

FIGURE 16.1 EXAMPLE LIST OF DEVICE DRIVERS

BASE ADDRESS	TYPE	NAME/UNITS
0006:2BF8	CHARDEV	NUL
10D2:0000	BLOCKDEV	1 UNIT
10A5:0000	CHARDEV	LASR
0FE2:0000	BLOCKDEV	2 UNITS
0C26:0000	CHARDEV	SMARTAAR
09A5:0000	CHARDEV	MS$MOUSE
0070:0BB3	CHARDEV	CON
0070:0C68	CHARDEV	AUX
0070:0D17	CHARDEV	PRN
0070:0E15	CHARDEV	CLOCK$
0070:0EE5	BLOCKDEV	3 UNITS
0070:0C7A	CHARDEV	COM1
0070:0D29	CHARDEV	LPT1
0070:2071	CHARDEV	LPT2
0070:2083	CHARDEV	LPT3
0070:2095	CHARDEV	COM2

13 CHARDEVs, 3 BLOCKDEVs

As well as installable device drivers, DOS comes with certain built-in *resident device drivers*. The resident drivers service the keyboard, the screen, the real-time clock, the parallel printer, the serial port, and at least one disk drive (so that the machine can boot). In addition, there is a resident driver named NUL which, as the name suggests, does nothing.

The device drivers are stored in memory as a linked list, where the first doubleword in each driver is a far pointer to the next driver in the list. The NUL device serves as the list header (which is one reason for its existence), the resident drivers are entered prior to the installed ones, and the last driver is the list has its pointer set to FFFF:FFFF.

When DOS adds new entries to the list, they are not always added at the beginning. However, the list is constructed in such a way that new character devices appear ahead of old ones. In this way, if a new character device named PRN, say, is added, it will be the one found first by DOS when traversing the list, and so will replace the original PRN. On the other hand, block devices are not assigned names, but rather consecutive letters A:, B:, C:,···, and these letters reflect the order in which the drivers were installed. Since new block devices are given different letters, they never replace previous block devices. Of course, if a block device is installable, it can be removed from CONFIG.SYS, but if it is resident, it is unchangeable.

Figure 16.1 shows the list of device drivers, resident and installed, in a typical machine. Most of the information in Figure 16.1 should be understandable. One new item is that, in the case of block devices, it is possible to install several "units" with one device driver. For example, if the last letter used by a previous block driver was D: and the number of units is three, then the new units would be designated E:, F:, and G:.

In one of the exercises at the end of the section we make suggestions for writing a utility to produce a listing of devices on the host machine, formatted like Figure 16.1.

Now that we have explored the general ideas underlying device drivers, it is time to study the device driver format in detail. The code for a device driver splits neatly into two parts: the first part is common to all drivers, and we will examine it in Section 4; the second part is specific to the driver, and we will illustrate it with two examples, a minimal laser printer driver, called LASR, in Section 6, and a ramdisk driver in Section 7. Both of these drivers can be seen in the list in Figure 16.1. (The ramdisk is the 1-unit block driver.)

Unfortunately, in order to understand the functioning of installable device drivers completely, it is first necessary to consider some aspects of the structure and formatting of disks under MS-DOS, and we will look at this topic in the next section.

The reader who is solely interested in writing character device drivers or who is willing to take a few things on faith can safely omit Section 3.

SECTION 3 THE FORMAT OF MS-DOS DISKS

We will only look at those aspects of disk structure used in the present chapter: the nature of the Bios Parameter Block, the structure of the File Allocation Table, and the general form of a directory entry. The exercises at the end of this section are intended to help the reader in exploring these ideas. We do not need to know about the physical structure of the disk, which involves such things as the number of tracks and heads and the interleave factor. For a discussion of these matters, and for more details on the aspects that we do discuss, we refer the interested reader to [3] or [24].

Every disk formatted under DOS is divided into *sectors*. A sector is just a contiguous storage area on the disk of a size that must be a power of two and is chosen when the disk is formatted. The standard sector size for MS-DOS disks is 512 bytes.

The first few sectors on a disk are used to store various records telling the system exactly how the disk is formatted and enabling it to locate the data stored on the remainder of the disk. In particular, the first sector (sector 0) is called the *boot sector*. It contains information on the disk format and a short program (the *boot program*) that is run if an attempt is made to boot the system using that disk. Of course, if the disk does not contain the standard system files (named IO.SYS and MSDOS.SYS in the Microsoft version of MS-DOS — see Chapter 2), any attempt to boot from the disk will fail. Figure 16.2 shows a hex dump of the first few paragraphs of the boot sector of a typical 360K MS-DOS floppy.

