

The Assembly

(Written By Criminal2)

Assembly nedir ?

Assembly, makina dilinin(CPU'ya has özel bir alfabe) ingilizce kısaltmalara dönüştürülerek oluşturulmuş bir dildir. Bu özelliğinden ötürü tüm programlar (hangi dil ile yazılmış olusa olsun) bu dile dönüştürülebilir.(Disassembly).

Neden assembly kullanmalıyız ?

Assembly ile yazdığınız programlar sadece sizin istediğiniz kodları içerirler.Bu yüzden çok küçük programlar yazabilirsiniz. örneğin 800 byte büyüklüğünde olup pencere çıkartan programlar yazabilirsiniz.

Tom Raider : Dark Angel adlı oyunu oynadınız mı. Oyun açılmadan önce yaptığınız ayarlarda SSE,3DNow,SSE2 gibi SIMD sistemlerinin kullanılıp kullanılmayacağını ayarlayabiliyorsunuz. MMX,SSE,SSE2,3DNow gibi sistemlerle renk,matris hesaplamalarınızı çok daha hızlı yapabilirsiniz.Bu tekniğe kısaca SIMD(Single Instruction Multiple Data) denilmektedir. Assembly ile bu yöntemleri kullanabilirsiniz. C++ ile uygulamanın imkansız olduğu programlama tekniklerini assembly ile kullanabilirsiniz.

Bir program yazalım girilen fonksiyonun grafiğini çizsin. Doğal olarak önce verilen stringi paçalayan ne yapılması gerektiğini anlayan bir fonksiyon yazacaksınız. Bunu c++ ile yazdığınızda ve fonksiyonu her çağırdığınızda verilen string'i tekrar ayırtmak zorunda kalacaksınız.Veya bunun benzeri olan bir işlem yapacaksınız. Ama assembly kullanırsanız programınız verilen fonksiyonun işlemlerini yapan yeni bir program parçası oluşturabilir. Ve kat kat daha hızlı çalışan bir program elde etmiş olursunuz.

Ön Hazırlık

Assembly ile programlamaya başlamadan önce yazdığınız kodları makina diline çeviren bir "Assembler"a ihtiyacınız olacak. www.masm32.com adresinden Microsoft Assembleri indirebilirsiniz. Tamamen ücretsizdir. www.intel.com adresinden "IA-32 Manulas"i indirmeniz zorunlu değildir.(buda ücretsiz) Ancak bu 3 el kitabı size takıldığınız konularda çok faydalı olacaktır. Dikkati bakarsanız bu dökümandaki pek çok resiminde oradan alındığını görebilirsiniz ☺ Ayrıca bu kitaplar tüm assembly komutlarının açıklamalarında içermektedir. <http://home.t-online.de/home/Ollydbg> adresinden OllyDbg'yi indirmelisiniz. Bu bizim can kurtaran simidminiz işler ters gittiğinde bizim kurtacak ☺

Bu dökümanı bitirdiğinizde Windows da gui içeren uygulamalar yazabileceksiniz.

Temel Sayı Tipleri:

Bu bölümde sadece tabloların yeterli olacağını düşündüm. Bir tipin maximum değeri 2^X-1 dir. X=bit sayısı. **Bu metinlerdeki tüm hesaplama ve işlemler 16'lık**

(Hexdecimal) taban üzerinden yapılmaktadır !

Sayılar hafızda tam tersi bir şekilde yer alırlar.Mesela 45 F6 BC 32 değeri hafıza şöyle gözükür 32 BC F6 45 eğer ki siz bunu 10'luk tabana çevirmek isterseniniz $32*10^0+BC*10^1+F6*10^2+45*10^3$ işlemi ile elde edebilirsiniz.(dikkat hexdecimal 10 onluk tabanda 16'ye eşittir !!!)

