www.mohandesyar.com

عنـــــــوان

آشنایی با برنامه نویسی اسمبلی ویندوز

پژوهش و نگارش : وحید نصیری

بهار ۱۳۸۴

قسمت اول

نگارش ۱

مقدمه :

اکثر برنامه نویسان از اسمبلر (و یا زبان اسمبلی) گریزان هستند. دلیل خود را هم سختی درک و کاربرد آن ذکر می‌کنند. در مقابل کسانی هم که به اسمبلی تسلط دارند، به شدت مورد تکریم و احترام گروه‌های برنامه نویسان واقع می‌شوند. هدف از انتشار این مطالب طرد این طرز فکر منفی در مورد اسمبلر است.

در ابتدا شاید این سوال مطرح شود که اسمبلر چیست؟ اگر بخواهیم آنرا بسیار ساده تعریف کنیم ، می‌توان گفت اسمبلر زبان پردازشگر است و سطح پایین‌تری از آنرا نمی‌توان تصور کرد (البته بجز مقادیر بایتی دستورالعمل‌ها و اینستراکشن‌ها). هر دستور آن بوسیله برنامه اسمبلر مستقیما به اعدادی که توسط پردازشگر قابل اجرا است ترجمه می‌شود.

مزیت استفاده از اسمبلر بر سایر زبانهای برنامه نویسی، سرعت است. سرعتی محض، خام و تخفیف نیافته. حتی با بهینه سازی‌هایی که کامپایلرهای مدرن سایر زبانهای برنامه نویسی انجام می‌دهند، کد اسمبلری که با دست بهینه سازی شده است بسیار بسیار سریعتر اجرا می‌شود.

امروزه از اسمبلر برای نوشتن هر نوع برنامه‌ای استفاده نمی‌شود. هر چند می‌توان کل یک برنامه را با اسمبلر نوشت اما با وجود زبانهای سطوح بالاتری مانند C++ و غیره، شاید انجام اینکار نوعی مازوخیسم به‌نظر آید! و باید در نظر داشت که برای بسیاری از کاربردها سرعت C++ و یا حتی زبانهای دات نت قابل قبول هستند.

زمانی استفاده از اسمبلر حائز اهمیت خواهد شد که سرعت انجام عملیات جزو موارد بحرانی محسوب گردد. برای مثال در برنامه های گرافیکی و اعمال مختلف با بیت‌مپ‌ها.

نوشتن برنامه اسمبلی در VC++:

در C++ با استفاده از قطعات کدی که با __asm مشخص می‌شوند می‌توان به زبان اسمبلی برنامه نوشت. برای مثال:

```
DWORD Function(DWORD dwValue)
{
   __asm
   {
      mov eax, dwValue
      add eax, 100
      mov dwValue, eax
   }

   return dwValue;
}
```

در اینجا شما تعدادی اینستراکشن اسمبلر را قرار گرفته در یک قطعه __asm ملاحظه می کنید. نگران معنای قسمت‌های مختلف آن نباشید. فقط بدانید که کار اینستراکشن mov انتقال داده‌ها است و یا add برای افزودن داده‌ها مورد استفاده قرار می‌گیرد. کد فوق تنها عدد ۱۰۰ را به آرگومان ورودی افزوده و حاصل را بر می‌گرداند.

تا اینجا نوشتن چند خط کوتاه از کد داخل قطعه‌ای __asm مشکل نیست. اما مثال زیر را در مورد بکار گیری if..else در نظر بگیرید:

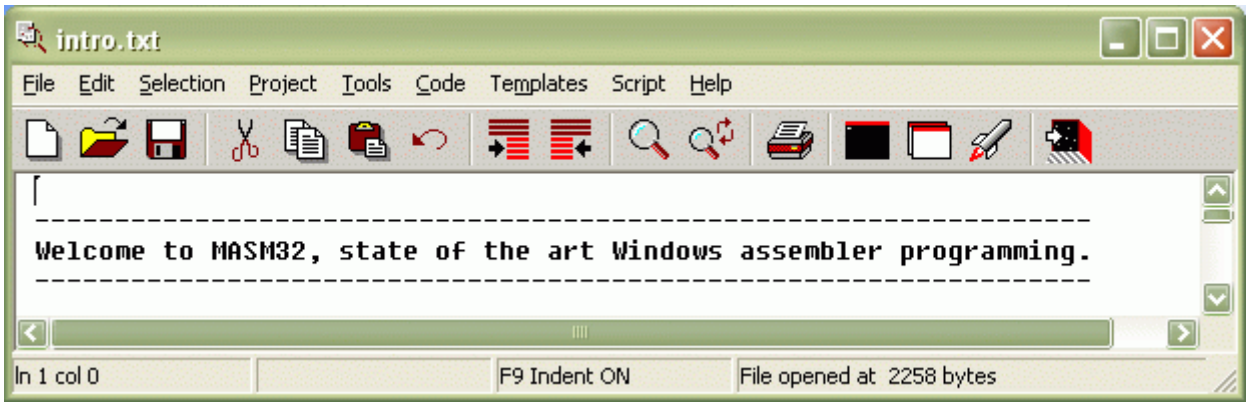
```
DWORD Function2(DWORD dwValue1, DWORD dwValue2)
{
   DWORD dwValue3 = 0;

#if 0
   // this is the Assembler
   if (dwValue1 == dwValue2)
      dwValue3 = 1;
   else
      dwValue3 = 2;
#endif

   __asm
   {
      mov eax, dwValue1
      ; this is the test of the values, i.e. if dwValue1 == dwValue2
      cmp eax, dwValue2
      ; jump to 'Else' if they are not equal
      jne Else
      mov eax, 1
      jmp EndIf

   Else:
      mov eax, 2

   EndIf:
      mov dwValue3, eax
   }

   return dwValue3;
}
```

بازهم در مورد دستورالعمل های اسمبلر بکار گرفته شده نگران نباشید. اما واقعا باید برای پیاده سازی هر if..else ساده این مقدار کد نوشت؟

استفاده از قطعات asm__ برای کد نویسی اسمبلر همانند استفاده از Notepad و دستورات خط فرمان برای برنامه نویسی C++ بجای استفاده از IDE قدرتمند VC++ می‌باشد. برای برنامه نویسی اسمبلر نیز Microsoft Macro Assembler موجود است. این اسمبلر از سال ۱۹۸۰ ارائه شده و مجانی می‌باشد. همچنین قابلیت تولید فایلهای obj. قابل استفاده در VC++ را نیز دارا است. امروزه نام MASM32 را یافته و در آدرس زیر قابل دریافت است:

http://www.masm32.com/

قویا توصیه می شود قبل از ادامه مطالعه مطالب ، آنرا دریافت و نصب کنید.



شکل ۱– نمایی از MASM32

خوب! این اسمبلر دارای چه مزایایی است؟

- دارای راهنمایی بسیار جامع ، قرار گرفته در مسیر msasm32\help\ می‌باشد.
- ثانیا حاوی macro های فراوانی جهت سهولت برنامه نویسی اسمبلی است.

برای مثال اگر کد قبلی را بخواهیم در MASM بنویسیم نتیجه به شکل زیر خواهد بود:

```
Function proc dwValue1:DWORD, dwValue2:DWORD
   mov eax, dwValue1
   .if eax == dwValue2
      mov eax, 1
   .else
      mov eax, 2
   .endif
   ret
Function endp
```

به طور قطع موافقید که کد فوق بسیار خواناتر می‌باشد.

آشنایی با زبان اسمبلی

زبان اسمبلی جهت جایگزینی کد باینری خامی که یک پردازشگر می‌تواند آنرا درک کند، بوجود آمده است. سالها قبل ، زمانیکه زبانهای برنامه نویسی سطح بالا وجود نداشتند، برنامه‌ها به زبان اسمبلی نوشته می‌شدند. زبان اسمبلی مستقیما بیانگر دستوراتی است که یک پروسسور می‌تواند اجرا کند. برای مثال :

```
add eax, edx
```

اینستراکشن فوق دو مقدار را با هم جمع می‌کند. EAX و EDX رجیستر هستند، حاوی مقادیر بوده و درون پردازشگر ذخیره می شوند. کد فوق نهایتا به 66 03 C2 ترجمه می شود (در مبنای شانزده). پردازشگر آنرا خوانده و دستورات مربوطه را اجرا می کند. زبانهای سطح بالاتری مانند C ابتدا کد خود را به اسمبلی تبدیل و نهایتا آنرا به کدهای باینری ترجمه خواهند کرد:

C code
```
a = a + b;
```
>> Compiler > >
Assembly
```
add eax, edx
```
>>Assembler>>
Raw output (hex)
```
66 03 C2
```

به اینستراکشن‌ها Opcodes هم گفته می‌شود. Opcode ، اینستراکشنی است که یک پردازشگر می‌تواند درک کند.

تفاوت مهم بین اسمبلی داس و ویندوز:

برنامه های تحت داس از وقفه‌ها (interrupts) بعنوان "تابع" استفاده می کنند. برای مثال int 13 برای کار با فایلها و غیره. در ویندوز با API سر و کار داریم (Application Programming Interface). این اینترفیس حاوی توابعی است که شما می توانید در برنامه‌های خود از آنها استفاده نمایید. سایر تفاوت‌ها نیز به مرور در ادامه ذکر خواهند شد.

متغیرها:

در اسمبلر چیزی همانند متغیرها در C++ وجود خارجی ندارند. در اسمبلر، آدرس‌های حافظه و رجیسترها در اختیار شما هستند. برای مثال پروسسور اطلاعی در مورد وجود متغیری با نوع صحیح به نام nMyInteger ندارد همچنین در مورد بکارگیری کلاسی به نام CMyClass نیز باخبر نیست. پروسسور تنها چیزی را که می‌داند در مورد رجیسترها و یا دسترسی به مقداری از حافظه در آدرسی مشخص است.

خوب! رجیستر چیست و چگونه می توان به حافظه دسترسی پیدا کرد؟

رجیسترها:

رجیستر همانند یک متغیر است اما تنها برای استفاده پردازشگر بکار می‌رود. تنها مجموعه‌ی مشخصی از رجیسترها در تراشه‌ی یک پردازشگر وجود دارند. می‌توان یک رجیستر را به این صورت نیز تعریف کرد : یک رجیستر متغیری است از پیش تعریف شده در پردازشگر و به صورت فیزیکی در تراشه آن موجود است.

این رجیسترها می‌توانند اعدادی را هم اندازه‌ی تعداد بیتی که یک پردازشگر در هر لحظه می‌تواند پردازش کند ، نمایش دهند. برای مثال در پردازشگری ۳۲ بیتی ، اندازه این اعداد ۳۲ بیت خواهد بود. در C++ به آنها DWORD نیز گفته می‌شود.

نمایش سه اندازه داده بومی پردازشگرهای ۳۲ بیتی اینتل و یا سازگار با آن در حالت هگز:

BYTE 00
WORD 00 00
DWORD 00 00 00 00

همچنین این اعداد، unsigned هستند (بدون علامت و همواره مثبت). در اینجا اگر به اعداد منفی نیاز شد ، منفی یک معادل 0xFFFFFFFF است و منفی دو معادل 0xFFFFFFFE بوده و الی آخر (در مورد اعداد درآینده بیشتر صحبت خواهد شد).

در پروسسورهای جدید اینتل رجیسترهای چندی از پیش تعریف شده هستند اما شما تنها ۶ عدد از آنها را باید در برنامه‌های خود بکار بگیرید:

```
eax - Accumulator Register
ebx - Base Register
ecx - Counter Register
edx - Data Register
esi - Source (for memory operations) register
edi - Destination (for memory operations) register
```

هر رجیستر در اینجا قابل تجزیه به بایتهای سازنده آن است. برای مثال در مورد EAX داریم:

al : اولین بایت (پایینی) در کلمه (word) پایینی ثبات (رجیستر) EAX = ۸ بیت

ah : دومین بایت (بالایی) در کلمه (word) پایینی ثبات (رجیستر) EAX = ۸ بیت

ax : کلمه (۲ بایت) پایینی در رجیستر EAX = ۱۶ بیت

EAX : کل رجیستر معادل ۴ بایت = ۳۲ بیت

0000	00	00
EAX		
	AX	
	AH	AL

0000	00	00
ECX		
	CX	
	CH	CL

0000	00	00
EBX		
	BX	
	BH	BL

0000	00	00
EDX		
	DX	
	DH	DL

همانطور که ملاحظه می نمایید چنین تقسیم بندی در مورد ECX ، EBX و EDX نیز وجود دارد (و البته نه برای ESI و EDI).

این نامگذاری‌ها از نوع پروسسور مشتق شده است. 'E' در اینجا به معنای Extended است. پروسسورهای ۱۶ بیتی تنها دارای رجیسترهایی مانند ax, bx, cx, dx و غیره بودند. هنگامیکه این نوع پروسسورها توسعه داده شدند ، ۱۶ بیت اضافی را با E مشخص کردند (و تنها در پردازشگرهای ۳۸۶+ مهیا هستند). هیچ روش مستقیمی برای دسترسی به ۱۶ بیت بالایی رجیسترها موجود نیست.

برای مثال اگر EAX حاوی EA7823BBh باشد ، اجزای کوچکتر آن حاوی مقادیر مشخص شده خواهند بود:

EAX	EA	78	23	BB
AX	EA	78	23	BB
AH	EA	78	23	BB
AL	EA	78	23	BB

eax = EA7823BB (32-bit)
ax = 23BB (16-bit)
ah = 23 (8-bit)
al = BB (8-bit)

در حالت کلی انواع مختلفی از رجیسترها وجود دارند:

الف) رجیسترهای عمومی که برای هرکاری می توانند مورد استفاده قرار گیرند:

eax (ax/ah/al)	Accumulator
ebx (bx/bh/bl)	Base
ecx (cx/ch/cl)	Counter
edx (dx/dh/dl)	Data

ب) رجیسترهای سگمنت که سگمنت حافظه بکار گرفته شده را مشخص می کنند:

CS	code segment
DS	data segment
SS	stack segment
ES	extra segment
FS (only 286+)	general purpose segment
GS (only 386+)	general purpose segment

این سگمنت‌ها در ویندوز کاربردی ندارند. زیرا ویندوز یک flat memory system می‌باشد. تحت داس حافظه به سگمنت‌هایی 64Kb ایی تقسیم شده بود. بنابراین برای مشخص کردن یک آدرس حافظه ، نیاز به ذکر سگمنت و آفست بود . برای مثال :

0172:0500 (segment:offset)

در ویندوز سگمنت‌ها ۴ گیگ حجم دارند. بنابراین نیازی در بکارگیری آنها نیست. سگمنت‌ها همواره رجیسترهایی ۱۶ بیتی هستند.(در مورد سگمنت‌ها در ادامه در قسمت معماری حافظه بیشتر توضیح داده خواهد شد.)

ج) رجیسترهای اشاره گر:

esi (si)	Source index
edi (di)	Destination index
eip (ip)	Instruction pointer

از این نوع رجیسترها بعنوان رجیسترهای عمومی تا زمانیکه مقادیر اصلی آنها حفظ گردند نیز می توان استفاده کرد (البته بجز eip) . دلیل نامگذاری آنها نیز استفاده از آنها به جهت ذخیره سازی آدرس های حافظه است. EIP حاوی اشاره گری است به اینستراکشنی که قرار است اجرا شود.بنابراین از آن بعنوان رجیستری عمومی نمی توان استفاده کرد.