The boot sector always begins with a **jmp** instruction, which is a jump to the boot code and is allocated the first three bytes on the disk. Sometimes this is encoded as a **SHORT** jump, which only requires two bytes of machine code, and it is then followed by a **nop** to pad the length to three bytes. This is the configuration shown in Figure 16.2. In other cases, the first three bytes are encoded as a **NEAR** jump, which uses three bytes with no **nop** being necessary. In our example, the displacement contained in the **jmp** is 34H, and so the target is actually 36H. As an exercise, we ask the reader to disassemble the boot code on a floppy and attempt to decipher its meaning.

The next eight bytes are reserved for the machine manufacturer to insert an identification code.

From our point of view, the next 19 bytes are the most interesting part of the boot sector. They constitute the *Bios Parameter Block*, or *BPB*, and have been highlighted on the first

FIGURE 16.2 THE BEGINNING OF A BOOT SECTOR

HEXDUMP

```
2088:0000  EB 34 90 49 42 4D 20 20-33 2E 32 00 02 02 01 00  k4.IBM  3.2.....
2088:0010  02 70 00 D0 02 FD 02 00-09 00 02 00 00 00 00 00  .p.P.}..........
2088:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 0F  ................
2088:0030  00 00 00 00 01 00 FA 33-C0 8E D0 BC 00 7C 16 07  ......z3@.P.|..
2088:0040  BB 78 00 36 C5 37 1E 56-16 53 BF 2B 7C B9 0B 00  ;x.6E7.V.S?+|9..
2088:0050  FC AC 26 80 3D 00 74 03-26 8A 05 AA 8A C4 E2 F1  |,&.=.t.&..*.Dbq
2088:0060  06 1F 89 47 02 C7 07 2B-7C FB CD 13 72 67 A0 10  ...G.G.+|{M.rg .
(THE BPB OCCUPIES THE RANGE OF OFFSETS 0BH-1D)
```

THE FIELDS OF THE THE BPB

OFFSETS	VALUE	MEANING
00h-02h	-----	Machine code for a short jmp and a nop
02h-0bh	-----	Text: vendor and version number
0bh-0ch	0200h	Bytes per sector
0Dh-0dh	02h	Sectors per cluster
0eh-0eh	0001h	Reserved sectors
10h-10h	02h	Number of FATs
11h-12h	0070h	Number of root directory entries
13h-14h	02d0h	Total sectors on disk
15h-15h	fdh	Media descriptor byte
16h-17h	0002h	Sectors per FAT
18h-19h	~~0090h~~ 0009h	Sectors per track
1ah-1bh	0002h	Number of heads
1ch-1dh	0000h	Number of hidden sectors

(THE BPB OCCUPIES THE RANGE OF OFFSETS 0BH-1DH)

display in Figure 16.2. These bytes contain information about the structure of the disk, and are shown again on the second part of Figure 16.2, along with their meanings. We will explain those BPB fields that are most important to us.

First of all, sectors are organized into contiguous groups called *clusters*. The number of sectors in a cluster must be a power of two, and any file on the disk will be allocated a whole number of clusters. For instance, if (as here) the cluster size is 2 and the sector size is 512, then any file, no matter how small, will use at least 1024 bytes of disk space. If a file contains 1025 bytes, it will use 2048 bytes of space on the disk.

Although it is peripheral to our main theme, we will briefly discuss the significance of cluster size in efficient disk usage. The numbers in the last paragraph already suggest a weakness of large cluster size: it can be wasteful of space, especially when many small files are stored on the disk. But, on the other hand, we will see that when a file is stored in several clusters on the disk, these clusters may not be contiguous and the operation of finding the correct "next" cluster involves a non-trivial computation. Hence, small cluster size tends to slow disk access whenever large files are being used. A general guideline for selecting cluster size is that large hard disks will likely be used to store large files and so it is probably safe

TABLE 16.1 FORMAT OF A 32-BYTE DIRECTORY ENTRY

OFFSETS	CONTENTS
00h-07h	Filename
08h-0ah	Extension
0bh-0bh	Attribute
0ch-15h	Reserved
16h-17h	Time file most recently changed
18h-19h	Date file most recently changed
1ah-1bh	Starting cluster number
1ch-1fh	Filesize (double precision integer)

to give them large cluster size. For a different reason, to be discussed below, it turns out that large disks *must* be given a large cluster size.