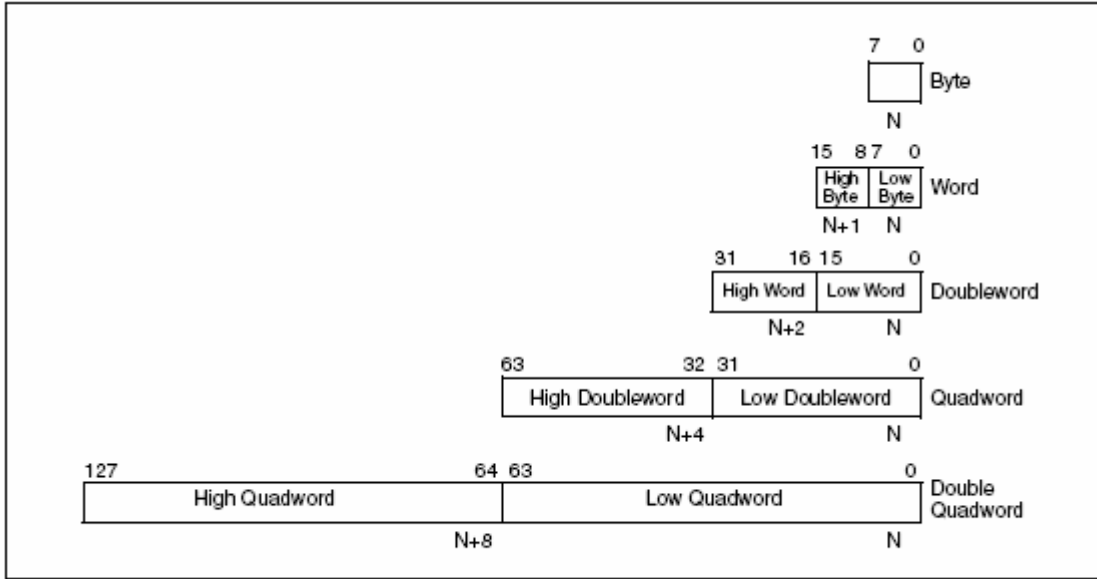


Figure 4-1. Fundamental Data Types

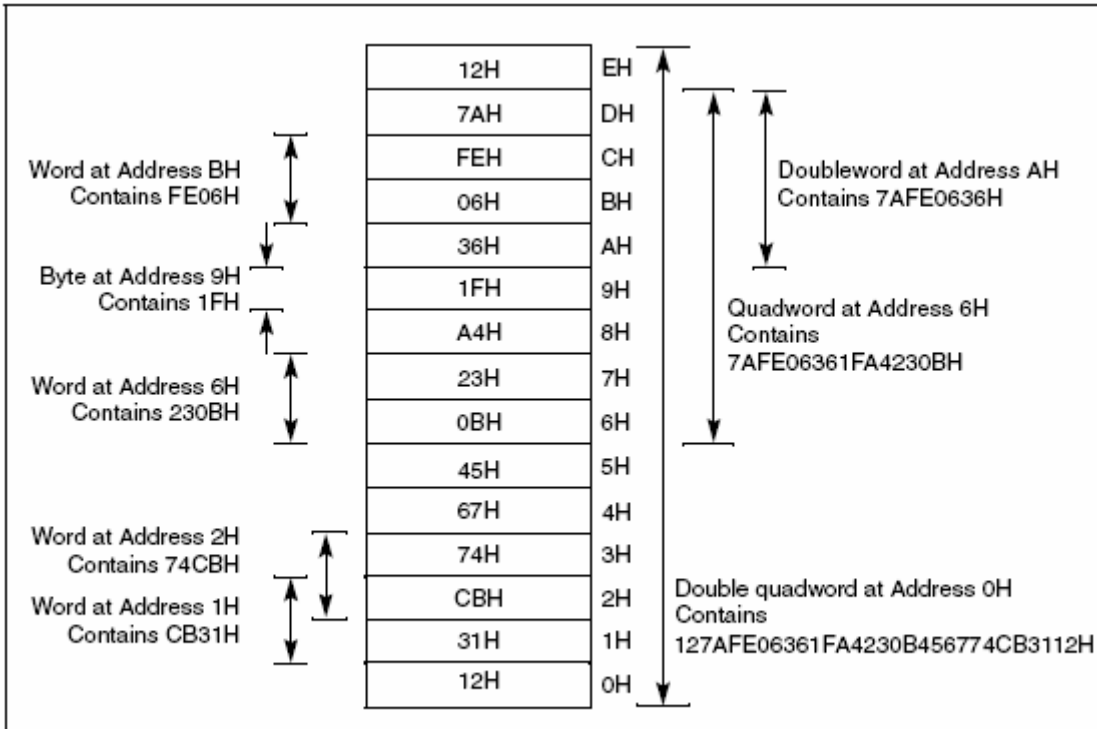


Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

Çalışma Tipleri:

CPU'un 3 çeşit çalışma tipi vardır. Bunlarda sadece 2'si bizi ilgilendiriyor.

Real Mode: Eskiden DOS işletim sisteminde kullandığı moddur. Tipik özelliği aynı anda sadece 1 programın çalıştırılabilmesidir. (Bu metni okurken winamp'ı açamazsınız ☺)

Protected Mode: Windows, Linux ve diğer pek çok modern işletim sisteminin kullandığı moddur. Modern CPU'lar (386 ve daha üst modeller) bu modda çalışmak için özel olarak dizayn edilmişlerdir. Tipik özelliği aynı anda pek çok programın çalıştırılabilmesidir.

Temel CPU Register'leri

İşlemci programların kullanması için 16 tane temel *Register* barındırır. (*Register*(kabaca)=X bitten oluşan sayıları içine alabilen "kutucuk"lar.örneğin 16-bit uzunluktaki bir register 0 ile $2^{16}-1$ (65535) arasında istenilen değeri alabilir.(Sadece TAM sayı))

- **Genel Kullanım Registerleri:** 8 tane genel kullanıma açık registerlardır.(32-bit)
- **Segment Registerleri:** 6 tane segment değerini tutan registerlardır.(16-bit)
(“Opssss, “segment”te neyin nesi ?!” Az sonra hafıza yönetiminde açıklayacağım ☺)
- **EFLAGS(Program durum ve kontrol) Registeri:** EFLAGS registeri programın durumunu ve limitlerini bildirir.
- **EIP(Instruction Pointer)(Türkçe:??):** Kısaca EIP bir sonraki komutun yerini gösterir.(32-bit)

Genel Kullanım Registerleri:

32-bit genel kullanım registerleri EAX,EBX,EDX,ECX,ESI,EDI,EBP ve ESP'dir (isimlerini şimdilik bilmenize gerek yok ne işe yaradıkları bilin yeter)

- Mantıksal ve aritmetik işlemler
- Adres hesaplamaları
- Hafıza noktalarının gösterimleri

Bu registerlar ile yapılır.

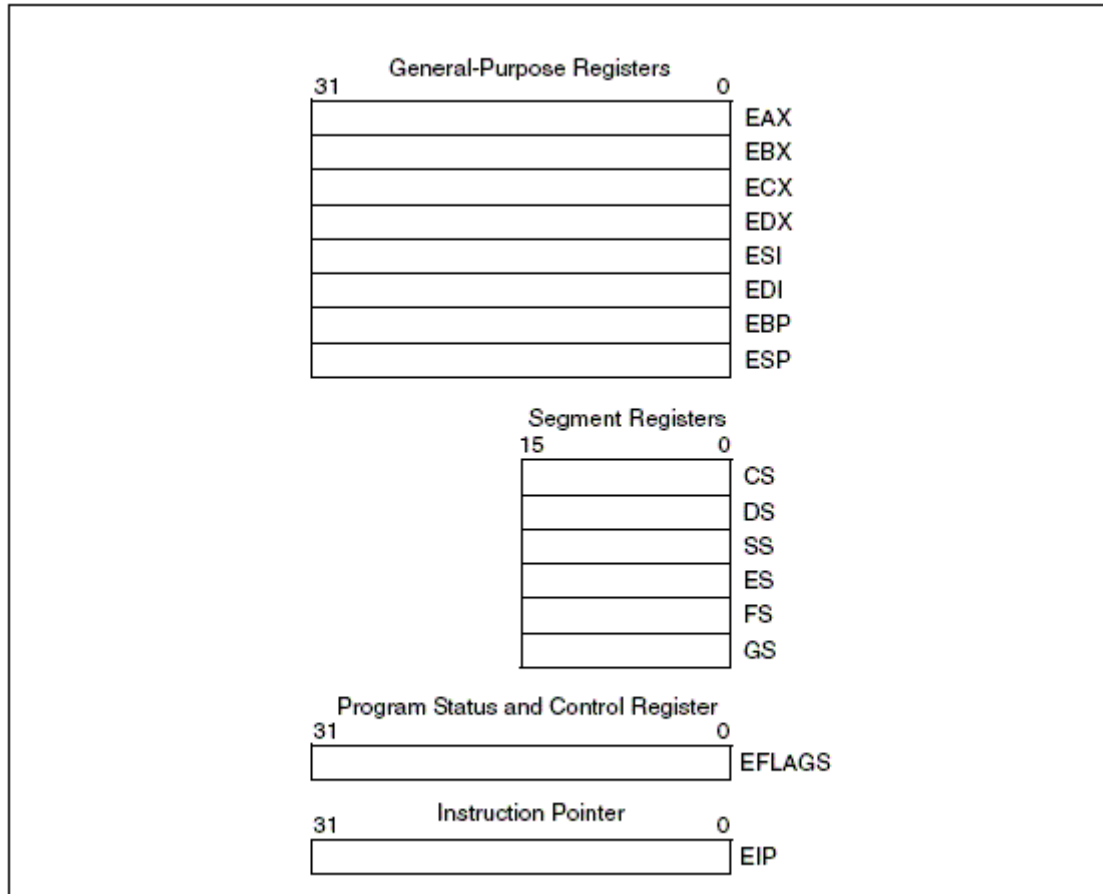


Figure 3-3. General System and Application Programming Registers

- **EAX:** Akü(?) Registeri, Kısaca tüm matematiksel işlemler,adres hesaplamaları vb.. işlemleri için kullanılır. İçerdiği değerın korunma gibi bir zorunluluğu yoktur. Her işlemde ilk önce kullanılan registerdir. Önemli: Bir fonksiyon çağırdığınızda onun

geriye döndürdüğü değer eax ile gelir.(Aslında bu sadece bir kabullenmedir. Win+Linux böyle yapınca böyle olmuş yani siz kendi işletim sisteminizde yok ebx ile döndüreceğim diyebilirsiniz...)