د) رجیسترهای پشته (stack)

esp (sp)	Stack pointer
ebp (bp)	Base pointer

ESP موقعیت فعلی پشته را نگه می دارد.EBP در توابع بعنوان اشاره گر به متغیرهای محلی استفاده می شود.

نگاهی بر معماری حافظه در ویندوز و داس:

الف) DOS و WIN 3.xx

در داس و برای مثال ویندوز سه و یک دهم (در برنامه های ۱۶ بیتی) ، حافظه به قطعاتی (سگمنت هایی) تقسیم شده است. این سگمنت‌ها اندازه‌ای 64Kb ایی دارند.در این حالت برای دسترسی به حافظه نیاز به اشاره‌گرهای سگمنت و آفست می باشد.

تقسیم بندی حافظه به قطعات:

MEMORY				
SEGMENT 1 (64kb)	SEGMENT 2 (64kb)	SEGMENT 3 (64kb)	SEGMENT 4(64kb)	and so on

نمایش آفست‌ها در یک سگمنت (آفست مکانی است درون یک سگمنت):

SEGMENT 1(64kb)					
Offset 1	Offset 2	Offset 3	Offset 4	Offset 5	and so on

در اینجا حداکثر ۶۵۵۳۶ سگمنت و همچنین درون یک سگمنت حداکثر ۶۵۵۳۶ آفست می‌تواند وجود داشته باشد.
در این حالت نمایش یک آدرس حافظه به صورت زیر خواهد بود:

SEGMENT:OFFSET

برای مثال :

0030:4012 (all hex numbers)

یعنی سگمنت ۳۰ و آفست ۴۰۱۲ در سگمنت ذکر شده .

در قسمت قبل با رجیسترهای سگمنت آشنا شدید. در ادامه توضیحات بیشتری در مورد آنها را خواهید یافت:

- CS و یا Code segment : حاوی شماره قطعه ای که کد جاری درحال اجرا است.
- DS و یا Data segment : سگمنت داده ای که قطعه جاری داده های خود را از آن تامین می کند.

رجیسترهای اشاره‌گر در اغلب اوقات حاوی آفست ها هستند (همچنین رجیسترهای عمومی نیز می توانند برای این منظور بکار برده شوند). IP بیانگر آفست اینستراکشنی است (در CS) که هم اکنون در حال اجرا است.SP نگهدارنده ی آفست موقعیت فعلی پشته(stack) است (در SS).

ب) ویندوز ۳۲ بیتی

در ویندوزهای ۹۵ و یا بالاتر نیز سگمنت ها وجود دارند اما اندازه‌ی آنها ۴ گیگابایت است و البته با تغییر رجیسترهای سگمنت ، ویندوز احتمالا دچار اختلال شدید در کارکرد خواهد شد و از کار می‌افتد.به این حالت flat memory model گفته می‌شود.تنها آفست‌ها وجود داشته و اکنون ۳۲ بیتی هستند یعنی دربازه صفر تا ۴۲۹۴۹۶۷۲۹۵ می‌توانند قرارگیرند.بنابراین هر موقعیت در حافظه تنها با یک آفست مشخص می شود.این مورد یکی از مهمترین مزیت‌های سیستمی ۳۲ بیتی نسبت به سیستمی ۱۶ بیتی است.بنابراین اکنون می‌توان رجیسترهای سگمنت را فراموش کرد و بر روی سایر رجیسترها متمرکز شد.این مورد سبب ساده تر شدن برنامه نویسی اسمبلی و همچنین بالا رفتن سرعت اجرای کد می گردد.

اینگونه برنامه‌ها لزاما باید بر روی پردازنده‌های ۳۸۶ و یا بالاتر اجرا شوند.

برای برنامه نویسانی که تحت داس برنامه نویسی اسمبلی انجام داده اند فرمت فایلهای اجرایی ویندوز از جهاتی شبیه به فایلهای com داس می باشند ، دارای یک سگمنت که شامل کد و داده است بوده و هر دو مستقیما با آفست‌ها کار می‌کنند و نیز از روش سگمنت/آفست برای آدرس دهی کمک نمی‌گیرند.

حالت پیش فرض آدرس دهی در برنامه‌های مدل تخت (flat) ، از نوع آدرس دهی NEAR در بازه ۴ گیگابایت است.

آشنایی با حالت حافظه‌ی محافظت شده (Protected Mode Memory) :

در سیستم عاملهایی با خصوصیت real mode مانند سیستم عامل داس ، امکان جای نویسی (overwrite) قسمت هایی از کد سیستم عامل توسط برنامه‌ای که صحیح نوشته نشده باشد میسر است.این امر در صورت وقوع سبب از کار افتادن سیستم عامل خواهد گردید و نهایتا نیاز به بوت مجدد می‌باشد.

برای مثال اگر تحت داس مشغول برنامه نویسی مستقیم نواحی حافظه ویدیویی CGA که از B800 شروع می شوند باشید ، برنامه در اثر اشتباهی در یک حلقه می‌تواند کل ناحیه بالایی آنرا تا رسیدن به حداکثر بازه آدرس دهی در داس جای نویسی کند.این مورد سبب جای نویسی همه چیز منجمله تنظیمات BIOS در حافظه می گردد.

حالت محافظت شده‌ی حافظه جهت جلوگیری از وقوع این نوع رخدادها ایجاد شده است.مدیریت حافظه‌ی محافظت شده ، بازه‌ی آدرسی را که یک برنامه می تواند در آن بنویسد را کنترل کرده و در صورت بروز تخلفی از این بازه‌ی تعریف شده ، برنامه را خاتمه می‌دهد.

این مدل محافظت از حافظه ، در ویندوزهای ۱۶ بیتی نیز مهیا بود، اما بدلیل روش بکار گرفته شده در شبیه سازی multitasking در این نوع ویندوزها ، امکان جای نویسی قسمت‌هایی از حافظه که تحت مالکیت سایر برنامه‌ها بودند ، نیز میسر بود.

تغییرات حاصل شده در سخت افزارها و پیاده سازی multitasking بر مبنای سخت افزار در ویندوزهای ۳۲ بیتی ، مدیریت حافظه‌ی محافظت شده را قابل اطمینان تر کرده است.اساس برنامه نویسی در سیستم عاملی با حالت محافظت شده‌ی حافظه ، تنها خواندن و یا نوشتن در حافظه‌ای است که برنامه به آن دسترسی دارد.

با توجه به اینکه اسمبلر تقریبا اجازه‌ی خواندن و نوشتن در هر بازه‌ی آدرسی را می‌دهد شما باید بادقت بیشتری آنرا بکار گیرید.برای مثال اگر شما بافری به اندازه ی 10k را برای خواندن تخصیص داده باشید و سعی در خواندن 20k نمایید ، با خطای page read fault مواجه خواهید شد.این نوع خطاها از طرف سیستم عامل بعنوان یک exception به برنامه گوشزد شده و اگر برنامه راهی را برای مواجه با آن پیش بینی نکرده باشد ، از طرف سیستم عامل خاتمه می‌یابد.

آشنایی با Opcodes :

Opcodes اینستراکشن‌های پردازشگر هستند.در حقیقت اینها نگارش "متنی قابل خواندن" کدهای خام هگز هستند.به همین دلیل اسمبلی ، سطح پایین ترین زبان برنامه نویسی است که وجود دارد (هر دستور آن مستقیما به کدهای هگز تبدیل می‌شود). به بیانی دیگر فاز کامپایل و تبدیل زبانی سطح بالا به زبانی سطح پایین خود بخود حذف می گردد و تنها کاری که در اینجا صورت می گیرد تبدیل کدهای اسمبلر به داده‌های خام هگز است.
در سطح سخت افزاری پردازشگرهای اینتل و یا سازگار با آن ، دستورالعمل ها (اینستراکشن‌ها) درون مدرات مربوطه قرار گرفته‌اند و به آنها Opcodes گفته می‌شود.

برای مثال برای کپی مقداری ۳۲ بیتی مساوی **56 A7 00 FE** به درون رجیستر EAX ، از Opcode مساوی A1 استفاده می‌شود (Mov EAX). بنابراین نتیجه نهایی مساوی است با :

A1 FE 00 A7 56

در اینجا جهت ساده‌تر کردن برنامه نویسی اسمبلی به گروهی از Opcodes مشابه نامهایی اعطاء گردیده تا در استفاده از آنها نیازی به فراگیری کدهای هگز نباشد.به این نامها mnemonics گفته می‌شود.
یک mnemonic نامی است اختصاص داده شده به خانواده‌ای از Opcodes که کاری مشابه را در پردازشگر انجام می‌دهند.

معرفی تعدادی Opcodes محاسباتی:

MOV

این اینستراکشن برای کپی مقداری از یک مکان به مکانی دیگر بکار می رود وکار آن ، انتقال داده‌ها درون پروسسور می‌باشد. برای مثال:

```
mov eax, 100
```

در اینجا ۱۰۰ به رجیستر EAX انتقال داده می شود. خط فوق را می توان معادل eax=100 دانست.به اینگونه اعداد، Immediate numbers نیز گفته می‌شود.

حالت کلی این دستورالعمل به شکل زیر است و مبدا و مقصد باید هم اندازه باشند:

```
mov (destination), (source)
```

برای مثال دستور زیر نادرست است (اندازه مبدا و مقصد یکی نیست):

```
mov al, ecx ; NOT VALID
```

مثالهایی بیشتر در این زمینه:

```
mov al, bl          ;move the lower byte of ebx into the lower byte of eax
mov al, 0ffh        ;move 0xFF into the lower byte of eax
mov ah, 0ffh        ;move 0xFF into the high byte of the low word (2-bytes) of
                    eax
mov ax, 0ffffh      ;move 0xFFFF into the low word of eax
mov eax, 0ffffh     ;move 0xFFFF into eax
mov eax, 12345678h  ;mov loads a value into a register (note: 12345678h is a
                    ;hex value because of the 'h' suffix)
mov cl, ah          ;move the high byte of ax (67h) into cl
sub cl, 10          ;substract 10 (dec.) from the value in cl
mov al, cl          ;and store it in the lowest byte of eax.
```

دریافت آدرس و محتوای آدرسی در حافظه:

می‌توان محتوای حافظه را نیز با استفاده از [] (که به آن "محتوای" نیز گفته می‌شود) به یک رجیستر و یا برعکس منتقل نمود :

```
mov al, [esi]       ;move the byte contained in the memory address
                    ;in register esi into the lower byte of eax
mov [edi], bl       ;move the byte value in the lowest byte of ebx
                    ;into the memory address in register edi
mov cx, [esi]       ;move the word (2-byte) value contained in the
                    ;memory address of register esi into the lower
                    ;word of ecx
mov [edi], edx      ;move the dword (4-byte) value contained in edx
                    ;into the memory address contained in register edi
```

همچنین می‌توان در اینجا آفست را نیز در نظر گرفت:

```
mov al, [esi + 3]   ;move the byte contained in the memory address
                    ;in register esi + 3 into the lower byte of eax
mov [edi + 2], dx   ;move the lower word (2-bytes) contained in
                    ;edx into the memory address contained in the
                    ;register edi + 2
```

برای درک موارد فوق ، مثال جدول زیر را در نظر بگیرید:

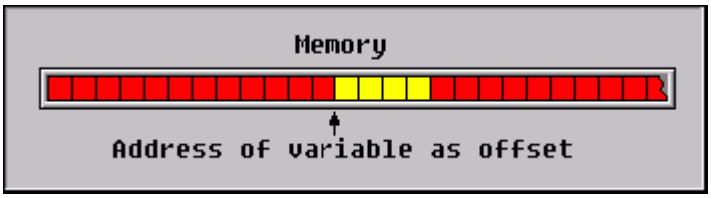
offset	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	40	41	42
data	0D	0A	50	32	44	57	25	7A	5E	72	EF	7D	FF	AD	C7

همانطور که پیش تر نیز ذکر شد در ویندوز موقعیت‌های حافظه با آفست مشخص می‌شوند.البته بدیهی است که مقادیر فوق ۳۲ بیتی هستند.برای مثال آفست 3A معادل با 0000003Ah است.برای دریافت محتوای آن می شود از دستور زیر استفاده کرد:

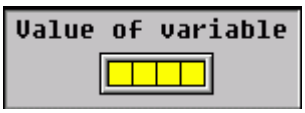
```
mov  eax, dword ptr [0000003Ah]
```

معنای خط فوق این است که مقدار ۳۲ بیتی آفست 3A را در رجیستر eax قرار بده.بعد از اجرای دستور فوق مقدار eax مساوی 725E7A25h خواهد بود.همانطور که ملاحظه نمودید ترتیب قرار گیری داده‌ها در حافظه به فرمت little endian است (ترتیب بایتها معکوس می‌شود.). مثال زیر این مطلب را روشن تر خواهد نمود:

مقدار ۳۲ بیتی 10203040 hex در حافظه به صورت 10, 20, 30, 40 ذخیره می شود و همچنین مقدار ۱۶ بیتی 4050 hex در حافظه به صورت 50, 40 ذخیره می‌شود.



شکل ۲– آدرس یک متغیر در حافظه (جایی که در حافظه واقع شده است).



شکل ۳– مقدار آدرس ذکر شده در حافظه.

به عملیات زیر **dereferencing** نیز می‌گویند:

```
mov eax, lpvar ; copy address into eax
mov eax, [eax] ; dereference it
mov nuvar, eax ; copy eax into new variable
```

همچنین دریافت مقادیر حافظه در اندازه‌های مختلف نیز میسر است:

```
mov  cl, byte ptr [34h] ; cl will get the value 0Dh (see table above)
mov  dx, word ptr [3Eh] ; dx will get the value 7DEFh (see table above, remember the reversed byte order)
```

ذکر اندازه مقادیر دریافتی الزامی نیست (به صورت پیش فرض ۳۲ بیت در نظر گرفته می‌شود). برای مثال:

```
mov  eax, [00403045h]
```

مثالی دیگر (به سایز نهایی منتقل شده دقت بفرمایید):

```
mov  eax, 403045h ;make eax have the value 403045 hex.
mov  cx, [eax] ;put the word size value at the memory location EAX (403045) into register CX.
```

بنابراین می‌توان نتیجه گرفت که mov ، نامی است اختصاص داده شده برای خانواده‌ای از Opcodes که کاری شبیه به هم را انجام می‌دهند.

اندازه متغیرهایی که می توان انتقال داد:

در حالت کلی نمی‌توان داده‌ای را در رجیستری که اندازه آن با اندازه داده اصلی متفاوت است قرار داد.برای مثال عبارت زیر نامعتبر است:

```
mov eax, cl ; this fails as eax is 32 bit, cl is 8 bit
```

برای اینکه بتوان cl را در یک رجیستر ۳۲ بیتی قرار داد باید ابتدا آنرا با استفاده از روش‌های مختلفی تبدیل کرد:

```
movzx  eax, cl ; zero extend unsigned integer
movsx  eax, cl ; sign extend signed integer
```

و یا :

```
Xor  eax, eax ; clear eax
mov  al, cl ; copy cl into al
```

و یا :

```
mov  al, cl ; copy cl into al
cbw ; convert BYTE in AL to WORD in AX
cwde ; convert WORD in AX to DWORD in EAX
```

نوع‌های داده‌ای که می‌توان در رجیسترها قرار داد:

سه نوع مقدار داده‌ای پایه را می‌توان در رجیسترها قرار داد : immediate ، حافظه و یا رجیستری دیگر.