It is possible to reserve any number of sectors (at the start of the disk) to store specialized information or for any other reason. The boot sector is considered to be a reserved sector and most disks have no others. Hence the normal value for the number of *reserved sectors* is one.

Each file stored on the disk must have a *directory entry*. In addition, the volume label and subdirectory names are stored as directory entries. A directory entry requires 32 bytes of disk storage and is formatted as shown in Table 16.1.

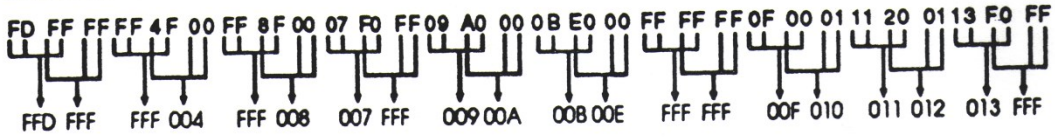
The interpretation of some of the fields in a directory entry can be quite technical, and we refer the reader to [3] or [24] for details. For example, we have already encountered the formats used for the date and time fields. (See Section 4 of Chapter 12.) One item of interest to us is that the directory entry for a file contains the starting cluster number of the file, which enables the operating system to find the beginning of the file on the disk. Figure 16.3 contains a dump of the first part of a disk directory.

A DOS disk can contain many subdirectories, each of which has its own directory entry. Whether a given directory entry represents a file or a subdirectory or a volume label is communicated by the value of its *attribute*. (The attribute was defined in Section 3 of Chapter 8.) In general, a DOS directory can contain an arbitrary number of entries: the list of entries in the directory just grows longer, as needed. The only exception to this rule is the root directory, which has a maximum number of entries that is recorded in the BPB. In our example, this number is 70H = 112. Since DOS knows this number and also knows that each directory entry requires 32 bytes, it can easily compute how many sectors are required for the root directory. In this case we get 112 * 32 / 512 = 7.

These sectors are allocated near the start of the disk, but behind another important storage area, the *file allocation table*. The file allocation table (known as the FAT) is so important that most DOS disks contain more than one copy of it, with two being the normal number. When the FAT is updated, all copies are changed. In principle, if one copy of the FAT becomes corrupted, it is possible to recover the vital information that it contains from one of the backup copies. There is good reason to keep at least one backup of the FAT, since without it the disk is virtually worthless. The disk whose boot record is shown in Figure 16.2 contained two copies of the FAT but, unfortunately, present versions of MS-DOS do not make any use of extra copies of the FAT.

FIGURE 16.3 EXAMPLE: DIRECTORY AND 12-BIT FAT

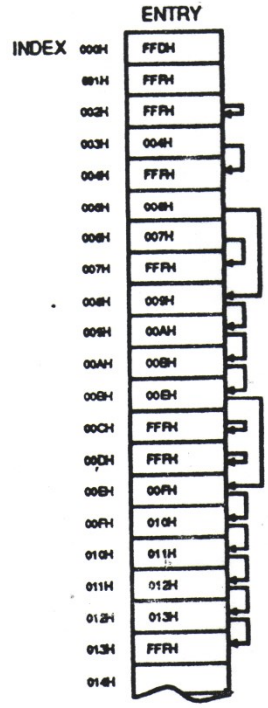
BEGINNING OF 12-BIT FAT AS STORED:



TRANSLATION INTO FAT ENTRIES

EDITED DEBUG DUMP OF BEGINNING OF DISK DIRECTORY

```
46 49 4C 45 31 20 20 20-20 20 20 20 00 00 00 00  FILE1       ....
00 00 00 00 00 00 6F 8D-41 13 02 00 00 04 00 00  ......o.A.......
4C 45 46 54 20 20 20 20-43 30 30 20 00 00 00 00  LEFT    C00 ....
00 00 00 00 00 00 05 A0-41 13 05 00 00 2A 00 00  ......N A.N..H..
E5 48 41 50 54 31 37 20-44 4F 43 20 00 00 00 00  eHAPT17 DOC ....
00 00 00 00 00 00 4D A0-43 13 3A 00 80 24 01 00  ......M C.:..$..
46 49 4C 45 34 20 20 20-20 20 20 20 00 00 00 00  FILE4       ....
00 00 00 00 00 00 27 9D-41 13 06 00 80 04 00 00  ......'.A.......
46 49 4C 45 31 31 20 20-20 20 20 20 00 00 00 00  FILE11      ....
00 00 00 00 00 00 27 9D-41 13 03 00 80 04 00 00  ......'.A.......
46 49 4C 45 31 32 20 20-20 20 20 20 00 00 00 00  FILE12      ....
00 00 00 00 00 00 27 9D-41 13 1A 00 80 04 00 00  ......'.A.......
46 49 4C 45 37 20 20 20-20 20 20 20 00 00 00 00  FILE7       ....
00 00 00 00 00 00 6F 8D-41 13 0C 00 00 04 00 00  ......o.A.......
46 49 4C 45 38 20 20 20-20 20 20 20 00 00 00 00  FILE8       ....
00 00 00 00 00 00 6F 8D-41 13 0D 00 00 04 00 00  ......o.A.......
E5 49 4C 45 39 20 20 20-20 20 20 20 00 00 00 00  eILE9       ....
00 00 00 00 00 00 27 9D-41 13 0E 00 80 04 00 00  ......'.A.......
E5 49 4C 45 31 30 20 20-20 20 20 20 00 00 00 00  eILE10      ....
00 00 00 00 00 00 27 9D-41 13 10 00 80 04 00 00  ......'.A.......
E5 49 4C 45 31 31 20 20-20 20 20 20 00 00 00 00  eILE11      ....
00 00 00 00 00 00 6F 8D-41 13 12 00 00 04 00 00  ......o.A.......
```



A FAT always begins with two overhead entries: a 12-bit FAT opens with the byte pattern

```
mediaDescriptor, 0FFH, 0FFH
```

which exactly fills 24 bits = 2 FAT entries; a 16-bit FAT starts with

```
mediaDescriptor, 0FFH, 0FFH, 0FFH
```

which again exactly fills 32 bits = 2 FAT entries.

The first part of Figure 16.3 shows the initial 30 bytes of the FAT of a floppy disk, and how the 20 corresponding FAT entries are obtained from these bytes.

TABLE 16.2 MEANINGS OF THE FAT ENTRIES

12-BIT FAT VALUE	16-BIT FAT VALUE	MEANING
000h	0000h	Cluster not in use
001h	0001h	This value is not used
ff0h - ff6h	fff0h-fff6h	Reserved
ff7h	fff7h	Bad cluster
ff8h-fffh	fff8h-ffffh	Last cluster in a file
Any other value	Any other value	Link to the next file cluster

As we pointed out previously, the directory entry for a file contains the number of its starting cluster. The FAT contains the "linkage" enabling the operating system to find the remaining clusters constituting the file. Of course, since the directory entry contains the file size, it is a trivial matter to determine how many clusters the file occupies. But the snag is that these clusters do not always occur contiguously on the disk.

Here is how the FAT is used to find the clusters corresponding to a given file. First, the clusters are numbered sequentially throughout the data area of the disk, and the FAT contains an entry corresponding to each cluster. The directory entry for the file contains the first cluster number, say N_1, for the file. The N_1-entry in the FAT then contains the second cluster number, say N_2, the N_2-entry in the FAT contains the third cluster number, and so on. The process continues until all the clusters have been found. Recall that the operating system knows in advance how many clusters there will be but, as a double check, the FAT entry with index equal to the last cluster number contains a special code to indicate that the end of the file has been reached. The values of this and other special codes are shown in Table 16.2.

Two different sizes are used for FAT entries: "small" disks generally have FAT entries of size 12 bits, while "large" disks have FAT entries of size 16 bits. We will examine the small case more closely. Because each cluster number must have a corresponding FAT number, and since $2^{12} = 4096$, it follows that a disk with a 12-bit FAT is restricted to a maximum size of about 4000 clusters. Since we are now talking about a (relatively) small disk, it would seem reasonable to limit the cluster size to two sectors, or 1024 bytes, using the standard DOS sector size. Hence the disk size would be limited to about 4 megabytes. DOS is actually somewhat more liberal than this: it uses a 12-bit FAT if the disk has fewer that 20740 sectors and a 16-bit FAT otherwise. In any case, it is clear that most hard disks use a 16-bit FAT.