- **ECX(Counter):** Tüm sayma,tekrarlama vb.. işlemleri için zorunlu olarak kullanılır.
- **EDX(Data):** eax'in uzak akrabası denebilir. eğer bir işlem sırasında sonuc eax'e sığmayacak kadar büyükse(32-bit'en büyükse) taşan kısım edx'e aktarılır.
- **ESI(Source):** Genel string işlemleri için kullanılır pek çok komut ds:esi veya ds:edi ile adreslenmiş işlemler yapar
- **EDI(destination):**ESI'in aynısı
- **ESP(Stack Pointer):** Bu özel bir register daha sonra geleceğiz.

Bun registerların düşük(low, 0-16 bit'leri) kısımlarının ve onlarında düşük kısımlarının özel adları vardır. Örneğin eax'in düşük kısmı ax'dir (yani başındaki "e"yi at)

General-Purpose Registers							
31	16	15	8	7	0	16-bit	32-bit
		AH		AL		AX	EAX
		BH		BL		BX	EBX
		CH		CL		CX	ECX
		DH		DL		DX	EDX
		BP					EBP
		SI					ESI
		DI					EDI
		SP					ESP

EFLAG Register:

Şuanda bu register üzerine ne söylesem sizin için anlamsız olacaktır.Bu registerin detaylarını aktif programlaya başlayınca bol bol kullanacağız.

Protected Mode Hafıza Sistemi

Su ana kadar bu burada okuduklarınız içinde hatta assembly öğrenme sürecinizdeki kilit noktalardan biri. Eğer bir problem yaşarsanız <http://forum.ceviz.net> forumuna sorabilirsiniz. Kesinlikle atlamayın.

Assembly'daki komutların,kabullenmelerin cartların ve curtların hiçbir önemi yoktur. Önemli olan tek şey mantığı kavramaktır.

RAM'lerinizi bir cetvel gibi düşünün fakat cetvelin üzerindeki sayılar cm değil kaçınıcı byte olduğunu gösterebilir. İşte buradaki "kaçınıcı byte"a *fiziksel adres* denilmektedir. Yok bu cetvel RAM'lerde değilde bir x noktasında ise "kaçınıcı byte"a *offset* denilmektedir.

Windows(kısaca tüm multi tasking işletim sistemleri) açılırken fiziksel adresin başlangıcına yakın bir noktadan itibaren bir tablo oluşturur.Bu tablonun(Global Descriptor Table) büyüklüğünü ve yerini GDTR adlı çok özel (1,5dword büyüklünde) bir registere aktarır. Bu tablonun elemanları 8 byte büyüklüdedir ve adlarına "*Segment Descriptor*" denir. Bu küçük elemanlar içlerinde belirli bir fiziksel hafıza bölgesine ait büyüklük, tip, kimin tarafından kullanılacağı gibi bilgiler barındırırlar.
(şimdi bu bilgiyi buzdolabına koyun daha sonra kullanacağız.)

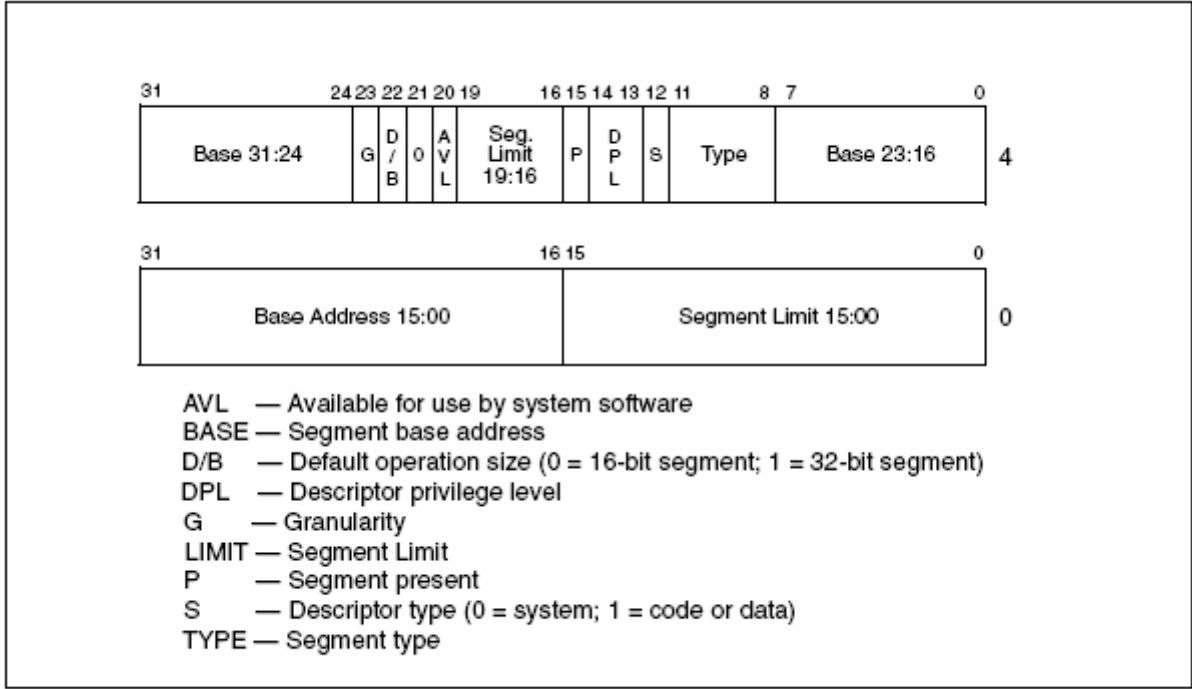


Figure 3-8. Segment Descriptor

(bu şekle bakıp “aaaa bunlar ne böyle” demeyin. Bunu sadece s.d hakkında azda olsa bir fikriniz olması için koyuyorum orada bahsedilen terimleri bilmeniz gerekmiyor....)

Bir programa çift tıkadık ve çalıştırdık. Program çalışmaya başlamadan önce programa 4GB adreslenebilir bir hafıza(Virtual Memory) atanır.(sizin pc’inizin 1mb ram+100mb hdd’si olsabile gene 4GB’dır) Bu hafıza tamamen **zahiridir !**. Fakat eğer hafızanın bir yerine yazılmak istenirse önce o nokta “var” edilir.Bir başka deyişle o nokta fiziksel hafızada bir x noktasına giden bir kapıymış gibi davranır ve o noktaya yapılan tüm işlemlerden aslında fiziksel belleğin x noktası etkilenir. Bu sebepten program 4gb hafızası olmasına rağmen işgal ettiği gerçek bellek sadece okuduğu/yazdığı kadardır.

Peki CPU gerçek fiziksel bellek adresini nasıl bulur ? işte burada devreye daha önce sözünü ettiğimiz segment registerleri giriyor. Bir segment registerinin değeri GDT’deki elemanın tablonun kaçınıcı byteda olduğunu(offsetini) gösterir.