عملوند Immediate معمولا یک عدد است اما می‌تواند یک رشته نیز باشد (که توسط اسمبلر به معادل اسکی خود تبدیل می‌شود). برای مثال:

```
mov  al, "a" ; string literal
mov  edx, 0 ; numeric immediate
```

و در مورد حالتهای دیگر پیش تر به تفصیل بحث شد. تنها باید دقت داشت که انتقال یک عملوند حافظه به دیگری میسر نیست. زیرا opecode ایی تعریف شده برای آن در پردازشگر وجود خارجی ندارد. بنابراین عبارت زیر نامعتبر است:

```
! mov mVar, lpMem ; this fails, no opcode to do it.
```

این عملیات را به صورت زیر می توان انجام داد:

```
! mov eax, lpMem ; copy memory value into register.
! mov mVar, eax ; copy register into memory value.
```

همچنین روش کندتری برای این منظور نیز مهیا است:

```
! push lpMem ; push memory value onto the stack.
! pop mVar ; pop it back off as another memory value.
```

اشاره‌گرها:

در اسمبلی هنگامیکه از دستورالعملی مانند خط زیر استفاده می شود:

```
Lea  eax, MyVar
```

آدرس متغیر در eax قرار می‌گیرد و سپس هنگامیکه eax به متغیر دیگری کپی می‌شود :

```
Mov  lpMyVar, eax
```

این متغیر اشاره‌گری خواهد بود به آدرس ذکر شده.

انتقال یک اشاره‌گر بوسیله مقدار آن (by value):

```
mov eax, lpMyVar ; copy the VALUE into eax
push eax ; push it as a parameter
call MyProcedure ; call the procedure
```

و یا انتقال یک اشاره‌گر بوسیله ارجاعی از آن (by reference) :

```
lea eax, lpMyVar ; load the ADDRESS into eax
push eax ; push it as a parameter
call MyProcedure ; call the procedure
```

در این حالت پس از خاتمه روال باید از روش زیر برای بدست آوردن آدرس استفاده کرد:

```
mov eax, lpMyVar
mov eax, [eax]
```

ADD, SUB, MUL, DIV

تعداد زیادی از Opcodes در محاسبات بکارگرفته می‌شوند. عملکرد بسیاری از آنها از نامشان قابل حدس زدن است. Add جهت جمع، sub جهت تفریق ، Mul جهت ضرب و div به منظور تقسیم بکار می روند.

نحوه استفاده از Add :

```
add  destination, source
```

در این حالت خواهیم داشت:

destination = destination + source

برای مثال :

Destination	Source	Example
Register	Register	add ecx, edx
Register	Memory	add ecx, dword ptr [104h] / add ecx, [edx]
Register	Immediate value	add eax, 102
Memory	Immediate value	add dword ptr [401231h], 80
Memory	Register	add dword ptr [401231h], edx

و برای تفریق و ضرب خواهیم داشت :

```
Sub  destination, source ;(destination = destination - source)
mul  destination, source ;(destination = destiantion * source)
```

عملیات تقسیم کمی متفاوت است (بسته به اندازه source ، خارج قسمت در eax و باقیمانده در edx قرار می گیرد):

div source ; (eax = eax / source, edx = remainer)

Source size	Division	Quotient stored in ...	Remainder Stored in...
BYTE (8-bits)	ax / source	AL	AH
WORD (16-bits)	dx:ax* / source	AX	DX
DWORD (32-bits)	edx:eax* / source	EAX	EDX

مبدا در این حالت یکی از موارد زیر می‌تواند باشد:

- an 8-bit register (al, ah, cl,...)
- a 16-bit register (ax, dx, ...)
- a 32-bit register (eax, edx, ecx...)
- an 8-bit memory value (byte ptr [xxxx])
- a 16-bit memory value (word ptr [xxxx])
- a 32-bit memory value (dword ptr [xxxx])

تذکر :

مبدا در این حالت یک عدد معمولی نباید باشد زیرا پردازشگر اندازه آنرا نمی‌تواند تعیین کند.

BITWISE OPERATIONS

تمام اینستراکشن‌های بیتی بر روی مقادیر مبدا و مقصد عمل می کنند (بجز Not) . روش عملیات نیز به صورت خلاصه مطابق جدول زیر است:

Instruction	AND				OR				XOR				NOT	
Source Bit	0	0	1	1	0	0	1	1	0	0	1	1	0	1
Destination Bit	0	1	0	1	0	1	0	1	0	1	0	1	X	X
Output Bit	0	0	0	1	0	1	1	1	0	1	1	0	1	0

مثال یک :

```
mov    ax, 3406
mov    dx, 13EAh
xor     ax, dx
ax = 3406 (decimal), which is 0000110101001110 in binary.
dx = 13EA (hex), which is 0001001111101010 in binary.
```

مثال دو:

```
mov ecx, FFFF0000h
not ecx
FFFF0000 is in binary 11111111111111110000000000000000 (16 1's, 16 0's)
If you take the inverse of every bit, you get:
00000000000000000111111111111111 (16 0's, 16 1's), which is 0000FFFF in hex.
So ecx is after the NOT operation 0000FFFFh.
```

IN/DECREMENTS

این اینستراکشن‌ها مقادیر مکانی از حافظه و یا یک رجیستر را یک واحد کم و یا زیاد می کنند.

```
inc   reg  -> reg = reg + 1
dec  reg -> reg = reg - 1
inc   dword ptr [103405]  -> value at [103405] will increase by one.
dec  dword ptr [103405] -> value at [103405] will decrease by one.
```

NOP

این اینستراکشن مطلقا کار خاصی را انجام نمی دهد! صرفا جهت پرکردن فضا و یا وصله کردن (patching) یک کد مورد استفاده قرار می گیرد.

Bit Rotation and Shifting

تذکر : اکثر مثالهای زیر از اعدادی ۸ بیتی ، صرفا جهت روشن شدن مطلب استفاده می کنند.

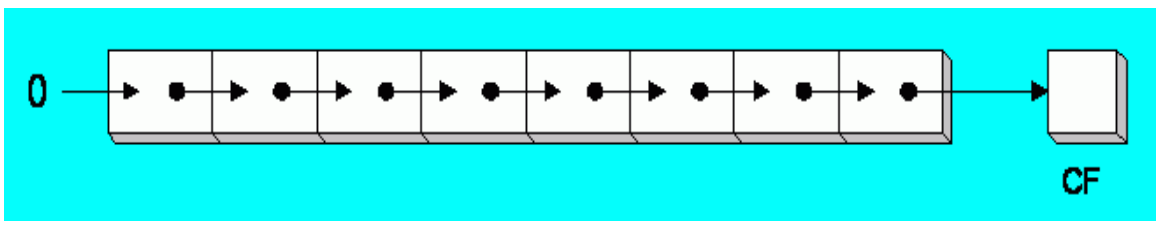
توابع شیفت:

Shifting functions
SHL destination, count
SHR destination, count

SHL و SHR به تعداد count ، بیت‌های یک مکان از حافظه و یا یک رجیستر را به چپ و راست انتقال می‌دهند.

مثال :

```
; al = 01011011 (binary) here
shr  al, 3
```



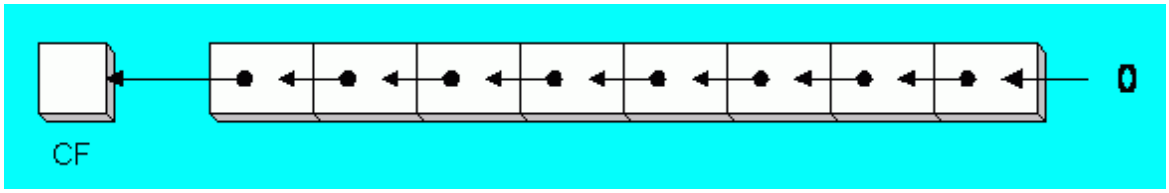
شکل ۴ – نحوه‌ی انجام SHR

بدین معنا که تمام بیت های al را سه مکان به سمت راست انتقال بده. بنابراین نتیجه معادل ۰۰۰۰۱۰۱۱ خواهد بود.بیت‌هایی که در سمت چپ قرار می گیرند با صفر پر شده و بیت‌های سمت راست، حذف خواهند شد. بیت‌هایی که حذف می گردند درcarry-flag ذخیره خواهند شد.carry-bit ، بیتی است در رجیسترهای Flag پردازشگر. این رجیسترها همانند رجیسترهای eax و غیره به صورت مستقیم قابل دسترسی نیستند (هر چند Opcodes مخصوص اینکار نیز وجود دارد). اما محتوای آنها وابسته است به نتایج اینستراکشن‌ها. این بیت معادل است با آخرین بیتی که حذف شده است.

shl همانند shr است اما با این تفاوت که بیت‌ها را به سمت چپ انتقال می دهد.

مثال :

```
; bl = 11100101 (binary) here
shl   bl, 2
```



شکل ۵– نحوه‌ی انجام SHL

پس از اجرای دستور فوق مقدار bl مساوی ۱۰۰۱۰۱۰۰ خواهد بود.بیت‌هایی که در سمت راست قرار گرفتند با صفر پرشده و carry-bit مساوی یک می‌باشد.

از SHL و SHR برای انجام ضرب و تقسیم های سریع نیز می توان استفاده کرد.شیفت یک مقدار به میزان n بیت به چپ مساوی است با آن مقدار ضربدر 2^n . برای مثال :

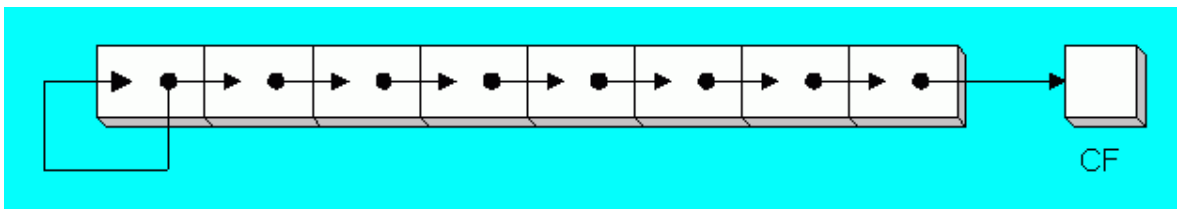
```
Mov   dl,5
Shl    dl,2    ; dl = 5 * 2^2 =20
```

همچنین شیفت یک مقدار به میزان n بیت به راست مساوی است با تقسیم آن مقدار بر 2^n . برای مثال:

```
mov   dl,80
shr    dl,1       ; DL = 40
shr    dl,2       ; DL = 10
```

دو Opcodes دیگر نیز در این زمینه وجود دارد:

```
SAL destination, count ;(Shift Arithmetic Left)
SAR destination, count ;(Shift Arithmetic Right)
```



شکل ۶– نحوه‌ی انجام SAR

SAL با SHL یکی می باشد. اما SAR دقیقا با SHR یکی نیست.SAR همیشه بیت‌های سمت چپ را با صفر پر نمی کند.اگر عدد منفی باشد ، بیت های سمت چپ با یک پر خواهد شد (در حقیقت با MSB و یا Most significant bit پر می‌شود).

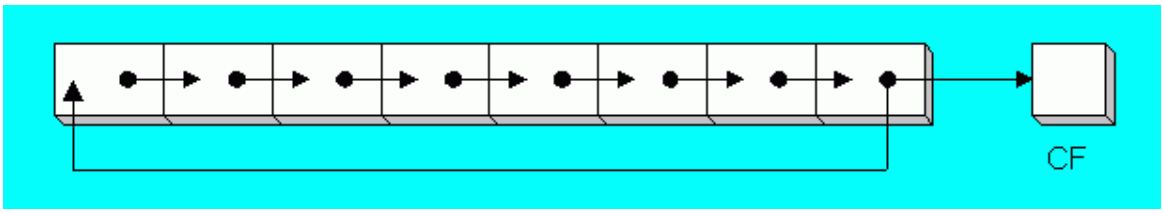
برای مثال اگر عدد ۸ بیتی منفی ۵ را ۲ بیت به سمت راست شیفت دهیم حاصل 00111110b خواهد شد. اما اگر بجای SHR از SAR استفاده کنیم حاصل نهایی 11111110 می‌شود.

توابع چرخش:

Rotation functions
rol destination, count ; rotate left
ror destination, count ; rotate right
rcl destination, count ; rotate through carry left
rcr destination, count ; rotate through carry right

توابع چرخش همانند توابع شیفت هستند اما بیت‌هایی که به بیرون رانده شده‌اند در جهتی دیگر به درون انتقال می‌یابند.

ROR :



شکل ۷– نحوه‌ی انجام ROR

مثالی در مورد ROR :

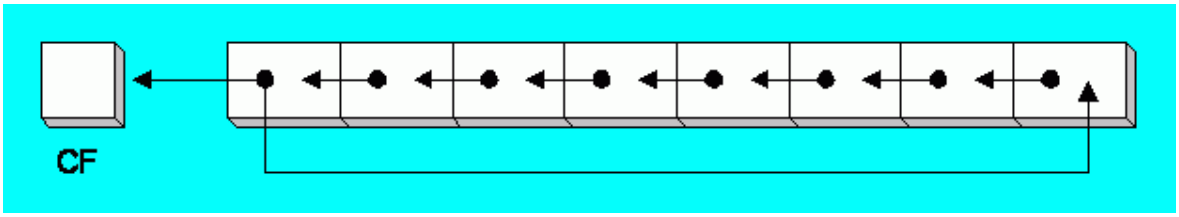
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
Before	1	0	0	1	1	0	1	1	
Rotate, count= 3				1	0	0	1	1	0 1 1 (Shift out)
Result	1	1	0	1	0	0	1	1	

همانطور که در جدول فوق مشخص است بیت ها دوران یافته است.هر بیتی که به بیرون شیفت داده شده از سمتی دیگر وارد گردیده است. همانند اینستراکشن های شیفت ، آخرین بیتی که به بیرون انتقال داده شده در carry-bit ذخیره می‌شود.

مثالی دیگر:

```
mov al,11110000b
ror al,1 ; AL = 01111000b

mov dl,3Fh
ror dl,4 ; Dl=F3h
```

ROL :



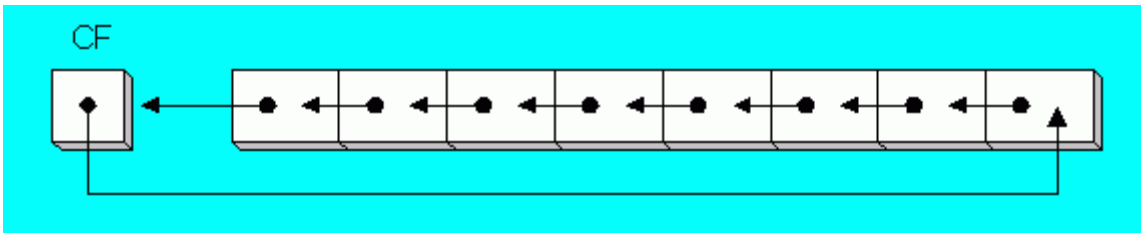
شکل ۸– نحوه‌ی انجام ROL

مثال :

```
mov al,11110000b
rol al,1  ; AL = 11100001b

mov dl,3Fh
rol dl,4  ; DL = F3h
```

RCR و RCL که در ادامه توضیح داده خواهند شد شبیه به ROR و ROL هستند اما از carry bit برای تعیین آخرین بیت خارج شده استفاده می کنند.