Finally, we can check that a reasonable number of sectors has been allocated to each copy of the FAT in the example in Figure 16.2. According to Figure 16.2, there are 02D0H = 720 sectors, or 360 clusters. Hence the FAT requires 360 12-bit entries. In addition, we will see that the FAT contains a few overhead entries, and so this translates to about 550 bytes per FAT, whence each FAT should be allocated two sectors, as was done.

The only remaining item of interest to us in Figure 16.2 is the *media descriptor byte*, which, as the name suggests, is a code telling DOS whether the disk is a hard disk or a floppy of any of various sizes or densities. Table 16.3 contains a list of the most important values taken by this byte, and their interpretations.

TABLE 16.3 THE VALUES OF THE MEDIA DESCRIPTOR BYTE

VALUE	DISK TYPE	BEGINNING DOS VERSION
0f8H	Fixed disk	2.0
0f0H	3.5 inch, 2-sided, 18-sector	3.2
0f9H	3.5 inch, 2-sided, 9-sector	3.2
0f9H	5.25 inch, 2-sided, 15-sector	3.0
0fch	5.25 inch, 1-sided, 9-sector	2.0
0fdh	5.25 inch, 2-sided, 9-sector	2.0
0feh	5.25 inch, 1-sided, 8-sector	1.0
0ffh	5.25 inch, 2-sided, 8-sector	1.1

Immediately below this display is a dump of the start of the disk directory. This information was obtained using **DEBUG**, one of the older Microsoft debuggers, and the technique for obtaining such a display is discussed in the exercises at the end of this section. As Table 16.1 shows, a directory entry begins with the filename and, at offsets 1AH and 1BH, contains the starting cluster number of the file. When a file is deleted, **DOS** replaces the first letter in its name with the special byte 0E5H, as well as marking all its assigned clusters with value 000H (or 0000H in the 16-bit case) to indicate that they are free. A number of deleted files can be seen in the directory in Figure 16.3.

The reader should check the accuracy of the FAT diagram at the right of Figure 16.3. For each file that has not been erased, we have printed the starting cluster number in bold. Starting with this number, it is a simple matter to "follow the FAT chains" to find the locations of the later clusters in the file. As a final check, it should be noted that the size of each file (bytes 1CH-1FH) is consistent with the number of clusters in its chain.

Note on the Following Exercises. The object of the next few exercises is to explore the structure of disks written in the standard **DOS** format, and a word of caution may be in order at this point. When writing programs that examine disk structure, and especially programs that write to disks, it is only sensible to do all development work using a floppy with expendable data as the target. Before experimenting with such a utility on a hard disk, be sure to make a backup.

In order to study the structure of a disk, it is necessary to read its contents first, and this is one area in which **CODEVIEW** is weaker than its predecessors **DEBUG** and **SYMDEB**. Either of these earlier debuggers can be asked to read from or write to particular sectors of a disk, but **CODEVIEW** cannot. This weakness of **CODEVIEW** is not really serious since most programmers have a copy of **DEBUG** (which comes free with **MS-DOS**).

For anyone familiar with **CODEVIEW**, the earlier debuggers are easy to use. To enter either **DEBUG** or **SYMDEB**, just type its name at the **DOS** prompt. To load a file into either debugger, give its name as a command-line parameter. Any kind of file, executable or not, can be loaded, and will be found at the location appropriate for a .COM file, namely at cs:0100. Both debuggers have (very primitive) on-line help, accessed from the command line by the command '?'. The main command that we care about is 'L', for *load*. As well as the

L *fileName*

option, which works like the corresponding **CODEVIEW** menu command, there is an

L *Address Drive Start Number*

option, which loads sectors from a disk drive. *Address* is the address (*segment:offset*) at which they will be loaded. If the segment is not specified, the default is **cs**. *Drive* is the designator of the drive to load from, where 0 means **A:**, etc. *Start* is the starting sector number, and *Number* is the number of sectors to load. The correct command to load the boot sector of **A:** at **cs**:0100 is therefore

```
L 100 0 0 1
```

Once the data from the boot sector are in memory, it is probably a good idea to store them on disk. This requires three commands:

n *fileName*

which supplies a filename for the forthcoming write,

rcx *Number*

which places the number of characters to write in the **cx** register, and

w *Address*

which writes the prescribed number of characters, starting at *Address*, to *fileName*. Again if no segment is specified in *Address*, the default is **cs**.