Fakat bunu yanında bazen LDT(Local Descriptor Table) ve Paging de kullanılır.Bu durum şu sonuc ortaya çıkar:

(Segment:Offset)

10 tane program 0032:0400000 adresinde birbirlerini karşmadan farklı hafızalarla çalışabilirler.

(Protected mod’un en en önemli özelliği budur !!! Bu yapı sayesinde multi tasking yapılabilir.)

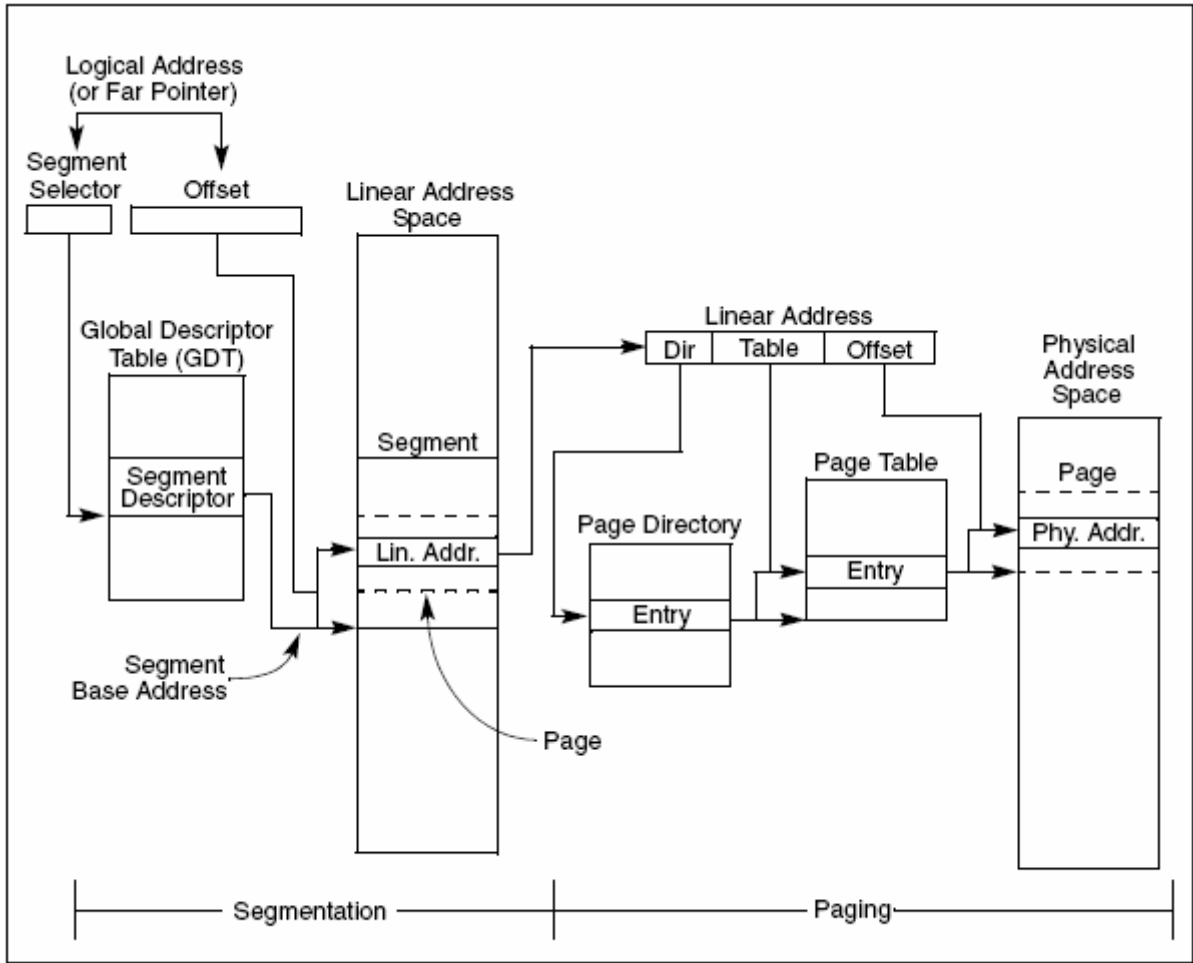


Figure 3-1. Segmentation and Paging

Segment Registerleri:

- **CS(Code Segment):**Program kodlarının bulunduğu bölümü temsil eder.
- **GS=FS=DS(Data Segment):** Programın kullandığı bilgilerin bulunduğu bölüm
- **ES(Extra Segment):** Adı üzerinde extra
- **SS(Stack Segment):** DS'in özelliklerin gösteren bir bölümdür.Hatta windosda ds=ss'dir. Stack ne işe yarar ? nedir ? gibi sorulara alıştığınız üzere ilerleyen sayfalarda cevap vereceyim ☺

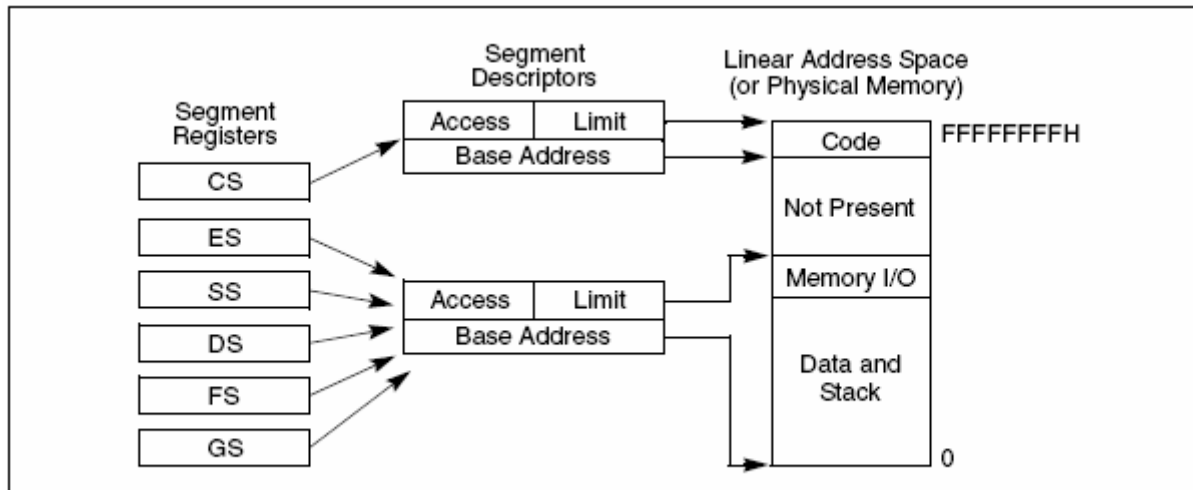


Figure 3-3. Protected Flat Model

Windows(2k|XP) kullandığı hafıza modeli:(aslında tam olarak DEĞİL yaklaşık çünkü DS=SS=ES olması gerekiyor)