RCL :

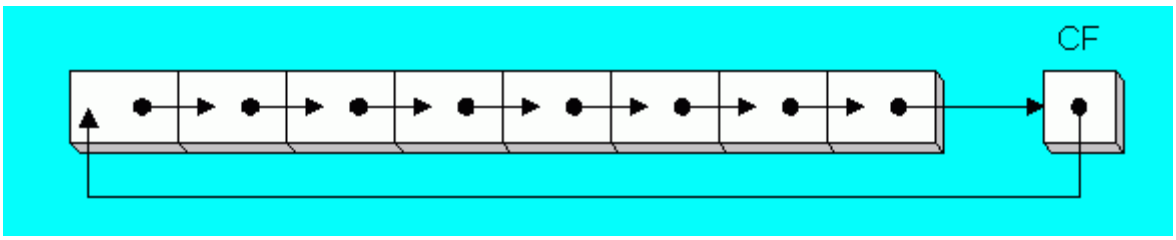


شکل ۹– نحوه‌ی انجام RCL

مثال:

```
clc             ; CF = 0
mov bl,88h      ; CF,BL = 0 10001000b
rcl bl,1        ; CF,BL = 1 00010000b
rcl bl,1        ; CF,BL = 0 00100001b
```

RCR :



شکل ۱۰– نحوه‌ی انجام RCR

مثال :

```
stc       ; CF = 1
mov ah,10h      ; CF,AH = 00010000 1
rcr ah,1 ; CF,AH = 10001000 0
```

مثالی دیگر:

رجیستر EAX را در نظر بگیرید. برای خواندن اولین بایت رجیستر به صورت زیر عمل می کنیم (بیت های صفر تا ۷):

```
mov byteval, al ; move 1st byte size value in to variable.
```

برای خواندن دومین بایت رجیستر داریم (بیت های ۸ تا ۱۵):

```
mov byteval, ah ; move 2nd byte size value into a variable.
```

برای خواندن اولین word آن (بیت های صفر تا ۱۵):

```
mov wordval, ax ; move 1st word size value into a variable.
```

برای خواندن بیت‌های ۱۶ تا ۳۱ رجیستر ، باید بیت‌های موجود در آن چرخانده شوند تا بتوان آنها را با استفاده از اینستراکشن‌های موجود استخراج کرد.برای مثال:

```
rol  eax, 16 ; rotate EAX by 16 bits
or
ror  eax, 16 ; rotate EAX by 16 bits
```

در این حالت ۱۶ بیت پایین در ۱۶ بالا قرار گرفته و متقابلا ۱۶ بیت بالا در ۱۶ بیت پایین قرار می گیرند.

Exchange

جهت جابجایی دو رجیستر و یا یک رجیستر و مکانی در حافظه بکار می‌رود.

```
eax = 237h
ecx = 978h
xchg  eax, ecx
eax = 978h
ecx = 237h
```

ساختار فایل‌های منبع (source) اسمبلی:

فایلهای منبع اسمبلی (*.asm) به قطعات زیر تقسیم می‌شوند:

```
.code ; code section starts here
.data ; data section starts here
.data? ; un initialized data starts here
.const ; constants section starts here
```

سایر قطعات در مورد ریسورس‌ها و غیره هستند که در آینده در مورد آنها بیشتر توضیح داده خواهد شد.

قسمت code ، حاوی کدهای برنامه است. قسمت Data ، حاوی داده‌های برنامه بوده و دارای دسترسی خواندن و نوشتن می‌باشد.کل قسمت data در فایل exe نهایی قرار می گیرد.
قسمت data? حاوی داده‌هایی بدون مقدار دهی اولیه هستند.در فایل exe لحاظ نشده و جزئی از حافظه‌ی در نظر گرفته شده توسط ویندوز برای برنامه می‌باشند.
قسمت const حاوی داده‌هایی ثابت با دسترسی فقط خواندنی هستند.در کل بهتر است ثوابت در فایلهای ضمیمه تعریف شده و سپس به برنامه ملحق شوند. از آنها بعنوان مقادیر immediate می‌توان استفاده کرد.

فایلهای اجرایی در ویندوز مانند فایلهای exe ، dll و غیره دارای قالب portable executable و یا PE هستند.قطعات فوق درPE-header با چند ویژگی خاص مشخص می‌شوند:

```
Section name
RVA (relative virtual address)
Offset
Raw size
Virtual size and flags.
```

RVA مکانی است نسبی در حافظه ، جایی که قطعه مورد نظر بارگذاری می‌شود. نسبی در اینجا به معنای نسبت به base address است (آدرسی در حافظه که برنامه در آن بارگذاری می‌شود).

این آدرس نیز در PE-header قرار دارد اما توسط PE-loader قابل تغییر است.آفست ، آفستی خام در فایل exe است ، جایی که داده‌هایی آغازین در آن قرار دارند.اندازه مجازی ، اندازه‌ای است که نهایتا در حافظه خواهد داشت.flags ، پرچم‌هایی هستند برای حالتهای فقط خواندنی ، اجرایی و غیره.

یک مثال :

```
.data
Number1  dd 12033h
Number2  dw 100h,200h,300h,400h
Number3  db "blabla",0

.data?
Value  dd ?

.code
mov  eax, Number1
mov  ecx, offset Number2
add   ax, word ptr [ecx+4]
mov  Value, eax
```

برنامه فوق به خوبی اسمبل نخواهد شد اما اهمیتی هم ندارد.در قطعات داده برنامه، هر کدی که قرار گیرد به فایل exe نیز منتقل شده و هنگامیکه برنامه در حافظه بارگذاری می‌شود در مکانهای مشخصی از حافظه قرار می گیرند.در قسمت data برنامه فوق سه برچسب (label) Number1 ، Number2 و Number3 قرار دارند.این برچسب ها آفست موقعیت خود را در برنامه نگهداری می کنند.بنابراین از آنها می‌توان جهت تعیین مکان برچسب‌ها در برنامه استفاده کرد.

در اینجا :

```
DD  = dword
DW = word
DB  = byte
```

رشته ها را نیز می‌توان بکار گرفت زیرا آنها لیستی از بایت‌ها می باشند.

در کد فوق ، قسمت داده‌ها در حافظه به شکل زیر خواهد بود:

33,20,01,00,**00**,01,00,02,00,03,00,04,**62**,6c,61,62,6c,61,00 (all hex numbers)
(every value is a byte)

در اینجا Number1 به موقعیتی در حافظه که بایت ۳۳ قرار دارد اشاره می کند، Number2 به موقعیت ۰۰ و Number3 به موقعیت ۶۲ (با حروف درشت مشخص شده اند) اشاره دارند.

اگر در برنامه خود از دستور زیر استفاده کنید :

```
mov  ecx, Number1
```

یعنی :

```
mov  ecx, dword ptr [location where the dword 12033h is in memory]
```

مقدار موجود در موقعیت حافظه Number1 دریافت می‌شود.

اما

```
mov ecx, offset Number1
```

به معنای دستور زیر است:

```
mov ecx, location where dword 12033h is in memory
```

موقعیت Numebr1 در حافظه دریافت می‌شود.

بنابراین نتیجه نهایی دو دستور زیر یکی است:

```
(1)
mov ecx, Number1

(2)
mov ecx, offset Number1
mov ecx, dword ptr [ecx] ( or mov ecx, [ecx])
```

برای تعریف داده ها به شکل زیر نیز می توان عمل کرد:

```
.data?
ManyBytes1  db  5000  dup (?)

.data
ManyBytes2  db  5000  dup (0)
```

در اینجا :

5000 dup = 5000 duplicates

برای مثال Value db 4,4,4,4,4,4,4 معادل است با Value db 7 dup (4) .

با توجه به قسمت قرار گرفتن ManyBytes1 ، در فایل اجرایی نهایی قرار نگرفته و صرفا ۵۰۰۰ بایت در حافظه برای آن در نظر گرفته می‌شود اما ManyBytes2 در فایل اجرایی قرار گرفته و حجم آنرا ۵۰۰۰ بایت بیشتر می کند.

پرش‌های شرطی:

در قسمت code از برچسب‌هایی به شکل زیر نیز می توان استفاده کرد:

```
.code

mov  eax, edx
sub   eax, ecx
cmp  eax, 2
jz  loc1
xor   eax, eax
jmp  loc2
loc1:
xor   eax, eax
inc   eax
loc2:
```

نکته :

xor eax, eax به معنای eax=0 است.

آنالیز کد فوق به شکل زیر است:

```
Mov  eax, edx ; put edx in eax
sub   eax, ecx ; substract ecx from eax
cmp  eax, 2
```

در اینجا به اینستراکشن جدیدی به نام cmp که جهت مقایسه دو مقدار بکار می‌رود بر می‌خوریم. اگر این دو مقدار با هم مساوی بودند، Z-flag (zeroflag) تنظیم خواهد شد.Zeroflag همانند carry ، بیتی است در رجیستر flag داخلی. در ادامه داریم:

```
jz  loc1
```

jz یک پرش شرطی است. یعنی Jz = jump if zero ، و یا بعبارتی پرش کن اگر zeroflag تنظیم شده باشد.loc1 برچسبی است برای پرش به اینستراکشنی که در پرش شرطی ذکر شده است. ادامه کد بصورت زیر است:

```
cmp  eax, 2 ; set zero flag if eax=2
jz  loc1 ; jump if zero flag is set
```

بصورت خلاصه دو خط فوق بدین معنا هستند : اگر eax مساوی ۲ بود به loc1 پرش کن.

در کد فوق دستور زیر به معنای پرش غیر شرطی است و همواره اجرا می شود :

```
jmp   loc2
```

اگر کد فوق را بخواهیم به زبان C ترجمه کنیم حاصل به شکل زیر خواهد بود:

```
if ((edx-ecx)==2)
{
    eax = 1;
}
else
{
    eax = 0;
}
```

رجیسترهای flag :

مهمترین و پرکاربردترین رجیسترهای flag به شرح زیر هستند :

رجیستر	معنا
ZF (Zero flag)	این پرچم زمانی تنظیم می شود که نتیجه محاسبات صفر باشد.
SF (Sign flag)	اگر تنظیم شده بود بدین معنا است که نتیجه محاسبات منفی است.
CF (Carry flag)	حاوی آخرین بیتی است که پس از محاسبات در سمت چپ قرار می گیرد.
OF (Overflow flag)	بیانگر سرریز محاسبات است.(بدین معنا که نتیجه محاسبه در مقصد جای نگیرد.)

جدول انواع پرش‌ها:

در جدول زیر لیستی از انواع پرش‌های شرطی بر اساس وضعیت پرچم مربوطه آورده شده‌اند:

Opcode	Meaning	Condition
JA	Jump if above	CF=0 & ZF=0
JAE	Jump if above or equal	CF=0
JB	Jump if below	CF=1
JBE	Jump if below or equal	CF=1 or ZF=1
JC	Jump if carry	CF=1
JCXZ	Jump if CX=0	register CX=0
JE (is the same as JZ)	Jump if equal	ZF=1
JG	Jump if greater (**signed**)	ZF=0 & SF=OF
JGE	Jump if greater or equal (**signed**)	SF=OF
JL	Jump if less (**signed**)	SF != OF
JLE	Jump if less or equal (**signed**)	ZF=1 or SF!=OF
JMP	Unconditional Jump	-
JNA	Jump if not above	CF=1 or ZF=1
JNAE	Jump if not above or equal	CF=1
JNB	Jump if not below	CF=0
JNBE	Jump if not below or equal	CF=1 & ZF=0
JNC	Jump if not carry	CF=0
JNE	Jump if not equal	ZF=0
JNG	Jump if not greater (**signed**)	ZF=1 or SF!=OF
JNGE	Jump if not greater or equal (**signed**)	SF!=OF
JNL	Jump if not less (**signed**)	SF=OF
JNLE	Jump if not less or equal (**signed**)	ZF=0 & SF=OF
JNO	Jump if not overflow (**signed**)	OF=0
JNP	Jump if no parity	PF=0
JNS	Jump if not signed (**signed**)	SF=0
JNZ	Jump if not zero	ZF=0
JO	Jump if overflow (**signed**)	OF=1
JP	Jump if parity	PF=1
JPE	Jump if parity even	PF=1
JPO	Jump if paity odd	PF=0
JS	Jump if signed (**signed**)	SF=1
JZ	Jump if zero	ZF=1

البته در اکثر مواقع نیازی به دانستن وضعیت پرچم مربوطه نیست و معنای اینستراکشن بسیار واضح است.برای مثال :

"Jump if above" means:
cmp x, y
jump if x is above y

توضیحاتی در مورد اعداد:

استفاده از اعداد صحیح و یا اعشاری در بسیاری از زبانهای برنامه نویسی تنها به تعریف متغیر وابسته است. در اسمبلر این مورد کاملا متفاوت است.محاسبات اعشاری توسط رجیسترهایی در FPU co-processor انجام می شود. در مورد این رجیسترها در آینده بیشتر صحبت خواهد شد.

- اعداد صحیح : در زبان سی اعداد صحیح ، signed و unsigned هستند.اعداد unsigned همانطور که پیش‌تر ذکر گردید اعدادی مثبت هستند.برای درک این موضوع به جدول زیر دقت بفرمایید (اندازه ها در مثال زیر بایت هستند):

Value	00	01	02	03	...	7F	80	...	FC	FD	FE	FF
Unsigned meaning	00	01	02	03	...	7F	80	...	FC	FD	FE	FF
Signed meaning	00	01	02	03	...	7F	-80	...	-04	-03	-02	-01

در اعداد علامت دار ، بازه ی صفر تا 7F مربوط به اعداد مثبت و از ۸۰ تا FF ، اعداد منفی قرار دارند و برای مثال در حالت DWORD ، بازه صفر تا 7FFFFFFF جهت اعداد مثبت و از ۸۰۰۰۰۰۰۰ تا FFFFFFFFh مربوط به اعداد منفی است.همانطور که ملاحظه می نمایید در اعداد منفی MSB تنظیم شده است. به این بیت ، بیت علامت نیز گفته می‌شود.

در مورد اعداد منفی و مثبت (اعداد علامت دار) ، اعمال جمع و تفریق بدون هیچ نکته خاصی قابل انجام است.برای مثال :

```
Calculate: -4 + 9
FFFFFFFC + 00000009 = 00000005. (which is correct)

Calculate 5 - (-9)
00000005 - FFFFFFF7 = 0000000E (which is correct, too ( 5 - -9 = 14)
```

اما در مورد ضرب ،تقسیم و مقایسه اعداد علامت دار اینستراکشن‌های خاص زیر وجود دارند:

```
imul   and   idiv
```

و حالت های مختلف آنها به صورت زیراست:

```
imul src
imul src, immed
imul dest,src, 8-bit immed
imul dest,src
idiv src
```

همچنین پرش های علامت دار و بدون علامت نیز وجود دارند که لیست آنها در جدول مربوطه ارائه شد. برای مثال ja یک پرش بدون علامت است :

```
cmp ax, bx
ja somewhere
```

اما jg پرشی با علامت می باشد:

```
cmp ax, bx
jg somewhere
```

آشنایی با تعدادی دیگر از Opcodes :

TEST

Test بر روی دو عملگر مبدا و مقصد عمل and را انجام داده و رجیستر flag را بر مبنای نتیجه تنظیم می کند. نتیجه به تنهایی ذخیره نمی گردد.