In the present case, the commands might look like

```
n bootSect
rcx 200
w 100
```

The first 100H bytes in the sector can be displayed on the screen exactly as in **CODEVIEW**:

```
d 100 L 100
```

If the user does not wish to see these bytes on the screen, but only have them available in a file, then all the previous operations can be accomplished in one line, using redirection of output:

```
L 100 0 0 1; > bootSect; d 100 L 100; > con
```

The **u** (*unassemble*) command works like the **CODEVIEW** version. Studying our example in Figure 16.2 for a moment shows that the actual boot code begins at offset 36H (the target of the initial jump) and ends at offset 16DH (since everything after that is just text). A reasonable u-command in that case would be

```
u 36 16d
```

It is also a good idea to save the unassembled listing to disk, and this is most easily done using redirection:

```
> bootSect; u 36 16d;  > con
```

Finally, we can state the problems.

Exercise 16-1. Start with any bootable DOS disk and make an unassembled listing of the boot code, as outlined previously. Then trace through the code by hand. It should be possible to make sense of nearly all of it. Repeat the process with a non-bootable disk.

Exercise 16-2. Write a procedure (perhaps for installation in USER.LIB) that will read a chosen number of sectors from a disk into a buffer, beginning with any desired sector number. **Suggestion:** Use either **DOS int 25H** or **BIOS int 13H**. These interrupts are both described in Appendix 2. However, **int 13H** requires some more explanation.

A disk drive contains a number of *platters*, or separate disks, each divided into concentric *tracks*, and each track is further divided into *sectors*. In most modern disk formats, data is recorded on both sides of the platters, hence each platter has two *heads* associated with it. In the description of **int 13H**, the word *cylinder* is used as a synonym for track. A double-sided floppy disk format then corresponds to one platter and two heads.

In the ROM BIOS terminology, the head and cylinder numbering begin with 0, but the track numbering begins with 1! (In the MS-DOS numbering system, the sector numbering starts with 0 and increases linearly throughout the disk.) If we represent the disk drive parameters as C/H/S (cylinders/heads/sectors), then a 360K floppy is a 40/2/9, a 720K floppy is an 80/2/9, a 1.2M floppy is an 80/2/15, a 1.44M floppy is an 80/2/18, and many hard disks have the form ?/?/17. To translate the MS-DOS linear sector numbering into the ROM BIOS C/H/S form, it is also necessary to know the order in which the sectors are accessed. The order used corresponds to 'S' varying most rapidly and 'C' varying least rapidly.

Exercise 16-3. In the first display in Figure 16.2 we gave a primitive hex dump of the start of the boot sector and in the second display we showed a nicely formatted listing of the various fields of interest. Write a utility that will produce output of this second type for any DOS formatted disk.

Exercise 16-4. In Section 3, we outlined the structure of a FAT. The only way to really understand that discussion is to trace through part of one, as we did in Figure 16.3. This is much easier to do in the 16-bit FAT case than in the 12-bit case.

a. Write a utility that will read a 12-bit FAT from a disk and convert the entries into hexadecimal integers, formatting the results as a column whose typical entry has the form

```
FAT entry number: value contained in this entry.
```

b. Prepare a floppy containing a badly "fragmented" file as follows. Begin with a freshly formatted floppy, and fill it with small files. Randomly erase some of these files so that the disk data have "holes." Then write a large text file to the disk. Use a text file so that, at the end of the exercise, you will know whether you have successfully found its data.

c. Using the utility written in part a, find the sequence of cluster numbers for the text file. The special meanings associated with some FAT values were given in Table 16.2.
d. Read the data from these clusters into another file, or a sequence of files, and check that you have indeed found all of the original text file.

Exercise 16-5. (A lengthy problem!) Write a utility that will attempt to recover deleted files from a disk. In response to the command line

```
undel fileName
```

it should do its best to find the data from the deleted file named *fileName* and restore its directory entry and cluster linkage in the FAT. It is sufficient to write the program so that it works with 12-bit FATs on floppies, but it is easier, more useful, and also more dangerous to develop it using 16-bit FATs so that it will work on hard disks. The choice is yours — perhaps you would like to do a version that can handle both sizes of FAT.
Suggestions: Recall from our discussion of Figure 16.3 that a directory entry is not erased until it is reused. With luck, it will still be there, intact, except that the first byte will have been changed to 0E5H. Hence it is (relatively) easy to find the directory entry and fix the first letter. At this point, we will also have access to the starting cluster number and the file size, which tells us the total number of clusters. The next step seems unreasonably optimistic, and its only justification is that it works most of the time. Assume that the clusters whose entries in the FAT contain zero (i.e., are available) and that follow the starting cluster and are closest to it contained the remainder of the file. Patch these FAT entries so that they have the correct linkage and hope for the best.