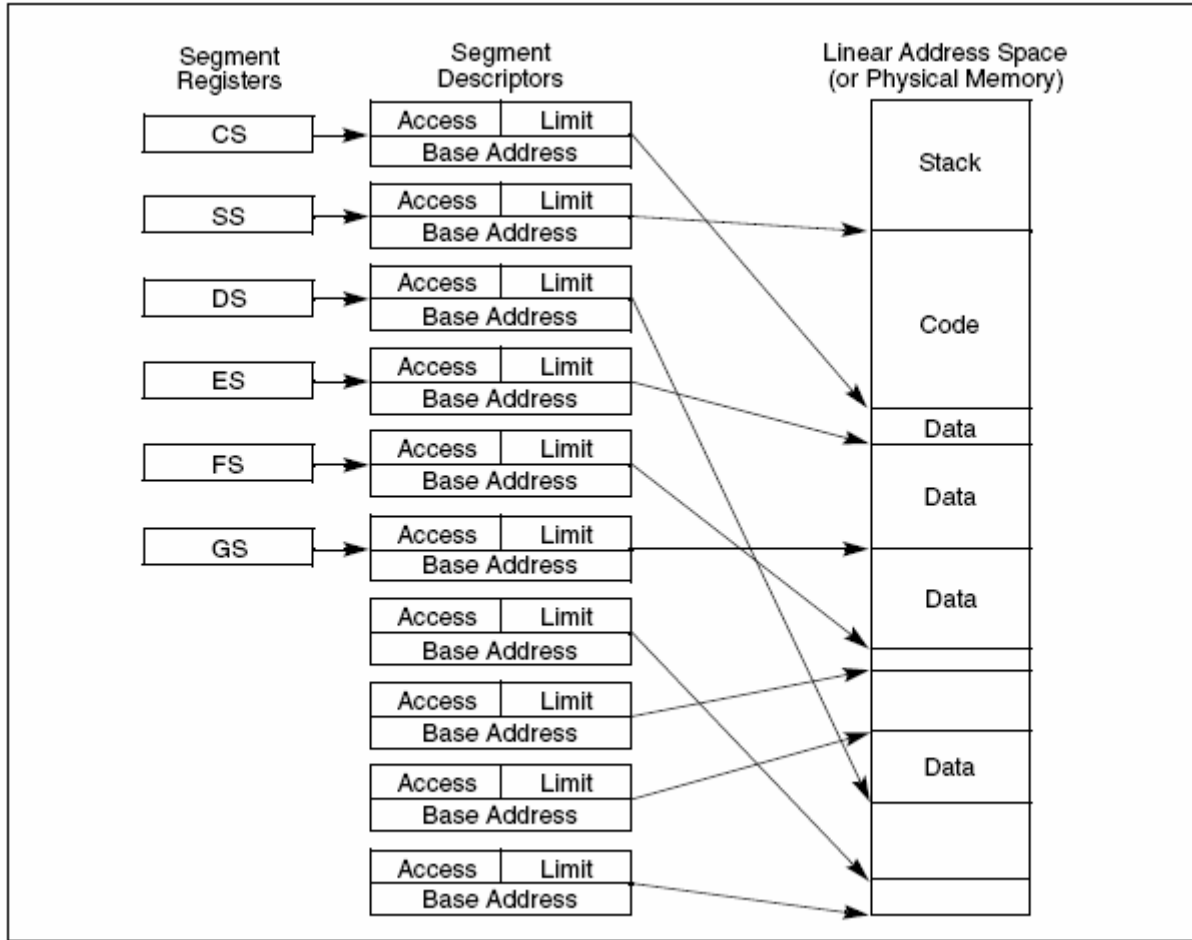


Figure 3-4. Multi-Segment Model

Sesleri duyar gibiyim 7.sayfa bitti hala tık yok ☺

Protection Rings

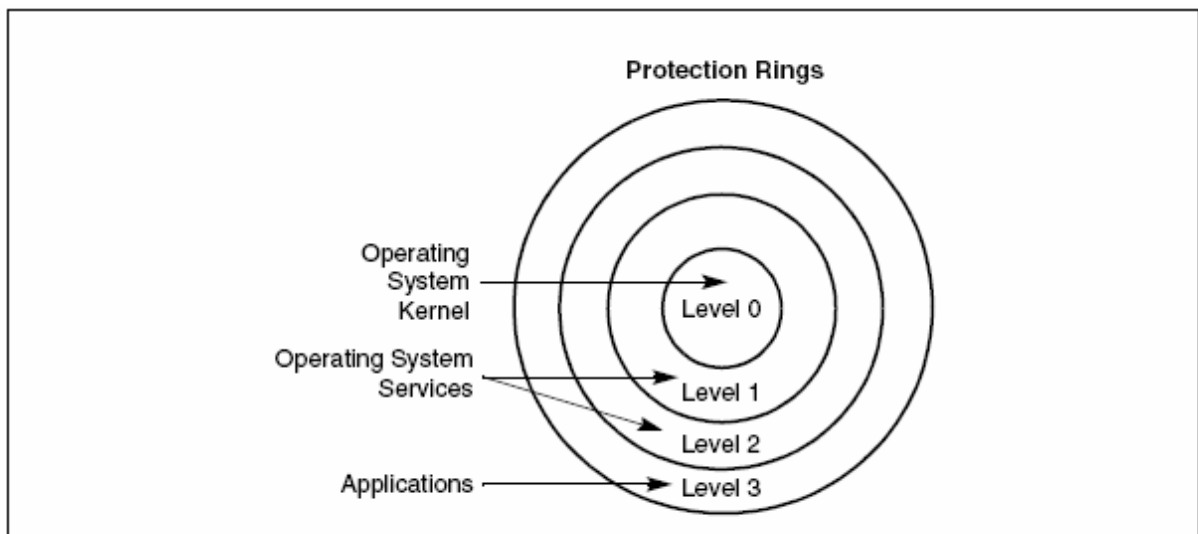


Figure 4-2. Protection Rings

Hala okuyormusunuz ?

Farz edelim 2 program aynı anda aynı dosyaya veya bellek bölgesine yazmaya çalıştı. Bu bir kaos yaratır öyle değil mi ? Bu tip problemleri engellemek ve giriş çıkış aygıtları ile belirli bir standarta iletişim kurabilmek için "Protection Levels" geliştirilmiştir.

- **Ring-0:** İşletim sisteminin çalıştığı seviyedir. Hiç bir sınırlama yoktur.
- **Ring-1,2:** Önemsiz....
- **Ring-3:** Maximun sınırlama. Bu seviyedeki programlar hiçbir giriş/çıkış aygıtına ulaşamaz (ekrana yazı bile yazdıramaz.) Dosya giriş çıkışı yapamaz. Kısacası hiçbir şey yapamaz. (Bunlar varsayılan düzeydir. İşletim sistemi özel yetkilendirme yapabilir.)

Peki tüm programlar bu seviyede ise "not defteri" nasıl çalışıyor ? işletim sistemi kendine mesaj atılmasına atılmasına izin verir. (çeşitli yöntemlerle). Mesela program işletim sisteminine "X dosyasının Y byte'ına şunu yaz" der işletim sistemi "no problem" deyip yazabilir. Veya isteği geri çevirebilir. İşte bu programın, diğer programlarla ve donanın ile uyumu bu sayede gerçekleşir.