مثال:

```
test  eax, 100b ; (b-suffix stands for binary)
jnz   bitset
```

در اینجا پرش صورت خواهد گرفت اگر سومین بیت از سمت راست در eax تنظیم شده باشد.

معمولترین استفاده از test بررسی این مورد است که آیا یک رجیستر صفر است یا نه؟

```
Test  ecx, ecx
jz  somewhere
```

در این مثال پرش صورت خواهد گرفت اگر ecx صفر باشد.

STACK OPCODES

قبل از آشنایی با اینستراکشن‌های مربوطه باید در مورد پشته بیشتر توضیح داده شود.پشته مکانی است در حافظه که توسط رجیستر esp به آن اشاره شده و از آن برای نگهداری مقادیر موقتی استفاده می‌شود.دو اینستراکشن push و pop

جهت قرار دادن و دریافت مقادیر از پشته بکار می‌روند.مقداری که آخر از همه در پشته قرار مـی گیـرد ، در ابتـدا خـارج خواهد شد. به مثال زیر دقت بفرمایید:

```
(1) mov  ecx, 100
(2) mov  eax, 200
(3) push ecx ; save ecx
(4) push eax
(5) xor   ecx, eax
(6) add  ecx, 400
(7) mov edx, ecx
(8) pop  ebx
(9) pop  ecx
```

توضیح:

۱. ۱۰۰ در ecx قرار می گیرد.

۲. ۲۰۰ در eax قرار می گیرد.

۳. ecx بر روی پشته قرار داده می‌شود (اولین قرارگیری).

۴. eax بر روی پشته قرار می گیرد.

۵. در مراحل ۵ و ۶ و ۷ بر روی ecx عملیاتی صورت گرفته و مقدار آن تغییر می کند.

۸ در اینجا ebx دوباره از پشته دریافت می‌شود و اکنون مقدار آن ۲۰۰ می‌شـود. (آخـرین مقـدار قـرار داده شـده ، اولین مقدار دریافت شده)

۹ در اینجا eax از پشته دریافت شده و اکنون مقدار آن دوباره ۱۰۰ می‌شود. (اولین مقدار قرار داده شده ، آخـرین مقدار دریافت شده)

برای توضیح بیشتر در مورد اتفاقات رخ داده شده در حافظه، به جداول زیر دقت بفرمایید:

Offset	1203	1204	1205	1206	1207	1208	1209	120A	120B
Value	00	00	00	00	00	00	00	00	00
					ESP				

(the stack is here initially filled with zeroes, but in reality it is not like this. ESP indicates the offset that ESP points to)

```
mov ax, 4560h
push ax
```

Offset	1203	1204	1205	1206	1207	1208	1209	120A	120B
Value	00	00	**60**	**45**	00	00	00	00	00
			ESP						

```
mov cx, FFFFh
push cx
```

Offset	1203	1204	1205	1206	1207	1208	1209	120A	120B
Value	**FF**	**FF**	**60**	**45**	00	00	00	00	00
	ESP								

pop edx

Offset	1203	1204	1205	1206	1207	1208	1209	120A	120B
Value	FF	FF	60	45	00	00	00	00	00
					ESP				

در اینجا edx مساوی 4560FFFFh خواهد شد.

رجیسترهای مهیا در یک پردازشگر x86 ، منابع محدود بوده و بین انواع مختلف پروسه‌های در حال اجرا به اشتراک گذاشته می‌شوند. بنابراین استفاده صحیح از آنها برای نوشتن کدی قابل اطمینان بسیار اساسی است.

یک پردازشگر x86 دارای ۸ رجیستر عمومی است:

EAX EBX ECX EDX ESI EDI ESP EBP

در این بین دو رجیستر ESP و EBP منحصرا برای شروع و خاتمه یک روال مورد استفاده قرار می‌گیرند.بنابراین به صورت مؤثر ، تنها ۶ رجیستر عمومی جهت برنامه نویسان مهیا است.

در ویندوزهای ۳۲ بیتی در مورد کار با این ۶ رجیستر روشی خاص موجود است. این ۶ رجیستر به دو گره تقسیم می‌شوند. گروه اول را آزادانه می توان مورد استفاده و تغییر قرار داد . اما گروه دوم باید مقادیرشان حفظ شوند.

رجیسترهای زیر باید مقادیرشان حفظ شوند:

EBX ESI and EDI

رجیسترهای زیر را آزادانه می توان مورد استفاده قرار داد:

EAX ECX and **EDX**

مثال :

```
TestProc proc var1:DWORD, var2:DWORD
   push ebx
   push esi
   push edi
   ; ------------------------------------
   ; write code that uses EBX ESI and EDI
   ; ------------------------------------
   pop edi
   pop esi
   pop ebx
   ret
TestProc endp
```

در این تابع فرض شده است که قسمتی از کد ، رجیسترهای EBX ESI and EDI را مورد استفاده قرار داده و تغییر می‌دهد.بنابراین نیاز است تا با استفاده از اینستراکشن های مربوط به پشته ، مقادیر آنها را پیش از تغییر ذخیره و سپس پس از تغییر به سیستم عامل بازگشت داد.

هنگام استفاده از توابع API ویندوز ، رجیسترهای ذکر شده کاملا مراقبت و حفظ می‌شوند. اما بدیهی است که این توابع نیز می‌توانند آزادانه سایر رجیسترهای را استفاده و تغییر دهند. بنابراین اگر پس از فراخوانی یک تابع API نیاز به استفاده از سایر رجیسترها است باید به این مورد دقت کرده و با استفاده از اینستراکشن‌های پشته ، مقادیر رجیسترها را حفظ نمود.

برای بهینه سازی کدی که از توابع API درون یک حلقه استفاده می‌کند بهتر است از رجیسترهای EBX ESI and EDI جهت ساخت حلقه استفاده کنید. زیرا این رجیسترها در فراخوانی های توابع API حفظ شده و سربار تحمیلی ناشی از حفظ سایر رجیسترها را در پشته ندارند.

دستورالعمل‌هایی جهت ساده سازی حفظ رجیسترها:

با استفاده از دو اینستراکشن pushad و popad می‌توان تمام رجیسترهای قطعه‌ای از کد را که مخلوطی از دستورات اسمبلی و توابع API ویندوز است ، حفظ و بازیابی نمود. برای مثال:

```
; assembler code
pushad
; Write the code you wish to use to display the data you require
popad
; more assembler code
```

و اگر از پرش‌های شرطی استفاده می‌شود و حفظ flag های مربوط به آنها مورد نیاز است ، می توان از PUSHFD و POPFD استفاده کرد.این دستورالعمل‌ها حالت پرچم‌های پردازشگر را حفظ می‌کنند.

نگاهی دقیق‌تر به پشته:

پشته بازه‌ای از حافظه است که برای ذخیره سازی موقتی داده‌ها می‌تواند مورد استفاده قرار گیرد (درون یک روال و یا روشی متداول برای ارسال پارامترها به یک روال). پشته در کد بوسیله دستورالعمل های push و pop قابل دستیابی است.مقداری که در آخر بر روی پشته قرار می‌گیرد ، در ابتدا از آن بازیابی می‌گردد.

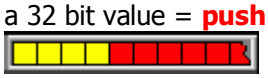
هنگامیکه داده‌ای بر روی پشته قرار می گیرد ، پردازشگر اشاره‌گر پشته (ESP) را کاهش می‌دهد و هنگام pop شدن داده‌ای این اشاره‌گر افزایش می‌یابد.

شماتیک یک پشته در زیر ارائه شده است.هر مربع بیانگر یک بایت است. بالای پشته ، در سمت چپ هر عکس قرار گرفته است. رنگ های مختلف در جهت مشخص کردن داده‌های مختلفی است که بر روی پشته قرار گرفته اند.

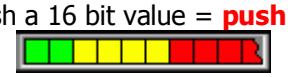
Existing stack layout

Pushes

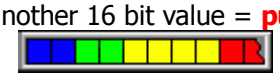
Push a 32 bit value = **push eax**



Push a 16 bit value = **push cx**



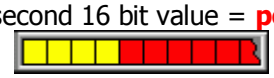
Push another 16 bit value = **push dx**



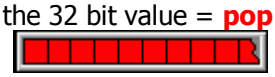
Pops

Pop a 16 bit value = **pop dx**

Pop second 16 bit value = **pop cx**



Pop the 32 bit value = **pop eax**



برای مثال کدی در این باره:

```
mov  edx, 100
mov  ecx, 500
; --------------------------------
; push the 2 values onto the stack
; --------------------------------
push  edx ; edx has the value 100
push  ecx ; ecx has the value 500
; ------------------------------
; pop the 2 values off the stack
; ------------------------------
pop  eax ; eax has the value 500
pop  ecx ; ecx has the value 100
```

همانطور که پیشتر نیز گفته شد ، روال متداول ویندوزهای ۳۲ بیتی بدین شرح است که رجیسترهای EAX,ECX,EDX آزادانه قابل تغییر هستند. اما مقادیر رجیسترهای EBX,ESI,EDI درصورت تغییر درون کد باید حفظ شوند. در اغلب حالتها رجیسترهای EBP و ESP مستقیما مورد استفاده قرار نمی‌گیرند (زیرا به صورت خودکار در ابتدا و انتهای روالها استفاده می‌شوند).

متوازن کردن پشته:

هنگام استفاده از پشته باید تقارنی از لحاظ تعداد بایت push شده و تعداد بایت pop شده وجود داشته باشد.در غیر اینصورت هنگام پایان کد یک روال ، ادامه کد از آدرسی نادرست آغاز شده و تقریبا انحصارا سبب از کار افتادن برنامه می‌شود.

نکاتی در مورد استفاده از پشته:

پشته در مورد داده‌هایی که می توان در آن ذخیره و یا بازیابی کرد بسیار انعطاف پذیر است. در اینجا چند نکته کوچک وجود دارند که هنگام کار با پشته بسیار مفید هستند: برای مثال می توان یک مقدار ۳۲ بیتی را بر روی پشته قرار داد و سپس دو مقدار ۱۶ بیتی را بازیابی نمود.

```
; ----------------------------------
; push a 32 bit value onto the stack
; ----------------------------------
push edx
; ----------------------------------
; pop two 16 bit values off the stack
; ----------------------------------
pop ax
pop cx
```

با توجه به اینکه ۴ بایت بر روی استک قرار گرفته و ۴ بایت بازیابی شده در آخر پشته متوازن شده است.

از پشته برای کارهای بسیاری استفاده می‌شود.برای مثال می توان مقداری را بر روی پشته قرار داد و سپس زمانی که به آن نیاز است آنرا بازیابی کرد.به این صورت نیازی به تخصیص دادن یک متغیر در حافظه برای انجام اینکار نیست.

برای مثال :

کپی داده ها بین یک یک رجیستر و یک عملوند حافظه:

```
push ecx
pop memVar

or

push memVar
pop edx
```

و یا بین دو متغیر حافظه:

```
push memVar1
pop memVar2
```

بجای استفاده از رجیستر برای انجام اعمال فوق:

```
mov edx, memvar1
mov memvar2, edx
```

به این صورت تعداد رجیسترهای مورد استفاده را می توان به صورت بهینه‌ای کاهش داد.

CALL & RET

call به قطعه ای از کد پرش کرده و سپس به محض رسیدن به اینستراکشن ret متوقف گردیده و روند عادی کار ادامه می‌یابد. آنها را می‌توان معادل توابع و رویه ها در سایر زبانهای برنامه نویسی در نظر گرفت. برای مثال:

```
..code..
call   0455659
..more code..

Code at 455659:
add   eax, 500
mul   eax, edx
ret
```

هنگامیکه کد به سطر مربوط به call می‌رسد به کد قرار گرفته در آدرس 0455659 پرش کرده و تا ret ادامه می‌یابد. سپس روال عادی کار ادامه یافته و اینستراکشن پس از call اجرا می‌شود.

در ابتدای اجرای call ، رجیستر EIP (اشاره‌گری به اینستراکشن بعدی که باید اجرا شود) بر روی پشته قرار داده می‌شود. در آخر اینستراکشن ret سبب pop شدن این رجیستر می‌گردد.

آرگومانهای یک تابع نیز با استفاده از push های قبل از call قابل دریافت هستند:

```
Push  something
Push  something2
Call  procedure
```

درون یک call ، آرگومانها از پشته خوانده شده و استفاده می‌شوند.

نکته:

در ویندوز و توابع ویندوز به صورت استاندارد ، خروجی یک تابع در eax قرار می گیرد (البته بدیهی است که از هر رجیستر مناسبی برای این امر می توان استفاده کرد اما این یک استاندارد پذیرفته شده تحت ویندوز است).

برنامه نویسی asm32 با استفاده از masm :

مطالعه این قسمت کاملا اختیاری است ، اما همانگونه که پیش تر نیز ذکر شد، استفاده از masm برنامه نویسی اسمبلی ویندوز را به شدت ساده‌تر می کند.

ساختارهای تصمیم گیری و حلقه ها:

Masm حاوی ماکروهایی جهت سهولت تعریف حلقه‌ها و یا ساختارهای تصمیم گیری است:

```
.IF, .ELSE, .ELSEIF, .ENDIF
.REPEAT, .UNTIL
.WHILE, .ENDW, .BREAK
.CONTINUE
```

.If

با استفاده از ماکروی If. (نقطه قبل از آنرا فراموش نکنید) ، استفاده از ساختارهای تصمیم گیری به شدت ساده تر شده و دیگر نیازی به درگیر شدن با جزئیات انواع پرش‌ها نمی‌باشد.

مثال یک:

```
.IF eax==1
;eax is one
.ELSEIF eax=3
; eax is three
.ELSE
; eax is not one or three
.ENDIF
```

مثال دو (if تو در تو نیز مجاز است):

```
.IF eax==1
    .IF ecx!=2
        ; eax= 1 and ecx is not 2
    .ENDIF
.ENDIF
```

البته مثال فوق را به شکل ساده تر زیر نیز می‌توان عملی نمود:

```
.IF (eax==1 && ecx!=2)
; eax = 1 and ecx is not 2
.ENDIF
```

عملگرهای زیر در اینجا مجاز می‌باشند:

==	is equal to
!=	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than to equal to
&	bit-test
!	logical NOT
&&	logical AND
||	logical OR
CARRY?	carry bit set
OVERFLOW?	overflow bit set
PARITY	parity bit set
SIGN?	sign bit set
ZERO?	zero bit set

Repeat

این عبارت قطعه‌ای از کد را تا زمانیکه شرط آن معتبر باشد اجرا و تکرار می کند. مثال:

```
.REPEAT
; code here
.UNTIL eax==1
```

کد فوق ، تا زمانیکه eax مساوی یک باشد ، قطعه کد قرار گرفته بین repeat و until را اجرا و تکرار می کند.