If you feel especially industrious, it is possible to improve the utility by giving it a "manual" mode in which it makes a screen dump of each promising cluster, letting the user decide whether to include it or look further.

SECTION 4 THE DEVICE-INDEPENDENT CODE

The structure of installable device drivers is quite strictly mandated by **DOS**. On the one hand, this means that the programmer must know the rules completely in order to write a working driver but, on the other hand, many parts of the driver are trivial to write, since the coding amounts to little more than following a template.

Although it is not strictly necessary, we prefer to split the code for a device driver into two parts. The first part contains code that is the same for all device drivers, and so need not be rewritten for each new driver. The easiest way to combine this code with the second, device-dependent code is by means of **MASM's INCLUDE** directive. Figure 16.4 shows our version of the device-independent code.

Before talking about the code in Figure 16.4 in detail, we will briefly discuss the general structure of device drivers, keying our discussion to the definitions given in Figure 16.4.

First, as mentioned in Section 2, device drivers are basically .COM programs, except that they begin at offset zero in the code segment. They are written and prepared like other

.COM programs: they must consist of a single segment (or group), execution must begin at the beginning of that segment (or group), and they must be processed by **EXE2BIN.**

If a hardware device is associated with an installable device driver, it can automatically be controlled by a number of the **int 21H** functions. The file handle functions are probably the most important, but there are others. Examples are the disk directory functions (if it is a block device) and **function 44H** of **int 21H**, which is called *I/O Control for Devices* and will be discussed later.

When an application program uses an **int 21H** function to access a device, **DOS** encapsulates the essential information about the function call in a data structure called a *Request Header* and calls the *Strategy Routine* of the device driver (starting on line 106 of Figure 16.4). The Strategy Routine is passed the address of the Request Header in **es:bx** and should save that address.

Next **DOS** calls the *Interrupt Routine* (starting on line 116 of the driver code). The Interrupt Routine is basically a switching station that directs the service request to the correct *device driver command* and returns a completion code afterward.

The name "Interrupt Routine" is not well chosen — it suggests that perhaps calls to this procedure are asynchronous in the same way that hardware interrupts are. This is not the case because this procedure is called by **DOS** when required. Since the two routines, Strategy and Interrupt, are always called sequentially, it may seem artificial to make them separate procedures, and indeed it is. The standard explanation is that the formalism is set up in this way so that, should **DOS** ever become multi-tasking, the Strategy Routine is already in place to do something more useful, namely, to arbitrate in the scheduling of overlapping calls to the Interrupt Routine.

The actual commands to which the device driver may respond are listed on lines 74-98. As the comments indicate, some of these commands are intended for character devices and some for block devices. In fact, a typical device driver will support a relatively small subset of these commands. The next section will be devoted to a discussion of the most important (from our viewpoint) of the device driver commands.

Now we will look at some of the coding details of Figure 16.4:

Line 1. The semicolon before the **TITLE** directive is *not* a typo. We are going to **INCLUDE** this code as part of another module and consequently the present line 1 will occur at a later position in the combined file. However, **MASM** only allows the **TITLE** directive to occur on line 1 and so the simplest expedient was to comment it out.

Line 9. In all the device driver commands, **es:di** points to the Request Header, and we can make the code more readable by supplying a suggestive name for this quantity by means of an **EQUATE**. The only weakness of this approach is that *reqHdr*, the chosen name, is an anonymous reference and so, depending on the other operand of an instruction, may require a size override. It is not practical to supply a size override as part of the **EQUATE** name since it is sometimes used as a byte and sometimes as a word.

Lines 11-30. Each device driver has an *attribute word* (defined later by a **DW**), and these **EQUATE**s explain the meanings of the significant bits in that word. Notice that the early bits are set only for the various resident drivers. The middle bits indicate whether or not a driver supports some particular functions. Bit 15 is obviously important.