Daha önce bahsettiğimiz "Segment Descriptor" daki DPL-Descriptor privilege level bu seviyeleri ayarlamakta kullanılır. (sadece değil !)

Temel Assembly Komutları

• MOV

"Mov" komutu 2 register veya 1 bellek bölgesi ile 1 register arasında byte, word veya dword uzunluğundaki verinin aktarımını sağlar. Önemli olan 2 değişkeninde aynı büyüklükte olmasıdır.

mov alıcı, gönderici

```
mov eax,edx ;eax=edx
mov ax,cx ;ax=cx
mov dword ptr ds:[esi],eax ;esi tarafından adreslenmiş bellek bölgesine eax'in değerini koy.
burada esi belleğin offsetidir. (c'deki pointer gibi)
mov eax,dword ptr ds:[ecx]
mov ecx,123456 ;ecx=12346
```

```
mov edx,cx ;Yanlış edx=dword,cx=word
mov si,dword ptr ds:[ebx] ;Yanlış si=word
mov al,byte ptr ds:[ecx] ;Doğru
mov word ptr [edx],word ptr [ecx] ;Yanlış, 2'side bellek bölgesi olamaz.
mov ds,123 ;Yanlış segment registerleri direkt parametre alamazlar.
```

aşağıdaki gibi olmalıdır.

```
mov ax,123
mov ds,ax
```

• XCHG

"Xchg" komutu 2 register ve bir bellek bölgesini karşılıklı değiştirilmesi için kullanılır. Mov için geçerli olan tüm kurallar bu komut içinde geçerlidir.

xchg param1,param2

```
mov eax,1234 ;eax=12345
mov edx,6789 ;edx=6789
xchg eax,edx ;artık eax=6789 edx=12345
```

• LEA

“Lea” komut verilen bir bellek bölgesinin offsetini hesaplar.Eğer programının yuzen kod parçaları içermiyorsa kullanmanız gerekmez.Masm32 “offset” macrosu ile kullandığınız değişkenlerin offsetini sabit bir değer olarak elde etmenizi sağlar.

lea alıcı,değişken

```
lea eax,dword ptr cs:[12345] ;gibi
```

• **ADD**

“Add” komutu 2 paramterisini toplayıp sonucu ilk paramtereye aktarır.(Parametreler mov komutunkilerle aynı özelliktedirler.

add alıcı,gönderici

```
add eax,1234 ;eax=eax+1234
add ecx,edx ;eax=eax+edx
add eax,dword ptr es:[esi] ;eax=eax+esi ile adrelenen değer.
add dword ptr ds:[2324],edi
```

• **SUB**

“Sub” komutu add ile aynı özellikleri taşır.Sadece bu komut toplamak yerine çıkartır.

sub alıcı,gönderici

```
sub eax,1234 ;eax=eax-1234 eğer eax<1234 ise FFFFFFFF-1234
sub edx,edx ;edx=0
```

• **MUL**

“Mul” komutu $eax * \text{parametre}$ işlemini yapar.Sonucun düşük kısmı $eax,$ ’e yüksek kısmı ile edx ’e aktarılır. Parametre mov komutunda belirtildiği gibidir.

mul çarpan

```
mov eax,1234 ;eax=1234
sub edx,edx ;edx=0
mov ecx,4 ;ecx=4
mul ecx ;eax=1234*4
```

• **DIV**

“Div” komutu $eax / \text{parametre}$ işlemini yapar $eax = \text{bölüm}$ $edx = \text{kalan}$.

div bölen

• **INC**

“Inc” komutu parmetre olarak aldığı değeri 1 artırır.

inc parametre

```
inc eax
inc dword ptr ds:[edi]
inc word ptr ss:[3443]
```

• **DEC**

“Dec” ile aynı özelliklere sahiptir.Fakan bu komut 1 artırmak yerine 1 azaltır.

dec parametre

• **CMP**

“Cmp” 2 paramtereyi kıyaslar ve eflags registerini duruma göre değiştirir.Bu komut az sonra geleceğimiz “Jump” komut kombinasyonları ile kullanılır.

cmp p1,p2

```
cmp eax,edx ;eğer eax=edx ise ZF=1
;eax!=edx ise ZF=0
;eax<edx ise CF=1
```

• **Jxx Komut seti**

“Jxx”(Jump) komutları istenen koşul oluştuğunda verilen paramtereyi eip’e aktarırlar.Bir başka deyişle programın paramterede verilen noktadan devam etmesini sağlarlar.

Jxx parametre

- **Jump:** Koşulsuz sıçrama

- **Je:** Eğer $zf=1$ ise yani cmp 'in parametreleri eşit ise
- **Jb,Jc:** Eğer $cf=1$ ise yani cmp 'de $p1 < p2$ ise
- **Ja:** Eğer cmp 'de $p1 > p2$ ise ($cf=0$ ve $zf=0$)
- **Jbe:** $p1 \leq p2$ ise ($cf=1$ veya $zf=1$)
- **Jnb:** $p1 \Rightarrow p2$ ise ($cf=0$)
- **Jnz,Jne:** $p1 < p2$ ise ($zf=0$)

`Jump eax` ;gibi

`Jnb dword ptr[edi]` ;gibi

EFLAGS: Daha önce “daha sonra açıklacağım” dediğim eflagsa sonunda geldik. Hemem bir hatılatma yapalım eflaga registeri program durumu ve kontrolünü sağlıyordu. Fakat bu register diğerlerinde farklı olarak direkt ulaşılamaz. Üzerindeki bitlerin değerini değiştirilen özel komutlar bulunur. Örneğin “cli” komutu interrupt flag’ının sıfırlar. Tüm Interruptlar devre dışı kalır. Veya “sti” komutu interrupt flag’ının değerini 1 yapar. Interruptlar kullanılabilir hale gelir. (Opssss,”Interrupt nedir ?” yakında...)