While

این عبارت قطعه کد را تا هنگامیکه شرط آن معتبر باشد اجرا و تکرار می کند. مثال:

```
.WHILE eax==1
; code here
.ENDW
```

برای خارج شدن از حلقه می توان از عبارت break. استفاده کرد. مثال :

```
.WHILE edx==1
   inc eax
   .IF eax==7
      .BREAK
   .ENDIF
.ENDW
```

در اینجا اگر eax مساوی ۷ شود حلقه خاتمه خواهد یافت.

همانند زبان C از عبارت continue نیز درون حلقه برای صرفنظر کردن از اجرای کدهای بعد از آن می توان استفاده کرد.

به صورت خلاصه :

IF - ELSE IF - ELSE

The C version:	**The MASM version:**

`if ( var1 == var2 )`
`{`
`    // Code goes here`
`}`
`else`
`if ( var1 == var3 )`
`{`
`    // Code goes here`
`}`
`else`
`{`
`    // Code goes here`
`}`
	`.if ( var1 == var2 )`
	`    ; Code goes here`
	
	`.elseif ( var1 == var3 )`
	`    ; Code goes here`
	
	`.else`
	`    ; Code goes here`
	
	`.endif`

DO - WHILE

The C version:	**The MASM version:**

`do`
`{`
`    // Code goes here`
`}`
`while ( var1 == var2 );`
	`.repeat`
	`    ; Code goes here`
	
	`.until ( var1 != var2 )`

WHILE

The C version:	**The MASM version:**

`while ( var1 == var2 )`
`{`
`    // Code goes here`
`}`
	`.while ( var1 == var2 )`
	`    ; Code goes here`
	
	`.endw`

Invoke

یکی از مهمترین مزایای masm نسبت به نمونه‌های مشابه آن مانند tasm و nasm ، بکارگیری invoke جهت استفاده ساده تر از رویه‌ها و توابع می باشد.

حالت معمول استفاده از توابع به شکل زیر است:

```
push  parameter3
push  parameter2
push  parameter1
call  procedure
```

با استفاده از invoke داریم:

```
invoke   procedure, parameter1, parameter2, parameter3
```

لازم به ذکر است که کد نهایی اسمبل شده فوق با حالت معمول هیچ تفاوتی ندارد اما استفاده از invoke ،کد را خواناتر و ساده‌تر می‌نماید.

برای استفاده از invoke در مورد رویه‌ها باید prototype تعریف کرد. برای مثال:

```
PROTO STDCALL testproc: DWORD, :DWORD, :DWORD
```

عبارت فوق رویه‌ای را به نام testproc با سه آرگومان با اندازه DWORD تعریف می کند.اکنون :

```
Invoke  testproc, 1, 2, 3, 4
```

پس از اجرای کد فوق ، masm خطایی را در مورد بکارگیری یک آرگومان اضافی به شما گوشزد خواهد کرد. همچنین نوع و اندازه پارامترهای ورودی نیز در این حالت دقیقا بررسی می‌شوند.

در عبارت invoke می توان بجای offset از addr استفاده کرد.

رویه‌ها به شکل زیر تعریف می‌شوند:

```
testproc PROTO STDCALL :DWORD, :DWORD, :DWORD
.code
testproc proc param1:DWORD, param2:DWORD, param3:DWORD
ret
testproc endp
```

در این مثال رویه testproc با سه آرگومان ورودی تعریف می‌شود. prototype ذکر شده جهت استفاده در عبارت invoke تعریف شده است.

```
testproc  PROTO STDCALL :DWORD, :DWORD, :DWORD
.code
testproc proc param1:DWORD, param2:DWORD, param3:DWORD
   mov ecx, param1
   mov edx, param2
   mov eax, param3
   add edx, eax
   mul eax, ecx

   ret
testproc endp
```

عملکرد مثال فوق به صورت زیر است:

```
testproc(param1, param2, param3) =
    param1 * (param2 + param3)
```

و نتیجه مطابق استاندارد ذکر شده در EAX قرار گرفته است.

برای تعریف متغیرهای محلی می‌توان به صورت زیر عمل کرد:

```
testproc proc param1:DWORD, param2:DWORD, param3:DWORD
   LOCAL var1:DWORD
   LOCAL var2:BYTE
   mov ecx, param1
   mov var2, cl
   mov edx, param2
   mov eax, param3
   mov var1, eax
   add edx, eax
   mul eax, ecx
   mov ebx, var1
   .IF bl==var2
       xor eax, eax
   .ENDIF

   ret
testproc endp
```

از این متغیرهای محلی خارج از رویه نمی‌توان استفاده کرد زیرا در ابتدای کار بر روی پشته قرار گرفته و در آخر هنگام ret از پشته حذف می‌شوند.

اسمبلی مقدماتی ویندوز

پس از ذکر مقدمات ، اکنون طرز استفاده از اسمبلی در ویندوز را می‌توان آموخت.

API

اساس برنامه نویسی ویندوز بر مبنای windows API استوار شده است (Application programming interface) و مجموعه‌ای از توابع مهیا شده توسط سیستم عامل را در اختیار برنامه نویس قرار می‌دهد (بنابراین همانطور که پیشتر نیز ذکر شد، در اینجا وقفه‌های داس با API ویندوز جایگزین شده اند).تمام برنامه‌های تحت ویندوز از این توابع استفاده می‌کنند.این توابع در dll های سیستمی مانند kernel ، gdi و غیره قرار گرفته‌اند.در حالت کلی برای تعدادی از توابع ، دو نگارش Ansi و یونیکد تعریف گردیده و به نحوه‌ی رفتار و ذخیره سازی رشته‌ها مربوط می‌شود.در حالت Ansi ، هر بایت بیانگر یک سمبول (کد اسکی) بوده و مختوم به نال می باشد (از 0-byte درجهت مشخص کردن انتهای رشته استفاده می شود.). یونیکد از قالب widechar که به ازای هر سیمبول از ۲ بایت استفاده می کند، کمک می گیرد.در این حالت امکان تعریف زبانهایی که نیاز به کاراکترهای بیشتری مانند زبان چینی هستند، فراهم می گردد.در این حالت انتهای رشته با دو بایت صفر مشخص می‌شود.ویندوز برای مشخص نمودن این موارد در توابع به شکل زیر عمل می کند:

MessageBoxA (A-suffix for ansi)
MessageBoxW (W-suffix for widechar (unicode))

بکار گیری dll ها و توابع API ویندوز

جهت بکارگیری توابع API ویندوز نیاز به فراخوانی dll ها در برنامه است.این کار توسط import libraries (فایلهای lib) میسر است.این کتابخانه‌ها در جهت بارگذاری پویای فایلهای dll ضروری هستند (برای مثال بارگذاری پویا در آدرس پایه ای پویا در حافظه). در بسته win32asm که در آدرس win32asm.cjb.net قابل دریافت است، فایلهای کتابخانه ای اکثر dll های سیستمی ارائه شده‌اند.

برای بارگذاری یک فایل کتابخانه‌ای می‌توان از عبارت includelib استفاده کرد.برای مثال:

```
includelib  C:\masm32\lib\kernel32.lib
```

در اغلب مثالها از روش زیر (مسیردهی نسبی) استفاده می‌شود:

```
includelib  \masm32\lib\kernel32.lib
```

البته الحاق یک فایل کتابخانه‌ای به برنامه به تنهایی کافی نیست و نیاز به فایلهای inc و یا include file نیز می‌باشد.این نوع فایلها با استفاده از برنامه کمکی l2inc به صورت خودکار از فایلهای کتابخانه‌ای قابل تولید هستند.

برای الحاق کردن فایلهای inc به برنامه به صورت زیر عمل می‌شود:

```
Include  \masm32\include\kernel32.inc
```

در فایلهای inc ، prototypes توابع تعریف شده در یک کتابخانه جهت بکار گیری آنها توسط عبارت invoke موجود هستند.برای مثال :

```
kernel32.inc:
...
MessageBoxA proto stdcall :DWORD, :DWORD, :DWORD, :DWORD
MessageBox textequ <MessageBoxA>
...
```

در این فایلها توابعی از نوع Ansi با حرف A در انتها مشخص نشده‌اند و با تعریف نمونه‌ی اصلی یکی هستند.

پس از الحاق فایلهای کتابخانه ای و فایلهای inc می‌توان از توابع موجود در کتابخانه استفاده کرد. برای مثال:

```
Invoke  MessageBox, NULL, ADDR MsgText, ADDR MsgTitle, NULL
```

معرفی windows.inc :

فایل inc ویژه ای به نام windows.inc وجود دارد که حاوی تمام ساختارها و ثوابت تعریف شده در API ویندوز است.برای مثال ثوابت مربوط به حالتهای مختلف یک MessageBox به صورت زیر در آن تعریف شده اند:

```
MB_OK equ 0
MB_OKCANCEL equ ...
MB_YESNO equ ...
```

بنابراین از این اسامی بعنوان ثوابت می‌توان استفاده کرد:

```
invoke MessageBox, NULL, ADDR MsgText, ADDR MsgTitle, MB_YESNO
```

برای استفاده از آن:

```
include \masm32\include\windows.inc
```

ساختار کلی یک فایل منبع اسمبلی (*.asm) در ویندوز

```
.486
.model flat, stdcall

option casemap:none

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
includelib \masm32\lib\gdi32.lib

include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
include \masm32\include\gdi32.inc
include \masm32\include\windows.inc

.data

blahblah

.code

start:

blahblah

end start
```

معانی قسمت های مختلف ذکر شده به صورت زیر است:

486	به اسمبلر می گوید که باید opcodes جهت پردازشگرهای ٤٨٦ (یا بالاتر) تولید کند.از 386. نیز می توان استفاده کرد اما 486. در اغب حالات پاسخگو است.
.model flat, stdcall	استفاده از مدل حافظه تخت (که پیشتر در مورد آن صحبت شده بود) و تعریف stdcall calling convention ، بدین معنا که پارامترهای تابع از راست به چپ push می‌شوند (آخرین پارامتر در ابتدا بر روی پشته قرار می گیرد). این مورد تقریبا حالتی استاندارد برای تمامی توابع ویندوز است.
option casemap:none	نگاشت کاراکترها را به حروف بزرگ ، کنترل می کند.برای اینکه فایل windows.inc صحیح بکار گرفته شود باید به none تنظیم گردد.
includelib	در بالا توضیح داده شد.
include	در بالا توضیح داده شد.
.data	شروع قسمت داده‌های برنامه . پیشتر توضیح داده شد.

.code	شروع قسمت کد برنامه . پیشتر توضیح داده شد.
start: end start	برچسبی برای شروع و خاتمه برنامه. البته برچسب شروع هر نام دلخواهی می تواند باشد برای مثال: startofprog: end startofprog

آشنایی بیشتر با کتابخانه‌ها :

روش ایجاد کتابخانه‌ها بسیار شبیه به کد نویسی متداول است.هدف از ایجاد آنها کپسوله کردن تعدادی از ماژولهای برنامه است تا بتوان از آنها به دفعات در قسمت‌های مختلف برنامه بدون نیاز به کپی کردن کل کد آنها در قسمت های تکراری ، استفاده کرد.

هنگامیکه شما از کتابخانه ای در برنامه خود استفاده می‌کنید ، تمام کد مورد نیاز توسط linker اسمبلر از فایل کتابخانه دریافت و به برنامه ضمیمه می‌شود. بنابراین هنگام توزیع فایل exe نهایی نیازی به ارائه کتابخانه‌ها نمی‌باشد .

لازم به ذکر است که linker تنها قسمت‌هایی از کتابخانه را که در برنامه مورد استفاده قرار گرفته است به فایل اجرایی نهایی ضمیمه می‌کند . به این صورت حجم نهایی فایل اجرایی بسیار بهینه خواهد بود.

ساختار یک فایل کتابخانه در مرحله کد نویسی به صورت زیر است:

```
.386 ; set processor type
.model flat, stdcall ; memory model & calling convention
option casemap :none ; case sensitive
; ------------------------------
; Include any needed INCLUDE files
; or procedure prototypes.
; ------------------------------
; include \masm32\include\windows.inc
; include \masm32\include\gdi32.inc
; include \masm32\include\user32.inc
; ------------------------------------------------------
; Write the prototype for the procedure name below ensuring
; that the parameter count & size match and put it in your
; library include file.
; ------------------------------------------------------
; YourModule PROTO :DWORD,:DWORD etc.....
.code
; ------------------------------------------------
YourModule proc par1:DWORD,par2:DWORD etc.....
ret
YourModule endp
; ------------------------------------------------
```

همانطور که ملاحظه می‌کنید کدنویسی آن با برنامه نویسی متداول تفاوتی ندارد. پس از اطمینان حاصل کردن از صحت عملکرد یک رویه به راحتی می‌توان از آن یک فایل کتابخانه ای ساخت.

برای ساخت فایل lib از کد فوق می توان به صورت زیر عمل کرد:

```
@echo off
\masm32\bin\ml /c /coff *.asm
\masm32\bin\lib *.obj /out:yourlib.lib
: The following line works as well
: \masm32\bin\link -lib "*.obj" "/out:yourlib.lib"
```

پس از ساخت فایل lib ، باید جهت استفاده از آن ، تعاریف رویه‌های موجود در کتابخانه را در یک فایل inc قرار داد تا سایرین بتوانند از کتابخانه استفاده نمایند.

روش فراخوانی این فایلهای نهایی در قسمت‌های قبل توضیح داده شدند.

اولین برنامه!

هدف از این برنامه نمایش hello world معروف به کمک یک MessageBox است.

```
.486

.model   flat, stdcall

option    casemap:none

includelib   \masm32\lib\kernel32.lib
includelib   \masm32\lib\user32.lib

include     \masm32\include\kernel32.inc
include     \masm32\include\user32.inc
include     \masm32\include\windows.inc

.data
MsgText   db   "Hello world!",0
MsgTitle   db   "This is a messagebox",0

.code
start:
   invoke MessageBox, NULL, ADDR MsgText, ADDR MsgTitle, MB_OK or  MB_ICONINFORMATION
   invoke ExitProcess, NULL
end start
```

توضیحات بیشتری در مورد کد فوق:

در این مثال از دو تابع API به شرح زیر (مطابق MSDN) استفاده شده است:

int **MessageBox**(
HWND hWnd, // handle of owner window
LPCTSTR lpText, // address of text in message box
LPCTSTR lpCaption, // address of title of message box
UINT uType // style of message box
);

Parameters

hWnd	Identifies the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.
lpText	Points to a null-terminated string containing the message to be displayed.
lpCaption	Points to a null-terminated string used for the dialog box title. If this parameter is NULL, the default title Error is used.
uType	Specifies a set of bit flags that determine the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

همانطور که ملاحظه می فرمایید تابع MessageBox ، ۴ آرگومان را می‌پذیرد: دستگیره‌ای به پنجره‌ی والد! ☺ ، اشاره‌گری به رشته پیغامی که باید نمایش یابد ، اشاره‌گری به رشته عنوان MessageBox و در آخر نوع MessageBox .

با توجه به اینکه برنامه ما پنجره‌ای ندارد می توان hWnd را مساوی Null قرار داد.lpText اشاره‌گری است به رشته ما. این مورد بدین معنا است که آرگومان ذکر شده ، آفست مکانی از حافظه ، که متن مورد نظر در آن قرار گرفته است را نیاز دارد.

اگر به کد دقت بفرمایید نحوه معرفی رشته‌ها به صورت زیر است :

```
.data
MsgText   db  "Hello world!",0
MsgTitle   db  "This is a messagebox",0
```

data. به معنای شروع قسمت داده‌ها است. یک رشته مجموعه ای از بایت ها است (که با db مشخص شده) مختوم به نال (بایت صفر که مشخص گردیده است).

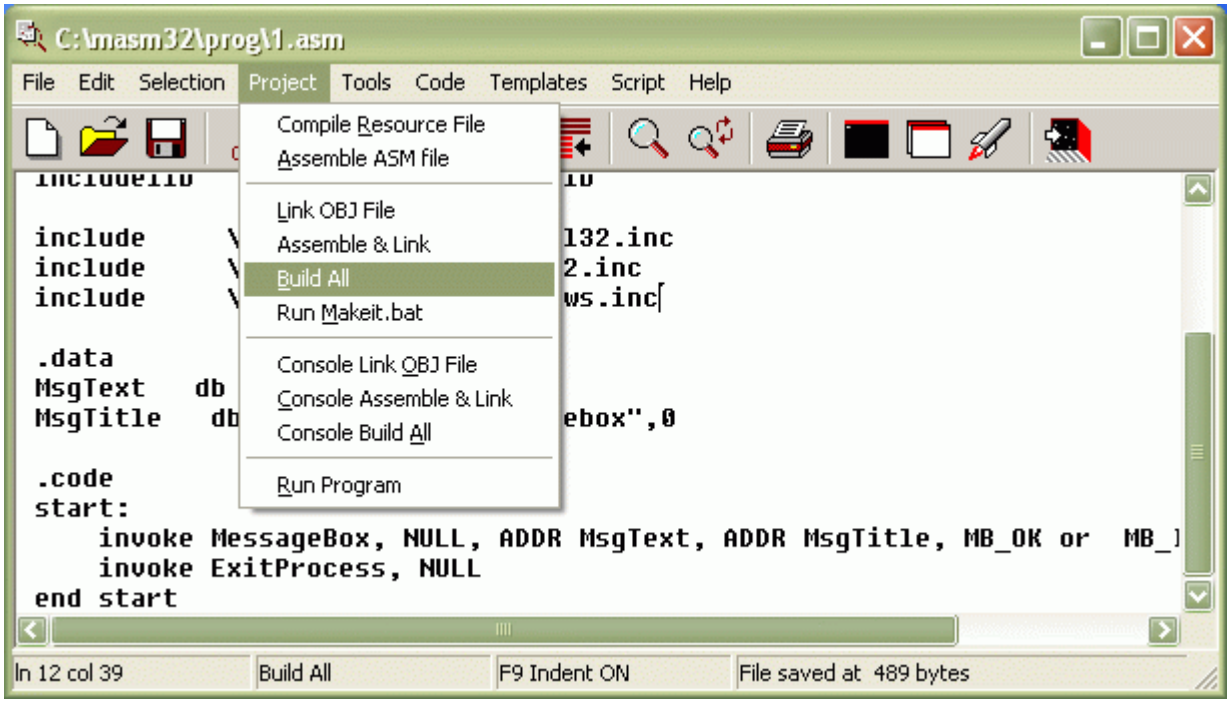
اگر از تابع زیر استفاده نشود و پس از نمایش MessageBox روی دکمه Ok آن کلیک شود برنامه crash خواهد کرد.

VOID **ExitProcess**(

UINT uExitCode // exit code for all threads
);

برنامه ها در ویندوز از ExitProcess برای خاتمه کار خود استفاده می کنند.

این مثال بسادگی در masm32 قابل build و اجرا است (شکل زیر):



شکل ۱۱– build و اجرای اولین برنامه asm32.

و پس از اجرای برنامه خواهیم داشت! (برنامه‌ای به حجم کمتر از ۲ و نیم کیلوبایت)

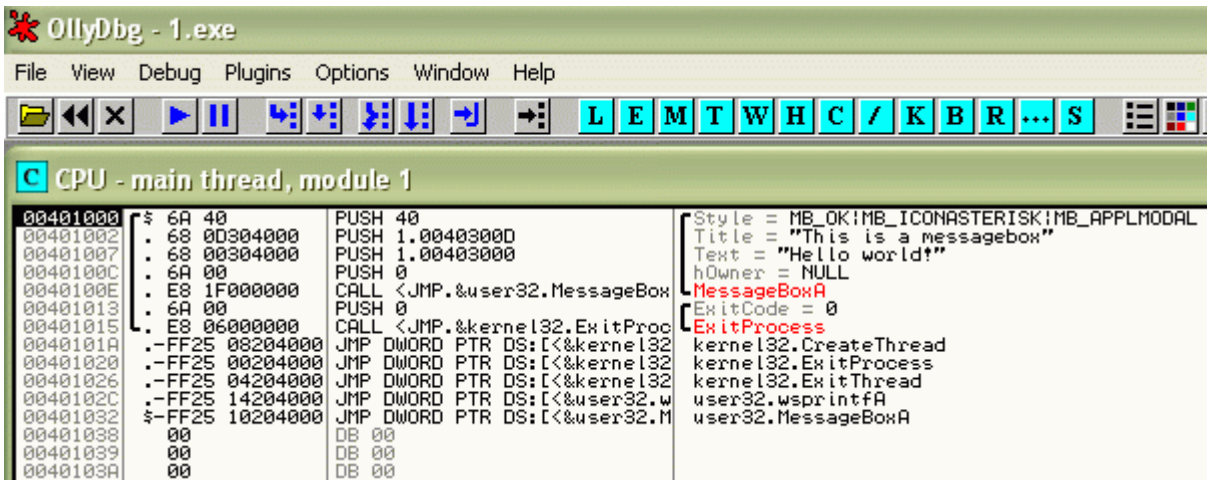
شکل ۱۲– نمایی از اجرای اولین برنامه

اگر علاقمند باشید disassembly فایل exe را مشاهده کنید، ابتدا برنامه OllyDbg را دانلود کرده و سپس فایل exe را در آن باز کنید. این برنامه از آدرس خانگی آن در http://home.t-online.de/home/Ollydbg/ قابل دریافت است (و یا اولین لینک پس از جستجو در گوگل).

در شکل زیر نحوه‌ی ترجمه نهایی کد نوشته شده، نحوه قرارگیری پارامترهای تابع MessageBox بر روی پشته و اطلاعاتی دیگر مانند Opcodes معادل دستورات و غیره را می‌توان مشاهده کرد.

در اینجا مفهوم جمله زیر که در ابتدای کار نیز ذکر گردید بیشتر مشخص می‌شود:

"هر دستور اسمبلی بوسیله برنامه اسمبلر مستقیما به اعدادی که توسط پردازشگر قابل اجرا است ترجمه می‌شود."



شکل ۱۳– نمایش دیس اسمبلی اولین برنامه نوشته شده در OllyDbg .

کار با رشته‌ها در اسمبلی ویندوز:

در ویندوز داده‌های رشته ای تقریبا به صورت انحصاری مختوم به نال هستند. البته انواع دیگری از رشته ها در ویندوزهای ۳۲ بیتی نیز موجود هستند ، مانند رشته‌های یونیکد . اما اغلب توابع API با رشته‌های مختوم به نال (zero terminated strings) کار می‌کنند. لازم به ذکر است که کار با رشته ها بر روی توالی بایت‌ها (بجای کار با کاراکترها) نیز قابل انجام است.

تکنولوژی رشته‌های مختوم به نال بسیار قدیمی است اما اساس بسیار ساده ای دارد. از لحاظ مصرف حافظه بسیار مقرون به صرفه بوده و پردازش آنها نیز بسیار سریع است.

روش تعریف یک رشته در قسمت data. به صورت زیر است (به شکل آرایه ای از بایت های مختوم به نال):

```
MyString  db  "This is a string",0
```

در ویندوزهای ۳۲ بیتی ، اینستراکشن‌های کار با رشته‌ها (به همراه REP/REPE/REPNE) از رجیسترهای ESI ، EDI و ECX بعنوان شمارشگر حلقه استفاده می کنند.این دستورالعمل ها کار خود را تازمانیکه شرط مورد نظر برآورده شود تکرار می‌نمایند.

مثال:

در اینجا source ،آدرس بافر منبع جهت کپی است. همچنین فرض شده است که اندازه بافر dest (بافر مقصد برای کپی کردن بایت های منبع) به اندازه کافی بزرگ می‌باشد.ln تعداد بایتی است که باید کپی شود.

```
cld ; set direction flag forward
mov esi, source ; put address into the source index
mov edi, dest ; put address into the destination index
mov ecx, ln ; put the number of bytes to copy in ecx
; ------------------------------------------------
; repeat copying bytes from ESI to EDI until ecx = 0
; ------------------------------------------------
rep movsb
```

در این مثال اینستراکشن movsb ، هر بایت را از ESI به EDI کپی کرده و یک واحد از ECX می کاهد. زمانی کار rep خاتمه می یابد که ecx مساوی صفر شود.

هنگامیکه یک رشته مختوم به نال کپی می‌شود می‌توان جهت اتمام کار ، کد اسکی مربوطه (صفر) را بررسی کرد.

برای مثال:

```
cld ; clear direction flag to read forward
mov esi, source ; put address into the source index
mov edi, dest ; put address into the destination index
label:
   lodsb ; load byte from source into AL and inc ESI
   stosb ; write AL to dest and inc EDI
   cmp al, 0 ; see if its an ascii zero
   jne label ; read the next byte if its not
```

برای بالا بردن سرعت بهتر است هر بایت را از منبع به AL انتقال داد و سپس از AL به مقصد. بر روی پنتیوم و پردازنده‌های جدیدتر ، MOV/INC از LODSB/STOSB سریعتر هستند (بدلیل استفاده از dereferencing و عمل کردن رجیسترهای مورد استفاده بعنوان آدرس‌های حافظه) .

```
  mov esi, source ; put address into the source index
  mov edi, dest ; put address into the destination index
label:
  mov al, [esi] ; copy byte at address in esi to al
  inc esi ; increment address in esi
  mov [edi], al ; copy byte in al to address in edi
  inc edi ; increment address in edi
  cmp al, 0 ; see if its an ascii zero
  jne label ; jump back and read next byte if not
```

کد فوق طولانی‌تر است اما بر روی پردازشگرهایی با dual pipelines (که به آن pairing هم گفته می‌شود)، به صورت اسمی ۲ برابر سریعتر اجرا می‌شوند (این مورد برای اینستراکشن‌های قدیمی‌تری مانند stosb صادق نیست).

معرفی Anonymous labels :

این نشان‌گذاری در MASM معرفی شده و به صورت زیر است (تعریف برچسب بدون تعریف نامی برای آن):

@@:

در این حالت :

jmp @F یعنی پرش به جایی که اولین برچسب :@@ در ادامه کد تعریف شده است.

jmp @B یعنی پرش به جایی که اولین برچسب :@@ قبل از پرش ذکر شده وجود دارد.

مثالی دیگر در مورد کار با رشته‌ها:

فیلتر کردن یک رشته به صورتی که فقط اعداد آن استخراج شوند.

عدد اسکی معادل "0" مساوی ۴۸ و عدد اسکی معادل "9" مساوی ۵۷ است.

```
NumOnly  proc  source :DWORD,  dest:DWORD
  mov ecx, source ; put address into ecx
  mov edx, dest ; put address into edx
@@:
  mov al, [ecx] ; copy byte at address in ecx to al
  inc ecx ; increment address in ecx
  ; ------------------------------------------------
  ; perform byte modification, replacement or omissions
  ; ------------------------------------------------
  cmp al, 0 ; test for zero first
  je @F ; exit loop on ASCII zero
  cmp al, "0" ; string literal
  jb @B ; if below 48 "0", jump back
  cmp al, "9" ; string literal
  ja @B ; if above 57 "9", jump back
  ; ------------------------------------------------
  mov [edx], al ; copy byte in al to address in edx
  inc edx ; increment address in edx
  jmp @B
@@:
  mov [edx], al ; copy ASCII zero to address in EDI

  ret
NumOnly endp
```

کار با ماکروها:

از ماکروها جهت خلاصه کردن اعمال تکراری در یک کد می توان کمک گرفت. روش ایجاد یک ماکرو به شکل زیر است:

```
Do_This_Operation MACRO
; write your code here
ENDM
```

سپس به سادگی با نوشتن نام آن در کد می توان از آن استفاده کرد:

```
Do_This_Operation
```

هنگامیکه اسمبلر به نام ماکرو در کد شما می‌رسد،آنرا عینا بسط داده و در کد جایگزین می‌نماید (برخلاف یک رویه).به همین جهت حجم نهایی کد در این حالت بیشتر می‌باشد.

همچنین یک ماکرو پارامتر نیز می‌تواند بپذیرد. برای مثال:

```
RGB MACRO red, green, blue
   xor eax, eax
   mov al, blue ; blue
   rol eax, 8
   mov al, green ; green
   rol eax, 8
   mov al, red ; red
ENDM
```

روش استفاده از آن به شکل زیر است:

```
RGB 125, 175, 225
or
RGB Byte1, Byte2, Byte3
or
RGB cl, ch, dl
```

سرعت اجرای ماکروها از رویه‌ها بیشتر است زیرا سربار کار با پشته در آن حذف شده است.

پارامترهای محلی در ماکروها:

مشکل زمانی رخ می‌دهد که درون تعریف یک ماکرو ، برچسب نیز تعریف شده باشد و این ماکرو بیش از یکبار فراخوانی شود. با توجه به اینکه اسمبلر کد ماکرو را صرفا بدون هیچ تغییری در مکان فراخوانی کپی می‌نماید این امر سبب بروز خطای برچسب‌های تکراری می گردد.

خوشبختانه این مشکل در MASM با بکار گیری رهنمود LOCAL (DIRECTIVE) برای تعریف برچسب در ماکروها به صورت خودکار حل می گردد.

برای مثال :

```
MyMacro MACRO Parameter1, Parameter2 etc ...
  LOCAL MyLabel
  ; asm code
  jmp MyLabel
  ; asm code
  MyLabel:
  ; other asm code
ENDM
```

باید دقت داشت که رهنمود LOCAL در اینجا با بحث تعریف آن در رویه‌ها متفاوت است.

MASM برچسب‌هایی را که بدین شکل تعریف می‌شوند در انتها به صورت 00001?، 00002? و ... در کد قرار می‌دهد تا از تکراری نبودن آنها اطمینان حاصل شود.

کار با ساختارها:

یک ساختار ، گروهی از داده ها در کنار هم است (با اندازه ای مشخص) تا بتوان با آنها همانند یک واحد رفتار کرد.

برای مثال :

```
RECT STRUCT
  left    DWORD ?
  top     DWORD ?
  right   DWORD ?
  bottom  DWORD ?
RECT ENDS
```

علامت ? در اینجا به معنای عدم مقدار دهی اولیه عضو تعریف شده است.

اعضای این ساختار به صورت ۴ مقدار DWORD به صورت متوالی در حافظه قرار می گیرند.

در قسمت data. می توان مقدار اولیه نیز برای اعضای آن در نظر گرفت اما درون رویه ها به شکل زیر عمل می‌شود:

```
LOCAL Rct :RECT
; code
mov Rct.left,   1
mov Rct.top,    2
mov Rct.right,  3
mov Rct.bottom, 4
```

لازم به ذکر است که اعضای ساختار ، یک عملوند حافظه می‌باشند.بنابراین انتقال یک عملوند حافظه‌ی دیگر به آن به صورت مستقیم میسر نیست. در این حالت یا باید از پشته کمک گرفت و یا از رجیسترها.