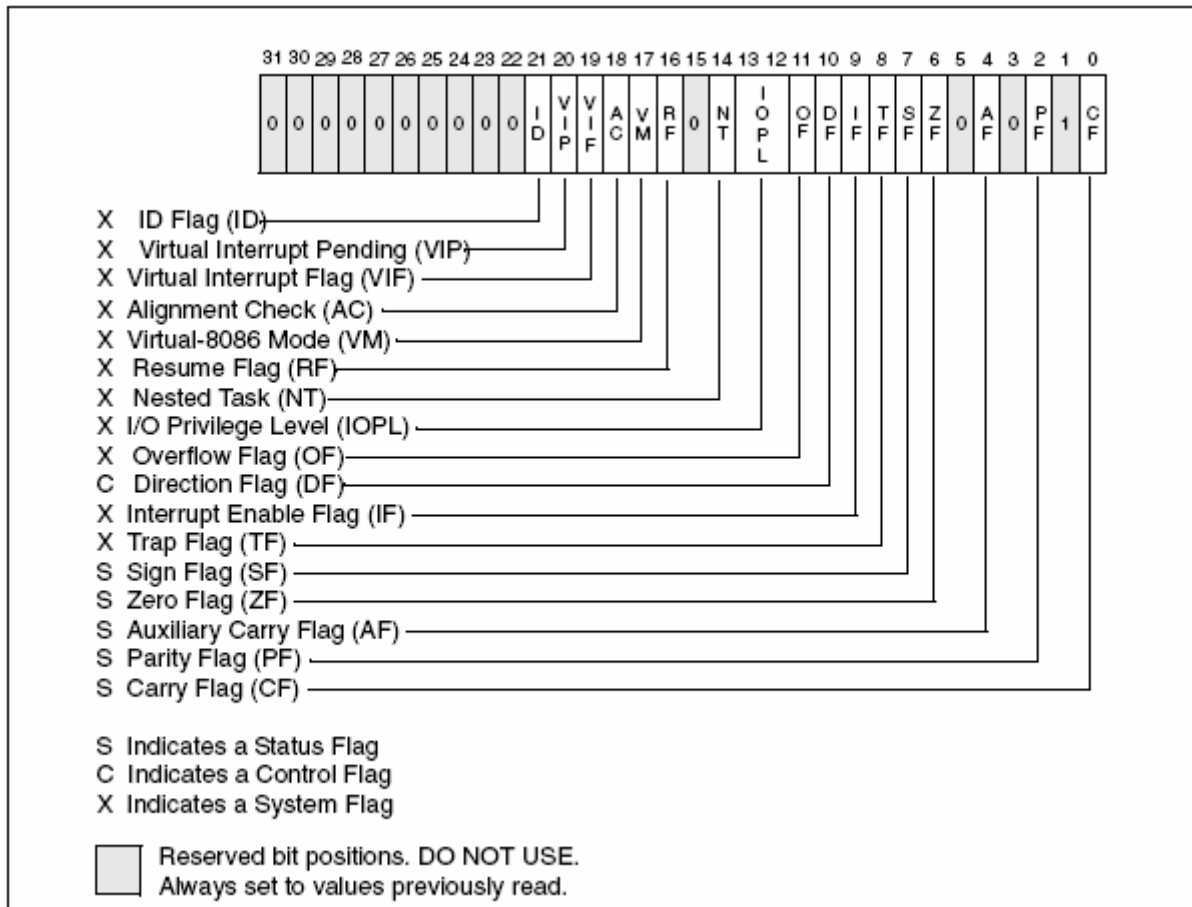


Figure 3-7. EFLAGS Register

Bu şekilde bizi ilgilendiren şimdilik sadece CF ve ZF flagları.

• **PUSH**

Ewet önemli noktlardan birine daha geldik. Sizin programınız hafızaya yüklendiğinde işletim sistemi ona küçük işlemler ve fonksiyonlara parametre aktarımı için bir miktar hazır bellek atar (Stack). Fakat bu bellek biraz gariptir ? normalde tüm işlemlerde programın akışında, hafıza işlemlerinde offsetler hep küçükten büyüğe doğru gider ama stack'ta bu tam tersidir stack'ın offseti (ki bunun esp (stack pointer) gösterir) stack doldukça küçülür. Uzun lafı kısası stack fonksiyon parametrelerini aktarımı için ayrılmış küçük bir hafızadır.

“Push” komutu parametre olarak aldığı değeri stack'a koyar esp'den değerin uzunluğunu çıkarır.

push parametre

```
push eax
push 1234
push dword ptr ds:[edi]
```

“push eax” in yaptığı iş şuna eşittir.

```
mov dword ptr ds:[esp],eax
sub esp,4
```

- **POP**

“Pop” tam olarak push’un karşıtıdır. Stack’taki bir değeri parametreye aktarır ve esp’ye onun uzunluğunu ekler.

pop parametre

```
“pop eax” şunu yapar
mov eax,dword ptr ds:[esp]
add esp,4
```

- **CALL**

“Call” komutu fonksiyonları çağırmak için kullanılır. Komut keninden soraki komutun başlangıcı stack’a koay ve hedefe sıçrar. Fonksiyon işi bittiğinde “ret” komut ile stacktan devam edilecek noktanın adresini alır. Stack pop’lanır ve sıçranır. Şimdilik canınızı sıkmayın kullandıkça problem kalmaz...

call fonksiyon adresi

“call benimf” in yaptığı iş şuna eşittir
benimf:

```
.....
....
....
pop eax
jmp eax
```

başlangıç:

```
push offset x1
jmp benimf
```

x1:

- **RET**

“Ret” komutu call ile yapılan çağrıları döndürmek için kullanılır. Ret komutun yaptığı iş şuna benzer (ama eax harcanmaz !!)

```
pop eax
jmp eax
eğer ret komutu yanında bir parametre alırsa şu hale gelir
pop eax
add esp,parametre
jmp eax
```

- **AND,XOR,OR**

bu komutların kullanım şekli aynıdır. hepside isimlerini aldıkları lojik işlemleri yaparlar. Sonuc alıcaya kaydedilir. Flagları etkilerle bazen karşılaştırma içinde kullanılırlar

komut alıcı,gönderici

```
xor eax,eax ;eax=0
or edx,edx ;eğer edx=0 ise zf=1 değilse zf=0
```

- **NOT**

“Not” komutu verilen parametredeki 1’leri 0, 0’ları 1 yapar ☺

not parametre

- **LOOP**

“Loop” komut her çalıştığında ecx’in değerini 1 azaltır. Eğer $ecx \neq 0$ ise parametresindeki yere sıçrar. özellikle terkarlarda kullanılır.

loop parametre

```
mov ecx,12
lpx:
....
.... ;bu bölüm 12 kere tekrarlanacak
loop lpx
```

tabii ki sadece loop çözüm değildir başka yöntemlerde vardır.

```
mov ecx,12
lpx:
....
....
dec ecx ;ecx'i 1 azalt
or ecx,ecx ;ecx=0 ?
jnz lpx ;değilse lpx'e git
```

- **MOVSX**

“Movsb” DS:ESI adresindeki 1 byte’ı ES:EDI adresine yazar. Herhangi bir parametre almaz. “Movsw” byte yerine word, “Movsd” ise dword aktarır

- **CMPSX**

“Cmpsb” DS:ESI adresindeki 1 byte’ı ES:EDI 1 byte ile karşılaştırır.”Cmpsw” 1 word’u,”Cmpsd” “1 dword’u karşılaştırır.

- **REPE**

ZF=0 ve $ecx > 0$ olduğu sürece parametre olarak verilen komut tekrarlanır ve ecx’in değeri 1 azaltılır.

aşağıda kod ds:esi’deki 10 byte’ı es:edi’ye kopyalar.

```
mov ecx,10
repe movsb
aşağıdaki kod ds:esi’deki ve es:edi 10 byte’ı karşılaştır eğer farklı bir byte bulursa ecx değeri 0’a ulaşmadan duru.
mov ecx,10
repe cmpsb
```

- **SHR**

“Shr” komutu 1.parametresini 2. parametresinin değeri kadar sağa kaydırır. ve boşlukları 0 ile doldurur.

shr p1,p2

```
mov eax,0ABCDEF12h
shr eax,16 ;artık eax=ABCD
```

- **SHL**

“Shl” shr’ini aynıdır sadece bu sefer sağa değil sola kaydırır.

shl p1,p2

Tabii ki assembly komutları bu kadar değil. Yukarıdakiler sadece 1/10’u Tamamını görmek için www.intel.com adresinden “IA-32 manuals”i indirin.