برای استفاده از ساختارها در توابع ویندوز می توان از روش زیر استفاده کرد:

ابتدا می توان اعضای ساختار را با مقادیر لازم مقدار دهی کرد و سپس آدرس آنرا به تابع انتقال داد.

```
Invoke   APIcall,parameter1,parameter2,ADDR Rct
```

و اگر رویه ای نوشته اید که نیاز به دریافت مقادیر ساختار بجای آدرس آن دارد می توان به شکل زیر عمل کرد:

```
MyProc proc par1:DWORD,par2:DWORD,MyRect:RECT

mov  eax, MyRect.left ; copy first member into EAX
```

ساختارهای تو در تو:

یکی از مواردی که در تعدادی از توابع API ویندوز مرسوم است استفاده از ساختارهای تو در تو می باشد که روش تعریف آنها در MASM به شکل زیر است:

```
MyNestedStruct STRUCT
   item1 RECT <>
   item2 POINT <>
MyNestedStruct ENDS
```

این ساختار از دو ساختار RECT که پیشتر تعریف شد و ساختار زیر استفاده می کند:

```
POINT STRUCT
   x DWORD ?
   y DWORD ?
POINT ENDS
```

برای استفاده از آن:

```
LOCAL mns:MyNestedStruct

mns.item1.left
mns.item1.top
mns.item1.right
mns.item1.bottom
mns.item2.x
mns.item2.y
```

بکارگیری پیشرفته ساختارها:

در طراحی تعدادی از توابع API ویندوز مرسوم است که ساختارها بعنوان آدرس به آنها انتقال داده شوند.MASM برای سهولت انجام اینگونه امور ، راه حل ساده تری را ارائه داده است.

برای مثال اگر نیاز باشد تا آدرس ساختار RECT را به یک رویه انتقال داد ، روش متداول آن به صورت زیر است:

```
Invoke  MyFunction,ADDR Rct
```

تعریف این تابع نیز به صورت زیر است:

```
MyFunction  proc  lpRect:DWORD
```

روش آدرس دهی پارامترهای مختلف ساختار به شکل زیر است:

```
mov eax, lpRct
mov [eax],      DWORD PTR 10
mov [eax+4],    DWORD PTR 12
mov [eax+8],    DWORD PTR 14
mov [eax+12], DWORD PTR 16
```

برای ساختاری ساده همانند مثال ذکر شده این روش شاید زیاد مشکل نباشد اما با نمونه‌های پیچیده تر احتمال بروز خطا به شدت افزایش می‌یابد.

در MASM رهنمودی به نام ASSUME جهت سهولت اینگونه مقدار دهی‌ها تعریف شده است:

```
ASSUME  eax:PTR RECT
 mov  eax, lpRct
 mov  [eax].left,      10
 mov  [eax].top,       12
 mov  [eax].right,     14
 mov  [eax].bottom, 16
ASSUME  eax:nothing
```

در این حالت به اسمبلر گفته می‌شود که با رجیستر eax همانند ساختار RECT رفتار کن.

ASSUME eax:nothing به این معنا است که کار با رجیستر به این شکل خاص به پایان رسیده است.

روش دیگری نیز وجود دارد (type casting) :

```
mov   eax, lpRct
mov  (RECT PTR [eax]).left,      10
mov  (RECT PTR [eax]).top,       12
mov  (RECT PTR [eax]).right,     14
mov  (RECT PTR [eax]).bottom,  16
```

اضافه کردن فایلهای اسمبلر به ++VC :

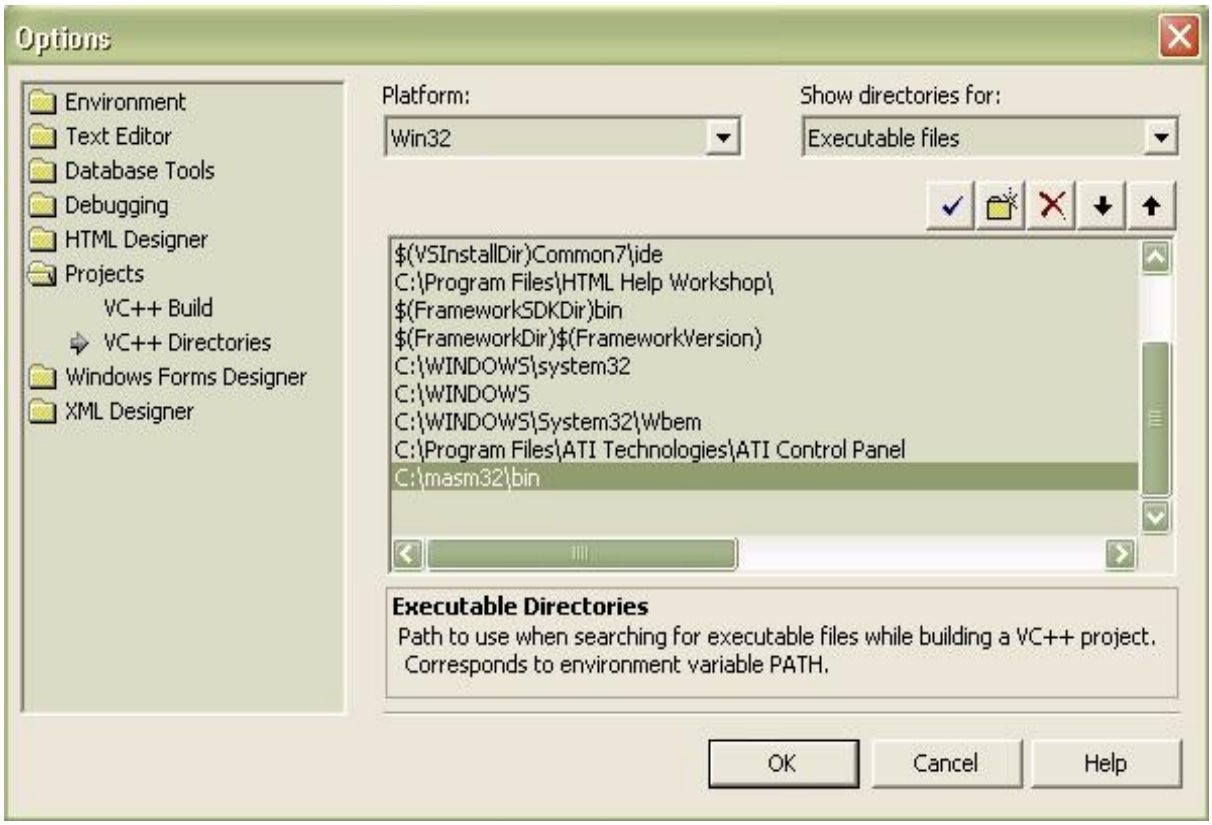
همانطور که پیشتر نیز گفته شد فایلهای obj ساخته شده در MASM با linker مربوط به ++VC سازگار هستند. برای استفاده از آنها در ++VC :

ابتدای مسیر دایرکتوری bin فولدر masm32 را مشخص نمایید (از link.exe آن می‌خواهیم استفاده کنیم). برای مثال :

C:\MASM32**bin**

سپس در IDE مربوط به ++VC به منوی Tools ، قسمت زیر مراجعه نمایید:

Tools/Options/Projects/Visual C++ Directories

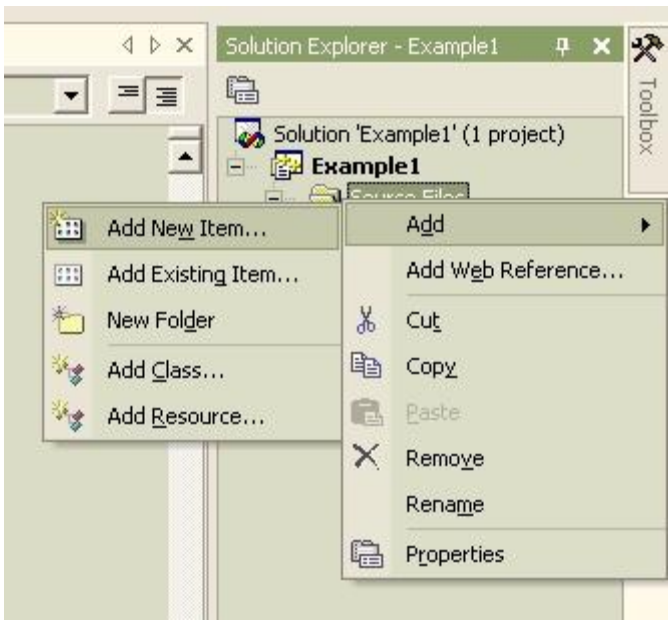


شکل ۱۴– تنظیمات IDE

سپس دایرکتوری ذکر شده را به این مسیر اضافه نمایید و آنرا به پایین لیست انتقال دهید (مهم).

در ادامه یک برنامه console ایجاد کنید.

سپس مطابق شکل زیر فایل جدیدی به نام test.asm را به پروژه اضافه کنید.



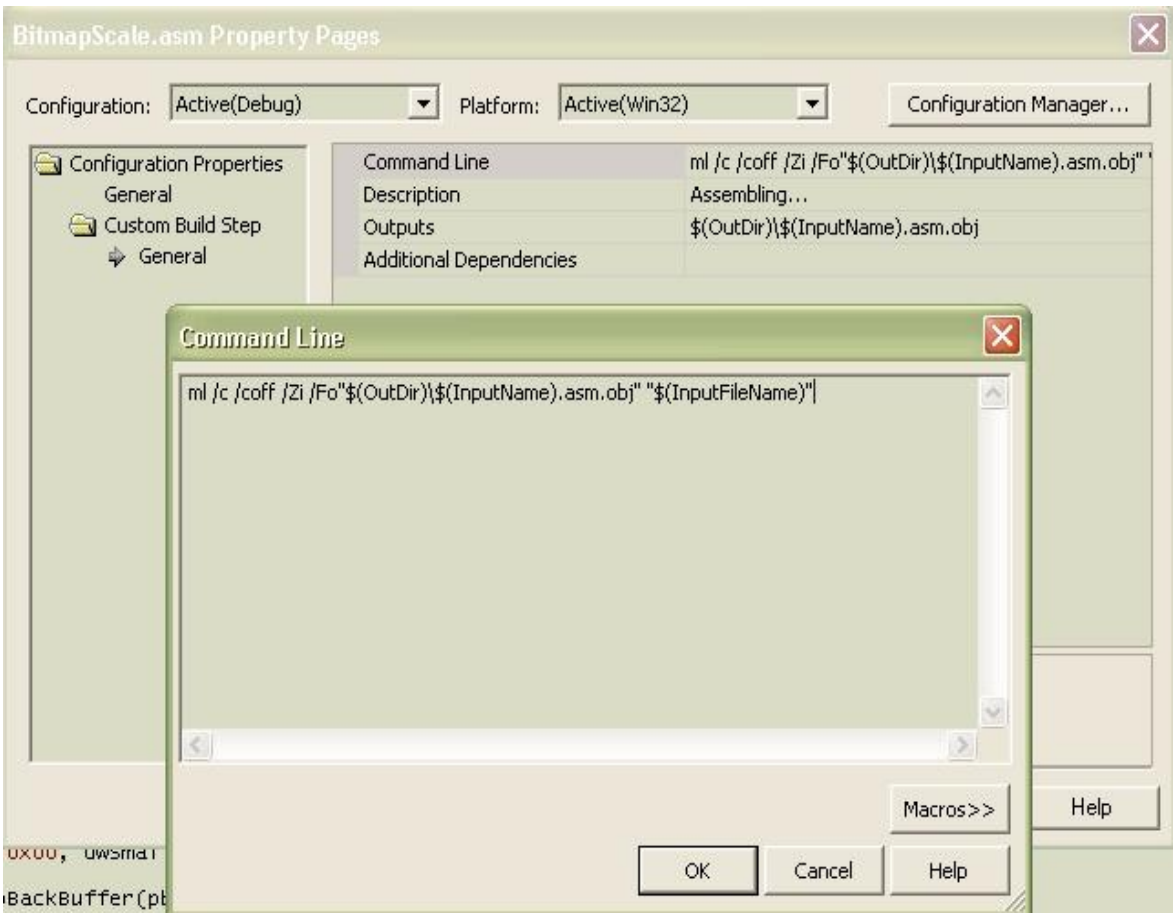
شکل ۱۵– اضافه کردن فایلی جدید به پروژه

کد ساده زیر را به فایل test.asm اضافه نمایید (در مورد ساختار استاندارد آن پیشتر صحبت گردید):

```
.486
.model flat, stdcall
option casemap :none

.code

TestProc proc dwValue:DWORD

   mov eax, dwValue
   add eax, 100
   ret

TestProc endp

end
```

این کد عدد ۱۰۰ را به ورودی اضافه کرده و سپس حاصل را بر می گرداند.

در ادامه باید خواص build این پروژه را تنظیم کرد. روی فایل کلیک راست کرده و سپس از منوی ظاهر شده گزینه خواص را انتخاب کنید (شکل زیر).



شکل ۱۶– تنظیم کردن خواص Build.

قسمت custom build step/general را انتخاب کرده و سپس خط فرمان زیر را در آن وارد کنید:

```
ml /c /coff /Zi /Fo"$(OutDir)\$(InputName).asm.obj" "$(InputFileName)"
```

همچنین در قسمت outputs آن بنویسید:

```
$(OutDir)\$(InputName).asm.obj
```

به این صورت حالت debug build آماده شد.

برای حالت release build می توانید پارامتر Zi/ را حذف نمایید.

فراخوانی کد اسمبلر توسط C++:

برای تعریف تابع نوشته شده در اسمبلر در C++ به صورت زیر عمل می کنیم:

```
extern "C" unsigned int __stdcall TestProc(unsigned int dwValue);
```

و نهایتا داریم:

```
int main(int argc, _TCHAR* argv[])
{
  unsigned int dwValue = 100;
  unsigned int dwReturn = TestProc(dwValue);

  printf("%d\n", dwReturn);
  getchar();

  return 0;
}
```

در اینجا علاوه بر اینکه خروجی صحیحی حاصل خواهد شد ، می‌توان با تنظیم کردن breakpoint بر روی تابع TestProc ، کد اسمبلی را درون VC++ IDE نیز دیباگ کرد.

درخواستی از خواننده:

اگـر در مـتن فـوق خطـایی را مـشاهده فرمودیـد (فنـی یـا غیرفنـی) ، لطفـا آنـرا بـا نویـسنده بـه آدرس vahid_nasiri@yahoo.com مطرح بفرمایید.

مآخذ:

1- Win32Asm Tutorial by Thomas Bleeker. Web: http://www.madwizard.org
2- Masm32's help files.
3- Iczelion's Win32 Assembly tutorials. Web: http://win32assembly.online.fr/tutorials.html
4- An Introduction to Assembly Language I/II/III by Darwen.
Web: http://www.codeguru.com/Cpp/Cpp/cpp_mfc/tutorials/article.php/c9411
5- Win32 Assembly parts 1-6 by Chris Hobbs.
Web: http://www.gamedev.net/reference/list.asp?categoryid=20#101
6- Win32 Assembler Coding Tutorial. Web: http://www.deinmeister.de/w32asm1e.htm