Interrupts

Interruptlar(kesmeler ?) bir bu kadar daha yazı kaldırabilecek bir konudur.Kabaca interruptlar jmp yada call komutuna benzer bir şekilde çalışırlar. Fakat bir interrupt çağırıldığında sistem ring değiştirebilir. “Int interrupt numarası” komut ile çağırılırlar. (Kaseti geri sarın “Protection Rings” bölümünde ring-3 programlarının işletim sisteminine mesaj iletmesinden bahsetmişim.Bu mesajlar win2k|nt(xp değil)’de int2eh ile linux’de int80h ile iletilir.) (aslında call ilede değiştirebilir fakat int.’lar daha hızlıdır. Win9x sistemleri “call gates” yardımıyla bu değişimi gerçekleştirirler.) Yukarıdak gibi bir yazılım tarafından kullanılan interruptlara “Software Interrupts” denir.Bunları isteyen kafasına göre değiştirebilir(Peki nasıl ? birazdan...)

Aslında interruptların en önemli özelliği bir donanım aygıtı tarafından da çağırılılabiliyor olmalarıdır.(“Hardware Interrupts”)(Ör: Klavyenin bir tuşuna basmamız bir interruptu çağırır.)

Bazen programcılar hatalı programlar yazarlar.

ör:

```
sub eax,eax
```

```
div eax
```

bu kod eax’i “0”a bölmeye çalışmaktadır ki böyle bir şey mümkün değildir bu durumda Int 0 işletim sistemine hatayı haber verir.(ICQ misali).

Tarihten Kısa Kısa(geyik): Yukarıda dikkat ederseniz WinXP’in mesajlaşma sistemi ile ilgi birşey söylemedim. Intel Ve Amd, PentiumII(veya dengi) ve üzeri işlemcilerde yeni bir komut çıkartılar adında “SYSENTER” ve “SYSEXIT” dediler. Bu komutlar int’lerden çok daha hızlı bir mesajlaşma sistemi sağlıyordu. Fakat sysexit’in çok iyi bir özelliği vardı.Bu programa dönüş komuydu.cs’ye 16 ekliyor, ss’ye de cs+24 değerini veriyordu. ve bunlar **DEĞİŞTİLEMİYORDU**. Doğal olarak linux bu hizmetten yararlanamadı. Fakat bak sen şu tesadüfeki win2k ve winNt sistemlerindeki int2eh tıpa tıp bu işlemi yapıyordu. WinXP’de mesajlaşma sistemi için sysenter ve sysexit kullanmaya başladı. Linux’de avucu yaladı ☺

HELLO, WORLD !

Eğer asm ile uğraşmayan biriyseniz bura kadarki kısmı tek hamlede okumuşsanız küçük çaplı “Page Fault”lar yaşayabilirsiniz. Hem dikkatinizi tekrar odaklamanız hemde artık “programlama” kısmına geçmemizden dolayı. Yukarıdaki “devasa” başlığı attım ☺

Masm32’yi kurduğunuzu tahmin ediyorum.

Win9x Kullanıyorsanız:

C:\Autoexec.bat dosyasını not defteri ile açın içine şu satırı ekleyim kaydedin

```
SET PATH=%PATH%;C:\MASM32\BIN;
```

Daha sonra sisteminizi yeniden başlatın.

WinNt|2k|Xp kullanıyorsanız:

Bilgisayarım → Özellikler → Gelişmiş → Ortam Değişkenleri

“Path” değişkenini seçip düzenleyin sonuna şunu ekleyin

```
;C:\MASM32\BIN;
```

Şimdi ilk Windows programımızı yazmaya hazırız.....

Önce sevdiğin bir yerde bir klasör açın adını “Test” yapın.

Aşağıdaki dosyaları not defteri yardımıyla yaratın.(Sadece c&p yapın bazı yerlerde satır atlamadığı halde world öyle gösteriyor.

build.bat

```
@echo off
set file=test
ml /c /coff /nologo %file%.asm
link /SUBSYSTEM:WINDOWS /MERGE:.idata=.text /MERGE:.data=.text
/MERGE:.rdata=.text /SECTION:.text,EWR /IGNORE:4078 /out:%file%.exe %file%.obj
del *.obj > NUL
echo.
@pause
```

test.inc

```
;noktalı virgül açıklama satırı anlamına gelir. Aşağıdaki dosyalar windowsun kullandığı
;sabitleri ve kernel32.dll,user32.dll'ın fonksiyonlarının adlarını içeriyorlar.
; "include" komutu ile onları projemize dahil ediyoruz.
;Win9x'de user32.dll ve kernel32.dll windows kernel'ini oluşturur ve ring-0'dır !!
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
```

test.asm

```
.586p                ;kullanacağımız cpu
.model flat,stdcall
option casemap:none
include test.inc     ;test.inc'i ekle
.data
    szMsg1          db    "Evet/Hayır",0
    szMsg2          db    "Evet",0
    szMsg3          db    "Hayır",0
    szTitle         db    "Test",0
.code
start:
    push MB_YESNO   ;MB_YESNO windowsun belirlediği sabit bir sayı değeri
    push offset szTitle
    push offset szMsg1
    push 0
    call MessageBox
    push MB_OK
    push offset szTitle
    cmp eax, IDYES  ;IDYES windowsun belirlediği sabit bir sayı değeri
    je EvetX
    push offset szMsg3
    jmp msgx
EvetX:
    push offset szMsg2
msgx:
    push 0
    call MessageBox
```

```
    push 0
    call ExitProcess
end start
```

Şimdi build.bat dosyasını çalıştırın İŞTE Test.exe hazır ilk programınız. Size evet yada hayır sorusunu soracak. Programın açıklamaları 2. yazımda olacak.

Ama bu biraz hammalık değil mi ?

Bırakın gereksiz işlerle masm32 uğraşsın.

Şimdi test.asm şöyle değiştirin. masm macro ları gerisini halleder.

test.asm

```
.586p                ;kullancağımız cpu
.model flat,stdcall
option casemap:none
include test.inc     ;test.inc'i ekle
.data
    szMsg1          db    "Evet/Hayır",0
    szMsg2          db    "Evet",0
    szMsg3          db    "Hayır",0
    szTitle db      "Test",0
.code
start:
    invoke MessageBox,0,offset szMsg1,offset szTitle,MB_YESNO
    .if eax==IDYES
        invoke MessageBox,0,offset szMsg2,offset szTitle,MB_OK
    .else
        invoke MessageBox,0,offset szMsg3,offset szTitle,MB_OK
    .endif
    invoke ExitProcess,0
end start
```

Çooooook daha kolay değil mi ????

Bir sonraki yazı neleri içerecek ?

- Bu program nasıl çalıştı ?
- Pencereler
- Exe mimarisi